

RETI DI CALCOLATORI E INTERNET

Giulio Federico



Mi chiamo Giulio Federico e ho deciso di condividere i miei appunti, frutto di un mese di studio (senza aver potuto seguire alcuna lezione) per un semplice motivo: venire in conto a chi non potrà frequentare le prossime lezioni per via del Covid-19.

I miei appunti saranno un **supporto**. Infatti il mio consiglio rimane sempre quello di studiare da chi ne sa più (libro di testo) e di utilizzare i miei appunti solo come supporto per capire gli argomenti che dovrebbero poi essere integrati meglio con il libro di testo "Reti di calcolatori un approccio top down sesta edizione Kurose-Ross". Con questo non voglio dire che studiando da questi appunti non si possa passare l'esame, anzi confido che possiate pure prendere un bel voto anche studiando solo da questi appunti, ma se voleste capirne ancora di più dovreste sempre dare un occhiata parallelamente al libro di testo perché potrei aver sbagliato o dimenticato qualcosa.

Detto questo a voi la scelta. Gli appunti riassumono più di 900 pagine di testo e sono divise in teoria e pratica (laboratorio). Non fatevi ingannare dalle circa 300 pagine. Come potrete notare ci sono tantissime immagini che io ritengo vitali alla comprensione dell'argomento. Quindi vi posso assicurare che di effettive dovete studiare solo un 150 pagine, non male no?

Gli argomenti sono spiegati in modo davvero chiaro e ringrazio soprattutto il canale youtube <https://www.youtube.com/channel/UCJQJ4GjTiq5lmn8czf8oo0Q> per la chiarezza espositiva (seppure basilare ma utile per iniziare) degli argomenti e il politecnico per alcune video lezioni molto chiare.

In allegato, vi lascio anche il **progetto 2019-2020**.

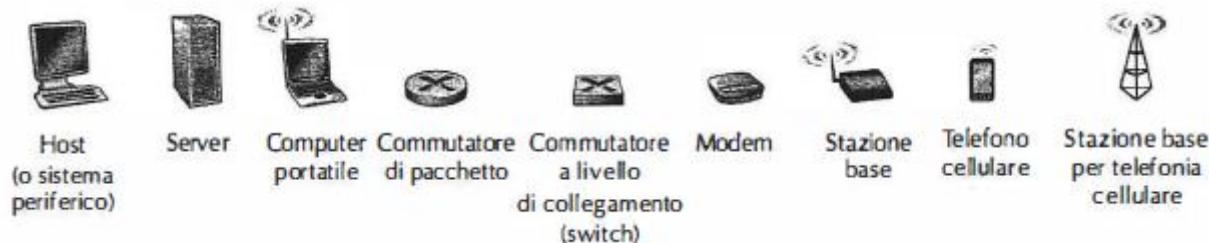
Buono studio e auguri.

| CAPITOLO 1



Cos'è Internet?

Internet è una *rete di calcolatori* che connette tantissimi dispositivi, detti **host** oppure **sistemi periferici (end system)**.



Talvolta gli host vengono suddivisi in **client** e **server**. I client effettuano le richieste e i server "servono" la richiesta inviando i dati. I server stanno nei **data-center**. Per esempio Google ha più o meno 50 datacenter e in totale più di un centinaio di migliaia di server raggruppati in questi datacenter.

I collegamenti vengono detti **rete di collegamenti** o **commutatori di pacchetti(packet switch)**, e possono essere di diversa natura (cavi di rame, fibre ottiche, onde elettromagnetiche) e per questo hanno diverse **velocità di trasmissione (transmission rate)** misurata in bit/s (**bps**).

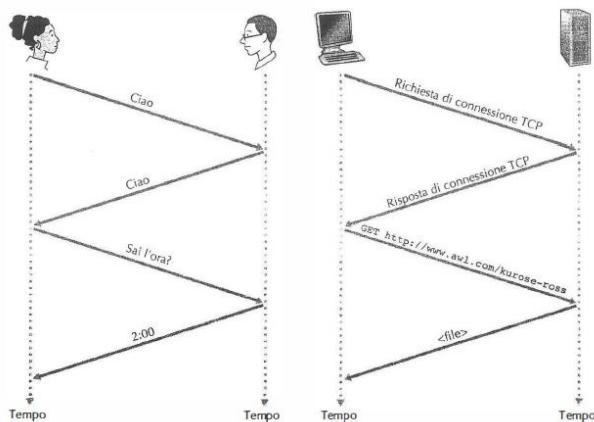
Per essere inviati i dati vanno divisi in sottoparti e ciascuno etichettato con un header; ciascuno di esso viene detto **pacchetto**. Tutti i pacchetti che arrivano a destinazione vengono poi riassemblati per formare il dato in origine. Il percorso tra l'inizio e la destinazione, comprensivo di tutti i collegamenti e dei **commutatori di pacchetto** viene detto **percorso (path)**. Il pacchetto procede verso la destinazione come fosse un camion che trasporta i dati, le strade sarebbero i collegamenti, i commutatori gli incroci e i palazzi attorno i sistemi periferici.

I sistemi periferici, che possono accedere a internet tramite i cosiddetti **Internet Service Provider (ISP)**, insieme a tutta l'architettura di rete detta fanno uso di protocolli come **TCP** o **IP** che controllano il singolo pacchetto.

Internet fornisce anche il suo servizio alle cosiddette **applicazioni distribuite** (email,navigazione sul web, social network), ossia sistemi periferici che, collegati appunto a internet, si scambiano dati reciprocamente.

Spesso quando da un sistema periferico si richiedono dei dati da un altro sistema periferico, il quale avrà magari il suo modulo software descritto con un linguaggio totalmente diverso da quello del mittente, si devono utilizzare le cosiddette **API Internet** ossia "come deve essere fatta una certa richiesta da X perché Y la possa comprendere".

Cos'è un Protocollo?



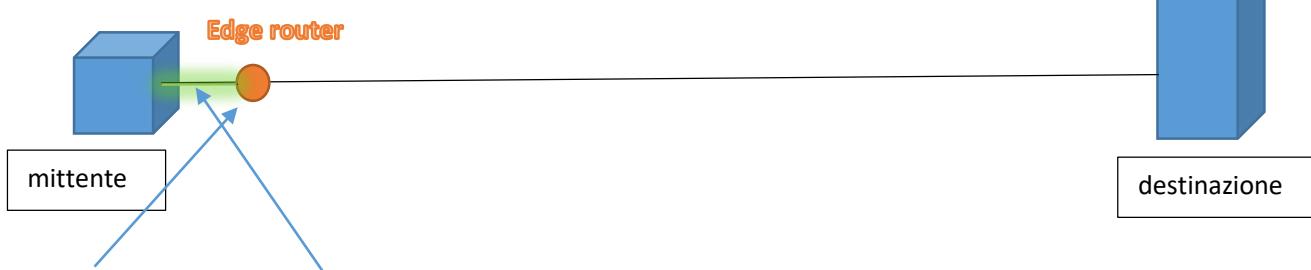
Per comprendere cosa sia un **protocollo** facciamo un'analogia. Una persona A per chiedere l'ora a un'altra persona B deve adottare un certo *comportamento*. Deve prima salutare, e se B saluta (ossia risponde con lo stesso comportamento) allora la comunicazione viene fatta. Un protocollo non è altro che un comportamento al quale sottostare se due entità vogliono comunicare tra loro. Inoltre il protocollo deve essere lo stesso tra le due identità (il protocollo

educato non andrebbe bene con il protocollo maleducato, perché se A saluta e B gli dice 'non rompermi' allora probabilmente A non continuerà la comunicazione con B).

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

Le reti di accesso

Il percorso di rete è rappresentabile come:



Il primo router che si incontra nel percorso è detto **edge router (router di bordo)**. Esso sarebbe quello a cui si connette il sistema periferico per "entrare nel mondo dell'internet". A connetterli fisicamente ci pensa la **rete di accesso**, ne esistono diverse. Quindi nota bene che il router non è sempre quello a cui ci collegiamo, ma il tramite ultimo con cui la nostra rete si interfaccia con quella esterna.

Accesso residenziale

Ne abbiamo di due tipi:

- **DSL (Digital Subscriber Line):** in genere c'è una compagnia telefonica che fornisce il servizio di telefonia fissa. Però *utilizzando la stessa linea telefonica* detta anche *doppino telefonico* in rame che collega l'abitazione con la centrale locale si riesce, tramite un **modem DSL** che comunica con una **DSLAM** (sempre dentro la centrale locale), a usarlo per una connessione internet. In questo caso la compagnia telefonica fa anche da ISP.

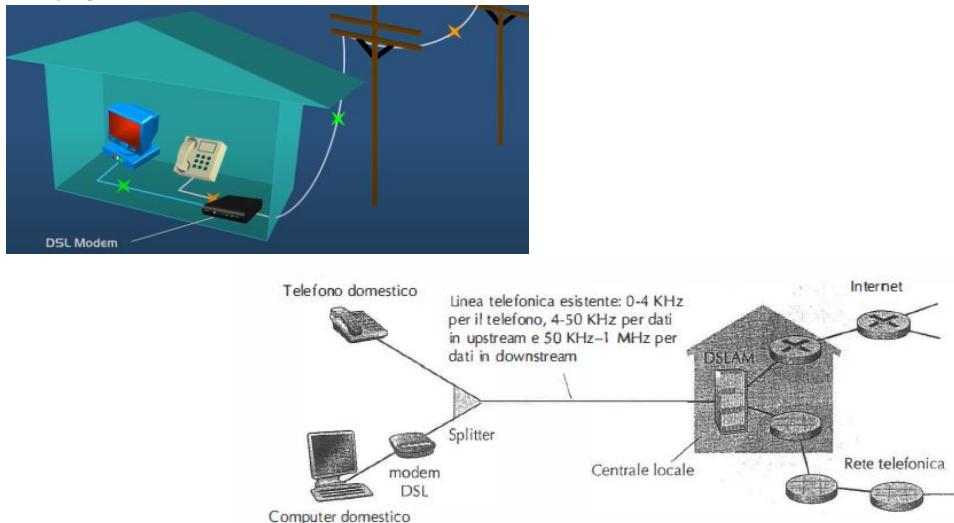
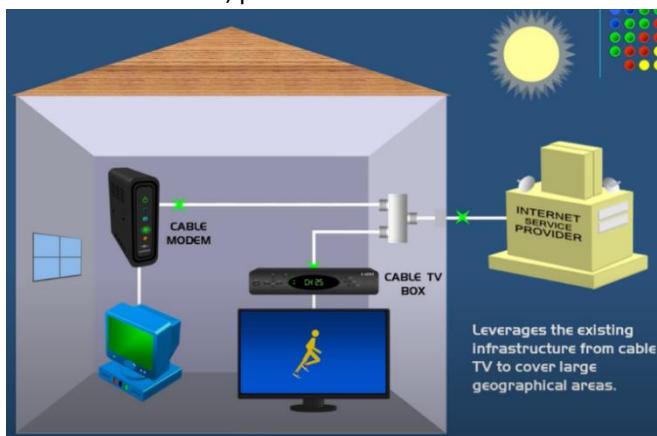


Figura 1.5 Accesso a Internet tramite DSL.

Il computer domestico invia dati digitali al model DSL (collegato dietro con la linea telefonica) che li converte in *toni ad alta frequenza* che viaggiano sulla linea telefonica per poi arrivare al DSLAM ed essere riconvertiti in formato digitale. Questo flusso dati tra client (modem DSL) e server (Centrale Locale) viene detto **upstream**, mentre il contrario viene detto **downstream**. L'accesso DSL viene detto **asimmetrico** perché spesso questi due flussi sono a velocità diverse, spesso privilegiando il downstream in quanto più spesso si scarica piuttosto che caricare dati. Quando è asimmetrico viene detto **ADSL** mentre quando è simmetrico si dice **SDSL**. Per distanze superiori ai 6 chilometri tra client e server *non è* una buona scelta la DSL, ma per brevi distanze risulta veloce, nessuna condivisione con altri ed economico.

- **Internet via cavo:** in questo caso non si sfrutta l'esistente linea telefonica ma il cavo che l'azienda fornisce al client per usufruire del servizio di televisione. Questo cavo può essere usato, sempre facendo richiesta alla stessa, per connettersi a internet.



L'ISP o l'azienda con la quale si sta usufruendo del servizio di televisione, fornirà essa stessa il **modem O cable modem**, oppure un **gateway** che include un modem, uno switch e un wifi router in un solo strumento come in figura.



Quando ricevi il gateway basta collegargli dietro lo stesso cavo che si usa per ricevere dati per la televisione.

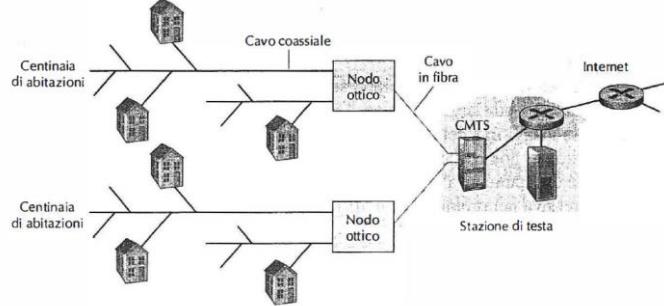


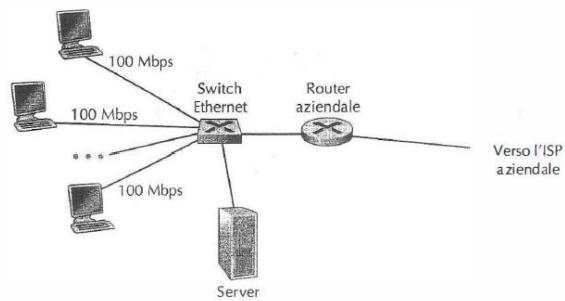
Figura 1.6 Rete di accesso ibrida a fibra e cavo coassiale.

Nei sistemi **HFC** (ossia un sistema che impiega sia la fibra ottica che il cavo coassiale) la fibra ottica viene usata nei tratti più lunghi e i cavi coassiali per le abitazioni. La ramificazione (da fibra a cavi coassiali) avviene nei **nodi ottici**. Questi nodi servono a superare il limite della connessione via cavo che invece non ha la DSL. Ogni nodo ottico si ramifica per servire dalle 500 alle 2000 abitazioni. Ciascun nodo poi convoglia tutto verso un unico **CMTS** (*cable modem termination system*) che si occupa di tradurre i segnali analogici in segnali digitali e divide il canale in uno di upstream (30,7 Mbps) e uno di downstream (42,8 Mbps). Lo svantaggio degli HFC è che ogni utente dovrà condividere il traffico e lo stesso canale con altri, quindi in situazioni di picco, ne risentirà molto la connessione.

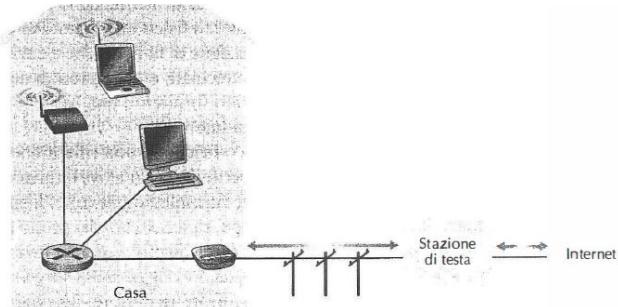
Similmente alla DSL anche qui esiste un modem chiamato cable modem al quale ci si connette fisicamente tramite *ethernet*.

Accesso aziendale(e anche residenziale)

Internet è la rete delle reti. Cosa si intende per LAN? E' un acronimo che sta per *Local Area Network*. Se abbiamo diversi computer, e collegiamo questi tramite ethernet a uno *switch* tramite *lan ethernet* (ossia un collegamento in rete locale tramite fili) abbiamo creato una rete locale. Ovviamente questo switch servirà solo per veicolare in modo bidirezionale il traffico con il router aziendale, che è sempre collegato a Internet tramite uno dei modi visti sopra.



In questo caso si ha comunque un collegamento che necessita di fili. Si possono collegare alcuni di quei pc in figura anche senza fili, in modalità *wireless* e si dice tramite *lan wireless*, ossia un collegamento senza fili allo switch.



In realtà i computer si collegano a un **access point** (che sarebbe un ripetitore), che ovviamente va comprato dopo aver comprato i router in quanto si collega con questo, trasformando la rete camblata in rete wireless.

I MEZZI TRASMISSIVI

Si dividono in due fasce: **mezzi vincolati** in cui le onde vengono contenute in mezzi fisici, mentre nei **mezzi non vincolati** le onde si propagano nell'atmosfera e spazio esterno.

Mezzi vincolati

Sono mezzi vincolati:

- **Doppino di rame intrecciato:** è il meno costoso e il più usato tra i mezzi vincolati. E' usato principalmente nelle reti telefoniche, LAN e sarebbero due fili di rame di dimensione minore di 1mm intrecciati a spirale per evitare interferenze.
- **Cavo coassiale:** è costituito da due conduttori di rame, ma stavolta concentrici. A differenza del doppino non risente quasi di alcuna interferenza grazie a questa struttura ma anche grazie a uno speciale isolamento. Pertanto può raggiungere elevate frequenze di trasmissione e può inoltre essere condiviso tra più sistemi periferici.
- **Fibra ottica:** è un cavo costituito da filamenti vetrosi che conduce impulsi di luce, che sarebbero i bit. Sono velocissimi, permettono addirittura il trasporto di 10-100 Gigabit/s. Sono utilizzati per le grandi distanze perché oltre a essere immuni alle interferenze hanno anche segnali debolmente attenuati durante tutto il percorso. Per il costo elevato del materiale e dell'impianto necessario non sono la soluzione per le reti locali.

Mezzi non vincolati

Sono mezzi non vincolati:

- **Canali radio terrestri:** sono vie o propagazioni di segnale all'interno dello spettro elettromagnetico. Dipendono molto dall'ambiente circostante e dalla distanza al quale il segnale deve essere inviato.

- **Canali radio satellitari:** qui si usano due tipi di satellite: geostazionari o a bassa quota. In ogni caso il satellite riceve trasmissioni su una certa banda di frequenza, rigenera il segnale e lo trasmette a un'altra frequenza.

Il nucleo della rete

Commutazione di pacchetto

Dalla sorgente alla destinazione abbiamo visto possono esserci collegamenti e commutatori di pacchetti (switch o router). Per **commutazione di pacchetto** si intende la suddivisione dei messaggi lunghi in parti più piccole note come **pacchetti**. Se un pacchetto è formato da L bit e la velocità da una sorgente al commutatore è R bps allora abbiamo che un pacchetto viene trasferito in $\frac{L}{R}$ secondi.

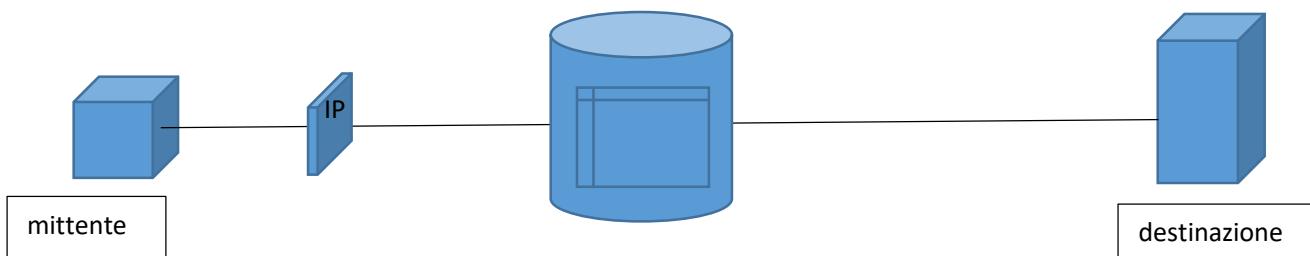
I commutatori si interpongono tra la sorgente e la destinazione e la maggior parte di loro NON trasmette i singoli bit di un pacchetto subito appena li ha ricevuti, ma aspetta di avere tutti gli L bit del pacchetto prima di inviarlo. Questo tipo di trasmissione è detta **store-and-forward**, cioè memorizza e poi inoltra.



Supponiamo che la sorgente debba inviare un messaggio alla destinazione. Lo suddivide in tre pacchetti. Invia il primo. Al commutatore arrivano i primi bit degli L del pacchetto e li continua a memorizzare fino a quando non ha l'intero pacchetto, ossia dopo $\frac{L}{R}$. Lo stesso tempo si spende dal commutatore alla destinazione. Quindi il tempo totale per l'invio di un pacchetto è di $\frac{2L}{R}$.

Per ogni collegamento (nella figura solo uno) il router ha un **buffer di output** (o **coda di output**) in cui accoda i vari pacchetti che devono essere trasmessi in uscita su quel collegamento. Supponiamo che il pacchetto 1 come prima è però già arrivato al router. Adesso il router lo trasmette bit per bit alla destinazione. Nel frattempo però arriveranno in entrata i bit del secondo pacchetto fino al pacchetto intero. Tali pacchetti possono arrivare da più collegamenti in entrata. Allora questi pacchetti si metteranno *in coda* fino a quando non possono essere anche loro trasmessi (dopo che l'intero pacchetto 1 lo è stato) generando un ulteriore ritardo noto come **ritardo di accodamento**. La coda è però finita quindi può succedere che un ennesimo pacchetto in entrata non vi trova posto. In questo caso si ha **perdita di pacchetto**. Questa **congestione** si verifica spesso quando la velocità di trasmissione in uscita è nettamente inferiore alla velocità di trasmissione in ingresso e quando nella stessa uscita vogliono essere trasmessi tanti pacchetti di diverse entrate.

Ma quando un pacchetto arriva al router, come fa questo a decidere su quale collegamento in uscita deve andare?

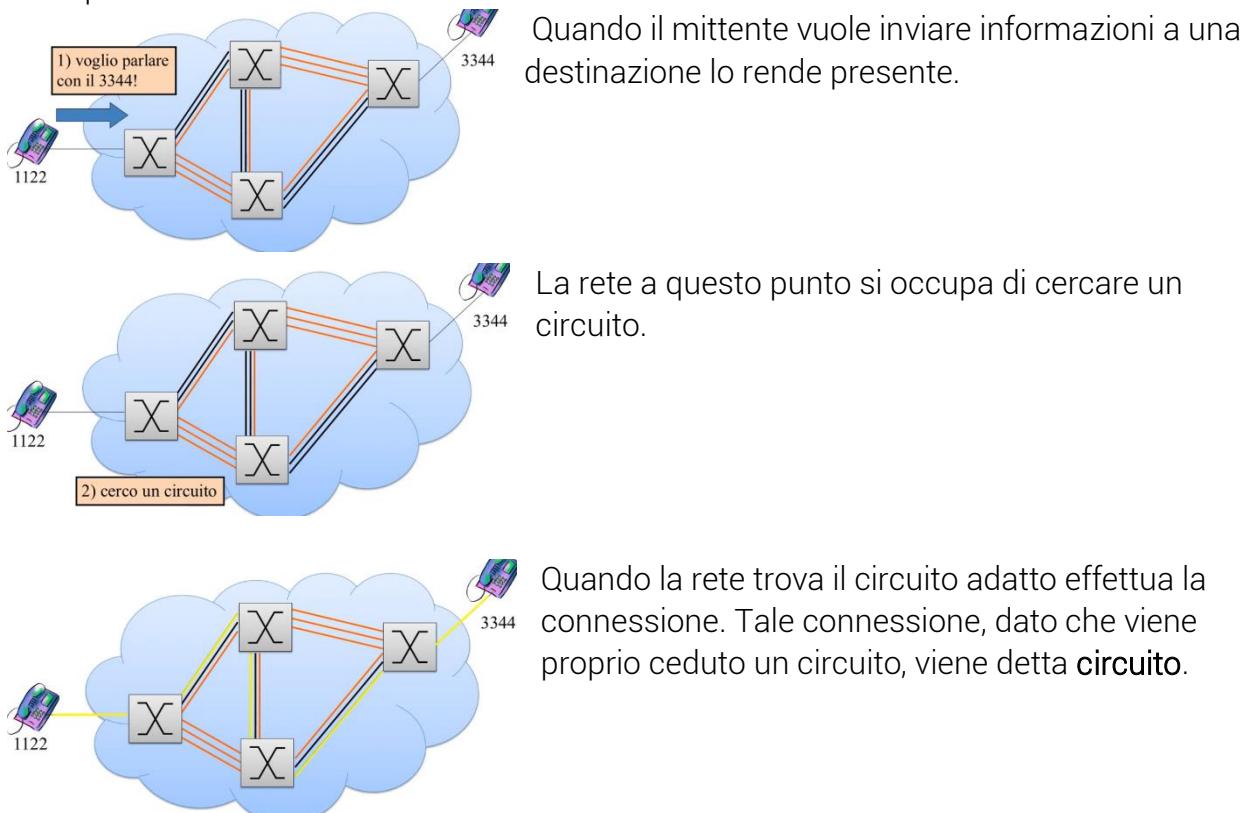


Ogni pacchetto ha nella sua intestazione l'indirizzo IP della destinazione. Il router possiede una tabella detta **tavella di inoltro** che dato un indirizzo IP sa su quale uscita deve trasmettere il pacchetto.

Adesso possiamo capire perché il router debba immagazzinare i bit dei singoli pacchetti prima di inviarlo. Ogni messaggio viene suddiviso in pacchetti ma ogni pacchetto viene inviato bit per bit. Devo quindi avere il pacchetto totale per poterne leggere l'header, solo allora saprà verso dove instradarlo.

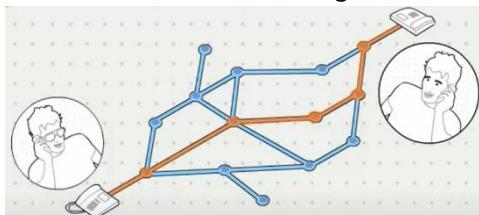
Commutazione di circuito

Abbiamo visto qual è il problema della commutazione di circuito. La *condivisione delle risorse* implica code di attesa e a volte anche perdite. Un modo per risolvere questo è la **commutazione di circuito** le risorse vengono invece riservate. Le reti telefoniche ne sono un esempio.

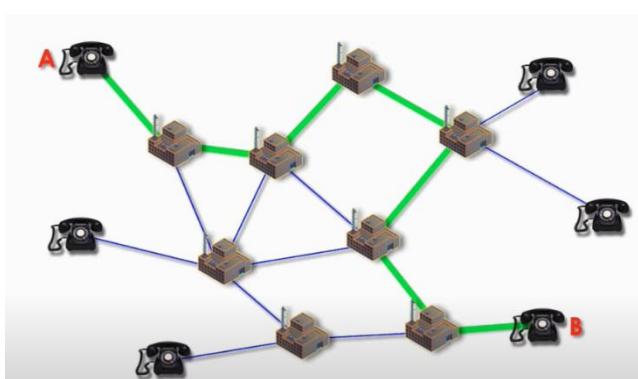




In pratica la commutazione di circuito fa in modo automatico quello che facevano le famose centraliniste degli anni 50'.

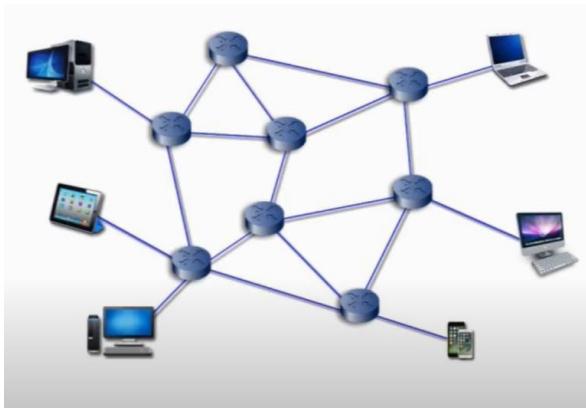


Ogni volta che si vuole inviare un pacchetto a destinazione si sfrutta la rete telefonica che permette di trovare un percorso fisico fisso tra sorgente e destinazione.



Notiamo subito una cosa. La **connessione end-to-end** (da una estremità all'altra) è subito stabilita qui. Quindi ogni commutatore (nb: no router ma commutatore) del percorso sa già dove inviare ogni singolo pacchetto, quindi non c'è alcun bisogno di fare store-and-forward su ogni commutatore. Ciascuna connessione o circuito può essere fatta

tramite due modalità di **multiplexing**: a *divisione di frequenza* o a *divisione di tempo*. Col primo la rete dedica in modo equo a ciascun circuito una banda di frequenza, e ciascuna banda ha una sua **larghezza di banda** che determina la velocità di trasmissione dei dati. La seconda invece il circuito è dedicato per un tot di tempo.

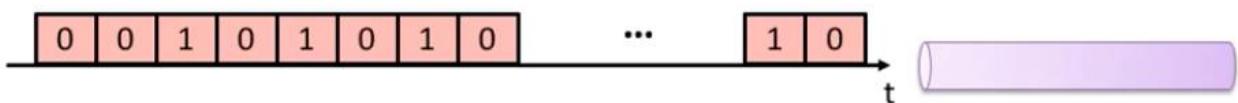


Un esempio di rete a commutazione di pacchetto è l'**Internet**, ossia una rete fatta di commutatori di tipo router. Risulta più efficiente della commutazione di circuito perché anche se quest'ultima garantisce la connessione, le prestazioni promesse, la sicurezza e un tempo costante e deterministico (cioè prevedibile/calcolabile) di trasferimento ha il grosso svantaggio di avere poca flessibilità, infatti difficilmente la velocità di ciascun circuito

può essere aumentata, poi i circuiti, qualora le sorgenti hanno un basso tasso di attività, sono inutilizzati quando invece potrebbero servire ad altre sorgenti e infine sono molto costosi (ecco perché paghiamo tanto di telefonia). La commutazione di pacchetto diventa la scelta migliore ma bisogna trovare una soluzione ai suoi svantaggi che vedremo saranno risolti adottando delle strategie che ne assicureranno l'affidabilità di ricezione (TCP) e la sicurezza (crittografia etc...).

Alcuni concetti

Ogni collegamento o **link** ha una sua **velocità di trasferimento** misurata in **bps** (bit/s) che indica quanti bit al secondo passano nel link.



Spesso però si usano dei multipli:

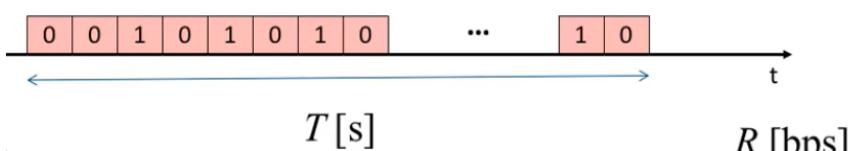
$$1 \text{ kbps (kb/s)} = 10^3 \text{ bps}$$

$$1 \text{ Mbps (Mb/s)} = 10^6 \text{ bps}$$

$$1 \text{ Gbps (Gb/s)} = 10^9 \text{ bps}$$

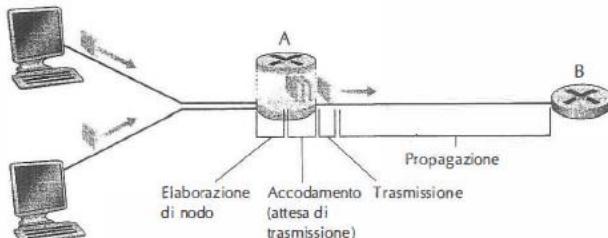
Ogni link ha poi un suo **tempo di trasferimento** ossia il tempo in secondi che il link impiega per trasferire L bits che è data da $T = \frac{L}{R}$

$$L \text{ [b]}$$



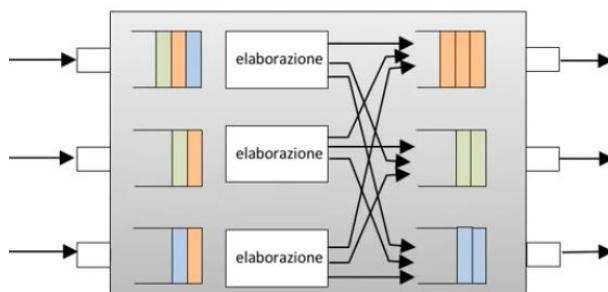
Ritardo nella commutazione di pacchetto

Vediamo in particolare quali sono questi ritardi che in totale formano il **ritardo di nodo** ma che singolarmente possono essere visti come:



- Ritardi di elaborazione
- Ritardi di accodamento
- Ritardi di trasmissione
- Ritardi di propagazione

Figura 1.16 Ritardo di nodo al router A.



Ricordando la dinamica: ogni router ha delle entrate e uscite (dette **schede di rete** o **interfacce di rete**). Deve salvare via via le parti di ogni pacchetto fino a comporlo tutto per leggere l'header. Pertanto esisterà in ingresso una coda per ogni scheda. A pacchetto ricostruito si legge l'IP (elaborazione) e solo adesso può commutare (legge la tabella per vedere la coda di uscita in cui inserire il pacchetto).

Ritardo di elaborazione d_{elab}

Banalmente è il ritardo che si ha nel leggere l'header del pacchetto e consultare la tabella. Tale ritardo però include anche quello dovuto ai controlli per vedere se ci sono stati errori durante il trasferimento in entrata del pacchetto.

Ritardo di accodamento d_{acc}

E' il ritardo che ha il pacchetto una volta messo nella coda di uscita ad aspettare il suo turno per essere trasmesso.

Ritardo di trasmissione d_{trasm}

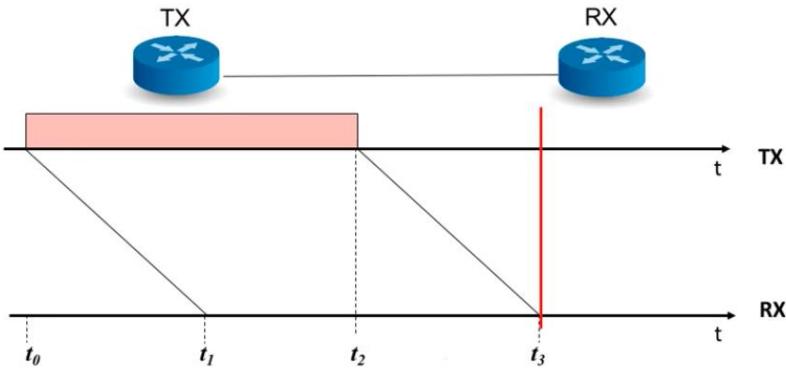
E' il ritardo che ha il link nel trasferire tutti gli L bites che formano il pacchetto a velocità di trasmissione R che sappiamo essere L/R.

Ritardo di propagazione d_{prop}

E' il ritardo che è dovuto al materiale su cui viaggia il singolo bit e si trova come $\frac{D}{v}$ dove D è la lunghezza del link e v è la velocità di propagazione del singolo bit sul link. La velocità di

trasmissione R è diversa da quella di propagazione v. Con un'analogia idraulica si ha che R indica la portata, ossia quanti bit passano al secondo, mentre v indica la velocità con cui ogni bit percorre il link.

Che differenza c'è tra ritardo di trasmissione e di propagazione?



Supponiamo che TX vuole inviare a RX i famosi L bit. Supponiamo che il link abbia velocità di trasmissione R. A tempo t_0 TX invia il primo bit. Questo primo bit viaggia sul link e impiega un tempo di propagazione t_1 . Appena dopo arriva il secondo bit, poi il terzo e così via fino all'ultimo a tempo t_3 . Il tempo di trasmissione sarebbe $T = \frac{L}{R} = t_2 - t_0$ che quindi sarebbe il tempo impiegato idealmente a trasferire tutti gli L bit se non ci fosse il ritardo di propagazione, mentre quest'ultimo è $\tau = t_1 - t_0$ e quindi è chiaro che il tempo totale non è solo dato dal tempo di trasferimento, ma viene traslato dal tempo di propagazione, quindi il tempo totale per trasferire L bit è $T_{tot} = T + \tau$

In pratica più riduciamo la distanza tra i due assi minore è il ritardo di propagazione.

Quindi il **ritardo di nodo** è in totale:

$$d_{elab} + d_{acc} + d_{trasm} + d_{prop}$$

In generale se ci sono N-1 router, trascurando il ritardo di accodamento, si ha un ritardo totale tra la sorgente e la destinazione detto **end-to-end delay** pari a:

$$T_{tot} = N \cdot (d_{elab} + d_{trasm} + d_{prop})$$

Per ottenere una misura più precisa dei ritardi possiamo utilizzare Traceroute.

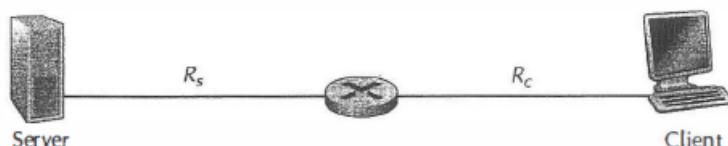
NB: Il tempo di trasmissione indica quanto tempo ci vuole perché il trasmettitore inserisca i bit sulla linea. Immagina che un uomo abbia un cesto di 10 mele e davanti a lui abbia un canale che riesce a propagare ciò che gli si viene messo a velocità v. Se un uomo impiega 5 secondi a inserire ogni volta una mela dentro quel conduttore si avrà quindi un tempo di trasmissione pari a $10 \times 5 = 50$ secondi per inserire TUTTE le mele dentro il canale.

Ovviamente inserita la prima questa viaggerà a velocità v. Quindi il tempo T di trasmissione è il tempo necessario al trasmettitore per inserire TUTTI gli L bit sul canale che poi ha una sua velocità v di trasmetterne uno a uno.

Throughput

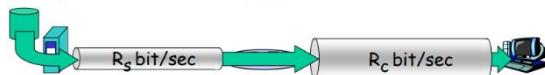
Il **throughput** (pronunciato freepoot) è un'altra misura delle prestazioni end-to-end. Esso indica la frequenza alla quale i bit sono trasferiti tra mittente e ricevente. Non si deve confondere con la velocità di trasmissione. Quest'ultima è caratteristica di ogni link, ma ogni link può avere differenti velocità, mentre il throughput è caratteristico dell'intero collegamento end-to-end. Per fare un'analogia ogni link ha una sua massima capacità che sarebbe la velocità di trasmissione, ossia riesce a trasferire un numero X di bit al secondo; però in ingresso possono arrivarci meno bit e quindi effettivamente in un secondo trasferisce quelli che realmente ci sono per essere trasferiti. Quindi la quantità di informazione trasportata in un momento è detta throughput.

Supponiamo questa situazione:

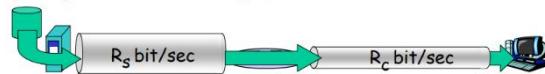


Possono succedere due cose:

$R_s < R_c$ Qual è il throughput medio end to end?



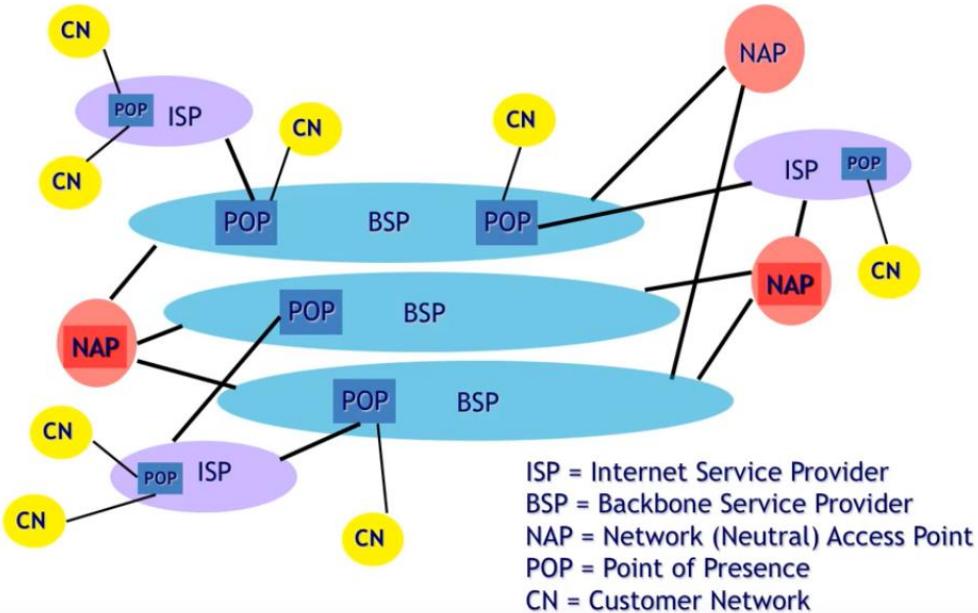
$R_s > R_c$ Qual è il throughput medio end to end?



- nel primo caso il throughput è R_s in quanto per esempio in un secondo arrivano al router R_s bit. Anche se il router in uscita ha un collegamento capace di trasportare R_c bit in un secondo, dovrà comunque portare fuori R_s bit e quindi è come se fosse a velocità R_s .
- Nel secondo caso il throughput è di R_c in quanto anche se al router arrivano in ingresso R_s bit al secondo, in uscita non può portarne tanti ma sempre R_c e quindi è come se sin dall'inizio ci fosse stato un unico collegamento a R_c bit/s.

quanto anche se al router arrivano in ingresso R_s bit al secondo, in uscita non può portarne tanti ma sempre R_c e quindi è come se sin dall'inizio ci fosse stato un unico collegamento a R_c bit/s.

Architettura fisica di Internet



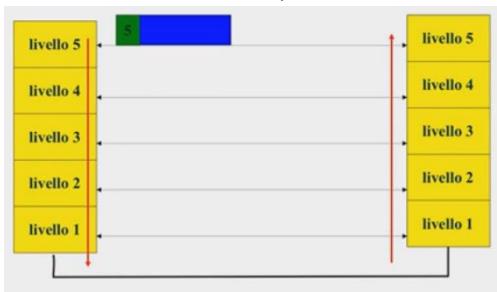
Internet è costituito da tutte queste entità fisiche. Noi ci troviamo sempre nel **customer network** che altro non sarebbe che la LAN ossia una rete di computer collegati a formare una rete e per entrare in questa gigantesca architettura chiamata nel globale Internet la LAN è collegata tramite una **POP** (sarebbe il router) all'**ISP**, ossia chi fornisce internet (vodafone,tim,...insomma colui che ci vende il servizio). A loro volta questi ISP sono collegati tramite sempre un dispositivo di bordo al **BSP**, ossia un ISP che tendenzialmente non ha utenti finali, cioè un fornitore che fornisce servizi non agli utenti finali come noi ma ai vari ISP. Gli **BSP** sono suddivisi per aree geografiche (un **BSP** in Italia, uno in Francia e così via). Questi **BSP** sono poi legati tra loro attraverso dei punti di interconnessione chiamati **NAP**, ossia punti in cui diversi **BSP** condiviscono stessi dispositivi di rete.

Ci si collega al CustomerNetwork tramite i modi visti prima:

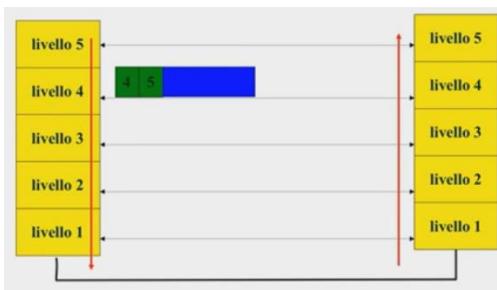
- DSL
- Via Cavo
- Via wireless (reti radio, reti cellulare(LTE))

Architettura a strati dei protocolli

Un servizio di comunicazione tra due entità (es. web browser e web server) viene implementato attraverso una catena di sottoentità che si occupano di supportare il servizio. Questo nella pratica vuol dire che l'informazione che A deve inviare a B viene via via arricchita di altre informazioni dall'alto verso il basso e quando arriva a B, prima di riceverlo effettivamente, fa il processo inverso.

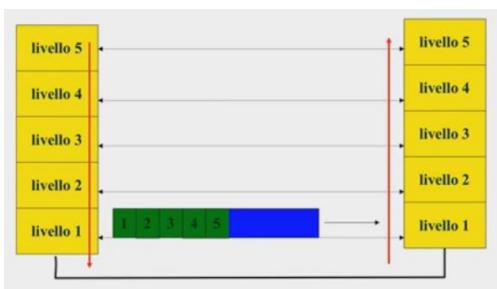


Due entità A e B sarebbero dispositivi raffigurati in toto dalle due pile. Supponiamo che A vuole inviare un informazione a B. Un certo livello preparerà tale informazione con i dati veri e propri e un header specifico di quel livello.



Successivamente questo livello superiore invia l'informazione a quello inferiore che lo arricchisce di un suo specifico header.

Tutto questo succede all'interno della stessa macchina di A.



Alla fine l'informazione arriva al livello 1 e vi arriva arricchita da tutti gli header inseriti dai vari livelli superiori.



Al lato ricevitore si fa il processo inverso. Dal livello 1 fino al livello 5. Ogni entità prende l'informazione fornita dal livello inferiore, prende il proprio header, lo legge e lo passa al livello superiore e così via...

(<https://www.youtube.com/watch?v=aaJ1KcCDz-c> mostra un'animazione molto a grandi linee ma utile per farsi una prima idea)

Quindi architettura a strati vuol dire che l'informazione parte da un livello alto (un processo applicativo), scende lungo una pila e via via viene arricchita con pezzettini di bit chiamati

header. Questo procedimento è detto **incapsulamento**. Il pacchetto di un certo livello viene detto **PDU** e viene incapsulato in altri pacchetti.

Ma perché ogni livello deve aggiungere un certo header?

Ciò che ci sta scritto negli header è un qualche cosa che serve a supportare delle **funzionalità**.



a. **Pila di protocolli a cinque livelli di Internet**

Tutte queste funzionalità sono disposte in questi 5 livelli che sono:

- Applicativo
- Trasporto
- Rete
- Collegamento
- fisico

Livello applicativo

Il livello applicativo contiene tutti quei protocolli di comunicazione che sono più vicini all'utente finale. Sono protocolli applicativi:

- HTTP ---> consente la richiesta e il trasferimento di documenti web
- SMTP ---> consente il trasferimento dei messaggi di posta elettronica
- FTP ---> consente il trasferimento di file tra due sistemi remoti
- DNS ---> consente di tradurre indirizzi di host in indirizzi di rete a 32 bit

Tramite i protocolli applicativi due sistemi periferici si scambiano pacchetti detti **messaggi**.

Livello di trasporto

Prima di arrivare al sistema periferico di destinazione, per tali messaggi si devono stabilire dei protocolli di trasporto. I due protocolli sono TCP, che garantisce sicurezza, affidabilità ma a discapito di troppi controlli che ne rallentano l'invio e la ricezione, e UDP, che evita tutti questi controlli (se non un minimo), quindi non assicura nulla, neppure che un messaggio arrivi a destinazione, a favore però di un più veloce invio del pacchetto.

In entrambi i casi ciascun messaggio viene suddiviso in **segmenti**.

Livello di rete

Il livello di trasporto fornisce al livello di rete ciascun segmento con un indirizzo di destinazione, quindi si occupa solo di stabilire punto di partenza e punto di arrivo del

segmento. Il livello di rete metterà a disposizione il suo servizio di consegna del segmento, detto **datagramma**, verso l'host destinatario. I campi del datagramma e le regole a cui si devono attenere i sistemi periferici nel leggerli sono decisi dal protocollo di rete e il più famoso è detto **protocollo IP**. Il livello di rete contiene inoltre diversi **protocolli di instradamento** detto **routing**, protocolli che scelgono il percorso ottimale di rete da utilizzare per consegnare le informazioni dal mittente al destinatario.

Livello di collegamento

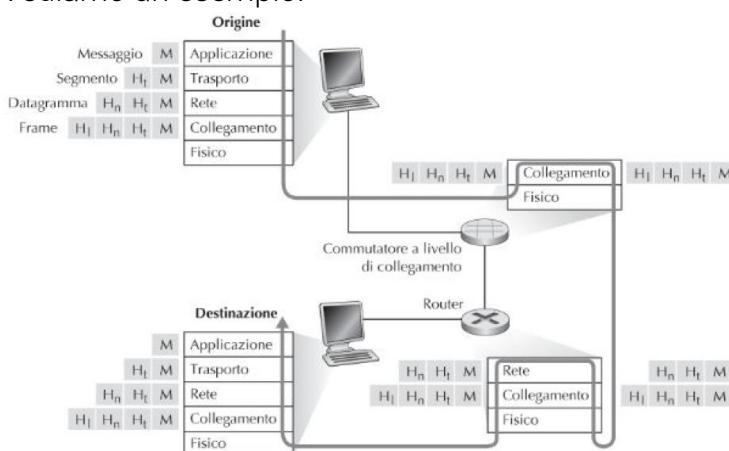
Una volta creato il datagramma, ci si affida al livello di collegamento. Tale livello è quello che decide come e quando inserire nel livello fisico il datagramma. Le tempistiche sono decise da fattori come linea occupata, linea con probabilità alta di collisioni tra pacchetti e altro. Il livello di collegamento, lo vedremo farà molte altre cose.

Livello fisico

Il livello fisico si occupa di mettere i singoli bit dei pacchetti sul canale fisico. Esistono vari protocolli e in genere sono tipici per ciascun tipo di mezzo fisico possibile.

Nota bene. Prima avevo rappresentato i due sistemi A e B come mittente e ricevente, ma nel mezzo ci sono tanti dispositivi e quindi certe informazioni (header) possono essere letti da questi e rigenerati in un certo modo per poi essere rinviati a B o al prossimo dispositivo in mezzo.

Vediamo un esempio:



L'host A vuole inviare un messaggio a livello di applicazione **M** alla destinazione B.

Allora M viene consegnato al livello di trasporto che gli concatena delle informazioni **H_t**. In questo header potrebbero esserci informazioni che il ricevente, nel proprio livello di trasporto potrebbe leggere. Possono essere bit per il rilevamento degli errori.

Il segmento viene passato al livello di rete che concatena a **H_t** un header **H_n** formando il datagramma (che include indirizzi di mittente e destinatario) e così via fino a quando il pacchetto arriva al commutatore che lo fa salire decapsulandolo fino al livello di rete per capire dove deve andare. Modifica/aggiunge l'header e riincapsula il tutto facendolo scendere verso una specifica uscita di rete. Il pacchetto farà così fino a quando arriverà a destinazione e risalendo tutti i livelli (decapsulandosi ogni volta) raggiunge l'aspetto M che aveva in origine quando era stato inviato.



Livello Applicativo

Contiene tutti i protocolli che sono più vicini all'utente finale, ossia a quell'utente che usa le applicazioni per internet (o di rete) come e-mail, World Wide Web, e-commerce, trasferimento file, telefonia su IP (realizzata col protocollo VoIP che permette conversazioni che si avrebbero su reti telefoniche ma utilizzando internet), giochi online, video-conferenze etc...

Queste applicazioni sono software in esecuzione su una macchina (detta anche terminale o host) che vogliono comunicare con un'altra applicazione, quindi un altro software, su un'altra macchina utilizzando la rete. Tutto ciò che sta in mezzo alla rete (router, switch...) detto **core network** non hanno livello applicativo/software. Infatti abbiamo visto che al più si fermano al livello 3 (in realtà vedremo che non è proprio così, ma concettualmente va bene pensarla così).

Quali ingredienti ci vogliono per comunicare con un processo applicativo su un diverso terminale?

- Devo sapere come si chiama → indirizzamento
- Devo sapere come comunicare con lui, quindi che tipi di messaggi posso scambiare, la sintassi e la semantica di questi messaggi e le regole su come e quando inviare e ricevere i messaggi → protocollo

Architetture principali di rete & socket

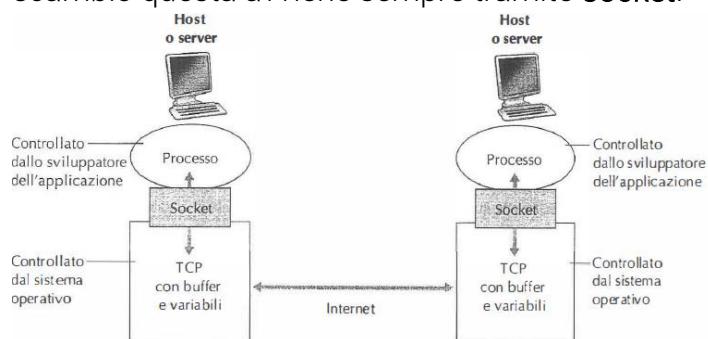
Quando svilupperemo un'applicazione di rete dovremmo progettare l'architettura **dell'applicazione** che in genere è sempre di due tipi:

- **Client-Server:** in questa architettura vi è un host sempre attivo chiamato *server* che risponde alle richieste di molte altri host detti *client*. Nota bene come in questa architettura nessun host comunica con nessuno se non unicamente con il server. Un unico server a volte (per esempio nei social network) non riesce a rispondere a tutte le richieste dei client se questi sono troppi. Si usano quindi dei **data center** che ospitano tanti server (oltre a storage) che virtualmente ne creano uno molto potente.

- **Peer-to-Peer:** qui i server nei datacenter non esistono (quasi sempre) e si sfrutta la comunicazione da host detti *peer* che fungono sia da client che da server. Dato che la comunicazione non passa per il server allora si dice *peer-to-peer* ossia da peer a peer. Uno dei suoi punti di forza è la *scalabilità* in quanto ogni volta un peer si aggiunge a questa architettura, anche se genera carico di lavoro, aggiunge anche capacità di servizio all'intero sistema.

N.B: a volte nelle architetture P2P useremo i termini client e server, ma questo non deve confonderci. Se un peer richiede un file a un altro peer allora il primo si può sempre chiamare client e il secondo server.

Due processi comunicanti si scambiano *messaggi* e qualsiasi sia la direzione di scambio questa avviene sempre tramite **socket**.



Il socket fa da raccordo tra i due livelli: applicazione e trasporto. Nell'esempio per il trasporto si sta usando TCP.

Grazie alla socket si può già definire una mezza idea di come funziona **l'indirizzamento**. Abbiamo detto essere il primo ingrediente perché un processo comunichi con un altro. Affinchè un processo possa comunicare con un altro ha bisogno di:

- Indirizzo dell'host che ospita il processo → **indirizzoIP**
- Identificativo del processo destinatario → **numero di porta**

Siccome i processi comunicano tra loro mandandosi messaggi, e questi messaggi vengono ricevuti/inviati dai/ai socket allora dobbiamo dire su quale numero di porta lavorano questi socket. Quindi ogni processo che può interagire con altri lo fa usando un socket a una certa porta (perché sulle altre magari ci sono altri socket di altri processi).

Servizi principali di trasporto

Abbiamo visto che un processo applicativo spinge fuori dei messaggi verso il destinatario. Ciascuno di loro utilizza dei socket per inviare e ricevere questi messaggi. Ma la responsabilità della consegna di questi messaggi è affidata ai protocolli di trasporto con cui i socket si interfacciano per aver consegnato questi messaggi. Internet mette a

disposizione due tipi di protocolli di trasporto e quando si progetta un applicazione di rete bisogna sempre chiedersi quale dei due protocolli di trasporto adottare:

- **Protocollo TCP:** e' orientato alla connessione, cioè viene istaurata prima una connessione tra l'host e il server tramite scambio di messaggi di controllo tra questi. Effettuato questo handshake di controllo, solo dopo si possono far fluire i messaggi a livello applicativo. Un servizio del genere garantisce un trasporto affidabile (nessuna perdita di dati), un controllo del flusso (il trasmittitore regola la velocità in base al ricevitore) e si controlla la congestione che impedisce di sovraccaricare la rete. Tutto questo a sfavore di alcuna garanzia su ritardi e banda.
- **Protocollo UDP:** non è orientato alla connessione, quindi non esiste alcun handshake tra host e server prima del fluire dei messaggi a livello applicativo che quindi vengono inviati ma non si ha certezza che vengono ricevuti, o perlomeno non nello stesso ordine di invio. Tutto questo a favore però di una elevata velocità di trasmissione. Un protocollo del genere è ideale per applicativi real-time.

Protocollo HTTP

Il WEB è stata la prima applicazione di rete che ha reso Internet la rete delle reti e che ha permesso di farla conoscere anche fuori dagli ambienti universitari dove era unicamente usata negli anni '90.

L'obiettivo finale del protocollo http è di supportare il servizio di web browsing, ossia scambiare oggetti chiamate **pagine web** ciascuna delle quali può contenere oggetti multimediale (file di testo, immagini, pezzi di software...). Ciascuno oggetto (pagina o contenuto della pagina) può essere indirizzato con un URL:

<http://www.nomeHost.it:numeroPorta/oggetto>

In genere la porta viene omessa perché gestita in maniera automatica. HTTP funziona con architettura client-server, quindi esiste il client-http che invia la richiesta (**http-request**) e poi un server-http risponde (**http-response**). L'HTTP è un protocollo **stateless**, ossia il processo server-http non mantiene uno stato delle richieste http che gli sono state inviate, quindi non ricorda la storia. Pertanto se A invia una richiesta http e poi un'altra identica, queste richieste vengono trattate in modo indipendente e il server invia per due volte lo stesso oggetto.

A livello di trasporto si appoggia a uno affidabile quindi al TCP. In particolare vediamo la **comunicazione/set up tra client e server http**:

- 1) Il client-http inizia una connessione TCP verso la **porta 80** del server-http
- 2) Il server-http accetta la connessione TCP dal client-http
- 3) Adesso il client-http e il server-http si possono scambiare messaggi di controllo (o pagine web) e ciascuno invierà (preleverà) tali messaggi attraverso i socket rispettivi

che nel caso di ricezione porteranno a livello applicativo i messaggi, mentre nel caso di invio porteranno al livello di trasporto i messaggi.

- 4) La connessione viene chiusa.

http ha due modi di gestire il punto 3). Il primo modo prevede di scambiare i dati in modo **non persistente**, per cui client e server aprono una connessione TCP per ogni oggetto che il client deve scaricare, in altre parole per ogni oggetto si seguiranno i punti da 1 a 4 e alla fine dell'invio di ogni oggetto il server chiederà al client di chiudere la connessione che poi potrà se vuole riaprire. Il secondo modo, detto **persistente**, prevede una connessione che rimane attiva per tutta la durata della sessione, quindi se il client deve scaricare 10 oggetti, allora la connessione terminerà solo dopo averli scaricati tutti mantenendola attiva per tutta la sessione. Quest'ultima modalità di connessione si distingue ulteriormente in **persistente senza pipelining**, ossia si mantiene attiva la connessione e il client per ogni oggetto fa una richiesta e la successiva la può fare solo dopo che l'oggetto richiesto gli viene consegnato; si dice che le richieste http sono inviate in *serie*; l'altra è **persistente con pipelining** in cui il client può inviare una richiesta http successiva ad una anche se quella precedente non è stata ancora soddisfatta ; si dice che le richieste http sono inviate in *parallelo*.

Esempio di connessione non persistente

Supponiamo di digitare nel nostro client-http il collegamento

<https://www.unipi.it/home/index.html>

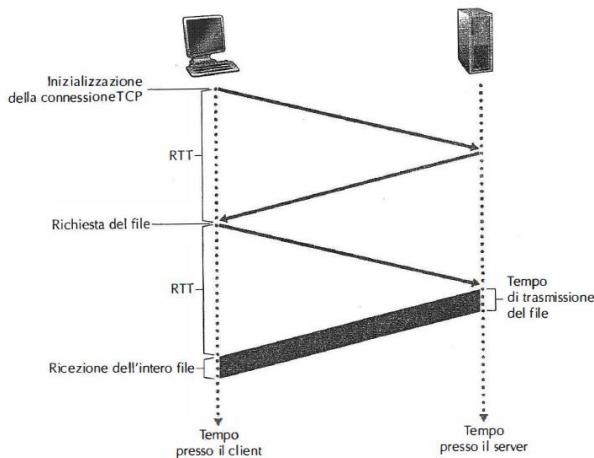
Supponiamo che tale collegamento contenga un file main in formato html e 10 immagini in formato jpeg.

- 1) Il client http apre una connessione TCP verso il server http www.unipi.it sulla porta 80
- 2) Il server http è sempre in esecuzione su www.polimi.it ed è sempre in attesa sulla porta 80. Accetta la connessione o lo notifica al client
- 3) Il client http invia una richiesta http di volere l'oggetto /home/index.html
- 4) Il server http riceve la richiesta http e invia una risposta http contenente il file html.
- 5) Il client http riceve il messaggio di risposta, quindi visualizza il file html. Il server chiude la connessione TCP.

Il client si accorge che il file html possiede altri oggetti jpeg collegati e quindi ripete i passi da 1 a 5 per tutti e 10 gli oggetti.

In modalità *persistente* la connessione rimane aperta.

Ma quanto tempo impiega un file ad essere ricevuto dal momento della sua richiesta?



Che sia persistente o meno, il primo passo è sempre lo stesso: effettuare la connessione TCP. Il client apre la connessione inviando un piccolo segmento TCP al server. Il server accetta di aprire la connessione e lo notifica al client inviandogli un segmento TCP. La terza fase dell'istaurazione di una connessione TCP vuole infine che anche il client confermi il messaggio del server. Questo lo vedremo verrà chiamato handshake a tre vie. Il tempo in cui un pacchetto viaggia dal client al server e dal server al client viene detto **round-trip time (RTT)**. Il terzo passo di una connessione TCP può anche prevedere che si invii non solo una conferma al server su quanto ricevuto (ossia "ho capito che mi hai accettato la connessione") ma anche iniziare a inviare delle richieste. L'HTTP procede con la prima richiesta dell'oggetto /home/index.html proprio in questo terzo passo. Per una connessione persistente ci vorranno RTT di tempo per stabilire la connessione e poi $10 \times (RTT + T)$ per l'invio totale degli oggetti (con T tempo di download). Mentre per una connessione non persistente si avrà $10 \times (RTT + RTT + T)$ in quanto la connessione si apre e si chiude per ogni file da scaricare.

Formato dei messaggi HTTP

Supponiamo che un clienti abbia già stabilito la connessione col server. Come fa a chiedergli che vuole un certo oggetto?



Questo è solo uno dei possibili messaggi inviabili.

Prima riga è detta **request line**:

- campo **metodo** → GET, POST, HEAD, PUT
- campo **URL** → /somedir/page.html indica l'oggetto che voglio
- campo **versione http** → http/1.1 indica la versione del protocollo

Dalla seconda riga si dicono **header lines**:

- **host** → www.someschool.edu indica dove si trova l'host in cui risiede l'oggetto
- **connection** → close indica al server che può chiudere la connessione appena ha inviato l'oggetto richiesto, in altre parole si sta richiedendo una connessione *non persistente*.

- **User-agent** → il tipo di browser che sta facendo la richiesta
- **Accept-language** → fr indica che si vorrebbe ricevere l'oggetto in lingua francese ma se non c'è verrà comunque inviata quella di default.

Un possibile uso di certi campi (a parte quelli ovvi) da parte del server potrebbe essere il seguente. Dallo user-agent il server può anche capire che si tratta di un dispositivo mobile e quindi gli manda quella risorsa però adattata per dispositivi mobile.
tutti i messaggi sono testuali, codificati in ASCII, quindi in formato testuale il messaggio sopra sarebbe:

```
GET /somedir/page.html HTTP/1.1\r\n
Host: www.someschool.edu\r\n
Connection: close\r\n
User-agent: Mozilla/5.0\r\n
Accept-language: fr\r\n
\r\n
```

N.B: il campo host è come se fosse IP-SERVER.

Nota come \r\n servono per andare a capo

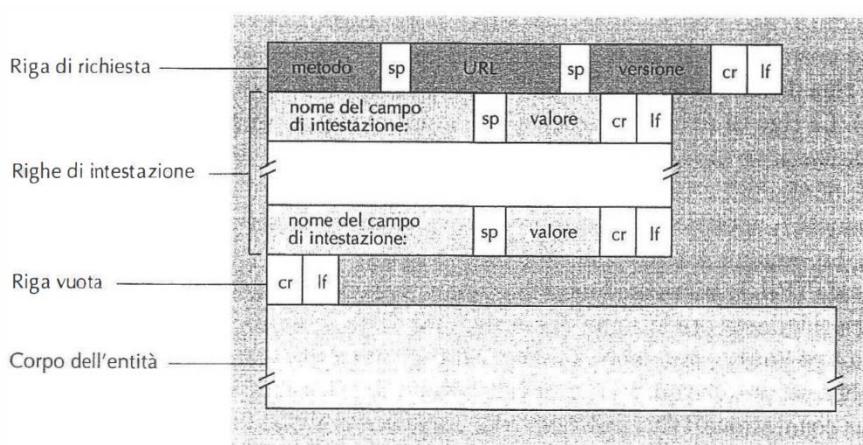
Questo è solo un esempio di messaggio.

NB: Il metodo GET invia la richiesta di una pagina così come fa POST, solo che GET accoda all'URL i parametri che si vogliono inviare al server (di solito &var=1 indica che var è 1) e sono visibili all'utente in quanto sono incorporati nell'URL stesso, mentre POST lascia inalterato l'url dell'oggetto che si vuole ottenere, e mette i dati nel payload del messaggio (vedi dopo). Quindi GET non è fatto per passare dati sensibili e inoltre si possono solo passare dati testuali e di breve lunghezza.

Vediamo in generale come può essere formato:

I messaggi http si distinguono in *richiesta* e *risposta*.

MESSAGGI DI RICHIESTA HTTP



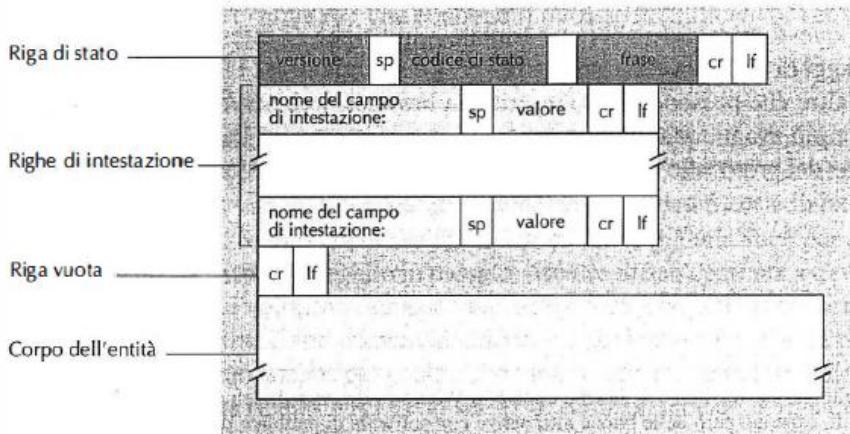
La linea di richiesta è quella già vista.
Sp,cr e lf stanno per spazio,carry flag.
Ogni riga di intestazione ha sempre lo stesso formato:

- Nome del campo
- Valore

La entity body è la parte dati ed è opzionale. Per

esempio potrebbe essere necessaria quando il server, per inviare l'oggetto, richiede una password che gli viene appunto inviata insieme alla richiesta tramite questo campo.

MESSAGGI DI RISPOSTA HTTP



Molto simile a quella di richiesta, solo che la prima riga si dice **status line** e serve per comunicare al client quale è l'esito della richiesta da lui effettuata.

Il corpo dell'entità (payload) stavolta molto spesso è piena e

contiene l'oggetto web richiesto dal client.

Esempio

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data ...
```

il codice 200 è il codice inserito del **codice di stato** quando la richiesta ha esito positivo e Ok è una frase che esplica meglio quest'accettazione della richiesta (spiega meglio il codice 200).

Il primo campo dell'header contiene Date che contiene la data di generazione della risposta,

poi c'è il Server che ha generato la risposta; Last-Modified è usato dal server per comunicare al client quando è stata, sull'oggetto, l'ultima modifica fatta; un altro header per esempio è il content-length che indica il numero di byte contenuti nel data...e così via...

Vediamo in particolare i più comuni **codici di stato**:

200 : La richiesta ha avuto successo e si invia l'informazione

301 (*Moved Permanently*) : l'oggetto che si sta richiedendo è stato spostato su un nuovo URL. Tale nuovo URL si trova nel campo *Location* dell'header di risposta e automaticamente il client rigenera una richiesta su quell'indirizzo.

400 (*Bad Request*) : la richiesta fatta dal client non è stata compresa dal server. In genere si usa anche per un errore generale.

404 (*Not Found*) : l'oggetto richiesto non è stato trovato.

505 (*http version not supported*) : il server non dispone del protocollo http richiesto.

cookie

Li vediamo spessissimo quando navighiamo in rete. Cosa sono? Sappiamo che i server http sono *stateless* cioè non memorizzano nulla. Supponiamo però che i web server abbiano bisogno di identificare gli utenti che navigano tra le loro risorse anche se questi non si sono mai registrati nel sito, magari per consigliargli qualcosa in base alle richieste http fatte in precedenza (che è quello che fanno gli e-commerce per esempio). A questo scopo http adotta i **cookie**. Quando il server riceve una richiesta per la prima volta da un utente che non ne ha mai fatte (oppure ha cancellato i cookie memorizzati) allora il server crea un'entra di indice *id* per l'utente in un proprio database e gli consegna la risposta http con la risorsa che voleva ma nell'intestazione inserisce anche il codice seguente:

Set-Cookie: id

Il browser client, che vede questa intestazione nell'header della risposta capisce che deve salvare una coppia host-id nei cookie che gestisce così che in futuro costruirà le richieste http allo stesso server inserendo l'intestazione:

Cookie: id

In questo modo il server legge dalla richiesta chi è l'utente e accede nel database dove magari ha salvato le sue precedenti navigazioni.

I proxy http (Web Cache)

Non esistono solo proxy http ma anche proxy per altri protocolli applicativi. I **proxy http** (o web cache) nascono per diminuire la latenza nella risposta di un server web nel consegnare la pagina richiesta. L'idea è quella di *spostare* la disponibilità di quella pagina web dal web server a una qualsiasi altra parte che sia più vicina all'utente. Queste macchine particolari che detengono queste pagine sono i proxy. Un proxy è quindi un altro server fondamentalmente, che parla anche lui col protocollo http e mantiene in locale le copie delle pagine del server di origine. Questa sua memoria viene chiamata **memoria cache**.

Vediamo come avviene questa comunicazione. Il browser di ogni client deve essere impostata in modo tale da comunicare sempre col proxy. Quindi è importante notare che il client comunica SEMPRE col proxy.

- 1) Il browser client stabilisce una connessione TCP con il proxy server e invia la richiesta http per l'oggetto specificato.
- 2) Il proxy controlla se ha una copia della pagina richiesta. Se ne ha una allora gliela invia altrimenti contatta lui stesso il server in origine (trasformandosi da server a client) per richiedere tale pagina che una volta ricevuta memorizzerà.
- 3) Consegnà la pagina al client. Qualora il client o qualcun altro la richiedesse lui se la ritrova già senza dover bisogno di contattare il server.

Quando si parla di proxy web o web cache non si può non parlare del **GET condizionale**. Se un proxy detiene una certa pagina di proprietà di un altro web server, è ovvio che deve accertarsi che tale contenuto non sia variato nel tempo altrimenti sta detenendo in cache una copia non consistente.

```
HTTP/1.1 200 OK
Date: Sat, 8 Oct 2011 15:39:29
Server: Apache/1.3.0 (Unix)
Last-Modified: Wed, 7 Sep 2011 09:23:24
Content-Type: image/gif
(data data data data data ...)
```

Per questo motivo quando un proxy riceve dal web server una risposta http con la pagina web, riceve anche una particolare linea chiamata **last-modified** che indica l'ultima modifica effettuata sul file. Il proxy, allora, invia ai browser client che la richiedono la pagina e memorizza questa nella propria cache. Però stavolta affianca a tale pagina anche un campo che riporta l'ultima modifica di valore pari a quella del campo last-modified.

In questo modo il proxy qualche volta contatta il server per domandargli se il contenuto di quella pagina sia cambiato o meno.

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

che esorta il web server a inviargli la pagina se e solo se l'ultima modifica fatta risale ancora a quella data.

```
HTTP/1.1 304 Not Modified
Date: Sat, 15 Oct 2011 15:39:29
Server: Apache/1.3.0 (Unix)
(corpo vuoto)
```

proprio per recapitare il messaggio il più velocemente possibile, oltre che a evitare di sprecare banda.

Per questo motivo quando un proxy riceve dal web server una risposta http con la pagina web, riceve anche una particolare linea chiamata **last-modified** che indica l'ultima modifica effettuata sul file. Il proxy, allora, invia ai browser client che la richiedono la pagina e memorizza questa nella propria cache. Però stavolta affianca a tale pagina anche un campo che riporta l'ultima modifica di valore pari a quella del campo last-modified.

Gli invia un GET condizionale, ossia una richiesta HTTP con un campo **if-modified-since** di valore pari a quello de last-modified

Il web server, qualora non abbia cambiato niente, risponde con un messaggio http che riporta il codice **304** e la frase **Not Modified** per indicargli che la pagina che detiene nella cache è ancora consistente. Inoltre nota come nel corpo non c'è nulla

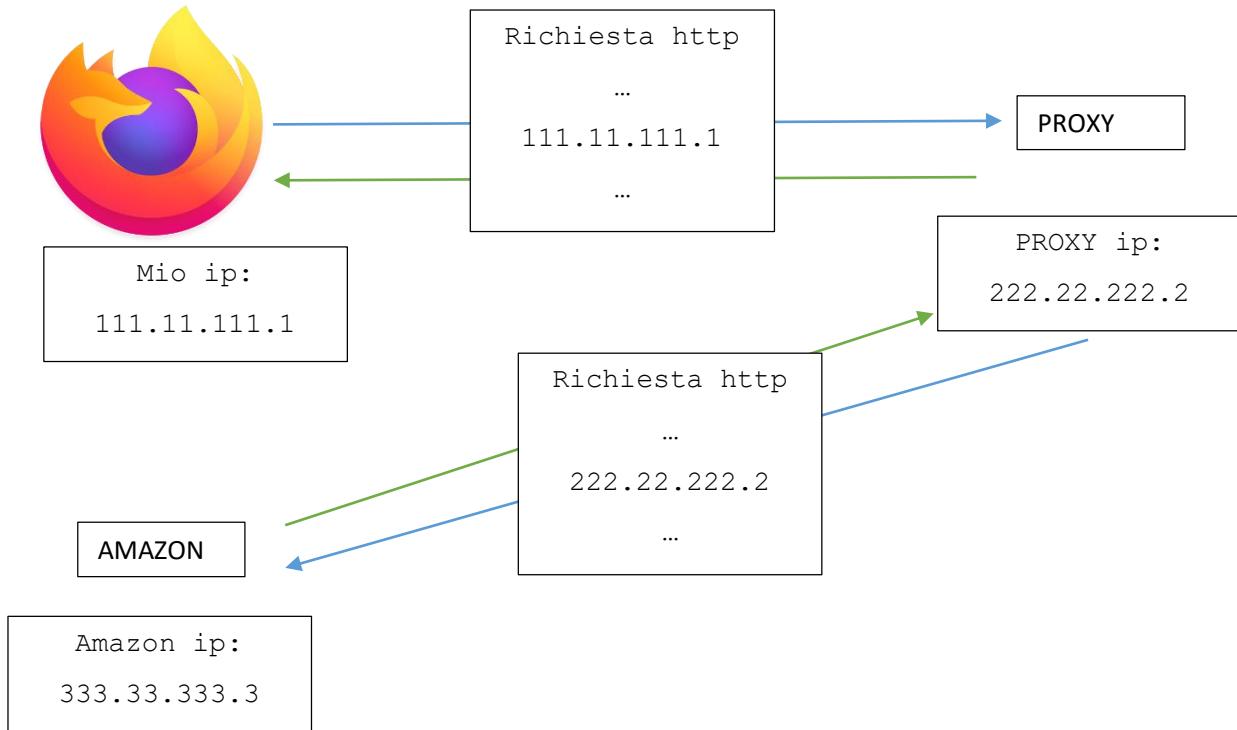
SMANETTIAMO CON I PROXY

I proxy hanno un utilità da hacker. Quando si vuole navigare in modo anonimo spesso si fa uso di una VPN, ma esiste un altro modo. Navigare in modo anonimo vuol dire che quando vogliamo mandare una richiesta http a un server, questo server nella richiesta http legge un ip che non è il nostro. Supponiamo di voler contattare amazon.



Quando farò la richiesta, amazon si accorgerà chi sono.

Se invece uso un proxy, il che vuol dire impostare qualcosa che dica al browser di veicolare le mie richieste a un proxy che le farà ad amazon al posto mio..



Quindi *"usare un proxy equivale a essere fisicamente su un altro computer di IP 222.22.222.2 e fare con quello delle richieste."*

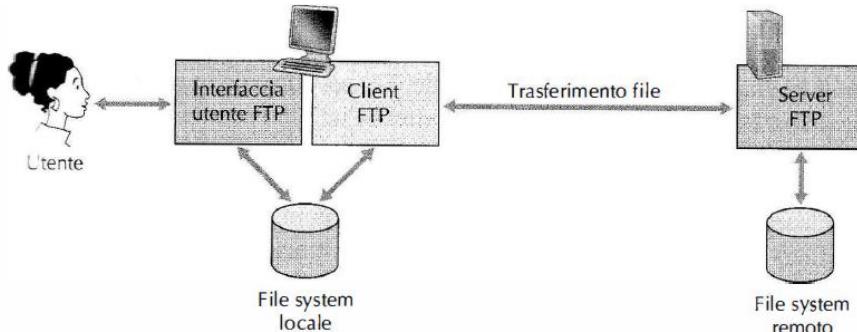
Quindi quando ci vorremmo collegare ad amazon quale indirizzo vedremo nella http response?

Nel primo caso ovviamente 333.33.333.3, perché avendo comunicato direttamente con Amazon è stato lui a risponderci.

Nel secondo caso la response-http avrà la firma di 222.22.222.2 perché abbiamo detto al proxy "fai tu la richiesta ad amazon e quando hai ricevuto la pagina inviamela", pertanto l'handshake http si fa tra me e il proxy.

Il protocollo FTP

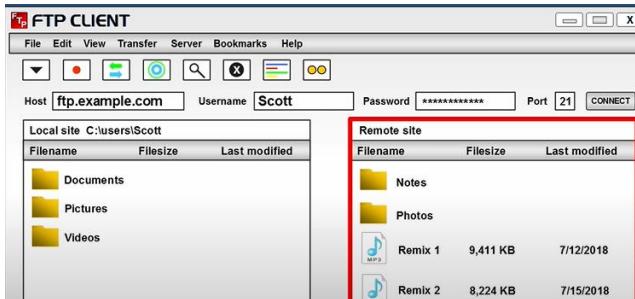
Se HTTP è il protocollo usato per inviare e ricevere pagine web, FTP è il protocollo adottato a livello applicativo per inviare o ricevere file a/da un file system remoto.



Supponiamo di avere un file e permettere a chiunque di scaricarlo tramite internet. Quello che si fa è caricare tale file su un *ftp server* dal quale poi tutti potranno scaricarlo. Ci sono essenzialmente due modi per farlo:



Un primo modo è utilizzare un browser web e scrivere l'indirizzo del server. Nota come non si sta utilizzando http ma ftp nell'url.



Un altro modo è quello di utilizzare un software dedicato (come Filezilla) che permette tramite una interfaccia semplice di tipo drag and drop di effettuare trasferimenti.

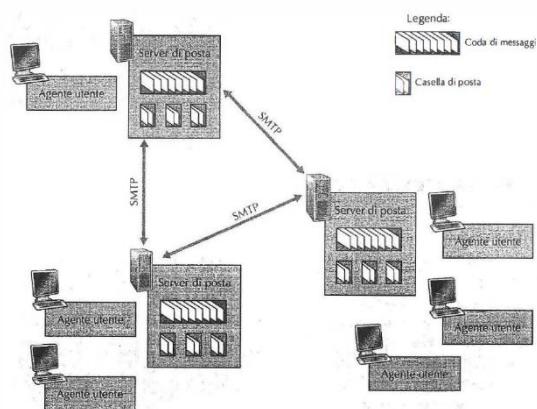
Nota che sia HTTP che FTP sono protocolli di trasferimento di file. Entrambi i protocolli utilizzano TCP. Il client FTP però stabilisce due connessioni TCP con il server FTP, mentre http solo una. Inoltre il server FTP verifica se il client FTP ha l'autorizzazione e in tal caso invia o riceve i file, pertanto spesso si richiede una autenticazione che però rende FTP non sicuro in quanto tali messaggi vengono inviati in chiaro (ossia non vengono criptati). HTTP però non è fatto per entrambe le vie, ciò è usato per richiedere file di piccole dimensioni (come pagine web) e la richiesta è sempre unidirezionale; inoltre una volta che il server http invia la pagina, questa è solo visualizzata del browser del client e non viene scaricata (stateless). Mentre ftp è migliore per invii bidirezionali di file anche e soprattutto di grandi dimensioni e permette anche il salvataggio di ciascun file nel proprio host o il trasferimento di un file sul server remoto.

Perché il client FTP stabilisce due connessioni con il server FTP?

La prima connessione è detta **connessione di controllo** sulla porta 21 e in tale connessione vengono inviati al server identificativo, password, controllo. Se il controllo va a buon fine si apre **parallelamente** (quindi mentre è aperta la connessione di controllo) un'altra connessione nota come **connessione di dati** sulla porta 20 in cui vengono trasferiti i file. Nota bene che viene stabilita una connessione di dati per ogni file trasferito. Quindi la connessione dati è *non persistente*.

Il protocollo SMTP & i protocolli di accesso

Il *Servizio di posta elettronica* è un po più composito del *servizio di web browsing*.



Il servizio di posta elettronica è costituito da:

- **User agent**: un processo applicativo che permette all'utente di mandare email (es. Outlook)
- **Mail Server**: sono contattati dagli user agent
- **Protocollo SMTP**: per trasferire email dal mittente (mail server del mittente) al destinatario (mail server del destinatario)
- **Protocolli di accesso (POP3,IMAP)**: per scaricare dal proprio mail server le email.

NB: Google GMAIL non rientra in questo mondo perché utilizza http.

Quindi *ho due protocolli* perchè i mail server tra loro comunicano tramite SMTP, mentre lo user agent con il proprio mail server comunica tramite i protocolli di accesso se è in downlink (cioè usa il collegamento per scaricare (dal server al client)) mentre comunica in SMTP anche lui se è in uplink (cioè usa il collegamento per inviare (dal client al server)).

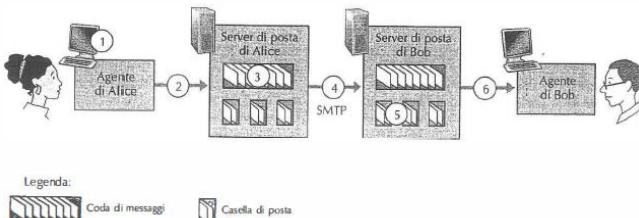
I mail server sono le centrali in cui viene salvata la posta, essi contengono:

- *Una coda*, che sarebbe una casella di posta, per ogni user agent che appartiene a quel mail server e raccoglie tutti i messaggi inviai allo stesso chiamata **mailbox**.
- *Una coda* per ogni user agent che appartiene a quel mail server e raccoglie tutti i messaggi inviai dall'user-agent.

Quindi ogni mail server gestisce le due code di uscita e ingressi di posta, sarebbe come un bucalettere.

PROTOCOLLO SMTP

E' un protocollo applicativo client-server. Vediamo cosa succede in generale quando A(Alice) vuole mandare un messaggio a B(Bob).



- A usa il proprio user-agent per inviare il messaggio a B. Quindi gli fornisce il messaggio+l'indirizzo di posta elettronica di B
- Lo user agent di A invia il messaggio al proprio mail server che lo mette in una coda di uscita.
- Il mail server di alice si accorge allora di avere in uscita un nuovo messaggio da inviare, quindi istaura una connessione TCP **persistente** sulla porta 25 col mail server di B
- Dopo una serie di controlli (handshaking) il client SMTP (il mail server di A) invia il messaggio sulla connessione TCP al server SMTP (il mail server di B) che lo riceve e lo mette nella *mailbox* di B.
- Quando succederà che B vorrà vedere se ci sono nuovi messaggi, userà il proprio user agent che leggerà il messaggio sul proprio mail server nella mailbox.

Il servizio di trasporto usato è quindi il TCP perché voglio usare un servizio affidabile con controllo e recupero d'errore.

Un **grosso problema dei messaggi SMTP** sta nel corpo del messaggio. L'HTTP non impone nulla circa il payload che può essere testuale (se pagine html,xml), codifica jpeg (se immagine), codifica video e altro ancora. Quindi non deve essere necessariamente testuale. In SMTP invece anche il payload del messaggio deve essere codificati in modo testuale (ASCII 7 bit). Questo vuol dire che se nel corpo della email noi mettiamo immagini bisogna trovare un modo per convertire la codifica jpeg in codifica ASCII a 7 bit e poi una volta arrivata al mail server bisogna al contrario convertire da ASCII a 7 bit alla codifica jpeg originaria. Questo problema è dovuto al fatto che SMTP è vecchio come protocollo (più di http) e prima non servivano che brevi email testuali e basta più.

Vediamo un classico **colloquio** tra client e server SMTP:

Supponiamo che il client SMTP abbia già istaurato una connessione TCP col server SMTP.

L'handshaking successivo consta di tre step

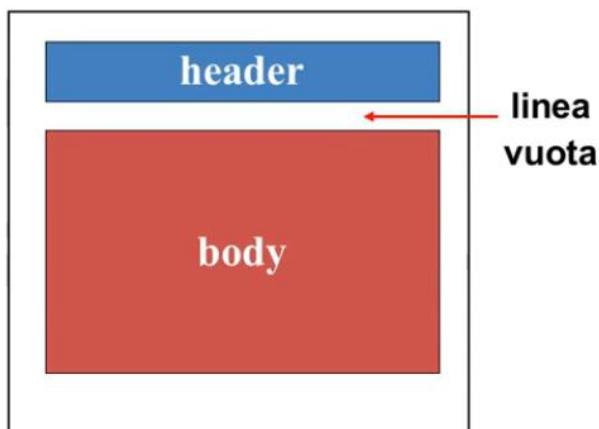
- 1) Il client si presenta al server
- 2) Il client invia il messaggio
- 3) Si chiude la connessione

apertura	S: 220 antoniomailserver.com C: HELO matteomailserver.com S: 250 Hello matteomailserver.com, pleased to meet you C: MAIL FROM: <matteo@matteomailserver.com> S: 250 matteo@matteomailserver.com... Sender ok C: RCPT TO: <antonio@antoniomailserver.com>
invio	S: 250 antonio@antoniomailserver.com ... Recipient ok C: DATA S: 354 Enter mail, end with "." on a line by itself C: Oggi corri al Giuriati? C: . S: 250 Message accepted for delivery
chiusura	C: QUIT S: 221 antoniomailserver.com closing connection

Le righe nere (S) sono del server, mentre le rosse sono del client (C). La prima cosa che il server dice è "ciao io mi chiamo antoniomailserver.com", e il client risponde presentandosi. Ciascun codice specifica l'esito delle operazioni precedenti. Quindi la 220 indica che la connessione ha avuto successo. HELO (abbreviazione di HELLO) è usato dal client per salutare.

Adesso il client può inviare il messaggio. Con MAIL FROM indica da chi è inviata l'email (*perché il mail client può gestire anche diversi utenti*), e il server risponde con un codice di esito. Ogni comando del client è seguito da uno del server con un codice. Con RCPT si specifica il destinatario (*perché anche qui il mail server gestisce più utenti*) ...con DATA si dice che si vuole inviare il corpo del messaggio, e appena il server è pronto, il corpo del messaggio viene inviato. Quando col codice 250 si dà la conferma che il messaggio è stato salvato nel mailbox del server SMTP allora il client con QUIT chiude la connessione e il server fa lo stesso.

Quindi il **formato delle email SMTP** da inviare al proprio mail server seguono questo formato:



Ogni email ha un header e un corpo. Nell'esempio c'era solo il corpo. L'header può contenere:

- To
- From
- Subject

Le prime due sono obbligatorie e vengono usate per la comunicazione sopra fatta.

PROTOCOLLO DI ACCESSO DI ACCESSO

Se l'user vuole comunicare col mail server per inviare (uplink) allora parlerà in SMTP, altrimenti in dowlink parlerà in tre modi possibili:

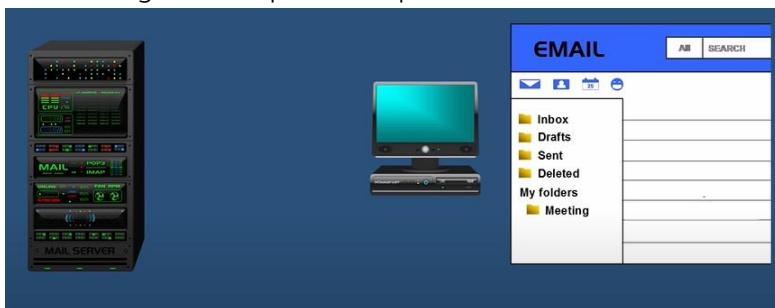
- POP3
- IMAP
- HTTP

Vediamo il funzionamento di POP3

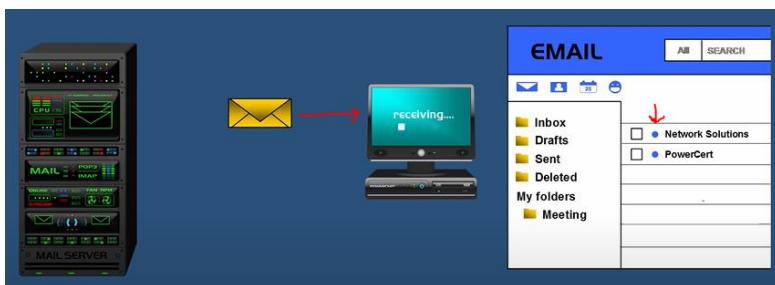
La richiesta di dowlink verso il server SMTP nostro consiste di 4 passi:

- 1) Si istaura una connessione TCP sulla porta 110 tra user agent e server SMTP.
- 2) Lo user agent invia al server nome utente e password per identificarlo (fase di autorizzazione).
- 3) Lo user agent recupera i messaggi di posta. In questa fase (fase di transazione) può anche marcare i messaggi per la cancellazione.
- 4) Appena lo user agent scrive QUIT si ha la terza fase (fase di aggiornamento) in cui si conclude la sessione POP3. In questa fase il server elimina i messaggi di posta marcati.

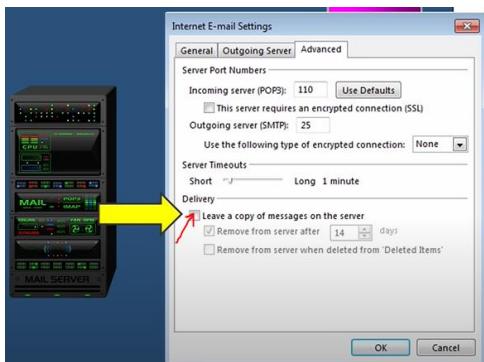
A livello figurativo questo è quello che succede.



Inizialmente il nostro user agent come Outlook non visualizza nulla. Per visualizzare i messaggi istaura una connessione tcp sulla 110.



Dopo di che il mail server invierà tramite POP3 le email che lo user agent ha richiesto di scaricare e appariranno nell'interfaccia.



Come vedremo la POP3 di norma è impostata nella modalità 'scarica e cancella', ossia una volta che ha scaricato l'email la cancella sul server. Questo ci impedisce di riottenere la stessa da un altro dispositivo. Ma possiamo impostarla per salvare una copia sul server.

Ovviamente tutto questo è semplificato, ma quando facciamo check su quella opzione o quando apriamo la carta dei "messaggi in arrivo" non facciamo altro che far parlare tramite POP3 lo user con il mail server. La comunicazione POP3 tra user agent e mail server è davvero elementare. In ogni comunicazione il server reagisce dicendo una tra le due risposte:

- +OK (seguito opzionalmente da dati) → indica al client che il precedente comando è andato a buon fine.
- -ERR → indica al client che qualcosa con il precedente comando è andata male.

La fase di *autorizzazione* prevede solo due comandi:

```
telnet mailServer 110
+OK POP3 server ready
user bob
+OK
pass hungry
+OK user successfully logged on
```

La fase di *transazione* prevede un dialogo che può essere "scarica e cancella" o "scarica e mantieni". Vediamo una transazione del primo tipo:

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: (bla bla ...
S: .....
S: .....bla)
S: .
C: dele 1
C: retr 2
S: (bla bla ...
S: .....
S: .....bla)
S: .
```

Lo user usa */list* per dire al server "dimmi tutti i messaggi che hai e la relativa dimensione". Dopo di che col comando *retr n* lo user indica al server di volere il messaggio da lui segnato con n. Il messaggio gli viene inviato.

Siccome è una transazione del primo tipo allora marca i messaggi con *dele* per dire che nella fase di aggiornamento deve essere poi eliminato. Quindi *dele* serve a marcare.

Fa lo stesso col messaggio 2.

Alla fine usa *quit* per chiudere la transazione e inizia la fase di aggiornamento. **Nota** come tutto questo è semplificato (si riduce a click e drag&drop) dall'interfaccia user-agent.

```
C: dele 2
C: quit
S: +OK POP3 server signing off
```

Il protocollo POP3 gestisce solo lo scaricamento dei messaggi in un'unica cartella e non ha nulla a che fare con lo scaricamento dei messaggi inviati o di altre cartelle personali sullo user-agent. Inoltre **non permette la sincronizzazione**, ossia se scarico ed elimino una email sul mio user agent, se quella stessa email era stata scaricata su un altro dispositivo, rimarrà tale. Il protocollo **IMAP invece** serve invece a fare le stesse cose del POP3 ma in più permette all'utente di creare e spostare cartelle e messaggi, permette la **sincronizzazione**, quindi se mi arriva una email su un dispositivo o ne elimino una ne risentiranno tutte le interfacce di tutti i dispositivi. Inoltre IMAP è utile come protocollo anche quando si hanno connessioni lente e quindi si vuole solo una parte del messaggio (magari solo il testo).

Quindi a seconda del protocollo si contatteranno mail server diversi perché diverse sono le capacità di cui hanno bisogno. Utilizzando POP3 si contatterà un server nella forma pop.mailserver.com (es. pop.gmail.com) mentre utilizzando IMAP sarà imap.mailserver.com

DNS: RISOLUZIONE DI NOMI SIMBOLICI

Il protocollo **DNS** cerca di risolvere il seguente problema. I server sono indirizzabili tramite *indirizzi IP(32)*. Se vi dicesse a che server appartiene questo indirizzo IP:

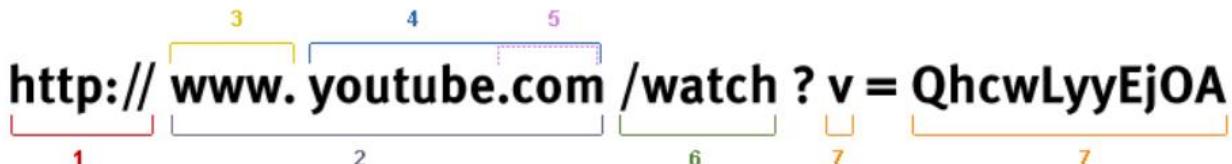
74.125.206.99

probabilmente non lo sapete, eppure lo usate ogni giorno. Sarebbe (uno dei tanti) www.google.com. Occorre quindi una mappatura tra indirizzi IP e i nomi simbolici così che un utente utilizzi il nome simbolico senza problemi, mentre un "qualsiasi" lo traduce.

Per realizzare questo servizio devo:

- avere in rete un database distribuito che ospiti queste coppie
- avere un protocollo applicativo basato su UDP (perché mi interessa velocità, al massimo in caso di errore ripeto) per accedere dentro questi database

Prima di continuare facciamo chiarezza su certi concetti.



- 1) è il protocollo
- 2) viene chiamato **host name** e comprende:
 - 3) **sottodominio** www
 - 4) **dominio** youtube.com che comprende:
 - 5) **top level domain** com

Quindi l'host name, ossia il nome identificativo e univoco col quale un computer si presenta in rete, include sottodominio, dominio e top level domain. Questo ci servirà quando parleremo dei record dns.

Il database che mantiene i dati, in gergo dns viene detto **name server**.

Il DNS oltre a supportare la traduzione dei nomi offre anche i seguenti servizi aggiuntivi:

- *host aliasing*: oltre ai **nomi canonici** che sarebbe il nome associato all'indirizzo IP, si possono usare dei sinonimi. Per esempio invece di www.google.com si può usare l'*alias* google.com e invocando il DNS si può ottenere il nome canonico, così come quando lo si invocava per ottenere l'indirizzo IP.

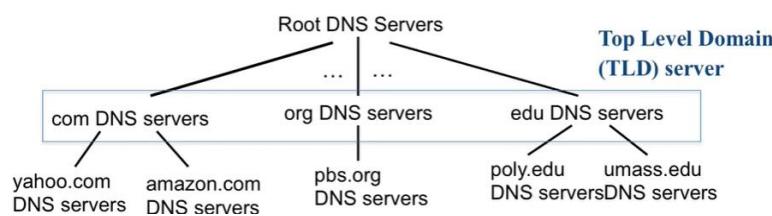
ALIAS ----->NOME CANONICO----->INDIRIZZO IP

- *Mail Server Aliasing*: quando parlando dell'SMTP abbiamo detto che lo user agent contattava un mail server SMTP. Potrebbe connattare il DNS server per farsi dire quale sia il nome canonico dell'alias hotmail.com, questo solo se l'hostname del server hotmail.com è solo un alias e non il nome effettivo canonico.
- *Load Distribution*: certe pagine web (tipo google) che vengono molte volte richieste possono essere fornite da macchine fisiche diverse. Quindi google ha molti server che dispongono della stessa pagina. Supponiamo che una macchina fisica di google ha troppo carico. Allora si va a modificare dinamicamente la riga del database DNS modificando la mappatura così da ridirigere www.google.com non al solito IP ma ad un altro che magari ha meno carico.

NAME SERVER (IL DATABASE DNS)

Non è un architettura centralizzata. Quindi "dentro la rete" ci stanno tanti name server gerarchicamente distinti, ossia il DNS utilizza tantissimi server sparsi per il mondo, o meglio *distribuiti* nel mondo (da cui architettura distribuita). In uno dei tanti server è possibile trovare la mappatura voluta, ma non tutti possono avere tale informazione. Ecco perché i server dns sono organizzati in modo gerarchico.

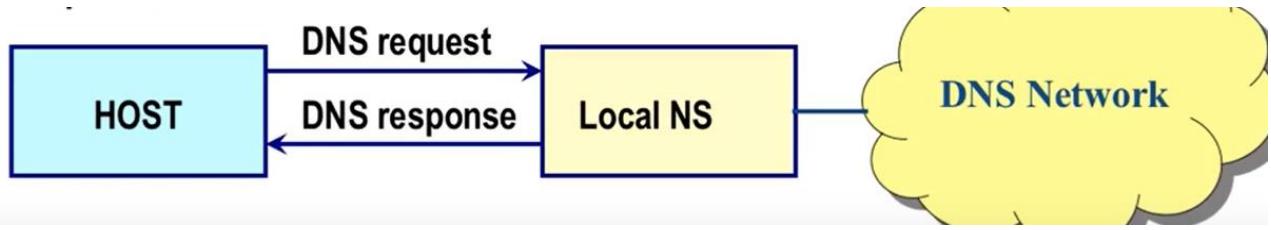
Vediamo come funziona il percorso se volessi cercare il server www.amazon.com



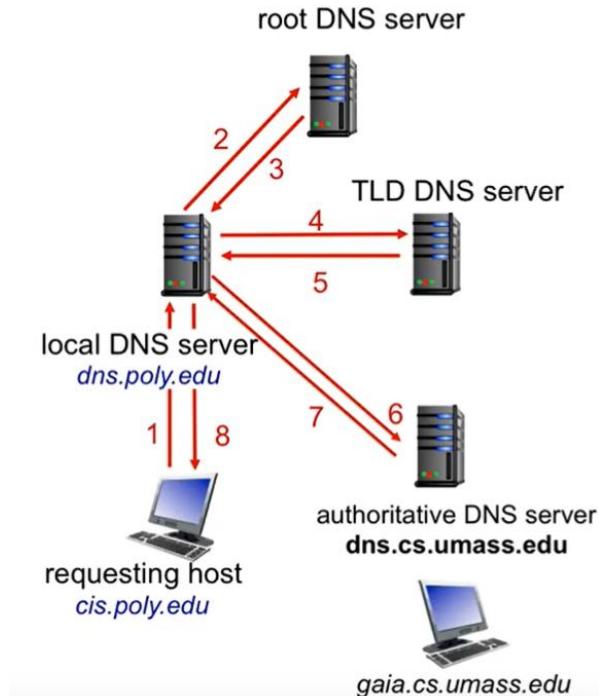
La radice è detta **root dns server** e loro sanno a quali **TLD server** rivolgersi a seconda del dominio. In questo caso si rivolge a uno o più *com DNS servers*. Di TLD ce ne sono tanto quanti sono i domini.

Una volta giunto sul o sui TLD vengo indirizzato ad altri dns server (o a uno solo) detti **dns server autoritativi**. Infine uno di questi mi restituisce l'IP dell'hostname cercato.

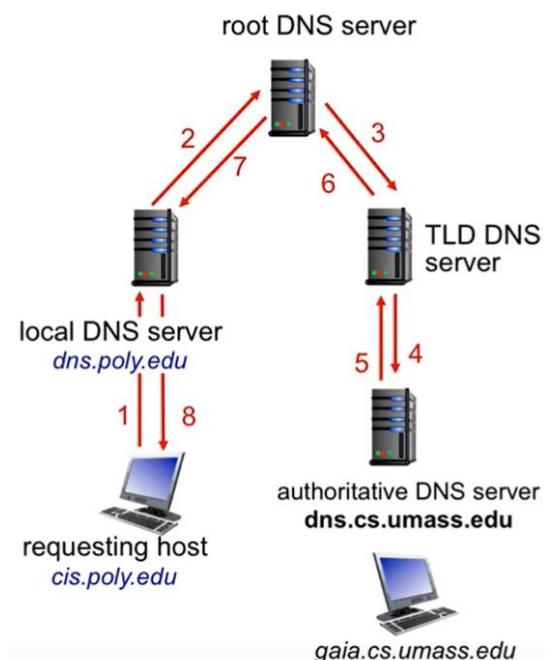
Esistono altri tipi di name server che sono i **local name servers**. Ogni ISP ha un name server locale. Questo è quello che io contatto sempre quando invoco una richiesta di dns. Sarà poi lui a contattare la gerarchia qualora non sapesse la traduzione.



COME FA IL NAME SERVER LOCALE A OTTENERE L'INFORMAZIONE DALLA RETE LOCALE?
Esistono due modalità:



1) **Modalità interattiva:** l'host (noi utente) contattiamo il proprio local dns server per risolvere un nome simbolico (www.google.com). Lo LNS contatta uno dei 13 root server. Questo segnala indietro all'LNS qual è l'altro name server a cui accedere per avere altre informazioni per procedere con la ricerca. Lo LNS contatta il TLD dns server. Anche lui segnala indietro il name server autoritativo da contattare. Lo LNS contatta il name server autoritativo, quindi contiene la traduzione che ritorna a LNS. Ogni richiesta utilizza UDP come trasporto.



2) **Modalità ricorsiva:** lo scenario è sempre lo stesso. Questa volta è il root name server stesso a prendersi carico di chi contattare, e anche i tld e i name server autoritativi. Questi ritornano la traduzione ai tld e poi ai root e infine al Ins. Questa modalità è poco usata perché carica troppo il collo di bottiglia del sistema dns.

DSN CACHING

Indipendentemente dalla modalità esiste uno scambio di informazione tra nameserver della stessa gerarchia e name server che sono di gerarchia superiore. Quello che il DNS implementa sono anche delle tecniche di caching che già abbiamo visto nei proxy. La differenza è che qui un name server (che non è autoritativo) può decidere di memorizzare non la pagina web come nei proxy ma la mappatura. Nota come ho detto name server non autoritativi. Quindi nella modalità iterativa o ricorsiva che sia, quando l'LNS riceve dal name server autoritativo la mappatura, lui può decidere di memorizzarla. Questo fa sì che quando gli arriva una richiesta, invece di contattare i name server gerarchici fino a quello autoritativo, diventa "lui stesso autoritativo".

RECORD DEI NAME SERVER

Resource Record	Name, Value, Type, TTL
-----------------	-------------------------------

I record (detti *resource record*) presenti nei database dei name server hanno 4 campi:

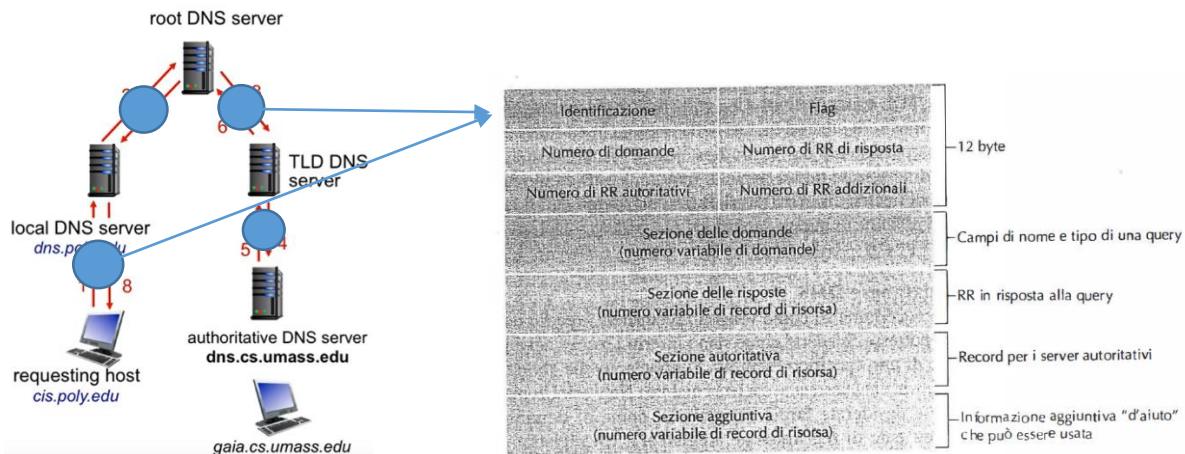
- **Name e Value:** dipendono dal valore di Type.

- se Type=A allora Name è il nome dell'hostname e Value è il suo indirizzo IP.
[www.youtube.com, 145.37.39.126, A]
- Se Type=NS allora Name è un dominio e Value è l'hostname del DNS server autoritativo che sa come ottenerne l'indirizzo IP
[youtube.com, dns.foo.com, NS]
- Se Type=CNAME allora Name è un sinonimo e Value è il suo nome canonico
[foo.com, relay1.bar.foo.com, CNAME]
- Se Type=MX allora Name è un sinonimo e Value è il suo nome canonico ma a differenza di prima è di un mail server.
[foo.com, mail.bar.foo.com, MX]

Il TTL noto come **time to live** indica il tempo per cui può essere memorizzato questo record in cache prima di essere eliminato. Questo serve perché magari i record associati a un dominio possono cambiare e quindi per rendere consistente l'indirizzamento bisogna dire a chi preleva il record "guarda puoi salvartelo, ma al massimo per un tempo TTL perché poi cambierà".

Nota bene: il record A è quindi il record tipico dei name server autoritativi, ma col meccanismo del caching dns potrebbe questo trovarsi anche nell'LNS.

FORMATO DEI MESSAGGI DNS



Tutti questi messaggi di comunicazione DSN hanno un formato ben preciso:

- Header

- o **Identificativo**: serve a identificare il messaggio così che quando si ottiene risposta per lo stesso, il client possa capire a quale richiesta che ha fatto corrisponde una tale risposta.
- o **Flags**: di due bit indica nel primo se il messaggio è di richiesta(0) o risposta(1), mentre nel secondo bit indica se la risoluzione deve essere iterativa o ricorsiva.
- o **Numero di richieste**: una singola richiesta dns può in realtà contenere richieste multiple. Tale numero di richieste si trova nella parte di *payload* e quindi questo campo indicherà quante ne conterrà la relativa sezione *domande* del payload sotto descritta.
- o **Numero di risposte**: idem se però si tratta di un messaggio di risposta. Ovviamente se il messaggio è di risposta allora il campo sopra è zero e viceversa.
- o **Numero di record autoritativi**: mi dicono nel payload quanti record sono autoritativi (di tipo A).

- Payload

- o **Domande**: indica il nome richiesto e il tipo di domanda, ossia se per quel nome si vuole l'indirizzo IP (A) o il server di posta (MX)
- o **Risposte**: se è una risposta contiene la risposta alla richiesta di domanda, ossia uno o più record DNS.
- o **Competenza**: record per server autoritativi.
- o **Informazioni aggiuntive**: informazioni utili extra.

COME SI AGGIUNGE UN DOMINIO ALLA RETE DNS?

Se nel nostro pc abbiamo un server che gestisce un sito web, allora devo avere un nome simbolico e tale nome simbolico deve essere inserito dentro uno o più dei name server che fanno parte della gerarchia. Questa procedura di registrazione la gestiscono i **DNS registrar**, entità che hanno accesso in scrittura (quindi possono aggiungere righe) ai name servers. Voglio registrare il dominio mypage.com che deve essere unico e disponibile. Vado quindi dai DNS registrar a cui fornisco tutti gli indirizzi IP delle macchine che gestiscono i servizi di

`mypage.com`, ossia i nostri server autoritativi. Il DNS registrar va nel TLD autoritativo per il dominio `.com` e vi scriverà nel database almeno queste due righe:

[`mypage, dns.mypage.com,NS`]

Per dire qual è il name server locale che gestisce il dominio nuovo.

[`dns.mypage.com,212.212.212.1,A`]

Che indica dove si trova come IP quel name server locale.

Eventualmente si può anche aggiungere un record con Type=MX per il mail server.

APPLICAZIONI PEER-TO-PEER

Non esiste più una distinzione *statica* tra server e client, però in una qualsiasi sessione è possibile che tra due macchine ci sia un rapporto client-server. Non ho colli di bottiglia (quindi non ho il problema che se il server cade allora cade il servizio), è altamente scalabile (il che apporta più server quindi più capacità). La complessità di questa architettura sta nella gestione perché non essendoci un server, devo trovare un qualche modo.

SCALABILITA'

La scalabilità aiuta ad avere **tempi di distribuzione** molto minori rispetto a un'architettura client-server. Vediamo perché. In una *client-server* il tempo minimo è:

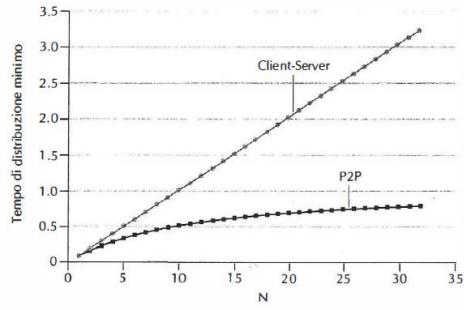
$$T_{min} \geq \max\left\{\frac{NF}{u_s}, \frac{F}{d_{min}}\right\}$$

Dove N sono il numero di client che il server deve servire prima di noi, F sono il numero di bit del file da dare in upload e u_s è la velocità che ha il server nel dare F bits in upload, mentre d_{min} è la velocità più bassa di download tra tutti i client (perché la velocità di servizio di un server dipende anche da quanto veloce sia il client stesso). Questa relazione indica che il tempo di distribuzione del file da F bit a N client è al minimo pari al massimo tra il tempo totale che impiega il server a servire tutti gli N client a un tempo per ciascuno pari a $\frac{F}{u_s}$ con il tempo massimo che un client impiega per scaricare.

Nell'architettura P2P la relazione è questa:

$$T_{min} \geq \max\left\{\frac{NF}{u_s + \sum_1^N u_i}, \frac{F}{u_s}, \frac{F}{d_{min}}\right\}$$

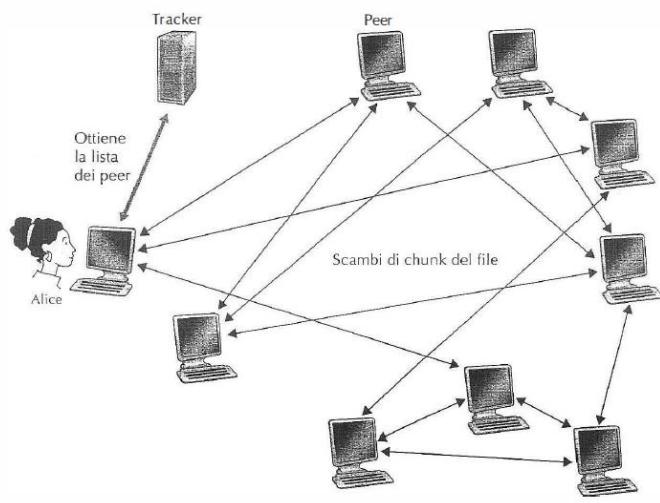
All'inizio il file è detenuto da un solo peer server. Quindi il tempo di attesa sicuramente sarà al minimo pari a quanto impiega all'inizio questo server peer a distribuire a un secondo peer l'intero file $\left(\frac{F}{u_s}\right)$. Se quando un client riceve una parte la reinvia subito a un altro degli $N-1$ client restanti allora è come se la velocità totale aumentasse, quindi la prima indica proprio che è come se il server aumentasse la velocità propria u_s incrementandosi con quelle u_i degli altri. La terza la sappiamo già, dobbiamo considerare il tempo di download del peer a velocità più bassa.



Notiamo come aumentando il numero N dei client, il tempo di distribuzione nel caso client-server aumenta in modo lineare, mentre con quello P2P la tempistica viene notevolmente ridotta perché i client oltre a scaricare fanno anche da server, quindi non sono solo passivi. In termini di tempo di distribuzione la P2P è superiore, ma in termini di velocità uno a uno il client server è meglio.

PROTOCOLLO BIT-TORRENT

E' un protocollo P2P per la distribuzione di file.

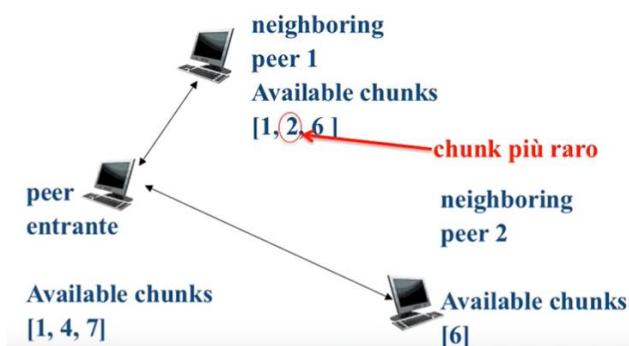


I **tracker** sono i dispositivi che tengono traccia di tutti i dispositivi attivi in un certo **torrent**, ossia un gruppo di peer che si scambiano **chunk**, pezzi del file originario di uguale dimensione (256kbyte quasi sempre).

Come si entra a far parte di un torrent? I peer che vogliono entrare a far parte di un torrent si devono registrare presso un tracker per ottenere la lista dei peer attivi. La lista (inviata dal tracker) contiene gli indirizzi IP di tutti quei peer che

partecipano al torrent. Il peer nuovo stabilisce connessioni TCP con un sottoinsieme di loro (neighboring peers). Questi gli inviano una lista dei chunk disponibili. Il nuovo peer può decidere da chi scaricare un chunk e quale chunk scaricare. Una volta che ha scaricato l'intero file può decidere egoisticamente di lasciare il torrent o rimanerci per servire gli altri. Inoltre non si deve per forza scaricare tutti i chunk ma è possibile scaricarne solo una parte per poi ritornare nel torrent in un altro momento.

Ma quali sono i messanismi con i quali un peer sceglie quali chunk scaricare?



Seguendo il principio del **rarest first** il peer decide di scaricare dagli altri i chunk più rari come primi. Che vuol dire? A sx abbiamo un *peer entrante* che deve scegliere i chunk da scaricare. Questo possiede i chunk 1,4,7 ed è collegato con il *peer1* e il *peer2*. Gli mancano solo i chunk 2 e 6. Sceglie di scaricare il 2 perché il 6 è disponibile a entrambi. In questo modo si tenta di privilegiare il download dei chunk più rari perché semmi il peer1 dovesse abbandonare la rete allora non mi troverei più quel chunk.

Il peer1 implementerà anche lui dei meccanismi per scegliere se e quanto velocemente inviare i chunk a chi glieli richiede. Il principio che segue è di servire il peer se quel peer gli sta inviando chunk al massimo rate (quindi è come se premia il richiedente acconsentendo l'invio se questo però possiede una grande velocità di trasferimento e dedica la banda quasi totalmente a lui). Ognuno serve solo i peer **unchocked**, ossia quei peer che gli stanno inviando i chunk a una velocità/rate elevata (di solito sono 4), e ogni 10 secondi aggiorna questa lista controllando se c'è chi ha un rate ancora più alto. Inoltre ogni 30 secondi si aggiunge alla lista "premiati" un peer causale. Questo meccanismo è detto **tit-for-tat** e serve a far rimanere in piedi questo sistema in quanto se non ci fosse tale sistema allora ogni utente si farebbe inviare i dati senza partecipare. Infatti per questo nuovo peer entrante in modo casuale è come se si stesse scommettendo su di lui. Se questo invia chunk e lo fa a una buona velocità allora il peer1 (ormai lo abbiamo chiamato così) lo tiene ogni 10 secondi nella sua lista, altrimenti lo ripudia e di contro il nuovo peer non potrebbe più ottenere alcun dato.

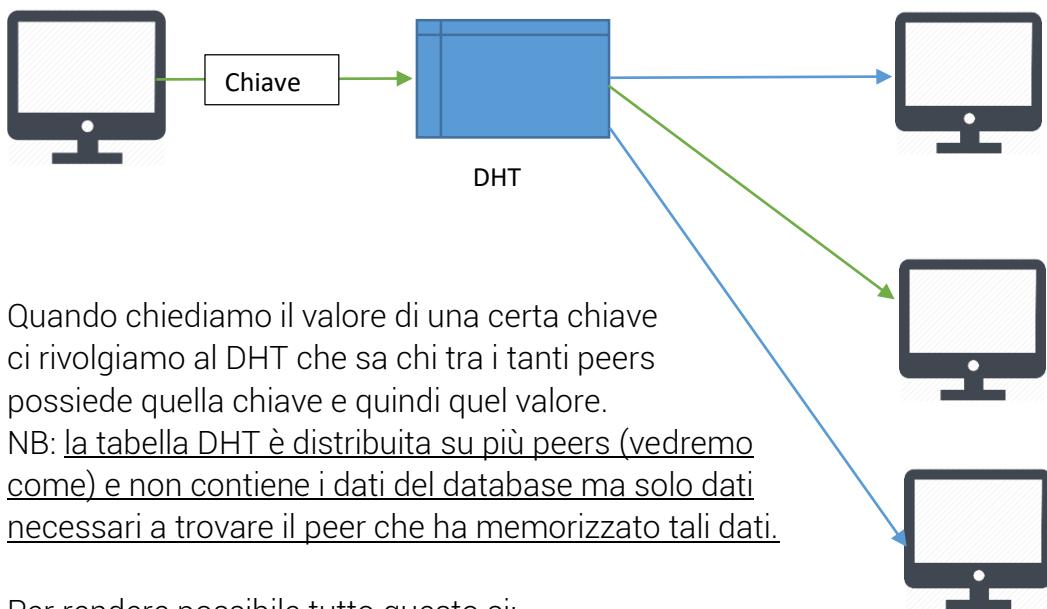
DHT: Tabelle Hash distribuite

Il protocollo BitTorrent ha bisogno di un tracker quindi si potrebbe dire che l'architettura è una p2p centralizzata, questo ad indicare che c'è sempre un entità principe che in questo caso possiede gli indirizzi IP di tutti i peer che partecipano al torrent. Ma se rendessimo anche lo stesso tracker distribuito, allora arriveremmo alla p2p pura.

Scopo di questo paragrafo è capire come distribuire un intero database su più peer. Nota bene che esisteranno due livelli: il database stesso distribuito e un database particolare che conterrà i riferimenti per trovare chi detiene cosa. Immagina di avere una rubrica telefonica con 100 numeri di telefono, o meglio 100 coppie (nome, cellulare). Un **database distribuito** è per esempio una distribuzione delle 100 coppie in 10 computer/peer in cui ciascuno detiene 10 coppie. Molto ma molto semplificando, nella *blockchain* ogni peer detiene *tutte* le 100 coppie, mentre qui vogliamo solo che ciascuno possieda una frazione delle 100. Per questo abbiamo bisogno di un database, anch'esso distribuito chiamato **DHT** che ci permette di sapere dato un nome quale peer detiene quella coppia (nome,cellulare). Supponiamo di voler sapere quale peer possiede il numero di telefono di Marco, ossia la coppia (Marco,

cellulare), allora interrogo il DHT che mi dice che il peer di indirizzo IP 112.111.2.3 possiede la coppia. Allora io lo contatto e mi faccio fornire il numero.

E' chiaro come quindi nel caso del BitTorrent il peer entrante interroga il DHT usando come nome l'id di un certo torrent e il DHT ritorna l'indirizzo IP o gli indirizziIP dei peer che partecipano a quel torrent.



Quando chiediamo il valore di una certa chiave ci rivolgiamo al DHT che sa chi tra i tanti peers possiede quella chiave e quindi quel valore.

NB: la tabella DHT è distribuita su più peers (vedremo come) e non contiene i dati del database ma solo dati necessari a trovare il peer che ha memorizzato tali dati.

Per rendere possibile tutto questo si:

- Assegna un identificatore ad ogni peer che implementa la DHT. Tale identificatore è un valore che sta nell'intervallo $[0, 2^n - 1]$
- Anche le chiavi devono essere interi nell'intervallo $[0, 2^n - 1]$

L'ultima richiesta porta a porsi il seguente problema. Se le chiavi sono letterali (es. nome film), ibride (codicefiscale) come si può pretendere che siano tutte intere? Questi si fa associando ad ogni chiave un valore intero tramite una **funzione hash**:



Nota però che l'associazione è molti a uno, quindi chiavi diverse possono avere lo stesso identificativo numerico.

Ma se qualcuno volesse aggiungere una coppia [CHIAVE,VALORE] all'interno di questo database distribuito, quale peer dovrebbe essere il responsabile? Per decidere chi deve esserlo si sfrutta il fatto che chiavi e identificatori di peer stanno sullo stesso intervallo. Quindi se:

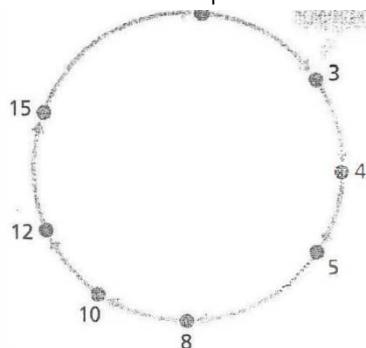
- La chiave è uguale all'identificativo di uno dei peer allora sarà lui a memorizzare la coppia.
- Se un peer con lo stesso identificativo della chiave non esiste allora sarà il peer *successivo* a prendersene carico, ossia il peer il cui identificativo viene dopo o è più vicino tra i successivi (se chiave=12 e i peer attivi sono 1,4,7,14,17 allora se ne prenderà carico il peer 14).

- Se la chiave però ha un valore maggiore degli identificativi dei peer attivi (tipo chiave= 12 e i peer attivi sono 3,4,7,9) allora si fa modulo 2^n della chiave.

Stabilito come deve essere memorizzata la coppia data una chiave, ci si domanda adesso *come fa il peer ad ottenere l'indirizzoIP del peer più vicino?*. In una architettura P2P i peer sono tantissimi e pensare che ogni peer debba memorizzare per ogni peer il suo indirizzoIP e il suo identificativo e contando che ad ogni nuova entrata tutta la rete deve essere aggiornata, è una strada da evitare.

DHT CIRCOLARE

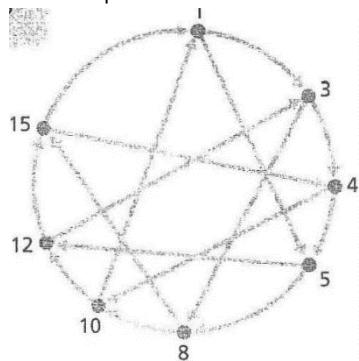
Per risolvere la questione di prima si organizzano i peer in cerchio.



Ogni peer contiene un riferimento a un successore e a un predecessore peer, ossia [CHIAVE,INDIRIZZO-IP] dei due peer. Siccome la DHT nella sua totalità sarebbe un database che contiene tutte quelle tuple allora si dice che il DHT è *distribuito* e ogni peer appunto ne implementa una sottoparte (solo due tuple).

Supponiamo che il peer 3 voglia sapere la chiave 11 a quale peer appartiene. Contatta sempre il suo successore, il 4 che controlla di aver memorizzato tale chiave. Non è così quindi

contatta il suo successore che è il 5 e anche lui non è responsabile. Procedendo così si arriva al 12 che memorizza la chiave 11 e quindi può mandare il messaggio al peer 3. Tutto questo crea evidentemente una serie di messaggi che possono appesantire la rete.

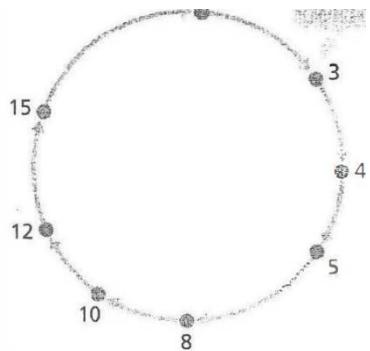


Per questo motivo si dota ciascun nodo/peer del riferimento di più peer. Per esempio quando al 4 arriva l'interrogazione su chi possiede la chiave 11, non rimanda l'interrogazione al 5 ma contatta subito il 10 perché sa che sicuramente lui tra tutti i suoi contatti è quello più vicino a chi possiede la chiave 11.

TURNOVER DEI PEER

Eliminazione di un peer

Ma come si aggiornano i riferimenti nei DHT quando succede una cosa del genere?



Per prima cosa bisogna che ogni peer tenga traccia del primo e del secondo successore. Supponiamo che il 5 vada via.

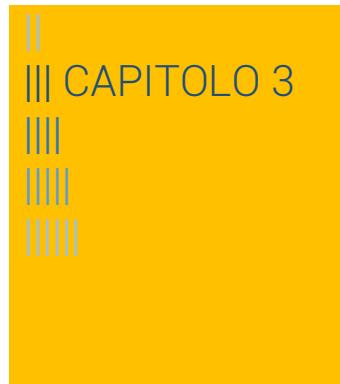
Come si aggiorna la DHT? I nodi che lo precedono, ossia 3 e 4 devono aggiornarsi.

Il nodo 4 memorizzava come primo successore il 5 e come secondo successore l'8. Adesso memorizzerà l'8 come primo e il 10 come secondo (quest'ultima cosa è ottenuta dal 4 chiedendo all'8 quale sia il suo primo successore).

E' importante notare che questi tipi di controlli sul chi c'è nella rete e chi invece l'ha abbandonata vanno fatti ripetutamente. Questo in genere viene fatto da ogni nodo verso i suoi successori inviando un messaggio di ping e aspettandosi una risposta. Allo stesso modo il nodo 3 lascerà 4 come primo successore e memorizzerà 8 come secondo. E' importante anche notare che deve muoversi per primo il precedente a chi ha abbandonato (altrimenti 3 non saprebbe chi mettere come secondo successore).

Aggiunta di un nuovo peer

Supponiamo il peer 13 si voglia unire alla rete. Contatta l'unico peer che conosce, per esempio il peer 1 e lo interroga su chi sia il predecessore e il successore di 13. La notizia gira fino al 12 che sa di essere il suo predecessore. Allora contatta il 13 che inserisce 15 come successore e 12 come predecessore e contatta il 12 per dirgli di modificare il suo successore immediato con 13 e mettere quello che per lei era primo come secondo 15. Ovviamente 13 contatterà 15 per sapere chi sia il suo primo successore che diventa il secondo di 13.



Livello Di Trasporto

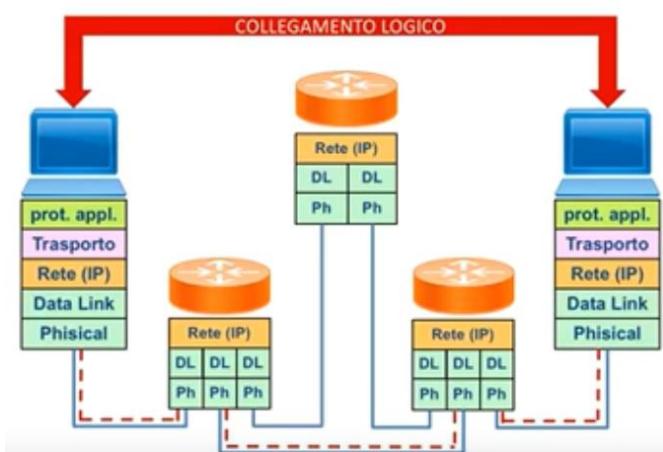


a. **Pila di protocolli a cinque livelli di Internet**

Ci troviamo nel quarto livello (sistema OSI) della pila protocollare.

Qui troviamo tutti quei protocolli usati da protocolli applicativi per il trasporto di informazione end-to-end.

I due protocolli di cui parleremo sono TCP e UDP hanno l'obiettivo "macroscopico" di creare un tubo che collega le due macchine che ospitano programmi applicativi che vogliono scambiarsi informazioni:

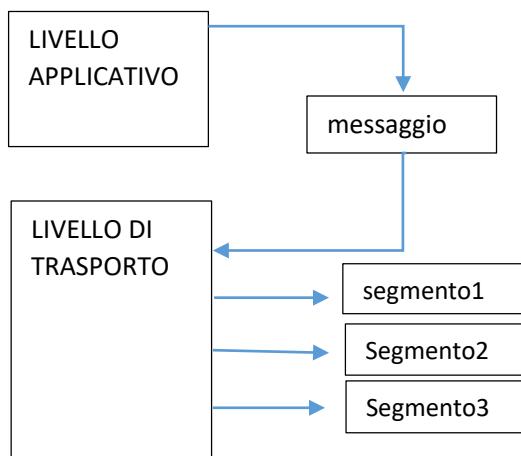


Come il protocollo applicativo anche quello di trasporto opera sugli **end system** o anche detti host.

Il protocollo di trasporto si occupa di creare quel **collegamento logico** tratteggiato in rosso. Il collegamento è logico perché sfrutta quelli che invece sono **collegamenti fisici** in blu, e in questo caso sfrutta tre salti fisici facendo sì che la comunicazione avvenga sfruttando il primo router, poi dal primo al terzo e dal terzo alla destinazione. Quindi due processi applicativi che usano il livello di trasporto si vedranno come connessi direttamente e organizzano la loro comunicazione proprio in questo modo senza preoccuparsi del fatto che in mezzo ci possa essere una gigantesca infrastruttura di router.

NB: il livello di trasporto si occupa della comunicazione logica tra processi che stanno su macchine diverse, mentre il livello di rete si occupa della comunicazione logica degli host che ospitano tali processi.

Il livello applicativo consegna i messaggi applicativi al livello di trasporto che li spezza (se necessario) in più pezzi detti **segmenti** ai quali assegna una intestazione e consegna al livello di rete che ne formerà il datagramma.



INDIRIZZAMENTO: LE PORTE

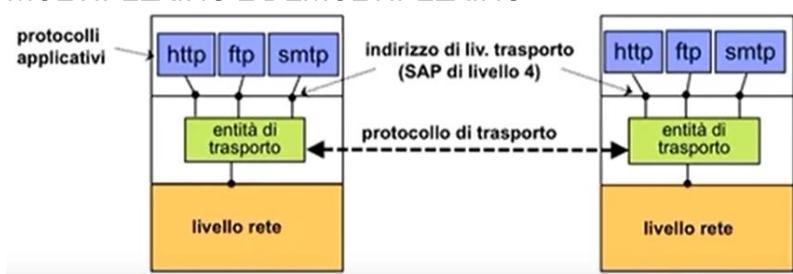
Quando un processo applicativo client istaura una connessione (TCP o UDP) con un processo applicativo server, la porta in cui contattare il server è standard mentre viene scelta in modo random quella del client.

Il numero di porta (che sono stringhe di 16 bit) sono divise in tre categorie:



- *Numeri noti*: sono gli indirizzi riservati a quei protocolli applicativi standard come http,smtp,ftp... per esempio la porta standard per l'http è 80.
- *Numeri registrati*: sono assegnati a specifiche applicazioni da chi ne fa richiesta.
- *Numeri dinamici*: sono riservati al client

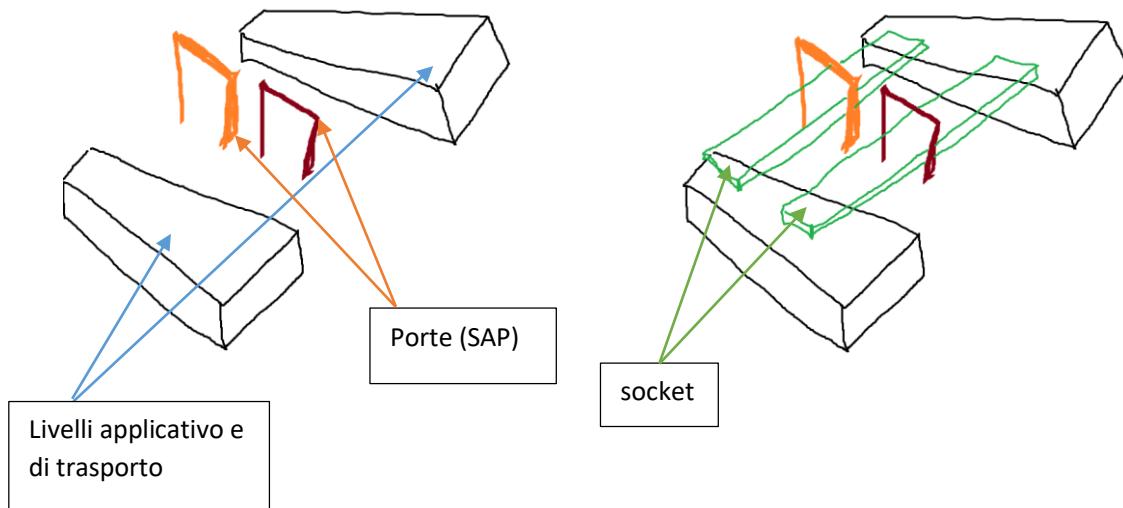
MULTIPLEXING E DEMULTIPLEXING



La **multiplazione** è una funzione che offre *anche* il livello di trasporto. Cosa si intende? Quando diverse entità di un livello usano lo stesso servizio di un entità inferiore

allora c'è bisogno di multiplazione. Per esempio più protocolli applicativi

(http,ftp,smtp..) possono utilizzare lo stesso servizio di trasporto offerto dal livello di trasporto. La multiplazione serve a radunare tutti i frammenti di dati che arrivano in questo caso dal livello applicativo (la http manda la sua, la ftp manda la sua...) e creare segmenti con intestazioni utili a chi poi li riceverà a capire a quale processo vanno inviati. Lato ricevente infatti, il livello di trasporto preleva dal livello di rete questi segmenti e leggendo questa intestazione sa a quale processo applicativo inviare i dati, e questa funzione si dice **demultiplexing**. Resta da capire cosa vengono messe in queste intestazioni. Ogni livello comunica con il livello adiacente tramite uno dei punti di accesso che fanno da raccordo detti Service Access Point (SAP). Nel caso di livello applicativo e livello di trasporto questi SAP altro non sono che le **porte**. Le socket sono proprio dei punti che utilizzano questi SAP per infilare dati tra i due livelli.



Un socket quindi oltre ad indicare un oggetto software per il programmatore, indica anche una coppia

IndirizzoIP:porta = socket

che identifica macchina e processo univocamente.

Nelle intestazioni quindi ci verrebbe di dire che bisogna mettere:

- indirizzoIP e porta dell'host che vuole comunicare
 - indirizzoIP e porta dell'host destinatario
- Eppure non è così. Nelle intestazioni vengono solo messe:
- porta dell'host che vuole comunicare
 - porta dell'host destinatario

Questo perché lo ripeto il livello di trasporto si occupa solo del collegamento logico tra processi! Sarà poi il livello di rete a inserire un suo header che specifichi indirizzoIP dell'host mittente e dell'host destinatario perché il suo compito è creare il collegamento tra host.

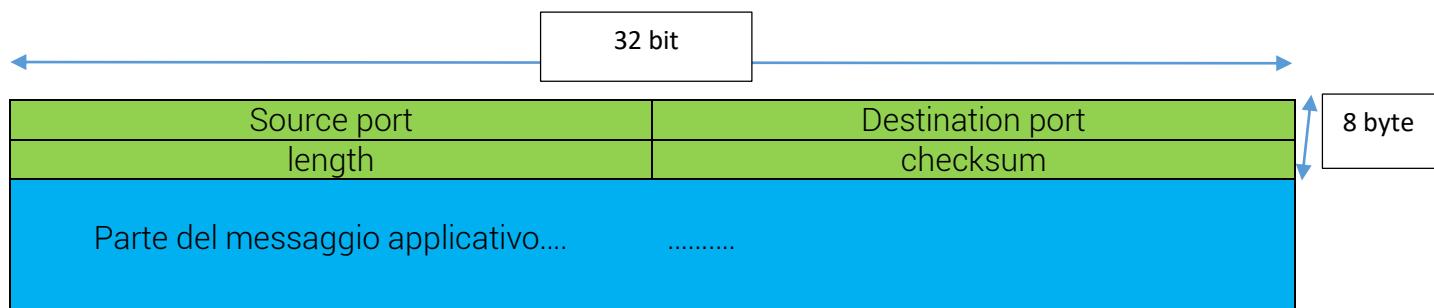
NB: i SAP sono presenti come punti di ingresso tra due livelli e in ogni livello, ma cosa questi siano dipende dal livello a cui si trovano. Qui noi li abbiamo visti come punti di ingresso tra livello applicativo e trasporto, e quindi sono porte virtuali, numeri per

identificare un processo, ma per esempio potrebbero pure essere tra il livello di collegamento e quello fisico, queste porte possono non essere più virtuali, ma fisiche: posso decidere se instradare un pacchetto tramite interfaccia wifi o interfaccia Ethernet.

UDP: User Datagram Protocol

E' un protocollo che non garantisce la consegna e non esercita alcun controllo sugli errori nel flusso che arriva, e non è orientato alla connessione perché si inviano segmenti UDP senza prima mettere su un canale con l'entità destinataria (non c'è handshake).

In questo protocollo si creano **segmenti UDP** che contengono nel loro payload una parte del messaggio applicativo ma anche un header che ospita informazioni su come multiplare e demultipilare il traffico.



l'**header** (in verde) è fatta dalla *porta sorgente* e dalla *porta di destinazione* nei primi 8 byte e contiene un campo di *lunghezza* che mi dice quanto è lungo tutto il segmento applicativo (header+payload). Questo campo è utile al ricevitore per allocare e deallocare memoria, quindi deve sapere quanto spazio necessita il segmento. Il *checksum* sono una sequenza di bit che il ricevitore, una volta letti, capisce se l'informazione arrivata è consistente o meno. L'header è detto **overhead** e l'UDP, poiché ne ha uno piccolo, si dice che ha un minore overhead.

Il **payload** (in blu) contiene una parte del messaggio applicativo.

CHECKSUM UDP

Particolare importanza ha il checksum (16bit). UDP non controlla errori ma fornisce un checksum per un minimo controllo d'errore. Questi bit indicano al ricevente se il trasferimento ha causato alterazione delle informazione, magari dovuti a disturbi nei collegamenti, o quando sono stati memorizzati nei router.

Supponiamo che l'header è il seguente:

0110011001100000	0101010101010101
100011100001100	checksum
Parte del messaggio applicativo....	

Il checksum lato mittente va calcolato sommando intanto le prime due:

$$\begin{array}{r}
 0110011001100000 \\
 0101010101010101 \\
 \hline
 1011101110110101
 \end{array}$$

Poi si calcola la somma tra il risultato e la stringa di bit del campo lenght:

$$\begin{array}{r}
 1011101110110101 \\
 1000111100001100 \\
 \hline
 0100101011000010
 \end{array}$$

Se esiste un riporto finale, questo si somma all'inizio.

Nel checksum si registra il complemento di questa stringa, quindi invertiamo i bit e salviamo:

0110011001100000	0101010101010101
1000111100001100	1011010100111101
Parte del messaggio applicativi....	

In realtà si dovranno tenere in considerazione anche i dati applicativi come gruppi da 16 bit da sommarsi anche loro.

Quando questo segmento arriverà al ricevitore, farà la stessa cosa, sommerà i tre campi e senza complementare la sommerà al checksum. Il segmento risulterà corretto se il risultato è composto da tutti 1.

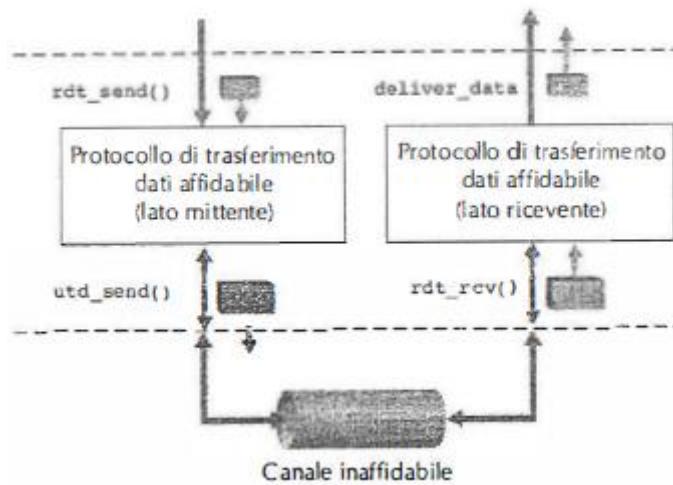
COSTRUZIONE DI UN PROTOCOLLO DI TRASFERIMENTO DATI AFFIDABILE

Ogni strato protocollare può essere implementato in software, hardware o in una delle due combinazioni. Il livello applicativo così come quello di trasporto sono implementati in software, mentre quello di rete è sia in hardware che in software. Cosa vuol dire? Andremo a costruire dei protocolli che rendono affidabile la comunicazione tra due macchine, la cui comunicazione di solito se ci si affida ai livelli di rete e oltre, risulta inaffidabile. I protocolli di trasferimento lavoreranno con:

- dati che il livello applicativo gli darà e che dovrà consegnare al livello di rete
- dati che dal livello di rete dovrà consegnare al livello applicativo

In ogni caso dovrà ogni volta trattare "pezzi" di bit con dei significati e a seconda di questi dovrà prendere delle decisioni per *rimediare all'inaffidabilità* dei livelli inferiori.

Il protocollo che costruiremo sarà chiamato **reliable data transfer** o **rdt** e verrà costruito facendo varie supposizioni fino alla versione finale.



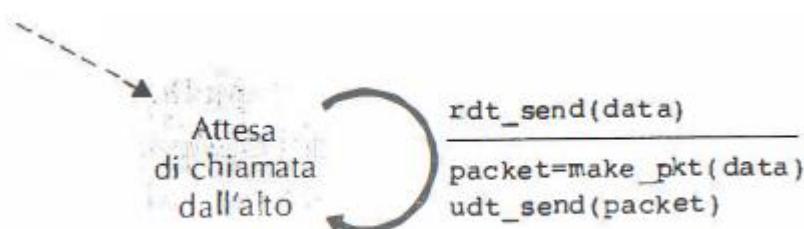
Il livello applicativo A quando ha dei dati li consegna al livello applicativo B chiama **rdt_send()**. Questa funzione prende i dati, fa quello che deve fare, e li manda tramite **utd_send()** al livello rete. Questo lo spedisce a B. Arriva al livello di rete di B la cui parte software avvisa di avere nuovi dati. Il livello di trasporto li preleva con **rdt_recv()** e li elabora. Il livello applicativo quando vuole preleva il dato elaborato tramite **deliver_data()**.

Nota bene: Abbiamo supposto A mittente e B ricevente. Ma la comunicazione è full-duplex. In generale:

- **rdt_send()** prende i dati dal livello applicativo e li elabora (forma il pacchetto...)
- **utd_send()** prende il risultato sopra e lo invia alla rete
- **rdt_recv()** prende dalla rete il pacchetto e lo elabora
- **deliver_data()** consegna l'elaborazione al livello applicativo

RDT 1.0: SUPPONIAMO UN CANALE SOTTOSTANTE AFFIDABILE e TRASMISSIONE ORDINATA E ASSICURATA

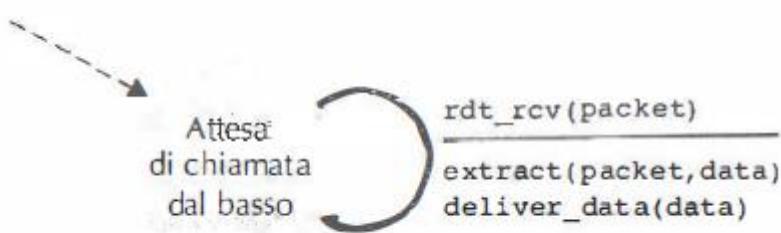
Se consideriamo il canale sottostante affidabile, la **FSM** (finite-state-machine) esprime cosa succede lato mittente e ricevente:



Lato mittente

Si è in attesa. Il livello applicativo chiama **rdt_send** passandogli i dati che vuole mandare al ricevente. La linea orizzontale indica cosa fa la **rdt_send**, quindi le azioni da lei intraprese una volta chiamata: crea un pacchetto **packet** e lo invia al livello di rete tramite **utd_send**

azioni da lei intraprese una volta chiamata: crea un pacchetto **packet** e lo invia al livello di rete tramite **utd_send**



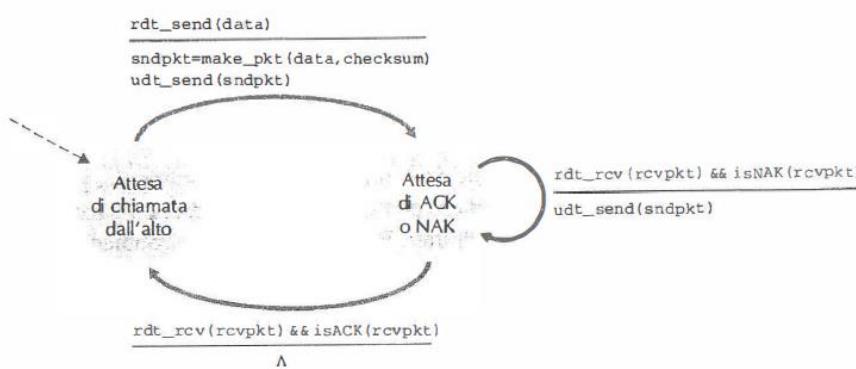
quindi lo estraе (ossia prende solo il payload) e lo invia al livello applicativo.

Questo protocollo è ideale. Non conta che il livello di rete è inaffidabile, quindi il ricevente non notifica nulla al mittente, non conta di rallentare per rispettare certi limiti di assorbimento del ricevente etc...

RDT 2.0: SUPPONIAMO UN CANALE SOTTOSTANTE CON ERRORI SUI BIT MA TRASMISSIONE ASSICURATA E ORDINATA

Adesso ci allontaniamo leggermente dal modello ideale di protocollo supponendo sempre che il canale consegna sempre i pacchetti e sempre nell'ordine corretto (cose che in realtà non sono mai certe) e supponiamo però che questi pacchetti possano avere bit corrotti (ossia 0 al posto di 1 e viceversa).

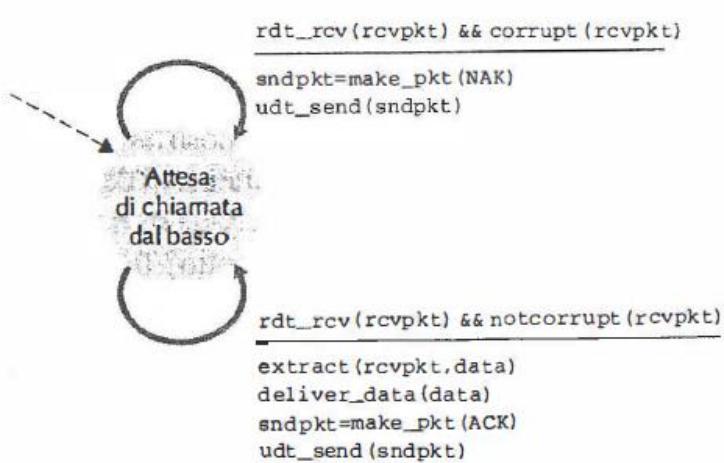
Rdt2.0 rientra nei **protocolli ARQ**, ossia quei protocolli che permettono al ricevente di inviare notifiche (**acknowledgement**) positive o negative su come hanno ricevuto il pacchetto e al mittente di rinviare tale pacchetto.



Lato mittente

E' in attesa che il livello applicativo chiama il protocollo. Lo chiama tramite `rdt_send`. Questo prende i dati, forma il pacchetto che stavolta contiene un **checksum**, ossia una stringa di bit che

(vedremo) consentirà al ricevitore di capire se il pacchetto ricevuto è stato corrotto o meno. Inviato alla rete si mette in attesa di un ack(risposta positiva) o nack(risposta negativa). Il livello di rete lo avverte che ha qualcosa. Lo preleva. Se è un nack allora rimanda lo stesso pacchetto (ci ritenta), altrimenti non fa nulla (il simbolo Λ indica che l'evento non genera azioni, se non cambiare stato).



deve però avvisare della consegna andata a buon fine anche il mittente che è in attesa di risposta, pertanto crea un pacchetto ACK e lo invia alla rete.

Lato ricevitore

Al solito è in attesa. La parte software della rete lo avverte che ha un pacchetto. Lo preleva tramite rdt_rvc().

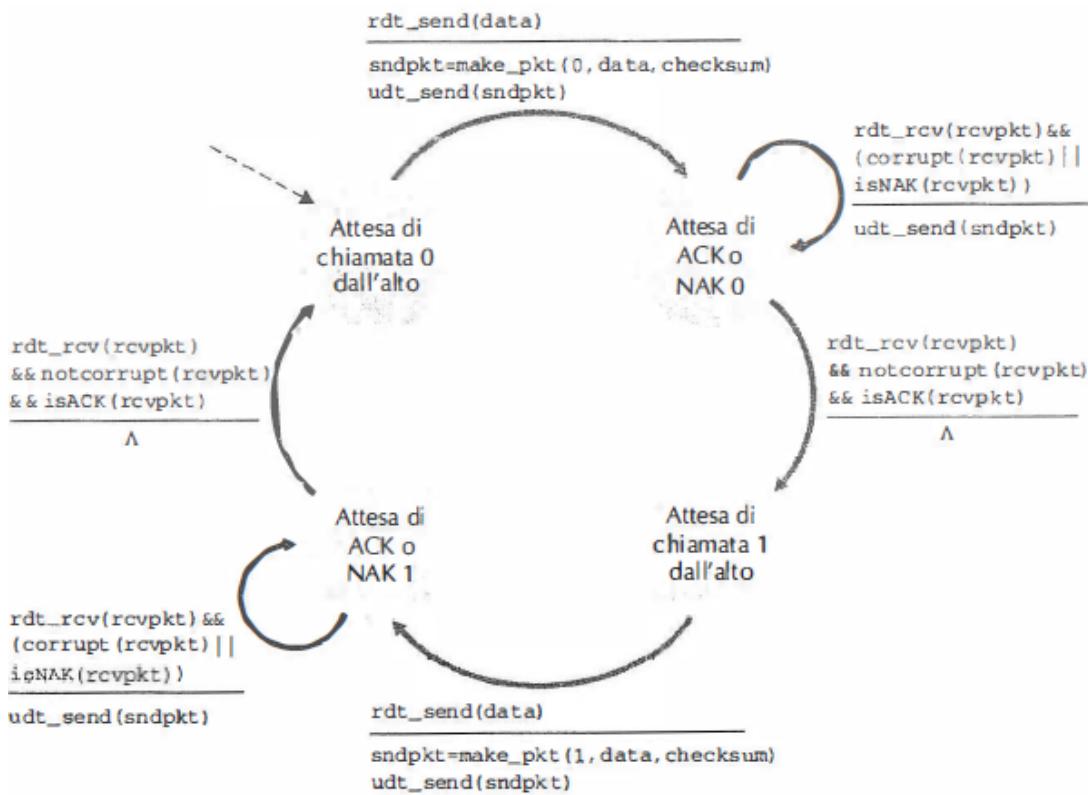
Ne controlla il checksum e se il pacchetto è corrotto rimane nello stato di attesa ma crea un pacchetto anche lui di NAK (potrebbe pure essere un bit) e lo invia in rete al mittente. Se invece non è corrotto, ne estra il payload e lo invia al livello applicativo,

RDT 2.1

Ma oltre ai pacchetti inviati dal mittente, è possibile che gli ack inviati dal ricevente sia corrotti (cioè si manda un ACK ma si riceve un NACK) ed è possibile che il trasmettitore invii quindi gli stessi pacchetti perché fraintende la risposta. Nota che la garanzia che il canale invii i pacchetti nell'ordine del trasmettitore non evita la duplicazione; il trasmettitore può inviare due volte lo stesso pacchetto e per ricezione ordinata significa che ricevo ordinatamente quei due pacchetti. Quindi può darsi che il ricevente mandi un ack ma sia interpretato come un nack e può essere che il ricevitore memorizzi duplicati. Basta fare due cose:

- 1) ogni pacchetto inviato dal trasmettitore deve riportare un valore che oscilla tra 0 e 1. Una volta 0, un'altra volta 1, poi ancora 0 e ancora 1 e così via. Così se il ricevitore riceve il pacchetto 0 e poi di nuovo pacchetto 0 sa che è lo stesso.
- 2) ogni ack/nack possiede anche lui un checksum per validarne l'integrità

mittente



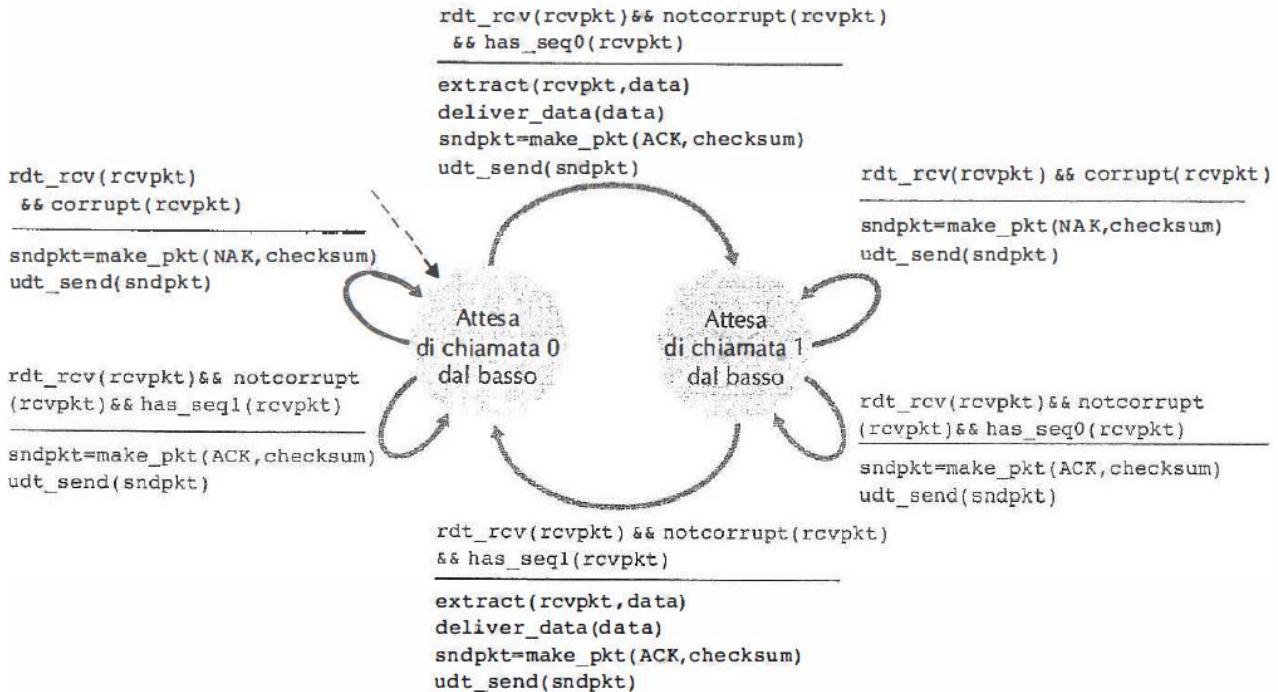
Gli stati sono due perché invia sempre pacchetto0,pacchetto1, ma che ovviamente ad ogni volta che si ricomincia si tratta di pacchetti diversi.

Al solito il mittente è in attesa dell'applicazione. Questa chiama rtd_send per mandare i dati. Viene fatto il pacchetto (con checksum e numerato con 0) e viene mandato. Si aspetta un ack o un nack.

- o Se è un nack oppure è corrotto (e lo si può vedere dal checksum) allora si rimane nello stato e si ritenta rimandando lo stesso pacchetto0
- o Se è un ack ma è anche integro allora si passa ad attendere che l'applicazione voglia mandarne un altro

Si rifa tutto quanto appena detto simmetricamente per il pacchetto1

Ricevente



Il ricevente rimane in attesa sulla chiamata0. Vuol dire che aspetta il pacchetto0. Rimane in questo stato se:

- Il pacchetto0/1 è corrotto
- Il pacchetto non è corrotto ma è pacchetto1, che già ha ricevuto

In entrambi i casi manda un NAK (con checksum).

Se però il pacchetto0 arriva ed è integro allora ne estrae il payload, lo consegna all'applicazione e manda un ack passando nello stato di attesa del pacchetto1.

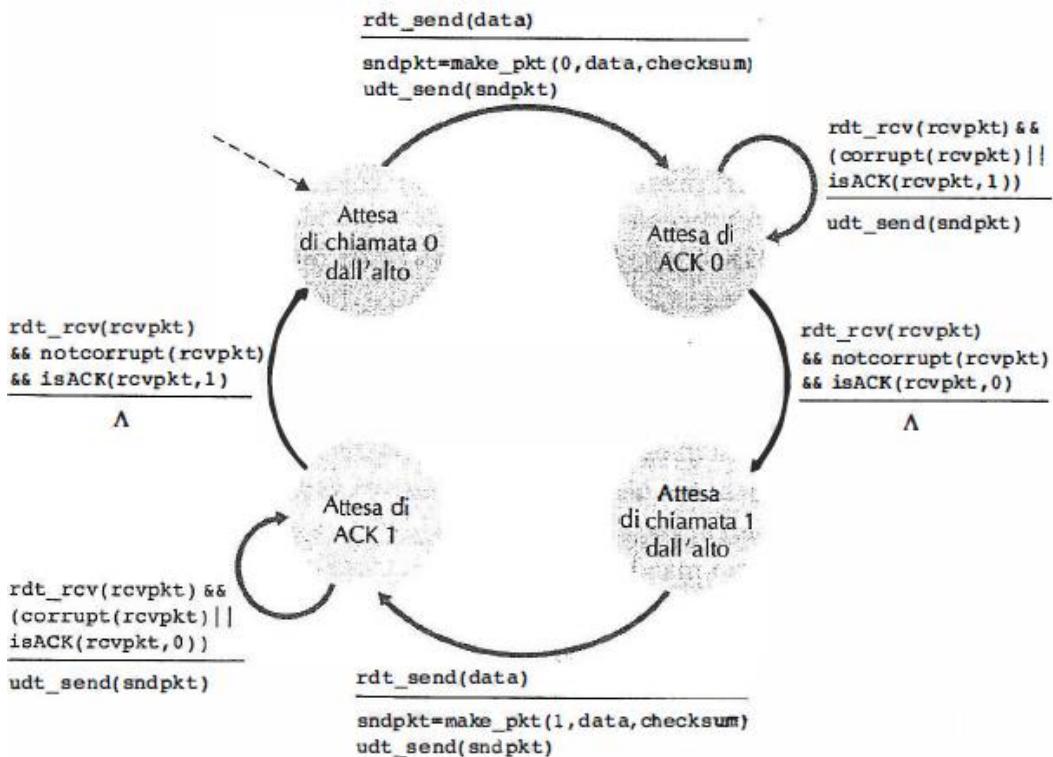
Anche qui si ripete il ciclo ma con il pacchetto1

RTD 2.2

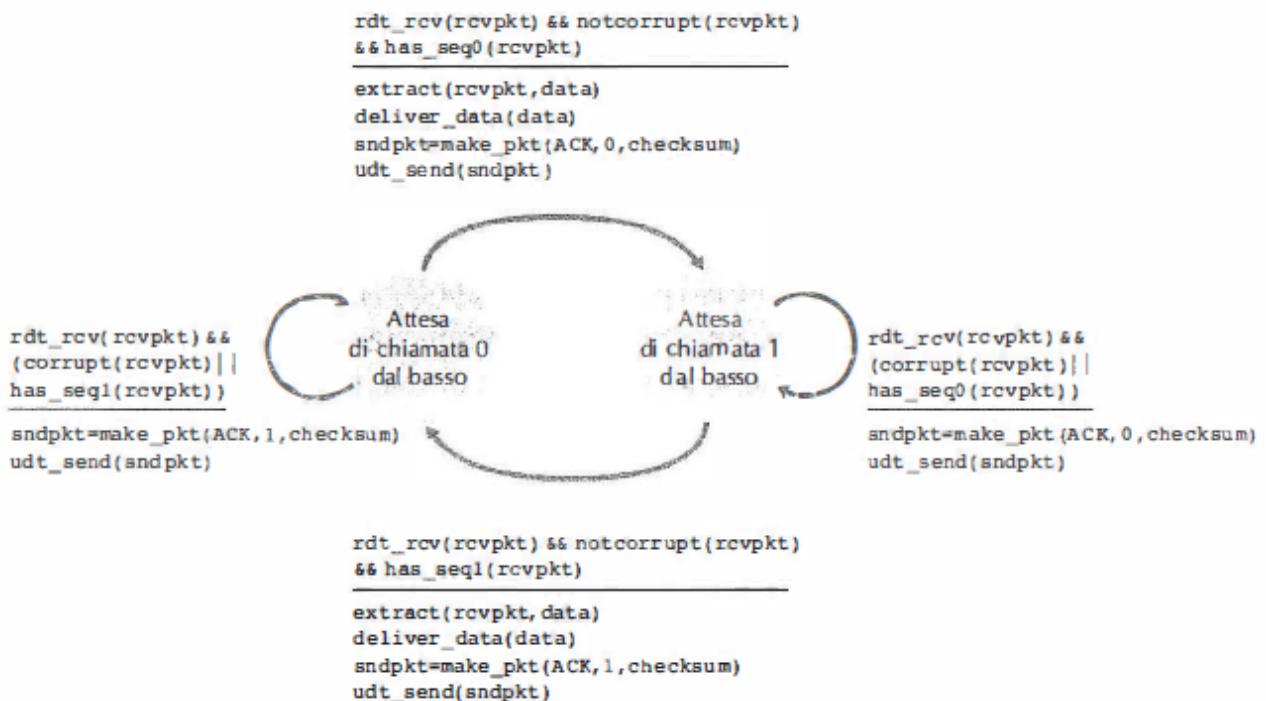
C'è un modo per fare a meno di nack? La risposta è sì. Basta solamente numerare anche l'ack così da far capire al trasmettitore che quell'ack è riferito a un certo pacchetto. Se il trasmettitore invia il pacchetto0 si aspetta di ricevere l'ack 0. Supponiamo lo riceva. Adesso come prima aspetta e quando l'applicazione vuole manda il pacchetto1. Se il ricevente invia un ack1 allora si ricomincia, altrimenti se l'ack è 0 significa che non è riferito al pacchetto1 ma ancora a quello precedente, ergo non ha ricevuto il pacchetto1.

La macchina a stati si modifica leggermente:

Mittente



Ricevente



Sintesi RDT

Abbiamo ottenuto un protocollo che, supponendo che il canale faccia recapitare sempre i pacchetti e nell'ordine di invio del trasmettitore, sa:

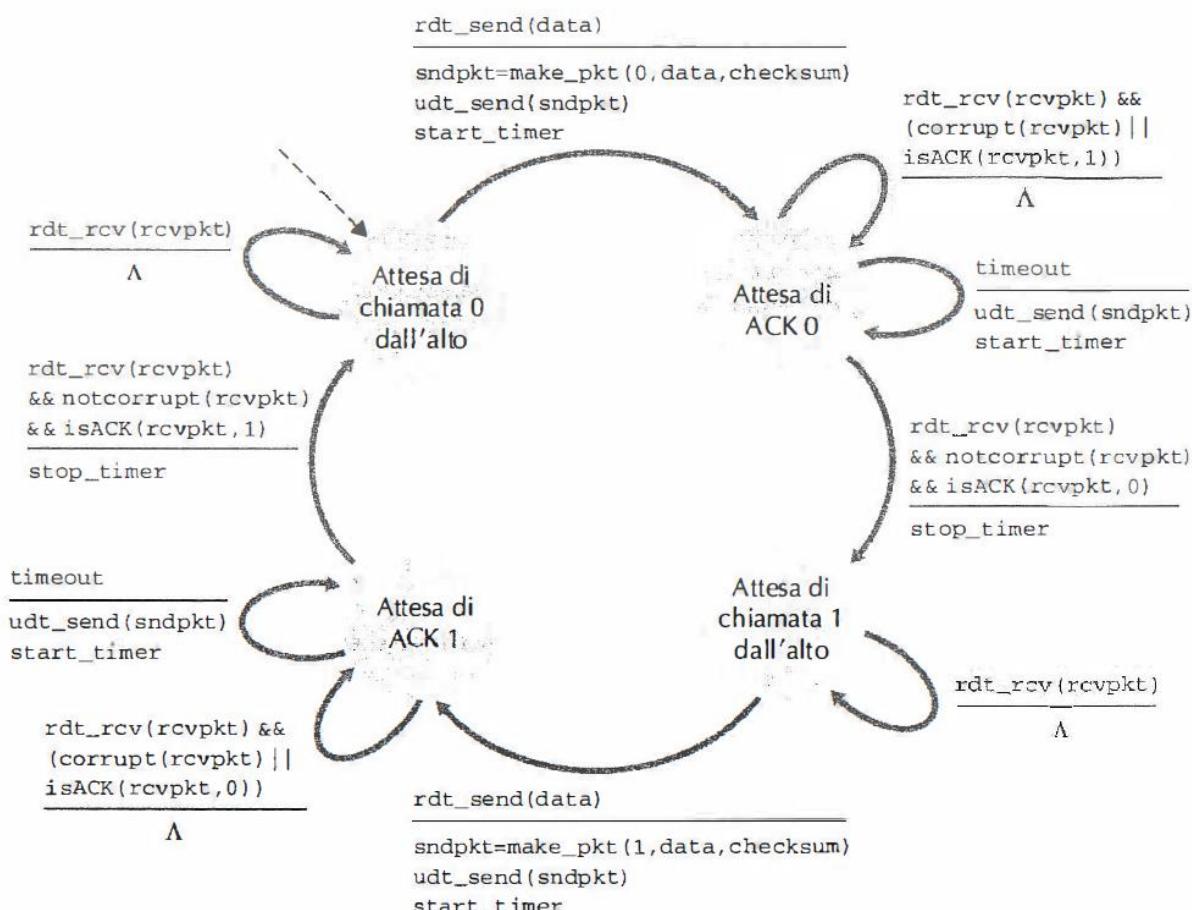
- distinguere duplicati se il trasmettitore invia consecutivamente lo stesso pacchetto
- verificare l'integrità dei pacchetti ricevuti

- verificare l'integrità degli ack ricevuti
- utilizzare solo ack per dare risposte positive e negative

RDT 3.0: SUPPONIAMO UN CANALE SOTTOSTANTE NON PIU' AFFIDABILE CON TRASMISSIONE NON Più ASSICURATA E IN ORDINE

Togliamo anche l'ultima ipotesi ideale. Il canale sottostante, oltre che danneggiare i bit, può anche smarrire i pacchetti. Assegniamo al mittente la responsabilità di rilevare questo fatto. Quindi la parte del ricevitore rimane uguale alla RDT2.2. Ogni volta che invia un pacchetto attiva un timer scaduto il quale reinvia il pacchetto. Questo timer è in generale pari almeno al RTT.

Mittente



Quando deve inviare il pacchetto 0 avvia anche un timer e passa in attesa. Rimane lì se:

- il timer scade e non ha ancora ricevuto alcun ack, il che vuole significare che:
 - il pacchetto si è smarrito: allora il ricevitore è ancora in attesa. Riattivo il timer e rimando il pacchetto.
 - l'ack si è smarrito. Riattivo il timer e rimando il pacchetto. Per l'RDT 2.2, il ricevitore riceve lo stesso pacchetto però.

Supponiamo mandi l'ack del pacchetto0 e si mette nello stato di attesa del pacchetto1. L'ack di 0 si perde. Allora in questo caso il trasmettitore rimanda il pacchetto0 e il ricevitore (ved rtd2.2) vede arrivarsi non pacchetto1 ma pacchetto0 e manda un nack ossia un ack con 0 che arrivato al trasmettitore interpreta come l'ack che gli era mancato.

- Se riceve l'ack ma questo è riferito al pacchetto1 oppure è corrotto. Qui non fa nulla perché aspetta che termina il timer così da rifare quanto detto sopra.

PROTOCOLLI DI RITRASMISSIONE

Prima di parlare del TCP parliamo di quali sono i tre protocolli che nascono da quanto detto sopra.

Esistono 3 tipologie di protocollo di ritrasmissione che non sono necessariamente dei protocolli a livello di trasporto, anzi possono essere utilizzati a diversi livelli della pila protocollare. Vedremo poi come il TCP mischia questi concetti per poi dare vita al suo meccanismo di controllo di errore.

- Stop and Wait
- Go-Back-N
- Selective Repeat

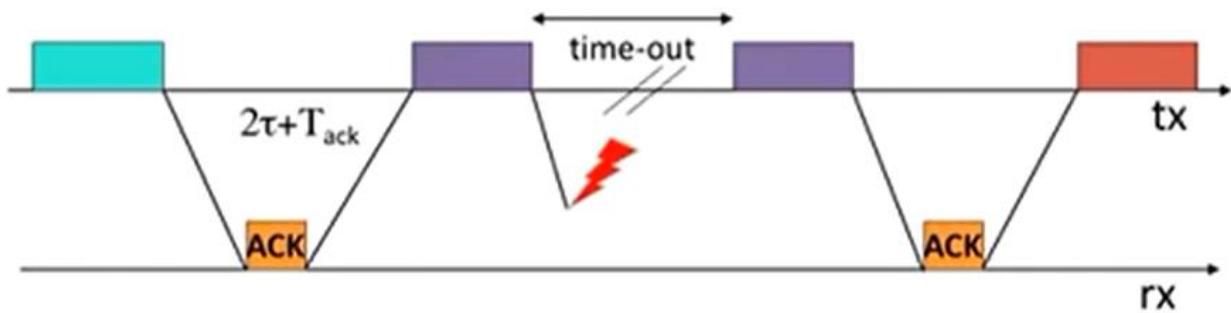
L'rdt3.0 corrisponde allo stop and wait. I successivi due invece si basano sul concetto di **pipeline**, ossia inviare fino a N pacchetti senza dover aspettare nessun ack da parte del ricevitore.

Protocollo Stop And Wait

E' un protocollo che utilizza solo due elementi:

- Solo ACK
- Contatore di time-out

L'idea è semplice: il protocollo dice "puoi inviare un nuovo pacchetto solo se hai ricevuto dal trasmettitore un riscontro positivo". Ma quanto dura questa attesa da parte del trasmettitore?



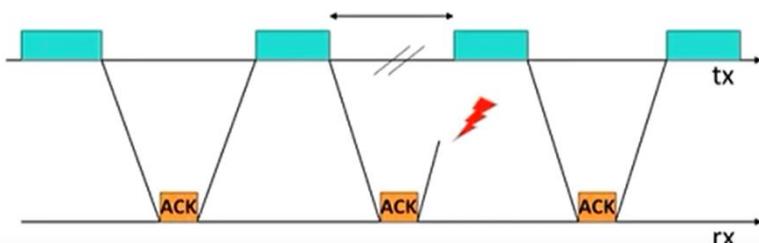
Funzionamento corretto se $\text{time-out} \geq 2\tau + T_{\text{ack}}$

Come vediamo nella figura il trasmettitore deve inviare 3 pacchetti: celeste, viola e arancione. Per ogni invio, il trasmettitore attiva un timer che è il tempo consentito al ricevitore per inviargli l'ACK rispettivo. Se il ricevitore lo riceve entro quel tempo allora passa subito a preparare l'altro pacchetto. In questo caso il pacchetto celeste viene inviato e si riceve un ACK entro il tempo consentito. Si riattiva il timer e si reinvia il pacchetto viola. Il trasmettitore si accorge che è scaduto il timer e non avendo ricevuto alcun ACK allora intende *implicitamente* (da qui l'aspetto implicito del protocollo) che il pacchetto è stato perso o aveva errori e quindi passa a reinviarlo.

Quanto deve essere dimensionato questo *timer*?

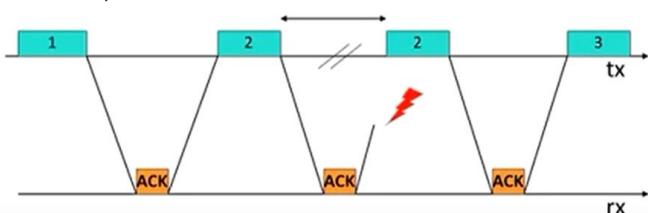
Non deve essere ne troppo breve (altrimenti fraintendo ogni volta che c'è stato errore e reinvio magari pacchetti anche corretti) e ne troppo lungo (altrimenti prima di controllare che mi sia arrivato qualcosa aspetto molto tempo, perdo efficienza). Il valore ottimo è pari a *RTT*.

Ci sono però due problemi.

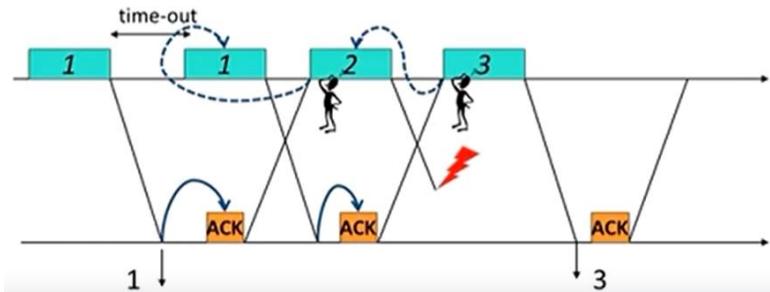


In questa situazione il ricevitore ha ricevuto il primo pacchetto. Lo notifica al trasmettitore, tutto ok. Poi riceve il secondo pacchetto e anche per lui lo notifica. L'ACK però si perde e quindi il

trasmettitore, non ricevendo l'ACK, assume che il pacchetto due sia stato perso quindi lo reinvia. Il ricevitore vede arrivarsi un pacchetto. Se i pacchetti non fossero *numerati* allora il ricevitore non si accorgerebbe mai che ha ricevuto lo stesso pacchetto e lo memorizzerebbe nuovamente (**duplicati**). Quindi i pacchetti vanno numerati (ripeto infatti che lo stop and wait è l'rdt 3.0)



Vediamo quest'altra situazione:

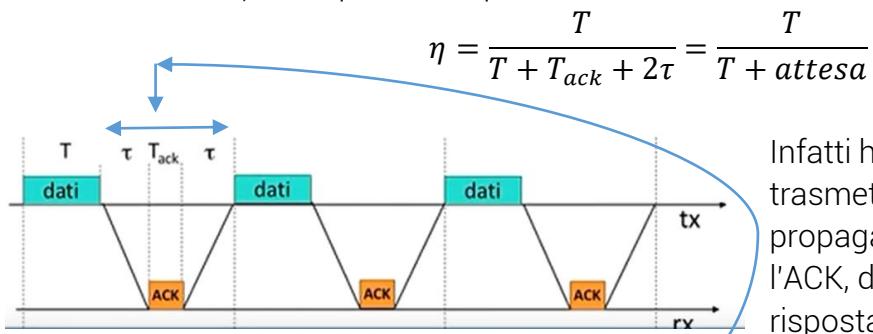


contemporaneamente o subito dopo arriva l'ACK dell'istanza precedente del pacchetto 1 che però viene fraintesa dal trasmettitore come un "OK" sull'ultima istanza del pacchetto 1. Per non **interpretare male gli ACK** bisogna numerare anche loro. Tale numerazione è sufficiente che sia binaria (0,1).

Tale protocollo funziona correttamente, quindi riesce a risolvere i problemi di un tipico protocollo di ritrasmissione: riconoscimento errori e ritrasmissione di pacchetti. Ma quali sono le prestazioni di questo protocollo?

L'**efficienza di un protocollo** misura generalmente qual è la percentuale di occupazione del canale di trasmissione; può essere vista la frazione di tempo per cui il canale è usato per il controllo degli errori.

Nel caso migliore (quindi nessun errore -> invio pacchetto, ricevo riscontro, invio pacchetto, ricevo riscontro...) il tempo di occupazione è:



Infatti ho bisogno di T per trasmettere gli L bit dei dati, di τ per propagarli, di T_{ack} per preparare l'ACK, di ancora τ per propagare la risposta.

Questo è il caso ottimo e il tempo $T_{ack} + 2\tau$ è tempo di attesa, tempo in cui non si fa nulla. Quindi $\eta \in (0,1)$ dove 1 (caso ideale e non raggiungibile) sarebbe l'efficienza migliore in cui quel tempo in mezzo sia nullo, mentre 0 sarebbe quando quel tempo li è davvero grande rispetto al tempo T di trasmissione. Per accorgercene dividiamo tutto per T :

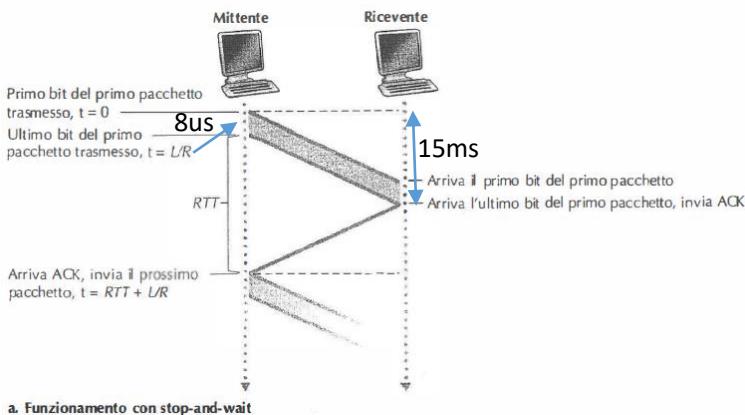
$$\eta = \frac{1}{1 + \frac{T_{ack}}{T} + \frac{2\tau}{T}}$$

Mostra come l'efficienza diminuisce in funzione del tempo di propagazione. Questo mi dice che in sistemi in cui la propagazione del singolo bit è davvero esagerata, il protocollo stop and wait è da evitare.

Vediamo un esempio numerico per capirci meglio.

Supponiamo di voler mandare un pacchetto di 8000bit. Questo viene segmentato in pacchetti da 1000bit ciascuno. Il tempo T di trasmissione è $8\mu s$ mentre il RTT è di $30ms$. In questo caso noi stiamo supponendo per un tempo $T_{ack} = 0$ per semplicità.

Supponiamo che il trasmettitore invii il pacchetto 1. Per qualche motivo il ricevitore, nonostante lo riceve, tarda a inviare l'ACK. Scade però il time-out quindi il trasmettitore torna a inviare il pacchetto 1 ma



Quindi il pacchetto arriva a destinazione dopo $15\text{ms} + 8\text{us} = 15,008\text{ms}$ perché c'è il tempo T di preparazione più 15ms di propagazione che si deduce dall' RTT.
Contando un tempo $T_{ack} = 0$ allora si avranno solo 15ms per spedire l'ACK (si suppone piccolissimo).

Quindi in totale tra primo bit inserito e ricezione dell'ACK è passato un tempo di 30,008ms. In questo tempo abbiamo spedito 1000bit (perché ogni pachetto a tale dimensione). Supponiamo molto fantasiosamente che dopo un tempo T il trasmettitore mette tutti i bit e nello stesso tempo li riceve. Il tempo impiegato a inserire i bit è stato ottimo in confronto al tempo in cui li ha ricevuti perché $\eta = 1$. Quindi il 100% di efficienza. Mentre nel nostro caso l'efficienza è:

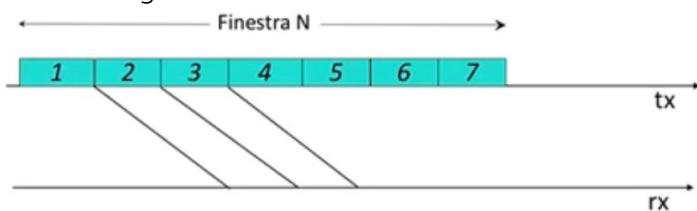
$$\eta = \frac{1}{1 + \frac{0}{T} + \frac{2 * 15}{0,008}} = 0,0027$$

Il che vuol dire che su 8us di tempo impiegato a mettere i bit sul collegamento, a causa del protocollo stop-and-wait, la **natura fisica della linea di trasmissione** ha causato una perdita di efficienza enorme, cioè ho sfruttato i 30ms solamente per lo 0,27%, cioè il restante 99,73% del tempo non è stato sfruttato per inviare alcun dato se non controlli.

Il WIFI usa un protocollo del genere perché di solito la distanza è tra i 10/100 metri e inoltre l'informazione si propaga su un mezzo quasi senza impedimenti quindi il τ è piccolo. Ma anche il Bluetooth funziona così.

Protocollo Go-Back-N

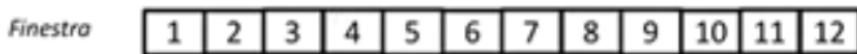
E' un protocollo che estende lo stop-and-wait in cui viene "rilassato" il vincolo "manda il prossimo pacchetto solo quando hai ricevuto l'ACK del precedente". In altre parole si fissa un numero N di pacchetti che si possono inviare prima del riscontro del primo di questi. Tale protocollo è uno degli approcci di invio con **pipeline**, ossia di invio pacchetti senza attendere gli ACK.



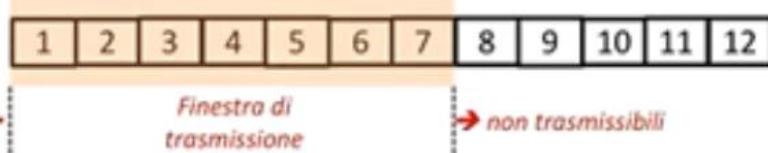
Si definisce **finestra N** proprio questo numero massimo di pacchetti inviabili prima di ricevere il riscontro del primo. Si può vedere lo stop-and-wait come un go-back-N con N=1.

Questo protocollo sfrutta il **principio di scorrimento della finestra (sliding window)** che è più facile da spiegare con un esempio.

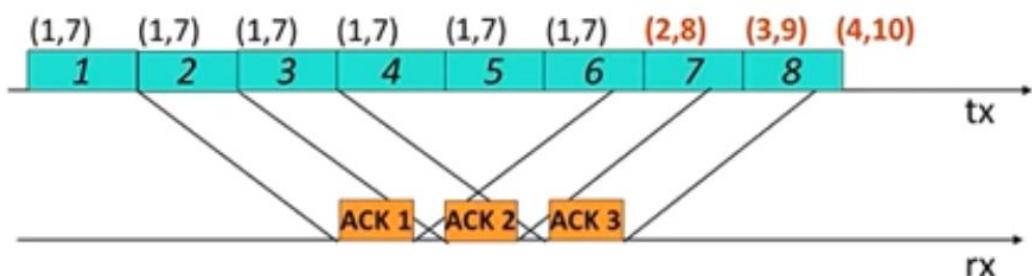
Supponiamo di avere 12 segmenti/pacchetti da inviare.



Si definisce per questo protocollo una finestra N pari a 7.



Questo vuol dire che posso trasmettere fino a 7 pacchetti senza dover per forza aspettare il primo ACK (o NACK). Quello seguente è l'andamento degli ACK (si sta supponendo assenza di errori):



Quindi il trasmettitore trasmette il pacchetto 1,2,3,4,5,6 e prima di trasmettere il 7 gli arriva l'ACK del pacchetto 1. Come si comporta?



il trasmettitore *scorre* la finestra di un posto, quindi i nuovi estremi di trasmissione saranno 2 e 8. Avendo già inviati i pacchetti fino al 6 allora

rimane 7 e 8. Quest'ultimo non era possibile inviarlo prima perché non faceva parte della finestra N.

A un certo punto all'invio del 7 arriva anche l'ACK del secondo pacchetto.



Allora anche questa volta la finestra *scorre* di un posto cambiando nuovamente gli estremi, ovvero i pacchetti

trasmissibili senza preoccuparsi degli ACK dei pacchetti precedenti.

E così via...

Un protocollo del genere, in assenza di errori (quindi in assenza di pacchetti smarriti o altri errori già visti prima) ha un efficienza $\eta = 1$ (100%) perché l'attesa al denominatore è nulla.

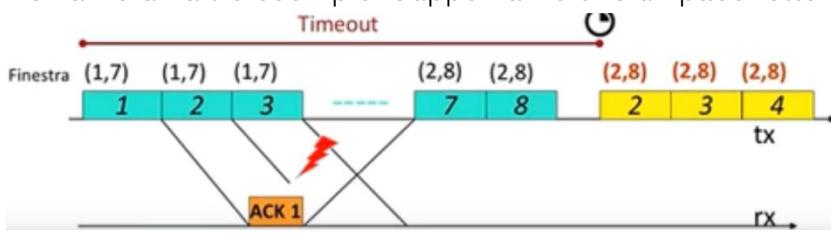
Ma cosa succede in caso di errore?

Dello stop-and-wait **eredita lo stesso concetto di timer/timeout**. E' vero che adesso non devo aspettare l'ack di un pacchetto k per poter mandare quello k+1 e successivi, ma **ogni volta che si invia un segmento, se il timer non è già avviato per un altro segmento, lo si attiva per questo**. Quindi se ho una finestra di N=2:

- Mando segmento0, attivo timer per segmento0

- Mando segmento1, il timer è in uso dal segmento0
- Arriva ack segmento0, libero il timer(*)
- Sposto estremi finestra [1,3]
- Mando segmento2, attivo timer per segmento2
- Mando segmento3, il timer è in uso dal segmento2
- Sono arrivato al limite della finestra. Non mi resta che sperare che arrivi l'ack del segmento2 prima dello scadere del suo timer altrimenti riprendo a inviare dal segmento2.

Vediamo un altro esempio. Supponiamo che un pacchetto venga perso.



La finestra è [1,7].

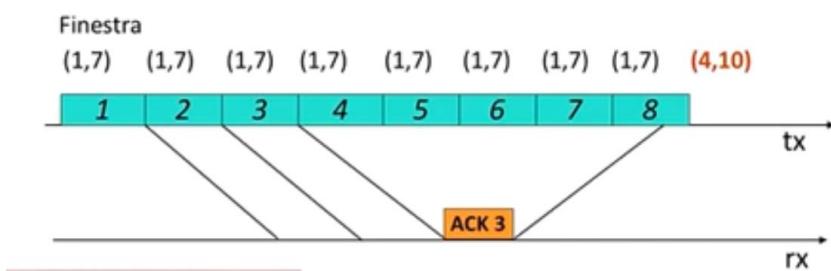
Invio il segmento1 e attivo il timer. Procedo a inviare i pacchetti fino al 7 e per questi segmenti non attivo il timer perché ancora è impegnato

per il segmento1. Il timer arriva proprio all'invio del segmento7. Sposto la finestra [2,8]. Quindi posso inviare un nuovo segmento, l'ottavo. Lo invio e attivo il timer che è libero perché ho ricevuto l'ack del segmento1. Mi fermo perché ho raggiunto il limite della finestra. Poiché il segmento2 è stato perso, non riceverò mai un ack e quindi scadrà il timer del segmento8 e riprenderà a trasmettere dal primo estremo 2. In tutto questo **il ricevitore non ha mai memorizzato pacchetti dal 3 all'8 perché fuori sequenza**. Quindi non ci sono problemi di **duplicati** perché se i pacchetti non arrivano in sequenza non vengono memorizzati.

(*) il timer viene attivato ogni volta che invio se e solo se non è già stato attivato per qualcun altro di cui ancora non è scaduto. Se il timer scade si inviano tutti i pacchetti già inviati (non vuol dire fino al massimo della finestra, ma solo quelli già inviati; poi se si vuole si continua anche con i restanti).

Si dice che in go-back-n il riscontro ha valore **comulativo** o **collettivo**. Che vuol dire?

- Questo, entro certi limiti, rimedia alla perdita di ACK

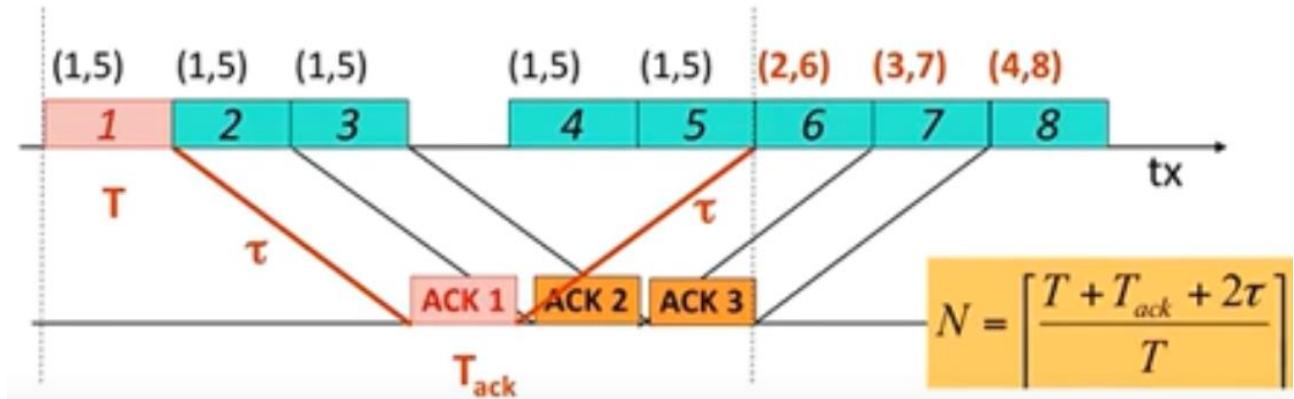


Supponiamo che per certi motivi il ricevitore ricevi sì i tre pacchetti ma notifichi solo la ricezione del 3. Intanto secondo la logica del ricevitore, se notifico un numero 3 allora è ovvio che per lui i precedenti sono già stati memorizzati. Quindi quando arriva l'ACK (ricordiamo che anch'essi sono numerati) numero 3 al trasmittitore ma lui si aspettava quello relativo al pacchetto 1 allora può tranquillamente scorrere la finestra di tre posti.

stati memorizzati. Quindi quando arriva l'ACK (ricordiamo che anch'essi sono numerati) numero 3 al trasmittitore ma lui si aspettava quello relativo al pacchetto 1 allora può tranquillamente scorrere la finestra di tre posti.

Ma quanto devo farla lunga questa finestra N?

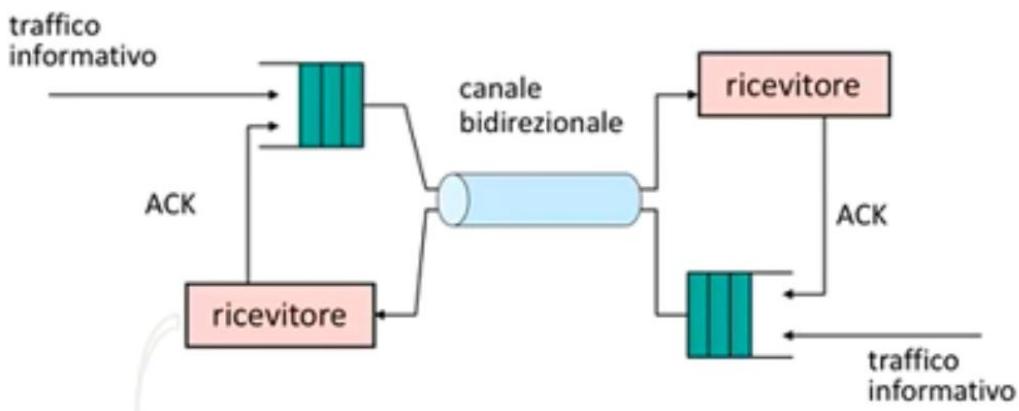
Se N è piccolo ci stiamo avvicinando al caso dello stop-and-wait, quindi perdiamo in efficienza, ma se aumentiamo troppo allora in caso di errore si dovranno ritrasmettere N pacchetti nuovamente. Il valore ottimo è uguale al numero di pacchetti inviabili in $T + RTT$.



N è quindi pari al tempo $T + \tau + T_{ack} + \tau$ (tempo di invio e ricezione notifica pacchetto) fratto T .

GO-BACK-N IN FULL DUPLEX

Questi meccanismi di errore sono in **full-duplex**, cioè esiste un controllo degli errori da *ambo* i lati perché la comunicazione non è solo da trasmettitore a ricevitore ma può anche essere viceversa. Il canale può essere usato quindi anche da parte del ricevitore per inviare suoi pacchetti in contemporanea. Col go-back-n visto prima non si inviavano ack se i segmenti non venivano ricevuti in modo **sequenzialmente corretto** mentre nel full-duplex siccome il ricevitore ha comunque necessità di mandare pacchetti, gli ack (particolari) vengono mandati insieme a segmenti del ricevitore se questi ne ha la necessità di invio.



Quindi quello che abbiamo chiamato trasmettitore e ricevitore non è proprio così, su ogni lato ci sono entrambe le entità. Quindi non ha tanto senso parlare di ricevitore e trasmettitore.

Qui gli ACK possono essere di due tipi:

- ack come li conosciamo;
- segmenti inviati dal senso opposto (full-duplex) che incorporano anche ack per ottimizzare i tempi (invece di separare ack e segmenti).

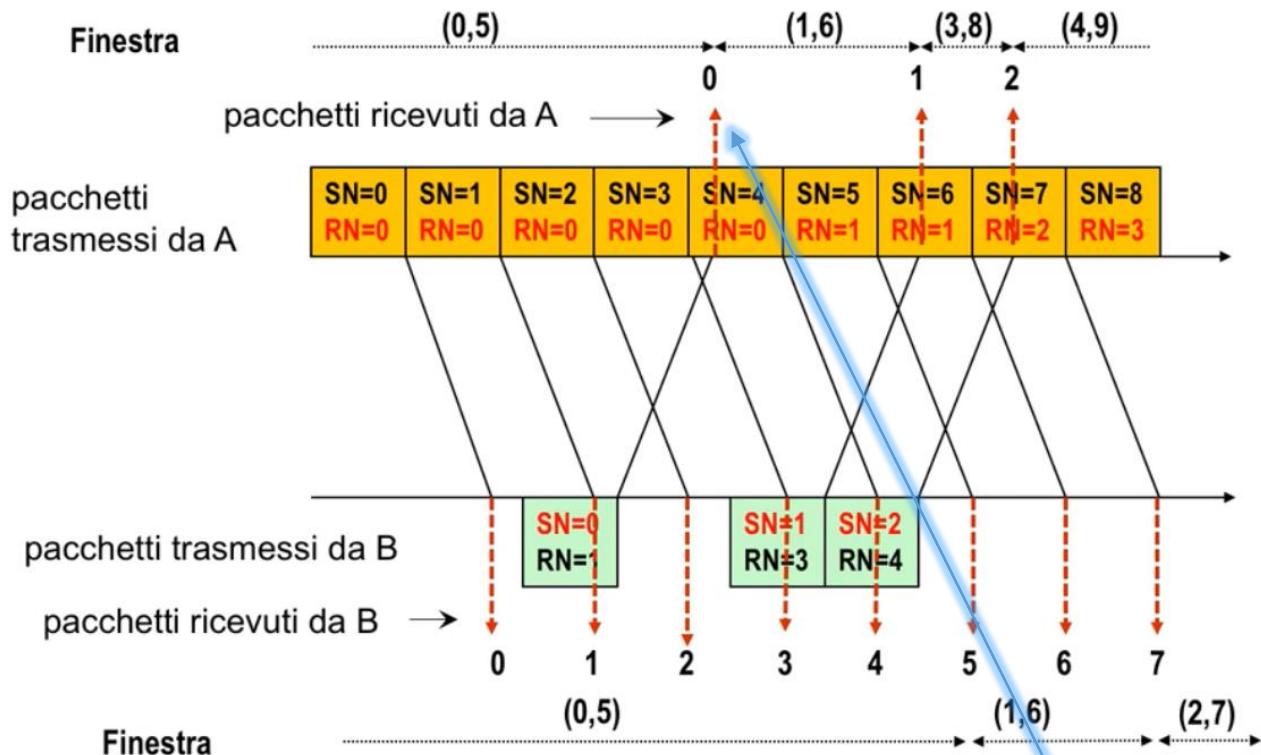
Quindi in generale non c'è più differenza tra ack e segmento che vuole portare info.

L'insieme forma segmenti del tipo:



Dove SN indica "ti sto inviando il pacchetto..." e RN indica "mi aspetto di ricevere il pacchetto...".

Vediamo un esempio. Per semplicità consideriamo una situazione senza errori.



Come possiamo notare ormai utilizziamo A e B invece di trasmettitore e ricevitore. Ognuno ha una propria finestra.

In questa situazione A inizia a trasmettere. Trasmette il suo pacchetto che porta negli header anche quelle due informazioni che in questo caso sono SN=0,RN=0, come a dire che ha inviato il pacchetto0 e mi aspetto ancora il pacchetto0, continua inviando il pacchetto1 (SN=1) e continuando ancora ad aspettare il pacchetto0 (RN=0). Continua così fino al pacchetto4. In quell'istante gli arriva un primo pacchetto da parte di B che porta due informazioni. SN=0 indica che quello è il pacchetto0 che gli sta mandando e RN=1 indica che ha memorizzato i suoi pacchetti dallo 0 all'1. Quindi A può scorrere la finestra a nuovi estremi.

A allora quando invia il pacchetto 5 (SN=5) mette RN=1 a indicare che adesso si aspetta di ricevere il pacchetto numero1 (dicendo implicitamente che quello zero lo ha memorizzato). Quando B riceve questo pacchetto può anche lui scorrere la sua finestra.

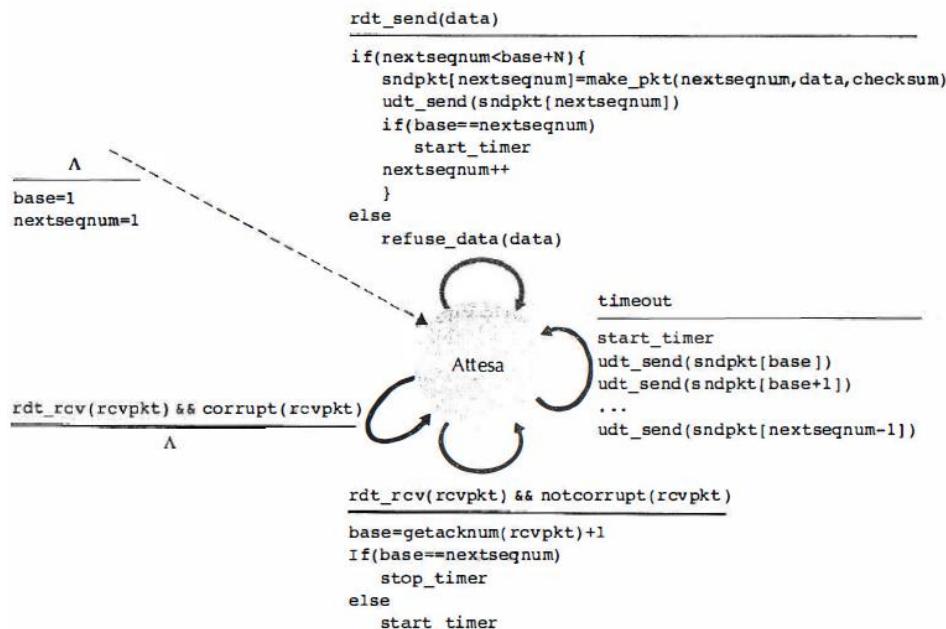
Questo modo di sfuttare gli ack per mandare propri pacchetti si dice essere in piggyback e lo vedremo col TCP.

GO BACK N MACCHINA A STATI

Simulazione GO-BACK-N

<http://computerscience.unicam.it/marcantoni/reti/applet/GoBackProtocol/goback.html>

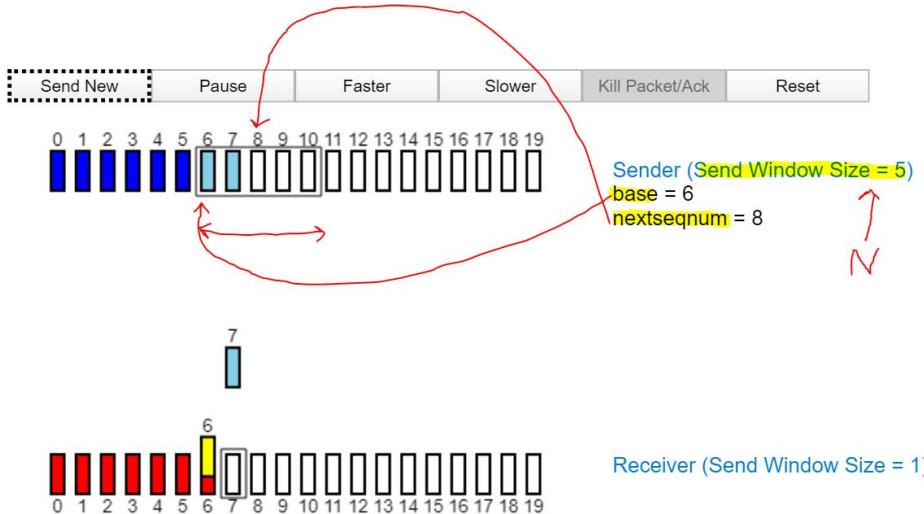
Mittente



Descrivo ogni funzione che viene eseguita ad ogni evento.

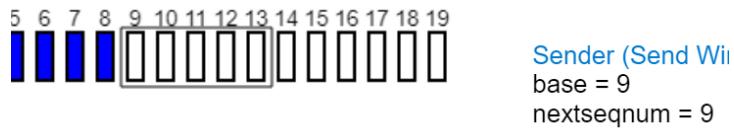
1) Inizialmente sto in attesa ponendo:

- Base=1: è un indice che dinamicamente indica la finestra [base, base+N-1]
- Nextseqnum=1: indica il numero di pacchetto (SN) prossimo da inviare
- N il numero massimo di pacchetti inviabili senza ricevere ack



2) L'applicazione vuole inviare dei dati dall'altra parte.

- o Se il nextseqnum rientra nell'intervallo [base,base+N-1], quindi se la finestra non è piena, allora si crea il solito pacchetto e lo invia. Se la base è
 - uguale al prossimo da inviare (questo succede solo quando si è all'inizio come mostrato nella figura qui sotto)



Allora si attiva il timer.

- Altrimenti si invia il segmento ma senza attivare il timer (ci penserà la ricezione dell'ack a farlo)
 - Se la finestra è piena, l'invio viene rifiutato. In realtà si potrebbe inserire in un buffer di invio e riprenderlo quando si libera la finestra.
- In ogni caso si incrementa nextseqnum.
- 3) Allo scadere del timer, se non si è ricevuto l'ack per un segmento dentro [base, base+N-1] allora si rinvia dalla base a nextseqnum-1

Esempio



Sender (Send Window Size = 5)
base = 5
nextseqnum = 9

Invio 4 pacchetti.



Receiver (Send Window Size = 1)



Sender (Send Window Size = 5)
base = 5
nextseqnum = 9

Si perdonano il 7 e l'8. Idem se si perdonano i relativi ack.



Receiver (Send Window Size = 1)

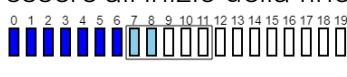


Sender (Send Window Size = 5)
base = 5
nextseqnum = 9

Sender (Send Window Size = 5)
base = 7
nextseqnum = 9

Arrivano gli ack di 5 e 6 e la finestra si sposta di uno ogni volta che riceve l'ack. Per quello che

vedremmo al punto 4) ad ogni ack ricevuto si riattiva il timer (a meno di non essere all'inizio della finestra), quindi arrivato il 6 parte il timer per il 7.



Sender (Send Window Size = 5)
base = 7
nextseqnum = 9



Receiver (Send Window Size = 1)

Scaduto il suo timer vengono reinviati da base a nextseqnum-1.

NB: di solito si inviano tutti gli N pacchetti alla volta. Ma in generale, scaduto il timer si rimandano solo quelli inviati prima, ossia da base a nextseqnum-1

4) Se si riceve un ack:

- Se è corrotto non si fa nulla
- Se è integro, se l'ack è relativo al pacchetto di indice base allora si stoppa solo il timer.
- Se è integro ma l'ack si riferisce a un pacchetto precedente ad altri inviati allora si riattiva il timer.

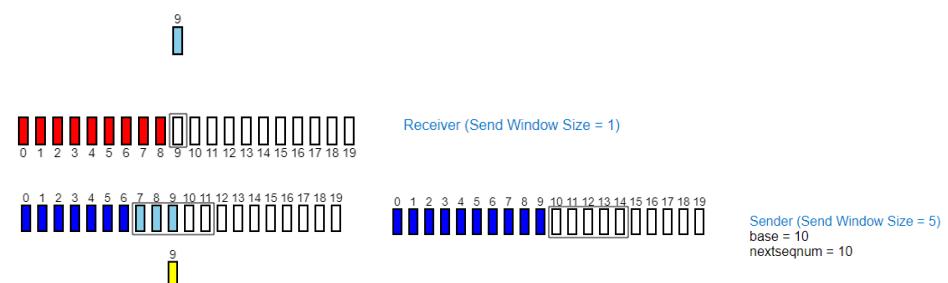
In generale, se è integro si attiva il timer ogni volta che si riceve un ack, tranne quando non esistono altri pacchetti successivi inviati e in attesa di ack.

NB: la riga `base = getacknum(rcvpkt)+1` implementa quello che abbiamo chiamato **ack cumulativo**.

Esempio



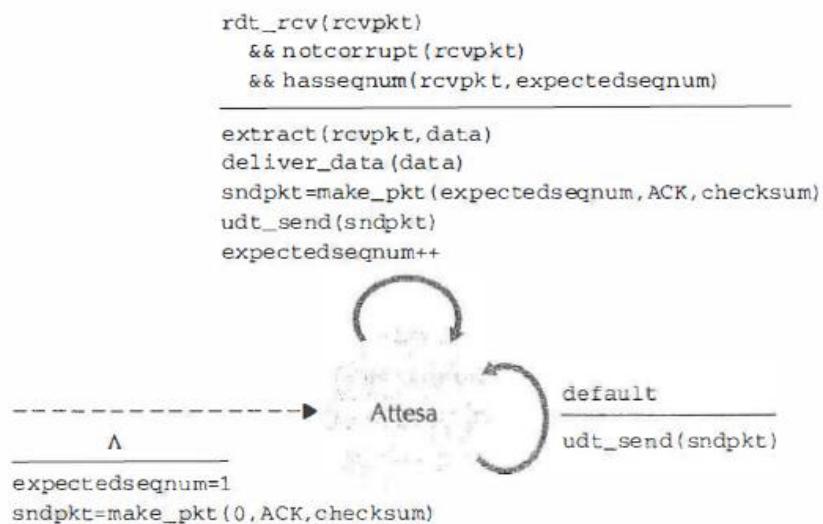
Viene inviato il segmento 9. Il timer è ancora occupato per il 7.



Viene inviato il segmento 9. Il timer è ancora occupato per il 7.

L'ack del segmento 9 è stato ricevuto correttamente ed è cumulativo. Infatti la finestra si sposta [10,14] e per il 2) si stoppa il timer.

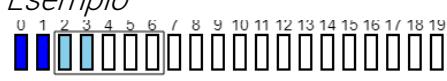
Ricevente



Qui la variabile *expectedseqnum* evita di inserire tanti stati e indica il numero di pacchetto che si aspetta di ricevere.

- 1) All'inizio è 1 (la base del ricevitore infatti è 1, si sono accordati).
- 2) Quando arriva un pacchetto, se non è corrotto e se è quello che si aspetta (questo permette la memorizzazione dei pacchetti solo se in **sequenza**) allora lo estrae, lo consegna all'applicazione e crea e invia un pacchetto con checksum, con ACK (che sarebbe l'SN del ricevitore) e l'*expectedseqnum* per dire al mittente quale pacchetto si aspetta e indirettamente ha funzione **cumulativa** perché gli indica che tutti i pacchetti con SN<*expectedseqnum* sono arrivati correttamente.
- 3) Il *default* indica che in tutti gli altri casi diversi da 1) e 2) il ricevitore manda l'ack del pacchetto più recentemente memorizzato. Perché?

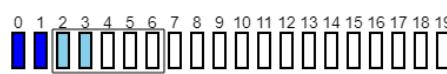
Esempio

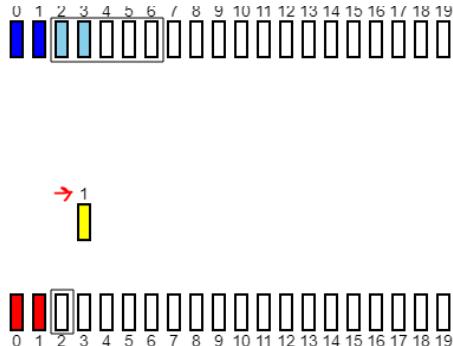


Il ricevitore si aspetta il pacchetto 2 che sta arrivando.



Poi però il pacchetto 2 si smarrisce ma il ricevitore ha *expectedseqnum* ancora pari a 2. Il pacchetto 3 gli arriva, ma genera come vediamo dopo con un ack riferito al 3.





Come possiamo vedere, dal 2) il pacchetto non è nella sequenza corretta e quindi il 3) di default produce l'ack relativo all'ultimo pacchetto ricevuto correttamente, ossia l'1. Quando arriva al mittente infatti `base = getacknum(rcvpkt)+1` fa sì che `base=1+1=2` e quindi la situazione rimane inalterata.

Protocollo Selective Repeat

E' un protocollo che cerca di risolvere un problema fondamentale del protocollo go-back-N. Se ricordiamo bene, il trasmettitore inviava pacchetti fino alla disponibilità della finestra N. Di contro il ricevitore li memorizzava e inviava un ACK per notificare il pacchetto o un gruppo di essi (cumulativo). Però poteva capitare che un pacchetto k venisse perso. Pertanto se i pacchetti i-esimi, con $i > k$, arrivavano al ricevitore, questo li **scartava**. Quando scadeva il timer allora il trasmettitore si accorgeva che c'era stato un errore e quindi riprendeva a **ritrasmettere** tutti i pacchetti. E' proprio qui che entra in gioco il selective-repeat. Perché ritrasmettere pacchetti che magari erano arrivati correttamente ma il ricevitore aveva scartato perché non erano nella *sequenza corretta*? Quindi più che altro con questo protocollo il ricevitore quando si vedrà arrivare un pacchetto *fuori sequenza non lo scarterà* stavolta, ma lo memorizzerà in buffer in attesa così che il trasmettitore si accorga che c'è stato un errore e ritrasmetta solo i pacchetti mai arrivati. Si noti che con questo protocollo non ha più senso parlare di *ACK cumulativi*. Cambiano le strutture dei riscontri. Non abbiamo più ACK singoli ma **ACK+bitmap** delle trame ricevute o meno.



Dove RN dice "la prima trama persa è la 5" mentre il bitmap, riferendo la prima posizione appunto alla RN=5 indica con 1 quelle ricevute e con 0 quelle perse. Questo è ogni riscontro inviato.

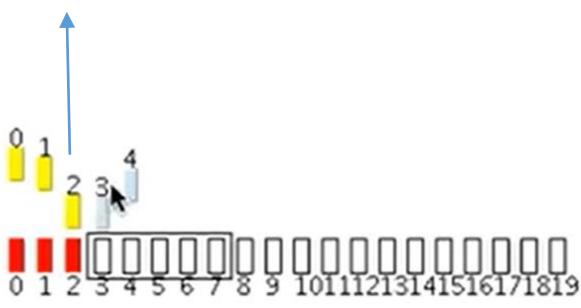
Vediamo un esempio che ci chiarirà i dubbi.



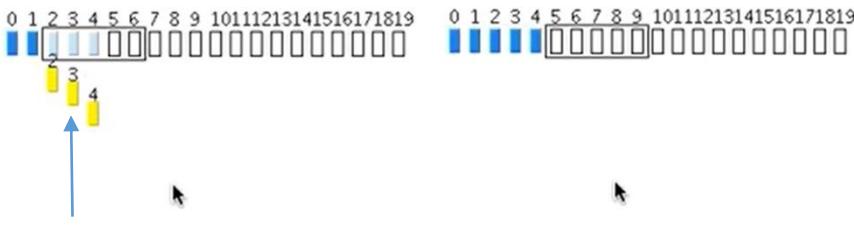
Abbiamo una finestra N di 5.



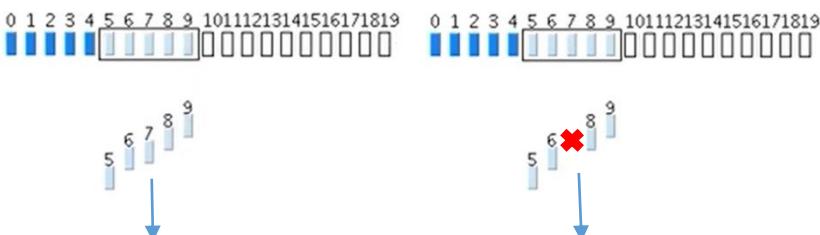
Il trasmettitore invia tutti e 5 i pacchetti



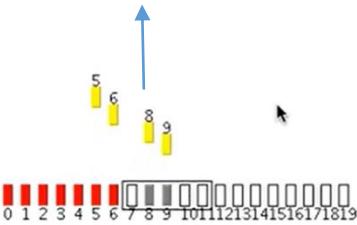
Ogni volta che al ricevitore arriva un pacchetto correttamente manda il riscontro (in giallo). Nota come il riscontro è mandato per ogni pacchetto (no cumulativo).



Ad ogni riscontro ricevuto, il trasmettitore scorre la propria finestra. In questa situazione ha ricevuto il riscontro di tutti e 5 i pacchetti.



Il trasmettitore riprende la trasmissione inviando altri 5 pacchetti ma per qualche motivo il pacchetto 7 viene smarrito.



Il ricevitore vede arrivarsi il 5 e il 6 che sono **in sequenza** dunque li memorizza normalmente. Dato che manca il 7, allora i successivi 8 e 9 sono **fuori sequenza** quindi li memorizza in un **buffer**(in grigio) in attesa della ritrasmissione del pacchetto 7.

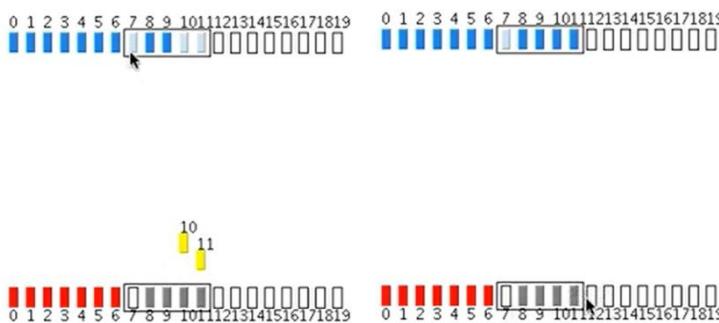
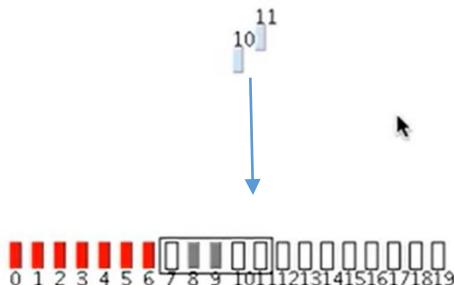


Ricevuto l'ACK del 5 scorre la finestra, poi del 6 e la scorre ancora. Poi riceve quella dell'8 e del 9 ma come possiamo vedere resta bloccata alla 7 anche se memorizza la ricezione degli ACK 8 e 9.





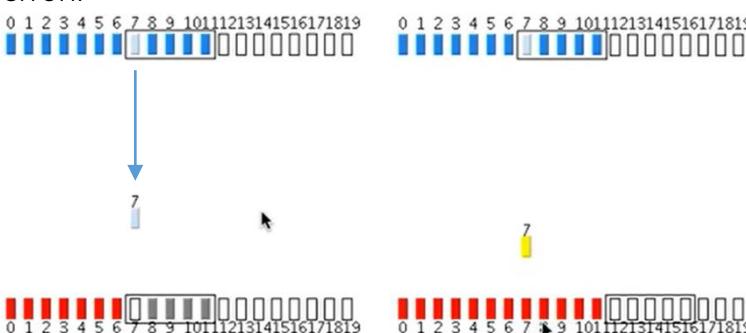
Nel frattempo, col 5 e 6 la finestra ha permesso, scorrendo, di inviare anche i pacchetti 10 e 11.



Il ricevitore continua a ricevere pacchetti fuori sequenza che memorizza nel buffer. Il trasmettore riceve gli ACK dei pacchetti 10 e 11. La finestra non scorre perché la regola è: la dimensione della finestra deve essere sempre pari a N indipendentemente se i pacchetti sono stati ricevuti o meno.

Nonostante si possa scorrere all'arrivo di 10 e 11, non lo si può fare perché l'estremo sinistro continua a essere 7 e dal 7 fino all'11, indipendentemente dai pacchetti i cui ACK sono stati ricevuti (come 8 e 9) o meno (come il 7), il numero deve sempre essere N (5).

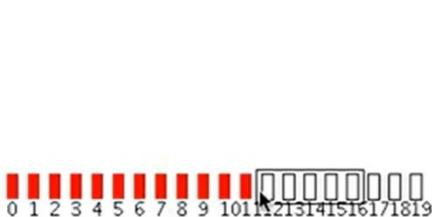
Allora si arriva al punto in cui **scade il timeout** e il trasmettitore trasmette i pacchetti con errori.



Il ricevitore riceve il pacchetto smarrito (7) e **sblocca** tutti quelli memorizzati nel buffer mandando l'ACK del 7.



All'arrivo dell'ACK del 7 a finestra si trova completate tutte e N e quindi si sposta.



Siccome non serve un estremo dal quale partire a inviare tutti i pacchetti come col GBN ma vengono inviati solo i pacchetti effettivamente persi, allora ogni pacchetto ha un suo timer logico scaduto il quale lo si rimanda.

Controllo di flusso

Come fanno due entità che comunicano a controllare il flusso, cioè a controllare il rate? Il ricevitore può avere una capacità di ricevere informazione diversa (non sincrona) rispetto al



trasmettitore, quindi quest'ultimo butta informazioni sul canale (di norma) a rate diversi da quello con cui il ricevitore riesce a processare. Questo nella pratica vuol dire che

ogni ricevitore avrà un **buffer** limitato a W posizioni.

Se il ritmo di lettura dalla coda è più lento del ritmo di scrittura nella coda allora possono avere problemi di **buffer overflow**. Il **controllo di flusso** serve proprio a evitare questo overflow, e ciò si ottiene semplicemente limitando il rate del trasmettitore.

Per fare questo si usa proprio la **finestra di scorrimento mobile** sul trasmettitore che deve essere dimensionata esattamente quanto il buffer di ricezione. Anche qui la finestra scorre quando arriva un riscontro al trasmettitore, ma questo riscontro è quello che arriva non dal fatto che il ricevitore ha ricevuto correttamente il pacchetto e quindi lo memorizza nel buffer (in grigio, ricordate?), ma quando il pacchetto viene tolto dal buffer ed effettivamente letto.

Quindi in controllo di flusso il trasmettitore invia i pacchetti quanto è la finestra N, dimensionata quanto il buffer, e il ricevitore per quanto memorizza nel buffer i pacchetti, genererà l'ACK solo quando lo estrae dal buffer e solo a quel punto la finestra del trasmettitore può scorrere.

TCP: Trasmission Control Protocol

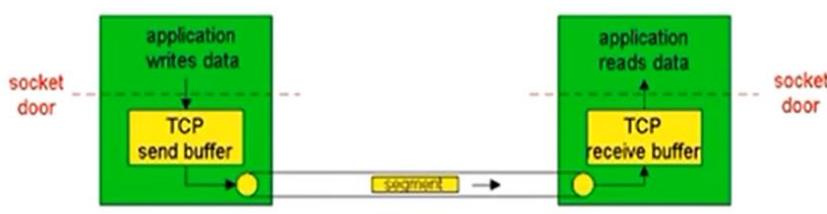
E' un protocollo di trasmissione che garantisce la consegna:

- *in sequenza*: tutto ciò che viene emesso dal trasmittitore viene ricevuto nella stessa sequenza dal ricevitore
- *nessuna perdita di dati o errori*
- *controllo di flusso e congestione*

E' un protocollo *orientato alla connessione*, ossia due entità in TCP possono scambiarsi dati se e solo se prima viene fatto un setup/handshake. Non confondiamo l'orientamento alla connessione con la commutazione di circuito perché TCP si basa su una rete a pacchetto, ciò che sta sotto è "l'internet", quindi il fatto che la connessione debba essere istaurata avviene solo a livello 4, sotto ho comunque protocolli che operano a pacchetti. I pacchetti che si scambiano vengono detti **segmenti**, che sono il risultato della conversione del flusso di dati proveniente dal livello applicativo. Genericamente le connessioni sono

- **full-duplex**
- **punto a punto**: cioè la TCP ha luogo tra un singolo mittente e singolo destinatario, non può dunque essere *multicast* (ossia un mittente e molti destinatari)

FLUSSO DI DATI

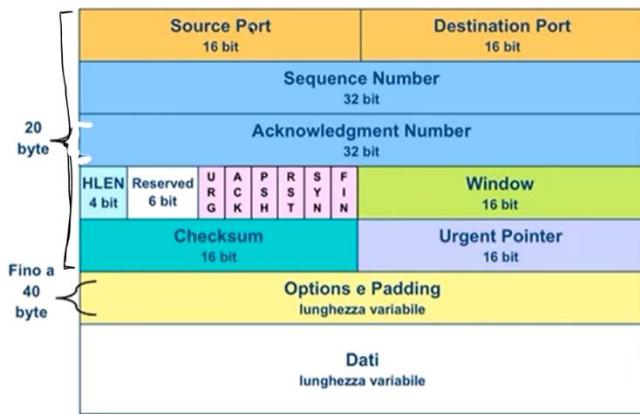


L'applicazione (quindi il livello applicativo) trasmette i dati (flussi di byte) al TCP che li accumula in un **buffer**. Quando avvengono particolari condizioni il TCP prende una

parte dei dati nel buffer e forma un **segmento** (nota come l'applicazione non ne decide le tempistiche, lascia decidere al TCP quando inviare in rete i dati). La massima dimensione di ciascun segmento è detta **MSS** (maximum segment size), impostata in funzione di quale sia l'unità trasmissiva massima dal mittente al destinatario, detta **MTU** (ovviamente considerando non solo il payload ma anche l'header). Il **buffer di ricezione** del ricevente invece contiene tutti i segmenti inviati (ovviamente ha fisicamente un limite).

FORMATO DEI SEGMENTI

Nella UDP c'era il checksum che si preoccupava dell'integrità del pacchetto, qui vediamo come invece il singolo segmento TCP porta più informazioni.



come l'UDP si ha la

- **source port**
- **destination port**

assegnate in maniera statica ai processi
lato server e dinamica a quelli client.

L'UDP col suo checksum si limita a garantire l'integrità del pacchetto ma non l'affidabilità, cioè ci assicura che se il pacchetto arriva allora è integro, mentre il

TCP toglie il dubbio a quel se: il pacchetto arriva sempre.

E per questo ha bisogno dei due seguenti campi:

- **sequence number (SN)**: numero di sequenza del primo byte del segmento (utilizzato dal mittente e dal destinatario per marcare l'iesimo segmento. Immaginalo come l'identificativo di ogni segmento)
- **Acknowledgment Number (ACK o AN)**: indica il prossimo numero di sequenza che mi aspetto (utilizzato dal mittente e destinatario per dare conferma che il precedente è stato ricevuto e che adesso si aspetta uno con SN pari a AN). NB: Sarebbe AN ma a volte lo tratta come ACK. Per me ha più senso AN perché si confonde col bit ACK che vedrete dopo ma si usa spesso anche ACK. In ogni caso se vedete ACK con valore maggiore di 1 allora quello è l'aknowledgemtn, altrimenti è il flag ACK.

Per il controllo di flusso mi serve uno spazio che il ricevitore dovrà riempire per dire al trasmettitore qual è lo spazio disponibile nel suo buffer di ricezione, e questo campo è:

- **Window**

Come l'UDP dovrà garantire l'integrità dei bit ricevuti

- **Checksum**

Poi vi sono altri campi come:

- **HLEN**: che in 4 bit indicherà la lunghezza dell'header perché quest'ultimo può anche contenere *Options e Padding*.
- i **flag URG,ACK,PSH,RST,SYN,FIN**: che servono a implementare diverse procedure di segnalazione, per esempio per il setup della connessione, per l'abbattimento della connessione etc... in particolare:
 - **URG**: il segmento viene marcato come urgente.
 - **ACK**: viene usato per indicare che il valore trasportato nel campo di acknowledgement (AN) è valido, questo indica al ricevente che l'informazione AN indica quale segmento si aspetta di ricevere e che i precedenti sono stati recepiti con successo.
 - **PSH**: indica al trasmettitore che i dati in arrivo non devono essere bufferizzati ma subito mandati al livello applicativo.

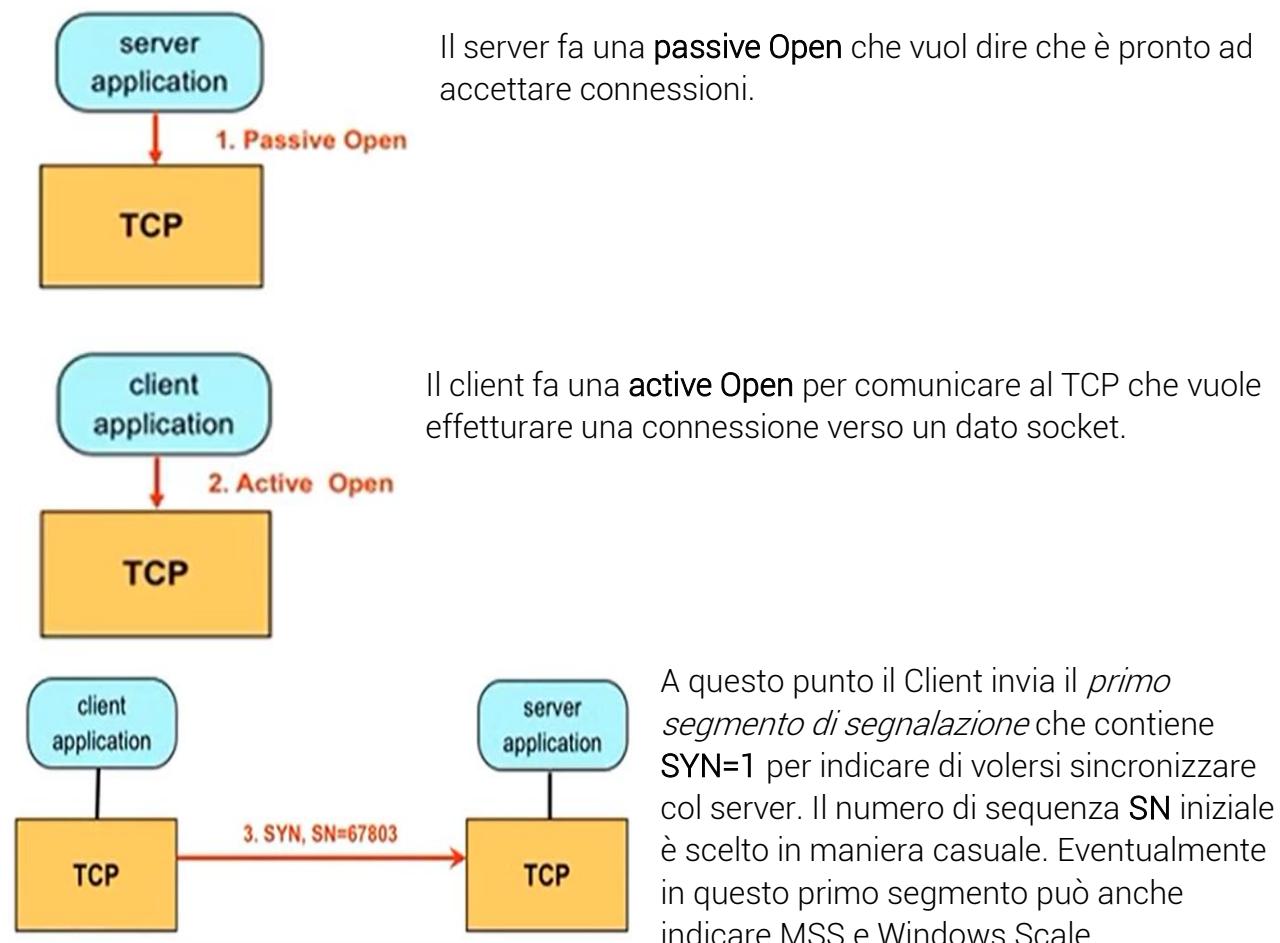
- **SYN**: indica se il segmento è un segmento che serve solo alla connessione TCP (quindi generalmente non hanno payload).
- **RST e FIN**: usati per l'abbattimento della connessione.
- **Options e Padding**: contengono un'arricchimento delle segnalazioni fatte. Vengono usate per contrattare la MSS o il fattore di scala della finestra, ma anche per riempire l'header in modo da essere un multiplo di 32 bit.
- **Dati**

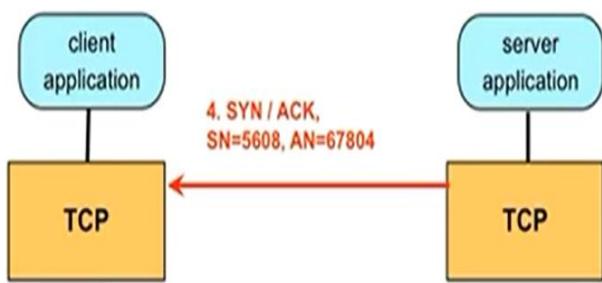
L'ACK può essere trasmesso in un messaggio a sé stante, o essere inviato in un pacchetto utente in direzione opposta contenente anche delle info (caso full duplex) da parte del ricevente, ossia in **piggyback**(lo abbiamo visto nelc aso dell'http).

SETUP DELLE CONNESSIONI

(qui l'ACK è il flag, AN è l'acknowledgement)

Prima della *call setup*, ossia della connessione vera e propria, le applicazioni lato client e server devono comunicare con il software TCP.



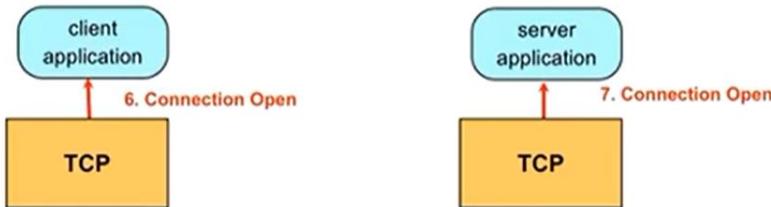


Se il server accetta, per notificarlo invia un *secondo segmento* con **ACK=1** a indicare che la validità dell'**AN** è **SYN=1** a indicare di aprire una connessione nel senso opposto perché anche lui vuole inviare dati (la TCP è **full-duplex**). Imposta **AN=67804** a indicare che il precedente è stato ricevuto e aspetta il prossimo. Poiché la comunicazione è full-duplex anche il server dovrà scegliere un suo **SN**. Alla ricezione di questo segmento però, la connessione è ancora unidirezionale, da sx verso dx.



Il Client accetta la connessione full-duplex inviando un *terzo segmento* con **SN=67804** e confermando quello precedente con **AN=5609** la cui validità è sempre indicata da **ACK**.

Come possiamo notare, prima di inviare i veri e propri dati si deve stabilire una connessione che prevede l'invio e la ricezione di tre segmenti e per questo è detta **handshake a tre vie**.



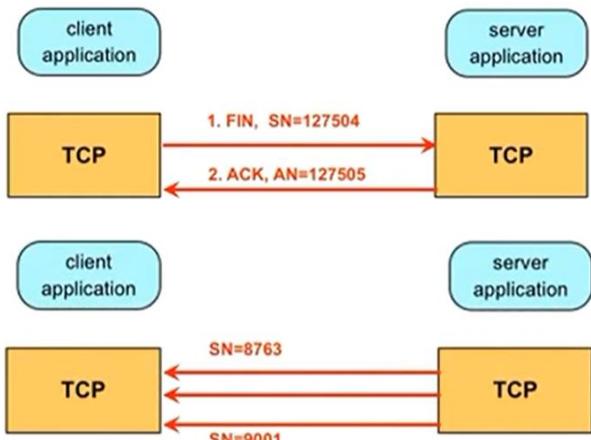
Alla fine di questo handshake ci saranno delle primitive che segnaleranno al livello applicativo l'apertura o meno della connessione.

TEAR DOWN (CHIUSURA) DELLE CONNESSIONI

Vediamo come si butta giù la connessione.

La chiusura può essere:

- **esplicita** quando *una delle due* esplicitamente invia un segmento con **FIN=1** e dall'altra parte si riceve conferma.



NB: la chiusura può anche essere voluta dal server. In questo esempio è voluta dal Client.

Il flag final *chiude la connessione solo nella direzione di chi invia il segmento con FIN=1* quindi nell'esempio (1) il server può comunque continuare a inviare segmenti. Nell'esempio (2) infatti invia vari segmenti. L'altra parte dovrà comunque generare riscontri, solo che questi riscontri non conterranno dati (payload).



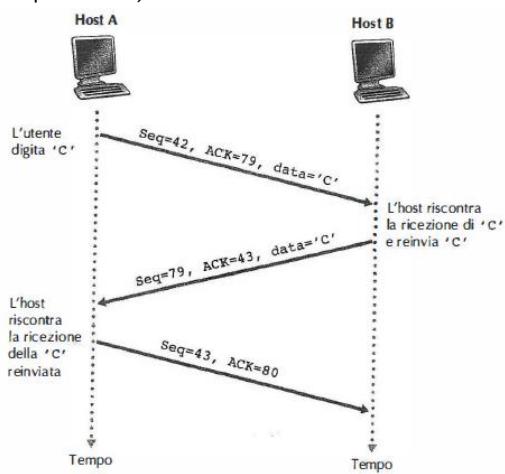
Appena anche l'altra parte vuole chiudere la connessione allora questa è chiusa da ambo i lati definitivamente.

- **Brutale:** chi vuole chiudere brutalmente la connessione senza neppure avere un riscontro setta il flag RST=1 (reset).
- **Implicita:** viene stabilito un timeout tale per cui se entro un tot non si scambiano dati la connessione viene abbattuta.

NUMERI DI SEQUENZA (SN)

Nei precedenti esempi ho trattato l'identificativo SN di ogni segmento come fatto con l'UDP. Ma abbiamo detto che SN indica il sequence number del primo byte del segmento.

Supponiamo che dobbiamo inviare 500.000 byte di dati e il TCP li impacchetta in 500 segmenti con MSS=1000byte. Il primo segmento avrà SN=0, perché il suo primo byte è il primo byte ad essere inviato dei 500.000 byte totali. Il secondo segmento avrà SN=1000 perché una volta che si sono inviati tutti i 1000 byte del primo segmento, il secondo avrà il suo primo byte che sarebbe il millesimo-e-uno dei 500.000. E così via. Si noti che non vengono mica numerati i singoli byte di ogni segmento, ma se si sa che ogni segmento consta di 1000 byte allora se il destinatario ne riceve solo 500 e manda un ACK con AN=501 allora è come se implicitamente tutti i singoli byte sono numerati e il mittente capisce che una metà non sono arrivati (nel laboratorio vedremo come si sfrutterà questo meccanismo per capire, dal valore ritornato dalla recv, che si sono ricevuti meno byte di quelli che ci si aspettava).

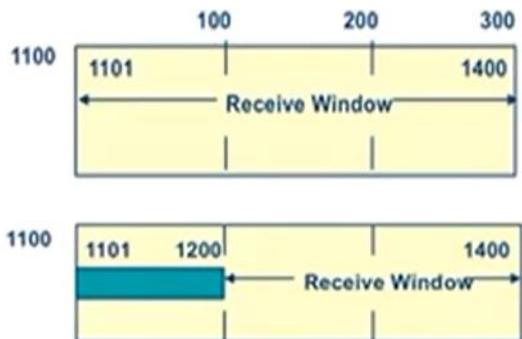


In questo esempio il client sceglie a random un sequence number 42 e il server 79. Nota che i sequence number sono relativi ai byte dei payload perché dall'applicazione arrivano flussi di dati, quindi payload. Quando il server lo riceve manda il suo messaggio con SN=79 e AN=43 perché si aspetta il prossimo carattere. In questo caso infatti il segmento ha SN=42 e il successivo SN=43 perché il payload è formato dal carattere ASCII di 'C' che sta su 8 bit e 42 sarebbe l'SN del primo byte.

CONTROLLO DI FLUSSO

Il controllo di flusso serve per controllare il rate del trasmettitore sulla base della capacità del buffer del ricevitore nell'assorbire dati. Nel campo **window** il ricevitore segnala, durante l'istaurazione della connessione, quale sia la capacità del proprio buffer.

Si definisce **Receive Window** lo spazio disponibile nel buffer di ricezione del ricevitore per nuovi dati. Quindi non è lo spazio disponibile totale, ma quello che ogni volta rimane disponibile.



al trasmettitore.

Si memorizza la larghezza totale in **RcvBuffer** e questa resta tale. Si definisce invece **LastByteRcvd** come l'ultimo byte inserito nel buffer di ricezione e **LastByteRead** come l'ultimo, tra i byte inseriti, ad essere letto. Nell'esempio (2) sopra avremmo:

RcvBuffer = 300

LastByteRcvd = 1200

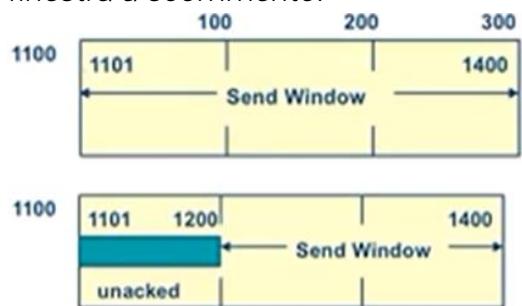
LastByteRead = supponiamo sia 1150.

Quello che passeremo in **WindowSize** sarà:

$$\text{rwnd} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}] = 300 - [1200 - 1150] = 250$$

Quindi il buffer di ricezione può ospitare ancora 250 byte.

Si definisce **Send Window** la parte inutilizzata del buffer di invio, ossia i byte che possono essere trasmessi senza attendere ulteriori riscontri. Sarebbe quella che chiamavamo *finestra a scorrimento*.



All'inizio è massima. Nessun byte inviato.

Poi inizio a inviare i byte da 1101 a 1200 e quindi la SendWindow si riduce.

Se ricevo un riscontro sul byte 1200 allora la finestra si allarga e diventa di estremi [1201,1500] (quindi ripristina la capienza di 300 originaria).

L'host mittente si salva due variabili: **LastByteSent** come l'ultimo byte mandato e **LastByteAcked** come l'ultimo byte di cui si è ricevuto l'ack. Il ricevente invia ogni volta rwnd dentro window.

Se quindi rwnd è lo spazio rimanente nel buffer del ricevente, allora il trasmettitore, per evitare che il ricevitore vada in buffer overflow, deve sempre verificare che **LastByteSent**-**LastByteAcked** (che sarebbe quindi il numero di messaggi ancora nel buffer del ricevitore) siano minori o uguali a rwnd

Infatti all'inizio la Receive Window può per esempio essere di 300, che va dal byte 1100° al 1400°. Questo valore di 300 viene passato al trasmettitore tramite la **WindowSize**. Poi lui magari se la salva in una variabile locale.

Se il buffer di ricezione si riempie di 100 byte allora lo spazio si riduce e la receive window è di 200. Questo valore, che cambia dinamicamente, viene scritto dentro il **WindowSize** dei segmenti che invia

LastByteSent-LastByteAcked <= rwnd

Quali sono i problemi nell'uso del buffer di ricezione?

Il problema viene detto **silly window syndrome**, ossia sindrome della finestra stupida. Intanto è da capire che non è quando il buffer di ricezione viene svuotato che il trasmettitore viene avvertito, ma quando viene svuotato e il ricevitore manda l'ack. Ora, la sindrome della finestra stupida si verifica quando il ricevitore è molto lento a svuotare il buffer di ricezione. Questo cosa comporta? Supponiamo di avere un buffer di ricezione pieno.

- 1) Il ricevitore svuota 22byte dal buffer, li elabora, manda l'ack. Spazio disponibile= 22byte
- 2) Il trasmettitore può mandare solo segmenti da 22 byte, di norma un header per qualsiasi segmento consta di 20 byte quindi manderebbe solo 2byte di payload. Lo manda. Adesso il buffer di ricezione è di nuovo pieno.
- 3) Il ricevitore svuota 22byte dal buffer, li elabora, manda l'ack. Spazio disponibile= 22byte
- 4) Il trasmettitore può mandare solo segmenti da 22 byte, di norma un header per qualsiasi segmento consta di 20 byte quindi manderebbe solo 2byte di payload. Lo manda. Adesso il buffer di ricezione è di nuovo pieno.

E così via...

Come possiamo notare ogni volta, a causa del header molto grande non possiamo inviare grandi payload. Non sarebbe meglio forse aspettare che si liberi un po per inviare più payload in una sola volta invece di dedicare il segmento quasi interamente all'header?

L'idea è che quindi il ricevitore, svuotando il buffer, non avverte mai il trasmettitore che ci sono byte disponibili in ingresso fino a quando non si è svuotato per metà oppure per una porzione almeno pari a MSS.

Quali sono i problemi nell'uso del buffer del trasmettitore?

Abbiamo detto che il TCP legge i dati nel buffer di trasmissione e ogni tanto crea i segmenti che invia dall'altra parte. Se l'applicazione crea nel buffer pochi dati e lentamente, allora il TCP può essere costretta a creare segmenti con grande header ma con pochi payload. La soluzione è mandare comunque il segmento piccolo se il ricevitore è abbastanza veloce a svuotare il buffer, oppure aspettare che il buffer si riempia almeno per una dimensione del segmento pari a MSS. Questo problema è nota come **silly window syndrome** anche qui.

CONTROLLO DELLE PERDITE E DELL'ERRORE E TRASPORTO AFFIDABILE

Il TCP adotta il meccanismo del **Go-Back-N** ma leggermente modificato. Una prima differenza è che interpreta ogni errore come dovuto a overflow di una delle code di router attraversati a causa della congestione, e inoltre il time-out assume valori dinamici come descritto successivamente.

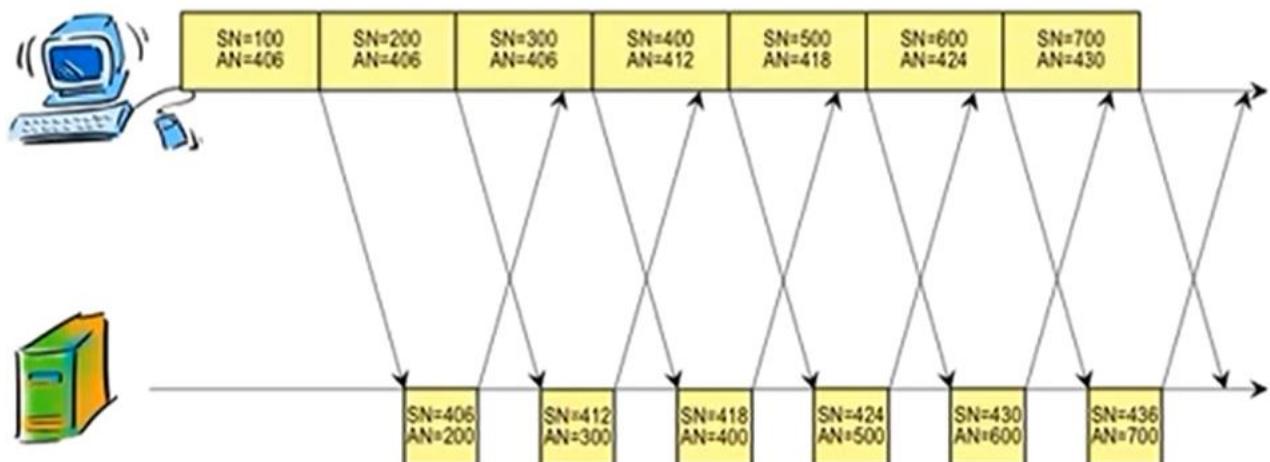
Col meccanismo del GBN ho quindi una finestra di trasmissione, trasmetto tutti i segmenti/byte di quella finestra e mi aspetto di ricevere un riscontro entro il timeout dell'ultimo segmento che non ha ricevuto l'ack che se scade torno a trasmettere da quello. La dimensione della finestra (N) non avrà un valore statico ma dipenderà dal controllo di flusso e di congestione.

Vediamo come il TCP garantirà un **trasporto affidabile** che include quindi:

- 1) Un buffer di ricezione che non presenterà mai duplicati
- 2) Un buffer di ricezione che non avrà buchi (segmenti mancanti)
- 3) Un buffer di ricezione che conterrà i dati nell'ordine così come sono stati inviati.

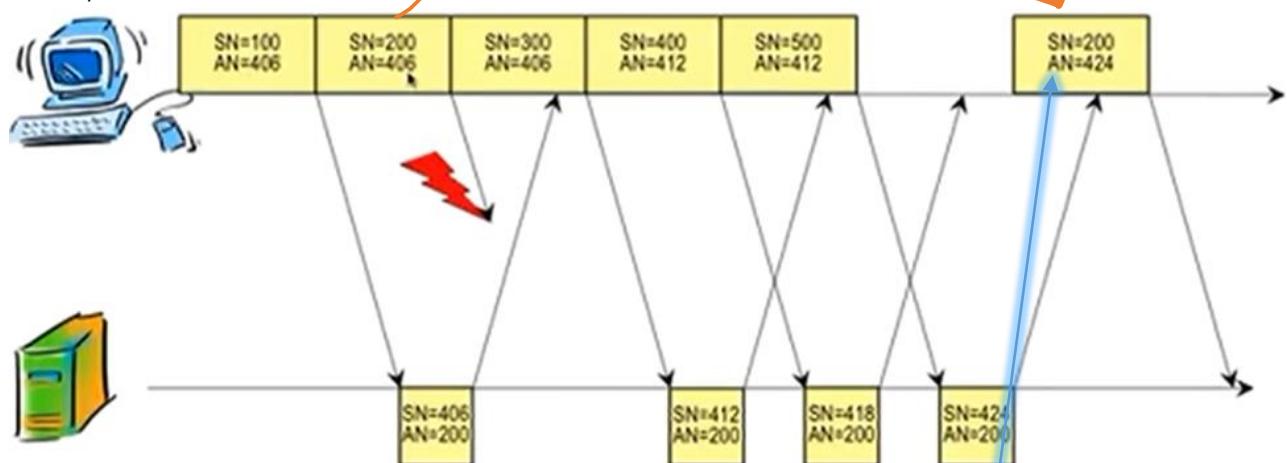
Esempio senza errori

Si suppone che a inizio connessione i due si siano detti che il computer invia segmenti grandi 100byte, mentre il server di 6 byte e sempre in fase di connessione hanno stabilito l'inizio del sequence number dei loro pacchetti. La SlidingWindow supponiamo sia di 4. Come sappiamo il server, data la connessione full-duplex invia oltre alla notifica anche se vuole un payload.



Il computer invia i segmenti 100,200 senza ricevere alcun pacchetto 406. Poi il server invia un suo segmento (**che sia o meno col payload qui non importa**) con SN=406 e AN=200 a confermare che ha ricevuto correttamente fino al segmento con SN=200 (e che si aspetta adesso il 200). E così via come per il GBN.

Esempio con errori



La finestra è [1,4]

Il computer manda SN=100 e si aspetta ancora il pacchetto 406, attivo timer

 manda SN=200 e si aspetta ancora il pacchetto 406

 manda SN=300 e si aspetta ancora il pacchetto 406

Arriva l'ack del server tramite il suo segmento. Il timer si ferma. La finestra si sposta [2,5]

Il computer manda SN=400 e si aspetta ancora il pacchetto 412, attivo timer

 manda SN=500 e si aspetta ancora il pacchetto 412

 si ferma perché ha raggiunto il limite della finestra.

Adesso è bloccato e fino a quando non riceve un ack/segmento da parte del ricevitore che abbia AN=300 (in modo tale da confermargli di aver ricevuto 200) non procede. Siccome gli arrivano SN=412 e SN=418 tutti ancora con AN=200 allora scade il timer e quindi si riprende a reinviare tutti i segmenti da 200 a 500 ancora una volta.

Il trasporto affidabile circa la (2) è dimostrato. Il segmento non viene perso quindi non si avranno buchi.

Come si comporta il mittente e il ricevente a seconda degli eventi si può riassumere da questo pseudo-codice:

```

NextSeqNum = InitialSeqNumber
SendBase = InitialSeqNumber

loop (per sempre) {
    switch (evento)

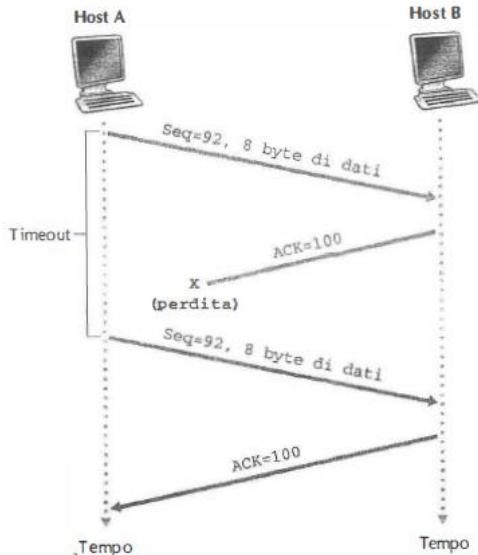
        evento: dati ricevuti dall'applicazione a livello superiore
        crea il segmento TCP con numero di sequenza NextSeqNum
        if (il timer attualmente non è in funzione)
            avvia il timer
        passa il segmento a IP
        NextSeqNum = NextSeqNum + lunghezza(dati)
        break;

        evento: timeout del timer
        ritrasmetti il segmento che non ha ricevuto ACK con
        il più piccolo numero di sequenza
        avvia il timer
        break;

        evento: ACK ricevuto, con valore del campo ACK pari a y
        if (y > SendBase) {
            SendBase = y
            if (esistono attualmente segmenti senza ACK)
                avvia il timer
        }
        break;
    }

} /* fine del loop */

```



Dove **NextSeqNum** è il prossimo SN da inviare e **SendBase** è quello del più vecchio segmento inviato che non ha ancora ricevuto nessun ack.

Differenza col GBN: nel GBN scaduto il timer si reinviava da quello che non ha ricevuto l'ack fino all'ultimo inviato; nel TCP si reinvia invece solo quello il cui timeout è associato sperando che i prossimi ack che tardano ad arrivare confermino la ricezione dei pacchetti già inviati.

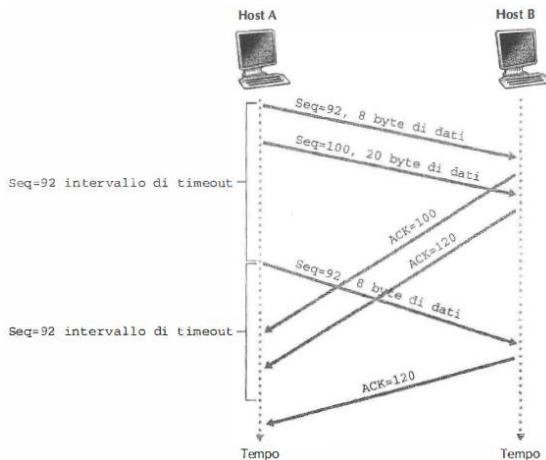
A invia un segmento di 8 byte di dati il cui SN=92, e attiva il timer/timeout.

B lo riceve e manda l'ack con AN=100 per dire che il 92 l'ha memorizzato e si aspetta il 100 (perché il segmento ha 8byte di dati quindi $92+8=100$), ma questo viene perso.

Intanto il timer va avanti fino a quando termina.

Appena termina si riprova a inviare il segmento 92 e il ricevente B si ritrova di nuovo lo stesso pacchetto, quindi invia l'ack ma non lo rimemorizza (**nessuna duplicazione**).

Uguale al GBN qui.



Qui invia due segmenti.

Il ricevitore invia i due ack e **memorizza i due pacchetti**.

Il timer scade. Per l'arancione si rimanda il più vecchio segmento senza ack.

(questa è una grande differenza col GBN).

Ques'ultimo avrebbe mandato non solo 92 ma anche 100. Mentre il TCP si limita solo a inviare il più vecchio evitando di inviare i successivi già inviati perché spera che da lì a presto arriveranno gli ack dei successivi pacchetti)

Gli ack di prima sono ricevuti solo adesso. Per il

verde l'if è verificato, quindi sendBase=100, si attiva per lui il timer e si **riaspetta l'ack**. Solo che prima della conclusione del timer arriva anche l'ack 120. Un'altra **differenza col GBN è che il TCP in questo caso non si è rinviato di nuovo il pacchetto 100**. L'if dell'arancione non è rispettato, non si attiva il timer ma sendBase=120.

Arriva un ack 120 dopo il timer, ma sendBase=120 quindi l'if dell'arancione non fa partire nessuna azione.

VARIANTI DEL TCP: RADDOPPIO DEL TIMEOUT

Quando scade il timer il TCP invia solo il più vecchio pacchetto che non ha ancora ricevuto l'ack. Il GBN invece inviava anche i successivi se erano stati già inviati. Questo perché il TCP intuisce che ci può essere stato un problema di rallentamento nel buffer o un problema di congestione della rete, quindi confida nel fatto che il timer scade per due motivi essenzialmente:

- I pacchetti sono stati ricevuti, ma prelevati dal buffer di ricezione lentamente e quindi lentamente arrivano gli ack -----> la soluzione è aspettare, è inutile rinviiare tutti i pacchetti.
- I pacchetti non sono stati ricevuti perché la coda è piena zeppa di pacchetti e alcuni probabilmente non potranno neppure entrarci (overflow) -----> la soluzione continua ad essere la stessa: aspettare, è inutile inviare pacchetti che probabilmente andranno persi.

La soluzione che si adotta è di raddoppiare il timeout ogni volta che il timer scade. Se prima era di 1 secondo, al suo scadere si riavvia un timer di 2 secondi. Se scade anche questo, al suo riavvio sarà di 4 secondi, e così via...

L'unica volta che il timer viene riavviato prendendo in considerazione EstimateRTT e DevRTT (vedi il paragrafo relativo, si trova dopo) è quando avviene l'evento blu o verde.

VARIANTI DEL TCP: ACK DUPLICATI (FAST RETRANSMIT)

Il raddoppio del timer aggiunta una cosa e ne distrugge un'altra. Supponiamo che in tutto questo raddoppio del timeout si è arrivati ad avere un timeout di 10 secondi. Questo timer è troppo lungo. Quindi se invio un pacchetto attivando un timer di 10 secondi, **se viene ripreso, lo rinvierò dopo 10 secondi**. Ci vuole una soluzione.

Supponiamo che il mittente invii in sequenza i pacchetti 92,100,120,135 e 141. Quando il

ricevitore riceverà 92 e poi 120 (che **non memorizza** ricordiamo), poiché si aspettava 100 capisce che il mittente lo ha mandato ma si è perso, quindi c'è stato un *buco*. L'unica alternativa è aspettare che il timer finisca così che, non ricevendo l'ack per il 100 capisce che lo ha perso. Ma il timer può come abbiamo detto essere troppo grande, come faccio ad avvisarlo ma soprattutto a fargli capire che non è dovuto al buffer di ricezione ed evitargli di far scadere il timer che causerebbe un attesa ancora superiore? L'unico modo che conosciamo sarebbe inviargli in mezzo al timer un nack, ma il tcp non lo prevede.

Si ricorre a quelli che vengono detti **ack duplicati**, che in realtà sono 3 ack in sequenza col numero di pacchetto mancante (100). Il numero 2 sarà

chiaro affrontando il *problema della congestione* che affronteremo a breve.

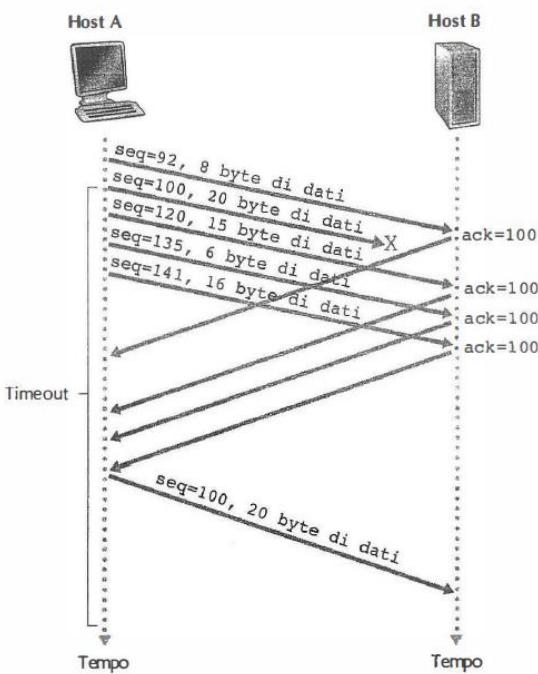
Modifichiamo lo pseudo-codice così da includere questa casistica.

```
evento: ACK ricevuto, con valore del campo ACK pari a y
    if (y > SendBase) {
        SendBase = y
        if (esistono attualmente segmenti che non
            hanno ricevuto un ACK)
            avvia il timer
    }
    else { /* un ACK duplicato per un segmento */
        incrementa il numero di ACK duplicati ricevuti
        per y
        if (numero di ACK duplicati ricevuti per y == 3) {
            /* ritrasmissione rapida */
            rispedisci il segmento con numero di sequenza y
        }
    }
    break;
```



Il terzo evento rimane uguale ma contempla anche il nostro caso. Nell'esempio sopra il mittente riceve il primo ack a conferma del pacchetto 92. Si avrà quindi SendBase=100. Adesso il mittente si aspetta un ack con y=120 a conferma della ricezione di 100. Ma riceve per 3 volte (si conteranno) un ack con y=100

(che quindi rientra nell'else). In questo caso si rispedisce il pacchetto perso.



COMPORTAMENTO RICEVITORE: SINTESI

Sintetizziamo il comportamento del ricevitore.

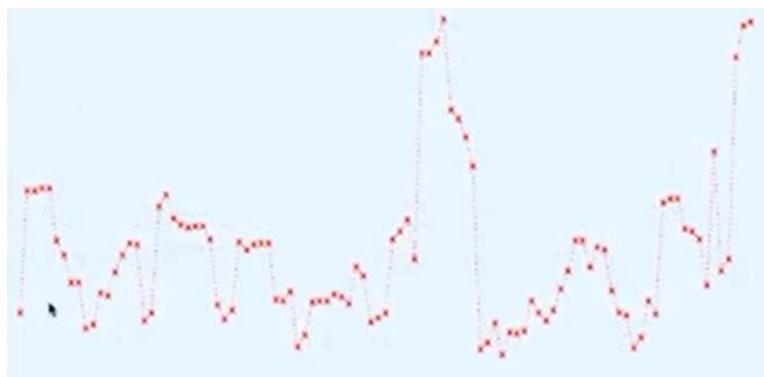
- Se gli arriva un segmento nel corretto ordine, non manda subito l'ack, ma aspetta 500ms per ricevere anche il successivo segmento così da inviare un ack cumulativo e unico.
- Se arriva un segmento con numero di sequenza superiore a quello atteso. C'è stato un buco quindi invia un ack duplicato per il byte che invece attendeva.
- Alcune versioni del tcp memorizzano i pacchetti fuori sequenza, in generale qui non è così. Il mittente può inviare quanti pacchetti vuole, se però per qualche motivo arrivano fuori sequenza (per esempio qualcuno si perde) il ricevitore memorizzerà solo fino ai pacchetti in sequenza che si aspetta.

STIMA DEL RTT (ANDATA E RITORNO) PER LA GESTIONE DEL TIME-OUT

Il TCP deve stabilire il valore del timeout per ogni segmento che invia (se il timeout non è già in uso da qualcuno), scaduto il quale deve reinviare i pacchetti. Questo timeout deve essere ovviamente maggiore di RTT ma per vari motivi (come la congestione della rete) questo RTT non è statico ma varia nel tempo facendo sì che anche il time-out non possa essere statico. C'è quindi bisogno di valutare dinamicamente, in base alla storia passata, ma dando comunque maggiore peso a quella recente, il valore medio di RTT.

Il TCP non associa ad ogni segmento inviato un timer (le risorse sono limitate), ma quando invia un segmento, se il timer non è associato ad alcun segmento precedentemente inviato di cui ancora si aspetta l'acknowledgement, allora lo si imposta per lui. Questo timer si avvia quando il segmento viene inviato e si stoppa quando riceve l'ack. In questo modo si è misurato l'RTT di quel segmento (e adesso il timer è libero per misurare l'RTT di un altro segmento).

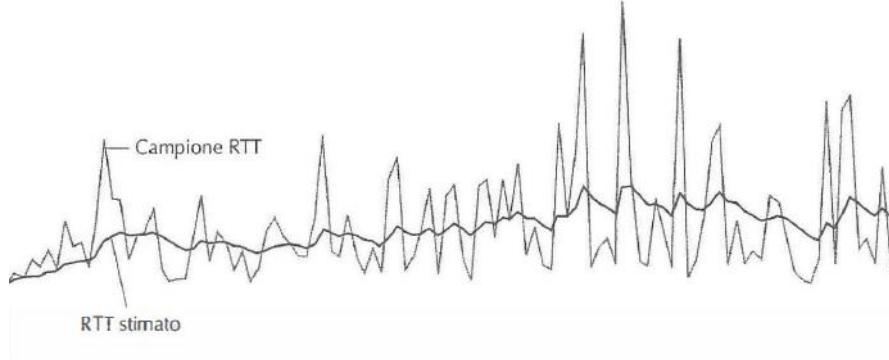
Il TCP salva per ogni segmento (non per quelli ritrasmessi) a cui è possibile associare il timer in memoria i valori di RTT ogni volta che un segmento riceve il rispettivo acknowledgement. Quindi si avranno una serie di $\{RTT^i\}$ che quindi rappresentano vari campioni misurati.



Usa questi valori applicandogli un filtro passa basso (ossia uno strumento matematico che taglia via i picchi di una curva molto variabile) che nella pratica equivale a calcolare l'RTT stimato:

$$\text{EstimateRTT}^i = (1 - a)\text{EstimateRTT}^{i-1} + aRTT^i$$

A seconda dei valori di α si darà più peso alla storia passata o a quella recente. Di solito si preferisce quest'ultima e quindi $\alpha = \frac{1}{8}$ e tale media in statistica si chiama **media mobile esponenziale ponderata**.



Un altro parametro statisticamente usato è il **DevRTT** che indica la **stima della deviazione o varianza**:

$$\text{DevRTT}^i = (1 - b)\text{DevRTT}^{i-1} + b|\text{RTT}^i - \text{EstimateRTT}^i|$$

con $b = \frac{1}{4}$. La $|\text{RTT}^i - \text{EstimateRTT}^i|$ misura quanto sono disperso rispetto alla media stimata, mentre DevRTT^i è quella filtrata che da mediamente una stima di quanto generalmente, in relazione ai valori passati e presenti, ci sono state queste fluttuazioni.

Qual è il valore ottimo del time-out? Adesso che abbiamo quei due parametri statistici possiamo calcolare per ogni volta che si riceve un acknowledgement il valore di time-out come:

$$\text{Time-Out} = \text{EstimateRTT} + 4\text{DevRTT}$$

CONTROLLO DELLA CONGESTIONE

Questo è un altro servizio che offre il TCP. Il controllo di flusso, lo abbiamo visto, evita una sorta di congestione del buffer del ricevitore, ma non quello della *rete*. Il TCP, essendo un protocollo end-to-end, coinvolge fisicamente anche i dispositivi di *router*. Anche questi ultimi avranno un loro buffer e di conseguenza devo preoccuparmi che anche lì non si abbiano eventi di congestione.

Ricordiamo però che i router sono dispositivi di livello 3 (ossia parlano fino al livello 3 della pila protocollare), quindi non posso usare un approccio esplicito come fatto nella controllo di flusso, e inoltre la rete generalmente non ha l'*intelligenza* di notificare eventi di congestione. Quindi l'approccio di internet non fa altro che demandare tale controllo al tcp. Questo controllo non verrà fatto dal tcp inserendo negli header delle informazioni in più.



Piuttosto ci doteremo *lato trasmettitore* (e non ricevitore) di una **congestion windows (CWND)** che varierà dinamicamente in base agli eventi che osserva come ricezione di ack, timeout etc... Ovviamente il trasmettitore dovrà trasmettere i segmenti, secondo quanto visto col controllo di flusso, in base a una sola delle due finestre (non può mica dare conto a entrambe). La regola è dunque che ogni volta si sceglie come finestra quella che ha **valore minimo** tra RCVWND (buffer di ricezione) e CWND.

In ogni caso, tutte le volte che scade un timeout senza che abbia ricevuto un riscontro, la tcp interpreta la perdita del pacchetto come dovuta alla congestione della rete, e poiché l'interpretazione è sempre questa allora ogni volta che avviene una perdita **riduce** la finestra CWND.

Quali sono gli algoritmi usati per valutare la congestion window?

Esistono due algoritmi diversi, usati in maniera dinamica, che vengono usati uno o l'altro in base a uno delle due possibili fasi in cui può trovarsi la connessione tcp.

Le due **fasi** della connessione TCP sono:

- Slow Start
- Congestion Avoidance

A seconda della fase in cui si trova applica uno stimatore o meccanismo di update della CWND diverso.

Come faccio a dire in quale fase mi trovo?

Esiste una variabile **SSTHRESH** ("slow start threshold", ossia "soglia di slow start") lato trasmettitore che viene usata per determinare la fase attuale in cui si trova la CWND:

- Se CWND < SSTHRESH si è in Slow Start
- Se CWND > SSTHRESH si è in Congestion Avoidance

Fondamentalmente questa variabile segna il confine tra "sono in buone condizioni" (Slow Start) e "mi sto avvicinando alla congestione" (Congestion Avoidance).

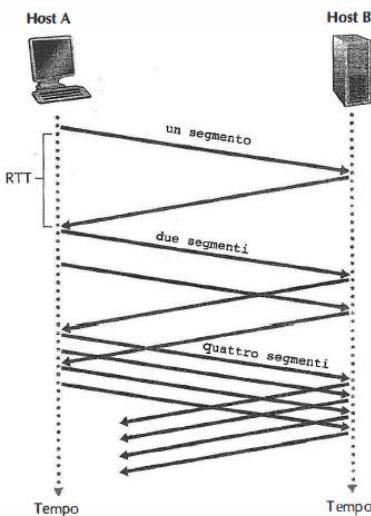
Slow start

E' la fase in cui si trovano tutte le connessioni window quando vengono create.

CWND è settato alla dimensione di un segmento MSS e SSTHRESH di solito ha un valore molto più elevato.

Regola: la CWND viene incrementata di un segmento MSS alla volta per ogni ACK ricevuto perché implicitamente la ricezione dell'ack indica che la rete funziona bene per adesso.

Esempio:



All'inizio il buffer di ricezione avrà dimensione maggiori di CWND (che è a 1). Quindi si sceglie come finestra su cui regolarsi CWND.

Pertanto posso inviare al massimo 1 segmento. Appena ne ricevo l'ack incremento CWND di 1. Adesso CWND è a 2.

Posso inviare al massimo 2 segmenti. Ricevo i due ack. CWND passa a 4.

Posso inviare al massimo 4 segmenti. Ricevo i quattro ack. CWND passa a 8.

E così via...

Vado avanti fino a quando:

- 1) Si ha il primo evento di congestione (fondamentalmente ogni volta si perde un pacchetto abbiamo detto oppure
- 2) CWND < STHRESH cioè sto ancora in StartSlow oppure
- 3) CWND < RCWND per quanto detto prima sul valore minimo da prendere come finestra.

Ovviamente se aumento la CWND significa che posso inviare più segmenti e quindi aumenta il rate che è calcolabile come:

$$\text{Rate (o banda)} = \frac{\text{CWND}}{\text{RTT}} = \frac{\text{quantità di informazioni inviate}}{\text{tempo di andata e ritorno di ciascuna}}$$

Quindi la velocità di trasmissione cresce **esponenzialmente**.

Quando avviene la 1) significa che devo ridurre la velocità, e in particolare porrò STHRESH = CWND/2 (con CWND il valore che aveva un attimo prima della congestione) e di nuovo CWND=1MSS.

Congestion Avoid

Se passo a questa fase allora sono *meno aggressivo* nell'aumentare CWND (e di conseguenza anche il rate).

Regola: la CWND viene incrementata di $\frac{1}{CWND}$ alla volta per ogni ACK ricevuto.

Quindi il rate cresce **linearmente**. Infatti se CWND=10MSS allora si possono inviare 10 segmenti. Al primo ack ricevuto succede che CWND=CWND+1/CWND=10+1/10=10,1; poi al secondo ack ricevuto CWND=10,2 ; al terzo ack ricevuto CWND=10,3 e così via. Ci vorranno in totale la ricezione di 10 ack perché CWND si possa incrementare di uno, il che equivale a dire in generale che CWND aumenta di 1MSS ogni CWND ack mandati e ricevuti.

L'incremento lineare termina per lo stesso motivo dello slow start, ossia quando si perde un pacchetto. In questo caso si pone STHRESH=CWND/2 e CWND=1 e si ritorna in slow start.

Ricordiamo sempre che si controllerà il **minimo** tra la CWND e la RCWND in entrambe le fasi.

Fast Recovery & Fast Retransmit: una risposta al perché della logica degli ack duplicati

Abbiamo detto che il protocollo tcp interpreta ogni perdita come dovuta alla congestione. Ha questo pregiudizio verso la rete. Questo significa che se scade il timeout e non si riceve l'ack per il pacchetto mandato, allora la colpa è della rete e *punisce* la CWND riducendola (e quindi riducendo il rate). Gli ack duplici servono proprio a evitare questo scopo, ossia quando il ricevitore vede un *buco* nei pacchetti ricevuti, *poiché sono arrivati i successivi segmenti a quello atteso* significa che non c'è una vera e propria congestione della rete e quindi, oltre che evitargli una rigenerazione del timer ancora più grande, vuole dire al trasmettitore prima del termine del timeout "guarda che non è stata colpa della congestione della rete, questa non ha problemi a sostenere il tuo ritmo di invio, riprova a inviarmelo senza che raddoppi il timer.".

Fino adesso se si era in Slow Start o in Congestion Windows, all'arrivo dei 3 ack si faceva quanto diceva la vecchia versione del tcp chiamata **TCP Tahoe**: non esistono ack duplicati e quando il timer scade si dimezza $SSTHRESH = CWND/2$ e si pone $CWND = 1MSS$, si reinvia il segmento perso e ci si riporta sempre allo Slow Start. Quando si perde al massimo un pacchetto, il TCP Tahoe è generalmente un ottimo modo di procedere, ma se i pacchetti persi sono più di uno questa versione potrebbe mantenere a 1 CWND così a lungo da rallentare fortemente la rete.

Una nuova versione, chiamata **TCP Reno**, prevede invece una *nuova fase* chiamata **Fast Recovery** nella quale si entra da entrambe le fasi che conosciamo impostando:

- $SSTHRESH = CWND/2$
- $CWND = SSTHRESH + 3MSS$
- Reinviando il segmento perso

Nella fase di Fast Recovery si rimane ogni volta che si ricevono nuovamente ack duplicati e ogni volta si imposta:

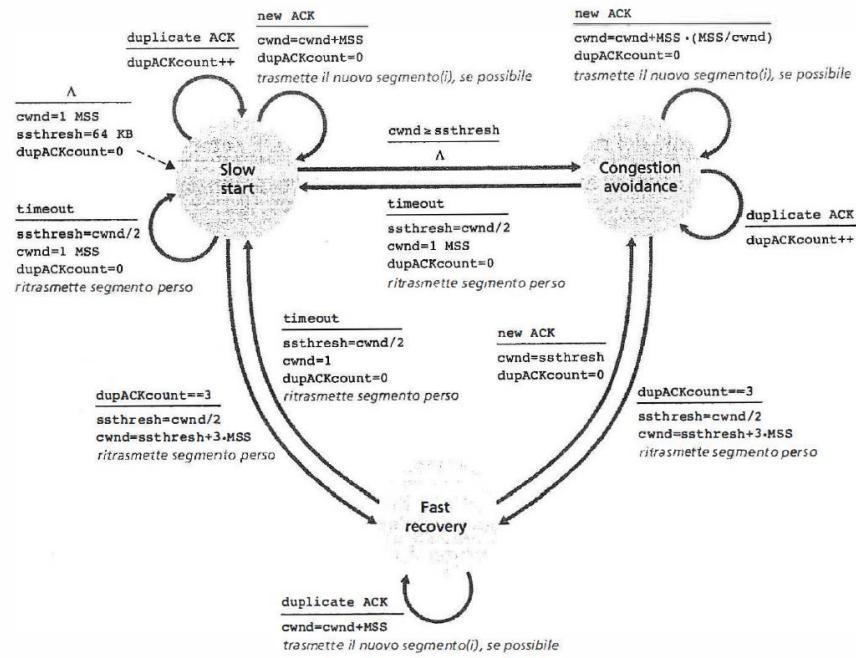
- $CWND = CWND + 1MSS$ (lineare)

Questa versione, a differenza della prima che riparte sempre dallo Slow Start imputando quindi la colpa sempre alla congestione della rete, da un forte peso (quindi suppone la rete davvero congestionata) qualora il timeout scada senza ricevere un semplice ack, e in questo caso riparte lentamente come prima ossia dallo Slow Start ponendo:

- $SSTHRESH = CWND/2$
- $CWND = 1$
- Ritrasmette il pacchetto perso (quando entra nella fast recovery ritrasmette il pacchetto e quindi attiva il timer. Se, sempre nella fast recovery scade il timer, allora davvero la rete è congestionata e quindi ritrasmette il pacchetto per una terza volta e riprende dallo Slow Start).

Mentre se arriva un semplice ack allora significa che il ricevitore non avverte di aumentare il rate e procede quindi linearmente passando a Congestion Avoidance per ponendo:

- $CWND = SSTHRESH$, ossia non invio $cwnd=1$ pacchetto ma continuo con quanto inviavo prima.

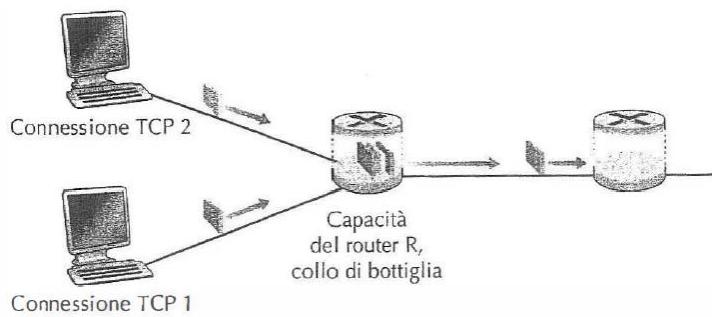


APPROCCI AL CONTROLLO DI CONGESTIONE DELLA RETE

Abbiamo già visto come questo protocollo affronta il problema della *congestione della rete*. Esistono fondamentalmente due *approcci* al controllo di congestione della rete:

- 1) **Controllo di congestione end-to-end:** il livello di rete demanda al livello di trasporto il controllo della congestione che quindi, come già visto col TCP basa il controllo solo sul comportamento che può vedere (perdita segmenti). Il TCP quindi adotta questo approccio perché adotta il protocollo di rete IP (che vedremo) che non fornisce alcun supporto.
- 2) **Controllo di congestione assistito dalla rete (Network-assisted):** i router hanno una piccola intelligenza: sanno comunicare col mittente/destinatario fornendogli un feedback sullo stato dei suoi buffer mandando a volte un particolare valore di bit, o un **choke packet** al mittente che dice esplicitamente "sono congestionato", oppure ancora il router modifica un certo campo di un pacchetto che è transitato da lui così che quando arriva al destinatario gli capire al destinatario che deve mandare al mittente una notifica di congestione (questo però, a differenza del primo necessita di un tempo di RTT). Un esempio sono le reti ATM col protocollo ABR.

FAIRNESS



Il concetto di **fairness** si applica *solo* a connessioni TCP. Per *fair* si intende *equità*, in questo caso equità nella distribuzione di banda (velocità trasmissiva) ad ogni periferica collegata. Se ci sono K connessioni TCP che, anche se hanno poi

percorsi diversi, condividono un link che ha capacità trasmissiva R, in ogni caso si tende a distribuire tale velocità in modo equo che quindi sarà R/K per ogni connessione.

Abbiamo visto che il **rate** per ogni connessione TCP dipende dalla CWND. Aumentandola o diminuendola si varia anche il rate. Questo perché il TCP offre un supporto alla *congestione della rete*. Protocolli come *UDP* che invece non controllano neppure il flusso o gli errori non hanno nessuna CWND a cui riferirsi e quindi il rate è costante. Se ne infischiano di chi c'è oltre a loro, spedendo i dati sempre allo stesso rate costante.

Nota bene: UDP non regola il proprio rate, ossia il mittente spedisce nel buffer del router sempre con lo stesso rate. Se non c'è overflow fortuna per lui, ha immesso un pacchetto che invece altri come il TCP non sono riusciti a immettere perché magari, riducendo il proprio rate, sono stati più lenti; se invece è sfortunato, il pacchetto viene scartato. Nota come a UDP interessa solo infilare i propri pacchetti dentro il buffer prima degli altri. Se insieme a lui ci sono connessioni TCP, queste vengono **soffocate** da chi è connesso con UDP perché abbiamo da un lato (UDP) una connessione che non cambia il rate in relazione alla congestione della rete, e una (TCP) che invece lo fa. Pertanto la TCP, a causa della *prepotenza* dell'UDP si vedrà diminuire CWND e quindi il rate.

Protocollo ATM ABR

E' un protocollo ATM a *bit rate avviabile*. Vuol dire che la rete stessa riesce a dare info al mittente e al destinatario sulla sua condizione di congestione. In particolare sfrutta switch (sarebbero i router nella terminologia ATM) che sono capaci di modificare i pacchetti in transito (chiamati **celle**) e di inviare anche loro particolari celle chiamate **celle AR**. Inoltre ATM fa uso di **circuiti virtuali**, ossia si basa sempre sull'internet (commutazione di pacchetto) ma ciascun commutatore può ricordare chi vi passa e dunque tenere traccia/controllo delle singole sorgenti che vi trasmettono.

L'idea è che ogni tanto ATM invia in mezzo all'invio delle celle dati, delle celle AR (una ogni 32 celle dati) che hanno 2 particolari bit (CI, NI) e un campo particolare (ER) che a seconda di come vengono modificati possono permettere alla sorgente e alla destinazione di comprendere come comportarsi. Quando uno switch è congestionato modifica il bit **EFCI** della cella dati diretta al destinatario. Quando questo riceve poi una cella RM e prima ha ricevuto una cella dati con bit EFCI a 1 deve modificare lui il bit CI della cella RM e inviarla al mittente per avvertirlo della congestione.

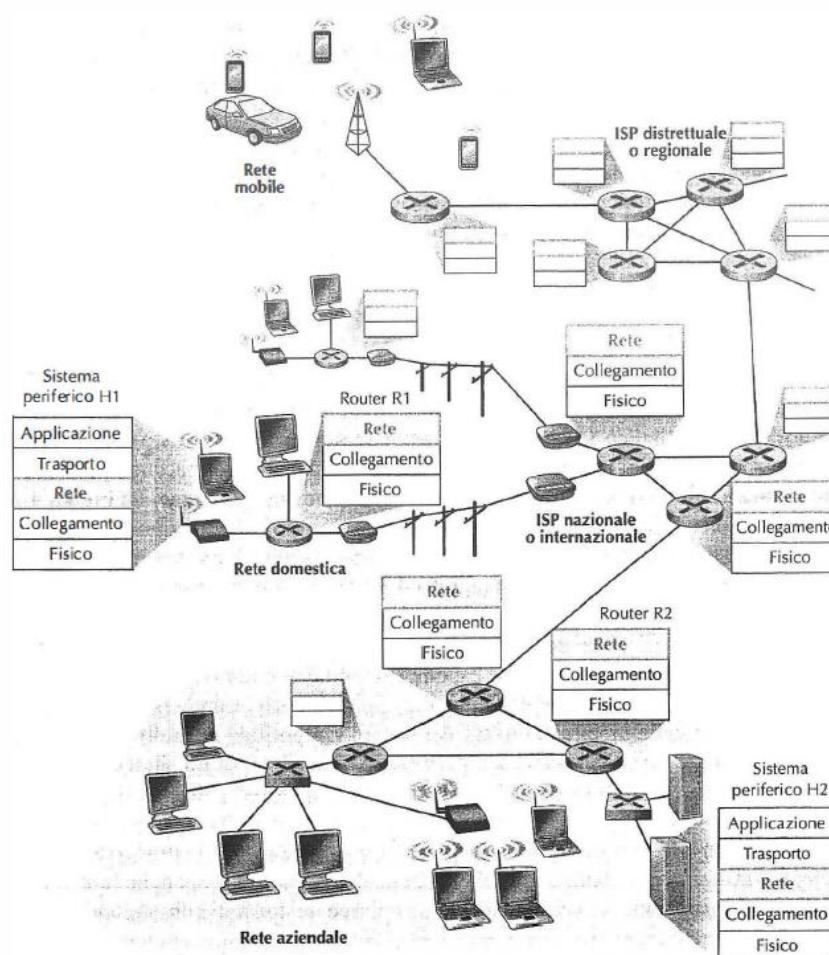
Un altro modo è che sia lo stesso switch a impostare il bit CI per indicare la congestione, mentre può lui stesso mettere a 1 il bit NI per indicare una moderata congestione. Invia tale cella al mittente che saprà della forte o moderata congestione.

Un ultimo campo delle celle RM è ER che consta di due byte che modificano tutti gli switch del percorso e che alla fine conterrà la velocità minima sopportabile tra tutti gli switch alla quale mittente e destinatario dovranno attenersi.



CAPITOLO 4

LIVELLO DI RETE



Abbiamo due host h1 e h2 e l'immensità di collegamenti e router con tutti i provider, i trasmettitori che da vita e possibilità di esistere all'intera rete.

Una classica comunicazione a livello di rete tra h1 e h2 è la seguente:

- 1) L'applicazione passa i *dati* al livello trasporto che passa *segmenti* al livello di rete che li incapsula in **datagrammi** che consegna al router R1.

- 2) Da R1 il datagramma passa attraverso tanti router fino ad arrivare al router R2.
- 3) R2 consegna al livello di rete di H2 il datagramma che lo spacchetta ottenendo un segmento da dare al livello di trasporto che spacchetta ottenendo i dati che consegna al livello applicativo.

I **servizi/funzioni** offerti dal livello di rete sono:

- **Inoltro (forwarding)**: il livello di rete definirà delle istruzioni tali che ogni router quando gli arriva un pacchetto li legge e capisce dove inoltrarlo (verso quale altro router).
- **Instradamento (routing)**: poiché esistono tanti router (interfacce di rete) e quindi tanti possibili percorsi, le istruzioni sopra dette verranno generate tramite algoritmi di instradamento.

A livello di trasporto avevamo che il TCP (così come altri protocolli) gestiva tutte le funzionalità possibili offerte dal livello di trasporto. Mentre qui *cooperano* più protocolli insieme; un protocollo si occupa per esempio dell'inoltro e un altro ancora del routing. Inoltre una parte dei protocolli di rete sono implementati anche dagli attori router.

Altri servizi aggiuntivi potrebbero essere (opzionali):

- Consegna garantita
- Consegna garantita con ritardo limitato
- Consegna ordinata
- Banda minima garantita
- Jitter limitato: ossia il tempo tra due invii è uguale a quello di ricezione
- Servizi di sicurezza

Tutti queste funzioni sono *opzionali*; cioè di base ogni protocollo di rete si impegna solo a fare del suo meglio per consegnare ogni pacchetto (**servizio best-effort**) ma non escludendo l'ipotesi che può perderlo.

Quello che studieremo noi non implementa nulla delle funzioni aggiuntive se non la sicurezza. Infatti il TCP/IP usa IP come protocollo di rete che demanda al TCP la consegna garantita (tramite controllo di errore e ritrasmessione), consegna ordinata etch...

ATM

Alcuni protocolli, come ATM, offrono una parte di questi servizi aggiuntivi oltre al best.effort. In realtà ATM è più un architettura di rete e esistono due modelli di servizio ATM che sono:

- 1) **CBR**: il servizio di rete ATM a bit rate costante dota il flusso di pacchetti (detti *celle*) di un canale virtuale con proprietà simili a quello di un collegamento trasmisivo a banda fissa tra mittente e ricevitore. Questi due si mettono inizialmente d'accordo sui valori massimi di ritardo end-to-end di una cella, variabilità di tale ritardo (*jitter*) e la percentuale di celle perdute o consegnate in ritardo. Il modello di servizio si impegna a far sì che i valori per quei campi siano inferiori a quelli stabili. In sintesi garantisce consegna, banda costante, ordinamento e temporizzazione.
- 2) **ABR**: il servizio di rete a bit rate disponibile non esclude la perdita di celle (e neppure la garanzia di consegna), ma garantisce l'ordinamento e un minimo tasso di banda. A differenza della CBR garantisce un servizio di congestione della rete, anzi più che un servizio, una *indicazione* al mittente tramite una notifica (di solito un semplice bit)

che la rete è congestionata e così al livello di trasporto si adatta di conseguenza a ridurre il rate.

RETI ORIENTATE ALLA CONNESSIONE E NON

Esistono approcci all'inoltro e all'istradamento diversi. Ricordate che nel livello di trasporto abbiamo parlato di orientamento alla connessione (TCP) e non orientato alla connessione? Anche il livello di rete offre questi due tipi di servizi; le reti di calcolatori che mettono a disposizione il servizio con connessione sono chiamate **reti a circuito virtuale (VC)** mentre quelle che offrono solo il servizio ma senza connessione sono dette **reti datagram**.

Internet è una rete datagram e quindi ragioneremo da ora in poi con reti datagram"

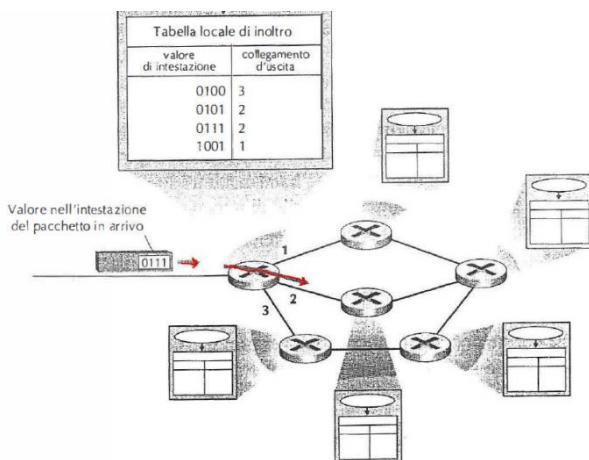
Questo vuol dire che la tabella di inoltro che vedremo sarà quella delle reti datagram e quindi ogni router userà un approccio datagram.

RETI A CIRCUITO VIRTUALE (VC)

Un **circuito virtuale** è virtuale perché in realtà non esiste fisicamente un tale circuito fatto in quel modo come invece lo è per la commutazione di circuito, piuttosto rappresenta una via di mezzo tra commutazione di circuito e commutazione di pacchetto in quanto sfrutta una rete a commutazione di pacchetto, pertanto impacchetta i flussi di dati piuttosto che inviarli in continuo, ma sfrutta una connessione o canale che sia statico e sempre quello come la commutazione di circuito permetteva. Questo permette di non impiegare inutilmente il canale nei tempi morti ma solo quando ne viene fatta richiesta. È una fitta rete fatta di router e tanti collegamenti/circuiti, però, vedremo, ogni router quando arriva un pacchetto prende una scelta decisa da noi alla creazione del circuito virtuale, allora si viene a creare come una sorta di circuito appunto virtuale.

Gli ingredienti sono:

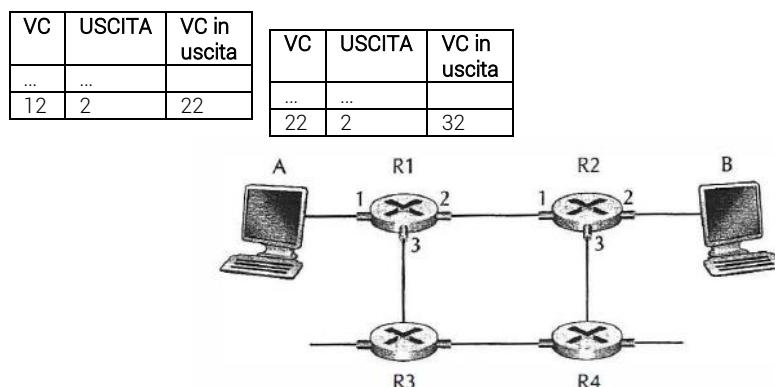
- Un percorso, ossia collegamenti tra router, che unisca l'host sorgente con la destinazione
- Numeri VC
- Tabella di inoltro in ciascun router
- Righe nella tabella di inoltro dedicate al circuito virtuale in esame



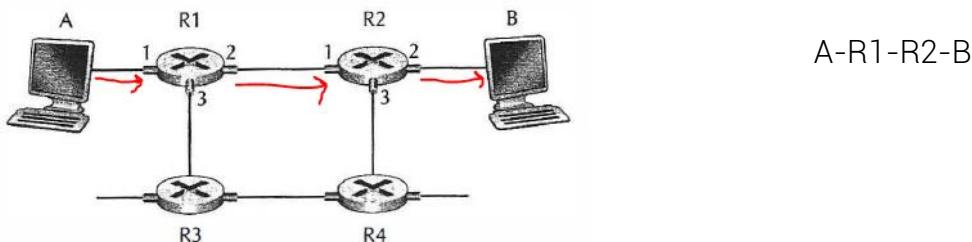
Ogni router possiede una **tabella di inoltro**. Il router quando riceve un pacchetto legge un certo campo nell'header che è l'identificativo del circuito virtuale a cui il pacchetto appartiene, ossia il **numero VC**. Va a vedere nella tabella quale collegamento in uscita si è riservato ai pacchetti di questo circuito virtuale VC che passano da lui e lo si immette in quel collegamento di uscita. Le tabelle di inoltro non cambiano, questo implica un circuito sempre uguale come con la commutazione di circuito.

Spiegerò il funzionamento delle reti a circuito virtuale con un esempio.

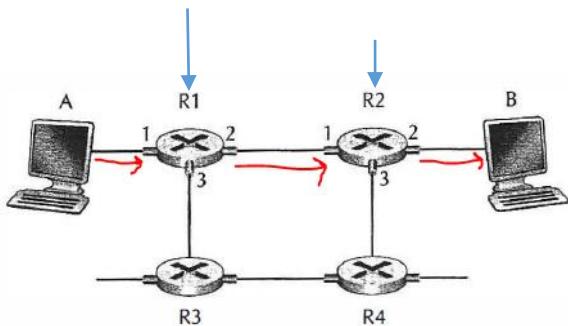
Abbiamo un host sorgente A di indirizzo ip IP_SORG e un host destinazione B con indirizzo ip IP_DEST.



- 1) **Istaurazione:** Il livello di trasporto contatta il livello di rete e gli specifica IP_DEST. Il livello di rete allora tramite *algoritmi di istradamento* stabilisce il circuito virtuale dalla sorgente:



Dopo di che per ogni router del percorso aggiunge nella tabella di inoltro una riga riservata a quel circuito (*servizio di inoltro*).



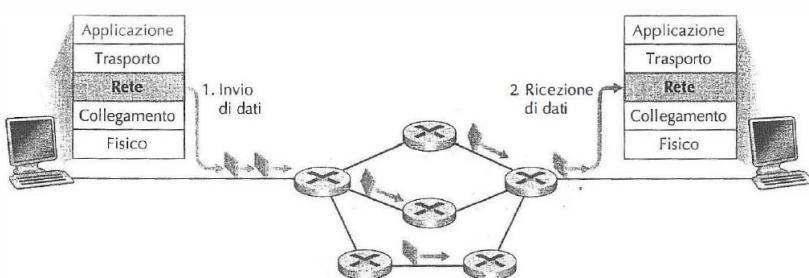
Così il pacchetto esce da A con VC=12. Entra nel router e quindi viene inoltrato all'uscita 2 con un nuovo VC=22. Entra nel secondo router e viene inoltrato all'uscita 2 con nuovo VC=32.

- 2) **Trasferimento dati:** creato il circuito i pacchetti possono essere inviati.
- 3) **Terminazione:** quando A o B vogliono terminare allora basta che uno informi l'altro e vengono eliminate le righe nei router del percorso relativi al circuito che si vuole cancellare.

RETI DATAGRAM

Un altro approccio di inoltro e instradamento è quello delle reti che forniscono un servizio senza connessione, ossia reti **datagram**. Nelle reti datagram, così come quelle VC, il pacchetto viene decorato con *l'indirizzo di destinazione* dell'host B da raggiungere. Non viene messa su nessuna connessione, quindi non si stabilisce alcun circuito virtuale, ergo i router non devono memorizzare lo stato della connessione.

Ogni router dispone di una **tavella di inoltro** anche qui dove però fa corrispondere un indirizzo IP a una delle interfacce di collegamento (ossia ai piedini di uscita). Quindi quando un pacchetto vi arriva (con l'indirizzo IP di destinazione), il router guarda nella tabella di inoltro dove farlo inoltrare. I router si aggiornano dinamicamente tramite algoritmi interni di routing. Ques'ultima cosa porta a un problema: se un pacchetto k segue un certo percorso, è dovuto al fatto che n router hanno nella propria tabella di inoltro una uscita ben precisa

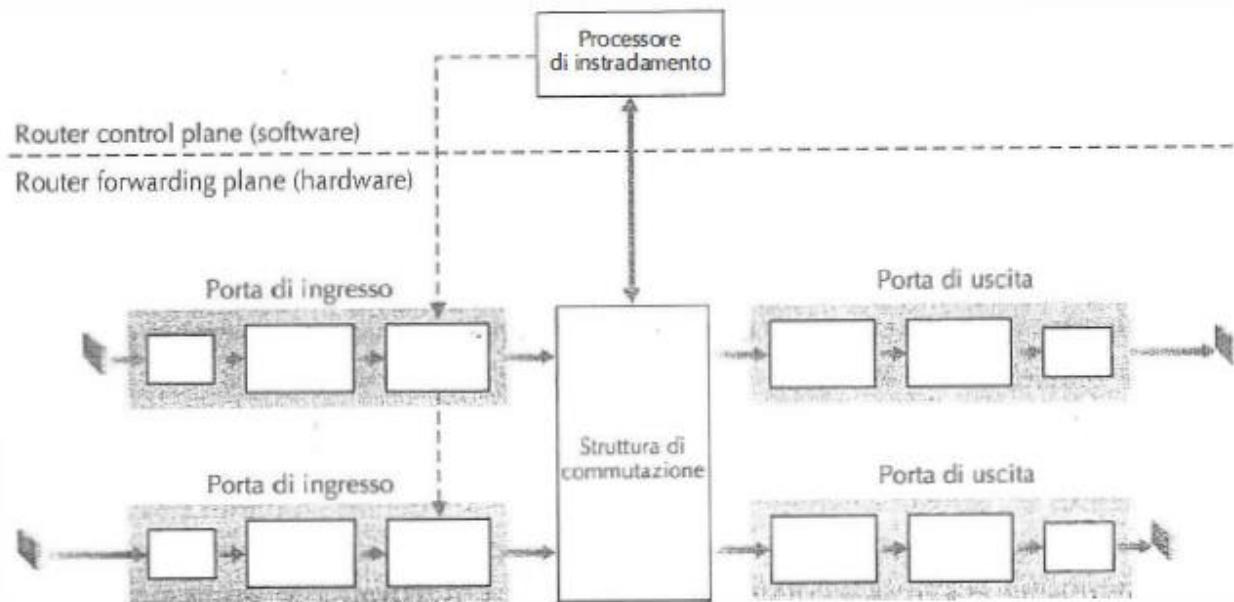


per l'indirizzo di destinazione che il pacchetto porta con se. Se all'arrivo del pacchetto k+1 succede che le tabelle di inoltro si aggiornano (il che avviene ogni 1-5 minuti) questo pacchetto, che nella successione di

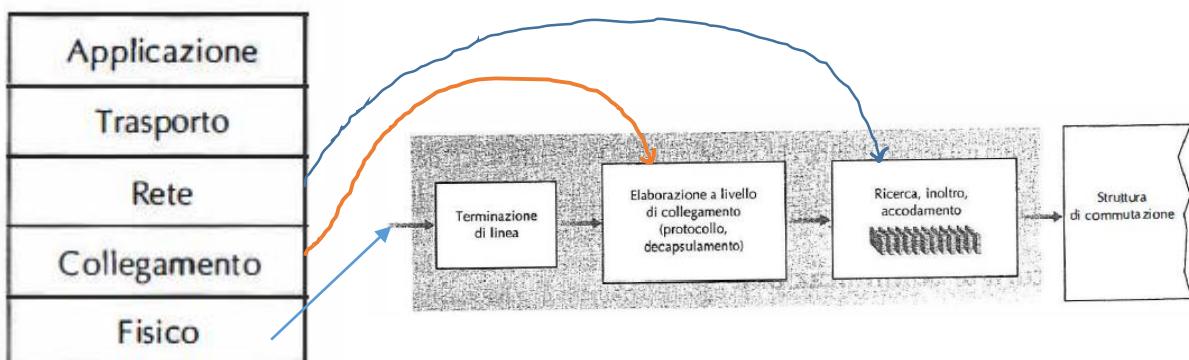
invio aveva ordine successivo a quello k, prendendo una strada diversa potrebbe anche arrivare prima di quello k°. Quindi le reti datagram non garantiscono ordine di ricezione, piuttosto una consegna **fuori sequenza**, questo perché a differenza dei circuiti virtuali non esiste un circuito stabile durante tutta la connessione ma cambia constantemente a seconda di certi avvenimenti.

COSA SI TROVA DENTRO UN ROUTER?

Un router è un piccolo computer costituito da porte di I/O, memoria e un processore di istradamento, che altro non è che una CPU. La struttura interna di un router è la seguente:



Come sappiamo, un router implementa fino a 3 livelli della scala protocollare. Infatti ogni porta di ingresso è fatta così:



a. Pila di protocolli a cinque livelli di Internet

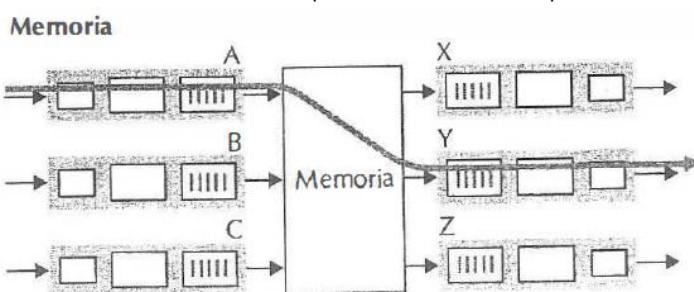
In modo speculare anche la porta di uscita.

Le **porte di ingresso** svolgono 3 funzioni: terminano il collegamento fisico in ingresso, elaborano al livello di collegamento il pacchetto ricevuto decapsulandolo e consegnando il datagramma alla terza funzione che implementa il livello di rete. Qui si consulta la *tavella di*

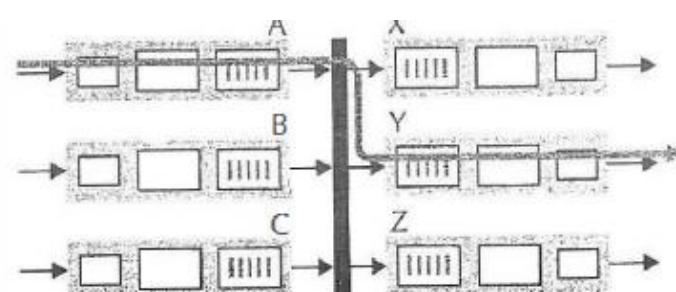
inoltro in due modi: il processore legge l'indirizzo di destinazione portato dal pacchetto e determina su quale porta di uscita deve essere inoltrato; oppure una copia della tabella di inoltro può essere installata direttamente sulla porta di ingresso e aggiornata dal processore. In ogni caso si andrà a consultare una tabella di inoltro nella quale la ricerca sarà fatta di hardware per questione di prestazioni e tramite la struttura di commutazione raggiungerà la porta di uscita corretta. Ancora in questo terzo blocco vengono memorizzati in un *buffer* tutti i pacchetti in attesa di essere consegnati alla **struttura di commutazione**. Sarà grazie a quest'ultima che ogni pacchetto potrà raggiungere la porta di output decisa dalla tabella di inoltro. Su questo buffer possono crearsi code così lunghe da riempirlo. In questo caso esistono due decisioni: si elimina un pacchetto della coda per far spazio a quello in arrivo o si elimina il pacchetto stesso in arrivo. Quest'ultima è preferibile perché permette al mittente di accorgersi della congestione.

Le **porte di uscita** presentano le stesse caratteristiche ma speculari. Inoltre l'accodamento può avvenire anche sul suo terzo blocco (che in realtà sarebbe il primo da sx) quando la commutazione da ingresso a uscita avviene troppo velocemente. Anche qui quindi si possono avere perdite.

La **struttura di commutazione** può essere di 3 tipi:



Di **memoria** in cui quando il pacchetto arriva nella porta di I/O va a memorizzarsi in memoria mandando un interrupt al processore che lo legge dalla memoria, ne estra la destinazione, controlla la tabella di inoltro e lo copia nel buffer della porta di uscita (proprio come fa la CPU con l'I/O, il bus e la memoria).

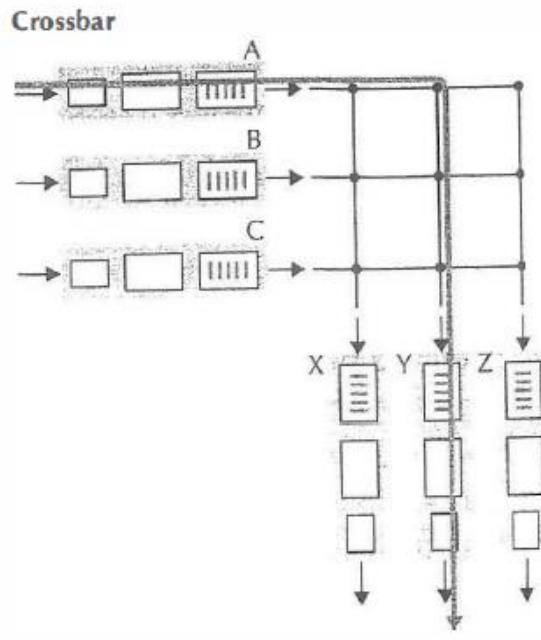


Tramite **bus** in cui il pacchetto nel terzo blocco viene decorato con una intestazione di commutazione dalle stesse porte di ingresso senza intervento del processore. Attraverso il bus il pacchetto arriva a *tutte* le porte di

output ma solo una lo accoglierà leggendo l'etichetta di intestazione che porta appunto la porta locale di output attraverso cui passare. Il bus è unico, quindi una grande limitazione e che più pacchetti che arrivano su porte di input diverse dovranno aspettare che il bus sia libero per loro.

NB. Quando si dice che il router implementa fino al livello 3 è da prendere con le pinze. In realtà il router implementa tutti i livelli della pila protocolare ma "funzionalmente" svolge le funzioni di livello 1-2-3. Per esempio ha dentro un

processore che effettivamente fa girare un software. Ma siccome non c'è interattività con l'utente e potrebbe anche non utilizzare alcun protocollo di trasporto allora è come se arrivasse fino al livello 3.



Tramite rete di interconnessione identica al precedente ma utilizza più bus per gestire parallelamente più pacchetti in ingresso in cui per n porte si avranno $2n$ bus: n orizzontali e n verticali. Ogni punto di incrocio può essere aperto o chiuso, e in questo caso formare un circuito dalla porta di ingresso alla porta di uscita. Questo modello supera il precedente. Infatti sono possibili più commutazioni da una porta di input a una porta di output in parallelo se però la porta di output

è diversa, altrimenti si accodano. Se un pacchetto arriva sulla prima porta di input A e vuole andare a Y allora si chiuderà l'incrocio per formare il bus A-Y. Se arriva un secondo pacchetto anche su A allora aspetta, se arriva su B allora se deve andare su X non aspetta, si chiude il punto di incrocio e si forma il bus B-X, altrimenti se deve andare su Y aspetta.

PROTOCOLLO INTERNET (IP)

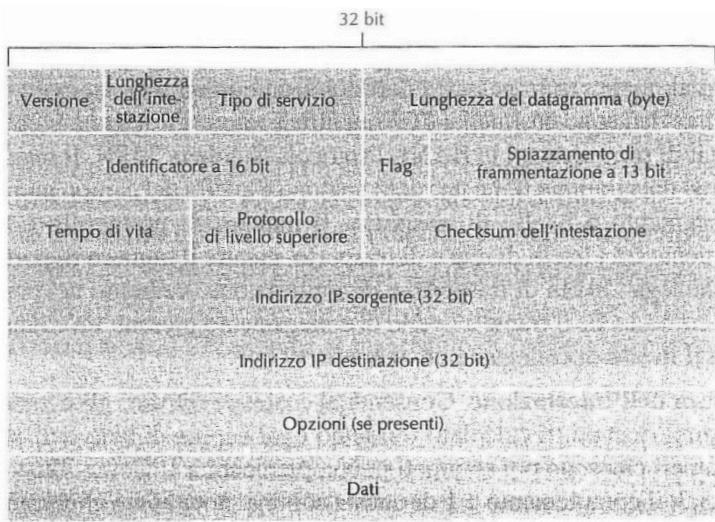
Attualmente si utilizzano due versioni del protocollo internet e sono: **IPv4** e **IPv6**. Se ricordiamo, il livello di rete non ha un unico protocollo che fa tutto quindi il protocollo internet fa parte di uno dei protocolli che costituiscono poi il livello di rete. In particolare il protocollo IP richiede alcune funzioni accessorie che vengono svolte da:

- 1) Protocollo DHCP
- 2) Protocollo ICMP
- 3) Protocollo ARP

IPv4

Formato dei datagrammi IPv4

I datagrammi sono il formato dei pacchetti che escono dal livello di rete.



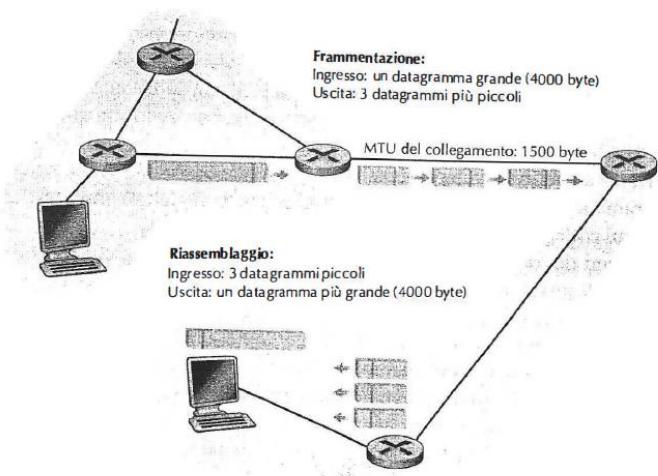
- **Tipo di servizio:** esistono diversi tipi di datagrammi che richiedono differenti servizi anche aggiuntivi che abbiamo visto inizialmente. Allora tali bit specificano il tipo di servizio che vuole il datagramma.
 - **Lunghezza del datagramma:** sono riservati 16bit per esprimere la lunghezza totale (header+payload) del datagramma. Quindi ogni datagramma può essere al massimo di 65.535 byte (ma di solito al massimo arrivano a 1500 byte).
 - **Identificatore, flag e spiazzamento di frammentazione:** sono tutti campi che hanno a che fare con la frammentazione che vedremo dopo (ipv6 non la contempla).
 - **Tempo di vita:** ogni volta che un datagramma viene elaborato da un router questo campo si decrementa. Se arriva a 0 il datagramma deve essere scartato.
 - **Protocollo di livello superiore:** ospita un numero che identifica il protocollo di trasporto a cui passare il payload del datagramma. Se 6 indica TCP, se 17 UDP.
 - **Checksum dell'intestazione:** ha lo stesso significato del checksum finora studiato, serve a stabilire l'integrità del datagramma.
 - **IndirizzoIP del sorgente e della destinazione**
 - **Opzioni:** facoltative
 - **Dati:** spesso contengono l'intero segmento proveniente dal livello di trasporto che quindi è diventato payload per il livello di rete.

- **Numero di versione:** 4 bit che indicano la versione del protocollo IP del datagramma.

- **Lunghezza dell'intestazione:** il datagramma ha delle opzioni facoltative. Senza opzioni in genere il datagramma ha un header di 20 byte dopo i quali inizia il payload. Con le opzioni questi numeri variano. Allora la lunghezza dell'intestazione indica la lunghezza dell'header o equivalentemente da quale byte inizia il payload.

Frammentazione dei datagrammi IPv4

I datagrammi IPv4, a differenza degli IPv6 possono essere **frammentati**. Che significa? Il livello di rete dovrà consegnare un datagramma al livello di collegamento che adesso si chiamerà **frame**. Esistono diversi tipi di collegamento e ciascuno può trasportare frame non più grandi di quanto stabilito dalla **MTU** (unità massima di trasmissione). Se questa fosse di 1500byte ma al livello di collegamento si consegna un datagramma di 3000byte come fa a trasportarlo? Allora si può pensare di fare al massimo datagrammi da 1500byte, ma anche questo è sbagliato. Ogni collegamento ha la sua MTU e quindi se un router riceve in ingresso un frame di 1500byte perché il collegamento in ingresso aveva una MTU di 1500byte ma scopre che lo dovrà dare in uscita su un collegamento in cui la MTU è di 500byte come si comporta? Per questo si è deciso di *frammentare* i datagrammi in più **frammenti** che hanno dimensione *variabile* e verranno *riassemblati* sui sistemi periferici di destinazione. Chi frammenta i datagrammi? Ciascun router. Vediamo un esempio:



- un collegamento con MTU pari a 1500.
- 3) Il router quindi creerà 3 frammenti che avranno lo stesso header ma in particolare lo stesso *identificativo* solo che contengono un diverso **spiazzamento** (che indica che in quel frammento vi sono i byte del datagramma originale che partono da un certo numero) e un **flag** che indica se il frammento è l'ultimo (0) o no (1). La frammentazione avviene in questo modo. Si considera il valore di MTU. Questo è 1500 quindi il primo frame sfrutterà il massimo possibile quindi sarà di 1500(20 di header e 1480 di payload), il secondo lo stesso e il terzo sarà formato dai restanti byte e quindi sarà di 1040 (sempre header compreso).
 - 4) Arrivato all'host di destinazione possono succedere due cose:
 - a. Se il protocollo di trasporto era UDP allora sapendo che il protocollo IP non è affidabile se si perde qualcosa il protocollo IP del destinatario non consegnerà il datagramma perché perdendo qualche frammento non sarà capace di comporre il datagramma originale a causa della non ritrasmissione di UDP.
 - b. Se il protocollo di trasporto è TCP allora se si perde qualche frammento il protocollo IP del destinatario scarta gli interi frammenti giunti correttamente e aspetta che TCP rimandi l'intero datagramma.

Indirizzamento IPv4

Ogni indirizzo IP è formato da 32 bit.

Esempio: 11010001 01000010 11111101 00000001

Però di solito si usa la notazione a punto con base decimale. In pratica ciascun byte dei 4 che formano l'indirizzo IP è scritto in decimale:

209.66.253.1

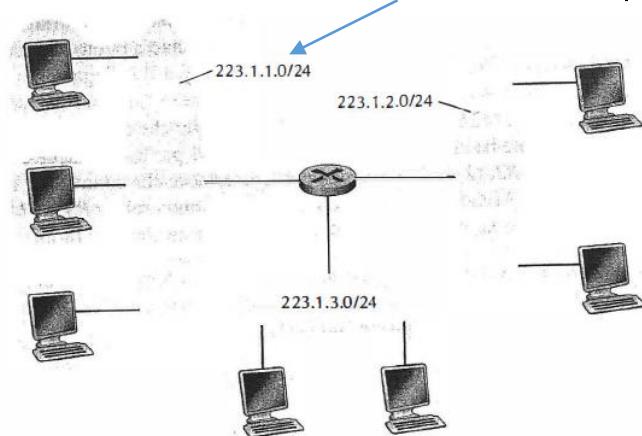
Ogni host e router hanno un loro indirizzo IP che li identifica univocamente nell'internet.

L'indirizzo non può essere scelto arbitrariamente ma dipende dalla **sottorete** o **rete IP** a cui è collegata. Una sottorete si crea quando uno o più host si collegano a un router. Ma un router a sua volta può appartenere a un'altra sottorete gerarchicamente più grande.

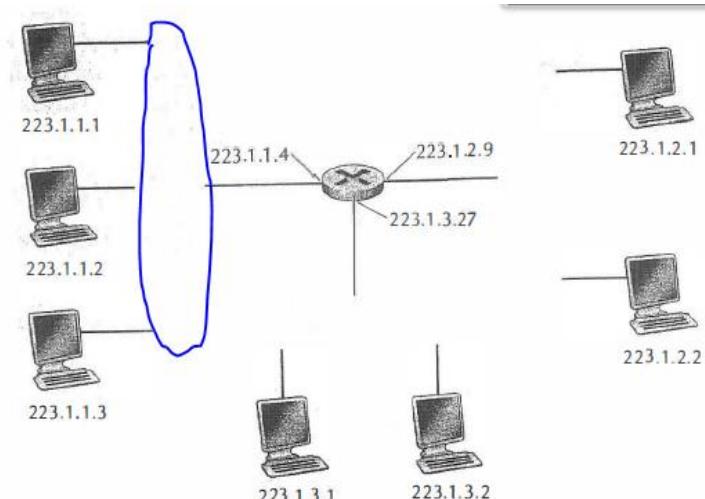
Una volta che un host o un router fa parte di una sottorete a esso gli viene assegnato un indirizzo IP così fatto:

x.y.z.w/n

/n viene detto **maschera di sottorete** o **subnet** e indica che gli n bit più significativi di x.y.z.w individuano l'**indirizzoIP della sottorete** o **netip**.



Se i tre host si uniscono alla rete collegandosi al router (il quale può essere collegato con altre sottoreti) formano la sottorete 223.1.1.0/24 (di netip 223.1.1.0) a indicare che tutti gli host che si collegheranno avranno indirizzi IP nella forma 223.1.1.x con x appunto variabile. Tipicamente tutti questi host sono collegati a un'unica entrata tramite **switch**. Quindi i 32-n bit meno significativi sono gli **hostid** e identificano l'id di ogni host nella sottorete.



E' importante capire che a un router viene assegnato un **blocco di indirizzi ip** come visto sopra e non ha un indirizzo IP specifico. Sono piuttosto le sue **interfacce ad avere un indirizzoIP**. Anche gli host non hanno un indirizzo IP, sono le **schede di rete** ad averlo.

All'inizio dell'Internet si era pensato di strutturarare i 2^{32} indirizzi IP possibili in un certo numero di sottoclassi. Questo approccio è detto **classfull addressing**. E' stato sostituito però da un nuovo approccio che vedremo perché con il classfull si è arrivati a esaurire tutti gli indirizzi IP.

Il nuovo approccio è detto **CIDR**, in cui la netmask è *arbitraria* (n):

x.y.z.w/n

indica che per una rete viene riservato un insieme di 2^{32-n} indirizzi *contigui* che hanno i primi n bit più significativi tutti uguali.

Esempio: 123.345.88.0/18 indica che alla rete vengono riservati 16384 indirizzi IP aventi i primi 18 bit più significativi uguali e vanno da 123.345.88.0 a 123.345.151.255.

Ma come vengono assegnati gli indirizzi IP?

Un amministratore di rete deve contattare il proprio ISP che gli fornisce degli indirizzi attingendo da un blocco più grande precedentemente allocato. Lo stesso ISP deve richiedere a sua volta un blocco di indirizzi. Gli indirizzi IP pubblici vengono assegnati da una autorità globale chiamata /CANN. Tutti gli indirizzi IPv4 sono esauriti, quindi non sono più disponibile. La soluzione è **IPv6** (in cui l'indirizzo IP si estende a 128 bit).

Un messaggio è detto di **broadcast** se viene mandato verso un indirizzo IP 255.255.255.255. Un messaggio del genere che parte da un host dice letteralmente "manda questo messaggio a *tutte* le entità presenti nella sottorete".

Inoltro in IP

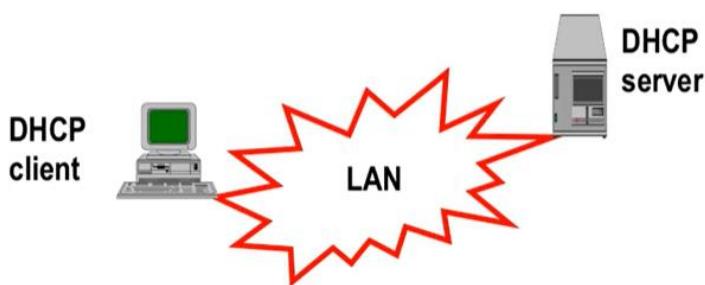
Potrei spiegare come funziona l'inoltro in IP, ossia come fa un host a inviare un pacchetto da un host a un altro che si trova sulla stessa rete locale o su un'altra, ma dovrei anticipare concetti come MAC address, switch etc... Quindi invece di dirvi "date per scontato questo", vi rimando all'omonima sezione che sarà però affrontata nel capitolo 5.

PROTOCOLLO DHCP: ASSEGNAZIONE DINAMICA DEGLI INDIRIZZI

Per assegnare un indirizzo IP a una interfaccia di rete posso farlo in due modi:

- **Staticamente**: a una interfaccia assegno un indirizzo IP e quello sarà persistente, per tutta la sua vita. Non è efficiente per due motivi:
 - ad oggi chi si collega a internet lo fa in mobilità e quindi si collega a reti diverse che hanno indirizzi di sottorete diversi.

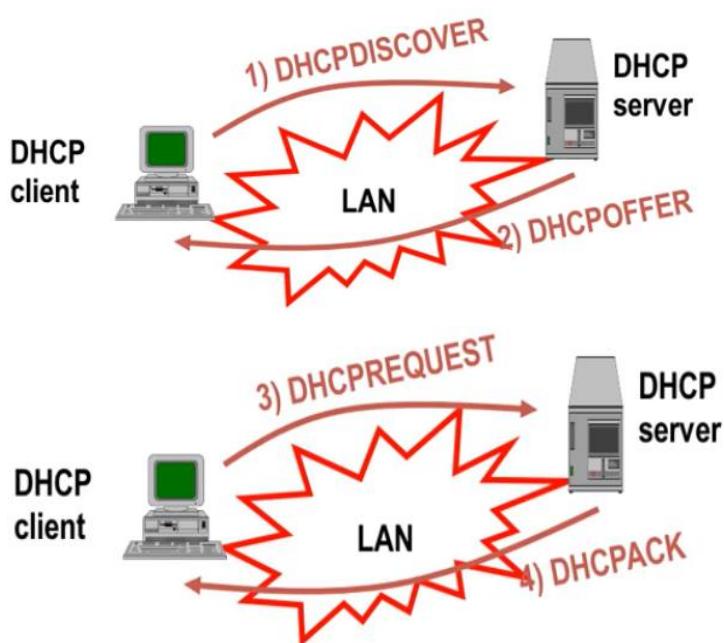
- se quell'indirizzo IP mi è assegnato in maniera persistente vuol dire anche che il mio indirizzo IP è quello anche quando non sono collegato a internet, quindi è una risorsa sprecata che potrei invece usare per qualcun altro.
- **Dinamicamente:** significa che qualcuno assegna un indirizzo IP in maniera dinamica che viene assegnato solo quando serve. Questo "qualcuno" è il protocollo DHCP che per questa sua capacità automatica è definito *protocollo plug-and-play*. Ci sono tre possibili associazioni tra host che vuole connettersi e indirizzo IP assegnatogli dal protocollo:
 - **Associazione statica:** esiste un server che ha una tabella a cui ad ogni indirizzo fisico corrisponde un indirizzo IP. Questo vuol dire che se un certo host *non* è collegato a internet allora non avrà indirizzo IP, ma ogni qual volta vi si collega *e far una richiesta esplicita* gli sarà associato sempre *lo stesso* indirizzo IP.
 - **Associazione automatica:** uguale alla statica ma la richiesta è automatica.
 - **Associazione dinamica:** uguale alla automatica ma l'indirizzo IP è consegnato in maniera casuale tra quelli disponibili (cioè non assegnati ad altri della sottorete).
 Ogni volta che si lascia la rete, l'indirizzo IP nel server diventa disponibile e se non ci sono indirizzi IP disponibili allora la connessione viene rifiutata.



DHCP è un protocollo di tipo *client-server* ed è un **protocollo di tipo applicativo** che si appoggia sull'UDP (tanto siamo in una rete locale, collegati tra fili e quindi non ci sono errori, ho solo bisogno di velocità) , il client gira sulla porta 68 e il server sulla porta 67, che serve però per

supportare il servizio di interconnessione di livello3. Il **ServerDHCP** si occupa di assegnare ad ogni host che ne fa richiesta un indirizzo IP. Il **ClientDHCP** è un software che gira su tutte le macchine che vogliono connettersi ed è attivo a fare richieste al serverDHCP e a comunicare con lui (tramite ovviamente scambio di pacchetti) per ricevere alla fine l'indirizzo IP.

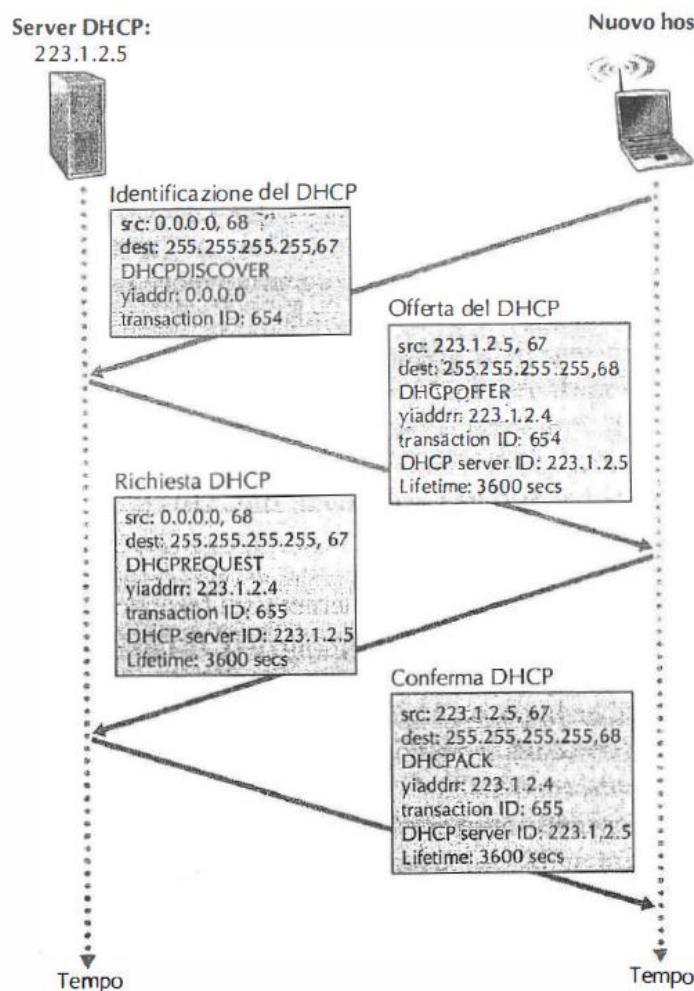
Qual è il flusso logico di una sessione logica tipo del DHCP?



Un client DHCP che vuole connettersi invia la richiesta che è un messaggio chiamato **DHCPDISCOVER** contenente il proprio *indirizzo fisico*. Il serverDHCP legge il contenuto e se possibile gli invia un messaggio **DHCPOFFER** contenente l'indirizzo IP da assegnargli.

Se il Client accetta l'assegnamento gli invia un **DHCPREQUEST** per dirgli "fantastico, accetto di navigare con l'indirizzo IP che mi hai assegnato", e il server di contro chiude il loop mandando un messaggio **DHCPACK** contenente tutte le informazioni di configurazione necessarie.

Vediamo meglio la sequenza di messaggi:



Ogni messaggio individua:

- src: chi sta mandando la richiesta
- dest: verso chi viene mandata
- tipo di messaggio
- yiaddr: indirizzoIP scelto per chi ne fa richiesta
- transaction ID: un semplice id del messaggio

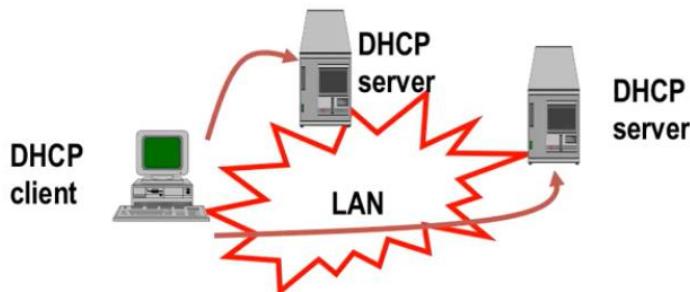
Il client non sa dove si trova il serverDHCP. Quindi invia un *messaggio broadcast* verso tutti quelli che appartengono alla sottorete. Ecco il perché del 255.255.255.255 verso la porta 67. Tutti riceveranno il DHCPDISCOVER ma solo il serverDHCP risponderà. Il client mette src = 0.0.0.0 perché questo indirizzo speciale indica che l'host non ha ancora un indirizzo IP.

Con il DHCPOFFER il serverDHCP sta dicendo "guarda io sono il server e ti offro una connessione alla sottorete

con indirizzo IP 223.1.2.4 per una *lifetime* di soli 6 minuti." Ecco perché si chiama

DHCPOFFER, perché è un'offerta che il server fa al client e che questo può accettare o meno.

Ma perché anche dopo la DHCPOFFER comunicano ancora con messaggi broadcast???



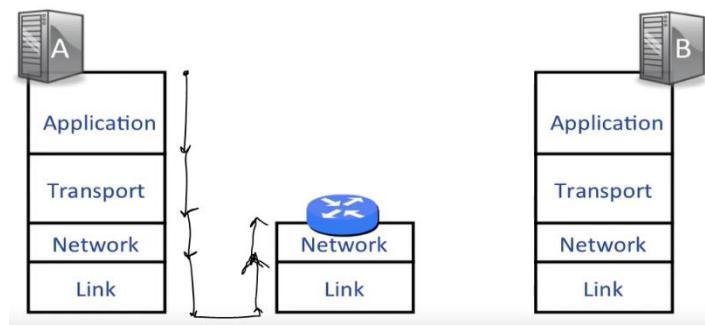
Il client non sa dove si trova il server DHCP e ne quanti ce ne sono in una stessa sottorete. Allora invia a tutti la richiesta e tutti possono rispondere. Sarà il client a decidere quale offerta accettare e quale rifiutare.

Qualche nota in più.

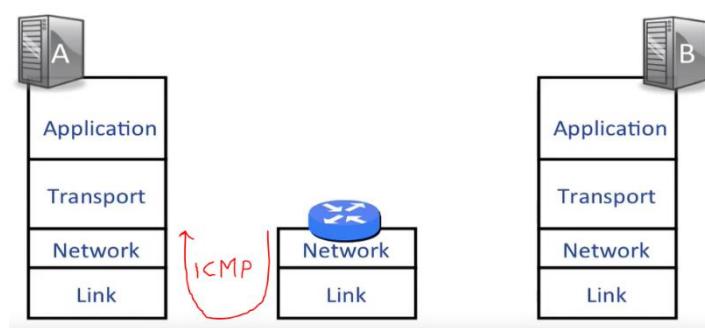
Un server DHCP gira su un server apposito oppure su un router.

Quando il client si disconnette abbiamo detto che il server rende disponibile per qualcun altro quell'indirizzo IP. Questo avviene quando il cliente notifica di volersi disconnettere tramite un messaggio **DHCPRELEASE**.

PROTOCOLLO ICMP



Supponiamo che A vuole inviare un pacchetto a B e quando il pacchetto (contenente l'indirizzo di destinazione) viene letto dal router e questo capisce che nella propria tabella questo indirizzo non esiste come si comporta ?



Il router in questo caso manda un messaggio che segnala al livello network che la destinazione non è stata trovata (*destination network unreachable*).

Il protocollo ICMP è un servizio che usa il livello di rete per comunicare al livello di rete con host e router. È una sorta di sistema di messaggistica tra entità per comunicare solo errori o report. Siccome usa IP per mandare i propri messaggi così come TCP allora è un protocollo che sta tra il livello di trasporto e quello di rete, ma è più di rete in quanto è usato solo per una comunicazione a livello di rete infatti è usato non solo da host ma anche da router. Esistono quindi due tipi di messaggi:

- **Error reporting** (sono generati dalle entità che si accorgono dell'errore)
 - o destination Unreachable (tipo 3)
 - o Source Quench (tipo 4)
 - o Time Exceeded (tipo 11)
 - o Parameter Problem (tipo 12)
 - o Redirection (tipo 5)
- **Query** (servono per fare il testing della connettività ip con un paradigma richiesta risposta)
 - o Echo request/Reply (tipo 8, 0)
 - o TimeStamp Request/Reply (tipo 13/14)

Indipendentemente se il tipo è error reporting o query, il formato di un messaggio ICMP è:



Dove:

- **Type**: specifica il tipo di messaggio (tipo3, tipo2 etc...)
- **Code**: è un codice che specifica ancora di più la causa del tipo sopra.
- **Checksum**: classico per validarne l'integrità
- **Resto dell'header**
- **Sezione Dati**

Esempio Error Reporting: Destination Unreachable

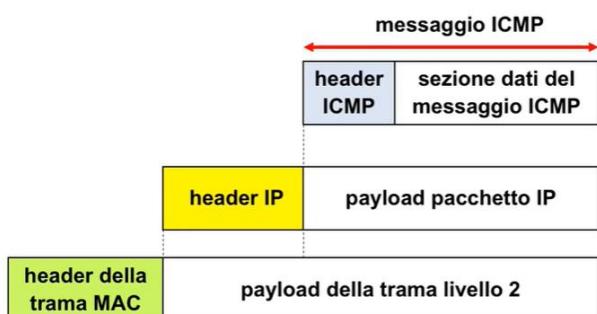
Se un pacchetto IP arriva in un router e si genera un errore allora il router genera un messaggio ICMP che torna indietro alla sorgente. In realtà la cosa è più particolare. Supponiamo che un pacchetto viene scartato per qualche motivo da un router.

type (3)	code (0-12)	checksum
non usato (0)		
header + primi 64 bit del pacchetto IP che ha causato il problema		

Il router crea un messaggio IP del formato sopra con tipo 3 (destination Unreachable), il codice è un numero da 0 a 12 che esplica il significato dell'errore, il checksum, una

restante parte dell'header non utilizzata e un particolare payload. Il router prende il pacchetto IP che ha generato l'errore, gli toglie l'header e i primi 8 byte del suo payload e lo

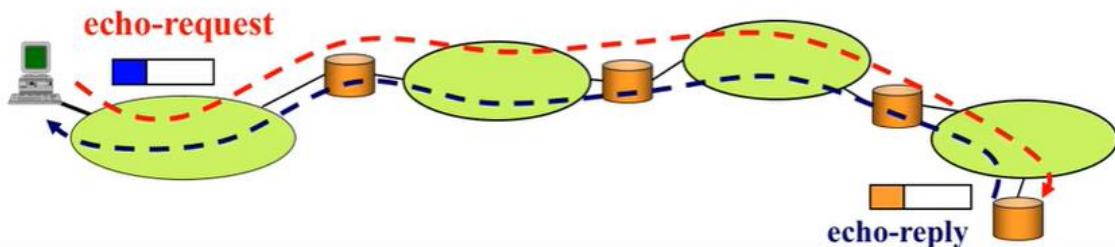
mette come payload del messaggio ICMP. Questo per far avere una visione più chiara alla sorgente di chi ha causato problemi.



Come ogni messaggio, trovandoci al livello di rete dovremmo inserirlo **incapsularlo** in un datagramma dove nel campo *protocollo* dovrà indicare ICMP. Poi per passarlo al livello di collegamento si crea il frame e così via..

Esempio Query: Echo request/Reply

Prevedono una interazione tra chi fa la richiesta e chi ne da la risposta. Se una entità IP vuole sapere se esiste un percorso IP per raggiungere una entità IP può farlo creando dei messaggi di *echo request* in ICMP, e se esiste un percorso allora la destinazione risponde con un *echo reply*. Questo indirettamente indica che le tabelle di routing sono ben configurate.



Ecco un messaggio di Echo request/reply:

type (8 request, 0 reply)	code (0)	checksum
identifier	sequence number	
optional data		

Il tipo è 8 se riguarda una richiesta, 0 se un reply. Non hanno codice ma hanno un checksum. Il resto dell'header è diviso in identifier che

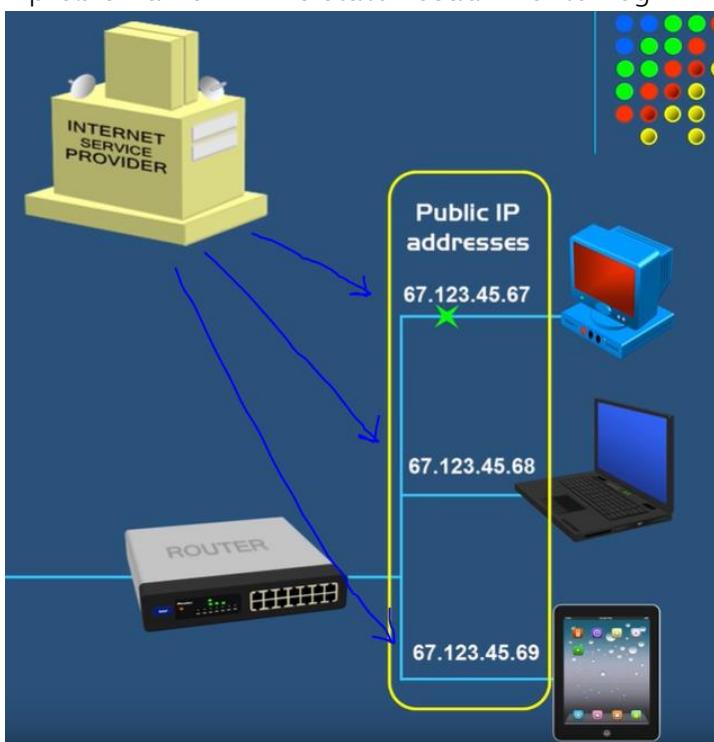
identifica la sessione (la risposta riporta lo stesso identifier della richiesta), e un sequence number che serve per distinguere i vari messaggi (avendo identifier uguale). La stringa di payload serve solo per avere una corrispondenza più forte tra echo request e echo reply, infatti se il mittente inserisce dei bit allora l'echo per farsi a maggior ragione riconoscere replica nel suo reply quei bit.

NAT: Network Address Translation

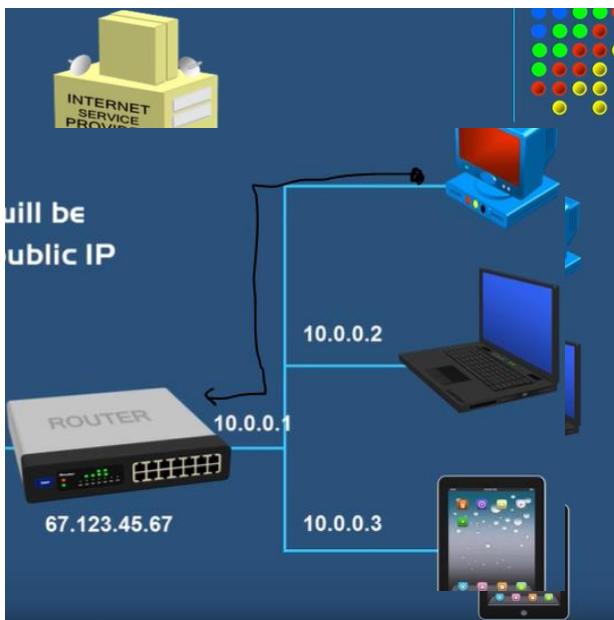
Abbiamo parlato di indirizzi pubblici. Cosa sono gli allora indirizzi privati?

Mentre un **indirizzo IP pubblico** identifica l'host univocamente nella rete mondiale e permette all'host di navigarvi e inviare/ricevere dati con altri host sparsi nella rete, un **indirizzo IP privato** invece è un indirizzo che identifica l'host univocamente nella stessa LAN. Può essere uguale a qualsiasi altro indirizzo IP privato in un'altra LAN perché serve solo a far comunicare diversi host tra di loro nella loro LAN e dunque non viene usato per instradare dati attraverso il router nella rete globale.

Il problema dell'IPv4 è stato l'esaurimento degli indirizzi IP.



E' davvero impensabile che al crescere dei dispositivi di una rete domestica si debba richiedere per ciascuno di essi all'ISP un indirizzo IP pubblico.

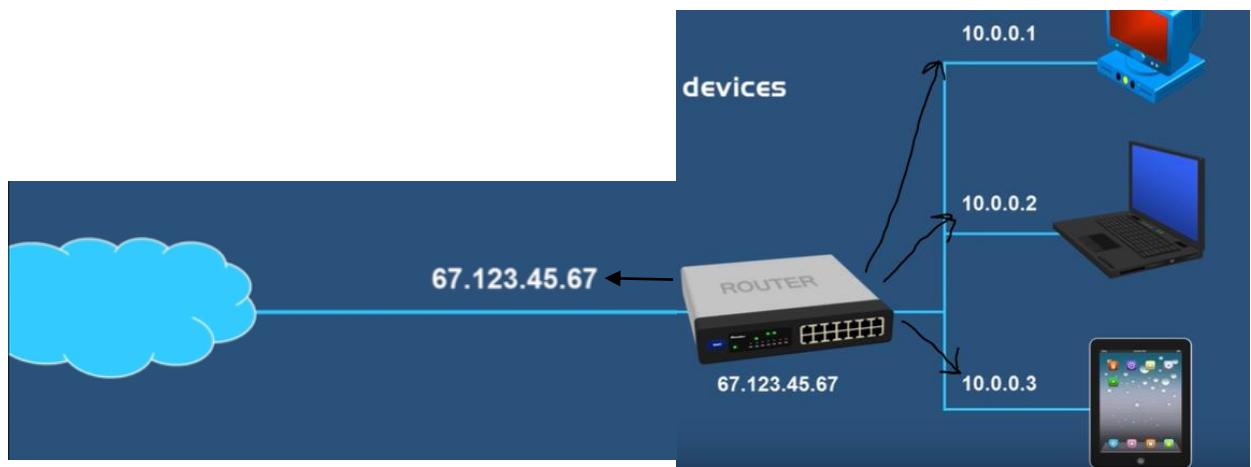


Quello che succede invece è che il router richiede un indirizzo IP al provider solo per se stesso, o meglio a un Server DHCP che sicuramente appartiene al provider.

E che poi il router assegnerà indirizzi IP privati a ciascun dispositivo collegato al router tramite un proprio server DHCP.

In questo modo si sono risparmiati 3 indirizzi IP pubblici. Ma sorgono due problemi. Come fa ciascun host a instradare dati nella rete globale?

Se i router sono abilitati al NAT (ciò significa che se non lo sono allora l'assegnamento degli indirizzi IP sarà come visto prima, e sono tutti pubblici), allora quando un host di una LAN vuole comunicare con la rete globale il suo indirizzo IP verrà tradotto, come succede per tutti gli altri host, in un unico indirizzo IP uguale per tutti che è quello del router:



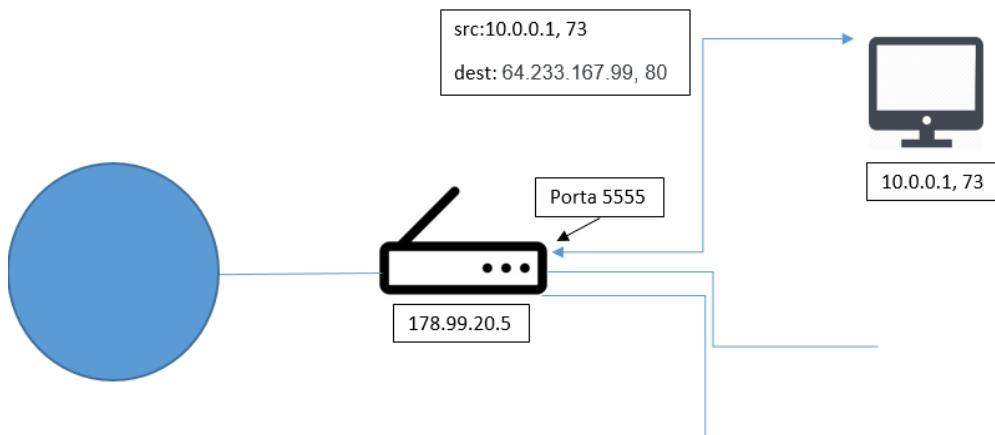
Semplicemente tutti gli host è come se avessero lo stesso IP address. Ma in andata è semplice, è il ritorno che porta problemi. Quando un server nella rete globale risponde, come si fa a capire quale dei tre host (in questo esempio) abbia fatto la richiesta?. Ogni router abilitato al NAT possiede all'interno una **tabella di traduzione NAT** che riporta per ogni riga le seguenti informazioni:

	LATO WAN	LATO LAN
riga i-esima	indirizzoIP router, porta scelta	indirizzoIP privato dell'host, porta a cui inviare i dati

NB: Spesso si parla di NAT ma si intende la sua evoluzione PAT. Qui stiamo in realtà parlando della sua evoluzione chiamata appunto PAT perché ormai è diffusa quasi solo questa. Il motivo è che il NAT puro non utilizza alcuna porta nella tabella per distinguere i diversi host. Questo causa il problema che se il router possiede un solo IP pubblico può concedere di utilizzarlo a un solo host alla volta e non di più contemporaneamente perché quando arriva la risposta non saprebbe discriminare i casi, mentre col PAT anche se tutti utilizzano lo stesso indirizzo IP pubblico, a ciascuno è stata data una porta diversa per cui è facile distinguerli.

Supponiamo che un host abbia indirizzoIP privato 10.0.0.1 ed è attivo sulla porta 73.

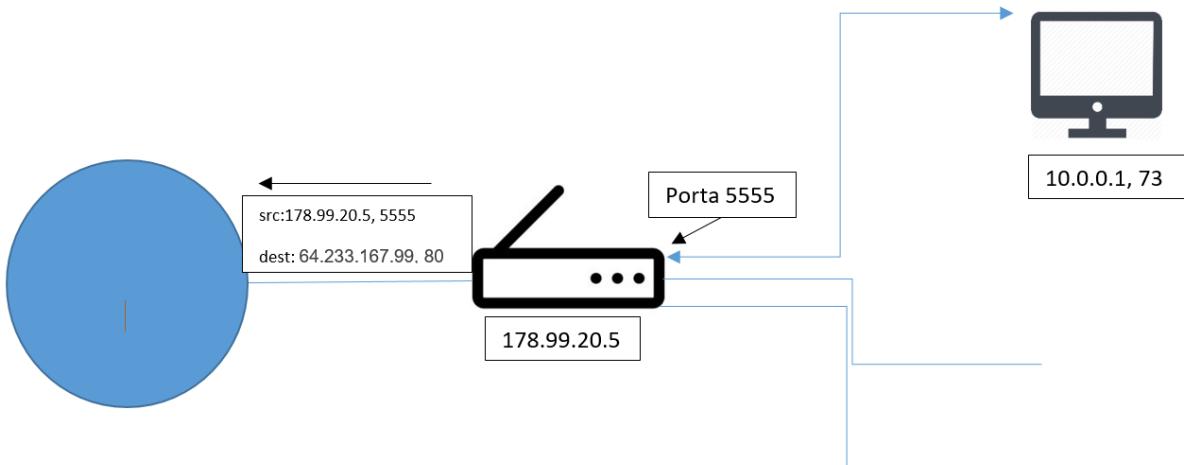
Supponiamo che si voglia collegare a facebook e quindi fa la richiesta al router:



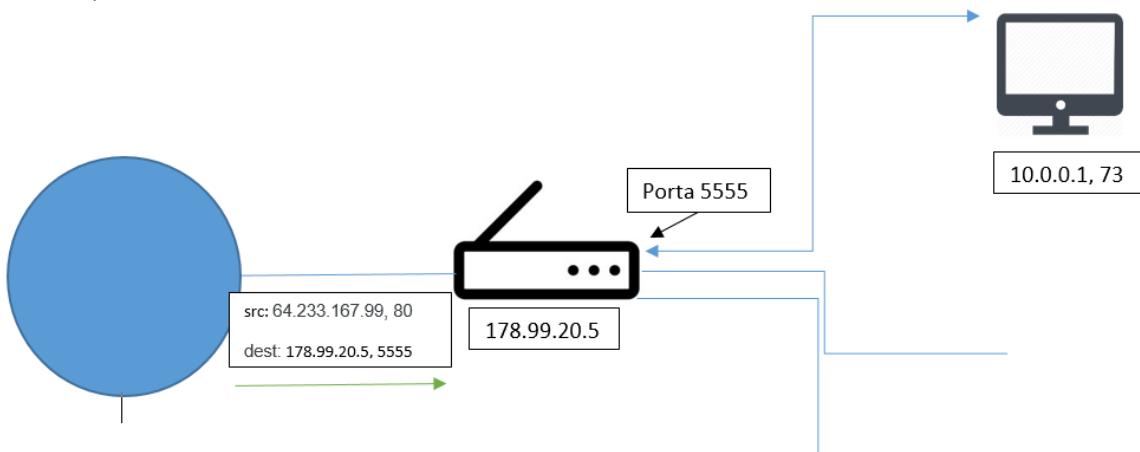
Allora il router assegna la porta 5555 come identificativo di uno dei suoi host che vuole entrare nella rete globale.

	LATO WAN	LATO LAN
riga i-esima	178.99.20.5, 5555	10.0.0.1, 73

Fa lui la richiesta.



Il server risponde:



Il router consulta la tabella di traduzione NAT e vede che il server ha inserito come destinazione una porta 5555. Così il NAT controlla la tabella e capisce che 5555 è la porta assegnata all'host 10.0.0.1:73, così modifica nel pacchetto la porta 5555 scrivendogli 73 e anche l'indirizzo IP 128.99.20.5 scrivendogli 10.0.0.1.

NB. Col passaggio da IPv4 a IPv6 il NAT non sarà più usato.

IPv6

Il numero di dispositivi che richiedono connettività è molto maggiore dei 2^{32} possibili offerti dalla IPv4, infatti la IANA ha esaurito da 5 anni (2015) tutti gli indirizzi IPv4.

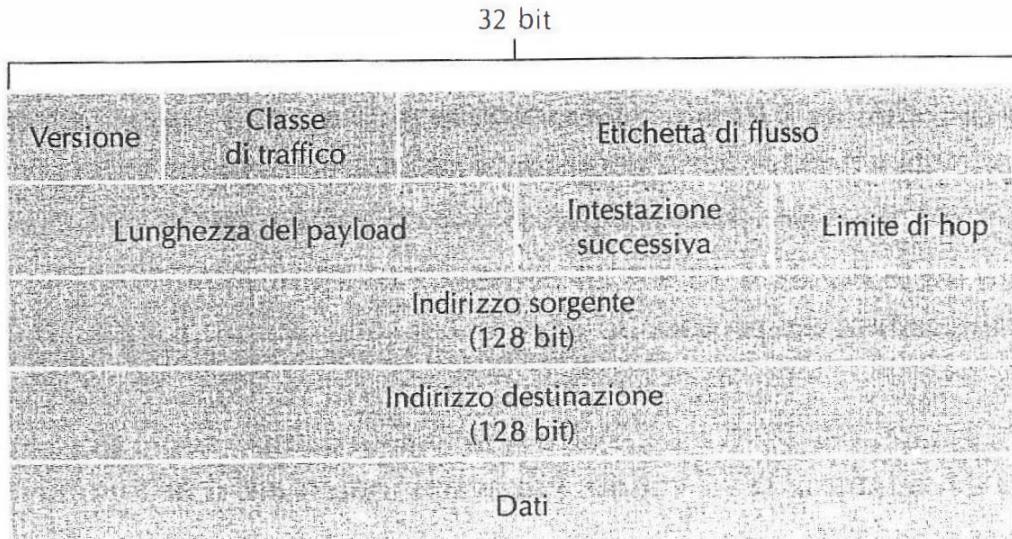
La differenza primaria riguarda la lunghezza degli indirizzi:

da 32 bit -----> a 128 bit

Cambiano inoltre i protocolli di segnalazione che gestiscono gli indirizzi IPv6. Infatti:

- *ICMPv6* è la nuova versione e si occupa in più di quello che faceva l'*ARP* in IPv4
- *DHCPv6* modificato per il nuovo protocollo
- *Routing*

FORMATO



E' un formato più semplice di quello di IPv4. Le novità sono:

- **Classe di traffico e etichetta di flusso:** definisce la priorità del traffico e tramite l'etichetta di flusso etichettare tutti i pacchetti IP quelli che appartengono allo stesso flusso.
- **Limite di hop:** limita il numero di hop nel pacchetto IP. Decrementato di 1 da ogni router, quando arriva a zero viene eliminato.
- **Intestazione Successiva:** equivalente al Protocollo dell'IPV4.

IPv6 non ospita campi per la **frammentazione** perché non ne permette la possibilità. Se una sorgente invia un pacchetto troppo grande allora il router manda un messaggio ICMP che riporta "pacchetto troppo grande" e procede a eliminarlo. Questo perché IPv6 tende ad essere più veloce e quindi evita frammentazione e **riassembaggio** che tolgo tempo.

Manca anche il **checksum** che i progettisti IP hanno per esperienza visto non avere poi una così grande utilità, piuttosto è elevato il tempo di elaborazione e quindi sempre per lo stesso motivo detto prima hanno preferito toglierlo.

Scompare anche il campo **opzioni**, però non del tutto. Infatti il campo *Intestazione Successiva* del protocollo IPv6 può puntare a un header TCP, UDP oppure a uno **extension header**. Queste ultime fanno le veci delle opzioni di IPv4.

NOTAZIONE

Sono indirizzi da 128bit scritti stavaolta in esadecimale in cui ogni 2 byte si scrive il relativo esadecimale.

8000:0000:0000:0000:8965:0678:A45C:87D3

Se la coppia è formata da zeri allora possono essere omessi:

8000::8965:678:A45C:87D3



In IPv4 avevamo detto che ogni interfaccia di rete aveva il suo univoco indirizzo IP, mentre con IPv6 ad una stessa interfaccia di rete possono essere assegnati indirizzi diversi a seconda del *destinatario* a cui sono collegati o all'*uso* che se ne deve fare.

PASSAGGIO DA IPv4 a IPv6

Non c'è alcun problema oggi giorno al passaggio da IPv6 a IPv4, questo perché i nuovi sistemi sono retrocompatibili, ossia capaci di gestire anche indirizzi IPv4 (fondamentalmente IPv4 è un IPv6 di soli 32 bit). Il problema è come si fa a gestire indirizzi IPv6 quando i sistemi sono capaci di gestire IPv4.

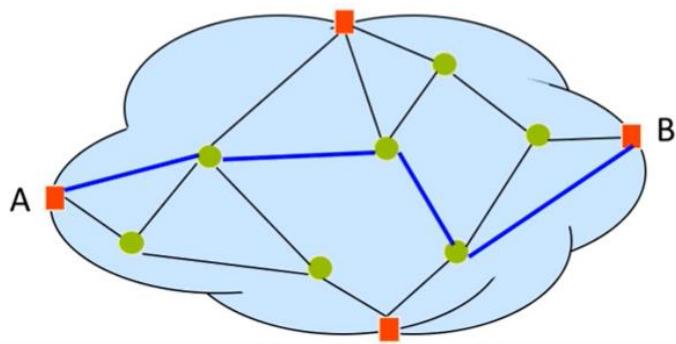
Esistono due approcci possibili:

- **Dual Stack:** tutti i nodi (server, router) hanno un *doppio stack* che replica le stesse risorse in connettività IPv4 e IPv6. Tramite l'uso dei DNS si incanala la comunicazione verso gli stack IPv4 o IPv6 a seconda se il nodo che manda la richiesta abbia indirizzo IPv4 o IPv6. C'è un solo problema: se tra due host che supportano IPv6 ci siano nodi che supportano oltre a IPv6 anche IPv4 la cosa diventa infattibile. C'è proprio bisogno di due percorsi differenti per ciascuna versione.
- **Tunneling:** se devo far passare un pacchetto in IPv6 dentro una rete IPv4 allora incapsulo i pacchetti IPv6 dentro pacchetti IPv4. Il tunnelling risolve il problema fondamentale del dual stack.

INSTRADAMENTO

Abbiamo visto che quando un pacchetto arriva in un router questo ne decide ,guardando la **tabella di inoltro o forwarding**, il trasferimento da un *collegamento in ingresso* a un *collegamento in uscita* al router stesso. Attraverso invece i **protocolli di instradamento** si aggiornano queste tabelle di inoltro in ogni router che fa parte del percorso da A a B attraverso gli stessi router che comunicano tra loro. Il percorso migliore viene scelto da **algoritmi di instradamento**.

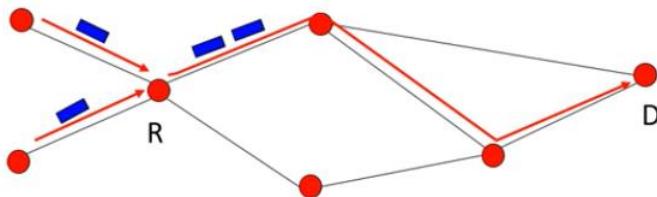
Quando un pacchetto deve andare da A a B, indipendentemente da tutti i router del percorso, ci sono due router in particolari che sono il punto di inizio e arrivo dei dati. Per A il primo router con cui si interfaccia alla rete globale è detto **first-hop router** o **router sorgente**, mentre quello di B si chiama **router di destinazione**. Trovare il percorso ottimale da A a B significa trovare il percorso ottimale dal router sorgente a quello di destinazione.



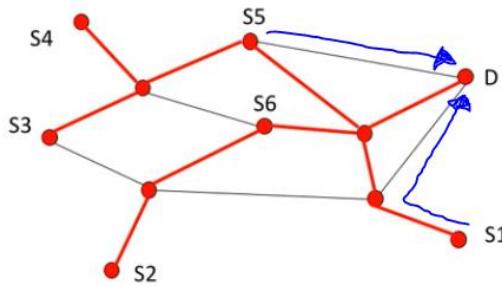
Tratteremo algoritmi **unicast**, ossia che hanno come destinazione uno e un solo nodo.

Ricordiamo che ci stiamo basando su Internet che in genere è una **rete datagram**. Quindi considereremo **tabelle di inoltro** fatte da prefisso e interfaccia di uscita. Pertanto quando un pacchetto arriva al router, si legge l'indirizzo IP di destinazione, si va a vedere dove in questa tabella è messo e lo si inoltra nella corretta porta di uscita. Tale tabella di inoltro è stata riempita da una **tabella di routing** calcolata dal **processore di instradamento** tramite **algoritmi di routing** (che adesso vedremo). E' sempre lo stesso processore di instradamento che poi distribuirà una copia di queste tabelle di inoltro sulle schede di ingresso di ciascun router.

La *politica di routing* si basa sull'indirizzo IP di destinazione e dell'inoltro al successivo nodo (next-hop routing), questo ha delle conseguenze:



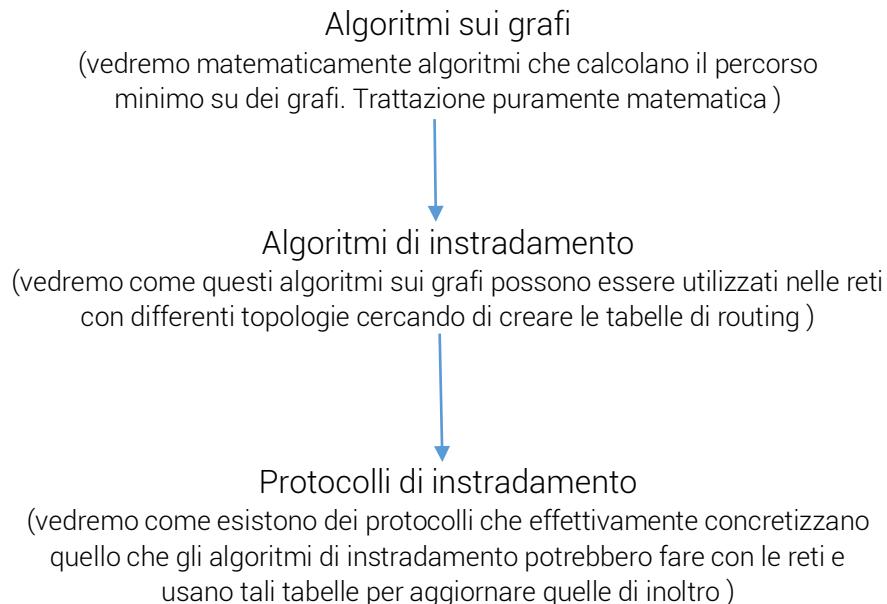
Qualsiasi sia la provenienza di due pacchetti quando arrivano in un router, se arrivati a quel router essi avranno lo stesso indirizzo IP di destinazione allora da lì in poi seguiranno la stessa strada.



Non è quindi possibile instradare in maniera indipendente flussi di traffico provenienti da sorgenti diverse perché i protocolli di instradamento troveranno per ogni router il miglior percorso che da lui conduce a tutti i possibili altri router.

NB: I singoli algoritmi *sui grafici* che vedremo permetteranno di trovare il percorso minimo tra un nodo A e un nodo Z. Però gli algoritmi *di routing* devono dotare ogni router di una tabella (di inoltro) tale che per ogni rotta avranno un percorso minimo stabilito. Quindi se mi trovo in A e voglio andare in Z, controllo la tabella e so dov'è il prossimo hop. Se voglio andare a Y controllo la tabella e so dov'è il prossimo hop e così via. Pertanto nei grafici vedremo come calcolare il percorso minimo da A a B ma poi sarà esteso da A a tutti e in tutti i router si farà lo stesso.

Quello che vedremo adesso seguirà questa logica:



ALGORITMI SUI GRAFI

ALGORITMO DI BELLMAN-FORD

Le **ipotesi** di funzionamento dell'algoritmo sono:

- I pesi degli archi possono tranquillamente essere sia positivi che negativi
- Non deve però esistere alcun ciclo che si chiude con lunghezza negativa

Ma che senso può avere un peso negativo? A livello fisico nessuno, ma a livello pratico servono per favorire la probabilità che si possa scegliere un certo nodo. Se voglio che il flusso di traffico passi da un certo nodo, mettere tutti i suoi archi entranti di peso negativo favorirà la scelta come collo di bottiglia.

Lo **scopo** è di trovare *i cammini minimi* da un nodo sorgente *s* a *tutti gli altri*. Quindi mi calcola tutto l'albero.

Definisco i valori iniziali:

$$D_s^h = 0 \quad \forall h$$

$D_i^0 = \infty \quad \forall i \neq s$ (con $s=1$ nodo di riferimento, ma potrebbe essere anche un altro)

n.b: $D_i^0 = \infty$ si legge: "posso raggiungere il nodo 1 dal nodo i utilizzando al massimo zero archi con un costo infinito", perché impossibile.

$D_s^h = 0 \quad \forall h$ si legge: "la distanza del nodo di riferimento/sorgente, che sarebbe 1 nel nostro caso, sarà 0 per ogni numero di archi possibili", infatti da un nodo raggiungo il nodo stesso con 0 archi.

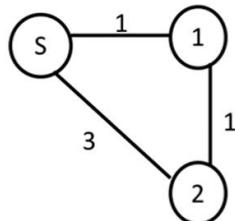
Ogni altro D_i^h , che indica il cammino minimo dal nodo i al nodo 1 a patto che il numero di archi/hop siano $\leq h$ si troverà iterando:

$$D_i^{h+1} = \min \left[D_i^h, \min(D_j^h + d_{ij}) \right]$$

In pratica si eseguono un numero di iterazioni pari al numero di archi esistenti. In ognuna di queste iterazione si applica quella formula al nodo $i \neq s$ se e solo se esiste un cammino di h archi passante per un nodo j che lo conduce a s . Se esiste allora si applica quella formula e se $D_j^h + d_{ij}$ risulta minore rispetto a prima, il nodo i ha trovato un cammino più lungo in termini di archi ma meno costoso passante per j che ne diventa il next-hop.

Bellman-Ford è utilizzato come algoritmo decentralizzato in Distance Vector.

Esempio:



Inizializzazione

$$D_s^0 = D_s^1 = D_s^2 = 0$$

$$D_1^0 = D_2^0 = \infty$$

ITERO

Iterazione1: significa che $h=1$, quindi devo andare a vedere per ogni nodo qual è il percorso con numeri di hop = 1 che consente a quel nodo di raggiungere la sorgente.

$D_1^1 = \min(D_1^0, D_s^0 + 1) = \min(\infty, 1) = 1$ il percorso che deve fare 1 per raggiungere s con $h=1$ archi ha costo uno.

Quindi il **next-hop di 1 è S** perché con S si ha costo minore.

$D_2^1 = \min(D_2^0, D_s^0 + 3) = 3$ il percorso che deve fare 2 per raggiungere s con $h=1$ archi ha costo 3.

Quindi il **next-hop di 2 è S** perché con S si ha costo minore-

Se mi fermassi qua capirei subito che non va bene in quanto il nodo 2 per raggiungere la sorgente ha un cammino a costo minore che passa per il nodo 1

Iterazione2: significa che $h=2$, quindi devo andare a vedere per ogni nodo qual è il percorso

con numeri di hop =2 che consente a quel nodo di raggiungere la sorgente.

$D_2^2 = \min(D_2^1, D_2^1 + 1) = 1$ il percorso che deve fare 1 per raggiungere 2 con h=2 archi ha costo uno.

Il next-hop di 1 continua ad essere S.

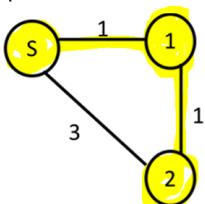
$D_2^2 = \min(D_2^1, D_2^1 + 1) = 2$ il percorso che deve fare 2 per raggiungere s con h=1 archi ha costo 3.

Quindi il nuovo next-hop di 2 è 1.

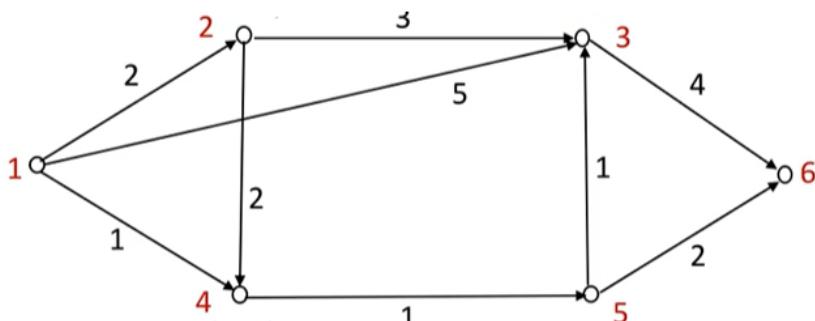
Termino le iterazioni perché ho fatto $N-1=3-1=2$ iterazioni.

Ad ogni iterazione si deve sempre confrontare il nodo i non i restanti (s e 2 in questo caso).

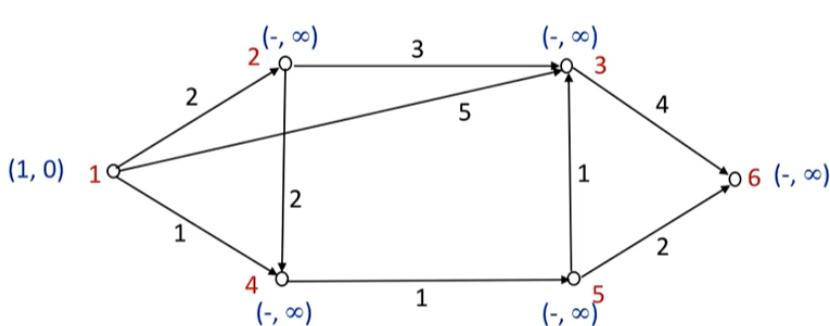
Riassumendo ho che il next-hop (ossia prossimo router) di 1 è S mentre quello di 2 è 1 quindi l'albero è:



Esercizio 2. Modo grafico di risoluzione

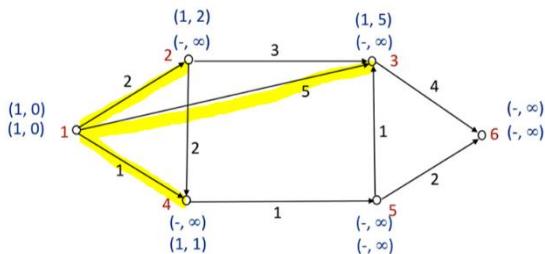


Voglio trovare l'albero dei cammini minimi da 1 a tutti gli altri.



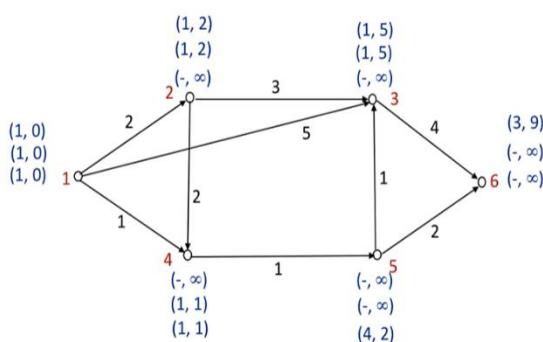
Inizializzo

Ogni (n, L) sul nodo i indica il che n è il next-hop di i e L è il costo per arrivarci.



Iterazione con $h=1$

Prendo solo i nodi che distante $h=1$ hop dal nodo uno (e sono 2,3 e 4). Le etichette cambiano secondo questa logica: prendo ad esempio quella del nodo 3 che è $(1,5)$. Posso raggiungere il nodo 1 usando come next-hop proprio 1 per un costo di 5.



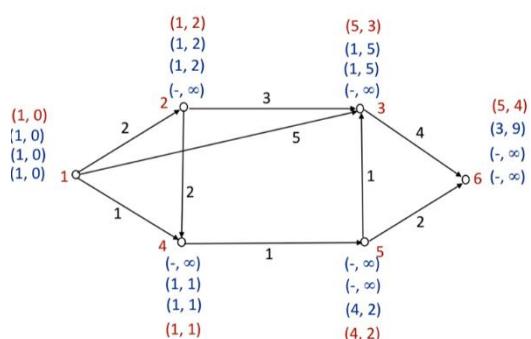
(3,9)

Iterazione con $h=2$

Rifaccio quanto sopra, ma con $h=2$. Inoltre se trovo un cammino devo vedere se con $h=1$ ne ho ottenuto uno a costo minore.

Esempio 1. Per il nodo 3 ho che rimango con $(1,5)$ perché l'unico percorso con due archi è 3-2-1 ma ha lo stesso costo di 5.

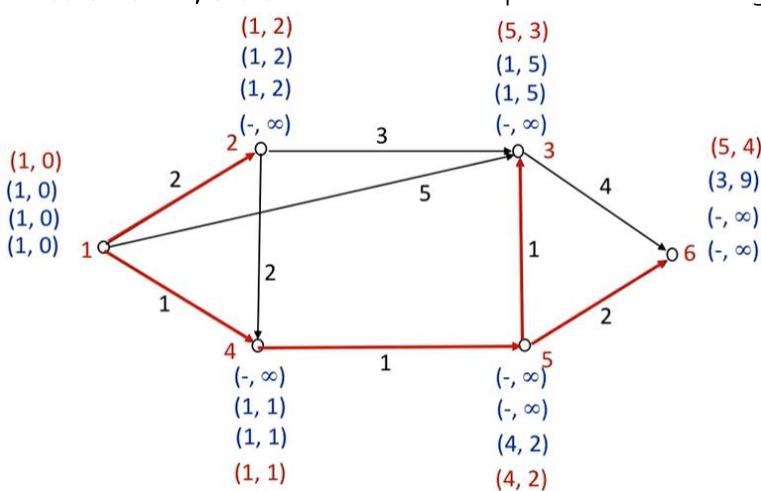
Esempio 2. Per il nodo 6 ho un solo percorso che usa due archi ed è 6-3-1 quindi next-hop=3 e costo=9 ->



Iterazione 3 con $h=3$

Cambia il nodo 3 perché esiste un percorso con 3 archi (3-5-4-1) che ha costo minore pari a 3, quindi per il nodo 3 si avrà etichetta $(5,3)$. Cambia anche l'etichetta del nodo 6.

Le iterazioni 4, 5 e 6 non cambiano quindi avrà convergenza.



L'albero dei cammini minimi si trova evidenziando per ogni nodo l'arco che connette questo al suo next-hop. Per esempio il next-hop del nodo 3 è il nodo 5 quindi l'arco 3-5 farà parte dell'albero.

ALGORITMO DI DIJKSTRA

Le **ipotesi** di funzionamento dell'algoritmo sono:

- I pesi degli archi possono essere solo positivi (altrimenti utilizzo il precedente)

Lo **scopo** è quello di trovare i cammini minimi da tutti i nodi a un nodo di riferimento 1.

L'inizializzazione prevede l'assegnamento a un insieme P del solo nodo di riferimento:

$$P = \{1\}$$

e l'inizializzazione delle distanze di ogni nodo dal nodo di riferimento:

$$D_1 = 0 \text{ e } D_j = \infty$$

quindi si pongono inizialmente le distanze dalla sorgente infinito. Inoltre si pongono indefiniti i next-hop dei nodi tranne per quello sorgente:

$$\text{next_hop}_j = - , \text{ next_hop}_1 = 0$$

L'iterazione prevede di:

- 1) Scegliere un **nodo** i con $i \in P$ tale che:

$$D_i = \min_{j \in (P)} D_j$$

Ossia il nodo con distanza minima dalla sorgente (nell'attuale iterazione).

Estrarre questo nodo da P :

$$P := P \setminus \{i\}$$

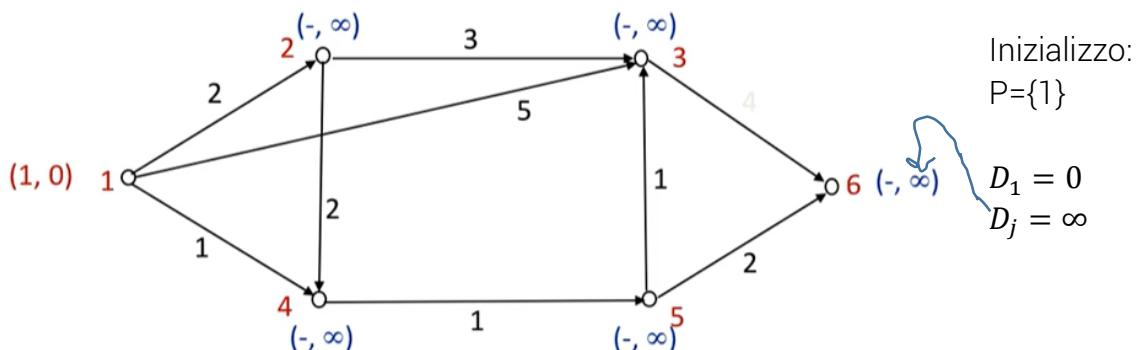
- 2) Estratto il nodo i si individuano i nodi j adiacenti a quello appena estratto e per questi si fa quanto segue:

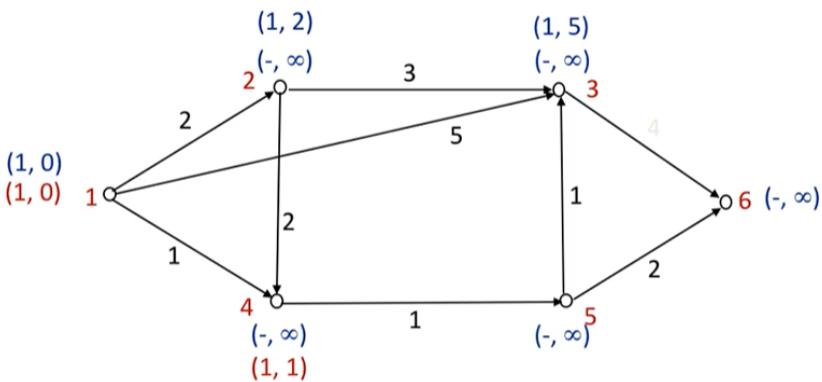
$$\text{se } D_j > D_i + d_{ij} \text{ allora } P := P \cup \{j\}, \text{ next_hop}_j = i, D_j = D_i + d_{ij}$$

Se $P=0$ (ossia non ci sono più nodi da controllare perché sono stati tutti tolti da P) allora STOP.

In bellman io dovevo controllare ad ogni iterazione per ogni nodo se ci fosse un percorso minimo migliore di quello precedente. In Dijkstra siccome c'è l'assunzione che i pesi non sono negativi allora ad ogni iterazione non controllerò tutti i nodi (ma appunto quelli dell'insieme $N - P$ aggiornato ad ogni iterazione).

Esempio:



**Iterazione 1**

Estraggo in P quello che ha D_j minore, ossia $1 \rightarrow P = 0$

Il nodo 1 ha come nodi adiacenti il 2, 3 e 4.

- $\infty >? 0 + 2$ si allora

$$D_2 = 2,$$

$$\text{nexthop}_2 = 1,$$

$$P = \{2\}$$

- $\infty >? 0 + 5$ si allora

$$D_3 = 5$$

$$\text{nexthop}_3 = 1,$$

$$P = \{2,3\}$$

- $\infty >? 0 + 1$ si allora

$$D_4 = 5$$

$$\text{nexthop}_4 = 1,$$

$$P = \{2,3,4\}$$

Poiché P non è vuoto si continua.

Alla seconda iterazione estraggo da P il nodo 4 perché ha D_i più piccolo tra i nodi in P e continuo fino a che P non diventi vuoto.

Alla fine avrà per ogni nodo il suo nexthop e come per bellman prenderò l'arco che congiunge il nodo col suo nexthop.

CONFRONTO TRA BELLMAN-FORD E DIJKSTRA

Oltre a quello già detto prima, vediamo altro.

COMPLESSITÀ'

L'algoritmo di *Bellman-Ford* per ogni h da 1 al numeroDiArchi nel caso peggiore in cui ogni volta per ogni arco ci sia un percorso con h archi e nel caso tale percorso esiste passando da tutti gli $N-1$ nodi allora la complessità è $O(n^3)$, però questo è un caso davvero raro e pertanto in linea generale ha complessità $O(AN)$ con A vertici e N nodi.

L'algoritmo di *Dijkstra* richiede $N-1$ iterazioni e in media ogni iterazione richiede N operazioni nel caso peggiore, quindi abbiamo una complessità $O(n^2)$

ALGORITMI DI INSTRADAMENTO/ROUTING

L'algoritmo di instradamento può essere:

- **Globale**: algoritmi globali hanno una conoscenza globale e completa della rete, quindi gli si danno in input tutti i collegamenti tra i nodi e i loro costi e da come output il percorso a costo minimo tra sorgente e destinazione. Un esempio è il **link state (LS)**.
- **Decentralizzato**: algoritmi decentralizzati calcolano il percorso a costo minimo in modo distribuito e iterativo. Ciascun nodo è consci solo dei nodi adiacenti e dei costi dei legami con loro. Un esempio è il **distance vector (DV)**.

Entrambi i due tipi possono essere poi:

- **Statici**: se i percorsi cambiano raramente e quindi basta un intervento umano (alla tabella di inoltro).
- **Dinamico**: se invece gli instradamenti variano al variare del volume di traffico o della topologia della rete.

ALGORITMO DISTANCE VECTOR

Questo algoritmo è usato da *protocolli di instradamento* che si trovano davanti una topologia di rete decentralizzata. Utilizzano questo *algoritmo di instradamento* per ricavarne la tabella di routing su ogni router. Starà poi a loro aggiornare quella di inoltro.

Questo algoritmo di instradamento si basa sull'*algoritmo sui grafi di Bellman-Ford*.

L'idea di base è la seguente: "poiché la rete è decentralizzata allora ogni nodo conoscerà solo i collegamenti con i nodi adiacenti. Ogni router per ricostruire una tabella di routing completa (quindi come se volesse riportare la rete decentralizzata in globale) invia periodicamente dei messaggi chiamati **distance vector (DV)** che a seconda di regole che vedremo dopo causerà l'aggiornamento iterativo come a effetto domino di tutta la rete fino a quando tutti i router avranno costruito una tabella di routing che esprime la situazione della rete nel globale".

Formato del DV: -----> DV = [destinazione, costo]

Se inviato dal mittente A verso B vuole dire che per raggiungere il router *destinazione* intraprende un cammino che ha costo *costo*.

All'inizio ogni router possiede una tabella di routing fatta solo così:

Da A verso	Link	Costo
A	0	0

Ossia ogni router possiede solo l'informazione su se stesso. (nell'esempio è il router A)

Bisogna solo sapere una regola.

Un router invia i DV solo:

- Ai propri nodi adiacenti
- Periodicamente o quando il *ricalcolo* di una distanza cambia.

Di conseguenza il ricalcolo è fatto ogni volta che:

- Cambia la topologia della rete (nuova aggiunta o rimozione)
- Riceve un DV da un router adiacente

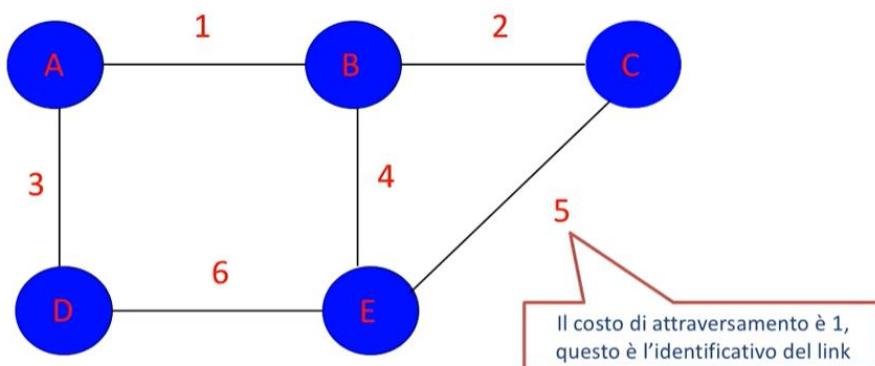
Di conseguenza quando riceve un DV quello che fa è questo:

- Se la tabella di routing non possiede la destinazione nel campo del DV allora la inserisce nella tabella con relativa distanza incrementata del costo del link col quale è collegata.
- Altrimenti :
 - o Controlla il campo destinazione:
 - Se nella tabella di routing è uguale a se stessi (al ricevente) allora non fare nulla.
 - Se nella tabella di routing è diversa da se stessi (al ricevente) si guarda il link. Se:
 - tale link porta al mittente del DV allora non si guarda il costo ma forzatamente si incrementa il costo sempre incrementato del costo del link. (raramente si usa, ma genera il problema di *counting infinity*).
 - Tale link non porta al mittente si guarda il costo attuale e se è maggiore di quello nuovo ricevuto + costo collegamento allora lo scambi con quello nuovo.

NB. Quando un router riceve DV e aggiorna la propria tabella di routing, se:

- o L'aggiornamento ha prodotto nuovi risultati si manda TUTTA la tabella di routing sottoforma di DV
- o Se non c'è stato aggiornamento non si manda nulla.

Esempio

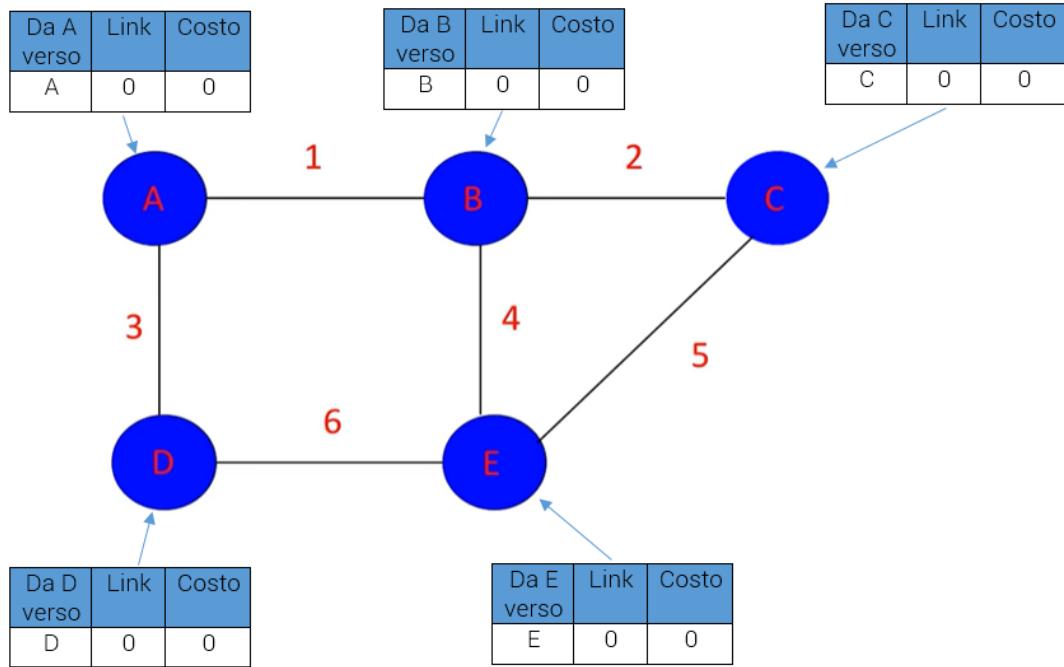


NB. Il costo di ogni link è 1. Quello negli archi è solo il numero link che collega i due router.

NB. Quando quindi un router inserirà una entry nella propria tabella di routing e dirò "incrementa di 1" lo dirò

perché intendo che il costo di ogni collegamento è 1"

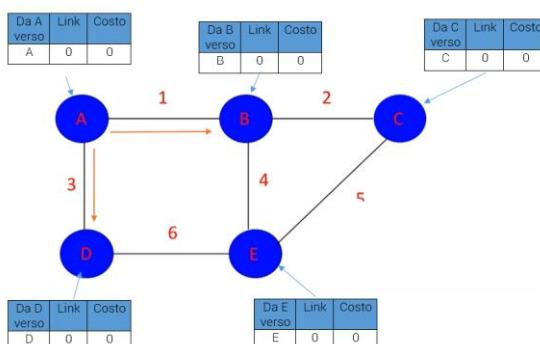
All'inizio tutti i router all'attivazione non hanno nessuna conoscenza topologica ma sanno solo a chi sono collegati e hanno come tabella di routing una tabella contenente la sola informazione su se stessi:



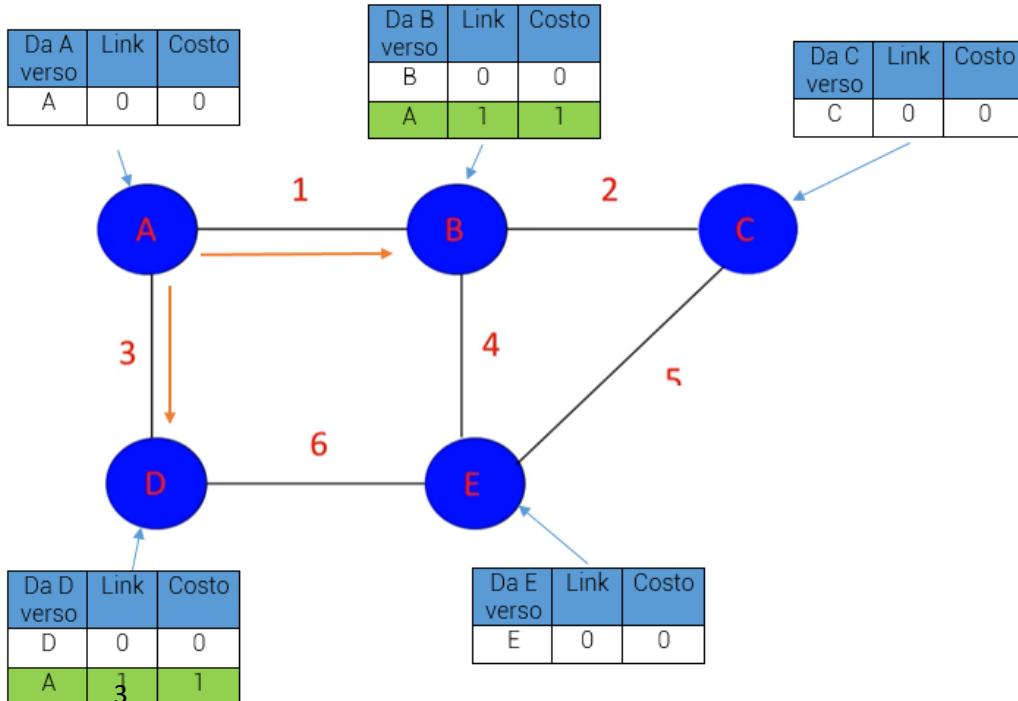
A un certo punto il router A invia il seguente DV ai router adiacenti:

$$DV = [A, 0]$$

per dire ai vicini "della topologia della rete conosco solo me stesso e per raggiungermi ho costo nullo".



Quando B e D ricevono il DV di A leggono la destinazione del DV e si accorgono di non avere A nella propria tabella di routing, quindi inseriscono una new entry con A e con il costo incrementato di 1 (ripeto, incremento di 1 perché il collegamento ha costo 1 per tutti).

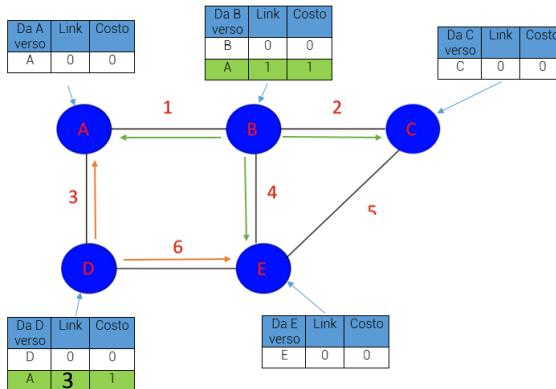


In verde evidenzio le novità.

I nodi B e D, poiché ricevono DV che causano aggiornamenti della tabella di routing, inviano ai propri vicini i loro DV reperendoli dalla tabella di routing appena aggiornata.

B invia verso A,E e C $\xrightarrow{\text{-----}} [B, 0] \text{ e } [A, 1]$
 D invia verso A e E $\xrightarrow{\text{-----}} [D, 0] \text{ e } [A, 1]$

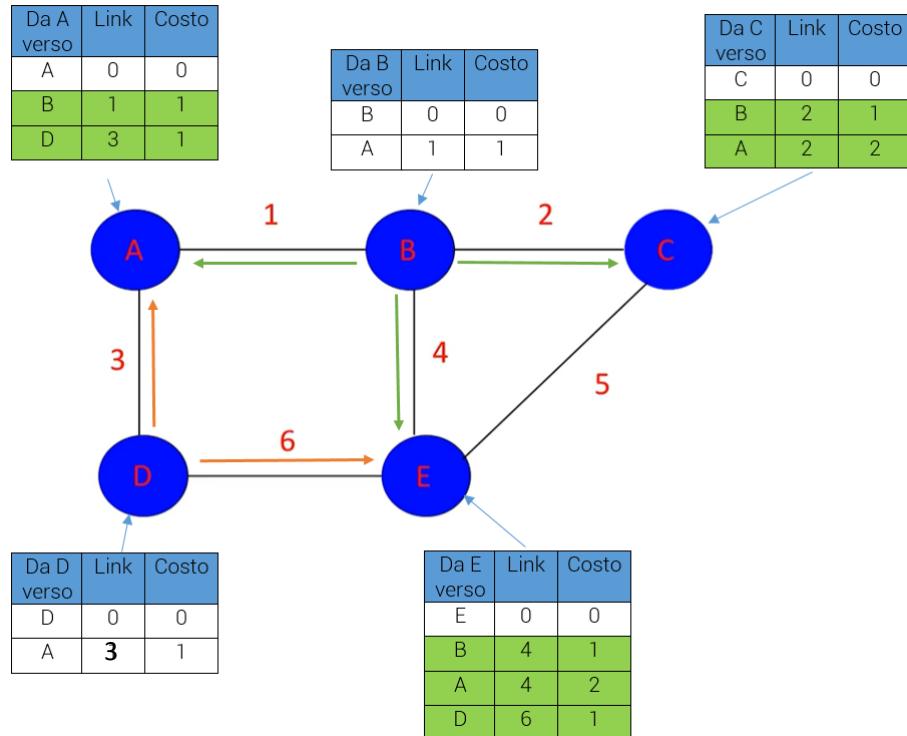
Nota come si invia anche ciò che non si è aggiornato; basta che una sola entry si aggiorna per costringere il router a mandarle tutte.



Tutti i router A,E e C ricevono i DV. Alcuni DV per certi nodi hanno un campo destinazione non presente nella tabella di routing, quindi lo aggiungono direttamente. Qui avvengono due casi particolari:

- 1) A riceve un DV [A,1] da B e da D, e siccome ha costo superiore a quello attuale lo lascio.

- 2) E riceve da B due nuovi campi. Le new entry nella tabella devono comunque avere link uguale.
- 3) Sempre la E riceve due volte [A,1]. Si suppone che lo riceva uno, se poi quando lo riceve l'altro ha costo minore allora lo si scambia. Nel nostro caso i costi sono uguali.

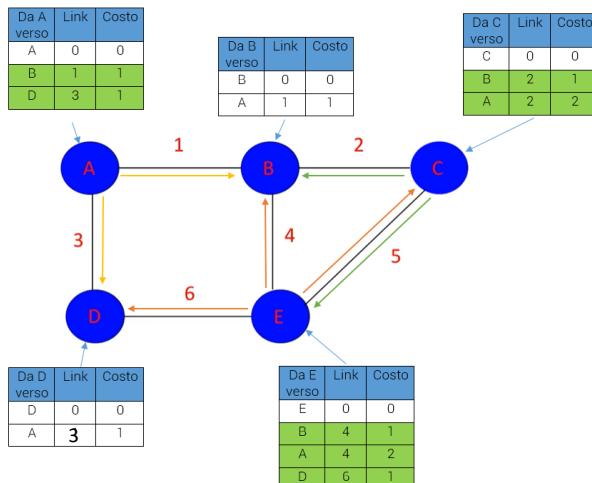


Dato che A,E e C hanno nuovi DV allora devono inviare anche loro i DV ai vicini:

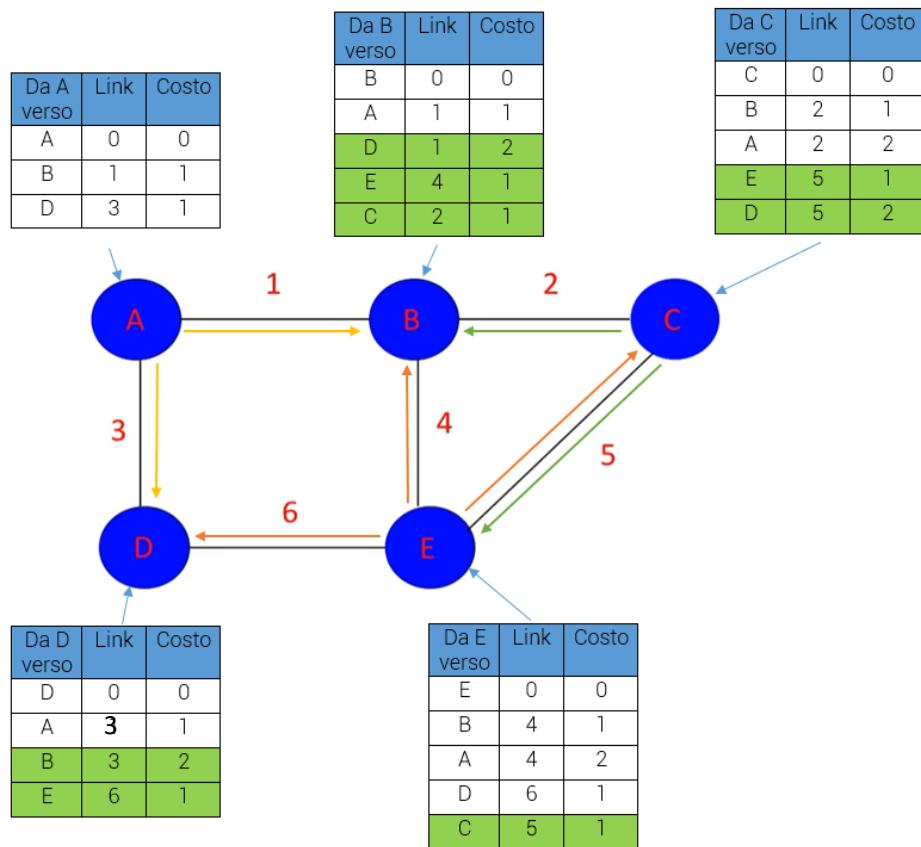
A invia verso D e B $\text{-----} \rightarrow [A, 0], [B, 1] \text{ e } [D, 1]$

E invia verso D,B e C $\text{-----} \rightarrow [E, 0], [B, 1], [A, 2] \text{ e } [D, 1]$

C invia verso B ed E $\text{-----} \rightarrow [C, 0], [B, 1] \text{ e } [A, 2]$



I router B, D, E e C ricevono i DV e aggiornano le loro tabelle di routing:



Sia B,D,E che C hanno aggiornato le proprie tabelle di routing quindi procedono a inviare i loro DV:

B invia verso A, E e C -----> [B, 0], [A, 1], [D, 2], [E, 1], [C, 1]

D invia verso A ed E -----> [D, 0], [A, 1], [B, 2] e [E, 1]

E invia verso D, B e C -----> [E, 0], [B, 1], [A, 2], [D, 1], [C, 1]

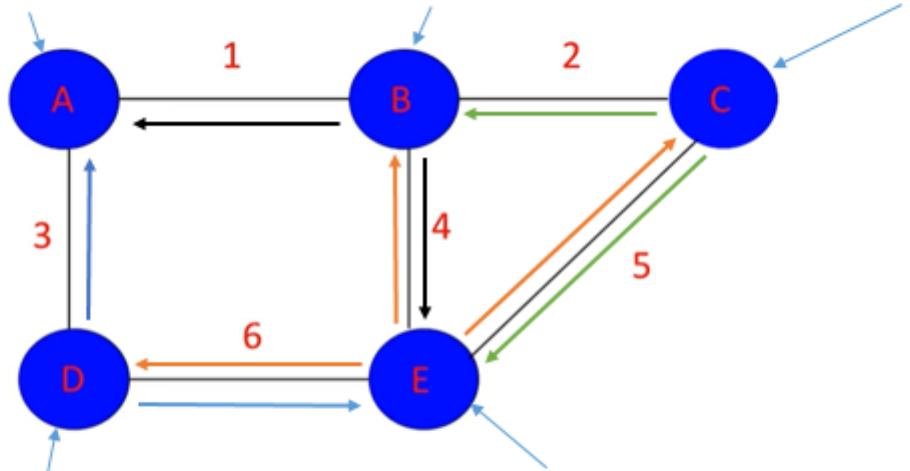
C invia verso B ed E -----> [C, 0], [B, 1], [A, 2], [E, 1] e [D, 2]

I router ricevono i pacchetti e aggiornano le proprie tabelle di routing:

Da A verso	Link	Costo
A	0	0
B	1	1
D	3	1
E	1	2
C	1	2

Da B verso	Link	Costo
B	0	0
A	1	1
D	1	2
E	4	1
C	2	1

Da C verso	Link	Costo
C	0	0
B	2	1
A	2	2
E	5	1
D	5	2



Da D verso	Link	Costo
D	0	0
A	3	1
B	3	2
E	6	1
C	6	2

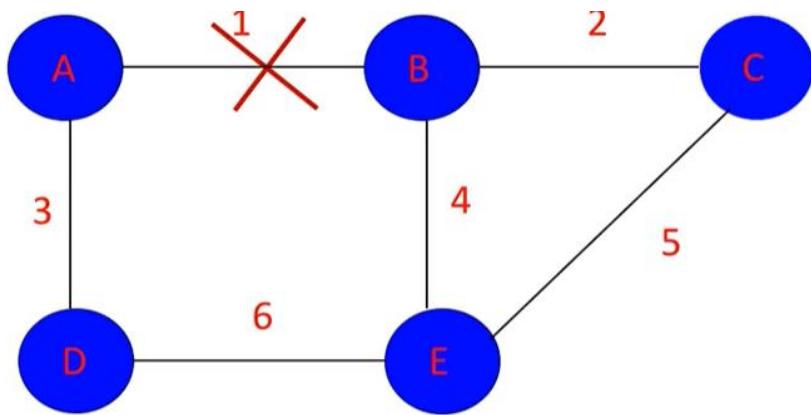
Da E verso	Link	Costo
E	0	0
B	4	1
A	4	2
D	6	1
C	5	1

Come si può notare dalla figura di sopra solo i router A e D hanno risentito del ricalcolo, saranno quindi loro a inviare ID...In realtà però non succederà nessun effetto perché nessuno aggiornerà niente.

Vediamo che l'algoritmo **converge** e si conclude in modo autonomo senza alcun messaggio di "fine operazione". Quando periodicamente qualcuno invierà DV allora se ci saranno state modifiche alla rete si riprenderà quanto fatto sopra.

AGGIORNAMENTO TABELLA DI ROUTING A FRONTE DEL CAMPO DI TOPOLOGIA

Supponiamo che la topologia cambi:



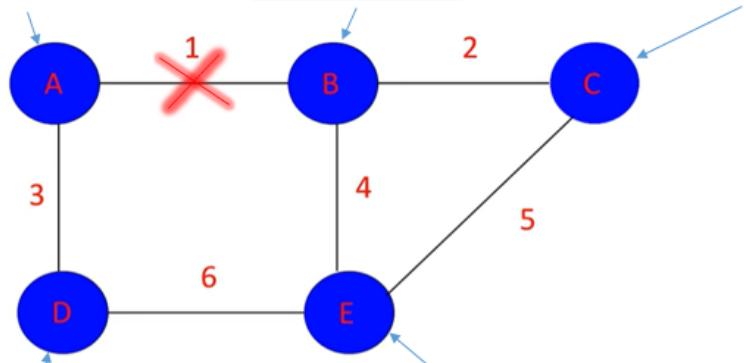
Come fanno i router ad aggiornare la propria tabella di routing in modo che sia consistente con la topologia di rete attuale?

Sappiamo che *periodicamente* i router mandano DV per verificare che la rete non sia cambiata. Se è E a mandare i suoi DV sicuramente non genera alcun cambiamento ai router D, B e C. Se invece è il router A o B a mandare DV allora succede qualcosa. Il link 1 non esiste più a causa dell'abbattimento. Questo evento è marcato da A e B in questo modo:

Da A verso	Link	Costo
A	0	0
B	1	INF
D	3	1
E	1	INF
C	1	INF

Da B verso	Link	Costo
B	0	0
A	1	INF
D	1	INF
E	4	1
C	2	1

Da C verso	Link	Costo
C	0	0
B	2	1
A	2	2
E	5	1
D	5	2



Siccome il link 1 non esiste allora equivale a mettere come costo un costo infinito, da qui Bellman Ford.

Da D verso	Link	Costo
D	0	0
A	3	1
B	3	2
E	6	1
C	6	2

Da E verso	Link	Costo
E	0	0
B	4	1
A	4	2
D	6	1
C	5	1

Cambiando le loro tabelle di routing trasmettono pertanto i nuovi DV che risultano essere:

A invia verso D $\xrightarrow{\text{-----}} [A, 0], [B, \text{INF}], [D, 1], [E, \text{INF}] \text{ e } [C, \text{INF}]$

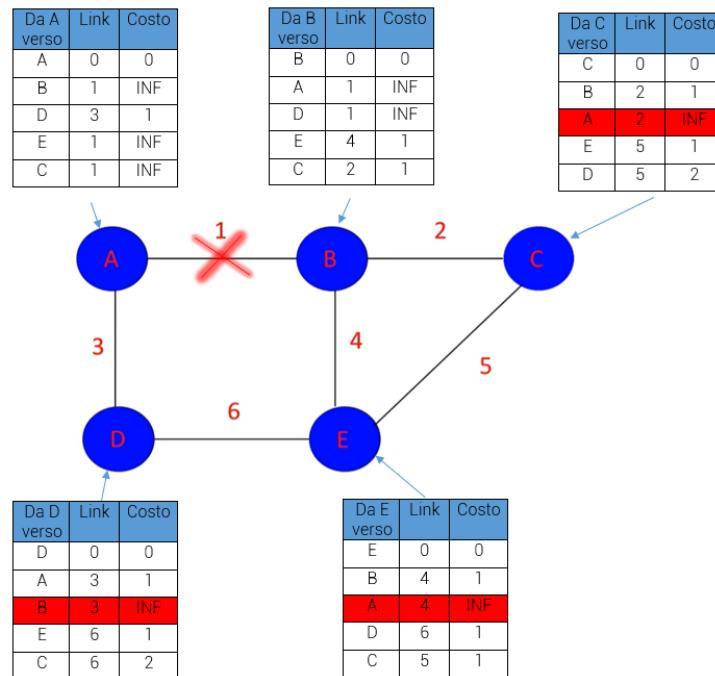
B invia verso E e C $\xrightarrow{\text{-----}} [B, 0], [A, \text{INF}], [D, \text{INF}], [E, 1], [C, 1]$

I router D, E e C ricevono i DV. A rigor di logica non si dovrebbero aggiornare perché per ciascuna destinazione che hanno già presente nella tabella di routing hanno il costo relativo minore di infinito. Però qui subentra un'altra *regola*:

"Se un nodo A riceve per una certa destinazione DEST un valore INF da B, allora anche lui la metterà a INF se e solo se per raggiungere quella DEST ha bisogno di B".

Ecco cosa succede.

- Il router D riceve i DV che di novità interessano solo le destinazioni B,C ed E. Il loro costo è infinito, questo significa che il link è caduto. Allora controlla nella propria tabella di routing qual è il link di collegamento in B,C ed E.
 - Da D verso B prende il link 3. Questo è collegato ad A che gli ha mandato il DV di segnalazione del link mancante. Quindi lo marchio.
 - Da D verso C (così come da D verso E) abbiamo come link 6 che non è quello che ci lega con A che ha mandato la segnalazione (questo vuol dire che D trova un altro modo per raggiungere C ed E). Quindi non lo marchio.
- Il router E riceve i DV e sulle novità A e D controlla:
 - Da E verso A percorre il link 4 che lo collega a B che gli ha inviato la segnalazione. Quindi lo marchio.
 - Da E verso D percorre il link 6 che non lo collega con B. Quindi non lo marchio.
- Il router C riceve i DV e sulle novità A e D controlla:
 - Da C verso A percorre il link 2 che lo collega a B che gli ha inviato la segnalazione. Quindi lo marchio.
 - Da C verso D percorre il link 5 che non lo collega con B. Quindi non lo marchio.



Vedendosi cambiate anche loro le proprie tabelle di routing allora mandano i DV:

D invia verso A ed E -----> [D, 0], [A, 1], [B, INF], [E, 1] e [C, 2]

E invia verso D,B e C -----> [E, 0], [B, 1], [A, INF], [D, 1] e [C, 1]

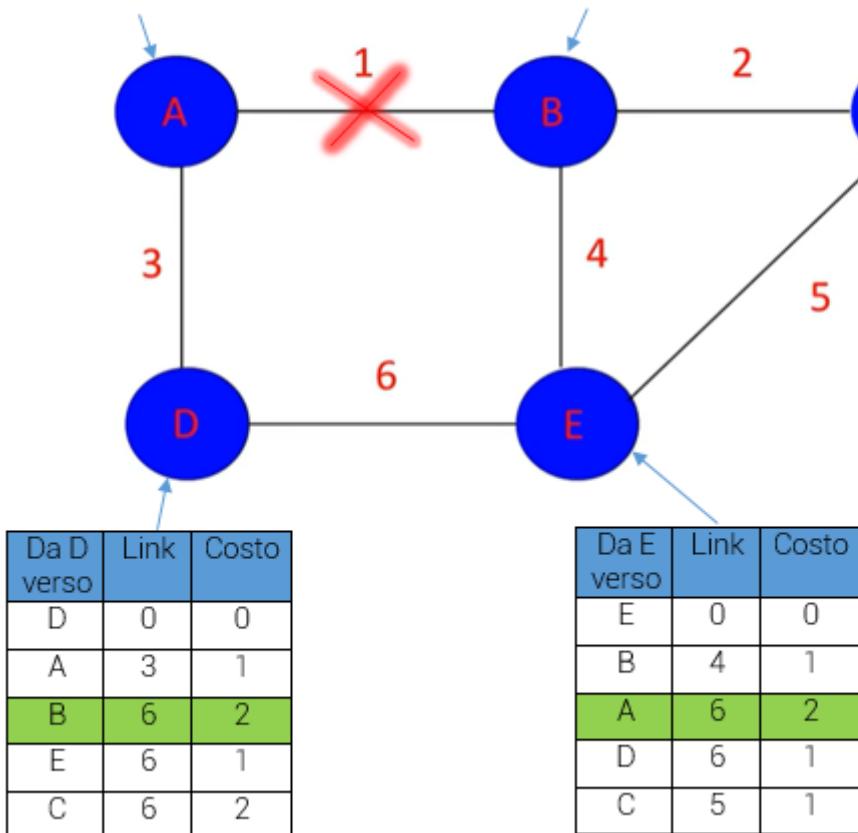
C invia verso B ed E -----> [C, 0], [B, 2], [A, INF], [E, 1] e [D, 2]

Adesso l'aggiornamento segue la regola sopra ma più che altro le regole sempre usate nel confronto dei costi. (ho già contato la risposta in DV dei router)

Da A verso	Link	Costo
A	0	0
B	3	3
D	3	1
E	3	2
C	3	3

Da B verso	Link	Costo
B	0	0
A	4	3
D	4	2
E	4	1
C	2	1

Da C verso	Link	Costo
C	0	0
B	2	1
A	5	3
E	5	1
D	5	2



Ricordati sempre che quando un router ha una entry INF e genera l'effetto domino sopra visto, in ogni caso, tutte le tabelle di tutti i router dovranno in qualche modo togliersi quell'inf. Se pertanto resta qualche inf allora è errore perché ci deve sempre essere un percorso (minimo) da un router a un altro. Per esempio prima dell'eliminazione del link il router B prendeva dal link 1 per arrivare ad A. Adesso prende dal link 4 che poi prenderà dal link 6 che poi prenderà dal link 3. Abbiamo visto come anche in questo caso l'algoritmo converge.

CARATTERISTICHE

Ha il grosso **vantaggio** dell'essere semplice nell'implementazione.

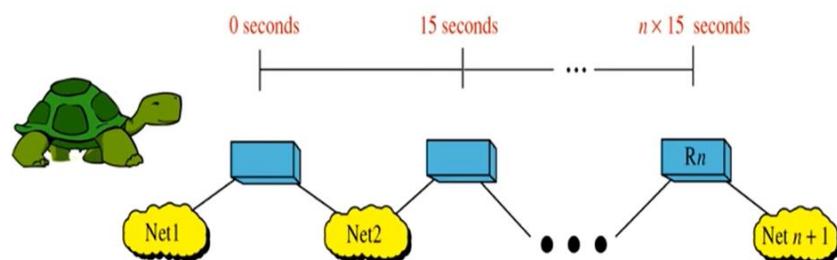
Ha invece diversi **svantaggi**:

- Lentezza nella convergenza
- Se anche N-1 nodi sono velocissimi, ne basta anche solo 1 lento che tutta la velocità della convergenza dipende da lui

- Anche per un piccolo cambiamento si possono avere moti cicli
- Se ad un router arrivano due pacchetti semanticamente legati tra loro ma prima che arrivi il secondo aggiorni la propria tabella allora potrebbe indirizzare verso la stessa direzione ma in tempi e percorsi diversi che potrebbero creare problemi sulla consegna in sequenza.
- Ci sono casi detti *counting to infinity* che causano la non convergenza.

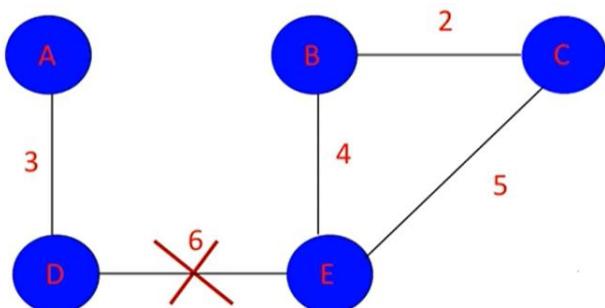
VELOCITA' DI CONVERGENTE

Uno degli svantaggi abbiamo detto che è la velocità di convergenza. Infatti il tempo di convergenza aumenta allumentare dei nodi in modo proporzionale.



Se infatti nel caso peggiore ogni router è collegato sequenzialmente con gli altri e il tempo per raggiungere il next hop è di 15 secondi allora un aggiornamento in un tempo $n \times 15$ sec.

COUNTING TO INFINITY



Supponiamo che si guasti il link 6. Abbiamo in questo caso due reti distinte non collegate tra loro. In questo caso le tabelle di routing dovrebbero mostrarmi due reti distinte.

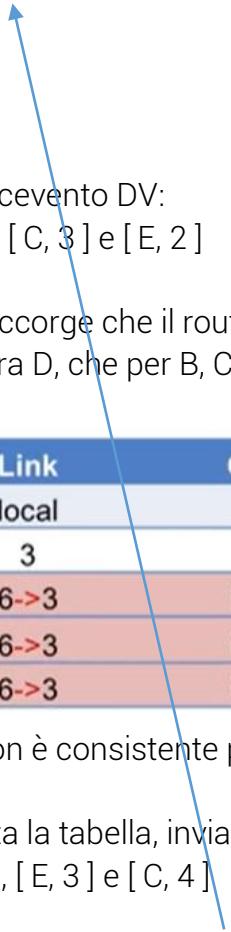
Il nodo D aggiorna la sua tabella:

Da D verso	Link	Costo
D	local	0
A	3	1
B	6	2->inf
E	6	1->inf
C	6	2->inf

Possono succedere due casi:

- Il router D invia ad A il suo DV per primo. A questo punto A aggiorna la sua tabella di routing perché si accorge che il link 6 non c'è più. Questo è il caso in cui si ha convergenza, quindi nessun problema.
- Il router A, come tutti gli altri router, invia il suo segnale periodico a D prima che questo gli mandi il suo DV.

Da A verso	Link	Costo
A	0	0
B	3	3
D	3	1
E	3	2
C	3	3



Ecco cosa gli può arrivare a D ricevento DV:

$$DV = [A, 0], [B, 3], [D, 1], [C, 3] \text{ e } [E, 2]$$

Se D legge questi DV si accorge che il router A può raggiungere il nodo B (così come anche C ed E). Allora D, che per B, C ed E ha costo infinito, aggiorna la sua tabella si routing:

Da D verso	Link	Costo
D	local	0
A	3	1
B	6->3	inf->4
E	6->3	inf->3
C	6->3	inf->4

E' chiaro che la tabella non è consistente perché non esiste nessun percorso per raggiugere B,E e C.

Comunque sia, aggiornata la tabella, invia i DV ad A.

$$DV = [D, 0], [A, 1], [B, 4], [E, 3] \text{ e } [C, 4]$$

Per ognuna delle mete A possiede costi minori quindi non dovrebbe cambiare nulla. Però sa che per B,D,E e C ha sì quei costi che sono minori, ma li possiede grazie a D. Se D raggiunge per esempio B con costo 4 come fa a permettere ad A di raggiungerlo con costo 3? Per questo motivo la regola dice che se si può raggiungere una destinazione tramite un link che passa per un certo nodo e quello stesso nodo ti sta inviando un DV su quella destinazione, allora devi aggiornare la tua tabella. In questo caso la tabella si aggiorna B,E e C.

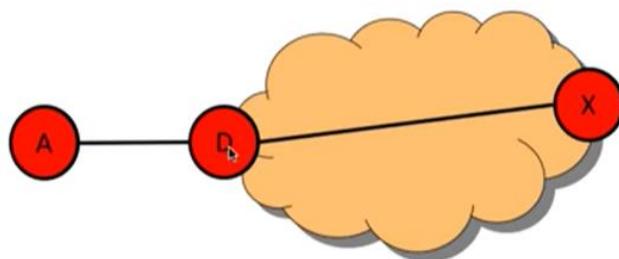
Da D verso	Link	Costo
D	0	0
A	3	1
B	3	6
E	3	5
C	3	6

D rimanda DV ad A con i nuovi aggiornamenti. Capiamo bene che si crea un loop che vede ad ogni iterazione un aumento di 2 dei costi. **Non c'è convergenza** perché dopo tot di iterazioni questi valori andranno verso l'infinito.

RIMEDI AL COUNTING TO INFINITY

Quali sono i rimedi?

Un modo è detto **split horizon semplice**.



Se ci facciamo il problema del counting to infinity nasce perché nell'esempio un nodo A dice a un nodo D come raggiungere X tramite il nodo stesso D. Nell'esempio del counting to infinity fatto prima, il ciclo infinito si crea perché A dice a D che è in grado di raggiungere B,C ed E tramite D stesso.

La soluzione è che A non deve comunicare a D quelle destinazioni che riesce a raggiungere grazie a D stesso. Quindi il Distance Vector che utilizza lo Split-Horizon utilizza DV leggermente modificati. Prima un router mandava DV identici per ogni router adiacente; adesso qui la struttura della lista dei DV mandati a un router dipende dal router adiacente. Infatti se certe righe contengono destinazioni raggiungibili grazie a un nodo adiacente, allora sicuramente il DV mandatogli sarà differente da quello mandato a un altro router adiacente che però non permette di raggiungere quella destinazione grazie a lui.

Nel problema del counting to infinity avremmo che A invierebbe per D solo il DV:

$$DV = [A,0] \text{ e } [D,1]$$

Omettendo B,C ed E (raggiungibili grazie a D) che causerebbero il loop.

Un'altra versione dello split horizon è nota come **split horizon tramite Poison Reverse** in cui si hanno ancora DV specifiche per ogni nodo adiacente ma stavolta non si omettono quelle destinazioni raggiungibili grazie a quel nodo adiacente, piuttosto si marcano come INF i costi di quelle righe quando le si mandano.

Perché conviene utilizzare lo split horizon tramite poison reverse?

Quando i router si mandano questi DV i ricevitori non fanno altro che allocare memoria (lo vedremo nei protocolli di routing). Se il ricevitore sa già quant'è la dimensione di ogni singolo DV allora gli verrà facile allocare una memoria

prestabilità, altrimenti se come nello split horizon semplice i pacchetti possono avere dimensioni differenti, allora il ricevitore dinamicamente avrà difficoltà ad allocare memoria perché ogni volta ne dovrà riservare una diversa.

ALGORITMO Link State (LS)

Ogni router conosce la topologia completa (quindi anche i collegamenti e i relativi costi) della rete globale e quindi sono disponibili in input all'algoritmo.

Ma come fanno ad avere questa conoscenza globale?

Ogni nodo inizialmente non sa granché, pertanto invia uno *stato dei suoi collegamenti* a tutti gli altri nodi della rete. Questi pacchetti contengono le identità e i costi dei collegamenti connessi al router che li invia e sono chiamati **LSP** ossia **Link State Packet**. Di conseguenza ogni router può memorizzare localmente tutti questi pacchetti in un **database** e sulla base di quanto ricevuto crearsi localmente una mappa della rete e far girare l'algoritmo localmente su quella rete creata senza interagire con gli altri.

La seconda differenza è che in Distance Vector si invia tale informazione solo agli adiacenti, mentre qui si invia in **flooding**, un tipo di broadcast.

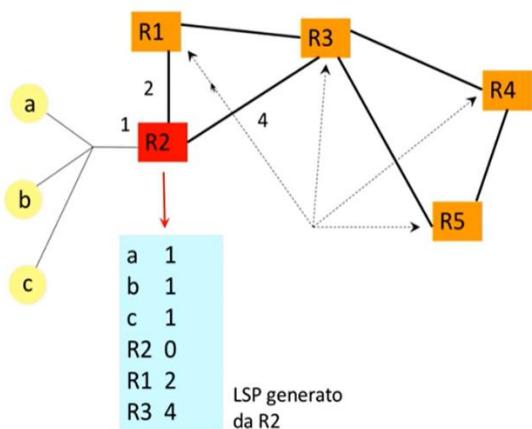
Terza differenza è che il Distance Vector si basa sull'algoritmo sui grafi di Bellman-Ford, mentre questo si basa su **Dijkstra**.

I **vantaggi** sono che è più flessibile in quanto ogni router ha già tutto quello che ci serve localmente; inoltre non è necessario inviare LSP periodicamente ma solo quando avviene un cambiamento per aggiornare la mappa; infine tutti i nodi vengono subito informati dei cambiamenti.

Gli **svantaggi** sono che è necessario un protocollo dedicato a "scoprire" qual è il proprio stato; è necessario l'utilizzo del broadcast ma con un riscontro in quanto devo avere la certezza che tutti i router hanno ricevuto l'LSP altrimenti avrebbero una mappa non corretta. Il numero dei messaggi sono circa $N \times E$ con E numero archi.

La tecnica di invio degli LSP è di broadcast, ma di un tipo particolare detto **flooding** in cui quando si riinvia un messaggio ricevuto, invece che farlo verso tutti i *nodi delle uscite* lo si fa verso tutti quelli in uscita ma meno quello da cui l'ho ricevuto.

Questo non riesce comunque ad evitare i loop. Infatti consideriamo il seguente esempio:



Supponiamo che R2 invia il proprio LSP verso R1 e R3 che sono i suoi collegamenti in uscita (a,b e c formano una LAN, non sono router). Allora R1 e R3 inoltreranno l'informazione. R1 la inoltra solo a R3 così che, evitando di inviarla a R1, si evita il loop (grazie al floating). R3 non l'invia anche lui a R2, ma a R4,R5 e R1. R1 riceve quindi di nuovo lo stesso pacchetto, però stavolta nulla gli impedisce di inviarlo a R2 perché lo ha ricevuto da R3 quindi risconosce lui come mittente. E' chiaro

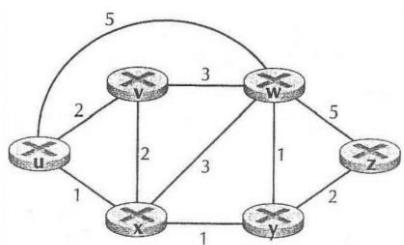
che si ricomincia a loop dall'inizio.

Per risolvere questo problema basta marcare ogni LSP con un **numero di sequenza SN** così che quando un router riceve un LSP lo salva nel database solo se non possiede nessun LSP con quell'SN.

Per sicurezza viene sempre messo un **contatore di hop**, ossia una sorta di time to live, per cui una volta scaduto non viene più inoltrato perché scartato.

Esempio

Dopo che i router hanno ricostruito la mappa tramite inoltro e ricezione di LSP si troveranno un database nel formato:



Da	Verso	Link	Costo	Sequence Number
A	B	1	1	1
A	D	3	1	1
B	A	1	1	1
B	C	2	1	1
B	E	4	1	1
C	B	2	1	1
C	E	5	1	1
D	A	3	1	1
D	E	6	1	1
E	B	4	1	1
E	C	5	1	1
E	D	6	1	1

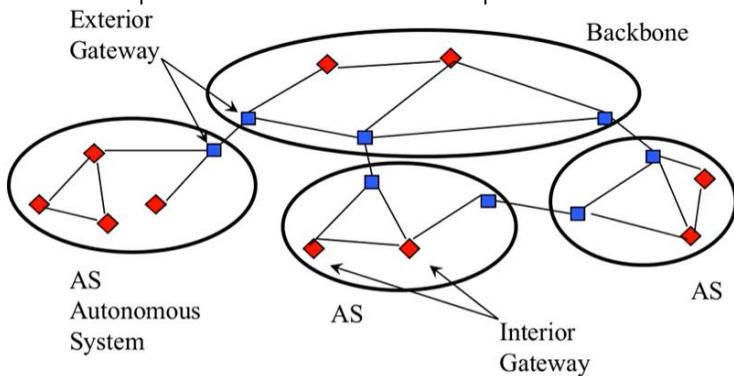
Sarà su questo database che ogni router farà girare l'algoritmo di Dijkstra.

PROTOCOLLI DI INSTRADAMENTO/ROUTING

Vedremo adesso i protocolli di routing che utilizzano ciascuno uno dei due algoritmi di routing visti sopra. C'è da dire una cosa però. I due algoritmi di routing sono infatti bili da applicare nel mondo reale per due motivi:

- **Scalabilità:** Internet è fatta da migliaia e migliaia di host. Un algoritmo come il Distance Vector per esempio ingombrerebbe tutta la rete per troppo tempo e sicuramente non arriverebbe neppure a convergere. Per non pensare agli LSP che sarebbero davvero troppo grandi e inoltre ingolferebbero la rete.
- **Autonomia amministrativa:** se dovessimo davvero far girare un unico algoritmo su tutti i router del mondo sicuramente non si darebbe libertà agli amministratori anche di piccole reti di poter decidere autonomamente dove inoltrare i dati e in che modo (non tutti vogliono per forza usare un algoritmo preciso). Inoltre è anche un motivo di *sicurezza* non voler rivelare la topologia della propria rete agli altri.

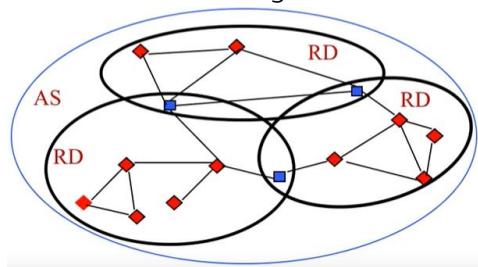
La soluzione è organizzare Internet in una rete di **sistemi autonomi (AS)**, ossia gruppi di router indipendenti nella scelta del protocollo di instradamento dagli altri:



Si viene così a creare una sorta di *gerarchia* in Internet. Qui abbiamo 3 AS e un **backbone**, ossia una *dorsale* che collega i vari AS tra loro. Ogni AS può fare la scelta del proprio protocollo di routing. I router rossi vengono detti **Interior Gateway**, e sono

router che parlano *solo* un protocollo di routing detto **IGP (Interior Gateway Protocol)**, quello deciso dall'AS di cui fanno parte. I router blu invece vengono chiamati **Exterior Gateway** e servono da raccordo con il backbone e quindi il resto della rete e di solito devono parlare *diversi* protocolli di routing detti **EGP (Extern Gateway Protocol)**.

Vedremo che i router blu (detti anche di bordo) useranno un protocollo che non userà nessuno dei due algoritmi di routing.



Inoltre all'interno degli AS possono crearsi altri gruppi IGP chiamati **Routing Domain (RD)** in cui può girare un algoritmo di routing differente dagli altri RD all'interno della stessa AS. Quindi, importante, è che alcuni router di bordo possono anche trovarsi all'interno di ciascuno IGP e pertanto possono trovarsi a tradurre protocolli basati di LS a protocolli basati su DV.

I protocolli usati sugli IGP basati sul *Distance Vector* sono:

- 1) RIP
- 2) IGRP

Mentre quelli basati sul *Link State* sono:

- 1) IS-IS
- 2) OSPF

Mentre i protocolli usati sugli EGP sono basati su un particolare algoritmo di instradamento non ancora visto che si chiama *Path Vectore* vedremo solo:

- 1) BGP

PROTOCOLLO DI ROUTING RIP

E' un protocollo *IGPe* usa l'algoritmo di routing *Distance Vector*. Come metrica, ossia costo del percorso, si usa il *numero di hop*, ossia la distanza da una destinazione corrisponde al numero di router che devo attraversare. Per INF si usa 16 in quanto 16 hop è il limite del protocollo, ossia per arrivare a una destinazione, se non voglio che si usi un certo percorso allora metto 16 (questo significa che in un cammino verso una destinazione sono accettati al massimo 15 router).

I pacchetti di RIP, che sono pacchetti di DV, sono incapsulati in *UDP*. Quindi anche se questi pacchetti sono di livello 3, si comportano come messaggi applicativi, perché se ricordiamo i dati applicativi venivano incapsulati in UDP. Questi segmenti UDP vengono poi incapsulati dentro pacchetti IP con destinazione broadcast 255.255.255.255 (è un *broadcast limitato* alla piccola rete, non vanno fuori).

FORMATO DEI MESSAGGI RIP

Sarebbero il formato dei DV visti nell'algoritmo DV.

Command	Version	Reserved
Family	All 0s	
Network address		
All 0s		
All 0s		
Distance		

Repeated

- **Command:** se 1 indica che il messaggio è di richiesta, 2 se di risposta
- **Versione:** indica la versione di RIP (questa è la v1 ma esiste anche la v2)
- **Family:** indica la famiglia di indirizzi usati. Si potrebbe

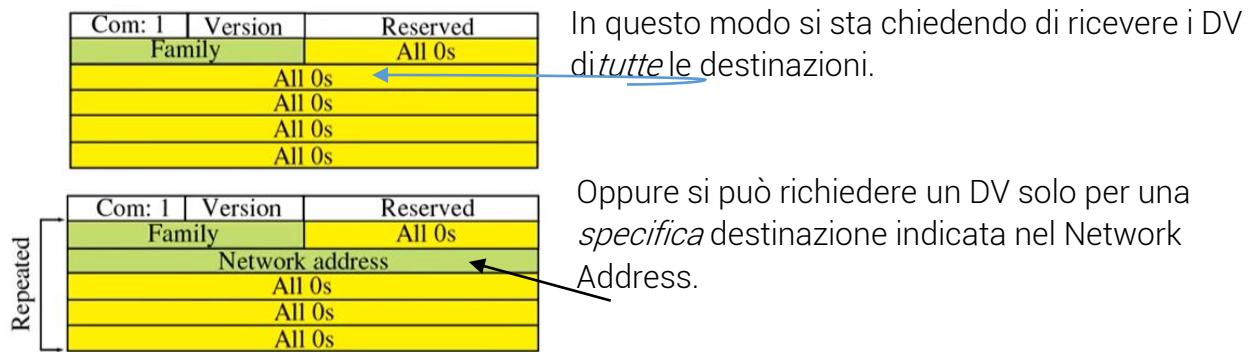
usare RIP per percorsi che non usano indirizzi IP per ciascun nodo ma altro. Noi comunque useremo il valore 2, ossia per indicare indirizzi IP.

- **Network Address e Distance:** sono la coppia DV=[nodo,costo] dove il costo abbiamo detto è misurato in numero di hop.

Esistono due tipi di messaggi: di richiesta e di risposta. Quelli di risposta seguono la logica vista nei DV, ossia vengono inviati ogni 30 secondi e contengono i propri DV.

Questi vengono detti *non stimolati* perché sono mandati alla scadenza di un timer; quelli *stimolati* sono invece gli stessi messaggi (magari su un dominio di destinazione più ristretto) che però vengono mandati solo su richiesta dal primo tipo di messaggio detto di richiesta.

I messaggi di **richiesta** servono per chiedere informazioni topologiche, ossia servono per chiedere ai router di inviargli DV e di solito si usano quando un router viene appena attivato. Esistono due tipi possibili di richieste:

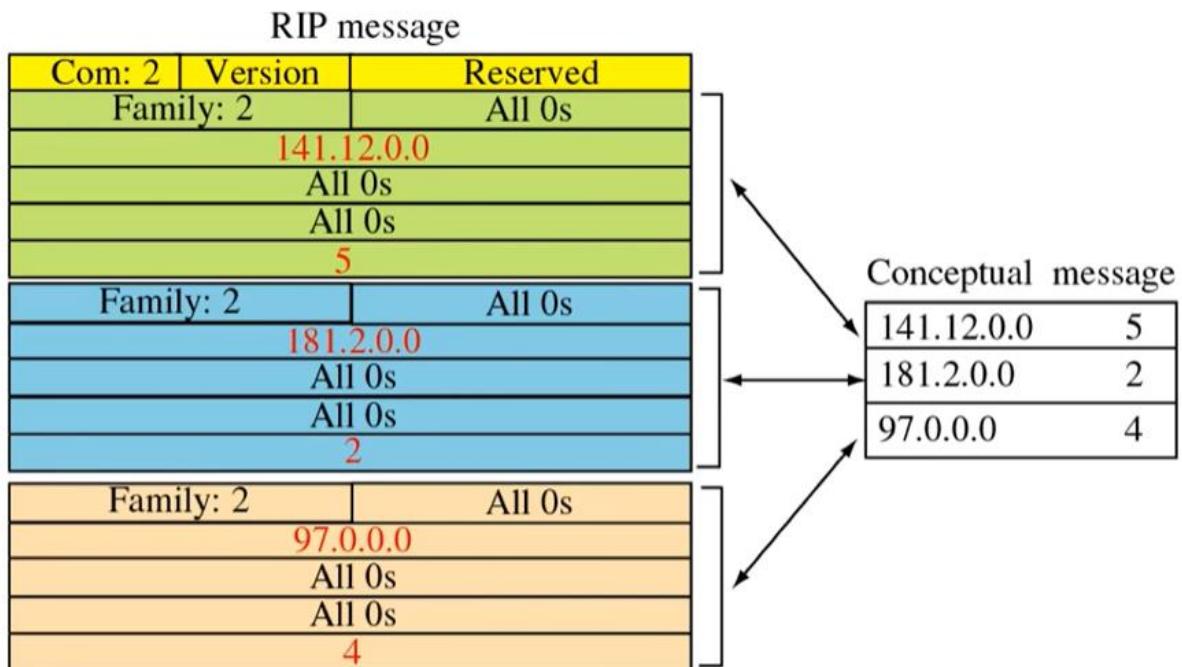


I messaggi di **risposta** contengono la lista dei DV.

Supponiamo che un router voglia mandare il seguente DV:

$$DV = [141.12.0.0, 5], [181.2.0.0, 2], [97.0.0.0, 4]$$

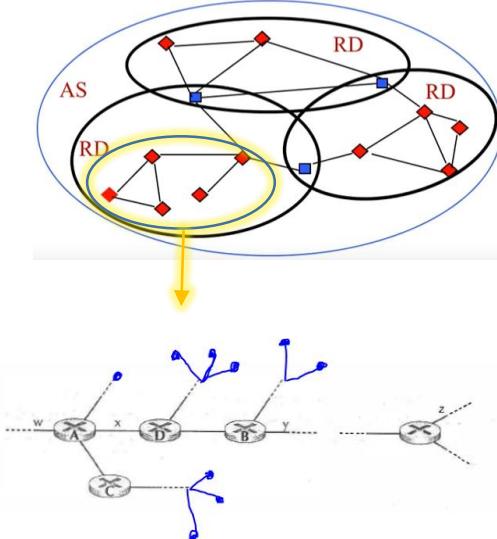
Allora avremmo questo *unico* messaggio di risposta:



Nota però che un singolo pacchetto UDP che incapsula il DV può avere al massimo 25 rotte. Per DV con rotte in numero maggiori di 25 si avranno più pacchetti UDP.

I router adiacenti si scambiano messaggi dopo all'incirca 30 secondi, e se dopo 180 secondi non riceve risposta dal destinatario allora lo considera irraggiungibile, pertanto modifica la propria tabella di routing e propaga l'informazione ai vicini.

Abbiamo detto che i messaggi (di richiesta o risposta) vengono inviati usando UDP come trasporto. Ma RIP è un protocollo di rete e non potrebbe utilizzare un protocollo di trasporto. In realtà abbiamo detto che un router svolge funzioni importanti solo fino al livello 3, per questo si dice che implementa fino al livello 3 della pila protocollare. Ma in realtà in esso esiste un processo applicativo chiamato **routed** che quindi opera a livello di rete e usa UDP per inviare appunto questi messaggi DV. Però siccome routed non ha iterazioni con l'utente e siccome la scelta di UDP non è necessaria e non serve all'utente allora si dice che implementa fino al livello 3.



Questa rappresenta ogni router Internal Gateway e tutti parlano lo stesso protocollo (RIP). Ogni router può essere collegato ad altri router o ad altri host formando delle *sottoreti*. E' questo ciò che si intende per destinazione.

Ogni router ha una **tabella di routing** che incorpora quella di **inoltro**:

Sottorete di destinazione	Router successivo	Numero di hop verso la destinazione
w	A	2
y	B	2
z	B	7
x	—	1
***	***	***

E ogni messaggio DV viene visto con la stessa forma.

PROTOCOLLO DI ROUTING OSPF

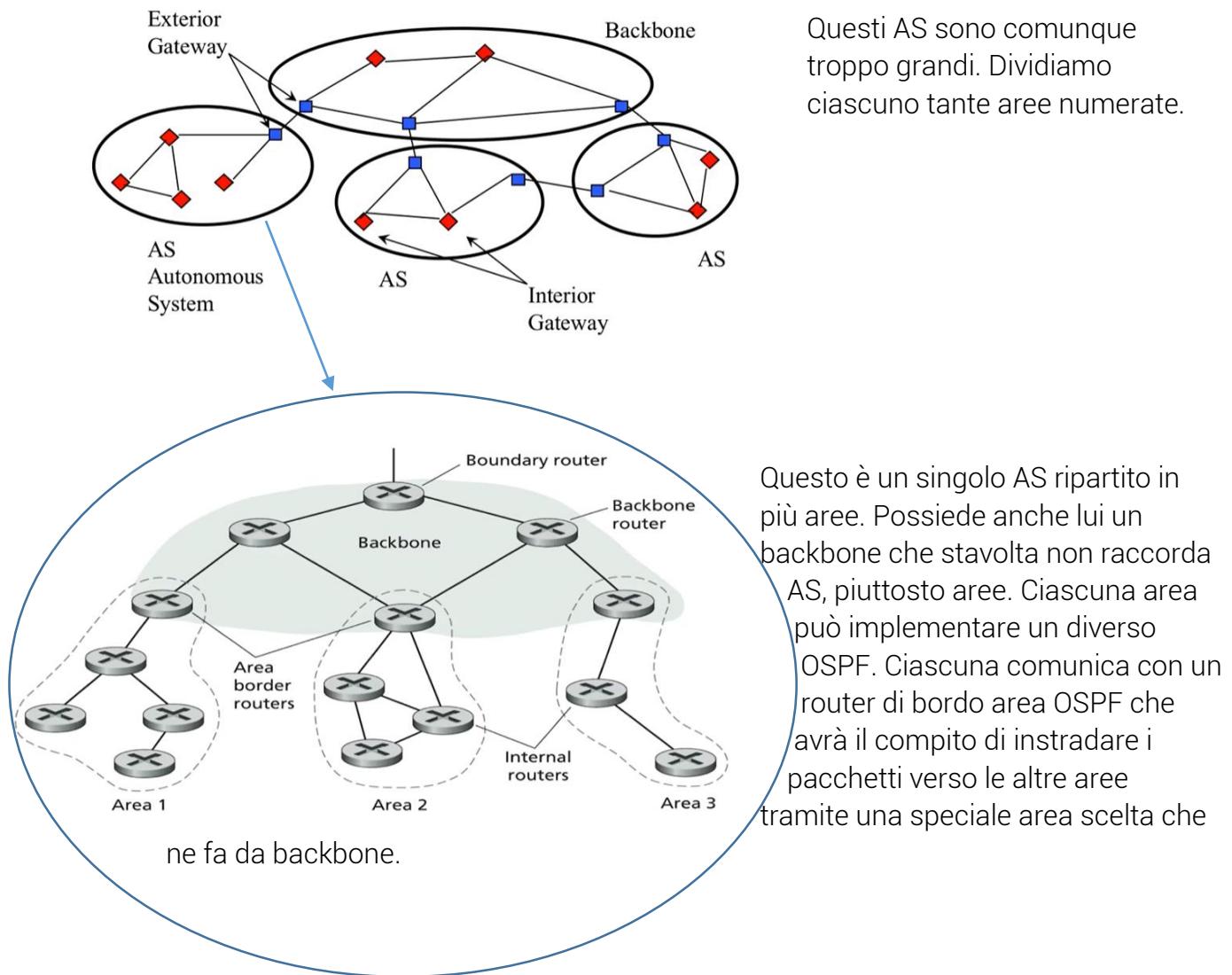
E' un protocollo /GPe usa l'algoritmo di routing *Link State*. Quindi ogni router invia lo stato del link LS a tutti gli altri così che ogni nodo si costruisca localmente il database topologico e su questo grafo fa girare l'algoritmo di Dijkstra.

OSPF non usa UDP ma direttamente IP; questo è un problema perché gli LS devono arrivare per forza ed essere riscontrati. Quindi OSPF implementa lui stesso un meccanismo di

recupero di errore invece che appoggiarsi a TCP e quindi è come un **protocollo di trasporto**, difatti come TCP e UDP ha anche lui il suo codice di protocollo che è 89.

Oltre al servizio di instradamento è in grado di supportare il **routing gerarchico**. Cosa significa? Siccome Internet è costituita da molti AS, tavolta abbastanza grandi, OSP offre la possibilità di suddividere ciascun AS in aree numerate e un area di backbone.

Consideriamo la gerarchia vista prima:



Tutti gli **Interior Gateway** di ogni area hanno una conoscenza dettagliata dell'area in cui stanno e anche una conoscenza sintetica di ciò che sta fuori, ossia vedono il **router di bordo area** (una sorta di External Gateway) come direttamente collegato a *tutte le destinazioni*.

Mentre in RIP ci stavano solo pacchetti di invio e risposta, in OSPF abbiamo diversi pacchetti:

- **Hello:** utili per la scoperta del "vicinato"
- **Link State Update:** utili a ogni router per rappresentare i messaggi LS
- **Link State Request:** utili a mandare richieste verso una rotta specifica

- **Link State Acknowledgement:** utili per implementare il meccanismo di recupero degli errori detto prima.

Inoltre ogni messaggio può essere autenticato così da garantirne la **sicurezza** in caso di invio di messaggi da parte di altri router maligni. Se poi esistono **più percorsi con lo stesso costo** OSPF permette anche la scelta di questi percorsi alternativi ma dallo stesso costo, così da far fluire il traffico verso la stessa direzione non tutto in un unico percorso. Infine permette piena libertà agli amministratori di rete di decidere il **costo** per link.

PROTOCOLLO DI ROUTING BGP

E' un protocollo che non è basato né sul DV né sull'LS. E' un protocollo **EGP** quindi gira all'interno della *core network* e serve all'instradamento tra AS. Non uso DV o LS perché il core network è troppo grande e possono accadere i problemi descritti prima. Per questo motivo il BGP si basa su un *algoritmo di routing* tutto suo chiamato **Path Vector**.

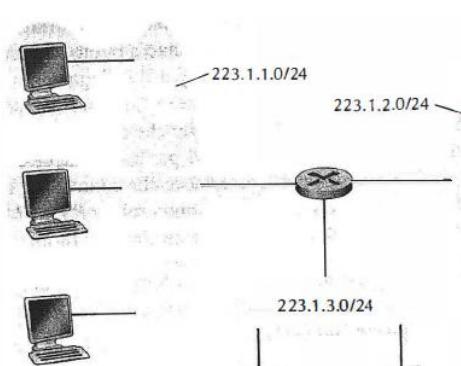
Il BGP serve solo a far sì che le sottoreti di un AS possano comunicare a tutti gli altri la loro presenza in rete. Quindi scopo dei BGP peer (quelli che fanno da raccordo con altri AS) è quello di tenere salvata una lista del tipo:

RETE	ROUTER SUCCESSIVO	PERCORSO
N01	R01	AS2,AS5,AS7,AS12
N02	R07	AS4,AS13,AS6
N03	R09	AS11,AS12,AS8,AS6
...

Questo si legge: "per raggiungere una sottorete N01 (che magari neppure appartiene al router BGP es. 112.0.34.0/23) devo usare il router R01 e il percorso che faccio passare per gli AS di identificativo 2,5,7 e 12."

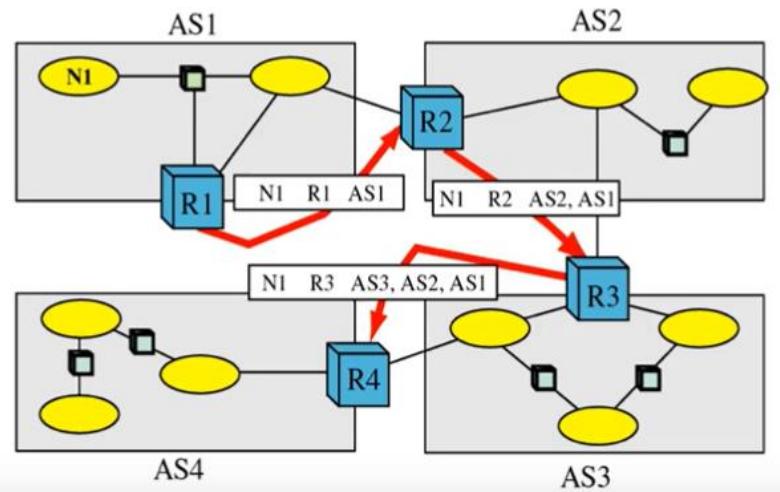
Quelli sopra altro non sono che i *PathVector* inviati tra i router BGP per specificare come raggiungere una certa sottorete in Internet. Notiamo come ogni AS viene quindi identificato tramite un **numero di sistema autonomo** che è univoco, che come gli indirizzi IP vengono

assegnati da ICANN. Supponiamo di aver creato una piccola azienda con un certo numero di server (computer) che vogliamo siano raggiungibili da tutti. Allora contattiamo un ISP locale che ci fornirà un router al quale collegheremo tutto. Questo router di controllo sarà collegato a un altro router nell'ISP locale. L'ISP ci fornirà anche un blocco di indirizzi IP, per esempio un blocco /24 indica che i primi 24 bit identificano l'indirizzo della sottorete e quindi restano solo 8 bit di spiazzamento posso collegare al massimo 256 dispositivi (quindi



fondamentalmente l'ISP mi sta fornendo 256 indirizzi IP). Quindi distribuirò questi indirizzi IP alle varie entità come web server, DNS o altro. Affinchè però effettivamente la mia sottorete e quindi in ultimo i miei server siano visibili e accessibili in Internet, oltre ai miei settaggi interni, devo far sì che tutti i router conoscano il mio prefisso (sarebbe N01 dell'esempio), ossia l'intervallo di indirizzi IP che mi competono. Questo lo fa l'ISP locale stesso tramite BGP. Ossia userà BGP per pubblicizzare questo prefisso anche ad altri BGP collegati a lui che a loro volta faranno lo stesso.

Quando però un router annuncia un nuovo prefisso durante una sessione BGP(vedi dopo) include anche un certo numero di **attributi BGP**. Un prefisso+attributi è detto **rotta**. Di conseguenza i peer BGP si scambiano annunci di rotte.



è detta **sessione BGP**. In particolare se un BGP peers comunica con un router interno al proprio AS allora si dice **sessione iBGP**, altrimenti se comunica col **BGP peers** di un altro AS si dice **sessione eBGP**.

Ciò che si scambiano è un *Path Vector* nella forma:

SOTTORETE	ROUTER SUCCESSIVO	PERCORSO
-----------	-------------------	----------

che abbiamo già visto prima. N1 vuole dire al mondo intero di esistere. Pertanto apre una sessione iBGP con R1, il quale invia a R2, aprendo una sessione eBGP, quel patch vector che R2 memorizzerà proprio in quella forma a dire: "se voglio raggiungere N1 devo prendere dal router R1 passando un solo sistema autonomo AS1". Di contro R2 lo dirà a R3 inviandolo nel formato in figura che R3 memorizzerà in quella forma a dire: "se voglio raggiungere la sottorete N1 devo andare nel router R2 e passerò per due sistemi autonomi AS2-AS1".

Vediamo come il costo non è più un peso, piuttosto il percorso da fare. Questo attributo viene detto **AS_PATH**. Il router BGP da prendere è indicato dall'attributo **NEXT_HOP**. In realtà esistono altri attributi (qui non evidenziati) che sarebbero l'**ORIGIN**, che indica il protocollo RIP,OSPF o altri, da cui proviene l'informazione.

Quindi per ogni messaggio che un router BGP vuole inviare a un altro router BGP si deve sempre aprire una connessione TCP e i messaggi (path vector) sono distinti in:

- **OPEN**: per aprire una connessione TCP e gestire l'autenticazione reciproca dei router

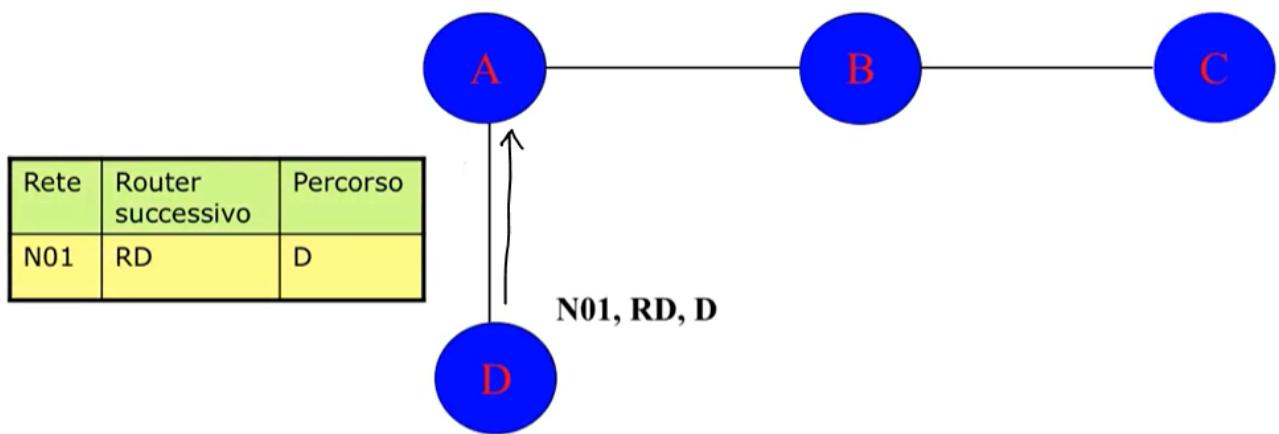
Supponiamo di avere 4 AS che tra loro parlano BGP, ossia che i router di bordo (R1,R2,R3,R4) possono aprire una connessione TCP con gli altri sulla porta 179. Un flusso BGP tipo potrebbe essere un apertura TCP da R1 a R2, una da R2 a R3 e un'altra da R3 a R4.

i router R1,R2,R3 e R4 sono detti **BGP peers** e la connessione TCP

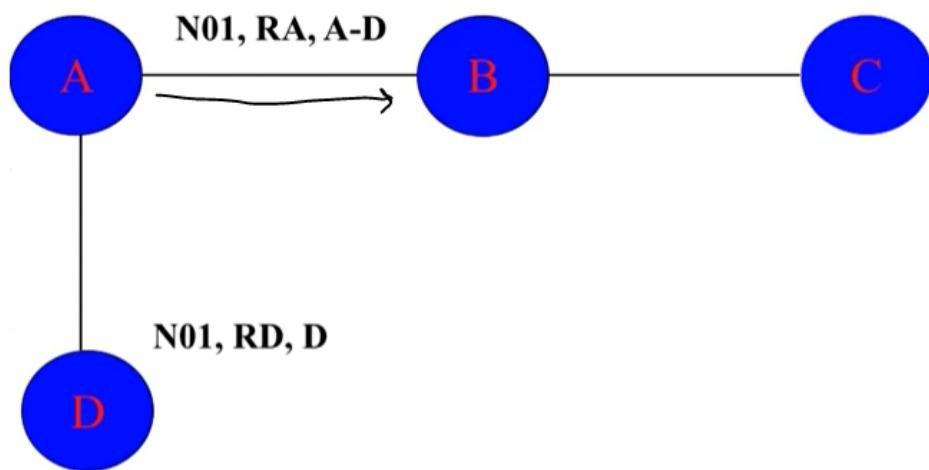
- **UPDATE:** per annunciare una nuova rotta
- **KEEPALIVE:** per non far cadere la connessione TCP istaurata qualora manchino messaggi di UPDATE.
- **NOTIFICATION:** per notificare errori o chiudere la connessione.

E' importante capire che mentre nell'DV e nell'LS è chiaro il criterio (scelgo il cammino minimo), il BGP invece non definisce nessun criterio, piuttosto offre solo un'informazione sul percorso per raggiungere una certa sottorete. Sarà poi dal BGP in poi la stessa AS a implementare un suo routing. Inoltre ogni BGP può decidere se droppare o meno i path vector come nell'esempio:

Supponiamo che in blu siano raffigurati gli AS. Un router BGP appartenente a un sistema autonomo D vuole notificare ad A una nuova sottorete N01. Pertanto apre una connessione TCP col router BGP di A e gli invia il seguente path vector.



Il router BGP di A memorizza questo path vector e lo leggerà come: "se voglio raggiungere la sottorete N01 dovrò prendere un router BGP di nome RD e passare per un solo sistema autonomo di identificativo D". Fatto questo ha il dovere di propagare l'informazione.

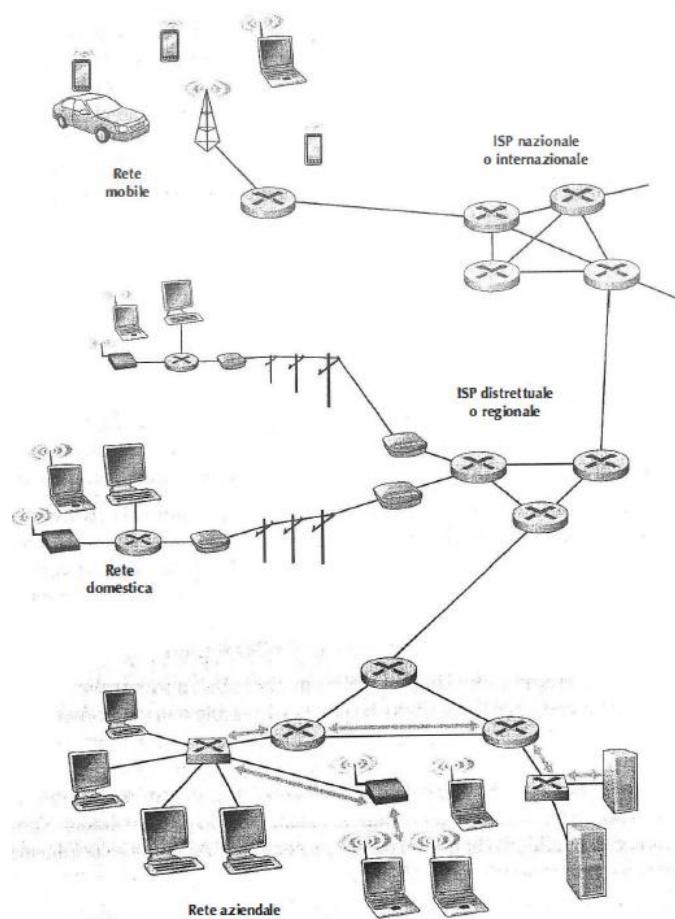


Supponiamo però che B abbia la seguente politica di routing: "non voglio avere nulla a che fare con A". Se questa è quindi la politica di B allora B non può raggiungere la destinazione

N01 perché dovrebbe passare per A. Quindi droppa il path vector. Quindi la metrica che si usa non è misurabile.



LIVELLO DI COLLEGAMENTO

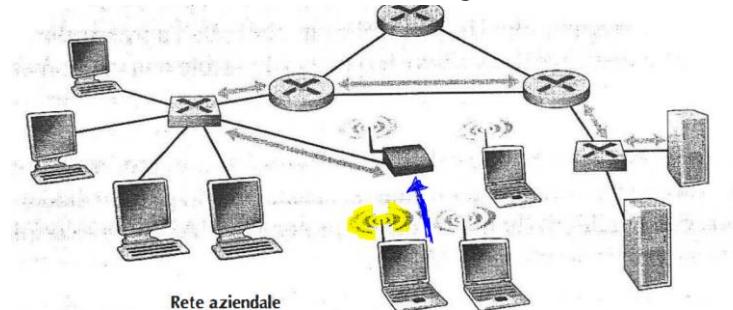


Descrivere il livello di collegamento significa fondamentalmente rispondere alle seguenti domande:

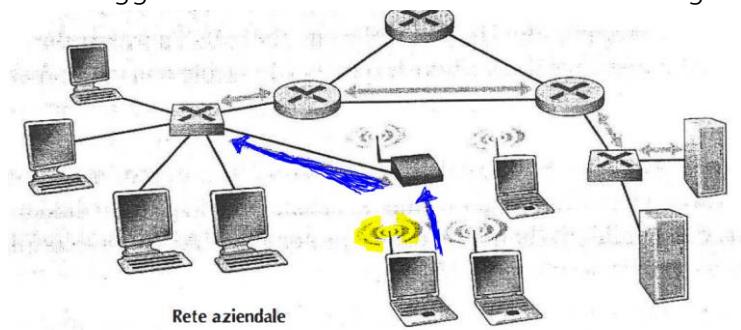
- 1) Come viaggiano i pacchetti da un router ad un altro?
- 2) Esistono indirizzi a livello di rete locale?
- 3) Che differenza c'è tra collegamenti diretti (PPP ossia Point to Point) e quelli condivisi?

4) Tanto altro...

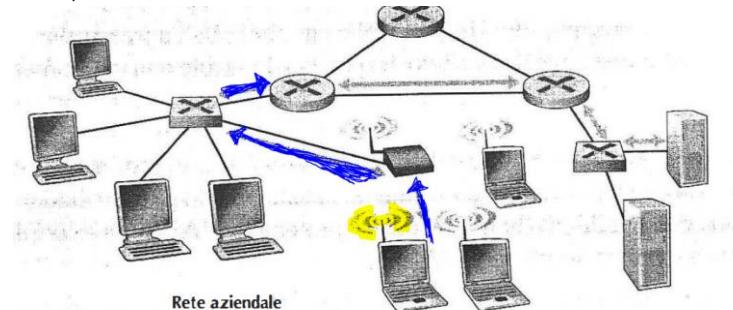
Vediamo un esempio di come si propaga un datagramma da un host wireless a un server: Il datagramma parte dall'host e arriva tramite collegamento wireless all'access point WIFI



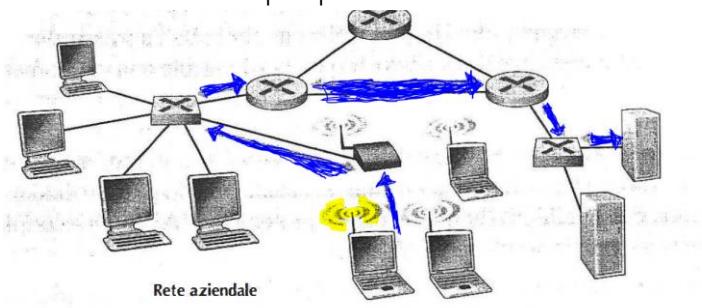
Dall'access point WiFi viaggia verso uno switch attraverso un collegamento Ethernet



Dallo switch arriva al router tramite sempre Ethernet (se non ci fossero stati altri host non ci sarebbe stato bisogno dello switch così che l'access point WiFi veniva collegato direttamente col router):



Dal router viaggia verso un altro router per poi finire allo switch e andare verso il server



Il datagramma in realtà viaggia in un pacchetto noto a livello di collegamento come **frame** del **livello di collegamento**.

Servizi livello di collegamento:

Le **funzionalità** offerte dal livello di collegamento sono:

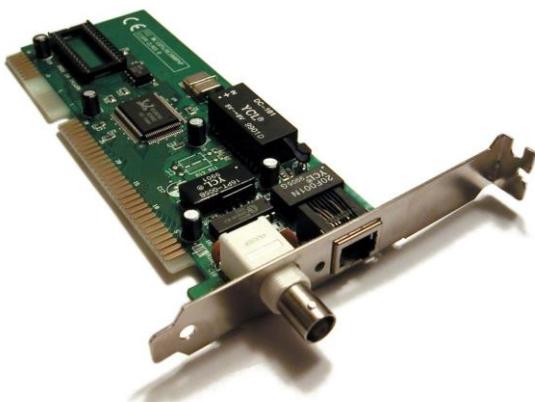
- 1) Trasporto di datagrammi da un host a un altro. Questo è il servizio **base**.
- 2) **Framing**: tutti i protocolli di collegamento o quasi incapsulano un datagramma nel payload di un *frame* che ha poi una sua intestazione.
- 3) **Accesso al collegamento**: studieremo il *protocollo MAC* che coordinerà la trasmissione dei frame sul collegamento, ossia specificherà le regole per immettervi un frame.
- 4) **Consegna affidabile**: alcuni protocolli di collegamento forniscono anche una consegna affidabile con recupero di errore esattamente come fa il TCP, ossia con scambi di acknowledgement. Può servire avere tale meccanismo a livello di collegamento piuttosto che a livello di trasporto per non costringere gli stessi protocolli di trasporto e applicazione alla ritrasmissione dei pacchetti ma risolvere tutto direttamente dal collegamento. Inoltre i problemi si risolvono sul singolo collegamento, mentre quello di trasporto deve recuperare gli errori ovunque questi succedano.
- 5) **Rilevazione degli errori**

Ma dov'è implementato il livello di collegamento?

Il livello di collegamento è implementato nelle **schede di rete** (Network Interface Card o anche detta **NIC**). Le schede di rete sono ciò che permette a un apparato elettronico-

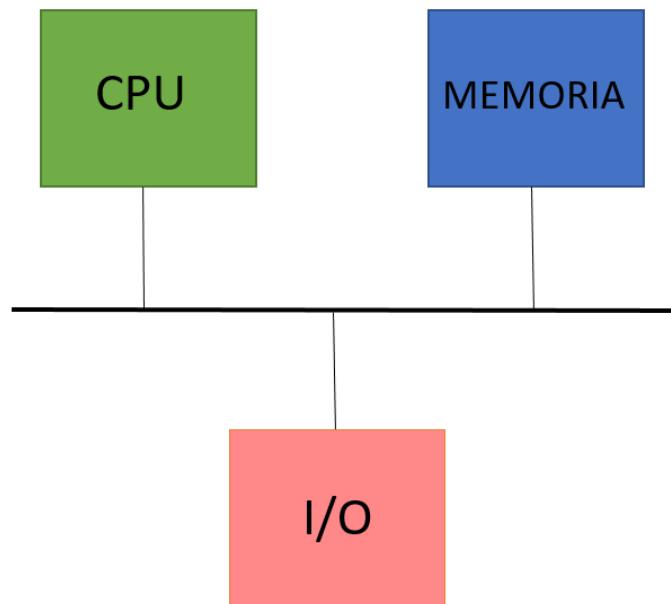
informatico la connessione ad una rete informatica e la conseguente trasmissione/ricezione dei dati.

Una scheda di rete del genere è detta **sceda di rete cablata** perché permette una connessione a internet tramite un collegamento fisico, spesso con tecnologia Ethernet. In pratica questa scheda è quella che troviamo inserita lateralmente nel nostro laptop.

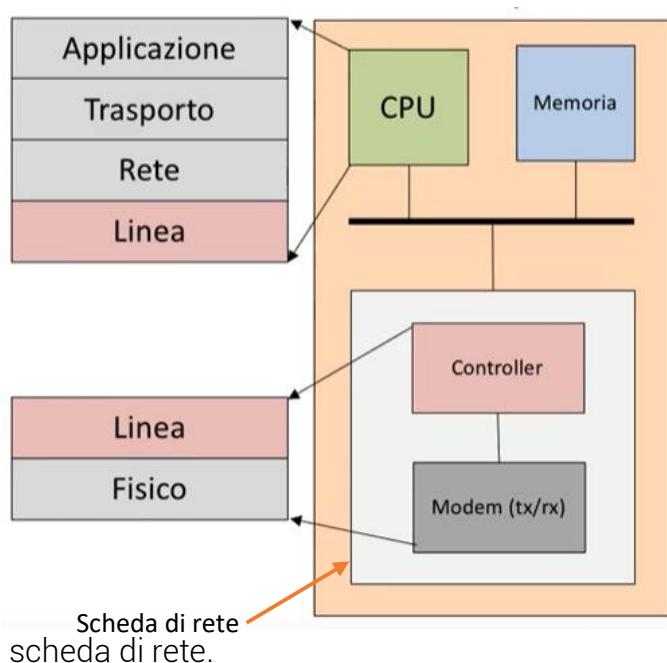


Una scheda di rete che permette invece un collegamento senza fili (che sia WiFi o Bluetooth) è detta **scheda di rete wireless**.

Il livello di collegamento è implementato da queste schede di rete. Ricordate lo schema base di un elaboratore?



Questo è un tipico schema interno di un **host**. Una scheda di rete altro non è che una interfaccia di I/O che si aggiunge a un elaboratore.



Il cuore di una scheda di rete è il **controller a livello di collegamento**. Questo esegue in **hardware** la maggior parte dei servizi offerti dal livello di collegamento quali framing, accesso al collegamento etc... vediamo però che una parte della linea di collegamento è implementata dall'host stesso, dalla sua CPU e quindi in **software**. Si tratta però di funzionalità basilari come attivazione del controller, assemblaggio informazioni di indirizzamento.

Quindi il livello di collegamento non è tutto implementato in hardware e quindi non è tutto implementato dalla sola

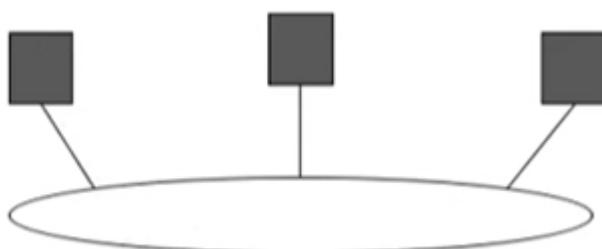
La parte software implementa più che altro ruoli importanti non tanto dal lato di invio ma dal lato di ricezione, in quanto sull'host gira un software che risponde alla stessa scheda, in particolare risponde agli **interrupt** del controller quando gli arrivano uno o più frame. Inoltre il software gestisce condizioni di errori e come detto prima anche il passaggio del datagramma dal collegamento del controller fino al livello di rete.

TIPI DI COLLEGAMENTO

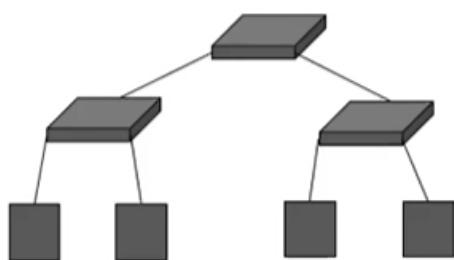
Quale funzionalità/servizio viene implementato da un *protocollo di collegamento* dipende dal tipo di collegamento e anche dalla natura fisica di esso.



Collegamenti punto-punto (PP) sono i collegamenti più facili da gestire perché mittente e ricevente hanno un canale fisico (es. fibra ottica) dedicato alla loro trasmissione.



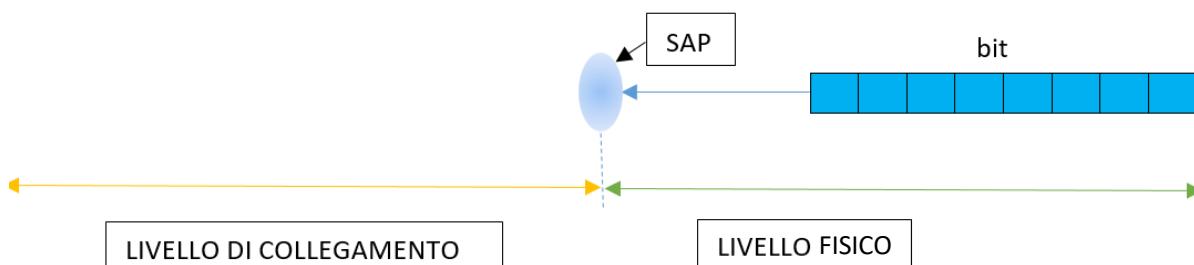
Collegamenti broadcast dove il canale condiviso è intrinsecamente condiviso. Se qualcuno invia qualcosa, chiunque abbia una scheda di rete collegata a questo canale ne riceve a livello fisico l'informazione.

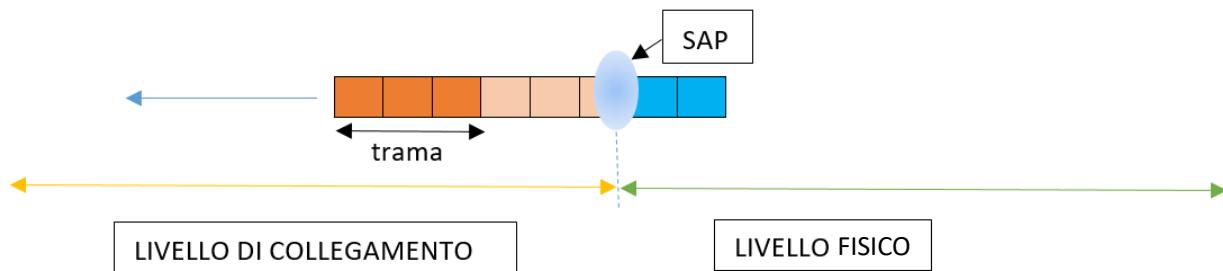


Collegamenti commutati sono collegamenti fisici in cui gerarchicamente posso arrivare a creare collegamenti punto punto. Per esempio quello in mezzo potrebbe essere uno switch a cui un host è collegato P2P e lo switch di controllo è collegato P2P a un'altra entità gerarchicamente superiore.

COSTRUZIONE DELLA TRAMA (FRAMING)

Siamo sulla scheda di rete che ha un interfaccia verso un certo collegamento fisico. In questo collegamento arriveranno dei segnali che codificheranno la stringa di bit in origine trasmessa. Quindi nel service access point tra il livello di collegamento e quello fisico arrivano i bit. E' compito del livello di collegamento quello di decidere quali e quanti dei bit che passano costituiscono una **trama**. Questa funzionalità o questo servizio offerto viene detto **framing**.





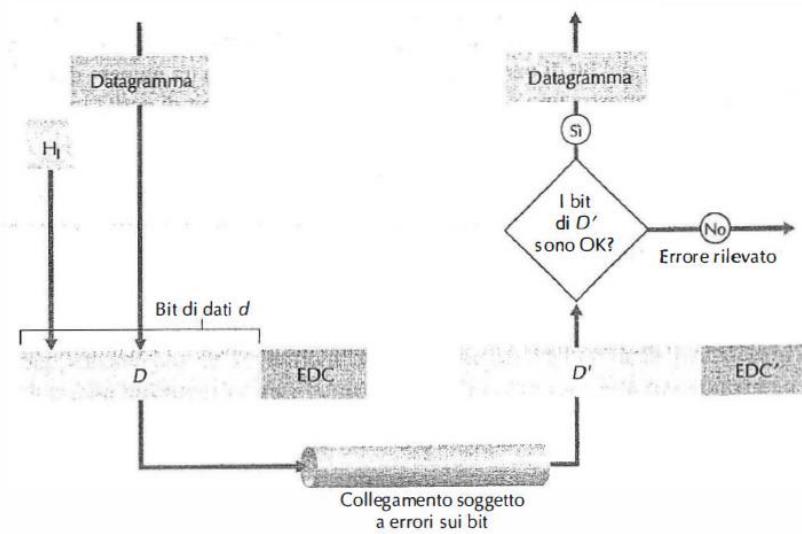
Esistono due modi per fare questo:

- 1) **Delimitatori di trama:** il ricevitore cerca delle sequenze di bit particolari che servono per delimitare l'inizio e la fine della trama.
- 2) **Segnalazioni passate dal livello fisico:** insieme con la stringa di bit non ordinata che arriva dal livello fisico arriva da questo anche una segnalazione sull'inizio e la fine del flusso dei bit (quasi mai usata).

CONTROLLO D'ERRORE

Anche il livello di collegamento può avere o no (è opzionale) un protocollo per il controllo dell'errore.

Le cause a livello di trasporto erano condizioni di code piene nei router per esempio, a livello di collegamento gli errori riguardano sono l'affidabilità del collegamento. Quindi il controllo degli errori riguardano il **controllo di errori fisici**. Pertanto qui l'obiettivo non è il recupero dei segmenti persi ma degli errori fisici.



ricevente userà il frame ricevuto con l'EDC per rilevare e eventualmente correggere l'errore. Insomma, una sorta di checksum.

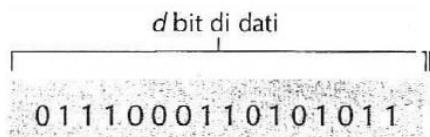
A seconda del contenuto e dell'uso dell'EDC esistono i seguenti metodi per la rilevazione e a volte anche della correzione degli errori:

- 1) **Parity Check** (insieme alla versione bidimensionale)
- 2) **Checksum**
- 3) **Controllo a ridondanza ciclica**

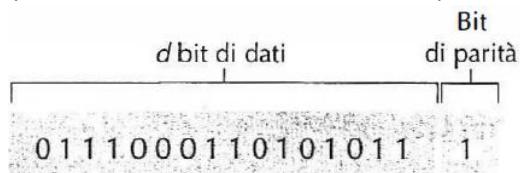
NB: tutti questi meccanismi di controllo di errore sono implementati in hardware. Però a livello di collegamento si preferisce il 3). Quando parlerò della ridondanza ciclica spiegherò il perché.

Parity Check

Supponiamo di voler mandare questa sequenza di bit:



Come EDC sceglio un particolare bit. Nello **schema di parità pari** (esiste quello dispari che è facilmente intuibile da quello che diremo) si sceglie di aggiungere un **bit di parità** che sia 1 per far sì che il numero di 1 nei d bit sia pari, 0 altrimenti. Nella figura abbiamo 9 bit a 1 nei d bit, quindi inseriamo 1 per far sì che siano in numero pari:



Al livello di collegamento arriva un datagramma. Come al solito ogni livello mette un suo header H_1 , e così farà anche il livello di collegamento. Inoltre però per rilevare l'errore inserisce nel frame creato anche una sequenza di bit chiamati **EDC** ossia **error detection and correction**. Quando il frame arriverà al ricevente, passando per un canale fisico soggetto a errori, il

Quando il ricevitore riceverà tale sequenza di $d+1$ bit conterà il numero di bit. Se sono dispari allora sicuramente *qualcuno* è stato corrotto. Ci si accorge subito di due cose: se invece di uno si corrompono un numero pari di bit il ricevente non è in grado di rilevare l'errore; se anche si rileva l'errore non si sa dove questo sia, pertanto non si può correggere. Per questo motivo è più efficiente la versione **bidimensionale** del parity check. Supponiamo di avere la seguente stringa di bit:

1 0 1 0 1 1 1 1 0 0 1 1 1 0

Si organizzano visivamente tali bit in blocchi da i righe e j colonne. Scegliamo $i=5, j=3$.

1	0	1	0	1
1	1	1	1	0
0	1	1	1	0

Per ogni riga si trova il bit di parità. Per ogni colonna si trova il bit di parità.

1	0	1	0	1
1	1	1	1	0
0	1	1	1	0

→

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

Quindi occhio, non si sta facendo alcuna somma. I bit aggiunti faranno parte dell'EDC. Inviamo tale frame e supponiamo il ricevitore riceva una sequenza che visivamente anche lui organizzerà in blocchi:

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

L'errore è quello evidenziato. Vediamo come rileva l'esistenza di un qualche errore. Parte dalla prima riga e vede se il parity check è corretto. Poi fa lo stesso con la colonna. Appena il conto (la parità) alla riga i e alla colonna j non torna allora il bit (i,j) è corrotto.

Intuiamo quindi che non solo si è rilevata l'esistenza di un errore ma lo si è anche individuato. Questa capacità è detta **forward error correction**. Questo schema consente di rilevare l'esistenza di più errori e a differenza del primo anche l'individuazione. *Ma* quando gli errori diventano tanti allora può garantire solo l'esistenza.

Checksum

E' lo stesso del checksum UDP che abbiamo affrontato tempo fa. In pratica si ricevono n bit dall'alto e li si organizzano in gruppi da k bit. Ciascun gruppo viene inteso come un numero e lo si somma con gli altri. Il risultato lo si complementa e ne diventa l'EDC.

Controllo a ridondanza ciclica (CRC)

Di solito il checksum è lasciato al livello di trasporto perché è implementato dal software del sistema operativo dell'host. Essendo software (quindi più lento dell'hardware puro), abbiamo bisogno di algoritmi semplici e veloci come il checksum, ovviamente a costo di avere un controllo limitato sugli errori.

A livello di collegamento si preferisce quindi il CRC perché più complesso ma girando in hardware risulta più veloce.

Il CRC considera ogni stringa di bit come dei polinomi. Per esempio:

11000010

Viene visto come:

$$1 \cdot x^7 + 1 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x + 0$$

Supponiamo proprio di avere quella stringa di bit da inviare al ricevitore. A questa stringa di $n = 8$ bit si affianca una stringa di $n + 1$ bit chiamata **generatore**.

Supponiamo sia:

100011101

Anche lui avrà un polinomio corrispondente:

$$1 \cdot x^8 + 0 \cdot x^7 + 0 \cdot x^6 + 0 \cdot x^5 + 1 \cdot x^4 + 1 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x + 1$$

L'algoritmo prevede di dividere questi due polinomi per ottenere il **resto**. Le regole della divisione e della moltiplicazione sono le stesse in aritmetica modulo 2, mentre l'addizione e la sottrazione sono sempre le stesse ma il riporto/prestito non esiste. In questo modo addizione e sottrazione sono identiche e possono essere fatte con uno XOR. Prima di procedere, il trasmettitore e il ricevitore devono mettersi d'accordo sulla versione CRC da usare. Una CRC a 8 bit prevede di affiancare 8 bit di zeri alla stringa da mandare, una CRC a 12 bit ne prevede 12, 16 ne prevede 16 e così via. Supponiamo di usare una CRC a 8 bit:

La parte in rosso è la sequenza da mandare, mentre gli 8 bit a zero sono quelli derivanti dalla versione CRC a 8 bit.

1	1	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	0	1						

Adesso metto sotto il generatore scelto in direzione *della prima cifra ad 1* dei dati da trasmettere.

Eseguiamo lo XOR bit a bit. Adesso è chiaro a cosa servono gli 8 bit a zero: servono a fare lo xor con la cifra mancante dei dati.

1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	0	1							
0	1	0	0	1	1	1	0	0	1						
1	0	0	0	1	1	1	1	0	1						

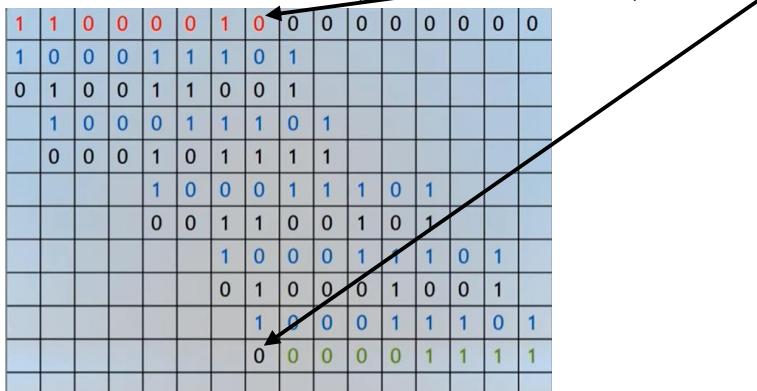
Rimettiamo il generatore sempre in direzione della prima cifra (stavolta del risultato ottenuto) che sia 1.
Eseguiamo la somma nuovamente.

1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	0	1							
0	1	0	0	1	1	1	0	0	1						
1	0	0	0	0	1	1	1	0	1						
0	0	0	1	0	1	1	1	1	1						

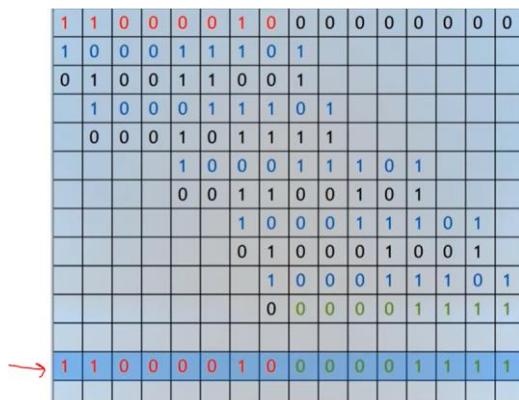
Ottenuta la somma rifacciamo di nuovo quanto fatto prima. Riprendiamo il generatore, lo mettiamo sotto:

E rifacciamo la somma e così via...

Quando ci fermiamo? Ci si ferma quando il bit meno significativo dei dati (quelli in rosso) ha nel risultato della somma, nella sua direzione, il bit a zero.



La sequenza in verde, che non sono 9bit ma 8bit, rappresenta il **resto**. Fatto questo cosa si invia al ricevitore? Al ricevitore si inviano i dati da trasmettere + il resto (questo sarà in EDC).



Cosa fa il ricevitore?

Può fare due cose:

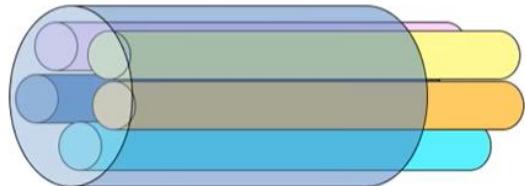
- 1) Rifà esattamente quanto appena fatto dal trasmettitore calcolando il resto della sequenza rossa con CRC a 8 bit e confrontare tale resto ottenuto con l'EDC ricevuto. Se risultano uguali non ci sono stati errori.
- 2) Prende l'intera sequenza ricevuta (rosso+verde) e applica il CRC a 8 bit. Se ottiene un resto zero allora non ci sono errori.

Abbiamo un ottimo tasso di riconoscimento degli errori, con CRC a 8 bit circa del 99.6%. Questo vuol dire che di tutti i possibili errori, il 99.6% è rilevato. Scegliendo invece il CRC a 12 o 16 e così via il tasso di riconoscimento sale fino al 99.9%.

Quindi col CRC al dato che viene inviato si aggiunge una sequenza di bit (in verde) del resto. Vedremo che nel FRAME ethernet questo campo, chiamato proprio CRC, dovrà ospitare il resto.

MULTIPLAZIONE

Multiplare a livello di collegamento vuol dire "come faccio ad assegnare un singolo collegamento broadcast a diversi flussi di traffico?". In sostanza vuol dire che dato un canale con una certa capacità visogna trovare n sottocanali da assegnare alle diverse sorgenti di traffico:



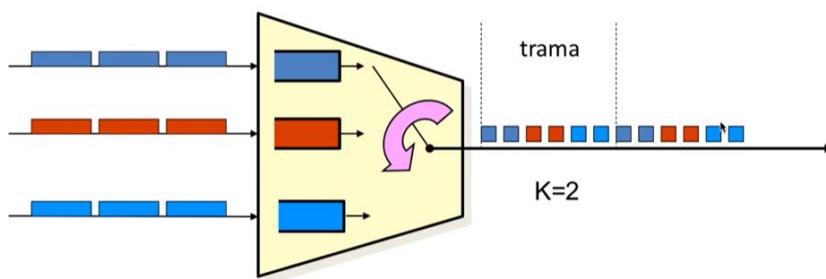
Quindi dato un tubo bisogna trovare quei piccoli tubi in figura.

Ovviamente il canale è sempre lo stesso, fisicamente vuol dire raccogliere il traffico, coordinare chi deve passare sul canale, e **demultiplare**, ossia capire a quale tubicino *virtuale* fa parte quel frame che come tutti è passato sullo stesso canale *fisico*.



Esistono 3 modi di eseguire una multiplazione del canale.

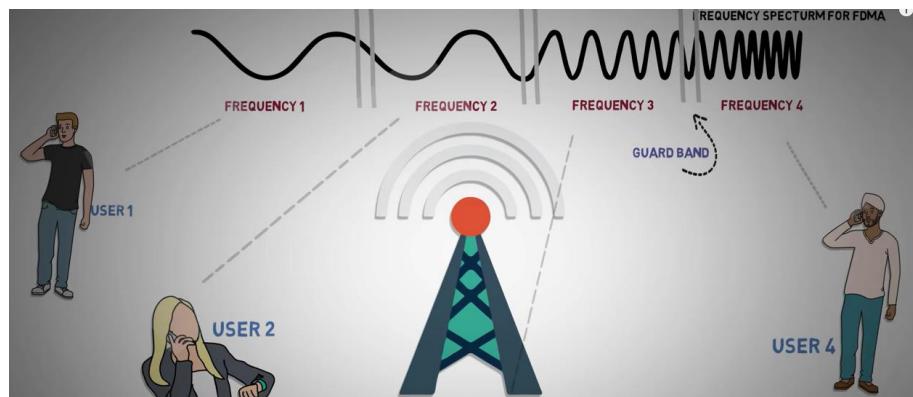
1) TDMA



Il canale di per se ha una velocità di trasmissione di R bps. TDM verifica quanti sono gli host che devono collegarsi al canale broadcast; siano questi N . Se il canale ha velocità R bps significa che ogni secondo invia R bit. Allora suddivide il tempo in **intervalli di tempo** chiamati **time frame**. Ciascun time frame poi lo divide in N parti uguali chiamati **slot temporalali**. In pratica ogni nodo ha un tempo riservato per inviare dei dati, chiamato slot (per esempio 0.3 secondi) passato il quale gli viene revocato il canale e ceduto a un altro che riempie gli altri slot. Ovviamente se ci sono N host è come se la velocità del canale per ciascuno fosse di R/N . Il problema è che un host deve aspettare del tempo anche se gli altri $N-1$ non fanno nulla quando gli

viene concesso lo slot. Non può essere decentralizzata perché c'è bisogno di un server centrale che gestisca i tempi.

2) FDMA



La FDMA divide la banda di frequenza in N parti detti **sottocanali** riservati ai vari host. Quindi a partire da un canale di R bps ne ottiene N diversi da R/N bps ed è possibile far comunicare contemporaneamente gli host. Inoltre un'altra differenza della TDA è che può essere decentralizzata. In ricezione ognuno sarà dotato di filtri passabanda per selezionare il segnale della sorgente di cui si vuole ottenere l'informazione.

3) CDMA

Invece degli slot o dei sottocanali, ai nodi vengono riservati dei codici speciali con i quali vengono marcati tutti i bit dei dati inviati simultaneamente con altri nodi sullo stesso canale. I riceventi, conoscendo il codice, sapranno come demoltiplicarlo.

COLLEGAMENTI PP: PROTOCOLLO PPP

Il protocollo PPP (point-to-point protocol) è un protocollo a livello di collegamento utilizzato nei collegamenti punto a punto:



Prevede come protagonisti:

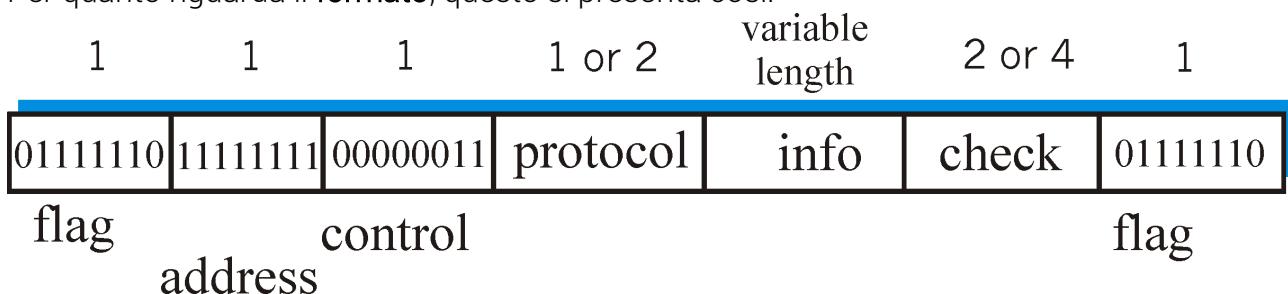
- Un sender
- Un receiver
- Un link

Il protocollo offre diverse **funzionalità** quali:

- *Packet framing*: incapsula un datagramma IP in un frame..

- *Bit transparency*: non esistono particolari byte o sequenze di byte che non possono essere inviati.
- *Error detection*: è capace di rilevare lato ricezione se c'è stato un errore ma non può localizzarlo, ossia non *error correction* ma neppure garantisce la ricezione in ordine. Per questo motivo ci si affida al tcp al livello superiore.
- *Connection liveness*: tramite periodici invii ci si accerta che il link sia ancora attivo.
- *Network layer address negotiation*: in pratica prima di inviare dati sul link avviene come col tcp una fase iniziale dove si scambiano o meglio negoziano alcuni parametri che poi saranno utilizzati durante tutta la comunicazione.

Per quanto riguarda il **formato**, questo si presenta così:

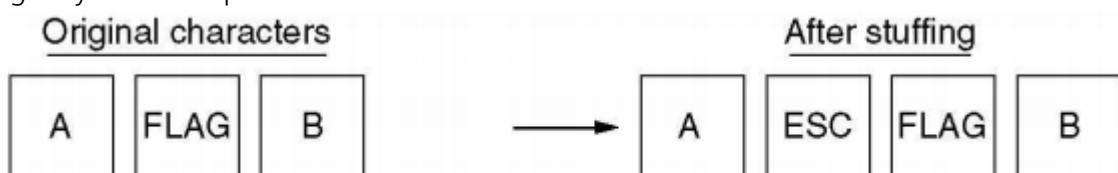


Dove:

- *Flag*: si trovano all'inizio e alla fine e sono 8 bit 01111110 in cui all'inizio vogliono indicare al ricevente l'inizio di una nuova trama, mentre nel trailer voglio indicare la fine di essa.
- *Address*: insignificante nel point-to-point di tipo unicast, come quello che vediamo. Era stato pensato in un probabile point to multiple-point e infatti l'indirizzo con tutti 1 equivale a un broadcast.
- *Control*: non utilizzato.
- *Protocol*: indica il protocollo di rete utilizzato. Da noi è sempre IP.
- *Info*: sarebbe il payload. La dimensione di esso è di norma 1500 byte ma questa lunghezza può essere contrattata durante la fase di negoziazione.
- *Check*: ospita il CRC del controllo a ridondanza ciclica.

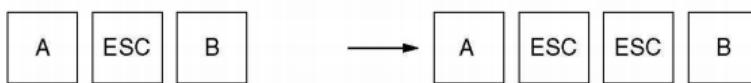
Ma cosa succede se in mezzo al payload vi siano byte di check? Meccanicamente il ricevitore li fraintenderebbe come byte di inizio e fine. Quello che potremmo fare è rendere **non utilizzabile** tale byte, ma questo andrebbe contro a uno dei servizi offerti quali il bit transparency.

Allora la soluzione è di fare quanto segue: si prende il payload e lo si controlla byte per byte. Ogni volta che si incontra un byte di flag si modifica il payload inserendo prima del byte di flag il byte di escape 01111101.



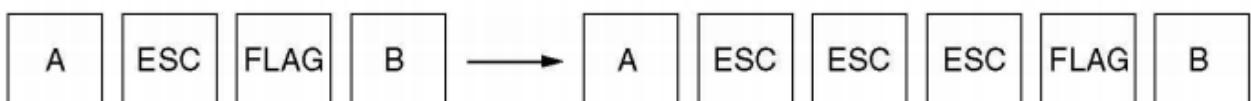
Lato ricevitore si legge da sx verso dx e quando si incontra ESC lo si elimina e si valuta il byte successivo come un byte di dati e non di flag.

Ma a questo punto diventa lo stesso escape un carattere non consentito, e sappiamo che non possiamo permetterci questo. La soluzione si modifica leggermente: si legge il payload e ogni volta che si incontra un escape o un flag lo si fa precedere da un escape.



Notiamo che c'è un ESC, allora lo facciamo precedere da un ESC. Lato ricevitore si leggerà

questo payload modificato e quando si incontra un ESC lo si elimina e si valuta il byte successivo come byte di dati.



Questo è il caso peggiore ma funziona ugualmente. A sx abbiamo quello del mittente. Lo legge e nota che c'è un escape e così lo fa precedere da un altro escape. Salta al prossimo byte e nota che c'è un flag allora lo fa precedere da escape. Lato ricevitore si farà sempre la stessa cosa anche lì: si elimina ogni escape incontrato ma ogni volta lo si elimina si salta al byte successivo a quello davanti dell'escape appena eliminato.

Lato mittente questa tecnica viene detta **byte stuffing**, mentre lato ricevitore si dirà **byte unstuffuing**.

Il protocollo PPP utilizza anche il protocollo **LCP** per scambiarsi dati quali autenticazione se magari al livello di collegamento si vuole fare tale controllo, e utilizza lo stesso PPP per comunicare.

COLLEGAMENTI BROADCAST

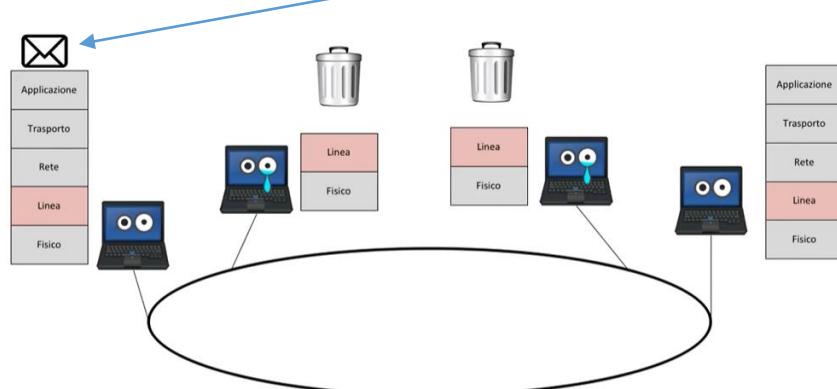
Il collegamento broadcast ha diversi nodi trasmittenti e riceventi collegati tutti allo stesso canale condiviso. Quando uno di loro trasmette un frame, tutti gli altri lo ricevono.

Ethernet (col cavo coassiale o quelli più moderni in fibra ottica o doppino intrecciato) e Wireless LAN sono esempi di tecnologie con collegamenti broadcast.

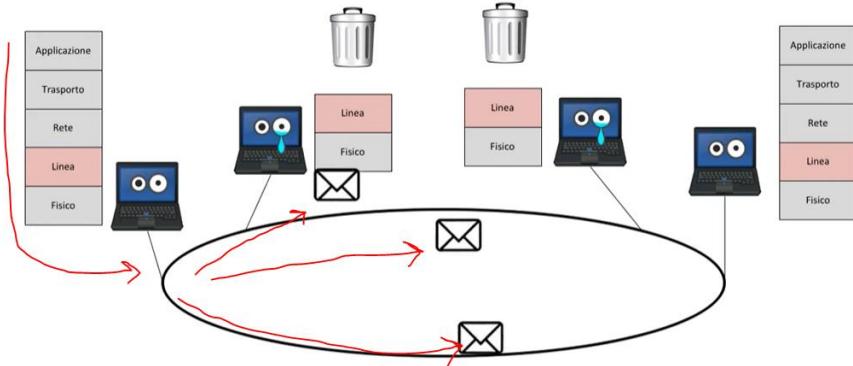


Il fatto che tutti i nodi collegati allo stesso canale ricevano lo stesso frame non vuol dire che tutti lo accettino. Siccome tutti i nodi sono collegati a livello fisico, allora a livello superiore si prenderà la scelta di quali frame accettare o meno.

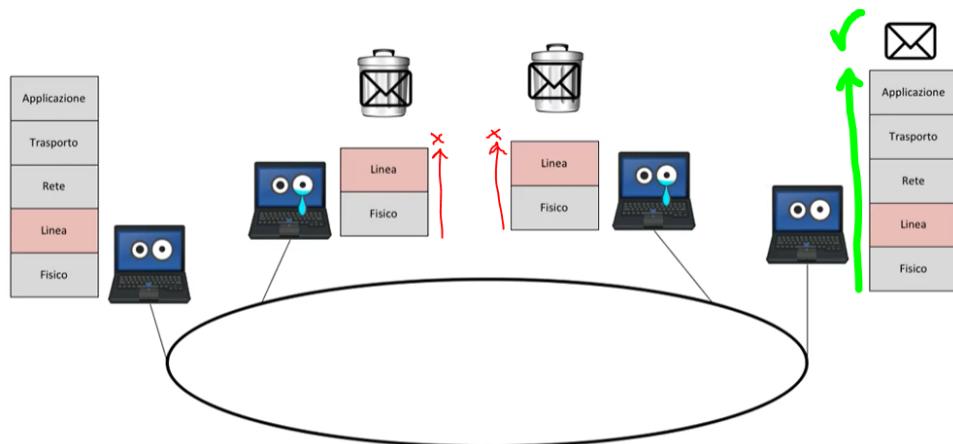
Supponiamo che il primo host voglia inviare un messaggio.



Una volta sul canale, tale messaggio verrà diffuso a tutti.

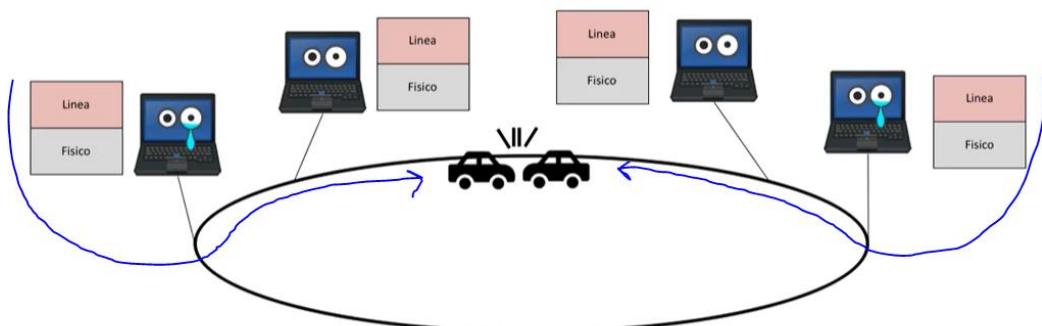


Il secondo e il terzo host decidono di non accettare quel frame, quindi non viene propagato fino al livello applicativo, a differenza del quarto host che invece lo accetta.



Vediamo quindi come l'accettazione o il rifiuto di un frame viene deciso e gestito dal **livello di collegamento**. Ovviamente il frame inviato sarà preparato con l'indirizzo fisico MAC della scheda di rete come sorgente e quello MAC del destinatario. Ogni host spacchetterà il frame e si accorgerà se la propria scheda è quella destinataria o meno.

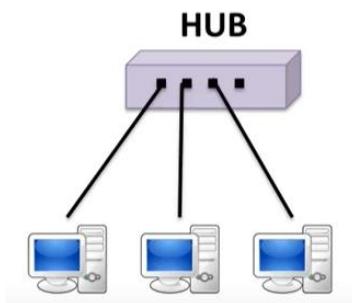
Ma cosa succede se un host invia un frame contemporaneamente ad un altro?



In questo caso si parla del **problema dell'accesso multiplo** che causa le **collisioni**. Agli host arriva sempre qualcosa, ma sarebbe la fusione dei due pacchetti in modo casuale e disordinata.

Quindi in canali broadcast mi serve un **vigile** che coordini l'accesso al canale broadcast. Questi vigili si posso implementare al livello fisico tramite multiplazione oppure come *protocolli* che vengono implementati al livello di collegamento per gestire questo problema dell'accesso multiplo.

L'entità fisica che permette di collegare insieme i bus delle singole entità con lo stesso canale viene detta **HUB**.



L'HUB è molto simile allo switch e al router. Abbiamo visto come funziona il router e tra poco vedremo anche come funziona lo switch (vd. Reti commutate più avanti).

Adesso vediamo come funziona un HUB.



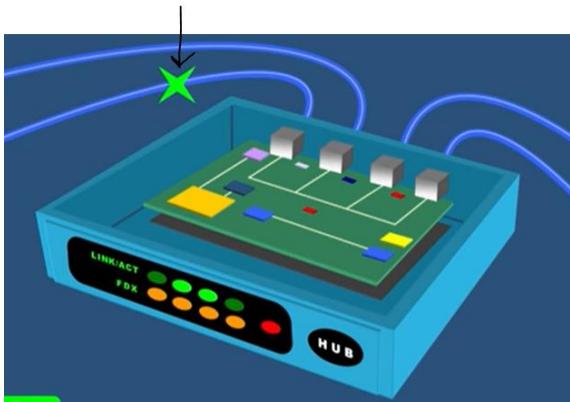
Come potete notare, un hub è strutturalmente molto somigliante a un router o a uno switch.



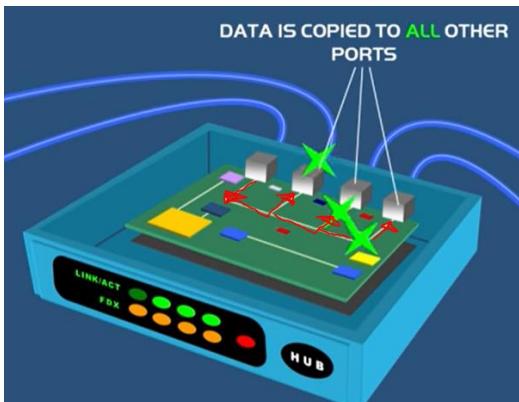
All'HUB così come al router e allo switch si possono collegare diversi host.
Ciò che quindi differenzia un HUB dagli altri è proprio la struttura interna.
Apriamolo.



All'interno non c'è alcuna intelligenza. Difatti l'HUB è la versione stupida degli altri due.



Infatti quando arriva un dato...



Questo dato è copiato su tutte le altre porte.



Abbiamo quindi realizzato un collegamento in broadcast.

NB: vedi la sezione "Dall'HUB allo SWITCH" per capire ancora meglio.

I protocolli ad accesso multiplo o protocolli MAC si dividono in:

- Protocolli a suddivisione del canale
- Protocolli ad accesso casuale
- Protocolli a rotazione

I protocolli a suddivisione del canale sono già stati affrontati quando abbiamo parlato di multiplazione, e quindi sono TDMA, FDMA e CDMA.

Pertanto affrontiamo solo gli altri due.

Protocolli ad accesso casuale

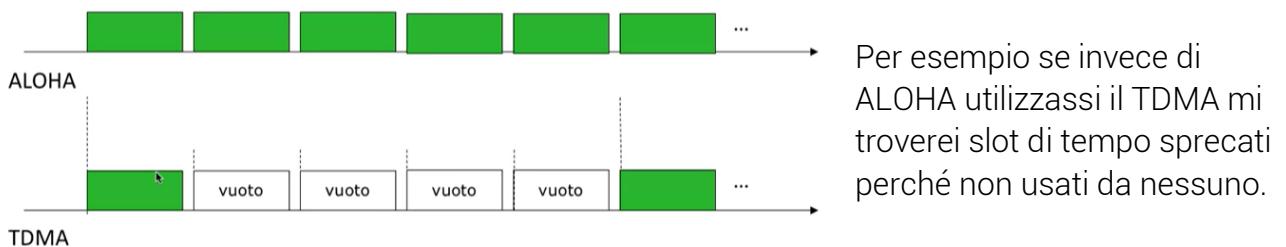
In questi protocolli ogni host sfrutta a pieno gli R bps del canale come fosse solo. Continua a mandare pacchetti nel canale insieme a tutti gli altri fino a quando non si verifica una collisione, verificata la quale i due host attivano un **tempo casuale** dopo il quale reinviare lo stesso pacchetto (questo dovrebbe evitare sovrapposizione e quindi un eventuale collisione). I due protocolli che vedremo sono:

- 1) ALOHA
- 2) Slotted ALOHA
- 3) CSMA

ALOHA

E' quello più usato come protocolli ad accesso casuale. Un nodo che ha bisogno di trasmettere trasmette subito, senza alcuna verifica del canale. Se c'è una collisione si aspetta un tempo casuale per rinviare lo stesso pacchetto. Non c'è coordinamento né segnalazione tra nodi trasmettitori.

Il **vantaggio** è che se il canale è davvero usato solo da me allora sfrutto a pieno la capacità.

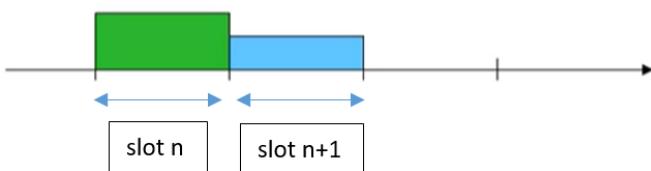


Quindi ALOHA come tutti i protocolli ad accesso casuale sono molto *efficienti* quando però il traffico è basso. Se il traffico è alto si ha lo **svantaggio** di avere troppi conflitti che invece non si avrebbero col TDMA perché ciascuno avrebbe riservato il suo slot.

SLOTTED ALOHA

In questa variante i nodi non possono trasmettere quando vogliono, ma rimane comunque sempre la possibilità di trasmettere tutti insieme. Il tempo infatti viene discretizzato in *slot* di L/R secondi (tempo di trasmissione del pacchetto, supponendo in questa trattazione che tutti i pacchetti siano di L) e i nodi possono anche tutti insieme mandare un pacchetto solo all'inizio dello slot. Immagina come se ci fosse un metronomo che scandisce il tempo di invio.

Dato che i nodi possono trasmettere solo all'inizio, possono verificarsi solamente due situazioni:



Un nodo invia un pacchetto nello slot n, e un altro nello slot n+1. Non c'è collisione perché mandandoli all'inizio di slot diversi non possono collidere.



Oppure i due frame collidono completamente.

Quindi non esistono casi in cui un frame collide "giusto un po dopo" dato che la regola dice espressamente di inviare solo quando inizia lo slot.

Facciamo un'analisi sulla probabilità di collisione.

Supponiamo che ci siano N stazioni o nodi che devono trasmettere. Supponiamo che tutte le generiche stazioni delle $N-1$ trasmettano in un certo slot con probabilità p . Allora la probabilità che queste non trasmettono è:

$$(1 - p)^{N-1}$$

Supponiamo di voler sapere la probabilità che la stazione 3 abbia successo. Tale valore è:

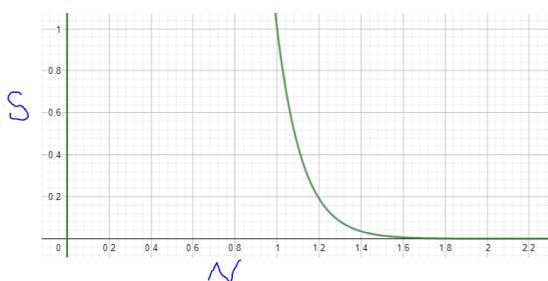
$$p \cdot (1 - p)^{N-1}$$

Supponiamo invece di non essere interessati a una particolare stazione ma ci chiediamo la probabilità che in uno slot una qualunque stazione trasmetta e abbia successo. Tale valore è:

$$S = N \cdot p \cdot (1 - p)^{N-1}$$

Questo è anche il numero medio di trasmissioni con successo in uno slot che chiamamo **efficienza**. Pertanto l'efficienza è la frazione di slot riusciti in presenza di molti nodi attivi che hanno molti pacchetti da inviare. Ci domandiamo quale deve essere il valore di p per massimizzare l'efficienza.

Ricordiamo che S rappresenta anche quanto si sta realmente sfruttando della capacità del canale: $S=0$ si sfrutta niente, $S=1$ si sfrutta il massimo.



avrà collisione.

Se $p = 0.999$ ossia la probabilità è certa che in uno slot si inviano pacchetti, allora se $N=1$ si avrà $S=1$ ossia throughput massimo. Ci accorgiamo però che S cala vertiginosamente al crescere degli N , anzi diventa 0 se esistono già anche soli due nodi che trasmettono perché ovviamente se è sicuro che si trasmette e i nodi sono due allora sicuramente si

Se $p=0.1$ allora noto che per un solo nodo attivo ($N=1$) avrò $S=0.1$ ossia sto sfruttando solo il 10% delle possibilità del canale. Se $N=2$ non avrò $S=0.2$ ma leggermente di meno sempre per via delle collisioni, però sicuramente anche per grandi N non avrò S nulli.

Il problema quindi è capire per quale p si ha S con efficienza massima. Questo valore si trova calcolando il valore di p tale per cui S è massimo, ossia:

$$\frac{\partial}{\partial p} Np(1 - p)^{N-1} = 0 \implies p = \frac{1}{N}$$

Con tale p si ha che la probabilità di successo all'aumentare dei nodi è:

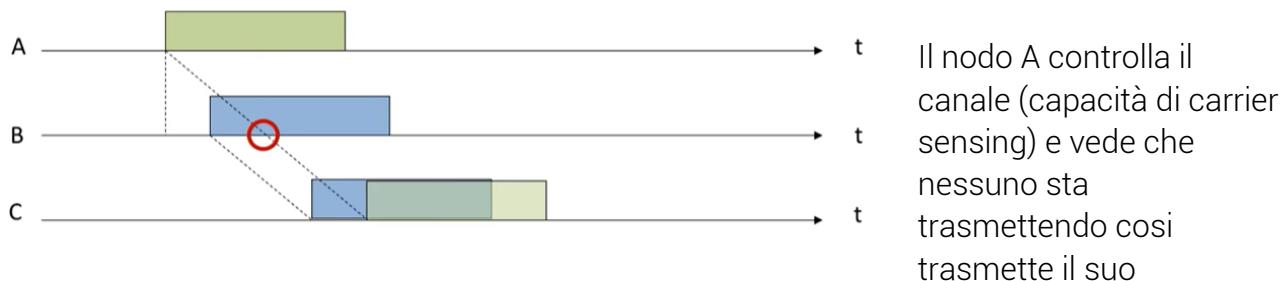
$$\lim_{N \rightarrow \infty} \left(1 - \frac{1}{N}\right)^{N-1} = \frac{1}{e} \approx 0.37$$

Si dimostra che avremmo un'efficienza ancora minore ($S=0.18$) per la ALOHA normale.

CSMA

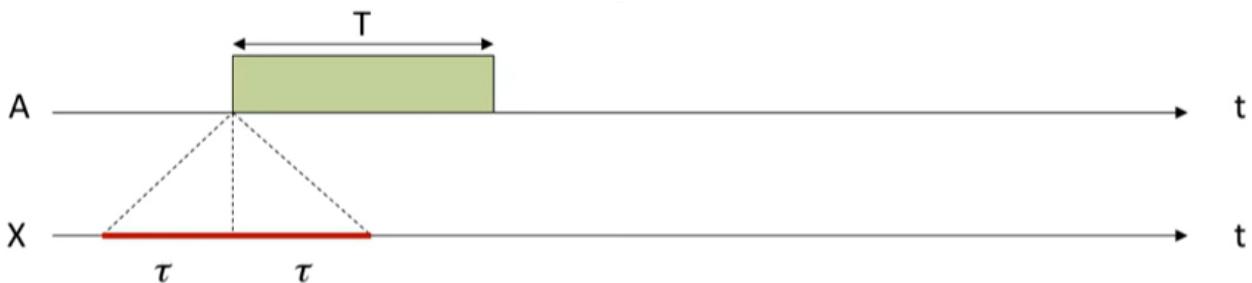
L'ALOHA nelle due versioni è dunque maleducata perché *prima di parlare* ossia prima di trasmettere *dovrebbe ascoltare* ossia verificare che qualcuno non stia già trasmettendo. Se prima di trasmettere verifico se c'è qualcuno che sta inviando posso comunque incorrere a collisioni ma in questo caso dovute solo a ritardi di propagazione. Il protocollo CSMA ascolta prima di parlare, ossia un nodo trasmette se il canale è libero. Diciamo che CSMA è un'altra versione dell'ALOHA, una sorta di Slotted ALOHA con **carrier sensing**, ossia con capacità di ascoltare il canale.

Vediamo il motivo di queste collisioni:



pacchetto. Di contro il nodo B sta in un altro posto rispetto ad A e quindi quando lui controlla dalla sua posizione il canale non vede ancora passare nessuno e mette il suo pacchetto. Durante la emissione del pacchetto può verificarsi la collisione.

Per calcolare l'**efficienza** si consideri questa figura:



Il tempo T è il tempo di trasmissione, ossia il tempo impiegato da un nodo per prendere un pacchetto e metterlo interamente sul collegamento. Invece τ è il tempo di propagazione. In pratica se X è un nodo che sta dall'altra parte del canale, ascolterà il canale ma sarà *cieco* per 2τ , ossia in quella sezione rossa chiamato **periodo di vulnerabilità**.

Immaginatevi un anello e due host X ed A messi di fronte, quindi i due stanno alla massima distanza. X ascolta il canale e non sente nulla. Mette il suo pacchetto. La velocità del collegamento è τ , questo vuol dire che dopo tale tempo arriverà il primo bit nei pressi di A che farà sì che A si accorga della trasmissione di X. Se A dovesse ascoltare il canale da

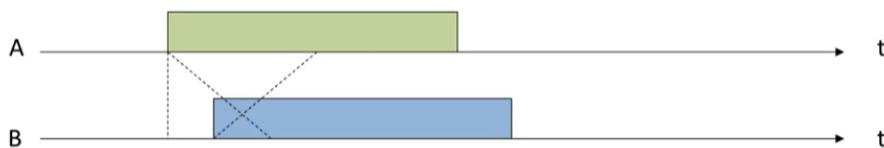
quando X ha messo su il pacchetto a giusto una frazione di tempo prima di τ (sarebbe la prima metà della linea rossa) allora non se ne accorgerebbe e ci sarebbe collisione.

Specularmente (quindi l'altra metà della linea rossa) succederebbe per X se A trasmettesse a metà della linea rossa.

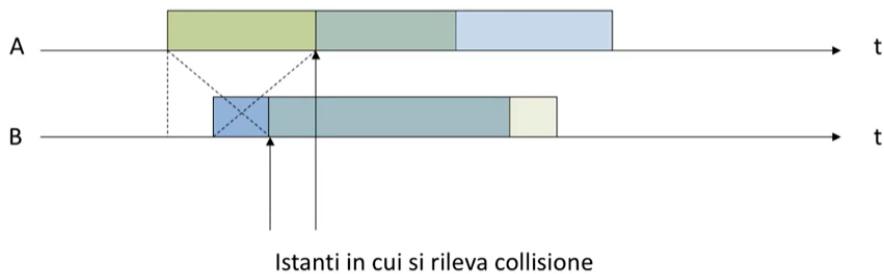
L'efficienza è quindi misurata come: $a = \frac{\tau}{T}$. Minore è il tempo/ritardo di propagazione maggiore sarà l'efficienza, ossia minore sarà il periodo di vulnerabilità (la linea rossa si riduce fino a scomparire).

CSMA-CD

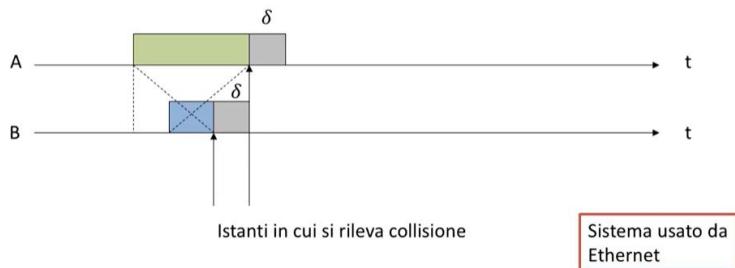
Questo è un accorgimento al CSMA, infatti CD sta per *Collision Detect*. Con questa tecnica le stazioni che trasmettono possono accorgersi che c'è qualcuno che sta per trasmettere *mentre* stanno mettendo sul collegamento il pacchetto. In tal caso interrompono la trasmissione. Quindi come prima possono ascoltare se c'è qualcuno, ma a differenza di prima possono rinunciare a tutto se nel mentre trasmettono avvertono qualcuno.



Al momento di inizio della trasmissione sia A che B non rilevano occupazione del canale, quindi trasmettono. Appena però sia A che B rilevano il primo bit dell'altro, interrompono subito l'esecuzione senza aspettare che avvenga una collisione perché probabilmente ci sarà.



Qual è il vantaggio? Che da quegli istanti in poi il canale è libero e può essere usato da altri per nuove trasmissioni. Però quando si rileva il caso sopra, A e B non smettono subito di trasmettere ma **continuano per un po' a collidere** in modo che tutte le altre stazioni si accorgano che c'è una collisione e poi per essere sicuri (che sia A o B) che la collisione non è stata dovuta a un segnale spurio sul canale (che quindi effettivamente non era occupato dall'altro).



Questo sistema di collision detect è usato da reti locali che sfruttano il protocollo **Ethernet**. L'**efficienza** di CSMA-CD, ossia la frazione di tempo medio durante la quale i frame sono trasferiti sul canale senza collisioni in presenza di un alto numero di nodi attivi e con un elevato numero di frame da inviare è:

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

dove d_{drop} è il tempo massimo che occorre al segnale per propagarsi fra una coppia di schede di rete; mentre d_{trasm} è il tempo necessario a trasmettere un frame della maggior dimensione possibile.

Ma una volta avvenuta la collisione, quanto si deve aspettare? Se aspettano un tempo uguale allora ricollideranno ancora, se aspettano un intervallo troppo grande si avrà un canale di trasmissione non sfruttato per troppo tempo, se piccolo aumenterà la probabilità di ricollidere. Il tempo di **backoff** è calcolato utilizzando un valore K di attesa dentro un insieme fatto da $[1, 2^n - 1]$ dove n è l'n-esima collisione. Questo algoritmo è detto **attesa binaria esponenziale**.

Protocolli a rotazione

In questi protocolli si cerca di soddisfare entrambi i requisiti desiderati dall'accesso multiplo, ossia quello che quando esiste un solo nodo attivo questo trasmetta a R bps, e poi che se ci sono N nodi attivi allora ciascuno trasmetta a R/N bps. I protocolli ad accesso casuale soddisfano la prima ma non la seconda a causa delle collisioni. I protocolli di rotazione li soddisfano entrambi.

Il **protocollo di polling** prevede un nodo master che interella a turno tutti gli altri inviandogli un messaggio in cui dice a un determinato nodo "il canale è tuo per un tot di frame". In quel momento quel nodo può trasmettere a R bps. Terminato il massimo numero di frame inviabili, sarà il nodo master stesso ad accorgersene osservando il canale, e così facendo contatta l'altro nodo per dirgli che può trasmettere e così via. A causa del *ritardo di polling*, ossia il tempo per notificare a un nodo il permesso di trasmettere si ha che in realtà non trasmetterà a un tasso di R bps, ma leggermente meno. Inoltre se il master si guasta è

possibile eleggerne uno nuovo tramite algoritmi specifici. Bluetooth utilizza il protocollo di polling.

Un problema di struttura del protocollo di polling è che non può essere decentralizzato; c'è sempre bisogno di una unità centrale (il master) che regoli tutto. Un protocollo che è identico al polling ma può essere decentralizzato è il **protocollo di token-passing**, ossia a passaggio di gettone. Ogni nodo quando possiede il token può inviare frame fino al massimo consentito. Quando ha termianato passa il token *sempre allo stesso* nodo. Quest'altro trasmetterà fino a un massimo e anche lui avrà sempre lo stesso nodo a cui passare il token. Inoltre se un nodo non ha alcun frame da inviare passerà subito il token. Ovviamente un problema serio è quando uno dei nodi si danneggia.

RETI LOCALI COMMUTATE

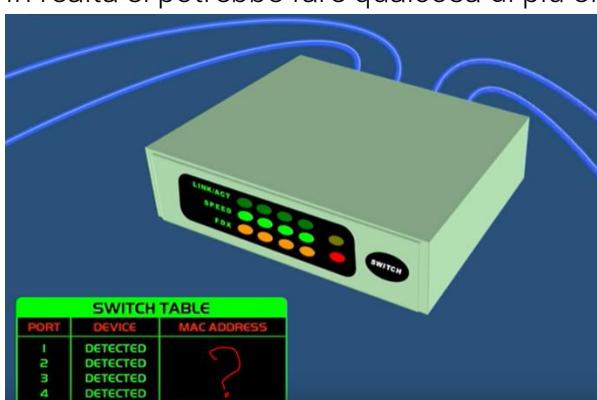
Il WiFi è intrensicamente un canale broadcast, ossia la trasmissione di un certo dispositivo viene ricevuta da tutti quelli collegati alla rete. In caso di reti cambiate fino adesso abbiamo visto solo che le singole stazioni venivano collegate ad un HUB che ne faceva da ripetitore di segnale verso tutte le altre stazioni. In questo modo le reti con **collegamento broadcast** prevedono che ciò che si manda venga inviato a tutte le interfacce di rete e quindi necessitano di *protocolli ad accesso multiplo* per evitare quanto più possibile le collisioni. Le reti con **collegamento commutato** invece, grazie allo *switch*, riescono a evitare le collisioni. (Vedi nel paragrafo successivo, verso la fine)

Dall'HUB allo SWITCH

LIVELLO FISICO

L'HUB è un **dispositivo a livello fisico**, ossia si limita a ricevere bit e ritrasmetterli su tutte le porte.

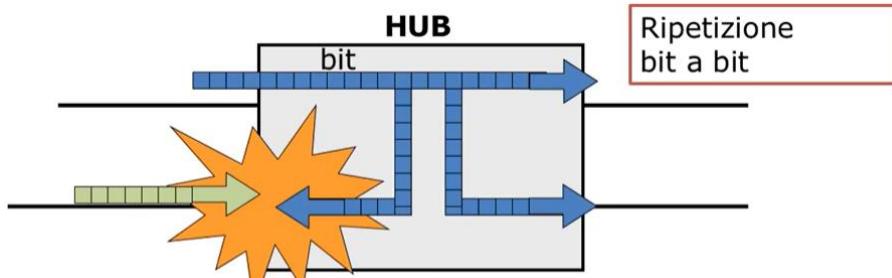
In realtà si potrebbe fare qualcosa di più efficiente. La soluzione è sostituirlo con uno **switch**



Lo SWITCH è davvero molto simile all'HUB, ma leggermente più intelligente.

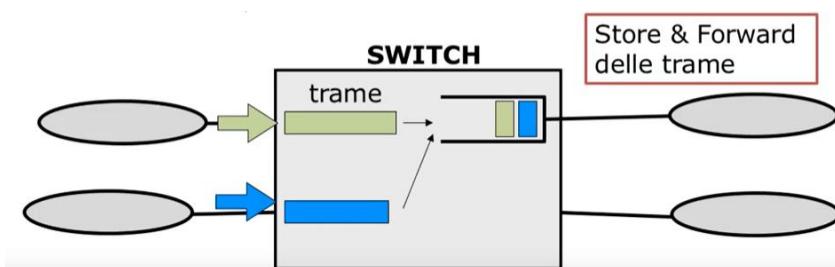
LIVELLO DI COLLEGAMENTO
LIVELLO FISICO

Lo SWITCH è un dispositivo a livello di collegamento, ossia non ritrasmette singoli bit ma interi frame.



qualche altra porta stava inviando dei bit e quindi avveniva una collisione.

Nell'HUB quello che succedeva è che esso riceveva bit da una porta e li replicava su tutte le altre. Poteva succedere però che

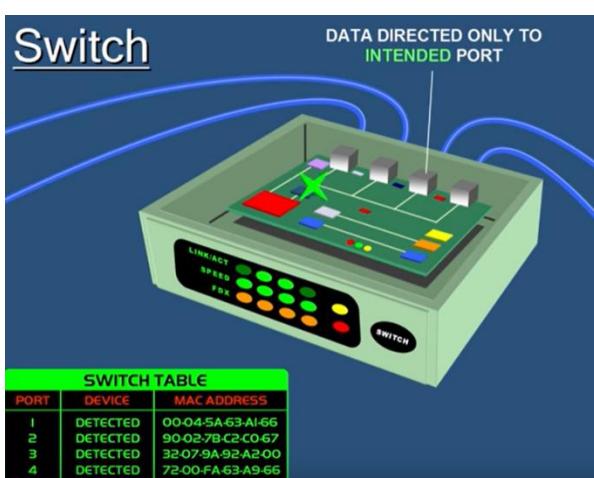


Lo SWITCH invece lavora con i frame. Li riceve totalmente, li processa (ossia li legge) e a seconda di ciò che vi sta scritto nella trama li inoltra verso una determinata porta. In pratica come il router

implementa lo store & forward visto in precedenza.

Lo switch è completamente **trasparente** ai nodi i quali mandano il frame verso un dato nodo ignari del fatto che verrà processato dallo switch.

Fondamentalmente lo switch è molto simile a un router, però quest'ultimo lavora a livello di rete. Pertanto per lui saranno gli indirizzi IP a decidere verso dove inoltrare il pacchetto. Ma allora cosa legge uno switch?



Lo SWITCH infatti ha una **switch table**, equivalente a una tabella di routing ma di livello 2. La tabella presenta un indirizzo fisico MAC (vediamo dopo cosa sia) e una porta. La tabella si legge: "se voglio raggiungere un certo indirizzo MAC prendo una certa porta".

Attraverso la tabella viene fatta l'operazione di **filtraggio e inoltro/scarto** verso una interfaccia. Supponiamo arrivi un frame dall'interfaccia x che porti un indirizzo di destinazione MAC. Sono possibili 3 casi:

- L'indirizzo MAC si trova nella tabella e l'interfaccia a cui inoltrarlo è diversa da x, ossia dal mittente. A questo punto si prende il frame e lo si pone nel buffer dell'interfaccia di uscita.
- L'indirizzo MAC si trova nella tabella e l'interfaccia a cui inoltrarlo è uguale x, ossia dal mittente. A questo punto il frame viene scartato.
- L'indirizzo MAC non si trova nella tabella. A questo punto il frame viene messo nel buffer di uscita di tutte le interfaccie meno che x, ossia lo si inoltra in broadcast.

Ma come viene riempita questa tabella? Nel router la tabella di routing veniva riempita dai protocolli di instradamento. Nello switch non è così. Si dice che lo SWITCH riempia la tabella in modo **implicito**, quasi **osservando il traffico che da lui vi passa**. Ma come può imparare automaticamente da quale porta si raggiunge una certa stazione?

Gli switch si dicono essere **plug-and-play** perché non necessitano di alcun intervento manuale per popolare la loro tabella. L'idea è di avere una tabella con un campo in più rispetto a quella vista prima:

Indirizzo	Interfaccia	Tempo
01-12-23-34-45-56	2	9:39
62-FE-F7-11-89-A3	1	9:32
7C-BA-B2-B4-91-10	3	9:36
....

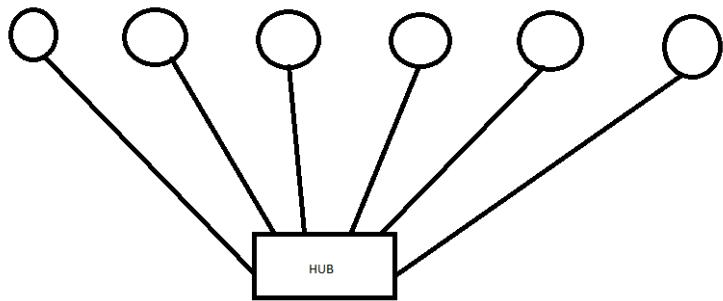
In pratica quando arriva un frame da una certa sorgente, lo switch controlla se l'indirizzo MAC sorgente è presente in tabella. Se non è presente lo inserisce insieme anche all'interfaccia di ingresso dal quale sta inviando, e quindi anche dalla quale potrà ricevere. Inserirà anche l'orario perché dopo un certo *tempo di invecchiamento* detto **aging time** in cui il router non riceve nulla da quell'indirizzo MAC, significa che probabilmente non c'è più e quindi può essere sovrascritto. In questo modo riempie autonomamente la tabella.

I **vantaggi** dello SWITCH rispetto allo HUB sono quindi chiari:

1) Eliminazione delle collisioni

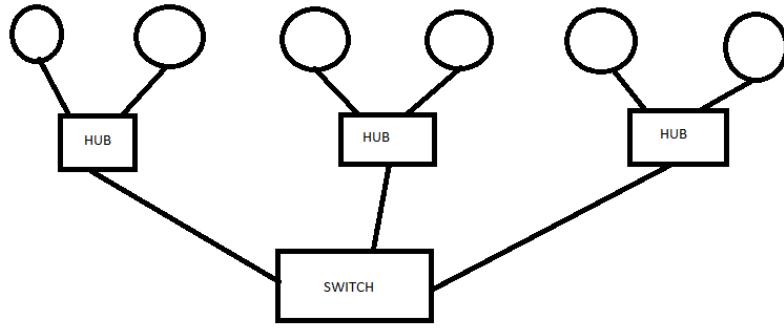
In realtà le collisioni vengono **ridotte** se non si fa dell'altro che ora spiego.

Supponiamo di avere questa rete a collegamento broadcast:



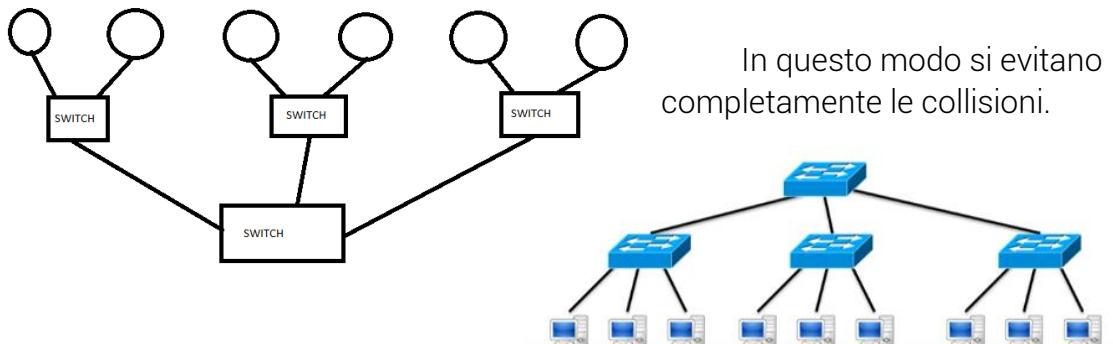
Come sappiamo, qui le collisioni sono possibili e anzi molto probabili.

Siamo andati oltre introducendo gli switch:



Normalmente si intendeva che ad ogni porta dello switch fosse collegato un **dominio** di rete che può normalmente essere un solo host o una rete a collegamento broadcast.

Nel caso sopra lo switch **elimina le collisioni** tra i vari domini di rete ma non le evita tra gli host dei domini stessi. Per questo motivo si è giunti oggi giorno ad avere **reti completamente commutate** in cui si usano solo switch.



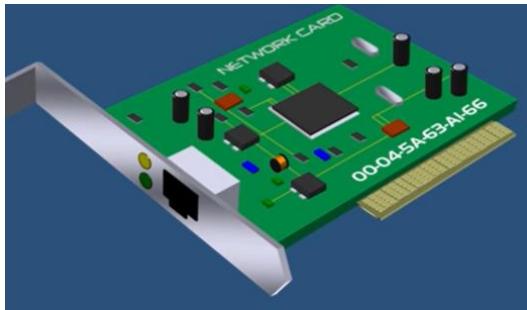
In questo modo si evitano completamente le collisioni.

- 2) **Collegamenti eterogenei:** grazie al fatto che lo switch isola completamente i due collegamenti tra entrata e uscita, allora i due collegamenti possono pure essere realizzati fisicamente con tecnologie diverse.

Mac Address

Abbiamo visto che host e router (o meglio le loro interfacce/schede di rete) hanno indirizzi IP. Questi sono *indirizzi a livello di rete*. A livello di collegamento esistono altri tipi di indirizzi chiamati **MAC address**.

L'indirizzo MAC è lungo 6 byte (2^{48} bit) e sono scritti in esadecimale scrivendo ogni byte con due cifre esadecimali e salvati in una eprom nella stessa scheda di rete:



es. 00-04-5A-63-A1-66

NB: Anche il MAC address ha un equivalente indirizzo broadcast--> FF-FF-FF-FF-FF-FF

Protocollo di risoluzione degli indirizzi ARP

E' un protocollo a *livello di rete* che permette di sapere l'indirizzo MAC che corrisponde a un certo indirizzo IP. Su ogni host esiste una tabella detta **tabella ARP** memorizzata su una *cache* a lei dedicata in cui sono memorizzate coppie indirizzoIP-MACaddress.

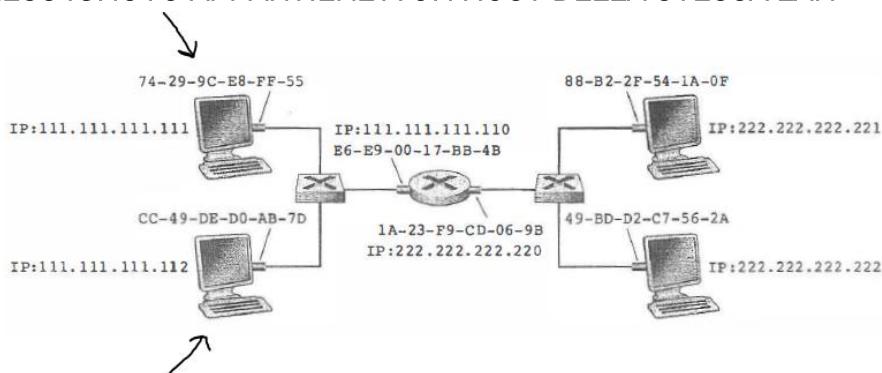
Nota bene: ARP risolve soltanto gli indirizzi IP per i nodi della stessa sottorete (fa parte anche il router di bordo). Se un nodo a Torino cerca di utilizzare ARP per risolvere l'indirizzo IP di un nodo di Venezia, ARP segnala un errore.

Ogni volta che preparo un pacchetto IP in cui nell'header inserisco l'SRC e il DEST sottoforma di indirizzo IP, quando consegno il pacchetto al livello di collegamento questo inserirà un header fatto di SRC e DEST in termini stavolta di MAC address.

NB: il livello di rete non passa al livello di collegamento solo il pacchetto IP, ma anche l'indirizzo MAC di destinazione che rileva quando consulta la tabella ARP. Il livello di collegamento quindi si limita solo a creare il frame inserendo nel payload lo stesso datagramma e come indirizzo di destinazione header del frame l'indirizzo mac consegnato. Ogni richiesta viene chiamata **ARP request**, mentre ogni risposta **ARP reply**.

Cosa succede se non trovo alcun riferimento? Distinguiamo due casi:

IL MAC ADDRESS IGNOTO APPARTIENE A UN HOST DELLA STESSA LAN



Per capire se l'host di destinazione appartiene alla propria LAN, quello che fa è di controllare che l'indirizzoIP della sottorete a cui appartiene (111.111.111.0) sia uguale a quella della sottorete di destinazione (111.111.111.0), e questo lo fa al solito modo, verificando che i primi *n* bit più significativi siano uguali.

Verificato che il destinatario appartiene alla stessa LAN, quello che fa l'host mittente è inviare uno speciale pacchetto chiamato **pacchetto ARP** che fungerà da *richiesta*. In pratica un pacchetto ARP di richiesta è così fatto:

Indirizzo IP sorgente	MAC address sorgente
Indirizzo IP destinazione	FF-FF-FF-FF-FF-FF

ARP lo manderà alla scheda di rete che ne creerà il frame.

Nel nostro esempio abbiamo che:

111.111.111.112	CC-49-DE-D0-AB-7D
111.111.111.111	FF-FF-FF-FF-FF-FF

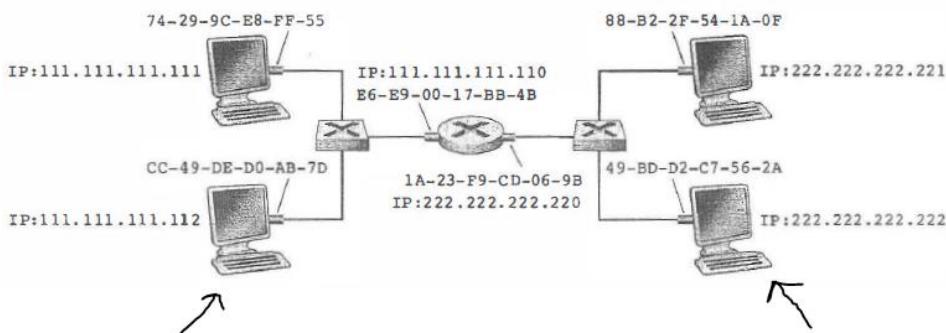
Come destinazione usiamo l'indirizzo MAC di **broadcast** perché vogliamo *interrogare tutti i nodi della LAN* per sapere a chi di loro corrisponde quell'indirizzo IP.

Uno di loro (nel nostro caso quello sopra) verificherà di avere quell'indirizzo IP di destinazione e se è così risponderà con un pacchetto ARP di **risposta**:

111.111.111.111	74-29-9C-38-FF-55
111.111.111.112	CC-49-DE-D0-AB-7D

A differenza di quello di richiesta, quello di risposta è **unicast**, ossia diretto all'interessato e presenta nel campo MAC address sorgente quello cercato che verrà salvato nella propria tabella ARP. Adesso il mittente può ricominciare a trasmettere consultando una tabella ARP stavolta consistente.

IL MAC ADDRESS IGNOTO NON APPARTIENE A UN HOST DELLA STESSA LAN



Lo capisce verificando che l'indirizzo della sua sottorete (111.111.111.0) non corrisponde a quella di destinazione (222.222.222.0). A questo punto sa che il destinatario si trova fuori la propria LAN e può raggiungerlo solo tramite il router di bordo. Ora, se non conosce il MAC address del router, ma conosce il suo IP, si ricade nel caso di prima (il router appartiene alla LAN). Comunque sia, una volta conosciuto il suo indirizzo MAC invia il seguente frame:

111.111.111.112	CC-49-DE-D0-AB-7D
222.222.222.222	E6-E9-00-17-BB-4B

Nota come ¾ del pacchetto riportino le sole informazioni di se stesso e sull'host destinatario, mentre il MAC address è quello del router.

Da qui in poi ci penserà il router a inoltrarlo fino all'host desiderato. Per vedere come funziona questo tipo di inoltro vedere il prossimo paragrafo in particolare l'inoltro indiretto.

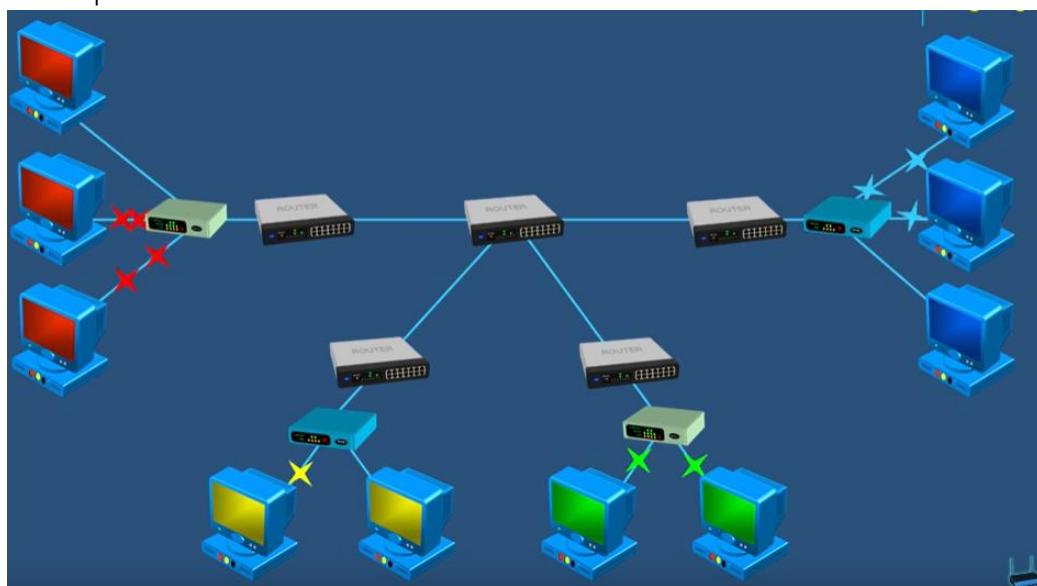
INOLTRO IN IP

(Questa sezione appartiene al capitolo 4 ma per maggiore chiarezza verrà affrontata solo adesso)

Il protocollo IP è una tecnica di **internetworking**, ossia capace di far comunicare più *reti locali*. Nel trasferimento di pacchetti tra due host, l'inoltro può essere:

- **Diretto** quando la destinazione si trova nella stessa rete locale (o anche detta rete IP)
- **Indiretto** quando la destinazione non è nella stessa rete locale

Di solito le **reti locali** o **reti IP** o **LAN** vengono create tramite HUB o meglio ancora SWITCH, mentre la **connessione di più reti locali tra di loro** viene realizzata tramite ROUTER. Questo è un concetto importante.

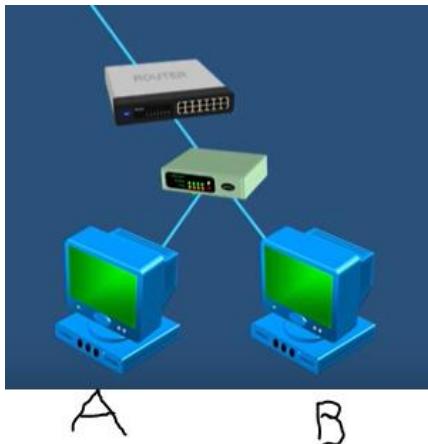


Questa figura mostra 4 reti locali. Tutte e 4 sono realizzate tramite SWITCH. Fino a quando ogni host vuole comunicare con un altro host appartenente alla stessa rete locale allora **basterà il solo livello di collegamento** a collegare i due host, e infatti lo SWITCH non fa che questo. Quando però un host vuole comunicare con uno non appartenente alla stessa LAN allora **è necessario un collegamento a livello di rete** e qui entra in gioco il router.

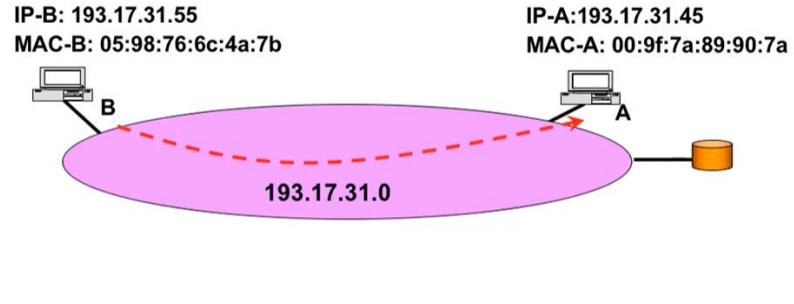
Iniziamo con calma.

INOLTRO DIRETTO

Vediamo cosa ci vuole perché due host della stessa rete locale possano comunicare.



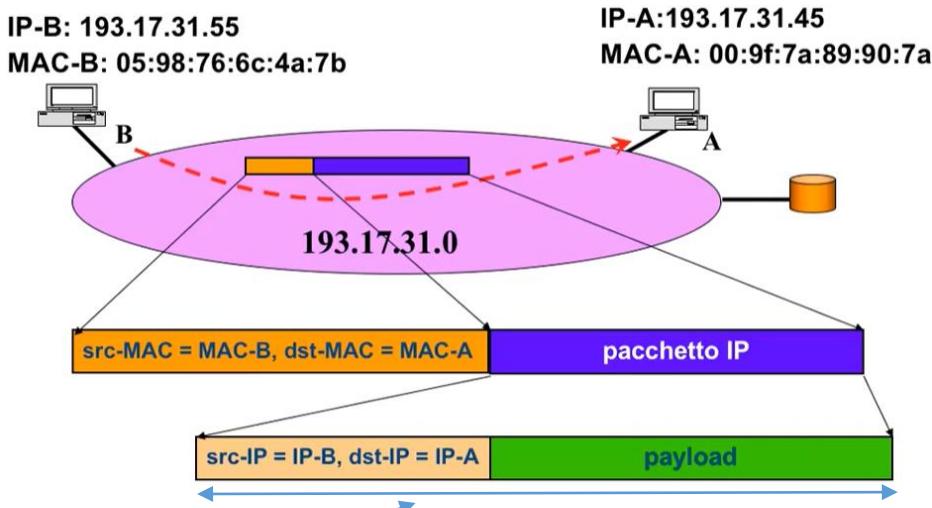
In generale potrebbe essere una rete locale più grande



Supponiamo che B voglia mandare un pacchetto ad A.

B conosce l'indirizzo IP di A. La prima cosa che deve fare è capire se A si trova nella stessa rete. Questo è facile, basta sapere l'indirizzo IP della sottorete (193.17.31.0) che si può trovare direttamente dal proprio indirizzo IP e verificare che se anche l'indirizzo della sottorete di A coincide con quello trovato.

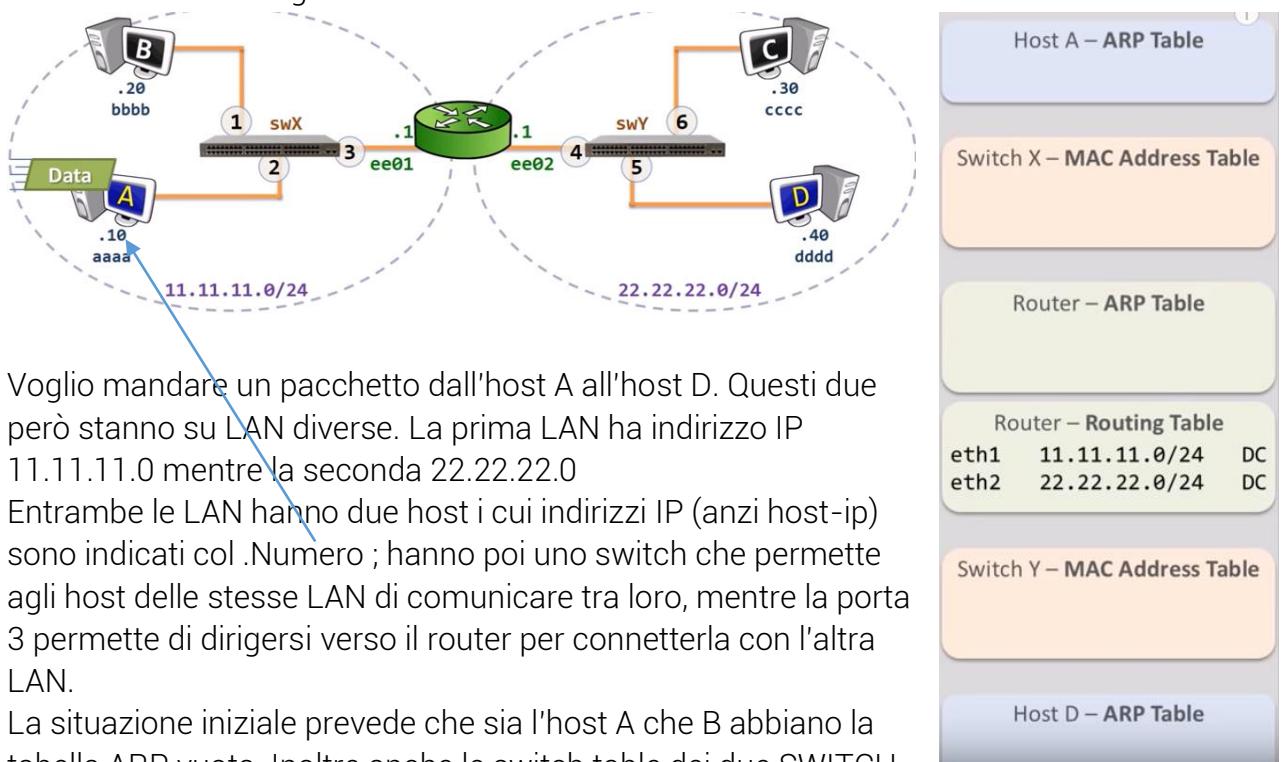
Quando B verifica che A appartiene alla stessa rete locale consulta quindi una tabella particolare, che risiede nella memoria di ogni host, chiamata **tabella ARP**, in cui per ogni indirizzo IP vi è il rispettivo indirizzo MAC (se non esiste la tupla cercata si invia un ARP request attendendo un ARP reply). Poiché A appartiene alla propria rete, sapendo che in mezzo c'è uno switch che commuta a seconda dell'indirizzo MAC di destinazione, prepara il seguente pacchetto:



Ossia consegna il pacchetto IP al livello di collegamento che aggiunge all'header l'indirizzo MAC di sorgente (utile per riempire la tabella in caso di mancanza) e di destinazione. In questo modo il frame arriva allo switch che lo commuta ad A.

INOLTRO INDIRETTO

La situazione è la seguente:

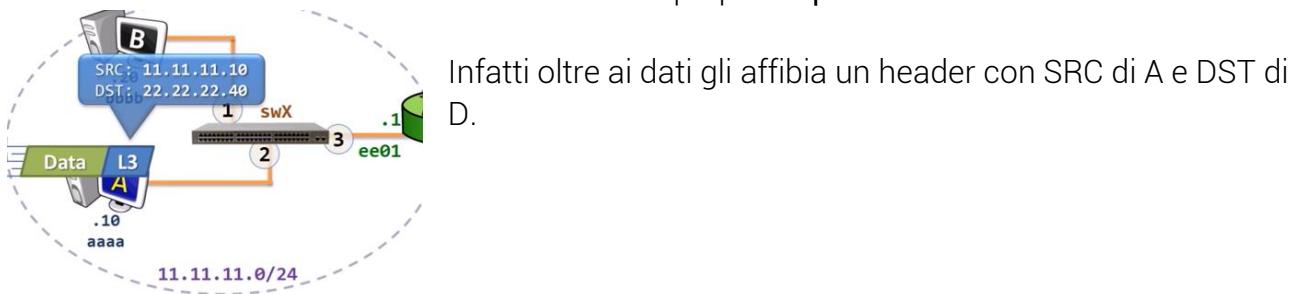


Voglio mandare un pacchetto dall'host A all'host D. Questi due però stanno su LAN diverse. La prima LAN ha indirizzo IP 11.11.11.0 mentre la seconda 22.22.22.0. Entrambe le LAN hanno due host i cui indirizzi IP (anzi host-ip) sono indicati col .Numero ; hanno poi uno switch che permette agli host delle stesse LAN di comunicare tra loro, mentre la porta 3 permette di dirigersi verso il router per connetterla con l'altra LAN.

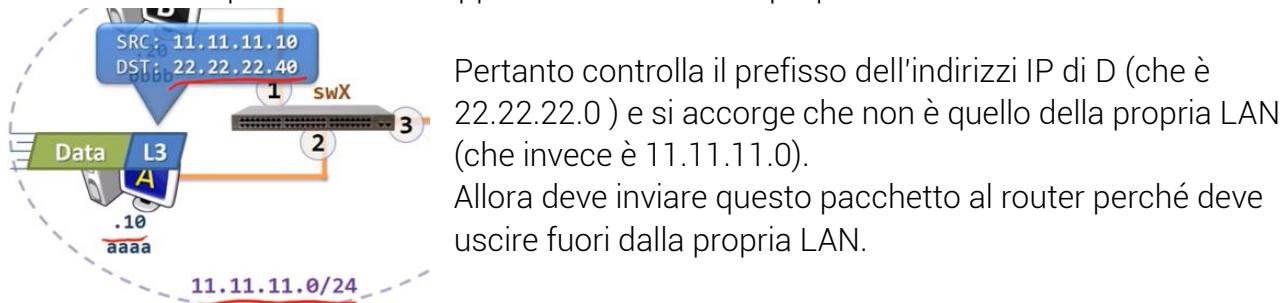
La situazione iniziale prevede che sia l'host A che B abbiano la tabella ARP vuota. Inoltre anche le switch table dei due SWITCH risultano vuote. Infine il router ha la sua ARP vuota (ricorda che chiunque trasmetta o riceva pacchetti IP deve avere una tabella ARP) e infine una tabella di routing.

NB: Stiamo davvero partendo da zero: senza nessuna ARP configurata, ne switch table. Così vedremo ogni singolo intoppo come si possa risolvere.

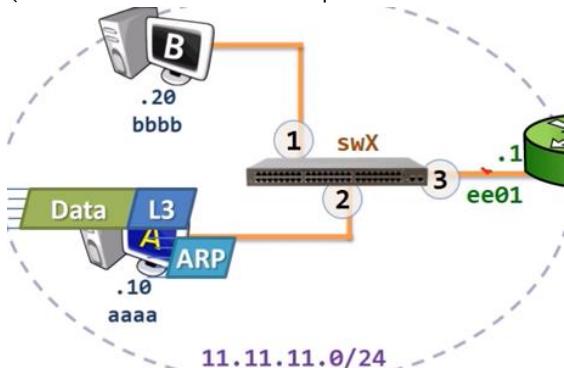
L'HOST A conosce l'indirizzo IP dell'host D così prepara il **pacchetto IP**:



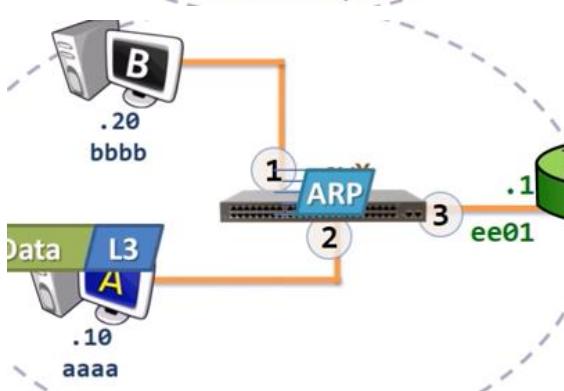
Adesso deve capire se l'host D appartiene o meno alla propria rete LAN.



Come sappiamo dobbiamo inoltrare il pacchetto al router di bordo perché è lui che può permetterci di uscire fuori dalla nostra piccola rete locale e connetterci col mondo. Ho bisogno del suo MAC address ma per sfortuna nell'ARP non ne abbiamo alcun riferimento (è un caso sfortunato, potremmo avere un ARP bella e pronta).

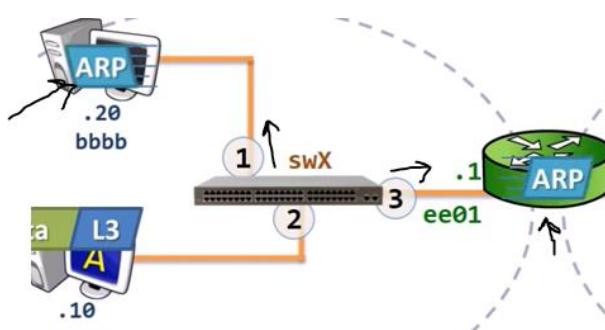


Allora l'host A prima di inviare il pacchetto invia un **pacchetto ARP di richiesta** in broadcast per domandare a chi appartenga l'indirizzo IP 11.11.11.1 (almeno l'IP lo conosce).

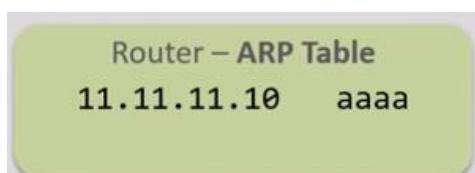


L'ARP arriva allo switch. Sappiamo che lo switch table apprende da solo che gli manca l'indirizzo MAC dell'entrata 2, quindi lo ricava dal MAC SRC del pacchetto di richiesta:

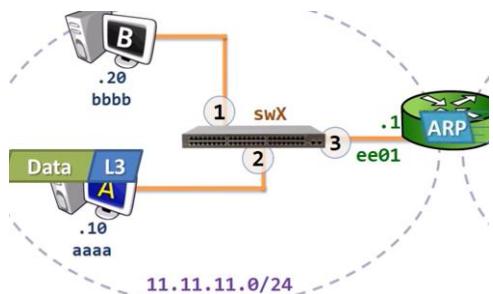
Switch X – MAC Address Table
2 aaaa.aaaa.aaaa



Il collegamento in broadcast prevede che il pacchetto ARP sia inviato a tutti.

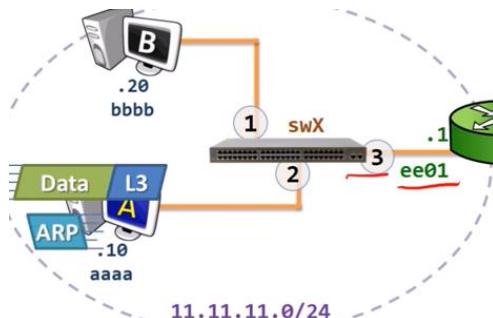


La prima cosa che fa il router è salvarsi l'indirizzo MAC dell'host A. Ricorda che così come A anche il router ha una tabella ARP (un router è come un host), ma siccome è collegato a due LAN allora ne avrà due (qui sono messe insieme).



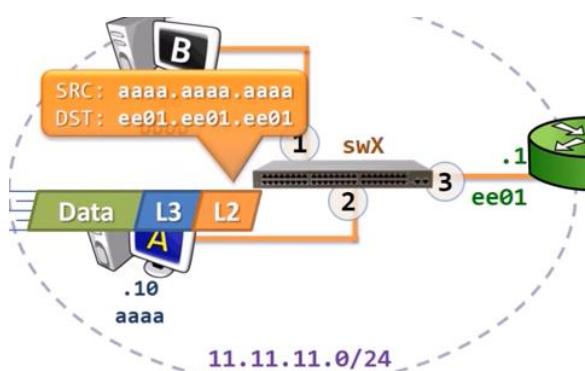
Il router genera il pacchetto ARP di risposta e lo invia allo switch il quale si autocompleta nuovamente:

Switch X – MAC Address Table	
2	aaaa.aaaa.aaaa
3	ee01.ee01.ee01

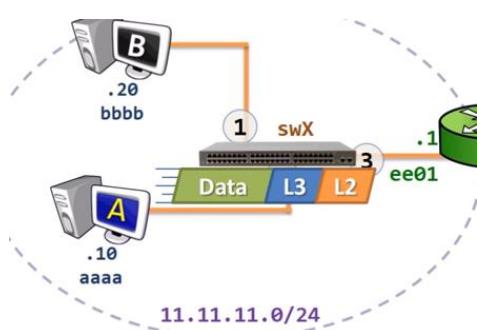
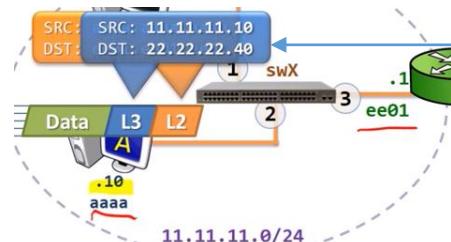


La risposta arriva correttamente ad A perché lo switch legge nel frame ARP l'indirizzo MAC di destinazione aaaa.aaaa.aaaa che possiede già. Adesso A conosce il MAC del router contenuto nel MAC SRC del frame, quindi aggiorna la sua tabella ARP.

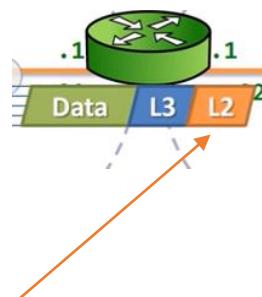
Host A – ARP Table	
11.11.11.1	ee01

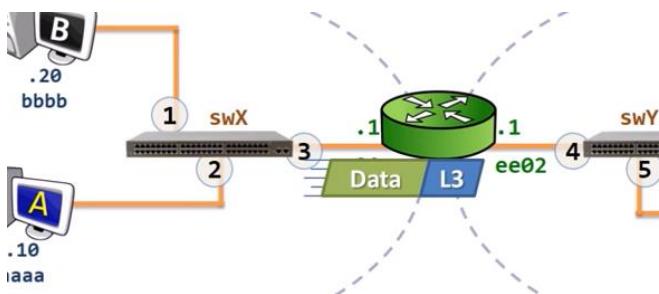


Adesso la scheda di rete ha il MAC address del router e quindi può costruire l'header. Non dimenticare che DEST IP è sempre quello dell'host destinatario e non del router!



Il frame arriva allo switch che legge l'indirizzo MAC del router e lo inoltra a lui.





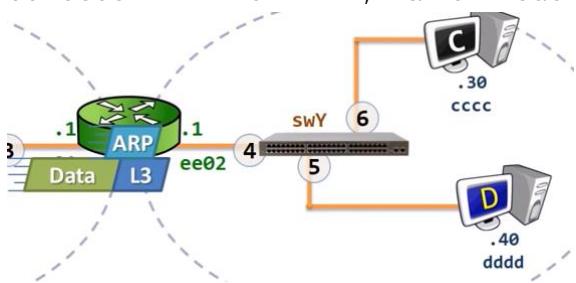
Quando il router riceve il frame legge l'header e capisce dal MAC DEST di essere proprio lui il destinatario, così porta il datagramma (Data+L3) a livello di rete per elaborarlo.

NB: quando A invia al router il pacchetto, in realtà è come se lo stesse inviando a una delle sue interfacce di rete. Immagina ogni interfaccia di rete di un router come un host esattamente uguale agli altri (infatti ha un indirizzo IP 11.11.11.1 e un MAC address eee01). Il router avendo due interfacce di rete allora è come se avesse due host.

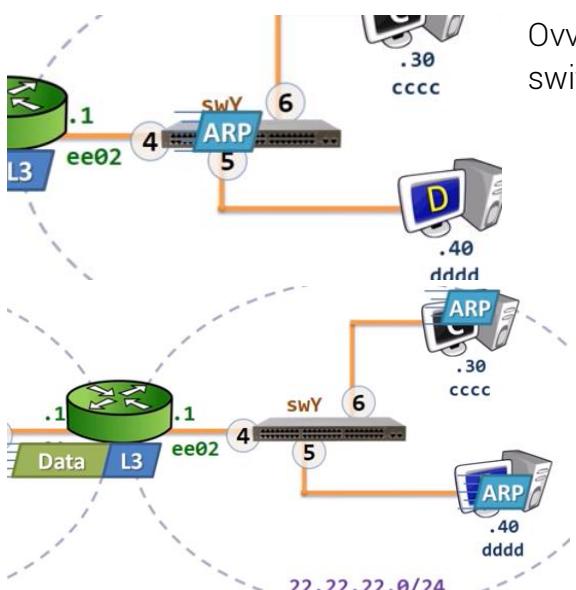
Router – Routing Table			
eth1	11.11.11.0/24	DC	
eth2	22.22.22.0/24	DC	

Leggendo L3 capisce che l'IP DEST appartiene alla sottorete 22.22.22.0 quindi inoltra il pacchetto alla sua interfaccia di rete che (come fosse un host) appartiene alla LAN di indirizzo 22.22.22.0, ossia l'interfaccia di rete di MAC address eth2.

Fatto questo è come se ci trovassimo in un host che è collegato con C e D. Questo host conosce l'indirizzo IP di D, ma non il suo MAC address.



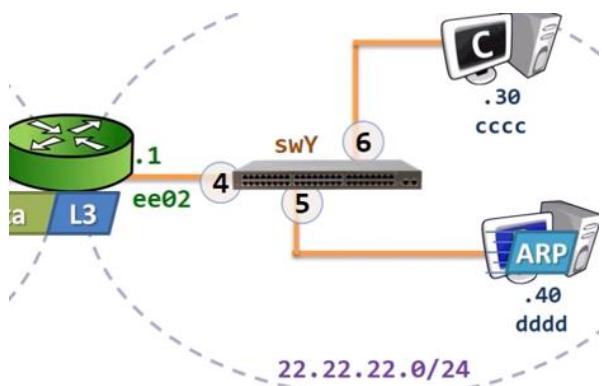
Quindi invia un pacchetto ARP di richiesta.



Ovviamente arrivando allo switch, ed essendo la sua switch table vuota, si riempie autonomamente:

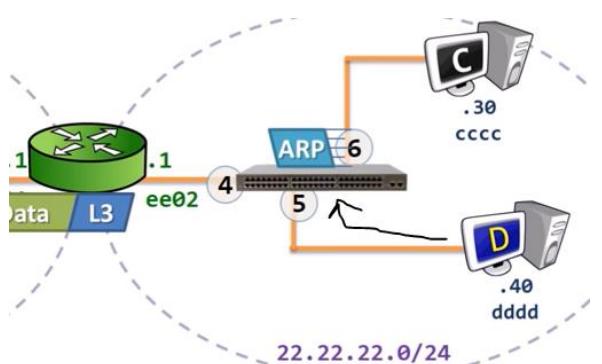
Switch Y – MAC Address Table	
4	eee02.ee02.ee02

Il messaggio viene inviato in broadcast.



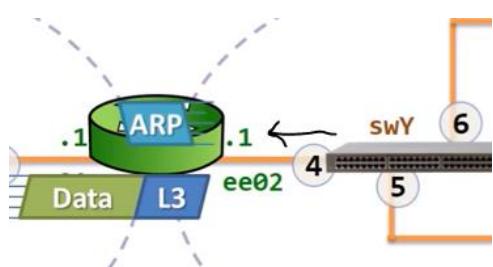
Solo l'host D accetta perché riconosce il proprio indirizzo IP e ne approfitta per riempire anche la propria tabella ARP:

Host D – ARP Table
22.22.22.1 ee02



Al solito lo switch riceve da una porta che non ha ancora alcuna corrispondenza e si autocompleta:

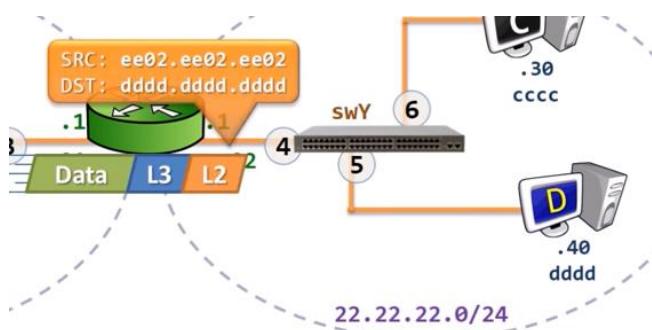
Switch Y – MAC Address Table
4 ee02.ee02.ee02
5 dddd.dddd.dddd



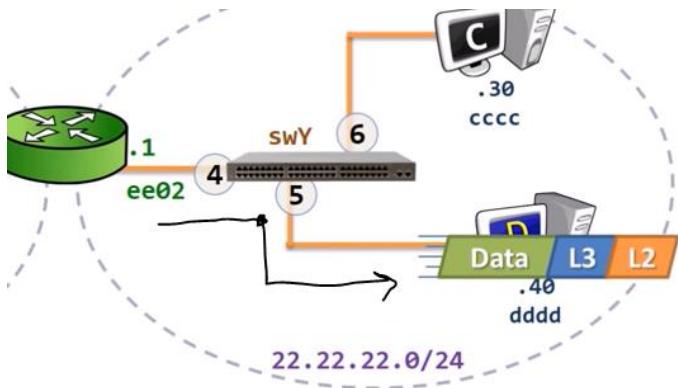
Inoltre lo switch riesce a inoltrarlo al router (o meglio alla sua interfaccia di MAC ee02 perché ha letto il MAC DEST nell'pacchetto ARP e consultato la switch table

Router – ARP Table
11.11.11.10 aaaa
22.22.22.40 dddd

A questo punto il router riesce a capire dall'ARP di risposta che il MAC address di D è dddd e così completa la sua tabella di ARP.

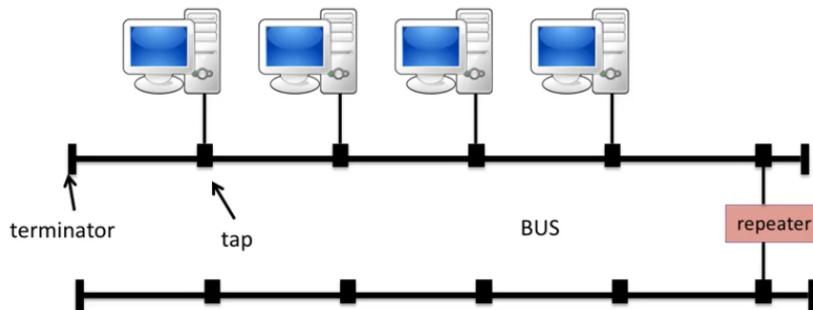


Adesso conoscendo il MAC di D crea il frame con l'header riempito del MAC di D e lo invia allo switch che già ne conosce la porta di uscita grazie alla propria switch table.



PROTOCOLLO ETHERNET

All'inizio il protocollo Ethernet si basava su un mezzo fisico condiviso da varie stazioni.



Attraverso dei **tap**, ossia delle interfacce verso un unico canale broadcast implementato tramite cavo coassiale, potevano immettere dei dati.

Il problema è quello di *accesso multiplo*, quindi collisioni e altro che già abbiamo visto. Negli anni 90 invece siamo passati dalla struttura in cui tutti sono collegati allo stesso canale ma con **topologia a stella** tramite HUB. Ovviamente tale tipologia è più gestibile rispetto a prima in cui si aveva un unico cavo che percorreva tutti gli host della LAN, ma anche con gli HUB sappiamo che, operando a livello fisico, è come se comunque la situazione non sia cambiata (non si evitano le collisioni).

Una successiva innovazione fu di usare le **fibre ottiche** (*Fast Ethernet*), ma qui solo per migliorare la performance (rimane comunque l'uso dell'HUB e quindi i problemi che non risolve).

Cos'è quindi questo protocollo Ethernet? Il protocollo Ethernet stabilisce il formato con cui inviare i dati a livello di collegamento in modo tale che possa essere ricevuti ed elaborati correttamente. Abbiamo visto nell'inoltro IP che quando il livello di rete passa alla scheda di rete il datagramma, questa compone il suo **frame** e lo manda al livello fisico. Vediamo dunque questo pacchetto frame che struttura possiede:

Preamble	DEST	SRC	TIPO	DATA	CRC
----------	------	-----	------	------	-----

Dove:

- **Preamble (8 byte)**: i primi 8 byte del frame sono il preamble dove 7 dei quali sono 10101010 e l'ottavo è 10101011 e servono per risvegliare la scheda di rete dei riceventi e sincronizzare i loro clock con quello del trasmittente. Gli ultimi due bit dell'ottavo servono però solo ad avvertire la scheda ricevente che le cose importanti stanno arrivando. Ma a cosa servono questi bit? Il trasmittente può trasmettere a 10,100,1000 bps ma questo non vuol dire che sulla LAN ethernet i dati viaggeranno a questa velocità. Quindi questi byte servono al ricevente a prepararsi al livello di ritmo a quelli che sono effettivamente il rate degli invii (quindi è come se fossero bit utili solo a sincronizzarsi prima che arrivino i dati buoni).
- **Dest (6 byte)**: indirizzo di destinazione MAC. Se il ricevente vi legge il proprio indirizzo allora *lo trasferisce al livello di rete*, altrimenti *lo scarta* (può capitare per esempio durante gli invii broadcast).
- **Src (6 byte)**: contiene l'indirizzo MAC di sorgente.
- **Tipo (2 byte)**: per quanto utilizziamo IP come protocollo di rete, non è così scontato che si utilizzi questo. Pertanto il ricevente al livello di collegamento deve sapere qual è il protocollo di rete superiore che si sta usando. Per IP il tipo è 0800 (esadecimale). Però sappiamo che un frame può essere utilizzato solo al fine di **pacchetto ARP**, e per questo si usa il codice 0806 (esadecimale).
- **CRC (4byte)**: ospita il *resto* ottenuto applicando il CRC che servirà per controllare l'integrità dei dati.

Il protocollo Ethernet fornisce un servizio **non affidabile** in quanto quando il frame arriverà al ricevitore B possono succedere due cose:

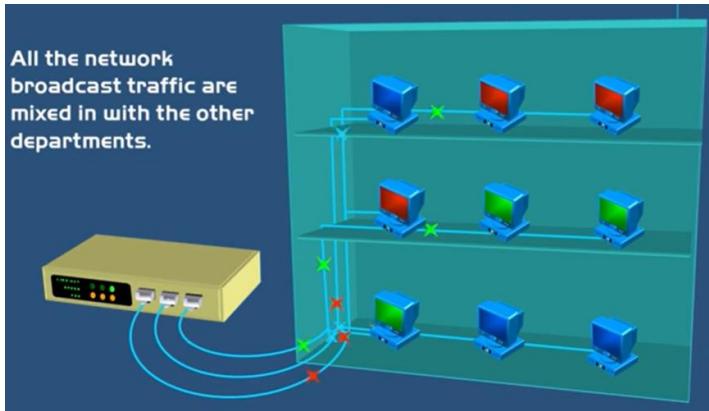
- 1) Il frame riporta un campo MAC address in dest che corrisponde al ricevente. Quindi viene trasportato al livello di rete.
- 2) Il frame riporta un campo MAC address in dest che *non* corrisponde al ricevente. Quindi viene scartato.
- 3) Il frame riporta un CRC tale che si rileva un errore sull'integrità dei dati.

In **tutti questi casi** il protocollo Ethernet non avvisa il mittente dell'esito tramite ack e quindi il mittente (affidandosi al solo protocollo Ethernet) non potrà mai sapere l'esito dell'invio. Ma sappiamo che questo non è un problema se si usa TCP al livello di trasporto.

Inizialmente, quando si usava la topologia a bus o a stella con gli hub, il protocollo Ethernet comprendeva il CSMA-CD per affrontare il problema delle collisioni. Adesso, con la topologia a stella ma con l'uso di switch, usando la commutazione di pacchetto store and forward tali collisioni sono evitate nelle LAN completamente commutate.

VLAN: Virtual Local Area Network

Cos'è una VLAN e qual è la differenza con la LAN che conosciamo?



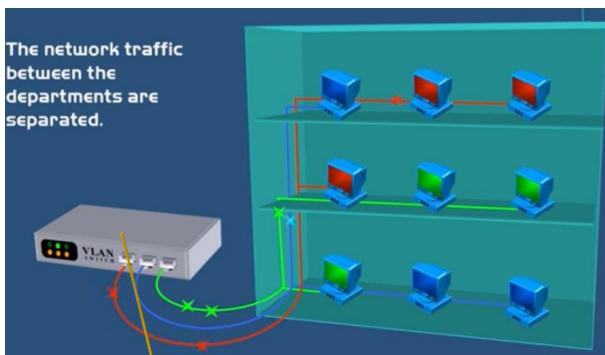
Giallo = host dirigenti

Siamo abituati a una LAN realizzata tramite SWITCH. Tutti gli host vengono collegati alle porte della switch. In questa figura abbiamo 3 tipi di host (blu, rosso e giallo). Questi 3 tipi potrebbero essere tipi di host che con gli altri due gruppi non hanno nulla a che fare.

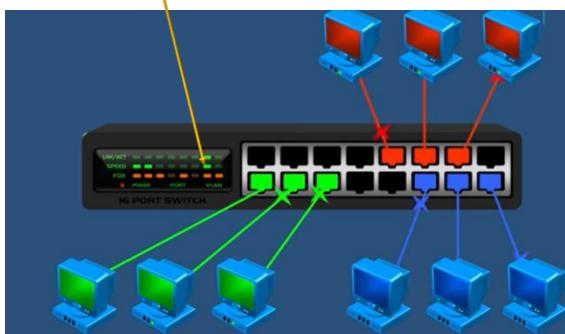
Per esempio rosso=host dipendenti
Blu= host clienti

Quando qualcuno di loro invia un messaggio di broadcast lo fa logicamente perché vorrebbe comunicare con gli altri *dello stesso gruppo*. Eppure sappiamo che uno switch manda i messaggi in broadcast a tutti gli host a essa collegata.

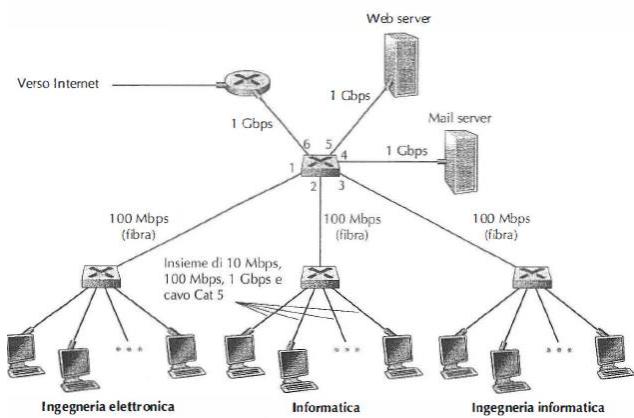
Sarebbe bello creare 3 gruppi logicamente legati tra loro (per ruoli/scopi/...) e veicolare il traffico solo tra questi gruppi. Vorremmo quindi creare dei **gruppi virtuali** o meglio delle **LAN Virtuali (VLAN)**. Questo è possibile tramite **un particolare SWITCH che supporti la virtual local area network**.



Questo SWITCH speciale riesce a creare tre canali virtuali.



In una LAN basata sulle **porte**, queste vengono divise in gruppi e a ciascun gruppo è riservata una VLAN. Per esempio le porte 9,10 e 11 sono riservati ai verdi. Se un host si collega a quella porta (anche se qui sono occupate già tutte per ogni LAN), si unisce alla VLAN.



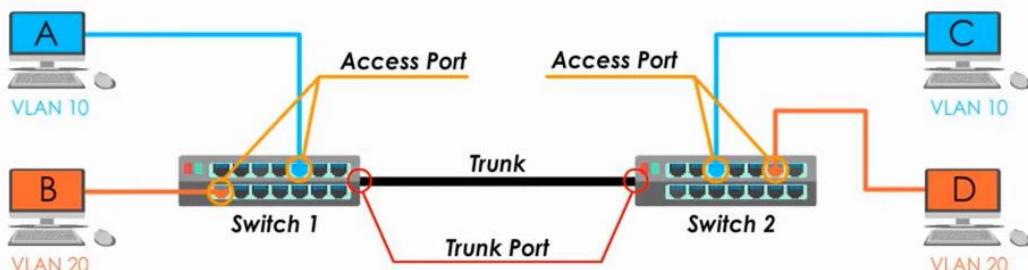
Supponiamo quindi di avere questa LAN. Posso sostituire i 3 switch con uno solo che supporti la VLAN e organizzare le porte in modo tale da crearne tre VLAN.

Lo switch che supporta la VLAN viene programmata tramite un **software per la gestione dello Switch** in cui programma una tabella di associazione tra porte e VLAN. A livello hardware quello che fa è di consegnare i frame alle rispettive VLAN.

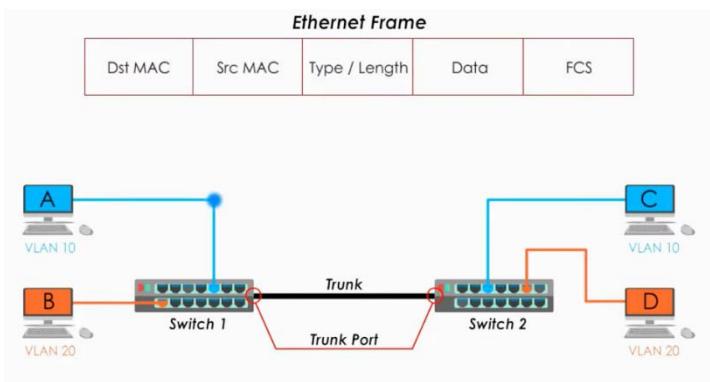
TRUNKING E TAGGING



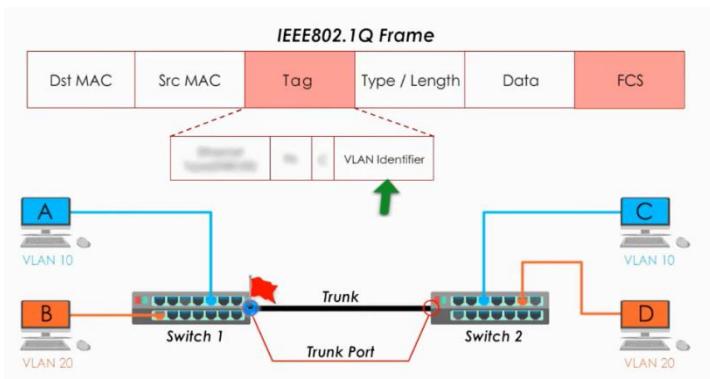
Supponiamo di avere due switch VLAN separati i quali ospitano due host ciascuno. I due host appartengono a una VLAN (celeste e arancio). Quindi abbiamo che A e C sono logicamente legati perché appartengono entrambi alla VLAN 10 ma non lo sono fisicamente; lo stesso per B e D. Come facciamo a far comunicare due host appartenenti alla *stessa* VLAN ma collegati a switch VLAN *diversi*?



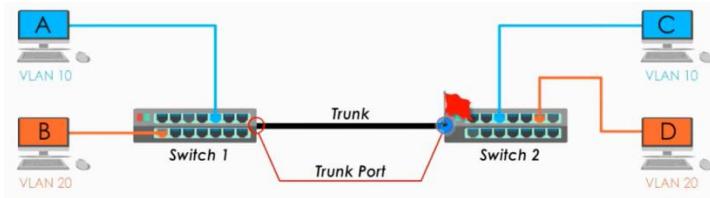
Si utilizza la **tecnica di trunking** in cui due switch VLAN vengono collegati tramite cavo su due porte speciali chiamate **trunk port** (le altre vengono chiamate *access port*). Quando qualcuno di una VLAN X manda un frame, tale frame viene ricevuto come sappiamo dagli stessi host appartenenti alla VLAN X (da tutti se broadcast, da uno particolare altrimenti). Tramite la trunk port invece passano tutti i frame scambiati in uno switch VLAN.



Tieni a mente però che se A manda un frame, tale frame segue ancora lo standard Ethernet visto prima. Anche se dovesse attraversare il collegamento trunk e arrivare all'altro switch, come fa lo switch a sapere a quale VLAN appartiene i frame in arrivo dal primo switch?



TPID con valore fisso a 81-00 (esadecimale) e alla fine del frame un campo FCS' che sarebbe il CRC ricalcolato.

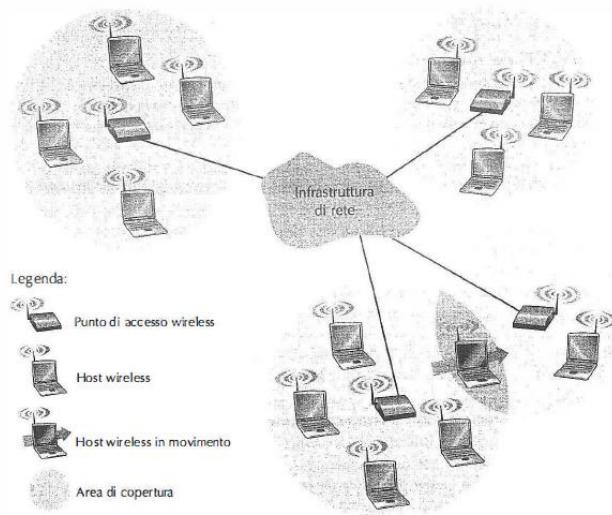


lo invia alla porta corretta.

Si usa un **formato esteso** di quello conosciuto in cui con la tecnica di **tagging** si inserisce una **etichetta VLAN** che in un particolare campo specifica la **VLAN identifier**, ossia l'identificativo della VLAN a cui appartiene il frame. In particolare il campo tag è anche costituito da un

Ricorda però che l'etichetta serve solo allo switch 2 e non all'host C, quindi una volta arrivato allo switch 2, questo ne rimuove l'etichetta riportando il formato del frame a quello standard e

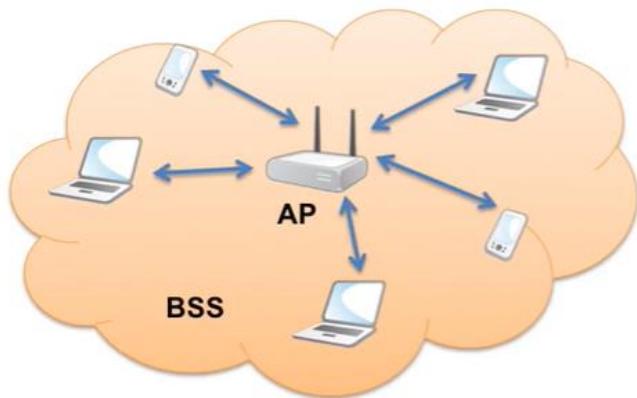
COLLEGAMENTO WIRELESS



Un host effettua un **collegamento wireless** verso una stazione base quando il collegamento è senza fili. Una **stazione base** è un componente che permette all'host ad essa collegata, ossia che si trova nella sua *area di copertura*, di trasmettere e ricevere dati da/a qualsiasi altra parte; ne fa quindi da mediatore. Un esempio sono i **ripetitori di cella** (nelle reti cellulari) oppure gli **accesso point** nelle LAN. Quando un host wireless si sposta da un area di copertura ad un'altra cambia l'access point, e questo processo si dice **handoff**.

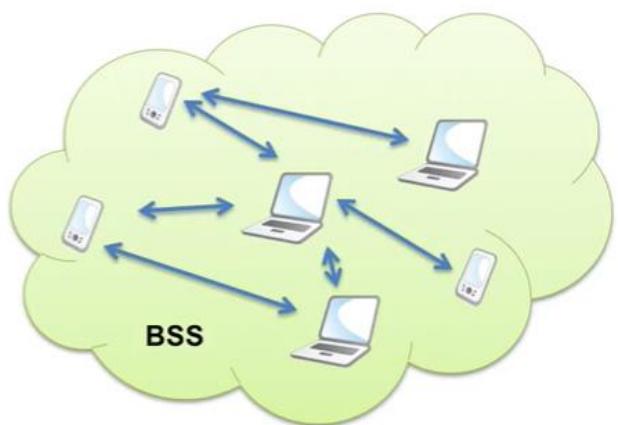
Le reti vengono poi distinte in due categorie:

con infrastruttura



Queste tipologie di reti forniscono i servizi tradizionali di rete (come instradamento, indirizzamento) ai loro host wireless grazie alla stazione base (access point **AP**).

senza infrastruttura (o ad hoc)



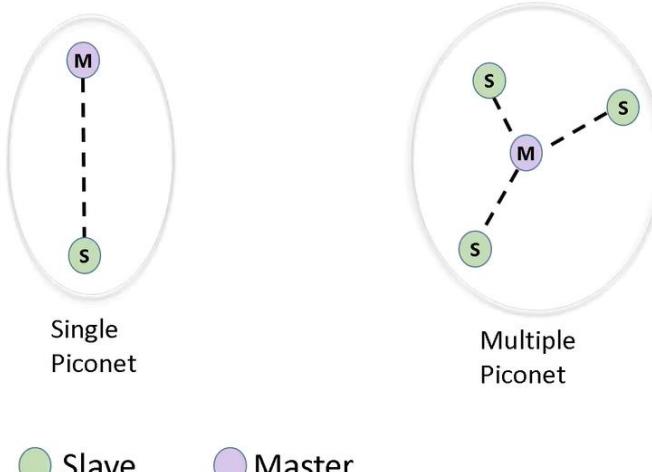
Queste tipologie di reti non hanno alcuna stazione base a cui si connettono, quindi devono da loro prevedere i servizi di instradamento, indirizzamento, DNS e altri. Un esempio è la connessione bluetooth.

Che siano centralizzate o decentralizzate le infrastrutture, questa collezione di stazioni che vogliono comunicare tra loro vengono chiamate **BSS** ossia Basic Service Set.

Dobbiamo riflettere su una cosa. Alla fine tra rete wireless e cablata cambia solo il mezzo attraverso cui passano i dati. Pertanto basterà sostituire la scheda di rete Ethernet con la scheda di rete wireless e lo switch con l'accesso point. Pertanto dal livello di rete in su **tutto resta uguale**, mentre **cambia solamente** dal livello di collegamento in giù.

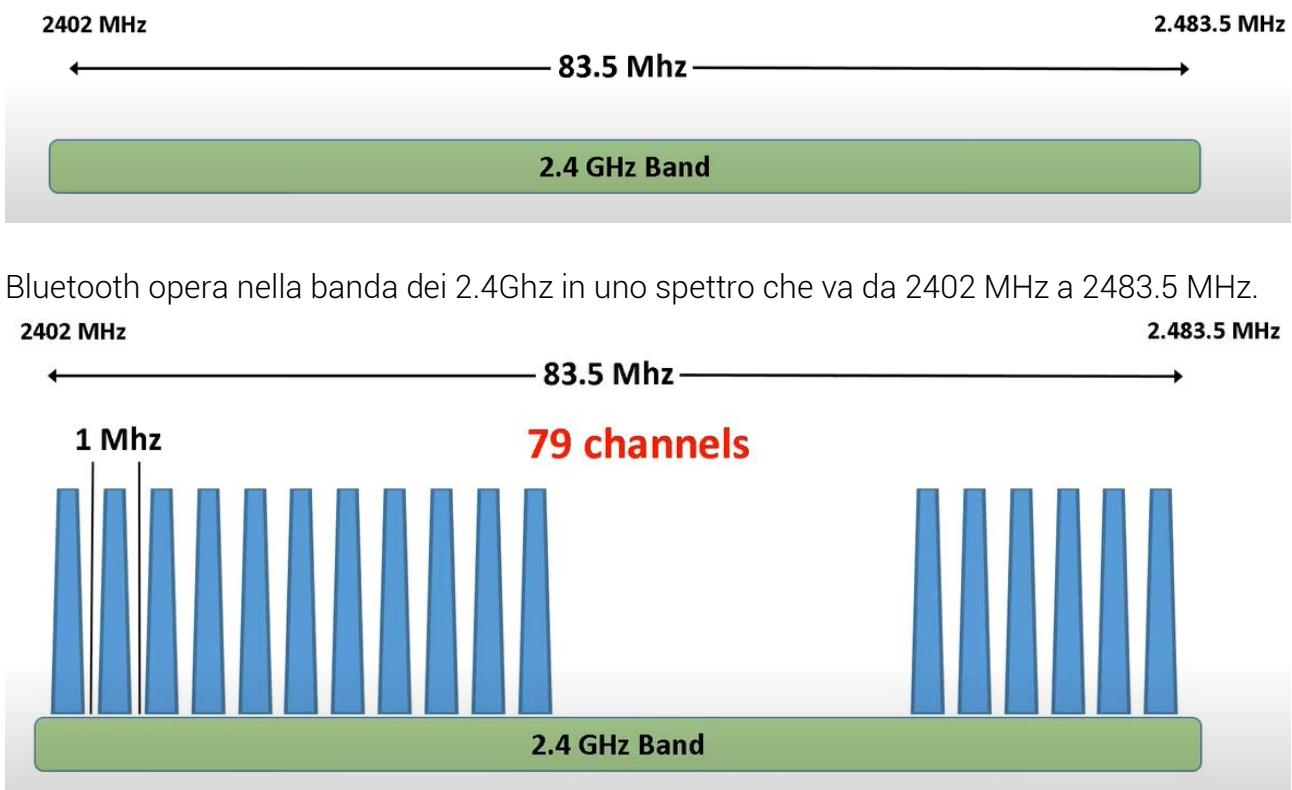
BLUETOOTH

E' una tecnologia a onde radio che permette la connessione wireless tra dispositivi a corto raggio. Esistono essenzialmente due topologie di architetture bluetooth: piconet e scatternet.

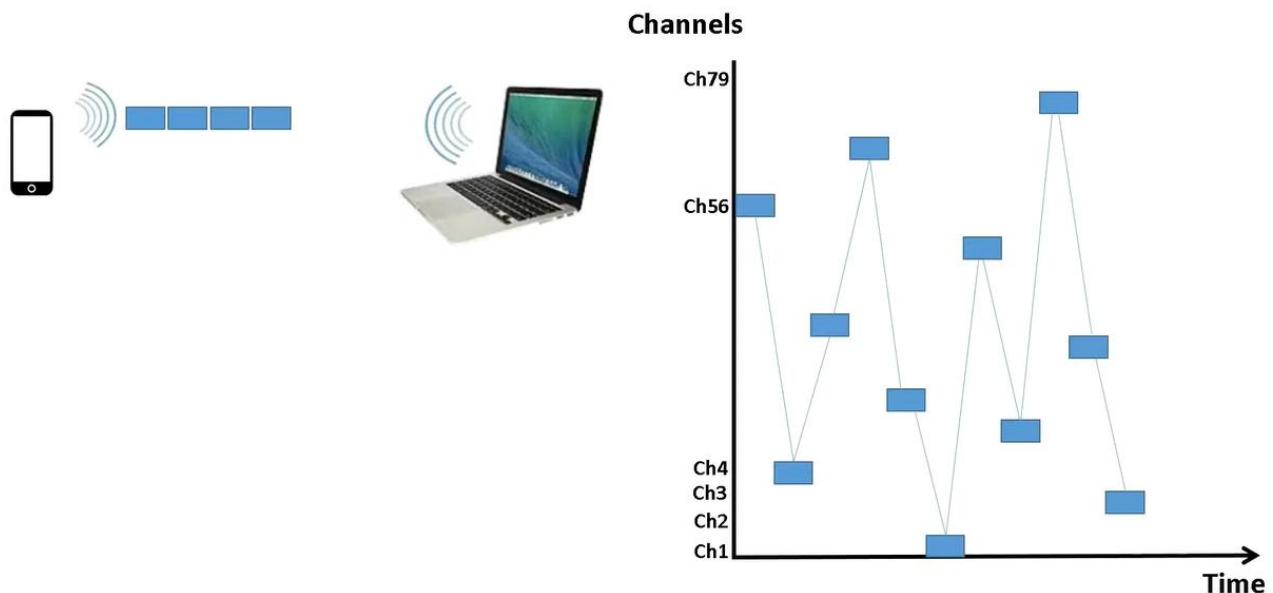


Nella piconet la comunicazione può essere singola (ossia tra due dispositivi) o multipa (tra più dispositivi). In ogni caso ci sarà sempre un solo master che deciderà quando uno slave può trasmettere o meno e nessuno slave può comunicare con nessun altro slave. Nella piconet sono possibili al massimo gruppi da 8 dispositivi comprensivi del master.

Le scatternet invece sarebbero gruppi di piconet collegati tra loro. E' bene puntualizzare che nelle scatternet un master per una sua piconet potrebbe essere uno slave per un'altra piconet. Un esempio è quello in grigio che è master per la sua mentre è slave per quella centrale.



Questo spettro è diviso in 79 canali di un MHz di banda l'uno. Per la trasmissione utilizza TDMA con slot da 625 microsecondi nei quali viene cambiato il canale in modo pseudo-casuale. Questa tecnica è chiamata FHSS e per maggiore chiarezza la spiegherò meglio con un esempio:



Nella figura abbiamo uno smartphone collegato via bluetooth a un pc che invia una serie di pacchetti. In TDMA sappiamo che tali pacchetti vengono messi in degli slot. Pertanto ogni 625microsecondi un pacchetto viene inviato. La tecnica dell'FHSS consiste nel inviare il primo pacchetto scegliendo un canale random (per esempio Ch56), poi il secondo scegliendone un altro ancora (Ch4) e così via.

Ma qua'è il beneficio di una trasmissione del genere?

Intanto che evita l'interferenza con reti che utilizzano sempre uno stesso canale trasmittivo e poi per complicare la vita a un hacker che si vede cambiare la frequenza ogni 625 microsecondi.

WiFi: LAN WIRELESS

Esistono 3 versione di WiFi base definiti dallo standard IEEE 802.11:

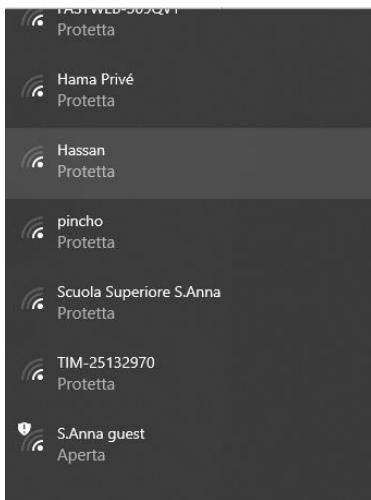
Standard	Gamme di frequenze (negli Stati Uniti)	Velocità di trasferimento dati
802.11b	Da 2,4 a 2,485 GHz	Fino a 11 Mbps
802.11a	Da 5,1 a 5,8 GHz	Fino a 54 Mbps
802.11g	Da 2,4 a 2,485 GHz	Fino a 54 Mbps

La 802.11b e la 802.11g nonostante usano la stessa banda di frequenza, la g utilizza una diversa tecnica al livello fisico che gli permette

di raggiungere una velocità di trasferimento dari fino a 54 Mbps.

Essi utilizzano il *protocollo CSMA/CA* che vedremo dopo.

Prima di inviare dati, le stazioni del BSS devono *associarsi* a un AP. Un AP è istallato da un amministratore che gli assegna un identificativo (nome della rete) chiamato **SSID**.



Esempi di SSID

Questi sopra sono tutti AP raggiungibili e associabili. Quando ci "connettiamo al wifi" non stiamo facendo altro che associarci a un AP.

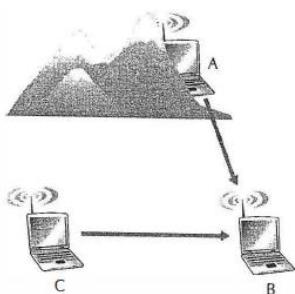
Abbiamo detto che ciò che cambia dal passaggio alla rete cablata Ethernet alla rete wireless è fondamentalmente il livello di collegamento. Abbiamo infatti una leggere modifica ai **frame**:

Controllo del frame	Durata	Indirizzo1	Indirizzo2	Indirizzo3	Numero di sequenza	Indirizzo4	payload	CRC
---------------------	--------	------------	------------	------------	--------------------	------------	---------	-----

Dove:

- **Controllo frame (2 byte)**: specifica varie cose (tipo protocollo, cosa stiamo guardando, informazioni energetiche (è una tecnologia radio) e altre informazioni di controllo)
- **Durata (2 byte)**: qui viene scritta la durata della trasmissione che andrà in corso a breve.
- **Indirizzi1,2,3,4 (6 byte)**: ho bisogno di 4 campi di indirizzo perché nel caso peggiore un terminale (indirizzo1) che è associato a un AP (indirizzo2) dovrà parlare con un altro terminale (indirizzo3) che è associato a un altro AP (indirizzo4). Quindi nel caso peggiore avrò bisogno di *4 indirizzi MAC*.
- **Numero di sequenza (2 byte)**: serve per identificare in maniera univoca il frame.

Perché ho bisogno di un numero di sequenza? A causa della natura elettromagnetica del WiFi, esso è soggetto a interferenze lungo il tragitto che potrebbero causare corruzioni così spesso che avremmo bisogno di un **protocollo di trasferimento affidabile anche a livello di collegamento**.



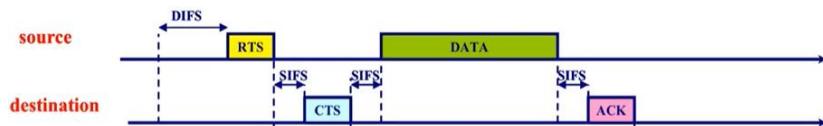
Un esempio è il **problema del nodo nascosto** in cui A propaga un segnale che per sua natura può essere alterato da ostacoli fisici come anche una montagna. In questi casi A e C non saprebbero della relativa esistenza e quindi potrebbero "sentire" un canale sempre libero. (vedi CSMA/CA).

Inoltre il CRC del WiFi è molto più potente come codice di quello Ethernet.

CSMA/CA: IL PROTOCOLLO MAC PER IL WiFi

Abbiamo capito che diverse stazioni possono collegarsi senza fili a *un unico access point condiviso*. Questo implica che il canale è condiviso e quindi potrebbe essere soggetto a trasmissioni contemporanee da diverse stazioni. Così come per le reti cablate condivise, anche qui c'è bisogno di un *protocollo MAC o protocollo ad accesso multiplo* per coordinare le varie stazioni. Per il WiFi si utilizza un particolare protocollo MAC che si chiama **CSMA/CA** che sta per CSMA (quello che già conosciamo) con **prevenzione di collisione (collision avoid)** perché a differenza delle reti cablate come Ethernet che hanno una distorsione del segnale tenue e dunque ci si può accorgere delle collisioni perché appunto sono segnali si "spessore" simile a quelli trasmessi (quindi una stazione riesce a sentire che quello che gli arriva mentre sta trasmettendo è un segnale di collisione), per le reti wireless che sono soggette a forti attenuazioni del segnale, anche se A sta trasmettendo e B lo stava già

facendo da prima, il segnale di B potrebbe arrivare così distorto ad A che questo non "valuterebbe" quel segnale come una collisione.



La situazione iniziale è che nel caso generale molte stazioni stanno mandando pacchetti al ricevitore.

- 1) Quando vuole mandare ascolta il canale. Se è libero aspetta DIFS prima di inviare RTS contente il tempo in cui vuole rimanere connesso per la trasmissione di DATA. Se dopo DIFS il canale risulta ancora libero allora trasmette RTS e aspetta CTS che gli sarà mandato dal l'AP. Gli altri che riceveranno CTS capiranno che per quel tempo richiesto nel RTS dovranno stare fermi. Aspetta SIFS e trasmette aspettando dopo SIFS l'ACK. Se non riceve l'ack ritenta usando il protocollo fino a quando dopo un tot di volte si arrende e scarta il frame.
- 2) Se il canale non è libero attiva un timer casuale che rientra nel range $[0, 2^n - 1]$ con n numero n-esimo del tentativo di trasmissione. Tale timer viene decrementato ogni volta che si rileva il canale inattivo mentre si congela ogni volta si rileva la trasmissione di qualcuno. Scaduto tale timer il canale è sicuro libero e trasmette.



A un certo punto uno di loro chiede al ricevitore AP (Wireless AP) che è *pronto per mandare (ready/request to send)* e lo fa inviando un pacchettino chiamato **RTS**. In realtà tale pacchetto viene inviato in *broadcast* a tutti anche se è indirizzato al ricevitore. Questo RTS è una richiesta al ricevitore di poter occupare il canale per un tot di tempo; infatti in RTS c'è scritto

anche il tempo totale in cui tenere il canale occupato.



Quando il ricevitore può, manderà in *broadcast* a tutti un pacchetto **CTS** (*clear to send*) che sarà interpretato da chi ha mandato l'RTS come il segnale che il canale è libero e può inviare pacchetti, e da tutti gli altri come il segnale che devono stapparsi per un tot di tempo.

RTS e CTS sono pur sempre pacchetti inviati in un canale condiviso quindi anche loro potrebbero generare collisioni ma questi pacchetti sono così tanto piccoli che la possibilità di collisione è bassissima.

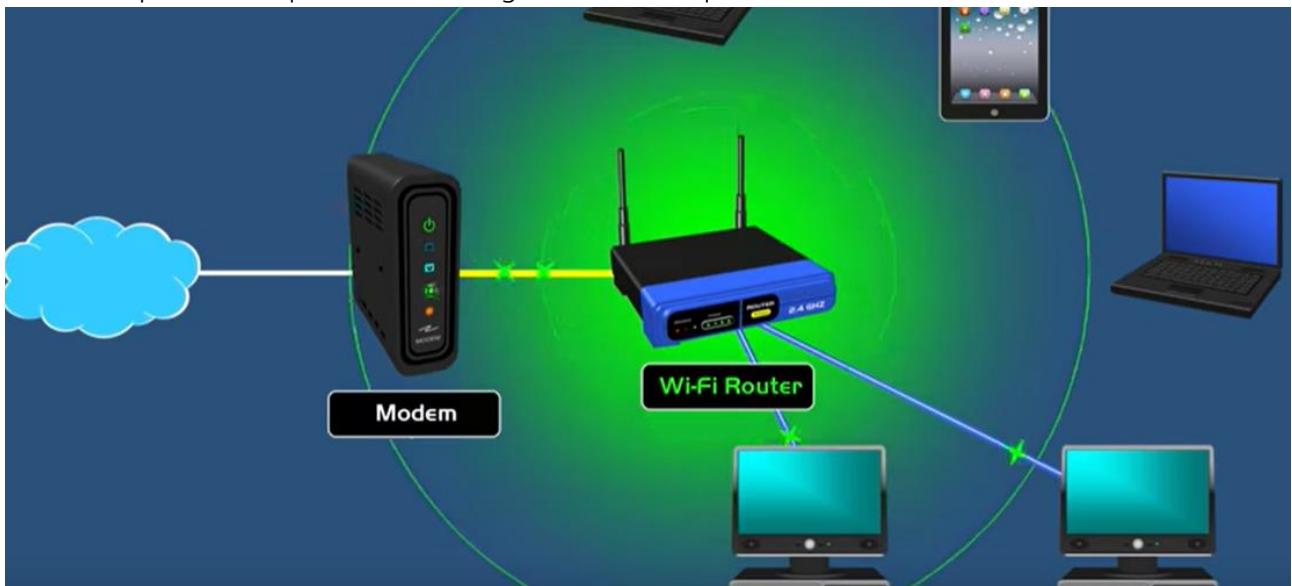
NB: la probabilità di collisione tra RTS/CTS e gli altri pacchetti è comunque ancora di più diminuita dal fatto che si esegue sempre il *carrier sensing*, ossia prima di mandare RTS e CTS si ascolta comunque il canale.

Il periodo di tempo dopo il quale il trasmittente, accortosi che il canale è inattivo, manda l'RTS è chiamato **DIFS**. Di contro l'AP, una volta ricevuto l'RTS aspetta un tempo **SIFS** prima di mandare il CTS. Il trasmittitore aspetterà anche lui SIFS di tempo e poi trasmetterà i dati, che quando il ricevitore li riceverà tutti aspetterà SIFS di tempo per mandare l'**ACK**.

Il CSMA/CA risolve il problema del *terminale nascosto* grazie al fatto che se una stazione non arriva a sentire un'altra, allora ci penserà l'AP col suo CTS ad avvertire all'altra di fermarsi.

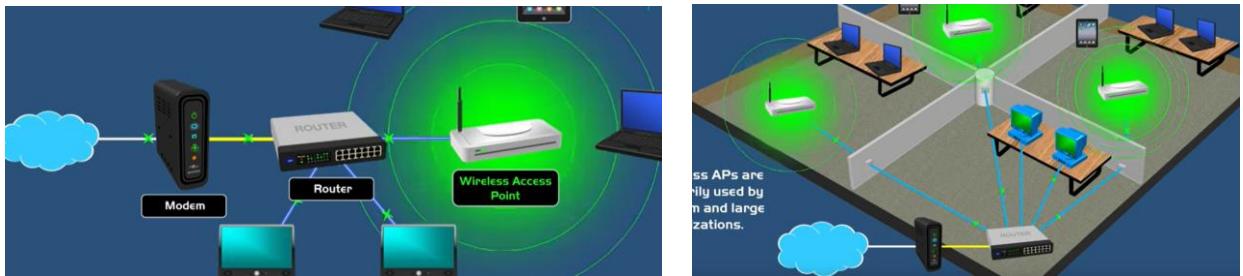
WAP: Wireless AP

Abbiamo parlato di questo AP o meglio WAP. Ma qual è la **differenza col router**?

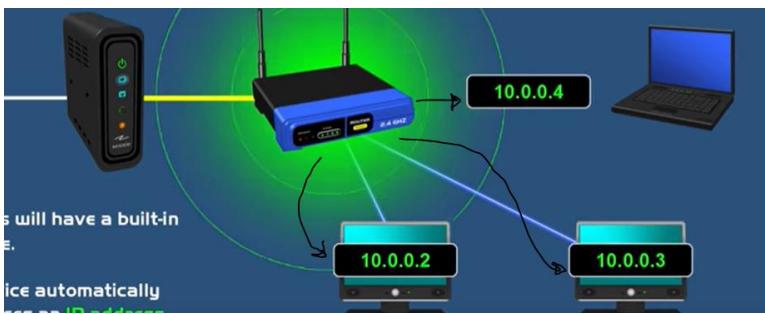


Essenzialmente il router è quello che fa il grosso del lavoro. Riesce a connettere in wireless diversi dispositivi fornendo pure la possibilità di connettersi tramite Ethernet in quanto spesso contiene all'interno SWITCH.

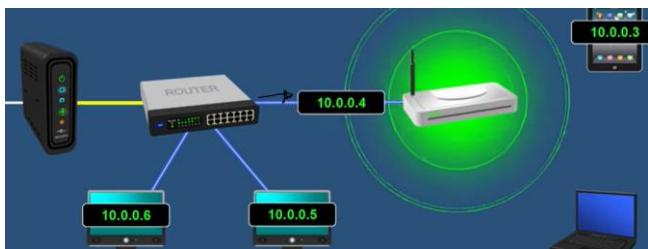
Ma cosa succede se alcuni dispositivi sono troppo lontani dal segnale del router? Dobbiamo acquistarne un altro?



Assolutamente no. Ci pensa il WAP che funge da **ripetitore** e nient'altro. Il WAP corrisponde a uno switch wireless quindi è di livello 2.



Il router spesso fa da server DHCP per fornire direttamente gli indirizzi IP a chi ne fa richiesta.



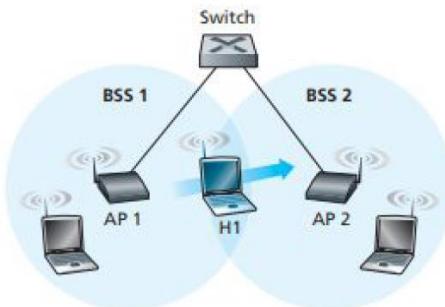
L'AP può invece fungere solamente da canale di comunicazione attraverso cui veicolare gli indirizzi IP assegnati dal router.

MOBILITA' IN IP

(vedi il paragrafo sopra e l'autoapprendimento degli switch per avere più chiaro quanto segue)

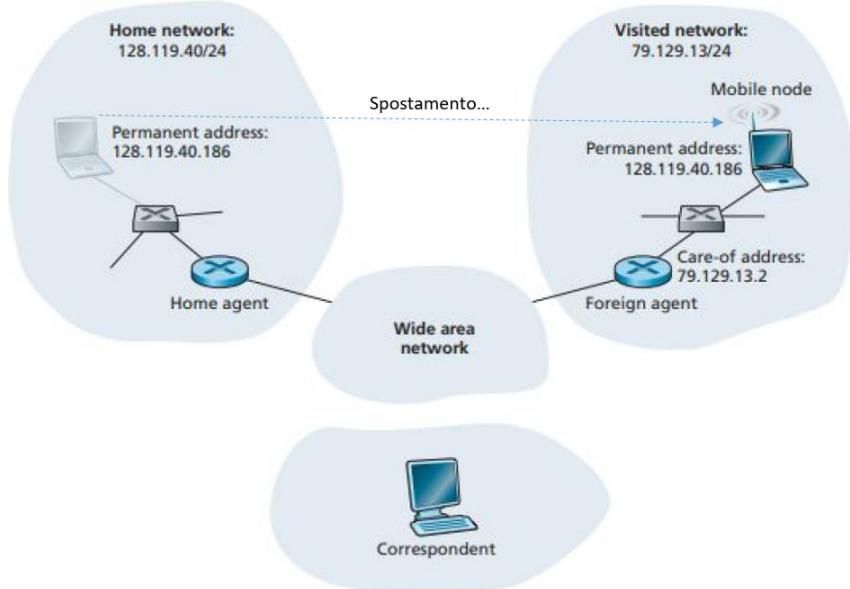
Cosa succede quando un host si muove nella rete? Consideriamo due casi:

MOVIMENTO ALL'INTERNO DELLA STESSA SOTTORETE



H1 si sta muovendo da BSS (che comunica con AP1) a BSS2 che comunica con AP2. Cosa succede? Nulla. H1 continua a mantenere lo stesso indirizzo IP in quanto AP1 e AP2 sono solo dei ripetitori di livello 2, non c'entrano nulla (non sono router). Quindi allontanandosi, il segnale da H1 a AP1 e viceversa stenta ad arrivare fino a quando AP2 (spesso con lo stesso SSID di AP1) diventa la stazione con segnale più forte. Grazie all'autoapprendimento degli switch, il fatto che H1 comunichi adesso con AP2 non crea danni perché lo switch cambierà la porta di ingresso da cui riceve i dati in quanto vede arrivarsi l'indirizzo MAC di H1 da un'altra porta.

MOVIMENTO DA UNA SOTTORETE A UN'ALTRA



NB: qui faremo riferimento a un *nodo mobile* attaccato via ethernet ma la stessa cosa equivale a un nodo mobile collegato via wireless.

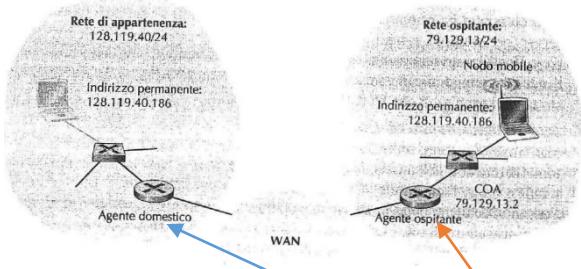
Si definisce **rete appartenente** quella rete a cui "di solito" quell'host è collegata (es. la rete di casa nostra). Si definiscono **agenti domestici** tutte le entità della rete appartenente.

Di contro si definisce **rete ospite** quella che sta ospitando il nodo mobile e **agenti ospiti** quelle entità che permettono della sua mobilità (es. la rete dell'università quando ci si va).

Si definisce poi **corrispondente** un qualsiasi host che vuole comunicare col nodo mobile.

Per capire come comportarsi in questi casi, bisogna affrontare il paragrafo *Indirizzamento* e a seguire quello dell'*istradamento* del nodo mobile.

INDIRIZZAMENTO DEL NODO MOBILE



Quando un nodo si sposta sulla rete ospite, questa deve avvertire tutti che il nodo si trova ospitato da lei. Perché? Ovviamente se un qualsiasi corrispondente volesse inviargli un pacchetto si *deve cambiare l'istradamento* che dovrà portare all'attuale posto dove si trova il nodo mobile. Quindi la rete ospite

comunicherà una *nuova rotta* aggiornando le informazioni di istradamento e le tabelle di inoltro, mentre la rete di appartenenza dovrà cancellare le vecchie informazioni di istradamento.

Particolare importanza ha l' **agente ospitante** il quale svolge due funzioni:

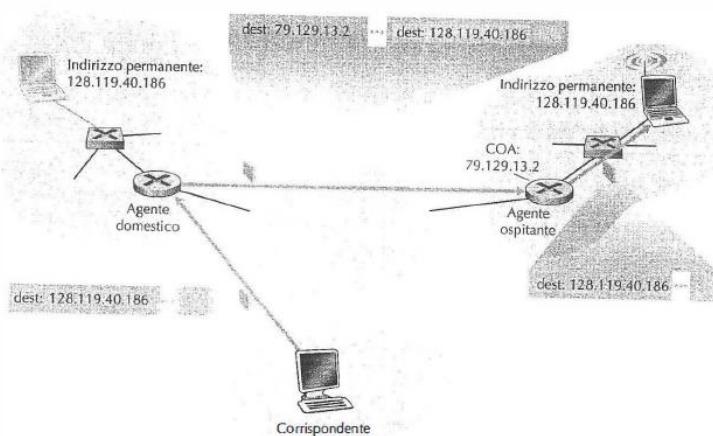
- 1) Definisce un **indirizzo di mediazione COA** da attribuire al nodo ospite. In pratica il nodo mobile avrà due indirizzi IP: l'**indirizzo permanente** (128.119.40.186) datogli dalla rete di appartenenza e un indirizzo IP COA (79.129.13.2).
- 2) È lui che informa la rete di appartenenza parlando con l'**agente domestico** di avere ospite il nodo e di conoscerne il COA.

Vediamo nel prossimo paragrafo come il pacchetto viene istradato verso il nodo mobile.

INSTRADAMENTO VERSO IL NODO MOBILE

Per adesso chi è a conoscenza di dov'è il nodo mobile sono solo i due agenti domestico e ospitante. Esistono due approcci di routing/istradamento per far sì che un datagramma a livello di rete giunga verso il nodo mobile.

ROUTING INDIRETTO



proprio indirizzo IP (dell'agente domestico intendo). Quindi incapsula il datagramma in un

Il corrispondente invia il frame come se il nodo mobile fosse nella sua rete. Quando l'agente domestico capisce dall'IP che tale datagramma è rivolto al nodo che adesso si trova in una rete ospite, incapsula tale datagramma in uno più grande:

COA	SRC	datagramma
-----	-----	------------

Quindi usa come indirizzo di destinazione COA per mandarlo all'agente ospitante e come SRC il

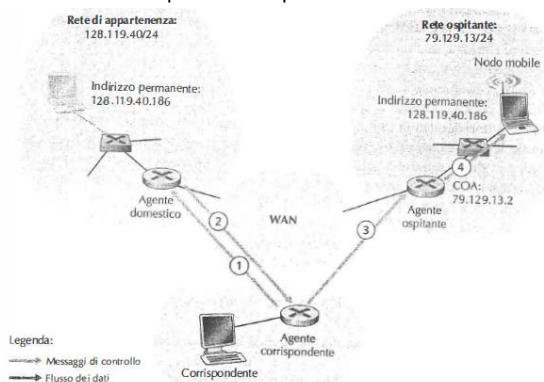
altro datagramma. Lo invia al livello di collegamento che inserirà il suo header per costruire il frame diretto verso l'agente ospitante nel modo classico che conosciamo.

Appena il frame arriva all'agente ospitante lo decapsula fino al livello di rete. Qui però non legge il datagramma in arancio ma quello in blu perché il datagramma originale è stato alterato dall'agente domestico. Allora capisce su quale uscita inviarlo ma prima di farlo lo decapsula di nuovo per ottenere quello originale a cui poi al livello di rete verrà creato il frame. Da qui in poi funziona come l'inoltro in IP (magari il router non conosce il mac del nodo mobile e contatta lo switch tramite ARP per averlo e così via..).

NB: Semmai invece volesse essere il nodo mobile a contattare il corrispondente ovviamente non c'è alcun problema. Continua ad usare il proprio IP e come destinatario quello del corrispondente.

ROUTING DIRETTO

Quando si esegue un routing indiretto il problema è quello dell'**instradamento triangolare**. Per capirci, supponiamo che il corrispondente sia giusto giusto all'interno della stessa rete ospite. Semmai volesse contattare il nodo mobile, che magari si trova anche accanto a lui di qualche cm, dovrebbe comunque contattare l'agente domestico che poi dovrebbe contattare quello ospite e così via.

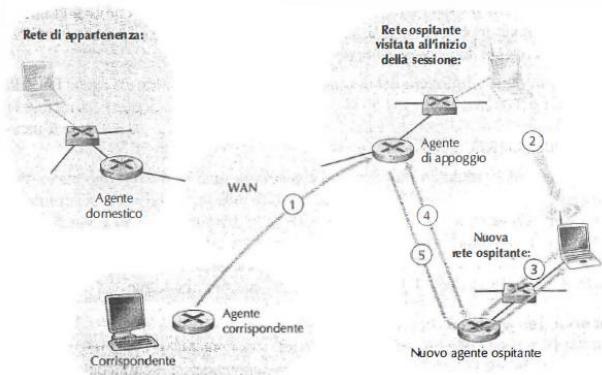


Il **routing diretto** permette invece a un corrispondente, tramite il suo **agente corrispondente** di ottenere direttamente il COA del nodo mobile e inviare il frame direttamente all'agente ospitante.

Il routing diretto porta però i seguenti problemi:

- 1) Un nuovo protocollo per far sì che l'agente corrispondente contatti quello domestico per ricevere il COA.
- 2) Supponiamo che il corrispondente ottenga il COA. Quando lo ottiene non ricontatta mica l'agente corrispondente per rifarselo dare ad ogni sessione. Quindi se il nodo mobile si dovesse spostare in un'altra rete ospite, per il corrispondente il COA rimane sempre quello di prima.

Questo si risolve con un protocollo usato nelle reti **GSM** e prevede di fare quanto segue:



Il Corrispondente continua a usare il COA, quindi utilizza sempre il routing diretto. Però se il nodo mobile dovesse cambiare rete ospitante allora quando riceve il COA dal **nuovo agente ospitante**, quest'ultimo dovrà avvisare l'**agente di appoggio** che il nodo ospite si trova da lui e che se qualcuno lo dovesse contattare deve reindirizzarlo (tramite sempre encapsulamento) dal nuovo agente ospitante.

IP MOBILE

Vediamo dunque più nel dettaglio quali sono questi protocolli per la mobilità. Vedremo quindi:

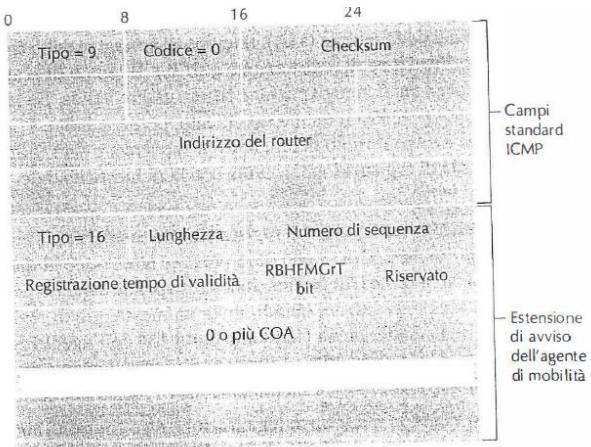
- 1) Protocolli per la ricerca dell'agente
- 2) Protocolli per la registrazione del COA presso l'agente domestico
- 3) Protocolli per l'instradamento/routing indiretto dei datagrammi

La 3) mette dunque in chiaro che nei protocolli IP per la mobilità si sceglie il *routing indiretto*.

RICERCA AGENTE

PROTOCOLLI PER LA RICERCA DELL'AGENTE

Quando un nodo mobile si muove tra reti diverse deve capire se si trova nella propria rete domestica o se si trova in una ospite. In ogni caso deve fare una **ricerca dell'agente**. Il nodo quindi invia una **richiesta dell'agente** o **solllicitazione dell'agente** ossia un messaggio in broadcast ICMP con tipo=10 che quando verrà intercettato dall'agente (domestico o ospite) lo contatterà in unicast con un altro messaggio ICMP chiamato **avviso dell'agente** che riporta:



- come tipo=9 (ricerca/scoperta router)
- Codice=0
- Checksum
- Indirizzo del router ossia il proprio indirizzo IP così che il nodo mobile può salvarselo

E poi un'estensione che riporta in particolare nei bit RBHFMGrT i seguenti significati:

- H indica che per quel nodo mobile è un agente domestico

- F indica che per quel nodo mobile è un agente ospite
- R indica che il nodo mobile deve registrarsi presso lui
- M e G indicano se oltre all'incapsulamento IP in IP si utilizzerà un'altra forma

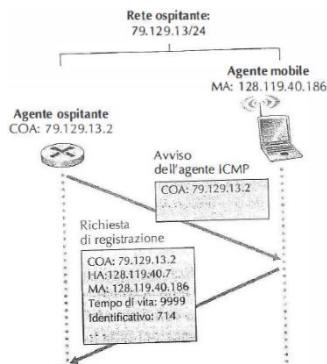
Poi abbiamo un campo:

- COA che elenca tutti i COA a disposizione del nodo

Nota che periodicamente possono essere gli stessi agenti a inviare i ICMP per controllare e in caso avvisare i nodi mobili prima che questi inviano l'avviso dell'agente.

REGISTRAZIONE

PROTOCOLLI PER LA REGISTRAZIONE



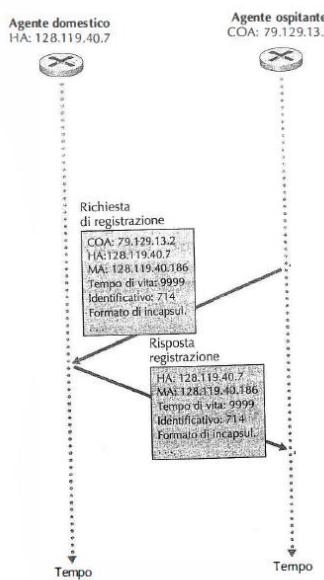
Come avviene la registrazione.

Siamo arrivati al punto in cui l'agente ha inviato un messaggio ICMP al nodo mobile con una lista di COA possibili.

Il nodo mobile risponde con un **messaggio IP trasportato in un datagramma UDP** e inviato alla porta 434 e riporta essenzialmente:

- Il COA scelto
- L'indirizzo HA dell'agente domestico da avvisare
- Il suo indirizzo permanente

- La durata della registrazione
- L'identificativo della registrazione usato fino alla fine.



Ricevuto il pacchetto, l'agente ospitante avvisa quello domestico a cui appartiene il nodo mobile inviandogli il COA che ha scelto, l'indirizzo IP dell'agente domestico e del nodo permanente a cui si riferisce, poi il formato di come debba essere incapsulato il pacchetto che gli viene inviato (così che lui sappia poi come decapsularlo correttamente), l'identificativo del messaggio e la durata della connessione.

L'agente domestico può così associare a quel nodo permanente il COA quando dovrà incapsulare i pacchetti. Risponde all'agente ospitante inoltrando lo stesso pacchetto senza il COA.

Quest'ultimo pacchetto viene poi inviato al nodo mobile da quello ospitante e si completa la registrazione.

PROBLEMI A LIVELLO SUPERIORE

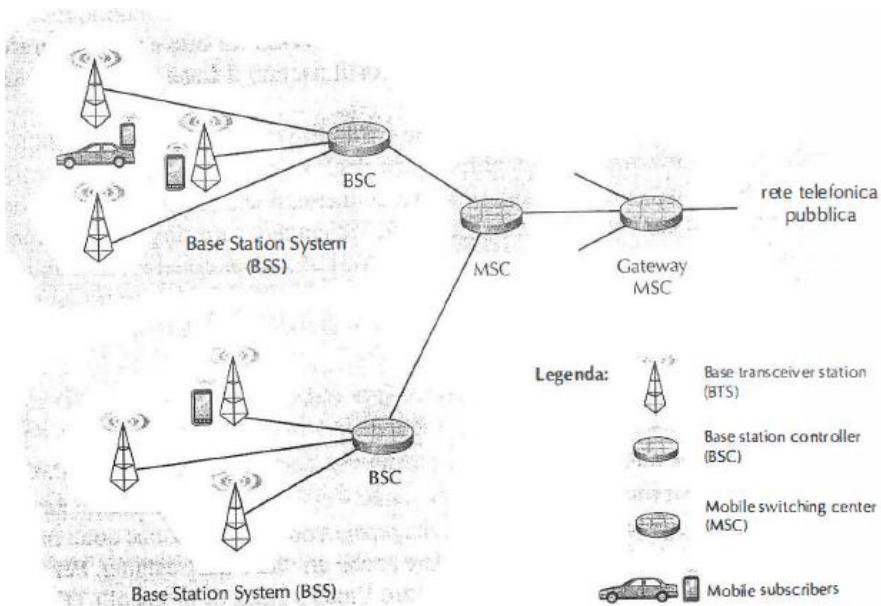
Abbiamo detto che le reti wireless differiscono da quelle cablate solo a livello di collegamento e di rete. Per questi due livelli bisogna smontare un po' la tecnologia col la quale abbiamo lavorato con le cablate. In realtà anche i protocolli di trasporto risentono del collegamento wireless.

Sappiamo che il TCP ritrasmette ogni segmento perso. Sappiamo che attraverso gli acknowledgement può sapere che è andato perso ma non perché. Sappiamo che imputerà sempre la colpa alla congestione e quindi ridurrà la propria finestra. Ma nei collegamenti wireless la maggior parte degli acknowledgement inviati dal ricevitore potrebbero riguardare non la perdita del pacchetto o la mai avvenuta ricezione, piuttosto la corruzione dei bit. Interpretare però sempre come un problema dovuto alla congestione non farà altro che diminuire la finestra di congestione anche quando il canale risulta libero.

Questo porta a 3 possibili approcci per risolvere il problema:

- 1) **Recupero locale**: si recuperano gli errori nei bit quando e dove avvengono. Questi protocolli includono ARQ 802.11
- 2) **Consapevolezza del trasmittente TCP del collegamento wireless**: dotare l'entità della capacità di capire se la connessione è wireless (in tal caso non considerare gli ack dovuti alla congestione ma alla corruzione dei bit), mentre se è cablata considerare gli ack come dovuti alla congestione.
- 3) **Suddivisione del canale**: ogni connessione end-to-end viene splitata in due connessioni a livello di trasporto: una dall'utente mobile all'access point wireless e una cablata da questo all'altro punto finale.

ARCHITETTURA DI UNA RETE CELLULARE



GSM è lo standard che definisce la tecnologia usata nella comunicazione della telefonia cellulare *internazionale*. A seconda delle **generazioni** della tecnologia si definisce:

- **1G** le tecnologie di prima generazione che hanno permesso solo comunicazioni audio;
- **2G** le tecnologie di seconda generazione

che permettevano la fonia ma anche il traffico dati come internet (col 2.5G in verità).

- **3G** che permette invece sia la voce che i dati con prestazioni sempre più corpose.

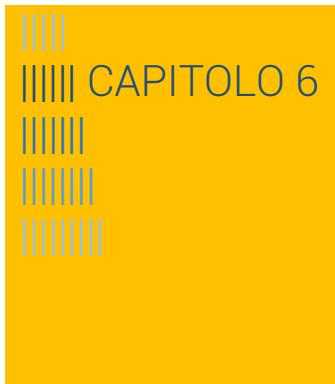
La tecnologia GSM che sta alla base vede l'area geografica come divisa in celle:



In ciascuna di queste celle sono presenti delle **BTS**, ossia delle stazioni che trasmettono il segnale. Tutte queste stazioni sono collegate a gruppi a un'unica **BSC** ossia stazione base di controllo che alloca canali radio BTS agli utenti mobili. Tutti i BSC sono poi collegati all'**MSC** che stabilisce se un utente ha

l'autorizzazione a connettersi o meno.

GSM utilizza un mix di FDMA e TDMA in cui ogni canale è ripartito in sottobande di frequenza e ogni frequenza è suddivisa in frame e slot. Ogni terminale sarà associato a un BTS e avrà il suo slot dedicato per inviare e ricevere dati.



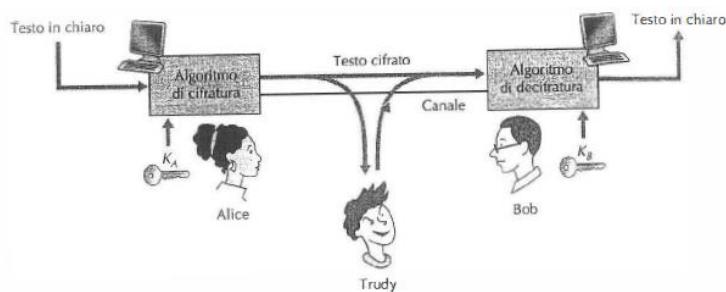
SICUREZZA NELLE RETI

Per comunicazione sicura si intende:

- *riservatezza*: solo il mittente e il destinatario devono comprendere i messaggi che si scambiano.
- *Integrità del messaggio*: il contenuto dei messaggi non deve subire alterazioni.
- *Autenticazione*: il mittente e il destinatario devono essere sicuri dell'identità dell'altro.
- *Sicurezza operativa*: devo operare in rete nella più grande sicurezza.

Per la riservatezza si procede a **crittografare** i messaggi.

CRITTOGRAFIA



Il messaggio originale che Alice vuole inviare a Bob viene chiamato **testo in chiaro**, mentre quello *cifrato* per evitare di essere compreso da Trudy viene detto **messaggio cifrato** e viene ottenuto tramite **algoritmi di cifratura**.

Gli algoritmi di cifratura sono standard e pubblicamente disponibili; quindi è opportuno utilizzare gli algoritmi insieme a delle **chiavi**. Alice avrà una chiave K_A (che sta per key di Alice) ed è la chiave utilizzata dall'algoritmo di cifratura; mentre Bob avrà una chiave K_B che

viene usata negli **algoritmi di decifratura** in cui si ottiene il testo in chiaro dal messaggio cifrato ottenuto.

Esistono due tipi di chiavi:

- Chiavi simmetriche $K_A = K_B$
- Chiavi pubbliche $K_A \neq K_B$

CRITTOGRAFIA A CHIAVE SIMMETRICA $K_A = K_B$

Vediamo come vengono cifrati e decifrati i messaggi usando chiavi simmetriche. Quindi Alice e Bob conoscono entrambi l'algoritmo e possiedono entrambi la **stessa** chiave per cifrare e decifrare.

CIFRARIO DI CESARE

Dato un messaggio m si procede a sostituire ciascuna lettera nel testo con un'altra dell'alfabeto che risulta sfasata da quella di k posizioni. Per esempio se $k=3$ allora sostituisco tutte le 'a' con le 'd'. In questo caso l'algoritmo è conosciuto, standard, pubblico e se Trudy dovesse sniffare pacchetti dalla rete e k fosse anche lui standard saprebbe come fare. Ecco il motivo di accompagnare ciascun algoritmo con una chiave. Infatti in questo algoritmo il valore di k è proprio la chiave $\rightarrow K_A = K_B = k$. Bob, di contro, conoscendo k effettuerà il processo inverso sostituendo ogni lettera con la lettera corrispondente a k posizioni prima nell'alfabeto.

Un evoluzione del cifrario di Cesare è il **cifrario monoalfabetico** in cui si sostituisce una lettera con un'altra qualsiasi. Le possibili combinazioni in un alfabeto di 26 lettere sono 26! possibili combinazioni. Se il testo cifrato fosse usato come password e fosse rintracciato da Trudy, questo impiegherebbe un tempo elevatissimo per trovare la password corretta. La *chiave* corrisponde alla tabella che associa ad ognuna delle 26 lettere una lettera diversa. Tuttavia basandosi su statistiche si potrebbe scoprire parte della tabella (per esempio le lettere che compaiono di più sono la 'e' e la 'a' quindi possono già ricavare parte della tabella e scoprire il resto per somiglianze).

CIFRARIO A BLOCCHI

Dato un messaggio in chiaro quello che si fa è suddividerlo in blocchi da k bit.

010110001111

010 110 001 111 k=3

Ciascun blocco viene poi dato in pasto all'algoritmo che userà una chiave che altro non è che una tabella di corrispondenza, per esempio potrebbe essere:

INGRESSO	USCITA
000	110
001	111
010	101
011	100
100	011
101	010
110	000
111	001

Pertanto i blocchi in uscita saranno

101 000 111 001

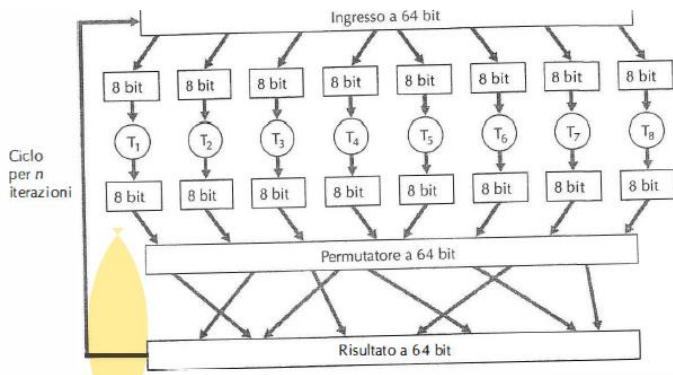
che compattiamo...

101000111001

Questo è il testo cifrato.

Trudy dovrebbe provare $2^k! = 2^3! = 8!$ combinazioni per trovare la chiave/tabella corretta. Notiamo che se $k=64$ allora Trudy dovrebbe impiegare $2^{64}!$ combinazioni, ossia maggiore è k maggiore è la sicurezza. Però queste chiavi sono tabelle che sia da Alice che da Bob devono essere mantenute in locale, il che significa memorizzare 2^{64} coppie ingresso-uscita il che è troppo anche se pensiamo che devono rigenerare la tabella con coppie diverse.

Un'alternativa è quindi la seguente:



Le n iterazioni sullo stesso blocco di 64 bit servono a far sì che possano variare tanti bit. Ora se supponiamo che la funzione permutazione operi sempre allo stesso modo, la chiave di Alice e Bob sarebbero le 8 tabelle.

Si può scegliere per esempio $k=64$ bit e quindi trattare blocchi da 64 bit. Invece di usare una tabella di 2^{64} bit si utilizzano 8 tabelle da 2^8 bit e l'ingresso si divide in 8 blocchi da 8 bit che vanno ciascuno in una delle 8 tabelle. Dopo di che si assembla il risultato, si *permuta*, e lo si ridà in ingresso per altre n iterazioni.

CIFRARIO A BLOCCHI CONCATENATI (CBC)

Il problema del cirfrario a blocchi è che Trudy potrebbe capire la cifratura quando vede blocchi cifrati uguali. Si ricorre pertanto al CBC che consiste non più nel generare il blocco cifrato $c(i)$ come $c(i) = K_A(m(i))$ ma per ogni blocco si sceglie una stringa di k bit chiamata $r(i)$ e quello che si farà sarà generare il rispettivo blocco cifrato come $c(i) = K_A(m(i) \oplus r(i))$ ossia utilizzando come input dell'algoritmo il risultato tra il blocco originale e la stringa di k bit.

NB: quello che sarà inviato non sarà più solo $c(i)$ ma anche $r(i)$. Ora se per ogni blocco bisogna trasmettere due informazioni questo vuol dire che si richiede più banda.

Ci vorrebbe trovare un modo tale per cui Bob, una volta ricevuto il primo $r(0)$ sa quali sono i successivi utilizzati da Alice senza che questa ogni volta li invii. Non si usa un r qualsiasi, piuttosto si utilizza lo stesso c trovato ad ogni passaggio (tranne all'inizio). Infatti la relazione da usare sarà:

- 1) Si elabora $c(0) = K_A(m(0) \oplus r(0))$ con $r(0)$ chiamato **vettore di inizializzazione VI**.
Si invia a Bob la coppia $c(0), r(0)$
- 2) Si elabora $c(1) = K_A(m(1) \oplus c(0))$
Si invia a Bob solo $c(1)$
- i) Si elabora $c(i) = K_A(m(i) \oplus c(i - 1))$
Si invia a Bob solo $c(i)$

Quindi Bob saprà che per il primo blocco si utilizza una certa stringa, ma poi per i successivi si utilizza $r(i)$ come $c(i - 1)$

CRITTOGRAFIA A CHIAVE PUBBLICA $K_A \neq K_B$

Il problema della crittografia a chiave simmetrica è che in qualche modo Alice e Bob devono accordarsi per quale chiave di codifica/decodifica usare. Questo implica che il canale debba essere sicuro, il che non è scontato.

Per questo si è preferito utilizzare la *crittografia a chiave pubblica*. Se Alice deve inviare un messaggio a Bob allora Bob deve possedere due chiavi: una pubblica K_B^+ e una privata K_B^- . Chiunque voglia comunicare con Bob deve procurarsi la chiave pubblica e con quella codificare il messaggio $c = K_B^+(m)$ da inviare a Bob. Questo utilizzerà la sua chiave privata per codificare il messaggio, ossia $K_B^-(c) = m$

Il problema è quindi trovare una coppia di chiavi tale che:

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

A far questo ci pensa l'algoritmo RSA.

RSA

RSA prevede che ciascuno possegga una coppia di chiavi pubblica e privata.

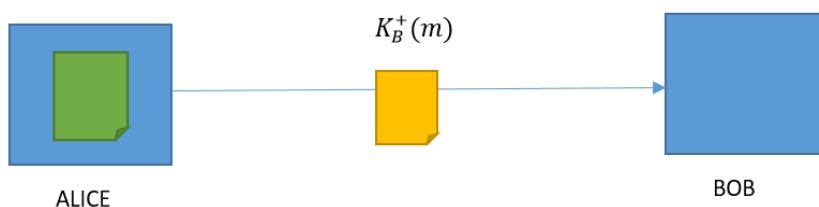
La **chiave pubblica** può solo criptare:

$$K^+(m)$$

La **chiave privata** può solo decriptare ciò che è stato criptato con la chiave pubblica:

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

Quando Alice vuole inviare un messaggio a Bob, usa la sua (di Bob) chiave pubblica per cifrare i messaggi.



Cosa impedisce a Trudy di infilarsi e decriptare il messaggio di Alice? In fondo la chiave è pubblica quindi conosco la chiave con la quale ha criptato il messaggio no? Ricordo per quanto detto prima che la **chiave pubblica serve solo a criptare e non può decriptare**, quindi non è possibile la seguente relazione $K_B^+(K_B^+(m)) = m$, ossia non posso utilizzare la chiave pubblica per decodificare ciò che ho codificato con la stessa chiave pubblica. Pertanto il canale può tranquillamente essere non così tanto sicuro, riuscirei comunque a non far decriptare il messaggio a Trudy.



L'algoritmo RSA quindi serve a generare la coppia chiave pubblica chiave privata che permetta questo scenario.

Sfrutta importanti proprietà matematiche come assiomi riguardo operazioni come somma, sottrazione e moltiplicazione in modulo n.

Quello che fa è questo:

- 1) Sceglie due numeri primi p, q
- 2) Calcola $n = pq, z = (p-1)(q-1)$
- 3) Sceglie un numero e tale che $e < n, e \neq 1, e$ primo con z (ossia non hanno divisori in comune)
- 4) Sceglie un numero d tale che $ed - 1$ sia divisibile da z (ossia il resto è nullo)

La chiave pubblica di Bob è allora:

$$K_B^+ = (n, e)$$

Mentre quella privata è:

$$K_B^- = (n, d)$$

Con queste chiavi lo scenario di prima è assicurato. La chiave pubblica per criptare il messaggio è usata in questo modo:

Dato un messaggio da inviare:



Il messaggio cripato si trova come:

$$c = m^e \text{ mod } n \longrightarrow \text{yellow box}$$

Per decriptare si usa:

$$\text{yellow box} \xrightarrow{m = c^d \text{ mod } n} \text{green box}$$

Integrità dei messaggi

Abbiamo visto che l'RSA permette di far mandare ad Alice un messaggio a Bob senza che Trudy possa decifrarne il contenuto perché pur conoscendo la chiave pubblica, poiché questa può solo cifrare e non decifrare, non può vederne il contenuto. Questo anche su un canale poco sicuro. Quindi abbiamo visto fin'ora come si garantisce la **riservatezza**.

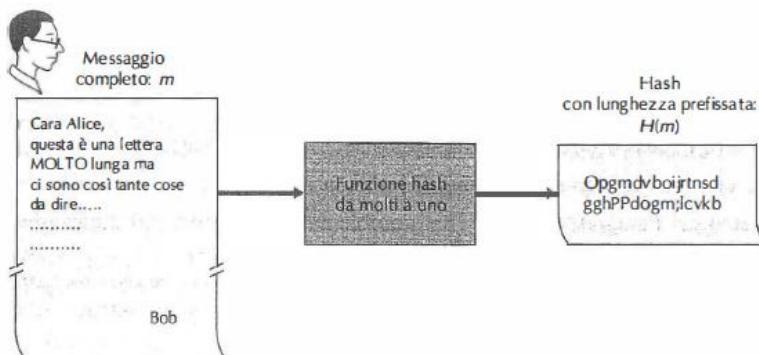
Vabbene che Trudy non può vederne il contenuto ma può alterarlo e può anche "fingersi" Alice inviando a Bob dei messaggi tramite la chiave pubblica di Bob usata precedentemente da Alice.

Dobbiamo quindi trovare un modo per garantire l'**integrità dei messaggi**, ossia:

- 1) Proteggere i messaggi.
- 2) Autenticare il mittente.

Per proteggere i messaggi da alterazioni useremo le *funzioni hash* mentre per autenticare il mittente useremo il *MAC* (non c'entra nulla col MAC address).

FUNZIONI HASH



Le funzioni hash H prendono in ingresso un messaggio m di qualsiasi lunghezza e restituiscono una stringa, di lunghezza fissa, detta **hash**. I checksum o i CRC ne sono un esempio. Questi però non riescono ad essere funzioni hash **senza collisioni**, nel senso che dati due messaggi diversi è possibile avere hash uguali, il che è grave. Supponiamo che per verificare l'integrità di un messaggio, così come fatto col checksum, si invii la coppia (m,h) così che il ricevente saprà che m è corretto se e solo se $H(m)=h$. Supponiamo che Alice voglia mandare a Bob un messaggio con scritto 100 euro di pagamento. Quello che invia a Bob è il messaggio m e l'hash ottenuto lo chiamo h . Supponiamo che Trudy si infili dentro e sostituisca il messaggio con un altro il cui hash è sempre h ma il messaggio originale è 120 euro. Quando arriverà a Bob lui controllerà l'integrità e vedrà che è corretto quindi penserà che dovrà pagare 120 euro.

Per questo motivo si utilizzeranno hash tali che non esistano due messaggi diversi che hanno lo stesso hash, e questi sono **MD5** e **SHA-1**.

Cosa abbiamo risolto?

In realtà abbiamo solo risolto il fatto che se si manda un messaggio m il ricevente riesce a capire solo se è integro. Infatti se Alice vorrà mandare m , ne calcolerà l'hash $h=H(m)$ e invierà la coppia (m,h) . Bob calcolerà $H(m)$ e verificherà che sia h . Ma nulla impedisce a Trudy di creare un suo messaggio m' (i 120 euro visti prima), calcolarne l'hash h' e inviare la coppia (m',h') . Una volta ricevuta da Bob, questo non potrà che rendersi conto che è integro anche un tale messaggio. Bisogna quindi autenticarne la paternità.

MAC

Ora che sappiamo che il messaggio arrivato è integro o meno, devo trovare un modo per far sapere al ricevente che sta comunicando con l'utente che lui pensa sia. Bisogna autenticare il messaggio perché come visto Trudy può comunque mandare la coppia (m,h) e dire di essere Alice.

Quello che si fa è usare una **chiave di autenticazione s** (stringa di bit) che sia Alice che Bob conoscono. Alice crea il messaggio m e con SHA-1 quello che codifica è $h=H(m+s)$.

$H(m+s)$ viene chiamato **codice di autenticazione del messaggio (MAC)**. Alice invierà quindi la coppia (m,h) .

Bob riceve il messaggio e conoscendo s quello che farà sarà sommare al messaggio m ricevuto la chiave s , calcolarne l'hash e controllare se risulta uguale all' h ricevuto.

Il MAC è quindi un meccanismo di crittografia **simmetrica** perché conoscono entrambi l'algoritmo (SHA-1) e posseggono entrambi la stessa chiave (MAC). Spesso per chiave MAC si intende s .

FIRMA DIGITALE

Abbiamo visto che il MAC è un meccanismo simmetrico che quindi obbliga le due parti a scambiarsi in una fase iniziale una chiave segreta che abbiamo chiamato *chiave di autenticazione s* . Per questo si usa molto spesso la **firma digitale** in quanto ha lo stesso scopo del MAC ma è un meccanismo di crittografia asimmetrico. Infatti Alice e Bob non devono scambiarsi alcuna chiave. Durante la trattazione che affronteremo sembrerà quasi di parlare dell'RSA, ma per quanto similissimi, operano in modo opposto.

Con l'RSA si garantiva che un messaggio inviato fosse riservato, non scopribile da Trudy.

Con la firma digitale si garantisce invece la paternità di un messaggio.

Supponiamo che Alice voglia mandare un messaggio a Bob. Alice possiede come visto anche nell'RSA (stiamo sempre parlando di crittografia asimmetrica) due chiavi, una pubblica e una privata che hanno lo stesso identico significato di prima, ossia:

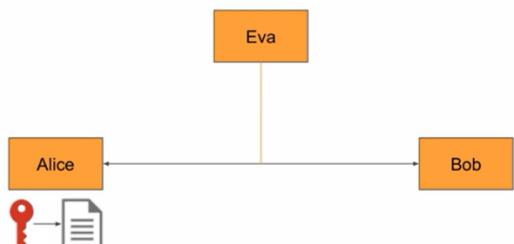
Chiave pubblica cripta.

Chiave pubblica decripta ciò che la chiave privata ha criptato.

Chiave privata cripta.
chiave privata decripta ciò che la chiave pubblica ha criptato.

Nell'RSA la riservatezza era garantita dal fatto che se Trudy vedeva passarsi il file criptato, avendo solo la chiave pubblica poteva solo criptarlo (e non gli serve) e basta, perché non può decriptare con una chiave pubblica quello che la chiave pubblica ha criptato.

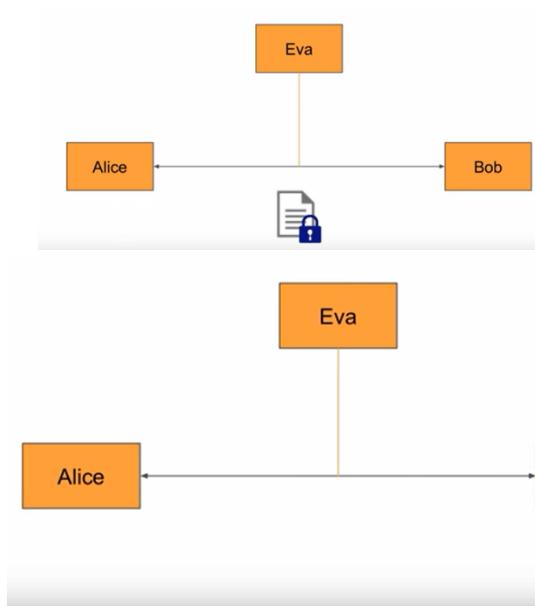
Qui nella firma digitale si fa l'inverso:



Alice vuole mandare un messaggio a Bob. Manda la coppia (m, h) dove

$$h = K_A^-(m)$$

Stavolta per firmare ognuno utilizza la sua chiave privata.



Eva (Trudy femminile ;) si vede passare davanti il file. Di Alice può avere solo la sua chiave pubblica. Dalle 4 proprietà delle chiavi dette sopra può solo *decriptare ciò che la chiave privata di Alice ha criptato*. E in effetti è così perché tanto a noi non interessa garantire la riservatezza ma la paternità.

Infatti semmai Bob volesse dimostrare che quel documento è stato firmato da Alice basterà applicare:

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

Dove m è il documento di cui vuole dimostrare la paternità di Alice. Inoltre si può anche verificare in contemporanea l'integrità di m .

CERTIFICAZIONE DELLA FIRMA DIGITALE

Ma quando nell'esempio di prima Bob per dimostrare la paternità di Alice del file inviato utilizza la chiave pubblica di Alice, chi mi accerta che quella chiave pubblica sia corretta e non appartenga invece a qualcun altro? La relazione tra una data chiave pubblica e una determinata entità è stabilita da una **autorità di certificazione (CA)**. Quando qualcuno ottiene una chiave pubblica e privata ottiene anche un certificato che contiene la chiave pubblica e informazioni sul proprietario come indirizzo Ip, nome etch.. Tale certificato è crittografato con la chiave privata della CA. Per dimostrare che è lui, il proprietario deve mandare oltre alla propria chiave pubblica anche il certificato. Il ricevente userà la chiave pubblica della CA per ricavarne il contenuto, ossia la chiave pubblica, da confrontare con quella inviata dal proprietario.

SICUREZZA EMAIL

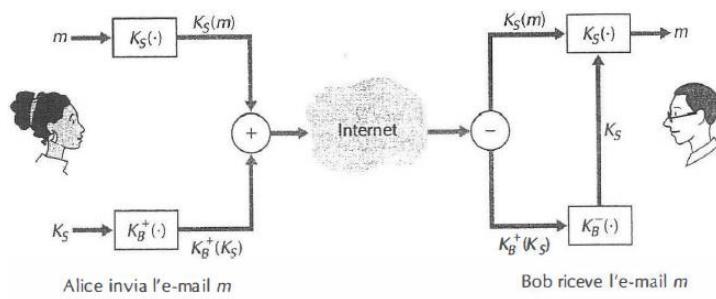
SICUREZZA A LIVELLO APPLICATIVO: EMAIL

Abbiamo studiato prima tanti strumenti che ci garantiscono alcuni la riservatezza, altri l'autenticità, altri l'integrità. Proviamo a vedere inizialmente come si applicano questi concetti per rendere sicuro il servizio di posta elettronica e dopo come si assembla tutto per garantire non uno ma tutte e 3 le garanzie di sicurezza.

GARANZIA DI RISERVATEZZA

Quando Alice invia un messaggio a Bob vorrebbe che ci fosse riservatezza, che Trudy non riuscisse a scoprire il messaggio originale. Per fare questo il metodo migliore è usare l'*RSA*. Bob rilascia la sua chiave pubblica per esempio su un server o sul suo sito web. Chi vuole comunicare con lui utilizza la sua chiave pubblica per cifrare e Bob la sua chiave privata per decifrare. Il messaggio in transito non sarà possibile leggerlo perché Trudy non può con la chiave pubblica decriptare ciò che è stato criptato con la chiave pubblica.

Quello che possiamo fare è inviare il messaggio m però criptato con la chiave pubblica di Bob come visto nell'*RSA* e niente più. Se però i messaggi però sono troppo lunghi la chiave pubblica si rivela inefficiente.



Pertanto Alice utilizzerà una **chiave di sessione simmetrica K_s** da lei scelta con la quale cripterà il messaggio da mandare. Poi con la chiave pubblica di Bob cripterà invece la chiave di sessione. A Bob manderà la coppia $(K_s(m), K_B^+(K_s))$ e lui tramite la sua chiave privata per prima cosa decripterà $K_B^+(K_s)$ per ottenere K_s così che utilizzerà quest'ultimo per decriptare $K_s(m)$ in quanto simmetrica.

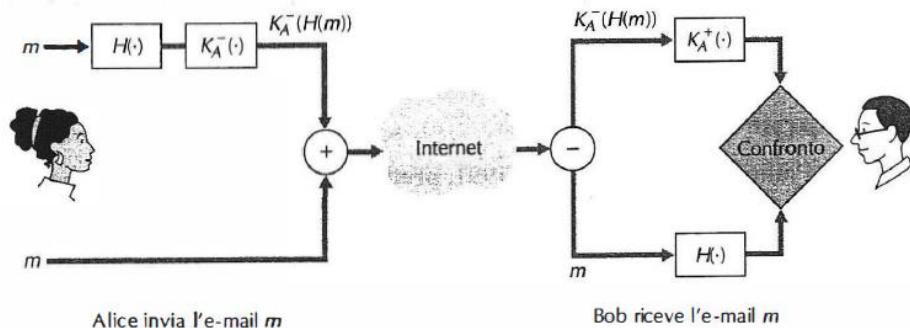
chiave privata per prima cosa decripterà $K_B^+(K_s)$ per ottenere K_s così che utilizzerà quest'ultimo per decriptare $K_s(m)$ in quanto simmetrica.

AUTENTICAZIONE MITTENTE E INTEGRITA' DEI MESSAGGI

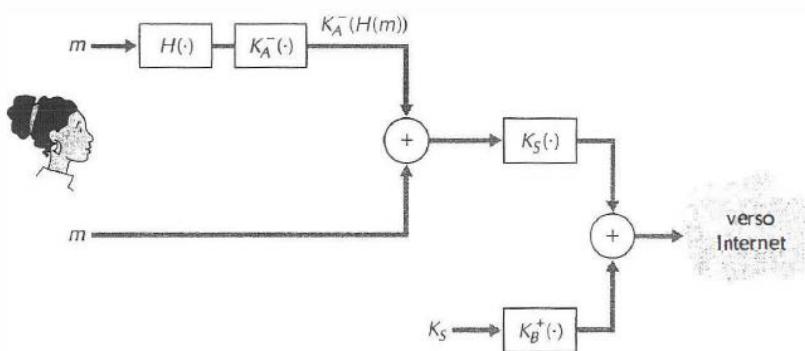
Le migliori tra quelle viste sono la firma digitale per autenticarsi e MD5 per l'hash (ma andrebbe bene anche SHA-1). Alice applica una funzione H hash al messaggio m e di ciò che ottiene lo firma (quindi lo cifra) con la sua chiave privata. Manda la coppia:

$(m, \text{messaggio cifrato con hash e con chiave privata (ossia firma)})$

Bob di contro utilizzerà la chiave pubblica di Alice alla firma per ricavare il messaggio cifrato con hash, e lo confronta con l'hash eseguito su m.



ASSEMBLIAMO TUTTO PER IL NOSTRO SISTEMA DI SICUREZZA EMAIL FINALE



Abbiamo unito gli schemi. Il primo per l'integrità e l'autenticità. Dopodichè per far sì che messaggio e firma siano riservati si applica il secondo schema con chiave di sessione.

Infatti:

- 1) Per l'integrità prepara la coppia (m, h) con $h=H(m)$.
- 2) Deve però dire che la coppia (m, h) è inviata da lui quindi ne firma l'hash h ottenuto creando la coppia (m, h') .

A questo punto se non si facesse altro il ricevitore applicherebbe la chiave pubblica ad h' per ricavarne h e conoscendo la funzione hash H la applicherebbe ad h per verificare che dia m .

- 3) Però bisogna garantire che la coppia (m, h') viaggi con riservatezza. Per questo motivo cripterà tutto con una chiave da lei scelta K_S generando m' e mandando $(m', K_B^+(K_S))$.

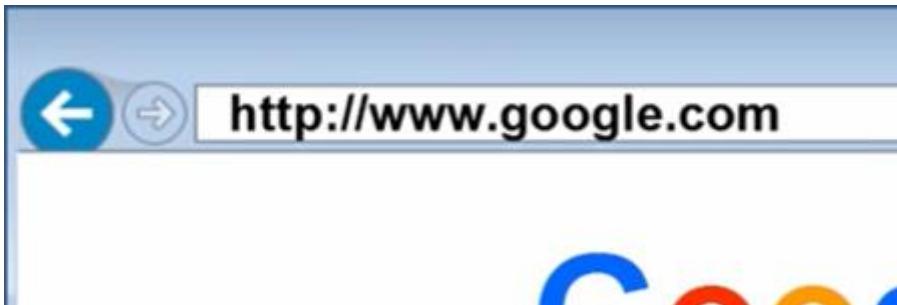
SICUREZZA A LIVELLO DI TRASPORTO: TCP con SSL

Il protocollo SSL è supportato dalla maggior parte dei browser e per quanto gira a livello applicativo nella pila protocolare, è a tutti gli effetti un protocollo di sicurezza per il trasporto. Qui lo vediamo usato nel TCP.

A cosa serve?



Scriviamo www.google.com



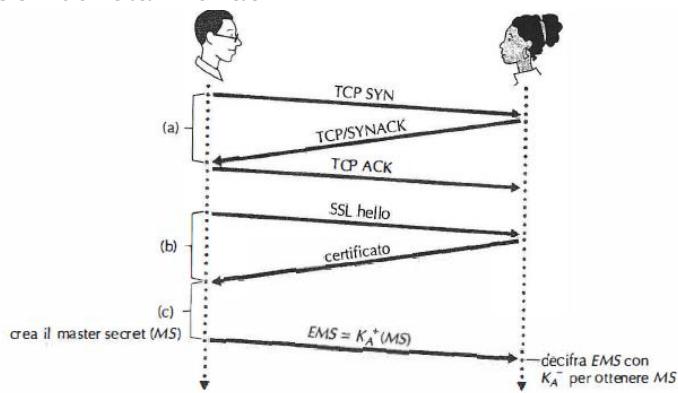
Quello che succede è che viene messo *http* all'inizio. Questo vuol dire che stiamo utilizzando HTTP per ricevere la pagina di google. HTTP però manderà messaggi sempre in chiaro.



Un hacker potrebbe mettersi in mezzo per sniffare dati dalla rete e vederne il contenuto. Ovviamente fino a quando i dati non sono sensibili, HTTP va più che bene, ma quando si utilizza per mandare password, numero di carta etc.. è meglio avere un canale sicuro.

Un esempio di uso di SSL è quando stiamo comprando un prodotto su un e-commerce fornendo dati sensibili. Quando l'url comincia con **https** (nota la s finale) allora significa che a livello di trasporto si sta usando SSL per garantire la sicurezza.

SSL consta di 3 fasi:



FASE A

Si stabilisce una connessione TCP con il server con cui si vuole scambiare messaggi.

FASE B

Si vuole avere la conferma che quel server è chi dice di essere.

FASE C

Si scambiano i messaggi.

NB: Nella FASE B si ricorre al meccanismo di crittografia asimmetrico con chiavi private e pubbliche per il seguente motivo: come visto per il meccanismo simmetrico, nessuno da

garanzia che il canale sia affidabile quindi non potendosi scambiare chiavi simmetriche devono necessariamente ricorrere alla crittografia asimmetrica in questa fase con l'invio della chiave pubblica con allegato il certificato.

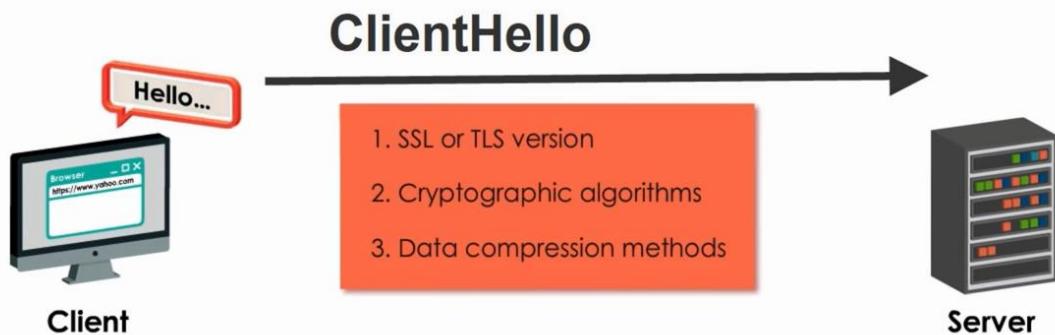
Nella FASE C si ricorre invece alla crittografia simmetrica. Si usa per l'ultima volta quella asimmetrica con la quale si passa una MS (genererà le chiavi simmetriche) crittografata dalla chiave pubblica del server.

Supponiamo di avere un client che vuole stabilire una connessione TCP sicura:



La FASE A è identica a una qualsiasi connessione TCP già vista.

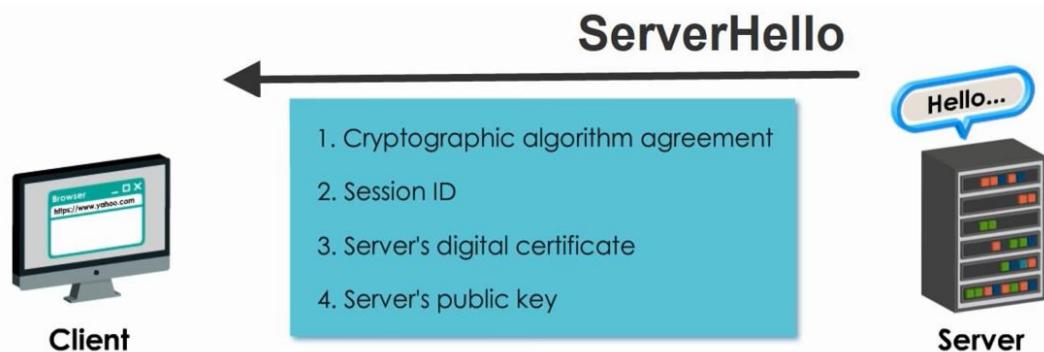
FASE B



In questa fase il client manda un messaggio di Hello al server in cui esplica:

- La versione di protocollo (sempre SSL nel nostro caso)
- La lista degli algoritmi crittografici che supporta il client
- (non ci interessa)

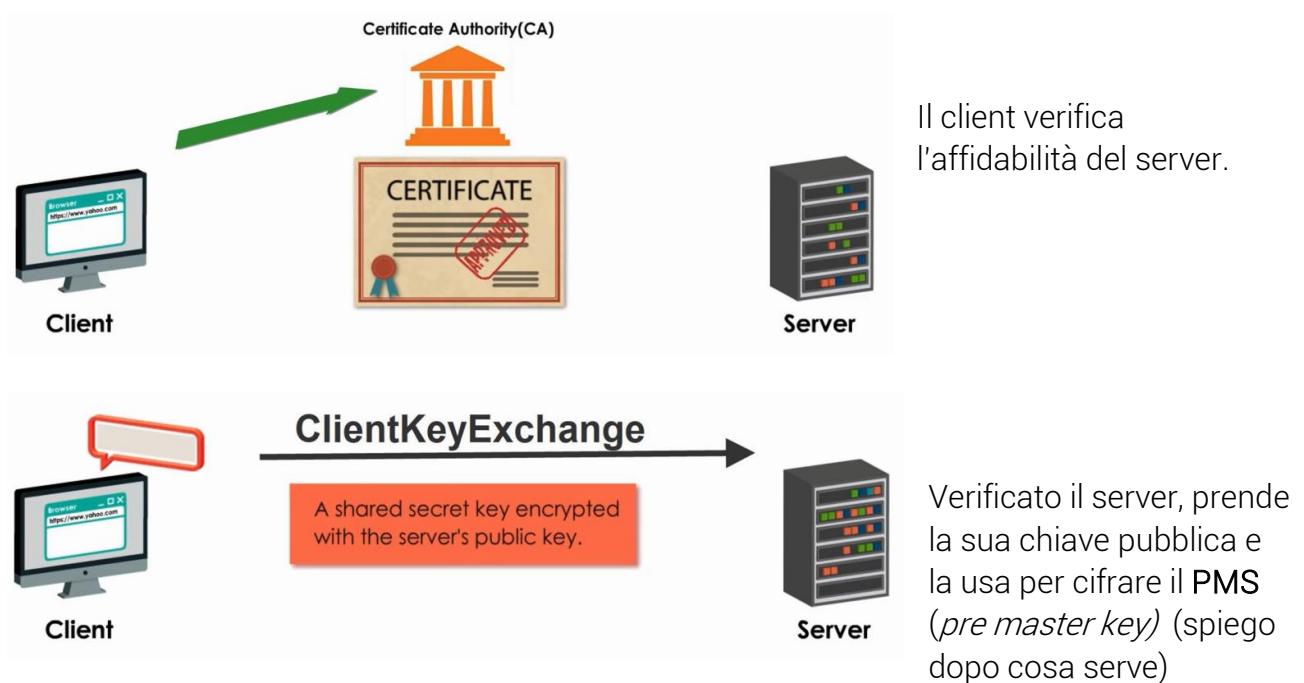
FASE 3



Il server risponde con un ServerHello in cui esprime:

- Quale algoritmo a chiave simmetrica, pubblica e MAC ha deciso dalla lista del client.
- Un certificato digitale per la chiave pubblica.
- La chiave pubblica del server.

FASE C



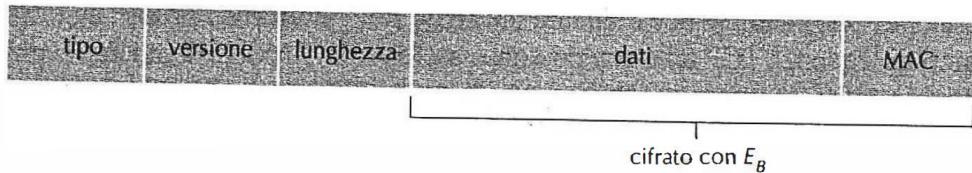
Sia il server che il client adesso posseggono il PMS. Ciascuno calcola l'**MS** (ossia il master key) nella propria stazione a partire dal PMS. Ottenuto l'**MS** lo si divide in quattro per formare 4 chiavi di cifratura *simmetriche*.

- E_B chiave di cifratura di sessione per i dati inviati da Bob ad Alice
- M_B chiave di MAC di sessione per i dati inviati da Bob ad Alice
- E_A chiave di cifratura di sessione per i dati inviati da Alice Bob
- M_A chiave di MAC di sessione per i dati inviati da Alice a Bob

Queste 4 chiavi (uguali sia per BoB che per Alice) vengono usate per scambiarsi dati in sicurezza sulla connessione TCP. In particolare Bob utilizzerà M_B per autenticarsi e E_B per crittografare il messaggio.

Nb: perché 4 chiavi? In effetti non ha senso se stiamo parlando di meccanismo simmetrico, ne basterebbero 2. L'idea è che per aumentare la sicurezza se ne usano 4. Quando Bob invia i dati allora Alice sa che deve decifrarli utilizzando E_B (invece che E_E) e che il MAC in quella sessione, essendo BOB a inviare i dati, è M_B .

Ma se a livello di applicazione (dove gira SSL) i messaggi sono inviati per intero al livello di trasporto, e TCP li vede come flussi di byte da spezzettare, non posso usare il MAC sul messaggio se poi i vari frammenti che lo compongono viaggiano senza MAC (tranne il primo). Allora l'SSL organizza i flussi di dati che arrivano al TCP in **record** di questo formato:



Quindi se ho un messaggio da inviare, SSL lo spezzetta in frammenti. Ciascun frammento lo incapsula in un *record* nella sua parte dati. Il campo tipo indica se il messaggio è usato per aprire o chiudere la connessione o trasferire dati. La lunghezza indica la parte dei dati e del MAC. Nota che dati e MAC non sono uniti ma rappresentano la coppia (m, h) discussa nel paragrafo del MAC, quindi i dati sono i messaggi mentre h è $H(m+s)$ con $s = M_B$. Per riservatezza entrambi vengono cifrati con E_B . Quando l'SSL di Alice riceverà il segmento TCP, lo decapsulerà per ricavarne il record e decifrerà la coppia cifrata per ricavare m e $MAC(h)$, ossia applicherà la chiave simmetrica E_B ai dati(m) cifrati e al $MAC(h)$ cifrato. Decifrato il contenuto dovrà constatare se è autentico e quindi sommerà M_B (ossia s) ai dati appena decifrati, li cifrerà con E_B e vedrà se otterrà il $MAC(h)$.

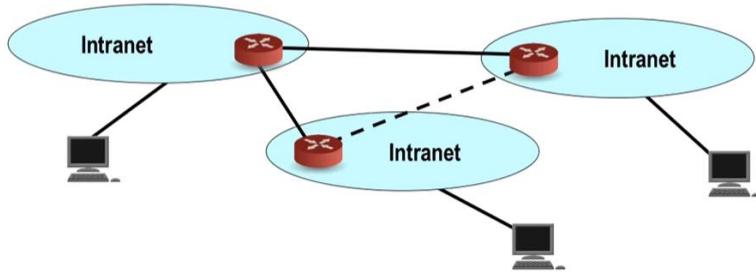
Trudy in questo modo non può decifrare nulla. Però supponiamo che ad Alice stiano arrivando due record inviati da BOB. Vabbene che Trudy non può decifrarne il contenuto, però può comunque "ignorantemente" replicarlo. In particolare può invertire l'ordine dei due record e siccome Alice quando ne verificherà l'autenticità non risconterà problemi (perché il MAC è applicato ad ogni singolo record) ci potrebbero essere dei problemi.

La soluzione è tenersi memorizzati dei **numeri di sequenza** oltre a quelli TCP che Bob e Alice solo conoscono e questi verranno inclusi nel MAC, ossia quando Alice invia la coppia (m, h) quell' h sarà così fatto: $h = H(m+s+NS)$. Quando Bob riceverà il record saprà che oltre ad s dovrà sommare anche NS .

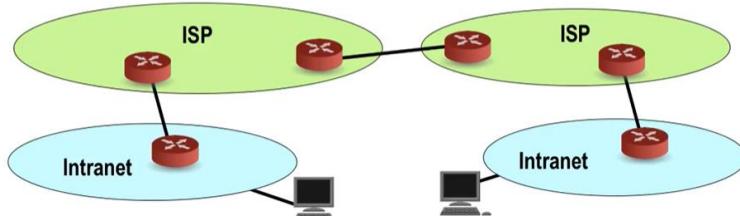
SICUREZZA A LIVELLO DI RETE: IPsec

Prima di parlare dell'IPsec introduciamo il problema che lui risolve.

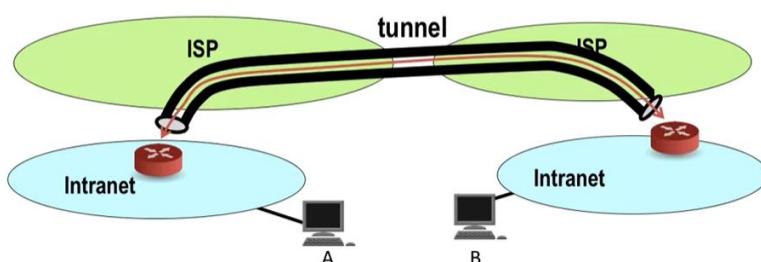
Supponiamo di avere diverse **intranet**, ossia porzioni di rete che costituiscono l'Internet caratterizzati però dall'uso di specifici protocolli di sicurezza. Supponiamo che una sede che possiede diverse intranet (che ripeto, rispettano tutti un determinato protocollo di sicurezza) voglia collegarle tra loro.



Una soluzione potrebbe essere quella di installare una propria rete di collegamento (che comprende router, collegamenti, infrastruttura DNS ecc...) o affittare da terzi parti queste reti già esistenti. In ogni caso i costi sono molto elevati.



Quello che invece si fa è usare una **virtual private network (VPN)** ossia una rete privata virtuale che si appoggia sulla già esistente Internet pubblica risparmiandoci molti costi di installazione.



Per creare questo *tunnel*/di collegamento si incapsulano i pacchetti IP dentro pacchetti IP.

Quindi ogni host appartenente alla propria intranet genererà normalmente un pacchetto IP che

conosciamo:

dest	sorg	payload
------	------	---------

E lo invia al router rosso. Questo impacchetterà in un altro pacchetto IP, ossia quello di prima diventerà il payload del nuovo:

dest	sorg	dest	sorg	payload
------	------	------	------	---------

Di solito gli host dell'intranet hanno indirizzi privati che sarebbero quelli dest e sorg in verde, mentre il nuovo header IP aggiunto dal router rosso inserisce indirizzi IP pubblici perché il pacchetto dovrà viaggiare all'interno dell'Internet pubblico. In particolare il sorg in blu è l'indirizzo IP dell'interfaccia del router mittente, mentre il dest è l'IP dell'interfaccia del router ricevente. Una volta spacciato, vedrà l'indirizzo IP verde privato dest al quale inviarlo tra i propri host.

N.B: il router ricevitore deve sapere, prima di mandare il pacchetto al livello di rete, che il pacchetto IP da trattare è quello incapsulato, ossia il payload.

Ora supponiamo che due intranet debbano scambiarsi dati sensibili. Un esempio sarebbero un quartier generale e una filiale. I dati che circolano sono sensibili. Ora se la rete di collegamento tra le due intranet fosse fatta nel primo modo, ossia tramite installazioni proprie non ci sarebbe alcun problema di sicurezza perché a quella rete ci girano solo loro. Se invece ci interfacciamo a una rete pubblica, ossia utilizziamo come percorso l'Internet pubblico per giungere alla filiale, chiunque potrebbe sniffare i dati. Allora quello che si fa è **cifrare il datagramma in verde**.

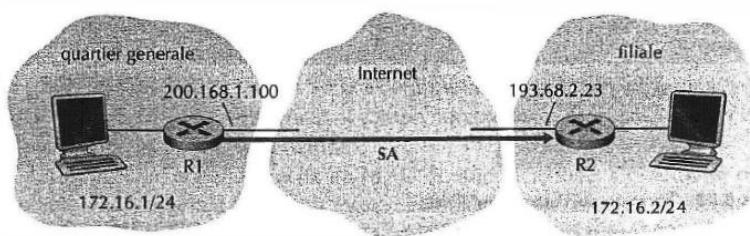
IPsec CON ESP

Esistono due versioni del servizio IPsec e sono: AH e ESP. Siccome AH fornisce solo autenticazione e integrità, mentre ESP fornisce anche riservatezza (cosa importante nelle VPN) allora affronteremo ESP.

ASSOCIAZIONI SICURE SA

Prima di tutto bisogno creare questo canale di comunicazione tra sorgente e destinazione da parte dei due router rossi.

Questi vengono chiamati **associazioni di sicurezza (SA)**. Le AS sono però unidirezionali e quindi per una comunicazione bidirezionale ci vuole un'altra SA. Chi gestisce la SA è chi invia i dati.



In questo esempio R1 gestisce la SA di comunicazione con R2. Il router R1 memorizzerà **lo stato della SA**, ossia:

- L'**SPI**, che è un identificativo a 32 bit della SA
- L'interfaccia di origine

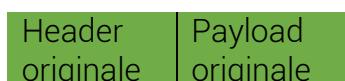
(200.168.1.100) e destinazione (193.68.2.23) della SA.

- Il tipo di codifica
- La chiave di codifica
- Il tipo di controllo di integrità
- La chiave di autenticazione

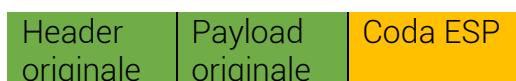
E' quindi chiaro che un gateway può memorizzare più SA all'interno di un **database di associazione sicura (SAD)**.

DATAGRAMMI Ipsec

Al gateway arriva il seguente datagramma IPv4



Appende infondo un campo "coda ESP"



Cifra il risultato usando l'algoritmo e la chiave specificati nel SA



Appende all'inizio un campo "intestazione ESP"



Questo pacchetto è chiamato **enchilada**.

Adesso crea un MAC così ---> $MAC = h = H(\text{enchilada} + s)$ con s la classica chiave di autenticazione che conosciamo particolare per quella SA. Appende il MAC.



Alla fine crea la classica intestazione IPv4 e la appende:



Come detto prima l'header originale ospiterà come indirizzo IP di sorgente quello dell'host e di destinazione quello dell'host finale, mentre la nuova intestazione vedrà come indirizzo IP di sorgente quello dell'interfaccia del primo router e come destinazione quella del secondo router.

Il campo **coda ESP** contiene:

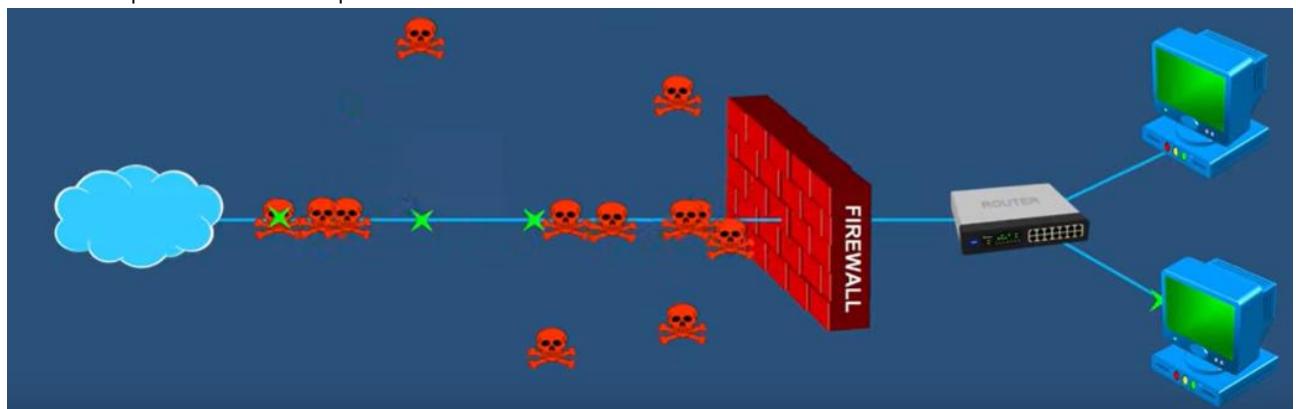
- *Padding*: utile agli algoritmi di cifratura a blocchi per avere header originale+payload originale multipli della lunghezza del blocco.
- *Lunghezza*: indica al ricevente quanti byte di padding sono stati inseriti e che quindi devono essere rimossi.
- *Intestazione successiva*: indica il tipo di dato (UDP o altro) che contiene la sezione verde, che per noi è sempre un datagramma IP.

Mentre il campo **intestazione ESP** contiene:

- *SPI*: ossia l'identificativo del SA a cui appartiene e che quindi può usare come indice nel SAD per recuperare le info che gli servono alla decriptazione.
- *Numero sequenza*: utile per evitare ripetizioni malevoli.

FIREWALL

Il Firewall è una combinazione di hardware e software che si pone come muro a dividere una rete privata da una pubblica.



Esistono 3 tipi di categorie di firewall:

- Filtri di pacchetto
- Filtro con memoria di stato
- Gateway a livello applicativo

Vediamoli a uno a uno.

Filtr di pacchetto



Un firewall di solito è un componente a se come quello in figura oppure può essere inglobato in un router di quelli classici che conosciamo.

Un firewall a filtro di pacchetto *filtra* i pacchetti secondo diverse **politiche di filtraggio** che in genere sono:

- Per indirizzo IP di sorgente o destinazione
- Tipo di protocollo nel campo del datagramma IP entrante
- Numero di porte sorgente o destinazione
- Bit di flag
- Tipi di messaggio ICMP

Inoltre le regole sopra esposte possono essere diverse per interfaccia di router.

Le regole sono implementate in delle tabelle nei router chiamate **liste di controllo degli accessi** (ovviamente per una tipica interfaccia di router):

Azione	Indirizzo sorgente	Indirizzo destinazione	Protocollo	Porta sorgente	Porta destinazione	Bit di flag
consenti	222.22/16	al di fuori di 222.22/16	TCP	>1023	80	qualsiasi
consenti	al di fuori di 222.22/16	222.22/16	TCP	80	>1023	ACK
consenti	222.22/16	aldi fuori di 222.22/16	UDP	>1023	53	-
consenti	al di fuori di 222.22/16	222.22/16	UDP	53	>1023	-
blocca	qualsiasi	qualsiasi	tutti	qualsiasi	qualsiasi	tutti

Supponiamo che la propria sottorete sia 222.22/16

Per esempio la prima riga dice (il router prende il datagramma IP e legge i campi sorgente, destinazione, e altro...):

"consenti il passaggio in uscita a tutti i datagrammi con indirizzo sorgente della propria sottorete e destinazione fuori la propria sottorete, che viaggiano con protocollo TCP dalla porta 1023 diretti verso la 80 qualsiasi sia il tipo di flag"

Una politica del genere stabilisce equivalentemente "alla sottorete 222.22/16 consenti solo connessioni http (xe la porta è 80) verso il mondo".

La seconda è particolare, e dice:

"consenti ai datagrammi di chiunque sia fuori dalla sottorete propria di arrivare alla sottorete propria solo se veicolati con TCP e diretti alla porta 80 e con ACK=1"

Una politica del genere dice "fai entrare i datagrammi dal resto del mondo solo se sono rivolti alla porta 80, quindi come prima solo traffico web". Però qui si pretende che l'ACK sia 1. Cosa vuol dire? Per istaurare una connessione TCP bisogna effettuare lo handshake a 3 vie. Se A vuole istaurare un canale unidirezionale allora manderà il primo messaggio che avrà necessariamente ACK=0 perché ha cominciato lui. Tutto il resto avrà ACK=1 perché valida i pacchetti ricevuti prima. Quindi in genere "l'ACK=0 è tipico di colui che vuole istaurare un canale di comunicazione unidirezionale". Pertanto se qui l'ACK si pretende essere solo 1 allora significa che dal mondo non si vuole altro che ricevere al massimo pacchetti, mai che il mondo instauri un suo canale personale con noi per inviare lui i dati.

Filtri con memoria di stato

Risolvono un problema ben preciso che succedeva nell'ultimo caso visto prima. Filtrare per ACK=1 non è sufficientemente una buona soluzione di difesa. Basterebbe manipolare il datagramma inserendo 1 al posto di 0 e riusciremmo a superare il firewall.

I filtri con memoria di stato prevedono due tabelle. La prima è la stessa di prima, solo che prevedono una nuova colonna con il campo "verifica connessione", mentre la seconda tabella è detta **tabella di connessione** in cui vengono annotate tutte le connessioni TCP in corso. Vediamo come fanno a risolvere quel problema:

Azione	Indirizzo sorgente	Indirizzo destinazione	Protocollo	Porta sorgente	Porta destinazione	Bit di flag	Verifica della connessione
consenti	222.22/16	al di fuori di 222.22/16	TCP	>1023	80	qualsiasi	
consenti	al di fuori di 222.22/16	222.22/16	TCP	80	>1023	ACK	X
consenti	222.22/16	al di fuori di 222.22/16	UDP	>1023	53	-	
consenti	al di fuori di 222.22/16	222.22/16	UDP	53	>1023	-	X
blocca	tutto	tutto	tutto	tutto	tutto	tutto	

La lista di controllo è identica a quella di prima. Quindi il datagramma con ACK=1 malevolo (quindi che dovrebbe riportare ACK=0) riesce a passare il controllo secondo la politica della seconda riga però adesso la riga dice anche di prevedere per quelli che passano una verifica della connessione.

Indirizzo sorgente	Indirizzo destinazione	Porta sorgente	Porta destinazione
222.22.1.7	37.96.87.123	12699	80
222.22.93.2	199.1.205.23	37654	80
222.22.65.143	203.77.240.43	48712	80

Se l'indirizzo di sorgente del datagramma coincide con uno di destinazione nella tabella delle connessioni allora significa che quel pacchetto fa veramente parte di una connessione già avviata e della quale siamo sicuri sia affidabile.

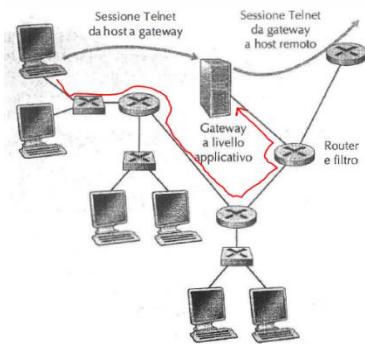
Gateway a livello applicativo

Avete mai sentito del *Big Firewall of China*? Per chi non lo sapesse è un gigantesco firewall che permette quel regime totalitario che sappiamo esiste in Cina. Questo gigantesco muro impedisce la navigazione degli utenti verso specifiche area della rete che hanno contenuti che vanno contro le politiche della cina. Ecco molto ma molto ma molto approssimativamente un gateway a livello applicativo è un esempio.

In particolare i firewall che abbiamo visto sono firewall che arrivano fino al livello di trasporto. Infatti possono scartare pacchetti a seconda del protocollo di trasporto, oppure per indirizzo IP (livello di rete) etch... Adesso vogliamo che i firewall scartino anche al livello applicativo.

Supponiamo di avere degli utenti di una sottorete e di permettere solo ad alcuni che dispongono delle giuste credenziali di accesso di andare a girovagare fuori nel web.

Supponiamo che queste credenziali siano riferite all'applicativo Telnet. Se usassimo solo gli strumenti visti prima, non c'è nessuna entità a livello di rete che controlla dati applicativi come nome utente e password.

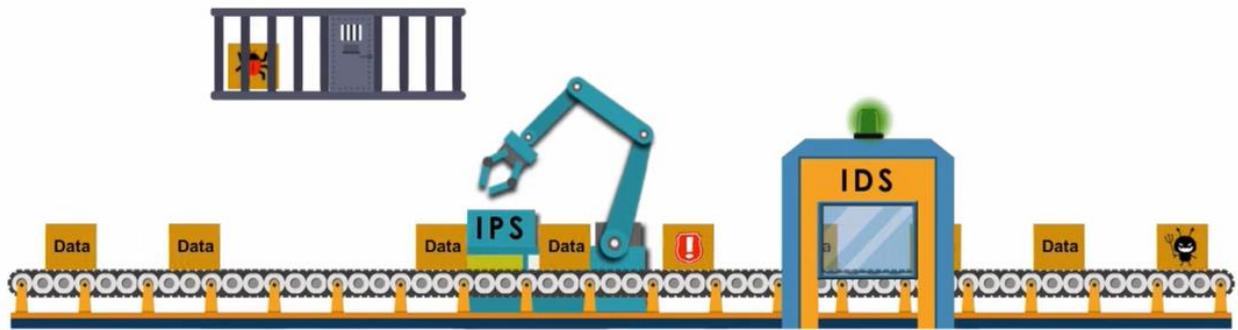


La prima cosa che fa l'host quindi è di instaurare una connessione Telnet verso il Gateway che contiene le credenziali e inviargli le proprie. Il datagramma si dirigerà verso un firewall a filtro visto prima. Questo ha la seguente politica: "faccio passare dalla sottorete interna solo quei datagrammi che riportano l'indirizzo IP del gateway."

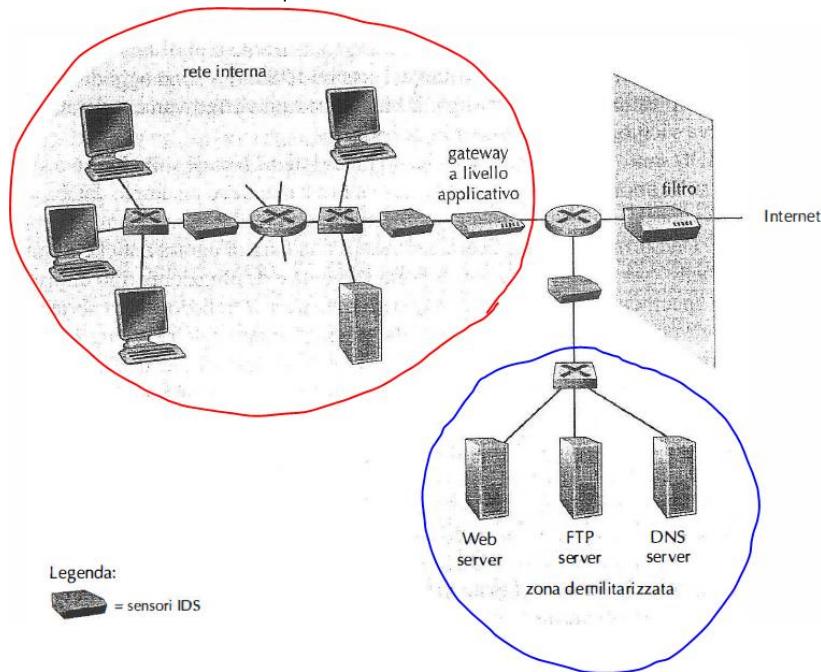
A questo punto il Gateway chiude la connessione se le credenziali sono errate, altrimenti chiede all'utente verso quale altro server nella rete globale vuole connettersi. Notiamo che quindi il Gateway sarà server per l'host (gli invia i dati) e allo stesso tempo suo client (invia gli stessi al server ultimo). Notiamo però che un server gateway è necessario per ogni applicazione come e-mail, FTP, http etch... pertanto un **limite dei gateway a livello applicativo** è un controllo approfondito del pacchetto (fino a livello applicativo) ma solo per una specifica applicazione.

IDS e IPS: sistemi di rilevazione ed eliminazione delle intrusioni

Gli IDS e gli IPS sono sistemi che aumentano il controllo approfondito dei pacchetti, come fa il gateway applicativo.

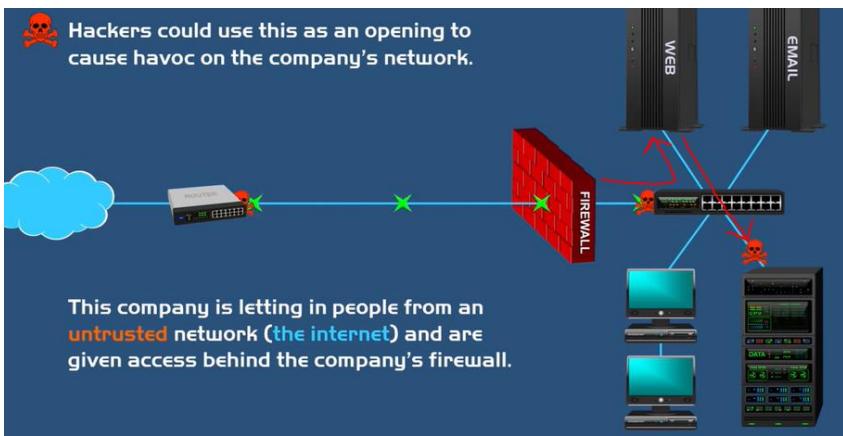


IPS e IDS lavorano insieme e spesso sono un'unica cosa. IDS rileva secondo due modalità (firma o anomalie che dopo vedremo) quei pacchetti che potrebbero essere dannosi avvertendo l'IPS, il quale decide cosa farne.



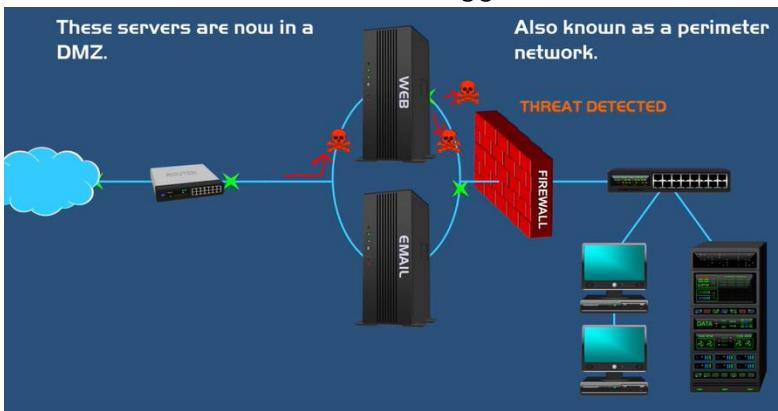
Di solito un'organizzazione dispone la struttura della sua sottorete in due aree. L'area rossa molto controllata prevede un gateway applicativo e due IDS/IPS, mentre un area blu che prevede solo un IDS/IPS. Entrambe le aree hanno un firewall all'inizio. Perché mettere il Web Server, l'FTP e il DNS Server in una zona separata da quella rossa?

Questi 3 elementi sono quelli che ricevono tanto ma tanto gigabit di traffico in poco tempo. Se mettessimo i 3 server nell'area rossa, significa che l'IPS/IDS dovrebbe anche lui subire gigabit di dati da controllare. Siccome il controllo non è veloce, perché vedremo dopo si basa su firme o anomalie e quindi c'è da fare un controllo delicato e "lungo", può capitare che troppo "bombardamento" causi delle perdite/sviste di pacchetti. Pertanto mettiamo i più "ricercati" della sottorete in una **DMZ**, ossia in una zona demilitarizzata.



Inoltre se i server fossero nell'area blu sarebbero facilmente raggiungibili perché appunto sono server quindi è probabile che quasi tutti i pacchetti passino dal firewall, e questo causa una **breccia** nel sistema che potrebbe andare a colpire altre entità della sottorete che

invece non dovrebbero essere raggiunte così facilmente.



Come detto prima la soluzione è metterli in una zona tutta loro.

Un IDS basato sulle **firme** vuol dire che quando un pacchetto passa avverte l'IPS se quel pacchetto risulta essere pericoloso. Per decidere se un pacchetto lo è o meno si guarda la sua firma, ossia la sua caratteristica che se corrisponde a una delle tante contenute in un *database* allora è da marcare. Di solito questo database è riempito da esperti nella sicurezza informatica che registrano le caratteristiche di pacchetti malware noti. E' proprio questo il limite delle firme, che pacchetti malevoli di nuova firma risulterebbero innoqui. Per questo a volte si usa un IDS basato sulle **anomalie**, ossia gli IDS osservano costantemente il traffico e fanno delle deduzioni statistiche, come per esempio un eccessivo traffico di pacchetti ICMP, una eccessiva scansione delle porte etc...ma siccome è difficile distinguere il traffico anomalo da quello non allora si usano le firme.

||| LABORATORIO 1 |||

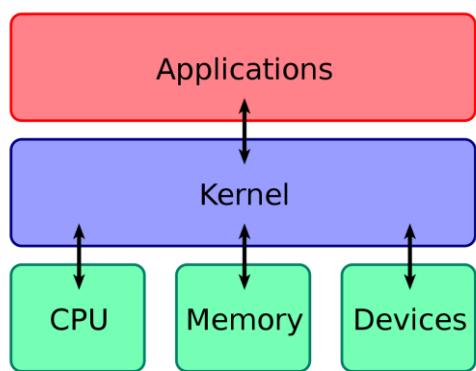
INTRODUZIONE AI SISTEMI UNIX/LINUX

Useremo VirtualBox per far girare Linux Debian 8.2 (scaricabile da <http://www2.ing.unipi.it/c.vallati/files/reti/debian-studenti.ova>)

UNIX

Unix è un sistema operativo *multitutente*, *multiprogrammato*, *modulare* (ossia fatto di programmi semplici, componibili e riusabili) e *portabile* (ossia indipendente dalla architettura hardware di processori sulla quale deve girare).

Il suo nucleo di sistema gestisce l'esecuzione di più processi gestito dallo scheduler a divisione di tempo.



E' diviso in 3 strati.

La parte inferiore è il livello di *hardware* in cui vi sono la CPU, la memoria e i dispositivi di I/O. Al livello intermedio c'è il **kernel**, ossia colui che controlla e gestisce via software l'hardware. Infine al livello superiore c'è il livello **applicativo** in cui risiedono tutte le applicazioni lato utente che di solito sono scritte al livello più alto e interagiscono con il kernel tramite le *system calls*, ossia chiamate di sistema.

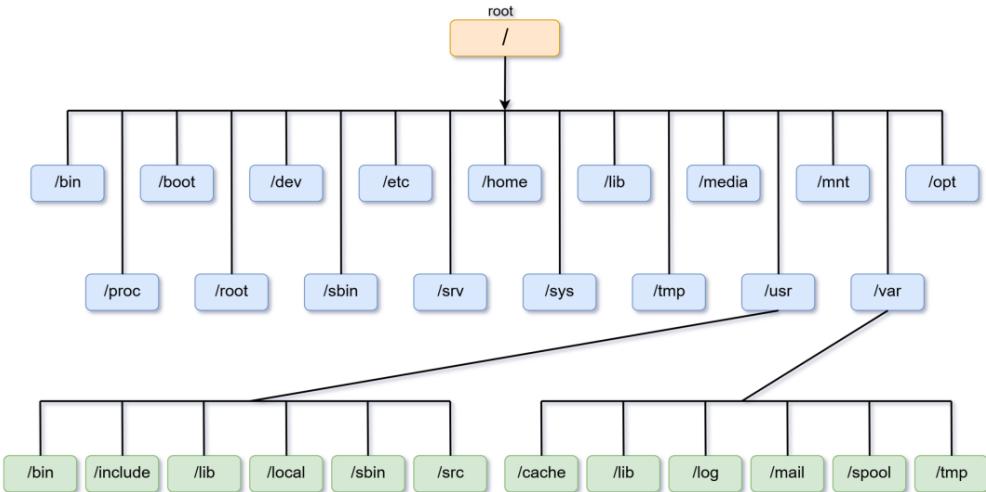
UTENTI

Esistono diverse tipologie di UTENTI all'interno di UNIX.

Un **utente root** è l'amministratore di sistema e gode di qualsiasi privilegio, a differenza dell'**utente normale** che è un semplice utilizzatore del sistema e ha privilegi limitati. Di solito si crea un utente root solo se necessario.

FILE SYSTEM

Un file system è l'insieme delle strutture dati e dei metodi che ci permettono di accedere ai dati e ai programmi presenti in un sistema di calcolo.



Di solito ha una rappresentazione grafica (tramite shell per esempio) che rimanda alla metafora delle cartelle che contengono documenti e altre cartelle.

/ è la cartella/directory principale.

/home è la directory che contiene tutte le directory personali degli utente. Per esempio se creiamo l'utente **Rory** allora si avrà una nuova directory **home/Rory** che contiene tutte le directory, i dati, i programmi e le impostazioni dei programmi da **Rory** utilizzate.

/bin contiene molti programmi per la gestione del sistema, ossia quelli che di solito qualsiasi utente normale può usare tramite riga di comando.

/sbin è analogo al **/bin** ma contiene programmi utili all'amministrazione del sistema e quindi l'utente necessita di essere **root**.

/etc contiene file di configurazione

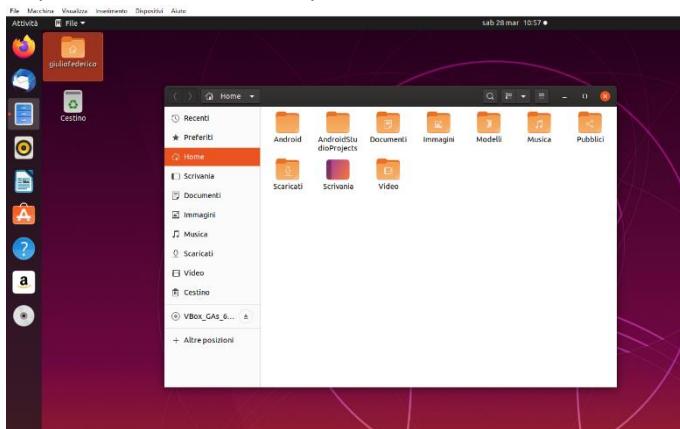
/lib contiene tutte le librerie condivise del sistema

/media rende accessibile i dispositivi rimovibili come CD, DVD, chiavetta USB.

Per descrivere come raggiungere una determinata directory si usa il **path** che può essere *assoluto* (ossia mi parto dalla radice **/home/Rory/...**) oppure *relativo* (ossia mi parto dalla directory in cui mi trovo). Ricorda che Unix è case sensitive.

SHELL

Una *Shell* è un'interprete dei comandi che consente all'utente di richiedere informazioni al SO, e può essere di due tipi:

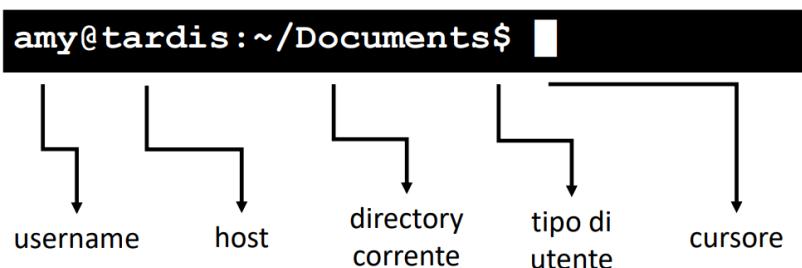


La **shell grafica**, meno potente, ma più intuitiva permette all'utente una semplice navigazione all'interno del file system e la richiesta di esecuzione di programmi tramite semplice click.

```
glulofederico@giuliofederico-VirtualBox:~/Scrivania$ cd ..
glulofederico@giuliofederico-VirtualBox:~$ ls
Android      Documenti  Modelli  Pubblici  Scrivania
AndroidStudioProjects  Immagini  Musica  Scaricati  Video
glulofederico@giuliofederico-VirtualBox:~$
```

La **Shell testuale** invece è molto più potente, ma richiede una certa destrezza che un utente medio non possiede. Esistono diverse shell testuali per Unix come sh, csh, tcsh, **bash**, zsh ma si differenziano solo per l'aspetto e funzioni avanzate perché i comandi sono gli stessi.

Una Shell testuale mostra ripetutamente un prompt:



Il tipo di utente indica se Amy è un utente normale (\$) oppure un utente root (#)

USCIRE DALLA SHELL

Basta scrivere logout oppure exit oppure CTRL+D

ARRESTO E RIAVVIO

Può farlo solo l'utente root e basta scrivere shutdown -h now

COMANDI BASE

cd per spostarsi tra directory

pwd per stampare il percorso *assoluto* della directory corrente

ls (percorso) per stampare il contenuto della directory indicata nel percorso. Con l'opzione:

- **-l** si hanno informazioni sui permessi,

```
drwxr-xr-x 3 giuliofederico giuliofederico 4096 mar 1 21:05 Android
```

d: Indica che la cartella Android è una directory

rwx: la prima terna si riferisce ai permessi che ha l'utente proprietario del file, e può leggerlo (r), modificarlo (w) ed eseguirlo (x)

r-x: la seconda terna si riferisce ai permessi che ha il gruppo di appartenenza del file, e possono leggerlo (r), non possono modificarlo (-) e possono eseguirlo

r-x: la terza terna, in questo caso uguale alla seconda, si riferisce però a tutti gli altri utenti

3: è il numero di hard link associati a quel file (prossimo paragrafo)

giuliofederico: indica il nome del proprietario del file

giulio federico: indica il nome del gruppo

4096: indica la grandezza in byte

mar 1 21:05 indica l'ultima modifica

- **-a** mostra i file nascosti

man (comando) per sapere cosa fa un comando

mkdir (nome directory) crea una nuova directory

rmdir (nome directory) rimuove una directory

cp (src1,scr2,...,dst) copia (file o directory) src1,src2,... dentro dst. Se volessi copiare *tutto* il contenuto di una cartella X dentro a DEST allora cp -r X DEST, ma

questo copierebbe anche X stesso. Quindi in caso scrivere cp -r X/. DEST

mv (cartellaDaRinominare) (nuovoNome) rinomina col nuovoNome la cartellaDaRinominare

mv (cartellaDaSpostare) (cartellaDiDestinazione) il comando mv serve anche a spostare

una cartella dentro un'altra cartella. Infatti se cartellaDiDestinazione non esiste come path allora si intenderà il comando mv come di rinomina piuttosto che di spostamento.

cat (file1.txt) (file2.txt) concatena due file e mostra l'output

rm (file) rimuove un file. Utilizzare *rm -rf cartella* per eliminare la cartella e tutto il suo contenuto in un colpo solo.

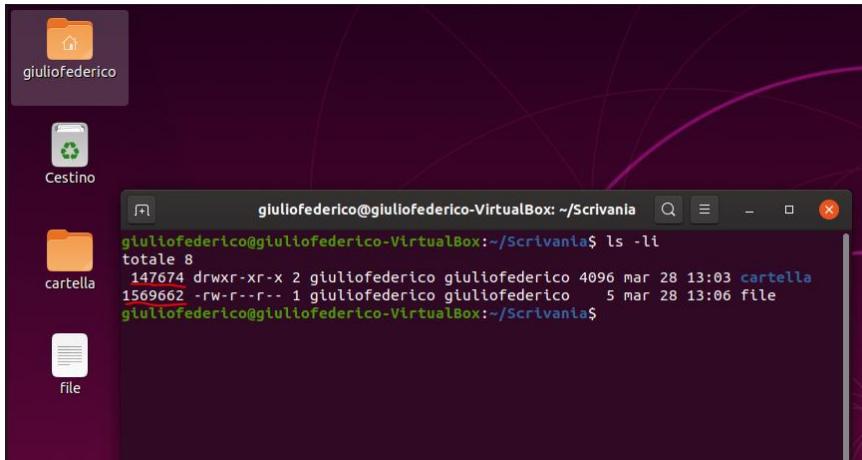
vi file1.txt ,file2.c, ... creo differenti file con differenti estensioni. All'invio si aprirà una

finestra di editor. Premere 'a' per iniziare a scrivervi, ESC per terminare, :w per salvare. Se abbiamo creato più file contemporaneamente possiamo, dopo aver salvato quello di prima, usare :next per passare al successivo.

Per scrivere più istruzioni in un'unica linea basta separare ciascuna con un ;

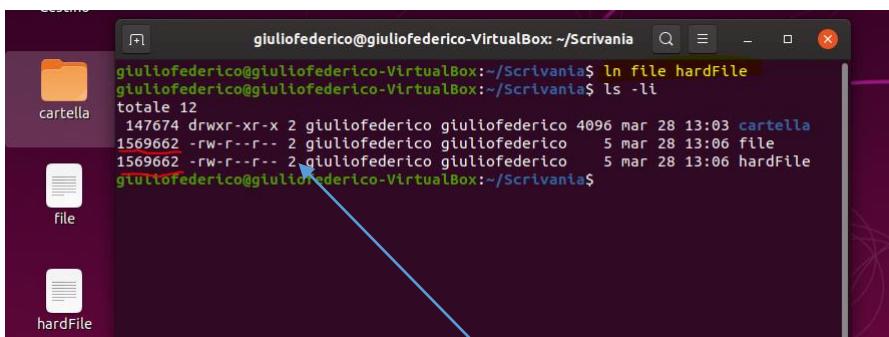
HARD LINK e SOFT LINK

Ad ogni file o cartella in Unix è associato un identificativo detto **inode**.



Il comando **ls -li** mostra non solo le informazioni viste prima, ma aggiunge anche l'inode di ciascun file (segnato in rosso).

Un **hard link** è una vera e propria copia speculare di un file. Non confondete il concetto classico di copia. Se noi dovessimo copiare un file, qualsiasi modifica sul nuovo file di copia non modificherebbe l'originale. Diciamo quindi che un hard link è un altro nome per accedere allo *stesso file*.

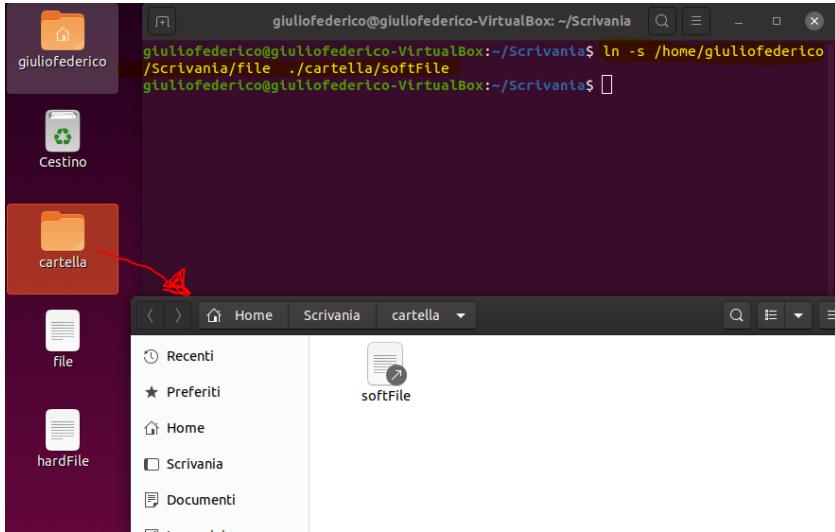


Col comando:
ln fileOriginale hardLink
creo un nuovo hardLink.
Se notiamo, infatti, il
nuovo hardlink di *file* l'ho
chiamato *hardFile* e
possiede *lo stesso* inode
di *file*.

Pertanto una modifica a uno qualsiasi degli hardlink influenza tutti i restanti hardlink creati, compreso il file originario. Inoltre *un file non è totalmente cancellato fino a quando esiste anche solo un hardlink*. Quindi quel numero 2 che vediamo (e che vedevamo nel paragrafo

di prima) indica il numero di hardlink associati a quel file (nota come nella prima immagine era 1).

Un **softlink** invece è un file di peso quasi nullo perché contiene solamente un riferimento (path) al file originario.



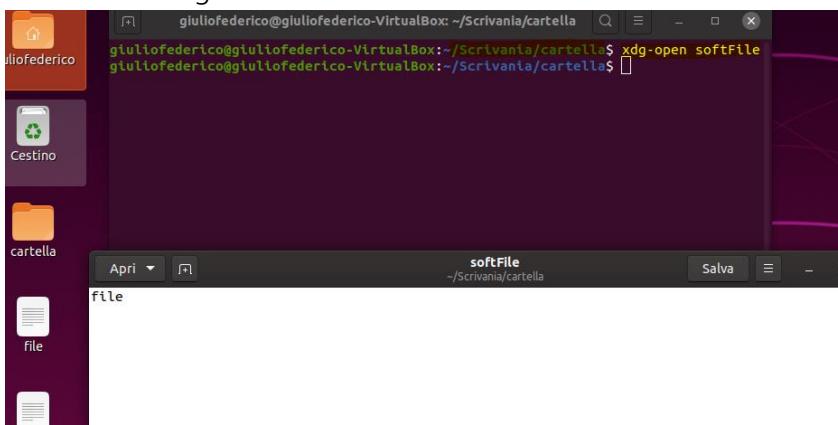
Col comando:

ln -s pathFile pathSoftLink

In pratica nel pathFile indico qual è il percorso di *file* e dico di creare un softlink contenente quel percorso dentro però la cartella *cartella* col nome *softFile*.

Per vedere il percorso che contiene (ossia quello che porta a file) scrivere da dentro *cartella* il comando:
readlink -f softFile

Ma a cosa serve? Possiamo eseguire il file *file* da dentro *cartella* piuttosto che da dove si trova lui in origine.



Col comando *xdg-open* ho aperto il *file* ma da un'altra directory usando il suo softlink chiamato *softFile*.

E' importante capire che un softlink è solo un file contenente un riferimento a dove si trova il file originario così che quando un programma vuole eseguirlo (come *xdg*) sa dove trovarlo. Se il file originario cambia località o viene eliminato, allora il softlink *non si aggiorna* e non serve a nulla. Inoltre il softlink ha un proprio inode.

REDIREZIONE DELL'I/O

In Unix ogni programma quando viene avviato viene dotato di 3 canali di comunicazione verso le periferiche tastiera e video. I tre canali sono *stdin* (per l'input), *stdout* (per l'output) e *stderr* (per gli errori). Internamente ogni programma vede questi 3 canali come dei file a cui può accedere; se legge dal file *stdin* significa che sta leggendo i flussi di byte che arrivano

dalla tastiera al file stdin; se scrive sul file stdout significa che sta scrivendo sul file stdout che è collegato al monitor e quindi è come se quei caratteri fossero trasportati dal file in una parte della memoria video; idem agli errori come stderr, solo che si chiama stderr.

Per esempio supponiamo di scrivere:

```
File Modifica Visualizza Cerca Terminale Aiuto
studenti@studenti:~/Scrivania$ ls -l
totale 8
drwxr-xr-x 2 studenti studenti 4096 mar 28 17:29 a
-rw-r--r-- 1 studenti studenti   67 mar 28 18:44 file.txt
studenti@studenti:~/Scrivania$
```

Quando scrivo il comando `/s -/` il flusso di byte viene mostrato sulla shell perché il flusso proveniente da stdout è collegato al monitor.

Se volessi veicolare tale flusso di byte dentro un file.txt basterà fare così:

```
File Modifica Visualizza Cerca Terminale Aiuto
studenti@studenti:~/Scrivania$ ls -l > file.txt
studenti@studenti:~/Scrivania$
```

Se aprissimo il file.txt ci accorgerebbero che ha un nuovo contenuto che è quello che prima spuntava in output sulla shell.

Idem con i messaggi di errore solo che si usa `2>`. Se scrivo un comando errato, il flusso di byte proveniente da stderr va direttamente sul monitor a meno che non utilizzi `>2`.

NB: che sia stdout o stderr usare `>>` per appendere alla fine, perché l'uso singolo di `>` o `>2` sovrascrive.

Per l'input:

```
File Modifica Visualizza Cerca Terminale Aiuto
studenti@studenti:~/Scrivania$ cat
```

Cat sta aspettando un flusso derivante da stdin perché di standard è quello.

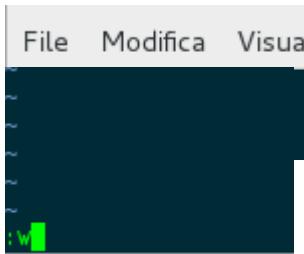
```
File Modifica Visualizza Cerca Tel
studenti@studenti:~/Scrivania$ cat
universo
```

Scrivo universo riempiendo il file stdin.

Posso invece veicolare i flussi di byte in input provenienti da un file che ho creato io piuttosto che dal file stdin riempito dalla keyboard.

```
File Modifica Visualizza Cerca Terminale ,
studenti@studenti:~/Scrivania$ vi file.txt
```

Creo o apro il file.txt col comando vi



Premere a per iniziare a scrivere. Scrivo universo.

Per salvare premo ESC, poi scrivo :w e premo invio. Premo CTRL+Z per ritornare alla shell.

```
File Modifica Visualizza Cerca Terminale Ai
studenti@studenti:~/Scrivania$ cat < file.txt
universo
studenti@studenti:~/Scrivania$
```

Con < redirigo l'input.

```
File Modifica Visualizza Cerca Terminale Ai
studenti@studenti:~/Scrivania$ ls -l | cat
totale 8
drwxr-xr-x 2 studenti studenti 4096 mar 28 17:29 a
-rw-r--r-- 1 studenti studenti 10 mar 28 19:41 file.txt
studenti@studenti:~/Scrivania$
```

Quello che ls-l doveva dare è quello che vediamo dalla seconda riga in poi, ma è stato dato come input a cat che stampa la stessa cosa.

Inoltre il softlink ha un proprio inode.

LESS & head/tail

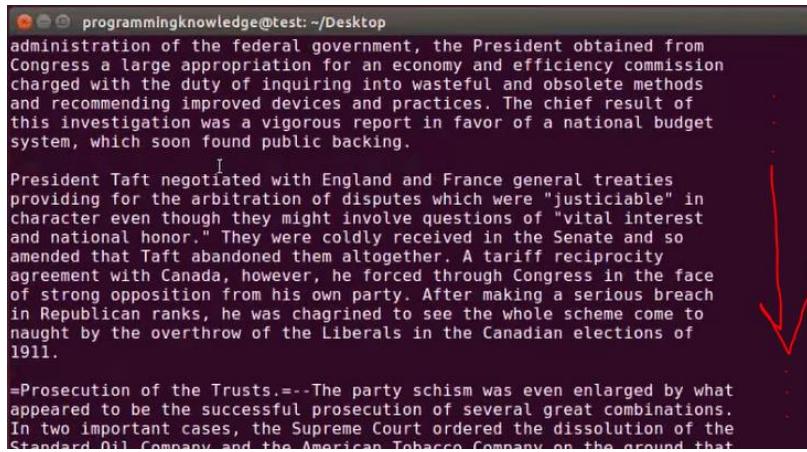
Il comando:

less nome.txt

serve a leggere un po alla volta il file nome.txt e offre varie opzioni per gestire la lettura. Serve quando abbiamo a che fare con grandi file. Senza il comando less dovremmo usare echo oppure cat e nome del file per vederne il contenuto:

```
programmingknowledge@test:~/Desktop/
programmingknowledge@test:~$ cd Desktop/
programmingknowledge@test:~/Desktop$ cat big.txt
```

Appena do invio però...

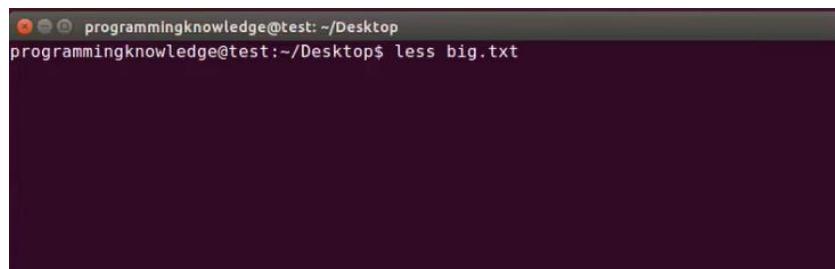
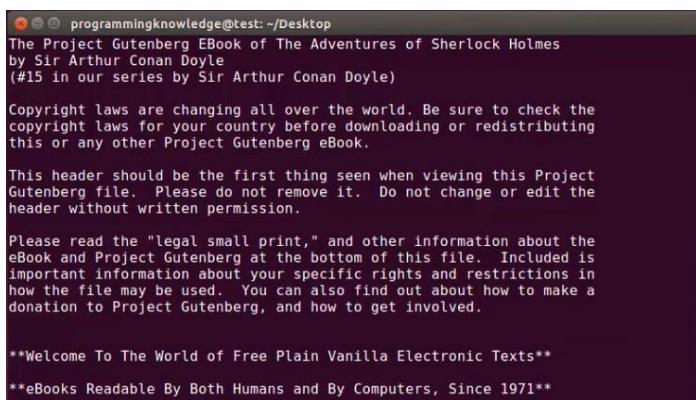


```
programmingknowledge@test: ~/Desktop
administration of the federal government, the President obtained from
Congress a large appropriation for an economy and efficiency commission
charged with the duty of inquiring into wasteful and obsolete methods
and recommending improved devices and practices. The chief result of
this investigation was a vigorous report in favor of a national budget
system, which soon found public backing.

President Taft negotiated with England and France general treaties
providing for the arbitration of disputes which were "justiciable" in
character even though they might involve questions of "vital interest
and national honor." They were coldly received in the Senate and so
amended that Taft abandoned them altogether. A tariff reciprocity
agreement with Canada, however, he forced through Congress in the face
of strong opposition from his own party. After making a serious breach
in Republican ranks, he was chagrined to see the whole scheme come to
naught by the overthrow of the Liberals in the Canadian elections of
1911.

=Prosecution of the Trusts.--The party schism was even enlarged by what
appeared to be the successful prosecution of several great combinations.
In two important cases, the Supreme Court ordered the dissolution of the
Standard Oil Company and the American Tobacco Company on the ground that
```

Non termina quasi più. Come faccio a leggere il file a poco a poco, quasi *sfogliando le pagine* e leggere solo quello che al momento mi serve e non dover aspettare che sia tutto carico?

```
programmingknowledge@test: ~/Desktop
The Project Gutenberg eBook of The Adventures of Sherlock Holmes
by Sir Arthur Conan Doyle
(#15 in our series by Sir Arthur Conan Doyle)

Copyright laws are changing all over the world. Be sure to check the
copyright laws for your country before downloading or redistributing
this or any other Project Gutenberg eBook.

This header should be the first thing seen when viewing this Project
Gutenberg file. Please do not remove it. Do not change or edit the
header without written permission.

Please read the "legal small print," and other information about the
eBook and Project Gutenberg at the bottom of this file. Included is
important information about your specific rights and restrictions in
how the file may be used. You can also find out about how to make a
donation to Project Gutenberg, and how to get involved.

**Welcome To The World of Free Plain Vanilla Electronic Texts**
**eBooks Readable By Both Humans and By Computers, Since 1971**
```

Mi mostra solo un tot di righe partendo dall'inizio e se voglio leggere altro basterà premere la freccia in basso per scorrere la pagina verso sotto aggiungendo una nuova riga in più.

Inoltre esistono due comandi **head** e **tail** che mostrano le prime n righe (**head**) o le ultime n righe (**tail**). Per esempio:

head file.txt

mostra le prime 10 righe del file.txt perché di default n=10.

tail -2 file.txt

mostra le ultime due righe del file.txt

SU e SUDO

Siamo sulla shell e momentaneamente stiamo utilizzando un certo utente. Se vogliamo passare a un altro utente useremo il comando:

su nomeutente

Ci verrà chiesta la password.

Se invece vogliamo accedere all'utente root di sistema basterà solo scrivere:

su

Esempio:

```
File Modifica Visualizza Cerca Terminale
studenti@studenti:~/Scrivania$ su
Password:
root@studenti:/home/studenti/Scrivania#
```

Ho evidenziato i due simboli (\$ e #) che indicano che stavo operando come utente normale e poi come utente root.

Il comando **sudo -u nomeUtente comando** permette di eseguire un comando come se fossimo altri utente. Può richiedere la password.

ESERCIZI: UNIRE 3 FILE.TXT IN UN UNICO

cat f1.txt f2.txt f3.txt > ftot.txt

cat prende in ingresso f1.txt...f3.txt e veicoliamo il risultato in un nuovo o già esistente file. Usare **>>** se si vuole appendere invece che sovrascrivere (magari ftot esisteva già e aveva già un suo contenuto).

ESERCIZI: SCRIVERE A CAPO

echo -e "Milano \nPerugia e Asti" > fcitta.txt

anche se la redirezione verso un nuovo o già esistente file dell'output derivante, basta usare echo con l'opzione **-e**, con le virgolette e l'uso dell'escape **\n**

ESERCIZI: ORDINARE ALFABETICAMENTE

La funzione **sort** prende in ingresso uno o più file txt e li ordina *per righe*. Se prende in ingresso più file li tratta come uno unico. L'ordinamento è fatto per righe nel senso che si guardano solo le iniziali di ciascuna. Se R1 e R2 sono due righe, si scambiano le righe solo se l'iniziale di R1 è alfabeticamente superiore all'iniziale di R2.

NB1: Le singole righe non vengono ordinate al loro interno

NB2: Nell'ordine alfabetico, i numeri hanno precedenza alle lettere

Ho creato due file.

```
File Modifica Visualizz prova.txt
B Alice
Z Vittorio
A Pietro
~
```

```
File Modifica Visualizza Cerca
S Silvana C Carlo 1 Roberto
A Patrick
I Ben
234 Zodiac
1111 Kevin
111 Tom
~
```

prova2.txt

Scrivo:

sort prova.txt prova.txt prova2.txt

E ottengo:

```
studenti@studenti:~/Esercitazione$ sort prova.txt prova2.txt
1111 Kevin
111 Tom
1 Ben
234 Zodiac
A Patrick
A Pietro
B Alice
S Silvana Carlo I Roberto
Z Vittorio
studenti@studenti:~/Esercitazione$
```

Notiamo le seguenti cose:

- 1) Il controllo si fa per righe, ma se hanno l'iniziale uguale si controlla la seconda e così via. Per esempio 1111 Kevin e 111 Tom avevano le prime 3 uguali, ma vince 1111 Kevin perché ha un 1
- 2) I numeri hanno priorità rispetto i caratteri alfabetici
- 3) Le righe singolarmente non vengono ordinate all'interno

Se si vuole un *ordine inverso* bisogna inserire l'opzione –r

ESERCIZI: METACARATTERI

I metacaratteri più usati sono:

* indica "qualsiasi carattere o numero"

- Es. ls * indica di mostrare qualsiasi file nella directory
- Es. ls a* indica di mostrare qualsiasi file nella directory che cominci con a e il resto sia qualsiasi cosa e in qualsiasi lunghezza.
- Es. ls f*c indica di mostrare qualsiasi file nella directory che cominci con f e il resto sia qualsiasi cosa e in qualsiasi lunghezza ma termini con c

[] indica un range di caratteri/numeri che può assumere

- Es. ls [a-d, f, S]* indica di mostrare qualsiasi file nella directory che cominci con una lettera tra a e d, oppure f, oppure S maiuscola e il resto sia qualsiasi cosa e in qualsiasi lunghezza.

|| LABORATORIO 2



CONFIGURAZIONE DI INTERFACCE DI RETE, GATEWAY E DNS

IP

Il comando **ip** è simile al comando *ipconfig*. Con questo comando si possono gestire le interfacce di rete, impostare gateway, DNS etc..

Per visualizzare tutti gli indirizzi IP associati alle interfacce di rete si utilizza il comando:

`ip addr show`

```
giuliofederico@giuliofederico-VirtualBox:~/Scrivania$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:0e:7d:62 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
        valid_lft 86398sec preferred_lft 86398sec
    inet6 fe80::835:627b:47c5:619e/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
```

La prima interfaccia è chiamata */o* che sta per *localhost/loopback*. Localhost è infatti il nome dell'indirizzo IP 127.0.0.1 (sempre lo stesso) associato a una particolare interfaccia di rete chiamata **loopback**. Non esiste una interfaccia di rete del genere, ma è virtuale e serve a permettere di comunicare con noi stessi. In realtà, trovandoci su una macchina virtuale tutte le interfacce sono virtuali. E' molto probabile che la seconda interfaccia *enp0s3* sia in una macchina *non* virtuale chiamata *eth0*.

Le interfacce possiedono dei **flag** all'interno delle parentesi angolari `<>`. Quando troviamo **BROADCAST** significa che l'interfaccia supporta l'invio verso tutti quelli collegati alla stessa rete; se troviamo **MULTICAST** significa come il broadcast che possiamo inviare verso più host invece che ad un unico (unicast), ma non per forza a tutti ma magari a un gruppo; **UP** indica che l'interfaccia è attiva; **LOWER_UP** indica che nell'interfaccia è collegato un cavo di rete ed è quindi connessa alla rete. Successivamente **mtu** indica la grandezza in byte dei datagrammi IP; **link/** indica l'indirizzo MAC e **brd** l'indirizzo broadcast. A livello di rete questo può anche non essere il solito 255.255.255.255 ma può essere il cosiddetto indirizzo broadcast orientato che presenta il prefisso della rete e poi 255 alla fine.

In questo momento ogni interfaccia possiede un *indirizzo IPV4* visibile nella voce *inet* (tralasciamo *inet6* che si riferisce a IPV6). Questi indirizzi sono dati automaticamente dal DHCP. Infatti quando lo connettiamo alla rete LAN, il pc invia un messaggio DHCP request con porta di destinazione 67 e sorgente 68 veicolato con UDP, incapsulato in un datagramma IP con indirizzo broadcast 255.255.255.255 e sorgente 0.0.0.0 (sono io), a sua volta incapsulato in un frame ethernet con destinazione broadcast FF:FF...quando il router riceve il messaggio dallo switch lo fa salire fino al livello applicativo dove gira il server DHCP che invia il DHCP ACK contenente l'indirizzo IP da assegnare, quello del DNS, quello del gateway e quello della netmask.



Se non usasse il DHCP lo avrebbe fatto *staticamente* in modo manuale. Nella figura a sx *non* si sta usando il DHCP e si fa tutto a mano. Mentre semmai si fosse fatto check sull'opzione "ottieni un IP automaticamente" allora ci avrebbe pensato il DHCP.

Quello che fa il DHCP è di settare tutto **dinamicamente**. Possiamo anche farlo **staticamente** ma dobbiamo non solo settarci un nostro indirizzo IP (stando attenti al conflitto con gli altri), ma anche ricavarci la netmask, l'IP del gateway e del DNS.

In generale la struttura del comando è:

ip addr add <ip address>/<netmask> dev <interface>

Dove sono da completare solo ciò che è tra parentesi angolari. IP address è il nuovo ip statico da assegnare, netmask è la sottorete e interface indica il nome dell'interfaccia a cui si sta assegnando quell'indirizzo IP.

```
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 08:00:27:47:fe:bc brd ff:ff:ff:ff:ff:ff
    inet 120.1.12.5/24 brd ff:ff:ff:ff:ff:ff scope global eth0
        valid_lft forever preferred_lft forever
    inet 120.1.12.5/12 brd ff:ff:ff:ff:ff:ff scope global eth0
        valid_lft forever preferred_lft forever
```

Ho attualmente un indirizzo IP 120.1.12.5/24 e netmask /24

```
root@studenti:/home/studenti# ip addr add 180.14.10.9/255.255.0.0 dev eth0
root@studenti:/home/studenti# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 0.0.0.0 scope host lo
        valid_lft forever preferred_lft forever
    inet ::1/128 brd :: scope host lo
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 08:00:27:47:fe:bc brd ff:ff:ff:ff:ff:ff
    inet 120.1.12.5/24 brd ff:ff:ff:ff:ff:ff scope global eth0
        valid_lft forever preferred_lft forever
    inet 120.1.12.5/12 brd ff:ff:ff:ff:ff:ff scope global eth0
        valid_lft forever preferred_lft forever
    inet 180.14.10.9/16 brd ff:ff:ff:ff:ff:ff scope global eth0
        valid_lft forever preferred_lft forever
```

Nota che la netmask può anche essere scritta in forma breve come /16
Abbiamo **multipli indirizzi IP**, il che va contro a quanto studiato. Ma non ci deve spaventare. Il comando ip addr add serve proprio a evitare di comprare 5 schede di rete se si vuole essere raggiunti in 5 differenti modi/indirizzi. Pertanto ad ogni scheda di rete è possibile associare più indirizzi IP.

Per *rimuovere* un certo indirizzo IP basta scrivere in generale:

ip addr del <ip address>/<netmask> dev <interface>

Se si vuole *rimuovere tutti gli indirizzi IP* basta scrivere:

`ip addr flush dev <interface>`

Per *abilitare/disabilitare* un interfaccia (ossia avere il flag UP) basta scrivere in generale:

`ip link set <interface> up` per abilitarla
`ip link set <interface> down` per disattivarla

Tutte queste modifiche statiche non sono memorizzate dopo un riavvio. Il comando `ip` è spesso usato solo per modifiche temporanee e non permanenti. Se si vuole che all'avvio del sistemi si trovi l'interfaccia in un certo stato bisogna andare a modificare il file `/etc/network/interfaces` (usiamo vi o nano).

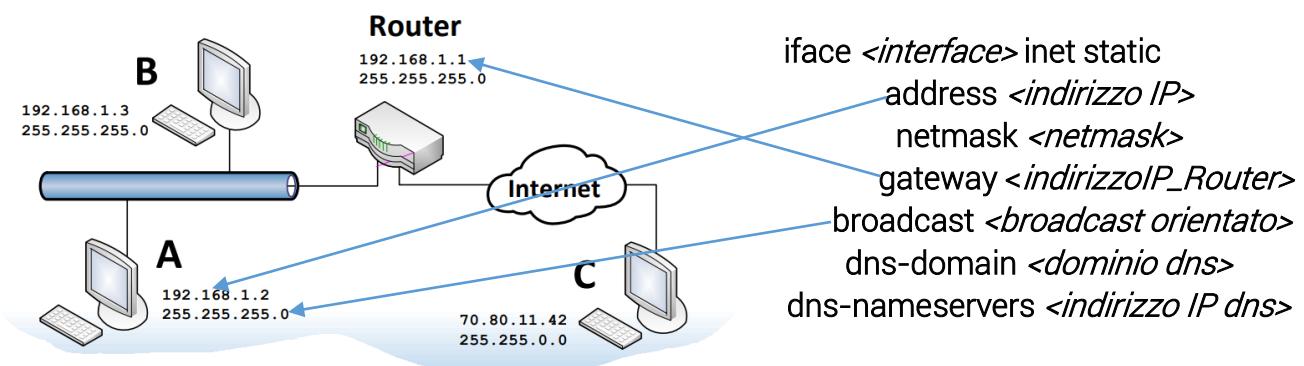
Questo è un file che dobbiamo modificare.

Supponiamo di volere che l'interfaccia eth0 abbia determinate caratteristiche perenni. Per attivare le interfacce con i comandi `ifup <interface>` (fatto automaticamente all'avvio del sistema) o disattivarle `ifdown <interface>` bisogna marcare le interfacce con:

`auto <interface>`

In questo modo i comandi ifup e ifdown lavoreranno solo sulle interfacce che nel file interfaces sono marcate con auto.

Adesso l'interfaccia può essere inizializzata (inet) manualmente e quindi staticamente scrivendo:



Se si vuole lasciare che sia il DHTC a configurare il tutto dinamicamente basta scrivere:

`iface <interface> inet dhcp`

Se si usa `ifup -a` si attivano tutte le interfacce che sono precedute da auto (ma ifup -a è chiamato automaticamente all'avvio del sistema).

Inoltre quando un host A deve inviare un pacchetto deve scegliere verso quale interfaccia di rete di uscita inviarlo.

Col comando:

`ip route show`

viene mostrata la tabella di routing di ogni host.

```
studenti@studenti:~$ ip route show
default via 192.168.1.1 dev eth0
169.254.0.0/16 dev eth0  scope link  metric 1000
192.168.1.0/24 dev eth0  proto kernel  scope link  src 192.168.1.2
```

La prima riga indica il **default gateway**, e si legge: "per ogni indirizzo IP che scelgo come destinazione di invio, usare il default gateway di indirizzo 192.168.1.1 raggiungibile all'uscita eth0". Il default gateway, seppure visibile come prima riga, rappresenta l'alternativa ultima a cui inviare il pacchetto se in una delle successive righe non esiste una rotta esplicita per quell'indirizzo. Per settare il default gateway si usa il comando:

```
ip route add default via 192.168.1.1
```

Per aggiungere invece una **rotta esplicita** per un certo indirizzo IP o bloccoIP basta usare il comando:

```
ip route add 192.168.1.0/24 dev eth0
```

Una rotta esplicita serve a instradare un pacchetto attraverso una interfaccia di rete quando l'host di destinazione appartiene alla sottorete e quindi non abbiamo bisogno del default gateway per andare fuori.

Per **scoprire una rotta** dato un indirizzo ip si utilizza:

```
ip route get <indirizzoDaScoprire>
```

Per esempio ip route get 192.168.1.2, che appartiene alla propria sottorete, dato che abbiamo aggiunto una rotta esplicita ci dirà essere:

```
192.168.1.2 dev eth0 src 192.168.1.2
```

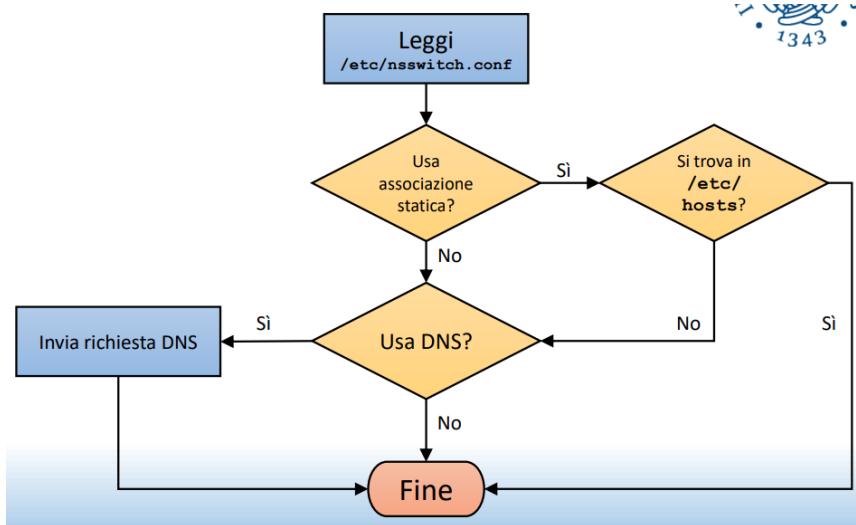
ossia "la rotta per andare verso 192.168.1.2 è l'uscita eth0 di indirizzo IP 192.168.1.2".

Mentre se dovessi scrivere ip route get 170.254.0.1, che non appartiene alla propria sottorete, mi darebbe:

```
170.254.0.1 via 192.168.1.1 dev eth0 src 192.168.1.2
```

ossia "la rotta 170.254.0.1 è raggiungibile prendendo l'uscita eth0 di indirizzo IP 192.168.1.2 attraverso però il router di indirizzo IP 192.168.1.1".

Invece di utilizzare indirizzi IP potremmo usare dei **nomi**. Così come per ottenere gli indirizzi IP si può ricorrere al DHCP o lo si può fare staticamente nel file interfaces, anche per la **risoluzione dei nomi** si può ricorrere alla stessa logica:



Questo schema dice come si comporta un calcolatore quando deve tradurre un nome nel suo indirizzo IP. La prima cosa che fa è andare a guardare il file `/etc/nsswitch.conf` che contiene delle voci delle quali a noi interessa **hosts**. In questa riga ci sono le sorgenti di dove trovare le informazioni per gli host (quindi la traduzione che cercavamo). Un esempio di quello che potremmo trovare che rappresenta lo schema su è questo:

hosts: files dns

Questo vuol dire che la prima sorgente dove trovare la coppia nome-indirizzoIP da controllare si trova in un file, e se non vi si trova nulla si contatta direttamente il DNS. Il file dove staticamente ogni elaboratore può tenere le traduzioni per evitare di contattare il DNS e tutta la catena gerarchica si trova in `/etc/hosts`. Pertanto come modalità abbiamo scelto di controllare prima questo file così da velocizzare la ricerca e in caso negativo di contattare il DNS.

Ma i DNS avranno anche loro un indirizzo IP. Un calcolatore sa che quando dovrà contattare un DNS, gli indirizzi IP di quelli che può contattare si trovano in `/etc/resolv.conf`.



SERVER DHCP e TEST DI CONNETTIVITA'

SERVER DHCP

Quanto detto prima circa il server DHCP era **lato client**. Abbiamo detto che per una certa interfaccia di rete si usava DHCP per ottenerne l'indirizzo IP e tutto il resto contattando il server DHCP. Il client non ha bisogno di configurare il nome o l'indirizzo del server DHCP perché i suoi messaggi di richiesta sono e saranno sempre in broadcast.

In questo paragrafo passiamo al **lato server**, ossia vediamo come implementare un *server DHCP* sulla nostra macchina che risponda ai client che ci contattano per ricevere l'indirizzo IP e quant'altro visto prima.

La prima cosa da fare è installare il pacchetto che contiene il necessario per creare un server DHCP:

```
apt-get install isc-dhcp-server
```

Terminata l'installazione dobbiamo andare nel file */etc/default/isc-dhcp-server* in cui, aprendolo, dovremmo trovare vuoti questi due campi:

```
# Don't use options -ci or -pr here; use DHCPv
CPD_PID instead
#OPTIONS=""

# On what interfaces should the DHCP server (dhcpcd)
# requests?
#           Separate multiple interfaces with spaces, e.g.
#           eth1".
INTERFACESV4=""
INTERFACESV6=""
```

Questi indicano l'indirizzo in formato IPv4 o IPv6 della scheda di rete che sarà in ascolto dei client che cercheranno di contattare il nostro server DHCP che stiamo costruendo.

Per vedere le mie schede di rete messe a disposizione, lo sappiamo basta fare *ip addr show*.

```
giuliofederico@giuliofederico-VirtualBox:/etc/default$ ip add
r show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state U
NNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
        inet 127.0.0.1/8 brd 127.255.255.255 scope host lo
            valid_lft forever preferred_lft forever
    inets 1:1/28 scope host
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc f
q_codel state UP group default qlen 1000
    link/ether 08:00:27:0e:7d:62 brd ff:ff:ff:ff:ff:ff
        inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic nop
refixroute enp0s3
            valid_lft 78726sec preferred_lft 78726sec
        inets fe80::835:627b:47c5:619e/64 scope link noprefixout
e
            valid_lft forever preferred_lft forever
```

Come scheda di rete voglio scegliere la *enp0s3*. Vediamo quindi come fare a dire che sarà quella scheda di rete ad accogliere le DHCPDISCOVER e DHCPREQUEST.

Ritorniamo al file `isc-dhcp-server` però apriamolo con `vi` per modificarlo ma utilizziamo `sudo` perché dobbiamo essere amministratori:

`sudo vi isc-dhcp-server`

```
# Additional options to start the server
# Don't use options -CPD_PID instead
#OPTIONS=""

# On what interfaces should the server
# respond to DHCP requests?
#       Separate multiple interfaces by commas
#       eth1".
INTERFACESv4="enp03" ←
INTERFACESv6=""
```

Ho aggiunto l'interfaccia.

Andiamo ora a fare le configurazioni che ci servono per dire al nostro server chi deve servire, per quanto tempo e altro ancora.

Andiamo nel file `/etc/dhcp/dhcpd.conf` e apriamolo con `sudo nano` (nuovo editor che vi consiglio invece di `vi`) per modificarlo. La prima cosa da fare è la seguente:

```
→ # option definitions common to all supported networks...
→ #option domain-name "example.org";
→ #option domain-name-servers ns1.example.org, ns2.example.or
```

Sono campi con i quali possiamo essere rintracciati dai DNS se invece dell'IP abbiamo un dominio.

Ma non avendolo dobbiamo commentare le due righe.

```
→ # If this DHCP server is the official
→ # authoritative server for a network, the authoritative directive
→ → authoritative;
```

Andiamo invece a decommentare la riga `authoritative` perché siamo gli unici server DHCP nella rete locale, e `authoritative` serve proprio a dire questo.

Andiamo a fare la configurazione più importante:

```
# A slightly different configuration for an internal subnet
subnet 10.5.5.0 netmask 255.255.255.0 {
    range 10.5.5.100 10.5.5.200;
    # option domain-name-servers ns1.internal.example.org;
    # option domain-name "internal.example.org";
    option subnet-mask 255.255.255.0;
    option routers 10.5.5.1;
    # option broadcast-address 10.5.5.31;
    default-lease-time 600;
    max-lease-time 7200;
}
```

Per **subnet** si intende la sottorete a cui appartiene il server DHCP e per **netmask** il solito. Con **range** si indica il range di indirizzi IP da consegnare a chi ne fa richiesta; in questo caso posso gestire fino a 100 indirizzi IP. Con **default-lease-time** si intende la durata standard che il server

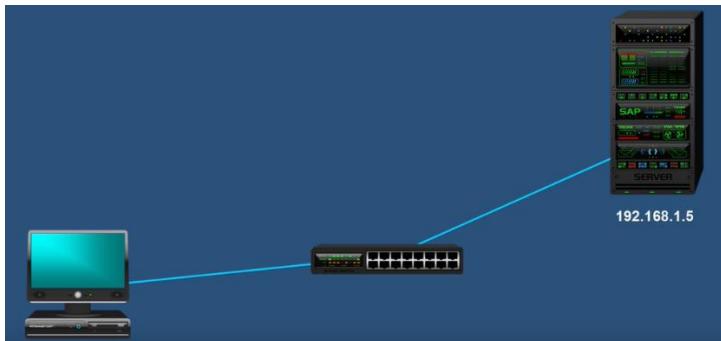
concede a un host nel tenere uno specifico indirizzo IP; se però quell'host richiede specificatamente un'altra durata, allora il parametro **max-lease-time** specifica quanto al massimo può concedere.

Terminate le configurazioni avviamo il server:

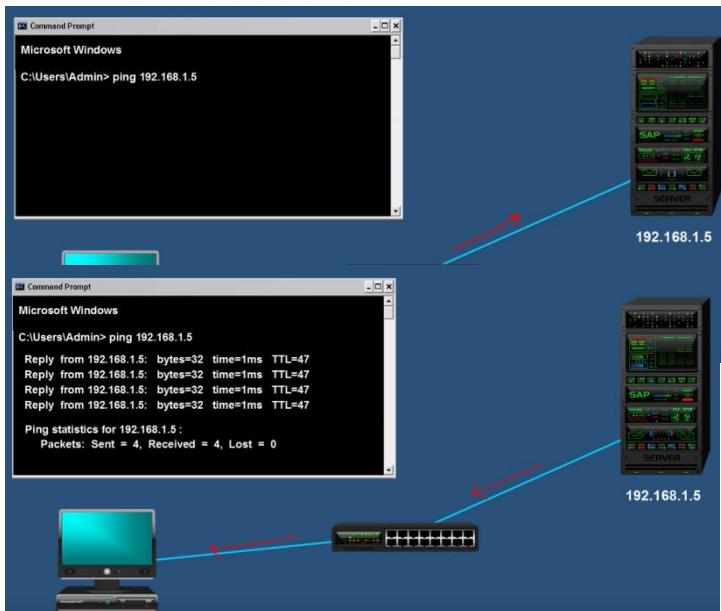
`systemctl restart isc-dhcp-server.service`

PING

Il comando **ping** è una utility di amministrazione. Manda ad un certo indirizzo IP (o nome simbolico che poi verrà tradotto dal DNS) dei pacchetti ICMP per testare la connettività in rete verso quella destinazione specificata.



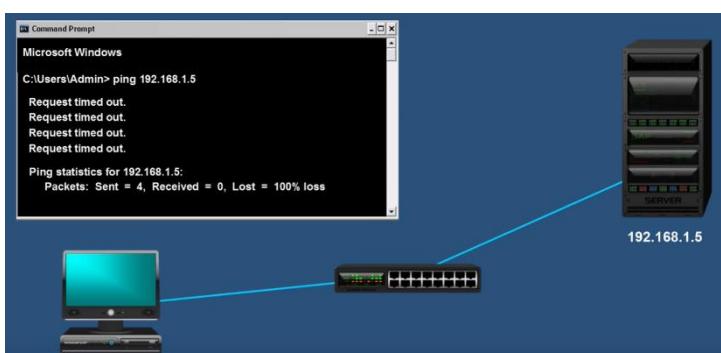
Supponiamo di avere la seguente situazione. Voglio testare la connettività verso l'host 192.168.1.5 (non so il nome quindi utilizzo l'IP).



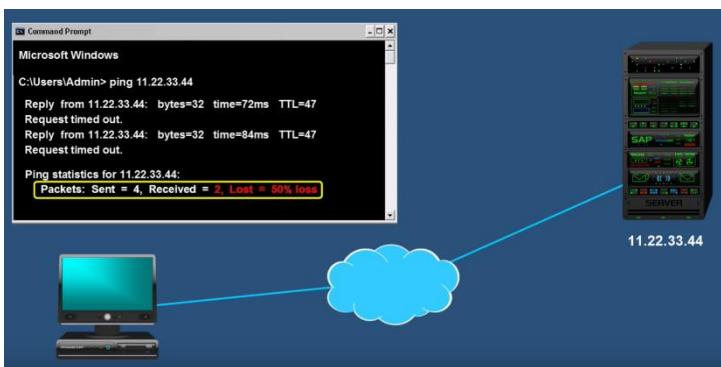
Abbiamo inviato 4 pacchetti ICMP di tipo **ICMP echo Request** all'host di destinazione.

La destinazione ci invia la risposta di tipo **ICMP echo reply** per ogni pacchetto ricevuto. In questo caso ne ho inviati 4 (da 32 byte) e ne ho ricevuti 4 con un RTT (indicato da time) di 1ms per ognuno.

Cosa succede se non ottengo risposta? Ci sono varie possibilità:



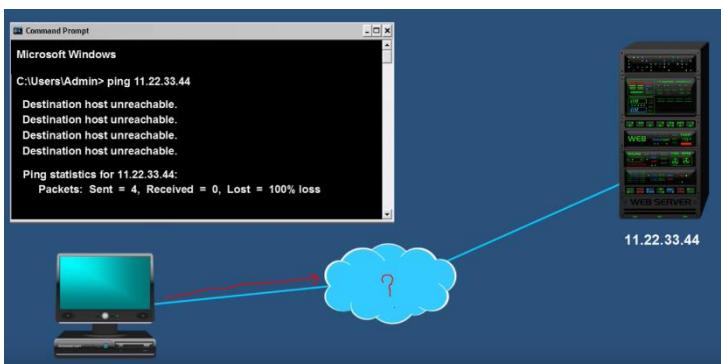
Quando ci viene mostrato il messaggio **request timed out** vuol dire che:
1) il server è spento
oppure
2) il server ha un firewall che non lascia passare i nostri pacchetti



Una seconda possibilità è di ricevere risposta solo per alcuni pacchetti.

Questo può essere dovuto a:

- 1) congestione di rete
- 2) collegamenti difettosi, modem difettosi o connessioni non stabili.



Una terza possibilità, indicataci quando ci arriva un messaggio **Destination host unreachable**, indica che i pacchetti sono arrivati al router ma:

- 1) il pc non è collegato a internet
- 2) il router non ha alcuna informazione su come instradare il pacchetto verso la destinazione.

Di norma ping invia **un numero infinito** di pacchetti. Se però utilizziamo l'opzione:

`ping -c <quanti> <indirizzoIP_destinazione>`

Inoltre con l'opzione `-s <numeroByte>` si possono decidere le dimensioni dei byte da mandare (ovviamente il pacchetto sarà di dimensioni pari a `numeroByte+headerICMP`)

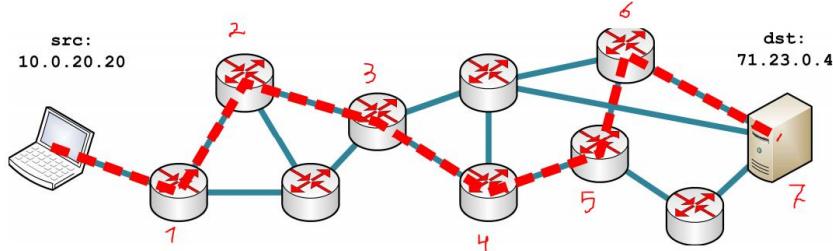
Allora si fermerà a *quanti* pacchetti.

Con ping si può **testare la propria connettività in rete** tramite per esempio `ping indirizzoIP`, oppure contemporaneamente **testare anche il DNS** tramite per esempio `ping yahoo.com`. Un altro uso è quello di **testare la propria scheda di rete** facendo il ping sulla stessa, ossia `ping localhost` (oppure `ping 127.0.0.1`).

TRACEROUTE

Nel comando **ping** abbiamo visto che ogni record aveva una **TTL** ossia una *time to live* che specifica quanti router può al massimo attraversare prima di morire/non essere più valida. Per esempio se TTL=54 allora può attraversare 54 router. Ogni router quindi, quando riceve un ICMP da ping decrementa il TTL e poi propaga il pacchetto, ma colui il quale si trova ad avere, dopo il decremento, un TTL pari a 0 non propaga più il pacchetto ma invia un **TIME EXCEEDED**, ossia un messaggio di errore ICMP *con il suo indirizzo alla sorgente che ha inviato il pacchetto*.

Una utility che sfrutta questo meccanismo è **traceroute**. Grazie a questo meccanismo riesce a mappare il percorso di ogni pacchetto preparando ogni volta pacchetti con TTL sempre crescenti fino a quando non si arriva alla destinazione.



Con ping so che "va tutto bene" nell'interagire con la destinazione di indirizzo IP 71.23.0.4, ma adesso voglio vedere nel mezzo gli indirizzi IP di tutti i router (o meglio delle singole interfacce di rete che attraverso) nel percorso da me fino a dst.

traceroute 71.23.0.4

- Prepara un pacchetto con TTL=1, e lo invia. Il router1 lo riceve e decrementa TTL. Adesso TTL è nullo e quindi non propaga il pacchetto ma manda un TimeExceeded con il suo indirizzo IP: indirizzoIP_router1
In questo modo la sorgente riceve l'ICMP e sa che il primo router ha quell'indirizzo IP.
- Prepara un pacchetto con TTL=2, e lo invia. Il router1 lo riceve e decrementa TTL. Adesso TTL=1 e quindi può proagarlo al router2. Il router2 decremente TTL che adesso è a zero quindi non lo propaga ma prepara il messaggio ICMP di TimeExceeded con l'indirizzoIP: indirizzoIP_router2
- Prepara TTL=3...
- Prepara TTL=4...5...6...
- Prepara TTL=7. Quando il la dst invia il TimeExceeded allora termina tutto.

Il traceroute può dare come output il **simbolo ***

Questo vuol dire che il pacchetto non è stato mai ricevuto, oppure è stato perso a causa della congestione o bloccato da firewall. Però a volte è semplicemente per il fatto che le porte UDP di ogni hop sono bloccate mentre quelle ICMP sono permesse. Infatti può succedere che ping funziona ma traceroute no, o ancora che traceroute funziona su windows mentre su linux no. Questo perché traceroute su windows prepara pacchetti ICMP mentre traceroute su linux li prepara come UDP. Per forzarlo ad usare pacchetti ICMP usare l'**opzione -I**.

SCHEDE DI RETE SPIONE

Di norma le schede di rete che vedono arrivarsi un pacchetto, controllano sei il MAC address di destinazione è quello loro, e se è così lo prelevano altrimenti scartano il pacchetto. Per avere interfacce che "non si fanno solo i fatti loro", le ho chiamate *interfacce spione*, basta scaricare un software:

```
apt-get install tcpdump
```

Questo software riesce a settare una interfaccia in una modalità che tecnicamente non si dice spiona, ma **promisqua**, ossia visualizza tutti i pacchetti che circano sulla rete locale.

Col comando:

tcpdump –i <interfaccia>

impostiamo una determinata interfaccia come promisqua. Col comando:

tcpdump –c <quanti>

impostiamo il numero di pacchetti che si vuole al massimo osservare. Col comando:

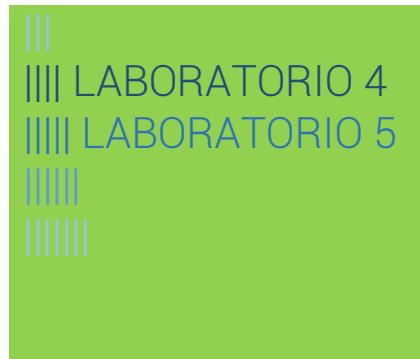
tcpdump –w <nome_file>

decidiamo dove scrivere il pacchetto i-esimo che ha visualizzato e per leggere tale file basta fare **tcpdump –r <nome_file>**.

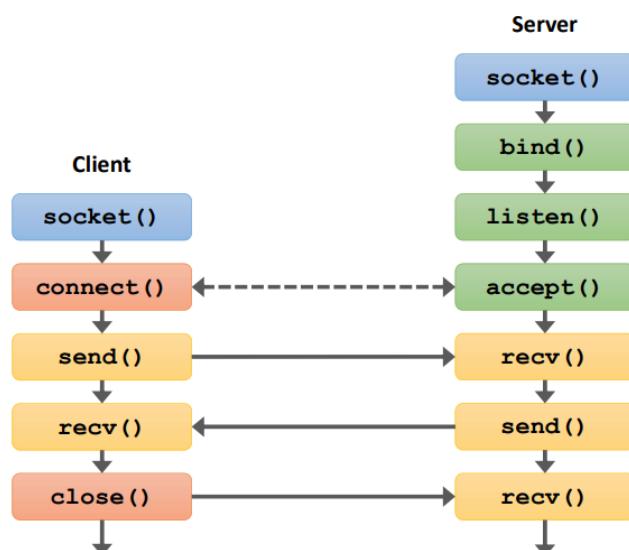
Possiamo pure decidere una nicchia di cosa osservare, e quindi impostiamo dei filtri. Quelli possibili sono:

tcpdump port <numeroPorta> osserva solo i pacchetti che hanno porta di
dest=numeroPorta

tcpdump host <indirizzoIpHost> osserva solo l'host di indirizzo ip *indirizzoIpHost*
etc...



PROGRAMMAZIONE SOCKET



Seguiremo questo schema. Ogni paragrafo si occuperà di una determinata funzione e alla fine di ogni paragrafo aggiungeremo al codice una parte in più che realizza quella funzione.

NB: il C non permette la dichiarazione di variabili al di là dell'inizio di ogni blocco {}, eppure certe versioni (come la mia) non creano problemi, ma ovviamente potrebbero crearli in termini di portabilità del codice. Pertanto io dichiarerò le variabili come normalmente fatto col c++, ma dovete ricordare che col c si dichiarano tutte all'inizio.

SOCKET TCP

SERVER & CLIENT

CREARE SOCKET: socket()

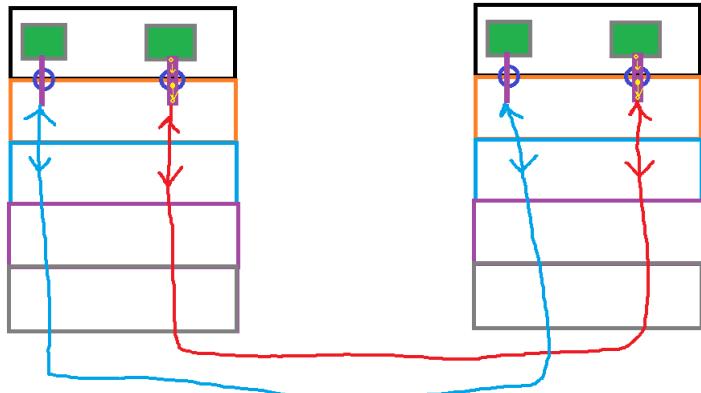
Per creare una socket si utilizza la funzione:

```
#include <sys/types.h>
#include <sys/socket.h>

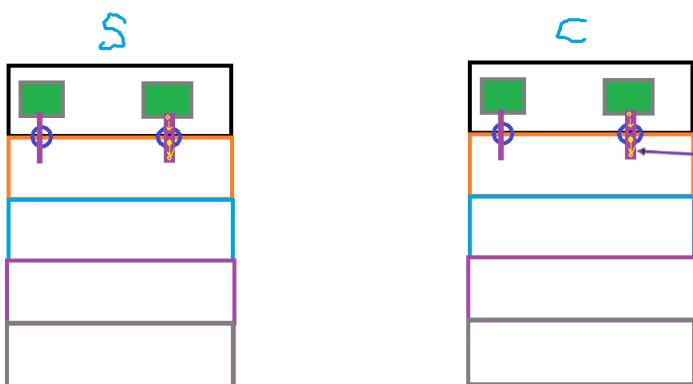
int socket(int domain, int type, int protocol);
```

man 7 ip
man 2 socket
man 7 socket

E' la prima funzione che sia il client che il server chiamano. Perché?



La socket è quel ponticino viola (fa da interfaccia) tra il livello applicativo e quello di trasporto. Attraverso questo ponte si inviano i dati al livello di trasporto (ci penserà poi lui e a seguire i successivi livelli inferiori a trasportarlo al destinatario) e sempre attraverso questo ponte si possono ricevere dati.



Tramite la funzione:

socket(domain,type,protocol)

abbiamo creato quel ponticino viola.

Per esempio:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h> /* superset of previous */

1 tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
2 udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

Quindi:

- **Domain**: può essere **AF_LOCAL** per un socket tra processi locali, o **AF_INET** per un socket tra processi che appartengono a macchine diverse utilizzando IPv4.
- **Type**: specifica il tipo di trasporto che può essere **SOCK_DGRAM** per usare un protocollo UDP, mentre **SOCK_STREAM** per un trasporto affidabile con TCP.

Quindi esistono fondamentalmente due tipi di socket: **datagram socket** e **stream socket** (anche se ne esiste un terzo, *raw socket*). La funzione ritorna un **descrittore di socket**, ossia un identificativo che permetta a chi ha creato il canale di identificarlo semmai ne dovesse creare altri. Se qualcosa è andato storto ritorna **-1** ed è possibile stampare il messaggio di errore tramite la funzione **strerror()** che vanno a vedere la variabile globale **errno** com'è stata settata dopo l'errore (a seconda del valore positivo assunto, significherà un ben determinato errore).

Quindi dove siamo arrivati?

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
    // 1) CREAZIONE SOCKET
    int id_socket = socket(AF_INET, SOCK_STREAM, 0);

    ...
    return 0;
}
```

SERVER

ASSEGNAZIONE UN INDIRIZZO A UNA SOCKET: bind()

Abbiamo creato il ponte viola, quindi la struttura ma adesso dobbiamo anche dargli un indirizzo, ossia dire attraverso quale SAP effettua il raccordo tra i due livelli. Sappiamo che una socket è identificata da un indirizzo IP:porta

Noi useremo indirizzo IPv4 ed esiste una struttura apposta già pronta chiamata:

```
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr sin_addr;
};

/* Internet address */
struct in_addr {
    uint32_t s_addr;
};
```

La *sa_family_t* vuole il domain visto prima, noi sceglieremo appunto AF_INET dato che stiamo trattando IPv4.

La *in_port_t* vuole il numero di porta (la SAP). Lo *in_addr* vuole un indirizzo a 32 bit (uint32_t).

L'indirizzo IP deve essere però espresso in **network order**, ossia il formato deve essere *big-endian*, ossia:

Il numero 0000-0010-1111-0111 viene memorizzato così:

indirizzo A	0000
Indirizzo A+1	0010
Indirizzo A+2	1111
Indirizzo A+3	0111

Ogni calcolatore/host però può lavorare non con big-endian ma con little-endian che prevede la memorizzazione in ordine inverso. Quando lavoriamo sul nostro host (magari per fare calcoli o altro) dobbiamo rispettare il suo formato, ma quando inviamo dati in rete dobbiamo rispettare il nework order. Per questo nel file *<arpa/inet.h>* esistono 4 funzioni:

da little-endian a big-endian
e viceversa

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

La prima prende un hostlong (ossia 32 bit) memorizzati nel formato dell'host (big o little) e ritorna i 32 bit in formato big-endian per il network. La terza effettua il processo inverso, dato un indirizzo sicuramente big-endian perché arrivato dalla rete, lo trasformiamo nel formato dell'host. Servono per gli indirizzi IP. La seconda e la quarta fanno la stessa cosa ma a 16bit. Servono per le porte.

Esempio:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
    const char* src = "192.222.255.3";
    struct in_addr* indirizzolP;

    //in c i puntatori si allocano così (equivale a una new)
    indirizzolP = malloc(sizeof(struct in_addr));

    //trasformo da notazione a punto a notazione decimale
    int a = inet_nton(AF_INET, src, &(indirizzolP->s_addr));
    /*adesso ho che s_addr= 67100352 che equivale a 3.255.222.192 (little-endian)
     *perché il mio calcolatore lo memorizza così
    */

    //prima di mandarlo in rete lo devo trasformare nel formato big-endian
    uint32_t indIP_big_endian = htonl(indirizzolP->s_addr);
    /*adesso ho indIP_big_endian =3235839747 che equivale a 192.222.255.3 (big-endian)
```

La prima funzione che ho usato serve per convertire una stringa dalla notazione a punto (detta di presentazione) dell'indirizzo IP a quella decimale.

```
int inet_nton(int af, const char* src, void* dst);
```

- **af**: famiglia (`AF_INET`)
- **src**: stringa del tipo "ddd.ddd.ddd.ddd"
- **dst**: puntatore a un'istanza di struttura `in_addr`

conversione indirizzolP da notazione a punto a decimale

```
const char* inet_ntop(int af, const void* src,
                      char* dst, socklen_t size);
```

- **af**: famiglia (`AF_INET`)
- **src**: puntatore a un'istanza di struttura `in_addr`
- **dst**: puntatore a un buffer di caratteri lungo `size`
- **size**: deve valere almeno `INET_ADDRSTRLEN`

conversione indirizzolP da decimale a notazione a punto

NB: la `inet_nton` inserisce i byte dentro l'area di memoria `src` secondo il formato big endian

La funzione `bind()` serve a collegare un blocco `sockaddr_in` a una socket:

```
int bind(int sockfd, const struct sockaddr *addr,
        socklen_t addrlen);
```

[man 2 bind](#)

- **sockfd**: descrittore del socket
- **addr**: puntatore a struttura di tipo `sockaddr`
N.B.: usiamo `sockaddr_in`, quindi occorre un cast del puntatore
- **addrlen**: dimensione di `addr`
- La primitiva `bind()` restituisce 0 se ha successo, -1 se si verifica un errore

Ritorna 0 se ha successo,
-1 se non ha avuto
successo (in questo caso
aggiorna sempre `errno`).

```
ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
```

Dove siamo arrivati?

```
int main()
```

// 1) CREAZIONE SOCKET: socket()

```
int id_socket = socket(AF_INET, SOCK_STREAM, 0);
```

//2) INIZIALIZZAZIONE INDIRIZZO IP e PORTA SOCKET: bind()

```
//creo blocco
```

```
struct sockaddr_in* indirizzoSocket;
```

```
//creo area
```

```
indirizzoSocket = malloc(sizeof(struct sockaddr_in));
```

```
//inizializzo
```

```
indirizzoSocket->sin_family = AF_INET;
```

```
indirizzoSocket->sin_port = htons(4242);
```

```
inet_pton(AF_INET, "192.168.4.5", &(indirizzoSocket->sin_addr));
```

```
//collego indirizzo alla socket
```

```
int esitoCollegamento = bind(id_socket, (struct sockaddr*)indirizzoSocket, sizeof(struct sockaddr_in));
```

```
return 0;
```

```
}
```

NB1: la bind ha bisogno come secondo argomento di un sockaddr_in, per questo creiamo tale blocco prima e lo settiamo.

NB2: la bind non accetta come argomento sockaddr_in ma un suo simili sockaddr. Pertanto noi eseguiamo una conversione esplicita.

METTERSI IN ATTESA SI RICHIESTE: listen()



Ora che abbiamo creato il ponte, lo abbiamo posizionato in un punto preciso dandogli un indirizzo, non ci resta che rendere aperta la fila per tutti i richiedenti della connessione

```
int listen(int sockfd, int backlog);
```

man 2 listen

- **sockfd**: descrittore del socket
- **backlog**: dimensione della coda, cioè quante richieste dai client possono essere in attesa di essere gestite
- La funzione restituisce 0 se ha successo, -1 su errore

Quindi la listen *non* si mette in ascolto e quando arriva una richiesta l'accetta, ma permette di creare una coda di connessioni.

Quindi se arriva una richiesta da parte di un client e la coda è piena, questa connessione viene rifiutata. Possono avere una coda solo socket SOCK_STREAM, quindi non si avverte il client che la sua richiesta è stata scartata, perché lavorando col TCP ritenterebbe una nuova connessione subito dopo.

Dove siamo arrivati?

```
int main()
{
    // 1) CREAZIONE SOCKET: socket()
    int id_socket = socket(AF_INET, SOCK_STREAM, 0);

    // 2) INIZIALIZZAZIONE INDIRIZZO IP e PORTA SOCKET: bind()
    //creo blocco
    struct sockaddr_in* indirizzoSocket;
    //creo area
    indirizzoSocket = malloc(sizeof(struct sockaddr_in));
    //inizializzo
    indirizzoSocket->sin_family = AF_INET;
    indirizzoSocket->sin_port = htons(4242);
    inet_nton(AF_INET, "192.168.4.5", &(indirizzoSocket->sin_addr));

    //collego indirizzo alla socket
    int esitoCollegamento = bind(id_socket, (struct sockaddr*)indirizzoSocket, sizeof(struct sockaddr_in));

    // 3) METTERSI IN ASCOLTO: listen()
    int esito = listen(id_socket, 10); //posso gestire al massimo 10 richieste

    return 0;
}
```

ACCETTARE UNA RICHIESTA: accept()

Arrivano le richieste e grazie alla listen() abbiamo permesso la creazione di una coda. Adesso è il momento di aprire il tornello e far passare qualcuno.

```
int accept(int sockfd, struct sockaddr *addr,
           socklen_t *addrlen);
```

man 2 accept

- **sockfd**: descrittore del socket
- **addr**: puntatore a una struttura (vuota) di tipo **struct sockaddr** dove viene salvato l'indirizzo del client
- **addrlen**: dimensione di **addr**

Nota che la primitiva è **bloccante**, ossia una volta chiamata se la coda risulta vuota allora ci si blocca alla riga di codice della accept() e il calcolatore multiprocesso potrebbe nel frattempo fare altro e ritornare alla riga accept() solo quando arriva una richiesta.

NB: la accept() ritorna la socket da cui leggere!!!!

Dove siamo arrivati?

```
// 1) CREAZIONE SOCKET: socket()
int id_socket = socket(AF_INET, SOCK_STREAM, 0);

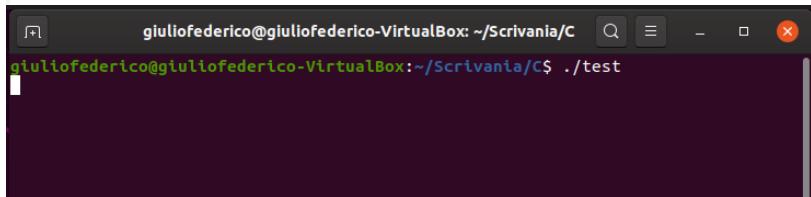
// 2) INIZIALIZZAZIONE INDIRIZZO IP e PORTA SOCKET: bind()
//creo blocco
struct sockaddr_in* indirizzoSocket;
//creo area
indirizzoSocket = malloc(sizeof(struct sockaddr_in));
//inizializzo
indirizzoSocket->sin_family = AF_INET;
indirizzoSocket->sin_port = htons(4242);
inet_nton(AF_INET, "192.168.4.5", &(indirizzoSocket->sin_addr));

//collego indirizzo alla socket
int esitoCollegamento = bind(id_socket, (struct sockaddr*)indirizzoSocket, sizeof(struct sockaddr_in));

// 3) METTERSI IN ASCOLTO: listen()
int esito = listen(id_socket, 10); //posso gestire al massimo 10 richieste

// 4) POSSO ACCETTARE UNA RICHIESTA (se esiste): accept()
//creare un area di memoria dove salvare l'indirizzo socket del client
struct sockaddr_in* indirizzoSocketClient;
indirizzoSocketClient = malloc(sizeof(struct sockaddr_in));
//creo una zona di memoria per un puntatore alla lunghezza dell'area di sopra
int* indirizzoLen;
indirizzoLen = malloc(sizeof(int));

//provo a vedere di accettarne una
int id_socketClient = accept(id_socket, (struct sockaddr*)indirizzoSocketClient, indirizzoLen);
return 0;
}
```



Se già dovessimo avviare il programmino, vedremmo metterci in attesa (quindi bloccati) perché non c'è nessuno in coda.

CLIENT

Per creare la socket si fa la stessa identica cosa fatta col server.

INVIA UNA RICHIESTA A UN SOCKET REMOTO: connect()

Anche il client ha una sua socket. Una volta creata bisogna attraverso questa inviare una richiesta alla socket del server di interesse:

```
int connect(int sockfd, const struct sockaddr* addr,
           socklen_t addrlen);
```

man 2 connect

- **sockfd**: descrittore del socket locale
- **addr**: puntatore alla struttura contenente l'indirizzo del socket remoto
- **addrlen**: dimensione di **addr**
- La primitiva restituisce 0 se ha successo, -1 se si verifica un errore

Anche questa primitiva è bloccante.

Dove siamo arrivati?

```

1 //CLIENT
2 #include <arpa/inet.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8
9 int main(){
10
11     // 1) CREAZIONE SOCKET: socket()
12
13     int id_socket = socket(AF_INET, SOCK_STREAM, 0);
14
15     // 2) INVIO DI RICHIESTA A UN SERVER REMOTO: connect()
16
17     //creazione blocco che contiene info sul socket remoto da contattare
18     struct sockaddr_in* indirizzoServerSocket;
19     indirizzoServerSocket = malloc(sizeof(struct sockaddr_in));
20     indirizzoServerSocket->sin_family = AF_INET;
21     indirizzoServerSocket->sin_port= htons(4242);
22     inet_pton(AF_INET, "10.0.2.15", &(indirizzoServerSocket->sin_addr));
23
24     //invio richiesta
25     int esito_richiesta = connect(id_socket, (struct sockaddr*)indirizzoServerSocket, sizeof(struct sockaddr_in));
26
27
28
29
30
31     return 0;
32
33 }
```

Possiamo già eseguire una prima prova. Ovviamente server.c e client.c sono due file diversi che verranno fatti partire da diversi terminali.

Ovviamente ho scelto come indirizzoIP del server uno tra quelli disponibili delle mie schede di rete (ip addr show per vedere quali).

Quello che vedremo è un server che si mette in attesa e un client che appena fa la richiesta si mette in attesa di ricezione (che avviene subito) e entrambi i programmi terminano così senza fare altro.

CLIENT & SERVER

INVIARE I DATI: send()

Valida sia per il client che per il server viene usata per inviare dati **dopo** che due socket sono stati connessi tra loro (infatti vedremo che come argomenti non prende nulla del socket destinatario ma infila solo nell'imbuto del socket i dati sapendo che è collegato già col destinatario).

```
ssize_t send(int sockfd, const void* buf, size_t len,
             int flags);
```

man 2 send

- **sockfd**: descrittore del socket
- **buf**: puntatore al buffer contenente il messaggio da inviare
- **len**: dimensione del messaggio (in byte)
- **flags**: per settare le opzioni (per adesso mettiamolo a 0)
- La funzione restituisce il **numero di byte inviati**, -1 su errore

Anche questa primitiva è bloccante, e lo è fino a quando non ha terminato di inviare il messaggio.

Dove siamo arrivati?

```
int main(){
    // 1) CREAZIONE SOCKET: socket()
    int id_socket = socket(AF_INET, SOCK_STREAM, 0);

    //2) INVIO DI RICHIESTA A UN SERVER REMOTO: connect()
    //creazione blocco che contiene info sul socket remoto da contattare
    struct sockaddr_in* indirizzoServerSocket;
    indirizzoServerSocket = malloc(sizeof(struct sockaddr_in));
    indirizzoServerSocket->sin_family = AF_INET;
    indirizzoServerSocket->sin_port= htons(242);
    inet_pton(AF_INET, "10.0.2.15", &(indirizzoServerSocket->sin_addr));

    //invio richiesta
    int esito_richiesta = connect(id_socket, (struct sockaddr*)indirizzoServerSocket, sizeof(struct sockaddr_in));

    //3) INVIO DATI AL SERVER
    //creo messaggio
    char buffer[1024];
    strcpy(buffer, "Ciao Universo");
    int len = strlen(buffer);

    //invio
    int esito = send(id_socket, (void*)buffer, len, 0);

    return 0;
}
```

RICEVERE I DATI: recv()

Valida sia per il client che per il server viene usata per ricevere dati **dopo** che due socket sono stati connessi:

```
ssize_t recv(int sockfd, const void* buf, size_t len,
            int flags);
```

man 2 recv

- **sockfd**: descrittore del socket
- **buf**: puntatore al buffer in cui salvare il messaggio
- **len**: dimensione in byte del messaggio desiderato
- **flags**: per settare le opzioni
- La funzione restituisce il **numero di byte ricevuti**, -1 su errore, 0 se il socket remoto si è chiuso (vedi più avanti)

Anche questa è bloccante fino a quando non riceve tutti i byte.

NB: sockfd è la socket ricevuta dalla accept()!!!!

//5) ATTENDO DI RICEVERE I DATI: recv()

```
//ricevo
esito_ricezione = recv(id_socketClient, (void*)buffer, 13,0);

if(esito_ricezione < 0)
    perror("errore nella ricezione dei dati");
```

Dove id_socketClient è quella ricevuta dalla accept().

CHIUSURA CONNESSIONE: close()

Valida sia per il

- **Chiude un socket**

- Non può più essere usato per inviare o ricevere dati

```
#include <unistd.h>

int close(int fd);
```

man 2 close

- **fd:** descrittore del socket
- La funzione restituisce 0 se ha successo, -1 su errore
- L'host remoto riceverà 0 dalla **recv()**

NB: fd è il proprio socket. La chiusura la fa di solito il client perché in genere il server è sempre attivo e saprà che il client ha chiuso quando nel tentativo di leggere con recv() riceverà 0.

Server.c & client.c

Server.c

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

int main(){
    int id_socket;
    struct sockaddr_in indirizzoSocket;
    uint32_t a;
    int esitoCollegamento;
    int esito;
    struct sockaddr_in indirizzoSocketClient;
    int len;
    int id_socketClient;
    char buffer[1024];
    int esito_ricezione;
    // 1) CREAZIONE SOCKET: socket()
    id_socket = socket(AF_INET, SOCK_STREAM, 0);
    // 2) INIZIALIZZAZIONE INDIRIZZO IP e PORTA SOCKET: bind()
    // pulisco (resettando) l'area dove metterò l'indirizzo
    memset(&indirizzoSocket, 0, sizeof(indirizzoSocket));
    //inizializzo
    indirizzoSocket.sin_family = AF_INET;
    indirizzoSocket.sin_port = htons(4200);
    inet_pton(AF_INET, "127.0.0.1", &(indirizzoSocket.sin_addr));
    esitoCollegamento = bind(id_socket, (struct sockaddr*)&indirizzoSocket, sizeof(indirizzoSocket));
}
```

```

if(esitoCollegamento<0)
    perror("errore nel collegamento");

//3) METTERSI IN ASCOLTO: listen()

esito = listen(id_socket, 10); //posso gestire al massimo 10 richieste
if(esito<0)
    printf("errore nel listen()");

//4) POSSO ACCETTARE UNA RICHIESTA (se esiste): accept()

len = sizeof(indirizzoSocketClient);
//creo una zona di memoria per un puntatore alla lunghezza dell'area di sopra

//provo a vedere di accettarne una
id_socketClient = accept(id_socket, (struct sockaddr*)&indirizzoSocketClient, &len);
if(id_socketClient<0)
    printf("errore nell'accept()");

//5) ATTENDO DI RICEVERE I DATI: recv()

//ricevo
esito_ricezione = recv(id_socketClient, (void*)buffer, 13,0);

if(esito_ricezione < 0)
    perror("errore nella ricezione dei dati");
else if( esito ==0 )
    printf("il client ha terminato la connessione");

//6) CHIUDO

close(id_socket);
return 0;

```

(nb: invece di esito_ricezione ho scritto esito.)

Client.c

```

#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
    int id_socket;
    struct sockaddr_in* indirizzoServerSocket;
    int esito_richiesta;
    char buffer[1024];
    int len;
    int esito;
    // 1) CREAZIONE SOCKET: socket()
    printf("creo socket\n");
    id_socket = socket(AF_INET, SOCK_STREAM, 0);

    //2) INVIO DI RICHIESTA A UN SERVER REMOTO: connect()
    printf("invio richiesta al server...\n");
    //creazione blocco che contiene info sul socket remoto da contattare
    indirizzoServerSocket = malloc(sizeof(struct sockaddr_in));
    indirizzoServerSocket->sin_family = AF_INET;
    indirizzoServerSocket->sin_port= htons(4200);

    inet_pton(AF_INET, "127.0.0.1", &(indirizzoServerSocket->sin_addr));

    // indirizzoServerSocket->sin_addr.s_addr = htonl(INADDR_LOOPBACK);

    //invio richiesta
    esito_richiesta = connect(id_socket, (struct sockaddr*)indirizzoServerSocket, sizeof(struct sockaddr_in));
    if(esito_richiesta<0)
        printf("errore nel connect()");

```

```

//3) INVIO DATI AL SERVER
printf("Richiesta accettata. Invio dati al server...\n");
//creo messaggio
strcpy(buffer, "Ciao Universo");
len = strlen(buffer);

//invio
esito = send(id_socket, (void*)buffer, len, 0);

if(esito<0)
    perror("errore nell'invio:");

//4) CHIUDO CONNESSIONE: close()
close(id_socket);

return 0;
}

```

NOTE

Consiglio vivamente, dopo ogni chiamata di primitiva, di effettuare un controllo degli errori tramite sterror("mex") per capire cosa è successo. Spesso le funzioni in cui si hanno più problemi sono la bind() e la recv(). Per la bind ci si potrebbe confondere nel dimenticare che inet_pton(..) fornisce già il formato big-endian e quindi sarebbe errato fare un htonl sull'indirizzo che ha convertito da notazione a punto a decimale; inoltre spesso le porte sono già in uso da altri processi o anche gli stessi indirizzi. Per verificare quali tra questi processi utilizzano una determinata porta basta scrivere:

lsof -wni tcp:<numeroPorta>

Se spunta qualche riga significa che alcuni processi identificati con PID la stanno usando e questo potrebbe causare l'errore nel bind ***address already in use***. Pertanto server.c non potrà partire se prima non "uccidiamo" i processi che stanno utilizzando quella porta, e questo si fa con:

kill -9 PID

dove PID è appunto l'identificativo del processo ottenuto prima. In alternativa si potrebbe cambiare semplicemente porta.

Se si volesse eliminare il processo non dal suo id ma dal suo nome (il che è conveniente così da eliminare tanti processi uguali ma con pid diversi) basterà fare:

kill -9 \$(pgrep <nomeprocesso>)

Con la funzione recv() ci si dimentica spesso che il socket da cui leggere è quello ottenuto dalla accept().

Come indirizzoIP del server si possono scegliere uno degli inet IPv4 delle interfacce di rete disponibili. Per vedere quali basta fare:

ip addr show

In genere si usa INADDR_LOOPBACK (che equivale al localhost) ossia 127.0.0.1, oppure un indirizzo speciale noto come INADDR_ANY che dice "qualsiasi indirizzo di interfacce disponibili mi va bene".

E' bene notare che le socket TCP sono **orientate alla connessione**, e quindi vuol dire che si stabilisce un canale nel quale la comunicazione è full-duplex perché per garantire tutti i servizi del TCP c'è bisogno che il server comunichi. Questo vuol dire che con il socket ottenuto dopo l'accept possiamo anche usarlo per inviare dati al client:

client.c

```
//3) INVIO DATI AL SERVER
//creo messaggio
strcpy(buffer, "Ciao Universo");
len = strlen(buffer);

//invio
esito = send(id_socket, (void*)buffer, len, 0);

if(esito<0)
    perror("\nerrore nell'invio:");

//ricevere risposta dal server
int esito_risp = recv(id_socket, (void*)buffer, len, 0);
if(esito_risp<0)
    perror("\nerrore risposta:");
else if (esito_risp ==0)
    printf("\n il server ha chiuso la connessione");
else {
    for(int i=0; i<esito_risp; i++)
        printf("%c",buffer[i]);
}
```

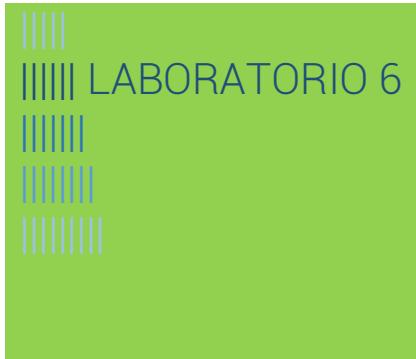
server.c

```
int new1 = accept(id_socket1, (struct sockaddr*)&indirizzoSocket1, &len1);
if(new1<0)
    perror("socket1: errore nell'accettare la richiesta");
printf("socket %d attivato\n", id_socket1);

esito_ricezione = recv(new1, (void*)buffer, 13, 0);

if(esito_ricezione < 0)
    perror("errore nella ricezione dei dati\n");
else if(esito_ricezione == 0)
    printf("il client ha terminato la connessione\n");
else {
    for(int i=0; i<esito_ricezione; i++)
        printf("%c",buffer[i]);
}

//invio risposta al client
char risp[2]={0,'k'};
int esito_risp = send(new1, (void*)risp, sizeof(risp), 0);
close(new1);
```



PROGRAMMAZIONE CONCORRENTE e MULTIPLEXING DELL'IO

Esistono due tipologie di server:

- 1) **Server iterativo:** serve una richiesta alla volta (`accept()`) e la elabora, mettendo in coda quelle che arrivano.
- 2) **Server concorrente:** serve più richieste alla volta (`accept()`) e per ognuna crea un processo (per chi ha fatto java stiamo parlando di thread) chiamato processo figlio che la elabora.

SERVER CONCORRENTE

Un server concorrente è capace fondamentalmente di fare le seguenti cose:

- 1) Se arriva una richiesta, lui si sdoppia. Lo sdoppiamento consiste in:
 - a. Creare un altro processo il cui sorgente è la copia esatta del codice sorgente del server.c (non viene creato un altro file però)
 - b. Questo nuovo processo figlio avrà un PID tutto suo.
 - c. L'esecuzione di questo processo figlio inizierà dal punto esatto in cui è stato creato.
 - d. Questo nuovo processo si prenderà carico della richiesta per poi morire (`close()` e `exit()`)
- 2) Creato un nuovo processo figlio che nel frattempo elabora da se la richiesta, il server a loop ritorna libero per accettare una nuova richiesta (`accept()`). Se ne arriva un'altra si riparte dal punto 1)

Quindi un server concorrente si occupa solo di creare processi figli a cui delegare l'elaborazione della richiesta. Ovviamente si potrebbe pensare che i server concorrenti siano migliori di quelli iterativi, e molto spesso è così, ma non si deve pensare che tutto questo sia gratis. Non è fisicamente possibile eseguire in contemporanea 1000 processi figli (almeno con un solo processore). Al tempo $T=t$ è in esecuzione sempre e solo un processo. Quindi in

una esecuzione concorrente, il processore viene ceduto a un processo X e può essere revocato nel mentre per consegnarlo ad un altro e così via per poi ritornare a quello X dopo un tot di tempo. Quindi se in un server iterativo quel processo (che sarebbe il server padre stesso) avrebbe impiegato un tot di tempo, adesso deve essere consapevole del fatto che ci sono altri 999 processi a cui dare il processore e quindi potrebbe terminare dopo un tempo molto maggiore che nel caso iterativo.

SDOPPIARE UN PROCESSO: fork()

```
#include <unistd.h>
pid_t fork(void);
```

[man 2 fork](#)

- La primitiva **fork()** **sdoppia il processo**

- Nel processo padre, restituisce il *process identifier (PID)* del processo figlio
- Nel processo figlio, restituisce 0

Il tipo **pid_t** è un intero con segno. Nella GNU C Library è un **int**.

Nota bene: è come se stessi sdoppiando server.c ma nel server.c del padre si avrà che il figlio avrà un PID reale (tipo 5667) mentre nel server.c del figlio lui vedrà se stesso con PID=0 (infatti ogni processo vede se stesso con PID=0).

Come ho modificato il server.c per renderlo concorrente?

```
//3) METTERSI IN ASCOLTO: listen()
printf("Il server è in ascolto...\n");
esito = listen(id_socket, 10); //posso gestire al massimo 10 richieste
if(esito<0)
    printf("errore nel listen()\n");

//4) POSSO ACCETTARE UNA RICHIESTA (se esiste): accept()
len = sizeof(indirizzoSocketClient);
//creo una zona di memoria per un puntatore alla lunghezza dell'area di sopra
while(1){
    //provo a vedere di accettarne una
    printf("Il server è in attesa di una nuova richiesta.\n");
    id_socketClient = accept(id_socket, (struct sockaddr*)&indirizzoSocketClient, &len);
    if(id_socketClient<0)
        printf("errore nell'accept()\n");

    //creo processo figlio
    pid = fork(); //qui è la modifica rispetto al server.c originale

    if(pid == -1)
        perror("errore nella creazione del processo figlio:");
    //sono nel processo figlio
    if(pid == 0){
        printf("Richiesta accettata. Creazione processo figlio.\n");
        close(id_socket);
    }
    //5) ATTENDO DI RICEVERE I DATI: recv()
    //ricevo
    esito_ricezione = recv(id_socketClient, (void*)buffer, 13,0);

    if(esito_ricezione < 0)
        perror("errore nella ricezione dei dati\n");
    else if( esito_ricezione == 0)
        printf("il client ha terminato la connessione\n");
    else {
        for(int i=0; i<13; i++)
            printf("%c",buffer[i]);
    }
    close(id_socketClient);
    exit(0);
}

//6) CHIUDO
close(id_socketClient);

}
return 0;
```

Quando creo il processo figlio questo avrà un suo PID, ma all'interno del server.c avrà un identificativo 0 che lo differenzierà dal padre. In quell'esatto punto ho creato due processi uguali; quindi è come se avessi due file server.c con lo stesso codice che stanno eseguendosi parallelamente. Però siccome il sorgente del server.c è unico devo differenziare le istruzioni che deve eseguire il padre (che si limita solo a chiudere il socket client dopo aver creato il figlio) e quelle che deve eseguire il figlio (con PID=0) (che chiude il socket padre, recupera i dati dal client e se non ci sono errori chiude e termina (exit(0))). In quest'ultimo punto è come se il server.c copia non esistesse più.

SOCKET NON BLOCCANTI

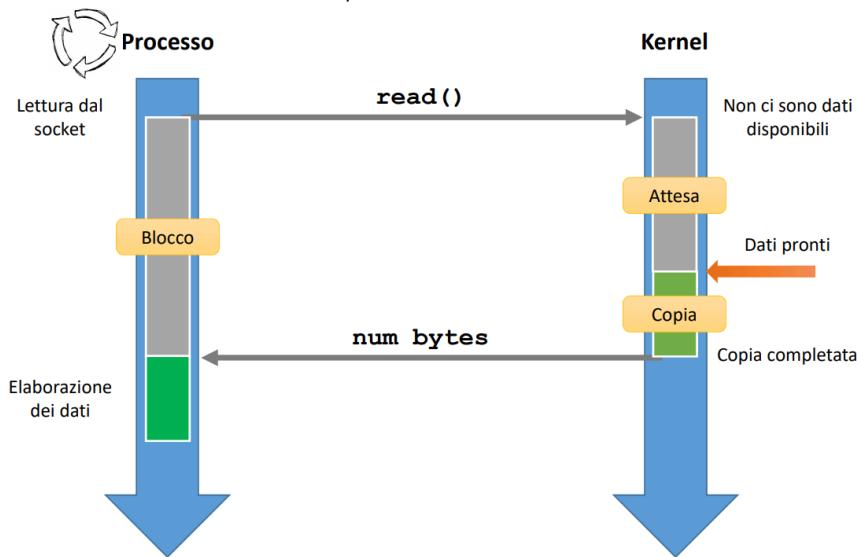
SOCKET NON BLOCCANTI

Per i socket bloccanti abbiamo visto che le primitive:

- `connect()` blocca il processo finché il socket non è connesso
- `accept()` blocca il processo finché non arriva una richiesta di connessione
- `send()` blocca il processo finché tutto il messaggio non è stato inviato (il buffer di invio potrebbe essere pieno)
- `recv()` blocca il processo finché non ci sono dati disponibili o finché tutto il messaggio richiesto non è disponibile (flag `MSG_WAITALL`)

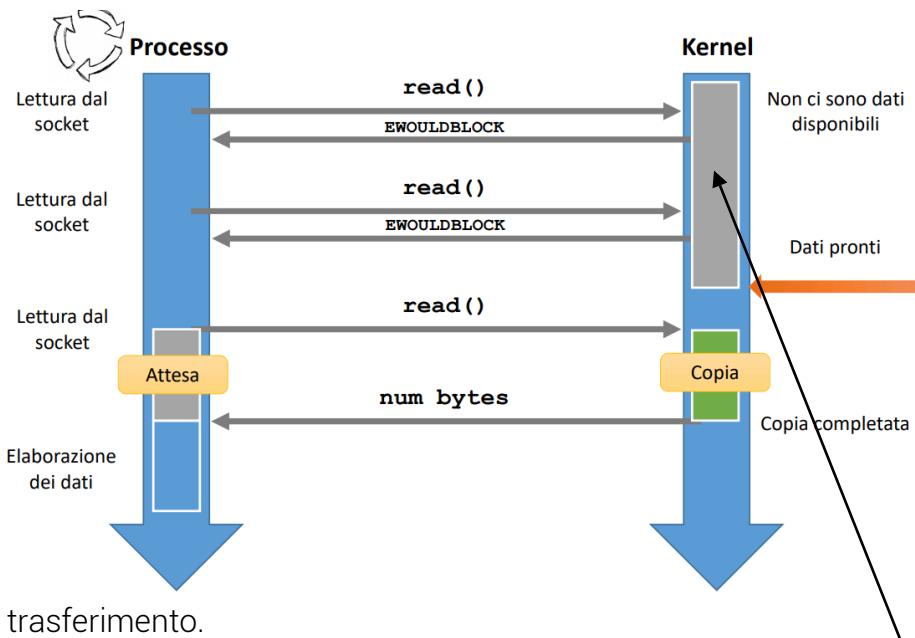
bloccano il processo.

Questo cosa vuol dire in pratica?



Supponiamo che `read()` sia una primitiva come `recv()` che vuole leggere. Se il socket è bloccante significa che quando viene chiamata `read()` ci si blocca fino a quando non esistono dati tali per cui la `read()` è stata invocata. Questo può significare che se il server chiama la `read()` prima che il client spedisca i dati tramite la `send()` allora si metterà in attesa.

Per i socket non bloccanti questa attesa non esiste. Non fraintendete però. Per socket non bloccanti si intende che se quando si chiama una funzione non esistono subito i dati allora si salta all'istruzione successiva, ma qualora dovessero esistere dati, ci si blocca comunque per l'attesa di trasferire i dati.



trasferimento.

Un socket non bloccante si crea così:

```
socket(AF_INET, SOCK_STREAM|SOCK_NONBLOCK, 0);
```

A livello di codice visto fin'ora non cambia nulla ma bisogna prevedere che l'istruzione successiva possa partire senza che la precedente abbia effettivamente svolto il suo ruolo. Per questo nei socket non bloccanti hanno molta importanza i tipi di errore che avvengono dopo ogni esecuzione di primitiva:

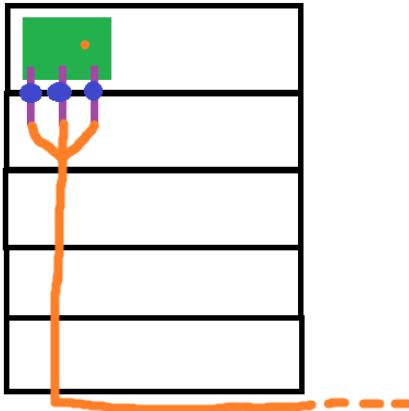
- **connect()**, se non può connettersi, restituisce -1 e imposta `errno` a `EINPROGRESS`
- **accept()**, se non ci sono richieste, restituisce -1 e imposta `errno` a `EWOULDBLOCK`
- **send()**, se non può inviare tutto il messaggio (il buffer è pieno), restituisce -1 e imposta `errno` a `EWOULDBLOCK`
- **recv()**, se non ci sono messaggi, restituisce -1 e imposta `errno` a `EWOULDBLOCK`

Sempre in riferimento alla `read()` se questa viene chiamata ogni qual volta non ci sono i dati allora termina subito (magari potremmo richiamarla a loop fino a quando non ci sono i dati, come nell'esempio a fianco). Se invece alla chiamata di `read()` becchiamo la presenza dei dati allora attendiamo il

Questi errori si generano sulla variabile `errno` quando una volta invocata la primitiva ci si ritrova qua

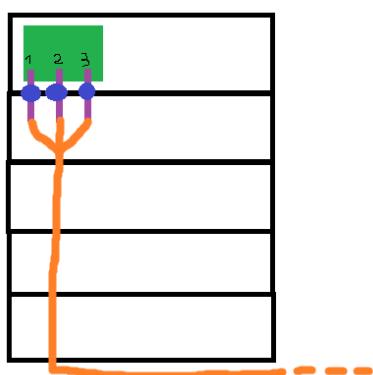
GESTIONE DI PIU' SOCKET BLOCCANTI: select()

Una socket altro non è che un ponte che ha un indirizzoIP e una porta (indirizzoIP:porta). Se un processo può disporre di più socket vuole dire che può disporre di più punti per raggiungerlo. Può avere differenti indirizziIP oppure (come di solito si fa) diverse porte.



In questo caso il livello di trasporto è in grado di eseguire il multiplexing di ciò che gli arriva propagando i flussi di dati nelle corrette socket che giungono allo stesso processo applicativo.

Qual è il problema se tutte quelle socket sono bloccanti?



```
int new1 = accept(socket1,...)
int new2 = accept(socket2,...)
int new3= accept(socket3,...)
```

Se l'ordine delle accept su ogni socket è questo allora la accept() della socket2 verrà servita solo quando la accept() della socket1 ha qualche dato e quindi smette di bloccarsi. Allo stesso modo la socket3 deve aspettare non solo che alla socket1 arrivi qualcosa ma anche che alla socket2 arrivi qualcosa, quando magari lei ha già qualcosa da ritirare da molto prima.

Per risolvere questo problema si potrebbero usare le socket non bloccanti e fare un controllo su errno. Però di solito si usa procedere in questo modo.

- 1) Si crea una lista di socket il cui tipo è **fd_set**. Questa lista contiene tutte le socket da monitorare. Le funzioni per manipolare la lista sono:

```
/* Rimuovere un descrittore "fd" dall'insieme "set" */
void FD_CLR(int fd, fd_set* set);

/* Controllare se un descrittore "fd" è nell'insieme "set" */
int FD_ISSET(int fd, fd_set* set);

/* Aggiungere un descrittore "fd" all'insieme "set" */
void FD_SET(int fd, fd_set* set);

/* Svuotare l'insieme "set" */
void FD_ZERO(fd_set* set);
```

- 2) In una variabile **int** si tiene sempre il numero identificativo della socket più grande attualmente da monitorare.

Pertanto avremmo un numero n di socket da monitorare che sono state infilate dentro una lista perché voglio sapere quale tra quelle socket è:

- pronta per una lettura, ossia:

- C'è almeno un byte da leggere
 - Il socket è stato chiuso (e quindi si leggerà il byte 0)
 - C'è un errore (e quindi si leggerà -1)
- Pronta per la scrittura, ossia:
- c'è spazio nel buffer per scrivere
 - c'è un errore (la scrittura restituirà -1)

Chi si accorge di questi due eventi (possibilità di leggere o scrivere sul proprio socket) è la funzione **select()**:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set* readfds, fd_set* writefds,
           fd_set* exceptfds, struct timeval* timeout);
```

man 2 select

- **nfds**: numero del descrittore più alto tra quelli da controllare, +1
- **readfds/writefds**: lista di descrittori da controllare per la lettura/scrittura
- **exceptfds**: lista di descrittori da controllare per le eccezioni (non ci interessa)
- **timeout**: intervallo di timeout
- Restituisce il **numero di descrittori pronti** (-1 in caso di errore)
- **È bloccante**: si blocca finché uno dei descrittori controllati non è pronto, oppure finché non scade il timeout

Cos'è che fa?

Prevediamo intanto due liste diverse a seconda che le socket servano per scrivere o per leggere. La select prende al massimo queste due liste con dentro le socket da monitorare e si blocca fino a quando su queste socket non c'è possibilità di leggere o scrivere e quando questo accade si sblocca modificando la lista con le sole socket in cui si può leggere o scrivere:



Supponiamo che le prime due sono socket di lettura e le altre di

scrittura. Si chiama la select su questa lista e ci si sblocca. Se dopo ci si ritrova una volta sbloccati con una lista di 4 e 321 significa che è possibile leggere dalla socket 4 e scrivere nella socket 21.

```
#include <sys/socket.h>
#include <netinet/in.h>

struct timeval {
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};

• timeout = NULL
    • Attesa indefinita, fino a quando un descrittore è pronto
• timeout = { 10; 5; }
    • Attesa massima di 10 secondi e 5 microsecondi
• timeout = { 0; 0; }
    • Attesa nulla, controlla i descrittori ed esce immediatamente (polling)
```

Il **timeout** è una struttura da modificare come NULL se si attende all'infinito su quella riga di codice fino a quando non succede qualcosa; oppure tutta 0 per dire di non attendere nulla, ossia o c'è qualcosa o esci subito; infine si possono esprimere secondi e microsecondi di attesa massima.

Esempio:

```

int main(){
    int id_socket1, id_socket2;
    struct sockaddr_in indirizzoSocket1, indirizzoSocket2;
    int len1, len2;
    int esitoCollegamento1, esitoCollegamento2;
    int esitoListen1, esitoListen2;

    fd_set listaSocket;
    int numeroMassimoDiSocket;

    // 1) CREAZIONE DEI SOCKET: socket()
    //socket1
    id_socket1= socket(AF_INET, SOCK_STREAM, 0);
    printf("socket %i creato\n",id_socket1);
    //socket2
    id_socket2= socket(AF_INET, SOCK_STREAM, 0);
    printf("socket %i creato\n",id_socket2);

    numeroMassimoDiSocket = id_socket2;

    //2) INIZIALIZZAZIONE INDIRIZZO IP e PORTA DEI SOCKET: bind()
    //publico (resetando) l'area dove metterò l'indirizzo
    memset(&indirizzoSocket1,0,sizeof(indirizzoSocket1));
    memset(&indirizzoSocket2,0,sizeof(indirizzoSocket2));

    //inizializzo socket1
    indirizzoSocket1.sin_family = AF_INET;
    indirizzoSocket1.sin_port = htons(4200);
    indirizzoSocket1.sin_addr.s_addr = htonl(INADDR_ANY);
    esitoCollegamento1 = bind(id_socket1, (struct sockaddr*)&indirizzoSocket1, sizeof(indirizzoSocket1));
    len1 = sizeof(indirizzoSocket1);
    if(esitoCollegamento1<0)
        perror("errore nell'indirizzare la socket1");
    else
        printf("socket %i inizializzato\n",id_socket1);

    //inizializzo socket2
    indirizzoSocket2.sin_family = AF_INET;
    indirizzoSocket2.sin_port = htons(4201);

    indirizzoSocket2.sin_addr.s_addr = htonl(INADDR_ANY);
    esitoCollegamento2 = bind(id_socket2, (struct sockaddr*)&indirizzoSocket2, sizeof(indirizzoSocket2));
    len2 = sizeof(indirizzoSocket2);
    if(esitoCollegamento2<0)
        perror("errore nell'indirizzare la socket2");
    else
        printf("socket %i inizializzato\n",id_socket2);

    //3) METTERSI IN ASCOLTO: listen()
    //listen socket1
    esitoListen1 = listen(id_socket1, 10); //posso gestire al massimo 10 richieste
    if(esitoListen1<0)
        perror("errore nel listen():");
    else
        printf("socket %i in ascolto\n", id_socket1);

    //listen socket2
    esitoListen2 = listen(id_socket2, 5); //posso gestire al massimo 5 richieste
    if(esitoListen2<0)
        perror("errore nel listen():");
    else
        printf("socket %i in ascolto\n", id_socket2);

    while(1){
        printf("\nrestart\n");
        //azzero la lista
        FD_ZERO(&listaSocket);
        //aggiungo i due socket alla lista per monitorarli
        FD_SET(id_socket1, &listaSocket);
        FD_SET(id_socket2, &listaSocket);

        //uso select per modificare la lista lasciando i soli socket che hanno qualcosa
        select(numeroMassimoDiSocket+1,&listaSocket,NULL,NULL,NULL);

        //controllo quale socket ha qualcosa
        //se è la socket1
        if(FD_ISSET(id_socket1, &listaSocket)){
            char buffer[1024];
            int esito_ricezione;
            int new1 = accept(id_socket1, (struct sockaddr*)&indirizzoSocket1, &len1);
            if(new1<0)
                perror("socket1: errore nell'accettare la richiesta");
            printf("socket %d attivato\n", id_socket1);

            esito_ricezione = recv(new1, (void*)buffer, 13, 0);

            if(esito_ricezione < 0)
                perror("errore nella ricezione dei dati\n");
        }
    }
}

```

```

else if(esito_ricezione == 0)
    printf("il client ha terminato la connessione\n");
else {
    for(int i=0; i<esito_ricezione; i++)
        printf("%c",buffer[i]);
}

close(new1);

}

//se è la socket2
if(FD_ISSET(id_socket2, &listaSocket)){
    char buffer[1024];
    int esito_ricezione;
    int new2 = accept(id_socket2, (struct sockaddr*)&indirizzoSocket2, &len2);
    if(new2<0)
        perror("socket2: errore nell'accettare la richiesta");
    printf("socket %d attivato", id_socket2);

    esito_ricezione = recv(new2, (void*)buffer, 13, 0);

    if(esito_ricezione < 0)
        perror("errore nella ricezione dei dati\n");
    else if(esito_ricezione == 0)
        printf("il client ha terminato la connessione\n");
    else {
        for(int i=0; i<esito_ricezione; i++)
            printf("%c",buffer[i]);
    }

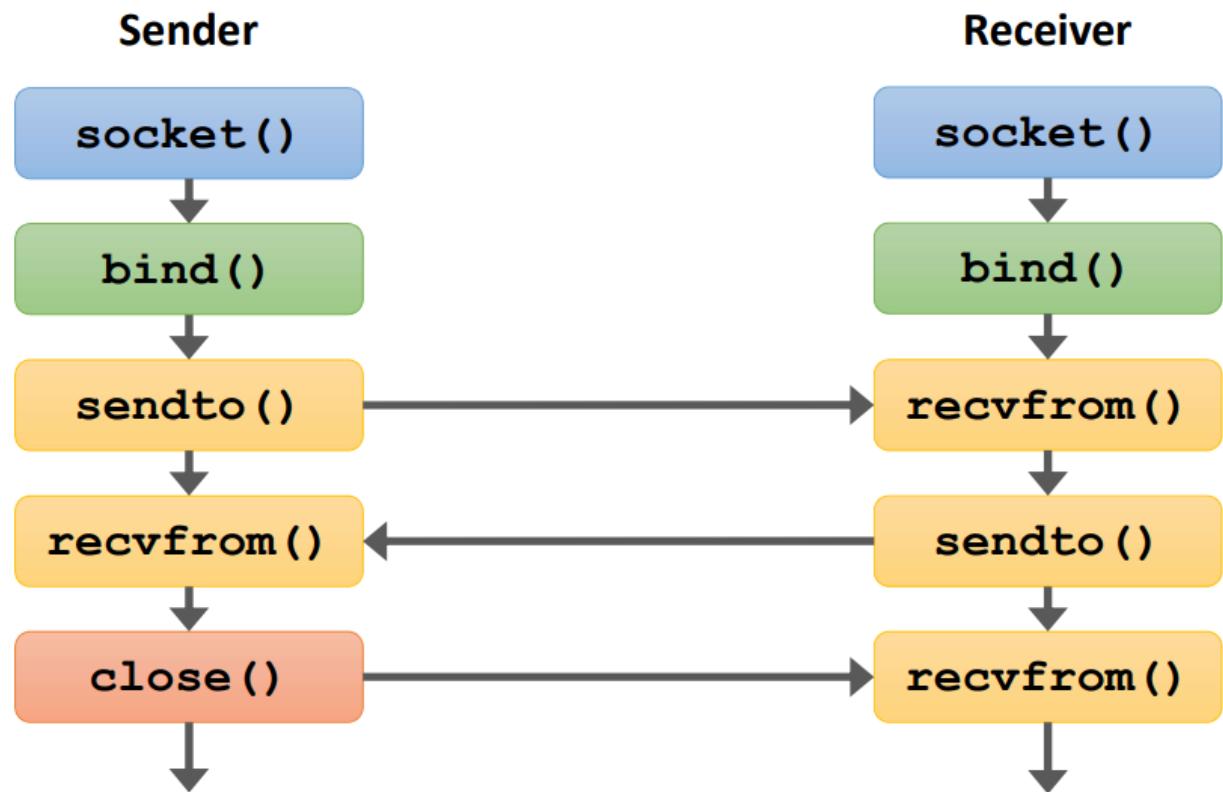
    close(new2);
}

}

return 0;
}

```

SOCKET UDP



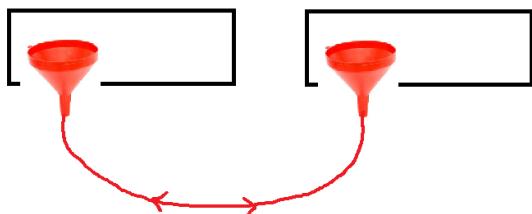
A differenza delle socket TCP, le **socket UDP**:

- sono connectionless → non esiste connect() che prevede l'handshake a tre vie.
- rapido: non prevede però nessun riordino e ripristino.

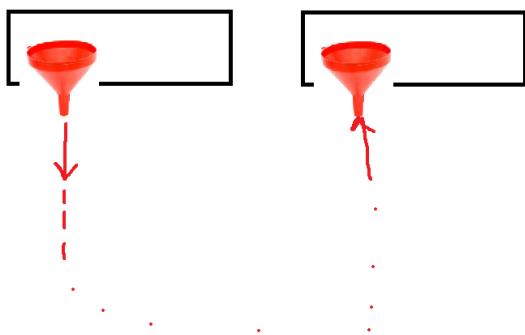
NB: Il TCP è orientato alla connessione, questo vuol dire che si stabilisce un canale dentro il quale anche il server se vuole può inviare dati. Per questo motivo abbiamo bisogno di una `listen()` e di una `accept()` lato server e soprattutto di una `connect()` lato client.

Con UDP invece questo canale non esiste. Il client conosce l'indirizzo del Server e gli invia dati senza che il server abbia modo di inviarli al client (tanto non deve comunicargli nulla). Per questo motivo si sono eliminate certe funzioni.

INVIARE DATI: `sendto()`



La `send()` studiata finora aveva bisogno solo della socket e da dove prelevare i dati da inviare perché l'indirizzo di destinazione del server era implicito in quanto la socket era già connessa:



Mentre con `sendto()` abbiamo bisogno dell'indirizzo del server perché la socket non è connessa. Inoltre il canale non è full-duplex. Se il server volesse inviare dati al client lo farà anche lui con `sendto()` tramite la propria socket.

```
ssize_t sendto(int sockfd, const void* buf, size_t len,
               int flags, const struct sockaddr* dest_addr,
               socklen_t addrlen);
```

- `sockfd`: descrittore del socket
- `buf`: puntatore al buffer contenente il messaggio da inviare
- `len`: dimensione in byte del messaggio
- `flags`: per settare le opzioni (lasciamolo a 0)
- `dest_addr`: puntatore alla struttura contenente l'indirizzo del destinatario
- `addrlen`: lunghezza di `dest_addr`
- Restituisce il **numero di byte inviati** (o -1 in caso di errore)
- È **bloccante**: il programma si ferma finché non ha scritto tutto il messaggio

RICEVERE DATI: recvfrom()

```
ssize_t recvfrom(int sockfd, const void* buf, size_t len,
                 int flags, struct sockaddr* src_addr,
                 socklen_t addrlen);
```

- **sockfd**: descrittore del socket
- **buf**: puntatore al buffer contenente il messaggio da inviare
- **len**: dimensione in byte del messaggio
- **flags**: per settare le opzioni
- **dest_addr**: puntatore a una struttura vuota per salvare l'indirizzo del mittente
- **addrlen**: lunghezza di **dest_addr**
- Restituisce il **numero di byte ricevuti**, -1 in caso di errore, oppure 0 se il socket remoto si è chiuso
- È **bloccante**: il programma si ferma finché non ha letto *qualcosa*

Dato che non c'è una connessione, e quindi il server non può usare quel canale per inviare messaggi al client, dovrà salvare l'indirizzo del client dal quale ha ricevuto il messaggio in una struttura dati per poi utilizzare eventualmente un `sendto()` con quell'indirizzo.



LABORATORIO 7

FIREWALL

Esistono due tipi di firewall, quelli che operano nel:

- 1) **network layer**: sono chiamati **package filtering** e operano a livello di TCP/IP controllando gli header IP, TCP e UDP
- 2) **application layer**: operano a livello di applicazione e sono più efficaci ma bisognano di molte risorse computazionali (gateway a livello applicativo).

PACKET FILTERING

FIREWALL A LIVELLO DI RETE

Abbiamo visto che una possibile politica di filtraggio si attua tramite le **liste degli accessi**. Questa **tavella di regole** contiene appunto delle **regole** che specificano l'azione da intraprendere (che può essere DROP o ACCEPT) per un pacchetto che rientra in determinate caratteristiche. Il packet filtering può essere **stateless** nel senso che controlla campi statici come indirizzo di sorgente o destinazione oppure **stateful**, ossia oltre alla lista degli accessi viene abbinata una **tavella delle connessioni** che mantiene traccia delle connessioni TCP e UDP in corso.

Vediamo il packet filtering di tipo *stateless*.

Indice	IP sorgente	Porta sorgente	IP destinatario	Porta dest.	Azione
1	131.114.0.0/16		131.114.54.4	80	SCARTA
2	0.0.0.0	23	112.143.2.2		ACCETTA

Regola 1: scarta tutti i pacchetti provenienti dalla sottorete 131.114.0.0/16 destinati alla porta 80 del destinatario 131.114.54.4

Regola 2: accetta tutti i pacchetti diretti al destinatario 112.143.2.2, provenienti dalla porta 23 (Telnet) di qualsiasi sorgente

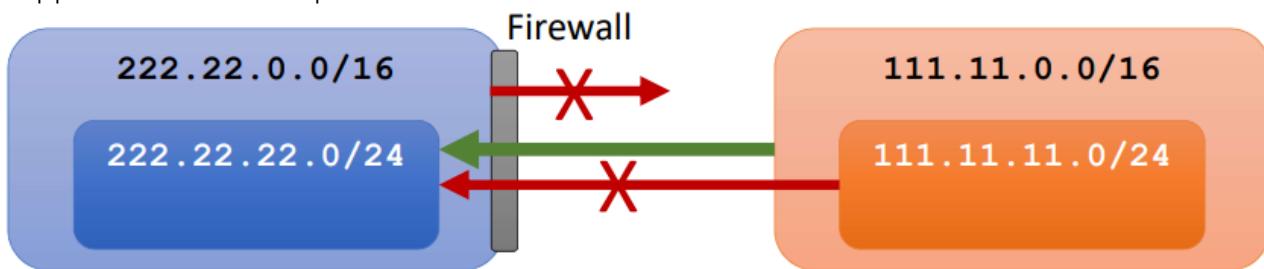
Lo pseudo codice che controlla il pacchett è il seguente:

Per ogni pacchetto p :

1. estrai l'header di p
2. **for** $i := 1$ to n_regole **do**
 - 2.2 considera la regola r_i (cioè quella con $\text{Index} == i$)
 - 2.3 **if** $\text{caratteristiche}(p) == \text{caratteristiche}(r_i)$ **then**
 - 2.3.1 Esegui l'azione della regola r_i
 - 2.3.2 **break**
 - 2.4 **end if**
 - 2.5 $i := i + 1$
3. **end for**

Il break evidenziato vuole sottolineare come **per ogni pacchetto si attua una e una sola regola**. Pertanto se si trova un pacchetto le cui caratteristiche rientrano in una i -esima regola, allora la si applica e si esce subito non controllando le restanti.

Ciò che notiamo da questo pseudocodice è che **l'ordine è importante**. Un pacchetto potrebbe rientrare in più regole, ma il break serve a fermarlo alla prima. Pertanto bisogna anche pensare all'ordine (dal più restrittivo al meno restrittivo) con cui riempire le tabelle. Supponiamo di avere questa situazione:



La sottorete $222.22.0.0/16$ non può accedere a internet, quindi un firewall dovrebbe evitare che pacchetti *interni* vadano fuori.

Una sua sottorete, la $222.22.22.0/24$ è raggiungibile dalla sottorete esterna $111.11.0.0/16$ ma non da un'altra sottorete esterna $111.11.11.0/24$

✗

Indice	IP sorgente	Porta sorgente	IP destinatario	Porta dest.	Azione
1	111.11.0.0/16		222.22.22.0/24		ACCETTA
2	111.11.11.0/24		222.22.0.0/24		BLOCCA
3	0.0.0.0		0.0.0.0		BLOCCA

Sarebbe un errore riempirla così perché la $111.11.11.0/24$ rientra nelle caratteristiche della $111.11.0.0$ e quindi la si accetta.

✓

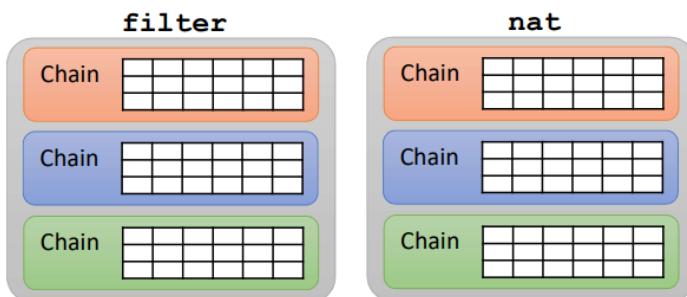
Indice	IP sorgente	Porta sorgente	IP destinatario	Porta dest.	Azione
1	111.11.11.0/24		222.22.0.0/24		BLOCCA
2	111.11.0.0/16		222.22.22.0/24		ACCETTA
3	0.0.0.0		0.0.0.0		BLOCCA

Bisogna quindi scambiare i record delle due regole per avere un ordinamento dal più restrittivo al meno restrittivo.

E' quindi **l'ultima regola** a definire la cosiddetta **regola di default** che se *blocca tutto* allora il firewall si dirà **inclusivo**, altrimenti se *consente tutto* si dirà **esclusivo**.

PACKET FILTERING CON NETFILTER e IPTABLES

Netfilter è un componente del kernel di Linux che permette di intercettare e manipolare i pacchetti ma offre anche funzionalità al livello di rete come configurazioni complesse di NAT e filtraggio stateless e stateful dei pacchetti. **Iptables** è il programma che serve agli amministratori per configurare netfilter e quindi le regole dei filtri e così via.



Iptables lavora su due tipi di tabelle: filter e nat. Ciascuna tabella possiede all'interno diverse chain.

Il comando:

```
iptables [-t table] -L [chain]
```

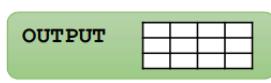
permette di visualizzare della *table* filter o nat la relativa *chain*. Se non si sceglie la *table* sarà di default filter.

FILTER STATELESS

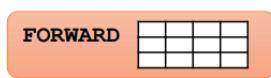


La tabella filter ha 3 catene:

- **input** è destinata a contenere le regole per i pacchetti destinati ai processi locali.



- **Output** destinata a contenere regole per i pacchetti che devono uscire fuori dai processi locali.



- **Forward** è destinata ai pacchetti che passano dal sistema ma non sono ne inviati dall'interno del sistema né è tale sistema la destinazione finale, ma questo viene usato solo come passaggio.

```
root@giuliofederico-VirtualBox:/home/giuliofederico/Scrivania# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
         

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
         

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Per visualizzare queste tabelle basta usare il comando **iptables -L** per vedere le 3 chain della tabella filter.

In questo caso ogni campo è vuoto e ci sono solo le intestazioni semantiche.
Come si aggiungono le regole?

```
# iptables [-t table] -A chain rule-specification
```

↑
Describe la regola,
vediamo fra poco il formato...

Ogni regola aggiunta viene messa all'ultimo posto a meno che non si specifichi la posizione:

```
# iptables [-t table] -I chain [num] rule-specification
```

Per quanto riguarda la rimozione di regole:

- Per rimuovere una regola dalla catena:

```
# iptables [-t table] -D chain rule-specification  
# iptables [-t table] -D chain num
```

- Per rimuovere tutte le regole da una o più catene:

```
# iptables [-t table] -F [chain]
```

- Per cambiare la regola di default (*policy*) DROP/ACCEPT:

```
# iptables [-t table] -P target
```

Per quanto riguarda il formato delle rule-specification è il seguente:

- | | |
|--------------------------------|---|
| • -p <protocollo> | protocollo (TCP, UDP, ICMP, ...) |
| • -s <address> | indirizzo sorgente |
| • -d <address> | indirizzo destinazione |
| • --sport <port> | porta sorgente |
| • --dport <port> | porta destinazione |
| • -i <interface> | interfaccia di ingresso |
| • -o <interface> | interfaccia di uscita |
| • -j <target> | azione (DROP/ACCEPT) |

Per esempio:

```
# iptables -A OUTPUT -p tcp -d 10.0.5.4 --dport 80 -j DROP
```

Aggiungi in fondo alla catena OUTPUT della tabella filter una regola che scarti tutti i pacchetti TCP destinati alla porta 80 (HTTP) dell'host 10.0.5.4

```
# iptables -A INPUT -p udp -s 121.0.0.0/16 -j ACCEPT
```

Aggiungi in fondo alla catena INPUT della tabella filter una regola che lasci passare Tutti i pacchetti UDP provenienti dalla sottorete 121.0.0.0/16

```
# iptables -A INPUT -p icmp -i eth0 -j DROP
```

Aggiungi in fondo alla catena INPUT della tabella filter una regola che scarti tutti i pacchetti ICMP provenienti dall'interfaccia d'ingresso eth0

Questa è la tabella risultante dopo delle aggiunte (3):

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT    udp  --  nip-121-0-0-0.onqnetworks.net/16  anywhere
DROP      icmp --  anywhere             anywhere

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DROP      tcp  --  anywhere             10.0.5.4          tcp dpt:http
```

Se riavviassimo il sistema, le tabelle non rimarrebbero nel loro stato attuale di modifica ma si resetterebbero. Per salvare tutto basta veicolare l'output verso un file:

iptables-save > file

All'avvio del sistema, per ricaricare la tabella faremo:

iptables-restore < file

FILTER STATEFUL

Abbiamo visto che le liste degli accessi lavorano spesso anche insieme alle tabelle delle connessioni. Nella teoria queste due tabelle erano diverse, ossia per ogni riga, se questa prevedeva un controllo nella tabella delle connessioni, si andava in un'altra tabella.

Qui la tabella filter può contenere 3 campi che agiscono da tabella delle connessioni. In particolare ad ogni riga di filter si possono aggiungere 3 possibili campi:

- **NEW**: il pacchetto porta con se uno stato che indica che vuole iniziare una nuova connessione.
- **ESTABLISHED**: il pacchetto porta con se uno stato che indica che è associato a una connessione già esistente.
- **RELATED**: il pacchetto porta con se uno stato che indica che sta stabilendo una nuova connessione ma è associato a una connessione già esistente. Per esempio FTP stabilisce due connessioni: una per le informazioni di controllo e una per i dati veri e propri.

Per aggiungere uno o più di questi campi a una delle righe di una delle chain di filter si usa

-m state --state <tipo>

Dove tipo è uno o più tra NEW,ESTABLISHED e RELATED.

Supponiamo di volere che un host 192.168.10.1 sia accessibile via ssh (porta 22) solo dal computer 192.168.10.5 [ma non possa iniziare sessioni ssh da solo](#).

```
# iptables -P DROP
# iptables -A INPUT -p tcp ! -s 192.168.10.5
    -d 192.168.10.1 --dport 22
        -m state --state NEW,ESTABLISHED -j ACCEPT
# iptables -A OUTPUT -p tcp -o eth0 -s 192.168.10.1
    -d 192.168.10.5 --sport 22
        -m state --state ESTABLISHED -j ACCEPT
```

Intanto ci si sta riferendo a uno specifico host quindi la tabella è di INPUT e poi di OUTPUT. La prima regola si legge "Accetta il pacchetto in ingresso solo se proveniente da 192.168.10.5 e diretto verso

192.168.10.1 sulla porta 22 ma che sia un pacchetto che voglia stabilire una nuova connessione tcp oppure che si riferisca ad una tcp già stabilita."

Quindi se arriva per la prima volta un pacchetto che vuole stabilire una connessione, allora potrà passare così come i restanti che non sarebbero passati se prima non ci fosse stata una connessione.

La seconda regola dice "Accetta di far uscire pacchetti dall'interfaccia eth0 provenienti dall'host interno 192.168.10.1 dalla porta 22 e destinato a 192.168.10.5 solo se una connessione è stata già stabilita". Questo impone che 192.168.10.1 possa partecipare a una connessione solo se già stabilita, ma dalla prima regola chi può creare una nuova connessione è solo 192.168.10.5

Altro esempio. Vogliamo che il firewall blocchi le connessioni dall'esterno ma permetta quelle che partono dalla rete 192.168.10.0/24

```
# iptables -A FORWARD -s 192.168.10.0/24 -i eth0
    -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT
# iptables -A FORWARD -d 192.168.10.0/24 -i eth1
    -m state --state NEW -j DROP
```

Siccome non si parla di uno specifico host interno, ma la protezione è più esterna, allora

usiamo FORWARD, che è la prima tabella da cui si filtrano i pacchetti prima di andare internamente alla rete locale. La regola dice "Accetta tutte le connessioni da 192.168.10.0/24 sull'interfaccia di ingresso eth0 solo se vogliono stabilire una nuova connessione oppure se si riferiscono ad una già stabilita o ancora che sono associati ad una connessione già stabilita."

La seconda regola dice: "scarta tutti quei pacchetti che hanno come destinazione (nota la d) 192.168.10.0/24 in ingresso su eth1 che vogliono stabilire una nuova connessione".

NAT e PAT

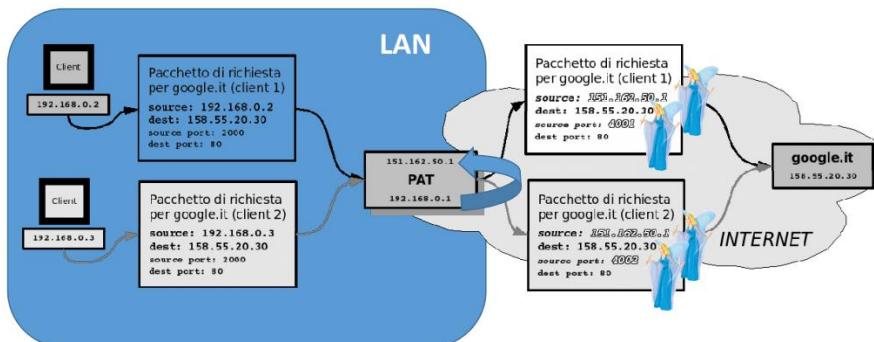
Abbiamo già affrontato il **NAT** nella teoria, quindi non dirò altro. Però sottolineo una cosa. Il NAT discusso era in realtà il **PAT**, ma ormai dato che si installa nelle reti domestiche sempre il **PAT** diciamo che quando si parla di NAT si parla di PAT. Il NAT pure in origine prevedeva un'associazione 1:1 tra indirizzo IP privato e pubblico. Questo significa che per ogni indirizzo privato si doveva associare un indirizzo pubblico. Pertanto se si hanno 10 dispositivi privati e il router ha a disposizione un solo indirizzo IP pubblico, allora potranno certamente uscire tutti e 10 nell'internet globale ma sicuramente a uno a uno. Per questo il NAT si è evoluto a PAT così che si possa associare lo stesso indirizzo IP a tutti e 10 ma si riesca a distinguere ciascuno dal numero di porta come visto prima.

L'estensione di NAT che si chiama **PAT** sta per Port Address Translation o **NAT sovraccaricato**. Il PAT non è un miglioramento del NAT, ma è solo un modo per far sì che il router possa assegnare anche indirizzi IP privati identici a host differenti in quanto utilizza anche la porta per differenziarli.

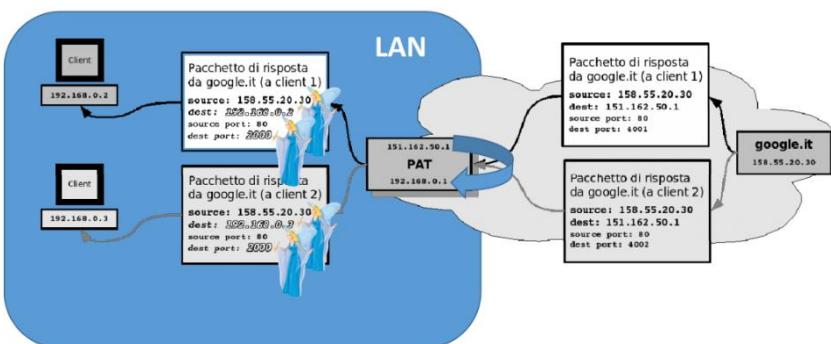
NAT
123.2.3.4

PAT
123.2.3.4:8080

Nota bene: quando un host vuole collegarsi fuori dalla rete può avere di per sé una sua porta detta **porta interna**, ma il router gli associa (a livello socket per esempio) una **porta esterna**.



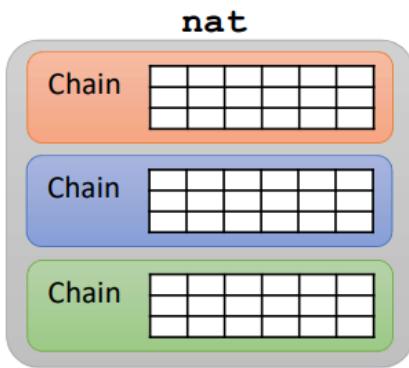
avranno quella porta ma l'indirizzo (come visto anche col NAT) del router.



Ogni host ha un proprio IP e una propria porta. In questo caso entrambi hanno la stessa *porta interna* 2000. Quando fanno richiesta, il router associa a ciascuno una *porta esterna* diversa e all'esterno i pacchetti

Quando google risponde lo fa mettendo come porta di destinazione quella del mittente come normalmente sia. Quando il PAT riceve il pacchetto e legge la porta esterna 4001, sa che appartiene all'host 2000 e così lo inoltra a lui.

Abbiamo visto prima che il nat è la seconda tabella su cui lavora iptables:



Col comando iptables si può modificare le tabelle NAT oppure NAPT. In genere infatti si sceglierà sempre un NAT e in più se si vuole, si specifica anche il sovraccarico PAT. Sarà più chiaro con l'esempio.

Le 3 chain sono:

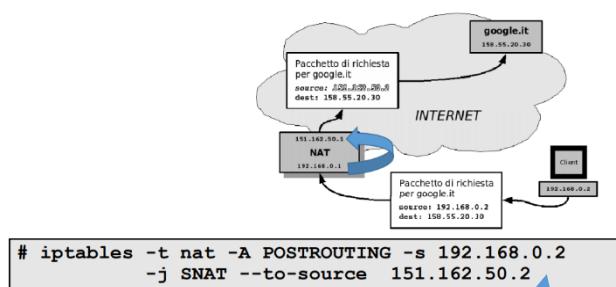
- Prerouting
- Output
- postrouting

Quello che è importante capire è che il NAT (o l'estensione NATP) deve fare due cose:

- 1) Per i pacchetti che vogliono uscire deve cambiare l'indirizzo sorgente con quello del router (o uno degli ip pubblici disponibili al router).
- 2) Per i pacchetti che vogliono entrare deve cambiare l'indirizzo destinatario con quello dell'host interno a cui in realtà è destinato il pacchetto.

Per la 1) si parla di S-NAT, ossia un NAT o NATP in cui si modifica la Sorgente, mentre per la 2) si parla di D-NAT, ossia un NAT o NATP in cui si modifica la Destinazione.

S-NAT (pacchetti in uscita)



```
# iptables -t nat -A POSTROUTING -s 192.168.0.2
-j SNAT --to-source 151.162.50.2
```

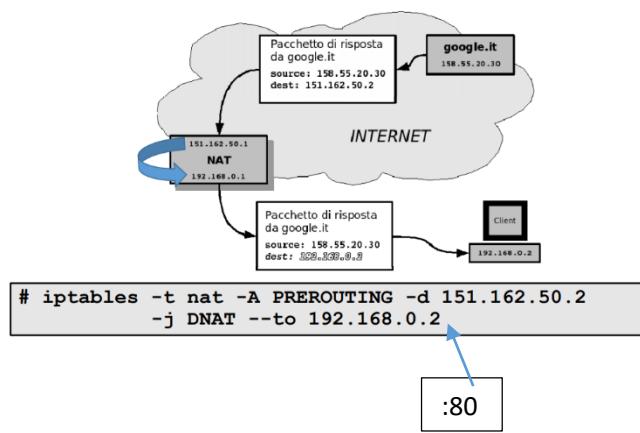
:4001-4100

esempio S-NAT con NAT semplice

L'IP privato dell'host viene sostituito con quello pubblico del router. La regola dice: "inserisci nella tabella POSTROUTING, che si occupa dei pacchetti in uscita, la regola che i pacchetti provenienti dall'host 192.168.0.2 devono aver cambiata la sorgente con 151.162.50.2.

Se io qui avessi messo anche questo significa che stavo facendo uno S-NAT con PAT, ossia gli dicevo "...sorgente con 151.162.50.2 ma impostami la porta anche tra un pool (range) che sta tra 4001 e 4100.

D-NAT (pacchetti in ingresso)

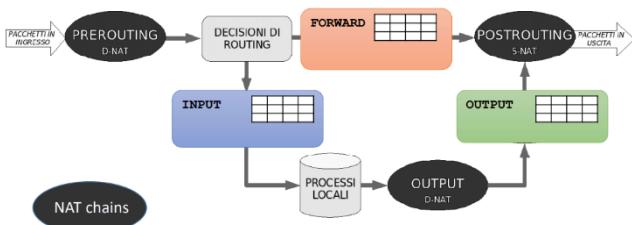


esempio DNAT con NAT semplice

Nella tabella PREROUTING, che si occupa di gestire le regole dei pacchetti in ingresso, inserirà la regola "per tutti i pacchetti in ingresso aventi come destinazione 151.162.50.2 cambiamo l'indirizzo di destinazione in 192.168.0.2"

Anche qui avrei potuto fare un DNAT con PAT dicendogli di non modificare solo la destinazione al livello IP ma anche la porta impostandola ad 80.

FILTER e NAT insieme

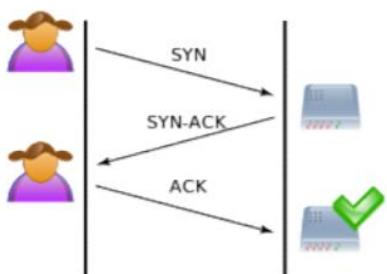


Abbiamo visto che il filter applica regole su pacchetti con sorgente e destinazione reali/concreti, insomma quelli finali. Pertanto nella catena di composizione stanno sempre in modo tale da vedere indirizzi "reali".

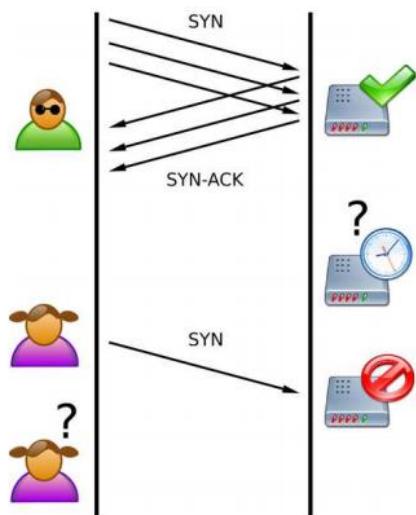
Syn Flood: attacco DOS

Un Syn Flooad è un attacco di tipo denial of service, ossia uno di quegli attacchi che mira a esaurire le risorse di un sistema così da rendergli impossibile fornirlo ad altri utenti e quindi mandandolo in tilt. Questo particolare attacco DOS si chiama Syn perché sfrutta i messaggi di richiesta di connessione tcp Syn inviati da un host al server al quale richiede tale connessione.

Three-way handshake TCP/IP (uso normale)



In una richiesta normale e corretta si usa concludere l'handshake a 3 vie.



In un attacco Syn invece si inviano ripetuti Syn creando un primo canale di comunicazione che però sappiamo va solo da sx verso dx, mettendo in una attesa infinita il server che si aspetta l'ultimo ACK per completare la connessione. Per ogni Syn ricevuto il server ha allocato risorse e riservato una connessione in quanto se arriva qualcun altro non è giusto che il precedente perda il posto. Ma così facendo nega a tutti la connessione perché ha ricevuto troppi Syn, ossia troppe richieste di riservare connessioni.

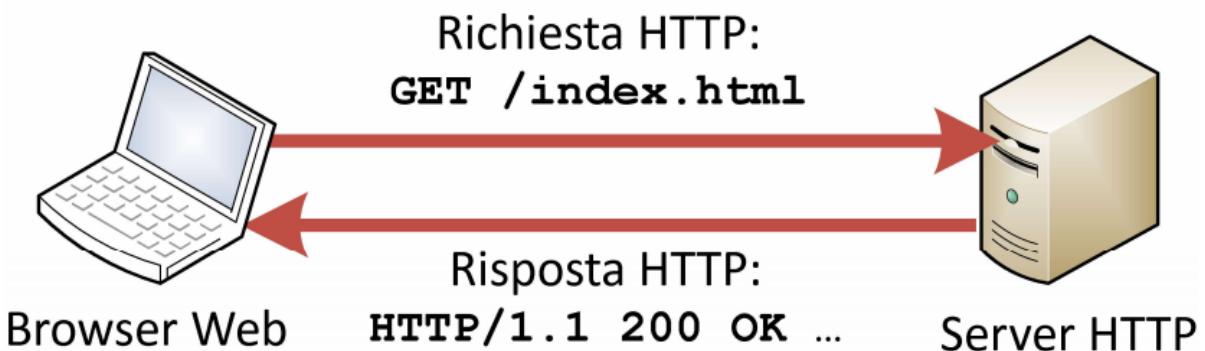
Quello allora che si fa è di accettare un solo pacchetto Syn al secondo:

```
# iptables -A INPUT -p tcp
          --syn -m limit --limit 1/s -j ACCEPT
```



APACHE HTTP

APACHE HTTP SERVER



Andare a ripassare il paragrafo PROTOCOLLO HTTP.

Un **server Apache** è un applicazione che permette a un proprietario di sito web di gestire le sue macchine che consentono agli utenti di far raggiungere il proprio sito. Il suo compito è quindi quello di monitorare, organizzare e mettere in sicurezza il lavoro necessario per accedere ai file richiesti da un browser utente per il proprio sito. In sostanza il Web Server Apache installato sulla tua macchina *gestisce in sicurezza il trasferimento di pagine web richieste da un browser*.

Di norma il programma su Debian è noto come **apache2**, e qualora non fosse installato basterà come al solito scrivere:

```
sudo apt install apache2
```

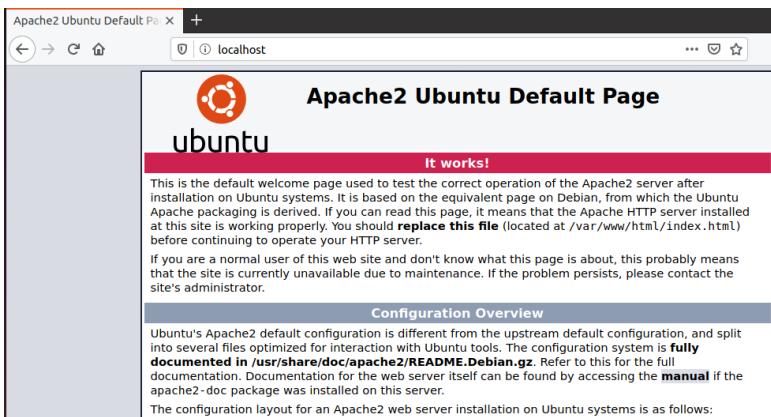
Per avviare il server si usa:

```
service apache2 <command>
```

dove command può essere *start*, *stop*, *restart*, *reload*. La differenza tra reload e restart sta nel fatto che reload è più veloce perché fa un riavvio di apache semplice, mentre restart lo riavvia totalmente (quindi è consigliabile solo quando il reload non aggiorna i cambiamenti o quando sappiamo di aver fatto cambiamenti sostanziosi).



Quando il server apache è stoppato allora non possiamo vedere il nostro sito perché non c'è un server web che ne gestisce tutto l'invio http di ciò che serve a farlo comparire.



Appena però lo attivo (start) allora il web server sarà in ascolto sulla porta 80 per ricevere richieste e rispondere con una pagina HTML di default. La pagina mostrata si trova nel percorso /var/www/html/index.html

CONFIGURAZIONE WEB SERVER APACHE

Il web server Apache viene configurato tramite il file principale `/etc/apache2/apache2.conf`. Questo file include dentro se dei file.conf di configurazione, come `ports.conf` che stabilisce la porta su cui sta in ascolto e questa inclusione la fa tramite la direttiva `include`. Il web server però è **modulare** e **multisito**. La prima ci indica che possiamo personalizzare il suo funzionamento attivando e disattivando certi moduli, mentre la seconda indica che può gestire più siti.

Oltre quindi ai file.conf ce ne sono alcuni che sono direttamente collegabili a tre macrocategorie impersonate da tre cartelle:

```
/etc/apache2/
|-- apache2.conf
|   '-- ports.conf
|-- mods-enabled
|   '-- *.load
|   '-- *.conf
|-- conf-enabled
|   '-- *.conf
|-- sites-enabled
|   '-- *.conf
```

- **Mods-enabled** contiene file (in realtà soft link) di configurazione complessi dei moduli.
- **Conf-enabled** contiene file (in realtà soft link) di configurazione semplici/basilari dei moduli.
- **Sites-enabled** contiene file (in realtà soft link) di configurazione degli host virtuali.

Queste 3 cartelle vengono incluse nel file principale tramite la direttiva `includeOption`.

Parlo di soft-link perché queste 3 cartelle hanno ognuna le loro omonime chiamate però rispettivamente *mods-available*, *conf-available* e *sites-available*. Quando si vuole scrivere un file di configurazione, si deve in realtà per prima cosa metterlo in una di queste 3 cartelle *available* (a seconda della categoria a cui appartiene). Le rispettive omonime *enabled* contengono i soft-link di quei file nelle rispettive *available* che sono stati attivati (io potrei infatti creare un file di configurazione, ma non per forza in questo momento attivarlo → implica la personalizzazione di cui si parlava prima).

Per rendere attivo un file di configurazione dentro la cartella *conf-available* si utilizza il comando:

a2enconf <nomeFile>

mentre per disattivarlo si usa il comando:

a2disconf <nomeFile>

Per rendere attivo un file di configurazione dentro la cartella *mods-available* si utilizza il comando:

a2enmod <nomeFile>

mentre per disattivarlo si usa il comando:

a2dismod <nomeFile>

Per quanto riguarda la cartella *sites-available* verrà discussa nel sottoparagrafo "Direttive per siti web".

Nota bene: dopo ogni modifica è consigliabile fare il *reload* o al massimo il *restart* per rendere effettive le modifiche.

Nota bene: quando si scrive il file non usare l'estensione *.conf*, basta il nome.

DIRETTIVE GLOBALI

Una **direttiva globale** importante che contiene il file principale di configurazione apache2.conf è la direttiva **ServerRoot**:

```
Apri ▾ F apache2.conf [Sola lettura]
/etc/apache2

# NOTE! If you intend to place this on an NFS (or otherwise network)
# mounted filesystem then please read the Mutex documentation (available
# at <URL:http://httpd.apache.org/docs/2.4/mod/core.html#mutex>);
# you will save yourself a lot of trouble.
#
# Do NOT add a slash at the end of the directory path.
#
#ServerRoot "/etc/apache2"
#
# The accept serialization lock file MUST BE STORED ON A LOCAL DISK.
#
#Mutex file:${APACHE_LOCK_DIR} default
```

Indica quale path utilizzare per risolvere i path relativi delle altre direttive. Per esempio sempre nel file apache2.conf esiste la *include ports.conf*; grazie al ServerRoot questa sarà risolta come: /etc/apache2/ports.conf

Altre due direttive globali importanti sono:

```
Apri ▾ F apache2.conf [Sola lettura]
/etc/apache2

#
# Timeout: The number of seconds before receives and sends time
#
Timeout 300

#
# KeepAlive: Whether or not to allow persistent connections (more
# one request per connection). Set to "Off" to deactivate.
#
KeepAlive On

#
# MaxKeepAliveRequests: The maximum number of requests to allow
# during a persistent connection. Set to 0 to allow an unlimited
# We recommend you leave this number high, for maximum performance
#
MaxKeepAliveRequests 100

#
# KeepAliveTimeout: Number of seconds to wait for the next request
# same client on the same connection.
#
KeepAliveTimeout 5
```

KeepAlive che indica se la connessione è persistente o meno.

KeepAliveTimeout indica il tempo massimo che il server attenderà una richiesta dal client in una connessione permanente. Se scaduto il tempo il client non ha fatto alcuna richiesta, allora la connessione viene chiusa.

Una diretta globale già vista è la **Listen <porta>** che indica il numero di porta sulla quale stare in ascolto ma questo è fatto dal file ports.conf incluso come detto prima.

La diretta globale **ErrorLog** specifica invece il file su cui scrivere i log di errore.

DIRETTIVE PER SITI WEB -----

Come detto prima il server web apache è multisito, ossia permette di gestire più siti.

Attraverso il campo host della richiesta http riesce a discriminare quale sito sia.

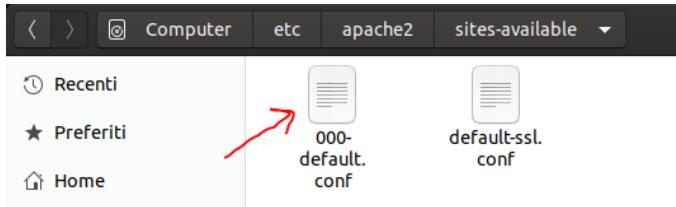


I siti si trovano dentro la cartella `sites-available` e vengono attivati (ossia finiscono nel file di configurazione principale apache2.conf) tramite il comando:

`a2ensite <nomeFile>`

E vengono disabilitati alla gestione tramite il comando:

`a2dissite <nomeFile>`



Esiste un sito web default nel file `000-default.conf` che viene subito abilitato ed è relativo al protocollo http (invece quello accanto è relativo all'https).

Vediamo com'è fatto dentro:

```
<VirtualHost *:80>
    ServerName www.example.com
    ...
    DocumentRoot /var/www/html
</VirtualHost>
```

Lo schema generale è

```
<VirtualHost ip:porta>
    ...
</VirtualHost>
```

Questo tipo di virtual host è detto **name-based** in quanto non si distinguono i differenti siti web per indirizzi IP, il che permette di risparmiare i costi nell'assegnare a ciascuno il suo, ma sui nomi consentendo quindi un *unico indirizzo IP condiviso*.

Infatti all'interno è così fatto:

```
<VirtualHost *:80>
    ServerName mio_sito.home.lan
    ServerAlias alias1.home.lan alias2.ciao.abc mio_sito
    DocumentRoot "/var/www/cartella_mio_sito/"
    DirectoryIndex pagina_principale.html
    <Directory "/var/www/mio_sito">
        Require ip 127.0.0.0/8
        Require ip 192.168.0.0/16
    </Directory>
</VirtualHost>
```

Altre direttive interne sono:

- **ServerName** ospita il nome del sito e quindi permette la discriminazione.
- **DocumentRoot** specifica la directory dove trovare i file del sito, ossia quelli che saranno messi a disposizione al client.
- **DirectoryIndex** è opzionale e specifica il file da prelevare nel caso non sia specificato dal cliente. In pratica se l'utente cerca miosito.com è carino e classico uso inviargli una home page di benvenuto, a meno che non ne richieda una specifica.

Un'altra direttiva è **UserDir**. Di norma è disabilitata ma può essere attivata tale modulo come `a2enmod userdir.conf`. Una volta abilitato cosa ci permette di fare? Se guardiamo dentro il

file userdir.conf notiamo che ci dice che la cartella di default dove cercherà i file è *public_html* nella home. Pertanto ritorniamo su VirtualHost e inseriamo la direttiva UserDir public_html e nella nostra home creiamo una cartella nomeUtente/public_html in cui inseriremo tutti i file che vogliamo siano accessibili. In questo modo l'utente digitando:

www.mysite.com/~username/

accederà alla nostra cartella home.

MODULI MULTI-PROCESSO

Apache accetta più richieste contemporaneamente e lo fa tramite i **Multi-Processing Module (MPM)** che gestiscono i socket, fanno il binding delle porte e accettano e servono connessioni.

Apache supporta 3 tipi di MPM:

- Prefork
- Worker
- Eventi

PREFORK

Il modulo prefork implementa un server multi-processo ma senza thread. Questo vuol dire che nel caso peggiore di una sola cpu possono essere eseguiti contemporaneamente più processi ma ciascuno occupa sue risorse private di memoria che non condivide con altri. Un processo padre crea StartServer numero di processi figli (simile a quando si faceva fork()). Ciascun figlio è detto *worker* e può gestire un numero massimo di richieste pari a MaxRequestsWorker e ogni volta che serve una può essere riciclato/riusato di nuovo per una nuova connessione fino a un numero massimo di MaxConnectionsPerChild. Il padre cerca di tenere un numero x di figli inattivi, con x che va da MinSpareServers a MaxSpareServers.

WORKER

Il modulo worker è identico al prefork ma realizza anche il multi-thread. Infatti ogni figlio può generare due tipi di thread: un solo thread listener che si occupa di attendere la richiesta e worker quelli che ricevono dal listener la richiesta per servirla. A livello di codice è più complicato e meno compatibile ma sicuramente il fatto di usare thread worker e listener (che quindi fondamentalmente sostituiscono gli stessi figli) fa sì che la memoria sia molto condivisa. Ogni figlio può creare un massimo di ThreadPerChild workers.

EVENT

E' un worker migliorato in quanto se un worker capisce che il client tarda a inviargli una richiesta allora passa il controllo del socket al listener che lo darà ad un altro thread. Quando arriverà la richiesta del client allora gli sarà assegnato un nuovo worker libero.