



# UNIVERSITÀ DI PISA

Dipartimento di Ingegneria  
Corso di Laurea Triennale in Ingegneria Informatica

Appunti

Reti Informatiche

**Professori:**  
Prof. Anastasi  
Prof.ssa Righetti

**Autore:**  
Enea Passardi

---

Anno Accademico 2025/2026

# Indice

Unimap . . . . .	5
Introduzione . . . . .	9
<b>I Concetti fondamentali</b>	<b>10</b>
<b>1 Definizione di internet</b>	<b>11</b>
1 Protocolli . . . . .	12
<b>2 Struttura di internet</b>	<b>14</b>
1 Reti di accesso residenziali . . . . .	14
1.1 Reti di accesso aziendali . . . . .	15
1.2 Reti di accesso mobili . . . . .	15
2 Link di collegamento . . . . .	16
3 Il core di internet . . . . .	16
4 Struttura di internet . . . . .	18
<b>3 Analisi delle prestazioni nelle reti</b>	<b>19</b>
1 Throughput nelle reti . . . . .	20
<b>4 Modello stratificato</b>	<b>22</b>
1 Stratificazione dei protocolli . . . . .	23
2 Incapsulamento . . . . .	24
<b>II Applicazioni di rete</b>	<b>25</b>
<b>5 Principi delle applicazioni in rete</b>	<b>26</b>
1 Modello clinet-Server . . . . .	26
2 Modello Peer-To-Peer . . . . .	26
3 Modello a comunicazione di processo . . . . .	26
4 Servizi di trasporto disponibili per le applicazioni . . . . .	27
4.1 Protocolli UDP e TCP . . . . .	28
<b>6 HyperText Transfer Protocol</b>	<b>29</b>
1 Comunicazione HTTP . . . . .	29
2 Formato dei messaggi HTTP . . . . .	31
3 Interazione utente-server: i cookie . . . . .	32
4 Web caching . . . . .	33
5 Versioni del protocollo HTTP . . . . .	34

<b>7 Simple Mail Transfer Protocol</b>	<b>36</b>
1 SMTP . . . . .	37
1.1 Formato dei messaggi di posta . . . . .	38
2 Protocolli di accesso alla posta . . . . .	38
<b>8 Domain Name System</b>	<b>39</b>
1 Servizi offerti dal DNS . . . . .	39
2 Struttura del DNS . . . . .	41
3 Messaggi DNS . . . . .	41
<b>9 Applicazioni peer-to-peer</b>	<b>43</b>
1 Trasferimento file . . . . .	45
2 Protocollo BitTorrent . . . . .	46
<b>10 Trasmissione multimediale</b>	<b>48</b>
1 Streaming HTTP . . . . .	48
2 Content Distribution Network . . . . .	49
<b>III Livello di collegamento</b>	<b>50</b>
<b>11 Servizi del livello di collegamento</b>	<b>51</b>
1 Funzionalità del livello <i>data-link</i> . . . . .	52
2 Implementazione del livello di collegamento . . . . .	52
3 Rilevazione e correzione degli errori . . . . .	52
3.1 Controllo di parità . . . . .	53
3.2 Checksum . . . . .	54
3.3 Controllo a ridondanza ciclica . . . . .	54
3.4 Controllo di parità bidimensionale . . . . .	54
4 Trasferimento dati affidabile . . . . .	55
5 Protocolli a pipeline . . . . .	59
5.1 Go-back-N . . . . .	60
5.2 Selective Repeat . . . . .	60
5.3 Conclusioni . . . . .	61
6 Point-to-Point Protocol . . . . .	61
<b>12 Collegamenti ad accesso multiplo</b>	<b>63</b>
1 Protocolli a partizionamento di canale . . . . .	64
2 Protocolli ad accesso casuale . . . . .	64
3 Protocolli a rotazione . . . . .	67
<b>13 Reti locali commutate</b>	<b>68</b>
1 Ethernet . . . . .	69
2 Comunicazione nelle LAN . . . . .	70
3 Switch a livello di collegamento . . . . .	71
4 LAN virtuali . . . . .	72
5 WAN basate su switch . . . . .	74
<b>IV Livello di rete</b>	<b>76</b>
<b>14 Struttura e servizi del livello di rete</b>	<b>77</b>
1 Modelli di servizio . . . . .	78
2 Struttura del router . . . . .	79
3 Protocollo IP . . . . .	81
4 Indirizzamento IPv4 . . . . .	83
4.1 Protocollo DHCP . . . . .	84

5	Network Address Translation . . . . .	86
5.1	Port-Forwarding . . . . .	87
6	Indirizzamento IPv6 . . . . .	87
7	Address Resolution Protocol . . . . .	88
8	Internet Control Message Protocol . . . . .	90
9	Forwarding generalizzato . . . . .	90
10	Middlebox . . . . .	91
<b>15</b>	<b>Algoritmi di instradamento</b>	<b>92</b>
1	Il problema dell'instradamento . . . . .	92
2	Algoritmo di instradamento <i>link state</i> . . . . .	93
3	Algoritmo di instradamento <i>distance vector</i> . . . . .	94
4	Link state vs Distance vector . . . . .	96
5	Instradamento gerarchico . . . . .	96
5.1	BGP . . . . .	98
<b>V</b>	<b>Livello di trasporto</b>	<b>99</b>
<b>16</b>	<b>Introduzione e servizi del livello di trasporto</b>	<b>100</b>
1	Multiplexing e Demultiplexing . . . . .	100
<b>17</b>	<b>User Datagram Protocol</b>	<b>102</b>
1	Struttura di un segmento UDP . . . . .	103
<b>18</b>	<b>Transmission Control Protocol</b>	<b>104</b>
1	Struttura dei segmenti TCP . . . . .	104
2	Timeout e stima dell'RTT . . . . .	105
3	Controllo di flusso del TCP . . . . .	106
4	Controllo di congestione . . . . .	108
4.1	Controllo di congestione assistito dalla rete . . . . .	108
4.2	Controllo di congestione End-To-End . . . . .	109
5	Inizio e chiusura di una connessione TCP . . . . .	111
6	Fairness . . . . .	112
<b>VI</b>	<b>Sicurezza delle reti</b>	<b>114</b>
<b>19</b>	<b>Introduzione</b>	<b>115</b>
<b>20</b>	<b>Principi della crittografia</b>	<b>116</b>
1	Chiavi . . . . .	117
2	Crittografia a chiave simmetrica . . . . .	117
3	Crittografia a chiave pubblica . . . . .	118
4	Integrità dei messaggi e firma digitale . . . . .	118
4.1	Firma digitale . . . . .	120
5	Autenticazione . . . . .	121
<b>21</b>	<b>Sicurezza nella rete</b>	<b>123</b>
1	Sicurezza a livello applicazione e PGP . . . . .	123
2	Sicurezza a livello di trasporto . . . . .	125
3	IPsec e reti virtuali private . . . . .	127
3.1	Il datagramma IPSec . . . . .	128
3.2	IPsec security database . . . . .	129
4	Approfondimento TLS . . . . .	129

<b>22 Sicurezza operativa delle reti</b>	<b>130</b>
1 Firewall . . . . .	130
1.1 Firewall packet-filter . . . . .	131
2 Firewall stateful packet filter . . . . .	131
3 Filtro gateway . . . . .	131
4 Sistemi di rilevamento delle intrusioni . . . . .	132
 <b>VII Reti wireless e mobili</b>	 <b>134</b>
<b>23 Wireless e reti mobili</b>	<b>135</b>
1 Standard 802.11 . . . . .	137
1.1 Protocollo MAC di 802.11 . . . . .	138
1.2 Formato del pacchetto IEEE 802.11 . . . . .	140
2 Mobilità nelle sottoreti . . . . .	140
3 Funzionalità avanzate di 802.11 . . . . .	141
4 Personal Area Network . . . . .	142
 <b>24 Accesso cellulare a internet</b>	 <b>143</b>
1 Tecnologie per l'accesso alla rete mobile . . . . .	143
2 Gestione della mobilità . . . . .	143
 <b>VIII Laboratorio</b>	 <b>146</b>
<b>25 Configurazione delle reti nei sistemi UNIX</b>	<b>147</b>
1 Configurazione dell'interfaccia di rete . . . . .	147
2 Configurazione del default gateway . . . . .	148
3 DHCP . . . . .	149
4 DNS . . . . .	150
5 ICMP . . . . .	150
6 Analisi dei pacchetti . . . . .	151
 <b>26 Programmazione di applicazioni in rete</b>	 <b>152</b>
1 Socket . . . . .	152
1.1 Programmazione lato server . . . . .	154
1.2 Programmazione lato client . . . . .	154
1.3 Scambio di dati attraverso un socket . . . . .	154
2 Gestione di richieste multiple . . . . .	156
2.1 Server concorrente . . . . .	156
3 Server iterativo . . . . .	157
4 Socket UDP . . . . .	159
 <b>27 Apache HTTP Server</b>	 <b>160</b>
1 Direttive globali . . . . .	161
1.1 Direttive per siti web . . . . .	161
2 Multi-Processing Module . . . . .	162
 <b>28 Firewall</b>	 <b>164</b>
1 Network-layer Firewall . . . . .	164
2 Firewall in Linux . . . . .	165
3 Stateful filtering . . . . .	166

## Unimap

1. **Lun 22/09/2025 15:30-17:30 (2:0 h)** non tenuta: Sciopero (GIUSEPPE ANASTASI)
2. **Mar 23/09/2025 13:30-15:30 (2:0 h)** lezione: Presentazione del corso. Introduzione a Internet. La Periferia di Internet: Host, Links, Reti di accesso. Tipologie di reti di accesso. (GIUSEPPE ANASTASI)
3. **Mer 24/09/2025 15:30-17:30 (2:0 h)** lezione: Il Core di Internet. Packet Switching, Store & Forward, Internet Service Providers (ISP). Delay, Packet Loss, Throughput. Sicurezza. (GIUSEPPE ANASTASI)
4. **Ven 26/09/2025 10:30-12:30 (2:0 h)** lezione: Organizzazione a livelli. Internet/OSI protocol stack. Applicazioni di rete. Paradigmi client-server e peer-to-peer. Differenze e analogie (GIUSEPPE ANASTASI)
5. **Lun 29/09/2025 15:30-17:30 (2:0 h)** lezione: Applicazioni di rete. Requisiti delle applicazioni. Servizi disponibili su Internet. Applicazioni client-server di uso comune e relativi protocolli. Web. Protocollo HTTP: connessione persistente/non persistente, formato dei messaggi. (GIUSEPPE ANASTASI)
6. **Mar 30/09/2025 13:30-15:30 (2:0 h)** laboratorio: Introduzione e modalità d'esame. Generalità sui dispositivi connessi in rete. Indirizzi IPv4, subnet mask, default gateway. Configurare interfacce di rete nei sistemi Linux (Ubuntu), Il comando ip. Gli oggetti link, addr. Visualizzare la configurazione di rete. Abilitare e disabilitare interfacce. (FRANCESCA RIGHETTI)
7. **Mer 01/10/2025 15:30-17:30 (2:0 h)** lezione: Protocollo HTTP. Formato dei messaggi. Cookies. Caching. Conditional GET. Limiti di HTTP 1.1. HTTP 2.0 e 3.0 (cenni). (GIUSEPPE ANASTASI)
8. **Ven 03/10/2025 10:30-12:30 (2:0 h)** lezione: Posta Elettronica: Organizzazione e componenti. Protocollo SMTP. Formato del messaggi. Protocolli di accesso alla posta elettronica: IMAP. Domain Name System. Servizi. Organizzazione Distribuita e Gerarchica. Risoluzione di un nome: approccio iterativo e ricorsivo. (GIUSEPPE ANASTASI)
9. **Lun 06/10/2025 15:30-17:30 (2:0 h)** lezione: Domain Name System. Formato dei record. Esempi. Applicazioni P2P. Ricerca di Contenuti P2P: indice centralizzato; indice completamente decentralizzato (query flooding), indice gerarchico (supernodi) (GIUSEPPE ANASTASI)
10. **Mar 07/10/2025 13:30-15:30 (2:0 h)** laboratorio: Configurare interfacce di rete nei sistemi Unix con Netplan. Invio di pacchetti all'interno e all'esterno della sotto-rete di appartenenza. Configurazione del gateway. Inserimento, modifica, e ottenimento di rotte. Il Domain Name System (DNS). Configurazione tramite i file /etc/hosts, /etc/resolv.conf, ed /etc/nsswitch.conf. (FRANCESCA RIGHETTI)
11. **Mer 08/10/2025 15:30-16:30 (1:0 h)** esercitazione: Calcolo del tempo di distribuzione di un file a N utenti mediante architettura client-server e P2P (GIUSEPPE ANASTASI)
12. **Mer 08/10/2025 16:30-17:30 (1:0 h)** lezione: Applicazioni P2P per Distribuzione/Condivisione di Contenuti. Protocollo BitTorrent. Applicazioni di streaming. (GIUSEPPE ANASTASI)
13. **Ven 10/10/2025 10:30-12:30 (2:0 h)** lezione: Streaming Video. Dejittering Buffer. DASH. Content Distribution Networks (CDN). Esempi. Reti a collegamento Diretto. Collegamenti Punto-Punto. Livello Fisico e Livello Data Link. Servizi del livello Data Link. (GIUSEPPE ANASTASI)
14. **Lun 13/10/2025 15:30-17:30 (2:0 h)** lezione: Error detection (parita, checksum, CRC). Error correction (parita bi-dimensionale). Trasferimento affidabile dei dati basato su ack e ritrasmissione. Defizione di protocollo di trasferimento affidabile di dati (RDT). Varie versioni di protocollo RDT: link ideale, link che introduce errore sui bit. (GIUSEPPE ANASTASI)

15. **Mar 14/10/2025 13:30-15:30 (2:0 h)** laboratorio: Il Dynamic Host Configuration Protocol (DHCP). Come avviene l'assegnazione dinamica degli indirizzi IP. Gestione congiunta di indirizzi statici e dinamici. Server DHCP Kea. Test di connettività: il comando ping. Come interpretare l'output e come impostare i parametri per studiare le eventuali criticità della connessione. L'algoritmo traceroute. (FRANCESCA RIGHETTI)
16. **Mer 15/10/2025 15:30-17:30 (2:0 h)** lezione: Protocollo RDT: link che introduce errore sui bit, versione NAK free, link che introduce perdite. Prestazioni del protocollo RDT in modalità stop & wait e pipeline. Gestione delle perdite nei protocolli point-to-point: Stop-and-Wait, Go-back-N. (GIUSEPPE ANASTASI)
17. **Ven 17/10/2025 10:30-12:30 (2:0 h)** lezione: Gestione delle perdite nei protocolli point-to-point: Selective Repeat. Protocollo PPP: servizi, formato del frame, byte stuffing/unstuffing, negoziazione dei parametri. Reti basate su link condiviso broadcast. Protocollo di accesso multiplo: classificazione. Accesso TDMA. (GIUSEPPE ANASTASI)
18. **Lun 20/10/2025 15:30-17:30 (2:0 h)** esercitazione: Analisi dei pacchetti, tool tcpdump. Esercitazione riassuntiva degli argomenti trattati nelle precedenti lezioni con due macchine virtuali su virtualbox. Configurazione di una macchina virtuale come server DHCP, per fornire connettività all'altra. Test della connettività. (FRANCESCA RIGHETTI)
19. **Mar 21/10/2025 13:30-15:30 (2:0 h)** esercitazione: Protocolli di accesso multiplo random-based. Protocollo Aloha: algoritmo di accesso, prestazioni. Limiti del protocollo Aloha. Protocolli CSMA. Protocolli di accesso a rotazione (polling, token based). (GIUSEPPE ANASTASI)
20. **Mer 22/10/2025 15:30-16:30 (1:0 h)** lezione: Reti Locali: indirizzamento, topologie, protocollo MAC. Reti Locali Ethernet: Formato del frame. Tipo di servizio. Protocollo MAC (CSMA/CD). Standard di livello fisico. (GIUSEPPE ANASTASI)
21. **Mer 22/10/2025 16:30-17:30 (1:0 h)** lezione: Calcolo della dimensione minima del frame Ethernet (GIUSEPPE ANASTASI)
22. **Ven 24/10/2025 10:30-12:30 (2:0 h)** lezione: Reti packet-switched. Link-layer switch. Capacità di self-learning. Switched Ethernet. Proprietà di switched Ethernet. Instradamento in switched Ethernet. Reti packet-switched wide-area. Servizio di tipo connectionless e connection. Instradamento nelle due tipologie di servizio. Link virtuale. Interconnessione di reti: concetti preliminari. (GIUSEPPE ANASTASI)
23. **Lun 27/10/2025 15:30-17:30 (2:0 h)** laboratorio: Introduzione alla programmazione distribuita, il paradigma client-server. Richiami sul linguaggio C. Dichiarazione di variabili e strutture, gestione dinamica della memoria: malloc e free. Ingresso e uscita tramite scanf e printf. Gestire le stringhe. Lettura e scrittura su file. Panoramica sui socket e system call per la gestione dei socket. Primitiva socket(). Endianess e funzioni di conversione. Indirizzi IP in formato presentation e network. Instaurazione di una connessione TCP: la primitiva accept() (FRANCESCA RIGHETTI)
24. **Mar 28/10/2025 13:30-15:30 (2:0 h)** esercitazione: Interconnessione di reti. Concetti preliminari. Routing vs. Forwarding. Piano dei dati e Piano di Controllo. Protocolli di Routing e SDN. Router: struttura interna e funzionalità. (GIUSEPPE ANASTASI)
25. **Mer 29/10/2025 15:30-17:30 (2:0 h)** lezione: Protocollo IP. Formato del datagram IP. Frammentazione dei datagram. Esercizio. Schema di indirizzamento IP. CIDR. Acquisizione degli indirizzi. Assegnazione dinamica degli indirizzi. Protocollo DHCP. (GIUSEPPE ANASTASI)
26. **Ven 31/10/2025 10:30-12:30 (2:0 h)** lezione: Protocollo NAT/PAT. Esempio. IPv6: formato datagram, transizione verso IPv6. (GIUSEPPE ANASTASI)

27. **Lun 03/11/2025 15:30-17:30 (2:0 h)** esercitazione: Instaurazione di una connessione TCP: la primitiva connect(). Listening socket e socket connesso. Scambio di dati tramite send() e recv(). Esercitazione sulla prima parte di programmazione di applicazioni distribuite con i socket. Implementazione di un TCP "Hello Server", monoprocesso, che scrive "Hello!" ai client che si connettono. Mostrata esecuzione e soluzione. Consegnata per l'esercizio da svolgersi per casa su "Echo Server". (FRANCESCA RIGHETTI)
28. **Mar 04/11/2025 13:30-15:30 (2:0 h)** lezione: Transizione verso IPv6: dual architecture, tunneling. Conversione degli indirizzi IP in indirizzi MAC corrispondenti. Protocollo ARP. Protocollo ICMP. Forwarding generalizzato. Modello Match+Action. Open Flow. Esempi. Middleboxes. Evoluzione di Internet. Introduzione al livello trasporto. (GIUSEPPE ANASTASI)
29. **Mer 05/11/2025 15:30-17:30 (2:0 h)** lezione: Servizi del livello di Trasporto. Multiplexing e demultiplexing dei messaggi in modalità connectionless (UDP) e connection-oriented (TCP). Protocollo UDP: formato del pacchetto, calcolo del checksum. Protocollo TCP. Apertura e Chiusura della connessione. (GIUSEPPE ANASTASI)
30. **Ven 07/11/2025 10:30-12:30 (2:0 h)** laboratorio: Mostrata esecuzione e soluzione. Consegnata per l'esercizio da svolgersi per casa su "Echo Server". Server concorrente. Primitiva fork() e implicazioni della clonazione di processi sui socket. Esempio di server concorrente. Socket bloccanti e non bloccanti. Gestione delle attese e comparazione con i socket bloccanti. Introduzione all'I/O multiplexing. File descriptor sets. Le macro FD\_SET, FD\_ISSET, FD\_CLR, ed FD\_ZERO. La primitiva select(). (FRANCESCA RIGHETTI)
31. **Lun 10/11/2025 15:30-17:30 (2:0 h)** non tenuta: Lezione non tenuta a causa di altro impegno istituzionale del docente (GIUSEPPE ANASTASI)
32. **Mar 11/11/2025 13:30-15:30 (2:0 h)** esercitazione: Esercitazione sulla seconda parte di programmazione di applicazioni distribuite con i socket. Implementazione di un TCP "Echo Server", multi-processo, che serve più di un client alla volta. Mostrata esecuzione e soluzione. Strutture per la gestione delle date. Consegnata dell'esercizio da svolgersi per casa "Time Server". Text and binary protocols (FRANCESCA RIGHETTI)
33. **Mer 12/11/2025 15:30-17:30 (2:0 h)** lezione: Protocollo TCP: trasferimento affidabile dei dati, calcolo del Time-out, stima del RTT. Considerazioni sull'algoritmo di stima del RTT. Impostazione del Timeout Interval. Fast retransmission. ACK generation. Controllo del flusso. (GIUSEPPE ANASTASI)
34. **Ven 14/11/2025 10:30-12:30 (2:0 h)** lezione: Controllo di flusso: calcolo dello spazio libero e della finestra di trasmissione. Controllo di Congestione. Controllo Network-assisted e End-to-end. Controllo di congestione nel protocollo TCP. Fasi di Slow Start e Congestion Avoidance. Fairness. (GIUSEPPE ANASTASI)
35. **Lun 17/11/2025 15:30-17:30 (2:0 h)** laboratorio: Risoluzione collaborativa con spiegazione dell'esercizio sul time server UDP e poi TCP dato per casa. Presentazione progetto. Apache HTTP server. Richiami sul protocollo HTTP. Avvio, arresto, riavvio, e visualizzazione dello stato con i comandi apache2ctl e service. Apache HTTP server. File di configurazione, struttura, direttive e direttive contenitore. Direttiva include. Configurazioni e moduli disponibili da abilitare. (FRANCESCA RIGHETTI)
36. **Mar 18/11/2025 13:30-15:30 (2:0 h)** lezione: Introduzione alla sicurezza. Sicurezza di rete. Minacce alla sicurezza. Aspetti della sicurezza. Riservatezza della comunicazione. Crittografia a chiave simmetrica. Algoritmi di cifratura a chiave simmetrica. Centro di distribuzione delle chiavi (KDC). Crittografia a chiave pubblica (GIUSEPPE ANASTASI)
37. **Mer 19/11/2025 15:30-17:30 (2:0 h)** lezione: Integrità del messaggio. Funzioni Hash (MD5, SHA1). Algoritmi per generare il MAC. Firma digitale. Certification Authority. Endpoint Authentication. (GIUSEPPE ANASTASI)

38. **Ven 21/11/2025 10:30-12:30 (2:0 h)** laboratorio: Server HTTP Apache: Le direttive globali ServerRoot, KeepAlive, KeepAliveTimeout, Listen, ed ErrorLog. Virtual hosting. Abilitazione e disabilitazione di virtual host: i comandi a2ensite/a2dissite. Direttiva VirtualHost, ServeName, DocumentRoot. Apache HTTP server, MPMs. Panoramica sull'utilizzo di processi e thread. L'MPM prefork, worker e event. Correzione esercizi sulla configurazione di Apache. (FRANCESCA RIGHETTI)
39. **Lun 24/11/2025 15:30-17:30 (2:0 h)** laboratorio: Correzione esercizi sulla configurazione di Apache. Firewall, packet filtering, funzionamento nei sistemi Linux. Formato delle regole e processazione, tabelle e chain nei sistemi Linux. Configurazione del firewall. Sintassi dei comandi iptables -L, -A, -I, e -D per listare, aggiungere, inserire e rimuovere regole. Stateful filtering. Introduzione al problema del routing. Data plane e control plane. (FRANCESCA RIGHETTI)
40. **Mar 25/11/2025 13:30-15:30 (2:0 h)** lezione: Routing: Concetto di costo e suo significato: distanza, congestione, single-hop. Come considerare più obiettivi per determinare il path ottimo. Algoritmi link state e loro funzionamento: algoritmo di Dijkstra. Il problema delle oscillazioni. Algoritmi distance vector. Equazione di Bellman-Ford per la stima dei Distance vector. Determinazione del next hop. Aggiornamento delle stime. Confronto fra algoritmi link state e distance vector: complessità, convergenza, robustezza. (FRANCESCA RIGHETTI)
41. **Mer 26/11/2025 15:30-17:30 (2:0 h)** lezione: Sicurezza a livello applicazione: e-mail sicura, PGP. Sicurezza a livello trasporto. Transport Layer Security (TLS). Sicurezza a livello di rete. IPSec. Reti Privati Virtuali. Protocollo ESP. (GIUSEPPE ANASTASI)
42. **Ven 28/11/2025 10:30-12:30 (2:0 h)** lezione: Firewall e sistemi anti-intrusione. RETi Wireless e Mobili. Reti wireless con infrastrutture e senza infrastruttura (ad hoc). Classificazione di reti wireless (GIUSEPPE ANASTASI)
43. **Lun 01/12/2025 15:30-17:30 (2:0 h)** lezione: Routing gerarchico. Concetto di autonomous system (AS). Algoritmi intra-AS e inter-AS. Routing inter-AS: il border gateway protocol (BGP). Concetto di policy. Sessioni eBGP e iBGP. Formato del BGP advertisement. Gli attributi AS-PATH e NEXT-HOP. Selezione della migliore rotta (shortest AS-PATH). Hot potato routing. Aggiornamento della tabella di forwarding. Considerazioni conclusive. (FRANCESCA RIGHETTI)
44. **Mar 02/12/2025 13:30-15:30 (2:0 h)** lezione: Caratterizzazione dei link wireless. Classificazione di reti wireless. Reti con infrastruttura, ad hoc, ibride. Reti WiFi. Protocollo CSMA/CA. Problema del nodo nascosto. Virtual Carrier Sensing. Formato del frame. (GIUSEPPE ANASTASI)
45. **Mer 03/12/2025 15:30-17:30 (2:0 h)** lezione: Seminario Prof. Lenzini: Origini e Prospettive di Internet (FRANCESCA RIGHETTI)
46. **Ven 05/12/2025 10:30-12:30 (2:0 h)** lezione: Reti WiFi: Rate Adaptation, Power management, Gestione della mobilità. Cenni su LiFi. Wireless PANs. Bluetooth. Accesso cellulare a Internet. Architetture di una rete cellulare 4G (GIUSEPPE ANASTASI)
47. **Mar 09/12/2025 13:30-15:30 (2:0 h)** esercitazione: Esercitazione finale con Q&A sul progetto (FRANCESCA RIGHETTI)
48. **Mer 10/12/2025 15:30-17:30 (2:0 h)** lezione: Evoluzione verso le reti cellulari 5G. Gestione della mobilità nelle reti cellulari e in Internet. Routing indiretto e routing diretto. Gestione della mobilità successiva. Impatto della mobilità sui protocolli di livello superiore. (GIUSEPPE ANASTASI)
49. **Lun 15/12/2025 10:30-12:30 (2:0 h)** fuori sede: Visita di istruzione al GDC di S. Piero a Grado (GIUSEPPE ANASTASI)

## Introduzione

La presente dispensa nasce dall’esperienza personale di uno studente che ha seguito il corso di Reti Informatiche nell’anno accademico 2025/2026. Quasi tutti gli argomenti trattati all’interno del corso sono riassunti all’interno della dispensa, eccetto per i seguenti:

- La trattazione sulle **Content Distribution Network** può risultare poco approfondita.
- Non è approfondita la spiegazione sul funzionamento di **4G e 5G**.
- Mancano alcune informazioni sul protocollo **BGP** e sulla tipologia di messaggi.

Si consiglia comunque di affidarsi anche all’utilizzo del libro per avere delle spiegazioni e/o dei dettagli aggiuntivi che potrebbero essere richiesti in sede di orale. Il prof. Anastasi è, infatti, molto puntiglioso e pretende una certa comprensione degli argomenti del corso; è inoltre consigliato imparare e memorizzare degli schemi che spieghino gli argomenti richiesti in sede di orale, in quanto capita spesso che vengano richiesti dal professore.

## Argomenti del corso

Il corso di Reti Informatiche si propone di fornire una visione completa dell’infrastruttura di Internet, integrando le basi teoriche con l’esperienza pratica di laboratorio. Per affrontare con successo la materia, è fondamentale possedere conoscenze pregresse relative all’architettura dei calcolatori, alla gestione dei processi e alla programmazione in linguaggio C, poiché il progetto d’esame e le esercitazioni richiedono l’uso delle primitive socket per la comunicazione tra processi. Il percorso didattico inizia con una distinzione tra la visione utente e quella ingegneristica della rete, analizzando i componenti del nucleo e della frontiera, i modelli di commutazione di pacchetto e di circuito, e i parametri fondamentali per la valutazione delle prestazioni come il throughput e il ritardo.

L’analisi dei livelli protocollici parte dalle applicazioni di rete, dove vengono messi a confronto i paradigmi client-server e peer-to-peer attraverso lo studio di protocolli come HTTP, DNS e BitTorrent. Scendendo verso il livello di trasporto, il corso approfondisce le differenze tra TCP e UDP, concentrandosi sui meccanismi di trasferimento affidabile dei dati, sulla gestione del timeout e sul controllo della congestione. Proseguendo verso il livello network, viene introdotta l’astrazione dell’internetworking e il protocollo IP, con particolare attenzione alle funzioni di routing e forwarding, alla frammentazione dei datagrammi e alla gestione degli indirizzi tramite NAT per far fronte alla scarsità di IPv4.

La trattazione prosegue con il livello di collegamento, dove si studiano i metodi per la mitigazione degli errori e i protocolli di accesso multiplo per la gestione delle collisioni in reti LAN ed Ethernet, includendo il concetto di reti virtuali VLAN. Un’ampia sezione è dedicata alla sicurezza informatica, che spazia dalla crittografia simmetrica e pubblica alla firma digitale e all’uso di firewall e sistemi di rilevamento delle intrusioni. Il programma è completato da un’introduzione alle reti wireless e mobili, che affronta sfide specifiche come la path loss e il problema del nodo nascosto, e dalle attività di laboratorio focalizzate sulla configurazione di rete in ambiente Linux, la programmazione di server web e lo studio degli algoritmi di instradamento gerarchico. L’intero insegnamento è guidato dalla consapevolezza ingegneristica che non esistono soluzioni universali, ma che ogni decisione tecnologica dipende strettamente dal contesto applicativo.

## Licenza

Quest’opera è distribuita con licenza Creative Commons “Attribuzione – Non commerciale – Condividi allo stesso modo 3.0 Italia”.



# Parte I

## Concetti fondamentali

# Capitolo 1

## Definizione di internet

Il termine *internet* non ha una definizione precisa e strutturata; tuttavia possiamo provare a definire internet partendo dal definire il punto di vista da cui lo guardiamo. In particolare, possiamo vedere internet sotto due punti di vista:

- Il punto di vista a **dadi e bulloni**, tipico dei programmatori e di chi studia internet dall'interno.
- Il punto di vista dell'utente medio che usa internet senza conoscerne la struttura.

Se guardassimo internet dal primo punto di vista potremmo definire internet come *una rete di calcolatori che interconnette centinaia di milioni di dispositivi (host) in tutto il mondo*. All'interno di questa grande foresta non troviamo solamente gli **host**, ma anche altri dispositivi fondamentali come: i **commutatori di pacchetti** le **reti di collegamenti**. Nello specifico, i sistemi periferici (analoghi a host) sono connessi tra di loro attraverso commutatori di pacchetti e reti di collegamenti. D'altra parte, le reti di collegamento diverse possono essere costituite da mezzi fisici diversi e quindi trasmettere a velocità diverse (dove la velocità viene misurata in **bit/secondo**) e sarà nostro compito studiare anche questo aspetto. Possiamo quindi schematizzare internet come:

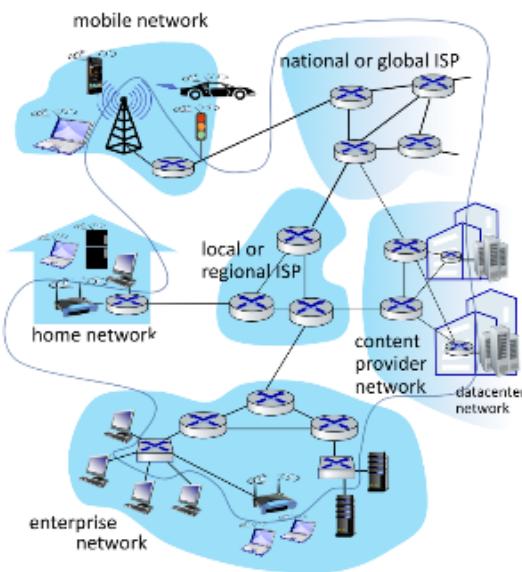


Figura 1.1: Rappresentazione schematica della struttura di internet

Il termine *reti di calcolatori* è di fatto **obsoleto**, infatti nelle realtà di oggi la maggior parte dei

dispositivi connessi non sono più calcolatori, ma vi è tutta una serie di dispositivi che possono essere connessi alla rete (basti pensare anche a un condizionatore intelligente). I sistemi periferici accedono a internet attraverso i cosiddetti **ISP** (o **Internet Service Provider**) e anch'essi possono essere di diverse tipologie.

Tuttavia, oltre al punto di vista dell'ingegnere, abbiamo anche una visione "dall'alto", la visione di chi conosce internet dall'esterno e vede tale infrastruttura come un **provider di servizi** che mette a disposizione delle applicazioni che permettono a utenti diversi di comunicare. Durante il corso seguiremo, ovviamente, la visione a **dadi e bulloni**.

## 1 Protocoli

Vista la struttura così tanto particolare di internet diventa fondamentale stabilire delle regole che regolino la comunicazione tra host e commutatori di pacchetti, tra host e host e tra commutatori di pacchetti e commutatori di pacchetti. Questo insieme di regole dovrebbe, idealmente, garantire che la comunicazione avvenga senza problematiche e che i pacchetti arrivino a destinazione integri; l'insieme di queste regole prende il nome di **protocollo**. Possiamo dare anche una definizione formale di protocollo

Un **protocollo** definisce il formato e l'ordine dei messaggi scambiati tra due o più entità in comunicazione, così come le azioni intraprese in fase di trasmissione e/o ricezione di un messaggio o di un altro evento.

I protocolli sono fondamentali in tutto, anche nella comunicazione tra umano e umano; si prenda come esempio la situazione in cui si deve chiedere l'ora ad uno sconosciuto

- Nella prima fase viene scambiata una forma di saluto, questa fase corrisponde quindi ad un handshake, necessario ad ottenere l'attenzione dell'altro interlocutore.
- Nella seconda fase si pone il quesito, chiedendo quindi l'ora. Una volta posto il quesito si rimane in attesa della risposta.
- Una volta ricevuta la risposta si ha una fase di termine della comunicazione.

Se mancasse la prima fase, ad esempio, non è garantito che l'interlocutore si accorga che gli stiamo facendo una domanda. Oppure, se nella seconda fase il primo interlocutore non si mette in attesa si potrebbe rischiare di terminare la comunicazione ancora prima di ricevere la risposta. L'utilizzo di un protocollo diventa quindi fondamentale per una corretta comunicazione. Anche la fase di terminazione è fondamentale, poiché fintanto che non vi è un termine della comunicazione esplicito il secondo interlocutore potrebbe rimanere in attesa del primo interlocutore, che magari invece (dal suo punto di vista) ha terminato la comunicazione.

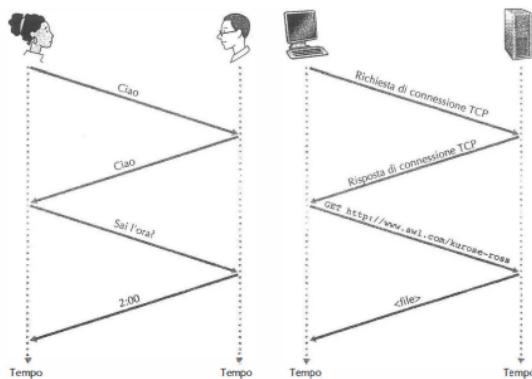


Figura 1.2: A sinistra un protocollo di comunicazione umana, a destra un protocollo di comunicazione di rete. Si nota immediatamente come i due protocolli siano sostanzialmente identici

Un protocollo di rete è quindi simile a un protocollo umano, a eccezione del fatto che le entità che

si scambiano messaggi e che intraprendono azioni sono componenti hardware o software di qualche dispositivo. Le reti di calcolatori fanno quindi uso estensivo dei protocolli, usando anche protocolli diversi per realizzare compiti diversi. Data l'importanza di questi protocolli è necessario che esiste un accordo sulle funzioni svolte da ogni singolo protocollo e, quindi, è necessario che esista uno **standard**. Gli standard di internet sono sviluppati dall'**Internet Engineering Task Force**. I documenti sugli standard sono detti **Request For Comment**. Inizialmente si trattava di richieste generiche di commenti per risolvere problemi architetturali sulle reti precedenti a internet. Ad oggi grazie a questi **RFC** possiamo definire vari protocolli, tra cui i due maggiormente utilizzati su internet: **TCP/IP**, che di fatto costituiscono una vera e propria suite.

# Capitolo 2

## Struttura di internet

Come anche detto nel capitolo precedente, gli host sono collegati alla rete internet per mezzo di reti di accesso. Formalmente definiamo la rete di accesso come quella rete che connette fisicamente un sistema al suo *edge router*, dove con *edge router* intendiamo il primo router sul percorso che un pacchetto dovrà attraversare per passare da host mittente a host destinatario. Una prima distinzione riguarda il modello che adottiamo per la nostra rete; nel nostro caso studieremo due modelli:

- Modello **Client-Server**, dove dividiamo gli host in due tipologie: **client** e **server**. Nello specifico
  - I **client** richiedono servizi.
  - I **server** offrono tali servizi
- Modello **Peer-To-Peer**, dove superiamo la distinzione netta tra client e server. Ogni host si trova allo stesso livello degli altri e prende il nome di **peer**.

Il modello **Client-Server** non è però esente da precisazioni, infatti la divisione tra **client** e **server** prevede la possibilità che un server, per fornire un servizio a un client, sia costretto a richiedere un ulteriore servizio a un altro server, diventando quindi sia client che server.

Le reti di accesso possono essere cablate o wireless e, insieme ad esse si dividono in: **reti residenziali**, **reti istituzionali** e **reti mobili**.

### 1 Reti di accesso residenziali

Con reti residenziali intendiamo reti di piccole dimensioni, composte da pochi host e che riguardano abitazioni o piccoli complessi abitativi.

La prima forma di accesso a internet per questa tipologia di reti è stata quella cosiddetta **Dial-Up-Model**. In questa tecnologia **modem** e **router** erano distinti: il modem si connette al router per mezzo della rete telefonica. Questa tipologia di rete era pensata per trasmettere un segnale vocale e non una sequenza di bit, quindi il compito del modem era eseguire una modulazione e una demodulazione delle sequenze di bit in segnale vocale e viceversa.

Oggi i due accessi residenziali più diffusi sono il **DSL (Digital Subscriber Line)** e l'accesso **via cavo**. Nella **ADSL** (dove la **A**, abbreviazione per **Asymmetric**, indica che il **bitrate in downstream** è maggiore del **bitrate in upstream**) abbiamo una banda maggiore, con una parte dedicata alle comunicazioni telefoniche e una parte dedicata solo a internet. Inoltre modem e router sono accoppiati in un unico oggetto. Invece nella rete via cavo abbiamo sempre una distinzione tra modem e router (almeno nelle vecchie reti che si appoggiavano sulla linea telefonica) e un canale unico (cavo) condiviso tra i diversi dispositivi.

La tecnologia che, ad oggi, è la più veloce e che si suppone diventerà presto anche la più utilizzata è la **fibra ottica**. In questa tecnologia non abbiamo più l'utilizzo di un segnale elettrico per mezzo di fili di rame, ma di un segnale luminoso; i vantaggi di una tale scelta sono evidenti:

- Maggiore velocità di trasmissione del segnale.
- Riduzione dei disturbi dovuti a fenomeni naturali.

Anche in questo caso abbiamo una distinzione da fare. Se la fibra arriva direttamente all'abitazione diciamo che la tecnologia è **FTTH** (*Fiber-To-The-Home*), altrimenti la tecnologia è **FTTC** (*Fiber-To-The-Cabinet*). Possiamo riassumere quanto detto fino ad ora nella seguente immagine

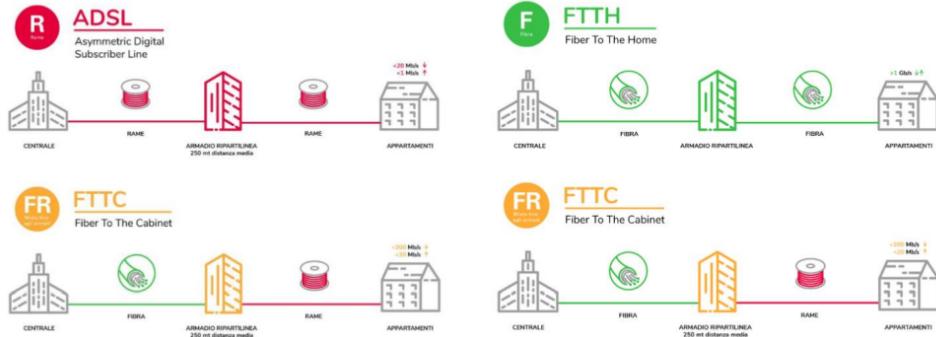


Figura 2.1: Differenze tra le varie tecnologie di rete

Le reti a fibra ottica sono - banalmente - le più veloci, con un bitrate che può arrivare a fino a un Gb/s nel caso della **FTTH**.

## 1.1 Reti di accesso aziendali

Nelle aziende e nelle università, e sempre più nelle abitazioni, per collegare i sistemi periferici ai router di bordo si utilizza una **rete locale**. Prendiamo come esempio la seguente situazione:

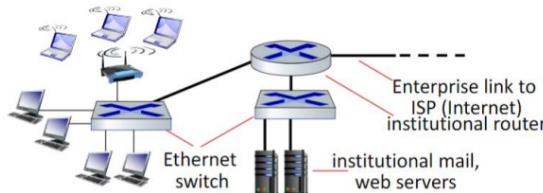


Figura 2.2: Rappresentazione di una rete istituzionale suddivisa in LAN

Esistono due principali tecnologie su cui sono basate le LAN: **tecnologia ethernet** e **tecnologia wireless**. Nelle LAN basate a tecnologia **ethernet** si connettono numerosi dispositivi per mezzo di **doppini di rame**. Tutti i doppini sono poi connessi ad un dispositivo noto come switch dove ogni dispositivo ha un canale a esso dedicato. Lo switch è successivamente connesso ad un altro switch o direttamente al router. Nelle LAN wirelessabbiamo uno o più access point presenti all'interno dell'edificio a cui si possono connettere i dispositivi entro un raggio di poche decine di metri, connesso poi a una rete aziendale, che probabilmente contiene una rete ethernet cablata, ed è a sua volta connesso ad internet.

## 1.2 Reti di accesso mobili

Questa tipologia di reti rientrano nell'insieme delle reti wireless, ma presentano delle differenze importanti. In primo luogo queste reti sono messe a disposizione da un operatore telefonico, che pone la sua infrastruttura di rete in tutto il territorio dove opera. Quindi, rispetto alla rete istituzionale si ha un volume ipotetico di traffico che è ancora maggiore vista la maggiore estensione geografica della rete. Ciò però porta ad avere delle prestazioni che sono mediamente inferiori rispetto a una qualsiasi rete istituzionale.

## 2 Link di collegamento

La comunicazione su internet avviene sempre tra due dispositivi; l'**host mittente** invia dell'informazione all'**host destinatario** per mezzo di **pacchetti**. Questi **pacchetti** sono composti da sequenze di bit (il cui significato verrà studiato in seguito) che vengono trasmesse per mezzo di un canale fisico, ma non prima di essere state convertite in un segnale fisico **compatibile** con la tipologia di canale fisico scelto. Quindi, una volta generati i bit, essi viaggiano su dei mezzi fisici che presentano forme e dimensioni eterogenee; le principali tecnologie di canali sono

- **Link guidati:** Le onde vengono contenute in un mezzo fisico.
- **Link non guidati:** Le onde si propagano nello spazio esterno.

Si potrebbe pensare a un link guidato come a un treno che viaggia sui binari, il percorso del treno è definito a priori dai binari e non ammette, generalmente, delle variazioni. Invece, si potrebbe pensare ai link non guidati come delle auto, il percorso scelto non è guidato da binari, ma vi è spazio di manovra per spostarsi o decidere il percorso. Tra i **link guidati** troviamo alcuni esempi notevoli. Una delle caratteristiche che associamo ad ogni mezzo di trasmissione guidato è una grandezza nota

Mezzo fisico	Descrizione	Caratteristiche
Doppino in rame	Due fili di rame (con guaina o meno) intrecciati a formare una spira e circondati da una guaina	
Coassiale	Due conduttori in rame concentrici e paralleli e circondati da una guaina	
Fibra ottica	Fibre di vetro avvolte da una guaina	Bitrate molto alto

Tabella 2.1: Principali tecnologie per la realizzazione di link guidati

come **bitrate**. Il **bitrate** (talvolta noto come **banda**) specifica il numero massimo di bit che si può trasmettere su quel mezzo, lo indichiamo con la lettera  $R$  e la sua unità di misura è **bit/secondo**. Supponendo che si vogliano trasferire  $L$  bit su un canale con  $R$  di bitrate, possiamo definire come il tempo necessario a trasferire il pacchetto (o **Transmission Packet Delay**) il seguente rapporto

$$\text{Transmission Packet Delay} = \frac{L}{R} [s] \quad (1)$$

Questo tempo non è tuttavia uguale al tempo di propagazione, che è invece il tempo necessario per i bit per, partendo dall'host mittente, arrivare all'host destinatario (che dipende anche dal mezzo fisico).

**Comunicazioni Wireless** Nell'ultimo periodo stanno prendendo molto piede anche le comunicazioni **wireless**, quindi comunicazioni *senza-cavo*. In questa tipologia di comunicazioni il pacchetto viaggia nell'aria e non ha quindi un canale guidato che lo guida al destinatario. Dal punto di vista dell'utilizzabilità e dell'adattabilità questo approccio è sicuramente vantaggioso, andando però a perdere diversi punti in termini di affidabilità e sicurezza.

## 3 Il core di internet

Il nucleo della rete è il centro nevralgico dello scambio delle comunicazioni tra **host**. Al suo interno non troviamo né **reti di accesso** e né **host**, ma soltanto router e altri dispositivi di instradamento. L'unico scopo di questo centro è quello di ricevere pacchetti e instradarli verso il destinatario. La gestione del core può essere fatta in due modi diversi

- **Commutazione di pacchetto:** Nella commutazione di pacchetto abbiamo dei link condivisi dove i pacchetti di vari host diversi tra loro transitano, talvolta mescolandosi tra di loro, per poi essere instradati verso il rispettivo mittente.

- **Commutazione di circuito:** La commutazione di circuito è l'approccio più utilizzato nella telefonia classica e prevede che esista un canale dedicato tra **mittente** e **destinatario** valevole per tutta la comunicazione.

Il primo approccio è quello più utilizzato al giorno d'oggi, infatti considerato l'enorme traffico di rete che esiste ogni giorno su internet e considerando anche la minor efficienza l'approccio a **commutazione di circuito** è diventato obsoleto. Ad oggi, la maggior parte dei commutatori di pacchetto utilizza la trasmissione **store-and-forward**: il commutatore riceve il pacchetto, lo memorizza e successivamente lo instrada verso una **linea di uscita** che conduce verso il destinatario finale. Prendiamo come esempio la seguente situazione

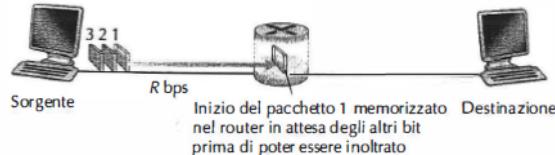


Figura 2.3: Commutazione di pacchetto **store-and-forward**

In questo esempio la sorgente deve inviare alla destinazione tre pacchetti, ognuno di  $L$  bit. All'istante mostrato in figura 2.3 la sorgente ha già trasmesso parte del pacchetto 1 e i primi bit del pacchetto uno sono giunti al router. Il router, adottando la politica **store-and-forward**, non potrà inoltrare questi pacchetti; al contrario dovrà memorizzare per intero i bit del pacchetto. Solo dopo aver ricevuto i bit del pacchetto il router inizia la trasmissione sul collegamento in uscita. La sorgente inizia la trasmissione all'istante  $t = 0$ ; all'istante  $t = L/R$  ha trasmesso l'intero pacchetto e quest'ultimo è stato memorizzato dal router (tralasciamo i ritardi di preprocessing del pacchetto nel router). In tale istante, il router, comincia a trasmettere il pacchetto sul canale in uscita e all'istante  $t = 2L/R$  il pacchetto sarà giunto a destinazione. Supponiamo ora però che ci siano  $n$  router che si infrappongono tra **sorgente** e **destinatario**, in questo caso il tempo necessario che il pacchetto impieghi a raggiungere il destinatario (supponendo l'inizio dell'emissione all'istante  $t = 0$ ) corrisponde banalmente a

$$d_{\text{end} \rightarrow \text{end}} = \text{Tempo end-to-end} = n \frac{L}{R} \quad (2)$$

Ogni commutatore di pacchetti, solitamente, riceve pacchetti da diverse sorgenti; per ciascuno di questi, il router, mantiene un **buffer di output** per conservare i pacchetti che sta per inviare su quel collegamento. I buffer di output rivestono un ruolo chiave nella commutazione di pacchetto: se un pacchetto in arrivo richiede l'invio verso la sua destinazione, ma trova occupato il canale dalla trasmissione di un pacchetto, allora si metterà in coda sul buffer. Questo fatto implica che ogni pacchetto, oltre ad un ritardo di trasmissione, sia anche soggetto a un **ritardo di accodamento** (di cui però, come ci possiamo facilmente accorgere, non possiamo fare altro che delle stime medie). Dato che la dimensione del buffer è finita, un pacchetto può trovare il buffer pieno e, quindi, andare perduto; diciamo che in questo caso si è verificato un **packet loss**

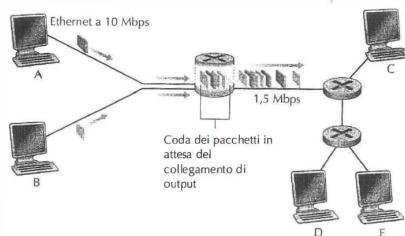


Figura 2.4: Commutazione di pacchetto con diverse fonti

La perdita di pacchetti si verifica, generalmente, quando il canale di ingresso ha una velocità di

trasmissione maggiore di quella in uscita. Possiamo immaginare il router come una diga, se la portanza in uscita è minore della quantità di acqua che entra nella diga, la diga finirà per riempirsi e strabordare. Affinché quindi si eviti la situazione che, in gergo, viene chiamata **collo di bottiglia** (*bottleneck*) è sufficiente che si presti attenzione alla differenza tra la capacità trasmissiva  $R_i$  in entrata e la capacità trasmissiva  $R_o$  in uscita.

## 4 Struttura di internet

Supponiamo di avere milioni di reti di accesso e poniamoci il problema di come connettere tutte queste reti cercando di coniugare due proprietà: sicurezza e costi. Questo è il problema che si posero gli ingegneri quando internet vide la luce. Per prima cosa enunciamo due approcci che sicuramente non rappresentano delle soluzioni corrette:

- Una grande rete di interconnessione che connette ogni rete di collegamento con ogni altra rete di collegamento. Si nota immediatamente come questa idea non sia scalabile e né tanto meno attuabile dal punto di vista economico; affinché sia possibile realizzarla sono necessari  $n^2$  collegamenti (con  $n$  numero di reti di accesso).
- Introdurre una rete di router attraverso cui viaggiano i pacchetti. La rete, in questo modello, verrebbe gestita da un unico **Internet Service Provider** che si occuperebbe di fornire servizi a un cliente dietro compenso. Questo approccio "monopolista" presenta problemi sotto diversi aspetti: economico, politico e tecnico. Economico poiché realizzare un'unica grande rete di interconnessione sarebbe costoso, politico poiché l'**ISP** avrebbe il controllo su tutta la rete in modo "dittoriale" e tecnico poiché in caso di problemi l'intera infrastruttura collassa.

L'idea vincente è quella di strutturare la rete come una serie di reti di interconnessione gestite da diversi **ISP**; in questo contesto distinguiamo due tipologie di **Internet Service Provider**: ISP di livello 1, ISP di livello 2 e ISP regionali. I primi sono **ISP** con copertura internazionale, mentre i secondi sono **ISP** che si basano su servizi e infrastrutture offerte da ISP di livello 1. Insieme agli **ISP** troviamo anche degli **Internet Exchange Point** il cui scopo è connettere i vari **ISP** e dei **Content Provider Network** il cui scopo è quello di fornire servizi collegati i loro datacenter ad internet.

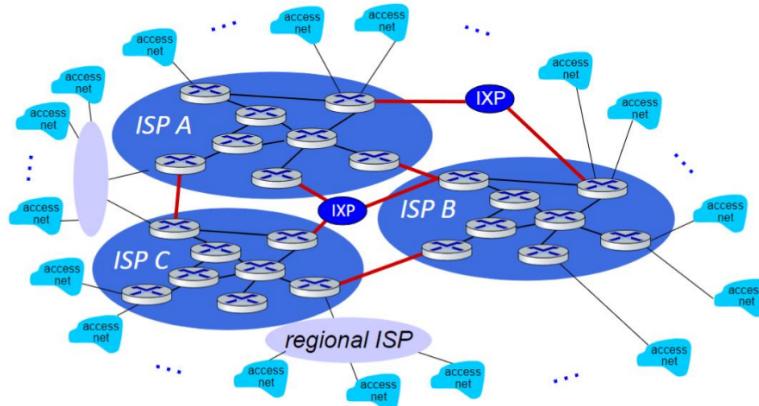


Figura 2.5: Struttura di Internet

L'esempio più famoso di **Content Provider Network** è **google**, il quale dispone di datacenter sparsi in tutto il mondo che accedono alla rete pubblica cercando di bypassare gli ISP di alto livello.

# Capitolo 3

## Analisi delle prestazioni nelle reti

Idealmente, vorremmo che i servizi internet fossero in grado di spostare una quantità di dati qualsiasi tra due sistemi periferici istantaneamente e senza alcuna perdita di dati. Nella realtà però non è possibile; nelle reti di calcolatori il **throughput**, cioè la quantità di dati al secondo che può essere trasferita tra due sistemi periferici, è necessariamente limitato dalla presenza di ritardi (di varia natura) e dalla possibilità che alcuni pacchetti vadano persi. Le fonti del ritardo nella trasmissione di un pacchetto possono essere diverse, nel nostro caso ne distinguiamo quattro:

- Un ritardo di **preprocessing** ( $d_{\text{proc}}$ ), definito come il tempo richiesto per **esaminare l'interrogazione del pacchetto** e per **determinare dove dirigerlo** (in realtà possono esserci anche altri contributi a questo ritardo, che per ora però consideriamo trascurabili). Nei router ad alta velocità questi ritardi sono generalmente trascurabili e nell'ordine dei millisecondi.
- Un ritardo di **queueing** ( $d_{\text{queue}}$ ) dovuto al posizionamento del pacchetto in coda. Come già detto, non è determinabile a priori e dipende dal posizionamento del pacchetto e dalla congestione della rete.
- Un ritardo di **trasmissione** ( $d_{\text{trans}}$ ) dovuto alla generazione del segnale fisico sul mezzo di comunicazione e pari al rapporto tra la lunghezza in bit del messaggio e il bitrate del canale.

$$d_{\text{trans}} = \text{Transmission Packet Delay} = \frac{L}{R}$$

- Un ritardo di **propagazione** ( $d_{\text{prop}}$ ) dovuto alla propagazione del segnale fisico sul mezzo di comunicazione. Anch'esso presenta una formula esplicita ed è definito come il rapporto tra la **lunghezza** del link di comunicazione  $d$  e la **velocità** di propagazione  $v$

$$d_{\text{prop}} = \text{Tempo di propagazione} = \frac{d}{v}$$

Occorre prestare molta attenzione alla differenza che esiste tra **ritardo di trasmissione** e **ritardo di propagazione**. Il ritardo di trasmissione quantifica il tempo necessario a trasmettere un pacchetto in uscita, ed è sola funzione della lunghezza del pacchetto e del **bitrate** del canale. Il **ritardo di propagazione** invece è la quantità di tempo richiesta da parte del router per trasmettere in uscita il pacchetto, ed è funzione della distanza tra due router.

Sommando tutti le componenti di ritardo introdotte fino ad ora possiamo definire il ritardo totale su un nodo come

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{prop}} + d_{\text{trans}}$$

Una delle componenti principali di questa equazione è il ritardo di accodamento, essa infatti è l'unica componente per cui non si hanno alcune indicazioni e si può solo fare delle stime *statistiche*. La natura di questo ritardo dipende da diversi fattori: **velocità di arrivo del traffico**, **velocità di trasmissione del collegamento** e anche **natura del traffico**. Denotiamo con  $a$  la

**velocità media di arrivo dei pacchetti nella coda**, espressa in pacchetti/secondo. Se  $L$  è la dimensione di un pacchetto, allora possiamo definire la **velocità media di arrivo dei bit in coda** come il prodotto  $La$ . A partire da questa definizione possiamo introdurre anche l'**intensità di traffico**, definita come il rapporto tra la velocità media di arrivo dei bit in coda e il bitrate del canale

$$\text{IDT} = \text{Intensità di traffico} = \frac{La}{R} \quad (1)$$

Se  $\text{IDT} > 1$ , allora la velocità media di arrivo dei bit nella coda supera il **bitrate** di uscita del canale e il sistema è **instabile**. Nel caso in cui  $\text{IDT} \leq 1$  allora la natura del traffico influisce sul ritardo di ritardo di coda, ma non si ha il rischio di congestione. Nella situazione reale però, il processo di arrivo in coda è casuale. In altre parole, gli arrivi non seguono uno schema e i pacchetti sono distanziati da quantità di tempo casuali. In questa situazione l'intensità di traffico di solito non è sufficiente a caratterizzare le statistiche sui ritardi. Nonostante ciò però questa grandezza permette comunque di dare una caratterizzazione ai ritardi di accodamento

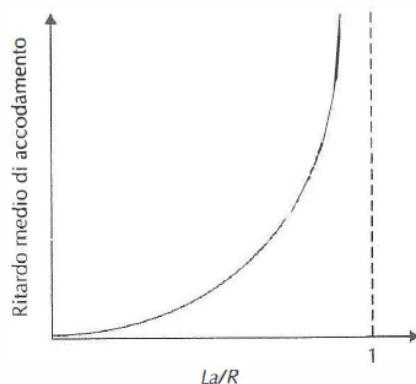


Figura 3.1: Ritardo medio di accodamento in funzione dell'intensità di traffico

Di fatto, anche nel caso in cui si verifichi una perdita di pacchetti dovuta all'accumularsi di altri pacchetti in coda al buffer non si hanno grossi problemi, è infatti sufficiente che l'host destinatario rimandi nuovamente il pacchetto.

**Comando traceroute** Per ottenere una stima del ritardo in una rete è possibile utilizzare il comando **traceroute**. Il funzionamento del comando è relativamente semplice: quando l'utente specifica il nome di un host di destinazione, il modulo utente invia un certo numero di pacchetti speciali verso tale destinazione, questi pacchetti passano attraverso dei router. Quando un router riceve uno di questi pacchetti manda un messaggio che torna all'origine e che contiene il nome e l'indirizzo del router.

## 1 Throughput nelle reti

Oltre al ritardo e alla perdita dei pacchetti un'altra misura delle prestazioni in una rete di calcolatori è un **throughput end-to-end**. Possiamo definire questa grandezza come il numero di bit trasmessi dalla sorgente alla destinazione per unità di tempo. In particolare

- Il **throughput istantaneo** in ogni istante di tempo è la velocità alla quale la destinazione sta ricevendo il file.
- Il **throughput medio** studia invece la velocità alla quale la destinazione sta ricevendo il file su un intervallo di tempo più ampio.

Supponiamo di studiare il throughput nel caso in cui abbiamo una sorgente (server) che cerca di spedire dei pacchetti a una destinazione (client)

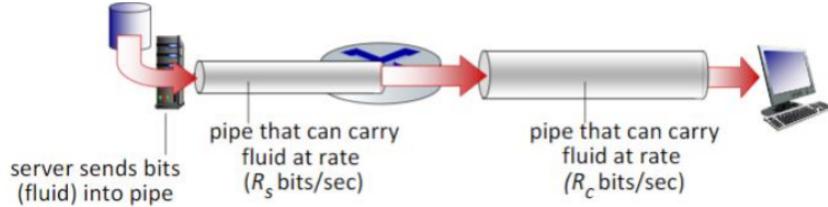


Figura 3.2: Situazione esempio

Consideriamo uno scenario ideale in cui un server comunica con un client attraverso un router intermedio. Per analizzare il *throughput* (cioè la velocità effettiva di trasferimento dei dati), immaginiamo i bit come un fluido che scorre in condotte, dove ciascun collegamento rappresenta una condotta con una certa capacità. Il server è connesso al router tramite un collegamento con capacità di trasmissione  $R_s$  bps (bit per secondo), mentre il router è connesso al client con un collegamento di capacità  $R_e$  bps.

- Se  $R_s < R_e$ , il server rappresenta il collo di bottiglia della rete. I bit fluiranno alla velocità massima supportata da  $R_s$ , e quindi il throughput end-to-end sarà pari a  $R_s$  bps.
- Se invece  $R_e < R_s$ , il router non è in grado di inoltrare i dati alla stessa velocità con cui li riceve dal server. In questo caso, il throughput sarà limitato da  $R_e$  bps.

È importante notare che, nel secondo scenario, i bit in eccesso si accumulano nel router, causando un aumento indefinito della coda se la trasmissione continua a lungo. Questa situazione è indesiderabile, poiché può portare a congestione e perdita di pacchetti.

# Capitolo 4

## Modello stratificato

Abbiamo ampiamente visto che le reti sono di fatto modelli complessi su cui operano una moltitudine di diversi attori che operano in modo simultaneo per comunicare con altri, con un traffico dati che, ad oggi, è ampiissimo. Tale struttura richiede che alla base ci sia una forma di organizzazione tale da permettere che questo traffico possa giungere, nel modo migliore possibile, a destinazione. Alla base di tutta questa organizzazione troviamo i **protocolli** (ampiamente trattati nei capitoli precedenti). L'idea che possiamo però sfruttare è quella di pensare alla rete come una qualsiasi organizzazione complessa, con delle strutture definite; prendiamo come esempio la seguente organizzazione con una struttura **gerarchica** e a **livelli**:

- Ufficio.
- Segreteria.
- Servizio Logistico.
- Corriere.
- Centro di spedizione.

Ipotizziamo ora di avere due organizzazioni, chiamate A e B, con medesima struttura in cui due *officer* cercano di comunicare scambiandosi una lettera. Affinché ciò sia possibile è necessario che avvengano tutti una serie di passaggi intermedi che coinvolgono tutti gli altri reparti: una volta scritta la lettera, l'*officer* la pone in una cassetta (egli non si pone infatti il problema di cosa succeda nelle strutture sottostanti). Successivamente la segreteria preleva, dalle varie cassette, tutte le lettere scritte dai vari *officer*, le divide in plachi e le consegna all'ufficio logistico. L'ufficio logistico, una volta ricevute le lettere, le divide in scatoloni in funzione del mittente (decide quindi un percorso per le varie lettere) e le prepara alla spedizione. Il corriere preleva i vari pacchi e li porta al centro di spedizione che, una volta ricevuti i pacchi li spedirà verso le rispettive destinazioni.

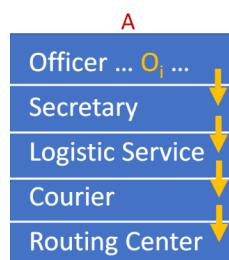


Figura 4.1: Struttura dell'organizzazione dell'organizzazione A

Il problema che ci poniamo ora è, una volta che l'organizzazione A ha spedito la lettera, come venga ricevuta dall'organizzazione B. Una volta che il pacco arriva al centro di spedizione della

destinazione verrà preso in consegna da un corriere. Il corriere porterà quindi il pacco presso il centro logistico dell'organizzazione B. Il centro logistico in questione porterà quindi il pacco alla segreteria del mittente e, infine, la lettera giungerà all'*officer* dell'ufficio B. La situazione appena ottenuta è schematizzabile nel seguente modo

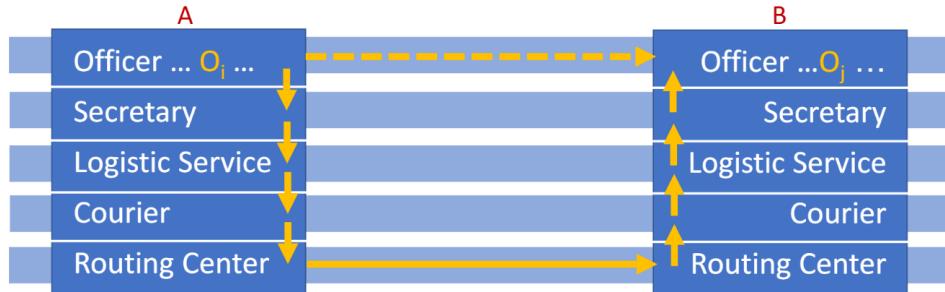


Figura 4.2: Protocollo di spedizione della lettera

Come ci accorgiamo facilmente dalla figura 4.2 la comunicazione tra i due uffici non è per nulla semplice, anzi, richiede tutta una serie di fasi intermedie e di servizi che all'*officer* sono invisibili, ma senza i quali non è possibile garantire l'arrivo della busta a destinazione.

## 1 Stratificazione dei protocolli

Dall'esempio delle due organizzazioni possiamo facilmente renderci conto che anche internet rappresenta un'organizzazione complessa e - come tale - necessita di un'organizzazione analoga, chiamata in gergo **a strati**. Ogni livello viene detto **layer** e mette a disposizione una serie di servizi ai livelli successivi. Un'organizzazione di questo tipo, oltre ai vantaggi già citati, comporta anche vantaggi per quanto riguarda la **scalabilità** del sistema stesso; per modificare un servizio non è necessario modificare servizi di altri livelli. Nel modello di internet distinguiamo cinque livelli

- Livello **applicazione**.
- Livello **trasporto**.
- Livello **rete**.
- Livello **collegamento**.
- Livello **fisico**.

Il livello di **applicazione** è la sede delle applicazioni di rete e dei relativi protocolli. In questo livello possiamo trovare tutte le applicazioni client-server che comunicano tra di loro scambiandosi messaggi. Il livello di trasporto **colleziona** e **trasferisce** i messaggi del livello applicativo; in particolare, il compito principale del livello di trasporto è mettere a disposizione una trasmissione dei dati end-to-end funzionale e sicura all'interno di una rete. A questo scopo il livello di trasporto acquisisce i dati dal livello di applicazione e li inoltra al livello di rete. All'occorrenza esso può suddividere i dati anche in unità più piccole o raggrupparli in pacchetti per trasportarli più facilmente. Un pacchetto, una volta attraversato il livello di trasporto viene chiamato **segmento**. Il livello di rete si occupa di trasferire i pacchetti da un host a un altro. In particolare, il livello di rete decide, guardando l'indirizzo di destinazione, il prossimo salto (*next-hop*) da far fare al pacchetto; con *next-hop* si intende l'indirizzo del router successivo nel percorso verso il destinatario. Un **segmento** una volta attraversato il livello di rete diventa un **datagramma**. Il livello di collegamento instrada un datagramma attraverso una serie di router tra la sorgente e la destinazione. Per trasferire un pacchetto da un nodo a quello successivo, il livello di rete si affida ai servizi del livello di collegamento, passandogli il datagramma da inviare al *next-hop*. Un **dataframe** che passa dal livello di rete al livello applicativo prende il nome di **frame**. L'ultimo livello, quello fisico, ha come compito di spostare interi frame da un elemento di rete a un altro adiacente, il ruolo del livello fisico è trasferire i singoli bit del frame da un nodo a quello successivo.

**Modello ISO/OSI** Prima del modello di internet attuale esisteva un modello analogo, ma con una sostanziale differenza: il modello **ISO/OSI**. Questo modello prevedeva l'esistenza di altri due livelli, interposti tra il livello applicazione e il livello di trasporto.

- **Livello di Presentazione:** fornire servizi che consentono ad applicazioni che permettono di comunicare di interpretare il significato dei dati scambiati. Questi servizi comprendono cifratura e compressione dei dati ad esempio.
- **Livello di Sessione:** fornisce la delimitazione e la sincronizzazione dello scambio di dati, compresi i mezzi per costruire uno schema di controllo e di recupero degli stessi.

## 2 Incapsulamento

Il concetto di incapsulamento è uno dei concetti principali dell'architettura a strati di internet. L'idea è che ogni qualvolta il pacchetto attraversa uno strato, egli aggiunga un pezzettino di informazioni a tale pacchetto; questo pezzettino di informazione, detto **header** e contiene informazioni utili al medesimo livello della pila di destinazione. Prendiamo la seguente figura

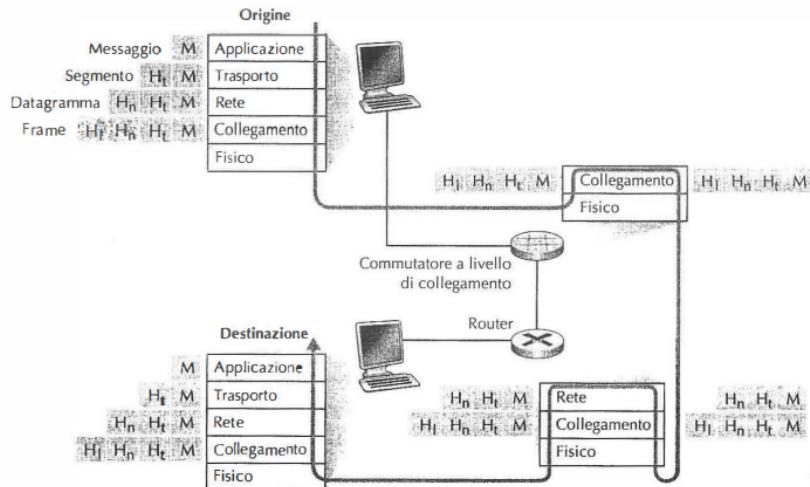


Figura 4.3: Host, Router e commutatori a livello di collegamento

Come si nota in figura 4.3 ad ogni livello viene aggiunto un piccolo **quadratino** al pacchetto, questo pacchetto (*header*) contiene delle informazioni che saranno utili al rispettivo livello della pila **destinataria**. Ad esempio, presso un host mittente un **messaggio a livello applicazione** viene passato al livello di trasporto e, nel caso più semplice, questo livello gli concatena un **header** aggiuntivo che sarà utile al livello trasporto del ricevente. Le informazioni aggiunte potrebbero includere dati che consentono al livello trasporto lato ricevente di consegnare il messaggio all'applicazione desiderata, così come potrebbero includere dei bit per il rilevamento degli errori che consentono al ricevente di determinare l'eventuale cambio di alcuni bit del messaggio. Anche nel passaggio da livello di trasporto a livello di rete vengono aggiunte informazioni aggiuntive, quali l'indirizzo del prossimo commutatore (*next-hop*) verso cui il pacchetto dovrà essere instradato. Infine, il pacchetto verrà passato al livello di collegamento, che aggiungerà altre informazioni proprie.

## Parte II

# Applicazioni di rete

# Capitolo 5

## Principi delle applicazioni in rete

La progettazione di programmi che operano in rete a livello applicativo richiede innanzitutto che venga predisposto un progetto dettagliato dell’architettura dell’applicazione, completamente diversa dall’architettura di rete. Tale architettura è infatti, per il programmatore, fissa e fornisce applicazioni per uno specifico insieme di servizi. La scelta del programmatore si rivolgerà quindi a due principali architetture: modello **Client-Server** e modello **Peer-to-Peer**.

### 1 Modello Client-Server

Nell’architettura **Client-Server** abbiamo due tipologie di host, **Client** e **Server**, dove il **client**, rappresentato solitamente da un browser, richiede un servizio al **server**, rappresentato solitamente da un **server web**. Si osservi che in una applicazione **client-server** i client non comunicano direttamente tra loro e, quindi, neanche i browser comunicano direttamente tra di loro. I server, da canto loro, devono rispettare tutta una serie di caratteristiche fondamentali affinché possano erogare un servizio al client:

- Il servizio deve essere **noto e disponibile a un indirizzo fisso**.
- Il server deve essere sempre acceso e deve avere un indirizzo IP permanente.
- Il server deve essere **ricco di risorse**.

Spesso in un’applicazione client-server la mole di traffico è tale da rendere impossibile per un singolo server gestire tutti i client. Per questo motivo i servizi vengono erogati attraverso dei data-center, quindi un’insieme di server che vengono virtualizzati e appaiono come un unico grande server.

Il client, d’altro canto, non deve essere sempre raggiungibile e inoltre può avere un indirizzo **dinamico**, quindi variabile nel tempo.

### 2 Modello Peer-To-Peer

Nell’architettura **peer-to-peer** l’infrastruttura di server in data center è minima o del tutto assente; si sfrutta, invece, la comunicazione diretta tra coppie di host, chiamati **peer (pari)**, collegati in modo intermittente. I peer non appartengono a un fornitore di servizi, ma sono dei dispositivi normalissimi come, ad esempio, dei computer; di fatto è come se fossero più vicini a client che a server. Questo fatto permette una maggiore **scalabilità** e **convenienza economica**, ma con una banda di trasmissione generalmente minore e delle problematiche relative alla **sicurezza**.

### 3 Modello a comunicazione di processo

Nel corso di calcolatori abbiamo visto due modalità con cui due processi dello stesso host comunicano tra di loro, la prima modalità, detta a **variabili condivise**, prevede che due processi

possano comunicare tra di loro attraverso delle **variabili condivise**, nella seconda modalità, detta a **scambio di messaggio**, prevede che i processi si scambino dei messaggi tra di loro. Nelle applicazioni web la prima modalità non è sostanzialmente possibile e quindi si ricorre alla seconda modalità: due host in rete comunicano tra di loro scambiandosi messaggi, intermediati dalla rete. Un processo invia messaggi nella rete e riceve messaggi dalla rete attraverso un’interfaccia software detta **socket**. Possiamo immaginare un processo come una casa e il **socket** come una **porta**; un processo che vuole inviare un messaggio a un altro processo o a un altro host, fa uscire il messaggio dalla propria porta. Il processo presuppone quindi l’esistenza di una infrastruttura esterna che possa portare il messaggio a destinazione.

Come nella posta tradizionale, affinché la consegna sia possibile, è necessario che sia il mittente che il destinatario siano identificabili in modo univoco. In rete ciò è possibile attraverso un particolare indirizzo, composto da 32 bit e noto come **indirizzo IP**. Inoltre, oltre a tale indirizzo è necessario specificare anche un indirizzo del processo di destinazione sul destinatario; tale compito è assolto dal **numero di porta**.

## 4 Servizi di trasporto disponibili per le applicazioni

Quando un processo inoltra un messaggio attraverso un socket, è compito del livello di trasporto del ricevente di consegnare i messaggi alla socket del processo ricevente. Il compito del programmatore è quindi quello di scegliere, in funzione di diversi criteri, scegliere il protocollo a livello trasporto. In generale possiamo definire tre criteri di scelta:

- Trasferimento dati affidabile
- Troughput.
- Temporizzazione.
- Sicurezza.

**Trasferimento dati affidabile** In una rete di calcolatori i pacchetti possono andare perduti. Questa perdita può avere diverse entità, come il riempimento di un buffer o la corruzione di un bit di un pacchetto. Tale perdita però non è sempre tuttavia un problema, potrebbero infatti esistere delle applicazioni che tollerano perdite (*loss-tolerant*): in particolare le applicazioni multimediali ad uso personale possono tollerare una certa quantità di dati perduti, poiché tale perdita non causa problematiche così tanto gravi. Considerato questo fatto possiamo distinguere due protocolli:

- Protocolli che **forniscono un trasferimento dati affidabile**: Un protocollo che offre questo servizio (*reliable-data-transfer*) permette al processo mittente di passare i propri dati alla socket e sapere con assoluta certezza se i dati sono arrivati senza errori al mittente.
- Protocolli che **non forniscono un trasferimento dati affidabile**: Un protocollo che non offre questo servizio non dà la possibilità al mittente di sapere se i dati sono arrivati correttamente al mittente.

**Troughput** Nel contesto delle reti e, in particolare, nel contesto di una comunicazione tra due processi è il tasso al quale il processo mittente può inviare bit al processo ricevente. Tale grandezza non è costante, infatti con connessioni (quindi socket) che si aprono e chiudono dinamicamente dividendosi la banda è impossibile determinare esattamente questa grandezza. Tuttavia, i protocolli a livello trasporto possono offrire un servizio di **troughput** disponibile garantito. Con tale servizio l’applicazione può richiedere un **troughput garantito** di  $r$  bit/s e il protocollo di trasporto assicurerà che il **troughput** sia sempre almeno di  $r$  bit/s. Tuttavia non è sempre possibile per il protocollo garantire tale **troughput** e, in tal caso, l’applicazione dovrà codificare i dati ad un livello inferiore oppure rinunciare del tutto. A tal proposito, possiamo anche in questo caso dividere le applicazioni in due tipologie

- Applicazioni **sensibili alla banda**: Applicazioni che hanno requisiti di **troughput**, come le applicazioni multimediali.

- Applicazioni **elastiche**: Applicazioni che non hanno requisti specifici di **throughput** e che possono quindi usarne tanta o poca, in funzione di quella disponibile.

**Temporizzazione** Un protocollo a livello trasporto può offrire anche servizi relativi alla temporizzazione, come ad esempio che un bit, inviato dal mittente, arrivi a destinazione entro e non oltre un tempo  $t$  definito (generalmente nell'ordine dei millisecondi). Questo tipo di servizio può interessare applicazioni che lavorano in tempo reale che, per essere efficaci, richiedono stretti vincoli temporali sulla consegna dei dati.

**Sicurezza** Infine, un protocollo di trasporto può offrire agli host servizi di sicurezza. Per esempio, nell'host mittente, un protocollo di trasporto può cifrare tutti i dati trasmessi dal processo mittente e, nell'host di destinazione, il protocollo di trasporto può decifrare tali dati prima di passarli all'applicazione. Altri servizi possono garantire l'integrità dei dati e l'autenticazione tra i due host.

## 4.1 Protocolli UDP e TCP

I due protocolli principali a livello trasporto sono: **TCP** e **UDP**. Il primo protocollo fornisce un servizio orientato alla connessione il trasporto affidabile dei dati; in particolare:

- **Servizio orientato alla connessione.** Il protocollo TCP fa in modo che *client* e *server* si scambino informazioni di controllo a livello trasporto prima che i messaggi a livello applicazione comincino a fluire. Questa procedura viene detta **handshake** e mette in allerta client e server, preparandoli alla partenza dei pacchetti. La connessione che si viene a creare è *full-duplex* e viene detta **connessione TCP**.
- **Servizio di trasferimento affidabile.** I processi comunicanti possono contare su TCP per trasportare i dati senza errori e nel giusto ordine. Oltre alla gestione dei pacchetti, TCP, mette anche a disposizione un meccanismo di controllo della congestione, il quale riguarda più che altro il benessere generatale dell'infrastruttura di rete.

Il secondo servizio, UDP, è un protocollo di trasporto leggero e senza fronzoli, dotato di un modello di servizi minimalista. UDP è senza connessione e non prevede quindi una fase di **handshake**, e fornisce un servizio di trasferimento dati **non affidabile** che non si assicura che il pacchetto arrivi a destinazione e né tanto meno che arrivi nel giusto ordine. Inoltre, **UDP** non mette a disposizione nessun meccanismo di gestione della congestione.

**NOTA:** entrambi i protocolli verranno ripresi in seguito in una parte apposita del corso.

# Capitolo 6

## HyperText Transfer Protocol

Il web, come abbiamo detto, permette a diversi dispositivi di scambiarsi risorse e di comunicare attraverso una serie di servizi offerti dai vari strati che compongono il protocollo Internet. Al centro di questi scambi troviamo le **pagine web**. Una **pagina web** può essere definita come un insieme di oggetti. Un **oggetto** è, più semplicemente, un file accessibile tramite un **URL** (*Uniform Resource Locator*), comunemente noto come indirizzo web. Ogni **URL** è a sua volta composto da due parti principali: il nome dell’*host* del server che ospita l’oggetto e il percorso dell’oggetto all’interno del server. Il protocollo HTTP, chiamato anche *HyperText Transfer Protocol*, costituisce il cuore di questo scambio di pagine web. Questo protocollo, posto a livello applicativo, è implementato in due programmi, client (su cui è implementato il browser web) e server (su cui è implementato il web server), in esecuzione su sistemi periferici diversi che comunicano tra di loro scambiandosi messaggi HTTP. Il protocollo definisce sia la struttura del messaggio e sia la modalità con cui client e server si scambiano i messaggi.

Il protocollo HTTP presenta alcune caratteristiche peculiari. La prima riguarda il modo in cui due browser possono interpretare una stessa pagina web, che potrebbe essere leggermente diverso (HTTP non si occupa quindi dell’interpretazione dei client). Inoltre, HTTP è classificato come un protocollo **senza memoria di stato** poiché i server **non mantengono informazioni sui client**.

### 1 Comunicazione HTTP

Quindi, il **browser web** implementa il lato client del protocollo HTTP, mentre, invece, un **web server** implementa il lato server del protocollo HTTP e contiene gli oggetti indirizzabili tramite URL. Il protocollo HTTP definisce in che modo i client web richiedono le pagine ai web server e come questi ultimi le trasferiscono ai client. HTTP utilizza come protocollo di comunicazione il **TCP**, l’idea è che il client inizi una connessione TCP con il server. Una volta stabilita la connessione, i processi accedono a TCP attraverso le proprie socket. Il client invia e riceve le risposte HTTP attraverso la propria interfaccia socket.

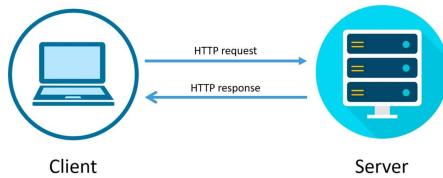


Figura 6.1: Comunicazione HTTP

In molte applicazioni per internet client e server comunicano per lunghi periodi di tempo, con richieste frequenti da parte del client che vengono gestite opportunamente dal server. A seconda dell’applicazione però, la temporizzazione delle richieste potrebbe essere diversa (richieste in

sequenza, a intervalli, periodiche, etc.). Gli sviluppatori quindi, nel creare un'applicazione che interagisce su TCP, devono prendere una decisione fondamentale:

- Ogni coppia (richiesta, risposta) viaggia sulla stessa connessione TCP.
- Ogni coppia (richiesta, risposta) viaggia su una diversa connessione TCP.

Questa scelta differenzia due tipologie di connessioni TCP, una connessione detta **persistenza** e una connessione detta **non persistente**. Ognuno di questi approcci porta con se vantaggi e svantaggi, così come delle differenze sostanziali di comportamento.

**Connessioni non persistenti** Analizziamo il comportamento della comunicazione nel protocollo HTTP con connessioni **non persistenti**. Il processo client inizia una connessione TCP con il server sulla porta 80 e, una volta stabilita la connessione, il client (attraverso la propria interfaccia socket) invia al server un messaggio di richiesta HTTP che include il percorso dell'oggetto desiderato. Il processo server riceve il messaggio di richiesta attraverso la propria interfaccia socket associata alla connessione, gestisce la richiesta (recuperando eventualmente la risorsa) e incapsula l'oggetto in un messaggio di risposta HTTP. Il processo server, una volta mandato il pacchetto di risposta, comunica a TCP di terminare la connessione; TCP chiuderà la connessione solo quando sarà sicuro che il pacchetto è giunto a destinazione correttamente. Il client HTTP riceve il messaggio di risposta e la connessione TCP termina. Introduciamo anche una grandezza, nota come **round-trip-time** che indica il tempo necessario ad un pacchetto per viaggiare dal client a server e viceversa, di cui cerchiamo di calcolare una stima

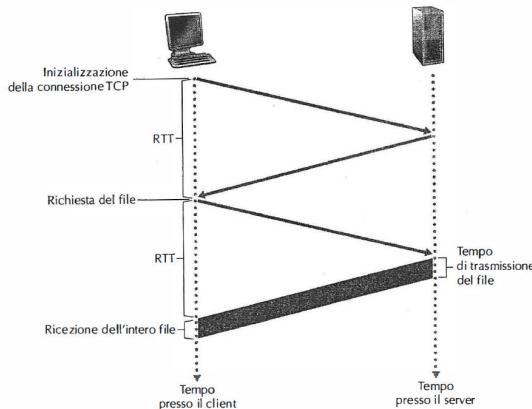


Figura 6.2: Calcolo approssimato del *round-trip-time*

Come notiamo facilmente in figura 6.2 la prima fase della comunicazione TCP è la fase di **handshake**, che permette di stabilire la connessione tra i due host; l'**handshake** TCP è detto *three-way handshake* e richiede, quindi, tre diverse fasi:

- Il primo scambio avviene da parte del client, che richiede di iniziare l'handshake.
- Il secondo scambio avviene da parte del server, che conferma l'inizio dell'handshake.
- Il terzo scambio avviene da parte del client, che conclude l'handshake, stabilendo la connessione.

Le prime due parti della stretta di mano richiedono un **RTT**. Dopo il loro completamento, il client invia un messaggio di richiesta HTTP combinato con la terza parte dell'handshake. Quando il messaggio di richiesta arriva al server, quest'ultimo inoltra il file HTML sulla connessione TCP. Quindi il costo totale è di 2 **RTT**.

**HTTP con connessioni persistenti** Le connessioni non persistenti presentano alcuni limiti: il primo è che per ogni oggetto occorre stabilire una nuova connessione. Per ciascuna di queste connessioni si deve allocare un buffer e mantenere le variabili TCP sia nel client sia nel server. In secondo luogo, ciascun oggetto riceve un ritardo di risposta di due **RTT**.

Nelle connessioni persistenti il server lascia la connessione TCP aperta dopo l'invio di una risposta, per cui le richieste e le risposte successive tra gli stessi client e server utilizzeranno la medesima connessione. Il server chiude la connessione quando essa rimane inattiva per un certo intervallo di tempo (configurabile).

## 2 Formato dei messaggi HTTP

Un'altra delle caratteristiche che studiamo del protocollo HTTP è il **formato dei messaggi**, quindi come sono suddivisi e strutturati. Il protocollo HTTP definisce due tipologie di messaggi: **messaggi di richiesta HTTP** e **messaggi di risposta HTTP**. La prima tipologia di messaggi ha una struttura generale del tipo

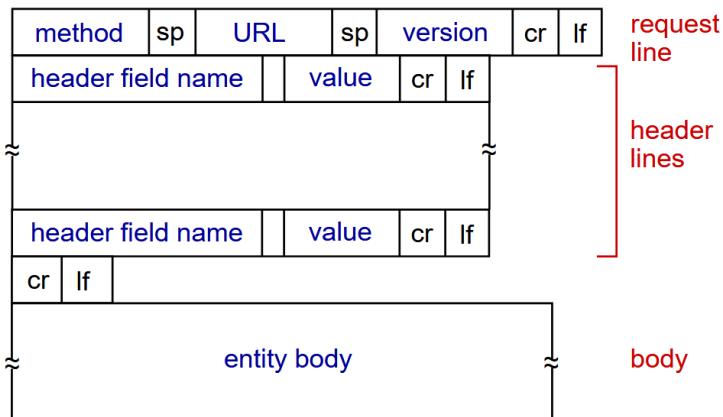


Figura 6.3: Formato generale di un messaggio di richiesta HTTP

Il metodo è costituito da caratteri ascii, così da essere leggibile da parte dell'utente. Inoltre, il messaggio è suddiviso in sezioni: **riga di richiesta**, **righe di intestazione**, **una riga vuota** e **corpo dell'entità**. Tutte queste sezioni sono suddivise da *caratteri di ritorno al carrello* e, internamente, da *caratteri di spazio*. La prima riga, quella di richiesta, presenta tre campi

- Il campo **metodo**, che specifica se la richiesta è di tipo: `post`, `get`, `head`, `put` o `delete`. Se una richiesta non comprende un trasferimento dati da un form al server è di tipo `get`, altrimenti, generalmente, è di tipo `post`. In particolare
  - Il metodo `put` permette agli utenti di inviare un oggetto a un percorso specifico su uno specifico web server.
  - Il metodo `head` serve agli sviluppatori per verificare la correttezza del codice prodotto.
- Il campo **URL**, che specifica l'indirizzo della risorsa che si cerca di accedere.
- Il campo **HTTP-version**, che specifica la versione del protocollo HTTP.

La righe di intestazione contengono invece contenuto invece diverse informazioni relative al pacchetto. Una di queste righe contiene il nome dell'host su cui è contenuta la risorsa (e.g `www.amazon.com`), la linea di intestazione `connect` specifica se la connessione è persistente o meno; in particolare: `connection: close` specifica che la connessione è di tipo non persistente. La riga di intestazione `user-agent` specifica il tipo di browser che sta effettuando la richiesta al server (e.g `Mozilla/5.0`). Queste informazioni sono solo alcune delle possibili righe di intestazioni possibili fornite dal protocollo HTTP. Oltre alle righe di intestazione troviamo anche un campo **body**; questo

campo risulta utile per, come già accennato, tutti i messaggi il cui metodo è **post** e che inviano al server dei dati prelevati da un form compilato dal client.

La seconda tipologia di messaggi HTTP sono i **messaggi di risposta HTTP**. Anche in questo caso possiamo dividere il messaggio in tre sezioni: **riga di stato iniziale**, sei **righe di intestazione** e il **corpo**.

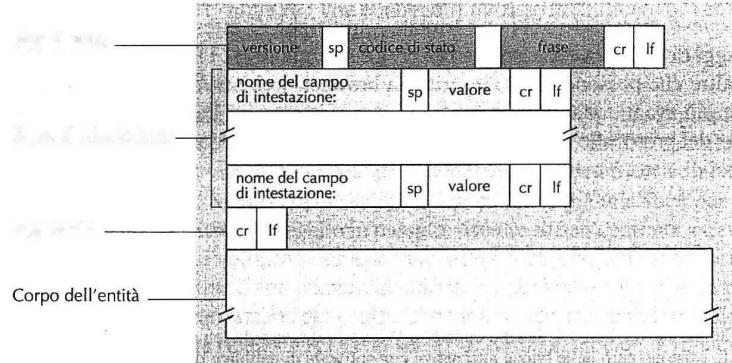


Figura 6.4: Struttura di un messaggio di risposta HTTP

Il corpo rappresenta il fulcro stesso del messaggio: contiene l'oggetto richiesto. La riga di stato presenta invece tre campi

- Versione del protocollo.
- Un codice di stato. Il codice di stato permette di identificare le varie tipologie di errore che si possono verificare nella richiesta di una risorsa.
- Un messaggio corrispettivo del codice di stato. Il messaggio specifica esattamente la tipologia di errore che si è verificata.

Anche in questo caso le righe di intestazione possono essere diverse e varie: possiamo trovare una riga **connection**, che specifica se la connessione è persistente o meno, o possiamo trovare una riga **date** che specifica data e ora di creazione e invio della risposta, o possiamo anche trovare una riga che specifica un'altra informazione come ad esempio **last-modified**, che specifica l'istante e la data in cui l'oggetto è stato creato o modificato per l'ultima volta. I codici di errore rappresentano una delle principali caratteristiche delle risposte HTTP; tra i più comuni troviamo

<i>Code</i>	<i>Error</i>	<i>Message</i>
200	OK	la richiesta ha avuto successo
301	Moved Permanently	l'oggetto richiesto è stato trasferito a nuovo URL
400	Bad Request	la richiesta non è stata compresa dal server
404	Not Found	la risorsa non è stata trovata dal server
505	HTTP Version Not Supported	la versione HTTP del client non è supportata dal server

Tabella 6.1: Tabella dei più comuni codici di stato nel protocollo HTTP

Nel caso del codice di errore **Moved Permanently** l'indirizzo dove è stata spostata la risorsa viene inserito all'interno del campo **location** dell'intestazione.

### 3 Interazione utente-server: i cookie

Una delle caratteristiche che rendono l'HTTP così potente è la possibilità di gestire un grandissimo numero di connessioni simultanee. Tuttavia, questa possibilità viene a scapito della mancanza di

**stato** e, quindi, dalla impossibilità del server di ricordare l’utente. A questo scopo HTTP introduce una funzionalità, in uso nella maggior parte dei server relativi ad applicazioni commerciali, chiamata **cookie**. I **cookie** permettono quindi di ottenere due vantaggi principali:

- Autenticare gli utenti.
- Limitare l’accesso a contenuti privati.
- Fornire contenuti in funzione della propria identità.

Supponiamo che un utente di nome susan acceda per la prima volta al server di [www.amazon.com](http://www.amazon.com). Supponiamo inoltre che in passato abbia visitato anche [www.ebay.com](http://www.ebay.com). Quando giunge la richiesta sul server di amazon, il sito crea un identificativo univoco e una voce nel proprio database, indirizzata dal numero identificativo. A questo punto il server risponde a Susan, includendo nella risposta HTTP l’intestazione **Set-cookie: id**, dove **id** corrisponde all’identificativo generato dal server. Quando il browser di susan riceve il messaggio di risposta, aggiunge una riga al file dei cookie includendo l’identificativo **id**. Mentre susan continua a navigare nel sito di amazon, ogni volta che richiede una pagina web, il suo browser consulta il suo file dei cookie, estrae il numero identificativo e lo pone nella richiesta HTTP in una riga di intestazione.

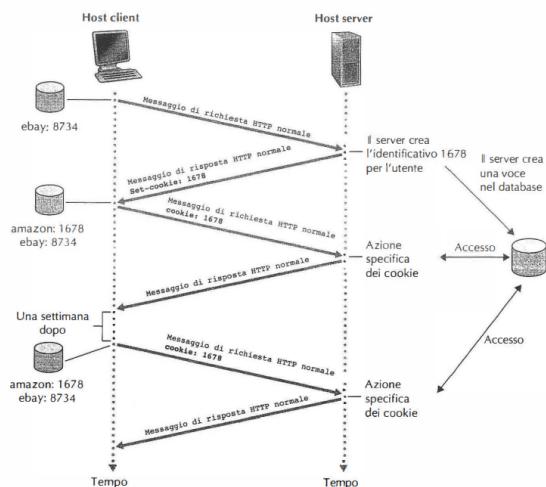


Figura 6.5: Memorizzazione dello stato utente con i cookie

Facendo in questo modo è possibile monitorare l’attività di un utente sul sito. Infatti, pur non sapendone il nome è possibile sapere quali pagine l’utente ha visitato e in che modo ha interagito con il nostro sito.

## 4 Web caching

Durante il corso di calcolatori elettronici era stata introdotta una memoria ad **accesso veloce**, detta **cache**, dove si trovano memorizzate tutte le informazioni (**cacheline**) utilizzate più di recente dalla CPU e, quindi, le informazioni che la CPU potrebbe utilizzare nel breve termine. Un’idea simile può essere trovata anche in rete, in particolare, è possibile sfruttare un dispositivo, chiamato **proxy server**, che permette di conservare copie di oggetti recentemente richiesti. Il browser di un’utente può essere configurato in modo che le richieste HTTP dell’utente vengano inoltrate direttamente al **proxy server**, che si occuperà di gestirle. In particolare:

- Il browser stabilisce una connessione TCP con il proxy server e invia una richiesta HTTP per l’oggetto specificato.
- Il proxy controlla la presenza di una copia dell’oggetto memorizzata localmente. Se l’oggetto viene rilevato, lo inoltra all’interno di messaggio di risposta.

- Se, invece, la cache non dispone dell’oggetto, apre una connessione TCP verso il server di origine. Poi il proxy invia al server una richiesta HTTP per l’oggetto. Una volta ricevuta tale richiesta, il serve di origine, include l’oggetto nella risposta HTTP.
- Quando il proxy riceve l’oggetto ne salva una copia nella propria cache interna e successivamente lo inoltra all’utente nel messaggio di risposta HTTP.

Il proxy è quindi contemporaneamente sia client che server: quando riceve richieste da parte del browser si comporta come un server, quando deve ricevere l’oggetto dal server si comporta come client. Generalmente è l’ISP stesso che compra e configura un proxy server all’interno della propria rete (e.g. unipi ha un proprio proxy).

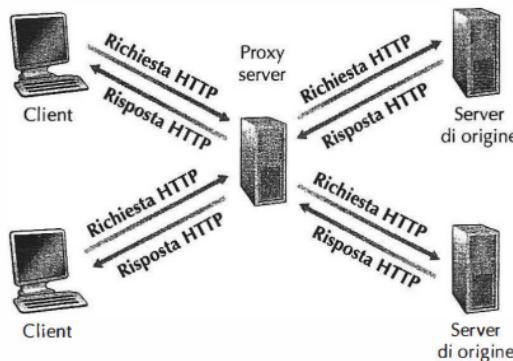


Figura 6.6: Client che richiedono oggetti attraverso un proxy server

Il **web caching** ha permesso di ridurre sostanzialmente i tempi di risposta alle richieste dei client. I proxy permettono anche di ridurre sostanzialmente il traffico sul collegamento di accesso ad internet, con il vantaggio di non dover aumentare l’ampiezza di banda e, quindi, riducendo i costi. Le problematiche del web caching sono, anche se in modo diverso, analoghe a quelle della cache di un calcolatore generale; in particolare, la copia di un oggetto ospitato nel proxy server potrebbe essere scaduta. Con scaduta intendiamo che la copia contenuta all’interno del server potrebbe essere stata modificata senza propagare tale aggiornamento anche al proxy server. Fortunatamente, HTTP presenta un meccanismo che permette alla cache di verificare la correttezza dei suoi dati; questo meccanismo prende il nome di **conditional GET**. Un messaggio di richiesta HTTP viene detto **GET condizionale** se rispetta due condizioni:

- Utilizza il metodo `get`.
- Include una riga di intestazione `if-modified-since`:

Supponiamo che un utente abbia richiesto un oggetto *x* non presente nel proxy server. Come abbiamo visto, il proxy server chiede l’oggetto al server originario e lo memorizza nella propria memoria, mandando una copia anche all’utente. Quando l’utente richiede la pagina, in un momento futuro, il server proxy verifica di avere la copia aggiornata inviando una richiesta del tipo

```

GET /fruit/kiwi.gif HTTP/1.1
Host: www.nomeHost.com
If-modified-since: Wed, 7 Sep 2011 09:23:24
  
```

la riga `If-modified-since` contiene come valore quello dell’intestazione `Last-Modified` inviata al server nella richiesta precedente. Nel caso in cui l’oggetto non sia stato modificato, il server restituirà un pacchetto con corpo vuoto e con un’intestazione `Not Modified`.

## 5 Versioni del protocollo HTTP

Le limitazioni della prima versione del protocollo HTTP (HTTP/1) riguardano le richieste multiple. Quando il server riceve più connessioni da parte dello stesso client sulla medesima connessione

TCP le accetta tutte e le gestisce secondo una logica **First-Come-First-Served**, se la prima richiesta è una richiesta voluminosa avviene un rallentamento delle altre richieste che comporta una congestione generale della rete (**Head of Line blocking**), rendendo questa versione molto inefficiente. Nella versione HTTP/2 è stata introdotta una maggiore flessibilità nella gestione dei pacchetti in uscita attraverso due variazioni:

- L'ordine di servizio non è più necessariamente **FCFS**.
- Si dividono gli oggetti in frame e ogni frame viene trattato singolarmente.

La prima variazione garantisce che non si possa verificare il problema di **HOLB**, però crea il problema opposto, se facessimo passare prima i pacchetti più piccoli, il pacchetto voluminoso potrebbe aspettare tempi lunghissimi prima di essere trasmesso (**starvation**). La seconda variazione garantisce il principio di **fairness**, quindi che ogni pacchetto abbia la stessa importanza, eliminando il problema creato con la rimozione della gestione **FCFS**. Nell'HTTP/3 aggiunge il meccanismo della sicurezza che, purtroppo, nell'HTTP/2 non era garantito; per farlo opera sul protocollo a livello di trasporto UDP. Questa differenza sostanziale, oltre a permettere di introdurre dei meccanismi di sicurezza, permette anche di incrementare l'efficienza, in termini di trasmissione, dei pacchetti.

# Capitolo 7

## Simple Mail Transfer Protocol

La posta elettronica è sicuramente una delle applicazioni che, ad oggi, ospita uno dei maggiori volumi di traffico in internet. Possiamo dire che rappresenta una delle più importanti **killer application** di internet. Ponendo una visione ad alto livello possiamo schematizzare la posta elettronica nel seguente modo

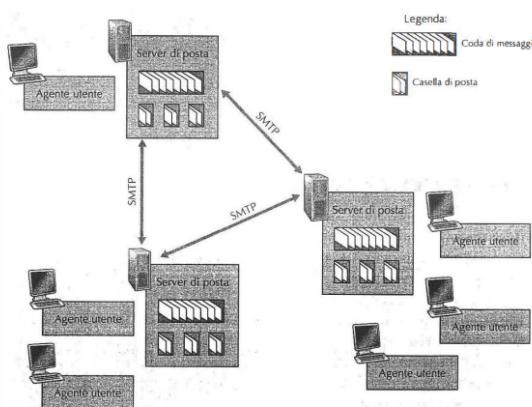


Figura 7.1: Visione ad alto livello del sistema di posta elettronica di internet

Da questa immagine possiamo notare la presenza di tre elementi fondamentali che si coordinano tra di loro: **user agent** (i fornitori del servizio mail), **server di posta** e **protocollo SMTP**. Supponiamo che un mittente, Alice, voglia inviare una mail a un destinatario, Bob. Quando Alice ha finito di comporre il messaggio, il suo **user agent** lo invia al server di posta, dove viene posto nella coda dei messaggi in uscita. Quando il messaggio arriva in testa alla coda di uscita, il server mail di Alice manderà il pacchetto al server mail di Bob e, nel momento in cui Bob vuole leggere il messaggio il suo **user agent** lo recupera dalla casella di posta nel suo mail server. I **mail server** costituiscono il centro di gravità teorico dell'infrastruttura del servizio di posta elettronica, gli scambi di messaggi avvengono tra mail server, mentre gli user agent comunicano solo con i rispettivi mail server. Così facendo è possibile anche avere una conferma del fatto che un messaggio sia stato mandato o meno: se il server di Alice non può consegnare il messaggio al server di Bob, lo pone in una coda di messaggi e cerca di trasferirla in un secondo momento. Se, dopo qualche giorno, ancora non è stato possibile mandare il messaggio, esso viene rimosso e Alice viene notificata della mancata consegna.

## 1 SMTP

Quando un server invia posta a un altro server agisce come client **SMTP**; quando, invece, la riceve, funziona come **server SMTP**. Il protocollo **SMTP**, o *Simple Mail Transfer Protocol*, è un protocollo di **livello applicativo** che utilizza il trasferimento **TCP** per l'invio di posta tra **mail server**. Nell'esempio precedente avevamo già dato l'ideal del funzionamento di tale protocollo; proviamo ora a formalizzarlo sempre con lo stesso esempio:

- Alice apre il suo user agent per la posta elettronica e scrive una mail, fornendo anche l'indirizzo di posta elettronica di Bob.
- Lo user agent di alice invia al messaggio al suo mail server, dove viene collocato in una **coda di messaggi**.
- Il **client SMTP**, che in questo caso è il serve mail di Alice, apre una connessione TCP con il **server SMTP**, che in questo caso è il server mail di Bob.
- Dopo un handshake TCP, il **client SMTP** invia il messaggio di Alice sulla connessione TCP.
- Il **server TCP** riceve il messaggio, che viene posizionato nella casella di bob.
- Bob, quando lo ritiene opportuno, apre il proprio user agent per leggere il messaggio.

Una delle particolarità del protocollo SMTP è che non fa uso di mail server intermedi (quindi vi è una comunicazione diretta tra i due server), anche se la connessione avviene tra due parti diverse del mondo. Inoltre, il tipo di connessione TCP utilizzata tra il client e il server è **persistente**: *nel caso ci fossero altri messaggi da inviare, il client invierebbe il messaggio sulla medesima connessione*.

Quindi, il primo passo viene fatto dal client, il quale stabilisce una connessione TCP sulla porta 25 verso il **mail server** destinatario; nel caso in cui il server sia disattivato il client tenterà un invio successivo dopo un certo intervallo di tempo. Una volta stabilita la connessione il client e il server fanno una sorta di handshake, e il client indica al server

- Indirizzo e-mail del mittente.
- Indirizzo e-mail del destinatario.

Dopo questo handshake, il client invia il messaggio di posta elettronica al server; se ci fossero altri messaggi da inviare il client utilizzerebbe la medesima connessione, altrimenti comunicherebbe al server di chiudere la connessione. Quindi, il formato di comunicazione tra client e server ha forma

```
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:<bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Ti piace il ketchup?
C: Che cosa ne pensi dei cetrioli?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```

Come si nota facilmente dalla trascrizione, il server invia risposte ad ogni comando, e ciascuna presenta un codice di risposta e qualche spiegazione. Il client inizia la comunicazione presentandosi specificando il proprio identificativo (**HELO crepes.fr**). Successivamente il server risponde confermando l'inizio della comunicazione e, a questo punto, il client può comunicare indirizzo mail del mittente e del destinatario. Una volta terminata questo handhsake dal client, attraverso il comando **DATA**, è possibile per quest'ultimo inviare il testo del messaggio di posta elettronica. Una volta ricevuta la comunicazione il client può chiudere la connessione con **QUIT**.

**Confronto con HTTP** L’idea alla base del protocollo **HTTP** e dell’**SMTP** è la medesima, garantire il trasferimento di file tra due dispositivi. La differenza principale sta, tuttavia, nel fatto che HTTP è un protocollo **pull**, quindi qualcuno carica degli oggetti su un web server e gli utenti utilizzano l’**HTTP** per *attirarli* (*pull*) a se; SMTP è un protocollo **push**: il mail server di invio *spinge* i file al mail server in ricezione. Un’altra differenza è che il protocollo SMTP deve comporre l’intero messaggio in ASCII a 7 bit (anche se il messaggio contiene immagini binarie o caratteri non a 7 bit), mentre HTTP non impone tali vincoli.

## 1.1 Formato dei messaggi di posta

Come abbiamo visto anche per l’**HTTP**, anche nel protocollo **SMTP** esiste un formato standard per i messaggi di posta. Questo standard (**RFC 5322**) specifica che un messaggio di posta elettronica debba essere costituito da una serie di righe di intestazione e il corpo del messaggio, separati da una riga senza alcun contenuto. Anche in questo caso abbiamo alcune parole chiave che sono obbligatorie e altre sono opzionali. Un esempio di intestazione SMTP potrebbe essere

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Alla ricerca del significato della vita
```

Dopo l’intestazione segue quindi una riga vuota e, solo successivamente, il corpo del messaggio.

## 2 Protocolli di accesso alla posta

Quando SMTP consegna il messaggio di Alice al mail server destinatario, questo lo colloca nella casella di posta di Bob. Il problema che ci poniamo ora è in che modo Bob sia in grado di accedere alla propria casella di posta elettronica per visualizzare i messaggi che gli sono arrivati. Infatti Bob non può utilizzare il protocollo SMTP, in quanto esso è un protocollo di tipo **push**, non di tipo **pull**. La soluzione a questa problematica può essere sintetizzata attraverso due approcci possibili:

- Utilizzare un particolare **protocollo di accesso alla posta** definito *ad hoc* per questo scopo.
- Utilizzare il **protocollo HTTP** per scaricare la casella di posta elettronica in locale.

I due principali protocolli di accesso alla posta sono **POP3** e **IMAP**. Il primo protocollo, cioè POP3, è un protocollo molto semplice, con delle funzionalità limitate, ed entra in azione quando lo user agent apre una connessione TCP verso il mail server. L’idea del protocollo è quella di definire tre diverse fasi: la prima, cioè quella di autorizzazione, prevede che il client si identifichi, la seconda, cioè quella di transazione, prevede che lo user agent recuperi i messaggi (potendo anche mandare messaggi per cancellazione, rimuovere marcatori di cancellazione e ottenere delle statistiche) e, infine, la terza fase, cioè quella di aggiornamento. **IMAP**, diversamente da **POP3**, permette di eseguire le operazioni che **POP3** esegue in locale, direttamente su un server. In particolare, un server **IMAP** associa a una cartella ogni messaggio arrivato al server; in altri termini, i messaggi in arrivo sono associati alla *inbox* del destinatario. Quest’ultimo può poi spostare il messaggio in una nuova cartella creata dall’utente, leggerlo, cancellarlo e così via. Il protocollo **IMAP** fornisce comandi per consentire agli utenti di creare cartelle e spostare i messaggi da una cartella ad un’altra, così come anche comandi che consentono agli utenti di fare ricerche nelle cartelle remote sulla base di criteri specifici.

L’ultimo approccio, cioè quello della posta web, si basa sull’utilizzo del protocollo HTTP. Lo user agent è quindi un semplice browser web e l’utente comunica con la propria casella di posta via HTTP. Quando un destinatario vuole accedere alla propria casella, il messaggio e-mail viene spedito dal server di posta al browser di Bob usando il protocollo HTTP. Quando Alice vuole inviare un messaggio di posta elettronica, quest’ultimo viene spedito dal browser web al server di posta su HTTP anziché su SMTP.

# Capitolo 8

## Domain Name System

L'identificazione è uno dei problemi che per primi si sono presentati agli albori di internet. Un qualsiasi dispositivo su internet, come **endpoint** e **router**, è identificato su internet da un indirizzo IP; un codice composto da 32 bit diviso in 4 gruppi da 8 bit ciascuno. Tuttavia, per un essere umano, ricordare un numero considerevole di sequenze numeriche è qualcosa di quantomeno complesso. Potrebbe essere quindi utile riuscire ad identificare un qualsiasi indirizzo e, quindi, una qualsiasi applicazione, attraverso un **nome simbolico**. Ad esempio, dato un indirizzo 67.21.1.5, associare un nome del tipo `www.pippo.com`. Ciò viene reso possibile dal protocollo **DNS** (o *Domain Name System*); possiamo immaginare questo protocollo come una funzione che associa un nome simbolico a un indirizzo IP

$$\text{DNS} : \text{Indirizzo IP} \rightarrow \text{Nome simbolico} \quad (1)$$

DNS è quindi composto da due elementi, un database distribuito implementato in una gerarchia di **DNS server** e un **protocollo a livello applicativo** che permette agli host di interrogare il database. I server DNS sono generalmente macchine che eseguono un software chiamato **BIND**, mentre il protocollo **DNS** è un protocollo che si basa su **UDP** e che lavora sulla porta 53.

### 1 Servizi offerti dal DNS

DNS viene utilizzato da protocolli a livello applicativo, come ad esempio **HTTP**, **SMTP** e **FTP**, per tradurre i nomi di host forniti dall'utente in indirizzi IP. Il protocollo SMTP offre, tra le altre cose, anche un servizio di **host aliasing**: nel caso in cui un host abbia un nome complicato da ricordare (che chiamiamo **hostname canonico**) è possibile referenziarlo con uno o più sinonimi (che chiamiamo **alias**) e il protocollo DNS si occuperà di tradurre l'**alias** nell'**hostname canonico**. Il DNS permette anche di distribuire il carico tra server replicati. I siti con molto traffico vengono infatti replicati su più server diversi, ciascuno dei quali con un indirizzo IP differente. Nel caso in cui si abbiano dei server replicati, allo stesso hostname canonico andrà quindi associato un'insieme di indirizzi IP e il DNS a rotazione smisterà le richieste di rete verso uno di questi server, andando a dividere il traffico tra i vari server in modo equo. L'idea del funzionamento del protocollo è la seguente:

- Il client DNS, che coincide con la macchina dell'utente, estraе dall'URL il nome dell'host.
- Il client DNS invia una **query** contenente l'hostname al server DNS.
- Il client DNS attende una risposta da parte del server, la quale includerà l'indirizzo IP del server.
- Una volta ricevuto l'indirizzo IP dal DNS, il browser può dare inizio a una connessione TCP verso il processo server HTTP collegato sulla porta 80.

Proviamo a fare un esempio per far comprendere meglio il funzionamento del DNS. Supponiamo che l'host `cis.poly.edu` voglia l'indirizzo IP di `gaia.cs.umass.edu`. Supponiamo, inoltre, che il DNS server locale per `cis.poly.edu` sia `dns.poly.edu`, mentre un server autoritativo per `gaia.cs.umass.edu` sia `dns.umass.edu`. Per prima cosa l'host `cis.poly.edu` invia una richiesta per l'indirizzo ip di `gaia.cs.umass.edu` al proprio DNS server locale. Il DNS server locale invia a questo punto una richiesta, che contiene l'indirizzo da tradurre, a uno dei **root server** vicini. Il **root server** prende nota del suffisso `edu` e restituisce al server locale un elenco di indirizzo IP per i **TLD server** responsabili di `edu`. Il server locale invia quindi il messaggio di richiesta a uno di questi TLD che, a loro volta, prenderanno nota del suffisso `umass.edu` e risponderanno con l'indirizzo IP del server autoritativo `dns.umass.edu`. Infine, il DNS server locale rimanda il messaggio di richiesta direttamente a `dns.umass.edu`, che risponde con l'indirizzo IP di `gaia.cs.umass.edu`.

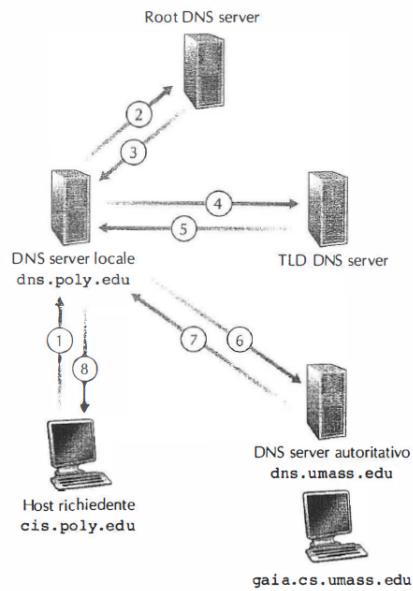


Figura 8.1: Interazione tra server DNS

Il primo problema che però non abbiamo considerato è che non è sempre detto che il TLD server conosca il server autoritativo per quel dato nome, ma non è in generale così. Può capitare anche una situazione in cui un TLD server conosca un server DNS intermedio, il quale a sua volta conosce un server autoritativo per quell'hostname. Nell'esempio che abbiamo fatto potrebbe infatti succedere che `dns.umass.edu` non conosca l'indirizzo IP di `gaia.cs.umass.edu`, ma conosca l'indirizzo del server DNS `dns.cs.umass.edu`. Inoltre, se da una parte si ottiene un grandissimo vantaggio in termini di usabilità e di semplicità nella ricerca delle pagine web, l'aggiunta del DNS introduce un ritardo che talvolta può essere anche sostanziale per le applicazioni internet che lo utilizzano. Fortunatamente però esiste un meccanismo che permette di ridurre questo ritardo, detto **DNS caching**. L'idea alla base di questo meccanismo è molto semplice. In una concatenazione di richieste, il DNS server che riceve una risposta DNS può mettere in cache le informazioni contenute e riutilizzarle nel caso vengano richieste nuovamente, senza dover inoltrare una nuova richiesta verso il **root server**.

**Approccio ricorsivo** L'approccio introdotto poco sopra è detto **approccio iterativo** e non è l'unico approccio che possiamo utilizzare. Esiste infatti un altro approccio, detto **approccio ricorsivo**, che si basa sull'idea che non sia il server DNS locale a fare tutto il lavoro di scambio di messaggi con i vari server DNS, ma ricorsivamente la richiesta scenda di livello per poi ritornare al server DNS locale. In particolare, il server DNS locale inoltra la richiesta al **root server DNS**. Quest'ultimo, a sua volta, inoltrerà la richiesta al **TLD server DNS** che codifica quel particolare dominio. Il **TLD server DNS** dovrà quindi instradare il pacchetto al serverd DNS autoritativo

che codifica l'indirizzo cercato. Una volta che il server DNS autoritativo risponde, il pacchetto risalirà fino al root server facendo il percorso all'indietro, per poi, infine, tornare al server DNS locale.

## 2 Struttura del DNS

L'idea alla base della struttura del DNS è quella di un **sistema distribuito**. In particolare, DNS è costituito da una serie di server DNS che, tutti insieme, costituiscono un grande **database distribuito** e da un protocollo a livello applicativo che permette agli host di interrogare questo database. I vantaggi di adottare una struttura di questo tipo sono molteplici; in particolare:

- **Limitazione dei guasti:** se si prevedesse un unico server DNS si verrebbe a creare quello che, in gergo tecnico, viene detto **single point of failure**. Un punto all'interno della rete il cui guasto causerebbe un collasso sistematico dell'intera infrastruttura.
- **Volume di traffico:** distribuendo il database tra diversi server è possibile limitare enormemente il traffico in entrata sui singoli server, evitando così la perdita di pacchetti e di informazione.
- **Riduzione delle distanze:** con un singolo server DNS non si potrebbe garantire una vicinanza minima con tutti i client, con conseguente diminuzione delle prestazioni per i pacchetti che arrivano da più lontano.
- **Manutenzione:** se si prevedesse di centralizzare il database si verrebbe a creare il problema della necessità di aggiornamenti costanti per tenere conto dei nuovi host.

Dal punto di vista dell'utente DNS è una scatola nera che ha come funzione principale quella di fare delle traduzioni in modo diretto e veloce. Dal punto di vista interno invece, la struttura è gerarchica e organizzata su livelli; esistono diverse tipologie di DNS e ognuna di questa si occupa di tradurre classi di **hostname** specifici. I **root server** sono in cima alla gerarchia dei server DNS, questi server hanno il compito di smistare le richieste degli utenti ad altri server DNS che si occupano della traduzione di classi di indirizzi; questi server sono detti **top-level domain server**. I **top-level domain server** si occupano della gestione dei domini di primo livello, come: **org, com, edu, etc.** Quando una richiesta arriva da un **root server** a un **TLD server**, il **TLD server** si occuperà di instradare la richiesta verso il server DNS che contiene l'effettiva traduzione dell'indirizzo che si sta cercando; questi server sono chiamati **DNS server autoritativi**. I **DNS server autoritativi** si trovano in fondo alla gerarchia e si occupano della traduzione effettiva dei **nomi simbolici in indirizzi IP**. Esiste anche una tipologia di server DNS che non appartengono a questa gerarchia, detti **DNS server locali**, i quali sono implementati direttamente dagli ISP locali e permettono la traduzione dei nomi delle pagine internet.

## 3 Messaggi DNS

I server che implementano il database distribuiti di DNS memorizzano dei **record di risorsa** (detti anche **resource record**), tra cui quelli che forniscono le corrispondenze tra nomi ed indirizzi. In generale, la struttura di un **RR** contiene i seguenti campi:

(**Name**, **Value**, **Type**, **TTL**)

Il **TTL** identifica il **time-to-live**, quindi il tempo residuo di vita di un record, cioè determina quando una risorsa vada rimossa dalla cache del server DNS. I campi **name** e **Value** dipendono dal valore del campo **Type**; in particolare:

- Se **Type = A** allora **Name** è il nome dell'host e **Value** è il suo indirizzo IP. Pertanto questo tipo di record fornisce la corrispondenza tra **hostname standard** e **indirizzo IP**.
- Se **Type = NS** allora **Name** corrisponde a un dominio e **Value** è l'hostname del server DNS autoritativo che sa come ottenere gli indirizzi IP degli host nel dominio.

- Se **Type** = CNAME allora **Value** rappresenta il nome canonico dell'host per il sinonimo **Name**.
- Se **Type** = MX, allora **Value** è il nome canonico di un mail server che ha il sinonimo **Name**.

Il secondo tipo di record viene utilizzato per instradare le richieste DNS successive alla prima. Il terzo fornisce agli host richiedenti il nome simbolico del relativo hostname. Un server DNS autoritativo per un certo set  $\mathcal{X}$  di hostname contiene, ovviamente, un record di tipo A per ogni hostname contenuto in  $\mathcal{X}$ . Se un server non è autoritativo per un certo hostname, allora conterrà un record di tipo NS per il dominio dell'hostname e include anche un record di tipo A, che fornisce l'indirizzo IP del DNS server nel campo **Value** del record NS.

La struttura del messaggio DNS è costituita da un'intestazione da 12 byte che contengono un certo numero di campi. Tra questi possiamo trovare un identificatore a 16 bit che identifica la richiesta, un campo flag costituito da diversi bit: il primo bit indica se il messaggio è di richiesta o di risposta, il secondo bit viene impostato quando un client desidera che il DNS server effettui ricorsione quando non dispone di record, il terzo bit determina se il DNS server a cui è stata richiesta la ricorsione, per l'appunto, la supporta e, infine, l'ultimo flag determina se il server che risponde è **autoritative** per quel dominio. Inoltre, oltre all'intestazione troviamo anche una **sezione delle domande**, la quale contiene informazioni sulle richieste che stanno per essere effettuate. Insieme alla sezione delle domande troviamo anche la sezione delle risposte, la quale contiene i record di risorsa relativi al nome originariamente richiesto. Le ultime due sezioni sono la **sezione autoritaria** e la **sezione aggiuntiva**.

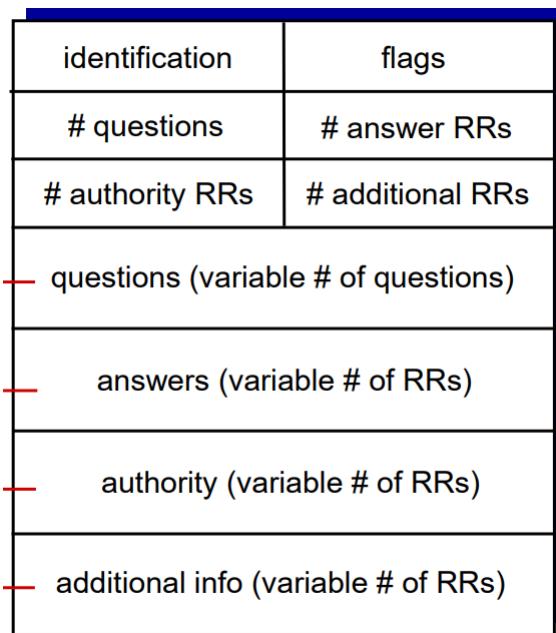


Figura 8.2: Struttura di un record DNS

Uno dei passaggi fondamentali per un'azienda è quello di registrare il proprio record DNS. Supponiamo che una start-up, appena fondata, voglia registrare il proprio dominio **www.networkutopia.com** presso un ente di registrazione (**register**). Questo **register** è un'azienda che verifica l'unicità del nome di dominio e lo inserisce nel database **DNS**, ovviamente in cambio di una somma di denaro. La registrazione richiede che l'azienda dia al **register** i **nomi** e gli **indirizzi IP** dei **DNS** server autoritativi, sia primario che secondario. Una volta passati questi dati l'ente si accerterà che venga inserito un record di tipo NS e uno di tipo A nei **TLD** server relativi al suffisso com.

# Capitolo 9

## Applicazioni peer-to-peer

L'altra categoria di applicazioni che dobbiamo considerare sono le applicazioni cosiddette **peer-to-peer**. Ricordiamo che in questo modello il principio di distinzione dei ruoli del **client-server** decade a favore di un modello in cui ogni **host** è - di fatto - sia client che server allo stesso tempo e rientra sotto al più generale nome di **peer**. In questo modello gli host sono tutti alla pari e, oltre a ciò, non vi è necessità di ricchezza di risorse del singolo peer, così come non è necessario che un peer sia sempre e comunque acceso. Esistono due tipologie di applicazioni per cui si rende utile l'utilizzo del modello **peer-to-peer**:

- **File Sharing.**
- **Instant Messaging**

Tuttavia, entrambi questi casi presentano delle problematiche non indifferenti. Nel primo caso, quello del trasferimento di file, è importante capire *chi ha cosa*; quando un host A richiede una certa risorsa è necessario che sappia chi possiede quella risorsa prima di iniziare il trasferimento e in un sistema decentralizzato, come quello **peer-to-peer**, è nativamente difficile come operazione. Nel secondo caso, quello della messaggistica, non è sempre detto che un utente destinatario di una comunicazione sia collegato sempre dallo stesso dispositivo e, quindi, l'host mittente deve sapere a quale dei dispositivi il destinatario è connesso.

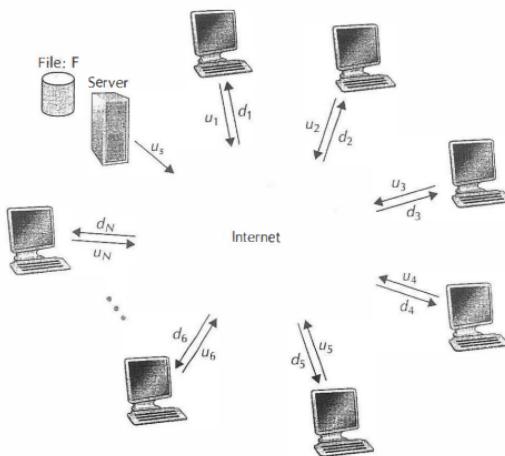


Figura 9.1: Distribuzione dei file **peer-to-peer**

La soluzione a queste problematiche è quella di introdurre un indice dei contenuti. L'indice dei contenuti è un database costituito da coppie (**key**, **values**) dove **key** è la risorsa e **value** è l'indirizzo IP; ad esempio

(Led Zeppelin IV, 203.17.123.38)

I **peer** possono sia **interrogare** il database per conoscere la locazione di una risorsa a cui sono interessati, sia **inserire** delle coppie nel caso vogliano rendere disponibile una risorsa al resto dei **peer**. La gestione di questo database può essere pensata in modo centralizzato, quindi con un solo database posto su un server che riceve le richieste dei vari peer e, una volta trovata la corrispondenza chiave valore, restituisce una risposta. Tuttavia questa soluzione presenta dei limiti non indifferenti:

- Il primo problema è che così facendo si crea un **single-point-of-failure**. Se per qualche motivo il server andasse offline il sistema sarebbe inutilizzabile.
- Il secondo problema deriva dall'enorme traffico di richieste che transiterebbe sul server, rischiando di avere un **packet-loss** considerevole (**bottleneck**).

Inoltre, sebbene l'applicazione sia **peer-to-peer**, possiamo immaginare che la transazione con l'**indice dei contenuti** sia di tipo **client-server** dove: il client è il **peer** e l'**indice dei contenuti** è il server.

Nell'approccio decentralizzato, detto **query flooding**, ogni **peer** che mette a disposizione dei contenuti genera una parte dell'indice dei contenuti. Ogni dispositivo diventa quindi gestore della propria parte dell'indice dei contenuti, cioè la parte dell'indice relativa ai contenuti che egli rende disponibile agli altri. Questo approccio ha il suo punto di forza nella facilità di caricamento di contenuti nell'indice nei contenuti, ma ha il suo punto di debolezza nella ricerca dei contenuti. Se un **peer** cerca contenuti deve iniziare a chiedere agli altri **peer** vicini, quindi quelli collegati da una connessione TCP, se hanno a disposizione quella particolare risorsa. A seguito di questa richiesta si possono verificare due situazioni:

- I vicini hanno la risorsa.
- I vicini non hanno la risorsa.

Nel secondo caso, quindi se la risorsa non viene trovata dai vicini, ogni **peer** vicino al richiedente propaga la richiesta per la risorsa ai propri vicini e così via fintanto che qualcuno non ha la risorsa; il termine **query flooding** si riferisce proprio a questa inondazione di richieste che avviene nella rete. Questo invio di query si traduce però in un **overhead** importante sull'infrastruttura e, soprattutto, il tempo di attesa per una risorsa potrebbe essere molto alto.

Per ridurre i tempi di attesa è quindi necessario utilizzare un approccio che sia ibrido tra modello client server e modello **peer-to-peer**, pur restando maggiormente vicino al secondo. Supponiamo di introdurre dei nodi speciali, detti **supernodi**, il cui scopo è quello di funzionare come indice dei contenuti per un set  $\mathcal{X}$  di nodi. Quando un nuovo nodo entra nel sistema si deve, prima di tutto, registrare presso il supernodo dichiarando quali risorse mette a disposizione degli altri. Il supernodo, preso atto di questi contenuti, li memorizza all'interno del proprio indice dei contenuti, andando quindi a codificare una porzione dell'indice dei contenuti del sistema. Quando un dispositivo cerca una risorsa interroga il supernodo presso cui si è registrato; il quale a sua volta controlla il suo indice dei contenuti:

- Se si trova un riscontro nell'indice dei contenuti allora è sufficiente segnalare l'indirizzo ip a cui trovarla al peer che l'ha richiesta.
- Se non si trova un riscontro diretto nell'indice dei contenuti del supernodo allora, quest'ultimo, inizierà una comunicazione con gli altri supernodi chiedendo chi ha la risorsa. Se uno dei supernodi trova un riscontro lo invierà l'indirizzo IP al supernodo mittente che, a sua volta, lo invierà al peer che lo ha richiesto.

Quindi, i supernodi formano tra di loro una vera e propria rete di comunicazione. Inoltre, sempre questi supernodi, rivestono un'importanza maggiore rispetto agli altri e, quindi, devono anche rispettare alcune caratteristiche:

- Maggiore disponibilità, sia in termini di risorse che in termini di tempo.
- Devono essere facilmente raggiungibili da parte di altri host.
- Devono garantire una capacità di trasmissione che sia sufficientemente alta.

## 1 Trasferimento file

Potrebbe essere quindi utile chiedersi quale tra il paradigma **Client-Server** e il paradigma **Peer-To-Peer** sia migliore dal punto di vista dell'efficienza. In generale non è possibile fare una stima valevole in generale, poiché questa misura dipende dalla metrica di riferimento; nel nostro caso utilizziamo come metrica il **tempo di distribuzione** per un file. Idealmente ci piacerebbe, essendo il tempo di distribuzione una sorta di **ritardo**, minimizzare questa grandezza, cioè trovare una maggiorazione che sia sufficientemente piccola. Ipotizziamo di voler distribuire un file di  $F$  bit, a  $N$  client, con una banda di upload del server pari a  $\mu_s$ . Idealmente, per riuscire a distribuire il file a un singolo client, non facendo alcune assunzioni sulla capacità di download del client, ci vorrebbe

$$\text{Tempo distribuzione singolo client} = T_D = \frac{\text{Dimensione del file}}{\text{Capacità upload server}} = \frac{F}{\mu_s}$$

Considerato che nel peer si trova un numero di client pari a  $N$  allora il tempo di distribuzione sarà pari al tempo di distribuzione per un singolo client moltiplicato per il numero di client; in termini matematici:

$$\text{Tempo distribuzione file} = N \cdot T_D = N \cdot \frac{L}{\mu_s}$$

Ipotizziamo ora che un client possa scaricare a una velocità di download pari a  $d_i$ , allora il tempo necessario per scaricare il file sarà pari al rapporto tra la dimensione del file e la capacità di download del client. Ipotizzando di avere  $N$  client possiamo definire  $d_{\min}$  come il minimo delle velocità di download dei client e possiamo definire il rapporto tra  $L$  e  $d_{\min}$  come la velocità a cui il client più lento riesce a spedire:

$$\text{Rate download client più lento} = \frac{L}{d_{\min}}, \quad d_{\min} = \min\{d_1, \dots, d_n\}$$

Ovviamente, anche se un server riuscisse a distribuire il file a un rate  $\mu_s$ , non potrebbe comunque spedire ad un tasso maggiore rispetto a quello di download del client. Per tale motivo, possiamo dire che il tempo di distribuzione per un particolare file, nel modello client server, corrisponde al massimo tra il tempo di distribuzione per  $N$  client e il tempo di dowload per il client più lento

$$\text{Tempo di distribuzione} = D_{es} = \max \left\{ N \frac{F}{e_s}, \frac{F}{d_{\min}} \right\} \quad (1)$$

Il tempo di distribuzione aumenta quindi con il numero di peer in modo proporzionale. Possiamo fare una stessa considerazione anche per quanto riguarda il **peer-to-peer**. Infatti all'istante iniziale possiamo dire che solamente il server possiede la risorsa, mentre i peer, che ancora non hanno scaricato nulla, non possono inviare dati ad altri peer. Man mano però che i peer iniziano ad accumulare blocchi di informazione (ovviamente cercheranno di scaricare blocchi di informazione diversi da quegli degli altri) potranno iniziare a condividere anche loro i vari blocchi agli altri client. Possiamo quindi dire che il tempo di distribuzione di un file per un singolo client diventa

$$\text{Tempo di distribuzione singolo client} = \frac{F}{\mu_s + \sum_{i=1}^n \mu_i}$$

Supponendo di avere  $N$  client all'interno del sistema possiamo concludere che il tempo di distribuzione per il file sarà dato dal prodotto del tempo di distribuzione del client per  $N$  client. Il tempo di distribuzione totale per il modello peer-to-peer diventa quindi pari al massimo tra il tempo di distribuzione client-server e quello che abbiamo appena calcolato:

$$\text{Tempo di distribuzione} = D_{es} = \max \left\{ N \frac{F}{e_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum_{i=1}^n u_i} \right\} \quad (2)$$

Andando a fare delle considerazioni di natura analitica possiamo giungere alla conclusione che il peer-to-peer sia un modello migliore nel caso in cui si cerchi di minimizzare il **tempo di distribuzione**. D'altra parte però non è possibile fare la stessa considerazione se si varia la metrica da di riferimento. Possiamo osservare anche graficamente che il modello peer-to-peer, per il tempo di distribuzione, è migliore. Nell'architettura client-server il tempo di distribuzione cresce linearmente all'aumentare del numero di peer, mentre quello peer-to-peer segue un andamento asintotico:

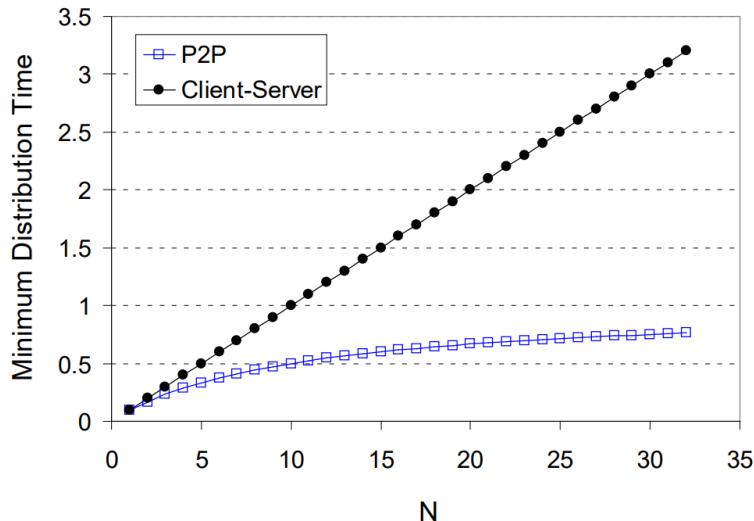


Figura 9.2: Differenze di prestazioni tra modello client-server e peer-to-peer

## 2 Protocollo BitTorrent

Un tipo di protocollo di distribuzione **peer-to-peer** è il protocollo **BitTorrent**. In questo protocollo l'insieme di tutti i peer che partecipano alla distribuzione un particolare file prende il nome di **torrent**; per via della loro natura, i torrent, sono entità dinamiche che variano a seconda di come variano i peer al loro interno. I peer in un **torrent** scaricano unità di informazione, dette **chunk**, con una dimensione tipica di 256 kbyte. Un peer che entra in un torrent non avrà inizialmente alcun chunk di dati, una volta che però inizia ad accumularli inizierà a sua volta a inviarli ad altri **peer**. Una volta che un peer ha ottenuto tutti i chunk della risorsa può fare due azioni principali:

- Attendere e continuare a inviare chunk.
- Disattivare la comunicazione, smettendo di inviare chunk.

Il modello alla base del protocollo **BitTorrent** è quello ibrido: ogni torrent ha un nodo di infrastruttura presso cui ogni peer che intende partecipare al torrent si deve registrare, detto **tracker**; possiamo anche dire che ogni tracker è in grado di gestire diversi torrent contemporaneamente. Il **tracker** ha anche un'altra funzionalità, periodicamente ogni **peer** deve segnalare al **tracker** che è ancora connesso al torrente, così da poter tenere una lista di **peer attivi** sempre aggiornata. Nel momento in cui un peer vuole ottenere un file deve prima di tutto trovare il torrente che si occupa di scambiare questo file; per farlo deve comunicare con un **torrent server** specificando il titolo della risorsa in questione. Una volta ricevuto il file .torrent, che tra le altre cose contiene l'indirizzo IP del tracker server del torrente, il peer dovrà registrarsi presso tale tracker; supponiamo che questo peer sia chiamato **Alice**. Il tracker invierà ad Alice la lista dei peer attivi per quel torrent. A questo punto Alice proverà a iniziare una connessione **TCP** con un sottoinsieme dei **peer** presenti in questa lista (nota bene, la lista potrebbe non essere aggiornata, poiché il torrent comunica gli aggiornamenti al tracker a intervalli temporali).

Tutti i peer con cui Alice riesce ad aprire la connessione TCP sono detti **vicini** di Alice e tra questi peer Alice dovrà decidere sia da chi scaricare i chunk e sia a chi inviarli. Per quanto riguarda

la prima scelta si utilizza il protocollo **rearest-chunk first**: si determina tra i vari che gli mancano chunk quelli che sono più rari (cioè quelli con il minore numero di copie) e li richiede per primi. In questo modo permette di rendere più o meno uguali il numero di copie di ciascun chunk del torrent. La scelta di una politica di questo tipo ha due motivazioni principali:

- **Sopravvivenza del file.** Se un chunk fosse presente in una sola copia e il client che lo possiede si disconettesse, quel chunk andrebbe perso e nessun client sarebbe in grado di scaricarlo.
- **Velocità di scaricamento.** Più i pezzi rari vengono diffusi velocemente, più i peer avranno qualcosa da scambiarsi tra loro.

Per determinare, invece, a quali peer inviare i chunk si utilizza una tecnica nota come **tit-for-tat**. Alice invierà i chunk ai primi quattro vicini che gli spediscono a loro volta chunk alla velocità più alta. Questi quattro vicini sono detti **unchocked** e il loro aggiornamento avviene ogni, circa, 10 secondi. Inoltre, a intervalli di 30 secondi, Alice sceglie anche un peer casuale tra i suoi vicini a cui non sta inviando dati, noto come **optimistically unchoked**, il quale *potrebbe* diventare uno dei quattro **unchocked**. In questo modo è possibile per Alice variare i propri peer **unchocked**, potendo includere o meno il peer **optimistically unchoked** a sfavore di uno dei 4 **unchocked**.

**L'innovazione di BitTorrent** L'innovazione principale di BitTorrent risiede proprio in questo meccanismo di **incentivazione alla collaborazione**. A differenza dei protocolli precedenti, BitTorrent risolve il problema del **free-riding**: se un utente non carica dati (non collabora), la sua velocità di download verrà drasticamente limitata dagli altri peer. In questo modo, l'interesse egoistico del singolo nel voler scaricare velocemente lo costringe a contribuire attivamente alla salute e alla velocità dell'intera rete.

# Capitolo 10

## Trasmissione multimediale

Una delle applicazioni principali che si trovano a livello applicativo è lo streaming video. Con streaming video intendiamo la trasmissione di un film, di un video o di una live. Il video, in generale, è costituito da una sequenza di frame che si susseguono per tutta la durata del video a frequenze (**trasmission-rate**), idealmente, costanti. Ogni **frame** è a sua volta costituito da una matrice di **pixel**, dove, ogni pixel è costituito da due cifre esadecimali di cui 6 bit rappresentano il colore in scala **RGB**. La trasmissione di file multimediali pone due diverse problematiche:

- **Scalabilità.** Queste applicazioni hanno un numero di utenti molto ampio e, di conseguenza, un volume di traffico considerevole.
- Eterogeneità dei dispositivi. Esistono moltissimi dispositivi diversi che permettono di scaricare dei file multimediali e, quindi, anche diverse velocità di ricezione.

### 1 Streaming HTTP

In questo modello il contenuto multimediale si trova su un server **HTTP** ad un **URL** specifico. Quando l'utente vuole vedere il video deve, prima di tutto, stabilire una connessione **TCP** con il server e richiedere la risorsa. Il server a questo punto inizia a mandare parti del contenuto alla massima velocità a cui è consentito farlo. Man mano che i dati arrivano all'utente, vengono memorizzati in un buffer interno. A questo punto il client colleziona pacchetti e attende che il quantitativo dei pacchetti superi un **limite minimo di pacchetti**. Una volta superato questo limite il client inizia a mostrare il video sullo schermo dell'utente; in particolare, il client:

- Preleva un pacchetto dal buffer.
- Decomprime il pacchetto.
- Lo mostra sullo schermo dell'utente.

Il problema della trasmissione deriva principalmente dalla qualità del video: qualità più alte necessitano di banda maggiore, che non sempre può essere garantita. Ad oggi esiste un meccanismo, noto come **DASH**, che permette di adattare la **quantità dei bit** alla **banda disponibile**. L'idea del meccanismo è relativamente semplice, il server memorizza diverse versioni dello stesso chunk del video compresso a qualità differenti (quindi con diversi algoritmi di compressione). Il client misura costantemente il **throughput** disponibile e, in funzione della misurazione, consulta un manifesto. All'interno di questo manifesto si associano a vari intervalli di **throughput** l'URL del chunk che meglio si associa a quella banda passante. A questo punto il client inizierà a richiedere chunk di video alla qualità che corrisponde alla migliore per la sua banda passante. In questo modo l'utente è in grado di fruire il video senza interruzioni, anche se a una qualità minore.

## 2 Content Distribution Network

Il problema della distribuzione dei file deriva dal fatto che, ad oggi, tantissime aziende di streaming trasmettono enormi quantitativi di video su internet. Riuscire a mandare in streaming una tale quantità di dati a utenti sparsi in tutto il mondo fornendo riproduzione continua e alta interattività diventa una grandissima sfida. L'idea più immediata per risolvere questo problema potrebbe essere quella di costruire enormi datacenter, memorizzare tutti i video in essi e mandarli in streaming direttamente; tuttavia un tale approccio presenta tre problemi:

- **Distanza tra Client e Server.** La posizione del datacenter potrebbe influenzare molto la capacità di un utente di accedere ai contenuti in streaming. Se infatti il Client fosse molto lontano dal server, i pacchetti diretti verso quest'ultimo dovranno attraversare un cammino passando per moltissimi ISP. Se uno dei collegamenti ha un throughput minore del tasso di consumo del video, anche il throughput totale end-to-end lo sarà, causando all'utente continui e fastidiosi fermi immagine.
- **Eterogeneità nel cosmo delle risorse.** I video non vengono consumati in modo omogeneo, è infatti possibile che alcuni video vengano richiesti più di altri. Se un video popolare viene invitato molto spesso, magari sullo stesso collegamento, si corre il rischio di consumare, inutilmente, della preziosa banda.
- **Single Point of Failure.** Se il datacenter o il collegamento tra esso e internet smettono di funzionare, tutte le risorse messe a disposizione da quel datacenter cessano di esistere su internet, causando non pochi disagi agli spettatori.

per superare queste problematiche i maggiori distributori di contenuti video utilizzano delle **Content Distribution Network**. Una CDN gestisce server distribuiti in molti posti diversi, memorizza copie dei video e di altri contenuti web nei server e cerca di dirigere le richieste degli utenti al punto della CDN in grado di offrire il servizio migliore. Le CDN possono essere sia **private**, come quella di google che distribuisce i contenuti di youtube, sia **di terze parti**, che invece fornisce contenuti per conto di diversi fornitori. Quindi, una volta posizionati i cluster, la CDN replica i contenuti su di essi. I video raramente richiesti o comunque che sono popolari solo in alcuni paesi non verranno ovviamente copiati su ogni CDN, ma solo su un numero ristretto. Inoltre, molte CDN non spingono i loro video verso i cluster, ma utilizzano una strategia simile a quella del web caching: se un client richiede una video a un cluster che non lo ha memorizzato, il cluster recupera il video da un archivio centrale o da un cluster che lo ha e ne memorizza una copia locale mentre lo manda in streaming al client.

## Parte III

# Livello di collegamento

# Capitolo 11

## Servizi del livello di collegamento

Riprendiamo ora lo studio della rete partendo dal basso, andando a studiare come due **endpoint** si scambiano informazioni all'interno della rete. Supponiamo di avere due **endpoint** che vogliono comunicare tra di loro e supponiamo che tra essi esista una connessione diretta costituita da un mezzo fisico. Schematizziamo l'esempio come in figura

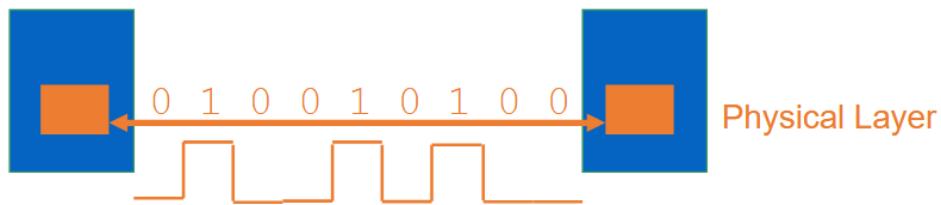


Figura 11.1: Due **endpoint** in comunicazione attraverso un collegamento diretto

Quello che l'host mittente vuole comunicare all'host destinatario è una sequenza di bit, che però non può essere trasmessa in quanto tale sul mezzo fisico. Affinché questi due host possano comunicare è necessario che avvenga una fase di **codifica** del messaggio: questa fase di codifica deve trasformare un messaggio *logico*, costituito da una sequenza di bit, in un segnale **fisico** che sia compatibile con il mezzo di trasmissione scelto; ad esempio

- Impulso di corrente nel caso di fili costituti da materiali conduttori.
- Impulso luminoso nel caso della fibra ottica.

A tale scopo viene deputato in particolare dispositivo, detto **trasmettitore**, che si occupa di codificare il messaggio e, solo successivamente, instradarlo sul mezzo fisico. Dualmente, dal lato del ricevitore, dovrà trovarsi un **ricevitore**, il cui compito è **decodificare** il segnale fisico in un messaggio logico, che sia idealmente **uguale** a quello che è stato inviato dal mittente. Tutte queste fasi riguardano il livello fisico, il quale implementa solo le funzioni di **codifica** e **decodifica** delle informazioni attraverso apparati di **trasmissione** e **ricezione**.

Tuttavia i mezzi fisici non sono ideali, esistono infatti moltissimi fenomeni fisici che possono alterare i segnali che passano sul mezzo e, quindi, il messaggio ricevuto potrebbe essere anche molto diverso dal messaggio originale; questa quantità di errore è detta **bit-error-rate**. Il problema è quindi quello di cercare di creare una comunicazione che sia quanto più possibile affidabile, cioè una comunicazione che metta al centro l'integrità del messaggio, anche a costo di rallentare la velocità di trasmissione. Per fare ciò dobbiamo introdurre sopra il livello fisico una serie di funzionalità che permettono di garantire l'integrità del messaggio; queste funzionalità sono incluse nel **livello di collegamento (data-link)**.

## 1 Funzionalità del livello *data-link*

Il livello di collegamento mette a disposizione tutta una serie di servizi, il cui scopo è proprio quello di mantenere un trasferimento che sia quanto più possibile integro e coerente. Il primo servizio messo a disposizione è quello di **framing**. L'idea è quella di suddividere il messaggio in dei pacchetti più piccoli, detti **frame**, e di incapsularne questi **frame** attraverso un **header** e un **trailer**. In particolare:

- L'**header** permette di specificare alcune informazioni di contenuto, tra cui la posizione del blocco interno al frame nella sequenza iniziale.
- Il **trailer** permette di specificare informazioni di verifica o di controllo del frame.

Insieme al servizio di **framing**, il livello di collegamento mette a disposizione un servizio di **accesso al collegamento**. Definisce quindi un protocollo di accesso al mezzo trasmittivo e che specifica le regole con cui immettere i frame nel collegamento. Ad esempio, nel caso che abbiamo preso in analisi all'inizio del capitolo, il protocollo permette l'accesso al mezzo fisico quando il canale risulta libero. Un altro servizio fornito dai protocolli a livello di collegamento è quello di **consegna affidabile**. Questi servizi sono spesso utilizzati per collegamenti soggetti a elevati tassi di errore, allo scopo di correggere l'errore localmente, senza dover procedere a una ritrasmissione dei dati da sorgente a destinazione. Quando però un collegamento presenta un basso **bit-error-rate** questo servizio può risultare anche trascurabile, limitando quindi l'**overhead** totale sul sistema. Inoltre, visto e considerato quanto detto all'inizio, il livello di controllo mette a disposizione dei meccanismi di rilevazione e correzione degli errori attraverso l'utilizzo di alcuni bit che possono essere agilmente controllati dal nodo di arrivo.

## 2 Implementazione del livello di collegamento

Il protocollo del livello di collegamento è sostanzialmente realizzato da un **adattatore di rete**, noto anche come **scheda di rete**. Il cuore di questo componente è costituito dal **controller a livello di collegamento**, ove la maggior parte delle funzionalità è implementata in **hardware**. Dal punto di vista del mittente il controller prende in ingresso un datagramma, lo incapsula in un frame di livello collegamento e lo trasmette sul canale di comunicazione. Dall'altro lato, un controller riceve l'intero frame, estrae il datagramma e lo consegna al livello di rete

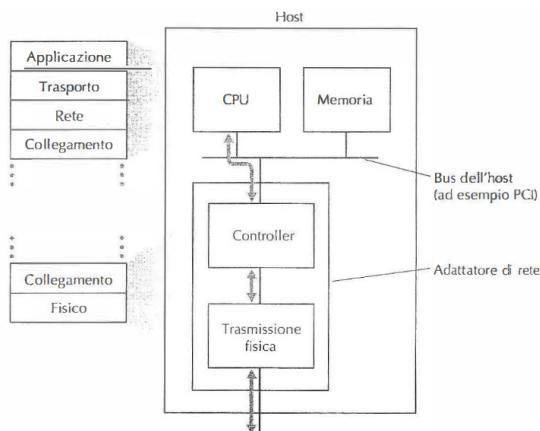


Figura 11.2: Relazione dell'adattatore di rete con le altre componenti del sistema

## 3 Rilevazione e correzione degli errori

Uno dei servizi offerti da livello di collegamento è un servizio che permette la **rilevazione** e la **correzione** degli errori. Come abbiamo infatti detto la trasmissione non è ideale e i mezzi fisici

sono soggetti a fenomeni di varia natura che ne possono alterare il segnale, portando ad avere un messaggio non corrispondente all'origine. L'idea della tecnica di rilevazione e correzione degli errori è di aggiungere, al **datagramma** in uscita, generalmente in coda, dei bit aggiuntivi, detti **EDC** (o *Error Detection Correction*). Supponiamo di avere la seguente situazione

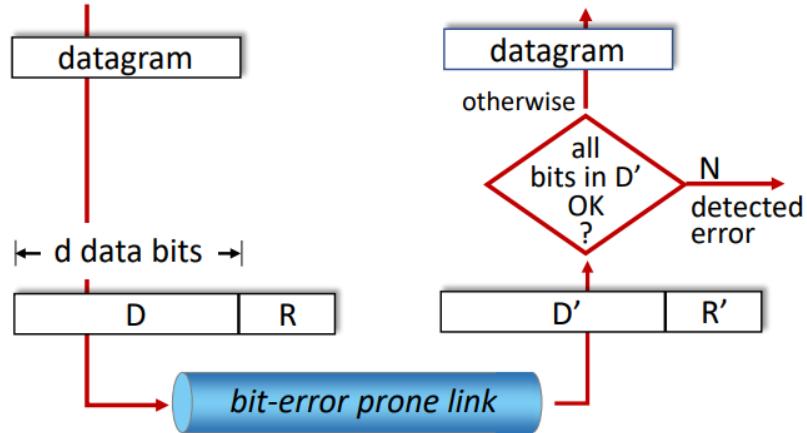


Figura 11.3: Rilevazione e correzione degli errori. I bit **EDC** sono indicati con **R**, o **bit di ridondanza**

Il mittente invia quindi il datagramma costituito da  $D$  e  $EDC$  e, il destinatario, riceverà il datagramma costituito da  $D'$  e  $EDC'$ , il quale potrebbe essere eventualmente diverso per via di alterazioni del segnale sul mezzo. Il primo problema che si presenta è la necessità del destinatario di determinare se  $D'$  coincide con  $D$  potendo contare solo su  $D'$  e  $EDC'$ , ed è quindi importante stabilire delle convenzioni che permettano di farlo. Tuttavia queste tecniche non sono esenti da errori; in particolare

- Nel caso in cui i bit  $d$  siano stati modificati, ma i bit di **EDC** no, il datagramma potrebbe essere considerato non corrotto nonostante lo sia.
- Nel caso in cui i bit  $d$  non siano stati alterati, ma i bit di **EDC** si, il datagramma potrebbe essere considerato corrotto nonostante non lo sia.

Abbiamo quindi delle situazioni in cui si può mancare la rilevazione dell'errore o delle situazioni in cui vi è un falso allarme. L'idea è quindi di optare per uno schema di rilevazione che permetta di minimizzare la probabilità che queste eventualità avvengano.

### 3.1 Controllo di parità

La forma più semplice di rilevamento degli errori è quella che utilizza un solo bit di **parità**. Supponiamo che si debba inviare un datagramma da  $d$  bit. In uno schema di parità il mittente include un bit addizionale e scegliere il suo valore in modo da rendere pari il numero di bit che costituiscono il datagramma; quindi:

- Se il numero di bit di  $d$  è dispari, allora  $R = 1$ .
- Se il numero di bit di  $d$  è pari, allora  $R = 0$ .

Una volta che il datagramma è giunto al mittente, esso dovrà solamente contare il numero di bit che costituiscono il datagramma: se il numero è pari il datagramma è corretto, altrimenti è alterato. Nonostante questa tecnica sia facilmente implementabile è poco utilizzata. Il motivo è dato dal fatto che ogni qualvolta il numero di bit alterati è pari, allora, vi è una mancata rilevazione dell'errore.

### 3.2 Checksum

Nelle tecniche che sfruttano il checksum, un pacchetto da  $d$  bit, viene suddiviso in delle sequenze, della stessa dimensione, pari a  $k$  bit che vengono successivamente sommate. In particolare, una volta suddiviso il pacchetto, viene fatta la somma in complemento a 1 di tutte le sequenze ottenute, ottenendo un’ulteriore sequenza di  $k$  bit. Questa sequenza viene infine inserita all’interno del pacchetto, componendo così un datagramma di  $d + k$  bit. Una volta giunto al destinatario avviene la rilevazione dell’errore: quest’ultimo scomponete nuovamente il pacchetto ed esegue la somma di tutte le sequenze, per poi confrontarla con quella ricevuta in ingresso. Se le due somme sono uguali allora il datagramma non è stato alterato, altrimenti sì.

Questa tecnica è utilizzata anche a livello di trasporto, dove i protocolli **TCP** e **UDP** utilizzano il checksum su tutti i campi dell’intestazione. L’utilizzo di questa tecnica, a livello di trasporto, rispetto a tecniche più sofisticate, è dovuta dal fatto che il livello di trasporto è implementato a livello software ed è quindi necessario disporre di tecniche che siano facilmente implementabili. Mentre la rilevazione degli errori a livello di collegamento è implementata in **hardware** e possiede dei circuiti dedicati, per cui è possibile anche utilizzare gli algoritmi più sofisticati e computazionalmente meno efficienti.

### 3.3 Controllo a ridondanza ciclica

Una tecnica di rilevazione degli errori largamente utilizzata nelle reti più moderne si basa su dei **codici di controllo a ridondanza ciclica**. Supponiamo che il **sender** stia cercando di inviare un pacchetto  $D$  costituito da  $d$  bit ad un **reciever**. Per sfruttare questo tipo di controllo è necessario che **sender** e **reciever** si mettano, prima, d’accordo su una stringa di  $r+1$  bit, chiamata **generatore** (indicata con  $G$ ). A questo punto il **sender** unirà al messaggio  $D$  una stringa di  $r$  bit,  $R$ , ottenendo un datagramma da  $d+r$  bit. Questa sequenza sarà intepretata come un numero binario esattamente divisibile per  $G$ . A questo punto il processo di controllo è semplice, una volta arrivata a destinazione la stringa di  $d+r$  bit verrà divisa per  $G$ :

- Se l’operazione restituisce un resto nullo allora il messaggio è inalterato.
- Se l’operazione restituisce un resto non nullo allora il messaggio è alterato.

Tutti i calcoli del **CRC** sono eseguiti in aritmetica modulo due senza riporti nelle addizioni e prestiti nelle sottrazioni.

### 3.4 Controllo di parità bidimensionale

Uno dei problemi degli algoritmi visti fino ad ora è la loro incapacità di correggere eventuali errori nei pacchetti, essi sono infatti solamente in grado di identificare se il pacchetto è alterato, non dove è stato alterato. Esistono però delle tecniche che consentono di identificare e anche di correggere gli errori, senza dover richiedere il pacchetto al **sender**. L’algoritmo a cui faremo riferimento è noto come **schema di parità bidimensionale**: si compone una matrice suddividendo il pacchetto in gruppi da  $k$  bit. Una volta composta la matrice si calcola il bit di parità per ogni riga e per ogni colonna.

			Riga di parità	
Colonna di parità				
	$d_{1,1}$	$\dots$	$d_{1,j}$	$d_{1,j+1}$
	$d_{2,1}$	$\dots$	$d_{2,j}$	$d_{2,j+1}$
	$\dots$	$\dots$	$\dots$	$\dots$
	$d_{i,1}$	$\dots$	$d_{i,j}$	$d_{i,j+1}$
	$d_{i+1,1}$	$\dots$	$d_{i+1,j}$	$d_{i+1,j+1}$

Figura 11.4: Caption

Una volta composta la matrice il pacchetto verrà inviato al **reciever** con l’aggiunta dei bit di

parità ottenuti. Il controllo diventa poi estremamente semplice, per ogni riga e per ogni colonna si verifica se il numero di bit è pari e, in caso non affermativo, l'intersezione tra la **riga alterata** e la **colonna alterata** corrisponderà al bit alterato.

## 4 Trasferimento dati affidabile

Uno dei servizi offerti dal livello di collegamento è quello del trasporto dati affidabile (o **reliable data transfer**). Idealmente ci piacerebbe che il trasferimento avvenga in modo affidabile, garantendo l'integrità e la correttezza del messaggio. Affinché sia possibile rendere affidabile un canale fisico, generalmente non affidabile, occorre definire un protocollo che si trova, in senso logico, al di sopra del canale fisico. Questo protocollo prende il nome, con molta fantasia, **RDT**; il funzionamento è relativamente semplice

- Dal lato **sender** il protocollo mette a disposizione una primitiva `rdt_send()` per inviare dati in modo affidabile.
- Dal lato **reciever** il protocollo mette a disposizione una primitiva `rdt_receive()` per ricevere dati in modo affidabile.

Entrambe queste primitive sfruttano i sistemi di **error-detection** e **error-correction** che abbiamo visto fino ad ora insieme a delle primitive `udt_send()` e `udt_receive()` che operano in modo non affidabile. Esistono diverse versioni di questo protocollo, che vanno dalla versione 1.0 fino alla versione 3.0, man mano aumentando di complessità. Nell'**rdt 1.0** il protocollo ha un funzionamento immediato e descrivibile attraverso un formalismo a **macchina a stati finiti**

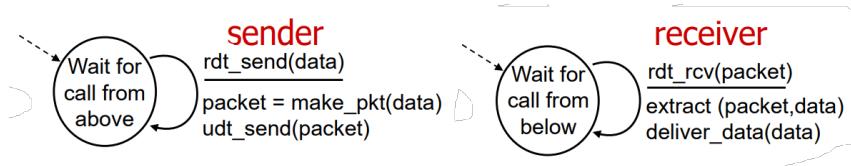


Figura 11.5: Formalismo **FSM** per **rdt 1.0**

Come si osserva immediatamente dalla macchina a stati finiti, il problema principale di questo protocollo è legato ad un'ipotesi fondamentale che viene fatta a priori: **il canale sottostante e il mezzo trasmisivo sono ideali**. L'unico evento che permette di attivare la macchina lato **sender** è che il client abbia un blocco di dati da inviare. A questo punto viene invocata una `rdt_send()` che, di fatto, è composta da due operazioni

- `make_packet(pkt)`: funzione che permette di costruire il frame da inviare a partire da un pacchetto passato dai livelli successivi. Supponiamo che il risultato della funzione sia memorizzato in `packet`
- `udt_transfer(packet)`: trasferisce il pacchetto attraverso il canale non sicuro.

Dal punto di vista del **reciever** si ha una situazione opposta. Infatti l'unica azione che permette di avviare la macchina a stati finiti, per questo attore, è che arrivino dei dati dal livello sottostante. Una volta ricevuti questi dati si procede con una `rdt_rcv()`, costituita anch'essa da due operazioni:

- `extract(packet)`: estrae il pacchetto verificando la presenza di eventuali errori. Supponiamo che il risultato della funzione sia memorizzato in `pkt`
- `data_transfer(pkt)` restituisce al livello soprastante il pacchetto.

Ovviamente questa versione permette solo di comprendere intuitivamente il funzionamento del protocollo **rdt**, ma non è in alcun modo utilizzabile nella realtà per via dell'assunzione iniziale. Nella seconda versione del protocollo rimuoviamo l'ipotesi di affidabilità del canale fisico e utilizziamo un approccio **stop and wait**. L'idea di questo approccio è che a ogni invio il **sender** attenda,

prima di inviare un altro pacchetto, che il **receiver** mandi un segnale di conferma: **ok** (*acknowledgement*) se il pacchetto è arrivato ed è integro, altrimenti **no** (*negative acknowledgement*). In base al valore mandato dal **receiver** il **sender** può decidere di rimandare il pacchetto o procedere con il successivo; questo fatto si traduce nella necessità di avere due stati dal lato del **sender**, dove, lo stato iniziale corrisponde all'arrivo di un pacchetto dal livello soprastante e il secondo stato è quello di attesa del segnale. Nel formalismo a macchine a stati finiti otteniamo una struttura del tipo

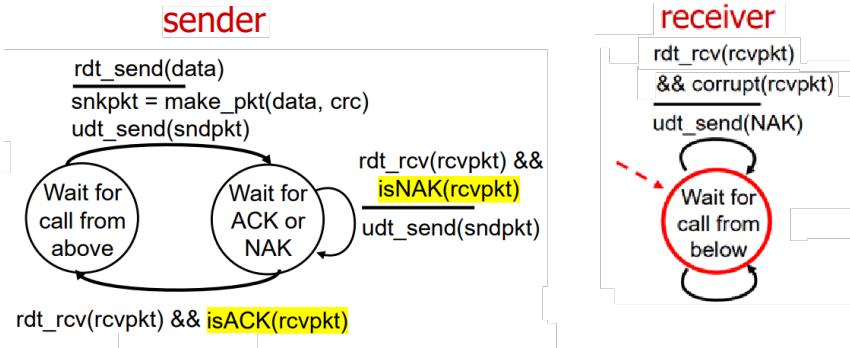


Figura 11.6: Formalismo a stati finiti per rdt 2.0

Questa versione contiene però un errore importante: è stato assunto che i pacchetti di **ACK** e i **NACK** non siano affetti da alcun errore di trasmissione. Vista la natura dei mezzi trasmittivi, una assunzione di questo tipo è problematica. Una prima idea di soluzione potrebbe essere quella di trattare ogni pacchetto di conferma corrotto come un **NACK** implicito. Sicuramente, a primo impatto, un approccio di questo tipo non è molto vantaggioso: ad ogni pacchetto corrotto il **sender** si trova costretto a rimandare il medesimo pacchetto al **receiver** senza alcuna idea sulla natura della conferma. L'unico modo affinché una soluzione di questo tipo funzioni è quella di andare a cambiare il funzionamento del protocollo sia lato **sender** che lato **receiver**. Il primo passo è quello di introdurre un numero di sequenza, cioè la posizione del pacchetto nella pipeline, ad ogni pacchetto. Insieme al numero di sequenza dobbiamo introdurre anche un contatore sia per il **sender** che per il **receiver** (nell'esempio ci limitiamo a un contatore a singolo bit), che tenga traccia del pacchetto che, rispettivamente, deve inviare/ricevere. Introducendo queste due modifiche otteniamo una struttura della macchina a stati finiti del **sender** così strutturata

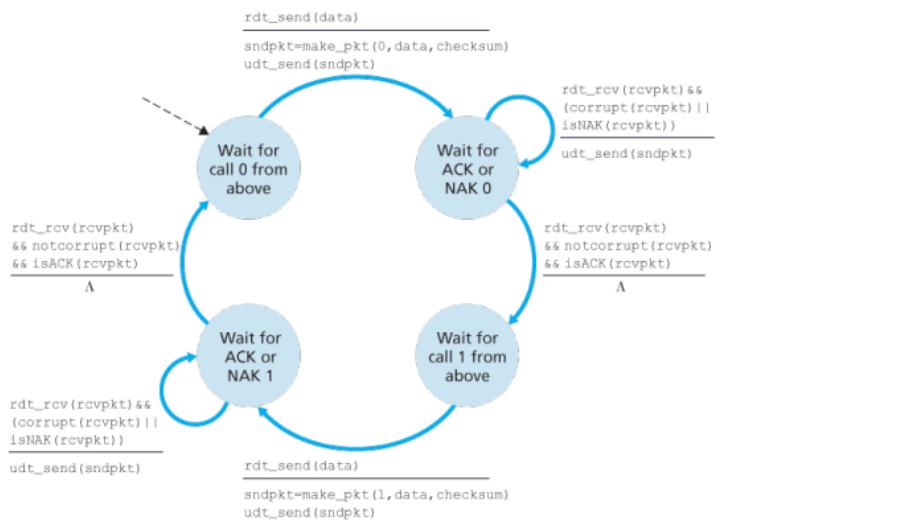


Figura 11.7: rdt 2.1. Macchina a stati finiti del sender

Ovviamente dobbiamo duplicare gli stati, così da avere due stati per ogni numero di sequenza possibile. Il funzionamento generale rimane poi invariato rispetto alla versione `rdt 2.0`, con l'unica differenza che si aggiunge una possibile azione del **sender** e del **reciever**

- Il **sender**, quando riceve un pacchetto di **ACK/NACK** corrotto andrà ad inviare nuovamente il pacchetto.
- Il **reciever** dovrà implementare un controllo sul numero di sequenza del pacchetto affinché possa rilevare se ha ricevuto un nuovo pacchetto o un duplciato.

Si può concludere facilmente che anche la struttura del **reciever** andrà modificata introducendo un ulteriore stato, così da avere uno stato per ogni numero di sequenza, e introducendo il controllo sul numero di sequenza:

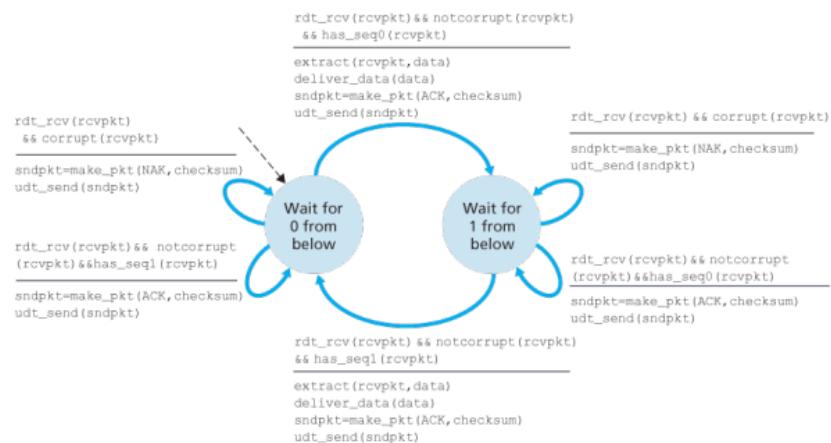


Figura 11.8: `rdt 2.1`. Macchina a stati finiti del sender

Questo approccio è, in generale, funzionante per una situazione reale. Possiamo però provare a ottimizzarlo ulteriormente: supponiamo di avere un singolo pacchetto che permetta di fare sia da **NACK** che da **ACK**. Questa versione dell'`rdt`, detta `rdt 2.2`, utilizza gli **ACK** sia in caso di conferma che di conferma negativa; supponiamo che sia arrivato un pacchetto  $p$  corrotto e che il **reciever** lo voglia segnalare al **sender**, l'idea potrebbe essere quella di inviare, nuovamente, il segnale di **ACK** per il pacchetto  $p - 1$ . In altre parole, un **ACK** duplicato per il medesimo pacchetto indica che il **reciever** non ha capito l'ultimo pacchetto inviato e, quindi, deve essere ritrasmesso. Nonostante a primo impatto possa sembrare contro-intuitivo, possiamo trovare tante situazioni reali in cui un protocollo di questo tipo viene utilizzato; ad esempio, supponiamo che due persone, **Mario** e **Pippo**, debbano scambiarsi informazioni stradali

```

Mario: prendi la seconda uscita alla rotonda.
Pippo: va bene.
Mario: svolta a destra al primo incrocio.
Pippo: va bene.
Mario: svolta di nuovo a destra
Pippo: "informazioni incomprendibili"
Mario: svolta di nuovo a destra
Mario: poi... "informazioni incomprendibili"
Pippo: va bene fino alla svolta a destra al primo incrocio.
Mario: poi svolta a sinistra alla seconda uscita.
    
```

In questo esempio notiamo entrambe le funzionalità che sono state discusse per l'`rdt 2.1/2.2`. Nel primo caso vediamo come, nel momento in cui la conferma di pippo, non è arrivata chiaramente a Mario, egli abbia ripetuto il messaggio. Per quanto riguarda il secondo caso, Pippo, per segnalare che non aveva capito l'ultima informazione, ha utilizzato un messaggio di **ACK**. Non

ha quindi avuto bisogno di utilizzare un messaggio di **NACK**, ottimizzando, in un certo senso, la comunicazione.

La stessa idea è applicabile anche per la trasmissione affidabile dei dati semplicemente modificando la struttura della funzione `is_ack`, affinché esegua un controllo sul numero di sequenza. In particolare, se il numero di sequenza relativo all'**ACK** corrisponde al numero di sequenza del pacchetto precedente, allora si procede al rinvio del pacchetto.

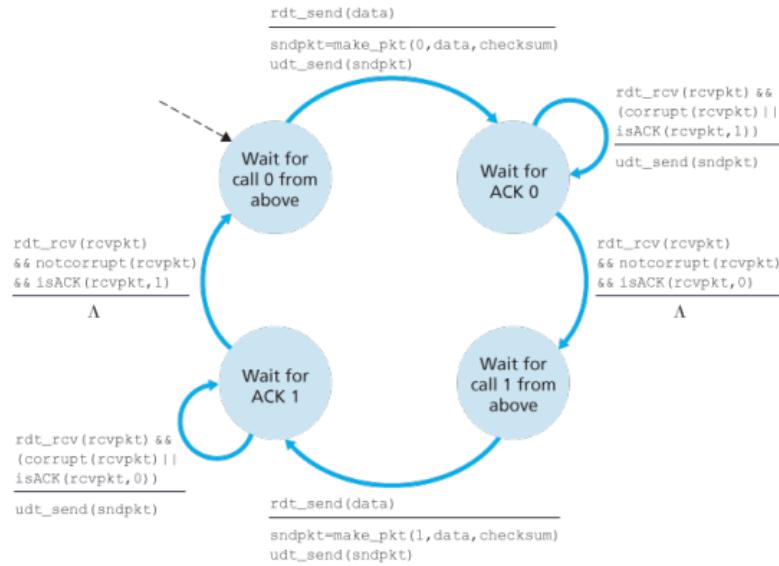


Figura 11.9: `rdt 2.2`. Macchina a stati finiti del *sender*

Lo stesso ragionamento lo possiamo applicare al per il **reciever**, dove il segnale di **NACK** dovrà essere inviato spedendo un segnale di **ACK** per il pacchetto con il numero di sequenza del pacchetto precedente.

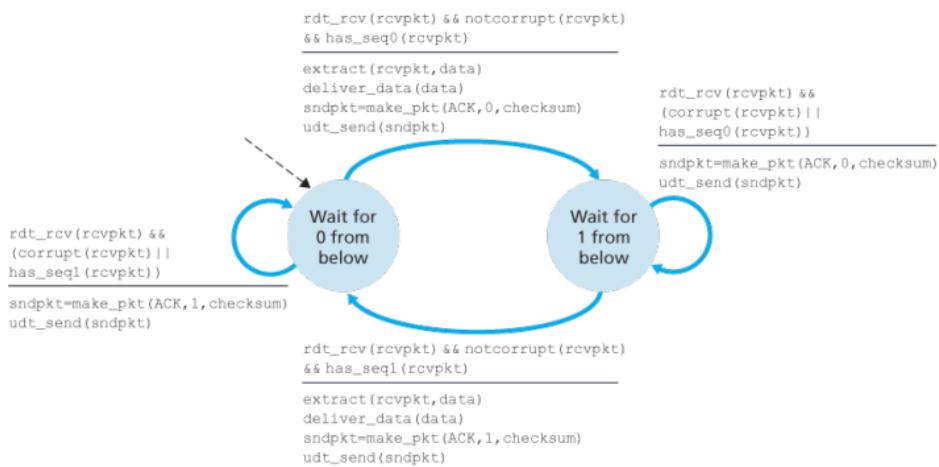


Figura 11.10: `rdt 2.2`. Macchina a stati finiti del *reciever*

Nell'ultima versione del protocollo `rdt`, quella 3.0, viene rimossa un'ulteriore ipotesi. Fino ad ora avevamo infatti supposto che i pacchetti potessero essere corrotti, ma mai persi. Introduciamo ora la possibilità che alcuni pacchetti possano andare persi durante il passaggio nel mezzo trasmissivo. Anche in questo caso possiamo sfruttare un'analogia dei protocolli di comunicazione umani; supponiamo infatti che Alice e Bob stiano parlando al telefono

```

        Alice: Ciao Bob!
        Bob: Ciao Alice!
        Alice: Come stai ?
        Bob: ...
        > attesa di un certo intervallo di tempo
        Alice: Come stai ?
        Bob: tutto bene dai. Te come stai ?
    
```

Come si nota immediatamente dall'esempio, nel momento in cui la risposta di Bob è andata persa, Alice ha atteso un certo intervallo di tempo e, successivamente, ha rispedito il pacchetto precedente. In altri termini, al passaggio di un certo intervallo di tempo (**timeout**) senza risposta, il **sender** o il **reciever** rimanderanno nuovamente il pacchetto. Per introdurre questa modifica è sufficiente introdurre un ramo aggiuntivo nello stato di attesa dell'**ACK**; in particolare, ad ogni invio di pacchetto viene fatto corrispondere l'avvio di un timer e se, una volta che il sistema si trova nello stato di attesa dell'**ACK**, il timeout viene violato, allora, il pacchetto verrà inviato nuovamente.

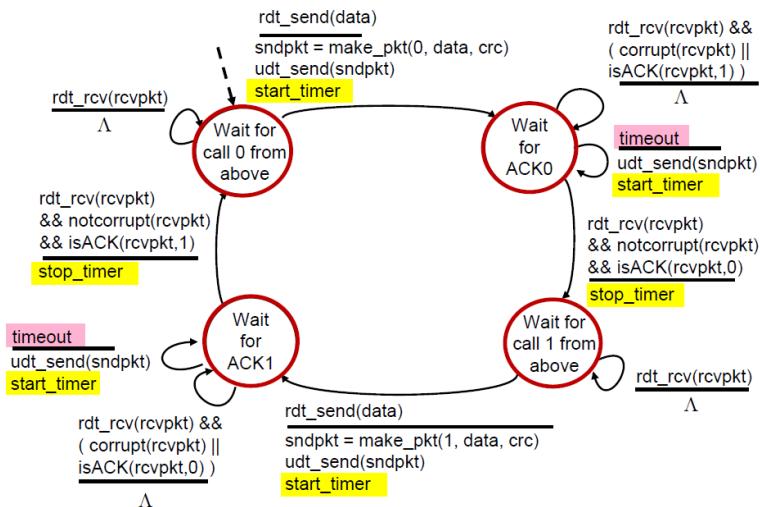


Figura 11.11: **rdt 3.0**. Macchina a stati finiti del sender

Nonostante questa versione sembri funzionante, presenta comunque un problema non indifferente: se il timeout è troppo piccolo si verifica il rischio di avere un **timeout prematuro**, inviando nuovamente un pacchetto che non era stato perso, ma per cui l'**ACK** ha ancora da arrivare.

## 5 Protocolli a pipeline

Nella sezione sull'**rdt** è stato approfondito un approccio del tipo **Stop-and-Wait**, dove ad ogni invio il **sender** si pone in attesa di una conferma del **reciever**. Questo approccio, per quanto sicuro possa essere, risulta estremamente sconveniente dal punto di vista computazionale. Un altro approccio possibile è quello di prevedere che il **sender** possa inviare più pacchetti senza dover attendere l'**ACK** del **reciever**. Questo approccio però presenta dei limiti, in quanto non è possibile spedire un numero infinito di pacchetti di fila, è necessario fissare un tetto massimo di pacchetti che si possono spedire per un singolo invio. Inoltre, affinché sia possibile adottare questa tipologia di protocolli è necessario introdurre dei meccanismi addizionali:

- Aumentare il range di numeri di sequenza.
- Introdurre un meccanismo di buffering, sia per il **sender** che per il **reciever**.

La dimensione del buffer o il range dei numeri di sequenza dipendono da come il protocollo gestisce tutta una serie di problematiche che si possono verificare, come pacchetti **corrotti** e/o **persi** e ritardi sul mezzo trasmissivo. I due protocolli a pipeline principali sono:

- Go-back-N.
- Selective repeat.

### 5.1 Go-back-N

L'idea alla base del protocollo Go-Back-N è che il mittente possa trasmettere più pacchetti senza dover attendere alcun ACK, ma non possa avere più di un certo numero di pacchetti  $n$ , senza ACK, all'interno della pipeline. In altri termini, l'ACK del **Go-Back-N** è cumulativo e non riferito al singolo pacchetto. Prendiamo in esame la seguente situazione:

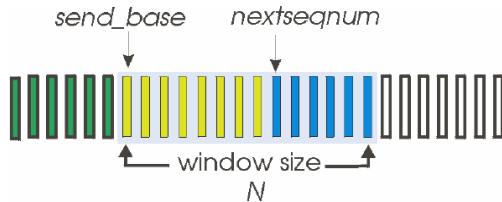


Figura 11.12: Pipeline di messaggi. L'indice **base** indica il primo pacchetto non ancora spedito, mentre, l'indice **nextseqnum** indica il primo numero di sequenza non ancora assegnato ad un pacchetto

Il funzionamento del **Go-Back-N** è relativo semplice. Se il livello superiore chiama infatti la funzione per trasferire i dati (`rdt_send`) allora il mittente controlla, prima di tutto, se la finestra sia piena, ossia se vi siano  $N$  pacchetti in sospeso senza ACK. Se la finestra non è piena, crea e invia il pacchetto, con conseguente aggiornamento delle variabili, mentre se la finestra è piena, allora il mittente restituisce i dati a livello superiore e ritenterà in seguito. Nel **Go-Back-N** l'arrivo di un ACK con numero di sequenza  $n$  indica che tutti i pacchetti con un numero di sequenza minore o uguale a  $n$  sono stati correttamente ricevuti dal destinatario. Infine, un terzo evento di cui si deve tenere conto è lo scadere di un timeout. Se infatti si verifica un timeout, il mittente invia nuovamente tutti i pacchetti spediti che ancora non hanno ricevuto un ACK. Il destinatario del Go-Back-N è infatti un'entità che di sua natura manda gli ACK per i pacchetti arrivati in ordine e li consegna a livello superiore, mentre non memorizza i pacchetti che arrivano fuori ordine, ma li scarta, duplicando l'ACK per l'ultimo pacchetto che arriva in ordine.

### 5.2 Selective Repeat

Questo protocollo adotta invece un approccio diverso da quello adottato dal **Go-back-N**. Nel **selective repeat** l'**ACK** viene inviato per il singolo pacchetto piuttosto che per l'intero gruppo. Ad esempio, supponiamo di avere la stessa situazione di prima

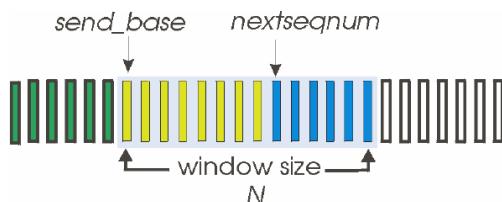


Figura 11.13: Pipeline di messaggi

Man mano che il **sender** manda pacchetti, il **receiver** li memorizza all'interno del proprio buffer e manda la conferma per ogni singolo pacchetto. Se il pacchetto non arriva o, per qualche motivo, l'**ACK** del **receiver** per quel pacchetto non arriva, il **sender** si limita solamente a rimandare il pacchetto in questione, senza dover rimandare necessariamente anche gli altri.

### 5.3 Conclusioni

Il primo dei due protocolli, per quanto possa sembrare più inefficiente, era estremamente utilizzato nel periodo in cui erano in uso i **mainframe**. Quindi grandi calcolatori, ricchi di risorse, da cui altri calcolatori scaricavano dati. Essendo i primi ricchi di risorse computazionali, non impiegavano molta fatica a inviare pacchetti a grandissime velocità e, quindi, si preferiva alleggerire il carico dal lato dei **reciever**, limitando il numero di **ACK**. Mentre, ad oggi, dove i calcolatori sono tutti estremamente potenti si predilige un approccio del secondo tipo, che ottimizza il numero di pacchetti spediti.

## 6 Point-to-Point Protocol

Tra i protocolli a livello **data-link** troviamo il protocollo **Point-to-Point**. Questo protocollo funziona per dispositivi che sono direttamente connessi mediante un canale, che può essere fisico o logico; generalmente il mezzo trasmissivo era una linea telefonica. Il **PPP** è un protocollo *data-link* che, come tale, implementa **alcune** delle funzionalità di questo livello:

- **Packet framing.** Il PPP definisce una struttura esatta per i frame che vengono scambiati tra due host.
- **Error detection.**
- **Bit transparency.** Il PPP permette al livello soprastante di trasmettere qualunque sequenza di bit senza limitazioni sul formato.
- **Negoziazione dei parametri.** Il PPP prevede che prima di uno scambio mittente e destinatario decidano quali parametri del frame utilizzare, facendo una vera e propria negoziazione.

Il PPP tuttavia non offre alcuni servizi che invece sappiamo essere fondamentali nelle comunicazioni, primo tra tutti il servizio di **reliable-data-transfer** e, in secondo luogo, il servizio di **error correction**. Abbiamo detto che tra i servizi offerti vi è quello di framing del pacchetto, possiamo quindi introdurre la struttura di un frame secondo **PPP**:

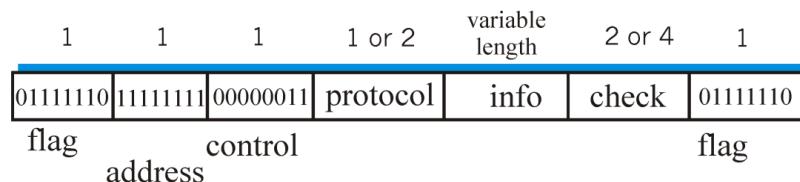


Figura 11.14: Struttura di un frame secondo PPP

Il primo campo che possiamo notare è chiamato **flag**; questo campo, che si trova in **testa** e in **coda** al frame, ha come scopo principale quello di inquadrare effettivamente il messaggio nel frame, definendo un confine che il **reciever** è in grado di riconoscere. I due campi successivi, quello **address** e **control**, sono campi che non hanno uno scopo ben preciso:

- Il campo **address** era stato introdotto per poter avere una comunicazione **point to multi-point**.
- Il campo **control** è stato pensato per una prospettiva futura.

Questi campi possono essere negoziati e, quindi, esclusi dalla comunicazione tra due dispositivi. Il campo dati è, invece, un campo di lunghezza variabile, compresa tra 0 e 1500 byte, anch'esso negoziabile, che ha come unico scopo quello di memorizzare i dati che devono essere inviati al destinatario. Il campo **check**, anch'esso variabile tra due e quattro byte, contiene i bit necessari al controllo dell'errore, che viene fatto mediante **Controllo a Ridondanza Ciclica**.

Una delle problematiche legate alla trasmissione **PPP** è quella del bit transparency. Infatti se nel campo data venisse inserita una sequenza di bit corrispondente a una **sequenza nota**, ad

esempio quella di flag, il **receiver** potrebbe avere dei problemi nell'interpretazione del pacchetto. Affinché sia quindi possibile permettere al livello soprastante di mandare una qualsiasi sequenza di bit è opportuno che vengano introdotte delle **sequenze di escape**. Queste sequenze vanno poste prima del byte che deve essere trasmesso.

# Capitolo 12

## Collegamenti ad accesso multiplo

Il collegamento in rete tra host può essere di due tipologie: **collegamento punto a punto** e **collegamento broadcast**. Nel capitolo scorso abbiamo studiato il primo caso, quello nel quale un collegamento costituito da un **trasmettente** a un'estremità del collegamento e da un unico **ricevente** all'altra. In questo capitolo studieremo invece la seconda tipologia di collegamento, quello in cui il collegamento è condiviso tra una moltitudine di host diversi. Quindi, come abbiamo appena detto, nel collegamento **broadcast** abbiamo un mezzo condiviso su cui vari host possono trasmettere e ascoltare. L'accezione **broadcast** indica che ogni comunicazione è indirizzata dall'**host trasmettente** verso ogni altro **host ricevente** che è in ascolto sul canale. Da un punto di vista reale possiamo immaginare un collegamento **broadcast** come una qualsiasi lezione universitaria, dove il docente parla e tutti gli alunni (idealmente) ascoltano. Affinché però una connessione di questo tipo possa funzionare è necessario che esistano tre presupposti:

- Il mezzo deve essere condiviso.
- Quando un host manda un frame sul mezzo condiviso gli altri host devono rimanere in attesa.
- Gli host devono coordinarsi tra loro.

Nel caso della lezione universitaria il mezzo fisico è l'aria, nel caso di una connessione tra dispositivi il mezzo fisico è, solitamente, o ethernet o wi-fi. Quando si parla di connessioni **broadcast** il problema principale è però quello che viene fuori analizzando il terzo punto: la condivisione tra host. Se infatti gli host non si coordinassero i pacchetti potrebbero scontrarsi e corrodere a vicenda; questa situazione viene detta in gergo **collisione**. La stessa cosa la potremmo dire anche per una lezione universitaria, se infatti tutti gli studenti iniziassero a fare domande contemporaneamente allora sarebbe impossibile per il docente e per gli altri colleghi in ascolto riuscire a capirle. L'unica soluzione a quello che, in gergo, viene detto **problema dell'accesso multiplo** è creare un **protocollo di accesso al canale** che definisca un'insieme di regole con cui gli host possono accedere al mezzo trasmissivo senza, o comunque minimizzando, collisioni. Per definire il protocollo occorre definire un'insieme di regole; riprendendo l'analogia della lezione universitaria potremmo prendere un'insieme di regole del tipo

"Tutti possono parlare"  
"Alzare una mano per parlare"  
"Se qualcuno sta parlando, non interromperlo"  
"Rimani in ascolto se qualcuno sta parlando"

Nelle reti non è però così semplice determinare un'insieme di regole per un semplice motivo: il canale è unico. Nell'esempio abbiamo supposto, implicitamente, che ci siano più canali che permettono la comunicazione, corrispondenti ai sensi umani. In una rete di calcolatori abbiamo solo un canale su cui solo un dispositivo può comunicare e, quindi, non possiamo definire una regola che emuli l'*alzata di mano*. Esistono alcuni protocolli che possiamo utilizzare, alcuni meno efficienti e altri più efficienti:

- Un primo approccio potrebbe essere quello che ogni host attenda un tempo  $t$  casuale, diverso idealmente per ogni host, prima di comunicare.
- Un secondo approccio potrebbe essere quello che un host che vuole parlare deve prima possedere il token. Il token sarà passato casualmente dall'host che lo possedeva a un altro.

Entrambi questi protocolli possono essere funzionali, ma presentano comunque delle problematiche non indifferenti. Dobbiamo quindi cercare di definire un protocollo in modo più formare e, per farlo, dobbiamo prima definire quello che ci piacerebbe che il protocollo facesse. Idealmente ci piacerebbe che il nodo o più nodi ( $M$ ) che vogliono trasmettere lo possano fare alla massima velocità possibile che, nel caso del singolo nodo questa velocità dovrebbe essere pari al **throughput** del canale  $R$ , mentre nel caso di nodi molteplici ci piacerebbe che fosse  $R/M$ . Inoltre, idealmente, il protocollo dovrebbe essere completamente decentralizzato, rimuovendo quindi la necessità di un entità centrale che regola la comunicazione. Esistono diversi protocolli che possono rispettare i nostri requisiti, ma in generale possiamo raggrupparli in tre classi differenti:

- **Protocolli basati sul partizionamento del canale.** In questa tipologia di protocolli il canale trasmissivo viene partizionato a seconda di un criterio che può essere
  - Basato sulla frequenza.
  - Basato sul tempo.
  - Basato sul codice.
- **Protocolli ad accesso casuale.** Questi protocolli riprendono l'idea che avevamo enunciato in precedenza; ogni host attende un tempo casuale prima di comunicare.
- **Protocolli ad accesso circolare.** Gli host decidono dei turni per parlare.

## 1 Protocolli a partizionamento di canale

Questa classe di protocolli, come abbiamo detto, suddivide il canale in funzione di un criterio scelto dal programmatore; nel nostro caso utilizzeremo una divisione basata sul **tempo**. Il protocollo **Time Division Multiple Access** parte quindi da una suddivisione del tempo in **intervalli di tempo** e, successivamente, ogni intervallo viene poi suddiviso in **slot**. Nel caso specifico, supponiamo che ci sia uno slot per ogni host.

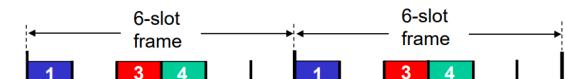


Figura 12.1: Assegnazione slot nei protocolli a comunicazione di canale

I problemi di un protocollo come questo sono, in realtà, disparati; il primo problema deriva dal fatto che un generico host potrà usare il mezzo trasmissivo solo per una frazione della sua capacità trasmissiva, inoltre, un protocollo di questo tipo necessita di un unità centrale che si occupi di definire esattamente cosa vuol dire **istante di tempo** e **slot**. Tuttavia, nonostante questi due svantaggi, **TDMA** ha il vantaggio di rimuovere completamente il problema delle **collisioni**.

## 2 Protocolli ad accesso casuale

In questa tipologia di protocollo ogni singolo dispositivo è in grado di trasmettere alla massima velocità consentita dal canale, cioè  $R$ . Tuttavia questo fatto preclude la possibilità che non si verifichino delle **collisioni** e, quindi, tutti i protocolli devono mettere a disposizione dei meccanismi che permettano di far giungere tutti i **frame** a destinazione. L'idea generale dei protocolli ad accesso casuale è che, nel caso si verifichino delle collisioni, gli host ritrasmettano i loro **frame** fintanto che non arrivano tutti a destinazioni. Le differenti scelte, arbitrarie, del tempo di attesa operate dai diversi nodi possono consentire ai **frame** di attraversare il canale senza ulteriore collisioni. A questa categoria appartengono tantissimi protocolli; i principali sono:

- **Slotted ALOHA.**
- **ALOHA.**
- **CSMA.**

Nel protocollo **Slotted ALOHA** si assume che ogni frame consista esattamente di  $L$  bit e il tempo sia suddiviso in **slot** di  $L/R$  secondi. Inoltre, valgono anche le ipotesi che i nodi cominciano la trasmissione dei frame solo all'inizio degli slot, i nodi sono sincronizzati in modo che tutti sappiano quando iniziano gli slot e, qualora in uno slot due o più frame collidano, tutti i nodi della rete rilevino l'evento prima del termine degli slot. Il funzionamento del protocollo è estremamente semplice:

- Quando un nodo ha un nuovo frame da spedire, attende fino all'inizio dello slot successivo e poi trasmette l'intero frame.
- Se non si verifica una collisione, l'operazione ha avuto successo, quindi non occorre effettuare una ritrasmissione e il nodo può predisporre l'invio di un nuovo frame
- Se si verifica una collisione, il nodo la rileva prima del termine dello slot e ritrasmette con probabilità  $p$  il suo frame durante gli slot successivi, fino a quando l'operazione non ha successo.

La probabilità  $p$  è un numero che varia tra 0 e 1; possiamo vederlo come il lancio di una moneta truccata, dove la testa, che esce con probabilità  $p$ , corrisponde alla ritrasmissione del pacchetto, mentre, la croce corrisponde alla ritrasmissione nello slot immediatamente successivo. Questo protocollo permette di ottenere dei vantaggi considerevoli: primo tra tutti la possibilità del canale di trasmettere continuamente alla massima velocità fintanto che il nodo è attivo, inoltre, **Slotted ALOHA** è anche fortemente decentralizzato poiché ogni singolo nodo è in grado di rilevare le collisioni e decidere indipendentemente quando ritrasmettere. Tuttavia questo protocollo presenta due problemi principali: il primo problema deriva dal fatto che quando ci sono molti nodi attivi la probabilità che più nodi scelgano lo stesso slot aumenta, mentre, il secondo problema è che una grande quantità di slot resterà vuota. Per comprendere meglio il secondo problema è utile analizzare l'**efficienza** del protocollo **Slotted ALOHA**, definita come *la frazione di slot (il numero di slot) in cui trasmette un solo nodo in presenza di un elevato numero di nodi attivi che hanno sempre un elevato numero di pacchetti da spedire*. Supponendo di avere  $n$  nodi la probabilità che un dato slot sia vincente è dato dalla probabilità che un solo nodo trasmetta, mentre i rimanenti  $n - 1$  rimangono inattivi:

$$\text{Probabilità che } (n - 1) \text{ nodi non spediscono} = P_S = (1 - p)^{n-1}$$

La probabilità di successo di un nodo è data dalla probabilità che questo nodo spedisca nello slot vincente, quindi pari a  $p \cdot P_S$  e, infine, siccome ci sono  $n$  nodi, la probabilità che uno tra gli  $n$  nodi spedisca nello slot vincente è pari a

$$\text{Efficienza} = N \cdot p \cdot (1 - p)^{n-1}$$

Studiando il limite per infinito di questa grandezza si osserva come l'efficienza di questo algoritmo sia pari solamente al 35% e, quindi, estremamente bassa. Nella precedente versione del protocollo, quello dell'**ALOHA** puro, l'efficienza era addirittura la metà di quella dello **Slotted ALOHA**.

In un'analogia con la comunicazione umana, possiamo vedere i protocolli di **ALOHA** come delle persone maleducate che continuano a parlare senza preoccuparsi del fatto che qualcun altro stia già comunicando. Tuttavia però, nella comunicazione umana, abbiamo anche protocolli che permettono di comunicare in modo civile e regolato, riducendo il numero di collisioni e aumentando la quantità di dati trasmessi; in particolare definiamo due regole per la comunicazione civile

- **Ascoltare prima di parlare.** Se qualcun altro sta parlando è necessario aspettare che abbia concluso. Nel mondo delle reti chiamiamo questo fatto **rilevamento della portante**: un nodo ascolta il canale prima di trasmettere.

- **Se qualcun altro comincia a parlare insieme a voi, smettete di parlare.** Nel modo delle reti questo fatto viene chiamato **rilevamento della collisione**: il nodo che sta trasmettendo rimane contemporaneamente in ascolto del canale. Se il nodo che sta trasmettendo rileva che un altro nodo sta trasmettendo un frame che interferisce con il suo, allora arresta la comunicazione.

Queste due regole sono alla base dei protocolli **CSMA** (*Carrier Sense Multiple Access*) e **CSMA/CD** (*CSMA with Collision Detection*). Il motivo per cui è necessario il meccanismo di rilevamento delle collisioni deriva dal fatto che i canali non sono ideali; prendiamo come esempio la situazione in cui all'istante  $t_0$  un nodo  $A$  ha da spedire datagramma. Il primo passo è quello di rilevamento della portante, quindi rileva se sul canale sta trasmettendo qualcun altro e supponiamo che  $A$  trovi il canale libero (deve essere libero per 9.6 microsecondi, cioè il tempo necessario a trasmettere 96 bit), il rilevamento avviene misurando la potenza di ricezione (se la potenza di ricezione è maggiore di quella a cui ha spedito si è verificata una collisione). A questo punto inizierà a trasmettere lungo il canale senza alcuna preoccupazione. Supponiamo che all'istante  $t_1$  anche il nodo  $B$  abbia un datagramma da spedire e analizziamo cosa succede

- Nel caso ideale il rilevamento della portante di  $B$  restituirà come risultato che il canale è attualmente occupato.
- Nel caso però in cui per qualche motivo all'istante  $t_1$  i bit di  $A$  non sono ancora arrivati a  $B$ , il rilevamento della portante restituirà che il canale è libero.

Nel secondo caso,  $B$ , nel pieno rispetto del protocollo, inizierà a spedire i propri frame senza preoccupazioni sul canale. Ad un certo punto però i frame di  $B$  inizieranno a fare interferenza con i frame di  $A$  e, se non viene previsto un meccanismo di rilevamento della collisione si corre il rischio di non accorgersi dell'interferenza. Nell'algoritmo **CSMA/CD** questo meccanismo viene implementato e, se  $A$  rileva che i suoi pacchetti stanno facendo interferenza con quelli di  $B$ , allora interromperà immediatamente la sua trasmissione, manda un segnale **32/48 bit-jam signal**, e si metterà in attesa per un intervallo di tempo **casuale** prima di ritrasmettere nuovamente. L'accezione **casuale** implica però delle problematiche non trascurabili

- Se l'intervallo di tempo è grande e il numero di nodi è piccolo, le attese potrebbero diventare considerevoli.
- Se l'intervallo di tempo è piccolo e il numero di nodi è grande, le attese potrebbero non essere sufficienti a limitare le collisioni.

L'idea migliore è quella di adottare un algoritmo, chiamato **binary exponential backoff**, che risolve tale problematica. L'idea dell'algoritmo di backoff esponenziale è quella di raddoppiare l'intervallo ad ogni collisione che si verifica; in particolare, nel momento in cui viene riscontrata la  $n$ -esima collisione, l'algoritmo stabilisce un valore  $K \in [0, 1, 2, \dots, 2^n]$  di attesa. Successivamente viene calcolato un tempo di attesa pari a:

$$\text{Attesa} = r \times \text{Slot Time}$$

Dove lo **Slot Time** è una costante definita dal protocollo stesso e identifica il tempo necessario affinché un segnale percorra la distanza massima del cavo (andata e ritorno) più il tempo di reazione dei jam signal (sostanzialmente è un RTT del segnale nella rete). Una volta atteso questo intervallo è necessario ripetere il rilevamento della portante. L'algoritmo quindi aumenta le attese esponenzialmente man mano che si verificano le collisioni, adattando dinamicamente il sistema al numero di nodi. Anche in questo caso possiamo definire l'efficienza dell'algoritmo come la frazione di tempo media durante la quale i frame sono stati trasferiti sul canale senza collisioni in presenza di un elevato numero di nodi attivi e con un'elevata quantità di frame da spedire. Definiamo anche  $d_{\text{prop}}$  come il tempo massimo che occorre al segnale per propagarsi tra una coppia di schede di rete; in altri termini, detto  $N$  l'insieme delle schede di rete

$$d_{\text{prop}} = \max_{(i,j) \in A} d_{\text{prop}, i}, \quad A = N \times N$$

definiamo anche  $d_{\text{trasm}}$  come il tempo necessario per trasmettere un frame di maggior dimensione possibile. A questo punto possiamo definire l'efficienza di CSMA/CD come

$$\text{Efficienza} = \frac{1}{1 + \frac{5d_{\text{prop}}}{d_{\text{trasm}}}} = \boxed{\frac{d_{\text{trasm}}}{d_{\text{trasm}} + 5d_{\text{prop}}}}$$

Osserviamo abbastanza facilmente che minore è il ritardo di propagazione tra le schede di rete, maggiore è l'efficienza dell'algoritmo. Inoltre, anche all'aumentare di  $d_{\text{trasm}}$  l'efficienza tende a crescere, il motivo è abbastanza intuitivo: nel momento in cui un frame si appropria del canale può trattenerlo per un periodo di tempo estremamente lungo; di conseguenza il canale svolge lavoro produttivo per più tempo.

### 3 Protocolli a rotazione

Tra le caratteristiche desiderate dei protocolli a **broadcast** troviamo il fatto che se ci sono  $m$  nodi attivi ci piacerebbe che ciascuno di essi abbia un **throughput** di  $R/M$  bit. Questa caratteristica non è però presente nei protocolli **ALOHA** e **CSMA**, dove invece l'accesso al canale è casuale. La terza e ultima possibilità implementativa è quella dei **protocolli a rotazione**. Uno dei principali protocolli a rotazione è il protocollo di **polling**, in questo protocollo uno dei nodi, disegnato come principale, interella a turno gli altri nodi. In particolare, il nodo principale invia un messaggio al nodo 1, comunicandogli che può trasmettere fino ad un certo numero di frame. Dopo che il nodo 1 ha trasmesso questi frame, il nodo principale, invia al nodo 2 la stessa comunicazione. Facendo in questo modo abbiamo un nodo che si pone da arbitro della comunicazione, ottenendo così due vantaggi principali:

- **Annulloamento delle collisioni.** La presenza di un arbitro fa sì che nessun nodo inizi a trasmettere sul canale fintanto che non è il suo turno.
- **Maggiore efficienza.** Riducendo il numero di collisioni e impiegando tutti gli slot temporali si aumenta l'efficienza generale del sistema

Tuttavia però la procedura di **polling**, quindi di inoltro dell'autorizzazione a parlare, introduce un **overhead** sul sistema considerevole e, inoltre, se il nodo principale si guasta, l'intero sistema collassa. Una secondo tipo di protocollo a rotazione, detto **protocollo token-passing**, annulla la necessità di una centralizzazione dell'arbitro. In questo caso non esiste un nodo principale, ma viene usato un messaggio di controllo, detto **token**, che funge da autorizzazione a parlare e circola tra i nodi seguendo un ordine ben prefissato. Nonostante rispetto al **polling** si ottengano dei vantaggi dovuti alla decentralizzazione, anche in questo caso si presentano delle problematiche:

- Un guasto relativo al nodo con il token manda l'intero sistema in collasso.
- Se un nodo si "dimentica" di mandare il token è necessario invocare delle procedure per rimetterlo in circolazione.

# Capitolo 13

## Reti locali commutate

Supponiamo di avere una rete di un'organizzazione, quindi una rete che interconnette diversi dipartimenti, ognuno con dei dispositivi al loro intero. Una **rete locale commutata** è una rete locale che opera mediante switch connessi tra di loro attraverso varie tecnologie, una delle più diffuse è quella chiamata **Ethernet**. La struttura generale di una rete locale è:

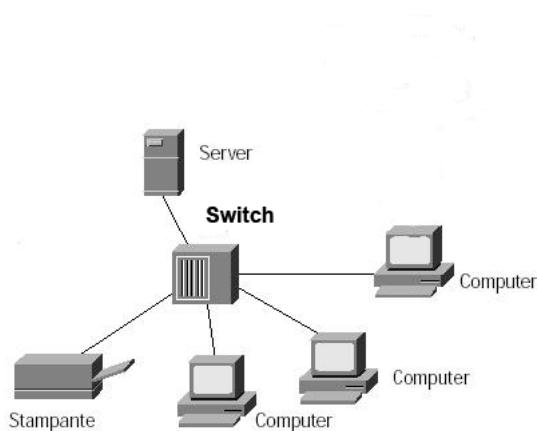


Figura 13.1: Esempio della struttura di una **LAN**

A loro volta, gli switch, lavorando a livello 2 dello stack protocollare, di conseguenza non riconosceranno e non utilizzeranno indirizzi a livello di rete per scambiare informazioni tra i vari host. La mancanza di un'identificazione dei nodi crea un problema in termini di riservatezza, poiché l'unica comunicazione possibile diventa quella broadcast e, quindi, un ipotetico messaggio privato rischierebbe di essere ascoltato da tutti i dispositivi. Affinché quindi sia possibile commutare i pacchetti tra host diversi senza dover ricorrere a una comunicazione broadcast è necessario definire uno schema di indirizzamento basato su un principio di identificazione dei nodi della rete. A livello due dello stack protocollare si utilizzano degli indirizzi univoci noti come **indirizzi MAC**. Ogni adattatore di rete possiede un indirizzo, l'**indirizzo MAC**, univocamente determinato dal produttore al momento della creazione della scheda; questo indirizzo è costituito da 6 byte (così da avere  $2^{48}$  possibili combinazioni) in esadecimale scritti nella seguente forma

$$\text{Indirizzo MAC} \longleftrightarrow 88\text{-}B2\text{-}2F\text{-}54\text{-}1A\text{-}0F \quad (1)$$

L'indirizzo MAC di una scheda di rete ha una struttura piatta e invariabile nel tempo; si potrebbe dire che l'indirizzo **MAC** è come una sorta di **codice fiscale** della scheda di rete che collega il dispositivo a **internet**. Bisogna quindi prestare attenzione alla differenza tra **indirizzi MAC** e **indirizzi IP**: mentre i primi corrispondono al **codice fiscale**, i secondi corrispondono all'**indirizzo postale** e, quindi, possono variare nel tempo e hanno una struttura gerarchica. L'indirizzo MAC

permette quindi di identificare un dispositivo nella rete locale, mentre l'indirizzo IP permette di identificare un dispositivo nella rete globale. Come risulta quindi facile immaginare, indirizzi IP e indirizzi MAC coesistono come due aspetti diversi di un qualunque dispositivo:

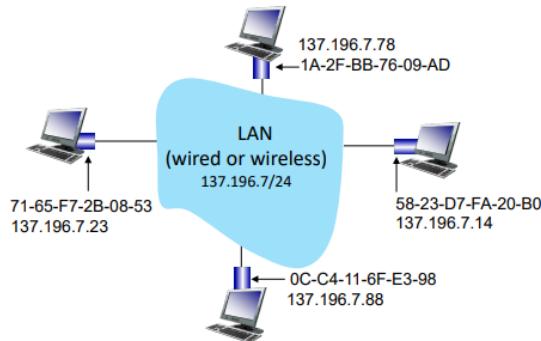


Figura 13.2: Enter Caption

Uno dei vantaggi principali dell'indirizzo MAC è data dalla sua portabilità, ogni dispositivo può essere univocamente identificato indipendentemente dalla rete locale in cui si trova.

## 1 Ethernet

Nel mondo delle reti cablate ethernet è quella che più di tutti è riuscita a prendere piede. Il motivo principale è dato dal fatto che ethernet è stata la prima tecnologia ad altà velocità e a vasta diffusione, rendendo quindi questa tecnologia utilizzatissima. Inoltre, ethernet è una tecnologia molto più semplice da utilizzare rispetto a connessioni cablate come **ATM**, **token-ring** e **FDDI**. Ethernet definisce anche un formato per i suoi frame, costituito da 6 campi e così strutturato



Figura 13.3: Formato di un messaggio ethernet

Il campo dati contiene l'effettivo **datagramma IP**. Ethernet definisce una lunghezza massima per questo campo, detta **unità massima di trasmissione**, pari a circa 1500 byte; se il datagramma supera questa dimensione dovrà essere frammentato in frame più piccoli, mentre, se il datagramma ha dimensione inferiore all'MTU (48 byte) dovrà avvenire un riempimento dei bit mancati. Il secondo campo, detto **indirizzo di destinazione**, contiene l'indirizzo MAC della scheda di rete di destinazione. Il terzo campo, quello dell'**indirizzo sorgente**, contiene l'indirizzo **MAC** del mittente del frame. Il campo **tipo** contiene informazioni sul tipo di protocollo a livello di rete utilizzato. Originariamente infatti esistevano diversi protocolli a livello di rete che potevano essere utilizzati, ad oggi, invece, l'unico protocollo in uso è il protocollo IP. Successivamente troviamo il campo relativo al controllo di ridondanza ciclica, il quale permette al destinatario di rilevare eventuali errori di trasmissione. Esiste anche un campo particolare, noto come **preambolo**, costituito da 8 byte; in particolare:

- I primi 7 byte permettono di risvegliare le schede di rete dei riceventi e di sincronizzare il loro clock con quello del trasmittente. La struttura dei primi 7 byte è 10101010.
- L'ultimo byte del preambolo avvisa il ricevente che i dati sono in arrivo. La struttura dell'ultimo byte è 10101011.

La dimensione complessiva di un frame ethernet può quindi variare da [64, 1518] byte. Le tecnologie basate su ethernet hanno il vantaggio di essere economicamente vantaggiose, ma non garantiscono

un servizio affidabile a livello di rete. Quando la scheda di rete B riceve un frame da A, lo sottopone a un controllo a ridondanza ciclica, ma non invia alcun **acknowledgment** ad A sull'accettazione o meno del frame a livello superiore; nel caso il controllo a ridondanza ciclica non sia stato passato, il frame viene solamente scartato. Quindi, A non sa se il frame trasmesso abbia superato il controllo CRC o meno e si limiterà quindi a inviare il frame successivo. Questo fatto si traduce nella conseguenza che il flusso di datagrammi che attraversano il livello di rete potrebbe presentare delle lacune.

Inoltre, ethernet utilizza il protocollo **CSMA/CD** per la trasmissione in broadcast del segnale. Questo algoritmo funziona in modo analogo a quanto detto nel precedente capitolo:

- La scheda diretta crea i frame.
- Se la scheda di rete rileva che il mezzo è libero per 96 bit, inizia la trasmissione.
- Se invece la scheda di rete rileva che il mezzo non è libero, allora attende altri 96 bit.
- Se la scheda di rete trasmette l'intero frame senza rilevare un'ulteriore trasmissione allora la scheda di rete ha terminato.
- Se la scheda di rete rileva un'altra comunicazione in corso allora abortisce la sua comunicazione e manda un **48-bit jam signal**.
- A questo punto si applica l'algoritmo di **exponential backoff** per decidere l'attesa. Dopo 17 tentativi falliti il frame verrà distrutto.

Per trasmettere 96 bit ci vogliono circa  $9.8 \mu s$ . Il segnale **48-bit jam signal** è un segnale che viene inviato a una potenza maggiore, consentendo agli altri nodi di individuare la collisione nel momento in cui si verifica.

**Come mai la dimensione minima del campo dati è 48 byte ?** La motivazione deriva dalla necessità di poter, per un mittente, rilevare una ipotetica collisione sul mezzo condiviso prima di smettere di ascoltare il canale. Supponiamo di avere uno standard ethernet basato su cavo coassiale e supponiamo che il cavo possa avere una lunghezza di 250 metri. Ipotizziamo che il nodo **A** voglia spedire un frame sul cavo. Idealmente, per il meccanismo di **CSMA/CD**, il mittente dovrebbe essere in grado di rilevare una collisione, dovrebbe cioè rimanere in ascolto sul canale per un tempo sufficientemente grande. Lo slot time, cioè il tempo che un segnale impiega per arrivare dall'altra parte del cavo è fissato a  $51.2 \mu s$ , quindi, ipotizzando di avere una velocità di 10Megabit/s

$$10\text{Mbit/s} \cdot 51.2\mu\text{s} = 512\text{bit} = 64\text{byte}$$

Se inviasse meno di 64byte sulla rete non sarebbe in grado di rilevare un eventuale collisione con un altro dispositivo che a sua volta ha iniziato a spedire.

## 2 Comunicazione nelle LAN

Affinché sia possibile comunicare per una scheda di rete è obbligatorio per essa inserire l'indirizzo MAC di destinazione all'interno del datagramma che verrà poi passato sul canale fisico; in particolare:

- Se la scheda di rete vuole inviare il messaggio in broadcast dovrà inserire all'interno del datagramma un indirizzo MAC speciale, detto **indirizzo MAC broadcast**. Tale indirizzo è semplicemente strutturato come una sequenza di tutti 1.
- Se la scheda di rete vuole inviare il datagramma a un host specifico dovrà inserire il suo **MAC address** all'interno del datagramma.

Occasionalmente uno switch potrebbe scegliere di inviare comunque il datagramma in broadcast a tutte le sue interfacce di uscita e, in quel caso, sarà il compito di chi le riceve analizzare se il datagramma era diretto alla sua scheda.

### 3 Switch a livello di collegamento

Lo switch è un componente fondamentale all'interno delle reti locali. A differenza di un semplice **hub**, che si limita a prendere i bit che gli arrivano e a ripeterli verso tutte le sue uscite ignorando la presenza di indirizzi MAC o altro, uno switch offre tutta una serie di funzionalità che insieme garantiscono una comunicazione affidabile e sicura:

- **Inoltro e filtraggio** dei frame. La seconda funzionalità permette allo switch di determinare se un pacchetto debba essere scartato o meno. La prima funzione, quella dell'inoltro, consiste nell'individuazione dell'interfaccia verso cui il frame deve essere instradato.
- **Rilevazione e gestione** delle collisioni attraverso il protocollo di accesso multiplo **CSMA/CD**

Notiamo immediatamente una differenza sostanziale tra **hub** e **switch**: mentre il primo lavora a livello meramente fisico, il secondo lavora a livello di collegamento, potendo quindi accedere a indirizzi MAC. Lo switch è quindi in grado di fare un **inoltro selettivo** verso l'uscita associata a quell'indirizzo MAC, senza spedire il pacchetto a tutti i dispositivi. Lo switch è, inoltre, **trasparente** ai nodi; quando un nodo vuole comunicare con un altro nodo non inserirà l'indirizzo dello switch, ma quello del mittente. Nonostante quindi **hub** e **switch** siano dispositivi estremamente diversi tra di loro, un nodo connesso alla rete non saprà a quale dei due dispositivi è connesso. Supponiamo di avere la seguente situazione:

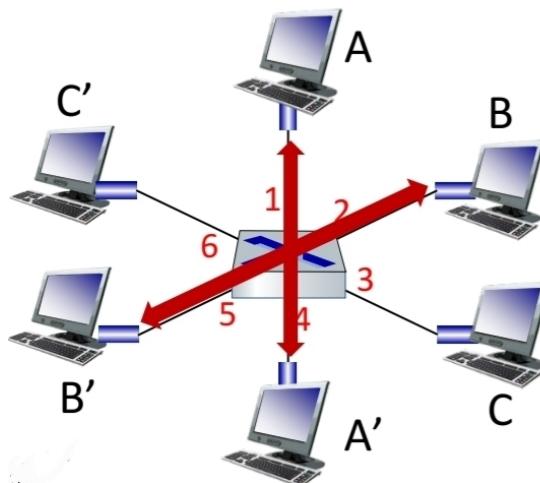


Figura 13.4: Rete locale con 6 host connessi attraverso uno switch

Uno dei problemi centrali che uno switch deve risolvere è quello dell'associare ad ogni linea d'uscita l'indirizzo MAC del dispositivo che vi si trova connesso. Idealmente ci piacerebbe che lo switch fosse in grado di creare queste associazioni in modo automatico e autonomo, senza bisogno di un intervento esterno di un utente o di un amministratore di rete. Per farlo dobbiamo prima definire una struttura dati interna allo switch, detta **MAC table**, costituita da tre campi: **indirizzo MAC**, **interfaccia d'uscita** e **timestamp**.

Indirizzo MAC	Interfaccia d'uscita	timestamp
01-12-24-21-43-12	2	9:39
04-1A-BC-26-12-54	1	9:32

Tabella 13.1: MAC table di uno switch. Il parametro **timestamp** identifica l'ora in cui è stato inserito il record. Ogni record ha un intervallo di tempo, detto **aging time**, in cui se non invia frame, lo switch cancella il record dalla tabella

Una delle proprietà fondamentali dello switch è proprio la capacità di **autoapprendimento**. Attraverso un semplicissimo algoritmo lo switch è in grado di popolare la propria tabella MAC. Supponiamo che all'istante  $t = 0$  la tabella MAC dello switch sia vuota e supponiamo che un nodo  $A$  voglia comunicare con un nodo  $C$ . Allora si eseguono i seguenti due passi:

- Il nodo  $A$  prepara il frame, specificando l'indirizzo MAC del destinatario, e lo invia.
- Lo switch aggiunge il record relativo al mittente all'interno della propria MAC table.
- Lo switch controlla il campo destinatario del frame. Se trova una corrispondenza con un record della tabella invia quel frame verso la linea d'uscita relativa. Altrimenti invia il pacchetto in broadcast.

La risposta del mittente al messaggio seguirà la medesima logica. In questo modo abbiamo un dispositivo che oltre a autoapprendere è anche **plug-and-play**. Basta semplicemente collegare i segmenti di LAN alle sue interfacce, senza dover configurare le tabelle al momento dell'installazione e senza bisogno di interventi esterni da utenti o amministratore di rete. I vantaggi dell'utilizzo degli switch a livello di collegamento sono innumerevoli, primo tra tutti l'eliminazione delle collisioni. In una LAN costituita da switch e senza uso di hub, i frame, vengono memorizzati all'interno di un buffer dello switch e trasmessi non più di uno per istante di tempo. Inoltre, l'utilizzo di una LAN basata su switch permette di avere dispositivi di natura diversi e che usano mezzi fisici diversi connessi tutti insieme senza alcun problema derivato dalla differenza dei segnali.

Supponiamo ora avere una rete istituzionale, costituita quindi da diversi dipartimenti, ognuno con una propria LAN. Gli switch che collegano gli host di un dipartimento sono a loro volta collegate a un unico switch, in una struttura gerarchica ad albero:

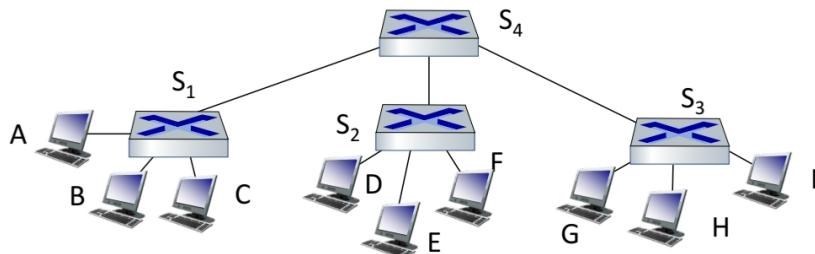


Figura 13.5: Rete istituzionale. I diversi dipartimenti sono connessi attraverso degli switch che, a loro volta, sono connessi ad uno switch di livello più alto

La comunicazione, in questo caso, diventa leggermente più complessa. Supponiamo che all'istante  $t = 0$  tutte le tabelle degli switch siano vuote e, inoltre, che il nodo  $A$  voglia comunicare con il nodo  $G$ . A questo punto quello che accade è analogo a quanto succede nel caso di un solo switch:

- Il frame arriva allo switch che memorizza il MAC del mittente e l'interfaccia di uscita.
- Non trovando alcuna corrispondenza per il MAC del destinatario, lo switch, invierà il pacchetto in broadcast.

Il pacchetto arriverà quindi a  $S_4$  che, non trovando ancora nessuna corrispondenza per il mittente, inoltrerà nuovamente i pacchetti in broadcast a tutti gli switch. In ultima battuta il pacchetto arriverà anche ad  $S_3$  e, infine, al destinatario.

## 4 LAN virtuali

Nella sezione scorsa abbiamo visto come le reti istituzionali moderne siano organizzate secondo una scala gerarchica, così che ogni gruppo abbia la propria LAN commutata connessa alle LAN commutate degli altri gruppi attraverso un albero di switch. Tale struttura però presenta alcuni problemi non indifferenti

- **Mancanza di isolamento del traffico.** Nonostante il traffico sia localizzato all'interno di un gruppo o di un solo switch, tutto il traffico broadcast dovrà attraversare l'intera rete istituzionale (**singolo dominio di broadcast**). Inoltre, se un messaggio fosse riservato ad un particolare settore, ma fosse indirizzato ad un indirizzo non ancora registrato, il messaggio verrebbe inviato comunque nel dominio di broadcast, esponendo informazioni potenzialmente riservate.
- **Uso inefficiente degli switch.** All'aumentare dei gruppi aumenta il numero di host e, se ogni gruppo ha un numero ristretto di utente, si corre il rischio di non utilizzare tutte le interfacce di uno o più switch.
- **Gestione degli utenti.** Se un dipendente dovesse muoversi tra più gruppi sarebbe necessario cambiare la posatura della rete per connetterlo ad un altro switch.

Tali problemi possono essere risolti utilizzando un unico switch che supporti una **virtual local area network**. L'idea è che questo switch, detto switch VLAN, permetta di definire più reti locali virtuali su una singola infrastruttura fisica di rete locale. Gli host all'interno di una **VLAN** comunicano tra loro come se fossero tutti connessi ad uno switch privato. In una VLAN basata sulle porte le interfacce dello switch devono essere divise dal gestore di rete in gruppi, ognuno dei quali costituisce una VLAN e le cui porte formano un dominio di broadcast ristretto. In questo modo risolviamo due problemi

- **Isolamento del traffico.** I pacchetti di ogni gruppo sono isolati dai pacchetti di un altro gruppo. Questo fatto, oltre a ridurre il traffico di rete, aumenta anche la sicurezza delle comunicazioni
- **Scalabilità.** Se un dispositivo volesse connettersi alla VLAN del dipartimento di informatica sarebbe sufficiente, per il gestore della rete, configurare la porta su cui viene connesso il nuovo dispositivo come appartenente alla VLAN del gruppo scelto.

Tuttavia si viene a creare un nuovo problema: isolando le due **VLAN** diventa estremamente complesso riuscire a inviare dei pacchetti tra i vari gruppi. Una possibile soluzione sarebbe quella di connettere una delle porte della **VLAN** a un router esterno, specificando che egli appartiene a tutte le **VLAN**. Un frame che deve essere inviato tra i due dipartimenti salirebbe di un livello, diventando un datagramma IP, per poi essere elaborato dal router, il quale lo instradarebbe verso la **VLAN** di destinazione. Supponiamo ora che i dipartimenti siano ospitati su edifici diversi dove debba essere fornito accesso alla rete e che, ovviamente, quest'ultimo possa far parte della **VLAN** del dipartimento. Ovviamente i nodi saranno posti anche su switch diversi e sarà dunque necessario che questi due switch possano scambiarsi informazioni. L'approccio utilizzato è quello detto **VLAN trunking**; in questo approccio si utilizza una porta speciale per ogni switch che viene configurata come porta di trunking per connettere due VLAN switch. La porta di trunking appartiene a tutte le VLAN e i frame inviati a qualunque VLAN vengono inoltrati attraverso il collegamento di trunking all'altro switch. I frame che transitano attraverso la porta di trunking hanno un formato particolare, definito dallo standard **802.1Q**

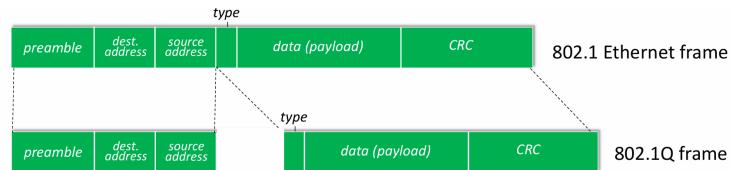


Figura 13.6: Struttura di un frame di trunking secondo lo standard 802.1Q

La forma del pacchetto è del tutto analoga a quella dei frame ethernet con l'aggiunta di un campo, detto **etichetta VLAN**, che codifica la VLAN di appartenenza del frame. Questo campo viene aggiunto dallo switch lato mittente e decodificato dallo switch lato destinatario.

**Interconnessioni di VLAN** Come abbiamo appena detto all'interno della sezione, l'interconnessione di VLAN è un problema concreto derivato dalla natura stessa delle VLAN. Esistono diverse possibili soluzioni per gestire questa problematica, nel nostro caso approfondiremo il metodo del **router-on-a-stick**. L'idea è quella di collegare un singolo cavo tra **switch** e **router**, configurandolo come **trunk** (solitamente usando il protocollo 802.1Q), su cui passano i frame di tutte le VLAN etichettati. Il router gestisce questo traffico tramite delle **sotto-interfacce** logiche (una per ogni VLAN). In particolare, il compito del router è quello di fare **routing IP**: riceve il pacchetto dalla sotto-interfaccia della VLAN mittente, rimuove il tag, analizza l'indirizzo IP di destinazione e, se necessario, re-incapsula il pacchetto con il nuovo tag della VLAN di destinazione, inoltrandolo sulla sotto-interfaccia corrispondente."

## 5 WAN basate su switch

Fino ad ora abbiamo considerato reti che, tutto sommato, avevano una dimensione limitata, massimo una rete di una istituzione. Possiamo però provare a estendere la dimensione della rete a ordini di grandezza più grandi, con un numero molto maggiore di host connessi insieme; queste reti entrano sotto al nome più generale di **Wide Area Network**.

La prima idea che possiamo analizzare è quella di suddividere la struttura della nostra rete in una struttura ad albero, con degli switch ordinati per livello e posti in una struttura gerarchica. Questa struttura presenta però dei problemi non indifferenti, primo tra tutti, introduce una bassissima tolleranza ai guasti. Se infatti uno switch di alto livello si guastasse, una parte dell'intera rete verrebbe isolata dalle altre. Affinché sia possibile risolvere questo problema è necessario utilizzare un'infrastruttura a **mesh**:

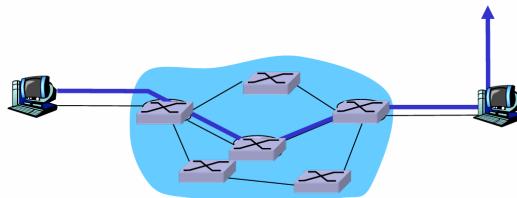


Figura 13.7: Struttura di una *Wide-Area-Network*

In questo modo è possibile risolvere il primo problema relativo ai guasti. Questa struttura introduce il problema del routing, quindi quello della scelta del percorso ottimale da un nodo *A* ipotetico a un nodo *B* ipotetico. A tal proposito occorre distinguere le due tipologie di connessioni:

- **Connectionless**: i pacchetti sono inviati senza la creazione di un canale virtuale prima. Ogni pacchetto segue quindi un percorso diverso e, per tale motivo, non è detto che giungano ordinati al destinatario.
- **Connection-oriented**: prima dell'inizio del trasferimento, mittente e destinatario, creano un **canale virtuale** diretto. Tutti i frame scambiati seguiranno lo stesso percorso e, quindi, giungeranno al mittente ordinati.

Un canale virtuale è, in termini astratti, analogo a una rete a commutazione di circuito. Una volta stabilito un canale virtuale i due host sono connessi da un canale diretto e invariabile per tutta la durata della connessione. Affinché però sia possibile creare un canale virtuale è necessario che i due host operino una sorta di handshake. Un canale virtuale è quindi una vera e propria struttura definita da una serie di collegamenti punto a punto tra host e switch definita come:

- Un percorso da mittente a destinatario.
- Un numero identificativo per ogni collegamento punto a punto.

Inoltre, ogni switch deve avere al suo interno una tabella in cui le informazioni su questi canali virtuali sono memorizzati. Questa tabella, nota come **forwarding table**, contiene un'entrata per ogni **canale virtuale** che coinvolge lo switch ed ha una struttura del tipo

Incoming interface	Incoming VC#	Outgoing interface	Outgoing VC#
1	12	3	22
2	63	1	18

Tabella 13.2: Forwarding table di uno switch. Ogni entrata corrisponde a un segmento di un canale virtuale in cui lo switch è coinvolto.

Nel caso in cui la comunicazione sia di tipo **connectioless** non vi è alcun canale virtuale e i pacchetti vengono indirizzati direttamente senza preoccuparsi del percorso che seguiranno. La tabella di forwarding è dunque estremamente semplificata e contiene il MAC address del destinatario, la linea d'uscita e il **TTL** del pacchetto.

# Parte IV

## Livello di rete

# Capitolo 14

## Struttura e servizi del livello di rete

Immediatamente sopra al livello di collegamento troviamo il livello di rete. Abbiamo visto come la connessione attraverso switch permetta di creare delle reti locali, reti istituzionali e anche reti di grandi dimensione. Supponiamo ora di avere una *rete di reti*, quindi di avere diverse reti che devono essere interconnesse tra di loro. Ci rendiamo conto che la sola presenza degli switch non è sufficiente a reggere una struttura del genere. Introduciamo quindi altri elementi, detti **router**. Così facendo possiamo interconnettere diverse reti tra di loro in modo facile:

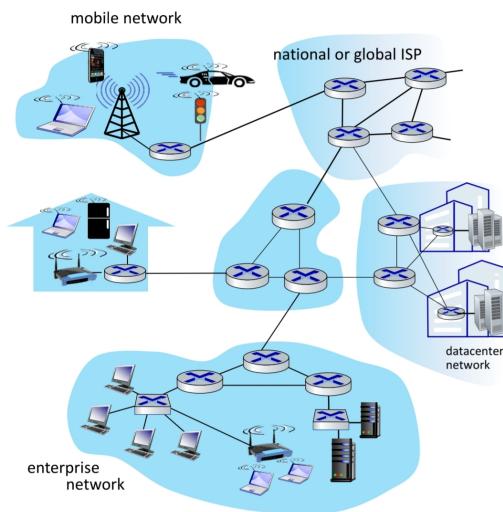


Figura 14.1: Rete complessa. Si nota come attraverso una serie di router sia possibile interconnettere in modo molto semplice reti diversi

Il livello di rete prende segmenti dal livello di collegamento, li incapsula in un datagramma, cioè in un pacchetto a livello di rete e, infine, li instrada verso il livello sottostante. Dal lato ricevente, invece, il percorso è opposto: il livello di rete riceve dal livello di collegamento dei frame, che spacchetta in datagrammi e passa al livello di collegamento. Il componente centrale del livello di rete è il router. Il router è un dispositivo intermedio che ha il compito di connettere più reti. In particolare, il compito di un router, è quello di prendere dei datagrammi IP, analizzare l'intestazione per determinare l'indirizzo IP di destinazione e inoltrare il datagramma verso una porta di uscita corrispondente all'indirizzo destinazione. Affinché ciò sia possibile è necessario che tutti i router implementino due meccanismi fondamentali:

- **inoltro** (o **forwarding**): Quando il router riceve un pacchetto deve inoltrarlo sul collegamento in uscita più appropriato.

- **instradamento** (o **routing**): Quando il router riceve un pacchetto deve determinare, attraverso specifici algoritmi detti **algoritmi di routing**, il percorso migliore su cui instradare il pacchetto.

occorre prestare attenzione alla differenza tra i due termini, con **inoltro** si intende all'azione locale con cui il router trasferisce i pacchetto da un'interfaccia di ingresso a un'interfaccia di uscita, mentre con **instradamento** intendiamo, invece, il processo di rete che determina i percorsi dei pacchetti nel loro viaggio da sorgente a destinazione. Per fare un'analogia con la realtà, possiamo dire che il **routing** è la determinazione dell'intero tragitto per arrivare, ad esempio, da Roma a Milano, mentre il **forwarding**, corrisponde alla scelta dell'uscita delle rotonde che si troveranno lungo il percorso. Questa differenza tra **routing** e **forwarding** permette di distinguere due piani di funzionamento diversi del router:

- Il piano di dati, che corrisponde alla funzione dell'**inoltro**, determina come i datagrammi in arrivo al router saranno mandati in uscita. Questo piano sfrutta una struttura dati, chiamata **tabella di inoltro**, che contiene le associazioni tra indirizzi e porte di uscita.
- Il piano di controllo, che corrisponde alla funzione del **routing**, ha come obiettivo quello di trovare il tragitto più breve verso una qualsiasi destinazione, estrarre la porta d'uscita per quella direzione e salvarla in una struttura dati utilizzata dal piano dati. Esistono due approcci principali per realizzare il piano di controllo
  - Nella prima modalità si utilizzano degli algoritmi di routing. L'idea è che i router, scambiandosi informazioni, riescano, ognuno, a creare la propria tabella di inoltro e il proprio grafo della rete. Le informazioni scambiate da ogni router riguardano principalmente i propri vicini.
  - Un secondo approccio, detto **Software Defined**, sfrutta un approccio centralizzato. Un server, posto in un cloud, calcola tutti i percorsi ottimi per arrivare da una sorgente a una destinazione.

Il piano di controllo ha quindi quello di stabilire il percorso, lasciando poi al piano di dati il compito di seguirlo.

I router, nel piano di controllo, estraggono dal campo intestazione il valore che utilizzano come indice nella **tabella di inoltro**. Il risultato indica su quale interfaccia di uscita instradare il pacchetto. La tabella di inoltro viene invece popolata dal piano di controllo del router attraverso uno dei due approcci.

## 1 Modelli di servizio

Nello studio del livello di rete è anche utile soffermarsi su quali servizi sono offerti o, comunque, garantiti. Il **modello di servizio** definisce le caratteristiche di trasporto *end-to-end* di dati tra sistemi periferici di invio e di ricezione. Alcuni esempi di servizi che potrebbero essere richiesti per la trasmissione sono:

- **Consegna garantita con ritardo limitato.** Questo servizio assicura che il pacchetto giunga a destinazione entro un certo intervallo di tempo.
- **Consegna ordinata.** Questo servizio garantisce che i pacchetti giungano alla destinazione nell'ordine con cui sono stati inviati.
- **Banda minima garantita.** Questo servizio garantisce che ci sia un **troughput** minimo per il canale trasmittivo.
- **Servizi di sicurezza.** Questo servizio che la comunicazione avvenga in modo sicuro. In particolare, si potrebbe richiedere la comunicazione avvenga in modo criptato utilizzando un sistema a chiave, etc.

Il livello di rete mette a disposizione un solo servizio, noto come **best-effort**, ossia *con il massimo impegno possibile*. Il servizio **best-effort** non garantisce nessuno di questi servizi, né che si tratti di sicurezza, né che si tratti di banda minima garantita; cerca di garantirli al meglio delle sue possibilità. Il **best-effort** non è tuttavia l'unico modello di servizio esistente, architetture di rete diverse da quella internet hanno modelli di servizio diversi che offrono anche servizi

Architettura di rete	Modello di servizio	Garanzia sulla banda	Garanzia consegna	di	Indicazione di congestione
Internet	Best-effort	Nessuna	Nessuna		No
ATM	CBR	Tasso costante garantito	Sì		Non si verifica congestione
ATM	ABR	Minimo garantito	Nessuna		Sì

Tabella 14.1: Confronto tra modelli di servizio e architetture di rete

Il motivo per cui il **best-effort** è diventato il modello di servizio più in uso dipende dal fatto la banda dei dispositivi intermedi è molto alta. Questo fatto implica che non è così fondamentale che siano garantiti tutti questi servizi, poiché è implicito che, con una banda alta, questi servizi siano già garantiti. Inoltre, il meccanismo **best-effort** è estremamente semplice e funziona bene fintanto che la banda è sufficiente.

## 2 Struttura del router

Un router è, sostanzialmente, un computer dotato di: porte di ingresso e di uscita, di una rete di commutazione che permette il reindirizzamento di un pacchetto tra una porta di ingresso e una di uscita, e da una serie di algoritmi che implementano diverse funzionalità e servizi, tra cui l'**inoltro** e l'**instradamento**. Funzionalmente possiamo rappresentare la struttura di un router nel seguente modo:

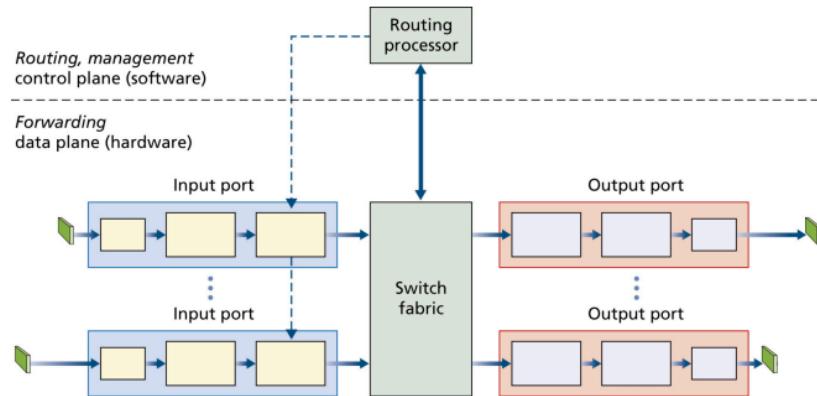


Figura 14.2: Visione funzionale della struttura interna di un router

Una porta di input ha il compito di eseguire diverse funzionalità. Le prime due funzionalità riguardano i livelli sottostanti dello stack, infatti la porta di ingresso svolge le funzioni a livello fisico di terminazione di un collegamento in ingresso al router e, inoltre, svolge anche tutte quelle funzioni a livello di collegamento necessarie per inter-operare con le analoghe funzioni dall'altro lato del collegamento. Insieme a queste funzionalità, ogni porta di input, implementa anche le funzionalità di **queuing** dei pacchetti. In particolare, ogni porta ha un suo buffer interno in cui i pacchetti vengono accodati e gestiti da un microprocessore dedicato. Una volta che un pacchetto giunge in cima al buffer sarà opportunamente gestito, attraverso un analisi dell'indirizzo mittente

contenuto nell'header, e instradato verso la destinazione opportuna. Le tecniche di **forwarding** sono in realtà molteplici:

- **forwarding basato sull'indirizzo di destinazione.** Questa è la tipologia di **forwarding** che abbiamo utilizzato implicitamente fino ad ora. Questa tecnica si basa sul controllo del solo indirizzo IP del destinatario.
- **forwarding generalizzato.** In questa forma di **forwarding** oltre all'indirizzo IP del destinatario, viene controllato anche l'**indirizzo di origine** e la **porta**.

Per tutto il corso faremo comunque riferimento alla prima tecnica di **forwarding**. Adottando questa tecnica la tabella di forwarding ha una struttura abbastanza semplice, in cui per ogni interfaccia d'uscita associamo l'intervallo degli indirizzi che si trova su quell'interfaccia di uscita:

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000	through
11001000 00010111 00010111 11111111	
11001000 00010111 00011000 00000000	through
11001000 00010111 00011000 11111111	
11001000 00010111 00011001 00000000	through
11001000 00010111 00011111 11111111	
<i>Otherwise</i>	3

Tabella 14.2: Esempio di tabella di forwarding di un router

Ogni intervallo può essere riscritto considerando solo il prefisso, quindi l'insieme dei bit che sono comuni a tutti gli indirizzi di quell'intervallo. In questo modo si ottiene una tabella semplificata e più leggibile.

Prefix	Interface
11001000 00010111 00010	0
11001000 00010111 00011000	1
11001000 00010111 00011	2
<i>Otherwise</i>	3

Con questa struttura il router confronta un prefisso dell'indirizzo di destinazione del pacchetto con una riga della tabella; se c'è corrispondenza, il router inoltra il pacchetto al collegamento associato. Nel caso in cui si verifichino corrispondenze multiple, il router adotta la regola di corrispondenza a prefisso più lungo; in altre parole, viene determinata la corrispondenza più lunga all'interno della tabella e i pacchetti vengono inoltrati all'interfaccia di collegamento associata. Analogamente alle porte di ingresso troviamo anche le porte di uscita, il cui compito è memorizzare i pacchetti che provengono dalla struttura di commutazione e li trasmettono sul collegamento in uscita, operando le funzioni necessarie del livello di collegamento e fisico.

Il secondo componente, anch'esso centrale nella struttura del router, è la **rete di commutazione**. Il funzionamento di questa rete è relativamente semplice: commutare i pacchetti dalle interfacce di ingresso verso le interfacce di uscita. Esistono diverse tecnologie con cui è possibile realizzare una rete di commutazione:

- **Approccio basato su memoria:** Questo approccio era tipico dei primi router che erano, di fatto, dei calcolatori tradizionali e la cui commutazione tra porte di ingresso e di uscita veniva effettuata sotto il controllo diretto della CPU. Le porte di ingresso e di uscita funzionavano come normalissimi dispositivi di I/O. Quando sopraggiungeva un pacchetto, la porta di ingresso ne segnalava l'arrivo tramite una interruzione e quindi la copiava nella memoria del processore di instradamento, che procedeva a estrarre dall'intestazione l'indirizzo di destinazione. Quindi, individuava nella tabella di inoltro l'appropriata porta di uscita nel cui buffer copiava il pacchetto,

- **Approccio basato su BUS:** In questo approccio le porte di ingresso trasferiscono un pacchetto direttamente alle porte di uscita tramite un bus condiviso e senza intervento da parte del processore di instradamento. Questo viene tipicamente fatto aggiungendo una etichetta interna di commutazione al pacchetto che indica l'aperta locale di output alla quale il pacchetto deve essere trasferito quando viene messo sul bus. Una volta ricevuto dalla porta designata, l'intestazione verrà rimossa, poiché necessaria solo localmente.
- **Approccio basato su interconnessione:** Un modo per superare le problematiche introdotte dall'approccio a bus (che, per inciso, sono le stesse problematiche che abbiamo visto nelle reti **broadcast**) è possibile utilizzare una rete di interconnessione più sofisticata. Una matrice di commutazione è una rete di interconnessione che contiene  $2n$  bus che collegano  $n$  porte di ingresso a  $n$  porte di uscita. Ogni bus verticale interseca tutti i bus orizzontali a un punto di incrocio che può essere, in qualsiasi momento, aperto o chiuso dal controller della struttura di commutazione.
- **Approccio multistage:** In questo approccio abbiamo una rete di interconnessione locale costruita utilizzando degli switch posti in collegamento tra di loro. Questa rete di switch non definisce un percorso esatto da porta di ingresso a porta di uscita, permettendo anche di avere connessioni multiple in contemporanea. Questo approccio può essere ulteriormente scalato aumentando il numero di reti di interconnessioni esistenti, ognuna costituita da  $n^2$  switch e con una diversa topologia.

L'ultima componente intera al router è il **processore di instradamento**, il quale esegue protocolli di instradamento e gestisce le tabelle di inoltro e le informazioni sui collegamenti attivi.

Una delle problematiche interne al router riguarda l'elaborazione dei pacchetti in uscita. Come abbiamo detto in precedenza, i pacchetti vengono presi dalla memoria della porta di uscita e trasmessi sul collegamento uscente; questa fase però richiede alcuni passaggi intermedi non trascurabili

- Selezionare il pacchetto da togliere dalla coda.
- Togliere il pacchetto dalla coda di ingresso

Una delle problematiche principali che si possono formare sono le code. È infatti evidente che si possano formare delle code sia in ingresso che in uscita, è sufficiente che la velocità della rete di commutazione sia o molto maggiore o molto minore delle rispettive velocità delle porte. Se le code crescono, superando la dimensione massima del buffer, si può verificare una **perdita di pacchetti**. In assenza di sufficiente memoria è necessario decidere se scartare o meno un pacchetto. Nel caso si decida di scartarlo si applica una politica detta **drop-tail**, mentre se si decide di non scartarlo è necessario scegliere un criterio di eliminazione dei pacchetti. Un altro problema si verifica quando l'elemento che si trova in testa, per qualche motivo, non può essere instradato verso la propria interfaccia di uscita, causando un blocco della coda; questo problema viene anche detto **Head-Of-Line blocking**.

Nelle code in uscita, se vi sono più pacchetti accodati, è necessario che uno **schedulatore di pacchetti** stabilisca in quale ordine trasmetterli. La selezione può avvenire tra diversi algoritmi, alcuni esempi sono il **Fist-Come First-Served** o il **Round-Robin**. Tuttavia, generalmente, l'algoritmo adottato è quello detto **Weighted Fair Queuing**. Questo algoritmo ripartisce equamente il collegamento uscente tra diverse connessioni che hanno pacchetti accodati per la trasmissione.

### 3 Protocollo IP

Esistono due protocolli IP attualmente in uso, il primo, che studieremo durante questo corso, è il protocollo **IPv4**, mentre il secondo, ad uso oggi su internet, è il protocollo **IPv6**. Il protocollo **IPv4** definisce un formato per i datagrammi a livello di rete; un datagramma è definito secondo la seguente struttura

- **Numero di versione (4 bit).** Questi bit specificano la versione del protocollo IP del datagramma. Ad oggi esistono due versioni del protocollo **IP**, la versione **IPv4** e la versione

IPv6. Questo campo ha anche il compito di determinare, in base alla versione, come il livello di rete dovrà leggere il datagramma.

- **Lunghezza dell'intestazione (4 bit).** Dato che un datagramma **IPv4** può contenere un numero di variabile opzioni, questi 4 bit indicano dove iniziano effettivamente i dati del datagramma.
- **Tipi di servizio (8 bit).** I bit relativi al tipo di servizio sono stati inclusi nell'intestazione per distinguere diversi tipi di datagrammi. Infatti possono esistere tipologie diverse di datagrammi come, ad esempio, i datagrammi in tempo reale
- **Lunghezza del datagramma (16 bit).** Lunghezza del datagramma. Rappresenta la lunghezza totale del datagramma IP, intestazione più dati, misurata in byte.
- **Identificatore, flag, spiazzamento di Frammentazione (32 bit).** Questi tre campi hanno a che fare con la frammentazione.
- **Tempo di vita (8 bit).** Il campo **TTL** è stato incluso per assicurare che i datagrammi non restino in circolazione per sempre nella rete. Questo campo viene decrementato di un'unità ogni volta che viene elaborato da un router.
- **Protocollo (8 bit).** Questo campo viene utilizzato solo quando il datagramma raggiunge la destinazione finale. Il valore del campo indica lo specifico protocollo a livello di trasporto al quale vanno passati i dati del datagramma.
- **Checksum dell'intestazione (16 bit).** Consente ai router di rilevare gli errori sui bit nei datagrammi ricevuti.
- **Indirizzi IP sorgente e destinazione (64 bit).** Quando un host crea un datagramma, inserisce il proprio indirizzo IP nel campo indirizzo IP dell'origine e quella della destinazione nel campo indirizzo IP di destinazione.
- **Opzioni (0-40 byte).** Questi campi consentono di estendere l'intestazione IP.
- **Dati ( ).** Il campo dati contiene il segmento a livello di trasporto da consegnare alla destinazione. Questo campo

Un datagramma potrebbe avere, potenzialmente, una dimensione maggiore di quella di un frame al livello di trasporto. In particolare, se  $M$  corrisponde alla grandezza del datagramma IP, potrebbe verificarsi la condizione in cui

$$M > \text{Maximum Transmission Unit}$$

La massima unità di trasmissione è definita come la massima quantità di dati che un frame a livello di collegamento può trasformare. Inoltre, un altro problema nasce dal fatto che ogni tecnologia di collegamento ha una diversa **MTU**. La soluzione a questi problemi è quella di frammentare i dati del datagramma IP in due o più datagrammi IP più piccoli, detti **frammenti**, e quindi trasferirli sul collegamento. Questi frammenti vengono spediti all'interno della rete e vengono riassemblati soltanto alla destinazione, infatti far riassemlare i frammenti direttamente ai router avrebbe creato una complessità inutile, portando anche a perdere prestazioni. Per consentire la frammentazione dei datagrammi all'interno della struttura, in particolare dell'intestazione del datagramma IP, i campi:

- **Identificazione.** Questo campo corrisponde a un numero che viene associato al datagramma per identificarlo.
- **Flag.** All'interno di questo campo troviamo due flag: il primo specifica se il pacchetto è un frammento, mentre il secondo se è l'ultimo frammento del pacchetto.
- **Spiazzamento.** Questo campo specifica la posizione del primo byte del frammento all'interno del datagramma originale.

Un meccanismo di questo tipo crea la necessità per il destinatario di definire dei meccanismi per il riordino e la deframmentazione dei pacchetti. Inoltre, l'introduzione del meccanismo della frammentazione richiede che venga definito un **timeout** oltre il quale, se il pacchetto non è stato ancora riassemblato, viene scartato.

## 4 Indirizzamento IPv4

Ogni indirizzo all'interno di una rete è identificato da due indirizzi: il primo, l'indirizzo MAC, è stato già discusso nello scorso capitolo, mentre il secondo è un indirizzo di 32 bit, scritto in **notazione decimale puntata**, chiamato **indirizzo IP**. La differenza tra questi due indirizzi sta nel fatto che l'indirizzo MAC è assegnato, in modo centralizzato, da un'organizzazione mondiale ed è univoco per ogni **NIC**, invece, l'indirizzo IP viene assegnato dal provider della rete e non è univoco. Ogni indirizzo IP è costituito da due parti:

- La prima parte dell'indirizzo identifica la rete fisica su cui il nodo è collegato.
- La seconda parte identifica l'host specifico all'interno della rete fisica.

Quindi tutti i dispositivi appartenenti ad una stessa sottorete avranno la prima parte dell'indirizzo IP tutta uguale. Anche se, la definizione di sottorete definisce quest'ultima come l'insieme dei nodi che possono comunicare direttamente senza passare attraverso un router, quindi l'indirizzo IP trova relativamente poca utilità. Supponiamo di avere una interete costituita da diverse sottoreti connesse da un router

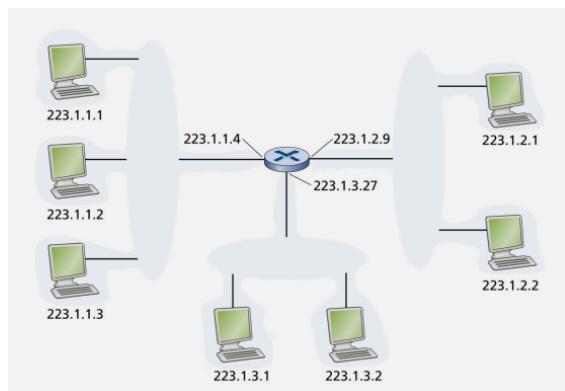


Figura 14.3: Esempio di rete con 3 sottoreti e un router centrale.

Dalla figura 14.3 notiamo immediatamente che ogni interfaccia del router è dotata di un diverso indirizzo IP; questo indirizzo viene chiamato **indirizzo di gateway**, cioè l'indirizzo a cui vengono spediti tutti i pacchetti relativi a host esterni alla rete. Per definire quanti bit erano relativi alla parte di rete era, in passato, stato definito un ordinamento gerarchico degli indirizzi IP, che prevedeva una suddivisione in 5 classi

Classe	Parte di rete	Parte di host
A	8 bit	24 bit
B	16 bit	16 bit
C	24 bit	8 bit

Tabella 14.3: Organizzazione gerarchica degli indirizzi IPv4

Mentre le prime tre classi furono effettivamente utilizzate per l'indirizzamento, le classi D ed E ebbero un ruolo limitato. La classe D era utilizzata per il multicast, mentre la classe E venne riservata per sviluppi futuri. Questo approccio venne però ben presto abbandonato a favore di

un approccio meno restrittivo e limitativo, che permettesse la configurazione dinamica delle reti. L'approccio moderno, non basato sulle classi, è detto CIDR (o Classless InterDomain Router). In questo approccio gli indirizzi sono sempre di 32 bit, ma la separazione tra le due parti diventa più generale e non definita a priori; in particolare:

- Diciamo che la parte più significativa identifica la sottorete di appartenenza dell'host.
- Diciamo che la parte meno significativa identifica l'host all'interno della rete.

Questo approccio, per quanto più flessibile, pone però un problema, cioè quello di identificare la sottorete di appartenenza di un host conoscendo l'indirizzo. A tal proposito, IP, assegna ad ogni sottorete anche una **maschera di sottorete**. Questa maschera indica quanti dei bit dell'indirizzo costituiscono la parte di rete; ad esempio: 192.168.1.2/24 ha una maschera di sottorete (/24) che specifica che i primi 24 bit dell'indirizzo costituiscono la parte di rete. Una maschera di rete può essere anche scritta in forma esplicita, ad esempio, nel caso di prima, la maschera di sottorete corrisponde a 255.255.255.0. Dato un indirizzo qualsiasi, per ricavare l'indirizzo della sottorete, è sufficiente fare l'**and bit a bit** tra l'indirizzo IP e la maschera di sottorete.

La presenza di questa forma di indirizzamento IP crea la necessità di assegnare ad ogni host di una rete un suo indirizzo. Affinché sia possibile fare questo assegnamento è necessario che, a priori, un amministratore di rete richieda che gli venga allocato un blocco di indirizzi dal suo ISP. L'ISP, che si trova, gerarchicamente parlando, al di sopra dell'amministratore dovrà selezionare un blocco di indirizzi dal suo pool di indirizzi, che a sua volta è allocato da un ISP di grado gerarchico maggiore. Al di sopra di questa catena gerarchica troviamo l'entità suprema, chiamata ICANN, che si occupa di assegnare i blocchi di indirizzi ai vari ISP di alto livello (che poi li assegneranno a quelli di più basso livello e così via). Una volta ottenuti gli indirizzi, si pone il problema di come suddividerli tra i vari host della rete; in particolare, distinguiamo due modalità possibili di assegnamento:

- **Assegnamento permanente dell'indirizzo IP.** L'amministratore di rete configura, attraverso i file di configurazione, l'indirizzo IP dell'interfaccia dell'host.
- **Assegnamento automatico dell'indirizzo IP.** Ogni qualvolta un host si connette alla rete, un particolare protocollo, si occupa di assegnare dinamicamente l'indirizzo IP agli host che lo richiedono.

Si nota immediatamente come la prima modalità di assegnamento sia certamente quella meno efficiente, poiché richiede un'attenzione costante dell'amministratore di rete e, soprattutto, che ogni host sia configurato manualmente. La seconda modalità è sicuramente quella più funzionale e efficiente, il compito di assegnare gli indirizzi IP viene assegnato a un protocollo (detto protocollo DHCP) che viene eseguito su un server particolare. L'unico intervento necessario dell'amministratore della rete è quello di abilitare questa funzionalità.

## 4.1 Protocollo DHCP

Uno dei problemi principali del protocollo IP è la necessità di assegnare ad ogni dispositivo un proprio indirizzo IP. Questa assegnazione, senza alcun protocollo che la velocizzi, risulterebbe estremamente lenta e macchinosa, per non parlare delle problematiche dal punto di vista funzionale. Per risolvere questo problema è stato introdotto un protocollo, detto **protocollo DHCP** (o **Dynamic Host Configuration Protocol**). Il protocollo DHCP è un protocollo di livello applicativo che ha il compito di assegnare un indirizzo IP e tutto il necessario affinché sia possibile connettersi, su richiesta, ad un host che si connette alla rete. DHCP è un protocollo **client-server**, dove il client è un host che si è appena connesso alla rete, ma senza un indirizzo IP specifico, mentre il server è un server DHCP (che ad oggi risiede direttamente sul router) vero e proprio. Per i nuovi host della rete, il protocollo DHCP si articola in quattro passi:

- **DHCP Discovery.** Il primo compito di un host appena collegato alla rete è l'identificazione del server DHCP con il quale interagire. Questa operazione è fatta utilizzando un messaggio di **DHCP discovery**, che il client invia in broadcast su tutta la sottorete. La struttura del messaggio avrà una forma del tipo:

```
src: 0.0.0.0, 68
dest: 255.255.255.255, 67
yiaddr: 0.0.0.0
transaction ID: 654
```

Il primo campo, **src**, è popolato da un indirizzo particolare che viene utilizzato dagli host che non hanno alcun indirizzo IP (sostanzialmente, è come se di dicesse *questo host*). Il secondo campo specifica che il messaggio deve essere inviato in broadcast.

- **DHCP Offer.** Un server DHCP, che riceve un messaggio di identificazione, risponde al client con un messaggio **DHCP Offer**. Questo messaggio viene comunque inviato in broadcast a tutti i dispositivi della sottorete, poiché ancora il mittente originale non possiede alcun indirizzo IP. Ciascun messaggio di offerta contiene l'**ID** della transazione del messaggio di identificazione ricevuto, l'indirizzo IP proposto al client (**yiaddr**), la maschera di sottorete e la durata della concessione dell'indirizzo IP

```
src: 223.1.2.5, 67
dest: 255.255.255.255, 68
yiaddr: 223.1.2.4
transaction ID: 654
lifetime: 3600s
```

Infatti la concessione del DHCP non è eterna, ogni indirizzo ha un certo intervallo di vita massimo oltre il quale non è più valido e occorre, per il client, richiederne uno nuovo.

- **DHCP Request.** Idealmente, il client si troverà a dover scegliere tra diverse offerte, provenienti da diversi server DHCP. Tuttavia la scelta non viene fatta secondo nessuna metrica particolare, solitamente viene scelto l'indirizzo dell'offerta arrivata per prima. Una volta scelto quale delle offerte accettare, il client, risponderà con un messaggio **DHCP Request**. La struttura di questo messaggio sarà:

```
src: 0.0.0.0, 68
dest: 255.255.255.255, 67
yiaddr: 223.1.2.4
trans ID: 655
lifetime: 3600 secs
```

- **DHCP Acknowledge.** Il server risponde al messaggio **DHCP Request** con un messaggio **DHCP Acknowledge**, che conferma i parametri richiesti. La struttura di questo messaggio sarà:

```
src: 223.1.2.5, 67
dest: 255.255.255.255, 68
yiaddr: 223.1.2.4
transaction ID: 655
lifetime: 3600 secs
```

Un fatto curioso che però può essere utile discutere è che, anche al momento dell'**ACK**, il messaggio viene inviato in broadcast. Il funzionamento del DHCP prevede infatti che l'assegnazione dell'indirizzo IP al client sia valida solo al momento dell'invio del **DHCP Acknowledge**, prima di questo momento l'indirizzo IP non è ancora assegnato ad alcun dispositivo. Un altro fatto interessante riguarda il **lifetime**, se un client volesse utilizzare l'indirizzo IP per un tempo maggiore della sua concessione, dovrebbe utilizzare dei meccanismi offerti dal protocollo IP per il rinnovo della concessione.

## 5 Network Address Translation

Uno dei problemi dell'approccio visto a questo momento è che il numero di dispositivi all'interno di una LAN non è - in effetti - costante. All'interno di una LAN i dispositivi possono aumentare e diminuire costantemente e ciò crea, nell'approccio visto fino ad ora, un grandissimo problema nell'assegnamento degli indirizzi IPv4. Infatti, se l'ISP dovesse allocare un blocco diverso per ogni sottorete dovrebbe anche tenere conto del fatto che il numero di dispositivi può variare, quindi tenere conto di un certo limite superiore di indirizzamento per quella LAN. Però in questo modo il numero di blocchi assegnabili diventa molto minore e, soprattutto, definire questo limite diventa estremamente complesso. La soluzione a questo problema prende il nome di **NAT** (o **Network Address Translation**). Supponiamo di avere una generica rete così strutturata

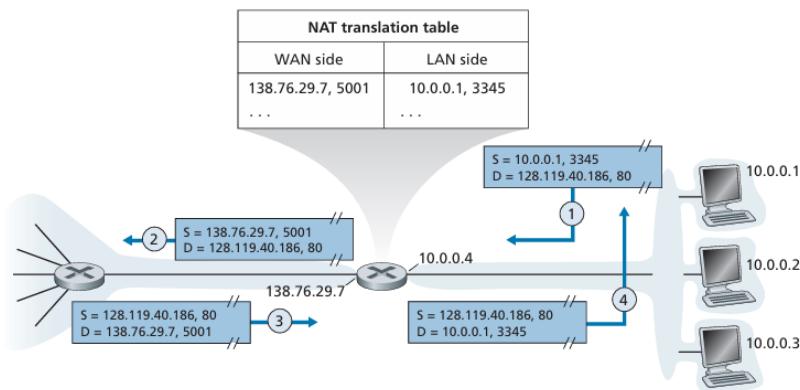


Figura 14.4: Esempio di router NAT.

L'idea del NAT è quella di suddividere gli indirizzi IPv4 in due classi: indirizzi **privati** e indirizzi **pubblici**. Un meccanismo simile lo possiamo trovare anche nella realtà, si pensi alla nostra rete come ad un condominio. L'indirizzo del condominio è analogo per tutti gli appartamenti che si trovano al suo interno, ma ogni appartamento è poi identificato da un numero detto **interno**. Gli indirizzi privati sono quindi gli **interni**, identificativi che hanno senso solo all'interno del "condominio", mentre gli indirizzi **pubblici** hanno senso per chiunque si trovi sulla rete, in sostanza è come se fossero **via** e **numero civico** del condominio. Nella rete presente nell'immagine lo spazio di indirizzamento 10.0.0.0 definisce un **realm**, con indirizzi **privati**, quindi indirizzi che hanno senso solo per i dispositivi che si trovano all'interno della rete privata. Mentre l'indirizzo 138.76.29.7 costituisce l'indirizzo **pubblico**, ed è quindi l'indirizzo utilizzato da qualunque altro dispositivo, esterno alla rete privata, che sia intenzionato a comunicare con essa.

I router abilitati al **NAT** non appaiono come router agli altri dispositivi, ma si comportano come un **unico** dispositivo con un **unico** indirizzo IP. Quindi un dispositivo esterno alla rete non avrà idea dell'esistenza della rete privata, ma inserirà l'indirizzo IP del router NAT. Il problema che si pone però è come sia possibile per il router nat capire a quale dei dispositivi instradare il pacchetto. La soluzione a questo problema è quella di utilizzare una **tabella di traduzione NAT**, rappresentata in figura 14.4, che associa ad ogni indirizzo **privato** della rete l'interfaccia di uscita. Analizziamo l'esempio in figura 14.4 ripercorrendo i passi fatti dal NAT:

- Ipotizziamo che un utente, interno alla rete privata, all'indirizzo 10.0.0.1, richieda una pagina web da un server con indirizzo IP 120.119.40.186. L'host che richiede la pagina sceglie una porta casuale tra quelle disponibili e la utilizza come porta sorgente, mentre alla sua porta destinataria assegna **80**
- Il router NAT riceve il datagramma, genera per esso un nuovo numero di porta di origine (es. 5001), sostituisce l'indirizzo IP sorgente con il proprio indirizzo e il numero di porta con il numero di porta scelto da lui e, infine, instrada il pacchetto verso il web server.

- Il web server, che è ignaro della presenza del NAT, genera il pacchetto di risposta HTTP inserendo nel datagramma indirizzo e numero di porta del router che ha mandato la richiesta.
- Il router NAT, una volta ricevuta la richiesta, consulta la tabella di traduzione utilizzando il numero di porta di e l'indirizzo IP di destinazione per trovare l'indirizzo IP e il numero di porta dell'host richiedente.
- Il router NAT, una volta trovato l'host richiedente, riscrive, nell'indirizzo IP e nel numero di porta del pacchetto di risposta, i dati dell'host che aveva fatto la richiesta e, infine, instrada il pacchetto.

Ovviamente, al momento in cui il router genererà un numero di porta da associare alla connessione con l'host, dovrà consultare la tabella di traduzione NAT affinché il numero non risulti duplicato per quell'indirizzo IP. Essendo il numero di porta di 16 bit, il protocollo NAT può supportare più di 60.000 connessioni simultanee con ogni indirizzo IP della rete privata. Uno dei problemi principali del NAT è che la connessione **host-to-host** diventa estremamente complicata.

### 5.1 Port-Forwarding

Supponiamo ora di voler introdurre un server all'interno della mia rete privata, che avrà quindi un indirizzo IP privato. Il problema diventa quindi che un qualsiasi host che vuole comunicare con quel server dovrà conoscere il suo indirizzo IP privato, cosa che però sappiamo non essere possibile. L'idea potrebbe essere quella del **NAT statico**, o **Port Forwarding**, cioè di inserire all'interno della NAT table un associazione permanente tra un particolare numero di porta un indirizzo IP. Ad esempio

$$\text{Qualsiasi indirizzo IP, 80} \rightarrow 10.0.0.2 \quad (1)$$

Una qualsiasi richiesta da un generico indirizzo IP sulla porta 80 sarà quindi rediretta verso l'indirizzo IP privato del server.

## 6 Indirizzamento IPv6

Nel momento in cui ci si rese conto che gli indirizzi IPv4 non sarebbero bastati, venne scelto di creare una nuova tipologia di indirizzamento IP, detto indirizzamento IPv6. La prima differenza risiede nel fatto che un indirizzo IPv6 è costituito da 128 bit invece che da 32, aumentando così il numero di indirizzi da  $2^{32}$  a  $2^{128}$ . Consequenzialmente alla grandezza dell'indirizzo, anche il formato del datagramma IPv6 è diverso; in particolare

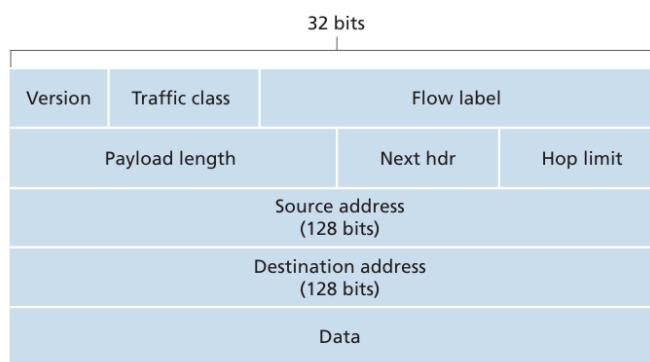


Figura 14.5: Formato del datagramma IPv6

Oltre all'aumento sostanziale della dimensione degli indirizzi, il protocollo IPv6 supporta anche indirizzi **anycast**, funzionalità che permette di inviare un pacchetto ad un host (generalmente il più vicino) all'interno di un particolare gruppo. Inoltre, alcuni campi dell'intestazione del datagramma IPv4 sono stati rimossi, permettendo di ottenere un'intestazione di lunghezza fissa (40 byte) e, quindi, più veloce da elaborare. Agli albori, internet non prevedeva l'esistenza tipi di traffico, ma

era pensato solo per il trasferimento di file. Il protocollo IPv6 permette invece di differenziare questi flussi di traffico, ad esempio, per differenziare i flussi di traffico video da quelli audio, da quelli relativi a file. Questi cambiamenti permettono quindi di ottenere un datagramma semplificato

Campo	Significato
Versione	
Classe di traffico	Campo a 20 bit utilizzato per identificare un flusso di dati
Etichetta di flusso	
Lunghezza del payload	
TTL	
Indirizzi sorgente/destinazione	
Data	

Tabella 14.4: Struttura di un datagramma IPv6

Un'altra differenza tra le due versioni riguarda anche la **frammentazione**. Il protocollo IPv6 non consente **frammentazione** e **riassemblaggio** dei pacchetti sui router intermedi, ma solo su sorgente o destinazione. Se ad un router arriva un pacchetto troppo grande per poter essere inviato al destinatario verrà eliminato e il router invierà al destinatario un messaggio di errore **ICMP** del tipo *pacchetto troppo grande*. Il motivo di questa scelta dipende proprio dal fatto che frammentazione e **riassemblaggio** sono operazioni che consumano troppo tempo; trasferire questo onore ai sistemi periferici, invece che ai router, velocizza enormemente il traffico. Inoltre, siccome sia livello di trasporto che livello di collegamento calcolano i propri checksum, i progettisti di questo protocollo hanno ritenuto che questa funzionalità fosse ridondante e l'hanno quindi rimossa. Infine, il campo opzioni non fa più parte della struttura del datagramma IP standard, anche se, a differenza dei due campi precedenti, non è stato propriamente rimosso: è una delle possibili intestazioni successive cui punta l'intestazione IPv6.

**Transizione da IPv4 a IPv6** Il problema della transizione da IPv4 a IPv6 è che, mentre i nuovi sistemi IPv6 sono retro-compatibili, i sistemi IPv4 esistenti non sono compatibili con la nuova versione IPv6. Fintanto che la transizione non sarà completa, deve esistere un modo per permettere la coesistenza tra router IPv4 e router IPv6. Per farlo esistono due tecniche

- **Tunneling.** Supponendo che due nodi IPv6, connessi tra loro da dei router IPv4 intermedi, vogliono comunicare. L'idea del tunneling è quella di includere il datagramma IPv6 all'interno del campo dati IPv4. Questo datagramma IPv4 viene quindi instradato verso una rete di router **IPv4**, detti **tunnel**, e, una volta giunto a destinazione, viene estratto il datagramma **IPv6**.
- **Tecnica della doppia pila.** Questa tecnica prevede che ogni router **IPv6** sia dotato anche di una implementazione **IPv4** completa. Tali nodi sono capaci di inviare e ricevere sia datagrammi **IPv4**, sia datagrammi **IPv6**. Ne consegue che un router di questo tipo debba essere dotato sia di un indirizzo IPv4 e un indirizzo IPv6 e, inoltre, debba anche distinguere agilmente i due datagrammi.

Ad oggi non è possibile quantificare a che punto siamo della transizione. Uno dei pochi dati a riguardo proviene da google, il quale ha pubblicato uno studio in cui attesta che solo il 45% dei clients accedono ai loro servizi tramite **IPv6**.

## 7 Address Resolution Protocol

Abbiamo visto, durante lo studio di questo e del capitolo scorso, che su internet coesistono due tipologie di indirizzi: indirizzi **MAC** e indirizzi **IP**. Affinché sia però possibile utilizzare entrambi in modo cooperativo e coerente è necessario che sia definita una funzione di conversione che, da un indirizzo MAC permette di ricavare un indirizzo IP e viceversa. Infatti, quando un datagramma IP viene passato al livello successivo, viene incapsulato in un frame con degli ulteriori campi

intestazione, tra cui **indirizzo MAC sorgente** e **indirizzo MAC destinatario**. Il secondo campo non è però noto al mittente e, quindi, viene giustificata la necessità di un protocollo che permetta, conoscendo l'indirizzo IP destinatario, di conoscere il MAC del destinatario. Su internet questo compito viene affidato al protocollo **ARP** (o **Address Resolution Protocol**).

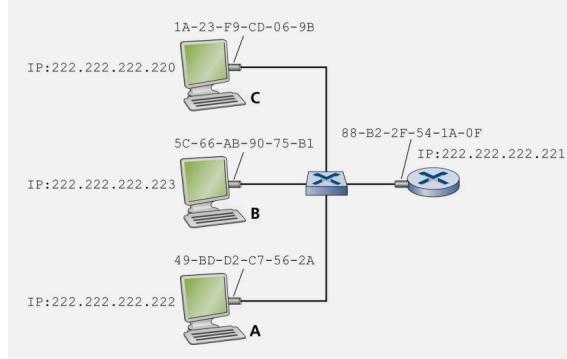


Figura 14.6: Protocollo ARP. Esempio di rete locale con 3 host e un router

L'idea alla base del meccanismo rimane quella di utilizzare una tabella di corrispondenza tra **indirizzi IP** e **indirizzi MAC**, detta **tavella ARP**. L'idea è che questa tavella si comporti come una sorta di memoria cache, con un suo **TTL** per ogni campo oltre il quale la voce viene invalidata. Questa necessità è data dal fatto che gli indirizzi IP non sono fissi e, quindi, una voce di una tabella **ARP** non sarà sempre valida. Una volta definita la tabella è necessario popolarla, ma anche in questo caso il meccanismo è estremamente intuitivo: se non esiste la corrispondenza per quell'indirizzo IP si invia uno speciale pacchetto in broadcast. In particolare, riprendendo l'esempio di figura 14.6, se il nodo A vuole comunicare con il nodo B, una volta passato il datagramma IP a livello di collegamento, al momento del controllo della corrispondenza nella tabella ARP, possono succedere due cose:

- Se viene trovata una corrispondenza tra l'indirizzo IP e l'indirizzo MAC viene popolato il campo relativo del frame.
- Se non viene trovata una corrispondenza, il nodo trasmittente costruisce uno speciale pacchetto, chiamato **pacchetto ARP**, che viene inviato in broadcast.

Il pacchetto **ARP** contiene diversi campi, tra cui i campi relativi agli indirizzi IP e MAC del mittente e del destinatario. In particolare, un pacchetto ARP di richiesta avrà nel campo relativo all'indirizzo MAC del destinatario, l'indirizzo MAC di broadcast, mentre il pacchetto di risposta avrà questo campo popolato con l'indirizzo MAC del dispositivo che ha trovato la corrispondenza con il proprio indirizzo IP. Una volta tornato al mittente, la tabella ARP, verrà conseguentemente aggiornata creando una nuova voce. I pacchetti di richiesta e di risposta ARP avranno quindi medesimo formato, permettendone una gestione agile e semplificata. Inoltre, anche in questo caso troviamo un'analogia del protocollo con la realtà: supponiamo che in una stanza ci siano diverse persone, una persona *A* vuole comunicare con una persona *B*, ma conosce solo l'indirizzo di casa, la città e il codice postale e non il codice fiscale. L'idea del protocollo ARP è quella che avvenga uno scambio di questo tipo:

```

A (rivolgersi a tutti): chi è il collega che
abita in Via Roma 12, Empoli, 50053
B (rivolgendosi ad A): sono io, il mio codice fiscale è XXXXXXXX.
A: Ciao XXXXXXXX, come stai ?
B: ...

```

L'analogia è immediata: A, non conoscendo il codice fiscale di B, chiede in broadcast a chiunque si trovi nella stanza chi ha quell'indirizzo di casa. B, riconoscendo l'indirizzo detto da A come il proprio indirizzo, risponderà alla comunicazione indicando il proprio codice fiscale, che verrà quindi salvato da A. Ci sono anche altri due aspetti interessanti del protocollo ARP: il primo

aspetto interessante è che ARP è un protocollo di tipo **plug-and-play**, poiché non richiede alcun intervento esterno per funzionare.

## 8 Internet Control Message Protocol

Insieme al protocollo IP e all'algoritmo di instradamento, la terza componente fondamentale del livello di trasporto è il protocollo **ICMP**. Questo protocollo, definito a supporto del protocollo IP, permette ai router o agli host di scambiarsi informazioni a livello di rete: il suo uso più tipico è la notifica degli errori. Ad esempio, un host che cerca di connettersi ad una rete non raggiungibile riceverà un messaggio **ICMP** di errore. Esistono innumerevoli codici di errore ICMP, tuttavia i principali sono:

Tipo ICMP	Codice	Descrizione
0	0	risposta echo (a ping)
3	0	rete destinazione irraggiungibile
3	1	host destinazione irraggiungibile
3	2	protocollo destinazione irraggiungibile
3	3	porta destinazione irraggiungibile
3	6	rete destinazione sconosciuta
3	7	host destinazione sconosciuto
3	4	riduzione (controllo di congestione)
8	0	richiesta echo
9	0	annuncio di un router
10	0	scoperta di un router
11	0	TTL scaduto
12	0	intestazione IP errata

Tabella 14.5: Tipi di messaggio ICMP.

Il motivo per cui ICMP non è direttamente assimilato a IP è dato dal fatto che i messaggi ICMP vengono trasportati nei datagrammi IP: ossia, i messaggi ICMP vengono trasportati come payload di IP, esattamente come i segmenti TCP o UDP. I messaggi **ICMP** hanno un campo **tipo** e un campo **codice** e contengono l'intestazione e i primi 8 byte del datagramma IP che ha provocato la generazione del messaggio. In questo modo permettono al mittente di determinare esattamente quali datagrammi hanno generato l'errore. Un esempio di comando che genera un messaggio ICMP, nello specifico, un messaggio ICMP di tipo 8 e codice 0, è il comando `ping <indirizzo ip>`.

## 9 Forwarding generalizzato

Fino a questo momento abbiamo visto un solo tipo di forwarding, cioè quello basato sull'indirizzo del destinatario. In questa tipologia di forwarding l'unico parametro che utilizza il router per instradare il pacchetto (operazione di forwarding) è l'indirizzo IP del destinatario. Un altro possibile approccio, detto **forwarding generalizzato**, prevede che vengano utilizzati molteplici campi per poter prendere la decisione sul forwarding. Questa versione utilizza un approccio detto **match-plus-action**, cioè si definisce un **match**, quindi un'insieme di campi su cui basare la nostra decisione, e un **action**, quindi cosa succede quando consultando la tabella si trova una corrispondenza con quel particolare match. Ovviamente anche la tabella di forwarding dovrà essere diversa rispetto a quella dell'approccio **destination-address based**; in particolare, la nuova tabella è chiamata **tabella di flusso**. Un protocollo utilizzato per popolare la tabella di flusso è detto **OpenFlow**. Questo protocollo definisce un formato per la tabella basato su tre campi:

- **Match.** Questo campo mette insieme tutti i campi presi in considerazione per la condizione. Le informazioni, come detto anche in precedenza, possono essere anche relative a livelli dello stack protocollare diversi.
- **Action.** Questo campo specifica cosa fare quando avviene un riscontro con il rispettivo campo **match**. Le azioni possibili sono molteplici, ad esempio:
  - *Mandare il pacchetto verso una porta.*
  - *Eliminare il pacchetto.*
  - *Modificare i campo nell'header.*
  - *Incapsulare il frame.*

- **Stats.** Questo campo contiene dei contatori che possono essere aggiornati. Il loro scopo principale è per definire delle statistiche generali sulla voce della tabella.

Il vantaggio di questo protocollo deriva dall'enorme possibilità implementativa che offre. Utilizzando campi diversi all'interno del campo match è possibile implementare diversi dispositivi. Ad esempio, utilizzando le regole di **OpenFlow** è possibile implementare un router, uno switch, un firewall o anche un server NAT.

## 10 Middlebox

Abbiamo visto come nella rete, oltre ai router, che di fatto costituiscono l'elemento nevralgico e centrale di questo livello, grazie alla loro capacità di instradare i pacchetti determinando, al contempo, il percorso ottimale. Tuttavia, in mezzo a questa fitta rete di router, troviamo anche altri attori il cui scopo va ben oltre l'inoltro. Tutti questi dispositivi possono rientrare sotto alla definizione di **middlebox**

Ogni dispositivo intermedio che esegue funzioni al di fuori del normale funzionamento di un router e che si trova sul percorso tra un host sorgente e un host destinatario

La traduzione degli indirizzi con il NAT, o anche i servizi di sicurezza come i firewall, sono chiari esempi di middlebox, poiché si trovano nel mezzo alla comunicazione tra due host, ma le loro funzioni vanno oltre l'inoltro dei pacchetti.

# Capitolo 15

## Algoritmi di instradamento

Uno degli aspetti fondamentali del livello di rete e, in particolare, dei router è la loro capacità di decidere quale sia il percorso migliore, per ogni pacchetto, per arrivare a destinazione. Questa decisione viene presa attraverso degli **algoritmi di routing**. Fino ad ora abbiamo fatto una distinzione tra

- Piano di controllo (**routing**).
- Piano dei dati (**forwarding**).

Il secondo piano riguarda la capacità del router di muovere i pacchetti tra le interfacce di ingresso e le interfacce di uscita utilizzando una tabella di forwarding. Il primo piano è invece quello che costruisce questa tabella di forwarding attraverso delle informazioni che riesce a calcolare. In particolare, all'interno di questi capitoli studieremo i cosiddetti **problemi di instradamento**.

### 1 Il problema dell'instradamento

Una possibile formulazione del problema dell'instradamento passa dal formalismo astratto dei grafi. Ricordiamo che un grafo  $G = (N, E)$  è un insieme di  $N$  nodi e un insieme  $E$  di archi, ove ciascun arco collega una coppia di nodi. Nel contesto dell'instradamento a livello di rete, i nodi del grafo, sono di fatto i router della rete, mentre gli archi che connettono tali nodi rappresentano i collegamenti fisici tra i router. Ad ognuno di questi archi possiamo associare una metrica, cioè un valore, detto **costo**. In genere questa metrica deve essere decisa a priori e può indicare diversi parametri che caratterizzano un collegamento:

- Lunghezza del collegamento.
- Congestione del collegamento.
- Velocità di collegamento.
- Prezzo del collegamento.

Sempre per mantenere una certa coerenza con il formalismo matematico dobbiamo anche considerare tutti i percorsi tra due nodi  $(i, j)$  che non esistono nella realtà come percorsi con un costo  $C(i, j)$  pari a  $+\infty$ . Inoltre supponiamo anche che il percorso  $(i, j)$  abbia medesimo costo indipendentemente dal verso in cui lo percorriamo:

$$c(i, j) = c(j, i)$$

Il problema dell'instradamento riguarda quindi cercare un percorso che massimizzi o minimizzi una particolare metrica scelta. Nel formalismo dei grafi, un **percorso**, identifica una sequenza di nodi  $(x_1, x_2, \dots, x_n)$  tali che ciascuna delle coppie  $(x_1, x_2), (x_2, x_3), \dots, (x_{n-1}, x_n)$  appartengono

all’insieme degli archi validi. Il costo di un percorso è definito come la somma dei costi di ogni singolo arco che compone il percorso

$$\text{Costo del percorso} = C_{P,k} = \sum_{i=1}^n c(x_{i-1}, x_i), \quad (x_{i-1}, x_i) \in E$$

In generale quindi esistono diversi possibili percorsi che collegano un generico host  $A$  della rete a un generico host  $B$ . Tuttavia esistono dei percorsi con un costo più alto oppure percorsi con dei costi più bassi; l’idea alla base degli algoritmi di routing è quella di prendere il percorso con costo minimo tra quelli presenti:

$$C_P = \min_{k=1,\dots,m} C_{P,k} \quad (1)$$

Ci sono due approccio tradizionalmente usati per il **forwarding**: **per-router control** e il **Local Centralized Control**. Nel primo approccio il percorso a costo minimo viene calcolato iterativamente in modo distribuito. Nessun nodo possiede informazioni sul costo di tutti i collegamenti di rete; inoltre, all’avvio del sistema, ogni router conosce soltanto le informazioni relative ai propri vicini, cioè ai propri nodi **adiacenti**. A partire da questa conoscenza, l’algoritmo calcola una prima stima dei vari percorsi possibile e, man mano che i router vicini inviano nuove informazioni, ricalcola iterativamente i percorsi. Nel secondo approccio ogni router calcola il percorso a costo minimo tra una sorgente e una destinazione avendo a disposizione una conoscenza globale e completa della rete. In altre parole, l’algoritmo riceve in ingresso tutti i collegamenti tra i nodi e i loro costi. Ciò richiede che l’algoritmo ottenga, in qualche modo, tutte le informazioni prima di effettuare il calcolo effettivo sul percorso. Appartengono alla prima categoria gli algoritmi cosiddetti **distance vector**, alla cui base troviamo l’**equazione di bellman-ford**, mentre appartengono alla seconda categoria gli algoritmi cosiddetti **link state**. Un ulteriore criterio su cui è possibile classificare gli algoritmi di instradamento è il criterio di suddivisione in algoritmi:

- **Statici.** In questa tipologia di algoritmi i percorsi cambiano molto raramente e, spesso, come risultato di un intervento umano.
- **Dinamici.** In questa tipologia di algoritmi i percorsi cambiano molto velocemente, costringendo a un ricalcolo della tabella di forwarding molto frequente

## 2 Algoritmo di instradamento link state

Come abbiamo detto nella scorsa sezione, gli algoritmi **link state** sono quegli algoritmi che necessitano che ogni nodo intermedio della rete (**router**) abbia una conoscenza **globale** e **assoluta** di tutti i collegamenti presenti sulla rete. Per arrivare ad avere questa conoscenza, prima che la comunicazione tra host possa iniziare, è necessario che i router abbiano una fase di scambio preliminare in cui ciascun nodo invia informazioni sullo stato dei suoi collegamenti a tutti gli altri nodi della rete; per raggiungere questo scopo è quindi necessario adottare un algoritmo di **link-state broadcast** (**OSPF** ad esempio). Uno degli algoritmo **link state** più conosciuti è l’algoritmo di **Dijkstra**. L’idea di questo algoritmo è quella di calcolare il percorso a costo minimo da un nodo, detto **origine**, a tutti gli altri nodi della rete. L’algoritmo di **Dijkstra** rispetta una proprietà fondamentale: dopo la  $k$ -esima iterazione, i percorsi a costo minimo sono noti a  $k$  nodi di destinazione e, tra i percorsi a costo minimo verso tutti i nodi di destinazione, questi  $k$  percorsi hanno i  $k$  costi più bassi. Definiamo anche la seguente notazione

- $D(v)$ . Indica il costo minimo del percorso dal nodo origine alla destinazione  $v$  per quanto concerne l’iterazione corrente dell’algoritmo.
- $p(v)$ . Immediato predecessore di  $v$  lungo il percorso a costo minimo dall’origine a  $v$ .
- $N'$ . Sottoinsieme di nodi contenente tutti e soli i nodi  $v$  per cui il percorso a costo minimo dall’origine a  $v$  è definitivamente noto.

L’algoritmo globale di instradamento consiste in un passo di inizializzazione seguito da un ciclo che viene eseguito per una volta per ogni nodo del grafo. Una volta terminato, l’algoritmo, avrà

calcolato il percorso minimo dal nodo di origine  $u$  a tutti gli altri nodi. Possiamo schematizzare il funzionamento dell'algoritmo nei seguenti passi:

---

**Algorithm 1** Algoritmo LS dal nodo origine  $u$ 


---

**Inizializzazione:**

```

 $N' = \{u\}$ 
for all nodi  $v$  do
    if  $v$  è adiacente a  $u$  then
         $D(v) = c(u, v)$ 
    else
         $D(v) = \infty$ 
    end if
end for

```

**Ciclo**

```

while  $N' \neq N$  do
    determina un  $w \notin N'$  tale che  $D(w)$  sia minimo
    aggiungi  $w$  a  $N'$ 
    for all nodi  $v$  adiacenti a  $w$  e non in  $N'$  do
         $D(v) = \min(D(v), D(w) + c(w, v))$   $\triangleright$  il nuovo costo verso  $v$  è il vecchio costo verso  $v$ 
        oppure il costo del percorso minimo noto verso  $w$  più il costo da  $w$  a  $v$ 
    end for
end while

```

---

Nel momento in cui l'algoritmo termina, abbiamo per ciascun nodo i suoi predecessori lungo il percorso a costo minimo dal nodo di origine. Complessivamente l'algoritmo esegue un numero di confronti pari a  $n(n+1)/2$  e, di conseguenza, diciamo che l'implementazione vista fino ad ora dell'algoritmo ha complessità, nel caso peggiore, di ordine  $\Theta(n^2)$ . Tuttavia, utilizzando un'implementazione più sofisticata che sfrutta strutture dati come **heap**, il costo scende anche fino a  $\Theta(n \log(n))$ .

Negli algoritmi di instradamento *link-state* (LS), ogni router calcola i percorsi a costo minimo eseguendo periodicamente l'algoritmo di Dijkstra su una visione globale della rete. Quando la metrica del collegamento non è costante ma *dipende dal traffico*, ovvero quando il costo  $c(u, v)$  di un arco  $(u, v)$  è una funzione crescente del traffico che attraversa l'arco, si introduce una dipendenza tra tre elementi: **percorso**, **traffico** e **costo**. Infatti un aumento del traffico sulla rete comporta anche una variazione di costo di alcuni archi, che a sua volta porta a una variazione del percorso. Tuttavia, una variazione del percorso comporta che il traffico aumenti su altri archi, portando a un ulteriore ricalcolo del percorso. Una situazione di questo tipo non sarebbe problematica se non per un fatto che non è possibile ignorare: i router si aggiornano tutti insieme e, quindi, tutti ricalcoleranno lo stesso percorso, ridirigendo il traffico verso lo stesso percorso. Si nota immediatamente come si viene a formare una situazione in cui i percorsi non tendono a convergere, ma a variare ciclicamente. Per risolvere questo problema esistono due possibili soluzioni:

- Usare dei costi degli archi che **non siano dipendenti dal traffico della rete**, in questo modo si costringe l'algoritmo a convergere verso un insieme di percorsi costante.
- **Evitare la sincronizzazione tra router**, evitando che i router si aggiornino tutti allo stesso istante si elimina di fatto il problema.

### 3 Algoritmo di instradamento distance vector

A differenza degli algoritmi *link state*, gli algoritmi *distance vector*, non necessitano di una conoscenza **assoluta** e **globale** dei collegamenti di ogni altro nodo della rete, ma calcolano il **percorso a costo minimo** man mano che ricevono informazioni dagli altri router **adiacenti** sulle variazioni dei percorsi della rete; diciamo quindi che l'algoritmo è **asincrono** e **distribuito**.

L'equazione cardinale di questi algoritmi è **l'equazione di Bellman-Ford**, definita come

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

Tale relazione è piuttosto intuitiva. Infatti, se dopo aver viaggiato da  $x$  a  $v$ , consideriamo il percorso a costo minimo da  $v$  a  $y$ , il costo del percorso tra  $x$  e  $y$  sarà quindi dato dalla somma  $c(x, v) + d_v(y)$ . Dato che dobbiamo iniziare viaggiando verso qualche vicino  $v$ , il costo minimo da  $x$  a  $y$  è il minimo di  $c(x, v) + d_v(y)$  calcolato su tutti i nodi adiacenti  $v$  adiacenti a  $x$ . L'idea alla base è la seguente, ciascun nodo  $x$  inizia con  $D_x(y)$ , una stima del costo del percorso a costo minimo da se stesso al nodo  $y$ , per tutti i nodi in  $N$ . Sia  $D_x = [D_x(y) : y \in N]$  il vettore delle distanze del nodo  $x$ , che è il vettore delle stime dei costi da  $x$  verso tutti gli altri nodi  $y$ , in  $N$ . Con l'algoritmo **distance vector**, ciascun nodo  $x$  mantiene i seguenti dati di instradamento:

- Per ciascun vicino  $v$ , il costo  $c(x, v)$  da  $x$  al vicino  $v$ .
- Il vettore delle distanze del nodo  $x$  contenente la stima presso  $x$  del costo verso tutte le destinazioni  $y$ .
- I vettori delle distanze di ciascuno dei suoi vicini, ossia  $D_v$ , per ciascuno dei suoi vicini.

L'idea di questo algoritmo è che di tanto in tanto i vicini di un nodo mandino una copia del proprio vettore delle distanze (**distance vector**) ai propri vicini. Quando un nodo  $x$  riceve un nuovo vettore da uno dei suoi vicini, lo salva e utilizza la formula di **Bellman-Ford** per ricalcolare tutti i percorsi e aggiornare il proprio vettore. A questo punto possono succedere due cose: se il **distance vector** di  $x$  è cambiato anche solo per uno dei percorsi, allora manderà una copia a tutti i suoi vicini, mentre se il vettore non è cambiato non manderà alcuna copia. A questo punto i vicini faranno lo stesso, inviando le proprie copie dei vettori delle distanze ai loro vicini. Anche in questo caso è possibile dare una schematizzazione dell'algoritmo sotto forma di pseudocodice

---

**Algorithm 2** Algoritmo distance vector (DV)

```

1: A ciascun nodo  $x$ :
2: Inizializzazione:
3: for tutte le destinazioni  $y \in N$  do
4:    $D_x(y) = c(x, y)$                                       $\triangleright$  se non è adiacente, allora  $c(x, y) = \infty$ 
5: end for
6: for ciascun vicino  $w$  do
7:    $D_w(y) = ?$                                           $\triangleright$  per tutte le destinazioni  $y \in N$ 
8: end for
9: for ciascun vicino  $w$  do
10:   invia il vettore delle distanze  $D_x = [D_x(y) : y \in N]$  a  $w$ 
11: end for

12: ciclo
13: while true do
14:   attendi (finché vedi cambiare il costo di un collegamento verso qualche vicino  $w$ 
      o finché ricevi un vettore delle distanze da qualche vicino  $w$ )
15:   for ogni  $y \in N$  do
16:      $D_x(y) = \min_v \{c(x, v) + D_v(y)\}$ 
17:   end for
18:   if  $D_x(y)$  è cambiato per qualche destinazione  $y$  then
19:     invia il vettore delle distanze  $D_x = [D_x(y) : y \in N]$  a tutti i vicini
20:   end if
21: end while

```

---

Negli algoritmi **distance vector**, una variazione del costo in un percorso, rilevata quando viene aggiornato il proprio **vettore delle distanze**, comporta un invio del proprio vettore a tutti i nodi adiacenti. Tuttavia una soluzione di questo tipo funziona bene nel caso in cui il costo di

un particolare collegamento diminuisca, mentre nel caso in cui il costo del collegamento aumenti si potrebbero verificare alcune problematiche. Questa problematica, detta **count-to-infinity** si verifica quando un collegamento si guasta o il suo costo aumenta molto velocemente, l'informazione potrebbe propagarsi molto lentamente. Ipotizziamo di avere la seguente situazione:

Prima che il costo del collegamento cambi abbiamo le seguenti informazioni:  $D_y(x) = 4$ ,  $D_y(z) = 1$ ,  $D_z(y) = 1$  e  $D_z(x) = 5$ . All'istante  $t_0$ ,  $y$  rileva che il costo del collegamento è passato da 4 a 60 e calcola il suo nuovo percorso a costo minimo verso  $x$  con la formula:

$$D_y(x) = \{60 + 0, 1 + 5\} = 6$$

Ovviamente, con la nostra visione della rete, possiamo dedurre facilmente che questo costo è attraverso  $z$  è errato. Tuttavia l'unica informazione che  $y$  possiede è che il costo diretto verso  $x$  è 60 e che  $z$  ha ultimamente detto a  $y$  di essere in grado di raggiungere  $x$  a un costo di 5. Pertanto al fine di arrivare a  $x$ ,  $y$  ora farebbe passare il percorso per  $z$ , aspettandosi che questo sia in grado di giungere a  $x$  con un costo pari a 5. All'istante  $t_1$ , abbiamo un *instradamento ciclico*: al fine di giungere a  $x$ ,  $y$  fa passare il percorso per  $z$  e  $z$  lo fa passare per  $y$ . Un ciclo in un instradamento assomiglia a un buco nero: un pacchetto destinato a  $x$  che arriva a  $y$  o  $z$  all'istante  $t_1$ , rimbalzerà avanti e indietro tra questi due nodi per sempre.

Per risolvere questo problema è possibile ricorrere alla **inversione avvelenata**. L'idea è semplice, se  $z$  instrada tramite  $y$  per raggiungere la destinazione  $x$ , allora  $z$  avvertirà  $y$  che la sua distanza verso  $x$  è infinita, cioè  $z$  comunicherà a  $y$  che  $D_z(x) = +\infty$ . Dato che  $y$  crede a questa bugia, non tenterà mai di instradare verso  $x$  passando per  $z$ .

## 4 Link state vs Distance vector

I due algoritmi studiati utilizzano approcci complementari nel calcolare l'instradamento. Nell'algoritmo **Distance vector** ciascun nodo dialoga solo con i vicini direttamente connessi, informandoli delle stime a costo minimo da sé stesso verso tutti gli altri nodi. Nell'algoritmo **Link State** ciascun nodo dialoga con tutti gli altri nodi della rete via broadcast, ma comunica loro solo i costi dei collegamenti direttamente connessi. Possiamo quindi elencare le differenze principali:

- **Complessità dei messaggi.** Gli algoritmi **link state** richiedono che ciascun nodo conosca il costo di ogni collegamento della rete. Questo fatto implica l'invio di  $O(|N| \cdot |E|)$  messaggi, inoltre, ogni qualvolta cambia il costo di un collegamento deve essere comunicato a tutti i nodi, causando una notevole perdita di tempo. Negli algoritmi **distance vector**, a ogni iterazione, l'algoritmo potrebbe obbligare a scambi di messaggi tra nodi adiacenti. Quando cambiano i costi dei collegamenti, l'algoritmo **distance vector** propaga i risultati dei costi cambiati se il nuovo costo ha causato la variazione del percorso a costo minimo per uno o più nodi connessi a tale collegamento.
- **Velocità di convergenza.** L'implementazione classica degli algoritmo **link state** è un algoritmo  $\Theta(N^2)$  che richiede  $\Theta(|N| \cdot |E|)$ . Gli algoritmi **distance vector** invece possono convergere lentamente e possono presentare dei cicli di instradamento.
- **Robustezza.** Negli algoritmi **link state** un router può comunicare via broadcast un costo sbagliato per uno dei suoi collegamenti connessi. Negli algoritmi **distance vector** un nodo può comunicare percorsi a costo minimo errati a tutte le destinazioni.

## 5 Instradamento gerarchico

Fino ad ora abbiamo visto un routing basato su due possibili tipologie di algoritmi: algoritmi **link state** e algoritmi **distance vector**. Il problema di entrambe queste tipologie di routing è che sono relative a delle reti all'interno dello stesso dominio, cioè **intra-ISP**. Nella realtà la situazione è diversa, infatti non abbiamo un unico ISP e un unico dominio, ma una moltitudine di ISP e una moltitudine di router, portando a due problemi nell'instradamento che abbiamo visto fino ad ora:

- **Scalabilità.** All'aumentare del numero di router il tempo richiesto per calcolare, memorizzare e comunicare le informazioni di instradamento diventa proibitivo.

- **Autonomia amministrativa.** Sebbene tante volte non si consideri nello sviluppo di nuovi algoritmi la volontà delle singole società di amministrare i propri router, scegliendo magari l'algoritmo di routing preferito o nascondendo alcuni aspetti dell'organizzazione, nella realtà è necessario fare attenzione a questo aspetto.

La soluzione che potremmo pensare è quella di suddividere i router in gruppi, detti **sistemi autonomi (AS, Autonomous System)**. Questi gruppi di router sono generalmente posti sotto lo stesso controllo amministrativo, il quale è in grado di decidere sia l'algoritmo di instradamento (che verrà quindi adottato da tutti i router del **sistema autonomo**) sia altre configurazioni su quali informazioni possono essere esposte all'esterno; possiamo immaginare una situazione di suddivisione della rete nel seguente modo:

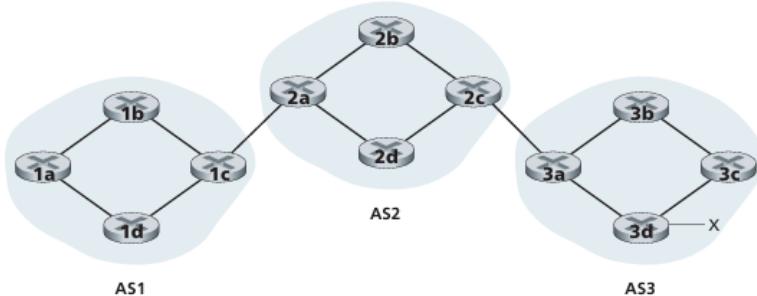


Figura 15.1: Rete con tre **sistemi autonomi**

Quindi, all'interno dello stesso **sistema autonomo** ogni router esegue lo stesso algoritmo di routing, scelto tra algoritmi **link state** e **distance vector**. Il problema diventa ora quello di connettere tra di loro diversi sistemi autonomi appartenenti anche a organizzazioni tra di loro differenti. Per farlo è necessario deputare un router all'interno di ogni sistema autonomo allo scopo di interconnettere il suo AS di appartenenza con gli altri AS presenti sulla rete internet; questi router sono detti **gateway router**. Il vantaggio di un'organizzazione di questo tipo è, in primis, la possibilità di poter pubblicizzare alcune rotte, mantenendone nascoste altre, isolando così il traffico interno. Tuttavia, affinché un router gateway funzioni correttamente, è necessario che esegua, contemporaneamente, due algoritmi diversi:

- **Algoritmo Intra-AS.** Questo algoritmo permette l'instradamento tra i router che si trovano all'interno del sistema autonomo ed è scelto da chi possiede il sistema autonomo.
- **Algoritmo Inter-AS.** Questo algoritmo permette l'instradamento tra i diversi **gateway router**, cioè tra i diversi **sistemi autonomi**.

Di conseguenza è anche necessario che mantenga aggiornate due tabelle di routing, una tabella relativa al sistema autonomo e una relativa all'esterno del sistema autonomo. In generale, tra i protocolli utilizzati per l'instradamento interno ad un sistema autonomo troviamo: **RIP**, **EIGRP** e **OSPF**, mentre per quanto riguarda i protocolli di instradamento tra sistemi autonomi troviamo il protocollo **BGP**. Analizzando la prima classe di protocolli di instradamento, cioè gli algoritmi **Intra-AS**, possiamo osservare che:

- Il protocollo **RIP** è un tipo di protocollo **distance vector**, utilizzato per reti con un basso numero di hop e che si basa su uno scambio periodico (generalmente ogni 30 secondi) di messaggi **RIP response** che contengono informazioni di instradamento (fino a 25 sottoreti) e di messaggi **RIP advertisement** di risposta.
- Il protocollo **OSPF** (Open Shortest Path First) è un protocollo **link-state** intra-AS, successore di RIP, progettato per superarne le limitazioni di scalabilità e convergenza. Utilizza una tecnica di **flooding** per diffondere le informazioni sullo stato dei collegamenti (LSA - Link State Advertisement) a tutti i router della rete. Ogni router costruisce una mappa topologica completa e identica dell'intero sistema autonomo (o della sua area), chiamata **LSDB** (Link State Database).

State Database). Su questo grafo, ogni nodo esegue localmente l'algoritmo di **Dijkstra** per calcolare l'albero dei cammini minimi (Shortest Path Tree) verso tutte le sottoreti, determinando così le rotte ottimali da inserire nella tabella di instradamento. Per gestire in modo efficiente e scalabile reti molto grandi (evitando che il database LSDB diventi enorme e che l'algoritmo di Dijkstra richieda troppa CPU), OSPF impone una struttura gerarchica a due livelli:

- **Aree Locali (Standard Areas)**. La rete viene suddivisa in gruppi di router chiamati "aree". I router interni a un'area conoscono la topologia dettagliata solo della propria area. Il flooding dei LSA di tipo dettagliato rimane confinato all'interno dell'area, riducendo il traffico di controllo
- **Backbone (Area 0)**. È l'area principale (obbligatoriamente identificata come Area 0) che funge da dorsale per l'intero sistema. Il suo compito primario è interconnettere le aree locali tra loro. Tutto il traffico che deve transitare da un'area all'altra deve obbligatoriamente passare attraverso il Backbone.
- **Area Border Router (ABR)**. Sono i router speciali che appartengono sia a un'area locale che al Backbone. Il loro compito è "riassumere" le informazioni di raggiungibilità delle sottoreti della propria area e pubblicizzarle verso il Backbone (e viceversa), permettendo così il routing *inter-area*.

## 5.1 BGP

Quando si parla di protocolli **Inter-AS**, il protocollo che trova maggior utilizzo è il protocollo **BGP** (o *Border Gateway Protocol*). L'idea del protocollo è quella di dare la possibilità ad un sistema autonomo di pubblicizzare la propria esistenza comunicandola al resto della rete. In particolare, BGP, fornisce ad ogni sistema autonomo la possibilità di:

- Ottenere informazioni sulla raggiungibilità delle reti di destinazione di AS vicini.
- Determinare le rotte per le altre reti utilizzando delle informazioni di raggiungibilità.
- Propagare informazioni di raggiungibilità a tutti i router interni al sistema autonomo.

Il funzionamento del protocollo BGP è estremamente complesso e articolato, tuttavia è possibile cercare di darne quantomeno una spiegazione superficiale. In termini generali, in BGP, coppie di router si scambiano informazioni di instradamento utilizzando connessioni TCP semi permanenti usando la porta 179. Generalmente esiste una connessione TCP per ogni collegamento tra due router in due diversi sistemi autonomi. Inoltre esistono connessioni TCP anche per router interni alla stessa rete; ogni router ai capi di una connessione TCP viene detto **BGP peer** e la connessione TCP è detta **BGP session**. Le **BGP session** sono a loro volta classificabili in due tipologie

- **Sessione eBGP** se coinvolge due sistemi autonomi differenti.
- **Sessione iBGP** se coinvolge router appartenenti al medesimo sistema autonomo.

In BGP le destinazioni non sono degli host, ma piuttosto dei prefissi CIDR che rappresentano una sottorete o una collezione di sottoreti. Inoltre, quando un router annuncia un particolare prefisso per una sessione BGP, include anche un certo numero di attributi BGP. Quindi, possiamo dire che la combinazione di **prefisso** e **attributi** rappresenta, per **BGP**, una **rotta**. Questi attributi possono specificare diverse informazioni, come, ad esempio, i sistemi autonomi attraverso cui è passato l'annuncio del prefisso (attributo **AS-PATH**), oppure l'indirizzo IP del router da utilizzare per raggiungere una determinata rete di destinazione (attributo **NEXT-HOP**).

# Parte V

## Livello di trasporto

# Capitolo 16

## Introduzione e servizi del livello di trasporto

Nei due capitoli precedenti abbiamo visto il funzionamento del **livello di collegamento** e del **livello di rete**. Il quarto e penultimo livello del stack protocollare è il **livello di trasporto**. L'idea è che un protocollo a livello di trasporto mette a disposizione una comunicazione **logica** tra processi applicativi che risiedono su host differenti. Con **comunicazione logica** si intende il fatto che per i due host tutta l'architettura di rete è nascosta, dal loro punto di vista è come se esistesse un canale diretto tra i due host. I pacchetti a questo livello sono noti come **segmenti**, ottenuti attraverso una **segmentazione** dei pacchetti ricevuti dal livello applicativo e aggiungendo ad ognuno di questi **segmenti** una **intestazione di trasporto**. L'idea è che i pacchetti provenienti da processi diversi siano passati a livello di rete come un unico flusso di dati in uscita, mentre, dal lato destinatario, questo flusso di dati in ingresso dal livello di rete dovrà essere suddiviso e ricomposto in pacchetti pronti per essere distribuiti ai vari processi mittente. Queste due operazioni prendono il nome, rispettivamente, di **multiplexing** e **demultiplexing**. Inoltre, altre funzionalità del livello di trasporto possono essere:

- Trasporto affidabile dei dati.
- Controllo della congestione.
- Controllo del flusso

Queste funzionalità, come vedremo, non sono però obbligatorie e dipendono dalla scelta del protocollo a livello di trasporto. Questa scelta può essere fatta sia in modo da garantire queste funzionalità (protocollo TCP) o senza garantirle (protocollo UDP).

### 1 Multiplexing e Demultiplexing

Dal punto di vista del destinatario l'host riceve segmenti dal livello di rete. Il livello di trasporto ha il compito di inviare questi segmenti al processo applicativo appropriato in esecuzione sulla macchina host. Il livello di trasporto non consegna però il segmento direttamente al processo, esiste infatti un'astrazione intermedia, cioè quella del socket, che vincola il livello di trasporto a consegnare il segmento al socket associato al processo destinatario. Riprendendo l'analogia dell'ufficio e della posta, è come se la portineria, una volta ricevute le lettere, non le consegnasse direttamente al destinatario, ma alla cassetta postale che si trova al di fuori dell'ufficio.

Nella nostra analogia il livello di trasporto non è altro che una portineria sostanzialmente, che si occupa di svolgere due soli compiti, il primo compito è quello di raccogliere le lettere di ogni ufficio e di darle al correre, il secondo è quello di prendere le lettere dal corriere e di smistarle tra i vari uffici. In termini informatici queste due fasi prendono il nome di:

- **Multiplexing.** Questa fase corrisponde alla consegna delle lettere al postino. Il livello di trasporto raduna i vari frammenti di dati da diverse socket, li incapsula (ognuno con intestazioni a livello di trasporto) in dei segmenti e li passa al livello di rete.
- **Demultiplexing.** Questa fase corrisponde allo smistamento delle lettere tra i vari uffici. Il livello di trasporto riceve un unico flusso di segmenti dal livello di rete e, successivamente, li manda nei socket appropriati.

Notiamo immediatamente che la fase di **demultiplexing** ha un grado di difficoltà nettamente maggiore rispetto alla fase di **multiplexing**. Supponiamo di avere due client che cercano di scaricare, contemporaneamente, da un server HTTP, una generica risorsa web:

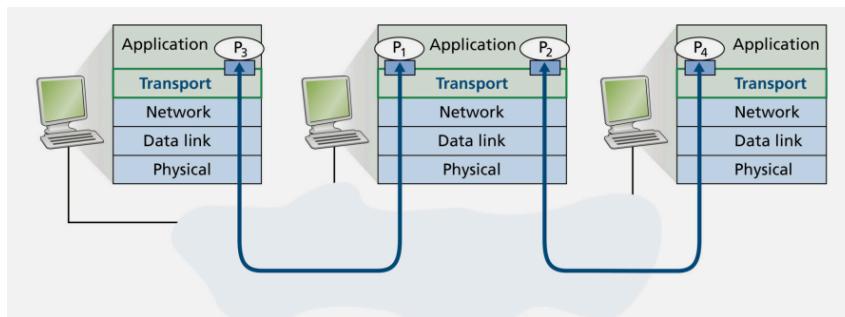


Figura 16.1: Esempio di comunicazione tra client e server. Sulla macchina server si trovano due processi  $P_1$  e  $P_2$  con due numeri di porta diversi, mentre sulle macchine client si trovano due processi.

Se i due client iniziano a inviare le proprie richieste in istanti di tempo vicini, nello specifico, in istanti di tempo sufficientemente vicini affinché le due richieste possano arrivare in contemporanea, allora, in ingresso al server arriverà un unico flusso di dati. Il livello di trasporto del server si dovrà occupare di fare **demultiplexing** sul flusso di dati affinché possa indirizzare correttamente le richieste verso i socket corretti, una volta generate le risposte i socket instraderranno verso il livello di trasporto i pacchetti, i quali verranno incapsulati all'interno di un **segmento** con dei campi relativi al protocollo a livello di trasporto, e successivamente verranno commutati (**multiplexing**) come un unico flusso di dati verso il livello di rete. Osserviamo quindi che la fase di **multiplexing** è nettamente meno complessa di quella di **demultiplexing**, poiché non richiedere alcuna operazione di analisi dei segmenti.

Affinché sia possibile operare la fase di demultiplexing è necessario che all'interno del segmento a livello di trasporto siano presenti due campi, uno che permette di identificare da quale processo (quindi il socket), sulla macchina mittente, ha iniziato la comunicazione e un campo che permette di identificare quale processo, sulla macchina di destinazione, è destinatario della comunicazione. Questi due campi prendono il nome di:

- **Campo numero di porta di origine.**
- **Campo numero di porta di destinazione.**

I numeri di porta sono costituiti da 16 bit e vanno da 0 fino a 65.535; i primi 1023 numeri sono riservati per essere usati da protocolli applicativi, mentre gli altri sono liberamente utilizzabili dai processi. Le due fasi, cioè quella di **multiplexing** e **demultiplexing**, sono diverse anche in funzione del protocollo a livello di trasporto che viene scelto:

- **Multiplexing orientato alla connessione.** In questa forma di multiplexing il protocollo a livello di trasporto è il TCP.
- **Multiplexing non orientato alla connessione.** In questa forma di multiplexing il protocollo a livello di trasporto è l'UDP.

# Capitolo 17

## User Datagram Protocol

Il primo protocollo a livello di trasporto che vediamo è il protocollo **UDP**, o **User Datagram Protocol**. Questo protocollo, nella sostanza, offre solamente i servizi minimi che un protocollo a livello di trasporto deve offrire, quindi:

- **Multiplexing e Demultiplexing** non orientati alla connessione.
- Una forma di **checksum**, quindi di controllo degli errori, minima e molto semplice.

L'UDP non aggiunge quindi nulla al protocollo IP e ha un funzionamento che è estremamente semplice. UDP prende i messaggi da livello applicativo, aggiunge un'intestazione estremamente semplice e trasferisce il pacchetto verso il livello di rete. A differenza del TCP, quindi, non esiste alcuna forma di handshake tra le varie entità di invio e di ricezione e, quindi, diciamo che UDP è un protocollo **non orientato alla connessione**. Questo fatto implica anche che non vi è alcuna sicurezza sulla consegna o meno del pacchetto al destinatario, poiché non è prevista alcuna forma di **ACK**.

All'apparenza sembra quindi che UDP sia inferiore al TCP a tal punto da non avere nessuna applicabilità nei protocolli reali. In realtà però esistono diversi protocolli a livello applicativo che utilizzano UDP come protocollo a livello di trasporto, il DNS e il DHCP sono tra questi, il motivo è presto detto:

- **Minor ritardo di trasferimento.** Non appena un processo applicativo passa i dati a UDP, quest'ultimo li impacchetta e li trasferisce immediatamente a livello di rete, TCP, invece, dispone un meccanismo di controllo di congestione che ritarda l'invio a livello di trasporto quando uno o più collegamenti tra l'origine e la destinazione diventano eccessivamente congestionati. Inoltre, per ogni pacchetto, richiede che venga inviata una conferma prima di spedire il successivo.
- **Nessuna connessione stabilita.** TCP, come abbiamo visto, stabilisce, prima dell'invio dei pacchetti, una connessione tra mittente e destinatario (handshake a tre vie). UDP, invece, da canto suo, non stabilisce alcune connessione, ma invia a raffica i pacchetti a livello di rete.
- **Nessuno stato di connessione.** TCP mantiene lo stato della connessione nei sistemi periferici. Questo stato include quindi un buffer di ricezione e di invio, insieme a dei parametri per il controllo della congestione, dei parametri sul numero di sequenza e di ACK. UDP, invece, non conserva lo stato della connessione e non tiene traccia di questi parametri.
- **Minore spazio usato per l'intestazione del pacchetto.**

Un esempio di applicazione che utilizzano UDP è quello delle applicazioni in tempo reale. Questo tipo di applicazioni richiedono infatti che venga garantita una velocità di trasmissione minima, tollerando però una certa perdita di dati. Il modello TCP non si adatta quindi bene a queste applicazioni, mentre il modello UDP è sicuramente più adeguato a quanto richiesto.

## 1 Struttura di un segmento UDP

La struttura di un segmento UDP è quindi molto semplice e contiene pochissimi campi. L'interazione è costituita da solo quattro campi: i primi due sono la **porta di origine** e la **porta di destinazione**, mentre, immediatamente sotto, troviamo un campo **lunghezza** e un campo **checksum**. I primi due campo sono fondamentali per le fasi di **multiplexing** e **demultiplexing**, il campo lunghezza specifica invece la lunghezza del campo dati del segmento UDP e il campo checksum serve per il controllo degli errori. Il corpo del segmento contiene il messaggio proveniente dal livello applicativo la cui dimensione può essere variabile.

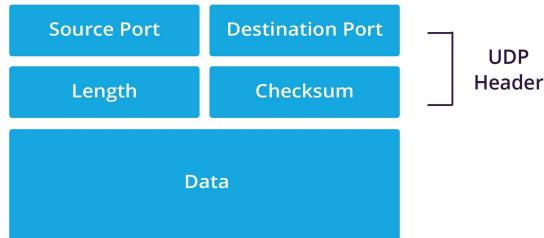


Figura 17.1: Struttura del segmento UDP

Il campo **checksum** viene utilizzato per determinare se i bit dei segmenti UDP sono stati alterati durante il loro trasferimento. Il mittente UDP effettua il complemento a uno della somma di tutte le parole da 16 bit nel segmento, e l'eventuale riporto finale viene sommato al primo bit. Tale risultato viene posto nel campo **checksum** UDP. Il destinatario ricalcola la stessa identica somma e fa il confronto con il **checksum** presente all'interno del campo del pacchetto: se sono uguali il pacchetto non è stato alterato, altrimenti sì. Questo meccanismo rappresenta un esempio del celebre **principio end-to-end**, poiché la verifica di integrità viene fatta solo a livello di **endpoint**.

# Capitolo 18

## Transmission Control Protocol

Il protocollo TCP, detto anche **Transmission Control Protocol**, è il secondo protocollo a livello di trasporto che vediamo. La prima differenza con il protocollo UDP è che, nel protocollo TCP, prima di iniziare il trasferimento dei pacchetti, i processi devono effettuare un **handshake**, ossia devono inviarsi reciprocamente alcuni segmenti preliminari per stabilire i parametri del successivo trasferimento dati (diciamo quindi che TCP è un **protocollo orientato alla connessione**). Inoltre, dal momento in cui avviene l'handshake, dal punto di vista dei processi, è come se esistesse un vero e proprio canale virtuale che li collega direttamente, chiamato **pipe**, e che garantisce due proprietà fondamentali:

- **Ordinamento dei pacchetti.** Il pipe, nonostante sia un'astrazione, è comunque un canale diretto tra i due processi che passa da una serie di dispositivi intermedi predefiniti. Di conseguenza, i pacchetti seguono tutti lo stesso percorso e, di conseguenza, arriveranno ordinati al destinatario.
- **Trasferimento affidabile.** Attraverso dei meccanismi di controllo del protocollo TCP è garantito che i pacchetti arrivino tutti al destinatario, senza nessun tipo di perdita.

Una connessione TCP offre un servizio full-duplex: su una connessione TCP tra un processo A e un processo B, su host diversi, i dati al livello applicativo possono fluire da A a B e viceversa. Un altro aspetto interessante del TCP riguarda il fatto che gli standard non definiscono il momento in cui TCP dovrebbe inviare pacchetti, l'unico vincolo è che lo faccia **quando è più conveniente**. La massima quantità di dati prelevabili e posizionabili in un segmento è detta **Maximum Segment Size**, o **MSS**. Questa grandezza è facilmente determinabile dalla seguente espressione

$$\text{MSS} = \text{Dimensione frame} - \text{Header collegamento} - \text{Header rete} - \text{Header trasporto}$$

Andando infatti a togliere gli header di livello due e di livello tre, rispettivamente, dal frame e dal datagramma si ottiene il segmento a livello di trasporto. Rimuovendo anche l'header a livello di trasporto si riesce a determinare quanti dati a livello applicativo possano essere inseriti in un segmento.

### 1 Struttura dei segmenti TCP

La struttura di un segmento TCP è estremamente più complessa rispetto a quella di un segmento UDP. La struttura base del segmento rimane comunque uguale, con un intestazione e un corpo dati limitato a una dimensione massima detta MSS. I campi presenti nell'intestazione sono:

- Numero di porta di origine/destinazione. Analoghi a quelli dell'UDP e necessari nelle fasi di **multiplexing/demultiplexing**.
- **Numero di sequenza** e **Numero di acknowledgment**. Necessari per il trasferimento dati affidabile.

- **Finestra di ricezione.** Ogni qualvolta viene inviata una conferma per il pacchetto in entrata, in questo campo, viene inserita la dimensione residua del buffer.
- **Lunghezza dell'intestazione.** Questo campo specifica la lunghezza dell'intestazione TCP in multipli di 32. La necessità di un campo di questo tipo dipende dal fatto che il numero di opzioni dell'intestazione TCP è variabile.
- **Opzioni.** Questo campo è facoltativo e di lunghezza variabile, viene utilizzato quando mittente e destinatario negoziano la dimensione massima del segmento o quando è necessario specificare altre opzioni.
- **Flag.** Questo campo contiene sei bit. Il bit di **ACK** viene usato per indicare che il valore trasportato nel campo **acknowledgement** è valido. I bit **RST**, **SYN** e **FIN** vengono utilizzati per impostare e chiudere la connessione; in particolare
  - Il bit **SYN** viene alzato in fase di apertura della connessione.
  - Il bit **FIN** viene alzato in fase di chiusura della connessione.
  - Il bit **RST** viene alzato quando la comunicazione va in errore e si intende ricominciare la trasmissione da capo.

Il bit **PSH** serve a segnalare al destinatario di inviare immediatamente il segmento a livello superiore. Infine, il bit **URG** serve a indicare nel segmento la presenza di dati che l'entità mittente a livello superiore ha marcato come urgenti. Altri due bit, **C** ed **E**, servono per il controllo esplicito della congestione.

Due campi fondamentali all'interno del segmento TCP sono i campi **numero di sequenza** e **il numero di acknowledgement**. Abbiamo infatti visto che entrambi questi campi hanno un compito centrale nel trasferimento dati affidabile offerto da questo protocollo. TCP vede i dati come un flusso di byte non strutturati e non necessariamente ordinati. Il **numero di sequenza** è il numero, nel flusso complessivo di byte, del primo byte del segmento; in altri termini specifica la posizione di quel segmento all'interno del flusso di byte. Il **il numero di acknowledgement** è il numero di sequenza del prossimo pacchetto che un ipotetico host A attende dall'host B. Supponiamo che l'host A abbia ricevuto dall'host B tutti i byte numerati da 0 a 535 e che A stia per mandare un segmento all'host B. L'host A, essendo in attesa del pacchetto con numero di sequenza 536, scriverà tale numero all'interno del campo **il numero di acknowledgement** del frame che sta preparando. Dato che TCP effettua l'acknowledgement solo dei byte fino al primo byte mancante nel flusso, si dice che tale protocollo offre **acknowledgement cumulativi**; se per esempio arrivassero i byte da 0 a 512 e, successivamente, i byte da 1000 a 1500, TCP manderebbe l'**ACK** solo per quelli arrivati in ordine.

## 2 Timeout e stima dell'RTT

TCP, in quanto protocollo **rdt**, utilizza un meccanismo di timeout e di ritrasmissione per recuperare i segmenti persi. Uno dei problemi dell'implementazione di questo meccanismo è proprio quello della stima di **timeout**, cioè della stima dell'intervallo di tempo oltre il quale è necessario rispedire il pacchetto. Ovviamente questo intervallo dovrebbe essere più grande del **round-trip-time**, ossia del tempo necessario da quanto si invia un pacchetto a quando si riceve la risposta. In termini matematici

$$\text{Timeout} > \text{Round-Trip-Time}$$

Il problema diventa quindi determinare una stima del round-trip-time per questo protocollo. Nel nostro caso, l'**RTT**, è definito come il tempo intercorso tra l'invio di un segmento e la ricezione dell'**ACK** e lo chiamiamo **SampleRTT**. La misurazione di **SampleRTT** non viene fatta per ogni segmento; ad ogni istante di tempo si sceglie uno dei pacchetti per il quale non è ancora arrivato l'**ACK** e si misura **SampleRTT**. Successivamente viene fatta una **media mobile esponenziale ponderata**, chiamata **EstimatedRTT**, tra i vari **SampleRTT**. In termini matematici:

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

Inoltre, è importante notare che **TCP** non misura **SampleRTT** per pacchetti ritrasmessi, ma solo per pacchetti inviati per la prima volta. Questa formula è una generalizzazione di quello che è la stima dell'algoritmo iterativo per il calcolo dell'**EstimatedRTT**. All'istante  $t = 0$  ancora nessun pacchetto è stato inviato e, di conseguenza, **SampleRTT** non può essere calcolato e, allora, imponiamo un valore iniziale di **SampleRTT** pari a  $\text{RTT}_0$ .

$$\begin{aligned}\text{ERTT}_1 &= \text{RTT}_0 \\ \text{ERTT}_2 &= \alpha \cdot \text{RTT}_1 + (1 - \alpha) \cdot \text{ERTT}_1 \\ \text{ERTT}_3 &= \alpha \cdot \text{RTT}_2 + (1 - \alpha) \cdot \text{ERTT}_2 \\ &\dots \\ \text{ERTT}_{n+1} &= \alpha \cdot \text{RTT}_{n+1} + (1 - \alpha) \cdot \text{RTT}_{n-1} + \dots + (1 - \alpha)^n \cdot \text{RTT}_0\end{aligned}$$

Andando a portare a fattore comune un coefficiente  $(1 - \alpha)$  ci rendiamo conto di una cosa:

$$\begin{aligned}\text{ERTT}_{n+1} &= \alpha \cdot \text{RTT}_{n+1} + (1 - \alpha)[\alpha \cdot \text{RTT}_{n-1} + \dots + (1 - \alpha)^{n-1} \cdot \text{RTT}_0] = \\ &= \boxed{\alpha \cdot \text{RTT}_{n+1} + (1 - \alpha) \cdot \text{ERTT}_n}\end{aligned}$$

Che corrisponde proprio alla formula che avevamo scritto in precedenza. Ponendo, ad esempio,  $\alpha = 0$  arriviamo a una situazione in cui la stima è sempre uguale al valore iniziale  $\text{RTT}_0$ , mentre ponendo  $\alpha = 1$  la stima sarà sempre uguale all'ultimo valore misurato. L'idea è quindi che

- Per valori di  $\alpha$  molto vicini ad uno si ottiene una stima molto reattiva, ma poco stabile.
- Per valori di  $\alpha$  molto vicini a zero si ottiene una stima molto stabile, ma poco reattiva.

Oltre a dare una stima del **Round-Trip-Time** è opportuno anche definire una sua **variabilità**. Lo standard definisce la variazione di **RTT**, cioè **DevRTT**, come una stima di quanto **SampleRTT** si discosta rispetto a **EstimatedRTT**

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

Anche in questo caso abbiamo una **media esponenziale ponderata** tale per cui se i valori di **SampleRTT** hanno delle fluttuazioni limitate, allora **DevRTT** assume valori molto piccoli. Quindi, conoscendo i valori di **EstimatedRTT** e **DevRTT** è necessario cercare di calcolare una stima del **Timeout**. Chiaramente l'intervallo non può essere inferiore a quello di **EstimatedRTT**, altrimenti potrebbero avvenire delle ritrasmissioni non necessarie e, inoltre, non dovrebbe essere troppo maggiore di **EstimatedRTT**, altrimenti TCP non ritrasmetterebbe velocemente il segmento perduto. Idealmente ci poacerebbe che il **timeout** a **EstimatedRTT** più un certo margine che dovrebbe essere grande quando c'è molta fluttuazione nei valori di **SampleRTT**. Tutti questi aspetti vengono presi in considerazione dalla seguente equivalenza matematica

$$\text{Timeout} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT} \tag{1}$$

Il problema principale legato alle ritrasmissioni deriva dal fatto che l'intervallo di **timeout** è considerevolmente lungo e, per tale motivo, ogni pacchetto perso aumenta altrettanto considerevolmente il ritardo **end-to-end**. Affinché sia quindi possibile limitare questo ritardo è possibile sfruttare il meccanismo degli **ACK duplicati**, relativi a un segmento il cui **ACK** è già stato ricevuto dal mittente, i quali permettono al mittente di verificare la perdita dei pacchetti ben prima dell'intervallo di **timeout**. Nel caso in cui si verifichi una perdita di dati, cioè il destinatario riceve dei pacchetti con numero di sequenza maggiore di quello atteso, può inviare nuovamente l'**ACK** per il pacchetto con numero di sequenza precedente a quello atteso (quindi l'ultimo per cui ha inviato l'**ACK**), duplicando di fatto l'**ACK** per quel pacchetto. Il mittente, quando riceve tre **ACK** duplicati, effettua una **ritrasmissione rapida**, rispedendo il segmento mancante prima che scada il timer.

### 3 Controllo di flusso del TCP

Uno dei servizi offerti dal protocollo TCP riguarda il **controllo di flusso**. Questo servizio ha come scopo principale quello di evitare che il mittente saturi il buffer del ricevente. La necessità di

questo meccanismo deriva dal fatto che **sender** e **receiver** potrebbero avere anche caratteristiche molto diverse tra di loro e, quindi, anche capacità di elaborazione diverse tra di loro. In particolare, se:

Capacità di invio Mittente  $\gg$  Capacità di elaborazione Destinatario

potrebbe verificarsi una situazione in cui il buffer del destinatario si satura molto velocemente, portando a una conseguente perdita di pacchetti. Il **controllo di flusso** è pertanto un servizio di confronto sulla velocità, dato che paragona la frequenza di invio del mittente con quella di lettura del destinatario. TCP offre controllo di flusso facendo mantenere al mittente una variabile chiamata **finestra di ricezione** che, in sostanza, fornisce al mittente un'indicazione dello spazio libero disponibile nel buffer destinatario. Per calcolare lo spazio libero residuo del buffer si utilizzano delle variabili; in particolare, ne distinguiamo tre:

- **LastByteRead**. Numero dell'ultimo byte nel flusso di dati che il processo applicativo in B ha letto dal buffer.
- **LastByteRcvd**. Numero dell'ultimo byte, nel flusso dati, che proviene dalla rete e che è stato copiato nel buffer di ricezione di B.
- **LastByteAcked**. Numero dell'ultimo byte, nel flusso dati, per cui è stato inviato l'**ACK**.

Ipotizziamo di avere un buffer, all'interno del quale sono presenti sia pacchetti che sono stati letti, pacchetti per cui è stato inviato l'**ACK**, ma che ancora non sono stati letti e altri pacchetti arrivati fuori ordine per i quali, per funzionamento del protocollo TCP, non è possibile inviare l'**ACK**; prendiamo come esempio la seguente situazione: Ipotizziamo anche che il buffer abbia una

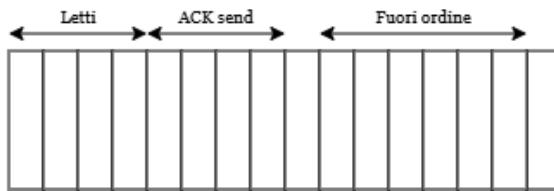


Figura 18.1: Buffer del ricevitore

dimensione pari a **BufferSize**. Per calcolare lo spazio occupato all'interno del buffer è possibile calcolare la differenza tra l'ultimo byte ricevuto e l'ultimo byte letto, infatti lo spazio che si trova tra **LastByteAcked** e il primo bit fuori ordine ricevuto **non è considerabile come spazio vuoto**; lo spazio occupato all'interno del buffer sarà quindi:

$$\text{Spazio Occupato} = \text{LastByteReceived} - \text{LastByteRead} \leq \text{BufferSize} \quad (2)$$

Dentro la finestra di ricezione andrà però inserito lo spazio libero all'interno del buffer, definito come la differenza tra il **BufferSize** e lo **Spazio Occupato**:

$$\text{Finestra di ricezione} = \text{BufferSize} - \text{Spazio Occupato} \quad (3)$$

Anche il mittente tiene conto di due contatori, il primo relativo all'ultimo bit per cui è stato inviato l'**ACK**, chiamato **LastByteAcked**, e un altro relativo all'ultimo byte inviato, chiamato **LastByteSent**. La differenza tra questi due indici, che indica il numero di byte spediti per cui ancora non si è ricevuto l'**ACK**, ricopre un ruolo fondamentale, poiché fintanto che viene rispettata l'uguaglianza

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{Finestra di Ricezione}$$

Il mittente non manderà mai in overflow il buffer del destinatario. Proviamo ora a supporre che il mittente non abbia ricevuto, a partire da un certo istante  $t$  di tempo, altre informazioni sulla dimensione della finestra di ricezione. Ipotizziamo anche che dall'ultimo aggiornamento sulla finestra di ricezione abbia inviato **LastByteSent** – **LastByteAcked** byte e che abbia ancora da inviare **DatiResidui** byte; allora, il mittente, potrà inviare al massimo

$$\text{Finestra di Ricezione} - (\text{LastByteSent} - \text{LastByteAcked})$$

Supponiamo ora che il buffer del destinatario, ad un certo istante temporale  $t_i$ , si saturi, cioè **Finestra di Ricezione** = 0, e che dopo averlo notificato all'host mittente non abbia più nulla da inviare. Il processo applicativo del destinatario, ad un certo istante  $t_k > t_i$ , svuoterà il buffer; questo fatto crea però un problema, l'host mittente non viene notificato dello svuotamento del buffer, poiché l'unico caso in cui riceve dei pacchetti dal destinatario è nel caso di **ACK** o di pacchetti dati. Si forma quindi una sorta di **stallo** del mittente, il quale si trova impossibilitato a mandare dati al destinatario. Per risolvere questo problema, le specifiche del TCP, richiedono che l'host mittente continui a inviare segmenti con un byte di dati quando la finestra di ricezione del destinatario è zero. Il destinatario risponderà a questi segmenti con un **ACK**. Prima o poi il buffer inizierà a svuotarsi e i riscontri conterranno un valore non nullo per la **Finestra di ricezione**.

## 4 Controllo di congestione

Uno dei meccanismi più importanti introdotti dal protocollo TCP riguarda il controllo della **congestione**. Il concetto di **congestione** può essere definito, in modo molto semplicistico, come una situazione in cui degli host inviano troppi dati e troppo velocemente affinché la rete li possa gestire correttamente. Questa situazione, indesiderata, si verifica sui router intermedi che si trovano tra **mittente** e **destinatario**. L'idea della **congestione** trova riscontro anche nella realtà: è sufficiente immaginare una rotonda in cui si ha molto traffico a determinati orari. Il motivo per cui si genera la **congestione** in questi dispositivi è data dal fatto che il numero di pacchetti in arrivo al router è molto maggiore maggiore del numero di pacchetti che il router è in grado di instradare

$$\text{Capacità in ingresso router} \gg \text{Capacità di uscita router}$$

Un router può essere infatti un dispositivo intermedio per diverse connessioni TCP e anche per diversi trasferimenti UDP e, quindi, può ricevere un numero molto alto di pacchetti in ogni unità di tempo. La congestione è una situazione indesiderata poiché può creare diverse conseguenze negative sulla rete:

- Un aumento, anche considerevole, del ritardo **end-to-end** tra i due host.
- Un aumento della **packet loss**, cioè della perdita di pacchetti. La congestione dei buffer in ingresso, infatti, potrebbe portare il router a decidere di scartare alcuni pacchetti.

Prima di parlare della risoluzione dei problemi di congestione è opportuno rimarcare che tra **controllo di flusso** e **controllo di congestione** esistono delle differenze sostanziali. La prima situazione riguarda soltanto l'host destinatario e, in particolare, la dimensione del suo buffer interno, mentre la seconda situazione può riguardare uno degli  $n$  dispositivi intermedi posti tra le due sorgenti di dati.

In senso generale, l'idea del controllo di congestione è quella di cercare di adattare, dinamicamente, il numero di pacchetti inviati da ciascun dispositivo in modo da rimuovere del carico elaborativo sul router affinché si risolva la congestione. Tuttavia, nonostante l'idea possa essere semplice all'apparenza, si deve comunque tenere conto che i pacchetti su un router possono provenire da diverse sorgenti e, quindi, risulta difficile coordinare tutti questi dispositivi. Possiamo distinguere due approcci con cui si possono risolvere i problemi di congestione:

- **Controllo di congestione end-to-end**. Il livello di rete non fornisce supporto esplicito al livello di trasporto per il controllo di congestione la cui presenza deve essere rilevata dai sistemi periferici sulla base dell'osservazione del comportamento della rete.
- **Controllo di congestione assistito dalla rete**. I componenti a livello di rete forniscono un feedback esplicito al mittente sullo stato di congestione della rete. Questo avviso dovrà essere inviato a tutti i dispositivi che stanno inviando dei datagrammi, avvisandoli di ridurre la quantità di datagrammi in uscita.

### 4.1 Controllo di congestione assistito dalla rete

Il primo meccanismo di controllo della congestione richiede che siano gli stessi dispositivi congestionati ad avvisare gli host che stanno inviando i pacchetti della situazione di congestione. Per farlo esistono due modi possibili:

- Il primo approccio è quello di utilizzare un pacchetto particolare, detto **choke package**, che riferisce "sono congestionato" agli host della rete.
- Il router imposta un campo particolare in un pacchetto che fluisce dal mittente al destinatario, per indicare la congestione. Alla ricezione del pacchetto marcato, il destinatario notifica al mittente l'indicazione di congestione.

Il secondo approccio è, sicuramente, più lento del primo, poiché dal momento che il pacchetto viene inviato dal mittente dovrà passare almeno un Round-Trip-Time prima che egli abbia la notifica della congestione. La terza possibilità è quella di utilizzare un algoritmo, chiamato **ATM ABR**, che utilizza la commutazione di pacchetto basata su circuito virtuale. L'idea del controllo di congestione **ATM ABR** è che sia il mittente stesso a calcolare in modo esplicito la massima velocità alla quale è in grado di trasmettere. Il sistema messo a disposizione da **ATM ABR** si basa su un sistema di feedback a circuito chiuso che permette alla rete di comunicare direttamente lo stato di congestione al mittente.

## 4.2 Controllo di congestione End-To-End

Il secondo approccio, quello **End-To-End**, prevede che ciascun mittente ponga un limite alla propria velocità di invio in funzione della congestione della rete. Questo limite viene calcolato iterativamente valutando la situazione di congestione della rete in modo dinamico

- Una bassa congestione porta il mittente ad alzare la velocità trasmittiva.
- Un'alta congestione porta il mittente ad abbassare la velocità trasmittiva.

La prima domanda che dobbiamo porre è come sia possibile però per il mittente ridurre la propria velocità trasmittiva e con quale metrica calcola tale velocità. Il meccanismo di congestione TCP fa tenere agli estremi della connessione di una variabile aggiuntiva: la **finestra di congestione**, indicata con **cwnd**, che impone un vincolo alla velocità di trasmissione di traffico sulla rete da parte del mittente. In particolare, la quantità di dati che non hanno ancora ricevuto un **ACK** inviata da un mittente non può eccedere il minimo tra i valori della finestra di congestione (**cwnd**) e di quella di ricezione (**rwnd**):

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

Ipotizziamo tuttavia che il buffer del destinatario sia sufficientemente grande da ignorare il problema del controllo di flusso. Il vincolo limita quindi solo in modo indiretto la velocità trasmittiva del mittente, infatti, all'inizio di ogni **RTT** consente al mittente di trasmettere, al massimo, **cwnd** di dati sulla connessione; al termine del **RTT** il mittente riceve gli **ACK** relativi ai dati. Quindi, la velocità di invio del mittente è circa pari al rapporto tra la finestra di congestione e l'**RTT**

$$\text{Rate [byte/s]} = \frac{\text{Finestra di congestione}}{\text{Round-Trip-Time}} = \frac{\text{cwnd}}{\text{RTT}}$$

Il secondo problema deriva dall'identificazione, da parte del mittente, della congestione di uno o più router intermedi, infatti senza alcuna tipologia di feedback è difficile per un mittente capire se la rete sia congestionato o meno. Definiamo come **evento di perdita** per il mittente l'occorrenza di un **timeout** o della ricezione di **tre ACK duplicati** da parte del destinatario. L'arrivo degli **ACK** è, per il mittente, l'indicazione di un corretto funzionamento della rete e l'assenza di un problema di congestione. Per tale motivo, se gli **ACK** arrivano con una frequenza relativamente bassa, allora la finestra di congestione verrà ampliata molto lentamente, mentre se gli **ACK** arrivano con una frequenza alta, avverrà l'opposto, cioè la finestra di congestione verrà ampliata più velocemente. Dato che TCP utilizza gli **ACK** come misura della congestione della rete, in funzione della quale amplia automaticamente la finestra di ricezione, diciamo che **TCP è auto-temporizzato**.

Il secondo problema che ci dobbiamo porre è la determinazione della velocità alla quale il mittente TCP dovrebbe trasmettere. Se infatti i client trasmettessero troppo velocemente, potrebbero congestionare la rete portando a tutte le conseguenze già descritte in precedenza. Le versioni più moderne del protocollo TCP definiscono dei principi guida per rispondere a questa domanda:

- Un segmento perso implica **congestione forte**, quindi i tassi di trasmissione del mittente TCP dovrebbero essere decrementati quando un segmento viene perso. In particolare, l'arrivo di tre **ACK duplicati** corrisponde a una congestione **limitata**, mentre lo scatto di un **timeout** per un pacchetto è sintomo di congestione forte.
- Un ACK indica che la rete sta consegnando i segmenti del mittente al ricevente e, quindi, il tasso di trasmissione del mittente può essere aumentato quando arriva un ACK non duplicato.

L'algoritmo che studieremo per il controllo di congestione TCP utilizza una tecnica che si basa sulla seguente idea: la velocità di trasmissione viene incrementata all'arrivo di ogni ACK fintanto che non si verifica un evento di perdita; a questo punto la velocità di trasmissione viene decrementata. Il mittente aumenta quindi la sua velocità trasmissiva fintanto che non si verificare un evento di perdita, momento nel quale diminuisce bruscamente la sua velocità di trasmissione, per poi ricominciare da capo.

**Algoritmo di controllo di congestione** Quindi, abbiamo descritto l'idea generale sulla quale si basano gli algoritmi di controllo della congestione. Tuttavia, l'implementazione effettiva dell'algoritmo è leggermente più articolata di quella vista fino ad ora. L'algoritmo, standardizzato nell'**RFC 5681**, per il controllo della congestione che studieremo si basa su tre fasi distinte:

- Slow start.
- Avoidance.
- Fast Recovery.

Quando si stabilisce una connessione TCP, il valore della finestra di congestione viene inizializzato ad un valore molto basso, circa pari a 1 MSS. Siccome però, almeno all'istante iniziale, la velocità di trasmissione possibile è sicuramente più alta di MSS/RTT, il mittente TCP vorrebbe scrosciare velocemente quanto può aumentare cwnd. Durante questa fase iniziale il valore della finestra di congestione (aka. cwnd) viene incrementato di un MSS per ogni segmento per cui viene ricevuto un ACK. Questo fatto implica che alla trasmissione successiva verranno inviati due segmenti di dimensione MSS e, quindi, una volta ricevuti i due ACK, la velocità trasmissiva verrà aumentata di 2 MSS, assestandosi su 4 MSS. Di fatto, nella fase di slow start, la velocità cresce seguendo un andamento esponenziale. A questo punto, man mano che la finestra di congestione cresce, possono succedere due cose:

- Avviene un fenomeno di congestione.
- Viene superata una soglia limite.

Nel primo caso, la finestra di congestione viene riportata ad uno e la soglia per entrare nella fase di **congestion avoidance** viene posta alla metà valore che aveva la finestra di congestione. Nel secondo caso si entra nella fase di **congestion avoidance**. In questa fase, piuttosto che per aumentare esponenzialmente la velocità di trasmissione, si utilizza un **approccio** più conservativo, incrementando cwnd linearmente di un singolo MSS per ogni dieci ACK ricevuti. Questa fase dell'algoritmo procede fintanto che non si verifica una tra queste due situazioni:

- **Arrivo di un triplice ACK duplicato.** In questo caso la soglia viene posta uguale alla metà della finestra di ricezione precedente, mentre la nuova finestra di congestione sarà la metà della precedente più un coefficiente pari a tre MSS.
- **Scadenza di un timeout.** In questo caso la soglia viene posta alla metà della finestra di congestione precedente e la nuova finestra di congestione viene messa a 1 MSS.

Nella prima situazione si passa nella fase di **fast-recovery**, mentre nella seconda situazione si torna nella fase di **slow-start**. Il funzionamento della fase di **fast recovery**, a grandi linee, si basa su una gestione intelligente delle soglie di trasmissione. Quando viene rilevata la perdita, il protocollo dimezza il valore attuale della finestra di congestione (cwnd) e imposta la soglia di slow start a questo nuovo livello. Invece di ricominciare la trasmissione da zero, il sistema "congela" la

finestra su un valore di sicurezza che permette di mantenere il flusso di dati attivo, assumendo che la rete sia ancora parzialmente operativa. Durante questo breve intervallo, il protocollo attende la conferma dell'avvenuta ricezione del pacchetto smarrito che è stato nel frattempo reinviato. Questa tecnica prende il nome di **AIMD**, cioè **Additive-Increase Multiplicative-Decrease**, e osservandone il grafico è possibile notare un andamento detto a **dente di sega**:

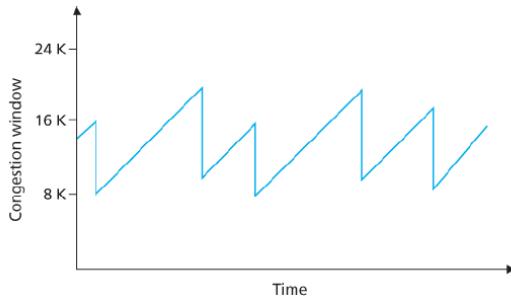


Figura 18.2: Andamento a *dente di sega* dell'algoritmo AIMD

Tuttavia, quello dell'**AIMD**, non è l'unico approccio possibile per il controllo di congestione. Esiste, infatti, anche un algoritmo detto **cubic** che invece di utilizzare un tipo di incremento lineare quando si ricevono gli ACK, ha un andamento che può essere approssimato ad una funzione cubica. Ciò permette di arrivare più velocemente al valore di soglia nelle prime fasi per poi diminuire l'incremento man mano che ci si avvicina al suddetto valore.

## 5 Inizio e chiusura di una connessione TCP

Nell'introduzione abbiamo accennato all'idea che il protocollo TCP, prima di permettere il trasferimento dati tra i due host, prevede che venga fatto un handshake per negoziare i parametri della comunicazione. L'handshake del protocollo TCP viene anche detto **handshake a tre vie** poiché i due sistemi periferici, per stabilire una connessione, si scambiano tre segmenti. La struttura di questo passaggio può essere schematizzata nella seguente immagine

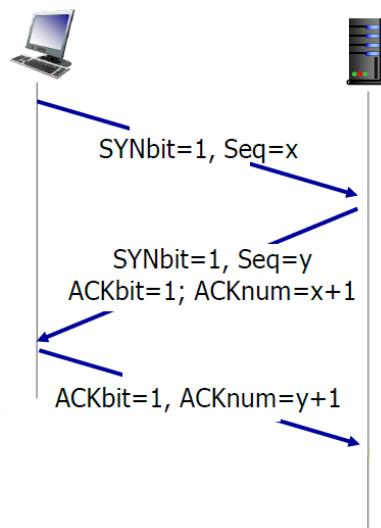


Figura 18.3: Struttura dell'handshake a tre vie

Nella prima fase il client stabilisce una connessione con un server. Il client invia un segmento **SYN**, o **Synchronize**, contenente il numero di sequenza iniziale generato casualmente da parte del client. Questo numero di sequenza permette di identificare e sincronizzare i dati durante la connessione.

Il server riceve il segmento **SYN** del client e risponde inviando un segmento **SYN-ACK** al client. Questo segmento **SYN-ACK** contiene un altro numero di sequenza iniziale generato casualmente dal server. Inoltre, il server incrementa il numero di sequenza iniziale generato da client e lo invia come **ACK** per confermare la ricezione del segmento **SYN** del client. Il client riceve il segmento **SYN-ACK** del server, incrementa il numero di sequenza di uno e invia un segmento **ACK** al server. Il segmento **ACK** conferma a sua volta la ricezione del segmento **SYN-ACK** del server.

L'handshake non è però l'unica fase, insieme al trasferimento dati, del TCP. Infatti esiste una procedura relativa alla chiusura della connessione TCP. I passi di questa operazione sono quattro e sono anche relativamente semplici: il client invia un messaggio di controllo **FIN** al server, il quale, una volta ricevuto il messaggio, risponde a sua volta con un segmento di **ACK**. A questo punto il server inizierà a chiudere la connessione e, una volta terminata, invierà un segmento **FIN** al client, il quale risponderà con un **ACK** al server. La connessione verrà effettivamente terminata solo nel momento in cui il server riceverà l'**ACK** finale del client.

## 6 Fairness

Consideriamo adesso  $K \in \mathbb{N}$  connessioni TCP, ciascuna con un differente percorso **end-to-end**, ma che hanno in comune un router intermedio con capacità trasmissiva di  $R$  bps che costituisce il collo di bottiglia del sistema. Supponiamo che ogni connessione stia trasferendo file di grandi dimensioni e che non ci sia traffico UDP attraverso il collo di bottiglia. Si dice che un meccanismo di controllo di congestione è **fair** se e solo se la velocità trasmissiva media di ciascuna connessione è approssimativamente  $R/K$ ; in altre parole, ciascuna connessione ottiene la stessa porzione di banda di collegamento. Lo studio della fairness di un algoritmo è qualcosa che, anche se all'apparenza non sembra, è fondamentale quando si crea e si sviluppa un algoritmo. Per semplicità di trattazione consideriamo il caso di due connessioni TCP che condividono un collegamento con capacità trasmissiva  $R$ . Assumiamo che le connessioni abbiano uguali valori di **MSS** e **RTT**, che debbano trasmettere grandi quantità di dati e che non vi siano altre connessioni TCP o datagrammi UDP che attraversano il collegamento condiviso. Immaginiamo ora di rappresentare un piano cartesiano che riporta sugli assi i throughput dei due dispositivi:

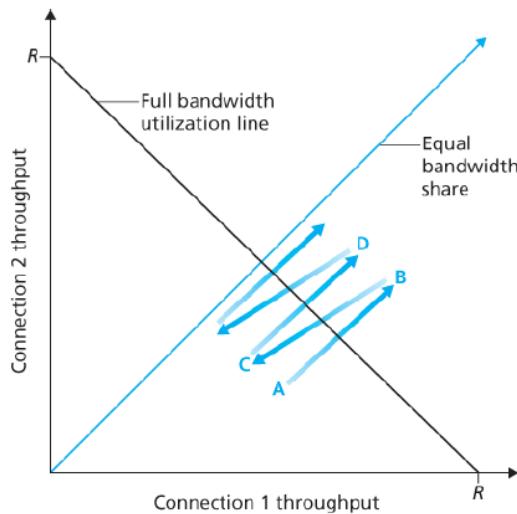


Figura 18.4: Rappresentazione del throughput dei due host su un piano cartesiano

Se TCP sta suddividendo la larghezza di banda del collegamento in modo uguale tra le due connessioni, allora il throughput dovrebbe cadere sulla bisettrice del primo quadrante. Idealmente, la somma dei due throughput dovrebbe essere uguale a  $R$ . Si nota immediatamente come il throughput dei due host si sposta lungo la bisettrice del primo quadrante aumentando di un **MSS** per volta. Nel momento in cui si verifica una congestione i due throughput vengono dimezzati, per poi ricominciare

ad aumentare in modo lineare. Quello che si può anche notare è che la suddivisione del canale è equa per entrambi i dispositivi e, quindi, possiamo dire con certezza che TCP è un protocollo **fair**.

**Fairness e UDP** Il controllo di congestione, nonostante garantisca **fairness** e **sicurezza**, potrebbe portare anche ad uno strozzamento considerevole della velocità trasmittiva del mittente. Questo fatto è anche il motivo principale per cui molte delle applicazioni multimediali in rete preferiscono ricorrere all'utilizzo del protocollo UDP come protocollo a livello di trasporto. Questo protocollo, non implementando nessun algoritmo di controllo della congestione può spedire al massimo **rate** disponibile per tutta la durata della trasmissione. Dal punto di vista del TCP, quindi, le applicazioni UDP non sono **fair**: non cooperano con altre e né adeguano la loro velocità trasmittiva in caso di congestione. Di fatto, considerato che tutte le connessioni TCP adeguano la propria velocità in funzione della congestione, potrebbero anche finire per essere strozzate dai trasferimenti UDP della rete, che potrebbero invece saturare il mezzo trasmittivo. Una delle idee adottate dai progettisti TCP è quella di inizializzare diverse connessioni parallelo per trasferire il contenuto delle pagine.

## Parte VI

# Sicurezza delle reti

# Capitolo 19

## Introduzione

Fino ad ora abbiamo visto il funzionamento della rete nel suo aspetto più fisico, cioè nell'aspetto che riguarda come funzionano i protocolli e l'infrastruttura che vi si trova alla base. Ipotizziamo ora di avere due host, Bob e Alice, che vogliono comunicare in rete; per semplicità di trattazione supponiamo di ignorare l'infrastruttura di rete tra i due host limitandoci all'astrazione offerta dai socket. Per quanto abbiano visto fino ad ora Bob invia il messaggio ad Alice sulla rete senza alcuna forma di cifratura o di modifica, invia il messaggio così come lo ha creato. Questa visione però si basa su un assunto ideale: tutti gli utenti sulla rete hanno uno scopo non malevolo, assunto che però sappiamo essere ben lontano dalla realtà di internet. Rimuoviamo questa assunzione e ipotizziamo ora che un terzo utente, chiamato **Rudy**, si immetta nella comunicazione tra **Bob** e **Alice**, intercettando il messaggio, modificandolo e inoltrandolo nuovamente verso **Alice**. Si crea quindi una situazione in cui Alice riceve un messaggio opportunamente alterato, senza possibilità di sapere però che il messaggio differisce dall'originale. A tal proposito, fin dagli albori di internet, si è sviluppata una scienza vera e propria, nota come **sicurezza informatica**, che ha come scopo quello di sviluppare tutta una serie di metodi e meccanismi affinché sia possibile mantenere questa comunicazione sicura e inattaccabile. Con sicurezza intendiamo come sia possibile comunicare in modo sicuro tra due attori della stessa rete senza che un ipotetico utente malintenzionato possa intercettare la comunicazione leggendo e/o alterandolo il messaggio. In altri termini, vorremmo che la nostra comunicazione in rete rispetti 4 proprietà:

- **Riservatezza.** Solo mittente e destinatario dovrebbero essere in grado di comprendere il contenuto del messaggio trasmesso. A tale scopo esiste un meccanismo, noto come **cifratura**, che permette di nascondere il messaggio a utenti non autorizzati.
- **Integrità del messaggio.** Oltre all'autenticazione di mittente e destinatario, occorre che il contenuto della comunicazione non subisca, durante la trasmissione, alterazioni dovute a cause fortuite o a manipolazioni.
- **Autenticazione.** Mittente e destinatario devono essere reciprocamente sicuri della loro identità, cioè devono poter confermare che l'altra parte sia effettivamente chi dichiara di essere.
- **Sicurezza operativa e disponibilità.** Nel caso in cui la comunicazione avvenga tra client e server è necessario che le risorse del server siano sempre disponibili e consistenti per tutta la durata della comunicazione.

# Capitolo 20

## Principi della crittografia

Come abbiamo anche descritto nel capitolo scorso, parlando dell'esempio di Alice e Bob, gli utenti in rete non sono sembra benevoli, ma talvolta possiamo trovare anche utenti malevoli, il cui scopo è quello di rubare dei dati o di mettere offline un servizio. Nel caso di Bob e Alice, l'utente malevolo era Rudy, che ponendosi nel mezzo della comunicazione cercava di rubare il messaggio che i due utenti si stavano scambiando. Se Bob e Alice non prendessero le giuste precauzioni, rendendo sicuro il canale, Rudy avrebbe la vita facile a rubare il messaggio: è dunque necessario che venga creato un canale **end-to-end** sicuro tra i due host, che cifri il messaggio in modo da renderlo nascosto a utenti malintenzionati.

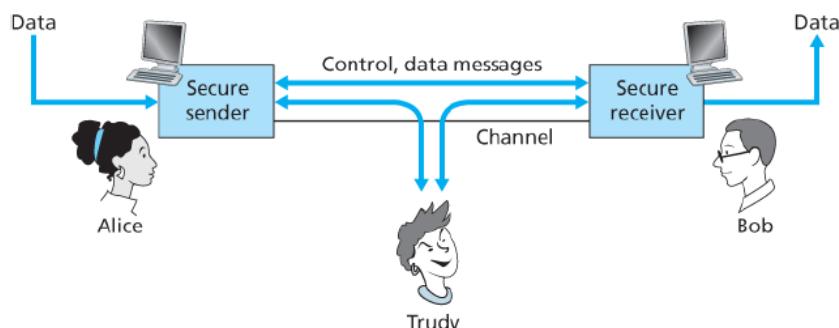


Figura 20.1: Mittente, destinatario e l'intruso

La crittografia è, ad oggi, definita come quella scienza che studia tutti quei mezzi che rendano la comunicazione tra due entità come **intellegibile** solo ad alcuni host che ne sono autorizzati. Il problema centrale di questa scienza è schematizzabile nell'esempio di Alice e Bob: **Bob** vuole comunicare con **Alice** su un canale **insicuro**, trattandolo però come se fosse un canale sicuro. Per proteggere la comunicazione i due agenti devono adottare un metodo di cifratura che permetta a **Bob** di spedire un *messaggio m*, detto **testo in chiaro**, sotto forma di *crittogramma c*, detto **messaggio cifrato**, incomprensibile a un ipotetico critto-analista X in ascolto sul canale, ma di cui sia facile la decifrazione da parte di **Alice**. Se definiamo come  $M$  come lo **spazio dei messaggi**, tale per cui  $m \in M$  e  $C$ , tale per cui  $c \in C$ , come lo **spazio dei crittogrammi**, possiamo definire le operazioni di *cifratura* e *decifratura* come due funzioni tali per cui:

$$\begin{array}{ll} \text{Cifratura del messaggio} & \mathcal{C} : MSG \rightarrow CRT \\ \text{Decifrazione del messaggio} & \mathcal{D} : CRT \rightarrow MSG \end{array}$$

Nella prima operazione si trasforma un generico *messaggio m* in **chiaro** in un *crittogramma c* applicando una funzione  $\mathcal{C}$ . Nella seconda operazione si ricava il *messaggio m* a partire dal *crittogramma c* applicando una funzione  $\mathcal{D}$ . La crittografia è quindi l'unico modo in cui è possibile

rendere un canale non sicuro, sicuro. In pratica, le tecniche di crittografia consentono al mittente di mascherare i dati in modo che un intruso non possa comprenderne il contenuto, a patto però che il ricevente sia in grado a sua volta di poter decifrare il messaggio recuperando i dati originali.

## 1 Chiavi

Abbiamo quindi visto che attraverso opportuni algoritmi di **cifratura** e **decifratura** è possibile riuscire a criptare un generico **messaggio in chiaro**  $m$  in un **messaggio cifrato**  $c$ . Tuttavia gli algoritmi di cifratura, soprattutto quelli ad uso civile, sono standardizzati e pubblicamente disponibili a tutti attraverso dei particolari RFC. Quindi, affinché questo genere di algoritmi possa essere usato in modo sicuro è necessario che, abbinato all'algoritmo, esista qualcosa che permetta, una volta applicato l'algoritmo, un messaggio che sia criptato anche nel caso in cui sia noto il funzionamento di quest'ultimo. Questo strumento aggiuntivo prende il nome di **chiave**; definiamo una generica chiave  $K_i$  che, applicata a un algoritmo di cifratura, permette di ottenere un messaggio cifrato  $c$  tale per cui, data una chiave  $K_j \neq K_i$ , si ha che  $K_j(m) \neq K_i(m)$ . Una volta definito il concetto di chiave è possibile distinguere i due tipi di algoritmi di cifratura

- **Algoritmi di cifratura a chiave simmetrica.** In questa categoria di algoritmi di cifratura la chiave  $K_a$  del mittente deve essere uguale alla chiave  $K_b$  del destinatario.
- **Algoritmi di cifratura a chiave pubblica.** In questa categoria di algoritmi di cifratura ogni attore della comunicazione ha sia una **chiave pubblica** e sia una **chiave privata**.

## 2 Crittografia a chiave simmetrica

La prima tipologia di algoritmi di cifratura sono quelli a **chiave simmetrica**, cioè quegli algoritmi dove mittente e destinatario hanno la medesima chiave, che viene utilizzata sia per la cifratura e sia per la decifratura del messaggio. La storia di questi algoritmi è antichissima e risale, addirittura, ai tempi dei romani. La tipologia di sistemi crittografici utilizzati in tempi antichi era detta **per sostituzione**, poiché ogni lettera veniva sostituita con un'altra lettera scelta, attraverso un algoritmo, tra quelle disponibili. La versione più elementare è quella in cui ogni lettera viene sostituita, in modo circolare, con quella che si trova a  $n$  posizioni successive

$$\text{Nuova lettera} = (\text{Lettera messaggio} + n) \% \text{ Numero di Lettere dell'alfabeto}$$

In epoca moderna possiamo distinguere due grandi classi di cifratura a chiave simmetrica: **cifrari a flusso** e **cifrari a blocchi**. Nei cifrari a blocchi si utilizzano diversi cifrari per sostituzione  $C_1, \dots, C_k$  per ogni lettera del messaggio; ad esempio, si potrebbe cifrare la prima cifra con il cifrario  $C_2$ , la seconda con  $C_5$ , la terza con  $C_4$  e così via, ricominciando dal primo della sequenza una volta terminata la sequenza di cifrari. L'approccio di tipo stream è invece tipico delle tecnologie come **Bluethoot**, **TLS** e Connessioni wireless in generale. Esistono tre tipologie di cifrari a blocchi comunemente utilizzati:

- **DES** (*Data Encryption Standard*). Questo algoritmo definito in uno standard del **NIST** del 1993, sfrutta blocchi di dimensione di 64 bit e una chiave simmetrica di solo 56 bit.
- **3DES** (*3-Data Encryption Standard*). Questo algoritmo utilizza una tripla cifratura **DES** dei blocchi, in modo da rendere più sicuro l'algoritmo.
- **AES** (*Advanced Encryption Standard*). Questo algoritmo sfrutta blocchi blocchi di 128 bit e chiavi di 128, o 192 o 256 bit.

L'unico modo per un possibile utente malevolo di bucare questi algoritmi è quello di provare con degli attacchi a **forza bruta**, cioè provare tutte le possibili combinazioni della chiave fintanto che non si riesce a decifrare il messaggio. Ovviamente, maggiore è la lunghezza della chiave e maggiore sarà la difficoltà nell'identificarla con un attacco di forza bruta, basti pensare al fatto che nel caso dell'**AES**, nel peggiore dei casi, quindi provando tutte le combinazioni possibili, sarebbero necessari miliardi di anni prima di trovare la chiave.

**Problema dello scambio di chiavi** Il problema principale derivante dall'utilizzo di algoritmi di crittografia a chiave simmetrica deriva dalla necessità tra **sender** e **receiver** di scambiarsi opportunamente la chiave senza che un potenziale utente malevolo possa riuscire a capirla. Esistono fondamentalmente tre approcci per risolvere questo problema

- Scambiarsi la chiave incontrandosi fisicamente.
- Utilizzare un sistema di cifratura a chiave pubblica per cifrare la chiave.
- Ricorrere a un **Key Distribution Center**.

I primi due approcci sono, evidentemente, sconvenienti per ovvi motivi. Infatti se le due persone fossero a distanze importanti sarebbe impossibile incontrarsi di persona per scambiarsi le chiavi. Inoltre, utilizzare un sistema a chiave pubblico per cifrare una chiave e usare successivamente quello privato sarebbe computazionalmente sconveniente, tanto varrebbe usare direttamente quello a chiave pubblica. Il terzo approccio, che è anche quello più conveniente, si basa sull'utilizzare un'organizzazione fidata, nota come **Key Distribution Center**. Ipotizziamo che Alice e Bob vogliano comunicare, ma che nessuno dei due conosca la chiave privata dell'altro. A questo punto Alice contatta il KDC, comunicando di voler comunicare con Bob, includendo anche un **nonce** all'interno della richiesta. Il KDC riceve a questo punto la richiesta di Alice, genera una **chiave di sessione** e la manda ad Alice, includendo anche un ticket cifrato con la chiave di Bob, che contiene a sua volta la chiave di sessione e l'identità di Alice. A questo punto Alice manda il messaggio a Bob, che lo decifra, ottenendo così sia l'autenticazione di Alice e sia la chiave di sessione per comunicare.

### 3 Crittografia a chiave pubblica

Questa tipologia di algoritmi di cifratura, rispetto agli algoritmi di cifratura a chiave simmetrica, rimuove completamente il problema relativo alla necessità di scambio della chiave tra i due interlocutori. Infatti, nella cifratura a chiave pubblica ogni interlocutore ha due chiavi:

- Una **chiave pubblica** ( $K^+$ ), accessibile da chiunque, con lo scopo di cifrare il messaggio da inviare all'interlocutore relativo.
- Una **chiave privata** ( $K^-$ ), non accessibile, con lo scopo di decifrare il messaggio cifrato con la chiave pubblica una volta giunto a destinazione.

Per comunicare con Bob, Alice prima di tutto si procura la chiave pubblica di Bob e codifica il messaggio in chiaro utilizzando tale chiave e un dato algoritmo di cifratura, generando quindi un messaggio cifrato  $K_B^+(m)$ . A questo punto Alice può inviare il messaggio a Bob che, quando lo riceverà, potrà ricavare il messaggio originale utilizzando la chiave privata e un algoritmo di decifratura, vale quindi che

$$m = K_B^-(K_B^+(m))$$

Tuttavia l'utilizzo di un sistema di cifratura a chiave pubblica introduce una complessità maggiore di quella utilizzata per un corrispettivo sistema di cifratura a chiave privata. Infatti, ad oggi, si utilizza un approccio misto, come quello dell'RSA, che permette di calcolare la chiave sfruttando un meccanismo a chiave pubblica e, una volta calcolata la chiave per entrambi gli host, usare un meccanismo a chiave simmetrica.

### 4 Integrità dei messaggi e firma digitale

Un altro degli aspetti fondamentali della sicurezza informatica è quello relativo al concetto di **integrità dei messaggi**, tema noto anche come **autenticazione dei messaggi**. Possiamo formalizzare il problema della dell'integrità dei messaggi tornando all'esempio di **Alice** e **Bob**. Supponiamo infatti che Bob riceva un messaggio, che può essere cifrato o in chiaro, e che creda che sia stato inviato da Alice. Bob a questo punto dovrà verificare due cose:

- Che il messaggio sia stato effettivamente inviato da Alice e non da qualche altro utente sotto falsa identità.
- Che il messaggio, assunto che sia stato inviato da Alice, non sia stato alterato durante il tragitto verso Bob.

Le componenti fondamentali per assolvere a questo compito sono quelle che in gergo vengono chiamate **funzioni hash crittografiche**. Una funzione **hash** è una funzione  $H : X \rightarrow Y$  che prende un ingresso una stringa generica  $m$ , e calcola una **stringa di lunghezza prefissata**, detta **hash**. Ogni funzione hash deve rispettare tre caratteristiche fondamentali

- Per ogni elemento  $x \in X$  è computazionalmente facile calcolare  $f(x)$ .
- Per la maggior parte degli elementi di  $y \in Y$  è computazionalmente difficile determinare un  $x$  tale che  $f(x) = y$ .
- È computazionalmente difficile trovare due messaggi  $x_1, x_2 \in X$  tale per cui  $H(x_1) = H(x_2)$ .

La prima proprietà è fondamentale affinché le funzioni hash non generino un overhead troppo grande rispetto al messaggio, mentre le altre due proprietà servono per scopi puramente crittografici. Attraverso queste proprietà si garantisce l'impossibilità, per un malintenzionato, di poter sostituire un messaggio con un altro messaggio che sia protetto dalla funzione **hash**. In termini matematici, dobbiamo fare in modo che la funzione di **hash** sia strettamente biunivoca. Anche il chermes di internet ha delle proprietà tipiche delle funzioni di hashing, ma si può facilmente dimostrare che l'ultima caratteristica non viene rispettata

Message	ASCII				Checksum
	Representation				
I O U 1	49	4F	55	31	
0 0 . 9	30	30	2E	39	
9 B O B	39	42	4F	42	
	B2	C1	D2	AC	

Message	ASCII				Checksum
	Representation				
I O U 9	49	4F	55	39	
0 0 . 1	30	30	2E	31	
9 B O B	39	42	4F	42	
	B2	C1	D2	AC	

Figura 20.2: Esempio di checksum. Si nota immediatamente come il checksum rimanga invariato anche se i messaggi sono diversi.

Due messaggi, differenti tra di loro, hanno lo stesso **checksum**. L'algoritmo di controllo dell'integrità usato al livello di trasporto è quindi inefficace per quanto riguarda l'autenticazione del messaggio. Tuttavia, considerato il livello a cui viene utilizzato non è, di fatto, un problema. Ad oggi esistono due algoritmi utilizzati nell'ambito dell'hashing dei messaggi: **MD5** e **SHA**. Il primo, detto **Message Digest 5**, è un algoritmo in grado di calcolare una **hash** di 128 bit con un processo a 4 fasi. Il secondo algoritmo, detto **secure hash algorithm**, calcola un hash a 160 bit. Attraverso queste funzioni hash crittografiche è quindi possibile garantire l'integrità del messaggio in modo estremamente facile; ipotizziamo, nuovamente, che Alice e Bob vogliano comunicare:

- Alice crea un messaggio  $m$  e calcola la stringa  $h = H(m)$ .
- Alice aggiunge  $h$  al messaggio  $m$ , creando il messaggio esteso  $(m, h)$  e lo manda a Bob.
- Bob riceve il messaggio esteso  $(m, h)$  e calcola  $H(m)$ . Se  $H(m) = h$ , Bob conclude che il messaggio è integro.

Un approccio di questo tipo presenta una falla non indifferente: non è infatti previsto alcun meccanismo di autenticazione. Supponiamo infatti che **Trudy** intercetti il messaggio di Alice  $(m, h)$  e ne legga il contenuto; **Trudy** potrebbe ora creare un messaggio alterato  $m_f$ , con rispettivo hash  $h_f = H(m_f)$ , comporre il messaggio  $(m_f, h_f)$  e, infine, inviare tale messaggio a Bob. Bob, una volta ricevuto il messaggio da **Trudy**, controlla attraverso la chiave hash crittografica se il messaggio è stato alterato o meno, senza però poter capire se il messaggio arriva da Alice o da **Trudy**. Per garantire, oltre all'**integrità**, anche l'**autenticazione** del mittente è necessario aggiungere un ulteriore tassello a quanto detto fino ad ora: si utilizza una **chiave di autenticazione**. Usando questo *segreto condiviso* tra Bob e Alice, l'integrità del messaggio viene realizzato nel seguente modo:

- Alice crea un messaggio  $m$ , concatena  $s$  (cioè il segreto condiviso) con  $m$  per creare una stringa di  $s + m$  bit, calcola la hash  $H(m + s)$ . La stringa ottenuta è detta **codice di autenticazione del messaggio (MAC, Message Authentication Code)**.
- Alice aggiunge il **MAC** al messaggio  $m$ , creando il messaggio esteso  $(m, H(s+m))$ , e lo manda a Bob.
- Bob riceve il messaggio esteso  $(m, h)$  e, avendo ricevuto  $m$  e conoscendo  $s$ , calcola il MAC  $H(s+m)$ . Se  $H(s+m) = h$ , Bob conclude che va tutto bene.

La presenza della **chiave di autenticazione** crea però un problema, analogo a quello anche visto nei sistemi di crittografia a chiave **simmetrica**, cioè il problema della **distribuzione della chiave**.

## 4.1 Firma digitale

Nel mondo digitale, per indicare il titolare o il creatore di un documento, o dichiarare di essere d'accordo con il suo contenuto, si ricorre alla **firma digitale**, una tecnica di crittografia che consente di raggiungere svariati obiettivi. Così come la firma tradizionale, anche la firma digitale deve rispettare tutta una serie di requisiti:

- La firma è **autenticata** e **non falsificabile**, dunque costituisce priva che chi l'ha prodotta è veramente colui che ha sottoscritto il documento.
- La firma **non è riutilizzabile**, e quindi risulta legata strettamente al documento su cui è stata apposta.
- Il documento firmato **non è alterabile**, e quindi chi ha prodotto la firma è sicuro che questa si riferirà solo al documento sottoscritto nella sua forma originale.
- La firma **non può essere ripudiata** da chi l'ha apposta, e costituisce dunque prova legale di un accordo o di una dichiarazione contenuta nel documento.

Una versione **digitale** della firma è quindi fondamentale quando si opera sulle reti dove si scambiano quotidianamente innumerevoli transazioni. Esistono diversi approcci per progettare la firma digitale; un'idea, ad esempio, potrebbe essere quella di utilizzare il **MAC** come firma digitale. Però ci si rende conto, anche abbastanza facilmente, che la chiave non sarebbe unica, poiché replicata tra Bob e Alice; ciò potrebbe portare ad una situazione dove Bob firma per Alice e viceversa.

Un approccio sicuramente più efficacie è quello di utilizzare la **cifratura a chiave pubblica**. Come abbiamo visto in questa tipologia di cifratura Bob possiede una **chiave pubblica** e una **chiave privata** che sono **uniche** per lui. Ricordiamo anche che l'utilizzo della chiave pubblica come **firma digitale** sfrutta la seguente relazione matematica:

$$K_B^-(K_B^+(m)) = m = K_B^+(K_B^-(m))$$

Supponiamo ora che Bob voglia inviare un messaggio ad Alice. Il primo passo viene fatto da Bob, che prende il messaggio e utilizza la sua chiave pubblica e un qualsiasi algoritmo di cifratura  $c(m, k^-)$  per creare la stringa  $K_B^-(m)$ . Successivamente Bob compone il messaggio  $(m, K_B^-(m))$  e lo invia ad Alice. Alice, a questo punto, conoscendo la chiave pubblica di Bob, dovrà calcolare  $K_B^+(K_B^-(m))$  e nel caso in cui risulti che  $K_B^+(K_B^-(m)) = m$ , allora conclude che Bob ha inviato il

messaggio. L'utilizzo della crittografia a chiave pubblica per le firme digitali presenta il problema per cui la cifratura e la decifratura dei dati sono onerose dal punto di vista computazionale. Un approccio più efficiente è quello di prendere il messaggio  $m$ , di lunghezza arbitraria, ed elaborare un'*impronta digitale* dei dati di lunghezza prefissata  $H(m)$ . Bob, invece di firmare l'intero messaggio con  $K_B^-(m)$ , potrebbe limitarsi a calcolare solo a firmare l'impronta digitale, calcolando  $K_B^-(H(m))$ . Considerando che la lunghezza di  $H(m)$  è generalmente minore del messaggio  $m$ , lo sforzo computazionale per creare la firma digitale è sostanzialmente ridotto.

Analizzando i protocolli di firma descritti fino ad ora possiamo riscontrare una debolezza che deriva proprio da una delle caratteristiche della crittografia a chiave pubblica: le chiavi di cifratura sono pubbliche e quindi non richiedono un incontro diretto tra gli utenti per il loro scambio. Una chiave pubblica potrebbe essere infatti recuperata dalla pagina web del mittente, oppure attraverso una **email**: non vi è quindi alcuna certezza sul fatto che la chiave pubblica che sto usando sia realmente di Bob. Per evitare attacchi come questi sono nate le **Key Certification Authority**, enti preposti alla certificazione e alla validità delle chiavi pubbliche. La CA autentica l'associazione (**utente, chiave pubblica**), emettendo un **certificato digitale**. Un certificato digitale contiene alcune informazioni

Nome campo	Descrizione
Formato	Un'indicazione del formato e del numero di versione
Ente	Il nome della CA che l'ha rilasciato.
Firma	Specifica l'algoritmo utilizzato dalla CA per firmare il certificato
Nome dell'emittente	Identificativo della CA che rilascia il certificato.
Periodo di validità	Inizio e fine del periodo di validità del certificato.
Nome del soggetto	Identificativo dell'entità la cui chiave pubblica è associata al certificato.
Chiave pubblica	Chiave pubblica del soggetto.

Tabella 20.1: Parametri del certificato rilasciato da una CA

Se Alice vuole comunicare con Bob può richiedere la chiave pubblica di quest'ultimo alla CA, che risponde a **Alice** inviando il certificato digitale di **Bob**. Alice, conoscendo anche la chiave pubblica della CA stessa, può controllare anche la validità del certificato stesso controllando il periodo di validità e, soprattutto, la firma prodotta dalla CA su di esso. Un eventuale crittoanalista potrebbe anche intromettersi sulla comunicazione falsificando il certificato, tuttavia una situazione di questo tipo richiederebbe altre forme di autenticazioni e, quindi, assumiamo che sia una situazione impossibile.

**Collo di bottiglia sulla CA** Esistono alcuni casi in cui non ci si rivolge direttamente alla CA, ma è sufficiente che il certificato, certificato digitalmente da una CA, sia inviato dall'utente stesso.

## 5 Autenticazione

Con **autenticazione** di un punto terminale si intende il processo attraverso il quale una entità prova la sua identità a un'altra entità sulla rete. L'idea è quindi quella di definire un **protocollo di autenticazione** che sia basato unicamente sullo scambio di messaggi o di dati. Ipotizziamo di chiamare il nostro protocollo **ap<numero versione>** e distinguiamo quattro possibili versioni:

- ap1.0.
- ap2.0.
- ap3.0.
- ap3.1.

- ap4.0.
- ap5.0.

La prima versione, che è anche quella più elementare, cioè **ap1.0**, presuppone che Alice invii un messaggio a Bob dichiarando solamente la sua identità. Il problema di questa forma di autenticazione è che però può funzionare fintanto che due utenti si vedono dal vivo, o parlano. Nella realtà di internet una forma di autenticazione basata sul semplicemente dichiarare la propria identità è estremamente debole. Supponiamo ora che Alice abbia un indirizzo di rete conosciuto che utilizza abitualmente, Bob potrebbe verificare l'identità di Alice guardando questo indirizzo di rete sul datagramma IP che trasporta il messaggio. Questa forma di autenticazione è quella dell'**ap2.0**, anche essa è però inefficace, poiché non è particolarmente difficile creare un datagramma IP contenente un indirizzo IP artefatto e poi inviarlo attraverso il protocollo di collegamento al primo router. Una volta giunto al primo router il pacchetto sarà poi instradato verso il destinatario. L'approccio appena descritto prende il nome di **spoofing** IP, poiché richiede che un ipotetico attaccante, prima di creare il datagramma artefatto, ricavi l'indirizzo IP dell'utente di cui vuole rubare l'identità. Il terzo protocollo possibile, cioè quello dell'**ap3.0**, prevede l'utilizzo di una **password** che viene inviata al server come forma di autenticazione. Tuttavia se questa password viene inviato in chiaro è relativamente semplice per un possibile intruso intercettarla e usarla per comunicare al posto di Alice. Nella versione **ap3.1** la password non viene più inviata in chiaro, ma viene prima crittografata attraverso un meccanismo a chiave simmetrica, con la chiave condivisa tra Bob e Alice, evitando così che un possibile attacco di **spoofing** sulla rete. Tuttavia anche questo approccio non è immune ai così detti **attacchi di replica**: Trudy potrebbe infatti inserirsi nella comunicazione, registrare il messaggio cifrato di Alice e successivamente riprodurla per inviarla a Bob sostenendo di essere Alice.

Quindi, abbiamo visto che sia **ap3.0** e **ap3.1** non sono del tutto immuni a dei possibili attacchi, come il **replay attack** o lo **spoofing**. Il motivo è presto detto, Bob non può stabilire chi se chi sta comunicando con lui sia effettivamente Alice o se i messaggi che stava ricevendo erano una riproduzione registrata. Possiamo risolvere il problema adottando un'idea simile a quella adottata dal protocollo TCP nell'handshake a tre vie: utilizziamo un numero che il protocollo userà soltanto una volta, detto **nonce**, nel seguente modo:

- Alice invia un messaggio di autenticazione a Bob.
- Bob sceglie un **nonce**  $R$  e lo manda ad Alice.
- Alice utilizza la chiave simmetrica segreta che condivide con Bob per codificare il **nonce** e gli invia il risultato indietro.
- Bob decifra il messaggio ricevuto: se il nonce è quello da lui inviato, allora Alice è autenticata.

Diciamo che il **nonce** è un numero che viene utilizzato **once in a lifetime**. Infatti ad ogni nuova comunicazione viene utilizzato un **nonce** diverso, rendendo difatti inefficaci anche gli attacchi di **replay**. Un ultima modifica che possiamo apportare al protocollo **ap4.0** è quella di prevedere l'utilizzo della chiave pubblica, così da eliminare anche il problema dello scambio delle chiavi.

# Capitolo 21

## Sicurezza nella rete

Nello scorso capitolo abbiamo visto i principi della scienza della **crittografia**, abbiamo studiato anche le tecnologie e i meccanismi alla base dei servizi di sicurezza che possono essere offerti e utilizzati per lo scambio di informazioni tra due host. Il discorso che dobbiamo affrontare adesso è come vengano effettivamente utilizzati queste tecnologie e questi protocolli all'interno di internet. In particolare, dobbiamo definire il **livello** a cui tali tecnologie vengono applicate e, inoltre, anche quali sono queste tecnologie. Iniziamo con il dire che quando la sicurezza viene fornita a uno specifico protocollo a livello di applicazione, l'applicazione fa uso di quel protocollo usufruirà di uno o più servizi tra **riservatezza, autenticazione e controllo di integrità**. Quando la sicurezza viene fornita a un protocollo a **livello di trasporto**, tutte le applicazioni che fanno uso di quel protocollo usufruiranno dei servizi di sicurezza di quel protocollo a livello di trasporto. Quando la sicurezza viene fornita a livello diretto tra due host, tutti i segmenti a livello di trasporto usufruiscono dei servizi di sicurezza a livello di rete. Ogni livello della rete è in grado quindi di fornire dei servizi di sicurezza in grado di garantire tutti i servizi che ci aspetterebbe da un protocollo di sicurezza; nello specifico:

- A livello applicazione troviamo tutta una serie di **librerie di funzioni**, come ad esempio il **PGP** (o **Pretty Good Privacy**, che permettono di garantire questi servizi).
- A livello di trasporto troviamo il protocollo **TLS** (o **Transport Layer Security**).
- A livello di rete troviamo il protocollo **IPsec**, che trova ampio uso all'interno delle **VPN**.

Il motivo per cui abbiamo servizi di sicurezza a diversi livelli è relativamente semplice. In primo luogo, nonostante la sicurezza a livello di rete offra un grado di protezione sufficiente, attraverso cifratura dei datagrammi, non offre però un servizio altrettanto valido a livello utente. Prendiamo, ad esempio, il caso di un sito **e-commerce**; quest'ultimo non potrà affidarsi alla sicurezza a livello IP per autenticare un cliente che sta facendo acquisti online.

### 1 Sicurezza a livello applicazione e PGP

Partiamo analizzando il funzionamento della sicurezza a livello applicazione ipotizzando di dover costruire un sistema di sicurezza per un servizio di posta elettronica sicuro su internet. Il primo passo è ovviamente quello di definire quali requisiti saranno fondamentali per la nostra applicazione, così da poter capire cosa implementare e cosa tralasciare. Supponiamo ora di dover offrire tutti i servizi che abbiamo visto nello scorso capitolo: **riservatezza**, vogliamo infatti che il messaggio tra **Alice** e **Bob** non possa essere letto da **Trudy**, **autenticazione**, **Alice** e **Bob** devono essere sicuri delle reciproche identità e **integrità**, il messaggio che va da Alice verso Bob e viceversa non dovrà essere alterato. Una volta definiti i requisiti del nostro sistema di sicurezza è importante definire anche come sviluppare il nostro sistema di sicurezza; in particolare:

- La prima possibilità è quella di costruire la propria libreria di funzioni di sicurezza direttamente all'interno del codice dell'applicazione.

- La seconda possibilità è quella di fare affidamento a una libreria di funzioni di sicurezza esterna alla nostra applicazione.

Un classico esempio di **libreria di funzioni** che mette a disposizione delle funzioni e dei servizi di sicurezza è detta **Pretty Good Privacy**: un'applicazione client-server per la posta elettronica, tra le prime a essere diffusamente impiegate in internet. Analizziamo ora come il protocollo **PGP** implementi questi servizi utilizzando l'esempio di Alice e Bob:

**Confidenzialità** Per garantire la confidenzialità, Alice sceglie arbitrariamente una chiave simmetrica  $K_S$ , cifra il suo messaggio  $m$  con  $K_S$  e codifica la chiave simmetrica con la chiave pubblica di Bob,  $K_B^+$ . A questo punto concatena il **messaggio cifrato** e **chiave simmetrica** cifrata per formare un unico messaggio  $(K_S(m), K_B^+(K_S))$  che invia all'indirizzo e-mail di Bob.

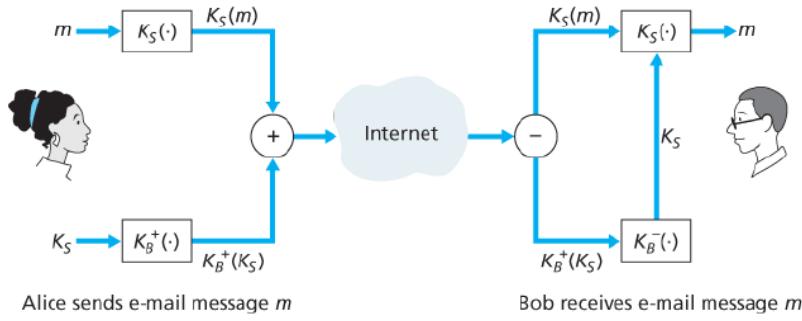


Figura 21.1: Confidenzialità nell'applicazione PGP

Quando Bob riceve il messaggio, decifra la chiave simmetrica attraverso la sua chiave privata e, successivamente, decifra con la chiave simmetrica il messaggio.

**Autenticazione del messaggio e integrità** Per garantire l'autenticazione di **mittente** e **destinatario** utilizziamo il meccanismo delle **firme digitali** e una **funzione di Hash**. In particolare, Alice applica una **funzione hash** al suo **messaggio**, cifra il risultato con la sua chiave privata  $K_A^-$ , per creare la **firma digitale**, concatena messaggio in chiaro con la firma e lo invia, come unico messaggio, all'indirizzo e-mail di Bob.

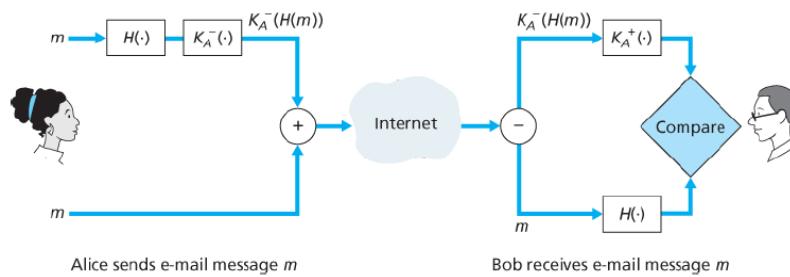


Figura 21.2: Autenticazione e Integrità nell'applicazione PGP

Quando Bob lo riceve, applica la chiave pubblica di Alice,  $K_A^*$ , all'hash del messaggio che è stata firmata e confronta il risultato con quanto ottenuto con la sua funzione di hash. Se coincidono può essere abbastanza sicuro che il messaggio provenga da Alice e che non sia stato alterato durante il tragitto.

**Approccio combinato** Unendo le due configurazioni viste in precedenza possiamo ricavare una procedura in cui Alice confeziona un *pacco* iniziale che contiene al suo interno il messaggio originale e l'hash firmato digitalmente. Successivamente genera una chiave simmetrica con la quale

cifra il messaggio precedentemente creato, cifra la chiave simmetrica con la chiave pubblica di Bob, compone un messaggio che contiene anche la chiave simmetrica cifrata e manda tutto verso Bob.

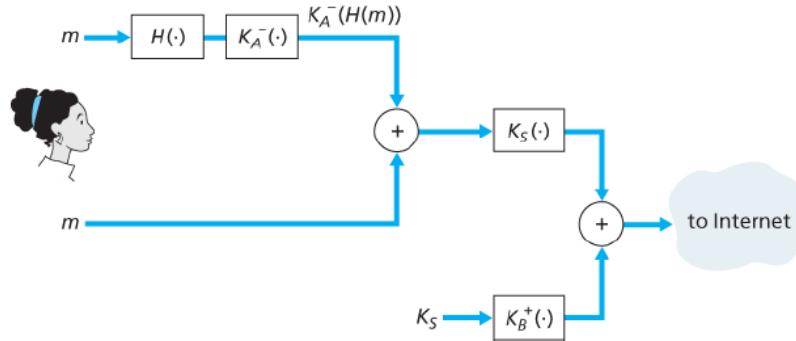


Figura 21.3: Approccio combinato nell’invio di messaggi nel PGP

Bob, una volta ricevuto il messaggio, dovrà applicare alcuni dei passaggi visti fino ad ora all’inverno, così da poter ricavare il messaggio originale e verificare che il mittente sia davvero Alice.

## 2 Sicurezza a livello di trasporto

Un secondo approccio all’implementazione dei meccanismi di sicurezza nella rete è quella di implementarli direttamente a **livello di trasporto**; in particolare, l’idea è quella di utilizzare un protocollo, che si trovi ad un livello superiore al **TCP**, che sia in grado di renderlo sicuro. Questo livello, posto tra **livello applicativo** e **livello di trasporto**, è anche detto **livello middleware**. Il protocollo utilizzato a questo livello è detto **Transport Layer Security** (o **TLS**); questo protocollo presenta un funzionamento abbastanza complesso, ragion per cui studieremo una versione semplificata del protocollo, per poi introdurre anche alcuni elementi relativi al protocollo reale. In generale TLS, appoggiandosi comunque al protocollo TCP, presenta quattro fasi diverse:

- **Handshake iniziale.**
- **Derivazione della chiave.**
- **Trasferimento dei dati.**
- **Chiusura della connessione.**

I dati che vengono scambiati dal protocollo TLS sono detti **record**. Ogni record presenta una struttura ben precisa costituita da diversi campi, tra cui: un campo **tipo** che specifica se il **record** è un messaggio **handshake** o **dato**, un campo **versione**, un campo **lunghezza** per estrarre i record TLS dal flusso dati e un campo **MAC**.

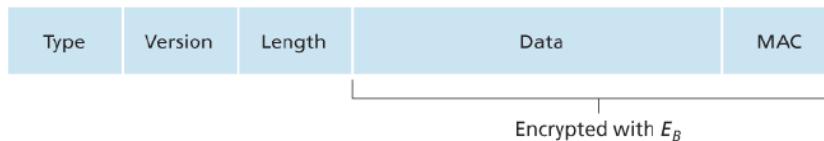


Figura 21.4: Struttura di un record TLS

Durante la prima fase, cioè quella di **handshake**, Bob ha bisogno di stabilire una connessione TCP con Alice, verificare che Alice sia realmente Alice e inviarle una chiave segreta principale che verrà utilizzata per entrambi per generare tutte le chiavi simmetriche di cui hanno bisogno per la sessione TLS. Una volta stabilita la connessione TCP, Alice e Bob, si scambieranno i rispettivi certificati rilasciati da una CA accreditata. Bob, a questo punto, genera un **master secret**: un

valore segreto che verrà usato solo per questa sessione di TLS e dal quale verranno derivate altre chiavi:

- Due chiavi per la trasmissione da Bob verso Alice.
- Due chiavi per la trasmissione da Alice verso Bob.

Il master secret, prima di essere inviato, viene cifrato con la chiave pubblica di Alice, per creare un **Encrypter Master Secret** (o EMS) che viene inviato ad Alice. Alice decifra l'**EMS** con la sua chiave privata e ottiene **MS**. A questo punto i due host possono iniziare con la fase di **derivazione delle chiavi**. Infatti nonostante il master secret possa essere usato come chiave simmetrica per le successive cifrature, è comunque più sicuro che Alice e Bob usino ciascuno chiavi crittografiche differenti; **Bob** genera quindi due chiavi: una chiave  $E_B$ , che serve per la cifratura di sessione per i dati inviati da Bob ad Alice, e una chiave **MAC** di sessione per i dati inviati da Bob ad Alice. Alice, parallelamente a Bob, farà la medesima operazione, generando le chiavi  $E_A$  e  $M_A$ . Ora che Bob sa con certezza che sta comunicando con Alice, ed entrambi condividono le stesse quattro chiavi di sessione, i due possono iniziare a scambiarsi dati in sicurezza sulla connessione TCP. Si suddivide il flusso di dati in **record** a cui aggiunge un **MAC** per la verifica dell'integrità e che poi cifra. Per creare il MAC, Bob passa i dati del record, il **type** del record e il numero di sequenza assieme alla chiave  $M_B$  a una funzione hash.

**TLS nella realtà** Come abbiamo già detto, quella descritta fino ad ora è solo una versione semplificata del protocollo TLS. Il protocollo semplificato presenta infatti alcune falle di sicurezza, non è ad esempio immune da attacchi di **replay** e né da attacchi cosiddetti di **syn**. La prima differenza rispetto al protocollo semplificato è relativa alla fase di **handshake** e riguarda il fatto che non è richiesto che i due host usino uno specifico algoritmo di cifratura a chiave simmetrica o uno specifico algoritmo di cifratura a chiave pubblica o che usino uno specifico MAC, ma viene data la possibilità di sceglierli tra una serie di opzioni disponibili in fase di **handshake**. La seconda differenza, necessaria per proteggersi dagli attacchi di tipo replay, è quella di introdurre l'utilizzo di un **nonce**, così da garantire la **freschezza** del messaggio. Riassumendo, alcuni dei passi del reale handshake sono:

- Il client invia, insieme al **nonce**, la lista degli algoritmi crittografici da lui supportati.
- Il server decide, dalla lista, un algoritmo a chiave simmetrica, uno a chiave pubblica e un algoritmo MAC. Restituisce poi al client le sue scelte, insieme ad un nonce e al suo certificato.
- Il client verifica il certificato, estrae la chiave pubblica del server, genera un **pre-master secret**, lo cifra con la chiave pubblica del server e lo invia.
- Client e Server, utilizzando un algoritmo di derivazione della chiave, in combinazione con il **nonce** e il **PMS**, generano il **master secret** effettivo.
- Client e Server si comunicano i **MAC** di tutto l'handshake, così da evitare un possibile attacco **man in the middle**. In particolare, alla fine di tutta la comunicazione, il client calcola l'hash di tutti i messaggi che client e server si sono scambiati, ottenendo un **MAC**. A questo punto il server calcolerà il suo **MAC** e farà una comparazione con quello ricevuto dal client.

I **nonce** sono utilizzati per evitare attacchi di tipo reply: cambiando il **nonce** cambieranno le chiavi crittografiche, ed essendo i **nonce** casuali e diversi per ogni sessione, non è possibile usare vecchi messaggi per instaurare una nuova sessione. Esistono anche altre fasi dell'handshake TLS reale, ma non sono state affrontate durante il corso. La seconda differenza tra **TLS** semplificato e **TLS** reale è relativa alla chiusura della connessione. Sappiamo, dalla teoria del TCP, che per terminare la connessione è sufficiente che Bob invii un segmento **TCP-FIN** ad Alice. Tuttavia un approccio di questo tipo presenta una falla di sicurezza non indifferente: Trudy si pone nel mezzo della connessione TLS e la termina prematuramente con un messaggio TCP-FIN. In questo modo Trudy potrebbe fare credere ad Alice di aver ricevuto tutti i dati da Bob, quando in realtà non è vero. Per proteggere la connessione da questo tipo di attacchi si introduce un vincolo di chiusura; la connessione TCP può essere chiusa solo quando la connessione TLS soprastante è stata chiusa.

Il limite principale del TLS è però proprio la sua enorme complessità in termini di operazioni. Infatti, essendo comunque un protocollo che si appoggia su TCP, sono comunque presenti tutte quelle fasi e meccanismi (come il controllo di flusso e di congestioen) che caratterizzano il TCP. Un approccio alternativo potrebbe essere quello di utilizzare un diverso protocollo a livello di trasporto, come l'**UDP**, abbinato a un protocollo, anch'esso soprastante, che implementa le funzionalità che in precedenza erano compito del TCP più la sicurezza; un esempio di protocollo di questo è il **protocollo QUIC**.

### 3 IPsec e reti virtuali private

Un ulteriore modo di garantire la sicurezza è quella di fornirla direttamente a livello di rete. Il protocollo **IPSec**, o **IP Secure**, è un protocollo che rende sicuri i datagrammi IP tra due entità come host e router. Questo protocollo viene ampiamente utilizzato da diverse organizzazioni per creare delle **Virtual Private Network** che però fanno uso della rete internet pubblica. **IPSec** è quindi un **protocollo di sicurezza a livello di rete**, che garantisce tutti i servizi offerti anche da gli altri protocolli visti fino ad ora:

- **Riservatezza dei dati.** Il campo dati di ogni pacchetto in uscita dovrà essere quindi cifrato, indipendentemente dal tipo di pacchetto contenuto al suo interno.
- **Autenticazione della sorgente/destinazione.**
- **Integrità dei dati.**

Oltre a questi servizi, IPSec fornisce anche dei meccanismi per la difesa da attacchi di tipo **replay** o **man in the middle**.

Ipotizziamo ora che un'istituzione, sparsa in diverse parti del mondo, manifesti il desiderio di fare in modo che i propri host e i propri server possano comunicare in modo sicuro e riservato. L'idea più elementare potrebbe essere quella di installare una rete fisica indipendente, staccata dalla rete internet pubblica, dotata di router, server DNS e DHCP; questo tipo di rete prendono il nome di **reti private**. L'idea presenta però un problema, ha un costo che è proporzionale alla dimensione della rete di cui abbiamo bisogno, il che, supponendo una rete molto grande, corrisponde a un costo molto alto. Quindi, invece che installare una rete privata, si cerca di creare una **rete virtuale privata** che però si appoggi alle infrastrutture della rete pubblica.

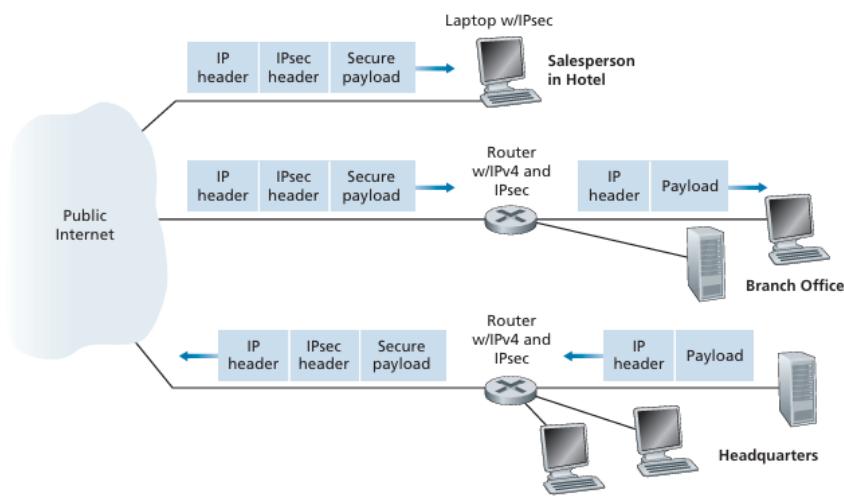


Figura 21.5: Esempio di **Virtual Private Network**

Una VPN è quindi in grado di copperire a tutte le caratteristiche che sono desiderabili all'interno di una rete privata, mantenendo un costo ridotto grazie al fatto che si appoggia alla **rete pubblica**. IPSec, protocollo alla base delle VPN, è un protocollo molto complesso, definito da circa 12 RFC, costituito da diversi protocolli e meccanismi. Due sono i principali protocolli della suite IPsec:

- Il protocollo **Authentication Header**. Il protocollo AH fornisce solo l'autenticazione della sorgente e l'integrità dei dati.
- Il protocollo **Encapsulation Security Payload**. Il protocollo **ESP** fornisce, oltre che autenticazione e integrità, anche la riservatezza dei dati.

Quindi, l'idea alla base delle VPN è che i pacchetti, fintanto che girano all'interno delle reti dell'organizzazione, lo facciano *in chiaro*. Nel momento esatto in cui però devono transitare all'interno della rete pubblica, una volta giunti all'interno del router, vengono cifrati e inoltrati sulla rete pubblica. Prima di procedere con l'invio dei datagrammi sicuri tra mittente e destinatario è necessario creare un canale logico, **unidirezionale**, a **livello di rete**, chiamato **Security Association**. Queste **associazioni di sicurezza**, che essendo connessioni sono **connection-oriented**, sono identificate da alcuni dati che permangono all'interno dei dispositivi che implementano il canale

Campo	Descrizione
SPI	Detto anche <b>Indice dei Parametri di Sicurezza</b> , è un identificatore a 32 bit.
Interfaccia di origine della SA	Indirizzo del router di origine
Interfaccia di destinazione della SA	Indirizzo del router di destinazione
Tipo di codifica utilizzata	Il tipo di crittografia utilizzata, ad esempio 3DS o CBC.
Chiave di codifica	Chiave
Controllo di integrità	Il tipo di algoritmo di hashing utilizzato, ad esempio <b>SHA</b> o <b>MD</b> .
Chiave di autenticazione	

Tabella 21.1: Identificatori della **security association**

Il router mittente, ogni volta che costruisce un datagramma IPsec da inoltrare su una delle sue **SA**, accede alle relative informazioni di stato per determinare come autenticare e cifrare il datagramma. Analogamente, il router mittente, mantiene le stesse informazioni di stato per questa **SA** per autenticare e decifrare tutti i datagrammi IP.

### 3.1 Il datagramma IPsec

IPsec prevede due forme diverse di pacchetto: quella chiamata **tunnel**, dove il pacchetto viene incapsulato all'interno di un vero e proprio **datagramma IPsec** e una modalità **trasporto**, dove avviene solo la cifratura del payload del messaggio. Essendo la prima versione quella più appropriata per le **VPN**, sarà anche quella a cui faremo riferimento. Analizziamo quindi la struttura del datagramma IPsec:

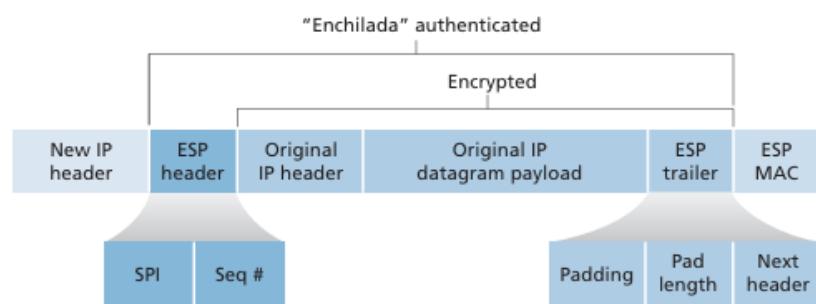


Figura 21.6: Formato del datagramma IPsec

Per ricavare questo datagramma a partire da un datagramma IPv4 si deve passare da una serie di

passi. Il primo passo è quello di appendere in fondo al datagramma IPv4 un campo **coda ESP**. Il secondo passo è quello di cifrare il risultato usando l'algoritmo e la chiave specifica all'interno della **SA**. Il terzo passo è quello di appendere all'inizio dei dati cifrato un campo chiamato **intestazione ESP**, il pacchetto così ottenuto è detto **enchilada**. Si crea quindi un'autenticazione **MAC** sull'**enchilada** utilizzando l'algoritmo e la chiave specificata nell'**SA**. Il penultimo passo è quello di appendere il **MAC** alla fine dell'**enchilada**, ottenendo il **payload**. Infine, si crea una nuova intestazione IP con tutti i campi di intestazione classici di **IPv4**, lunghi normalmente 20 byte e la appende prima del payload calcolato al punto precedente. Il primo campo contenuto all'interno dell'header ESP è, ovviamente, il **Security Parameter Index** relativo all'**SA** del pacchetto. Il secondo campo è invece un numero di sequenza utilizzato per prevenire eventuali attacchi di replay. All'interno del trailer è invece contenuto

- Un campo **padding**, di 8 bit, utilizzato come riempimento nel caso in cui la dimensione del blocco da cifrare sia minore di quella richiesta dall'algoritmo di cifratura.
- Un campo **next header**, sempre di 8 bit, che specifica cosa c'è dentro il pacchetto ESP cifrato (pacchetto UDP, TCP, ICMP).
- Un campo **pad length**, anch'esso di 8 bit, che specifica la lunghezza del padding.

Considerato quindi la grandezza dell'header ESP, della

### 3.2 IPsec security database

Ogni entità IPsec memorizza le informazioni di stato di tutte le loro **SA** all'interno di un database, che risiede nel kernel del sistema operativo, chiamato **database delle associazioni di sicurezza (SAD, Security Association Database)**. Ogni **entrata** di questo **databse** è indicizzata da un indice, chiamato **security parameter index** e contiene al suo interno tutte le informazioni già specificate in precedenza. Un'altra importante questione riguarda il comportamento che il router quando riceve un datagramma non sicuro da un host all'interno della VPN per un qualsiasi host esterno; nello specifico, se il datagramma debba essere convertito in un diaigramma IPsec o meno e, nel caso in cui venga convertito, quale SA utilizzare. Il problema viene risolto facendo in modo che l'entità IPsec, insieme al SAD, mantenga anche un **Security Policy Database**. Questo database indica che tipi di datagrammi debbano essere elaborati con IPsec in funzione degli indirizzi IP di sorgente e destinazione e del tipo di protocollo.

## 4 Approfondimento TLS

Nell'ultimo punto sull'handshake reale del protocollo TCP è stato accennato il passo tra client e server di comunicarsi il rispettivo MAC di tutto l'hanshake. Il motivo di questa scelta deriva dalla necessità di proteggersi da un possibile  **downgrade attack**; ipotizziamo infatti la seguente situazione:

- Il client manda la sua cipher suite in chiaro.
- Un malintenzionato si inserisce all'interno della comunicazione e inserisce all'interno della cipher suite, al posto di AES, DES.
- Il server, non avendo altre possibili scelte, adotta DES come algoritmo di cifratura.

Il malintenzionato è quindi riuscito a downgradare la trattativa tra client e server, portando all'utilizzo di un algoritmo di cifratura facilmente rompibile attraverso un attacco di brute force. Alla fine della comunicazione, quindi, mittente e destinatario creano una sorta di **MAC**, chiamato anche **verify\_data**, composto nel seguente modo

```
verify_data = PRF(Master Secret, "Client Finished", H(tutti i messaggi scambiati))
```

Dove PFR è una Pseudo-Random Function e H è una funzione di hash. Una volta che entrambi hanno calcolato il rispettivo **MAC** per tutto l'**hanshake**, possono procedere a scambiarselo con le chiavi generate attraverso il master secret e a confrontarlo. Se i due MAC differiscono allora l'handshake non è da ritenersi valido.

# Capitolo 22

## Sicurezza operativa delle reti

Abbiamo visto nel corso del capitolo che internet è un ambiente tutt'altro che sicuro, dove i malintenzionati sono in attesa di poter rubare dati o rendere inaccessibili dei servizi. Fino ad ora ci siamo occupati principalmente del primo problema, andando a definire tutti quei metodi e protocolli con cui è possibile garantire **confidenzialità**, **autenticazione** e **integrità**. Tuttavia rimane un altro problema, cioè quello della possibilità di un malintenzionato di **negare** dei servizi, saturando quella che è la loro capacità di gestire le richieste attraverso degli attacchi di tipo **Denial-Of-Service** oppure andando a eseguire attraverso delle sequenze di pacchetti, delle azioni non autorizzate all'interno della rete. Per risolvere questi due problemi sono stati introdotti diversi sistemi di difesa, tra cui:

- Firewall.
- Sistema di rilevamento delle intrusioni.

### 1 Firewall

Il **firewall**, letteralmente **muro di fuoco**, è una **combinazione di hardware e software** che separa una rete privata dal resto di internet e consente all'amministratore di **controllare e gestire il flusso di trasferimento** tra il mondo esterno e le risorse internet, stabilendo quali pacchetti lasciare transitare e quali bloccare. Un firewall è quindi un vero e proprio muro che si pone tra una rete e il mondo esterno:

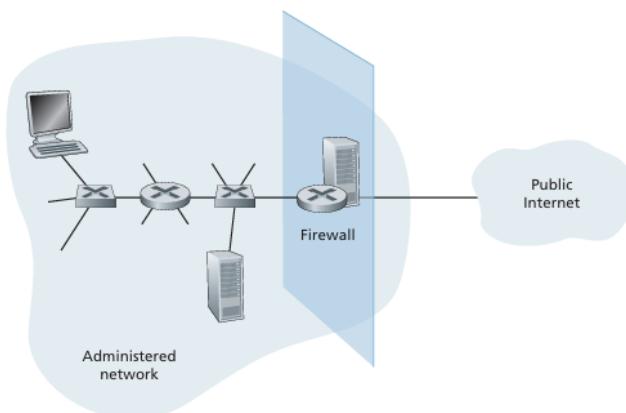


Figura 22.1: Posizionamento del firewall rispetto la rete interna e l'internet pubblico

Quindi, il firewall ha tre obiettivi: il primo è quello di fare da **unico** tramite tra il traffico che si

muove dall'esterno verso l'interno della rete e viceversa, il secondo obiettivo è quello di permettere il passaggio al solo **traffico autorizzato**, cioè solo del traffico ammesso dalle **politiche di sicurezza**, mentre il terzo obiettivo deve essere quello di essere immune dalla penetrazione. Il terzo punto può sembrare scontato, ma è fondamentale che un firewall venga installato e progettato correttamente, poiché la presenza di un firewall mal funzionante potrebbe creare un falso senso di sicurezza all'interno dell'organizzazione, che è una situazione ben peggiore di non avere proprio sicurezza.

I firewall possono essere suddivisi in tre categorie: **filtri di pacchetto (packet-filter)**, **filtri con memoria di stato (stateful-filter)** e **gateway a livello applicativo (application-level gateway)**.

### 1.1 Firewall packet-filter

Una rete privata è, generalmente, connessa alla rete pubblica del suo ISP attraverso un router. Il router è quindi quel dispositivo attraverso cui transitano tutti i pacchetti diretti in **ingresso** e in **uscita** dalla rete privata, ed è anche quel dispositivo che si occupa del **filtraggio dei pacchetti**. Questa operazione prevede due passi fondamentali:

- **Verifica.** Viene controllata l'**intestazione** del datagramma in entrata sul router.
- **Azione.** Vengono applicate delle regole di filtraggio stabilite dall'amministratore che possono accettare il **pacchetto**, oppure **rifiutarlo**.

La verifica sul datagramma può essere fatta analizzando diversi campi, tra cui **indirizzo IP** sorgente e destinazione, **tipo di protocollo**, **porta sorgente** e/o **porta destinazione**. Anche l'analisi del **flag** TCP può essere considerata nelle regole di filtraggio dei pacchetti. In generale, un amministratore di rete, configura il **firewall** secondo quelli che sono i vincoli imposti dall'organizzazione stessa, che possono anche tener conto di altre metriche come **produttività degli utenti** e **utilizzo di banda**. L'insieme delle regole specificate all'interno di un firewall è detto **Access Control List** e può essere costruita utilizzando due approcci: un **approccio pessimista** e un **approccio ottimista**. Il primo approccio, cioè quello **pessimista**, impone che tutto il traffico sia vietato tranne quello che proviene da indirizzi esplicitamente ammessi. Il secondo approccio, invece, impone che tutto il traffico sia ammesso tranne quello esplicitamente vietato. Un esempio di **ACL** di tipo **pessimista** è la seguente: INSERIRE TABELLA

## 2 Firewall stateful packet filter

In un filtro di pacchetti tradizionale le decisioni vengono prese **pacchetto per pacchetto**, in modo del tutto indipendente, portando quindi a dover ripetere le operazioni diverse volte. I filtri con **memoria di stato** tengono invece traccia delle connessioni TCP e usano questa conoscenza per prendere le decisioni di filtraggio. Ad esempio:

- Se arriva un datagram TCP che specifica una porta relativa a una connessione non più attiva allora non c'è alcun motivo per cui quel pacchetto dovrebbe essere lasciato entrare.
- Se arriva un datagram TCP che specifica una porta relativa a una connessione **attiva**, allora non c'è alcun motivo per cui quel pacchetto non dovrebbe essere lasciato passare

Questo approccio richiede che il dispositivo che si occupa del filtraggio di pacchetto deve anche tenere conto di tutte le connessioni TCP attive. La **ACL** ha quindi un campo aggiuntivo che specifica se è necessario controllare la connessione **TCP** a cui questo dispositivo fa capo.

## 3 Filtro gateway

Il filtraggio di pacchetto opera a livello di rete, ogni pacchetto che giunge all'interno del router viene analizzato singolarmente a livello di rete controllando alcuni parametri all'interno dell'intestazione. Con questo approccio è quindi possibile bloccare il traffico proveniente da una particolare porta o

da una particolare rete, ma non è possibile bloccare interamente un servizio a livello applicativo; ad esempio, nel caso in cui volessi evitare delle connessioni SSH verso l'esterno non potrei usare un **firewall packet filter**. Quindi, per discriminare utenti e servizi è necessario usare un approccio non basato su un filtraggio; in particolare, si deve installare un dispositivo, chiamato **application gateway**, che lavora proprio a livello applicativo. Ipotizziamo ora che l'utente cerchi di comunicare SSH verso l'esterno in una rete con firewall a livello applicativo:

- Per prima cosa l'utente deve instaurare una connessione SSH con l'**application gateway** inviando le sue credenziali.
- L'**application gateway** controlla che l'utente sia accreditato a utilizzare tale servizio verso l'esterno.
- In caso non sia accreditato termina la connessione, altrimenti chiede all'utente il nome dell'host a cui si vuole connettere, imposta una sessione SSH con il server indicato e inoltra a questo i dati che arrivano dall'utente e viceversa.

Il gateway non svolge solo quindi la funzione di "controllore" della rete, ma si configura come un vero e proprio intermediario tra i dispositivi e la rete esterna. Tuttavia questa tecnica non è priva di svantaggi, infatti occorre avere un gateway diverso per ogni applicazione, inoltre c'è anche un costo in termini di prestazioni in quanto tutti i dati devono passare attraverso il gateway.

## 4 Sistemi di rilevamento delle intrusioni

I firewall, indipendentemente dalla tipologia che viene utilizzata, hanno dei problemi che non possono essere trascurati. Uno di questi problemi è che il firewall non controlla **approfonditamente** anche gli altri campi del pacchetti e, inoltre, non controlla la **sequenza** dei pacchetti. Quindi, se da una parte un firewall mi consente di proteggere la rete, dall'altra non rende comunque la rete immune da attacchi del tipo **scanning DDOS**, cioè attacchi in cui i vengono inviati tanti pacchetti, all'apparenza innocui, che però cercano di fare la stessa identica operazione. Questi pacchetti potrebbero essere infatti ammessi dentro la rete e, come il cavallo di troia, compromettere uno o più servizi. Occorre quindi introdurre un nuovo dispositivo che si occupi di eseguire un controllo più approfondito sui campi dei vari pacchetti che arrivano all'interno della rete; questi dispositivi sono noti come **sistemi di rilevamento delle intrusioni (IDS, Intrusion Detection System)**. Un IDS, data la sua natura, è in grado di rilevare un'ampia gamma di attacchi, compresi:

- Mappatura della rete.
- Scansione delle porte.
- Worm
- etc.

Un'organizzazione può impiegare uno o più sistemi **IDS** nella propria rete istituzionale. In particolare, l'organizzazione che viene adottata generalmente è quella di avere un singolo firewall di tipo **packet filter** all'interno della rete e una serie di **IDS** che controllano delle **porzioni di traffico**, in modo da ridurre il carico sul singolo **IDS**. Tuttavia anche gli **IDS** hanno dei problemi: tendono a generare dei falsi positivi, che magari possono portare a un blocco della rete per un intervallo di tempo (molto piccolo generalmente). I sistemi **IDS** sono divisibili in due categorie: **sistemi basati su firme** e **sistemi basati sulle anomalie**. Un sistema basato su firme mantiene un ampio database di firme degli attacchi, cioè di insiemi di regole (firme) che riguardano un'attività di intrusione. Una firma può semplicemente essere una lista di caratteristiche di un singolo pacchetto o mettere in relazione una serie di pacchetti. Un sistema basato sulle anomalie invece mantiene dei profili di traffico, quando osserva il traffico delle operazioni normali. Guarda poi i flussi di pacchetti che sono statisticamente insoliti e agisce di conseguenza. Questi **IDS** hanno la particolarità è che non fanno affidamento sulla conoscenza di precedenti aggressioni, per cui possono potenzialmente rilevare nuovi e non documentati attacchi.

**Zona demilitarizzata** I controlli di sicurezza sono sicuramente molto utili all'interno di una rete privata, poiché permettono di limitare eventuali attacchi e di controllare il traffico in uscita e il traffico in entrata. Tuttavia, su alcuni servizi, queste regole possono essere molto restrittive; si pensi ad un web server che deve esporre il sito web dell'organizzazione. Possiamo quindi pensare di suddividere la rete in due regioni:

- Una regione a **alta sicurezza**.
- Una regione a **più bassa sicurezza**.

La regione a bassa sicurezza viene detta **zona demilitarizzata (DMZ, DeMilitarized Zone)**, la quale è protetta solo dal **packet filter**. I motivi per cui realizzare, all'interno della rete di un'organizzazione, una DMZ sono molteplici: un server HTTP che contiene il sito dell'applicazione non può essere protetto da delle regole troppo restrittive, inoltre una DMZ può permettere anche di identificare i pattern di eventuali attacchi attraverso l'introduzione di un **honeypot**.

## **Parte VII**

# **Reti wireless e mobili**

# Capitolo 23

## Wireless e reti mobili

Fino ad ora abbiamo studiato delle reti basate su un collegamento **via cavo**, dove i vari host sono **statici** e difficilmente si muovono rispetto alla posizione in cui sono originariamente posti. Nella realtà però la situazione è ben diversa: gli host sono per la maggior parte **mobili** e tendono a cambiare rete molto frequentemente. Per questo motivo l'approccio basato sulla comunicazione via cavo diventa poco efficiente e scalabile, infatti pensare di collegare tutti i dispositivi del mondo, in ogni punto in cui si trovano, via cavo, è impossibile. Dobbiamo quindi adottare un approccio cosiddetto **wireless**, cioè senza cavo, dove i dispositivi sono connessi senza filo. Possiamo immaginare una rete wireless nel seguente modo:

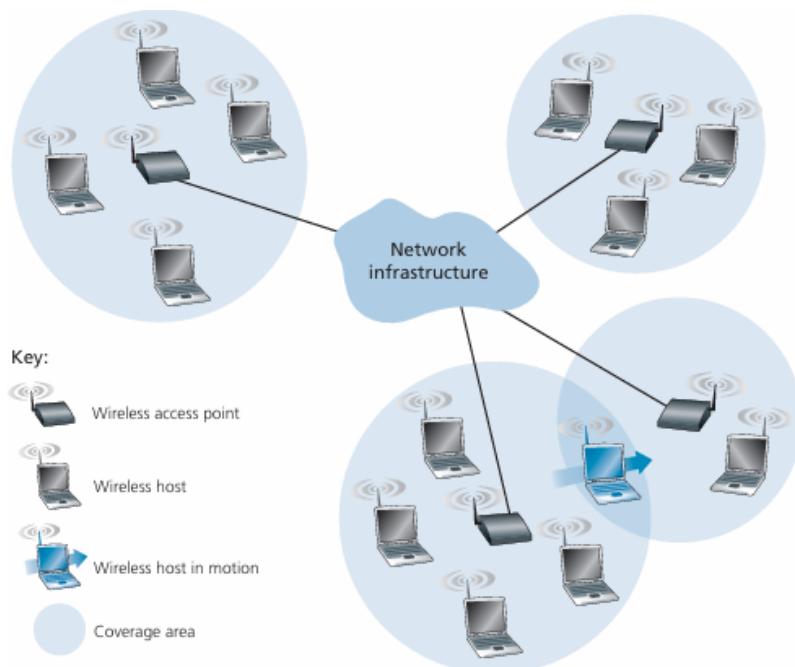


Figura 23.1: Esempio di rete wireless

L'approccio wireless è quindi definito principalmente per gli endpoint, mentre l'infrastruttura di rete centrale è comunque basata su una connessione **via cavo**. La connessione wireless tra **endpoint** e **infrastruttura di rete** è garantita attraverso dei dispositivi chiamati **stazioni base**. Queste **stazioni base** sono il componente chiave dell'architettura di rete wireless, esse sono infatti responsabili dell'**invio** e della **ricezione** dei dati tra gli host wireless a essa associati. Ogni **stazione base** ha una sua **area di copertura** all'interno della quale è in grado di gestire le connessioni

in ingresso e in uscita; fintato che un host si trova all'interno dell'area di copertura diciamo che è **associato alla stazione base**. Sono esempi di **stazione base** due dispositivi:

- **Ripetitori di cella.**
- **Access point.**

Gli host associati a una stazione base sono considerati come operanti in **modalità infrastruttura**, in quanto tutti i tradizionali servizi di rete, sono forniti dalla rete attraverso al **stazione base**. Un tipo particolare di reti, dette **reti ad hoc**, hanno la particolarità di non avere alcuna infrastruttura di rete fissa a cui connettersi, dovendo loro stessi provvedere ai servizi di *instradamento, assegnazione degli indirizzi e DNS*. Una prima classificazione delle reti può essere fatta combinando due aspetti: il primo aspetto riguarda se un pacchetto nella rete attraversa un solo collegamento wireless oppure più d'uno; se vi è una infrastruttura, come stazione base, oppure no:

- **Hop singolo, con infrastruttura.** Queste reti hanno una stazione base che si collega a una rete cablata più grande.
- **Hop singolo, senza infrastruttura.** Non vi è nessuna stazione base collegata alla rete wireless, tuttavia uno dei nodi della rete può coordinare la trasmissione degli altri nodi.
- **Hop multipli, con infrastruttura.** Esiste una stazione base, collegata tramite cavo alla rete più grande. Tuttavia, alcuni nodi potrebbero dover fare affidamento su altri nodi wireless per poter comunicare con la stazione base.
- **Hop multipli, senza infrastruttura.** Non c'è una stazione base e i nodi possono dover ritrasmettere i messaggi a parecchi altri nodi per raggiungere la destinazione.

Quindi, a livello puramente teorico, per sostituire una rete cablata con una rete ethernet è sufficiente cambiare la scheda ethernet dell'host con una scheda di rete wireless e lo switch con un access point; non è richiesta alcuna variazione rispetto a quanto visto fino ad ora. Tuttavia occorre comunque prestare attenzione ad alcuni aspetti relativi alle connessioni wireless. In generale, le radiazioni elettromagnetiche si attenuano quando attraversano determinati ostacoli. Anche nello spazio libero l'intensità del segnale si attenua al crescere della distanza percorsa. Un secondo aspetto riguarda le **interferenze da parte di altre sorgenti**, infatti è possibile che due o più sorgenti, che trasmettono sulle stesse frequenze, finiscano per interferire con i rispettivi segnali, causando del rumore e dei disturbi. Un terzo problema riguarda la **propagazione su più cammini**, non è infatti detto che tutti i pacchetti seguano lo stesso percorso, qualche onda eletromagnetica potrebbe anche riflettersi su una particolare superficie o oggetto, percorrendo una strada diversa rispetto a quella di un altro pacchetto. Quindi, gli host che ricevono il segnale riceveranno un segnale che è combinazione di una forma **degradata** del segnale (data magari da delle attenuazioni dovute alle distanze o a delle interferenze) e un **rumore di fondo** dell'ambiente. Il **rapporto segnale rumore** è una misura relativa dell'intensità del segnale ricevuto, cioè dell'informazione che è stata trasmessa. Il **SNR** viene generalmente misurato in dB, ed è definito come

$$\text{SNR [dB]} = 20 \log_{10} \left( \frac{\text{Ampiezza segnale ricevuto}}{\text{Ampiezza segnale rumore}} \right)$$

La codifica del segnale determina anche il **bit error rate**, cioè la probabilità che trasmesso un bit esso sia errato. Nella comunicazione wireless, rispetto alla comunicazione tradizionale, abbiamo un problema, noto come **problema del nodo nascosto**. Ipotizziamo di avere due stazioni **A** e **C** che intendono comunicare con una terza stazione **B**, ipotizziamo anche che **A** e **C** siano bloccate da una barriera che renda impossibile la loro comunicazione. Se **A** e **C** provassero a comunicare con **B** potrebbe accadere che i segnali delle due stazioni collidano in prossimità della stazione **B** rendendo impossibile la comunicazione. Inoltre, sia **A** che **C** non potrebbero essere a conoscenza della collisione, in quanto non hanno possibilità di sapere che l'altra stazione sta provando a comunicare.

**Confronto tra BER e SNR** Sorge quindi spontaneo chiedersi quale tra **bit error rate** e **signal-to-noise ratio** sia la grandezza migliore per la misurazione della qualità della comunicazione. In termini puramente generali non è possibile definire quale delle due grandezze sia migliore, anche perché, in termini matematici, **BER** e **SNR** sono legate da una relazione: *a parità di modulazione e codifica, all'aumentare di SNR, BER diminuisce*. Inoltre, la sola SNR non è sufficiente per analizzare le prestazioni di un sistema, poiché due sistemi con lo stesso SNR possono avere anche BER molto diversi tra di loro; in particolare, la SNR non tiene conto delle **tecniche di modulazione**, della **codifica** e delle **caratteristiche del canale**. Possiamo quindi concludere dicendo che:

- La BER è una metrica end-to-end che tiene conto di tutti gli effetti sul canale, compresi quelli derivati dalle scelte di tecniche di modulazione e codifica.
- La SNR è una misura puramente fisica che tiene conto solo della natura dei segnali che viaggiano nel mezzo.

## 1 Standard 802.11

Estremamente diffuse nei luoghi di lavoro e nelle case private, le LAN wireless sono oggi la tecnologia più diffusa per l'accesso a internet all'interno di una LAN. Nel tempo si sono diffuse diverse versioni dello standard 802.11 (wifi), tuttavia tutti e tre gli standard hanno in comune alcune caratteristiche fondamentali:

- Utilizzano lo stesso protocollo di accesso al mezzo, cioè CSMA/CA e la stessa struttura del frame a livello di collegamento.
- Possono ridurre la loro frequenza trasmittiva per raggiungere distanze più consistenti e Possono funzionare sia in modalità infrastruttura sia in modalità **ad hoc**.

L'idea di utilizzare il **collision avoidance** deriva dal fatto che il problema del nodo nascosto non permette di rilevare sempre le collisioni, richiedendo quindi che le collisioni vengano evitare a priori piuttosto che gestite quando si verificano. La differenza tra i vari standard è quindi relativa alla velocità trasmittiva più che al funzionamento vero e proprio; in particolare, possiamo rappresentare l'evoluzione dello standard, in funzione della velocità trasmittiva, nella seguente tabella

IEEE 802.11 standard	Year	Max data rate	Range	Frequency
802.11 b	1999	11 Mbps	30 m	2.4 GHz
802.11 g	2003	54 Mbps	30 m	2.4 GHz
802.11 n (WiFi 4)	2009	600 Mbps	70 m	2.4, 5 GHz
802.11 ac (WiFi 5)	2013	3.47 Gbps	70 m	5 GHz
802.11 ax (WiFi 6)	2020	14 Gbps	70 m	2.4, 5 GHz
802.11 af	2014	35–560 Mbps	1 Km	54,790 MHz
802.11 ah	2017	347 Mbps	1 Km	900 MHz

Il principale elemento costitutivo delle reti wifi è il **basic service set**, che contiene una o più stazioni wireless e una stazione base centrale, detta **access point**. Prima di poter inviare dati nella rete, le varie stazioni wireless, devono prima associarsi con un **access point**. Ogni access point possiede alcune configurazioni definite dall'amministratore: uno o più identificativi che rientrano sotto al nome di **SSID** (o **Service Set Identifier**) e un **numero di canale** che identifica la **banda di frequenza** su cui l'**access point** lavora. Ognuno di questi **access point** invia periodicamente dei **frame beacon**, che contengono il proprio codice **SSID** e il proprio indirizzo **MAC**. Un dispositivo wireless che intende associarsi a un access point avrà quindi due possibilità:

- **Scansione passiva.** Un qualsiasi dispositivo wireless che entra all'interno dell'area di copertura di uno o più **access point** dovrà analizzare periodicamente tutte le undici frequenze possibili così da determinare quali **access point** sono disponibili per un associazione.

- **Scansione attiva.** Un dispositivo potrebbe scegliere di inviare in **broadcast** un frame sonda che verrà ricevuto da tutti gli **access point** nel raggio di copertura dell'host wireless. L'**access point** risponderà al frame sonda con un frame di risposta.

Lo standard 802.11 non definisce un algoritmo per selezionare l'**access point** con il quale associarsi: tale algoritmo viene lasciato ai progettisti di firmware e software dell'host wireless. Un criterio di decisione che viene tipicamente utilizzato è quello di scegliere l'**access point** il cui **frame beacon** arriva con potenza del segnale maggiore, in quanto quella è la rete che avrà sicuramente maggiore velocità rispetto alla posizione dell'host. Una volta individuato il punto di accesso con il quale associarsi, l'host wireless invia un frame di richiesta di associazione all'**access point**, il quale risponde con un frame di risposta di associazione. Una volta associato con un **access point** l'host vorrà entrare nella sottorete, cioè vorrà che gli venga assegnato un indirizzo IP. In genere, quindi, l'host invia un messaggio di **DHCP discovery** sulla sottorete e esegue tutta la fase di assegnamento del protocollo DHCP, per poi lavorare normalmente come in una qualsiasi altra rete. L'associazione con un **access point** potrebbe anche richiedere una fase intermedia di **autenticazione**; esistono diversi possibili approcci per la realizzazione del meccanismo di autenticazione: una prima ipotesi potrebbe essere quella di consentire l'accesso alla rete wireless sulla base di indirizzo **MAC** di **destinazione**, oppure utilizzando nome **utente** e **password**.

## 1.1 Protocollo MAC di 802.11

Una volta che un host wireless è associato a un **access point**, può iniziare a trasmettere e ricevere frame dati da e verso la rete a cui l'**access point** è connesso. Tuttavia, poiché stazioni multiple potrebbero voler trasmettere frame di dati contemporaneamente sullo stesso canale, è necessario un protocollo ad **accesso multiplo** per coordinare le trasmissioni. Il protocollo scelto per la gestione del mezzo condiviso è **CSMA con prevenzione di collisioni**, cioè **CSMA/CA**; esistono due motivi principali per questa scelta

- La possibilità di rilevare delle collisioni richiede la capacità di inviare e ricevere contemporaneamente. Tuttavia, essendo in genere la potenza del segnale ricevuto nettamente inferiore alla potenza del segnale trasmesso, sarebbe molto oneroso creare un hardware in grado di rilevare le collisioni.
- Anche se l'adattatore fosse in grado di trasmettere e di ricevere allo stesso istante, non sarebbe comunque possibile rilevare tutte le collisioni per via del problema del terminale nascosto.

Data la mancanza di un meccanismo di rilevazione delle collisioni, una volta che una stazione inizia a trasmettere un frame, lo trasmette per intero. Ciò, in combinazione al fatto che le collisioni sono frequenti, può significativamente degradare le prestazioni di un protocollo ad accesso multiplo.

Inoltre, siccome il tasso di errore dei bit nei canali wireless è molto alto, allora il protocollo 802.11 utilizza uno schema di *avvenuta ricezione/ritrasmissione* a livello di collegamento. Quando la stazione di destinazione riceve un frame che passa il controllo CRC, attende per un intervallo di tempo, noto come **SIFS** (o **Short Inter-Frame Space**), dopo il quale invia al mittente un frame di conferma di avvenuta ricezione. Se la trasmittente non riceverà il frame entro un arco di tempo stabilito, presupporrà un errore e ritrasmetterà il frame. Proviamo ora a descrivere esattamente i passi fatti da una trasmittente che vuole comunicare con una ricevente:

- Se inizialmente la stazione percepisce il canale inattivo, allora attende un certo intervallo di tempo, noto come **DIFS** (o **Distributed Inter-Frame Space**) prima di iniziare a trasmettere il frame.
- Se il canale non risulta inattivo, la stazione sceglie un valore casuale di ritardo, detto **tempo di backoff**, usando un attesa binaria esponenziale e decrementa questo valore solo quando il canale viene percepito come inattivo per DIFS. In particolare:
  - Se il canale viene rilevato libero per DIFS è possibile iniziare a decrementare il contatore.
  - Il contatore viene decrementato in funzione di slot, cioè, per ogni **slot time** in cui il canale è libero si decrementa il timer di backoff.

L'intervallo di **backoff** viene scelto a partire da due valori, chiamati rispettivamente **CW\_min** e **CW\_max**. Ogni qualvolta si verifica una collisione questa finestra viene ampliata secondo la seguente equazione

$$CW_n = \min(2 \cdot CW_{\min} + 1, CW_{\max})$$

Una volta calcolato il backoff, all'interno dell'intervallo  $[0, CW_n]$ , si moltiplica per lo **slot time**.

- Quando il contatore arriva a zero la stazione trasmette l'intero frame e aspetta il frame di conferma.
- Il receiver riceve il frame, attende un intervallo di tempo pari a SIFS e successivamente invia il messaggio di conferma.
- Se riceve la conferma, il ricevente sa che il frame è stato ricevuto correttamente e, qualora avesse un altro frame da inviare, ripeterà le operazioni da capo (attendendo prima un piccolo intervallo di tempo chiamato **Post-Backoff**).

Ovviamente, il valore di **DIFS** deve essere maggiore dell'intervallo **SIFS**, così da dare priorità ai messaggi di conferma, evitando il formarsi di collisioni che costringerebbero il mittente a dover rispedire dei frame che il ricevente ha ricevuto correttamente.

**Virtual Carrier Sensing** Una soluzione molto elegante al problema del nodo nascosta utilizza uno **schema di prenotazione** (o **virtual carrier sensing**). Questa soluzione utilizza due pacchetti speciali: un pacchetto **request to send** (o **RTS**) e un pacchetto **clear to send** (**CTS**). Ipotizziamo di avere un access point  $A_1$  e due terminali  $H_1$  e  $H_2$  che intendono comunicare con l'access point:

- Il nodo  $H_1$  invia un frame **RTS** verso l'access point (solo dopo aver rilevato il canale libero per DIFS) richiedendo di poter usare il canale esclusivamente per tutta la durata della connessione.
- L'access point, una volta ricevuto il pacchetto, attende SIFS e risponde inviando il frame **CTS**. Questo pacchetto viene rilevato da tutti i nodi nell'area di copertura dell'access point.
- Il nodo  $H_1$ , una volta ricevuto il pacchetto **CTS**, attende SIFS e, successivamente, può iniziare a trasmettere i suoi frame verso l'access point.
- Una volta terminata la ricezione del frame di dati, l'access point attende un ultimo SIFS e invia un frame di **ACK**.

Lo scopo del pacchetto **CTS** è duplice: da una parte autorizza l'host che per primo aveva mandato il pacchetto **RTS** a iniziare la comunicazione, dall'altra parte, essendo inviato in broadcast, segnala a tutti i dispositivi associati a quell'access point che il canale è occupato e di attendere. All'interno del pacchetto **CTS**, oltre ad essere indicato l'host autorizzato, è presente anche un campo **duration** che specifica per quanto tempo il canale è occupato. Il campo duration è un campo che si può trovare sia all'interno del frame **RTS** e del frame **CTS**: nel frame **RTS** conterrà l'intervallo di tempo che va dal momento in cui l'access point riceve il frame al momento in cui manda l'**ACK** per il frame dati, mentre nel frame **CTS** conterrà l'intervallo di tempo che va dall'istante in cui il client riceve **CTS** all'istante in cui il server manda l'**ACK**; si noti bene che questi intervalli comprendono anche le attese **SIFS**. Tuttavia, l'utilizzo di questa tecnica non risolve il problema del nodo nascosto, poiché i si possono comunque verificare delle collisioni in corrispondenza dell'invio dei frame **RTS**, d'altro canto però, anche in caso si verificassero delle collisioni, sarebbe comunque una situazione meno catastrofica della collisione sui frame dati. Inoltre, essendo i pacchetti **RTS** e **CTS** molto piccoli, una loro, poco probabile, collisione non causerebbe un problema particolarmente grave. Tuttavia, lo scambio di questi frame introduce un **overhead** importante e consuma risorse del canale. Per tale motivo, questi frame sono utilizzati solo quando un host vuole trasmettere lunghi frame di dati; il pratica, una stazione wireless può impostare una soglia oltre la quale utilizzare i frame **RTS** e **CTS**.

## 1.2 Formato del pacchetto IEEE 802.11

La struttura di un frame 802.11 sono molto simili ai frame ethernet, ma contengono campi specifici per l'utilizzo nei collegamenti wireless. La struttura del frame è così strutturata:

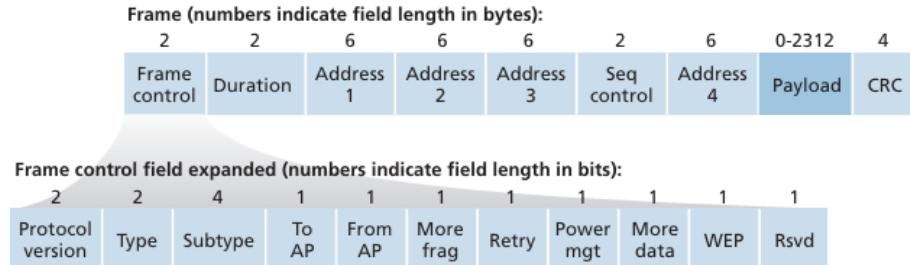


Figura 23.2: Struttura di un frame 802.11

Il cuore del frame è il campo payload, che solitamente consiste di un datagramma IP o di un pacchetto **ARP**. Anche se la dimensione massima può essere di 2312 bit, ma in genere non raggiunge mai i 1500. Per gestire l'**error detection** troviamo un campo **CRC** di 4 byte. La differenza principale tra i frame ethernet e i frame 802.11 è la presenza, all'interno del frame, di quattro diversi campi indirizzo, ciascuno dei quali può contenere un indirizzo MAC di 6 byte. Tre campi indirizzo sono necessari a scopi d'interconnessione, mentre il quarto campo indirizzo è impiegato nelle reti ad hoc, ma non in quelle con un infrastruttura. In particolare, lo standard definisce questi campi nel seguente modo:

- Il secondo indirizzo è l'indirizzo **MAC** della stazione che trasmette il frame. Quindi, se una stazione wireless trasmette il frame, viene inserito all'interno di questo campo il suo indirizzo **MAC**. Se invece è l'**access point** a trasmettere, in questo campo si inserirà l'indirizzo **MAC** dell'**access point**.
- Il primo indirizzo è l'indirizzo MAC della stazione wireless che riceve il frame. Quindi, se una stazione wireless trasmette il frame, viene inserito in questo campo l'indirizzo MAC dell'**access point** di destinazione. Se invece è l'**access point** a trasmettere, in questo campo si inserirà l'indirizzo **MAC** della stazione wireless.
- Il terzo indirizzo **MAC** corrisponde all'interfaccia del router connessa alla sottorete definita dall'**access point**.

Il campo **duration** specifica il tempo, espresso in microsecondi, per il quale il mezzo trasmisivo sarà riservato a partire dalla fine della trasmissione del frame corrente. In altri termini, indica il tempo residuo necessario a completare lo scambio di frame iniziato, includendo anche frame di risposta, frame di conferma e intervalli di tempo obbligatori. I campi contenuti all'interno del **frame control** hanno invece tre scopi principali: identificare il tipo di frame, specificare come deve essere instradato e specificare come deve essere trattato.

## 2 Mobilità nelle sottoreti

Una delle opportunità principali offerte dall'introduzione delle reti wireless deriva dalla possibilità, per gli host, di potersi muovere da un **BSS** a un altro rimanendo comunque connesso alla rete e in grado di comunicare con tutti i servizi che fino a quel momento stava utilizzando. Ipotizziamo ora di avere due **BSS** che si trovano all'interno della stessa sottorete IP; se un host  $H_1$  si sposta da  $\text{BSS}_1$  a  $\text{BSS}_2$  manterrà, per via del fatto che i due **BSS** sono nella stessa sottorete, il medesimo indirizzo IP che aveva in precedenza e le medesime connessioni TCP che aveva in precedenza. La mobilità nelle sottoreti introduce però un problema: come è possibile per lo switch sapere che un host si è spostato da un AP all'altro. Grazie alla qualità di autoapprendimento degli switch, questo genere di problema può essere affrontato positivamente nel caso di spostamenti occasionali. Gli switch non

sono però progettati per supportare utenti che si spostano di frequente e che vogliono mantenere le proprie connessioni TCP nel cambio di **BSS**. Nella tabella di inoltro dello switch troviamo un'entrata che associa l'indirizzo MAC di  $H_1$  alla porta attraverso la quale  $H_1$  può essere raggiunto:

- Se  $H_1$  è inizialmente collegato a **BSS<sub>1</sub>**, allora il datagramma destinato a  $H_1$  sarà inoltrato attraverso **AP<sub>1</sub>**.
- Se  $H_1$  cambia il **BSS**, passando a **BSS<sub>2</sub>**, allora il datagramma destinato a  $H_1$  dovrebbe essere inoltrato attraverso **AP<sub>2</sub>**.

Quello che accade al cambio di **BSS** è che lo switch mantiene comunque all'interno della propria tabella di inoltro l'associazione  $H_1$  e **AP<sub>1</sub>** fintanto che non scade e, quindi, una serie di frame in uscita verso  $H_1$  potrebbero essere mandati verso l'access point sbagliato. Tuttavia ciò non è un problema: nel momento in cui la tabella ARP si aggiorna la trasmissione potrà essere rifatta nuovamente.

### 3 Funzionalità avanzate di 802.11

Una delle funzionalità messe a disposizione nel protocollo 802.11 è l'**adattamento del tasso**. Abbiamo visto che esistono diverse tecniche possibili per la modulazione del segnale che si associano a scenari di **signal-to-noise ratio** distinti: il segnale infatti non è qualcosa di binario, cioè, non è corretto dire che il segnale **arriva** o **non arriva**; il segnale degrada nello spazio in modo lineare man mano che aumenta la distanza. Quindi:

- Se ci troviamo molto vicini all'access point, dato l'alto **SNR**, è possibile utilizzare una tecnica di modulazione a livello fisico, che fornisce un tasso di trasmissione elevato, mentendo un **EBR** basso.
- Se ci allontaniamo dall'access point **SNR** diminuisce all'aumentare della distanza e, quindi, la tecnica di modulazione dovrà essere variata in modo da abbassare il bitrate affinché sia possibile mantenere basso il **BER**.

Il protocollo 802.11 è quindi in grado di variare autonomamente il proprio tasso trasmittivo selezionando in modo dinamico la tecnica di modulazione da utilizzare a livello fisico in base alle attuali o recenti caratteristiche del canale. Se un nodo invia due frame consecutivi senza ricevere un riscontro, il tasso trasmittivo ricade al successivo valore più basso. Se per dieci frame consecutivi viene confermata la ricezione o scade il timer che tiene traccia del tempo dall'ultima sostituzione del tasso, il tasso trasmittivo incrementa alla soglia successiva.

Un altro elemento di cui occorre tenere conto è la **gestione dell'energia**; in particolare, il protocollo 802.11 cerca di minimizzare l'utilizzo di energia cercando di minimizzare la quantità di tempo in cui le funzionalità di ascolto, trasmissione e ricezione sono attive. Un nodo è in grado di alternare esplicitamente tra gli stati di **wake** e di **sleep**; un nodo indica all'**access point** che sta per disattivarsi impostando a uno un particolare bit di gestione energetica nell'intestazione di un frame 802.11. Il nodo imposta poi un **timer**, perché torni a svegliarsi nel momento in cui l'**access point** deve inviare il **beacon**, cioè un frame che contiene delle informazioni di sincronizzazione. L'**access point** è a conoscenza del fatto che il nodo sta per "andare a dormire" o sta dormendo fintanto che il timer non scatta e di conseguenza tutti i frame destinati verso quel nodo saranno messi in attesa per quando si risveglierà. Oltre alle informazioni di sincronizzazione il beacon contiene anche la lista di tutti i nodi che hanno dei frame in attesa all'interno della memoria dell'access point; a questo punto possono succedere due cose:

- Se il dispositivo riconosce all'interno del **beacon** il proprio indirizzo MAC rimarrà sveglio così da poter ricevere i pacchetti.
- Se il dispositivo non riconosce all'interno del beacon il proprio indirizzo MAC potrà tornare a dormire.

## 4 Personal Area Network

Lo standard 802.11 Wi-Fi è rivolto alla comunicazione tra dispositivi che si trovano anche a grandi distanze, fino a 100 metri. Tuttavia è possibile utilizzare il medesimo protocollo anche per comunicazioni su aree molto ridotte; questo è il caso delle **PAN**. Una **PAN**, o **Personal Area Network**, è definibile come una **rete ad uso personale**. L'idea alla base della PAN è quella di poter sostituire i collegamenti tramite filo in collegamenti wireless tra dispositivi che si trovano su diverse distanze, come, ad esempio, una tastiera o un mouse. Uno dei protocolli principali alla base delle **Personal Area Network** è il **bluetooth**. La rete bluetooth opera su frequenze nella banda a 2,4 GHz in modalità TDM: il tempo viene suddiviso in slot di tempo di circa 625 microsecondi ognuno. In ogni slot, si può trasmettere in uno tra 79 canali, cambiando il canale in modo pseudo-casuale da uno slot all'altro. Questa strategia, conosciuta come **FHSS** (o **Frequency-Hopping Spread Spectrum**), ripartisce nel tempo le trasmissioni sullo spettro di frequenza. Le reti bluetooth sono delle reti **ad hoc**, quindi non necessitano di alcuna infrastruttura di rete per funzionare, richiedendo però che i dispositivi si organizzino da soli, dapprima in una **piconet**. Un elemento è scelto come **master** e gli altri operano in modalità **slave**. Il nodo master regola la **piconet** e può trasmettere in ogni slot dispari, mentre lo **slave** può trasmettere solo dopo che il master ha comunicato con lui nello slot precedente.

# Capitolo 24

## Accesso cellulare a internet

Ad oggi tantissimi dei dispositivi che si trovano sulla rete sono dei dispositivi mobili, come cellulari e tablet, potrebbe essere utile estendere le reti mobili affinché siano in grado di supportare sia la voce, sia anche gli accessi wireless alla rete internet. Idealmente, questo accesso dovrebbe avere una velocità di trasmissione piuttosto elevata, riuscendo comunque a mantenere le sessioni TCP degli utenti anche in caso di spostamento del dispositivo. Ad oggi esistono diverse tecnologie che permettono di offrire tale servizio; in particolare, ad oggi le tecnologie principali sono: 2G, 3G, 4G e 5G.

### 1 Tecnologie per l'accesso alla rete mobile

La prima tecnologia di rete che vediamo è la rete 2G, la quale permette di connettere la voce all'interno di una rete wireless. Il termine cellulare si riferisce al fatto che un'area geografica è suddivisa in aree di copertura dette **celle**. Ogni **cella** contiene una **stazione base** che prende il nome di **BTS** (o **Base Transceiver Station**) che scambia segnali con le stazioni mobili della cella. L'**area di copertura** di una **cella** dipende dai molti fattori:

- Presenza di palazzi nella cella.
- Altezza della stazione base.
- Potenza della trasmissione del BTS e della stazione mobile.

Lo standard definisce una combinazione di **FDM** e **TDM** per l'interfaccia area. Nei sistemi misti in canale è quindi ripartito in sotto bande di frequenza all'interno delle quali il tempo è suddiviso in frame e slot. In particolare, se il canale è suddiviso in  $F$  sotto bande e il tempo è suddiviso in  $T$  slot, i canali potranno supportare  $F \times T$  chiamate contemporanee. La versione successiva al 2G è il 3G, dove l'obiettivo è quello di riuscire, oltre che a supportare la voce, supportare anche i servizi di rete attraverso delle reti wireless. Il nucleo della rete dati cellulare 3G connette reti di accesso radio a internet. La core network inter opera con le componenti della rete cellulare esistente per la voce, così da non dover necessariamente creare una nuova rete, ma potendo operare parallelamente sulla rete telefonica. Nella core network di 3G ci sono due tipi di nodi: **Serving GPRS Support Nodes** e **Gateway GPRS Support Nodes**. La prima tipologia di nodi, cioè **SGNS**, ha il compito di consegnare datagrammi a/da nodi mobili nella rete di accesso alla quale il nodo è connesso. Le due tecnologie ad oggi più utilizzate per le reti mobili sono, rispettivamente, il 4G e il 5G.

### 2 Gestione della mobilità

Uno dei problemi che ad oggi i progettisti di reti si trovano a affrontare normalmente è quello relativo alla mobilità degli host all'interno della rete globale. In particolare, possiamo distinguere tre diverse tipologie di host:

- Host con nessuna mobilità. Sono gli host che rimangono all'interno della stessa sottorete, ogni loro spostamento è quindi confinato all'interno del perimetri definito dalla sottorete stessa.
- Host con mobilità media. Sono host che cambiano rete con una frequenza media, costringendo a cambiare, con la rete, anche l'indirizzo IP.
- Host con alta mobilità. Sono host che si spostano con una frequenza molto alta, richiedendo quindi dei meccanismi ad hoc per la gestione degli spostamenti.

Per gestire la mobilità di host che si spostano di frequente non possiamo immaginare di disconnettere e riconnettere ogni volta l'host dalla rete, in quanto potrebbe causare dei ritardi nelle connessioni non indifferenti. L'idea corretta è quella di fare in modo che l'utente utilizzi lo stesso identico indirizzo IP anche quando si sposta tra le varie reti; tuttavia utilizzando il protocollo IP normale non è possibile implementare una soluzione di questo tipo, ed è quindi necessario ricorrere all'utilizzo del protocollo IP mobile, un'estensione del protocollo IP. A tal proposito, definiamo alcuni attori di questo protocollo:

- Host mobile. Un dispositivo mobile che si muove tra le varie sottoreti.
- Home network. Definiamo come home network la rete di appartenenza dell'host e definiamo anche come **permanent address** l'indirizzo che viene assegnato in modo permanente all'host mobile.
- **Home agent**. All'interno della home network introduciamo un dispositivo noto come hoome agent che gestisce la comunicazione coi dispositivi che vogliono comunicare con l'host mobile quando questo non si trova presso la home newtork.

Definiamo anche come **correspond** l'host che vuole comunicare con l'host mobile mentre questo si trova in movimento. Affinché sia possibile per il correspond comunicare con l'host mobile è necessario che avvenga una prima fase di registrazione, dove l'host mobile entra nella **visited network** (cioè la sottorete in cui si trova in quel momento) e si registra al **foreign agent**, fornendo informazioni come: **indirizzo IP**, **indirizzo IP della home network** e **indirizzo IP del proprio home agent**. La seconda fase corrisponde a una fase di **segnalazione**, il **foreign agent** contatta l'**home agent** per segnalare che l'host mobile si trova presso la rete da lui rappresentata; facendo in questo modo l'home agent conosce la posizione del relativo host mobile. Nel momento in cui il **correspond** intende comunicare con l'host mobile possono avvenire operazioni diverse, a seconda che sia stato scelto di utilizzare il **routing diretto** o il **routing indiretto**:

- **Routing indiretto**. Il **correspond** scrive all'indirizzo permanente dell'host mobile, l'**home agent** intercetta il pacchetto e controlla l'indirizzo di destinazione, accorgendosi che qualcuno vuole comunicare con l'host mobile. A questo punto l'**home agent** invia il pacchetto all'**host mobile**, poiché conosce la sottorete presso cui si trova in questo momento. Infine, l'**host mobile** potrà rispondere direttamente al **correspond**.
- **Routing diretto**. Il **correspond** scrive all'indirizzo permanente dell'host mobile, l'**home agent** intercetta il pacchetto e controlla l'indirizzo di destinazione, rendendosi conto che qualcuno vuole comunicare con l'host mobile. A questo punto, a differenza del routing indiretto, l'**home agent** segnala al **correspond** l'indirizzo a cui l'host mobile può essere raggiunto all'interno della **foreign newtork**. Infine, il **correspond** apre una connessione con l'host mobile.

La soluzione del routing indiretto permette di risolvere il problema derivato dalla necessità di triangolare i tre attori all'interno del routing diretto, infatti abbiamo visto che il **correspond** comunica con l'**home agent**, che comunica a sua volta con l'host mobile e che, infine, comunica con il **correspond**; questa catena di comunicazioni allunga i tempi di comunicazione nel caso in cui l'host mobile si trovi nella stessa sottorete del **correspond**. Tuttavia occorre fare una precisazione, nel momento in cui si invia il pacchetto all'host mobile, indipendentemente dalla scelta del tipo di routing, il pacchetto dovrà essere incapsulato (nel caso del routing diretto dovrà incapsularlo il **correspond**, mentre nel caso del routing indiretto sarà l'**home agent** a farlo) all'interno di un pacchetto IP mobile che specifica come indirizzo di destinazione l'indirizzo del **foreign agent** presso cui l'host mobile è attualmente registrato.

**Effetto sui livelli superiori** Agli albori, internet, non era stato pensato con lo scopo di utilizzare comunicazioni wireless, ma solo comunicazioni wired e, per tale motivo, l'introduzione delle comunicazioni wireless ha portato non pochi problemi ai protocolli a livello superiore. Il protocollo TCP potrebbe infatti assumere che una perdita di pacchetti sia frutto di una congestione della rete, quando invece non lo è e dipende dallo spostamento dell'host mobile a un'altra sottorete.

# Parte VIII

## Laboratorio

# Capitolo 25

## Configurazione delle reti nei sistemi UNIX

Come abbiamo visto durante il corso, due dispositivi in rete sono identificati da un indirizzo, chiamato **indirizzo IP**, che li rende indistinguibili rispetto ad altri host presenti in rete. L'indirizzo è un indirizzo è strutturato in quattro gruppi da otto bit ciascuno, per un totale di 32 bit di indirizzo:

192.168.1.12 (1)

Ad oggi tuttavia, vista l'enorme mole di dispositivi connessi in rete, il numero di indirizzi IPv4 è diventato insufficiente e, quindi, si è adottata un indirizzo a 128 bit, chiamato IPv6 (che però non verrà affrontato durante questo corso).

### 1 Configurazione dell'interfaccia di rete

La prima sezione riguarda la configurazione di una singola interfaccia, la quale può essere fatta utilizzando il comando ip specificando alcune opzioni che permettono la gestione e la configurazione di tale interfaccia. Ogni modifica fatta mediante il comando IP non sopravvive al riavvio della macchina, quindi la configurazione di rete in Ubuntu viene gestita attraverso altri due tool che verranno approfonditi in seguito. Alcuni comandi utili per la configurazione mediante ip sono:

```
> abilitare/disabilitare un'interfaccia  
sudo ip link set <interface name> up  
sudo ip link set <interface name> down  
  
> aggiungere/rimuovere un indirizzo IP ad un'interfaccia  
sudo ip addr add 10.0.3.17/24 brd 10.0.3.255 dev <interface name>  
sudo ip addr del 10.0.3.17/24 brd 10.0.3.255 dev <interface name>  
  
> rimuovere tutti gli indirizzi ip a un'interfaccia  
sudo ip addr flush dev <interface name>  
  
> mostrare le intefaccie attive  
ip addr show
```

Il comando ip addr show permette quindi di mostrare quali interfacce sono attive in questo momento, mostrando un output strutturato su più righe, suddiviso per interfaccia. Come abbiamo detto, le modifiche fatte con IP non hanno alcuna persistenza nel sistema una volta che viene spento. Affinché si possa fare delle modifiche o delle configurazioni persistenti è necessario utilizzare uno di due tool messi a disposizione da ubuntu

- Netplan.
- Network manager.

Netplan fornisce un’interfaccia di configurazione per le interfacce; in particolare, genera file di configurazione nel formato comprensibile dal meccanismo di rendering sottostante (**NetworkManager** nel nostro caso). Inoltre, **netplan**, fornisce un’interfaccia di comunicazione per le interfacce basata su un linguaggio di **markup** chiamato **YAML**. Questi file di configurazione rispettano una standard per la denominazione che prevede una condizione fondamentale:

Il nome del file deve iniziare con un prefisso numerato per controllarne l’ordine di esecuzione all’accensione

Senza specificare alcuna configurazione, andando a analizzare il contenuto della cartella `/etc/netplan/` si troveranno due file, il primo è chiamato `01-network-manager-all.yaml`, che contiene la semplice configurazione di rete, in cui tutto viene delegato al **NetworkManager**, e il file `50-cloud-init.yaml`, che specifica al **NetworkManager** che l’interfaccia di rete `enp0s1` verrà configurata automaticamente dal **DHCP**. Il nome del primo file contiene come prefisso `01`, così da specificare che deve essere il primo file di configurazione da leggere in fase di accensione. Per configurare un’interfaccia di rete sarà quindi necessario creare un nuovo file, rispettando lo standard, e poi inserire il codice:

```
network:  
  version: 2  
  ethernets:  
    enp0s1:  
      addresses: [192.168.10.2/24]
```

Una volta scritta la configurazione è sufficiente utilizzare il comando `sudo netplan try`, il quale prima segnalerà gli errori e, successivamente, chiederà conferma prima di applicare le modifiche (se non si da conferma entro 120 secondi le modifiche non saranno applicate). Suddividere gli indirizzi ha diversi vantaggi:

- Crea un isolamento logico tra i vari servizi assegnando a ognuna un diverso indirizzo IP.
- Risulta facilitata la migrazioni di servizi.
- Aggiungere più indirizzi IP a un interfaccia può facilitare la segmentazione logica delle reti per motivi di **sicurezza**, **gestione** o altri requisiti.

Possiamo anche fare in modo che l’assegnazione di indirizzi ad un’interfaccia sia manuale, quindi non sia automatica all’accensione. Per farlo è sufficiente cambiare l’impostazione `dhcp4: true` con `activation-mode: manual`. Una volta applicate queste modifiche l’interfaccia sarà comunque `up`, ma non avrà un indirizzo.

**Configurazione delle interfacce senza NetPlan** Le configurazione delle interfacce può avvenire anche senza `netplan`, è sufficiente installare il modulo `ifupdown`, il quale permette di configurare le interfacce con le informazioni contenute nella cartella `/etc/network/interfaces`.

## 2 Configurazione del default gateway

Il default gateway è il router verso cui l’host invia i pacchetti destinati a reti non locali. Il default gateway principale deve essere unico, non ci possono essere più default gateway tra cui scegliere poiché si potrebbero presentare dei problemi

- Routing ambiguo: il kernel non sa quale uscita usare quando trova più regole di default.
- Asimmetria del traffico: pacchetti in uscita da un router e risposte che tornano da un altro.
- Connettività intermittente.

Per visualizzare la routing table utilizziamo il comando `ip show route` e, nel caso volessimo raggiungere altre rotte, dovremmo utilizzare il comando `ip route add indirizzo IP dev interfaccia` (in altri termini, tutto il traffico di rete verso `indirizzo` IP passerà attraverso l’interfaccia `interfaccia`). Sempre utilizzando la libreria di comandi `ip`, per specificare il default gateway, si dovrebbe usare il

comando `ip route add default via 192.168.1.1`. Se però volessimo avere una configurazione permanente del default gateway si dovrebbe comunque andare all'interno del file di configurazione dell'interfaccia e inserire all'interno del codice

```
routes:  
    to: default  
    via: 192.168.1.1
```

### 3 DHCP

Uno degli ulteriori tasselli che dobbiamo aggiungere alle nostre configurazioni di rete su unix è quello del protocollo DHCP (**Dynamic Host Configuration Protocol**). Il protocollo DHCP permette infatti di assegnare automaticamente l'indirizzo IP ad un host che si connette alla nostra rete attraverso un processo a 4 fasi:

- DHCP Discovery.
- DHCP Offer.
- DHCP Request.
- DHCP Acknowledge.

All'interno di Unix troviamo il server DHCP **kea**, successore di **ISC DHCP**, orientato ad ambienti di rete moderni. Una volta installato potremmo fare tutte le modifiche necessarie semplicemente utilizzando il file di configurazione `/etc/kea/kea-dhcp4.conf`, andando a eseguire successivamente il comando `systemctl restart kea-dhcp4-server.service`. Esistono anche altri due comandi utili per verificare lo stato di funzionamento del server e verificare eventuali errori

```
systemctl status kea-dhcp4-server.service  
sudo journalctl -u kea-dhcp4-server -b
```

All'interno del file di configurazione possiamo trovare differenti sezioni: la prima, cioè la sezione `interface-config`, specifica su quali interfacce di rete il server DHCP si mette in ascolto, la seconda, la sezione `interfaces`, specifica l'elenco delle interfacce in cui il server DHCP riceve e invia pacchetti, il terzo, detto `control-socket`, abilita il canale di controllo locale per inviare comandi a runtime senza riavviare il `deamon`, il quarto campo, cioè `lease-database` contiene la locazione e la procedura con cui il server DHCP salva gli assegnamenti di IP:

- `memfile` imposta il salvataggio sul file `/var/lib/kea`.
- `lfc-interval` specifica l'intervallo per la compattazione periodica dei file di lease.

Successivamente troviamo una sezione `expired-lease-processing`, che specifica dei timer e degli argomenti per la gestione dei lease scaduti; ad esempio: `reclaim-timer-wait-time` specifica di controllare ogni 10s per dei lease scaduti, mentre `flush-reclaimed-timer-wait-time` specifica di rimuovere, ogni 25 secondi, in blocco, gli assennamenti che si trovano all'interno del file da più di `hold-reclaim-time`. Esistono poi tre timer che vengono notificati dal server dhcp verso il client:

- `renew-timer`. Dopo 900s il client prova a rinnovare con il server che gli ha dato il lease.
- `rebind-time`. Dopo 1800s, se il rinnovo non riesce, il client ritenta in broadcast verso qualunque server.
- `valid-lifetime`. Durata, espressa in secondi, del lease.

Infine, troviamo la sezione `option-data`, che specifica delle opzioni globali secondo una gerarchia del tipo: `host > subnet > classe > globale`. Le assegnazioni di indirizzi sono fatte per subnet. Affinché dal lato client sia possibile abilitare l'interfaccia all'utilizzo del DHCP è necessario rimuovere la riga "addresses:" e specificare una riga nella forma "dhcp4:true".

## 4 DNS

Un altro tassello fondamentale all'interno delle nostre configurazioni di rete è il **DNS**, cioè quel meccanismo che permette di ricercare un sito o un servizio conoscendo il suo nome simbolico, senza però conoscere il suo indirizzo IP. Affinché però sia possibile fare una ricerca all'interno del database DNS è necessario conoscere l'indirizzo di almeno un server DNS; all'interno del file `/etc/resolv.conf` viene specificata la lista di tutti i root server DNS che l'host può contattare. Per effettuare una richiesta manualmente è possibile usare il comando `nslookup <nome dominio>`. Per configurare DNS specifici sull'interfaccia di rete con netplan è necessario aggiungere, dopo la riga che specifica se la configurazione dell'indirizzo IP è automatica o manuale, il codice:

```
nameserver:  
  search:  
    "mycompany.local"  
  addresses:  
    10.10.10.253  
    8.8.8.8
```

Il campo `search` specifica il **dominio di ricerca aggiunto alle query**. Ogni volta che viene cercato un **hostname** senza un dominio completo, il sistema proverà ad aggiungere `mycompany.local` alla fine del nome per completare la richiesta nella rete locale.

## 5 ICMP

L'**internet control message protocol** è un protocollo della suite IP definito per lo scambio di messaggi di controllo, esecuzione e diagnostica. L'idea di questo protocollo è quella di fornire una serie di servizi per la rilevazione, ma non per la correzione, degli errori. Un pacchetto **ICMP** è incapsulato in un pacchetto IP e contiene

- Un campo `Type` che identifica la categoria del messaggio.

Codice	Requisito
0	Echo reply
3	Destination Network
8	Echo request
11	Time to Live expired
30	Traceroute

Tabella 25.1: Tipi dei pacchetti ICMP

- Un campo `Code` che specifica nel dettaglio il tipo.
- Un campo `Checksum` per il controllo di integrità.
- un campo `Data` per delle variabili dipendenti dal tipo.

I messaggi dell'ICMP possono essere messaggi informativi, quindi utilizzati per la diagnostica ed il controllo, o messaggi di errore, utilizzati invece per segnalare problemi di consegna o di instradamento. Un esempio di comando della suite **ICMP** è il comando

```
ping <indirizzo_ip>
```

che permette di testare la connettività tra un host e un altro host remoto di cui si conosce l'indirizzo IP o l'URL. Il comando invia uno o più messaggi di tipo **echo request** e attende messaggi di tipo **echo reply**. In particolare

- Il comando ping crea un pacchetto ICMP di tipo **echo request**.

- Il pacchetto è encapsulato in un pacchetto **IP** e inviato al destinatario.
- L'host di destinazione, se attivo e configurato per rispondere, genera un pacchetto **ICMP Echo Reply** e lo instrada verso il mittente.
- Il pacchetto torna all'host sorgente.

Il tempo tra l'invio e la ricezione è misurato e mostrato come **RTT**. Le risposte a questo comando possono essere tre: la prima, **network unreachable**, indica che la rete non è raggiungibile, la seconda, 100% packet loss, indica che nessuno dei pacchetti è arrivato a destinazione e, infine, il terzo, **unkwnon network** indica che non è stato possibile risolvere il nome dell'host specificato. Il comando **ping** ha però diverse limitazioni, può essere infatti bloccato da un firewall con delle politiche particolarmente stringenti, inoltre misura solo la raggiungibilità a livello IP, non la disponibilità del servizio applicativo e, infine, non distingue tra congestione di rete, filtri, o host realmente spento. Un comando più sofisticato, che risolve parzialmente alcuni di questi problemi, è il comando **traceroute**. L'idea di questo comando è quella di conoscere il percorso che il pacchetto IP effettua per raggiungere un destinatario e, per farlo, utilizza l'invio di pacchetti **sonde**. Una **sonda** è un pacchetto UDP con un TTL crescente. Il funzionamento è relativamente semplice:

- Si manda una sonda e si attende.
- Se il pacchetto è di tipo **ICMP Destination Unreachable** allora ci fermiamo.
- Altrimenti si incrementa il TTL e si riparte dal punto 1.

Tuttavia, in alcuni casi, le risposte arrivano con un tipo di sonda diverso: **ICMP Echo** o **TCP SYN**.

## 6 Analisi dei pacchetti

Un'interfaccia collegata a una rete è in grado di ricevere e visualizzare tutti i pacchetti che circolano sul mezzo condiviso. Di solito, l'interfaccia conserva e analizza solo il traffico riservato a lei, ma settando l'interfaccia in modalità promiscua è comunque possibile intercettare tutti i pacchetti. Esiste un comando, chiamato **tcpdump**, che permette di settare tale modalità, specificando anche alcune opzioni

Codice	Requisito
<b>-c</b>	Specifica il numero di pacchetti da visualizzare
<b>-i</b>	Specifica l'interfaccia da usare
<b>-q</b>	Visualizza meno informazioni
<b>-n</b>	Non risolve i nomi DNS
<b>-v, -vv, -vvv</b>	Specifica un livello di dettaglio crescente
<b>-w nomefile</b>	Scrive l'output in un file.
<b>-r nomefile</b>	Legge da un file precedentemente creato.

Tabella 25.2: Opzioni del comando **tcpdump**. I formati dei file relativi alle opzioni **-w** e **-r** devono avere estensione **.pcap**

# Capitolo 26

## Programmazione di applicazioni in rete

In questo capitolo verrà analizzata la **programmazione distribuita**. Dal corso di sistemi operativi abbiamo visto che la cooperazione tra processi, che è un tipo di sincronizzazione, può avvenire in due modalità:

- **Sincronizzazione mediante strutture dati.** Il caso dei **semafori**.
- **Sincronizzazione mediante scambio di informazioni.** Il caso della memoria condivisa, dello scambio di messaggi e delle chiamate a procedura remota.

Tuttavia il caso che studieremo noi è quello della cooperazione tra processi che risiedono su macchine diverse, cioè il caso di un **sistema distribuito**. In questo caso la comunicazione può avvenire solo mediante lo scambio di informazioni, nello specifico, mediante lo scambio di messaggi. Abbiamo anche visto che esistono due possibili architetture di rete, quella **peer-to-peer** e quella **client-server**; l'architettura che prendiamo come riferimento, che è anche quella principalmente usata per i sistemi distribuiti.

### 1 Socket

Lo scambio di messaggi tra processi su macchine diverse è possibile attraverso l'astrazione dei socket. Questa astrazione pone un canale virtuale astratto tra i due processi, che permette di ignorare l'architettura di rete sottostante, focalizzandosi solo sulla cooperazione tra i due processi. Un socket è identificato da due elementi

- Un indirizzo host. Nel modello TCP/IP corrisponde all'indirizzo IP.
- Un indirizzo del processo. Nel modello TCP/IP corrisponde al numero di porta.

Un socket è quindi una vera e propria estremità di un canale di comunicazione, è come se fosse una casella postale all'interno della quale vengono inseriti i pacchetti verso quel particolare processo sulla macchina client o server. Il linguaggio C mette a disposizione delle primitive per la creazione e la gestione dei socket definite all'interno delle librerie

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

La funzione fondamentale di queste librerie è quella che permette di creare un socket, chiamata **socket()**. La primitiva è così definita

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

Il parametro **domain** ammette due possibili valori, il primo, detto **AF\_LOCAL**, specifica che la comunicazione avviene sulla stessa macchina, mentre il secondo, detto **AF\_INET**, specifica che

la comunicazione avviene su macchine diverse e che si utilizza il protocollo IPv4 abbinato a uno dei due protocolli a livello di trasporto. Il parametro `type` specifica la tipologia di socket: se vale `SOCK_STREAM` indica che il socket è di tipo TCP, altrimenti, se vale `SOCK_DGRAM` indica che il socket è di tipo UDP. L'ultimo campo, `protocol`, potrebbe specificare un altro protocollo a livello di trasporto, ma nel nostro caso sarà sempre messo a 0. La funzione restituisce un descrittore di file che rappresenta il socket e servirà a manipolare il socket attraverso altre primitive, oppure -1 se si è verificato un errore.

Una volta creato il socket è necessario associare un indirizzo IP, per farlo è quindi necessario utilizzare delle strutture dati particolari. La gestione degli indirizzi viene affidata alla struttura `sockaddr_in`, dotata di tre campi

- Il campo `sin_family`, di tipo `sa_family_t`, permette di specificare la famiglia di indirizzi IP utilizzata.
- Il campo `sin_port`, di tipo `in_port_t`, specifica la porta.
- `sin_addr`, di tipo `struct in_addr`, specifica l'indirizzo IP effettivo.

La struttura `struct in_addr` è dotata di un solo campo, cioè l'indirizzo IP. Una dei vincoli imposti dalla definizione di questi campi è che i valori binari siano scritti secondo l'ordine specificato dalla rete, quindi il **big-endian**. Se per qualche motivo il formato dell'host non fosse corrispondente al **big-endian** sarebbe quindi sufficiente utilizzare una delle funzioni di conversione messe a disposizione dal linguaggio:

```
uint32_t htonl(uint32_t hostlong); // host to network long
uint16_t htons(uint16_t hostshort); // host to network short
uint32_t ntohl(uint32_t netlong); // network to host long
uint16_t ntohs(uint16_t netshort); // network to host short
```

Anche il formato stesso degli indirizzi deve essere coerente con il formato specificato dalla struttura del linguaggio: generalmente, all'interno di un calcolatore, viene utilizzato un**formato numerico a 32 bit** binario, mentre il **formato di presentazione**, cioè quello effettivamente usato in rete, deve avere una struttura detta a **notazione decimale puntata**. Per convertire dalla seconda notazione alla prima si utilizza la funzione

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

Dove `af` (address-family) indica la famiglia di indirizzi, `src` è una stringa stringa del tipo "ddd.ddd.ddd.ddd" e `dst` un puntatore a una struct `in_addr`. Viceversa, per convertire dal formato **numerico** a quello **decimale puntato** si utilizza la funzione

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

Dove `af` (address-family) indica, anche in questo caso, la **famiglia di indirizzi**, `src` è un puntatore a una struttura `in_addr`, `dst` è un puntatore a un buffer di caratteri lungo `size` e, infine, `size` è un intero che specifica la lunghezza del buffer tale per cui il suo valore minimo deve essere almeno `INET_ADDRSTRLEN`. Proviamo ora a fare l'esempio di avvio di un server passivo; il primo passo è sicuramente quello di includere le librerie, successivamente è possibile creare il socket attraverso la funzione

```
int sd = socket(AF_INET, SOCK_STREAM, 0);
```

Una volta creato il socket è anche necessario associare un indirizzo IP a quest'ultimo e, quindi, è necessario definire un'istanza della struttura `sockaddr_in`, popolandola poi con opportuni campi:

```
struct sockaddr_in my_addr;
memset(&my_addr, 0, sizeof(my_addr));
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(4242);
inet_pton(AF_INET, "192.168.4.5", &my_addr.sin_addr);
```

## 1.1 Programmazione lato server

Abbiamo quindi creato l'indirizzo del socket, o almeno la struttura che lo descrive, e il socket vero e proprio. Tuttavia, ad ora, il socket, non è ancora associato all'indirizzo IP e alla porta; per creare questa associazione tra **indirizzo IP**, **porta** e **socket** è necessario usare la primitiva `bind`:

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

Il primo parametro, cioè `sockfd`, corrisponde al **descrittore del socket**, il secondo parametro, cioè `addr`, è un puntatore ad una struttura di tipo `struct sockaddr` alla quale dobbiamo passare, facendo un cast esplicito, la nostra struttura `sockaddr_in`, e infine il terzo parametro, cioè `addrlen`, indica la dimensione di `addr`. Nel caso del server passivo che abbiamo preso in esame la struttura della `bind` sarà così fatta

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

La funzione restituirà 0 se ha successo, altrimenti -1. Ipotizziamo ora che il nostro server sia **passivo**, cioè che venga utilizzato solo per ricevere richieste di connessione; per fare in modo che il server si metta in ascolto è possibile utilizzare la primitiva

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

dove `sockfd` specifica il descrittore del socket, mentre `backlog` specifica la dimensione della coda, cioè il numero di richieste che possono rimanere in attesa di essere gestite. Nell'esempio che abbiamo preso in esame

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

La richiesta di connessione da parte di un client al socket del server arriva prima al sistema operativo del server: il kernel la mette quindi in attesa all'interno di una coda fintanto che il server non l'accetta attraverso una primitiva `accept`. Questa primitiva permette infatti di accettare una richiesta di connessione pervenuta sul socket, che però, ovviamente, dovranno essere di tipo `SOCK_STREAM`:

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

Anche in questo caso il primo parametro è il descrittore del socket. Il secondo parametro è invece particolare, poiché rappresenta un puntatore a una struttura **vuota** che servirà a salvare l'indirizzo del client di cui viene gestita la richiesta. La funzione restituisce un nuovo descrittore di socket, che sarà quello utilizzato per la comunicazione effettiva.

## 1.2 Programmazione lato client

Dal punto di vista del client la questione è leggermente diversa, infatti anche in questo caso è necessario associare il **socket** alla corrispondente struttura `sockaddr_in`. Tuttavia, a differenza della programmazione lato server, nella programmazione lato client non si utilizza più la primitiva `bind`, ma la primitiva `connect`

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

dove il primo parametro è un **descrittore di socket**, il secondo parametro è un puntatore a una struttura che contiene l'indirizzo del server (si dovrà dunque, come nel `bind`, fare il cast) e il terzo parametro è la lunghezza della struttura `addr`.

## 1.3 Scambio di dati attraverso un socket

Ovviamente nel nostro caso non ci limiteremo alla creazione di soli server statici e di soli client statici, ma studieremo il modo e le modalità con cui è possibile comunicare in rete attraverso i client. La comunicazione viene resa possibile tra socket utilizzando due primitive

- `send()`.

- `recv()`.

La primitiva `send` permette di mandare un messaggio attraverso un **socket** connesso; la struttura della primitiva è la seguente

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

Il primo parametro è, come visto tante volte, il descrittore del socket. Il secondo e il terzo parametru sono tra loro collegati e sono, rispettivamente, il puntatore al buffer che contiene il messaggio e la lunghezza (in `byte`) del messaggio. Il quarto parametro permette di specificare delle opzioni aggiuntive, ma per ora ci limitiamo a lasciarlo a 0. La funzione `send` è **bloccante**, cioè blocca il mittente fintanto che non è stato scritto tutto il messaggio; in altri termini:

- `send` non spedice il pacchetto, ma copia ideati all'interno di un buffer del kernel che, successivamente, li frammenterà in pacchetti **TCP**.
- Se per qualche motivo il buffer del kernel è quasi pieno, la `send` bloccherà l'esecuzione del mittente fintanto che c'è spazio.

La funzione restituisce al mittente il numero di byte effettivamente inviati. La funzione duale alla `send` è la `recv`. Questa funzione permette di prelevare un messaggio da un socket connesso

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

Il primo parametro è, come visto tante volte, il descrittore del socket. Analogamente alla `send`, `buf` e `len`, sono rispettivamente il puntatore al buffer in cui salvare il messaggio e la dimensione in byte del messaggio desiderato. Il quarto parametro specifica delle opzioni aggiuntive. La funzione restituisce il numero di byte ricevuti, -1 in caso di errore o 0 se il socket remoto si è chiuso. Anche in questo caso la funzione è bloccante.

L'ultima funzione di utilità è quella che permette la chiusura del socket. Questa funzione, definita all'interno di `unistd.h`, ammette come parametro il descrittore del socket

```
int close(int fd);
```

La funzione restituisce zero se ha successo, oppure -1. Possiamo riassumere tutto quello detto fino ad ora nella seguente immagine

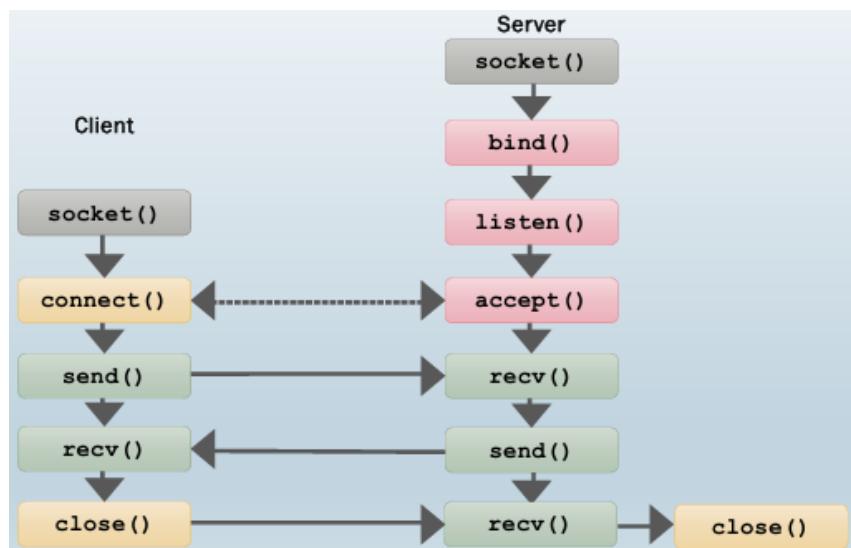


Figura 26.1: Recap sul funzionamento di una connessione client-server basata su socket

Questa immagine permette solo di vedere il caso in cui vi è un singolo client che cerca di connettersi ad un server. Tuttavia, nel caso di un maggior numero di client, è necessario adottare delle strategie più complesse.

## 2 Gestione di richieste multiple

Uno delle problematiche principali che un programmatore si trova ad affrontare nella programmazione distribuita è la gestione delle **richieste multiple**. Con **richieste multiple** intendiamo la possibilità di diversi client di connettersi contemporaneamente a uno stesso server, potendo eseguire delle operazioni in contemporanea. Questo tipo di situazioni richiedono l'implementazione di meccanismi appositi a livello di server, che permettano una gestione parallelizzata delle richieste in entrata. Possiamo pensare a due approcci possibili per la gestione di richieste multiple:

- **Server iterativo.** Il server è in grado di gestire una sola richiesta per volta e le richieste che non sono gestite vengono accodate all'interno di un buffer.
- **Server concorrente.** Il server gestisce le richieste "in parallelo" e per ogni richiesta accettata viene creato un nuovo thread o processo figlio.

### 2.1 Server concorrente

Nella seconda modalità, per creare un processo figlio, è possibile utilizzare la funzione `fork`. Nel momento in cui questa funzione viene eseguita, un processo figlio, con i medesimi descrittori del padre, viene creato e inserito all'interno della coda dei processi pronti. Ovviamente, siccome i descrittori vengono duplicati, padre e figlio devono chiudere il descrittore che non utilizzano, così da evitare di mantenere aperte delle risorse che non sono necessarie; in particolare: il **padre** chiude il **descrittore del socket connesso al client** e il **figlio** chiude il **descrittore del socket in ascolto**. Prendiamo in esame il seguente esempio, dove supponiamo di aver già inizializzato correttamente i socket

```
while(1){  
    new_sd = accept(sd, (struct sockaddr*) &cl_addr, &len);  
    pid = fork();  
    if (pid == -1) // ERRORE FORK.  
    if (pid == 0){  
        close(sd);  
        // gestione della richiesta  
        close(new_sd);  
        exit(0);  
    }  
  
    close(new_sd);  
}
```

Ad ogni nuova richiesta accettata si crea il processo figlio, successivamente il figlio chiude il socket utilizzato per accettare nuove richieste (`close(sd)`), gestisce la richiesta e, infine, chiude anche il socket per quella particolare richiesta (`close(new_sd)`). Un approccio basato sull'utilizzo della funzione `fork()` presenta alcuni vantaggi non indifferenti:

- Isolamento tra processi. Ogni processo possiede una propria area di memoria e delle risorse private, non è quindi necessario che vengano implementati dei meccanismi di protezione sulle risorse condivise.
- Sicurezza. L'isolamento tra processi permette anche di garantire una maggiore sicurezza del codice.

Tuttavia, una soluzione di questo tipo introduce delle criticità legate all'efficienza e all'utilizzo delle risorse. Infatti, la creazione di un processo figlio, è un'operazione costosa per il sistema operativo in termini di tempo di CPU e memoria, poiché richiede la duplicazione delle tabelle e dei descrittori del processo padre. Un'opzione più leggera per la creazione di un server concorrente è quella di utilizzare la libreria `pthread`, creando, piuttosto che un processo per ogni richiesta, un thread apposito per ogni richiesta. Ogni qualvolta viene creato un nuovo thread è necessario anche inizializzare altri elementi:

- Un semaforo, per regolare l'accesso in mutua esclusione alle risorse condivise del processo di appartenenza del thread. Il tipo del semaforo è chiamato `pthread_mutex_t`.
- Tre funzioni per la gestione dei thread:
  - Una funzione `pthread_create` che permette di creare un nuovo thread che parte eseguendo la `start_routine`.
  - Una funzione `pthread_join` che permette a un thread di bloccarsi in attesa della terminazione di un altro thread.
  - Una funzione `pthread_detach` che permette di rendere un thread indipendente, liberando automaticamente le sue risorse alla terminazione senza che altri thread debbano attenderlo
  - Una funzione `pthread_exit` che permette a un thread di terminare e di mettere a disposizione un suo risultato.

Prendiamo come esempio il seguente codice, dove all'inizio definiamo la funzione `manage_request` che permette la gestione delle richieste da parte del server, successivamente si definisce un ciclo infinito, dove per ogni richiesta viene creato un nuovo thread, a cui viene passato il socket della richiesta e, infine, viene applicata la funzione `pthread_detach` per rendere il thread indipendente:

```
void* manage_request(void* arg);

int main(){
    // inizializzazione dei thread e delle strutture necessarie
    ...
    for(;;){
        new_sd = accept(sd, (struct sockaddr*) &cl_addr, &len);
        pthread_t id;
        pthread_create(&id, NULL, manage_request, new_sd);
        pthread_detach(&id);
    }
}
```

Ovviamente, se da una parte, utilizzando una soluzione multithreading si alleggerisce il carico complessivo sul sistema dal punto di vista elaborativo, si crea il problema della gestione in mutua esclusione delle risorse condivise. In altri termini, è necessario che ogni risorsa condivisa sia opportunamente protetta dall'utilizzo di semafori.

### 3 Server iterativo

Il primo approccio possibile per la gestione delle richieste multiple è quella del server iterativo. In particolare, l'idea è quella di utilizzare dei **socket non bloccanti**, cioè socket che non si bloccano sulle operazioni in attesa che finiscano, ma che proseguano la loro esecuzione normalmente. Di default, i socket sono bloccanti:

- `connect()`. Si blocca finché il socket non è connesso.
- `accept()`. Si blocca finché non c'è una richiesta di connessione.
- `send()`. Si blocca finché l'intero messaggio non è stato inviato.
- `recv()`. Si blocca finché non c'è qualche dato disponibile.

La modalità dei socket bloccanti è sicuramente più semplice da utilizzare e va bene per applicazioni in cui i blocchi temporanei non sono problematici o in cui viene utilizzato il multithreading per gestire la concorrenza. La modalità non bloccante è utile in applicazioni che devono gestire molte connessioni simultaneamente e dove bloccare il programma per un singolo socket sarebbe inefficiente. Per settare un socket come **non bloccante** è necessario specificare un particolare flag al momento della creazione del socket chiamato `SOCK_NONBLOCK`.

Facendo in questo modo le operazioni non si bloccano in attesa dei risultati, ma restituiscono degli errori di diversa natura:

Comando	Descrizione
<code>connect()</code>	se non può connettersi subito restituisce <code>-1</code> e setta <code>errno</code> a <code>EINPROGRESS</code> .
<code>accept()</code>	se non ci sono richieste restituisce <code>-1</code> e setta <code>errno</code> a <code>EWOULDBLOCK</code> .
<code>send()</code>	se non riesce a inviare tutto il messaggio subito, restituisce <code>-1</code> e setta <code>errno</code> a <code>EWOULDBLOCK</code> .
<code>recv()</code>	se non ci sono messaggi restituisce <code>-1</code> e setta <code>errno</code> a <code>EWOULDBLOCK</code> .

Tabella 26.1: Effetto delle operazioni nel caso di socket non bloccanti

Una tecnica alternativa a quella dei socket non bloccanti, è quella di poter controllare più socket nello stesso momento, senza dover bloccarsi su uno in particolare. Questa tecnica è nota come I/O multiplexing. L'idea è quella di controllare più socket contemporaneamente e di gestire il primo che risulta pronto. La funzione centrale per implementare questo approccio è quella della `select`, che prende come parametri:

- `nfds`. Il numero del descrittore più alto di quelli da controllare sommato a 1.
- `readfds`. Lista dei descrittori da controllare per la lettura.
- `writelfds`. Lista dei descrittori da controllare per la scrittura.
- `exceptfds`. Lista dei descrittori da controllare per le eccezioni.
- `timeout`. L'intervallo di timeout entro il quale se non sono stati trovati descrittori pronti, la funzione termina

La funzione restituisce il numero di descrittori pronti. Questa funzione è bloccante, poiché blocca il processo fintanto che uno dei descrittori tra quelli controllati diventa pronto oppure finché il `timeout` non scade. Possiamo anche definire quando un socket è pronto; in particolare Un socket di lettura/scrittura è pronto quando:

Socket Lettura	Socket Scrittura
C'è almeno un byte da leggere	C'è spazio nel buffer per scrivere
Il socket è stato chiuso	C'è un errore.
Il socket è in ascolto e ci sono connessioni effettuate.	
C'è un errore	

Tabella 26.2: Effetto delle operazioni nel caso di socket non bloccanti

Una funzione `select` è bloccante fintanto, tra le altre cose, che non scade un timer. Per rappresentare questo timer si utilizza una struttura particolare, chiamata `timeval`, dotata di due campi: un campo `tv_sec` che specifica i secondi e un campo `tv_usec` che specifica i microsecondi. Inoltre, abbiamo anche detto che la `select` ammette come parametro l'insieme dei descrittori su cui stare in ascolto; questo insieme è una struttura `fd_set`. Questa struttura può essere manipolata attraverso delle primitive:

```
// Rimuovere un descrittore da un set
FD_CLR (int fd, fd_set* set);
```

```
// Controllare se un descrittore è nel set
FD_ISSET(int fd, fd_set* set);

// Aggiungere un descrittore al set
FD_SET(int fd, fd_set* set);

// Svuotare il set
FD_ZERO(fd_set* set);
```

L'idea della funzione è relativamente semplice, inserisco all'interno del set tutti i descrittori su cui si intende rimanere in ascolto e successivamente si esegue la `select`; dopo la select si trova, all'interno dei set di lettura, i socket pronti.

## 4 Socket UDP

I socket che abbiamo visto fino a questo momento sono **socket TCP**, cioè socket che stabiliscono una connessione tra **Client** e **Server** attraverso un handshake a tre vie. Tuttavia, una delle possibilità introdotte dalla programmazione distribuita è quella di avere dei socket **UDP**, cioè dei socket che non stabiliscono delle connessioni, ma che si limitano a instradare i pacchetti verso il livello di rete senza preoccuparsi di niente. Distinguiamo, in particolare, due funzioni:

- La funzione `sendto()` permette di mandare un messaggio attraverso un socket all'indirizzo specificato. Questa funzione accetta come parametri: `sockfd`, cioè il descrittore del socket, `buf`, quindi il buffer contenente il messaggio da inviare, `len`, la dimensione in byte del messaggio, `flags`, per settare delle opzioni, `dest_addr`, quindi il puntatore alla struttura in cui è salvato l'indirizzo destinatario e `addrlen`, quindi la dimensione di `dest_addr`.
- La funzione `recvfrom()` permette di mandare un messaggio attraverso un socket all'indirizzo specificato. Questa funzione accetta come parametri: `sockfd`, cioè il descrittore del socket, `buf`, quindi il buffer contenente il messaggio ricevuto, `len`, la dimensione in byte del messaggio, `flags`, per settare delle opzioni, `src_addr`, quindi il puntatore alla struttura in cui è salvato l'indirizzo del mittente e `addrlen`, quindi la dimensione di `src_addr`.

Usando `connect` è possibile associare un socket UDP a un indirizzo IP remoto: il socket invierà e riceverà pacchetti solo da quell'indirizzo.

# Capitolo 27

## Apache HTTP Server

Lo scambio di ipertesto sul web avviene mediante protocollo HTTP. Questo protocollo, il cui nome completo è **HyperText Transfer Protocol**, è un tipo di protocollo basato su architettura **client-server**, dove il **browser dell'utente** (client) richiede un documento HTML e il sever HTTP risponde. La particolarità di questo protocollo è quella di essere **stateless**: il **server** non mantiene traccia delle precedenti connessioni e, per tale motivo, ogni scambio **richiesta/risposta** è indipendente, anche se proviene dal medesimo host. All'interno della nostra macchina virtuale utilizziamo il **server web apache**, cioè un servizio che ci permette di mettere a disposizione pagine web. L'installazione può essere fatta mediante il comando

```
sudo apt install apache2
```

Dopo l'installazione il server dovrebbe avviarsi automaticamente, tuttavia è possibile verificare lo stato del server apache in due modalità diverse

- Utilizzando un comando. In questo caso, oltre allo stato del server, è possibile anche ottenere tutta una serie di informazioni utili, come la documentazione, il PID del processo del server, la memoria occupata, etc. Il comando da utilizzare è attraverso il comando

```
systemctl status apache2
```

- Accedendo al localhost. Aprendo un qualunque browser all'indirizzo del **localhost** si è in grado di osservare se il server è attivo. L'indirizzo del localhost, generalmente 127.0.0.1, è contenuto all'interno di un filesystem.

La configurazione di apache è abbastanza modulare e avviene attraverso delle direttive contenute all'interno del file di configurazione /etc/apache2/apache2.conf. Queste direttive possono essere di due tipologie

- Direttive atomiche.
- Direttive contenitori.

Non tutte le configurazioni sono contenute all'interno del file di configurazione principali, ma si osserva che all'interno del file è possibile trovare dei riferimenti a dei file contenuti all'interno di /etc/apache2/conf\_available. Quindi tutti i pezzi di configurazione andranno inseriti all'interno di questa directory. Per abilitare questi file e quindi creare un link simbolico per poter includere il file è necessario utilizzare il comando

```
a2enconf <nome_file>
```

Il comando crea quindi il link nella directory /etc/apache2/conf\_enabled. La configurazione base del server prevede che il file di configurazione principale includa tutto quello che si trova all'interno di conf-enabled. Tuttavia, quando si crea un nuovo link di configurazione, è necessario riavviare il server. Il comando che invece permette di disabilitare un file è

```
a2disconf <nome_file>
```

Anche in questo caso sarà necessario riavviare il server successivamente.

**Comandi per l'interazione con un server apache** Per interagire con un server apache possibile utilizzare due diverse classi di comandi da terminale:

- `apa2ctl <comando>`, dove i comandi principali sono: **start**, **stop**, **restart**, **status** e **configtest**.
- `service apache2 <comando>`, dove i comandi principali sono: **start**, **stop**, **restart**, **reload**.

## 1 Direttive globali

Le direttive globali sono impostate nel file di configurazione principale di Apache, cioè il file `apache2.conf`. All'interno di questo file possiamo trovare delle configurazioni generali che vengono applicate al server apache nel suo complesso, indipendentemente dai siti web o virtual host ospitati. In queste direttive rientrano

- Impostazioni di sicurezza.
- Porte su cui apache ascolta.
- Directory predefinite.
- Impostazioni di logging.

Le direttive vengono applicate a tutte le richieste e a tutte le risorse servite dal server, a meno che non vengano sovrascritte a livello di virtual host o directory. La prima direttiva che vediamo è quella che permette di specificare la directory principale contenente i file di configurazione di apache

```
ServerRoot /etc/apache2
```

Eventuali path relativi specificati nelle altre direttive sono risolti a partire da questa directory. Questa direttiva è configurata in modo automatico all'avvio del servizio `apache2ctl`. Esistono anche delle direttive che permettono di specificare come gestire le connessioni:

- **KeepAlive**. Specifica se offrire o meno le connessioni persistenti.
- **KeepAliveTimeout**. Specifica quanti secondi attendere la richiesta successiva dal client sulla stessa connessione prima di chiuderla.
- **MaxKeepAliveRequest**. Specifica quante richieste può fare un client sulla stessa connessione prima di chiuderla.

Per specificare le porte che utilizzerà apache per mettersi in ascolto di una connessione è la direttiva **Listen**. Questa direttiva, definita all'interno di `/etc/apache2/ports.conf`, è obbligatoria affinché il server possa partire correttamente. La direttiva **ErrorLog** permette invece di specificare il file di log degli errori, cioè specifica il **path assoluto** del file dove salvare eventuali messaggi di errore. Il formato e la verbosità dei messaggi possono essere definite con le direttive, rispettivamente, **ErrorLogFormat** e **LogLevel**.

### 1.1 Direttive per siti web

Nel caso più semplice, cioè quello di un server Web in esecuzione su una macchina con un solo indirizzo IP e che offre un solo sito web, risulta abbastanza semplice la configurazione dei server. Tuttavia, nel caso in cui si voglia ospitare più siti web all'interno l'approccio basato sull'ospitare ogni sito è non ottimale, poiché:

- Necessità di una grande quantità di risorse.
- Necessità di una diverse macchina per ogni sito, con aumento dei costi.

Apache offre un servizio di **virtual host**, che permette di virtualizzare più siti sulla stessa macchina andando a discriminare per nome.

Come per altri pezzi di configurazione e moduli, i siti si mettono all'interno della cartella `/etc/apache2/sites-available` e possono essere abilitati/disabilitati con i comandi

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

Un sito abilitato ha un soft-link all'interno di `sites-enabled`. Per definire un virtual host si utilizza una direttiva contenitore `VirtualHost`, che specifica indirizzo IP e porta a cui sarà disponibile il sito web (si può lasciare `"*:80"`), con l'aggiunta di tutta una serie di campi che servono a definire le caratteristiche della pagina, come ad esempio:

- `ServerName`. Specifica a quale nome è disponibile il server.
- `ServerAdmin`. Indirizzo email dell'amministratore del sito.
- `DocumentRoot`. Specifica la directory che contiene i file del sito, cioè quelli che sono messi a disposizione del client.

Un'altra direttiva utile è la `DirectoryIndex`, la quale permette di specificare il file da prelevare in caso non venga specificato dal client, facendo in modo che l'utente, nel momento in cui digita `www.mysite.com`, riceva in realtà il file `www.mysite.com/index.html` (questa direttiva è abilitata di default e contenuta nel file `etc/.../dir.conf`). Ipotizziamo anche di avere un sito che permette ai singoli utenti di caricare dei file all'interno di una cartella e di renderli disponibili all'indirizzo `www.mysite.com/-username`, il modulo dir mette a disposizione una direttiva `UserDir` che permette proprio di specificare la cartella in cui gli utenti possono caricare la pagina. Questo modulo non è tuttavia abilitato di default e si trova all'interno `/etc/apache2/mods-available/userdir.conf`.

## 2 Multi-Processing Module

Apache permette anche la gestione di più richieste in contemporanea. Per farlo, ricorre a un **Multi-Processing Module**, cioè un modulo responsabile di gestire i socket, fare binding delle porte, accettare connessioni e servirle, eventualmente usando processi thread e figli. Apache supporta diversi MPM, disponibili a seconda del sistema operativo

- **Prefork**. Implementa un server multiprocesso senza utilizzare thread. All'avvio, il padre lancia un certo numero di processi figli (**Pre-Forking**), i figli restano in ascolto, accettano le connessioni e le servono, per poi tornare in ascolto.
- **Worker**. Implementa un server **multiprocesso** e **multithread**. Il processo padre genera un certo numero di processi figli, e ogni processo figlio genera: un thread listener che accetta e smista le connessioni, un certo numero di thread workers che servono.
- **Event**. Una versione migliorata di **Worker**. Oltre ad accettare le connessioni, il thread listener gestisce anche quelle temporaneamente inattive.

Quindi, la prima versione di server apache MPM è la Prefork. Al momento dell'avvio del server, un processo padre lancia un certo numero di processi figli, il quali: rimangono in ascolto, accettano connessioni e le servono, ritornando disponibile dopo aver servito ogni connessione. Il padre gestisce il pool dei figli, cercando di mantenerne sempre alcuni disponibili. Possiamo anche utilizzare delle primitive per configurare questo tipo di server:

- `StartServers`. Indica quanti figli devono essere generati all'avvio.
- `MaxRequestWorkers`. Indica quanti figli massimi possono essere creati.
- `MinSpareServers/MaxSpareServer`. Indica l'intervallo massimo di figli inattivi che possono essere presenti.

- **MaxConnectionsPerChild.** Indica quante connessioni può gestire un figlio prima di essere ucciso.

I vantaggi di questa soluzione sono: una **compatibilità globale** con tutti i moduli di apache, che invece potrebbero non supportare una soluzione basata sul multithreading e una **sistematicità massima**, poiché un problema su un processo causa una perdita della singola connessione. Tuttavia, gli svantaggi principali di una soluzione come questa sono l'occupazione di memoria data dalla creazione di un nuovo processo. Inoltre, è necessario anche prestare attenzione al numero di processi inattivi.

La seconda possibilità è quella del server **Worker**, dove abbiamo diverse differenze: il numero di figli è più flessibile, poiché ogni figlio avrà un numero di worker opportunamente definito per gestire le richieste. Anche in questo caso abbiamo delle costanti che definiscono alcune limitazioni:

- **StartServer.** Indica il numero di processi figli generati all'avvio.
- **MaxRequestWorkers/MinRequestWorkers.** Indica l'intervallo del numero di processi inattivi.
- **MaxConnectionsPerChild.**
- **ThreadsPerChild.** Indica quanti thread possono essere generati da un processo figlio.

L'ultima versione dell'MPM è quella **Event**, che ottimizza l'efficienza generale dell'**MPM Worker** andando a introdurre una gestione delle connessioni inattive:

- Se un thread worker rileva che una connessione è inattiva per un certo intervallo di tempo, restituisce la connessione al listener e passa a gestire un'altra richiesta. Nel momento in cui il client tornerà a comunicare, il listener assegnerà la richiesta ad un altro worker.
- Se un thread worker sta servendo un client con una connessione lenta e il buffer di invio del socket inizia a saturarsi, il worker, invece di attendere, restituisce il controllo del socket al listener che lo assegnerà ad un altro worker non appena sarà nuovamente scrivibile.

# Capitolo 28

## Firewall

I firewall sono dei dispositivi e/o dei programmi applicativi, che permettono il controllo e il filtraggio delle connessioni in ingresso e in uscita applicando delle specifiche regole di **blocco** e **filtraggio**. Esistono due tipologie principali di firewall:

- Uno che opera a livello di rete (**network firewall**).
- Uno che opera a livello di singola macchina (**host-based firewall**).

Esiste un ulteriore divisione tra il funzionamento dei vari tipi di firewall, che riguarda il livello dello stack protocollare a cui lavorano: **Network layer firewall** e **Application Layer firewall**.

### 1 Network-layer Firewall

All'interno del primo tipo di firewall possiamo definire due tipologie di firewall: firewall **stateless** e **statefull**. Nei primi ogni pacchetto viene analizzato singolarmente, solo sulla base di campi statici come indirizzo sorgente o di destinazione. Nei secondi si tiene traccia delle connessioni TCP e degli scambi UDP in corso, discriminando connessioni **leggitive** da connessioni **illeggitive**. Un firewall, indipendentemente dal tipo, definisce al suo interno delle regole di comportamento, contenute all'interno di una tabella, e dette **criteri**; queste regole contengono:

- IP sorgente.
- Porta sorgente.
- IP destinatario.
- Porta destinatario.
- Azione.

L'importanza delle regole è identificata dall'indice a cui si trovano all'interno della tabella, quindi, più l'indice è basso e più la regola è alta. Più nello specifico: per ogni pacchetto il firewall analizza l'header, scorre la tabella delle regole e appena trova una regola che corrisponde alle caratteristiche del pacchetto analizzato, intraprende l'azione corrispondente. L'ultima regola viene detta **regola di default** e definisce un firewall che la applica come **firewall inclusivo**. Il motivo è dato dal fatto che per ammettere una connessione all'interno della nostra rete è necessario definire una regola specifica che lo accetti, cioè che lo **includa**. Se invece il firewall di default ha una regola di accept, allora viene detto **firewall esclusivo**.

## 2 Firewall in Linux

I firewall a livello di macchina in linux possono essere configurati a partire da due programmi: per le funzionalità di **packet filtering** e **NAT/PAT** si utilizza **netfilter**, mentre per le funzionalità di configurazione della tabella delle regole si utilizza **iptables**. Il secondo comando lavora su diverse **tables**, ognuna specifica per una funzionalità. A sua volta, ognuna di queste tabelle, contiene diverse **catene** e, infine, ogni **catena** contiene una lista di regole da applicare a una categoria di pacchetti. Una di queste tabelle, detta **filter**, ha tre catene possibili

- **Catena INPUT.** Per i pacchetti destinati ai processi locali.
- **Catena OUTPUT.** per i pacchetti indirizzati in uscita dai processi locali.
- **Catena FORWARD.** Per i pacchetti che passano attraverso l'host, ma che non sono direttamente destinati all'host.

Un'altra delle tabelle possibili che troviamo all'interno del di linux è quella che permette la gestione del **Network Address Translation** e del **NAP**. La tabella **NAT** ha tre catene:

- Catena **PREROUTING**. Per fare il **D-NAT** dei pacchetti in arrivo, cioè alterare indirizzo/porta di destinazione dei pacchetti in arrivo. Serve per implementare il meccanismo del **port forwarding**.
- Catena **OUTPUT**. Per fare il **D-NAT** dei pacchetti in uscita dai processi locali prima del routing.
- Catena **POSTROUTING**. Per fare il **S-NAT**, cioè alterare indirizzo/porta dei pacchetti in uscita

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

**Gestione delle tabelle su linux** Per visualizzare le tabelle è possibile usare il comando **iptables**. Attraverso questo comando è possibile visualizzare anche singole tabelle e singole catene, è sufficiente specificare il comando in questo modo

```
iptables [-t tables] -L [chains]
```

Sempre con questo comando è anche possibile modificare le regole associate alle singole catene: sarà sufficiente specificare, accanto a **[chains]**, una tra le seguenti **rule specification**:

- Per aggiungere una regola in fondo alla catena:

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

- Per aggiungere una regola in una posizione specifica:

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

- Per rimuovere una regola dalla catena:

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

- Per rimuovere tutte le regole dalla/e catena/e:

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

- Per cambiare la regola di default (*policy*) **DROP/ACCEPT**:

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

Le regole non vengono salvate permanentemente, infatti senza ulteriori comandi le regole importanti vengono eliminate ad ogni riavvio della macchina; in particolare, distinguiamo due comandi:

```
iptables-save > file  
iptables-restore < file
```

### 3 Stateful filtering

Possiamo specificare nella regola un criterio basato sullo stato della connessione TCP di cui un pacchetto fa parte. Questo permette quindi di creare delle regole che includano anche la possibilità di accettare o di rifiutare una connessione anche in base allo stato stesso della connessione, ad esempio: si potrebbe impostare una regola che permette di accedere ad un host via **SSH** solo dal computer dell'amministratore, senza però che questo host possa iniziare le connessioni da solo.

Queste regole possono essere implementate anche per proteggersi dai **SYN ACK**, cioè l'invio massivo di messaggi di **SYN** per cercare di saturare le capacità della rete attraverso dei messaggi di sincronizzazione.