

Lez. 1

Internet: descrizione degli “ingranaggi”

- Miliardi di dispositivi di calcolo sono contemporaneamente connessi ad internet. Questi sono definiti come sistemi periferici (end system) o host (poiché ospitano le applicazioni di rete) e lo scopo ultimo di internet è quello di interconnetterli tra di loro. Le applicazioni di rete (Web, Telegram, Whatsapp...) sono infatti applicazioni di rete che in ultima istanza sono in esecuzione su sistemi periferici, sono gli host che eseguono le applicazioni di rete, e per questo le applicazioni di rete risiedono ai confini di internet (edge).

Con host oltre che a calcolatori comuni come cell, pc etc... si fa riferimento anche a macchine più potenti (server) su cui sono in esecuzione i programmi che erogano i servizi su cui noi comuni mortali accediamo quotidianamente.

- Spostandoci verso l'interno abbiamo i Comutatori di pacchetto (packet switches). Date n linee di ingresso e n linee di uscita, il commutatore di pacchetto si occupa di inoltrare i pacchetti da una linea ad un'altra. Si parla di Router se lavorano a livello di rete e di Switch se lavorano a livello di collegamento.

- Tra tutti i router e tra tutti i nostri dispositivi cablati fisicamente ci sono delle linee di comunicazione. Si parla in questo senso di Reti di Collegamenti (communication links) e ne esistono di diversi tipi: fibra ottica, rame (cavo ethernet), segnali radio, satellite...

Ciò che differenzia questi canali è la velocità di trasmissione (transmission rate o bandwith == larghezza di banda).

-L'ultimo ingranaggio di internet è rappresentato dalle reti. Con rete si intende una collezione di host, router e collegamenti gestiti da una singola organizzazione (rete aziendale, domestica, mobile)

Internet non è una singola enorme rete gestita da un'unica entità, ma è formata da una serie di reti più o meno grandi e a più livelli interconnesse tra di loro a formare una rete di reti.

(Alcune anticipazioni)

ISP ==Internet Service Provider (fornitori del servizio di Internet)

Sono le entità che ci consentono l'accesso ad Internet. Due dispositivi interconnessi nella mia rete domestica potrebbero dialogare senza dover niente a nessuno, ma se volessi dialogare con una francese me tocca fa l'abbonamento a Iliad (gli ISP gestiscono una rete generalmente molto grande)

Ma le reti degli ISP non coprono di certo tutto il globo, quindi a loro volta hanno bisogno di interconnettersi ad altri ISP di livello eventualmente superiore a formare una gerarchia per cui si può arrivare ad avere copertura globale per Internet.

Reti di distribuzione di contenuti sono reti private di grandi conduttori di servizi (es. Google) che cercano di aggirare la rete pubblica per arrivare il più vicino possibile ai propri utenti.

Tra i sistemi periferici si passa dai PC desktop ai server, dai dispositivi mobili alle auto etc...

Vi sono col passare del tempo sempre più cose che oggi rappresentano host ma che in passato non erano connesse (es. termostato, che collegato ad internet può dare altri servizi come fornire statistiche) -> il termine "rete di calcolatori" è ormai obsoleto (Internet of Things).

Internet rappresenta quindi una rete di reti con ISP interconnessi tra loro, fornendo quindi ampia copertura.

Ogni qual volta che abbiamo due entità che in Internet interagiscono reciprocamente questa interazione avviene secondo Protocolli che regolano lo scambio d'informazioni. Ad esempio usando il protocollo Skype si possono fare chiamate tra dispositivi potenzialmente distanti continenti tra loro, oppure via protocollo 4/5G possiamo mettere in comunicazione un'auto con la stazione base della rete mobile, via protocollo IP possiamo invece inoltrare i pacchetti attraverso la rete etc...

Se due entità dialogano allora il dialogo è regolato da Protocolli.

Quando si dialoga si presuppone che vi sia comprensione reciproca del protocollo tramite cui dialoghiamo. In questo senso entrano in gioco gli enti di Standardizzazione. I protocolli di Internet sono in primo luogo standardizzati dalla **IETF == Internet Engineering Task Force**, gli standard di Internet prendono il nome di **RFC == Request for Comments**.

Esistono anche altri enti di standardizzazione, come **IEEE 802 LAN/MAN Standards Committee** per Ethernet, wireless Wi-Fi e altro.

Internet: descrizione dei “servizi”

Da questo punto di vista, Internet rappresenta una infrastruttura che offre dei servizi alle applicazioni (Web, streaming di video e musica, giochi, messaggistica). Quindi le applicazioni in sé, in un certo senso, non fanno parte di internet ma si poggiano ad esso.

In quanto infrastruttura di servizi Internet offre ai vari processi in esecuzione nelle macchine Host un'interfaccia di programmazione, detta interfaccia Socket, che permette a questi processi remoti di interconnettersi tra loro permettendo il "trasporto" di informazioni tra estremi remoti delle applicazioni in rete. Si ha un "hook" che consente alle applicazioni mittente/destinataria di "connettersi" usando proprio il servizio di trasporto di Internet, analogamente al servizio postale.

Es. Ho il browser, la pagina che voglio sta in qualche server canadese, usando il protocollo di trasporto di Internet posso connettermi a quel server utilizzando in particolare il protocollo applicativo HTTP per richiedere quella pagina.

Cos'è un Protocollo?

Protocolli Umani rappresentano interazioni, scambi di messaggi specifici per un determinato "contesto della domanda. Ad esempio a "che ore sono?" si risponde con un orario, non con una cosa arbitraria. Un altro aspetto dei protocolli è regolare quali azioni svolgo quando ricevo un messaggio o capitano altri eventi (ad es. se dopo aver chiesto l'ora il tipo non mi risponde, in tal caso ci si deve comportare di conseguenza)

Quando si parla di Protocolli di Rete la situazione è molto simile, ma dobbiamo scambiare le persone con i dispositivi. Tutta l'attività di comunicazione di Internet è regolata, come abbiamo visto, da protocolli.

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

Il contenuto dei messaggi che scambio con un'altra entità **deve seguire una struttura nota a priori** (scambio un certo tipo d'informazioni in un certo modo in base al protocollo a cui faccio riferimento, es. se chiamo via Skype non sto giocando a Elden Ring).

I messaggi (informazioni) non arrivano istantaneamente, c'è un tempo di ricezione che dipende da diversi fattori.

Es. Protocollo di rete:

Richiesta di connessione TCP

Risposta di connessione TCP

GET http://gaia.cs.umass.edu/kurose_ross

<file>

(dove TCP rappresenta il protocollo che permette al processo del Browser di comunicare al processo del Server per chiedergli la pagina, usando a livello applicativo lo specifico protocollo HTTP)

Uno Sguardo da vicino alla struttura di Internet

- Capita di fare informalmente riferimento agli host come Client e Server.

In realtà i **Server rappresentano gli Host che erogano servizi**, mentre **i Client gli host che richiedono servizi**.

I Server si trovano tipicamente nei **data center** (grandi edifici in cui vi sono tantissime macchine interconnesse tra loro a formare un cluster).

Es. Data Center di Amazon, Google etc...

Cloud Computing: Piuttosto che acquistare, possedere e mantenere i data center e i server fisici, è possibile accedere a servizi tecnologici, quali capacità di calcolo, archiviazione e database, sulla base delle proprie necessità affidandosi a un fornitore cloud. Pago solo quanto mi è necessario usare.

- Reti di Accesso e Mezzi Trasmissivi

La **rete di accesso** è la rete che collega gli host periferici al loro **edge router** (ovvero il primo router del cammino verso host non appartenenti alla stessa rete di accesso). Entra in gioco quando a partire dal mio host devo comunicare con host che non fanno parte della mia rete.

Quando studiamo le reti di accesso dobbiamo guardare vari parametri, come il **tasso di trasmissione** e se l'accesso è dedicato o condiviso.

Esistono varie tecnologie che permettono di connetterci al nostro ISP

-Reti di accesso: accesso via cavo

Si ha una **stazione di testa** (cable headend) da cui si dirama un cavo coassiale condiviso tra più abitazioni. Su questo cavo passano segnali elettrici che possono avere diverse frequenze, secondo un **Multiplexing a divisione di frequenza FDM canali diversi sono trasmessi in bande di frequenza diverse.** (televisione, trasmissione dei dati...)

Nell'abitazione è presente uno **splitter**, cioè un dispositivo che a partire da queste frequenze in base alla loro ampiezza sono mandate alla tv o al modem via cavo. **Modem sta per Modulator Demodulator**, e si occupa di **modulare** segnali digitali (0 e 1) in segnali analogici elettrici, e viceversa di **demodulare**.

Esiste anche una soluzione ibrida **HFC (hybrid fiber coax)** in cui nel mezzo è buttata anche un po' di fibra ottica per velocizzare le cose. Il cavo in fibra ottica è in questo caso piazzato tra la stazione di testa e nodi distrettuali da cui si dirama il cavo coassiale per raggiungere le abitazioni. Si ha in questo caso nella stazione di testa da cui partono i vari cavi una CMTS (sistema di terminazione del modem via cavo)

Si nota come la velocità di trasmissione risulta essere **asimmetrica** nei due versi: se scarico (**Downstream**) dai 40Mbps (40 milioni di bit) agli 1.2Gbps (1.2 miliardi di bit). Se invece carico (**Upstream**) dai 30 ai 100 Mbps. Velocità quindi inferiore in upstream, diverse ragioni per cui ciò avviene, vedremo nella DSL nel dettaglio una di queste.

Nel caso della HFC, così come in generale l'accesso via cavo, l'utenza domestica **condivide la rete di accesso** per cui i segnali sono in realtà propagati a tutte le abitazioni. Se scarico un file, questo va tutte alle abitazioni (protetti via uso di crittografia). Inoltre se ho 40Mbps sul cavo tocca condividere con gli altri, quindi se molti scaricano contemporaneamente non tocca aspettare i secoli.

-Reti di accesso: DSL (digital subscriber line)

In questo caso l'ISP non è più l'operatore della TV via cavo ma tipicamente l'operatore della linea telefonica. Si utilizza lo stesso doppino utilizzato per il telefono. Il trucco, così come per la rete di accesso via cavo, è di suddividere la banda su doppino in più canali: un canale resta dedicato al telefono (0-4 kHz), un altro in Upstream (4-50 kHz) e un altro in Downstream (50-100 kHz). Avremo anche in questo caso uno splitter che porta le frequenze dedicate alla telefonia al telefono, e le altre al Modem.

Il dispositivo che raccoglie i fili delle nostre case si chiama DSLAM (DSL Multiplexer) e, a differenza della CMTS in cui la linea era condivisa, si ha in questo caso una **linea dedicata per ogni casa**.

Non si deve tuttavia, anche in questo caso, **pretendere di navigare su Internet alla velocità dichiarata dall'operatore** per una serie di fattori come limitazioni del provider (più paghi più veloce), distanza, qualità del materiale e interferenze.

La cosa interessante del DSLAM è che, in maniera analoga allo splitter, viene bipartito il segnale: la voce sulla linea telefonica DSL finisce nella rete telefonica, mentre i dati sulla linea telefonica DSL vanno su Internet.

-Reti di accesso: Fibra Ottica (FETTx)

Diversi tipi in base a dove arriva effettivamente la fibra ottica:

- **FTTH** - Fiber-to-the-home (1 Gbps in downlink)
- **FTTB** - Fiber-to-the-building o Fiber-to-thebasement (soluzione intermedia per cui la fibra arriva al piano terra o all'edificio in generale)
- **FTTC o FTTS** - Fiber-to-the-cabinet o Fiber-tothe-street (100/200 Mbps in downlink) (la fibra arriva al cabinet, che deve trovarsi a distanza inferiore ai 300 metri dall'abitazione, da casa al cabinet infatti ci si arriva via tecnologia DSL usando doppini in rame. Maggiore è la distanza minore le prestazioni.
- **FTTN** - Fiber-to-the-node (>300m)
- **FTTW o FTTR** - Fiber-to-the wireless o Fiber-tothe-radio (letteralmente "fibra fino alla base radio"). La Fibra arriva ad un'antenna che trasmette poi il segnale via radio.

Quanto più il collegamento ottico arriva vicino alla destinazione, tanto maggiore sarà la velocità raggiunta nell'ultimo tratto (es. attraverso le diverse tecnologie DSL)

Vediamo nel dettaglio la **FTTH**:

Esistono due architetture:

- **Active Optical Network (AON)**: sono delle Ethernet commutate, cioè una serie di ripetitori con trasmettitori e recettori ottici
- **Passive Optical Network (PON)**: non ho una trasmissione del segnale tramite trasmettitori e recettori ma lavoro direttamente con il segnale ottico.

Approfondendo il tipo **PON**, si ha una similitudine con la distribuzione via cavo coassiale visto in precedenza: nella centrale locale ho un OLT (optical line terminator) dove arriva la fibra ottica. L'OLT ha la funzione di prendere il segnale ottico e di trasformarlo in dati numerici da poi immettere nei router. La fibra diramata dall'OLT può servire più abitazioni, in genere un centinaio, con uno splitter nel mezzo necessario a multiplare

il segnale nelle varie abitazioni. Si ha una **linea condivisa** come era per la connessione via cavo, per cui se scarico arrivano i file pure agli altri ma crittografati (inoltre non sempre si hanno le prestazioni sperate se più persone scaricano contemporaneamente).

-Rete di Accesso FWA (mista fibra/radio, Fixed Wireless Access).

Si tratta del caso in cui si usano le onde radio (tecnologia wireless) non per ragioni di mobilità, ma per via del fatto che non ho i mezzi per estendere un cavo in fibra ottica. Così facendo riesco a portare Internet a velocità piuttosto elevate, fino ai 100Mbps per rete a banda ultralarga, e si utilizzano tecnologie radio (wireless) diverse, tra cui il 5G.

Quindi per connettere una casa ad Internet:

Modem via cavo, DSL, vari tipi di Fibra.

Fibra e Cavo condivisi, DSL linea dedicata ma, nel momento della moltiplicazione in presenza di alto traffico, la velocità potrebbe essere compromessa (e altri fattori ancora che compromettono).

Nelle nostre case tuttavia non abbiamo solo il modem, ma a tutti gli effetti una rete domestica che interconnette gli host presenti nell'abitazione tramite filo (ethernet) o in maniera wireless.

Si parla di **Reti Locali Wireless**.

Di fatto i 3 componenti **Modem, Router e Access Point Wireless** sono combinate in un unico dispositivo, il **WLAN**, che si collega alla fibra o al doppino nel caso della DSL e che presenta porte ethernet se voglio attaccare via cavo, altrimenti supporta il WiFi.

Il WiFi è una delle principali tecnologie di supporto wireless a corto raggio (100 m circa).

Si ha poi l'**Accesso Wireless su scala geografica**, fornito dagli operatori mobili che si estende per decine di km. Le prestazioni sono limitate rispetto a un classico WLAN e si sfruttano tecnologie radio 4G e 5G.

-Reti di accesso: accesso aziendale

Per aziende, università etc...

Simili alle reti che abbiamo a casa, un po' più complesse. Oltre ai router (si occupa dell'inoltro pacchetti), ai Modem e agli Host si hanno dispositivi detti **switch** che operano a livello di collegamento.

Le reti aziendali non sono inoltre tipicamente connesse via DSL, ma tramite tecnologie a più alta velocità (es. ethernet).

-Reti di accesso: reti dei data center

Per erogare servizi per client si deve far riferimento a macchine molto potenti, in questo senso si fa riferimento a Data Center dove decine di migliaia di computer sono interconnessi tra loro e connessi ad internet. Vi è all'interno dei Data Center una rete molto veloce che interconnette i vari host, vi è poi una rete che interconnette i vari Data Center a Internet.

Host: invio dei pacchetti di dati

Negli host risiedono delle applicazioni che comunicano tramite scambio di messaggi. Se un'applicazione vuole mandare un messaggio (informazioni) usa l'interfaccia socket del proprio host per chiedere di inviare il suddetto messaggio. L'implementazione dello stack di rete nell'host si occuperà poi di portare le informazioni a destinazione seguendo un percorso in Internet. *L'host che decide di inviare un messaggio si occupa di suddividerlo in frammenti più piccoli, detti **PACCHETTI**, di lunghezza **L bit**.*

I pacchetti vengono immessi sul canale di comunicazione ad una velocità definita come **tasso di trasmissione R (detta anche capacità del link)** espressa in **bit/sec**. Se ho un collegamento da un Gbps significa che io host

posso trasmettere bit alla velocità di un miliardo al secondo, 10 Mbps 10 milioni di bit al secondo.

Data la velocità di trasmissione R e la lunghezza del pacchetto L definiamo **ritardo di trasmissione del pacchetto** il *tempo necessario a trasmettere effettivamente i bit (L/R)*.

Ritardo di Trasm. del pacchetto == Tempo nec. per trasmettere pacchetti di L bit nel collegamento == L/R

Per comunicare, un host mette i bit nel canale di comunicazione, che viaggiano da un'estremità (trasmettitore) a un'altra (ricevitore) passando per un mezzo che può essere **vincolato** (guided media: i segnali si propagano per mezzo solido quindi attraverso un cavo di qualche tipo), **non vincolato** (unguided media: se il segnale viaggia liberamente nello spazio frapposto, es. radio).

Vi sono diversi mezzi con cui possiamo implementare il canale, l'esempio più tipico è il **doppino intrecciato** (*twisted pair*) costituito da due fili di conduttore (tipicamente rame) intrecciati tra loro. L'intreccio serve a evitare interferenze. Ne esistono di diverse categorie che differiscono per la loro struttura, infatti fili più performanti non solo intrecciano più coppie ma hanno anche diversi strati di materiale schermante per limitare interferenze. Tanto più abbiamo schermatura tanto più porno ad alta qualità.

Categoria 6 10Gbps Ethernet (distanze inferiori a 100m).

Si ha poi il **cavo coassiale** (visto per la rete via cavo) per cui si hanno due conduttori di rame concentrici. Grazie alla presenza di uno strato schermante e alla struttura di **banda larga** può supportare diversi canali ad alta frequenza per cui posso avere anche centinaia di Mbps per canale.

La **fibra ottica** passa per un materiale flessibile e sottile che conduce impulsi di luce, ciascuno dei quali rappresenta un bit. Si possono avere frequenze diverse e la caratteristica più eclatante è il fatto che supporti elevatissime velocità di trasmissione (fino a decine e centinaia di Gbps se collego due router via fibra con linea dedicata). Se ho poi un cavo con molte fibre posso spingermi ancora più in alto.

Inoltre si ha un'attenuazione molto bassa anche per centinaia di chilometri, per cui non necessita di utilizzare tanti ripetitori per trasmettere gli impulsi per lunghe distanze. Altra caratteristica risiede nel fatto che la fibra essendo un segnale di luce è totalmente immune all'interferenza elettromagnetica (a diff. del caso degli altri cavi basati sull'elettricità)

La tecnologia della fibra ottica è quella utilizzata nei tratti oceanici per collegare tra loro i vari continenti.

Per quel che riguarda i **Canali Radio** non necessita ovviamente di cavi, il segnale è trasmesso infatti nello spettro elettromagnetico. Sono mezzi direttamente broadcast, cioè se mando un segnale radio questo viene ricevuto in **broadcast** da tutti. Sarà ancora una volta la crittografia a proteggere la segretezza delle informazioni. Pertanto si deve adottare un uno schema **half-duplex**, per cui si deve *alternare trasmissione e ricezione* (altrimenti ci si parlerebbe sopra).

L'ambiente può interferire pesantemente sulla propagazione dei segnali: riflessione, ostruzione da parte di ostacoli, interferenze...

Esistono diversi tipi di canali radio, la differenza principale risiede nella scala di propagazione e nell'efficienza di trasmissione.

- WLAN (WiFi) decine di metri, alpiù centinaia di Mbps,
- wide-area (4 e 5G) decine di km
- Bluetooth (utilizzato in sostituzione dei cavi, distanze brevi e velocità molto limitate)
- Microonde terrestri (punto a punto, canali fino a 45 Mbps)
- Satellitari (fino a meno di 100 Mbps in downlink per Starlink) e ritardo punto-punto di 270 ms (geostazionari)

Per i Satellitari si hanno satelliti a bassa quota (LEO), media quota (MEO) e geostazionari (GEO). Il vantaggio dei GEO è che stanno fermi in un punto del cielo e le antenne non hanno quindi di seguire qualcosa che si muove, lo svantaggio è ovviamente la distanza (più tempo impiegato a propagare il segnale, non si può superare la velocità della luce, si arriva anche oltre i 270ms, si ricorda che la mente umana per percepire qualcosa come istantaneo non si deve andare sopra i 100ms).

Pertanto si utilizzano satelliti di comunicazione in base all'esigenza a quote sempre più basse, lo svantaggio è che sti satelliti si muovono più veloci della terra a ruotare e quindi tocca seguirli con l'antenna.

Lez. 2

Riassuntino

Nodi, dispositivi e router connessi tramite linee di comunicazione che possono essere di diverso tipo (DSL, cavo, fibra etc...).

Rete == insieme di dispositivi, commutatori di pacchetto e collegamenti gestiti da una singola organizzazione. Non avremmo un internet globale se queste reti non fossero interconnesse tra loro -> rete di reti.

Internet of Things (non più solo calcolatori nelle reti)

ISP internet service provider, fornitore dell'accesso a internet

Nella nostra abitazione o istituzione abbiamo la rete di accesso, basata su tecnologie come ethernet e wifi, questa rete deve essere connessa all'ISP per andare oltre la nostra rete

Esistono diverse reti di accesso, come residenziale, aziendale o mobile

La rete di accesso ha il compito di connettere gli host agli edge router, cioè il primo router nel cammino verso l'esterno della rete di accesso.

Differenza sostanziale tra DSL e connessione via cavo: nel primo caso collegamento dedicato, nel secondo caso cavo coassiale per cui collegamento condiviso.

A casa nostra abbiamo quindi una piccola rete locale, in cui potrebbe essere presente un router o anche un access point wireless (wifi), che però non ci rendiamo nemmeno conto di avere perché contenuti nella stessa scatola.

I mezzi trasmissivi hanno sempre prestazioni limitate dalla lunghezza del mezzo (ethernet)

La fibra ottica rappresenta un'eccezione in questo senso, anche centinaia di km (cavi sottomarini in fibra ottica). Così come il doppino di rame può contenere più coppie intrecciate, anche il cavo di fibra ottica può contenere tante più "fibre" (trasmetto più dati).

Comunicazione wide area -> 4G, 5G

Per la trasmissione satellitare satelliti quali GEO (molto distanti) girano sincroni con la rotazione terrestre, quindi non serve che le antenne si muovano nel puntarli, tuttavia rappresentano uno svantaggio in termini di distanza (280 ms di ritardo)

Nucleo della rete

Si tratta di una **maglia** (o **mash**) di collegamenti che hanno come ultimo fine quello di connettere tra loro i sistemi periferici (host).

Internet utilizza a tale scopo una tecnologia chiamata **commutazione di pacchetto** (**packet switching**). Si ricorda che una delle funzioni degli host è quella di prendere i messaggi applicativi, frammentarli in pacchetti e chiedere al servizio di rete di mandarli a destinazione. Per farlo ciascun router del cammino riceve il pacchetto e lo **inoltra (forwards)** rilanciandolo su un collegamento in uscita verso un altro router ancora fino ad arrivare a destinazione, lungo un **percorso (path)** dalla sorgente al punto di arrivo.

Il nucleo della rete ha **due funzioni chiave**:

1) Funzione di “**inoltro**” (*forwarding* o *switching*), è **un’azione locale** a ciascun router (già anticipata appena prima) che sposta i pacchetti in arrivo nel collegamento d’ingresso del router verso il collegamento d’uscita appropriato.

Per farlo il router guarda dentro l’intestazione del pacchetto, dove è scritto l’indirizzo di destinazione (in forma digitale, bit/numero). Il router appresa questa informazione accede a una sua struttura dati, detta **tabella di inoltro** che gli suggerisce, dato l’indirizzo di destinazione, dove inoltrare il pacchetto.

2) Funzione di “**instradamento**” (*routing*):

Ovviamente la tabella di inoltro non contiene l’indirizzo specifico di dove si trova l’host a cui spedire il pacchetto: nel mondo esistono infatti miliardi di host, sarebbe scomodo e inefficiente.

Il discorso è quindi come quello che si fa con la posta: se voglio spedire una lettera a New York il postino non sarà interessato alla via e al civico esatto di dove devo spedirla, ma guardando la città di New York saprà che deve essere affidata ad un centro specifico che la manderà in America, dove si occuperanno loro di inoltrarla specificatamente al destinatario.

*L’instradamento è quindi **un’azione globale** che determina i percorsi presi dai pacchetti dalla sorgente a destinazione, e lo fa secondo un*

algoritmo di instradamento. (analogo sotto certi aspetti a Google Maps, trovo il “cammino minimo” a livello globale prima di inviare i pacchetti. L’azione di “inoltro” rappresenterebbe la scelta di quale incrocio prendere di volta in volta per raggiungere destinazione).

Commutazione di pacchetto: store-and-forward

Abbiamo detto che se ho un pacchetto di L bit e lo posso spedire attraverso un collegamento che regge R bit/sec, allora il tempo impiegato è L/R (analogo a dire 100 km a 100 km/h \rightarrow 1 ora)

Ricordiamo kb == 10^3 bit, Mb == 10^6 bit, 1 ms == 10^-3 sec

Store and forward: invio un pacchetto al router, dopo L/R secondi arriva tutto il pacchetto. I bit non arrivano tutti insieme, ma man mano che li trasmetto. Immaginiamo di avere un router tra sorgente e destinazione. Il router non può prendere e rilanciare il bit appena lo riceve, poiché prima deve leggere l’indirizzo di destinazione. Pertanto i router in internet operano in modalità store and forward, *cioè un router deve ricevere completamente un pacchetto prima di poterlo inoltrare al passo successivo*, questa tecnica ha una ripercussione importante sul tempo impiegato ad attraversare n collegamenti.

Immaginiamo 3 pacchetti da inviare, S sorgente, D destinazione e R router nel mezzo.

S 321	R	D
S 21	R 3	D
S 1	R 2	D 3
S	R 1	D 23
S	R	D 123

Tot: $4L/R$. Senza router nel mezzo ci avrei impiegato $3L/R$, store and forward vantaggioso!

Il ritardo (tempo impiegato) da un capo all’altro (**end to end**) per la trasmissione di un pacchetto su un percorso di N collegamenti di pari velocità R sarà:

$$\underline{\text{dend-to-end} = N \frac{L}{R}}$$

(tralasciando il ritardo di propagazione ed altre forme di ritardo)

Immaginiamo ora di avere P pacchetti. Quanto tempo impiegato a trasmetterli? Dopo N L/R secondi arriva il primo pacchetto. Ne dovranno arrivare altri P-1, ma i pacchetti rimangono consecutivi nell'essere lanciati, quindi al prossimo L/R arriva un pacchetto, al prossimo ancora un altro per un totale di P-1.

$$\text{dend-to-end} = (\mathbf{N} + \mathbf{P} - 1) \text{ L/R}$$

(tralasciando il ritardo di propagazione ed altre forme di ritardo)

Ritardo di accodamento

Nel packet switching le risorse sono condivise, in particolare un pacchetto viene trasmesso su un canale di uscita solo se questo è libero, ovvero se non sto trasmettendo altro. **Che succede se trasmetto altro?**

In tal caso il pacchetto in attesa viene posto in un **buffer**, a formare una coda di pacchetti che attendono di essere serviti. **L'accodamento (queuing) si verifica quando il lavoro arriva più velocemente di quanto possa essere servito.**

Ese. ho un link che trasmette a 10Mb/s, ma contemporaneamente deve inviare dati a 100Mb/s, arrivano più bit di quanti ne riesca a rilanciare.

Se il tasso di arrivo in bps eccedesse il tasso di trasmissione in bps per un tempo sufficientemente lungo, poiché non ho memoria infinita allora non vi sarebbe più spazio nei buffer per mettere altri bit in attesa... **perdita di dati** (buffer in overflow).

Dovendo costruire una rete di commutatori e collegamenti per trasmettere dati, *Internet utilizza prevalentemente la commutazione di pacchetto* (quindi store and forward), ma esiste un'alternativa tipica nella rete telefonica tradizionale chiamata **commutazione di circuito**.

In questa tecnologia risorse quali buffer e velocità di trasmissione sono riservate per ciascuna comunicazione tra sistemi periferici nell'arco dell'intera sessione di comunicazione

Ciò porta al vantaggio di avere **risorse dedicate**, per cui non posso trovarmi nella situazione di far arrivare dati al commutatore e non avere la possibilità di rilanciarli. Avrò quindi **velocità di trasferimento costante e garantita** (utile per il telefono, comunicazione real time, so a priori quanta banda mi serve e la riservo in anticipo).

C'è però un **problema grave**, più evidente nelle reti di calcolatori. Se riservo una risorsa per una comunicazione, quando i due sistemi non comunicano avviene che **la risorsa è sprecata!**

Due tipi di commutazione di circuito: **FDM** (frequency division multiplexing) e **TDM** (time division multiplexing).

Supponiamo di avere un canale di comunicazione, ma voglio la possibilità di avere più circuiti. Se ho anche un solo canale di comunicazione, **se questo ha una banda sufficientemente grande posso dipartirla!** Le tecniche per dividerle sono la FDM e TDM.

FDM

La luce rappresenta un'onda elettromagnetica. A seconda della frequenza di un'onda, la luce assume colori diversi. Si dice che la luce bianca contiene tutte le frequenze. Se in mezzo alla luce bianca metto un prisma (esperimento di Newton), questa viene scomposta nell'arcobaleno (sto scomponendo le varie frequenze).

Se prendo tre sorgenti, R, B e V, e le uso tramite lampaggi per trasmettere informazioni, se le sparo sulla stessa fibra ottica dall'altro capo se sparano R, B e G vedrò bianco, se sparano R e B viola e se sparano R e V vedrò giallo.

Essendo le combinazioni limitate si può dedurre, in base al colore che arriva, quali sono stati i colori trasmessi dall'altro capo in modo non ambiguo e quindi dedurre i dati inviati dalla sorgente (non si perde l'informazione!).

Il concetto della FDM è la stessa, non applicata alla luce, ma più in generale a dei segnali. **Se ho segnali che hanno frequenze diverse, posso metterli insieme ottenendo un segnale fuso da cui posso dedurre le frequenze inviate originariamente.**

Un mezzo non può trasportare qualsiasi frequenza, ma un certo numero di frequenze determinate dalla banda passante. Data la banda passante di un mezzo, quindi l'insieme di tutte le frequenze, le divido in blocchetti e assegno a ciascun utente uno di questi blocchetti. Possiamo immaginarlo come con i

colori: ad ogni utente assegno un colore, in base al colore che arriva capisco quali colori sono stati spediti.

Poiché la velocità di trasmissione è legata all'ampiezza della banda che ho, restringendola potrò mandare l'informazione ad una velocità più bassa.

TDM

Il canale viene suddiviso in questo caso secondo il dominio del tempo. Divido il canale in elementi di durata fissa, detti **frame**, che si ripetono nel tempo, e ciascun frame lo divido in slot temporali. Ottengo così n slot che posso assegnare periodicamente a ciascun utente.

Permetto all'utente di trasmettere in un tempo pari alla durata del frame alla massima velocità della banda di frequenza, in tal modo ci si alterna continuamente nella trasmissione. Ognuno trasmette alla massima velocità del canale, ma solo per un momento ristretto (che dipende da frame), per cui la velocità media sarà ovviamente più bassa di quella massima.

Quale è meglio tra **commutazione di pacchetto** e **commutazione di circuito**?

Un grosso limite della commutazione per circuito è l'efficienza.

Se ho infatti un collegamento da 1 Gbs e voglio avere circuiti da 100 Mbs, posso avere allora al più 10 utenti collegati al collegamento. Ma comunque se qualche utente non sfrutta il collegamento allora sto sprecando la banda. **La commutazione di pacchetto evita questo problema** perché *non riservo in anticipo le risorse*, semplicemente non appena arrivano i pacchetti se il canale è libero allora trasmetto. *Il problema è che se trovo qualcun altro in trasmissione non posso trasmettere, devo attendere.*

Ci si rende conto che in questo senso, per lo meno riguardo il traffico in rete, conviene usare comunque la commutazione di pacchetto. Immaginiamo ancora di avere un collegamento da 1 Gbs e circuiti d'entrata da 100 Mbs, e ipotizziamo che ogni utente utilizzi il collegamento il 10% del tempo. Allora indubbiamente starei sprecando risorse tramite la commutazione di circuito!

Nel caso invece della commutazione del pacchetto, immaginando di avere 35 utenti, si può dimostrare via distribuzione binomiale che la probabilità di avere più di 10 utenti attivi contemporaneamente se ognuno di essi usa il

collegamento il 10% del tempo è pari ad al più 0.0004! (1-0.0004 è più del 96%, ho probabilità superiore al 96% di non avere più di 10 utenti attivi)

In definitiva il vantaggio della commutazione a pacchetto è che, poiché non devo conservare la banda a priori ma la servo on demand, posso supportare più utenti di quelli che potrei fare allocando staticamente le risorse.

Ciò conviene per l'invio a raffica, garantendo un approccio più semplice (i router non devono riconoscere che ci sono circuiti con certe "regole").

Tuttavia, proprio perché non riservo a priori le risorse, non ho garanzia di trovarle quando mi servono! (rischio la congestione del buffer e la perdita di dati)

Gerarchia degli ISP

Immaginiamo di avere una “costellazione” di diversi ISP, ne abbiamo milioni, come li collego tra loro? Tramite una maglia completa avrei un sistema *non scalabile* (mantiene inalterata la sua gestibilità indipendentemente da quante organizzazioni lo utilizzano), $O(n^2)$.

L’idea è allora la seguente: esiste una sorta di **ISP globale** a cui sono connessi tutti gli altri, in questo modo gli ISP più piccoli diventano clienti degli ISP globali (fornitori) tramite un accordo economico (si paga per inoltrare i dati).

Se un business è profittevole ciò comporta competizione; esistono quindi più ISP globali di transito. È tuttavia necessario che i vari ISP siano sempre interconnessi tra loro (altrimenti non vi sarebbe copertura globale), e in questo senso esistono gli **IXP (Internet Exchange Point)**. Si tratta di strutture, veri e propri edifici, che consentono l’incontro tra diversi ISP che lì giungono tramite le proprie terminazioni. In questo modo è possibile interconnettere il tutto garantendo lo scambio di pacchetti di dati.

Un altro approccio, più diretto, è quello del **peering link**: due ISP si interconnettano tra di loro tramite un link diretto garantendo uno scambio a costo zero.

È difficile che un ISP globale sia in grado di coprire tutte le città di ogni nazione, in questo senso esistono ISP di gerarchia inferiore come **ISP nazionali** o **regionali**. Esistono poi tra ISP di gerarchia diversa dei collegamenti attraverso un **PoP (point of presence)**. Si tratta di router del tutto dedicati al collegamento tra ISP di diversa gerarchia (in genere anche qui collegamento garantito da un contratto economico).

In generale in Internet vale la regola che peering tra ISP di pari gerarchia è gratuito, mentre quando un ISP di livello inferiore si collega ad un ISP di livello superiore tocca sborsare la grana (accordo commerciale).

Un ulteriore ruolo in questa situazione lo svolgono i **fornitori di servizi** (Google, Microsoft etc...). Questi gestiscono una propria rete globale, cioè oltre ad avere data center sparsi in tutto il mondo, ma questi data center non sono collegati tra loro tramite Internet ma tramite una rete proprietaria (in molti tratti realizzata in fibra ottica, con prestazioni molto elevate). Inoltre alcuni di questi data center svolgono la funzione di connettere la rete del fornitore al resto di Internet. Come lo fa?

Si tenta in primo luogo di accordarsi con gli ISP di livello più basso, che però hanno numeri più alti (reti di accesso etc...). Lavorando in questo modo i fornitori evitano di servirsi degli ISP di livello più alto, che si farebbero pagare. *Ciò ha anche un'altra importante conseguenza: si gestisce meglio la fornitura del servizio ai propri utenti.*

Più vicino infatti si arriva all'utente finale, migliori saranno le prestazioni percepite dei servizi.

Si arriva così all'[architettura di Internet](#):

- Si ha un numero ristretto di **ISP Tier 1** (ISP globali di transito)
- Rete di **fornitori di contenuti** (come Google), ovvero reti private che connettono i loro data center ad Internet spesso aggirando ISP Tier 1 e regionali e accordandosi con ISP di livello più basso (ovviamente anche questi fornitori possono connettersi agli IXP, dove diversi ISP si incontrano)

Torniamo ora a parlare di commutazione di pacchetto.

Come si verificano ritardi e perdite?

Abbiamo già detto che tramite commutazione di pacchetto (invio di volta in volta i pacchetti) i pacchetti si accodano nel buffer del router, aspettando il proprio turno per la trasmissione. Se il tasso di arrivo di pacchetti sul collegamento eccede per un determinato tempo la velocità di trasmissione del collegamento a partire dal router che li deve spedire allora, poiché il buffer non ha memoria infinita, si rischia di perdere informazioni.

Quando un pacchetto è pronto per essere trasmesso, si perde un certo tempo per trasmetterlo (*per leggere l'intestazione di instradamento, accedere alla tabella di instradamento, fare un controllo di errore sui bit etc...*), si parla in questo senso di **ritardo di elaborazione**. Questo è tipicamente un ritardo irrisorio, dell'ordine dei microsecondi (ms, 10^{-6} sec, un milionesimo di secondo).

Si parla di **ritardo di accodamento** quando un pacchetto 1 viene inoltrato sulla linea di uscita opportuna e quella linea è già occupata dall'inoltro di un altro pacchetto 2, per cui il pacchetto 1 deve attendere in coda nel buffer. Il ritardo di accodamento è variabile poiché dipende da quali pacchetti arrivano al router e in quale momento (si tratta di un processo casuale), detto in parole povere dipende dal **livello di congestione del router**.

Il **ritardo di trasmissione** lo abbiamo già ampiamente visto, è rappresentato dalla formula $d_{trasm} = L/R$ e dipende quindi dal numero di bit che voglio trasmettere (L, in bit) e dalle “prestazioni” del mezzo attraverso cui passano (R, in bit/sec).

Il **ritardo di propagazione** è il corrispettivo del ritardo di trasmissione ma in senso puramente fisico, legato al mezzo di trasmissione. Data **d lunghezza del collegamento fisico e v velocità di propagazione (c.a. velocità della luce, $\approx 2 \times 10^8$ m/sec)** si ha $d_{prop} = d/v$.

dprop e dtrasm profondamente diversi tra loro, vedi analogia carovana pg 33 PDF.

Si parla infine di **ritardo end-to-end** come *l'accumulo di tutti i ritardi per ogni nodo lungo il percorso sorgente-destinazione per l'invio di un pacchetto*, si ha:

$$\text{dend-to-end} = \sum_{\text{perogni-}i} (\text{delab-}i + \text{dacc-}i + \text{dtrasm-}i + \text{dprop-}i)$$

NB tutti i ritardi si misurano in ms.

Sia **a** la *velocità media di arrivo dei pacchetti*, **L** la *lunghezza del pacchetto* (in bit) e **R** la *velocità di trasmissione* (in bit/sec).

Allora definiamo come “**intensità del traffico**” la formula $\frac{L \cdot a}{R}$, dove inoltre **L** a rappresenta la velocità di arrivo dei bit e **R** la velocità di servizio dei bit.

Assumendo un traffico casuale in termini qualitativi (la velocità media di arrivo dei pacchetti non è ovviamente costante e dipende da fattori casuali) si possono fare le seguenti assunzioni:

- Se $L \cdot a / R$ è si avvicina a zero allora il ritardo medio di accodamento è piccolo
- Se $L \cdot a / R$ si avvicina ad 1 allora ho un ritardo medio di accodamento grande
- Se $L \cdot a / R > 1$ allora ho più “lavoro” di quanto possa essere servito (arrivano più pacchetti di quanti ne possa mandare) e quindi il ritardo tende a infinito! Posso avere perdite!

In definitiva, non dobbiamo avere intensità di traffico maggiore di 1 (assicurate perdite), ma neanche troppo vicine a 1 (sotto 0.4). È necessario tenersi bassi poiché man mano che mi avvicino ad 1 cresco esponenzialmente e se raggiungo 1 vado all’infinito.

È un po’ come quando mi trovo in una strada leggermente trafficata, basta poco (incidente) perché si stia fermi per tre ore.

Per calcolare questi ritardi esiste un comando specifico in Linux, **traceroute**. Si tratta di un programma diagnostico che fornisce una misura del ritardo dalla sorgente al router lungo i percorsi Internet punto-punto. In particolare, per ogni nodo i del percorso:

- Invia tre pacchetti (per fare una media) che raggiungeranno il router i sul percorso verso la destinazione (campo time-to-live == i, in questo modo so chi è il primo router, il secondo, il terzo etc... (viene incrementata mano che il pacchetto viene inoltrato))
- Il router i restituisce i pacchetti al mittente
- Il mittente calcola l'intervallo tra trasmissione e risposta

(sotto l'esempio)

```
1 cs-gw (128.119.240.254) 1 ms 1 ms 2 ms
2 border1-rt-fa5-1-0.gw.umass.edu (128.119.3.145) 1 ms 1 ms 2 ms
3 cht-vbns.gw.umass.edu (128.119.3.130) 6 ms 5 ms 5 ms
4 jn1-at1-0-0-19.wor.vbns.net (204.147.132.129) 16 ms 11 ms 13 ms
```

...

7 nycm-wash.abilene.ucaid.edu (

198.32.8.46) 22 ms 22 ms 22 ms

(collegamento trans-oceanico, nota come schizza il ritardo da 22 a 106)

8 62.40.103.253 (62.40.103.253) 104 ms 109 ms 106 ms

(perché si scende da 106 a 104? Si torna al discorso casualità! In questo caso evidentemente quando ho mandato i tre pacchetti al router in riga 8 ho avuto una certa congestione nel percorso che, quando poi ho mandato i pacchetti al router in riga 9, non avevo più! (tempo di accodamento variabile!)

9 de2-1.de1.de.geant.net (62.40.96.129) 109 ms 102 ms 104 ms

...

17 * * * (ovvero nessuna risposta: risposta persa, router non risponde)

Perdita di pacchetti

- la **coda** (anche detta **buffer**) che precede un collegamento ha capacità finita
- quando il pacchetto trova la coda piena, viene scartato (e quindi va perso)
- il pacchetto perso può essere ritrasmesso dal nodo precedente, dal sistema terminale che lo ha generato, o non essere ritrasmesso affatto

NB la coda sta davanti alla linea d'uscita, se un router ha tre linee di uscita tipicamente ogni linea ha la sua coda, ciascuna di esse può far overflow separatamente. Quando si parlava di intensità del traffico, quei pacchetti non erano in arrivo al router ma in arrivo alla coda (quindi si parlava del traffico dedicato ad una specifica linea)

Abbiamo visto che le prestazioni delle reti sono date da tre parametri principali quindi: **Ritardi, Perdite** (che sono per lo più causate dalla congestione, ovvero il fatto che un pacchetto trova una coda piena davanti alla linea di uscita) e **Throughput**.

Throughput

Il **Throughput** rappresenta la frequenza di arrivo del bit a destinazione. Come per la velocità di un'automobile si ha l'autovelox che misura la velocità istantanea e il tutor che misura la velocità media, allo stesso modo quando si guarda con che frequenza arrivano i bit (10 al sec, 100 al sec etc...) posso essere interessato alla **frequenza istantanea** che calcolo su intervalli brevi o **frequenze medie** su intervalli più lunghi.

Esempio: dopo aver scaricato tutto un file di 10 Mb, se ci ho impiegato un secondo allora il throughput medio è stato di 10 Mbs (quanti bit ho ricevuto nel tempo).

*Si farà riferimento a scenari del tipo sorgente-destinazione connesse allo stesso router con collegamenti di banda **Rs** e **Rc** (server e client) (banda da sorgente al router Rs, banda dal router a destinazione Rc). Si trascureranno tutti gli altri tipi di ritardo e momenti di alto traffico.*

Se invio le informazioni con velocità Rs minore della velocità Rc con cui mando le informazioni, in media il client riceverà informazioni pari alla velocità minore (Rs). Immagina infatti di avere 1 Mbs per Rs e 100 Mbs per Rc, leggo più velocemente di quello che ricevo e quindi. Vale lo stesso ma all'opposto se $Rs > Rc$: il throughput medio sarà Rc.

In definitiva il throughput end-to-end dipende dalla velocità di trasmissione dei collegamenti attraversati dal flusso di dati.

Se il percorso non è interessato da altro traffico allora questo sarà di media uguale al **$\min\{R_i\}$** dove R_i è la velocità di trasmissione dell'iesimo collegamento (velocità di throughput media è pari a quella del collegamento con prestazioni più basse). È quindi il collegamento con prestazioni inferiori

a determinare la velocità di throughput media e ad essere quindi chiamato collo di bottiglia.

In Internet si hanno collegamenti con velocità di trasmissione molto elevate; quindi non si può ragionare semplicisticamente come in precedenza. Si deve far riferimento alla possibilità che server diversi e client diversi, nel loro comunicare, passino attraverso collegamenti in comune (vedi figura pag 42 PDF 2).

Immaginiamo di avere 10 server e 10 client che comunicano e passano per lo stesso collegamento di velocità di trasmissione R (bit/sec). Allora il throughput end-to-end in questo caso sarà dato, per ciascuna connessione, da **$\min(R_c, R_s, R/10)$** dove R/10 perché ho 10 connessioni in comune per lo stesso mezzo di velocità di trasmissione R!

Poiché solitamente R molto molto grande, i colli di bottiglia sono tendenzialmente dettati da client o server.

Riassumendo: throughput frequenza di arrivo di bit, istantanea e media. Quella **media** la calcolo facilmente: periodo di tempo, bit che ho ricevuto, **bit ricevuti/tempo**.

Colli di bottiglia, se devo giungere più collegamenti è il minimo tra tutti i server, tutti i client e R del mezzo per cui questi client e server comunicano / numero connessioni.

Lez. 3

Riassuntino

Host, commutatori di pacchetto e collegamenti sono le principali componenti dell'Internet, questi possono essere collegati in una rete, Internet rete di reti. Ricordiamo l'importanza dei protocolli (che approfondiremo) e la standardizzazione.

Si ricorda che gli host sono connessi a reti di accesso che li connettono al loro router di bordo, primo host del cammino fuori dalla rete di accesso.

Per connettersi ad Internet esistono diverse tecnologie di accesso (cavo, fibra, DSL). Cavo è un mezzo di comunicazione condiviso, tutti i pacchetti inviati dal CMDS attraverso il cavo coassiale saranno ricevuti da tutti gli utenti e viceversa quando gli utenti inviano un pacchetto devono fare attenzione a non collidere l'uno con l'altro.

La DSL (o ADSL dove A sta per asimmetrica, download >> upload). A seconda della distanza ed altri fattori velocità diverse.

Non è detto che la fibra arrivi direttamente al computer, ma a seconda della distanza avremo prestazioni differenti. Fibra a casa: variante attiva come fosse un ethernet ma con cavi in fibra ottica, variante passiva che ricorda il collegamento via cavo ma leggermente diversa. In questo caso infatti dalla centrale ONT parte un cavo con la fibra che poi tramite splitter ottico si moltiplica in più fibre, allo stesso modo lo splitter combinerà i segnali ottici dalle varie reti di accesso per spararli alla centrale locale.

A casa si ha una rete domestica di cui riconosciamo il modem (connesso al nostro ISP), il router che si occupa di instradare i pacchetti, dispositivi connessi via cavo o tramite WAP (wireless access point). Tutto ciò è in genere fornito in un unico scatolotto.

Wireless con raggio di un centinaio di metri (es. rete domestica) o area geografica (raggio di chilometri), da 10 a 300 Mbs con 4G+.

Differenza tra reti di accesso aziendali e domestiche: presenza di switch. Altra differenza è la presenza di commutatori di pacchetto che collegano agli ISP più veloci del normale.

I server che forniscono i servizi internet che usiamo non sono un'unica macchina ma più macchine collegate tra loro in veri e propri data center. Per reti di data center si intende la rete dentro questi ultimi, è diversa dalla rete dei content provider che invece s'allarga a tutto internet.

La commutazione di pacchetto inizia con la suddivisione delle informazioni da inviare in pacchetti, la lunghezza di ogni pacchetto è limitata ad un certo valore max. Satelliti GEO più comodi perché mi basta stare fermo ma ritardo maggiore. Instradamento, algoritmo di instradamento, etc...

Coda (buffer) dei pacchetti posta di fronte alla linea d'uscita -> tutto il discorso sull'intensità di traffico si applica nel momento in cui vado a depositare del traffico diretto ad una specifica linea d'uscita.

Se esaurisco la memoria del buffer, che si riempie se non posso inviare i dati (traffico di rete), allora perdo messaggi. Per la commutazione di pacchetto i pacchetti possono anche prendere strade diverse (se salta qualche collegamento

si passa agli altri).

Differenza più importante tra commutazione di pacchetto e di circuito: col circuito le risorse sono riservate per tutta la durata di comunicazione tra host attraverso una fase iniziale di setup della connessione, che dà luogo alla formazione di un circuito. Il vantaggio è che avendo risorse dedicate i dati in arrivo hanno garanzia di essere serviti a velocità costante, mentre nel caso della commutazione di pacchetto le risorse sono assegnate on demand. Tuttavia se utilizzo un circuito le risorse allocate sono sprecate. Quindi si utilizza una commutazione o un'altra in base all'esigenza.

Non stiamo parlando per ora di mezzi condivisi con utenti che mandano dati, ma semplicemente di una singola linea che il router deve usare per trasmettere pacchetti. (il resto del riassunto esattamente tutta la roba vista in lez. 2)

Sicurezza di rete

Studiando l'architettura di Internet sui protocolli ci si rende conto che non è stato studiato molto per la sicurezza. Ciò è dovuto al fatto che agli albori internet era fatto per utenti mutualmente fidati, collegati da “una rete trasparente”.

È oggi necessario pensare a come i malintenzionati possano attaccare le reti informatiche, come difenderle e come progettare architetture immuni agli attacchi. Vediamone alcuni:

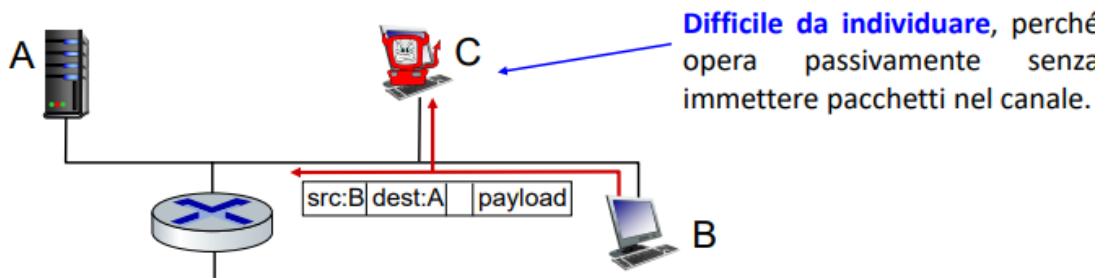
Malintenzionati: intercettazione dei pacchetti

Analisi dei pacchetti (packet sniffing)

Abbiamo visto in precedenza come nella connessione via cavo, wireless ed ethernet condivisa come i pacchetti sono inviati in *broadcast*, cioè sono effettivamente ricevuti da tutti gli host connessi, è ristretto nella policy dell'host l'idea di scartare quelli che non gli sono destinati.

Un malintenzionato potrebbe usare un software, detto **packet sniffer** (vedi Wireshark packet sniffer gratuito), che operando in modalità promiscua *non effettua lo scarto ma legge/registra tutti i pacchetti* (tra cui eventualmente anche le password!) che l'attraversa.

Questo attacco è molto difficile da individuare poiché passivo, il packet sniffer non immette alcun pacchetto nel canale.



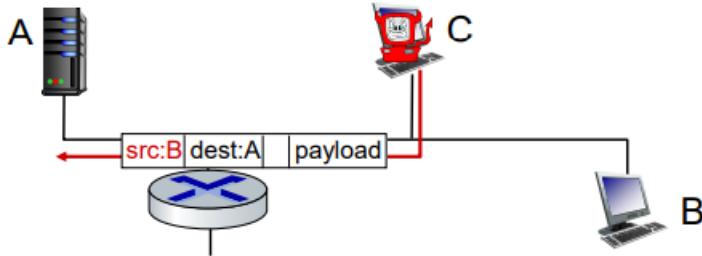
IP spoofing

Rappresenta l'iniezione di pacchetti con indirizzo sorgente falso.

I pacchetti contengono un indirizzo del destinatario e anche l'indirizzo del mittente. Nel protocollo di base non abbiamo nulla che garantisca che il mittente sia quello lì scritto, quindi un malintenzionato può mandare pacchetti con indirizzo sorgente diverso da quello reale.

Esistono diverse ragioni per cui qualcuno possa fare ciò, anzitutto un'identità falsa permette di ostacolare le azioni per rintracciare la sorgente di un attacco, oppure si potrebbe accedere a comunicazioni ristrette soltanto a determinati

utenti. Un altro scenario è quello per cui il malintenzionato invia tanti messaggi verso B, montando un attacco di negazione di servizio (Denial of Service, **Dos**) contro B basato sull'amplificazione del traffico generato da C.



Denial of Service, DoS

Gli aggressori rendono una rete, un host o un altro elemento infrastrutturale non disponibili per gli utenti legittimi. Esistono 3 categorie di attacchi Dos:

-**Attacchi alla vulnerabilità dei sistemi**: sapendo a priori la vulnerabilità dell'host invio un numero esigui di pacchetti costruiti ad arte per causare il vlocco di un servizio o lo spegnimento dell'host (si sfruttano vulnerabilità di applicazioni o sistemi operativi).

-**Bandwidth flooding** (inondazione di banda): Se un host ha una velocità di accesso R_s , se sono in grado di sostenere un traffico prossimo a R_s per un tempo sufficiente posso bloccare il traffico legittimo, rendendolo inaccessibile. Invio quindi in pratica un numero massivo di pacchetti all'host obiettivo.

-**Connection flooding** (inondazione di connessioni): stabilisco un gran numero di connessioni TCP con l'host obiettivo, impedendogli di accettare le connessioni legittime.

Nel caso del bandwidth flooding, un singolo client ha R_c molto inferiore rispetto alla R_s del server, ed inoltre sarebbe molto facile da identificare e bloccare. Per questa ragione gli aggressori organizzano tendenzialmente l'attacco in modo **distribuito (Distributed denial of service, DDoS)**.

I malintenzionati, dopo aver selezionato l'obiettivo, irrompono in n host diversi attraverso la rete, inserendo in essi dei malware che li inseriscono in una **botnet**, cioè una rete di macchine “zombie” (infettate) che a comando possono essere istruite per inviare pacchetti verso l'obiettivo.



È più difficile da identificare il fautore dell’attacco ed è più difficile da bloccare. È possibile far comunque fronte a questi attacchi tramite l’implementazione di:

Linee di difesa

-**Autenticazione**: dimostrare che siamo chi diciamo di essere. Inserisco nel pacchetto una dimostrazione del fatto che i pacchetti che invio vengono effettivamente da me. Al livello di rete cellulare questa garanzia l’abbiamo tramite un’identità hardware attraverso la carta SIM; in internet tradizionale non esiste un’assistenza hardware di questo tipo.

-**Riservatezza**: attraverso la cifratura; quando arrivano le informazioni in broadcast la crittografia ci permette di rendere i messaggi illeggibili a chi non possiede la chiave di decifrazione.

-**Integrità**: va a braccetto con l'autenticazione; non solo garantisco che i pacchetti li ho inviati io ma in più faccio in modo di farti capire se il pacchetto è stato manomesso o comunque contraffatto.

-**Restrizioni di accesso**: in uno scenario come ethernet condivisa, basta attaccarsi e si è dentro la rete. Tramite VPN si possono usare strumenti software o hardware possiamo impedire l'accesso alla rete se non a chi è autorizzato tramite credenziali.

-**Firewalls**: middlebox che si occupano di filtrare il traffico. Si tratta in generale di una funzione che può essere implementata in router, integrata nei client o implementata tramite applicazioni:

- Off by default: filtrare i pacchetti in entrata per limitare i mittenti, i destinatari e le applicazioni
- Rilevare/reagire agli attacchi dos
- Protezione da IP spoofing (es. impedire l'ingresso in una LAN di pacchetti provenienti da altre reti ma il cui mittente dichiarato appartiene alla LAN)
- Impedire connessioni a applicazioni etc...

Livelli di protocollo e modelli di riferimento

Le reti sono complesse, con molti “pezzi”:

Host, Router, svariate tipologie di Mezzi Trasmissivi, Applicazioni, Protocolli, Hardware e Software.

Esiste un modo di organizzare l'architettura delle reti e/o la nostra trattazione sulle reti? (capire cosa sta accadendo?) Sì.

Facciamo riferimento ad un esempio più concreto, *l'organizzazione di un viaggio aereo*, il trasferimento da punto a punto con persone che portano dei bagagli. Un viaggio di questo tipo è descrivibile tramite una sequenza di servizi che consumiamo.

Per prima cosa ci rechiamo all'aeroporto di partenza, dove ci rechiamo in biglietteria per acquistare il biglietto. Con il biglietto preso ci rechiamo al check-in per registrare il bagaglio per poi, dopo aver superato i controlli di sicurezza, arriviamo al gate dove entriamo nell'aereo che, tramite la pista di decollo prende il volo e, sotto il controllo dell'instradamento aereo, arriva a destinazione. L'aereo atterra così sulla pista d'atterraggio, si esce dal gate, si ritirano i bagagli e se si è fatto ritardo si va in biglietteria per protestare.

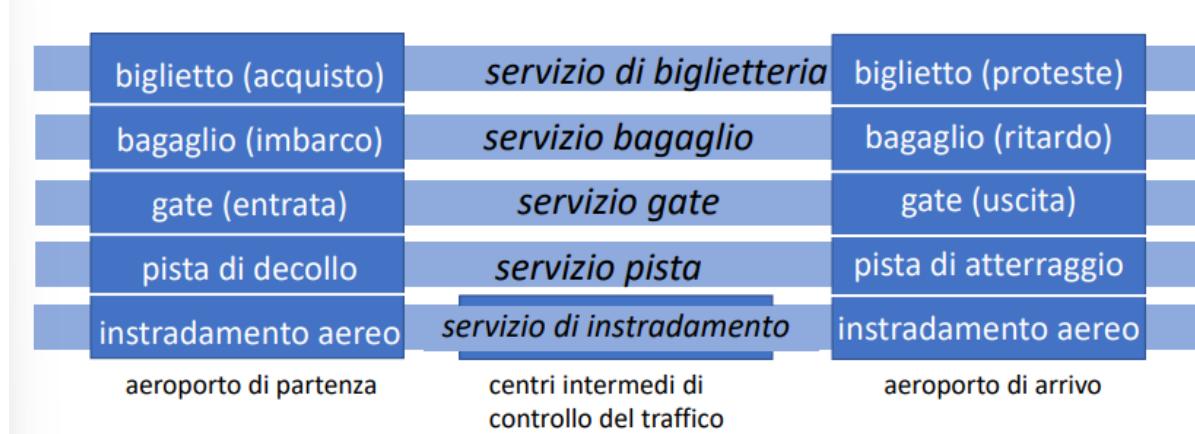
Nell'immagine si vede come si utilizzano i servizi in un ordine ben preciso: dall'alto verso il basso ad un'estremità (nella città di partenza) e dal basso verso l'alto una volta arrivati nella città di arrivo.



Nelle due estremità trovo la stessa funzione implementata in maniera distribuita: instradamento, pista per partire o atterrare, gate per entrare o uscire, bagaglio e biglietteria. Ciò che definisce un **livello (layer)** è il fatto che implementa un servizio:

-effettuando determinate azioni all'interno del livello

-**utilizzando i servizi del livello immediatamente inferiore** (ad esempio il servizio di gate è in funzione della pista: il gate d'ingresso viene utilizzato per arrivare alla pista di decollo e il gate d'uscita per uscire dalla pista, ma perché il servizio di pista funziona? Grazie al servizio d'instradamento aereo (livello immediatamente inferiore).



La **stratificazione** è utile perché *permette un approccio* alla progettazione/discussione di sistemi complessi come l'Internet. In particolare permette un **approccio al sistema** in termini di parti che lo compongono e di **interazioni** tra di essi, limitando tra l'altro le possibili interazioni (ho interazioni solo tra livelli adiacenti).

Ciò porta anche ad una **modularizzazione** che facilita la manutenzione e l'aggiornamento del sistema, perché posso intervenire nell'implementazione di un livello fintanto che non cambio il servizio offerto al livello superiore, e quindi le modifiche ad una procedura non devono influire sul resto del sistema.

Si farà riferimento all'architettura di Internet come **Pila di protocolli (Protocol Stack)** di internet, dove si fa riferimento al fatto che **ogni livello implementa determinati protocolli**.

Pila di protocolli (Protocol Stack) di Internet

In internet si hanno **5 livelli**, dall'alto verso il basso:

- **Applicazione (application layer)**: ospita le applicazioni di rete, per cui questo livello contiene i protocolli per cui i processi applicativi dialogano tra di loro (HTTP, IMAP, SMTP, DNS etc...);
- **Trasporto (transport layer)**: si occupa del trasferimento dati tra processi (in esecuzione su host differenti). Il livello di trasporto di internet fornisce alle applicazioni un’interfaccia (interfaccia socket) che permette a un processo in un host di comunicare con un altro. (TCP, UDP, trasferimento affidabile a connessione e inaffidabile senza connessione)
- **Rete (network layer)**: trasferimento di pacchetti di rete (detti datagrammi) da un host a un altro (IP, protocolli di instradamento per determinare le tabelle di instradamento)
- **Collegamento (link layer)**: trasferimento di dati tra elementi di rete vicini. La differenza con il livello di rete è che in quel caso avevo i due host ben definiti e nel mezzo tutto internet, il livello di collegamento è limitato ad elementi vicini (ad esempio trasferimento di pacchetti tra router vicini nell’instradamento) (Ethernet, PPP, 802.11 (WiFi))
- **Fisico (physical layer)** si occupa dell’effettiva trasmissione del bit sul “filo” generando e/o interpretando i segnali fisici.

A livello di applicazione ho processi che comunicano con un protocollo applicativo scambiandosi messaggi. A livello di trasporto ho processi che trasferiscono dati incuranti di ciò che contengono. A livello di rete io faccio comunicare gli host, non i processi



Vediamo che succede più nel dettaglio.

Le applicazioni (**livello di applicazione**) comunicano tramite messaggi usando il servizio di trasporto; il protocollo del livello di trasporto **incapsula** il messaggio del livello applicativo, **M**, con un **header Ht** di livello di trasporto in un'unità più grande definita **segmento**, che viene trasferita all'altra estremità del livello di trasporto. (*Hn usato dal protocollo a livello di trasporto per implementare il proprio servizio, aggiungendo un'intestazione a livello di trasporto*).

Ma come fa il livello di trasporto a far arrivare il segmento all'altra estremità? Tramite l'uso del **livello di rete**. Il segmento viene passato al livello di rete ed il protocollo del livello di rete **incapsula** il segmento del livello di trasporto [**Ht | M**] con un header a livello di rete **Hn**, per creare un **datagramma** a livello di rete. (*Ht usato dal protocollo a livello di rete per implementare il proprio servizio, aggiungendo un'intestazione a livello di rete*).

Tramite l'uso del **livello di collegamento** il datagramma di rete [**Hn | [Ht | M]**] viene incapsulato con un header di collegamento **Hl** per creare un **frame** a livello di collegamento (unità più grande ancora, intestazione a livello di collegamento etc...), si vorrà poi inviare il frame al router più vicino

Per farlo si arriva quindi al livello più in basso, il **livello fisico**, che **trasmette i bit del frame** [**Hl | [Hn | [Ht | M]]**] attraverso il mezzo trasmissivo.

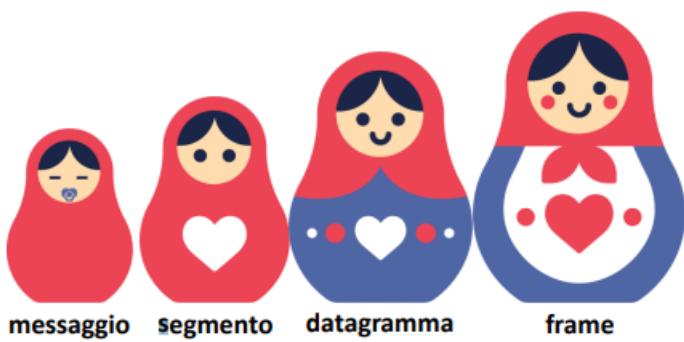
Modello di Servizio (Service Model)

Un livello implementa dei protocolli che possono implementare diversi servizi. Il complesso di servizi implementato da un livello è detto **Modello di Servizio**.

Protocolli diversi possono implementare servizi diversi, es. il livello di collegamento può offrire servizi diversi (Ethernet, WiFi, PPP).

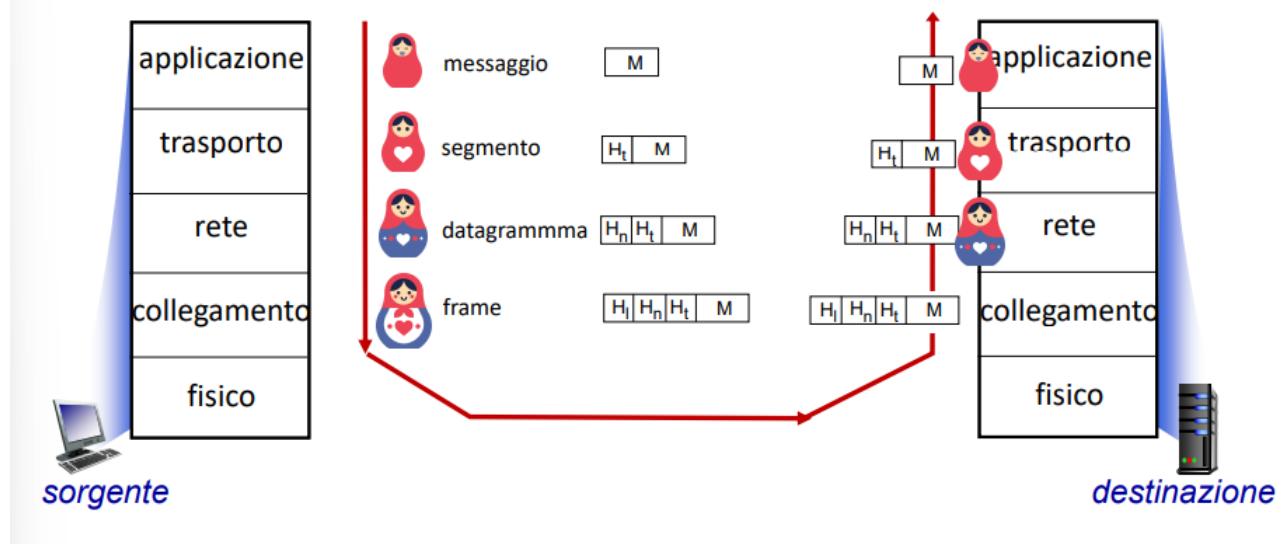
Inoltre a parità di protocollo di collegamento (es. Ethernet) potrei avere a livello fisico mezzi diversi che mi obbligano ad usare protocolli diversi (es. ethernet su fibra ottica userà protocollo diverso da ethernet su doppino intrecciato etc...).

Per capire l'incapsulamento si può pensare al processo come a una Matrioska:



Si parte dal messaggio che viene inserito in “Matrioske” sempre più grandi. Giunte a destinazione riapro le Matrioske ritornando al messaggio originale.

Servizi, Stratificazione e Incapsulamento



Frame --- Datagramma --- Segmento --- Messaggio (quando estraggo)

Man mano che incapsulo aggiungo intestazione, per cui quando arrivo a destinazione per prima cosa tolgo l'intestazione di collegamento, poi di rete, poi di trasporto per poi arrivare al messaggio vero e proprio.

(come se impilassi in uno stack nella sorgente e rimuovessi dallo stack a destinazione)

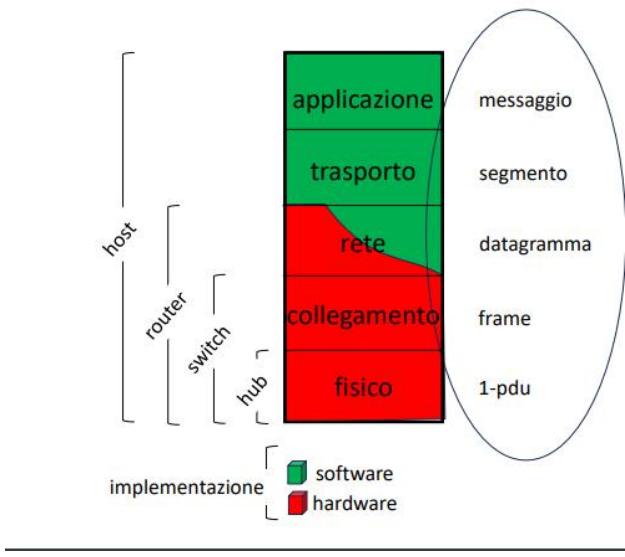
Finora abbiamo visto il funzionamento della Pila di protocolli di internet per due host adiacenti. Vediamo ora nel dettaglio cosa succede con uno scenario end to end con un percorso più lungo.

Incapsulamento: una visione end to end

Il messaggio scende lungo la pila protocollare dell'host dove saranno aggiunte intestazioni, il messaggio incapsulato viene sparato dalla scheda di rete dell'host sorgente verso uno switch che lo inoltra al router. Il router riceve il frame con le intestazioni e, dopo una serie di controlli, il livello di collegamento consegnerà al livello di rete del router il datagramma (salvo di livello, da frame a datagramma). Nel datagramma sono presenti varie informazioni, tra cui l'indirizzo del destinatario. *Il livello di rete del router analizza l'intestazione del datagramma e capisce in che uscita inviare il messaggio tramite la funzione di inoltro.* A questo punto il livello di rete passa il messaggio al livello di collegamento che aggiunge nuovamente le intestazioni di collegamento e lo spara etc... fino ad arrivare a destinazione, dove il messaggio viene decapsulato.

I vari livelli sono quindi implementati in maniera distribuita tra più nodi, non tutti i nodi implementano tutti i livelli ma ne implementano sottinsiemi in funzione delle loro capacità.

Si parte dal livello fisico andando verso l'alto, tipicamente si utilizza un'implementazione hardware (nella scheda di rete, che connette l'host al collegamento) dalla rete in giù e software (es. algoritmi di instradamento) dalla rete in su.



Nel momento in cui il datagramma viene inoltrato l'intestazione può cambiare. Quando abbiamo parlato ad esempio di traceroute viene decrementato il timetolive di 1, per cui l'intestazione sarà diversa.

Come si vede in figura sopra; **host** implementano tutti i livelli, **router** fino alla rete, **switch** fino a collegamento, **hub** (che possiamo immaginare come giunzione a livello fisico che si limita a propagare segnali) solo livello fisico.

Messaggio, segmento, datagramma e frame sono detti in generale **n-PDU** (**Protocol Data Unit**), rappresentano la singola unità di informazione scambiata tra pari attraverso un protocollo ad un livello n. Tipicamente una PDU è formata dall'informazione specifica per il protocollo, che fa parte dell'intestazione, e ha un carico utile (**payload**) che è la payload del livello superiore.

Modello di riferimento ISO/OSI

Un altro modello simile allo stack di Internet ma che prevede due livelli in più: **presentazione** e **sessione**.

Il livello di **presentazione** si occupa *dell'interpretazione dei dati*, in particolare aspetti quali crittografia, compressione etc...

Il livello di **sessione** si occupa della sincronizzazione, checkpointing, ripristino dello scambio di dati.

Questi servizi, se *necessari*, devono essere implementati nelle applicazioni, poiché Internet “manca” nella pila di questi due livelli. Ma sono necessari? Lo vedremo successivamente.

Wireshark

Programma opensource per il packet sniffing. Come funziona? Nell’host il programma usa un’interfaccia detta **pcap** (packet capture) per farsi dare dal livello di collegamento una copia di tutti i frame Ethernet inviati/rivevuti. Il sistema operativo si occupa infatti di gestire i livelli fino a quello di trasporto, mentre poi le applicazioni sono i processi in esecuzione sul sistema operativo. Wireshark non utilizza la tipica interfaccia dei protocolli di trasporto, ma un’altra interfaccia per farsi dare le copie dei frame.

(Storia da 1:07:00 audio Lez 3)

Lez. 4

Livello di Applicazione

Il livello di applicazione è dove risiedono le applicazioni di rete. L’obiettivo è capire a livello concettuale-applicativo gli aspetti salienti dei protocollo di applicazioni di rete, guardando ciò di cui abbiamo bisogno a livello di trasporto e come conviene architettare le applicazioni. Faremo ciò riferendoci ad alcuni protocolli di applicazioni note.

Alcune diffuse applicazioni in rete: posta elettronica, trasferimento file, Web, Forum, Social Media, videogiochi online, messaggistica istantanea, streaming

video (esiste in due forme: video-clip memorizzati (es. youtube) oppure streaming vera e propria (vedo il video man mano che lo scarico)), telefonia via Internet (voice-over-IP, VoIP, es. Skype), shell o desktop remoto etc..

Scrivere un'applicazione in rete significa *scrivere un'applicazione in esecuzione sui sistemi periferici*, il nucleo della rete non esegue applicazioni in rete e non scriviamo l'applicazione su un singolo sistema periferico, ma su più sistemi. I programmi che girano su queste macchine remote dovranno comunicare tra loro, tipicamente utilizzando i servizi del livello di trasporto in Internet.

Es. il Browser Web deve comunicare con il server Web per ottenere la pagina di suo interesse.

Esistono due principali paradigmi per architettare le applicazioni (per quel che riguarda, in particolare, lo schema di comunicazione): **client-server** e **peer-to-peer**.

Paradigma Client-Server:

Ho un **host sempre attivo** (il **server**) che ha un **IP fisso** (IP == Indirizzo in Internet) e che è spesso ospitato in un Data Center o moltiplicato in un cluster per motivi di scalabilità. *Il server deve essere sempre attivo poiché è colui che eroga il servizio, fornisce le informazioni.*

Il **client** è invece un *host che contatta il server per ottenere le informazioni utili al funzionamento del servizio*. Non deve essere sempre attivo, e per questo i client possono avere indirizzi **IP dinamici**.

Nell'architettura client-server i client non comunicano mai tra di loro direttamente, ogni comunicazione deve essere sempre mediata dal server.

Stiamo considerando uno scenario del tipo “**interazione richiesta-risposta**”, ma esistono anche scenari di tipo **Push** in cui dopo esserci registrati nel server è quest’ultimo che ci chiede in maniera proattiva le informazioni.

Paradigma Peer-to-Peer (tra pari, P2P):

Non abbiamo l’host server sempre attivo, ma più host (**peer**) comunicanti direttamente tra loro. Questa architettura ha **scalabilità intrinseca**, perché se aggiungo un peer questo da una parte chiede un servizio (aggiunge carico al sistema), ma dall’altra può esso stesso fornire ulteriori servizi per altri peer aggiungendo capacità di servizio.

I peer non sono fissi, possono presentare quindi **IP dinamico**, ciò comporta una certa difficoltà di gestione.

Finora abbiamo parlato di host, ma ciò che ci interessa davvero sono i processi all’interno di essi (sono loro che avranno il ruolo di server o client).

Un **processo** rappresenta un programma in esecuzione su un host, *il processo client è colui che dà inizio alla comunicazione* mentre *il processo server è colui che attende di essere contattato*.

In una sessione di comunicazione un processo svolge sempre uno dei due ruoli: o è client o è server. In un’architettura peer-to-peer tuttavia uno stesso processo può svolgere i due ruoli in sessioni differenti (un peer può essere client di un altro peer ma server per un altro ancora).

I processi comunicanti scambiano tra loro informazioni tramite Messaggi, che sono scambiati tramite il livello di trasporto di Internet (forma di inter-process-comunication remota)

Nella parte di sistemi si è vista una serie di inter-process-communication mechanism relegato a processi locali nella macchina (pipe, segnali etc..).

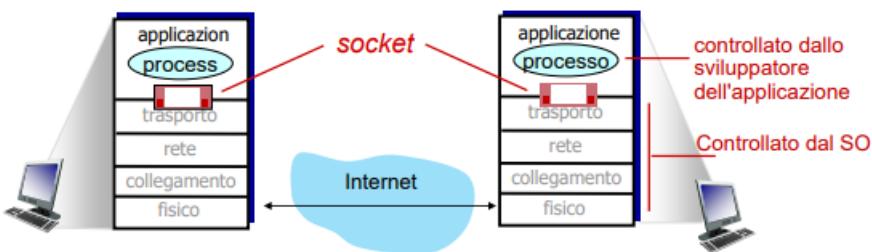
Come fanno le applicazioni ad utilizzare il livello di trasporto di Internet?
Si necessita di un interfaccia, definita **Interfaccia Socket**.

Socket

Ogni processo ha una o più socket che possiamo immaginare come una **porta** attraverso la quale il processo può far uscire messaggi e (presupponendo che vi sia una infrastruttura idonea allo scopo) farli entrare nella porta del processo di destinazione.

L’interfaccia delle socket ci permette quindi di richiedere il servizio di trasferimento, una volta però incaricato il livello di trasporto di inviare dati **ne perdo il controllo!** Per questo è necessario scegliere accuratamente il livello di trasporto per non avere problemi.

Si ricorda ancora una volta che i livelli inferiori della pila protocollare di Internet sono controllati dal Sistema Operativo, mentre il livello di applicazione è controllato da noi sviluppatori. La socket si frappone esattamente tra i due.



Quando inviamo una lettera a qualcuno, dopo averla scritta e inserita nella busta scriviamo l’indirizzo del ricevente ed eventualmente anche quello del mittente (per avere una risposta). Vale lo stesso per la comunicazione tra host via Internet, ma cosa rappresenta l’indirizzo di un processo comunicante su Internet? Si parla di **identificatore**, ogni processo ne ha uno (necessario per ricevere messaggi). L’identificatore comprende sia *l’Indirizzo IP* (univoco a 32 bit, per l’host) che un **numero di porta** (port number) associato al processo in esecuzione nell’host.

Non è sufficiente conoscere l’indirizzo IP dell’host su cui è in esecuzione il processo per identificare il processo stesso, poiché sullo stesso host possono essere in esecuzione molti processi contemporaneamente!

Per la scelta dei numeri di porta si ha una certa libertà di movimento, anche se numeri di porta sotto il 1024 riguardano solamente processi con privilegi di root (dette porte well-known, dovrebbero essere assegnate ad applicazioni di rete ben note).

Esempio: http server: 80; mail server: 25.

Per inviare un messaggio HTTP al server gaia.cs.umass.edu: IP 128.119.245.12 e numero di porta 80.

Ogni volta che ho entità remote che comunicano necessito di un protocollo che governi la comunicazione. Abbiamo già visto in generale cosa definisce un protocollo, caliamo la definizione generale sul livello di applicazione.

Un protocollo a livello di applicazione definisce:

Tipo di messaggio scambiato (richiesta, risposta), **Sintassi dei messaggi** (quali sono i campi nel messaggio e come sono descritti), **Semantica dei messaggi** (significato delle informazioni nei campi), **Regole** (un protocollo definisce delle regole che disciplinano quando e come inviare un messaggio e cosa devo fare nel momento in cui voglio rispondere).

Chiunque può implementare un protocollo in rete (se voglio fare un'applicazione devo definire come i processi comunicano tra loro, quindi definire un protocollo). Se definisco un protocollo “segreto”, *non permettendo ad altri di implementarlo*, allora il protocollo è definito come **proprietario**.

La maggioranza di applicazioni note tuttavia (specie quelle open-source) sfruttano **protocolli di pubblico dominio** che possono quindi essere implementate liberamente.

Nel caso delle applicazioni di rete protocolli di pubblico dominio sono spesso definiti tramite **RFC** (request for comments), standardizzati dall'ente **IETF** (internet engineering task force).

I processi comunicano attraverso il livello di trasporto. Internet fornisce due protocolli di trasporto: **TCP** e **UDP**. Ma quale devo scegliere? Prima di rispondere alla domanda vediamo *quali sono i tipici requisiti che un'applicazione può avere nei confronti di un livello di trasporto*:

-**Perdita di dati**: la mia applicazione può tollerare la perdita di dati? (tipicamente applicazioni multimediali (audio) possono tollerare una certa perdita di dati, ma altre come trasferimento file, transazioni web richiedono un trasferimento 100% affidabile).

-**Sensibilità rispetto al Fattore Tempo**: alcune applicazioni come telefonia via Internet o giochi online richiedono che i ritardi siano bassi per essere “efficaci”.

- Throughput**: il throughput è la frequenza di arrivo o trasferimento dei dati. Alcune applicazioni (dette *elastiche*) sono adattabili a throughput diversi (es. scarico file), altre (dette *sensibili alla banda*, ad esempio quelle multimediali) per essere efficaci richiedono un'ampiezza di banda (throughput) minimo.
- Sicurezza**: cifratura, integrità dei dati etc... (qualcuno potrebbe avere interesse ad attaccare la mia applicazione?).

Requisiti del servizio di trasporto di alcune applicazioni comuni

<u>applicazione</u>	<u>tolleranza alla perdita di dati</u>	<u>throughput</u>	<u>sensibilità al fattore tempo</u>
trasferimento file	no	variabile	no
posta elettronica	no	variabile	no
documenti Web	no	variabile	no
audio/video in tempo reale	sì	audio: 5kbps-1Mbps video:10kbps-5Mbps	sì, centinaia di ms
streaming audio/video memorizzati	Sì	come sopra	sì, pochi secondi
giochi interattivi	sì	fino a pochi kbps	sì, centinaia di ms
messaggistica istantanea	no	variabile	sì e no

Application Layer: 2-13

Messaggistica istantanea sì e no per sensibilità al tempo poiché ad esempio nel parlare è un requisito, ma nell'inviare messaggi non è eccessivamente stringente.

Servizi dei protocolli di trasporto in Internet

TCP

-**Trasporto affidabile**: fra processi di invio e ricezione, dati consegnati senza errore, perdite (dati corrotti) e nell'ordine di invio.

-**Controllo di flusso**: il mittente non vuole sovraccaricare il destinatario (se ho una macchina molto potente e dall'altra parte ho una macchina più lenta e invio troppi dati, il destinatario se non fa in tempo a caricarli e potrebbe avere delle perdite). Il controllo di flusso permette di regolare la *velocità di invio in*

funzione della capacità del destinatario di processare i dati.

-**Controllo della congestione**: agisce sempre sulla velocità di invio del mittente ma lo fa *in funzione non del destinatario, ma della congestione generale della rete* (“strozza” il processo d’invio quando la rete è sovraccarica).

-**Orientato alla connessione**: prima che la comunicazione possa avere luogo, è necessario che le parti stabiliscano una connessione tramite un’operazione detta “*hand-shaking*” nella quali vengano inizializzate delle variabili interne al livello e siano scambiate delle informazioni di controllo. Non interessa la rete (non viene riservato nulla in essa) ma solo i pari che devono interconnettersi e non va confusa con il setup delle reti a commutazione a circuito (che lavora in modo diverso, riservando spazio nella rete).

In poche parole quindi TCP offre setup della connessione.

-**Non offre**: temporizzazione, garanzie su un’ampiezza di banda minima (throughput), sicurezza.

UDP

-**Trasporto di dati inaffidabile**: posso inviare dati e non ho garanzia che vengano ricevuti.

-**Non offre**: affidabilità, controllo di flusso, controllo della congestione, temporizzazione, ampiezza di banda minima (throughput), sicurezza, setup della connessione.

Ma quindi a che serve UDP? Si tratta di un protocollo che *offre il minimo indispensabile per trasformare protocolli IP (di rete) in protocolli di trasporto*. *Non avere connessione, sicurezza, controllo del flusso etc.. permette un maggiore controllo delle applicazioni sulla temporizzazione e invio dei messaggi*. (ritardi più brevi possibile).

Inoltre UDP permette anche di “barare” perché, mentre TCP preserva la congestione, UDP se ne disinteressa ed inoltra comunque i dati.

Applicazioni Internet: protocollo a livello applicazione e protocollo di trasporto

applicazione	protocollo a livello applicazione	Protocollo di trasporto sottostante
trasferimento file	FTP [RFC 959]	TCP
posta elettronica	SMTP [RFC 5321]	TCP
documenti web	HTTP [RFC 7230, 9110]	TCP
telefonia via Internet	SIP [RFC 3261], RTP [RFC 3550], o proprietario	TCP o UDP
streaming audio/video	HTTP [RFC 7230], DASH	TCP
giochi interattivi	WOW, FPS (proprietario)	UDP o TCP

Tipicamente dove ho bisogno di affidabilità TCP, dove posso avere perdita UDP.

Tuttavia UDP è spesso associato a TCP perché i firewall possono essere configurati per bloccare UDP, per cui quando succede l'applicazione converte UDP in TCP e il gioco è fatto.

Per lo streaming video uso la tecnica del buffering (attendo qualche secondo per poter vedere di più successivamente).

Né UDP né TCP sono nativamente sicuri, infatti questi protocolli sono stati standardizzati molti decenni fa, in quegli anni ancora ci si trovava in una visione ideale di un Internet sicuro (fintanto che non è stato utilizzato a scopi commerciali, quando era solo ente di ricerca).

Non ho alcuna cifratura e le password sono inviate in chiaro.

Per aggiungere sicurezza si utilizzano i **TLS** (*transport layer security*), si tratta di uno standard implementato a livello di applicazione sotto forma di libreria. Sostanzialmente creano un socket virtuale sopra il socket TCP, per cui io metto i dati nella socket TLS e questa me li cifra inviando il messaggio cifrato in chiaro tramite TCP (non importa che via TCP l'inoltro del messaggio sia in chiaro, chi lo intercetta ha tra le mani un messaggio cifrato). Quando TCP lo porta a destinazione TLS lo decifra.

Web e HTTP

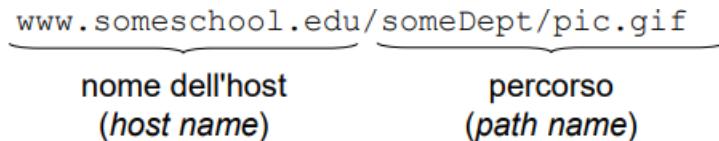
Il Web è un media **ipertestuale**, ma a differenza di altri ipertesti (es. encyclopedie su disco) è globale e può essere ospitato da un server. Il Web è

fatto da varie componenti (Browser, server etc...) in vari formati, quello che noi vedremo (HTTP) è solo uno degli ingredienti (ci serve molto altro per fare il Web: linguaggi di rappresentazione, software, client, server etc...).

HTTP è inoltre usato come base di protocolli di molte altre applicazioni, non solo il Web. Di fatto quindi HTTP è protocollo di comunicazione sul Web, ma non è il suo unico ruolo.

Nel Web si ha a che fare con **pagine web**, ognuna di esse è costituita da **oggetti**. Ogni oggetto può essere file HTML, immagine JPEG, script Javascript, foglio di stile CSS, file audio etc...

Una pagina Web è formata da un **file HTML di base** che include diversi oggetti referenziati, ognuno di essi referenziato ad un **URL** del tipo:



L'host name rappresenta la macchina a cui chiederemo l'informazione, il percorso identifica (in uno spazio di dati gerarchico) l'oggetto in questione.

HTTP (hypertext transfer protocol) è un protocollo di livello applicazione che usa il modello **client-server** dove il client tipicamente è il **Browser** che richiede di visualizzare oggetti mentre il server è colui che li fornisce.

Client diversi possono effettuare richieste a un server nello stesso momento. Uno dei vantaggi del Web, per come sono definiti i suoi standard è che è **Platform Independent**, non solo è in grado di far dialogare host diversi ma anche macchine intrinsecamente diverse (Linux, Windows, Mac etc..).



HTTP usa TCP perché si tratta di un protocollo di trasferimento di informazione, e in quanto tale non tollera perdite.

Fasi di funzionamento per HTTP:

- 1. Il client inizializza la connessione TCP (crea una socket) con il server, sulla porta 80*
- 2. Il server accetta la connessione TCP dal client*
- 3. Messaggi HTTP (messaggi di un protocollo di applicazione) scambiati tra browser (cliente HTTP) e server Web (server HTTP)*
- 4. Connessione chiusa TCP*

In HTTP 1.0 abbiamo **connessioni non persistenti**, cioè una connessione diversa per ogni oggetto.

HTTP è anche detto protocollo **stateless**, cioè “*senza stato*”. *Infatti il server non mantiene informazioni dettagliate sulle richieste fatte dal client*. Ciò non vuol dire che una richiesta non possa modificare lo stato del server (esistono richieste che possono inviare dati al server), *ma che se faccio due richieste al server la seconda richiesta non è fatta in funzione della richiesta precedente!*

Ma perché non mantenere lo stato? Poiché **protocolli che mantengano lo stato sono complessi!** Anzitutto è richiesto di memorizzare lo stato passato (troppi stati da immagazzinare in toto), ma ancora più grave se uno dei due si blocca/crasha i due potrebbero non avere più una “visione” comune dello stato e ci dovrebbe essere una riconciliazione...

HTTP 1.0 seguiva i seguenti passaggi secondo la logica di “**connessione non persistente**”: connessione TCP aperta; almeno un oggetto trasmesso su una connessione TCP e infine connessione TCP chiusa. A ciò conseguiva che lo scaricamento di oggetti multipli richiedeva connessioni multiple!

Con HTTP 1.1 vengono introdotte invece **connessioni persistenti**, per cui i passaggi sono: connessione TCP aperta, *più oggetti possono essere trasmessi su una singola connessione* TCP tra client e server e infine connessione chiusa.

RTT == *tempo impiegato da un piccolo pacchetto per andare dal client al server e tornare dal client (include ritardi di elaborazione, accodamento, propagazione)*

Vediamo il vantaggio della connessione persistente rispetto alla non persistente osservando una problematica di quest'ultima. Quando devo aprire una connessione TCP devo fare un handshake a 3 vie (3 messaggi: richiesta connessione, accettazione e il terzo messaggio necessario a veicolare i messaggi applicativi (lo vedremo meglio più avanti)). *Quindi solo dopo 2 roundtrip time RTT il server può trasferirmi i dati!* (più altri roundtrip time per ogni singola richiesta di messaggio).

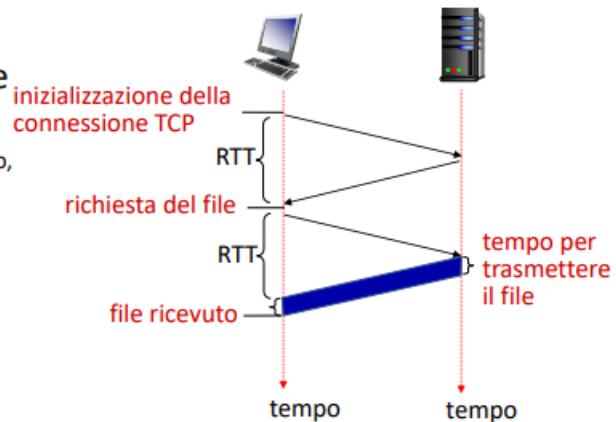
Con una connessione non persistente impiego sempre del tempo per stabilire la connessione, per fare la richiesta e poi fare la trasmissione effettiva.

In generale, per una connessione non persistente, il tempo di risposta è **2 RTT + tempo trasmissione file**.

RTT (definizione): tempo impiegato da un piccolo pacchetto per andare dal client al server e ritornare al client (include ritardi di elaborazione, accodamento, propagazione)

Tempo di risposta (per oggetto):

- un RTT per inizializzare la connessione TCP
- un RTT perché ritornino la richiesta HTTP e i primi byte della risposta HTTP
- tempo di trasmissione del file/oggetto



Tempo di risposta con connessioni non persistenti = 2RTT+ tempo di trasmissione del file

La richiesta non persistente ha quindi lo svantaggio di spendere **2 RTT** per ogni oggetto, avere un **overhead** del sistema operativo per ogni connessione TCP, far sì che i Browser aprano spesso connessioni TCP parallele per caricare gli oggetti referenziati.

Con la connessione persistente invece *il server lascia la connessione TCP aperta dopo l'invio di una risposta, permettendo ai successivi messaggi tra gli stessi client e server di essere trasmessi sempre sulla stessa connessione aperta. Inoltre il client invierà le richieste non appena incontra un oggetto referenziato e un solo RTT per tutti gli oggetti referenziati!*

Messaggio di richiesta HTTP

Si tratta di un protocollo testuale, la richiesta è formata come segue:

riga di richiesta (*request line*) → GET /index.html HTTP/1.1\r\n
 (comandi GET, POST, HEAD) carattere di nuova linea (*line-feed*)

righe di intestazione (*header lines*) →
 Host: www-net.cs.umass.edu\r\n
 User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:80.0) Gecko/20100101 Firefox/80.0 \r\n
 Accept: text/html,application/xhtml+xml\r\n
 Accept-Language: en-us,en;q=0.5\r\n
 Accept-Encoding: gzip,deflate\r\n
 Connection: keep-alive\r\n
 \r\n

Un carriage return e un, line feed all'inizio della linea indicano la fine delle righe di intestazione

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Application Layer: 2-26

\r carattere di ritorno a capo (carriage return)

\n carattere di nuova linea (line feed) (sempre prima r di n)

Le header lines sono della forma di coppie header-valore. È lì che troviamo l'indirizzo dell'host (header host, valore). User Agent (chi fa la richiesta).

Accept: come “accetto” determinate specifiche (es. accetto la lingua inglese-us), per l'encoding si fa riferimento al fatto che file di testo sono facilmente comprimibili e vengono inviati compressi per non sprecare banda.

Connection: keep-alive serve al client per dire al server di mantenere la connessione aperta (connessione persistente). Il server capisce che una richiesta è finita quando trova \r\n.

Alcuni campi di intestazione nei messaggi di richiesta:

Host

- hostname e numero di porta (se assente, si assume 80 per HTTP e 443 per HTTPS) del server al quale sarà inviata la richiesta. Obbligatorio in HTTP/1.1; se assente, il server può rispondere con un 400 Bad Request.

User-Agent

- Identifica l'applicazione, il sistema operativo, il vendor e/o la versione dello user agent che sta effettuando la richiesta

Accept

- Tipi di contenuto, espressi come media type, compresi dal client

Accept-Language

- Linguaggi naturali o locale preferiti dal client

Accept-Encoding

- Algoritmi di codifica (tipicamente, la compressione) compresi dal client

Connection

- Controlla se la connessione rimarrà aperta al termine dello scambio richiesta/risposta. Il valore close indica che la connessione sarà chiusa (default in HTTP/1.0); altrimenti, una lista non vuota di nomi di header (in genere solo keep-alive), che saranno rimossi dal primo proxy non trasparente o cache, indica che la connessione rimarrà aperta (default in HTTP/1.1)

Application Layer: 2-27

Messaggi di Richiesta HTTP

GET per ottenere un oggetto.

Nella riga di richiesta posso anche spedire dei dati. Si tratta del metodo **POST**, necessario per quando l'utente deve inserire un input ad esempio in un form di una pagina web.

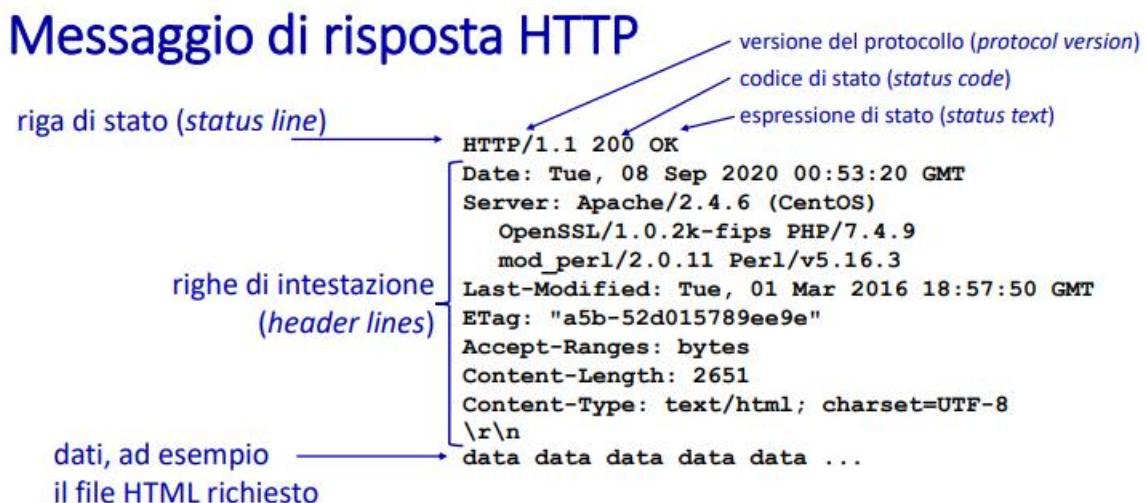
Anche il metodo **GET** può essere usato nei form, scrivendo nell'URL di destinazione i valori del form all'interno di una query string (metto ? e poi coppia-chiave valore separati da &).

`www.somesite.com/animalsearch?monkeys&banana`

Il metodo **HEAD** è come se fosse una GET ma restituisce solo gli header della risposta, e non l'oggetto.

Infine il **PUT** viene utilizzato per creare un nuovo oggetto, quindi se si rimpiazza un oggetto con un altro. Non tutti i server implementano PUT, è disponibile a seconda degli scenari (non posso modificare una pagina web come me pare).

Messaggio di risposta HTTP



Alcuni campi di intestazione nei messaggi di richiesta:

Date

- la data e l'ora in cui il messaggio è stato originato

Server

- descrive il software usato dal server di origine per gestire la richiesta (nota: troppi dettagli possono aiutare i malintenzionati a attaccare il server)

Last-Modified

- la data e l'ora in cui il server di origine crede che l'oggetto sia stato modificato per l'ultima volta

Accept-Ranges

- Indica il supporto del server ai download parziali: il valore, se diverso da none, indica l'unità che si può usare per esprimere l'intervallo richiesto

Content-Length

- lunghezza in byte del corpo dell'entità inviato al ricevente (o che *sarebbe* stato inviato nel caso di un richiesta HEAD)

Content-Type

- *media type* (che indica un formato) del corpo dell'entità inviato al ricevente (o che *sarebbe* stato inviato nel caso di un richiesta HEAD)

Codici di Stato della risposta HTTP

Si trovano nella prima riga del messaggio di risposta dal server al client. Sono definiti da RFC 9110 e sono raggruppati in 5 categorie, **discriminate dalla prima cifra**.

1xx Informational

- una risposta intermedia per comunicare lo stato di connessione o l'avanzamento della richiesta prima di completare l'azione richiesta e inviare una risposta finale (assenti in HTTP/1.0)

2xx Successful

- La richiesta è stata ricevuta con successo, compresa e accettata

3xx Redirect

- Il client deve eseguire ulteriori azioni per soddisfare la richiesta.

4xx Client Error

- La richiesta è sintatticamente scorretta o non può essere soddisfatta

5xx Server Error

- Il server ha fallito nel soddisfare una richiesta apparentemente valida

Application Layer: 2-32

Tra i principali codici di Stato della risposta HTTP:

200 OK

- La richiesta ha avuto successo; l'oggetto richiesto viene inviato nella risposta

301 Moved Permanently

- L'oggetto richiesto è stato trasferito; la nuova posizione è specificata nell'intestazione Location: della risposta

400 Bad Request

- Il messaggio di richiesta non è stato compreso dal server

404 Not Found

- Il documento richiesto non si trova su questo server

406 Not Acceptable

- L'oggetto richiesto non esiste in una forma che soddisfa i vari Accept-*

505 HTTP Version Not Supported

- Il server non ha la versione di protocollo HTTP

Provatate HTTP (lato client)

1. Collegatevi via netcat al vostro server web preferito:

```
% nc -c -v gaia.cs.umass.edu 80 (for Mac)  
>ncat -C gaia.cs.umass.edu 80 (for Windows)
```

- apre una connessione TCP alla porta 80 (porta di default per un server HTTP) dell'host gaia.cs.umass.edu.
- tutto ciò che digitate viene trasmesso alla porta 80 di gaia.cs.umass.edu

2. type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php HTTP/1.1  
Host: gaia.cs.umass.edu
```

- digitando questo (premete due volte il tasto Invio), trasmettete una richiesta GET minima (ma completa) al server HTTP

3. guardate il messaggio di risposta trasmesso dal server HTTP!

(oppure usate Wireshark per guardare la richiesta e la risposta HTTP catturate)

Lez 5

Riassunto

Scrivere un'applicazione di rete significa scrivere programmi che eseguiamo tra sistemi periferici remoti tra loro, sfruttando la rete per veicolare messaggi tra questi host.

Le applicazioni di rete sono eseguite sugli host e non sul nucleo della rete, che viene mantenuto per scelta progettuale più “snello” (l'evoluzione del nucleo è più complessa e sarebbe più difficile sviluppare applicazioni tenendo conto di questo).

Riguardo le applicazioni di rete e i loro protocolli esistono due principali paradigmi: client to server (server fisso, sempre attivo in attesa di richieste da client, i client non comunicano mai tra loro) e peer to peer (scalabilità intrinseca, se aggiungo un host questo richiede informazioni ma è anche in grado di fornirne).

Non si parla tanto di host in quanto client e server quanto di processi client e server. L'interfaccia socket permette la comunicazione tra processi remoti, se voglio comunicare con un processo lo devo identificare (IP per identificare la macchina su cui è in esecuzione il processo e Numero di Porta per identificare il processo in sé).

Un protocollo definisce la struttura dei messaggi scambiati e le azioni che devono essere compiute quando un messaggio viene inviato o ricevuto e

quando succede qualcos'altro (questo “qualcos'altro” tipicamente TimeOut, cioè non ricevo risposta).

Quattro requisiti principali che le applicazioni possono richiedere al servizio di trasporto, e in funzione di questi requisiti si sceglie tra TCP e ODP. Differenza sostanziale sta nell'affidabilità del trasferimento (cioè i dati inviati vengono ricevuti dal destinatario senza perdere dati, senza errori (no dati corrotti) e nell'ordine prestabilito).

La prima applicazione di rete che abbiamo visto è il Web, divenuta popolare alla fine degli anni '90 che ci permette di pubblicare in modo decentralizzato un ipertesto (fluire di informazioni multimediali in modo non lineare).

Ciò che è pubblicato nel Web è definito in ultima istanza come oggetto o **risorsa**, ognuna di esse è identificata da un URL. Questo è costituito da due parti principali: hostname e path. Sappiamo che le macchine sono identificate da IP address, infatti l'hostname che identifica la macchina è convertito in IP grazie al servizio a livello applicazione DNS (Domain Name System).

Il web si basa principalmente sul protocollo HTTP (hypertext transfer protocol), ma non è solo HTTP a definire tutto il Web. Altri standard concorrono alla formazione del Web, tra questi l'URL, linguaggi come HTML e CSS (che definiscono il formato standard noto a tutti i client (user agent) per poter formattare e visualizzare le pagine) etc...

Standard necessari per interoperabilità, cioè capacità di comprendersi anche a partire da host diversi. HTTP protocollo con paradigma client-server (es. browser-web). Server Web famoso è quello Apache HTTP.

Per portare i messaggi da un'estremità all'altra è necessario il protocollo del livello di trasporto. HTTP usa TCP perché ha bisogno di trasferimento affidabile. Per contattare il server web l'user agent ha bisogno dell'ip della macchina e il numero di porta (che non bisogna scrivere nell'url perché di base il server web via http sta in ascolto sulla porta 80).

HTTP stateless perché non viene mantenuta memoria dettagliata delle richieste fatte dai client, ogni richiesta indipendente dalle altre. Ciò semplifica lo sviluppo delle applicazioni, non obbliga il server a mantenere uno stato ed evita necessità di restaurare lo stato se client e server si bloccano.

HTTP via TCP con connessioni non persistenti (close, per ogni richiesta una nuova connessione, RTT roundtrip time) e connessioni persistenti (keep alive, la connessione rimane la stessa in cui si fanno più scambi richiesta-risposta, solo una volta finito viene chiusa la connessione).

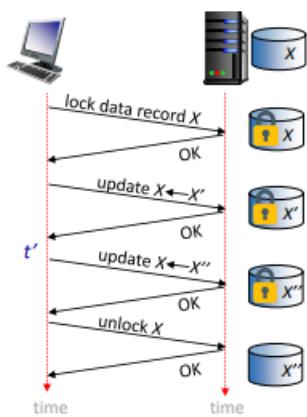
GET ottenere informazioni ma anche inviarne in quantità limitata, POST per inviare più dati

Livello di applicazione (parte 2)

Mantenere stato utente/server: i Cookie

Abbiamo visto che **HTTP è un protocollo stateless**, che si contrappone ad un protocollo con stato del seguente tipo:

un protocollo con stato: il client fa
due modifiche a X, o nessuna



Integrare lo stato per HTTP sarebbe complicato; *cosa succederebbe se la connessione si blocca?* Annulliamo le modifiche? Sarebbe complicato rimettere d'accordo client e server sullo stato precedentemente stabilito, per questo HTTP stateless.

Perciò di base tutte le richieste relative ad HTTP sono gestite in maniera indipendente. Questa tuttavia rappresenta una **limitazione!** Immaginiamo di accedere a gmail, per prima cosa inserisco le mie credenziali e poi accedo alla

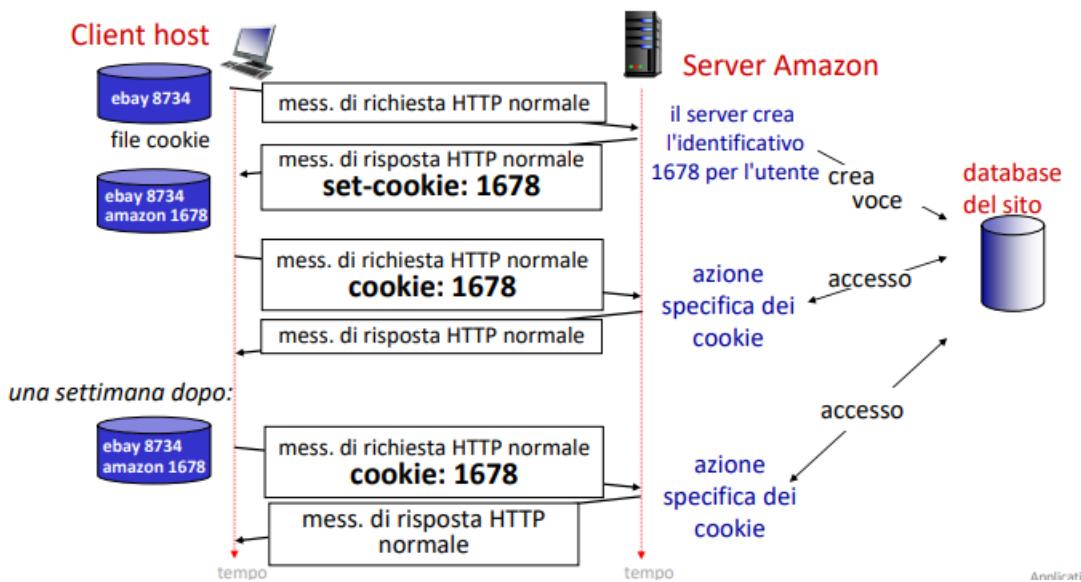
casella di posta. Ma è chiaro che, poiché le richieste successive non mi richiedono di riloggare, è mantenuta una qualche forma di stato (le richieste successive sono eseguite nel contesto del mio utente e quindi in funzione di richieste precedenti!)

Ciò è reso possibile grazie all'aggiunta ad HTTP del meccanismo dei **Cookie**, basato su quattro elementi:

- 1) **Una riga di intestazione nella risposta HTTP** che permette al server di chiedere al client di memorizzare l'informazione di stato cookie
- 2) **Una riga d'intestazione nel messaggio di richiesta** che permette al client di riinviare quel cookie al server
- 3) **Un file** con il quale il client salva il cookie mantenuto sul sistema terminale dell'utente e gestito dal browser dell'utente
- 4) **Un database del server** nel quale quest'ultimo si salva l'associazione tra cookie e le informazioni di stato

Nel caso di Gmail ad esempio il database avrà scritto che quel cookie corrisponde alla nostra utenza e quindi ogni messaggio che si porta dietro quel cookie sarà gestito nel contesto della nostra utenza.

Vediamo ora un esempio di uso di cookie in ambito più generico:



Il client ha già visitato in passato ebay e ha i cookie per quest'ultimo. Ora vuole visitare amazon, poiché sta contattando amazon non include i cookie di ebay e poiché amazon non trova cookie per l'user agent (è la prima volta che accede) ne genera uno, creando un ID univoco per quell'user agent che salva nel suo database associandolo a delle informazioni e che restituisce in un cookie al client. Quando il client effettua una nuova richiesta ad amazon stavolta ha un file cookie, che viene inviato ad amazon, quest'ultimo leggendo il cookie accede il database per comportarsi di conseguenza e la risposta sarà normale, senza bisogno di creare un nuovo cookie.

Si noti come i cookie possono avere vita variabile e anche dopo settimane di vita essere reinviati ancora ad amazon.

I cookie possono essere usati per:

- autorizzazione
- carello degli acquisti
- raccomandazioni
- stato della sessione dell'utente (e-mail Web)

I cookie in realtà rappresentano coppie chiave-valore, per cui abbiamo cookie del tipo jsessionID che rappresenta l'identificatore della sessione, ma ad esempio anche cookie lang che rappresentano la lingua.

Vediamo ora in azione i cookie in ambito pubblicitario (pg 6 PDF). Immaginiamo di voler accedere alla pagina del NY Times che contiene un'immagine pubblicitaria. *Per far sì che l'immagine sia visibile all'utente quest'ultimo è costretto a visitare il server pubblicitario (AdX nell'esempio).*

Dal punto di vista dei cookie: nytimes crea un ID per l'user agent e gli associa il fatto che ad una certa data ha visto una pagina di sport. Viene quindi restituita la pagina con incluso quel cookie (si ricorda che il cookie è creato dal server e memorizzato dal client). **I cookie impostati dai siti che visitiamo a tutti gli effetti sono detti **Cookie di “prima parte”** (“*first party*”).**

Nel contattare invece il sito pubblicitario per ottenere il ban pubblicitario viene creato da quest'ultimo un nuovo cookie restituito insieme all'immagine. Sarà in questo caso presente un'ulteriore riga di intestazione, la **Referrer**, che contiene l'URL da cui proveniamo. Quindi il server pubblicitario potrà creare un ID associandovi la pagina da cui proveniamo e restituire quel cookie.

Cookie di questo tipo, generati da siti web che non abbiamo scelto di visitare, sono detti **Cookie di “terze parti” (“*third party*”).**

Immaginiamo adesso di contattare un nuovo sito: socks.com. In questo caso il client non ha cookie per questo sito, quindi non ne invia uno, ma ancora una volta c'è un'immagine pubblicitaria nel sito. Quindi viene contattato il server pubblicitario (sempre AdX) a cui passo il vecchio cookie di terze parti che mi aveva impostato per nytimes e il nuovo Referrer che è il sito dei calzini.

Quindi il server AdX accede alla parte del database con il vecchio cookie con referrer nytimes e aggiunge anche che stavolta ho visitato il sito dei calzini, restituendo in pubblicità un'immagine appropriata (magari legata proprio allo sport, visto che avevo visitato una pagina del genere in precedenza). E così si ripete con altri siti che facciano riferimento a pubblicità del genere!

Sorge un problema: **AdX può tracciare (track) il mio comportamento in base ai siti che visito!**

In particolare i cookie possono essere usati per tracciare il comportamento di un utente su un dato sito (cookie di prima parte, sono di mio interesse affinché il sito si ricordi chi sono, ciò che mi interessa etc...) **ma anche per tracciare il comportamento di un utente su più siti (cookie di terze parti) senza neppure che l'utente abbia mai scelto di visitare il sito del tracker!** Il tracciamento, tra l'altro, può anche essere invisibile all'utente.

Per limitare i danni dei tracker (che eventualmente rivendono informazioni legate all'utente) il tracciamento di terze parti tramite cookie è disabilitato per impostazione predefinita su browser come Firefox e Safari, ed attualmente è in atto un'eliminazione graduale dei cookie di terze parti nel browser Chrome con l'obiettivo di bloccare tutti entro il terzo trimestre 2024.

Del resto, quando i cookie possono identificare un individuo, questi sono considerati dati personali soggetti a normativa GDPR (eu general data protection regulation).

Web Cache

È uno dei modi per migliorare le prestazioni nel Web.

L'idea di base è evitare di rispondere alle richieste con l'origin server, ma far rispondere ad un'entità terza (la **web Cache**) che è un server locale più vicino ai client.

Si configurano i client di modo che le richieste, in base a un certo protocollo, vadano al web Cache vicino e non all'origin server, in questo modo **se una richiesta non può essere soddisfatta dalla Cache questa la inoltra all'origin server per poi creare una copia locale dell'oggetto restituito**. In questo modo se un altro client in un secondo momento richiede lo stesso oggetto lo prende direttamente dalla Cache!

La web Cache agisce quindi come **server** nei confronti dello user agent e come **client** nei confronti dell'origin server. Se la Cache prendesse la copia

dell'oggetto e la mantenesse per sempre, *si rischierebbe di avere informazioni non più aggiornate*, e per questo il tempo di vita degli oggetti nella cache è controllato tramite riga Cache-Control e comando **max-age=<seconds>** in modo assistito dall'origin server (il server scrive **no-cache** se l'oggetto non può essere memorizzato).

Perché il web Caching?

- 1) **Riduce il tempo di risposta alle richieste dei client** (la cache è più vicina, si evita tutto il RoundTrip Time RTT di Internet che può essere maggiore)
- 2) **Se la cache si trova direttamente nella stessa rete di accesso ad esempio di una rete istituzionale, può ridurre il traffico sul collegamento di accesso** (che rappresenta spesso collo di bottiglia, poiché mentre i server sono connessi ad alta velocità il collegamento di accesso è spesso quello più lento)
- 3) **Si permette all'origin server di ricevere meno richieste**, garantendo la possibilità di avere provider “scadenti” che comunque offriranno dati con efficacia

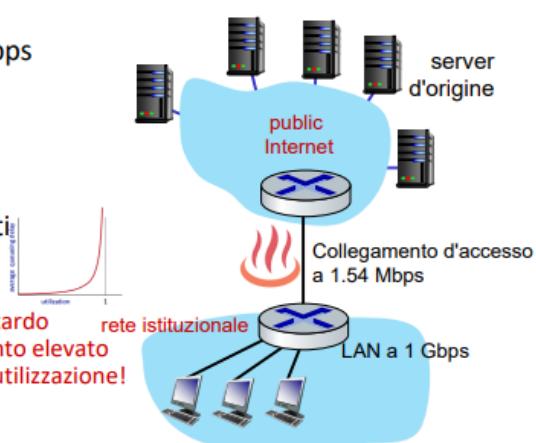
Esempio di caching

Scenario:

- velocità collegamento d'accesso: 1.54 Mbps
- RTT dal router istituzionale al server: 2 s
- dimensione di un oggetto: 100K bits
- frequenza media di richieste dai browser istituzionali al server d'origine: 15/s
 - velocità media di trasmissione dei dati ai browser: 1.50 Mbps

Prestazioni:

- utilizzazione del collegamento d'accesso = .97 *problema: ritardo d'accodamento elevato con elevata utilizzazione!*
- utilizzazione della LAN: .0015
- end-end delay = ritardo di Internet + ritardo del collegamento d'accesso + ritardo della LAN
= 2 s + minuti + microsecondi



Opzione 1: collegamento d'accesso più veloce

Scenario:

- velocità collegamento d'accesso: ~~1.54~~ Mbps
 - RTT dal router istituzionale al server: 2 s
 - dimensione di un oggetto: 100K bits
 - frequenza media di richieste dai browser istituzionali al server d'origine: 15/s
 - velocità media di trasmissione dei dati ai browser: 1.50 Mbps

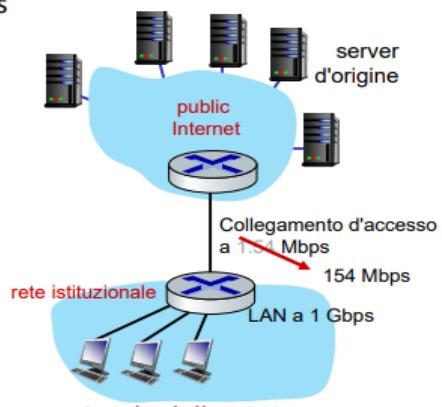
Prestazioni:

- utilizzazione del collegamento d'accesso = .97 → .0097
 - utilizzazione della LAN: .0015
 - end-end delay = ritardo di Internet +
msecs ← ritardo del collegamento d'accesso + ritardo della LAN
= 2 s + millisecondi + microsecondi

rete istituzionale

LA

Costo: collegamento d'accesso più veloce (costoso!)



Application Layer: 2-50

Opzione 2: installare un web cache

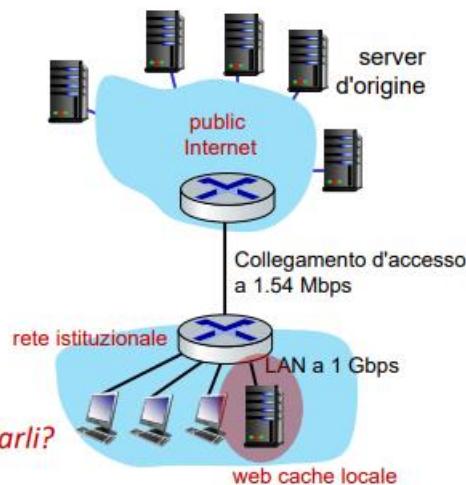
Scenario:

- velocità collegamento d'accesso: 1.54 Mbps
 - RTT dal router istituzionale al server: 2 s
 - dimensione di un oggetto: 100K bits
 - frequenza media di richieste dai browser istituzionali al server d'origine: 15/s
 - velocità media di trasmissione dei dati ai browser: 1.50 Mbps

Costo: web cache (economica!!)

Prestazioni:

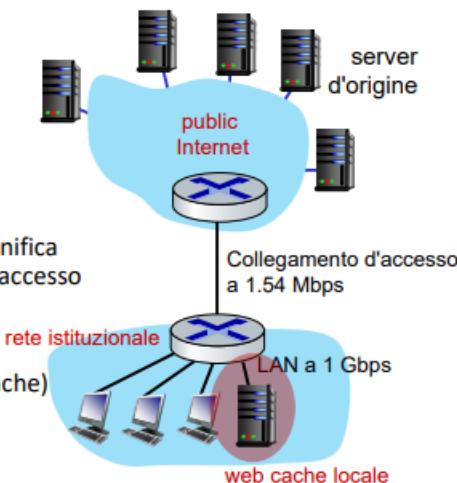
- utilizzazione LAN: .?
 - utilizzazione del link di accesso = ? *come calcolarli?*
 - ritardo end-end medio = ?



Calcolo dell'utilizzo del collegamento di accesso e del ritardo end-end con la cache:

supponiamo una percentuale di successo (hit rate) pari a 0.4:

- il 40% delle richieste sarà soddisfatto dalla cache, con ritardo basso (msec)
- 60% delle richieste sarà soddisfatto dal server d'origine
 - tasso di trasmissione sul collegamento d'accesso
 $= 0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
 - utilizzazione collegamento d'accesso $= 0.9 / 1.54 = .58$ significa basso (msec) ritardo di accodamento al collegamento d'accesso
- ritardo end-end medio:
$$\begin{aligned} &= 0.6 * (\text{ritardo dai server d'origine}) \\ &\quad + 0.4 * (\text{ritardo quando richiesta soddisfatta dalla cache}) \\ &= 0.6 (2.01) + 0.4 (\sim \text{msecs}) = \sim 1.2 \text{ secs} \end{aligned}$$



ritardo medio end-end inferiore che con un collegamento a 154 Mbps (e meno costoso!)

Nota come nella prima slide c'è un ritardo d'accodamento dovuto al fatto che si richiedono 1.5 Mb/s di dati con un mezzo di trasmissione a velocità 1.54 Mb/s!

Per risolvere il problema si potrebbe pensare di aggiornare il collegamento di accesso, comprandone uno 100 volte più veloce... ma ciò è eccessivamente costoso!

Con la cache si spende meno e si guadagna in termini di ritardo end-to-end, tuttavia per calcolarli si devono fare delle assunzioni ed il ritardo end-to-end sarà per questo motivo "medio". Si assume in particolare la percentuale di richieste esaudite dalla Cache e il resto dal server d'origine.

Abbiamo finora considerato la Cache come server a sé stante, magari presente proprio nella rete di accesso o comunque significativamente più vicino ai client di quanto non lo sia il server d'origine. Ma **il ruolo di Cache può anche essere assunto dall'host stesso, in particolare dal suo Browser** (che potrebbe mantenere in memoria un oggetto senza doverlo richiedere successivamente).

Un obiettivo primario della Cache è quello di non inviare un oggetto se non è una copia aggiornata. Per farlo il client invia un messaggio di richiesta con un'intestazione: ***If-modified-since: <data>***

Quando il server origin riceve questa richiesta, se l'oggetto non è stato modificato restituisce **304 Not Modified**, lasciando intendere che la copia in Cache è ancora valida e che quindi non deve rispedire i dati aggiornati (riducendo il tempo di trasmissione). Altrimenti viene inviato dal server l'oggetto aggiornato. (Si parla di **GET CONDIZIONALE**)

Nota sul caching

Il caching può essere effettuato da:

- una web cache, ossia uno speciale tipo di proxy, cui il browser invia le richieste invece che indirizzarle all'origin server.
- oppure, dal browser stesso, che conserva una copia degli oggetti richiesti in precedenza

In entrambi i casi, occorre prestare attenzione al problema dell'aggiornamento degli oggetti: vedi riga di intestazione *Cache-Control* e *GET condizionale*.

Con HTTP1.1 si possono usare connessioni persistenti e chiedere in **pipeline più oggetti** (cioè chiedo insieme n oggetti senza attendere che mi siano restituiti). In particolare il server risponde in ordine (**FCFC**, first come first served scheduling) alle richieste GET).

Tuttavia questa seconda peculiarità è soggetta all'**HOL** (*head of line blocking*) per cui oggetti piccoli devono aspettare per la trasmissione “dietro” ad oggetti più grandi richiesti prima di loro.

Un ulteriore problema di HTTP 1.1 è che quando ho più richieste se c’è uno stallo o perdita si bloccano tutte.

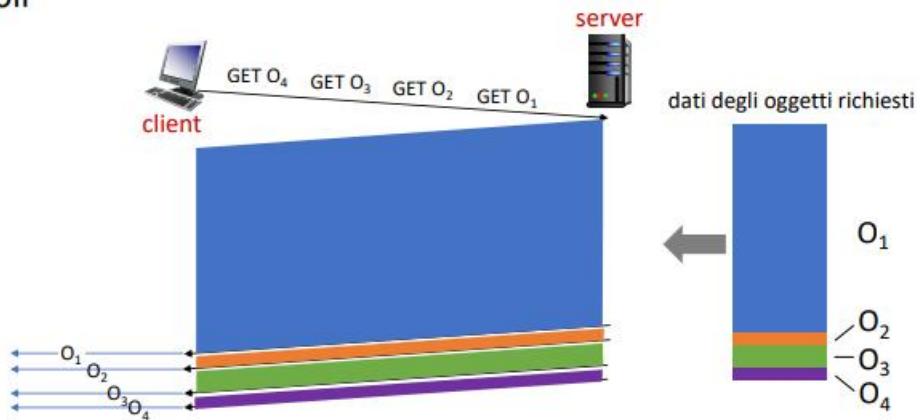
HTTP/2

HTTP/2 risolve questi problemi di **HTTP1.1** (approfondiremo come risolve il primo) garantendo maggiore flessibilità del server nell’invio degli oggetti al client.

- Eredita metodi, codice di stato e la maggior parte dei campi di intestazione da HTTP 1.1 (non ho problemi di retrocompatibilità)
- Il server non deve più inviare gli oggetti nell’ordine in cui sono richiesti
- Invio “**push**” al client oggetti aggiuntivi, prima che li abbia richiesti (il server quando visito una pagina sa gli oggetti che devo vedere, si evita quindi diversi RTT di richiesta)
- Risolve l’**HOL** frammentando gli oggetti in frame e intervallandoli

Vediamo chiaramente con un confronto grafico tra il comportamento di HTTP1.1 e HTTP/2 nel mitigare il problema dell’HOL

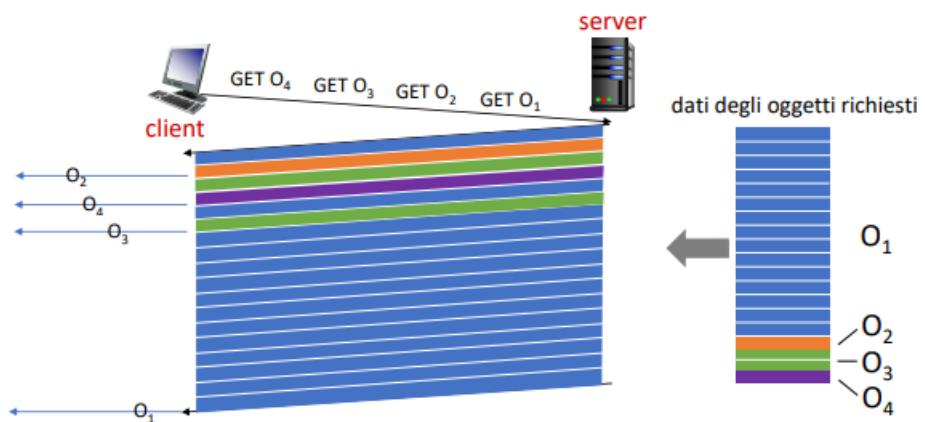
HTTP 1.1: il client richiede 1 oggetto grande (es., file video) e 3 oggetti più piccoli



oggetti consegnati nell'ordine in cui sono stati richiesti: O₂, O₃, O₄ aspettano dietro O₁

mitigazione

HTTP/2: oggetti divisi in frame, trasmissione de frame interlacciata



O₂, O₃, O₄ consegnati rapidamente, O₁ leggermente ritardato

Application Layer

E-mail

Ha tre componenti principali:

- **User agents** (o agenti utenti) che corrispondono ai nostri Browser, sono i programmi di posta elettronica
- **Mail servers** (o server di posta) usati per trasferire la posta tra i server e tra server e user agent (ma solo in invio)
- **Simple Mail Transfer Protocol SMTP**

I mail server hanno una **casella di posta** per ogni utente in cui viene inserita la posta in arrivo e una **coda di messaggi**, dove vengono posti i messaggi di uscita. Quando una persona vuole inviare un messaggio di posta con il suo user agent usa SMTP per inviare il messaggio al server, che lo mette nella coda di messaggi e prova ad inviarlo.

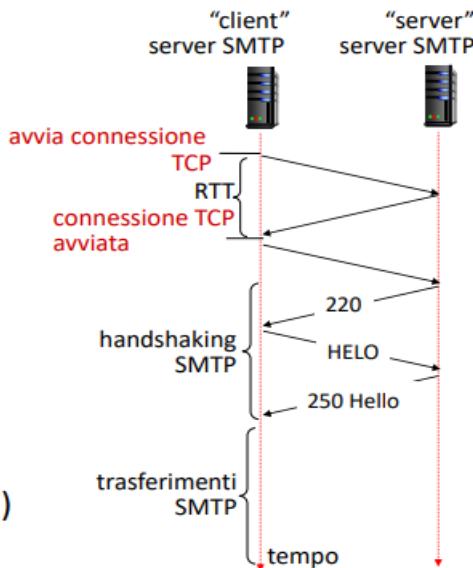
È necessario l'utilizzo di server intermediari per lo scambio di posta elettronica, per diversi motivi:

Lo User Agent non manda direttamente la lettera a destinazione perché magari quell'host non è attivo nel momento in cui invio la posta. Per cui metto la posta nella coda di uscita del mio mail server e tenterà lui di inviarla, ogni 30 min. Se dopo un determinato tempo non viene inviata, quella posta viene scartata notificando lo user agent a sua volta con una mail.

Del resto è per lo stesso motivo che la casella di posta è su un server e non sulla nostra macchina: mentre è spenta potrei ricevere delle mail!

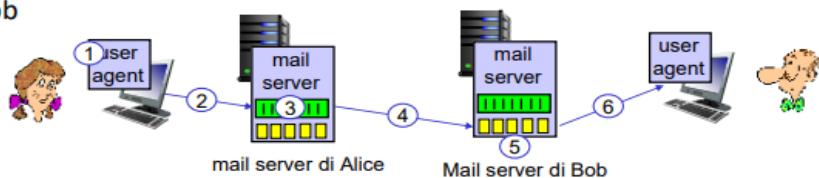
Anche SMTP usa il protocollo **TCP** (per garantire un trasferimento affidabile) dove i server sono in ascolto **sulla porta 25**.

Come prima vi è un RTT di connessione a cui segue l'handshake SMTP. Dopo aver stabilito la connessione TCP il server risponde con 220, a cui l'user agent risponde con un HELLO a cui il server risponde con un 250 Hello, dopodiché può avvenire il vero e proprio trasferimento ed infine chiusa la sessione SMTP e quindi anche la sessione TCP.



Scenario: Alice invia un'e-mail a Bob

- 1) Alice usa il suo user agent per comporre il messaggio da inviare "a" ("to") bob@someschool.edu
- 2) lo user agent di Alice invia un messaggio al server di posta di Alice; il messaggio è posto nella coda di messaggi
- 3) il lato client di SMTP apre una connessione TCP con il mail server di Bob
- 4) il client SMTP invia il messaggio di Alice sulla connessione TCP
- 5) il mail server di Bob pone il messaggio nella casella di posta di Bob
- 6) Bob invoca il suo user agent per leggere il messaggio



Esempio di interazione SMTP

```

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

```

Confronto HTTP e SMTP:

HTTP è protocollo **pull**, poiché tipicamente scarico oggetti.

SMTP è protocollo **push**, poiché tipicamente invio oggetti.

Entrambi hanno un'interazione comando/risposta in ASCII, ma c'è una differenza. HTTP lo usa solo per l'intestazione, mentre il corpo dell'entità può avere un tipo di carattere qualunque (anche immagini, trasferite in formato binario). Anche SMTP assume che il messaggio sia sempre testuale, e se voglio inviare altri media devo trasformarli in dei caratteri secondo determinate convenzioni. Questo tende ad espandere la conversione del messaggio e complica il processo richiedendo maggiore elaborazione.

Inoltre in HTTP la fine del messaggio (per capire fin dove ad esempio leggere il binario di un'immagine) è denotata da content-length, nel caso della posta invece la fine è indicata da un punto in una riga singola

In HTTP ogni oggetto è encapsulato nel suo messaggio di risposta, mentre in SMTP più oggetti vengono trasmessi in un unico messaggio.

NB. si usa il protocollo IMAP per il recupero delle mail dal Mail server, e non SMTP

Lez 6

(dettagli esercitazione lez 5 primi 15 min registrazione lez 6)

DNS Domain Name System

Si tratta del servizio di directory di Internet. Vediamo un parallelismo con il mondo reale e le persone: ognuno di noi, a seconda del contesto, è identificato in modo diverso (nome, codice fiscale, carta d'identità etc..)

Per quel che riguarda gli Host e i router di Internet, si hanno gli **indirizzi IP** a 32 bit e i "nomi" (**hostname**, es. cs.umass.edu). Sono identificatori usati a scopi diversi: IP per la funzione di instradamento usata dal livello di rete, hostname usato dalle persone. È necessario, al fine di mappare anche gli hostname e riconoscere logicamente gli indirizzi IP, tradurre l'uno nell'altro.

In questo senso si affronta il problema della **risoluzione dei nomi**.

L'approccio più statico possibile è quello di avere un **file hosts** (/etc/hosts

nei sistemi POSIX) che associa ad un indirizzo IP a uno o più hostname ad esso associati. Si tratta di un file locale a un nodo, e non abbiamo garanzie che ogni nodo abbia gli stessi hostname. Se così fosse sarebbe molto problematico, immaginiamo ad esempio che per uno ad un indirizzo IP è associato lo stesso hostname di un altro host con diverso indirizzo IP.

Inizialmente ('70) il file hosts era affidato ad un'unità centralizzata con un unico megafile. Ciò però portava a diversi problemi: necessità di aggiornare continuamente e installare il file di volta in volta per ogni nodo. Si generava quindi un enorme traffico verso la macchina che manteneva il file (soluzione che non scala!)

Oggi per ovviare a questi problemi il DNS è quindi un **database distribuito** implementato in una gerarchia di **name server** (sistema distribuito in più nodi di calcolo, non più centralizzato). Opera tramite un **protocollo a livello d'applicazione** che può essere usato dagli host, router e server per effettuare *la risoluzione dei nomi* (tradurre i nomi in indirizzi). Quindi ho più name server distribuiti che presentano il file hosts a cui dobbiamo fare riferimento per usufruire del servizio DNS.

Si tratta di un protocollo a livello d'applicazione poiché è una funzione usata da tutte le applicazioni: è come se tutte le applicazioni dipendono da questa. Il DNS quindi, così come molte altre funzioni critiche, sono implementate a livello di applicazione nelle parti periferiche della rete (non nel nucleo, sarebbe difficile da gestire).

- Il DNS offre anzitutto il *servizio di Traduzione degli hostname in indirizzi IP*
- Il DNS offre anche un altro *servizio che offre la possibilità di mappare un certo nome (alias) ad un nome canonico più complicato* (es. amazon.com ha in realtà un nome più complicato).
- Inoltre possiamo associare ad un nome a dominio un alias di un nome canonico di un mail server (*mail server aliasing*), per cui ad esempio dato il dominio google.com ho un nome canonico che è il server web di Google e possiamo associargli tramite record specifico il nome canonico di un'altra

macchina che fa da mail server. (il servizio di posta viene restituito con record diverso).

Si ribadisce: identificatori o nome o IP. Ma il nome può essere sinonimo di un nome più complicato che può avere altri sinonimi.

- Un ulteriore servizio è quello della *load distribution* (distribuzione del carico di rete) per cui se ad esempio chiedo www.google.com non mi viene restituito un solo indirizzo IP, ma più di uno (ho più server che gestiscono lo stesso servizio). In questo modo, scegliendone di volta in volta uno diverso per ogni host, si distribuisce il carico.

DNS: servizi, struttura

Servizi DNS

- Traduzione degli hostname in indirizzi IP
- host aliasing
 - nome canonico e alias
- mail server aliasing
- load distribution (*distribuzione del carico di rete*)
 - server Web replicati: più indirizzi IP corrispondono a un solo nome

Q: Perché non centralizzare il DNS?

- un *single point of failure* (punto di vulnerabilità)
- volume di traffico
- database centralizzato distante
- manutenzione

R: non scala!

- Solo i server DNS di Comcast: 600B query DNS al giorno
- Solo i server DNS Akamai: 2,2T query DNS al giorno

Pensare al DNS

un enorme database distribuito:

- ~ miliardi di record, ciascuno semplice

gestisce molti *trillioni* di interrogazioni al giorno :

- *molte* più letture che scritture
- *è importante*: quasi tutte le transazioni Internet interagiscono con il DNS - i millisecondi contano!

organizzativamente e fisicamente decentralizzato

- milioni di organizzazioni diverse responsabili dei loro *record*

"a prova di proiettile": affidabilità, sicurezza

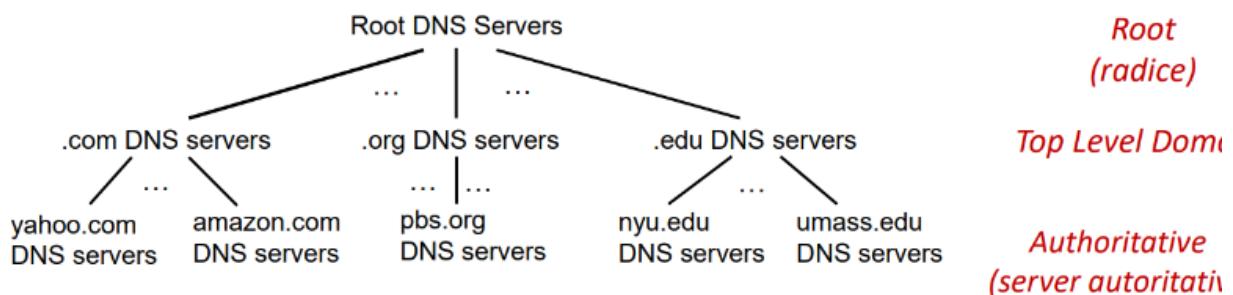


Si deve quindi pensare al DNS come un database distribuito e gerarchico.

Ma come è organizzata la gerarchia dei server DNS?

In cima abbiamo una manciata di **Root DNS Servers**, server radice, che sono considerati l'ultimo punto di contatto. Se non so come **risolvere** un hostname contatto uno dei Root Server. Questi server non contengono la classica mappatura di hostname-indirizzo IP associato, ma fanno riferimento a un server di livello inferiore detto **TLD** (Top Level Domain) che è responsabile di uno specifico nome a dominio di primo livello (praticamente l'ultima parola dopo il punto per un hostname es. .com, .org etc..).

Anche il TLD non contiene tipicamente non contiene la mappatura degli hostname per ogni nome a dominio di primo livello, ma fa riferimento a tanti server di livello più basso (detti **Authoritative Servers**) che gestiscono effettivamente la mappatura tra hostname di una specifica organizzazione e i suoi IP.



Se voglio registrare miosito.uniroma2.it devo prima comunicare al server .it che voglio pubblicare questo nome a dominio, quello registrerà al suo interno che per accedere al sito deve far riferimento a quel nome.

Questa gerarchia a tre livelli **in realtà ne presenta anche ulteriori**: talvolta i server di competenza (con la mappatura effettiva) si trovano a livelli più bassi (es. art.uniroma2.it).

Man mano che scendo nella gerarchia, sto considerando un suffisso sempre più grande. Solo quando arrivo all'ultimo server più basso ho la mappatura effettiva.

Il root name server è una **funzione critica, incredibilmente importante per Internet** (senza non funzionerebbe) ed è gestito dal protocollo **DNSSEC** che offre sicurezza.

Si hanno nel mondo **13 root name server logici, ciascuno con il proprio indirizzo IP**. Questi server sono replicati più volte in tutto il mondo mantenendo il loro stesso indirizzo IP (ca 200 server negli USA, in totale oltre 1813).

Questi root name server sono gestiti da 12 operatori, coordinati dalla IANA.

I DNS server autoritativi sono propri di ciascuna organizzazione ed offrono mapping ufficiali da hostname a IP per ogni host dell'organizzazione, possono essere mantenuti dall'organizzazione in sé o dal service provider (es. se compro un dominio da aruba è questa che si occupa di mantenere il sito aggiornando e tutto il resto).

Al di fuori della gerarchia si ha tipicamente un **DNS name server locale**, che svolge **la funzione di default name server**. Quando mi collego ad internet viene automaticamente configurato sulla mia macchina un default name server, che rappresenta il nostro punto di riferimento. Se devo risolvere un hostname è questo server che si occupa di inizializzare il processo.

Il name server locale svolge anche un **servizio di cache**: quando chiedo un nome al name server locale che ancora non ho cercato il name server locale lo trova seguendo la struttura a livelli descritta in precedenza e lo salva nella propria cache, di modo che se ricercherò successivamente il sito successivamente il suo hostname sarà già risolto nel name server locale (a me più vicino, meno tempo per risolvere!). Anche in questo caso vi sarà un **timetolive** per evitare di avere dati stantii, tipicamente 2 giorni. Il DNS server locale è fornito dal mio ISP e talvolta implementato direttamente nel modem.

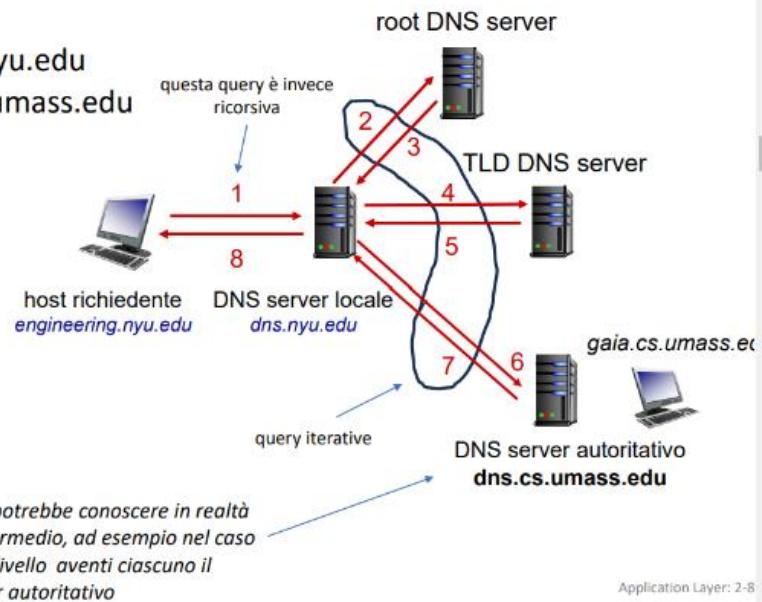
Vediamo come funziona più nel dettaglio.

DNS: interrogazione iterativa

Esempio: l'host engineering.nyu.edu vuole l'indirizzo IP di gaia.cs.umass.edu

Query iterativa

- Il server contattato risponde con il nome del server da contattare
- "Io non conosco questo nome, ma puoi chiederlo a questo server"



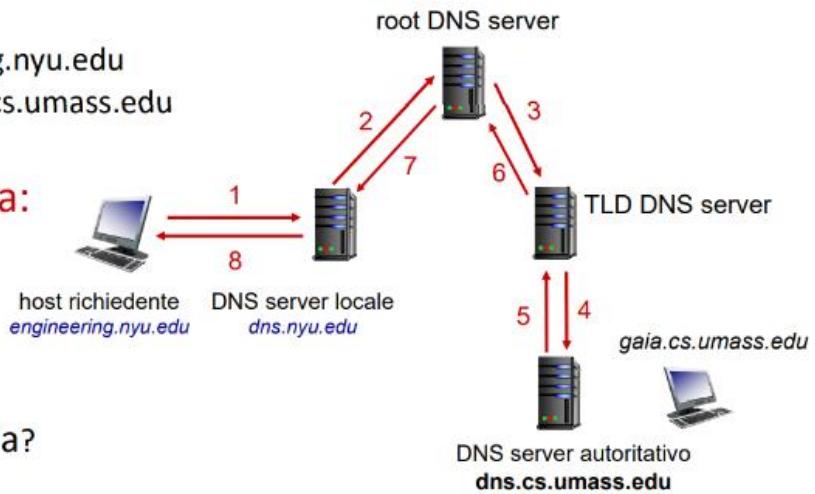
Il mio host ha bisogno di risolvere l'hostname. Per prima cosa contatta il name server locale. Se non lo possiede chiede al root name server il quale non gli risponde con l'IP corrispondente, ma prendendo nota del top level domain contenuto nell'hostname (.com, .org etc..) risponderà con un referral al TLD DNS server corrispondente, che verrà quindi contattato e a sua volta manderà un referral al DNS server autoritativo, che contiene e restituisce l'indirizzo IP corrispondente. In ognuno di questi livelli può in realtà avvenire del caching, cioè se un indirizzo IP che non è nativamente ospitato nel server può essere presente in cache e venir restituito immediatamente. (ciò avviene principalmente con il DNS server locale).

DNS: interrogazione ricorsiva

Esempio: l'host engineering.nyu.edu vuole l'indirizzo IP di gaia.cs.umass.edu

Interrogazione ricorsiva:

- Affida il compito di tradurre il nome al server contattato
- carico pesante ai livelli superiori della gerarchia?



È un approccio che si ha quando il name server più ad alto livello si occupa di contattare direttamente il name server più competente, fino a risolvere l'hostname e restituire l'indirizzo dai livelli più bassi fino al root che poi restituisce al server locale e poi all'host richiedente (quindi il DNS server locale fa effettivamente una sola query, al server DNS root).

La ricorsione non è il massimo perché affida ai server di gerarchia più alta maggiore carico.

Si possono anche combinare i due approcci: ad esempio server più in alto approccio iterativo (non li carica troppo di lavoro) e per quelli più in basso ricorsivo.

Record DNS

I dati sono memorizzati in delle strutture chiamate **RR** (resource record), che sono delle quadruple di formato (*name, value, type, ttl*).

In base al type il record avrà name e value diverso.

type=A

- *name* è l'hostname
- *value* è l'indirizzo IP

type=NS

- *name* è il dominio (ad esempio, *foo.com*)
- *value* è l'hostname dell'autoritative name server per questo dominio

type=CNAME

- *name* è il nome alias di qualche non "canonico" (nome vero)
- *www.ibm.com* è in realtà *servereast.backup2.ibm.com*
- *value* è il nome canonico

type=MX

- *value* è il nome del server di posta associato a *name*

Applicati

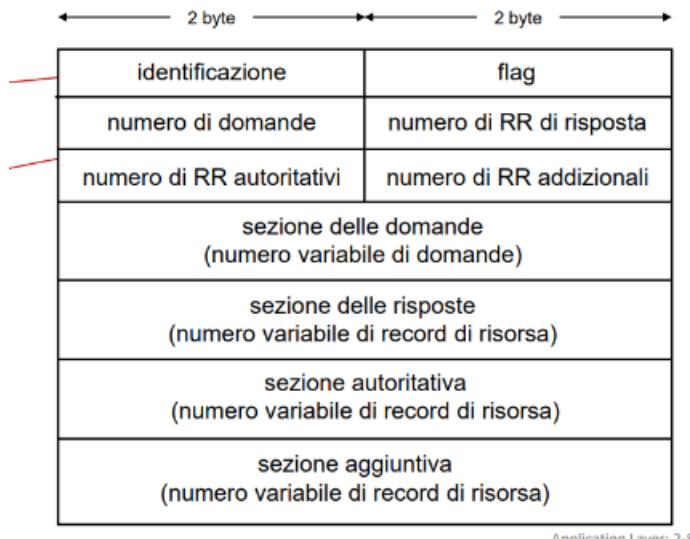
Type A per la conversione hostname-IP. Il **Type NS** serve per il meccanismo di referral visto in precedenza (non so risolvere direttamente l'hostname, ma posso darti quello di un nameserver che gestisce quel dominio tramite referral con hostname e suo indirizzo IP). **CNAME** serve per risolvere l'hostname nel suo hostname “vero”, più complesso (es. www.ibm.com è in realtà *servereast.backup2.ibm.com*). Infine il **Type MX** è quello per cui al nome del server è associato come value il nome del suo corrispettivo server di posta elettronica.

Quindi ad uno stesso nome a dominio posso associare un hostname canonico (più complesso) e un hostname canonico per il server di posta.

Vediamo ora il formato delle richieste (**query**) e delle risposte (**reply**).

Entrambe hanno lo stesso formato, ciò che cambia è il numero di bit che descrivono se è richiesta o risposta. Ogni messaggio ha **un'identificatore** (16 bit sia domanda che risposta) utile per associare richiesta con risposte. Ciò perché tra i protocolli a livello di trasporto DNS può usare sia TCP che UDP. Mentre con TCP è semplice riconoscere dov'è la richiesta e la risposta (stessa connessione), ma con UDP quando faccio una richiesta non è detto che ottenga una risposta immediatamente. Quindi se faccio in UDP due richieste come faccio a sapere, dopo aver ottenuto le risposte, chi è la risposta di chi? Devo matchare gli identificatori.

Un messaggio può avere anche vari **flag**, che identificano se si tratta di domanda o risposta, di richiesta di ricorsione, del fatto che la ricorsione è disponibile, che si tratti o meno di un DNS server autoritativo etc..



Si hanno poi dei **contatori** che ci dicono per le 4 sezioni in basso quanti sono gli elementi per ogni sezione.

Vediamo nel dettaglio le sezioni. **La prima sezione contiene le richieste.** Senza entrare troppo nei dettagli, si possono avere richieste del tipo *mi dai il value per un record di type A il cui nome è hostname?* Nella **risposta** ritroveremo una copia della domanda e una **o più risposte** (*perché più risposte?* Per la questione del carico di rete, talvolta certi hostname possono avere più server di gestione e quindi più indirizzi IP).

Abbiamo poi la **sezione autoritativa**, utilizzata per il meccanismo di **referral**. Non conosco la risposta alla tua domanda ma ti do un record NS (name server) per un DNS server che è più competente di me a cui puoi inoltrare la richiesta.

Nella **sezione aggiuntiva**, poiché nella autoritativa via referral viene dato solo un record NS (e quindi con l'hostname del server più competente), *si trova tipicamente un record di Type A* per risolvere l'hostname del DNS a cui si è fatto il referral, in generale si trovano informazioni extra che possono essere usate.

Inserire record nel database DNS

Esempio: abbiamo appena avviato la nuova società “Network Utopia”

- Registriamo il nome networkuptopia.com presso il **DNS registrar** (ad esempio, Network Solutions, oppure un altro dei concorrenti accreditati dall'ICANN)
 - forniamo al registrar il nome e gli indirizzi IP degli authoritative name server (primario e secondario)
 - il registrar inserisce due RR nel TLD server .com:
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
- Inseriamo localmente nell'autoritative server
 - un record A per www.networkutopia.com
 - un record MX per networkutopia.com

Essendo il DNS una funzione critica, non solo deve essere scalabile e a prova di proiettile rispetto al normale traffico, ma anche resistere ad eventuali attacchi.

Per questi sono state introdotte varie estensioni tra cui la **DNSSEC**, che permettono di evitare vari problemi tra cui quelle di attacchi di spoofing. Chi fa spoofing intercetta le query DNS e restituisce risposte fasulle, dando ad esempio indirizzi di macchine malevoli invece di quelle originali. La DNSSEC in questo senso offre servizi di autenticazione.

Per gli attacchi DDoS invece su **DNS root è piuttosto complesso**, poiché sono molti, vi è filtraggio del traffico (non ci si aspetta traffico elevato) e grazie ai DNS locali molte macchine mantengono in cache gli indirizzi IP dei server LTD, consentendo di aggirare i root server.

Attaccare i server TLD risulta molto più semplice in questo senso (più esposti perché molto trafficati, si pensi solo al dominio .com).

Peer To Peer

Si tratta di applicazioni di rete *che non hanno un server sempre attivo*, ma che sfruttano più coppie di peer per avere da una parte più carico ma dall'altra più

forza lavoro, tramite il concetto di **scalabilità intrinseca** (se si unisce un nuovo peer non prende solo informazioni ma le condivide pure).

Analizzeremo queste architetture facendo riferimento al problema della **distribuzione dei file**, consideriamo ad esempio uno scenario in cui si vuole distribuire un file di dimensione F da un server con banda u_s in upload a N client diversi (con banda d_i in download). Il server, per caricare una copia di un file, impiega mediamente F/u_s . Per inviare N copie NF/u_s . Per scaricare una copia l'iesimo client impiega tempo pari ad F/d_i .

Il client che impiega effettivamente più tempo a scaricare è quello con banda inferiore, per cui il tempo per distribuire il file F ad N client usando l'approccio client-server è il seguente:

$$\boxed{\begin{array}{l} \text{Tempo per distribuire } F \\ \text{a } N \text{ client usando} \\ \text{l'approccio client-} \\ \text{server} \end{array} D_{c-s} \geq \max\{NF/u_s, F/d_{min}\}}$$

aumenta linearmente in N

Vediamo ora la stessa situazione ma con approccio P2P.

Il server, che è colui che ha a disposizione inizialmente il file, deve caricarne almeno una copia, con tempo F/u_s . Ciascun peer impiegherà F/d_i per scaricarlo, e ancora una volta il download più lento sarà legato al peer con download minore. Ma questa volta, **volendo considerare quanto ci vuole a trasmettere tutte le N copie, possiamo considerare il fatto che le varie porzioni del file saranno sparse nella rete e quindi un nuovo nodo potrebbe richiedere il file non dal server iniziale, ma da un altro peer che lo ha scaricato in precedenza!** Abbiamo quindi con buona approssimazione una capacità di upload totale che, oltre a quella del server, è anche quella di tutti i peer.

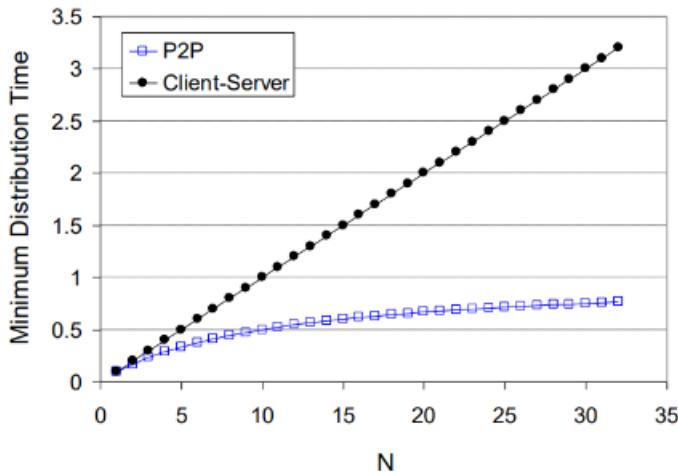
$$\boxed{\begin{array}{l} \text{Tempo per distribuire } F \\ \text{a } N \text{ client usando} \\ \text{l'approccio P2P} \end{array} D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}}$$

aumenta linearmente in N ...
... ma anche questo, dato che ogni peer porta con sé la capacità di servizio

Application Layer

Client-server vs. P2P: example

banda di upload del client = u , $F/u = 1$ ora, $u_s = 10u$, $d_{min} \geq u_s$



P2P non ha più una crescita propriamente lineare, poiché man mano che aumentano i peer aumenta anche la capacità in upload e quindi il tempo minimo di distribuzione.

BitTorrent

Esempio concreto di P2P. Abbiamo un gruppo di peer che si scambiano il file detto **torrent**, interno al quale è presente un server detto **tracker** che ha come principale funzione quella di tenere traccia dei peer che fanno parte del torrent. Normalmente il file viene scaricato in parti detti **chunk** in genere di 256 kb.

Succede ciò che segue, una volta che un peer entra a far parte del torrent (non in ordine, semplicemente le cose che accadono):

- *Chiedo al tracker la lista dei peer.* Egli non me li darà tutti, ma una cinquantina con cui cercherò di stabilire una connessione. I peer con cui riesco a stabilire una connessione sono detti **peer vicini** (“**neighbors**”). Informa periodicamente il tracker che è ancora nel torrent
- *Mentre scarica chunk,* può trasmetterne ad altri peer vicino. L’insieme di vicini con cui comunico non è sempre lo stesso, poiché peer possono uscire così come nuovi peer entrare
- *Una volta che il peer ha acquisito l’intero file* un peer può (egoisticamente) lasciare il torrent oppure (altruisticamente) rimanere come **seeder**.

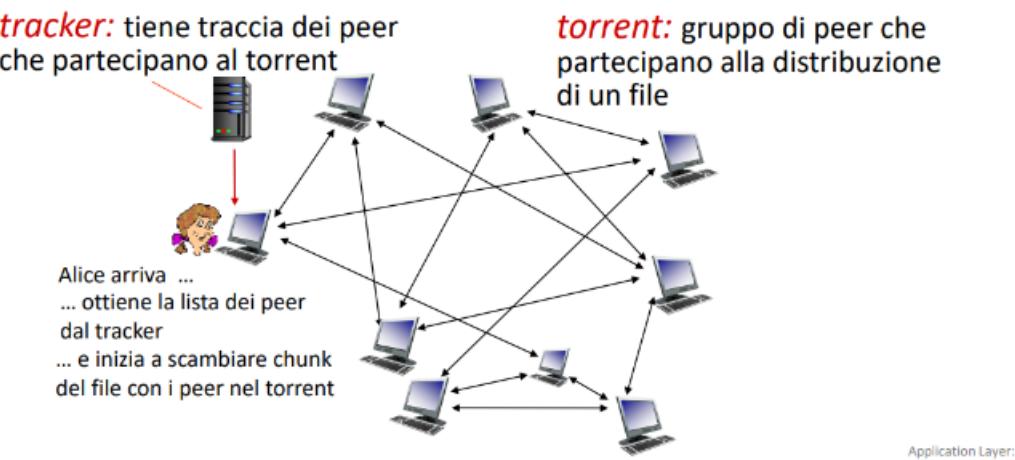
Quindi il tracker suggerisce i peer con cui dialogare. Ma come dialoghiamo con essi effettivamente?

Il dialogo è basato sulla **richiesta di chunk**:

- In ogni momento, peer diversi hanno sottinsiemi diversi di chunk
- Periodicamente, chiedo ai peer vicini l'elenco di chunk in loro possesso
- Chiedo ai peer di inviarmi i chunk mancanti, adottando la strategia del **rarest first**: chiedo anzitutto i chunk meno diffusi per uniformare la distribuzione dei chunk, migliorando la disponibilità globale e aumentando le possibilità di scambio (maggiore throughput), il fatto che un chunk sia presente su più peer rende la rete più robusta!
- *Un peer appena entrato può non applicare questa strategia ma chiedere un chunk casuale*, che può scaricare più velocemente. Un peer che sta per scaricare totalmente il file, invece, può adottare la strategia **endgame** e richiedere gli stessi chunk d'interesse a più peer simultaneamente (cancellando le richieste pendenti appena riceve un blocco)

Distribuzione di file P2P: BitTorrent

- file diviso in chunk (parti), in genere di 256 kB
- i peer nel torrent inviano/ricevono chunk del file



Application Layer:

Il problema fondamentale di queste reti distribuite è l'egoismo (se un peer prende senza condividere, **aumenta il carico senza contribuire alla velocità di servizio della rete**).

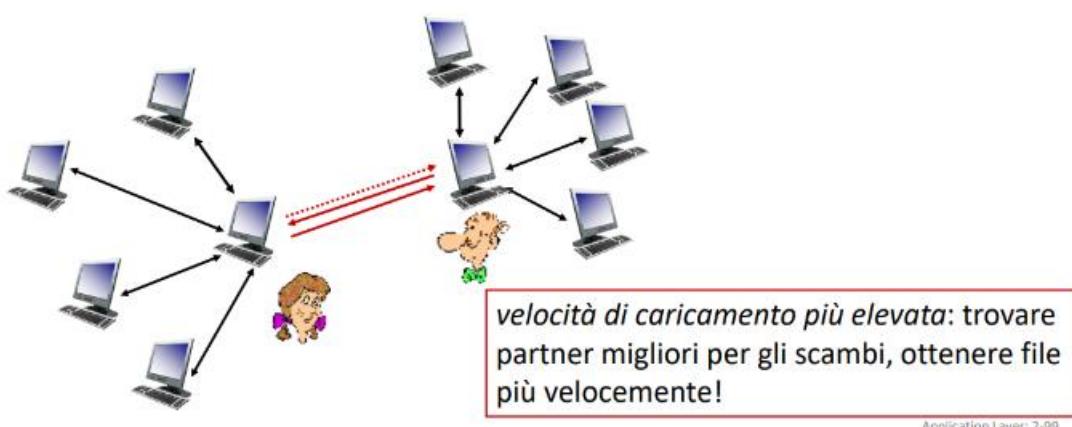
In questo senso BitTorrent utilizza una strategia detta **tit-for-tat**:

un vicino mi consente di scaricare dati solo se gli trasmetto i miei! In particolare dei miei vicini solo quattro saranno quelli a cui invio i dati, detti **unchoked**, che sono gli stessi che mi inviano i chunk alla velocità più alta. Questi 4 peer sono determinati ogni 10 secondi, gli altri peer sono detti **choked** poiché non gli invio nulla.

Ma come faccio a ricevere chunk se io non ne ho nemmeno uno? Casualmente, ogni 30 secondi, ogni peer seleziona in modo casuale un vicino detto **optimistically unchoked** a cui inizio a inviare chunk senza aspettarmi di ricevere nulla. In questo modo si garantisce che peer senza chunk ne possano ottenere e inoltre questo peer a cui inizio a trasmettere potrebbe diventare uno dei 4 vicini che mi inviano dati a sua volta!

BitTorrent: tit-for-tat

- (1) Alice scelte Bob come "optimistically unchoked"
- (2) Alice diventa uno dei primi quattro fornitori di Bob; Bob ricambia
- (3) Bob diventa uno dei primi quattro fornitori di Alice.



Streaming video e CDN

CDN == *Reti di distribuzione di contenuti (Content Distrib. Networks)*

Si parla di servizi come Youtube, Amazon Video, Netflix etc...

Si tratta di un servizio molto popolare che implica molte sfide perché possa funzionare: **Scala** – come raggiungo miliardi di utenti? **Eterogeneità** – utenti diversi hanno capacità diverse (ad es. cablati o mobili, ricchi di larghezza di banda o poracci...)

Soluzione: avere **un'infrastruttura distribuita** a livello di applicazione (similmente al DNS, tanto carico distribuito → infrastruttura distribuita).

Un **video** è **una sequenza di immagini** visualizzate a tasso costante (es. 24 FPS).

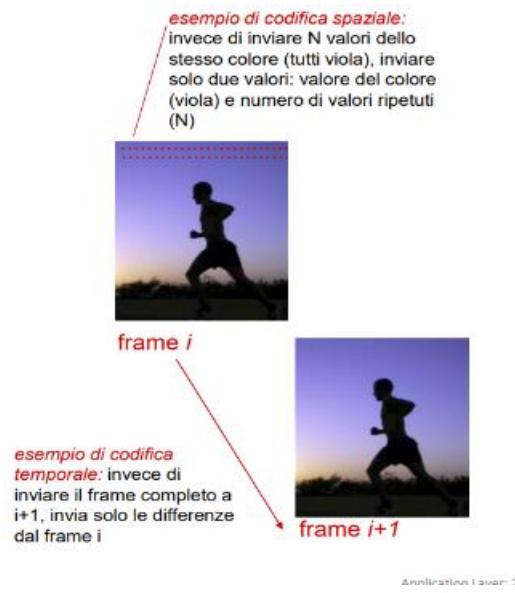
Un'**immagine** è un **array di pixel**, dove ogni **pixel** è *rappresentato da bit*.

Ogni pixel (picture element) rappresenta un colore diverso. Il colore di un pixel è tipicamente definito da tre valori (**R G B**, red green blue, tricromia).

L'approccio più semplice è il seguente: codifico R, G e B con 256 numeri (un byte per ogni colore) per cui un pixel che è insieme di questi tre colori fondamentali mi richiede **3 byte**. (1000 pixel -> 3000 byte!).

I video, essendo sequenze di immagini, adottando semplicisticamente questo schema richiederebbero troppo spazio. Si può fare di meglio (anche per singole immagini) adottando uno **schema di codifica (codec, codifica-decodifica)**.

I codec applicano modifiche reversibili per cui si utilizza la ridondanza **all'interno (ridondanza spaziale)** e **tra le immagini (ridondanza temporale)** per ridurre il numero di bit utilizzati per la codifica.



Si possono inoltre avere due tipi di codifica:

- **CBR (bit rate costante)** cerco di codificare ogni frame con lo stesso numero di bit
- **VBR (bit rate variabile)** per cui il bit rate cambia a seconda della quantità di rindondanza

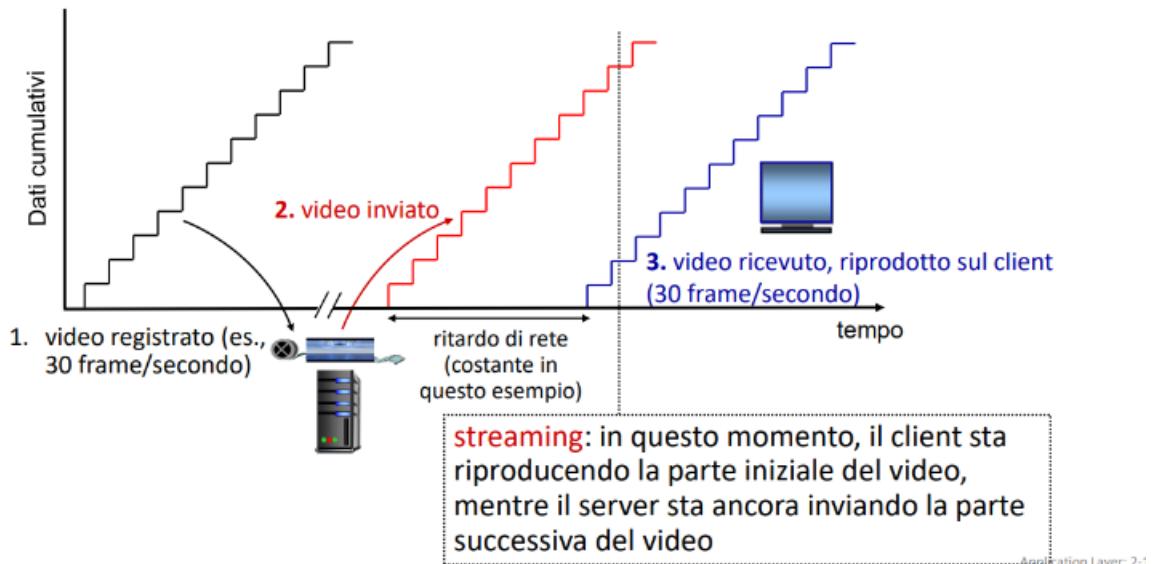
Maggiore bitrate -> maggiore dettaglio/qualità

Nello scenario seguente faremo riferimento a **contenuti registrati**, non parliamo della diretta che ha ulteriori problemi da affrontare.

Il principale problema in questo scenario è che **la larghezza di banda** tra client e server può **variare nel tempo** (con i livelli di congestione della rete), per cui potrei non essere in grado di sostenere in continuazione lo stesso flusso di bit. La stessa congestione di rete può inoltre causare la perdita di pacchetti.

Osserviamo un grafico in cui i dati sono trasmessi cumulativamente nel tempo:

Streaming video di contenuti registrati



Sto trasmettendo il file con un tasso fisso. Ogni gradino nel grafico rappresenta il fatto che ogni tot secondi è inviato un nuovo frame (quindi è inviato un nuovo numero di bit costante). Il ritardo di rete è il motivo per cui prima di iniziare un video devo aspettare il caricamento. Questo approccio di consumazione dei dati si chiama streaming perché avviene “consumazione” dei dati nel mentre questi sono ancora in corso di trasmissione. Il vantaggio è che ho tempi di risposta più brevi e se interrompo la visione posso anche interrompere la trasmissione.

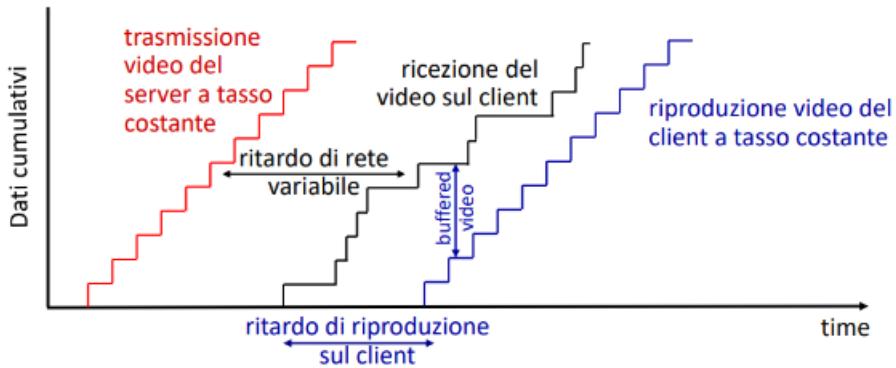
Nello streaming tuttavia uno dei vincoli principali è quello **di riproduzione continua**: non voglio schermate di caricamento, quando la riproduzione inizia dovrebbe procedere secondo i tempi di registrazione originali.

Il grafico visto non tiene conto dei **ritardi** che, come detto, sono **variabili (jitter)**, ma considera un ritardo costante.

È proprio per via di questi jitter che **la curva di trasmissione non è identica a quella di ricezione** (come si vede nel seguente grafico) e se iniziassi a riprodurre i dati non appena ricevuto il primo frame potrei non aver ancora ricevuto i frame successivi (la cadenza con cui mi arrivano i frame non matcha quella di riproduzione). Ciò che faccio è *quindi differire la riproduzione di una quantità detta ritardo di bufferizzazione di modo che possa mantenere una certa differenza tra i dati che ho riprodotto e quelli ricevuti fino a quel momento*. Non voglio infatti che la curva blu non vada sopra a quella nera, se

succedesse vorrebbe dire che a un dato istante dovevo riprodurre più di ciò che ricevevo! La spostò affinché possa assorbire ritardi variabili di ricezione.

Streaming video di contenuti registrati



- ***buffering lato client e ritardo di riproduzione:*** compensare il ritardo aggiunto dalla rete, il jitter (variazione) del ritardo

Lez. 7

Ricapitoliamo lo streaming video. Si tratta di uno dei grandi consumatori di larghezza di banda di Internet (Netflix, Prime, Youtube...).

Due sfide: scala (raggiungere miliardi di utenti) e eterogeneità (ogni utente ha capacità ed esigenze diverse).

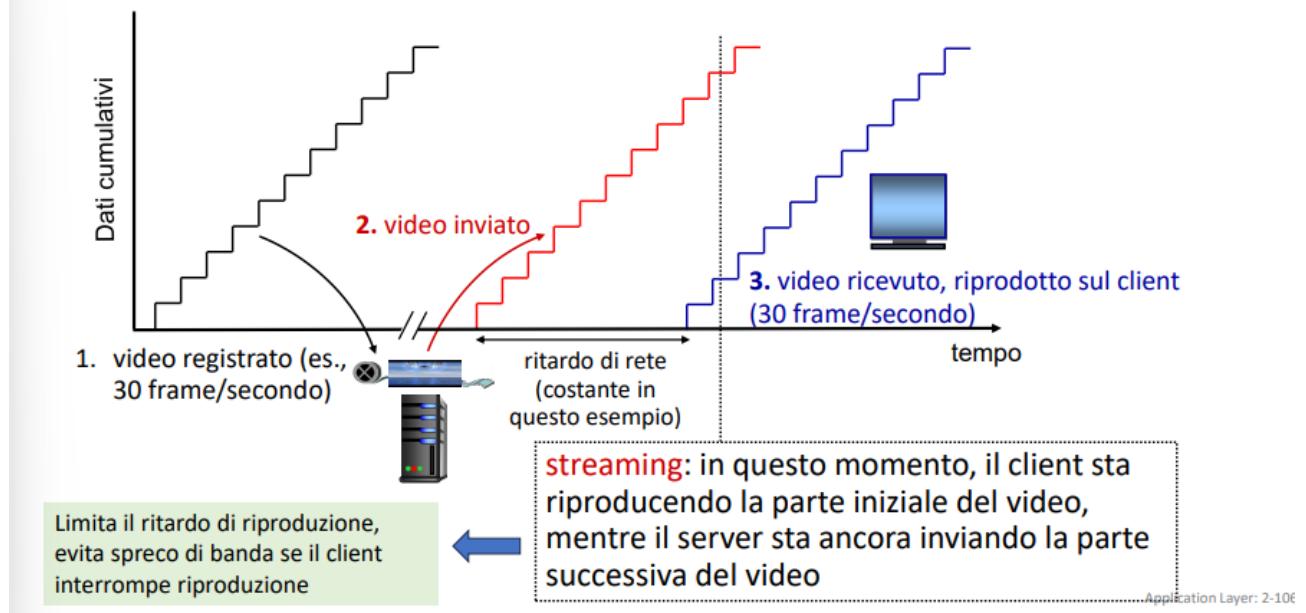
Soluzione: similmente al DNS, applicare una struttura distribuita a livello di applicazione.

(attualmente codifica immagini preferibile via costant bit rate perché so già quanto devo scaricare).

Ci concentriamo solo su contenuti registrati e tralascieremo quelli in diretta, che sono simili ma hanno anche delle complicazioni aggiuntive.

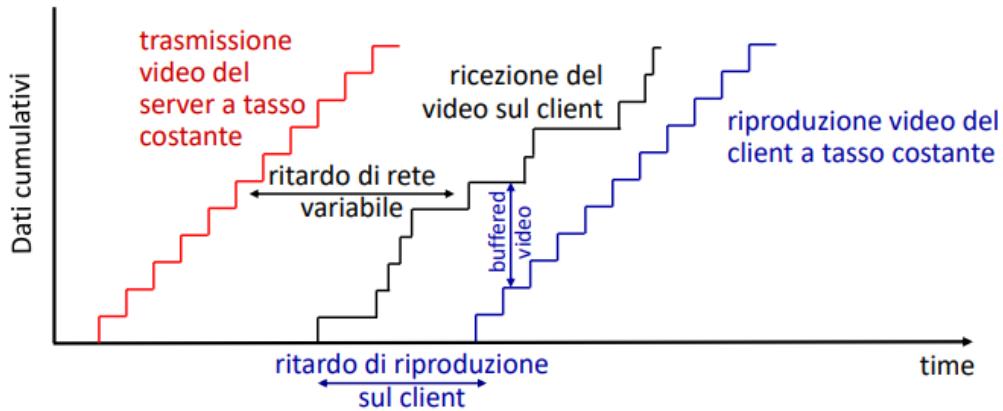
Il problema risiede nella variabilità della banda. Vediamo il primo diagramma: il video è registrato a tasso costante (codifica CBR, numero frame fissi al secondo). Ogni scalino rappresenta la codifica di un nuovo frame. Supponiamo di trasmettere il video a velocità costante (gradini rossi). Il client riceve quindi la sequenza di bit che rappresentano i frame con ritardo costante, quindi semplicemente nel grafico trasliamo i gradini blu a dx. Nel mondo ideale quindi non appena l'utente riceve il primo bit può iniziare la riproduzione.

Streaming video di contenuti registrati



Nella parte tratteggiata verso l'inizio degli scalini blu, l'utente sta riproducendo il video nel mentre sta scaricando frame successivi, da qui il termine **streaming!** Ciò riduce notevolmente il ritardo di riproduzione rispetto a scaricare tutto il contenuto in una volta ed evita spreco di banda se interrompo nel mezzo la riproduzione (se avessi scaricato l'altra metà non mi sarebbe servita più).

Il requisito più importante dello streaming è che una volta iniziata la riproduzione questa deve proseguire linearmente, senza interruzioni/perdite (**vincolo di riproduzione continua**). Ad ostacolare questo requisito vi è il fatto che i ritardi di rete sono variabili (**jitter**), quindi avrò bisogno di una coda (buffer) al lato client per soddisfare i vincoli di riproduzione continua. Ulteriori sfide riguardano le interattività del client come pausa, avanzamento veloce, riavvolgimento etc... ed il fatto che alcuni pacchetti persi devono essere ritrasmessi.



- **buffering lato client e ritardo di riproduzione:** compensare il ritardo aggiunto dalla rete, il jitter (variazione) del ritardo

In rosso come sono trasmessi i dati, in nero come sono ricevuti. Si distorce in quel modo perché talvolta il ritardo è più lungo o più corto (variabile). La curva di riproduzione (blu) deve però coincidere con quella di trasmissione (riproduzione continua). Se iniziassi a riprodurre il filmato non appena ricevo il primo bit potrei non aver ancora ricevuto i successivi per via del ritardo variabile, per ovviare al problema aspetto del tempo prima di riprodurre il video accumulando dati (**buffer video** nel grafico) per “assorbire” la variabilità di ritardo. L’importante è che la linea blu non vada mai sopra quella nera (in tal caso chiedo più dati di quanti ne ho ricevuti).

Se il server spedisse con una frequenza maggiore di quella di riproduzione, il buffer crescerebbe sempre di più

Esistono tre principali approcci per lo streaming video:

- **Streaming UDP:** *sfrutta il fatto che UDP non ha controllo di congestione e di flusso, per cui il client ha controllo abbastanza preciso sull’invio dei dati.* Il server trasmette i dati con cadenza tale da garantire bitrate appropriato (es. bit rate 2 Mbps e pacchetti da 8000 bit → il server invia un pacchetto ogni 4 ms)

Per via di questo controllo preciso il lato client ha buffer equivalente a pochi secondi di video. Un problema di questo approccio è che però, se si hanno dei cali di banda, i pacchetti non arrivano alla velocità con cui sono inviati e quindi non viene rispettato il vincolo di riproduzione continua. Con UDP è inoltre necessaria un’ulteriore connessione di controllo per

cui il client possa inviare comandi (pausa, avanti etc..) (infatti UDP è senza connessione, il server nemmeno sa se il client riceve il pacchetto)

- **Streaming HTTP:** è come se stessi scaricando un qualsiasi altro oggetto web. Lo scarico via connessione TCP con protocollo applicativo HTTP, essendo però contenuto multimediale ci si aspetta come detto che il player scarichi un po' di dati nel buffer per poi iniziare a riprodurre.

Differenze rispetto a UDP per quel che riguarda trasmissione e ricezione: non posso inviare a frequenze precise come UDP. Se la velocità di ricezione è maggiore del bit rate del video, il buffer del client continua a crescere anche durante la riproduzione (**prefetching**) fino al riempimento, dopodiché il **controllo del flusso** (proprio di TCP) limiterà il tasso di trasmissione al tasso di consumo del client.

Per la questione legata all'utente che può saltare parti del video esiste un header in HTTP chiamato **Range** che permette di specificare da dove scaricare l'oggetto.

Tramite **Streaming dinamico adattivo su HTTP (DASH)** non ho un unico file, ma diversi (ognuno con URL diversa) spezzettati in *segmenti* ciascuno dei quali codifichiamo a vari bitrate, *garantendo la possibilità di scegliere tra versioni con qualità differenti anche durante la riproduzione*.

DASH

Il file video è diviso in più chunk/segmenti ed ognuno di essi è codificato in più versioni, con bit rate differenti. Versioni diverse sono memorizzate in file diversi e ogni file è replicato in nodi diversi **CDN**.

Esiste poi il **file manifest** che fornisce gli URL per i diversi chunk.

Quando voglio fare lo streaming per prima cosa il client prende il manifest file, e decide quale chunk scaricare. Durante lo streaming, stimando periodicamente la banda da server a client (**encoding rate**), può scegliere la versione con il bit rate più alto sostenibile data la lunghezza di banda corrente ed eventualmente scaricare quei chunk invece di quelli attuali.

- “*intelligenza*” sul client: il client determina
 - *quando* richiedere un chunk (in modo che non si verifichi la starvation del buffer o l'overflow)
 - *che encoding rate richiedere (qualità più alta quando c'è più larghezza di banda)*
 - *dove* richiedere il chunk (può richiedere dal server che è "vicino" al client o ha banda larga)



Possiamo quindi dire che lo streaming video può essere implementato su internet combinando *Algoritmi di Codifica* (comprimo il filmato), *DASH* e *buffering di riproduzione* (invece di riprodurre subito aspetto “caricando” dati).

Content distribution networks (CDNs)

Per implementare servizi di streaming sapendo che potrebbero esserci centinaia di migliaia di utenti simultanei, non basta un solo server o un unico grande cluster. Conviene utilizzare un **architettura distribuita geograficamente!**

- **opzione 1:** unico, enorme data center

- singolo punto di rottura (single point of failure)
- punto di congestione della rete
- percorso lungo (e possibilmente congestionato) verso i clienti lontani

.... molto semplicemente: questa soluzione *non è scalabile*

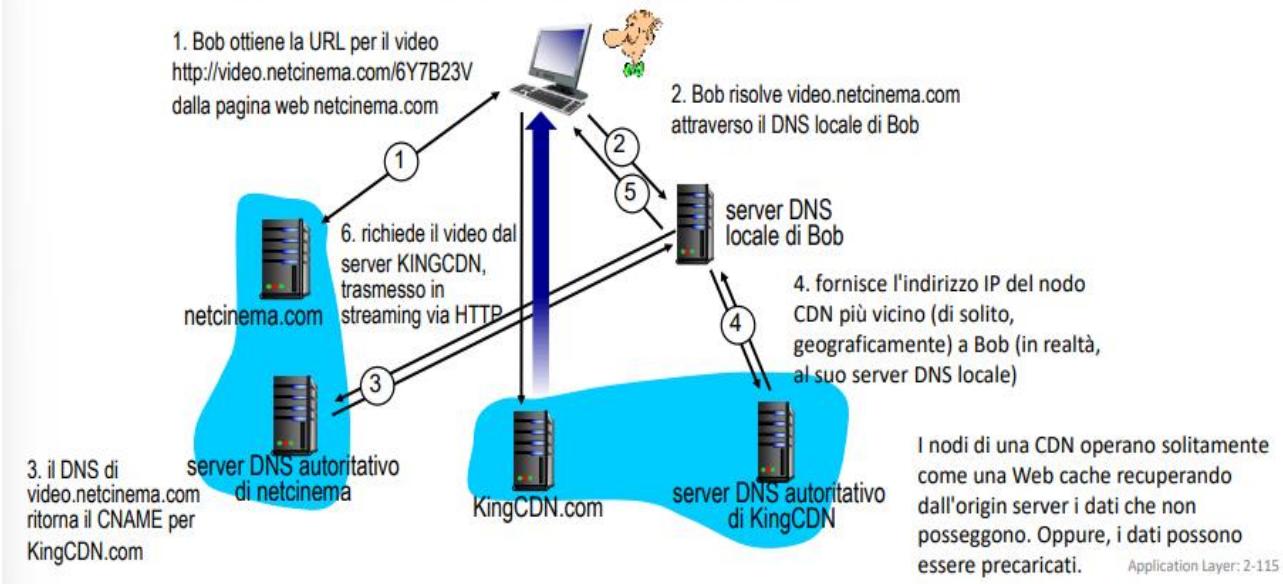
Due approcci per realizzare questa architettura:

- **Enter Deep:** i server CDN sono collocati dentro le reti di accesso in modo di essere molto vicini agli utenti garantendo maggiori prestazioni. Il prezzo da pagare è *maggior complessità gestionale e di manutenzione* (es. Akamai gestisce oltre 240k server in più di 120 Paesi)
- **Bring Home:** i server CDN sono sparsi in *pochi grandi cluster* (decine ad esempio) in **IXP** (internet exchange point) vicino alle reti di accesso (es. usato da Limelight)

Vediamo come funziona una CDN. Ipotizziamo che Bob voglia vedere un video su NetCinema. Contatta l'homepage dove ottiene l'URL del filmato. Il DNS locale di Bob risolve (trova l'ip) dell'URL in questione contattando il server autoritativo di netcinema che risponderà con un CNAME (che permette di ricavare da un host alias il suo vero nome canonico). Questo nome canonico punta al server autoritativo della CDN, che viene quindi contattato dal server DNS locale di Bob e gli fornisce l'IP del server che gli consentirà di vedere il filmato.

Bob (client) richiede il video <http://video.netcinema.com/6Y7B23V>

- video memorizzato sulla CDN a <http://netcinema.KingCDN.com/6Y7B23V>



Il server autoritativo della CDN può stabilire, in base al DNS server locale che lo contatta, *fornire l'IP del cluster più vicino a quel DNS* (garantendo un servizio migliore).

Anche Netflix usa una CDN, *la differenza sta nel fatto che la opera lui direttamente*. Netflix memorizza le copie dei suoi contenuti su vari nodi sparsi in tutto il mondo della sua **CDN OpenConnect**.

Quando l'abbonato richiede il contenuto, il fornitore di servizi restituisce il manifesto attraverso il quale il client recupera i contenuti alla massima velocità supportabile. Il client può inoltre scegliere una copia diversa se il percorso di rete è congestionato.

La cosa interessante della CDN di Netflix è che **si osserva come questa non possieda nulla a livello di Nucleo della Rete**. Per Netflix Internet rappresenta null'altro che un servizio di interconnessione tra host.

Livello di Trasporto

Il livello di trasporto si trova sotto il livello di Applicazione e sopra quello di Rete. Abbiamo finora visto il livello di Applicazione dove risiedono le applicazioni di rete (che sono in esecuzione sugli host, **non nel nucleo**). Il livello di trasporto ci interessa perché le applicazioni sono formate da programmi in esecuzione (**processi**) su host remoti, che si scambiano messaggi (es. GET fiefff.mp4 chiedo un video etc...).

Ma come fa il processo del Browser a inviare una richiesta al processo del server e il server a inviare la risposta? Ha bisogno del livello di trasporto, **che supporta la comunicazione remota tra processi in esecuzione tra host differenti** (si parlano tra loro a tutti gli effetti processi diversi, non gli host).

Il livello di trasporto offre un servizio (**comunicazione logica**) che permette ai processi di comunicare come se le macchine su cui sono eseguiti fossero direttamente connesse, ignorando l'infrastruttura a livello di rete sottostante che permette la materiale trasmissione di dati (in un lungo percorso formato da numerosi collegamenti e commutatori di pacchetto, dall'host mittente al destinatario).

Per questa ragione i protocolli di trasporto sono anche detti end-to-end, poiché si occupano della comunicazione da un capo all'altro della rete.

Il livello di trasporto **è implementato solo nei sistemi periferici** (host), i router non lo implementano.

Con una descrizione ad alto livello, **come è implementato il livello di trasporto?** L'applicazione usa l'interfaccia con il livello di trasporto socket per passare il messaggio a livello di trasporto. Questa può frammentare il messaggio in parti più piccole, combinarla con un'intestazione creando un **segmento** che verrà mandato all'altro capo. Come ci arriva? Viene consegnato a livello di rete che si occupa di trasmettere all'altro capo. Una volta arrivato il livello di trasporto prende dal livello di rete i segmenti e li riassembra in messaggi passandoli al livello di applicazione.

I router nel cammino da un host all'altro operano solo sull'intestazione del datagramma, ignorando il segmento encapsulato al suo interno (quindi non livello di trasporto per router)

Più protocolli di trasporto sono disponibili per le applicazioni (**TCP e UDP**).

TCP affidabile con connessione, UDP non affidabile senza connessione.

- **livello di trasporto:**
comunicazione logica tra
processi
 - si basa sui servizi del livello di rete e li potenzia

- **livello di rete:**
comunicazione logica tra
host

→ multiplexing/demultiplexing



Il fatto di prendere il servizio di rete, che mette in comunicazione host, e trasformarlo in un servizio che permette di mettere in comunicazione diversi processi, richiede il meccanismo di multiplexing e demultiplexing.

Concettualmente è analogo al discorso legato alla commutazione di circuito, in cui due commutatori sono connessi a un singolo collegamento ma lo faccio apparire come fosse multiplo facendoci passare più circuiti.

Allo stesso modo il livello di rete mi permette sì di far comunicare singoli host, ma tramite il meccanismo di multiplexing e demultiplexing posso costruirci sopra un meccanismo di comunicazione tra entità molteplici.

- **UDP:** User Datagram Protocol
 - inaffidabile, consegne senz'ordine
 - estensione "senza fronzoli" di IP: solo comunicazione tra processi e controllo degli errori
- **TCP:** Transmission Control Protocol
 - comunicazione tra processi affidabile, consegne nell'ordine originario
 - controllo di flusso
 - controllo della congestione
 - instaurazione della connessione
- servizi *non* disponibili:
 - garanzie su ritardi
 - garanzie su ampiezza di banda

UDP protocollo **senza connessione** (non ha fase preliminare per stabilirla) per cui i dati potrebbero non essere consegnati nell'ordine corretto o non essere consegnati affatto. Si cerca di ovviare al problema implementando un controllo degli errori. UDP rappresenta quindi un'estensione di IP (protocollo a livello di rete), si occupa solo di mettere in comunicazione due processi (tramite multiplexing e demultiplexing) e fa dei controlli per gli errori.

Multiplexing e demultiplexing rappresentano uno dei servizi più importanti a livello di trasporto da implementare.

Per TCP si possono implementare molti altri servizi per fornire un modello più complesso, ma multiplexing e demultiplexing resta la caratteristica che definisce il livello di trasporto.

TCP è un altro protocollo a livello di trasporto **basato su connessione** (comunicazione preceduta da una fase di handshaking durante la quale, tramite scambio di segmenti di controllo, si conviene che la connessione è stata stabilita).

Si tratta di un protocollo di comunicazione affidabile, per cui *i dati sono consegnati nell'ordine corretto senza buchi o errori*. TCP implementa il **controllo del flusso** (regola la velocità di trasmissione in funzione del ricevente) ed implementa un meccanismo di **controllo di congestione** (anch'esso regola la velocità di trasmissione).

Né UDP né TCP offrono **garanzie sui ritardi** o di **throughput**. Ciò rende complicato come visto (streaming) implementare applicazioni multimediali, che sono sensibili alla temporizzazione e alla banda.

(Si ha congestione nel momento in cui la velocità dei dati in ingresso ad una linea di uscita si avvicina di molto alla banda di quest'ultima, per cui l'intensità di traffico (rapporto) si avvicina ad 1, per cui la lunghezza media del buffer cresce e con essa i ritardi. Nel momento in cui un pacchetto in arrivo trova la cosa piena, o lui o un pacchetto in coda vanno persi.

La congestione si manifesta quindi con ritardi che aumentano e perdita di pacchetti)

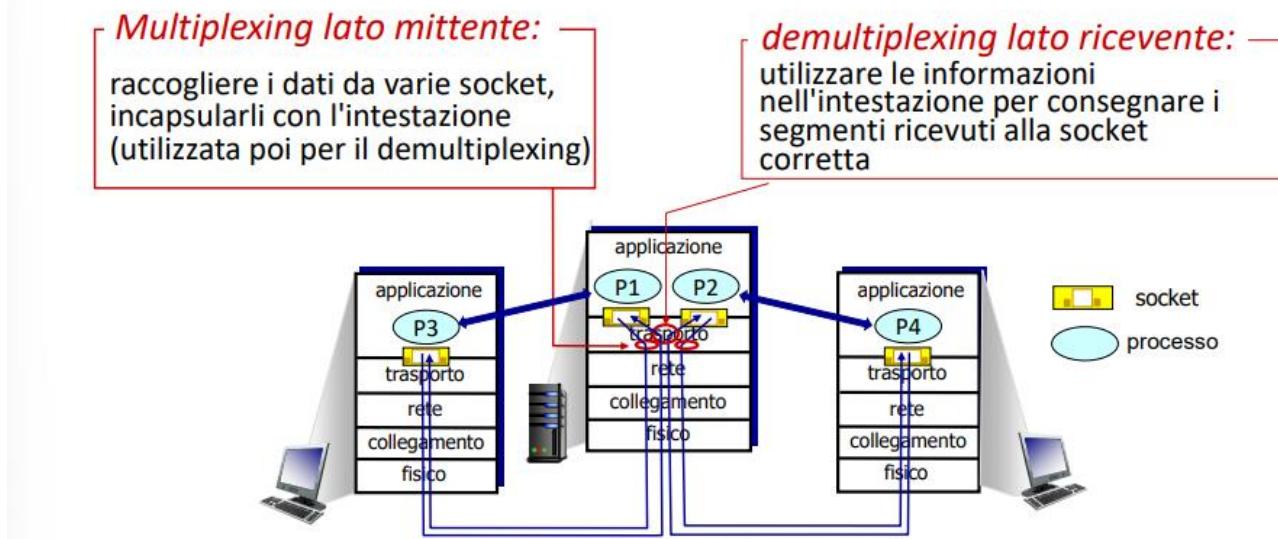
Multiplexing e Demultiplexing

Abbiamo più processi, ciascuno con la propria socket.

Si

Si ha **multiplexing** quando il livello di trasporto *raccoglie i dati da più socket, incapsulandoli con opportuni dati di intestazione* (che serviranno nella fase di demultiplexing) in segmenti, consegnandoli poi a livello di rete.

Il demultiplexing, come il multiplexing, avviene sempre a livello di trasporto. Si ha **demultiplexing** quando il livello di trasporto *riceve dal livello di rete i segmenti, guardando l'intestazione* via demultiplexing il livello di trasporto *deve capire a quali delle socket su quell'host deve fornire i dati*.



Multiplexing come più strade che si uniscono in un'unica più grande, demultiplexing come un'unica strada che si dirama in strade più piccole.

Come implementare il demultiplexing?

Si ricorda che processi comunicanti sono identificati da un indirizzo IP (dell'host su cui sono in esecuzione) e da un numero di porta (del processo specifico).

Parliamo del **datagramma**, che a livello di rete “incapsula” il segmento del livello di trasporto che incapsulava a sua volta il messaggio a livello applicativo. Ogni datagramma presenta un **indirizzo IP del mittente** e un **indirizzo IP di destinazione**, ogni datagramma **trasporta un singolo segmento** ed ogni segmento ha *numero di porta di origine* e *numero di porta di destinazione* (per questo non serve implementare il livello di trasporto nei router).

L'host destinatario, una volta ottenuto il datagramma, *usa gli indirizzi IP e i numeri di porta per inviare il segmento alla socket appropriata.*

Demultiplexing UDP (senza connessione)

Vediamo un esempio pratico in python. Quando creo una socket, si deve specificare il numero di porta.

```
mySocket = socket (AF_INET, SOCK_DGRAM)  
mySocket.bind(('', 9157))
```

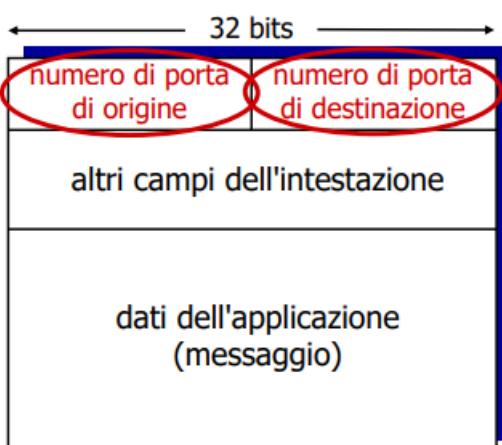
La socket che ho creato è in ascolto sulla porta **9157**. Al posto della stringa vuota dovrei indicare l'indirizzo IP nel caso in cui abbia un host con più schede di rete o indirizzi IP. *Se lascio stringa vuota significa che la socket è in ascolto su qualsiasi indirizzo IP di quella macchina.*

Quando creo un datagramma da inviare ad una socket UDP, devo specificare

- **Indirizzo IP di destinazione**
- **Numero di porta di destinazione**

Quando invio i dati il segmento viene passato a livello di rete e quando l'host lo riceve (UDP) controlla il numero di porta di destinazione del segmento e lo invia alla socket con quel numero di porta. In UDP il numero di porta e indirizzo IP d'origine servono esclusivamente come “indirizzo di ritorno” per un'eventuale risposta.

Pertanto più datagrammi UDP con stesso indirizzo indirizzo IP e numero di porta di destinazione, *anche se provenienti da IP e numeri di porta diversi tra loro, vengono mandati alla stessa socket.*



formato dei segmenti TCP/UDP

Demultiplexing TCP (orientato alla connessione)

Il discorso si fa leggermente più complicato.

La socket TCP è identificata da **4 parametri**:

- **Indirizzo IP di origine**
- **Numero di porta di origine**
- **Indirizzo IP di destinazione**
- **Numero di porta di destinazione**

Stavolta l'host ricevente usa i quattro valori (quadrupla) per inviare il segmento alla socket appropriata.

Il server è in ascolto di connessioni su una **socket passiva** che ha un indirizzo IP con un certo numero di porta, che riceve richieste di connessione da tutti gli host. Quando però viene accettata una connessione, viene creata un'ulteriore socket detta **socket connessa** che è associata ad uno specifico client (con stesso indirizzo IP e numero di porta della socket passiva, ma porta e IP dell'host di origine specifici per chi ha chiesto la connessione di modo da distinguerlo da tutti gli altri).

Riassunto

- Multiplexing, demultiplexing: basato sui valori dei campi dell'intestazione del segmento o del datagramma
- **UDP:** demultiplexing usando (solo) il numero di porta e indirizzo IP di destinazione
- **TCP:** demultiplexing usando la quadrupla di valori: indirizzi di origine e di destinazione, e numeri di porta
- Multiplexing/demultiplexing avviene a *tutti i livelli* (ogni volta che entità diverse vogliono usare i servizi del protocollo di livello inferiore)

Nella creazione di una socket abbiamo usato '' (che in Python equivale a '0.0.0.0' nel caso di IPv4) per indicare qualunque indirizzo IP dell'host; tuttavia, avremmo potuto specificare uno in particolare.

UDP: User Datagram Protocol

- protocollo di trasporto di Internet "senza fronzoli"
- servizio di consegna "best effort" (massimo sforzo), i segmenti UDP possono essere:
 - perduti
 - consegnati fuori sequenza all'applicazione
- **senza connessione**
 - no handshaking tra mittente e destinatario UDP
 - ogni segmento UDP è gestito indipendentemente

Perché esiste UDP?

- nessuna connessione stabilita (che potrebbe aggiungere ritardo)
- semplice: nessuno stato di connessione nel mittente e nel destinatario (perciò un server può gestire più client)
- intestazioni di segmento corte
- senza controllo della congestione
 - UDP può sparare dati a raffica!
- controllo più preciso a livello di applicazione su quali dati sono inviati e quando (utile per requisiti di tasso di invio minimo e ritardi limitati)

Perché usare UDP? **L'assenza di connessione evita i ritardi di connessione.**

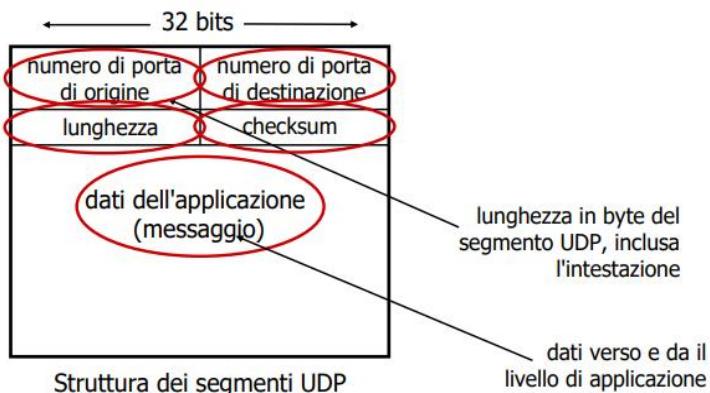
Non è quindi richiesto il mantenimento dello stato e l'assenza di controllo di congestione permette a UDP di inviare dati senza restrizioni.

UDP è utilizzato da protocolli come **DNS, SNMP e HTTP/3** e applicazioni per lo streaming multimediale (tolleranti alle perdite e sensibili alla frequenza).

Esistono inoltre protocolli che permettono di aggiungere affidabilità a livello di applicazione per UDP.

A livello di trasporto per UDP viene generata una **checksum** per controllare l'integrità dei dati a livello di ricezione, inoltre *UDP non frammenta i messaggi applicativi, pertanto è necessario che le applicazioni non eccedano alla dimensione del pacchetto.*

Struttura dei segmenti UDP



Segmento UDP a 32 bit. La prima riga ha 16 bit per il numero di porta sorgente e altri 16 per quello di arrivo. Si hanno poi 16 bit per la lunghezza (che include sia intestazione che dati) e 16 bit per la checksum. Si hanno infine i dati dell'applicazione (il messaggio).

Checksum

Obiettivo: individuare gli “errori” (bit alternati) nel segmento trasmesso.

Immaginiamo di voler trasmettere i numeri 5 e 6. UDP ne calcola la somma (11) prima di spedire il messaggio. Dopodiché trasmette i numeri che voleva mandare, 5 e 6, e poi -11 (in checksum). Si supponga ora che l'host di destinazione abbia ricevuto qualcosa di diverso (es. 4 e 6). Calcola la somma (10) e somma con il checksum (-11), se viene un valore != 0, come in questo caso, significa che ho un errore e quindi chiedo di rimandarmi il messaggio.

NB. se la somma dei messaggi inviati fosse stata ancora 0, non è detto necessariamente che non avessi errore (es. spedisco 5 e 6 e gli arriva 3 e 8, somma comunque 11).

I controlli di errore ci danno sempre quindi una garanzia probabilistica che non vi sia errore. Se il test fallisce ho un errore, se non fallisce non è detto che non ci sia.

Più a basso livello, come implemento davvero il checksum?

mittente:

- tratta il contenuto del segmento come una sequenza di interi da 16 bit (inclusi i campi dell'intestazione UDP e gli indirizzi IP)
- **checksum**: complemento a 1 della somma (in complemento a 1) della sequenza di interi a 16 bit (considerando il campo checksum uguale a 0)
- pone il valore della checksum nel campo checksum del segmento UDP

ricevente:

- calcola la checksum allo stesso modo del mittente (ma sommando anche il valore ricevuto del checksum)
- Il risultato è costituito da tutti bit 1 (-0 nell'aritmetica del complemento a 1)?
 - Sì - nessun errore rilevato. Ma potrebbero esserci errori nonostante questo? Lo scopriremo più avanti....
 - No - errore rilevato
- Altre implementazioni verificano la checksum calcolandola (allo stesso modo del mittente) e confrontandola col valore ricevuto

Transport Layer: 3

Complemento a uno per rappresentare i numeri interi (anche negativi).

La somma tra numeri per la checksum di Internet è diversa tra numeri rappresentati normalmente per un solo motivo: *se alla fine ho un riporto per la cifra più significativa invece di metterlo come nuova cifra più significativa lo sposto all'inizio e lo risommo.*

esempio: sommare due interi da 16 bit

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
a capo	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
somma	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Nota: quando si sommano i numeri, un riporto dal bit più significativo deve essere sommato al risultato

Il complemento a uno di questa somma rappresenta semplicemente l'inversione dei bit. La somma parziale sono i numeri in sé sommati, la checksum il complemento a 1 della somma fatta prima di inviare i dati. Sommando checksum e la somma dei dati ricevuti dovrei ottenere una sequenza di 16 uno perché non vi siano stati (sicuramente) degli errori (perché la checksum complemento a 1 della somma originaria).

Rappresentazione in complemento a uno (4 cifre):

decimale	positivo	negativo
1	0001	1110
2	0010	1101
0	0000	1111
3	0011	1100
7	0111	1000

Se sommiamo un numero e il suo complemento a 1, otteniamo sempre tutti 1.

Lez 8

(fino a 21:00 della rec approfondimento su alcuni comandi della scorsa esercitazione)

Concetto di base del livello di trasporto: *offrire un'astrazione di comunicazione tra processi sopra a un livello di comunicazione offerto dal livello di rete che invece riguarda gli host.* La possibilità di passare da host a processi è detta **multiplexing e demultiplexing**.

UDP è protocollo di trasporto elementare, le uniche cose che infatti aggiunge sono multiplexing e demultiplexing più un minimo controllo degli errori (**checksum**). L'intestazione di UDP è molto piccola, totale 64 bit (8 byte).

Il controllo degli errori avviene tramite interpretazione delle parole a 16 bit come numeri in notazione complemento a 1 e li sommiamo. La differenza è che il riporto dalla cifra più significativa non diventa nuovo 1 più significativo nel risultato ma lo risommo a quella meno significativa del risultato. La somma che otteniamo la complementiamo, ottenendo il checksum da includere. Poiché la checksum di Internet è di fatto uguale alla somma con bit invertiti, quando

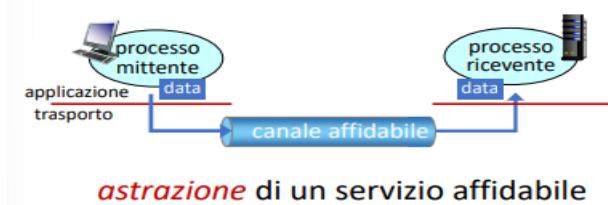
faremo la somma dei dati arrivati al destinatario più la checksum ci aspettiamo di ottenere in risultato una serie di 1 che, in complemento a 1, è -0.

Ricordiamo che con la checksum ci permetteva di dire che se il controllo fallisce l'errore c'è, ma che se invece non fallisce non è detto che comunque non vi sia errore (questione probabilistica).

Trasferimento affidabile di dati

Vogliamo definire **un'astrazione di un servizio affidabile**, attraverso il quale il processo mittente possa inviare i dati a un processo destinatario in modo **affidabile** (cioè *senza errori, in ordine, senza duplicati...*).

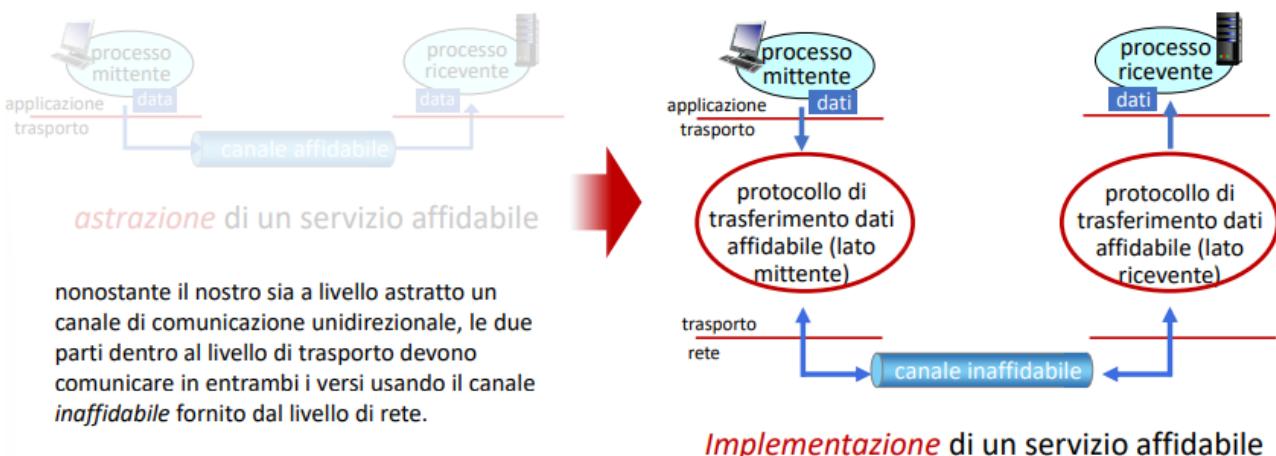
Nel nostro caso, per quel che riguarda il trasferimento affidabile, ci concentreremo inizialmente sul trasferimento unidirezionale di dati (TCP invece protocollo full duplex che permette trasferimento bidirezionale).



Questa astrazione, che vedono le applicazioni, è *implementata dal livello di trasporto*. Concretamente il livello di trasporto è un pezzo di codice eseguito nel mittente e un pezzo di codice eseguito nel destinatario, e operando secondo determinati protocolli tramite uno scambio di messaggi implementeranno l'astrazione. *Mittente e destinatario lavorano ovviamente di fatto da sé, non hanno visione globale e sanno solo ciò che gli arriva.*

Mittente e destinatario comunicano tra loro tramite **il livello di rete, che in Internet non è affidabile** (posso perdere pacchetti, inviarli corrotti etc...). Un'altra osservazione, che si vede nella figura sottostante, è che si hanno frecce bidirezionali.

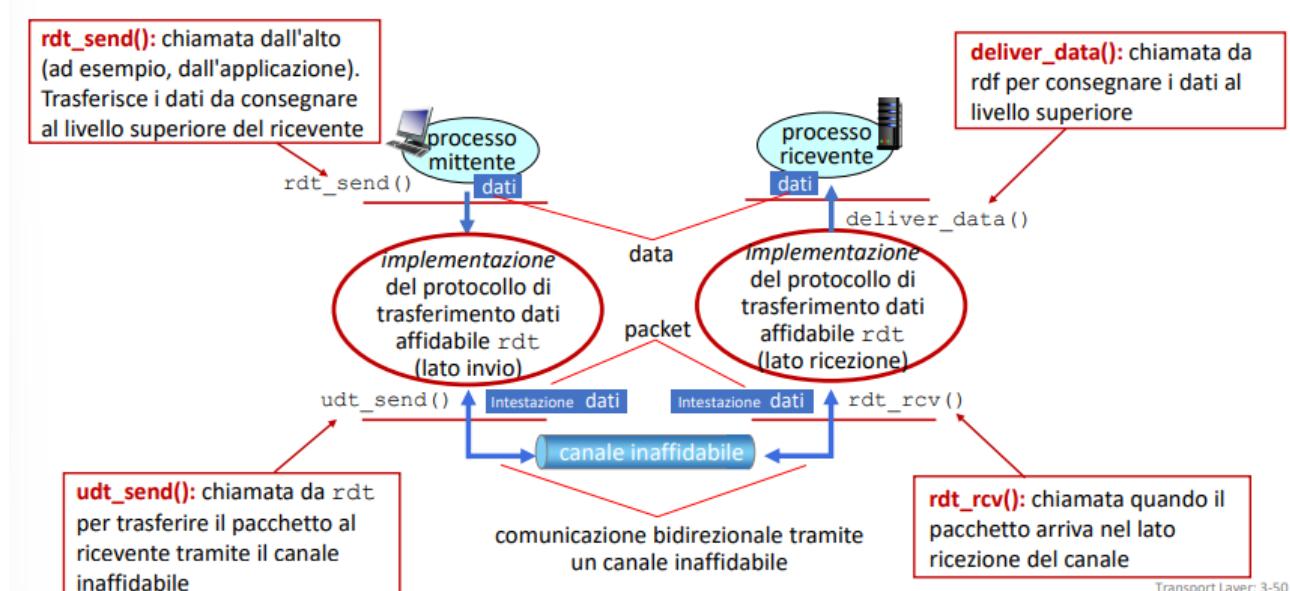
Ciò è dovuto al fatto che, *nonostante a livello logico stiamo considerando una comunicazione unidirezionale da processo a processo, a livello di implementazione di fatto avremo bisogno di scambiarci messaggi nei due versi* (come vedremo).



La principale **complessità** nel protocollo di trasferimento dati affidabile risiede proprio nel fatto che dipende fortemente, per comunicare, dalle caratteristiche di un canale inaffidabile (livello di rete!).

Il mittente e il ricevente, come detto, non conoscono ciascuno lo “stato” dell’altro, se ad esempio il messaggio è stato ricevuto. Come faccio a farglielo sapere? **Comunicandolo attraverso un ulteriore messaggio.**

È necessario che questi elementi a livello di trasporto si scambino dei messaggi il cui scopo è rendere l’altro consapevole del proprio stato



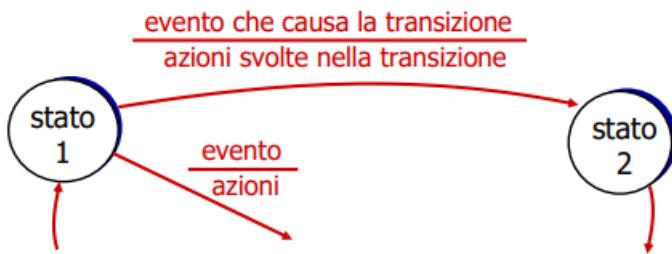
Si farà riferimento alle seguenti API per descrivere il passaggio di dati. A **livello di applicazione** chiamiamo **rdt_send** la funzione chiamata dal

processo per inviare i dati, **`deliver_data`** la callback che manda i dati dal livello di trasporto a quello applicativo (mando via `rdt_send`, al destinatario arriva la `deliver_data`).

A **livello di trasporto** invece (quindi propriamente implementativo) avremo la funzione **udt_send** per il mittente. Dopo che i dati sono stati coronati con l'intestazione per formare un segmento, **udt_send** è la funzione che si occupa di inviarlo tramite il livello di rete. Dall'altra parte è la funzione **rdt_recv()** che si occupa di indicare che il pacchetto è stato ricevuto a livello di rete e che quindi i dati saranno consegnati al livello di trasporto.

Anche il destinatario può usare **udt_send** per inviare i segmenti di riscontro al mittente. (*rdt == reliable data transfer, udt == unreliable data transfer*).

Specificheremo il comportamento del mittente e del destinatario applicando delle **macchine a stati finiti**, con l'idea che uno dei due sistemi (mittente o destinatario) possa trovarsi in uno stato che, dati degli eventi/azioni, possano portarlo ad uno stato diverso. Avremo un arco per ogni transizione, etichettato dall'azione che l'ha causata.



Tramite l'approccio via stati finiti, oltre ad essere meno soggetti a fraintendimenti rispetto ad un'eventuale versione testuale, è possibile provare che il protocollo specificato abbia determinate proprietà (come vedremo).

Canale affidabile (rdt 1.0)

rdt == reliable data transfer protocol

Iniziamo dall'approccio più semplice possibile, immaginando (cosa non vera) che **il livello di rete sia affidabile**. Si hanno in questo caso macchine a stati finiti banali: *il mittente incapsula i dati in un pacchetto che viene inviato tramite il livello di rete. Il destinatario, ricevuto il pacchetto, ne estrapola i dati e li consegna al livello superiore.*

Stiamo implementando un canale di trasporto affidabile su un canale di trasporto affidabile, per questo macchine banali, distinte tra mittente e ricevente

(il ricevente nemmeno ha bisogno di reinviare un messaggio di conferma di aver ricevuto il messaggio, dando per scontato che il livello di rete sia affidabile).

- Canale sottostante perfettamente affidabile
 - nessun errore nei bit
 - nessuna perdita di pacchetti
- FSM *distinto* per il mittente e il ricevente:
 - il mittente invia i dati nel canale sottostante
 - il ricevente legge i dati dal canale sottostante



Nota come stiamo chiamando i segmenti pacchetti. Ciò è dovuto al fatto che questi meccanismi possono essere utilizzati a qualunque livello protocollare della pila di internet, qualora si faccia riferimento alla volontà di costruire canale affidabile sopra uno non affidabile (es. una radio inaffidabile per disturbo della frequenza etc... potrei costruirci sopra comunicazione affidabile applicando le stesse logiche).

Canale con errori nei bit (rdt 2.0)

La situazione inizia a complicarsi. Voglio costruire un canale affidabile su uno non affidabile... in particolare come devo comportarmi se i bit possono essere corrotti nell'invio?

Vediamo una corrispondenza reale: quando detto dei numeri al telefono li detto uno alla volta, aspettando che l'altro mi dia conferma. Faccio affidamento su un feedback del destinatario, per essere sicuro che abbia ricevuto correttamente i dati, che può essere sia positivo che negativo.

Allo stesso modo si può cercare di rilevare se il pacchetto ricevuto è corrotto o meno e di conseguenza inviare una notifica al mittente.

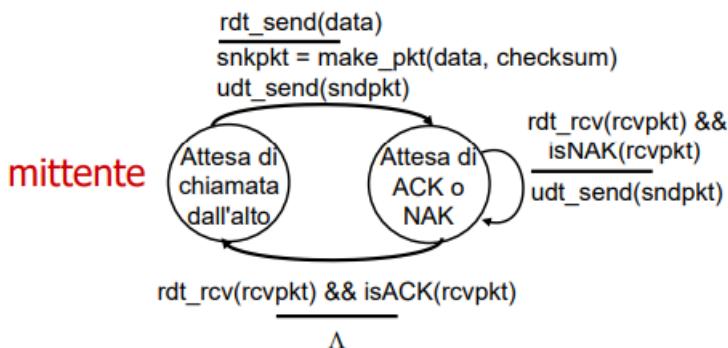
Per capire se i dati sono corrotti posso ad esempio usare una **checksum**. (più in

generale degli *error detection codes*).

In particolare, dal lato mittente, se ricevo un **ACK** (acknowledgment, notifica positiva) ciò significa che il ricevente sta comunicando di aver ricevuto il pacchetto corretto, per cui si continua la trasmissione.

Nel caso invece si riceva un **NAK** (negative acknowledgment) allora il pacchetto che ho inviato conteneva degli errori una volta arrivato al ricevente: quindi **ritrasmetto**.

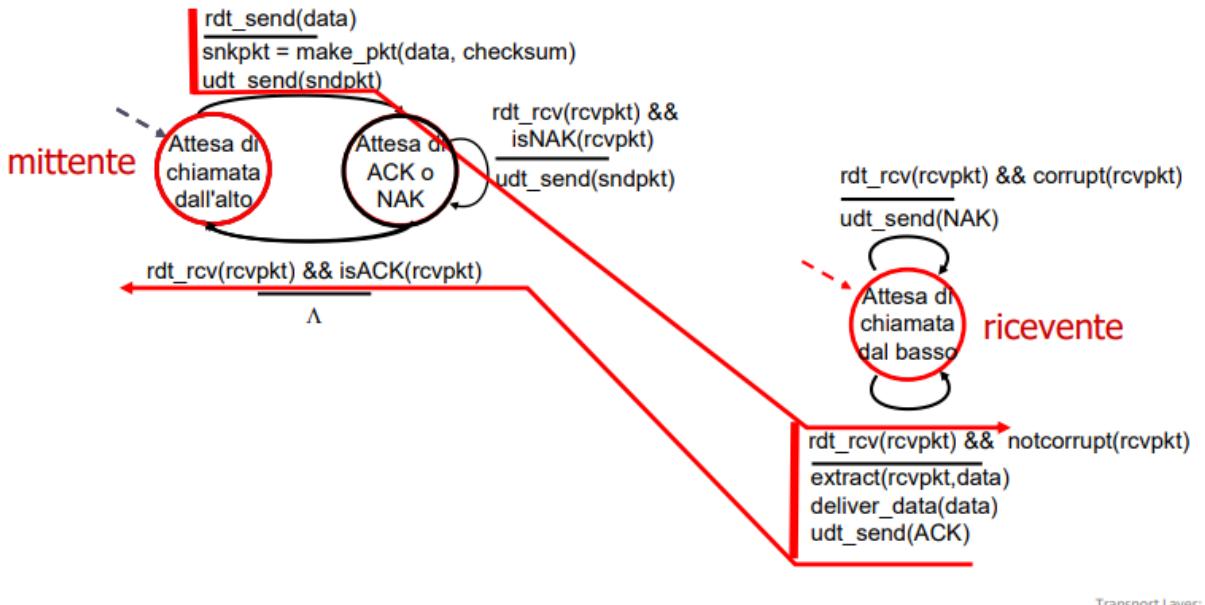
Affinché questo metodo funzioni è necessario che il mittente invii un pacchetto alla volta aspettando sempre, prima di inviare un nuovo pacchetto, la risposta del destinatario. Si parla in questo senso di metodo **stop and wait** (non invio finché non ricevo riscontro, positivo o negativo che sia).



(**Mittente**) Dopo aver ricevuto i dati dal livello applicativo, li trasformo in pacchetto invertendone anche i bit per la checksum, per poi inviarlo. Quando li invio cambio stato perché devo attendere la risposta del destinatario! **Se ACK** allora torno allo stato di prima per inviare un nuovo pacchetto, **se NAK** resto nello stato di attesa e rimando il pacchetto finché non ricevo un ACK.

Si noti come il mittente non sa se il destinatario ha ricevuto correttamente i pacchetti (non sa niente del suo stato) finché non gli arriva un responso! Per questo abbiamo bisogno di protocolli che permettano questi meccanismi di comunicazione.

rdt2.0: operazione senza errori



Transpoert Layer: 3

(Destinatario) Se mi è arrivato il pacchetto presumibilmente senza errori (controllo checksum) allora estraggo i data dal pacchetto e li faccio salire a livello applicativo, inviando poi al mittente il fatto che il pacchetto mi è arrivato correttamente (**ACK**). Nel caso invece di errori non estraggo il pacchetto e mando un **NAK**, aspettando di nuovo che mi venga inviato il pacchetto finché non mi arriva senza errori.

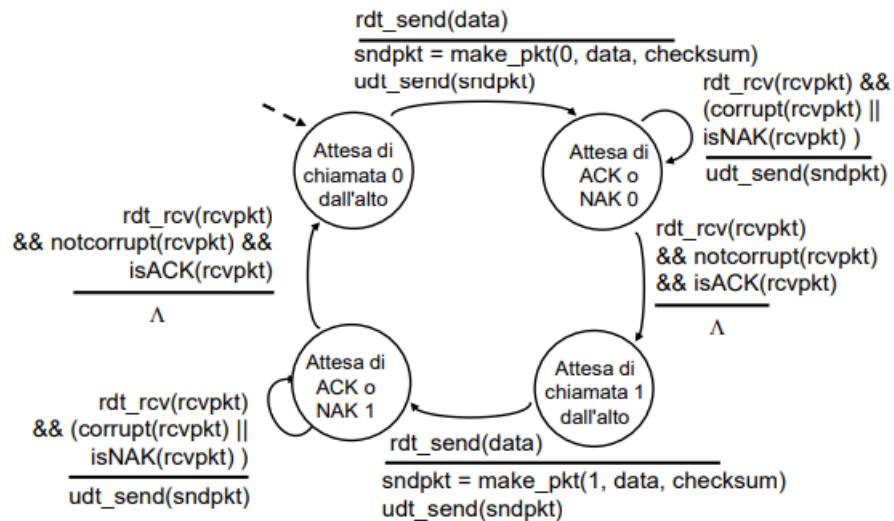
Problema: un assunzione del genere è eccessivamente superficiale. Che succede infatti **se a loro volta i messaggi di ACK o NAK contengono un errore?** A quel punto il mittente non saprà cosa è accaduto e *ritrasmettere potrebbe portare a duplicati*.

Corrispondenza reale: se dettando il numero per telefono l'altra persona non mi dà conferma sulla cifra appena detta, io gliela ridico (ritrasmetto). Ma come faccio ad essere certo che, nel non avermi detto nulla, quella persona non abbia sentito? Potrebbe anche essere che non mi sia arrivata la risposta di conferma e che quindi, nel ridettare, scriva due volte la stessa cifra (da qui il problema dei duplicati).

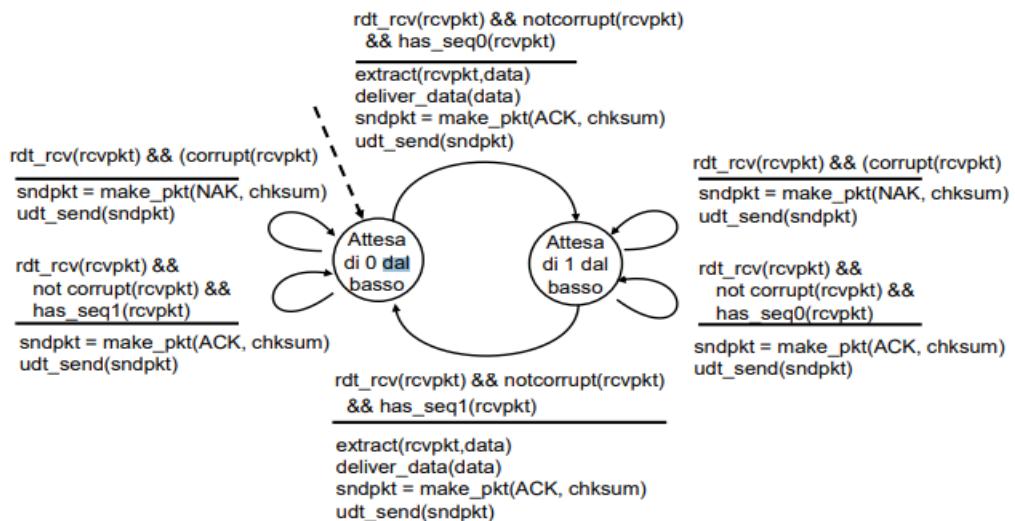
Gestione dei duplicati (rdt 2.1):

- Se ACK o NAK è alterato allora il mittente ritrasmette il pacchetto;
- Il mittente aggiunge un numero di sequenza ad ogni pacchetto, in modo che il ricevitore scarti (non consegna a livello applicativo) il pacchetto duplicato.

Mittente:



Ricevente:



(ancora una volta protocollo in stop and wait)

Perché mi basta sequenziare i pacchetti con due valori, 0 o 1? Perché, essendo il protocollo di tipo stop and wait, mi basta semplicemente distinguere (da destinatario) se sto ricevendo il pacchetto precedente duplicato oppure quello corrente. Per rdt 2.1 teoricamente non è

necessario registrare i NAK o ACK con la sequenza corrispondente, poiché l’eventualità di un duplicato è totalmente gestita dal destinatario con i suoi due stati.

(**Mittente**) Il mittente trasmette il pacchetto con numero di sequenza 0, transendo in uno stato in cui attende ACK o NAK. Se ricevo ACK allora passa ad un nuovo stato dove è pronto ad inviare un nuovo pacchetto, con stavolta numero di sequenza 1. Dopo aver inviato il pacchetto 1 passa allo stato in cui attende ACK o NAK, se riceve ACK, torna allo stato “iniziale” dove trasmette il pacchetto con numero di sequenza 0 etc... (4 stati totali!).

Se invece, trovandomi in uno dei due stati di attesa di ACK/NAK, ricevo NAK allora rimando il pacchetto.

(ricorda che se il mittente riceve ACK/NAK corrotti allora ritrasmette)

(**Destinatario**) Mi bastano stavolta due stati: il destinatario deve solo discriminare infatti pacchetto corrente o duplicato. Gli stati codificano il numero di sequenza che mi aspetto, inizialmente 0. Se ricevo un pacchetto non corrotto che ha numero di sequenza 0 prendo i dati, li consegno all’applicazione e invio un’ACK 0. Transito quindi nello stato di attesa di pacchetto con numero di sequenza 1, dove succede la stessa cosa (se arriva pacchetto 1 allora mando ACK 1 e transito nell’altro stato).

Posso avere dei duplicati in questo protocollo quando ho ricevuto ad es. il pkt 0, mando ACK corrotto e quindi il mittente mi rimanda il pkt 0. A quel punto però, essendo transitato nello stato di attesa del pkt 1, so che il pkt 0 è duplicato e quindi lo scarto (non lo mando al processo applicativo) mandando poi un ACK che dica al mittente di inviarmi il pkt 1.

Riguardo i NAK, si procede come per rdt 2.0: mando NAK solo dal momento che ho ricevuto pacchetti corrotti. Se il NAK è a sua volta corrotto, il mittente rimanda il pacchetto giusto comunque e quindi ok.

mittente:

- aggiunge il numero di sequenza al pacchetto
- saranno sufficienti due numeri di sequenza (0,1). Perché?
- deve controllare se gli ACK/NAK sono danneggiati
- il doppio di stati
 - lo stato deve "ricordarsi" se il pacchetto "corrente" ha numero di sequenza 0 o 1

ricevente:

- deve controllare se il pacchetto ricevuto è duplicato
 - lo stato indica se il numero di sequenza previsto è 0 o 1
- nota: il ricevente *non* può sapere se il suo ultimo ACK/NAK è stato ricevuto correttamente dal mittente

Stesso protocollo senza NAK: rdt 2.2

Si hanno le stesse funzionalità di rdt 2.1, utilizzando però **solo gli ACK**. *Invece di inviare NAK, il destinatario invia infatti un ACK dell'ultimo pacchetto ricevuto correttamente* (includendo, come per 2.1, il numero di sequenza di pacchetto nell'ACK). In questo modo il mittente, che *riceve ACK duplicato*, si comporta sostanzialmente come avrebbe fatto se gli fosse arrivato un NAK: *ritrasmette il pacchetto corrente!*

Come vedremo il procollo **TCP** utilizza questo approccio senza NAK.

Canali con errori e perdite: rdt 3.0

Esiste un’ulteriore possibilità: che succede se, oltre ad esserci la possibilità di invio di duplicati e pacchetti corrotti, accade che il pacchetto inviato viene proprio perso?

La checksum, i numeri di sequenza, le ACK e le ritrasmissioni aiutano... ma non sono sufficienti.

Approccio: il mittente attende un ACK per un tempo “ragionevole”

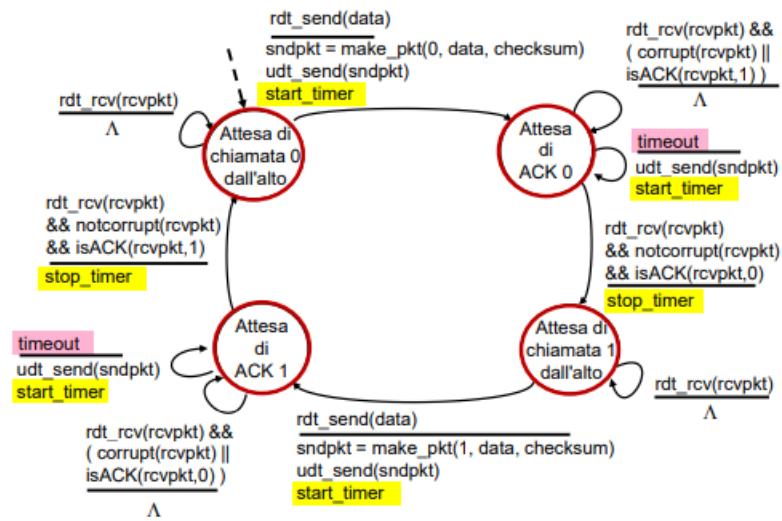
(attende quindi conferma del fatto che il pacchetto inviato sia arrivato a destinazione)

- Ritrasmette il pacchetto se il tempo è stato superato (**timeout**)
- Se il pacchetto (o l'ACK) non sono stati persi, ma sono arrivati dopo il timeout:
 - La ritrasmissione sarà duplicata, ma l'uso dei numeri di sequenza gestirà la cosa

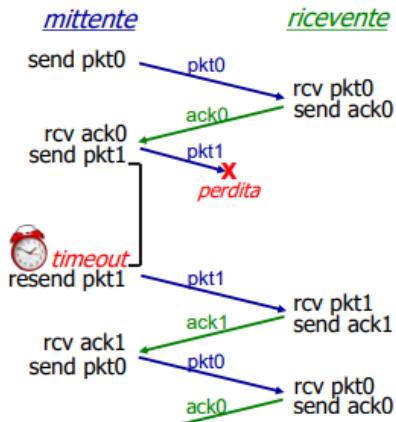
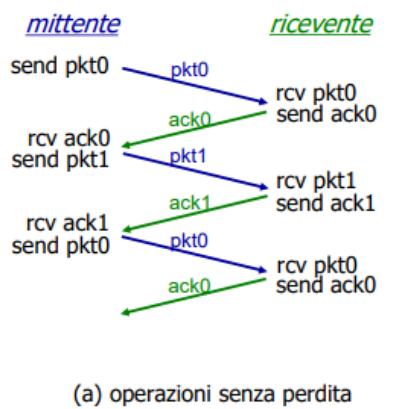
Si utilizza quindi un timer per il conto alla rovescia (countdown) per interrompere (interrupt) dopo un periodo di tempo prestabilito.

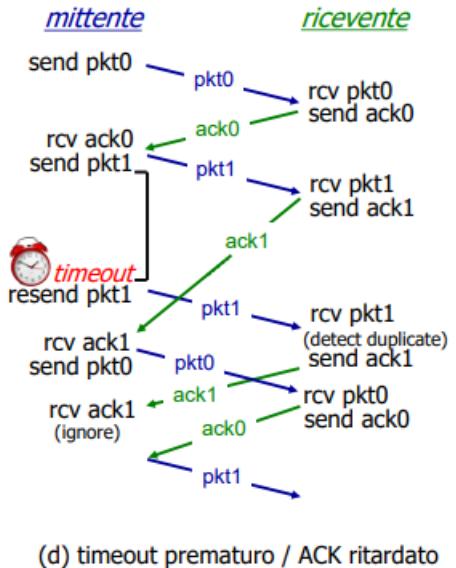
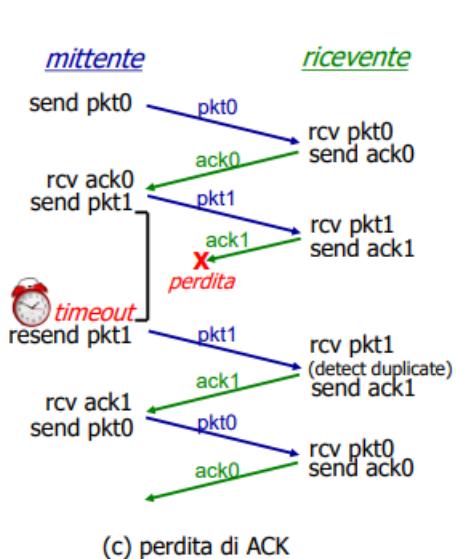
In poche parole, quando il mittente invia un pacchetto di tipo 0 fa partire il countdown via start_timer e passa allo stato di attesa di ACK . Non appena arriva l'ACK ferma il countdown e passa allo stato per inviare il pacchetto di tipo 1 e, mandato il pacchetto, riparte il countdown passando al quarto stato (attesa ACK di tipo 1). Se in uno dei due stati di attesa si arriva al timeout allora si rimanda il pacchetto corrente e riparte il timer.

rdt3.0 mittente



rdt3.0 in azione





Transport

Prestazioni di rdt 3.0 (stop and wait)

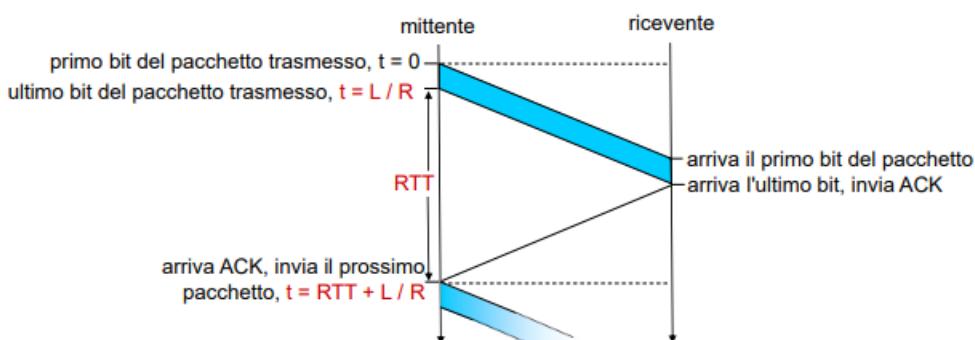
Anche questo è un protocollo di tipo **stop and wait**, che ha pessima utilizzazione.

U_{mittente}: **utilizzazione** == **frazione di tempo in cui il mittente è stato effettivamente occupato nell'invio del bit sul canale.**

Es. Immaginiamo un collegamento da 1 Gbps, con ritardo di propagazione di 15 ms, e pacchetti da inviare di 1000 byte. (nel calcolo finale 30.008 sotto perché $15\text{ms} \times 2 + L/R$) Tempo per trasmettere un singolo pacchetto sul collegamento:

$$D_{trasm} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \mu\text{s}$$

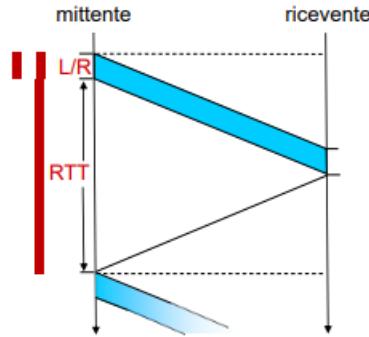
Il primo bit del pacchetto è trasmesso a $t = 0$, l'ultimo a $t = L/R$. Quando arriva l'ultimo bit del pacchetto al destinatario questo deve inviare un ACK, che arriva al mittente a $t = RTT$ (roundtrip time). Per cui il secondo pacchetto viene inviato dall'utente al tempo $t = RTT + L/R$



$$U_{\text{mittente}} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.000267$$



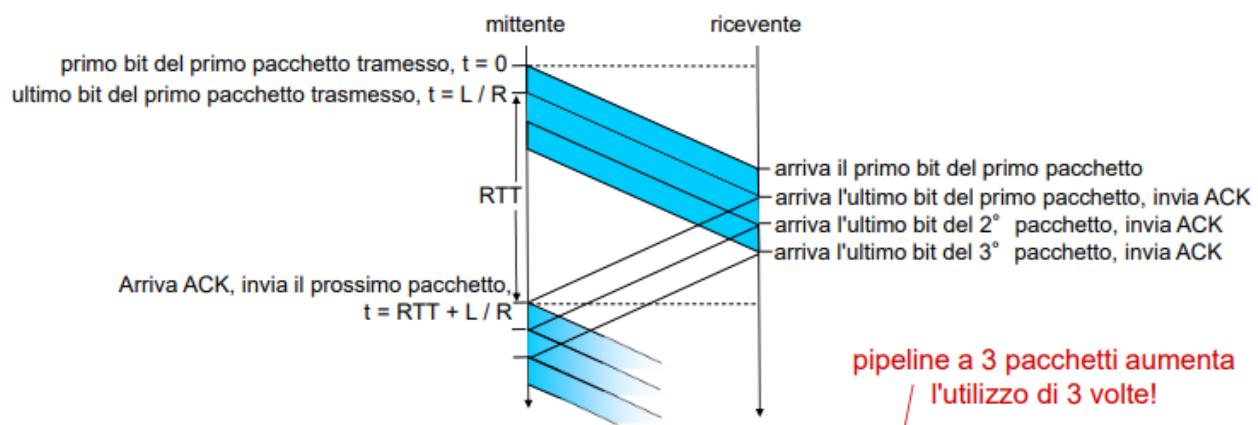
- il throughput effettivo è $L / (RTT + L/R) = U_{\text{mittente}} \cdot R = 267 \text{ kbps}$
- le prestazioni del protocollo rdt 3.0 fanno schifo!
- il protocollo limita le prestazioni dell'infrastruttura sottostante (canale)

Transp

Il mittente è stato impegnato a trasmettere bit in una frazione di tempo irrisoria (U_{mittente}). Ciò è dovuto al fatto che tocca aspettare l'ACK.

Come ovviare al problema che stiamo usando il canale solo per una minima frazione del tempo? Allo stesso modo che in HTTP, si può adoperare il **pipelining**: il mittente ammette più pacchetti in transito contemporaneamente, senza attenderne il riscontro uno alla volta come per il classico **stop and wait**.

Il pipelining migliora l'utilizzo perché invece di inviare un singolo pacchetto e aspettare di ricevere l'ACK corrispettivo, mando ulteriori pacchetti nell'attesa! Vediamo nel pratico come migliora la situazione inviando 3 pacchetti invece di 1 in pipeline:

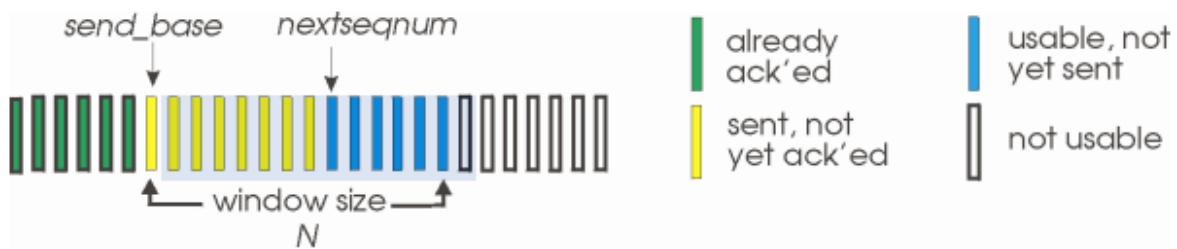


$$U_{\text{mittente}} = \frac{3L/R}{RTT+L/R} = \frac{0.0024}{30.008} = 0.00081$$

Il problema di attuare questo cambiamento è che, ovviamente, *l'algoritmo di stop and wait non funziona più correttamente*. È necessario complessificarlo ed esistono in questo senso due approcci: **Go-Back-N** e **Select-Repeat**.

Go-Back-N

Il mittente può inviare fino a N pacchetti non riscontrati, dove N è detta dimensione della finestra. **Per farlo adotta più numeri di sequenza per identificare i pacchetti inviati** (non più solo due).



Nella window size avrò i pacchetti inviati ma non ancora riscontrati (giallo) e quelli che posso e devo ancora inviare. Il send_base rappresenta in questa sequenza il primo giallo e il nextseqnum (numero di sequenza del prossimo pacchetto) il primo blu.

(**Mittente**) A differenza dello stop and wait, qui non abbiamo al più un singolo pacchetto da riscontrare ma N (dimensione della finestra). L'algoritmo applica il metodo del **Riscontro Cumulativo**: **ACK(n)** starà a significare che i pacchetti riscontrati fino a quel momento sono stati pacchetti con numero di sequenza minore o uguale a n. Alla ricezione di ACK(n) quei pacchetti sono tutti “verdi” (ho avuto la conferma che sono arrivati correttamente) e viene quindi spostata la finestra per iniziare l’invio dei nuovi pacchetti iniziando con il pacchetto n+1.

L’algoritmo è detto Go-Back-N perché *se c’è un timeout ritrasmetto tutti i pacchetti a partire dal send_base*, ed il timer è impostato di volta in volta per il pacchetto più vecchio in transito.

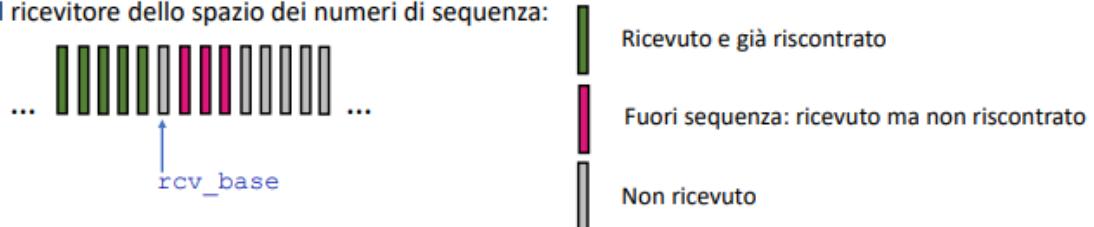
(**Destinatario**) Nel momento in cui riceve correttamente un pacchetto, riscontra sempre il pacchetto precedente per poi passare avanti (se si aspetta e riceve 3 riscontra 2 per identificare il fatto che la sequenza è corretta). Il destinatario non ha quindi bisogno di sapere nulla se non qual è il prossimo pacchetto che deve ricevere, detto **recv_base**. Che succede se invece di ricevere recv_base riceve un altro valore fuori sequenza? Lo ignora e riscontra $recv_base - 1$ (l’ultimo che ha ricevuto correttamente). L’idea di ignorare è dovuta al fatto che per come è

strutturato l'algoritmo, comunque una volta arrivato al timeout il mittente reinvierà la sequenza di pacchetti.

- Alla ricezione di un pacchetto fuori sequenza:

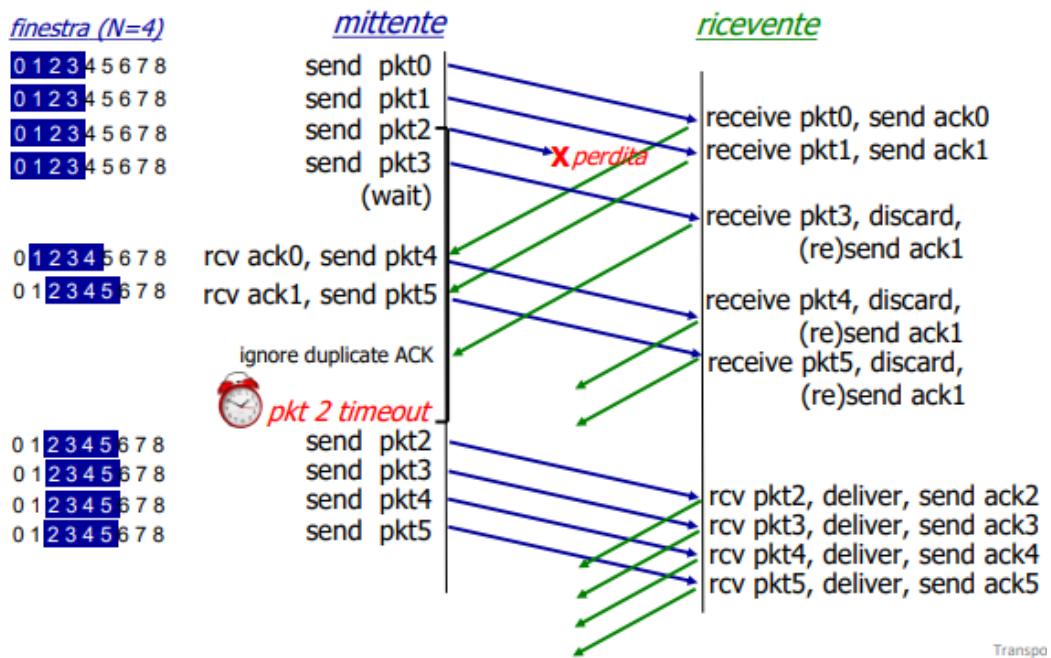
- può scartarlo (non è salvato) o inserirlo in un buffer: una decisione implementativa
- rimanda un ACK per il pacchetto con il numero di sequenza più alto in sequenza

Vista del ricevitore dello spazio dei numeri di sequenza:



NB. Per eventualmente ritrasmettere la sequenza di pacchetti il mittente deve tenere questi ultimi in un buffer, il ricevente deve solo sapere qual è il numero di sequenza che deve attendere (l'uso di un buffer anche per il ricevente è decisione implementativa).

Go-Back-N in azione

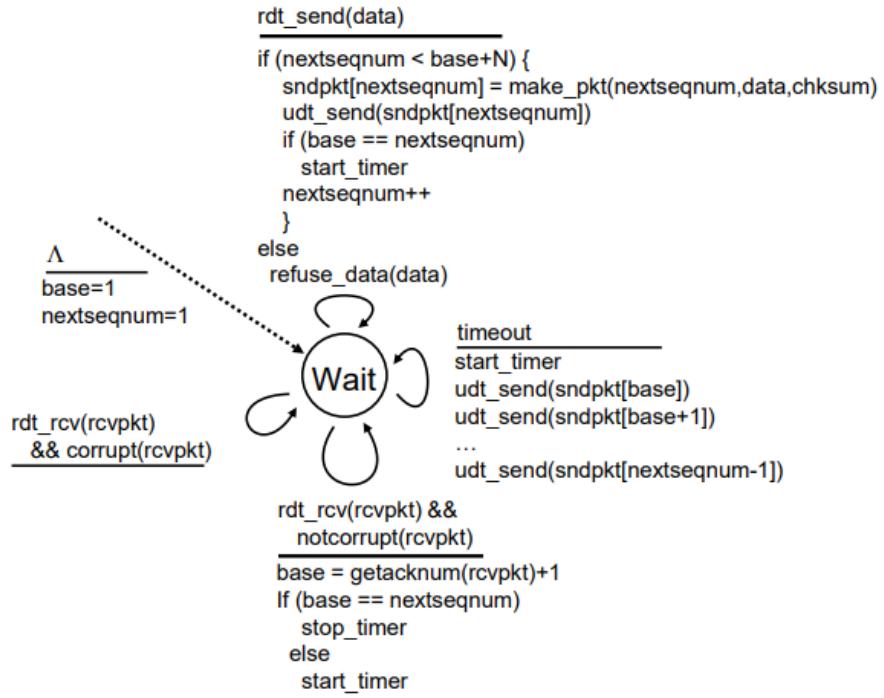


Vediamo in azione l'algoritmo. Il mittente ha dimensione della finestra pari a 4, nell'esempio è perso il pacchetto 2. Dopo aver ricevuto il pacchetto 3, il mittente si accorge che non è il pacchetto che si aspettava e quindi reinvia

l'ACK relativa al pacchetto 1, l'ultimo ricevuto correttamente. Si noti come, per la regola del Riscontro cumulativo descritto prima, solo una volta ricevuto l'ACK del primo pacchetto (0) il mittente invii il pacchetto 4 (dimensione della finestra 4, poteva inviare soltanto i primi 4 pacchetti insieme 0 1 2 3). Inviato il pacchetto 4 ancora una volta il destinatario lo ignora perché fuori sequenza. I pacchetti vengono ignorati fintanto che non arriva il timeout per il mittente di attesa di riscontro per il pacchetto 2, che quindi viene reinviato riprendendo normalmente il ciclo dell'algoritmo.

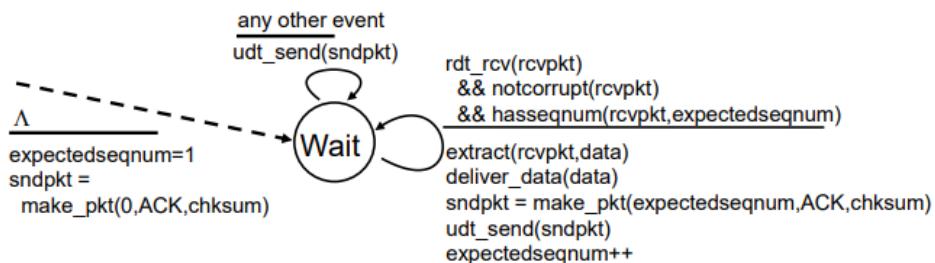
(basso livello, non da sapere):

Go-Back-N: FSM esteso del mittente



Se l'applicazione vuole inviare dati la prima cosa che deve fare un algoritmo pipeline è verificare che la finestra contenga ancora numeri di sequenza utilizzabili, cioè $\text{nextseqnum} < \text{base} + N$ (base sta a significare la base con cui ordino in sequenza i pacchetti, es. base 2 01, base 3 012. Con minore stretto garantisco che il nextseqnum è nella finestra e che quindi posso usare dello spazio per inviare i pacchetti in pipeline). Se ho timeout dopo aver inviato i pacchetti poiché sto ragionando con il GoBackN reinvio tutti i pacchetti della finestra a partire dalla base.

Go-Back-N: FSM esteso del ricevente



Solo ACK: invia sempre un ACK per pacchetti ricevuti correttamente col più alto numero di sequenza *in-ordine*

- può generare ACK duplicati
- deve solo ricordare **expectedseqnum**
- pacchetti fuori ordine:
 - scarta (non salva nel buffer): *nessun buffering lato ricevente!*
 - Re-invia ACK per il pacchetto con il più alto numero di sequenza in ordine

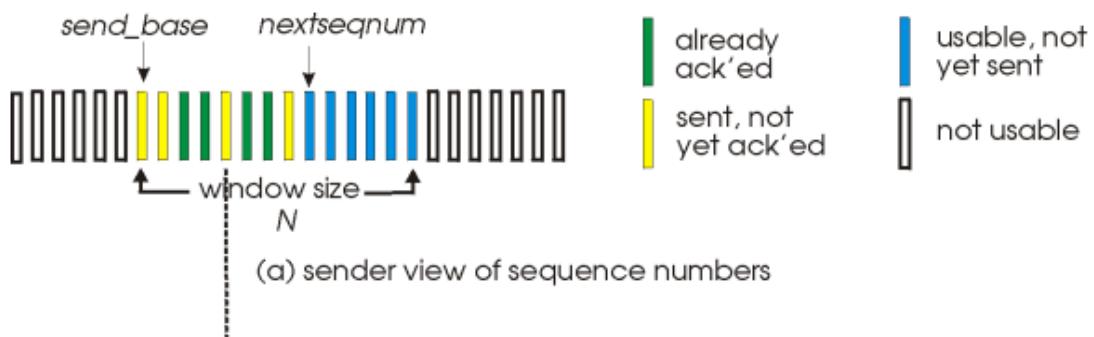
Se il destinatario riceve un pacchetto con il numero di sequenza atteso non corrotto lo estraе, consegna all'applicazione e invia l'ACK per quel pacchetto, incrementando il numero di sequenza atteso. In ogni altro caso manda il riscontro per l'ultimo pacchetto ricevuto correttamente.

Questo FSM indica il comportamento all'inizio della trasmissione, in particolare il lambda maiuscolo (Λ) a sx indica che pacchetto crea anzitutto il ricevente. Essendo in base 1, ci si aspetta che il primo pacchetto inviato dal mittente abbia numero di sequenza 1. L'ACK creata di default e inviata se qualcosa va storto è quindi di tipo 0, di modo che il mittente capisca che deve reinviare. Se invece il pacchetto arriva con numero di sequenza aspettato viene creata ACK con quel valore.

Selective Repeat

Nel Selective Repeat **gli ACK sono selettivi**, ovvero il destinatario riscontra ogni pacchetto ricevuto, uno ad uno.

Per fare questo è necessario mantenere un *timer separato per ogni pacchetto*. Quando un pacchetto va in timeout allora viene ritrasmesso il singolo pacchetto ad esso associato. Inoltre è *necessario un buffer per il destinatario dove possa porre i pacchetti che gli arrivano non in sequenza*. Quando invece gli arriva un pacchetto in sequenza, lo manda direttamente a livello applicativo.



Essendo il riscontro selettivo, potrei avere pacchetti non ancora riscontrati inviati prima di pacchetti riscontrati (vedi giallo prima di verde).

Una volta ricevuti i dati dall'alto (dal livello applicativo), il **mittente** li invia se ha spazio nella finestra. Fa partire il timer alla cui scadenza ritrasmette il pacchetto. Quando riceve un ACK, se è relativo a un pacchetto ancora nella sua finestra, marca il pacchetto come ricevuto. Se ha completato una serie di pacchetti riscontrati, può allora spostare in avanti la finestra. Il **ricevente**, d'altra parte, una volta ricevuto un pacchetto della finestra lo riscontra e lo mette in buffer se non in ordine oppure lo consegna direttamente al livello applicativo se in ordine.

Selective repeat: mittente e ricevente

mittente

dati dall'alto:

- se nella finestra è disponibile il successivo numero di sequenza, invia il pacchetto

timeout(n):

- ritrasmette il pacchetto n , riparte il timer

ACK(n) in [sendbase, sendbase+N-1]:

- marca il pacchetto n come ricevuto
- se n è il numero di sequenza più piccolo, la base della finestra avanza al successivo numero di sequenza del pacchetto non riscontrato

ricevente

pacchetto n in [rcvbase, rcvbase+N-1]

- invia ACK(n)
- fuori sequenza: buffer
- in sequenza: consegna (vengono consegnati anche i pacchetti bufferizzati in sequenza), la finestra avanza al successivo pacchetto non ancora ricevuto

pacchetto n in [rcvbase-N, rcvbase-1]

- ACK(n)

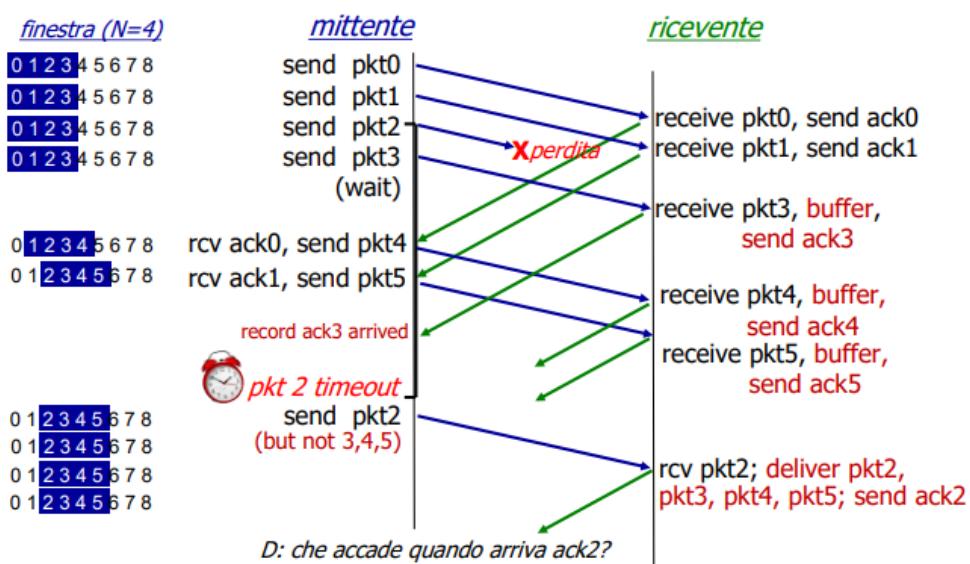
altrimenti:

- ignora

Il ricevente deve riscontrare pacchetti con numero di sequenza al di sotto di $rcvbase$ (già consegnati all'applicazione), per far avanzare la finestra del mittente (rimasta indietro perché gli ACK non sono stati ricevuti)

Trasmettitore 2.81

Selective Repeat in azione



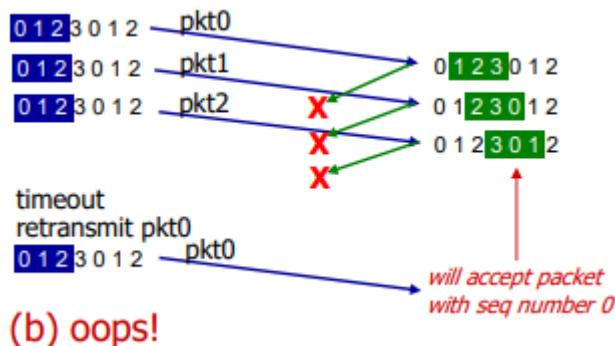
A differenza del Go-Back-N, quando è perso il pacchetto 2 arrivato il pacchetto 3 al ricevente non riscontro il 2, semplicemente metto il 3 in buffer perché non è in sequenza! Continua il ciclo fintanto che per il mittente il riscontro per il

pacchetto 2 arriva al timeout, a quel punto reinvia il pacchetto 2 che viene mandato direttamente a livello applicativo (non in coda, la sequenza per 2 è giusta perché il mittente ha già ricevuto 0 e 1 all'inizio).

Ma che succede una volta arrivato ACK 2 al mittente?

Si sposta la finestra per inviare altri pacchetti. C'è però un **problema**. Non si hanno ovviamente numeri di sequenza infiniti, ma si utilizza **un'aritmetica modulare** (modulo n).

Bisogna adattare la dimensione della finestra con il modulo della nostra finestra, per fare in modo che non si creino situazioni come la seguente:



In questo caso ho dimensione della finestra pari a 3 e numeri di sequenza modulo 4 (0123 0123 etc..).

Nell'esempio sono stati inviati i pacchetti 0 1 2 e l'ACK di riscontro è stato perso per tutti i tre pacchetti. **Il problema è che questo il destinatario non lo sa**, quindi questo ha già spostato la sua finestra, mentre il mittente ancora no. Per cui, una volta terminato il timeout del pacchetto 0, il mittente lo reinvia e il destinatario, ricevendolo, non è in grado di riconoscere il fatto che è un duplicato e quindi lo "accetta" (pensa sia uno 0 successivo, e non il duplicato di quello che ha già ricevuto).

Ma quindi quale relazione è necessaria tra la dimensione di numeri di sequenza e la dimensione della finestra per evitare uno scenario simile?

La finestra non deve essere più grande della metà del modulo dei numeri in sequenza (se ho pacchetti numerati 0 1 2 3, la finestra deve essere al massimo 2).

Lez 9

Riassunto

Trasferimento dati affidabile: voglio creare un canale di comunicazione tra due entità tc i dati che trasmetto arrivano a destinazione senza alterazioni, perdite e nell'ordine in cui sono stati inviati (non voglio naturalmente duplicati).

Abbiamo introdotto questo concetto per TCP a livello di trasporto, ma posso applicare i seguenti principi per avere un trasferimento dati affidabile anche in circostanze diverse (es. collegamenti wireless, comunicazione radio etc...). Per questo quando parliamo dei dati scambiati non abbiamo fatto riferimento direttamente ai segmenti del livello di trasporto ma abbiamo usato il termine più generico “pacchetto”.

Nonostante stiamo astraendo una comunicazione da mittente a destinatario, per far sì che funzioni l'affidabilità è necessario che il destinatario invii delle notifiche di conferma (positive o negative, ACK o NAK), per cui in realtà più a basso livello la comunicazione è bidirezionale. È necessario l'uso delle notifiche perché mittente e destinatario non possono mai sapere lo stato dell'altro in ogni momento, per cui hanno bisogno di un feedback.

Abbiamo visto diverse implementazioni in base a quanto il canale sottostante sia affidabile o meno.

Nel caso di canale affidabile sottostante (rdt 1.0) oltre ad assumere che i pacchetti arrivino al destinatario integri, in ordine e senza duplicati assumiamo anche che il ricevente sia in grado di sostenere il tasso con cui riceve i dati, senza mai dover chiedere al mittente di rallentare (**controllo del flusso**).

Rdt 3 è la variante più sofisticata poiché gestisce anche le perdite. *Nelle reti la causa principale di perdite è la congestione* (associata al riempimento della coda) oppure per interferenze o pacchetti estremamente corrotti. È introdotto un timer per gestire l'eventualità di perdite: parte il timer quando invio il pacchetto e, se entro un tot di tempo non mi è arrivata l'ACK di questo, lo rimando.

TCP

TCP è, insieme a UDP, uno dei principali protocolli di trasporto di Internet. UDP era inaffidabile e senza connessione, si tratta in pratica di IP a cui abbiamo aggiunto solo checksum e multiplazione/demultiplazione.

TCP è invece un **trasporto affidabile orientato ad un flusso di byte**, cioè il mittente e il destinatario si scambiano un flusso di byte. TCP non ha “**nessun confine ai messaggi**” a differenza di UDP, che inviava datagrammi con un confine ben delineato (mentre con TCP si manda un flusso generico di byte). Tuttavia, poiché le applicazioni inviano messaggi, *quando si usa TCP a livello applicativo deve esserci un modo per capire quando finisce un messaggio*.

Si tratta di un **protocollo punto-punto** poiché *lavora con singolo mittente e singolo destinatario* (esistono altri protocolli con multicasting, dove un singolo mittente può inviare a più destinatari).

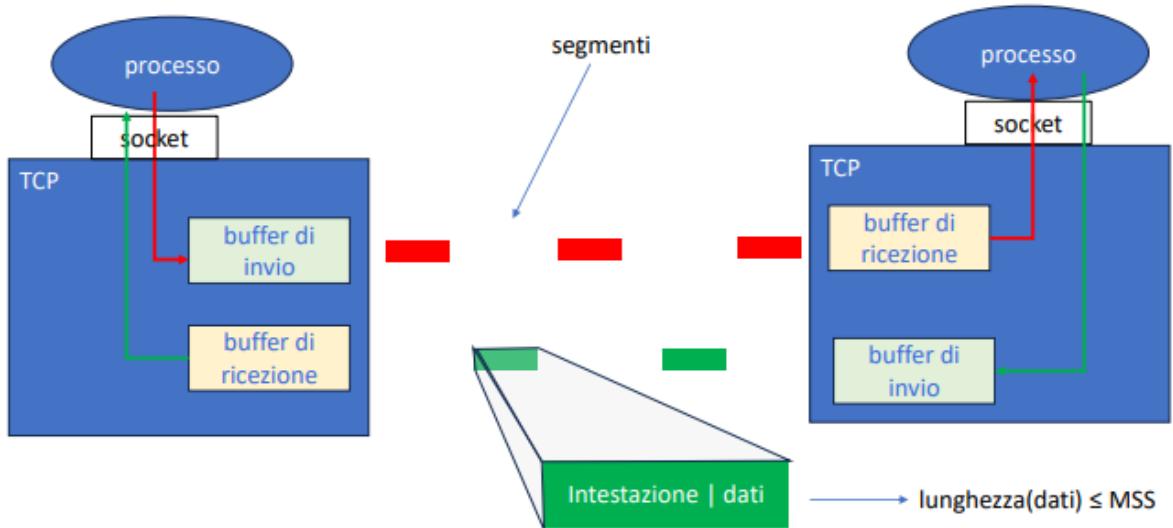
Lavora con un algoritmo **full duplex**, cioè *i dati possono fluire in un verso e nell'altro nello stesso istante e nella stessa connessione*. Per l'affidabilità utilizza **ACK cumulativi**, un po' diversi da quanto visto nella lezione scorsa e il **pipelining**. In particolare *la dimensione della finestra per il pipelining è definita dal controllo di flusso e di congestione* (quindi TCP contribuisce al controllo di flusso, non sovraccarica cioè il destinatario, e il controllo della congestione, riduce cioè la velocità di invio in funzione della congestione di rete).

TCP è **orientato alla connessione**, cioè prima di poter trasmettere le parti devono accordarsi attraverso un **handshaking a tre vie** (scambio di messaggi di controllo, tre vie perché sono 3 segmenti speciali) necessario a inizializzare gli stati di entrambi prima di iniziare lo scambio di dati.

La connessione TCP è diversa dai circuiti della commutazione a circuito. Mentre la commutazione a circuito è una tecnologia dedita allo sviluppo di una rete fatta di commutatori e collegamenti e che quindi utilizziamo nel nucleo della rete, le connessioni TCP sono interamente implementate nei sistemi periferici. Quindi *quando installiamo un circuito in una rete a commutazione di*

circuito questo fatto deve essere noto a tutti i router coinvolti, perché questi dovranno riservare delle risorse per quest'ultimo. Invece essendo le connessioni TCP costrutti nei sistemi periferici sono totalmente opache ai router. (i router lavorano fino a livello di rete, non gli interessa ciò che accade a livello di trasporto).

TCP: Buffer di invio e ricezione TCP.

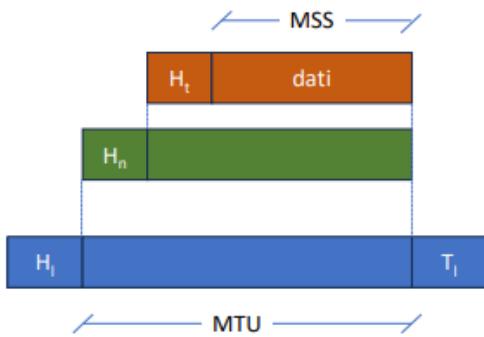


Un processo invia dei dati che vengono collocati in un buffer di invio. Qui periodicamente, secondo una certa logica dettata dall'algoritmo di controllo del flusso, TCP prende dei dati, forma un segmento, poi pacchetto di rete e li spedisce. Ricevuto il pacchetto di rete il destinatario estraе da esso il segmento e lo consegna a livello di trasporto ponendolo in un buffer di ricezione. Verrà poi consegnato a livello applicativo.

Il singolo segmento è formato da un'intestazione e da dei dati applicativi, la cui lunghezza non può essere maggiore di un parametro detto **MSS** (*maximum segment size*).

In generale si vuole che un segmento non venga spezzato in più pacchetti di rete. Per questo si vuole che un segmento stia in un unico pacchetto di rete, che a sua volta deve entrare in un solo frame del livello di collegamento.

TCP: MSS



La dimensione massima del segmento TCP predefinita per IPv4 è di 536. Per IPv6 è 1220. Se un host desidera impostare MSS su un valore diverso da quello predefinito, la dimensione massima del segmento viene specificata come opzione TCP, inizialmente nel pacchetto TCP SYN durante l'handshake TCP.

- MSS + lunghezza(H_t) + lunghezza(H_n) \leq MTU

Valori tipici:

$$\text{lunghezza}(H_t) = 20 \text{ B}$$

$$\text{lunghezza}(H_n) = 20 \text{ B}$$

$$\text{MTU}_{\text{Ethernet}} = 1500 \rightarrow \text{MSS} = 1460$$

- Un host può determinare il MSS guardando la MTU del collegamento locale; ma ciò non offre garanzie circa altri collegamenti intermedi
- Può essere negoziato durante la connessione con l'opzione MSS
- Path MTU Discovery: permette di scoprire il valore più piccolo della MTU lungo il percorso da mittente a destinatario

Se un pacchetto IP eccede la MTU su un collegamento di uscita, il router dovrà frammentarlo (in IPv4) oppure scartarlo (in IPv6)

Ho dei dati di lunghezza MSS. Li metto in un segmento con un'intestazione H_t , metto tutto in un pacchetto con una certa intestazione H_n e metto a sua volta tutto in un frame con intestazione H_l .

Ciò che ci interessa è che il livello di collegamento impone un valore massimo alla dimensione del frame, detto **MTU** (*maximum transmission unit*).

Si ha quindi che **l'MSS + l'intestazione di rete H_n + l'intestazione di trasporto H_l deve essere \leq MTU**.

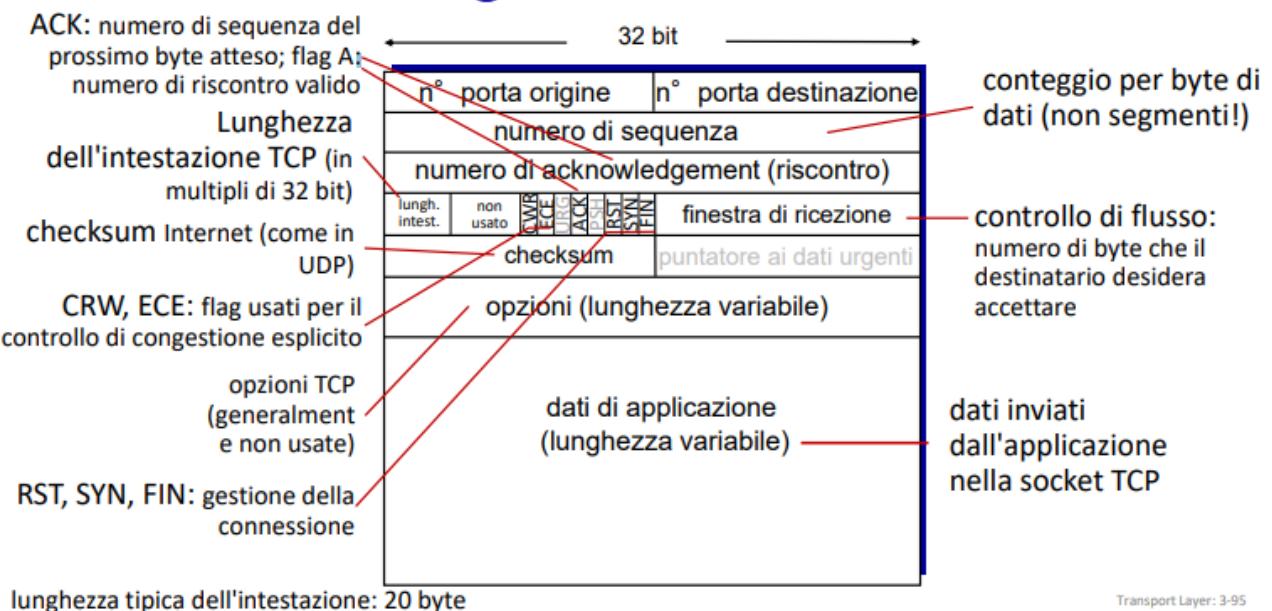
Se si escludono tutti i campi opzionali, sia H_n che H_l sono lunghe 20 byte. Per cui, nel caso di Ethernet, si ha che l'MTU è 1500 \rightarrow MSS al più 1460.

Un host può *determinare un MSS guardando il suo tipo di collegamento e sapendo la sua MTU*. Può inoltre comunicare con il destinatario in fase di handshake per capire se può instaurare un MSS di una certa grandezza (se il destinatario ha collegamento con MTU più bassa, deve ridurre altrimenti il destinatario non reggerebbe). Si può inoltre scoprire l'MTU di tutti i

collegamenti coinvolti nel percorso mittente-destinatario *tramite algoritmi di MTU Discovery*.

Se un pacchetto IP arrivato eccede la MTU, allora viene inoltrato un messaggio al mittente dove si comunica la cosa ed il pacchetto sarà frammentato (IPv4) o scartato (IPv6).

Struttura dei segmenti TCP



Transport Layer: 3-95

Il numero di porta di origine e quello di destinazione servono, come in UDP, per la multiplazione. La differenza tra UDP e TCP per quel che riguarda la multiplazione è che in UDP una socket è identificata solo da un indirizzo IP e dal suo numero di porta, mentre in TCP una socket è identificata sia da IP che da porta propria e dell'altro host con cui comunico.

Si ha poi il **numero di sequenza**: a differenza di ciò che abbiamo visto la lezione scorsa TCP non etichetta i pacchetti, ma **il numero di sequenza è il numero di byte inviati**. Se quindi invio il primo byte il numero di sequenza sarà 0, dopo aver inviato 10 byte il prossimo numero di sequenza sarà 10 etc...

Il numero di ACK è il numero di sequenza del prossimo byte atteso (il primo dopo tutti i byte ricevuti in ordine). Se quindi trasmetto i primi 10 byte (ho quindi un pacchetto con numero di sequenza 0 che contiene 10 byte) non

riscontro 0 o 9, ma riscontro 10 (perché dopo aver ricevuto i byte da 0 a 9, mi aspetto il byte 10).

Checksum per il controllo di errori sui bit. Ho poi delle opzioni di lunghezza variabile, in questo senso necessito sopra 4 numeri per indicare la lunghezza totale dell'intestazione, affinché si possa capire dove finisce l'intestazione e dove finiscono i dati. Questi 4 numeri sono indicati in multipli di 32 bit, per cui 1 non indica 1 bit ma 32 bit (4 byte).

Si hanno poi dei **flag** (come per il controllo di congestione) e una finestra di ricezione necessaria al **controllo del flusso**.

Numeri di sequenza e ACK di TCP

Come in **GoBackN**, il mittente tiene traccia nella sua finestra dei pacchetti già mandati e di cui aspetta riscontro. Poiché aspetta **riscontri cumulativi** non ci saranno dei riscontri selettivi dei pacchetti riscontrati nel mezzo (come nel selective repeat).

Appena riceve riscontro di un byte: tutti i byte fino a quello (escluso) risultano riscontrati e la finestra si sposta a quello successivo.

Quando è inviato un segmento, il sequence number è l'indice del primo byte disponibile. Quando arriva un riscontro significa che il destinatario ha ricevuto tutti i byte precedenti a quello che sto riscontrando (tutto il giallo diventa verde fino al byte riscontrato escluso). È come se il mittente con l'ACK ci stesse dicendo qual è il prossimo byte che attende.

Numeri di sequenza e ACK di TCP

Numeri di sequenza:

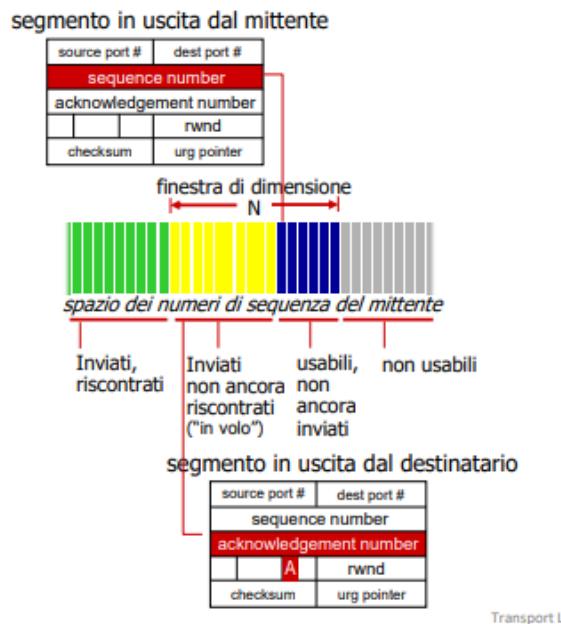
- "numero" del primo byte nel segmento nel flusso di byte

ACK:

- Numero di sequenza del prossimo byte atteso dall'altro lato
- ACK cumulativo
- RFC 2018: Acknowledgment Selettivo (che non studieremo)

D: come gestisce il destinatario i segmenti fuori sequenza?

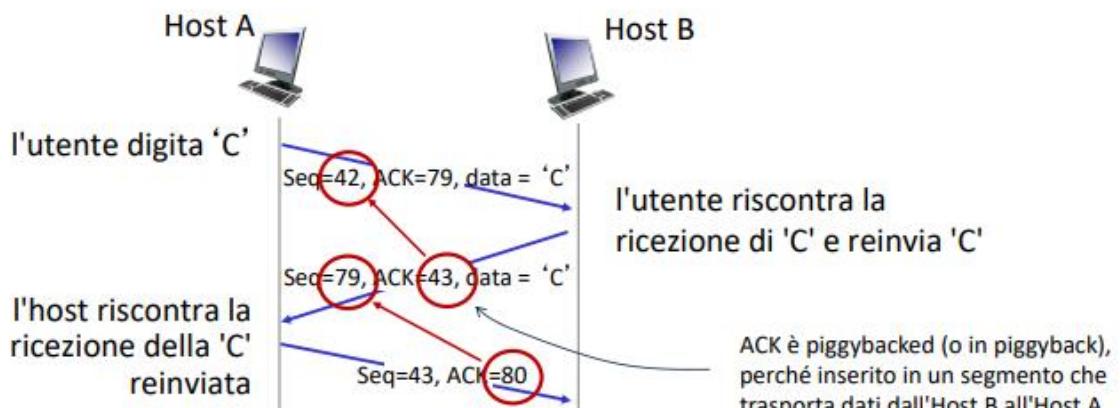
- R: la specifica TCP non lo dice – dipende dall'implementatore



Transport Layer: 3-4

Per gestire i segmenti fuori sequenza si agisce tendenzialmente via buffer.

Numeri di sequenza e ACK di TCP



L'host A invia il byte con numero di seq 42 a B e l'ACK con n di seq 79 (cioè si aspetta quel byte). B invia quel byte e chiede l'ACK del byte successivo, 43. A allora chiede il byte per lui successivo a 79, 80 e invia inoltre il byte 43 richiesto da B.

Ma come stabiliamo il timeout? Dipende tutto dal roundtrip time **RTT** (*tempo trascorso da quando si invia un segmento a quando se ne riceve l'ACK*). Tuttavia RTT non è noto, ma variabile.

Se il timeout è più piccolo di RTT non posso per definizione ricevere riscontro,

quindi in quel caso il **timeout** è troppo piccolo. Se invece il **timeout** è troppo grande allora ritardo nel riparare la perdita. L'ideale è avere quindi timeout vicino a RTT, ma leggermente di più per sicurezza.

Ma RTT, essendo variabile, non lo conosco. Lo devo stimare.

Come lo stimo? Anzitutto il mittente può campionare RTT vedendo il tempo che trascorre tra l'invio di un pacchetto e la ricezione dell'ACK (ciò viene fatto ovviamente solo per i pacchetti non ritrasmessi. Si parla in questo caso di **SampleRTT**. Ma SampleRTT varia, non converrebbe cambiare continuamente il timeout. Ciò che si fa è quindi una **media mobile livellata esponenziale**, relativa alle misure più recenti.

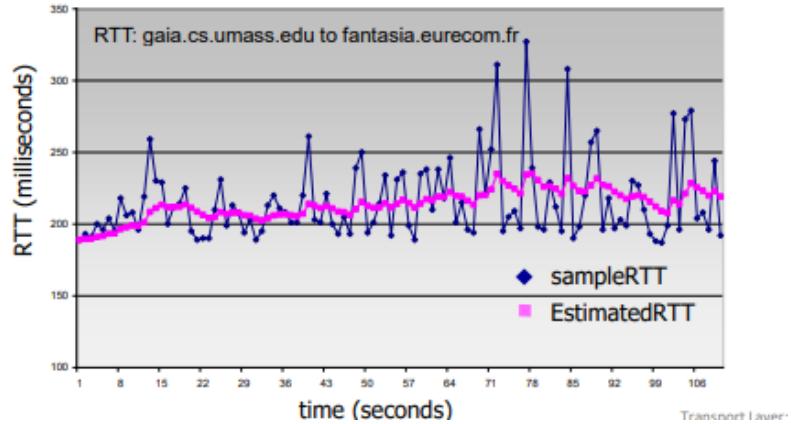
L'idea è di prendere il campione e lo combino con quello di stima precedente pesando uno con fattore α e l'altro con $1 - \alpha$ (che sommati tra loro mi riportano a 1). La nuova stima rappresenta a tutti gli effetti una combinazione tra la nuova stima e quella precedente. Tanto più alpha è grande, tanto minore sarà il peso della stima precedente.

Ha senso dimenticare i campioni precedenti in funzione della stima attuale poiché vogliamo agire a variazioni del livello di congestione, che si traduce direttamente nell'aumento attuale di RTT. L'influenza dei vecchi campioni decresce infatti esponenzialmente, in questo modo i campioni più recenti riflettono meglio la congestione attuale. Il valore raccomandato di alpha è 0.125.

TCP: tempo di andata e ritorno (round trip time) e timeout

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

- exponential weighted moving average (EWMA): media mobile esponenziale pesata
- l'influenza dei vecchi campioni decresce esponenzialmente
 - i campioni più recenti riflettono meglio la congestione attuale della rete
- Valore raccomandato:
 $\alpha = 0.125$



Per stabilire infine il margine di sicurezza sommo all'EstimatedRTT quattro volte la stima della variazione di RTT, che si calcola ancora una volta con una media mobile livellata esponenziale. In questo caso però il campione sarà il valore assoluto del campione corrente meno la stima attuale, utilizzando Beta raccomandato pari a 0.25.

- Intervallo di timeout: **EstimatedRTT** più un “margine di sicurezza”
 - grande variazione di **EstimatedRTT**: margine di sicurezza maggiore

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



RTT stimato

“margine di sicurezza”

- **DevRTT**: EWMA della deviazione di **SampleRTT** da **EstimatedRTT**:

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

(tipicamente, $\beta = 0.25$)

TCP

Vediamo ora, in modo semplificato, **come si comporta TCP**.

(Mittente) Quando l'applicazione vuole inviare dei dati TCP **crea un segmento con il numero di sequenza** (*il numero del primo byte del segmento*

nel flusso di byte) e, dopo averlo inviato, fa partire un timer se non era già in funzione. Si può immaginare il timer come se fosse associato al segmento più vecchio ancora non riscontrato, con intervallo di scadenza definito come **TimeOutInterval**.

In caso di timeout, a differenza di GoBackN (dove ritrasmettevamo tutto), ritrasmette solo il segmento che ha scatenato il timeout (come detto quindi, quello più vecchio non riscontrato) e riavvia il timer. Ogni volta che scatta il timeout questo viene raddoppiato (per aumentare la possibilità che l'ACK legata al pacchetto reinviato possa tornare in tempo), poi il timeout viene resettato normale quando arriva l'ACK.

In caso venga ricevuto l'ACK, aumenta sendbase (perché ACK cumulativo, riscontra tutto ciò che è stato inviato prima) e se ci sono ancora byte inviati ma non riscontrati riavvia il timer.

(Ricevente) Che succede se riceve un segmento correttamente e ordinato?

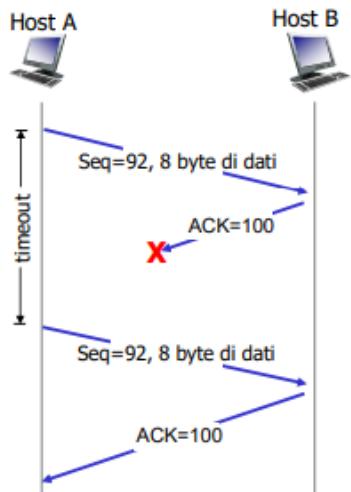
Normalmente, per come abbiamo visto, potrebbe già spacchettarlo e mandarlo a livello applicativo, ma in realtà per come è pensato TCP aspetta del tempo. Ciò per via del fatto che potrebbe arrivare a breve un successivo segmento in ordine dopo che è arrivato quello che ancora non è stato riscontrato. In questo modo il destinatario si ritrova due segmenti in ordine e *invia un solo riscontro cumulativo per entrambi*, prima di consegnare i dati a livello applicativo.

Che succede se arriva un segmento che è fuori ordine (successivo a quello richiesto) e quindi si crea un buco? Il destinatario **manda subito un ACK duplicato** (capiremo il perché nella ritrasmissione rapida) per i byte che si sta attendendo (estremità inferiore del buco, di modo che questo venga colmato)

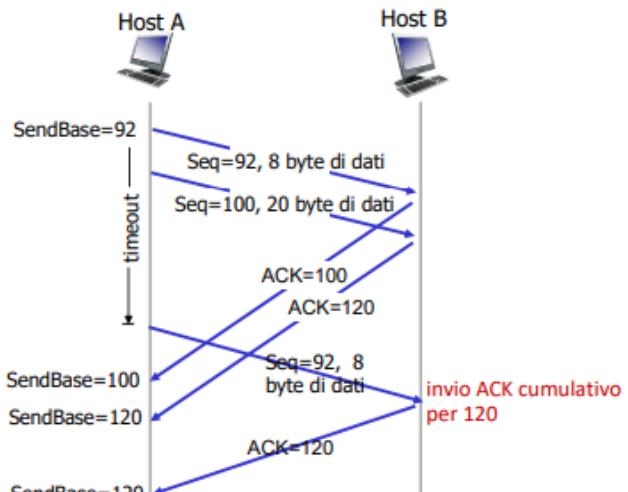
Evento presso il destinatario	Azione del ricevente TCP
Arrivo ordinato di segmento con numero di sequenza atteso. Tutti i dati fino al numero di sequenza attivo sono già stati riscontrati.	ACK ritardato. Attende fino a 500 millisecondi per l'arrivo ordinato di un altro segmento. Se in questo intervallo non arriva il successivo segmento, invia un ACK.
Arrivo ordinato di segmento con numero di sequenza atteso. Un altro segmento ordinato è in attesa di trasmissione dell'ACK.	Invia immediatamente un singolo ACK cumulativo, riscontrando entrambi i segmenti ordinati.
Arrivo non ordinato di segmento con numero di sequenza superiore a quello atteso. Viene rilevato un buco.	Invia immediatamente un ACK duplicato , indicando il numero di sequenza del prossimo byte atteso (che è l'estremità inferiore del buco)
Arrivo di segmento che colma parzialmente o completamente il buco nei dati ricevuti.	Invia immediatamente un ACK, ammesso che il segmento cominci all'estremità inferiore del buco

Transport L4

TCP: scenari di ritrasmissione



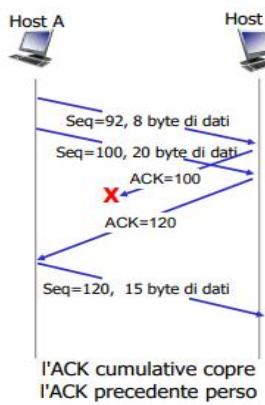
scenario con ACK perso



timeout prematuro

Mando il byte 92 con 8 byte. B quindi dovrebbe inviare l'ACK di 100, ma questo viene perso. Scade il timeout e viene rimandato lo stesso segmento, per cui il destinatario rinvia l'ultimo ACK già mandato (100).

Scenario timeout prematuro: A manda primo segmento 92 e poi il secondo segmento 100. Il primo segmento viene riscontrato con 100 e il secondo con 120, ma il riscontro arriva dopo che è scattato il timeout. **A quindi non rimanda tutto come farebbe GoBackN** ma solo il primo segmento non riscontrato (92). Poiché stiamo usando riscontri cumulativi, B non riscontrerà solo questo segmento ma tutti i dati che ha precedentemente ricevuti, quindi 120 (perché il destinatario ha già ricevuto i dati).



In quest'immagine si capisce *l'importanza dell'ACK cumulativo nel pipelining*: mentre il primo ACK viene perso il secondo mi riscontra tutti i byte precedenti, perciò il mittente può spostare in avanti la sua finestra di invio e continuare a trasmettere nuovi dati.

Ritrasmissione Rapida

Immaginiamo che il mittente invii tanti segmenti in pipeline. *Se viene perso il secondo segmento, dal terzo segmento arrivato al destinatario in poi continuerò a riscontrare il primo segmento ricevuto.* Quindi il mittente, vedendo tutti questi ACK con stesso numero di sequenza, capisce che si è creato un buco!

Ciò che accade è infatti che, secondo la **ritrasmissione rapida**, una volta ricevuti 3 ACK addizionali per gli stessi dati (3 ACK duplicati) viene rispedito dal mittente il segmento non riscontrato con il più piccolo numero di sequenza.

Ciò è da immaginare combinato con la bufferizzazione dei pacchetti fuori ordine per il destinatario, quando ho colmato il buco ho tutto in ordine e quindi posso spacchettare.

Controllo del flusso

Se bevo dalla fontanella ok, se bevo da un idrante esplodo. Il concetto è che il mittente non deve bombardare il destinatario di dati. Per farlo indico nei segmenti di risposta, nel campo **rwnd** (*receive window, finestra di ricezione*) quanto spazio il destinatario ha ancora nel buffer (e quindi quanti altri byte è pronto ad accettare).

Controllo di flusso == il destinatario controlla il mittente in modo che non invii troppi dati o troppo velocemente.

RcvBuffer == dimensione del buffer del ricevente, allora la rwnd sarà la differenza tra la RcvBuffer e i dati che sono in buffer che non ho letto.

I dati in buffer che non ho letto li calcolo come ultimo byte ricevuto meno l'ultimo byte letto

```
rwnd = RcvBuffer - (LastByteRcvd - LastByteRead)
```

Il buffer è tipicamente lungo 4096 byte, ma può essere regolato dagli host in base alle opzioni della socket oppure regolato dal sistema operativo.

Grazie a questa comunicazione, il mittente è in grado di limitare l'invio di dati garantendo che il buffer del destinatario non vada in overflow.

Gestione della connessione TCP

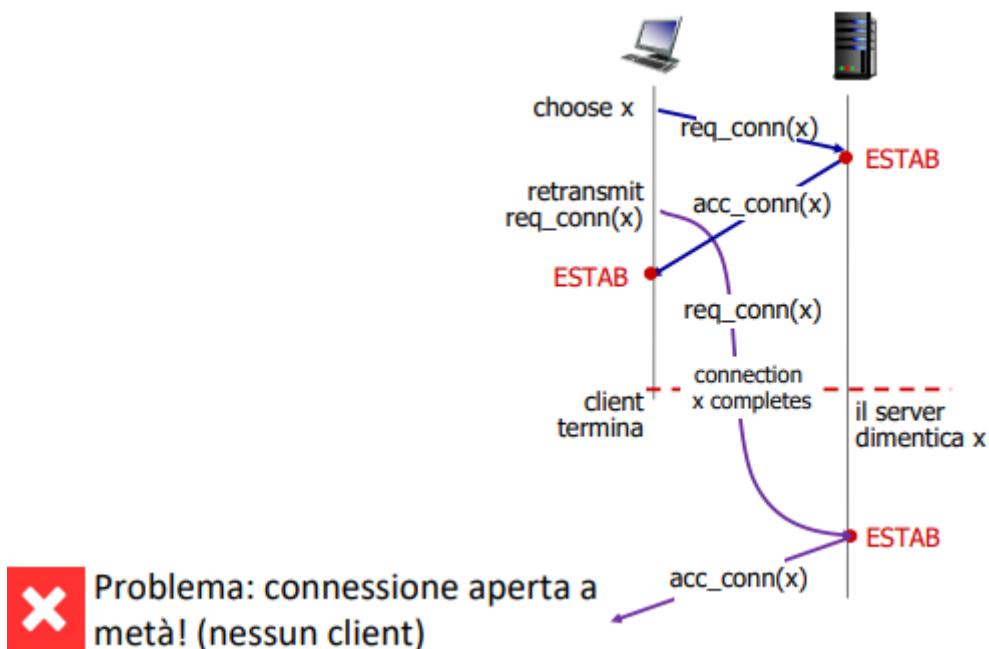
Stretta di mano tra i due host (**handshaking**), che serve a dimostrare che entrambi sono disponibili ad una connessione.

(Per il multiplexing avevamo visto come il server dovesse creare un socket in ascolto per dire che è in attesa su una certa porta. Il concetto è che se ricevo una richiesta di connessione su una porta in cui non sono in attesa allora non l'accetto.)

L'handshaking inoltre è necessario al fine di concordare i parametri di connessione (es. i numeri di sequenza iniziali e i receive buffer).

Approccio più semplice: Handshake a 2 vie.

Il client chiede la connessione e il server accetta. Il problema di questo approccio è che, come al solito, il client e il server non sanno nulla dello stato altrui ad ogni istante. Che succede quindi se il client, prima di aver ricevuto la conferma di connessione dal server, ritrasmette la richiesta di connessione e quella richiesta arriva al server dopo che la vecchia connessione è già stata chiusa? *Il server non può sapere che si tratta di un duplicato della vecchia connessione, quindi ne stabilisce una a vuoto.*



Ancora peggio se accettasse dei dati per questa nuova connessione!

TCP Handshake a 3 vie

Necessario ad ovviare ai problemi visti con il 2 way handshake.

Il client è inizialmente in **stato di closed** e il server in **stato di listen**.

Il client richiede la connessione al server, inviando un segmento **SYN** con il numero di sequenza iniziale passando poi allo **stato SYNsent**.

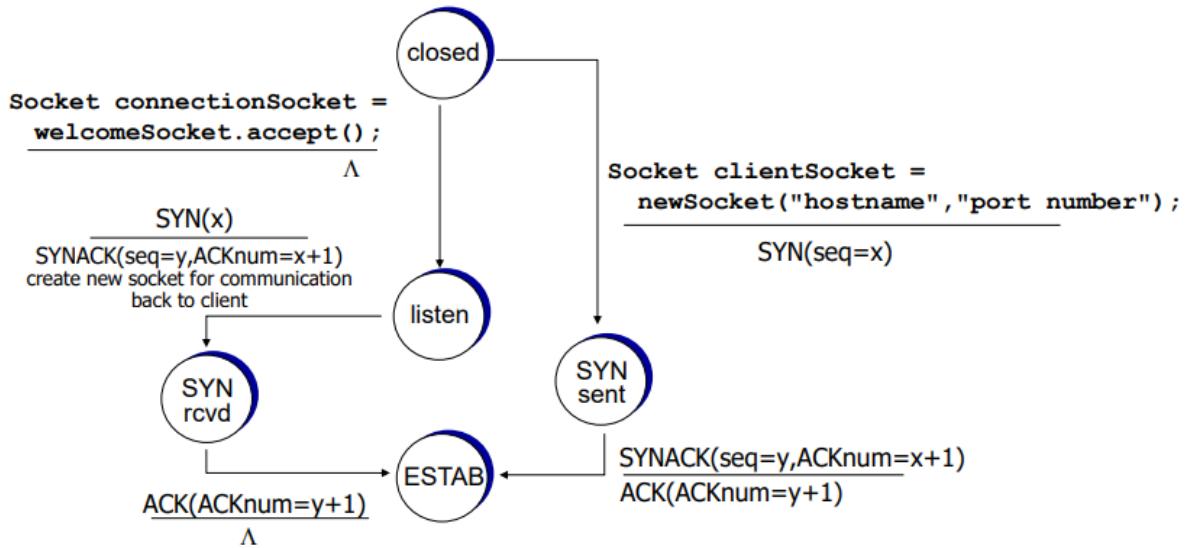
Alla ricezione del SYN il server passa nello **stato SYNrcvd** e invia al client il **SYNACK**, che rappresenta l'acknowledgment del SYN ricevuto e a sua volta un SYN (affermare di voler stabilire la connessione).

In particolare quando il ricevente invia un ACK il flag ACK è 1, quando viene mandato un SYN il flag SYN è 1. In questo caso, essendo SYNACK, sono 1 entrambi.

Si noti come il SYN del client ha numero di sequenza x, ma poiché il server sta riscontrando un SYN e non dei dati generici il SYNACK per riscontrare ha numero di sequenza x+1, incrementa solo di 1! (ho ricevuto x, ora mi aspetto x+1).

Il client, ricevuto il segmento SYNACK, risponde con un ACK e passa in uno **stato ESTAB** (*established*). Quest'ACK può anche contenere già dei dati di livello applicativo. Quando il server (ancora nello stato SYNrcvd) ottiene l'ACK passa nello **stato ESTAB** senza fare nulla: *si è finalmente creata la connessione*.

TCP 3-way handshake FSM



Ma perché è migliore rispetto al 2 way?

Dato che la connessione non viene immediatamente stabilita ne deriva una “maggiore sicurezza”: non c’è più il rischio di stabilire una connessione anche quando in realtà il client non l’ha richiesta per quell’istante! Infatti, ad esempio, se il server manda il SYNACK dopo aver ricevuto la richiesta quando il client ha già lasciato la connessione, egli la riceve ma non si trova nello stato SYNsent per ignora e dopo un po’ il server chiude la connessione.

Attacco SYN FLOOD

Attacco di sicurezza per cui vengono volutamente mandati dei SYN senza voler stabilire davvero la connessione. In particolare l’aggressore manda un segmento TCP SYN con IP fasullo, il server rispondendo alloca e inizializza le variabili e i buffer di connessione (sfrutta delle risorse!) e invia il segmento SYNACK alla porta e all’indirizzo IP (fasullo) di origine, transitando nello stato SYNrcvd.

La rete tenta di consegnare il segmento SYNACK all’IP fasullo, raggiungendo eventualmente un altro host che non c’entra nulla e non risponderà. Il server non riceverà quindi l’ACK ed eventualmente (dopo un minuto o più) rilascerà tutte le risorse associate a questa connessione mezza aperta.

In questo modo l'aggressore è riuscito a consumare risorse del server. Se ci si organizza per inviare numerosi SYN a un server obiettivo, si può montare un attacco di DoS (spesso sotto forma di DDoS).

Una contromisura a questo attacco di rete è il **SYN cookie**. L'idea è che, alla ricezione del segmento SYN iniziale, il server calcoli l'hash degli indirizzi IP e numeri di porta di origine e destinazione e di una chiave segreta, producendo un cookie con numero di sequenza iniziale. Inserisce il cookie dentro al segmento SYNACK e non alloca alcuna risorsa, né tantomeno memorizza il cookie.

Se il segmento SYN era legittimo, il client alla ricezione di SYNACK invia un ACK che usa come numero di Acknowledgment cookie + 1 (?).

Alla ricezione di questo ACK, il server può capire che è legato al SYN precedente perché gli basta calcolare Acknowledgment -1 per ottenere il cookie. L'inclusione della chiave segreta impedisce all'aggressore di creare un cookie valido.

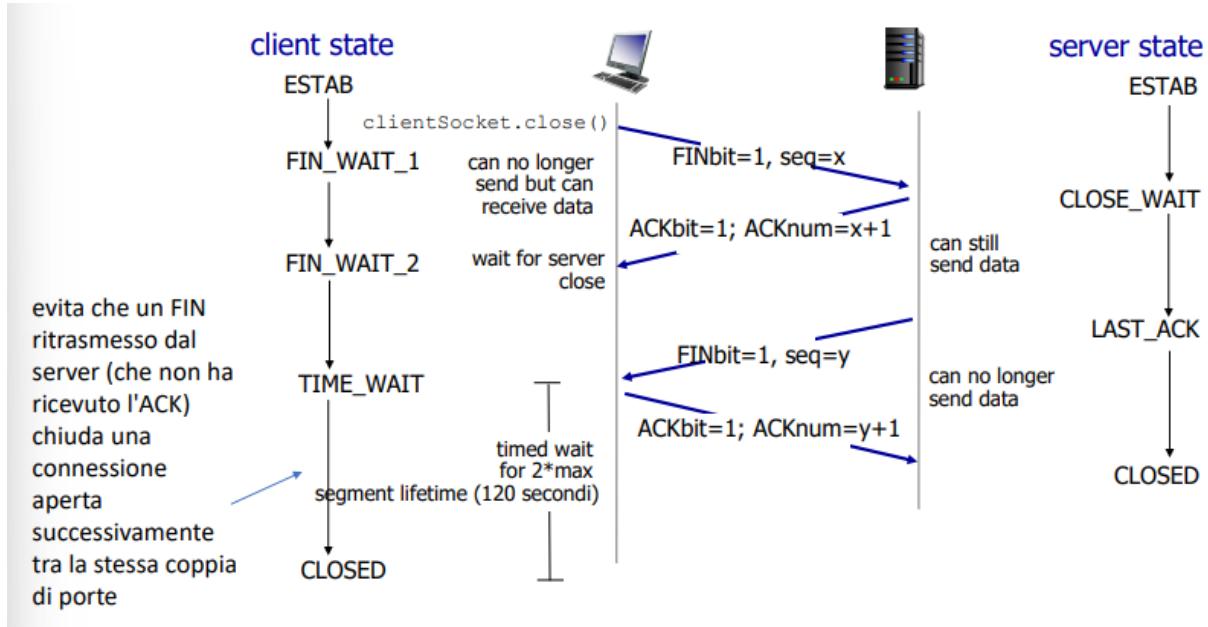
*In poche parole quindi via **SYN cookie** codifico nel sequence number iniziale (tramite un hash che combina anche una chiave segreta) tutti i dati del SYN che ho ricevuto. In questo modo, quando poi ottengo l'ACK finale, poiché posso ricostruire il numero di porta, il sequence iniziale etc... posso ricalcolare l'hash e vedere se coincide (evitando così di sprecare risorse se dall'altra parte avevo un malintenzionato).*

NB. Con l'handshake a 3 vie i numeri di sequenza iniziali non sono zero, ma sono scelti a caso. In questo modo mi irrobustisco rispetto a cose che arrivano fuori ordine. (immaginiamo ad esempio che arrivi dal server un SYNACK di una connessione vecchia, mi accorgo che non va bene perché ha un numero di sequenza che non è quello attuale che ho inviato scelto casualmente!)

La connessione viene infine chiusa tramite i segmenti con i flag FIN = 1.

Quando il client vuole chiudere una connessione, manda un segmento con il flag FIN = 1, passa così nello stato FIN_WAIT_1. Quando il server, che si trova in una connessione stabilita, riceve il segmento con FIN, sa che il client vuole chiudere la sua estremità (cioè non manderà più dati). Perciò il server risponde con un segmento di ACK e passa dallo stato ESTAB allo stato CLOSE_WAIT. Il client passa quindi allo stato FIN_WAIT_2, dove può ancora continuare a ricevere dati dal server. Quando il server decide di chiudere invia a

sua volta un segmento FIN al client (passando allo stato LAST_ACK) che (passando allo stato TIME_WAIT) risponderà con un ACK. Una volta ricevuto l'ACK, il server avrà ufficialmente chiuso la connessione mentre il client dovrà aspettare, prima di chiudere, tempo pari a 120 secondi (gestito dal sistema operativo).



Perché questo TIME_WAIT per il client di 120 secondi? Perché l'ACK potrebbe non essere ricevuto, e in tal caso il server potrebbe ritrasmettere FIN. (120 secondi è il doppio del max tempo in cui un pacchetto può rimanere in Internet, per essere sicuri).

Non posso permettermi di perdere la ritrasmissione del FIN, immaginiamo infatti di chiudere e riaprire la connessione. A quel punto se mi arrivasse il FIN con numeri di sequenza giusti mi chiuderebbe la connessione, e lì sarebbe un problema.

Congestione

Il controllo di flusso riguardava la possibilità che il singolo mittente sovraccaricasse il destinatario.

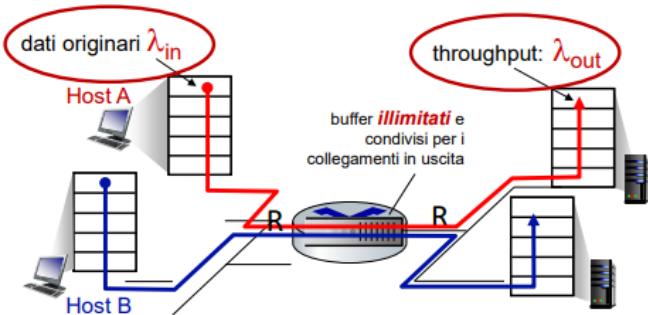
Congestione == molti mittenti che mandano troppo e troppo velocemente.

I sintomi di congestione sono **lunghi ritardi** (*accodamento nei buffer dei router*) e **pacchetti persi** (*overflow nei buffer dei router*), vediamo il perché.

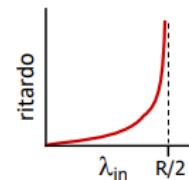
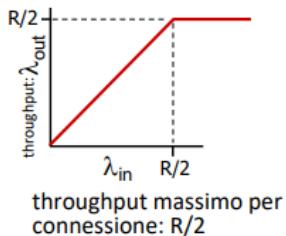
Cause/costi della congestione: scenario 1

Caso più semplice:

- un router con buffer illimitati
- capacità dei collegamenti di ingresso e uscita: R
- due flussi
- nessuna ritrasmissione



D: Cosa accade quando il tasso di arrivo λ_{in} si avvicina a $R/2$?



Transpo Layer: 3-129

Caso più semplice: due host che inviano dei dati a due destinatari diversi e passano attraverso un router. Supponiamo che il router abbia buffer illimitati e che ciascun host invii dati con un tasso (throughput, frequenza di trasmissione) λ_{in} , poiché non vi sono perdite né ritrasmissioni i dati che invio arriveranno prima o poi con ritardo finito per cui il traffico immesso è uguale al traffico ricevuto (throughput lambdaout = throughput lambda in).

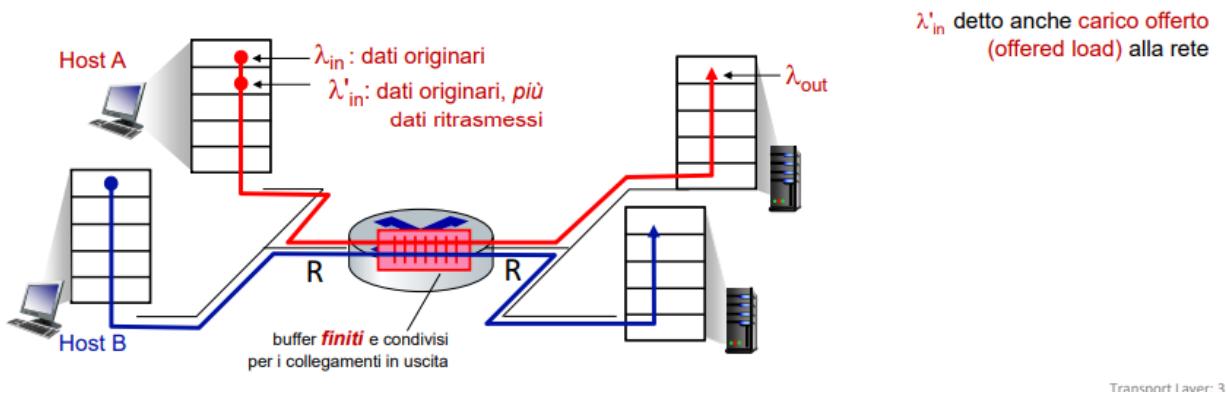
Ciò è vero fino a quando lambda in non si avvicina a $R/2$, dove R è la capacità dei collegamenti di ingresso e uscita del router. Poiché sono due i client che mandano roba, il router deve dividere a metà la sua capacità per cui al più mi fa trasmettere a $R/2$. Quindi tasso di invio maggiore di $R/2$ non mi serve a nulla. Man mano che continuiamo a saturare il collegamento in uscita però l'utilizzazione del canale (e quindi l'intensità del traffico) si avvicina a 1, per cui si manifestano ritardi sempre più grandi.

Quindi congestione elevata comporta ritardi elevati.

Caso 2: code finite

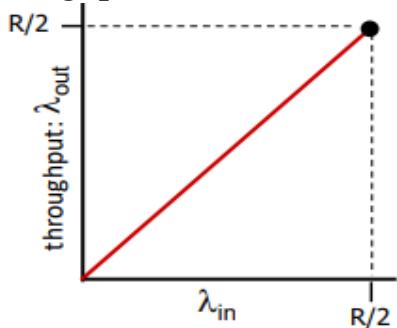
Potrei in questo caso avere delle perdite, per cui talvolta i mittenti dovrebbero effettuare delle ritrasmissioni nel caso in cui dei dati vengano persi. Per via di queste ritrasmissioni, l'input del livello di trasporto è $\lambda_{in}' \geq \lambda_{in}$

- un router, buffer *finiti*
- il mittente ritrasmette pacchetti perduti (scartati perché il buffer era pieno)
 - input del livello di λ_{in}
 - input del livello di traporto include le *ritrasmissioni* : $\lambda'_{in} \geq \lambda_{in}$

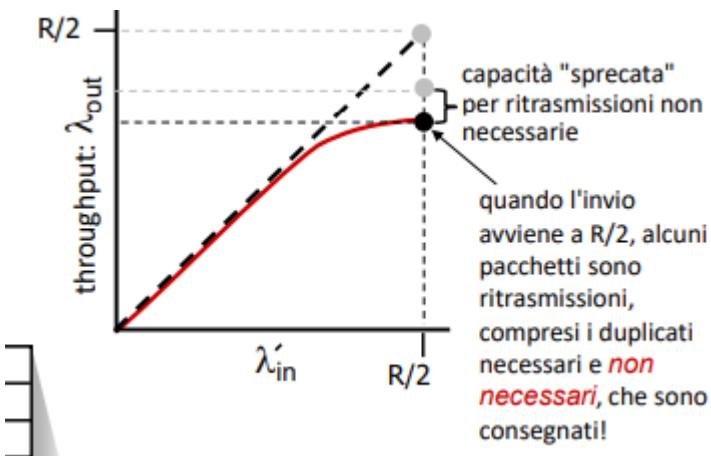


Se (utopisticamente) il mittente avesse **conoscenza perfetta**, cioè fosse in grado di inviare dati solo quando c'è spazio disponibile nei router, allora non ci sarebbero perdite per cui non dovrebbe ritrasmettere e quindi $\lambda_{in}' = \lambda_{in}$

Finché non arriviamo a $R/2$, in questa idealizzazione, come prima avremo che il throughput cresce linearmente assieme al traffico immesso.



Realisticamente però non è così e talvolta dei pacchetti si perdono a causa dei buffer pieni, per cui è necessaria la ritrasmissione. *Si crea un gap tra i dati ricevuti dall'applicazione e quelli inviati perché uno spazio di quelli da inviare è occupato anche dai pacchetti da ritrasmettere. Questo gap si intensifica se vengono ritrasmessi pacchetti inutilmente*, che erano arrivati al destinatario ma di cui non era arrivata in tempo l'ACK (timeout prematuro).

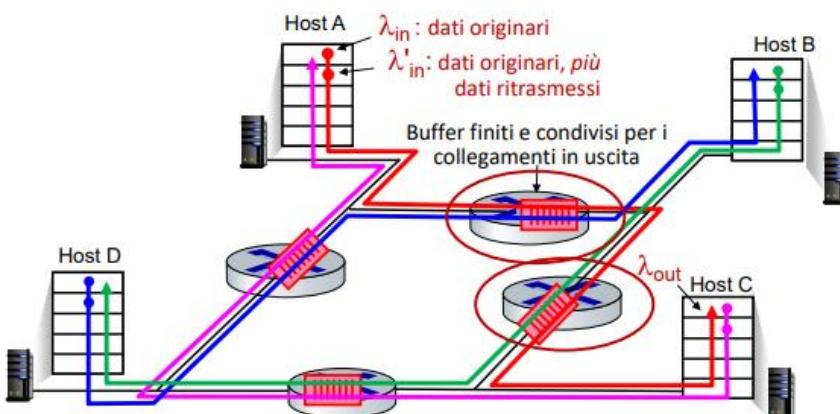


Caso 3:

- quattro mittenti
- percorsi multi-hop
- timeout/ritrasmissione

D: che cosa accade quando λ_{in} e λ'_{in} aumentano?

R: quando λ'_{in} aumenta, tutti i pacchetti blu in arrivo alla coda in alto sono scartati, throughput blu $\rightarrow 0$



Questo è il caso peggiore in assoluto, si parla di **collasso di congestione**.

Quattro connessioni tra le varie coppie, ogni connessione attraversa due router. Supponendo che una connessione possa spedire ad una velocità maggiore di R (velocità di collegamento dei router) le connessioni immediatamente vicine la router (“al primo hop”, dall’host direttamente al router) rubano la banda alle connessioni provenienti da altri router. Nel router in alto, ad esempio, passano solo i dati del rosso, quelli del blu no! Tuttavi il rosso, a sua volta, arrivato nel router a destra sarà bloccato dal traffico verde che arriva da B (rosso blocca blu ma bloccato da verde). Vale lo stesso per il verde che è bloccato dal rosa e dal rosa che è bloccato dal blu. Nell’essere bloccati ciò che succede è che i dati restano in coda, ma se restano in coda per troppo tempo scade il timeout e i dati vengono ritrasmessi all’infinito, *la rete smette di funzionare*.

Lez 10

Riassuntone

Abbiamo visto il protocollo TCP, uno dei due principali protocolli del livello di trasporto (protocollo perché offre servizi differenti per le applicazioni). Il complesso dei servizi forniti da un protocollo prende il nome di **modello di servizi**.

In base all'applicazione che sviluppo, scelgo un modello diverso (quello che meglio si adatta).

UDP si tratta di un'estensione molto semplice del protocollo del livello di rete IP, a cui è aggiunta una checksum (per un minimo controllo d'errore) e **multiplexing e demultiplexing**.

A livello di rete ho un trasferimento di dati da host a host. Tuttavia ogni host vuole comunicare contemporaneamente con processi diversi (analogia delle case, in ogni casa ho più persone che vogliono spedire diverse lettere). Per questo il livello di trasporto fa multiplexing quando invia (prende tutti i messaggi dai processi e li invia al servizio di rete “convogliandoli in un'unica strada”) e demultiplexing quando arrivano gli vengono consegnati i dati dal livello di rete (ridistribuisce tutto ai vari processi, capisce a quali socket specifiche deve inviare i dati).

L'aggiunta di multiplexing e demultiplexing permette di “trasformare la comunicazione tra host” in “comunicazione tra processi”, e questi sono i servizi base che vengono forniti da ogni protocollo di livello di trasporto (perché per definizione il livello di trasporto permette la comunicazione tra processi sfruttando il servizio di rete che di base non lo prevede).

Ma perché diciamo che UDP non è affidabile se ha la checksum, e quindi fa controllo d'errore? Perché se i dati non sono arrivati proprio al mittente, o sono arrivati corrotti, finisce lì. Il mittente non li ritrasmette. Questo è dovuto al fatto che UDP è pensato come protocollo semplice, che offre il vantaggio di un controllo molto preciso di come e quando i dati sono inviati.

TCP al contrario è molto più complesso, ha un modello di servizi decisamente più elaborato. **La semantica di TCP è un flusso di byte affidabile consegnato**

in sequenza.

“Nessun confine ai messaggi” perché, mentre in UDP il messaggio inviato era un’entità discreta (datagrammi) e rappresentava la singola richiesta, in TCP scrivo dei byte che sono ricevuti dal destinatario che dovrà sapere quando finirà la richiesta. È compito del livello applicativo ideare un modo per cui si possa identificare la fine di una richiesta (come in HTTP, dove un messaggio è formato da un’intestazione e un’entità e la fine dell’intestazione è riconosciuto dalla riga vuota, mentre nell’intestazione è presente un campo “contentlength” che indica il numero di byte dell’entità).

TCP Full Duplex perché permette lo scambio di messaggi da entrambe le direzioni, ed è protocollo punto-punto perché riguarda una coppia di processi.

TCP orientato alla connessione (necessita di handshaking a 3 vie) e usa pipelining (inviare in sequenza più pacchetti senza aspettare per ognuno l’ACK come vorrebbe lo stop-and-wait, se non usassi pipeline prestazioni pessime in quanto non sfrutterei appieno la larghezza di banda).

La larghezza della finestra in TCP è regolata da due algoritmi: controllo del flusso e controllo di congestione.

I dati inviati di volta in volta non possono essere mai più grandi della maximum segment size MSS.

Si inviano flussi di byte di volta in volta. Il numero di sequenza di un segmento è rappresentato dal primo byte del segmento, mentre il riscontro è identificato dal numero del prossimo byte che viene atteso (es. invio il segmento 3 che contiene 20 byte, il riscontro sarà 23 (conto anche il 3, per questo il 23esimo è quello che aspetto)).

Timeout determinato come somma del valore medio di RTT + margine di sicurezza (che è 4 volte la stima della variabilità dell’RTT). *Ciò è per evitare timeout prematuri e quindi ritrasmissioni inutili.*

TCP non funziona esattamente come GoBackN, importanti differenze. La principale è che quando ho un timeout non ritrasmetto tutto ma solo il segmento associato al timer scattato (quello più vecchio, con numero di sequenza minimo). Ogni volta che scatta il timeout questo viene raddoppiato (per aumentare la possibilità che l’ACK legata al pacchetto reinviato possa tornare in tempo), poi il timeout viene resettato normale quando arriva l’ACK.

Altra differenza risiede nel fatto che se arriva un segmento fuori ordine il ricevente può non scartarlo e tenerlo in un buffer. Questo è vantaggioso poiché

il mittente ogni volta che al destinatario arriva un pacchetto non in sequenza riceve un ACK legata al pacchetto con sequenza giusta non ancora arrivato, e scattati i 3 ACK lo rimanda. In questo modo quando arriva quel pacchetto il destinatario, avendo in buffer anche gli altri, ha coperto il buco e può mandare a livello applicativo tanta roba insieme, inviando tra l'altro un riscontro cumulativo più grande per il mittente che potrà inviare più roba.

NB. *Con l'handshake a 3 vie i numeri di sequenza iniziali non sono zero, ma sono scelti a caso.* In questo modo mi irrobustisco rispetto a cose che arrivano fuori ordine.

Controllo di flusso si riferisce al destinatario che controlla il mittente affinché non invii dati più velocemente di quanto il destinatario non possa gestire.

Controllo di congestione controlla più mittenti affinché non invino troppi dati troppo velocemente perché la rete li gestisca. (focus del controllo di congestione sulla sostenibilità dell'intera rete).

Sintomi di congestione: ritardi e perdite.

Controllo della congestione TCP

Sono stati una serie di collassi di congestione nell'Internet degli albori che hanno portato allo sviluppo dei protocolli di congestione TCP.

- **Trasmissione dati affidabile**
 - reagisce alla perdita (e alla corruzione) dei pacchetti, possibilmente causata dalla congestione
 - "tratta i sintomi della congestione"
- **Controllo della congestione**
 - "cura la malattia"
 - evita che la malattia si aggravi fino a degenerare fino allo scenario di "collasso di congestione"

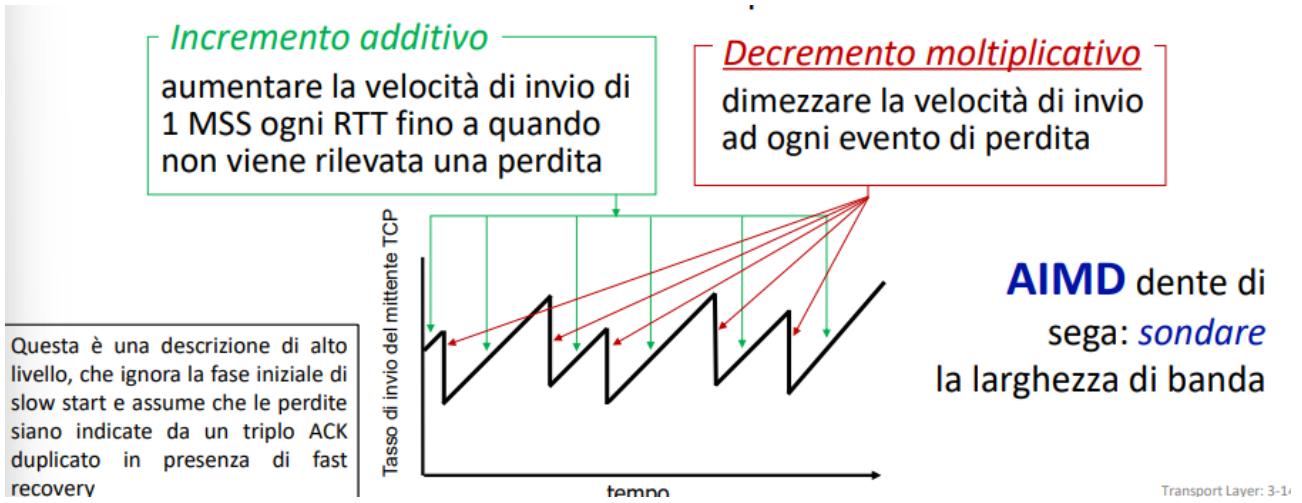
Nella sua variante classica, il controllo di congestione usa **l'approccio end-to-end** (*ovvero la congestione viene dedotta dagli host senza supporto della rete*). Vedremo evoluzioni che sfruttano anche informazioni dirette dalla rete per controllare la congestione.

Come funziona in linea di massima il controllo di congestione TCP?

Io mittente *aumento man mano la mia velocità di trasmissione* fintanto che continuo a ricevere ACK correttamente, cioè **finché non si verifica un evento di perdita**. Quando ciò avviene significa che nel tragitto c'è congestione, per cui *riduco subito la velocità per poi riprendere*.

In particolare *ciò che fa TCP è crescere linearmente e dimezzare quando rilevo congestione, per poi riprendere a crescere linearmente*.

Più formalmente si parla di **Incremento Additivo** in cui la velocità viene aumentata di 1 MSS ogni RTT e **Decremento Moltiplicativo** (AIMD) in cui la velocità viene dimezzata ad ogni evento di perdita.



Un approccio di questo tipo (detto a dente di sega) permette a TCP di “**sondare**” la banda disponibile perché, chiaramente, se si vuole evitare la congestione non trasmetto o trasmetto pochissimo coniugando anche l’obiettivo di trasmettere il più velocemente possibile (cresco fintanto che non c’è congestione).

Se il punto in cui la velocità di trasmissione per cui rilevo congestione è sempre lo stesso tenderò ad oscillare tra due valori (quel valore e la sua metà) altrimenti, con livelli di congestione variabile, potrei dimezzare prima o successivamente.

Questa descrizione è ad alto livello e fa riferimento alla fase di congestion avoidance (ho ignorato la fase di slow start che vedremo tra poco).

TCP Reno e TCP Tahoe

In **TCP Reno**:

- Nel caso in cui la congestione sia rilevata come *arrivo di un triplo ACK duplicato* (indica che il pacchetto è sicuramente andato perso) si passa ad uno stato di **fast recovery** e si applica il **Decremento Moltiplicativo AIMD**
- Se invece la congestione è rilevata dallo scadere del timer (timeout, ACK non arrivato in tempo) allora si taglia il tasso di invio ad 1 MSS e si torna nella fase di **slow start**.

Il Decremento Moltiplicativo AIMD non veniva utilizzato in una versione precedente a Reno di TCP, detta **TCP Tahoe**. In quella versione ogni volta che veniva rilevata perdita il tasso d’invio veniva ridotto a 1 MSS e si passava all’evento di **slow start**.

Il Decremento Moltiplicativo è quindi attuato quando la connessione si trova in stato di **congestion avoidance** e rileva triplice ACK duplicato.

Ma perché usare AIMD? Si tratta di un algoritmo asincrono e distribuito, si è anche dimostrato che ottimizza i flussi congestionati in tutta la rete e ha proprietà desiderabili di stabilità.

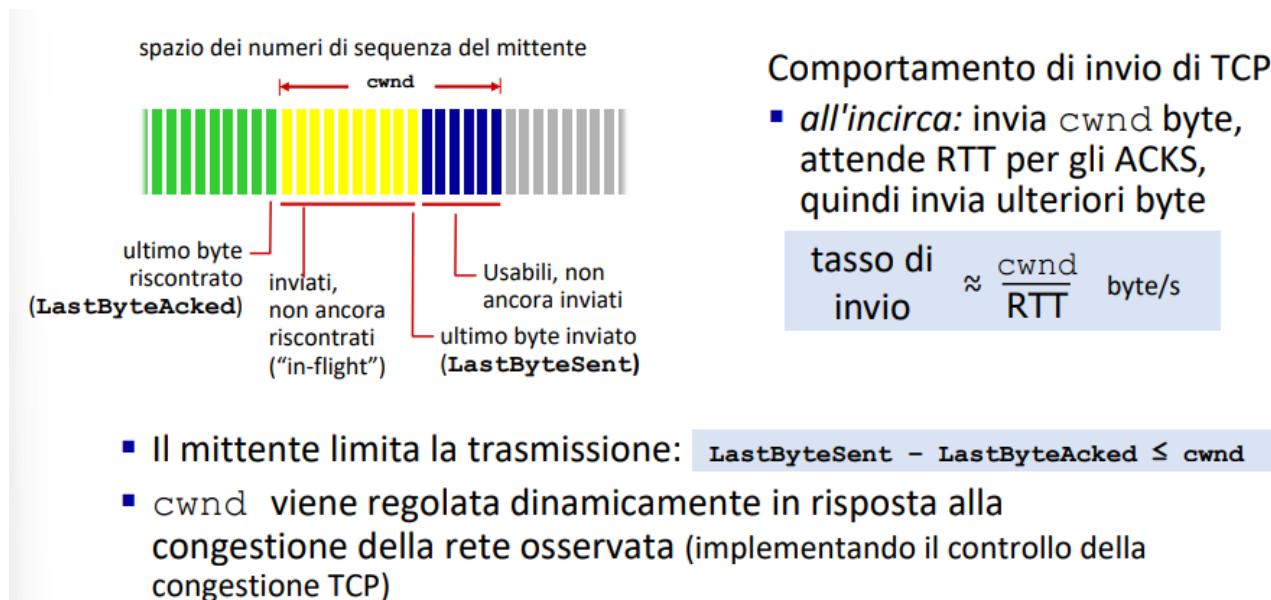
Controllo della congestione TCP: dettagli

Abbiamo detto informalmente che regoliamo la velocità di trasmissione.

Ma come faccio a farlo?

Abbiamo visto che nel pipelining inviamo un certo numero di byte non riscontrati adatti alla finestra. Abbiamo inoltre detto che, ***per fare controllo di flusso***, questa finestra è limitata dalla **rwnd** (*receivewindow*, finestra di ricezione) del destinatario.

Il controllo di congestione aggiunge un ulteriore vincolo a tutto ciò, la **cwnd** (*congestion window*).



Si hanno in verde i byte riscontrati (l'ultimo di questi è il **LastByteAcked**), in giallo i byte inviati ma non ancora riscontrati (l'ultimo è il **LastByteSent**) e in blu i byte che potrei utilizzare non appena ne avrò bisogno (me lo chiederà il

livello applicativo).

I byte grigi sono invece quelli che non posso inviare, perché *se li inviassi starei inviando troppo velocemente*. L'ampiezza del tratto giallo-blu è la mia cwnd, ed è questa che regola il mio tasso di invio.

TCP, per il controllo di congestione, limita la trasmissione ponendo la parte di byte in giallo \leq di cwnd.

Il **tasso di invio medio** è infatti proprio **cwnd/RTT** byte/sec, perché all'incirca vengono inviati cwnd byte e atteso RTT per il riscontro cumulativo per cui si possano inviare ulteriori byte. *Chiaramente al cwnd varia nel tempo: più c'è congestione più si riduce, meno ce n'è più si allarga.*

Anche il controllo di flusso però deve controllare il tasso di invio, e per farlo impone che l'ampiezza della finestra (diff. tra ultimo byte inviato e ultimo byte riscontrato) sia \leq della rwnd (byte liberi nel buffer di ricezione del destinatario).

Per rispettare il controllo di congestione e di flusso l'idea è di combinare i vincoli insieme, ponendo come ampiezza della finestra il minimo dei due:

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

- Il controllo del flusso regola la quantità e la velocità dei dati inviati in funzione della finestra di ricezione comunicata dal destinatario **rwnd** (byte liberi nel buffer di ricezione del destinatario)
- Quindi,

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{rwnd}$$

- Combinando questo vincolo con quello visto in precedenza

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

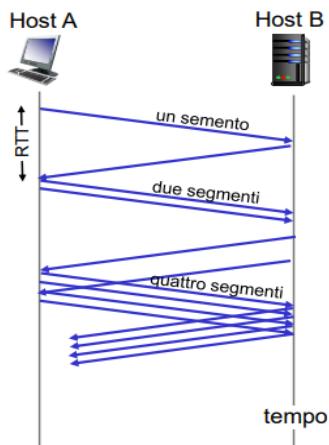
Assumendo che il buffer di ricezione sia sufficiente grande, possiamo trascurare il vincolo della finestra di ricezione (che assumiamo sempre maggiore della finestra di congestione)

Abbiamo detto che il comportamento per cui si arriva a Decremento Moltiplicativo è quello di una connessione a regime. Ma come parte una connessione TCP? Si parla di **Slow Start**.

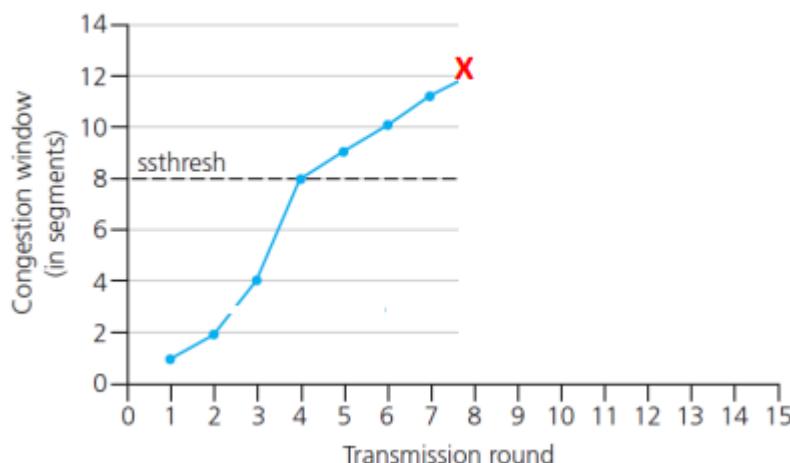
Quando inizia la connessione, la frequenza di trasmissione parte da 1 MSS e aumenta esponenzialmente raddoppiando di volta in volta (1, 2, 4, 8 etc...) fino

a quando non si verifica un evento di perdita. Ciò viene fatto, nel concreto, aumentando cwnd per ogni ACK ricevuto.

È vero quindi che il tasso iniziale è lento, ma poiché aumenta esponenzialmente si arriva a tassi di trasmissione molto veloci in poco tempo.

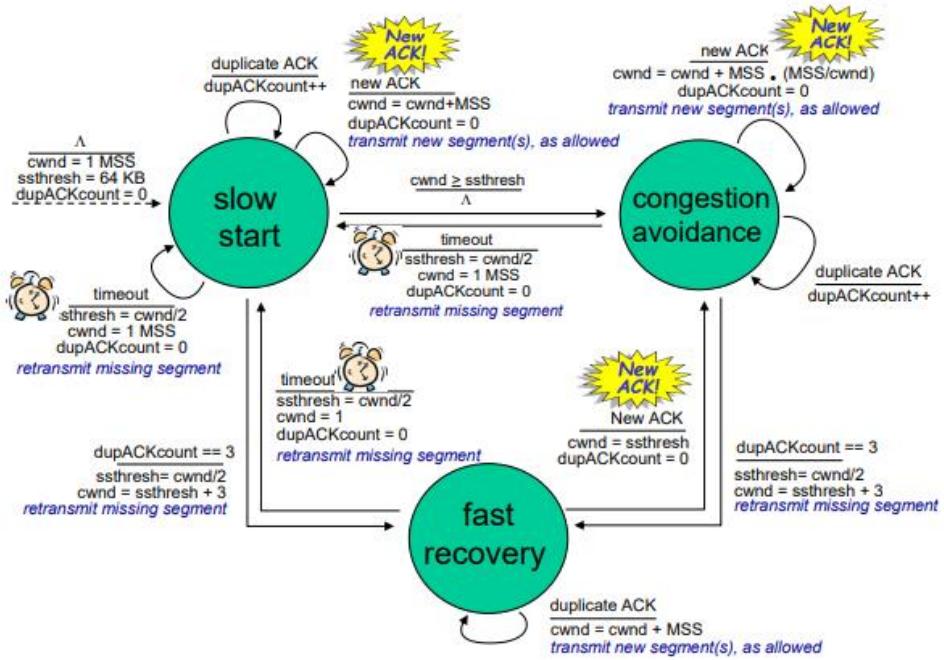


Ma quando converrebbe passare dall'aumento esponenziale a quello lineare (da slow start a **congestion avoidance**)? Quando cwnd raggiunge $\frac{1}{2}$ del suo valore prima del timeout!



(del resto se dalla metà del suo valore continuassi con l'incremento esponenziale arriverei immediatamente, di nuovo, al valore critico. Per questo conviene procedere con la crescita lineare da quel punto).

Quando arrivo al valore di **ssthresh** (variabile, inizialmente 64 KB, molto grande, perché appena parte viene trascurato) smetto la crescita esponenziale e passo a quella lineare, quella di congestion avoidance. *In caso di perdita, ssthresh è impostato a $\frac{1}{2}$ della larghezza che la cwnd aveva appena prima dell'evento di perdita.*



In TCP Tahoe se si ha una perdita si fa quindi sempre la stessa cosa: imposta la ssthresh a metà della cwnd, porta la cwnd a 1 e poi riparte lo slow start.

In TCP Reno si distinguono i due casi: se ho timeout si fa come TCP Tahoe, se ho triplice ACK si va per breve tempo in **fast recovery**: la ssthresh viene impostata a metà della cwnd, la cwnd viene dimezzata (AIMD).

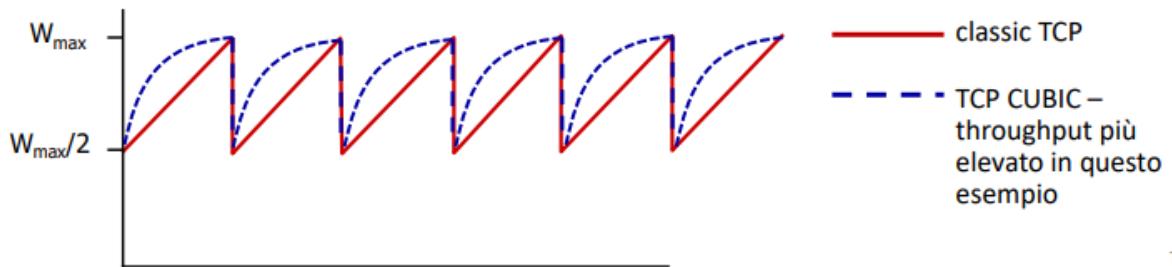
TCP CUBIC

È un'estensione di TCP Reno che sonda ancora meglio la larghezza di banda utilizzabile.

La differenza sostanziale rispetto a Reno risiede nella fase di **congestion avoidance**.

Sia **Wmax** la dimensione di cwnd all'istante in cui viene rilevata la perdita. Mentre in TCP Reno la banda veniva sondata secondo una crescita lineare durante la congestion avoidance (velocità costante), adesso l'intuizione è la seguente:

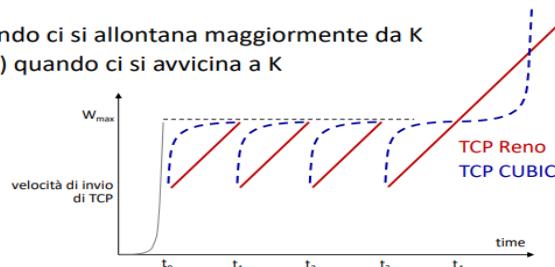
se la congestione è rimasta quella di prima, perché aumentare la frequenza in modo costante dal momento che ho dimezzato fino a quando non raggiungo di nuovo il punto critico di perdita? Non converrebbe invece iniziare aumentando il tasso di invio da Wmax/2 più velocemente per poi decrementarlo man mano che mi avvicino a Wmax (punto potenzialmente critico)?



Tras.

Formalmente, l'aumento della cwnd è dato da una **funzione cubica** della distanza tra l'istante corrente e l'istante K dove cwnd raggiungerà nuovamente W_{max} (e quindi eventualmente perdita di pacchetto e dimezzo).

- aumenta W come una funzione del **cubo** della distanza tra l'istante corrente e K
 - aumenti maggiori quando ci si allontana maggiormente da K
 - aumenti minori (cauti) quando ci si avvicina a K
- TCP CUBIC predefinito in Linux, il TCP più diffuso per i server Web più comuni



Controllo di congestione basato sul ritardo RTT

TCP (sia CUBIC che classic) rilevano la congestione primariamente con la **perdita di pacchetti**. La perdita di pacchetti si verifica infatti su un router in cui i dati si sono accumulati (per tanto traffico o provvisto di collegamento in uscita poco veloce) al punto da riempire il buffer. (quel router è detto collegamento **“collo di bottiglia”**, bottleneck)

Tuttavia la congestione può essere rilevata anche in altri modi. In particolare un modo più sottile per rilevarla ancor prima che si verifichi una perdita è legato al RTT: **se ho un router collo di bottiglia congestionato e invio sempre più dati aumenterà di volta in volta anche l'RTT** (tempo che mi arrivo la notifica del destinatario che ha ricevuto il pacchetto!) perché i miei dati finiscono nella coda che si allunga e che quindi aumenta il tempo di accodamento (componente di RTT) e quindi RTT cresce.

Misurando l'RTT, avrò valore RTTmin quando il percorso non è congestionato, in particolare se trovo sempre libero. Inoltre il throughput (frequenza di trasmissione) massimo che potrò avere sarà cwnd/RTTmin. Il throughput che però misurerò nel pratico sarà cwnd/RTT (che misuro sul momento).

- **Se il mio throughput misurato è vicino a quello massimo no congestione, posso inviare ancora di più.**
- **Se invece il throughput misurato è considerevolmente più piccolo di**

quello massimo inviare ancora più dati sarebbe insensato, si accoderebbero e contribuirei alla congestione. Per cui **mi conviene ridurre la velocità di trasmissione per far svuotare la coda ed evitare la congestione.**

- controllo di congestione senza indurre/forzare perdite
- massimizzare il throughput (“keeping the just pipe full... ”) ma mantenendo il ritardo basso (“...but not fuller”)
- un certo numero di TCP distribuiti adottano un approccio basato sui ritardi
 - BBR (Bottleneck Bandwidth and Round-trip propagation time) impiegato sulla rete dorsale (interna) di Google

Si previene la congestione così come la perdita di pacchetti.

Abbiamo visto finora che il controllo di congestione in TCP è basato sul fatto che gli host deducano lo stato di congestione guardandone i sintomi: **perdite e ritardi**.

Mentre TCP classico guarda solo le perdite, mentre evoluzioni recenti come appena visto considerano anche i ritardi (vantaggioso perché si evitano perdite).

Explicit congestion notification (ECN)

Un altro modo per essere proattivi ed evitare perdite è avere informazioni direttamente dalla rete. Si parla di protocolli **Explicit congestion notification**.

Ciò che accade in questo caso è che si **utilizzano 2 bit nel campo Type of Service (ToS) nell'intestazione del pacchetto IP** come ECN. Il mittente imposta questo ether a valore 10 per informare i router del fatto che è in grado di trattare quelle notifiche. Quando un pacchetto così marcato attraversa un router congestionato, questo lo vede e ci mette 11, per informare l'host a cui arriverà il pacchetto del fatto che questo ha attraversato un router congestionato. Quando il destinatario riceve un pacchetto IP di questa marcatura, nell'ACK setta il bit ECE per informare il mittente della presenza di una congestione. Quando il mittente apprende l'informazione, dimezza cwnd.

Viene quindi coinvolto in tutto ciò sia IP (necessario per informare i router, livello di rete) che TCP (sarà il destinatario a livello di trasporto a capire e informare il mittente della congestione).

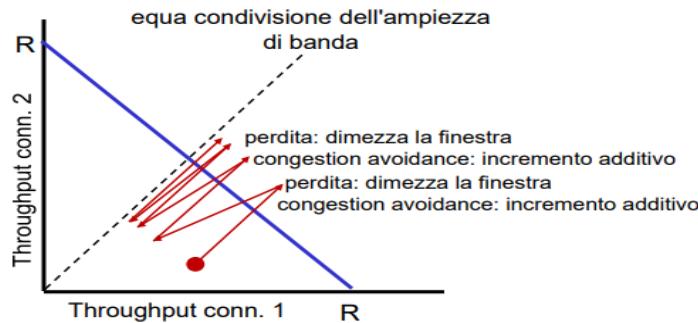
NB. Si ricordi che gli host possono solo decidere di rallentare, cambiare router se uno è congestionato (**instradamento**) è deciso dalla rete.

TCP Fairness

Ci chiediamo a questo punto: **TCP è fair?** Cioè date k connessioni su collegamento R, ognuno ottiene una parte della capacità che è all'incirca R/k ?

Esempio: due sessioni TCP in competizione:

- L'aumento additivo dà una pendenza di 1, quando il throughput aumenta
- la diminuzione moltiplicativa riduce il throughput in modo proporzionale



TCP è fair?

R: Sì, sotto assunzioni idealizzate:

- stesso RTT
- numero fisso di sessioni in congestion avoidance

Firness: tutte le app di rete devono essere "fair"?

Fairness e UDP

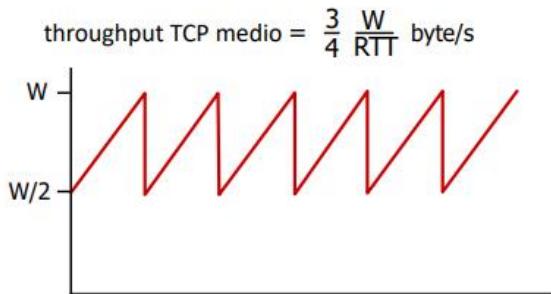
- Le app multimediali spesso non usano TCP
 - Non vogliono che la velocità sia ridotta dal controllo della congestione
- Usano invece UDP:
 - Inviano audio/video a velocità costante, tollerano la perdita di pacchetti
- Non esiste una "polizia di Internet" che controlli l'uso del controllo della congestione

Fairness, connessioni TCP parallele

- L'applicazione può aprire più connessioni parallele tra due host
- I browser web lo fanno, ad esempio il link di velocità R con 9 connessioni esistenti:
 - nuova applicazione usa 1 connessione TCP, ottiene tasso $R/10$
 - nuova applicazione usa 11 connessioni TCP, ottiene tasso maggiore di $R/2$

Descrizione macroscopica del throughput di TCP

- Valore medio del throughput come funzione della dimensione della finestra e di RTT?
 - ignoriamo slow start, assumiamo che ci siano sempre dati da inviare
- W: dimensione della finestra (misurata in byte) quando si verifica una perdita
 - dimensione media della finestra (numero di byte "in volo") è $\frac{3}{4}W$
 - throughput medio è $\frac{3}{4}W$ ogni RTT



Evoluzione delle funzionalità del livello di trasporto

TCP e UDP sono i principali protocolli di trasporto di Internet da ormai 40 anni. Negli ultimi 20 anni sono state sviluppate diverse varianti, ne abbiamo sviluppate alcune come TCP Reno e TCP CUBIC. Altre ancora sono state sviluppate per scenari molto specifici, come per trasferimenti di grandi dimensioni, reti wireless sensibili al mezzo (interferenze) etc...

Scenario	Sfide
Long, fat pipes (trasferimenti di dati di grandi dimensioni)	Molti pacchetti "in volo"; la perdita interrompe la pipeline
Reti wireless	Perdita dovuta a collegamenti wireless rumorosi, mobilità; il TCP la tratta come perdita di congestione
Long-delay links	RTT estremamente elevato
Reti dei data center	Sensibili alla latenza
Background traffic flows	Flussi TCP a bassa priorità, "in background"

Long fat pipes == elevato throughput a fronte di RTT molto alto, vediamo un esempio a cazzo

TCP su “long, fat pipes”

- esempio: segmenti di 1500 byte, RTT di 100ms, vogliamo un throughput di 10 Gbps
- richiede $W = 83\cdot 333$ segmenti in volo
- throughput in termini della probabilità di perdita dei pacchetti, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \cdot \sqrt{L}}$$

→ per raggiungere un throughput di 10 Gbps, occorre un tasso di perdita $L = 2 \cdot 10^{-10}$ – *un tasso di perdita davvero piccolo!*

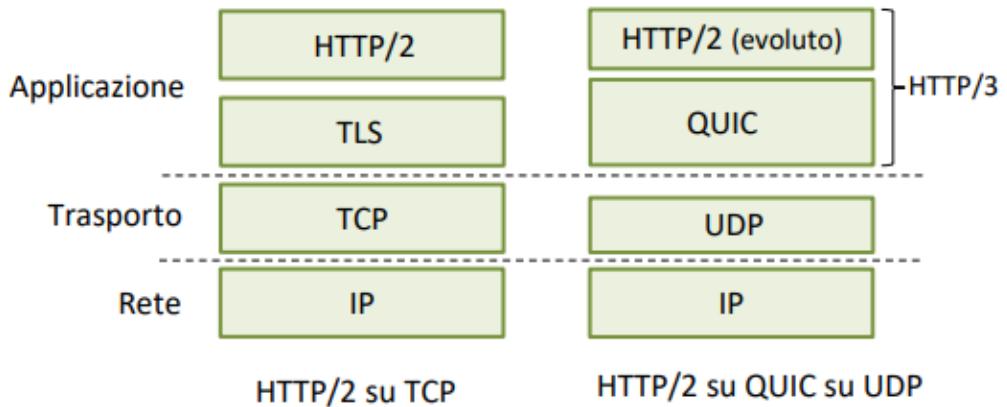
- versioni di TCP per scenari lunghi, ad alta velocità

QUIC: Quick UDP Internet Connections

Si tratta di un **protocollo a livello di applicazione**, sopra UDP.

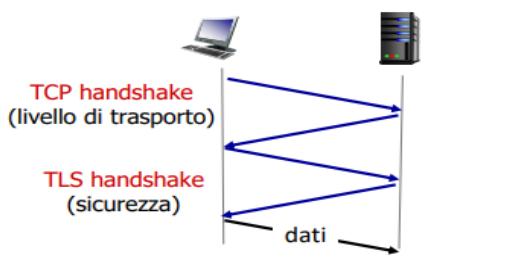
Quelli di Google si sono inventati il QUIC trasferendo parte dei compiti di TCP alle applicazioni appoggiandosi in tutto ciò su UDP.

In particolare, il QUIC è stato applicato ad HTTP per aumentarne le prestazioni. Mentre per HTTP/2 classico lo trovo sopra TLS (che sta a livello applicativo, per la cifratura) sopra TCP (livello di trasporto) sopra IP (livello di rete); HTTP/2 evoluto lo trovo sopra QUIC sopra UDP (trasporto) sopra IP. HTTP/3, che sta per essere standardizzato, userà lo stesso QUIC.

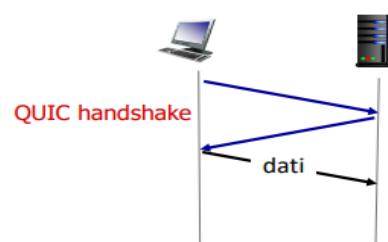


In QUIC è implementata la gestione della connessione (handshake), il controllo degli errori e della congestione (con algoritmi estremamente simili a quelli di TCP, del resto le tecniche sono le stesse con l'accortezza di implementarle a livello applicativo).

Perché fare tutto ciò? **Uno dei vantaggi di QUIC è che si può in un solo RTT stabilire lo stato della affidabilità, del controllo di congestione, di autenticazione e di cifratura.**



- TCP (stato per il trasferimento affidabile e per il controllo della congestione) + TLS (stato per l'autenticazione, autenticazione e per la cifratura)**
- **2 handshake in successione**



- QUIC: stato della affidabilità, controllo della congestione, autenticazione e cifratura**
- **1 handshake**

Inoltre QUIC produce il multiplexing di più “flussi” a livello di applicazione, gestendo separatamente l’affidabilità di ciascuno di essi evitando l’**HOL (head of line blocking)**.

Supponiamo in HTTP 1.1 di voler inviare più richieste. So che posso farlo su una connessione TCP persistente, inviando una richiesta dopo l'altra. Ma che succede se per esempio la seconda richiesta subisce un errore? Non può partire la terza che rimane bloccata (questi blocchi sono detti HOL).

Con QUIC invece ciascuna richiesta è associata ad un suo flusso, ciascuno dei quali implementa il controllo di affidabilità indipendentemente. Quindi se la seconda richiesta subisce un errore verrà recuperata, ma nel frattempo la terza non resta bloccata.

Capitolo 3: riassunto

- Principi alla base dei servizi del livello di trasporto:
 - multiplexing, demultiplexing
 - trasferimento dati affidabile
 - controllo del flusso
 - controllo della congestione
- Installazione, implementazione in Internet
 - UDP
 - TCP

Prossimamente :

- abbandonare la “periferia” (livelli di applicazione e di trasporto)
- per addentrarci nel “nucleo” della rete
 - piano dei dati
 - piano di controllo

Lez 14

Livello di rete

Livello applicativo e di trasporto hanno in comune il fatto di essere implementati esclusivamente negli host (client o server), *dispositivi che si trovano alla periferia della rete e su cui sono in esecuzione le applicazioni di rete*. In particolare sappiamo che il livello di applicazione è dove risiedono le applicazioni di rete, che sono formate da processi comunicanti tra loro attraverso la rete. Questi processi comunicano tra loro utilizzando il livello di trasporto.

Il livello di rete a cosa serve a questo punto? Data un'applicazione, ad esempio l'User Agent HTTP, per mandare messaggi a un altro host incapsula il messaggio in un segmento che viene consegnato al livello di trasporto. In particolare HTTP usa una connessione TCP, che garantisce di inviare un flusso di byte dall'altra parte in modo affidabile. Nel livello di trasporto i dati di

livello applicativo sono combinati con un'intestazione con informazioni di controllo utili al livello di trasporto. Il compito del livello di rete sarà quindi quello di mandare effettivamente i pacchetti provenienti dall'host mittente (che contengono i segmenti del livello di trasporto) all'host di destinazione, dove il livello di rete tirerà fuori dal pacchetto di rete (detto **datagramma**) il **segmento**, lo consegnerà all'adeguato livello di trasporto (nel nostro caso TCP) ed infine da lì il livello di trasporto (dopo aver eseguito l'algoritmo specifico, ad esempio mandare ACK etc..) se tutto è andato a buon fine decapsula il **messaggio** vero e proprio per poi consegnarlo al livello di applicazione.

Tuttavia **per poter trasferire i pacchetti lungo il percorso, il livello di rete deve essere implementato non solo negli host, ma anche in tutti gli altri dispositivi di rete** (in particolare nei **router**).

In tutti i router è presente un'implementazione del livello di rete che svolge l'importante **funzione di inoltro (forwarding)**: quando in un router arriva un pacchetto in ingresso, deve inoltrarlo nell'appropriato collegamento in uscita (inoltro svolto da ogni router, man mano il pacchetto segue la strada nella rete verso destinazione).

Quindi i router implementano solo ed esclusivamente fino a livello di rete (livello applicativo e di trasporto negli host).

Come detto, la funzione di inoltro si occupa di mandare il pacchetto, arrivato in ingresso, verso uno specifico collegamento in uscita. Per farlo vengono applicate delle regole prestabilite dalla **funzione di instradamento (routing)** che determina, più in generale, il percorso dei pacchetti dall'origine alla destinazione secondo specifici **algoritmi di instradamento**.

(rete come grafo con nodi router e host, voglio trovare i cammini minimi)

funzioni del livello di rete

- **inoltro (forwarding):** trasferisce i pacchetti da un collegamento di ingresso di un router al collegamento di uscita appropriato del router
- **instradamento (routing):** determina il percorso seguito dai pacchetti dall'origine alla destinazione
 - *algoritmi di instradamento*

analogia: fare un viaggio

- **inoltro:** attraversamento di uno svincolo seguendo le indicazioni dei cartelli
- **instradamento:** pianificazione dei percorsi verso tutte le destinazioni scegliendo tra i molteplici possibili e conseguente installazione dei cartelli



Inoltro come scegliere la direzione in uno svincolo, instradamento tutta la strada prestabilita dal navigatore. I cartelli sono stati messi a monte sapendo tutta la strada che devo compiere per andare verso una certa destinazione (quindi inoltro in funzione di instradamento).

Possiamo vedere le funzioni di inoltro e instradamento come appartenenti a due piani separati nel livello di rete: **piano dei dati** e **piano di controllo**.

Il **piano dei dati** si occupa di **funzioni locali**, eseguiti dal singolo router e determina semplicemente come i pacchetti in arrivo a una porta di ingresso del router devono essere inoltrati verso una sua porta d'uscita.

Il **piano di controllo** invece, occupandosi del problema dell'instradamento, implementa una vera e propria **logica di rete** molto più ampia della “locale” logica di inoltro. Viene infatti determinato come i pacchetti sono instradati tra i router lungo un percorso da host d'origine a host di destinazione. Esistono due principali approcci per implementare le funzioni del piano di controllo:

- **Algoritmi di instradamento tradizionali:** sono implementati in ogni singolo router e quindi l'instradamento avviene in modo distribuito secondo lo scambio d'informazioni con gli altri router
- **Software-Defined-Networking (SDN):** è un approccio più recente per cui la funzione di instradamento viene “scaricata” a un componente terzo esterno, un server remoto.

Vediamo l'inoltro e instradamento più nel concreto.

Arriva un pacchetto al router. Affinché questo venga inoltrato in modo

adeguato (e venga quindi scelto il collegamento di uscita corretto) il router guarda **i campi di intestazione del datagramma**, in particolare si guarda **l'indirizzo di destinazione**. Sulla base di questa informazione, secondo criteri specifici, il router accede a una **Tabella di Inoltro Locale**, dove esiste una corrispondenza per ogni possibile valore d'intestazione con il corrispettivo collegamento di uscita.

Ovviamente non si ha una riga per ogni possibile destinazione ma a ogni riga corrisponde un blocco di indirizzi.

Come sono definite le Tabelle di Inoltro Locali? Inizialmente venivano codificate a mano, ma chiaramente con la rete sempre più grande non è soluzione scalabile rispetto al numero di router che aumenta sempre di più e non è soluzione in grado di intervenire in modo efficace qualora vi siano dei cambiamenti.

Per questo nei router sono stati ben presto implementati gli algoritmi di instradamento per conciliare la funzione di inoltro automaticamente, definendo e mantenendo valide le tabelle di inoltro (algoritmi di instradamento tradizionali, vedi sopra). L'altra soluzione per definire le tabelle è, come detto prima, quella di usufruire di un SDN, un server o cluster remoto che eseguendo gli algoritmi di instradamento calcola le tabelle di inoltro installandole poi sui vari router (in questa ottica il piano di controllo nel singolo router risulta praticamente assente).

Come detto, il servizio principale offerto dal livello di rete è il trasferimento di pacchetti da un host mittente a un host destinatario seguendo un percorso formato da più collegamenti e router.

Questo servizio può essere specificato con ulteriori caratteristiche, definendo il **modello di servizio** per il “canale” che trasporta i datagrammi dal mittente al destinatario (cioè, sapendo che in generale il servizio è il trasferimento di pacchetti, esistono modi diversi per far sì che questo servizio sia soddisfatto).

Ad esempio, se necessito l'invio di singoli datagrammi, potrei volere un servizio con consegna garantita con ad es. un ritardo non superiore a 40 ms.

Se parliamo invece di invio di flussi di datagrammi, altri requisiti che potrebbero venire in mente sono l'ordine di consegna, richiedere una minima ampiezza di banda garantita (utile ad es. per applicazioni real time) oppure ancora far sì che, se mando pacchetti distanziati uno a uno secondo un certo lasso di tempo, il destinatario riceva i pacchetti distanziati ancora secondo quel lasso di tempo (utile ancora per applicazioni in real time).

In base all'architettura di rete a cui si fa riferimento, si hanno diverse garanzie.

Architettura di rete	Modello di servizio	Garanzie di qualità del servizio, <i>quality of service (QoS)</i>			
		Banda	Consegna	Ordine	Temporizzazione
Internet	best effort	nessuna	no	no	no
ATM	Constant Bit Rate	tasso costante	sì	sì	sì
ATM	Available Bit Rate	min. garantita	no	sì	no
Internet	Intserv Guaranteed (RFC 1633)	sì	sì	sì	sì
Internet	DiffServ (RFC 2475)	possibile	possibilmente	possibilmente	no

(da non sapere a memoria)

In apparenza il modello di Internet **best effort** è il peggiore di tutti in quanto non offre garanzie di alcun tipo (no garanzie sulla consegna, sui tempi, ordini, larghezza di banda riservata e temporizzazione).

Perché questo modello è quello ad aver avuto più successo?

Gli altri modelli, nel fornire garanzie, perdono in termini di semplicità.

È proprio la **semplicità del meccanismo** racchiuso nel best effort ad aver consentito l'ampia diffusione di questo modello.

Inoltre, nonostante non vi siano garanzie formali circa ritardi e perdite di pacchetti, già sappiamo che **se la capacità è molto più grande del traffico che transita sul collegamento, il ritardo sarà trascurabile e con esso anche la probabilità di perdita di pacchetti**.

Inoltre abbiamo visto come (ad esempio in protocolli di streaming del tipo DASH) **protocolli definiti a livello applicativo permettono di reagire a cambiamenti della banda disponibile**.

Inoltre, sempre riguardo le applicazioni, abbiamo visto che queste tendenzialmente non sono erogate da un singolo punto, ma **replicate in diversi datacenter (CDN)** che consentono di fornire servizi migliori da più luoghi diversi.

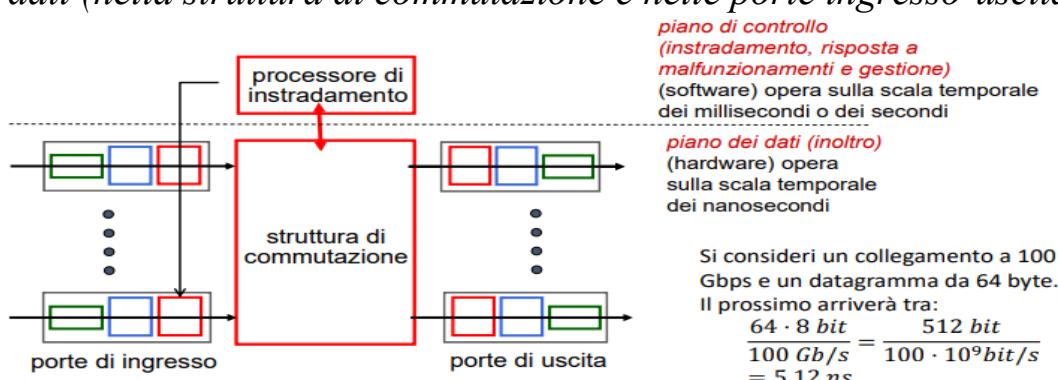
In questo senso l'utilizzo di protocollo TCP da parte di più applicazioni, che implementa il controllo della congestione, aiuta a far sì che non essendoci congestione no perdita pacchetti, no necessità di banda minima etc...

Nonostante l'assenza di garanzie quindi il modello best effort per internet è quello che ha preso maggiormente piede (anche perché applicazioni ad es. real time che necessitano di queste garanzie riescono a risolverle in altri modi) -> no garanzie -> semplicità, facile implementazione e scalabilità.

Architettura del router

Un router ha per definizione più porte d'ingresso e di uscita. Al centro si ha una **struttura di commutazione**, che svolge la funzione di transito dei pacchetti da una porta d'ingresso a una specifica porta di uscita. Si ha in aggiunta a ciò un **processore di instradamento**, che segue la logica degli algoritmi di instradamento per determinare quali sono le tabelle di inoltro locale e in generale configurare la struttura di commutazione.

Si riconosce ancora la differenza tra piano di controllo (nel processore) e piano di dati (nella struttura di commutazione e nelle porte ingresso-uscita).

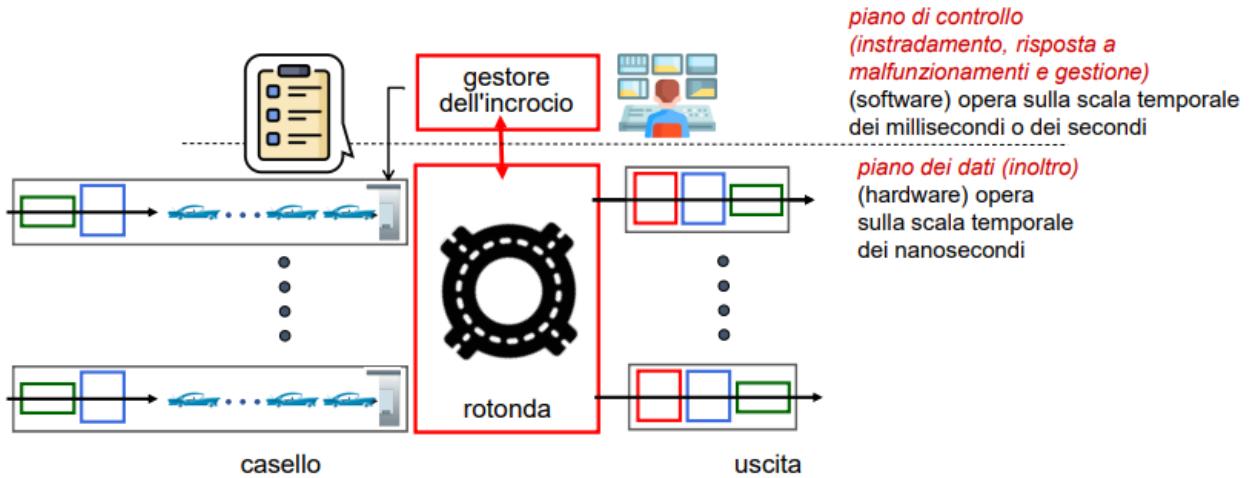


Piano di dati e piano di controllo operano su scale temporali molto diverse.

Il piano dati deve prendere i pacchetti d'ingresso e mandarli nella giusta uscita (inoltro), affinché la commutazione non sia collo di bottiglia (si mantenga all'incirca la stessa velocità che si ha in trasmissione considerando un collegamento che opera in Gb/s) è necessario che il tempo in cui opera sia nella *scala temporale dei nanosecondi*.

Al contrario il piano di controllo e quindi il processore di instradamento può lavorare in scale temporali più larghe, **tra i millisecondi e i secondi**. Ciò è dovuto al fatto che, ad es., non è necessario calcolare così frequentemente nuove tabelle di inoltro.

analogia per la architettura generica di router



Vediamo più nel dettaglio cosa è presente nelle varie componenti del router, partendo dalle **porte di ingresso**. La prima funzione svolta dalla porta di ingresso è quella di **terminazione di linea, legata al livello fisico**. Nel caso di un collegamento cablato ad es. la porta d'ingresso sarà la terminazione elettrica del filo.

Si passa poi al **livello di collegamento** in cui i bit sono raggruppati in unità chiamate **Frame** e interpretate come PDU (protocol data unit) di un protocollo di collegamento. In particolare a questo livello avviene il decapsulamento: viene tirato fuori il datagramma che viene consegnato al **livello di rete**.

Tipicamente livello fisico e di collegamento sono implementate in hardware, mentre il livello di rete è una via di mezzo. Hardware se consideriamo ciò che abbiamo appena descritto, software se consideriamo programmi che girano nel processore di instradamento.

Nelle architetture più recenti, la decisione sulla porta di destinazione avviene *localmente alla porta d'ingresso secondo una commutazione decentralizzata*. Due approcci:

- **Tradizionale (inoltro basato su destinazione)**: l'indirizzo di destinazione del pacchetto viene confrontato con la tabella di inoltro per decidere così la destinazione

- **Inoltro generalizzato**: la decisione viene presa sulla base anche di altri campi di intestazione, oltre che l'indirizzo

Qualsiasi sia l'approccio scelto si può vedere questa azione di ricerca e inoltro come caso specifico di “**match plus action**”.

Concentriamoci ora sull'approccio tradizionale. Dato un indirizzo a 32 bit, ho potenzialmente ordine di 2^{32} destinazioni (non considero quelle con tutti 0 e tutti 1, casi specifici).

Sarebbe impensabile considerare una tabella d'inoltro locale con per ogni riga una destinazione, per questo nelle tabelle di inoltro si considerano **blocchi di indirizzi**.

Destinazione basata sull'indirizzo di destinazione

forwarding table	
Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1
11001000 00010111 00011001 00000000 through 11001000 00010111 00011001 11111111	2
otherwise	3

Da valori con ultimi 13 bit da 1000 000000000 a 1011 111111111 destinazione 0, da bit successivo 11000 00000000 a 11000 11111111 destinazione 1 etc...

Che succede se gli intervalli non si dividessero così bene, cioè ci fossero destinazioni che cadono in stesse sequenze degli ultimi 13 bit?

<pre>11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111</pre>	0
<pre>11001000 00010111 00010000 00000100 through 11001000 00010111 00010000 00000111</pre>	3

La soluzione è quella di adottare la logica del **prefisso** più lungo. Nel momento in cui confronto l'indirizzo di destinazione con i valori in tabella, prendo il blocco a cui corrisponde il prefisso più lungo. Vediamo un esempio:

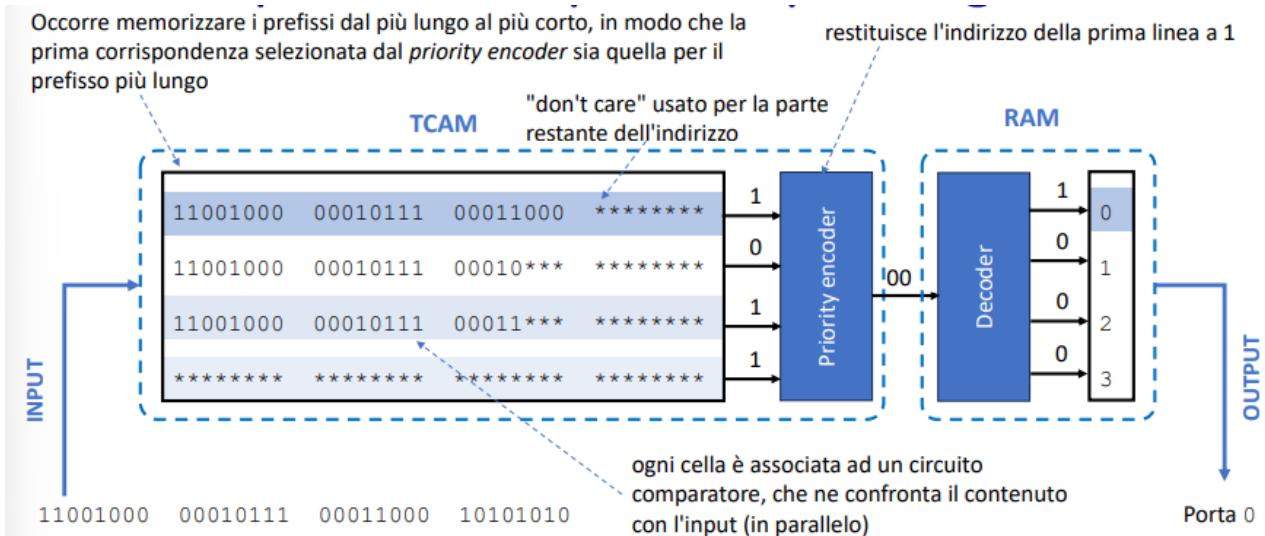
Intervallo di indirizzi di destinazione	Interfaccia
11001000 00010111 00010*** *****	0
11001000 00010111 00011000 *****	1
11001000 00010111 00011*** *****	2
altrimenti	3

esempi: 11001000 00010111 00010110 10100001 quale interfaccia?
 11001000 00010111 00011000 10101010 quale interfaccia?

Prefisso deriva dal fatto che a ogni blocco corrisponde la serie di bit con un certo prefisso. Nel primo caso, scelgo destinazione 0 perché i bit 110 non corrispondono agli 000 necessari per la destinazione 1, nel secondo caso invece, nonostante avrei due match possibili (0 e 1) scelgo proprio destinazione 1 poiché i bit coincidono ed il prefisso è più lungo (ho i 3 zeri necessari).

Questa operazione viene implementata in hardware tramite delle **TCAMs** (ternary content addressable memories). Si tratta di memorie particolari, che seguono la logica del **content addressable**: *un indirizzo IP a 32 bit è passato alla memoria che restituisce il contenuto della tupla nella tabella di inoltro corrispondente a quell'indirizzo in tempo sostanzialmente costante.*

Non si tratta di confrontare tutte le celle, scorrendole una dopo l'altra. *Per garantire tempistiche così veloci è necessario associare un comparatore per ogni riga, confrontando il contenuto dell'indirizzo in parallelo per ogni riga della tabella.* Occorre memorizzare i prefissi dal più lungo al più corto in modo che la prima corrispondenza selezionata sia quella per il prefisso più lungo! Ciò viene garantito dal **Priority Encoder** (lui permette di selezionare il bit a 1 più in alto), che passa poi il risultato a un modulo di RAM che lo decodifica e sceglie finalmente il collegamento d'uscita corrispondente.



La TCAM mi serve quindi per memorizzare i prefissi vari.
A differenza delle memorie “classiche” che prendono un indirizzo e restituiscono una cella/valore, la TCAM come visto funziona al contrario.
Prende in input un valore e restituisce in output un indirizzo (content addressable).

Struttura di commutazione (switching fabric)

Si occupa di trasferire i pacchetti dal collegamento di ingresso a quello di uscita appropriato. Per farlo è necessario che sia molto veloce, altrimenti potrei avere accodamento di pacchetti sulle porte di ingresso.

Affinché ciò non avvenga vogliamo che il **tasso di trasferimento** (cioè la velocità con la quale i pacchetti vengono trasferiti dalla porta di input a quella di output) sia veloce quanto la capacità complessiva delle porte di ingresso.
Ovvero se R è il tasso di linea e si hanno N ingressi, vogliamo che il tasso di trasferimento **sia $N \times R$** . Ciò perché se immagino che mi siano arrivati contemporaneamente N pacchetti, uno per ingresso, ci vorranno L/R prima che ne arrivino altri e nel frattempo con $N \times R$ avrò trasferito i pacchetti che mi erano arrivati. (si ignorano così i ritardi per pacchetti accodati nelle porte d’ingresso).

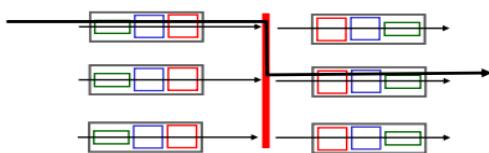
È per questa ragione che negli esercizi in cui si parla di accodamenti e ritardi si trascurano ritardi di elaborazione e di accodamento in input, si considerano solamente ritardi di coda sulla porta di uscita.

Tre approcci per la Commutazione: **in memoria**, **tramite bus** e **attraverso rete di interconnessione**.

Commutazione in memoria: è l'approccio utilizzato dai primi router. Questi erano implementati come calcolatori tradizionali, in cui le porte erano dispositivi di I/O. *Molto semplicemente in quest'architettura i pacchetti sono copiati in memoria e il processore di instradamento determina la porta di uscita.* Si nota che con quest'approccio ogni pacchetto deve transitare in memoria due volte, per cui *la velocità è limitata proprio dall'ampiezza di banda della memoria*.

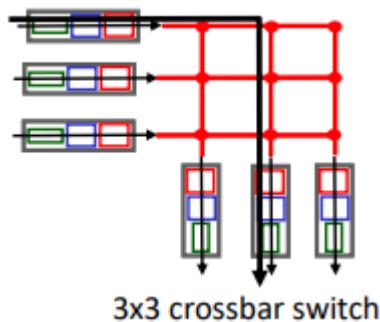


Commutazione tramite bus: l'idea è di “tirare fuori” il processore di instradamento e fare in modo che *la porta d'ingresso possa trasmettere direttamente i dati sull'opportuno collegamento di uscita servendosi di un bus che viene attraversato una sola volta*, ma che **resta comunque collo di bottiglia (bus contention)**. Inoltre il bus può essere usato da un solo pacchetto alla volta, quindi i pacchetti dovranno accodarsi anche se destinati a porte differenti. Tuttavia router di questo tipo con bus a 32 Gbps sono buone soluzioni per router domestici.

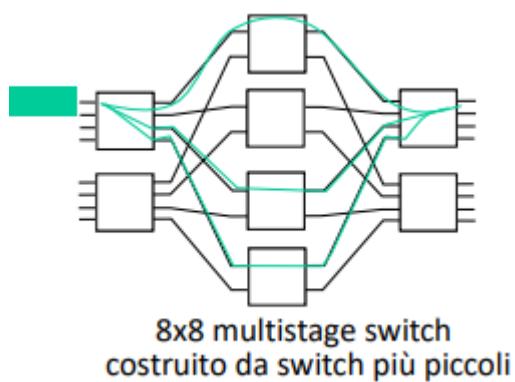


Commutazione attraverso rete di interconnessione: vi sono diversi approcci a questa soluzione, vediamo rapidamente la soluzione a **Crossbar Switch**. Ogni porta d'ingresso ha un bus dedicato (orizzontale in figura) e ogni porta d'uscita un bus dedicato (verticale). Ho $n \times n$ switch e un collegamento dal primo ingresso al terzo ad es. se è aperto lo switch in riga 1 colonna 3 della Crossbar (matrice di commutazione). Se un'altra porta d'ingresso il cui collegamento non contrasta uno già attivo (es riga 2 colonna 1) **allora posso trasferire pacchetti**

in parallelo! Vantaggio: posso far transitare contemporaneamente più pacchetti destinati a porte di uscita differenti.



Questa è un'architettura molto complessa e costosa ($n \times n$ switch), in generale non si usano spesso queste architetture ma si simulano usando delle reti a più stati di switch più piccoli (**multistage switch**). (si perde un po' di parallelismo, se ho ad es. due porte di uscita diverse i cammini potrebbero incrociarsi da qualche parte e quindi non potrei far transitare in parallelo due pacchetti).



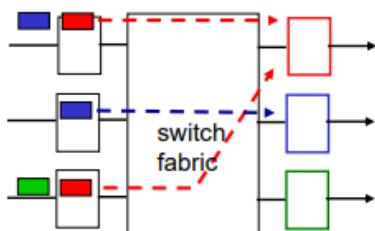
■ **sfruttare il parallelismo:**

- frammenta il datagramma in celle di lunghezza fissa all'ingresso
- commutare le celle attraverso la rete di commutazione, riassemblare il datagramma in uscita

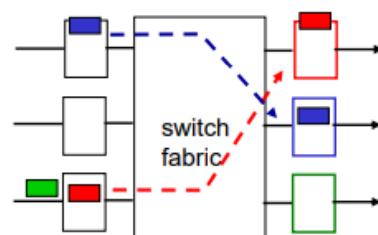
Perché tutte queste complicazioni? Si vogliono per alcuni router capacità impressionanti, con capacità di commutazione fino a centinaia di Tbps.

Parlando dell'accodamento sulle porte d'ingresso anche a questo livello si può presentare la problematica del **blocco in testa alla coda** (**HOL** head of the line blocking). Ho due pacchetti rossi e uno blu. In un colpo posso farli transitare, ma il rosso sotto è bloccato. La cosa peggiore è che dietro il rosso avevo un pacchetto verde che poteva anche transitare, ma è rimasto bloccato per colpa del rosso.

- **Blocco in testa alla coda [Head-of-the-Line (HOL) blocking]:** il datagramma accodato all'inizio della coda impedisce agli altri in coda di avanzare



contesa della porta di uscita: soltanto un datagramma rosso può essere trasferito.
Il pacchetto rosso in basso è **bloccato**



dopo il trasferimento di un pacchetto: il pacchetto verde sta sperimentando il **blocco in testa alla coda**

Network

Accodamento in uscita

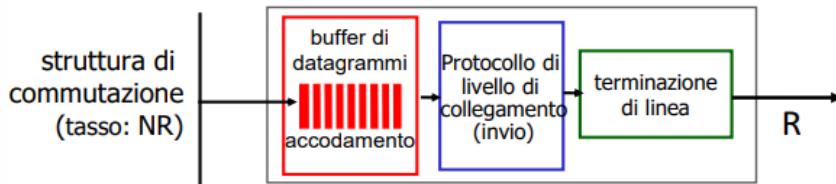
*La struttura di commutazione, come detto, può trasferire dati con un tasso **NR**. La terminazione di linea può tirarli fuori con velocità **R**. Vi è quindi possibilità che contemporaneamente vi siano più pacchetti d'ingresso di quanti non possano esserne spediti fuori da router. È necessario per questa ragione gestire un **Buffer** di datagrammi e un relativo accodamento.* (si ha in uscita,

come in ingresso, parte di livello di rete, di collegamento e di livello fisico).

Accodamento in uscita



questa è una slide importante



- **Buffering** richiesto quando i datagrammi arrivano dalla struttura di commutazione più velocemente del tasso di trasmissione del collegamento. **Drop policy:** quale datagramma scartare se il buffer non è sufficiente?
- **Disciplina di scheduling** sceglie tra i datagrammi in coda quale trasmettere

- I datagrammi possono essere persi a causa di congestione, mancanza di buffer
- Schedulazione con priorità
 - chi ottiene le migliori prestazioni, neutralità della rete

Network Layer: 4-33

Ma come gestire il buffer dei datagrammi?

-Quanto deve essere grande?

-Con che ordine tiro fuori i datagrammi?

-In caso di overflow scarto l'ultimo pacchetto arrivato o uno in coda?

Rispondiamo alla prima domanda. La questione è ancora dibattuta. Secondo l'RFC 3439 (rule of thumb) dato un RTT tipico di 250 ms e la capacità C del collegamento li dovrei moltiplicare per determinare la dimensione del buffer. Se ho quindi ad es. C = 10Gbps allora avrei dimensione del buffer pari a 2.5 Gbit.

Una raccomandazione basata su studi più recenti ha mostrato che se ho N flussi indipendenti la dimensione del buffer consigliata è $RTT \times C / radN$.

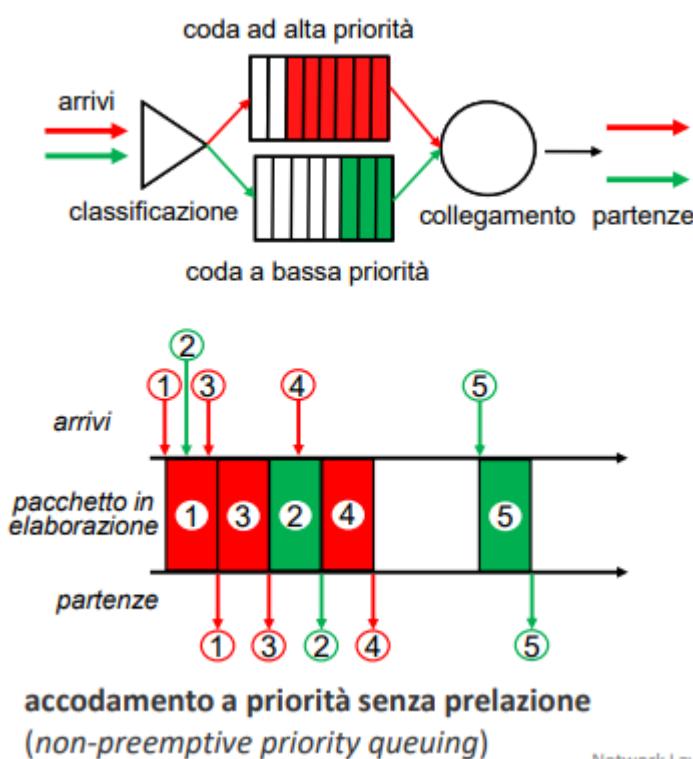
Ma perché dovrei ridurre il buffer? Anzitutto per motivi di costo, ma più importante è il fatto che la bufferizzazione fa sì che nonostante eviti la perdita di pacchetti se ho tanti pacchetti davanti riscontro **un ritardo RTT sempre più elevato!** Ciò comporta scarse prestazioni delle applicazioni real time, mittenti TCP meno reattivi alla congestione e alla perdita di pacchetti. Si ricorda infatti il controllo della congestione basato su ritardo: “mantenere il collegamento collo di bottiglia sufficientemente pieno (occupato) ma non più pieno (fuller)”.

In generale quindi bufferizzazione necessaria perché talvolta arrivano più dati di quanti non ne possa inviare, ma un buffer troppo grande non è la soluzione (anzi, creerebbe problemi).

Vediamo ora di rispondere alle due successive domande. Quali pacchetti scartare se la coda è piena? Due approcci principali: **Tail Drop** (scarto il pacchetto in arrivo) e **Priorità** (alcuni pacchetti sono marcati con una certa priorità, scarto quello con priorità più bassa, ad es. utile per far passare pacchetti con l'obiettivo di segnalare la congestione).

Rispondiamo ora all'ultima domanda: in che ordine tiro fuori i pacchetti dalla coda? Ci sono diverse possibilità:

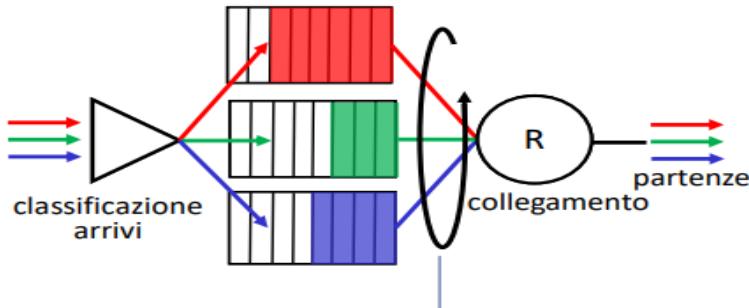
- **FIFO** (First In First Out): I pacchetti sono trasmessi in ordine di arrivo alla porta di uscita.
- **Schedulazione con priorità**: il traffico in arrivo è classificato ed accodato per classi (qualsiasi campo di intestazione può essere usato per la classificazione). Viene inviato il pacchetto dalla coda non vuota con priorità più alta (ho più code con priorità diverse). Quindi un pacchetto non sarà mai trasmesso fintanto che ci sarà un pacchetto con priorità più alta in attesa, anche se è arrivato dopo!



Network Layer: 4

Arriva il pacchetto 1, nel frattempo che trasmetto sono arrivati il pacchetto 2 e 3. Il 2 è arrivato prima del 3 ma ha priorità più bassa, quindi trasmetto 3 prima di 2. Arriva 4 mentre trasmetto 2, anche se 4 ha priorità più alta non può interrompere la trasmissione di 2 quindi aspetta che finisca e poi parte.

- **Round Robin (RR) Scheduling:** anche stavolta le code sono divise secondo classi di priorità, *ma stavolta il collegamento non scandisce sempre dall'alto verso il basso ma cicla*.



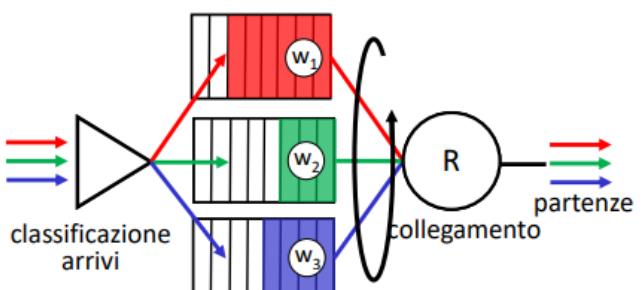
- **Weighted Fair Queueing (WFQ):** generalizza Round Robin. Stavolta non mi limito a ciclare per le code inviando un pacchetto per volta, ma cerco di garantire più servizio per i pacchetti con priorità più alta. Ciascuna classe i ha infatti un peso w_i e riceve una quantità ponderata di servizio ad ogni ciclo pari a $w_i / \sum_j w_j$

Weighted Fair Queueing (WFQ):

- generalizza Round Robin
- Ciascuna classe, i , ha un peso, w_i , e riceve una quantità ponderata di servizio in ogni ciclo :

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

- garanzia di larghezza di banda minima (per classe di traffico)



Barra laterale: Neutralità della rete

Cos'è la neutralità della rete (*net neutrality*)?

- **tecnica:** come un ISP dovrebbe condividere/allocare le proprie risorse
 - la schedulazione dei pacchetti e la gestione dei buffer sono i *meccanismi*
- **Principi sociali e economici**
 - proteggere la libertà di espressione
 - Incoraggiare l'innovazione, la competizione
- **Far rispettare politiche e leggi**

Ogni paese ha il proprio approccio alla neutralità della rete

2015 US FCC *Order on Protecting and Promoting an Open Internet*: tre regole definite “clear, bright line”:

- **no blocking** ... “non bloccherà i contenuti, le applicazioni, i servizi o i dispositivi non dannosi leciti, fatta salva una ragionevole gestione della rete.”
- **no throttling** ... “non devono pregiudicare o degradare il traffico Internet lecito sulla base del contenuto, dell'applicazione o del servizio Internet o dell'uso di un dispositivo non dannoso, fatta salva una ragionevole gestione della rete.”
- **no paid prioritization.** ... “non deve impegnarsi nella prioritizzazione a pagamento”

Nel 2017, la *Restoring Internet Freedom Order* ha annullato questi divieti, concentrandosi invece sulla trasparenza degli ISP.

ISP: telecommunications or information service?

Un ISP è un "servizio di telecomunicazioni" o un fornitore di "servizi di informazione"?

- la risposta è importante dal punto di vista normativo!

US Telecommunication Act del 1934 e 1996:

- *Titolo II*: impone "common carrier duties" ai *servizi di telecomunicazione*: tariffe ragionevoli, non discriminazione e richiede una regolamentazione
- *Titolo I*: si applica ai *servizi di informazione*:
 - no common carrier duties (*non regolamentato*)
 - ma concede alla FCC l'autorità "... necessaria per l'esecuzione delle sue funzioni"

Lez 15

Parliamo ora nello specifico di come è implementato il livello di rete in Internet.

Vi sono tre componenti principali: **algoritmi di instradamento** (implementabili come visto *direttamente nei router* o *tramite SDN* (server remoto), il loro output sono le **tabelle di inoltro locali** ai router), **protocollo IP** e **protocollo ICMP** (necessario per codificare gli errori e mandare altro genere di segnalazioni).

Si ricorda che la tabella di inoltro locale è una struttura dati che associa un indirizzo IP (tramite ragionamento di prefisso) a un opportuno collegamento in uscita, si tratta del punto di giunzione tra instradamento e l'inoltro. La tabella è generata secondo regole dettate dall'instradamento (*piano di controllo*), e viene eseguita "ciecamente" dal router (*piano di dati*) per garantire che man mano si raggiunga la destinazione.

Protocollo IP

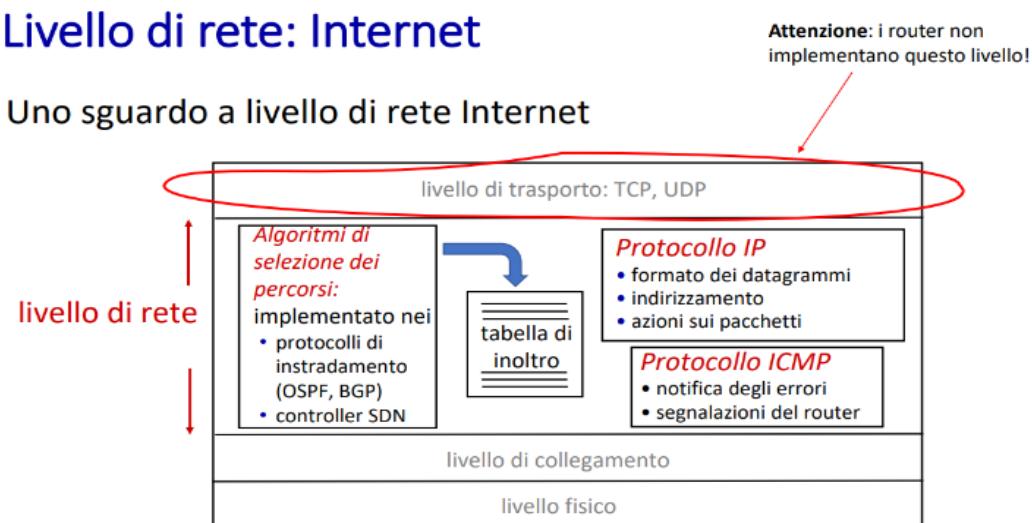
Si ricorda che un protocollo, in generale, definisce il formato dei messaggi scambiati, l'ordine e le azioni che dobbiamo compiere quando inviamo un pacchetto, lo riceviamo o avviene altro (tipicamente scadenza di un timer).

Necessitiamo di un protocollo a livello di rete perché nel caso in cui due entità dialoghino tra loro è necessario che questo dialogo sia governato da un protocollo (affinché si capiscano a vicenda), nel caso del livello di rete il protocollo principalmente utilizzato è quello IP (**Internet Protocol**).

IP e TCP sono i protocolli più importanti di Internet.

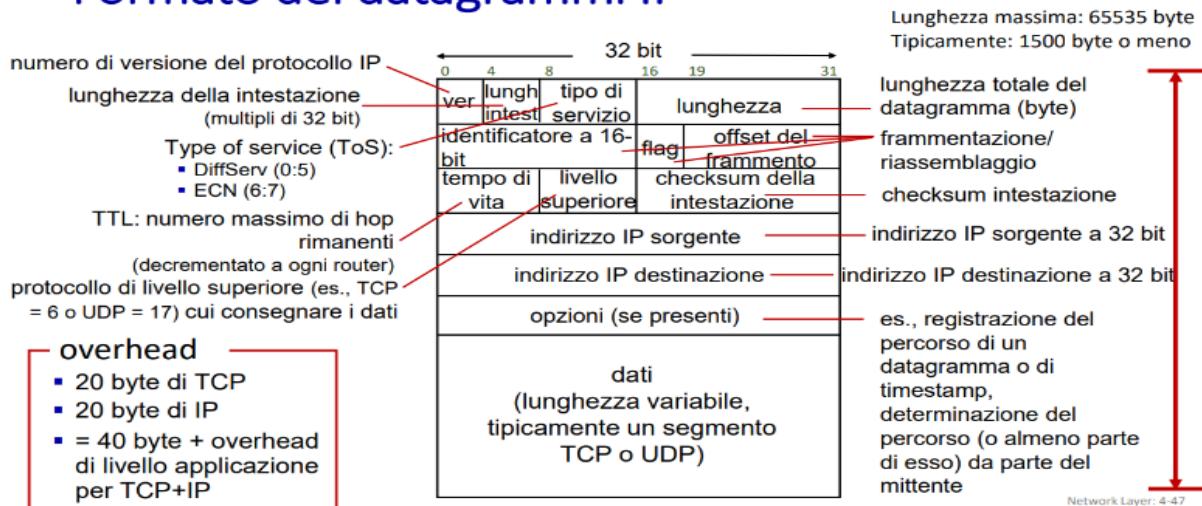
Il protocollo IP ci dice *come sono formati i Datagrammi, come sono formati gli indirizzi e come dobbiamo manipolare i pacchetti*.

Livello di rete: Internet



Vediamo ora il formato di IP (studia il formato delle varie DPU che le mette all'esame, TCP UDP IP IPv6 etc...)

Formato dei datagrammi IP



-**Ver** va dall'indice 0 a 3 (copre i primi 4 bit) ed è necessario per indicare *la versione di IP che si sta utilizzando* (quella che stiamo coprendo adesso è IPv4, IPv6 sarà più semplice);

-**Lunghezza Intestazione** va dai 4 ai 7 ed è indicata con multipli di parole a 32 bit (se trovo 0001 allora 32 bit). Essendo la lunghezza minima dell'intestazione 20 byte == 160 bit, il minimo che troverò qui è 0101, cioè $5 \times 32 = 160$ bit);

-**Type of Service (ToS)**: è spezzato in due parti, una è utilizzata da **DiffServ** per avere servizi differenziati per classi di traffico (l'abbiamo visto quando

abbiamo parlato di modelli di servizio diversi in Internet dal best effort), l'altra dedicata ai *bit ECN* necessari per il controllo della congestione. Quando abbiamo parlato della congestione abbiamo visto due approcci principali: quelli usati tradizionalmente in cui *si deduce la congestione sulla base di sintomi come perdite e ritardi*, e l'altro approccio in cui *la rete informa direttamente gli host (**ECN, explicit congestion notification**)*.

È un meccanismo che integra il livello di trasporto e rete: quando un router è congestionato mette i due bit legati all'ECN a 1 e il destinatario che riceve il datagramma, passando al livello di trasporto, nel caso TCP invia un ACK con il bit ECE (explicit congestion echo) attivato per cui il mittente ricevuto l'ACK dimezzerà la finestra di congestione e mette a 1 i bit CWR (congestion window reduced). (nb potrebbe chiedere sto ragionamento all'esame)

Perché 2 bit per ECN? Perché *il router non si deve limitare a notificare della congestione, ma deve anche sapere se mittente e destinatario sono abilitati a questo meccanismo* (standard più recente, non è detto che sia implementato in tutti gli host, se non è abilitato in uno degli host non ha senso sprecare tempo); -**Lunghezza** occupa 16 bit, campo dedicato alla lunghezza del datagramma. Se ho 16 bit il numero più grande che posso rappresentare, senza segno, è $2^{16}-1 = 65535$ byte. *Tuttavia la maggioranza dei datagrammi non sono così grandi, raggiungono circa 1500 byte o meno.* Ciò è dovuto ai limiti imposti dalla **MTU** (il livello di collegamento invia i dati sotto forma di unità dette Frame, che possono avere payload limitato detto MTU che nel caso Ethernet è 1500 byte); -**TimeToLive** (TTL) occupa 7 bit e rappresenta il numero massimo di hop rimanenti, viene decrementato al passaggio di ogni router. Rappresenta quanto tempo alpiù un datagramma può girare in rete, se viene raggiunto 0 allora il router butta il pacchetto. **Perché serve TTL?** Se nella rete, per qualche motivo, è presente un “ciclo” per cui il pacchetto torna sempre allo stesso router e più in generale resta in giro continuamente nella rete allora, oltre che occupare buffer e prestazioni di router inutilmente, se disgraziatamente arriva ad un host destinatario in un momento non previsto con numero di sequenza (caso TCP) adeguato allora viene accettato quando non c’entrava niente con i dati che l’host si aspettava di ricevere. *È necessario quindi, affinché resti valida la correttezza dei protocolli di trasporto affidabili e non si sprechino inutilmente risorse di rete, che i pacchetti dopo un po’ scompaiano;*

-**Livello Superiore:** svolge esattamente lo stesso ruolo del numero di porta nel protocollo TCP o UDP; quando si arriva a destinazione infatti deve essere decapsulato il segmento dal datagramma e dobbiamo sapere a che protocollo bisogna affidare questo segmento (TCP, UDP o altro). È quindi anche qui

presente una forma di multiplazione-demultiplazione simile a quella vista nel livello di trasporto;

-**Checksum** calcolata nello stesso modo visto per UDP, ma limitatamente all'intestazione. Perché averla sia a livello di rete che di trasporto? In linea teorica si tratta di protocolli indipendenti (potrei avere IP con protocollo del livello di trasporto che non supporta la checksum, più in generale livello superiore e inferiore che non implementano checksum).

In IPv6 la checksum è stata rimossa, perché? Abbiamo detto che in IPv4 calcoliamo la checksum su tutta l'intestazione, ma quando inoltro un pacchetto attraverso un router so che il campo d'intestazione TimeToLive cambia sempre! *Quindi devo per ogni router ricalcolare la checksum!* Esiste un trucco per farlo rapidamente sapendo di aver cambiato solo un campo, ma nei router l'inoltro è nei limiti dei nanosecondi quindi è comunque un'operazione in più. (ricordati come si calcola la checksum che potrebbe chiederla);

-**Indirizzi IP del mittente e destinatario**, indicati a 32 bit;

-**Opzioni**: variabili, senza questa l'intestazione occupa 20 byte. Considerando TCP, abbiamo 20 byte occupati per rappresentare il segmento, per un **Overhead** totale per IP di lunghezza minima pari a 40 byte (20 TCP + 20 IP) + overhead del livello di applicazione per TCP+IP. Ciò significa che se trasmetto un datagramma stiamo usando 40 byte solamente per le intestazioni;

-**Identificatore a 16 bit, flag e offset del frammento** necessari per la frammentazione e riassemblaggio dei datagrammi.

Frammentazione dei datagrammi IP

Sappiamo che l'unità massima di trasmissione (**MTU**) è *la massima quantità di dati che un Frame a livello di collegamento può trasportare*, in base al tipo di collegamento avrò una MTU diversa.

Quindi è chiaro che il datagramma, ancora più piccolo del frame, non può eccedere l'MTU. Ma se ciò accadesse? *La soluzione di IPv4 era di frammentare il datagramma IP in datagrammi IP più piccoli, per poi riassemblarli (secondo le regole definite nell'intestazione) una volta giunti a destinazione.*

Come avviene di preciso l'operazione?

Consideriamo un datagramma di 4000 byte che deve essere inoltrato su un collegamento di MTU == 1500 byte. $4000 > 1500 \rightarrow$ bisogna spezzare il datagramma.

Nello spezzarlo dobbiamo tenere in considerazione la dimensione dell'MTU, 1500. L'ideale è spezzare il datagramma in un numero di datagrammi pari a quello per cui la maggioranza di essi copre tutta l'MTU: nel nostro caso 2 da 1500 e l'altro da 1000 ($1500 \times 2 + 1000 = 4000$).

Tuttavia dobbiamo tenere conto dell'intestazione per ognuno di essi, avremo uno scarto di 40 byte in più totale (perché il datagramma da 4000 byte aveva già 20 byte d'intestazione inclusi, ora da lui ne ho derivati altri 3 con 20 byte d'intestazione per ognuno -> 40 byte in più) → primo datagramma da 1500 byte con 20 byte intestazione e 1480 campo dati, secondo uguale e terzo da 1040 byte totali di cui 20 d'intestazione.

Come ricombiniamo i frammenti? Ce lo dice il **flag offset**, che ci dice sostanzialmente la posizione rispetto al datagramma originale. Offset = 0 è il primo frammento, Offset = payload del primo frammento è il secondo etc... Ma perché nell'esempio è scritto offset 185 al secondo frammento e non 1480? Perché **questo campo è espresso in multipli di 8** ($185 \times 8 = 1480$).

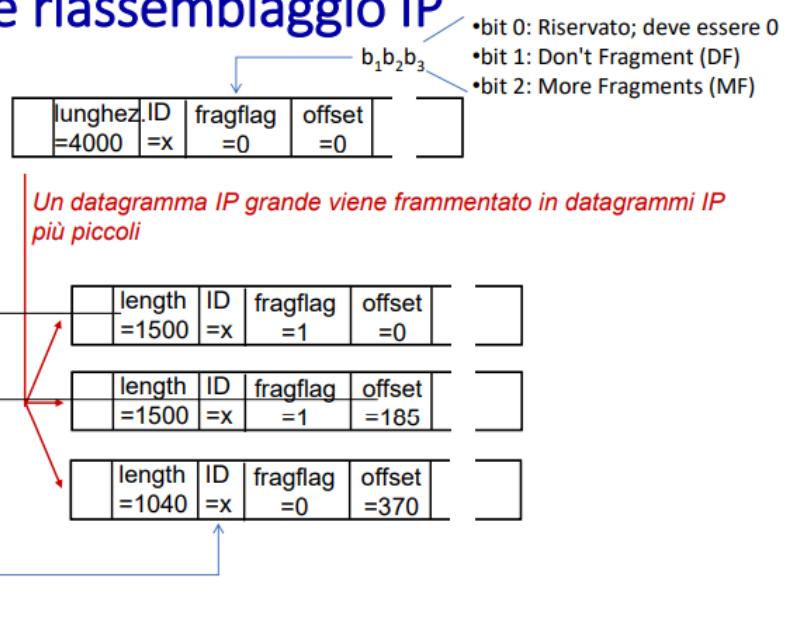
Ok che l'offset ci aiuta a riordinare i frammenti, ma come faccio a sapere che sono proprio i frammenti dello stesso datagramma? I frammenti dello stesso datagramma hanno stesso valore nel **campo ID** a 16 bit (definito secondo combinazioni mittente-destinatario-protocollo, se ho mittenti diversi posso chiaramente anche usare lo stesso ID).

L'ultimo campo dedicato alla frammentazione è il **fragflag**. Sono 3 bit: il primo (più a dx) è **Riservato** e deve essere sempre a 0. Il secondo è il bit **DF** (*Don't Fragment*) ed è a 0 se sto frammentando. Il terzo è il bit **MF** (*More Fragments*) ci dice che dopo questo pacchetto vengono altri frammenti. A che ci serve il terzo bit? L'offset serve a ordinare, l'ID a identificare frammenti dello stesso datagramma, questo bit mi serve a definire il fatto che ho finito di frammentare (bit MF a 0!).

Frammentazione e riassemblaggio IP

Esempio:

- Datagramma di 4000 byte
- MTU = 1500 byte



Tuttavia, poiché la frammentazione comporta perdita di tempo (cosa che non vogliamo nei router, che hanno nanosecondi a disposizione) e ha un'interazione infelice con il trasferimento affidabile (se mando un datagramma che viene frammentato e perdo un frammento mi tocca ritrasmettere tutto da capo) con IPv6 la frammentazione è stata rimossa.

Ciò che si fa quindi è cercare di utilizzare dimensioni per datagrammi ridotte (tendenzialmente quota 1500 byte) oppure utilizzare algoritmi di Path MTU Discovery, per cui l'host può scoprire l'MTU del collegamento verso il destinatario ed eventualmente decidere di inviare i dati o ridurne la dimensione.

Un semplice meccanismo per conoscere l'MTU nel collegamento di interesse è l'invio di pacchetti con bit DF don't fragment impostato a 1, se il router non può mandarlo perché eccede l'MTU allora lo scarta e invia al mittente il messaggio ICMP "Destination Unreachable: Fragmentation Required".

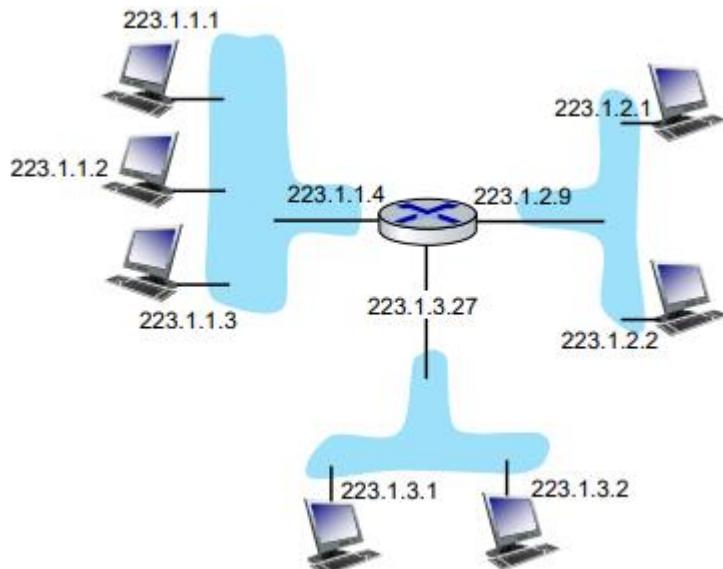
Il problema di questo metodo è che a volte questi messaggi di notifica ICMP possono essere bloccati per motivi di sicurezza, un mittente TCP rischia quindi di ritrasmettere più volte e inutilmente lo stesso pacchetto. Inoltre il percorso e quindi la MTU possono cambiare una volta che mi torna la notifica. Esistono chiaramente approcci più complessi ma robusti, come la manipolazione dei segmenti SYN in fase di instaurazione di connessione TCP, cambiando l'opzione relativa al MSS.

Indirizzamento IP

Un **Indirizzo IP** è un identificatore a 32 bit associato a ciascuna **interfaccia** di host e router. Quindi un indirizzo IP non identifica in sé l'host o il router, ma una delle sue interfacce.

Con **interfaccia** intendiamo la **connessione tra host/router con il collegamento fisico** (punto di snodo con cui l'host/router si connette al collegamento).

I router hanno tipicamente più interfacce (più collegamenti di uscita), mentre gli host ne hanno una o due (es... Ethernet cablata e/o Wireless).



notazione decimale puntata (*dotted decimal notation*)

$223.1.1.1 = \underbrace{11011111}_{223} \underbrace{00000001}_{1} \underbrace{00000001}_{1} \underbrace{00000001}_{1}$

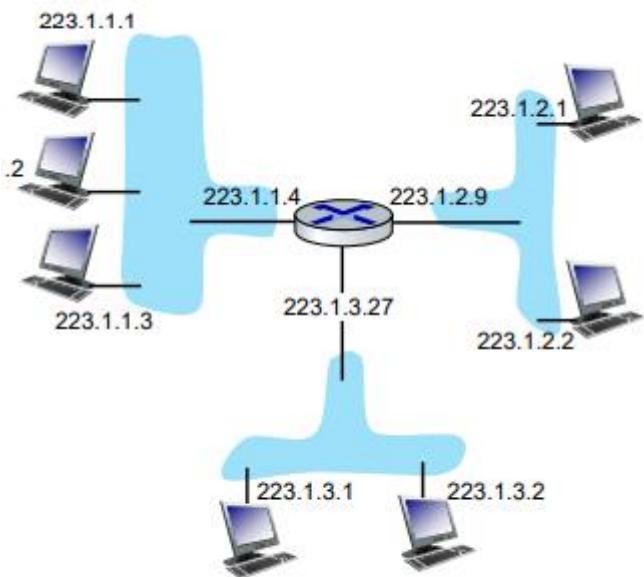
Ma come connetto tra loro le varie interfacce?

Nell'immagine si nota come tra i vari collegamenti, nello spazio host/router, è presente una sorta di sfumatura blu. Questa rappresenta un'infrastruttura a livello di collegamento che **connette reciprocamente le interfacce**: nel mezzo tra host e router ciò che connette fisicamente le interfacce e permette lo scambio di dati visto nei livelli superiori è *legato ai livelli di collegamento e al livello fisico, e dipende dalla tecnologia che si utilizza!*

Nel caso della rete cablata tipicamente lì in mezzo sono presenti degli **switches**, che *si occupano della funzione di commutazione dei pacchetti analoga a quello del router, ma che si spingono solo fino al livello di collegamento!*

È quindi possibile connettere determinati host anche solo tramite switch, senza l'ausilio di router. **L'insieme delle interfacce di dispositivi che possono**

raggiungersi fisicamente senza passare attraverso un router intermedio è detta sottorete (subnet).



rete composta da 3 sottoreti

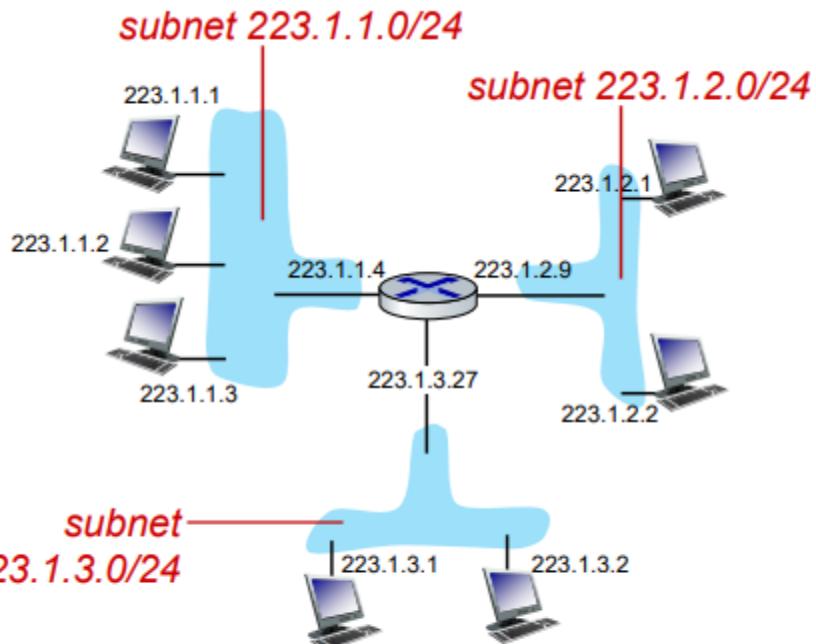
Gli indirizzi IP usati in ognuna delle sottoreti non sono arbitrari, hanno tutte lo stesso prefisso!

Gli indirizzi IP hanno quindi una struttura per cui sono divisi in due parti:

- **Parte della sottorete**: prefisso di bit di ordine superiore, lo hanno in comune gli host della stessa sottorete
- **Parte dell'host in quella sottorete**: i rimanenti bit di ordine inferiore

Sottoreti diverse devono avere prefissi diversi (salvo uso di indirizzi privati)

Chiariamo ancora una volta il concetto di sottorete. Per definire una sottorete dovrei sganciare tutte le interfacce di host e router, mi ritroverò così con un ammasso di collegamenti e switch (o antenne collegate tra loro nel caso wireless). **Ciascuno di quegli ammassi di “reti isolate” è una sottorete.**

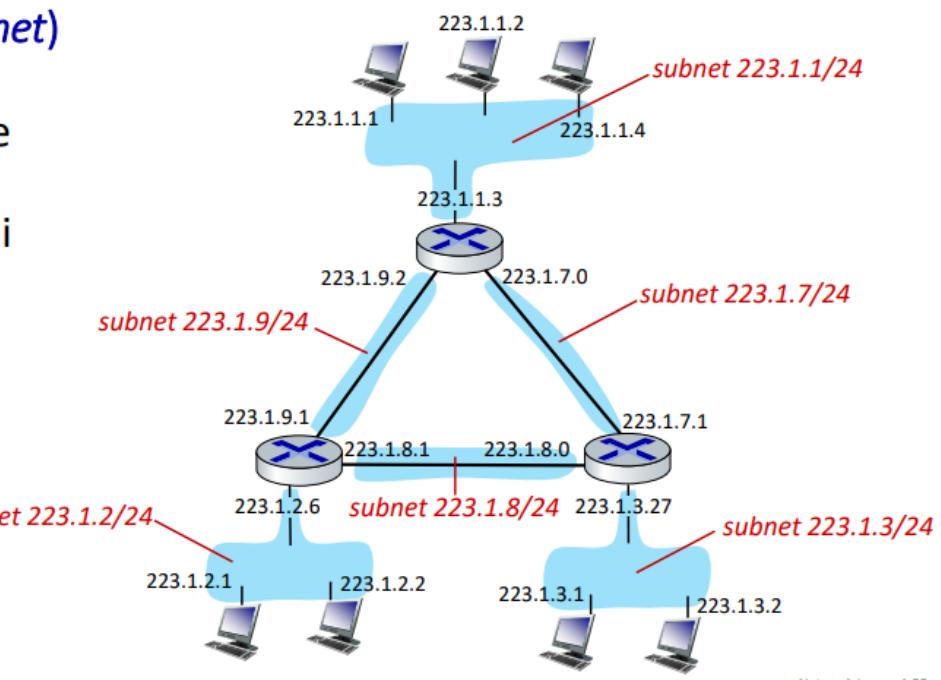


- maschera di sottorete (subnet mask): /24
 (24 bit di ordine superiore: parte di sottorete
 dell'indirizzo IP)

Si nota come anche lo “spazio” tra due router rappresenta una rete isolata e quindi, secondo quest’ottica, una sottorete.

Sottoreti (subnet)

- dove sono le sottoreti?
- cosa sono gli indirizzi di sottorete /24?



Abbiamo visto in breve che un indirizzo IP è formato da due parti: una legata alla sottorete, l’altra all’host. In generale una sottorete è identificata da un prefisso, che a sua volta può essere visto come indirizzo IP in cui mettiamo 0 a

tutti i bit relativi agli host! Dobbiamo specificare quanti bit fanno parte del prefisso. In questo senso si fa affidamento alla notazione **CIDR**.

La notazione a.b.c.d/x (es. 223.1.2/24) per rappresentare le subnet è detta notazione **CIDR** (**Classless InterDomain Routing**, “cider”) ci permette di rappresentare l’indirizzo della sottorete dandone una sorta di indirizzo in notazione decimale puntata o esadecimale (ad es. per IPv6) con valore x dopo / che indica il numero di bit dedicati al prefisso.

Per capire quali sono i bit effettivi meno significativi che saranno dedicati agli host di una certa sottorete è necessario chiaramente rappresentare la notazione decimale puntata in binario:

`223.1.2/24 == 11011111.00000001.00000010.00000000`

(primi 24 bit dedicati al prefisso, identificativi della subnet, ultimi 8 bit utilizzabili per localizzare gli host in quella sottorete, circa 2^8 possibilità eccetto due interfacce:

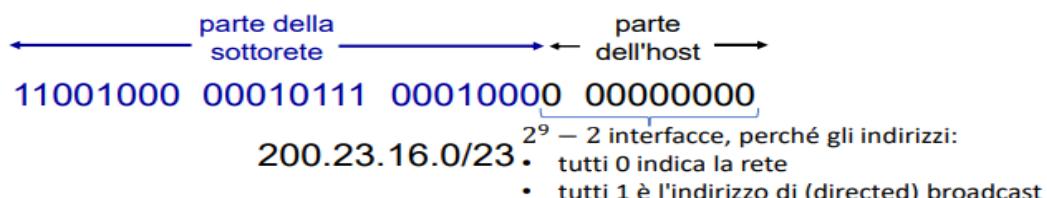
- 1- tutti gli host a 0 indica proprio la rete
 - 2- tutti gli host a 1 indica l'indirizzo di **(directed) broadcast**

CIDR è una notazione relativamente recente, prima si utilizzava la **maschera di sottorete (subnet mask)** in cui si mettevano ad 1 tutti i bit legati alla parte di sottorete. Nel caso sopra avremmo avuto 255.255.255.0

Un indirizzo IP di una subnet rappresentato in CIDR è comunque chiamato maschera di sottorete.

Dato un indirizzo IP specifico e una maschera, posso ottenere l'effettivo prefisso di rete semplicemente facendo l'AND tra i due.

Cosa importante di CIDR (**chiede all'esame**) è quella già detta prima, per capire quali sono i bit dedicati agli host della sottorete e quelli effettivi della sottorete devo prima convertire in binario la notazione decimale puntata!



Ciò è dovuto al fatto che se la lunghezza del prefisso non è multiplo di 8 allora c'è un byte che è mezzo di rete e mezzo di host, quindi tocca convertirlo in binario.

es. se mando un pacchetto dall'host 200.23.16.2 all'host 200.23.17.1 sapendo l'esistenza della subnet 200.23.16/23 devo passare per un router? No, stessa sottorete: **11001000.00010111.000100001.00000001** e **11001000.00010111.000100000.000000010** stesso prefisso!

Il **broadcast diretto**, per cui metto tutti 1 ai bit legati agli host della subnet, *consiste nel mandare i pacchetti a tutti gli host che stanno in quella sottorete. Poiché si tratta di un meccanismo di amplificazione e gli attacchi di negazione di servizio DoS si appoggiano su questo genere di cose*, tendenzialmente è una possibilità ristretta.

Un altro tipo di broadcast è il **broadcast limitato**, ha indirizzo tutti 1 per tutti i 32 bit (255.255.255.255) e *permette di inviare i pacchetti a tutti gli host solo della mia stessa sottorete*. Ciò è dovuto al fatto che è implementato come segue: quando voglio inviare i pacchetti all'indirizzo tutto ad 1 quando si passa al livello di collegamento quell'indirizzo IP rappresenta l'indirizzo di broadcast a livello di collegamento (che non si spinge fino ai router!). Quindi il livello di collegamento porta i pacchetti a tutti gli host della mia sottorete, ma quando incontra un router che potrebbe portarmi ad un'altra subnet lo “ignora”.

Quindi broadcast diretto permette di inviare in broadcast a un'altra sottorete, broadcast limitato essendo implementato come broadcast a livello di collegamento è limitato per definizione alla stessa sottorete del mittente.

Prima di Cider c'era un approccio **classful addressive**, ormai deprecato. Qui gli indirizzi erano suddivisi in classi A, B, C, D (multicast) ed E (reserved).

Concentrandoci sulle prime 3, queste erano classi con maschere di rete da 24, 16 e 8 bit. L'appartenenza a una certa classe per un host era determinata dal prefisso iniziale dell'indirizzo (prefissi che iniziavano per 0 classe A, con 10 classe B, con 110 classe C).

Il problema di questo approccio è che anzitutto, dovendo identificare le classi, devo sprecare dei bit iniziali (per A 1, per B 2, per C 3)...

Inoltre c'è un grande problema per la definizione di subnet: per la classe A ad esempio ho poche subnet e un numero enorme di host che possono appartenergli, per B più subnet ma meno host e ancora più subnet e meno host per C. Quindi la classe C è troppo grande per una casa, ma troppo piccola per un'azienda. Ma la B è a sua volta troppo grande per l'azienda! Indirizzi host "sprecati".

Non posso dosare gli indirizzi IP degli host.

Inoltre **non è supportato l'indirizzamento gerarchico**, che a breve vedremo e che permette di compattare le tabelle di instradamento.

Classe	Bit iniziali	parte della sottorete	Parte dell'host	Numero di reti	Numero di indirizzi per rete	Numero totale di indirizzi	Indirizzo iniziale	Indirizzo finale	Maschera di rete in dot-decimal notation	Notazione CIDR
Classe A	0	8	24	128 (2^7)	16,777,216 (2^{24})	2^{31}	0.0.0.0	127.255.255.255	255.0.0.0	/8
Classe B	10	16	16	16,384 (2^{14})	65,536 (2^{16})	2^{30}	128.0.0.0	191.255.255.255	255.255.0.0	/16
Classe C	110	24	8	2,097,152 (2^{21})	256 (2^8)	2^{29}	192.0.0.0	223.255.255.255	255.255.255.0	/24
Classe D (multicast)	1110	non definita	non definita	non definito	non definito	2^{28}	224.0.0.0	239.255.255.255	non definita	/4
Classe E (reserved)	1111	non definita	non definita	non definito	non definito	2^{28}	240.0.0.0	255.255.255.255	non definita	non definita

Chiaramente questo approccio è stato ormai abbandonato, in favore di CIDR (che ha i vantaggi di allocare in modo più efficiente blocchi di indirizzi e di aggregarli con conseguente riduzione delle tabelle di instradamento)

Ma come si ottiene un Indirizzo IP? O meglio, *come fa un host a ottenere un indirizzo IP all'interno della sua rete e come fa una rete ad ottenere il suo indirizzo IP?*

Vediamo di rispondere alla prima domanda.

Una rete può avere un'amministratore, il **sysadmin**, che decide l'assegnazione degli indirizzi IP e lo comunica agli host, installandolo “a mano”.

Un altro modo è attraverso *l'ausilio di un protocollo*, il **DHCP (Dynamic Host Configuration Protocol)** che consente a un host presente in una sottorete di ottenere in modo automatico (*plug and play*) l'indirizzo IP e altre informazioni attraverso *l'interazione con un server DHCP*.

Il server DHCP può anche rinnovare la concessione ad un indirizzo IP per un host che lo aveva ricevuto in precedenza e riciclare indirizzi IP che non sono più utilizzati da un host che ha abbandonato la subnet. Inoltre la dinamicità di questo protocollo favorisce il supporto per utenti mobili che si uniscono/abbandonano la rete (tuttavia quando mi sposto da una subnet all'altra ottengo un diverso indirizzo IP. In TCP una socket connessa è identificata dalla quadrupla indirizzo e numero di porta mittente/destinatario. **Pertanto nel momento in cui cambio subnet non posso mantenere una connessione TCP attiva).**

Ma come funziona nel concreto questo protocollo?

Immaginiamo un portatile che si unisce a una subnet con relativo DHCP server. Anzitutto è quanto succede nelle reti domestiche: si ha modem, router e punto di accesso inglobate tutte nel modem ADSL/Fibra che contiene tra l'altro anche il server DHCP.

1) La prima interazione è un messaggio di broadcast detto **DHCP discover** che consente al client di scoprire la presenza di un server DHCP (si manda quindi un messaggio all'indirizzo 255.255.255.255 per inviare in broadcast limitato, e uso la porta 67 del protocollo UDP).

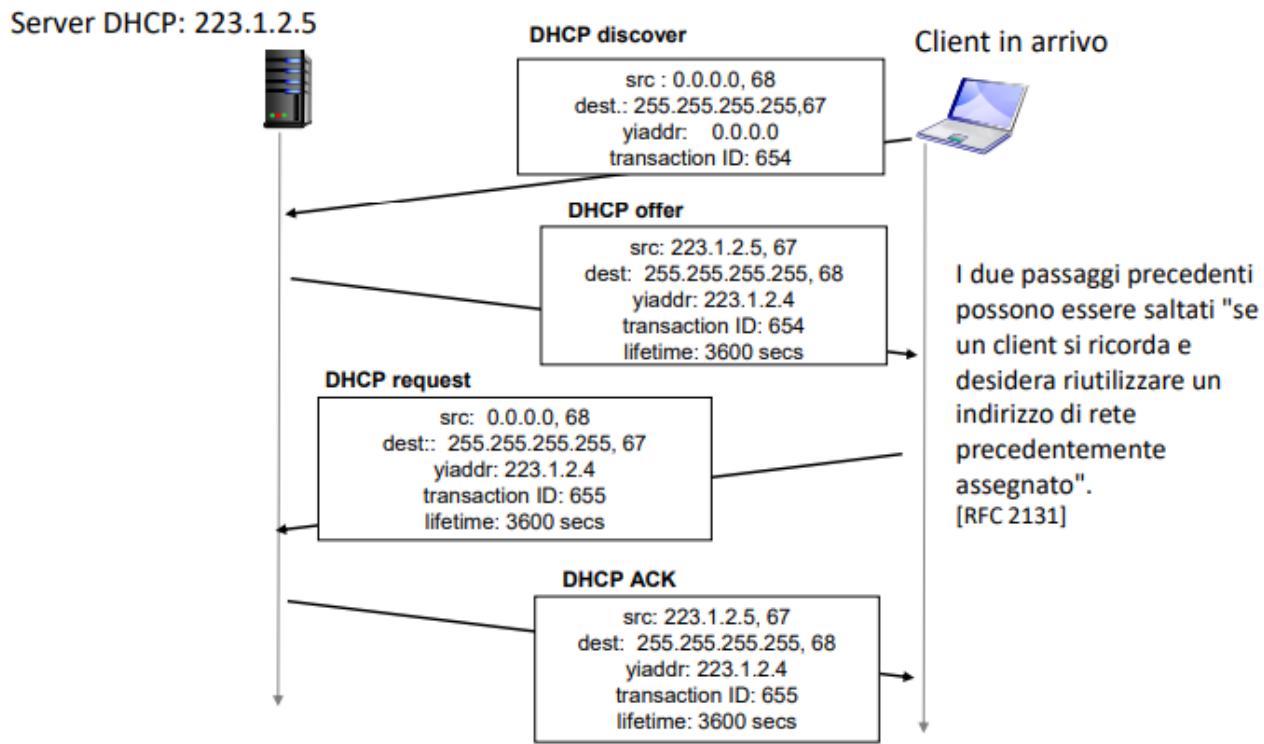
Si ha un ID di transazione per associare richieste e risposte (già visto nel DNS)

2) Il DHCP, ricevuto il messaggio discover, risponde con un messaggio di **DHCP offer** che ha la stessa struttura di prima (con ovviamente mittente e dest. invertiti) con campo yiaddress (youripaddress) contiene l'indirizzo IP proposto con associato un certo lifetime.

3) Essendoci la possibilità di aver ricevuto più offerte da DHCP diversi, il client risponde con un messaggio di richiesta **DHCP request** in cui chiede

quell'indirizzo IP specifico.

4) Se l'indirizzo è ancora disponibile il server accetta via **DHCP ack**.



Si nota come il destinatario è sempre broadcast, sia da parte del client che del server (il client deve cercare i server DHCP nella subnet, il server deve contattare il client che ancora non ha indirizzo IP specifico) e, altra cosa importante, **il mittente client è 0.0.0.0** (serve ad indicare un host nella rete senza ancora indirizzo IP. *Si nota anche come i primi due passaggi possono essere saltati se il client si “ricorda” e desidera riutilizzare un indirizzo di rete precedentemente assegnato, chiede direttamente al DHCP in questione.*

Come già accennato il server DHCP, oltre che l'indirizzo IP, può restituire altre informazioni come **l'indirizzo del router di first-hop per il client** (il primo per cui passa prima di lasciare la subnet), **nome e indirizzo IP del server DNS** e **la maschera di rete** (lunghezza del prefisso).

Perché è importante che un host conosca la maschera di rete della propria subnet? Perché se un datagramma (l'ho chiamato sempre messaggio prima ma alla fine il concetto è quello) è destinato a un host nella mia subnet non devo passare per il router (prendo il datagramma e lo incapsulo in un Frame con indirizzo quello dell'interfaccia del destinatario nella mia sottorete). **Se il destinatario è in un'altra sottorete, incapsulerò invece il datagramma in un Frame con indirizzo di destinazione quello del router!** Per prendere questa decisione (se mandare al router o direttamente al destinatario) devo chiaramente

sapere di che subnet faccio parte e di che subnet fa parte il destinatario (faccio l'and con il mio indirizzo e vedo se coincide con l'indirizzo della mia subnet).

Come fa invece una rete a ottenere un proprio indirizzo? Se ricordiamo abbiamo detto che gli host sono connessi ad una rete di accesso che li connette al primo router sul loro cammino, il “router di bordo”. La rete di accesso si connette ad Internet attraverso un collegamento di accesso (fibra ottica, wireless, ethernet etc...) che lo connette a un **ISP** (internet service provider). Quindi poiché il traffico in uscita della mia sottorete deve transitare per un ISP è proprio l'ISP a “rappresentare” le varie subnet e a gestirne gli indirizzi IP.

L'ISP possiede a tutti gli effetti un prefisso di indirizzi e da questo, estendendo il prefisso, può ricavare blocchi di indirizzi che può assegnare ad host diversi.

Supponiamo di voler estendere il seguente prefisso da 20 a 23 bit. Otteniamo un totale di 8 prefissi diversi (2^3)

D: Come fa la rete a ottenere la parte di sottorete dell'indirizzo IP?

R: ottiene l'assegnazione di una porzione dello spazio di indirizzi del suo provider ISP

Blocco dell'ISP 11001000 00010111 00010000 00000000 200.23.16.0/20

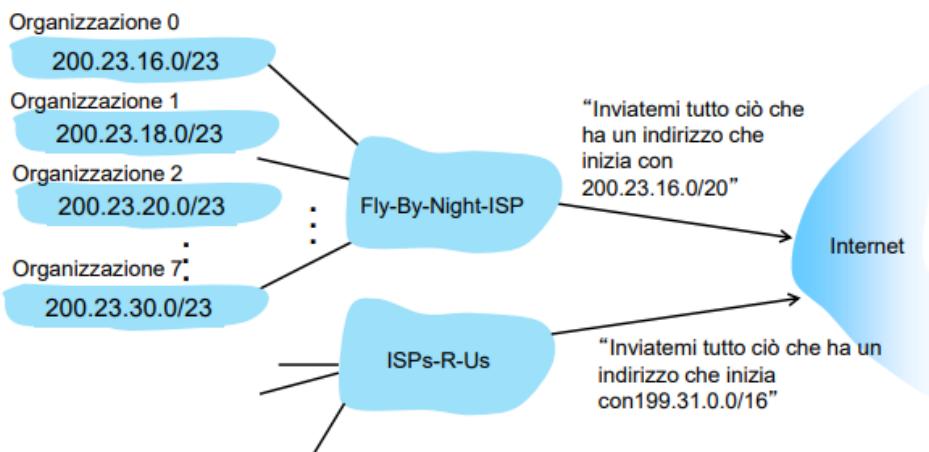
L'ISP può quindi allocare il suo spazio di indirizzi in 8 blocchi:

Organizzazione 0	<u>11001000 00010111 00010000</u> 00000000	200.23.16.0/23
Organizzazione 1	<u>11001000 00010111 00010010</u> 00000000	200.23.18.0/23
Organizzazione 2	<u>11001000 00010111 00010100</u> 00000000	200.23.20.0/23
...
Organizzazione 7	<u>11001000 00010111 00011110</u> 00000000	200.23.30.0/23
\		

Questa differenziazione in blocchi è importante per avere indirizzamento gerarchico, che garantisce di pubblicizzare in modo efficiente le informazioni di routing.

Un ISP comunica al mondo esterno che il traffico verso il prefisso di lunghezza 20 deve essere regolato verso di lui. A sua volta, quando arriva il pacchetto con prefisso 20, man mano aumenta il prefisso indirizzando i dati verso la sottorete appropriata (indirizzo di destinazione sempre più preciso, lo gestisce l'ISP direttamente mentre i router si sono limitati a prefissi più corti!).

L'indirizzamento gerarchico consente di pubblicizzare in modo efficiente le informazioni di routing:



Supponiamo ora che un'organizzazione voglia passare da un ISP all'altra e voglia mantenere il suo indirizzo IP. Allora il nuovo ISP può pubblicizzare il fatto di poter prendere il traffico per il prefisso di questa specifica organizzazione, che per i primi 20 bit coincide con quello dell'ISP precedente, ma che in realtà riguarda più specificatamente i primi 23 bit. Ecco che torna il ragionamento per "prefisso più lungo", si arriva a destinazione perché prima si guardano i primi 20 bit e si arriva al nuovo ISP perché il prefisso più lungo coincide proprio con quello dell'organizzazione che si è spostata.

Indirizzamento IP: ultime parole...

D: Come fa un ISP a ottenere un blocco di indirizzi?

R: ICANN: Internet Corporation for Assigned Names and Numbers

<http://www.icann.org/>

- Assegnazione degli indirizzi IP, attraverso **5 registri regionali (RR)** (che possono poi assegnare ai registri locali).
- Gestisce la zona radice del DNS, compresa la delega della gestione dei singoli TLD (.com, .edu , ...)

D: ci sono abbastanza indirizzi IP a 32 bit?

- L'ICANN ha assegnato l'ultima porzione di indirizzi IPv4 ai RR nel 2011.
- NAT (successivo) aiuta con l'esaurimento dello spazio degli indirizzi IPv4.
- IPv6 ha uno spazio di indirizzi a 128 bit

"Who the hell knew how much address space we needed?" Vint Cerf (riflettere sulla decisione di rendere l'indirizzo IPv4 lungo 32 bit)

Lez 16

Per capire se posso inviare un pacchetto senza passare per il router d'accesso (mia stessa sottorete) necessito di sapere la lunghezza del mio prefisso o la maschera della mia sottorete. Se il destinatario è nella mia sottorete incapsulo il datagramma in un Frame con destinazione proprio l'indirizzo del destinatario. In caso contrario, invio il pacchetto inserendo il datagramma in un Frame con indirizzo di destinazione del router.

Un altro indirizzo IP speciale (oltre a 0.0.0.0 che indica host senza IP e 255.255.255.255 che indica broadcast limitato) è 127.0.0.1 che è associato all'interfaccia di **loopback** (la possiamo immaginare come una scheda di rete aggiuntiva che è connessa a se stessa, il suo scopo è quello di permetterci di comunicare a noi stessi, tipicamente per parlare tra processi nella stessa macchina).

Con IPv4 si ha un ordine di grandezza di indirizzi disponibili pari a $2^{32} = c.a. 4kk$ di indirizzi. (non sono esattamente 2^{32} per via di indirizzi speciali e degli indirizzi privati (vedi dopo)).

Numero decisamente troppo piccolo, se si immagina anche solo di voler connettere un singolo host a persona nel mondo.

Si ricorda che il livello di rete trasmette pacchetti da mittente a destinatario lungo un percorso che consta diversi router, mentre il livello di collegamento

si occupa del trasferimento lungo un collegamento (da un nodo a quello adiacente)

NAT

Per ovviare al problema della “scarsità” di indirizzi IP, è stato introdotto il **NAT** (*network address translation*). Data una rete locale (es. rete domestica) ogni host che gli appartiene ha indirizzo IP differente.

Ma cosa succede quando voglio comunicare all'esterno, con la rete globale?

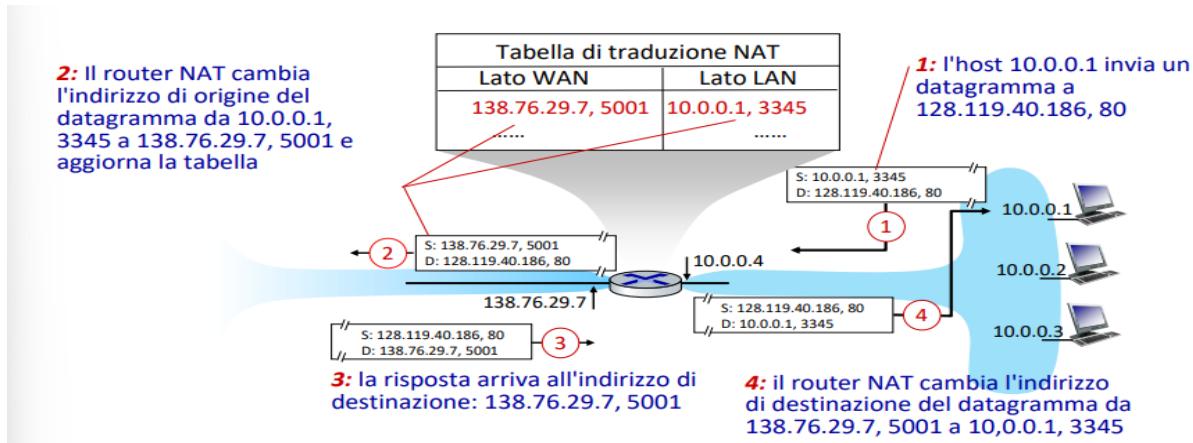
Con la tecnologia NAT, tutti i dispositivi appaiono all'esterno come avessero di fatto un unico indirizzo IP. Il vantaggio di ciò è che *bisogna quindi chiedere all'ISP un solo indirizzo*, invece che molteplici (uno per ogni host della mia sottorete).

Poiché inoltre l'indirizzo del mio host specifico non coincide esattamente con quello visto dall'esterno, *c'è un certo grado di flessibilità per cui posso cambiare il mio indirizzo IP senza dover riconfigurare tutti gli altri host della mia sottorete e senza doverlo comunicare all'esterno* (oggi è tipico avere IP dinamici cambiati di volta in volta dall'ISP, se ad es. spengo e riaccendo il router).

Un altro vantaggio di questo approccio è chiaramente *in termini di sicurezza*: di base un utente esterno non potrà connettersi ad un server presente nella nostra rete locale.

Utilizzare NAT ha senso solo se gli indirizzi nella nostra rete sono privati, cioè unici a livello locale. In particolare quindi, se nella mia sottorete ho un certo indirizzo IP, è possibile che in un'altra sottorete si stia utilizzando il mio stesso indirizzo IP (e qui il senso per cui si “risparmiano” indirizzi). Indirizzi privati di questo tipo sono **10/8** (prefisso con primo byte 10), **172.16/12** e **192.168/16**.

Come viene implementato NAT?



Nella nostra rete locale abbiamo un host, ad es. 10.0.0.1 che vuole inviare un datagramma all'host 138.76.29.7 incapsulando al suo interno un segmento a livello di trasporto la cui porta di destinazione è 80 (da cui deduciamo che stiamo accedendo a qualche server Web).

Il destinatario non appartiene chiaramente alla stessa rete locale del mittente, quindi il datagramma viene incapsulato in un frame in cui il destinatario in termini di MAC address (indirizzo di rete) è l'interfaccia del router di primo hop. Quando il router riceve questo datagramma, deve togliere come mittente l'indirizzo IP 10.0.0.1 perché di tipo privato e privo di senso al di fuori della propria sottorete (probabilmente esistono altri indirizzi IP identici in altre sottoreti). Il router sostituisce quindi l'indirizzo mittente con il proprio indirizzo NAT, ma deve anche ricordarsi di aver fatto questa sostituzione (se dovesse ricevere risposta) e quindi sostituisce anche il numero di porta mittente con un nuovo numero di porta la cui funzione principale è quella di chiave nella tabella di traduzione NAT. In questa tabella viene infatti salvata la corrispondenza tra numero di porta e indirizzo mittente modificati lato WAN (rete globale, wide area network) e numero di porta e indirizzo mittente originali lato LAN (rete locale). Il datagramma viene quindi inoltrato ed eventualmente arriva a destinazione. Se il destinatario risponde lo fa inviando dati all'indirizzo modificato, che corrisponde proprio a quello NAT del router.

Quando ciò accade, il router nel vedersi come destinatario del datagramma capisce di dover riconvertire indirizzo e quindi, attraverso la tabella di traduzione, risale all'indirizzo e al numero di porta originali inoltrando la risposta all'host locale.

Il NAT si trova tipicamente nei nostri router di accesso, che fungono da modem, router e quindi anche da NAT.

Il NAT è una tecnologia alquanto controversa: sappiamo infatti che i router *dovrebbero implementare fino al livello di rete*, senza interessarsi del segmento al livello di trasporto e limitandosi a trasferirlo. Con la tecnologia NAT invece il router **elabora anche il contenuto del datagramma** (quindi il segmento del livello di trasporto) e non si limita a leggerne il contenuto, ma arriva a cambiare indirizzo IP e numero di porta, violando il principio punto-punto secondo cui la rete dovrebbe limitarsi a trasferire i messaggi tra endpoint senza manipolarli. Inoltre il NAT era nato come modo per ovviare alla scarsità di indirizzi IP, ma ormai esiste da tempo una soluzione migliore: IPv6.

È vero inoltre che *NAT svolge una funzione di sicurezza*, impedendo che server in esecuzione nella rete locale siano indirizzabili dall'esterno, tuttavia questo può anche rappresentare un problema. Immaginiamo ad esempio se volessimo installare un server Web sulla nostra rete locale (e quindi volontariamente accessibile a tutti). Per ovviare a problemi di questo tipo sono state sviluppate tecnologie chiamate **NAT traversal** che permettono di bucare la barriera del NAT.

Difficile che NAT venga smesso di utilizzare: ampiamente utilizzato nelle reti domestiche e aziendali, nelle reti cellulari 4G/5G.

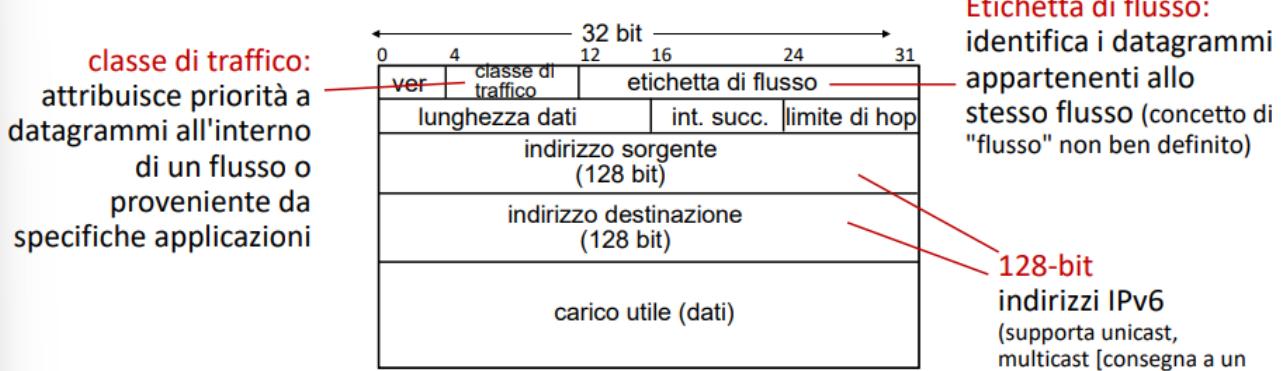
IPv6

La motivazione principale legata al suo sviluppo era che **lo spazio degli indirizzi in IPv4 a 32 bit era troppo scarso e sarebbe stato completamente allocato**. (sviluppo partito decenni prima dall'ultimo indirizzo IP allocato, visione lungimirante sapendo che non sarebbe stato facile passare da IPv4 a IPv6; ultimo blocco associato nel 2011 e di IPv6 si parlava già a fine anni '90). Ulteriori motivazioni per lo sviluppo di IPv6 sono legate alla **volontà di incrementare la velocità di elaborazione/inoltro dei router**: si è semplificata l'intestazione rendendola **di lunghezza fissa** (40 byte) ed eliminando elementi non propriamente vitali come **checksum e frammentazione**.

Ulteriore modifica apportata è stata quella di riconoscere il concetto di flusso

come “**first-class citizen**”: mentre in IPv4 il focus era esclusivamente sui datagrammi con IPv6 c’è la possibilità di riconoscere che una serie di datagrammi fanno parte di uno stesso flusso (tramite un numero di 20 bit) ed agire di conseguenza (con servizi differenziati per flussi o anche per classi di flussi, a seconda della loro “importanza”).

Formato del datagramma IPv6



Cosa manca (rispetto a IPv4):

- no checksum (per velocizzare l'elaborazione presso i router)
- no frammentazione/riassemblaggio (messaggio di errore ICMPv6 *Packet Too Big* con *MTU* del collegamento di uscita): in realtà, effettuato solo dal mittente e destinatario attraverso una *opzione*)
- no opzioni (disponibile come "intestazione successiva" del protocollo di livello superiore)

Network Layer: 4-86

-**Ver** sono i primi 4 bit e rappresentano il numero di versione. Purtroppo IPv6 non è retrocompatibile: se a un router abilitato a IPv4 arriva un datagramma IPv6 quello non lo riesce ad elaborare, al contrario la maggioranza dei router che elaborano IPv6 elaborano anche IPv4;

-**Classe di traffico** sono i successivi 8 bit ed è utilizzata per attribuire differente trattamento a datagrammi all'interno di uno specifico flusso o proveniente da specifiche applicazioni (quindi in una particolare classe di flusso), ciò permette il supporto a un servizio differenziato: es. dalla classe di traffico potrei capire che i datagrammi fanno parte di una classe di flusso in applicazione real-time (skype), quindi se il router implementa diffserv potrebbe trattarlo diversamente cercando di ridurre la latenza;

-**Indirizzi IP sorgente/destinatario:** da 32 bit di IPv4 a 128 bit (na fracca).

Oltre che per **unicast** e **multicast** (invia a più gente contemporaneamente) è implementata anche la funzionalità **anycast** (invia, dato un insieme di destinatari, a uno qualunque di questi) utile per il bilanciamento del carico;

-**Lunghezza dati:** quanto è grosso il payload

-**Limite di hop:** corrispettivo del vecchio **TimeToLive**, viene decrementato di 1 ad ogni hop e scartato dal router se arriva a 0. Necessario per evitare pacchetti

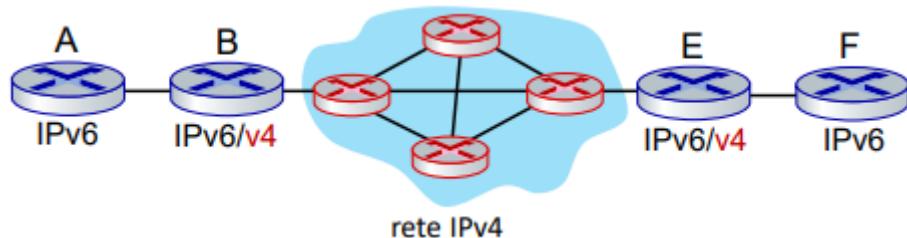
che girano all'infinito in rete (cicli) perché oltre che consumare traffico inutilmente potrebbero arrivare ad un destinatario che non c'entra nulla e quindi non rendere garantita l'affidabilità di protocolli come TCP.

Ciò che manca quindi rispetto a IPv4 è che **non vi è più checksum** (scoccava in IPv4 perché era limitata all'intestazione e quindi, poiché timetolive cambiava di hop in hop, bisognava ricalcolarla di volta in volta) o partizioni d'intestazione legate alla **frammentazione/riassemblaggio**: se il datagramma è troppo grande per l'MTU arriverà al mittente il messaggio di errore tramite **ICMPv6 "Packet Too Big"** con valore dell'MTU che lo ha provocato associato. Inoltre la **lunghezza dell'intestazione è fissa** per cui non vi sono più le opzioni, o almeno per come erano definite in IPv4. Il carico utile di un datagramma è il segmento di trasporto, dove il campo **Intestazione successiva** restituisce un numero che identifica un protocollo (TCP o UDP). In IPv6 le opzioni sono state quindi immaginate come un ulteriore livello di incapsulamento: **tra il datagramma e il segmento trovo in caso un'intestazione che rappresenta proprio l'opzione**.

Il passaggio da IPv4 a IPv6 non è cosa semplice, tanto che è **in corso da oltre 20 anni**. Non tutti i router possono infatti essere aggiornati contemporaneamente, come può quindi avvenire nel modo più efficiente possibile il passaggio?

Una delle principali soluzioni è quella del **tunneling**, la cui **idea di base** è **traportare un datagramma IPv6 come payload in un datagramma IPv4 tra router IPv4**. Questo stratagemma è ampiamente utilizzato in contesti come 4G e 5G.

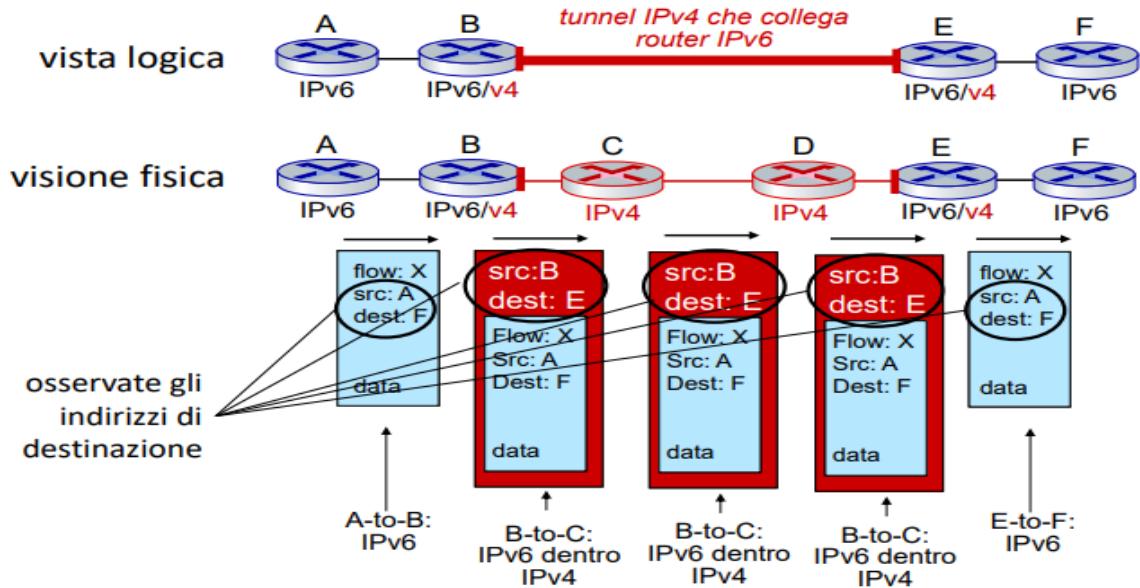
Immaginiamo una rete IPv4 che connette due router IPv6.



Chiaramente B ed E implementano anche IPv4 perché uno l'invia e l'altro lo riceve e lo spacchetta per riottenere IPv6. È come se la rete creasse un canale

tra B ed E sempre IPv6, solo che questo collegamento non è diretto, ma un “tunnel”.

B è consapevole di avere una rete IPv4 prima di poter giungere ad E (che implementa IPv6), quindi quando deve inviare ad E un pacchetto IPv6 lo impacchetta in IPv4 e lo inoltra nella rete. Quando E riceve un pacchetto IPv4 e si vede come destinatario lo spacchetta e ritira fuori IPv6, lo continua quindi ad inoltrare normalmente.



Ad oggi ancora non vi è totalità di utilizzo di IPv6, tutt’altro. Eppure è da 25 anni che si cerca di adottare questo nuovo standard. **Perché per IPv6 è così difficile, mentre cambiamenti a livello di applicazione hanno richiesto tempi molto più brevi?** (si pensi ai social media, il gaming, lo streaming etc...).

Perché, in generale, **cambiamenti a livello di rete (nel nucleo della rete) sono molto più difficili e lenti, mentre a livello applicativo più “semplice” e veloce.**

Inoltro Generalizzato

La funzione principale legata al piano di dati è **l’inoltro**, in particolare quello **basato su destinazione** (per cui inoltro in base all’indirizzo IP del destinatario). Si tratta di un caso specifico di un paradigma più generale chiamato **match plus action**, per cui trovo una corrispondenza e faccio qualcosa. Questo paradigma ci permette di generalizzare l’inoltro: si parla di **inoltro generalizzato**. *Il match non è più legato esclusivamente al prefisso più lungo di destinazione, ma può*

guardare più campi dell'intestazione (non solo quello di rete) e vi sono più possibili azioni corrispondenti (scarta/copia/modifica/logga il pacchetto).

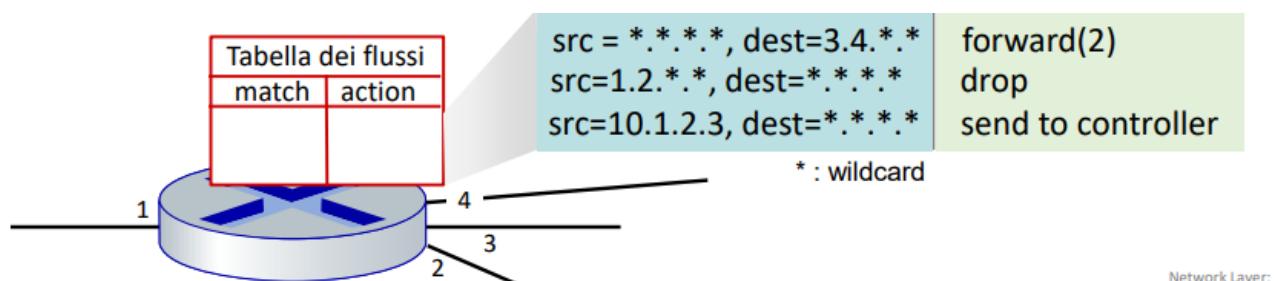
La tabella d'inoltro, in questo caso generale, è detta **tabella dei flussi** dove con flusso si intende combinazione di diversi campi d'intestazione (a livello di collegamento, rete o trasporto). Una tabella dei flussi definisce **regole**, dove una regola ha match e azioni corrispondenti.

match: pattern sui valori dei campi di intestazione;

actions: se trovo una corrispondenza scarto (**drop**), inoltro (**forward**), modifco l'intestazione (**modify**) oppure **invio al controllore**; (che può fare varie cose come aggiungere regole alla tabella dei flussi);

priorità: se vi sono pattern sovrapposti vi sarà modo di dare priorità all'uno o all'altro;

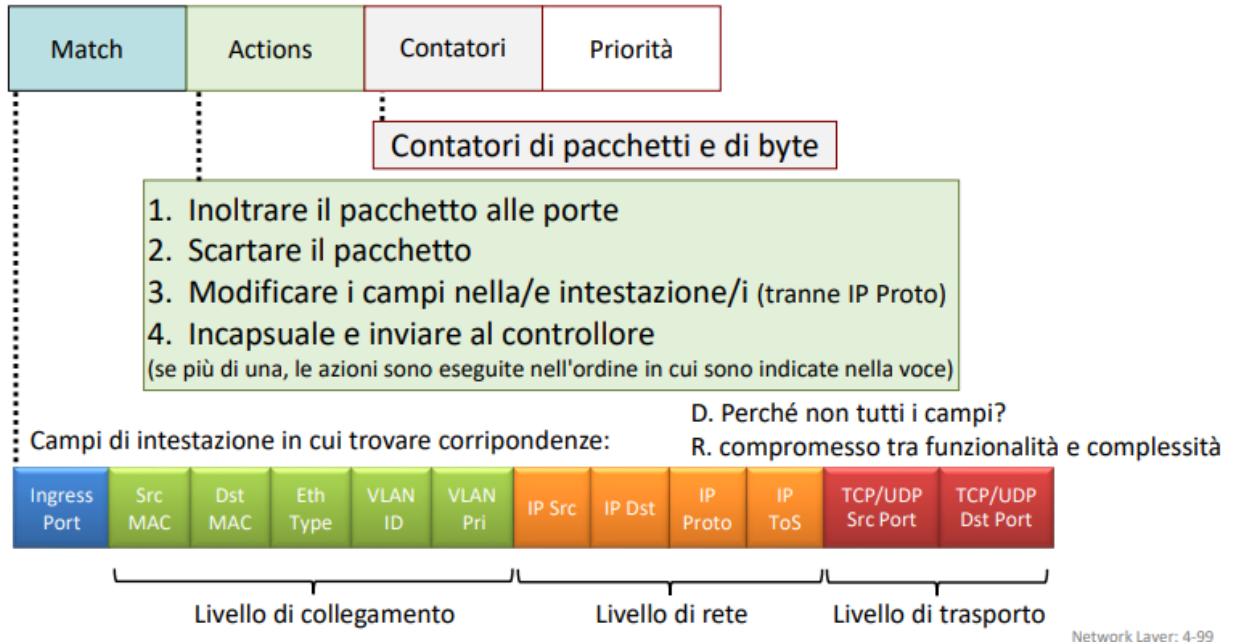
contatori: necessari ad esempio per delle statistiche; capire quanti pacchetti hanno fatto scattare una regola, quando è avvenuto l'ultimo aggiornamento etc...



Ecco dei semplici esempi: nel primo caso semplice inoltro, nel secondo caso se so che un pacchetto proviene da uno specifico indirizzo lo scarto indipendentemente dal destinatario (firewall molto semplice, basato sull'indirizzo IP del mittente, non offre grande protezione perché via spoofing IP il mittente può fingere di mandare messaggi da un'altra macchina).

Vediamo un esempio più concreto di una tabella di flusso basato sullo standard **OpenFlow**: si tratta di uno standard per inoltro generalizzato reso popolare dall'**SDN** (quello per cui il routing viene implementato da un controllore remoto).

OpenFlow fornisce infatti un protocollo che permette al controllore di interagire con i router per configurarli



Si vede come si tratta di inoltro generalizzato per via del matching che dipende da pattern d'intestazione differenti definiti anche a più livelli. Su questo OpenFlow possiamo prendere decisioni sulla base di 12 campi diversi. Vediamo qualche esempio di azioni corrispondenti a certi pattern.

Inoltro basato sulla destinazione:

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	VLAN Pri	IP Src	IP Dst	IP Prot	IP ToS	TCP s-port	TCP d-port	Action
*	*	*	*	*	*	*	51.6.0.8	*	*	*	*	port6

I datagrammi IP destinati all'indirizzo IP 51.6.0.8 devono essere inoltrati alla porta di uscita 6 del router.

Firewall:

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	VLAN Pri	IP Src	IP Dst	IP Prot	IP ToS	TCP s-port	TCP d-port	Action
*	*	*	*	*	*	*	*	*	*	*	22	drop

Bloccare (non inoltrare) tutti i datagrammi destinati alla porta TCP 22 (numero di porta ssh)

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	VLAN Pri	IP Src	IP Dst	IP Prot	IP ToS	TCP s-port	TCP d-port	Action
*	*	*	*	*	*	128.119.1.1	*	*	*	*	*	drop

Bloccare (non inoltrare) tutti i datagrammi inviati dall'host 128.119.1.1

Inoltro basato sulla destinazione a Livello 2:

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	VLAN Pri	IP Src	IP Dst	IP Prot	IP ToS	TCP s-port	TCP d-port	Action
-------------	---------	---------	----------	---------	----------	--------	--------	---------	--------	------------	------------	--------

* * 22:A7:23:
11:E1:02 * * * * * * * * * * * port3

frame di livello 2 con indirizzo MAC di destinazione 22:A7:23:11:E1:02 devono essere inoltrati alla porta di uscita 3

Load balancing

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	VLAN Pri	IP Src	IP Dst	IP Prot	IP ToS	TCP s-port	TCP d-port	Action
-------------	---------	---------	----------	---------	----------	--------	--------	---------	--------	------------	------------	--------

3 * * * * * * 10.1.*.* * * * * * port2
4 * * * * * * 10.1.*.* * * * * * port1

I pacchetti destinati a 10.1.*.* provenienti dalle porta 3 e 4 sono inviati rispettivamente sulle porta 2 e 1 (non possibile con l'inoltro basato sulla destinazione).

Nel primo esempio si vede come tramite OpenFlow posso implementare proprio la funzione di inoltro basato su destinazione visto la lezione scorsa!

Nel secondo esempio viene chiesto di bloccare tutto il traffico verso la porta 22 (porta well-known per il protocollo SSH secureshell che è un protocollo per l'accesso in remoto alle macchine), è bene quindi avere una regola di firewall che blocca il traffico destinato a quella porta.

Nel quarto esempio si vede come possa essere implementato l'inoltro generalizzato anche negli switch, poiché ho a disposizione pattern a livello di collegamento.

Nell'ultimo esempio si parla di Load Balancing (bilanciamento del carico) nel senso che non faccio transitare tutto il traffico lungo un unico collegamento ma lo inoltro su due porte d'uscita diverse (per sapere che può farlo il router guarda l'indirizzo di destinazione, capisce quindi che può prendere due strade diverse). (Normalmente in realtà quando si parla di load balancing si fa riferimento al fatto di avere più richieste in un servizio applicativo e per bilanciarle si applicano soluzioni tramite round-robin degli indirizzi che richiedono il servizio o distribuire a livello geografico i server che possono così offrire servizi in modo più efficiente, l'abbiamo visto coi CDN e DNS più in generale).

Quando si parla di inoltro generalizzato *non si parla specificatamente di router ma si parla di packet switch* (si parla infatti più in generale di commutatori di pacchetto che possono implementare varie funzioni)

Astrazione in OpenFlow

- **match+action:** astrae dispositivi differenti

Router

- **match:** prefisso IP di destinazione più lungo
- **action:** inoltro (*forward*) attraverso un collegamento

Firewall

- **match:** indirizzi IP e numeri di porta TCP/UDP
- **action:** consentire (*permit*) o negare (*deny*)

Switch

- **match:** indirizzo MAC di destinazione
- **action:** inoltra (*forward*) o inonda (*flood*)

NAT

- **match:** indirizzo IP e porta
- **action:** riscrive (*rewrite*) l'indirizzo e la porta

È importante ribadire come più in generale esista una **logica di rete globale**, *per cui pacchetti devono essere inviati da un host mittente ad uno destinatario*, di cui il controllore è “consapevole”. *I packet switch invece non ne sanno nulla, sta proprio al controller definire le regole affinché tutto funzioni correttamente.*

La logica di rete globale è “istanziata” nelle tabelle di flusso affinché i packet switch facciano il loro lavoro localmente, organizzando “inconsapevolmente” questi percorsi.

Abbiamo detto che con l'inoltro generalizzato non si parla più di router ma di packet switch più in generale, perché sono in grado di svolgere più funzioni a seconda delle regole che descriviamo. *Queste regole sono a tutti gli effetti una forma di programmazione*: programmare significa infatti istruire un hardware, dato un programma a fare qualcosa.

Questa programmazione è oggi sempre più generalizzata ed estesa, con linguaggi quali ad es. **P4**.

Middlebox

Con **middlebox** si intende *qualsiasi “box” intermedio che svolge funzioni diverse da quelle normali e standard di un router IP sul percorso dei dati tra host di origine e destinazione*.

Sappiamo che la funzione standard in Internet di un router basato sulla destinazione è l'**inoltro**. Con “percorso dei dati tra host” ci si vuole focalizzare sul fatto che le middlebox riguardano proprio il centro della rete. Le middlebox sono quindi scatolotti che nella rete svolgono funzioni diverse dai router basati su destinazione. Alcune di queste funzioni le abbiamo già viste: *Firewall, NAT, Load Balancer... ma anche Cache, Application-specific* (fornitori di servizi, CDN) etc..

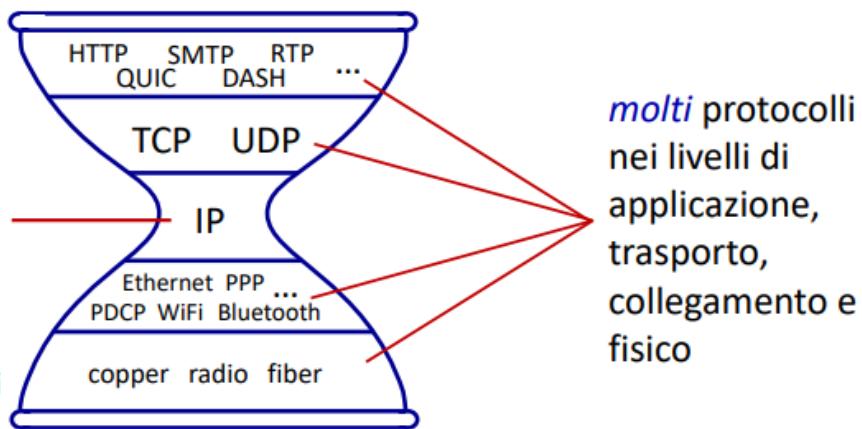
Inizialmente le middlebox *erano soluzioni hardware proprietarie* (chiuse, blackbox), da 10 anni a questa parte tuttavia è nato un approccio diverso simile a quello visto con Openflow. Proprio come quest’ultimo infatti si è passati ad hardware “**whitebox**” che implementano API aperte, abbandonando quindi le soluzioni hardware proprietarie. *Si è trasformato quindi ciò che era un semplice dispositivo hardware generico in rete in software*: ciò che infatti differenzia davvero un NAT da un Firewall ad esempio non risiede tanto nel comparto fisico per cui è possibile eseguirlo, quanto nel software (e quindi nella programmazione delle regole) che lo definiscono.

Con l’SDN abbiamo disaccoppiato il piano di controllo dal piano di dati, trasformando il piano di controllo in software in esecuzione su server remoti. Una logica simile di disaccoppiamento si ha nella **NFV (Network Functions Virtualization)**. L’idea alla base di questo approccio è di **eseguire le funzioni di rete (es. router, switch, firewall) utilizzando hardware “generico”** (COTS) tramite VM o container sfruttandone le risorse di calcolo e lo storage. *Sono usate svariate tecnologie per migliorare le prestazioni* e le funzioni di rete possono quindi essere anche eseguite in cloud. Vantaggio: molte più possibilità di programmazione (va a braccetto con SDN, che opera meglio con una struttura maggiormente programmabile).

Le clessidra IP

La "vita stretta" di Internet:

- *un protocollo a livello di rete: IP*
- *deve essere implementato da ognuno dei (miliardi di) dispositivi connessi a Internet*



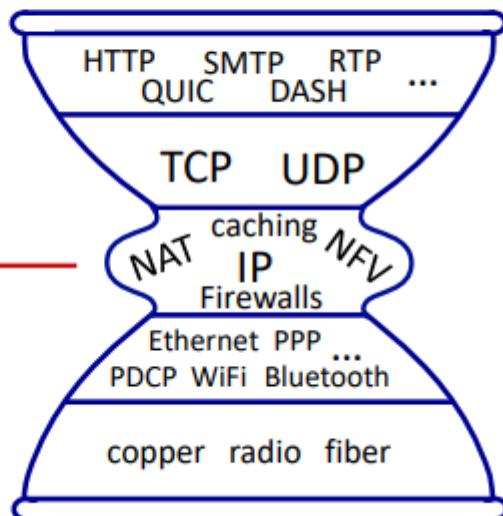
molti protocolli nei livelli di applicazione, trasporto, collegamento e fisico

IP offre servizio minimale ma omogeneo per chi sta sopra: trasferimento di datagrammi.

La clessidra IP, alla mezza età

Le "maniglie dell'amore" della mezza età su Internet?

- *middlebox, che operano all'interno della rete*



Dal semplice trasferimento fine a se stesso si è passati col tempo, via inoltro generalizzato e middlebox, a funzionalità sempre più varie anche a livello di rete.

Principi architetturali di Internet:

- **Connettività semplice** (trasferimento di datagrammi tra host);

- **Protocollo IP** (offre omogeneità a livello di rete e nasconde l'eterogeneità sottostante);
- **Intelligenza, complessità alla periferia** (edge) della rete.

Ma cosa si intende con l'ultimo principio?

Alcune funzionalità, come il controllo della congestione e il trasferimento affidabile, potrebbero teoricamente essere implementate nel nucleo della rete o nella periferia della rete. Tuttavia TCP ha deciso di implementare queste funzionalità a partire proprio dagli host (periferia), non è un caso. Si parla di **implementazione end-to-end** perché sono i terminali della comunicazione a implementare le funzionalità.

In realtà implementare e fare manutenzione di programmi in rete è molto più complesso di quanto non sia farlo alla periferia, si pensi alle rivoluzioni a livello applicativo degli ultimi anni e al tempo invece impiegato (e ancora in corso) nel passaggio da IPv4 a IPv6.

Parlando in modo più specifico di un'eventuale implementazione a livello di rete di affidabilità, ci rendiamo conto che vi potrebbero essere diversi problemi. Es. se un router crasha prima di trasmettere il pacchetto? Pacchetto perso.

Le funzionalità più complesse non riescono ad essere implementate completamente e correttamente senza la partecipazione dei livelli più alti che si limitano a una visione di periferia. **L'argomento end-to-end** suggerisce proprio di implementare questo tipo di funzionalità all'estremo, e non nel mezzo.

Ha senso talvolta “completare” queste funzionalità a livello di rete per migliorare le prestazioni.

Oggi l'intelligenza si trova non solo nella periferia, ma anche nel nucleo della rete (seppure ridotta rispetto alla periferia) (abbiamo visto middlebox, dispositivi di rete programmabili NFV etc..).

Inizialmente solo nel nucleo quando la comunicazione si limitava ai telefoni, pre-2005 in Internet solo in periferia.

Domanda: come sono calcolate le tabelle di inoltro (per l'inoltro basato sulla destinazione) o le tabelle dei flussi (per l'inoltro generalizzato)?

Risposta: dal piano di controllo



Rete telefonica del 20° secolo:

- intelligenza/calcolo negli switch di rete

Internet (pre-2005)

- intelligenza e calcolo nella periferia

Internet (post-2005)

- dispositivi di rete programmabili
- intelligenza, calcolo, infrastruttura massiccia a livello di applicazione alla periferia

Lez 18

Dopo aver parlato del piano di dati nel livello di rete (inoltro dei pacchetti nei tra i router, tabella di inoltro) passiamo al piano di controllo.

Si ricorda che l'approccio standard di Internet per l'inoltro è l'inoltro basato su destinazione, per cui si raggiunge una certa porta di uscita del router in base all'indirizzo di destinazione del pacchetto. Nell'inoltro generalizzato abbiamo visto invece che l'uscita viene determinata da molteplici fattori (anche livelli diversi da quello di rete, si parla di tabella dei flussi).

Ma come ricavare queste tabelle di inoltro? Viene calcolata dal **piano di controllo**, la cui funzione principale è quella di **instradamento** (determinare i percorsi che i pacchetti seguiranno dalla sorgente alla destinazione).

L'instradamento è esclusivamente una funzione legata al livello di rete che non può essere svolta da un router senza che abbia in qualche modo la conoscenza dell'intera rete. A seconda del tipo di approccio, si hanno diversi algoritmi di instradamento che permettono la scrittura delle tabelle di inoltro.

Il piano di controllo e dei dati sono quindi strettamente connessi tra loro (il piano di controllo controlla il piano dei dati, scrivendo le tabelle appropriate).

- **inoltro:** spostare i pacchetti dall'ingresso del router all'uscita del router appropriata
- **instradamento:** determinare il percorso seguito dai pacchetti dalla sorgente alla destinazione

piano dei dati

piano di controllo

Il router del resto sa di per sé solo qual è il next hop, non sa esattamente il percorso che il pacchetto dovrà seguire per l'intera rete. Inviando al next hop corretto secondo il calcolo delle tabelle dal piano di controllo (che è consci dell'intera rete), allora si simulerà l'instradamento.

Due approcci nel piano di controllo:

- **Tradizionale** (ogni router implementa il piano di controllo in maniera distribuita, ognuno di essi implementa algoritmi di instradamento)
- **SDN** (*software defined networking*) per cui *la funzione di controllo è logicamente centralizzata* (cioè è come se vi fosse un programma remoto che implementa il piano di controllo, ma in realtà non è un singolo programma ma un sistema distribuito)

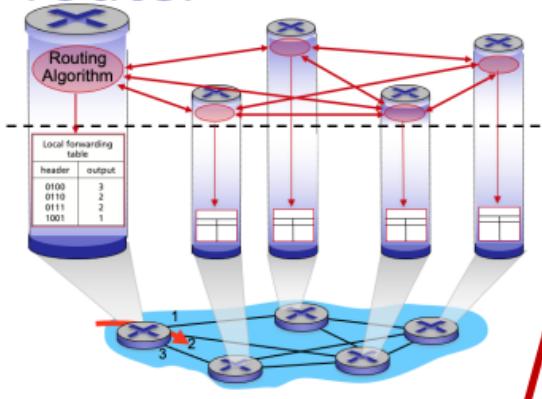
Perché SDN non è realmente centralizzato? Anzitutto per questioni di *fault tolerance e affidabilità*: con più repliche se muore un processo parte una replica che garantisce che il sistema continui a funzionare. Inoltre distribuendo il carico in più macchine *si può aumentare la capacità di sistema*.

Questo è però solo un dettaglio implementativo, dall'esterno il tutto appare centralizzato.

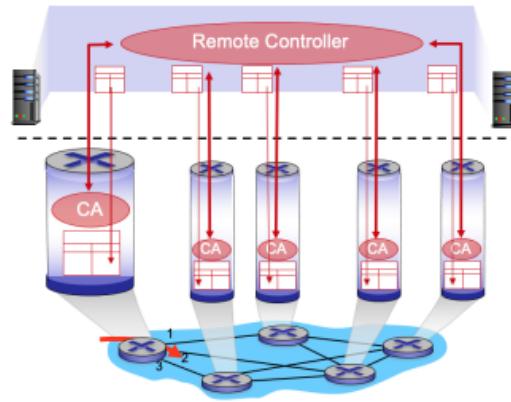
Nel caso dell'approccio tradizionale, abbiamo detto che ogni router implementa gli algoritmi di instradamento. Poiché, al fine di avere piena conoscenza della rete, è necessario che i router comunichino tra loro, è chiaro che esistono protocolli che determina la comunicazione tra loro (**protocolli di routing**).

Con SDN il piano di dati funziona esattamente come per l'approccio tradizionale: infatti ciò che gli interessa in fin dei conti è solo che le tabelle di inoltro siano calcolate nel modo corretto. Per quel che riguarda il calcolo effettivo delle tabelle, via SDN *un controllore remoto calcola le tabelle di inoltro e le installa nei vari router*.

Piano di controllo per router



Piano di controllo SDN



Prima di vedere nel dettaglio questi due approcci, vediamo gli algoritmi di instradamento.

Algoritmi di Instradamento

L’obiettivo principale di questi algoritmi è **calcolare percorsi “buoni” tra la sorgente e la destinazione**.

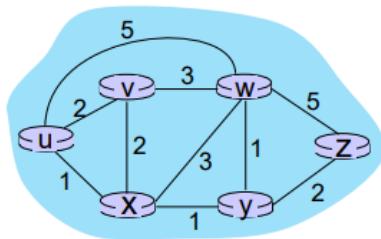
Un percorso/cammino rappresenta una sequenza di router che i pacchetti attraversano da origine a destinazione.

Con “*buon cammino*” si possono intendere diverse cose. In generale, si parla di *cammini di costo minimo in termini di tempo* (**spesso si considerano anche ulteriori vincoli**, ad esempio evitare di passare per router di specifici ISP etc...)

L’instradamento è uno dei primi 10 problemi nelle reti (come congestione e trasferimento affidabile).

Si implementano le reti come dei **grafi** (nodi N router, arco E percorso da router a router adiacenti, grafo pesato secondo specifici vincoli etc...):

Astrazione: grafo



grafo: $G = (N, E)$

$c_{a,b}$: costo del collegamento *diretto* che connette a e b

es., $c_{w,z} = 5$, $c_{u,z} = \infty$

z è adiacente o vicino
(neighbor)

costo definito dall'operatore di rete: potrebbe essere sempre 1, o proporzionale alla lunghezza fisica di un collegamento (ritardo di propagazione), o inversamente correlato alla larghezza di banda, o inversamente correlato alla congestione

Il costo di un *percorso* è uguale alla somma dei costi dei collegamenti attraversati.

N : insieme di router = { u, v, w, x, y, z }

E : insieme di collegamenti = { $(u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z)$ }

Per ora evitiamo di considerare nel grafo anche le sottoreti: infatti se devo comunicare con host nella mia sottorete comunichiamo direttamente senza passare per il router.

Costo infinito tra router non connessi tra loro direttamente.

Possiamo considerare gli algoritmi sulla base di *tre dimensioni*:

- **Grado di visibilità della rete** (quali informazioni utilizzo): in base a questo parametro posso classificare gli algoritmi in **globali** (*calcolo logicamente centralizzato o replicato su tutti i router basato sulla conoscenza completa della topologia e costo dei collegamenti in rete*, **algoritmi “link state”** es. Dijkstra) o **decentralizzati** (*calcolo iterativo basato sullo scambio di informazioni tra vicini, per cui inizialmente i router conoscono solo il costo dei collegamenti a loro vicini*, es. **algoritmi “distance vector”** es. Bellman-Ford).

- **Velocità di cambiamento dei percorsi**: negli algoritmi **statici** i percorsi vengono aggiornati lentamente (anche con intervento umano), mentre con quelli **dinamici** vengono aggiornati più rapidamente in modo periodico o in risposta del cambio di costo dei collegamenti e della topologia.

- **Sensibilità al carico**: gli algoritmi **sensibili al carico** sono quelli per cui il costo dei collegamenti riflette il livello corrente di congestione (es. correlato al ritardo di accodamento), quelli **insensibili al carico** no.

A causa di difficoltà sperimentate nell'uso di algoritmi sensibili al carico in passato, oggi si preferiscono algoritmi insensibili al carico.

(es. sensibile al carico, nota il collegamento congestionato e switcha per un altro. Gli altri router faranno lo stesso -> si ricrea congestione da un'altra parte, oscillazione tra più percorsi e quindi instabilità dei percorsi)

Dijkstra (link state)

È algoritmo link state, quindi globale (la topologia e il costo dei collegamenti nella rete sono noti a tutti i nodi). Queste informazioni sono ottenute tramite un algoritmo di link state broadcast. Tutti i nodi hanno le stesse informazioni.

Vengono calcolati i percorsi di costo minimo da un nodo sorgente a tutti gli altri nodi (fornendo quindi la tabella di inoltro a quel nodo).

Si tratta di un algoritmo iterativo: infatti dopo k iterazioni si conoscerà il cammino minimo verso k destinazioni.

- $c_{x,y}$: costo del collegamento diretto dal nodo x al nodo y ;
= ∞ se non sono vicini diretti
- $D(v)$: stima corrente del costo minimo del percorso dalla sorgente alla destinazione v
- $p(v)$: immediato predecessore di v lungo il percorso a costo minimo dall'origine a v
- N' : sottoinsieme di nodi contenente tutti (e solo) i nodi v per cui il percorso a costo minimo dall'origine a v è *definitivamente* noto

Instradamento "link-state": algoritmo di Dijkstra

```
1 Inizializzazione:
2    $N' = \{u\}$                                 /* calcola il percorso di minor costo da u a tutti gli altri nodi      */
3   per tutti i nodi v
4     se v è adiacente a u                      /* inizialmente conosce il costo del percorso diretto solo per i vicini diretti */
5       allora  $D(v) = c_{u,v}$                   /* ma potrebbe non essere di costo minimo                               */
6     altrimenti  $D(v) = \infty$ 
7
8 Ciclo
9   determina un w non in  $N'$  tale che  $D(w)$  sia minimo
10  aggiungi w a  $N'$ 
11  aggiorna  $D(v)$  per ciascun nodo v adiacente a w e non in  $N'$ :
12     $D(v) = \min(D(v), D(w) + c_{w,v})$ 
13  /* il nuovo costo verso v è il vecchio costo verso v oppure il costo del percorso minimo
14    noto verso w più il costo da w a v */
15 Ripeti il ciclo finché non si verifica che  $N' = N$ 
```

(vedi es. applicazione dell'algoritmo pg 16, potrebbe metterla all'esame).

Dijkstra costruisce anche l'albero dei cammini minimi tenendo traccia del predecessore, in caso di pareggi si può risolvere arbitrariamente.

Dijkstra costa $O(m+n\log(n))$ con m n. archi e n nodi. Poiché m alpiù n^2 , l'algoritmo costa $O(n^2)$. Tuttavia con implementazioni efficienti (es. Coda con Priorità) si scende a **$O(n\log n)$** .

Affinché Dijkstra sia a conoscenza dell'intera topologia della rete, viene utilizzato un **algoritmo di broadcasting** per cui ogni router trasmette in broadcast le proprie informazioni sullo stato dei collegamenti agli altri n router. Esistono algoritmi efficienti di questo tipo per cui sono necessari $O(n)$ attraversamenti per diffondere i messaggi. Poiché i messaggi riguardano n nodi si arriva a una complessità totale pari a **$O(n^2)$** per il broadcast.

Come detto prima, se il costo del collegamento dipende dal volume del traffico (algoritmo sensibile al carico), allora Dijkstra può causare oscillazione dei percorsi (tutti cambiano continuamente percorso, in quanto cambiano i costi in conseguenza a questi cambiamenti).

Bellman-Ford (distance vector)

Basato sulla equazione di **Bellman-Ford** (BF) (programmazione dinamica):

Equazione di Bellman-Ford

Sia $d_x(y)$: il costo del percorso di costo minimo da x a y .

Allora:

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

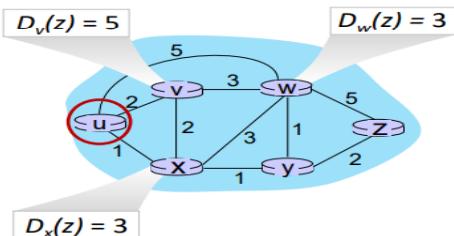
costo del cammino minimo da v a y

costo diretto del collegamento da x a v

\min calcolato su tutti i vicini v di x

Bellman-Ford: esempio

Si supponga che i nodi vicini di u , x, v, w , sappiano che per la destinazione z :



L'equazione di Bellman-Ford dice:

$$\begin{aligned} D_u(z) &= \min \{ c_{u,v} + D_v(z), \\ &\quad c_{u,w} + D_w(z), \\ &\quad c_{u,x} + D_x(z) \} \\ &= \min \{ 2 + 5, \\ &\quad 1 + 3, \\ &\quad 5 + 3 \} = 4 \end{aligned}$$

il nodo che raggiunge il minimo (x) è l'hop successivo sul percorso a costo minimo stimato verso la destinazione (z)

Network Layer: 5-35

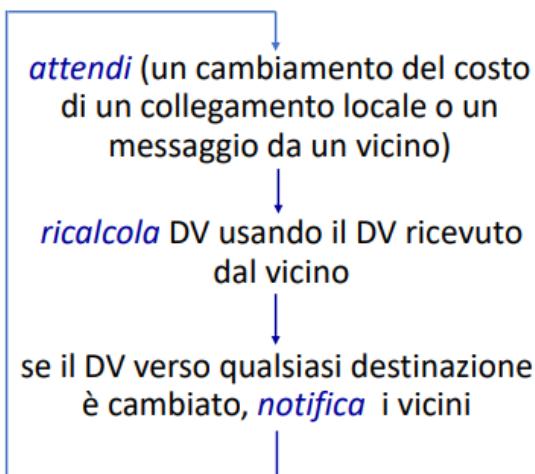
idea chiave:

- di tanto in tanto, ogni nodo invia ai vicini il proprio vettore delle distanze (stimate), *distance vector* in inglese
- quando x riceve un DV da un qualsiasi vicino, aggiorna la propria DV utilizzando l'equazione B-F:

$$D_x(y) \leftarrow \min_v \{ c_{x,v} + D_v(y) \} \text{ per ogni nodo } y \in N$$

- sotto certe condizioni minori e naturali, la stima $D_x(y)$ converge verso l'effettivo costo minimo $d_x(y)$

ciascun nodo:



iterativo, asincrono: ciascuna iterazione locale causata:

- cambiamento del costo del collegamento locale
- messaggio di aggiornamento del DV da un vicino

distribuito, auto-terminante:

ciascun nodo notifica i vicini *solo* quando la sua DV cambia

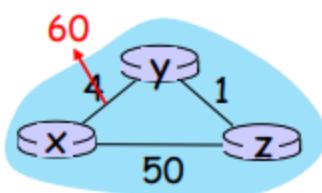
- i vicini notificano i loro vicini - *solo se necessario.*
- nessuna notifica ricevuta, nessuna azione intrapresa!

(vedi esempi Bellman-Ford da pg 37)

Ciò che bisogna ricordare **che differenzia gli algoritmi link-state da quelli distance-vector** è che nel primo (Dijkstra) vi è una parte iniziale di flooding in cui ogni router annuncia gli stati dei collegamenti a tutti gli altri router (quindi ogni router consapevole della situazione di rete!)

Nel caso di Bellman Ford un router non conosce invece la topologia della rete, ma solo il costo che i vicini hanno verso determinate destinazioni.

Ciò può comportare dei problemi: **conteggio all'infinito.**



Immaginiamo che il collegamento da y a x passi da pesare 4 a 60. Ma allora y, guardando le distanze di z da x, si accorge che quest'ultimo dista da x 5 (poiché z passa per y costo 1 e verso z costo 4, ancora non aggiornato). Ma allora y aggiorna la sua stima verso x a $1+5 = 6$. Poi z vede che la distanza verso x da y passa da 4 a 6, aggiorna la sua stima da 5 a $1+6 = 7$ e così via, fino a che z non

considera più conveniente il collegamento per y scegliendo quindi il collegamento diretto di costo 50.

Si può quindi chiaramente immaginare scenari dove invece, teoricamente, si diverge ad infinito (in realtà nel pratico chiaramente non è possibile, al limite si arriva ad un overflow).

Un modo per risolvere a monte questo problema è la tecnica **dell'inversione avvelenata** (poisoned reverse) per cui ogni nodo comunica al nodo adiacente attraverso cui "costruisce il suo cammino minimo" (suo next hop attuale) una distanza fasulla al nodo di destinazione pari a infinito.

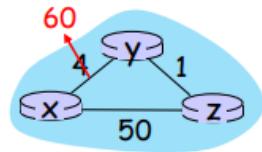
Se z instrada tramite y (cioè è il *next hop*) per giungere alla destinazione x , allora z avverrà y che la sua distanza verso x è infinita, ossia z comunicherà a y che $D_z(x) = +\infty$ (nonostante z sappia che $D_z(x) = 5$)

t_0 : y vede che il collegamento diretto con x ha un nuovo costo 60, ma continua a instradare attraverso il collegamento diretto perché per effetto dell'inversione avvelenata pensa che $D_z(x) = +\infty$; informa z del nuovo costo del percorso verso x

t_1 : z vede che il costo del percorso verso x tramite y è aumentato a 61, pertanto lo cambia in favore del collegamento diretto di costo 50; visto che ora non è più sul percorso verso x comunica a y che $D_z(x) = 50$

t_2 : y riceve l'aggiornamento da z e cambia l'instradamento verso x passando per z , pertanto $D_y(x) = 51$. Poiché z è ora sul percorso verso x , y avvelena il percorso inverso inviando a z $D_y(x) = +\infty$

L'inversione avvelenata risolve il problema del conteggio all'infinito solo nel caso di cicli che riguardano nodi adiacenti!

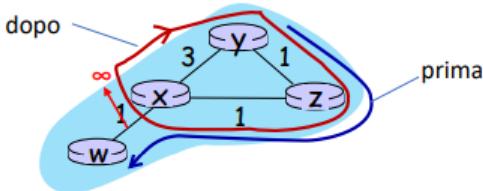


Network Layer:

Viene tuttavia risolto il problema solo nel caso di cicli che riguardano nodi adiacenti.

Come abbiamo anticipato, l'inversione avvelenata non risolve tutti i casi di conteggio all'infinito.

Inizialmente, y instrada verso w tramite z , il quale instrada tramite x , che sfrutta un collegamento diretto con w .



- x non può instradare tramite z (inversione avvelenata), ma decide di instradare tramite y , con costo $D_x(w) = 3 + D_y(w) = 6$, annuncia il nuovo costo
 - y riceve annuncio avvelenato da x , continua a instradare tramite z
 - z riceve annuncio da x , calcola $D_z(w) = 7$, annuncia il nuovo costo
 - y riceve annuncio da z , calcola $D_y(w) = 8$, annuncia il nuovo costo
 - x riceve annuncio da y , calcola $D_x(w) = 9$, annuncia il nuovo costo
 - z riceve annuncio da x , calcola $D_z(w) = 10$, annuncia il nuovo costo
- gli annunci si ripetono ciclicamente, preservando l'instradamento circolare venutosi a creare, il cui costo però aumenta progressivamente.

Quindi, per evitare problemi, l'idea è di trovare un modo efficace per rappresentare l'infinito affinché non si compiano troppe iterazioni prima di stabilizzare la rete.

Vettore delle distanze: rappresentazione dell'infinito

Nella pratica il conteggio all'infinito viene interrotto rappresentando l'infinito con un valore (finito!) scelto preventivamente.

RIP (RFC 1058) usa costi unitari per i collegamenti diretti e 16 per la rappresentazione dell'infinito. A tal riguardo si dice:

Ora si dovrebbe capire perché "infinito" è stato scelto per essere il più piccolo possibile. Se una rete diventa completamente inaccessibile, vogliamo che il conteggio all'infinito venga interrotto il prima possibile. L'infinito deve essere abbastanza grande da impedire che un percorso reale sia così grande. Ma non dovrebbe essere più grande del necessario. La scelta dell'infinito è quindi un compromesso tra le dimensioni della rete e la velocità di convergenza nel caso in cui si verifichi il conteggio all'infinito. I progettisti di RIP ritenevano che il protocollo non fosse pratico per le reti con un diametro superiore a 15.

Un altro svantaggio legato agli algoritmi distance vector, oltre al tempo impiegato a stabilizzarsi in casi come quello visto adesso, è la possibilità che un router comunichi informazioni errate. In particolare, se un router dice di avere un cammino di costo bassissimo verso qualsiasi destinazione, allora tutti ci passeranno -> **bucco nero**

Confronto tra gli instradamenti LS e DV

Complessità dei messaggi

LS: n router, $O(n^2)$ messaggi inviati

DV: scambio di messaggi tra router adiacenti; tempo necessario per la convergenza variabile

Velocità di convergenza

LS: algoritmo $O(n^2)$, che richiede $O(n^2)$ messaggi

- può avere oscillazioni

DV: può convergere molto lentamente

- può avere instradamenti ciclici
- problema del conteggio all'infinito

robustezza: che succede se un router si guasta o è compromesso?

LS:

- Un router può comunicare in broadcast un costo sbagliato per uno dei suoi collegamenti
- ciascun router calcola solo la *propria* tabella

DV:

- un router DV può comunicare *percorsi a costo minimo* errati ("ho un cammino di costo bassissimo verso qualsiasi destinazione"): *bucco nero*
- Il DV di ciascun router è usato dagli altri: gli errori si propagano attraverso la rete

Lez 19

Finora la nostra trattazione dell'instradamento è stata piuttosto idealizzata: abbiamo immaginato la rete come un grafo di router e collegamenti. In realtà non si possono considerare tutti i router allo stesso modo, allo stesso livello.

Internet è una rete di reti, non un'unica rete! Inoltre un approccio con un unico grafo che rappresenta tutto Internet non è chiaramente **scalabile** (miliardi di destinazioni), non saremmo in grado di *memorizzare tutte le destinazioni nelle tabelle di routing* e soprattutto *gli algoritmi distance vector impiegherebbero enorme tempo per convergere*. Inoltre, per questioni di **autonomia amministrativa** (internet rete di reti), *ogni amministratore di rete può voler controllare l'instradamento della propria rete in modo indipendente o voler nascondere dettagli sulla propria struttura interna*.

Di fatto, come vedremo in questa lezione, l'instradamento in Internet risolve entrambi i problemi di **scalabilità** e **autonomia amministrativa**.

Per affrontare questo problema il punto di partenza è raggruppare i router in regioni, note come **sistemi autonomi** (AS autonomous system), anche detti **domini**. Si tratta di "sottoreti" solitamente formate da router sotto la stessa amministrazione.

Nel caso di ISP molto grandi è possibile che non vi sia un unico AS, ma più AS tra loro interconnessi.

Si distinguono in questo senso gli AS in:

- **Intra-AS** (o intra-domain): si tratta dell'instradamento interno al sistema autonomo (rete).

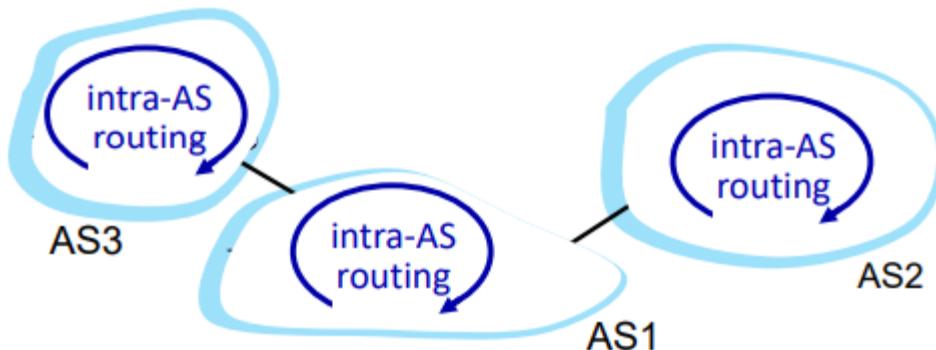
Tutti i router dell'AS devono eseguire lo stesso protocollo di instradamento interno al sistema autonomo. Se quindi abbiamo diversi AS, possono a tutti gli effetti essere eseguiti diversi protocolli.

È importante la presenza di un **router gateway** sul "bordo" dell'AS, che possa interconnettere il mio AS con altri AS.

- **Inter-AS** (o inter-domain): si tratta dell'instradamento *tra* sistemi autonomi. I gateway effettuano l'instradamento inter-AS (come pure l'instradamento intra-AS).

Ciò che comporta la suddivisione di Internet in vari AS e che **risolve il problema della scalabilità** è che *negli algoritmi link-state e distance-vector l'invio delle informazioni sullo stato della rete o sulle distanze è limitato all'AS in questione!*

Ciò comporta tabelle più piccole e maggiore velocità di convergenza. Come detto ogni sistema autonomo può usare il proprio algoritmo di instradamento, e affinché non sia isolato dagli altri AS è necessaria la presenza del **router gateway**. I gateway partecipano sia all'instradamento inter-AS che a quello intra-AS.



Ad un AS non interessa il protocollo di instradamento adottato da un altro AS (perché ciò riguarda solo il traffico di pacchetti in quell'AS), *ciò che gli importa sapere è che certi AS a lui "adiacenti" possono portare a determinate destinazioni.*

Avendo introdotto il concetto di AS, come riempie la tabella di instradamento il piano di controllo? *Distinguendo sulla base del fatto che la destinazione appartenga al mio stesso AS o ad altri AS.*

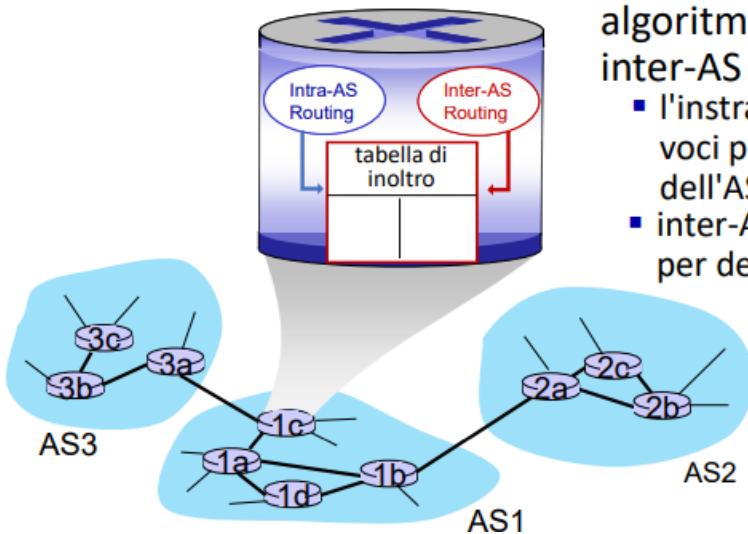


tabella di inoltro configurata dagli algoritmi di instradamento intra- e inter-AS

- l'instradamento intra-AS determina le voci per le destinazioni all'interno dell'AS
- inter-AS & intra-AS determinano le voci per destinazioni esterne

Se la destinazione sta fuori dal mio AS è necessario l'ausilio anche del routing inter-AS. Vediamo perché.

Immaginiamo che un router in AS1 riceva un datagramma destinato al di fuori di AS1. Allora quel pacchetto dovrebbe essere inoltrato ad un router gateway di AS1, ma quale? (1c o 1b?)

Dovrei sapere se la mia destinazione sta in AS3 o in AS2, meglio ancora quali destinazioni sono raggiungibili attraverso questi AS adiacenti a AS1.

L'instradamento inter-AS in AS1 deve saper fare proprio questo:

- 1) imparare quali destinazioni sono raggiungibili attraverso AS2 e quali attraverso AS3;
- 2) propagare queste informazioni di raggiungibilità a tutti i router in AS1.

Quindi esiste collaborazione tra inter-AS e intra-AS se devo inoltrare un pacchetto fuori dal mio AS perché *inter-AS mi dice quale gateway devo raggiungere, intra-AS “come” raggiungerlo* (quale algoritmo di instradamento interno utilizzo).

La volta scorsa abbiamo descritto i due principali algoritmi di instradamento, ma quando implementiamo parliamo a tutti gli effetti di **protocolli di instradamento**. Vediamo qualche protocollo di instradamento intra-AS:

- **RIP**: Routing Information Protocol. Algoritmo di tipo *distance-vector* (bellman ford) e standardizzato alla fine degli anni '80. Abbandonato verso l'inizio degli anni 2000 *a causa dei problemi che contraddistinguono distance vector (convergenza, conteggio all'infinito, buchi neri etc...)*.

- **EIGRP**: Enhanced Interior Gateway Routing Protocol. Anch'esso basato su DV (distance vector) raffinando la tecnica di RIP.
- **OSPF**: Open Shortest Path First. Si tratta del protocollo *attualmente più utilizzato*, si basa su *instradamento link-state* (Dijkstra). Vi è anche una controparte ISO (mentre OSPF è standard RFC) essenzialmente identico: il protocollo **IS-IS**.

Approccio link-state favorito rispetto a DV poiché più robusto (nonostante possa presentare *problemi di oscillazione* come visto se implementato anche in funzione della congestione e del traffico).

OSPF

Si tratta di un protocollo aperto di tipo **link-state**, che come visto si basa su una conoscenza “globale” della topologia della rete. È possibile realizzare questo tipo di instradamento tramite un processo di **flooding**: cuascun router invia in broadcast le informazioni circa lo stato dei collegamenti a tutti gli altri router nell’AS (direttamente via IP, senza usare TCP e UDP).

Si ricorda che è possibile utilizzare più metriche per determinare i vari costi di collegamento (tempo di percorrenza, ritardo, larghezza di banda...) e viene utilizzato l’algoritmo di Dijkstra per calcolare la tabella di inoltro.

Ogni router dispone quindi della topologia completa dell’AS.

OSPF supporta anche cammini di costo minimo multipli (possono essere registrati nelle tabelle più cammini con lo stesso costo minimo, tipicamente alternandoli. Attenzione quando si fa st’alternanza perché se alterno i vari pacchetti (1 da una parte, 2 3 dall’altra, 4 di nuovo dal primo router) allora se si usa TCP deve riordinare i pacchetti, se alterno troppe volte eccessivo sforzo di TCP nel riordinare. Di fatto quindi la strategia usata è di scegliere il router con cammino minimo in base alla destinazione, de dovo anda da una parte uso sempre lo stesso router così non ho problemi di riordinamento).

Altra roba di OSPF risiede nella sicurezza: *tutti i messaggi di OSPF sono autenticati* (per prevenire intrusioni dannose).

Visto così OSPF sembra semplicemente Dijkstra, ma in realtà fa qualcosa in più. Utilizza infatti l'approccio di instradamento gerarchico: l'AS è suddiviso in aree, una delle quali (denominata dorsale o area 0) ha lo scopo di interconnettere tutte le altre aree.

Si distinguono quindi **router locali** alle aree, router di dorsale, **router di confine d'area** (si trovano sia nell'area che nella dorsale) e **router di confine (boundary router)**, rappresenta sostanzialmente il *gateway router* di cui abbiamo parlato prima) che interconnette la mia AS con altri AS.

Come funziona l'instradamento gerarchico? Il flooding è limitato a un'area: in ogni area quindi è usato Dijkstra da ogni router locale per calcolare i cammini minimi ristretti a ogni area.

Ogni router di confine d'area annuncia alla dorsale quali destinazioni sono raggiungibili presso la propria area ed apprende dalla dorsale a quali router inviare i pacchetti a partire dalla propria area.

Quando un pacchetto è destinato fuori dall'area è inoltrato al router di confine d'area, che quindi sa a chi mandarli. Se tocca mandarli a un altro AS si passa chiaramente per il boundary router.

(Dettagli) Si possono avere più boundary router e più router di confine d'area, in tal caso si attuano approcci più complessi per decidere a partire ad es. da un router in un area a quale dei router di confine d'area inviare il pacchetto.

I boundary router potrebbero anche trovarsi al di fuori della dorsale, in un'area.

BGP

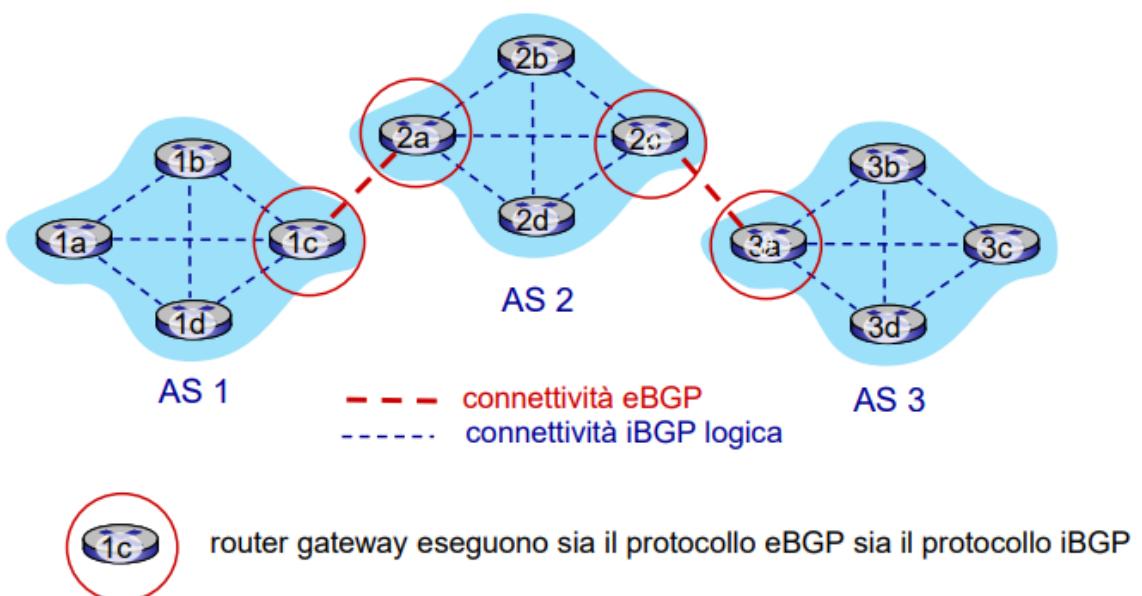
Vediamo ora un protocollo dedicato all'inter-AS, e quindi all'instradamento **tra** AS diversi.

Abbiamo visto che è possibile avere protocolli di instradamento diversi tra AS differenti, ma è chiaro che affinché comunichino tra loro i protocolli di instradamento inter-AS non possono variare (se tutti devono parlare con tutti, devono parlare la stessa lingua).

BGP (Border Gateway Protocol) è l'unico standard in questo senso. Questo protocollo permette ad ogni AS di:

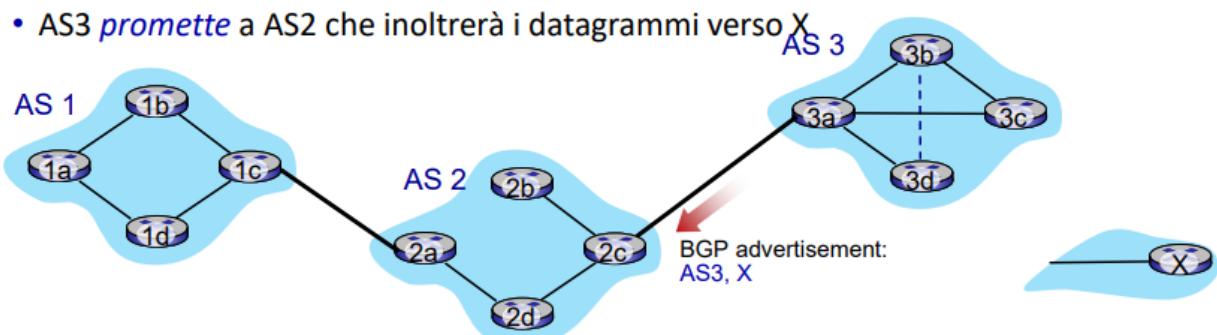
- ottenere da altri AS confinanti l'elenco di destinazioni che questi ultimi possono raggiungere (**eBGP**);
- Propagare le informazioni di raggiungibilità a tutti i router interni all'AS (**iBGP**);
- Determinare le rotte verso altre reti sulla base di informazioni di raggiungibilità e **politiche** (policy) (es. se non si vuole passare per uno stato di cui non mi fido, situazioni di guerra, economie di ISP etc...)
- Annunciare alle reti confinanti le informazioni sulla raggiungibilità delle destinazioni nel mio AS

I router gateway eseguono sia il protocollo eBGP che iBGP! In particolare tra due router gateway di ISP diversi viene eseguito il protocollo eBGP per ottenere vicendevolmente l'elenco delle destinazioni, mentre tra router interni all'AS a partire dal gateway iBGP per comunicare le informazioni ottenute dall'altro AS e quindi far sì che i router del mio AS populino la tabella di instradamento per garantire l'inoltro corretto di pacchetti destinati fuori dal mio AS.



Due coppie di router che comunicano via BGP sono detti **peers** e si scambiano messaggi BGP attraverso una *connessione TCP semi-permanente*. Definiamo questo scambio di messaggi, *atto ad annunciare percorsi verso diversi prefissi di rete di destinazione (quindi un altro AS)*, come **Sessione BGP**.

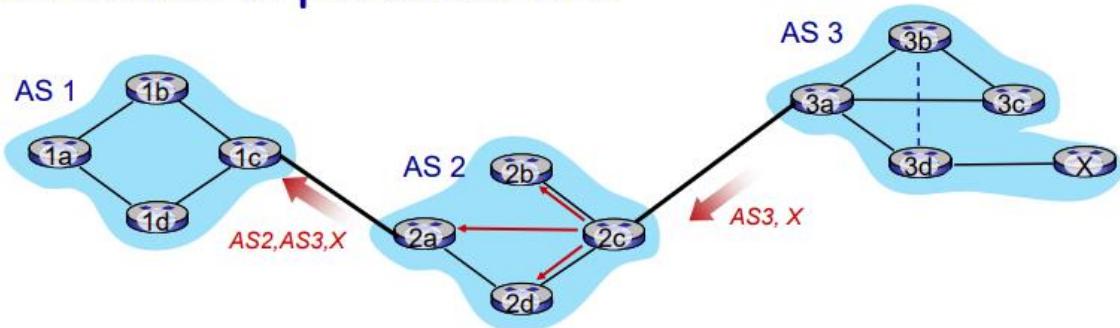
- Quando il gateway 3a di AS3 annuncia il percorso AS3,X al gateway 2c di AS2:
 - AS3 *promette* a AS2 che inoltrerà i datagrammi verso X



Esistono delle politiche sia quando decido di accettare una **Rotta** annunciata da un altro AS che quando sono io ad annunciarla.

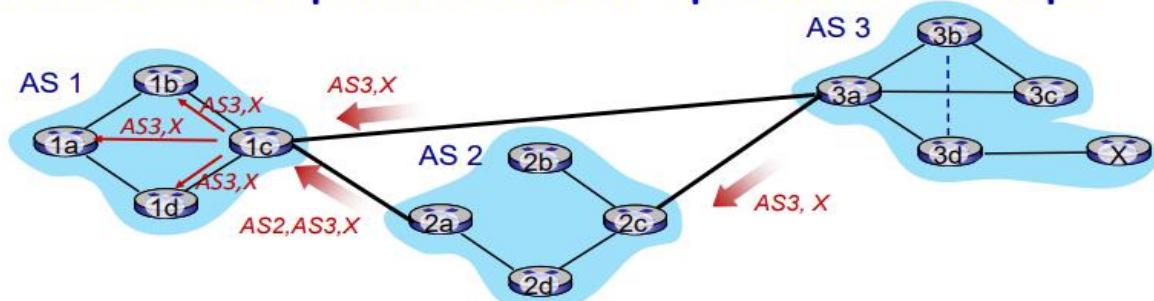
- I messaggi BGP sono scambiati tra peer su connessioni TCP
- Messaggi BGP [RFC 4371]:
 - **OPEN**: apre la connessione TCP al peer BGP remoto e autentica il peer BGP mittente
 - **UPDATE**: annuncia un nuovo percorso (o ritira il vecchio)
 - **KEEPALIVE**: mantiene in vita la connessione in assenza di UPDATE; inoltre ACK della richiesta OPEN
 - **NOTIFICATION**: segnala gli errori nel messaggio precedente; viene usato anche per chiudere la connessione
- Rotta (*route*) annunciata da BGP: prefisso + attributi
 - prefisso: la destinazione che viene annunciata
 - due attributi importanti:
 - **AS-PATH**: elenco degli AS attraverso i quali è passato l'annuncio del prefisso
 - **NEXT-HOP**: indirizzo IP dell'interfaccia del router che inizia l'AS-PATH
- **instradamento basato su politiche**:
 - Un gateway che riceve un annuncio di percorso usa una *import policy* per accettare/declinare il percorso (es., mai instradare attraverso AS Y).
 - Le politiche dell'AS determinano anche se *annunciare* un percorso a altri AS vicini

Annuncio di percorso BGP



- il router 2c in AS2 riceve l'annuncio del percorso **AS3,X** (attraverso eBGP) dal router 3a in AS3
- sulla base delle politiche di AS2, il router 2c in AS2 accetta il percorso AS3,X, e lo propaga (attraverso iBGP) a tutti i router in AS2
- sulla base delle politiche di AS2, il router 2a in AS2 annuncia (attraverso eBGP) il percorso **AS2, AS3, X** al router 1c in AS1

Annuncio di percorso BGP: percorsi multipli



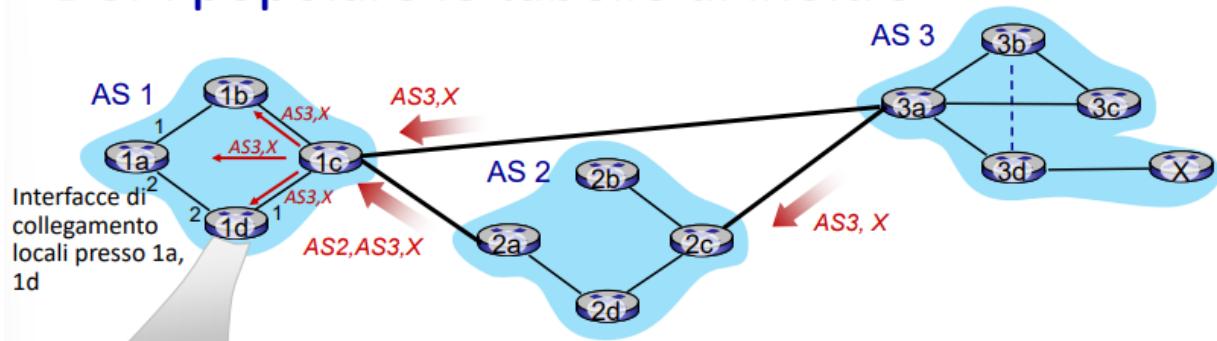
un router gateway potrebbe venire a conoscenza di percorsi **moltipli** verso una certa destinazione:

- il router gateway 1c di AS1 apprende il percorso **AS2, AS3, X** da 2a
- il router gateway 1c di AS1 apprende il percorso **AS3, X** a 3a
- sulla base di **politiche**, il router gateway 1c in AS1 sceglie il percorso **AS3, X** e annuncia il percorso dentro l'AS attraverso iBGP

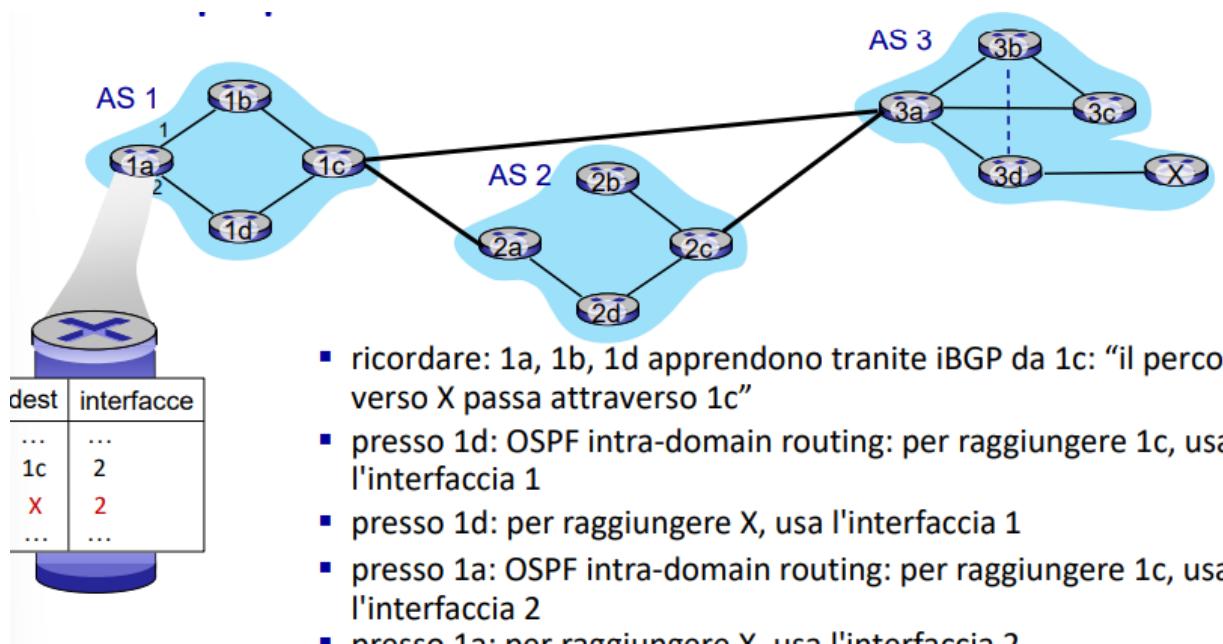
Quindi se ci sono percorsi multipli le possibilità, in ordine decrescente di priorità, sono le seguenti:

- Uso un **attributo di preferenza locale** (*decisioni politiche*, scelgo “personalmente” quale percorso prendere);
- Considero l'**AS-path più breve**;
- Considero il **percorso con next-router più vicino** (*instradamento a “patata bollente”*, voglio togliermi il pacchetto il prima possibile).

BGP: popolare le tabelle di inoltro



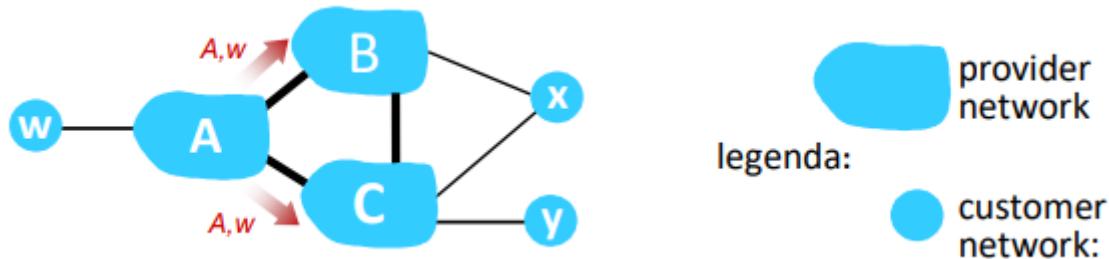
- ricordare: 1a, 1b, 1d apprendono tramite iBGP da 1c: "il percorso verso X passa attraverso 1c"
- presso 1d: OSPF intra-domain routing: per raggiungere 1c, usa l'interfaccia 1
- presso 1d: per raggiungere X, usa l'interfaccia 1



Approfondiamo ora come sono implementate le questioni “politiche”.

Quando facciamo instradamento **intra-AS** siamo interessati principalmente alle prestazioni e quindi alla **minimizzazione di costi**.

Quando facciamo instradamento **inter-AS** è molto più importante occuparsi di **questioni di natura politica**.



In generale un ISP vuole trasferire solo i pacchetti che sono originati dai propri clienti o destinati ai propri clienti, non vuole occuparsi dei pacchetti di altro traffico (traffico di transito) perché gli ISP guadagnano solo dalle robe dei propri clienti (far trafficare roba costa, se faccio trafficare roba che se non ci fosse non mi cambierebbe niente non conviene).

Nell'esempio sopra, a B di instradare traffico tra A e C che non ha a che fare con x non gli dà alcun vantaggio.

Per implementare queste politiche si possono “manipolare” gli annunci:

- A annuncia il percorso Aw a B e a C
- B *scegli di non annunciare* BAw a C!
 - B non riceve alcuna “entrata” per l'instradamento CBAw, visto che né C, A, w sono clienti di B
 - C *non* viene a conoscenza del percorso CBAw
- C instraderà CAw (non usando B) per raggiungere w

Altra cosa interessante: x è connessa ai due provider B e C, ma non è interessata al traffico tra i due:

- A,B,C sono **provider network**
- x,w,y sono **customer** (delle provider networks)
- x è **dual-homed**: connessa a due reti
- *politica da applicare*: x non vuole instradare da B a C attraverso x
 - .. quindi x non annuncerà a B un percorso verso C

Come già detto prima:

- Il router può conoscere più di un percorso verso l'AS di destinazione, seleziona il percorso in base a:
 1. valore dell'attributo di **preferenza locale**: decisione politica
 2. AS-PATH più breve
 3. router NEXT-HOP più vicino: instradamento a patata bollente
 4. identificatori BGP

Si può utilizzare CIDR per riassumere le destinazioni in prefissi più piccoli in modo da ridurre la dimensione delle tabelle di instradamento.

Perché diversi instradamenti Intra- e Inter-AS?

politiche:

- inter-AS: l'amministratore vuole avere il controllo sul modo in cui viene instradato il suo traffico, su chi passa attraverso la sua rete
- intra-AS: singolo amministratore, quindi le politiche sono meno problematiche

scalabilità:

- il routing gerarchico consente di ridurre le dimensioni delle tabelle e il traffico di aggiornamento.

prestazioni:

- intra-AS: può concentrarsi sulle prestazioni
- inter-AS: le politiche sono dominanti rispetto alle prestazioni

Piano di Controllo SDN

Quelli visti finora sono i così detti **router monolitici**, che contengono assieme l'hardware di commutazione (switching) e implementano il protocollo di instradamento in maniera proprietaria. Abbiamo visto come anche all'interno delle reti siano presenti, oltre ai router, delle “**middlebox**” che si occupano anche di altre funzioni a livello di rete oltre all'inoltro (come firewall, load balancers, NAT etc...).

Oggi si sta emigrando ad un approccio diverso, l'**SDN**.

Come già accennato, **con l'SDN si passa da un piano di controllo distribuito ad uno centralizzato tramite un controllore remoto, che riceve le informazioni di stato dai vari router e installa le tabelle di instradamento.**

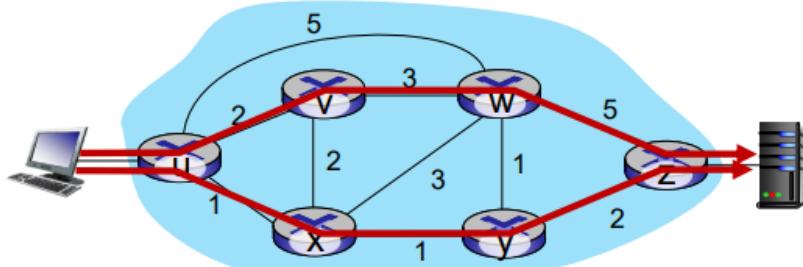
Perché dovremmo favorire l'implementazione SDN che è logicamente centralizzata? Perché è **più semplice** ed *evita errori di configurazione, ho controllo più flessibile sui flussi di traffico**.

Abbiamo inoltre visto come l'SDN si combini ad un approccio non più limitato solo all'inoltro tradizionale basato su destinazione ma usa un'API più sofisticata per programmare le azioni dei router.

Un esempio di queste API è **OpenFlow**, che permette di programmare non solo l'inoltro su destinazione ma anche **l'inoltro generalizzato**. Questo approccio centralizzato è più semplice perché calcola le tabelle centralmente e poi le distribuisce (mentre con programmazione distribuita devo fare il calcolo per ogni singolo router). Inoltre quest'implementazione centralizzata è aperta e non proprietaria, quindi promuove l'innovazione.

*negli algoritmi visti finora i percorsi sono basati sui costi dei collegamenti, quindi se voglio forzare un percorso l'unico modo per farlo è abbassare artificialmente i costi per quel percorso. L'operazione tuttavia è difficile da controllare.

Ingegneria del traffico: difficile con il routing tradizionale

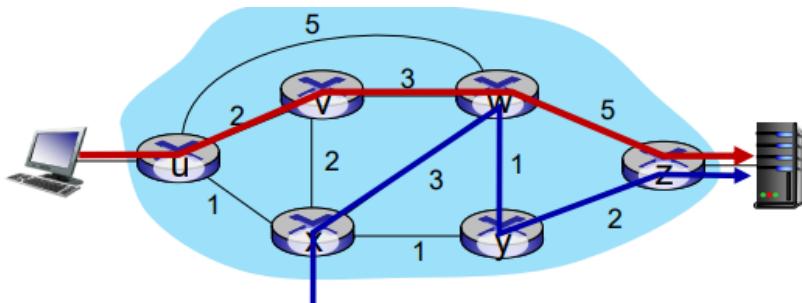


D: cosa succede se l'operatore di rete vuole che il traffico da u a z fluisca lungo $uvwxyz$, anziché $uxyz$?

R: è necessario ridefinire i pesi dei collegamenti in modo che l'algoritmo di instradamento del traffico calcoli le rotte di conseguenza (o necessitiamo di un nuovo algoritmo di instradamento)!

I pesi dei collegamenti sono le solo “manopole” di controllo: non c’è molto controllo!

Network



D: e se w volesse instradare il traffico blu e rosso in modo diverso da w a z ?

R: non può farlo (con l'inoltro basato sulla destinazione e l'instradamento LS e DV).

Abbiamo appreso che l'inoltro generalizzato e l'SDN possono essere usati per raggiungere *qualsiasi* instradamento si desideri

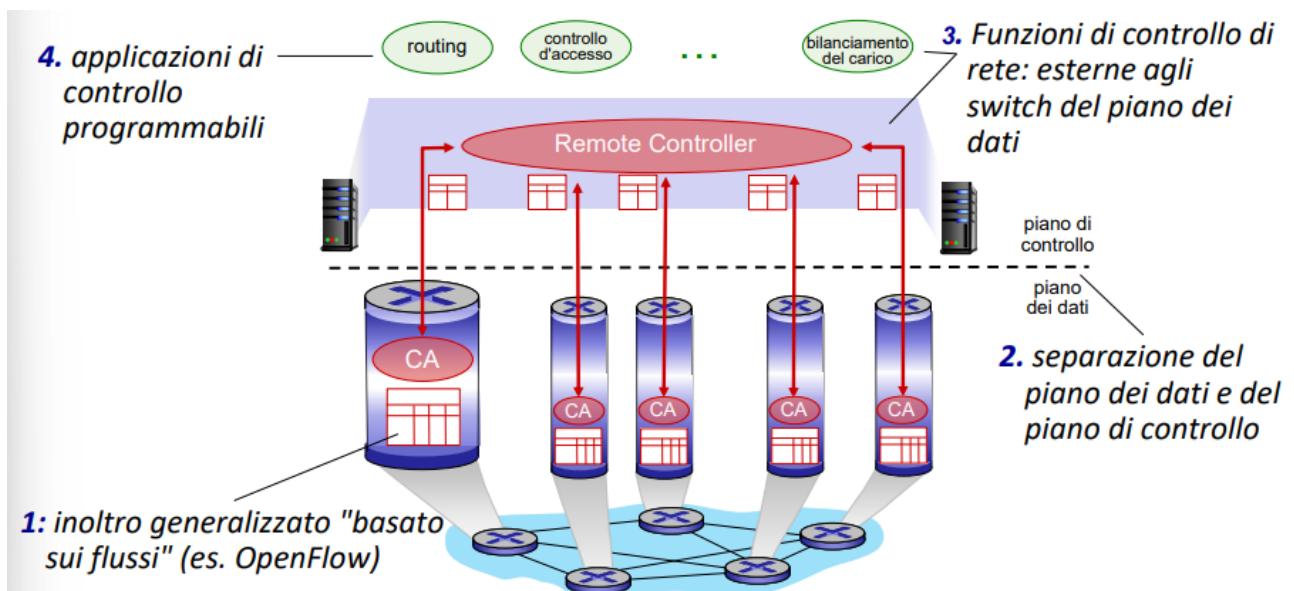
(infatti via inoltro generalizzato posso porre molti più vincoli per instradare come mi pare, senza basarmi solo sull'inoltro tradizionale basato su destinazione).

Come è implementato SDN?

Anzitutto è associato tendenzialmente ad una rete programmabile, con un API più complessa di un semplice inoltro basato su destinazione (es. API su inoltro generalizzato basato su OpenFlow).

Ricorda che quando si parla di SDN e inoltro generalizzato si fa riferimento ai router col nome di **packet switch (switch)**.

È il piano di controllo che si occupa di programmare ogni singolo switch. Il piano di controllo (separato chiaramente dal piano dei dati) contiene il controllore che possiamo vedere come una sorta di sistema operativo, sul quale sono in esecuzione diverse applicazioni che si occupano di diverse questioni (es. applicazione di routing, applicazione sul bilanciamento del carico, sul controllo d'accesso etc...). Queste applicazioni si servono delle informazioni fornite dal controllore remoto per fare i calcoli e si servono del controllore remoto per programmare direttamente i vari switch.



Switch del piano dei dati :

- switch veloci e semplici che implementano l'inoltro generalizzato del piano dei dati in hardware
 - tabella dei flussi (inoltro) calcolata, installata sotto la supervisione del controllore
 - API per il controllo degli switch basato su tabelle (es., OpenFlow)
 - definisce ciò che è controllabile e ciò che non lo è
 - protocollo di comunicazione con il controllore (es. OpenFlow)
-

Per comunicare con i vari switch “in basso” viene utilizzata un’interfaccia detta **Southbound** che usa diversi protocolli, come ad esempio OpenFlow.

Openflow esiste infatti sia come protocollo di comunicazione che come protocollo di programmazione dei router.

Il controllore SDN è una sorta di terra di mezzo, che mantiene lo stato della rete ricevendo degli annunci di stato tramite l’API Southbound dai vari switch.

Il controllore interagisce con le applicazioni di gestione “in alto” tramite l’API **Northbound**.

Il controllore non è quindi il vero e proprio cervello, che in realtà è costituito da queste varie applicazioni, che sono scorporate dal controllore.

Il controllore è implementato come sistema distribuito (logicamente comunque centralizzato) per garantire prestazioni, scalabilità, tolleranza ai guasti, robustezza e sicurezza.

OpenFlow come protocollo lavora tra controllore e switch (Southbound), usa **TCP** per lo scambio di messaggi con crittografia opzionale e *non va confusa con l'API OpenFlow* che viene utilizzata per specificare le azioni di inoltro generalizzate.

Esistono tre classi di messaggi in OpenFlow:

- **Controller to switch**; include i seguenti messaggi chiave:

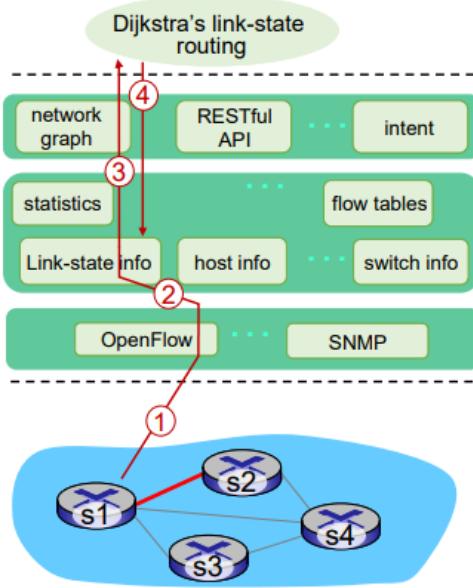
- **Features**: il controllore invia richieste agli switch per venire a conoscenza delle loro caratteristiche;
- **Configure**: vengono impostati i parametri di configurazione dello switch;
- **Modify State**: per aggiungere o modificare il loro stato (ciò include l'installazione delle voci nelle tabelle di flusso);
- **Packet out**: il controllore può inviare questo pacchetto da una specifica porta dello switch.

- **Asynchronous** (switch to controller); con i seguenti messaggi chiave:

- **Packet in** è il messaggio con il quale lo switch può inviare al controllore un pacchetto per farlo ispezionare
- **Flow removed** è una modifica con cui viene indicato che una voce nella tabella di flusso è stata cancellata (molto importante, lo abbiamo visto in Dijkstra: ogni switch deve comunicare lo stato di ciascuna porta al fine di garantire il corretto instradamento)
- **Port Status**: informare il controllore di una modifica su una porta.

Esempio importante da ricordare:

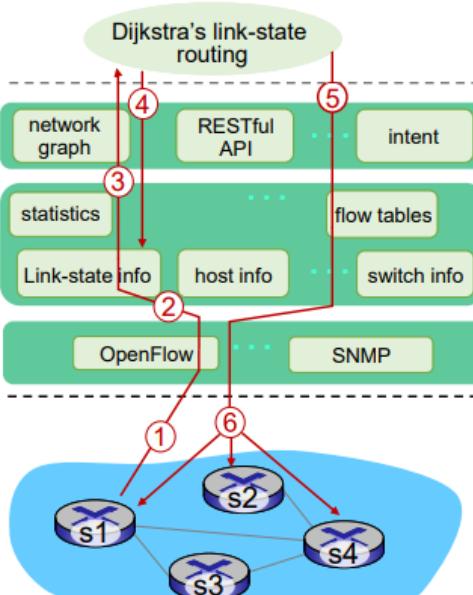
SDN: Esempio di interazione tra piano dei dati e piano di controllo



- ① S1, a causa di un guasto del collegamento, utilizza il messaggio di stato della porta OpenFlow per notificare il controllore.
- ② Il controller SDN riceve il messaggio OpenFlow, aggiorna le informazioni sullo stato del collegamento
- ③ L'applicazione dell'algoritmo di routing di Dijkstra si è registrata in precedenza per essere richiamata quando lo stato dei collegamenti cambia. Viene chiamata.
- ④ L'algoritmo di routing di Dijkstra accede alle informazioni sul grafo della rete, alle informazioni sullo stato dei collegamenti nel controllore e calcola nuovi percorsi.

Network Layer: 5-105

SDN: Esempio di interazione tra piano dei dati e piano di controllo



- ⑤ l'applicazione di link state routing interagisce con il componente flow-table-computation del controller SDN, che calcola le nuove tabelle di flusso necessarie.
- ⑥ il controllore utilizza OpenFlow per installare nuove tabelle negli switch che necessitano di un aggiornamento

OpenDelight e **ONOS** sono due esempi di controllori SDN.

Perché SDN logicamente centralizzato? Perché come detto prima in realtà SDN è a tutti gli effetti implementato in modo distribuito su più macchine (per garantire scalabilità etc...) ma in realtà simula come se fosse centralizzato.

Si noti come a differenza di altri scenari con SDN la sicurezza è stata implementata sin dal primo giorno.

SDN è stato sviluppato prevalentemente nello scenario intra-AS, la prossima sfida è estenderlo per inter-AS e altri scenari particolari come ultra-affidabilità, ultra-sicurezza, scenari real-time.
L'SDN è fondamentale per le reti cellulari 5G.

SDN: sfide selezionate

- Hardening del piano di controllo: sistema distribuito *dependable*, scalabile nelle prestazioni e sicuro
 - robustezza ai guasti: sfruttare la teoria forte dei sistemi distribuiti affidabili per il piano di controllo
 - *dependability*, sicurezza: "incorporati" fin dal primo giorno?
- reti, protocolli che soddisfano i requisiti specifici di missione
 - es., tempo reale, ultra-affidabilità, ultra-sicurezza
- Estensione oltre un singolo AS
- L'SDN è fondamentale per le reti cellulari 5G

Reliability (affidabilità): legato alla probabilità di fallimento.

Availability: probabilità di trovare un sistema in piedi.

Un sistema potrebbe quindi essere molto available perché funziona ma poco reliable perché fallisce spesso e viene rimesso subito dopo in piedi.

Dependability unisce i concetti di reliability e availability.

- Tabelle di inoltro calcolate da SDN rispetto a quelle calcolate da router:
 - solo un esempio di calcolo logicamente centralizzato rispetto al calcolo protocollare
- si potrebbe immaginare un controllo della congestione calcolato da SDN:
 - il controllore impone le velocità dei mittenti in base ai livelli di congestione segnalati dai router (al controllore)



Lez 20

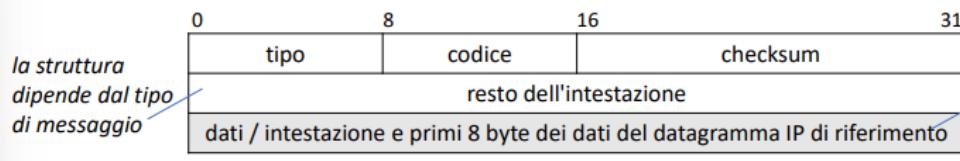
Concludiamo la trattazione del livello di rete parlando dei protocolli usati al fine di scambio di informazioni di controllo, di diagnostica e di monitoraggio e configurazione.

ICMP (*Internet Control Message Protocol*) è il primo protocollo che vediamo in questa categoria.

Si tratta di un protocollo usato da router e host per comunicare informazioni a livello di rete, come segnalazione di errori e richiesta/risposta echo (comando ping, che mi permette di capire che un host è raggiungibile e stimare l'RTT tra noi e quel dispositivo).

I messaggi del protocollo ICMP sono incapsulati all'interno dei datagrammi IP (allo stesso modo di come incapsuliamo all'interno di IP i segmenti del livello di trasporto). *Da un punto di vista implementativo quindi ICMP sembra corrispondere allo stesso gruppo dei protocolli di trasporto, ma non lo consideriamo tale perché non è usato dalle applicazioni di rete per trasferire i propri messaggi* (quindi non ha lo scopo di un classico protocollo del livello di trasporto).

Messaggio ICMP:



in alcuni casi, usato per determinare il datagramma che ha causato il messaggio e determinare il processo corrispondente (assumendo che i numeri di porta si trovino nei primi 8 byte)

Network Layer: 4-114

In generale, un messaggio ICMP è costituito da un'intestazione che consta di 8 byte. Il primo e il secondo campo, **tipo** e **codice**, permettono di implementare

una classificazione a due livelli dei messaggi ICMP (vedi figura sotto).

La checksum calcolata sempre nella stessa maniera.

Altri 8 byte che possono essere usati in maniera diversa o non essere usati affatto (tutti a 0) a seconda del tipo di messaggio.

Ultimi 8 byte che rappresentano il carico utile del messaggio ICMP, non sempre utilizzato. Rappresenta sostanzialmente *l'intestazione e i primi 8 byte di dati del datagramma IP che ha generato il messaggio ICMP*.

es. chiedo al router di inoltrare un pacchetto, quello mi dice che non si può fa -> viene inviato il messaggio ICMP al mittente con riferimento il datagramma che non poteva essere inoltrato.

Perché 8 byte di dati del datagramma? Nei dati di un datagramma si hanno i segmenti TCP e UDP. Un segmento è formato da un'intestazione e da dei dati, e nell'intestazione di un segmento si hanno i numeri di porta. *Avendo questi 8 byte si avranno anche i numeri di porta che permettono di capire gli specifici processi.*

Vediamo più nel dettaglio degli esempi di messaggi ICMP.

ICMP: internet control message protocol

			0	8	16	31
			tipo	codice	checksum	
resto dell'intestazione						
intestazione e primi 8 byte dei dati del datagramma IP di riferimento						
Tipo	Codice	Descrizione				
0	0	echo reply (ping)				
3	0	dest. network unreachable				
3	1	dest host unreachable				
3	2	dest protocol unreachable				
3	3	dest port unreachable				
3	4	fragmentation required				
3	6	dest network unknown				
3	7	dest host unknown				
4	0	source quench (congestion control - not used)				
8	0	echo request (ping)				
9	0	route advertisement				
10	0	router discovery				
11	0	TTL expired				
12	0	bad IP header				

Network Layer: 4-115

Si ha come anticipato una classificazione a due livelli. Per esempio i messaggi di tipo 3 sono genericamente messaggi di destinazione non raggiungibile. Con il codice però poi specifichiamo casi più specifici (3 0 rete non ragg., 3 1 host non ragg. etc...).

Unreachable == risulta avere distanza infinita

Unknown == non si trova una rotta idonea

3 4 è un altro messaggio importante: **fragmentation required**.

Sappiamo che i protocolli a livello di collegamento suddividono i dati in DPU detti Frame, che però hanno capacità massima in termini di carico utile detta MTU (maximum transmission unit). Quando invio un datagramma lo devo inserire nel frame rispettando l'MTU. Se non ce la faccio posso suddividere il datagramma originale in più frammenti che saranno ricostruiti nell'host di destinazione. Questa frammentazione può essere impedita impostando il flag "don't fragment" nell'intestazione del protocollo IPv4. Parlando di IPv6 la frammentazione è stata rimossa (per lo meno dai nodi intermedi, comunque con IPv6 quando invio il messaggio dal mio host posso decidere di frammentare il messaggio usando un'opzione). MTU 576 supportata da tutti i collegamenti IPv4 e IPv6 richiede MTU minima di 1280 byte.

I messaggi ICMP di **fragmentation required** ci possono servire per la Path MTU Discovery.

I messaggi ICMP visti finora sono inviati dai router, ma esistono messaggi che **possono essere inviati dagli host** come 3 2 **dest protocol unreachable** e 3 3 **dest port unreachable**: in questo caso sono messaggi inviati dall'host di destinazione quando il protocollo o la porta non sono attivi.

es. irrealistico: mando un messaggio UDP ma l'host di destinazione non lo supporta, mando indietro il messaggio ICMP protocollo non raggiungibile. Nel caso invece di porta non raggiungibile è chiaramente legato al livello di trasporto ed è un messaggio che può arrivarci solo se stiamo trasmettendo con UDP, perché con TCP già in fase di handshake avevamo stabilito che la porta di destinazione fosse raggiungibile.

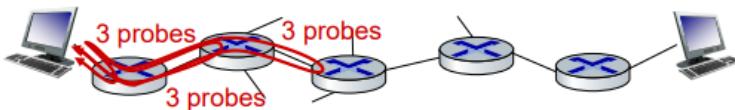
Con **echo request** (ping) 8 0 mando una richiesta echo aspettandomi una **echo reply** (ping) 0 0 che *contiene le stesse informazioni*.

Nel messaggio di tipo echo nei 16 bit tipicamente non usati possiamo trovare un identificatore (per capire quanti messaggi sto inviando in una sessione) e un numero di sequenza (per conoscere la risposta a ciascun messaggio), in generale mi potrebbe servire per calcolare l'RTT (anche se lo posso fare mettendo l'orario nei dati in cui mando e appena mi arriva la risposta sempre con quell'orario faccio il calcolo considerando l'orario attuale).

3	codice	checksum
identificatore		numero di sequenza
dati		

TTL expired 11 0: quando un router inoltra un pacchetto come prima cosa decrementa il campo TTL, e se questo arriva a 0 il pacchetto viene scartato. Mi serviva per evitare che pacchetti circolassero all'infinito nella rete sprecando risorse ed eventualmente minando il concetto di affidabilità se quel pacchetto arriva ad uno con numero di sequenza corretto anche se non c'entrava un cazzo. Quando un router scarta un pacchetto può (non è obbligato) inviare un messaggio TTL expired al mittente.

Questo può essere usato da un'utility chiamata **traceroute**. Io host voglio sapere quali router ci sono nel cammino verso un certo host di destinazione. Che posso fare? Posso sondare il percorso inviando una serie di pacchetti particolari. Del tipo mando pacchetti UDP con numero di porta "improbabile", così se arrivo all'host di destinazione ritorna il messaggio ICMP di dest port unreachable (criterio di arresto, vedi procedimento in toto qui sotto).



- la sorgente invia serie di segmenti UDP alla destinazione (con un numero di porta "improbabile")
 - 1° insieme ha TTL =1, 2° insieme ha TTL=2, ...
- un datagramma nella n -esimo insieme arriva all' n -esimo router:
 - il router scarta il datagramma e invia il messaggio ICMP alla sorgente (tipo 11, codice 0)
 - l'indirizzo IP del router si trova nel campo indirizzo sorgente dell'intestazione del datagramma che incapsula il messaggio ICMP
- quando il messaggio ICMP arriva alla sorgente: registrare gli RTT

criteri di arresto:

- Il segmento UDP arriva eventualmente a destinazione
 - la destinazione restituisce il messaggio ICMP "port unreachable" (tipo 3, codice 3)
- la sorgente si ferma

Gestione e configurazione della rete

Una **rete** (nota anche come **sistema autonomo**) è un *insieme di migliaia di componenti software e hardware che interagiscono tra loro*.

Non è quindi semplice farla funzionare, c'è bisogno di un processo di gestione

della rete che include il fatto di operare, integrare e coordinare hardware, software e persone che si occupano di monitorare, verificare, configurare, controllare le risorse della rete etc... (*tocca insomma verificare che la rete funzioni sempre come dovrebbe, ad es. accorgendoci che un dispositivo si è rotto o un collegamento si è degradato etc...*).

Tutte queste funzioni sono oggi “integrate” nell’SDN: il controllore logicamente centralizzato infatti, tra le altre cose, monitora anche la topologia della rete e lo stato dei collegamenti. Oggi vediamo protocolli usati prima di SDN per queste funzioni di gestione ma *che in realtà sono ancora attuali, poiché alcune di esse incluse ad es. come API southbound per ottenere informazioni dai dispositivi per il controllore.*

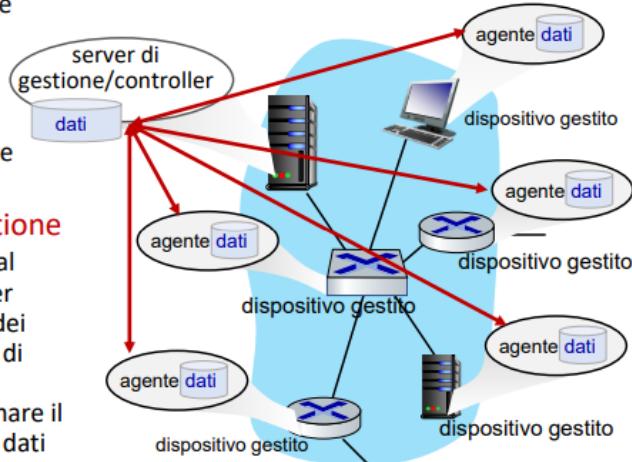
Vediamo anzitutto una descrizione ad alto livello di qual è l’architettura di gestione della rete.

Server di gestione:

raccolta, elaborazione e analisi delle informazioni, invio di informazioni e comandi, in genere con i gestori della rete (umani) nel loop

Protocollo di gestione di rete:

utilizzato dal server di gestione per interrogare lo stato dei dispositivi e agire su di essi; utilizzato dai dispositivi per informare il server di gestione di dati ed eventi.



Dispositivo di rete gestito:

apparecchiature con componenti hardware e software gestibili e configurabili

Dati: “stati” del dispositivo: dati di configurazione (assegnati dall’amministratore, come indirizzo IP), dati operativi (acquisiti da dispositivo, come i vicini OSPF), statistiche

In periferia ho gli host (chiamati così perché ospitano le applicazioni di rete, possono essere server o client). A livello di rete di accesso trovo gli switch (commutatori di pacchetto del livello 2, livello di collegamento) mentre nel nucleo della rete i router. Anche nelle nostre reti di accesso abbiamo un router, quello tipicamente integrato nel modem ADSL o fibra, primo salto al di fuori della nostra rete di accesso.

Tutti questi dispositivi sono apparati hardware ma anche componenti software che voglio poter gestire e almeno parzialmente configurare (dispositivi di rete gestiti). Ciò che ne permette la gestione è la presenza di un **agente** (in ogni

dispositivo) che un po' come per la SDN deve interagire con un **server di gestione/controller**.

Quando il server interagisce con l'agente ciò che nel pratico osserva sono dei **dati**, che definiscono lo stato del dispositivo in termini di tre categorie:

- **Dati di Configurazione**: dati che posso impostare per configurare il dispositivo (es. velocità del collegamento, dimensione del buffer etc..);
- **Dati Operativi**: dati scritti dal dispositivo nel mentre funziona (es. se si parla di router i dati operativi potrebbero essere le tabelle di instradamento che sto calcolando);
- **Statistiche**.

Sul server di gestione gira del software tramite il quale possiamo raccogliere i dati, elaborarli, analizzarli oppure inviare dati di configurazione.

*Ciò viene tipicamente fatto sotto un certo grado di supervisione dei gestori umani della rete (**human in the loop**).*

Per comunicare con gli agenti, il server di gestione utilizza il protocollo di gestione (permesso di comunicare in entrambi i versi).

In termini più pratici, come si approccia l'operatore di rete per gestire la rete? Tre possibili approcci:

- **CLI (linea di comando)** è l'approccio più semplice, l'operatore scrive i comandi su una console del dispositivo o esegue script da remoto attraverso ad es. un protocollo ssh
- **Protocollo SNMP/MIB (Simple Network Management Protocol)**: l'operatore interroga/imposta i dati (modellati come una **MIB** management information base, *una sorta di database*) utilizzando questo protocollo SNMP.
- **NETCONF/YANG**: poiché SNMP era storicamente più utilizzato per la lettura e quindi il monitoraggio che per la scrittura e quindi la gestione vera e propria, si è sviluppata quest'alternativa. È concettualmente simile a SNMP ma con enfasi maggiore sul definire la configurazione di rete piuttosto che definire la configurazione dei singoli dispositivi (configurazione multidispositivo, ne faccio di più insieme contemporaneamente boh). NETCONF è il nome del protocollo e YANG è il linguaggio tramite il quale definiamo quali dati sono accessibili nel dispositivo gestito.

Protocollo SNMP

SNMP può usare vari protocolli di trasporto, in genere utilizza UDP.

Dal punto di vista del protocollo di applicazione abbiamo due modalità di

comunicazione:

- la classica **richiesta/risposta** (il server di gestione, detto **client SNMP**, manda una richiesta all'agente SNMP che poi invia una risposta);
- **modalità trap**: l'agente in maniera proattiva invia un messaggio detto trap al server di gestione. Ciò serve per informare il server di un evento imprevisto.

Tipo di messaggio	Funzione
GetRequest GetNextRequest GetBulkRequest	manager→agente: "dammi dati" (istanze di oggetto, prossima istanza di oggetto in lista o tabella, blocco di dati).
SetRequest	manager→agente: imposta il valore di una o più istanze di oggetti MIB
Response	agente→manager: generato in risposta a una richiesta
Trap	agente→manager: informa il manager di un evento inatteso

Nel formato di questi messaggi (PDU) deve essere presente un ID che associa la richiesta alla risposta (si ricorda infatti che tipicamente si usa UDP per mandare questi messaggi)

SNMP accede quindi ai dati dal dispositivo (dati di configurazione, dati operativi, statistiche). Questi dati sono modellati, come detto, nella **MIB** (management information base) che può essere visto come *un database gerarchico: si ha una root con figli che hanno figli etc...*

In un certo modo ricorda un file system, come identifico un oggetto che voglio leggere in MIB? *Tramite un percorso definito dai vari componenti separati da un separatore. I componenti in questo caso sono numeri ed il separatore un punto.* **OID rappresenta quindi proprio un cammino nella base di dati MIB gerarchica**, che mi permette di identificare l'oggetto MIB specifico che potrò leggere o scrivere.

```

└─ SNMPv2-MIB(.1.3.6.1.2.1)
   └─ system(.1)
      ├─ sysDescr (.1)
      ├─ sysObjectID (.2)
      ├─ sysUpTime (.3)
      ├─ sysName (.5)
      ├─ sysContact (.4)
      ├─ sysLocation (.6)
      ├─ sysServices (.7)
      ├─ sysORLastChange (.8)
      └─ sysORTable (.9)
         └─ sysOREntry (.1)
            ├─ sysORIndex (.1)
            ├─ sysORID (.2)
            ├─ sysORDescr (.3)
            └─ sysORUpTime (.4)

```

- gerarchico
- ogni voce è indirizzata da un OID (*object identifier*)

Lo schema di questi dati è definito dal linguaggio **SMI** (stessa funzione di YANG per NETCONF).

- i dati operativi (e alcuni dati di configurazione) del dispositivo gestito
- raccolti **moduli MIB**
 - 400 moduli MIB definiti da RFC; molte più moduli MIB specifici del fornitore
- **Structure of Management Information (SMI)**: linguaggio di definizione dei dati
- esempio di variabili MIB per il protocollo UDP:



NETCONF

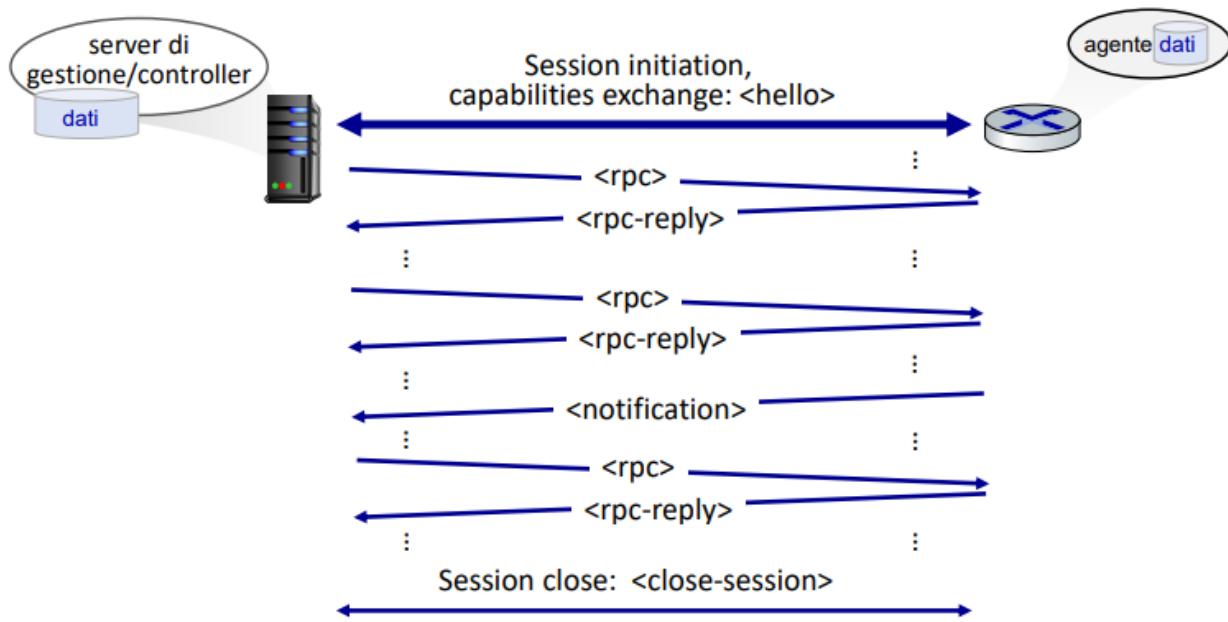
Per NETCONF il discorso è simile a SNMP, si tratta semplicemente di un’alternativa più moderna.

L’**obiettivo**, come per SNMP, è **gestire ma soprattutto configurare** (cosa per cui SNMP non era stata nativamente progettata) **in maniera attiva i dispositivi sulla rete**.

Come SNMP ha le trap, NETCONF ha le **notifiche** (sono la stessa cosa).

Come avviene l’interazione tra due estremi nel protocollo NETCONF?

Tramite una **chiamata a procedura remota rpc** fatta stabilendo una sessione con un protocollo di trasferimento dati affidabile e sicuro, inviando messaggi **xml** atti a codificare sia la richiesta che la risposta.



Dentro all'oggetto rpc possiamo trovare uno tra questi elementi, a seconda di quello che stiamo facendo:

Operazione	Descrizione
<get-config>	Recupera tutta o parte di una data configurazione. Un dispositivo può avere più configurazioni. C'è sempre una configurazione <i>running</i> , che descrive la configurazione corrente (in esecuzione) dei dispositivi.
<get>	Recupera tutti o parte dei dati operativi e della configurazione <i>running</i> .
<edit-config>	Modifica la configurazione (possibilmente in esecuzione) del dispositivo gestito. Quest'ultimo invia un <rpc-reply> contenente <ok>; altrimenti viene inviato un <rpcerror> con rollback.
<lock>, <unlock>	Bloccare (sbloccare) il datastore di configurazione sul dispositivo gestito (per bloccare i comandi NETCONF, SNMP o CLI da altre fonti).
<create-subscription>, <notification>	Abilita la sottoscrizione di notifiche di eventi dal dispositivo gestito

Vediamo un esempio di richiesta:

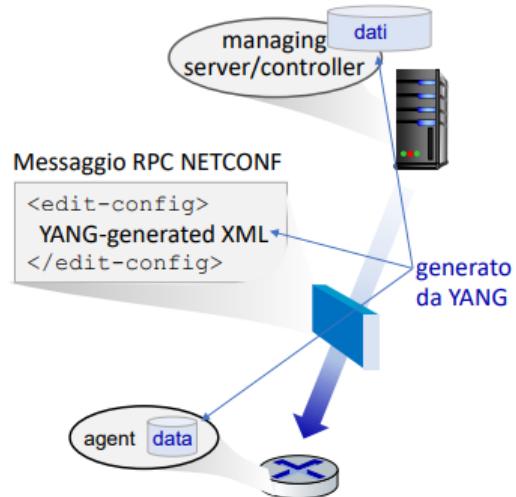
```

01 <?xml version="1.0" encoding="UTF-8"?>
02 <rpc message-id="101" nota l'id del messaggio
03   xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
04     <edit-config> cambia una configurazione
05       <target>
06         <running/> cambia la configurazione in esecuzione
07       </target>
08     <config>
09       <top xmlns="http://example.com/schema/
10         1.2/config">
11           <interface>
12             <name>Ethernet0/0</name>
13             <mtu>1500</mtu>      cambia la MTU dell'interfaccia Ethernet 0/0 in 1500
14           </interface>
15         </top>
16       </config>
17     </edit-config>
18   </rpc>

```

YANG

- linguaggio di modellazione dei dati utilizzato per specificare la struttura, la sintassi e la semantica dei dati di gestione della rete NETCONF
 - tipi di dati incorporati, come SMI
- documento XML che descrive il dispositivo; può essere generato dalla descrizione YANG
- può esprimere vincoli tra i dati che devono essere soddisfatti da una configurazione NETCONF valida
 - garantire che le configurazioni NETCONF soddisfino i vincoli di correttezza e di coerenza



Livello di Collegamento

Il livello di collegamento si situa sotto il livello di rete e sopra il livello fisico, **si occupa di trasferire i pacchetti da un nodo a quello adiacente** (mentre il livello di rete si occupa dell'instradamento lungo tutto il percorso). Quindi per ogni singolo passo il livello di rete si appoggia al livello di collegamento.

Il livello di rete mittente manda un pacchetto al destinatario. Non trovandosi nella stessa sottorete, viene posto in un frame destinato al router di primo hop (router di default), il livello di collegamento lo trasferisce al router e a quel punto lo manda al livello di rete. Il livello di rete si accorge di non essere il destinatario e quindi lo rimanda al livello di collegamento mettendo come destinatario il router dell'hop successivo (in base alle tabelle di instradamento) finché non si arriva al destinatario effettivo per cui il livello di rete non rimanda il pacchetto al livello di collegamento ma lo trasferisce al livello superiore.

Quando si parla di livello di collegamento i dispositivi che operano al livello due sono detti genericamente **nodi (host, router, switch == nodi)**.

Consideriamo **adiacenti** i *nodi collegati dallo stesso collegamento* (filo o rete wireless).

In uno stesso percorso si possono chiaramente avere tanti collegamenti anche di diversa natura (wireless, cablato etc...). Si ricorda che le unità di dati del livello di collegamento (livello due) sono dette Frame e dentro ci mettiamo i datagrammi del livello di rete.

Mentre con il livello di rete il concetto era attraversare n collegamenti, con il livello di collegamento si attraversa un collegamento alla volta.

Poiché ho diverse tecnologie di collegamento non sorprende si abbiano diversi protocolli di collegamento: es. primo hop WLAN (wireless LAN wi-fi) e secondo hop ethernet cablata...

Servizio principale del livello di collegamento == trasferire i dati da un nodo a quello fisicamente adiacente lungo un collegamento fisico.

Servizi del livello di collegamento:

Accesso al collegamento: è necessario un protocollo che si occupi di disciplinare l'accesso al mezzo trasmissivo (se ci arrivano dati diversi insieme non si comunica correttamente, dati corrotti).

Viene introdotto in questo senso l'indirizzo MAC (per identificare la sorgente e la destinazione ma limitandosi al livello 2, per questo diversi dagli indirizzi IP)

Consegna affidabile tra nodi adiacenti: stesso problema visto a livello di trasporto; voglio che i dati siano consegnati senza errori nei bit, in ordine. Si può fare a più livelli (applicativo QUIC, trasporto TCP, collegamento).

Il trasferimento affidabile a livello di collegamento è applicato soltanto quando il mezzo è molto soggetto ad errori (es. wireless interferenze) e quindi si vuole inquadrare l'errore nel modo più locale possibile *senza forzare il mittente a recuperare l'errore via end to end* (ovvero voglio evitare che sia l'host a informare il mittente dell'errore e farsi reinviare il messaggio corretto, tramite implementazioni nel mezzo che correggano preventivamente l'errore).

Controllo di Flusso: il mittente regola il trasferimento dei dati in funzione della capacità del destinatario (in questo caso il nodo adiacente) di gestirli.

Half duplex: le due estremità si devono alternare nella trasmissione

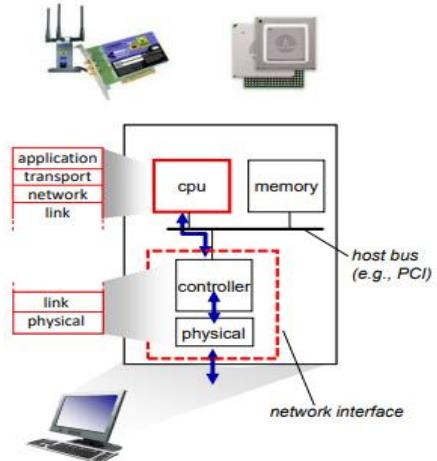
Full duplex: le due estremità possono trasmettere in contemporanea

Framing: i datagrammi vengono incapsulati in frame, aggiungendo un'intestazione e un trailer

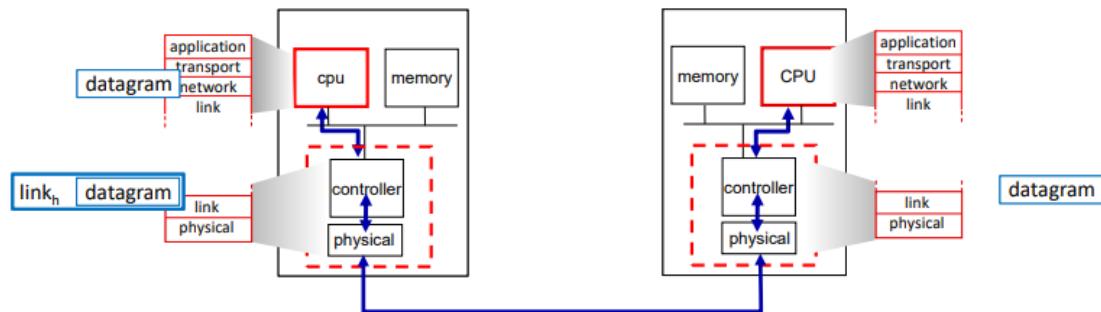
Rilevazione e Correzione degli errori: permette di far sì che il trasferimento sia effettivamente affidabile (se non rilevassi l'errore non lo sarebbe)

Implementazione del livello di collegamento negli host

- in ogni singolo host
- il livello di collegamento implementato (principalmente) dall'adattatore di rete (*network adapter*) o scheda di rete (*network interface card*, NIC)
 - implementa il livello di collegamento e quello fisico
 - si collega al *bus* di sistema
- combinazione di hardware, software e firmware



Adattatore di rete negli host



Lato mittente, il controllore:

- incapsula il datagramma in un frame
- aggiunge bit di controllo degli errori, implementa il trasferimento di dati affidabile, il controllo del flusso, etc.

Lato ricevente, il controllore:

- verifica la presenza di errori e si occupa del trasferimento dati affidabile, del controllo di flusso, etc.
- estraie il datagramma e lo passa al livello superiore

Rilevazione e Correzione degli errori:

La prima cosa per far sì che un trasferimento sia affidabile è rendersi conto della presenza di un errore, per farlo si usano tecniche di rilevazione dell'errore.

Viene inclusa nei messaggi un'informazione ridondante (detta **checksum** o più propriamente **EDC** Error Detection and Correction) che ci servono a capire se i dati sono stati corrotti.

Che succede se i dati sono stati corrotti? Se li voglio trasferire in modo affidabile li devo correggere, ma che significa correggere? Due approcci:

- **ARP**: scarto il pacchetto corrotto e me lo faccio ritrasmettere

- **Codici di Correzione degli Errori (FECC forward error correction codes)**: nella tecnica EDC è inclusa, oltre che la possibilità di rilevare un errore, anche di correggerlo direttamente.

Controllo di parità



Singolo bit di parità (parity bit):

- Rileva un numero *dispari* di errori

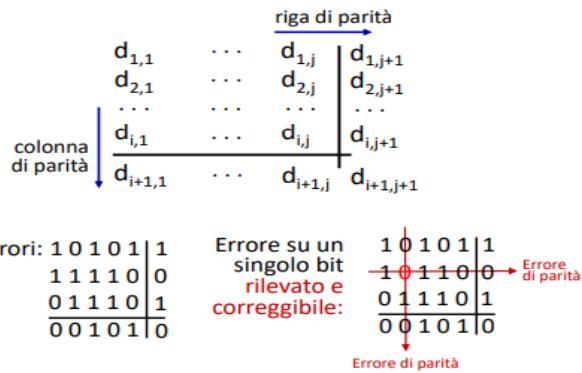


Il ricevente:

- calcola la parità dei d bit ricevuti
- lo confronta con il bit di parità ricevuto – se differente, allora è stato rilevato un errore

Parità bidimensionale:

- rileva tutte le combinazioni di al più 3 errori
- rilevazione e *correzione* di errori singoli



Usando l'**EDC del Controllo di Parità** decido di inviare il messaggio con l'informazione relativa al suo numero di uni (pari o dispari). Se un singolo bit del messaggio cambia, allora passo da pari a dispari o viceversa e quindi mi accorgo dell'errore (per questa ragione **questo controllo rileva solo un numero dispari di errori**).

Con la versione bidimensionale chiaramente si utilizza più spazio (n^2 se voglio mandare un messaggio n) ma ho la possibilità di **rilevare tutte le combinazioni di al più tre errori e di correggere i singoli errori**.

Il controllo di parità è adatto **quando la probabilità di errori nei bit è bassa** e **gli errori sono indipendenti** (nella realtà tuttavia gli errori tendono a verificarsi in burst).

<p>Due errori sulla stessa riga, rilevati su colonne differenti</p> <table border="1" style="margin-left: 20px; border-collapse: collapse; text-align: center;"> <tr><td></td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> <tr><td></td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td></td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td></td><td colspan="6"><hr/></td></tr> <tr><td></td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> </table> <p>Senza errori: 1 0 1 0 1 1 1 1 1 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0</p> <p>Errore su due bit rilevato ma non correggibile: 1 0 1 0 1 1 1 0 1 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0</p> <p>Errore di parità ↓ Errore di parità ↓</p>		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	<p>Due errori sulla stessa colonna, rilevati su righe differenti</p> <table border="1" style="margin-left: 20px; border-collapse: collapse; text-align: center;"> <tr><td></td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> <tr><td></td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td></td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td></td><td colspan="6"><hr/></td></tr> <tr><td></td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> </table> <p>Senza errori: 1 0 1 0 1 1 1 1 1 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0</p> <p>Errore su due bit rilevato ma non correggibile: 1 1 1 0 1 1 1 0 1 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0</p> <p>Errore di parità → Errore di parità →</p>		1	1	1	0	1	1		1	0	1	1	0	0		0	1	1	1	0	1		<hr/>							0	0	1	0	1	0
	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																																																																	
	1	1	1	0	1	1																																																																	
	1	0	1	1	0	0																																																																	
	0	1	1	1	0	1																																																																	
	<hr/>																																																																						
	0	0	1	0	1	0																																																																	
<p>Errore su due bit rilevato ma non correggibile: 1 0 1 0 1 1 1 0 1 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0</p> <p>Senza errori: 1 0 1 0 1 1 1 1 1 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0</p> <p>Errore su quattro bit non rilevato: 1 0 1 0 1 1 1 0 1 1 1 0 0 0 1 1 1 1 0 0 1 0 1 0</p> <p>Ma non significa che non si possano rilevare <i>alcuni</i> errori su quattro bit</p> <p>Errore su quattro bit rilevato ma non correggibile: 0 0 1 0 1 1 1 0 1 1 0 0 0 1 0 1 0 1 0 0 1 1 1 0</p> <p>Errore di parità ↓ Errore di parità ↓ Errore di parità ↓ Errore di parità ↓</p>																																																																							

Lez 21

Abbiamo visto come *lo scopo principale del livello di collegamento è trasmettere i datagrammi del livello di rete da un nodo a quello fisicamente adiacente lungo un singolo collegamento.*

Ogni tratto nell'intero percorso da host mittente a destinatario quindi può avere servizi diversi (wireless, cablato, fibra...).

Tra gli altri servizi che offre il livello di collegamento vi è quello di rilevazione e correzione degli errori: vengono aggiunti bit ridondanti al messaggio inviato e, secondo l'**EDC** (*error detection and correction*) che si utilizza (abbiamo visto checksum e bit di parità), vengono rilevati ed eventualmente corretti gli errori riscontrati.

Per correggere gli errori, come già detto nella scorsa lezione, esistono due

approcci: **ARC** (per cui viene fatto ritrasmettere il messaggio in toto) e **Codici di Correzione degli errori (FECC)** forward error correction codes), che invece si occupano di correggere direttamente l'errore nel messaggio corrotto.

Ma perché preoccuparsi di correggere errori a livello di collegamento se già sappiamo che se ne occupa TCP a livello di trasporto?

Quando un collegamento è molto soggetto a errori, può essere più conveniente risolvere il problema limitatamente al singolo collegamento piuttosto che richiedere che il mittente ritrasmetta il pacchetto (si impiegano molte mento risorse!).

Negli host il livello di collegamento è principalmente implementato a livello di **scheda o adattatore di rete**, dove troviamo un controllore che in hardware incapsula il datagramma nel frame e lo manda al nodo adiacente, il cui controllore estrapola il datagramma e, se non è destinato a lui, lo reincapsula e lo manda al nodo successivo secondo le regole descritte nelle tabelle di instradamento.

Torniamo agli EDC. Sappiamo che per ogni EDC esiste sempre una probabilità per cui l'errore non venga rilevato, il nostro obiettivo è ridurla.

Con il **controllo di parità bidimensionale** abbiamo visto come sia garantito sempre il rilevamento di al più 3 errori e la correzione se ve ne è stato uno solo. Tuttavia se ho 4 errori, **non è detto** che l'EDC li riesca a rilevare (caso in cui due errori sulla stessa colonna e gli altri due sulla stessa riga dei primi due).

Ma quanto è efficace in fin dei conti questo EDC? **Ha senso utilizzarlo chiaramente solo se la quantità di errori che ci si aspetta è bassa e gli errori sono indipendenti, ma nella realtà gli errori tendono ad occorrere in burst e quindi a non essere indipendenti.**

In pratica quindi se un bit è errato c'è una probabilità maggiore che anche i successivi lo siano.

Altro EDC che abbiamo già visto è la **checksum**:

Mittente: i dati sono divisi in sequenze da 16 bit, li sommo in complemento ad 1 (cioè li sommo e se ho un bit di riporto lo sommo al bit meno significativo, rivedi la checksum Lez 7, protocollo UDP) e una volta fatto ne faccio il complemento a 1 (inverto tutti i bit), ciò che ottengo è la checksum che invio al destinatario insieme al messaggio.

Ricevente: si rifa la somma in complemento a 1 come aveva fatto il mittente,

sommadi poi il checksum ricevuto. A quel punto se il risultato è costituito da tutti i bit a 1 allora significa che non sono stati rilevati errori (ma non è detto, come al solito, che non ve ne siano stati).

Nuovo EDC: **CRC** (codice di controllo a ridondanza ciclica)

Permette di rilevare una maggior percentuale di errori rispetto agli altri EDC che abbiamo visto.

L'idea è di prendere dei **D** dati (**d bit**) che voglio trasmettere. Il mittente e il destinatario si accordano su una sequenza **G** di **r + 1 bit**, detta **generatore o polinomio generatore**.

Intrepreto entrambi come dei numeri, su cui però utilizzeremo un'aritmetica particolare.

Aggiungiamo a D una stringa R di r bit. Ma cos'è R? Calcolo gli r bit di R in modo tale che la stringa <D,R> sia divisibile esattamente per G secondo l'aritmetica particolare.

Quando il ricevente riceve il messaggio, conoscendo G, divide <D,R> per G. Se il resto è diverso da zero, allora significa che è stato rilevato un errore.

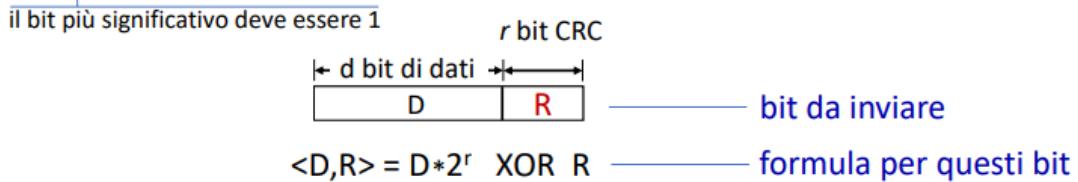
Il CRC è così potente per via del fatto che può in questo modo rilevare tutti gli errori a burst di lunghezza inferiore a r + 1 bit (se metto R di 32 bit, avrò protezione contro burst di 32 bit e così via).

CRC è largamente usato in Ethernet e in Wireless LAN.

Ma se questo CRC è più potente, perché UDP e altri protocolli usano EDC più semplici come bit di parità o checksum?

Perché CRC è implementato in software, e volendo risparmiare tempo in termini di microsecondi per certi protocolli non ci si può permettere l'ausilio di algoritmi così sofisticati.

- codifica di rilevamento degli errori più potente
- **D**: *dati da trasmettere* (d bit)
- **G**: sequenza di $(r + 1)$ bit concordata, detta *generatore* (definito nello standard CRC)



mittente: calcola r bit CRC, **R**, tali che $\langle D, R \rangle$ si *divisibile esattamente* da **G** ($\text{mod } 2$)

- il ricevente conosce **G**, divide $\langle D, R \rangle$ per **G**. Se il resto è diverso da zero: errore rilevato!
- può rilevare tutti gli errori a burst di lunghezza inferiore a $r+1$ bit (ovvero, errori di non più di r bit consecutivi)
- la frazione di burst più lunghi che può rilevare è approssimativamente $1 - 2^{-r}$
- largamente usato in pratica (Ethernet, 802.11 WiFi)

Vediamo ora sta cazzo di aritmetica di CRC: si tratta di aritmetica modulo 2 senza riporti nelle addizioni e prestiti nelle sottrazioni.

- addizione e sottrazione sono la stessa operazione, corrispondente all'or esclusivo (*exclusive or*, XOR) bit a bit

$$\begin{array}{r} 1011 + \\ 1101 = \\ \hline 0110 \end{array} \quad \begin{array}{r} 1011 - \\ 1101 = \\ \hline 0110 \end{array} \quad 1011 \text{ XOR } 1101 = 1011 \oplus 1101 = 0110$$

La moltiplicazione e divisione sono calcolate come al solito, tenendo a mente che poi la somma è fatta come appena detto

$$\begin{array}{r} 1011 \times \\ 101 = \\ \hline 1011 + \\ 0 + \\ \hline 1011 = \\ 100111 \end{array}$$

Questa "strana" aritmetica corrisponde al vedere le sequenze di bit come i coefficienti (**modulo 2**) di polinomi

Link Layer 21

$$\begin{array}{r}
 1011 \times \longrightarrow x^3 + x + 1 \\
 101 = \longrightarrow x^2 + 1 \\
 \hline
 1011 + \\
 0 + \\
 \hline
 1011 = \\
 100111 \longrightarrow x^5 + x^2 + x + 1
 \end{array}$$

assumendo che il primo bit sia 1, allora il grado del polinomio è uguale al numero di bit - 1

$$\begin{array}{l}
 (x^3 + x + 1) \cdot (x^2 + 1) = (x^5 + x^3 + x^2) + (x^3 + x + 1) = x^5 + \\
 (1 + 1) x^3 + x^2 + x + 1 = x^5 + x^2 + x + 1
 \end{array}$$

somma modulo 2

Ma come calcolo nel pratico R? Come detto, voglio che $\langle D, R \rangle$ sia divisibile per G, cioè deve essere multiplo di G.

Nei calcoli successivi 2^r perché bit più a sx di R a 1 $\rightarrow 2^r$ (R ha r bit) e XOR R non lo so (non si capisce un cazzo quando spiega porcoddio)
 Comunque $\langle D, R \rangle = D \cdot 2^r \text{ XOR } R$ in generale, voglio che sia multiplo di G quindi $D \cdot 2^r \text{ XOR } R = nG$

Codici di controllo a ridondanza ciclica: esempio

Il mittente vuole calcolare R
 tale che:
 $D \cdot 2^r \text{ XOR } R = nG$

... o equivalentemente (XOR R in entrambi i lati):

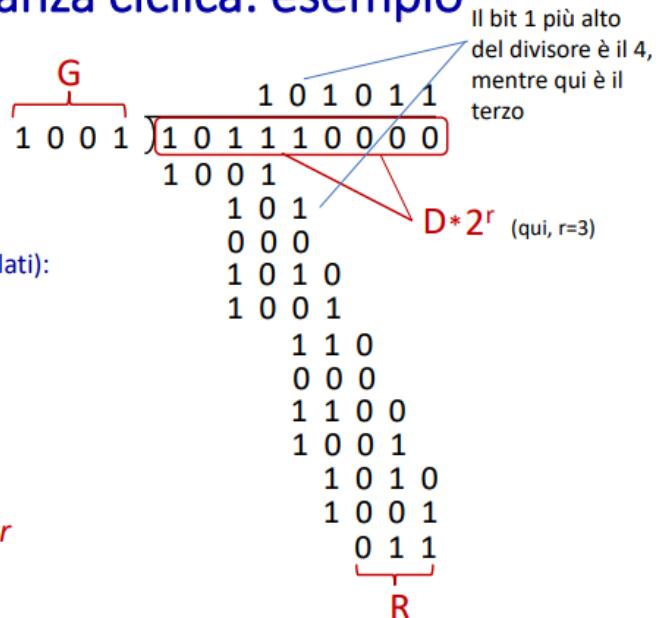
$$D \cdot 2^r = nG \text{ XOR } R$$

... che ci dice che:

se dividiamo $D \cdot 2^r$ per G il resto è precisamente R

$$R = \text{resto} \left[\frac{D \cdot 2^r}{G} \right]$$

algoritmo per calcolare R



A destra in alto: G ha 4 bit ($r+1$) quindi R ne deve avere 3, quindi aggiungo 3 zeri a D.

Fatto ciò, inizio una serie di divisioni. Divido i 4 bit (perché numero di bit di G == 4) più significativi di D per G (1011/1001). Si tratta di fare lo XOR: ottengo 0010. I due zeri a sinistra si tolgono, resto con 10. Ma devo ancora dividere per G (1001) e quindi mi servono altri due bit: prendo quindi i due bit successivi ai 4 che avevo usato prima per fare la divisione in D (101110). Ottengo così in riga 5 1010 e divido di nuovo per 1001 (G). Ottengo 11, ancora una volta mi servono altri due bit -> prendo i due successivi a D rispetto a quelli usati prima (10111000).

Devo quindi dividere 1100 per 1001, ottengo 101. Mi manca l'ultimo bit da aggiungere di D, 1010/1001 == 011. Questo è il mio valore di R che dovrò mettere al posto dei 3 zeri in D!

il mittente invia $T = D \cdot 2^r \text{ XOR } R$

il ricevente riceve $T' = T \text{ XOR } E = T + E$ (usando l'addizione CRC)

dove E è un numero (ovvero un polinomio), i cui bit a 1 indicano dove si è verificato un errore.

Siccome T è divisibile per G, T' sarà divisibile per G se e solo se E è divisibile per G.

Quindi, *G deve essere scelto in modo tale che NON divida i polinomi di errore.*

In pratica, si cerca di definire G in modo che rilevi diversi tipi di errore.

Come detto qua sopra, *anche il generatore non viene scelto casualmente ma con proprietà tali da consentire che questo possa rilevare diversi tipi di errore.*

Se G ha un numero pari di bit a 1, allora è in grado di rilevare qualsiasi numero dispari di errori.

Perché?

La moltiplicazione CRC equivale a fare lo XOR di copie shiftate dello stesso numero.

Se questo numero ha parità pari, lo XOR con un altro valore (es. il risultato parziale) produce un risultato che ha la parità di quest'ultimo.

Quindi i multipli di G devono avere parità pari.

È anche possibile definire G in modo che sia multiplo di 11 (cioè $x + 1$).

(non si capisce un cazzo da come lo spiega, non spreco nemmeno tempo a trascrivere questa parte. Recuperala in caso ma non penso la metta all'esame)

Collegamenti e protocolli di accesso multiplo

Esistono due tipi di collegamenti:

- **punto-punto**: *ho un singolo trasmittente e un singolo ricevente alle due estremità* (es. collegamento punto-punto tra host e switch ethernet)
- **broadcast**: *si hanno più nodi, ciascun frame viene ricevuto da tutti i nodi connessi* (es. Ethernet vecchia scuola con cavo condiviso, wireless LAN, 4G...)

Se ho un mezzo broadcast condiviso per cui ho più potenziali trasmettitori, se più nodi trasmettono contemporaneamente **si può verificare una collisione** per cui il segnale di entrambi viene corrotto! *È quindi necessario un protocollo di accesso multiplo che disciplina come i nodi che condividono un mezzo broadcast debbano trasmettere al fine di non collidere tra loro.*

È necessario far ciò in **maniera distribuita, senza ricorrere a un ulteriore canale**.

Supponiamo che il nostro canale condiviso abbia velocità massima di trasmissione **R**. Idealmente ciò che vorremmo è un protocollo che permetta a un singolo trasmettitore di trasmettere a velocità **R**. Se si hanno **M** nodi allora vorremmo una distribuzione equa della velocità con **velocità media R/M** per ogni nodo. Si vuole poi che *l'algoritmo sia totalmente decentralizzato*, senza cioè un nodo speciale che coordini le trasmissioni e *senza bisogno di sincronizzazione tra i vari nodi*.

Abbiamo tre classi principali di protocolli che si adattano a queste esigenze:

- **(1) Protocolli a Suddivisione del Canale (channel partitioning)**: *Il canale viene diviso in “pezzi” più piccoli* (in termini di tempo, frequenza o codice).

Li abbiamo già incontrati, sono la divisione per tempo **TDM** (time division multiplexing) e per bande di frequenza **FDM** (frequency division multiplexing). Un altro approccio che vedremo più avanti è quella per divisione di codice (i dati sono codificati in modo tale che se anche interferiscono fisicamente non si corrompono).

Nella TDM ogni trasmettitore trasmette a velocità **R** ma solo in una certa frazione di tempo con velocità media **R/M**, nella FDM è riservata continuamente una porzione della banda per far sì che ogni trasmettitore trasmetta continuamente a velocità **R/M**.

Nonostante la suddivisione del canale permetta chiaramente di evitare problemi di collisione, comporta altri tipi di problemi. In particolare, se il canale è poco utilizzato, poiché assegno in maniera statica ad ogni trasmettitore una porzione di tempo/banda/codice allora ne spreco se non trasmettono! Quindi ha senso utilizzarla solo se il canale è molto utilizzato.

- **(2) Protocolli ad Accesso Casuale:** quando un nodo ha dati da inviare, trasmette alla massima velocità consentita dal canale senza alcun coordinamento. Ciò tuttavia significa che due nodi possono trasmettere contemporaneamente e quindi “consente” le collisioni, ma è progettato appositamente per poterle **rilevare e recuperare** (attraverso ritrasmissioni)

Slotted ALOHA

Con questo protocollo si fanno diverse assunzioni: *tutti i frame hanno la stessa dimensione, il tempo è diviso in slot temporali tutti uguali di lunghezza pari al tempo di trasmettere un frame, le trasmissioni avvengono soltanto all'inizio degli slot e i nodi sono sincronizzati* (nel senso che sanno quando uno slot inizia).

Si assume con Slotted ALOHA che quando avviene una collisione questa prima del termine dello slot possa essere rilevata (tramite un meccanismo di ACK).

Se un nodo ha un nuovo frame da spedire viene spedito immediatamente, senza preoccuparmi se gli altri stanno o meno trasmettendo. Dopo che ho trasmesso, controllo se si è verificata una collisione. Se no posso inviare un nuovo frame nello slot immediatamente successivo, se sì randomizzo, ovvero: con probabilità p ritrasmetto il frame allo slot successivo, altrimenti con probabilità $1-p$ (come se lanciassi una moneta) aspetto e rifaccio il ragionamento allo slot successivo finché non invio il frame con successo.

Ma perché randomizzare?

Immaginiamo due persone che s'incrociano per strada, uno si deve spostare e l'altro rimanere fermo prima di passare. Allo stesso modo se tutti ritrasmettessero saremmo punto e a capo, di nuovo collisione, con la randomizzazione invece si arriverà al punto in cui uno trasmette mentre l'altro sta fermo.

I **vantaggi** di questo algoritmo è che è **semplice e altamente decentralizzato** (solo gli slot nei nodi devono essere sincronizzati). Il vantaggio più grande è ***che se è uno solo il nodo che sta trasmettendo allora a tutti gli effetti trasmetterà sempre alla massima velocità***, senza sprecare risorse nel collegamento.

Gli **svantaggi** risiedono nel fatto che **se si verificano collisioni allora si sprecano slot di tempo**, inoltre anche *il meccanismo di randomizzazione può determinare degli slot di tempo totalmente inutilizzati* (se tutti restano fermi), altro svantaggio il fatto che *deve esserci sincronizzazione degli orologi*.

Slotted ALOHA: efficienza

efficienza: frazione a lungo termine di slot riusciti (molti nodi, tutti con molti frame da inviare)

- *si supponga: N nodi con molti frame da inviare, ciascuno trasmette nello slot con probabilità p*
 - probabilità che un dato nodo ha successo in uno slot = $p(1-p)^{N-1}$
 - probabilità che un nodo qualunque abbia successo = $Np(1-p)^{N-1}$
 - efficienza massima: trovare p^* che massimizza $Np(1-p)^{N-1}$
 - per molti nodi, calcolare il limite di $Np^*(1-p^*)^{N-1}$ per N che tende all'infinito, si ottiene:

$$\text{efficienza massima} = 1/e = .37$$

- **al massimo:** canale usato per trasmissione utile solo per il 37% del tempo!



Solo il 37% degli slot svolge un lavoro utile. Pertanto, la velocità di trasmissione effettiva del canale non è R bps, ma solo 0.37 R bps! Un'analisi simile mostra anche che il 37% degli slot va a vuoto e il 26% degli slot subisce collisioni.

Esiste una variante di Slotted ALOHA detto **ALOHA puro** che non richiede la sincronizzazione. Il nodo trasmette non appena ha da trasmettere (non aspetta l'inizio di un nuovo slot) e dopo che ha terminato la trasmissione se si verifica una collisione si ragiona randomicamente come prima. Tuttavia **senza sincronizzazione ALOHA puro scende ancora più in basso del 37% dello Slotted ALOHA: si arriva al 18%** di efficienza massima del protocollo (infatti senza sincronizzazione aumenta la probabilità di collisione).

Ciò che rende così difficile l'approccio per i protocolli di accesso multiplo è il fatto che nei wireless non è banale osservare che un canale è occupato.

Con il filo la questione è più semplice, per questo Ethernet cablato vecchia scuola (condiviso) utilizzava un protocollo di accesso casuale detto **CSMA**.

CSMA (carrier sense multiple access)

Quando un trasmettitore ha nuovo frame da trasmettere, per prima cosa *“percepisce” il canale (rilevamento della portante)*. Se lo trova **inattivo** trasmette l'intero frame, se lo trova **occupato** *differisce la trasmissione*, per “non interrompere” gli altri, ripetendo il meccanismo finché non lo trovo libero.

Si può anche adottare un metodo leggermente più sofisticato, aggiungendo un rilevamento della collisione (**CSMA/CD**). *Quando sto trasmettendo continuo a percepire il canale, e se rilevo una collisione mi interrompo subito*, riducendo così gli sprechi del canale (cosa che ALOHA non poteva fare, poiché usata in trasmissioni wireless e quindi non facile rilevare immediatamente le collisioni a differenza del cavo).

Ma se si partiva dal presupposto che gli altri non trasmettono se io già sto trasmettendo e viceversa, perché mai potrebbero verificarsi delle collisioni? Per via del ritardo di propagazione: all’istante t0 un nodo trasmette, ma il suo segnale impiega del tempo per arrivare: gli altri nodi quindi potrebbero quindi in certe occasioni rilevare il canale inattivo quando in realtà non è così.

Quando mi accorgo della collisione prima di interrompermi invio un segnale di disturbo per assicurarmi che la collisione venga percepita da tutti.

Vediamo più nel dettaglio l’algoritmo CSMA/CD di Ethernet:

1- Ethernet riceve un datagramma dal livello di rete, crea quindi un frame

2- se Ethernet percepisce il canale:

-**inutilizzato**: avvia la trasmissione del frame;

-**occupato**: aspetta finché il canale è libero, poi trasmette;

3- se l’intero frame viene trasmesso senza collisioni allora o.k.

4- se durante l’invio viene rilevata un’altra trasmissione (collisione) allora si interrompe immediatamente e invia il segnale di disturbo (**Jam**)

5- inizia una pausa, ma quanto dura? Si entra nella **binary exponential backoff**: il tempo massimo che posso aspettare dipende da quante collisioni ho percepito. Dopo la m-esima collisione, si sceglie casualmente K tra {0,1,... 2^m-1 } e si aspetta un tempo pari al tempo di trasmissione di 512 bit per K, si torna poi allo step 2. (chiaramente maggiori collisioni == maggiore intervallo di backoff)

Il valore di m nell'algoritmo appena descritto viene comunque limitato a 10.

Efficienza CSMA/CD

- d_{prop} = massimo ritardo di propagazione tra due schede di rete
- d_{trasm} = tempo necessario per trasmettere un frame di dimensione massima

$$\text{efficienza} = \frac{1}{1 + 5d_{prop}/d_{trasm}}$$

- l'efficienza tende a 1
 - se t_{prop} tende a 0 (perché la trasmissione viene interrotta subito in presenza di collisioni, evitando sprechi)
 - se t_{trans} tende a infinito (perché un frame, appropriatosi del canale, lo impegnava a lungo)
- prestazioni migliori di ALOHA: e semplice, economico, decentralizzato!

La frazione $5d_{prop}/d_{trasm}$ si spiega come segue: se ho tanto da trasmettere trasmetto per più tempo quindi tempo di trasmissione d_{trasm} maggiore e il valore della frazione diminuisce, se invece ho ritardo di propagazione la frazione aumenta (del resto ho collisione) per cui l'efficienza diminuisce. L'efficienza tende quindi a 1 se d_{prop} tende a 0 o se d_{trasm} tende a infinito, **si tratta chiaramente di un protocollo migliore di ALOHA (ma non applicabile al wireless, solo filo! Non posso come detto rilevare con tanta facilità in wireless collisioni)**

Tirando le somme:

- **protocolli a suddivisione del canale sono efficienti ed equi soltanto quando si ha un carico elevato**, per cui tutti trasmettono continuamente. Sono infatti inefficienti a basso carico!
- **protocolli ad accesso casuale: efficienti a basso carico, un singolo nodo pulisce il canale completamente, ma ad alto carico overhead di collisione.**

Vediamo quindi finalmente l'ultima classe:

- **(3) Protocolli a Rotazione**: si cerca in qualche modo di prendere "il meglio" tra i protocolli a suddivisione del canale e ad accesso casuale!

Protocolli a rotazione: Polling

Con i protocolli di Polling *ho un controllore centralizzato che periodicamente invia a ciascun client un messaggio consentendogli di trasmettere per un tot di tempo*, per cui un client non trasmetterà finché non gli viene detto.

Il controllore può quindi attendere che ciascun client abbia esaurito il proprio turno oppure, in un approccio a rilevamento della portante, può decidere di lasciare spazio a un altro client non appena vede che l'altro non sta più trasmettendo.

Vi sono **diversi problemi** con questo approccio: anzitutto chiaramente tutto questo **traffico di controllo spreca banda**, inoltre si crea un ulteriore ritardo (**il ritardo di polling**) che rappresenta il tempo impiegato per notificare a un nodo il permesso di trasmettere. Ciò porta a un **throughput effettivo minore di R**. Inoltre avendo un controllore centralizzato, chiaramente quello rappresenta un **singolo punto di rottura** (se non funziona va a fanculo tutto il sistema)

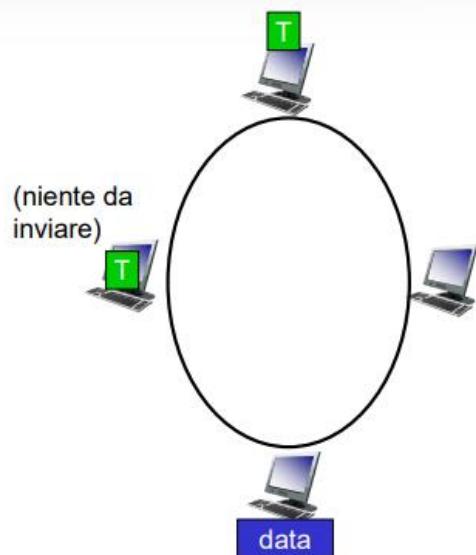
Il bluetooth usa Polling.

Token Passing

Si tratta di un protocollo a rotazione meno centralizzato rispetto al polling, immaginiamo che i client siano organizzati in maniera ciclica come in figura:

token passing:

- Messaggio di controllo deotto **token (gettone)** passato esplicitamente da un nodo al successivo, sequenzialmente
 - trasmette mentre possiede il token
- problemi:
 - overhead associato al token
 - latenza
 - singolo punto di rottura (token)
- Usato in: FDDI e token ring (IEEE 802.5)

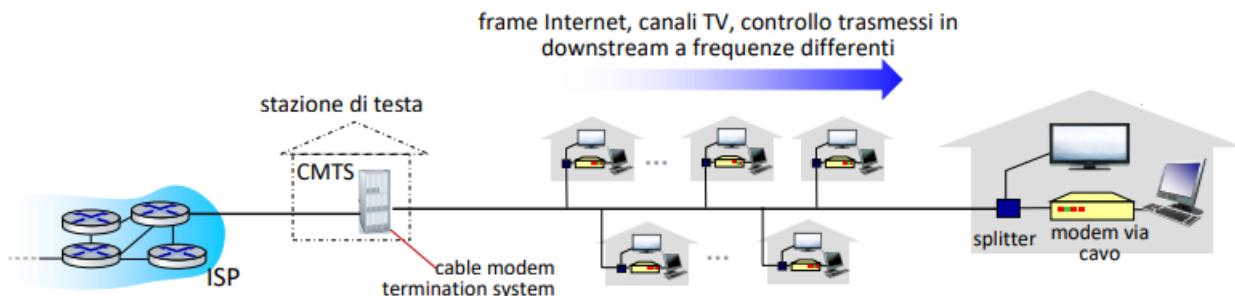


I client si passano in maniera periodica un token, che dà diritto di all'host che l'ha ricevuto di trasmettere. (ancora problemi di latenza e singolo punto di rottura legato al concetto di token, se non ce sta non se trasmette)

NB. Se ho un canale broadcast a singolo trasmittente chiaramente non ho bisogno di un protocollo ad accesso multiplo

Rete di accesso via cavo

(FDM, TDM, allocazione centralizzata e acc. casuale)

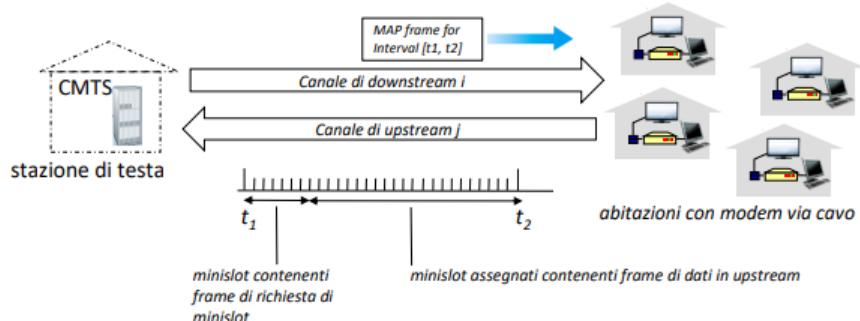


(quella tipica usata dagli americani che avevamo visto all'inizio)

Combina tutte le tecnologie che abbiamo visto:

- usa **FDM** per suddividere la banda del cavo coassiale tra canali TV, canali dati, di controllo in **downstream**. Essendo soltanto il CMTS a trasmettere in questa direzione, non ho problemi di accesso multiplo!
- in **upstream** entra in gioco il problema **dell'accesso multiplo** (più utenti devono trasmettere roba). Tutti gli utenti si contendono (secondo un algoritmo di accesso casuale, come quelli visti prima tipo CSMA) determinati slot temporali del canale upstream

Rete di accesso via cavo;



DOCSIS: specifiche di interfaccia del servizio dati via cavo

- FDM su canali di frequenze upstream e downstream
- TDM upstream: alcuni slot assegnati, altri sono contesi
 - Frame MAP in downstream: assegna i minislots in upstream
 - Richieste di frame in upstream (e dati) trasmessi con accesso casuale (binary backoff) in slot selezionati

Abbiamo visto l'altra volta tra le altre cose i protocolli ad accesso multiplo, legati a situazioni di broadcast dove più dispositivi trasmettono contemporaneamente ed è quindi necessario l'intervento di un protocollo che li gestica al fine di evitare collisioni e quindi dati corrotti.

Tre categorie: a suddivisione del canale (efficienti a carichi elevati, inefficienti se ho poco da trasmettere), ad accesso casuale (l'opposto di prima), a rotazione (svantaggio di avere spesso un singolo punto di rottura e più basso throughput a causa di un approccio centralizzato con un controllore).

LAN

Le **LAN** (**Local Area Network**, Reti di Area Locale) sono così chiamate perché sono reti che coprono un'area limitata come un'abitazione, una scuola, un ufficio o un edificio.

Esistono due tecnologie principali a cui la LAN fa affidamento:

- Ethernet (tecnologia che è usata anche in altri ambiti)
- WiFi

Abbiamo visto che, nel momento in cui dobbiamo dialogare con un'entità remota, abbiamo bisogno di indirizzarla e abbiamo visto che per farlo, man mano che si scende nel livello della rete, si aggiungono delle informazioni (es. numeri di porta a livello di trasporto per distinguere i processi, indirizzo IP usati a livello di rete per l'inoltro...).

A livello di collegamento per dialogare con altri nodi è necessario un indirizzo ulteriormente specifico: l'indirizzo MAC (Media Access Control, anche detti indirizzi fisici, o indirizzi LAN, o indirizzi Ethernet).

Ma se ho già gli indirizzi IP, perché mai me ne servirebbero di altri?

Si tratta anzitutto di una questione di indipendenza dei livelli, che permette di avere un livello di collegamento sul quale poter far girare diversi protocolli a livello di rete. Inoltre gli indirizzi MAC hanno vantaggi in termini di portabilità che non abbiamo con IP.

Strutturalmente e funzionalmente in cosa differiscono gli indirizzi IP e gli indirizzi MAC?

Gli **indirizzi IP** sono indirizzi a livello di rete (definiti dal protocollo IP), nella versione IPv4 sono a 32 bit, in IPv6 a 128 (maggiore per aumentare lo spazio di indirizzi che su Internet si stava esaurendo). Ancora IPv4 in giro, una soluzione che si era adottata per evitare l'utilizzo di troppi indirizzi è NAT (usa indirizzi IP privati visibili solo all'interno della rete locale (quella che vediamo oggi) e tradotti nel momento in cui si dialoga con l'esterno).

Gli **indirizzi MAC** sono usati al livello di collegamento per trasferire datagrammi da un nodo a uno fisicamente adiacente lungo un collegamento.

Ciò che dal punto di vista del livello di rete è una sottorete, in realtà può essere vista come una rete di livello 2 (quindi ad esempio una rete locale!).

Un indirizzo MAC presenta tipicamente **48 bit**:

es. 1A-2F-BB-76-09-AD

Gli indirizzi MAC sono assegnati alle **interfacce di rete**, che in realtà hanno due indirizzi: indirizzo MAC a livello di collegamento e indirizzo IP a livello di rete. La LAN (che in generale è una rete di livello 2) corrisponde a una sottorete agli occhi del livello 3!

Ciò che sappiamo avere in comune gli indirizzi di una stessa sottorete è il loro prefisso IP,(137.196.7/24 in questo caso ad esempio i primi 24 bit a sx identificano la LAN (ricorda che se il prefisso non è un multiplo di 8 di fare attenzione se chiede se un indirizzo IP appartiene a quel prefisso)).

Come nel caso degli indirizzi IP, ho il problema di dover allocare questi indirizzi MAC. A differenza degli indirizzi IP che possono anche essere dinamici, **gli indirizzi MAC sono associati Read-Only ad ogni adattatore dal suo produttore, che a sua volta ottiene un blocco dall'IEEE** (ente di standardizzazione che gestisce anche gli indirizzi MAC).

Quindi indirizzi MAC e IP hanno chiaramente caratteristiche differenti; *IP somiglia più a un indirizzo postale, con una sua struttura che cambia nel momento in cui mi sposto* (univoco nel momento in cui è assegnato, ma può cambiare ad esempio se cambio sottorete).

MAC address invece funziona più come fosse un codice fiscale, qualcosa che la scheda di rete si porta dietro dalla nascita senza cambiare mai.

In realtà è possibile cambiare MACaddress anche via software per questioni ad esempio di **privacy** (essendo costante e unico rappresenta una sorta di cookie, una rete a cui mi collego potrebbe usare il mio MACaddress per riconoscermi univocamente e seguire i miei percorsi di rete).

Proprio per la sua natura di “codice fiscale”, **la struttura di MAC è piatta**, non vi sono parti che servono a localizzare il nodo nella rete. Ciò garantisce la **portabilità**: è possibile spostare un’interfaccia da una LAN a un’altra (mentre con IP non era così semplice, dipendeva dalla sottorete).

Sorge una domanda, quando mando un datagramma IP indico come destinatario un indirizzo IP. Nel momento in cui lo invio sappiamo però che si deve scendere al livello di collegamento, incapsulando il datagramma in un frame con destinazione il nodo adiacente allo stesso collegamento. In particolare la destinazione, dal punto di vista del livello 2, sarà l’indirizzo MAC del destinatario, non quello IP. *Come faccio a tradurre l’indirizzo IP in MAC?* Esiste un protocollo per la risoluzione degli indirizzi, **il protocollo ARP**.

ARP

L’idea è che *ogni nodo IP* (cioè host o router) *presente sulla LAN ha memorizzato una Tabella ARP* dove esiste la **corrispondenza tra indirizzi IP e MAC** per alcuni dei nodi sulla LAN e un **TTL** (timetolive) per cui dopo un po’ di tempo la mappatura di questi indirizzi sarà dimenticata (in genere 20 min da quando la voce è stata inserita in tabella).

Ma come popolare questa tabella? Ci pensa il protocollo.

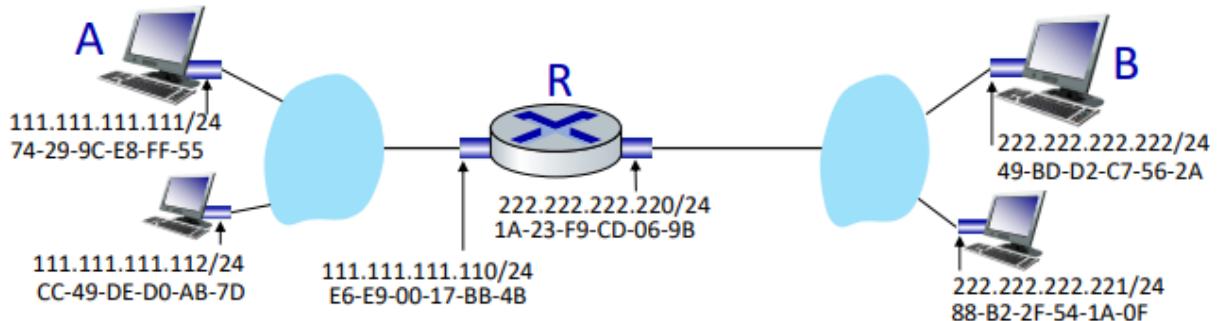
Quando un nodo A vuole scoprire l’indirizzo MAC di un nodo B prepara una **richiesta ARP** (che *contiene MAC e IP sorgente e IP destinatario*) e la inserisce in un Frame il cui indirizzo di destinazione chiaramente non può essere quello in B (non lo conosce) ma **l’indirizzo di broadcast a livello 2** (tutti 1, FF-FF-FF-FF-FF-FF).

Tutti i nodi della rete livello 2 (sottorete) riceveranno questo frame, che sarà passato a livello superiore. Se un nodo vede che l’indirizzo IP della richiesta ARP e il suo indirizzo IP non coincidono ignora, altrimenti se coincidono prepara una **risposta** in cui aggiunge il proprio MACaddress incapsulandolo in un Frame destinato al nodo che aveva fatto la richiesta ARP (perché grazie alla richiesta conosce anche il MACaddress di quel nodo!). A questo punto A riceve la risposta di B e salva il suo MAC nella tabella ARP locale.

ARP Spoofing o ARP Poisoning

- Un attaccante invia in una LAN risposte ARP contraffatte, inducendo l'associazione di un indirizzo IP a un certo indirizzo MAC
- Il protocollo ARP è senza stato e un nodo (host o router) aggiorna la propria ARP appena viene ricevuta una risposta ARP (a prescindere che questa faccia seguito a una effettiva richiesta)
- Alcuni "usi":
 - *denial-of-service* (DoS): associare diversi indirizzi IP allo stesso indirizzo MAC per sovraccaricarlo di traffico
 - *man-in-the-middle* (MITM): l'attaccante associa il proprio indirizzo MAC all'indirizzo IP di un altro nodo, in modo da intercettare (e magari modificare) il traffico destinato a quest'ultimo, per poi re-inoltrarglielo

Ma che succede se volessi inviare un datagramma al di fuori della mia sottorete?



Anzitutto, il nodo A conosce soltanto l'indirizzo IP del destinatario, non sa quanti bit stanno nel prefisso direttamente per identificare la sottorete.

Tuttavia A sa di appartenere a una sottorete /24, pertanto confronta i 24 bit più significativi del proprio indirizzo con quello dell'indirizzo di B, constatando che sono diversi e quindi capisce che B si trova in una sottorete differente (se lo chiede all'esame è da spiegare così).

Ciò che metterà A come destinazione a livello 2 è l'indirizzo MAC della scheda di rete del router di primo hop nella propria sottorete (R), di cui conosce l'indirizzo IP grazie a **DHCP** (protocollo di gestione della rete che assegna automaticamente indirizzi IP e altri parametri ai dispositivi connessi a una determinata rete).

DHCP però ci fornisce IP address, non MAC. Per risolvere la cosa si usa **ARP** secondo il meccanismo descritto prima.

Quando a R arriva il frame destinato a B lo porta a livello di rete e si accorge che non è il destinatario, ma in quanto router sa che deve eseguire l'inoltro e quindi procede. *In questo caso la seconda interfaccia di R è direttamente connessa alla sottorete in cui si trova B*: ciò che succede quindi è che il frame dovrà essere inoltrato direttamente a B passando per switch di livello 2. Quindi R identifica l'indirizzo MAC di B secondo il meccanismo di ARP (broadcast etc...) e, una volta ottenuto, manda il frame. Quando a B arriva finalmente il frame, lo passa in alto al livello di rete e fine.

Ethernet

Ma da cosa sono costituite nel pratico le LAN? Esistono varie tecnologie, ma tra quelle oggi dominanti c'è Ethernet.

Si tratta di una tecnologia ampiamente utilizzata, **semplice ed economica**, il cui successo deriva dal fatto che **è rimasta al passo relativamente la velocità** (dai 10 Mbps ai 400 Gbps!).

L'inventore di questa tecnologia è Bob Metcalfe, gli è valso il premio Turing.

Ethernet è stato ispirato dalle idee precedenti del protocollo ALOHA.

La prima idea di Ethernet risale agli anni '70/'80, con approccio a **bus**. Si aveva a tutti gli effetti un cavo coassiale che attraversava tutte le schede di rete. Tagliando un filo, si taglia la rete.

Successivamente si è passati, dalla metà degli anni '90 fino al 2000, alla **topologia a stella con hub**. L'hub è un dispositivo di livello fisico che si occupa di rigenerare i segnali ricevuti su un'interfaccia per ritrasmetterli su tutte le altre interfacce. Il **problema** di entrambi questi approcci è che **tutti i nodi sono nello stesso dominio di collisione**, per cui se due nodi trasmettono contemporaneamente avrà collisione!

- **bus**: popolare fino alla metà degli anni '90
 - tutti i nodi sono nello stesso dominio di collisione (possono collidere tra loro)
- **topologia a stella con hub**: popolare fino agli anni 2000
 - i nodi sono interconnessi da un hub (dispositivo a livello fisico che rigenera i segnali ricevuti su una interfaccia e li ritrasmette su tutte le altre interfacce), pertanto tutti i nodi sono nello stesso dominio di collisione
- **commutata (switched)**: oggi prevalente
 - **switch** di livello 2 attivo al centro
 - ogni "spoke" esegue un protocollo Ethernet (separato) (i nodi non si scontrano tra loro)



L'ultimo approccio, che è quello ancora oggi prevalente, è quello con **topologia commutata (switched)**. In questo approccio è presente uno **switch** tra i nodi che "governa" il traffico: se il nodo in alto deve mandare un frame al nodo a destra lo switch si occuperà di inoltrarlo e lo manderà solo se è libero il collegamento, altrimenti lo bufferizza (supporta trasmissioni simultanee tra nodi differenti!). Quindi gli switch sono dispositivi di livello 2 che operano in modalità store and forward secondo i principi che abbiamo già visto.

***: a che livello avviene il multiplexing e demultiplexing? Ho multiplexing e demultiplexing quando in un livello voglio seguire più entità a livello superiore. Quindi **non solo a livello di trasporto, anche a livello di collegamento** potrei voler seguire più livelli di rete (o ancora, a **livello di rete** potrei voler seguire più livelli di trasporto), e per capire a quale sto facendo riferimento uso proprio i 6 byte tipo (vedi dopo).

Vediamo ora nel dettaglio la struttura del Frame Ethernet.

l'interfaccia trasmittente incapsula il datagramma IP (o altro pacchetto di protocolli di livello di rete) in frame Ethernet



Abbiamo un **preamble** formato da una sequenza di **7 byte 10101010** (*necessari a "risvegliare" la scheda di rete del ricevente e sincronizzare il suo clock con quello del trasmittente*) seguiti da un byte **10101011** (*che serve ad informare il ricevente dell'inizio del frame vero e proprio*).

Successivamente troviamo **6 byte** dedicati agli **indirizzi di destinazione e sorgente** (per eventuale risposta). In particolare l'adattatore passa il frame a

livello di rete sse l'indirizzo di destinazione è il suo o di broadcast (es. per un pacchetto ARP).

Tipo: sono 2 byte dedicati ad indicare un protocollo di livello superiore (principalmente IP, ma sono possibili anche altri). Serve per demultiplexare (***) sul ricevitore.

Si hanno poi i **dati (payload)** effettivi e infine **CRC**, l'EDC di controllo di ridondanza ciclica presso il ricevitore a cui vengono dedicati **4 byte**. Se l'errore è rilevato, il frame viene scartato.

Vediamo ora le **caratteristiche principali del protocollo Ethernet**.

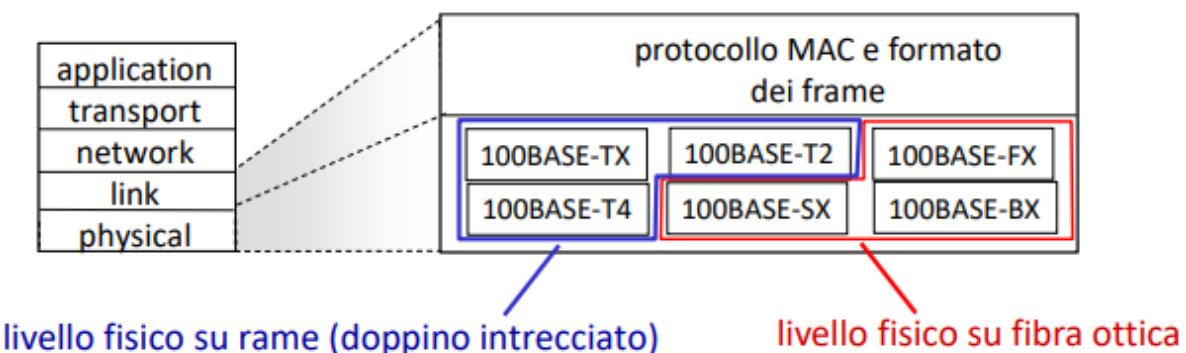
Si tratta di un protocollo **senza connessione**, cioè *privi di handshake* (che serve per essere sicuri che entrambi possano comunicare e settare dei parametri e inizializzare gli stati per la comunicazione affidabile) all'inizio della comunicazione tra le **NIC** (network interface card, scheda di rete) del mittente e destinatario.

Ethernet è inoltre un **protocollo non affidabile**: la NIC ricevente non invia ACK o NAK alla NIC mittente.

Ethernet inoltre utilizza come **protocollo MAC** il **CSMA/CD con binary backoff** (ne abbiamo parlato la scorsa lezione).

Ethernet non rappresenta un solo standard.

Ne esistono molti e diversi, ciò che li **acomuna** sono il **protocollo MAC** e il **formato dei Frame**, mentre ciò che li **differenzia** sono le **velocità** (da 2 Mbps a 80 Gbps) e i **mezzi di trasmissione** (cavo coassiale, doppino, fibra...)



100 sta per 100 Mbps, BASE trasmissione di byte a base (si utilizza un mezzo fisico), T sta per cavo intrecciato (TX, T4, T2 sono varianti sulla base del tipo effettivo di doppino).

Invece quelli che finiscono con X in generale stanno per trasmissione su fibra ottica.

Esiste un limite nella lunghezza dei fili, legata 1) alla capacità del mezzo di trasmettere segnale (man mano che aumenta la distanza il segnale si attenua) e 2) collisione.

Il limite di un cavo in rame di ethernet è tipicamente di 100 metri.

Switch

Lo Switch è un **commutatore di pacchetto di livello 2**.

Esso ha un **ruolo attivo** in quanto **memorizza e inoltra** (via store-and-forward) frame Ethernet (o di altro tipo) **filtrandoli** (esamina l'indirizzo MAC di destinazione del frame in arrivo e inoltra selettivamente il frame in uno o più collegamenti di uscita)

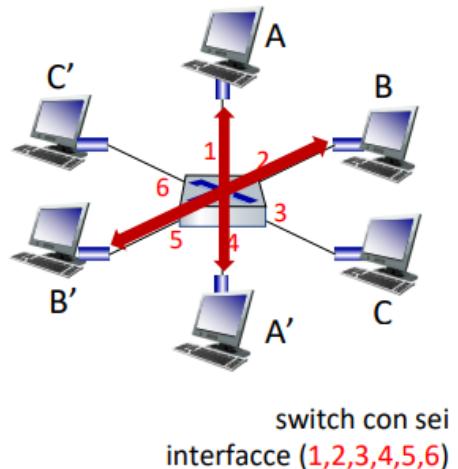
È un **dispositivo trasparente**: gli host sono inconsapevoli della presenza degli switch ed inoltre **gli switch non hanno indirizzi MAC** associati, o comunque se li hanno non hanno ruolo nella commutazione (la scheda di rete dello switch non sarà mai vista come destinatario nel percorso del pacchetto).

Poiché non determinano una giunzione elettrica tra i vari collegamenti, gli switch **supportano collegamenti eterogenei**: i collegamenti agli switch possono operare a velocità diverse e usare mezzi trasmissivi diversi.

Sono dispositivi **plug and play**, basati su **autoapprendimento** (non è necessario configurarli)

Dal momento che nodi diversi vogliono dialogare possono farlo senza collisioni dal punto di vista del protocollo CSMA/CD. Nel momento in cui due nodi vogliono parlare con lo stesso nodo lo switch mette in coda i frame, in modo da garantire che non vi sia collisione.

- gli host hanno connessioni dedicate, dirette con lo switch
- lo switch "bufferizza" i pacchetti
- il protocollo Ethernet è utilizzato su *ciascun* collegamento, così:
 - full-duplex: una singola coppia di nodi alle estremità del collegamento che possono trasmettere simultaneamente senza collisioni (es. perché i segnali viaggiano su fili dedicati nel cavo Ethernet), no CSMA/CD
 - half-duplex: il singolo collegamento half duplex è un dominio di collisione a sé
- **switching:** A-to-A' e B-to-B' possono trasmettere simultaneamente senza collisioni



Gli switch eliminano le collisioni, **l'unica collisione possibile è quella che si ha tra il mittente e lo switch stesso** (si mandano robe contemporaneamente a vicenda).

- ma A-to-A' e C-to-A' *non* possono accadere simultaneamente

Ciò avviene chiaramente solo se entrambi operano in modalità **half-duplex** (per def. di **full-duplex** infatti invece si può comunicare lungo il canale in modo bidirezionale senza problemi, per quello *in quel caso non serve CSMA/CD*, mentre in half-duplex non è possibile). Se invece operano appunto in half-duplex potrebbero esserci collisioni, quindi *si applica il protocollo CSMA/CD indipendentemente per ogni collegamento di questo tipo*.

L'ideale è organizzare il tutto con collegamenti full-duplex, che sono sempre realizzabili punto-punto, cioè collegamenti diretti (nel nostro caso ci interessano i collegamenti nodo-switch).

Ricapitolando: *in una rete commutata full duplex non ho mai collisioni e quindi CSMA/CD non serve.*

Ma come fa (in figura) lo switch a sapere che A' è raggiungibile tramite l'interfaccia 4 e B' tramite la 5?

Ciascuno degli switch ha una tabella di commutazione (switch table) in cui

ogni voce riporta l'indirizzo MAC del nodo, l'interfaccia che conduce a quel nodo e il time stamp TTL.

Chiaramente ciò ci ricorda le tabelle di inoltro dei router, ma come vengono create e mantenute in questo caso le voci nella tabella?

Abbiamo anticipato come gli switch siano dispositivi plug and play basati sull'autoapprendimento.

Immaginiamo che A voglia inviare un messaggio ad A' (per semplicità A e A' MACaddress). Il pacchetto arriva allo switch, lo switch avrà imparato che A sta sul collegamento 1 e metterà in corrispondenza anche un timetolive TTL (*necessario perché i collegamenti potrebbero venire aggiornati*).

Ma come fa ad aggiungere ora alla tabella A', che non lo conosce?

Allora applica il **flood**, cioè **inoltra su tutte le interfacce eccetto quelle che già conosce** (in pratica si tratta di broadcast ma, poiché prevede questa particolarità, è rinominato flood).

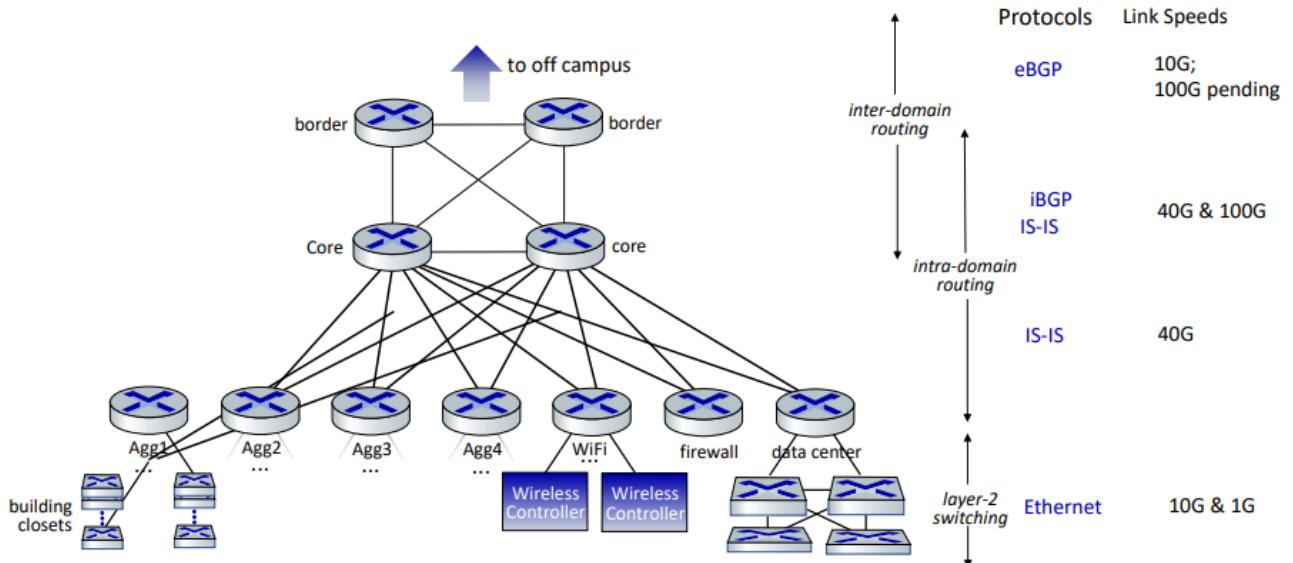
Se invece lo switch aveva già la destinazione salvata nella sua tabella di commutazione allora **invia il frame su quel collegamento selettivamente** (filtraggio), ovvero lo manda lungo quel collegamento soltanto se il frame non è stato ricevuto proprio a partire da quel collegamento (in tal caso destinatario e mittente coincidono, è da riaggiornare la tabella, **il frame viene scartato**).

Quando uno switch riceve un frame:

1. registra il collegamento in ingresso e l'indirizzo MAC dell'host mittente
2. indica la tabella degli switch utilizzando l'indirizzo MAC di destinazione
3. se viene trovata una voce per la destinazione
allora {
 - se la destinazione è sul segmento dal quale è arrivato il frame
 - allora scarta il frame
 - altrimenti inoltra il frame sull'interfaccia indicata dalla voce}
- altrimenti flood /* inoltra su tutte le interfacce eccetto quella di arrivo; in altre parole, manda il frame in broadcast (ma non cambia l'indirizzo MAC di destinazione) */

Link Layer: 6-84

Rete di un'Università:



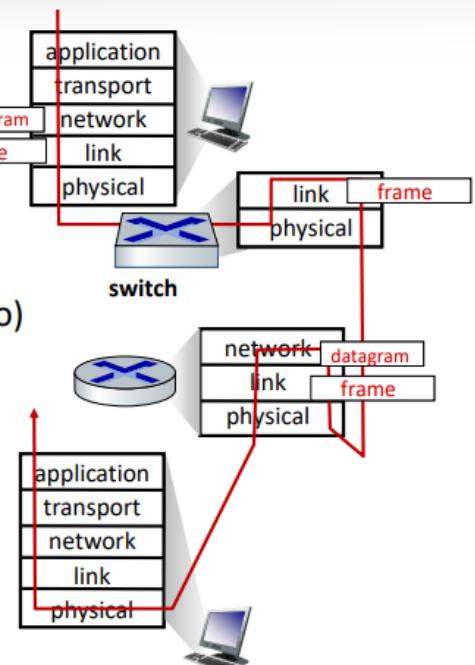
Switch e router a confronto

Entrambi lavorano in store-and-forward:

- **router:** dispositivi a livello di rete (esaminano l'intestazione a livello di rete)
- **switch:** dispositivi a livello di collegamento (esaminano l'intestazione a livello di collegamento)

Entrambi hanno tabelle di inoltro:

- **router:** calcolano le tabelle usando algoritmi di instradamento, indirizzi IP
- **switch:** autoapprendimento della tabella di inoltro usando il flooding, indirizzi MAC



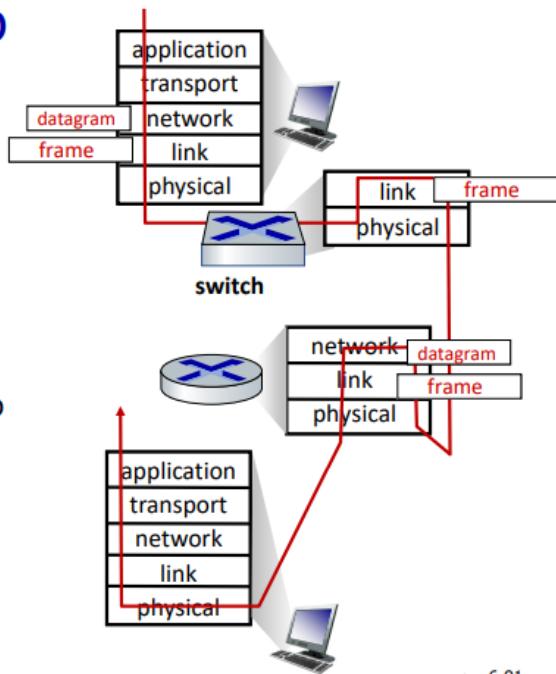
Switch e router a confronto

Topologia della rete:

- **router**: gli algoritmi di instradamento possono trovare percorsi ottimali (senza cicli) nonostante cicli nella topologia della rete; inoltre, il decremento del TTL farebbe scartare i pacchetti incastriati in potenziali instradamenti ciclici (es. dovuti a errori di configurazione)
- **switch**: gli switch devono essere interconnessi a albero (anche solo logicamente, grazie al protocollo *Spanning Tree Protocol*), per evitare che il traffico broadcast (in assenza di un campo TTL nei frame) resti in circolazione potenzialmente per sempre

Numero di nodi

- **router**: instradamento gerarchico, aggregazione degli indirizzi, etc...
- **switch**: tabelle ARP molto grandi nei nodi, ingente traffico ARP, frame broadcast, etc...



Link Layer 6-91

Isolamento del traffico

- gli **switch** inviano in broadcast i frame il cui indirizzo MAC di destinazione è sconosciuto, con un effetto a valanga in presenza di molteplici switch interconnessi. I frame broadcast sono inoltrati a tutti i nodi nella rete.
- i **router** inoltrano i pacchetti in accordo a percorsi determinati dalla funzione di instradamento.

VLAN

La VLAN sta per Virtual LAN.

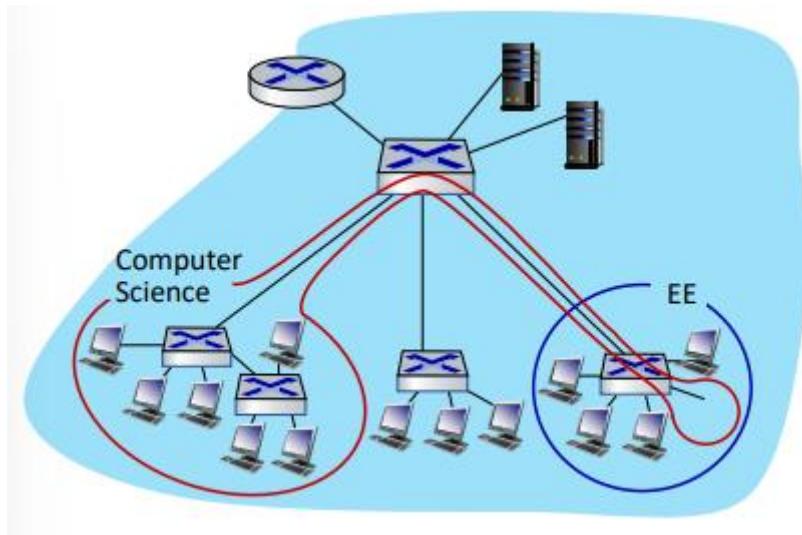
Abbiamo visto come con Ethernet è possibile implementare delle reti di livello 2 (che corrispondono alle sottoreti dal punto di vista del livello 3, sottoreti Internet).

Una domanda che potrebbe sorgere spontanea è: perché a questo punto utilizzare i router? Non potrei creare un'unica grande rete con solo switch?

Il problema è che Ethernet e in generale le reti di livello 2 **non sono pensate per essere scalabili**, la prima cosa che ci suggerisce ciò è che le reti di livello 2 si basano molto sul concetto di diffusione lungo un singolo dominio di broadcast (la rete di livello 2 in questione), che deve essere limitata nel numero di nodi!

Ciò inoltre può comportare **problemi di sicurezza e privacy** (es. inviare messaggi per cui le destinazioni nelle tabelle di commutazione non corrispondono, malintenzionati che possono sniffare i pacchetti).

Si hanno inoltre **problemi amministrativi**. Immaginiamo che un utente di Computer Science si sposti nell'ufficio EE (connesso **fisicamente** allo switch EE) ma voglia rimanere connesso **logicamente** allo switch CS. Come fare? Dovrei allungare il filo...



Esiste un'alternativa usando le così dette VLAN (Virtual Local Area Network). L'idea è che **posso usare le VLAN per poter definire più LAN virtuali su un'unica infrastruttura LAN fisica**.

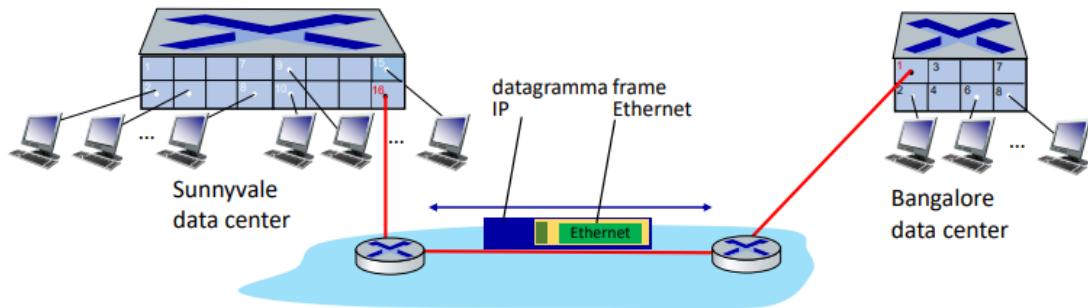
Quando si parla di **port-based VLAN** la storia è molto semplice e intuitiva: se uno switch ha ad esempio 16 porte, posso assegnarle in modo dinamico tra le varie VLAN: dal punto di vista logico sarà come ad esempio se avessi due switch; le prime 8 porte sono per uno switch e le altre 8 per un altro switch.

Come fare poi a farli parlare tra loro? Tramite un **routing** implementato nello switch originale! In pratica i produttori combinano gli switch con i router. In questo modo l'inoltro tra reti di livello 2 non sarà più supportato solo dal livello 2 ma sarà supportato con il livello di rete.

Una **porta trunk** trasporta frame tra VLAN definite su più switch fisici. Per capire di che VLAN sto parlando quando mando il frame questo deve contenere più informazioni legate proprio all'ID VLAN.

Il frame ethernet è “allungato” aggiungendo 2 byte di tag **Protocol Identifier**, un tag con **Informazioni di Controllo**.

EVPN: Ethernet VPN (altrimenti note come VXLAN)



Switch Ethernet di livello 2 connessi *logicamente* l'un l'altro (es., usando IP come *underlay*)

- frame Ethernet trasportati *dentro* a datagrammi IP tra siti
- “schema di *tunneling* per sovrapporre reti Layer 2 a reti Layer 3 ... funziona sull'infrastruttura di rete esistente e fornisce un mezzo per “allungare” una rete Layer 2”. [RFC 7348]

(non s’è capito un cazzo con st’ultima parte)

Lez 23

Wireless e Reti Mobili

Ci interessano perché si tratta di tecnologie assai prevalenti (per ogni dispositivo abbonato alla telefonia fissa 10 abbonati a quella mobile nel 2019).

Wireless e reti mobili non rappresentano sinonimi:

- **Wireless** == *comunicazione su collegamento wireless*, si intende proprio come mezzo tramite il quale far passare la comunicazione (mezzo non guidato, onde trasmissive);
- **Mobilità** == si intende *un dispositivo che si sposta* (eventualmente connesso alla rete tramite wireless). L'enfasi non è sul collegamento wireless, quanto sulle problematiche determinate dal fatto che l'utente cambia il proprio punto di aggancio alla rete.

(es. se mi connetto a un access point a casa mia con un certo indirizzo IP e poi a un access point all'università con diverso IP, scenario molto più complesso rispetto a spostarmi sempre sotto la copertura dello stesso access point)

Le componenti di una rete wireless sono:

- **Host** == come sappiamo chiamati così perché ospitano (cioè eseguono) le applicazioni di rete. *Non tutti gli host wireless sono uguali in termini di mobilità* (es. pc fisso (stazionario) vs smartphone (mobile)).
- **Collegamento Wireless** == *collegamento senza filo usato per connettere gli host*

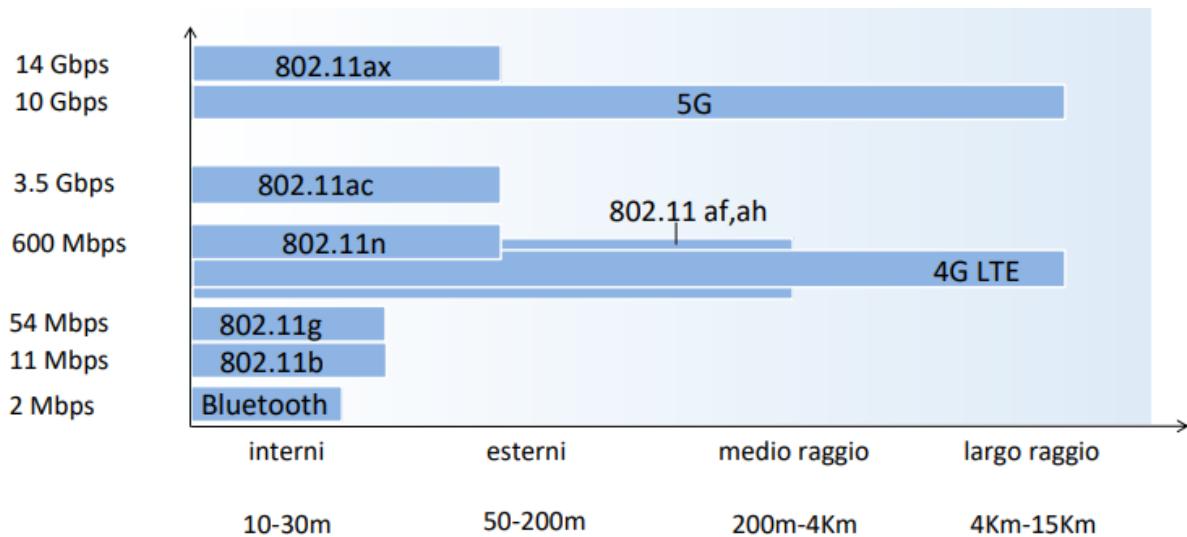
host wireless alla stazione base o per connettere host ad altri host (a seconda della struttura di rete che abbiamo). Il mezzo trasmittivo Wireless è chiaramente un **canale broadcast** (il segnale inviato è propagato secondo onde elettromagnetiche), si pone quindi il problema dell'**accesso multiplo** (più dispositivi trasmettono in contemporanea, collisioni). Avremo quindi bisogno di un protocollo **MAC** (Medium Access Control).

Non tutti i collegamenti Wireless sono identici, *cambiano tra loro tassi trasmisivi (velocità) e capacità di copertura.*

- **Stazione Base** == si tratta di un dispositivo tipo l'access point di casa o la torre della rete cellulare il cui compito principale è quello di *fungere da ripetitore (relay)* dei pacchetti dagli host connessi al resto della rete e viceversa.

(es. se voglio mandare un pacchetto fuori dalla mia rete locale lo mando anzitutto alla stazione base, che poi lo inoltrerà al resto della rete magari attraverso un collegamento cablato).

In molte reti wireless infatti le stazioni di base sono connesse ad un'altra *infrastruttura più ampia*, tipicamente cablata.



(chiaramente qua velocità in termini di massime prestazioni possibili).

Il **Bluetooth** è inteso come la *più semplice tecnologia per il rimpiazzo di cavi*. È utilizzato per accessori come mouse, tastiera e altro ed è anche utilizzato per costruire le così dette personal area network (reti personali). Raggio limitato e velocità bassa.

Per raggi sempre limitati ma più grandi del bluetooth abbiamo il dominio del **Wi-Fi**, negli anni sono stati sviluppati diversi standard sotto lo stesso nome di **802.11** che differiscono per due dimensioni: copertura e velocità (entrambe aumentate gradualmente fino allo standard più recente 802.11ax con 14 Gbps e

copertura fino a 100 m circa.

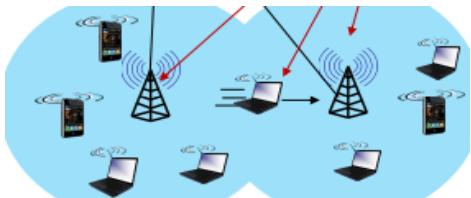
Altri standard Wi-Fi come af e ah usano frequenze particolari per estendere ulteriormente la copertura a discapito della velocità.

La **rete cellulare** 4G e 5G non sorprendentemente è quella a copertura più elevata.

Come possiamo costruire una rete wireless?

Esistono due principali modalità operative:

- **Modalità infrastruttura**: la rete wireless fa riferimento a un'infrastruttura di rete che contiene la stazione base. Questa infrastruttura fornisce i servizi di rete tipici come instradamento, indirizzamento etc... Quando si parla di questo tipo di reti wireless che presuppongono una stazione base non bisogna dimenticare il problema dell'**handoff**, per cui un host spostandosi passa dal raggio di copertura di una stazione base a quello di un'altra.



- **Modalità con reti ad hoc**: non ci sono stazioni base e gli host comunicano tra loro trasmettendo solo ad altri host entro la copertura del collegamento.

Quest'infrastruttura prevede che siano gli stessi host wireless a occuparsi dei servizi di rete come instradamento, traduzione nomi simil-DNS etc...

Tassonomia delle reti wireless

	hop singolo	hop multipli
infrastruttura (es., stazione base)	I host si collega a una stazione base, che lo collega al resto della rete: <i>Wi-Fi, rete cellulare</i>	c'è una stazione base collegata al resto della rete. Tuttavia, un host può dover ritrasmettere (<i>relay</i>) la propria comunicazione attraverso altri nodi wireless per comunicare con la stazione base: <i>alcune reti di sensori e Wi-Fi mesh</i>
nessuna infrastruttura	senza stazione base, uno dei nodi può coordinare la trasmissione degli altri: <i>Bluetooth</i>	nessuna stazione base, i pacchetti possono dover essere ritrasmessi attraverso diversi nodi wireless prima di giungere a destinazione: <i>mobile ad hoc networks (MANETs)</i> e <i>vehicular ad hoc network (VANET)</i>

Caratteristiche dei collegamenti Wireless

Vediamo le caratteristiche principali che differenziano il wireless dai collegamenti cablati.

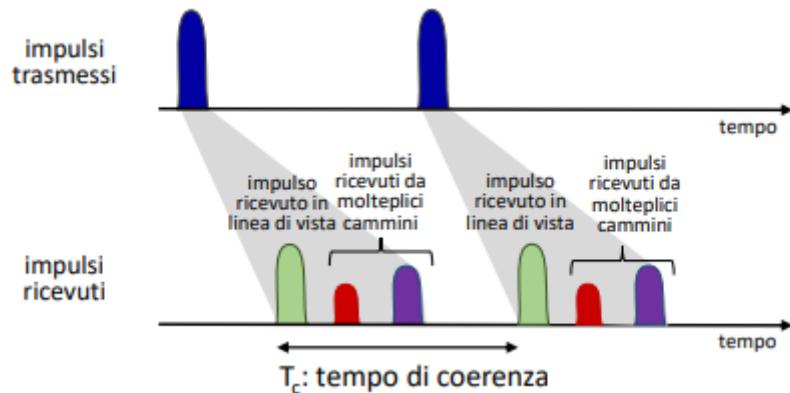
La prima che ci interessa è **l'attenuazione del segnale**. Questo fattore è anche presente nei collegamenti cablati, ma con il wireless è un problema molto maggiore e può essere causata principalmente da due fenomeni: *l'attraversamento di ostacoli* che assorbono parte della radiazione elettromagnetica e il attenuazione *di spazio libero* (l'onda si disperde con l'energia che si distribuisce).

In particolare *l'attenuazione di spazio libero è direttamente proporzionale a $(fd)^2$* con **f frequenza** e **d distanza**. Quindi maggiori la frequenza e la distanza maggiore l'attenuazione e del segnale.

Altro problema molto presente nei collegamenti wireless è la **Propagazione su cammini multipli** (*multipath propagation*): il segnale inviato in quanto onda elettromagnetica colpisce molti altri oggetti oltre agli host in sé, può quindi essere riflesso e in generale *arrivare alla stessa destinazione anche attraversando più percorsi e in tempistiche diverse*.



Il percorso in linea retta (**LOS**) sarà chiaramente il percorso più breve. Mandato il segnale questo sarà ricevuto dopo un certo tempo (ritardo di propagazione) e arriva però in diversi “picchi”: il primo è il segnale arrivato in linea di vista, poi una serie di altri picchi legati ai cammini alternativi.



Si pone quindi il *problema di capire come un impulso appartenga o meno a una certo bit di trasmissione o al successivo*.

Senza entrare nei dettagli l'**idea** è di introdurre un **Tempo di Coerenza**, cioè un intervallo in cui nel canale è presente il bit che vogliamo ricevere.

Ciò chiaramente influenza la massima velocità di trasmissione possibile, in quanto i tempi di coerenza non si possono sovrapporre e in quel tempo prestabilito posso trasmettere un solo bit (maggiori il tempo di coerenza, minore la velocità di trasmissione).

Altro problema molto comune nei canali Wireless è **l'interferenza**: accennato anche parlando di Ethernet ma nel Wireless problema molto più evidente.

Si hanno infatti due fonti principali di interferenze: una sono *altri dispositivi che trasmettono sulla stessa frequenza* (perché stanno competendo nello stesso standard per trasmettere o perché due standard diversi usano la stessa frequenza, es. 2.4 GHz usata da WiFi e Bluetooth).

Altra fonte di interferenze sono *altri dispositivi che emettono onde elettromagnetiche sotto forma di rumore elettromagnetico ambientale* (es. forni a microonde, motori).

Per evitare questo tipo di interferenze spesso la soluzione è cambiare banda (es. standard 802.11 recenti usano banda 5GHz).

Legata a questo discorso del rumore vi è il **Rapporto Segnale Rumore (SNR)**: si tratta del rapporto tra il segnale trasmesso che ho ricevuto (degradato da attenuazione, cammini multipli) e il rumore di fondo.

Senza entrare nei dettagli, un SNR più alto indica che il segnale ricevuto ha densità più grande del rumore di fondo. Ciò significa che si tratta di un *buon segnale*, poiché non è eccessivamente compromesso dal rumore, per cui posso comprendere i dati che mi sono trasmessi.

Maggiore SNR -> maggiore capacità di estrarre il segnale da quanto ricevuto discriminandolo dal rumore -> minore **Tasso di Errore sui Bit** (Bit Error Rate **BER**).

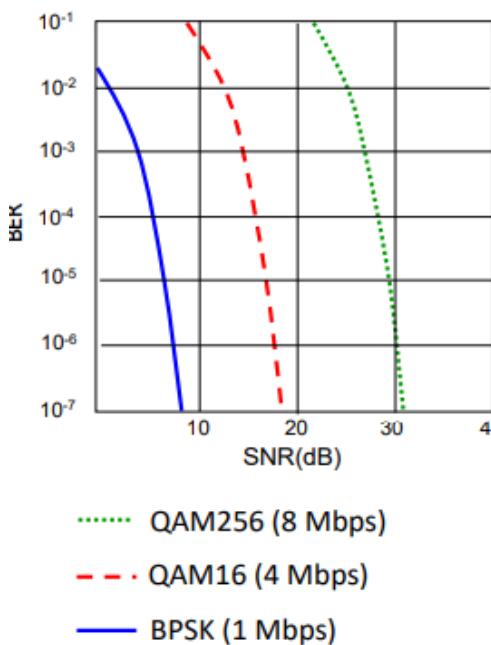
Bilanciamento di SNR e BER:

per un dato schema di modulazione (cioè per un dato approccio nel codificare i dati come segnale): **aumentare la potenza significa chiaramente aumentare SRN e quindi diminuire BER.**

Tuttavia non è così semplice. Oltre a questioni di salute, **trasmissioni a potenza maggiore implicano un maggiore consumo di energia** (critico per dispositivi a batteria) e **possono andare a interferire con altre trasmissioni.**

Per un dato SNR, **una tecnica di modulazione con più elevato tasso di trasmissione dei bit comporterà un BER più alto.** (nella figura sotto: trasmetto a un certo SNR, la curva rappresenta il tasso di trasmissione, se faccio potenza SNR 10 dB allora interseco la curva di tasso trasmisivo 4 Mbps a 10^{-1} BER).

L'SNR può cambiare con la mobilità e adattarsi dinamicamente (es. se si alza il tasso trasmisivo mi tocca alzare SNR o viceversa se si alza SNR posso alzare il tasso trasmisivo, lo stesso ma al contrario se abbasso).



Ultimo problema caratteristico dei collegamenti wireless è quello del **Terminale Nascosto**.

Dati tre terminali ABC, dove A e B possono parlare e B e C possono parlare mentre A e C non possono parlare, A e C possono causare (senza saperlo)

interferenza presso la destinazione B.

Rappresenta un problema perché in Wireless non è semplice rilevare le collisioni quanto lo è ad esempio con Ethernet! (in questo scenario quindi protocolli come CSMA/CD non si possono usare).

Si useranno altre soluzioni, in particolare per WiFi vedremo CSMA/CA.

Accesso multiplo a Divisione di Codice (CDMA)

Si tratta di un altro schema di accesso multiplo che avevamo anticipato in una delle scorse lezioni, rientra nella **divisione del canale** (gli altri erano ad accesso casuale e a rotazione) insieme a TDM e FDM.

Vedremo questo approccio con uno scenario molto semplificato: anzitutto semplifichiamo la matematica rappresentando i bit 0 e 1 con valori rispettivamente di -1 e 1, poi assumiamo che i segnali provenienti dai vari trasmettenti sono ricevuti tutti con la stessa intensità, lavoriamo poi in modo perfettamente sincronizzato.

Sotto queste assunzioni CDMA è molto semplice: **ad ogni utente è assegnato un codice** sotto forma di un vettore unico di valori (-1 e +1) **t.c. codici di due utenti sono ortogonali** (cioè il prodotto scalare tra i due è zero).

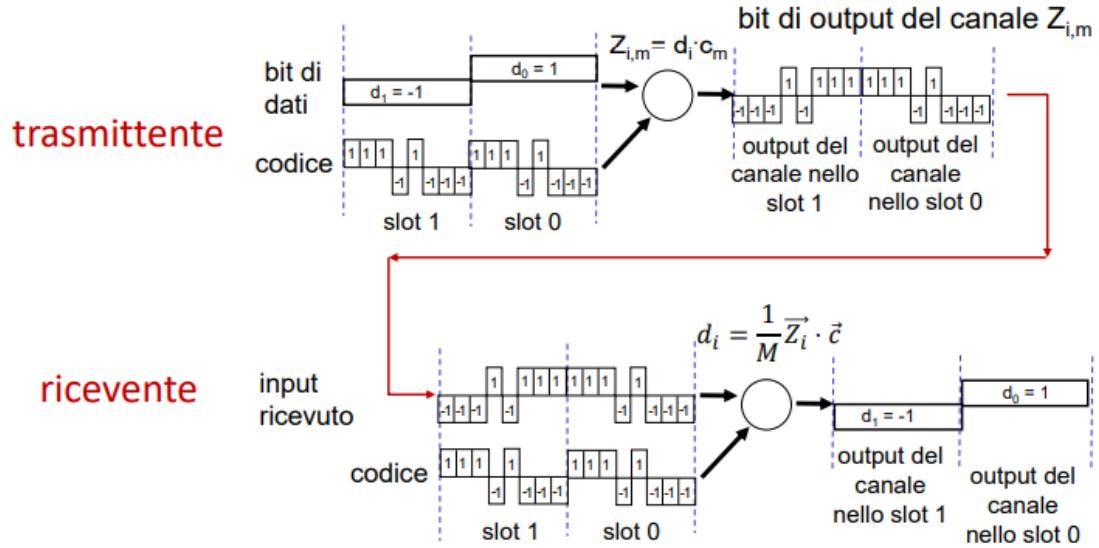
Immaginiamo che utente i debba comunicare un certo bit d_i , come lo codifica? Il suo bit lo codifica moltiplicando per il codice, quindi ottiene un vettore le cui componenti sono quelle del codice moltiplicato per il bit.

Se voglio ad esempio trasmettere il bit 1, ottengo il codice inalterato. Se voglio trasmettere il bit 0 (-1) ottengo il codice con tutti i valori di segno opposto.

Nello schema proposto un codice è lungo 8 bit, quindi se voglio trasmettere m bit di dati al secondo in realtà ne dovrò trasmettere $8xm$ bit di dati al secondo.

Dal punto di vista del ricevente, andrò a leggere i valori codificati. Prendo questi valori, ne faccio il prodotto scalare per il vettore del codice e il risultato lo divido per M (con M numero dei bit di codice) ottenendo il bit di dato originale.

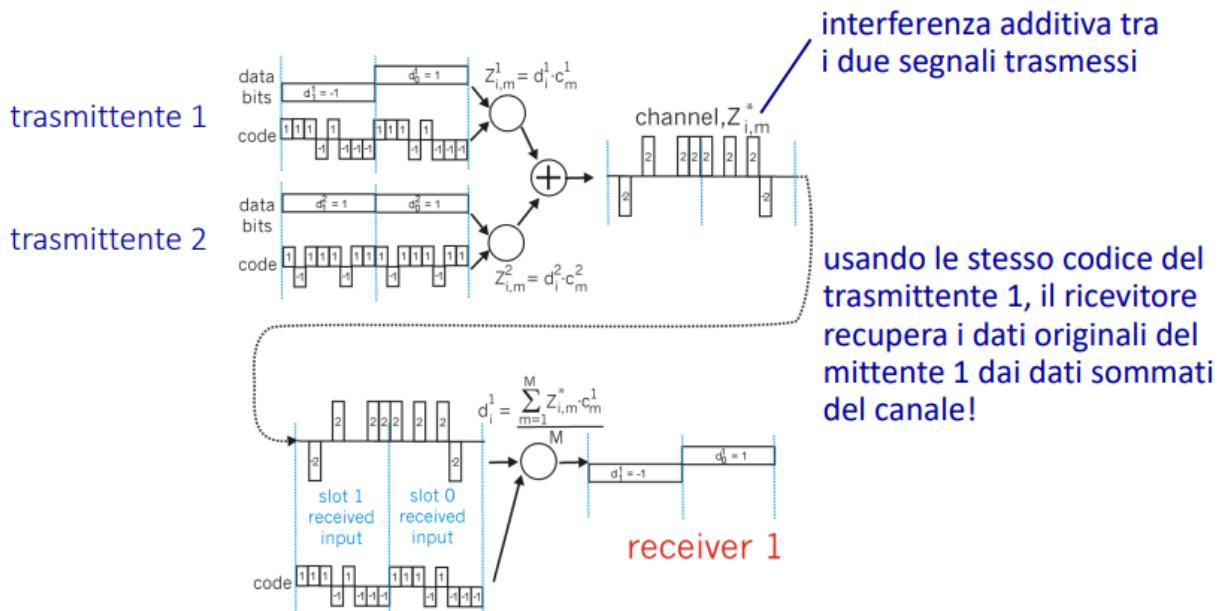
(nel caso dello slot 1 sotto $-1 \times 1 + -1 \times 1 + -1 \times 1 + 1 \times -1 + 1 \times -1 + 1 \times -1 + 1 \times -1 = -8$ divido per 8 $\rightarrow -1$ o.k.)



Ma che succede se due nodi trasmettono simultaneamente?

Come detto, ognuno trasmette con il proprio codice, se ne facciamo il prodotto scalare per come li abbiamo definiti deve venire 0.

Poiché per semplicità stiamo assumendo sincronizzazione perfetta, quando vi è una collisione si sommano i valori. Dal punto di vista del ricevente, per le formule definite prima, anche con valori sommati si ottiene il bit che intendeva inviare il mittente!



WiFi 802.11 wireless LAN

802.11n, ac e ax sono stati ribattezzati rispettivamente WiFi 4, 5 e 6.

IEEE 802.11 standard	Anno	Max data rate	Raggio	Frequenza
802.11b	1999	11 Mbps	30 m	2.4 GHz
802.11g	2003	54 Mbps	30m	2.4 GHz
802.11n (WiFi 4)	2009	600 Mbps	70m	2.4, 5 GHz
802.11ac (WiFi 5)	2013	3.47Gbps	70m	5 GHz
802.11ax (WiFi 6)	2020 (exp.)	14 Gbps	70m	2.4, 5 GHz
802.11af	2014	35 – 560 Mbps	1 Km	Bande televisive inutilizzate (54-790 MHz)
802.11ah	2017	347Mbps	1 Km	900 MHz

- Tutti usano CSMA/CA per l'accesso multiplo ed hanno versioni con stazione base e rete ad hoc

Standard più recenti come n ac e ax si sono spostati sulle frequenze a 5 GHz (si evitano interferenze e quindi si va più veloci).

Per standard dettati a coprire più spazio si usano frequenze diverse, come le bande televisive inutilizzate.

Interessante osservare come questi standard ad ampio raggio lavorano sui MHz, frequenze più basse (coerentemente con la relazione vista $(fd)^2$ per l'attenuazione).

802.11 può lavorare sia in **modalità infrastruttura** che **ad hoc**, noi vedremo la modalità infrastruttura in cui abbiamo un **punto di accesso** (access point **AP**) che *funge da stazione base* e denotiamo come **Basic Service Set (BSS)** il *complesso degli access point e tutti gli host connessi ad essi*.

Nella modalità ad hoc non abbiamo access point quindi il BSS è formato solo da host.

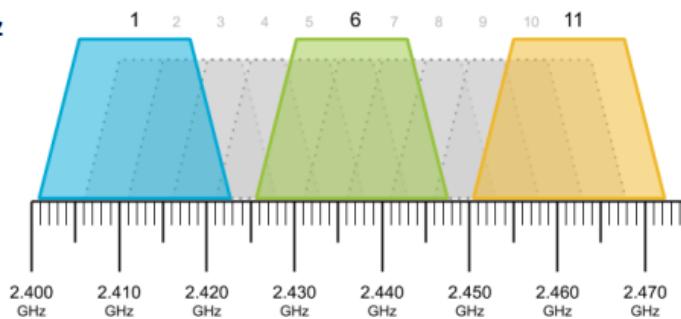
Abbiamo visto che i vari standard wi-fi coprono una certa banda di frequenza, ma tipicamente queste bande sono ulteriormente divise in **canali** che ne rappresentano varie porzioni. *Quando viene configurato l'AP è necessario indicare la banda che andremo ad utilizzare* (in questo modo configurando access point vicini con bande diverse si riducono le interferenze).

Nel caso dello standard 2.4 GHz abbiamo 11 bande di frequenza larghe 85 Mhz che si sovrappongono parzialmente. **Se vogliamo trasmettere senza interferenze chiaramente dovremo utilizzare bande che non si**

sovrappongono, e in generale quelle che non si sovrappongono sono quelle separate da 4 o più canali.

Con questo standard ci si convince che le uniche porzioni che non si sovrappongono sono 1 6 e 11, utilizzabili ad esempio da 3 AP diversi per essere sicuri di non avere interferenze e quindi triplicando la capacità trasmissiva massima di quello standard WiFi.

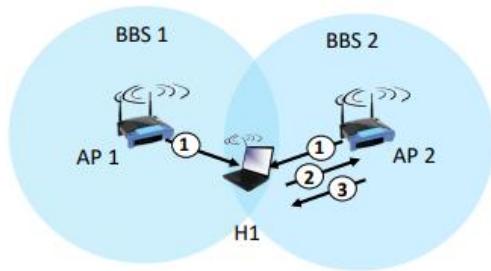
Esempio: 2.4 GHz



- Canali non sovrapposti: separati da 4 o più canali
- Ne esistono 3
- Possibilità di installare 3 AP nello stesso posto per avere il triplo del tasso aggregato massimo

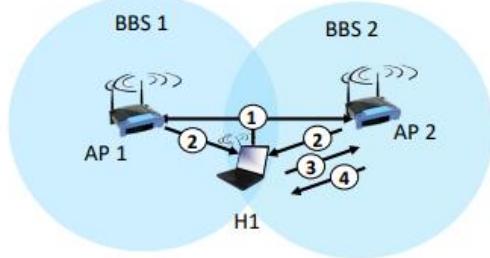
Nella terminologia di WiFi è molto importante il concetto di **associazione**, ovvero il fatto di collegare un host ad un AP. Questa avviene in varie modalità: **nell'approccio passivo** l'AP invia periodicamente dei **frame beacon** contenenti *il nome dell'AP (SSID) e il suo indirizzo MAC*. L'host sceglie l'AP a cui associarsi ricevendo il frame beacon e vedendo ad esempio se quell'AP rappresenta quello con densità maggiore. Può eseguire in maniera opzionale un'autenticazione e una volta associato *il prossimo passo per l'host è quello di inviare una richiesta DHCP nella sottorete attraverso l'AP per ottenere un indirizzo IP nella sottorete*.

Esiste anche un **approccio attivo** in cui è l'host a inviare dei **frame sonda** in broadcast a cui i vari AP rispondono con un frame sonda di risposta. Dopodiché l'host invierà un frame di richiesta di associazione all'AP a cui vuole connettersi e quest'ultimo confermerà o ribalterà il risultato.



scansione passiva:

- (1) frame beacon inviati dagli AP
- (2) invio di un frame di richiesta di associazione da H1 all'AP selezionato
- (3) invio di un frame di risposta di associazione dall'AP selezionato a H1



scansione attiva:

- (1) frame sonda di richiesta inviato in broadcast da H1
- (2) frame sonda di risposta inviato dagli AP
- (3) invio di un frame di richiesta di associazione da H1 all'AP selezionato
- (4) invio di un frame di risposta di associazione dall'AP selezionato a H1

Per i problemi legati all'impossibilità di rilevare collisioni (intensità del segnale trasmesso, debolezza del segnale ricevuto causa attenuazione, terminale nascosto etc..), **con Wireless non ci è possibile usare CSMA/CD.**

Con WiFi quindi tipicamente si usa il protocollo di **CSMA/CA** (collision avoidance). In quanto CSMA, effettua *il rilevamento della portante* (si mette in ascolto sul canale) e ***non trasmette se trova il canale occupato***.

Ma non tenta di rilevare eventuali collisioni, inviando sempre per intero il frame. Al limite, si può indirettamente rilevare la collisione come mancanza di riscontro da parte dell'AP (che invia degli ACK).

(eventuale domanda d'esame) Esistono **due ragioni principali per cui non si può con certezza fare rilevamento di collisione in Wireless** (e in realtà anche della portante): una è il **problema del terminale nascosto** (non vedo i segnali del terminale per cui non posso rilevare la sua trasmissione). Un altro problema è di natura tecnologica/economica: posso sì rilevare il segnale, ma avrà un'intensità molto inferiore rispetto a quello che sto emettendo e in generale ***è difficile dal punto di vista hardware architettare un dispositivo che trasmettendo con una certa potenza riesce contemporaneamente a rilevare un segnale di potenza molto inferiore***.

Vediamo più nel dettaglio come funziona CSMA/CA.

Per prima cosa **facciamo rilevamento della portante**. *Se il canale è percepito inattivo per un tempo detto DIFS* (distributed interframe space) allora trasmetto il frame per intero (senza rilevamento delle collisioni CD).

Se il canale viene percepito occupato 1) faccio un **binary exponential backoff** (come visto cioè scelgo casualmente un valore di attesa che cresce esponenzialmente man mano che reitero il processo).

2) Dopo aver aspettato DIFS il valore di backoff inizia ad essere decrementato solo quando il canale è percepito libero! (es. 10 sec 10 9 8 canale occupato mi fermo, canale libero 7 6 etc...).

3) Quando il valore arriva a 0 (può accadere solo se il canale è libero) allora il frame viene trasmesso per intero. 4) Se non viene ricevuto un ACK dal ricevente, a quel punto viene incrementato l'intervallo di backoff e si riparte con il punto 2).

Dal punto di vista del destinatario è molto semplice, quando gli arriva un frame fa controllo con crc e in caso non rileva errori manda ACK dopo un tempo **SIFS**.

Per quel che riguarda *l'evitare le collisioni* un altro meccanismo è quello di **prenotazione esplicita**.

Viene inviato un frame **RTS** (request to send) di richiesta di trasmissione usando CSMA che viene ricevuto dall'AP il quale invierà in broadcast una risposta di **CTS** (clear to send) dopo un tempo SIFS.

Poiché è CTS è ricevuto da tutti, funge da permesso per chi aveva mandato RTS e da ordine di differire agli altri. Ciò fa sì che non si abbiano collisioni.

Può sussistere una collisione di RTS, ma questi segmenti sono molto piccoli e non comportano quindi un grande spreco (il problema delle collisioni era che non interrompendo la trasmissione in modo puntuale spreco tempo a continuare a trasmettere un frame che è già andato in collisione).

Il meccanismo RTS è opzionale e ha senso usarlo quando devo trasmettere un frame grande (usarlo per ogni frame sarebbe inefficiente, devo comunque attendere conferma CTS prima di trasmettere e non ha senso per frame piccoli)

Struttura del frame 802.11



Indirizzo 1: indirizzo MAC dell'host wireless o AP che deve ricevere il frame (può non essere il destinatario finale)

Indirizzo 2: indirizzo MAC dell'host wireless o AP che trasmette il pacchetto (può non essere il mittente iniziale)

Indirizzo 4: usato solo in modalità ad hoc

Indirizzo 3: indirizzo MAC dell'interfaccia router cui l'AP è collegato

Indirizzo 3 gioca un ruolo cruciale nell'internetworking tra un BSS e una LAN cablata

Si hanno delle **informazioni di controllo**, una **durata** (che serve ad indicare nelle richieste RTS e risposte di clearance CTS il tempo che viene dato). Si hanno poi **4 indirizzi** e un **numero di sequenza**, che serve *per il trasferimento dati affidabile*. Abbiamo poi il **payload** e i bit per **CRC**.

Ma perché avere trasferimento di dati affidabile se già lo abbiamo a livello di trasporto?

Perché a livello di trasporto l'affidabilità è end-to-end, ma come ottimizzazione se gestiamo localmente a un collegamento l'affidabilità permette di evitare spreco di risorse nel rimandare il pacchetto lungo tutto il percorso!

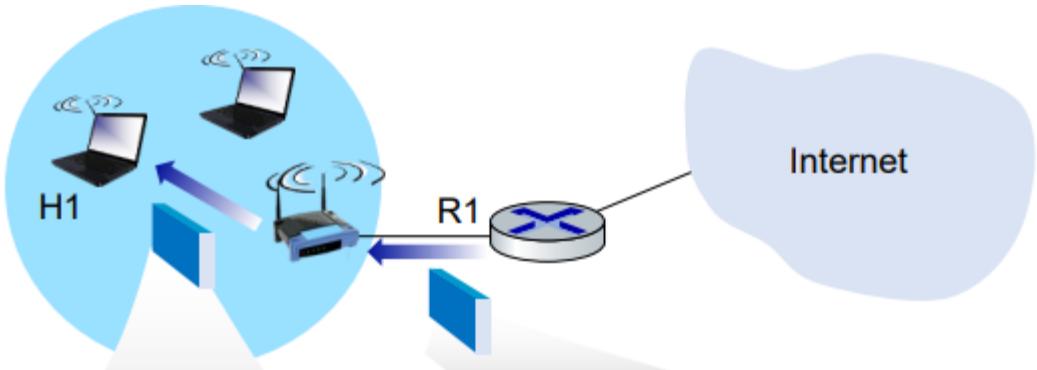
Perché abbiamo 4 indirizzi? (in realtà il 4 è usato solo per la modalità ad hoc, quindi vediamo solo i primi 3).

In WiFi si distingue l'entità che trasmette il frame dal mittente originale, e allo stesso modo si distingue il ricevente attuale da quello che è il destinatario ultimo del frame.

Indirizzo 1 == indirizzo MAC dell'host wireless o AP che deve ricevere il frame (non è necessariamente il destinatario finale)

Indirizzo 2 == indirizzo MAC dell'host wireless o AP che trasmette il pacchetto (non è necessariamente il mittente iniziale)

Indirizzo 3 == indirizzo MAC dell'interfaccia router a cui l'AP è collegato (per accedere ad altre sottoreti)



Immaginiamo che il router R1 debba inviare il datagramma ad H1. I dati saranno incapsulati in un frame il cui MAC address destinatario sarà H1 e come MAC address mittente l'interfaccia del router di questa sottorete.

Il frame arriva all'AP della sottorete con H1, e dovrà mandarlo ad H1. Dovrà creare quindi un frame di tipo 802.11 popolando opportunamente i 4 indirizzi. Sarà messo come indirizzo ricevente (che coincide con il destinatario in questo caso) H1 e come indirizzo di chi trasmette il proprio indirizzo MAC e come indirizzo MAC del mittente (di chi ha originariamente messo il datagramma nel frame) l'indirizzo del router R1.

Quando H1 riceve il datagramma tira fuori il payload e dovrà inviare un datagramma di risposta. Dovrebbe destinare il datagramma all'interfaccia del router, ma in realtà gli indirizzi di 802.11 saranno popolati come segue: l'AP sarà messo come ricevente poiché è lui che riceve il frame, metteremo nell'indirizzo 2 come colui che trasmette che è anche mittente H1, come indirizzo 3 (destinatario) ci si mette il router.

Mobilità all'interno della stessa sottorete

Mi sposto da un AP all'altro, questi sono connessi tra loro con uno switch (sempre livello 2, si tratta sempre della stessa sottorete) quindi l'host H1 deve mantenere lo stesso IP. **Ma come fa a sapere lo switch avendo un frame destinato per H1 da quale parte mandarlo dal momento che H1 si muove?**

Normalmente lo apprende la prima volta che riceve un frame inviato da H1 inserendolo nella sua tabella di commutazione.

Ma se l'host si sposta ok che può mantenere lo stesso IP, ma dal punto di vista della rete di livello 2 lo switch dovrebbe sapere che il collegamento è cambiato (ma gli switch non sono pensati per reagire a cambiamenti rapidi di posizione).

Un modo per risolvere il problema può essere che il nuovo AP invia un frame Ethernet broadcast con mittente H1 affinché lo switch apprenda la nuova porta per raggiungere H1.

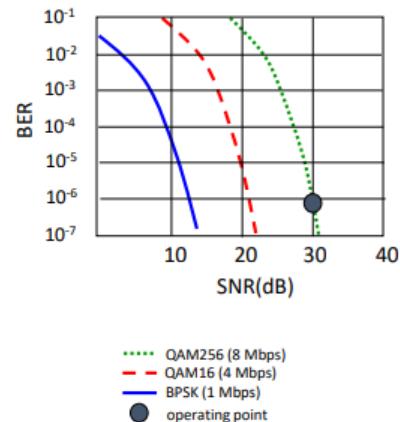
Esistono standard recenti come 802.11f pensati proprio per standardizzare questo meccanismo.

802.11: funzionalità avanzate

Adattamento del tasso trasmissivo

- La stazione base e la stazione mobile cambiano dinamicamente il tasso trasmissivo (tecnica di modulazione a livello fisico) come la stazione mobile si sposta e di conseguenza cambia l'SNR

1. SNR cala BER aumenta quando il nodo si allontana dalla stazione base
2. Quando il BER diventa troppo alto, passa un tasso trasmissivo inferiore ma con BER inferiore



Gestione dell'energia

- Nodo ad AP: "Sto per diventare inattivo fino al prossimo frame beacon"
 - AP sa che non deve trasmettere frame a questo nodo
 - il nodo si riattiva prima del successivo frame beacon
- frame beacon: contiene la lista dei nodi che devono ricevere frame in attesa sull'AP
 - il nodo rimane sveglio se devono essere inviati frame da AP a mobile; altrimenti dorme di nuovo fino al prossimo frame beacon

Lez 24

Riassunto

Wireless e Mobilità non sono sinonimi, nel primo caso si fa riferimento propriamente al mezzo comunicativo mentre nel secondo alle problematiche

derivative dallo spostamento degli utenti connessi (chiaramente questi utenti sono connessi via Wireless, quindi wireless presupposto per la mobilità).

Abbiamo descritto le caratteristiche principali del mezzo trasmissivo Wireless: **attenuazione, propagazione lungo cammini multipli, problema del terminale nascosto e interferenza**.

Queste caratteristiche differenziano Wireless dai collegamenti cablati e ci obbligano a fare degli aggiustamenti, ad esempio il problema del terminale nascosto e la difficoltà tecnica di trasmettere un segnale ad alta intensità e contemporaneamente ricevere un segnale di più bassa intensità comporta che nel mondo Wireless **non possiamo fare Collision Detection**, (chiede all'esame) quindi quando trasmettiamo inviamo il frame per intero e rileviamo dei problemi come mancanza di responso (ACK) da parte del ricevente (in genere la stazione base).

Da differenziarsi il concetto di ricevitore e destinatario (vedi WiFi, il frame contiene 3 indirizzi dove si distinguono chi trasmette dal mittente originale e chi riceve dal destinatario originale).

In alternativa al **Collision Detection** (CD) WiFi utilizza **Collision Avoidance** (CA) facendo di tutto per evitare le collisioni a priori.

Mentre in Ethernet non si trasmette se il canale è occupato e si trasmette non appena si libera, in WiFi con CA un host che trova un canale occupato entra comunque in un tempo di attesa casuale, in questo modo si riduce la possibilità di collisione con pacchetti inviati da altri host che hanno fatto la stessa cosa. *Ciò è importante perché in assenza di Collision Detection nel Wireless le collisioni sono costose (devo trasmettere interi Frame), in Ethernet ciò non è necessario poiché grazie a CD appena si rileva la collisione la trasmissione viene interrotta.*

Un'altra tecnica per evitare le collisioni in WiFi è la presenza di un meccanismo di eventuale **prenotazione** (il mittente invia una richiesta di trasmissione e il ricevente invia in broadcast una risposta di clearance se il canale è libero). È un meccanismo opzionale che serve a proteggere la trasmissione di pacchetti di grandi dimensioni.

Bluetooth

Esso serve per costruire le **Wireless Personal Area Network (WPAN)**, vale a dire reti d'area personale. Si tratta reti di diametro ridotto (meno di 10 m) *atte tipicamente alla sostituzione di cavi* (mouse, tastiera, cuffie...).

Non dobbiamo confondere questo tipo di reti con le **Body Area Network**, cioè reti dedicate a dispositivi indossabili con particolare enfasi a dispositivi biomedici. Chiaramente una rete di questo tipo può avvalersi di Bluetooth, ma non vale per tutte (per certi accessori, soprattutto legati alla salute, magari si usano approcci diversi, più veloci e sicuri).

Bluetooth usa la **banda ISM con frequenza 2.4-2.5GHz** e può garantire **fino a 3 Mbps** di tasso trasmissivo.

Questa banda è molto affollata, vi trasmettono versioni precedenti di WiFi, telefoni cordless, inoltre apparati elettrici come fornì a microonde o alcuni motori emettono radiazione elettromagnetica proprio in quelle frequenze. Pertanto Bluetooth è stato pensato con un'attenzione particolare alle interferenze.

Vediamo ora l'implementazione dell'accesso alla rete Wireless via Bluetooth. Bluetooth **combina TDM con FDM** (protocolli di suddivisione del canale rispettivamente in termini di tempo e frequenza, necessario per **l'accesso multiplo*** (si ricorda che Wireless è per sua natura segnale inviato in broadcast)).

Con TDM e FDM c'è garanzia che non vi siano collisioni.

Bluetooth combina TDM e FDM con slot tipicamente di 625 microsecondi in ognuno dei quali vari mittenti possono trasmettere in uno di 79 diversi canali (di frequenze differenti).

La cosa interessante di Bluetooth è l'uso della **tecnica frequency-hopping spread spectrum (FHSS)**, ovvero *un dispositivo non trasmette sempre dallo stesso canale ma da uno slot di tempo all'altro cambia canale di frequenza secondo una sequenza pseudo-casuale* (la determina il **nodo master**).

La ragione di FHSS risiede proprio nella volontà di **ridurre le interferenze** con altre fonti di radiazione elettromagnetica. (se utilizzassi sempre lo stesso canale potrei beccare un'interferenza prolungata che mi rovina tutto il segnale, se invece ogni tot cambio canale evito sprechi eccessivi).

*(protocolli visti per accesso multiplo: *suddivisione canale* (in cui rientrano TDM, FDM e suddivisione di codice), *accesso casuale* e a *rotazione*).

Bluetooth opera (in termini di **modalità operativa**) **ad hoc**, cioè senza infrastruttura.

La rete in cui opera Bluetooth è detta WPAN o **piconet** in quanto oltre ad avere dimensioni ridotte possono operarvi pochi dispositivi.

I dispositivi attivi possono infatti essere al più 8, di cui uno svolge il ruolo di

Master (che regola il la potenza di trasmissione, la sequenza di salti tra frequenze, il “clock” ovvero la temporizzazione degli slot, il **polling** dei client) e gli altri sono dispositivi **Client**.

Come detto il Master decide anche il polling dei client, ovvero il Client riceve il permesso di trasmettere da quest’ultimo -> Bluetooth non solo combina FDM e TDM, ma ci mette in mezzo pure il **protocollo a rotazione** del polling.

In aggiunta a questi dispositivi attivi si possono avere fino a 255 nodi in **parked mode**, ovvero nodi che “si addormentano” (per preservare la batteria) per poi risvegliarsi periodicamente (il Master si occupa di cambiare il loro status da parked ad attivi e viceversa).

(es. sensore per la finestra, fintanto che questa non si apre o chiude non deve trasmettere nulla, quindi si tiene in parked mode fintanto che non rileva certe frequenze).

La WPAN è quindi una rete ad hoc **bootstrapping**, per cui cioè i nodi si “autoassemblano” nella piconet seguendo due fasi:

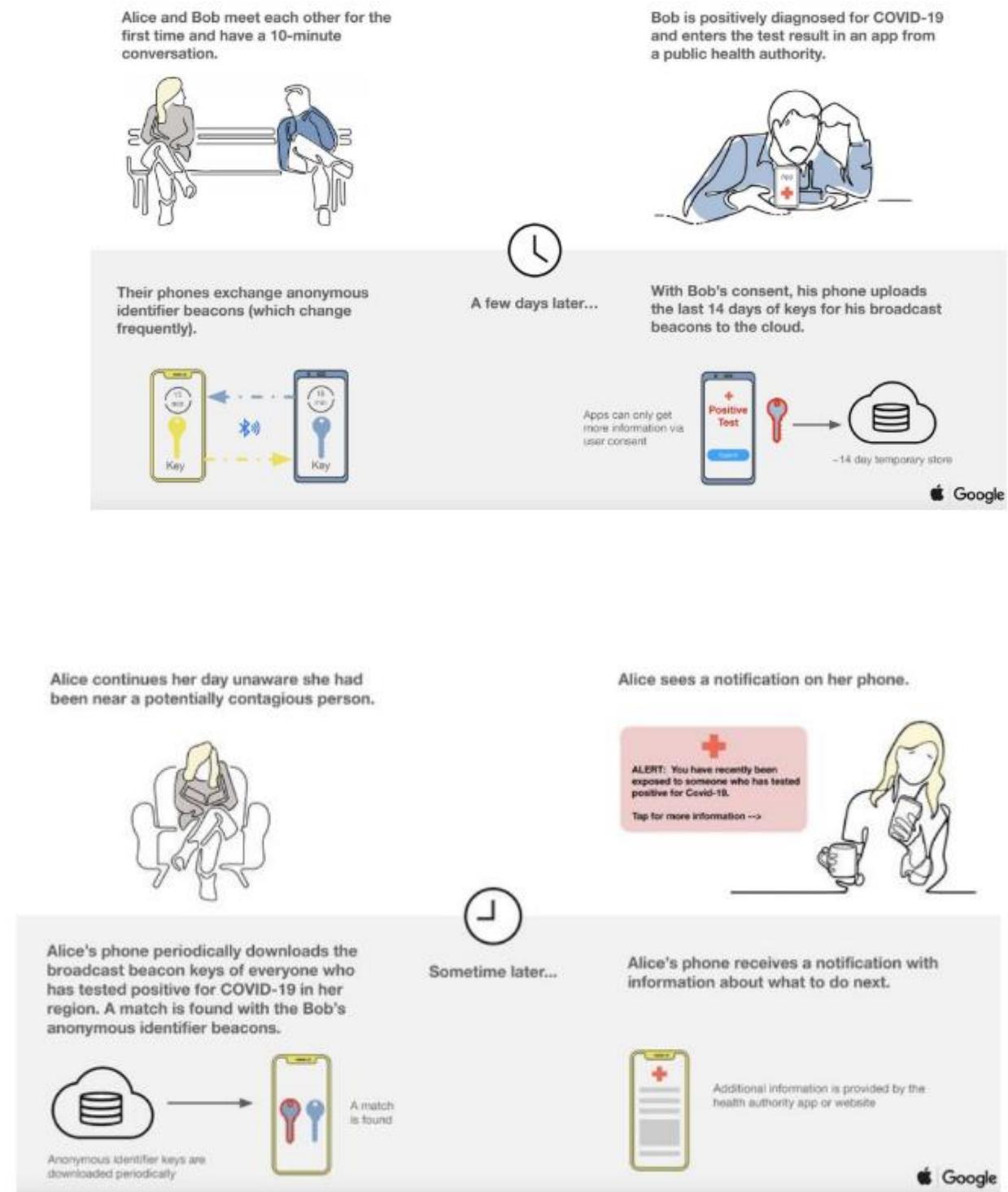
-**Neighbor Discovery**: il master invia ripetutamente messaggi *inquiry* in broadcast per vedere i nodi disponibili, i dispositivi in ascolto rispondono dopo un ritardo casuale (per evitare collisioni)

-**Paging**: il master manda a ciascuno dei dispositivi disponibili un invito, quando riceve un ACK manda al dispositivo del pattern per il *frequency hopping*, il *clock* relativo allo slot FDM e *un indirizzo* da usare nella comunicazione.

Beacon == **dispositivi che inviano frequentemente un messaggio, per esempio con un identificatore.**

Possono servire a vari scopi, es. stupido beacon trasmette appena torno a casa e fa accendere ad Alexa le luci. Vedi altro esempio sotto

Pandemia + Bluetooth



Reti Cellulari 4G e 5G

Si tratta della **soluzione per Internet Mobile in wide-area**, se mi muovo in area geografica ho bisogno di 4G e 5G.

I tassi di trasmissione arrivano fino a centinaia di Mbps (4G a condizioni ideali, no interferenze, vicinanza alla torre etc...).

similarità con Internet cablato

- Distinzione periferia/nucleo, ma entrambi appartengono allo stesso **carrier**
- Rete cellulare globale: una rete di reti
- Uso diffuso di protocolli: HTTP, DNS, TCP, UDP, IP, NAT, separazione tra piano di controllo e piano di dati, SDN, Ethernet, tunneling
- Interconnessione a Internet cablato

Differenze con Internet cablato

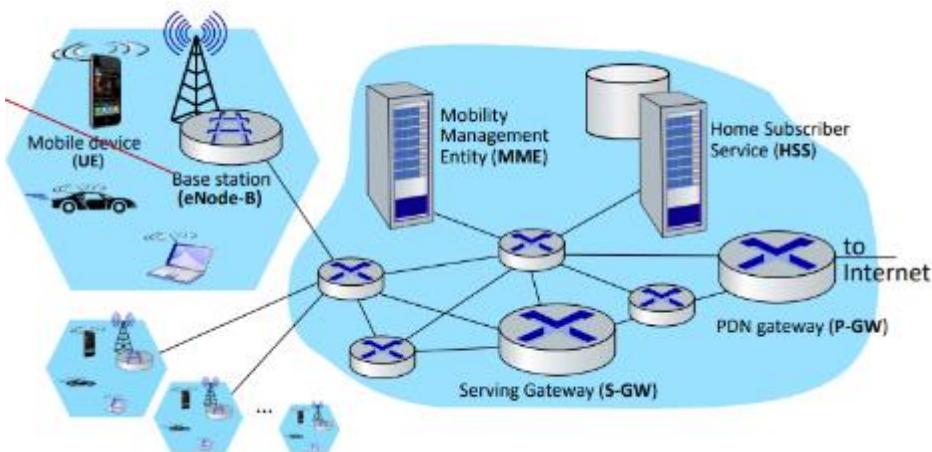
- Differenti protocolli livello di collegamento wireless
- Mobilità come servizio di 1° classe
- "identità" dell'utente (attraverso la SIM card)
- modello di business: gli utenti si abbonano a un operatore di telefonia mobile
 - forte nozione di "home network", contrapposta al roaming in una "visited network"
 - accesso globale, con infrastruttura di autenticazione, e accordi tra operatori

Similiarità con l'Internet cablato: Nella rete cellulare periferia e nucleo appartengono allo stesso **carrier**, ovvero operatore telefonico. La rete cellulare ha copertura globale tramite interconnessione di diverse stazioni (rete di reti). 4G e 5G fanno ampio uso di protocolli già visti come HTTP, DNS, TCP, UDP, IP, NAT e vi è separazione tra piano di controllo e piano di dati, si usano SDN, Ethernet e strategie di Tunneling. Non solo le reti sono interconnesse tra di loro, ma vi è chiaramente anche un interconnessione all'Internet pubblico (cablato).

Differenze con l'Internet cablato: Si hanno protocolli differenti a livello di collegamento Wireless. La mobilità assume in questo senso un'importanza primaria (a differenza delle reti cablate dove non è necessario tenerne conto così tanto), la rete cellulare è stata pensata proprio in funzione della mobilità. L'identità dell'utente è data da una SIM card, identificatore globale che non cambia (differenza rispetto al MAC address: la SIM ha una circuiteria che permette un processo di autenticazione per cui la rete può essere certa di star comunicando con un dispositivo specifico).

Nelle reti cellulari esiste un forte modello di business per cui gli utenti sottoscrivono un abbonamento con uno specifico operatore di telefonia mobile (che diventa la nostra **home network**, contrapposta alle **visited network** di altri operatori). L'accesso globale è garantito dagli accordi tra i vari operatori.

Elementi dell'architettura 4G



Nella rete 4G/5G abbiamo una **rete di accesso radio**, costruita attorno ad una **stazione base** (detta eNode-B, dove e sta per enhanced e B base station) che rappresenta la nostra torre cellulare a cui possono associarsi vari dispositivi mobili, detti anche **User Equipment (UE)**. La stazione base è simile all'access point WiFi ma ha un ruolo molto più attivo in termini di mobilità e interconnessione con le altre stazioni per ottimizzare l'uso della banda radio. Ogni stazione base determina una cella dove possono connettersi tutti gli UE nel raggio di copertura.

Questi UE presentano un **IMSI** (international mobile subscriber identity, cioè un ID a 64 bit memorizzato sulla **SIM card** (subscriber identity module). La differenza col MAC address (oltre al numero di bit, il MAC ne ha 48) *l'IMSI identifica anche la rete home a cui appartiene il dispositivo*. La SIM come detto contiene questo IMSI e non ha il solo scopo di memoria, ma anche di autenticazione per far sì che la rete sia certa di starsi connettendo a uno specifico dispositivo.

Queste reti di accesso sono connesse tra loro tramite un **nucleo**, detto **Enhanced Packet Core (EPC)**, connette tutte le celle *dello stesso operatore* ed è connesso al resto di Internet (collegamenti cablati che collegano le stazioni base al nucleo).

Nel nucleo, oltre ai vari router, troviamo:

-**SG-W e P-GW**: si tratta di due router particolari. Come visto ogni rete di accesso del fornitore deve essere connessa allo stesso nucleo (gestito sempre dal fornitore), che contiene vari router interconnessi tra loro fino ad arrivare al gateway che connette il carrier con il resto di Internet.

Lungo il percorso dal dispositivo mobile a Internet troviamo due router speciali: l'**S-GW (Serving Gateway)** e il **P-GW (PDN Gateway)**.

Il loro ruolo è legato alla mobilità, se infatti mi deve arrivare un pacchetto e mi sposto da una cella a un'altra *l'indirizzamento del pacchetto basato su destinazione non funziona!*

Anziché maneggiare le tabelle di inoltro l'idea per risolvere il problema nelle reti cellulari è adottare il meccanismo di **Tunneling**: *si crea un tunnel da P-GW a S-GW e da S-GW alla stazione base a cui è connesso il dispositivo*.

Abbiamo già parlato di Tunneling con IPv4 – IPv6: **il pacchetto viene incapsulato in un altro la cui intestazione determina l'inoltro**, “ignorando” quella che sarebbe stata la strategia adottata dall'indirizzamento originale.

Lo vedremo poi nel dettaglio, intanto l'importante è sapere che *il Tunneling porta ad un vantaggio in termini di mobilità* (se l'host si sposta da una cella a un'altra mi basta invalidare il vecchio tunnel e creare un altro tunnel dall'S-GW alla nuova cella).

- **Home Subscriber Service (HSS)**: è un **database che contiene informazioni circa gli abbonati**. Questo coopera con l'MME, entità che ha un ruolo attivo nella rete.

- **Mobility Management Entity (MME)**: si occupa del **setup del percorso (tunneling) tra mobile device e P-GW**, in particolare entra in gioco se c'è roaming tra celle di operatori diversi.

Si occupa **dell'autenticazione dei dispositivi in entrambi i versi**: autenticazione del dispositivo nella rete e contemporaneamente deve convincere il dispositivo che la rete su cui si trova è “genuina”, per farlo deve contattare l'HSS nella rete home del dispositivo.

Si occupa come detto della **mobilità**: **handover** dei device tra celle e **tracking/paging** della posizione dei device.

Quindi i tunnel si creano così: quando un device entra in una rete cellulare dialoga con l'MME di quella rete che farà autenticazione e setup dei vari tunnel.

Nel 4G (anche detto LTE) vi è **netta separazione tra piano di dati e piano di controllo**.

- **Piano di controllo**: si hanno nuovi protocolli specifici per la gestione della mobilità, sicurezza e autenticazione
- **Piano di dati**: nuovi protocolli a livello fisico e di collegamento, uso estensivo di tunnel per gestire la mobilità.

Ripetizione Tunneling: a livello logico P-GW e S-GW sono direttamente connessi (più a basso livello questo collegamento diretto è realizzato tramite tunneling tra vari router con destinazione finale S-GW). Quando S-GW riceve il pacchetto fa il decapsulamento e vede il destinatario, l'MMI ha già fatto il setup del tunnel verso quel destinatario che si trova in una certa stazione base: ancora una volta incapsulamento tunneling verso quella specifica stazione base (che a sua volta decapsula, vede il cellulare di destinazione e glielo manda).

Tra i vari protocolli LTE se ne hanno 3 specifici e importanti a livello di collegamento:

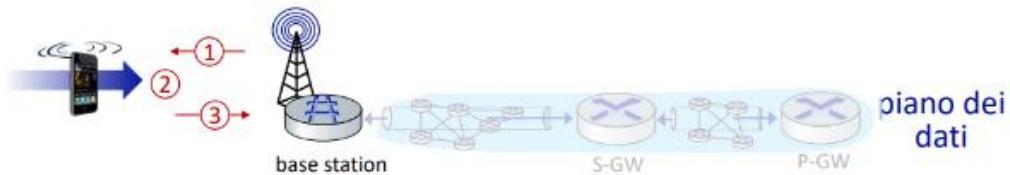
- **Packet Data Convergence**: per la compressione degli header e la cifratura;
- **Radio Link Control Protocol (RLC)**: per la frammentazione/riassembaggio dei frame e il trasferimento affidabile
- **Medium Access**: richiesta e uso di slot per la trasmissione radio (OFDM, tecnica simile a Bluetooth).

Vediamo a basso livello il Tunneling:

anzitutto l'incapsulamento avviene (come avveniva per IP, qua definiamo la cosa in modo più preciso) con ***l'incapsulamento del datagramma IP in un pacchetto UDP secondo le regole dettate da un protocollo di livello superiore detto GTP*** (non si tratta quindi di tunnel puramente IP ma ci sono di mezzo altri protocolli).

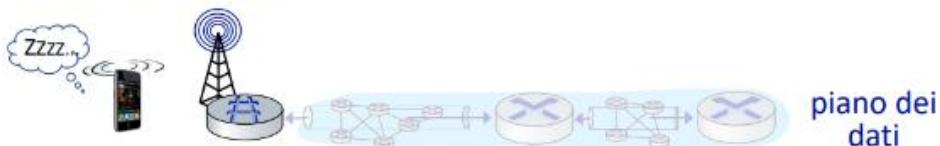
Se è invece il dispositivo mobile a inviare i dati il primo tunnel lo porta all'S-GW e un altro tunnel al P-GW, poi da lì verso il gateway di Internet.

Piano dei dati LTE: associazione con una BS



- ① BS invia in broadcast un segnale di sincronizzazione primario ogni 5 ms su tutte le frequenze
 - BS di più carrier possono inviare in broadcast segnali di sincronizzazione
- ② Il mobile node trova un segnale di sincronizzazione primario, quindi individua il secondo segnale di sincronizzazione su questa frequenza.
 - il mobile node trova quindi le informazioni trasmesse dalla BS: larghezza di banda del canale, configurazioni, informazioni sul vettore cellulare della BS.
 - il mobile node può ricevere informazioni da più stazioni di base, più reti cellulari
- ③ il mobile node sceglie con quale BS associarsi (ad esempio, preferendo la rete dell'operatore d'origine)
- ④ sono necessari altri passaggi per l'autenticazione, la creazione dello stato e la configurazione del piano dati.

LTE: sleep modes



come in WiFi, Bluetooth: i mobile node LTE possono mettere la radio a "dormire" per preservare la batteria:

- **light sleep:** dopo centinaia di millisecondi di inattività
 - si svelga periodicamente (centinaia di ms) per controllare le trasmissioni downstream
- **deep sleep:** dopo 5-10 secondi di inattività
 - la mobilità può cambiare le celle durante il sonno profondo: è necessario ristabilire l'associazione.

Abbiamo finora definito il fatto che un dispositivo ha una propria **rete home** in base al carrier a cui ha sottoscritto l'abbonamento. *Ma come può connettersi in roaming ad una stazione di base di un altro carrier?*

Grazie al fatto che i **P-GW** dei vari carrier sono interconnessi tra loro tramite **protocollo IPX** e tramite Internet pubblico.

Passaggio al 5G anche per consentire scenari di affidabilità massima e latenza bassa (immagina un chirurgo che opera a distanza).

Passaggio al 5G: motivazione

- **obiettivo:** incremento di 10x del bitrate di picco, riduzione di 10x della latenza, aumento di 100x della capacità di traffico rispetto 4G
 - **5G NR (new radio):**
 - due bande di frequenza: FR1 (450 MHz–6 GHz) and FR2 (24 GHz–52 GHz): frequenze delle onde millimetriche
 - non è retrocompatibile con il 4G
 - MIMO: antenne multiple direzionali
 - **frequenze delle onde millimetriche:** velocità di trasmissione dei dati molto più elevate, ma su distanze più brevi
 - pico-cell: diametro: 10-100 m
 - necessaria distribuzione massiccia e densa di nuove stazioni di base

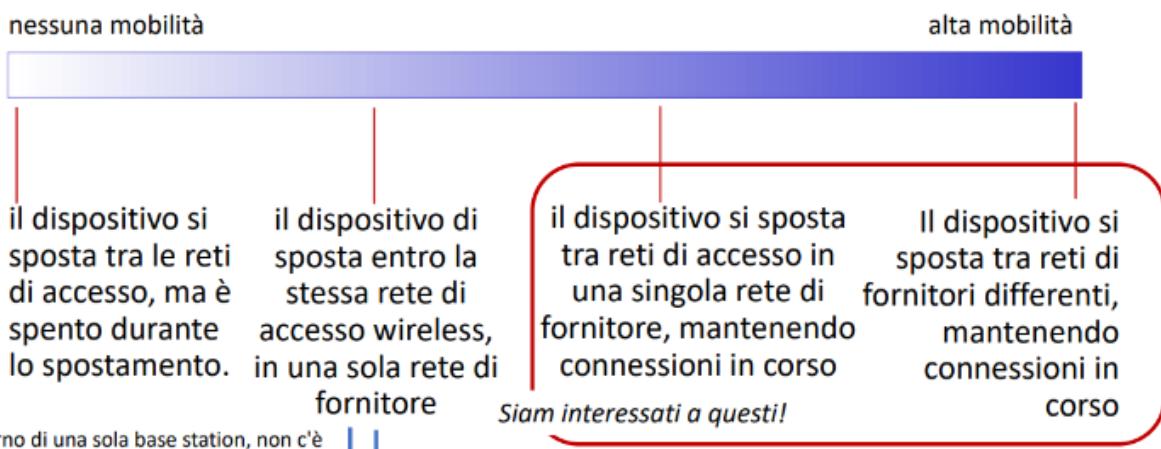
(frequenza più alta == mando roba più velocemente == attenuazione maggiore (quindi distanza percorsa minore))

Per tale ragione 5G usa tecniche complesse per evitare l'attenuazione (ragionando su riflessione del segnale e altra roba che non studiamo).

Mobilità

Cos'è la mobilità?

- spettro della mobilità, dal punto di vista del livello di rete:



all'interno di una sola base station, non c'è neanche mobilità a livello di collegamento

Lo spostamento quindi non ha significato puro in quanto spostamento fisico, quanto come spostamento tra reti diverse volendo comunque mantenere la connessione.

Come già detto diverse volte infatti, **il problema principale della mobilità risiede nella possibilità che un dispositivo si sposti da una rete all'altra**. In tal caso come fa la “rete” a sapere che deve inoltrare i pacchetti alla nuova rete (affinché quindi il dispositivo interessato resti connesso alla rete con lo stesso indirizzo IP)?

Esistono **due approcci**.

Il primo approccio, *puramente teorico*, **risiede nell’idea di lasciare che sia la rete (e quindi i router) a gestire la situazione**.

I router annunciano quindi il nome, l’indirizzo IP permanente del nodo mobile in vista inserendolo nella tabella di routing. Quando un indirizzo mobile cambia rete la tabella di inoltro dovrà essere aggiornata (anche in base ad accordi presi tra i carrier in modo che nel nucleo del carrier in cui si trova il dispositivo venga indicato un indirizzo con prefisso più lungo, eventualmente tutto l’indirizzo IP specifico).

Il problema di questo approccio è che non è scalabile per miliardi di dispositivi (come avevamo già detto quando abbiamo parlato di indirizzamento e inoltro, è necessario che il prefisso non superi un tot limite altrimenti avremmo bisogno di tabelle eccessivamente grandi).

Il secondo approccio, effettivamente applicabile, è quindi quello di **lasciare che siano gli end-system a gestire la situazione**. Esistono due approcci ulteriori in questa categoria:

- **Instradamento Indiretto** (indirect routing) per cui la comunicazione tra corrispondente e dispositivo mobile è **mediata dalla home network**, la quale inoltra tramite un tunnel i pacchetti alla rete visitata dal dispositivo.

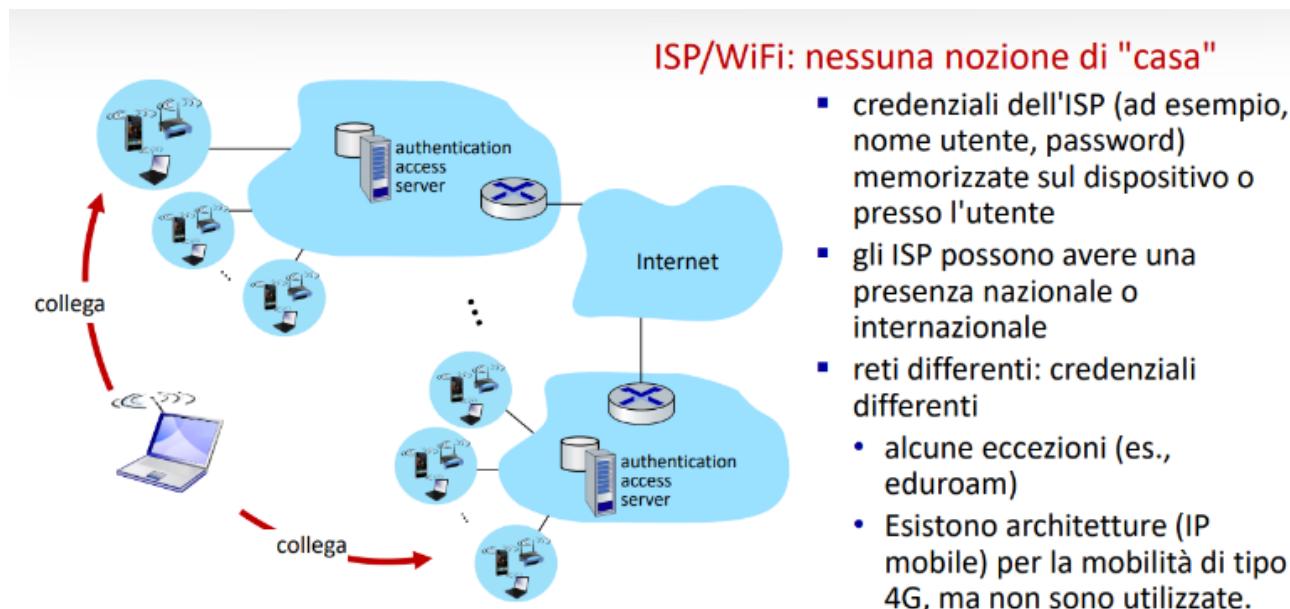
- **Instradamento Diretto** (direct routing) per cui il corrispondente ottiene l’indirizzo del dispositivo mobile nella rete visitata e invia i pacchetti direttamente al dispositivo usando il protocollo d’indirizzamento standard di internet.

Corrispondenza: se cerco un amico che viaggia molto per sapere dove sta chiedo ai genitori oppure guardo instagram sperando abbia messo roba di recente.

Come visto in 4G/5G esiste una differenza sostanziale tra **home network** e **visited network**. La prima rappresenta la rete del carrier (operatore) a cui il dispositivo è abbonato, l'HSS memorizza le informazioni circa l'identità del dispositivo e dei servizi abilitati.

La seconda rappresenta una qualsiasi altra rete, diversa da quella domestica (home) a cui mi associo.

Con WiFi/ISP invece non esiste una vera e propria nozione di "casa":



Come affrontare la mobilità?

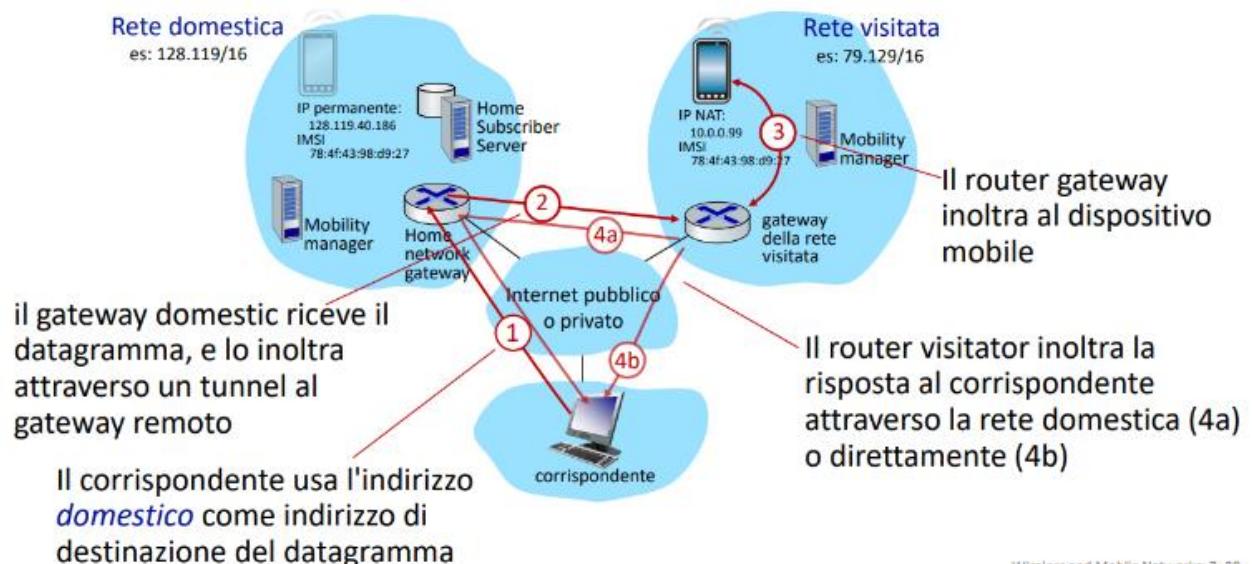
Ho il mio dispositivo che ha un IMSI specifico e un indirizzo IP permanente, registrati nell'HSS della rete home.

Nel momento in cui mi sposto in Roaming su un'altra rete visited allora ho con me il mio IMSI, ma essendo la rete diversa avrò bisogno di un altro indirizzo IP (che può essere anche NAT).

Ciò avviene come segue: anzitutto **il dispositivo nella visited network si associa al Mobility Manager della rete visitata**, questo poi **registra la posizione del dispositivo mobile nell'HSS della home network**.

In questo modo il Mobility Manager della rete visitata sa del dispositivo mobile mentre l'HSS sa la posizione attuale del dispositivo.

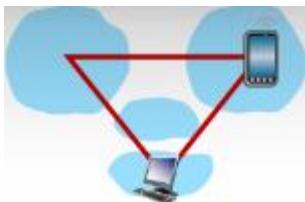
Ora, se voglio mandare un pacchetto al dispositivo mobile come faccio?
Vediamo prima l'approccio con **instradamento indiretto**.



Inizialmente il pacchetto è inoltrato all'indirizzo IP permanente del dispositivo mobile destinatario (tramite instradamento classico basato su destinazione), giungendo così alla sua rete domestica. Quando arriva giunge al router gateway della rete domestica, che sa grazie all'HSS qual è la posizione attuale del dispositivo ed è già stato stabilito, durante la fase di associazione del dispositivo con la rete visitata grazie al Mobility Manager, un tunnel verso l'SG-W nella rete visitata che potrà inoltrare il pacchetto verso il dispositivo mobile.

Il dispositivo potrà a questo punto rispondere in due modi: o viene inviata la risposta direttamente al mittente oppure attraverso la rete domestica (nella figura rispettivamente 4b e 4a).

Questo metodo di instradamento è detto triangolare, poiché si passa prima per la home network per poi giungere alla vera destinazione.



Contro: il routing indiretto è inefficiente se il corrispondente e il dispositivo mobile si trovano sulla stessa rete (dovranno comunque dialogare passando per il gateway).

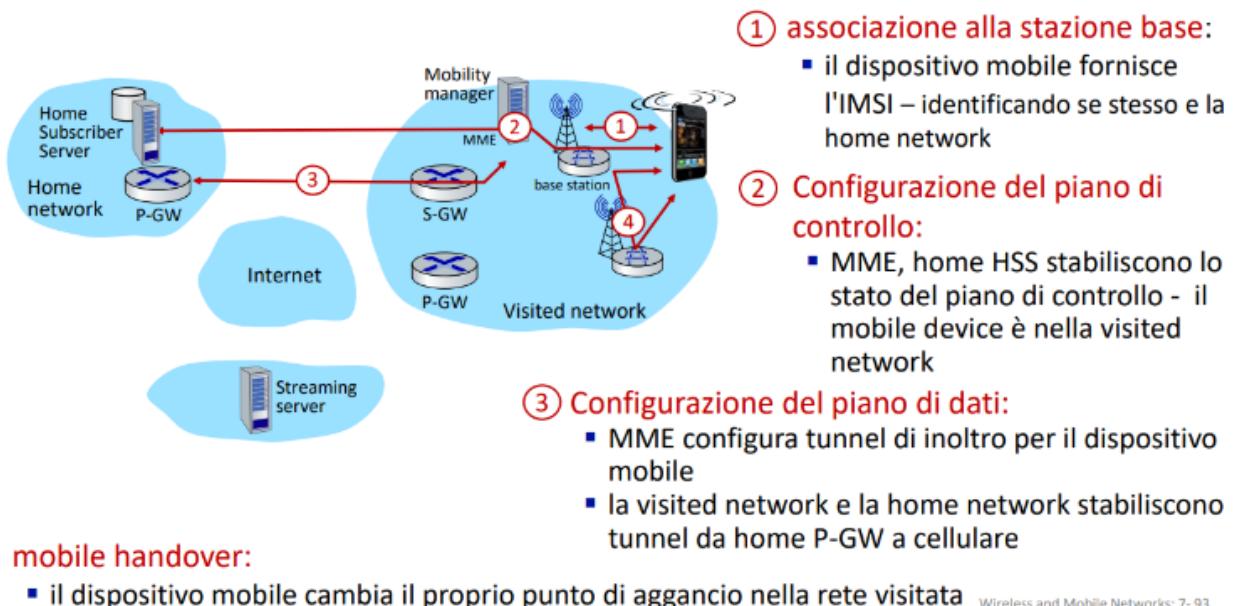
Pro: il dispositivo mobile nello spostarsi tra visited networks risulterà trasparente al corrispondente! (nel registrarsi ad una nuova rete visitata ci pensano l'HSS e l'MME a gestire il tutto). Quindi connessioni (come TCP) in corso tra corrispondente e dispositivo mobile possono essere mantenute!

Vediamo ora l'approccio del **routing diretto**.

Ciò che accade sostanzialmente è che invece di passare per dei tunnel il corrispondente si fa comunicare di volta in volta la posizione del dispositivo mobile a cui deve inviare pacchetti.

Questo approccio non ha i contro del routing indiretto, ma presenta l'enorme **difetto per cui il dispositivo mobile non risulterà trasparente al corrispondente** (quindi non posso mantenere connessioni). Il corrispondente infatti deve tenersi aggiornato chiedendo di volta in volta la posizione del dispositivo mobile in caso questo passi ad una nuova visited network.

Nel dettaglio i passi già descritti ma ora consideriamo 4G:



Wireless and Mobile Networks: 7-93

Si ha un tunnel tra PG-W nella home network e SG-W nella visited network e un secondo tunnel tra SG-W e stazione base nella visited network.

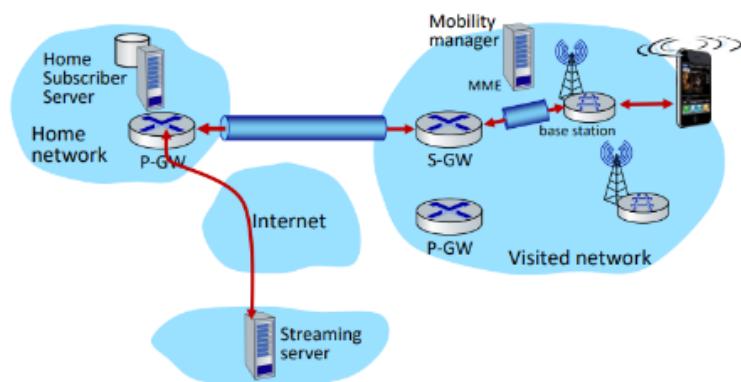
Il PG-W sta nella rete home perché è in questa rete che sta l'indirizzo permanente del dispositivo mobile, quindi i pacchetti che devono arrivare al dispositivo giungono anzitutto proprio nella rete home.

Chiaramente a livello di piano di controllo è importante che l'HSS sia aggiornata relativamente la posizione del dispositivo, per far sì che vengano eventualmente aggiornati i tunnel.

- Il dispositivo mobile comunica con l'MME locale attraverso il canale del piano di controllo con la BS
- L'MME usa l'informazione sull'IMSI del dispositivo mobile per contattare l'HSS della home network del dispositivo
 - Recupera informazioni per l'autenticazione, la cifratura e i servizi di rete
 - L'HSS nella home network sa ora che il dispositivo mobile è residente nella visited network
- La BS e il dispositivo mobile selezionano i parametri per il canale radio tra BS e dispositivo mobile nel piano di dati

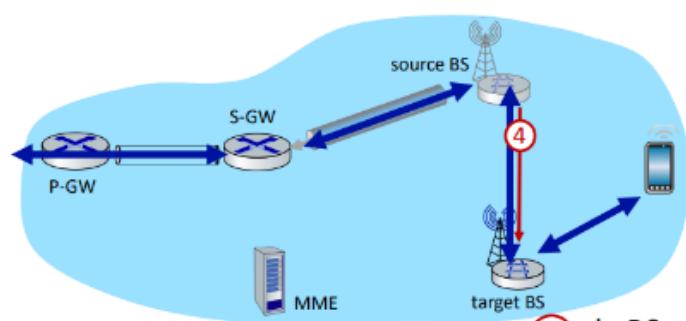
L'uso di due tunnel comporta anche una certa flessibilità: se infatti il dispositivo mobile cambia stazione base nella visited network il primo tunnel può rimanere anche lo stesso mentre il secondo deve solo cambiare estremità.

- **tunnel da S-GW a BS:**
quando il dispositivo mobile cambia stazione base, semplicemente cambia l'indirizzo IP dell'estremità del tunnel
- **tunnel da S-GW a home P-GW tunnel:**
implementazione dell'instradamento indiretto
- **tunneling attraverso GTP (GPRS tunneling protocol):** il datagramma del dispositivo mobile al server di streaming encapsulato utilizzando GTP all'interno di UDP, all'interno del datagramma



Handover tra stazioni base nella rete cellulare:

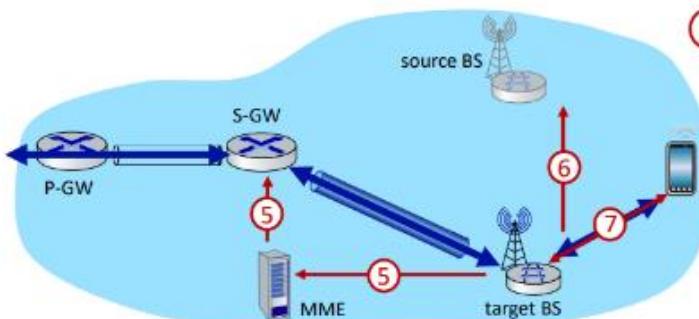
Handover tra BS nella stessa rete cellulare



- ① la BS corrente (source) seleziona il BS target, invia un messaggio di *Richiesta di Handover* al BS target
- ② la BS target pre-alloca slot temporali nel canale radio, risponde con un ACK con informazioni necessarie al dispositivo mobile per associarsi alla nuova BS
- ③ la BS source informa il dispositivo mobile della nuova stazione base
 - il dispositivo mobile può ora inviare attraverso la nuova BS - handover *appare* completo al dispositivo mobile

- ④ la BS source smette di inviare datagrammi al cellulare, inoltra invece alla nuova BS (che inoltra al dispositivo mobile attraverso il canale radio)

Wireless and Mobile Networks: 7- 10



- ⑤ la BS target informa l'MME che è la nuova BS per il dispositivo mobile
 - l'MME istruisce l'S-GW per cambiare l'estremità del tunnel alla (nuova) BS target

- ⑥ la BS target manda un ACK indietro alla BS source: handover completo, la BS source può rilasciare le risorse
- ⑦ i datagrammi del dispositivo mobile ora fluiscono attraverso il nuovo tunnel dal BS target all'S-GW

Wireless, mobilità: impatto sui protocolli di livello superiore

- logicamente, l'impatto *dovrebbe* essere minimo...
 - il modello di servizio best effort service model resta inalterato
 - TCP e UDP possono girare (e in effetti lo fanno) sul wireless e il mobile
- ... ma dal punto di vista delle prestazioni:
 - perdita/ritardo di pacchetti a causa di errori sui bit (pacchetti scartati, ritardi a causa di ritrasmissioni a livello di collegamento), e perdite di handover
 - TCP interpreta le perdite come congestione, riducendo la finestra di congestione senza motivo. Soluzioni:
 - recupero locale (infatti Wi-Fi implementa il trasferimento di dati affidabile)
 - consapevolezza del mittente dei collegamenti wireless (es. distinguendo le perdite causate dal wireless dalle perdite per congestione)
 - split connection (la connessione end-to-end divisa in una connessione da un capo all'access point wireless e una connessione dall'access point wireless all'altro capo)
 - traffico in tempo reale danneggiato dai ritardi
 - data la natura condivisa del canale wireless, occorre che le applicazioni considerino la larghezza di banda come una risorsa scarsa