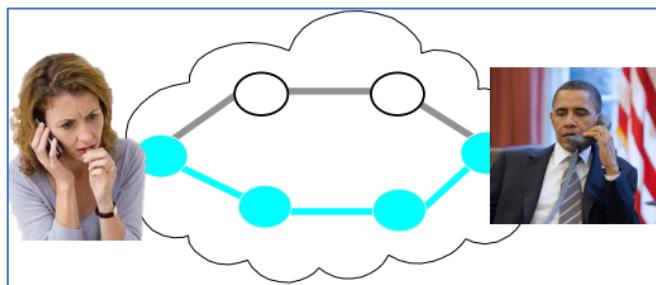


# RETI di CALCOLATORI

## 1) CONCETTI GENERALI

L'ITU-T (International Telecommunication Union) definisce COMUNICAZIONE = il trasferimento di informazioni secondo convenzioni, e **TELECOMUNICAZIONE** = trasmissione/ricezione di segnali che rappresentano informazioni. Definendo **SERVIZIO** = offerto da un gestore ai clienti per un'esigenza di telecomunicazione, vediamo le **FUNZIONI** (in una rete) = operazioni svolte nella rete al fine di offrire servizi, tra cui:

- **SEGNALAZIONE** (di UTENTE o di RETE) → segnalare alla rete di voler comunicare, ovvero trasferimento delle informazioni di controllo tra utente e rete (es. alzare la cornetta del telefono) [apertura, controllo, chiusura e gestione delle connessioni della rete];
- **COMMUTAZIONE** → individuare le risorse necessarie per connettere 2 utenti per il tempo necessario al trasferimento dei segnali, mediante unità funzionali (terminali [telefoni], centrali di smistamento [nodi di commutazione]), canali di trasmissione e circuiti di telecomunicazione;
- **TRASMISSIONE** → i 2 utenti possono comunicare, ovvero trasferimento di segnali da 1 punto ad 1 o più punti;
- **GESTIONE** → allacciamento di nuovi utenti (modifica e aggiunta di apparati e/o canali), sostituzione di vecchi apparati con moderni, gestione guasti, monitoraggio, controllo.



Parlando della **TOPOLOGIA DELLE RETI** (disposizione di insieme di nodi [di commutazione] e segmenti [canali] che fornisce un collegamento tra 2 o più punti) abbiamo:

- **MEZZO TRASMISSIVO** (come "via Lessona") = mezzo fisico per trasportare segnali tra 2 o più punti;
- **CANALE** (come la strada per andare da casa mia da Ga) = insieme di 1 o più mezzi trasmittivi concatenati.

Altre definizioni:

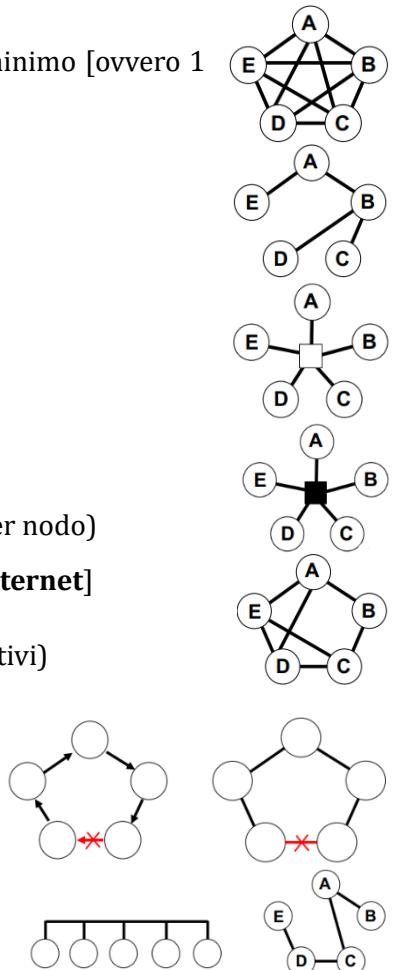
- **BANDA** = bit al secondo (Bit Rate [bit/s]);
- **CAPACITÀ** = per il singolo mezzo trasmittivo è la massima banda (massima velocità trasmittiva [bit/s]), mentre per il canale è la capacità del mezzo più lento al suo interno ("bottleneck");
- **TRAFFICO**:
  - o **OFFERTO** = bit/s che una sorgente cerca di inviare in rete;
  - o **SMALTITO (throughput)** = parte di traffico offerto [bit/s] che riesce ad essere consegnata alla destinazione [dunque evidente che throughput  $\leq$  capacità del canale e throughput  $\leq$  traffico offerto].

⚠ Ritornando sulla topologia delle reti, abbiamo **3 tipi di canali**:

- **PUNTO-PUNTO** = 2 nodi collegati agli estremi del canale, usato in modo paritetico (entrambi i versi);
- **MULTI-PUNTO** = 1 nodo centrale e tanti nodi periferici;
- **BROADCAST** = unico canale condiviso da tutti i nodi (informazione inviata da un nodo è ricevuta da tutti gli altri). Se i dati trasmessi contengono l'indirizzo del nodo destinazione, realizzo un punto-punto;

E vediamo che possiamo descrivere la topologia di una rete con un **GRAFO G = (V, A)** con **V** = insieme dei **nodi** e **A** = insieme degli archi (**canali**); gli archi possono essere orientati (unidirezionali) e non orientati (bidirezionali); definiamo **N = |V| = n° nodi**, e **C = |A| = n° canali** (archi). Ci sono diverse **TOPOLOGIE DI RETE**:

- **A MAGLIA COMPLETA**  $\rightarrow C = N(N - 1)/2$ 
  - **Pro:** tollerante ai guasti (molti percorsi alternativi, ma 1 solo percorso minimo [ovvero 1 solo canale])
  - **Contro:** tanti canali (usata solo con pochi nodi)
- **AD ALBERO**  $\rightarrow C = N - 1$ 
  - **Pro:** pochi canali (riduco i costi e semplifico i canali)
  - **Contro:** vulnerabile ai guasti (solo 1 percorso tra 2 nodi)
- **A STELLA ATTIVA**  $\rightarrow C = N$ 
  - **Pro:** pochi canali e gestione dei percorsi fatta dal centro stella
  - **Contro:** vulnerabile ai guasti al centro stella (solo 1 percorso tra 2 nodi)
- **A STELLA PASSIVA**  $\rightarrow C = 1$  ( $N$  fili che si comportano come 1 solo canale)
  - **Pro:** pochi canali (unico canale [broadcast])
  - **Contro:** vulnerabile ai guasti del centro stella (solo 1 percorso possibile per nodo)
- **A MAGLIA SEMPLICE**  $\rightarrow N - 1 < C < N(N - 1)/2$  [topologia più usata, es. **Internet**]
  - **Pro:** tolleranza ai guasti e numero di canali selezionabile
  - **Contro:** instradamento [ricerca di percorsi] complesso (+ percorsi alternativi)
- **AD ANELLO**  $\rightarrow C = N/2$  (**unidirezionale**) e  $C = N$  (**bidirezionale**)
  - **Pro:** 1 o 2 percorsi possibili per ogni coppia di nodi
  - **Contro:** solo l'anello bidirezionale assicura la sopravvivenza della rete in caso di guasto (ma a capacità dimezzata) [⚠ l'anello bidirezionale è la + semplice topologia che consente instradamento alternativo per i guasti]
- **A BUS**  $\rightarrow C = N - 1$  (bus attivo) e  $C = 1$  (bus passivo)
  - **Contro:** 1 solo percorso tra 2 nodi (obsoleto ormai)



⚠ Tutto ciò sempre distinguendo tra topologia **fisica** (tiene conto del percorso dei mezzi trasmissivi) e **logica** (definisce solo l'interconnessione tra nodi mediante i canali)!

A livello di **SERVIZI**, le reti possono essere **DEDICATE** (unico servizio) e **INTEGRATE** (più servizi) [ci sono servizi portanti o teleservizi]; ci sono **2 MODELLI** di accesso ai servizi:

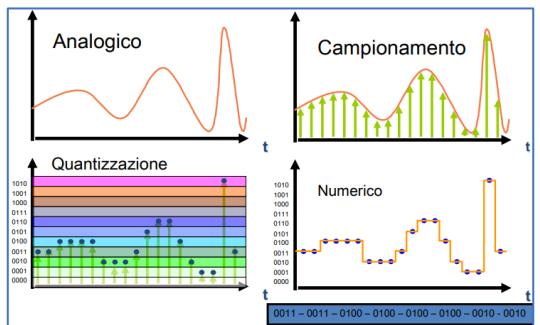
- **MODELLO CLIENT-SERVER**  $\rightarrow$  2 ruoli ben distinti: **CLIENT** (inizializza l'interazione con il server richiedendo un servizio) e **SERVER** (fornisce al client il servizio, dunque informazione contenuta nel server) [app Web]. I server vengono attivati all'accensione del computer su cui sono installati, sono sempre disponibili e in attesa delle richieste dei client; i client invece vengono attivati nel momento in cui l'utente richiede un servizio e si disattivano una volta terminato il servizio.
- **MODELLO PEER-TO-PEER**  $\rightarrow$  tutti gli applicativi sono **paritetici** (informazioni condivise) [telefono, torrent];

Come abbiamo già visto in diversi corsi, il segnale analogico (continuo nel tempo e in ampiezza) deve essere **convertito** in **digitale** (o numerico) (discreto nel tempo [**CAMPIONAMENTO**] e in ampiezza [**QUANTIZZAZIONE**]) per essere trasmesso. Ci sono 2 tipi di **TRASMISSIONE**:

- **PARALLELA** = 1 bit alla volta su ogni cavo (+ cavi);
- **SERIALE** = serializzo l'informazione e trasmetto 1 bit alla volta.

Il problema in entrambe è la **sincronizzazione** (durata deve essere nota sia al trasmettitore che al ricevitore); abbiamo trasmissioni:

- **ASINCRONA** = ogni byte è trasmesso separatamente dagli altri, usando dei segnali di temporizzazione (ad ogni ricezione occorre recuperare il sincronismo);
- **SINCRONA** = le informazioni da trasmettere sono organizzate in "trame". Trasmettitore e ricevitore sincronizzano i loro clock prima della trasmissione e li mantengono sincronizzati durante la trama.



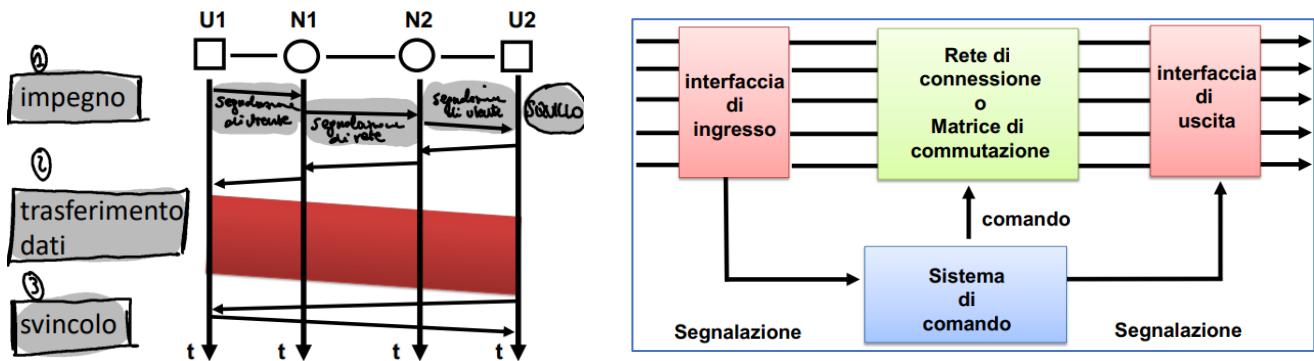
Nelle reti abbiamo **2 modi di trasferimento**:

## 1. CONDIVISIONE DI CANALE (solo strato 1):

- A. **MULTIPLAZIONE** = tutti i flussi di informazione sono disponibili in unico punto:
  - I. **DI FREQUENZA** (FDM-FDMA) → separazione ottenuta usando bande di frequenza diverse [es. radio e televisione]. Ci sono delle bande di guardia per evitare l'invasione non voluta di altre frequenze;
  - II. **DI TEMPO** (TDM-TDMA) → separazione mediante intervalli di tempo diversi [es. interrompere qualcuno che parla per parlare noi]. Ci sono tempi di guardia per attutire il fenomeno dei ritardi di trasmissione dovuti alle diverse distanze tra trasmettitore e ricevitori;
  - III. **DI CODICE** (CDM-CDMA) → separazione mediante usando codifiche diverse [es. persone che si parlano sopra, ma se sono di lingue diverse non si sovrappongono]. Garantisce resistenza a disturbi, ma servono codici riconoscibili;
  - IV. **DI SPAZIO** → sfruttare lo spazio del sistema per far coesistere più flussi di informazione in punti diversi [es. 2 oratori che parlano in 2 piazze diverse];
  - V. **STATISTICA** → assegnare una stessa frequenza, tempo, codice o spazio a 2 o più sorgenti intermittenti in contemporanea perché probabilisticamente queste non trasmettono insieme;
- B. **ACCESSO MULTIPLO** = i flussi di informazione accedono al canale da punti differenti;

## 2. CONDIVISIONE DI NODO (ricordando commutazione = processo di allocazione delle risorse necessarie all'interconnessione e al trasferimento delle informazioni):

- a. **COMMUTAZIONE DI CIRCUITO** = se i flussi sono continui [es. telefonia]. La rete alloca un intero **CIRCUITO** (collegamento fisico tra 2 terminali di utenti) ad ogni richiesta di servizio; il circuito è esclusivo dei 2 utenti per tutta la durata della comunicazione ("svincolato" solo una volta finita la comunicazione). Sotto a sinistra si vede anche la **struttura del nodo a commutazione di circuito** (ricordando che nel sistema di comando ci sarà anche l'algoritmo per capire qual è il prossimo nodo da toccare per arrivare all'utente U2):



**Pro:**

- banda costante garantita (ovvero velocità di trasferimento costante)
- ritardi di trasferimento costanti
- bassi ritardi nell'attraversamento dei nodi
- circuito "trasparente"

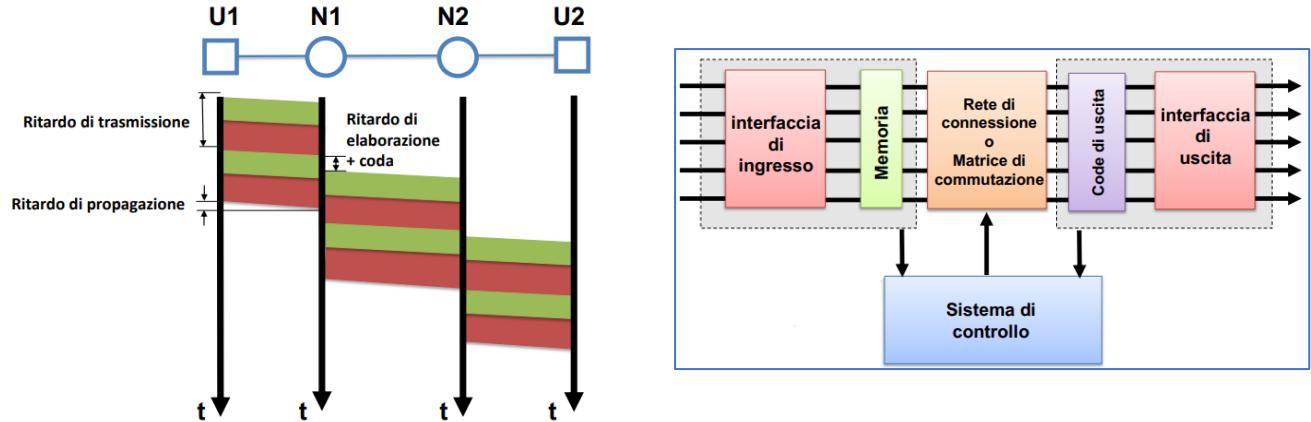
**Contro:**

- efficienza buona solo per sorgenti continue
- tempo di apertura del circuito
- nessuna possibilità di convertire i dati scambiati
- tariffazione in base al tempo di esistenza del circuito

- b. **COMMUTAZIONE DI PACCHETTO** = se i flussi sono intermittenti [es. trasmissione di file]. Il funzionamento è analogo alle "poste". L'informazione da trasferire è organizzata in **PDU** (Protocol Data Unit, che chiamiamo **PACCHETTO**, unità dati o cella) con al loro interno **PCI** (header o parte di **CONTROLLO**) e **SDU** (**DATI** da inviare, l'informazione che l'utente vuole inviare).



Ogni nodo di commutazione di pacchetto della rete fa lo “**STORE AND FORWARD**” memorizza il pacchetto, lo elabora, determina su quale canale inoltrarlo e lo mette in coda per la trasmissione su quel canale (una sorta di buffer); questo processo serve per disporre l'intestazione prima di fare l'instradamento, introdurre dei bit di protezione dagli errori (sempre sull'intestazione) e gestire le diverse capacità dei mezzi trasmissivi (che richiedono trasmissioni a bit rate diversi). Sotto a sinistra si vede anche la struttura del nodo a commutazione di pacchetto, che svolge funzione di memorizzazione sia all'entrata sia all'uscita (sia mista).



L'informazione di un utente può essere troppo grande e quindi dover essere **frazionata in pacchetti** (di dimensione fissa o variabile) [come i rar troppo grandi da Telegram]. Definiamo quindi (dalla figura qui sopra) i **ritardi** dovuti al:

- **CANALE:**
  - o **RITARDO DI TRASMISSIONE** = tempo che trasmettitore impiega per mandare i segnali di tutto il pacchetto [dipende dalla dimensione in bit del pacchetto e dalla velocità del canale];
  - o **RITARDO DI PROPAGAZIONE** = tempo che ha impiegato il segnale per andare da sorgente a destinazione (uguale per ogni bit di pacchetto, ovvero costante nel canale) [dipende dalla dimensione in metri del canale];
- **NODO:**
  - o **RITARDO DI ELABORAZIONE** [dipende dalla velocità di instradamento e di controlli sul pacchetto];
  - o **RITARDO DI ACCODAMENTO** = ritardo dovuto alla coda [dipende dal traffico sulla coda].

Alcune **formule** utili:

- Dimensione **P** di un pacchetto [Byte];
- Velocità di rete [bit/s];
- Ritardo di trasmissione di un pacchetto  $T_{TX} = P/V_{TX}$  con  $V_{TX}$  = velocità trasmissione sul canale;
- Dimensione in metri di un pacchetto su un canale **M** = velocità luce  $\cdot T_{TX}$
- **HOP** = transito da un nodo di commutazione al successivo (ovvero l'arco tra 2 nodi)

Per quanto riguarda la **LUNGHEZZA DEI PACCHETTI**, essa è determinata da alcuni fattori:

- i. possibilità di **PARALLELIZZAZIONE (PIPELINE)** = rende più veloce l'invio parallelo dei pacchetti (rispetto ad un unico grosso file), riducendo la latenza di attraversamento dei nodi (che si può ulteriormente ridurre aumentando la velocità di trasmissione) [quindi **pacchetti brevi**];
- ii. **RITARDO DI PACCHETTIZZAZIONE** = **pacchetti brevi** riducono questo ritardo;
- iii. **PERCENTUALE DI INFORMAZIONE DI CONTROLLO** (o frazione di informazione di controllo) = se ho **pacchetti lunghi** servono meno informazioni di controllo perché non devo riscriverle per ogni singolo pacchetto;
- iv. **PROBABILITÀ DI ERRORE** = **pacchetti corti** riducono la probabilità di errore (in quanto anche solo l'errore su 1 singolo bit invalida il pacchetto, quindi più grosso è il pacchetto, maggiore sarà l'informazione invalidata in presenza di errore) [ $P_{\text{Pacchetto corretto}} = (1 - p)^n$  con  $n$  = numero di bit per pacchetto, e  $p$  = probabilità di errore sul singolo bit].

**Pro:**

- o utilizzo efficiente delle risorse con sorgenti intermittenti

- controllo degli errori lungo il percorso
- possibilità di conversioni
- tariffazione in base al traffico trasmesso

**Contro:**

- non garanzie di banda (throughput)
- elaborazione di ogni pacchetto in ogni nodo
- ritardo di trasferimento variabile

⚠ Sempre nella commutazione di pacchetto, ci sono **2 modi di trasferimento**:

- **DATAGRAM** = quella vista finora, dove i nodi trattano ogni pacchetto in modo indipendente (infatti pacchetti con uguale sorgente e destinazione possono seguire percorsi diversi);
- **CIRCUITO VIRTUALE** = si avvicina alla commutazione di circuito, ma è diverso perché non si allocano le risorse per una singola comunicazione (per questo "virtuale"). La comunicazione è divisa in 3 fasi, ovvero apertura connessione [segnalazione] – trasferimento dati – chiusura [segnalazione]. Qui pacchetti con uguale sorgente e destinazione seguono lo stesso percorso (in questo modo bypass il dover applicare l'algoritmo di instradamento ad ogni pacchetto con lo stesso percorso). Inoltre nel datagram si identifica in ogni pacchetto la coppia sorgente/destinazione con identificatori globali, mentre nel circuito virtuale si identifica il circuito con identificatori locali (etichette [non si usa sempre la stessa]) [sorgente/destinazione usata per l'apertura del circuito].

Per quanto riguarda la **SEGNALAZIONE** si distinguono segnalazioni di utente (scambio di informazioni tra utente e nodo) e segnalazioni internodali (di rete); la segnalazione può essere:

1. ASSOCIATA AL CANALE = corrispondenza biunivoca tra canale controllante (informazioni di segnalazione) e canale controllato (informazioni di utente). Ce ne sono 2 tipi:
  - a. **IN BANDA** → canale controllante e controllato coincidono, ma sono usati in tempi diversi con TDM (multiplazione di tempo) [telefonia mobile];
  - b. **FUORI BANDA** → canale controllante e controllato sono distinti.
2. A CANALE COMUNE = 1 canale di segnalazione (che funziona a pacchetto) controlla più canali di informazioni di utente [telefonia fissa].

⚠ Per il progetto di una rete si deve definire come le informazioni sono emesse dalle sorgenti; tempo fa, per il progetto si dovevano sapere le richieste di servizio e la qualità del servizio per determinare le risorse necessarie!

Per quanto riguarda le **SORGENTI DI INFORMAZIONE** abbiamo:

- ANALOGICHE [voce, video, ...] → caratterizzate dalle loro caratteristiche spettrali;
- NUMERICHE/NUMERIZZATE [dati, voce numerizzata, etc...] → caratterizzate dalla velocità di cifra e dalla loro impulsività. Ce ne sono 2 tipi:
  - **CBR (Constant Bit Rate** = velocità costante) [telefonia, videoconferenze, ...] → caratterizzate dalla velocità, durata, processo di generazione delle chiamate;
  - **VBR (Variable Bit Rate** = velocità variabile) [file transfer, video streaming, ...] → caratterizzate dalla velocità di picco, velocità media, durata, processi di generazione delle chiamate.

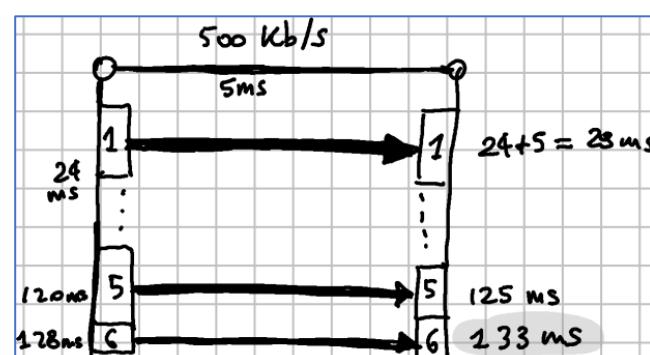
⚠ Dunque diversi tipi di informazione richiedono alla rete prestazioni diverse (quindi diversi indici di qualità, tra cui ritardo, velocità, probabilità di errore, probabilità di perdita, probabilità di blocco)!

### ESERCIZIO – Tempo di trasmissione di file

File da 8000B; pacchetti da 1500B (max) con trasmissione da 500kb/s e ritardo di 5ms sul canale:

$$n^{\circ} \text{ pacchetti} = \frac{8000}{1500} = 5 \text{ con resto di } 500B \rightarrow$$

$$\rightarrow \begin{cases} 5 \text{ pacchetti da } 1500B \rightarrow T_{TX,1} = \frac{1500 \cdot 8}{500 \cdot 10^3} = 24ms \\ 1 \text{ pacchetto da } 500B \rightarrow T_{TX,2} = \frac{500 \cdot 8}{500 \cdot 10^3} = 8ms \end{cases}$$



Dunque **T<sub>Totale trasmissione</sub>** = **133ms**.

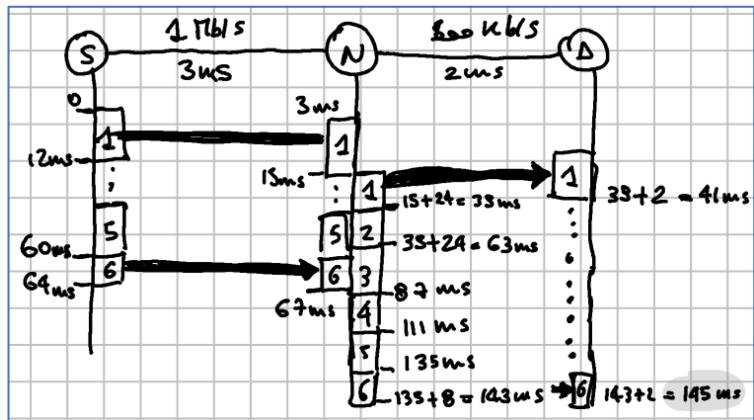
⚠ Avessimo avuto un nodo intermedio nel canale (con tecnica store and forward), avremmo ottenuto 157ms perché bisognerebbe aggiungere anche altri 24ms (perché ad ogni nodo intermedio si aggiunge una tappa e quindi si aggiunge il ritardo  $T_{TX}$  del 1° pacchetto)!

### ESERCIZIO – Tempo di trasmissione di file con canali con velocità diverse

File da 8000B; pacchetti da 1500B (max) con 1Mb/s e ritardo di 3ms sul 1° canale, e 500kb/s e 2ms sul 2° canale:

$$n^{\circ} \text{ pacchetti} = \frac{8000}{1500} = 5 \text{ con resto di } 500B \rightarrow$$

$$\rightarrow \begin{cases} 5 \text{ pacchetti da } 1500B \rightarrow \begin{cases} T_{TX,1}^1 = 12ms \\ T_{TX,1}^2 = 24ms \end{cases} \\ 1 \text{ pacchetto da } 500B \rightarrow \begin{cases} T_{TX,2}^1 = 4ms \\ T_{TX,2}^2 = 8ms \end{cases} \end{cases}$$



Dunque  $T_{\text{Totale trasmessione}} = 145ms$ .

⚠ Qui infatti dato che il 1° canale è più veloce del 2° si verificano degli accodamenti (**bottleneck**)!

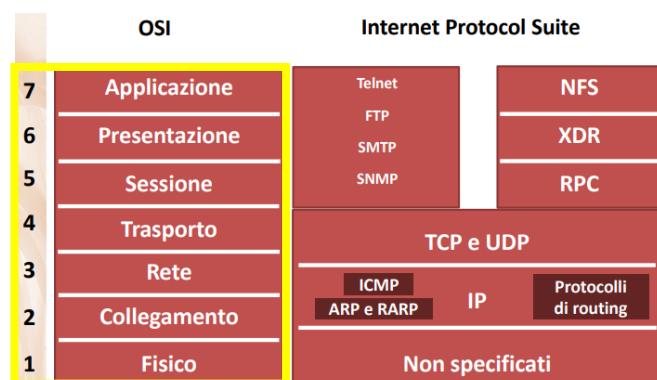
⚠ Se ci avessero detto che 100B ad ogni pacchetto sono l'intestazione, allora 8000B andavano spacchettati in pacchetti da 1400B (+100B di intestazione = 1500B) e quindi l'ultimo pacchetto sarebbe stato da 1000B (+100B di intestazione = 1100B).

⚠ Se ho un buffer di dimensione 3 e il pacchetto 4 arriva prima che il pacchetto 1 esca, allora il pacchetto 4 viene perso (non viene memorizzato nella coda)!

## 2) ARCHITETTURE e PROTOCOLLI DI RETE

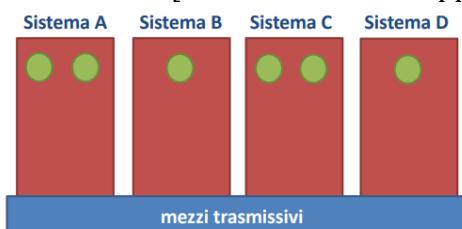
Le regole che definiscono l'interazione tra elementi dello stesso livello gerarchico di una rete si chiamano **PROTOCOLLI**; la gerarchia tra i protocolli definisce l'**ARCHITETTURA DI RETE**. Un protocollo prevede lo scambio di messaggi definendone tipologia (richieste o risposte), sintassi (struttura dei messaggi), semantica (significato di campi di bit nei messaggi) e temporizzazione.

Nelle reti si usano **ARCHITETTURE STRATIFICATE** per semplicità di progetto, gestione e standardizzazione e per la separazione di funzioni (bisogna garantire la struttura gerarchica). L'architettura di rete definisce il processo di comunicazione, le relazioni tra le entità coinvolte nella comunicazione, le funzioni necessarie per la comunicazione (es. conversione in/da bit) e l'organizzazione delle funzioni. Il modello di riferimento usato nelle reti è il modello **OSI** (Open System Interconnection) [importante conoscere il nome e numero degli stati].



Una rete è composta da **SISTEMI** (terminali, nodi, ...) collegati tra loro da **MEZZI TRASMISSIVI**, dove ogni sistema avrà dei processi applicativi (app) che dovranno comunicare tra loro attraverso i mezzi trasmissivi.

Ogni sistema è composto da **STRATI**; ogni strato realizza le proprie funzioni tramite le sue ENTITÀ (lo strato più elevato è più vicino all'utente [es. interfaccia di un app], mentre quello più basso è più vicino ai mezzi trasmissivi).



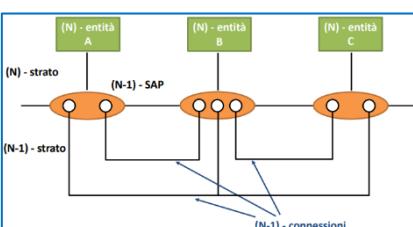
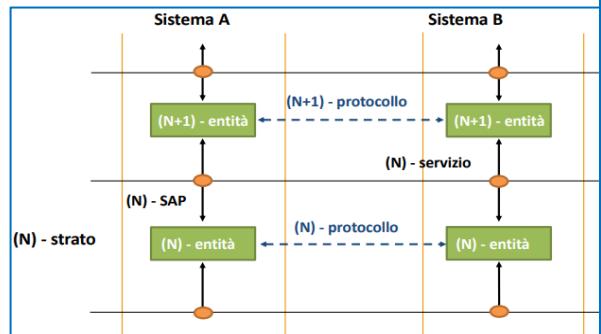
Le **ENTITÀ** sono gli elementi attivi dello strato (o sottosistema), cioè svolgono le funzioni di strato (interagiscono nello strato). Stratificazione della rete significa che ogni strato riceve servizi dallo strato inferiore e li usa con le proprie funzioni per migliorare/integrare il servizio ricevuto dallo strato inferiore, per poi passarlo allo strato superiore.

Per quanto riguarda il **SERVIZIO**, identifichiamo i **fornitori** di servizio, gli **utenti** di servizio e i **punti di accesso** al servizio (Service Access Point [SAP]). Un servizio può essere:

- **connection-oriented [CO]** = si stabilisce un accordo preliminare tra rete e interlocutori [es. login di un servizio in banca, dove serve sapere chi è l'interlocutore], come il circuito virtuale;
- **connectionless [CL]** = i dati vengono trattati in modo indipendente [es. guardare il meteo online, dove non serve loggarsi e sapere chi siamo], come il datagram.

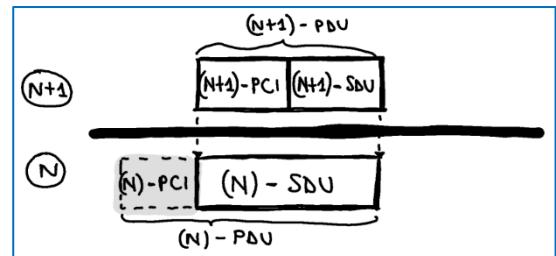
La **stratificazione** funziona come la **compartimentazione**, ovvero ogni strato percepisce gli strati inferiori come delle "**black box**" (ovvero usa il servizio fornito dagli strati inferiori, ma non sa né come sono fatti né quanti sono gli strati inferiori). Dunque abbiamo la connessione tra:

- **entità superiore (N+1) e inferiore (N) dello stesso sistema** [quindi ricezione del servizio] → **(N)-SAP**;
- **entità dello stesso strato (N) di sistemi diversi** → **(N)-protocollo**.



Una **CONNESSIONE** è una relazione esistente tra SAP dello stesso strato di **sistemi diversi** per lo scambio di dati tra interfacce.

In un sistema a strati, ogni strato (N) ha il proprio **N-PDU**, dove ha i dati dell'utente (**N-SDU**) e aggiunge le proprie informazioni di controllo (**N-PCI**, ovvero l'intestazione). Ogni strato inferiore tratta la PDU dello strato superiore come una "busta chiusa" a cui aggiungere solo un'intestazione (**INCAPSULAMENTO**) [ai dati vengono aggiunti tante intestazioni quanti sono gli strati attraversati nel sistema], mentre in ricezione avviene il processo inverso [come sbucciare la cipolla].

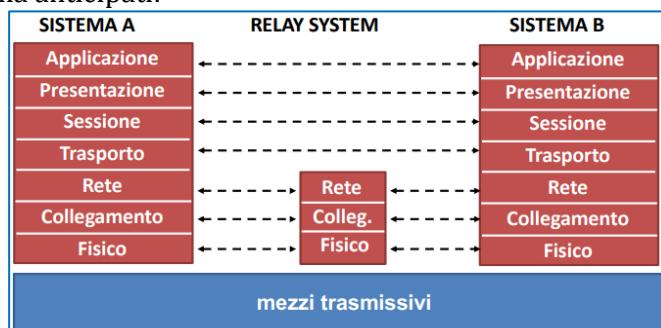


Sulle PDU esiste la possibilità di **SEGMENTAZIONE** e **CONCATENAZIONE**; queste possono avvenire in 2 modi:

- costruendo **più (N)-PDU** da **1 (N)-SDU**;
- costruendo **più (N-1)-SDU** da **1 (N)-PDU**.

⚠ Segmentare è **sconveniente** perché comporta overhead (ovvero più intestazioni inutili) e necessità di ri-assemblare in caso di ricezione; inoltre se perdo un pezzo, perdo tutto!

Vediamo ora i **7 STRATI** prima anticipati:



1. **STRATO FISICO** (physical layer) = **trasferimento dei bit** scambiati dalle entità di strato di collegamento [**PDU= bit o simboli**];
2. **STRATO DI COLLEGAMENTO** (data link layer) = **delimitazione dei PDU**, rilevazione e recupero degli **errori** di trasmissione, **controllo di flusso** [**PDU = trama o frame**]
3. **STRATO DI RETE** (network layer) = **instradamento, tariffazione** (e **controllo di flusso**) [**PDU = datagram**];

4. **STRATO DI TRASPORTO** (transport layer) = colma le carenze di qualità di servizio dello strato di rete, controlla errori, sequenze, flusso, esegue multiplazione/demultiplazione di connessioni (⚠ diversa dalla multiplazione vista in precedenza, perché qui multiplazione = mettere insieme informazioni da applicazioni diverse e mandarle in rete segnalando l'applicazione di provenienza) esegue la segmentazione dei dati in pacchetti (e la loro ricomposizione) [PDU = segmenti];
7. **APPLICAZIONE** (application layer) = fornisce ai processi applicativi i mezzi per accedere all'ambiente OSI (esempio di servizi: trasferimento di file, terminale virtuale, posta elettronica). Questo strato racchiude STRATO DI SESSIONE (session layer) e DI PRESENTAZIONE (presentation layer).

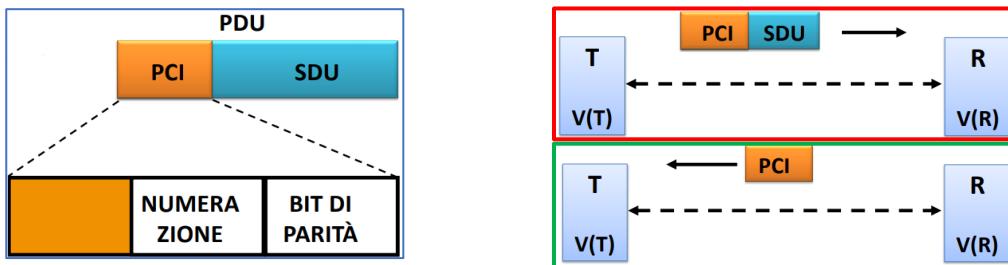
### 3) PROTOCOLLI A FINESTRA

I **PROTOCOLLI A FINESTRA** compaiono praticamente in tutte le reti; il loro scopo è **recupero di errori di trasmissione**, controllo del flusso di trasmissione e controllo della sequenza (infatti la trasmissione di bit su un canale non è mai esente da errori, perciò questi vanno rilevati e corretti [o recuperati]) [esempi di protezione sono i **bit di parità** e i codici di ripetizione]. Questi **bit di parità** vengono inseriti nell'intestazione del pacchetto (PCI).



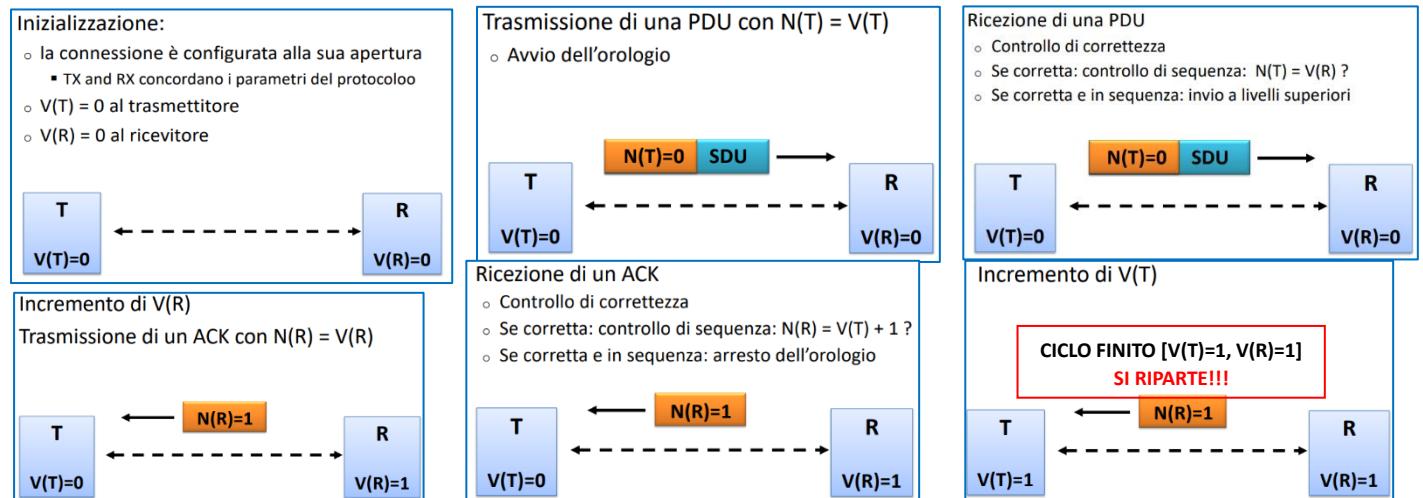
⚠ Se i **bit di parità** di un pacchetto sono errati, il **pacchetto viene scartato** (come se non fosse mai esistito)!

A seconda della quantità di bit di parità ci sono tecniche diverse: noi useremo **pochi bit** per cercare di rilevare gli errori in ricezione e permettere la **ritrasmissione** della PDU, ovvero la tecnica **ARQ** [Automatic Repeat reQuest], dove abbiamo nella PCI sia i **bit di parità** sia **bit di numerazione** sia gli **indirizzi**. Nelle immagini sotto vediamo che ci sono **2 PCI** (intestazioni): quella del **pacchetto dati** [che il **trasmettitore** manda al **ricevitore**] e quella del **pacchetto di riscontro (ACK o acknowledge)** [che viene rimandata dal **ricevitore** al **trasmettitore**].

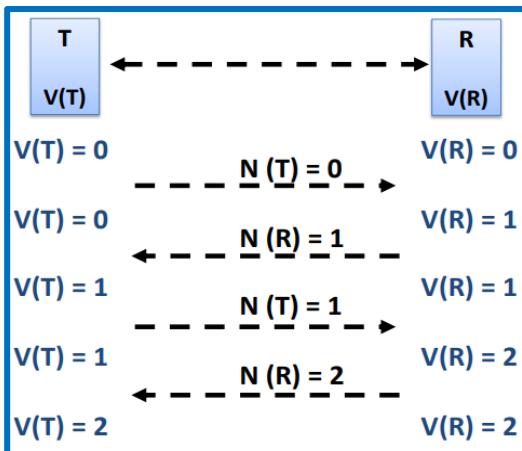


Ci sono **3 tecniche ARQ**:

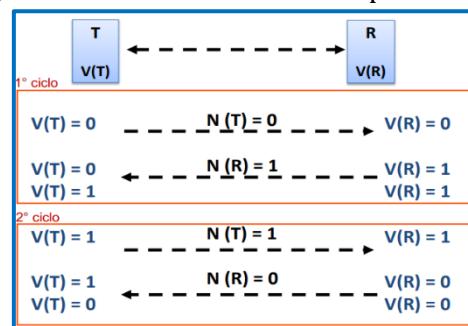
1. **STOP AND WAIT** ( $W_T = 1, W_R = 1$ ) → il **TRASMETTITORE** fa una **copia** del PDU (nel caso di ritrasmissione), invia la PDU, attiva un **timeout** (orologio) ponendosi **in attesa della conferma di ricezione (ACK)** e **se scade** il timeout prima dell'arrivo dell'ACK, **ripete la trasmissione riattivando il timeout**. Una volta ricevuto l'ACK, il trasmettitore controlla la correttezza dell'ACK, controlla il numero di sequenza e, se l'ACK è relativo all'ultima PDU trasmessa, ferma il timeout e manda la prossima PDU (altrimenti l'ACK è scartato). Il **RICEVITORE** controlla la **correttezza della PDU** e, se corretta, **controlla il numero di sequenza e invia l'ACK**; se la PDU è quella attesa, la sua **SDU** viene inviata ai **livelli superiori**. Visivamente:



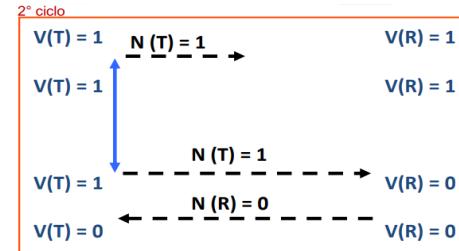
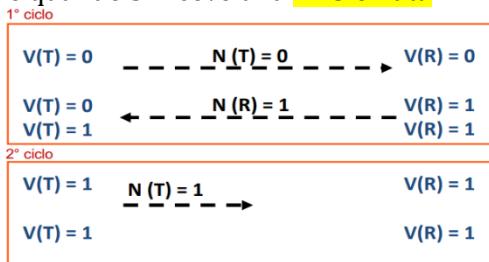
Visivamente:



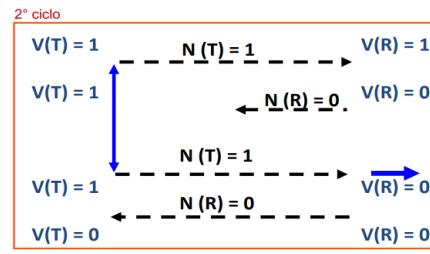
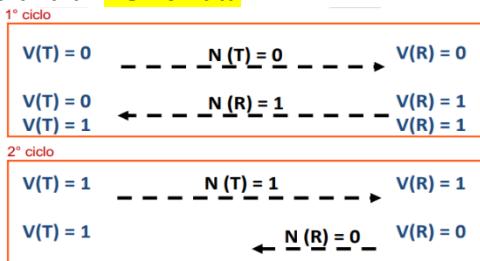
→ Se invece usassi lo STOP AND WAIT con **ALTERNATING BIT PROTOCOL**, avrei che i contatori non vengono incrementati ad ogni ciclo, ma lo 0 diventa 1 e al ciclo dopo l'1 diventa 0 (binario):



Cosa succede quando si riceve una PDU errata?

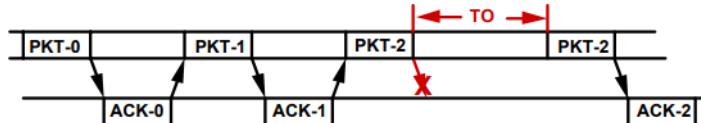


Quando si ha una un ACK errata?



Se abbiamo dei **CANALI NON SEQUENZIALI** non possiamo verificare malfunzionamenti (dunque possono verificarsi perdita di PDU o duplicazioni di PDU); spesso, se l'ACK è troppo lento, potremmo avere i contatori V(T) e V(R) disallineati. Si potrebbe usare una **numerazione modulo 4** (ovvero numeri da 0 a 3), ma anche qui il protocollo si blocca quando i due contatori sono disallineati (servono meccanismi per ripartire). Non c'è una soluzione al canale non sequenziale, ma si può ridurre la possibilità di malfunzionamento usando **più bit per la numerazione** o fissare un **tempo di vita massimo** per le PDU e gli ACK (TTL = contatore di quanti link sono attraversati dal pacchetto [quando è a zero, il pacchetto viene ammazzato]).

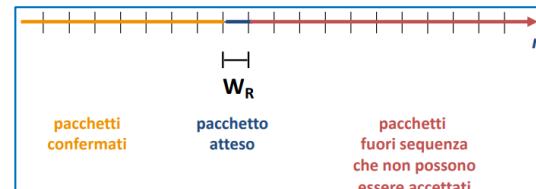
Lo STOP AND WAIT è **poco efficiente** perché bisogna aspettare sempre l'ACK.



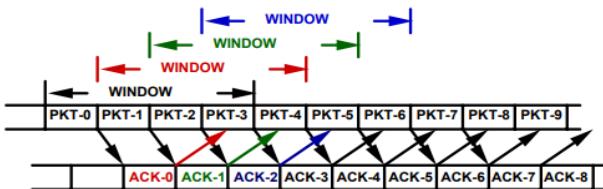
⚠ **FINESTRA DI TRASMISSIONE ( $W_T$ )** = quantità massima di PDU in sequenza che il trasmettitore è autorizzato ad inviare in rete senza averne ricevuto riscontro (ACK). Può anche essere vista come numero max di PDU inviate da uno stesso trasmettitore contemporaneamente presenti sul canale o in rete (dimensione limitata dalla quantità di memoria allocata in trasmissione).



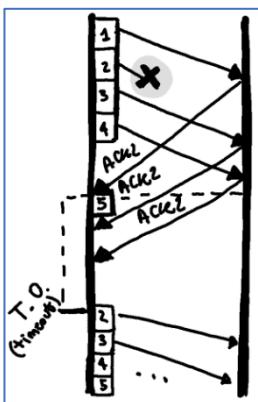
⚠ **FINESTRA DI RICEZIONE ( $W_R$ )** = sequenza di PDU che il ricevitore è disposto ad accettare e memorizzare (dimensione limitata dalla quantità di memoria allocata in ricezione). Finestra di ricezione unitaria (Go-back-N):



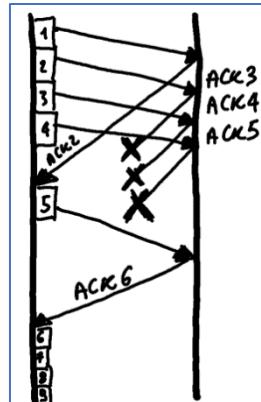
2. **GO BACK N** ( $W_T > 1, W_R = 1$ ) → il **TRASMETTITORE** con finestra  $N(>1)$  invia fino a  $N (= W_T)$  PDU, facendo di ognuna una **copia**, attiva **1 solo timeout** per le  $N$  PDU (resetto ad ogni trasmissione di PDU), **attende gli ACK**; per ogni ACK non duplicato ricevuto, fa scorrere in avanti la finestra di tanti pacchetti quanti sono i pacchetti confermati. Se scade il timeout prima della conferma di ricezione della PDU che ha settato il timeout, si ripete la trasmissione di tutte le PDU non ancora confermate. Il **RICEVITORE** invece si comporta come quello della **STOP AND WAIT** (controlla la correttezza della PDU e se corretta, controlla il numero di sequenza e invia l'ACK; se la PDU contiene il 1° numero di sequenza non ancora ricevuto, la sua SDU viene inviata ai livelli superiori) [1 **CICLO** = tempo intercorso tra l'invio della 1<sup>a</sup> PDU e la ricezione dell'ultimo ACK].



Quando si perde una PDU?



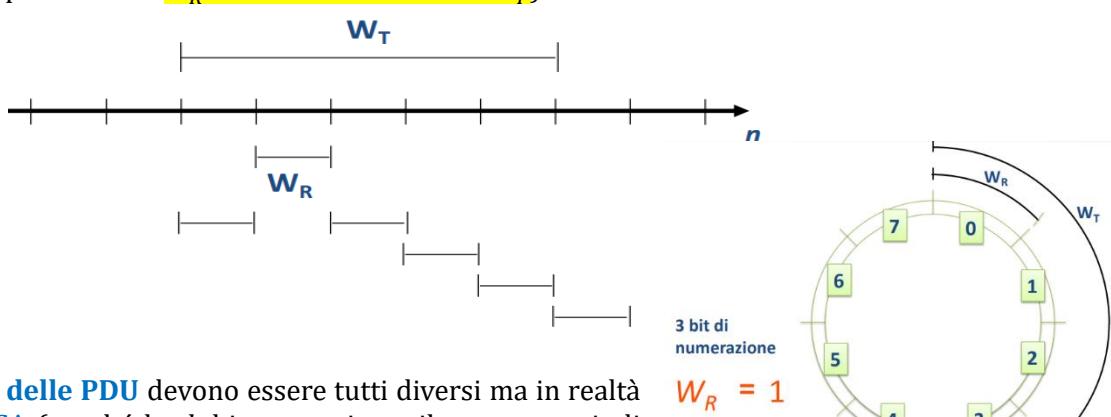
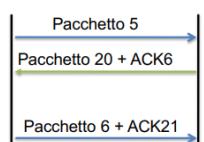
Quando si perdono degli ACK?



⚠ Gli **ACK** possono avere **semantica diversa** perciò trasmittitore e ricevitore si devono accordare; 3 tipi:

- **ACK CUMULATIVO** → notifica la corretta ricezione di tutti i pacchetti con numero di sequenza < di quello dell'ACK [es. ACK(n) significa aver ricevuto tutto in sequenza fino alla PDU(n) esclusa] (**noi useremo solo questo**);
- **ACK SELETTIVO** (o individuale) → notifica la corretta ricezione di un pacchetto specifico [es. ACK(n) significa aver ricevuto la PDU(n)];
- **ACK NEGATIVO** (NAK) → notifica la richiesta di ritrasmissione di un pacchetto specifico [es. NAK(n) significa "ritrasmetti la PDU(N)"].

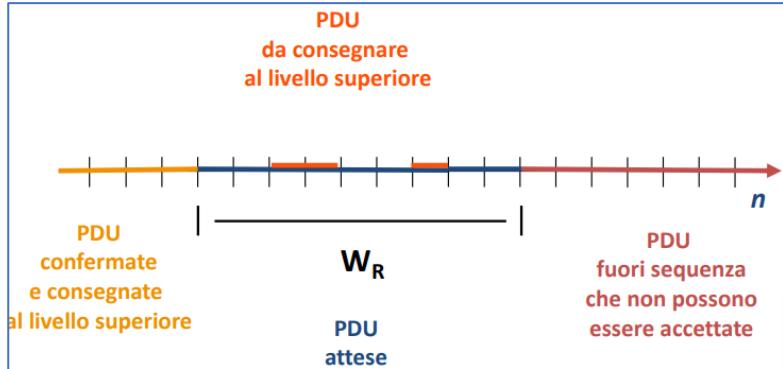
⚠ Nel caso di flussi di informazione **bidirezionali**, si può scrivere l'ACK nell'intestazione di PDU che viaggiano in direzione opposta (permette risparmio di ACK) [**PIGGYBACKING**]. Inoltre,  $W_T$  e  $W_R$  devono trovarsi solo in posizione tale che la finestra del ricevitore sia uguale a quella del trasmittitore +1 posto (specialmente  $W_R$  deve essere incluso in  $W_T$ ):



⚠ Idealmente i **numeri delle PDU** devono essere tutti diversi ma in realtà la numerazione è **CICLICA** (perché ho  $k$  bit per scrivere il numero, quindi  $W_T < 2^k$  e analogamente  $W_T + W_R \leq 2^k$ ).

Riepilogando, rispetto allo Stop and Wait, nel **Go Back N**, il **trasmettitore** è più **complesso**, mentre il **ricevitore** rimane **inalterato**; si possono qui usare le **conferme multiple** (ovvero ACK su gruppi di PDU) grazie all'ACK cumulativo (con **timeout sul ricevitore** e non sul trasmettitore) e si ha il limite sulle dimensioni della finestra  $W_T < 2^k$ .

3. **SELECTIVE REPEAT** ( $W_T > 1, W_R > 1$  [di solito uguali]) → qui il **RICEVITORE** tiene in memoria le PDU, ovvero può accettare PDU corrette ma fuori sequenza (più prestazioni). Noi vediamo il caso con ACK cumulativi e timeout associati alla finestra. Quindi il **ricevitore invia gli ACK per segnalare le PDU fuori sequenza di cui ha bisogno**.

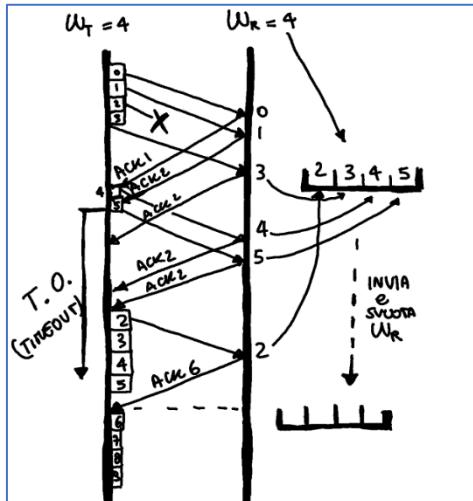


Quindi il **TRASMETTITORE** invia fino a  $N = W_T$  PDU, facendo di ognuna una copia e attiva un solo timeout (orologio) per le  $N$  PDU (resetto ad ogni trasmissione di PDU). Si pone poi in attesa degli ACK e per ogni ACK non duplicato ricevuto, fa scorrere in avanti la finestra di tante PDU quante sono quelle confermate (fino a qui come Go Back N); se invece scade il timeout, ripete la trasmissione di tutte le PDU non ancora confermate.

Il **RICEVITORE** riceve 1 PDU, controlla la correttezza, controlla il numero di sequenza e se è tutto ok, consegna la sua **SDU al livello superiore** (eventualmente insieme ad altre SDU ricevute). Se la PDU invece è corretta ma **non in sequenza**, se è entro la **finestra di ricezione** la **memorizza**, altrimenti la **scarta**. Invia comunque un ACK relativo all'ultima PDU ricevuta in sequenza.

⚠ Ricorda che  $R_{TT}$  = tempo di invio pacchetto e ritorno dell'ACK relativo, mentre  $T_{TX}$  = tempo di trasmissione (ritardo di trasmissione). La scrittura  $T_{TX} \cdot W_T \gg R_{TT}$  dice che l'ACK della PDU mi torna indietro molto prima che la finestra sia stata completamente trasmessa (condizione per avere più PDU mandate e ACK ritornati nella stessa finestra).

Nell'**esempio** vediamo che si perde PDU2; dunque PDU3 viene memorizzato nella finestra del ricevitore ( $W_R$ ) e viene inviato ACK2 fino a che non viene inviata la PDU2 (infatti vengono salvati dal ricevitore anche PDU4 e PDU5 nel frattempo). Quando arriva PDU2, viene svuotata la  $W_R$  e inviata al livello superiore; viene inviato ACK6 perché quelli prima li ho già inviati.



⚠ In caso di perdita singola, il Selective Repeat si comporta come il Go Back N; come abbiamo già accennato prima, si hanno vantaggi rispetto al Go Back N se  $R_{TT} < T_{TX} \cdot W_T$  (tempo trasmissione della finestra). Modificando il comportamento del trasmettitore (vincolandolo a ritrasmettere solo il 1° pacchetto perso nella finestra) si riduce l'occupazione del canale (si recupera 1 pacchetto perso ogni RTT). Con gli ACK selettivi si hanno migliori prestazioni.

⚠ Se al ricevitore arriva **PDU duplicata**, la causa è il **timeout del trasmettitore troppo breve!**

⚠ Anche qui vale la relazione  $W_T + W_R \leq 2^k$  (che se non rispettata ha come conseguenza pacchetti erroneamente accettati 2 volte e pacchetti erroneamente scartati!).

Parlando di **EFFICIENZA** (quanto bene sto usando il canale) dei protocolli appena visti abbiamo:

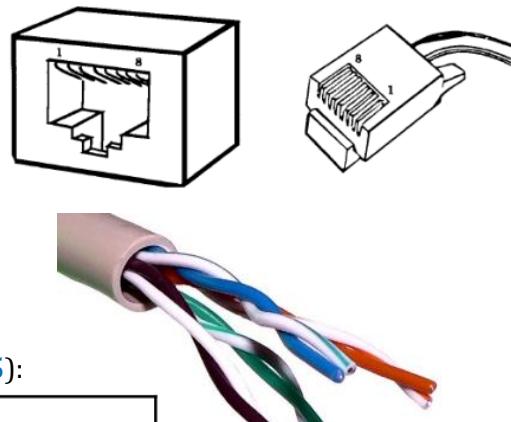
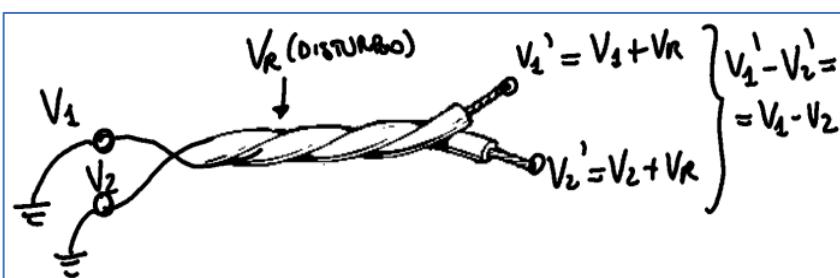
- Stop&Wait ideale (senza errori) →  $\eta = \frac{T_{TX}}{T_{TX} + 2t_p} = \frac{1}{1+2a} < 1$  (poco efficiente) con  $t_p$  = propagazione e  $a = \frac{t_p}{T_{TX}}$
- GoBackN ideale (senza errori) →  $\eta = \begin{cases} 1 & W_T \geq 1 + 2a \text{ [ottimo]} \\ \frac{W_T}{1+2a} & W_T < 1 + 2a \end{cases}$
- Throughput su canale a capacità C →  $\theta = \eta C \propto \frac{W_T}{a}$  (maggiore tempo di propagazione, minore è il throughput; maggiore è la finestra di trasmissione, maggiore è il throughput; maggiore è il RTT (Round Trip Time), maggiore è il throughput).

## 4) STRATO FISICO (1°)

Il **MEZZO TRASMISSIVO ottimale** è caratterizzato da **resistenza, capacità parassite e impedenza basse, resistenza alla trazione, flessibilità, facilità di collegamento** dei ricetrasmettitori; i **MEZZI ELETTRICI** presentano tutte queste caratteristiche, che dipendono dalla **geometria, dal numero di conduttori** (e distanza reciproca), dal tipo di **isolante e di schermatura**. Definiamo **PARAMETRI DI MERITO** dei mezzi elettrici:

- **impedenza** (in funzione della frequenza);
- **velocità di propagazione** del segnale (0.5c-0.7c per i cavi e 0.6c per le fibre ottiche);
- **attenuazione** (riduzione della potenza; cresce con la distanza e con la radice quadrata della frequenza);
- **diafonia** [o cross-talk] (misura del disturbo indotto da cavo vicino, cresce con la distanza fino a stabilizzarsi).

Il mezzo trasmissivo classico della telefonia è il **DOPPINO** (o coppia, "pair", **CAVO ETHERNET**), composto da 2 fili di rame ritorti (binati, "twisted") per ridurre le interferenze elettromagnetiche; ci sono 2 fili perché il segnale viene inviato come **differenza delle tensioni dei 2 fili** ( $V_1 - V_2$ ).



⚠ Esistono diverse categorie di doppino (anche detto **connettore RJ45**):

1	Telefonia analogica
2	Telefonia ISDN
3	Reti locali fino a 10 Mb/s
4	Reti locali fino a 16 Mb/s
5	Reti locali fino a 100 Mb/s e 1Gb/s
5e	Reti locali fino a 1 Gb/s e 2.5 Gb/s
6,6a	Reti locali fino a 10 Gb/s
7,7a	Reti ibride (dati/TV) fino a 1Gb/s. Reti locali fino a 10Gb/s
8	Reti locali fino a 40 Gb/s

Altro mezzo trasmissivo è il **CAVO COASSIALE** (usato maggiormente in passato), composto da un connettore centrale e 1 o più calze di schermo (maggiora **schermatura**, ma maggiori costi, difficoltà di installazione e meno velocità).



Altro mezzo trasmissivo (ma non elettrico, bensì **OTTICO**) è la **FIBRA OTTICA**, ovvero un minuscolo e flessibile filo di vetro costituito da 2 parti, **CORE** e **CLADDING**, con **indici di rifrazione diversi**; in questo modo, per la legge di Snell, il raggio luminoso introdotto nella fibra entro un certo "angolo di accettazione" si propaga confinato nel core, sbattendo nelle pareti di core e cladding.

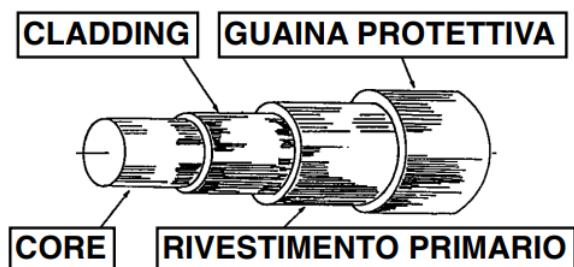
**Pro:**

- Totale **immunità da disturbi elettromagnetici** (in quanto trasmetto luce, non segnale elettrico);
- Alta **capacità trasmissiva** (Tbit/s);
- **Bassa attenuazione;**
- **Dimensione e costi contenuti.**

**Contro:**

- Solo collegamenti punto-punto;
- Difficili i collegamenti;
- Poca curvatura e soffre vibrazioni.

⚠ Vengono posati sul fondo del mare (interrati)!



Altro mezzo è il **CANALE RADIO** ("etero"), ovvero la trasmissione mediante le **antenne**; se almeno uno tra trasmettitore (TX) e ricevitore (RX) è in movimento, si parla di canale **RADIOMOBILE**. La qualità della

trasmissione, e quindi il bitrate [banda] raggiungibile, dipende dall'[Equazione di Friis](#) (o [PROPAGAZIONE NELLO SPAZIO LIBERO](#), ovvero [propagazione ideale senza ostacoli](#)):

$$\frac{P_{Ricevuta}}{P_{Trasmessa}} = G_T G_R \frac{\lambda^2}{(4\pi D)^2} \quad \text{con} \quad \begin{cases} \lambda = \text{lunghezza d'onda} \\ G_T = \text{guadagno antenna trasmissione} \\ G_R = \text{guadagno antenna ricezione} \\ 4\pi D = \text{distanza tra TX e RX} \end{cases}$$

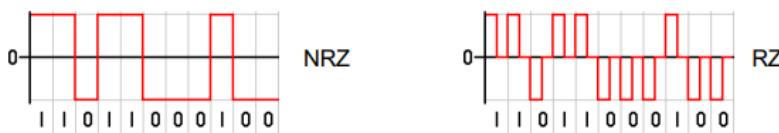
⚠ Dato che però la propagazione non è ideale, la [potenza ricevuta reale](#) è soggetta ad [attenuazione](#) dovuta a fenomeni atmosferici, interferenze, ostacoli; specialmente gli [ostacoli](#) comportano riflessioni e rifrazioni, che producono [copie del segnale trasmesso](#) (disturbando la ricezione e generando [Fading](#) [variazione veloce del segnale dovuta all'azione delle copie ricevute da percorsi diversi] e [Shadowing](#) [variazione lenta dell'ampiezza del segnale]).

La **TRASMISSIONE A DISTANZA** si realizza associando [bit](#) o [simboli](#) di informazione a segnali diversi; si usano [tecniche diverse](#) a seconda del mezzo fisico usato:

- [CODIFICHE DI LINEA](#) (mezzi elettrici, ottici) = rappresentazione di informazioni digitali con segnali digitali;
- [MODULAZIONI DIGITALI](#) (mezzi radio, elettrici, ottici) = rappresentazione di informazioni digitali con segnali analogici.

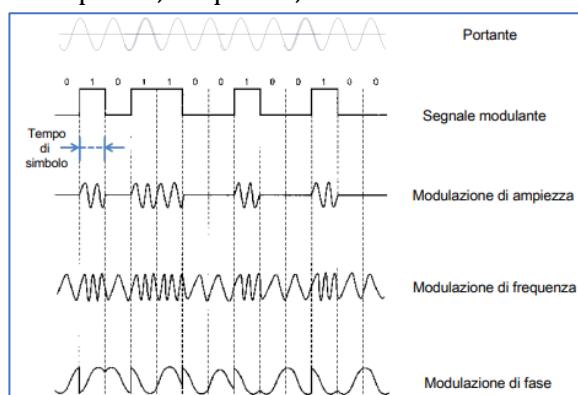
Parlando delle **CODIFICHE DI LINEA**, a seconda del [riferimento di tensione del segnale](#), abbiamo codifiche:

- **UNIPOLARI** → semplici (primitive); usano [tensione nulla = 0](#) e [tensione positiva = 1](#). [Poco usata](#) a causa dei [filtrati](#), [perdita di sincronismo](#) (se trasmetto lunghe sequenze dello stesso simbolo), [sovraffollamento](#) del LED di trasmissione nei mezzi ottici (con lunghe sequenze di 1 [luce]);
- **POLARI** → [2 livelli di tensione con polarità diverse](#) (si quantizza/discretizza maggiormente [meno componente continua in ampiezza]); ci sono 3 varianti:
  - **NRZ** (Non-Return-to-Zero) = non transizione su tensione nulla nel passaggio tra 2 bit consecutivi;
  - **RZ** (Return-to-Zero) = transizione su tensione nulla nel passaggio tra 2 bit consecutivi;
  - **Bifase** = ogni bit rappresentato da 2 livelli di tensione di polarità inversa (più usata nei mezzi elettrici).



- **BIPOLARI** → si usa [tensione nulla = 0](#) e [2 polarità opposte per l'1](#) (usate alternativamente, da cui la codifica è detta [AMI](#) [Alternate Mark Inversion]). Queste codifiche permettono l'uso di [simboli ternari \[-1, 0, 1\]](#), come nella codifica [8B6T](#) (8 bit codificati con 6 simboli ternari, ovvero  $2^8 = 256$  sequenze codificate con  $3^6 = 729$  sequenze, da cui [posso scegliere le migliori codifiche](#) scartando le altre [dunque [non si legge il singolo bit](#), ma si usa un [dizionario](#) in cui ho le sequenze associate alle migliori sequenze di codifica]);
- **nBmB** → codifiche in cui [simboli di n bit](#) sono [rappresentati da m bit](#) (con  $n < m$ ). Sono molto usate perché [richiedono meno banda](#) delle codifiche polari, [permettono il controllo](#) sulla scelta delle parole di codice ([limitando quelle con troppi 0 o 1 consecutivi](#)), limita la componente continua, fornisce [caratteri speciali](#) per la delimitazione dei pacchetti.

Per quanto riguarda le **MODULAZIONI DIGITALI** si usano per i [segnali ondulatori](#) (perché non si può usare la codifica di linea con le variazioni/impulsi); l'informazione del [segnaletica portante](#) ("carrier") è legata alle variazioni di ampiezza, frequenza, fase o le loro combinazioni ([variazioni codificate in un segnale modulante](#)).



Abbiamo quindi le modulazioni digitali:

- **ASK** (Amplitude Shift Keying) = modulazione di [ampiezza](#);
- **FSK** (Frequency Shift Keying) = modulazione di [frequenza](#);
- **PSK** (Phase Shift Keying) = modulazione di [fase](#);
- **QAM** (Quadrature Amplitude Modulation) = [ampiezza+fase](#).

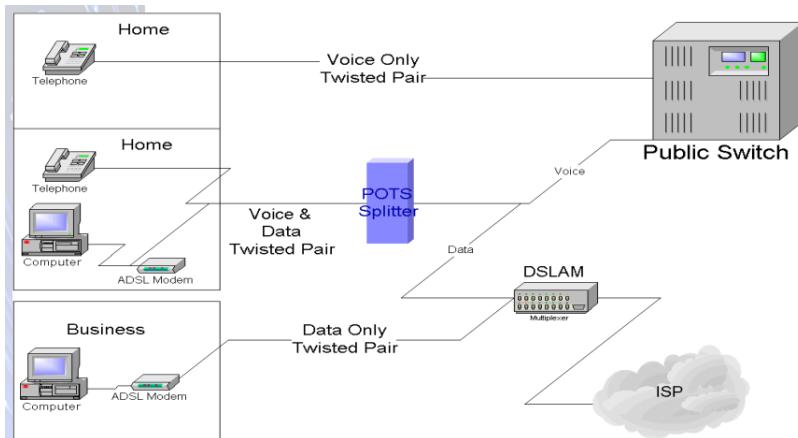
Oltre alla sigla troviamo il [numero di simboli del segnale modulante](#) (es. 8-PSK = modulazione di fase a 8 simboli [ogni simbolo rappresenta 3 bit]).

⚠ Modulazioni con tanti simboli (maggiori bitrate) si possono usare solo se il canale non distorce eccessivamente il segnale ricevuto).

Un utente si collega ad una rete sfruttando la **RETE DI ACCESSO** (apparati e mezzi trasmissivi che **collegano l'utente con il nodo di accesso** del gestore di servizi TLC) e la **RETE DI TRASPORTO** (apparati e mezzi trasmissivi appartenenti ad 1 o più gestori di servizi di TLC, destinati al **transito di dati tra 2 nodi di accesso**).

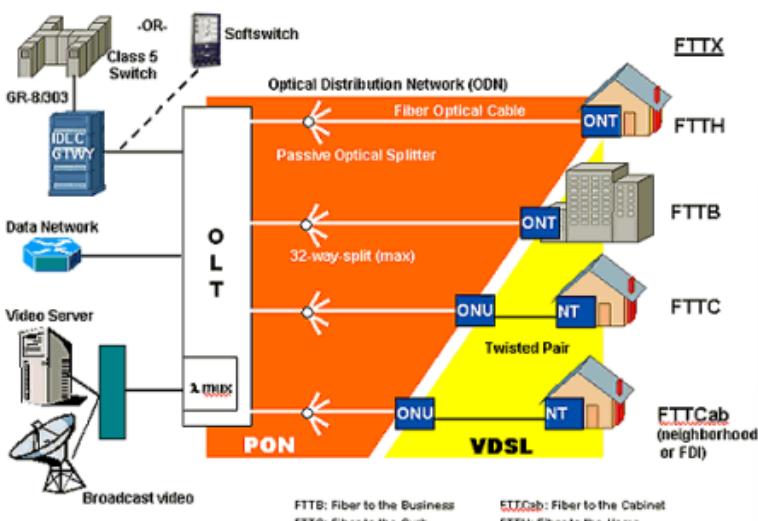
Parlando delle **RETI DI ACCESSO** (di tipo residenziale, mobile o istituzionale) è l'**ultima tratta della rete per arrivare all'utenza** residenziale (in inglese "local loop"); le principali tecnologie nelle reti di accesso sono:

- **DSL** (Digital Subscriber Line) → fornisce servizio dati ad alta velocità; la 1<sup>a</sup> diffusa è stata l'**ADSL** (Asymmetric DSL) [+ veloce in **DOWNSTREAM** (transito dei dati dalla rete all'utente) che in **UPSTREAM** (transito dei dati dall'utente alla rete)]. Per l'accesso alla rete si usa un **MODEM** (Modulatore e Demodulatore). La DSL fa **trasmissioni dei dati usando i cavi della rete telefonica pubblica** [quindi il bitrate dipende anche dal cavo e dalla distanza dall'armadio di strada ("cabinet")].



Infatti lo **SPLITTER** (filtro) effettua la **MULTIPLAZIONE IN FREQUENZA** per separare voce ( $f_{alta}$ ), dati ( $f_{bassa}$ ) in downstream (più banda) e dati in upstream (meno banda); il **DLSAM** multipla poi le comunicazioni verso il gestore (Internet Service Provider, ISP).

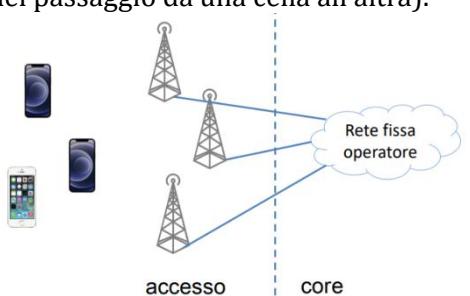
- **PON (Passive Optical Networks)** → usa la **fibra ottica** (quindi non elettrica); è composta da **OLT** (Optical Line Terminator, in centrale), **ONU** (Optical Network Units, le cabine/armadi di strada), **ONT** (Optical Network Terminals, a casa dell'utente) e **ODN** (Optical Distribution Network).



Ci sono 2 tipi di PON, ovvero **GPON** (Gigabit PON) e **EPON** (Gigabit Ethernet PON, ovvero si differenzia per il framing [organizzazione dei bit in trame] che segue la tradizionale rete ethernet). Entrambe combinano la **MULTIPLAZIONE IN LUNGHEZZA D'ONDA** (cioè **FREQUENZA**, ma basata sulle diverse lunghezze d'onda della luce [colori]) con la **MULTIPLAZIONE DI TEMPO** (se non bastano le lunghezze d'onda).

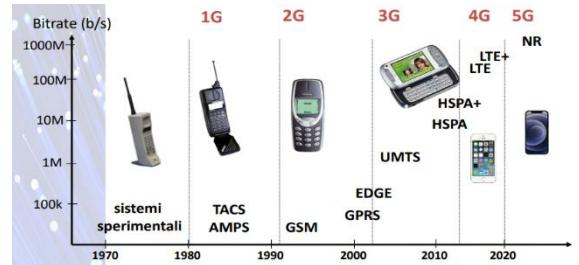
- **Reti cellulari** → offrono servizio voce/dati ad utenti residenziali o in mobilità. La **mobilità** è dovuta al fatto che la rete cellulare è composta da tantissime "**CELLE**" (**CELLA** = singola area coperta dall'**ANTENNA**, ovvero dove il segnale di questa è predominante sui segnali delle altre antenne); quindi è basata su una copertura capillare (95% del territorio) usando antenne di piccola portata (+ piccola, + veloce il bitrate, + costo per l'operatore perché ne servono di più). Questa rete è basata sul concetto di **ROAMING** (rintracciabilità dell'utente sul territorio) e **HANDOVER** (continuità della connessione nel passaggio da una cella all'altra).

La struttura della rete cellulare è basata su una **rete di accesso di tipo RADIO** (con le antenne) e la **rete di tipo CORE** (operatore).



Ci sono state varie **GENERAZIONI** della rete cellulare:

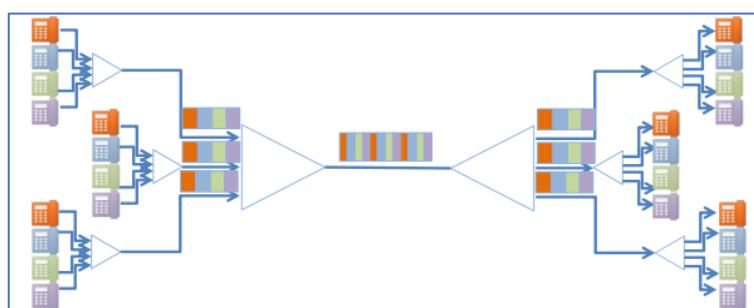
- 1G → solo analogico (voce modula la portante);
- 2G (GSM) → solo telefonica (+sms), oggi ancora i nostri telefoni hanno un chip GSM per gli sms (+ EDGE ovvero la "E" di quando prende male);
- 3G → introduce lo scambio di dati (scomparsa);
- 4G (LTE) → banda larga sui cellulari;
- 5G (NR) → banda elevata e ritardo bassissimo.



- **Reti Satellitari** → comunicazione unilaterale (i satelliti inviano a terra, ma noi non inviamo nulla ai satelliti); ci sono 3 tipologie di orbite:
  - **GEO** = per trasmissioni broadcast e servizi dati;
  - **MEO** = per GPS;
  - **LEO** = per telefonìa satellitare e servizi dati a bassa latenza.

La **RETE DI TRASPORTO** crea l'interconnessione tra reti di accesso; i suoi nodi sono commutatori telefonici e dati (**router**, strato 3) in una **topologia gerarchica a maglia** e i suoi canali sono in **fibra ottica**. È gestita da più operatori telefonici o dati in competizione. La trasmissione è interamente digitale, si è evoluta dalla rete telefonica tradizionale e si è strutturata con la **MULTIPLAZIONE DI TEMPO GERARCHICA** (meno cavi a bitrate elevatissimi per più comunicazioni divise in tempo); gerarchica perchè è come una "matrioska", ovvero ho più cavi che vengono di volta in volta raggruppati in singoli canali (per poi essere nuovamente riscomposti in uscita).

La rete di trasporto di oggi è basata su **gerarchie sincrone (SONET, SDH e STS)**.

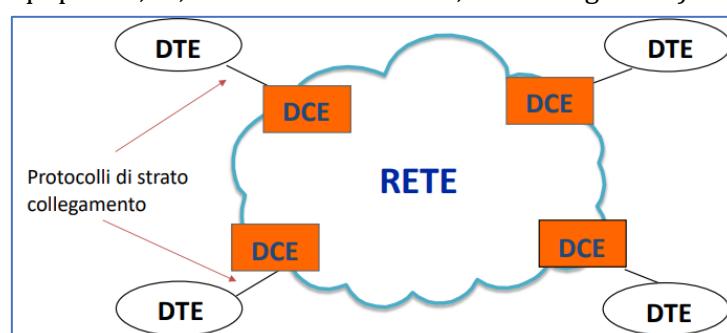


## 5) STRATO COLLEGAMENTO (2°)

Le funzioni principali dello **STRATO COLLEGAMENTO** sono:

- **Delimitazione della TRAMA** (ovvero la PDU dello strato collegamento) [con delimitatori esplicativi, indicatori di lunghezza, lunghezza fissa o silenzi tra pacchetti];
- **Multiplazione di protocollo**;
- **Indirizzamento locale**;
- **Rilevamento** (ed eventuale correzione) **di errore**;
- **Controllo di flusso** (con protocolli a finestra);
- **Protocolli accesso multiplo**.

I **PROTOCOLLI** dello strato collegamento derivano da un antenato comune, ovvero il protocollo **SDLC** della IBM; sono usati in **reti pubbliche** (con **collegamenti punto-punto** [derivato dal SDLC]) e **reti private** (con **collegamenti broadcast**). Ci sono 2 sottostrati nello strato collegamento, ovvero **LLC** (più vicino allo strato 3-rete) e **MAC** (più vicino allo strato 1-fisico). Visivamente, definendo DTE (Data Terminal Equipment, es. modem dell'utente) e DCE (Data Circuit-terminating Equipment, es. OTU della fibra ottica, ovvero il gestore):



Il formato generico delle **TRAME** (PDU strato2) è composto (come da immagine) dalla 2-SDU (ovvero il campo dati) e dalla 2-PCI (gli altri campi in figura).

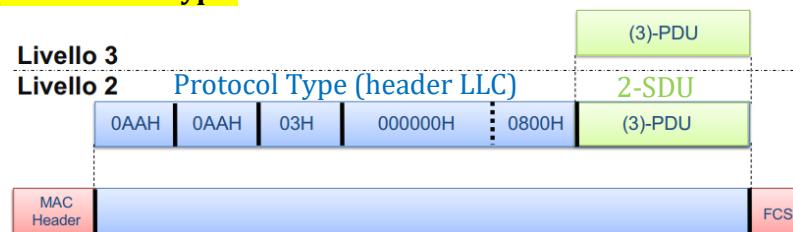
01111110	indirizzo	controllo	dati	CRC	01111110
8	8	8/16	>=0	16	8

⚠ Occorre però fare in modo che, se nella SDU c'è un byte 01111110 (ovvero un byte da trasferire uguale al delimitatore [flag] 01111110), questo non venga confuso con il delimitatore. Per farlo ci sono 2 tecniche, ovvero **BIT STUFFING** e **BYTE STUFFING** (analoghe ma una usa 1 bit di escape, mentre l'altra usa **1 byte di escape** [01111101] per segnalare che il prossimo byte sia uguale ad una sequenze di escape o ad un flag, ma venga gestito come byte da inviare).

Ora vedremo i principali protocolli di strato2 per le **RETI PUBBLICHE**, ovvero:

- **PPP (Point to Point Protocol)** → usato nella linea telefonica o ADSL, oltre che su connessioni SONET/SDH. Ha come obiettivi: **delimitazione delle PDU**, **byte stuffing**, **riconoscimento** (non correzione) **degli errori**, **multiplazione di più protocolli** di strato rete e negoziazione dell'indirizzo di rete (**IP**) [**non** correzione errori, controllo di flusso, mantenimento della sequenza, gestione collegamenti multipunto]. È diviso in 3 sottoprotocolli, ovvero INCAPSULAMENTO, LCP (Link Control Protocol = stabilisce il collegamento PPP e fa l'autenticazione [configurando la tipologia di connessione]) e NCP (Network Control Protocol = dopo l'eventuale autenticazione con successo, definisce le modalità di trasferimento delle PDU e negozia l'assegnazione dell'IP [per ogni protocollo superiore esiste un diverso protocollo NCP]);
- **ATM (Asynchronous Transfer Mode)** → rete a pacchetto con servizio a circuito virtuale su scala geografica con velocità elevate e bassa latenza.

Ora vedremo invece lo strato2 nelle **RETI LOCALI** (rete dove il **transito dei dati rimane su dispositivi sotto il mio controllo**, e non sotto il controllo dell'operatore) [es. la rete di casa nostra o di un laboratorio]. Parliamo infatti dell'**IEEE 802.2 LLC** (il numero 802 indica sempre la rete locale [in questo caso **802.2** perché ci riferiamo allo **strato 2** della **rete locale**]) [che come abbiamo detto comunica con lo strato3-rete]. Questo è un protocollo **orientato al byte**, non usa delimitatori (se ne occupa il MAC), non controlla gli errori (se ne occupa il MAC), le sue PDU non contengono l'indirizzo dei nodi sorgente e destinazione e hanno dimensione variabile; **l'unico compito dell'LLC è dare un campo "Protocol Type"**.



⚠ Ora vedremo i protocolli di accesso per reti locali (LAN)!

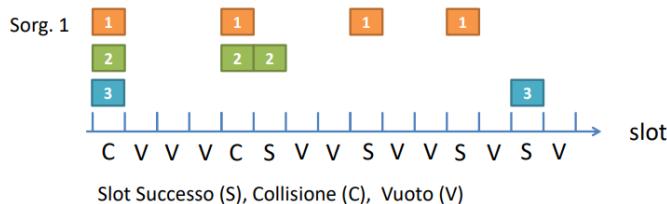
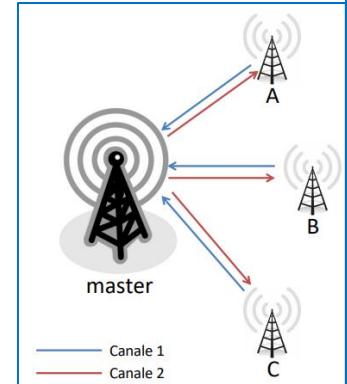
## 6) PROTOCOLLI di ACCESSO per RETI LOCALI (LAN)

Definendo **LAN** (Local Area Network o **RETI LOCALI**), i protocolli per le reti locali nascono per gestire **traffico impulsivo** (non costante, intermittente) e **trasmissione broadcast** [oggi tutte le reti locali hanno **topologia a stella**]. Un problema è la **condivisione del canale** (ne abbiamo già parlato in precedenza): se usiamo la **multiplazione** [flussi disponibili in un unico punto di accesso al canale], il problema è **concentrato**, mentre se usiamo l'**accesso multiplo** [flussi accedono al canale da punti differenti], il problema è **distribuito**. Una possibile soluzione potrebbe essere emulare la **MULTIPLAZIONE STATISTICA** (ne abbiamo parlato in precedenza): quindi nelle reti pubbliche usiamo degli algoritmi per decidere chi trasmette, ma nelle reti locali questo non va bene perché il **principio delle reti locali** è la **SEMPLICITÀ** (chip che costano poco) [le reti locali emulano il modo in cui interagisce l'uomo (es. moderatore che decide chi parla, alzata di mano per parlare etc...)]. Oltre ai ritardi visti prima (propagazione e consegna), vediamo il **ritardo di accesso** il tempo tra quando la PDU è pronta e quando inizia la trasmissione TX [introdotto con l'accesso multiplo].

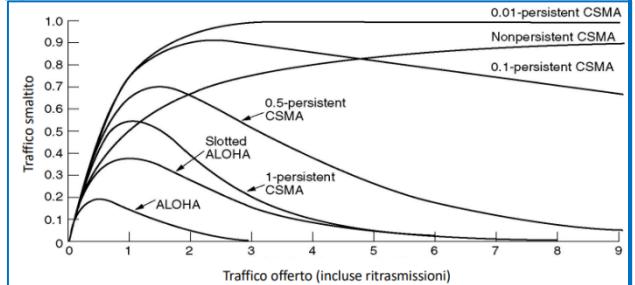
I protocolli LAN oggi sono “**a contesa**” (**ACCESSO CASUALE**, cioè **Ethernet** e **WiFi**) [un tempo erano diffusi invece quelli ad accesso ordinato e a slot con prenotazione]. Quando un nodo deve trasmettere, questo **trasmette a piena banda (NO MULTIPLAZIONE)** **senza coordinarsi** con gli altri nodi; se dei nodi trasmettono contemporaneamente, si ha una **COLLISIONE** (i protocolli MAC possono specificare come renderla meno probabile, come riconoscerla e come recuperarla).

Il 1° protocollo ad accesso casuale è stato l'**Aloha** di Abramson, basato sulla **commutazione di pacchetto su onde radio**. Come funziona? Si hanno i canali broadcast 1 e 2 multiplati in frequenza dove le stazioni A, B e C trasmettono sul canale 1 e ricevono sul 2, mentre il Master trasmette sul 2 e riceve sull'1; quindi se A deve inviare qualcosa a C, allora A lo invia al master e il master lo invia al C; essendo il canale broadcast, quando il Master trasmette a C, anche A "sente" la trasmissione e quindi si rende conto se C ha ricevuto o meno.

Aloha è semplice (non richiede sincronizzazione), inizia la trasmissione in qualunque istante, ma ha un'alta probabilità di collisione. Implementa lo Stop&Wait in caso di collisione (dopo timeout ritrasmetto la trama), ma eseguendo il **BACKOFF**, ovvero il timeout è un tempo casuale (per rompere il determinismo e per evitare nuovamente collisioni) [in caso di ulteriore collisione, si raddoppia il massimo tempo di attesa casuale (backoff esponenziale)]. Una variante è lo **SLOTTED Aloha**, dove il tempo viene diviso in slot temporali (intervalli separati da 2 colpi di clock), per cui dimezzo la probabilità di collisione.



⚠ Aloha è semplice, ma è un protocollo instabile in quanto ha throughput basso (a causa delle collisioni [che sono inevitabili, posso solo cercare di ridurle] e delle ritrasmissioni, che rendono il traffico offerto molto maggiore di quello smaltito) e ritardi di accesso non controllabili a priori in modo deterministico.



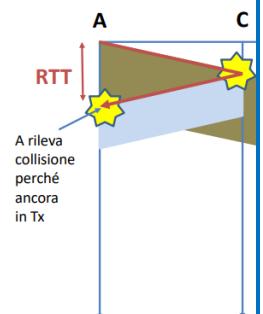
Per spingere più a destra il punto critico del traffico smaltito nel grafico (aumentare il throughput e diminuire le collisioni), si può introdurre il **CSMA** (Carrier Sense Multiple Access), ovvero un meccanismo in cui viene ascoltato il canale prima di trasmettere:

- se "sento" il canale libero → trasmetto;
- se "sento" il canale occupato → ritardo la trasmissione; 2 strategie:
  - **CSMA 1-persistente** = aspetto che si liberi il canale e trasmetto (ma potrebbero esserci più canali che aspettano per trasmettere e potrebbe generarsi la collisione una volta liberato);
  - **CSMA non-persistente** = riprovo a sentire se il canale si è liberato dopo un tempo casuale e se libero, trasmetto.

⚠ Diverso dalle collisioni, perché qui parlo della strategia di quando trovo il canale occupato (apposta per evitare le collisioni), mentre prima con il backoff gestivo cosa fare una volta avvenuta la collisione!

Varianti del CSMA sono:

- **CSMA-CD** [Collision Detection], adatto per mezzi cablati] → rileva le collisioni, ovvero la stazione che trasmette monitora il canale durante la trasmissione e, se sente altre trasmissioni, interrompe la propria. In questo modo faccio già una verifica della trasmissione, perciò non serve una conferma di ricezione (se la stazione non ha interrotto la trasmissione, vuol dire che è stata ricevuta correttamente). La condizione che va rispettata è che **durata minima della trama > 2 · tempo massimo di propagazione** perché altrimenti la trama raggiunge il ricevitore quando la sua trasmissione è già terminata (perciò la collisione non è rilevata!). Dunque su reti piccole (ovvero stazioni vicine) è più efficiente perché posso usare trame di breve durata. La variante preferita è quella 1-persistente (usato nella rete Ethernet).
- Pro: riduco gli sprechi perché sospendo la trasmissione se mi accorgo di una collisione;  
Contro: facile solo nelle LAN cablate: nelle LAN wireless non si usa in quanto la potenza di trasmissione "assorda" l'antenna, che quindi non riesce ad ascoltare il canale per rilevare le collisioni.
- **CSMA-CA** [Collision Avoidance], adatto per canali radio] → fa prevenzione delle collisioni (non le elimina in quanto sono inevitabili) per i mezzi radio (dove non possiamo usare il CSMA-CD). La stazione che trasmette "ascolta il canale: se il canale rimane libero per un periodo detto "**DIFS**", la stazione inizia a trasmettere, altrimenti sospende la trasmissione per un tempo casuale di backoff (come il CSMA non-persistent) [la

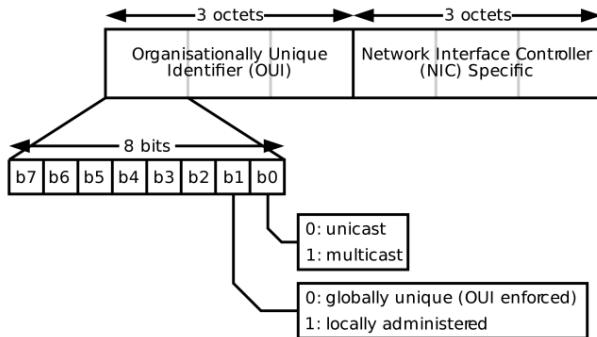


stazione decrementa il suo backoff solo mentre il canale rimane libero, e quando il backoff arriva a 0, ricontrolla che sia libero e trasmette]. La stazione che **riceve** invece verifica la correttezza della trama; se corretta, risponde con una **trama di ACK dopo un tempo "SIFS"** < DIFS (ACK ha la priorità su ogni altra trama). Se il trasmettitore non riceve l'ACK, va in timeout: estrae il tempo di backoff e inizia a decrementarlo e, quando arriva a 0, riparte con la procedura di trasmissione.

In caso di **COLLISIONE**, le stazioni coinvolte ripetono la trasmissione **raddoppiando il massimo tempo di backoff**. Inoltre, si hanno prestazioni migliori su reti piccole (riduco il periodo di vulnerabilità); questo protocollo viene usato dalle reti **WiFi 802.11**.

⚠ Ora vediamo gli **STANDARD PER LE RETI LOCALI** (802.1 = introduzione all'Internetworking di LAN, 802.2 = sottolivello LLC, 802.3 = CSMA-CD (tipo Ethernet), 802.11 = WiFi, 802.15 = Bluetooth).

Abbiamo già detto che il **sottostrato2 MAC** (quello più vicino allo strato fisico) si occupa della delimitazione di trama, rilevazione di errori e l'indirizzamento. Gli **INDIRIZZI MAC** permettono di identificare la scheda mittente e destinatario tra i nodi della LAN (come le poste con mittente e destinatario). Sono molto **lunghi** nonostante siamo in una rete locale perché vengono scritti per essere univoci nel mondo (non solo localmente alla rete); sono composti da **6 Byte** e sono decisi dal costruttore della scheda: i 3 MSB sono "OUI" (Organization Unique Id, ovvero identificano il costruttore), mentre i 3 LSB sono "EUI" (numerazione interna decisa dal costruttore). Il 7° bit del 1° Byte indica se l'indirizzo è **unico nel mondo [0]** o se **scelto localmente [1]**; l'8° bit del 1° Byte indica se l'indirizzo è **UNICAST [0]** (identificano una sola scheda) o **MULTICAST [1]** (identificano gruppi di schede). Se l'indirizzo è FF FF FF FF FF FF (ovvero **tutti 1**), è un indirizzo **BROADCAST** (ovvero si riferisce a tutte le schede che sentono il segnale).



⚠ Quindi, una scheda MAC, quando riceve un pacchetto (corretto), lo accetta se:

- indirizzo di destinazione = indirizzo della scheda;
- indirizzo di destinazione è broadcast;
- indirizzo di destinazione è multicast e il gruppo multicast è stato abilitato.

Parlando di **Ethernet**, è la più usata per LAN (Reti Locali) cablate e MAN (Reti Metropolitane). Una TRAMA Ethernet è composta da:

- “**Intestazione**” dello stato fisico [8 Byte]:
  - **Preamble** = 10101010... per sincronizzare trasmettitore e ricevitore;
  - **SFD** = fine del preamble (“11” alla fine apposta).
- **Intestazione MAC** [14 Byte]:
  - **Destinazione** (messa apposta prima della sorgente per far capire alla scheda se la trama è rivolta a lei oppure no [in tal caso la ignora]);
  - **Sorgente**;
  - **Ultimi 2 Byte** perché l'Ethernet non prevede lo standard LLC (in quanto Ethernet è commerciale e quindi differisce dal suo analogo 802.3, ovvero lo standard), quindi si deve aggiustare da solo per il Protocol Type.
- **Dati** (ovvero la 2-SDU [cioè la 3-PDU]) [46-1500 Byte];
- **FCS** (parte dell'intestazione) [4 Byte];
- **Inter-Packet GAP** [12 Byte].

BYTE	
7	Preamble = 101010.....
1	SFD = 10101011
6	Indirizzo MAC Destinazione
6	Indirizzo MAC Sorgente
2	Tipo protocollo livello superiore > 1500
46 - 1500	D A T I
4	F C S
12	Inter Packet GAP (silenzio)

Equivale a 12

Nell'**IEEE 802.3**, dove siamo sicuri di avere lo standard LLC, gli **ultimi 2 Byte** vengono usati per indicare la **lunghezza del campo dati** (tra 0 e 1500 Byte).

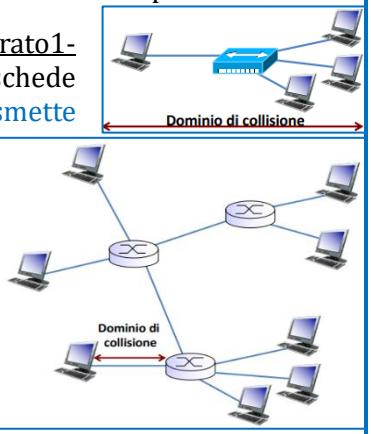
⚠ La lunghezza minima di una trama è di **64 Byte** perché con trame più piccole non sarebbe soddisfatta la condizione vista prima (che durata minima della trama  $> 2 \cdot \text{tempo massimo di propagazione}$ ); solo che la **trama Ethernet** mette il **minimo direttamente nel campo dati**, mentre la **trama 802.3** introduce il **Padding** (con numero di Byte compreso tra 0 e 46 in modo che si riempa la trama per arrivare al numero di 64 Byte minimo).

L'**Ethernet** usa **CSMA-CD 1-persistente**; se avviene una collisione, la stazione interrompe la trasmissione e invia una sequenza di **jamming**; all'avvenuta ricezione non segue un'**ACK** alla stazione che ha trasmesso. Se le stazioni sono nello stesso dominio di collisione, il  $T_{TX,min}$  deve essere  $> RTT_{max}$ . Riassumendo è un protocollo semplice (senza ACK e priorità), distribuito, il più diffuso al mondo.

Parlando dello strato fisico dell'Ethernet, all'inizio erano previste reti corte con trame di almeno 64B. Si è poi passati ad un **livello fisico commutato con canali punto-punto e topologia a stella** (passivo = hub, attivo = switch). Oggi si usano cavi UTP (doppino) o fibre ottiche e si usano praticamente solo **topologie a stella attiva** dove il centro stella preleva la trama che gli arriva, la memorizza e la ritrasmette solo al canale dove si trova solo la destinazione [in questo modo, mediante questa serie di connessioni punto-punto, non si hanno collisioni, in quanto ogni canale punto-punto è dedicato ad 1 solo nodo]. In questo modo, oggi si raggiungono i **Gb/s** ed è **retrocompatibile** anche con schede più lente collegate alla rete (evoluzione dell'Ethernet: la 10Gigabit-Ethernet [full-duplex, ovvero posso trasmettere e ricevere insieme], la 2.5/5Gigabit-Ethernet [versione intermedia per il dialogo con la rete WiFi] e la 100Gigabit-Ethernet [2 velocità differenti]).

Dato il cablaggio strutturato (negli uffici le reti telefoniche arrivavano ad un centralino, quindi topologia a stella), anche nell'Ethernet si è deciso di passare dalla topologia a bus a quella a **stella**; abbiamo visto 2 tipi di stella:

- **PASSIVA** → centro stella = HUB, ovvero un dispositivo multiporta che opera a strato1-fisico; quindi **non riconosce le trame**, ma le riceve e le ritrasmette a tutte le schede Ethernet; ma se 2 schede gli trasmettono 2 trame insieme (collisione), l'hub **ritrasmette a tutti** anche la collisione (male!!!). Quindi è una sorta di **specchio "inconsapevole"**;
- **ATTIVA** → centro stella = SWITCH, ovvero un dispositivo multiporta che opera a strato2-collegamento; quindi **riconosce le trame**: le riceve, le memorizza e le **ritrasmette solo su alcune porte** di uscita, secondo l'indirizzamento del protocollo MAC. Quindi è una sorta di **nodo "store-and-forward"** (riconosce, memorizza, ma non modifica le trame). Ad ogni porta dello switch si fa una connessione punto-punto con un solo nodo (quindi 1 solo trasmettitore e 1 solo ricevitore per canale), perciò **eliminate le COLLISIONI**.



La tecnica più diffusa di “switching” prevede che le stazioni non modifichino il loro comportamento a causa della presenza degli switch [gli switch sanno della loro esistenza reciproca, mentre le stazioni non sanno dell'esistenza degli switch] (**TRANSPARENT SWITCHING**, dove ogni apparato ha il suo indirizzo unico di livello 2 nella “**LAN estesa**” [cioè nell'insieme di segmenti di LAN interconnessi dagli switch], ogni switch ha il suo **switch\_ID** e un identificativo per ogni porta **port\_ID**). Per l'instradamento del transparent switching:

- **ADDRESS LEARNING** = acquisizione degli indirizzi e creazione della **tabella** contenente le coppie “**indirizzo MAC sorgente – port\_ID dello switch**” (più il **timer** associato alla entry); per ogni trama ricevuta, lo switch legge l'indirizzo MAC sorgente (S) e lo associa alla porta (X) da cui riceve la trama (eventualmente cancellando la vecchia entry), poi aggiorna il timer associato alla entry [S,X].
- **FRAME FORWARDING** = ritrasmissione di trame ricevute con filtraggio degli indirizzi, ovvero quando lo switch riceve una trama corretta con indirizzo MAC **unicast** con destinazione D dalla porta X, allora cerca nel database a quale porta è collegato il nodo di destinazione D e se il nodo:
  - ❖ è la porta sorgente X, si scarta la trama;
  - ❖ è la porta Y, inoltra la trama su Y;
  - ❖ non è in tabella, inoltra la trama su tutte le porte attive, tranne la sorgente X.

Se invece l'indirizzo MAC è **multicast o broadcast**, inoltra la trama su tutte le porte attive tranne la sorgente X. La duplicazione delle trame però potrebbe fare in modo che, a causa del “backward learning” (cioè l'address learning), nella tabella di uno switch, una trama sia memorizzata come ricevuta da 2 porte diverse dello switch (quindi siamo in presenza di un loop [anello]). Per risolverlo si usa l'algoritmo qui sotto (implementato direttamente nello switch)!

- **ALGORITMO SPANNING TREE** = algoritmo per evitare gli anelli nelle LAN (il problema sopra annunciato per cui la stessa trama può essere memorizzata su 2 porte diverse), ovvero crea un **albero logico tra gli switch, abilitando solo alcune delle porte** (per evitare gli anelli [loop]). Questo algoritmo richiede lo **switch\_ID**, un indirizzo multicast che raggiunga tutti gli switch e il **port\_ID** delle porte;

⚠ Definiamo **LAN VIRTUALE (VLAN)** le reti locali collegate fisicamente allo stesso segmento di rete, ma logicamente separate in LAN diverse [ovvero **fisicamente è una LAN unica**, ma **logicamente vengono partionate in più LAN**] (in questo modo separo i domini di broadcast e raggruppo utenti che comunicano maggiormente). Questa struttura di livello 2 deve essere supportata dagli switch (inserendo il tag dell'estensione IEEE 802.1Q): negli switch, le porte di accesso (ovvero quelle che comunicano direttamente con i nodi) rimuovono poi il tag della VLAN, mentre le porte "trunk" (ovvero quelle che comunicano con gli altri switch) non rimuovono il tag.

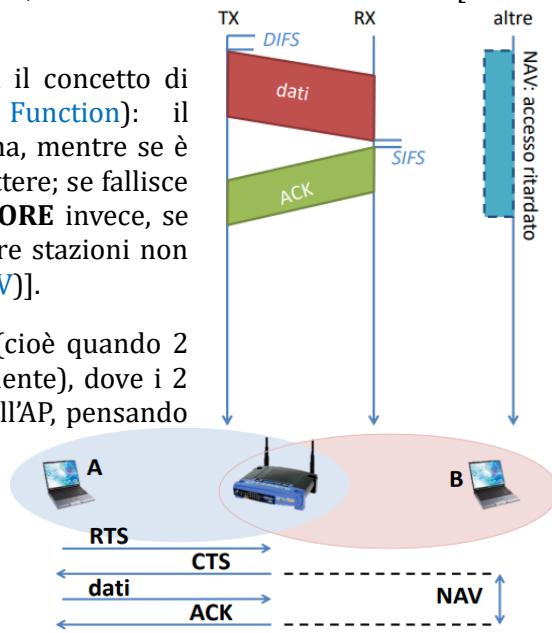
Parlando invece del **WiFi**, è il nome della certificazione (**Wireless Fidelity**) delle tecnologie corrispondenti allo standard **IEEE 802.11**, che copre la tecnologia delle **reti locali wireless** [strato fisico = via radio, strato MAC = Distributed Coordination Function (basato su CSMA-CA), interconnessione tra dispositivi e sicurezza]. L'architettura 802.11 può essere di 2 tipi:

- **Con infrastruttura** → se 2 dispositivi connessi alla rete WiFi vogliono comunicare, devono passare obbligatoriamente dall'AP (Access Point), che fa da mediatore tra i 2 (come uno switch, ma non trasparente, perché i dispositivi vedono l'AP);
- **Senza infrastruttura (WiFi Direct o "ad hoc")** → comunicazione diretta tra terminali, usata solo per applicazioni specifiche ("ad hoc"), perciò meno diffusa.

L'802.11 lavora su **bande di frequenza senza licenza** ("non licenziate") a **2.4GHz** (condivisa con molti altri dispositivi, es. bluetooth, telefoni cordless, fornì a microonde etc...) o **5GHz** (non condivisa, solo WiFi). Le velocità (bitrate) vanno dai **Mb/s** ai **Gb/s** (per le più recenti) e dipendono dallo standard usato e dalla qualità del canale tra l'AP e la stazione (infatti se ci si accorge che si perdono pacchetti, la trasmissione viene rallentata [come quando uno non capisce e si parla più piano]).

Come abbiamo già accennato, il **MAC** è basato sul CSMA-CA (con il concetto di contesa) ed è chiamato **DCF** (**Distributed Coordination Function**): il **TRASMETTITORE** se ha il canale libero per DIFS trasmette la trama, mentre se è occupato aspetta per un tempo casuale (backoff) e riprova a trasmettere; se fallisce (cioè non riceve l'ACK) il backoff diventa esponenziale. Il **RICEVITORE** invece, se riceve correttamente i dati, manda ACK dopo un tempo SIFS [le altre stazioni non fanno nulla per un tempo pari alla durata dello scambio di trame (**NAV**)].

Si può verificare però **collisione a causa del "terminale nascosto"** (cioè quando 2 terminali sono a portata dell'AP, ma non sono a portate reciprocamente), dove i 2 terminali non si rilevano reciprocamente e trasmettono entrambi all'AP, pensando che il canale sia libero. Per risolvere ciò, si usa il **DCF CON HANDSHAKING**: il TX invia una **microtrama ReadyToSend [RTS]** (contenente la durata dello scambio) all'RX prima della trama effettiva; l'RX risponde dopo un tempo SIFS con una **microtrama ClearToSend [CTS]** (contenente la durata rimanente dello scambio) che gli altri terminali sentono e quindi aspettano (non si eliminano le collisioni, ma si riducono).



⚠ Altro problema è che se c'è una stazione lenta collegata all'AP, il **throughput** (bitrate) di tutte le altre stazioni si uniforma alla stazione più lenta [**ANOMALIA DELLE VELOCITÀ TRASMISSIVE**]!

Parlando del **Bluetooth**, è il nome che racchiude lo **standard IEEE 802.15** che si occupa di **trasmissione radio a corto raggio** (collegamenti di periferiche e audio) usando la rete **PICONET**, composta da **1 nodo centrale (master)** e fino a **7 nodi periferici attivi (slave)** nel raggio di 10m; gli slave parlano solo con il master che li interella se vuole parlare con loro [collegando più piconet mediante un nodo "bridge" si genera **SCATTERNET**]. Quindi l'architettura è composta da alcuni strati tra cui il fisico (radio), **link control baseband** [basato sul master che interroga gli slave tramite POLLING] (stesso livello del MAC), **link manager** (gestione) e **L2CAP** [Logical Link Control Adaption] (multiplexing protocolli e gestione messaggi)]. Inoltre il **Bluetooth SIG** specifica le applicazioni da supportare, usando dei **PROFILI** (es. auricolari audio, tastiere, ...). Come detto prima, il Bluetooth usa la banda **2.4GHz** (o banda **ISM**, Industrial Scientific Medical) con tecnica trasmissiva detta **Frequency Hopping Spread Spectrum** (ovvero con 79 frequenze possibili con canali da 1MHz, dove i dispositivi saltano tra le frequenze secondo una sequenza fissata dal master, riducendo l'interferenza). Gli standard dal 4.0 in poi hanno introdotto la modalità **Low-Energy [LE]**.

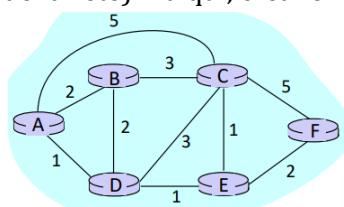
## 7) STRATO ROUTING (Instradamento, Strato di Rete) (3°)

L'**INSTRADAMENTO** (Routing) viene effettuato consultando le tabelle di instradamento per ogni PDU (in una rete datagram) o per ogni connessione (in una rete a circuito virtuale); queste tabelle contengono informazioni tipo qual è il prossimo router (“next-hop”) per ogni destinazione. Quindi si hanno indirizzi univoci, mapping degli indirizzi, tariffazione su rete pubblica e controllo di congestione.

Riguardo l'instradamento, ci sono **3 elementi da considerare**:

- **PROTOCOLLI DI INSTRADAMENTO** (routing protocols) → definiscono la modalità di scambio di informazioni sullo stato della rete, al fine di costruire le tabelle di instradamento;
- **PROCEDURE DI INOLTRO-PACCHETTI (FORWARDING)** → operazioni necessarie per instradare i singoli pacchetti verso la giusta porta di uscita (usando le tabelle per inoltrare i pacchetti);
- **ALGORITMI DI INSTRADAMENTO** (routing algorithms) → operazioni necessarie per scegliere un buon percorso dal nodo sorgente al nodo destinazione (date le informazioni sullo stato della rete). Da qui, creano le tabelle di instradamento.

La topologia della rete viene infatti trasformata in un **GRAFO PESATO** (nodi = vertici, canali = archi) [pesato perché si assegna un “costo” (peso) agli archi]; il “buon” percorso da scegliere è il **percorso a costo minimo**. Il **costo** degli archi dipende dalla distanza, dal ritardo, dal costo effettivo (\$), dal livello di congestione; può essere statico (costante) o variabile a seconda dello stato di rete.



Ci sono diversi **algoritmi che non richiedono il coordinamento dei nodi** (random [scelgo porta casuale], flooding [instradato verso tutte le porte disponibili], deflessione-hotpotato [instradato verso la porta corretta se libera, altrimenti su una porta libera]). Ma noi vogliamo il percorso “ottimo”, per questo si usano algoritmi più complessi. Definendo il **calcolo del percorso**:

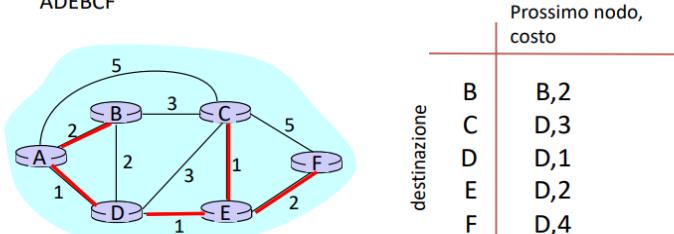
- **Centralizzato** = 1 nodo calcola il percorso di tutti gli altri;
  - ❖ **Pro**: possibili algoritmi complessi, instradamento coerente per tutti i nodi
  - ❖ **Contro**: sensibile ai guasti del nodo centrale e congestione intorno al nodo centrale
- **Distribuito** = tutti i nodi comunicano tra loro (con i protocolli di instradamento) e calcolano i percorsi;
  - ❖ **Pro**: resistente ai guasti, comunicazione uniforme su tutta la rete
  - ❖ **Contro**: richiede intelligenza a tutti i nodi (se comunicazioni errate → errato instradamento)

Allora classifichiamo gli **ALGORITMI DISTRIBUITI** a seconda di quanto i nodi hanno informazioni sulla rete:

- **LINK STATE [LS]** = informazione **GLOBALE** (**tutti i nodi** conoscono la topologia di rete completa, compresi i costi dei canali, e tutti i nodi si scambiano informazioni). Ogni nodo invia informazioni del costo dei suoi canali a tutti gli altri nodi (in broadcast) e quindi ogni nodo calcola i percorsi a costo minimo (ovvero i **CAMMINI OTTIMI**) verso tutti gli altri nodi, usando l'**ALGORITMO DI DIJKSTRA** (algoritmo iterativo [dopo  $k$  iterazioni si ottengono i cammini ottimi per  $k$  destinazioni] che funziona con solo costi positivi [ricorda algoritmi]). Per capirlo vediamo un **esempio**, definendo però:

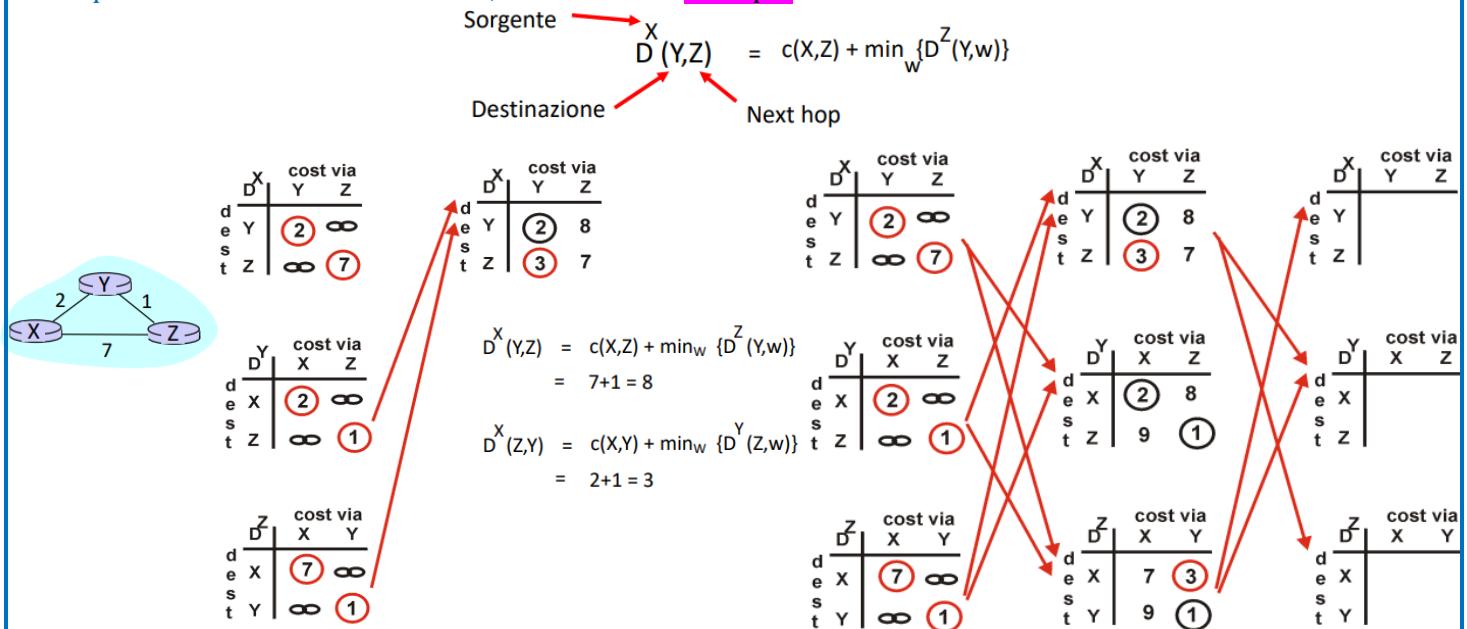
- $c(i,j)$  = costo del canale dal nodo sorgente  $i$  al nodo  $j$  [ $\infty$  se nodi non collegati direttamente];
- $D(j)$  = costo corrente del percorso migliore dalla sorgente alla destinazione  $j$ ;
- $p(j)$  = nodo precedente a  $j$  nel percorso da sorgente alla destinazione  $j$ ;
- $N$  = insieme di nodi per cui ho già trovato il cammino ottimo.

Step	start N	$D(B),p(B)$	$D(C),p(C)$	$D(D),p(D)$	$D(E),p(E)$	$D(F),p(F)$
0	A	2,A	5,A	1,A	infinity	infinity
1	AD	2,A	4,D		2,D	infinity
2	ADE	2,A	3,E			4,E
3	ADEB		3,E			4,E
4	ADEBC					4,E
5	ADEBCF					



⚠ Se abbiamo  $m$  nodi, ad ogni iterazione controllo tutti i nodi ancora non appartenenti ad  $N$  e li aggiungo alla distanza minima, perciò ho complessità quadratica  $O(M^2)$ . Quindi ci sono algoritmi migliori, a complessità logaritmica  $O(M \log(M))$ !

- **DISTANCE VECTOR [DV]** = informazione **PARZIALE** (i nodi conoscono solo i nodi a cui sono fisicamente collegati e solo i costi dei canali a cui sono collegati; **solo i nodi adiacenti** comunicano tra loro). Continuano fino a quando i nodi non scambiano più informazioni, terminando in modo autonomo (nessun segnale di "fine algoritmo"). Ogni nodo scambia periodicamente con i nodi adiacenti un vettore contenente le destinazioni che può raggiungere e la distanza dalle destinazioni misurata in costo. Il nodo che riceve il vettore lo confronta con la propria RT (routing table, tabella di instradamento [ne abbiamo parlato sopra]), aggiungendo nuove destinazioni, cambiando i percorsi se i nuovi sono più corti e modificando i costi se usa un nodo adiacente come miglior scelta. Quindi rispetto agli algoritmi LS, è facile, ma è lento a convergere e propaga errori di routing. Per implementarlo si usa l'**ALGORITMO DI BELLMAN-FORD**, cioè si usa la **tabella delle distanze** (**ogni nodo ha la sua tabella con 1 riga per ogni possibile destinazione e 1 colonna per ogni nodo adiacente**). È **iterativo, asincrono e distribuito** (ogni nodo avvisa quelli adiacenti solo quando il suo cammino ottimo verso una destinazione è cambiato); ogni nodo esegue un **loop infinito** (aspettando una modifica nel costo del canale o un messaggio da un nodo adiacente), ricalcola la tabella delle distanze e, **se il percorso migliore verso qualche destinazione è cambiato, avvisa i vicini**. Esempio:



⚠ Se viene **modificato il costo di un canale**, il nodo rileva la modifica, modifica la tabella delle distanze e, se la modifica migliore dei cammini, avvisa i nodi adiacenti ("good news travels fast"), mentre altrimenti si rileva il problema del "count to infinity"!

⚠ Se ho  $M$  nodi (con  $E$  canali ciascuno), la **complessità**:

- o LS → ogni nodo invia  $O(M)$  messaggi, ciascuno lungo  $O(E)$ , quindi  $O(E M)$
- o DV → ogni messaggio contiene tutte le destinazioni  $O(M)$  ed è mandato a  $O(E)$  vicini, quindi  $O(E M)$

Con **velocità di convergenza**:

- o LS → ogni volta che un link state è propagato, ho nuova topologia (**convergenza immediata**)
- o DV → scelte nodo dipendono dalle scelte degli adiacenti (**convergenza variabile**)

Cosa succede **se un nodo non funziona bene**?

- o LS → i nodi possono annunciare costi dei canali scorretti, ma dato che ogni nodo ha la sua tabella, al prossimo annuncio **tutto si corregge**;
- o DV → i nodi possono annunciare costi dei cammini scorretti, ma, dato che ogni annuncio è usato da tutti i nodi, gli errori si **propagano nella rete** (errori di routing creano anelli).

Lo strato 3 di rete gestisce le **problematiche legate allo strato 2** collegamento, tra cui:

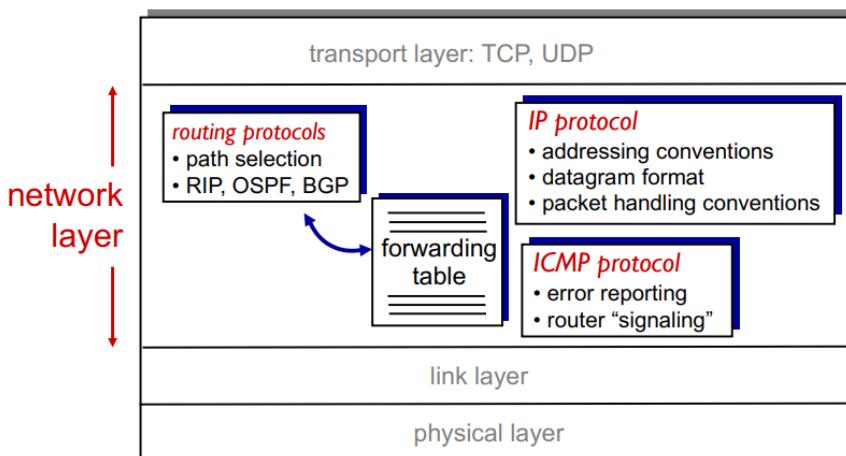
- **gestione non efficiente della ridondanza dei link** (canali, generati dall'algoritmo "spanning tree") [che potrebbe portare ad una "broadcast storm", ovvero un casino di pacchetti che girano];
- **eccessiva dimensione della tabella di routing degli switch** in quanto gli switch non supportano l'aggregazione ("route aggregation") degli indirizzi MAC in tabella (anche perché i MAC sono univoci). Proprio per questo ho bisogno di **nuovi indirizzi** (che saranno gli IP) dove ci sono cose comuni sugli indirizzi, in modo da poterli aggregare e ridurre le dimensioni della tabella;
- **gestione e limitazione del traffico broadcast** dello strato 2 (perché il broadcast è utile, ma bisogna limitarne l'espansione sulla rete [usando dei router per esempio]).

Quindi, specialmente per reti grosse, lo strato3 è essenziale. Ma **come è strutturato lo strato di rete?** I **pacchetti datagram** (**rete a pacchetto**) vengono incapsulati e inviati; chi li riceve, li manda allo strato4 di trasporto. Lo strato3 è basato su una rete di **ROUTER**, che esamina l'intestazione di tutti i datagram-IP; in ogni host e in ogni router ci sono i protocolli di strato3.

Definiamo **"FORWARDING"** = **inoltrare** i pacchetti da un router di input ad un router di output, e **"ROUTING"** = **intradare** i pacchetti, ovvero determinare il percorso dei pacchetti da una sorgente ad una destinazione (tabelle di instradamento).

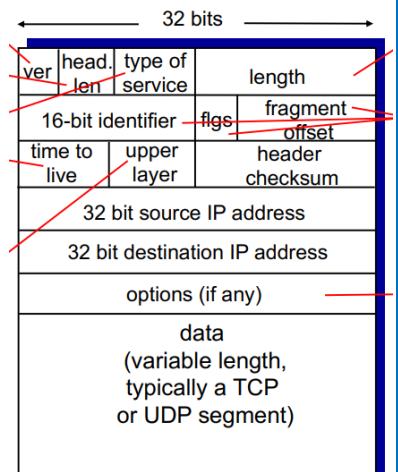
Le reti **DATAGRAM** danno uno strato3 **"connectionless"**, mentre le **VC** (Circuito Virtuale) lo danno **"connection"**; ma **qualie si preferisce?** La preferenza varia a seconda dell'applicazione. Ci sono 2 applicazioni principali:

- **ATM [VC]** → evoluta dalla telefonia, simile alla conversazione umana, **complessità nella rete**;
- **Internet [DATAGRAM]** → scambio di dati tra computer, c'è poca uniformità dei link (molti tipi diversi di link), prevede **semplicità nella rete e complessità all'edge** (cioè il terminale, la stazione) [la rete più importante al mondo, basata su IP, per questo ne parliamo nel dettaglio]. Qui sotto lo strato di rete della rete Internet:



La struttura dell'**IP DATAGRAM** (ovvero la PDU dello strato3) dipende anche dalla versione dell'IP: infatti IPv4 ha parole di 32 bit, mentre IPv6 ha parole di 128 bit (questo perché nell'IPv6 ci sono più indirizzi possibili). I campi del datagram-ip sono:

- **ver** → versione dell'IP (**IPv4** o **IPv6**);
- **header length** → lunghezza dell'intestazione (4 bit);
- **type of service** → tipo di servizio che viene effettuato (DS). Si usa nei servizi interni ai vari provider, non su Internet a livello globale;
- **length** → dimensione totale in byte del datagram (16 bit, quindi massima dimensione  $2^{16} - 1 = 65353$  Byte);
- **16-bit identifier + flg + fragment offset** → usati per frammentazione e **riassembaggio** (ne parleremo dopo);
- **time to live (TTL)** → numero massimo rimanente di "hops" che il datagram può attraversare (cioè numero di canali/archi che il pacchetto può ancora attraversare prima di essere ucciso). Per non fare rimanere inutilmente un datagram in rete;
- **upper layer (o protocol, payload)** → protocollo seguito dal datagram (quindi TCP, UDP o altri);
- **header checksum** → controlla errore sull'header del pacchetto, in quanto se è errata l'intestazione si butta il pacchetto [quindi **se è errata l'intestazione, il router scarta il datagram, se invece è errato il payload, chi riceve scarta il pacchetto**]. Dato che però il TTL va ricalcolato ad ogni router, viene ricalcolato ad ogni router anche l'header checksum (per controllare che anche il TTL nuovo sia ok);
- **32 bit indirizzo IP sorgente;**
- **32 bit indirizzo IP destinazione;**
- **opzioni;**
- **dati.**



Per quanto riguarda la **"fragmentation & reassembly"** (frammentazione e riassemblaggio) dei datagram, questo accade perché i link hanno un **MTU** (Max Transfer Size), quindi i datagram grossi vengono frammentati in datagram più piccoli nella rete; questi vengono riassemblati solo nella destinazione finale. **Oggi però questo non esiste più** (infatti i campi di frammentazione e riassemblaggio sono stati eliminati nell'IPv6).

Per quanto riguarda l'**indirizzo IP**, questo è un **identificatore da 32 bit usato per le interfacce di router e host** (ricordarsi infatti che l'indirizzo IP non è del dispositivo, ma dell'**INTERFACCIA**, ovvero la connessione tra host/router con il link fisico [un dispositivo può avere più interfacce]). Un indirizzo IP è nel formato **NNN.N.N.N**, numeri che usiamo per rappresentare i 32 bit dell'indirizzo:

$$223.1.1.1 = \underline{11011111} \underline{00000001} \underline{00000001} \underline{00000001}$$

223      1      1      1

Ma come sono connesse le interfacce? Gli indirizzi IP sono composti da parte **SUBNET** (bit alti, ovvero fino alla 5^ cifra, es. **223.1.2.1**) e parte **HOST** (bit bassi). Il **SUBNET** è l'insieme delle interfacce del dispositivo con la stessa parte subnet, ovvero una **LIS [Reti logiche]**; **ad ogni rete logica (subnet) corrisponde una rete fisica** (ovvero l'insieme dei dispositivi che posso fisicamente raggiungere), ovvero l'IP assume **corrispondenza biunivoca tra subnet e reti fisiche**. Host sulla stessa subnet = **comunicazione diretta**; Host su subnet diverse = **comunicazione indiretta** (passando da router, usata per usare gli indirizzi IP al posto del MAC).

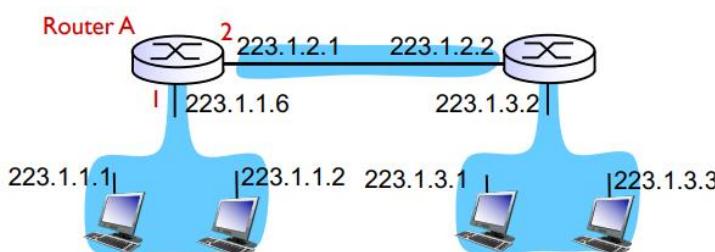
⚠ Anche i collegamenti tra router sono una subnet!

Parliamo di **ONE-ARM ROUTER** se abbiamo **più subnet nella stessa rete fisica**, dove nonostante i 2 host possano comunicare direttamente tra loro, essendo di 2 subnet diverse, devono comunque passare da un router (subnet diverse = comunicazione indiretta sempre) [come le **VLAN**, dove avevamo detto che se 2 VLAN diverse vogliono comunicare tra loro, allora usano un router].

Legato a ciò ci sono degli **INDIRIZZI IP SPECIALI**:

- **Subnetwork ID** → Subnet + tutti 0;
- **Limited broadcast** (ovvero broadcast su rete locale) → tutti 1;
- **Directed broadcast** for network (ovvero broadcast per tutti i router/host che hanno quella determinata subnet) → Subnet + tutti 1;
- **Loopback** → 127 + indifferente (solitamente 127.0.0.1), ovvero il localhost (es. quello che uso quando provo siti web su vscode) cioè mi fermo all'IP senza scendere allo strato2.

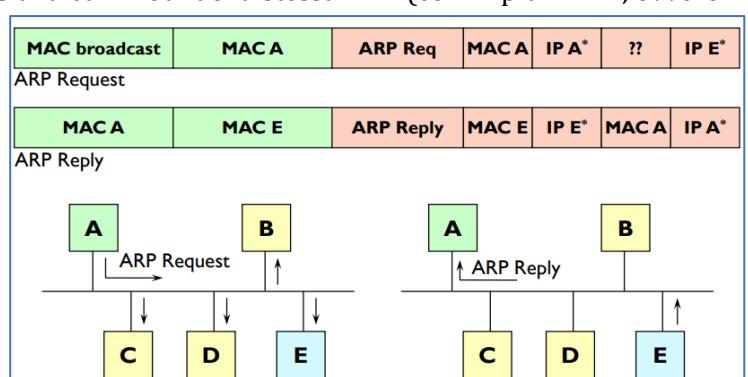
Come avevamo detto ogni dispositivo LAN ha il suo indirizzo LAN (ovvero il MAC) univoco nel mondo; mentre invece gli indirizzi IP hanno un legame logico a livello geografico (es. tutti i dispositivi connessi alla rete del poli avranno l'inizio dell'indirizzo IP uguale): quindi **non c'è la portabilità di avere un indirizzo univoco** ovunque vado, **indipendentemente da dove sono [MAC], ma si ha un significato topologico che ci permette di avere un routing scalabile [IP]**. Ma seppur scalabile, avremmo comunque tantissime righe nella tabella di indirizzamento di ogni router: per risolvere uso l'**INDIRIZZAMENTO GERARCHICO** a livello geografico.



Router A routing table	
destination	output link
223.1.1.0	1
223.1.2.0	2
223.1.3.0	2

Come si determina l'indirizzo MAC dell'interfaccia, sapendo il suo indirizzo IP? Ogni nodo (router/host) di una LAN ha la sua "**ARP TABLE**", con gli indirizzi IP/MAC di alcuni nodi della stessa LAN (con in più il TTL, ovvero il TimeToLive dopo cui l'indirizzo viene dimenticato).

Se un nodo vuole l'indirizzo MAC di un altro e ha solo il suo IP, allora **chiede in broadcast "chi ha questo indirizzo IP?"** [**ARP REQUEST**] e **riceve in risposta unicast** dal nodo con quell'IP "**sono io e questo è il mio indirizzo MAC!**" [**ARP REPLY**]. Questo fa parte del **protocollo ARP**, che sta a metà tra lo strato 2 e il 3; infatti i pacchetti ARP sono delle **trame**, non dei datagram!

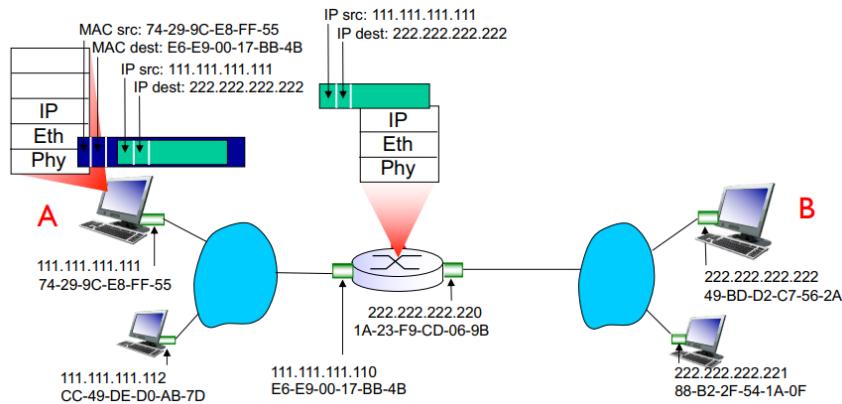


Quindi cosa succede nella **comunicazione indiretta**? [es. comunicazione siti web] (ovvero **attraverso un router**). Facciamo un **esempio** dove mandiamo un datagram dal nodo A al nodo B tramite R (router), supponendo che:

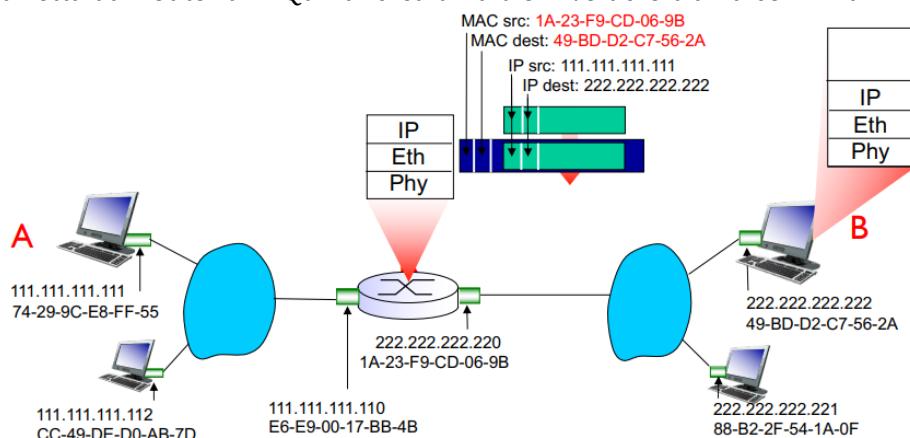
- **A conosca l'indirizzo IP di B** (questo avviene grazie ai **DNS** [Domain Name System], dove tramite l'URL [es. [www.facebook.com](http://www.facebook.com)] questo tira fuori l'IP di quella pagina, tramite un'associazione "nome\_sito-cifre\_indirizzoIP" → in questo modo si ha un plug&play perché anche la gente che non sa niente può usare Internet);
- **A conosca l'indirizzo IP del 1° hop router** (questo avviene con il **DHCP**);
- **A conosca l'indirizzo MAC di R** (questo avviene grazie ad **ARP**).

Fatte queste supposizioni vediamo i **passaggi**:

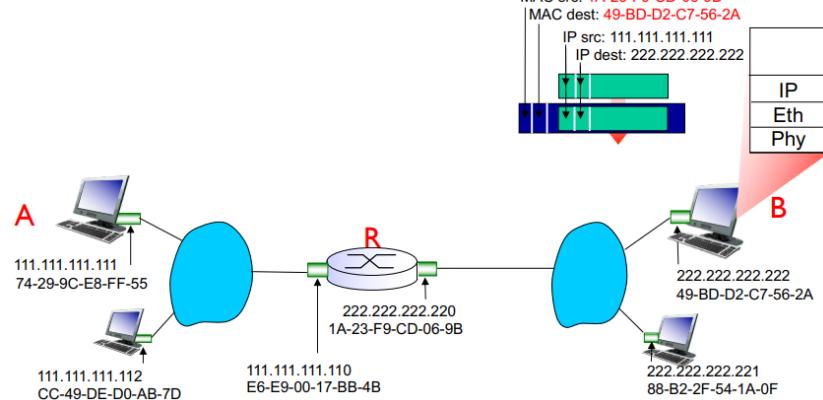
1. A crea il datagram IP con sorgente IP A e destinazione B. Dato che gli indirizzi IP sono su reti diverse, allora entra in gioco Ethernet e viene creata la trama con l'indirizzo MAC del gateway come involucro del datagram IP da mandare a B, in quanto per mandare il datagram IP a B, deve passare dal default gateway perché è una comunicazione indiretta (infatti l'indirizzo MAC è quella del gateway);



2. la trama arriva al router R, che toglie l'involucro trama e tiene solo il datagram IP. Il router è fondamentale perché imposta il MAC destinazione a B; questo perché il router, tramite l'IP di B capisce che basta ora una comunicazione diretta dal router a B. Quindi crea un altro involucro trama con l'indirizzo MAC di B e invia tutto a B;



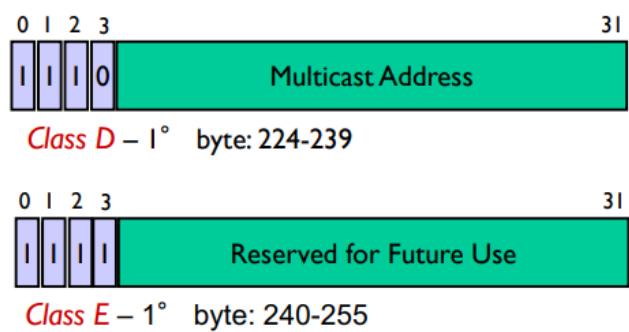
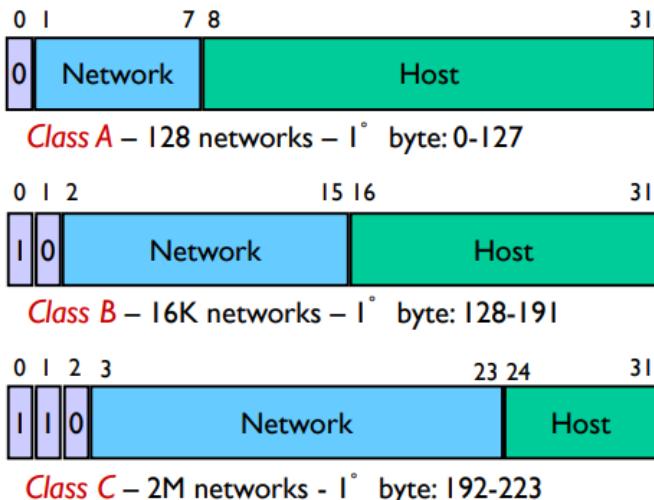
3. B riceve la trama: spacca l'involucro trama, spacca il datagram e tiene solo il segmento nel datagram, che invia agli strati sopra.



Ora parliamo invece dell'**INDIRIZZAMENTO IP**: all'inizio si usava il "**classful addressing**" (divisione statica tra la parte di rete e la parte di host di un indirizzo IP) [poco flessibile, senza subnet]. Poi si è passati al "**SUBNETTING**", dove si usa un numero di bit variabile per la rete (**VLSM**, Variable Length Subnet Masking) usando delle sottoreti

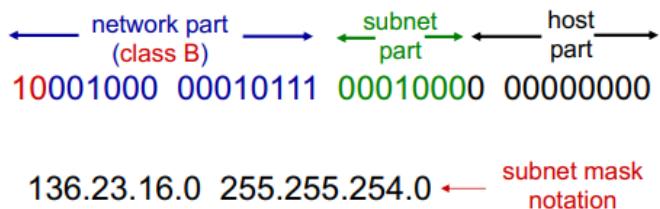
[SUBNET]. Poi però si è preferito passare al “**CLASSLESS ADDRESSING**”, ovvero si elimina il concetto di classi IP (quindi eliminato il concetto di subnet → si parla infatti di “net mask”).

Infatti in passato si avevano le “**classi**” che venivano segnate dai bit iniziali:



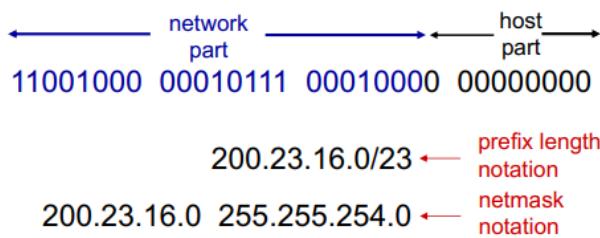
⚠ Le 1^ 3 classi sono IP per reti private?

Per quanto riguarda il **SUBNETTING** (ovvero la VLSM), si prende l’indirizzo IP di una delle classi viste sopra e si definiscono delle subnet più piccole, l’indirizzo IP (NETWORK\_ID) è del formato:



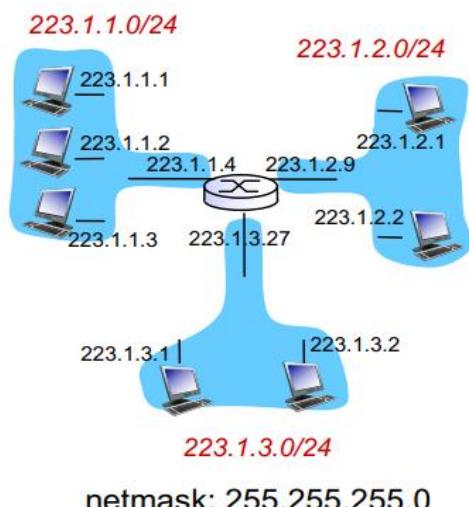
A questo indirizzo però si aggiunge la **SUBNET MASK** (altro indirizzo che viaggia in coppia con la NETWORK\_ID, composto da tutti “1” nella parte di subnet e tutti “0” nella parte di host [es. 111111111111111111111111-00000000]), il cui compito è quello di **identificare la rete** [infatti si aggiunge una colonna nella tabella dei nodi proprio per la subnet\_mask].

Per quanto riguarda invece il **CLASSLESS ADDRESSING** (o **CIDR**, Classless InterDomain Routing), si elimina il concetto di classe, ovvero ce ne sbattiamo dei 1^ bit della classe; gli indirizzi IP sono composti da NETWORK\_ID + PREFIX\_LENGTH (ovvero il numero di bit riservati alla net part e quindi tolti agli host) oppure NETWORK\_ID insieme a NETMASK (come prima tutti “1” nella parte di rete e “0” nella parte di host):

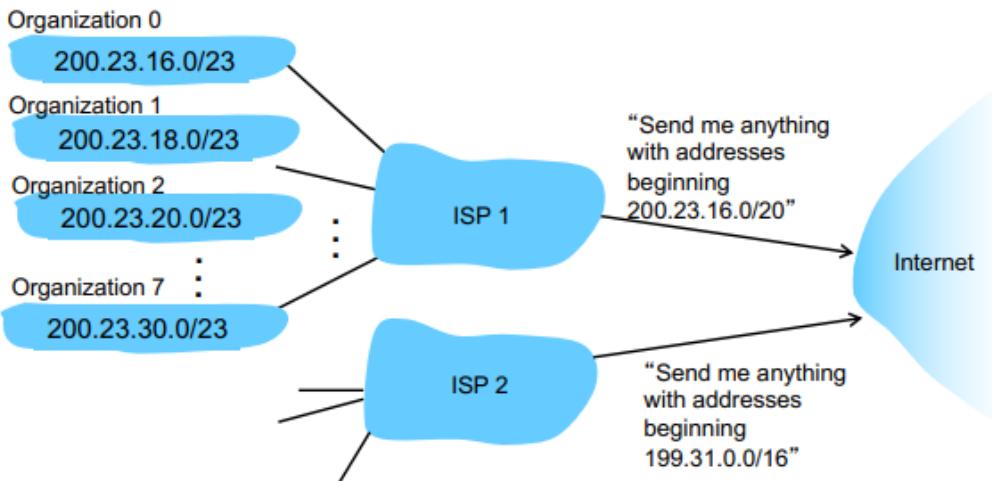


⚠ Sul 4° byte della mask (l’ultimo) non posso usare “254”: questo perché la NETMASK determina quanti bit sono riservati alla rete e quanti agli host; dunque “254” che sarebbe in binario “11111110” ci dice che ho **solo 1 bit riservato agli host** nella nostra rete; perciò avrò solo 2 indirizzi IP host disponibili nella nostra rete (non va bene!) [questo equivale a un’indirizzo con prefix\_length “/31”]! A sinistra un esempio di CIDR nella realtà.

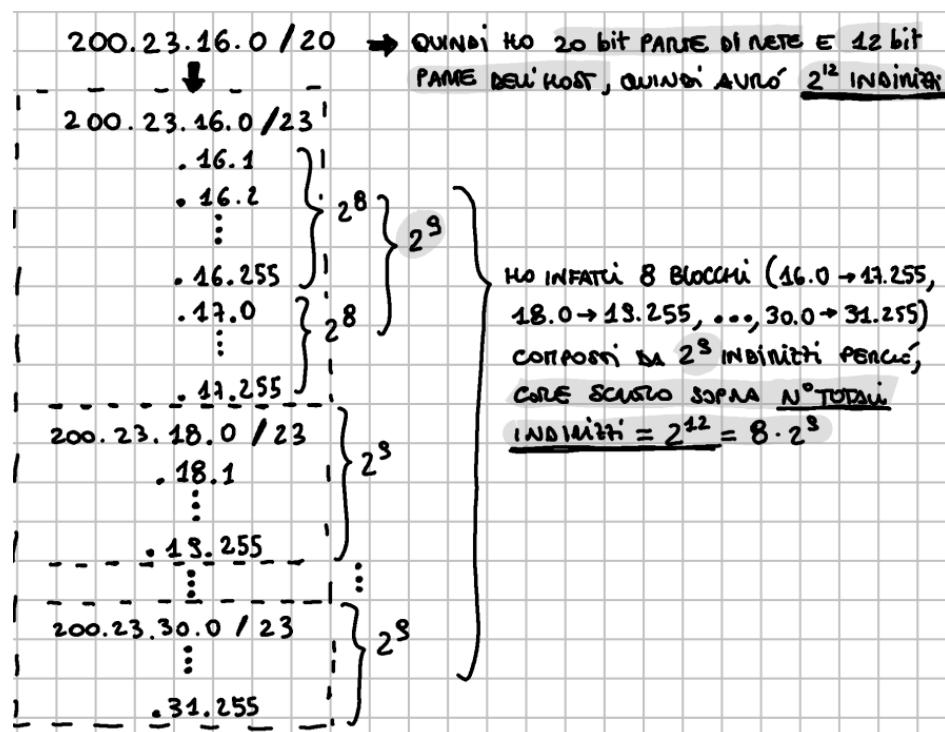
Quindi **ogni nodo della rete dovrà avere un indirizzo IP, una netmask e un default gateway (il 1° hop router)!!!**



Come vengono assegnate le net\_part degli indirizzi IP alle organizzazioni? Le associazioni che gestiscono gli IP usano grandi indirizzi IP (es. terminanti con "/8", ovvero quelle equivalenti alle classi A di una volta) ai grossi ISP (Internet Service Provider). Gli ISP danno a loro volta gli indirizzi IP di questo gruppo alle singole organizzazioni (es. terminanti con "/23"). Con questo **INDIRIZZAMENTO GERARCHICO** (necessario per **scalare il routing**), si **facilita il routing** (instradamento) nella rete perché **raggruppo più indirizzi IP in un ISP**: in questo modo i router in Internet comunicano con i vari ISP e non con i singoli indirizzi IP (saranno poi gli ISP che parleranno poi con i singoli indirizzi IP).



### Esercizio – Indirizzamento gerarchico



Come fa un host a capire i suoi indirizzi IP? Deve fare l'AND bit-a-bit (bit-wise AND) tra i suoi indirizzi IP e le sue netmask, ottenendo l'indirizzo della rete di appartenza (in questo caso 192.168.10.64):

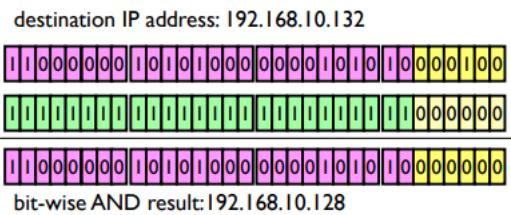
interface IP address: 192.168.10.69

1	1	0	0	0	0	0	1	0	1	0	0	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1
<hr/>													
1	1	0	0	0	0	0	0	0	0	0	0	0	0

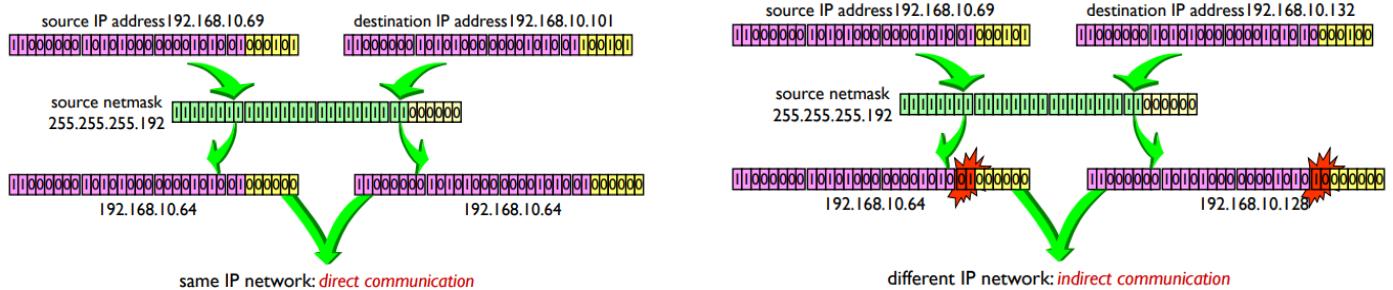
bit-wise AND result: 192.168.10.64

netmask: 255.255.255.192

Come fa un host a capire l'indirizzo IP della destinazione che deve raggiungere? Fa un AND bit-a-bit tra l'IP destinazione e la netmask del nodo (non della destinazione):



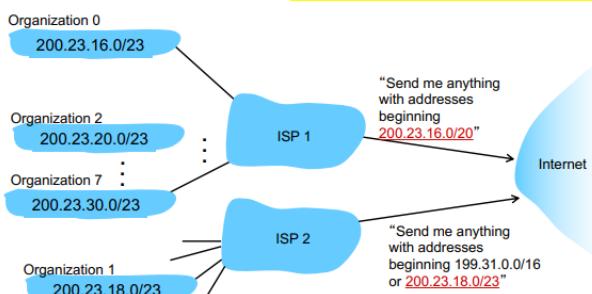
In questo caso si ottiene come indirizzo di rete 192.168.10.128, ovvero dice che il nodo destinazione non è sulla stessa rete del nodo di partenza (192.168.10.64), quindi abbiamo comunicazione indiretta (serve un router!):



Come fa un router a selezionare la corretta porta di uscita? Fa l'AND bit-a-bit tra l'indirizzo IP destinazione di un pacchetto e la netmask di ogni "entry" (riga) della sua routing table (tabella di indirizzamento):

routing table	
destination	output link
200.23.16.1 /23	1
199.31.0.0/16	2

▲ Si usa l'algoritmo di **LONGEST PREFIX MATCHING**, ovvero se ho 2 entry nella routing table, una più specifica (ovvero indirizzo più lungo, cioè prefix più grande, es. /23) e una meno specifica (es. /20), che puntano a 2 destinazioni diverse, allora si prende quella con indirizzo più specifico:



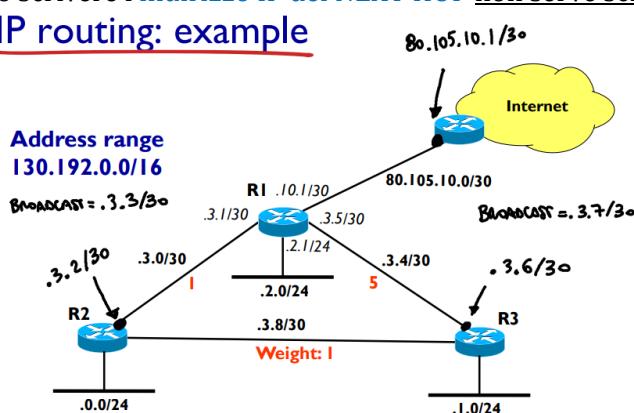
Destination Address Range	Link interface
11001000 00010111 0001***** *****	0
11001000 00010111 0001001* *****	1
11000111 00011111 ***** *****	1
otherwise	2

Come funziona invece una vera tabella di indirizzamento (routing table)? Ci sono 3 tipi di ROUTES:

- **DIRETTE (D)** → reti direttamente connesse al router. Devo indicare la porta locale del router che mi permette di raggiungere quelle reti; si trova l'**INTERFACCIA LOCALE** (perché si ha collegamento diretto).
- **INDIRETTE** → non scrivo l'indirizzo IP del canale usato, ma l'indirizzo IP dell'**interfaccia che devo raggiungere** (in quanto spesso dovrò passare da più "HOP") ovvero il mio **NEXT HOP [INTERFACCIA REMOTA]**:
  - o **STATICHE (S)** → scritte manualmente nella tabella;
  - o **DINAMICHE** → configurate dai protocolli nella tabella.

Nello scrivere l'indirizzo IP del NEXT HOP non serve scrivere la netmask (prefix), ma solo l'indirizzo IP.

### IP routing: example

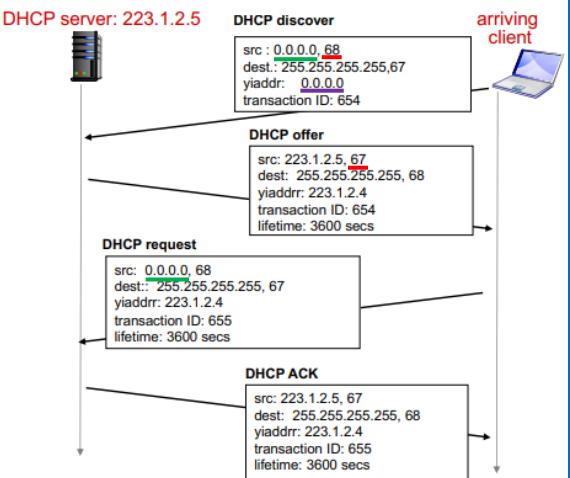


ROUTING TABLE R1 :

TIPO	DESTINAZIONE	NEXT-HOP	Primer (per i BACKUP) COSTO
D	130.192.2.0/24	130.192.2.1	1
D	130.192.3.0/30	130.192.3.1	1
D	130.192.3.4/30	130.192.3.5	1
D	80.105.10.0/30	80.105.10.1	1
S	130.192.0.0/24	130.192.3.2	2
S	130.192.1.0/24	130.192.3.2	2
S	130.192.3.8/30	130.192.3.2	2
S	0.0.0.0/0	80.105.10.2	2

Come fa un host ad avere un indirizzo IP? Si usa il **DHCP** (Dynamic Host Configuration Protocol, protocollo di strato applicazione [strato 5-6-7]) [approccio plug&play], ovvero l'host (DHCP client) riceve dinamicamente il suo indirizzo IP da un DHCP server nella rete con i seguenti passaggi:

- Host fa in **broadcast** la richiesta “**DHCP discover**” (il datagram [strato3] sarà infatti imbustato in una trama broadcast [strato2]). Questo perché tutti i nodi processano il datagram inviato, ma solo il DHCP server sa interpretarlo;
- Server risponde con “**DHCP offer**”. Sono delle offerte, infatti l'host riceve più offerte dal server DHCP, ma ne sceglie solo 1 (la prima);
- Host richiede il suo IP con “**DHCP request**”. L'host mette come indirizzo IP di partenza, non il suo, ma il **broadcast**: questo perché gli altri nodi sappiano l'indirizzo IP scelto dall'host;
- Server manda l'indirizzo IP dell'host con “**DHCP ack**”.



Le richieste/risposte sono fatte con i **datagram**: infatti troviamo gli **indirizzi IP**, ma affianco troviamo anche gli **indirizzi delle porte** (strato4) [16bit, quindi scritti come un singolo numero dopo la virgola]. Il DHCP ha un approccio “**SOFT STATE**”, ovvero il server imposta un tempo di vita (“**lifetime**”) all'indirizzo IP dell'host (dopo cui il server può riassegnare l'indirizzo IP) [a differenza dell'approccio **HARD STATE**, dove deve avvenire un'azione particolare (un evento, TRIGGER) per fare in modo che l'indirizzo IP sia rilasciato]. **Nelle reti si usa quasi solo SOFT STATE.**

⚠ I primi 2 passaggi sono opzionali, perché una volta finito il lifetime, un host può direttamente fare la “DHCP request” se rivuole lo stesso indirizzo IP!

⚠ Come fa l'ISP ad avere il blocco di indirizzi IP da assegnare ai clienti singoli? (ovvero l'**ADDRESS RANGE**). Lo richiede all'**ICANN** (Internet Corporation for Assigned), che assegna il prefix della rete (ovvero la **netmask** o **subnet mask**).

Noi nelle nostre case abbiamo i nostri indirizzi IP, ma non li abbiamo richiesti a nessuno. Com'è possibile? Usiamo degli indirizzi IP riservati a reti per uso privato (**INDIRIZZI IP PRIVATI**); questi non sono accessibili da aree remote della rete (in quanto non sono annunciati su Internet). Questi indirizzi riservati sono **1 di classe A, 16 di classe B, 256 di classe C**:

10.0.0.0 - 10.255.255.255	I class A network
172.16.0.0 - 172.31.255.255	16 class B networks
192.168.0.0 - 192.168.255.255	256 class C networks

Ma se queste reti private non si connettono ad Internet, **come posso andare su Internet?** Il **NAT** permette ad una rete privata di connettersi ad Internet (ne parleremo più avanti).

## →TOOLS PER ANALISI DI RETE:

Vediamo ora dei **tools per l'analisi di rete**:

- **PING** → invia pacchetto **ICMP ECHO REQUEST** e riceve **ICMP ECHO REPLY**, per **verificare che un remote host sia attivo** nel livello di rete. Ha sintassi **ping [options] destination**. Se otteniamo ICMP ECHO REPLY, tutto ok; altrimenti non possiamo dire nulla riguardo al fallimento (probabilmente a causa di attacchi DOS dove viene filtrato il traffico ICMP). Stimiamo **RTT** (Round-trip time) il **tempo tra l'invio di ICMP ECHO REQUEST e la ricezione di ICMP ECHO REPLY**.
- **TRACEROUTE** → **mostra il percorso per arrivare all'host di destinazione**. Ha sintassi **traceroute [options] destination**. I passaggi sono:
  - Host invia un **ICMP ECHO REQUEST** con **TTL** (lifetime) = **1** alla destinazione;
  - Il 1° router sul percorso (ovvero il default gateway) scarta il pacchetto e manda indietro **ICMP TTL Exceeded in Transit**;
  - Host invia **ICMP ECHO REQUEST** con **TTL = 2** alla destinazione;
  - Il 1° router lo passa al 2° che però, essendo di nuovo finito il TTL, manda indietro **ICMP TTL Exceeded in Transit**. E così via (ovvero **TTL crescente per conoscere tutti i nodi del percorso**)!

⚠ In realtà vengono mandati **più pacchetti per ogni TTL** per avere più misure della latenza tra 2 nodi (infatti è probabilistico il calcolo della latenza usando più stime). Questo perché IP non ha sempre gli stessi tempi, ma dipende dalla congestione del traffico della rete (**RETE IP NON OFFRE GARANZIE**)!

- **ARP** → mostra e modifica l'APR table (cache) di un host. Ha sintassi **arp [option] [IPAddr] [EthAddr]**:

Option	Description
-a	Show the current state of the ARP cache, distinguishing between static and dynamic values.
-d IPAddr	Remove from the ARP cache the association related to the host IPAddr.
-s IPAddr EthAddr	Add a static association between the layer-3 address IPAddr and the layer-2 address EthAddr.

- **ROUTE** → mostra e modifica la routing table. Ha sintassi **route [options] [commands] [parameters]**:

Option	Description
print	Show the routing table (Windows). Equivalent to netstat -r
-n	Show the routing table (Linux). Equivalent to netstat -r
add NetAddr mask NetMask Gateway	Add a new route for the network NetAddr/NetMask (ex. 10.0.0.0/255.255.255.0) to the routing table, reachable through the next hop Gateway (Windows).

- **IPCONFIG** → mostra e modifica parametri dell'IP stack (solo su windows). Ha sintassi **ipconfig [options]**:

Option	Description
[no param.]	Show the most important data of the TCP/IP configuration (addresses, netmask, gateway, DNS).
/all	Show all data of the TCP/IP configuration (addresses, netmask, gateway, DNS, DHCP lease, etc.).
/displaydns	Show the current content of the local DNS cache.
/flushdns	Delete the current content of the local DNS cache.

- **IP** → mostra e modifica parametri dell'IP stack (in Linux) [es. **ip address** e **ip route**];
- **NSLOOKUP** → usato per interrogare il DNS (risoluzioni DNS). Ha sintassi **nslookup [-options] [host] [server]**; 2 modalità operative: **interattiva** (per query multiple) e **non-interattiva** (per query singole). Nella modalità interattiva posso anche lanciare dei **subcommands** (es. **ls** di linux, che qui ci elenca le informazioni memorizzate nel server; **set** = setta le opzioni quando il comando è in modalità interattiva; **server** = setta il server di default usato).
- **TCPDUMP** [Linux]/**WINDUMP** [Windows] → **analizzatore di rete** (mostra a video i pacchetti http, ovvero poter vedere i vari campi dell'header Ethernet, il contenuto del datagram, il contenuto dell'header DCP). Oggi l'unico analizzatore di rete grafico (quindi quello che useremo) è **WIRESHARK**.

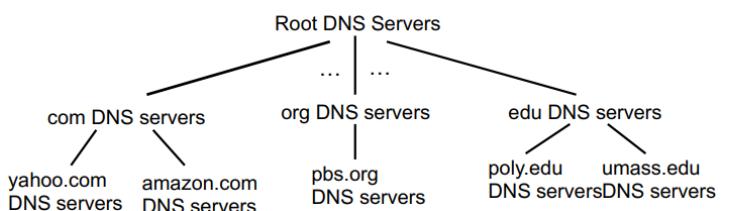
## 8) DNS

Il **DNS** (Domain Name System) è un protocollo di livello applicazione; è un database distribuito implementato nella **gerarchia dei nomi dei server**. Non sono i router a processare i messaggi del DNS, ma **sono i server che sono all'edge che traducono l'indirizzo web scritto a parole in indirizzi IP effettivi**; i router fanno semplicemente **"forwarding"** di questi messaggi.

Perché è distribuito e non centralizzato il DNS? Perché ci sarebbe un **single point of failure** (ovvero errore sul centralizzato = errore su tutto) e ci sarebbe una **congestione sul DNS** (troppo traffico sul DNS): quindi un DNS centralizzato **non scalerebbe**.

Com'è fatto questo DNS distribuito? Abbiamo una **struttura gerarchica dei server**: alla radice troviamo i **ROOT SERVER** (ovvero i punti di accesso a questa struttura gerarchica che hanno il compito di puntare ai top level domain server [domini di 1° livello]); preso un indirizzo web tipo **www.facebook.com**, i **top level domain server** [domini di **1° livello, TLD**] sono l'ultimo tratto dell'indirizzo. Questi contengono sottodomini, come di **2° livello** (es. **facebook** in questo caso) e **www**. Il **puntino non necessariamente divide 2 sottolivelli** (es. se chiamo **www.polito.dauin.ciao.it** magari dauin.ciao è una cosa unica, punto fittizio).

I server del **3° livello** sono **SERVER "AUTORITATIVI"** (avrò un server autoritativo per yahoo.com, un altro per poli.edu etc...) che contengono l'**associazione tra nome sito e indirizzo IP** (ci pensa il provider); questi server però avranno anche altri server se ci sono altri sottodomini (avrò **n server per gli n sottodomini**). Ogni server conosce gli indirizzi IP dei suoi sottodomini per puntare a questi (es. i root server dovranno conoscere gli indirizzi IP dei domini di 1° livello, quelli di 1° dovranno conoscere gli indirizzi IP dei domini di 2° livello, etc...).

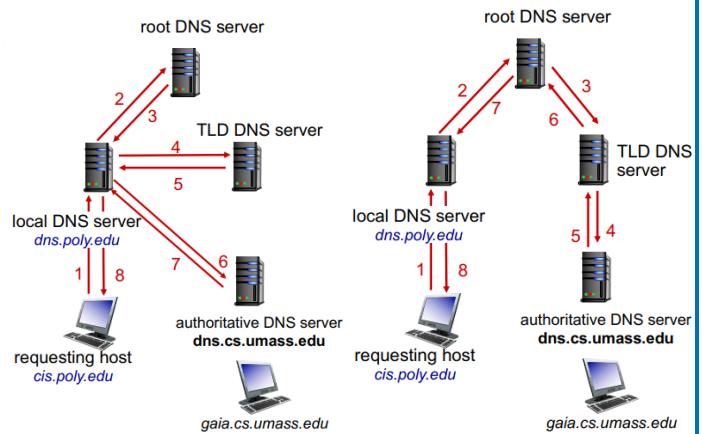


I **ROOT SERVER** dovranno essere tutti uguali tra loro (perché devono permettermi da ogni parte del mondo di accedere ugualmente ai vari server dei sottodomini, in maniera coerente); questo però non comporta una gestione problematica della sincronizzazione tra i vari root server a causa di una modifica, perché le modifiche a cui sono soggetti i root server avvengono **quando cambiano gli indirizzi IP dei top level domain o quando viene aggiunto un top level domain** (perciò quasi mai avvengono delle modifiche), e non quando viene modificata una pagina web (deve cambiare solo il suo IP, del contenuto della pagina non ci interessa se l'IP rimane costante).

Fuori dalla struttura gerarchica, troviamo i **LOCAL DNS Server**; questi si occupano della **CACHE del DNS** (i nostri dispositivi hanno una CACHE DNS che contiene gli ultimi messaggi con il server autoritativo, in modo da non richiederli se provo a riaccedere a quel sito; ma oltre ai nostri dispositivi anche i LOCAL DNS hanno una cache per fare ciò). Dato che tutti accediamo sempre agli stessi siti, ogni rete userà praticamente solo gli indirizzi nella sua cache. Infatti tutti i nostri "home gateway" contengono una local dns server, ma anche il provider avrà un local dns server.

Per quanto riguarda la **risoluzione DNS dei nomi dei siti** (ovvero l'intero processo di comunicazione di come il DNS riceve il nome del sito richiesto, lo trasforma nell'indirizzo IP e lo ridà al client che ha fatto la richiesta), ci sono 2 modalità:

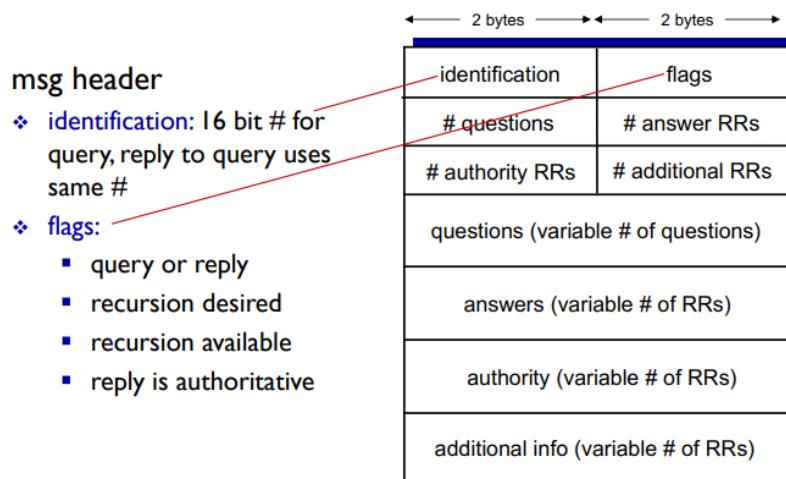
- **Iterativa** → il client chiede al LOCAL DNS server, che sta al centro delle comunicazioni con i vari server;
- **Ricorsiva** → il client chiede al LOCAL DNS server, che però inoltra la richiesta sul root server e così via.



Cosa succede alle richieste nel LOCAL DNS Server se Google cambiasse indirizzo IP (visto che il local server è fuori dalla gerarchia)? Nella **tabella dei server autoritativi** troviamo gli **RR (Resource Records)** nel formato (**key, value, type, ttl**) dove **key** (o name) = **chiave** (nome), **value** = **valore**, **ttl** = **TimeToLive** (usato dalla cache per sapere quanto tempo può tenere l'associazione chiave-valore). Il problema quindi si può risolvere mettendo a zero il TTL (quindi si riandrà a prendere dal server l'indirizzo IP e non dalla cache). Ultimo campo è il **type**, che **modifica il significato dei campi key e value**; può essere:

- **A** → **key** = nome dell'host, **value** = indirizzo IP (unico tipo con nome diretto, senza passare da nomi intermedi);
- **NS** → **key** = dominio, **value** = hostname del server autoritativo;
- **CNAME** → **key** = alias (es. [www.ibm.com](http://www.ibm.com)) è un alias perché il vero nome è servereast.backup2.ibm.com), **value** = vero nome (in questo caso sarebbe appunto servereast.backup2.ibm.com);
- **MX** → **value** = nome del mailserver associato con la key.

Per quanto riguarda la **struttura dei messaggi DNS** (ovvero le **DNS query** [domanda al server DNS] e le **DNS response/reply** [risposta dal server DNS]):



⚠ **RFC** = standard Internet definiti dall'IETF (pubblici, non a pagamento)!

Ora parliamo invece dell'**ICMP** (Internet Control Message Protocol), usato dagli host e dai router per comunicare informazioni di livello rete (strato3). L'**ICMP message** è composto dal campo **type**, campo **code** e 8 byte dell'IP datagram. Per quanto riguarda il **TRACEROUTE**, la sorgente manda una serie di UDP segments alla destinazione;

quando gli  $n$  datagram arrivano ai router, il router scarta i datagram e manda alla sorgente ICMP messages; quando questi ICMP messages arrivano alla sorgente, questa salva i RTT.

## 9) STRATO TRASPORTO (4°)

Questo strato risolve dei problemi legati al trasporto (**MULTIPLEXING** = poter distinguere messaggi di app diverse anche se hanno tutte la stessa destinazione IP; **DEMUTIPLEXING** = arrivato un messaggio, devo capire per quale applicazione è), alla **sicurezza** e alle **non-garanzie dell'IP** (l'IP non offre garanzie, quindi se perdo un pacchetto devo rifare tutto → perciò all'edge dovrò implementare un sistema per "reliable data transfer", ovvero **dovrò implementare dei protocolli a finestra** [dunque leggera perdita di prestazioni, ma al tempo stesso garanzie; quindi questo va bene nella rete Internet]). Altro servizio offerto è il **controllo di flusso** (evita il sovraccarico della destinazione [es. server di ultima generazione che comunica con un sensorino non deve mandare al sensorino dati alla massima velocità in quanto è inutile]) e il **controllo di congestione** (evita il sovraccarico della rete). **UDP** offre il **servizio obbligatorio**, ovvero solo **multiplexing e demultiplexing [connectionless]**; mentre **TCP** offre anche i **servizi facoltativi (reliable data transfer, controllo di flusso e controllo di congestione) [connection-oriented]**. Ricorda che lo strato 4 è all'edge (ovvero host, server, client e **NON al router** → il router non ha strato 4).

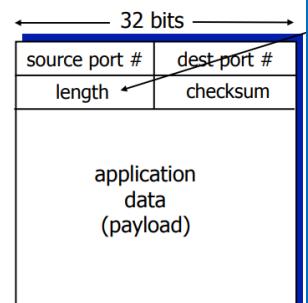
Quindi questo livello trasporto crea un **canale logico tra 2 processi applicativi in esecuzione sugli host (end-system, edges)**; come abbiamo detto prima, **UDP non è affidabile** (non protocollo a finestra), mentre **TCP sì**. Entrambi però **non offrono garanzie di banda e garanzie di ritardo**.

Si usano le porte per il **DEMUTIPLEXING**, ovvero per reindirizzare il segmento all'applicazione corretta:

- **CONNECTIONLESS DEMUTIPLEXING [UDP] → UDP socket identificato da IP destinazione e numero di porta;** quando un host riceve un segmento UDP, controlla la porta di destinazione e manda il segmento UDP a quella porta (il processo che riceve deve "ascoltare" quella porta [socket API]);
- **CONNECTION-ORIENTED MULTIPLEXING [TCP] → socket TCP identificato da indirizzo IP sorgente, porta sorgente, indirizzo IP destinazione, porta destinazione** (perché magari la comunicazione con un host è più veloce di un'altra rispetto ad un altro host, quindi serve identificarli); ogni processo associato ad 1 solo socket (oppure a più socket se associamo ad ogni socket un thread del processo).

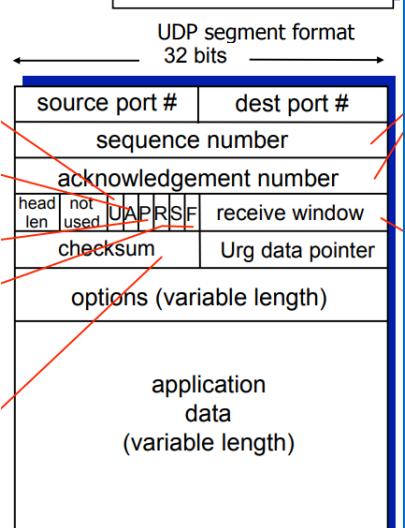
**Come fa un client a sapere il numero di porta giusto da contattare?** Si usano le "well-known ports", ovvero le porte da 0 a 1024, ognuna associata ad una particolare applicazione (es. http usa il protocollo TCP sulla porta 80). Posso anche non usare questo standard (ovvero usare porte randomiche), ma questo crea un problema ai client (i quali pensano di comunicare con la porta standard).

**Come funziona l'UDP (User Datagram Protocol)?** Come abbiamo già detto è un servizio **connectionless** (no handshaking tra sorgente e destinazione UDP; ogni segmento trattato indipendentemente dagli altri) che **offre solo multiplexing e demultiplexing, che non offre affidabilità**, ma offre **alte prestazioni** (infatti viene usato nello **streaming multimediale** [loss tolerant], nel **DNS** [c'è solo 1 domanda e 1 risposta, quindi non mi serve il protocollo a finestra]). Oltre a ciò, nell'UDP segment header troviamo anche un campo "**CHECKSUM**", ovvero **UDP fa anche "error detection"** (non correction, quindi se pacchetto sbagliato, lo scarto e non lo correggo).

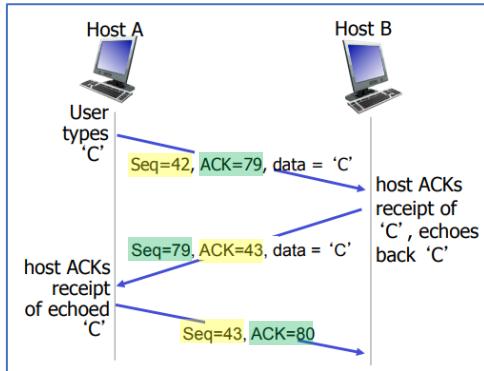


**Nel protocollo TCP possiamo implementare i protocolli a finestra** (noi abbiamo visto Go-Back-N [cumulative ack e 1 timeout] e Selective Repeat [individual ack per ogni pacchetto e 1 timeout per ogni pacchetto ancora non confermato]). **Come funziona TCP?** È **punto-punto** (1 sorgente e 1 destinazione) [infatti per trasmissioni multicast devo usare per forza UDP]; è **affidabile, "byte-stream"**, con un approccio "**pipeline**" (finestra di trasmissione, usata per aumentare/ridurre il bitrate, ovvero per il controllo di congestione e di flusso), "**full-duplex**" (ovvero si possono mandare dati in maniera **bidirezionale** nella stessa connessione), "**connection-oriented**" (handshaking tra sorgente e destinazione).

⚠ Nell'immagine del "**TCP segment format**", vediamo anche i **flag** (A = Ack number valid, R = Reset connection, S e F = aprire e chiudere la connessione), la **receive window** (per il controllo di flusso), il **checksum** (error detection), il **"sequence number"** (valore del 1° byte, riferito all'intero stream di bytes) e **"acknowledgement number"** (numero che mi aspetto dal prossimo byte) [questo perché TCP usa l'ACK cumulativo, ma la gestione fuori sequenza dipende dall'implementazione [= Go-Back-N]].

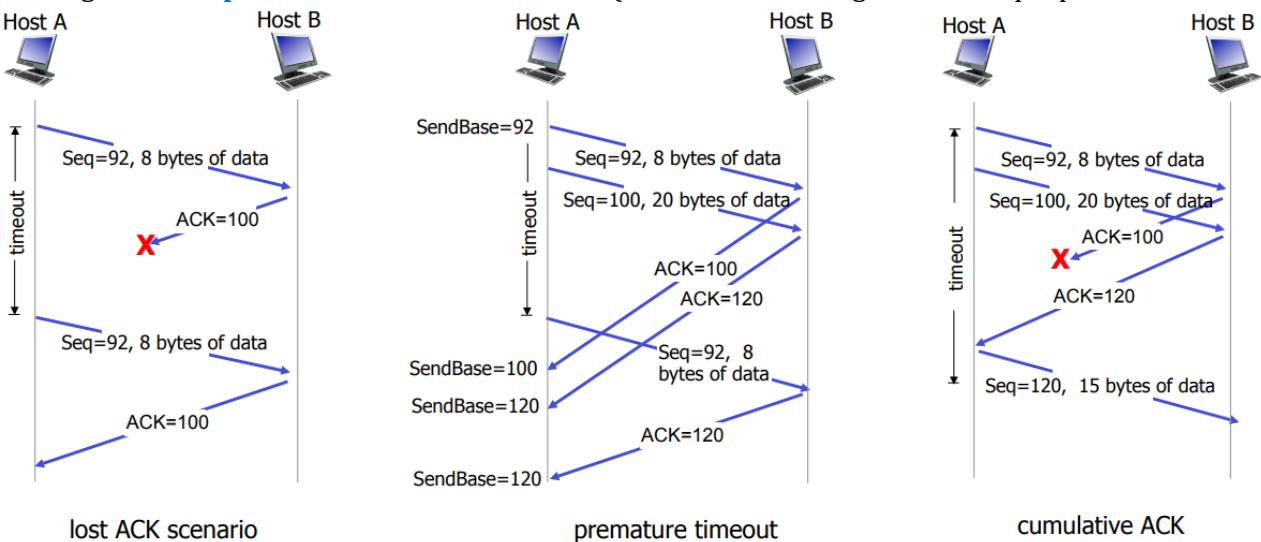


Vediamo un **esempio** di comunicazione TCP tra 2 host:



Per quanto riguarda l'**AFFIDABILITÀ** (ignorando inizialmente il problema degli ACK duplicati e del controllo di flusso e congestione), vediamo **come funziona TCP**. Vengono ricevuti dati dall'applicazione: **si crea un segmento con un certo numero di sequenza** (numero del 1° byte nel segmento) **e fa partire il timer** (se non è già attivo) [timer riferito al più vecchio segmento non ancora confermato (**unacked**)]; **quando scade il timeout, invio solo il segmento che ha causato il timeout** (non invio di nuovo tutta la finestra, quindi diverso dal Go-Back-N [quindi se si perdono molti pacchetti, si preferisce Go-Back-N, ma quindi **non se ne perdono troppi** se usiamo questo procedimento nel TCP (lo vedremo nel controllo di congestione e lo vediamo anche perché tutti i sistemi operativi hanno una finestra di ricezione > 1)]) e **faccio ripartire il timer**. Quando riceve l'ACK, si starta un timer se ci sono ancora segmenti non confermati.

Vediamo degli **scenari particolari di ritrasmissione** (ACK in ritardo vengono comunque presi dall'Host A!!!):



Infatti **situazione particolari** sono:

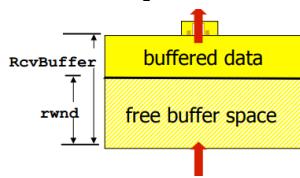
- Pacchetti in arrivo con numero di sequenza atteso n con tutti i pacchetti fino ad n già confermati (acked) → si usa **"delayed ACK"** (si aspetta per 500ms per il prossimo segmento e se non arriva si manda l'ACK);
- Pacchetti in arrivo con numero di sequenza atteso n con **1 pacchetto non ancora confermato (ack pending)** → si manda **1 singolo ACK cumulativo**;
- Pacchetto **fuori sequenza** in arrivo (con numero di sequenza maggiore di quello atteso [gap detected]) → si manda un **ACK duplicato** (indicante il numero di sequenza atteso del prossimo byte);
- Pacchetto in arrivo che mi completa il gap → mando un ACK.

Questi ACK duplicati si possono usare per il **TCP FAST RETRANSMIT**, ovvero, **dopo** aver ricevuto 3 ack con lo stesso numero (per lo stesso pacchetto) [ovvero **3 ACK duplicati**], il **trasmettitore ritrasmette il pacchetto cercato**, prima che scada il timeout (ovvero a furia di chiedere, verrà mandato il pacchetto).

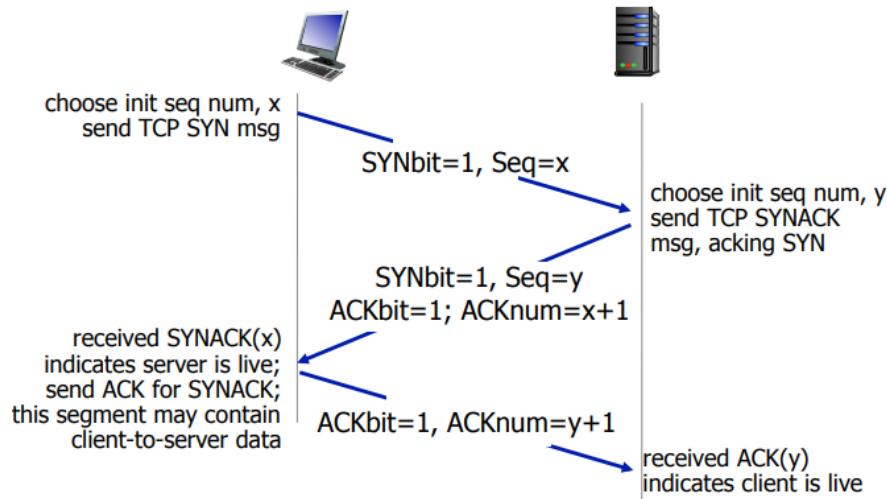
Per **settare il timeout pari al RTT** (RoundTripTime), posso usare l'EstimatedRTT (RTT stimato), ovvero parte dall'RTT misurato e ne faccio una stima con la media  $\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$ , e la deviazione standard  $\text{DevRTT} = (1 - \beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$ ; da cui trovo  $\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$ .

Per quanto riguarda il **CONTROLLO DI FLUSSO** (il **ricevitore controlla il trasmettitore**, così da non generare sovraccarico sul ricevitore), il ricevitore manda al trasmettitore la dimensione del "free buffer space" [**rwnd** →

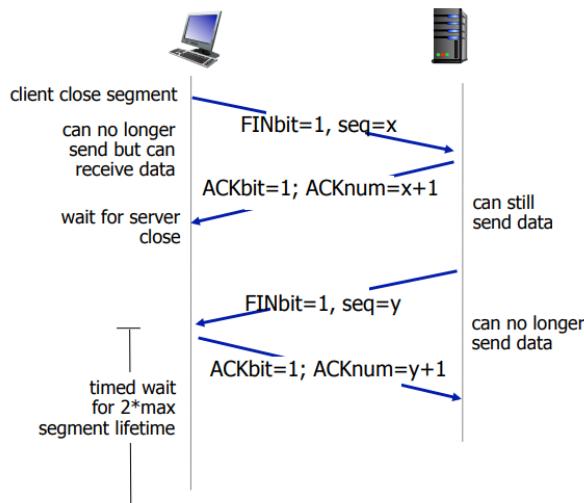
[receive window dimension] dentro il **RcvBuffer** (c'è un campo "receive window" apposito nel TCP message) e il trasmettitore si regola di conseguenza [ $twnd \leq rwnd$ ].



Prima di scambiarsi dati, il ricevitore e il trasmettitore fanno "TCP 3-way handshake" (ovvero decidono come instaurare la connessione e cosa vogliono ottenere dalla connessione), ovvero **aprono la connessione TCP**:



Invece per **chiudere la connessione TCP**, sia client sia server mandano un TCP segment con il **bit di FIN = 1** e rispondono reciprocamente con l'**ACK**:



Per quanto riguarda il **CONTROLLO DI CONGESTIONE** (congestione = traffico sulla rete a causa di sorgenti che mandano dati troppo velocemente in rete, per cui si perdono pacchetti e si generano ritardi), ci sono 2 modi per risolvere:

- **NETWORK-ASSISTED CONGESTION CONTROL** → router mandano dei segnali alle sorgenti per dire che si sta generando troppo traffico. Questo però complica i router, noi vogliamo invece avere i router "semplici" e gli edge (ovvero i terminali) più complessi [perciò TCP usa l'altro metodo];
- **END-END CONGESTION CONTROL** → controllo affidato agli edges (terminali, end-systems).

Il **controllo di congestione TCP** viene fatto usando il modello della **cwnd** (congestion window dimension, ovvero un buffer "fittizio" che mi indica quanto traffico c'è sulla rete) [ $twnd \leq \min\{rwnd, cwnd\}$ ]; proprio perchè la **cwnd** è fittizia, devo poterla stimare e lo faccio basandomi sul **trasmettitore**; ci sono diversi **algoritmi**:

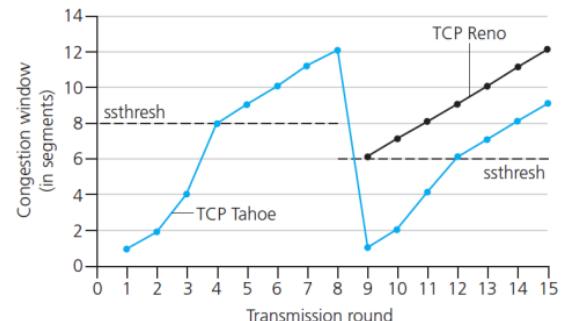
- **AIMD o CA** (addictive increase multiplicative decrease o congestion avoidance) → applicato durante la fase di **congestion avoidance**: mi avvicino linearmente al limite della cwnd, incrementando la cwnd di 1 MSS (maximum segment size) ogni RTT [additive increase]; quando si perde un pacchetto (scade timeout o ACK duplicato), si dimezza la cwnd [ricorda che il bitrate (o più generalmente rate)  $\approx \frac{cwnd}{RTT} \left[ \frac{\text{bytes}}{\text{sec}} \right]$ ].

- **SLOW START** → all'inizio  $cwnd = 1 \text{ MSS}$ , ma poi raddoppia la  $cwnd$  ogni RTT (**crescita esponenziale**, la si ottiene incrementando la  $cwnd$  di 1 MSS ogni ACK ricevuto e non ogni RTT); quindi inizialmente lento, ma la crescita esponenziale lo velocizza.

⚠ La perdita di un pacchetto è segnata da un timeout che scade o da ACK duplicati; il timeout però è peggio. Come reagisce TCP a questi eventi?

- Perdita di pacchetto da timeout →  $cwnd$  settata ad 1 MSS e la finestra cresce prima **esponenzialmente** fino al threshold (per poi crescere linearmente);
- Perdita di pacchetto da 3 ACK duplicati →  $cwnd$  dimezzata e poi cresce **linearmente**.

→ Quindi quando bisogna passare dal SLOW START (esponenziale) all'AIMD (o CA)? Quando  $cwnd$  arriva alla metà del suo valore prima del timeout (usiamo la variabile **ssthresh** posta ad  $\frac{1}{2}$  della  $cwnd$  appena prima della perdita di un pacchetto).



⚠ TCP è "fair", ovvero, se tutte le sue sessioni condividono lo stesso "bottleneck", ognuna dovrebbe avere lo stesso rate (se la banda c'è TCP gliela lascia)!

## 10) STRATO APPLICAZIONE (5°-6°-7°)

È il livello dove troviamo le innovazioni che interessano il mercato (es. Facebook). Per scrivere applicazioni di rete, non devo conoscere come funzionano i protocolli di rete (es. per un'applicazione web, non serve sapere come funzionano i router ["scatola nera"]). Abbiamo già detto che ci sono 2 strutture di applicazioni:

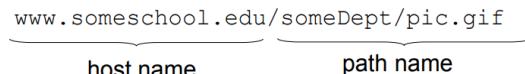
- **CLIENT-SERVER** → non paritetico; client interroga (non direttamente) il server (quindi si connette a intermittenza, IP dinamico), mentre server è un host always-on (IP permanente). **Sono di facile gestione perché il client comunica all'indirizzo IP del server;**
- **PEER-TO-PEER (P2P)** → tutti gli host sono **paritetici**, ovvero si scambiano informazioni tra loro → i peer connessi tra loro offrono il servizio. Qui infatti c'è la "self-scalability" (il sistema si adatta alle necessità del client, ovvero dei peer [peer sono sia client sia server]) [es. torrent, dove chiedo agli altri peer informazioni, ma nel mentre le condivido anche io]. Al tempo stesso, è molto **difficile gestire i sistemi P2P perché i peer vanno e vengono (cambiano IP)**.

⚠ Ricorda che **client e server non sono i "dispositivi"** ma i **PROCESSI**, perché un dispositivo può effettuare entrambi i processi (se vado su google sono un client, se faccio web server sono un server, ma sempre sullo stesso computer)!

Nello strato applicazione abbiamo **sia le applicazioni sia i protocolli**: le **applicazioni offrono i servizi**, i **protocolli offrono supporto alle applicazioni** (es. HTTP). Perché ci servono i protocolli di livello applicazione? Per definire il tipo di messaggi scambiati, la loro **sintassi**, la loro **semantica** e le **regole** (richiesta e risposta). Ci sono **protocolli aperti** (standard, pubblici) [es. HTTP, SMTP...] e **protocolli proprietari** (privati) [es. Skype, non l'applicazione ma il suo protocollo, che chiamiamo sempre Skype].

Che tipo di servizio un'app può chiedere alla rete (attraverso il socket)? **Integrità dei dati** (es. file transfer, ma non streaming multimediale [TCP]), **throughput** (es. streaming multimediale, ma non file transfer [UDP che non rallenta oppure TCP]), **ritardi [timing]** (es. telefonata, ma non file transfer) e **sicurezza** (encryption [dati criptati]). Riguardo alla sicurezza, i **protocolli dello strato trasporto non garantiscono sicurezza** (né TCP né UDP); quindi è stato inserito il **protocollo SSL [Secure Socket Layer]** (o TLS) tra l'applicazione e il protocollo TCP/UDP, che ci garantisce sicurezza (quindi non mi interfaccio direttamente con lo strato4-trasporto ma ho questo livello intermedio per la sicurezza) [es. HTTPS si appoggia su questi socket sicuri].

Partiamo dal **web** (pagina web = base HTML-file su cui troviamo vari oggetti [file html, immagini, JS, audio, etc...], che possono indirizzare con URL) e vediamo la **struttura degli URL**:



L'**HTTP** (Hypertext transfer protocol) è il protocollo che **supporta le app web con struttura client (browser)-server; usa il TCP** per connettere client-server (porta 80) [aperta connessione TCP, invio della richiesta HTTP, invio della risposta dal server e chiusura della connessione TCP] ed è **"stateless"** (il server non mantiene nessuna informazione legata alle richieste precedenti del client [cookies e sessioni]).

Ci sono 2 tipi di HTTP a livello di **connessione**:

- **persistente** → più oggetti inviati con una singola connessione TCP (**si preferisce questo**) [migliorato in http 2.0 per l'uso della pipeline, ovvero oggetti inviati in parallelo];
- **non-persistente** → 1 connessione TCP per ogni oggetto inviato.

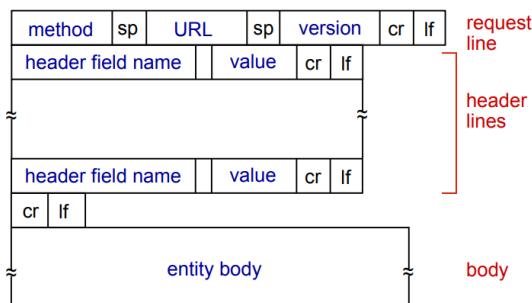
⚠ Oggi esiste lo standard HTTP 3.0 con dentro UDP + QUIC (un altro protocollo di strato4-trasporto che garantisce delle funzionalità aggiuntive, creato da Google)! [solo lettura]

HTTP è il primo protocollo che non ha formato binario, ma è **testuale**, ovvero i messaggi HTTP non sono codificati con dei bit, bensì con del testo. Infatti si hanno i comandi GET, POST e HEAD, che codificati in binario sarebbero 2 bit (3 comandi), ma qui con i testi usiamo molti più bit: questo perché **siamo più vicini allo sviluppatore** quindi si preferisce una struttura più “**user-friendly**”.

Vediamo una **RICHIESTA HTTP** (http 1.1):

The diagram illustrates an HTTP 1.1 request message. It starts with the **request line** (GET /index.html HTTP/1.1\r\n). This is followed by the **header lines**, which include fields like Host, User-Agent, Accept, Accept-Language, Accept-Encoding, Accept-Charset, Keep-Alive, and Connection, each ending with a carriage return character (\r) and a line-feed character (\n). A note indicates that a carriage return, line feed at the start of a line indicates the end of the header lines. Finally, there is the **entity body**, which contains the content of the request.

La **STRUTTURA** è:



I **metodi HTTP/1.0** sono **POST** (le richieste hanno un body proprio per questo, in quanto l'input preso dal form viene messo nel body e mandato al server), **GET** (URL, ovvero l'input del form viene messo nell'URL della richiesta [es. www.ciao.com/animale?scimmia]) e **HEAD** (chiede al server di lasciare un oggetto richiesto fuori dalla risposta); nell'**HTTP/1.1** troviamo anche **PUT** (aggiunge un file nel body) e **DELETE** (elimina il file nell'URL).

Le **risposte HTTP** hanno anche lo **STATUS LINE**

(o **PHRASE**) che contiene un codice numerico per indicare se è andato tutto bene, e una parte letterale per specificare meglio lo status; esempi sono:

- **200 OK** → richiesta andata bene, oggetto richiesto in questo messaggio di risposta;
- **301 Moved Permanently** → oggetto richiesto spostato, nuova posizione in questa risposta (legato all'HTTP REDIRECT);
- **400 Bad Request** → richiesta non capita dal server;
- **404 Not Found** → documento richiesto non trovato su questo server;
- **505 HTTP Version Not Supported**.

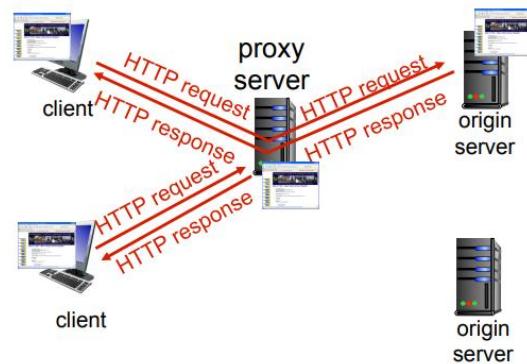
Parliamo dei **COOKIES** (già visti a Web): il client fa la richiesta HTTP, il server crea un **ID numerico per il client** specifico (es. 1678) e **lo salva nel cookie** con “**set-cookie: 1678**” (e lo salva quindi nel suo database); perciò, quando il client fa la stessa richiesta, il server lo riconosce e attiva una specifica azione (utile nei siti web dinamici, che hanno una certa risposta a seconda del client specifico). Quindi **si riesce ad implementare lo “stato”** (staticità) **che in HTTP mancava**.

⚠ I cookies sono un argomento delicato, perché il fatto di memorizzare determinate caratteristiche del client genera problemi a livello di **privacy** (es. per la profilazione degli utenti).

Altro argomento è l'utilizzo della **WEB CACHE** (come la DNS cache), ovvero **vari livelli di memoria cache** che mi servono per accedere ad un certo servizio **senza dover accedere ogni volta al server**, ma usando le informazioni contenute nella cache precedentemente salvate (e quindi riusate); dunque un **PROXY SERVER** (un aiuto per il server).

Sono utili per **velocizzare** l'accesso al server, bypassando l'accesso effettivo, fermandomi solo alla cache dove sono contenute le informazioni salvate precedentemente durante l'accesso al sito web; questo **limita anche la congestione** sul server.

⚠ La **cache locale dei browser** è usatissima, ma i **proxy server** non molto usati (anche perché per avere un proxy server trasparente, devo direttamente settarlo sul browser [**limita il plug&play**]!).



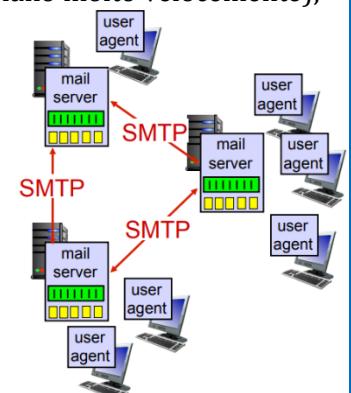
A differenza di ciò che avevamo detto per l'associazione nome-IP, i **contenuti dei siti cambiano molto velocemente**; questo può essere **problematico nelle cache**: perciò viene aggiunto l'header "**if-modified-since: data**", ovvero quando faccio una richiesta HTTP (GET HTTP), **chiedo al server se è stato modificato il contenuto della pagina dalla data "data"** (lo si vede dal campo "**last-modified**") e se:

- **NON è stato modificato** → ricevo come risposta **304 Not Modified** e **accedo alla cache del browser**;
- **È stato modificato** → ricevo come risposta **200 OK** (e quindi l'elemento richiesto).

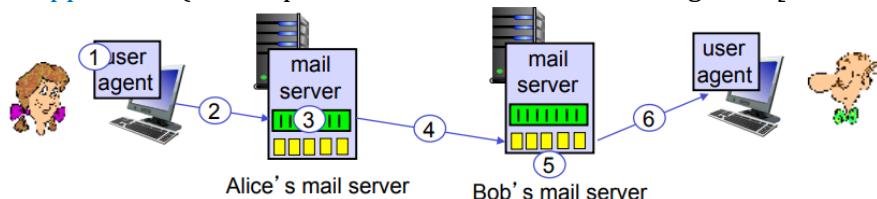
⚠ Ci sono anche **contenuti che non ha senso "cachare"** (es. le breaking news che cambiano molto velocemente); questo lo si fa con degli header aggiuntivi!

Parliamo della **POSTA ELETTRONICA (E-MAIL)**; ha 3 componenti principali:

- **USER AGENT** → il programma per scrivere e leggere mail (es. Outlook, Gmail);
- **MAIL SERVER** → "mailbox" contiene i messaggi in arrivo per l'utente (in figura sono i quadratini gialli) e "message queue" (coda di messaggi) contiene i messaggi da inviare in uscita (in figura è il rettangolo verde) [coda perché se non si riesce ad inviare il messaggio, rimane in coda per un successivo tentativo di invio];
- **PROTOCOLLO SMTP** → per l'**invio di email tra mail servers** (**CLIENT** = mail server inviante, **SERVER** = mail server ricevente).



L'invio di una mail comporta un **doppio salto** (ovvero passando da un mail server di ingresso [CLIENT] e un mail server di uscita [SERVER]):



**SMTP** usa **TCP** (per il trasferimento affidabile di un email da client a server) con **trasferimento diretto** (3 fasi di trasferimento: **handshaking** [di SMTP oltre che un altro handshaking TCP], **trasferimento di messaggi** e **chiusura**); i **comandi** sono anche qui **letterali-ASCII** (come HTTP) e le **risposte** sono "status code and phrase" (come HTTP). I messaggi devono essere codificati in **7-bit-ASCII** (e devono terminare con \n. \n). **Esempio** di interazione SMTP:

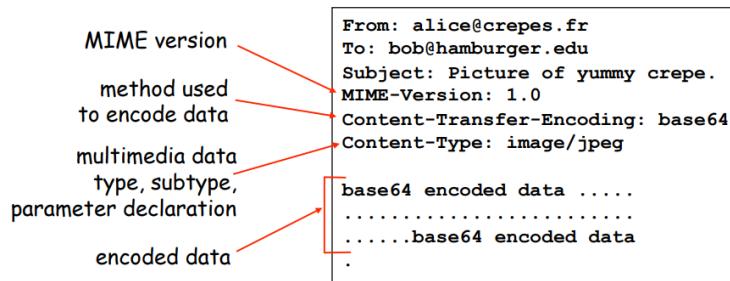
```

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

⚠ Un mail server accetta le mail che vengono da ogni dominio (se filtrassimo solo su email del dominio specifico, avremmo una posta interna), dunque ci si affida all'autenticazione dei domini mittenti, ma nel caso di spam e fishing, il destinatario deve occuparsene (quindi **filtrarle**).

Lo standard [RFC 822](#) prevede che i messaggi SMTP abbiano l'**header** (mittente, destinatario, soggetto [oggetto]) e il **body** (il messaggio effettivo, il **corpo** della mail [[ASCII](#)]); il mittente e destinatario nell'header non sono i comandi (del protocollo SMTP) **MAIL FROM** e **RCPT TO** (ovvio che ci sia una relazione, perché indicano la stessa cosa, ma sono diversi, in quanto **nell'header non troviamo i comandi ma solo gli indirizzi**).

Il protocollo SMTP può essere esteso con l'estensione **MIME** (Multimedia Mail Extension) [che va indicato con delle linee aggiuntive nell'**header** della mail]; infatti, **sempre** bit trasferiamo, ma segnalando questa estensione al client ricevente, questo **saprà che i bit ricevuti non dovranno essere interpretati come ASCII, ma come un file di una certa estensione** (content-type); è necessario il tipo di encoding (content-transfer-encoding) [es. ASCII]:



⚠ Per avere più content-type e content-transfer-encoding, si può mettere nell'header "**MIME Multipart**" (es. per mandare un file jpeg e anche html)!

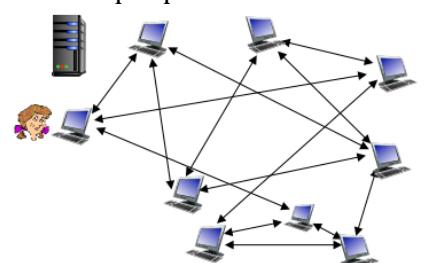
Una volta arrivata la mail al client destinatario, come fa a leggerla? Ci sono 3 **MAIL ACCESS PROTOCOL**:

- **POP** → nella authorization phase (con i comandi user [username] e pass [password]), ma dato che vengono scritti username e password in chiaro (non criptati) non è sicuro. Le risposte del server possono essere +OK e -ERR; nella transaction phase, il client ha i comandi list (elena le dimensioni dei messaggi), retr (recupera email dalla lista), dele (elimina una mail dalla lista) e quit (esce);
- **IMAP** → mantiene tutti i messaggi nel server, permette di organizzare la mail in cartelle, mantiene lo stato tra le sessioni;
- **HTTP** → servizi [WEBMAIL](#) (come gmail) che usano HTTP per interfacciarsi con le email (dunque la comunicazione in backend tra mail server rimane SMTP, ma cambia l'interfaccia dell'utente [HTTP]).

Parliamo ora della **ARCHITETTURA PEER-TO-PEER (P2P)**, dove abbiamo già detto che il singolo peer è **sia client sia server** (tutti paritari, non c'è un server always-on) e si connette in modo **intermittente** (dunque cambia di continuo IP) [applicazioni principali sono i Torrent e Skype (Telefonia su IP P2P)]; il **PEER-TO-PEER** è **autoscalabile**, ovvero aumentando i peer che vogliono accedere ad un servizio, non si creano problemi (perché sono gli stessi peer che richiedono e offrono il servizio, quindi **aumentando le richieste, aumentano anche le offerte di quel servizio**), invece nel [CLIENT-SERVER](#), il tempo per offrire il servizio aumenta all'aumentare dei client.

Un'importante applicazione è **BitTORRENT**, dove i file vengono divisi in pezzi più piccoli ("chunk") e ogni chunk è dato ad un peer; perciò lo scambio dei file equivale a richiedere ai vari peer un "chunk" per poi ricostituire il file. Ma **come faccio a capire quali peer offrono il chunk?** È stato introdotto un **INDEXING SYSTEM**, ovvero offrire al peer che chiede, gli IP dei peer che possono offrire dei chunk a lui utili; di questo se ne occupa il **TRACKER**, un **server che appunto tiene traccia dei peers partecipanti al torrent** [il gruppo di peers che scambiano i chunk di un file è detto [TORRENT](#)]. Quindi non è interamente P2P, perché:

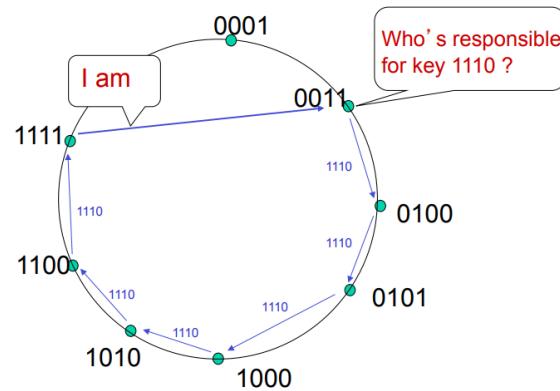
- **Scambio dei chunk** → P2P;
- **Indexing system** → [CLIENT-SERVER](#) (Tracker).



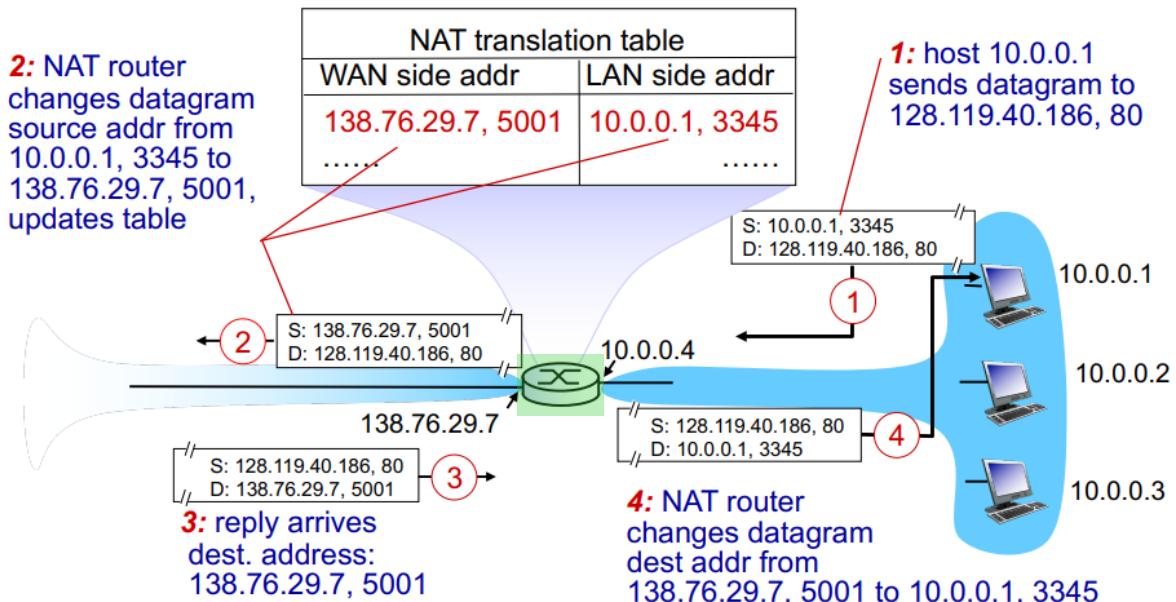
La dinamicità dei peer (che vanno e vengono) è detto **CHURN**. Quando un peer ha scaricato l'intero file, questo può rimanere (aiutando gli altri peer per futuri torrent) o andarsene; per questo BitTorrent usa il "**tit-for-tat**", ovvero **obbliga il peer ad aiutare gli altri peer** durante il suo **DOWNLOAD** (in questo modo obbliga la condivisione da parte del peer, in quanto lui sta aspettando che gli arrivi il suo file e quindi è nel sistema) basandosi su una **classifica di chi ha offerto più chunks** tra i vari peer (se sei tra le prime 3 posizioni [più un 4° random], BitTorrent fa l'**unchoke**, ovvero permette il download a questi peer, mentre gli altri possono solo uploadare i chunks).

Per quanto riguarda l'**UPLOAD**, ogni peer manda periodicamente la lista dei chunks di cui dispone; se il peer richiedente ha bisogno di alcuni chunks, l'algoritmo dice di preferire quelli meno diffusi tra gli altri peer (**Local Rarest First**) [questo perché si aumenta il numero dei peer con quel "raro" chunk e quindi si migliora complessivamente lo scambio P2P].

Il Tracker centralizzato però rende facile il "buttare giù" servizi P2P illeciti; perciò si è passati ad un indexing system "distribuito", ovvero i **DHT** (**Distributed Hash Table**), cioè database P2P distribuiti dove l'indexing system è anch'esso assegnato ai peer. Questo si fa assegnando una coppia (**key,value**) ai vari peer [es. (**titolo\_film**, IP dei peer con quel chunk)]; quando un peer cerca un contenuto, accede tramite la chiave (in questo caso titolo\_film) e il DHT ritorna i valori (in questo caso gli IP dei peer). Come viene gestito il tutto? Si converte la key in un intero (con le **funzioni di HASH**), si assegna un intero (chiave) ad ogni peer e si mette la coppia (**key,value**) nel peer più vicino a quello associato alla chiave (il peer responsabile di una certa chiave è il successore). La DHT è circolare, ovvero ogni peer è a conoscenza dell'esistenza di solo i peer suoi predecessore e successore.



▲ Separatamente da questo argomento, tornando allo strato3-Rete, abbiamo lasciato indietro il **NAT** (**Network Address Translation**). Alcuni range di indirizzi, come avevamo detto, sono riservati a reti private (non annunciati su Internet). Se un host di una rete privata vuole raggiungere un web server su Internet, può farlo? Dall'Host ad Internet, l'host manderà il pacchetto al router (**home gateway**), qui il pacchetto cambierà indirizzo IP sorgente, diventando quello pubblico del router (in figura passiamo quindi dall'host **10.0.0.1** ad avere come indirizzo IP [quando entra in Internet] **138.76.29.7**), salvando però quale host della rete privata ha mandato il pacchetto all'origine; da Internet all'Host, quando riceverò la conferma del pacchetto, tramite la tabella del router dove abbiamo salvato l'host di partenza, il router manderà la conferma all'host corretto:



Dato che il modo in cui il router reindirizza i diversi host nel NAT è basato sulle porte (livello4), questo ci dice che di base **ICMP non è supportato** (in quanto è di livello3, quindi non usa le porte) [si potrebbe risolvere aggiungendo un header]! Quindi il **NAT** permette di usare indirizzi IP locali (privati) nella rete; perciò questo ha ritardato l'obbligo di usare l'**IPV6** (perché non c'è bisogno di usare indirizzi IP più grandi in quanto mi raggruppa tutti i miei host locali, che vengono reindirizzati con le varie porte del router).

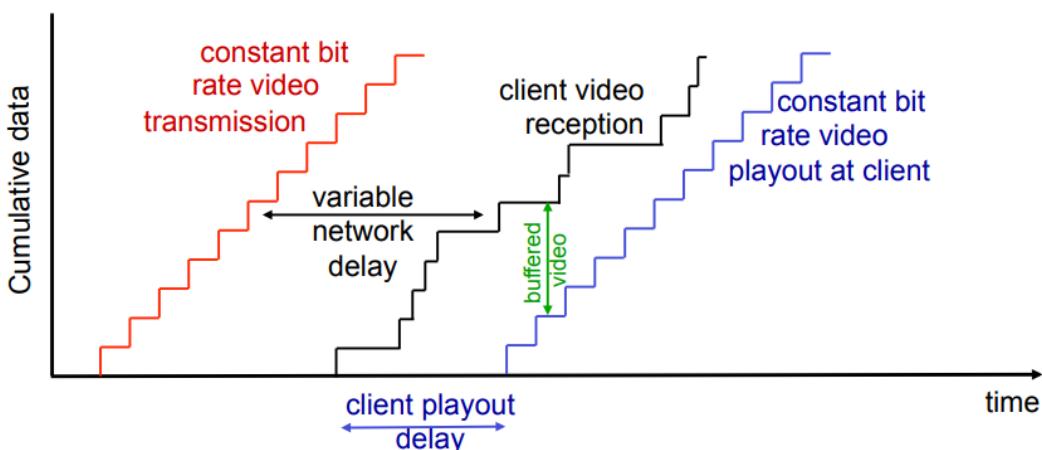
Quando invece la comunicazione inizia dall'esterno, non si genera la tabella del nostro "home gateway" con dentro l'host specifico (perché non è lui ad avere iniziato la comunicazione); perciò una soluzione potrebbe essere compilare la tabella a mano ["port forwarding"] oppure il "relaying" (usato in Skype; host chiede a nodo esterno di fare da relay [specchio] e anche client chiede a quel nodo di fare da relay, dunque riescono a comunicare).

# 11) MULTIMEDIA

Come abbiamo già visto, il segnale analogico audio viene **campionato** (telefono  $f_c = 8000\text{Hz}$ , wav  $f_c = 44100\text{Hz}$ ) e viene **quantizzato** (solitamente 8bit); il segnale video invece veniva digitalizzato digitalizzando i singoli frame come jpeg, ma oggi viene codificato riducendo la “**ridondanza spaziale**” (ovvero pixel vicini con lo stesso colore) e la “**ridondanza temporale**” (codificare i frame del video come differenza rispetto al frame precedente, e non come immagine isolata; questo perciò richiede che il primo frame di una sequenza di frame simili sia codificato con un maggior numero di bit rispetto ai seguenti, quindi genera una codifica “**VBR**” [Variable Bit Rate]).

⚠ Ricorda che “**STREAMING**” = si può visualizzare il video scaricato durante il trasferimento, “**DOWNLOAD**” = scarico il video e poi lo guardo, “**LIVE STREAMING**” = video catturato e trasferito contemporaneamente, “**CONVERSATIONAL**” (o INTERATTIVO) = esempio sono le videochiamate, dove deve essere presente interazione, quindi un delay (ritardo) bassissimo.

Ma come avviene lo **STREAMING STORED VIDEO** (ovvero streaming di video memorizzato sul server, al client) **IN RETE**? Il server manderà al client il video tramite la rete, ma, tra quando il video viene inviato e il client lo riceve, è introdotto un ritardo da parte della rete (**NETWORK DELAY**); questo non è un problema se il ritardo è costante, ma il network delay è **variabile** (dipende dalla congestione della rete). Come risolvo? Si pone prima del client un “**PLAYOUT BUFFER/DELAY**”, ovvero un buffer che mi compensa il network delay (funziona solo se il JITTER [ovvero la variazione del ritardo] non è troppo elevato).



Per lo streaming multimediale si usava sempre UDP (ritardo end-to-end più basso, quindi posso usare un playout buffer più piccolo), ma non ci sono ritrasmissioni e spesso UDP non passa attraverso i **firewall** (blocca le intrusioni nella rete). Quindi si preferisce TCP con **HTTP** (che passa nei firewall), ovvero mandare questi multimedia come se fossero oggetti web; il problema è che il **bitrate non è costante** a causa delle ritrasmissioni TCP, perciò servirebbe un playout buffer più grande. Proprio a causa di ciò abbiamo il **DASH** (**Dynamic Adaptive Streaming over HTTP**): il server divide il file video in chunk, ognuno codificato ad un rate diverso, e c'è il “manifest file” (dà gli URL dei chunk in cui è stato diviso il video); il client invece misura la banda, consulta il “manifest file” e richiede un chunk alla volta (quello con codifica massima rispetto alla banda misurata attuale).

Per quanto riguarda il **LIVE STREAMING**, l'obiettivo è dare all'utente la stessa esperienza della TV (bassi ritardi, giusto throughput, bassa perdita di pacchetti); per farlo abbiamo 2 opzioni:

- **UDP-based**: meglio per delay, peggio per recupero delle perdite di pacchetti;
- **TCP-based**: meglio per recupero delle perdite di pacchetti, ma con DASH si aggiusta anche delay e throughput.

**Ma come si mandano i contenuti a migliaia di utenti contemporaneamente?** Potremmo usare un “mega-server”, ma questo non scala (errore in un punto = errore ovunque, grossa congestione, lunghe distanze dagli utenti, copie multiple del video). Quindi si usano le **CDN** (**Content Distribution Networks**), ovvero si memorizzano copie multiple del video in punti distribuiti geograficamente, per garantire un accesso distribuito ai vari utenti geolocalizzati in punti diversi.

Ora parliamo invece del **VoIP** (**Voice-over-IP**), ovvero le **chiamate tramite Internet** (utili per features aggiuntive, riduzione dei costi, chiamate gratuite [no circuito, ma pacchetto], basate su **UDP**). Richiede un **basso ritardo** end-to-end per mantenere l'**interattività** (<150ms = bene, >400ms = male), includendo il ritardo di **rete**, del **playout** buffer e anche quello di **pacchettizzazione**. Qui a differenza della telefonia tradizionale, la **segnalazione** (ovvero quando l'utente manda la richiesta di contattare un altro utente) non coinvolge i nodi intermedi, ma se ne occupa

la procedura di ROUTING IP che abbiamo visto nel capitolo dell'IP (infatti qui siamo a livello applicazione) [il protocollo che studiamo per l'inizializzazione per le VoIP è il **SIP**]. Inoltre, ancora più che nei video, nella voce si ha una **tolleranza alle perdite** (1-10%).

▲ Una particolare VoIP è **Skype** (livello applicazione proprietario, quindi non standard): è composto da "Super Nodes" (supernodi con IP pubblico e funzioni speciali), "Clients" (i peers che si connettono tra loro per le chiamate VoIP), "Login Server" (tiene traccia di dove sono i vari clients/peers, quindi una sorta di tracker) e "Overlay Network" (posizionata tra i supernodi per localizzare i clients).

Il protocollo **RTP** (Real-Time Protocol) specifica la struttura del pacchetto che porta **dati audio e video**, ovvero fornisce supporto alle applicazioni "real-time" [si occupa dei **dati**, siamo in **UDP**]. Il suo **header** ha struttura:

<i>payload type</i>	<i>sequence number type</i>	<i>time stamp</i>	<i>Synchronization Source ID</i>	<i>Miscellaneous fields</i>
---------------------	-----------------------------	-------------------	----------------------------------	-----------------------------

- **Payload type** = indica il tipo di codifica usata;
- **Sequence number** = incrementato ogni pacchetto RTP mandato (rileva la perdita di pacchetti);
- **Time stamp** = tempo a cui quel campione è stato campionato (per sincronizzazione tra TX e RX);
- **Synchronization Source ID (SSRC)** = identifica la sorgente RTP (ogni stream nella sessione RTP ha SSRC ≠) [utile al mixer, ovvero il nodo centrale che mixa i segnali provenienti da più sorgenti contemporaneamente e mandarle a tutti i nodi (come multicast ma non multicast, in quanto siamo a livello applicazione)].

Il protocollo **SIP** (Session Initiation Protocol) offre una soluzione standard per l'**inizializzazione della chiamata VoIP** [si occupa del **controllo**, siamo in **UDP**]; fa squillare il telefono del chiamato, si occupa della codifica della chiamata e della fine della chiamata. Si occupa della gestione della chiamata (aggiungere video alla chiamata, cambiare la codifica durante la chiamata, invitare altri utenti, trasferire e mettere in attesa chiamate). SIP è un protocollo **testuale** (come HTTP).

Se voglio chiamare qualcuno, ma ho solo **username o email**, devo riuscire ad ottenere il suo IP; lo posso ottenere attraverso il dominio SIP (preso nella fase di registrazione dell'utente al SIP server). Quindi immaginiamo di voler chiamare qualcuno fuori dal proprio dominio (es. io sono nel dominio @polito.it, mentre il chiamato è in @unito.it): passo da un server DNS per raggiungere il SIP server di quel dominio; una volta raggiunto il SIP server di quel dominio, questo farà da **indexing table** (ovvero avrà salvato la relazione "email-IP" dell'utente) e potrà quindi **contattare il chiamato**. Un problema **analogo** avviene per lo scambio di email tra utenti di domini diversi (**STMP**), dove viene usato il "type MX" (ovvero mail exchange) che specifica il nome del server di posta (es. MX gmail.com mail.google.com).

La **registrazione** (fase in cui viene salvato nel SIP server il dominio, l'email e l'IP dell'utente) avviene con la funzione "**registrar**" (per la sicurezza di base viene usato il metodo della "firma digitale" con hash). Altra funzionalità che implementa è il "**proxy**" (intermediario per la comunicazione tra chiamante e chiamato, ovvero il server intermedio di cui non so l'esistenza che mi fa da tramite [lo abbiamo visto nelle web cache]).

Sapendo che IP è "best-effort" (ovvero se si perde il pacchetto si perde e basta), **come i multimedia vengono supportati dalla rete?** Ci sono diverse soluzioni di **SCHEDULING**:

- **Diff Serv** = differenziare il tipo di traffico (per esempio marcare come prioritario il traffico multimediale, in modo da evitare il queueing);
- **Int Serv** = tentativo di gestire i flussi Internet (rete a pacchetto) come se fossimo in una rete a circuito (rete integrata per i flussi);
- **Nulla** = viene usata questa su Internet, ovvero traffico trattato equamente (non differenzio), non supportato dalla rete ("making the best of best effort"), ma gestito tutto a livello applicazione.