

# Dispense di Reti di calcolatori

Leonardo De Faveri

A.A. 2021/2022

---

# *Indice*

---

<b>1 Introduzione</b>	<b>5</b>
1.1 Struttura di internet . . . . .	5
1.1.1 Nucleo della rete . . . . .	6
1.1.2 Periferia della rete . . . . .	6
1.2 Mezzi trasmissivi . . . . .	6
1.2.1 Doppini intrecciati . . . . .	7
1.3 Tecniche di commutazione . . . . .	8
1.3.1 Commutazione di circuito . . . . .	8
1.3.2 Commutazione a pacchetto . . . . .	9
1.4 Protocolli di comunicazione . . . . .	9
<b>2 Livello Applicativo</b>	<b>13</b>
2.1 Applicazioni di rete . . . . .	13
2.1.1 Architetture di applicazioni di rete . . . . .	13
2.2 Comunicazione in rete tra processi . . . . .	16
2.2.1 Socket . . . . .	17
2.3 Protocolli del livello Applicativo . . . . .	18
2.3.1 Protocollo HTTP . . . . .	18
2.3.2 Protocollo FTP . . . . .	24
2.3.3 Protocolli per la posta elettronica . . . . .	25
2.3.4 Protocollo DNS . . . . .	27
<b>3 Livello Trasporto</b>	<b>31</b>
3.1 Caratteristiche dei servizi offerti . . . . .	31
3.1.1 Multiplexing e demultiplexing . . . . .	31
3.1.2 Numeri di porta . . . . .	32
3.2 Protocollo UDP . . . . .	33
3.2.1 Controllo degli errori . . . . .	34
3.3 Trasferimento dati affidabile . . . . .	34
3.3.1 Protocollo Stop-and-Wait . . . . .	34
3.3.2 Finestre di trasmissione e acknowledgement . . . . .	36
3.3.3 Protocollo Go-back-N . . . . .	37
3.3.4 Protocollo Selective repeat . . . . .	38
3.4 Protocollo TCP . . . . .	40
3.4.1 Instaurazione di una connessione TCP . . . . .	42
3.4.2 Dimensione dei segmenti . . . . .	42
3.4.3 Chiusura di una connessione TCP . . . . .	43
3.4.4 Stimare l'RTT e scegliere l'RTO . . . . .	44

3.4.5	Controllo di flusso . . . . .	45
3.5	Gestione della congestione . . . . .	46
3.5.1	Modelli per sistemi a coda . . . . .	46
3.5.2	Cause della congestione . . . . .	47
3.6	Controllo della congestione in TCP . . . . .	49
3.6.1	Protocollo AIMD . . . . .	49
3.7	Versioni moderne del TCP . . . . .	55
3.7.1	Protocollo CUBIC . . . . .	55
3.7.2	Protocollo BBR . . . . .	56
3.7.3	Protocollo QUIC . . . . .	57
<b>4</b>	<b>Livello Rete</b> . . . . .	<b>58</b>
4.1	Funzioni del livello Rete . . . . .	58
4.2	Struttura e funzionamento di un router . . . . .	59
4.2.1	Porte di ingresso . . . . .	59
4.2.2	Sistemi di commutazione . . . . .	60
4.2.3	Conseguenze dei ritardi di commutazione . . . . .	61
4.2.4	Porte di uscita . . . . .	62
4.3	Protocollo IP . . . . .	62
4.3.1	Struttura dei pacchetti . . . . .	62
4.3.2	Frammentazione dei pacchetti . . . . .	64
4.3.3	Indirizzamento . . . . .	64
4.3.4	Indirizzi privati e NAT . . . . .	71
4.3.5	Indirizzi speciali . . . . .	73
4.4	Protocollo ARP . . . . .	74
4.4.1	Cenni sul livello 2 . . . . .	74
4.4.2	Principi dell'ARP . . . . .	75
4.4.3	Messaggi ARP . . . . .	75
4.4.4	ARP caching . . . . .	76
4.4.5	Proxy ARP . . . . .	76
4.5	Protocollo ICMP . . . . .	76
4.5.1	Messaggi ICMP . . . . .	76
4.5.2	Ping tramite ICMP . . . . .	77
4.5.3	Traceroute tramite ICMP . . . . .	77
4.6	Protocollo DHCP . . . . .	77
4.6.1	Princípio di funzionamento . . . . .	77
4.6.2	Gestione dei prestiti . . . . .	78
4.6.3	Messaggi DHCP . . . . .	79
4.6.4	Reti senza DHCP . . . . .	80
4.7	Il viaggio di un pacchetto attraverso la rete . . . . .	80
4.8	Protocollo IPv6 . . . . .	80
4.8.1	Struttura dei pacchetti IPv6 . . . . .	81
4.8.2	Tunneling tramite IPv4 . . . . .	81
4.8.3	Struttura e tipologia di indirizzi IPv6 . . . . .	82
<b>5</b>	<b>Metodi e protocolli di instradamento</b> . . . . .	<b>84</b>
5.1	Principi generali . . . . .	84
5.1.1	Rappresentazione delle reti come grafi non orientati . . . . .	84
5.1.2	Tipi di algoritmi di routing . . . . .	85
5.2	Algoritmo di Dijkstra . . . . .	85

5.2.1	Complessità e problemi dell'algoritmo . . . . .	87
5.3	Protocollo OSPF . . . . .	88
5.3.1	Autonomous system . . . . .	88
5.3.2	Funzionamento del protocollo . . . . .	88
5.3.3	OSPF gerarchico . . . . .	88
5.3.4	Traffic engineering . . . . .	89
5.4	Algoritmo di Bellman-Ford . . . . .	90
5.4.1	Logica di funzionamento . . . . .	90
5.4.2	Problema del count-to-infinity . . . . .	92
5.5	Protocollo RIP . . . . .	95
5.6	Confronto tra algoritmi link state e distance vector . . . . .	97
5.7	Protocollo BGP . . . . .	97
5.7.1	Principi di funzionamento . . . . .	97
5.7.2	Messaggi BGP . . . . .	99
5.7.3	Gestione delle rotte . . . . .	101
<b>6</b>	<b>Livello Data Link</b> . . . . .	<b>103</b>
6.1	Introduzione . . . . .	103
6.1.1	Servizi del livello Data Link . . . . .	103
6.1.2	Implementazione dei servizi Data Link . . . . .	104
6.2	Rilevamento e correzione di errori . . . . .	104
6.2.1	Bit di parità . . . . .	105
6.2.2	Ridondanza e interleaving . . . . .	105
6.2.3	Cyclic Redundancy Check . . . . .	106
6.3	Gestione degli accessi multipli ai canali . . . . .	107
6.3.1	Protocolli a ripartizione di risorse . . . . .	107
6.3.2	Protocolli ad accesso casuale . . . . .	108
6.3.3	Protocolli a turni intelligenti . . . . .	112
6.4	Protocollo Ethernet . . . . .	113
6.4.1	Organizzazione delle reti Ethernet . . . . .	113
6.4.2	Struttura dei frame Ethernet . . . . .	114
6.4.3	Caratteristiche del protocollo Ethernet . . . . .	115
6.5	Ethernet switching . . . . .	115
6.5.1	Domini di collisione . . . . .	116
6.5.2	Domini di broadcast . . . . .	116
6.5.3	Funzionamento degli switch . . . . .	117
6.5.4	Protocollo STP . . . . .	118
6.5.5	Organizzazione logica delle LAN . . . . .	119
6.6	Organizzazione e funzionamento delle reti wireless . . . . .	120
6.6.1	Elementi di una rete wireless . . . . .	120
6.6.2	Architettura di riferimento . . . . .	121
6.6.3	Protocolli per le comunicazioni wireless . . . . .	122
6.6.4	Collision avoidance nelle reti wireless . . . . .	124
6.6.5	Problema del terminale nascosto . . . . .	125
<b>A</b>	<b>Simboli e formule</b> . . . . .	<b>127</b>
A.1	Simboli . . . . .	127
A.2	Formule . . . . .	127
A.2.1	Ritardo di trasmissione . . . . .	127
A.2.2	Ritardo di propagazione . . . . .	127

A.2.3	BDP	127
A.2.4	Lunghezza in metri di un bit	128
<b>Glossario</b>		<b>129</b>
<b>Protocolli</b>		<b>131</b>

# *Capitolo Nr.1*

---

## *Introduzione*

---

Nel corso di questa trattazione andremo ad affrontare i temi fondamentali nello studio delle reti di calcolatori.

### **1.1 Struttura di internet**

Parlando di reti, la prima cosa che viene in mente è certamente internet, la rete per antonomasia. La rete internet, in realtà, non è una singola rete, ma è definita come la *rete di tutte le reti*.

Questo può far venire il sospetto che le reti di calcolatori siano strutture in qualche modo. Infatti è così! Internet è organizzato in una struttura gerarchica, al cui vertice, o nucleo, troviamo gli *ISP di livello 1*, al livello inferiore gli *ISP di livello 2*, poi gli *ISP di livello 3* e per finire, al livello più basso, si collocano gli *Host*, ovvero i *dispositivi terminali* degli utenti.

Ma cosa sono gli *ISP*? Un *ISP* è un *Internet Service Provider*, cioè una società o un ente che mette a disposizione l'infrastruttura necessaria a comunicare con altri utenti della rete. Ma vediamone meglio le differenze:

1. *ISP di livello 1*: (Telecom, AT&T, ...) sono pochi enti distribuiti nel mondo, ma che hanno una copertura nazionale/internazionale. Sono fittamente interconnessi e comunicano come pari;
2. *ISP di livello 2*: (Vodafone, Tim, ...) sono enti che si appoggiano all'infrastruttura di uno o più *ISP di livello 1* e hanno una copertura distrettuale/nazionale. Possono comunicare soltanto con gli *ISP* dei quali sfruttano l'infrastruttura;
3. *ISP di livello 3*: sono le cosiddette *reti di ultimo salto* o *reti di accesso* e permettono agli utenti finali di accedere alla rete e quindi hanno una copertura locale;

**Internet Exchange Point** Gli *ISP di livello 2* possono comunicare direttamente tra loro, senza scalare la gerarchia, attraverso punti particolari della rete detti *IXP*, *Internet Exchange Point*. Questi sono luoghi in cui le infrastrutture di due o più *ISP di livello 2* sono messe in comunicazione, permettendo così al traffico dati di passare da un'infrastruttura all'altra. Questo tipo di interscambi consentono, come vedremo, di ridurre il tempo necessario per il trasferimento dei dati.

**Reti single-homed e multi-homed** Gli *ISP di livello 3* possono appoggiarsi a uno o più *ISP di livello di 2*. Nel primo caso configurano una *single-homed network*, mentre nel secondo una *multi-homed network*.

### 1.1.1 Nucleo della rete

La struttura gerarchica appena descritta permette di distinguere due livelli della rete: un *nucleo* e una *periferia*.

Il *nucleo* corrisponde alla parte di rete gestita dagli *ISP di livello 1 e 2*, cioè la parte di infrastruttura non direttamente accessibile dagli utenti. È evidente che questa parte della rete è quella in cui circola la maggior parte del traffico, per cui rallentamenti, congestioni e guasti hanno conseguenze più gravi. Per questo motivo, i dispositivi fisici che compongono l'infrastruttura, oltre che essere più performanti delle loro controparti di *periferia*, sono collegati tramite strutture magliate. Ciò significa che tra due dispositivi esistono una pluralità di collegamenti e percorsi possibili. Questa ridondanza permette di ridurre il rischio di interruzioni del servizio, ma anche di distribuire il traffico in maniera più efficiente, evitando che si concentri su un unico punto.

### 1.1.2 Periferia della rete

La parte *periferica* di internet corrisponde alla somma delle *reti di accesso*, cioè alla parte di competenza degli *ISP di livello 3*. Questa parte della rete funge da interfaccia di collegamento per gli utenti e i dispositivi terminali, cioè inoltra verso il *nucleo* i dati da trasmettere e riceve i dati in arrivo per poi occuparsi di farli arrivare all'*host* o al gruppo di *host* destinatari.

Esistono diversi tipi di *reti d'accesso* che si distinguono per la dimensione e le tecnologie utilizzate. Dividendole per dimensione si hanno:

- *Reti d'accesso residenziali*: coprono l'area di un'abitazione;
- *Reti d'accesso aziendali*: coprono l'area di più edifici, o di un campus;

Suddividendo invece per tecnologia si hanno:

- *Reti d'accesso punto-punto*: i dispositivi sono connessi direttamente, o attraverso uno *Switch* al *Router*. È possibile un'ulteriore suddivisione:
  - *Reti con modem dial-up*: è una tecnologia vecchia e lenta che sfrutta i collegamenti telefonici in maniera esclusiva, cioè non permette di telefonare e accedere alla rete in contemporanea;
  - *Reti DSL*: sfrutta anch'essa i collegamenti telefonici, ma non in modo esclusivo;
  - *Reti FTTH*: il *router* locale è connesso all'*ISP di livello 2* tramite collegamenti in *fibra ottica* molto veloci;
- *Reti d'accesso wireless*: usano tecnologie wireless e tanti utenti si connettono allo stesso punto, detto *base station* o *access point*, il quale è connesso a un *router*;

## 1.2 Mezzi trasmissivi

All'interno delle reti, le informazioni, codificate in bit, viaggiano su dei *mezzi trasmissivi*. Ne esistono di tipologie diverse:

- *Mezzi guidati*: i segnali si propagano su un mezzo *fisico*
  - *Doppini intrecciati*: è un cavo costituito da due fili di rame intrecciati;
  - *Cavi coassiali*: è un cavo costituito da un'anima centrale di rame e una maglia costituita da cavi metallici intrecciati. L'anima e la maglia sono separati da uno strato isolante;

- *Fibra ottica*: l'informazione viaggia sotto forma di impulsi luminosi e questa è la soluzione ottimale per le lunghe distanze;
- *Mezzi a onda libera*: i segnali si propagano attraverso l'atmosfera e lo spazio esterno sfruttando i campi elettromagnetici
  - *Reti a microonde terrestri*;
  - *Reti Wi-Fi*: le più usate nelle *reti locali*;
  - *Reti cellulari*: le più usate per la mobilità;
  - *Reti satellitari*: sono ottimali per le lunghe distanze e per luoghi che non hanno accesso a *reti cellulari*;

I *mezzi a onda libera* esaltano la mobilità, in quanto non è richiesto un collegamento fisico con il punto d'accesso alla rete, ma sono maggiormente soggetti a problemi di interferenze e riflessioni dei segnali. Per i *mezzi guidati* vale il contrario.

### 1.2.1 Doppini intrecciati

Esaminiamo più a fondo i *doppini intrecciati* che sono quelli tipicamente usati nello standard *Ethernet*. Un cavo di questo tipo è formato da 4 coppie di fili di rame intrecciati detti *doppini*.

Per evitare problemi di interferenza tra i cavi o i singoli *doppini* è possibile usare delle schermature. Diverse configurazione delle schermature sono codificate con un sistema di lettere:

$X/YTP$

dove  $X$  indica il tipo di schermatura del cavo e  $Y$  la schermatura dei singoli doppini.

Schermatura del cavo:

- $U$ : unshilded;
- $F$ : foiled (rivestito con una lamina di alluminio);
- $S$ : rivestito con una maglia metallica (rame placcato in alluminio);
- $SF$ : entrambe;

Schermatura dei *doppini*:

- $U$ : unshilded;
- $F$ : shielded;

**Cavi patch VS cavi cross** Nello standard *Ethernet*, nelle prese *RJ45*, i 4 *doppini* sono collegati ai pin della presa mettendoli in riga affiancati. Nei *cavi patch* l'ordine dei *doppini* è uguale a entrambi i capi del cavo, mentre nei cavi *cross* i due *doppini* esterni sono scambiati di posizione e allo stesso modo sono scambiati anche i *doppini* interni.

## 1.3 Tecniche di commutazione

Abbiamo detto come i bit di dati viaggino da un dispositivo a un altro, ma non abbiamo ancora chiarito come siano organizzati questi trasferimenti. Ciò è definito dalla *tecnica di commutazione* utilizzata. Ne esistono due:

- *Commutazione di circuito*: prima del trasferimento viene stabilito a priori il percorso che il messaggio dovrà seguire;
- *Commutazione a pacchetto*: il messaggio viene scomposto in pacchetti che attraversano la rete in modo indipendente gli uni dagli altri;

### 1.3.1 Commutazione di circuito

Come accennato, questa *tecnica di commutazione* prevede che il messaggio venga trasmesso in blocco e che le risorse necessarie a collegare mittente e destinatario siano riservate per quella comunicazione.

Ad essere suddivisa in porzioni è l'*ampiezza di banda* (il *bandwidth*) secondo 3 parametri:

1. *Bit rate*: e.g. in una rete a  $10 \frac{Mb}{s}$  vengono creati due canali da  $5 \frac{Mb}{s}$  che vengono assegnati a due utenti senza un limite di tempo;
2. *Frequenza*: (*FDM*) la frequenza viene suddivisa in più canali che vengono assegnati agli utenti senza un limite di tempo;
3. *Tempo*: (*TDM*) ogni utente ha a disposizione per un tempo limitato tutta la frequenza del canale;

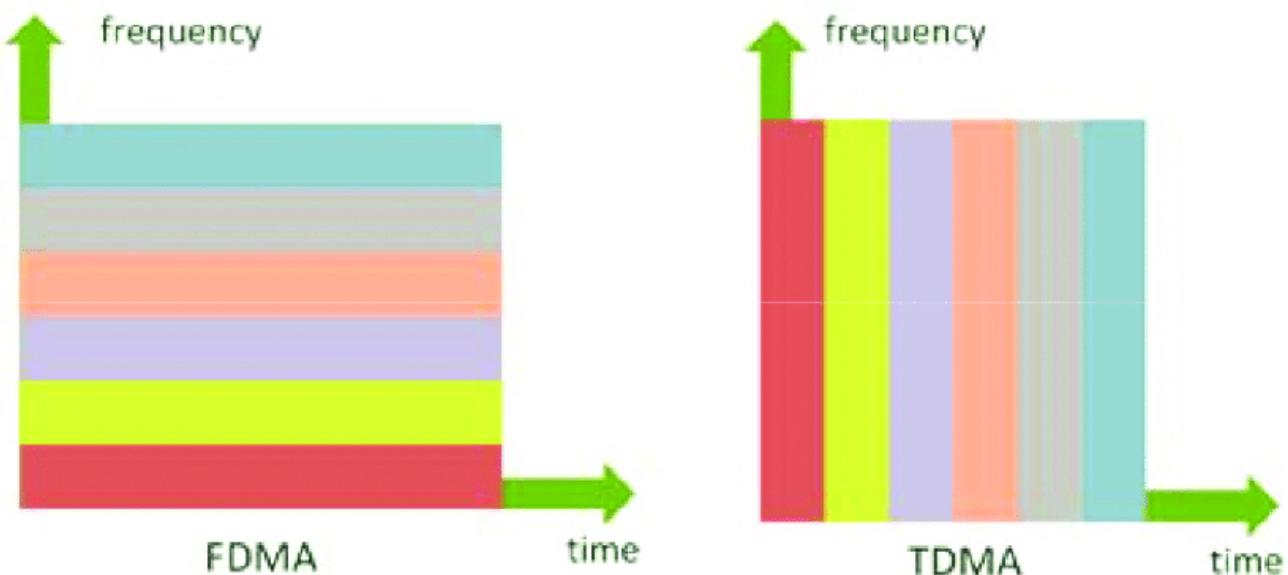


Fig. 1.1: *FDM VS TDM*

Il vantaggio di questo tipo di *commutazione* è che ogni utente al quale è stato assegnato un canale (o slot) sa quali risorse avrà a disposizione. Tuttavia, queste risorse devono essere negoziate prima della trasmissione e quindi è prevista una fase di *handshake* nella quale vengono prenotate e assegnate le risorse. Inoltre, quando l'utente non le usa, quelle risorse sono sprecate.

### 1.3.2 Commutazione a pacchetto

Diversamente da prima, qui il messaggio viene suddiviso in pacchetti di uguale dimensione (tipicamente 1.5MB) che vengono trasmessi singolarmente. Nel momento della trasmissione, il pacchetto utilizza tutta la banda del canale che non viene quindi ripartita tra gli utenti, ma condivisa. In questo modo, l'uso delle risorse avviene a seconda delle necessità e, infatti, questa *tecnica di commutazione* è adatta per situazioni in cui un utente utilizza la rete soltanto per pochi istanti.

Questa tecnica è soggetta a problemi di *congestione* provocata dall'accodamento dei pacchetti in attesa di essere trasmessi e ciò è anche diretta conseguenza del comportamento dei commutatori. I commutatori infatti, prima di trasmettere un pacchetto devono riceverlo per intero. Per questo motivo in ogni commutatore esiste un buffer nel quale vengono memorizzati i pacchetti in attesa di essere trasmessi e, quando questo è pieno, i nuovi pacchetti in arrivo vengono scartati.

Questo problema di perdita dei pacchetti viene risolto (se serve!) da determinati protocolli che verranno trattati più avanti.

**Ritardi nelle trasmissioni** Nella comunicazione in rete, specie con la *commutazione a pacchetto*, *ritardi* e *perdite* sono eventi molto frequenti. Il problema delle *perdite* è già stato discusso. Vediamo quindi quali sono le cause dei *ritardi*:

- *Ritardo di elaborazione del nodo*: il nodo richiede tempo per controllare la correttezza dei dati e per determinare il canale attraverso il quale trasmettere il pacchetto;
- *Ritardo di accodamento*: i pacchetti che devono essere inviati vengono inseriti in un buffer e in caso di congestione gli ultimi pacchetti possono dover attendere a lungo prima di essere trasmessi;
- *Ritardo di trasmissione*: dipende rapporto tra la dimensione in *bit* del pacchetto ( $L$ ) e la *frequenza di trasmissione* in *bit/s* del collegamento ( $R$ ), cioè  $L/R$ ;
- *Ritardo di propagazione*: dipende dal rapporto tra la *lunghezza* ( $d$ ) e la *velocità di propagazione* ( $s$ ) del collegamento fisico, cioè  $d/s$ ;

**NB.** Sebbene possano sembrare simili,  $R$  e  $s$  sono in realtà due grandezze molto differenti!  $R$  è espressa in *bit/s*, mentre  $s$  in *m/s* e tipicamente vale circa  $2 \cdot 10^8 m/s$ .

Il ritardo totale su un nodo è dato quindi dalla somma di tutti questi fattori:

$$d_{nodo} = d_{proc} + d_{accodamento} + d_{trasferimento} + d_{propagazione}$$

Da qui, possiamo definire il concetto di *throughput*:

---

#### Definizione 1 - Throughput.

*È definito throughput la frequenza, espressa in dati/unità di tempo, alla quale una certa unità di dati viene trasferita tra mittente e destinatario.*

## 1.4 Protocoli di comunicazione

Alla fine del paragrafo precedenti abbiamo accennato ai *protocolli di comunicazione*. Diamone qui una prima definizione:

---

## Definizione 2 - Protocollo.

---

Un protocollo definisce il formato e l'ordine dei messaggi scambiati fra due o più entità in comunicazione.

Esistono una gran varietà di protocolli e sono organizzati in uno *stack* detto *stack protocollare*. Esistono due principali *stack protocollari*: lo *Stack TCP/IP* e il *Modello ISO/OSI*.

La differenza tra i due è che il *modello ISO/OSI* è semplicemente un modello di riferimento per la definizione di altri *stack*, mentre il *TCP/IP* è lo *stack* usato nelle comunicazioni in internet.

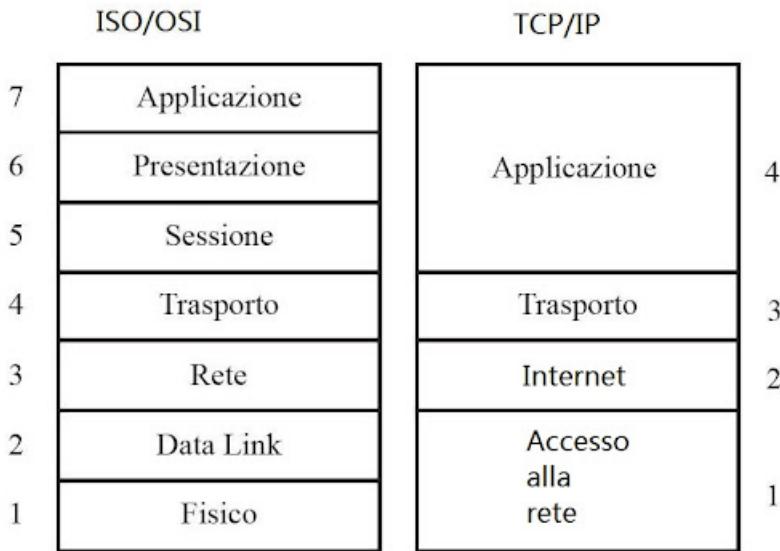


Fig. 1.2: ISO/OSI VS TCP/IP

**Livelli dello stack protocollare ISO/OSI** Esaminiamo il significato dei vari livelli dello stack *ISO/OSI*:

1. *Fisico*: gestisce il trasferimento dei singoli bit
2. *Data link*: organizza l'instradamento dei frame attraverso una serie di commutatori;
3. *Rete*: instrada i pacchetti verso la rete del destinatario;
4. *Trasporto*: gestisce l'instradamento dei segmenti verso una specifica applicazione attiva nella rete di destinazione;
5. *Sessione*: gestisce sessioni di comunicazioni e la sincronizzazione dei flussi di dati (e.g. streaming audio-video);
6. *Presentazione*: consente alle applicazioni di interpretare il significato dei dati gestendo parametri come compressione e cifratura;
7. *Applicazione*: fornisce alle applicazioni i mezzi per scambiarsi messaggi;

Il motivo per cui nello *stack TCP/IP* mancano i livelli *sessione* e *presentazione* è che possono essere inclusi nei protocolli di *livello Applicazione*;

**Incapsulamento** La stratificazione dei protocolli consente di rendere modulare la comunicazione in rete. La modularità porta numerosi vantaggi, tra i quali:

- *Trasparenza*: una modifica ai protocolli di un livello è trasparente ai protocolli di altri livelli;
- *Semplificazione*: il modello a strati permette di identificare più facilmente i diversi componenti di un sistema che altrimenti risulterebbe estremamente complesso;

**Comunicazione tra livelli** Ogni *livello (layer)* fornisce un servizio al *livello superiore*, il quale vede il *livello inferiore* come una black-box della quale sfruttarne i servizi.

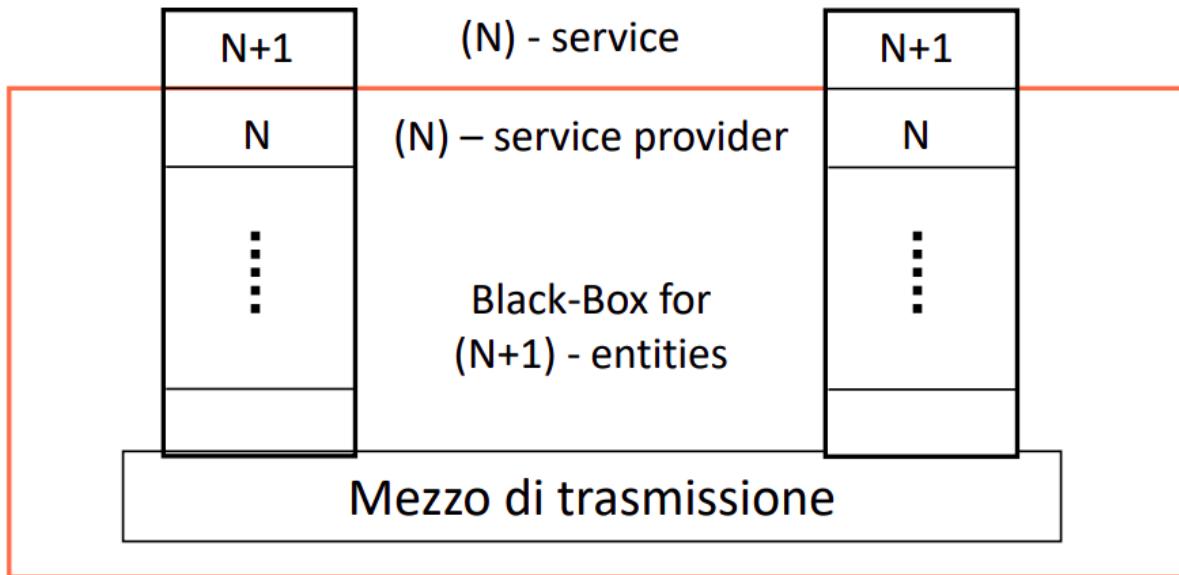


Fig. 1.3: Modularizzazione dei servizi

La comunicazione tra livelli avviene mediante *SAP*, in modo che un servizio del *livello N* è offerto all'entità di *livello N+1* attraverso un'interfaccia di programmazione detta appunto *SAP*. Lo scambio di informazioni tra entità dello stesso *livello*, invece, è regolata dai *protocolli*.

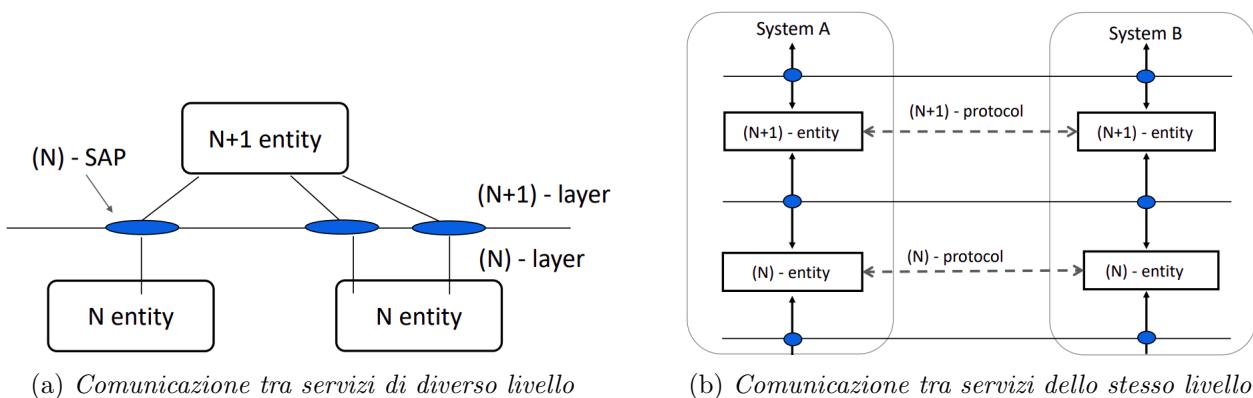


Fig. 1.4: Comunicazioni tra servizi

**Data unit** La suddivisione in livelli dei servizi cambia il modo in cui i dati, o meglio le *DU* da trasmettere vengono trattate. In un sistema a livelli, i dati da trasmettere da un *livello N* costituiscono un *N-SDU* (*SDU* di *livello N*).

A questo *N-SDU* il *livello* aggancia il proprio *N-PCI* (*PCI* di *livello N*), cioè aggiunge le informazioni necessarie al *protocollo* per funzionare. Il risultato è un *N-PDU* (*PDU* di *livello N*).

Ogni *livello* considera il *PDU* del *livello superiore* come una busta chiusa. Cioè, l'*N-PDU* è l'*SDU* di *livello N-1* e aggiungendo l'*(N-1)-PCI* all'*(N-1)-SDU* si ottiene l'*(N-1)-PDU*.

**Trasmissione e ricezione** Nel momento della trasmissione i dati veri e propri vengono progressivamente imbustati insieme al *PCI* di *livello*, partendo dal *livello Applicativo* e scendendo fino al *Fisico*. Per la ricezione, invece, si segue il processo inverso, quindi partendo dal *Fisico* e salendo all'*Applicativo* ogni *livello* rimuove il proprio *PCI*.

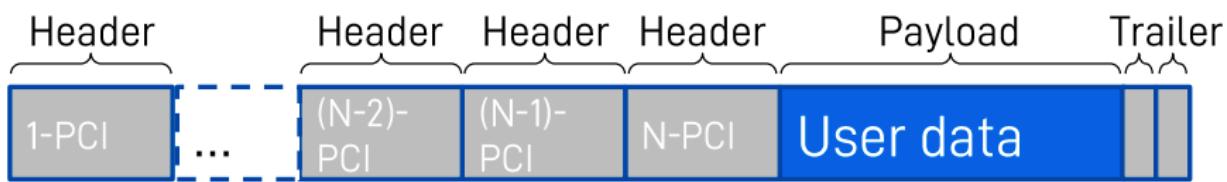


Fig. 1.5: Rappresentazione di un *PDU* alla fine del processo di imbustamento

Le *DU* possono essere:

- *Segmentate*: se la dimensione dei dati eccede il limite massimo, possono essere suddivisi in più *SDU*;
- *Assemblate*: per evitare inefficienze, più *N-SDU* di piccola dimensione possono essere aggregate in un unico *N-SDU* e trasmesse insieme;
- *Ri-assemblate*: è il processo inverso della *segmentazione*;

**Le PDU di livello** A seconda del *livello* in cui si trovano le *PDU* possono avere nomi diversi:

- *Applicazione*: messaggi;
- *Trasporto*: segmenti;
- *Rete*: pacchetti;
- *Data link*: frame;

## *Capitolo Nr.2*

---

### *Livello Applicativo*

---

Il *livello Applicativo* è l'ultimo livello dello *stack protocollare TCP/IP* e costituisce un'interfaccia per i processi che intendono comunicare sulla rete.

Vediamo più nel dettaglio le caratteristiche e le funzionalità, o meglio i *protocolli*, forniti da questo livello.

## **2.1 Applicazioni di rete**

Uno degli utilizzi più diffusi della rete è la creazione di *applicazioni di rete*, ovvero applicazioni software diffuse su più calcolatori.

### **2.1.1 Architetture di applicazioni di rete**

Le *applicazioni di rete* possono essere basate su architetture diverse:

- *Client-server*: esistono *server* che offrono servizi ad altri *host* detti *client*. I *client* possono comunicare soltanto con i *server*;
- *P2P*: non esistono *server*, ma tutti gli *host* sono pari tra loro. Ciascun *host*, o *peer*, può comunicare con qualsiasi altro *peer*;
- *Architetture ibride*: sono architetture che hanno sia componenti *client-server* che *peer-to-peer*;
- *Cloud computing*: le risorse sono distribuite su uno o più server dislocati nel territorio e sono virtualizzate in modo che ogni utente possa utilizzare solo le risorse di cui ha bisogno;

Ma vediamole più nel dettaglio.

**Architettura client-server** Nelle reti basate sul modello *Client-Server*, esiste un *server* centrale al quale molti *client* si connettono per richiedere una risorsa o usufruire di un servizio. Il *server* deve essere sempre attivo e avere un *indirizzo IP* fisso, così da essere raggiungibile dai *client*, i quali, possono comunicare solo con lui e non tra loro.

Esistono quindi 2 entità:

- *Client*: si tratta di un componente hardware o software che invia delle richieste al *server* e attende che questo risponda per poterne elaborare la risposta;

- *Server*: si occupa di soddisfare le richieste ricevute dai *client* e di restituire loro il risultato delle suddette richieste;

Tale modello ha come principale vantaggio la semplicità di gestione delle risorse condivise, che risiedono tutte nel *server*. Inoltre, permette di ridurre i costi lato *client* perché non è necessario che questi abbiano prestazioni elevate. Tuttavia, nel caso di guasti o malfunzionamenti del *server*, l'intera rete potrebbe risultare rallentata o inutilizzabile.

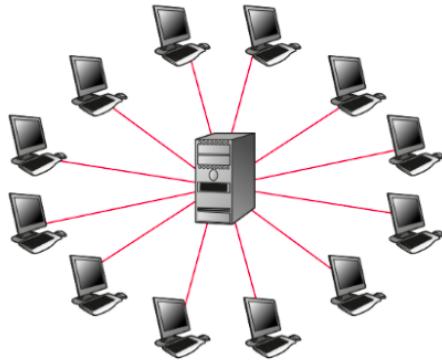


Fig. 2.1: Schema architettura *client-server*

**Architettura P2P** Il modello *P2P* prevede la presenza di entità autonome dette *peer* che scambiano tra di loro delle risorse. Ogni *peer* può sia condividere risorse che richiederne.

Esistono diversi tipi di architetture *P2P*:

- *P2P decentralizzato*: ogni *peer* svolge sia la funzione di server che di client. Il sistema è in grado di adattarsi autonomamente a un cambiamento nei nodi partecipanti senza richiedere interventi esterni e mantenendo attiva la rete;
- *P2P centralizzato*: Esiste un server centrale detto *directory server* che contiene informazioni utili alla localizzazione delle risorse tra i *peer*. Ogni *peer* informa il server del tipo di risorse che intende condividere, mentre, quando ne richiede una, chiede al *directory server* di localizzarla;
- *P2P ibrido*: Esistono alcuni *peer* detti *super-peer*, *super-nodi* o *ultra-peer* determinati dinamicamente da un algoritmo e che forniscono informazioni sulla localizzazione delle risorse agli altri *peer*, detti *leaf-peer*;

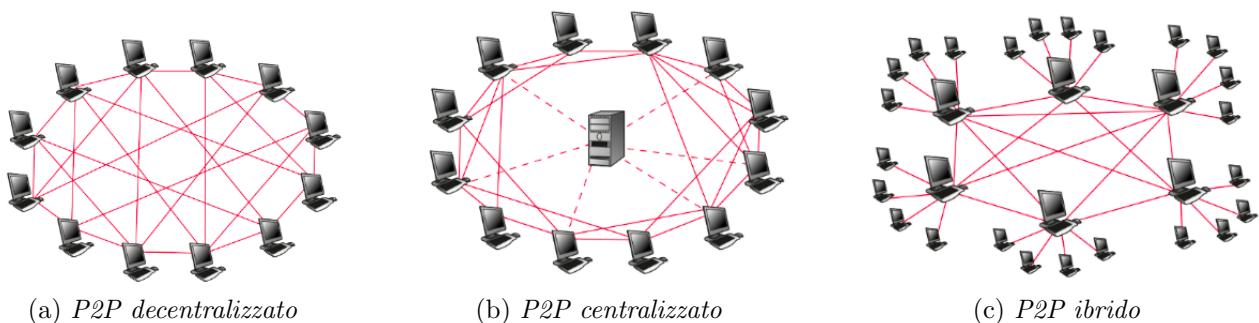


Fig. 2.2: Architetture *P2P* a confronto

**Cloud computing** Il *cloud computing* è un insieme di risorse IT (hardware e software) distribuite e accessibili attraverso la rete. In particolare, questa architettura viene implementata come un *sistema distribuito*, su diversi server dislocati nel mondo, che vengono utilizzati per fornire un servizio.

Per accedere alle risorse i client contattano un *endpoint*, cioè un server che funge da interfaccia e che si occupa di recuperare le risorse richieste contattando gli altri server della rete.

Questo modello porta con sé numerosi vantaggi:

- *Scalabilità*: è possibile aumentare o diminuire le risorse in base alle necessità;
- *Velocità di configurazione*: attivare, modificare o disattivare un servizio è immediato;
- *Risparmio per l'utilizzatore*: il costo varia in base all'utilizzo effettivo delle risorse, le quali, sono assegnate dinamicamente dal gestore con un sistema di *provisioning dinamico* e con granularità fine;
- *Risparmio per il fornitore*: le risorse sono virtualizzate, ovvero condivise tra tutti gli utizzatori. Ciò implica che il sistema può avere a disposizione meno risorse di quelle che servirebbero a soddisfare le richieste dei client se queste venissero fatte tutte contemporaneamente (è un evento raro);
- *Affidabilità*: eventuali disservizi sono risolti dal gestore e non dall'utilizzatore;

Ma ci sono anche delle criticità:

- *Sicurezza e privacy*: più utenti accedono allo stesso server per diverse ragioni, per cui è necessario assicurare che le attività di un utente non interferiscano con i dati o i processi degli altri;
- *Problemi internazionali di natura economica e politica*: uno stato potrebbe decidere di impedire l'accesso ad un certo servizio offerto da un ente estero (e.g. servizi di Google non accessibili in Cina);
- *Continuità del servizio offerto*: i fornitori di servizi devono garantire agli utenti la continua fruibilità dei propri servizi;
- *Difficoltà di migrazione dei dati*: su un server risiedono molti dati per cui è complicato trasferirli da un'altra parte;

**NB.** Quando si parla di *scalabilità* è possibile distinguere due casi:

- *Scalabilità orizzontale*: aggiungere nodi fornitori (server) alla rete;
- *Scalabilità verticale*: incrementare le risorse di un singolo nodo;

Nel caso del *cloud computing* viene esaltata la *scalabilità orizzontale*.

**Data center** I fornitori di servizi cloud, dovendo gestire molti server, si appoggiano a strutture attrezzate dove sono allocati i server di uno o più fornitori. Queste strutture sono dette *data center* e si occupano di garantire che i server funzionino anche in caso di problemi quali, ad esempio, blackout o incendi.

La figura sottostante mostra la struttura gerarchica della rete interna di un *data center*. In particolare, ogni server è collegato ad una rete d'accesso; le reti d'accesso sono poi collegate ad una rete intermedia che aggrega le comunicazioni provenienti dai server e le dirige verso la parte interna della rete.

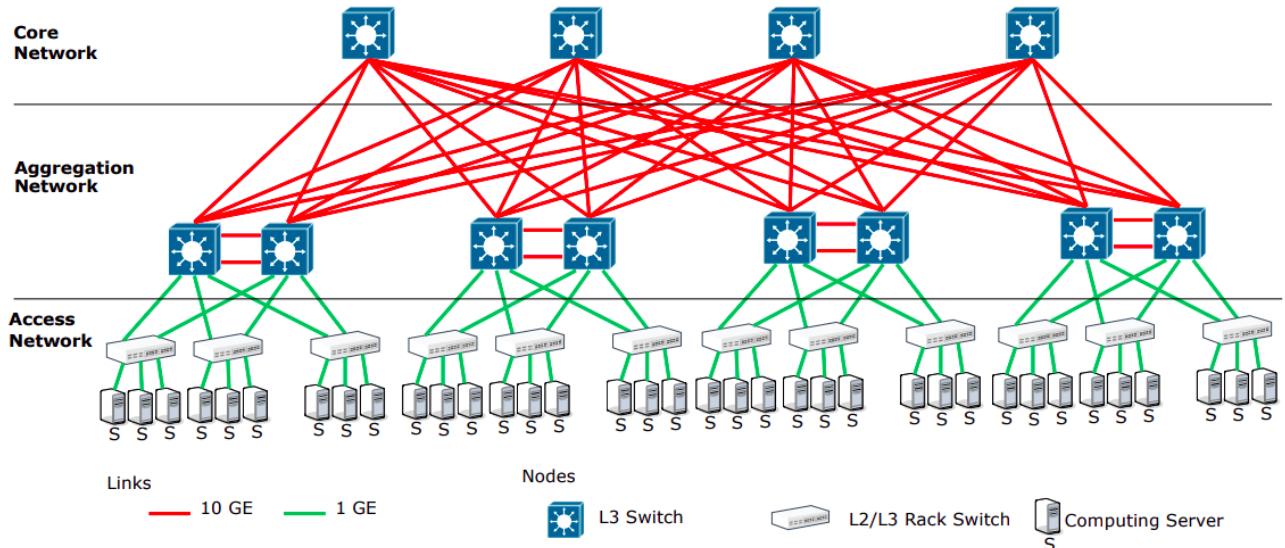


Fig. 2.3: Schema architetturale di un *data center*

**Content Delivery Network** La struttura interna di un *data center* può essere paragonata alla struttura organizzativa dei server di grandi aziende IT come Google. Google ha 14 “*mega-data center*” (100 000 server per *data center*) distribuiti nel mondo che servono contenuti dinamici e personalizzati (e.g. gmail, ricerche), 50 cluster (100-500 server per cluster) di calcolo “*bring home*” nelle reti degli *IXP* che servono contenuti statici (e.g. video di Youtube) e centinaia di cluster (decine di server per cluster) “*enter deep*” nelle reti degli *ISP* che forniscono contenuti statici (e.g. parti statiche nelle pagine dei risultati di ricerca) e permettono il *TCP splitting*.

Un’organizzazione di questo tipo permette la realizzazione di *CDN*, reti “*overlay*” per la diffusione di contenuti. L’idea alla base delle *CDN* è di portare i contenuti il più vicino possibile all’utente finale in modo da ridurre la latenza, evitare colli di bottiglia e migliorare le prestazioni generali della rete.

Esistono due tecniche per la realizzazione di *CDN*:

- *Enter deep*: vengono installati server fisici nelle reti degli *ISP* riducendo il ritardo e migliorando il *throughput* percepito dagli utenti;
- *Bring home*: vengono installati server fisici nelle reti degli *IXP* così da poter servire più *ISP* contemporaneamente;

## 2.2 Comunicazione in rete tra processi

La comunicazione tra utenti si realizza come comunicazione tra processi, ovvero programmi in esecuzione su un *host*. Processi che comunicano tra loro sullo stesso *host* usano *schemi interprocesso* definiti dal sistema operativo. Quando, invece, a comunicare sono processi su *host* diversi, lo fanno mediante lo scambio di messaggi sulla rete.

Nella comunicazione in rete distinguiamo due tipi di processi:

- *Processi server*: forniscono un servizio e attendono di essere contattati da altri processi,
- *Processi client*: usufruiscono di uno o più servizi richiedendoli a uno o più *processi server*.

**NB.** Nelle architetture *P2P* le applicazioni hanno sia *processi server* che *processi client*.

Per realizzare la comunicazione in rete tra processi, questi devono poter essere identificati all'interno della rete. Per farlo, è necessario conoscere gli *indirizzi IP* (indirizzi del *livello Rete*) e i *numeri di porta* dei processi (indirizzi del *livello Trasporto*) e questa coppia di parametri è detta *socket*.

### 2.2.1 Socket

---

#### Definizione 3 - Socket.

Una *socket* è un'interfacci di un host, creata da un'applicazione e controllata dal Sistema Operativo, tramite la quale un processo di quell'applicazione può scambiare messaggi con il processo di un'altra applicazione.

Le *socket* sono basate sull'architettura *client-server* quindi esistono *socket client* e *socket server*.

La creazione e l'utilizzo delle *socket* fa uso di API messe a disposizione da una *socket API* che permettono anche di scegliere il protocollo di *livello Trasporto* da utilizzare: *UDP* o *TCP*.

**Socket TCP** Le *socket TCP* utilizzando il protocollo *TCP* per il trasporto dei messaggi tra un processo e l'altro.

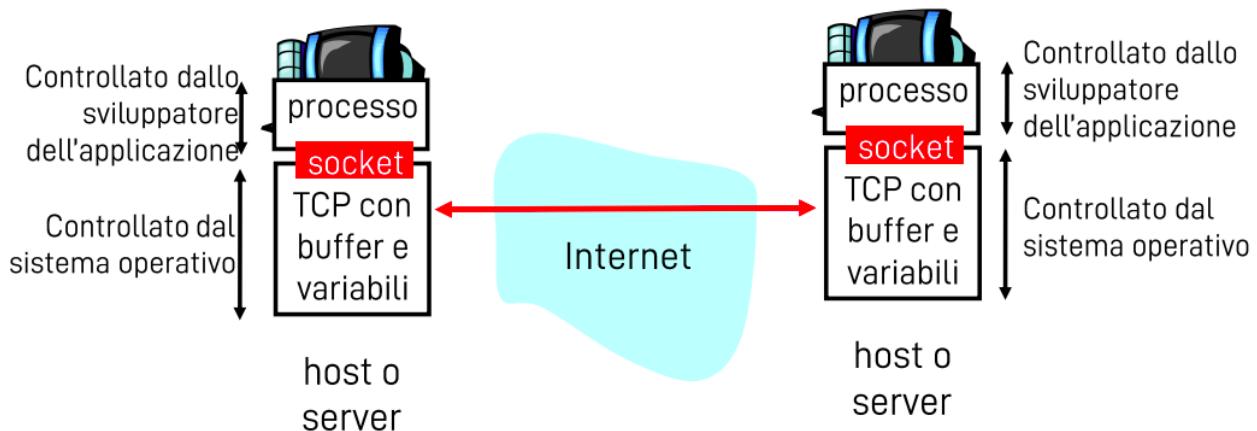


Fig. 2.4: Funzionamento *socket TCP*

Al momento della creazione della *server socket* viene specificato il *numero della porta* sulla quale il server si metterà in ascolto, mentre nella creazione della *client socket* vengono specificati l'*indirizzo IP* e il *numero di porta* del processo server.

Inoltre, quando il client crea la *socket* viene stabilita una connessione *TCP* con il server, il quale, risponde creando una nuova *socket* per comunicare con quel client. Questo comportamento consente al server di comunicare con più client contemporaneamente distinguendoli in base alla *porta sorgente*<sup>1</sup>.

**Socket UDP** Le *socket UDP* comunicano utilizzando il protocollo *UDP* e, diversamente da quanto avviene nelle *socket TCP*, il client non si connette con il server, ma include in ogni pacchetto l'*indirizzo IP* e il *numero di porta* del processo server. Di conseguenza, il server, per distinguere i client, deve estrarre da ogni pacchetto l'*indirizzo IP* e il *numero di porta* del mittente.

---

<sup>1</sup>Nel corso della trattazione questo verrà reso più chiaro

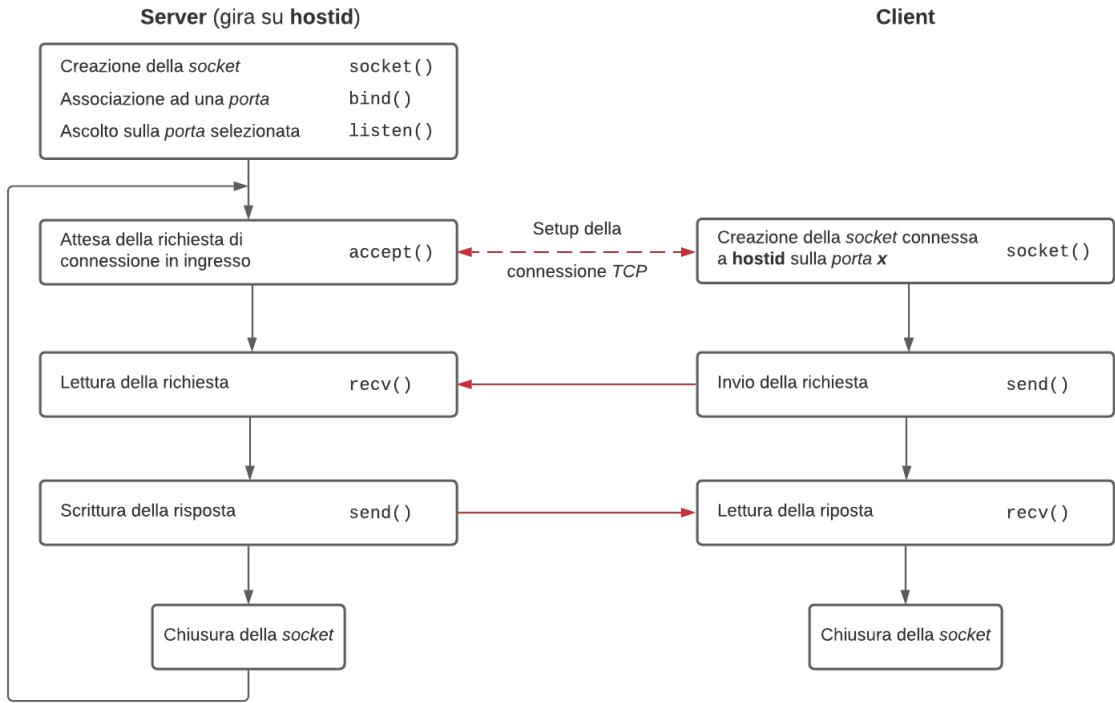


Fig. 2.5: Comunicazione tra *socket TCP*

## 2.3 Protocolli del livello Applicativo

Prima di passare alla discussione sui protocolli più importanti del *livello Applicativo* è meglio chiarire la terminologia.

Una pagina web è costituita da *oggetti*; un *oggetto* può essere un file **HTML**, un’immagine, un file audio, . . . ; una pagina web è costituita da un file base scritto in **HTML**, che solitamente include altri oggetti referenziati; ogni oggetto è referenziato da un **URL**.

---

### Definizione 4 - URL.

Un *URL* è una sequenza di caratteri che identifica univocamente una risorsa nella rete. La struttura di un *URL* è la seguente:

`protocollo://[username[:password]@]host[:porta] [</percorso>] [?queryString] [#fragment]`

### 2.3.1 Protocollo HTTP

L’**HTTP** è un protocollo basato sul modello *client-server*: il ruolo di *client* è ricoperto dai browser che richiedono, e ricevono, oggetti dai *server web*. La comunicazione tra *client* e *server* si realizza tramite lo scambio di messaggi, che vengono scambiati sfruttando *socket TCP*. In particolare, il *client* crea una connessione *TCP* sulla porta 80 del *server*, il quale risponde accettando la connessione. Questa connessione viene poi chiusa al termine dello scambio di messaggi.

**Connessioni persistenti e non** Le *connessioni HTTP*, ovvero le *connessioni TCP* create dal protocollo *HTTP*, possono essere di due tipi:

- *Persistenti*: prima che la connessione venga chiusa possono essere trasmessi più oggetti;

- *Non persistenti*: prima che la connessione venga chiusa viene trasmesso un solo oggetto;

---

### Definizione 5 - RTT.

---

È definito *RTT* il tempo di propagazione di andata e ritorno tra due host.

**NB.** Con *tempo di propagazione* si intende, ad esempio, il tempo impiegato da un piccolo pacchetto per andare dal *client* al *server* e ritornare al *client*.

Le *connessioni non persistenti* hanno un *RTT* doppio perché per ogni oggetto va riaperta una nuova *connessione TCP*. Questa caratteristica porta anche il server a dover far fronte ad un maggiore overhead del sistema operativo che deve gestire l'apertura di tutte quelle connessioni *TCP*. Inoltre, spesso accade che i browser aprano più connessioni in parallelo per il trasferimento degli oggetti referenziati.

Con *connessioni persistenti* tutti questi problemi non si presentano, ma se il *server* comunicasse con molti *client* potrebbe terminare tutte le porte disponibili e quindi potrebbe non poter più aprire nuove connessioni.

L'*HTTP/1.0* utilizzava *connessioni non persistenti*, mentre dalla versione *1.1* utilizza *connessioni persistenti*.

**Struttura dei messaggi** Il protocollo *HTTP* prevede due tipi di messaggi:

- *Richieste HTTP*: sono inviate dal *client*;
- *Risposte HTTP*: sono inviate dal *server* in risposta ad una richiesta;

I messaggi di *richiesta* hanno la seguente struttura:



Fig. 2.6: Struttura messaggio di *richiesta HTTP*

Il campo *metodo* può essere uno dei seguenti:

- *GET*: richiede un file al *server* scrivendo il percorso, ed eventuali altri dati, in chiaro nell'*URL*;
- *POST*: invia informazioni all'*URL* specificato scrivendo i dati nel corpo del *messaggio HTTP*;
- *PUT*: carica un file sul *server*;
- *HEAD*: richiede solo l'header della risposta senza la risorsa;
- *OPTIONS*: richiede l'elenco dei metodi permessi dal *server*;
- *DELETE*: cancella una risorsa sul *server*;
- *TRACE*: traccia una richiesta visualizzando come viene trattata dal *server*;

La *versione* invece può essere uno tra: *HTTP/1.0*, *HTTP/1.1* e *HTTP/2.0*. Tra le *righe d'intestazione* deve obbligatoriamente essere presente il campo *host* nel quale è indicato il nome dell'*host* da raggiungere.

I messaggi di *risposta* hanno una struttura pressoché uguale:

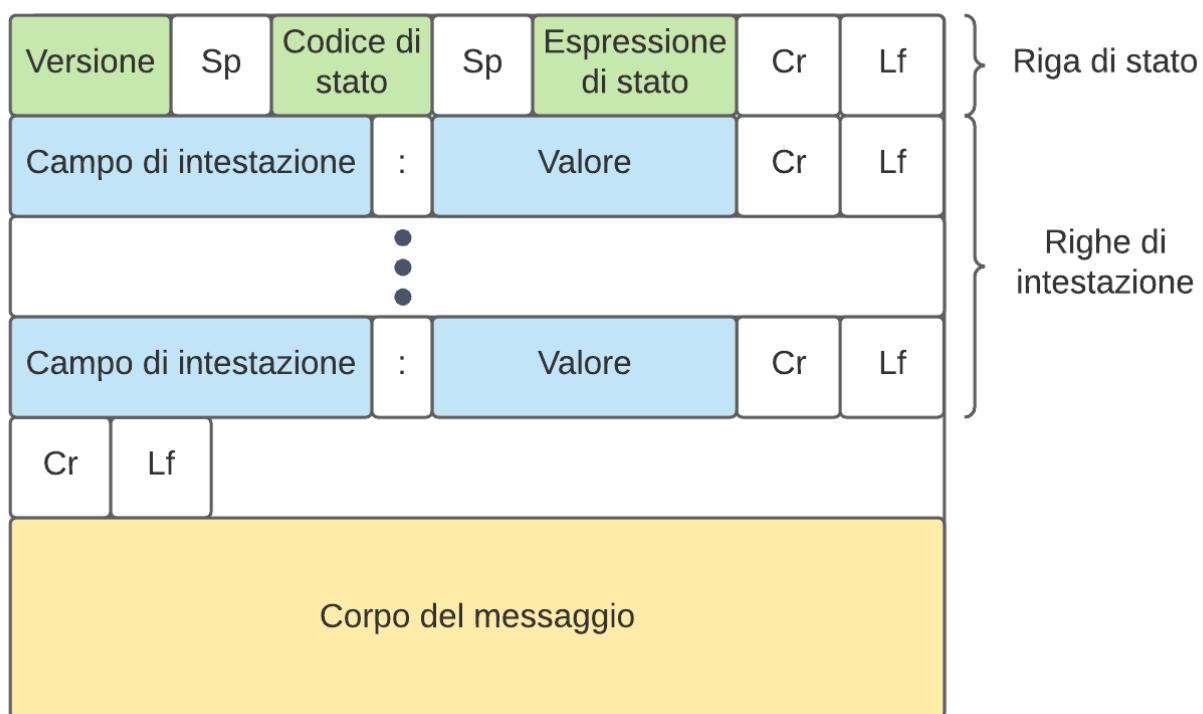


Fig. 2.7: Struttura messaggio di *risposta HTTP*

Il *codice di stato* è un numero di tre cifre che identifica il tipo di *risposta* ed è definito secondo il seguente formato:

- *1XX*: richiesta ricevuta, risposta in arrivo;
- *2XX*: richiesta ricevuta e soddisfatta;
- *3XX*: ridirezione necessaria;
- *4XX*: errore del client;

- 5XX: errore del server;

L'espressione *di stato* è invece una descrizione del *codice di stato*.

**HTTP/2.0** Il protocollo *HTTP/2.0* è un'evoluzione dell'*HTTP/1.1* ed è progettato per ridurre la latenza percepita dall'utente e l'utilizzo delle risorse di rete e dei server.

In particolare, l'*HTTP/2.0* tenta di utilizzare un'unica connessione tra client e server per richiedere tutte le risorse. È basato su *SPDY*, un protocollo del *livello Applicativo* per il trasporto di contenuti sul web con latenza minima. Per fare ciò, combina tre fattori:

- *Multiplexing di flussi*: su una singola *connessione TCP* possono transitare un numero illimitato di flussi di dati;
- *Priorità delle richieste*: il client può inviare un numero indefinito di richieste, specificando per ciascuna una priorità;
- *Compressione dell'header HTTP*: l'header *HTTP* viene compresso in modo da usare un minor numero di byte;

Il protocollo *HTTP/2.0* introduce un livello intermedio tra il *livello Applicativo* e il *Trasporto* (in realtà tra l'*Applicativo* e il *Sessione*), chiamato *binary framing*. Questo livello si occupa di gestire le modalità di incapsulamento e trasferimento dei *messaggi HTTP*.

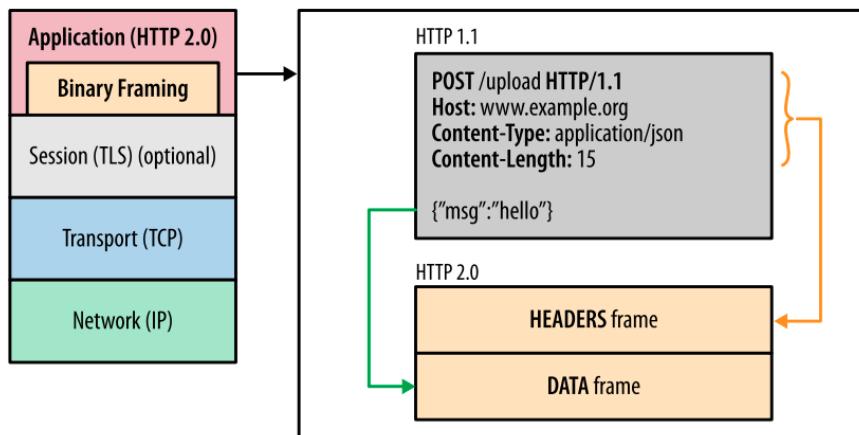


Fig. 2.8: Livello *binary framing*

Il *framing binario* lascia invariata la semantica dell'*HTTP*, ma ne modifica la codifica in fase di transito: ogni *messaggio HTTP* è suddiviso in *frame* più piccoli codificati in binario.

**NB.** Questa caratteristica dell'*HTTP/2.0* lo rende incompatibile con le precedenti versioni, quindi server *HTTP/1.x* e *HTTP/2.0* non possono comunicare tra loro.

Ogni *frame* è identificato da un valore univoco ed eventualmente anche da un livello di priorità. Ogni messaggio trasmesso, che sia una richiesta o una risposta, può essere suddiviso in uno o più *frame* e ogni *frame* trasporta un tipo specifico di dati: *header* o *payload*.

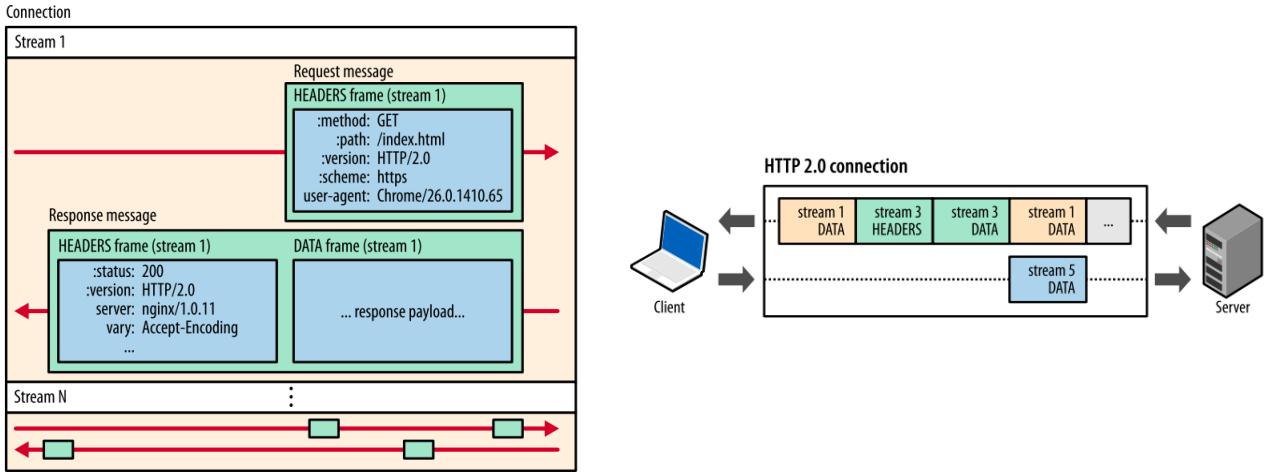


Fig. 2.9: Trasmissione dei *frame* negli *stream*

Tutte le comunicazioni avvengono all'interno di un'unica *connessione TCP* che può trasportare un numero illimitato di *stream* bidirezionali di byte. Su ogni *frame* è indicato l'identificativo unico dello *stream* sul quale sta viaggiando e questo consente di inviare i *frame* in modo indipendente, intervallandoli e ricomponendoli all'arrivo.

L'ordine di invio dei *frame* dipende, qualora sia stata impostata, dalla priorità indicata in senso crescente con un numero da 1 a 256. L'ordine di invio dei *frame* viene deciso dal server in base alle indicazioni fornitegli dal client tramite un *albero di priorità*.

In *HTTP/2.0* esiste quindi una sola connessione *TCP persistente*.

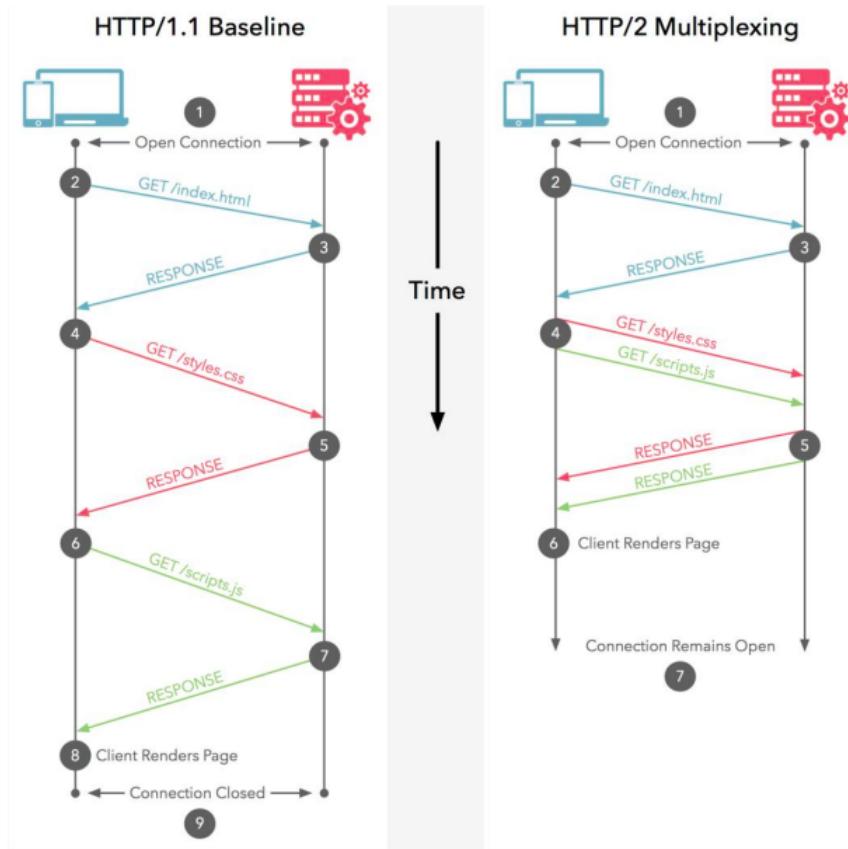


Fig. 2.10: Connessione *HTTP/1.1* VS *HTTP/2*

Per tentare di ridurre ulteriormente la latenza, l'*HTTP/2.0* introduce il concetto di *server push*. Quando il client fa una richiesta al server, questo oltre alla risorsa richiesta invia anche altre risorse ad essa collegate. Queste risorse inviate in più dal server sono dette *server promise*.

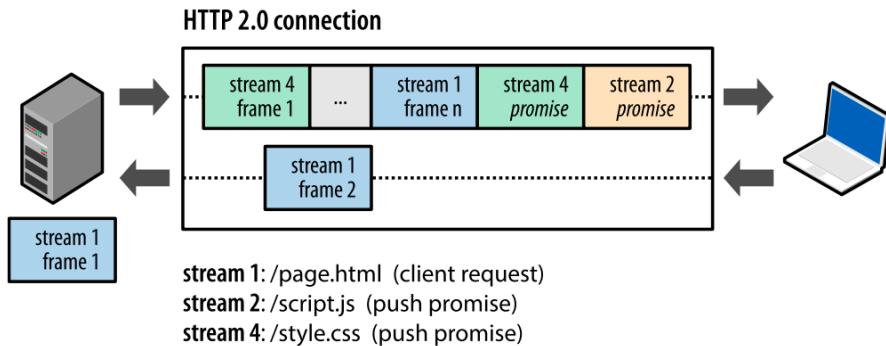


Fig. 2.11: *HTTP/2.0 server push*

**Comunicazioni senza stato e i cookie** Nell'*HTTP* la comunicazione tra client e server è priva di stato, ciò significa che il server non ha memoria delle precedenti richieste di un client.

Per simulare lo stato si usano i *cookie*, ovvero file generati dal web server e salvati sul client che contengono, tra le altre cose, informazioni sulle preferenze di un client riguardo un sito web (e.g. carrello della spesa nei siti di e-commerce).

In particolare, i *cookie* sono file contenenti una stringa di testo, una data di scadenza e un pattern per il riconoscimento dei domini di destinazione che vengono inviati al web server ogni volta che il client accede ad una pagina del sito. La stringa di testo contenuta in un cookie è detta *chiave di sessione* ed è utilizzata per identificare in modo univoco un client e i relativi dati che vengono salvati in un database accessibile al server web.

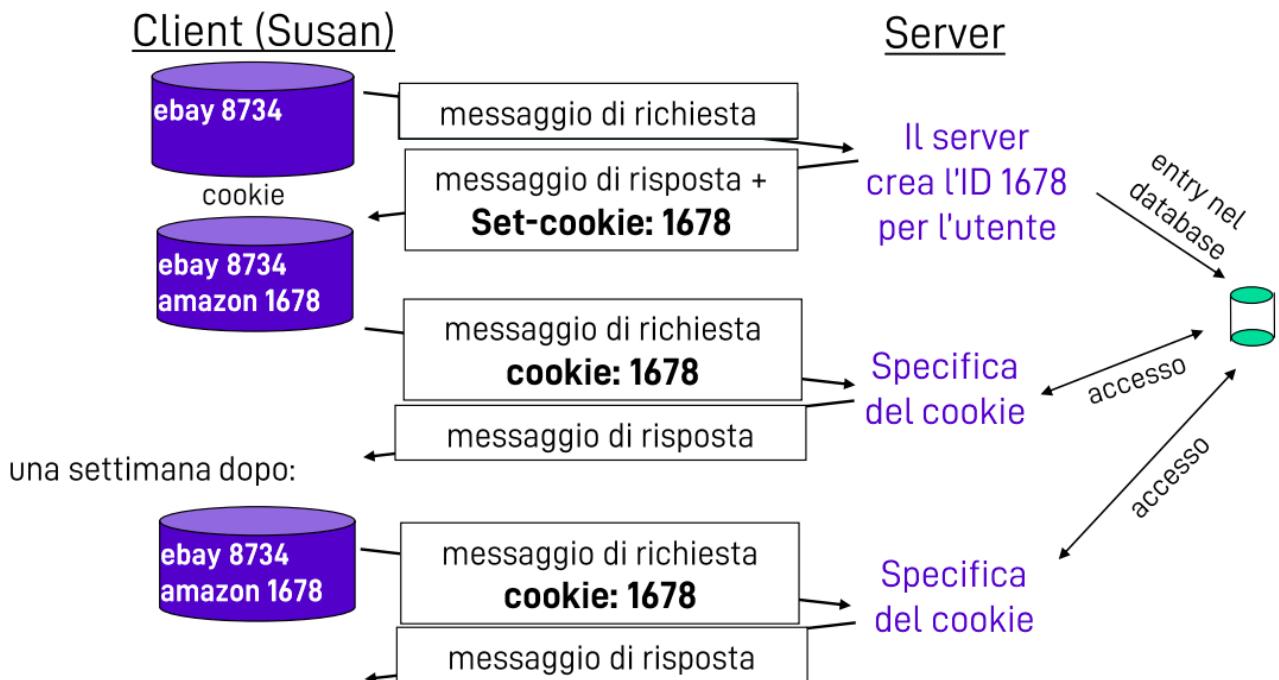


Fig. 2.12: Utilizzo dei *cookie*

La prima volta che un client accede ad una pagina web, il server crea una *chiave di sessione* e ne specifica il valore nel campo *Set-cookie* dell'*header HTTP* del messaggio di risposta. Il

browser del client salva quel valore e nelle successive richieste lo va ad inserire nell'*header* nel campo *Cookie*.

**Server proxy** Il *proxy* è un server intermedio che si pone tra client e server. Quando un client richiede una risorsa al server, la richiesta viene prima intercettata dal *proxy* che verifica la presenza della risorsa nella propria cache interna. Se nel *proxy* la risorsa richiesta non è presente o non è aggiornata, la richiesta viene inoltrata dal *proxy* al server di destinazione.

Quindi, il *proxy* riceve la risposta dal server e prima di inviarla al client, provvede a salvarla al suo interno, così da poterla riutilizzare per future richieste senza dover ricontattare il server.

Tuttavia, nel caso di risorse dinamiche è importante che queste siano aggiornate, quindi quando il *proxy* riceve una richiesta, invia un *messaggio HTTP* di tipo *HEAD* al web server e confronta la data di ultima modifica della pagina salvata con quella della pagina sul web server. Se la pagina salvata è scaduta, il *proxy* la aggiorna e risponde alla richiesta del client con la versione aggiornata. In alternativa ad una richiesta *HEAD* seguita da una *GET*, il *proxy* può utilizzare una *GET condizionale* che richiede la risorsa solo se la data di ultima modifica è successiva a quella indicata nell'*header* nel campo *If-modified-since*.

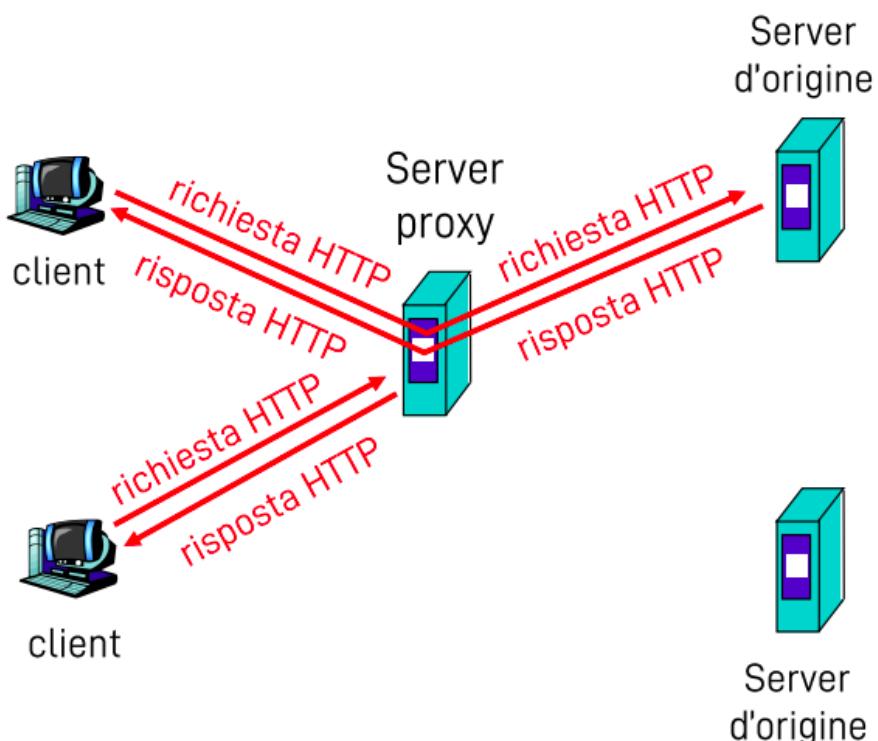


Fig. 2.13: Server *proxy*

### 2.3.2 Protocollo FTP

L'**FTP** è un protocollo *client-server* per il trasferimento di file tra client e server. Vengono usate due connessioni *TCP* sulle porte 21 e 22 del server. La prima è dedicata allo scambio dei comandi, mentre la seconda, viene aperta all'occorrenza e viene mantenuta attiva soltanto per il tempo necessario, ed è usata per il trasferimento dei file.



Fig. 2.14: Connessioni aperte dall'*FTP*

### 2.3.3 Protocolli per la posta elettronica

Nello scambio di messaggi di posta elettronica gli attori principali sono tre:

- *Agente utente*: permette la composizione e la lettura delle email;
- *Server di posta*: contiene i messaggi da recapitare all'utente e la coda di messaggi da trasmettere;
- *Protocollo di trasferimento*: gestisce il trasporto di email tra utente e server e tra server;

**Protocollo SMTP** È utilizzato per mettere in comunicazione i *server di posta* e, in particolare, trasmette i messaggi direttamente tra il server mittente e il server destinatario. Il trasferimento è articolato in tre fasi: *handshake*, *trasferimento* e *chiusura*.

L'interazione tra i server avviene mediante messaggi in formato ASCII a 7 bit; ad ogni comando corrisponde un messaggio di risposta formato da un codice di stato e un'espressione. Inoltre, l'**SMTP** trasmette più oggetti all'interno di uno stesso messaggio e utilizza *connessioni persistenti*.

```

S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr ... Sender ok
C: RCPT TO: <rob@hamburger.edu>
S: 250 rob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: From: alice@crepes.fr
C: To: bob@hamburger.edu
C: Subject: Important question.
C:
C: Do you like ketchup?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection

```

Fig. 2.15: Esempio di interazione tra *server SMTP*

È possibile trasmettere anche oggetti multimediali utilizzando l'estensione *MIME* e codificando i dati da trasmettere usando la codifica *base64* che rappresenta 6 bit usandone 8.

Per utilizzare l'estensione *MIME* è sufficiente aggiungere nell'intestazione del messaggio le seguenti tre righe:

```

MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

```

Fig. 2.16: Intestazione per messaggi multimediali

Ovviamente il **Content-type** dipende dal tipo di oggetto che si vuole trasferire.

**Protocolli POP3 e IMAP4** *POP3* e *IMAP4* sono usati per il trasferimento di messaggi tra *server* e *agenti utente*. Entrambi utilizzano il protocollo *TCP* per lo scambio di messaggi, ma a parte questo sono basati su filosofie opposte.

Il protocollo *POP3* prevede che l'utente si colleghi al server e scarichi tutte le email disponibili. Una volta scaricate, le email vengono eliminate dal server. D'altra parte, il protocollo *IMAP4* consente all'utente di accedere alle email mantenendole nel server e inoltre permette l'organizzazione dei messaggi in cartelle. Per questo motivo, l'*IMAP4* mantiene lo stato dell'utente tra le varie sessioni, mentre il *POP3* no.

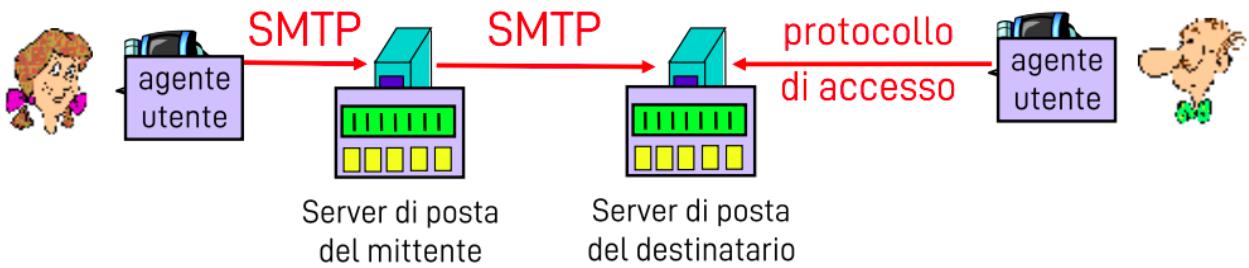


Fig. 2.17: Scambio di email

**NB.** Ci si potrebbe chiedere perché non si usa soltanto l'*SMTP*. La risposta è che l'*SMTP* può trasmettere un messaggio soltanto se il ricevente è online.

**Protocollo TLS** Il *TLS* è un protocollo che consente di rendere sicura la comunicazione tra due entità fornendo:

- *Autenticazione*: l'identità di mittente e destinatario viene verificata;
- *Integrità dei dati*: viene garantito che il messaggio non verrà manipolato durante la trasmissione;
- *Confidenzialità*: soltanto il legittimo destinatario sarà in grado di accedere al contenuto del messaggio;

Il funzionamento del *TLS* può essere suddiviso in tre fasi: *negoziazione fra le parti dell'algoritmo da utilizzare*, *scambio delle chiavi* e *autenticazione*, *cifratura simmetrica* e *autenticazione dei messaggi*.

È possibile combinare i tre protocolli per lo scambio di email con il *TLS* per rendere sicuro lo scambio di messaggi. Ovviamente, un *host* che non utilizza questa tecnologia non può comunicare con un altro *host* che la usa, quindi, vengono usati *numeri di porta* diversi dalle versioni "non sicure" di quei protocolli.

Esiste un'evoluzione del *TLS* chiamata *STARTTLS* che consente di comunicare sulle porte originali dei protocolli. In particolare, il client che intende assicurare la comunicazione, chiede al server l'instaurazione di una connessione cifrata, la sessione inizia in chiaro e diventa cifrata prima che vengano trasmessi dati sensibili.

### 2.3.4 Protocollo DNS

Ad ogni *nome di dominio* è associato un *indirizzo IP* (in realtà anche più di uno); il protocollo *DNS* consente di risolvere i *nomi di dominio*, ovvero di risalire all'*indirizzo IP* associato a un *nome*. Oltre a ciò, altri servizi offerti dal *DNS* sono:

- *Host aliasing*: è possibile associare degli alias ad un *nome di dominio*;
- *Mail server aliasing*: come per l'*host aliasing*, ma per i server mail;
- *Load distribution*: il *DNS* può essere usato per distribuire il carico delle richieste tra più server replicati, cioè con lo stesso nome. Ciò è possibile perché quando un *server DNS* riceve una richiesta, restituisce, se possibile, più di un *indirizzo IP*. L'ordine degli *indirizzi IP* dei server replicati viene cambiato ad ogni richiesta in modo da evitare che le richieste arrivino sempre ad unico server, congestionandolo;

**Struttura dei nomi di dominio** Prima di procedere oltre è bene chiarire i termini usati per descrivere i *nomi di dominio*. Il nome `drive.google.com.`, ad esempio, si struttura, a partire da destra, in:

- *Root*: è implicito in ogni nome ed è indicato con un punto (.) ;
- *TLD*: è il *dominio di primo livello* e può essere generico (e.g. `.com`, `.org`, ...) o nazionale (e.g. `.it`, `.de`, ...);
- *SLD*: è il *dominio di secondo livello* (`.google`);
- *Sottodominio*: è un sottodominio del *SLD* e serve ad identificare uno specifico *host* o gruppo di *host* all'interno del dominio (`drive`);

**Implementazione del DNS** Il *DNS* è implementato come una gerarchia di database distribuiti. Partendo dal vertice, la gerarchia di server DNS è organizzata in tre livelli:

- *Root server*: sono 13 in tutto il mondo e conoscono la posizione dei server che gestiscono i *TLD*;
- *TLD server*: gestiscono i *TLD* e conoscono la posizione degli *authoritative server* (*server di competenza*) che gestiscono i *SLD* di una determinata zona;
- *Authoritative server*: gestiscono i *SLD* e sanno risolvere tutti i nomi di dominio della loro zona di competenza;

Esistono 13 *root server* logici nel mondo, ma ognuno di essi è pesantemente ridondato per evitare interruzioni di servizio o perdite di dati. Anche i *TLD* e gli *authoritative server* sono ridondati per scongiurare congestioni, sovraccarichi e gli altri problemi visti per i *root server*.

Oltre ai server della gerarchia, ogni *ISP* ha un proprio *server DNS locale* detto *default name server* che riceve le richieste fatte dagli *host* della rete, le inoltra ai server della gerarchia e infine restituisce all'*host* interessato il risultato dell'interrogazione.

Gli *host* interrogano i *server DNS* sfruttando le funzioni fornite da un *resolver*. Si tratta di un'applicazione client, generalmente integrata nel sistema operativo, che permette di realizzare la risoluzione dei *nomi di dominio*.

**NB.** Ci si potrebbe chiedere perché non venga usato un unico server centralizzato. Il motivo è che una soluzione del genere comporterebbe enormi problematiche, quali:

- **SPOF**: il server diventerebbe un *SPOF* col risultato che un suo guasto renderebbe il servizio inutilizzabile in tutta la rete;
- *Volume di traffico*: dovendo gestire le richieste di tutta la rete il server sarebbe costantemente congestionato;
- *Distanza*: il server sarebbe stato distante dalla maggior parte degli utenti e per alcuni addirittura irraggiungibile per via del limite di *hop*;
- *Manutenzione permanente*: il server dovrebbe essere costantemente aggiornato per aggiungere e modificare i dati esistenti rendendolo inutilizzabile per la maggior parte del tempo;
- *Non scalabilità*: il server avrebbe avuto molte difficoltà ad adattarsi ai mutamenti del volume di traffico della rete;

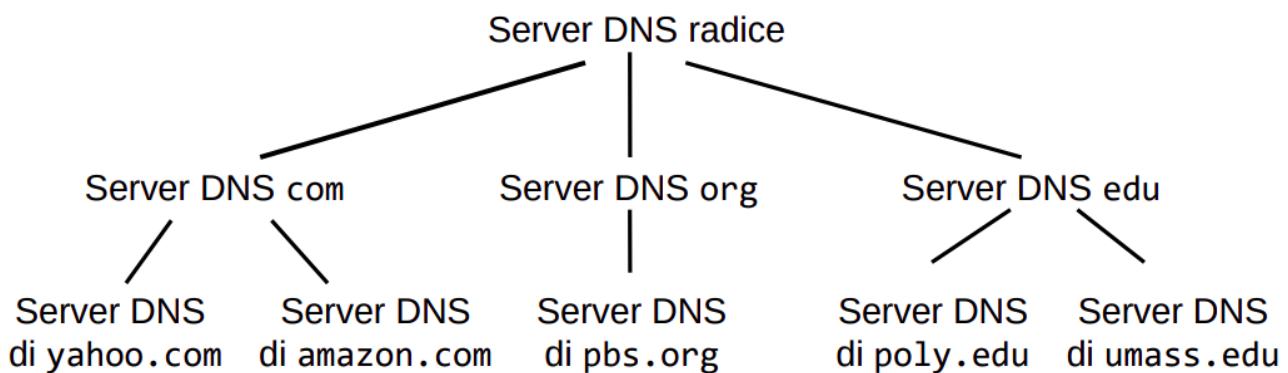


Fig. 2.18: Gerarchia dei *server DNS*

**Interrogare i server** Quando un *host* intende risolvere un *nome di dominio* contatta il proprio *default name server*. Se questo ha già la risposta all'interrogazione ricevuta, la restituisce direttamente all'*host*, altrimenti interroga la gerarchia. Per fare questo esistono due modalità: *iterativa* e *ricorsiva*.

Nella modalità *iterativa* il *server DNS locale* procede ad interrogare ogni livello della gerarchia. Partendo dal *root server*, richiede l'indirizzo del *TLD server* associato al *TLD* da risolvere, quindi interroga il *server di competenza* per il *nome* completo.

D'altra parte, nella modalità *ricorsiva*, il *server DNS locale* interroga il *root server*, questo quindi interroga il *TLD server* che a sua volta interroga l'*authoritative server*. Una volta trovata la risposta questa risale la gerarchia fino a tornare al *default name server*.

Solitamente i *server DNS* non rispondono a richieste fatte in modalità *ricorsiva* perché sono causa di congestione.

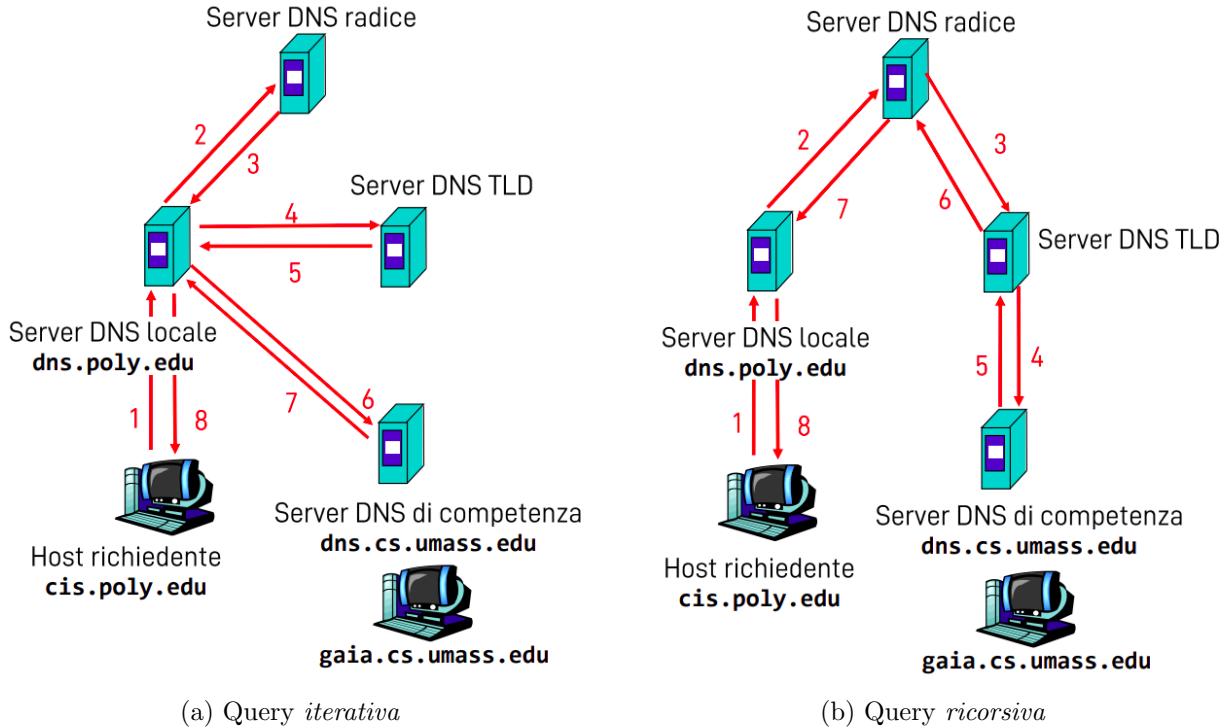


Fig. 2.19: Query *iterativa* VS query *ricorsiva*

**DNS caching** Quando un *server DNS* impara la mappatura di una di un *nome di dominio*, la salva in una cache. Queste informazioni vengono invalidate dopo un certo periodo di tempo in modo da costringere l'aggiornamento dei dati.

Soltanamente, per evitare di contattare i *root server*, i *server DNS locali* salvano gli indirizzi dei *TLD server*.

**Resource record** I *server DNS* memorizzano le informazioni in *RR* con questo formato:

$$(\text{name}, \text{value}, \text{type}, \text{ttl})$$

I valori più comuni per il campo *type* sono:

- **A:** associa il *nome di dominio* a un indirizzo *IPv4* (*name*: *nome di dominio*, *value*: *indirizzo IPv4*);
- **AAAA:** come il precedente, ma associa un indirizzo *IPv6*;
- **CNAME:** associa un *alias* al *nome canonico* (*name*: *alias*, *value*: *nome canonico*);
- **MX:** associa a un *nome di dominio* il proprio *mail server* (*name*: *nome di dominio*, *value*: nome del *mail server*);
- **NS:** associa al *nome di dominio* il nome del relativo *server di competenza* (*name*: *dominio*, *value*: nome del *server di competenza*);

**Messaggi DNS** Il protocollo *DNS* prevede due tipi di messaggi: *domande* e *risposte*. Entrambi usano lo stesso formato.



Fig. 2.20: Struttura di un messaggio *DNS*

I campi descrivono:

- *Identificazione*: un valore di 16 bit che identifica in modo univoco una domanda e la relativa risposta;
- *Flag*: i flag sono quattro:
  - *Domanda o risposta*;
  - *Richiesta di ricorsione*;
  - *Ricorsione disponibile*;
  - *Risposta di competenza*;
- *Domande*: campi per il *nome* richiesto e il tipo di domanda;
- *Risposte*: *RR* di risposta alla domanda;
- *Competenza*: *RR* dedicati per i *server di competenza*;
- *Informazioni aggiuntive*: informazioni extra che possono essere usate;

**Inserire record nel database** Quando si registra un *nome di dominio* presso un *registrar*, ovvero una società che garantisce l'unicità del *nome*, questa provvede ad inserire nel database del *server DNS* due *RR*. Nel primo viene associato il *nome di dominio* al relativo *server di competenza*, mentre nel secondo associa il nome del *server di competenza* al suo *indirizzo IP*.

# *Capitolo Nr.3*

---

## *Livello Trasporto*

---

I protocolli del *livello Trasporto* forniscono strumenti per la comunicazione logica tra processi di *host* differenti e vengono eseguiti dai sistemi terminali, cioè dal mittente per quello che riguarda l'invio dei dati e dal destinatario per la ricezione.

### **3.1 Caratteristiche dei servizi offerti**

Tipicamente, al momento dell'invio, i messaggi vengono suddivisi in *segmenti* e questi sono poi riassemblati al momento della ricezione.

I protocolli che appartengono a questo livello si appoggiano ai servizi offerti dal *livello Rete* che si occupa di gestire la comunicazione logica tra *host*. Quindi, il *livello Trasporto* è di fatto un potenziamento del *livello Rete*.

I protocolli di *trasporto* principali sono l'*UDP* e il *TCP*. Questi due servizi si differenziano per la filosofia con la quale si approcciano al trasporto dei dati: *best effort* per l'*UDP* e *affidabile* per il *TCP*. Il *TCP*, infatti, è un *protocollo connesso*, che garantisce l'arrivo di tutti i *segmenti* trasmessi permettendo al destinatario di ordinarli e occupandosi anche di gestire il controllo della congestione e del flusso. D'altra parte, l'*UDP* non fa nulla di tutto ciò, ma trasmette i *segmenti* cercando di ridurre al minimo l'overhead; scelta che ovviamente non permette di assicurare l'arrivo di tutti i dati.

**Controllo di congestione e di flusso** *Controllo della congestione e controllo del flusso* sono due concetti molto diversi che è bene chiarire subito:

---

#### **Definizione 6 - Controllo della congestione.**

*Il controllo della congestione permette di evitare che il mittente trasmetta più dati di quelli che la rete può gestire.*

---

#### **Definizione 7 - Controllo del flusso.**

*Il controllo del flusso permette di evitare che il mittente trasmetta più dati di quelli che il destinatario può gestire.*

#### **3.1.1 Multiplexing e demultiplexing**

*Multiplexing* e *demultiplexing* sono due operazioni realizzate rispettivamente dal mittente e dal destinatario.

Il *multiplexing* consiste nell'aggiunta ai dati trasmessi dalle *socket* di un header con le *PCI* del livello *Trasporto*. Le informazioni aggiunte consentono di identificare la *socket sorgente* e di *destinazione*. In fase di *demultiplexing* invece, quelle stesse *PCI* vengono usate per consegnare i *segmenti* ricevuti alla *socket* corretta.

**Funzionamento del demultiplexing** L'*host* riceve i *pacchetti IP* che nel proprio header trasportano gli *indirizzo IP* sorgente e di destinazione. Ogni *pacchetto IP* ha come *payload* un *segmento* del livello *Trasporto* nel cui header sono indicati i *numero di porta* sorgente e di destinazione. L'*host* utilizza quindi la coppia *indirizzo IP-numero di porta* per identificare la *socket* alla quale consegnare il *segmento*.



Fig. 3.1: Struttura di un *segmento TCP/UDP*

**Demultiplexing senza connessione** Nel caso del protocollo *UDP*, che è un protocollo non connesso, quando l'*host* riceve un *segmento*, legge i parametri della *socket* di destinazione (*indirizzo IP-numero di porta* di destinazione) e consegna il *segmento* a quella *socket*. *Segmenti* proveniente da processi diversi, ma con gli stessi parametri di destinazione vengono anch'essi destinati alla medesima.

**Demultiplexing con connessione** Il *TCP*, invece, per identificare una *socket*, usa anche i parametri di sorgente, quindi l'*host* ricevente utilizza tutti e quattro i parametri per inviare il *segmento* alla *socket* appropriata. Questo è conseguenza del fatto che un server può supportare più *socket TPC* contemporaneamente.

**NB.** I server *HTTP* creano *socket* differenti per ogni connessione con i client. E, addirittura, con versioni *non-persistenti* dell'*HTTP*, si ha una *socket* differente per ogni richiesta.

### 3.1.2 Numeri di porta

Quindi, la destinazione finale di un *segmento* non è un *host*, ma un processo in esecuzione su un *host*. L'interfaccia tra il livello *Applicativo* e il *Trasporto* è costituita dal già citato *numero di porta*: un valore numerico a 16 bit che identifica univocamente un processo all'interno di un *host*.

I *numeri di porta* per i servizi standardizzati sono noti e sono quindi detti *well-known*. Si tratta di valori compresi tra 0 e 1023 (incluso) che identificano un processo che fornisce un servizio standardizzato (e.g. porta 80 per l'*HTTP*, porta 25 per l'*SMTP*, ...).

I servizi non standard e le connessioni in ingresso a un client utilizzano invece *numeri di porta* con valori fino a 65535 ( $2^{16} - 1$ ) che sono decisi in modo automatico dal sistema operativo al momento della creazione di una *socket* o di instaurazione di una connessione.

**NB.** Le *porte well-known* sono anche dette *statiche*, mentre quelle assegnate dal sistema operativo sono dette *effimere*.

Una cosa importante da tenere a mente è che il *numero di porta* sorgente e di destinazione non sono quasi mai uguali. Questo perché i server restano in ascolto su una porta nota ai client, quindi quando un client contatta, o si connette con il server, lo fa su quella *porta*. Tuttavia, poiché più processi in esecuzione sullo stesso client potrebbero contattare lo stesso server, è necessario che ogni processo del client sia associato ad un *numero di porta* diverso da quello degli altri. Di conseguenza, non è possibile usare sempre lo stesso *numero di porta* scelto dal server. In relazione a ciò, è bene chiarire il concetto di *flusso di dati*:

### Definizione 8 - Flusso di dati.

*Un flusso è un gruppo di dati che appartengono alla stessa comunicazione logica.*

**NB.** Un'applicazione può aprire molteplici *connessioni* e veicolare molti *flussi*.

## 3.2 Protocollo UDP

Come già accennato, il protocollo *UDP* è orientato alla “consegna con minimo sforzo” (*best effort*), col risultato che i *segmenti UDP* potrebbero non giungere mai a destinazione o arrivare in un ordine diverso da quello di invio. È anche un protocollo non connesso quindi ogni *segmento* è gestito in modo indipendente dagli altri.

Se da una parte tutte queste caratteristiche rendono l'*UDP* un protocollo poco affidabile, lo rendono anche molto meno oneroso in termini di overhead e ritardi amministrativi: gli header sono più corti, non esiste un ritardo provocato dall'instaurazione di una connessione, non è necessario mantenere uno stato della comunicazione e, non facendo controlli di congestione, può inviare raffiche di dati.

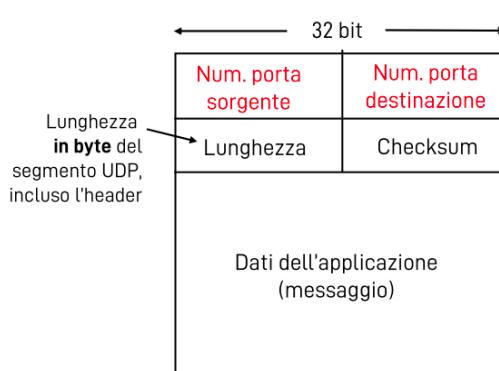


Fig. 3.2: Struttura di un *segmento UDP*

Proprio per questa sua “leggerezza”, l'*UDP* è particolarmente adatto ad applicazioni multimediali nelle quali piccole perdite sono tollerabili e che sono sensibili alla frequenza di trasferimento dei dati. Qualora fosse necessario rendere affidabile una comunicazione basata su *UDP* è necessario implementare dei controlli al livello di applicazione.

### 3.2.1 Controllo degli errori

Nell'header di un *segmento UDP* è presente un campo **Checksum** che contiene una stringa di bit che il destinatario usa per verificare la correttezza dei dati ricevuti.

In particolare, il mittente tratta l'intero *segmento* come una sequenza di parole da 16 bit quindi, somma tutte le parole e calcola il complemento a 1 del risultato. Il valore così ottenuto viene inserito nel campo **Checksum** del *segmento*.

Il ricevente, somma di nuovo tutte le parole da 16 bit del *segmento*, incluso il **Checksum**. Se il risultato di questa somma è una parola composta da 16 bit uguali a 1, allora è probabile che non ci siano errori<sup>1</sup>, altrimenti è certo che i dati sono danneggiati.

**NB.** Se quando si sommano le parole risulta un bit di riporto sul bit più significativo, questo deve essere sommato al risultato. Quindi, il riporto va sommato prima di calcolare il complemento a 1.

**NB.** Nessun sistema di rilevamento degli errori è perfetto. Per esempio, nel rilevamento di errori con bit di parità (vengono contati i bit pari a 1 e impostato a 0 un flag se il numero è pari, altrimenti dispari) si possono rilevare soltanto una quantità dispari di bit errati.

Oppure con il codice a ripetizione (la stessa stringa di bit viene trasmessa tre volte) è possibile rilevare e correggere errori se soltanto una stessa porzione di una stringa è diversa dalle altre, ma se gli errori sono molti non è più possibile essere certi che la correzione sia corretta.

## 3.3 Trasferimento dati affidabile

Esiste una classe di protocolli detti **ARQ** che si propongono l'obiettivo di recuperare i pacchetti persi. Questo tipo di protocolli usano pacchetti speciali detti **ACK** per notificare al trasmettitore la corretta ricezione di un pacchetto. In questa sezione vedremo alcuni esempi di questo tipo di protocolli.

### 3.3.1 Protocollo Stop-and-Wait

In questo tipo di protocollo, il mittente invia una **PDU** mantenendone però una copia. Quindi, imposta un timeout e attende la ricezione dell'**ACK** per quella **PDU**. Se entro lo scadere del timeout non riceve l'**ACK**, ritrasmette la **PDU**, altrimenti controlla, mediante codice *checksum* che l'**ACK** ricevuto non contenga errori e il numero di sequenza sia corretto. Se questi controlli sono verificati, allora procede all'invio della **PDU** successiva.

D'altra parte, quando il destinatario riceve una **PDU**, ne controlla *checksum* e numero di sequenza. Se sono corretti, invia l'**ACK** e passa la **SDU** ai protocolli del livello superiore, altrimenti elimina (drop) la **PDU**.

---

<sup>1</sup>Vedremo più avanti perché non se ne può avere la certezza

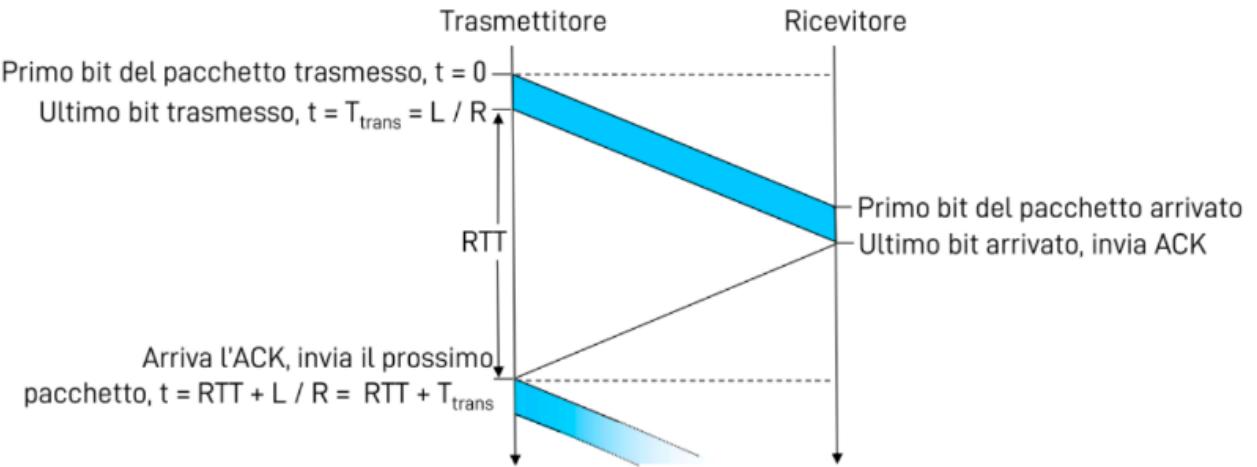


Fig. 3.3: Funzionamento protocollo *Stop-and-Wait*

**NB.** Nel calcolo del tempo  $t$  si è assunta come trascurabile la durata del pacchetto *ACK*.

**Efficienza** Se assumiamo  $R = 1Gbit/s$ ,  $RTT = 30ms$  e  $L = 8000bit$ , il **Ritardo di trasmissione** vale  $d_{trasferimento} = \frac{L}{R} = 8\mu s$ . Quindi, il *throughput* percepito a livello applicazione è:

$$\text{Throughput} = \frac{L}{d_{trasferimento} + RTT} = \frac{8000\text{bit}}{0.008\mu s + 30ms} = 33kByte/s$$

L'efficienza invece, vale:

$$\text{Efficienza} = \frac{d_{trasferimento}}{d_{trasferimento} + RTT} = 0.00027 = 0.027\%$$

Dai calcoli risulta evidente l'enorme inefficienza di questo protocollo, dovuta al fatto che per la maggior parte del tempo gli *host* restano in attesa.

**Pipelining** Con la tecnica del *pipelining* il mittente invia più pacchetti alla volta e tiene traccia del loro numero di sequenza.

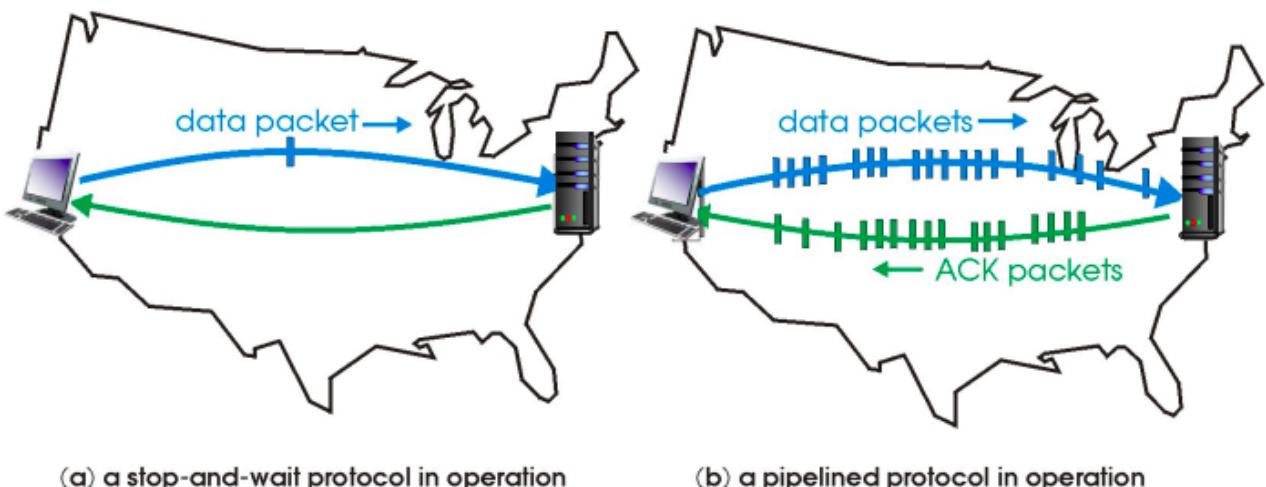


Fig. 3.4: *Stop-and-Wait* e protocollo con *pipelining*

L'utilizzo del *pipelining* permette di aumentare il *throughput* di un collegamento. Se, per esempio, si applica il *pipelining* al protocollo *Stop-and-Wait*, permettendogli quindi di inviare 3 pacchetti prima di mettersi in attesa, si triplica il *throughput*.

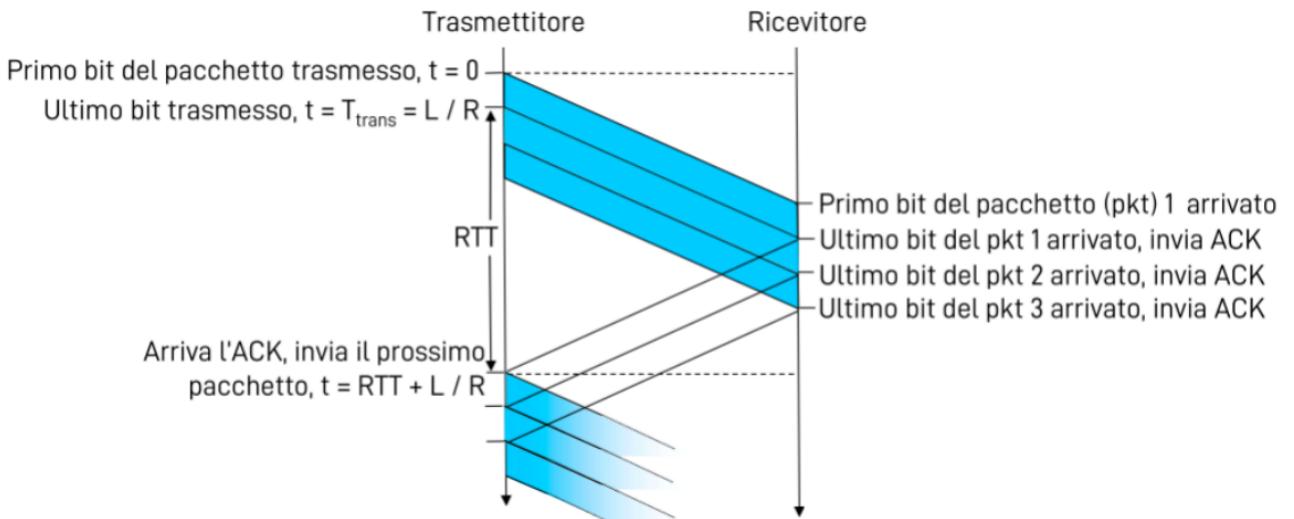


Fig. 3.5: Funzionamento protocollo *Stop-and-Wait* con *pipelining*

Ricalcolando il *throughput* si può vedere come risulti effettivamente triplicato:

$$\text{Throughput} = \frac{3L}{d_{trasferimento} + RTT} = 100k\text{Byte}/s$$

In generale, il *throughput* aumenta di tante volte quanti sono i pacchetti trasmessi prima della messa in attesa. Tuttavia, ciò vale fino a quando il tempo necessario a trasmettere quei pacchetti rimane inferiore al *RTT*.

### 3.3.2 Finestre di trasmissione e acknowledgement

**Finestre di trasmissione** Il numero di pacchetti trasmesso prima della messa in attesa, viene detto “*dimensione della finestra*”. Diamo quindi le seguenti definizioni:

#### Definizione 9 - Finestra di trasmissione - $W_T$ .

*La finestra di trasmissione, indicata in simboli come  $W_T$ , è l'insieme di PDU che il mittente può trasmettere senza avere ancora ricevuto l'ACK corrispondente. La dimensione massima della finestra è limitata dalla quantità di memoria allocata dal trasmettitore ed è indicata in simboli come  $|W_T|$ .*

#### Definizione 10 - Finestra di ricezione - $W_R$ .

*La finestra di ricezione, indicata in simboli come  $W_R$ , è l'insieme di PDU che il destinatario può ricevere e memorizzare. La dimensione massima della finestra è limitata dalla quantità di memoria allocata dal ricevitore.*

#### Definizione 11 - Puntatore low - $W_{LOW}$ .

*Il puntatore low, indicato in simboli come  $W_{LOW}$ , è un puntatore al primo pacchetto della finestra di trasmissione  $W_T$ .*

### Definizione 12 - Puntatore up - $W_{UP}$ .

Il puntatore up, indicato in simboli come  $W_{UP}$ , è un puntatore all'ultimo pacchetto già trasmesso e potrebbe non coincidere con l'ultimo pacchetto della finestra di trasmissione.

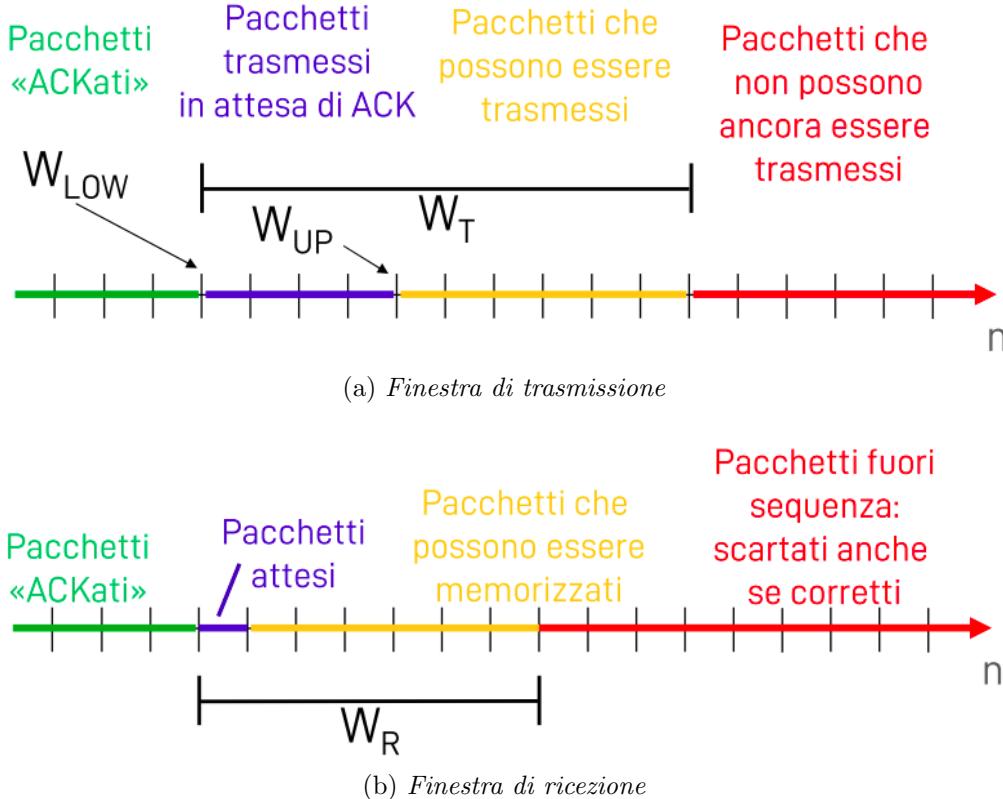


Fig. 3.6: Finestre di trasmissione e ricezione

**Pacchetti di acknowledgement** Finora abbiamo parlato dei pacchetti *ACK* senza specificare alcun tipo di dettaglio, ma in realtà esistono molteplici tipi di *acknowledgement*:

- *ACK individuale*: indica la corretta ricezione di un pacchetto specifico.  $ACK(n)$  significa che è stato ricevuto il pacchetto  $n$ ;
- *ACK cumulativo*: indica la corretta ricezione di tutti i pacchetti fino ad un certo indice.  $ACK(n)$  significa che sono stati ricevuti correttamente tutti i pacchetti fino al pacchetto  $n$  (escluso);
- *ACK negativo (NACK)*: richiede la ritrasmissione di un singolo pacchetto.  $NACK(n)$  significa che il pacchetto  $n$  deve essere ritrasmesso.

Con la tecnica del “*Piggybacking*” è possibile inserire un *ACK* in un pacchetto dati.

### 3.3.3 Protocollo Go-back-N

Nel protocollo *Go-back-N* il mittente può avere fino a  $N$  pacchetti senza *ACK* in pipeline. Il destinatario comunica la corretta ricezione dei pacchetti mediante *ACK cumulativi* e nel caso di pacchetti non ricevuti non trasmette alcun *ACK*. Fino a quando non verrà ricevuto il pacchetto

mancante, il protocollo continuerà a scartare tutti i successivi pacchetti ricevuti e, per ognuno, ritrasmetterà l'ultimo *ACK* inviato.

Il mittente ha un timer per il più vecchio pacchetto non confermato e quando scade ritrasmette tutti i pacchetti per i quali non ha ricevuto *ACK*.

Per la natura degli *ACK cumulativi*, alcuni dei pacchetti ritrasmessi potrebbero essere già stati ricevuti e scartati in precedenza. Questo protocollo soffre quindi di un'inefficienza intrinseca.

Nota: qui l'*ACKn* cumulativo riscontra i pacchetti  
fino a  $n$  inclusa per comodità di spiegazione  
(in rosso i veri *ACK* cumulativi)

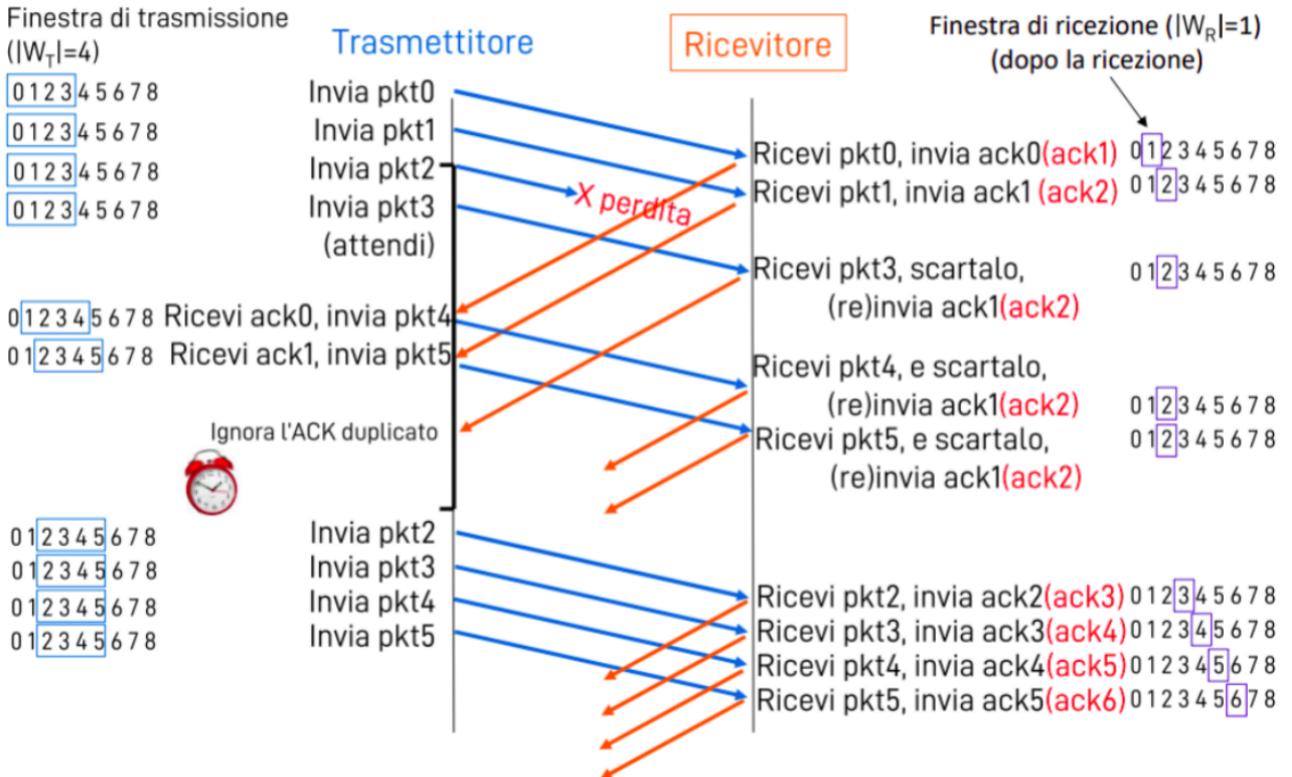


Fig. 3.7: Funzionamento protocollo *Go-back-N*

Ogni volta che il destinatario riceve un pacchetto e trasmette l'*ACK*, quel pacchetto viene trasferito all'applicazione del ricevitore.

### 3.3.4 Protocollo Selective repeat

Come nel *Go-back-N*, il mittente può avere fino a  $N$  pacchetti senza *ACK*, ma diversamente da prima, il ricevente invia *ACK individuali*. Il mittente mantiene un timer per ciascun pacchetto non confermato e quando scade ritrasmette il pacchetto associato a quel timer.

Un'altra differenza col *Go-back-N* è che nel *Selective repeat*, quando viene ricevuto un pacchetto successivo ad un pacchetto mancante, non viene scartato, ma viene salvato in un buffer.

Come prima, i pacchetti vengono consegnati all'applicazione del ricevitore ogni volta che l'*ACK* ad essi associato viene trasmesso. Nel caso di pacchetti persi, quando questi vengono ricevuti, vengono inviati all'applicazione ricevente il pacchetto appena arrivato e tutti quelli successivi che erano stati salvati nel buffer.

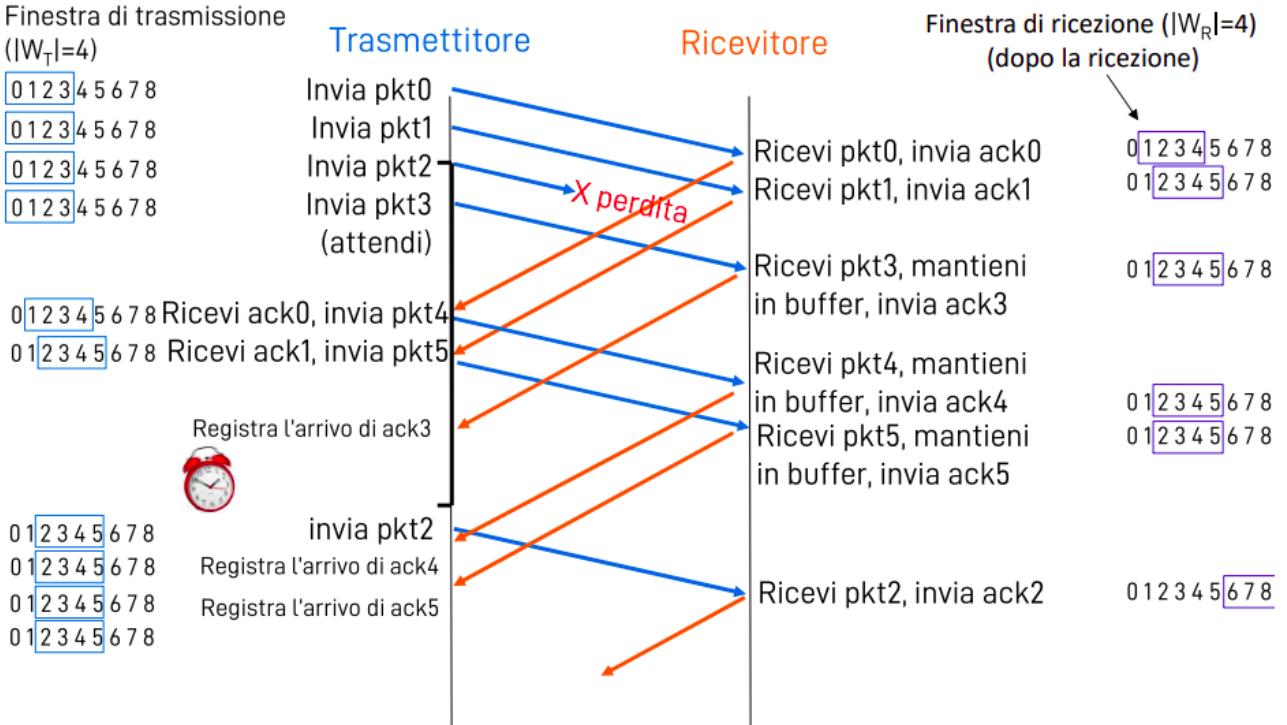


Fig. 3.8: Funzionamento protocollo *Selective repeat*

**Relazione tra numeri dimensione della finestra e numeri di sequenza** Nel protocollo *Selective repeat*, il numero di sequenza dei pacchetti è ciclico, cioè, se vengono usati  $k$  bit per codificare il numero di sequenza, si avrà un periodo, ovvero una quantità di numeri di sequenza, pari a  $2^k$ .

Fissato  $k$ , la dimensione totale delle *finestre di trasmissione* e *ricezione* deve esser minore o uguale a  $2^k$ , ovvero deve valere:

$$W_T + W_R \leq 2^k$$

Nel caso particolare in cui  $W_T = W_R$  deve valere:

$$W_T \leq 2^{k-1} = \frac{2^k}{2}$$

Il motivo per il quale deve sussistere questa condizione è che, in questo modo, i numeri di sequenza delle *finestre di trasmissione* e di *ricezione* non potranno mai sovrapporsi. La sovrapposizione va evitata perché altrimenti potrebbe accadere che il ricevente riconosca come pacchetti nuovi pacchetti che in realtà sono già stati ricevuti.

**Esempio di sovrapposizione** In questo esempio, i numeri di sequenza sono codificati su 2 bit e le *finestre* hanno entrambe dimensione 3, per cui  $W_T + W_R = 6 > 2^2 = 4$ .

Vengono inviati e correttamente ricevuti i primi tre pacchetti. Di conseguenza, la *finestra di ricezione* viene shiftata tre volte e, alla fine, rimane in attesa dei pacchetti con numeri 3, 0 e 1. Tuttavia, se accade che tutti e tre gli *acknowledgement* vengono persi durante la trasmissione, il mittente, non ricevendo nessuna conferma di ricezione, provvede ad impostare un timer per ciascuno di essi.

Allo scadere del primo timer, il mittente ritrasmette il pacchetto con numero di sequenza 0 nella *finestra di trasmissione*. Il destinatario lo riceve e, poiché nella sua *finestra di ricezione* è presente un pacchetto con numero 0, accetta il pacchetto ricevuto e trasmette un *ACK*. Il problema è che quel pacchetto in realtà era già stato ricevuto, quindi il ricevente si è ritrovato con un pacchetto duplicato.

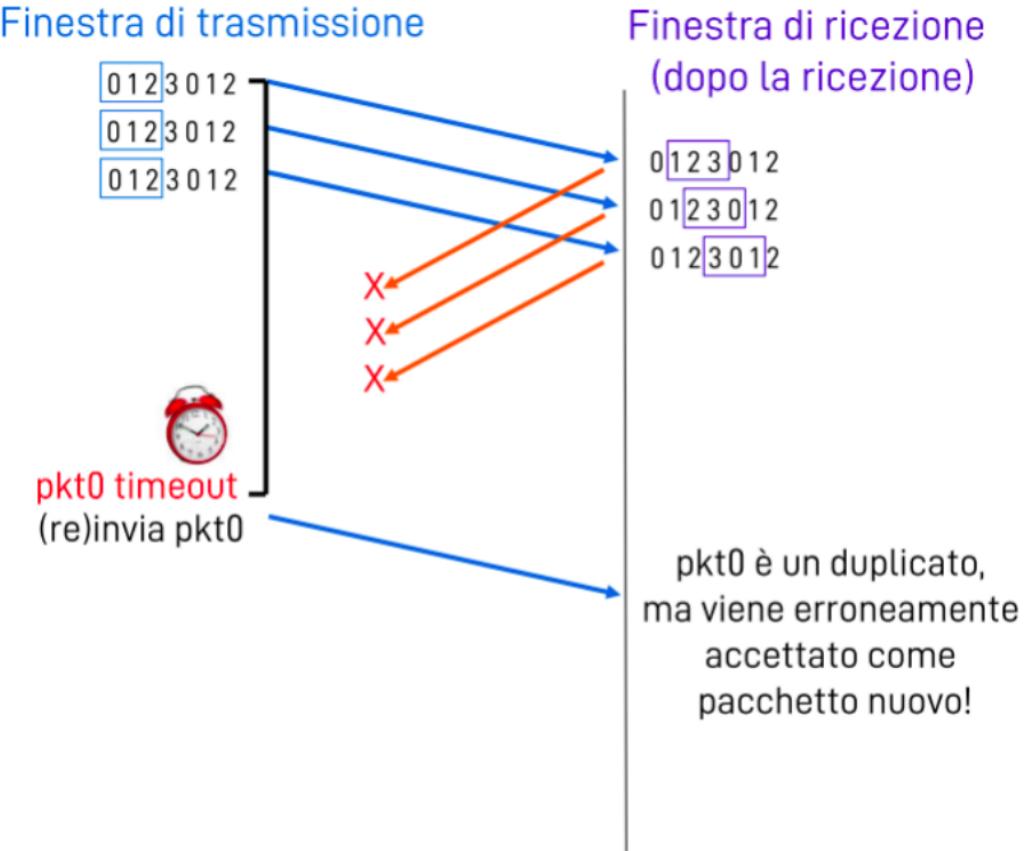


Fig. 3.9: Esempio sovrapposizione numeri di sequenza nel *Selective repeat*

**NB.** Nel protocollo *Go-back-N*, la condizione sui numeri di sequenza e sulle dimensioni delle finestre non vale, in quanto, il ricevente trasmette *ACK cumulativi* e non shifta la *finestra di ricezione* finché non riceve una sequenza completa di pacchetti. Di conseguenza, se i numeri di sequenza fossero codificati su  $k$  bit e la *finestra di trasmissione* avesse dimensione  $2^k - 1$ , nell'ipotesi in cui tutti i pacchetti trasmessi venissero ricevuti, ma andassero persi tutti gli *ACK*, se anche il mittente ritrasmettesse un pacchetto già ricevuto dal destinatario, questo lo scarterebbe poiché nella sua *finestra di ricezione* quel numero di sequenza non ci sarebbe più.

### 3.4 Protocollo TCP

Il protocollo **TCP** è un protocollo connesso che consente a due *host* di scambiarsi *segmenti* in modo affidabile e ordinato. In particolare, i *segmenti TCP* viaggiano all'interno di una connessione *full duplex* che consente trasmissioni in entrambe le direzioni.

Il *TCP* fa uso del *pipelining*, quindi esistono *finestre di trasmissione e ricezione* la cui dimensione viene stabilita in base a meccanismi di *controllo di flusso* e *congestione* e quindi varia nel tempo.

**Struttura dei segmenti TCP** Proprio per le garanzie offerte dal *TCP*, la struttura dei *segmenti* è più complicata rispetto a quanto visto con i *segmenti UDP*.

La dimensione della *finestra di ricezione* indicata nel campo *RWND*<sup>2</sup>, è rappresentata da un valore a 16 bit, che definisce il numero di byte che il destinatario può memorizzare. Di conseguenza, rappresenta anche la massima quantità di dati che può essere in transito durante un *RTT*.

<sup>2</sup>Nella figura si fa riferimento al campo Finestra di ricezione

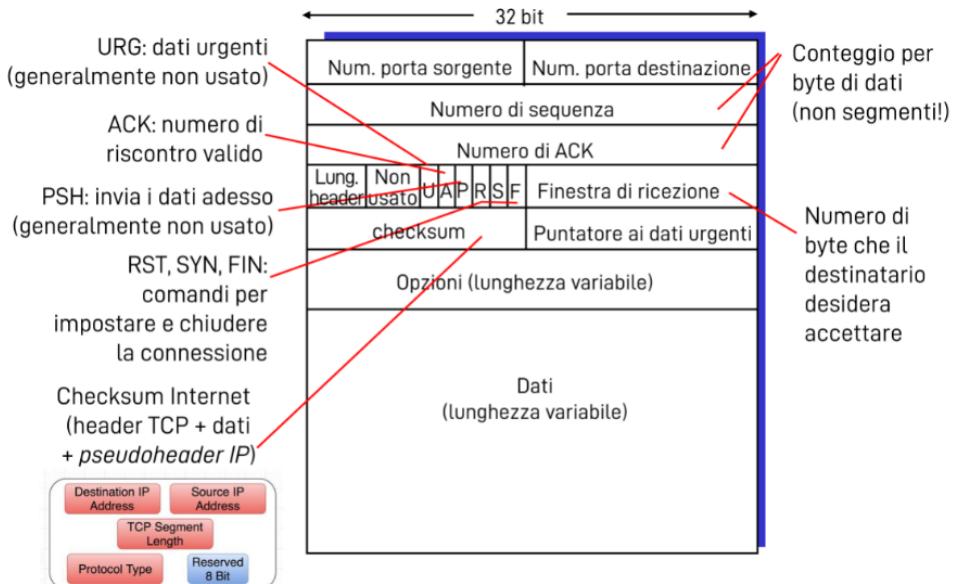


Fig. 3.10: Struttura di un *segmento TCP*

Poiché  $RWND$  è un valore a 16 bit, possono transitare contemporaneamente  $64kByte$ . Tuttavia, è possibile aumentare quel limite usando un meccanismo di *scalatura*, cioè decidendo che quel valore non rappresenta il numero di byte, ma un loro multiplo.

I campi **numero di sequenza** e **numero di ACK** indicano rispettivamente il numero del primo byte di quel segmento all'intero del *flusso di dati* e il numero di sequenza del prossimo byte atteso dall'altro lato, ricordando che il *TCP* utilizza *ACK cumulativi*.

**NB.** Per quanto riguarda la gestione dei *segmenti* fuori sequenza, il *TCP* non stabilisce una policy unica, ma dipende dall'implementazione.

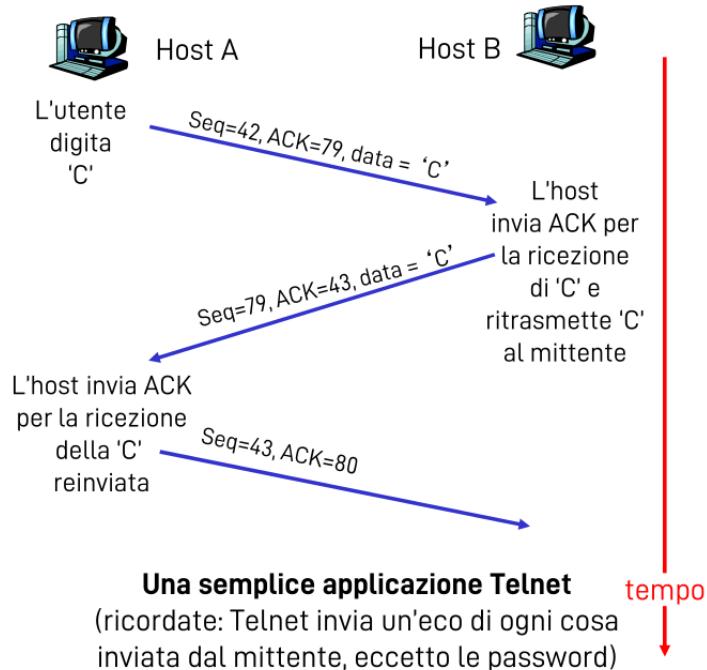


Fig. 3.11: Numeri di sequenza e ACK in una comunicazione TCP

### 3.4.1 Instaurazione di una connessione TCP

Una connessione *TCP* viene instaurata mediante un meccanismo detto *Three-way handshake*:

1. L'*host A* inizia la connessione, quindi, invia all'*host B* un *segmento* con flag **SYN** impostato a 1, porta sorgente pari ad *A*, porta destinazione *B* e numero di sequenza iniziale *x*;
2. L'*host B* riceve il *segmento* di inizializzazione e risponde con un *segmento* nel quale i flag **SYN** e **ACK** sono impostati a 1, le porte sorgente e destinazione sono rispettivamente *B* ed *A*, il numero di sequenza iniziale è *y* e il numero di **ACK** *x + 1*;
3. L'*host A* risponde inviano un *segmento ACK* con porta sorgente *A*, destinazione *B* e numero di **ACK** *y + 1*;

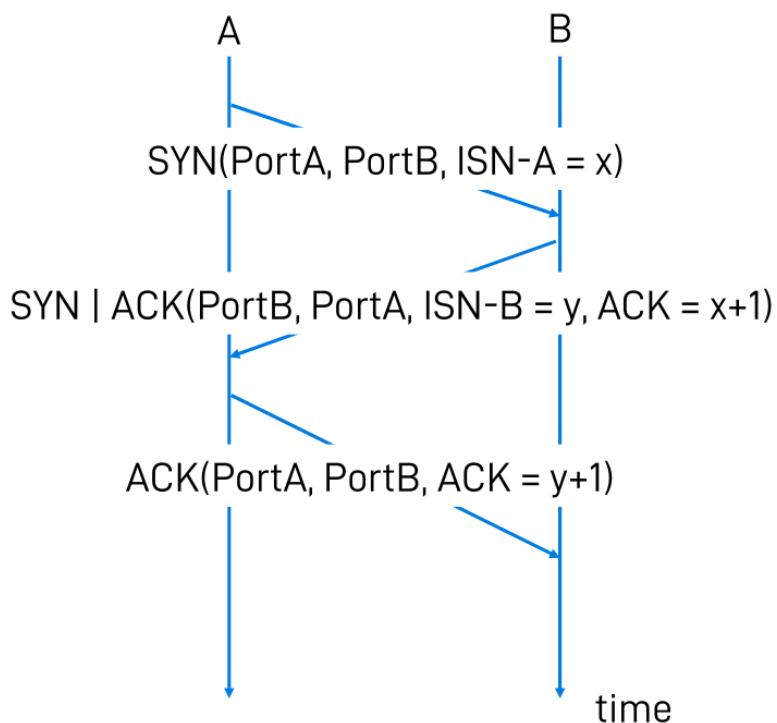


Fig. 3.12: Meccanismo di *Three-way handshake* nel *TCP*

**NB.** *ISN* sta per *Initial Sequence Number* ed è il numero di partenza dei numeri di sequenza che non iniziano necessariamente da zero.

### 3.4.2 Dimensione dei segmenti

Sebbene il protocollo *TCP* gestisca i dati organizzandoli byte, non invia mai singoli byte, ma cerca di accorparli in un unico *segmento* di dimensione massima<sup>3</sup>.

L'**MSS** dipende da un parametro del *livello* sottostante, il *livello Rete*, che si chiama **MTU**, il quale a sua volta dipende dall'*MTU* del *livello Data Link*. Tutti questi parametri dipendono anche dalle specifiche dei collegamenti attraverso i quali dovranno transitare i dati.

Comunque, l'*MSS* indica la dimensione massima del *payload*, cioè i dati trasportati dal *segmento*. Tuttavia, poiché non esistono meccanismi per la negoziazione di questo parametro, il

---

<sup>3</sup>Il *TCP* può essere costretto ad inviare singoli byte

*TCP* procede per tentativi, andando progressivamente ad incrementarlo fino a quando non viene perso qualche *segmento* o non viene ricevuta una comunicazione esplicita di incompatibilità<sup>4</sup>.

Di default, l'*MSS* viene impostata a 1460 byte (1500 byte per l'*MTU* del livello *Data Link* e 40 byte per gli *header TCP* e *IP*).

In ogni caso, esiste una dimensione minima fissata a 536 byte dovuta al fatto che il protocollo *IP* richiede una *MTU* minima di 576 byte (536 byte per il *payload* e 40 byte di *header*).

### 3.4.3 Chiusura di una connessione TCP

Poiché le connessioni *TCP* sono bidirezionali, al termine della comunicazione, vanno chiuse in entrambe le direzioni. Esistono due modalità di terminazione: una cosiddetta “gentile” e una “brusca”.

**Modalità “gentile”** L’*host* che intende terminare la connessione invia un *segmento* nel quale il flag *FIN* è impostato a 1 e il ricevitore risponde con un *ACK*. A questo punto la connessione è chiusa per metà, in quanto il primo *host* non può più trasmettere nulla, ad eccezione degli *acknowledgement*, mentre il secondo può continuare ad inviare *segmenti*.

Per terminare del tutto la connessione è necessario che anche il secondo *host* invii un *segmento* con flag *FIN* a 1.

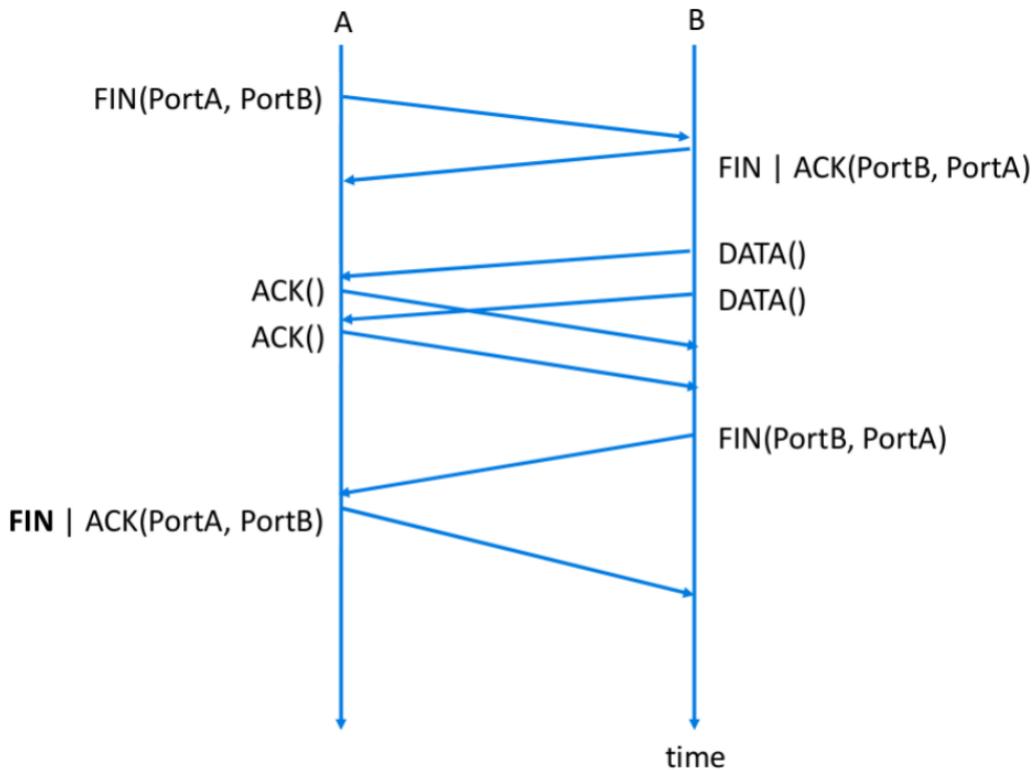


Fig. 3.13: Chiusura “gentile” di una connessione *TCP*

**Modalità “brusca”** Questa modalità viene usata per resettare connessioni non più gestibili o che si trovano in uno stato di errore (e.g. viene ricevuto un *ACK* su una connessione mai aperta). Per farlo, uno degli *host* invia un *segmento* con flag *RST* impostato a 1. Quindi, entrambi gli *host* liberano le risorse allocate dal sistema operativo per quella connessione.

**NB.** I server possono usare questa tecnica per chiudere velocemente le connessioni con i client.

---

<sup>4</sup>Non viene inviata sempre, dipende dalla configurazione dei singoli *host*

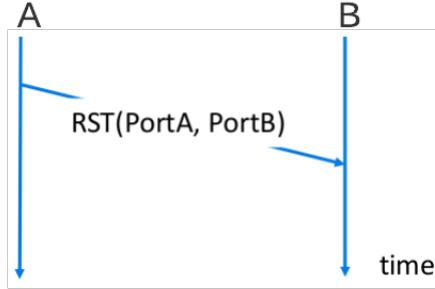


Fig. 3.14: Chiusura “brusca” di una connessione *TCP*

### 3.4.4 Stimare l’RTT e scegliere l’RTO

La scelta dell’*RTO*, ovvero la durata dei timer, influenza le prestazioni della comunicazione. Se viene scelto un valore troppo piccolo, c’è il rischio vengano effettuate delle ritrasmissioni non necessarie, mentre al contrario, un valore troppo grande rende il *TCP* troppo poco reattivo alle perdite. In ogni caso, l’*RTO* deve essere maggiore dell’*RTT*, che però è soggetto a variazioni.

#### Definizione 13 - SampleRTT.

*Il sampleRTT è il tempo misurato dalla trasmissione di un segmento alla ricezione del relativo ACK, ignorando le ritrasmissioni.*

Poiché il *sampleRTT* è diverso per ogni *segmento*, viene realizzata una stima partendo dalla media dei precedenti valori di *sampleRTT*<sup>5</sup>. Il tempo stimato viene calcolato, partendo dalla stima precedente, mediante una *media mobile esponenziale ponderata* e, in particolare, vale la seguente formula:

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT} \quad \text{per } \alpha = 0.125$$

Con questo tipo di formula, il peso dei campioni precedenti diminuisce esponenzialmente.

A questo punto, l’*RTO* può essere definito pari alla stima dell’*RTT* con in aggiunta un margine di sicurezza. Per fare ciò, bisogna innanzitutto sapere di quanto il valore stimato per l’*RTT* si discosta da quello reale:

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}| \quad \text{per } \beta = 0.25$$

Anche in questo caso, la deviazione viene calcolata a partire dal valore precedente. Il *DevRTT* costituisce il valore di partenza per la definizione del margine di sicurezza. Noto ciò, l’*RTO* è definito come:

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

**Inizializzazione dei valori** Ovviamente, all’avvio della comunicazione *EstimatedRTT* e *DevRTT* non sono definiti, quindi devono essere inizializzati a un qualche valore. Lo standard stabilisce che quando si è in possesso di una sola misura di *RTT*, si pongono  $\text{EstimatedRTT} = \text{SampleRTT}$  e  $\text{DevRTT} = \text{SampleRTT}/2$ . L’*RTO* viene invece inizializzato a un secondo. Col procedere della comunicazione i valori verranno progressivamente affinati.

<sup>5</sup>Si effettua una media pesata che da maggiore peso ai valori più recenti e progressivamente meno peso a quelli più vecchi.

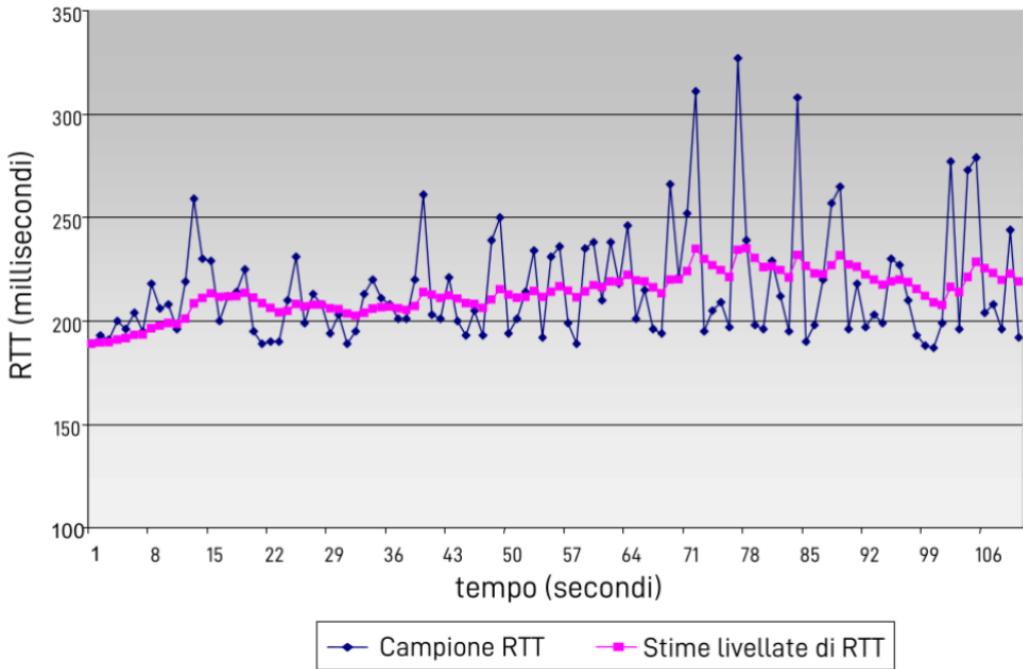


Fig. 3.15: Confronto tra *RTT* reale e stimato

### 3.4.5 Controllo di flusso

Il *Controllo del flusso*, come già accennato, è una delle funzionalità offerte dal protocollo *TCP* e permette al ricevitore di controllare la velocità di trasmissione del mittente in modo da evitare di sovraccaricarsi.

Per farlo, il ricevitore comunica al mittente la quantità di spazio ancora disponibile nel proprio buffer di ricezione e lo fa, indicando nell'*header* di ogni *segmento* il valore della *RWND*. Il mittente, quindi, limita la propria *finestra di trasmissione* al valore indicatogli dal ricevente.

Questa scelta garantisce che il buffer del destinatario non andrà mai in overflow costringendolo a scartare i *segmenti* ricevuti per mancanza di spazio.

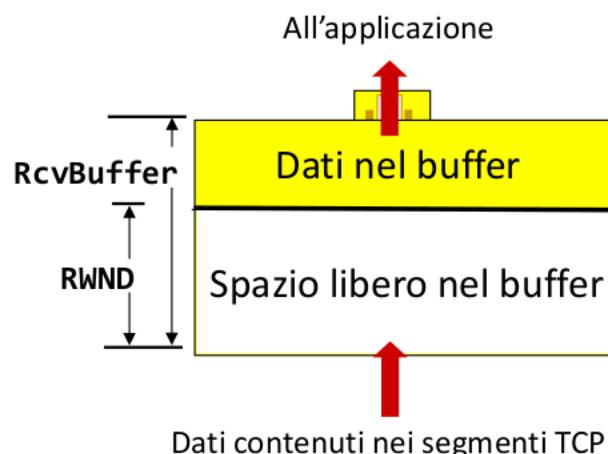


Fig. 3.16: Gestione del buffer di ricezione

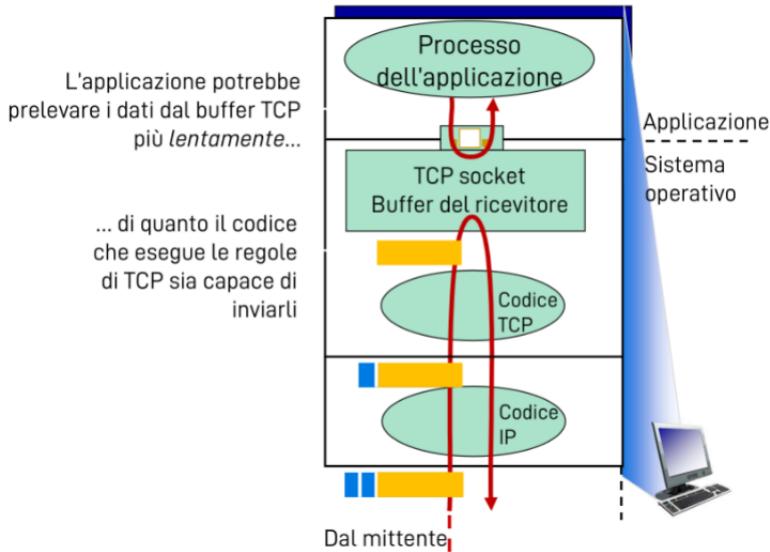


Fig. 3.17: Gestione dello *stack protocollare* al ricevitore

## 3.5 Gestione della congestione

Abbiamo già dato una definizione di **Controllo della congestione**, ma informalmente possiamo dire che la *congestione* si verifica quando troppi trasmettitori stanno inviando troppi dati e la rete non è in grado di gestire tutto quel traffico. Una rete congestionata comporta la perdita di *pacchetti*, dovuta all'overflow dei buffer nei *router*, e lunghi ritardi nel trasferimento, dovuti all'accodamento dei *pacchetti* nei buffer.

### 3.5.1 Modelli per sistemi a coda

Un *sistema a coda* comprende una “fila d’attesa”, realizzata solitamente con una *coda*, e un *server*. I parametri presi in considerazione sono due:

1. *Tasso di arrivo*  $\lambda$ : è il numero medio di *pacchetti*, o in generale di *unità di lavoro*, che entrano nella *coda* per unità di tempo;
2. *Tasso di servizio*  $\mu$ : è il tempo medio richiesto dal *server* per trasmettere un *pacchetto*, o in generale per concludere un lavoro;

Gli *arrivi* e i *tempi di servizio* sono distribuiti secondo distribuzioni statistiche. Ad esempio, nei *sistemi a coda* di tipo  $M/M/1$ <sup>6</sup>, gli *arrivi* sono distribuiti come  $\exp\left(\frac{1}{\lambda}\right)$ , i *tempi di servizio* come  $\exp\left(\frac{1}{\mu}\right)$  e c’è un solo *server* a ricevere i *pacchetti*.



Fig. 3.18: Schema di un *sistema a coda*

<sup>6</sup> $M$  sta per *Markovian* e si riferisce alla distribuzione esponenziale.

Le proprietà dei *sistemi a coda* possono essere calcolate analiticamente. Ad esempio, nei *sistemi M/M/1* si ha che:

- *Carico del server*:  $\rho = \frac{\lambda}{\mu}$ ;
- *Lunghezza media della coda*:  $L = \frac{\rho^2}{1-\rho}$  per  $\rho < 1$ ;
- *Probabilità che in un qualunque momento ci siano n pacchetti in coda*:  $\pi_n = (1 - \rho) \cdot \rho^n$ ;

Poiché  $\lambda$  e  $\mu$  sono medie statistiche e non delle costanti, i dati reali non coincidono sempre con quelli stimati. Inoltre, la probabilità che i *pacchetti* si accodino, anche se molto piccola, non è mai nulla.

### 3.5.2 Cause della congestione

Vediamo ora come si manifesta la *congestione* in alcuni scenari.

**Scenario 1** Si supponga di avere due trasmettitori e due ricevitori e che i dati debbano passare per un *router* con un buffer infinito. Se la *capacità del link* di uscita è  $R$  e non ci sono ritrasmissioni, il *throughput* massimo è  $\frac{R}{2}$ . Inoltre, se il *tasso*  $\lambda_{in}$  si avvicina a  $\frac{R}{2}$  si avrà a che fare con lunghi ritardi.

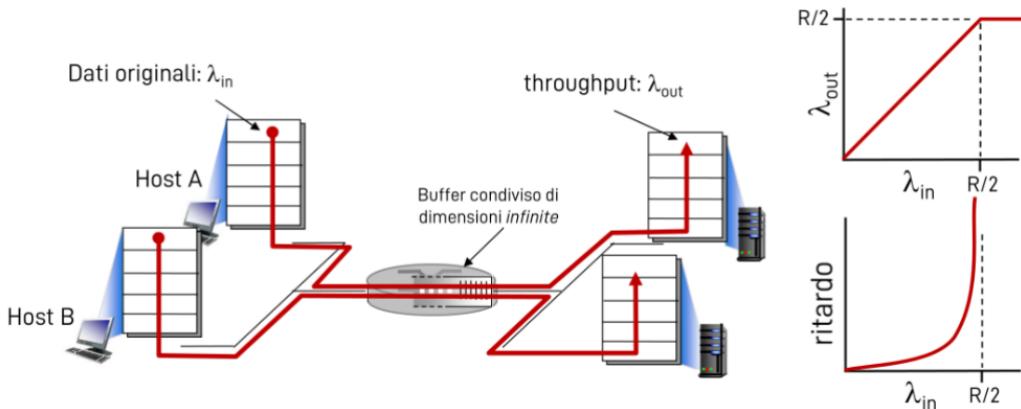


Fig. 3.19: *Congestione* - scenario 1

**Scenario 2** Si supponga di avere un *router* con un buffer di dimensione finita e che il mittente ritrasmetta i pacchetti finiti in timeout. Si indichi con  $\lambda_{in}$  il *tasso di arrivo* dall'applicazione al mittente e con  $\lambda_{out}$  il *tasso* percepito dall'applicazione del destinatario.

Poiché il mittente ritrasmette i pacchetti persi, vengono inviati più dati di quelli che si invierebbero se non ci fossero ritrasmissioni, di conseguenza se si calcola il *tasso di arrivo* al *livello Trasporto*, e lo si indica con  $\lambda'_{in}$ , si ha che  $\lambda'_{in} > \lambda_{in}$ , in quanto il *livello Trasporto* include anche tutte le ritrasmissioni.

In questo scenario possiamo distinguere due casi:

- *Caso ideale*: si ha una perfetta conoscenza della situazione e quindi il mittente può inviare dati solo quando sa che nel buffer del *router* c'è spazio per riceverli;
- *Caso realistico*: si ha una conoscenza limitata della rete e quando scatta un timeout, il mittente ritrasmette il *pacchetto* associato, ma potrebbe accadere che vengano consegnate entrambe le copie causando un dimezzamento del *throughput*;

Nel *caso realistico* è possibile ipotizzare cosa accadrebbe se il mittente ritrasmettesse i *pacchetti* solo quando è sicuro che questi siano andati persi, per esempio impostando timer piuttosto lunghi. In questo modo si potrebbe migliorare il *throughput*, tuttavia, è comunque possibile che a causa della *congestione* anche i pacchetti ritrasmessi vengano persi.

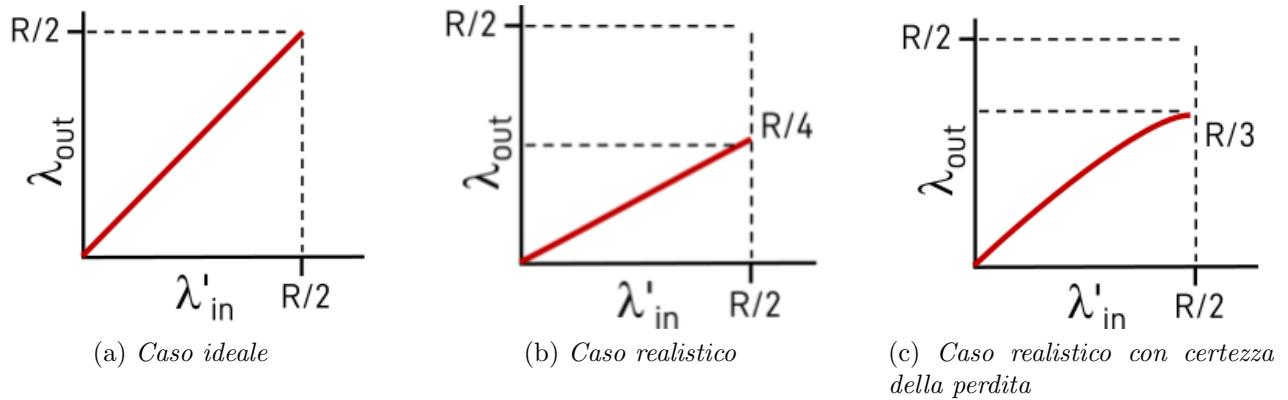


Fig. 3.20: *Congestione* - scenario 2

**Scenario 3** Si supponga che siano possibili più percorsi e che i *pacchetti* debbano passare attraverso più di un *router*. Se accade che  $\lambda'_{in}$  e  $\lambda_{in}$  aumentano, la maggior parte, se non tutti, i *pacchetti* trasmessi vengono scartati portando  $\lambda_{out}$  a 0.

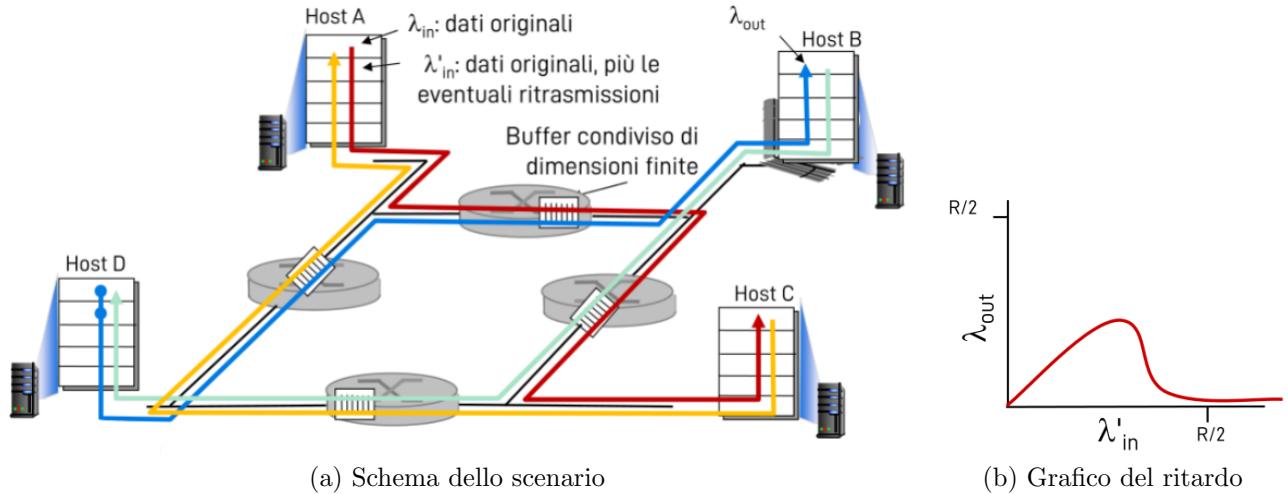


Fig. 3.21: *Congestione* - scenario 3

**NB.** Ogni volta che un *pacchetto* viene perso, tutte le risorse usate per portarlo fino a quel punto risultano sprecate.

In conclusione, possiamo affermare che con buffer infiniti non ci sono perdite, ma se il *tasso di invio* si avvicina troppo al *tasso di servizio* i ritardi si allungano di molto. Se i ritardi sono dovuti alla *congestione* provocata dalle ritrasmissioni fatte al termine di un timeout, si ha uno spreco di risorse per via di ritrasmissioni inutili. Infine, se la *congestione* porta alla perdita dei *pacchetti* si è costretti a pagare il costo della loro ritrasmissione e lo sforzo richiesto per ridurre il tempo  $\lambda'_{in}$  può provocare un crollo del *throughput* vanificando così gli sforzi fatti.

## 3.6 Controllo della congestione in TCP

Il *TCP* usa tecniche di *controllo della congestione* per adattare il *tasso di trasmissione* alle condizioni della rete ed evitare di congestionarla. Per fare ciò esistono diversi approcci:

- *Controllo di congestione end-to-end*: il livello di *congestione* viene stimato tenendo traccia dei *segmenti* persi e dei ritardi;
- *Controllo di congestione assistito dalla rete*: i *router* forniscono dei feedback agli *host* sullo stato della rete (e.g. viene impostato un bit per indicare la *congestione*);

Non esiste un unico protocollo per il *controllo della congestione*, ma implementazioni diverse del *TCP* adottano soluzioni distinte.

### 3.6.1 Protocollo AIMD

Nel protocollo *AIMD* il mittente aumenta progressivamente il *tasso di trasmissione*, cosa che coincide con l'aumento della dimensione della *finestra di trasmissione*, cercando di occupare tutta la banda disponibile. Quando viene rilevata una perdita, la dimensione della *finestra* viene ridotta.

Questo protocollo è organizzato in due fasi:

- *Additive increase*: ad ogni *RTT*, fino a quando non viene rilevata una perdita, la *finestra di trasmissione* viene aumentata di una *MSS*;
- *Multiplicative decrease*: quando viene rilevata una perdita, la *finestra di trasmissione* viene dimezzata;

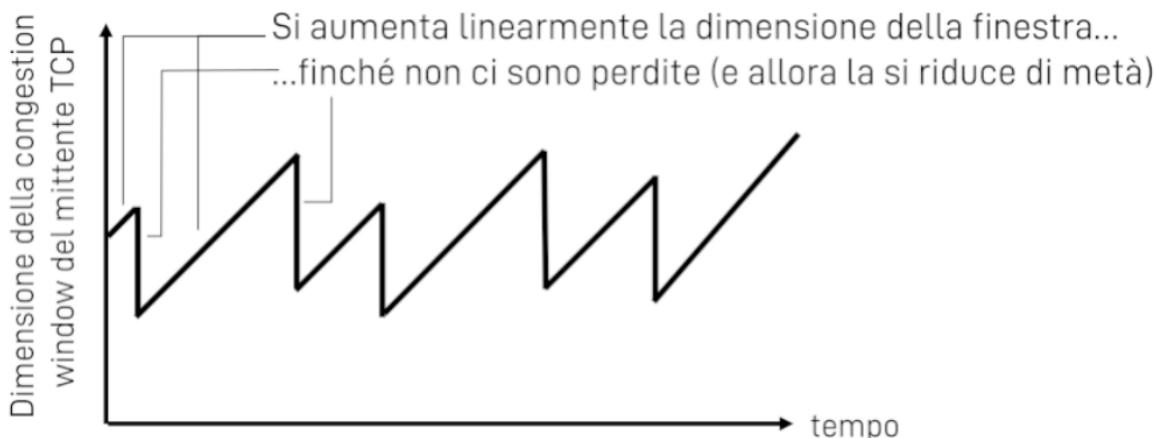


Fig. 3.22: Dimensione della *finestra di trasmissione* in funzione del tempo

**NB.** In realtà, più che di *finestra di trasmissione* sarebbe meglio parlare di *finestra di congestione*.

---

#### Definizione 14 - Finestra di congestione.

*La finestra di congestione è l'insieme delle PDU che possono essere inviate nella rete e la sua dimensione è definita in base alla quantità massima di dati che il mittente pensa di poter inviare senza sovraccaricare la rete.*

La dimensione della *CWND* è soggetta a variazioni durante la comunicazione in quanto il valore viene deciso dinamicamente dall'algoritmo di *controllo della congestione*. Un'altra cosa che cambia durante la comunicazione è la *finestra di trasmissione*.

### Definizione 15 - Finestra di trasmissione.

*La finestra di trasmissione è l'insieme delle PDU che possono essere inviate in rete senza saturare il ricevitore e la sua dimensione è definita come:*

$$|W_T| = \min(\text{CWND}, \text{RWND}) = \min(\text{CWND}, |W_R|)$$

**Fairness** Il protocollo *AIMD* permette di ottenere una distribuzione equa delle risorse di rete. Ciò significa che se  $k$  sessioni *TCP* si dividono lo stesso collegamento di banda  $R$  che fa da collo di bottiglia, ogni sessione dovrebbe percepire la stessa banda  $\frac{R}{k}$ . Questa caratteristica è detta *fairness*.

Ad esempio, se due connessioni stanno condividendo lo stesso collegamento, nel corso del tempo, il processo di *additive increase* fa aumentare la *banda* di entrambe le connessioni in modo lineare. D'altra parte, quando si passa alla fase di *multiplicative decrease*, la *banda* viene ridotta in modo proporzionale a quella che si stava utilizzando. Ciò significa che la diminuzione sarà maggiore per la connessione che stava occupando la fetta maggiore di *banda*.

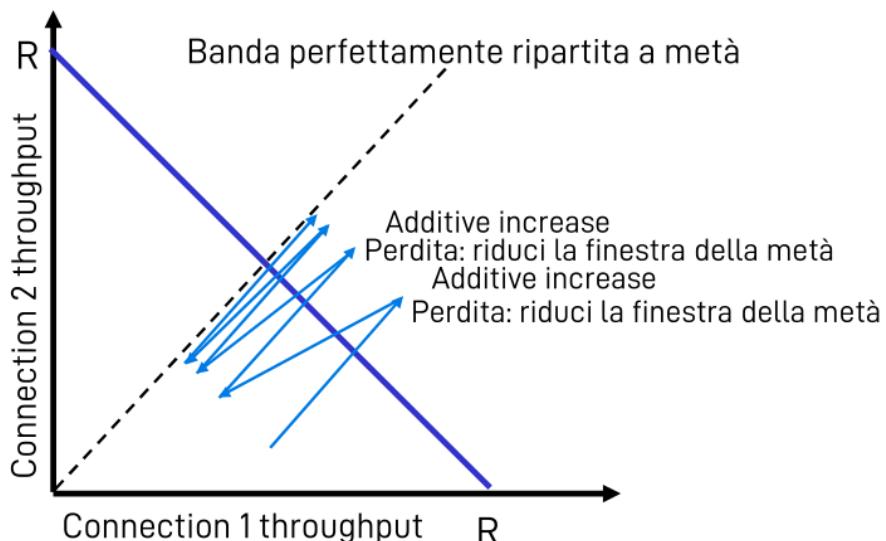


Fig. 3.23: Raggiungimento dello stato di *fairness*

**Slow Start e Congestion Avoidance** L'algoritmo *AIMD* suddivide la comunicazione in due fasi che si alternano:

- *Slow Start*: il mittente trasmette inizialmente un solo *segmento* e, per ogni *ACK* valido ricevuto, incrementa di una *MSS* la *CWND*, che quindi aumenta esponenzialmente la propria dimensione.

Quando la *CWND* raggiunge un valore soglia *SSTHRESH* l'algoritmo passa in regime di *Congestion Avoidance*;

- *Congestion Avoidance*: per ogni *ACK* valido ricevuto, la *CWND* aumenta di  $\frac{\text{MSS}}{\text{CWND}}$  byte ovvero  $\frac{1}{\text{CWND}}$  *segmenti*. Ciò significa che per ogni *RTT*, se vengono ricevuti tutti gli *ACK* attesi (sono tanti quanto è la *CWND*), la *CWND* viene incrementata di una *MSS* (un *segmento*). In questa fase l'incremento della *CWND* è lineare.

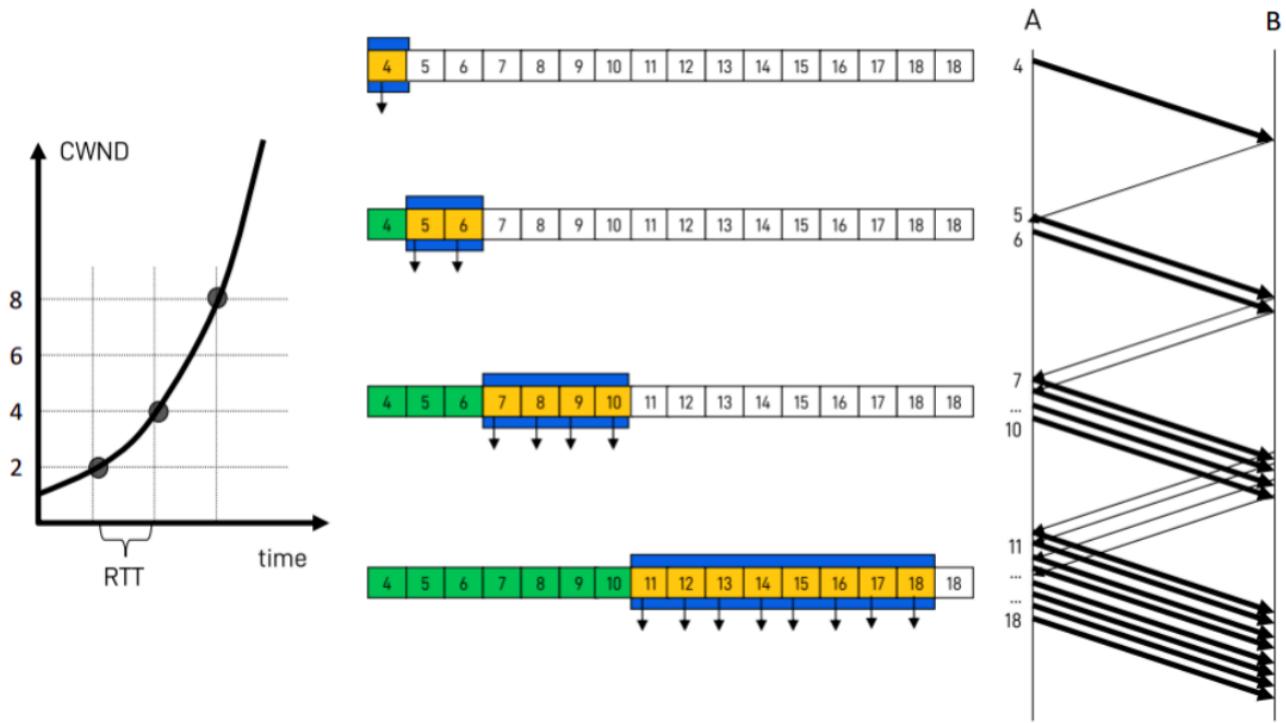


Fig. 3.24: Crescita della *CWND* in regime di *slow start*

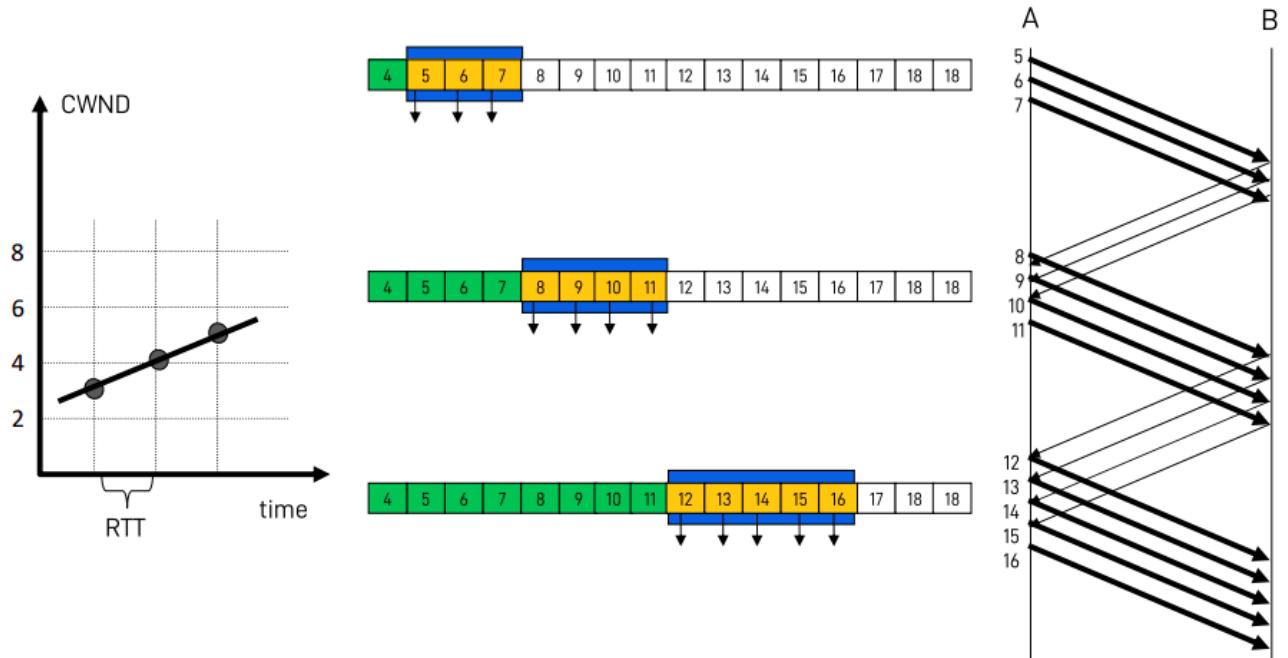


Fig. 3.25: Crescita della *CWND* in regime di *congestion avoidance*

Lavorando con l'*AIMD* è possibile modificare le prestazioni della comunicazione andando ad agire su quattro parametri:

- *CWND*: è possibile aumentarla per trasmettere di più, ma ciò rende più probabile andare a congestionare la rete;
- *SSTHRESH*: diminuendola si conclude riduce la fase di crescita esponenziale e si passa prima in *congestion avoidance* favorendo un approccio più prudente;
- **RTO**: aumentandolo si aumenta il tempo di attesa di ciascun *ACK*;

- $W_{LOW}$  e  $W_{UP}$ : è possibile ritardarne lo spostamento in modo, per esempio, di mantenere in memoria *segmenti* per i quali non si è ancora ricevuto l'*ACK*;

**Funzionamento dell'algoritmo** L'algoritmo parte in regime di *Slow Start* e inizializza la *CWND* e la *SSTHRESH*, rispettivamente, a 1 *MSS* e a *RWND*<sup>7</sup>. Essendo in *Slow Start*, per ogni *ACK* valido ricevuto la *CWND* viene incrementata di una *MSS* e il puntatore  $W_{LOW}$  viene spostato al primo byte (o *segmento*) non confermato. Se  $CWND \geq SSTHRESH$  l'algoritmo passa in *Congestion Avoidance*, altrimenti continua ad inviare pacchetti.

Quando l'algoritmo si trova in regime di *Congestion Avoidance*, per ogni *ACK* valido ricevuto la *CWND* viene incrementata di  $MSS \cdot \frac{MSS}{CWND}$  byte e il puntatore  $W_{LOW}$  viene spostato al primo *segmento* non validato. Fatto ciò, vengono trasmessi altri *segmenti*.

In entrambe le fasi, quando per un *segmento* non si riceve nessun *ACK*, ovvero quando scatta il timeout associato al *segmento*, la soglia di *SSTHRESH* viene abbassata a  $\max(\frac{CWND}{2}, 2)$ , viene raddoppiato l'*RTO*, viene reimpostata ad 1 *MSS* la *CWND* e, quindi, viene ritrasmesso il *segmento* perso.

**NB.** L'utilizzo del solo *RTO* non è efficiente in quanto costringe ad attendere lo scadere del timer prima di procedere con il rinvio.

**Fast retransmit e fast recovery** Un modo migliore di gestire le perdite può essere realizzato sfruttando la natura degli *ACK*, che in *TCP* sono *cumulativi*. In particolare, quando viene perso un *segmento*, alla ricezione dei successivi, il destinatario non risponde con gli *ACK* corrispondenti, ma ripropone l'ultimo *ACK* mandato prima della perdita.

Quindi, il mittente riceve degli *ACK* duplicati sulla base dei quali può trarre delle conclusioni. Il *segmento* potrebbe semplicemente essere in ritardo per cui non è necessaria una ritrasmissione, oppure potrebbe essere andato perso.

La tecnica del *Fast Retransmit* prevede che quando viene ricevuto il terzo *ACK* duplicato, il *segmento* indicato dall'*ACK* venga ritrasmesso (*fast retransmit*). Quando ciò accade l'algoritmo entra nella fase di *fast recovery*.

All'ingresso in *fast recovery* accadono le seguenti cose:

- La soglia *SSTHRESH* viene impostata a  $\frac{CWND}{2}$ ;
- Il puntatore  $W_{LOW}$  non viene spostato, ma rimane fermo sul primo *segmento* non validato;
- Il valore del puntatore  $W_{UP}$  viene assegnato alla variabile *RECOVER* ed indica l'ultimo *segmento* trasmesso prima dell'ingresso in *fast recovery*;
- La *CWND* viene impostata a *SSTHRESH* + 3 *MSS*.

A questo punto, per ogni *ACK* duplicato ricevuto, la *CWND* viene incrementata di una *MSS* e, se la *CWND* lo permette, il mittente continua a trasmettere, ma il puntatore  $W_{LOW}$  non viene spostato.

Alla ricezione di un *ACK* valido che includa un riscontro per il *segmento* *RECOVER*,  $W_{LOW}$  viene impostato al primo *segmento* non validato, *CWND* viene abbassato a *SSTHRESH* e l'algoritmo passa in *Congestion Avoidance*. Mentre, se viene ricevuto un *ACK* cosiddetto “parziale”, cioè che conferma un *segmento* precedente a quello di *RECOVER*, viene ritrasmesso il primo *segmento* non validato,  $W_{LOW}$  viene impostato a quel *segmento* e la *CWND* viene prima incrementata di 1 e poi ridotta del numero di *segmenti* validati da quando si è entrati in *fast recovery*. In pratica vale:  $CWND = CWND - \text{numero segmenti validati} + 1$ .

---

<sup>7</sup>In alcune implementazioni viene impostata a  $\frac{RWND}{2}$ .

### Esempio 1 - Fast retransmit e fast recovery.

Si supponga di essere in regime di Congestion Avoidance, che  $CWND = 5$  e  $W_T = [10, \dots, 14]$ .

Come si comporta il mittente se viene perso il segmento 11?

Il mittente invia tutti i segmenti della propria  $W_T$  e si mette in attesa degli ACK. Quando il destinatario riceve il segmento 10, risponde con ACK11, il segmento 11 viene perso, quindi alla ricezione dei segmenti 12, 13 e 14 il destinatario risponde sempre con ACK11.

Il mittente inizia a ricevere gli ACK. All'arrivo del primo ACK11, incrementa  $W_{LOW}$  di 1, quindi  $W_T$  diventa  $[11, \dots, 15]$ . Il mittente può quindi trasmettere il segmento 15.

Successivamente vengono ricevuti gli ACK11 duplicati e, all'arrivo del terzo duplicato, cioè dell'ACK associato al segmento 14, il mittente ritrasmette il segmento 11 ed entra in Fast Recovery:

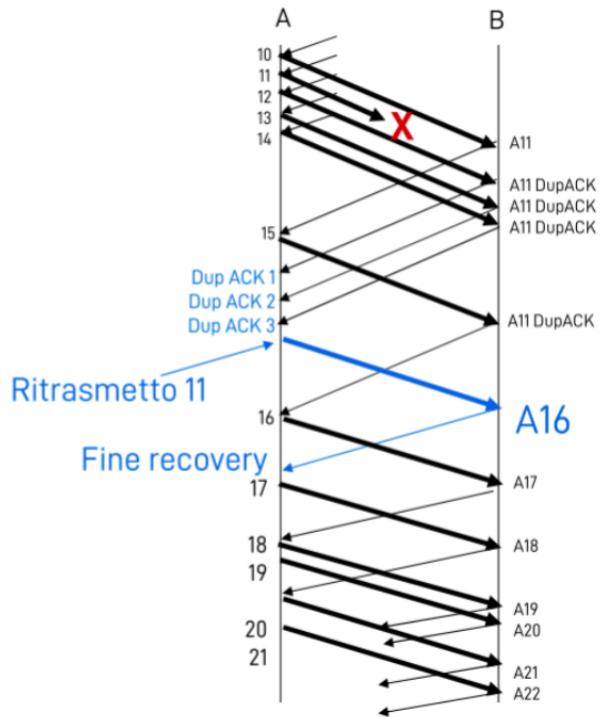
- $RECOVER = 14$ ;
- $SSTHRESH = \frac{CWND}{2} = 2$ ;
- $CWND = SSTHRESH + 3 = 5$ ;
- $W_T = [11, \dots, 15]$ ;

Quando il destinatario riceve il segmento 15 risponde con un quarto ACK11 che fa aumentare di una MSS la CWND del mittente, il quale può quindi procedere a trasmettere il segmento 16.

Contemporaneamente, il destinatario ha ricevuto il segmento 11 e quindi può confermare anche la ricezione di tutti i segmenti successivi già ricevuti: risponde con ACK16.

Il mittente riceve ACK16 e poiché include RECOVER, passa in Congestion Avoidance e imposta CWND a SSTHRESH, cioè a 2.  $W_{LOW}$  viene spostato a 16, quindi  $W_T = [16, 17]$ .

**NB.** Nella figura sottostante gli istanti nei quali il TCP si trova in fast recovery non sono mostrati.



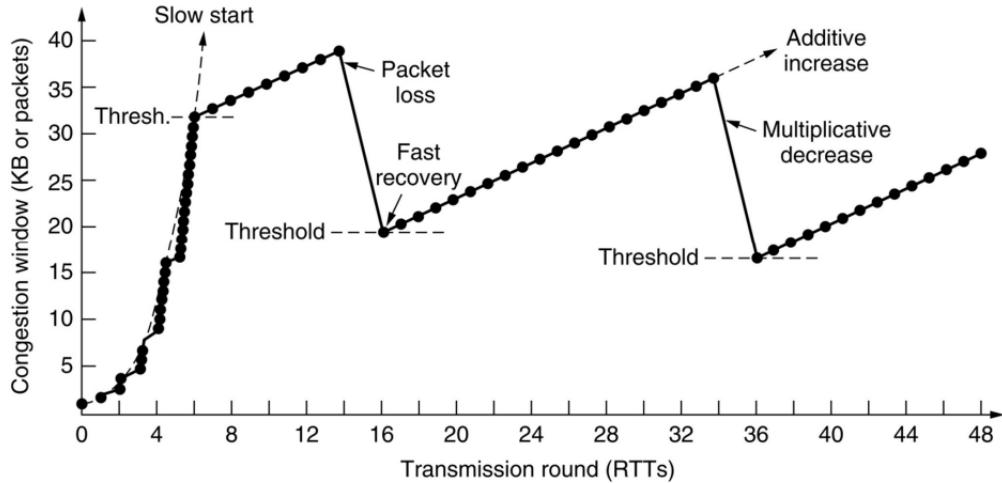


Fig. 3.26: *TCP con fast retransmit e fast recovery*

**Macchina a stati dell'algoritmo AIMD** L'algoritmo *AIMD* può essere efficacemente descritto come una macchina a stati:

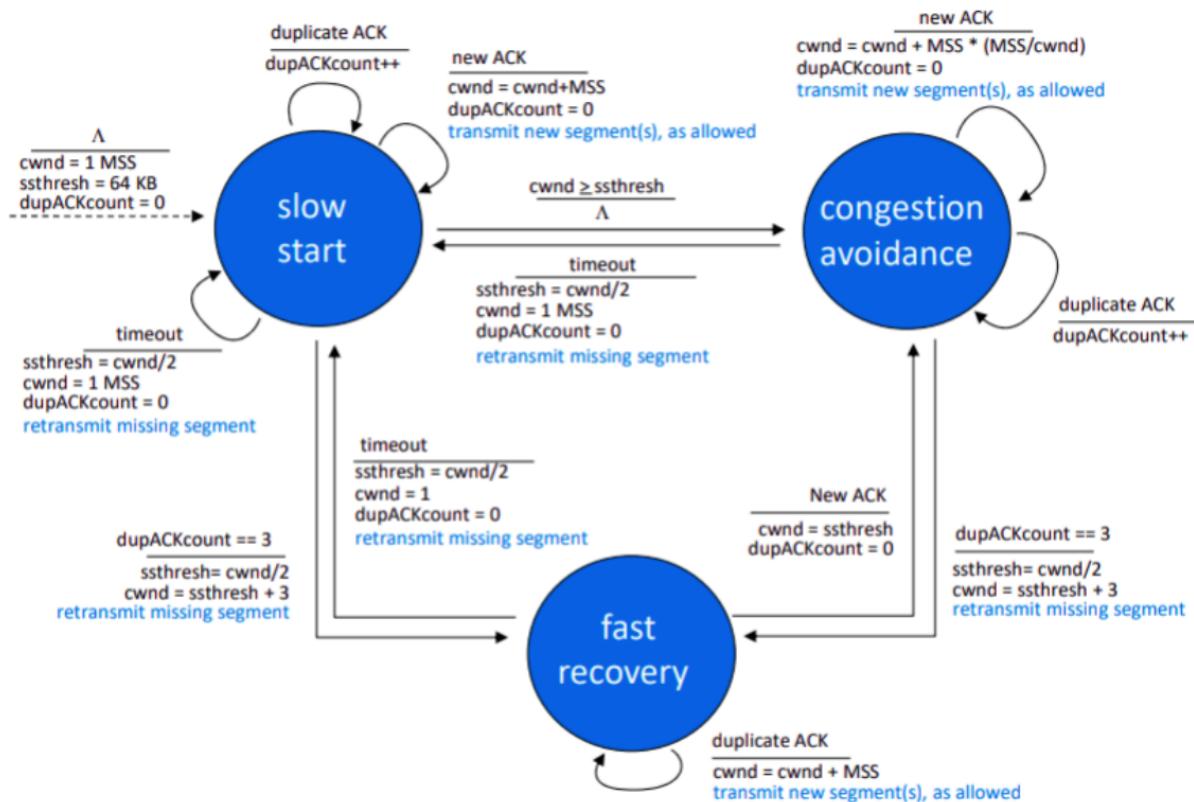


Fig. 3.27: Macchina a stati dell'algoritmo *AIMD*

**Calcolo del throughput** La seguente formula:

$$Thr(RTT, p) < \frac{MSS}{RTT} \cdot \frac{1}{\sqrt{p}}$$

permette di calcolare il limite superiore al *throughput* del *TCP*. Il parametro  $p$  indica la probabilità di perdere un *segmento*.

**Problemi di fairness** Il protocollo *TPC*, in realtà, non risolve del tutto i problemi di *fairness* e i motivi sono principalmente due:

1. Le applicazioni multimediali, o in generale le applicazioni che tollerano delle perdite, utilizzando il protocollo *UDP* per la trasmissione dei *segmenti*. Poiché il protocollo *UDP* non ha un meccanismo di *controllo della congestione*, è probabile che le comunicazioni *TCP* diminuiscano il *throughput*;
2. Le applicazioni possono aprire più connessioni *TCP* in parallelo tra due *host* e quindi, se due applicazioni condividono lo stesso collegamento, ma una ha aperto una sola connessione, mentre l'altra più di una, il *throughput* della prima risulterà inferiore al *throughput* complessivo della seconda.

## 3.7 Versioni moderne del TCP

La versione di *TCP* che abbiamo visto e che usava *AIMD* come protocollo per il *controllo della congestione* era *loss-based*, ovvero decideva la frequenza di invio dei *segmenti* basandosi soltanto su quelli persi. Tuttavia, esistono algoritmi più moderni che cercano di basarsi sul livello di *congestione* effettivo della rete.

### 3.7.1 Protocollo CUBIC

Il protocollo *CUBIC* gestisce la *congestione* facendo variare la dimensione della *finestra di congestione* basandosi su una funzione cubica del tempo. Ciò permette di ottenere una migliore scalabilità e stabilità su reti veloci e a lunga distanza.

In particolare, *CUBIC* utilizza entrambi i profili, concavo e convesso, di una funzione cubica per aumentare la dimensione della *finestra di congestione* e, in reti con *RTT* brevi, è progettato per comportarsi come il *TCP* standard. Questa adattabilità è realizzata impostando opportunamente il *coefficiente di diminuzione* della *finestra di congestione* che è 0.5 nel *TCP* standard (la *finestra* viene dimezzata) e 0.7 nel *CUBIC*.

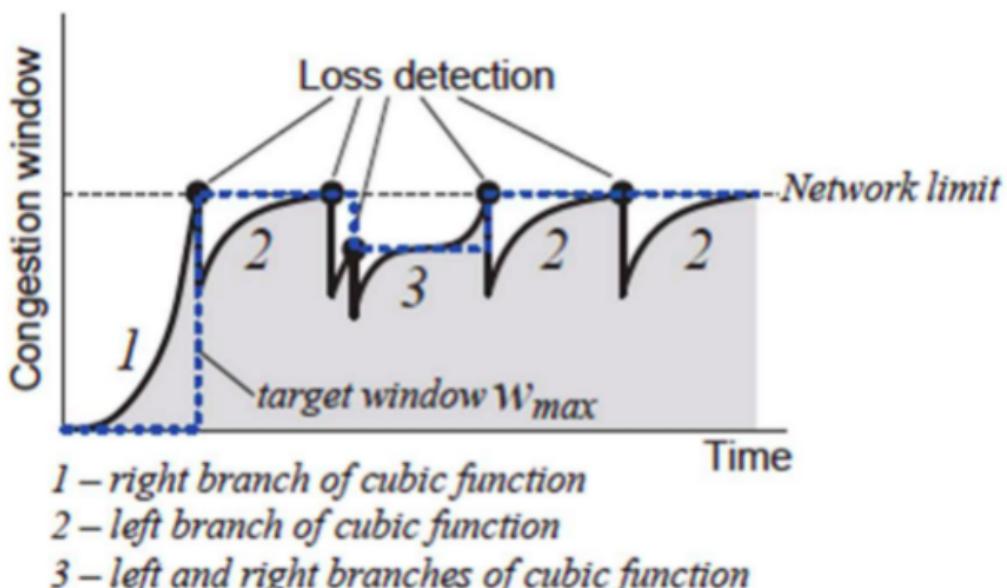


Fig. 3.28: *TCP CUBIC*

### 3.7.2 Protocollo BBR

*BBR* è un protocollo server-side sviluppato da Google nel 2016 e invece di basarsi sulle perdite, sfrutta due parametri, quali la *banda* del collegamento che funge da collo di bottiglia e l'*RTT*, per modulare la velocità di trasmissione dei *segmenti*. L'idea alla base del *BBR* è proprio riuscire a trasmettere i *segmenti* ad una velocità tale da evitare che si creino accodamenti.

**NB.** Il fatto che sia un protocollo *server-side* fa sì che non sia necessario implementare *BBR* anche sui client.

Il *BBR* inizia a trasmettere aumentando esponenzialmente il numero di *segmenti* trasmessi. Quando vede che i *segmenti* iniziano ad accodarsi, smette di trasmettere e aspetta fino allo smaltimento della coda. A questo punto, continua a trasmettere ad una frequenza alla quale non si creano accodamenti e tenta, ogni tanto, di incrementarla. Se l'incremento viene sostenuto dalla rete, cioè se continuano a non formarsi code, mantiene la frequenza aumentata, altrimenti ritorna a quella precedente. Questo ciclo si ripete periodicamente.

Un'altra cosa che il *BBR* fa periodicamente, è smettere di trasmettere e aspettare che la coda si svuoti del tutto. A quel punto trasmette un *segmento* e ne misura l'*RTT*. L'*RTT* misurato in quella situazione ideale viene impostato come *RTT minimo*.

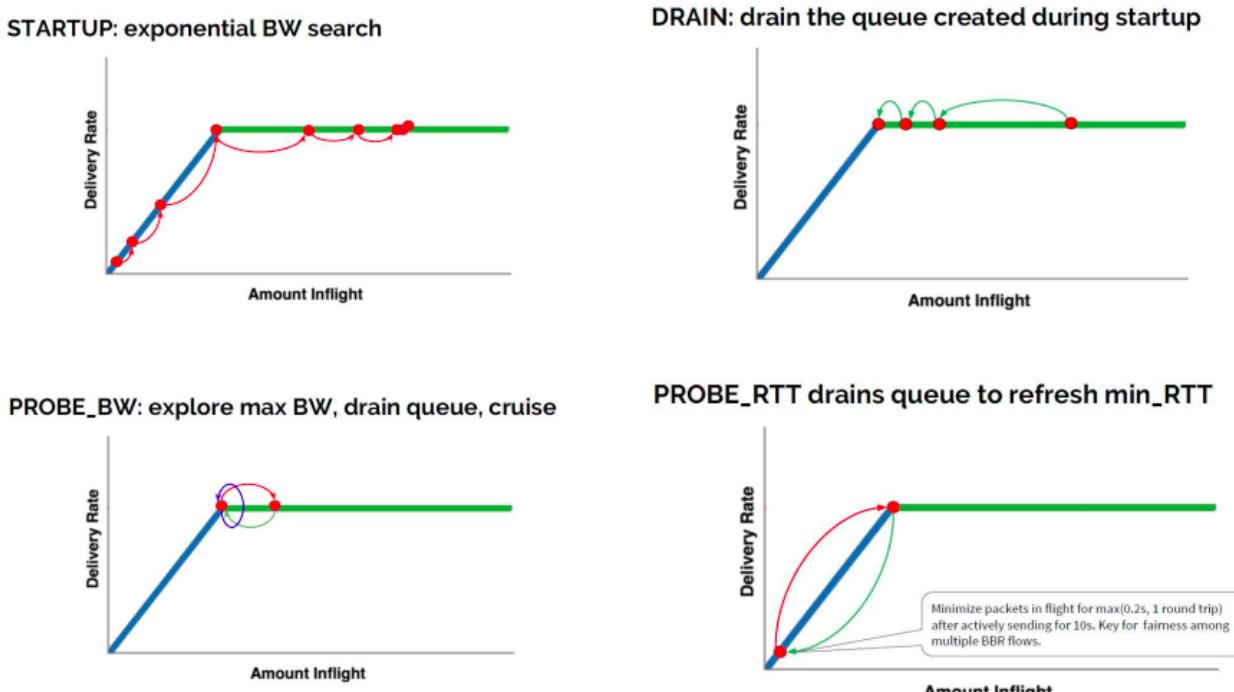


Fig. 3.29: *TCP BBR*

Secondo Google, il *BBR* permette di ottenere un *throughput* di circa  $9100\text{Mbit/s}$  contro i  $3.3\text{Mbit/s}$  del *CUBIC*. Inoltre, per le sue caratteristiche, il *BBR* è ottimo se combinato con l'*HTTP/2.0* e se utilizzato per raggiungere le cosiddette “reti di ultimo miglio”.

**NB.** *Reno* è il nome del *TCP* standard.

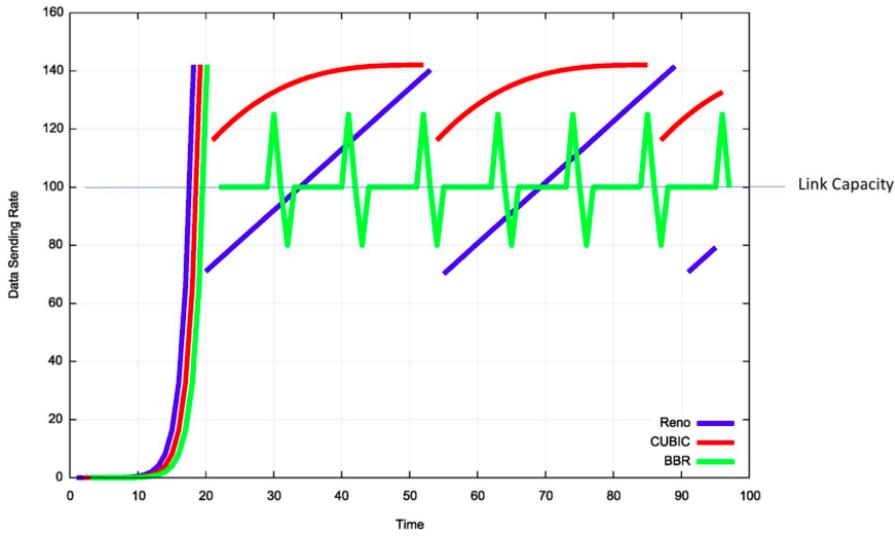


Fig. 3.30: *Reno VS CUBIC VS BBR*

### 3.7.3 Protocollo QUIC

Il protocollo *QUIC* mira ad essere equivalente ad una connessione *TCP*, ma riducendo l'overhead di connessione e utilizzando di base il protocollo *UDP* per il trasporto dei *segmenti*.

Il *QUIC* consente di ridurre l'overhead perché permette, nel processo di *handshake* iniziale, di incoporare anche i dati necessari all'instaurazione di una sessione *TLS*. In particolare, quando il client apre una connessione, il messaggio di risposta include anche i dati necessari all'uso della crittografia in *TLS*. In questo modo si evita di dover effettuare due *handshake* in successione: quello per il *TCP* e quello per il *TLS*.

Abbiamo detto però che il *QUIC* utilizza l'*UDP* invece del *TCP* e, infatti, la trasmissione si svolge attraverso flussi di dati *QUIC* che sono gestiti indipendentemente gli uni dagli altri. Ogni perdita viene gestita dal protocollo *QUIC* stesso e non dall'*UDP*.

Nel caso di cambi di rete, il protocollo *TCP* richiede la restaurazione di ogni connessione precedentemente utilizzata. Il *QUIC* invece, include un identificativo di connessione al server che è indipendente dalla fonte o dalla rete usata. Questo identificativo è memorizzato nel server e può essere usato dal client per ristabilire la connessione precedentemente inizializzata semplicemente inviando un *segmento* contenente quell'identificativo. Ciò permette di azzerare l'overhead necessario a riprendere la comunicazione dopo un cambio di rete.

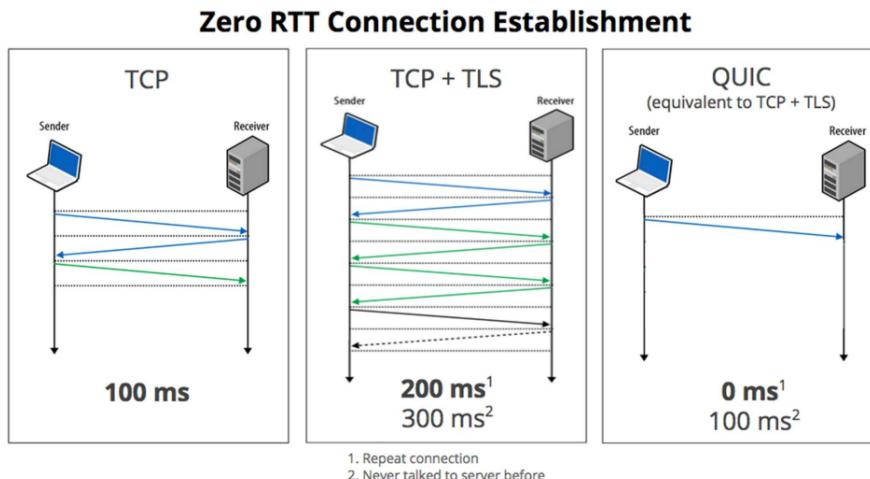


Fig. 3.31: Tempo necessario ad insitaurare una connessione

# *Capitolo Nr.4*

---

## *Livello Rete*

---

Il *livello Rete* si occupa di trasferire i *segmenti* ricevuti dal *livello Trasporto* alla rete di destinazione indicata. Il mittente, in particolare, dopo aver deciso la direzione in cui instradare i *pacchetti*, li incapsula all'interno di *frame* del *livello* sottostante. Giunti a destinazione, i *frame* e quindi i *pacchetti* vengono passati al *livello* superiore per poi essere consegnati al processo corretto.

Diversamente da quanto visto in precedenza, i protocolli del *livello Rete* non mettono in comunicazione diretta mittente e destinatario, bensì, ogni *router* trasmette i *pacchetti* ad uno dei *Router* ai quali è collegato, e quello ripete l'operazione fino a quando non si arriva alla rete di destinazione.

### **4.1 Funzioni del livello Rete**

Le funzionalità del *livello Rete* possono essere ripartite in due categorie in base alla loro scale: *locale* o *globale*.

Su scala *locale* i *router* eseguo l'operazione di *inoltro*, o *forwarding*, che consiste nello spostamento di un *pacchetto* da un'interfaccia del *router* ad un'altra. Agisce invece su scala *globale* l'operazione di *instradamento*, o *routing*, che permette di determinare il percorso di un *pacchetto*. L'operazione di *instradamento* è realizzata da particolari algoritmi di *routing*.

*Inoltro* e *instradamento* appartengono a due *piani* differenti che sono rispettivamente il *piano dati* e il *piano controllo*.

Il *piano dati*, o *data plane*, è una funzione locale di ogni *router* che determina come inoltrare un *pacchetto* da una porta di entrata a una porta di uscita dello stesso.

Il *piano controllo*, o *control plane*, invece, gestisce la logica globale della rete e lo fa determinando come instradare un *pacchetto* in un percorso *end-to-end*, cioè dall'*host* mittente all'*host* destinatario.

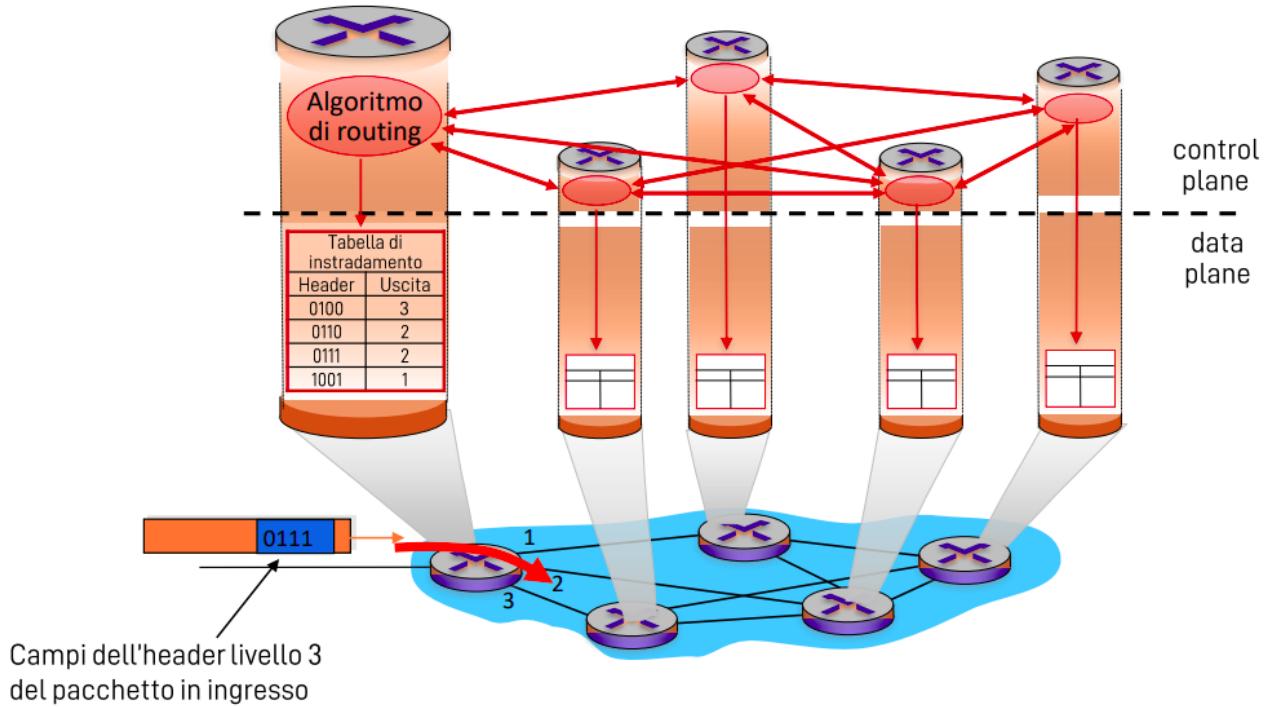


Fig. 4.1: *Data plane e control plane*

## 4.2 Struttura e funzionamento di un router

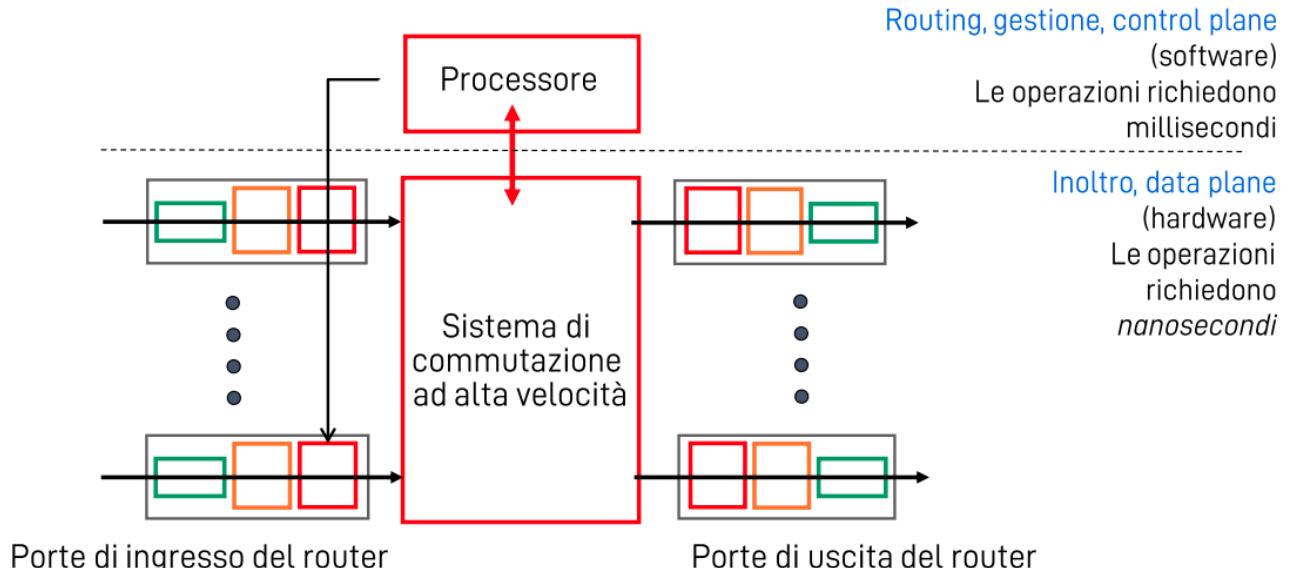


Fig. 4.2: Struttura di un *router*

### 4.2.1 Porte di ingresso

Ogni porta comprende tre componenti:

1. *Terminatore di linea*: riceve o invia i singoli bit;
2. *Livello Data Link*: un protocollo di *livello Data Link* per l'interpretazione dei bit (e.g. *Ethernet*);

3. *Sistema di commutazione decentralizzato*: è un componente che utilizza i valori dell'*header di livello 3* per decidere, usando una tabella di inoltro salvata in ogni porta, verso quale porta in uscita inoltrare un *pacchetto*.

Il *sistema di commutazione* di ogni porta è progettato in modo da ridurre al minimo il tempo di elaborazione perché l'obiettivo è quello di non introdurre ritardi eccessivi. In ogni caso, ogni porta è dotata di un buffer nel quale vengono accodati i *pacchetti* che devono essere *inoltrati*.

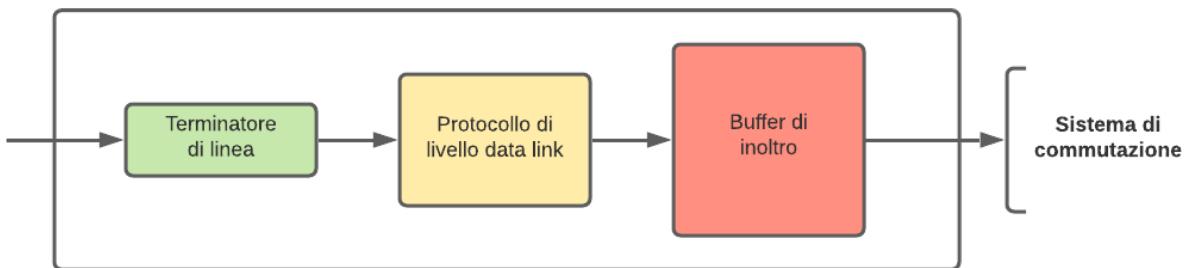


Fig. 4.3: Porta di ingresso

#### 4.2.2 Sistemi di commutazione

I sistemi di commutazione servono per trasferire i dati dalle porte di ingresso a quelle di uscita. La frequenza alla quale i *pacchetti* vengono trasferiti dagli ingressi alle uscite è detta *tasso di commutazione* e spesso è misurato come multiplo della velocità di comunicazione, ovvero, con  $N$  ingressi avremo una commutazione  $N$  volte più veloce della comunicazione. Esistono tre tipi di sistemi di commutazione: a *memoria*, a *bus* e a *matrice*.

**Commutazione a memoria** Questo sistema veniva usato nelle prime generazioni di *router* e di fatto faceva funzionare i *router* esattamente come i normali computer. I *pacchetti* venivano salvati in memoria, lì erano analizzati dalla CPU e quindi inviati verso la porta d'uscita.

La velocità di commutazione era limitata dalla banda dati della memoria e, inoltre, per ogni *pacchetto* erano necessari due accessi al bus di sistema.

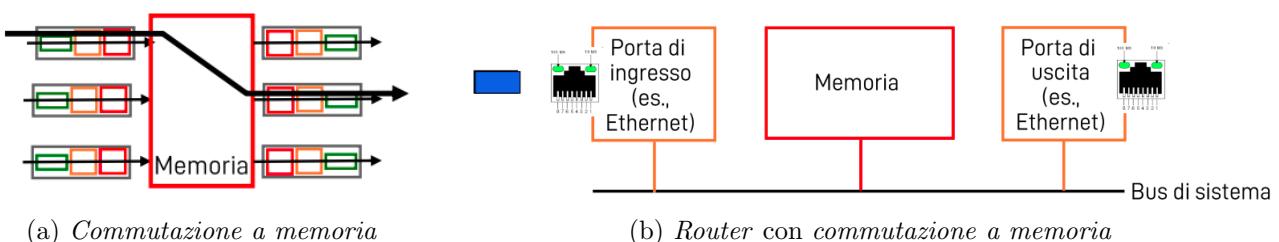


Fig. 4.4: *Commutazione a memoria*

**Commutazione a bus** In questo sistema si usa un bus dati condiviso attraverso il quale vengono trasferiti i *pacchetti*. Ovviamente la velocità di commutazione è limitata dalla banda del bus e i *pacchetti* devono transitare uno per volta.

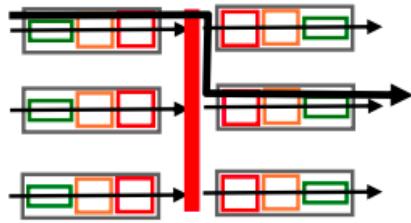


Fig. 4.5: *Commutazione a bus*

**Commutazione a matrice** Nei sistemi di commutazione a matrice vengono creati dei punti di interconnessione tra le linee di ingresso e le linee di uscita. Questa configurazione permette di superare i limiti di velocità della commutazione a bus perché più *pacchetti* possono transitare contemporaneamente. Inoltre, i *pacchetti* vengono frammentati in celle di lunghezza fissa.

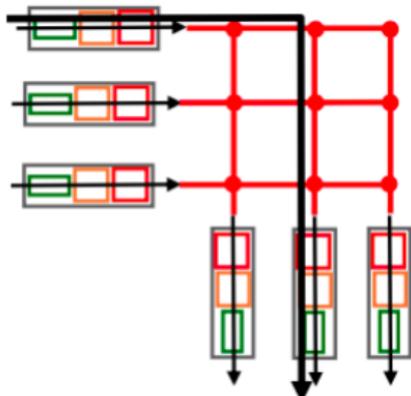


Fig. 4.6: *Commutazione a matrice*

#### 4.2.3 Conseguenze dei ritardi di commutazione

Un commutatore più lento della velocità complessiva delle porte di ingresso causa accodamenti agli ingressi e questo provoca ritardi e perdite nel caso in cui i buffer si riempiano.

Si parla di *Head of Line Block (HOL)* quando un *pacchetto* che si trova in testa alla coda di un buffer impedisce a quelli dietro di essere inoltrati. Questo blocco si verifica quando il commutatore è occupato da un altro *pacchetto* o quando la porta di uscita verso la quale il *pacchetto* in coda deve essere inoltrato è occupata dalla gestione di un altro *pacchetto*.

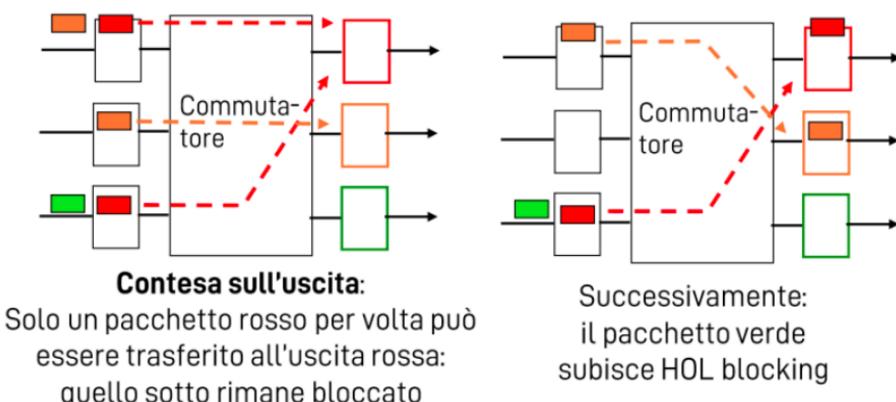


Fig. 4.7: *Head of Line Block*

#### 4.2.4 Porte di uscita

Le porte in uscita sono organizzate come quelle di ingresso, quindi con un buffer, un protocollo di *livello Data Link* e un terminatore di linea. Il buffer serve a memorizzare i *pacchetti* che devono essere spediti ed esistono diverse politiche, dette *polite di scheduling*, per la loro gestione: *FIFO* e *Priority scheduling*.

La politica *FIFO*, banalmente, invia i *pacchetti* in base al loro ordine di arrivo, mentre il *priority scheduling* decide l'ordine di invio sulla base della priorità assegnata ad ogni *pacchetto*.

Qualora si opti per uno *scheduling FIFO* bisogna comunque decidere come gestire i pacchetti in eccesso che non possono essere memorizzati nel buffer. Le cosiddette, *politiche di scarto* sono tre:

- *Tail drop*: tutti i *pacchetti* che non possono essere memorizzati vengono eliminati senza considerare altri parametri se non il tempo di arrivo;
- *Priority drop*: vengono eliminati i *pacchetti* con priorità più bassa in modo da fare spazio per gli altri;
- *Random drop*: i *pacchetti* da scartare vengono scelti casualmente;

**NB.** Il *priority scheduling* può andare in contrasto con quella che è la *network neutrality*.

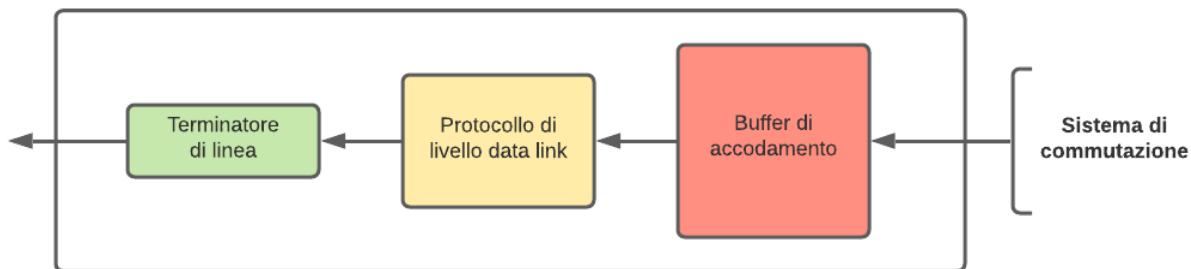


Fig. 4.8: Porta di uscita

**Dimensione dei buffer** Raccomandazioni recenti affermano che con  $N$  flussi di ingresso/uscita la quantità di memoria richiesta da ogni buffer è espressa dalla seguente espressione:

$$BM = \frac{RTT \cdot R}{\sqrt{N}}$$

### 4.3 Protocollo IP

#### 4.3.1 Struttura dei pacchetti

Un *pacchetto IP* ha un header organizzato in parole da 32 bit. I campi sono i seguenti:

- **Versione** (4bit): numero di versione del protocollo *IP* (4 o 6);
- **Lunghezza header** (4bit): numero di parole dell'header (5 se non ci sono opzioni);
- **Tipo di servizio** (8 bit): classe di servizio del *pacchetto*. In pratica è poco usato, ma potenzialmente potrebbe essere sfruttato per usare funzioni dette *DiffServ* e *Explicit Congestion Notification*;

- **Lunghezza totale** (16bit): numero totale di byte del *pacchetto* includendo sia l'header che il payload;
- **Id del pacchetto** (16bit): numero, solitamente sequenziale, usato per identificare i frammenti di un *pacchetto* e poterli poi riassemblare;
- **Flag** (3bit): i bit del campo specificano se si tratta del frammento di un *pacchetto* più grande e in particolare se è l'ultimo;
- **Offset del frammento** (13bit): indica in quale punto del *pacchetto* originale va inserito questo frammento (è espresso in multipli di 8byte);
- **TTL** (8bit): è un valore intero che viene impostato dal mittente. Ogni volta che il *pacchetto* passa per un *router* viene ridotto di 1 e quando arriva a zero viene eliminato;
- **Protocollo superiore** (8bit): specifica il protocollo di livello superiore usato dal payload;
- **Checksum dell'header** (16bit): è il complemento a 1 della somma di tutte le parole di 16bit dell'header;
- **Indirizzo IP sorgente** (32bit): indirizzo iniziale del mittente;
- **Indirizzo IP destinazione** (32bit): indirizzo della destinazione finale;
- **Opzioni IP**: solitamente è vuoto, ma può essere usato per controllare l'elaborazione e l'instradamento dei *pacchetti*;
- **Padding**: è un insieme di bit impostati a zero che vengono aggiunti se le opzioni non terminano ad un multiplo di 32bit per fare in modo che l'header sia un multiplo di 32bit;

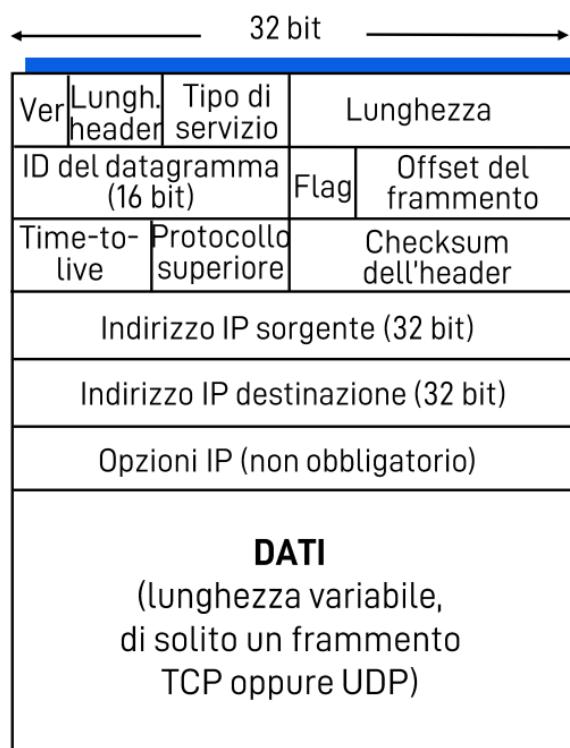


Fig. 4.9: Struttura di un *pacchetto IP*

### 4.3.2 Frammentazione dei pacchetti

Ogni *pacchetto* ha una dimensione massima che corrisponde a un limite imposto dall'hardware sul quale sta transitando. Questo valore limite è il già citato in precedenza *MTU*.

Poiché reti diverse possono avere *MTU* diversi, può capitare che un *pacchetto* risulti troppo grande per essere inviato attraverso una di quelle reti. Per questo motivo è possibile frammentare i *pacchetti* in oggetti di dimensione minore.

Il numero di frammenti necessari viene stabilito in base al valore di *MTU* e alla dimensione dell'header. Il payload viene quindi ripartito tra i frammenti e in ciascuno di essi viene incluso lo stesso header del *pacchetto* originale. Ovviamente alcuni campi dell'header vengono modificati per includere le informazioni necessarie alla gestione dei frammenti. In particolare viene impostato il campo *Flag*  $[0, D, M]$ :

- $D$ : è il flag “Do not fragment” che quando è impostato indica al ricevitore di non frammentare il *pacchetto* ed eventualmente di scartarlo se non fosse possibile inoltrarlo su una rete;
- $M$  è il flag “More fragments” e vale 1 su ogni frammento ad eccezione dell'ultimo;

Quando un *pacchetto* viene frammentato, non viene più riassemblato fino a quando non arriva al destinatario finale. Questo permette ai singoli frammenti di seguire percorsi diversi e soprattutto evita che ogni *router* debba ricomporre e in caso riframmentare di nuovo il *pacchetto*.

**NB.** I *router* che trattano i frammenti singolarmente indipendentemente gli uni dagli altri sono detti essere *stateless*.

Ovviamente, fino a quando il ricevitore non ha ricevuto tutti i frammenti non lì può ricomporre, quindi, quando arriva il primo frammento, il ricevitore imposta un timer, allo scadere del quale, se non sono arrivati tutti i frammenti, scarta quelli che ha memorizzato e ignora quelli che, in caso, dovessero arrivare.

Nella pratica però la frammentazione *IP* non si usa e infatti molti router non la implementano nemmeno. I motivi di questa scelta sono principalmente legati alla sicurezza:

- *Attacchi “overlapping fragments”*: sono attacchi che puntano ad “ingannare” i *router stateless* per permettere un traffico illecito di dati. Questo problema è risolvibile usando *router statefull* che però sono più costosi e difficili da realizzare;
- *Attacchi DDoS*: sono attacchi che puntano a congestionare un *host* evitando di proposito di inviare alcuni frammenti e impedendo, quindi, la ricomposizione;
- Errate implementazioni del codice di riassemblaggio possono provocare un crash del codice dei *router* permettendo così l'esecuzione di comandi arbitrari.

Inoltre, molti firewall si basano sull'ispezione degli header dei protocolli di *livello 4*, cosa che è impossibile con la frammentazione.

### 4.3.3 Indirizzamento

Gli *indirizzi IP* sono stringhe di 32 bit che vengono associate ad un'interfaccia di rete, che è un collegamento tra un *host*, o un *router*, e un collegamento fisico. Ad ogni interfaccia possono essere assegnati uno o più *indirizzi* diversi, ma a interfacce diverse non possono essere assegnati *indirizzi* uguali. Gli *indirizzi IP* pubblicamente raggiungibili devono essere univoci in tutta la rete.

**Struttura degli indirizzi IP** Gli *indirizzi IP* sono rappresentati mediante “dotted decimal notation”, ovvero, ogni byte codifica un valore intero positivo e ogni valore è separato dagli altri con un punto. Poiché un byte è composto da 8 bit, il valore di ogni ottetto va da 0 a 255 e quindi il range di *indirizzi* va da 0.0.0.0 a 255.255.255.255.

Generalmente, ogni *indirizzo IP* è diviso in due parti:

- *NetID*: è la parte iniziale dell’*indirizzo* e identifica la rete di appartenenza dell’*host* al quale è stato assegnato. Ogni rete internet è identificata in modo univoco;
- *HostID*: è la parte finale dell’*indirizzo* e identifica una specifica interfaccia collegata ad una rete.

In particolare, tutti gli *host* di una rete condividono lo stesso *NetID*, ma hanno *HostID* diversi.

**NB.** Nell’assegnamento degli *indirizzi IP* bisogna garantire che ad ogni *host* sia assegnato un *indirizzo* univoco, che i *NetID* siano coordinati a livello globale e che gli *HostID* possano essere decisi localmente senza bisogno di un coordinamento globale.

Per garantire l’univocità dei prefissi di rete a livello globale, sono stati istituiti degli enti appropriati. Uno di essi è l’**ICANN**, un ente che si occupa gestire l’assegnamento degli *indirizzi* e la risoluzione di controversie tra molteplici “pretendenti”. In realtà l’ICANN non distribuisce direttamente gli *indirizzi*, ma autorizza entità chiamate *registrar* a farlo. I *registrar* permettono agli **ISP** di prendere in carico una parte degli *indirizzi* e di distribuirli tra i propri clienti.

**Indirizzamento classful** Ovviamente è necessario stabilire un modo per distinguere il *NetID* dall’*HostID*. Inizialmente si era pensato ad un meccanismo a classi nel quale ogni classe stabiliva un numero fisso di bit da dedicare al *NetID*.

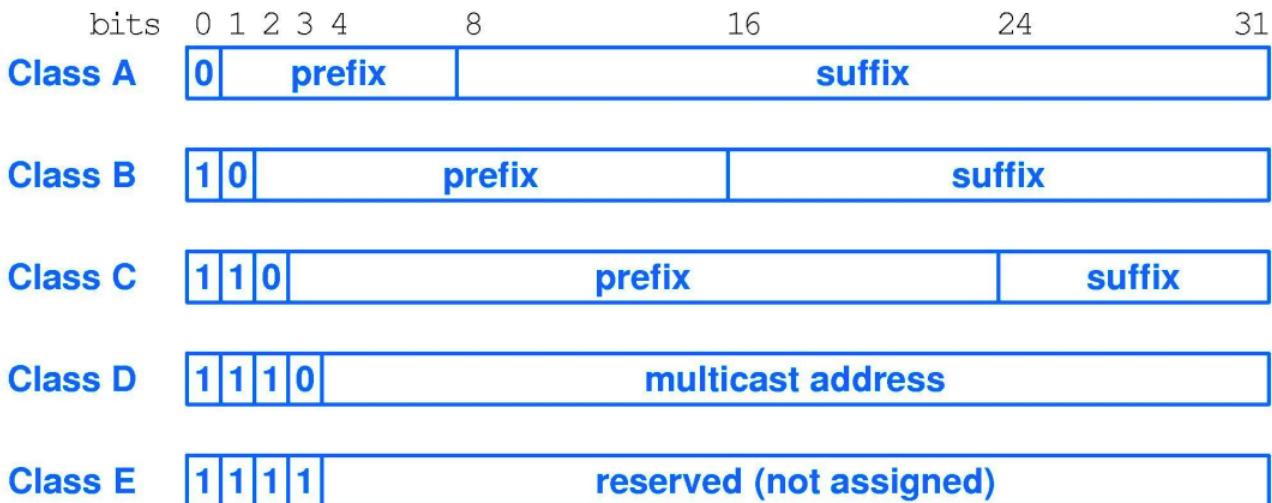


Fig. 4.10: Classi di *indirizzi IP*

Tuttavia, il sistema a classi si è rivelato inadatto in quanto gli utenti preferivano utilizzare classi con una porzione più ampia di indirizzi assegnabili, cosa che poi risultava in uno spreco.

**Indirizzamento classless** Con questo sistema, la suddivisione tra *NetID* e *HostID* è arbitraria e può essere fatta sulla base delle necessità di un utente.

Per esempio, se un’azienda necessitasse di 57 *indirizzi*, col sistema classful sarebbe necessario un *indirizzo* di classe C, che fornisce 256 *indirizzi*<sup>1</sup>, quando in realtà un *HostID* da 6 bit offrirebbe una suddivisione più efficiente.

<sup>1</sup>Vedremo più avanti che in realtà sarebbero 254

Il sistema classless permette di assegnare un prefisso di 26 bit e un suffisso da 6 ad un qualsiasi *indirizzo*. In pratica, se un *ISP* avesse a disposizione un indirizzo di classe C, potrebbe suddividere ulteriormente lo spazio di indirizzi “allungando” il *NetID*.

---

### Esempio 2 - Suddivisione di un indirizzo di classe C.

---

*Ad esempio, il seguente indirizzo di classe C:*

193.185.15.0

*corrisponde ad uno spazio di indirizzamento che va da 11000001.10111001.00001111.00000000 a 11000001.10111001.00001111.11111111.*

*Per suddividere lo spazio, l'ISP genera 4 NetID più lunghi:*

- NetID 1: 11000001.10111001.00001111.00xxxxxx;
- NetID 2: 11000001.10111001.00001111.01xxxxxx;
- NetID 3: 11000001.10111001.00001111.10xxxxxx;
- NetID 4: 11000001.10111001.00001111.11xxxxxx;

*Ognuna di queste reti può contenere 62 host perché gli HostID con tutti i bit impostati a 0 e a 1 sono riservati.*

Rimane comunque necessario stabilire un modo per distinguere il prefisso dal suffisso. La soluzione è l'utilizzo di una *netmask*, o maschera di rete, costituita come una stringa di 32 bit nella quale gli unici bit impostati a 1 sono quelli del prefisso. *Netmask* così definite permettono di risalire al *NetID* semplicemente eseguendo un AND bit-a-bit tra l'*indirizzo* di un *host* e la maschera.

---

### Esempio 3 - Applicazione di una netmask.

---

*Ad esempio, si prenda il seguente prefisso di rete:*

10000000.00001010.00000000.00000000 = 128.10.0.0

*con questa maschera:*

11111111.11111111.00000000.00000000 = 255.255.0.0

*Dato questo indirizzo:*

10000000.00001010.00000010.00000011 = 128.10.2.3

*l'AND bit-a-bit tra la netmask e l'indirizzo restituisce i primi 16 bit dell'indirizzo, ovvero:*

10000000.00001010.00000000.00000000 = 128.10.0.0

*che corrisponde proprio al prefisso di rete.*

Se notiamo, una maschera non è altro che una stringa di bit nella quale i primi tot bit sono impostati a 1 (e.g. nell'**esempio 2** sono i primi 26). Quindi, invece di indicare esplicitamente la maschera, è possibile usare la notazione **CIDR** e scrivere l'*indirizzo* seguito da /x dove x è il numero di bit a 1 (e.g. nell'esempio 2 scriveremmo /26).

---

#### Esempio 4 - Appliazione della notazione CIDR.

---

Si supponga che un ISP possieda il seguente blocco di indirizzi:

128.211.0.0/16

e che abbia tre clienti che necessitano rispettivamente di 12, 9 e 6 indirizzi.

Date le richieste, l'ISP calcola che ad ogni cliente serviranno 4, 4 e 3 bit per l'HostID, che corrispondono a 28, 28 e 29 bit di netmask.

Quindi, i 3 clienti potrebbero ricevere i seguenti indirizzi:

- Cliente 1: 128.211.0.16/28;
- Cliente 2: 128.211.0.32/28;
- Cliente 3: 128.211.0.48/28;

**NB.** I primi due clienti hanno prefissi diversi, ma la stessa maschera.

Ovviamente, quando a un cliente vengono assegnati degli indirizzi, questo può assegnarli ai propri *host* come meglio crede.

#### Conversione binario-decimale delle netmask

- $10000000 = 128 \Rightarrow /25;$
- $11000000 = 192 \Rightarrow /26;$
- $11100000 = 224 \Rightarrow /27;$
- $11110000 = 240 \Rightarrow /28;$
- $11111000 = 248 \Rightarrow /29;$
- $11111100 = 252 \Rightarrow /30;$
- $11111110 = 254 \Rightarrow /31;$
- $11111111 = 255 \Rightarrow /32;$

Le maschere /31 e /32 non hanno senso di esistere perché non forniscono *indirizzi assegnabili*.

**Regole di inoltro** Quando un *router* riceve un *pacchetto*, per decidere su quale interfaccia inoltrarlo, utilizza soltanto l'*indirizzo IP* di destinazione.

In particolare, ogni *router* gestisce una tabella di inoltro nella quale ad ogni interfaccia vengono assegnati gli indirizzi che permette di raggiungere.

Destinazione	Interfaccia
200.23.16.0/23	eth0
200.23.18.0/23	eth1
200.23.20.0/23	eth2

Fig. 4.11: Esempio di tabella di inoltro

Dato un *indirizzo* di destinazione, partendo dalla prima entry della tabella, si esegue l'AND bitwise tra l'*indirizzo* da inoltrare e la *netmask* indicata nelle singole entry. Viene individuata una corrispondenza quando il risultato dell'AND corrisponde all'*indirizzo* della entry.

---

#### Esempio 5 - Scelta dell'interfaccia di inoltro.

---

Valga la tabella di inoltro di cui sopra e sia 200.23.19.7 l'*indirizzo* da inoltrare.

Il router inizia a calcolare l'AND bitwise tra l'indirizzo da inoltrare e le varie netmask:

$$200.23.19.7 \ \& \ 255.255.254.0 \ (/23) \Rightarrow 200.23.18.0$$

In questo caso, l'indirizzo di inoltro è quello associato all'interfaccia `eth1` e quindi il pacchetto viene trasmesso a quell'interfaccia.

### Esempio 6 - Corrispondenze multiple.

Si consideri la seguente tabella di inoltro:

Destination	Interface
200.23.16.0/20	eth0
200.23.18.0/23	eth1

Dovendo inoltrare lo stesso indirizzo di prima:

$$200.23.19.7$$

si riscontra una corrispondenza con entrambe le entry, infatti:

$$200.23.19.7 \ \& \ 255.255.240.0 \ (/20) \Rightarrow 200.23.16.0$$

$$200.23.19.7 \ \& \ 255.255.254.0 \ (/23) \Rightarrow 200.23.18.0$$

In questo caso si segue la regola del "longest prefix matching" e si sceglie come interfaccia di inoltro quella associata all'indirizzo con prefisso più lungo.

Questo meccanismo di funzionamento delle tabelle di inoltro permette anche di aggregare più percorsi in un'unica interfaccia e, contemporaneamente, di destinare ad un'altra interfaccia i pacchetti indirizzati verso un indirizzo più specifico.

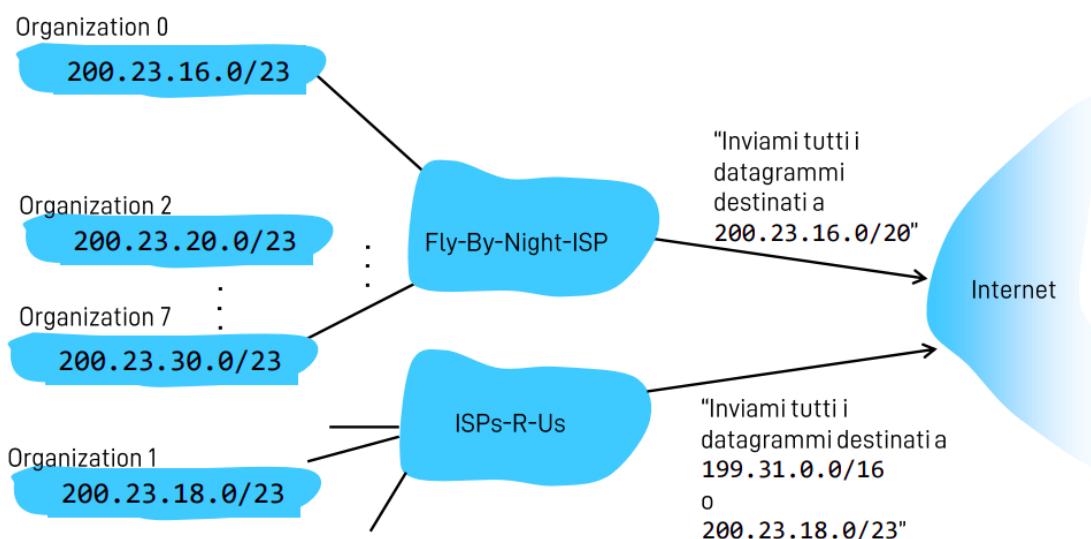


Fig. 4.12: Aggregazione di indirizzi nelle tabelle di inoltro

---

### Esempio 7 - Aggregazioni e casi specifici.

---

La tabella di inoltro associata all'immagine sopra è la seguente:

Destinazione	Interfaccia
200.23.16.0/20	Fly-By-Night-ISP
199.31.0.0/16	ISPs-R-Us
200.23.18.0/23	ISPs-R-Us

Dovendo inoltrare:

200.23.19.7

si rileva un riscontro con la prima e la terza entry, ma per la regola del “longest prefix matching” viene scelto 200.23.18.0/23 e quindi l’interfaccia *ISPs-R-Us*.

Nelle tabelle di inoltro esiste un’entry, l’ultima, che ha indirizzo 0.0.0.0/0 e viene scelta quando nessuna delle precedenti ha generato un riscontro. Questo valore è detto “*default gateway*” e negli *host* corrisponde solitamente all’indirizzo del *router*, mentre i *router* indicano l’indirizzo di un altro *router* che si suppone sappia come inoltrare il *pacchetto*.

L’ultimo caso da prendere in considerazione è quello in cui da un’interfaccia siano raggiungibili più *router*. In quel tipo di situazioni è necessario sapere quale *router* deve gestire la richiesta e per questo motivo, nelle tabelle di inoltro, viene indicato, se serve, anche l’indirizzo IP del *router* al quale inoltrare il *pacchetto*. Quel valore viene chiamato “*next hop*”.

Destinazione	Interfaccia	Inoltra a	"Next hop"
200.23.16.0/20	eth0	192.168.0.2	
199.31.0.0/16	eth1	192.168.1.2	
200.23.18.0/23	eth1	192.168.1.3	
0.0.0.0/0	eth2	192.168.2.7	

Fig. 4.13: Tabella di inoltro con “next hop”

**NB.** Nonostante la presenza dell’indirizzo *next hop*, gli indirizzi IP di sorgente e di destinazione non cambiano!

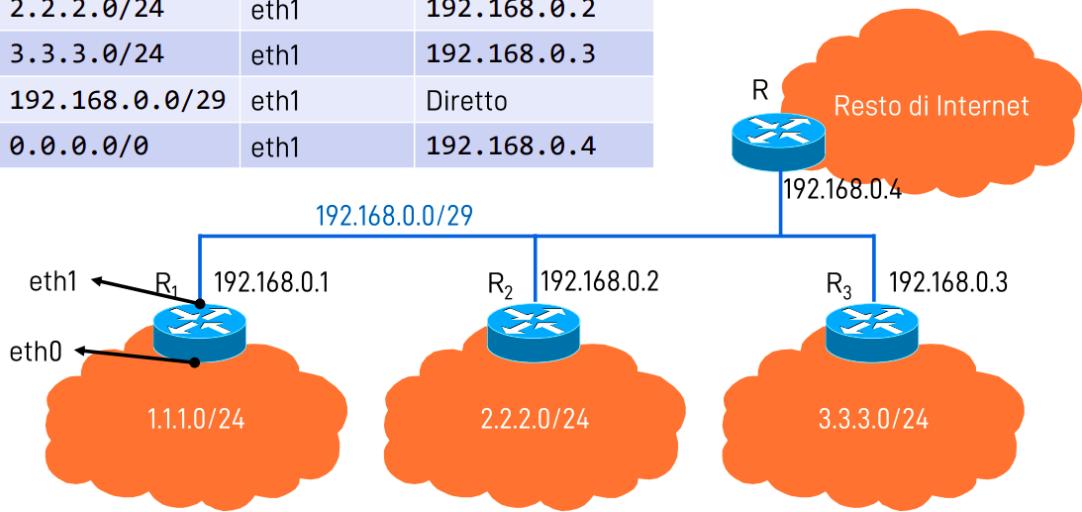
---

### Esempio 8 - Inoltro su reti contenenti più router.

---

La seguente figura rappresenta lo schema di indirizzamento, con relativa tabella di inoltro, di una rete alla quale sono collegati più router.

Destinazione	Interfaccia	Inoltra a
1.1.1.0/24	eth0	Diretto
2.2.2.0/24	eth1	192.168.0.2
3.3.3.0/24	eth1	192.168.0.3
192.168.0.0/29	eth1	Diretto
0.0.0.0/0	eth1	192.168.0.4



In particolare, quando un pacchetto raggiunge il router  $R_1$ , se è diretto verso la rete  $1.1.1.0/24$ , viene inoltrato verso l’interfaccia  $\text{eth}0$  senza indicare un indirizzo di “next hop” perché  $\text{eth}0$  è collegata direttamente alla rete da raggiungere.

D’altra parte, l’interfaccia  $\text{eth}1$  è collegata alla rete  $192.168.0.0/29$  che è adiacente ad altre reti. Di conseguenza, i pacchetti destinati ad host che stanno nelle reti  $2.2.2.0/24$  e  $3.3.3.0/24$  dovranno essere inoltrati all’interfaccia  $\text{eth}1$ , passare attraverso la rete  $192.168.0.0/29$  e raggiungere i router  $R_2$  e  $R_3$ . Poiché  $R_2$  e  $R_3$  sono raggiungibili agli indirizzi  $192.168.0.2$  e  $192.168.0.3$ , quegli indirizzi sono stati indicati come indirizzi di “next hop”.

Infine, i pacchetti destinati a reti non conosciute da  $R_1$  sono inviati al default gateway, che in questo caso, è il router  $R$  all’indirizzo  $192.168.0.4$ .

**NB.** Tipicamente, le tabelle di routing presenti negli *host* (non i *router*), hanno una struttura simile alla seguente:

Destinazione	Interfaccia	Inoltra a
192.168.0.0/24	eth1	Diretto
0.0.0.0/0	eth1	192.168.0.1

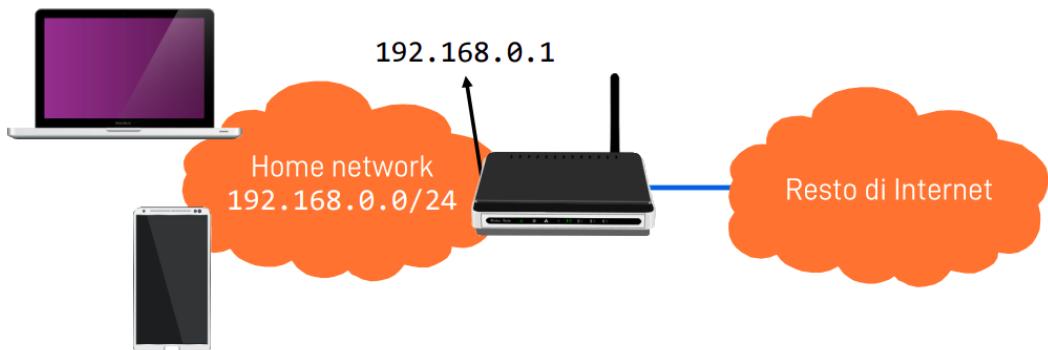


Fig. 4.14: Generica tabella di routing di un *host*

#### 4.3.4 Indirizzi privati e NAT

**Indirizzi pubblici e privati** Non tutti gli *indirizzi IP* possibili sono *pubblici*, ma ne esistono alcuni che sono, per l'appunto, *privati*. Questo tipo di indirizzi possono essere usati soltanto all'interno di reti locali e non sono raggiungibili da *host* che risiedono in altre reti, infatti i *router* sono quasi sempre impostati per scartare i *pacchetti* con indirizzi privati. Ovviamente, all'interno di una rete locale, devono comunque essere univoci.

Gli indirizzi privati sono distribuiti in 3 blocchi:

- 10.0.0.0/8: da 10.0.0.0 a 10.255.255.255;
- 172.16.0.0/12: da 172.16.0.0 a 172.31.255.255;
- 192.168.0.0/16: da 192.168.0.0 a 192.168.255.255;

**NAT** Visto che non è possibile comunicare in rete con *host* dotati di *indirizzi IP privati*, ci si potrebbe chiedere come sia possibile per essi inviare *pacchetti*. Le soluzioni possibili sono due:

- *Proxy*: usare un computer dotato sia di un *indirizzo pubblico* che di uno *privato*. Questo computer riceve tutte le richieste verso l'esterno e le esegue per conto dei mittenti;
- **NAT**: è un apparecchio che sostituisce l'*IP* e il *numero di porta* sorgenti di ogni *pacchetto* con il proprio *indirizzo IP*, che è pubblico, e un *numero di porta* casuale generato al momento;

In particolare, il *NAT* gestisce una tabella di questo tipo:

Da	A
IP sorgente privato	IP pubblico del NAT
IP destinazione pubblico	IP destinazione pubblico
Porta sorgente	Un'altra porta sorgente
Porta destinazione	Porta destinazione
Protocollo	Protocollo

Fig. 4.15: Tabella *NAT*

Quando un *host* della rete privata trasmette un *pacchetto* che ha per destinazione un *indirizzo IP pubblico*, il *NAT* inserisce in una tabella una nuova entry, nella quale, l'*indirizzo IP privato* dell'*host* e il *numero di porta*, vengono associati all'*indirizzo pubblico* del *NAT* e a un altro *numero di porta* generato al momento.

Quindi, il *pacchetto* viene ritrasmesso dal *NAT* con le sorgenti cambiate e quando l'*host* destinatario risponde, il *pacchetto* di risposta viene nuovamente intercettato dal *NAT* che sostituisce l'*indirizzo IP* e la *porta* di destinazione con i valori salvati nella tabella. A quel punto, il pacchetto può essere inoltrato alla sua destinazione.

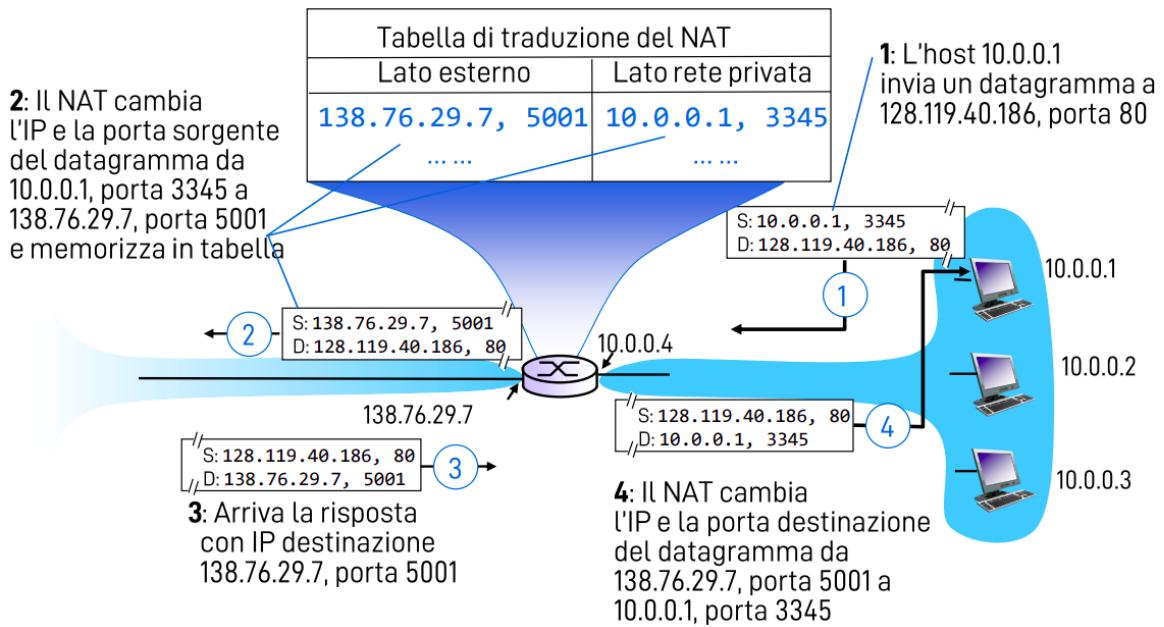


Fig. 4.16: Esempio di utilizzo del *NAT*

Il *NAT* è considerato una soluzione controversa in quanto porta con se molti vantaggi, ma anche alcune problematiche.

Tra i vantaggi, si ha che il *NAT* permette di ottenere fino a  $2^{16} \approx 60000$  connessioni simultanee con un solo *IP pubblico*, tamponando così il problema di carenza di indirizzi. Inoltre, il *NAT* impedisce a *host* esterni alla rete di comunicare con gli *host* interni se non sono stati questi ad avviare la comunicazione, in quanto, se un *host* esterno tentasse di comunicare, il *NAT* non saprebbe a chi inoltrare i *pacchetti* poiché non esisterebbe un'entry adatta nella tabella.

Quest'ultimo però è anche il primo degli svantaggi, perché i server devono poter ricevere comunicazioni e quindi si vede necessario utilizzare degli espedienti quali il *port forwarding*, l'*hole punching* o altri. Inoltre, il *NAT* viola due dei principi fondanti dell'architettura di internet: la trasparenza e l'incapsulamento.

La trasparenza è violata perché il *NAT* non è sempre trasparente ai programmi applicativi e l'incapsulamento è violato perché vengono modificati gli header di livello 3 e 4.

**NB.** In certi casi il *NAT* può essere l'unico modo di permettere a due *host* di comunicare se non si controllano tutti i *router* nel mezzo. Ad esempio, nella figura seguente l'*host* a sinistra ha bisogno del *NAT* sul *router R* per aprire una pagina sul server.

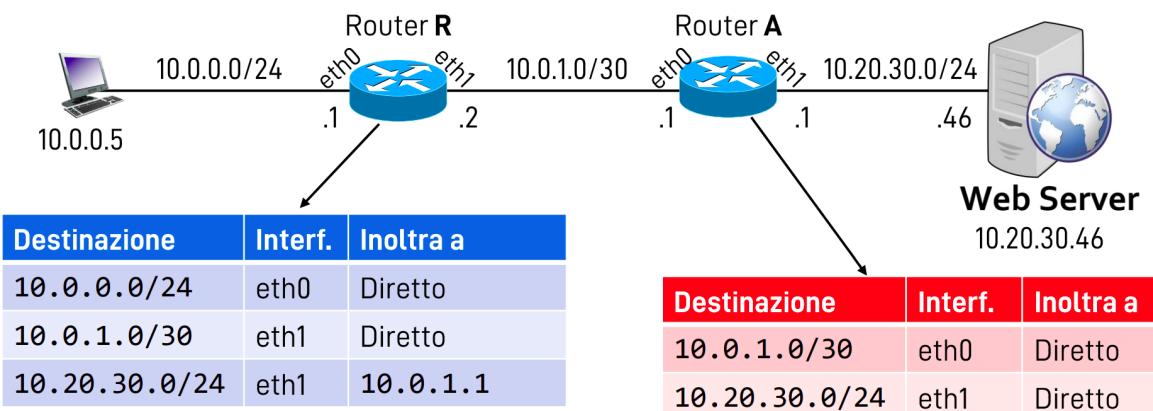


Fig. 4.17: Esempio di necessità del *NAT*

### 4.3.5 Indirizzi speciali

Esistono alcuni *indirizzi IP* che sono “speciali” e che hanno significati particolari.

**Indirizzi di rete** Sono indirizzi usati per riferirsi al prefisso di una rete e sono formati mettendo a 0 tutti i bit dell’*HostID*.

Per esempio, 128.211.0.16/28 è un *indirizzo di rete* perché in binario vale:

$$\begin{array}{c} \underline{10000000.11010011.00000000.0001} \quad \underline{0000} \\ \text{NetID} \qquad \qquad \qquad \text{HostID} \end{array}$$

D’altra parte, l’indirizzo 128.211.0.17/28 non può essere un *indirizzo di rete* perché è codificato come:

$$\begin{array}{c} \underline{10000000.11010011.00000000.0001} \quad \underline{0001} \\ \text{NetID} \qquad \qquad \qquad \text{HostID} \end{array}$$

ma può essere un *host* di quella rete.

**Indirizzi di broadcast di rete** Gli *indirizzi di broadcast di rete* o *directed broadcast addresses*, hanno tutti i bit dell’*HostID* impostati a 1 e si riferiscono a tutti gli *host* di una rete. I *pacchetti* con questo tipo di indirizzi vengono inoltrati dai *router* in singola copia fino a quando non viene raggiunto un *router* della rete di destinazione, il quale, provvede a consegnare una copia di quel *pacchetto* ad ogni *host* della rete.

Per esempio, data la rete 128.211.0.16/28, l’*indirizzo di broadcast di rete* è:

$$\begin{array}{c} \underline{10000000.11010011.00000000.0001} \quad \underline{1111} \\ \text{NetID} \qquad \qquad \qquad \text{HostID} \end{array}$$

**Indirizzo di broadcast di rete locale** L’*indirizzo di broadcast di rete locale* o *limited broadcast address* è un indirizzo in cui tutti i bit sono impostati a 1 e si riferisce a tutti gli *host* di una rete, ovvero:

$$255.255.255.255 \Rightarrow 11111111.11111111.11111111.11111111$$

La differenza con gli *indirizzi di broadcast di rete* è che i *pacchetti* con questo indirizzo non vengono mai inoltrato dai *router* e pertanto rimangono confinati all’interno della rete locale.

**Indirizzo “questo computer”** Questo indirizzo è composto da 32 bit a 0, ovvero:

$$0.0.0.0 \Rightarrow 00000000.00000000.00000000.00000000$$

ed è usato dai computer quando sono stati appena avviati e non hanno ancora un *indirizzo IP*.

**Indirizzi di loopback** Gli *indirizzi di loopback* sono tutti gli indirizzi appartenenti alla rete 127.0.0.0/8 e sono usati per testare applicazioni di rete in esecuzione su un computer. In particolare, i pacchetti con questo indirizzo, discendono lo *stack protocollare* fino al livello 3 e quindi lo risalgono per essere consegnati all’applicazione destinataria senza lasciare il computer.

**Indirizzi multicast** Gli *indirizzi multicast* servono ad inviare *pacchetti* ad un gruppo di *host* e sono indirizzi che iniziano con 1110 e quindi sono compresi tra 224.0.0.0 e 239.255.255.255.

La logica di funzionamento è uguale a quella degli *indirizzi di broadcast di rete* che consente di evitare di dover trasmettere più copie di uno stesso *pacchetto*. Il problema è che la maggior parte dei *router* bloccano i *pacchetti* con *indirizzi multicast* e quindi il loro utilizzo è fortemente limitato.

**Indirizzi link-local** Gli *indirizzi link-local* sono assegnati automaticamente agli *host* che non riescono a farsi assegnare un *indirizzo IP*<sup>2</sup>.

In particolare, gli *host* scelgono casualmente uno degli indirizzi della sottorete 169.254.0.0/16 e possono quindi comunicare soltanto all'interno di quella sottorete.

**Indirizzi IP per i router** Ad ogni *router* possono essere assegnati uno o più indirizzi, o meglio, è possibile assegnare almeno un indirizzo ad ogni interfaccia.

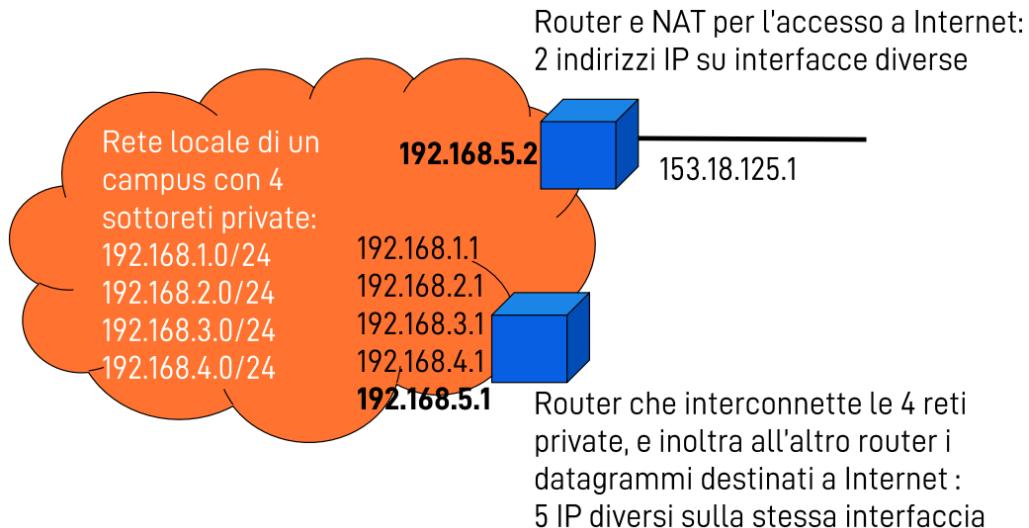


Fig. 4.18: *Router* con più indirizzi assegnati a un'interfaccia

Nella rete di questa immagine, tutto il traffico delle 4 sottoreti private viene gestito dal *router* in basso. Nel caso di traffico destinato all'esterno, il *router* trasmette quei *pacchetti* al secondo *router* e questo poi li invia in rete. Questo sistema permette di dividere i *domini di broadcast*<sup>3</sup>, ovvero limitare il numero di *host* che ricevono i *pacchetti* con *indirizzi di broadcast*, e alleggerire il carico del secondo *router*.

**NB.** Gli *indirizzi di rete* e *di broadcast di rete* sono il motivo per quale gli indirizzi assegnabili di una rete sono pari al numero di possibili *host* rappresentabili meno 2.

## 4.4 Protocollo ARP

### 4.4.1 Cenni sul livello 2

Prima di passare alla trattazione del protocollo *ARP* dobbiamo chiarire alcuni dettagli sul livello 2.

Ogni *pacchetto IP* viene encapsulato un *frame* di livello 2 e per poter trasmettere il *frame* è necessario specificare gli indirizzi di sorgente e di destinazione di quel livello. Questi indirizzi sono detti *indirizzi MAC* e sono stringhe di 48 bit, espresse come 12 cifre esadecimale, che identificano univocamente ciascuna interfaccia di rete.

---

<sup>2</sup>Vedremo più avanti cosa significa

<sup>3</sup>Un *dominio di broadcast* è l'insieme di tutti gli *host* che ricevono i *pacchetti* trasmessi in *broadcast*. C'è un'associazione uno-a-uno tra *sottorete* e *dominio di broadcast*.

#### 4.4.2 Principi dell'ARP

Dato che per trasmettere un *frame* è necessario conoscere l'*indirizzo MAC* del destinatario deve esistere un protocollo che consenta di scoprirla. Il protocollo *ARP* serve appunto a ricercare il *MAC* associato ad un'interfaccia con un certo *indirizzo IP* noto a priori.

In particolare, l'*host* che deve scoprire l'*indirizzo MAC* trasmette in broadcast una richiesta *ARP* specificando l'*IP* dell'interfaccia dell'*host* di cui ha bisogno di conoscere il *MAC*. La richiesta viene ricevuta da tutti gli *host*, ma soltanto l'interessato risponde specificando il proprio *MAC*.

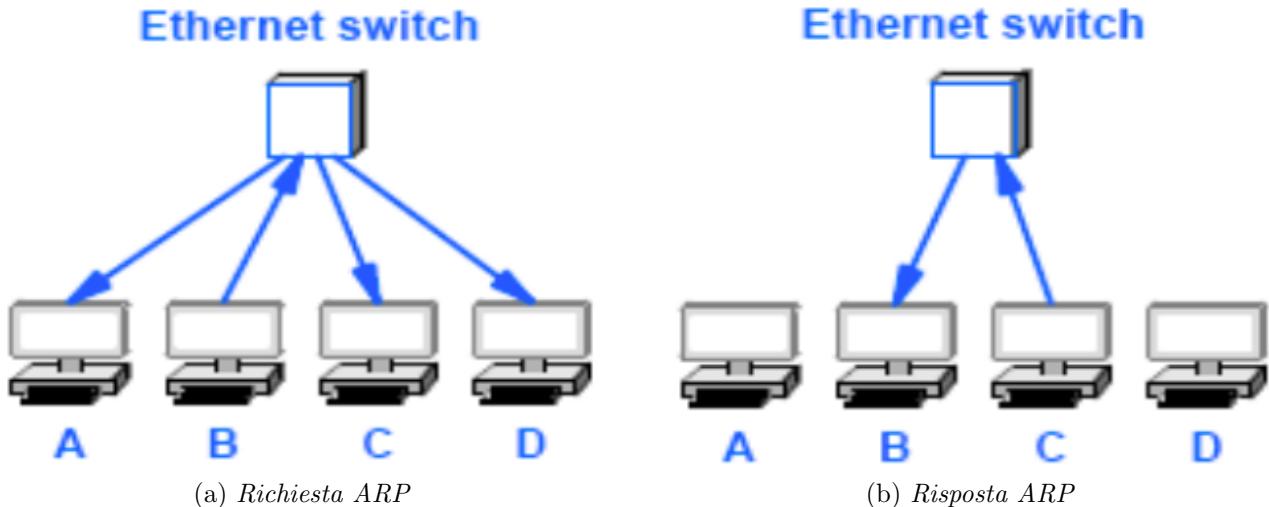


Fig. 4.19: Funzionamento del protocollo *ARP*

**NB.** Diversamente da quanto avviene a livello 3 in cui mittente e destinatario comunicano senza interessarsi degli *host* nel mezzo, al livello 2 tutte le comunicazioni sono punto-punto, quindi gli *indirizzi MAC* cambiano ad ogni salto. Inoltre, quando si deve trasmettere un *pacchetto* all'esterno della rete, non si cerca l'*indirizzo MAC* dell'*host* sull'altra rete, ma lo si lascia fare al *router*.

#### 4.4.3 Messaggi ARP

La seguente è la struttura di un messaggio *ARP*:

0	8	16	24	31			
<b>HARDWARE ADDRESS TYPE</b>		<b>PROTOCOL ADDRESS TYPE</b>					
HADDR LEN	PADDR LEN	OPERATION					
<b>SENDER HADDR (first 4 octets)</b>							
SENDER HADDR (last 2 octets)		SENDER PADDR (first 2 octets)					
SENDER PADDR (last 2 octets)		TARGET HADDR (first 2 octets)					
<b>TARGET HADDR (last 4 octets)</b>							
<b>TARGET PADDR (all 4 octets)</b>							

Fig. 4.20: Struttura di un messaggio *ARP*

Con **Hardware Address** o **HADDR** ci si riferisce all'*indirizzo MAC*, mentre con **Protocol Address** o **PADDR** all'*indirizzo di livello 3*. I messaggi *ARP* vengono trattati come *pacchetti* di livello 3 e quindi sono incapsulati all'interno di *frame* di livello 2.

#### 4.4.4 ARP caching

Per evitare di dover trasmettere un messaggio *ARP* per ogni *pacchetto*, vengono mantenute in memoria le risposte *ARP* ricevute in precedenza.

Le corrispondenze salvate vengono mantenute per 30 secondi prima di essere scartate e nel caso in cui si ricevano nuove risposte *ARP* relative a corrispondenze già salvate, vengono sovrascritte le precedenti.

#### 4.4.5 Proxy ARP

Nella situazione rappresentata nella figura seguente si ha un *router* che separa due gruppi di *host* che quindi stanno su reti diverse.

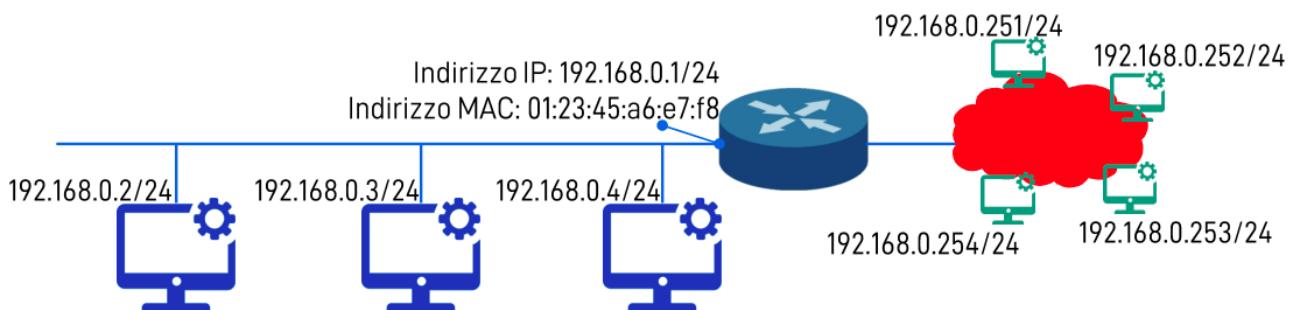


Fig. 4.21: *Proxy ARP*

Tuttavia, possiamo notare che sia gli *host* di sinistra che quelli di destra hanno indirizzi della stessa sottorete 192.168.0.0/24.

Questo è possibile perché il *router* funge da *proxy ARP*. Ovvero, quando un *host* sulla sinistra intende comunicare con un *host* sulla destra, o viceversa, invia una richiesta *ARP* che viene intercettata dal *router* il quale risponde con il proprio *indirizzo MAC*. A quel punto, è il *router* stesso che si incarica di inoltrare i *pacchetti* alla loro destinazione effettiva.

### 4.5 Protocollo ICMP

L'*IP* include un protocollo ausiliario chiamato *ICMP* che viene usato per notificare errori all'*host* mittente di un *pacchetto* e per trasportare altre informazioni utili.

*IP* e *ICMP* sono interdipendenti in quanto l'*IP* usa *ICMP* per segnalare errori e i *pacchetti ICMP* viaggiano su *pacchetti IP*.

#### 4.5.1 Messaggi ICMP

La struttura dei messaggi *ICMP* è estremamente semplice. L'header si compone soltanto di 3 campi: `type`, `code` e `checksum`.

I messaggi possono essere distinti in due classi: quelli per la segnalazione di errori (e.g. `Time Exceeded` e `Destination Unreachable`) e quelli usati per recuperare informazioni (e.g. `Echo Request` e `Echo Reply`).

**NB.** Per evitare di congestionare la rete, *ICMP* è progettato per non segnalare errori provocati da altri messaggi *ICMP*.

Campo Type	Descrizione
Destination unreachable	Il pacchetto non può essere consegnato
Port unreachable	Nessuno all'host destinatario ascolta sulla porta specificata
Time exceeded	Il campo TTL dell'header IP è sceso a 0 (routing loop?)
Parameter problem	Valore non valido in un campo dell'header IP
Source quench	Chiede alla sorgente di rallentare l'invio di dati
Redirect	Dice al router a quale host inoltrare il datagramma
Echo and echo reply	Usati da <b>ping</b> per capire se un host è attivo
Timestamp request/reply	Come i comandi echo, ma per l'orologio locale del router
Router advertisement/solicitation	Per proporsi come router, o per chiedere quali router ci sono nelle vicinanze
Fragmentation needed	Flag IP «non frammentare» a 1, ma datagramma eccede l'MTU

Fig. 4.22: Tipi di messaggi *ICMP*

### 4.5.2 Ping tramite ICMP

Il comando **ping** è implementato sfruttando le **Echo Request** e le **Echo Reply**. Con una **Echo Request** viene trasmesso un messaggio e il destinatario esegue una **Echo Reply** ritrasmettendo indietro lo stesso messaggio ricevuto.

Questo comportamento permette di verificare la raggiungibilità di un *host* e di misurarne l'**RTT**.

### 4.5.3 Traceroute tramite ICMP

Vengono inviati pacchetti *IP* con **TTL** sempre maggiore (1, 2, 3, ...) fino a quando non viene raggiunta la destinazione. Ogni volta che un *router* scarta un *pacchetto* con **TTL** nullo, trasmette un messaggio *ICMP* di tipo **Time Exceeded** con il proprio *IP*.

L'insieme di tutti i messaggi *ICMP* ricevuti permette di tracciare la rotta seguita dal un *pacchetto* per raggiungere una certa destinazione.

## 4.6 Protocollo DHCP

### 4.6.1 Principio di funzionamento

Il **DHCP** è un protocollo client-server che permette di assegnare dinamicamente gli *indirizzi IP* agli *host* di una rete. La procedura di assegnamento, in sintesi, si compone di 4 fasi:

1. L'*host* invia un messaggio **DHCP discover**;
2. Il server **DHCP** risponde inviano con un messaggio **DHCP offer**;
3. L'*host* accetta l'offerta del server;
4. Il server risponde con un **DHCP ACK**;

Poiché il client non ha un *indirizzo IP* tutte le comunicazioni avvengono in *broadcast*, ovvero il client contatta il server indicando come indirizzo sorgente 0.0.0.0, mentre il server risponde all'indirizzo 255.255.255.255.

Per distinguere tra loro più transazioni viene usato un **transaction ID** generato casualmente dal client. Il server risponde ad ogni richiesta indicando nei messaggi lo stesso valore di **transaction ID** della richiesta. In questo modo i client possono capire se la risposta è o meno destinata a loro.

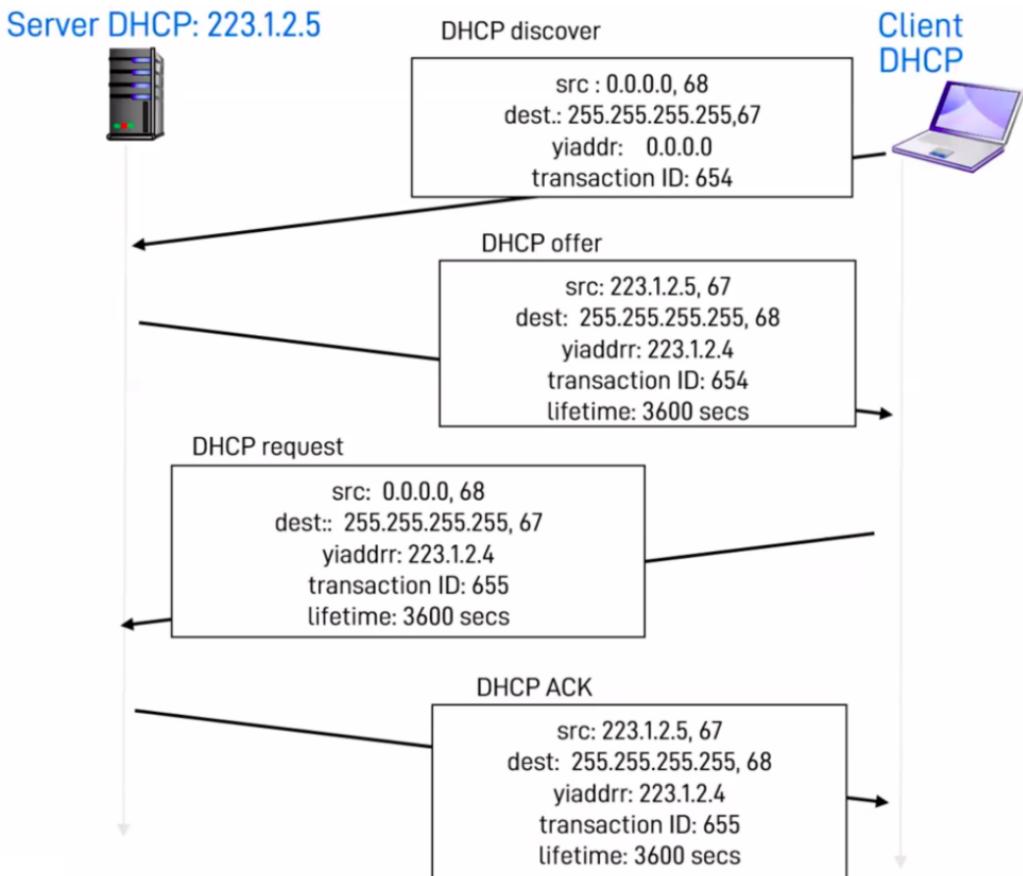


Fig. 4.23: Transazioni *DHCP*

Il campo **yiaddr** (your ip address) contiene l'*indirizzo IP* che il server *DHCP* propone al client.

**NB.** Il protocollo *DHCP* è un protocollo del *livello Applicativo*, infatti, server e client sono identificati da una *socket*. In particolare, il server opera sulla porta 67, mentre il client sulla 68. Inoltre, il *DHCP* utilizza il protocollo *UDP* e non potrebbe fare altrimenti visto che il client non ha un *indirizzo IP* con il quale instaurare una connessione *TCP*. Ovviamente, la gestione di perdite e duplicazioni è gestita bene dal protocollo.

#### 4.6.2 Gestione dei prestiti

L'*indirizzo IP* fornito dal *DHCP* è valido per una certa quantità di tempo al termine del quale, l'*indirizzo* ritorna disponibile per altre assegnazioni. Al termine di un prestito il client può richiedere un nuovo *IP* o l'estensione del prestito.

Solitamente i client, prima dello scadere del tempo, chiedono il rinnovo al server e questo generalmente lo autorizza. Il motivo per il quale il server tende ad autorizzare i rinnovi è che altrimenti gli *host* dovrebbero richiedere un nuovo indirizzo e riaprire tutte le connessioni che stavano utilizzando, causando molto traffico all'interno della rete. Tuttavia, qualora il server neghi l'estensione di un prestito, il client deve obbligatoriamente smettere di usare quell'*indirizzo* per non causare collisioni.

**NB.** Oltre all'*indirizzo IP*, il *DHCP* fornisce anche altre informazioni quali il *default gateway*, la *netmask* (o *subnetmask*) e il nome e l'*indirizzo* del server *DNS*.

#### 4.6.3 Messaggi DHCP

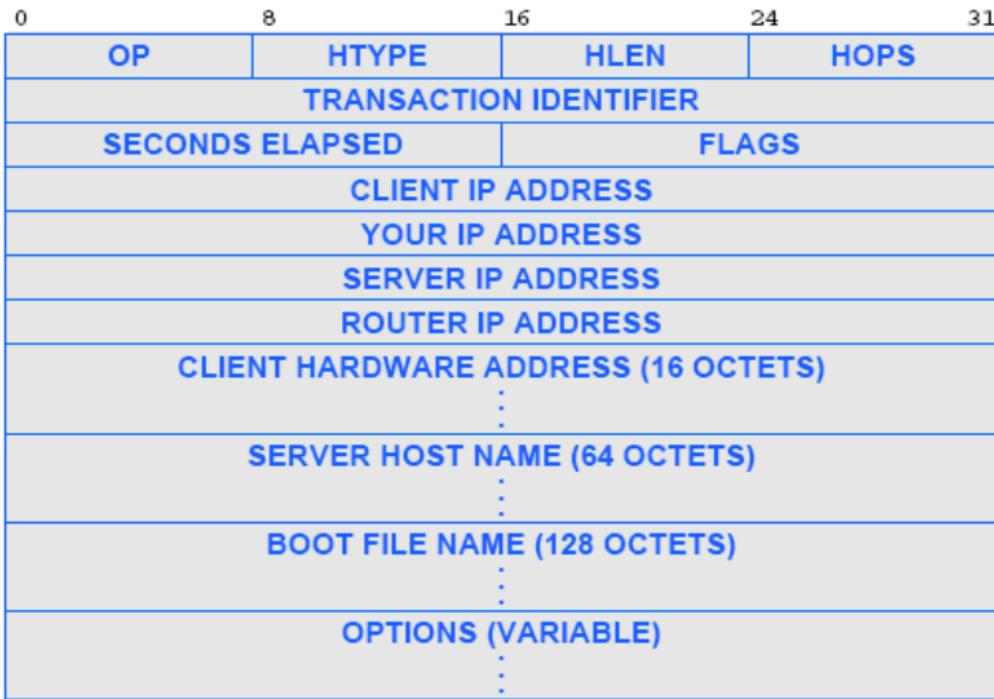


Fig. 4.24: Struttura di un messaggio *DHCP*

Vediamo nel dettaglio il significato dei vari campi:

- **OP**: specifica se si tratta di una richiesta o di una risposta;
- **HTYPE**: specifica il tipo di interfaccia hardware utilizzata;
- **HLEN**: specifica la lunghezza dell'*indirizzo MAC*;
- **FLAGS**: specifica se il mittente può ricevere broadcast o risposte dirette;
- **HOPS**: specifica quanti server hanno inoltrato la richiesta;
- **TRANSACTION IDENTIFIER**: è usato per far corrispondere le risposte alle richieste;
- **SECOND ELAPSED**: indica i secondi passati da quando l'*host* è entrato in funzione;
- **CLIENT IP ADDRESS**: *indirizzo IP* del client *DHCP*;
- **YOUR IP ADDRESS**: *indirizzo IP* proposto dal server (inizialmente vale 0.0.0.0);
- **SERVER IP ADDRESS**: *indirizzo IP* del server *DHCP*;
- **SERVER HOST NAME**: *nome di dominio* del server *DHCP*;
- **ROUTER IP ADDRESS**: *indirizzo* del *default gateway*;
- **CLIENT HARDWARE ADDRESS**: *indirizzo MAC* del client *DHCP*;
- **BOOT FILE NAME**: percorso di un file con istruzioni per configurare un il client all'avvio;

**NB.** Tutti i campi, ad eccezione di **OPTIONS**, hanno una dimensione fissa.

#### 4.6.4 Reti senza DHCP

Se una rete non ha un server *DHCP* e gli *host* non sono stati configurati staticamente<sup>4</sup>, ogni *host* si autoassegna un *indirizzo link-local*. Ovviamente, per assicurarsi che l'indirizzo scelto non sia già stato preso, ogni *host* trasmette una richiesta *ARP* contenente l'*IP* che si è assegnato. Se nessuno risponde si tiene l'indirizzo, altrimenti ne sceglie un altro e ripete l'operazione.

In realtà, è possibile configurare i *router* in modo che, per i soli messaggi *DHCP*, non blocchino i *pacchetti* con indirizzo 255.255.255.255 consentendo in questo modo l'utilizzo di server *DHCP* risiedenti su reti diverse.

### 4.7 Il viaggio di un pacchetto attraverso la rete

Vediamo ora come viene trattato complessivamente un *pacchetto* dal momento dell'invio fino alla sua ricezione.

Prendiamo la seguente rete:

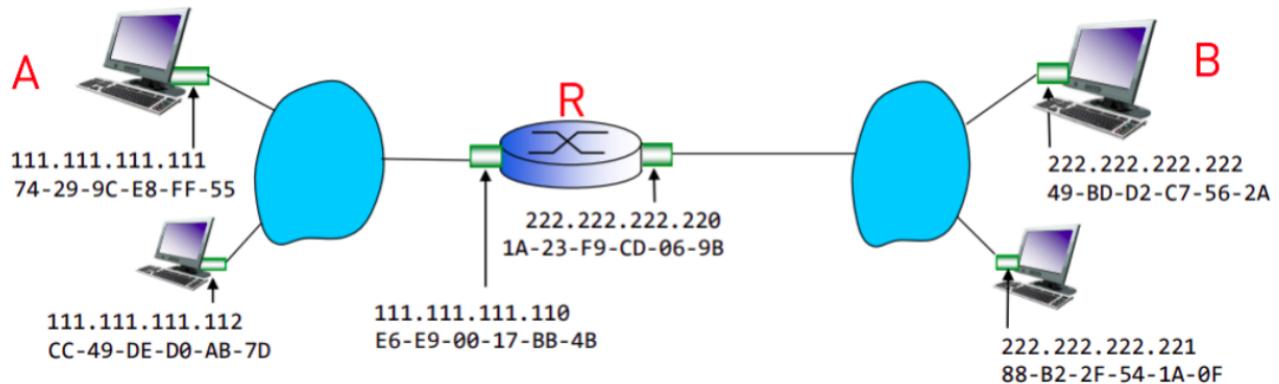


Fig. 4.25: Esempio di rete IP

Supponiamo che l'*host* *A* voglia inviare un *pacchetto* all'*host* *B* e che *A* conosca già l'*indirizzo IP* di *B* e gli *indirizzi IP* e *MAC* del *router*<sup>5</sup>.

Quindi, *A* crea un *pacchetto*, impostando come sorgente il proprio *IP* e come destinazione l'*indirizzo IP* di *B*, e lo incapsula in un *frame* nel quale gli indirizzi di sorgente e di destinazione sono rispettivamente il proprio *MAC* e il *MAC* di *R*.

*R* riceve il *frame*, ne estrae il *pacchetto* e analizzando la propria tabella di inoltro decide verso quale interfaccia inviarlo. In questo caso, sceglie l'interfaccia con indirizzo 222.222.222.220 e con *next hop* "diretto".

A questo punto, se *R* conosce l'*indirizzo MAC* di *B* crea un *frame* indicandolo come indirizzo di destinazione, altrimenti invia un messaggio *ARP* in broadcast sulla rete di *B*, *B* risponde indicando il proprio *MAC* e *R* lo salva nella propria *cache ARP*.

Infine, *B* riceve il *frame* e ne estrae il *pacchetto*.

### 4.8 Protocollo IPv6

Il protocollo *IPv6* è un'evoluzione dell'*IP*, o meglio dell'*IPv4*, ed è nato per ampliare la quantità di indirizzi disponibili, velocizzare l'elaborazione dei *pacchetti* introducendo header di dimensione fissa e migliorare la gestione della "qualità del servizio".

<sup>4</sup>Negli *host* configurati staticamente l'amministratore di rete inserisce tutte le informazioni necessarie

<sup>5</sup>A può conoscere l'*IP* del *router* grazie al *DHCP* e il *MAC* grazie all'*ARP*

#### 4.8.1 Struttura dei pacchetti IPv6

I *pacchetti IPv6* hanno uno o più header di dimensione fissa 40byte e sono organizzati come da figura sottostante: I campi sono molti meno rispetto all'*IPv4* e ce ne sono di nuovi:

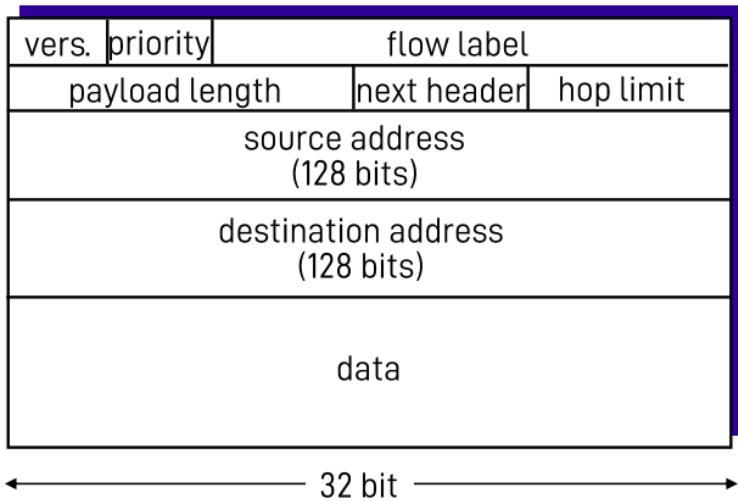


Fig. 4.26: Struttura di un *pacchetto IPv6*

- **Version:** indica la versione del protocollo;
- **Priority:** indica la priorità dei *pacchetti* che fanno parte dello stesso *flusso*;
- **Flow label:** etichetta che identifica il *flusso* di appartenenza dei *pacchetti*;
- **Payload length:** dimensione del payload;
- **Next header:** identifica il protocollo di livello 4 encapsulato nei dati;
- **Hop limit:** numero di *hop* rimasti al *pacchetto*;
- **Source address:** indirizzo sorgente;
- **Destination address:** indirizzo di destinazione;
- **Data:** payload del *pacchetto* o header successivo;

I campi **Checksum** e **Options** sono stati rimossi, ma come già detto, è possibile inserire un secondo header impostando un valore adeguato nel campo **Next header**.

**NB.** È stato modificato anche il protocollo *ICMP*, introducendo l'*ICMPv6* che aggiunge nuovi messaggi di errore, quale ad esempio il *Packet too big*, e funzioni per la gestione dei gruppi *multicast*.

#### 4.8.2 Tunneling tramite IPv4

Poiché molti *router* ancora non supportano l'*IPV6*, per permettere a *IPv4* e *IPv6* di coesistere, è stata introdotto il concetto di *tunneling*, ovvero, è possibile far viaggiare *pacchetti IPv6* attraverso porzioni di rete che non supportano *IPv6*, incapsulandoli all'interno di *pacchetti IPv4*.

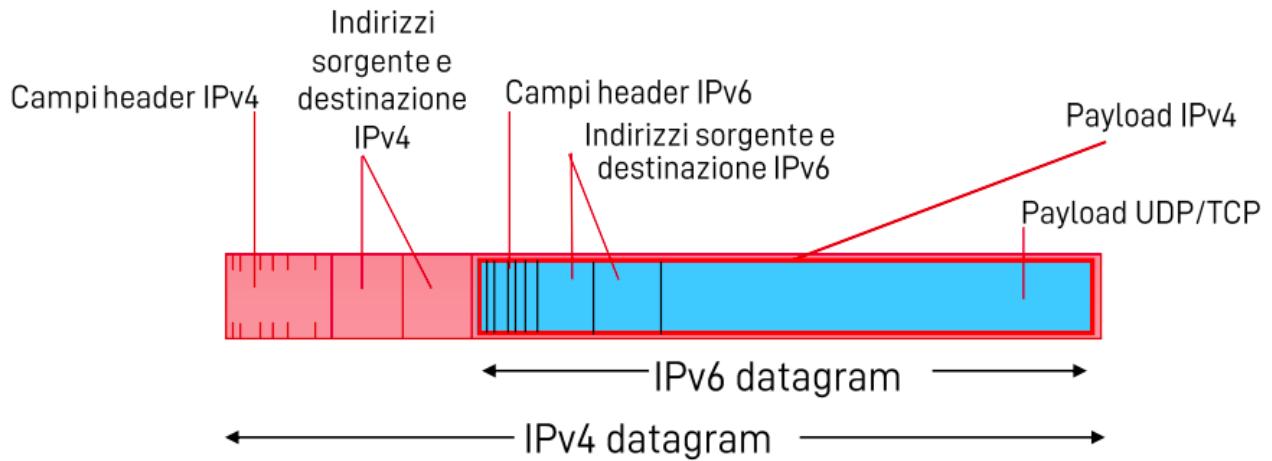


Fig. 4.27: Tunneling IPv6

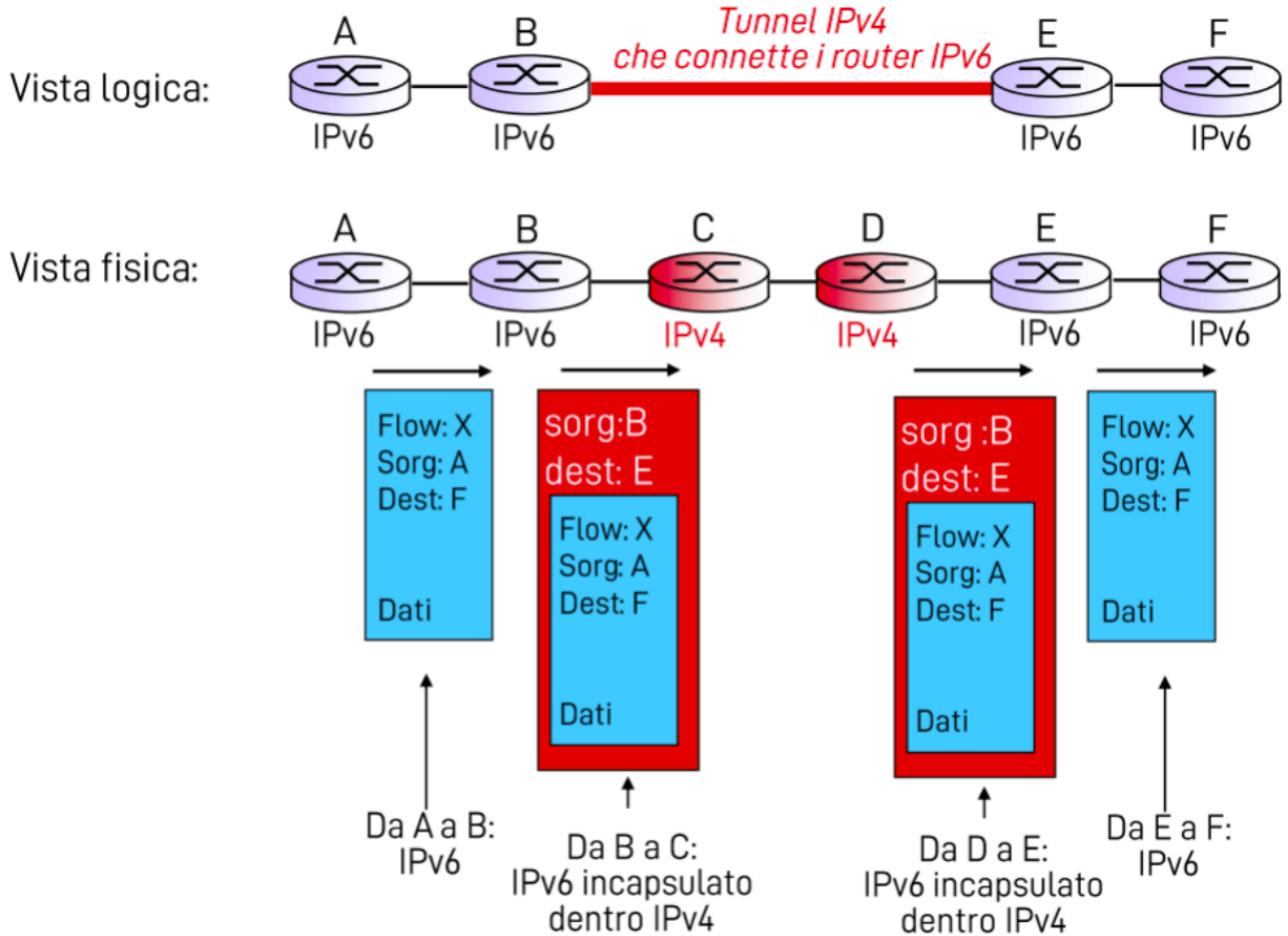


Fig. 4.28: Tunneling IPv6 vista logica e fisica

#### 4.8.3 Struttura e tipologia di indirizzi IPv6

Gli *indirizzi IPv6* sono codificati su 128 bit e a differenza di *IPv4*, vengono rappresentati da 32 cifre esadecimali divise in 8 quartetti, ad esempio:

2a03:2880:f108:0083:face:b00c:0000:25de

Per accorciarli è possibile omettere gli zeri iniziali di un quartetto e indicare con un solo zero un quartetto formato da quattro zeri, ad esempio l'indirizzo di prima può essere espresso come:

2a03:2880:f108:83:face:b00c:0:25de

Qualora vi siano più quartetti consecutivi di zeri, è possibile ometterli indicando al loro posto ::. Ad esempio il seguente indirizzo:

2a03:2880:f108:0000:0000:0000:0000:25de

diventa:

2a03:2880:f108::25de

Tuttavia, se esistono più gruppi di zeri, si può omettere solo il gruppo più lungo, altrimenti non si sarebbe più in grado di stabilire la forma completa dell'indirizzo.

**CIDR IPv6** In *IPv6* prefissi e suffissi sono gestiti col metodo *CIDR* come in *IPv4*, ad esempio la seguente rete:

2a03:2880:f108:83::/64

comprende gli indirizzi da:

2a03:2880:f108:83:0:0:0:0

a:

2a03:2880:f108:83:ffff:ffff:ffff:ffff

Soltanamente il prefisso è a sua volta diviso in *prefisso di routing* che identifica un'azienda o un'organizzazione e un *SubnetID* che identifica una specifica sottorete di quella organizzazione.

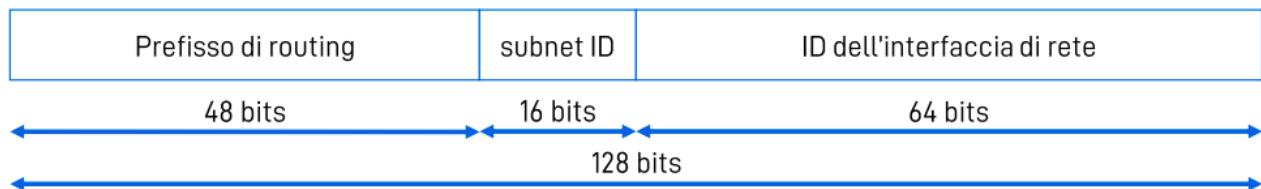


Fig. 4.29: Tipica suddivisione di un *indirizzo IPv6*

**Indirizzi speciali** Anche in *IPv6* esistono gli stessi indirizzi speciali visti nell'*IPv4*, ma la definizione di alcuni è diversa:

- *Indirizzo “questo computer”*: è una stringa di 128 zeri che in notazione semplificata sono rappresentabili come ::/128;
- *Indirizzo loopback*: ce n'è solo uno ed è ::1/128;
- *Indirizzi multicast*: sono tutti gli indirizzi della sottorete ff08::/8;
- *Link-local unicast*: corrispondono agli indirizzi *link-local* dell'*IPv4* e sono tutti quelli della sottorete fe80::/10;

È possibile mappare ogni *indirizzo IPv4* in un *indirizzo Ipv6* semplicemente interpretando i 32 bit dell'indirizzo come 8 caratteri esadecimales e costruendo l'*indirizzo IPv6* come:

::ffff:wwxx:yyzz

dove **ww.xx.yy.zz** sono i bit dell'*indirizzo IPv4*.

Ad esempio, l'*indirizzo Ipv4* 193.175.55.16 in *IPv6* diventa ::ffff:c1af:3710.

# Capitolo Nr.5

---

## Metodi e protocolli di instradamento

---

### 5.1 Principi generali

Finora abbiamo analizzato le procedure di inoltro dei *pacchetti* ipotizzando che i *router* fossero già configurati, cioè ci siamo occupati soltanto del *data pane*. In questo capitolo ci occuperemo invece del *control pane* e in particolare, andremo a vedere quali sono e come funzionano i protocolli che permettono ai *router* di costruire le proprie tabelle di inoltro.

I protocolli che permettono di fare questo sono detti protocolli di instradamento (o di routing) e il loro obiettivo è quello di trovare tutti i “buoni” percorsi da un mittente al destinatario attraverso una rete di *router*.

---

#### Definizione 16 - Percorso.

*Nell’ambito del routing, un percorso è una sequenza di router che un pacchetto deve attraversare per arrivare a una destinazione.*

La definizione di “buono” dipende dalle circostanze e alcune metriche di riferimento, ad esempio, possono essere: il costo economico, la frequenza di trasmissione e il livello di congestione.

#### 5.1.1 Rappresentazione delle reti come grafi non orientati

È molto conveniente, quando si studia una rete, vederla come un grafo non orientato nel quale i nodi corrispondono ai *router* e gli archi ai collegamenti tra essi. In questa trattazione ci riferiremo al grafo di una rete con la notazione  $G = (N, E)$  dove  $N$  è l’insieme dei *router* ed  $E$  l’insieme dei collegamenti.

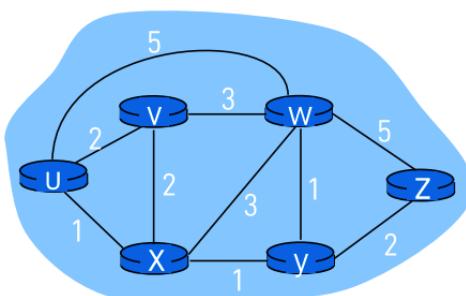


Fig. 5.1: Rete come grafo

Un'altra informazione che ci interessa considerare è il costo di ogni collegamento e di ogni percorso. Il costo di un collegamento è definito in base alla seguente funzione  $c$ :

$$c : N \times N \rightarrow \mathbb{N}$$

Ad esempio, nella figura precedente  $c(U, W) = 5$ . Di nuovo, il costo può essere definito sulla base di parametri diversi: il numero di salti, la *banda* o l'inverso della *banda* del link, la *congestione*, ecc. Algoritmi diversi possono usare parametri diversi.

**NB.** Un costo definito sulla banda di un link è proporzionale al costo monetario di quel collegamento, mentre una definizione basata sull'inverso della banda è proporzionale al tempo di attraversamento.

Il costo di un percorso è espresso dalla funzione *cost*:

$$\begin{aligned} cost : N \times \cdots \times N &\rightarrow \mathbb{N} \\ (n_1, \dots, n_k) &\mapsto \sum_{i=1}^k c(n_i, n_{i+1}) \end{aligned}$$

Ad esempio,  $cost(U, X, Y, Z) = c(U, X) + c(X, Y) + c(Y, Z) = 1 + 1 + 2 = 4$ .

### 5.1.2 Tipi di algoritmi di routing

Gli algoritmi di routing si dividono in base al modo in cui sono distribuite le informazioni sulla rete o in base al modo in cui sono configurate le tabelle di routing.

**Informazioni globali e distribuite** Gli algoritmi basati su informazioni globali partono dall'ipotesi che ogni *router* abbia le stesse informazioni sulla tipologia della rete e i costi dei collegamenti. D'altra parte, nel caso di informazioni distribuite, ogni *router* conosce solo i propri vicini e scambiando informazioni con essi riesce a ricostruire i percorsi migliori. La prima tipologia di algoritmi è detta essere a “*link state*”, mentre la seconda a “*distance vector*”.

**Configurazione statica e dinamica** Le tabelle configurate staticamente sono definite manualmente dall'amministratore di rete e rimangono invariate a meno di ulteriori interventi. In questo caso non vengono neanche usati algoritmi di routing che infatti sono necessari solo nel caso di tabelle generate dinamicamente.

## 5.2 Algoritmo di Dijkstra

L'algoritmo di Dijkstra è un algoritmo *link state* quindi suppone che ogni *router* conosca la topologia della rete e il costo dei collegamenti. Queste informazioni vengono fatte circolare mediante l'invio in broadcast di messaggi.

La tabella di inoltro viene costruita eseguendo più volte l'algoritmo e, ad ogni esecuzione, viene scoperto il percorso di costo minimo verso una destinazione. Di conseguenza, eseguendo  $k$  volte l'algoritmo, si ottengono i percorsi a costo minimo verso  $k$  destinazioni.

Prima di vedere la logica di funzionamento dell'algoritmo è necessario introdurre della notazione specifica:

- $D(v)$ : definisce il costo del percorso verso il *router*  $v$ ;
- $p(v)$ : definisce il predecessore del *router*  $v$  nel cammino verso esso;
- $N'$ : insieme dei *router* per i quali è già stato calcolato il cammino di costo minimo;

**NB.** Il costo  $c(n_1, n_2)$  del collegamento tra  $n_1$  e  $n_2$  vale  $\infty$  se questi non sono collegati.

---

### Frammento 1 - Implementazione algoritmo di Dijkstra.

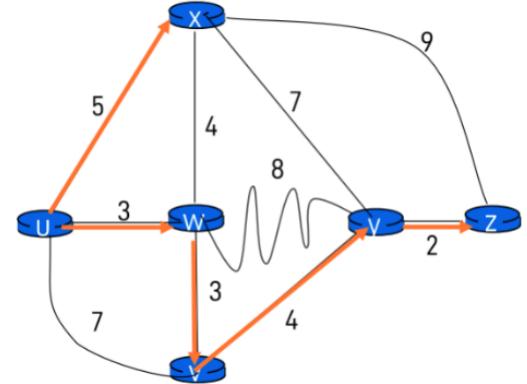
---

```
% Inizializzazione
N' = {u}                                     % N' è inizialmente pari al nodo corrente
foreach (v ∈ N) do
    if (v ∈ u.adj()) then
        D(v) = c(u, v)
    else
        D(v) = ∞
% Loop
do
    NODE w = min(N - N') % Trova w non ancora in N' tale che D(w) sia minimo
    N'.insert(w)          % Aggiunge w a N'
    foreach (v ∈ w.adj() - N') % Per ogni nodo v adiacente a w che non è in N'
        if (D(w) + c(w, v) < D(v)) then
            D(v) = D(w) + c(w, v)           % Aggiorna il costo del percorso verso v
            p(v) = w                         % Aggiorna il predecessore di v
    while (N'.size() < N.size())           % Finché non sono stati inseriti tutti i nodi
```

In sintesi, per ogni nodo  $v$ , se il costo che si paga per arrivare a  $w$  e superare il collegamento da  $w$  a  $v$  è minore del costo che attualmente si paga per arrivare a  $v$ , allora il percorso minimo verso  $v$  diventa il percorso passante per  $w$ . In caso contrario, il percorso già noto per arrivare a  $v$ , è migliore di quello che si avrebbe passando per  $w$ .

Passo	N'	D(v) p(v)	D(w) p(w)	D(x) p(x)	D(y) p(y)	D(z) p(z)
0	U	7,u	3,u	5,u	∞	∞
1	UW	6,w		5,u	11,w	∞
2	UWX	6,w			11,w	14,x
3	UWXV			10,v	14,x	
4	UWXVY				12,y	
5	UWXVYZ					

(a) Passi di costruzione della tabella di routing con *Dijkstra*



(b) Grafo di una rete

Fig. 5.2: Esecuzione dell'algoritmo di *Dijkstra*

**NB.** Questo algoritmo costruisce i percorsi minimi partendo dal *router* di destinazione e procedendo a ritroso sui predecessori.

**NB.** Nel caso risultassero più percorsi minimi dal costo uguale se ne sceglie uno arbitrariamente.

La tabella di inoltro risultante è la seguente:

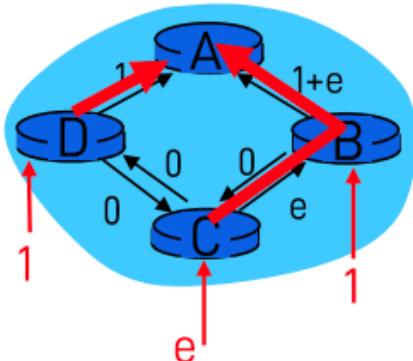
Destinazione	Inoltra a ("next hop")
v	w
w	w
x	x
y	w
z	w

Fig. 5.3: Tabella di inoltro risultante

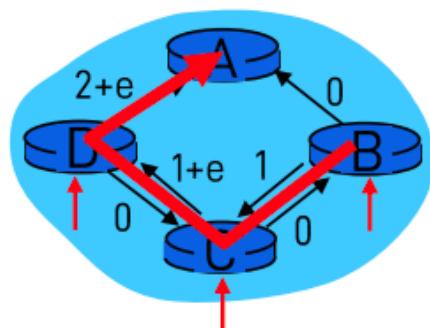
### 5.2.1 Complessità e problemi dell'algoritmo

In una rete con  $n$  nodi (*router*) vengono controllati tutti i nodi  $w \notin N'$  e quindi vengono eseguiti  $\frac{n(n-1)}{2} = O(n^2)$  confronti. Esistono tuttavia implementazioni più efficienti che portano la complessità a  $O(n \log n)$ .

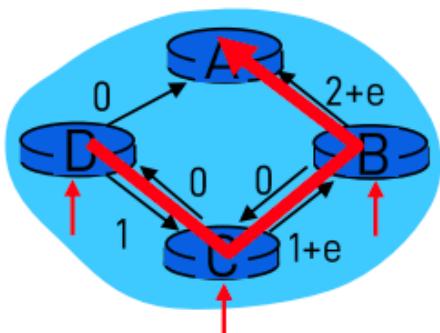
Il problema dell'algoritmo di *Dijkstra* è che se i costi sono definiti male l'algoritmo potrebbe oscillare. Ad esempio, se i costi sono definiti come la quantità di traffico trasportata dai collegamenti, vale:



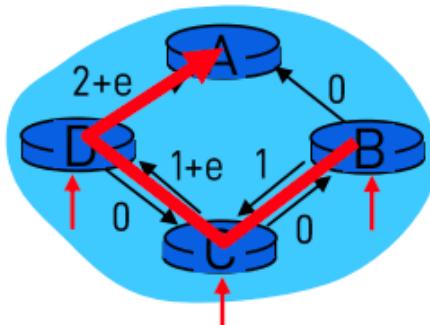
(a) Situazione iniziale:  $C$  genera un traffico di  $e$  byte,  $D$  e  $B$  di 1 byte



(b) Dati i nuovi costi, *Dijkstra* calcola nuovi percorsi generando costi diversi



(c) Nuovamente, dati i nuovi costi, *Dijkstra* calcola altri nuovi percorsi generando costi diversi



(d) Di nuovo ancora, dati i nuovi costi, *Dijkstra* calcola altri nuovi percorsi generando costi diversi

Fig. 5.4: Esempio di oscillazione dell'algoritmo di *Dijkstra*

## 5.3 Protocollo OSPF

### 5.3.1 Autonomous system

La rete internet è organizzata in moltissime sottoreti ognuna delle quali è di proprietà di una qualche entità più o meno grande (e.g. *ISP*, operatori di rete, aziende, ...). Idealmente, ciascuna di queste sottoreti è amministrativamente autonoma, ovvero configurabile liberamente dall'amministratore, e capace di collegarsi alle altre sottoreti. Queste caratteristiche hanno permesso la suddivisione di internet in *AS*:

---

#### Definizione 17 - AS - Autonomous System.

---

*Un autonomous system è un gruppo di router appartenenti ad uno stesso controllo amministrativo e identificato univocamente da un numero*<sup>1</sup>.

Questa suddivisione permette di scomporre il problema della generazione delle tabelle di inoltro in due sotto problemi: gestire il routing all'interno di un *AS* e tra *AS* diversi. A questo punto è lecito parlare di *Intra-AS routing* e *Inter-AS routing*.

Nel contesto dell'*Intra-AS routing* è stato introdotto il protocollo *OSPF*, un protocollo di tipo *link state* e che si serve dell'algoritmo di *Dijkstra* per il calcolo dei percorsi.

Una particolarità di questo protocollo è che non si serve del *livello di trasporto* per consegnare i pacchetti, bensì sfrutta l'indirizzo *multicast* 224.0.0.5.

### 5.3.2 Funzionamento del protocollo

L'*OSPF* è un protocollo molto semplice e infatti si compone soltanto di tre procedure:

1. *Protocollo di “Hello”*: è una procedura che gestisce lo scambio di messaggi di mantenimento che servono a testare i collegamenti per vedere quali sono ancora attivi e quindi capire anche quali tra i *router* adiacenti sono ancora raggiungibili;
2. *Protocollo di “Exchange”*: viene usato per comunicare ai *router* adiacenti con i quali si è appena entrati in contatto la topologia conosciuta della rete;
3. *Protocollo di “Flooding”*: viene usato per informare tutti i *router* di un cambiamento avvenuto nello stato dei collegamenti;

In particolare, la procedura di *flooding* prevede che un *router* trasmetta su tutte le sue interfacce un messaggio e che tutti gli altri *router* facciano lo stesso. Ovviamente, quando ciò avviene, il messaggio non viene ritrasmesso dall'interfaccia che l'ha ricevuto. Questa procedura viene detta essere di *flooding controllato* e comporta che vengano inviati tanti messaggi quanti sono i collegamenti o i *domini di broadcast*.

### 5.3.3 OSPF gerarchico

In reti con molti *router*, per ridurre il numero di messaggi inviati, si divide la rete in modo gerarchico. La gerarchia che si va a costituire ha due livelli: una dorsale, o *backbone*, e le reti di area.

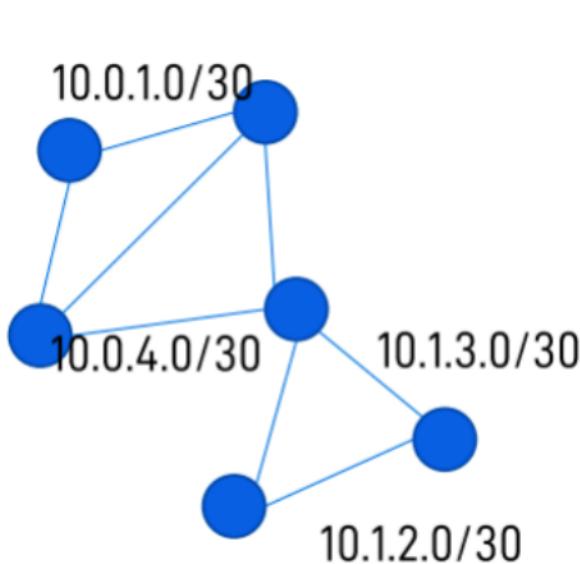
In questo modo, i messaggi circolano solo all'interno delle reti di area e i *router* conoscono solo la topologia della propria area e il cammino più breve verso le altre. Nello specifico, sono

---

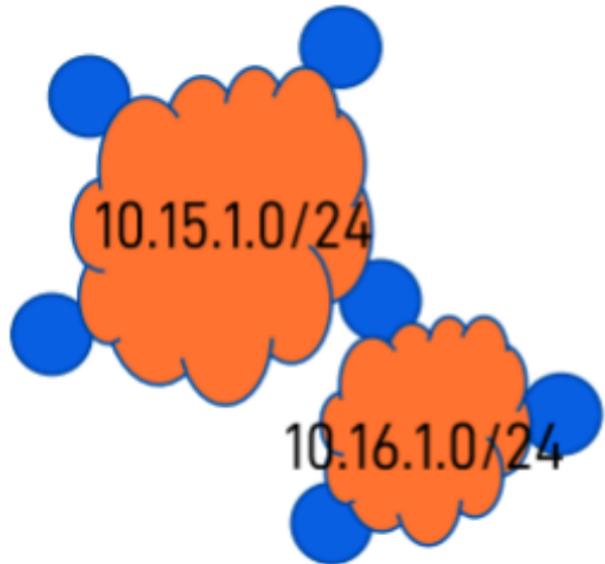
<sup>1</sup>I numeri identificativi sono assegnati centralmente dai registri regionali dell'*ICANN*

sono i *router di bordo* di ogni area a conoscere le informazioni relative alle rotte verso le reti interne alla propria area. Queste sono le informazioni che l'*OSPF* comunica a tutti gli altri *router di bordo* delle altre aree.

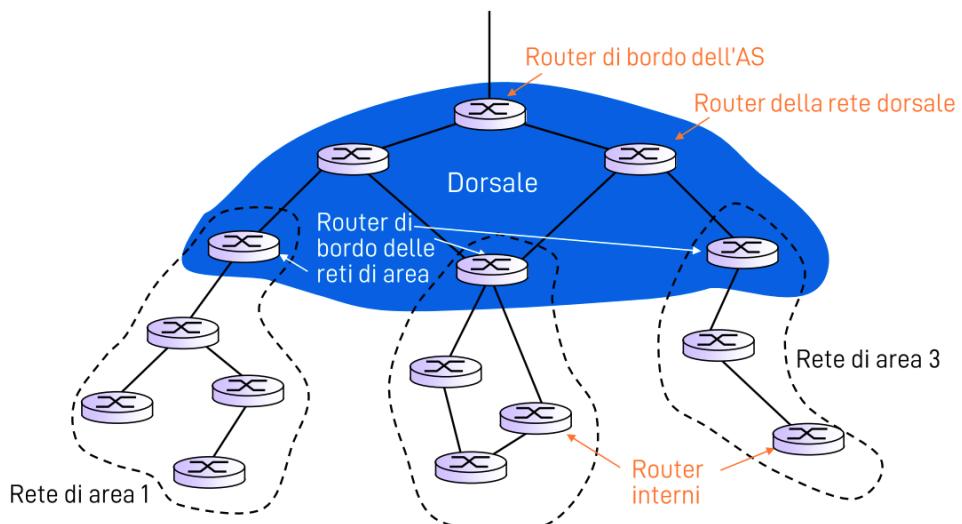
A sua volta, anche la dorsale è un'area a se stante e i *router* al suo interno comunicano mediante *OSPF*.



(a) 8 messaggi inviati



(b) 2 messaggi inviati



(c) I messaggi circolano solo all'interno delle reti di area

### 5.3.4 Traffic engineering

Poiché l'*OSPF* è un protocollo *link state* nel quale l'unica cosa che influenza la scelta di un percorso è il costo dei collegamenti. Gli amministratori di un *AS* potrebbero “modificare” il costo di alcuni collegamenti in modo da indurre il protocollo a indirizzare il traffico verso quelle rotte.

Questa possibilità inverte le relazioni di causa-effetto del protocollo, ovvero, dato un obiettivo di *traffic engineering* è possibile agire sul costo di un collegamento per raggiungere quell'obiettivo.

## 5.4 Algoritmo di Bellman-Ford

Il *Bellman-Ford* è un algoritmo di tipo *distance vector* e quindi parte dall'ipotesi che ogni *router* sappia soltanto quali *router* gli sono vicini e quanto costa raggiungerli. La conoscenza della rete perciò, non è globale, ma distribuita e ogni *router* scambia informazioni con i propri vicini per capire come raggiungere reti più distanti.

### 5.4.1 Logica di funzionamento

Introduciamo la seguente notazione:

- $N_x$ : è l'insieme dei vicini del *router*  $x$ ;
- $R_x$ : è la tabella di inoltro del *router*  $x$ :
  - $R_x[d]$ : è la riga della tabella relativa alla destinazione  $d$ :
    - \*  $R_x[d].cost$ : costo per raggiungere  $d$ ;
    - \*  $R_x[d].nexthop$ : indirizzo di *next hop*;
    - \*  $R_x[d].time$ : riferimento temporale al momento in cui il percorso è stato impostato;
- $D_x$ : *distance vector* del *router*  $x$ , ovvero il vettore contenente tutte le destinazioni raggiungibili da  $x$  e i relativi costi;

**NB.** I costi dei collegamenti sono anche detti distanze o metriche.

**NB.** Viene salvato l'istante di scoperta di un percorso in modo che quelli troppo vecchi possano essere scartati.

### Frammento 2 - Implementazione algoritmo Bellman-Ford.

```
% Inizializzazione
foreach (n ∈ Nx) do
    Rx[n].cost = c(x,n)                                % Per tutti i vicini n di x
    Rx[n].nexthop = n                                 % Imposta il costo per raggiungere n
    Rx[n].time = now()                               % Imposta n come next hop

% Ogni T secondi
<NODE, int> Dx = new <NODE, int>[Rx.size()]      % Distance vector di x
foreach (d ∈ Rx) do                                % Per tutte le destinazioni d conosciute da x
    Dx[d] = <d, Rx[d].cost>                      % Popola il distance vector
foreach (n ∈ Nx) do
    send(Dx, n)                                    % Per tutti i vicini n di x
                                                % Invia il distance vector

% Quando si riceve il distance vector Dy di un vicino y
foreach (<d, cost> ∈ Dy) do % Per ogni coppia destinazione-costo del vettore
    if (d ∉ Rx or cost + c(x, y) < Rx[d] or y == Rx[d].nexthop) then
        Rx.cost = cost + c(x, y)                      % Imposta il costo per raggiungere la d
        Rx.nexthop = y                                % Imposta il next hop
        Rx.time = now()                             % Imposta il tempo di scoperta
```

### Esempio 9 - Esempio di esecuzione dell'algoritmo Bellman-Ford.

Inizialmente ogni router conosce solo le rotte che lo collegano direttamente ai propri vicini.

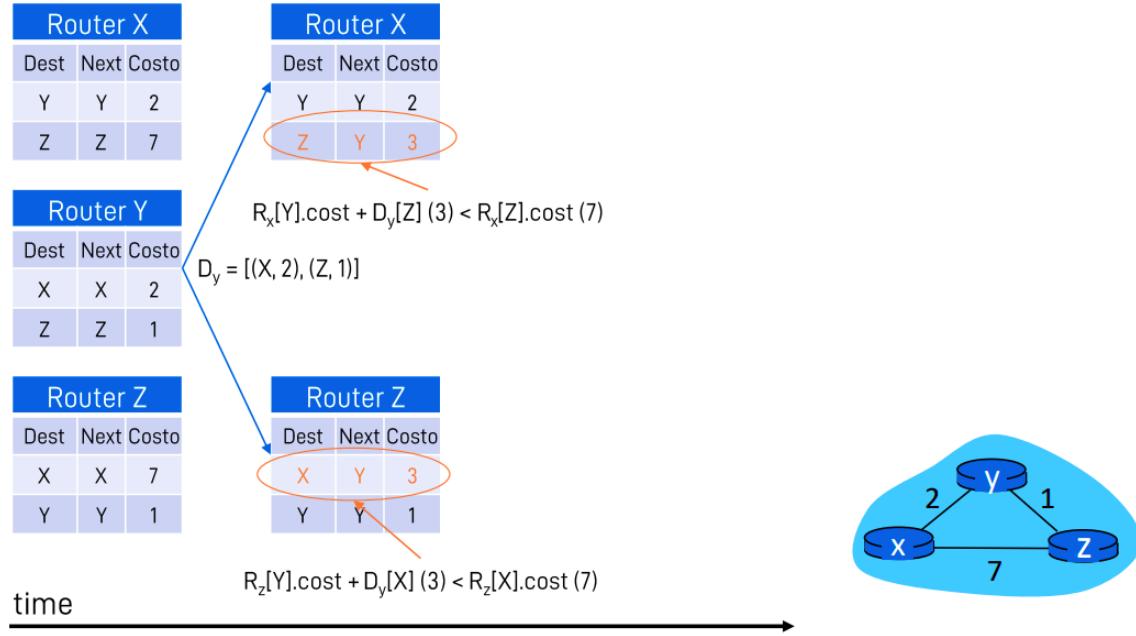


Fig. 5.5: Y trasmette il proprio *distance vector*

Dopo che Y ha trasmesso il proprio *distance vector*, X e Z confrontano le rotte ricevute con quelle che hanno già. Poiché il collegamento tra X e Z costa 7, mentre passando per Y si ottiene un cammino di costo 3, X e Z aggiornano le proprie tabelle modificando rispettivamente le rotte verso Z e verso X, sostituendole con un nuovo percorso passante per Y.

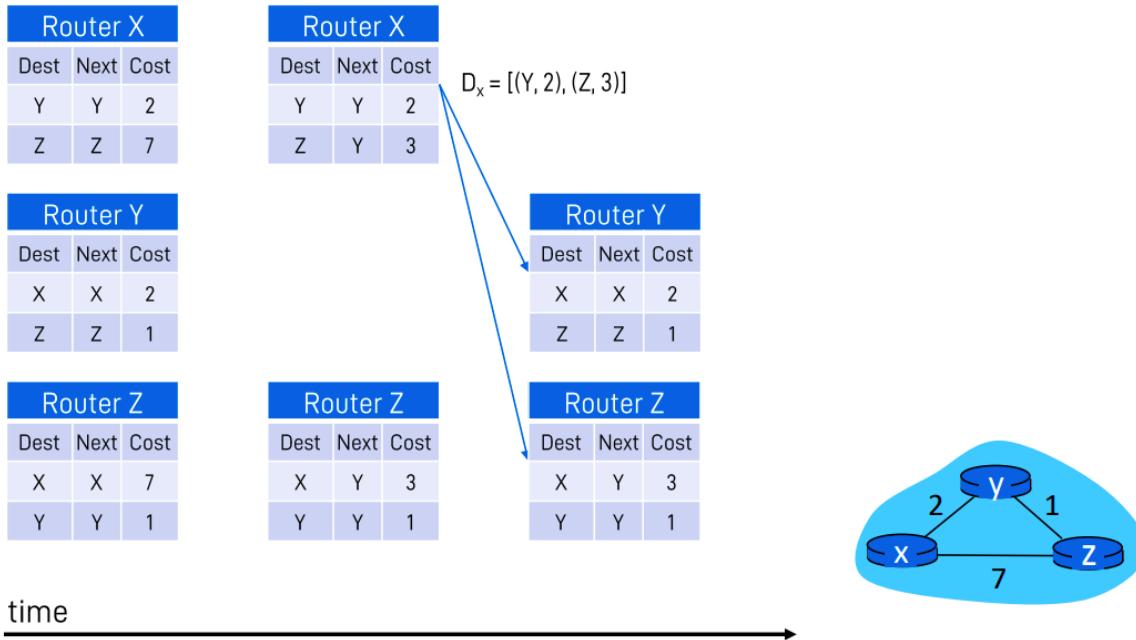


Fig. 5.6: X trasmette il proprio *distance vector*

Dopo che X ha trasmesso il proprio *distance vector*, né Y, né Z hanno necessità di aggiornare le proprie tabelle in quanto non scoprono destinazioni nuove e nemmeno percorsi migliori per le destinazioni già conosciute.

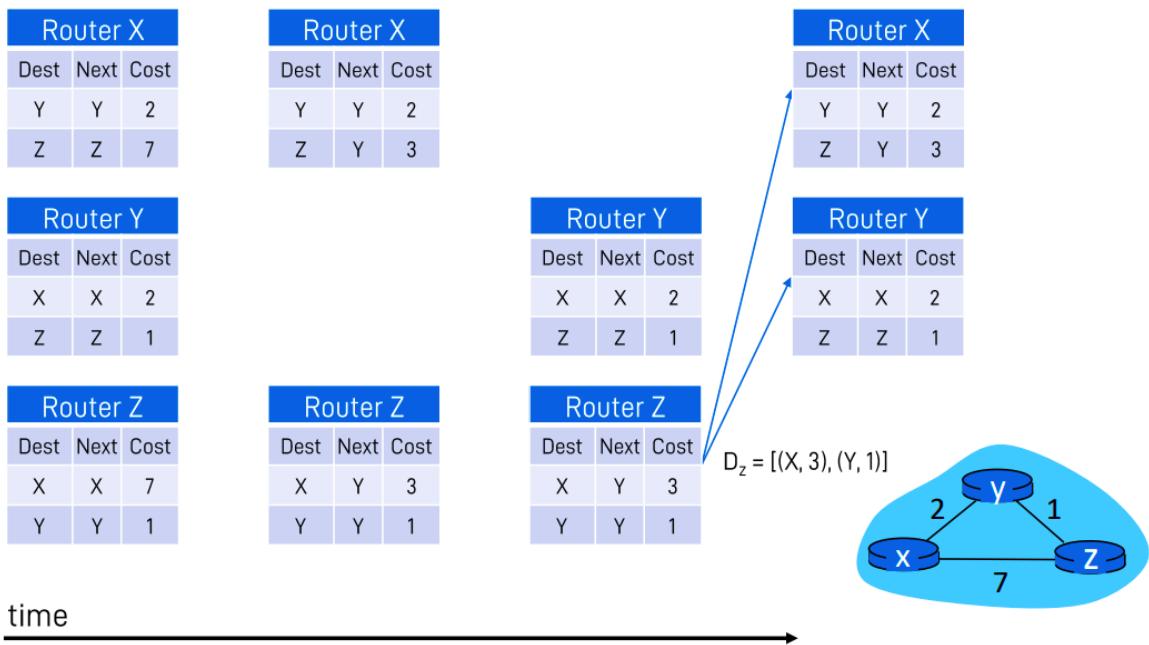


Fig. 5.7: *Z* trasmette il proprio *distance vector*

Infine, *Z* trasmette il proprio *distance vector*, ma anche in questo caso non c'è necessità di aggiornare le tabelle. A questo punto, tutti i router della rete sanno come raggiungersi nel modo più economico possibile.

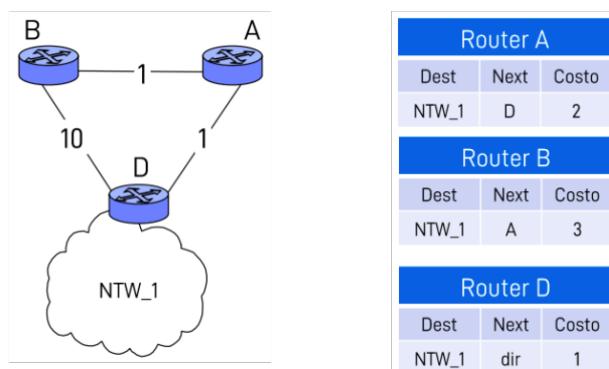
### 5.4.2 Problema del count-to-infinity

Il problema del *count-to-infinity* si verifica quando cade un collegamento di uno dei cammini brevi. Gli *host* che utilizzavano quel collegamento e che non sono ad esso direttamente collegati, non vengono informati del suo cambiamento di stato e quindi continuano ad inoltrare *pacchetti* attraverso quel percorso. Quando però il *router* collegato a quel link deve decidere dove inoltrare i *pacchetti*, sapendo di non poter usare quel collegamento, li ritrasmette al mittente. Il mittente a quel punto, aggiorna il costo di quel percorso e, se rimane il più conveniente, riprova ad utilizzarlo. Questo scambio continua fino a quando il percorso fallace risulta più conveniente di altri percorsi "sani". Quando finalmente il percorso difettoso risulterà non più conveniente, i *router* ne sceglieranno uno diverso.

---

#### Esempio 10 - Problema del count-to-infinity.

Si consideri la seguente configurazione di rete:



Supponiamo che il collegamento di costo 1 tra A e D si rompa. Se B deve inviare un pacchetto alla rete NTW\_1, lo inoltra verso il router A. Non sapendo a chi inviarlo, A lo ritrasmette a B.

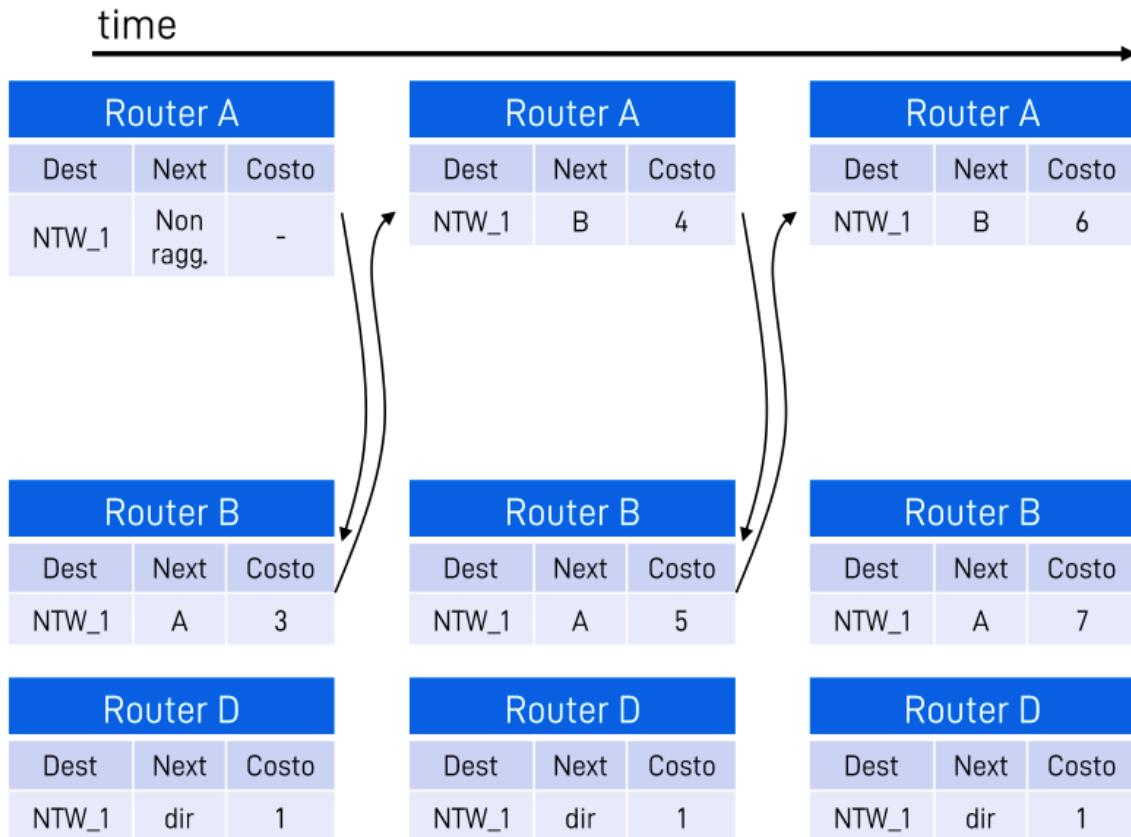


Fig. 5.8: Loop del *count-to-infinity*

Dopo molti scambi, il collegamento di costo 10 tra B e D risulterà più conveniente di quello che passa per A col risultato che B smetterà di reinoltrare il pacchetto ad A, rompendo il loop del *count-to-infinity*.

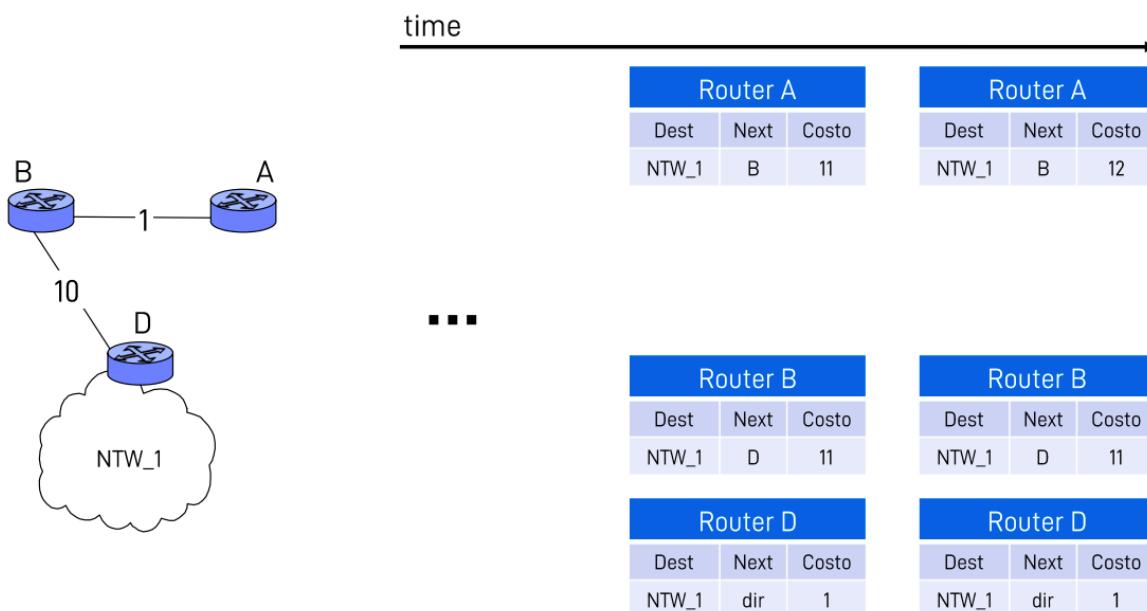


Fig. 5.9: Termine del loop e aggiornamento dei cammini brevi

**Soluzioni al count-to-infinity** Fortunatamente, esistono delle procedure che permettono di evitare l'insorgere di questo problema:

- *Limite al numero di hop*: viene impostato un limite al numero di *hop* (tipicamente 15) dei *pacchetti* che trasportano i *distance vector*. Questo permette di ridurre il tempo di convergenza;
- *Split horizon*: quando un *router* manda ad un vicino aggiornamenti al costo dei percorsi, omette quelli appresi da quello stesso vicino;
- *Poisoned reverse*: dati tre *router* *X*, *Y* e *Z*, fino a quando *X* raggiunge *Z* passando per *Y*, *X* comunica ad *Y* che  $D_x(Z) = \infty$ ;

**NB.** Nell'esempio precedente, con lo *split horizon*, *B* non avrebbe incluso la riga (NTW\_1, 3) nel proprio *distance vector* perché l'aveva appresa proprio da *A*.

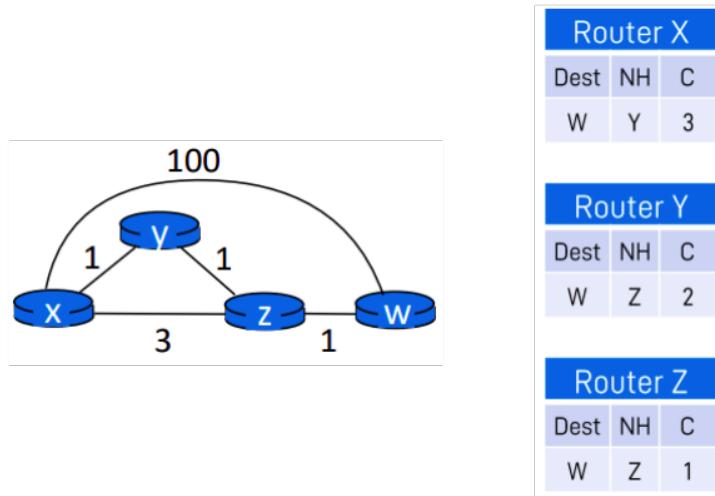
**NB.** Sempre nell'esempio precedente, se fosse stato utilizzato il *poisoned reverse*, *B* avrebbe incluso la riga (NTW\_1,  $\infty$ ) nel *distance vector* inviato ad *A* perché *B* raggiunge NTW\_1 passando per *A*. Questa riga avrebbe portato *A* a scegliere di non inoltrare di nuovo il *pacchetto* a *B*.

In generale possiamo affermare che le informazioni relative a miglioramenti nei costi dei cammini si propagano molto più velocemente rispetto alle informazioni sul peggioramento dei costi.

Purtroppo però, le procedure di *split horizon* e *poisoned reverse* non funzionano sempre, ma sono influenzate dall'ordine di invio dei *distance vector*. Infatti, poiché i *router* agiscono in modo indipendente tra loro, non è noto a priori l'ordine di invio dei *distance vector* e questo può portare allo sviluppo di cicli.

### Esempio 11 - Esempio di generazione di cicli con split horizon.

Si prenda la seguente configurazione di rete:



Supponiamo che il collegamento tra *Z* e *W* si rompa. Se adesso *X* trasmette a *Y* e *Z* il proprio *distance vector*, utilizzando la procedura di *split horizon*, si viene a creare un ciclo.

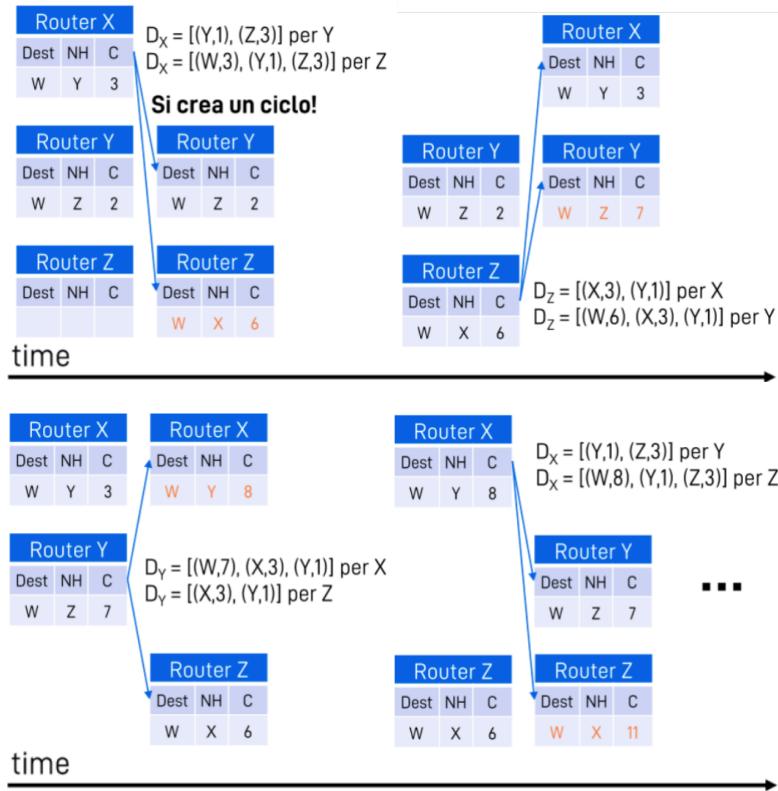


Fig. 5.10: Sviluppo del ciclo

Alla fine, il collegamento tra  $X$  e  $W$  di costo 100 risulterà conveniente e, quindi,  $X$  propagherà quell'informazione a tutti i suoi vicini terminando il ciclo.

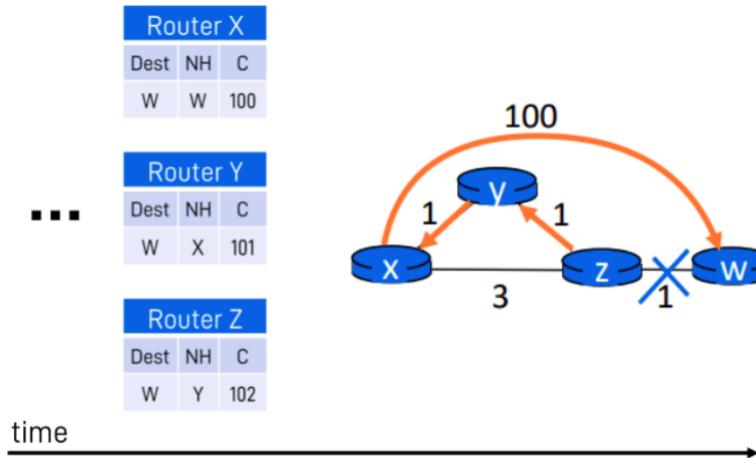


Fig. 5.11: Fine del ciclo

## 5.5 Protocollo RIP

Il **RIP** è un protocollo per il *routing intra-AS*, cioè interno a un *Autonomous System* ed è implementato utilizzando i *distance vector*. Il *RIP* è un protocollo semplice da implementare e gestire, ma è adatto solo a reti con una dimensione ridotta e comunque la convergenza è abbastanza lenta.

Il principio di funzionamento è molto semplice: ogni 30 secondi, o quando cambiano le tabelle di routing, il *RIP* trasmette, sulla porta 520 e all'indirizzo multicast 224.0.0.9, un

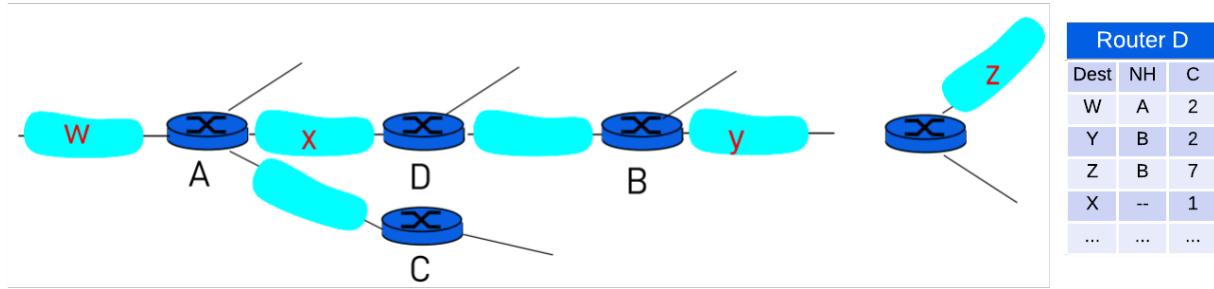
messaggio di *RIP advertisement*. Questo messaggio contiene un elenco che comprende fino a 25 sottoreti di destinazione all'interno dell'*AS* e la distanza del mittente rispetto a ciascuna sottorete.

Nel *RIP*, il costo di un percorso è dato dal numero di *hop* necessari ad attraversarlo e per limitare il tempo di convergenza viene impostato un limite di 15 *hop* e vengono considerati di costo infinito i percorsi che ne richiedono 16 o più. È comunque possibile che un *hop* abbia un costo più che unitario.

**NB.** Il *RIP* si serve di *UDP* per trasportare i messaggi.

### Esempio 12 - Esempio di funzionamento del RIP.

Si considerino la seguente rete e la tabella di inoltro del router D:



Dopo che A ha trasmesso il proprio *RIP advertisement*, D può modificare la propria rotta verso la sottorete Z impostando quella passante per A che ha un costo minore.



Fig. 5.12: Adattamento della tabella di inoltro

**Gestione dei guasti** Se un *router* non riceve notizie da un suo vicino per 3 minuti, quel vicino e il collegamento con esso vengono considerati guasti. Di conseguenza, il *router* modifica la propria tabella di instradamento locale e propaga l'informazione a tutti gli altri *router* adiacenti. Se il *RIP advertisement* porta i vicini a modificare le proprie tabelle di inoltro, quei *router* trasmettono a loro volta i propri *RIP advertisement* così da diffondere l'informazione su tutta la rete. Viene usata la tecnica del *poisoned reverse* per evitare loop.

**Gestione delle tabelle di inoltro** Come accennato in precedenza, il *RIP* è implementato come un protocollo di *livello Applicativo* e quindi usa una *socket* come interfaccia di trasmissione e ricezione. In realtà però, il protocollo *RIP* viene eseguito da un processo chiamato *routed* che mantiene le informazioni di instradamento e scambia i messaggi con i vicini.

## 5.6 Confronto tra algoritmi link state e distance vector

Negli algoritmi di tipo *link state* i messaggi vengono inviati ad ogni *router* e attraverso ogni collegamento, quindi se  $n$  è il numero di *router* ed  $E$  il numero di collegamenti, il numero di messaggi inviati è  $O(nE)$ . D'altra parte, gli algoritmi a *distance vector* comunicano solo con i propri vicini, ma come abbiamo visto, i costi dipendono dall'organizzazione della rete.

Per quanto riguarda i tempi di convergenza invece, gli algoritmi a *link state* sono soggetti a oscillazioni, mentre quelli a *distance vector* soffrono del problema del *count-to-infinity* e potrebbero generare cicli.

Infine, relativamente al comportamento in caso di guasti, con algoritmi *link state* i *router* gestiscono la propria tabella in modo indipendente, ma potrebbero trasmettere informazioni sbagliate sui costi. Anche utilizzando i *distance vector* i *router* potrebbero comunque comunicare costi sbagliati e poiché la tabella di ogni *router* è usata anche dagli altri, gli errori si propagherebbero attraverso la rete.

## 5.7 Protocollo BGP

Il *BGP* è un protocollo per la comunicazione *Inter-AS* che costituisce lo standard de facto nella comunicazione in internet. Prima di vederne nel dettaglio il funzionamento però, è bene ritornare sul concetto di *AS*.

Gli *AS* comunicano tra loro per condividere informazioni di raggiungibilità, ma ogni *AS* può decidere in autonomia quali sono i suoi punti di ingresso e di uscita e anche quali informazioni condividere con ciascun vicino.

Esistono anche *AS* particolari che non condividono informazioni proprie, ma fungono semplicemente da tramite per altri *AS*.

### 5.7.1 Principi di funzionamento

*BGP* è un protocollo del *livello Applicativo* e infatti utilizza il *TCP* per connettere coppie di *nodi* detti *BGP speaker*. Tra due *BGP speaker* possono esistere più connessioni contemporaneamente.

**Protocollo path vector** Il *BGP* è un protocollo di tipo *path vector* e ciò significa che oltre alle informazioni di raggiungibilità di una rete comunica anche il percorso che usa per raggiungerla. Ad esempio, un *path vector* è qualcosa come la seguente:

Destination	Next Hop	Path
14.8.0.0/16	AS 15	{15, 32, 571, 11}

Fig. 5.13: Esempio di *path vector*

L'idea alla base del *BGP* infatti, è che un *nodo* non condivida solo l'informazione su quali destinazioni può raggiungere, ma anche il percorso che utilizza.

In particolare, grazie a *BGP*, un *AS* in qualsiasi momento può condividere e ritirare le informazioni di raggiungibilità verso una o alcune delle proprie reti interne.

**Macchina a stati del BGP** Il funzionamento del *BGP* può essere descritto da una macchina a stati:

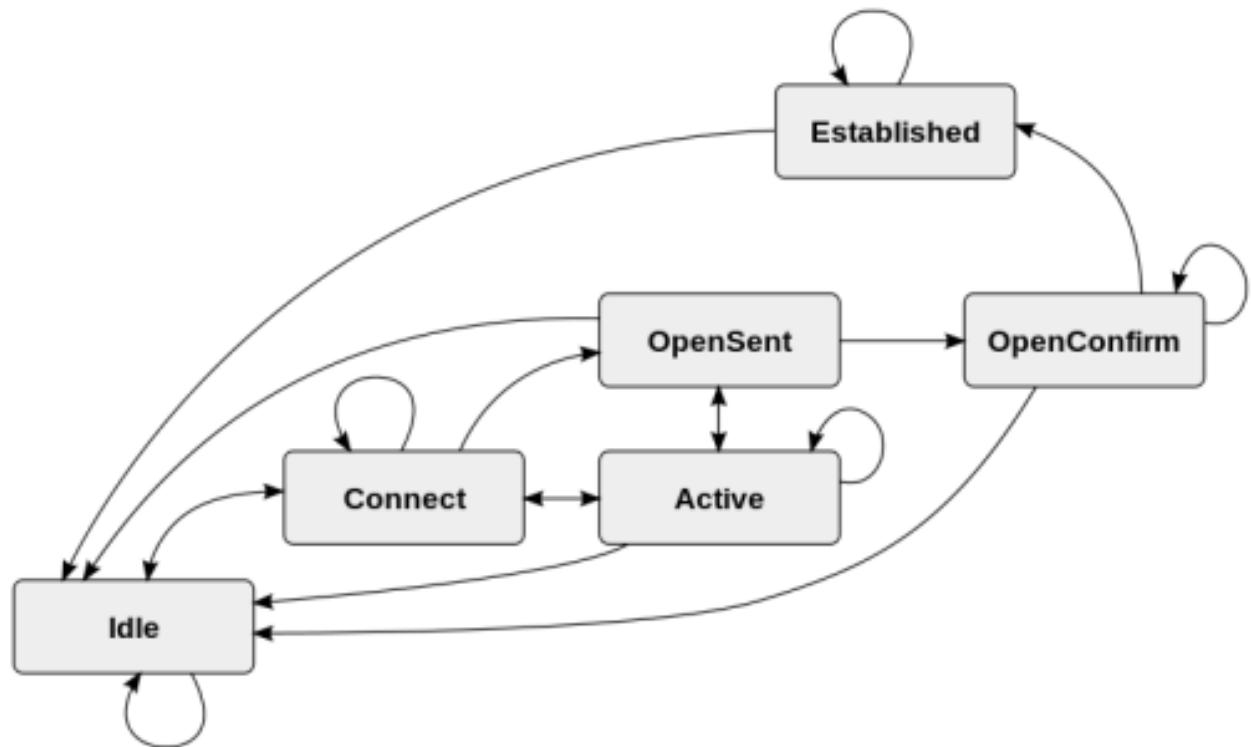


Fig. 5.14: Macchina a stati del *BGP*

**Interconnessioni e policy** Esistono tre tipi di connessioni tra *AS*:

- *Client-Provider*: il client paga il provider per essere raggiungibile;
- *Provider-Client*: il provider viene pagato per garantire la raggiungibilità di un client;
- *Peer*: due nodi sono peer quando condividono tra loro le rispettive informazioni di raggiungibilità;

Nello specifico, le interconnessioni sono regolate da contratti o *policy* che stabiliscono quali informazioni possono essere condivise con un certo vicino e quali no. Esistono due tipi di *policy*:

- *Ingress policies*: stabiliscono quali informazioni possono entrare all'interno dell'*AS*;
- *Egress policies*: stabiliscono quali informazioni possono uscire dall'*AS*;

**Best path BGP** Il concetto di *best path* nel *BGP*, o percorso migliore, è diverso da quello del routing classico. Nel routing classico si cerca il percorso più breve o meno costoso, nel *BGP* invece, non c'è un parametro generale, ma dipende dal singolo *AS*.

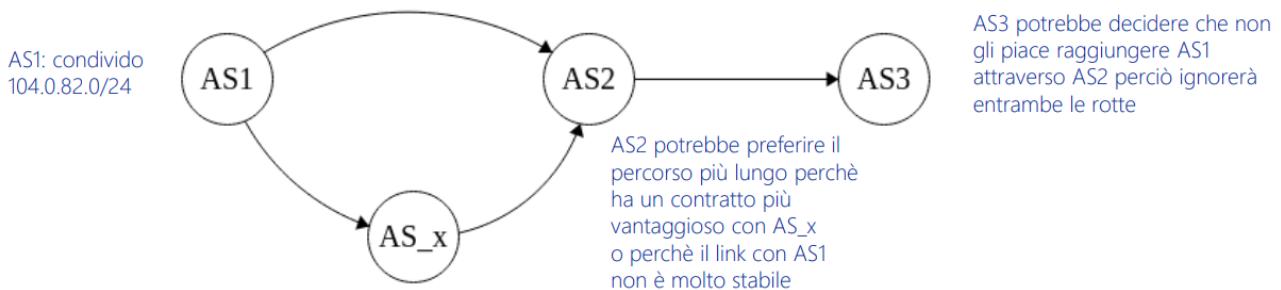


Fig. 5.15: Esempio di *best path BGP*

Comunque, quando due *BGP speaker* si scambiano tra loro le informazioni di raggiungibilità per le destinazioni, condividono soltanto i *best path* verso quelle destinazioni. Ovviamente, se un *nodo* decide di non voler attraversare certi *AS*, i percorsi che li includono non verranno mai considerati *best path* da quel *nodo* e quindi non verranno neanche condivisi.

**NB.** Il *best path* potrebbe non essere il percorso più veloce.

### 5.7.2 Messaggi BGP

Il protocollo *BGP* usa un sistema di messaggistica basata su quattro tipi di messaggi: *Open*, *Notification*, *KeepAlive* e *Update*.

**Messaggio Open** L'*Open* è un messaggio usato per aprire una connessione *BGP*. Se il messaggio viene accettato il destinatario risponde con un messaggio di tipo *KeepAlive* e soltanto a quel punto la connessione verrà considerata aperta.

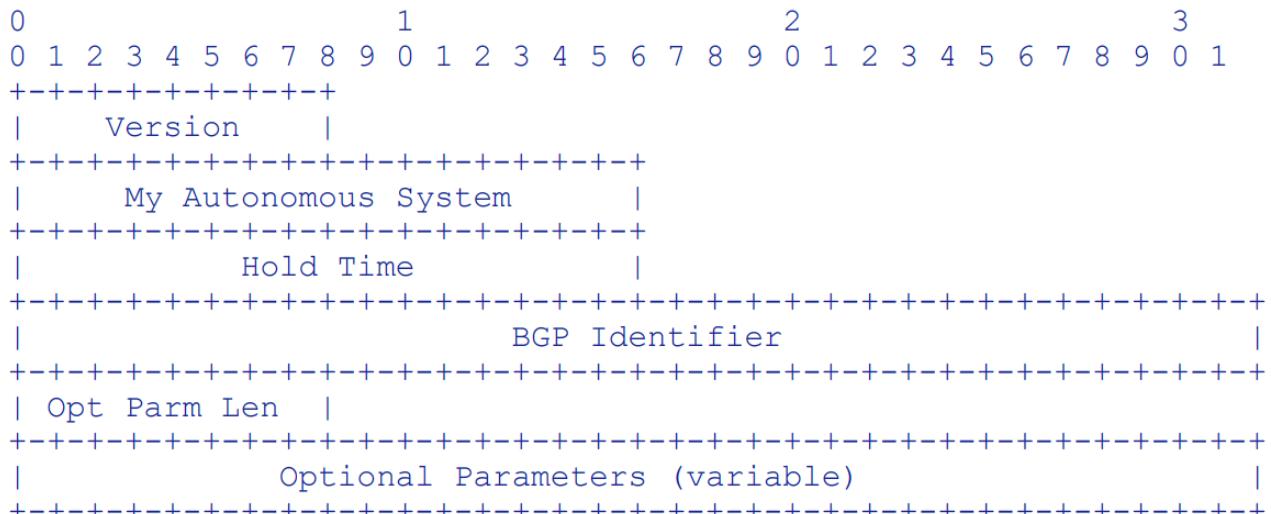


Fig. 5.16: Struttura di un messaggio di tipo *Open*

Il campo *Hold Time* indica per quanto tempo mantenere aperta la connessione. Ognuno dei due *BGP speaker* propone un *Hold Time* e viene scelto il minore. Il campo *BGP Identifier* contiene l'*indirizzo IP* identificativo dello *speaker* ed è uguale per tutte le sue interfacce.

**Messaggio Notification** Questo messaggio serve soltanto a informare un *BGP speaker* del verificarsi di un errore.

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1
-----	-----	-----	-----
Error code	Error subcode	Data (variable)	
-----	-----	-----	-----

Fig. 5.17: Struttura di un messaggio di tipo Notification

**Messaggio KeepAlive** Lo scopo dei messaggi di **KeepAlive** è impedire lo scadere dell’**Hold Time** di una connessione. Possono anche essere inviati per chiudere una connessione indicando un **Hold Time** di 0 secondi. Per non congestionare la rete, questi messaggi consistono soltanto di un header *BGP* di 19 byte senza ulteriori contenuti.

**Messaggi Update** Gli Update contengono le informazioni di raggiungibilità e gli attributi correlati e sono i messaggi responsabili per la diffusione delle informazioni. In particolare, le informazioni trasportate possono essere di due tipi:

- *Additive*: trasportano informazioni relative a nuovi percorsi;
  - *Sottrattive*: rimuovono dei percorsi esistenti;

L'azione di rimozione di una rotta è detta *withdraw* e fa uso di una specifica sezione dei messaggi *BGP* nella quale vengono indicate le rotte da cancellare. I *withdraw* vengono usati quando una destinazione non è più raggiungibile attraverso nessun percorso.



(a) Situazione iniziale

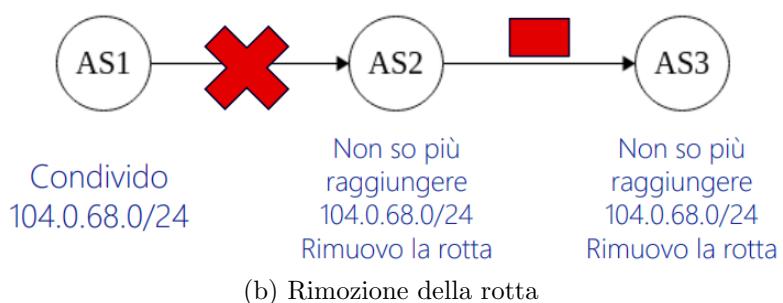


Fig. 5.18: Utilizzo dei messaggi di `Update` con `withdraw`

Quando un *AS* apprende una rotta migliore per una destinazione per condividere l'informazione con i suoi vicini può trasmettere un **Update** con il *withdraw* della rotta precedente e un secondo **Update** con la rotta nuova.

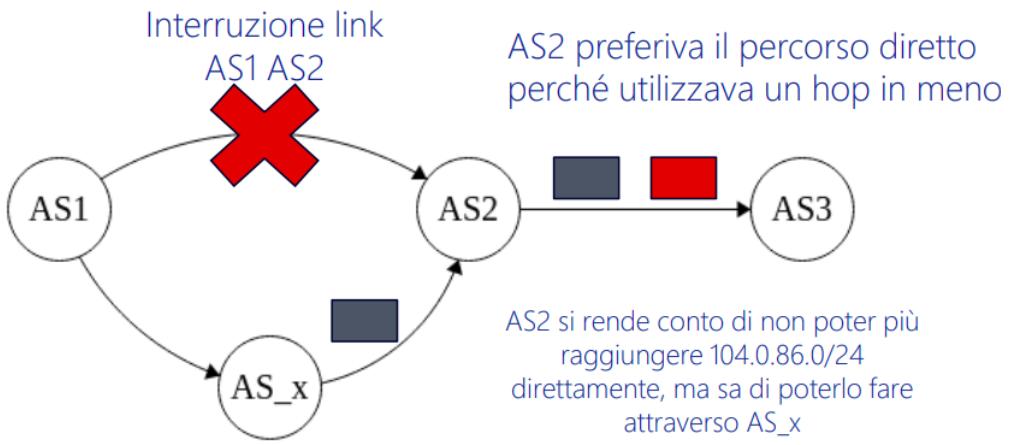


Fig. 5.19: Aggiornamento di una rota

Ciò può essere fatto anche con un solo messaggio usando il *withdraw implicito*.

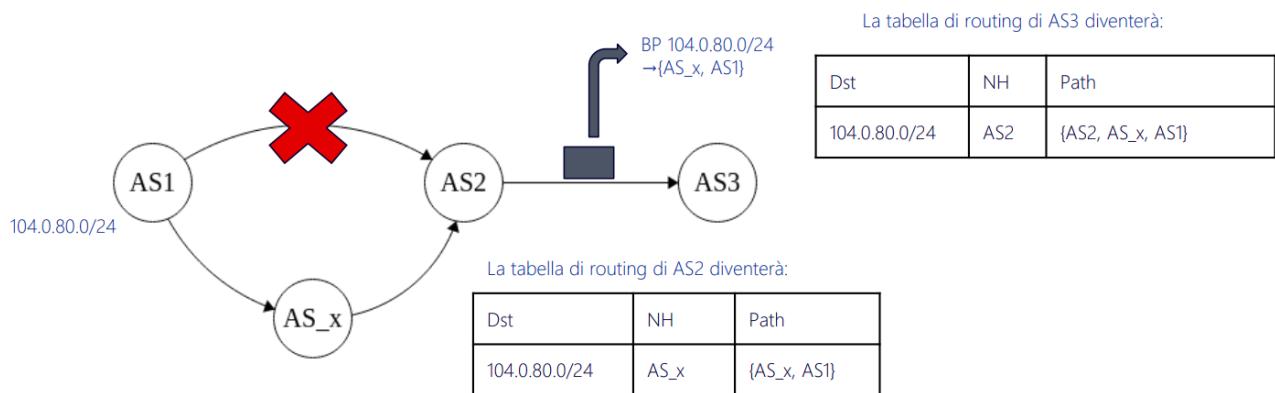


Fig. 5.20: Aggiornamento di una rota con *withdraw implicito*

### 5.7.3 Gestione delle rotte

Per poter risparmiare informazioni all'interno dei *pacchetti*, generalmente, le rotte vengono aggregate pagando una perdita di precisione nel percorso.

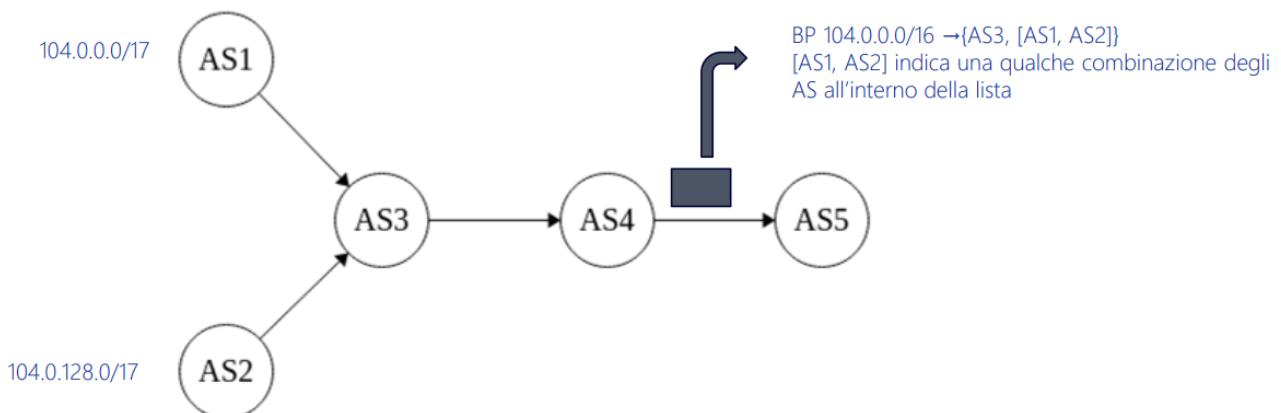


Fig. 5.21: Aggregazione delle rotte

**Filtri** I filtri controllano ciò che entra e ciò che esce da un *AS* e a seconda dei casi si parla di *ingress filters* e *egress filters*. È possibile definire filtri specifici per ogni connessione e possono essere generici (e.g. non lasciar passare nulla che contenga *AS19*) o molto precisi (e.g. si può arrivare ad analizzare i singoli *frame*). Ovviamente, tutti i *pacchetti* che non superano i filtri vengono scartati.

**Routing Information Base - RIB** Il *BGP* utilizza tre tabelle per gestire le rotte:

- **ADJ\_RIB\_IN**: contiene le rotte che sono state accettate in ingresso e che verranno valutate per ricercare percorsi alternativi;
- **Routing table**: contiene i *best path* attuali;
- **ADJ\_RIB\_OUT**: contiene le rotte che hanno superato i filtri in uscita e che devono essere condivise

Quando viene ricevuto un messaggio di **Update**, se il *pacchetto* supera i filtri in ingresso, viene modificata la tabella **ADJ\_RIB\_IN** e in particolare, se il messaggio trasporta una nuova rota, questa viene aggiunta alla tabella, se invece si tratta di un **Update** con *withdraw*, la rota viene rimossa. A quel punto, viene rivalutato il *best path* per la rota aggiornata e, se è migliore di quello attuale, oppure consente di raggiungere una nuova destinazione, vengono aggiornate anche la **Routing table** e la tabella **ADJ\_RIB\_OUT**. I nuovi cambiamenti vengono infine trasmessi a tutti gli altri vicini.

# **Capitolo Nr.6**

---

## ***Livello Data Link***

---

Per terminare la trattazione, vediamo il *livello Data Link* e in particolare, cerchiamo di capire quali sono i principi alla base dei servizi che offre e come sono implementate alcune tra le più comuni tecnologie di *livello Data Link*.

### **6.1 Introduzione**

Nella terminologia usata finora, *Router* e *host* costituiscono i nodi della rete e i canali di comunicazione tra due nodi sono detti *link* o *collegamenti*. Abbiamo anche già detto che le *PDU* di *livello Data Link* si chiamano *frame* e incapsulano i *pacchetti* del *livello Rete*. Diversamente da quanto visto finora, un *frame*, oltre a *header* e *payload*, include anche un *trailer* aggiunto dopo il *payload*. Mittente e destinatario sono identificati mediante *indirizzi MAC*.

Lo scopo dei servizi di questo livello è trasportare i *frame* da un nodo a un altro fisicamente adiacente. Ovviamente, un percorso può attraversare più *collegamenti* che potrebbero anche utilizzare protocolli diversi.

#### **6.1.1 Servizi del livello Data Link**

Vediamo allora nel dettaglio quali sono i servizi offerti dal livello 2:

- *Consegna affidabile tra nodi adiacenti*: questo servizio è sempre implementabile, ma è poco usato su *link* con bassi tassi di errore;
- *Controllo di flusso*: la velocità di trasmissione viene adattata alle possibilità di mittente e destinatario;
- *Rilevamento degli errori*: il ricevitore può identificare la presenza di errori ed agire di conseguenza;
- *Correzione degli errori*: il ricevitore può identificare e correggere gli errori senza richiedere ritrasmissioni;

Prima di procedere oltre, vediamo una caratterizzazione fondamentale dei *collegamenti*.

---

#### **Definizione 18 - Collegamento half-duplex e full-duplex.**

---

*Un collegamento nel quale i dati possono viaggiare in entrambe le direzioni contemporaneamente è detto essere full-duplex, altrimenti è half-duplex.*

### 6.1.2 Implementazione dei servizi Data Link

I servizi di livello 2 sono implementati su tutti gli *host* e generalmente sono gestiti dal firmware di un “adattatore di rete” o di un chip. Questi componenti sono poi direttamente collegati al bus di sistema dell’*host*.

**NB.** L’implementazione del *livello Data Link* richiede anche l’implementazione del *livello Fisico*.

Il processo di trasmissione di un *frame* è simile a quanto accade nei livelli superiori: Il mittente incapsula i dati in un *frame* e, se serve, aggiunge ulteriori bit per il controllo degli errori e per il controllo di flusso; il destinatario collabora al controllo di flusso<sup>1</sup>, controlla la presenza di errori e quindi estrae i dati per passarli ai livelli superiori.

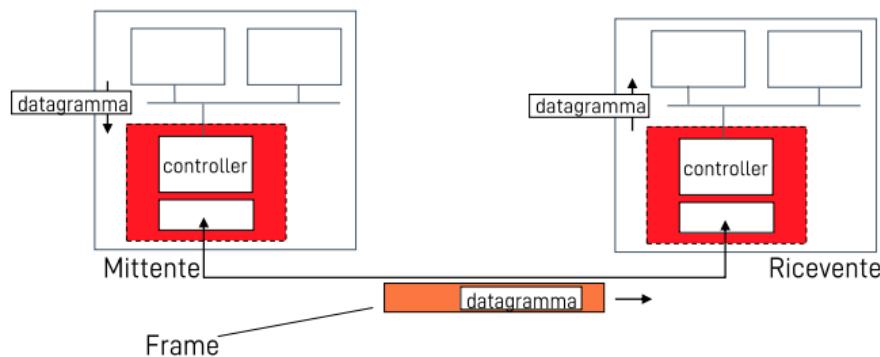


Fig. 6.1: Trasmissione di un *frame* di livello 2

## 6.2 Rilevamento e correzione di errori

I meccanismi di rilevamento e correzione degli errori richiedono l’utilizzo di bit ridondanti. Ad esempio, se EDC sono i bit ridondanti e D sono i dati da proteggere, il processo di correzione e rilevamento di errori è a grandi linee quello mostrato nella figura sottostante.

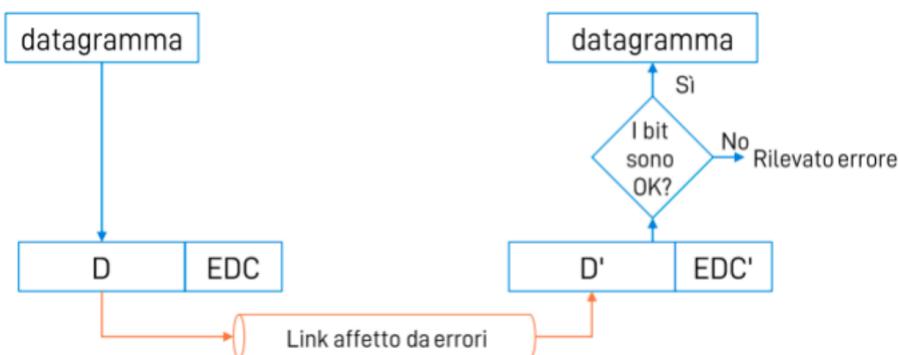


Fig. 6.2: Meccanismo di rilevamento e controllo degli errori

Occorre però notare che il rilevamento degli errori non è mai affidabile al 100%, ma in generale, maggiore è la ridondanza, maggiore è la protezione.

Vediamo quindi alcuni algoritmi per il controllo e la correzione degli errori.

---

<sup>1</sup>Vedremo più avanti come

### 6.2.1 Bit di parità

Questa tecnica si può realizzare in una o due dimensioni. Nel primo caso viene aggiunto un singolo bit che vale 1 se il numero di bit a 1 nella stringa originale è dispari. La parità su due dimensioni prevede invece che a partire dalla stringa di bit venga realizzata una matrice e che venga aggiunto un bit di parità su ogni riga e ogni colonna.

Il controllo a una dimensione permette soltanto il rilevamento di errori, mentre il controllo a due dimensioni permette anche la correzione.

<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="background-color: #0070C0; color: white;">data bits</th> <th style="background-color: #0070C0; color: white;">parity bit</th> </tr> </thead> <tbody> <tr> <td>0111000110101011</td> <td>1</td> </tr> </tbody> </table>	data bits	parity bit	0111000110101011	1		
data bits	parity bit					
0111000110101011	1					
(a) Controllo a una dimensione	(b) Controllo a due dimensioni	(c) Controllo a due dimensioni con correzione				

Fig. 6.3: Controllo di parità

Il controllo di parità a due dimensioni permette di correggere gli errori solo se in ogni riga e in ogni colonna si trova al più un bit errato. Comunque, in entrambe le modalità, il rilevamento non funziona se il numero di bit sbagliati è pari.

### 6.2.2 Ridondanza e interleaving

Questa tecnica ridonda ogni bit della stringa e con un'operazione di interleaving li riordina. Il risultato è una stringa di bit che resiste efficacemente a raffiche di errori, ovvero a errori su bit in successione.

Quando il destinatario riceve i dati, ripristina l'ordine dei bit con un'operazione di de-interleaving e ricostruisce il messaggio originale applicando una scelta a maggioranza su gruppi di bit.

#### Esempio 13 - Esempio di applicazione.

*Si supponga di voler trasmettere la stringa HELLO. Ogni carattere viene ridondato ottenendo, ad esempio:*

*HHH EEE LLL LLL 000*

*A questo punto con l'interleaving vengono riordinate le lettere:*

*HEL LOH ELL OHE LLO*

*Supponiamo quindi che una sequenza di 5 bit arrivi modificata al destinatario:*

*HEL LOH EXX XXX LLO*

*Dopo il riordino, il destinatario si ritrova la seguente stringa:*

*HHX EEX LXL LXL OXO*

*Scegliendo a maggioranza su ogni gruppo, si riesce comunque a ricostruire la stringa originale HELLO.*

### 6.2.3 Cyclic Redundancy Check

Il **CRC** è attualmente l'algoritmo più efficiente per il controllo degli errori. I dati vengono considerati come un numero binario  $D$ . L'algoritmo quindi, sceglie una sequenza di  $r + 1$  bit detta “generatore” e un altro numero  $G$  noto sia al mittente che al destinatario.

Quindi, il valore  $R$  del **CRC** viene composto scegliendo  $r$  bit in modo tale che  $D \text{ mod } G = R$  o, equivalentemente, che la concatenazione  $\langle D, R \rangle \text{ mod } G = 0$ . A questo punto, se quando il destinatario calcola  $\langle D, R \rangle \text{ mod } G$  ottiene un resto diverso da 0, c'è sicuramente un errore.

Questo sistema permette di rilevare anche errori a raffica, se questi hanno modificato meno di  $r$  bit in sequenza.

**NB.** Per concatenare  $D$  ed  $R$  è sufficiente applicare la seguente funzione:

$$\langle D, R \rangle = (D \cdot 2^r) \text{ XOR } R$$

**Calcolare il CRC** Se  $(D \cdot 2^r) \text{ XOR } R$  deve essere un multiplo di  $G$ , deve valere quanto segue:

$$(D \cdot 2^r) \text{ XOR } R = n \cdot G \quad \text{con } n \in \mathbb{Z}$$

Applicando lo XOR a entrambi i membri, il termine a sinistra si semplifica:

$$D \cdot 2^r = n \cdot G \text{ XOR } R$$

e segue che:

$$CRC = R = \text{resto}((D \cdot 2^r)/G) = (D \cdot 2^r) \text{ mod } G$$

#### Esempio 14 - Esempio di calcolo del CRC.

Supponiamo di aver ricevuto un messaggio  $D = 101110$  e di doverne verificare la correttezza. Siano  $G = 1001$  e  $R = 011$ .

Iniziamo calcolando  $D \cdot 2^r$ . Poiché  $R = 011$  è una stringa di 3 bit,  $r = 3$  e quindi:

$$D \cdot 2^r = 101110000^2$$

Quindi, calcoliamo il CRC e vediamo se coincide col valore atteso.

		1	0	1	1	1	0	0	0	0
1	-	1	0	0	1					
			1	0	1					
0	-	0	0	0						
			1	0	1	0				
1	-	1	0	0	1					
				1	1	0				
0	-	0	0	0						
				1	1	0	0			
1	-	1	0	0	1					
					1	0	1	0		
1	-	1	0	0	1					
					0	1	1			

Il resto è uguale ad  $R$ , quindi non ci sono stati errori e i dati sono corretti.

<sup>2</sup>Abbiamo semplicemente shiftato  $D$  di 3 posizioni verso sinistra

## 6.3 Gestione degli accessi multipli ai canali

Distinguiamo due tipi di collegamenti:

- *Punto-punto*: sono collegamenti tra due soli *host* e utilizzano protocolli per la comunicazione punto-punto (e.g. *PPP*);
- *Broadcast*: sono collegamenti condivisi accessibili da più *host*;

I *collegamenti broadcast* comportano una serie di problemi legati alla necessità di regolare le trasmissioni ed evitare le interferenze. Infatti, se due o più *host* trasmettessero nello stesso momento, i segnali delle rispettive trasmissioni si mescolerebbero tra loro. Se invece un *host* ricevesse due o più trasmissioni nello stesso momento, si verificherebbe una *collisione*.

Per evitare ciò, vengono usati protocolli di accesso multiplo, ovvero algoritmi distribuiti che determinano il modo e il momento in cui ogni *host* può trasmettere su un canale. Le informazioni necessarie per il funzionamento dei protocolli possono viaggiare sullo stesso canale dei dati o su un canale dedicato e in quel caso si parla di “*out-of-band*” *channel*.

Idealmente, vorremmo che in un *collegamento* a  $R_s^{bit}$ , il protocollo *MAC* utilizzato permetta ad un *host* di usare l’intera *banda*, quando è l’unico a trasmettere, o che questa venga divisa equamente tra tutti gli *host* che vogliono trasmettere. Inoltre, il protocollo dovrebbe essere semplice e funzionare in modo decentralizzato.

A questo punto possiamo suddividere i protocolli *MAC* in tre categorie:

1. *Protocolli a ripartizione delle risorse*: le risorse di un canale vengono suddivise equamente tra tutti gli *host* ad esso connesse;
2. *Protocolli ad accesso casuale*: il canale non viene suddiviso, ma si cerca di limitare il numero di *collisioni*;
3. *Protocolli a turni intelligenti*: gli *host* accedono al canale a turno e la durata dei turni varia in base alla quantità di dati da trasmettere;

### 6.3.1 Protocolli a ripartizione di risorse

All’interno di questa classe consideriamo tre protocolli: *TDMA*, *FDMA* e *CDMA*. All’inizio della trattazione abbiamo già descritto i principi di *TDMA* e *FDMA*. Nel primo vengono definiti degli slot temporali di dimensione fissa che vengono assegnati agli *host*. Gli *host* trasmettono con un sistema a turni e utilizzano tutta la banda disponibile. Nel caso dell’*FDMA* invece, lo spettro del canale viene suddiviso in sotto-bande che vengono assegnate ai trasmettitori. Ogni *host* può quindi trasmettere in qualsiasi momento e senza limiti di tempo. In entrambi i casi però, quando un *host* non ha bisogno di trasmettere, le risorse ad esso assegnate rimangono inutilizzate.

Diverso è invece il funzionamento del *CDMA*, nel quale, ogni *host* può usare tutta la banda e tutto il tempo e la ripartizione delle risorse avviene assegnando un “codice” diverso ad ogni *host*. Con codice si intende una sequenza di bit detti “chip” che commuta più rapidamente di quanto commutino i bit di dati. Quando un *host* trasmette, trasmette lo *XOR* tra la sequenza dei dati e il codice. Il ricevente ricalcolando lo *XOR* con lo stesso codice, riesce a recuperare i dati originali. Se invece la stringa ricevuta venisse “xorata” con un codice diverso, il risultato non avrebbe alcun significato.

Il problema di questa tecnica è che poiché il codice deve commutare più velocemente dei dati, ovvero  $T_{chip} > T_{bit}$ , è necessario diminuire la velocità di trasmissione dei dati, di fatto, rallentando la comunicazione.

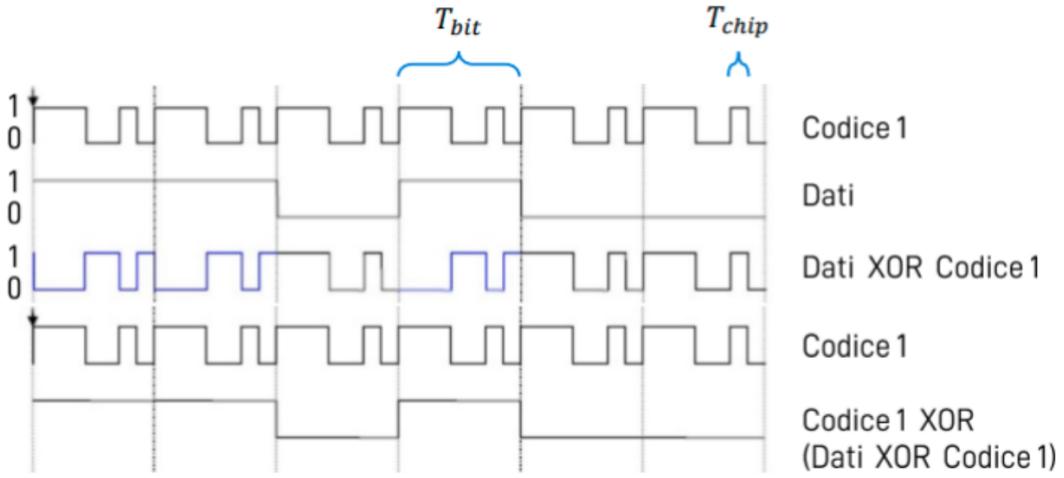


Fig. 6.4: Funzionamento del *CDMA*

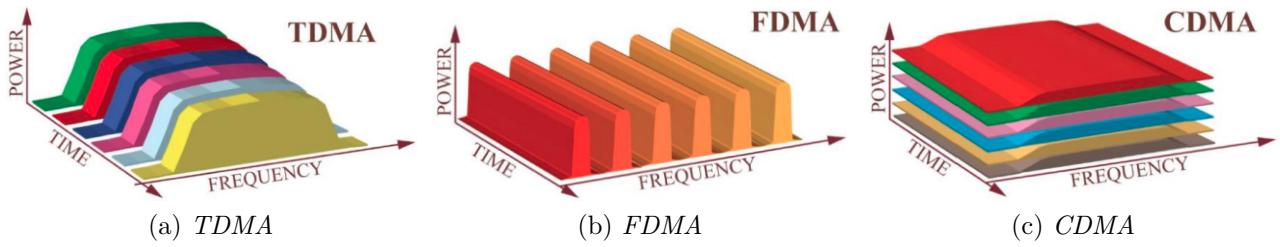


Fig. 6.5: Confronto tra protocolli a ripartizione di risorse

### 6.3.2 Protocolli ad accesso casuale

I protocolli ad accesso casuale prevedono che quando un *host* trasmette, lo faccia con la banda massima possibile. Non prevedono alcun coordinamento tra gli *host* prima della trasmissione e, di conseguenza, si possono verificare delle *collisioni*. Sono poi i protocolli stessi a stabilire come e se rilevare e recuperare le *collisioni*.

**Slotted ALOHA** Nello *Slotted ALOHA* tutti i *frame* sono della stessa lunghezza e il tempo viene suddiviso in slot di durata pari alla dimensione dei *frame*. Tutti gli *host* sono sincronizzati e possono trasmettere solo all'inizio di uno slot. Nel caso di *collisioni*, queste vengono rilevate da tutti gli *host* che stanno trasmettendo.

A questo punto il funzionamento è molto semplice: quando un *host* deve trasmettere, aspetta l'inizio dello slot successivo. Se non ci sono *collisioni*, continua a trasmettere anche nello slot seguente altrimenti, con probabilità  $p$ , ritrasmette nello slot successivo finché non ha successo.

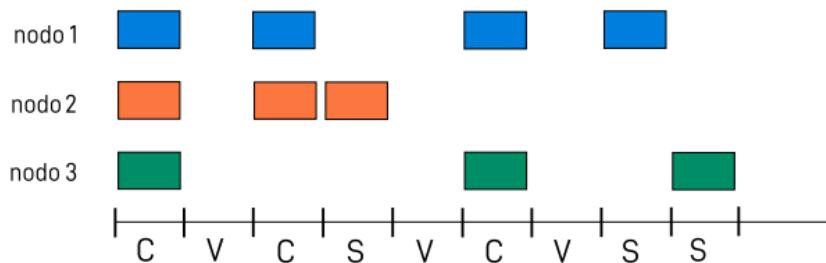


Fig. 6.6: Funzionamento dello *Slotted ALOHA*

I vantaggi di questo protocollo sono la semplicità di implementazione, la decentralizzazione e

la possibilità per un *host* di trasmettere in continuazione nel caso sia l'unico a doverlo fare. Tuttavia, la sincronizzazione richiede un coordinamento, alcuni slot potrebbero rimanere inutilizzati e le *collisioni* sono probabili e frequenti. Inoltre, questa modalità di rilevamento delle *collisioni* costringe ad aspettare la fine della trasmissione introducendo un'inefficienza di fondo.

Proviamo allora a calcolare l'efficienza di questo protocollo.

Tutti i *frame* hanno la stessa dimensione. Chiamiamo  $G$  il traffico offerto, cioè il numero medio di *frame* inviati sul canale da tutti gli *host*, includendo sia trasmissioni che ritrasmissioni. Ovviamente  $G \geq 0$ . La probabilità che in uno slot ci siano  $k$  *frame* da trasmettere è descritta dalla seguente variabile aleatoria di Poisson:

$$P[k] = \frac{G^k \cdot e^{-G}}{k!}$$

Il *throughput* ideale è 1 poiché nella situazione migliore in ogni slot c'è soltanto un *frame* da inviare rendendo impossibile il verificarsi di *collisioni*. Il *throughput* effettivo corrisponde invece al valore di  $P[k = 1]$ :

$$P[k = 1] = G \cdot e^{-G}$$

Il valore massimo di questa funzione si ottiene quando  $G^* = 1$ . Sostituendo quel valore nella funzione di probabilità si ottiene un *throughput* massimo pari a  $\frac{1}{e} \approx 0.368$ .

**ALOHA** L'*ALOHA*, o *ALOHA puro*, funziona come lo *Slotted ALOHA*, ma gli *host* non sono sincronizzati e quindi i *frame* vengono trasmessi immediatamente e non all'inizio di uno slot.

Il prezzo di questa maggiore semplicità lo si paga in efficienza in quanto la probabilità di *collizione* aumenta: un *frame* inviato al tempo  $t_0$  collide con i *frame* inviati nell'intervallo  $[t_0 - 1, t_0 + 1]$ .

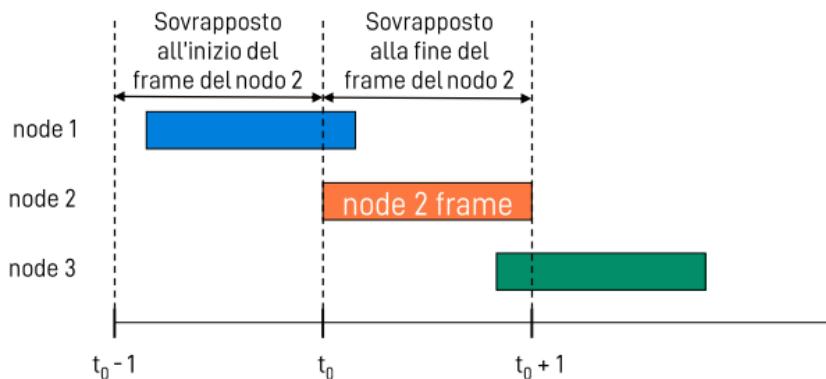


Fig. 6.7: Funzionamento dell'*ALOHA*

Proviamo a calcolare l'efficienza di questa versione. Le supposizioni solo le stessa fatte in precedenza, tranne per il *throughput* che stavolta corrisponde alla probabilità che un *frame* venga trasmesso all'interno dell'intervallo di vulnerabilità  $[t_0 - 1, t_0 + 1]$ .

Quindi, la probabilità che al tempo  $t$  venga trasmesso un solo *frame* è:

$$P[k = 1] = G \cdot e^{-G}$$

La probabilità che non ci siano *frame* la cui trasmissione è iniziata nell'intervallo  $[t_0, t]$  vale invece:

$$P[k = 0] = e^{-G}$$

La probabilità che una trasmissione abbia successo vale quindi:

$$P[k = 1] \cdot P[k = 0] = (G \cdot e^{-G}) \cdot e^{-G} = G \cdot e^{-2G}$$

Questa funzione ha valore massimo quando  $G = \frac{1}{2}$  e quindi il *throughput* massimo è pari a  $\frac{1}{2e} \approx 0.184$  che è la metà del *throughput* massimo dello *Slotted ALOHA*.

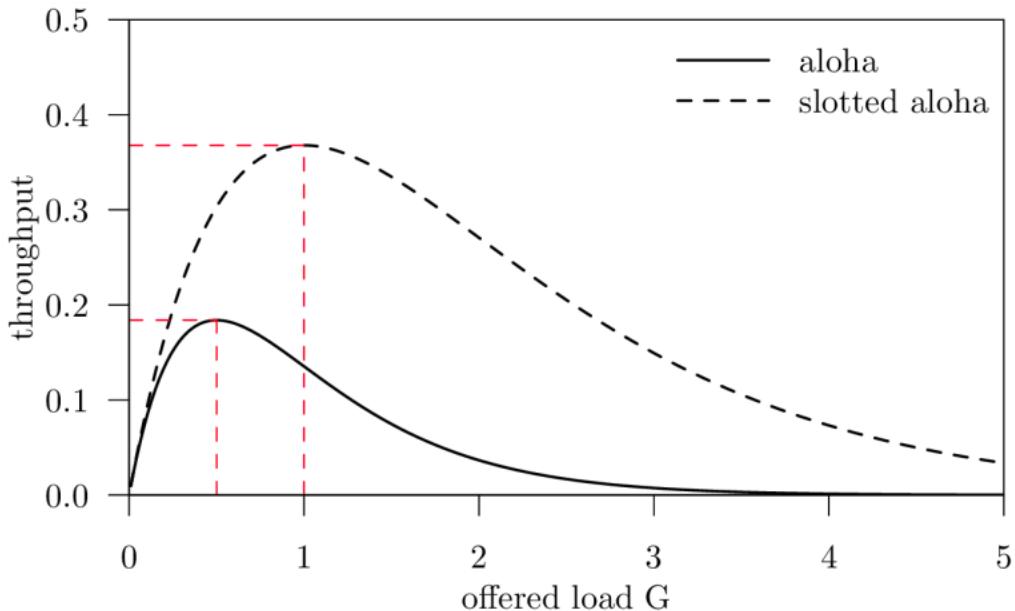


Fig. 6.8: Confronto tra *Slotted ALOHA* e *ALOHA*

**CSMA** Il **CSMA** è un protocollo basato sull’ascolto. In particolare, viene “ascoltato” il canale e, se è libero, viene trasmesso un *frame*, altrimenti viene ritardata la trasmissione. Esistono diverse versioni del protocollo che si distinguono in base al comportamento in caso di canale occupato.

- *CSMA 0-persistente*: se il canale è occupato l’*host* attende un tempo casuale maggiore del tempo di trasmissione e poi ritenta;
- *CSMA 1-persistente*: se il canale è occupato l’*host* attende fino a quando non si libera e poi trasmette;
- *CSMA p-persistente*: se il canale è occupato l’*host* attende fino a quando non si libera e poi con probabilità  $p$  trasmette il *frame* e con probabilità  $1 - p$  attende un tempo casuale maggiore del tempo di trasmissione prima di ritentare;

In tutti e tre i casi, se si verifica una *collisione*, l’*host* rimane in attesa per un tempo casuale e poi ritenta secondo quelle che sono le procedure utilizzate.

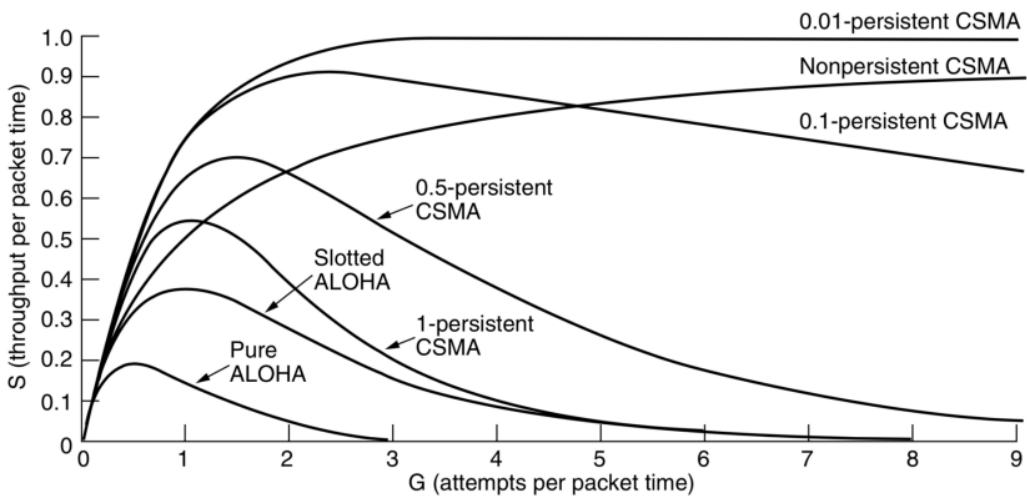


Fig. 6.9: Confronto tra le versioni del *CSMA*

Come fa il *CSMA* a gestire le collisioni?

Consideriamo il periodo di vulnerabilità. Esso dipende dal *tempo di propagazione*  $\tau$  e dal tempo richiesto  $T_a$  per rilevare se il canale è libero o meno. Se un *host* trasmette, ma il segnale non ha raggiunto tutti gli altri *host*, qualcun altro potrebbe iniziare a trasmettere. La finestra temporale all'interno della quale ciò potrebbe accadere corrisponde al periodo di vulnerabilità  $T_v = \tau + T_a$ . Per questo motivo, il *CSMA* solitamente si usa quando il *tempo di propagazione* è molto minore del *tempo di trasmissione*.

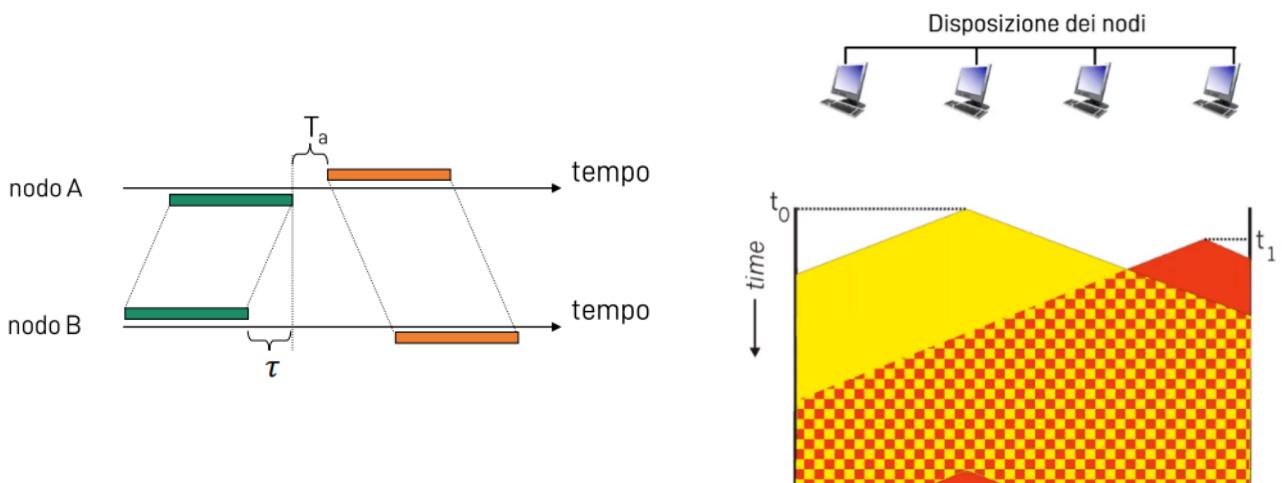


Fig. 6.10: *Periodo di vulnerabilità e collisioni nel CSMA*

Quindi, nel *CSMA* il *ritardo di propagazione* porta al verificarsi di *collisioni*. Quando ciò accade, l'intero *tempo di trasmissione* viene sprecato.

Per questo motivo esistono due ulteriori versioni del *CSMA*: *CSMA/CD* e *CSMA/CA* che permettono rispettivamente di rilevare ed evitare le *collisioni*. Il primo è in grado di rilevare le *collisioni* e quindi di interrompere la trasmissione riducendo lo spreco di risorse. Nel secondo invece, il funzionamento è uguale a quello che si ha nel *CSMA p-persistent*, ma il parametro  $p$  viene fatto variare in base alle condizioni della rete.

Il motivo per il quale esistono sia il *CSMA/CD* che il *CSMA/CA* è che il rilevamento delle *collisioni* viene fatto confrontando la potenza del segnale trasmesso con quella del segnale ricevuto. Questo confronto è molto difficile nel caso delle reti wireless in quanto la potenza del segnale trasmesso è molto maggiore della potenza del segnale ricevuto. Quindi, il *CSMA/CD* è

usato nelle reti cablate, mentre il *CSMA/CA* nelle reti wireless nelle quali, inoltre, il *tempo di trasmissione* è molto minore del *tempo di propagazione*  $\tau$  e del tempo  $T_a$  necessario per sondare il canale, ovvero:  $T \ll \tau + T_a$

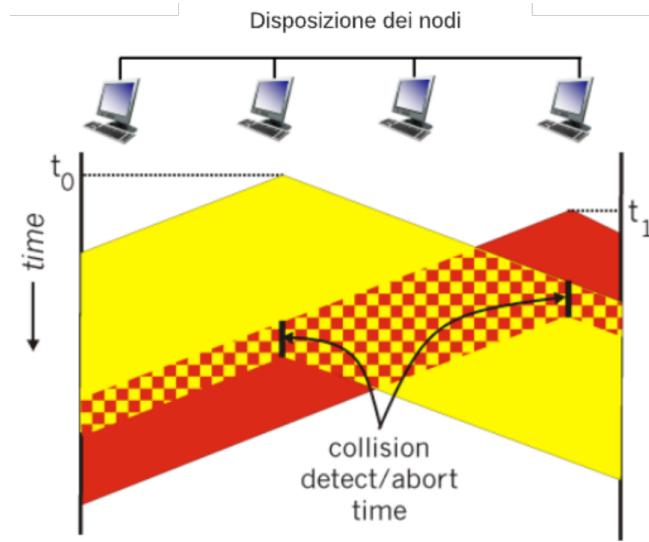


Fig. 6.11: Comportamento del *CSMA/CD*

### 6.3.3 Protocolli a turni intelligenti

**Polling** È un protocollo basato sul paradigma *master-slave* nel quale un *host*, il *master*, invita gli altri *host*, gli *slave*, a trasmettere a turno. Di solito è usato in reti nella quali gli *slave* hanno una bassa potenza di calcolo. Nonostante l'evidente semplicità, si hanno problemi legati all'alta latenza, all'overhead provocato dai messaggi di polling e alla presenza di un *SPOF*.

**Token passing** Esiste un *token*, un particolare *frame*, che conferisce all'*host* che ne è in possesso il diritto a trasmettere. Il *token* viene quindi fatto girare tra tutti gli *host* permettendo a tutti, prima o poi, di trasmettere. I problemi di questo protocollo sono gli stessi del precedente e in particolare, il *SPOF* è costituito dal *token*.

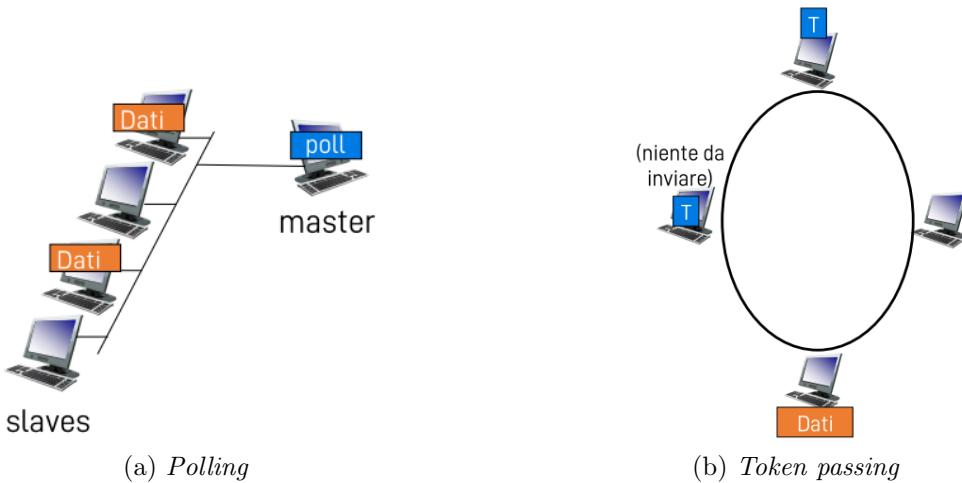


Fig. 6.12: *Polling* e *Token passing*

## 6.4 Protocollo Ethernet

I protocolli del *livello Data Link* e del *livello Fisico* sono standardizzati e appartengono al gruppo *IEEE 802*.

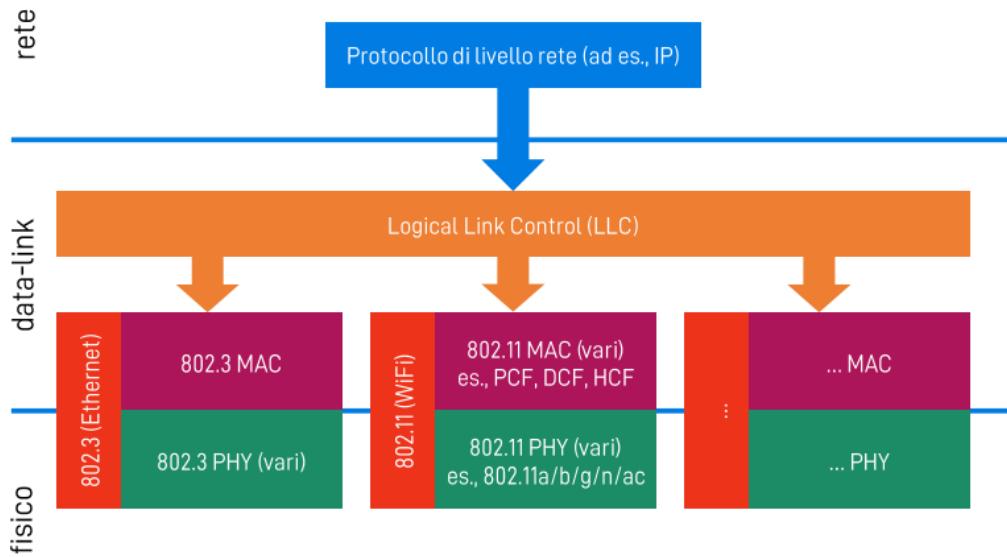


Fig. 6.13: Protocolli del gruppo *IEEE 802*

**NB.** Ogni protocollo del *livello Data Link* è associato ai rispettivi protocolli di *livello Fisico*. Nelle reti cablate il protocollo che di fatto è lo standard universale è il protocollo *Ethernet*. *Ethernet* nasce come protocollo proprietario col nome *DIX Ethernet* (Digital-Intel-Xerox) per poi essere standardizzato e liberalizzato.

### 6.4.1 Organizzazione delle reti Ethernet

Inizialmente le reti *Ethernet* erano organizzate secondo la topologia a bus e quindi vi era un unico mezzo di trasmissione condiviso al quale, mediante un *transceiver*, si collegavano tutti gli *host* di quella rete. I dati trasmessi da un *host* venivano ricevuti da tutti.

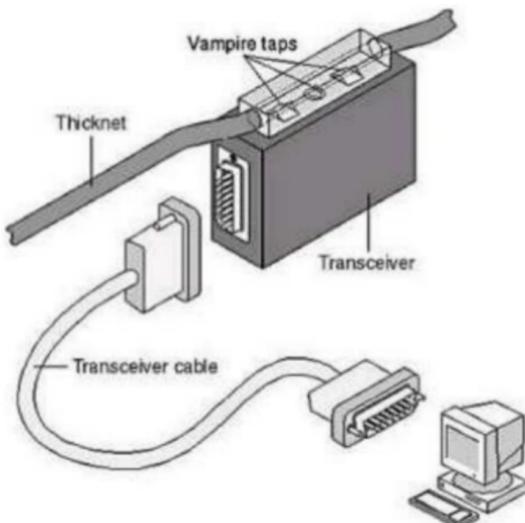


Fig. 6.14: *Transceiver Ethernet*

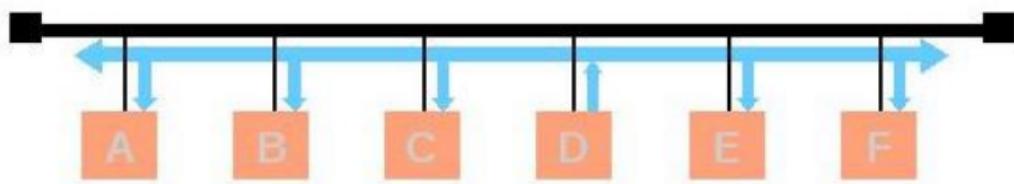


Fig. 6.15: Rete *Ethernet* con topologia a bus

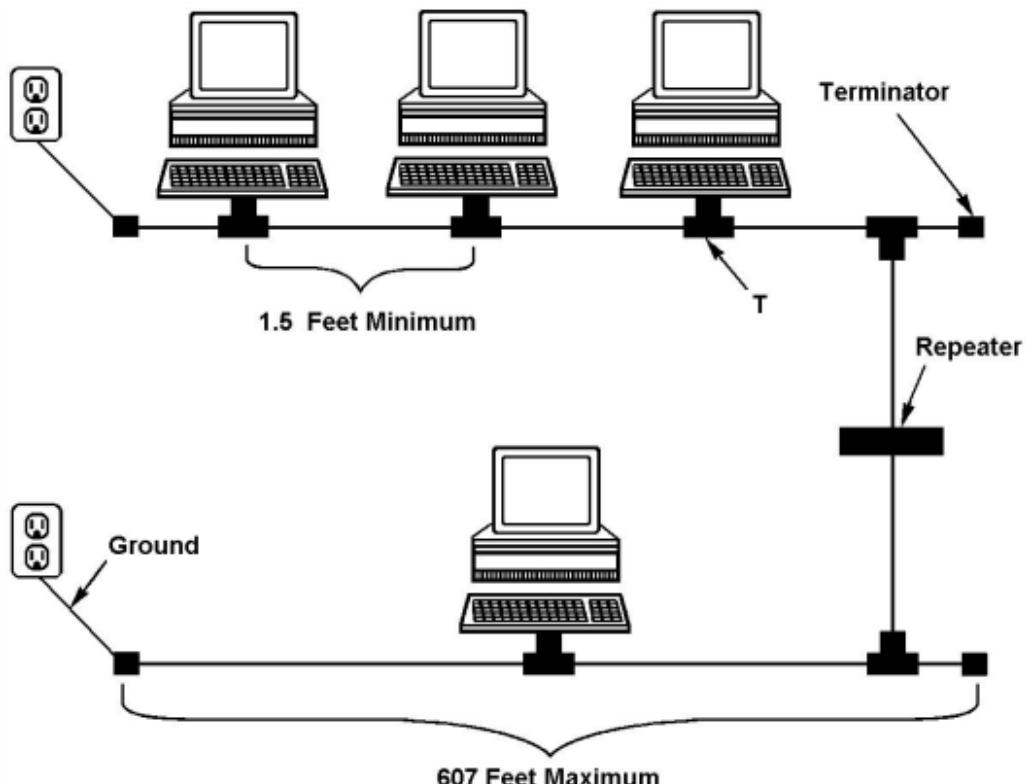


Fig. 6.16: Organizzazione di una rete *Ethernet* a bus

Il terminatore (Terminator nella figura sopra) è necessario per evitare che il segnale, raggiunta la fine del cavo, ritorni indietro.

Con la nascita degli hub prima e degli *Switch* dopo la topologia a bus è stata sostituita da quella a stella. In questa topologia, tutti gli *host* sono collegati ad un dispositivo centrale che riceve tutti i *frame* e li inoltra ad uno o più degli altri *host*.

#### 6.4.2 Struttura dei frame Ethernet

Le schede di rete, nel momento dell'invio, incapsulano i *pacchetti* di livello 3 all'interno di *frame* di livello 2. Nel caso del protocollo *Ethernet* la struttura dei *frame* è quella descritta nella figura sottostante.



Fig. 6.17: Campi di un *frame* *Ethernet*

Esaminiamo i singoli campi:

- **Preambolo**: è composto da 7 byte 10101010 e un byte 10101011 ed è utilizzato per sincronizzare il clock del ricevitore e del trasmettitore;
- **Indirizzo destinazione**: un gruppo di 6 byte con i quali è rappresentato l'*indirizzo MAC* di destinazione;
- **Indirizzo sorgente**: un gruppo di 6 byte con i quali è rappresentato l'*indirizzo MAC* sorgente;
- **Lunghezza**: è un valore a 16 bit che indica la dimensione in byte dell'intero *frame*;
- **Tipo**: identifica il protocollo di livello 3 trasportato;
- **Dati**: è il *payload* del *frame*;
- **CRC**: codice per effettuare il controllo degli errori con l'algoritmo *CRC*;

**NB.** L'indirizzo di destinazione è il secondo campo del *frame* in modo che sia possibile prendere decisioni di inoltro concorrentemente alla ricezione dei restanti byte. Il codice *CRC* invece, è in fondo al *frame* perché per eseguire i controlli d'errore è necessario averlo prima ricevuto per intero.

Quando una scheda di rete riceve un *frame* diretto a sé oppure un *frame* con un indirizzo di *broadcast*, passa il *payload* del *frame* al livello superiore, altrimenti scarta tutto il *frame*.

#### 6.4.3 Caratteristiche del protocollo Ethernet

Il protocollo *Ethernet* non è né un protocollo connesso, per cui non avviene alcuno scambio di messaggi di controllo tra mittente e destinatario, e né un protocollo affidabile. Non vengono infatti usati messaggi *ACK* o simili per gestire le ritrasmissioni e l'unico controllo che viene fatto è mediante il *CRC*. Di conseguenza, eventuali ritrasmissioni vengono gestite dai protocolli di livello superiore.

*Ethernet* implementa un protocollo *MAC* di tipo *CSMA/CD* “*unslotted*”. Ciò significa che quando la scheda di rete ha un *frame* da trasmettere, verifica lo stato del canale: se è libero trasmette, altrimenti trasmette appena possibile. Se la trasmissione termina senza rilevare altre trasmissioni, il *frame* viene considerato consegnato. In caso contrario, la scheda trasmette un segnale di “*abort*”. Ogni volta che si verifica una *collisione*, la scheda sceglie a caso un valore di *backoff*, cioè il tempo che deve attendere prima di riprovare a trasmettere, tra i valori 0 e  $(2^k - 1)T$  dove  $k \leq 7$  e  $T$  è un valore noto (e.g.  $50\mu s$ ) che rappresenta il tempo necessario a trasmettere 512 bit.

**NB.** Il valore di *backoff* può essere soltanto 0 o  $(2^k - 1)T$  e non un valore intermedio.

**NB.** Sebbene esistano molti standard *Ethernet* differenti, la struttura dei *frame* e i protocolli *MAC* sono comuni.

### 6.5 Ethernet switching

In precedenza abbiamo parlato di dispositivi quali hub e *switch*. Un hub non è altro che un ripetitore: ogni volta che riceve un *frame* lo inoltra, senza fare altro, su tutte le porte ad eccezione di quella di arrivo. Uno *switch* d'altra parte, è un dispositivo più attivo: quando riceve un *frame*, esamina l'*indirizzo MAC* di destinazione e seleziona di conseguenza le porte

verso le quali inoltrarlo. Inoltre, gli *switch* sono trasparenti agli altri *host* e sono in grado di configurarsi automaticamente, cioè possono associare ad ogni porta gli *indirizzi MAC* dei dispositivi da essa raggiungibili.

### 6.5.1 Domini di collisione

---

#### Definizione 19 - Dominio di collisione.

*Un dominio di collisione rappresenta la porzione di rete all'interno della quale possono verificarsi delle collisioni.*

La scelta della topologia e la scelta dei dispositivi di rete influenzano il numero dei *domini di collisione*. Utilizzando una topologia a bus, tutti gli *host* connessi al canale appartengono allo stesso *dominio* e quindi ogni *host* può collidere con qualunque altro. Nella topologia a stella invece, va considerato il dispositivo che funge da centro: se si tratta di un hub, esiste un unico *dominio*, mentre con uno *switch* si ottengono tanti *domini* quante sono le porte dello *switch* utilizzate. Ciò significa che ogni *host* comunica con lo *switch* su un canale dedicato e quindi, se il *collegamento* è di tipo *full duplex*, non si possono verificare *collisioni*. Inoltre, questo permette anche a coppie di *host* di comunicare simultaneamente.

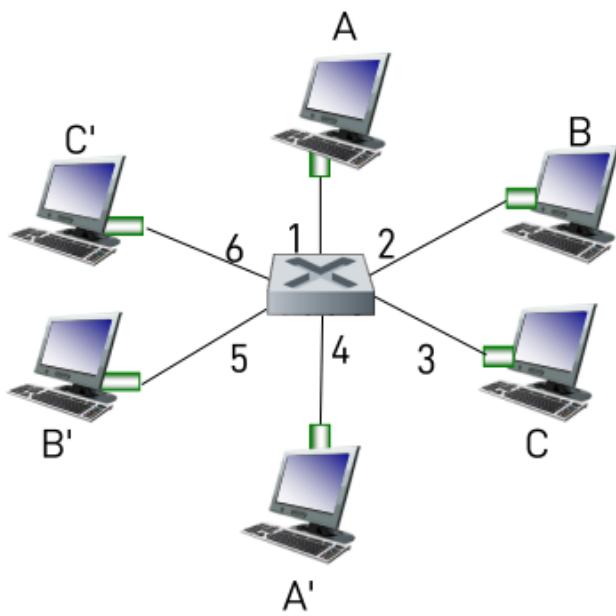


Fig. 6.18: Rete “switched”

Ad esempio, nella figura di cui sopra, le comunicazioni tra *A* e *A'* e tra *B* e *B'* possono avvenire in contemporanea senza che si verifichino *collisioni*.

### 6.5.2 Domini di broadcast

---

#### Definizione 20 - Dominio di broadcast.

*Un dominio di broadcast rappresenta la porzione di rete all'interno della quale può viaggiare un messaggio broadcast di livello 2.*

È importante ricordare che ad ogni *dominio di collisione* può corrispondere un solo *dominio di broadcast*. Il motivo è che se un *dominio di collisione* si estendesse su due *domini di broadcast*,

i messaggi trasmessi in broadcast dall'*ARP* non potrebbero raggiungere tutti gli *host* necessari e quindi si creerebbero due gruppi di *host* irraggiungibili.

**NB.** Ogni interfaccia di un *router* definisce un *dominio di broadcast*.

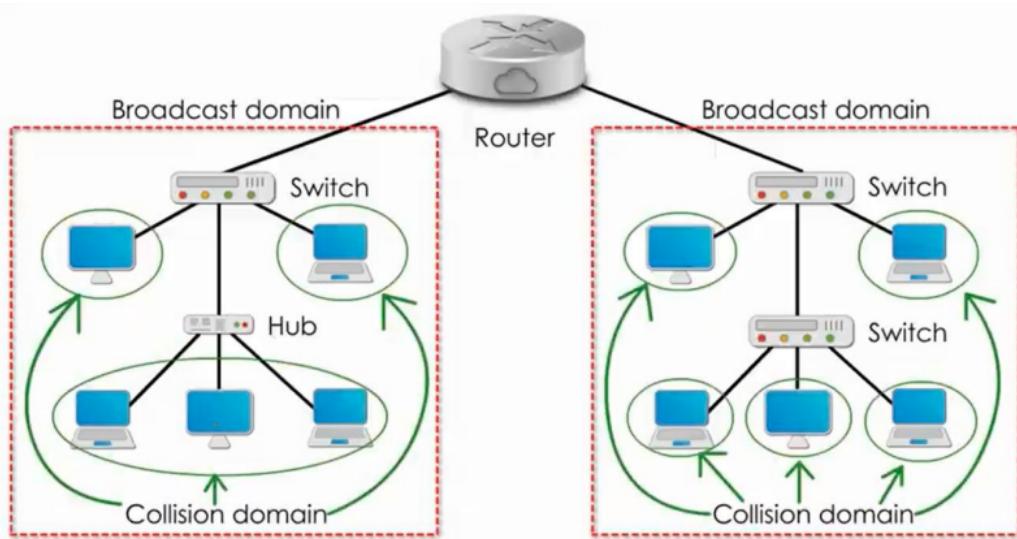


Fig. 6.19: Domini di collisione e di broadcast in una rete

### 6.5.3 Funzionamento degli switch

A questo punto ci chiediamo come facciano gli *switch* ad autoconfigurarsi e prendere le decisioni di inoltro. Come sarà facile immaginare, gli *switch* mantengono una tabella di questo tipo:

MAC address dell'host	Interfaccia per raggiungere l'host	Time to live
...	...	...

Fig. 6.20: Tabella di inoltro di uno *switch*

Il **Time to live** serve per mantenere aggiornate le entry scartando quelle troppo vecchie.

Configurare uno *switch* significa popolare la suddetta tabella e il processo che consente ai dispositivi di autoconfigurarsi è chiamato *backward learning*. In pratica, quando uno *switch* riceve un *frame*, aggiorna la tabella di inoltro, inserendo o aggiornando l'entry associata all'*indirizzo MAC* sorgente quindi, ricerca nella tabella l'*indirizzo MAC* di destinazione del *frame*: se l'entry esiste e l'interfaccia di uscita è diversa da quella di arrivo, inoltra il *frame*, altrimenti lo scarta. Se invece non viene trovata alcuna corrispondenza, il *frame* viene trasmesso in *flooding* su tutte le interfacce.

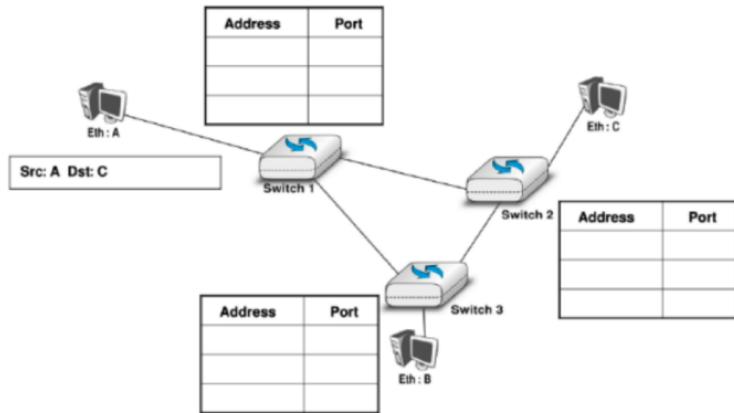
**NB.** Il motivo per il quale gli *switch* scartano i *frame* che dovrebbero essere inoltrati verso le stessa interfaccia di quella di arrivo è che, se è ciò accade, significa che a quella interfaccia è collegato un hub al quale sono collegati sia l'*host* sorgente che il destinatario. Di conseguenza, il *frame* ricevuto è già stato inviato dall'hub verso il legittimo destinatario e quindi se lo *switch* inoltrasse il *frame*, l'hub non farebbe altro che ritrasmetterlo di nuovo in *flooding* con la conseguenza che il destinatario riceverebbe due volte lo stesso *frame*.

### 6.5.4 Protocollo STP

Il processo di *backward learning* funziona anche con configurazioni più complesse degli apparati di rete. L'unica situazione nella quale si riscontrerebbero problemi è quella in cui esistono dei cicli tra gli *switch*.

#### Esempio 15 - Esempio di creazione di un ciclo.

Consideriamo per esempio la seguente configurazione:



Supponiamo che *A* debba inviare un frame a *C*. *Switch1* riceve il frame e non sapendo a chi inoltrarlo, lo trasmette in flooding. A questo punto, sia *Switch2* che *Switch3* ricevono il frame e di nuovo provano ad inviarlo in flooding. Sebbene con questo secondo passaggio il frame riesca ad arrivare alla sua destinazione, copie di esso continueranno a circolare potenzialmente all'infinito.

È evidente che questo problema può portare velocemente a saturare la capacità dei collegamenti. La soluzione a ciò è l'**STP**. L'**STP** prevede che tutti gli *switch* siano identificati per mezzo di un valore a 64 bit formato in modo tale che i primi 16 bit sono configurati dall'amministratore di reti, mentre i restanti 48 bit sono l'*indirizzo MAC* dello *switch*. Il protocollo a quel punto costruisce un albero con radice nello *switch* con ID minore. Questa modalità di scelta della radice permette all'amministratore di sceglierla nel modo più consono all'organizzazione della propria rete.

**Processo di costruzione dell'albero di copertura** Per creare l'albero di copertura gli *switch* si scambiano messaggi di controllo chiamati **BPDU** che contengono l'ID del mittente e il costo del collegamento. Quando uno *switch* *A* riceve una **BPDU** da un altro *switch* *B*, se l'ID di *A* è minore dell'ID di *B*, la porta di *B* verso *A* diventa la radice dell'albero per *B*, ovvero *A* diventa il padre di *B* nell'albero. Se invece l'ID di *A* è maggiore, la porta sulla quale *A* ha ricevuto il messaggio diventa la radice dell'albero per *A* e quindi *B* diventa padre di *A* nell'albero.

Se *A* diventa radice di *B*, *A* aggiorna il proprio costo a  $c = cost_A + cost_B$  e tutte le porte dalle quali *A* riceve **BPDU** con costo maggiore di  $c$  vengono identificate come “*designated ports*”, cioè figli di *A* nell'albero. I messaggi provenienti invece da porte il cui costo è uguale a  $c$  diventano “*blocked ports*” perché quelle porte sono potenziali padri di *B* e creerebbero cicli ai livelli più alti dell'albero se venissero usate.

Al termine del processo, ogni porta di collegamento tra *switch* sarà stata associata ad uno dei tre possibili tipi:

Tipo	Può ricevere BPDU	Può inviare BPDU	Può gestire i frame
Blocked	Sì	No	No
Radice	Sì	No	Sì
Designated	Sì	Sì	Sì

**Variazioni dell’albero di copertura** Le *BDPU* vengono inviate periodicamente per rilevare cambiamenti, ma ogni *switch* le trasmette soltanto verso le “*designated ports*”. In particolare, lo *switch* radice trasmette una *BPDU* verso tutte le proprie “*designated ports*” e gli altri *switch* a loro volta fanno lo stesso. Ogni *switch* poi, tiene traccia del momento in cui ha ricevuto l’ultima *BPDU*. Passato un certo periodo il processo di formazione dell’albero di copertura riparte da zero e fino a quando non si conclude gli *switch* non inoltrano il normale traffico dati.

### 6.5.5 Organizzazione logica delle LAN

Suddividere gli *host* di una *LAN* in gruppi è spesso una buona idea. Sia per ridurre le dimensioni dei *domini di broadcast* e aumentare così l’efficienza della rete, che per ridurre il carico di alcune porzioni di rete più sensibili. Un’altra buona ragione è la separazione di intenti, per esempio, si potrebbe voler separare i dispositivi accessibili dalle reti esterne dagli altri.

Per fare ciò in modo conveniente si ricorre alle *VLAN*. Le *VLAN* non sono altro che un insieme di porte di uno o più *switch*. Gli *switch* eseguono un processo di *backward learning* indipendente per ogni *VLAN* e i *frame* vengono inoltrati soltanto verso porte associate alla stessa *VLAN* della porta sorgente.

**NB.** Esistono *switch* che non supportano le *VLAN* e ne esistono altri che invece ne supportano di multiple.

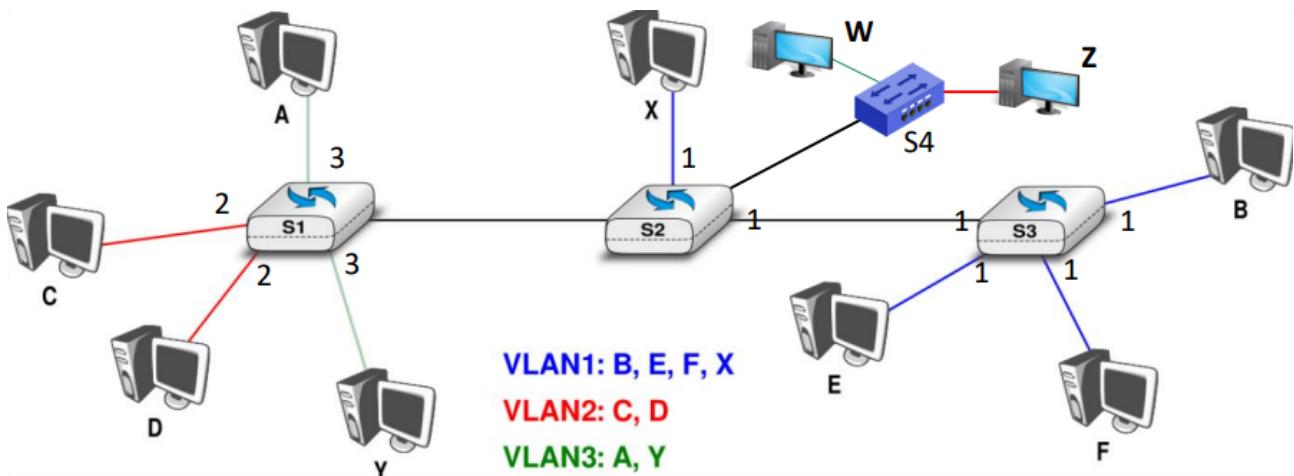


Fig. 6.21: Esempio di rete con tre *VLAN*

Dalla figura si vede come ogni interfaccia dei tre *switch* sia associata ad una *VLAN* con l’unica eccezione delle porte alle quali si collega il canale tra *S1* e *S2*. In quel caso le porte sono configurate in modalità *trunk* e ciò significa che permettono il passaggio del traffico di più di una *VLAN*.

C’è però un problema. Per poter capire a quale *VLAN* appartiene ogni *frame* sarebbe necessario inserire un identificativo nell’header, ma ciò non è possibile perché lo standard 802.3 di *Ethernet* non consente di inserire quest’informazione in nessuno dei campi già esistenti. Di conseguenza, fu introdotta una modifica allo standard con la realizzazione della versione 802.1Q di *Ethernet*.

Con il nuovo standard furono introdotti quattro nuovi campi nell'header:

- **VLAN protocol ID**: contiene un valore fisso a 0x8100;
- **Pri**: contiene il valore di priorità del *frame* ed è usato per realizzare politiche di quality of service;
- **CFI**: inizialmente specificava se l'ordine dei byte era big endian o little endian, ma successivamente dalla versione 802.5 fu utilizzato per reti a *token passing*;
- **VLAN identifier**: valore a 12 bit che identifica la *VLAN* di appartenenza;

**NB.** Nel campo **VLAN identifier** il valore 0 indica l'assenza di *VLAN*, mentre 0xFFFF è riservato.

**NB.** I campi **Pri**, **CFI** e **VLAN identifier** fanno parte in realtà di un unico campo logico chiamato **Tag**.

Esiste una forma di retrocompatibilità indiretta del nuovo standard col precedente. In particolare, non è necessario che gli *host* supportino le *VLAN*<sup>3</sup> in quanto gli *switch* possono manipolare i *frame* per aggiungere e togliere le informazioni necessarie. Quando una porta di uno *switch* riceve un *frame* “non taggato” aggiunge i campi necessari specificando come *VLAN ID* l'ID della *VLAN* associata a quella porta e allo stesso modo, quando deve inoltrare un *frame* verso un *host*, rimuove i campi aggiuntivi.

Se invece uno *switch* che non supporta le *VLAN* riceve un *frame* “taggato” ci sono due possibilità: nel caso peggiore, lo *switch* non riconosce il formato del *frame* e quindi lo scarta. In alternativa, se lo *switch* conosce lo standard, ma non è stato configurato, inoltra il *frame* senza considerare le *VLAN*.

Un *host* che usa le *VLAN* può inviare *frame* con *VLAN ID* diversi a seconda del tipo di dati che sta trasmettendo.

## 6.6 Organizzazione e funzionamento delle reti wireless

### 6.6.1 Elementi di una rete wireless

Gli elementi fondamentali di una rete wireless sono tre:

- *Host wireless*: sono i dispositivi utente in grado di connettersi alla rete e possono essere sia statici che mobili;
- *Stazione base*: è un dispositivo relay che fa da ponte tra la rete wireless e la rete cablata e infatti tipicamente è collegato ad una rete cablata;
- *Collegamenti wireless*: sono i mezzi trasmissivi che permettono ad *host* e *stazioni base* di comunicare. L'accesso al mezzo è gestito da appositi protocolli **MAC** e tipicamente, la *velocità di trasmissione* diminuisce all'aumentare della distanza;

Le reti wireless sono realizzabili secondo due modalità:

- *Infrastrutturata*: le *stazioni base* connettono gli *host* alla rete cablata e con un'operazione di *handover* gli *host* mobili possono cambiare la *stazione base* a cui collegarsi;
- *Ad hoc*: non esistono *stazioni base* e ogni *host* può comunicare soltanto con gli altri *host* all'interno del proprio raggio di copertura. Gli *host* si connettono direttamente tra loro con un processo di scoperta automatica dei vicini;

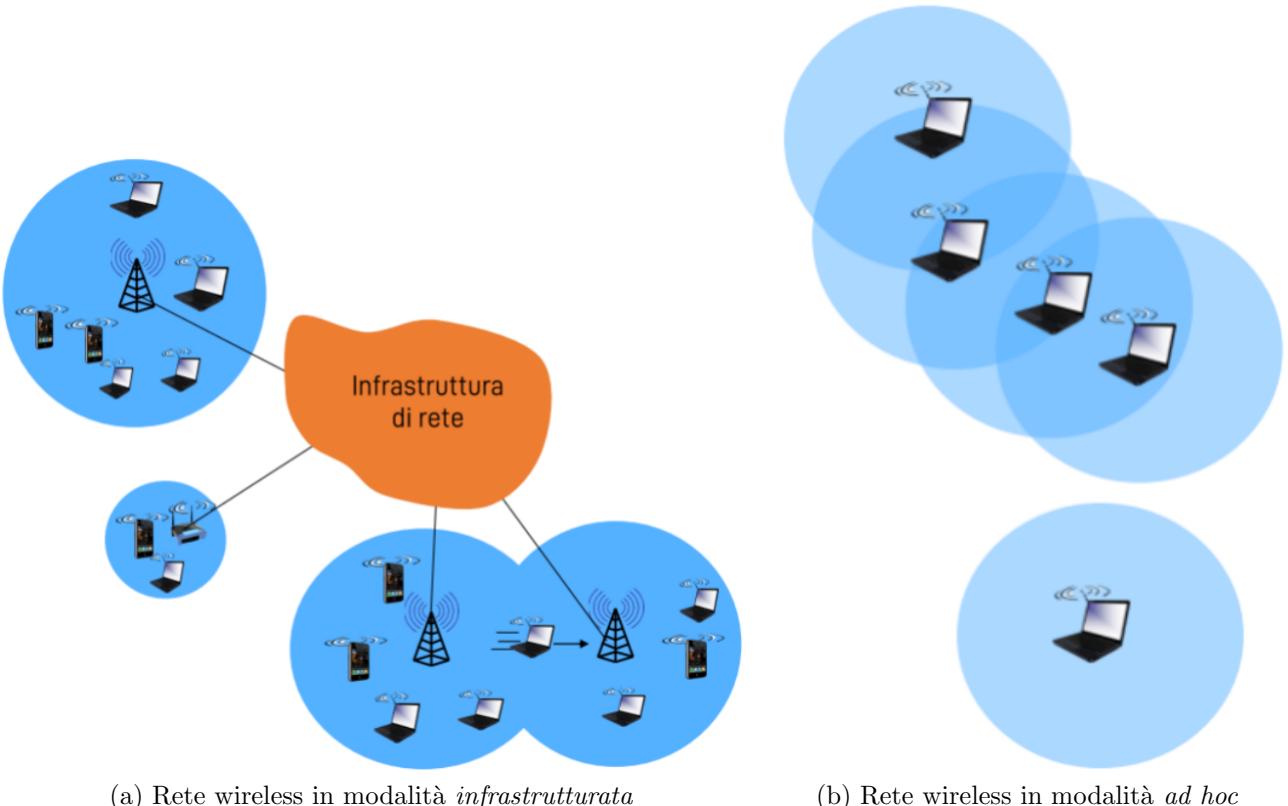


Fig. 6.22: Tipologie di reti wireless

**NB.** Nelle reti wireless realizzate con modalità *ad hoc* gli *host* più distanti, che quindi non sono direttamente collegati tra loro, comunicano per mezzo di percorsi multi-salto.

Detto questo possiamo classificare le reti wireless sia in base al tipo di infrastruttura che alla presenza o meno di percorsi multi-salto.

	Non multi-salto	Multi-salto
<b>Rete con modalità infrastrutturata</b>	L' <i>host</i> si connette alla <i>stazione base</i> che lo collega a internet	L' <i>host</i> trasmette il messaggio attraverso altri <i>host</i> prima di raggiungerne uno connesso a internet
<b>Rete con modalità ad hoc</b>	Non esiste nessuna <i>stazione base</i> e quindi nessun <i>host</i> può connettersi ad internet	Comunque non è possibile connettersi ad internet, ma la destinazione potrebbe lo stesso essere raggiunta

## 6.6.2 Architettura di riferimento

Solitamente le reti **WLAN** costruite con la modalità *infrastrutturata* si basano su un'architettura standard. Secondo questo modello di riferimento, tutti gli *host* e le *stazioni base* vengono chiamati *STA*. Gruppi di *STA* che usano lo stesso canale vengono raggruppati all'interno di un **BSS**. In ogni *BSS* è presente un **AP** che è un dispositivo in grado di connettere gli *STA* di un *BSS* al sistema di distribuzione, il quale, fornisce connettività verso altri *BSS* e, mediante

<sup>3</sup>Gli *host* possono non essere neanche a conoscenza dell'esistenza delle *VLAN*

un portale, anche verso internet. Infine, gruppi di *BSS* possono essere aggregati in un *ESS* a formare un'unica rete logica.

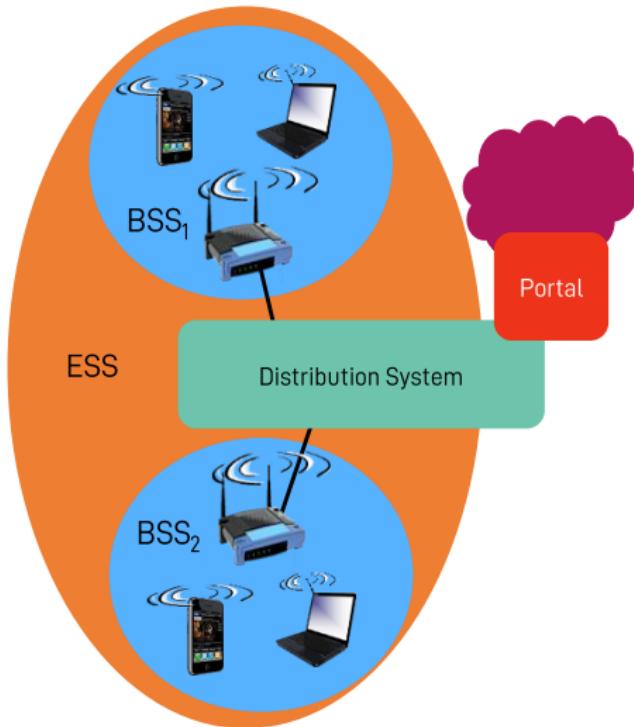


Fig. 6.23: Architettura di riferimento delle *WLAN*

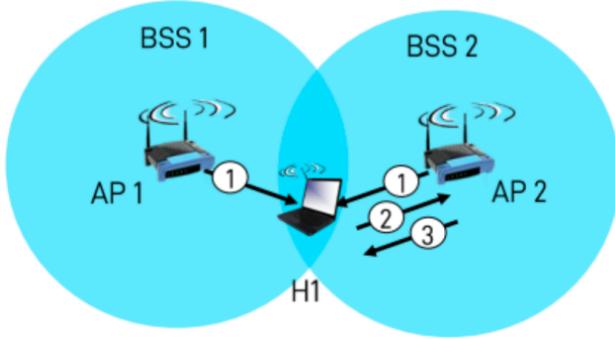
Specificando maggiormente, possiamo affermare che un *BSS* sia un insieme di *STA* che usano lo stesso protocollo *MAC* e competono per l'accesso allo stesso mezzo trasmittivo. I protocolli *MAC* possono funzionare in modo distribuito o essere controllati dagli *AP* che, di fatto, funzionano come degli *switch*. Inoltre, tutti gli *STA* vedono il proprio *ESS* come un'unica rete e quindi non sono consapevoli della divisione in *BSS*.

**NB.** I *BSS* possono essere isolati disattivando o rimuovendo i suoi *AP*.

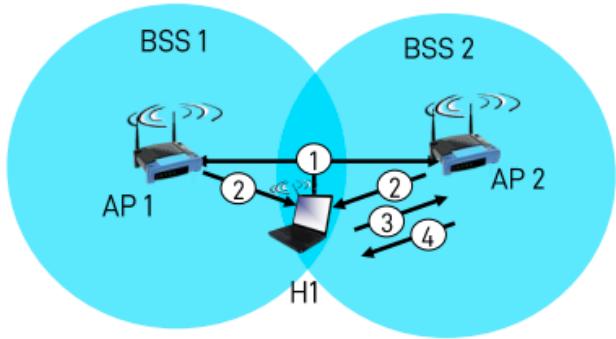
### 6.6.3 Protocolli per le comunicazioni wireless

Le comunicazioni wireless sono regolate dallo standard 802.11b che suddivide lo spettro di frequenze tra 2.4GHz e 2.485GHz in 11 canali. L'amministratore di ogni *AP* sceglie uno di quei canali e, se due *AP* vicini scelgono lo stesso, si creano interferenze. A quel punto gli *host* si devono associare ad un *AP* e, per farlo, rimangono in ascolto, su tutti i canali, alla ricerca di *frame* particolari chiamati *“beacon”*. Un *beacon* contiene un *SSID*, ovvero il nome di una rete, e l'*indirizzo MAC* dell'*AP* che l'ha inviato. A quel punto, l'*host* sceglie a quale *AP* associarsi, eventualmente autenticandosi, e mediante *DHCP* si fa assegnare un *indirizzo IP* di quella sottorete.

In realtà esistono due modalità di ricerca degli *AP*: *attiva* e *passiva*. Nella modalità *attiva*, un *host* invia a tutti i vicini un *frame* di *probe request* e tutti gli *AP* rispondono con una *probe response*. Nella modalità *passiva* invece, gli *AP* trasmettono i *beacon* e gli *host* rispondono inviando una richiesta di associazione all'*AP* scelto, il quale, a sua volta, risponde con un *frame* di conferma.



(a) Modalità *passiva*



(b) Modalità *attiva*

Fig. 6.24: Modalità di associazione degli *host* con gli *AP*

**Struttura di un frame wireless** La struttura dei *frame* per reti wireless è molto diversa dal corrispettivo per reti cablate.



Fig. 6.25: Struttura di un *frame* 802.11

Vediamo l'impiego di alcuni di questi campi:

- **Nav:** serve a specificare il tempo per il quale il canale rimarrà occupato;
- **Address 1:** *indirizzo MAC* di destinazione;
- **Address 2:** *indirizzo MAC* sorgente;
- **Address 3:** *indirizzo MAC* dell'interfaccia del *router* al quale l'*AP* è collegato;
- **Address 4:** è utilizzato soltanto nelle reti wireless *ad hoc*;

**Caratteristiche dei collegamenti wireless** È bene esplicitare il fatto che la comunicazione wireless è molto più difficile da realizzare rispetto alla controparte cablata. Il motivo, è che la potenza dei segnali radio tende ad attenuarsi molto durante la propagazione e quindi, i segnali arrivano a destinazione con una potenza minore. Inoltre, i segnali radio sono anche sottoposti a forti interferenze da altri dispositivi e a riflessioni che ne causano duplicazioni e sfasamenti.

**Problemi nel rilevamento delle collisioni** Per le ragioni espresse poc'anzi, la *collision detection* non è possibile nelle reti wireless e per capirne meglio il motivo può essere utile spiegare a grandi linee come si propagano i segnali radio. Quando un'antenna trasmette, il segnale si estende come sfere di raggio crescente a partire dall'antenna. Di conseguenza, la potenza impressa dall'antenna deve distribuirsi su superfici sempre più grandi la cui area cresce come il quadrato del raggio. L'attenuazione segue quindi una legge quadratica:

$$P_{rx} = k \cdot \frac{P_{tx}}{d^2}$$

dove  $d$  è il raggio della sfera, cioè la distanza dall'antenna, e  $k$  è una costante, tipicamente minore di 1, che conteggia altri fattori di attenuazione.

Un'altra cosa che va compresa, è che un'antenna non può trasmettere e ricevere contemporaneamente e che se in un singolo *AP* si mettessero due antenne, si verificherebbero autointerferenze. Se ad esempio considerassimo un *AP* con due antenne, una trasmittente e una ricevente, la ricevente capterebbe sia il segnale di un *STA* della rete che il segnale dell'altra antenna. Ovviamente, essendo la seconda antenna dell'*AP* molto più vicina alla ricevente, il suo segnale potrebbe interferire, o potenzialmente oscurare, quello proveniente dall'*STA*.

#### 6.6.4 Collision avoidance nelle reti wireless

Abbiamo visto che non possiamo implementare la *collision detection* nelle reti wireless, quindi, l'unica alternativa è la *collision avoidance*. Infatti, il protocollo *MAC* 802.11 è basato su *CSMA* con *collision avoidance*. Gli *STA* devono contendersi l'accesso al canale ogni volta che intendono trasmettere.

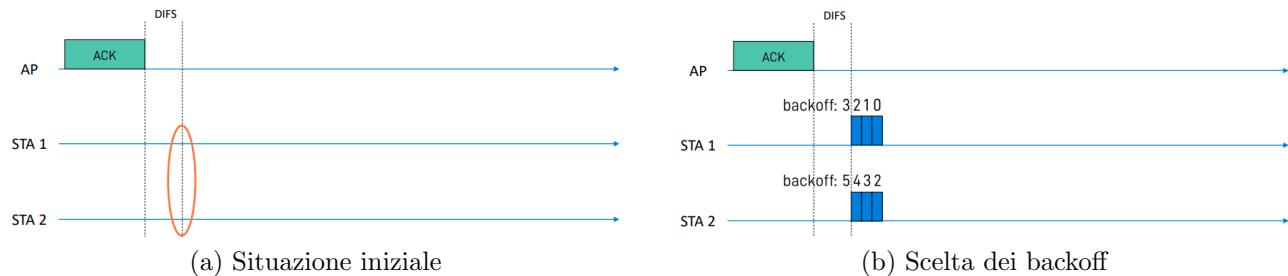
**NB.** In alcune versioni più avanzate di 802.11 è possibile concedere il canale a un *STA* per un periodo di tempo più lungo di un *frame*. Questo periodo è detto *TXOP* e permette al trasmittitore di inviare più *frame* in sequenza.

Quindi, vediamo nel dettaglio con un esempio il processo di contesa.

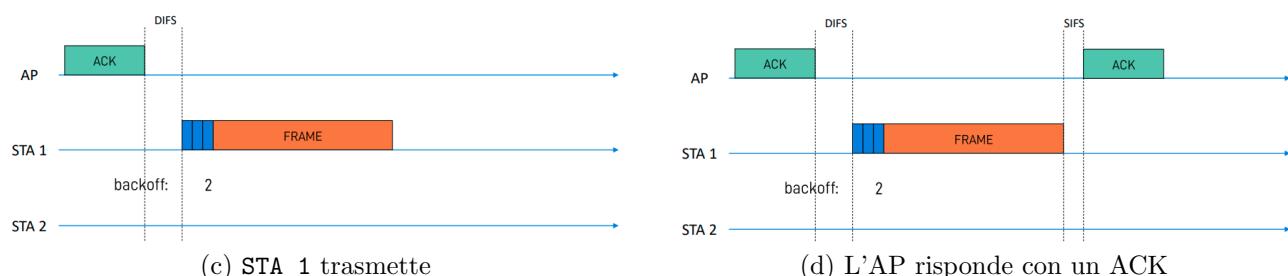
#### Esempio 16 - Esempio di funzionamento del protocollo MAC 802.11.

Supponiamo che *STA 1* e *STA 2* debbano trasmettere un *frame* all'*AP*. Entrambi i contendenti rimangono in ascolto sul canale fino a quando questo non viene percepito libero per un tempo pari a un *DIFS*. A quel punto, entrambi estraggono a caso un *backoff* tra 0 e *CW* – 1 e si mettono in attesa.

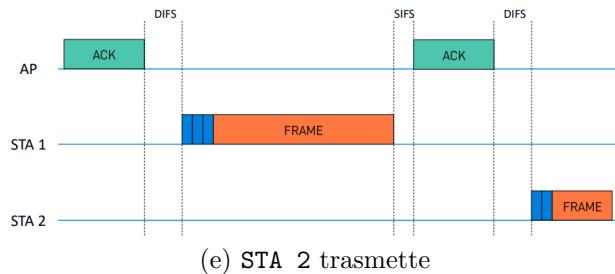
Supponiamo che *STA 1* e *STA 2* abbiano scelto un *backoff* di 3 e 5 rispettivamente.



Allo scadere del *backoff* di *STA 1*, *STA 1* testa il canale e, siccome è libero, trasmette l'intero *frame*. Quando l'*AP* ha ricevuto l'intero *frame* rimane in attesa per un *SIFS*<sup>4</sup> e quindi risponde con un *ACK*. Quando *STA 1* inizia a trasmettere, *STA 2* percepisce che il canale è occupato, quindi congela l'attuale valore di *backoff* e si mette in attesa per un tempo pari al valore scritto nel campo *NAV* dell'header del *frame* trasmesso da *STA 1*.



Allo scadere dell'attesa, del backoff e dopo aver lasciato passare un DIFS, se il canale è libero, **STA 2** trasmette il proprio frame.



Ma cosa sarebbe successo se **STA 1** e **STA 2** avessero scelto lo stesso valore di backoff?

Semplicemente, avrebbero iniziato a trasmettere nello stesso momento, si sarebbe verificata una collisione, ma senza che i due trasmettitori se ne rendessero conto. L'AP non riuscendo a ricevere nessuno dei due frame, non avrebbe trasmesso nessun ACK e quindi, dopo un DIFS, **STA 1** e **STA 2** avrebbero raddoppiato la CW e ripetuto la procedura vista in precedenza.

**NB.** Nelle reti wireless l'utilizzo degli *ACK* è necessario perché tra *collisioni*, errori di trasmissione e il problema del terminale nascosto è molto più probabile che i *frame* non arrivino a destinazione e, non essendo possibile rilevare le *collisioni*, è necessario avvisare il mittente dell'avvenuta consegna.

### 6.6.5 Problema del terminale nascosto

Alla fine della precedente sottosezione abbiano nominato il “problema del terminale nascosto”. Questo problema si verifica quando si hanno degli *STA* che vedono dei vicini comuni, ma non si vedono tra loro a causa di ostacoli o forti attenuazioni di segnale. Ciò può portare al verificarsi di *collisioni* sugli *STA* comuni.

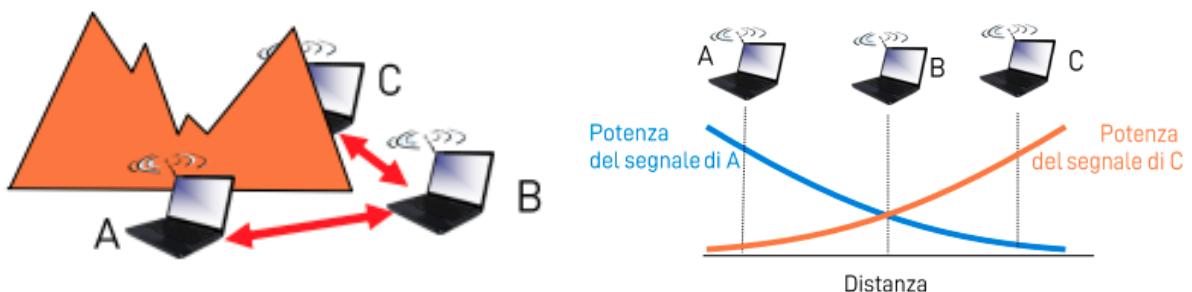


Fig. 6.24: *A* e *C* vedono entrambi *B*, ma non si vedono tra loro

Per evitare ciò, viene introdotta una procedura di *handshake* nel *CSMA/CA*. La fase di *handshake* serve per prenotare esplicitamente il canale e prevede lo scambio di due messaggi preliminari. L'*STA* intenzionato a trasmettere invia un *RTS* e il ricevitore risponde con un *CTS*. Il messaggio *CTS* serve sia a prenotare il canale per il trasmettitore che a notificare agli altri *STA* un'imminente trasmissione. Questo mitiga il problema del terminale nascosto al prezzo di un overhead maggiore.

**NB.** *RTS* e *CTS* sono messaggi brevi e se anche collidessero, la *collisione* durerebbe per meno tempo e verrebbero sprecate meno risorse radio.

<sup>4</sup> $SIFS < DIFS$

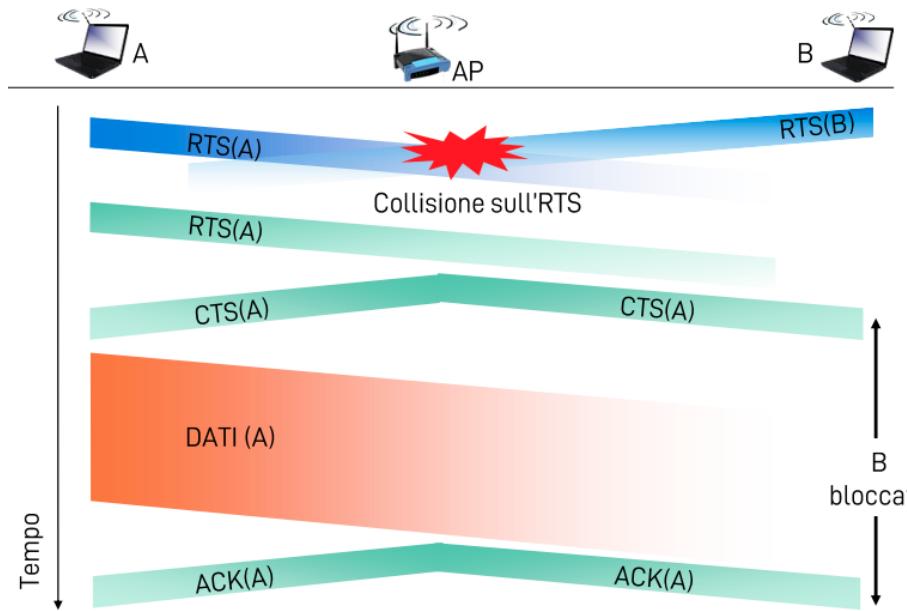


Fig. 6.25: Risoluzione del problema del terminale nascosto

Questa soluzione introduce però un ulteriore problema: il problema del terminale esposto.

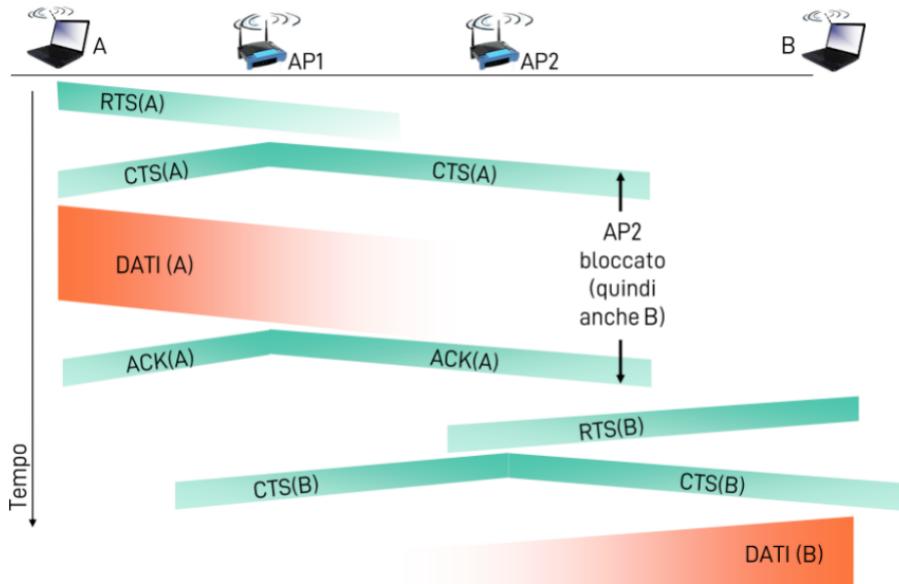


Fig. 6.26: Problema del terminale esposto

Nella situazione rappresentata nella figura di cui sopra, ad esempio, *B* è costretto a rimanere in attesa mentre *A* comunica con *AP1*. L'attesa però è inutile in quanto *B* avrebbe potuto comunicare con *AP2* senza provocare interferenze né *collisioni* su *AP1*.

A questo problema collaterale non esiste al momento una soluzione, ma lo si accetta perché il problema del terminale nascosto è più grave.

# *Capitolo Nr.A*

---

## *Simboli e formule*

---

### **A.1 Simboli**

1. *L*: dimensione in bit di un pacchetto;
2. *R*: frequenza di trasmissione (banda) di un collegamento in  $\frac{\text{bit}}{\text{s}}$ ;
3. *d*: lunghezza in m del collegamento fisico;
4. *s*: velocità di propagazione in  $\frac{\text{m}}{\text{s}}$  di un collegamento fisico;
5. *RTT*: tempo di propagazione necessario affinché un messaggio arrivi destinatario e ritorni al mittente;

### **A.2 Formule**

#### **A.2.1 Ritardo di trasmissione**

$$d_{\text{trasferimento}} = \frac{L}{R}$$

Il ritardo di trasmissione è il tempo necessario a trasmettere un pacchetto di dimensione *L* su un collegamento con frequenza di trasmissione, o banda, *R*.

#### **A.2.2 Ritardo di propagazione**

$$d_{\text{propagazione}} = \frac{d}{s}$$

Il ritardo di propagazione è il tempo necessario per propagare un segnale attraverso un collegamento fisico.

#### **A.2.3 BDP**

$$BDP = R \cdot d_{\text{propagazione}} = R \cdot \frac{d}{s}$$

Il **BDP** permette di calcolare il numero massimo di bit che in un dato momento possono transitare su un collegamento con banda *R* e ritardo di propagazione  $d_{\text{propagazione}}$ .

Questo valore massimo viene raggiunto solo se il *ritardo di trasmissione*, ovvero il rapporto  $\frac{L}{R}$ , è maggiore del *ritardo di propagazione*, cioè se:

$$d_{trasferimento} > d_{propagazione}$$

#### A.2.4 Lunghezza in metri di un bit

$$BL = \frac{d}{BDP} = d \cdot R \cdot d_{propagazione} = d^2 \cdot \frac{R}{s}$$

Il *BL* rappresenta lo spazio in *m* che deve essere coperto da un bit prima che sia possibile trasmetterne un altro.

---

## Glossario

---

### A

**ACK** Acknowledgement. 34, 37, 124

**AP** Access Point. 121

**ARQ** Automatic Repeat reQuest. 34

**AS** Autonomous System. 88

### B

**BDP** Bandwidth Delay Product. 127

**BL** Bit Length. 128

**BPDU** Bridge Protocol Data Unit. 118

**BSS** Basic Service Set. 121

### C

**CDN** Content Delivery Network. 16

**CFI** Canonical Format Indicator. 120

**CIDR** Classless Inter-Domain Routing. 66, 83

**CRC** Cyclic Redundancy Check. 106, 115

**CTS** Clear To Send. 125

**CW** Contention Window. 124

**CWND** Congestion Window. 50

### D

**DIFS** Distributed Inter-Frame Space. 124

**DSL** Digital Subscriber Line. 6

**DU** Data Unit. 12

### E

**ESS** Extended Service Set. 122

### F

**FDM** Frequency Division Multiplexing. 8

**FTTH** Fiber To The Home. 6

### H

**HOL** Head Of Line. 61

**Host** Sistema terminale. 5

**HTML** HyperText Markup Language. 18

### I

**ICANN** Internet Corporation for Assigned Names and Numbers. 65, 88

**ISP** Internet Service Provider. 5, 27, 65, 88

**IXP** Internet Exchange Point. 5

### L

**LAN** Local Area Network. 119

### M

**MAC** Media access control. 74, 103

**MSS** Maximum Segment Size. 42, 49

**MTU** Maximum Transfer Unit. 42, 64

### N

**NAT** Network Address Translator. 71

### P

**P2P** Peer-to-peer. 13

**PCI** Protocol Control Information. 12, 32

**PDU** Protocol Data Unit. 12, 34, 103

### R

**Router** Dispositivo di livello 3 che si occupa di inoltrare il traffico verso reti di un diverso dominio IP. 6, 58, 103

**RR** Resource Record. 29

**RTO** Retransmission TimeOut. 44, 51

**RTS** Request To Send. 125

**RTT** Round Trip Time. 19, 44, 49, 77

**RWND** Receiver Window. 40

### S

**SAP** Service Access Point. 11

**SDU** Service Data Unit. 12, 34  
**SIFS** Short Inter-Frame Space. 124  
**SLD** Second Level Domain. 27  
**SPOF** Single Point Of Failure. 28, 112  
**SSID** Service Set ID. 122  
**SSTHRESH** Slow Start Threshold. 50  
**Switch** Dispositivo di livello 2 che interconnette più dispositivi. 6, 114

## T

**TDM** Time Division Multiplexing. 8  
**TLD** Top Level Domain. 27

**TTL** Time To Live. 63, 77

## U

**URL** Uniform Resource Locator. 18

## V

**VLAN** Virtual Local Area Network. 119

## W

**WLAN** Wireless Local Area Network. 121

---

## *Protocoli*

---

### A

**AIMD** Additive Increase Multiplicative Decrease. [49](#)

**ARP** Address Resolution Protocol. [74](#)

### B

**BBR** Bottleneck Bandwidth and Roundtrip propagation time. [56](#)

**BGP** Border Gateway Protocol. [97](#)

### C

**CDMA** Code Division Multiple Access. [107](#)

**CSMA** Carrier Sense Multiple Access. [110](#), [124](#)

**CSMA/CA** Carrier Sense Multiple Access/- Collision Avoidance. [111](#)

**CSMA/CD** Carrier Sense Multiple Access/- Collision Detection. [111](#)

### D

**DHCP** Dynamic Host Configuration Protocol. [77](#), [122](#)

**DNS** Domain Name System. [27](#), [78](#)

### F

**FDMA** Frequency Division Multiple Access. [107](#)

**FTP** File Transfer Protocol. [24](#)

### H

**HTTP** HyperText Transfer Protocol. [18](#)

### I

**ICMP** Interne Control Message Protocol. [76](#)

**IMAP4** Internet Message Access Protocol. [26](#)

**IP** Internet Prococol. [43](#), [62](#)

### M

**MAC** Multiple Access Control. [107](#), [115](#), [120](#)

### O

**OSPF** Open Shortest Path First. [88](#)

### P

**POP3** Post Office Protocol. [26](#)

**PPP** Point-to-Point Protocol. [107](#)

### Q

**QUIC** Quick UDP Internet Connection. [57](#)

### R

**RIP** Routing Information Protocol. [95](#)

### S

**SMTP** Simple Mail Transfer Protocol. [25](#)

**SPDY** Speedy (networking protocol). [21](#)

**STP** Spanning Tree Protocol. [118](#)

### T

**TCP** Transfer Control Protocol. [17](#), [26](#), [31](#), [40](#), [97](#)

**TDMA** Time Division Multiple Access. [107](#)

**TLS** Transport Layer Security. [26](#)

### U

**UDP** User Datagram Protocol. [17](#), [31](#), [96](#)