

Appunti Reti Informatiche

Luca Seggiani
2025

1 Lezione del 23-09-25

1.1 Introduzione

Il corso si pone di presentare le nozioni di base sulle reti informatiche, le tecnologie di rete più diffuse, i protocolli Internet e lo sviluppo di applicazioni distribuite *client-server* e *peer-to-peer* (**P2P**).

In particolare il programma del corso comprende:

- Sviluppo di **applicazioni** in rete:
 - Client-server;
 - Peer-to-peer.
- Reti a **connessione diretta**:
 - Collegamenti punto-punto;
 - Reti locali.
- Reti a **commutazione di pacchetto**;
- **Interconnessione** di reti di tipo diverso;
- **Trasporto** end-to-end e protocolli;
- **Sicurezza**;
- Reti **wireless** e **mobili**, intese come caso particolare delle normali reti **cablate** (*wired*).

1.1.1 Applicazioni in rete

Nel dettaglio delle *applicazioni in rete*, vedremo come già detto i paradigmi *client-server* e *peer-to-peer*, di cui possiamo già fare alcuni esempi:

- Applicazioni client-server:

- Web;
- Trasferimento file;
- Posta elettronica;
- DNS;
- Ecc...

- Applicazioni peer-to-peer:

- Ricerca di contenuti;
- Torrent;
- Telefonia online;
- Ecc...

In questo ci avvarremo del concetto di **socket** come primitiva per la gestione della rete dal lato S/O.

1.1.2 Reti dirette, a commutazione e wireless

Inizieremo con lo studio di *collegamenti punto-punto*, e quindi di trasferimento affidabile di dati fra 2 punti. Vedremo poi le reti locali, ad accesso multiplo, e i casi particolari come *Ethernet*.

Vedremo quindi le reti a *commutazione di pacchetto* per la copertura di grandi regioni. Anche qui approfondiremo tecnologie come gli *switch*, ancora *Ethernet*, ecc...

Per quanto riguarda l'*interconnessione di reti* vedremo il protocollo Internet **IPv4**, il **routing** (cioè l'*instradamento*) e i protocolli di trasporto (**UDP** e **TCP**).

Parleremo anche di reti *wireless e mobili*, e quindi di tecnologie come **WiFi**, le **reti cellulari**, e reti senza infrastruttura come **Bluetooth**.

1.1.3 Sicurezza

Vedremo poi le minacce alla *sicurezza* e alcune soluzioni che abbiamo a disposizione per mitigarle. In particolare, tratteremo di **crittografia** e **integrità** dei messaggi.

Nello specifico parleremo di tecnologie a livello applicazione (**PGP**), a livello trasporto (**TLS** (usata in *HTTPS*)), a livello Internet (**IP-Sec**) e difese di sicurezza come **firewall** e **IDS**.

1.2 Terminologia

Iniziamo quindi a definire la terminologia di base usata nel corso, usando Internet come esempio.

1.2.1 Internet

La prima domanda che ci poniamo è "Che cos'è Internet?".

Visione ingegneristica

Iniziamo col vedere la definizione di Internet agli occhi di un ingegnere che si occupa di reti:

- Si tratterà di una rete che connette miliardi di *dispositivi*, detti **host** ("ospiti"), che eseguono *applicazioni in rete* al cosiddetto **edge** ("bordo") della rete.
- Una visione **interna** della rete ci dirà invece che è un insieme di **pacchetti** che viaggiano attraverso infrastruttura (*router, switch*), per raggiungere il loro destinatario.
- A livello **fisico** potremmo considerare le connessioni fisiche fra dispositivi, date da cavi, segnali radio, ecc...
- Infine, potremo organizzare le **reti** come collezioni di dispositivi, router e connessioni gestite da determinate organizzazioni.

Non è esattamente corretto parlare di "*reti di calcolatori*" in quanto oggi ad essere connessi a Internet sono tutta una gamma di dispositivi non necessariamente orientati al puro *calcolo*: è questo il caso del cosiddetto *Internet of Things* (**IoT**).

Possiamo quindi intendere Internet come una "rete di reti", cioè più **ISP** (*Internet Service Providers*) connessi fra di loro, che a loro volta connettono una gamma di dispositivi (host, router, switch, ecc...).

Per governare l'operazione di tali reti si necessita di **protocolli**, che definiscono il modo in cui si inviano e ricevono messaggi in rete.

In particolare, per quanto riguarda Internet notiamo l'**IETF** (*Internet Engineering Task Force*), organizzazione che gestisce diversi standard del settore (anche detti **RFC**, da *Request For Comments memoranda*).

Visione utente

Per l'utente, internet sarà un insieme di **infrastrutture** che forniscono **servizi** finali, fra cui il Web, telecomunicazioni, streaming, ecc... Dal punto di vista delle **applicazioni** in esecuzione sui dispositivi, Internet rappresenterà un'interfaccia di programmazione per consentire la comunicazione fra processi su una o più macchine. In questo parleremo di **hook** che permettono alle applicazioni di **connettersi** a Internet, cioè accedere ad un qualche protocollo di trasporto dei dati.

1.2.2 Protocolli

Un **protocollo** è una precisa *specifica* del formato secondo il quale due dispositivi in rete si scambiano informazioni. Solitamente i protocolli si sviluppano in più fasi, successive nel tempo, dove si portano avanti diverse operazioni necessarie alla comunicazione.

Esempi di protocollo sono il *protocollo Internet IPv4*, e il *protocollo di trasporto TCP* usato nel *Web* e visto nel corso di progettazione web.

1.2.3 Infrastruttura di Internet

Vediamo più nel dettaglio la struttura di Internet:

- Abbiamo detto che all'*edge* di internet ci sono i cosiddetti *host*, cioè i **client** e i **server**. Notiamo che non vogliamo riferirci alle macchine fisiche client o server, ma ai **processi** che si comportano come tali per l'implementazione di un'applicazione distribuita.
- I dispositivi *terminali* che abbiamo appena nominato accedono ad Internet attraverso le cosiddette **reti di accesso**, cablate o wireless e basate sulle tecnologie utilizzate (router).

Per collegare i sistemi terminali ai router si usano *reti residenziali*, *reti di accesso istituzionali* (scuola, lavoro, ecc...), nonché *reti wireless e mobili* (Wifi, o reti come 4G solitamente fornite da privati). In questo caso può interessarci la frequenza di trasmissione, in bit al secondo, di una rete di accesso, o se quella rete è ad accesso *condiviso* (pensa WiFi) o *dedicato* (pensa Ethernet).

Uno standard storico per le reti di accesso è quello della trasmissione sulla linea telefonica su **DSL** (*Digital Subscriber Line*). Negli Stati Uniti si è invece diffuso l'uso della linea televisiva cablata. Oggi, sfruttiamo invece tecnologie come **ADSL** (*Asymmetric Digital Subscriber Line*) e **FTTC** (*Fiber To The Cabinet*). La differenza principale fra queste è che la linea in ADSL è interamente in rame, sia dalla centrale all'armadio di ripartizione che dall'armadio ripartilinea agli utenti finali, mentre nella linea FTTC si porta il segnale all'armadio attraverso cavi in fibra ottica. Lo standard di ultima generazione è **FTTH** (*Fiber To The Home*), che prevede una linea in fibra ottica anche dall'armadio agli utenti finali.

Possiamo quindi vedere la rete locale (**LAN** (*Local Area Network*)) di una comune abitazione come composta da un router, connesso a un **modem** DSL (*modem* deriva da modulatore/demodulatore sulla linea telefonica dei messaggi Internet) o direttamente via cavo ad un altro centro di ripartizione, e ad eventuali dispositivi come

access point WiFi che offrono la connessione via rete mobile ai dispositivi finali (una cosiddetta **WLAN**, *Wireless Local Area Network*).

Altre soluzioni per le comunicazioni wireless sono rappresentati da reti **cellulari** su larga scala, che sono quelle usate dagli operatori telefonici (tecnologie come **4G**, ecc...).

- Dalle reti di accesso si arriva a Internet attraverso reti interconnesse di router, arrivando quindi alle *reti di reti* di cui stavamo parlando.

2 Lezione del 24-09-25

2.1 Comunicazione dati su Internet

Abbiamo visto la struttura a livello fisico della rete Internet. Vediamo adesso i meccanismi secondo cui la trasmissione di dati avviene. La rete Internet è una rete a **commutazione di pacchetto**, il compito degli host è di:

- Ottenere messaggi dalle applicazioni;
- Dividere quei messaggi in frammenti più piccoli, detti **pacchetti**, di dimensione L bit;
- Trasmettere quei pacchetti nella rete di accesso ad una *frequenza di trasmissione* (o **bit-rate**) R .

Il tempo T_{packet} necessario a trasmettere un pacchetto da L bit su una linea da R bit al secondo di bitrate sarà quindi semplicemente calcolato come:

$$T_{\text{packet}} = \frac{L}{R}$$

Il bitrate, detto anche frequenza di *link*, dipende appunto dal **link** (o *mezzo*) della trasmissione, cioè l'infrastruttura fisica che sta fra trasmettitore e ricevitore. Possiamo classificare 2 tipi di link:

- **Mezzi guidati**: segnali che si propagano in mezzi solidi (rame, fibra, cavi coassiali, ecc...).
 - Un celebre esempio di mezzo trasmittivo guidato è il classico **doppino telefonico**, o trasmettitore a *Twisted Pair (TP)*. Questo è formato da due fili di rame isolati e avvolti l'uno sull'altro, che permettono la trasmissione differenziale e quindi la riduzione dei rumori in *common-mode*;
 - Un altro tipo di mezzo trasmittivo guidato è il **cavo coassiale**, formato da due conduttori concentrici in rame separati da un dielettrico. Il segnale è trasferito come campo magnetico fra i due conduttori: questo permette bitrate più alti rispetto al normale doppino telefonico e una migliore schermatura dalle interferenze;
 - Infine, possiamo parlare della **fibra ottica**, formata da fibra di vetro che porta impulsi luminosi ad altissima velocità. Questa tecnologia presenta velocità di trasmissione estremamente alte, e vista la natura luminosa del segnale, non è suscettibile ad interferenze (alta affidabilità, cioè piccola frequenza di errore sui bit).

- **Mezzi non guidati:** segnali che si propagano nell'etere (segnali radio, ecc...). Questi sono solitamente meno sicuri ma significativamente più comodi per l'utente finale (possibilità di spostarsi, mancanza di cavi, ecc...).

La trasmissione wireless è suscettibile a fenomeni fisici come *riflessi*, *interferenze* e *ostruzione* da parte di oggetti fisici.

- Il **WiFi** è un esempio di mezzo di trasmissione non guidato che può raggiungere centinaia di Mbps su regioni locali;
- Reti wireless più ampie possono essere quelle **cellulari**, usate nella telefonia mobile;
- Infine si può parlare delle **reti satellitari**, usate per l'interconnessione di regioni geografiche fra di loro anche molto distanti.

2.1.1 Commutazione di circuito

Prima della commutazione di pacchetto si usava la tecnica della **commutazione di circuito** (ad esempio sulle linee telefoniche). Questo prevede di dedicare completamente una certa linea di trasmissione alla comunicazione fra due host, invalidandone quindi l'uso da parte di altri host.

Chiaramente, la commutazione di pacchetto permette un carico migliore della linea, dove più pacchetti provenienti da diverse fonti possono viaggiare a istanti temporali molto vicini fra di loro.

In particolare, la commutazione di pacchetto è utile per dati trasmessi in *burst*, mentre la rete a commutazione di circuito assicura minima congestione possibile a costo di occupazione completa della linea.

2.1.2 Commutazione di pacchetto

La tecnica della **commutazione di pacchetto** o *packet-switching* permette ad una rete di **router** interconnessi di ricevere ed instradare (letteralmente, "*routing*") pacchetti provenienti da più fonti in modo che raggiungano la loro destinazione.

Questo inserisce chiaramente un ritardo nel sistema, in quanto il router deve:

- Ricevere il pacchetto *completamente* ed memorizzarlo: questo richiede L/R secondi;
- Leggere l'header del pacchetto per capire il prossimo passo dell'instradamento;
- Trasmettere il pacchetto verso la sua nuova destinazione (un altro router o l'host finale), impegnando ancora L/R secondi.

Abbiamo quindi che il ritardo end-end immesso dal router è necessariamente di almeno $2L/R$ secondi, tralasciando il tempo necessario all'instradamento stesso.

Nel caso generale si abbiano N router (quindi $N+1$ link fra i router) e P pacchetti da inviare, dovremo considerare che il primo pacchetto arriva in $(N+1)\frac{L}{R}$ (deve attraversare tutti i link) e i successivi $P-1$ pacchetti arrivano in $(P-1)\frac{L}{R}$, per cui il tempo complessivo è:

$$T_{end-to-end} = (N + P)\frac{L}{R}$$

Se troppi pacchetti arrivano in un breve lasso di tempo, cioè se la frequenza di arrivo supera quella di trasmissione:

- I pacchetti verrano messi in coda finché non sarà possibile trasmetterli;
- I pacchetti possono essere persi se il buffer di memoria dedicato alla loro memorizzazione nel router si riempie.

2.2 Prestazioni della commutazione di pacchetto

Facciamo qualche considerazione ulteriore sulle prestazioni delle linee a commutazione di pacchetto. Abbiamo ottenuto il valore $T_{\text{packet}} = \frac{L}{R}$ per la trasmissione di un singolo pacchetto da L bit su una linea con bitrate R , e $T_{\text{end-to-end}} = (N + P)\frac{L}{R}$ per più pacchetti su un numero arbitrario di router.

Da quanto abbiamo detto nella scorsa sezione, un modello più sofisticato del packet-switching terrà conto di 4 sorgenti di ritardo:

- Ritardo di **trasmissione** T_{trans} , dato dalle caratteristiche del link. Come abbiamo già detto, questo vale:

$$T_{\text{trans}} = \frac{L}{R}$$

con L lunghezza del pacchetto in bit e R bitrate del link;

- Ritardo di **propagazione** T_{prop} , dato dalle proprietà fisiche del mezzo di trasmissione. In particolare, questo è il tempo fisico di trasmissione del segnale su un link, dato da:

$$T_{\text{prop}} = \frac{d}{s}$$

con d distanza del link e s velocità del mezzo di trasmissione. Chiaramente, per i nostri scopi s sarà una frazione significativa della velocità della luce $c \approx 3 \cdot 10^8$ m/s;

- Ritardo di **lavorazione** (instradamento) T_{proc} , dipende dalle caratteristiche del router ed è perlopiù costante;
- Ritardo di **accodamento** dato dalla presenza di code T_{queue} . Questo è il più complicato da trattare, in quanto dipende dal numero di pacchetti presenti nel buffer del router. Come vedremo fra poco, una buona euristica per la valutazione di questo ritardo (che è comunque trattabile solo in maniera statistica) è l'*intensità di traffico* sulla linea di trasmissione.

Sommendo queste sorgenti di ritardo potremo ottenere una stima del ritardo complessivo su un router (nodo) T_{node} :

$$T_{\text{node}} = T_{\text{trans}} + T_{\text{prop}} + T_{\text{proc}} + T_{\text{queue}}$$

Dati N nodi e P pacchetti, il ritardo end-to-end potrà quindi essere calcolato semplicemente come:

$$T_{\text{end-to-end}} = (N + P)T_{\text{node}}$$

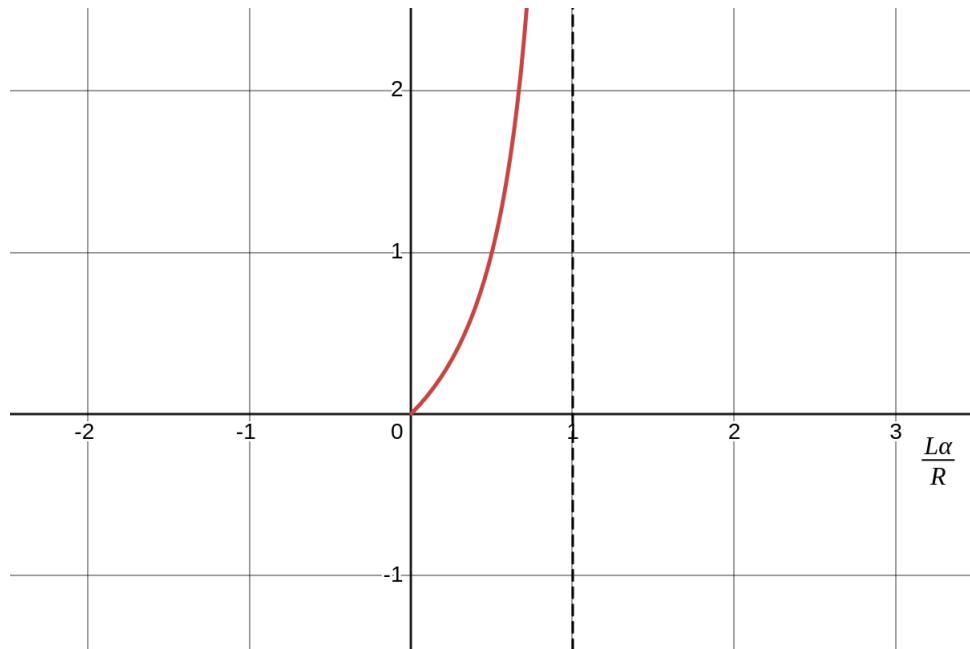
2.2.1 Ritardo di accodamento

Come anticipato, possiamo valutare statisticamente il ritardo di accodamento T_{queue} calcolando l'*intensità di traffico* su una linea:

$$I_{\text{traffic}} = \frac{L\alpha}{R}$$

presa α come la frequenza media di trasmissione di pacchetti sull'istante temporale.

Semplicemente tracciando la funzione si nota che se $I_{\text{traffic}} \geq 1$ il tempo di di accodamento tende a infinito (si ha necessariamente perdita di pacchetti), per cui vorremo mantenere $I_{\text{traffic}} < 1$, e idealmente $I_{\text{traffic}} \ll 1$.



2.2.2 Traceroute

Per fare diagnostica in situazioni reali, si possono usare software di **traceroute**, che inviano pacchetti con lo scopo di tracciare gli host incontrati e misurare il tempo impiegato nella trasmissione *round-trip*, o almeno capire se i pacchetti sono stati inviati o persi.

Più nello specifico, ipotizziamo di avere N router fra sorgente e destinazione. Traceroute invierà $N + 1$ pacchetti speciali sulla linea fra sorgente e destinazione, e ogni router reinvierà il primo di questo che riceve (l' n -esimo) indietro alla sorgente, assieme a informazioni temporali e il suo indirizzo. In questo modo, l' $N + 1$ pacchetto arriverà effettivamente alla destinazione, mentre gli N pacchetti precedenti scandaglieranno ognuno dei router sulla via per la destinazione.

In verità, questo procedimento viene ripetuto 3 volte, per avere risultati più accurati.

Vediamo ad esempio il traceroute che dal portatile su cui sto scrivendo gli appunti arriva ai server dell'Università di Pisa:

```
$ traceroute www.unipi.it
traceroute to www.unipi.it (131.114.104.6), 30 hops max, 60 byte packets
 1  _gateway (10.178.37.4)  3.745 ms  3.552 ms  3.494 ms
 2  172.16.133.129 (172.16.133.129)  300.183 ms  300.156 ms  300.133 ms
 3  172.16.133.38 (172.16.133.38)  60.081 ms  59.673 ms  60.235 ms
 4  172.16.17.50 (172.16.17.50)  52.910 ms  60.413 ms  59.354 ms
 5  151.7.56.70 (151.7.56.70)  61.571 ms  62.259 ms  60.645 ms
 6  151.7.56.66 (151.7.56.66)  89.876 ms  55.029 ms  86.501 ms
 7  151.6.3.199 (151.6.3.199)  52.064 ms  151.6.1.27 (151.6.1.27)  51.777
     ms  151.6.3.199 (151.6.3.199)  44.325 ms
 8  151.6.5.139 (151.6.5.139)  52.076 ms  151.6.1.213 (151.6.1.213)  52.842
     ms  151.6.5.139 (151.6.5.139)  52.399 ms
 9  garr.rom.namex.it (193.201.28.15)  52.662 ms  51.836 ms  52.516 ms
```

```

10  re1-rm02-rs1-rm02.rm02.garr.net (185.191.181.70) 52.317 ms 52.470 ms
    52.278 ms
11  rs1-rm02-rs1-pi01.pi01.garr.net (185.191.181.34) 56.893 ms 56.872 ms
    56.526 ms
12  rs1-pi01-rl1-pi01.pi01.garr.net (185.191.181.39) 56.495 ms 49.338 ms
    50.169 ms
13  rl1-pi01-ru-unipi.pi01.garr.net (193.206.136.90) 49.919 ms 49.808 ms
    50.003 ms
14  jser-jspg.unipi.it (131.114.191.90) 51.377 ms 50.215 ms 51.243 ms
15  * * *
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
21  * * *
22  * * *
23  * * *
24  * * *
25  * * *
26  * * *
27  * * *
28  * * *
29  * * *
30  * * *

```

I numeri a sinistra indicano l'indice del router raggiunto, seguiti dall'indirizzo e le misurazioni temporali nei 3 tentativi effettuati dal traceroute. In particolare, notiamo come i router da 2 a 8 sono anonimi, al 9 siamo arrivati nella rete del GARR, e dal router 15 in poi non abbiamo più ottenuto risposte ai pacchetti traceroute (probabilmente per via della configurazione dei router, il traffico passa comunque).

2.2.3 Throughput

Introduciamo il concetto di **throughput** come la frequenza (bit al secondo) con cui i bit vengono spediti da trasmettitore a ricevitore. Il throughput può essere:

- **Istantaneo:** calcolato ad un certo istante temporale (per quanto possibile dalla natura discreta della trasmissione);
- **Medio:** calcolato su un periodo temporale più lungo.

Ipotizziamo che un server debba inviare un file di F bit ad un client. La linea di comunicazione fra server e router ha capacità di link di R_S bit/secondo, mentre la linea fra router e client ha capacità di R_C bit/secondo. Chiaramente la capacità totale sarà:

$$R_F = \min(R_S, R_C)$$

cioè link con basse capacità di trasmissione rappresentano *bottleneck* per il throughput di tutto il collegamento fra dispositivi.

Facciamo adesso l'esempio di una rete condivisa fra più coppie client/server. In questo caso, date N coppie, la capacità della rete di interconnessione agli occhi di una singola coppia sarà, preso R come la capacità complessiva:

$$R_i = \frac{R}{N}$$

per cui ogni coppia vedrà una linea di comunicazione con capacità:

$$R_F = \min(R_S, R_C, \frac{R}{N})$$

2.3 Struttura di Internet

Abbiamo detto che Internet è una rete di reti.

- Gli host si collegano a internet attraverso le *reti di accesso* (reti LAN, istituzionali, reti mobili, ecc...);
- Le reti di accesso si collegano agli **ISP** (*Internet Server Provider*);
- Gli ISP devono essere collegati ad altri ISP per permettere la comunicazione fra host su diversi ISP (all'interno di **IXP** (*Internet eXchange Point*, ecc...))).

La struttura di reti interconnesse che si viene a formare è molto complessa, e la sua evoluzione è stata guidata da fattori economici e politici.

Abbiamo quindi una struttura gerarchica:

1. Al livello più alto troviamo i cosiddetti **tier 1 ISP** e i **content provider** (Google, Facebook, ecc...) che si occupano di copertura nazionale e internazionale. In particolare, i content provider preferiscono collegarsi direttamente agli IXP per risparmiare sugli ISP;
2. Seguono gli **IXP**, che collegano più ISP fra di loro, e gli ISP locali (regionali, ecc...);
3. Infine troviamo le reti di **accesso** locale (reti LAN, WLAN, ecc...).

2.4 Sicurezza Internet

La struttura di Internet espone i suoi utenti a diversi pericoli, fra cui:

- **Virus**: programmi maligni che si replicano modificando altri programmi;
- **Worm**: programmi maligni che si replicano con lo scopo di diffondersi in altri computer;
- **Spyware**: programmi maligni che cercano di ottenere informazioni che violano la privacy;

ed altre svariate categorie di *malware*.

2.4.1 Attacchi DoS

Un caso tipico di attacchi informatici in rete è quello degli attacchi **DoS** (*Denial of Service*), dove l'attaccante cerca di sfruttare le risorse di un server (memoria, larghezza di banda) al punto di renderlo inutilizzabile ad altri utente. Questo può essere fatto inviando traffico maligno al server (pacchetti molto grandi o corrotti).

La tecnica può essere espansa a più attaccanti (*botnet*) dando vita a tecniche più sofisticate (come il **DDoS**, *Distributed Denial of Service*).

2.4.2 Intercezione di pacchetti

Una problematica delle reti a commutazione di pacchetto è l'**intercezione** dei pacchetti instradati, o in inglese *packet sniffing*. Questo è più facile su reti wireless piuttosto che cablate, ed espone pericoli ovvi per la sicurezza (password, dati sensibili, ecc...).

Oggi, come vedremo, si usano protocolli che implementano forme di crittografia per mitigare questo tipo di problematiche.

2.4.3 Spoofing IP

Il problema di identità fasulle si presenta in Internet attraverso pacchetti malformati, con indirizzi sorgente sbagliati. Queste tecniche, seppure meno pericolose, possono comunque essere usate per confondere o comunque complicare il traffico sulle reti.

3 Lezione del 26-09-25

3.1 Livelli di protocolli

Analizzando l'architettura di Internet abbiamo introdotto il concetto di **protocollo**. Studiando il *core* della rete abbiamo poi visto che la rete è molto complicata e costruita su più *astrazioni*: host inviano pacchetti di applicazioni attraverso router su vari link, ecc...

Possiamo decidere di organizzare i protocolli che governano tali operazioni attraverso una struttura a *layer* o **livelli**. Ogni livello implementa un dato **servizio**, utilizzando le sue azioni interne, e basandosi sui servizi offerti dal livello sottostante.

Questo procedimento non è fuori luogo in sistemi complessi composti da diversi componenti in relazione fra di loro: permette infatti di gestire ogni componente singolarmente, senza doverci preoccupare di rompere la compatibilità con gli altri componenti (basta basarsi sullo stesso servizio sottostante e offrire lo stesso servizio al livello superiore).

Lo stack di protocolli internet che studiamo è quindi il seguente, a 5 livelli:

1. **Application**: il livello che supporta le applicazioni in rete. Protocolli *IMAP*, *SMTP*, *HTTP*;
2. **Transport**: il livello che supporta il trasferimento dati da processo a processo assicurando sicurezza e astrazione sul modello a pacchetto. Protocolli *TCP* e *UDP*;
3. **Network**: il livello che implementa il trasferimento di *datagrammi* da sorgenti a destinazioni. Protocolli *IP*, *routing*;
4. **Link**: il livello che implementa la trasmissione di dati tra elementi di rete fra di loro "vicini". Protocolli *Ethernet*, *802.11 (WiFi)*, *PPP*;
5. **Physical**: il livello fisico rappresentato dai bit sul mezzo di comunicazione.

Un diverso stack è quello presentato dal modello di riferimento **ISO/OSI**. Questo è diverso dal modello presentato prima in quanto presenta qualche livello in più:

1. **Application**: come sopra;
2. **Presentation**: il livello che permette alle applicazioni di interpretare il significato dei dati. Protocolli di *compressione*, *crittografia*, ecc...;

3. **Session:** il livello che permette *sincronizzazione, checkpoint, recupero* di dati, ecc...;
4. **Transport:** come sopra;
5. **Network:** come sopra;
6. **Link:** come sopra;
7. **Physical:** come sopra;

Queste funzioni, se strettamente necessarie, dovranno essere implementate nel livello application.

3.1.1 Incapsulamento di protocolli

Adesso che abbiamo stabilito una gerarchia di protocolli che permettono il collegamento internet, possiamo definire in maniera più precisa cosa accade quando ci colleghiamo, ad esempio, ad un werver HTTP per richiedere una pagina Web.

Dal nostro calcolatore, e in particolare dall'applicazione *browser* in esecuzione sul nostro calcolatore, attraversiamo tutti i livelli (application, transport, network, link, e physical) per arrivare ad un router (probabilmente quello della rete di accesso). Man di mano che scendiamo in livelli più bassi corrediamo il messaggio inviato dall'applicazione con altre informazioni di controllo, utili ai livelli più bassi. A questo punto il router ottiene il messaggio attraverso, magari solo il livello link e physical: questo procedimento si ripete, finché non raggiungiamo il server. Quando il messaggio raggiunge la macchina server, risaliamo tutti i livelli (physical, link, network, transport, application) per arrivare all'applicazione server vera e propria, perdendo nel frattempo le informazioni aggiunte dai livelli sottostanti in fase di trasmissione. A questo punto il server può interpretare il messaggio ed eventualmente rispondere.

Con questa sezione abbiamo concluso l'introduzione al funzionamento (ad alto livello) di Internet, visto sia dall'*edge* della rete (cioè dal punto di vista degli *host*) che dal *core* della rete (cioè dal punto di vista dell'*infrastruttura*) di rete.

3.2 Applicazioni in rete

Veniamo quindi allo sviluppo di **applicazioni** in rete, con l'obiettivo di tornare alle specifiche delle reti in un secondo momento.

In questa sezione vedremo i principi delle applicazioni Web *client-server* e *peer-to-peer* (P2P), in particolare approfondendo i protocolli Web (HTTP), e-mail (SMTP, IMAP), il sistema DNS, nonché la programmazione con l'API dei *socket* coi protocolli TCP e UDP.

Nostro focus sarà quindi il livello applicazione (e in minor parte di trasporto), e gli aspetti concettuali e di implementazione di applicazioni in rete.

Creare un'applicazione in rete significa scrivere programmi che:

- Girano su diversi sistemi;
- Comunicano via la rete.

Le applicazioni vengono scritte per gli **host**: i router non eseguono applicazioni utente, e anzi *non eseguono* nemmeno lo stack protocolare completo (si limitano al livello link e al massimo network). Sviluppare applicazioni per i sistemi all'*edge* della rete permette invece la facile e rapida propagazione delle stesse.

3.2.1 Paradigma client-server

Il paradigma **client-server** è il più comune per le applicazioni in rete. In questo caso individuiamo due agenti principali:

- Il **server** è un host sempre attivo, con indirizzo IP permanente, solitamente distribuito su data center, per permettere scalabilità (qui si sfruttano tecnologie come *load balancer*, ecc...);
- Il **client** è un host che stabilisce contatto intermittente col sever, che può avere indirizzo IP variabile, e che non interagisce mai direttamente con altri client.

Abbiamo quindi che l'identità di client e server è *forte*, la relazione fra i due è asimmetrica e ben definita. In termini di risorse, il client non dovrà avere particolari risorse computazionali, mentre il server dovrà essere capace di gestire le richieste di tutti i client.

Esempi di protocolli client-server sono il protocollo HTTP, i protocolli di posta IMAP e il protocollo di trasferimento file FTP.

3.2.2 Paradigma peer-to-peer

Nel paradigma **peer-to-peer** non esiste un singolo server always-on, ma ogni peer può comportarsi in modalità intermittente sia da client che da server. Questo significa che i peer ricevono servizi da altri peer, fornendo in cambio altri servizi.

Questa caratteristica permette l'*auto-scalabilità*: la rete P2P si sviluppa autonomamente man di mano che si aggiungono peer.

I peer hanno un identità molto più labile rispetto a quelle di client e server tradizionali: l'indirizzo IP può essere dinamico, non sono sempre online, il loro servizio potrebbe essere intermittente, ecc...

3.2.3 Socket

I **socket** sono l'API che implementa la connettività di rete per le applicazioni che scrivremo. Sono offerti dal sistema operativo e rappresentano un'astrazione per la connessione di rete (come i file rappresentano un'astrazione per il disco).

L'analogia tipica del socket è quella di una *porta*. I messaggi entrano dalla porta ed escono dalla porta: quello che sta dietro alla porta è parte dell'infrastruttura implementata prima dal sistema operativo (che implementa i livelli protocolli) e poi dalla rete Internet in sé per sé, ed è astratto via dal programmatore.

4 Lezione del 29-09-25

Continuiamo a vedere nel dettaglio l'ambito dello *sviluppo di applicazioni*.

4.1 Comunicazione fra processi

Abbiamo visto come, sebbene debbano girare su una vasta gamma di dispositivi ed interagire con svariate tecnologie di collegamento, queste possono essere riassunte nei paradigmi **client-server** e **peer-to-peer**. Inoltre, abbiamo visto come maggior parte della complessità dell'infrastruttura di rete sia astratta via nei moderni S/O dietro il meccanismo dei *socket*.

In particolare, vediamo i nostri applicativi come composti da **processi** in esecuzione su macchine fisiche, da cui **processi client** su macchine client e **processi server** su macchine server. Le modalità secondo la quale questi processi si scambieranno **messaggi** saranno rappresentate dalla cosiddetta **IPC** (*Inter Process Communication*).

Notiamo che non è necessario che la IPC sia su macchine diverse, due processi nella stessa macchina possono infatti comunicare fra di loro come se si trovassero su macchine diverse collegate in rete.

Quello che vanno ad implementare i **socket**, standard *de facto* apparso in origine nei sistemi operativi BSD, è appunto l'IPC fra più processi.

4.1.1 Modalità di indirizzamento

Per ricevere messaggi, i processi devono avere degli *identificatori*. Questi sono rappresentati da **indirizzi IP** su 32 bit. L'indirizzo IP della macchina su cui i processi girano non basta, in quanto chiaramente una macchina può avere più di un processo in esecuzione.

4.1.2 Protocolli livello application

Un protocollo di livello *application* dovrà a questo punto definire diverse specifiche sull'IPC:

- Il **tipo** di messaggi scambiati: questi possono solitamente essere *richieste* e *risposte*;
- **Sintassi** dei messaggi: quali campi presentano i messaggi e come i campi sono delineati nella struttura del messaggio;
- **Semantica** dei messaggi: il significato dell'informazione contenuta nei campi;
- **Regole**: su come e quando i processi possono ricevere o inviare messaggi.

Esistono diversi protocolli application, sia **aperti** (interoperabili per applicazioni di più sviluppatori, lo sono il protocollo HTTP, i protocolli di posta, ecc...) che **proprietari** (lo sono ad esempio i protocolli associati a prodotti software proprietari come Skype, ecc...). In particolare, i protocolli aperti possono essere sia **ufficiali** (supportati da agenzie sotto forma di RFC), o **non ufficiali** (basati su documenti non ufficiali ma di pubblico accesso, diventati standard *de facto* dopo grande adozione, ad esempio BitTorrent).

4.1.3 Servizi ad applicazioni

Iniziamo a vedere quali tipi di servizi potrebbero servire ad un applicazione in rete.

- **Integrità dati**: alcune applicazioni (trasferimento dati, transazioni, ecc...) potrebbero richiedere un trasferimento sicuro al 100%. Altre (videogiochi, streaming, ecc...) potrebbero invece poter tollerare perdite parziale dei dati in fase di trasmissione.
- **Throughput**: alcune applicazioni (multimedia, ecc...) richiedono una quantità minima di throughput per essere efficaci. Altre (le cosiddette applicazioni *elastiche*) ne prendono quanto non hanno a disposizione.
- **Tempo**: alcune applicazioni (ancora videogiochi, telefonia online, ecc...) potrebbero richiedere delay temporali molto contenuti per essere efficaci.
- **Sicurezza**: servizi più o meno critici richiedono livelli variabili di sicurezza (crittografia, ecc...).

Su questa base, potremmo classificare alcune applicazioni sulla base dei requisiti di trasporto che hanno:

Applicazione	Perdita dati	Throughput	Tempo	Sicurezza
Trasferimento file/download	Nessuna	Elastico	Indifferente	Dipende
E-mail	Nessuna	Elastico	Indifferente	Dipende
Web	Nessuna	Elastico	Indifferente	Dipende
Streaming	Tollerante	5 Kbps - 5 Mbps	Pochi secondi	Non importante
Videogiochi	Tollerante	5 Kbps - 5 Mbps	Pochi millisecondi	Dipende
Messaggistica istantanea	Nessuna	Elastico	Perlopiù indifferente	Critica

4.1.4 Protocolli livello transport

Potrebbe essere utile, per capire come realizzare le specifiche sopra descritte, studiare ad alto livello i due protocolli di trasporto principali:

- **TCP (Transmission Control Protocol)**: assicura il trasporto *affidabile* di messaggi tra processi, controllo di flusso e di *congestione*, ma ha promesse più scarse nel campo della temporizzazione e del throughput (sebbene assicuri un *throughput minimo*). Orientato alla *connessione* fra processi client e server, è più sicuro dell'alternativa;
- **UDP (User Datagram Protocol)**: meno sicuro e affidabile, non fornisce servizi di controllo di flusso o congestione, né throughput minimo. Questo lo rende adatto per soluzioni a basso overhead, dove le prestazioni sono più significative della correttezza dei dati.

Il controllo del flusso e delle congestioni si collega a quanto visto nella sezione 2.2:

- Il controllo del **flusso** si assicura che il mittente non potrà sovrapporre il destinatario con una mole troppo grande di messaggi;
- Il controllo delle **congestioni** assicura che i ruoter nell'infrastruttura fra mittente e destinatario non vengano sovraccaricati.

Vediamo quindi una tabella riassuntiva delle differenze fra TCP e UDP:

	TCP	UDP
Integrità	Assicurata	Non assicurata
Velocità	Minima assicurata	Massima
Controllo di flusso	Si	No
Controllo di congestione	Si	No
Paradigma	Connessioni	Senza connessioni

Possiamo quindi assegnare ad ognuna delle applicazioni viste nella tabella di sezione 4.13 un protocollo adatto:

Applicazione	Protocollo application	Protocollo transport
Trasferimento file/download	FTP	TCP
E-mail	SMTP	TCP
Web	HTTP	TCP
Streaming	HTTP, DASH	TCP
Videogiochi	WOW, FPS o proprietario	TCP o UDP
Messaggistica istantanea	HTTP o proprietario	TCP o UDP (telefonia)

4.1.5 Sicurezza in TCP

I socket TCP e UDP di base non forniscono particolari funzionalità di sicurezza: i dati sono trasmessi senza crittografia, per cui dati sensibili sono visibili in chiaro.

Si può sfruttare il protocollo (in verità protocollo *middleware*, che sta fra livello transport e livello application) **TLS** (*Transport Layer Security*) per fornire connessioni TCP crittografate, con integrità dei dati assicurata e autenticazione del destinatario.

Possiamo sfruttare TLS in 2 modi principali:

- TSL implementato in applicazione, che a sua volta interagisce con socket TCP;
- API che fornisce socket TLS, che ricevono dati in chiaro e li inviano su Internet crittografati.

La manifestazione più comune del protocollo TLS si vede negli URL delle pagine web, che iniziano con `http://` quando si usa HTTP su TCP puro, e con `https://` quando si usa HTTP su TCP con TLS.

4.2 Web e HTTP

Veniamo quindi a dettagliare la più famosa applicazione sviluppata su Internet, cioè il Web. Come abbiamo visto il Web è supportato dal protocollo **HTTP** (*HyperText Transfer Protocol*).

Ricordiamo quindi che una **pagina** Web consiste di oggetti di diversi formati che possono essere allocati su più Web server. Le pagine in sé per se sono anch'esse oggetti, consistono di file **HTML** (*Hypertext Markup Language*), indirizzabili assieme come tutti gli altri oggetti che le compongono da un **URL** (*Uniform Resource Locator*), in forma:

¹ `<schema>//<nome-host>/<percorso_risorsa>`

ad esempio, <https://www.bittorrent.org/index.html>.

4.2.1 Protocollo HTTP

Il protocollo HTTP è basato sul modello client-server (richiede un protocollo di livello transport che supporti connessioni client-server, quasi sempre TCP). In questo, il client è rappresentato dal *browser*, che compila richieste per ottenere pagine web, mentre il server è rappresentato dal *Web server*, che riceve le richieste dei browser e risponde inviando oggetti.

Nello specifico, una connessione HTTP si svolge come segue:

- Il client inizia la connessione TCP (lato applicazione, crea il socket) col server, alla porta 80;

- Il server accetta la connessione TCP del client;
- Messaggi HTTP vengono scambiati fra client (browser) e server sulla linea TCP;
- La connessione TCP viene chiusa.

Il protocollo HTTP è *privato di stato*, cioè non si mantiene nessuna informazione riguardo alle richieste passate del client.

Esistono 2 tipi di connessioni HTTP:

- **HTTP non persistente**: si apre la connessione TCP e si invia al più un oggetto sulla connessione prima di chiudere. In questo caso scaricare più oggetti richiede più connessioni TCP;
- **HTTP persistente**: si apre la connessione TCP e si inviano più oggetti sulla connessione prima di chiudere.

Definiamo il **RTT** (*Round-Trip Time*) come il tempo che un piccolo pacchetto impiega per viaggiare da client a server e ritorno.

- Nel caso dell'HTTP non persistente, richiediamo un RTT per iniziare la connessione TCP, un RTT per richiedere il file, più il tempo necessario a trasferire il file vero e proprio, per cui si impiega:

$$T_{\text{non-pers}} = 2\text{RTT} + T_{\text{tras}}$$

per ottenere ogni file.

- Nel caso dell'HTTP persistente, dovremmo comunque usare 2 RTT per iniziare la connessione e chiedere il primo file, ma ogni file successivo richiederà solamente il tempo RTT necessario a richiedere la risorsa, per cui risparmieremo tempo.

Chiaramente in questo caso avremo il problema di dover capire *quando* chiudere la trasmissione: magari dopo n richieste, dopo un tempo T (*Timeout*), ecc...

4.2.2 Richieste HTTP

Abbiamo visto come i messaggi HTTP appartengono a 2 tipi, *richieste* e *risposte*. I messaggi sono in formato ASCII, quindi leggibile dall'uomo.

In particolare, l'header di una richiesta HTTP ha la seguente forma generale:

```

1 GET /index.html HTTP/1.1
2 Host: www-net.cs.umass.edu
3 User-Agent: Firefox/3.6.10
4 Accept: text/html,application/xhtml+xml
5 Accept-Language: en-US,en
6 Accept-Encoding: gzip, deflate
7 Connection: keep-alive

```

Ogni riga è terminata da `\r\n`, caratteri di ritorno carrello e nuova linea, non riportati nell'esempio.

La prima riga definisce il **tipo** di richiesta (GET, POST, HEAD, ecc...), la **risorsa** richiesta e la **versione** del protocollo che il client vuole utilizzare. La seconda linea contiene poi l'host del server richiesto, e la terza il processo (qui il browser Firefox) che effettua la richiesta.

Le successive rige definiscono il tipo di oggetto che il client è disposto ad ottenere (tipo MIME, lingua, codifica e compressione, ecc...).

Infine, l'ultima riga stabilisce le regole di connessione richieste dal client, in questo caso `keep-alive` (quindi HTTP persistente).

Un doppio ritorno carrello e nuova linea segnala la fine delle linee di header e l'inizio dei dati veri e propri.

5 Lezione del 01-10-25

Continuiamo la discussione del protocollo HTTP.

5.0.1 Tipi di richiesta HTTP

Esistono più tipi di richiesta HTTP:

- **GET:** richiede una risorsa dal server;
- **POST:** invia informazioni al server, ad esempio per trasferire un form.
- **HEAD:** richiede solo l'intestazione o *header* della risorsa, ad esempio per controllare se ha già la versione più recente in cache;
- **PUT:** aggiorna o rimpiazza una risorsa ad un dato URL. Se non esiste, la crea;
- **DELETE:** Rimuove una risorsa a un dato URL;
- **CONNECT:** Stabilisce una connessione col server. Spesso è utilizzato per connessioni SSL (HTTPS);
- **TRACE:** Risponde con la stessa richiesta. Usata per motivi di debug, ad esempio dal programma `traceroute` visto in 2.2.2;
- **OPTIONS:** Descrive le opzioni di comunicazione per la risorsa interessata. Utile per trovare quali metodi HTTP sono supportati dal server.

Più informazioni sulle specifiche del protocollo HTTP per sviluppatori web possono poi essere trovate in <https://raw.githubusercontent.com/seggiani-luca/appunti-web/481107c15776fac537b7882b6e0becfc88b9886/master/master.pdf>.

5.0.2 Risposte HTTP

Ad una richiesta HTTP su un server web alla porta 80 segue una **risposta**. Questa ha la seguente forma generale. Questa ha un header dalla seguente forma generale:

```
1 HTTP/1.1 200 OK
2 Date: Sun, 26 Sep 2010 20:09:20 GMT
3 Server: Apache/2.0.52 (CentOS)
4 Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT
5 Content-Length: 2652
6 Keep-Alive: timeout=10, max=100
7 Connection: Keep-Alive
8 Content-Type: text/html
```

Anche queste righe sono separate da `\r\n`, caratteri di ritorno carrello e nuova linea.

La prima riga definisce la **versione** protocollo, e la risposta del server in codice (200) e testo (OK).

Esiste un sistema di codici che comunicano le varie situazioni che si possono creare alla risposta. Ad esempio, alcuni dei codici più tipici sono **200 (OK)**, **301 (Moved Permanently)**, **400 (Bad Request)**, **404 (Not Found)** e **505 (Version not supported)**. Vediamo come i codici che iniziano da **100** riguardano *messaggi informativi*, **200** condizioni di *succes-***so**, **300** di *redirezione*, **400** errori *client* e **500** errori *server*. Anche questo argomento viene approfondito all'URL riportato nella sezione precedente.

Seguono poi diverse coppie chiave-valore che contengono informazioni sul contenuto ottenuto, il server che l'ha fornito, ecc...

Possiamo rivolgere la nostra attenzione alle linee `Keep-Alive` e `Connection`: queste stabiliscono il tipo di connessione. Dall'esempio della scorsa lezione avevamo visto come una richiesta può esprimere la preferenza per connessioni `keep-alive` (in HTTP persistente). Qui il server ha consentito la connessione `keep-alive`, stabilendo un timeout di 10 secondi e al massimo 100 richieste HTTP accettate.

5.0.3 Meccanismo dei cookie

Avevamo detto che il protocollo HTTP è *stateless*, cioè privo di stato. Un modo per reintrodurre una qualche nozione di stato nel protocollo è adottare il meccanismo dei **cookie**.

Un cookie è una coppia chiave-valore, o semplicemente anche una sola chiave, che il server invia al client assieme ad una pagina per conservare informazione di stato. Il browser che ha intenzione di preservare lo stato potrà a questo punto annettere il cookie ad ogni richiesta successiva, permettendo al server di "ricordare" quale utente sta lanciando la richiesta.

Prevediamo quindi una nuova linea di stato nell'header delle risposte HTTP (`Set-Cookie:`) che dovrà contenere questi cookie, e una linea simile nell'header delle richieste HTTP (`Cookie:`) che vengono dal browser. A questo punto basterà mantenere un file (o volendo un database) sul browser utente che associa i cookie ad un dominio, e un database (qui obbligatoriamente, probabilmente anche massiccio) di *backend* sul Web server che associa un cookie ad ogni utente.

Con il meccanismo dei cookie il protocollo stateless dell'HTTP diventa effettivamente *stateful*, cioè capace di preservare lo stato fra più richieste (assunto che il browser sia disposto a reinviare ogni volta la stringa di cookie).

5.1 Cache Web

Per ridurre il tempo di accesso agli Web server (che possono trovarsi anche molto lontano dal browser) e ridurre il traffico sugli stessi, si può usare un sistema di *caching*, cioè redirezionare il browser verso una *cache Web* (o **server proxy**).

Un proxy duplica alcuni dei contenuti presenti sul server originale: se possiede la risorsa richiesta dall'utente, la restituisce, altrimenti contatta il server originale, se la procura e la fornisce all'utente. A questo punto mantiene la risorsa per richieste future (chiaramente gestendo la sua memoria, che è finita).

Chiaramente, i *miss* dei proxy sono più lenti di un accesso diretto al server originale, ma di contro la soluzione diventa viabile e effettivamente utile quando si riesce a raggiungere una certa quota di *hit*.

I proxy permettono a content provider meno distribuiti di distribuire contenuti più facilmente. Solitamente sono installati dagli ISP (sia privati che istituzionali), con l'effetto collaterale (positivo) della riduzione del traffico sia sui server originali che sulla linea di accesso a Internet dell'ISP stesso.

Una nota interessante è che il proxy si comporta contemporaneamente sia come **client** che come **server**: dal browser è visto come *server*, mentre dal server originale è visto come *client*.

5.1.1 GET condizionale

Potremmo chiederci come fa li server proxy ad assicurarsi di mandarci sempre la copia più aggiornata della risorsa richiesta.

La soluzione è, lato server originale, non inviare la risorsa se è già presente in cache: il server proxy può usare la linea di richiesta `If-modified-since`: per specificare l'ultima versione che ha a disposizione, e il server può rispondere con una nuova copia (se esiste), o alternativamente con il codice **304 (Not Modified)**.

Questo mantiene chiaramente un piccolo traffico in circolazione sul link, che però è comunque più piccolo del traffico richiesto per inviare risorse complete, e quindi non si traduce in carichi significativi per la rete.

5.2 Protocollo HTTP/2

L'obiettivo degli aggiornamenti del protocollo HTTP è principalmente quello di ridurre il ritardo in richieste HTTP multi oggetto.

HTTP/1.1 ha introdotto richieste GET multiple eseguite in *pipeline* su una sola connessione TCP. Il server risponde *in-order* usando l'algoritmo **FCFS** (scheduling First Come First Served). Secondo questo algoritmo, gli oggetti più piccoli devono aspettare la trasmissione dietro gli oggetti più grandi (**HOL blocking**, *Head Of Line blocking*).

Inoltre, il meccanismo di ritrasmissione dei segmenti TCP persi può rallentare ulteriormente le trasmissioni.

Potremmo pensare di risolvere i problemi di HTTP/1.1 usando altri algoritmi di scheduling:

- Magari inviando prima gli oggetti più piccoli. Questo però ritarderebbe indefiniteamente la trasmissione di oggetti più grandi, in quanto probabilmente ci sarà quasi sempre un'oggetto più piccolo;
- Usando un approccio **RR** (*Round Robin*), dove gli oggetti più grandi vengono divisi in *segmenti* più piccoli, e il server alterna fra i diversi client trasmettendo tali segmenti.

HTTP/2 ha aumentato la flessibilità lato server nell'inviare oggetti al client. In questo caso l'ordine di trasmissione degli oggetti è stabilito da codici di priorità inviati dai client, e da algoritmi più sofisticati lato server (come il RR visto prima). Inoltre, il server può inoltrare (*push*) ai client oggetti che non hanno richiesto.

In HTTP/2 restano comunque i problemi di stallo nel caso di ritrasmissioni di segmenti TCP persi: i browser sono invitati a mantenere più connessioni TCP parallele per ridurre gli stalli e aumentare il throughput complessivo.

Inoltre, non c'è sicurezza sopra il protocollo TCP.

HTTP/3 ha introdotto meccanismi di sicurezza, e controllo di errori per oggetto e congestioni su UDP. Approfondiremo questo aspetto quando parleremo del livello *transport*.

6 Lezione del 03-10-25

6.1 E-mail

L'**e-mail** è un'altra delle applicazioni di largo uso sviluppate su Internet. Come ogni altra applicazione, è supportata da diversi *protocolli*, fra cui **SMTP** (*Simple Mail Transfer Protocol*), **IMAP** (*Internet Message Access Protocol*), **POP** (*Post Office Protocol*, di cui la più nota è la versione *POP3*).

Nel sistema della posta elettronica ci sono 3 componenti principali:

- Gli **user agent**, cioè i client di posta elettronica usati dagli utenti per scrivere e ricevere messaggi di posta elettronica;
- I **server** di posta elettronica;
- Un **protocollo** di posta elettronica, prendiamo SMTP.

I mail server supportano:

- Una **mailbox** che contiene i messaggi in arrivo per ogni utente;
- Una **coda** di messaggi in uscita: questi vengono poi smistati nelle mailbox dei vari utenti;
- Il **protocollo** SMTP per ricevere i messaggi dai client e inviarli ad altri mail server (vedremo che i client non usano direttamente SMTP per richiedere la posta, ma sfruttano altri protocolli).

Come sempre, notiamo che con *mail server* si intende nello specifico il processo che gestisce le richieste degli utenti (e per estensione tutta la macchina (o macchine) che lo eseguono).

Il protocollo SMTP è un protocollo client-server: nonostante ciò, il mail server può comunicare con altri server, e in tal caso si comporta sia come client che come server.

- Si comporta come *client* quando deve inviare posta;
- Si comporta come *server* quando deve ricevere posta.

Questo è chiaro in quanto è il server che *invia* la posta a dover iniziare la comunicazione, mentre il server che *riceve* deve essere solo pronto, appunto, a ricevere.

6.1.1 Protocollo SMTP

Nello specifico, il protocollo SMTP è basato su TCP aperto alla porta 25. Il trasferimento è diretto, e come abbiamo già detto il server che invia si comporta come client per il server che riceve.

Ci sono 3 fasi di trasferimento:

- Handshake fra i due server;
- Trasferimento di messaggi;
- Chiusura della comunicazione.

L'interazione è richiesta/risposta come in HTTP (anche se le richieste vengono dette *comandi*), con messaggi codificati in ASCII a 7 bit.

I comandi contengono solo testo ASCII, mentre le risposte contengono un codice di stato seguito da testo, sempre ASCII, in linguaggio naturale. Questo è per un motivo diagnostico: i comandi client sono letti dalla macchina, mentre del server la macchina legge solo il codice di stato e il commento ASCII è di debug per i tecnici.

Facciamo un'esempio pratico per capire meglio il funzionamento del protocollo. Poniamo che Alice voglia inviare un messaggio a Biagio:

1. Alice `alice@unipi.it` usa l'user agent per comporre un messaggio e-mail al destinatario Biagio `biagio@altroente.edu`;
2. L'user agent di Alice invia il messaggio al mail server di `unipi.it` via SMTP, che lo mette nella coda dei messaggi;
3. Il mail server di Alice apre una connessione TCP, comportandosi come client, con il mail server `altroente.edu`;
4. Il messaggio di Alice viene trasferito verso tale mail server sulla rete TCP;
5. Il mail server di Biagio mette il messaggio nella mailbox di Biagio;
6. In un secondo momento, l'user agent di Biagio richiede la mailbox al suo mail server, che quindi gli invia il messaggio di Alice.

Qui il protocollo SMTP viene usato nella comunicazione fra l'UA di Alice ai server di `unipi.it`, e nella comunicazione fra questi e i server di `altroente.edu`. Per portare la posta dai server di `altroente.edu` all'UA di Biagio, invece, verrà usato un altro protocollo (come già detto IMAP o POP3).

Abbiamo quindi, per riassumere un'ultima volta, che i mail server si comportano come *client* quando devono *inviare* messaggi ad altri mail server, e come *server* quando devono *ricevere* messaggi da altri mail server. Verso gli user agent, invece, sono sempre server: si richiede al server di *inviare* una mail, e si richiede al server di *scaricare la mailbox* corrente (il server non ci contatterà come client per inviarci la posta come farebbe con un altro mail server, ma siamo noi a doverlo invocare).

6.1.2 Parole chiave SMTP

Vediamo l'esempio dei messaggi che viaggiano in una connessione TCP, in plain text, durante lo svolgimento del protocollo TCP. Mettiamo ad esempio di leggere la comunicazione fra il server `unipi.it` al servizio di Alice dell'esempio precedente, quando questo inoltra la posta al mail server `altroente.edu` di Biagio.

Qui s: significa server (`altroente.edu`) e c: significa client (`unipi.it`):

```

1 S: 220 altroente.edu
2 C: HELO unipi.it
3 S: 250 Hello unipi.it, pleased to meet you
4 C: MAIL FROM: <alice@unipi.it>
5 S: 250 alice@unipi.it... Sender ok
6 C: RCPT TO: <biagio@altroente.edu>
7 S: 250 biagio@altroente.edu ... Recipient ok
8 C: DATA
9 S: 354 Enter mail, end with ". " on a line by itself
10 C: Ciao Biagio !

```

```

11 C: .
12 S: 250 Message accepted for delivery
13 C: QUIT
14 S: 221 altroente.edu closing connection

```

Iniziamo con l'handshake: il server invia 220 (tutto ok) e il client si introduce con `HELO` e il suo indirizzo: a questo punto il server risponde con 250 (acknowledge dell'`HELO`).

Da qui in poi il client può comunicare la posta che vuole inviare. Con `MAIL FROM` si indica l'indirizzo di posta elettronica dell'utente che sta inviando posta, a cui segue una risposta 250 dal server che segnala se quell'utente è riconosciuto. Con `RCPT TO` si indica poi l'indirizzo di posta elettronica dell'utente destinatario, a cui segue un'altra risposta 250 dal server che segnala se quell'utente è presente sul mail server.

Segue poi la sezione `DATA`, a cui il server risponde con 354, e quindi la mail vera e propria. Alla fine del messaggio (chiuso con un `.` su una nuova linea) il server risponde con un'altro 250, e il client chiude la connessione con `QUIT`. L'ultimo acknowledge è del server con 221 (chiudo connessione).

6.1.3 Confronto fra SMTP e HTTP

Si può dire che HTTP è un protocollo di tipo **pull**, mentre SMTP è un protocollo di tipo **push**: in HTTP lato client si richiedono risorse, in SMTP si inviano. Inoltre, in SMTP si usano connessioni *persistenti*.

Abbiamo poi che in HTTP ogni oggetto è encapsulato in una risposta, mentre in SMTP si possono avere trasferimenti *multipart* (più messaggi di posta per connessione TCP), e che in SMTP si usa ASCII su 7 bit.

6.1.4 Protocolli di richiesta

Come abbiamo detto, il protocollo SMTP è effettivamente usato solo per la comunicazione fra mail server e per inviare messaggi a mail server: per richiedere la posta dal server si usano altri protocolli come **IMAP** (*Internet Message Access Protocol*) e **POP** (*Post Office Protocol*). Questi protocolli forniscono la possibilità di fare richieste di messaggi, eliminazione, accesso a directory di messaggi presenti sul server, ecc...

Inoltre, servizi come Gmail e Hotmail forniscono interfacce Web (via il protocollo HTTP) costruite su SMTP (per inviare messaggi) e IMAP (o POP, in particolare POP3) per richiedere messaggi.

La differenza principale fra IMAP e POP è che *POP* era pensato per singoli dispositivi: la posta veniva scaricata su locale e rimossa dal server, per cui non vi si poteva accedere da altri dispositivi. *IMAP* risolve questo problema mantenendo i messaggi sul server e concedendone l'accesso da parte di più dispositivi. In verità, oggi la maggior parte dei mail server è configurato per mantenere in remoto anche la posta scaricata via POP3. Per client locali questo significa che IMAP e POP3 distinguono fondamentalmente fra due politiche più o meno aggressive di caching, mentre i client su HTTP usavano in ogni caso IMAP (non si scarica mai nessun messaggio sul disco locale).

Per concludere, da quello che abbiamo appreso in 4.2.1, possiamo dire che POP è *stateless*, mentre IMAP è *stateful* (mantiene stato sotto forma di mail precedenti).

6.2 Il DNS

DNS (*Domain Name System*) è il sistema che usiamo per tradurre *nomi di dominio* DNS, semplici da ricordare per gli umani, in *indirizzi IP*, semplici da usare per le macchine.

6.2.1 Motivazione del DNS

Di base, un’**indirizzo IPv4** (IP versione 4) è rappresentato da un numero a 32 bit, diviso in 4 numeri da 8 bit rappresentati come unsigned e separati da punti: 8.8.8.8 (il servizio DNS di Google) sarebbe 00001000_00001000_00001000_00001000. Un certo numero di bit dal più significativo vengono detti **maschera** di rete, ed identificano la parte dell’IP che identifica la rete locale: i bit successivi identificano gli host. Ad esempio, 10.104.111.163/24 significa che i primi 24 bit identificano la rete, e i successivi 8 gli host su quella rete.

Il problema è che gli indirizzi IP sono difficili da ricordare: gli umani preferiscono indirizzi leggibili come `tizio.caio.com`, cioè i cosiddetti **nomi di dominio**.

6.2.2 Realizzazione del DNS

Per tradurre da indirizzi leggibili a indirizzi IP (necessari per aprire connessioni TCP o UDP e quindi comunicare effettivamente col server) si usa il DNS.

I problemi del DNS sono quindi:

1. Mantenere l’**univocità** dei nomi di dominio verso gli indirizzi IP;
2. Permettere di risalire (**risolvere**) da un nome DNS all’indirizzo IP corrispondente.

La soluzione che è stata data è di creare un *database distribuito* implementato con una gerarchia di tanti *name server* che contengono **registri** di nomi di dominio. Protocolli a livello application permettono quindi agli host (e a loro volta i name server) di **risolvere** nomi di domini DNS in indirizzi.

Questo rappresenta una funzione base di Internet, implementata al livello application, e che distribuisce la maggior parte della complessità all’*“edge”* di internet.

Per assicurare l’univocità si sono formate *autorità* come **ICANN** (*Internet Corporation for Assigned Numbers and Names*), e la sussidiaria **IANA** (*Internet Assigned Numbers Authority*) che definiscono quali domini possono essere usati in base all’area di applicazione, geografica, ecc... Queste autorità delegano quindi ad altre autorità il compito di assegnare nomi, i cosiddetti identificatori *top-level*, **TLD** (*Top Level Domain*) a determinate regioni geografiche (`.uk`, `.io`, ecc...) e gestirne i domini, e la cosa può chiaramente ripetersi ricorsivamente, fino al cosiddetto livello *autoritativo*, dove la traduzione in IP può effettivamente accadere.

In particolare, l’autorità di registrazione dell’identificatore top-level (`.it`) risiede a Pisa all’interno del CNR.

Come sappiamo, esistono anche domini top-level associati non ad entità geografiche ma generici, come `.com`, `.org`, ecc... Questi sono amministrati direttamente dall’ICANN, mentre i registri sono detenuti da compagnie private.

6.2.3 Funzionamento del DNS

La risoluzione dei nomi di dominio viene fatta in maniera simile a quella degli indirizzi IP, partendo da destra verso sinistra per risalire al name server che contiene l’IP cercato. Ad esempio, nel DNS `www.google.com`, prima si controlla il `.com` per capire il primo name server da contattare. Questo a sua volta prenderà il `google` per contattare il prossimo name server, e così via. Il `www` è arcaico e viene mantenuto principalmente per ragioni storiche.

Abbiamo quindi che i problemi prima nominati vengono risolti rispettivamente nei seguenti modi:

1. L'univocità dei nomi di dominio è assicurata da **autorità** nazionali ed internazionali (ICANN, IANA, ecc...);
2. La risoluzione da un nome di dominio DNS all'indirizzo IP corrispondente è assicurata da **name server** che contengono **registri** di nomi di dominio.

Non è detto che le autorità che gestiscono un TLD gestiscono anche il registro corrispondente (abbiamo detto che questo non è il caso per i TLD generali come .com).

6.2.4 Risoluzione DNS

Vediamo i dettagli tecnici. Un server DNS comunica con un attraverso protocollo coi client, fornendo i servizi:

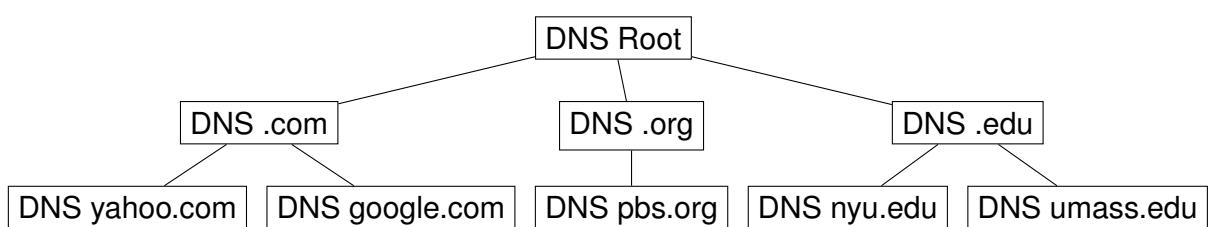
- **Risoluzione** da dominio a indirizzo IP;
- **Aliasing** degli **host**: più nomi possono puntare allo stesso IP;
- **Aliasing** dei server di **posta**: allo stesso dominio possono corrispondere web server come mail server;
- **Distribuzione del carico**: più server (con i loro IP) possono rispondere allo stesso nome di dominio per dividersi le richieste.

Abbiamo detto che il DNS è un database **distribuito**, quindi sviluppato gerarchicamente su più *name server*. Questa soluzione è stata scelta, al contrario di un database *centralizzato*, per una serie di motivi:

- Evita un singolo punto di fallimento;
- Riduce il volume di traffico su ogni name server;
- Il database centrale risulterebbe molto distante per larga parte degli utenti (distribuiti su tutto il mondo);
- La manutenzione è più semplice (si fa su unità più piccole).

6.2.5 Database DNS

La struttura gerarchica del DNS è modellizzabile come una serie di registri che rimanano l'uno all'altro:



I registri ad ogni livello sono raggruppati in diverse categorie:

1. **Root**, il registro base che punta agli altri registri;
2. **TLD** (*Top Level Domain*), che contiene i domini top-level nazionali (.jp, .it, ecc...) o generici (.org, .edu, ecc...);

3. **Autoritativi**, i server DNS che si raggiungono dai TLD (o da server DNS ad altri *sottodomini*) e che hanno effettivamente il compito di terminare la risoluzione per quel dominio restituendo un IP.

Quando un client vuole risolvere un DNS, ad esempio per `www.google.com`, compie i seguenti passi:

1. Si rivolge al root server per trovare il server DNS `.com`;
2. Si rivolge al server DNS `.com` per trovare il server DNS `google.com`;
3. Si rivolge al server DNS `google.com` per ottenere l'IP di `www.google.com`.

In verità, la situazione è più complicata: i server Root sono più di uno (13), e sono largamente replicati (più di 200 solo negli Stati Uniti). Questo è fondamentale in quanto la sicurezza del servizio DNS deve essere assicurata.

Addirittura, i server sia TLD che autoritativi vengono replicati, usando configurazioni *primario/secondario* o meccanismi come *anycast* per assicurare la ridondanza del servizio.

6.2.6 Server DNS locali

Ogni ISP fornisce solitamente un server DNS proprio, detto anche *Default Name Server*. Anche i DNS locali sono solitamente configurati con name server *primario/secondario* da contattare nel caso l'uno o l'altro fallisca.

Quando un host fa una richiesta DNS ad un server DNS locale, questo controlla la sua cache, comportandosi quindi da proxy, e inoltrando la richiesta al DNS Root (o molto più tipicamente a un servizio di risoluzione DNS, soprattutto nel caso di router che per motivi economici non implementano l'intero protocollo DNS) solamente se non è capace di soddisfare la richiesta da solo.

Il meccanismo di caching è supportato dalla struttura gerarchica dei nomi di dominio DNS: ad esempio se il DNS locale conosce `dns.nyu.edu`, potrebbe rispondere alle richieste di risoluzione di `engineering.nyu.edu` inviando richieste direttamente al server autoritativo `nyu.edu`, che ha già contattato.

Abbiamo quindi che un host connesso ad un ISP (come un comune router) si rivolge solitamente al servizio DNS fornito da tale ISP per le sue richieste DNS. Questo quindi usa un servizio di risoluzione pubblico come 1.1.1.1 (Cloudflare) o 8.8.8.8 (Google) per risolvere il DNS (si dice che si comporta da *forwarder DNS*), o in alcuni casi si comporta esso stesso come risolutore DNS contattando direttamente il server Root.

Nessuno ci nega (e a volte avviene) che l'host eviti il DNS locale rivolgendosi direttamente a un servizio di risoluzione.

6.2.7 Risoluzione iterativa/ricorsiva

La risoluzione dei nomi di dominio DNS da parte di un server DNS locale che si comporta come risolutore DNS (come da parte di un risolutore pubblico come quelli nominati sopra) può accadere in due modi principali: **iterativo** e **ricorsivo**:

- **Iterativo**: in questo caso il server DNS si rivolge ad altri server DNS accettando sia traduzioni complete che riferimenti ad altri server DNS: nel caso il server contattato non possa tradurre potrà almeno reindirizzarci verso un server che può farlo;

- **Ricorsivo:** in questo caso il server DNS accetta solo traduzioni complete, o al limite messaggi che segnalano il fallimento della richiesta. Questa è la modalità in cui vengono interrogati la maggior parte dei servizi di risoluzione pubblici.

Dal punto di vista interno la soluzione che si ha è che ogni DNS server contattato si prende la briga di contattare il successivo con una nuova richiesta DNS, e non di restituire solamente l'IP del prossimo elemento della catena, effettivamente sollevando il carico dal server DNS locale.

7 Lezione del 06-10-25

Continuiamo la discussione del DNS.

7.0.1 Caching DNS

Vediamo che anche nel DNS il caching può dare dei vantaggi in prestazioni non indifferentiVediamo che anche nel DNS il caching può dare dei vantaggi in prestazioni non indifferenti.

Quando un qualsiasi server DNS trova una data traduzione fra dominio DNS e indirizzo IP, la memorizza (per ridurre il traffico DNS) con un certo **timeout**, sostanzialmente una data di scadenza della traduzione (solitamente impostata a 2 giorni). Molto spesso i server TLD (che raramente cambiano) sono memorizzati per primi, in questo modo i server Root vengono visitati di rado.

Infine, possiamo assumere che sia il S/O usato dall'utente che utilizza il DNS, che l'applicazione (metti il *browser*) che usa per collegarsi in rete, mantengono una loro cache di traduzioni. Questo significa che l'applicazione, per risolvere un DNS, effettuerà i passaggi:

- Consulterà la sua cache interna;
- Se non trovato, consulterà la cache del S/O;
- Se non trovato, effettuerà un richiesta al server DNS locale;
- Questo cercherà nella sua cache e se non trovato continuerà con i passaggi visti da 6.2.5 a 6.2.7.

7.0.2 Record DNS

Vediamo la struttura dei *Resource Record (RR)* memorizzati dai server DNS.

Questi hanno struttura generale:

```
(name, value, type, ttl)
```

e diversi tipi, fra cui:

- **type=A (Address):** `name` è il nome dell'host, e `value` l'indirizzo IP;
- **type=CNAME (Canonical NAME):** `name` è l'*alias* per qualche nome "canonico", e `value` il nome canonico;
- **type=NS (Name Server):** `name` è il dominio e `value` è il nome di dominio del server autoritativo per questo dominio;
- **MX (Mail eXchange):** `value` è il nome del mail server associato a `name`.

7.0.3 Protocollo DNS

Vediamo alcune specifiche del protocollo DNS usato per inviare richieste (*query*) e risposte (*reply*). Anticipiamo subito che il protocollo DNS si basa su UDP e non TCP.

Abbiamo quindi che la struttura delle richieste e delle risposte è la stessa:

```

1 2 byte          2 byte
2 <id>           <flags>
3 # domande (query)    # risposte RR (reply)
4 # autorita' RR (reply) # RR aggiuntivi (reply)
5 ...

```

L'identificatore (<id>) è su 16 bit e viene usato per riconoscere i destinatari delle risposte. I flag (<flags>) sono vari e specificano:

- Se è un messaggio *query* o *reply*;
- Se si richiede ricorsione (per messaggi *query*);
- Se la ricorsione è disponibile (per messaggi *reply*);
- Se la risposta è authoritative, cioè viene da un server DNS authoritative.

7.0.4 Inserire record in DNS

Vediamo quindi il processo usato per registrare nomi di dominio nel DNS. Mettiamo che la startup "Alpha S.r.l." voglia registrare un dominio per il suo sito:

- Dovrà registrare il nome rivolgendosi ad un'azienda specializzata, e fornendo il nome di dominio (che non deve essere già stato registrato), l'indirizzo IP del server authoritative primario e secondario;
- L'azienda specializzata inserirà record di tipo NS e A in un server TLD (mettiamo .com):

```

1 (alpha.com, dns1.alpha.com, NS) # RR NS dal nome di dominio al DNS
      autoritativo
2 (dns1.alpha.com, 212.212.212.1, A) # RR A dal nome del DNS
      autoritativo al suo indirizzo

```

- A questo punto Alpha S.r.l. dovrà configurare i suoi server DNS authoritative inserendo record di tipo A e volendo MX (per la posta elettronica):

```

1 (www.alpha.com, 212.212.212.2, A) # RR A dalla pagina web all'IP del
      server che la fornisce
2 (alpha.com, mail.alpha.com, MX) # RR MX al nome del mail server
3 (mail.alpha.com, 212.212.212.3, A) # RR A dal nome all'IP del mail
      server

```

7.1 Applicazioni P2P

Abbiamo visto una certa gamma di applicazioni in rete che adottano il paradigma client-server: ad esempio il Web, l'e-mail, e proprio adesso il DNS.

Vediamo adesso l'altro grande paradigma, quello del **Peer-to-Peer**. Ricordiamo che la caratteristica delle applicazioni P2P è che non c'è un server centrale sempre attivo, ma sistemi host finali comunicano fra di loro in maniera arbitraria: i *peer* richiedono servizi ad altri peer, fornendo in cambio altri servizi.

Questa architettura è *auto-scalabile* (più peer, più grande la rete), complessa da gestire e (vogliamo) il più possibile decentralizzata.

Ci sono diverse applicazioni realizzabili come P2P; possiamo classificarne alcune:

- **Condivisione contenuti:** i peer rendono disponibili contenuti multimediali o in generale file ad altri peer, in maniera decentralizzata. Dal punto di vista legale questo fornisce una piattaforma abbastanza sicura per lo scambio di contenuti illegali (film e musica pirata, ecc...). Celebre è l'esempio di *Napster*;
- **Messaggistica istantanea:** gli *username* degli utenti devono essere mappati a certi indirizzi IP. Quando un utente va online un *indice* è notificato col suo indirizzo IP: a questo punto altri utenti possono contattarlo in P2P a tale indirizzo.

7.1.1 Indici P2P

Vediamo che in entrambi casi presentati nella scorsa sezione (nel primo non si è detto ma è chiaro riflettendo sulle soluzioni tecniche) abbiamo bisogno di **indici**: per stabilire quali peer contattare per ottenere una data risorsa, bisogna avere un modo di effettuare ricerche sui loro indirizzi IP.

Un indice P2P è organizzato come un database di coppie chiave-valore, ad esempio:

```
Led Zeppelin IV, 203.17.123.38
```

I peer effettuano *query* a questo database per trovare i peer che possiedono date risorse; i peer possono anche *inserire* nuove coppie chiave-valore (ad esempio per segnalare che possiedono una nuova risorsa).

Il database può essere realizzato in più modi:

1. Si può dislocare un singolo **indice centralizzato** per tutti i peer. I problemi di questo sistema sono ovvi: questo può essere facilmente individuato e messo fuori uso (cause legali, guasti, ecc...) o sovraccaricato per rendere l'intero sistema P2P inutilizzabile.

Diciamo infatti che applicazioni P2P con indici centralizzati sono P2P "*ibridi*", con funzionalità client-server per l'ottenimento degli IP dei peer.

Un sistema di questo tipo veniva usato da *Napster*, con celebri risultati (il server centrale viene chiuso e l'intera applicazione cade);

2. Si può sfruttare il **query flooding** per realizzare un indice *decentralizzato*: secondo questo paradigma ogni peer forma una parte dell'indice.

In questo caso la fase di distribuzione di contenuti è banale: non bisogna collegarsi ad un indice centralizzato, ma ogni peer diventa parte dell'indice nel momento stesso in cui inizia a condividere un contenuto.

Il problema giunge in fase di ricerca di contenuti: abbiamo che il client che deve ricevere contenuti deve limitarsi a *chiedere* agli altri peer "*vicini*" se hanno il contenuto desiderato. Definiamo i peer come *vicini* se hanno fra di loro una connessione TCP attiva. La richiesta può chiaramente diffondersi di vicino in vicino visitando l'intera rete o un suo sottoinsieme, e in questo modo si implementa effettivamente, se non per "forza bruta", un indice decentralizzato.

La rete che si viene a formare fra tutti i peer vicini in un sistema P2P di questo tipo viene detta **rete overlay**.

Per ridurre il traffico sulla rete P2P (sistemi di questo tipo tendono a emanare molto traffico) si può usare il **limited-Scope** query flooding, cioè inviare richieste non a tutti gli amici, ma a un sottoinsieme, magari impostando un time-to-live per le richieste oltre il quale i vicini al prossimo "hop" non necessitano più inoltrare la richiesta oltre.

Un'altro problema è chiaramente il *bootstrapping*: come trovare il primo set di vicini? In questo caso si possono usare scansioni, cache temporanee di peer, peer preimpostati (che hanno però il problema della centralizzazione) o l'intervento manuale dell'utente.

Questa era la soluzione usata da servizi come *LimeWire*;

3. Un'approccio che combina il meglio dei due scorsi è un'approccio **gerarchico**. In questo caso prevediamo una rete simile a quella usata nel query flooding, ma che prevede dei *Super Nodi (SN)* (o *Super Peer, SP*), cioè peer con alta bandwidth e disponibilità alle connessioni. Ogni SN gestisce una rete locale, con un suo indice locale, e i peer di quella rete lo interrogano come avrebbero interrogato il server centrale del primo caso. A questo punto la trasmissione avviene normalmente fra peer, ammesso che i peer siano all'interno della stessa rete locale gestita dal SN. Non si esclude la possibilità che un SN non trovi il contenuto richiesto nel suo indice locale: in questo caso è libero di collegarsi ad altri adottanto sostanzialmente il query flooding, che però sarà più efficiente per diverse ragioni (meno supernodi formano una rete più compatta, i supernodi hanno specifiche migliori di trasmissione, i supernodi sono pensati appositamente per questa operazione (a differenza dei peer normali che vogliono perlopiù scaricare e non dare), ecc...).

Chiaramente il problema sarà di non "*centralizzare*" troppo i SN, cosa che li renderebbe obiettivi di attacchi, cause legali, o comunque renderebbe più fragile l'intero sistema.

Questa soluzione veniva usata da servizi successivi a *LimeWire*, come ad esempio *FastTrack*;

4. Un approccio interessante è quello dato dalle **DHT** (*Distributed Hash Table*). In questo caso si mira a realizzare l'associazione chiave-valore attraverso un'**hash table distribuita** fra i nodi di un *overlay network*.

Ci dotiamo di una funzione di hashing *consistente*, cioè dove la rimozione di un bucket (di un bersaglio per la funzione) richiede lo spostamento delle sole chiavi che arrivavano a quel bucket. Usiamo quindi questi bucket per partizionare i peer: a certe chiavi corrisponderanno certi peer che conterranno i dati associati alle chiavi.

A questo punto la ricerca di contenuti si può fare come per il query flooding, cioè inviando richieste ai vicini finché non si trova un degli host che corrisponde alla chiave cercata (e quindi i contenuti desiderati). Una soluzione più intelligente è però strutturare la *topologia* dell'overlay network in modo che i salti possano essere informati: questo è fattibile quando si usa un apposito algoritmo di hashing, dove ogni nodo ha una qualche informazione su quali dei suoi nodi vicini sono più o meno "*vicini*" alla chiave cercata.

Questo è l'approccio usato da *BitTorrent* e *eMule*.

8 Lezione del 08-10-25

8.1 Confronto prestazionale fra CS e P2P

Vediamo di valutare quale approccio, fra client-server e peer-to-peer, per un job semplice come un trasferimento di file *minimizza il tempo di trasferimento*.

Poniamo di avere N peer (o client, comunque n host che vogliono ricevere il file) e un server che contiene il file (di dimensione F). All'istante in cui il file viene reso disponibile ogni peer richiede il file, e vogliamo minimizzare il tempo che passa affinché tutti lo abbiano ricevuto.

Sia u_s la capacità di upload del server, u_i la capacità di upload dell' i -esimo peer e d_i la capacità di download dell' i -esimo peer.

- Con l'approccio **client-server**, il server deve inviare N copie del file di dimensione F , che con capacità di upload di u_s dà un tempo minimo di NF/u_s . Ogni client deve quindi ricevere la sua copia del file, che con capacità di download di d_i diventa F/d_i . Visto che il più lento è quello che pregiudica tutta la statistica, prendiamo la sua capacità di download come d_{\min} e rifiniamo il bound:

$$T_{cs} \geq \max \left(\frac{NF}{u_s}, \frac{F}{d_{\min}} \right)$$

Notiamo che questo bound è $O(N)$.

- Con l'approccio **peer-to-peer**, il server dovrà inviare al minimo una copia del file, per cui il bound diventa F/u_s . Il client più lento sarà comunque un bottleneck, per cui c'è il termine F/d_{\min} . A questo punto l'ultimo bound viene preso come il tempo impiegato a trasferire il file a tutti, se tutti distribuiscono, cioè $NF/(u_s + \sum u_i)$. Questo dà il bound:

$$T_{p2p} \geq \max \left(\frac{F}{u_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum u_i} \right)$$

Vediamo che il termine significativo è l'ultimo: il primo non dipende da N , mentre il secondo lo fa in maniera "lasca" (aumentando N troveremo host più lenti, ma non linearmente e prima o poi stabilizzandoci). Il terzo termine dipende da N , ma ancora non linearmente. Posto \bar{u} come la velocità media di upload dei peer, possiamo approssimare come:

$$T_{p2p} \sim \frac{NF}{u_s + \sum_1^N u_i} \approx \frac{NF}{u_s + N\bar{u}}$$

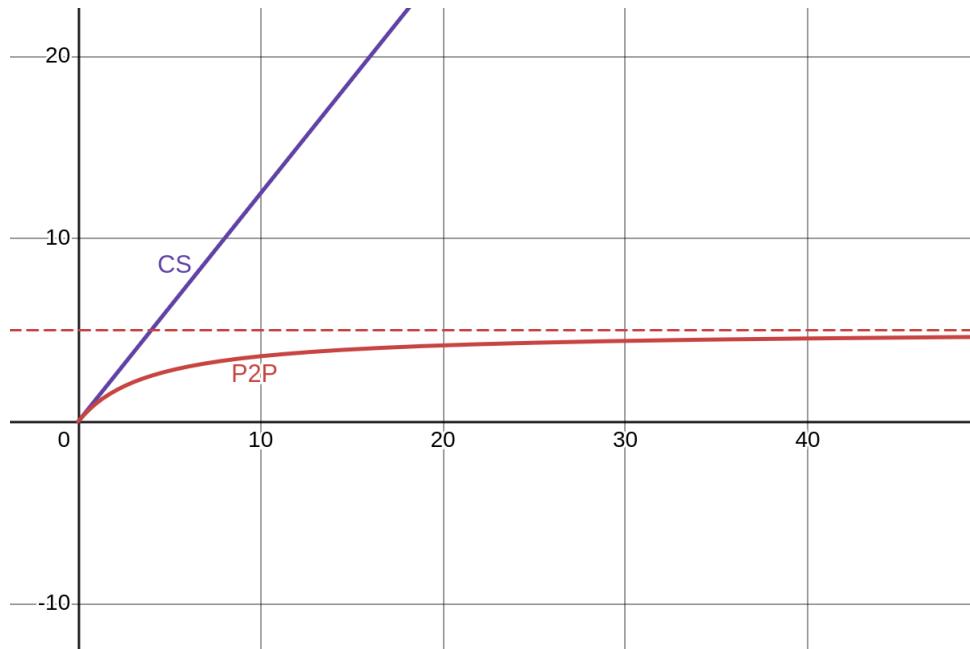
che per $N \rightarrow \infty$ è:

$$\lim_{N \rightarrow \infty} \frac{NF}{u_s + N\bar{u}} = \frac{F}{\bar{u}}$$

ammesso di conoscere \bar{u} .

Abbiamo quindi che il bound è migliore rispetto a quello dell'approccio client-server, in quanto prima o poi converge ad un valore finito.

Concludiamo tracciando su un grafico l'andamento complessivo dei bound più svantaggiosi dei tempi di trasferimento:



8.2 Protocollo BitTorrent

Approfondiamo le reti P2P, e le *overlay network*, studiando un protocollo reale: il **protocollo BitTorrent**.

In BitTorrent non è previsto un server centrale che distribuisce il file. Ogni file è diviso in più **chunk**, solitamente di 256 KiB ciascuno.

Un **torrent** è un gruppo di peer che si impegnano a scambiarsi chunk di un file. Un server più o meno centralizzato detto **tracker** traccia i peer che partecipano ad un torrent.

- Quando un utente vuole scaricare un file, chiede al tracker la lista dei peer che partecipano al torrent dedicato a quel file. Solitamente questo è un *metafile* che contiene l'IP del tracker, a cui si può quindi richiedere il torrent vero e proprio.
- I peer che l'utente riceve vengono detti **vicini**: sono connessi da una connessione TCP, e possono quindi scambiarsi dei messaggi (chunk del file). Possiamo definire così gli utenti collegati ad un torrent:
 - **Leecher**: ("*sanguisughe*"), non hanno una copia completa del file e lo stanno ancora scaricando;
 - **Seeders**: ("*alimentatori*"), hanno già il file, e lo rendono disponibile agli altri.
- Il trasferimento da seeder a leecher si fa un chunk per volta, adottando una politica **Rarest-First (RF)**: si inizia a scaricare dal chunk che meno peer sul torrent hanno disponibile. Questo è chiaramente mirato a ottenere i chunk che potrebbero sparire dalla rete il prima possibile.

La lista dei chunk disponibili è fatta richiedendo periodicamente a ogni peer quali chunk possiedono.

- Man di mano che i leecher ottengono chunk, devono anche iniziare a rimetterli nella rete ai leecher nuovi arrivati. Su BitTorrent si usa la politica **Tit for Tat (TT)**: si inviano chunk ai 4 peer che ci stanno inviando chunk a frequenza più alta. Questi top 4 vengono ricalcolati ogni 10 secondi, e gli altri vengono "strozzati" (non più serviti).

Questo significa che i peer potrebbero legarsi eccessivamente fra di loro: ogni 30 secondi si sceglie allora un'altro peer a caso, facendo una previsione *ottimistica* (quel peer potrebbe arrivare fra i nuovi top 4 più frequenti).

In ogni caso, l'obiettivo è quello di trovare buoni partner per il trasferimento file, in modo da ottenere il file più velocemente possibile.

La politica *Tit for Tat* di BitTorrent è chiaramente pensata per scoraggiare i *freeloader*: non si può scaricare se non si dà qualcosa in cambio, cioè si partecipa attivamente al trasferimento rimettendo chunk in circolo.

8.3 Streaming video e CDN

Vediamo come si possono creare tecnologie per il trasferimento massiccio di contenuti multimediali, con riferimento allo **streaming video** e i **CDN** (*Content Distribution Networks*).

I problemi sono la **scala** (come raggiungere molti utenti?) e l'**eterogeneità** (diversi utenti hanno diverse caratteristiche di trasmissione, dispositivi di visualizzazione, ecc...).

La soluzione è infrastruttura **distribuita** al livello application.

8.3.1 Video

Un **video** è una sequenza di immagini dette **frame** mostrate a frequenza costante (standard 24, 30, e 60 **FPS** (*Frames Per Second*)). Un **frame** è un'array di pixel, dove ogni pixel è rappresentato da 3 valori (corrispondenti ai colori primari additivi *rosso*, *verde* e *blu*).

La tecnica del **coding** consiste nell'usare ridondanza intrinseca dentro e fra i frame per ridurre i bit usati nel processo di codifica:

- Si usa la correlazione **spaziale**, dentro i singoli frame;
- Si usa la correlazione **temporale**, fra diversi frame.

Esistono più tipi di coding, fra cui notiamo:

- **CBR** (*Constant Bit Rate*), a bitrate costante;
- **VBR** (*Variable Bit Rate*), a bitrate variabile;

8.3.2 Streaming

Lo **streaming** di video allocati in remoto consiste nel spedire contenuti video abbastanza velocemente da renderli disponibile in *tempo reale* (a differenza del semplice download, che prevede una fase iniziale di acquisizione e una seconda fase di visualizzazione quando il file è ormai già tutto sul disco locale).

Il server dovrà quindi inviare al client frame con una frequenza prefissata (quella del contenuto video), e il client potrà mostrarli appena arrivati. Assunta una rete ideale,

quindi a ritardo *costante*, questo ci permetterebe di visualizzare il video così com'è (al pari di tale ritardo costante).

Purtroppo la rete non è ideale, e il ritardo è quindi *variabile* nel tempo. Possiamo quindi sfruttare il processo del **buffering**: aspettiamo un po' lato client prima di mostrare i frame (per un tempo detto *client playout delay*), mettendoli nel frattempo in un *buffer* di memoria. Quando iniziamo a visualizzare i frame, ci aspettiamo che il buffer sia abbastanza " *pieno*" da permettere la visualizzazione di un segmento di video abbastanza lungo da permettere l'acquisizione dei frame successivi, e così via.

9 Lezione del 10-10-25

Torniamo sull'argomento dello streaming video.

9.1 Tecniche avanzate per lo streaming

Avevamo introdotto l'idea del **playout delay**, dove bufferizzaviamo i dati video in ingresso per "spostare in avanti" in qualche modo la zona di dati compromessa dal *jitter* (irregolarità) della rete.

Visto che il jitter può essere massiccio, e la quantità di dati da inviare non indifferente, esistono alcune strategie più effiaci come ad esempio **DASH**.

9.1.1 DASH

DASH (*Dynamic Adaptive Streaming over HTTP*) è una tecnologia che prevede di memorizzare i file video in più chunk, ognuno memorizzato con ridondanza a diversi bitrate. Un *manifest file* fornirà gli URL di tutti i chunk, individuati quindi da indice nel video e livello di compressione.

Il *client* dovrà a questo punto misurare la velocità di trasmissione del server, e quindi consultare il manifest file: sulla base delle misurazioni fatte richiede il chunk di qualità massima sostenibile alla largezza di banda corrente. In questo modo può variare la qualità del video nel tempo, in modo da rispondere in maniera abbastanza elastica alle irregolarità della rete.

Questo richiede "*intelligenza*" dal client: questo deve sapere:

- *Quando* richiedere i chunk in modo da non incorrere né in starvation (restare senza chunk) né in overflow (non avere più spazio per i chunk);
- *A che bitrate* richiedere i chunk sulla base della banda disponibile;
- *Dove* richiedere i chunk: se i server sono duplicati dovrà scegliere il più efficiente.

9.2 CDN

Vediamo un'altra tecnologia, questa volta riguardante la *scalabilità* dei server che forniscono gigantesche quantità di contenuti video a numeri altrettanto grandi di utenti, simultaneamente, in tutto il mondo. Questo è il problema che viene affrontato dai **CDN** (*Content Distribution Network*).

L'idea è di copiare e servire diverse copie dei video su diversi server in siti geograficamente distribuiti. Esistono due approcci per la dislocazione di tali server:

- **Enter deep**: usare server CDN vicini agli access network: l'idea è di spostarsi il più vicino possibile agli utenti;

- **Bring home:** meno cluster, più grandi, distribuiti in **POP** (*Points Of Presence*) vicini ma non *sulla* rete di accesso degli utenti.

Facciamo un'esempio pratico, guardando al processo che un servizio di distribuzione segue per fornire un contenuto a un utente. Abbiamo che il meccanismo principale di redirezione è realizzato sfruttando il DNS:

- Il CDN (diciamo `kingcdn.com`) mantiene più copie del contenuto su diversi *nodi* CDN (magari `kingcdn.com/<slug-contenuto-1>`, `kingcdn.com/<slug-contenuto-2>`, ecc...).
- Il servizio di distribuzione rende disponibile il contenuto a `video.netcinema.com/<slug-contenuto>`;
- L'utente richiede un contenuto dal CDN, collegandosi a `netcinema.com` e navigando fino al contenuto richiesto a `video.netcinema.com/<slug-contenuto>` (l'opzione più probabile è che questo contenuto si trovi in un blocco HTML di tipo `<video>`);
- A questo punto, il DNS locale dell'utente chiede l'IP di `video.netcinema.com` ai server autoritativi di `netcinema.com`, che risponde con un record CNAME corrispondente all'IP del server autoritativo di `kingcdn.com`;
- L'utente viene quindi rediretto verso il nodo più vicino (potrebbe anche scegliere un'altro percorso se si verificassero variazioni (in peggio) del servizio). Da qui in poi il video è trasmesso dal nodo del CDN su HTTP.

9.3 Programmazione socket

Abbiamo introdotto il concetto di **socket** (dall'inglese per *presa*) come astrazione della comunicazione in rete per i processi in esecuzione su un S/O.

Lato pratico, il S/O renderà disponibili chiamate a sistema per effettuare almeno alcune operazioni base:

- Apertura di un socket: questo potrà essere **TCP** o **UDP**;
- Invio di dati al socket (o scrittura sul socket);
- Ricezione di dati al socket (o lettura dal socket).

Ricordiamo che il TCP è un protocollo orientato alla connessione, sicuro e affidabile, mentre UDP è un protocollo semplice per il trasferimento diretto ma non affidabile di dati.

9.4 Reti in connessione diretta

Finite le applicazioni, riprendiamo il discorso dal *basso*. Se avevamo discusso come creare applicazioni che sfruttassero la rete, adesso vogliamo discutere come la rete riesce in primo luogo a trasmettere bit fra più processi, o anche solo fra più calcolatori.

Iniziamo col considerare come nodi **adiacenti** nella rete comunicano fra di loro, cioè come 2 host con apposite interfacce di rete comunicano attraverso una connessione diretta: il caso più semplice è quello di un link punto-punto fisico.

Ci sono già diversi problemi da analizzare: come rendere la comunicazione **affidabile** (*framing, rilevamento errori e correzione*, ecc...). Introdurremo quindi il protocollo **PPP** (*Point to Point Protocol*), i protocoli ad **accesso multiplo**, e le **LAN** (*Local Area Network*).

9.5 Link punto-punto fisici

Supponiamo di avere 2 host (2 macchine) che vogliono comunicare fra di loro. Questi saranno provvisti ciascuno di un interfaccia per una porta seriale che potranno comandare (alto o basso), e le loro porte seriali saranno collegate da un link di qualche tipo (rame, fibra ottica, ecc...).

Le porte seriali permettono la **codifica** di bit attraverso variazioni di stato del mezzo fisico del link.

L'interfaccia della macchina mittente è detta **trasmettitore**, mentre l'interfaccia della macchina destinatario è detta **ricevitore**: bisognerà stabilire un protocollo per la comunicazione fra questi.

In questo caso il protocollo è semplice il *trasmettitore* dovrà occuparsi di **codificare** sequenze di bit in variazioni di stato sul mezzo di link, mentre il *ricevitore* dovrà occuparsi di **decodificare** tali variazioni di stato riportandole in bit sulla macchina destinatario.

9.5.1 Data link

Una caratteristica fondamentale dei link è che sono *inaffidabili*: il canale di trasmissione non è mai ideale (degradazione del segnale, rumore, interferenze, ecc....) e quindi la sequenza codificata potrebbe arrivare al ricevitore radicalmente cambiata rispetto a quella iniziale.

Chiameremo *bit error rate* la frequenza di bit persi sul mezzo di comunicazione.

Per gestire l'inaffidabilità intordurremo un livello superiore a quello di **link fisico**, detto di **link dati** (*data link* o anche *livello logico*).

- La prima cosa che prevede il data link è il **framing**: si divide la comunicazione in *payload* (insiemi di bit), e si incapsulano i tali payload fra *header* e *trailer* (sostanzialmente intestazioni e terminazioni), andando a formare il cosiddetto **frame**. Da qui in poi, con frame intenderemo sia l'intero frame che il suo payload a seconda del contesto.

Nell'**header** potremmo inserire diverse informazioni: prima fra tutte l'indice del frame nell'intero blocco di dati da trasferire.

Se la frequenza di errore è comparabile con la lunghezza dei frame, avremo sicuramente perdite di dati: riducendo la lunghezza dei frame potremmo avere più fortuna.

Questa tecnica introduce chiaramente un certo *overhead* di trasmissione, dato dal dover introdurre header e trailer. Ci permette però di dividere la trasmissione in unità contenute, su cui è più semplice fare controllo degli errori;

- Altri servizi sono forniti per il controllo degli errori di trasmissione: l'**error detection** mira a rilevare gli errori, mentre l'**error correction** mira a rilevarli e correggerli. Questo è piuttosto semplice nel caso di trasmissioni binarie: assunto di avere un rilevamento dell'errore granulare al bit, correggerlo significherà semplicemente invertirlo.

Una soluzione alternativa è comunque quella della **ritrasmissione** dei dati in errore, che implica però una comunicazione all'*indietro*, dal ricevitore al trasmettitore per la segnalazione dei frame corrotti.

- Ci sono poi altri servizi che potremmo voler fornire: ad esempio il **controllo di flusso** che permetta di moderare in qualche modo la frequenza di trasmissione.

Ad esempio il ricevitore potrebbe voler segnalare al trasmettitore di dover ridurre il bitrate, magari per problemi di overflow.

- Infine, vogliamo distinguere se offriamo servizi **half-duplex** o **full-duplex**:
 - **Half-duplex**: entrambi i nodi possono trasmettere, ma solo uno per volta;
 - **Full-duplex**: entrambi i nodi possono trasmettere, in *parallelo* (quindi contemporaneamente).

Configurazioni dove solo un nodo può trasmettere sono dette **simplex**.

9.5.2 NIC

Il livello data link è implementato nella cosiddetta **NIC** (*Network Interface Card*) all'interno di ogni host sulla rete. Questa implementa il livello data link e di conseguenza il livello fisico. Può essere Ethernet, WiFi, ecc...

La NIC è collegata al bus periferiche di un calcolatore (ad esempio bus PCI) e implementa una o più delle funzionalità offerte dal livello data link. Fa questo combinando *hardware, software e firmware*: l'importante è che sia capace di offrire al calcolatore i datagrammi inviati al livello data link (in questo sia il NIC che il calcolatore vedono il livello data link).

10 Lezione del 13-10-25

10.1 Rilevamento errori

Il *rilevamento degli errori* (**error detection**) è il meccanismo attraverso il cui verifichiamo se i dati arrivati attraverso il link fisico sono corretti.

Dobbiamo innanzitutto stabilire un codice di **rilevamento** degli errori. Per ogni frame di livello Datalink trasmettiamo d bit di dati (i bit D) e r bit di *rilevamento errore*, o di **ridondanza** (i bit R). Minuscolo è il numero di bit, maiuscolo è il campo di bit vero e proprio. Sia i bit D che i bit R vengono trasmessi attraverso il link, che ricordiamo è *suscettibile ad errori*.

Per ricavare i bit R si applica una certa funzione $H()$ (sostanzialmente di *hashing*) sui bit D . Assunto che mittente e destinatario usino la stessa funzione $H()$, una volta ricevuti i dati il destinatario potrà applicare $H()$ sui bit D ricevuti e ottenere la sua copia dei bit R da confrontare con quelli ricevuti.

Se i bit R corrispondono non ci sono stati errori (a meno della sfortunata circostanza in cui sia i bit D che R sono stati alterati per risultare erroneamente corretti, si sceglie la funzione $H()$ in modo che questo sia difficilmente vero), altrimenti qualcosa è andato storto nella trasmissione sul link.

I bit R , nel contesto del *framing* visto in 9.5.1, vengono inseriti nel *trailer* del frame.

10.1.1 Algoritmi di rilevamento errori

Vediamo alcune possibili funzioni $H()$ usate per fare rilevamento errori.

- **Parity checking**: posto che si voglia trasferire una parola da d bit: si sceglie un solo bit R di ridondanza, determinato come segue:
 - Se il numero di bit a 1 fra i bit D è pari, si imposta il bit in R a 1;

- Se il numero di bit a 1 fra i bit D è dispari, si imposta il bit in R a 0.

In questo caso il bit in R viene detto **bit di parità** (*parity bit*). Questo algoritmo viene detto di tipo **even parity**: l'algoritmo opposto si direbbe **odd parity**.

Questo algoritmo funziona bene solo nella circostanza in cui il numero di errori sui bit d è dispari (per quanto ci riguarda, 1). Quando si trasferiscono quantità relativamente basse di bit per parola su canali abbastanza affidabili (8, 16, 24 bit su rame o simili) abbiamo che è abbastanza sicuro. Con parole più grandi chiaramente può essere molto più soggetto ad errori.

- **Checksum:** questo sistema, detto della "somma di controllo" viene usato in UDP e TCP. Non si riferisce principalmente a frame, quindi parleremo di blocchi di dati generici.

In questo caso si prende ogni blocco di dati e si divide in frammenti da n bit. Sommando questi frammenti si ottiene un campo R su $r = n$ bit (quindi modulo 2^n) detto *checksum*, che può essere ricalcolato lato destinatario per fare rilevamento degli errori.

Questo metodo è più sensibile rispetto al bit di parità, e viene usato perlopiù a livello di *trasporto* UDP o TCP (più che a livello *data link*, dove si fanno altri tipi di controlli meno sensibili).

- **CRC (Cyclic Redundancy Check):** questo è il metodo usato più spesso in ambito di *Datalink*.

Supponiamo di avere R bit dati, e un *pattern* detto **generatore** G di $r+1$ bit (dove ricordiamo che R sono i bit di ridondanza e r il loro numero). Mittente e destinatario dovranno entrambi conoscere il generatore.

L'idea è quindi che in fase di trasmissione si prende un numero R di r bit tale che la concatenazione $D|R$ dei bit D ed R sia divisibile per G . Notiamo che le somme in CRC si fanno *modulo 2*, quindi senza riporti. In questo caso la relazione appena descritta sarà:

$$D \cdot 2^r \oplus R = nG$$

dove $D \cdot 2^r$ è semplicemente lo shift a sinistra di r bit di D , \oplus lo XOR (che implementa la somma modulo 2), e n una costante moltiplicativa naturale qualsiasi, che assicura che il lasto sinistro è divisibile per G .

Quando il destinatario riceve il frame la verifica è molto semplice: ha D ed R ottenuti sul link fisico, e noto G può effettuare la divisione. Se il resto è diverso da 0, allora si rileva un errore di trasmissione.

Resta quindi da vedere come R viene calcolato nella pratica. Vogliamo:

$$D \times 2^r \oplus R = nG \implies D \times 2^r = nG \oplus R$$

come ci è concesso dalle proprietà dello XOR, per cui R dovrà soddisfare:

$$R = \text{mod}(D \times 2^r, G)$$

dove *mod* indica il resto della divisione fra i due argomenti (cioè l'operatore modulo).

10.2 Correzione errori

Dopo aver discusso il *rilevamento degli errori*, vediamo come procedere nella **correzione degli errori** in caso se ne rilevino.

La soluzione più semplice sarebbe quella di ritrasmettere i frame sbagliati. Se poi si avesse un algoritmo di rilevamento errori che restituisse esattamente *quali* bit sono errati, basterebbe commutarli per risolvere gli errori.

In questo caso non serve più un codice a *rilevamento* degli errori, ma un codice a **correzione** degli errori. Per ogni frame trasmettiamo d bit di dati (i bit D) e r bit di *correzione errore*, o ancora di **ridondanza** (i bit EDC , da *Error Detection Code*).

Questi sono simmetrici al codice definito per il rilevamento errori: la differenza è che dati D ed EDC , lato destinatario possiamo rilevare esattamente l'errore di trasmissione (se c'è stato).

10.2.1 Algoritmi di correzione errori

Vediamo alcuni (1) modi per calcolare i bit EDC ed effettuare quindi la correzione degli errori.

- **Parity checking bidimensionale:** si sistemanano i bit D in una struttura matriciale, e si calcolano i bit di parità per ogni riga e per ogni colonna.

Ad esempio, si può dire:

$$D = \{0, 1, 0, \dots\} = \{d_1, d_2, d_3, \dots, d_d\} \Rightarrow D_m = \begin{pmatrix} d_{1,1} & d_{1,2} & \dots & d_{1,j} \\ d_{2,1} & d_{2,2} & \dots & d_{2,j} \\ \dots & \dots & \dots & \dots \\ d_{i,1} & d_{i,2} & \dots & d_{i,j} \end{pmatrix}$$

Posti i e j tali che $i \cdot j = d$ (numero di bit in D). A questo punto si orla D_m con i bit di parità di *riga* (parity_{row}) e i bit di parità di *colonna* (parity_{col}):

$$\begin{pmatrix} D_m & \text{parity}_{row} \\ \text{parity}_{col} & \end{pmatrix} = \left(\begin{array}{cccc|c} d_{1,1} & d_{1,2} & \dots & d_{1,j} & d_{1,j+1} \\ d_{2,1} & d_{2,2} & \dots & d_{2,j} & d_{2,j+1} \\ \dots & \dots & \dots & \dots & \dots \\ d_{i,1} & d_{i,2} & \dots & d_{i,j} & d_{i,j+1} \\ \hline d_{i+1,1} & d_{i+1,2} & \dots & d_{i+1,j} & \end{array} \right)$$

dove

$$\text{parity}_{row} : \quad d_{r,j+1} = \text{parity} \left(\sum_{c=1}^j d_{r,c} \right)$$

$$\text{parity}_{col} : \quad d_{j+1,c} = \text{parity} \left(\sum_{r=1}^i d_{r,c} \right)$$

A questo punto un errore su un singolo $d_{r,c}$ verrà rilevato in quanto incrocerà una riga e una colonna (l'intersezione andrà commutata). Due o più errori su righe e colonne disgiunte veranno similmente rilevati, a meno di casi di errori multipli su più righe e colonne (che potrebbero addirittura dare falsi positivi). Infine, un numero pari di errori sulla stessa riga o sulla stessa colonna ci permettono di rilevare, ma non correggere errori (vedremo 2 colonne sbagliate e una riga giusta, o viceversa, senza poter quindi incrociare).

Chiaramente i bit EDC nel trailer saranno molti di più: per la precisione $i + j$.

10.3 Trasferimento dati

Abbiamo visto alcuni algoritmi di *rilevamento errori*, introdotto l'ipotesi di effettuare un *reinvio* dei dati corrotti nel caso di errori, e visto anche un algoritmo di *rilevamento e correzione errori*.

Vediamo adesso come realizzare un livello superiore a quello data link, riportando indietro l'ipotesi del reinvio dati, cioè il cosiddetto livello *transfer* o **trasferimento**, che considereremo un livello *affidabile*.

Le considerazioni che facciamo adesso saranno quelle che nel modello OSI vengono implementate nel cosiddetto livello **transport**.

L'idea è quella di usare i servizi offerti dal livello data link per realizzare un'ulteriore astrazione, quella appunto di *trasferimento affidabile*.

Quello che farà la componentistica di livello transport sarà interagire con un *livello superiore* che fornirà *dati*: questi verranno incapsulati in un **pacchetto**, a sua volta incluso come *payload* di un frame di livello Datalink, e quindi spedito sulla linea inaffidabile vista finora.

10.3.1 Primitive di trasferimento

Per implementare questo tipo di livello *transfer* ci doteremo quindi di alcune primitive a servizio di un'altro *livello superiore*:

- `rdt_send()`: chiamata dal livello superiore (nella macchina *trasmettitore*), implementa l'invio sul canale affidabile (cioè implementa un protocollo **RDT** (*Reliable Data Transfer*));
- `udt_send()`: implementa l'invio sul canale inaffidabile fino al ricevitore;
- `rdt_recv()`: implementa il ricevimento sul canale affidabile lato ricevitore, cioè ottiene i dati sul canale inaffidabile, e li corregge (se necessario);
- `deliver_data()`: si occupa di inoltrare i dati ottenuti attraverso il protocollo RDT al livello superiore (nella macchina *ricevitore*).

Notiamo che le primitive `udt_send()` e `rdt_recv()` dovranno implementare un qualche tipo di comunicazione bidirezionale (ad esempio se la `rdt_recv()` vuole chiedere il reinvio di un frame perso).

10.3.2 Macchine a stati finiti

Per descrivere il protocollo **RDT** useremo il formalismo della *macchina a stati finiti* (**FSM**, *Finite State Machine*).

Una macchina a stati finiti rappresenta un *automa* dotato di **stati** e **transizioni** fra tali stati: dato uno stato ed un evento si può determinare la transizione successiva, e quindi come si evolve il protocollo.

Nelle prossime sezioni definiremo *iterativamente* versioni sempre più accurate di RDT rispetto a un qualche protocollo reale.

10.4 RDT 1.0

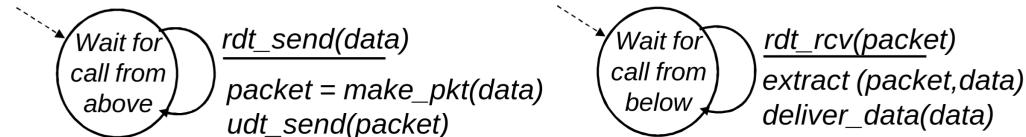
Iniziamo ad iterare sul nostro protocollo *RDT*. Assumiamo come prime ipotesi, largamente semplificate:

- Non ci sono errori di bit;
- Non ci sono perdite di frame.

Prevederemo quindi FSM diverse per *trasmettitore* e *ricevitore*:

- Il trasmettitore invia dati su un canale sottostante;
- Il ricevitore legge i dati dal canale sottostante.

L'FSM in questo caso sarà semplice:



- Il trasmettitore avrà il compito di aspettare la chiamata dall'alto, e una volta arrivata di creare un *pacchetto* da spedire sulla linea inaffidabile (incapsulandolo in un opportuno frame);
- Il ricevitore avrà il compito di aspettare la chiamata dal basso, e una volta ricevuta ricevere il pacchetto spedito dal trasmettitore. Una volta arrivato, dovrà estrarre i dati dal pacchetto e consegnarli al livello superiore.

10.5 RDT 2.0

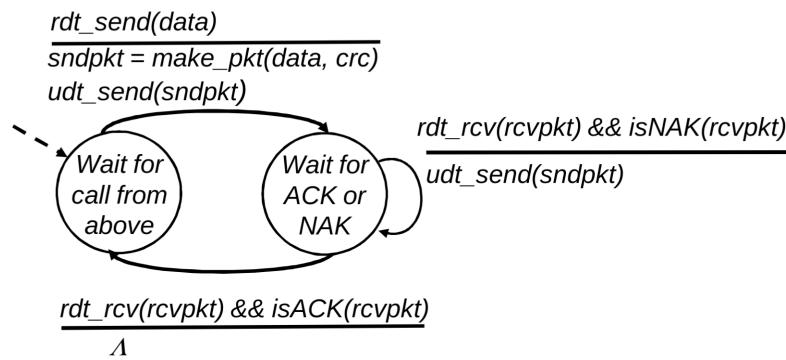
Introduciamo un mezzo di trasmissione inaffidabile che potrebbe avere errori di bit, e presumiamo che tale mezzo implementi un qualche livello sottostante (di tipo data link) che implementa correzione degli errori attraverso CRC o checksum (come visto in 10.1 e 10.2).

Il problema sarà: come riprendersi dagli errori? Abbiamo due modi principali:

- *Acknowledgment (ACK)*, significa che il ricevitore ha "capito", cioè ha ricevuto il frame correttamente;
- *Negative acknowledgment (NAK)*, significa che il ricevitore non ha capito, cioè non ha ricevuto il frame correttamente.

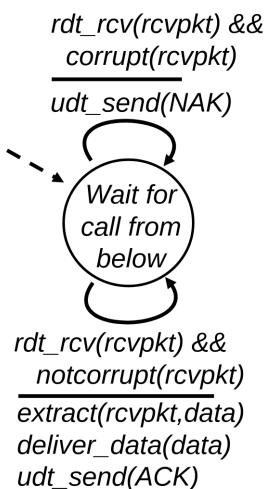
Quando il trasmettitore riceve un **ACK**, sa di poter procedere con il frame successivo, mentre quando riceve un **NAK** sa che deve reinviare il frame corrente.

In questo caso l'FSM del trasmettitore ha il seguente aspetto:



- Il trasmettitore avrà il compito, come prima, di aspettare la chiamata dall'alto e quindi inviare il pacchetto sulla linea inaffidabile;
- A questo punto dovrà aspettare per un ACK o un NAK dal ricevitore:
 - Se riceve un NAK, deve reinviare lo stesso frame;
 - Altrimenti, cioè se riceve un ACK, deve tornare ad aspettare la chiamata dall'alto per il pacchetto successivo.

Lato ricevitore l'FSM sarà invece:



- Il ricevitore dovrà restare in attesa della chiamata dal basso, e rispondere quindi ai pacchetti ricevuti su linea inaffidabile dal trasmettitore;
- Valutata la correzione degli errori sul pacchetto ricevuto, dovrà:
 - Inviare il NAK nel caso il pacchetto sia corrotto;
 - Inviare l'ACK e inoltrare il pacchetto al livello superiore nel caso il pacchetto sia stato ricevuto correttamente.

11 Lezione del 15-10-25

11.0.1 RDT 2.1

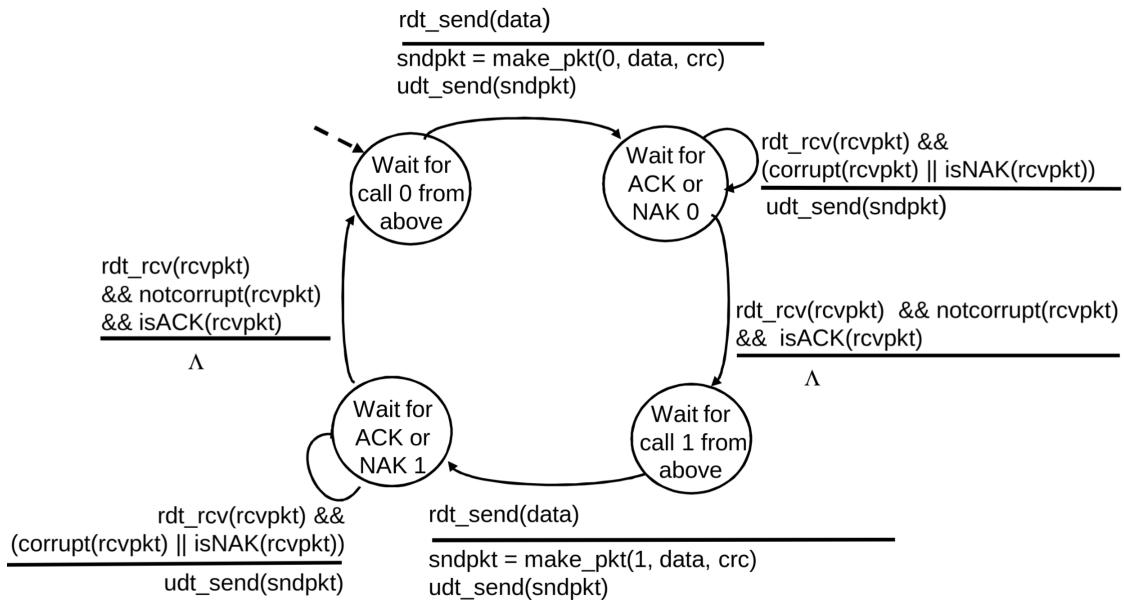
Avevamo introdotto il protocollo di trasferimento fittizio RDT 2.0. Questo era basato su segnali ACK/NAK mandati sul livello datalink dal destinatario al mittente.

Un problema che non avevamo considerato è che questi segnali possono venire corrotti, proprio come i dati veri e propri che volevamo trasmettere. Se il mittente riceve un ACK/NAK corrotto, non può semplicemente ritrasmettere: il destinatario potrebbe ricevere frame duplicati.

Per risolvere questo problema introduciamo fra le *informazioni di controllo* dei frammi inviati un cosiddetto **numero di sequenza**. Il mittente allega ad ogni frame un numero di sequenza, e il destinatario implementa un *contatore* del numero di sequenza. I frame con numero di sequenza già visto vengono ignorati.

Abbiamo che per un protocollo *stop-and-wait*, un contatore a 1 bit è più che abbastanza: si inviano due pacchetti alla volta.

In questo caso la macchina a stati del trasmettitore sarà la seguente:

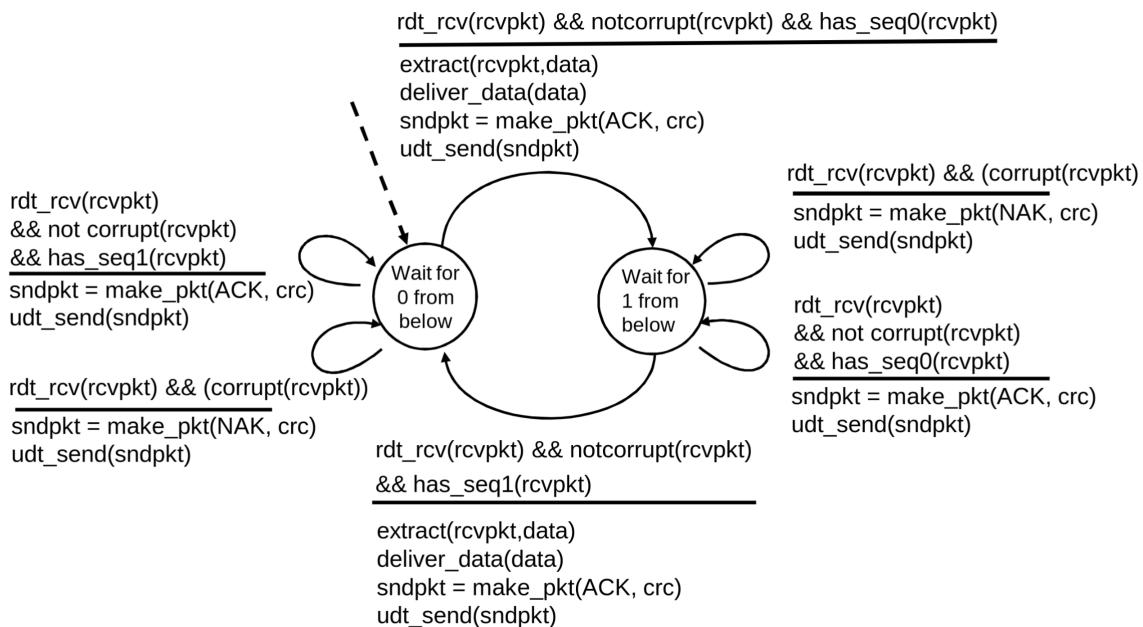


Quello che facciamo è:

- Aspettare la chiamata dall'alto ed inviare il primo pacchetto (con numero di sequenza 0);
- Continuare ad inviare il pacchetto finché non si riceve un ACK non corrotto;
- Aspettare la chiamata dall'alto ed inviare il secondo pacchetto (con numero di sequenza 1);
- Di nuovo, continuare ad inviare il pacchetto finché non si riceve un ACK non corrotto.

Questo andamento si ripete in maniera ciclica.

Il ricevitore potrà a questo punto eseguire il seguente automa:



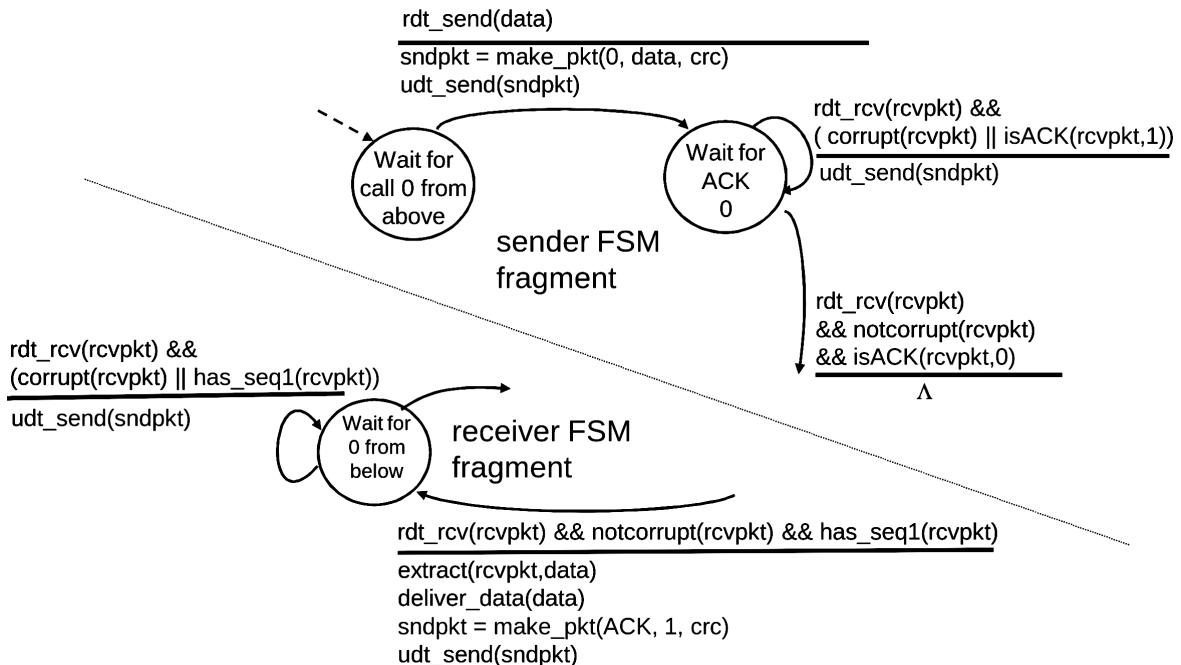
In questo caso il flusso è:

- Si aspetta per il primo pacchetto (dopo la chiamata dal basso), si invia un pacchetto ACK nel caso questo arrivi integro e con numero di sequenza 0, NAK altrimenti;
- Si aspetta il secondo pacchetto, comportandoci in maniera analoga, ma verificando che il numero di sequenza sia 1.

11.0.2 RDT 2.2

Si può semplificare l'approccio sopra riportato guardando semplicemente agli ACK e ai numeri di sequenza: in questo caso il ricevitore deve solo inviare ACK per gli ultimi pacchetti ottenuti integri. ACK per pacchetti già trasmessi verrano interpretati dal trasmettitore come avevamo interpretato i NAK fino ad ora.

In questo caso le macchine a stati si modificano come segue:



11.1 RDT 3.0

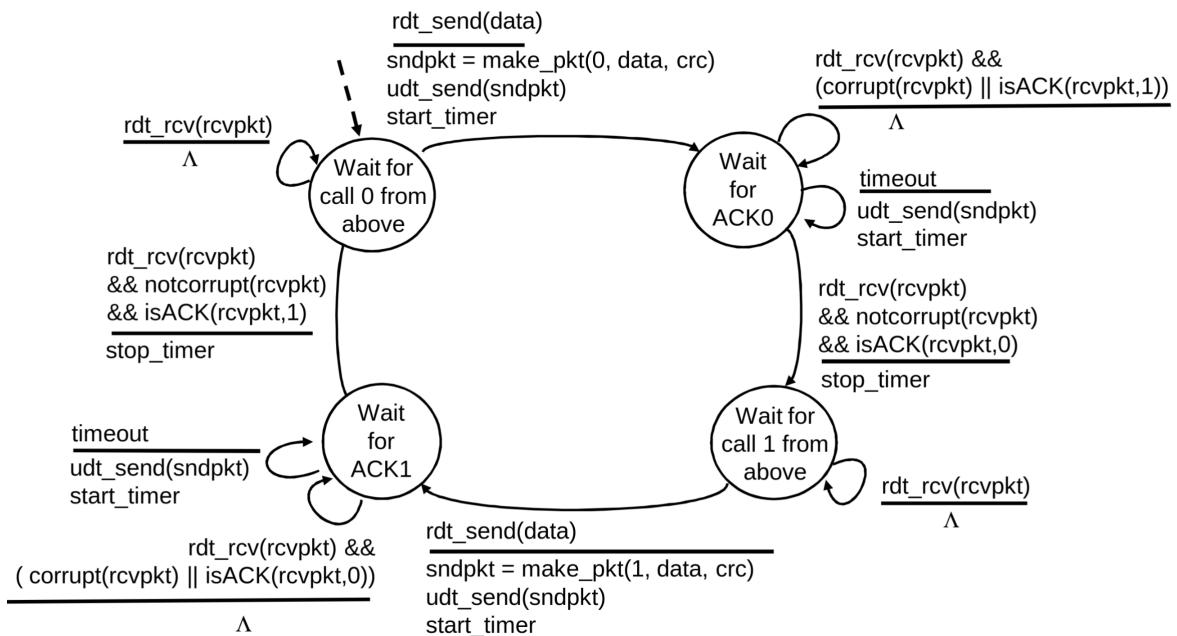
Facciamo una nuova assunzione sui canali: diciamo che il canale sottostante può anche *perdere* (sia data che ACK), mentre prima poteva solo avere errori.

In questo caso potremo sfruttare checksum, numeri di sequenza, ACK e ritrasmissioni, ma non basterà completamente a risolvere il problema.

Un primo approccio potrebbe essere, lato trammettitore, di aspettare un tempo ragionevole per l'arrivo dell'ACK da parte del ricevitore. *"Ragionevole"* può avere molti significati: assumiamo che il tempo sia tarato sul round-trip-time del link corrente. Dopo che questo tempo passa, quindi, si procede col reinvio del pacchetto:

- Se il pacchetto non era arrivato, il problema è risolto;
- Se il pacchetto era già arrivato e si era perso l'ACK, il ricevitore riceverà pacchetti duplicati: i numeri di sequenza risolvono già questo problema. Chiaramente, il ricevitore dovrà specificare il numero di sequenza del pacchetto per cui sta facendo ACK.

La macchina a stati del trasmettitore con questo approccio è:



Ciò che facciamo è analogo a RDT 2.1, con la differenza che:

- Quando inviamo il pacchetto, facciamo partire un timer;
- Se il timer fa timeout durante la fase di attesa per l'ACK, si reinvia il pacchetto al numero di sequenza corrente e si fa ripartire il timer.

Notiamo come si adotta un'approccio *lazy* ai pacchetti ACK corrotti o per pacchetti con numero di sequenza sbagliato (quello che interpretavamo come NAK): in questo caso si ignorano e ci si affida al timeout (che prima o poi arriverà) per effettuare il reinvio.

Lato ricevitore, nulla cambia rispetto a RDT 2.2.

11.1.1 Prestazioni di RDT 3.0

Facciamo alcune note sulle **prestazioni** che riusciamo ad ottenere.

Sia U_{sender} l'**utilizzazione trasmittitore**, cioè la frazione di tempo sul RTT usata dal trasmittitore in fase di invio effettivo del pacchetto.

Se assumiamo di avere un link da 1 Gbps, 15 ms di ritardo di propagazione e di dover trasmettere un pacchetto da 8000 bit. Il tempo per trasmettere un pacchetto sul canale sarà allora circa:

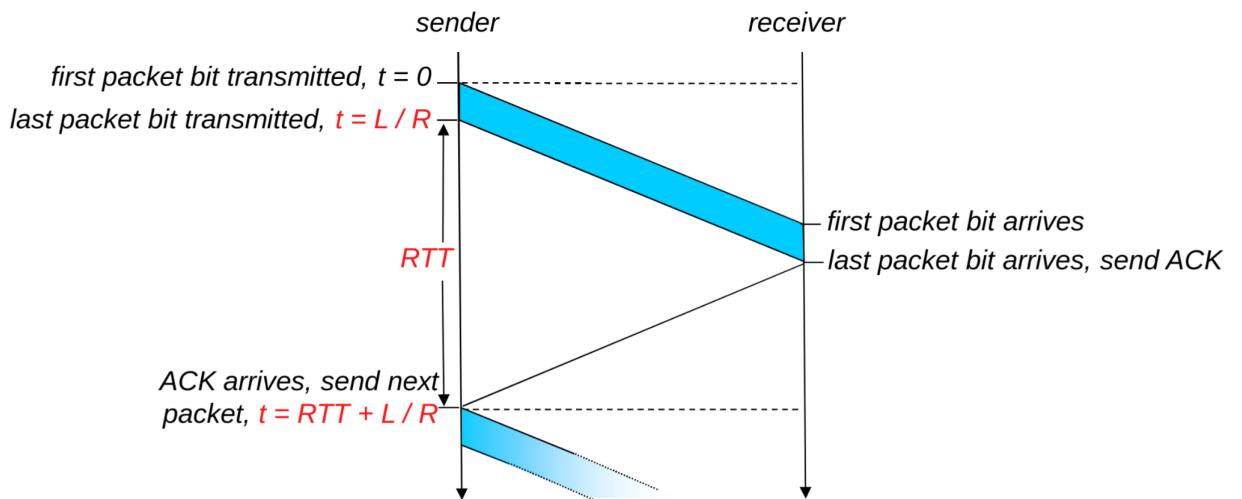
$$D_{\text{trans}} = \frac{L}{R} = \frac{8000}{1000} = 8 \text{ ms}$$

In condizioni ideali, quindi, avremo che dovremo inviare un pacchetto e aspettare un ACK (spedito in tempo trascurabile): questi sono due viaggi sul link (quindi un RTT dato dal doppio del ritardo di propagazione) e il tempo di trasmissione del pacchetto vero e proprio. Si ha quindi:

$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{0.0008}{30 + 0.0008} = 0.00027$$

Cioè l'utilizzazione è terribile: del tempo che impieghiamo a trasmettere un pacchetto, meno dell'0.001% è effettivamente usato per trasmettere i dati che ci interessano! Il problema è chiaramente che passiamo molto tempo ad *aspettare*, quando potremmo avvantaggiarci con l'invio dei pacchetti successivi.

Un diagramma che descrive la situazione incontrata è il seguente:



Dove vediamo che molta della banda disponibile (effettivamente il tempo) viene sprecata ad aspettare che i pacchetti vengano consegnati, e gli ACK ritornino, quando il tempo di trasmissione di pacchetto è invece piuttosto contenuto.

11.2 Pipelining

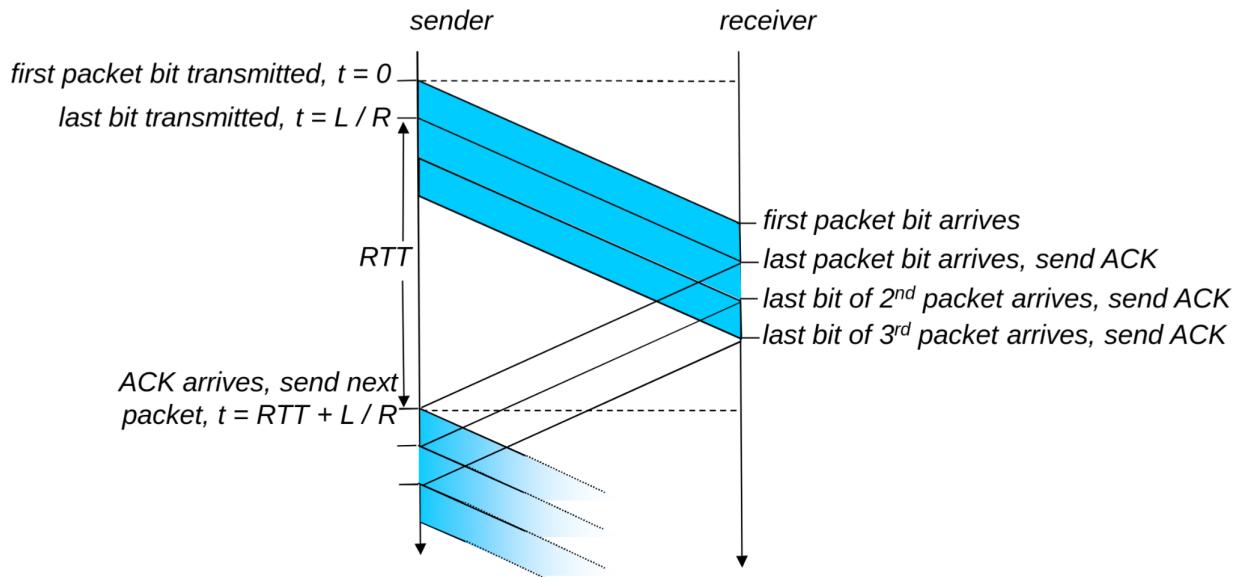
Il **pipelining** è un meccanismo attraverso il quale il trasmettitore può inviare più pacchetti in massa, senza aver ancora ricevuto l'ACK. Lo poniamo come alternativa allo *stop-and-wait* visto finora.

Riprendendo i conti della scorsa sezione, prendiamo ad esempio di inviare 3 pacchetti invece di 1 per ciclo di ACK. In questo caso si avrà:

$$U_{\text{sender}} = \frac{3L/R}{RTT + L/R} = \frac{0.0024}{30 + 0.0008} = 0.00081$$

che non è molto ma è già di più.

Vediamo cosa accade rispetto al grafico della scorsa sezione:



Come si nota dal diagramma, l'obiettivo è quello di riempire la finestra di tempo che passavamo ad aspettare l'arrivo del pacchetto e del seguente ACK, inviando altri pacchetti.

Chiaramente questo tipo di approccio comporta delle complicazioni lato ricevitore: dovremmo prevedere la bufferizzazione dei pacchetti in entrata, e ingrandire il contatore del numero di sequenza (2 valori non basteranno più).

Trascurando quanto avviene lato ricevitore, una domanda interessante è *quanti pacchetti possiamo anticipare prima dell'ACK successivo*: dovremmo riempire la finestra del RTT con più trasmissioni (ciascuna occupante tempo $\frac{L}{R}$), cioè inviare un numero di pacchetti al massimo pari a:

$$N = \frac{RTT}{L/R}$$

Approcci di questo tipo vengono detti a **sliding window** ("finestra scorrevole"), in quanto prendono buffer scorrevoli sul blocco di pacchetti da inviare.

11.2.1 Recupero errori in pipelining

Abbiamo visto che, se vogliamo implementare un protocollo in pipelining, dobbiamo adoperare alcune soluzioni tecniche:

- Buffering al trasmettitore dei pacchetti da inviare;
- Buffering al ricevitore dei pacchetti da ricevere, senza nessuna assicurazione che questi vengano ricevuti in ordine (serve il numero di sequenza);
- Un contatore per il numero di sequenza più grande al ricevitore.

Le specifiche di queste soluzioni vengono definite dal protocollo per il recupero dagli errori che adottiamo. Ne vederemo i 2 principali, che sono **go-back-N** e **selective-repeat**.

11.2.2 Go-back-N

In questo caso il trasmettitore mette fino a N pacchetti senza ACK in pipeline. Al ricevitore viene permesso solo di inviare ACK cumulativi: non effettua ACK se perde uno degli N pacchetti. Il trasmettitore mantiene quindi un timeout per gli ultimi N pacchetti inviati: se il timer scade reinvia tutti gli N pacchetti.

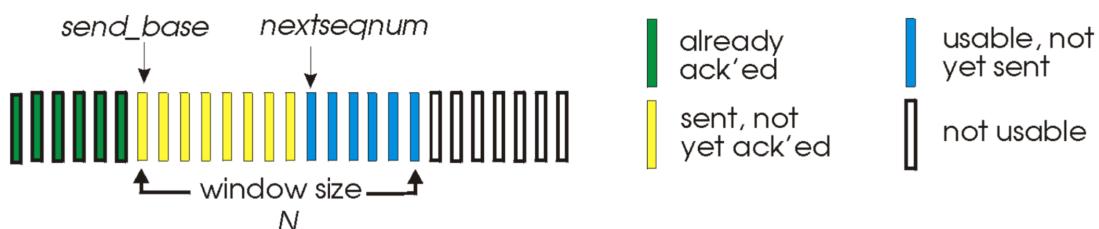
Dal punto di vista delle FSM si ha quindi che il trasmettitore spedisce fino a N pacchetti consecutivi sulla pipeline. Il ricevitore invia da parte sua ACK consecutivi al pacchetto con numero di sequenza più alto ricevuto fino a quel momento (se bufferizzare o meno i pacchetti che non combaciano col numero di sequenza aspettato è una scelta implementativa).

Quando il primo pacchetto della sequenza spedita viene ricevuto, il trasmettitore sposta in avanti la sua finestra e spedisce il successivo, spostandosi così lungo il blocco di pacchetti.

Se un pacchetto viene perso, il trasmettitore aspetta senza spostare la finestra: prima o poi scatterà il timeout del pacchetto perso e verrà reinviato. Lato ricevitore, possiamo aspettarci che questo passerà lo stesso intervallo temporale a inviare ACK sempre sull'ultimo pacchetto valido (con numero di sequenza consecutivo) ricevuto, e che il trasmettitore abbia ignorato tale ACK in quanto ascolta solo l'ACK più grande ricevuto.

Vediamo quindi come si implementa a livello di buffer l'approccio go-back-N. Vorremo mantenere nel trasmettitore due puntatori, uno a `send_base`, cioè il primo pacchetto non ancora soggetto ad ACK, e uno a `nextseqnum`, cioè il prossimo pacchetto da inviare (contenuti nella finestra) non ancora ricevuto dal livello superiore. Per fissare quanto detto prima, `send_base` avanza con gli ACK cumulativi del ricevitore, e `nextseqnum` avanza con i dati ottenuti dal livello superiore.

A questo punto potremo individuare le regioni:



- Il range da 0 a `send_base` (non compreso) sarà formato da pacchetti inviati e soggetti ad ACK, quindi ignorati;
- Il range da `send_base` a `nextseqnum` sarà formato da pacchetti inviati e per cui ancora non si è ricevuto ACK;
- Il range da `nextseqnum` a `send_base + N` (dimensione finestra) sarà formato da pacchetti non ancora inviati ma usabili (presenti nella finestra);
- Infine, il range da `send_base + N` in poi non è ancora considerato in quanto fuori dalla finestra.

11.2.3 Selective-repeat

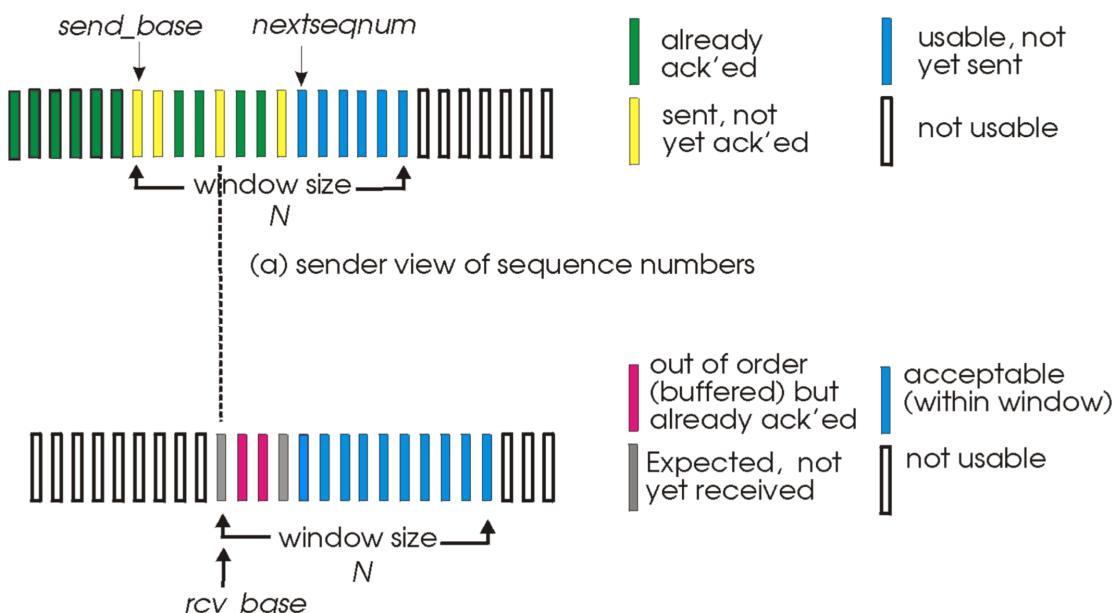
In questo caso il trasmettitore invia N pacchetti senza ACK in pipeline (come sopra). La differenza è che il ricevitore invia ACK per ogni pacchetto: il trasmettitore deve quindi

mantenere un timer per ogni pacchetto inviato, e nel caso questo timer faccia timeout inviare solo il pacchetto corrispondente.

Questo approccio è chiaramente più efficiente: si ritrasmettono solo i pacchetti persi, a costo di un'implementazione più costosa sia lato trasmittitore (dobbiamo mantenere più timer) che ricevitore (dobbiamo inviare ACK separati per ogni pacchetto).

Il caso in cui approcci come il go-back-N vengono comunque usati è quello di dispositivi *constraining* (come quelli che si usano nell'IoT): questi hanno prestazioni meno soddisfacenti e quindi impongono di usare protocolli meno efficienti.

Vediamo quindi anche il selective-repeat a livello di buffer e finestre.



Come notiamo dal grafico, lato trasmittitore le cose restano sostanzialmente uguali, se non per il fatto che il range da `send_base` a `nextseqnum` (che in go-back-N) avevamo considerato come formato da pacchetti inviati ma senza ACK) può adesso contenere sia pacchetti senza ACK che con ACK. Questo viene naturalmente dal fatto che accettiamo gli ACK come relativi a un singolo pacchetto, e non cumulativi fino a quel pacchetto.

Vediamo poi come anche il ricevitore deve fornire la sua finestra. In particolare, questa parte `rcv_base`, cioè il primo pacchetto che ci aspettiamo di ricevere ma non è ancora arrivato. Tutti i pacchetti nel range da `rcv_base` a `rcv_base + N` sono, per il ricevitore, pronti per essere ricevuti. Quando si ricevono, si fa l'ACK e si marchiano come ricevuti. Se è il pacchetto a `rcv_base` ad essere ricevuto, spostiamo avanti la finestra fino al prossimo pacchetto non ricevuto.

12 Lezione del 17-10-25

12.0.1 Pseudocodice del selective repeat

Continuiamo la discussione del selective repeat. Per farlo, non usiamo più un approccio FSM, ma descriviamolo in *pseudocodice*.

Vediamo allora il trasmittitore:

Algoritmo 1 Trasmettitore *selective repeat*

```

if ci sono dati dall'alto then
    Se il prossimo numero di sequenza è nella finestra, invia il pacchetto
end if
if timeout( $n$ ) then
    Reinvia il pacchetto  $n$  e riavvia il timer
end if
if ACK( $n$ ) nella finestra then
    Marchia il pacchetto  $n$  come ricevuto
    Se era il pacchetto non ricevuto con numero di sequenza minore, avanza la finestra
        fino al prossimo pacchetto non ricevuto
end if

```

e quindi il ricevitore:

Algoritmo 2 Ricevitore *selective repeat*

```

if ricevuto pacchetto  $n$  in finestra then
    invia ACK( $n$ )
    se è fuori ordine, bufferizzalo
    se è in ordine, consegna il pacchetto (e quindi i successivi bufferizzati), e avanza la
        finestra al prossimo pacchetto non ancora ricevuto
end if
if ricevuto pacchetto  $n$  prima della finestra then
    invia ACK( $n$ )
end if
if ricevuto pacchetto  $n$  dopo la finestra then
    ignoralo
end if

```

Notiamo quindi che sia il trasmettitore che il ricevitore mantengono le loro finestre, il primo per i pacchetti marchiati come da inviare, il secondo per i pacchetti che può bufferizzare.

12.1 Protocollo PPP

Vediamo quindi un protocollo reale di livello *datalink* che implementa le funzionalità che abbiamo descritto finora. Questo è il protocollo **PPP** (*Point-to-Point Protocol*), e viene oggi usato dai router per parlare direttamente l'uno con l'altro.

Le funzionalità del PPP sono:

- **Framing** pacchetti: provvede all'incapsulamento dei datagrammi livello network in frame livello datalink. Può portare dati livello network di qualsiasi protocollo di rete (non solo IP);
- **Trasparenza bit**: è capace di portare qualsiasi pattern di bit nel campo dati;
- **Rilevamento errori**: c'è ma non c'è correzione;
- **Verifica connessione**: permette di verificare e segnalare il fallimento del link al livello network;

- **Negoziamento indirizzi** al livello network: gli host possono imparare e configurare gli indirizzi di rete altrui.

Vediamo che non è presente quanto abbiamo detto sul *trasferimento affidabile* di dati: questo lo rende effettivamente un link **inaffidabile** (abbiamo visto che questa funzionalità viene reintrodotta a livello *transport*).

In particolare, PPP non è provvisto di:

- Correzione errori;
- Recupero da errori;
- Controllo di flusso;
- Consegnna fuori ordine;
- Comunicazioni *point-multipoint*.

Ci si aspetta invece che queste funzionalità vengano delegate a protocolli di livello superiore come *TCP*.

12.1.1 Pacchetto PPP

Iniziamo a vedere la struttura di un pacchetto PPP:

1 byte	1 byte	1 byte	1 byte	variabile	2 o 4 byte	1 byte
2 <flag>	<address>	<control>	<protocol>	<info>	<check>	<flag>

- I campi **flag** fanno da delimitatori per il framing;
- Il campo **address** è effettivamente inutile: il PPP non permette comunicazioni *point-multipoint* (più destinatari) e quindi è sempre impostato tutto a 1 (l'indirizzo di *broadcast*);
- Il campo **control** è ugualmente inutile;
- Il campo **protocol** stabilisce il protocollo di livello superiore da usare (ad esempio *TCP*);
- Il campo **info** contiene i dati veri e propri, ed è a dimensione variabile da 0 a 1500 byte (la dimensione massima può essere negoziata);
- Il campo **check** supporta il CRC per il rilevamento errori.

Notiamo che garantire la *trasparenza bit* non è immediato: se si spedisce un campo **info** che contiene una copia esatta del marcitore di flag terminale che ci aspettiamo, come dovrebbe fare il ricevitore a capire che tale campo va spedito e non è effettivamente il terminatore?

Risolviamo il problema sfruttando il **byte stuffing/unstuffing**, meccanismo per noi sostanzialmente analogo a quello delle *sequenze di escape*. Si antepone quindi al byte problematico un'altro specifico byte, di escape, che il ricevitore potrà poi interpretare come segnalatore (di prendere "alla lettera" il byte successivo), e quindi buttare via.

12.2 Link multipli

Veniamo ora a descrivere le reti formate da più **link di accesso**. La prima distinzione che dobbiamo fare è fra i due tipi di link che possiamo incontrare:

- Link **punto-punto**: sono del tipo che abbiamo visto finora, e collegano solo due dispositivi fra di loro;
- Link **broadcast** (detti anche a *mezzo condiviso*): sono tipici di tecnologie come il *vecchio Ethernet*, le reti mobili e reti wireless come 802.11 (*WiFi*). In questo tipo di rete i dispositivi possono parlare con tutti gli altri dispositivi, contemporaneamente.

Il problema delle reti broadcast è chiaramente l'**interferenza**: si possono verificare *collisioni* se un nodo riceve 2 o più segnali contemporaneamente.

12.2.1 Protocolli MAC

Dobbiamo quindi definire un protocollo di accesso multiplo, cioè un **algoritmo distribuito** che determini come i nodi condividono il canale (decida quale nodo può trasmettere). Il problema è che tale protocollo deve sfruttare comunicazione sul canale condiviso stesso, in quanto non abbiamo sempre a disposizione un altro canale *fuori banda* da usare per la coordinazione delle comunicazioni.

Quello che desideriamo è, dato un canale ad accesso multiplo **MAC** (*Multiple Access Channel*) con capacità di R bit al secondo, che:

1. Quando un nodo vuole trasmettere, può farlo a capacità R ;
2. Quando M nodi vogliono trasmettere, questi possono farlo a capacità R/M ;
3. Il sistema sia completamente decentralizzato: non ci siano nodi speciali che coordinano le comunicazioni, né clock di sincronizzazione;
4. Il sistema sia *semplice*.

Chiameremo i protocolli che ci permettono di fare ciò protocolli **MAC** (acronimo già visto).

12.2.2 Tassonomia dei MAC

Esistono tre classi principali di protocolli MAC:

1. **Partizionamento canali**: si divide il canale in "pezzi" (slot temporali, bande di frequenza), e si alloca ogni "pezzo" all'uso esclusivo di un nodo. Un idea interessante è quella della sovrapposizione di **codice**: in questo caso ogni nodo parla usando una codifica particolare, e si riesce a distinguere fra nodi rilevante tali codifiche;
2. **Accesso casuale**: il canale non viene diviso e si permettono le collisioni. In questo caso chiaramente l'approccio alla trasmissione sarà di:
 - Rilevare quando il canale è libero;
 - Aspettare un quanto temporale (sperabilmente piccolo e casuale, in modo da ridurre le collisioni);
 - Iniziare a trasmettere.

In questo caso chiaramente bisogna dotarsi di un sistema per *recuperare* le collisioni, che inevitabilmente prima o poi accadranno;

3. **Turni:** i nodi si dividono a turni, ma i nodi con più dati da inviare possono prendere turni più lunghi.

12.3 MAC a partizionamento

Iniziamo ad elaborare la tassonomia appena vista descrivendo protocolli MAC che appartengono ad ogni categoria.

12.3.1 Protocollo TDMA

Vediamo il primo protocollo MAC, il **TDMA** (*Time Division Multiple Access*). Questo fa parte della categoria (1) dei protocolli MAC, e quindi è a *partizionamento canali*.

Si fornisce l'accesso al canale di accesso in *round*. Ogni stazione ottiene uno slot di accesso a lunghezza fissa in ogni round. Gli slot non utilizzati tengono il canale a riposo.

Con questo protocollo riusciamo a soddisfare il requisito (2), in quanto il mezzo è condiviso ugualmente fra tutti i nodi. La decentralizzazione (requisito 3) non è immediata, in quanto bisogna capire quando i round iniziano e l'ordine di comunicazione ad ogni round. Immaginiamo che i nodi riescano comunque a coordinarsi automaticamente. Sicuramente il protocollo è però semplice (requisito 4).

13 Lezione del 21-10-25

Riprendiamo la discussione dei protocolli MAC.

13.0.1 Protocollo FDMA

Avevamo parlato del protocollo TDMA. Adesso vediamone la versione in partizionamento di stretto: l'**FDMA** (*Frequency Division Multiple Access*).

In questo caso si divide il canale in bande di frequenze. Le bande risultano così ugualmente distribuite fra tutti i nodi, e allo stesso modo il bitrate è diviso fra le bande.

IN questo il FDMA ha gli stessi pro e contro del TDMA: il requisito (2) è soddisfatto direttamente. La centralizzazione (3) è sempre un problema in quanto qualcuno dovrà almeno definire quali sono le bande. Infine, il protocollo è nuovamente semplice (4) (ammesso che uno conosca la trasformata di Fourier discreta).

13.1 MAC ad accesso casuale

Veniamo quindi a discutere i protocolli ad **accesso casuale**. In questo caso ipotizziamo che:

- Quando un nodo ha un pacchetto, vuole trasmetterlo al bitrate massimo R ;
- Non c'è alcun tipo di coordinazione *a priori* dei nodi.

In questo caso, chiaramente si incorre nel problema delle *collisioni*.

I protocolli **MAC random access** specificano come:

- Rilevare collisioni;

- Recuperare dalle collisioni (ad esempio ritrasmettendo con ritardi casuali).
- Esempi di protcoli MAC random access sono **ALOHA** (originariamente acronimo *Additive Links On-line Hawaii Area*) e **CSMA** (*Carrier Sense Multiple Access*).

13.1.1 Protocollo ALOHA slotted

Il protocollo **ALOHA** (*Additive Links On-Line Hawaii Area*) nasce da un progetto di Norman Abramson, per coordinare le comunicazioni fra le sedi dell'università delle Hawaii.

Vediamo la versione di ALOHA detta **ALOHA slotted** (tempo diviso in *slot*). Assumiamo che:

- Tutti i frame hanno la solita dimensione prefissata;
- Il tempo è diviso in slot di dimensioni uguali (corrispondenti al tempo necessario a trasmettere 1 frame);
- I nodi iniziano a trasmettere solo all'inizio degli slot;
- I nodi sono sincronizzati;
- Se 2 o più nodi iniziano a trasmettere all'inizio di uno slot, tutti gli slot rilevano una collisione.

La modalità di operazione dei nodi in ALOHA è quindi la seguente:

- Quando un nodo ottiene un frame da trasmettere, lo trasmette al prossimo slot;
- Se non c'è collisione, la trasmissione va a buon fine e il frame viene trasmesso;
- Se c'è collisione, il nodo ritrasmette il frame in ogni slot seguente con un certa probabilità p finché la trasmissione non riesce.

Le collisioni vengono verificate controllando se si riceve un ACK al termine della trasmissione: notiamo che questo rende identici (dal punto di vista del trasmettitore) gli errori di trasmissione di ACK e le collisioni. I trasmettitori in ALOHA assumono che ci sia comunque bisogno di ritrasmettere.

I pro dell'approccio sono che ogni nodo può trasmettere al bitrate R completo (1), il sistema è altamente decentralizzato (3) (assunta sincronizzazione), ed è semplice (4).

I contro sono che le collisioni riducono il bitrate effettivo, sprecando slot e quindi impedendo che M nodi possano trasmettere a capacità R/M (3). Inoltre, si possono verificare slot fermi. Infine, notiamo che abbiamo bisogno di sincronizzazione di clock o comunque una sincronia che permetta ai nodi di capire quando gli slot iniziano.

13.1.2 Efficienza di ALOHA slotted

Facciamo alcune considerazioni quantitative sull'efficienza del protocollo appena visto. Assumiamo che N nodi vogliono trasmettere frame negli slot con probabilità p .

La probabilità che un nodo trasmetta con successo in uno slot sarà:

$$p_{\text{nodo}} = p(1 - p)^{N-1}$$

per cui la probabilità che un *qualsiasi* nodo trasmetta con successo in uno slot sarà:

$$p_{\text{nodi}} = Np(1 - p)^{N-1}$$

Vogliamo quindi trovare p^* che massimizza p_{nodi} . Derivando p_{nodi} su p si ha:

$$\frac{d}{dp} p_{\text{nodi}} = N(1-p)^{N-1} - pN(N-1)(1-p)^{N-2} = N(1-p)^{N-2}((1-p) - p(N-1))$$

che imponiamo uguale a zero:

$$\frac{d}{dp} p_{\text{nodi}} = N(1-p)^{N-2}((1-p) - p(N-1)) = 0$$

Ora, $(1-p)^{N-2}$ è sempre diverso da 0, salvo il caso $p = 1$ (che però sarebbe troppo facile, significherebbe che tutti i nodi ritrasmettono sempre, e le collisioni sarebbero quindi assicurate). Inoltre, a meno che la nostra rete non sia banale, $N \neq 0$. Poniamo quindi $p \neq 1 \implies (1-p)^{N-2} \neq 0, N \neq 0$ e cancelliamo i rispettivi termini:

$$(1-p) - p(N-1) = 1 - p - pN + p = 1 - pN = 0 \Rightarrow pN = 1 \Rightarrow p = \frac{1}{N}$$

Per cui $p^* = \frac{1}{N}$ è la probabilità ottima dati N nodi.

Sostituiamo quindi p^* ottima in p_{nodi} per trovare la formula della probabilità ottima che non ci siano collisioni ad ogni turno:

$$p_{\text{nodi}} = N \frac{1}{N} \left(1 - \frac{1}{N}\right)^{N-1} = \left(1 - \frac{1}{N}\right)^{N-1}$$

Prendiamo quindi il limite per vedere qual'è l'efficienza ottima a $N \rightarrow +\infty$:

$$\lim_{N \rightarrow +\infty} p_{\text{nodi}} = \lim_{N \rightarrow +\infty} \left(1 - \frac{1}{N}\right)^{N-1} = \frac{1}{e}$$

L'occhio vispo del matematico si renderà subito conto che questo non è altro che la versione negativa del limite di Nepero (si rimanda ai testi di analisi per le dovute dimostrazioni), per cui l'utilizzazione complessiva nel limite di infiniti dispositivi (per quanto ci riguarda il caso peggiore) sarà $\frac{1}{e} \approx 37\%$, che è molto poco!

13.1.3 Protocollo ALOHA pure

Eliminiamo l'ipotesi della sincronizzazione dal protocollo ALOHA in versione slotted: in questo caso, appena arriva un frame, il nodo inizia subito a trasmettere. Nel caso di collisioni, finisce la sua trasmissione (o aspetta il tempo frame) e ritrasmette con probabilità p . Essenzialmente, abbiamo il comportamento di ALOHA slotted ma con tempi di inizio casuali. Chiamiamo questo protocollo **ALOHA pure**.

Questa soluzione è più facilmente implementabile nella realtà, in quanto è irrealistico pensare che più dispositivi (magari su reti *wireless*) possano sincronizzarsi efficientemente e in maniera affidabile al tempo frame.

La probabilità delle collisioni in questo caso incrementa: ogni frame inviato al tempo t_0 collide con i frame inviati fra $[t_0-1, t_0+1]$ (dove l'unità corrisponde allo slot temporale dedicato ad un frame, cioè lavoriamo sempre con periodo tempo di frame).

13.1.4 Efficienza di ALOHA pure

Ripetiamo le considerazioni matematiche fatte in 13.1.2 su ALOHA slotted, stavolta riferendoci ad ALOHA pure. La differenza principale dal punto di vista matematico sarà

che i frame possono iniziare a venire trasmessi in qualsiasi momenti, e non in *round* temporali discreti.

Abbiamo quindi accennato al fatto che ogni frame inviato al tempo t_0 collide con i frame inviati fra $[t_0 - 1, t_0 + 1]$, considerata come unità di tempo il tempo frame (i frame saranno comunque di dimensione prefissata, per cui il tempo frame sarà costante).

Gli slot temporali che dobbiamo considerare per le collisioni sono quindi 2:

$$[t_0 - 1, t_0) \cup (t_0, t_0 + 1]$$

Ognuno di questi slot temporali equivale al tempo frame, e come abbiamo detto in ogni slot di tempo frame i nodi provano a trasmettere con probabilità $(1 - p)$. Il risultato è che la probabilità complessiva che un nodo trovi il mezzo libero (non incorra in collisioni) è $(1 - p)^{2(N-1)}$.

Si ha quindi che la probabilità che un nodo trasmetta con successo in uno slot sarà:

$$p_{\text{nodo}} = p(1 - p)^{2(N-1)}$$

per cui la probabilità che un *qualsiasi* nodo trasmetta con successo in uno slot sarà:

$$p_{\text{nodi}} = Np(1 - p)^{2(N-1)}$$

Vogliamo quindi trovare p^* che massimizza p_{nodi} . Derivando p_{nodi} su p si ha:

$$\begin{aligned} \frac{d}{dp} p_{\text{nodi}} &= N(1 - p)^{2(N-1)} - 2N(N - 1)(1 - p)^{2(N-1)-1} \\ &= N(1 - p)^{2N-2} - 2Np(N - 1)(1 - p)^{2N-3} = N(1 - p)^{2N-3} ((1 - p) - 2p(N - 1)) \end{aligned}$$

che imponiamo uguale a zero:

$$\frac{d}{dp} p_{\text{nodi}} = N(1 - p)^{2N-3} ((1 - p) - 2p(N - 1)) = 0$$

Su $(1 - p)^{2N-3}$ e N valgono le stesse considerazioni già fatte. Poniamo quindi $p \neq 1 \implies (1 - p)^{2N-3} \neq 0, N \neq 0$ e cancelliamo i rispettivi termini:

$$(1 - p) - 2p(N - 1) = 1 - p - 2pN + 2p = 1 + p - 2pN = 0 \Rightarrow p(1 - 2N) = -1 \Rightarrow p = \frac{1}{2N - 1}$$

Per cui $p^* = \frac{1}{2N-1}$ è la probabilità ottima dati N nodi.

Sostituiamo quindi p^* ottima in p_{nodi} per trovare la formula della probabilità ottima che non ci siano collisioni ad ogni turno:

$$p_{\text{nodi}} = N \frac{1}{2N - 1} \left(1 - \frac{1}{2N - 1}\right)^{2(N-1)} = \frac{N}{2N - 1} \left(1 - \frac{1}{2N - 1}\right)^{2N-2}$$

Prendiamo quindi il limite per vedere qual'è l'efficienza ottima a $N \rightarrow +\infty$:

$$\lim_{N \rightarrow +\infty} p_{\text{nodi}} = \lim_{N \rightarrow +\infty} \frac{N}{2N - 1} \left(1 - \frac{1}{2N - 1}\right)^{2N-2}$$

Questo limite è leggermente più complicato di quello visto prima. Dividiamo in due termini:

$$\lim_{N \rightarrow +\infty} p_{\text{nodi}} = \lim_{N \rightarrow +\infty} \frac{N}{2N - 1} \cdot \lim_{N \rightarrow +\infty} \left(1 - \frac{1}{2N - 1}\right)^{2N-2}$$

Questo si può fare finché entrambi i termini ammettono limite finito. Dimostreremo che questo è il caso.

- Prendiamo il primo termine:

$$\lim_{N \rightarrow +\infty} \frac{N}{2N - 1} = \frac{1}{2}$$

e questo è banale;

- Prendiamo il secondo limite:

$$\lim_{N \rightarrow +\infty} \left(1 - \frac{1}{2N - 1}\right)^{2N-2} = \lim_{N \rightarrow +\infty} \left(1 - \frac{1}{2N - 1}\right)^{2N-1} \cdot \left(1 - \frac{1}{2N - 1}\right)^{-1}$$

Possiamo nuovamente dividere:

$$= \lim_{N \rightarrow +\infty} \left(1 - \frac{1}{2N - 1}\right)^{2N-1} \cdot \lim_{N \rightarrow +\infty} \left(1 - \frac{1}{2N - 1}\right)^{-1}$$

- Di questi, il primo limite è sempre la versione negativa del limite di Nepero, fatta la sostituzione $x = 2N - 1$:

$$\lim_{N \rightarrow +\infty} \left(1 - \frac{1}{2N - 1}\right)^{2N-1} = \lim_{x \rightarrow +\infty} \left(1 - \frac{1}{x}\right)^x = \frac{1}{e}$$

- Il secondo è invece banale:

$$\lim_{N \rightarrow +\infty} \left(1 - \frac{1}{2N - 1}\right)^{-1} = 1$$

Si ha quindi che il secondo limite è complessivamente:

$$\frac{1}{e} \cdot 1 = \frac{1}{e}$$

Il limite è quindi:

$$\lim_{N \rightarrow +\infty} p_{\text{nodi}} = \frac{1}{2} \cdot \frac{1}{e} = \frac{1}{2e}$$

Abbiamo quindi che l'utilizzazione complessiva nel limite di infiniti dispositivi sarà $\frac{1}{2e} \approx 18\%$: tanti calcoli per ottenere una brutta notizia!

Abbiamo quindi trovato un protocollo che è molto più semplice e decentralizzato (non richiede sincronizzazione), ma ha un utilizzazione del canale di comunicazione che è di molto minore della versione slotted. Potremmo voler trovare un approccio migliore.

13.1.5 Protocollo CSMA

Il protocollo **CSMA** (*Carrier Sense Multiple Access*) prevede di *ascoltare* prima di trasmettere: in questo caso si può rilevare la condizione del canale condiviso prima di provare ad accedervi. In particolare, dopo aver ascoltato:

- Se il canale è rilevato fermo, si trasmette l'intero frame;
- Se il canale è rilevato attivo, si differisce la trasmissione ad un secondo momento.

Questo è in qualche modo analogo al modo in cui gli umani usano mezzi condivisi (ad esempio l'etere quando parlano a voce): prima si ascolta cosa dicono gli altri, e poi si parla. La regola fondamentale è "*non interrompere gli altri!*".

CSMA ha una versione detta **CSMA/CD**, cioè *CSMA with Collision Detection*. In questo caso le collisioni vengono rilevate entro qualche tempo limitato. Le trasmissioni in collisione vengono abortite, riducendo lo spreco del canale. Questo è piuttosto facile per i mezzi cablati, più difficile per i mezzi wireless.

Il CSMA/CD è necessario in quanto le collisioni possono comunque accadere dopo l'ascolto del canale (*carrier sensing*): i tempi di propagazione significano infatti che due nodi potrebbero non sentire le trasmissioni appena iniziate l'uno dell'altro.

13.1.6 Rilevamento collisioni CSMA

Per rilevare le collisioni si potrebbe pensare di mettere a comparatore il segnale che il nodo sta trasmettendo e quello che sta ricevendo: se il delta è maggiore di qualche soglia, dev'essere che c'è un'altra sorgente di segnale e quindi siamo in collisione.

In verità l'approccio effettivamente usato è più semplice: invece di parlare di *segnali*, si parla di *potenze* rilevate sul mezzo di trasmissione. L'approccio è comunque funzionale: se il trasmettitore si aspetta di poter erogare una potenza P , rilevando potenze $P^* >> P$ sul mezzo potremo concludere con un certo grado di sicurezza di essere in collisione.

13.1.7 CSMA su Ethernet

Vediamo quindi l'implementazione di CSMA sul mezzo Ethernet. Vediamo cosa fa la **NIC** (*Network Interface Card*) quando vuole trasmettere un frame.

1. La NIC riceve il datagramma dal livello network, e ne crea un frame (siamo a livello datalink);
2. La NIC ascolta il mezzo (*channel sensing*):
 - Se il mezzo è rilevato fermo, inizia con la trasmissione del frame;
 - Se il mezzo è rilevato attivo, si aspetta finché non è fermo, e quindi si trasmette.
3. Se la NIC riesce a trasmettere l'intero frame senza collisioni, abbiamo finito;
4. Se si verificano altre trasmissioni mentre si trasmette, cioè una *collisione* (vedi sezione sopra), si abortisce la trasmissione e invia il cosiddetto segnale di **jam**: questo è un segnale a potenza più alta della media che ha lo scopo di avvisare gli altri trasmettitori che una collisione si è verificata;
5. Dopo aver abortito, la NIC entra in un **backoff esponenziale**:
 - Dopo la m -esima collisione, sceglie un K casuale fra $\{0, 1, 2, \dots, 2^m - 1\}$ (binario). Quindi il NIC aspetta per $K \times 512$ tempi bit (per *tempo bit* intendiamo il tempo necessario a trasmettere un bit con bitrate R di mezzo), e quindi torna al passo (2);
 - Il risultato è che più collisioni si hanno, più lungo è in media l'intervalllo di backoff (attesa).

Chiaramente, per implementare tale politica dovremo dotarci di un contatore per le m collisioni rilevate. Inoltre, sarà utile prevedere una dimensione massima per la finestra dove scegliere K , cioè un numero di collisioni oltre cui gli intervalli di backoff non continuano ad aumentare (in Ethernet questo numero è 17, per cui il tempo massimo è 2^{17} tempi bit).

Vediamo cosa abbiamo ottenuto con CSMA su Ethernet: il requisito di completa decentralizzazione è soddisfatto (3), il sistema è effettivamente abbastanza semplice (4), e le frequenze di trasmissione sono perlopiù soddisfatte, salvo collisioni da recuperare (requisiti (1) e (2)). Il problema rimasto è quello degli alti carichi: il backoff esponenziale implica che il sistema può rallentare fino a throughput nulli se si verificano collisioni particolarmente gravi (quindi inevitabilmente quando ci sono molti nodi).

13.2 MAC a turni

Veniamo quindi ai protocolli MAC basati sui turni.

13.2.1 Polling

Come primo esempio vediamo il meccanismo del **polling**. Prevediamo un nodo, detto *master*, che invita gli altri nodi (detti *slave*) a trasmettere a turno.

Questo approccio è usato spesso per dispositivi "stupidi" (è usato ad esempio in Bluetooth).

Per quanto riguarda i nostri requisiti, non è assolutamente decentralizzato (3), è abbastanza efficiente (requisiti (1) e (2)) ed è sempre abbastanza semplice (4).

Si comporta bene agli alti carichi (il nodo centralizzato governa gli altri assicurando throughput massimo) ma non ai bassi (si sprecano molti turni), proprio come TDMA (sezione 12.3.1).

I problemi sono poi l'*overhead* dato dal polling, la *latenza* introdotta e il fatto che il master rappresenta un *single point of failure*. Quest'ultimo problema potrebbe essere risolto prevedendo un algoritmo di *rielezione* da mettere in esecuzione al momento della morte del master. Questo, però, va chiaramente in contro al requisito (4) (semplicità).

13.2.2 Passaggio di token

Un altro modo per implementare comunicazioni a turni è il **token passing**. Questo token (di controllo) rappresenta un qualche segnalatore che chi vuole trasmettere deve possedere per poterlo fare. Il token viene passato da un nodo all'altro, sequenzialmente, in modo che tutti i nodi possano parlare.

I problemi principali saranno quindi la *latenza*, la *gestione* del token stesso, nonché il fatto che questo rappresenta nuovamente un *single point of failure* per l'intero sistema (se il token va perso, chi può parlare?).

La latenza si ha dal fatto che il token deve essere passato, e questo rappresenta un *overhead* (il tempo passato a passarsi il token non è passato a trasmettere).

14 Lezione del 22-10-25

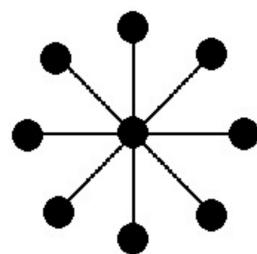
14.1 LAN

Le reti **LAN** (*Local Area Network*) sono reti ad accesso *locale*, quindi di estensione più o meno limitata ma comunque nell'ordine di abitazioni, edifici o al limite insiemi di edifici. La velocità in bitrate si aggira fra le centinaia di Mbps e le decine di Gbps.

Sono reti che sfruttano mezzi di tipo *broadcast*, e quindi richiedono sia un protocollo MAC per l'accesso al mezzo, che un meccanismo di **indirizzamento MAC**.

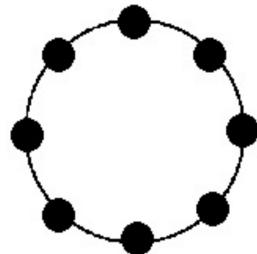
Il mezzo trasmissivo condiviso può assumere diverse topologie, cioè vi possono essere diverse *topologie di rete*. Ad esempio, possiamo nominare:

- Topologia a **stella**, dove un nodo centrale detto *hub* connette più nodi periferici:

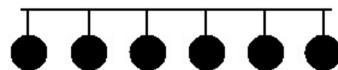


Viene detta anche *bus in the box*, in quanto l'hub rappresenta effettivamente una sorta di "bus" avvolto su sé stesso. Dal punto di vista elettronico l'hub non sarà altro che un amplificatore operazionale che amplifica il segnale e lo inoltra agli altri nodi;

- Topologia a **ring**, dove ogni nodo è collegato a 2 nodi adiacenti in una struttura circolare:



- Topologia a **bus**, dove tutti i nodi condividono un'unica linea di comunicazione condivisa:



Gli hub, dal punto di vista elettronico, dovranno essere "*smorzati*" con una resistenza che ne ancorà il valore quando questi non portano segnale. Questa viene detta *terminatore*.

14.1.1 Indirizzamento MAC

Gli indirizzi **MAC** (detti anche indirizzi *fisici*, *LAN* o *Ethernet*) vengono usati per inviare frame fra interfacce localmente e fisicamente connesse fra di loro (cioé stanti su rete LAN). L'indirizzo MAC è su 48 bit, solitamente incluso nel firmware della NIC, a volte configurabile da software. I primi 24 bit dei codici MAC sono solitamente assegnati (dalla IEEE) all'azienda produttrice della NIC. Le aziende acquistano infatti porzioni dello spazio di indirizzamento MAC, per garantire l'univocità. I bit successivi sono quindi univoci per i singoli prodotti (magari hanno bit superiori dedicati a codificare categorie di prodotti, ecc...).

Ai MAC associamo chiaramente gli indirizzi **IP** su 32 bit che abbiamo già visto più volte. Ricordiamo che questi vengono usati a livello 3, cioè livello network, per effettuare il cosiddetto *packet forwarding*. Noi adesso stiamo discutendo il livello 1, cioè livello link.

Una buona analogia per il rapporto fra MAC e IP è questa:

- L'indirizzo **MAC** è il *codice fiscale*;
- L'indirizzo **IP** è l'*indirizzo di casa*.

Chiaramente, il MAC è unico per tutti, mentre l'IP può cambiare. Può essere che lo stesso dispositivo apparga, con lo stesso MAC, a più indirizzi IP nel tempo.

14.2 Ethernet

Ethernet è una famiglia di tecnologie, dominanti nell'ambito delle reti locali (LAN) cablate.

14.2.1 Old fashioned Ethernet

Ethernet è stata la prima tecnologia LAN ampiamente usata. Semplice ed economica, nelle sue prime versioni (*old fashioned Ethernet*) sfruttava come mezzo un cavo coassiale usato come bus da più dispositivi. I dispositivi si allacciavano al bus attraverso i cosiddetti *tap* (spesso perforazioni dirette del cavo coassiale), e il bus veniva terminato con una resistenza da 50 Ohm. La velocità di trasmissione era di 10 Mbps.

14.2.2 Ethernet oggi

Oggi Ethernet non sfrutta più bus su cavi coassiali, ma dispositivi di hub (detti *switch*). La differenza precisa fra hub e switch verrà discussa in seguito.

I nodi si collegano all'hub attraverso il classico doppino telefonico (per Ethernet con connettore RJ48).

Ethernet è un protocollo di tipo:

- **Senza connessione**: non c'è nessuna forma di *handshaking* fra le NIC trasmettitore e ricevitore;
- **Inaffidabile**: le NIC ricevitore non trasmettono ACK o NAK alle NIC trasmettitore. I dati nei frame persi vengono recuperati solo se il mittente usa un protocollo di trasferimento dati affidabile (ad esempio TCP);
- Il protocollo **MAC** di Ethernet è *CSMA/CD unslotted* con *backoff binario*.

14.2.3 Frame Ethernet

Un **frame Ethernet** incapsula un datagramma IP (o un qualche altro pacchetto di protocollo network) in una struttura del tipo:

```

1 8 byte      6 byte   6 byte   6 byte   variabile 4 byte
2 <preamble>  <dest>   <src>    <type>   <data>     <crc>

```

- `preamble` viene usato per avere sincronia fra ricevitore e trasmettitore: consiste in 7 byte di 10101010 seguiti da un byte di 10101011.

Può essere utile una breve discussione di come si trasmettono le sequenze di bit su Ethernet: anziché trasmettere i bit come 1 o 0 fisici, si trasmettono come transizioni da 1 a 0 e viceversa. Ciò si ottiene facilmente modulando i dati in XOR con il clock del trasmettitore.

Questo permette al ricevitore di rilevare transizioni anziché valori statici, e permette quindi nella fase (piuttosto lunga) di trasmissione del preamble, la sincronizzazione dei clock.

- Gli indirizzi `dest` e `src` (rispettivamente *destinazione* e *sorgente*) sono su 6 byte, e sono di tipo MAC. Quando l'adattatore Ethernet riceve un frame con indirizzo MAC che combacia col suo (oppure un indirizzo broadcast, ad esempio per un pacchetto ARP, che vedremo fra poco), inoltra i dati incapsulati nel frame (campo `data`) al livello network. In caso contrario ignora il frame;
- `data` è il campo dati, cioè trasporta il *payload* (il datagramma IP o simile). Ha dimensione variabile dai 46 ai 1500 byte.
 - La dimensione massima $L_{\max} = 1500$ bit è fissata per evitare la degradazione del segnale. Ethernet prevedeva infatti una distanza massima di 500m di cavo (200m per cavi più sottili), e l'introduzione di **repeater** (*ripetitori*) per raggiungere distanze maggiori (fino a 4 in cascata, per un totale di 2.5 Km di copertura).
 - La dimensione minima $L_{\min} = 42$ bit è legata al rilevamento di collisioni su vecchio Ethernet coassiale.

Se poniamo come τ il tempo di propagazione da host a ripetitore, o da ripetitore a ripetitore, abbiamo che su una linea di distanza massima (4 ripetitori) il tempo di rilevamento di collisione è:

$$t_{\text{coll}} = 2\tau + \Delta r = \frac{2l_{\max}}{v} + \Delta r \leq \frac{L_{\min}}{R}$$

dove $l_{\max} = 500$ m è la distanza del link, e v la velocità di propagazione sul mezzo (per il rame circa 200.000 Km al secondo). Δr è invece un ritardo dovuto al repeater attraversato. L_{\min} sarà quindi la lunghezza minima del frame in modo che le collisioni possano essere rilevate in tempo utile (prima della completa trasmissione del frame). Abbiamo già messo a diseguagliaza il tempo di trasmissione L_{\min}/R , quindi ricaviamoci L_{\min} :

$$L_{\min} \geq \left(\frac{2l_{\max}}{v} + \Delta r \right) R$$

Il valore di 48 byte si ricava esattamente in questo modo. (Notiamo che finora si è considerato L_{\max} e L_{\min} come lunghezza del payload, in verità si tiene chiaramente conto dell'overhead dato dai campi di controllo).

- `type` indica il protocollo di livello network (solitamente IP);
- `crc` è un campo di ridondanza che permette il rilevamento di errori al ricevitore. I frame corrotti vengono persi.

Chiaramente, la privacy dei frame non è assicurata: ci potremmo fidare che tutte le interfacce NIC ignorano i frame che non le spettano, ma è possibile configurarle nella cosiddetta **modalità promiscua** per costringerle a rilevare tutti i frame, qualsiasi sia il destinatario.

L'**indirizzo di broadcast** è un indirizzo particolare a cui tutti i nodi rispondono (cioè da cui tutti i nodi ricevono frame), e viene usato, appunto, per fare *broadcast* di frame destinati alla totalità dei dispositivi in rete.

14.2.4 CSMA/CD su Ethernet

Vediamo come Ethernet implementa il protocollo CSMA/CD per permettere la condivisione del mezzo (MAC).

1. Quando la NIC riceve dati dal livello network, crea un frame;
2. Se la NIC rileva il mezzo fermo per 96 unità di tempo bit (con bitrate di 10 Mbps, 9.6 ms), e quindi trasmette;
3. Se la NIC riesce a trasmettere l'intero frame senza collisioni, il suo lavoro è finito;
4. Se la NIC rileva una collisione, abortisce ed invia un segnale di jam da 48 bit;
5. Dopo aver abortito, la NIC entra in *backoff esponenziale* come già visto in 13.1.7: quindi sceglie un K casuale fra $0, 1, 2, \dots, 2^m - 1$ alla m -esima iterazione, aspetta $K \cdot 512$ tempi bit, e torna al passo 2. Dopo 17 tentativi il frame è perso.

14.2.5 Livelli datalink e physical in Ethernet

Ethernet è definito da una vasta gamma di *standard* (100BASE-TX, 100BASE-T2, 100BASE-FX, 100BASE-T4, 100BASE-SX, 100BASE-BX), che hanno in comune il protocollo MAC e il formato dei frame. Le velocità invece differiscono ampiamente: 2, 10, 100 Mbps, oppure 1, 10, 40 Gbps. In particolare, i numeri nei nominativi degli standard indicano la velocità in Mbps o Gbps mentre le lettere in fondo rappresentano il tipo di mezzo fisico (T significa *twin pair* (doppino), F significa *fiber* (fibra), ecc...).

Questi standard definiscono informazioni utili al livello **datalink** (o semplicemente **link**) e **physical**. Abbiamo infatti già introdotto come il vecchio Ethernet sfruttava, al livello fisico, bus su cavi coassiali, mentre oggi sfrutta collegamenti punto-punto (via doppino telefonico) dai nostri dispositivi ad un unico switch Ethernet centralizzato.

15 Lezione del 24-10-25

15.1 Reti a commutazione di pacchetto

Iniziamo ad entrare nel dettaglio delle reti a commutazione di pacchetto, e quindi ad anticipare i concetti che saranno alla base del cosiddetto livello **network**. Per fare ciò introduciamo nuovi tipi di dispositivo, col compito di smistare pacchetti, fra cui **switch** e **router** (per adesso parliamo principalmente dei primi).

15.1.1 Hub Ethernet

Riprendiamo come esempio lo standard Ethernet.

Avevamo introdotto l'idea di **hub** nel moderno Ethernet dove tutti i dispositivi sulla rete si collegano in modalità punto-punto. Questo ci permetteva di implementare la cosiddetta topologia a *stella*.

Questi hub possono essere intesi come *ripetitori "stupidi"* a livello fisico, che si limitano a prendere i bit in entrata dal dispositivo ed inoltrarli a tutti gli altri dispositivi, allo stesso bitrate.

Questo significa che tutti i nodi connessi all'hub possono collidere con gli altri, non si ha nessun tipo di bufferizzazione dei frame inviati, e non si implementa nessun tipo di CSMA/CD a livello hub: il compito di rilevare le collisioni è assegnato alle NIC dei dispositivi.

15.2 Switch Ethernet

Gli hub Ethernet sono stati oggi soppiantati da dispositivi più intelligenti: gli *switch*.

Gli switch sono dispositivi di livello datalink, e hanno quindi un ruolo *attivo* nella rete. Possono infatti:

1. Immagazzinare i frame Ethernet ricevuti dai dispositivi;
2. Esaminare l'indirizzo MAC di destinazione del frame, e selettivamente inoltrarlo a uno o più link in uscita;
3. Implementare protocolli di rilevamento collisioni come CSMA/CD.

Questo però non nega che lo switch deve essere comunque **trasparente** agli host sulla rete: questi non devono dover essere al corrente della sua esistenza.

Un'altra caratteristica degli switch è che sono *plug-and-play* e non richiedono configurazione.

Il compito dello switch è quindi quello di **commutare** le linee di ingresso con le linee di uscite: questo significa che gli host riescono ad avere una connessione dedicata e diretta con lo switch, e attraverso questo l'illusione di una connessione dedicata e diretta con l'host con cui stanno parlando.

15.2.1 Switch table

Per realizzare la funzionalità di instradamento degli switch si sfrutta una tabella, detta **switch table**. Ogni entrata della switch table contiene:

- L'indirizzo MAC dell'host;
- L'interfaccia (semplicemente la linea di uscita) su cui si trova tale host;
- Un *time stamp* che rappresenta il tempo di vita dell'interfaccia.

Notiamo che propriamente dovremmo parlare di *nodi*, e non di *host*, in quanto i primi sono a livello datalink e i secondi sono a livello network. Ci fidiamo di capirci.

La popolazione della switch table avviene attraverso l'**apprendimento automatico** dello switch: questo infatti *impara* man mano che si inviano frame Ethernet quali host possono essere raggiunti attraverso quali interfacce.

Vediamo come si tramette il primo frame quando la switch table è vuota:

1. Quando il frame arriva allo switch, questo può associare l'interfaccia del mittente al suo indirizzo MAC (contenuto nell'header del frame);
2. Per inoltrare il frame, controlla la sua switch table (che è attualmente vuota): non può quindi fare meglio che inoltrarlo a tutti (in quanto non sa a quale interfaccia corrisponde il MAC destinatario). Chiamiamo questo processo *flooding*;
3. Questo processo si ripete: man mano che gli host inviano frame, lo switch riesce a farsi un'idea della struttura della rete popolando la switch table. Questa è una pretesa ragionevole in quanto immaginiamo che gli host che ricevono pacchetti prima o poi ne invieranno di altri, rendendosi disponibili all'interno della switch table.

In pseudocodice, il comportamento è simile al seguente:

```

1 on <frame_received>:
2   record(<link>, <sending_MAC>);
3   entry = switch_table.index(<receiving_MAC>);
4   if(entry) {
5     forward(entry.link);
6   } else {
7     forward(all); // flood
8   }

```

15.2.2 Gerarchie di switch

Gli switch Ethernet possono essere interconnessi fra di loro: uno switch che collega più host può a sua volta connettersi (assieme ad altri switch) ad un singolo switch principale, e questo processo può ripetersi ricorsivamente.

Ad esempio, in un palazzo potremmo immaginare che ogni piano ha il suo switch (che collega i dispositivi di tale piano), e questi switch di piano vengono connessi da un singolo switch per l'intero palazzo.

15.2.3 Differenze fra switch e router

Possiamo quindi anticipare le differenze fra switch e ruoter: sappiamo che gli switch sono dispositivi di livello 2 (datalink), che guardano agli header di frame (e quindi indirizzi MAC), mentre i router sono dispositivi di livello 3 (network), che guardano agli header di pacchetto (e quindi indirizzi IP).

Rifacendoci alla pila protocollare,abbiamo che gli switch implementano i livello 1 e 2 (fisico e link), mentre i router implementano i livelli 1, 2 e 3 (fisico, link e network).

Sia switch che router usano delle tabelle per governare l'instradamento dei messaggi ottenuti (frame per gli switch e pacchetti per i router), che negli switch sono dette *switch table* e nei router *routing table*.

15.3 Datacenter

I grandi fornitori di servizi su Internet sfruttano strutture dette **datacenter** per organizzare le loro macchine server. I datacenter rappresentano quindi l'altro "nodo", assieme a quello utente, dove si concentra la potenza di calcolo (e quindi in qualche modo l'"intelligenza" che ci fornisce il servizio).

Chiaramente i datacenter devono assicurare un servizio affidabile, continuativo nel tempo, e assicurare che il carico sulle macchine sia equamente distribuito.

15.3.1 Reti datacenter

All'interno dei datacenter si vanno a creare gerarchie di switch piuttosto complesse. I rack server sono composti da 20-40 *server blade*, che rappresentano gli host veri e propri. Ogni rack ha poi un suo switch detto **TOR** (*Top Of Rack switch*). I TOR sono connessi da switch di livello 1 (circa 16 per volta), e questi a loro volta sono connessi da switch di livello 2 (ancora, circa 16 per volta).

Infine, l'intera rete viene esposta all'esterno usando router (più di uno per assicurare la ridondanza del servizio).

15.4 VLAN

Quando le reti LAN diventano molto grandi, si iniziano ad avere diversi problemi dati dal fatto che si ha a disposizione un **singolo dominio di broadcast**: tutti i nodi devono ricevere tutto il traffico di livello 2 (ARP, DHCP, frame con MAC broadcast, ecc...). Questo porta chiaramente a problemi di efficienza, sicurezza e privacy.

Notiamo che in Ethernet moderno non è invece vero dire che si ha un **singolo dominio di trasmissione**: abbiamo visto come gli switch permettono di instradare il traffico fra singoli host, risparmiandolo agli altri host sulla rete.

Un'altra problematica è data dal fatto che il collegamento **fisico** agli switch è anche un collegamento **logico**: si potrebbe volere che un host collegato *физическo* ad un certo switch, debba essere collegato *логически* ad un altro switch.

Le **VLAN** (*Virtual Area Network*) sono la soluzione a questi problemi. Gli switch che implementano funzionalità VLAN possono essere configurati per definire più LAN *virtuali* su una singola infrastruttura LAN fisica.

15.4.1 VLAN a porte

Si possono avere **VLAN basate a porte**: le porte degli switch vengono raggruppate (da un certo software di configurazione dello switch), così che un singolo switch si comporti come più switch virtuali. Le switch virtuali sono confinate fra di loro: anche se è il solito switch fisico a gestire il traffico, questo non verrà instradato da una VLAN all'altra.

Invece delle porte, si potrebbero usare anche i MAC di destinazione dei frame Ethernet.

Per inoltrare i pacchetti fra più VLAN, si ha quindi bisogno di un vero e proprio router (stiamo effettivamente simulando più reti locali, e quindi più switch separati). Nella pratica, i produttori di switch VLAN spesso vendono combinazioni di switch e router.

Le VLAN possono essere distribuite fra più switch: in questo modo più switch fisici rappresentano una singola unità logica (o parte di essa, o di più unità logiche). Questo si implementa sfruttando una porta (**trunk port**) degli switch coinvolti al trasporto di frame da uno switch all'altro.

15.4.2 Frame VLAN

Dopo tutte le funzionalità di VLAN che abbiamo aggiunto agli switch, i frame che vengono inviati non potranno più essere a di tipo 802.1 Ethernet semplice, ma devono essere 802.1Q VLAN.

Quello che si fa è estendere il campo **type** del frame Ethernet di 4 byte, introducendo:

- 2 byte di **TPID** (*Tag Protocol Identifier*), che contiene una stringa fissa;

- 2 byte di **TCI** (*Tag Control Information*), che contiene 1 bit di identificatore VLAN e 3 bit di priorità.

15.5 WAN

Il principio dello switching può essere implementato su reti non solo locali, ma anche a grande copertura geografica. Le reti che si vanno a formare vengono dette **WAN** (*Wide Area Network*).

La topologia più adatta a questo punto non è più quella gerarchica che avevamo visto, ma una a *mesh*, dove ogni switch è collegato con un certo numero di altri switch, > 1 . Questo permette ridondanza nel caso i link, o addirittura interi switch, fuori operazione.

Avere ridondanza introduce il problema del **routing**: bisogna capire qual'è il percorso migliore da seguire per instradare frame da un host all'altro. Rimandiamo a dopo questo processo.

15.5.1 Forwarding

Parliamo invece del problema del **forwarding**, cioè come inoltrare effettivamente i pacchetti sul percorso desiderato.

Individuiamo innanzitutto i 2 tipi di servizio che potremmo offrire:

- **Connectionless**: in questo caso non si stabilisce alcuna connessione preliminare, e ogni pacchetto è gestito in maniera stateless come se fosse il primo. Questo tipo di servizio è detto anche **datagram** (in quanto è il modello di servizio usato in Internet).

In questo caso non prevediamo nessun tipo di connessione preliminare, e non chiediamo agli switch o router intermedi di mantenere alcuno stato sullo stato della connessione (in quanto non esiste un concetto di "connessione" in primo luogo).

I pacchetti vengono gestiti sulla base dell'indirizzo di destinazione: più pacchetti per la stessa destinazione potrebbero però prendere percorsi diversi sulla rete (magari per congestioni, ecc...).

Per gestire l'instradamento si fa uso di **forwarding table** simili alle switch table già viste: la differenza sta nel fatto che si possono mappare non singoli indirizzi, ma **range** di indirizzi, alle interfacce.

- **Connection**: prima che la comunicazione inizi, un *circuito virtuale* (o **VC**, *Virtual Circuit*) viene stabilito. Da questo punto in poi, ogni pacchetto inviato segue lo stesso percorso nella rete.

Un circuito virtuale è un modo per emulare la funzionalità delle reti a **commutazione di circuito** (ad esempio la vecchia rete telefonica). In questo caso si prevedono fasi esplicite:

1. Fase di **chiamata**, dove l'host mittente *chiama* l'host destinatario;
2. Fase di **accettazione** della chiamata, dove l'host destinatario inoltra un nuovo messaggio, appunto di accettazione della chiamata;
3. A questo punto la comunicazione può effettivamente avvenire. Chiaramente, prima o poi seguirà una fase di **chiusura** del circuito virtuale, dove con uno scambio di messaggi simile al precedente mittente e destinatario negoziano la chiusura delle comunicazioni;

Il circuito virtuale, dal lato implementativo, deve essere qualcosa di noto sia agli host che agli switch o router che appartengono alla rete.

Prevediamo quindi per ogni VC:

- Un **percorso** da mittente a destinatario;
- **Numeri VC**, uno per ogni link lungo il circuito;
- Le entrate delle **forwarding table** degli switch o router lungo il percorso (la trattazione è generica, vale sia al livello 2 che al livello 3).

I pacchetti che viaggiano su un VC portano il numero di VC. Questo può quindi essere cambiato da switch o router, consultando la loro forwarding table. Ciò significa che gli switch mantengono informazioni sullo stato della connessione.

15.6 Internetworking

Veniamo adesso ad introdurre il concetto di **internetworking**, cioè collegamento di più reti fra di loro.

Visto che più reti sfruttano standard diversi di comunicazione, bisognerà dotarci di una qualche "*lingua franca*" che ogni rete può usare per parlare con altre reti. Come abbiamo anticipato, questa sarà rappresentata dai **datagrammi** Internet.

16 Lezione del 28-10-25

Ritorniamo sull'argomento dell'*internetworking*.

Al livello network il livello superiore è il *transport*: vorremo prendere i *segmenti* transport e trasportarli in *datagrammi*.

- Il **trasmettitore** dovrà quindi prendere i segmenti dal livello superiore e incapsularli in datagrammi;
- Il **ricevitore** dovrà invece prendere i datagrammi, estrarne i segmenti e fornirli al livello superiore.

Questo protocollo è implementato in tutti i dispositivi a livello network: questo significa **host** e **router** (gli switch non sono dispositivi a livello network, ma a livello datalink).

Il compito dei *router* è quindi chiaro a livello network: questi leggono i campi header dei *datagrammi IP* che ricevono, e inoltrano (*intradano*, da *routing*) tali datagrammi verso la destinazione finale (attraverso la porta di uscita giusta) consultando una *routing table*.

16.0.1 Funzioni livello network

Il livello network implementa due funzioni fondamentali:

- **Forwarding**: spostare pacchetti da un *link di ingresso* del router ad un appropriato *link di uscita* (quella che prima abbiamo chiamato *porta di uscita*);
- **Routing**: determinare il percorso fatto dai pacchetti dalla sorgente alla destinazione. In questo caso dobbiamo dotarci di appositi *algoritmi di routing*.

16.0.2 Data plane e control plane

Abbiamo quindi che i router lavorano fondamentalmente su due *piani*: **piano dati** (*data plane*) e **piano di controllo** (*control plane*).

- Il **piano dati** è una funzione *locale*, che coinvolge il singolo router, e che riguarda lo spostamento del datagramma da porta di ingresso del router a porta di uscita;
- Il **piano di controllo** è invece una funzione *globale* alla rete, condivisa fra i router, che determina come il datagramma deve essere instradato fra i router nel percorso (**route**) da host mittente e host destinatario.

Esistono 2 modi per realizzare il piano di controllo:

- Adottare degli algoritmi di routing *tradizionali*, implementati direttamente nel router.

Chiaramente, questi algoritmi di routing dovranno essere distribuiti, cioè eseguiti da tutti i router che lavorano insieme per assicurare il corretto instradamento dei pacchetti (aggiornando le forwarding table di ogni router);

- Sfruttando il *Software Defined Networking (SDN)*, cioè implementando gli algoritmi di routing nei server e facendo semplicemente eseguire al router il percorso trovato.

In questo caso si dovrà prevedere un controllore remoto che si occupi di installare le forwarding table in ogni router.

16.0.3 Modello di servizio

Possiamo interrogarci quali sono le caratteristiche del servizio che Internet è capace di offrire attraverso il protocollo IP (quello che nell'Internet moderno implementa il livello network).

Abbiamo che IP non garantisce nulla riguardo a *bitrate*, *perdita di dati*, *ordinamento dei dati* o *temporizzazione*.

In questo rappresenta un servizio **best effort**:

- Non si garantisce la consegna assicurata dei pacchetti;
- Non si assicurano costanti minime di temporizzazione o l'ordinamento dei pacchetti;
- Non si assicura la larghezza di banda massima al flusso da host a host.

Il protocollo best effort potrebbe sembrare inizialmente svantaggioso, ma per Internet la fortuna è stata proprio rendere la rete "*stupida*", e concentrare l'intelligenza all'*edge* (agli host, si pensi ai protocolli di trasporto, ecc...).

Questo rende infatti molto più semplice ed economico realizzare l'infrastruttura di rete, e allocando abbastanza risorse in un secondo momento si riesce ad ottenere un servizio "*abbastanza buono*" anche per applicazioni intensive sul bitrate come le video-chiamate, ecc...

16.1 Router

Un **router** è in maniera estremamente generica un computer che ha delle **porte di entrata** e delle **porte di uscita**.

Fra porte di ingresso e porte di uscita deve essere posta una qualche **rete di switching** estremamente veloce che permetta la commutazione da ingresso a uscita. Questa implementa il *data plane*, e quindi il *forwarding*.

Questa dovrà quindi essere governata da un **processore di routing**. Questo implementa il *control plane*, e quindi il *routing*.

Le porte del router implementano i primi 2 livelli dello stack protocollare, cioè:

1. Livello **fisico**: si occupa della ricezione bit a bit del datagramma;
2. Livello **link**: ad esempio di tipo Ethernet, si occupa del rilevamento errori e si libera del frame di livello link;
3. Una volta superato il livello link, i payload dei frame di livello link (cioè i datagrammi veri e propri) vengono inoltrati effettuando un lookup sul loro campo header.

Il forwarding può essere implementato in 2 modi:

- Forwarding **basato su destinazione**: basato solo sull'indirizzo IP estratto dall'header di datagramma.

Abbiamo introdotto le *switch table* in 15.2.1. Le **forwarding table** usate dai router sono molto simili: si definiscono *range* di indirizzi che mappano a porte di uscita dei router.

Il modo più semplice per mappare range di indirizzi è memorizzare indirizzi **prefix** (*prefissi*), cioè memorizzare i primi n bit di indirizzo, e l'intero n stesso. Quando i range si sovrappongono, si usa il cosiddetto **longest prefix matching**, cioè si prova ad usare sempre l'indirizzo con prefisso di rete più lungo.

- Forwarding **generalizzato**: basato su tutti i campi dell'header di datagramma.

Si prevede solitamente una coda per i datagrammi in arrivo, che viene riempita quando i datagrammi arrivano più velocemente di quanto si riesca ad instradarli.

16.1.1 Rete di switching

La **rete di switching** (dette anche *switching fabric*) si occupa di trasferire i pacchetti dal link di ingresso al link di uscita. Definiamo **switching rate** la frequenza con cui i pacchetti vengono trasferiti da input a output. Solitamente viene espressa come un multiplo del bitrate di linea input/output: chiaramente con N input si cerca di avere uno switching rate pari ad N volte il bitrate di linea.

Esistono 3 tipi principali di switching fabric:

1. Basato su **memoria**: si direbbe a *memoria condivisa*. Solitamente viene infatti implementato nei computer tradizionali che fanno switching attraverso la CPU.

In questo caso si prendono i pacchetti in entrata e si copiano in memori asistema. Dalla memoria sistema si inoltrano quindi verso le porte di uscita.

Questo chiaramente è limitato dal bitrate della memoria (bisogna passare dal bus 2 volte per datagramma, una volta in entrata dalla porta di ingresso alla memoria, una volta in uscita dalla memoria alla porta di uscita);

2. A **bus**: si usa un bus per collegare le porte di ingresso direttamente alle porte di uscita.

In questo caso il problema principale è chiaramente la competizione sul bus, e il fatto che la velocità di trasferimento è limitata dal bitrate del bus;

3. A **rete di interconnessione**: in questo caso si sfruttano tecnologie sviluppate inizialmente per connettere i processori nei sistemi multiprocessore (*crossbar*, *clos networks*, ecc...).

L'evoluzione naturale di questi sistemi è l'uso di **switch multistage**: switch $n \times n$ realizzate attraverso più stadi formati da switch più piccole. Questo approccio può sfruttare un certo grado di *parallelismo*: si divide il datagramma in più *celle* di lunghezza fissa, che vengono immesse nello switch e riassemblate in uscita.

Notiamo che ulteriore parallelismo può essere sfruttato prevedendo più strutture di commutazione parallele (se vogliamo, più *data plane* paralleli).

16.1.2 Accodamenti in ingresso

Se la rete di switching ha bitrate più lento del bitrate combinato delle porte di ingresso, si potrebbero formare delle **code** alle porte di ingresso. Questo porta a delay ed eventualmente anche perdita di dati.

- Questo accade ad esempio in caso di **competizione** su una singola porta di uscita: se 2 pacchetti destinati alla stessa porta di uscita arrivano contemporaneamente a porte di ingresso diverse, uno dei due dovrà necessariamente aspettare che il secondo venga instradato;
- I pacchetti successivi a blocchi per competizione su porte di uscita possono quindi subire blocking **HOL** (*Head Of Line*): in questo caso bisogna attendere che la porta di ingresso si liberi perché il pacchetto in arrivo venga effettivamente processato.

16.1.3 Accodamenti in uscita

Anche le porte di uscita possono essere suscettibili a code: potremmo infatti richiedere del **buffering** in uscita quando la logica di commutazione è più veloce della logica di uscita (quella che implementa i livelli datalink e fisico, in quest'ordine).

16.1.4 Gestione di perdite

Anche in questo caso possono verificarsi delay ed eventualmente **perdita di dati**. Alcune politiche di gestione del buffer che potremmo adottare in questo caso potrebbero essere:

- **Tail drop**: si scarta ("drop") il pacchetto in arrivo sul buffer pieno;
- **Prioritaria**: si scartano i pacchetti su base prioritaria.

Può essere utile anche **marchiare** ("marking") quali pacchetti conviene scartare per primi in caso di congestione: ad esempio, i pacchetti ICMP saranno i primi ad essere scartati.

16.1.5 Scheduling in uscita

Anche su come si effettua lo **scheduling** dei pacchetti in uscita si possono fare delle considerazioni. Dovremo infatti prevedere un *server* (di livello link) che si occupa di accedere alla coda dei pacchetti in uscita, e immettere nel livello fisico i pacchetti.

Questi possono essere selezionati (*schedulati*) secondo più modalità:

- **FCFS** (*First Come First Served*): il primo pacchetto viene inoltrato;
- **Prioritaria**, cioè si assegna una qualche priorità assegnata dall'applicazione o al router al pacchetto;
- **RR** (*Round Robin*): si servono ciclicamente tutti i pacchetti;
- **Weighted fair queueing**: si cerca di assegnare probabilità eque di trasmissione ad ogni pacchetto.

Approcci alternativi al FCFS possono essere utili in quanto non tutte le applicazioni che girano in rete sono uguali: alcune sono più importanti di altri (l'esempio precedente, pensiamo ICMP contro streaming ad alta velocità).

- Un buon approccio ibrido può essere quello a **code multiple** FCFS. Si prevedono ad esempio due code, una **prioritaria** e una **non prioritaria**, gestite entrambe internamente come FCFS.

Il traffico in entrata viene classificato fra queste due code, e quindi il traffico prioritario viene spedito per primo.

Un problema apparente di questo approccio è la *starvation* del traffico non prioritario.

- Per risolvere il problema della starvation potremmo prevedere lo stesso meccanismo di classificazione, e distinguere fra le code non in maniera prioritaria ma usando il **WFQ** (*Weighted Fair Queueing*).

Questo non è altro che un approccio round robin generalizzato dove ogni classe i ha peso w_i e ottiene un livello di servizio (magari in probabilità) ad ogni ciclo di:

$$w_g = \frac{w_i}{\sum_j w_j}$$

Questo approccio è quello che viene più usato nei moderni router.

16.2 Protocollo IP

Il **protocollo IP** è oggi il protocollo a fondamento del livello network. Ne esistono di altri, ma in Internet si usa principalmente IP.

Il livello network, e quindi il protocollo IP, ha il compito di implementare algoritmi di **path-selection** (protocolli di routing) sulla base di date *forwarding table*.

Il protocollo IP in particolare si occupa di definire:

- Il **formato** dei datagrammi;
- Le modalità di **indirizzamento** a livello network;
- **Convenzioni** per la gestione dei pacchetti.

Il protocollo IP si affianca all'**ICMP** (*Internet Control Message Protocol*), inizialmente pensato per la notifica degli errori e la diagnostica, oggi usato perlopiù per diagnostica.

16.2.1 Datagramma IPv4

Il **datagramma IP** versione 4 ha una struttura più complessa di quella che avevamo visto ad esempio per i frame Ethernet.

Si hanno quindi almeno 5 parole da 32 bit di *header*, e una sezione di *payload* di lunghezza variabile:

```

1 16 bit          16 bit
2 <ver> <header length> <service type>   <length>           % header
3 <identifier>          <flag>   <fragment offset>
4 <time to live>       <upper layer>    <checksum>
5 <source address (IPv4)>
6 <destination address (IPv4)>
7 <options>
8 <payload> (lunghezza variabile)           % payload

```

- Il campo **version** è su 4 bit, ed è seguito dalla **lunghezza dell'header**, il **tipo di servizio** offerto (inutilizzato, qui si metteva ad esempio il *marking* di congestione) e la **lunghezza dell'intero datagramma**;
- La lunghezza massima di un datagramma IP è di 64 Kb, anche se solitamente si rimane sui 1500 bytes o meno;
- Il campo **time to live** determina gli *hop* di rete (trasferimenti da router a router) che il pacchetto può ancora avere prima di essere scartato. Assumiamo che ogni router decrementi questo campo, appunto, ad ogni hop effettuato;
- Il campo **upper layer** contiene informazioni sul tipo di protocollo di trasporto usato (TCP o UDP);
- Si ha quindi un campo di **checksum** per il controllo di errori;
- Seguono i campi di **indirizzo** (sorgente e destinazione IP);
- Si ha un campo opzionale di **options** specifiche al pacchetto;
- Infine c'è il **payload** vero e proprio.

17 Lezione del 29-10-25

Riprendiamo l'argomento dei datagrammi IP.

17.0.1 Overhead di TCP/IP

Ricordiamo che il datagramma IP ha 20 byte di overhead per l'header. Visto che spesso trasporta payload rappresentato da segmenti TCP, che hanno anch'essi 20 byte di overhead, possiamo dire che l'overhead complessivo è $20 + 20 = 40$ byte di informazioni non direttamente collegate al payload che vogliamo trasportare.

Abbiamo poi che alcuni campi dell'header IP sono legati al protocollo di trasporto sovrastante. Avevamo già notato infatti il campo *upper layer*, contenente informazioni sul protocollo (TCP o UDP) usato.

17.0.2 Frammentazione IP

Il protocollo IP supporta un meccanismo di **frammentazione** e **riassembaggio**, usato quando si inviano in rete datagrammi (header + segmento nel payload) più grandi della cosiddetta **MTU** (*Max Transmission Unit*, cioè la dimensione massima dei frame di livello datalink).

I campi **identifier**, **flags** and **fragment offset** dell'header riguardano esattamente questo: l'*identifier* identifica un datagramma, e il *fragment offset* la posizione all'interno del datagramma specificato del frammento corrente. In sostanza, un datagramma è composto da frammenti, che sono anch'essi inseriti datagrammi per la trasmissione in rete.

Il campo **flags**, su 3 bit, determina la modalità della frammentazione. In particolare abbiamo i bit:

```

1 bit 0: riservato
2 bit 1: DF (don't fragment)
3 bit 2: MF (more fragments)

```

- Il campo **DF** (*Don't Fragment*) significa che questo datagramma non dovrà essere frammentato, e che se è più grande della MTU dovrà essere scartato;
- Il campo **MF** (*More Fragments*) indica invece che questo datagramma è seguito da altri frammenti di un datagramma più grande.

Se vale 0 può significare equivalentemente che è l'ultimo frammento, o che come datagramma non era stato mai frammentato. Il destinatario gestisce datagrammi frammento arrivati fuori ordine controllando se i datagrammi ricevuti finora, coi loro offset, riempiono completamente un dato buffer. Visto che un datagramma frammento arrivato in anticipo lascia necessariamente un "buco" (i datagrammi sono inviati per offset incrementali), questo funziona.

Notiamo a questo punto che il *fragment offset*, che rappresenta effettivamente l'offset del frammento all'interno del datagramma completo, è su 13 bit (mentre la dimensione del datagramma è su 16 bit). Questo significa che misuriamo gli offset non in byte, ma in blocchi da 64 bit (8 byte, in quanto $16 - 13 = 3 \rightarrow 2^3 = 8$).

Vediamo quindi come devono comportarsi i router:

- Il **trasmettitore** di livello network avrà quindi il (semplice) compito di, nota l'MTU, suddividere (*frammentare*) i datagrammi troppo grandi in più datagrammi "*di frammento*", che poi inoltrerà singolarmente sulla rete.

Questo, chiaramente, nel caso il campo DF non sia abilitato: in caso contrario semplicemente scarterà il datagramma;

- Il **ricevitore** dovrà occuparsi invece di controllare l'header dei datagrammi IP, capire quando è stato inviato un frammento di un datagramma più grande (attraverso il campo **flags**), e aspettare successivamente tutti i frammenti necessari a *riassembolare* il datagramma.

Nel caso non tutti i frammenti arrivino in un quanto di tempo utile, si scarta il datagramma e si va avanti. Questo non ci turba, in quanto abbiamo detto IP è *best effort*.

17.1 Indirizzamento IP

Iniziamo quindi a vedere nel dettaglio l'**indirizzo IP**.

Avevamo già introdotto l'*indirizzo MAC*, o *fisico*, associato alla NIC e assegnato su base globale (quindi solitamente unico per ogni dispositivo, ecc...). Gli indirizzi IP, come vederemo, sono invece assegnati su base **dinamica**.

Un indirizzo IP è un identificatore su 32 bit per **interfacce** di *host* e *router*. Un'**interfaccia** è la connessione fra host/router e il livello fisico. Solitamente gli host hanno un'interfaccia e i router più di una.

17.1.1 Subnet

Una **subnet**, o *sottorete*, è un insieme di interfacce che possono comunicare fisicamente l'una con l'altra senza l'intervento di un router di livello 3. Sono sottoreti le reti LAN che abbiamo studiato in 14.1.

Gli indirizzi IP sono fatti per gestire le sottoreti:

- La prima parte dell'indirizzo identifica la sottorete, e quindi più dispositivi sulla stessa sottorete hanno gli stessi bit più significativi (un numero variabile di questi, specificato dalla **subnet mask** o *maschera di sottorete*);
- L'ultima parte dell'indirizzo identifica quindi i singoli dispositivi.

Riguardo alla maschera di sottorete, abbiamo che questa è notata come `<addr>/<mask>`, dove `<mask>` è un numero da 0 a 32 che identifica quanti dei primi bit dell'indirizzo sono dedicati alla sottorete.

Possiamo distinguere i seguenti tipi di indirizzo:

- Indirizzi di **tipo A**: 8 sottorete + 24 dispositivo;
- Indirizzi di **tipo B**: 16 sottorete + 16 dispositivo;
- Indirizzi di **tipo C**: 24 sottorete + 8 dispositivo;

17.1.2 CIDR

CIDR sta per *Classless InterDomain Routing*, e rappresenta il formato standard degli indirizzi IP appena descritto:

¹ `a.b.c.d/x`

dove `x` è il numero di bit nella porzione di sottorete dell'indirizzo. Notiamo che, a scapito di quanto abbiamo detto sul tipo degli indirizzi nella scorsa sezione, `x` non deve necessariamente essere multiplo di 8 bit, ma può essere su un numero arbitrario di bit fra 0 e 32.

Potremmo interrogarci su come vengono ottenuti gli indirizzi IP.

- Per gli host, la risoluzione dell'IP all'interno della sua sottorete è fatta dall'amministratore di sistema, o dal protocollo **DHCP** *Dynamic Host Configuration Protocol*;
- Per le sottoreti, gli amministratori di rete si rivolgono agli ISP, che a loro volta (attraverso una struttura gerarchica) si rivolgono all'**ICANN** (*Internet Corporation for Assigned Names and Numbers*).

17.2 DHCP

Il **DHCP** (*Dynamic Host Resolution Protocol*) è un protocollo di livello application che si occupa di fornire ai client di un server (il server DHCP, che si assume gestisca una certa sottorete) i loro indirizzi IP.

Questo permette agli host di accedere ai seguenti servizi:

- Richiedere ed ottenere un indirizzo IP quando si unisce ad una rete;
- Riutilizzare tale indirizzo finché si mantiene in contatto con la rete.

Si assume che il server DHCP sia collegato al router che serve la sottorete, in modo che possa servire tutte le sottoreti su tale router. Solitamente, DHCP sta sulla porta 67.

Quando un host si collega alla rete:

1. Avrà bisogno di un indirizzo IP, e quindi interrogherà la rete (con indirizzo broadcast, 255.255.255.255 (tutti i bit a 1)) per un server DHCP. Questa fase si dice di **DHCP discover**;
2. Il server DHCP, se presente sulla rete, risponderà al DHCP discover con un **DHCP offer**, cioè offrendo un indirizzo IP al client. Notiamo che anche la risposta del server è in broadcast: si disambigua fra host sfruttando un *transaction ID* univoco generato dal client;
3. Il client può quindi rispondere con una **DHCP request**, dove richiede di poter sfruttare l'indirizzo IP usato. Anche questo messaggio è inviato in broadcast: potrebbe essere inviato all'IP del sever (ormai noto dalla fase di offer), ma il protocollo viene definito in questo modo;
4. Il server invia allora l'ultimo messaggio, il **DHCP acknowledge**, inviato sempre in broadcast, che dà la conferma al client di poter iniziare ad usare l'indirizzo IP ottenuto.

Vediamo che le fasi del DHCP rispettano il cosiddetto acronimo **DORA**: *Discover, Offer, Request and Acknowledge*.

Notiamo inoltre che queste richieste sono associate ad un certo *lifetime*, che rappresenta il tempo di vita per cui la richiesta viene presa in considerazione dal server (oltre il lifetime, l'offerta dell'indirizzo IP viene scartata e quell'indirizzo può essere offerto ad altri host).

Il DHCP può fornire più informazioni, non solo l'indirizzo IP:

- Il **default gateway** della rete, cioè il router da contattare per uscire dalla sottorete;
- I **server DNS** predefinito e secondario sulla rete;
- La lunghezza della maschera di sottorete sulla rete locale (che chiaramente all'host è ignota).

Per gli indirizzi forniti dal server DHCP è solitamente previsto un **lease time**, cioè un tempo limite entro il quale l'indirizzo IP è valido.

18 Lezione del 31-10-25

18.0.1 Limitazioni di IPv4

Abbiamo visto come gli indirizzi IP (versione 4) formano uno spazio di 32 bit (dimensione 2^{32} , quindi circa 4 miliardi). L'**ICANN** (*Internet Corporation for Assigned Numbers Authority*) gestisce tale spazio, attraverso 5 **RR** (*Registri Nazionali*), che a loro volta allocano i loro registri privati. Questi registri allocano poi blocchi di indirizzi IP agli ISP, che a loro volta li forniscono ad organizzazioni, abitazioni, ecc... (secondo il **CIDR**, *Classless InterDomain Routing*).

Il problema è che lo spazio di indirizzi IPv4 (quelli a 32 bit) è stato esaurito nel 2011. Meccanismi come il **NAT** aiutano a risolvere questo problema, ma ciò non toglie che uno spazio a 32 bit per gli indirizzi Internet è effettivamente poco. Per questo motivo è stata introdotta la *versione 6* del protocollo IP (**IPv6**), che ha uno spazio di indirizzi a 128 bit.

La transizione verso IPv6 è attualmente in corso, e il protocollo ad oggi non è sempre supportato.

18.1 NAT

Il **NAT** (*Network Address Translation*) è un meccanismo secondo il quale più dispositivi su una rete locale condividono lo stesso indirizzo IPv4 per il resto del mondo.

Ricordiamo che alcuni indirizzi IP sono riservati:

- L'indirizzo *tutto a 0* (0.0.0.0) viene usato dagli host che ancora non hanno un loro indirizzo (ad esempio come sorgente nel DHCP);
- L'indirizzo *tutto a 1* (255.255.255.255) è l'indirizzo di *broadcast*. Esiste anche un'indirizzo di broadcast composto dalla maschera di rete, più il resto dei bit a 1;
- Alcuni spazi di indirizzamento sono **privati**: questi sono 10.0.0.0, 127.0.0.0, 192.168.0.0, ecc... Gli indirizzi privati non possono essere usati per collegarsi alla rete Internet pubblica (non sono indirizzi *pubblici*), ma si limitano alla rete locale dove vengono usati.

Il modo in cui possono tornare utili gli indirizzi privati è effettuando un'apposita traduzione dalla rete locale alla rete Internet pubblica, effettuata appunto dal **NAT**, quando i datagrammi IP che vengono inviati sulla rete privata sono destinati ad un host fuori dalla rete privata.

In questo caso si sostituisce all'indirizzo privato un indirizzo condiviso fra tutti gli host sulla rete privata (sostanzialmente quello del router). Dall'altra parte della rete, l'unica cosa che distinguerà datagrammi diversi dalla stessa rete NAT sarà il numero di porta.

Dal punto di vista implementativo, quindi, un router che supporta NAT deve:

- Per i datagrammi **in uscita**, rimpiazzare l'indirizzo privato col suo IP, e assegnare al datagramma una nuova porta;
- **Ricordare** in una tabella, detta *NAT Translation Table*, ogni coppia sorgente (indirizzo privato all'interno della rete, porta originale) destinazione (indirizzo pubblico, nuova porta);

- Per i datagrammi **in entrata**, effettuare la ricerca al contrario, rimpiazzando l'indirizzo pubblico con quello privato, e assegnando al datagramma la sua porta originale.

Un problema che potremmo avere è come indirizzare server su NAT. Nel nostro esempio attuale, dietro il NAT si trovava un client: questo fa le richieste, il router le traduce, il server inoltra risposte e il router sa come inoltrarle al client.

Un server, di contro, non potrebbe apparentemente essere indirizzato se si trovasse dietro un NAT. Una soluzione è quella di prevedere entrate nella NAT Translation Table, impostate manualmente, che associano ad esempio a ogni richiesta sulla porta 80 un singolo server all'interno del NAT. In questo modo ogni client che vuole accedere al web server all'indirizzo NAT (porta 80) potrà farlo, e il router tradurrà l'indirizzo all'indirizzo privato corrispondente al web server (porta 80).

18.2 IPv6

Abbiamo introdotto IPv6 IN 18.0.1, motivandone l'introduzione per:

1. Superare l'esaurimento degli indirizzi ormai associato ad IPv4;
2. Limitare l'overhead di 40 byte di header di IPv4.

Il protocollo IPv6, come vedremo, rimuove molte funzionalità presenti in IPv4 per essere più veloce ed efficiente.

18.2.1 Datagramma IPv6

Il **datagramma IP** versione 6 ha la seguente struttura:

```

1 16 bit          16 bit
2 <ver> <pri> <flow_label>           % header
3 <payload len>      <next hdr>  <hop limit>
4 <source address (IPv6)>
5 (128 bit)
6 <destination address (IPv6)>
7 (128 bit)
8 <payload> (lunghezza variabile)       % payload

```

- Il campo **version** è su 4 bit, come per IPv4. Seguono campi di **priorità** e identificatori di **flusso** (il concetto di flusso non è ben definito);
- Al contrario di IPv4, allochiamo la **lunghezza del payload** (anziché dell'intero datagramma). Questo anche per il fatto che IPv6 non prevede *opzioni*, che in IPv4 potevano variare la lunghezza dell'header;
- Il **checksum** non viene introdotto, in quanto andava ricalcolato ad ogni hop (si associa un **limite di hop**, quello che in IPv4 chiamavamo *time to live*, che va aggiornato ad ogni hop);
- **Next header** rappresenta il tipo dell'header di livello 4 incapsulato;
- Come sempre, segue il **payload** vero e proprio.

Notiamo che, oltre al checksumming e le opzioni, IPv6 rimuove anche il supporto alla *frammentazione*. Deleghiamo infatti tale compito al livello superiore, e ci limitiamo a scartare i datagrammi troppo grandi per il livello datalink.

Notiamo che anche il protocollo *ICMP* viene di conseguenza aggiornato, per fornire messaggi di notifica su quali datagrammi vengono scartati per dimensioni troppo grandi.

18.2.2 Coesistenza IPv4/IPv6

Ad oggi IPv4 e IPv6 *coesistono* nella rete Internet: la transizione non è stata effettivamente ultimata e gli host potrebbero usare uno dei 2 protocolli.

Questo può essere realizzato usando router che conoscono entrambi i protocolli.

19 Lezione del 04-11-25

19.0.1 Tunneling

Continuiamo a parlare della coesistenza fra IPv4 e IPv6. Abbiamo già introdotto nella scorsa lezione l'opzione della *dual architecture*, cioè lo sviluppo di router che parlano sia la "lingua" dell'IPv4 che dell'IPv6.

Un'altro approccio valido è il **tunneling**: si incapsulano i datagrammi IPv6 all'interno di datagrammi IPv4, in modo che questi possano essere instradati da infrastruttura pensata per IPv4 (cioè ad esempio attraverso una rete Ethernet che supporta solo IPv4).

Chiaramente questo approccio porta a 2 visioni separate:

- La **visione logica**, che vede i router IPv4/v6 come connessi, appunto, da un *tunnel IPv4*;
- La **visione fisica**, che vede i router IPv4/v6 semplicemente connessi ad una rete di router IPv4, su cui instradano anziché datagrammi IPv6, datagrammi IPv4 che incapsulano nel loro campo payload datagrammi IPv6.

19.1 ARP

Il protocollo **ARP** (*Address Resolution Protocol*) è usato per associare indirizzi MAC (utili per l'indirizzamento su rete locale) ad indirizzi IP.

Ogni nodo IP (cioè host e router) ha una tabella ARP che contiene mappe IP/MAC per gli altri nodi "adiacenti" (e chiaramente un certo time-to-live di validità dell'associazione, in quanto ricordiamo gli IP sono variabili (e fino ad un certo punto lo sono anche i MAC)).

Il funzionamento di ARP è il seguente:

1. Poniamo che *A* voglia inviare un datagramma a *B*, e *B* non sia nella tabella ARP di *A*;
2. *A* invierà quindi un pacchetto di richiesta ARP in broadcast a tutti i nodi sulla rete locale;
3. *B* riceverà il pacchetto di richiesta ARP, e se questo è ben formato sarà l'unico a rispondere, inoltrando ad *A* il suo indirizzo MAC;

4. A potrà quindi inviare il datagramma a *B*. Inoltre, ci aspettiamo che *A* memorizzi il MAC ricevuto nella sua tabella ARP (associandolo appunto all'IP di *B*): in questo modo per successive trasmissione non dovrà ripetere la richiesta.

19.2 ICMP

Il protocollo **ICMP** (*Internet Control Message Protocol*) è stato già introdotto più volte. Viene usato dagli host e dai router per scambiarsi informazioni di livello network come notifiche, errori, ecc... Inoltre, supporta il meccanismo del *ping*, cioè i pacchetti di **echo**.

Il protocollo ICMP è sempre un protocollo di livello network, ma costruito su IP. Ha infatti 3 campi che vengono incapsulati nel datagramma IP:

- **Tipo** del messaggio;
- **Codice** del messaggio, che assieme al tipo permette di specificare completamente la natura del messaggio ICMP;
- **Header e primi 8 byte** del datagramma IP che ha causato la notifica.

Alcuni esempi di coppie tipo e codice ICMP sono i seguenti:

Tipo	Codice	Descrizione
0	0	Echo reply (ping)
3	0	Network destinazione irraggiungibile
3	1	Host destinazione irraggiungibile
3	2	Protocollo destinazione irraggiungibile
3	3	Porta destinazione irraggiungibile
3	6	Network destinazione sconosciuto
3	7	Host destinazione sconosciuto
8	0	Echo request (ping)
9	0	Route advertisement, usato dai router per pubblicizzare un percorso
10	0	Router discovery, usato dai router per presentarsi
11	0	TTL del datagramma scaduto
12	0	Header IP malformato

Riguardo a quanto detto su tipo e codice notiamo come, ad esempio, il tipo 3 è destinato ad errori di trasmissione (con il codice che discrimina *quale* fra diversi errori di trasmissione).

19.3 Forwarding generalizzato

Avevamo introdotto il modello **match plus action** per il forwarding: quando arriva un pacchetto, si fanno combaciare i bit con la tabella di forwarding (*match*) e quindi si fa una qualche *azione*. Nel **destination based forwarding** (*forwarding basato su destinazione*), questa azione è di inoltrare i pacchetti verso l'indirizzo IP destinazione, basandosi solamente sull'indirizzo IP destinazione.

Nel **generalized forwarding** (*forwarding generalizzato*), invece, si cerca di usare tutti i campi dell'header IP per informare l'azione compiuta. Questa azione, infatti, potrà essere non solo l'*inoltrare* il pacchetto, ma anche lo *scartare*, *copiare*, *modificare* o *loggare* il pacchetto.

Chiaramente questo approccio prevede di fornirsi di un'astrazione diversa dalla tabella di forwarding, che chiamiamo **tabella di flusso**.

- Nella tabella di flusso associamo esplicitamente campi *match* a campi *action*.
- Il **flusso**, appunto, che questa tabella definisce sarà definito da diversi campi dell'header (di livello link, network e transport). L'esempio tipico è di prendere sia i campi sorgente che destinazione;
- Le **azioni**, come abbiamo già detto, sono più variegate di quelle previste dal modello basato su destinazione (possono prevedere di *scartare* pacchetti, *archiviarli*, ecc...);
- Occorre definire una **priorità** per i pattern di match che si sovrappongono (come disambiguamo su quale eseguire?);
- Infine dovemo prevedere **contatori** per quanti byte e quanti pacchetti coinvolgono le regole definite, a scopo di statistica.

19.3.1 OpenFlow

OpenFlow è un protocollo standard per il forwarding generalizzato.

Ogni entrata di una tabella di flusso OpenFlow ha 3 campi:

1. **Match**: può offrire un'operazione di match per diversi campi header MAC, IP e TCP/UDP;
2. **Action**: definisce diverse operazioni, fra cui:
 - Forwarding verso porte;
 - Drop di pacchetti (scarta pacchetti);
 - Modifica campi negli header di pacchetto;
 - Incapsula pacchetto e inoltra a *controller*;
 - Ecc...
3. **Stats**: permette di contare quanti pacchetti di un certo tipo sono stati analizzati, o quanti byte sono passati sul router, ecc...

L'astrazione match plus action permette di unificare tutta una serie di dispositivi diversi:

- **Router**: implementando regole di instradamento basate sui campi header IP;
- **Switch**: implementando regole di instradamento basate sui campi header MAC, magari appoggiandosi anche su azioni di *flooding* (trasmissioni in broadcast);
- **Firewall**: facendo match su indirizzi IP e numeri di porta TCP/UDP, e sfruttando azioni di drop o instradamento pacchetto;
- **NAT**: facendo match su indirizzi IP e numeri di porta TCP/UDP e sfruttando azioni di modifica dei pacchetti (riscrittura indirizzi UP e porte TCP/UDP).

Riassumendo, abbiamo quindi che l'astrazione *match plus action* è basata su operazioni di match su più di un campo header, e azioni più variegate rispetto al semplice inoltro. Permette di definire il cosiddetto *forwarding generalizzato*, e OpenFlow ne è un'implementazione standard. Abbiamo che attraverso questa operazione possiamo relazionare reti effettivamente **programmabili**, cioè il cui comportamento può essere configurato, a livello di tabelle di flusso, da dispositivi di ordine superiore (che magari automatizzano la configurazione sulla base di informazioni di livello più alto).

19.4 Middlebox

Vengono detti **middlebox** tutti i dispositivi che non sono né host né router. Ad esempio, il *NAT* è un middlebox, come lo sono il *firewall*, l'*IDS (Intrusion Detection System)*, i *load balancer*, le *cache* e tutta un'altra serie di dispositivi ad uso specifico ad applicazioni (ad esempio si pensi all'infrastruttura richiesta dai *CDN*).

Le middlebox nascono come soluzioni proprietarie, e quindi a sorgente e specifiche *chiuse*. In seguito, ci siamo spostati verso le cosiddette "*whitebox*", cioè hardware implementato attraverso API open source, programmabili attraverso il match plus action.

Questo permette di avvicinarsi al cosiddetto **SDN (Software Defined Networking)**, cioè la centralizzazione del controllo e della configurazione dei dispositivi che compongono la rete, spesso da remoto (*cloud*, ecc...).

19.5 Comunicazione fra processi

Abbiamo visto finora la comunicazione fra **host**.

Studiando il livello application, però, abbiamo visto che veramente siamo interessati all'**IPC (Inter Process Communication)**, cioè la comunicazione fra più processi all'interno della stessa o più macchine.

Possiamo basarci sui livelli fisici, datalink e network per realizzare il trasporto fra macchine, e ci basiamo su un ultimo livello, il livello **transport**, per realizzare l'IPC. Ricordiamo che i protocolli di questo livello sono **TCP (Transmission Control Protocol)** e **UDP (User Datagram Protocol)**. Oltre alla comunicazione diretta (almeno dal loro punto di vista) fra processi, vedremo come i protocolli di livello transport possono implementare servizi come:

- **Multiplexing e demultiplexing** di più messaggi sulla stessa linea;
- **Reliable data transfer**, già visto in 10.3 al livello datalink;
- Controllo di **flusso** e **congestione**.

Il compito dei protocolli di livello transport è quindi quello di permettere la comunicazione *logica* fra più processi applicazione in esecuzione su macchine diverse. Ricordiamo, *logica*, in quanto la comunicazione effettiva avviene attraverso l'intero stack protocollare (fisico, datalink, network e quindi transport).

Al livello transport la comunicazione avviene in **segmenti** passati al livello network. Questo perché il livello transport non vede pacchetti, ma vede *byte*: si vuole infatti creare l'astrazione di un *flusso* continuo di byte, e un segmento non è altro che una parte di questo flusso (che viene quindi diretta in rete sotto forma di pacchetti).

Il livello transport dell'host mittente divide quindi i messaggi di livello application in più segmenti, che vengono ricomposti dal livello transport dell'host destinatario per essere quindi consegnati al relativo livello application, realizzando effettivamente questa astrazione di *byte stream* fra livelli application (cioè fra processi).

20 Lezione del 05-11-25

Riprendiamo la trattazione della comunicazione fra processi, in particolare riferendoci al livello transport implementato alternativamente dai protocolli *TCP* e *UDP*.

20.1 Multiplexing

Abbiamo detto che l'obiettivo è implementare la comunicazione fra *processi*, a determinate *porte* (che sono astrazioni costruite al livello *transport*).

Abbiamo che se si vuole implementare questa trasmissione per *porte*, bisogna prevedere un passo di **multiplexing** al client dei messaggi di tutte le applicazioni (e quindi tutte le porte attive), e di **demultiplexing** al server dei messaggi provenienti verso tutte le applicazioni (e quindi tutte le porte attive).

20.1.1 Demultiplexing connectionless

Possiamo parlare di demultiplexing **connectionless** quando il protocollo non ha concetto di "connessione" (si pensi a UDP). In questo caso, l'host riceve datagrammi IP che contengono:

- Indirizzi IP sorgente e destinazione;
- Il segmento livello transport (nel payload);
- Numeri di porta sorgente e destinazione, contenuto nell'header del segmento di livello transport.

Quello che fa l'host è usare l'indirizzo IP e il numero di porta per dirigere il segmento verso il socket corrispondente (cioè il socket che è associato alla porta destinataria).

20.1.2 Demultiplexing connection-oriented

Nel caso di protocolli che invece hanno concetto di connessione (si pensi a TCP), si parla di demultiplexing **connection-oriented**.

Un socket TCP è identificato da più informazioni rispetto al semplice socket connectionless. In particolare, si adotta una 4-upla:

1. Indirizzo IP sorgente;
2. Indirizzo IP destinatario;
3. Numero di porta sorgente;
4. Numero di porta destinatario.

Il vantaggio di questo approccio è che possiamo associare più socket alla stessa porta, in quanto la connessione è identificata anche dai dati provenienti dal client.

Questo è il classico esempio dei server TCP, che prevedono un socket di *ascolto* (magari il socket alla porta 80 nei web server), e poi tutta una serie di socket, individuati dalla 4-upla vista sopra, che gestiscono i singoli client. Chiaramente, i singoli client hanno prima parlato col socket di ascolto per ricevere un socket alla loro 4-upla.

20.2 UDP

Veniamo quindi alla trattazione vera e propria del protocollo **UDP** (*User Datagram Protocol*).

Ricordiamo che questo era un protocollo semplice, privo di servizi aggiuntivi, e basato sul modello *best service* di servizio. Questo significa che i pacchetti possono essere persi, o consegnati fuori ordine all'applicazione.

Inoltre, caratteristica fondamentale del protocollo UDP è che è *connectionless*: non c'è alcun concetto di connessione fra trasmettitore e ricevitore, nessun tipo di handshaking prima o dopo la comunicazione, e ogni segmento UDP viene considerato come a sé stante.

I vantaggi dell'UDP sono:

- L'estrema semplicità e velocità: il RTT di handshaking è inesistente, e il supporto software al ricevitore e al trasmettitore può essere estremamente semplice (non bisogna mantenere informazioni di stato);
- La dimensione ridotta degli header;
- Non esiste controllo di congestione e di flusso: oltre a risparmiare tempo sui controlli, si ha che in alcuni casi UDP può funzionare anche nonostante la congestione.

Le applicazioni di UDP comprendono il DNS, SNMP, e anche protocolli RDT come HTTP/3 (che implementa il suo livello di RDT, detto QUIC). Inoltre, UDP è largamente usato in applicazioni ad alte prestazioni come lo streaming e i videogiochi.

20.2.1 Segmento UDP

L'header di segmento UDP è estremamente semplice:

```

1 16 bit      16 bit
2 <source port> <destination port> % header
3 <length>      <checksum>
4 <payload> (lunghezza variabile) % payload

```

- Abbiamo l'indicazione del **numero di porta** e l'**indirizzo** destinatario, usati per il demultiplexing;
- La **lunghezza** del payload, cioè il messaggio di livello application, che può essere variabile;
- Un campo di **checksum** per l'intero segmento.

Questo è calcolato prenendo i contenuti del segmento come sequenze di numeri su 16 bit e sommandoli bit a bit in complemento a 1. Il checksum è memorizzato in complemento a 1 nel campo `<checksum>`: quello che fa il ricevitore è ricalcolare il checksum e confrontarlo con quello nel pacchetto (fare la AND).

Se risultano uguali, tutto è (apparentemente) a posto. Altrimenti il segmento è corrotto.

- Segue il **payload** vero e proprio.

20.3 Protocollo TCP

Veniamo quindi alla discussione del protocollo **TCP** (*Transmission Control Protocol*).

Questo è un protocollo *punto-punto*, cioè astrae un canale di comunicazione (*pipe*) fra due host specifici, trasmettitore e ricevitore.

Implementa anche il concetto di *byte stream*, cioè un flusso di byte ordinato e affidabile (non si possono perdere byte).

Infine, ricordiamo che è *full duplex*, cioè permette la comunicazione bidirezionale sulla stessa connessione (trasmettitore e ricevitore possono scambiarsi di ruolo contemporaneamente sulla stessa connessione).

Come abbiamo anticipato, la dimensione massima di un segmento TCP su protocollo datalink di tipo Ethernet è 1460 byte, in quanto si perdono 20 byte di header datagramma IP e 20 byte di header segmento TCP (lo vedremo fra poco).

Notiamo inoltre che TCP supporta gli *ACK cumulativi* sui segmenti inviati, capacità di *pipelining*, e supporto al controllo di *flusso* e *congestione*.

20.3.1 Segmento TCP

Il segmento TCP è visibilmente più complesso del segmento UDP:

```

1 16 bit           16 bit
2 <source port>      <destination port> % header
3 <sequence number>
4 <acknowledgement number>
5 <length> <inutilizzato> <flags> <receive window>
6 <checksum>          <urgent data pointer>
7 <options> (lunghezza variabile)
8 <payload> (lunghezza variabile)           % payload

```

- Immancabili sono il **numero di porta** e l'**indirizzo** destinatario, situati nella stessa posizione del segmento UDP;
- Segue il **numero di sequenza**, su 32 bit.

Notiamo la particolarità che, essendo TCP orientato al byte, questo indicizza byte anziché segmenti.

- Il **numero di acknowledge** indica il numero di sequenza del prossimo byte aspettato: in congiunzione al flag A, segnala che il segmento è di ACK.
- Il campo **<length>** è riferito alla **lunghezza** del solo header TCP;
- Il campo dei **flag** prevede diversi bit, che sono:
 - Bit C ed E, pensati per notificare le congestioni (oggi di fatto inutilizzati);
 - Bit U, pensato per segnalare dati urgenti (oggi di fatto inutilizzato);
 - Bit A, già nominato, segnala che il segmento è di ACK;
 - Bit P, indica che il ricevitore dovrebbe inviare i dati al livello superiore immediatamente (oggi di fatto inutilizzato);
 - Bit R (**RST**), S (**SYN**) e F (**FIN**), usati per la gestione della connessione.
- Il campo **receive window** è utile al flow control, in quanto specifica il numero di byte che il ricevitore è pronto ad accettare. In questo modo si riescono ad evitare buffer overflow quando si trasmettono più byte di quanti il ricevitore è capace di memorizzare;
- Prevediamo un campo **checksum** come in TCP;
- L'**<urgent data pointer>** è legato alla gestione di dati urgenti (come i bit U e P), oggi è perlopiù inutilizzato;
- Segue, come in IP, una sezione di **opzioni** TCP a lunghezza variabile;
- Infine c'è il **payload** vero e proprio che verrà trasmesso al socket TCP.

20.3.2 Numeri di sequenza TCP

Abbiamo detto che i **numeri di sequenza** in TCP sono orientati al byte (e non al segmento).

Quelli che gli host vogliono rendere disponibili alle applicazioni sono stream di byte. Il campo numero di sequenza di un segmento TCP contiene l'indice del primo byte contenuto nel campo payload del segmento stesso.

Avevamo visto in 11.2 come i byte (o comunque le unità di informazione minima che vogliamo spedire, al tempo le avevamo chiamate pacchetti) possono trovarsi, all'interno di una pipeline, in uno di 4 stati:

1. Inviati e seguiti da ACK del destinatario, quindi sostanzialmente "dimenticati" sia da ricevitore che da trasmettitore;
2. Inviati e non ancora seguiti da ACK, anche detti *in-flight*;
3. Utilizzabili ma non ancora inviati dal mittente;
4. Non ancora utilizzabili dal mittente.

Per inviare informazioni riguardo alla pipeline, usiamo i campi *numero di sequenza* e *numero di acknowledgement* del segmento TCP. In particolare, un segmento dal mittente che contiene un certo numero di sequenza segnala che i byte ivi contenuti stanno passando dallo stato 3 (non ancora spediti) a 2 (appena spediti). In risposta, un segmento di ACK (quindi con bit A alzato e numero di acknowledgement impostato) rappresenta una transizione da stato 3 (invia e senza ACK) a stato 2 (invia e con ACK).

In altre parole, possiamo dire che il *numero di sequenza* punta all'inizio della sezione 3 come vista dal mittente, mentre il *numero di acknowledgement* punta all'inizio della sezione 2 come vista dal destinatario.

20.3.3 Gestione connessione TCP

Veniamo quindi a come TCP implementa la **gestione delle connessioni**. Iniziamo col considerare la fase di **apertura**.

Vediamo del tipico codice per creare una connessione fra client e server attraverso i classici socket di BSD:

- Lato client:

```

1 // creiamo un socket
2 int sc = socket(AF_INET, SOCK_STREAM, 0);
3 // richiediamo la connessione del socket all'indirizzo server
4 int sts = connect(sc, (struct sockaddr*) &serv_addr, sizeof(serv_addr));

```

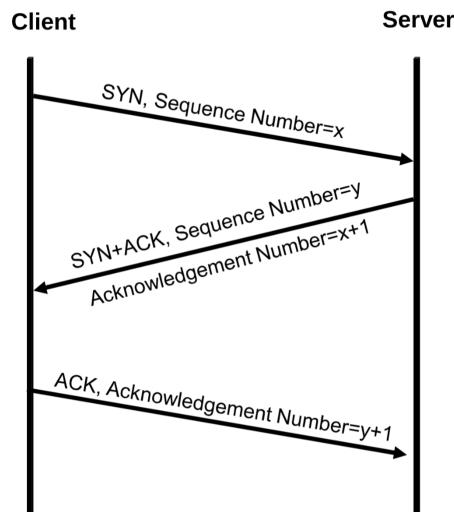
- Lato server:

```

1 // creiamo un socket di ascolto
2 int sl = socket(AF_INET, SOCK_STREAM, 0);
3 // leghiamo il socket di ascolto all'indirizzo del server
4 bind(sl, (struct sockaddr*) &serv_addr, sizeof(serv_addr));
5 // mettiamo in ascolto il socket
6 listen(sl, 10);
7 // accettiamo una connessione TCP col client dal socket di ascolto
8 int sc = accept(sl, NULL, NULL);

```

Quello che due host che eseguono questo codice mettono in atto in fase di connessione è il cosiddetto **handshake** a 3-vie:



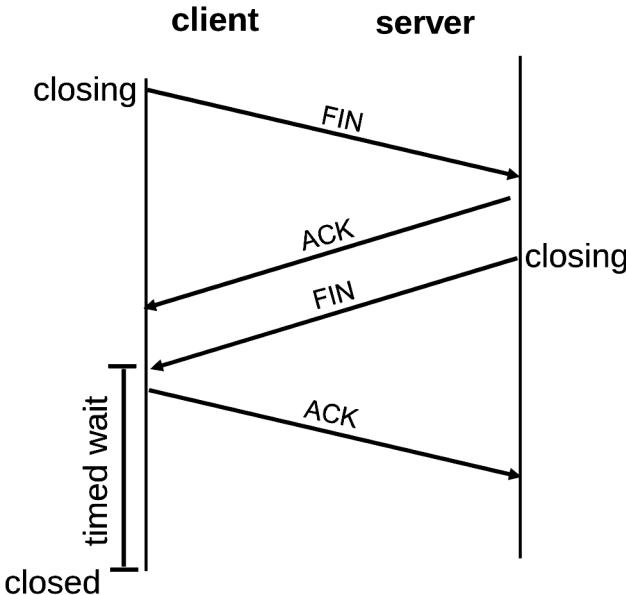
1. Inizia il client che si trova nel cosiddetto stato di LISTEN, che invia un messaggio TCP SYN al server. Il numero di sequenza viene scelto in maniera arbitraria. Con questa operazione il client transisce a stato SYN SENT;
2. Il server, anch'esso in stato di LISTEN risponde con un messaggio SYNACK, cioè l'ACK del messaggio SYN del client. Il numero di acknowledgement è impostato per seguire quello inviato dal client. Un altro numero di sequenza arbitrario è fornito dal server, e con questa operazione il server transisce a stato SYN RCVD;
3. Il client risponde con un ultimo ACK per il SYN ACK. Il segmento che contiene questo ACK potrebbe contenere dati per il server. Inoltre, il numero di acknowledgement è impostato per seguire quello inviato dal server. Con questa operazione, il client transisce a stato di ESTAB.

Ad ogni modo, questo ultimo ACK segnala al server che il client è vivo e pronto a trasmettere, e quindi di poter transire allo stato ESTAB.

I numeri di sequenza vengono scelti in maniera casuale per disambiguare, sul socket di ascolto, fra più richieste di connessione TCP concorrenti.

Veniamo all'inevitabile (a meno di errori) fase di **chiusura** della connessione. Questa è più complessa dell'apertura, in quanto l'ultimo ricevitore potrebbe avere dei dati ancora da leggere in fase di chiusura.

Il procedimento è quindi simile al seguente, ponendo che sia il client a richiedere la chiusura della connessione (non si riporta codice in quanto consiste semplicemente in due `close()` sui descrittori di socket):



1. Il client richiede la chiusura attraverso un segmento FIN;
2. Il server riceve il segmento FIN e risponde con un ACK.
Chiude quindi la connessione dal suo lato, inviando a sua volta un FIN;
3. Il client riceve il FIN del server, ed entra in una fase di *timed wait*, dove risponderà con ACK ad eventuali FIN;
4. Il server riceve l'ACK del cliente, e la connessione è effettivamente chiusa.

Interroghiamoci sui dettagli di questo processo. Abbiamo che in fase di creazione della connessione, sia server che client devono allocare risorse necessarie alla gestione della connessione (TCP è *stateful*, per cui richiede informazioni allocate sugli host). Queste informazioni dovrebbero ragionevolmente essere deallocate in fase di chiusura della connessione.

Questo però non è esattamente il caso: come abbiamo accennato, se l'host che chiude la connessione elimina subito i suoi descrittori, potrebbe verificarsi una situazione dove non è più in grado di fare ACK ad eventuali ultime richieste dell'altro host (mettiamo che il suo ACK si perde, ecc...). Per questo prevediamo una fase di *timed wait* (passo 3 nella sequenza appena vista), dove l'host che ha richiesto la chiusura aspetta, dopo la richiesta di chiusura dell'altro host, in modo da poter fare ACK ad eventuali richieste.

21 Lezione del 12-11-25

21.1 RDT su TCP

Abbiamo visto i frame del protocollo TCP e le sue funzionalità di gestione della connessione. Adesso concentriamoci su come TCP implementa il **RDT** (*Reliable Data Transfer*).

In 10.3 avevamo introdotto l'RDT su un mezzo qualsiasi (che al tempo associammo ad un link fisico inaffidabile). Vediamo adesso che tale mezzo inaffidabile può essere rappresentato anche da un link di livello network, come quello offerto dal protocollo IP. Quello che fa TCP è quindi implementare un livello di RDT al di sopra di questo link inaffidabile.

21.1.1 Stima del RTT

Un problema che dobbiamo risolvere prima di poter implementare l'RDT su TCP è quello della stima del **RTT** (*Round Trip Time*), cioè il tempo necessario a inviare un segmento e riceverne l'ACK.

Questo è infatti utile, ad esempio, per calcolare il valore del timeout per i segmenti TCP trasmessi:

- Se il timeout è troppo corto, si hanno ritrasmissioni inutili;
- Se il timeout è troppo lungo, si ha una reazione troppo lenta alla perdita di segmenti.

L'approccio che possiamo usare è quello di fare una stima *a posteriori* dei RTT precedenti, campionati in sede di trasmissioni precedenti. Una nota riguardo a questa stima è che preferiamo non valutare il tempo impiegato da segmenti che richiedono ritrasmissione, in quanto questi sono chiaramente degli *outlier* (sosterebbero la stima inutilmente verso l'alto).

Possiamo quindi realizzare la nostra stima come *media esponenziale mobile*:

```
1 avg_rtt = sample_rtt * h + avg_rtt * (1 - h)
```

dove `avg_rtt` è l'RTT medio che vogliamo considerare, `sample_rtt` è l'ultimo RTT che abbiamo valutato, e `h` è una qualche costante, con:

$$h \in (0, 1)$$

Al variare di `h` si può dare più o meno peso alle misurazioni passate dell'RTT. Una buona regola è di dare meno peso all'ultima misurazione (prendere `h < 0.5`, solitamente ~ 0.125).

Possiamo svolgere il calcolo dell'RTT stimato (chiamiamolo ERTT) sulla base degli RTT_i misurati, cioè espandere completamente la media esponenziale mobile:

$$ERTT_1 = RTT_0$$

$$ERTT_2 = h \cdot RTT_1 + (1 - h) \cdot RTT_1$$

$$ERTT_3 = h \cdot RTT_2 + (1 - h)h \cdot RTT_1 + (1 - h)^2 \cdot RTT_1$$

...

$$ERTT_{n+1} = h \cdot RTT_n + h(1 - h) \cdot RTT_{n-1} + h(1 - h)^2 \cdot RTT_{n-2} + \dots + (1 - h)^n \cdot RTT_0$$

che si riscrive semplicemente come:

$$ERTT_{n+1} = h \cdot RTT_n + (1 - h) \cdot ERTT_n$$

Quest'ultima formula è esattamente quello che lo pseudocodice riportato sopra codifica.

Risulta utile tenere conto anche della deviazione dell'RTT, stimata secondo lo stesso metodo:

```
1 avg_devrftt = abs(avg_rtt - sample_rtt) * k + avg_devrftt * (1 - k)
```

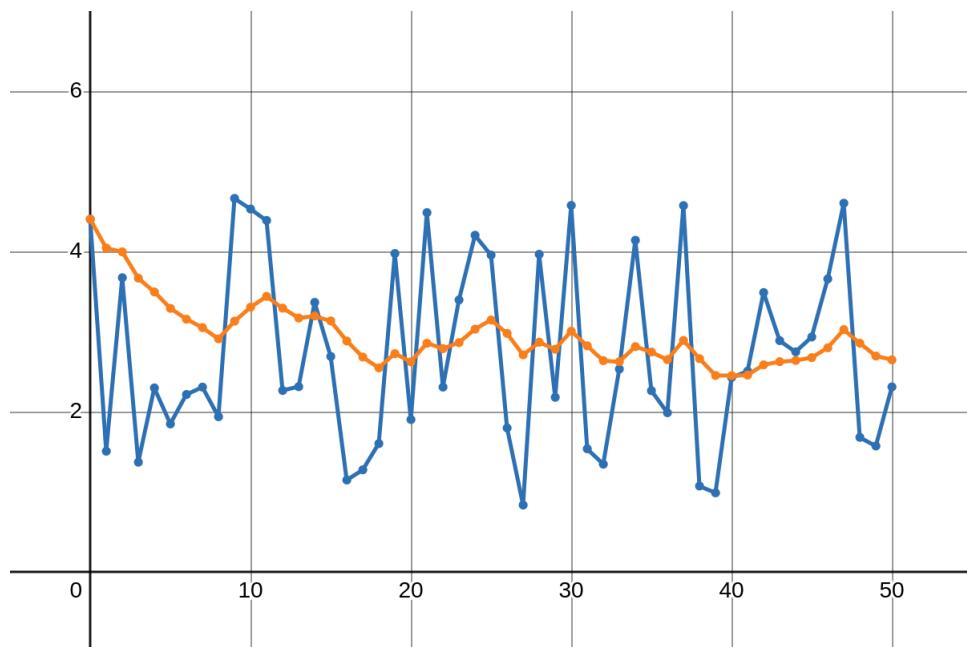
dove k è solitamente ~ 0.25 .

A questo punto avremo modo di calcolare il timeout ideale (secondo l'euristica presa), come:

```
1 timeout = avg_rtt + j * avg_devrss
```

dove j esprime la grandezza di un qualche "*margine di sicurezza*", con j solitamente ~ 4 .

Per concludere questa sezione, vediamo l'esempio di una stima su un dataset casuale, effettuata usando la media esponenziale mobile con $h = 0.125$:



dove in blu si è riportato la funzione originale, campionata ad intervalli $\Delta t = 1$ (per quanto ci riguarda, l'RTT misurato), e in giallo la media esponenziale moile.

21.1.2 Trasmettitore TCP semplificato

Iniziamo quindi a vedere come opera un **trasmettitore TCP**. Avremo che questo andrà sviluppato per *eventi*:

- Evento: dati ricevuti da applicazione.
 1. Crea un segmento con numero di sequenza pari al numero, nel bytestream, del primo byte nel segmento;
 2. Avvia il timer se non è già stato avviato. Consideriamo questo timer come un timer per il più vecchio segmento privo di ACK.

In questo TCP è più simile al *go-back-N* (vedere 10.3) che al *selective repeat*, con la differenza che ritrasmettiamo il primo segmento e non tutti i successivi per cui non si è avuto ACK. In ogni caso, c'è un solo timer in ogni momento.

Notiamo inoltre che il tempo di timeout è ottenuto dai calcoli visti nella scorsa sezione.

- Evento: timeout.
 1. Ritrasmetti il segmento che ha causato timeout (quindi il più vecchio segmento privo di ACK);

2. Riavvia il timer.
- Evento: ACK ricevuto.

In questo caso procediamo solo se l'ACK è per segmenti ancora privi di ACK.

1. Aggiorna lo stato interno impostando il segmento su cui si è fatto ACK come fornito di ACK;
2. Se ci sono ancora segmenti privi di ACK (i successivi su cui avevamo detto non si fa ACK come in *go-back-N*), riavvia il timer.

21.1.3 Ricevitore TCP semplificato

Discussiamo quindi il **ricevitore TCP**. Anche qui adottiamo un simile approccio ad *eventi*:

- Evento: arrivo di un segmento in ordine con numero di sequenza aspettato. Tutti i dati precedenti hanno già ricevuto ACK.
 1. Effettuiamo un ACK ritardato: iniziamo con aspettare 500ms;
 2. Quando i 500ms scadono, se non è stato ottenuto nessun segmento, si fa ACK.
- Evento: arrivo di un segmento in ordine con numero di sequenza aspettato. Almeno un'altro segmento ha un ACK in attesa (dall'ACK ritardato).
 - Si fa immediatamente un ACK cumulativo per gli ultimi due segmenti in ordine.
- Evento: arrivo di un segmento fuori ordine. Si rileva un buco.
 - Si fa immediatamente un **ACK duplicato**, indicando il numero di sequenza del prossimo byte aspettato.
- Evento: arrivo di un segmento che riempie parzialmente o completamente un buco.
 - Si invia immediatamente un ACK, ammesso che il segmento risieda nell'estremo inferiore del buco.

Abbiamo quindi introdotto una nuova funzionalità non ancora discussa: quella dell'**ACK duplicato**. Questo è un meccanismo che il ricevitore TCP può sfruttare per richiedere al trasmettitore una **ritrasmissione rapida**.

Sostanzialmente, abbiamo che una ritrasmissione normale avviene quando scatta il timeout su un certo segmento (che abbiamo detto essere il primo privo di ACK lato trasmettitore). In TCP si prevede anche che il ricevitore segnali esplicitamente al trasmettitore di aver perso un segmento, appunto attraverso gli ACK duplicati. Quando il trasmettitore riceve un ACK duplicato, reinvia immediatamente i dati richiesti senza aspettare il timer.

In verità, la maggior parte delle implementazioni di TCP non si aspettano propriamente ACK duplicati, ma aspettano una serie di ACK consecutivi con lo stesso numero di sequenza (nell'ordine di ~ 3) prima di effettuare una ritrasmissione.

21.2 Controllo di flusso TCP

Iniziamo quindi a vedere le basi del **controllo di flusso** in TCP. Questo non è da confondersi col *controllo di congestione*, che verrà visto in seguito.

L'obiettivo del controllo di flusso è infatti realizzare una coordinazione fra trasmettitore e ricevitore, che permetta al primo di modulare la sua velocità di trasmissione in modo che il secondo non incorra in problemi di overflow del buffer.

L'idea di fondo del controllo di flusso TCP è quello di permettere al ricevitore di "pubblicizzare" lo spazio di buffer libero. Ricordiamo che nell'header di segmento TCP (visto in 20.3.1) avevamo previsto un campo `<receive window>`. Questo può essere appunto usato per indicare la dimensione del buffer disponibile al ricevitore.

22 Lezione del 14-11-25

Riprendiamo la discussione del controllo di flusso TCP.

Avevamo introdotto come il ricevitore TCP poteva in qualche modo "pubblicizzare" lo spazio libero di buffer attraverso il campo `<receive window>` dell'header di segmento TCP. Ciò che fa il trasmettitore è quindi limitare il numero di segmenti privi di ACK (i cosiddetti segmenti *in-flight*) allo spazio pubblicizzato in `<receive window>`.

Interroghiamoci su come sono disposti i buffer di ricevitore e trasmettitore a tempo di trasmissione di una certa mole di dati.

- Vediamo innanzitutto il **ricevitore**. Ci aspetteremo di mantenere nel suo buffer i seguenti puntatori:
 - `last_byte_read`: punta all'ultimo byte letto dal livello applicazione;
 - `last_byte_acked`: punta all'ultimo byte ricevuto correttamente, su cui si è fatto ACK, non ancora letto dal livello applicazione;
 - `last_byte_recv`: punta all'ultimo byte ricevuto su cui non si è ancora fatto ACK. Notiamo che questo potrebbe trovarsi oltre `last_byte_acked` in quanto potrebbero esserci buchi nel buffer (segmenti in ordine non ancora ricevuti, per cui `last_byte_recv` punta a un segmento fuori ordine (o la fine di un loro blocco)).

Assunto che la dimensione completa del buffer è `buf_size`, ci chiediamo come quantificare lo spazio libero del buffer. Certamente i segmenti di "buco" presenti fra `last_byte_acked` e `last_byte_recv` non andranno considerati, in quanto se sono arrivati segmenti fuori ordine successivi, i segmenti precedenti sono stati trasmessi e quindi verranno ritrasmessi dal server a breve.

Calcoliamo lo spazio libero come `buf_size` meno lo spazio occupato, cioè:

```
1 receive_window = buf_size - (last_byte_recv - last_byte_read +
    buf_size) % buf_size
```

che sarà esattamente quello che inseriremo nei segmenti di ACK del ricevitore, al campo `<receive window>`.

- Vediamo quindi il **trasmettitore**. Nel suo buffer vorremo i puntatori:
 - `last_byte_acked`: punta all'ultimo byte inviato correttamente su cui si è ricevuto un ACK;

- `last_byte_sent`: punta all'ultimo byte inviato, su cui non si è ancora ricevuto ACK.

Quello che il trasmettitore dovrà fare sarà quindi mantenere il numero di byte privi di ACK al di sotto della `<receive window>` ricevuta, ergo assicurare che:

```
1 (last_byte_sent - last_byte_acked + buf_size) % buf_size <=
   receive_window
```

22.1 Controllo di congestione TCP

Vediamo quindi un altro meccanismo fornito da TCP: il **controllo di congestione**.

Una **congestione** si verifica quando le sorgenti di una rete inviano dati a una frequenza troppo alta per essere gestita dalla rete. Per le applicazioni si manifesta sotto forma di:

- Lunghi *ritardi* (dati dall'accodamento nei router);
- *Perdita* di pacchetti (dati dall'overflow nei buffer di router, cioè sempre nelle code).

22.1.1 Differenze col controllo di flusso

Potrebbe essere facile confondere *controllo di flusso* e *controllo di congestione* TCP. Ricordiamone le differenze:

- Il **controllo di flusso** si occupa di prevenire sovraccarichi al *ricevitore*, assunto che la rete stessa funzioni a pieno regime;
- Il **controllo di congestione** non si preoccupa invece del ricevitore (che assume come a capacità infinita), ma ha come obiettivo la salvaguardia della rete stessa. Questo significa che ha come obiettivo la prevenzione di sovraccarichi dell'infrastruttura di rete stessa (cioè dei router).

La soluzione per il controllo di flusso era stata ottenere esplicitamente lo spazio di buffer disponibile dal ricevitore e modulare il numero di segmenti *in-flight* su di esso. Per il controllo di congestione dovrà invece essere una limitazione complessiva della frequenza di trasmissione, coordinata con i router e fra più sorgenti all'interno della stessa rete.

22.1.2 Controllo di congestione network-assisted

Il controllo di congestione *network-assisted* (o **assistito dalla rete**) prevede che i router siano capaci di segnalare alle sorgenti che inviano dati il suo stato di congestione.

Ciò che vorremo segnalare sarà quindi il *livello* di congestione in corso, se non direttamente specificare alle sorgenti la frequenza di trasmissione desiderata.

Riguardo a *come* i router possono effettuare questo tipo di comunicazione a ritroso verso le sorgenti, possiamo fare alcune considerazioni: Avevamo detto, discutendo il segmento TCP in 20.3.1, che sono presenti due bit (C ed E) di notifica di congestione esplicita. Il router potrebbe aspettare, per un dato datagramma inviato, il successivo ACK per impostare tale bit. Era già stato appurato, però, che questa soluzione è effettivamente inutilizzata.

22.1.3 Controllo di congestione end-to-end

Vediamo quindi l'approccio alternativo al controllo di congestione, detto **end-to-end**, che viene più usato oggi.

In questo caso non prevediamo che i router ci diano alcun tipo di feedback, ma *rileviamo* la congestione agli host sulla base dei ritardi e la perdita di segmenti osservati.

Esistono diversi modi per rilevare la perdita di segmenti:

- Semplicemente, lo scatto del timeout di segmento segnala che quel segmento è stato realisticamente perso, e va reinviato;
- Come abbiamo introdotto in 21.1.3, una funzionalità del TCP è che un certo numero di *ACK duplicati* (avevamo detto 3) segnalano con molta probabilità che un segmento è stato perso.

Notiamo che la perdita dei segmenti non è direttamente legata alla congestione di rete: ad esempio i segmenti si possono perdere per malfunzionamenti a livelli inferiori. In ogni caso, i nostri obiettivo sono:

- Ottenere la *massima* frequenza di trasmissione;
- Contemporaneamente, ridurre al *minimo* le congestioni.

I due obiettivi sono chiaramente in contrasto.

22.1.4 Finestra di congestione

Vediamo quindi come le sorgenti agiscono sul traffico in uscita quando si rileva una congestione.

Quello che vorremo stabilire è una certa `congestion_window` (finestra di congestione), che ci darà una disequazione simile a quella vista per il controllo di flusso:

```
1 (last_byte_sent - last_byte_acked + buf_size) % buf_size <=
   congestion_window
```

`congestion_window` e `receive_window` potrebbero sembrarci simili. Effettivamente, il trasmettitore è limitato, in quanto a segmenti *in-flight*, da:

$$\text{window} = \min(\text{receive_window}, \text{congestion_window})$$

La differenza starà nel fatto che la `congestion_window` detterà anche la *frequenza* di trasmissione del trasmettitore. In particolare, vorremo che:

$$f = \frac{\text{congestion_window}}{\text{RTT}}$$

dovee l'RTT è il *Round Trip Time* stimato come in 21.1.1 (al tempo ci serviva per determinare il timeout), e `congestion_window` varia nel tempo sulla base della congestione di rete percepita dal trasmettitore.

Ciò che vogliamo fare è quindi, nella pratica, assicurarci di inviare al massimo `congestion_window` byte per ogni RTT.

22.1.5 Stima della finestra di congestione

Per la stima della finestra di congestione adottiamo un'approccio detto **AIMD** (*Additive Increase, Multiplicative Decrease*).

I trasmettitori in questo caso possono aumentare la frequenza di trasmissione finché non si verifica perdita di segmenti (che segnala congestione). A questo punto sono obbligati a ridurre la frequenza.

La particolarità è come si effettuano tali aumenti e riduzioni:

- L'*aumento* è **additivo**: si aggiunge una certa quantità di banda (1 MSS) ad ogni iterazione, portando ad un andamento lineare;
- La *riduzione* è invece **moltiplicativo**: si divide per 2 in evento di triplo ACK duplicato, portando a riduzioni molto più ripide degli aumenti.

Nel dettaglio, l'attività di controllo di congestione del trasmettitore dovrà svolgersi in *fasi*:

1. **Slow start**;
2. **Congestion avoidance**;
3. **Reazione ad eventi di timeout**.

La fase di *congestion avoidance* è quella vista finora. Analizziamo nel dettaglio le altre 2.

22.1.6 Slow start

Vediamo quindi la prima fase, quella di **slow start**.

All'inizio della connessione si stabilisce `congestion_window` uguale a 1 MSS, e quindi la frequenza di trasmissione:

$$f = \frac{MSS}{RTT}$$

La frequenza viene quindi aumentata esponenzialmente (raddoppiando i segmenti inviati, cioè incrementando `congestion_window` di 1 per ogni ACK ricevuto). Tale regime di operazione viene mantenuto fino a che, alternativamente:

- Si ha il primo evento di timeout o perdita segmento;
- La `congestion_window` raggiunge una certa *soglia prefissata*.

La soglia stessa viene modificata quando si verificano eventi di timeout. Vedremo poi come questo è necessario in quanto potremmo tornare nella fase di slow start in seguito.

Dalla fase di slow start si passa quindi alla fase di *congestion avoidance*, discussa nella scorsa sezione, dove si impiega l'approccio AIMD.

22.1.7 Reazione ad eventi di timeout

L'ultima fase, quella di **reazione ad eventi di timeout**, ci mostra come eventi di perdita pacchetti esplicita (triplo ACK duplicato) e eventi di timeout vengono trattati diversamente:

- Il *triplo ACK duplicato* viene gestito normalmente in fase di *congestion avoidance*, segnala perdita di segmenti e quindi richiede una divisione per due della `congestion_window`.

Nel caso il triplo ACK duplicato arrivi in fase di slow start, invece, si prende come un evento di perdita e si passa alla fase di congestion avoidance;

- Gli eventi di *timeout* vengono invece gestiti, quando in fase di *congestion avoidance*, passando nuovamente alla fase di *slow start*.

Questo spiega come mai, nella scorsa sezione, avevamo detto che la soglia della `congestion_window` viene aggiornata in sede di eventi di timeout: ogni volta che rientriamo nella fase di slow start vogliamo uscirne prima, con una finestra di dimensione massima più piccola.

Nella fase di slow start, l'evento di timeout è sempre visto come evento di perdita e quindi comporta transizione a fase di *congestion avoidance*.

22.1.8 Fast recovery

Esiste una quarta fase, non ancora vista, che esiste in qualche modo in maniera ancillare alla fase di *congestion avoidance*. Questa è la fase di **fast recovery**, dove si giunge in caso di triplo ACK duplicato.

Ciò che accade in fast recovery è che si aumenta `congestion_window` di 1 MSS per ogni ACK duplicato ricevuto sul segmento che a causato l'entrata in fase di fast recovery.

Questo meccanismo è opzionale per TCP, ed è stato introdotto in **TCP Reno**. Le versioni precedenti (dette **TCP Tahoe**) preferivano entrare in fase di slow start a partire sia da un evento di triplo ACK duplicato che di timeout.

22.1.9 TCP Cubic

Un'ulteriore evoluzione di **TCP Reno** è data da **TCP Cubic**.

Senza perdersi nei dettagli,abbiamo che TCP Cubic varia solamente la fase di *congestion avoidance*: in particolare prevediamo che la fase di aumento non sia lineare, ma *cubica*, quindi più veloce per aumentare il throughput.

TCP Cubic è usato di default ad esempio nei sistemi Linux.

22.1.10 Fairness TCP

Uno degli obiettivi di TCP è la **fairness**: nel caso di n host che condividono un link di bitrate R , vorremo che ogni host avesse un throughput pari a:

$$R_i = \frac{R}{n}$$

Immaginiamo allora l'esempio di due sessioni TCP in competizione, che sfruttano l'approccio AIMD:

- L'aumento additivo dà pendenza 1, in quanto è lineare;

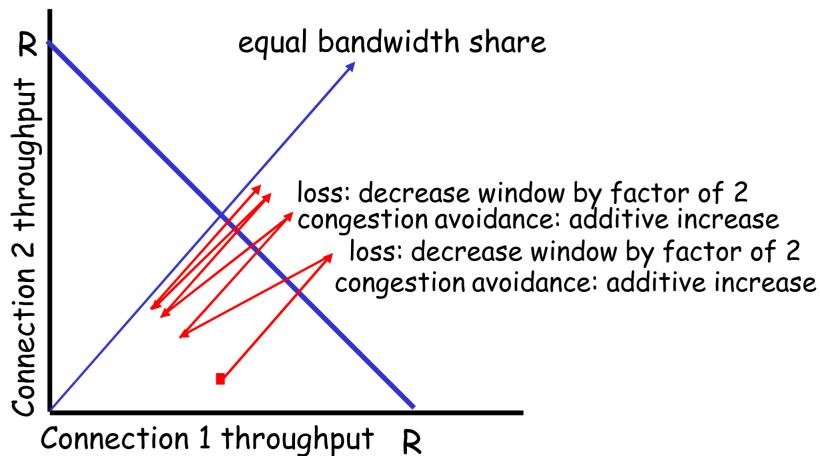
- La riduzione moltiplicativa riduce invece il throughput in maniera proporzionale.

Questa configurazione favorisce la fairness, in quanto:

1. Inizialmente, entrambe le sessioni cercano di ottenere nuova banda in maniera lineare, cioè aumentare le loro frequenze di trasmissione R_1 e R_2 ;
2. Prima o poi la frequenza di trasmissione cumulativa $R_1 + R_2$ supera R , per cui si verificano eventi di perdita. In questo caso, per AIMD, si ha una riduzione proporzionale della banda: questo significa che la sessione che aveva più banda perde *più* banda dell'altra sessione;
3. Questo processo si ripete in maniera iterativa ogni volta che le due sessioni superano in frequenza cumulativa il bitrate massimo R .

Il risultato è che si ha uno spostamento delle frequenze di trasmissione, sul lungo termine, verso la divisione equa della banda (in quanto chi invia di più ha effetti più significativi in fase di eventi di perdita).

Possiamo spiegare cosa succede usando il seguente grafico:



dove la linea piena indica la barriera di limite massimo ($R_1 + R_2 \leq R$) del bitrate, e la linea tratteggiata indica dove le sessioni si trovano in competizione equa. Come vediamo, il percorso seguito dalle frequenze di trasmissione si avvicina, iterativamente, verso quest'ultima linea.

Chiaramente questo discorso è valido solo in un mondo *ideale*: non ci aspettiamo che nella realtà tutti gli host rispettino il controllo di congestione TCP.

- I trasmettitori UDP, infatti, non hanno tale meccanismo e usano il rate che preferiscono in ogni momento. Per questo motivo si preferisce usare UDP per applicazioni ad altra prestazione come videogiochi o streaming video.
- Un altro modo per "barare" su TCP può essere quello di stabilire *più* connessioni TCP parallele: se vogliamo che ognuna di queste sia fair, aprendo m connessioni TCP si ha m volta la banda che ci meritiamo. Questo è ciò che fanno i browser: aprono diverse connessioni TCP con lo stesso server per ottenere contemporaneamente più dati della stessa pagina. Visto che tale approccio può essere effettivamente considerato poco *fair*, molti amministratori server limitano il numero di connessioni parallele che un singolo client può aprire.

22.2 Protocollo QUIC

Esite un protocollo nato per risolvere *anche* i problemi di fairness visti nell'ultima sezione, e per offrire un'evoluzione a TCP. Di recente si preferisce infatti usare il protocollo **QUIC** (*Quick UDP Internet Connection*) su HTTP/3.

Questo è un protocollo di trasporto, ma implementato a livello application sopra UDP. I vantaggi di QUIC sono che:

- Riduce il numero di handshake necessari, dai 2 di TCP su TLS a solo 1;
- Supporta nativamente le connessioni parallele, evitando le problematiche di fairness introdotte dalle connessioni parallele TCP.