

## Sommario

CAPITOLO 1: COMPUTER NETWORKS.....	5
Introduzione.....	5
Cos'è un protocollo?.....	6
La struttura "Esterna" del Network .....	6
I mezzi di trasporto fisici utilizzati .....	7
La struttura "Interna" del Network: il CORE.....	7
I ritardi: quanto ci mette davvero un pacchetto ad arrivare?.....	9
L'organizzazione Stratificata del Network .....	10
Sicurezza in un Network.....	11
CAPITOLO 2: APPLICAZIONI DI RETE.....	12
I concetti Principali da sapere.....	12
I paradigmi Architetturali .....	12
La questione Processo – Processo: Il Socket .....	13
I protocolli di Trasporto .....	13
I compiti di un protocollo Application Layer .....	14
Il protocollo HTTP: il più semplice di tutti .....	15
Ma quante connessioni TCP deve aprire? .....	15
La struttura dei messaggi di Richiesta e di Risposta.....	16
I Cookie .....	16
Il meccanismo di Web – Caching.....	17
Il Protocollo FTP .....	18
Posta Elettronica (SMTP, POP3, IMAP) .....	19
Qualche dettaglio in più: Il Protocollo SMTP .....	19
Tutto troppo facile: I protocolli di Accesso (POP3, IMAP) .....	20
DNS: Le Traduzioni di Internet.....	21
I servizi offerti dal DNS .....	21
La struttura gerarchica dei DNS server .....	22
Gli RR's: I Record del DNS.....	23
Le applicazioni P2P .....	24
I meccanismi del P2P : Dove si trovano le Risorse .....	25
Approfondimento: L'indice a Tabella Hash Distribuita.....	26
Trasferimento dei File: Il protocollo BitTorrent .....	26
CAPITOLO 3: RETI A CONNESSIONE DIRETTA.....	28
L'introduzione al Problema .....	28
I Servizi del Data-Link Layer .....	29
Error Detection & Error Correction .....	29
Un primo esempio: il bit di PARITA' .....	30

Parità Bidimensionale .....	30
Il metodo Checksum.....	31
Il CRC (Cyclic Redundancy Check).....	31
Reliable Data Transfer: nella top 10 degli argomenti più interessanti .....	32
RDT Protocol 1.0 : Il caso ideale.....	32
RDT Protocol 2.0 : I pacchetti che si corrompono.....	33
RDT Protocol 2.1 : I Numeri di Sequenza .....	34
RDT Protocol 2.2 : La versione NAK – Free .....	35
RDT Procol 3.0 : I Packet Loss .....	36
Trasferimento in PipeLine : Il Go – Back – N (GBN).....	36
L'ultimo Approccio : il Selective Repeat (SR) .....	38
Il protocollo PPP: trasporto a livello Data Link.....	40
I link ad Accesso Multiplo e relativi Protocolli .....	41
I protocolli a Partizione di Risorse : TDMA, FDMA e CDMA.....	42
I protocolli ad Accesso Casuale (1) : L'ALOHA .....	43
I protocolli ad Accesso Casuale (2) : CSMA e CSMA/CD .....	45
I Protocolli "Taking Turns" : I più fair sulla piazza .....	46
Le Reti Locali : Local Area Network (LAN).....	46
Gli indirizzi MAC .....	46
Ethernet(1) : Introduzione .....	47
Ethernet(2) : Il protocollo CSMA/CD .....	48
Ethernet (3) : Struttura del Frame .....	49
CAPITOLO 4 : LA NASCITA DELLE RETI SWITCHED.....	50
Dallo switch singolo alle reti VLAN .....	50
Link – Layer Switch .....	50
Interconnettere Switch .....	51
Rotta verso le VLAN (Virtual Local Area Network).....	52
Trasferimento Asincrono : le reti ATM .....	54
Caratteristiche del Circuito Virtuale VC.....	54
CAPITOLO 5 : LE INTER-RETI, IL NETWORK LAYER .....	55
Introduzione al Layer : Terminologia e Servizi Offerti .....	55
Com'è fatto un Router? .....	56
Il Viaggio del Datagram : Le Porte di Ingresso .....	57
Il Viaggio del Datagram : La switching Fabric .....	58
Il viaggio del Datagram : le porte di Uscita.....	58
Il protocollo IPv4 .....	59
IPv4 : Formato del Datagram .....	59
IPv4 : Necessità di Frammentare un Datagram, la MTU! .....	60
Indirizzamento nel protocollo IPv4.....	61

DHCP e NAT : “I router che implementano oltre il layer – 3” .....	64
Il Forwarding dei Datagram : I Parametri di Configurazione .....	66
I router che si scambiano messaggi : Il protocollo ICMP .....	67
Internet sta Collassando : IPv6 .....	68
L’intestazione del Datagram IPv6 .....	68
Le differenze da IPv4.....	69
Transizione da IPv4 a IPv6 : il Tunneling.....	69
Risoluzione degli Indirizzi IP : il protocollo ARP .....	70
CAPITOLO 6 : IL TRANSPORT – LAYER .....	71
Il comune Denominatore : il (De)Multiplexing .....	71
Demultiplexing nei protocolli Connectionless.....	72
Demultiplexing nei protocolli Connection – Oriented .....	72
Il protocollo UDP : User Datagram Protocol .....	72
L’Header del protocollo UDP .....	73
Il protocollo TCP : l’affidabilità in un colpo solo .....	73
Il formato del Segmento TCP .....	74
Gestione della Connessione TCP .....	75
Reliable Data Transfer nel protocollo TCP .....	76
Controllo di Flusso nel TCP .....	79
Il punto di Vista del Ricevitore .....	79
Il punto di vista del sender .....	80
CAPITOLO 7 : IL CONTROLLO DI CONGESTIONE .....	80
Introduzione alla Congestione .....	80
Controllo di Congestione nelle reti ATM.....	81
Controllo di Congestione nel TCP .....	82
Come fa il Sender a limitare il rate e a rilevare una congestione .....	82
L’algoritmo di controllo del Rate del sender .....	82
La fairness del TCP .....	84
CAPITOLO 8 : LA SICUREZZA IN RETE.....	85
Introduzione : Il bisogno della sicurezza .....	85
La Confidenzialità e la nascita della Crittografia.....	86
Crittografia a Chiave Simmetrica .....	87
Lo scambio della chiave .....	88
Crittografia a Chiave Pubblica.....	88
Integrità dei Messaggi.....	89
Il Digest, la “firma del messaggio” .....	89
Una prima soluzione di integrità : il Message Authentication Code (MAC) .....	90
Firma digitale utilizzando la Chiave Pubblica.....	91
La giusta Chiave Pubblica : I Certificati.....	92

Implementare la sicurezza in Rete .....	92
Sicurezza nel livello Applicazione : Pretty Good Privacy (PGP) .....	93
Sicurezza a Livello Trasporto : Secure Sockets Layer (SSL) .....	94
Sicurezza a Livello Network : Ipsec e VPN.....	96
Le mura della sottorete : i Firewall e gli IDS.....	98
I firewall a filtraggio di Pacchetto e gli Application Gateway.....	99
I paladini della giustizia : Intrusion Detection System (IDS).....	100
CAPITOLO 9 : Reti Wireless e Mobili .....	101
Le reti Wireless .....	101
Elementi costitutivi di una rete Wireless.....	101
Wireless LAN (Wi – Fi).....	102
Il protocollo MAC sulle reti Wi – Fi : CSMA/CA .....	103
Un nuovo meccanismo : Il Virtual Carrier Sensing.....	104
Formato del Frame, Mobilità e Power Management .....	105
La Mobilità in Internet .....	106
Cos'è la Mobilità dal punto di vista di Internet .....	106
L'idea di base e la possibile implementazione.....	106
Quale implementazione ha scelto Mobile IP? .....	109
Cosa succede ai protocolli di livello superiori? .....	110
Reti Wireless Senza Infrastruttura .....	110
Bluetooth (IEEE 802.15.1).....	110

**Se trovate errori, siete pregati di segnalarli e/o correggerli autonomamente, così da dimostrare a voi stessi che avete un ottimo spirito critico e che non ricevete passivamente le informazioni.**

**Enjoy,  
Angelo De Marco**

# RETI INFORMATICHE

## CAPITOLO 1: COMPUTER NETWORKS

Lezione 27/09 – 05/10

### Introduzione

La parola “Computer Network”, che indica una rete che connette dispositivi come laptop, desktop, è diventata datata visto il gran numero di dispositivi che si sono “aggiunti alla partecipazione” in rete (console, cellulari, IoT devices, ...).

Indichiamo dunque **ogni** protagonista che partecipa alla rete con la parola **Host** (perché ospitano un'applicazione che vuole comunicare), o **End System** (perché sono “ai bordi finali della rete”). Ogni host è collegato ad un altro mediante due classi di oggetti:

- **Communication Links:** rappresentano veri e propri collegamenti, in cavo o in fibra, e hanno un transmission rate, che si esprime in bit/secondo.
- **Packet Switches:** sono gli “smistatori di pacchetti”. Quando un host vuole mandare dati ad un altro segmenta il dato in pacchetti (aggiungendo anche un'intestazione al primo). Quando questi pacchetti arrivano a un Packet Switch, questo lo prende e lo manda da un'altra parte (su un altro packet switch, o alla sua destinazione finale). I due packet switches più d'uso comune sono i router, utilizzati nelle reti interne, e i link-layer switches, che di solito costituiscono la “rete d'accesso”, ovvero sono in diretta comunicazione con l'host che sta inviando il dato.

La sequenza di link e switch che un pacchetto attraversa dall'host che lo invia all'host che lo riceve è detto **route** o **path**.

Un qualsiasi Host deve accedere ad internet mediante un **ISP** (Internet Service Provider), che può essere la rete telefonica, la rete mobile, una rete universitaria/residenziale, ecc.

Un'altra visione di internet, oltre alla visione “infrastrutturale”, è quella che dice che **“Internet è un'infrastruttura che fornisce servizi alle applicazioni”**.

Le applicazioni di cui si parla sono le applicazioni distribuite, ovvero quelle che coinvolgono più End – System (host). L'applicazione gira sulla macchina Host, e pertanto c'è bisogno di “istruire” internet affinché possa spedire dati a un altro host che sta utilizzando la stessa applicazione. Per farlo si avvale della Internet Socket Interface, ovvero un'interfaccia che descrive le regole per inviare dati a un programma destinazione. In questa ottica, Internet “fornisce un servizio”, quello di poter scambiare dati tra le applicazioni distribuite.

## Cos'è un protocollo?

I protocolli definiscono il formato e l'ordine dei messaggi scambiati tra due entità comunicanti, e anche le azioni da eseguire conseguentemente alla trasmissione e/o alla ricezione del messaggio.

- **Nel linguaggio verbale** facciamo continuamente uso di protocolli (si chiede la parola prima di parlare, ci aspettiamo certe risposte a certe domande, e compiamo certe azioni quando qualcuno ci risponde in un certo modo).
- **Nel linguaggio dei network** i protocolli hanno la stessa idea di base, con la differenza che "gli attori" sono hardware e software. Qualche protocollo dovrà gestire il flusso di bit che fluisce sul cavo che collega due host, e qualche altro ancora dovrà gestire le regole con cui il router smista i pacchetti, ecc.

## La struttura “Esterna” del Network

Gli Host si dividono in due categorie: **Client** e **Server**. Il client è più specificatamente la “Macchina Client che ospita il Processo Client”, e ugualmente per il server. Solitamente i server sono macchine molto più potenti dei client, considerata l'enorme mole dei dati che memorizzano e distribuiscono.

Si parla di **rete d'accesso** per indicare tutto il complesso fisico che porta l'end sistem al primo router:

- **Connessione DSL:** è un tipo di accesso internet in cui la compagnia telefonica funge da *ISP*. Nelle case si ha un proprio Modem DSL che ha il compito di modulare i dati digitali ad alte frequenze. Infatti i segnali riguardanti i dati viaggeranno sullo stesso filo dove scorrono i segnali telefonici. Alle alte frequenze corrisponderanno i dati che provengono dall'end system, mentre alle basse ferquenze corrisponderanno i segnali relativi al telefono.

Quando il segnale raggiunge la CO (Central Office) della compagnia, smista il segnale verso il “Telephone Network” o verso “Internet” basandosi sulla frequenza.

Al ritorno, il DSLAM manda il segnale e uno **splitter** lo smista verso il modem DSL (che rimodula in dati digitali), o verso il telefono di casa.

- **Connessione Via Cavo:** è quella più utilizzata negli stati uniti, e sfrutta il cavo della TV. Dal centro dell'ISP c'è la fibra fino a un certo punto, dopodiché si passa a cavo coassiale che passa per un intera rete di case (anche centinaia). Questa rete è detta anche Hybrid Fiber Coax (HFC) proprio per i collegamenti usati, fibra e cavo coassiale.

All'interno della casa c'è bisogno di un modem che svolga pressoché le stesse funzioni del modem DSL, con la differenza che viene chiamato cable modem.

Il vantaggio della connessione via cavo è che ha in generale un rete maggiore che è però condiviso tra più persone. Se è solo una ad usarla, bene, altrimenti si andrà più lenti.

In entrambi i tipi di accesso alla rete, ci sono due cose da notare:

1. **Il bitrate è ASIMMETTRICO:** ciò vuol dire che c'è una velocità di trasmissione diversa verso DSLAM/CMTS (*upstream*), piuttosto che verso i modem DSL e cable (*downstream*). La velocità di downstream è maggiore poiché, pensandola alla maniera “Client-Server”, è il Server che di solito trasmette dati verso il client, e poche volte il viceversa.
2. **Il bitrate massimo potrebbe non essere raggiunto:** questo può esser dovuto a diverse cause. Una potrebbe essere la distanza dalla CO, o un'altra aver scelto un piano più economico, eccetera.

Le due connessioni precedenti fanno riferimento a un collegamento di tipo residenziale, ma nelle università, scuole, luoghi pubblici in generale si utilizza la LAN: Local Area Network.

Nell'approccio LAN, i due accessi più utilizzati sono tramite *Ethernet* o tramite *Wi - Fi*.

- **Ethernet**: gli end system sono collegati fisicamente a un ethernet switch, che poi è connesso fisicamente a un router e quindi alla rete internet. Con l'avvento dell'IoT questo tipo di approccio sta diminuendo.
- **Wi - Fi**: gli end system si collegano tramite wireless a un access point, che è poi collegato al network istituzionale (solitamente tramite ethernet).

Questi due approcci sono stati pensati per le reti istituzionali, ma si sono presto fatti strada nelle reti domestiche. Nelle case moderne, si sceglie di combinare uno dei primi due approcci (connessione via DSL o via cavo) con la connessione Wi-Fi e/o ethernet.

L'appuccio è il seguente: c'è un access point Wireless, detto base station, e c'è, ad esempio, un cable modem. C'è bisogno di un router che connetta la base station (ed eventuali PC wired) con il cable modem.

## I mezzi di trasporto fisici utilizzati

Connesso di qua, connesso di là, connesso a questo, connesso a quello, ma come sono connesse le cose?

- ✿ **Twisted-Pair Copper Wire ("DOPPINO TELEFONICO")**: sono fili di rame isolati e intrecciati tra loro, a formare spire. Più le spire sono strette, meno interferenza c'è durante la trasmissione del bit. A volte gruppi di spire sono avvolti in un cavo protettivo. Questo tipo di collegamento è quello utilizzato nel collegamento DSL.
- ✿ **Coaxial Cable ("CAVO COASSIALE")**: sono ancora due fili di rame conduttori, ma stavolta concentrici per consentire una velocità di trasmissione maggiore. È il tipo di collegamento che si trova nei collegamenti via cavo, ed è il tipo di collegamento utilizzato sui cable modem.

## La struttura "Interna" del Network: il CORE

Esistono due tipi di strutture interne che descrivono "la maniera in cui i dati viaggiano".

### PACKET SWITCHING

Il packet switching consiste nel dividere un lungo "messaggio" in più *pacchetti*. Questo pacchetti pacchetti sono smistati dal router. Esistono diverse tecniche con cui i router decidono quando e come mandare i pacchetti, e la più famosa è la **Store-and-Forward Transmission**. Il concetto è il seguente: "Trasmetti (forward) un pacchetto se e solo se ti è arrivato tutto (store)".

Immaginando di avere un pacchetto di  $L$  bit su un collegamento con un transmission rate di  $R$  bit al secondo, quello che succede è che questi sono trasmessi in  $L/R$  secondi.

Se ci mettiamo in una *situazione semplice* in cui abbiamo due end system connessi da un router, ecco quello che succede:

1. Al tempo 0 il sorgente inizia a trasmettere il primo pacchetto
2. Al tempo  $L/R$  il router ha completato la ricezione del primo pacchetto, e può iniziare a trasmetterlo.
3. Dunque, il destinatario riceverà il secondo pacchetto (per intero) dopo un tempo pari a  $2L/R$
4. **Se il router non aspettasse ma trasmettesse appena gli arriva il primo bit, non ci sarebbe questo ritardo di  $L/R$ .**

In generale, se un pacchetto deve viaggiare lungo  $N$  link (intermezzati da  $N-1$  router), questo impiegherà  $NL/R$  secondi ad arrivare. Vedremo che questo delay è necessario.

RAGIONAMENTO: Qual è il totale delay per  $P$  pacchetti che viaggiano su  $N$  link?

Ci sono due problemi nello store and forward:

- **Ritardi di Accodamento (queuing delays):** ovvero ritardi quando un pacchetto arriva effettivamente al router, ma ne sta trasmettendo già altri (appunto il pacchetto "si mette in coda").
- **Perdita del Pacchetto:** poiché il buffer interno del router (**output buffer**) deve essere di dimensione finita, può riempirsi. Se la coda di pacchetti si riempie e arriva a un pacchetto, si perde un pacchetto. Questo può essere quello che sta arrivando, o uno di quelli che è in coda.

Per decidere su quale dei link di "out" a cui il router è connesso va mandato il pacchetto, si utilizza un sistema basato sugli **IP ADDRESS** e sulle **FORWARDING TABLES** (Tabelle di routing). Il sorgente inserisce l'indirizzo IP del destinatario nell'intestazione del pacchetto, e il router consulta la sua tabella di routing per decidere su quale link dovrà mandare il pacchetto per avvicinarsi al destinatario. Questi meccanismi fanno parte dei *protocolli di routing*.

Nel caso del meccanismo packet switching, tutti condividono le risorse (buffer dei router, transmission router dei vari link). Ci sono casi in cui invece si vorrebbe una velocità e una "priorità garantita".

### CIRCUIT SWITCHING

Nel caso del circuit switching le risorse sono riservate fino al termine della comunicazione. È il caso delle telefonate: vogliamo un transmission rate garantito e vogliamo che nessuno si "intrometta" disturbando la comunicazione. Se il packet switching è un ristorante dove puoi non prenotare rischiando la fila, qui dovrà prenotare e aspettare di trovare posto, ma una volta avuto potrai immediatamente mangiare.

Un primo aspetto da notare è che un link può avere più circuiti. In questo caso si dovrà dividere in qualche modo il rate del link con i circuiti utilizzati. Se un link ha ad esempio un rate di 1Mbps, ed ha quattro circuiti (e quindi può ospitare quattro connessioni simultanee), dedicherà ad ognuna di loro 250Kbps. Anche se la velocità si "divide", nel caso del packet switching il link potrebbe intasarsi, e il queuing delay potrebbe essere problematico.

#### Come si dedica un po' di velocità al circuito?

- Un primo approccio è dividere in frequenza (Frequency Division Multiplexing FDM), ovvero si dedica una certa parte dello spettro del link a quel circuito. È il caso delle radio, che trasmettono in FDM a frequenza tra gli 88MHz e i 108MHz.
- Il secondo approccio è dividere in tempo (Time Division Multiplexing TDM), ovvero si permette di utilizzare alle connessioni l'intera larghezza di banda, dividendo il tempo in *FRAME* e dividendo ogni frame in *TIME SLOT*. In ogni frame, uno specifico time slot sarà dedicato alla connessione. L'obiettivo è garantire l'intera banda "a intervalli regolari". Il transmission rate è in questo caso calcolato come Numero di frame al secondo moltiplicato per il numero di bit assegnati allo slot.

Svantaggio: non si sfrutta il circuito creato durante i "tempi morti" (ad esempio quando le persone non parlano al telefono).

### I DUE A CONFRONTO

I sostenitori del circuit switching sostengono che il packet non si adatta bene ai sistemi in tempo reale, dove ritardi di lunghezza non prevedibili non sono ammessi, mentre i sostenitori del packet rispondono col fatto che è **largamente più efficiente**. Ecco due situazioni di esempio:

- Se ho un transmission rate di 1Mbps e 10 utenti, dividerò la banda del link dando 100Kbps ad ognuno. Devo limitare la velocità per tutti in ogni istante. Se gli utenti fossero attivi solo per il 10% del tempo, la probabilità che questi si trovino a trasmettere contemporaneamente è dello 0.0004 (bassissima). Primo vantaggio di efficienza: **permette la stessa banda a un numero superiore al triplo delle persone soddisfatte**.
- Se ho una banda di 1Mbps e ho 10 utenti in cui uno vuole spedire mille pacchetti da 1000 bit e gli altri 9 non vogliono far nulla (in totale  $1000 \times 1000 = 1\text{Mb}$ ). Mettiamo di poter mandare in TDM 1000 frame al secondo, e che ogni frame sia diviso in 10 slot (perché 10 sono gli utenti) da 1.000 bit. Per

mandare i mille pacchetti ci vorranno 10 secondi, perché ogni secondo ne vengono mandati 100 ( $1000/10$ ). Nel caso del packet switching invece sarebbe bastato un solo secondo.

CONCLUSIONE: La strada che si sta percorrendo è il packet switching.

## I ritardi: quanto ci mette davvero un pacchetto ad arrivare?

Se ho un pacchetto di  $L$  bit, e un end system deve trasmetterlo a un altro mediante un router, e i link hanno bitrate  $R$ , devo affrontare una serie di problematiche dovute a **ritardi** di natura fisica e non:

- **Il ritardo di Elaborazione:** è il tempo che impiega il router a riconoscere l'header del pacchetto e a consultare la sua tabella di forwarding per instradarlo correttamente. *Solitamente trascurabile, dell'ordine dei microsecondi.*
- **Il ritardo di Accodamento:** è un tempo variabile che il pacchetto deve attendere qualora sul link si stia già trasmettendo qualcosa, oppure se altri pacchetti sono arrivati prima di lui. È un tempo imprevedibile, con ordine *tra micro e millisecondi*.
- **Il ritardo di Trasmissione:** è il tempo che il router impiega a generare dal bit il corretto impulso fisico e a mandarlo sulla linea. È proprio questa necessità di generare impulsi che genera il bitrate del link. Dunque, il ritardo di trasmissione è esattamente  $L/R$ .
- **Il ritardo di Propagazione:** è il tempo che impiega un bit ad attraversare la linea, ed è funzione del mezzo trasmittivo utilizzato e della lunghezza dello stesso. In particolare, se  $d$  è la lunghezza e  $s$  è la velocità del segnale nel mezzo trasmittivo, il tempo di propagazione è pari a  $d/s$ .

Per quanto simili, **non bisogna confondere il ritardo di Trasmissione e il ritardo di Propagazione**: il primo è funzione della lunghezza del pacchetto e del bitrate del link, e coinvolge il tempo di conversione D/A dei bit del pacchetto. Il secondo non riguarda affatto il pacchetto, ma solo le *proprietà fisiche* del link (lunghezza e materiale). In generale essendo  $s$  prossima alla velocità della luce, il ritardo di Propagazione è trascurabile.

Complessivamente si introduce un **Total Node Delay (HOP)** che rappresenta il tempo con il quale il pacchetto arriva a destinazione, ed è pari alla somma dei quattro ritardi sopra elencati.

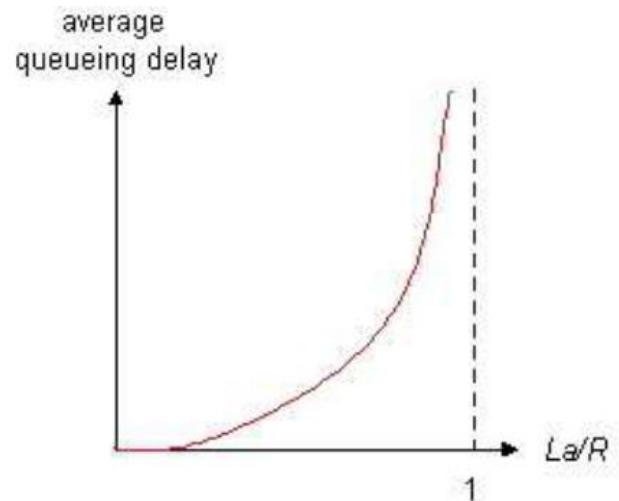
### PIU' PRECISIONE SUI RITARDI DI ACCODAMENTO

Essendo il ritardo di accodamento molto variabile è stato oggetto di numerosi studi. Cercando di stimare il comportamento della coda, immaginiamo di conoscere mediamente quanti pacchetti al secondo giungono al router, e indichiamo questo indicatore  $a$  (average packet/sec). Se ogni pacchetto immaginiamo sia di  $L$  bit, possiamo concludere che ogni secondo al router giungono  $L * a$  bit. Se c'è un bitrate di  $R$  bit/s, posso definire la mia Intesità di Traffico come  $La/R$ . In particolare:

- Se  $La/R > 1$ , allora arrivano più bit di quanti se ne possono mandare, e la coda diventa infinita (e così il ritardo di accodamento). Si noti che considerata la dimensione finita della coda questo comportamento si traduce in un'infinità di packet loss.
- Se  $La/R \leq 1$ , allora il ritardo dipende unicamente da "come" arrivano i pacchetti. Se infatti arrivano regolarmente, il ritardo potrebbe annullarsi del tutto, ma se arrivano *in burst* la coda si crea, e cresce tanto più velocemente quanto  $La/R$  è vicino ad 1.

Analizziamo due situazioni in cui poniamo  $La/R \leq 1$ :

- Immaginiamo che un pacchetto arrivi esattamente ogni  $L/R$  secondi. Ciò significa che vengono mandati  $R/L$  pacchetti al secondo ( $a = R/L$ ). Dunque  $La/R = 1$  ma per via della periodicità particolare dei pacchetti, considerato che il router



ne manda interamente via uno ogni L/R secondi, non ci sarebbe mai tempo di accodamento.

- Supponiamo invece che N pacchetti arrivino simultaneamente ogni  $N^*(L/R)$  secondi. Se succede questo,  $La/R = 1$  di nuovo, ma stavolta succede che dopo il primo non trova coda, mentre il secondo parte dopo L/R, l'ultimo partirà dopo  $(N-1)^*(L/R)$ !

In generale, quando  $La/R$  è un numero vicino ad 1, la coda si riempie sempre di più col passare del tempo (perché è probabile che ci siano dei "burst" periodici).

#### IL THROUGHPUT

Indicheremo con la parola **throughput** l'effettivo bitrate bit/s end-to-end, ovvero quanti bit arrivano al secondo effettivamente dall'Host A all'Host B. In particolare se ho un file di F bit, e impiego T secondi a farlo ricevere all'Host B, allora ho un throughput di  $F/T$  bit/s.

Il throughput dipende dal fatto che i link non hanno tutti lo stesso bitrate. I link che danno problemi e che diminuiscono la quantità di throughput sono detti **bottleneck link** (LINK A COLLO DI BOTTIGLIA).

Facciamo un paio di esempi:

1. Se ho un ruoter solo a dividere i due Host, e sul link HostA – Router ho un rate  $R_C$ , mentre sull'HostB – Router ho un rate  $R_S$ , i bit fluiscono in modo inaspettato rispetto a chi sta inviando i pacchetti. In particolare immaginiamo che sia  $R_C < R_S$  e che l'host B sia un server che vuole inviare un file all'host A, che è un client. In tempo  $L/R_S$ , il pacchetto arriva al router, ma ci vorrà un tempo  $L/R_C$  per giungere al client! **Comanda il rate più basso**, ovvero il throughput è  $\text{MIN}(R_S, R_C)$ . In questo caso, il link a collo di bottiglia è il link Client – Router.
2. Se avessi un client connesso a un network mediante un link, e avessi un server connesso allo stesso network mediante un altro network, la situazione può cambiare. Se infatti i link del network interno hanno R a ordini di grandezza maggiori, nessuno di loro sarà il collo di bottiglia e potremo di nuovo ricercarlo fra i due precedenti.

Ma può capitare che il network (*immaginato come un grande link di Rate R*) sia condiviso tra più utenti. Se più utenti usano contemporaneamente quel link, e R ha un'ordine di grandezza comparabile con i primi due rate, allora i colli di bottiglia possono essere all'interno del network. In particolare sarebbe da ricercare sul  $\text{MIN}(R_S, R_C, R/\text{numUtenti})$ .

Riassumendo, il throughput è il transmission rate del bottleneck link, che va però identificato!

## L'organizzazione Stratificata del Network

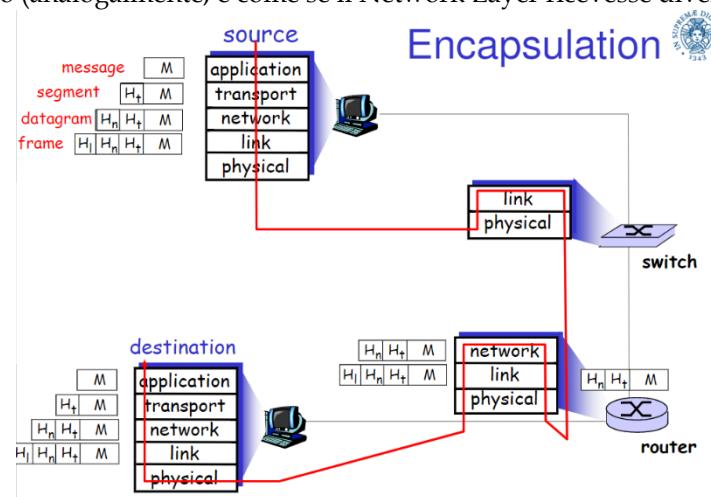
Perché è importante trovare un organizzazione, una suddivisione "a strati" (layer) del network? Questo favorisce la modifica dell'implementazione del servizio. Infatti stratificando l'implementazione posso modificare un singolo strato senza che gli altri ne risentano (a patto di mantenere gli "accordi" tra due layer adiacenti, e cioè i servizi che quello sopra di lui si aspetta utilizzerà ad esempio).

Si noti che questo è un **enorme vantaggio**, in quanto implementare diversamente una parte del sistema è **molto meglio di cambiare l'intero Sistema!**

L'idea è quella di suddividere le varie *funzioni (protocolli)* nei vari layer, e di conoscere i **servizi** che un layer offre al layer immediatamente sopra di lui (*service model*). L'insieme dei protocolli dei vari layer crea quello che si chiama **la pila dei protocolli (protocol Stack)** e in Internet consiste di protocolli appartenenti a 5 layer diversi:

- **APPLICATION LAYER:** è il livello in cui risiedono le applicazioni di rete e i loro protocolli (come HTTP). Sia i protocolli che le applicazioni sono interamente all'interno dell'End System. In particolare l'applicazione utilizza il protocollo per lo scambio di pacchetti da un'applicazione all'altra e prepara pacchetti speciali, che prendono il nome di **messaggi**.

- **TRANSPORT LAYER:** è il livello che garantisce che un messaggio arrivi effettivamente a destinazione. Si occupa di spezzettare il messaggio e aggiungere una breve intestazione. Ogni pacchetto preparato dal transport layer si chiama **segmento**.
- **NETWORK LAYER:** è dove risiedono il protocollo IP e i protocolli di routing. È noto anche come IP layer, considerato che l'IP è "la colla che tiene internet tutto unito". I pacchetti preparati in questo layer prendono il nome di **datagram**.
- **LINK LAYER:** è dove risiede il compito di trasportare i datagram, che prendono ora il nome di **frames**. All'interno troviamo i protocolli che gestiscono il trasporto del frame da un router all'altro, fino alla destinazione finale. Essendo i link diversi lungo il percorso, il frame subisce l'effetto di protocolli diversi lungo il percorso (analogalmente, è come se il Network Layer ricevesse diversi servizi dal Link Layer).
- **PHYSICAL LAYER:** più o meno come sopra, stavolta ci sono protocolli che gestiscono la trasmissione del singolo bit lungo il link.



## Sicurezza in un Network

Benché Internet sia stato designato per un gruppo di utenti *fidato*, col tempo si è esteso così tanto che si è resa inevitabile la creazione di alcuni soggetti pericolosi: i **bad guys**.

I Bad Guys sono quelle persone che ci attaccano con le tecniche più "fantasiose", cercando di cancellare i nostri dati, leggerseli, fingersi noi con qualcun altro, o cercano addirittura di eliminare un servizio che ci viene offerto. Gli attacchi che possono operare sono di varia natura:

### IL MALWARE: UN VERME CHE CRESCE

Se in qualche modo il bad guy riesce a infettare il nostro host con un *malware*, questo può **cancellare i nostri dati**, o installare uno **spyware**, che copia alcuni dati del nostro PC e li manda al Bad Guy grazie a Internet. Il **malware è capace di moltiplicarsi**, ed è capace di infettare altri Host partendo dal nostro, creando una rete di sistemi infetti che l'attaccante può utilizzare per altri tipi di attacchi (vedi DoS/DDoS). La rete di Host infetti creati si chiama **botnet**.

Un malware, moltiplicandosi, assume una tra due categorie:

- **Virus:** se per moltiplicarsi ha bisogno di un'iterazione con l'utente. Un esempio è quando cliccando in un link di una mail installiamo non volendo il malware.
- **Worm:** se invece non ha bisogno dell'iterazione dell'utente, ma ha solo bisogno che l'utente utilizzi un'applicazione network vulnerabile che è stata già infettata.

### GLI ATTACCHI AI SERVIZI: I DoS

Gli attacchi di tipo DoS, **Denial-of-Service**, servono all'attaccante per impedire che una certa rete di utenti fruisca di un servizio. Gli attacchi sono complessivamente assimilabili in tre categorie:

1. **Vulnerability Attack:** se l'attaccante conoscesse le vulnerabilità del server, potrebbe essere capace di mandare anche una breve serie di pacchetti per mandarlo in *stop* o addirittura in *crash*.

2. **Bandwidth Flooding:** l'attaccante invia un enorme numero di pacchetti al server per mandare la connessione in *congestione* (in pratica intasa il link). Per farlo, se il link ha un rate R, deve mandare R bit al secondo. Ma se un router vede "così tanti dati" inviati in burst da una singola fonte, può riconoscere l'attacco e bloccarlo. Per questo motivo questo tipo di attacco porta a un altro attacco, detto DDoS, **Distributed Denial-Of-Service**, in cui c'è un'intera rete di host infetti (botnet) che invia i pacchetti così che per il router sia difficile riconoscere che il link è sotto attacco.
3. **Connection Flooding:** si basa sul creare così tante connessioni TCP col server che i "legittimi utenti" non possono più istaurarne una.

#### *PACKET SNIFFING: I COCAINOMANI DELLA RETE*

I packet sniffers sono bad guy **passivi** che sono in grado di ricevere tutti i nostri pacchetti e farsene una copia. Questo può accadere in due modi:

- **Quando ci connettiamo in Wireless**, lo sniffer ha accesso a tutti i nostri pacchetti.
- **Se ci connettiamo in Wired**, lo sniffer può avere accesso a tutti i pacchetti che sono inviati in modalità *broadcast*.

Essendo elementi passivi, non c'è un vero e proprio modo di beccarli. L'unica soluzione è la crittografia, ovvero fare in modo che i messaggi contenuti nei pacchetti siano in qualche modo **cifrati**.

#### *I MASQUERADE: GLI ANTI-EROI*

Tramite un attacco di tipo **IP Spoofing**, una persona può manovrare il source address di un pacchetto. In questo modo è in grado di far capire al destinatario che il pacchetto viene da un'altra persona, e non dall'attaccante. Per difendersi da questo attacco c'è bisogno di introdurre un meccanismo che permetta al destinatario di verificare la reale identità del mittente (discorsi che ora non trattiamo).

## **CAPITOLO 2: APPLICAZIONI DI RETE**

*LEZIONE 05/10 – 20/10*

### **I concetti Principali da sapere**

Un'applicazione di rete **gira sull'end-system**, non sul core network. Il compito del network è quello di trasmettere *messaggi* da un processo che usa l'applicazione a un altro che usa la stessa applicazione da un'altra parte.

Ovviamente i due processi potrebbero essere sullo stesso end-system, ma non tratteremo questo caso (che non coinvolge ovviamente un network).

### **I paradigmi Architetturali**

Quando progettiamo un'applicazione di rete abbiamo anzitutto bisogno di poggiarci su una tra due *strutture*:

- **Client - Server** : è un tipo di struttura che coinvolge un'unica macchina Server e diversi Client. Il Server è quel processo che fornirà un servizio ai vari Client che genereranno richieste verso il primo. **I client non possono comunicare direttamente tra loro**, ma sempre e solo col server. Nonché il server appaia al client come uno, è difficile renderlo sempre disponibile, se fosse una sola macchina (un guasto al server implicherebbe un disservizio per tutti i client collegati). Per risolvere il problema sono nati i **dataCenter**, ovvero complessi di macchine Server (fino a centinaia di migliaia) che creano *virtualmente* una macchina Server unica.

Un'altra caratteristica, oltre ai ruoli ben definiti e la macchina Server che non è una *macchina normale*, consiste nel fatto che è preoccupazione del service provider procurarsi il server, che comporterà loro dunque costi non indifferenti.

- **Peer - To - Peer (P2P)** : è un tipo di struttura in cui non ci sono ruoli ben definiti, ma ogni host può sia ricevere richieste che rispondere a richieste. Per questo motivo gli host sono definiti *Peer*. Inoltre, **i vari host possono comunicare direttamente tra loro**, e da qui il nome della struttura. Per come è realizzato, questo meccanismo non richiede macchine particolarmente performanti come deve essere ad esempio un server, ma bastano i comuni laptop domestici. Una rete di peer che si interfaccia alla stessa applicazione ha diversi vantaggi: infatti una proprietà del sistema è la **self - scalability**, ovvero il numero dei Peer può crescere indefinitivamente senza restrizioni particolari.

Le applicazioni che si basano largamente su questo tipo di servizio sono le applicazioni *file sharing*, come bitTorrent, dove un peer scarica un file, oppure può contribuire a mandarlo a un altro peer.

È anche più economico, ma ha fallo di sicurezza maggiori dovute al fatto che la struttura è *decentralizzata*, e potrebbe non riuscire a raggiungere gli stessi livelli di performance che un grosso Server potrebbe offrire.

Il client - server si basa sul fatto che il server ha un indirizzo IP conosciuto da tutti. Esistono poi **architetture ibride** basate sui peer che sono “tracciati” dai server (nel senso che memorizzano informazioni in maniera più centralizzata).

### **La questione Processo - Processo: Il Socket**

È molto semplice pensare e dire che due Host si trasmettono pacchetti mediante la rete. Ma una volta raggiunto l'host, *come si direziona il pacchetto al processo che sta runnando quella specifica applicazione?*

La soluzione sta nell'interfaccia software **a socket**, ma andiamo con ordine.

Indipendentemente dal tipo di struttura, chiameremo **Processo Client** il processo che inizia una comunicazione, e **Processo Server** il processo che attende in attesa dell'inizio di una comunicazione.

Analogalmente, immaginiamo che ogni Host sia un'abitazione, e che ogni processo abbia una propria **porta**.

La funzionalità del Socket si basa proprio su questa analogia: per spedire processo - processo ci sarà bisogno di dargli due elementi:

- L'IP address dell'Host (stringa di 32 bit)
- **La porta del Processo** (stringa di 16 bit)

La destinazione di un messaggio preparato dal socket in questo modo non sarà l'host, non sarà neanche tanto il processo, ma sarà il socket su quell'Host associato a quel processo. Essendo il socket “direttamente correlato” al processo, si può parlare di comunicazione socket - to - socket.

*Di cosa si occupa fondamentalmente un Socket?*

Un socket è l'interfaccia tra l'application layer e il transport layer. Cioè significa che, in accordo al protocollo di trasporto scelto, il socket prepara il messaggio opportunamente. Similmente, il socket destinatario rimodula il messaggio per “spedirlo” all'application layer.

**L'utente che formula l'applicazione non ha il controllo di questa cosa, può solo scegliere quale protocollo di trasporto deve essere utilizzato!**

### **I protocolli di Trasporto**

Anzitutto, va detto che i protocolli di trasporto hanno “quattro parametri di valutazione”:

1. **Reliable Data Transfer (AFFIDABILITA')**: il protocollo ha questa proprietà se il pacchetto *arriva intero, integro* e, qualora siano stati mandati più pacchetti, questi arrivano nell'ordine corretto. Le applicazioni che richiedono questa proprietà sono tutti quelli che non tollerano data loss, ovvero tutte le applicazioni, ad esempio, di file transfer.
2. **Throughput**: il protocollo di trasporto potrebbe assicurare un certo throughput minimo. Ovvero assicurare l'utente che i pacchetti arriveranno al processo destinatario con una certa velocità media.

3. **Timing:** il protocollo può assicurare una certa velocità di trasmissione dei singoli bit. In particolare si può richiedere che un bit arrivi *con un certo ritardo massimo*. È l'esempio delle applicazioni telefoniche a chiamata, dove non si vuole che la voce arrivi in ritardo, creando fastidiose sovrapposizioni.
4. **Security:** argomento via via più centrale al mondo d'oggi. Il protocollo che ha questa proprietà assicura che i pacchetti arrivino senza possibilità che qualcuno possa *sniffarli* in maniera chiara (è l'esempio della crittografia).

I protocolli di trasporto che mette a disposizione Internet sono soltanto due:

### **TCP (Transmission Control Protocol)**

Il protocollo TCP gode di due caratteristiche principali:

- È affidabile (nel senso visto sopra).
- È **connection - oriented**, ovvero richiede che prima dell'inizio dello scambio di messaggi si stabilisca una connessione tra processo client e processo server. Quando questa fase (di handshaking) è chiusa, i due processi hanno preparato delle strutture dati per gli scambi, e si dice che si è aperta una connessione TCP. È compito dell'applicazione chiudere la connessione TCP.

Inoltre il TCP ha un altro paio di chicche secondarie, per nulla male:

1. È **flow - control**, ovvero il ricevitore è in grado di dire al trasmettitore "trasmetti di meno perché mi stai riempendo troppo velocemente" e in questo caso il protocollo abbatte la velocità di trasmissione.
2. È **congestion - control**, come sopra solo che si abbatte la velocità di trasmissione in caso di congestione su uno dei router nel percorso.

Anche se è affidabile, il TCP non assicura nessuno degli altri tre parametri (timer, throughput o security). Tuttavia, essendo la sicurezza diventata fondamentale, spesso gli si affianca un *estensione SSL (Secure Sockets Layer)*, che è un protocollo di sicurezza che viene eseguito a livello dell'applicazione e include la crittografia dei dati e l'autenticazione di tipo end – point (con cui il destinatario può assicurarsi su chi sia il vero mittente).

Astraendo un percorso, i dati dell'applicazioni passano al socket SSL, che poi passa i dati crittografati al socket TCP, che trasmette al socket TCP destinatario, che manda al socket SSL a decodificare il messaggio, da mandare al processo.ù

### **UDP (User Datagram Protocol)**

A differenza del protocollo TCP, non è affidabile, e non richiede una connessione iniziale tra processo client e processo server. Come il TCP, non assicura niente tra gli altri parametri.

FA SCHIFO? Non necessariamente, può essere (ed è) utilizzato dalle applicazioni che tollerano data loss (applicazioni di conferenza come Skype).

In generale nessuno dei due protocolli assicura throughput o timing, il che sarebbe una tragedia. Questi problemi devono essere risolti **progettando bene l'applicazione!**

## **I compiti di un protocollo Application Layer**

Un protocollo appartenente alla fascia dell'application layer deve specificare:

- **I tipi di messaggi scambiati:** ad esempio deve specificare che si tratti di un *messaggio di richiesta*, o di un *messaggio di risposta*.
- **La sintassi dei messaggi scambiati:** quanti campi compilare e come compilarli
- **La semantica dei messaggi scambiati:** ovvero specificare cosa significa ogni campo

- Le **regole** da seguire quando una persona vuole inviare un messaggio (e cosa deve fare dopo averlo inviato) e quelle da seguire quando una persona riceve un messaggio (e cosa deve fare dopo averlo ricevuto).

## Il protocollo HTTP: il più semplice di tutti

Il protocollo HTTP (**Hyper-Text Transfert Protocol**) è il protocollo dedicato a una delle applicazioni di rete più comuni: il Web.

La particolarità che rende il Web un'applicazione così comune e interessante è l'essere on demand: tutti vedono quello che vogliono *al momento che vogliono*, e non sono costretti ad "aspettare" (un po' come quando in radio si aspetta una certa canzone, non puoi decidere di farla trasmettere subito).

Il protocollo HTTP si basa sul rapporto tra un Client, che fa un certo numero di richieste, e un Server, che risponde alle richieste. Desiderio del Client è quello di ricevere **Web File**, che consistono di oggetti: documenti HTML e/o file audio, video, immagini, ...

Il protocollo si appoggia sul TCP, e pertanto è necessario aprire prima una connessione di questo tipo. Dopodiché il client formula un *messaggio di richiesta*, chiedendo cosa vuole, e il server risponde.

L'oggetto che il client può chiedere è sempre nella forma **IndirizzoServer | PathOggetto** e, nel Web, gli indirizzi del Server sono fissati.

In una situazione tipica, il Client chiede sulla connessione TCP un documento HTML, e dopo che il Browser ne fa il parsing scopre che ha bisogno di altri oggetti (è pratica comune che all'interno dei documenti HTML ci siano riferimenti a diversi oggetti ipertestuali). Allora il client si mette a chiedere al server questi oggetti, uno per uno.

### Ma quante connessioni TCP deve aprire?

Ci sono due approcci possibili:

1. **"A connessione non persistente"**, ovvero si deve creare una nuova connessione TCP per ogni oggetto che si vuole richiedere.
2. **"A connessione persistente"**, ovvero che si mantiene la connessione aperta e si chiude secondo altri criteri (ad esempio il server può decidere di chiudere la connessione se non riceve richieste per un po' di tempo).

*La connessione persistente è meglio.*

Infatti, usando la non persistente, e chiamato **Round - Time Trip (RTT)** il tempo che impiega un piccolo pacchetto a fare il percorso client – server – client, vediamo cosa succederebbe:

- Il client apre la connessione TCP, e il server chiude l'handshake, aprendo ufficialmente la connessione. Durante questo processo un piccolo segmento TCP è andato dal client al server, e un altro è andato dal server al client: 1 RTT.
- Il client formula il messaggio di richiesta, e il server risponde col messaggio di risposta più l'oggetto richiesto. Essendo i messaggi di richiesta e di risposta piccoli possiamo dire che si consuma 1 RTT + Il tempo di trasmissione del file.
- Il server chiude la connessione TCP (la connessione è effettivamente chiusa solo se si è sicuri che il client abbia effettivamente ricevuto il messaggio, il TCP è affidabile).

Dunque abbiamo utilizzato 2 RTT + Tempo, e questi utilizzeremo **per ogni oggetto richiesto**. Se la connessione fosse persistente, è solo il primo oggetto che ci mette questo tempo, mentre gli altri impiegano tutti **1 RTT + Tempo**.

È ragionevole pensare che una connessione persistente sia meglio, e che quella non persistente esistesse perché prima il numero di oggetti referenziati in un documento HTML erano pochissimi o addirittura assenti.

**Il protocollo HTTP è STATELESS:** non conserva alcuna storia delle richieste e vede ognuna di queste come "nuova"!

## La struttura dei messaggi di Richiesta e di Risposta

*Messaggi di Richiesta:*

1. **Request Line:** *Method | PathObject | VersioneProtocollo*

In questa linea si indica il metodo (GET, POST, HEAD, PUT, DELETE) con cui si vuole ottenere l'oggetto, il path dell'oggetto stesso, e la versione HTTP utilizzata.

2. **Header Lines:** *Keyword | ':' | Value*

Sono linee che trasmettono informazioni aggiuntive al server. Tra le più comuni ricordiamo:

- *Host:* indica l'host in cui si trova l'oggetto che si vuole recuperare. Potrebbe sembrare superfluo inserire questa linea a causa della connessione TCP già aperta verso "dove voglio comunicare", ma in realtà servirà in seguito.
- *Connection:* indica al server se al termine della gestione di questa richiesta potrà chiudere la connessione TCP o meno (siamo in un approccio di default persistente).
- *User - Agent:* indica il browser che si vuole utilizzare per leggere l'oggetto. Infatti, il server potrebbe conservare diverse versioni dell'oggetto a seconda dell'user - agent che lo richiede.
- *Accept - Language:* si comunica al server che si avrebbe piacere di ricevere l'oggetto in una determinata lingua e, se non esiste, si accetta la lingua di default.

3. **Una Linea Vuota**

4. **Entity Body:** si mettono qui alcune cose che si vogliono trasmettere al server (ad esempio finiscono qui i campi compilati di una form).

*Messaggi di Risposta:*

1. **Status Line:** *VersioneProtocollo | CodeStatus | StatusDescription*

Si indica qui la versione protocollo HTTP che ha utilizzato il server, un codice che rappresenta il risultato dell'operazione (si ricorda il celebre 404 FileNotFound) e una breve descrizione dell'errore/status.

2. **Header Lines:** *Keyword | ':' | Value*

È il gruppo di righe con cui il server comunica al client informazioni aggiuntive (che tipo di server ha mandato la risposta, se la connessione sarà chiusa al termine dell'operazione, la data dell'ultima modifica dell'elemento, ecc.)

3. **Una linea Bianca**

4. **Entity Body:** è il campo che verrà riempito con il file effettivamente richiesto!

## I Cookie

La tecnologia dei Cookie ha permesso al protocollo HTTP di diventare **stateful**. In generale grazie ai cookie un server può performare certe specifiche azioni rivolte allo specifico utente che fa la richiesta HTTP. In particolare, la tecnologia Cookie si basa su quattro componenti:

- *L'aggiunta di una linea Header Cookie:* nei messaggi di richiesta
- *L'aggiunta di una linea Header Set - Cookie:* nei messaggi di risposta

- **Un file Cookie** che viene analizzato e modificato dal browser, che si basa sul creare linea del tipo *NomeHostServer | IDCookie*
- **Un database back - end** che consulta il server quando deve inserire un record (quando un nuovo ID viene generato), o modificarlo (aggiungendo informazioni).

Il meccanismo si basa su un concetto molto semplice: **la prima volta** che entro in un sito mi viene associato un ID, e questo ID si riferirà da ora in poi a un record che *collezione informazioni* per scopi diversi: messaggi di benvenuto, evitare di ricompilare alcuni campi già compilati in precedenza, mostrare pubblicità interessante, ecc.

Nello specifico:

- La prima volta il Server risponde aggiungendo una linea Header *Set - Cookie: ID* dove comunica l'ID al client. Il browser vede quell'ID, e crea una nuova linea nei file Cookie dove annota la coppia *Hostname | ID*.
- Le volte successive nei messaggi di richiesta si potrà inserire l'header *Cookie : ID* per far capire al server che si tratta di me e questo potrà rispondere *performando inoltre azioni rivolte a me* e alla storia delle mie richieste consultando il back - end DB. Questo permette ad alcuni siti di e - commerce (come Amazon) di gestire la modalità **Paga - In - Un - Click** (ovvero grazie al record salvato nel database l'utente non dovrà reinserire nome, cognome, indirizzo, numero carta, ecc.).

I cookie sono anche oggetto di controversie *moral* in quanto sono visti come invasione di privacy. Questo è dovuto al fatto che grazie ai cookie **i siti possono imparare molto di noi**, e comportarsi di conseguenza, o addirittura vendere i nostri dati a terze parti (per pararsi il culo i siti oggi mettono l'avviso "Questo sito utilizza i cookie, accetti?").

## Il meccanismo di Web - Caching

Il meccanismo di web - caching aiuta gli utenti ad essere serviti **più velocemente**. Infatti essendo il Web pensato on - demand, l'utente finale si "scoccia subito" di aspettare, e può decidere di interrompere l'utilizzo del servizio.

Introducendo questo meccanismo, che prende il nome di **Proxy Server**, si inserisce un "host" tipicamente nella stessa rete di accesso del Client, in modo tale che se il Proxy riesce a soddisfare la sua richiesta, l'utente si vedrà soddisfatto in un tempo *trascurabile*.

Analizziamo nel dettaglio cosa succede:

1. Il client fa una richiesta HTTP in cui richiede un oggetto. Anche se il client indica il server come destinatario, il messaggio passa prima dal Proxy Server. Ciò vuol dire che **la connessione TCP che viene creata si crea col Proxy, e non col Server originale**.
2. *Se il Proxy ha l'oggetto desiderato*, allora è lui che spedisce l'oggetto al client (in tempi come abbiamo detto brevissimi).
3. *Se il Proxy non ha l'oggetto desiderato*, **apre una connessione TCP con il Server originale** e costruisce un messaggio di richiesta per questo.
4. Il server risponde, inviando l'oggetto desiderato. Il Proxy Server ne fa una copia e restituisce l'oggetto al client.

Si noti che durante il processo **il Proxy Server è contemporaneamente Client e Server**: Server nei confronti del Client originale, Client nei confronti del Server originale.

I vantaggi introdotti da questo sistema sono principalmente due. Anzitutto diminuisce la velocità di risposta (che è l'obiettivo che volevamo raggiungere). Ma c'è un effetto molto più importante: **riduce il traffico**

**generale sull'Internet.** Infatti, se tutti usano questo meccanismo è probabile che tante richieste siano risolte *in locale* e che dunque il traffico sulla rete si riduca alle richieste che il Proxy non è riuscito a gestire.

#### *Un esempio Pratico*

Immaginiamo di trovarci in una rete istituzionale, con rate 100Mbps in LAN. Dall'ethernet si è connessi al router "di internet" (cioè il primo nel core) con un link di rate 15 Mbps.

Caso vuole che la media delle richieste sia 15 al secondo, e che ogni oggetto richiesto pesi in media 1Mbps. Assumiamo infine che il tempo affinché il "primo router" ottiene la risposta del server passino 2 secondi. Da queste premesse, possiamo trarre alcune conclusioni:

1. **Se tutto resta così**, il traffico sulla rete LAN è del 15%, e dunque si introduce un ritardo trascurabile. Il link d'accesso (quello che connette ethernet e router) subisce un traffico del 100% (che sappiamo introdurre un ritardo di accodamento senza limiti). Questo non va affatto bene, perché aumenta il tempo di risposta (attualmente pari a 2 secondi + ritardoLAN) di un fattore di ordine di grandezza dell'ordine dei minuti, se non di più!
2. **Se aumentassimo la capacità del link di accesso**, diciamo da 15Mbps a 100Mbps, il traffico cala drasticamente dal 100% al 15%. Questa è una buona cosa, perché allora sia il ritardo LAN che il ritardo d'accesso si riducono ad essere trascurabili. La cosa brutta è il costo di upgrade della capacità del Link!
3. **Se introduciamo in LAN un Proxy Server**, e stimiamo che il 40% delle richieste è soddisfatto da quest'ultimo, le cose migliorano. Infatti il 40% delle richieste sarà soddisfatto in tempo trascurabile (si subisce solo il ritardo LAN che è molto piccolo), mentre il 60% delle richieste sarà soddisfatto in un paio di secondi (conta solo il ritardo di internet). Questo succede perché si è ridotto il traffico sul Link di Accesso, che scende dal 100% al 60% (sotto l'80% non si creano tempi di attesa lunghi).

In media si aspetterà un tempo pari a  $0.6 * (2 + \text{ritardoD'accesso} + \text{ritardoLAN}) + 0.4 * (\text{ritardoLAN})$ , e questo tempo è < **1.3 secondi**, che è un tempo medio ancora minore della soluzione che coinvolgeva il costo di upgrade del link.

L'istituzione dovrà comprare e installare un Proxy Server, ma costa molto di meno di aumentare la capacità di un Link!

L'introduzione di Proxy Server, o meglio di *tanti* Proxy Server, porta dunque indubbi vantaggi. Introduce però un altro problema: **la copia dell'oggetto tenuta in Cache potrebbe essere obsoleta!**

Per risolvere questa problematica è stato introdotto il meccanismo del **conditional GET**, ovvero il client può utilizzare il fatto che il Server risponde col campo *Last - Modified*: per dire al server Proxy che vuole una copia dell'oggetto più o meno recente. In particolare il Client può inserire nella richiesta HTTP un campo *If - Modified Since : Data* con cui dice "Proxy, mandami l'oggetto che hai tu a meno che il server non ti dica che il file è stato modificato da *Data* in poi".

Il Proxy allora ribalta la richiesta sul server, che risponde "si o no", in particolare risponde con codice 304 (*Not modified*) o col codice 200 (OK). Se il server risponde positivamente (ovvero comunica al proxy che il file è stato modificato dopo quella data), il proxy farà una nuova richiesta e si farà mandare la copia aggiornata (trattenendo una copia per lui), altrimenti manderà ciò che ha già.

## Il Protocollo FTP

Il protocollo FTP è il "predecessore" dell'HTTP, ed è ancora qualche volta utilizzato. In particolare si basa su due concetti fondamentali:

- Una **user Interface** che permette all'utente di inserire comandi per navigare all'interno di un file system (quello del server a cui si collega). L'interfaccia permette all'utente di inserire ASCII!
- Due **comandi**, PUT e GET, per fare upload o download di elementi dal server stesso.

La connessione tra client e server richiede due connessioni TCP.

1. **Una connessione di Controllo**, che serve ad autenticare l'utente e a permettere di inserirgli i comandi, che si apre quando l'utente all'inizio deve autenticarsi.
2. **Una connessione di Dati**, che serve a trasmettere / ricevere dati da / verso il server, e che si apre quando l'utente digita un comando.

La presenza di questa duplice connessione permette all'utente di inserire comandi anche mentre ci sono trasferimenti in corso. Inoltre, il protocollo diventa **stateful** perché la connessione di controllo permette di trasmettere sempre informazioni circa la current directory, o circa l'utente che si è autenticato prima.

## Posta Elettronica (SMTP, POP3, IMAP)

La posta elettronica è una delle applicazioni più antiche di Internet, ed è ancora una **delle più utilizzate**. La posta è un asincrono mezzo di comunicazione, in cui il mittente manda il messaggio senza sapere o preoccuparsi di cosa stia facendo il destinatario, e non ci sono vincoli sui tempi di risposta (infatti il mittente potrebbe inviare un messaggio a cui potrebbe aspettarsi o meno una risposta, e anche se se l'aspetta sa che la risposta non è certa).

I componenti principali di questo sistema sono due:

- Gli **user agent** sono i dispositivi che permettono all'utente di comporre / inviare / rispondere / visualizzare il messaggio. Un esempio di user agent sono Gmail, Outlook, ecc.
- I **mail server** sono i "contenitori", e sono i veri attori del protocollo dello scambio dei messaggi. In particolare, ogni mail server ha:
  - Una **message queue** dove si accodano i messaggi che sono inviati attraverso quel mail server e / o dove ritornano i messaggi che non sono stati inviati con successo (a causa, ad esempio, di guasti sul mail server destinatario). Se un messaggio non è inviato con successo per diverso tempo di solito si annulla il messaggio, notificando il mittente dell'accaduto.
  - Più **mailbox**, una per ogni possibile destinatario che si serve da quel Mail Server. Nella mailbox si accodano i messaggi rivolti a quello specifico destinatario.

Le cose, bypassando la conoscenza del principale protocollo (SMTP), funzionano più o meno così:

1. Il mittente compone il messaggio e clicca "invia" mediante il suo user agent.
2. Lo user Agent invia il messaggio nella message queue del Mail Server del mittente.
3. Il Mail Server del mittente **si comporta da client** e comunica col Mail Server del destinatario, che **si comporta da Server**.
4. Il messaggio è inviato sulla mailbox del destinatario.
5. Il destinatario invoca lo user agent per leggere i messaggi della sua mailbox, tra cui quello inviato ai passi precedenti!

### Qualche dettaglio in più: Il Protocollo SMTP

**SMTP (Simple Mail Transfer Protocol)** è il protocollo utilizzato per "spostare" messaggi dalla message queue di un Mail Server a una mailbox di un altro Mail Server.

Il Mail Server è **SMTP client** se invia il messaggio, **SMTP server** se riceve.

Il protocollo SMTP si basa su tre fattori:

- Una fase di handshaking iniziale

- Il trasferimento del messaggio su una connessione TCP. **Attenzione: è richiesto che il messaggio sia ASCII 7-Bit.** Questa limitazione è grossa, perché apparentemente non permette agli utenti di allegare file multimediali, se non codificati in ASCII. Questo introduce la necessità di un sistema di codifica del file multimediale, e di un sistema di decodifica dello stesso file.
- Una fase di chiusura

C'è da dire che la connessione TCP aperta tra i due Mail Server **non coinvolge attori intermedi**. Ciò significa che se il messaggio non viene recapitato resta sul Mail Server del mittente, e non si sposta su niente di intermedio. Nello specifico:

- Il client SMTP apre una connessione TCP (porta 25) con il server SMTP
- Comincia una fase di Handshaking, il server comincia con 220 <HostnameServer> e il client usa il comando *HELO <HostnameClient>*. Il server chiude la fase di handshaking, si può iniziare a comunicare.
- Il prossimo passo è chiarire mittente e destinatario, e il client lo fa coi comandi *MAIL FROM, RCPT TO* (**il server risponde ad ogni comando**).
- Dopotiché si dà un comando *DATA* e si manda il messaggio, che termina con una linea in cui c'è unicamente un '.' Il messaggio può avere delle linee di intestazione in cui si specificano, tra le altre cose, mittente destinatario e oggetto del messaggio. Se queste sono presenti, devono essere separate dal messaggio vero e proprio con una linea bianca.
- A questo punto il client può ricominciare da *MAIL FROM* e inviare altri messaggi, **sfruttando la connessione TCP persistente**, oppure dare un *QUIT* e aspettarsi dunque che il server chiuda la connessione TCP.

Il protocollo ha alcune caratteristiche comuni con l'HTTP, come l'utilizzo di connessioni TCP persistenti, o come il fatto che entrambi si basano sul "trasferire oggetti" da un posto a un altro. Tuttavia ci sono delle differenze importanti:

1. Il protocollo HTTP è di tipo **pull**: il client vuole dati dal server ed è chi vuole ricevere che inizia la connessione. Viceversa il protocollo SMTP è di tipo **push**: un mail server vuole inviare messaggi ad un altro mail server ed è chi vuole inviare che inizia la connessione.
2. Il protocollo HTTP non impone il vincolo dell' ASCII 7-Bit, evitando appunto che file multimediali debbano essere codificati in quel formato.
3. Il protocollo HTTP impone una richiesta per ogni oggetto, in SMTP posso includere tutti questi oggetti in un unico messaggio.

## Tutto troppo facile: I protocolli di Accesso (POP3, IMAP)

Nasce un problema. Il protocollo SMTP è di tipo Push, ovvero serve a inviare robe sul mail server del destinatario.

### Ma come fa il destinatario (tramite il suo user agent) a ripescare i messaggi dalla sua mailbox?

Una possibile soluzione è installare il Mail Server direttamente sulla macchina del destinatario. In questo modo non c'è bisogno di protocolli per accedere alla mailbox perché posso accedervi localmente. C'è però un problema: questo richiederebbe:

- Che ci sia un mail server su ogni possibile destinatario (spreco di spazio, tempo, e soldi)
- Che i computer dei destinatari **siano sempre accessi e connessi**. Se non è così, questi non possono ricevere alcuna mail perché il server non risponde all'apertura della connessione TCP sulla porta 25.

Ecco perché vengono pensati i protocolli di accesso alla mail, che si basano sul fatto che il Mail Server sia non nella stessa macchina dello user, sia sempre acceso, e ci si possa connettere.

## PROTOCOLLO POP3

Il protocollo **POP3(Post - Office Protocol - Version3)** si basa su dei semplici meccanismi:

- Aperta una connessione TCP sulla porta 110, il Mail Server chiede al Client di autenticarsi (**fase di autorizzazione**)
- Dopo essersi autenticato, il client può leggere la lista dei messaggi, scaricarli, e marcare i messaggi per "delete", ovvero segnalare la volontà di cancellarli o meno (**fase di transazione**)
- Quando il client fa quit, il server chiude la connessione TCP e rende permanenti le delete marcate (**fase di update**).

Il meccanismo di marcatura introduce due possibilità di lavoro: *download and delete* e *download and keep*. Il download and delete è sicuramente un'opzione considerabile visto il risparmio di spazio a cui si va incontro, ma download and keep permette di **non partizionare troppo le mail**. Se uno user utilizza diversi dispositivi, con lo modalità download and keep troverà le stesse mail su ogni dispositivo. Se fa download and delete troverà su ogni dispositivo solo quelle che ha deciso di scaricare su quel dispositivo!

*POP3 è stateless*, conserva alcune informazioni di stato (accesso, ricorda i messaggi che sono stati marcati per la delete), ma **non ricorda alcuna informazione tra sessioni POP3 differenti**.

Diverso è il protocollo **IMAP**, che permette anche una suddivisione in categorie e subdirectory dei vari messaggi.

## DNS: Le Traduzioni di Internet

Comunemente riusciamo a identificarci mediante diverse procedure: ci chiamiamo per Nome e Cognome, al più aggiungiamo la nostra data di nascita o il luogo di provenienza, ma non stiamo lì a dettarci i nostri codici fiscali.

Anche gli Host, così come gli umani, hanno dei nomi, gli **hostname**. Questi sono alfanumerici come i nostri nomi, ma *solo molto difficili da gestire per i router*, per cui sarebbe molto difficile gestire ad esempio la tabella di forwarding basandosi su lettere frutto anche di fantasia rispetto ai numeri.

Nella realtà ogni Host non è dunque chiamato con il suo hostname da un router, ma con il suo **indirizzo IP**, una sequenza di 32 bit scritta in notazione decimale puntata per semplicità (192.10.20.30 ogni byte separato da un punto e trasformato a decimale).

I router, dunque, vogliono leggere questo indirizzo IP perché, *scannerizzandolo* da sinistra a destra, possono intuire un certo tipo di ordinamento gerarchico, e quindi capire in quale rete si trova l'host (nella rete di reti che è internet). Quindi ai router viene molto comodo analizzare questi byte a gruppi per compiere i suoi protocolli di routing, ma gli umani certo non si ricordano di queste sequenze di byte / numeri decimali.

Per questo motivo nasce l'applicazione **DNS, Domain Name System**, che lega il linguaggio degli umani a quello dei router, operando una traduzione NomeAlfaNumerico – IndirizzoIP.

### I servizi offerti dal DNS

Come detto sopra, il DNS offre un mapping che traduce l'hostname nell'indirizzo IP corrispondente. L'applicazione DNS, anche se non ce ne accorgiamo, risulta tra quelle utilizzate di più, perché utilizziamo nomi mnemonici ogni volta, e non scriviamo mai un singolo indirizzo IP.

Se avete però pensato a ciò che scriviamo sul browser, beh, quello non è l'unico utilizzo. Infatti anche formulando richieste HTTP verso server, inserisco nel messaggio di richiesta l'*Hostname*: è il servizio DNS che fa arrivare la richiesta nel posto giusto, traducendo nell'indirizzo IP corrispondente! Il protocollo di trasporto utilizzato è **UDP**, perché preferiamo la *responsiveness* all'affidabilità del servizio.

Ci sono poi altri servizi utilissimi messi a disposizione da questa applicazione:

- **Host Aliasing:** definito *nome canonico* l'originale hostname che il DNS mette in correlazione con l'indirizzo IP dell'host, chiameremo *Alias* un altro nome mnemonico, ancora più mnemonico per noi comuni mortali. In particolare, preferiremo www.unipi.it a info.server1.unipi.it. Il DNS permette anche questo servizio (vedremo come lo implementa in seguito). Quando l'utente digiterà un alias, questo sarà tradotto nel nome canonico, e poi sarà il nome canonico ad essere tradotto nell'indirizzo IP (**necessario un passaggio in più**).
- **Mail Aliasing:** i Mail Server hanno un hostname, ma servono tantissimi host diversi. Ecco perché quando vogliamo rivolgerci a bob inseriamo bob@yahoo.mail, e non server-1@yahoo.mail. In genere si crea un alias per ogni host servito, e tutti vengono tradotti nel nome canonico del mail server che gestisce le loro mailbox!
- **Controllo del Traffico:** il servizio DNS serve a **costruire una rete di server MIRROR**, ovvero diversi *server - copia*, che offrono lo stesso servizio con gli stessi dati. Quando un server è copia di un altro, fare la richiesta a un server o a un altro produce sempre lo stesso risultato. Il meccanismo del DNS implementa proprio questa possibilità: assumere a un nome canonico un intero set di indirizzi IP. Solitamente gli indirizzi IP mostrati si mostrano sempre "a rotazione", così da gestire equamente il traffico tra tutti i server connessi a quel nome!

### La struttura gerarchica dei DNS server

Il DNS Server non può essere una struttura **centralizzata**. Questo è dovuto a diversi fattori:

- **Single Point of Failure:** se il server fosse solo uno, o al più fosse un'unica server farm, costituirebbe un unico "punto di rottura": se dovesse crashare, manderebbe in tilt l'intero sistema Internet.
- **Traffico:** se il server fosse solo uno, avrebbe da gestire le richieste DNS di tutti gli host. Queste comprendono tutte le richieste HTTP e tutte le richieste a un mail - server. Sarebbero ovviamente troppe, e si introdurrebbero dei ritardi significativi.
- **La distanza dal Server:** gli host vengono da tutto il mondo. Se il server fosse in America, anche chi è in Asia, in Europa, e negli altri stati dovrebbe collegarsi in America. Questo introduce un ritardo non indifferente, oltre che la possibile congestione dei vari bottleneck lungo il percorso.
- **Manutenzione:** se il server fosse solo uno, dovrebbe essere molto grande. Più un server è grande, e più necessita di manutenzione e di sistemi che permettono l'aggiornamento e l'inserimento di nuovi dati.

Convinti del fatto che una struttura centralizzata non può quindi funzionare, si scopre che il DNS Server in realtà è a struttura *gerarchica*: esiste un "albero" di DNS server visitati dalla root verso le foglie. Esistono principalmente tre tipi di DNS server:

- **I Root Server DNS:** sono gestiti da 13 organizzazioni e sono circa 400 in tutto il mondo. Tutte le richieste DNS arrivano a loro, ed è loro compito smistare le richieste rispondendo con l'indirizzo IP di un server DNS di livello più basso
- **I TLD (Top - Level - Domain) Server DNS:** sono quelli che hanno le traduzioni che mandano ai server di livello più basso nel loro "dominio di competenza". In particolare il server TLD .it conterrà le traduzioni in indirizzi IP dei server di livello più basso che gestiscono i vari domini .it.
- **Gli authoritative DNS server:** sono quelli che gestiscono un singolo dominio, e contengono la traduzione del dominio a loro associati e di tutti i suoi relativi sottodomini. In particolare contiene anche i vari set di indirizzi IP qualora ci siano Server Mirror.

*Esempio di traduzione www.amazon.tech.com*

1. Il client DNS manda la richiesta al root server DNS. Questo risponderà "*io non ce l'ho, chiedi a questo*" e restituisce l'indirizzo IP di uno dei TLD DNS Server che gestisce il dominio .com.

2. Il client DNS fa allora una richiesta al DNS server indicatogli nella risposta. Questo risponderà “*io non ce l'ho, chiedi a questo*” e restituisce il set di indirizzi IP dell'autoritative DNS Server che gestiscono il dominio *amazon.com*
3. Il client DNS sceglie (tipicamente) il primo di questi indirizzi IP restituitigli (vanno a rotazione successivamente) e fa una richiesta a cui finalmente trova risposta, perché il server authoritative contiene anche la traduzione dei sottodomini di *amazon.com*, e in particolare anche di *amazon.tech.com*.
4. **Il DNS authoritative manda sia la traduzione, sia il nome canonico (qualora quello richiesto non fosse già quello canonico).**

C’è in realtà un quarto tipo di DNS Server, che rimane fuori dalla struttura gerarchica finora discussa, ma ricopre un’importanza notevole: il **Local DNS Server**.

I Local DNS Server sono messi a disposizione dall’ISP o dall’organizzazione (che per qualsiasi motivo potrebbe comunque decidere di rivolgersi a un ISP ) e implementano un meccanismo di caching simile a quello che implementa il Proxy Server nell’HTTP.

In particolare ci sono due importanti vantaggi:

- **L’host dell’utente fa una sola richiesta**, chiede di tradurre un certo *HostName*. Questa richiesta viene indirizzata al DNS Server. È poi lui che comunica con la gerarchia dei DNS Server e solo quando ha la traduzione finale, la restituisce all’host.
- **Caching**: il local DNS Server può conservare una copia della traduzione, così da rispondere più in fretta la prossima volta e soprattutto evitando di generare troppo traffico verso la gerarchia dei server DNS. In realtà il Local può conservare anche una copia della traduzione relativa ai TLD DNS Server, così da bypassare i root nella catena delle richieste.

I meccanismi con cui il Local DNS comunica con l’intero “Albero” sono due:

- ❖ Meccanismo a **Iterated Query**: chiede ad ogni livello della gerarchia il prossimo DNS a cui chiedere, e si aspetta una risposta da quel livello. In particolare chiede al root, e il root risponde, chiede al TDL DNS Server, e lui risponde, poi chiede all’autoritative, e lui risponde.
- ❖ Nei meccanismi a **Recursive Query** la situazione è diversa: il local DNS Server fa iniziare un “percorso di richieste” che arrivano fino all’autoritative (nell’ordine: Local – Root – TDL – Auth). Dopodiché l’auth risponde al TDL, che risponde al root, che risponde infine al Local.

Questo tipo di soluzione permette al Local DNS Server di effettuare una sola richiesta, ma rischia di aumentare il traffico sulla rete di DNS, e dunque non è utilizzato.

## Gli RR’s: I Record del DNS

Gli RR (**R**esource **R**ecords) sono i record che i vari DNS consultano per poter dare una risposta. Tutti i record sono nel formato:

$$(Name, Value, Type, TTL)$$

Con il campo TTL (Time – To – Live) che indica quando tempo dovrebbe vivere quel record all’interno di una cache. Escluso questo campo, *Name* e *Value* assumono significati diversi a seconda del valore del campo *Type*.

- Se **Type = A**, allora *Name* è un hostname, e *Value* è il corrispettivo indirizzo IP. Tutti i DNS authoritative per quel dominio devono avere questi record, anche se non si esclude che altri Server possano avere lo stesso record.
- Se **Type = NS**, allora *Name* è il nome di un dominio e *Value* è l’hostname dell’autoritative server che può aiutarti a raggiungerlo. Questo tipo di record è quello che devono avere i TLD server, per

aiutare a raggiungere il router authoritative. I record di tipo NS sono sempre accompagnati da un record di tipo A che traduca l'indirizzo IP del nome dell'authoritative (altrimenti si saprebbe il nome ma non come raggiungerlo).

- Se **Type = CNAME**, allora *Value* è il nome canonico di *Name*
- Se **Type = MX**, allora *Value* è il nome canonico del mail server chiamato con l'alias *Name*. La possibilità di avere questi due record aiuta le compagnie ad utilizzare lo stesso alias sia per un mail server sia per un Web Server. Quando ci si vorrà riferire al Web, si cercherà tra i record di tipo CNAME, mentre quando si vorrà riferire al Mail, si cercherà tra i record di tipo MX.

### Chi inserisce questi record?

Quando una persona vuole registrare un dominio, chiede a un **registro** di fare il lavoro sporco al posto suo. In particolare si verifica l'unicità del nome che si vuole utilizzare, dopodiché è compito del registro inserire i record NS (e il corrispettivo A) nei TLD DNS Server. Questi record devono essere relativi ad (almeno due) server DNS authoritative per il dominio che voglio registrare.

## Le applicazioni P2P

Le applicazioni P2P sono, come già detto, tutte le applicazioni dove non ci sono ruoli definiti, e le coppie (**peer**) di host comunicanti possono intercambiare il loro ruolo di client e server. Questo tipo di modello garantisce anzitutto un "risparmio" di macchine computazionalmente veloci, in quanto non c'è la necessità di un server centralizzato, o comunque di macchine server che debbano prendersi tutte le richieste. Ovviamente, per gli stessi motivi non è necessario che ci siano macchine **sempre accese!**

In generale un'applicazione P2P è comoda quando si vuole **condividere un file a un certo numero di utenti**. Prendiamo un caso in esame, in cui ci si pone il problema di inviare un file grande F a un N peers.

Risolviamo il problema col modello client - server, e chiediamoci quale sia il **tempo di distribuzione**, ovvero il tempo affinché tutti i peer ricevano l'intero file. Assumiamo per semplicità che internet non costituiscia un bottleneck, e che tutti stiano facendo questo, ovvero che non abbiano applicazioni di rete concorrenti.

- Indichiamo con  $U_s$  la capacità di upload del server. Questo dovrà performare un invio **sequenziale** di N copie del file (una per ogni peer). Questo tempo sarà al minimo  $NF / U_s$ .
- Indichiamo  $U_i$  e  $D_i$  rispettivamente la capacità di upload e download del peer  $i$ -esimo. In particolare ogni peer deve scaricare il file, impiegando un tempo  $F / D_i$ . Ovviamente il *peer che domina questo ritardo* è il peer con capacità di download più bassa.

In un modello Client - Server, il tempo di distribuzione  $D_{CS}$  sarà **limitato da**:

$$D_{CS} \leq \max \left\{ \frac{NF}{U_s}, \min \left\{ \frac{F}{U_i} \mid i = 0 \dots N - 1 \right\} \right\}$$

Dunque questo tempo dipende linearmente dal numero dei Peer. Anche il secondo fattore dipende da N, poiché è più probabile che aumentando il numero dei Peer ne peschiamo qualcuno di più lento di tutti i precedenti, ma lo tralasceremo.

**Scegliendo il modello Client - Server, il tempo di distribuzione cresce senza limiti proporzionalmente al numero dei Peer.**

Scegliendo invece il modello P2P, le cose (almeno nell'ottica del tempo di distribuzione) migliorano:

- Il server basta che invii una copia del file. Il tempo si ridurrebbe a (ovviamente si parla sempre di valori minimi e non realistici)  $F / U_s$ .
- I peer devono ovviamente scaricare il file, impiegando come prima un tempo  $F / U_D$  che è limitato dal peer che ha capacità di download minore.

- C'è da stimare il tempo che complessivamente impiegano i Peer per trasmettersi il File. In generale, il carico totale che dovranno trasferire è NF, e utilizzeranno vari link. Non possono trasferire gli NF con una capacità maggiore della somma di tutte le loro capacità, dunque stabiliremo un minimo, dato da  $NF / (\text{Somma delle capacità di upload} + \text{capacità del server})$ .

Stavolta possiamo stimare  $D_{P2P}$ :

$$D_{P2P} \leq \max \left\{ \frac{F}{U_S}, \min \left\{ \frac{F}{U_I} \mid I = 0 \dots N - 1 \right\}, \frac{NF}{U_S + \sum_{i=1}^N U_I} \right\}$$

Stavolta è il terzo termine il più dipendente da N. La fortuna è che dipende da N anche al denominatore. Se ci mettiamo nel caso ideale in cui tutti i peer hanno lo stesso rate u, si ottiene un limite inferiore  $F/U$ . Il che è un gran risultato, perché possiamo "garantire" che qualunque sia il numero N dei peer, il tempo di distribuzione sarà limitato e "prevedibile".

### I meccanismi del P2P : Dove si trovano le Risorse

Il primo problema che sta alla base di ogni applicazione P2P è capire come fanno i vari peer a ricavare le proprie risorse. L'idea è quella di instaurare un DB che contenga coppie (KEY, VALUE) facendo in modo che un Peer faccia una prima richiesta inserendo la KEY, e cerchi la risposta VALUE. Questa prima fase (**Fase di Localizzazione delle Risorse**) potrebbe essere progettata in modello client – server, instaurando un server a cui i peer fanno queste "piccole richieste". Questo DB prende il nome di **INDICE**.

- Se l'indice è **centralizzato**, ci sono i problemi che tutti conosciamo: single – point of failure e rischio di congestione dovuto al fatto che tutte le richieste arrivano nello stesso punto. Nelle applicazioni client – server, questo problema è risolto col il meccanismo del **server – mirroring**, ovvero creare una rete di server distribuiti che possono rispondere alla stessa richiesta.
  - Fu il caso di **Napster**, che utilizzava l'indice centralizzato per permettere ai peer di capire a chi chiedere il file multimediale.

Il problema che così non viene risolto è **l'aggiramento del copyright**, perché i server sono "identificabili", ovvero si può capire chi è che sta fornendo l'indice.

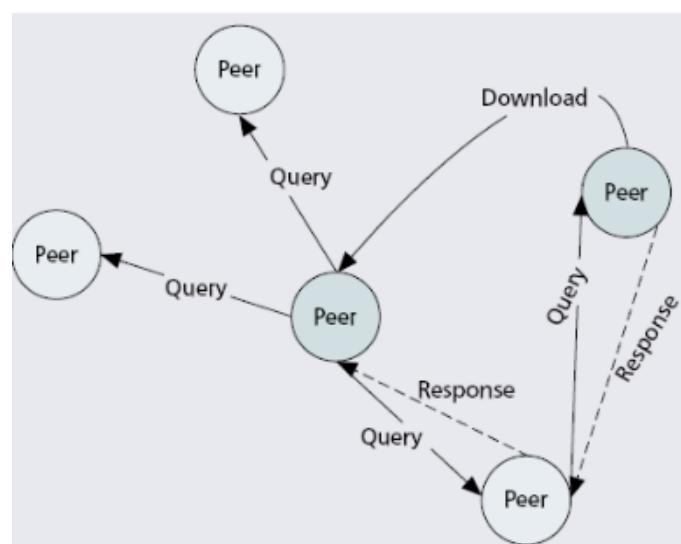
Per risolvere questo problema si introduce **L'indice Completamente Decentralizzato**, ovvero i peer che vorrebbero inserire dei record (key, value) all'interno di un indice (inseriscono i record per segnalare che loro hanno quel file), costituiscono ora un *proprio indice personale*.

L'idea è che se tutti i peer si suddividono l'indice, non si può fare causa a nessuno. I peer devono però allora capire a chi chiedere, e l'idea è quella di istituire un sistema di **passaparola**, confidando nel fatto che prima o poi la risorsa sarà localizzata.

L'indice è realizzato dunque sui **singoli peer**, e ogni peer realizza la parte dell'indice relativa ai contenuti che loro stessi mettono in condivisione.

I singoli peer, però, non stabiliscono connessioni TCP con l'intera comunità, ma solo con una parte proporzionalmente

piccola. Chiameremo i Peer con cui un Peer oggetto chiede una risorsa i suoi **vicini**. Se un Peer vuole dunque sapere *dove trovare una risorsa*, ha bisogno che gli altri consultino il proprio "pezzo di indice", fino a trovarlo e a restituire una risposta. Secondo il meccanismo di **Query Flooding**, il peer inizialmente chiede ai suoi vicini, che se non avranno la risorsa chiederanno ai loro vicini, e così via. In questa ottica è abbastanza sicuro che, se la risorsa è presente nella comunità, questa verrà trovata. Ci sono però alcune cose che vanno male:



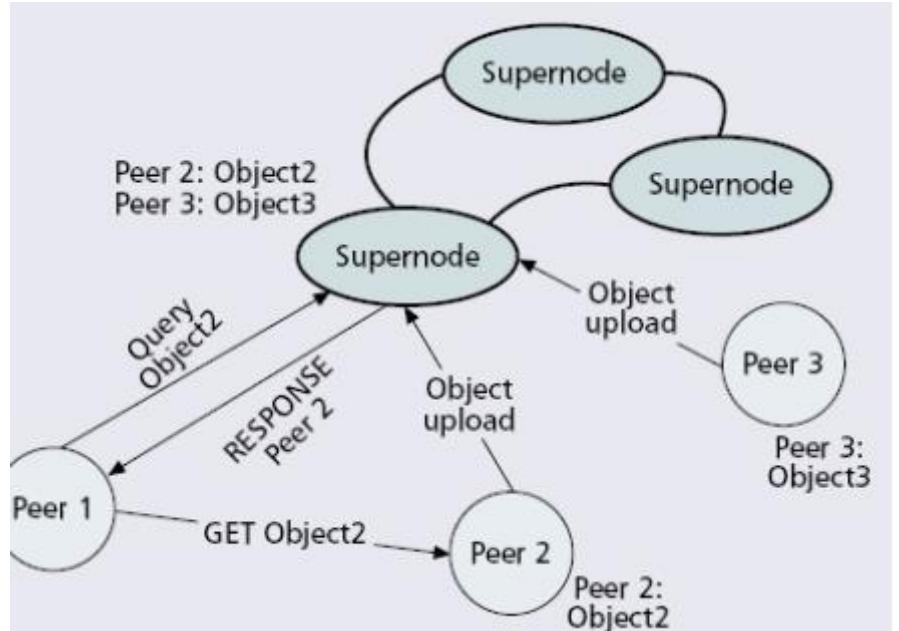
- **Il tempo per sapere dov'è la risorsa potrebbe essere troppo grande:** se infatti la risorsa sta in un Peer molto "lontano" (in termini di gradi di separazione), avrà bisogno di aspettare un certo numero di query di richiesta, il che potrebbe aumentare troppo il tempo che ci mette a conoscere "solo dove si trova il file".
- **Il traffico sulla rete aumenta:** è "inondata" di queste richieste di controllo.

Per risolvere questi problemi ci sono piccole ottimizzazioni possibili, ma niente che azzeri il rischio: si possono limitare i gradi di separazione, implementando un contatore di richieste (della serie: "Se arrivo alla 7 richiesta fermati"), oppure implementando dei meccanismi cache in ogni Peer così da avvicinare le risposte.

Un terzo possibile modo per risolvere le cose è implementando una **Struttura Gerarchica** nei Peer, così da ibridare i due sistemi precedenti. In questa struttura la particolarità è che i Peer non sono tutti uguali. Esistono dei Peer detti **Super Peer (o Super Nodes) (SN)**.

Questi Peer sono caratterizzati dall'avere caratteristiche di velocità diverse dai normali Peer. Hanno ad esempio un'elevata banda, o una capacità computazionale più alta del normale. Normalmente un SN è ad esempio fornito dalle istituzioni.

Ogni Peer si riferisce a un SN quando cerca una risorsa, e l'indice è smistato solo tra gli SN.



Se un Supernodo non riesce a rispondere al peer, inoltre la query alla sua rete di supernodi. Inoltre quando riceve la coppia (Key, Value), se la salva nella sua memoria da SN. In questo modo più richieste arrivano al supernodo, più ci sarà la possibilità che la prossima volta potrà rispondere direttamente lui.

### Approfondimento: L'indice a Tabella Hash Distribuita

#### Trasferimento dei File: Il protocollo BitTorrent

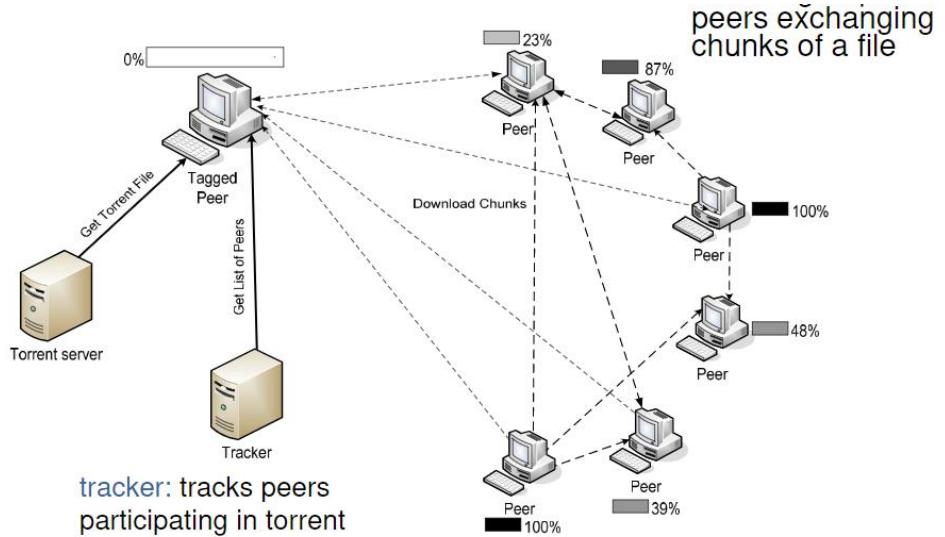
L'idea è che il trasferimento di un file avvenga grazie all'azione cooperativa di più peers. Dividendo il file in **chunks**, l'obiettivo è quello di farsi mandare diversi chunk da diversi peer contemporaneamente, così da aumentare il tempo di distribuzione del file.

Il protocollo **BitTorrent** si basa proprio su questo, tutta la comunità che condivide nel trasferire il file partecipa a un "*torrent di bit*". Ci sono alcuni attori principali da identificare:

- Un **Torrent Server** che contenga file .torrent dove si trovano informazioni di vario tipo, tra cui l'indirizzo del Tracker.
- Uno (o più) **Tracker**, che tiene traccia (il più real - time possibile) dei peer che stanno partecipando al torrent.
- I peer che stanno cercando di ricevere il file, ma non hanno ancora completato il trasferimento: le sanguisucche **Leecher**.
- I peer che hanno completato il trasferimento ma restano comunque disponibili nel torrent per inviare i chunk mancanti (**Seeders**).

Chiamiamo infine **Tagged Peer** il peer che stiamo osservando, che è interessato a ricevere il file. Quello che deve fare, in ordine, è:

1. Richiedere al torrent server il .torrent
2. Richiedere al Tracker (il cui indirizzo è nel .torrent) la lista dei Peer che stanno partecipando al torrent
3. Instaurare un **vicinato**, ovvero provare a instaurare connessioni TCP con un discreto numero di Peer. Quelli che partecipano al torrent potrebbero essere troppi, e alcuni potrebbero non rispondere, per cui in linea di massima non si comunica con l'intera comunità. Chiameremo **vicini** i peer con cui si è instaurata con successo una connessione TCP.
4. Richiedere, secondo una certa politica, i chunk che mancano al tagged peer, e condividere copie dei chunk che ho ricevuto.



La politica che si utilizza nel richiedere i chunk è quella **Rarest First**. Ovvero si fa un'operazione di richiesta circa i chunk mancanti, e si vede quanti peer hanno quel particolare chunk. Si chiede il trasferimento del chunk più raro, perché se i peer che ce l'hanno si disconnettessero, potrei non riuscire ad ottenerli.

Un'altra politica interessante è “*come decidere a chi mandare quanti Chunk e quanto frequentemente*”. Devo infatti fare una scelta basandomi su “quanto ci guadago”. La politica è quella **TIC FOR TAT**.

Secondo questo meccanismo, instauro una lista **TOP 4** dei 4 peer che mi inviano più roba (in termini di chunk/sec ad esempio). Rivaluto questa classifica a intervalli regolari, diciamo ogni 10 secondi. Invierò i miei chunk ai miei TOP 4 (se ne hanno bisogno). Questa politica porta a due conseguenze interessanti:

- Se sei troppo passivo, non ti tornerà niente. Infatti chi decide di non condividere i chunk che ha ottenuto finirà sempre in fondo alla lista dei TOP4 degli altri Peer, e si rallenterà drasticamente la sua velocità di ricezione.
- **Non possiamo essere sicuri che i TOP4 siano davvero la scelta migliore:** infatti potrebbe esserci qualche altro Peer che potrebbe mandarmi le cose più velocemente, ma non posso scoprirla perché non gli ho mai inviato dei chunk (e quindi non posso entrare nella sua TOP4). Per risolvere questo problema a intervalli regolari, diciamo ogni 30 secondi, **manderò un chunk a un peer a caso**. Se questo mi aiuterà a ricevere una risposta da lui, potrò valutare se inserirlo nella mia TOP4 o meno. Quando comunico con un peer in questo modo, si dice che “**Optimistically Unchoke**” quel peer (inglese di merda).

# CAPITOLO 3: RETI A CONNESSIONE DIRETTA

LEZIONE 20/10 – XX/YY

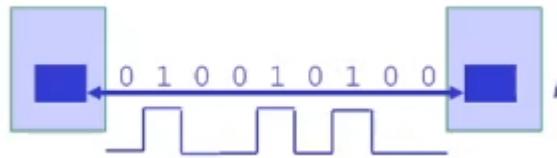
Indicheremo con **Reti a Connessione Diretta** tutte quelle reti che coinvolgono due host direttamente collegati, in maniera wired o wireless. Questo perché la rete generale non è altro che un “insieme di reti a connessioni di rete”, e tutti i problemi e le difficoltà di questo semplice modello si ripercuotono uguali sul caso generale. Vale dunque la pena di studiarli.

## L'introduzione al Problema

Chiameremo **Nodi** gli Host e i Router, e **Link** i canali di comunicazione che connettono due nodi adiacenti. Un certo Host deve inviare mediante il Link una stringa di bit all'altro Host. C'è bisogno principalmente di due dispositivi:

- **Trasmettitore:** un dispositivo che sia in grado di trasmettere impulsi di diversa natura, e facendo corrispondere impulsi diversi a livelli logici (0 o 1) diversi. C'è dunque bisogno di un algoritmo di codifica per cui partendo da una stringa di bit siamo in grado di generare una serie di segnali fisici opportuni sul canale di comunicazione.

I segnali fisici da generare variano in base al tipo di Link: se utilizzo un link in rame utilizzerò un segnale elettrico, se utilizzo una fibra ottica sarà un segnale luminoso e, se è wireless, dovrà avere segnali elettromagnetici per codificare i due livelli logici possibili.



- **Ricevitore:** un dispositivo che è in grado di ricevere i segnali fisici ricevuti e di decodificarli, ottenendo la stringa di bit che si è voluta trasmettere.

Il link è a **Half - Duplex** se c'è un Trasmettitore e un Ricevitore, **Full - Duplex** se da entrambe le parti si fanno entrambe le cose.

La codifica scelta (a scopo didattico) e di associare due livelli di tensione in base a una soglia di riferimento. Il valore superiore alla soglia rappresenta l'1 logico, mentre un valore inferiore alla soglia rappresenta lo 0 logico.

Un primo problema è il fatto che **la comunicazione non è perfetta**. Questo deriva da due fattori:

- La transizione impulso basso – impulso alto non è *immediata!* C'è un transitorio in cui il valore logico può essere frainteso
- La potenza ricevuta potrebbe essere troppo bassa, a causa dell'attenuazione del segnale man mano che si propaga. L'attenuazione è funzione del quadrato della distanza, e dunque è chiaro che, per distanze molto grandi, il ricevitore potrebbe finire per **non ricevere affatto alcun segnale!**
- Inoltre il ricevitore, conoscendo la velocità di trasmissione, deve decidere di campionare ed estrarre un valore del segnale continuo che gli arriva. Il **punto di campionamento è dunque importante**, perché devo riuscire a campionare al *centro* dell'impulso. Ma qual è il centro dell'impulso? Ricevitore e Trasmettitore hanno due clock diversi che alla lunga non sono più sincronizzati e riconoscere il centro è un problema!

In generale la sequenza ricevuta è diversa dalla sequenza trasmessa. La misura dell'errore è il **bit error rate**, ovvero la percentuale di bit errati durante una trasmissione. C'è bisogno di introdurre un meccanismo affinché il ricevitore riceva i dati corretti, e soprattutto che si accorga che ciò che ha ricevuto è sbagliato.

Purtroppo dobbiamo viaggiare con la sicurezza che il mezzo che utilizziamo è non affidabile. Per alcune applicazioni abbiamo bisogno di affidabilità al 100%. **Come si fa?**

Alla base di questo meccanismo c'è il **framing** (o meccanismo di tramatura), ovvero divido i dati che voglio inviare in *FRAME*, che sono parti del messaggio incapsulate tra una testa e una coda. Questo fu un meccanismo pensato perché, qualora il ricevitore fosse riuscito ad accorgersi dell'errore, non doveva essere necessario reinviare l'intera risorsa iniziale.

Ci saranno dunque frame che arrivano integri, e frame che arrivano non integri, e nel secondo caso deve esserci un modo per richiedere la trasmissione di quel particolare frame. Ci sono però dei buchi:

- Il ricevitore non sa cosa deve ricevere. Se lo conoscesse, non ci sarebbe bisogno della trasmissione, e pertanto bisogna implementare un meccanismo di **test** che dica se il frame ricevuto è integro o meno.
- Bisogna riconoscere i frame, identificandoli per bene affinché la ricostruzione vada a buon fine.
- Bisogna essere in grado di **richiedere la retransmissione di uno o più frame**

Ogni frame è incapsulato tra due informazioni di controllo:

1. Uno **Header** (messo in testa ai dati), che contiene informazioni di controllo vere e proprie
2. Un **Trailer** (messo in coda ai dati), che contiene i bit che permettono al ricevitore di effettuare un test di correttezza



Ma chi è che fornisce tutti questi servizi e risolve tutti questi problemi che si diffondono sul livello fisico? Esiste uno strato, il **Data Link Layer**, che si preoccupa proprio di offrire questi servizi.

## I Servizi del Data-Link Layer

Oltre al Framing, il Data Link Layer fornisce altri servizi:

- **Error Detection:** il ricevitore, grazie ad opportuni bit inseriti nel frame, deve essere in grado di decidere se il frame ricevuto è corretto o meno
- **Error Correction:** una volta deciso che il frame ha degli errori, in alcuni casi il ricevitore può essere autonomamente in grado di correggerli! Questo perché alcuni test sofisticati mi dicono anche *qual è il bit errato*. E se so che un bit è errato, si complementa per correggerlo! Questo metodo, che funziona se il numero di bit errati è limitato, è molto costoso
- L'alternativa consiste nella **Retrasmissione**, ovvero se ricevo un frame che contiene errori comunico di rivolgerlo. Questo meccanismo è riassunto nella proprietà di Reliable Data Transfer.
- **Flow Control:** il ricevitore può chiedere al trasmettitore di *rallentare*, perché non riesce a gestire i dati tanto velocemente quanto sono inviati.

### Dove sono all'interno del calcolatore queste funzionalità?

Nello spazio di I/O c'è la Network Interface Card (scheda di rete). L'interfaccia implementa il livello fisico (avendo ricevitore e/o trasmettitore). Il controllore invece fa da interfaccia tra la scheda e il bus.

Le funzionalità del Data Link sono implementate *parte in Hardware* all'interno del controllore, e *parte in Software* ed eseguite sotto forma di istruzioni.

## Error Detection & Error Correction

Il primo problema che si deve sempre porre il ricevitore è: "I bit che mi sono arrivati sono quelli che mi sono stati davvero mandati?". Infatti il ricevitore non conosce ciò che sarà trasmesso (anzi generalmente non si nemmeno aspetta una trasmissione). C'è dunque bisogno di un sistema, che vedremo essere **non sicuro al 100%** per capire se ci sono errori o meno:

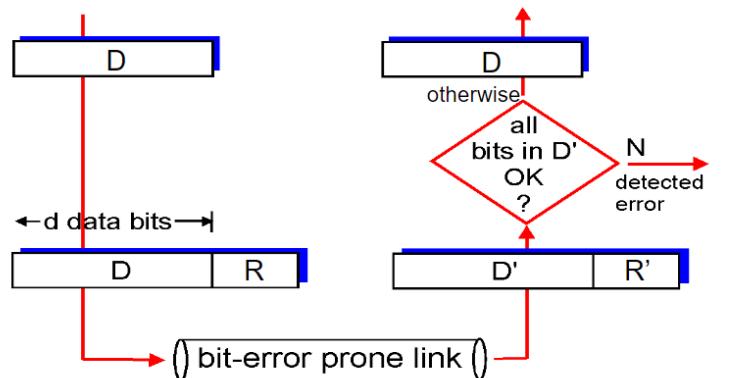
Cerchiamo di ragionare: dal livello network arriva un pacchetto di dati D (che contiene dati e linee di intestazione, ma noi vogliamo preservare tutto) e vogliamo mettere in coda a questo dei **bit di ridondanza** che ci aiutino in qualche modo a capire o meno se c'è stato un errore. Si deve identificare un algoritmo che, dato D, genera un certo R. Dopodiché il pacchetto si completa aggiungendo R in coda.

A questo punto si manda il pacchetto  $\langle D, R \rangle$  lungo il canale di trasmissione, che **sappiamo possibile fonte di errore**. Il ricevitore riceve dunque un pacchetto  $\langle D', R' \rangle$ , e conosce "il punto di stacco" delle due fasi (perché i due si sono messi d'accordo prima in qualche modo).

A questo punto il ricevitore applica lo stesso algoritmo utilizzato dal trasmettitore su  $D'$ .

Se il risultato è pari ad  $R'$ , allora **si conclude** che i bit  $D'$  sono uguali a  $D$ , perché l'algoritmo ha dato lo stesso risultato sia su  $D$  che su  $D'$ .

Ci sono una serie di forzature in questo ragionamento:



- **L'algoritmo non è necessariamente iniettivo:** ovvero non è detto che a cose diverse corrispondano sempre cose diverse. È dunque possibile che  $\text{Algoritmo}(D) = R'$ , ma  $D' \neq D$  perché l'algoritmo restituiva  $R'$  per più input possibili. Questo è un caso in cui l'errore non viene rilevato!
- **I bit di Ridondanza si possono corrompere:** ciò significa che magari  $\text{Algoritmo}(D') = R'$ , ma essendo  $R' \neq R$  non abbiamo alcuna sicurezza che valga anche  $D = D'$ .

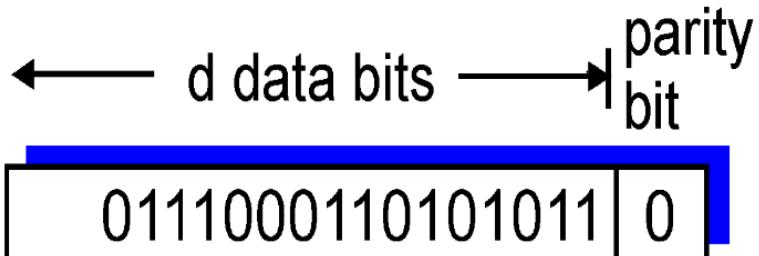
Ovviamente, più il campo R è grande, più è possibile individuare eventuali errori!

In generale, l'**error Detection non è infallibile**, ma non possiamo fare altro che fidarci.

Vediamo ora un po' di esempi di algoritmi per generare R.

### Un primo esempio: il bit di PARITA'

In questo caso il campo R è grande **un solo bit**, e dunque non è affatto costoso introdurlo (si parla di costi in termini di ritardi di trasmissione del pacchetto, non di elaborazione).



La parità può seguire uno schema pari o dispari. Nel caso di parità pari il bit di parità rende pari il numero di bit a 1, viceversa nel caso di parità dispari. Nell'immagine, ad esempio, si segue una parità dispari.

Se il ricevitore riceve il bit di parità (assumiamo non corrotto), ma non si trova d'accordo, allora **riesce a fare detection dell'errore**. Questo sistema, nonostante l'assunzione che il bit di parità non si corrompa, ha comunque un difetto, riesce a rilevare l'errore solo se a cambiare è stato un numero dispari di bit. Quando infatti cambia un numero pari di bit, la **parità non cambia**, e dunque tutto risulta corretto.

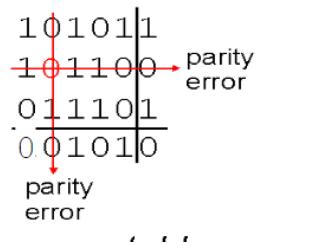
Questo non ci piace affatto, perché spesso gli errori arrivano *in burst* e dunque è più che possibile che coppie di bit si alterino. Il metodo è utilizzato nelle brevi trasmissioni, ma assolutamente non nelle lunghe (sia in termini di link che in termini di grandezza di D).

### Parità Bidimensionale

La parità bi - dimensionale è un primo esempio di **Error Correction**, in cui il ricevitore riesce a fare sia detection dell'errore, che correzione.

In un dominio binario, la correzione consiste nell'individuazione del bit errato. Se si capisce in qualche modo quale dei bit trasmessi è quello errato, basta complementarlo!

Questo tipo di parità si basa sul dividere i bit componenti il gruppo D "a matrice", e di inserire un bit di parità per ogni riga e per ogni colonna. In questo modo quando si corrompe un bit, l'errore viene identificato sia dalla parità di riga che dalla parità di colonna, e il bit errato sta all'incrocio!



*Nota sull'Error Correction:* la capacità di identificare un errore è correggerlo è nota come tecnica **FEC (Forward Error Correction)**. Avere questa possibilità è in generale oneroso, e porta tutta una serie di problemi di prestazione. È tuttavia preferibile, se non necessaria, nei sistemi dove la retransmissione potrebbe essere un sistema non utilizzabile. Ad esempio nei sistemi real - time, o nei sistemi dove il ritardo di propagazione è davvero troppo elevato.

## Il metodo Checksum

Questo metodo consiste nel dividere il frame in pacchetti di K bit e vedere le varie sequenze come numeri interi rappresentati su K bit. La somma di questi numeri forma il **checksum**, che è un numero la cui rappresentazione costituisce i bit di ridondanza.

Banalmente, se la somma torna uguale anche al ricevitore, a livello teorico non ci sono errori. Sappiamo che anche questo metodo ha delle fragilità, poiché la somma di cose diverse potrebbe portare allo stesso risultato.

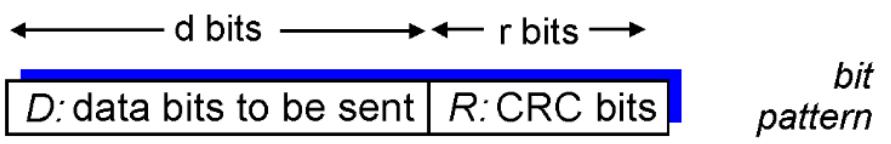
Questo è il metodo utilizzato a livello del Trasporto, perché l'error detection è implementata in software e dunque servono cose "precise", ma probabilmente "poco onerose".

Essendo la detection implementata in Hardware per quanto riguarda il Link - Layer, si preferisce un approccio che in software sarebbe troppo oneroso.

## Il CRC (Cyclic Redundancy Check)

Il metodo basato sui CRC codes è il più utilizzato attualmente a basso livello. Il metodo richiede che le due parti si mettano d'accordo concordando un pattern di r + 1 bit detto **generatore G**.

Dopodiché si fa in modo che il trasmettitore trovi R su r bit tale che  $D * 2^r \text{ XOR } R$  sia un numero divisibile per G nell'aritmetica degli interi (resto nullo). Nel metodo **non si considerano riporti o prestiti**, ciò significa che qualsiasi operazione di addizione o sottrazione è implementata con uno XOR. Dunque il pacchetto di dati che invio, che è la concatenazione di D e R, è esattamente il numero da testare su G. Ovviamente, l'errore è detectato se sul ricevitore il resto della divisione è diverso da 0.



$$D * 2^r \text{ XOR } R \quad \text{mathematical formula}$$

Resta solo un problema: **come fa il trasmettitore a generare R?** Si può rispondere a questa domanda svolgendo alcune semplici operazioni aritmetiche. Vogliamo che esista n intero tale che:

$$D * 2^r \text{ XOR } R = n * G$$

Aggiungendo R a destra e sinistra (che equivale a fare XOR R a destra e sinistra visto che non consideriamo i riporti), si ottiene:

$$D * 2^r \text{ XOR } (R \text{ XOR } R) = n * G \text{ XOR } R$$

Ed essendo  $R \text{ XOR } R = 0$ , ed essendo XOR 0 un'operazione che lascia l'operando invariato, si ottiene

$$D * 2^r = n * G \text{ XOR } R$$

E dunque significa che R è il **resto** della divisione tra  $D * 2^r$  e  $nG$ . Posto  $n = 1$ , non è altro che il resto tra il pacchetto di dati shiftato a sinistra di r bit e il generatore G!

## Reliable Data Transfer: nella top 10 degli argomenti più interessanti

Questo è un argomento a sé. Si dà un contesto generale, in quanto si parla di questo, sia a livello Data - Link, che a livello Trasporto, che a livello Applicazione. Per questo motivo parleremo di trasferimento affidabile di **pacchetti**, riferendoci però a pacchetti, datagram, frame, o quello che è.

L'obiettivo è dunque creare un **Reliable Data Transfer Protocol**. L'idea è di creare l'astrazione di un canale affidabile, che però sappiamo non esistere, in quanto al livello più basso ci saranno sempre dei link di livello Data - Link, che sappiamo non essere affatto affidabili.

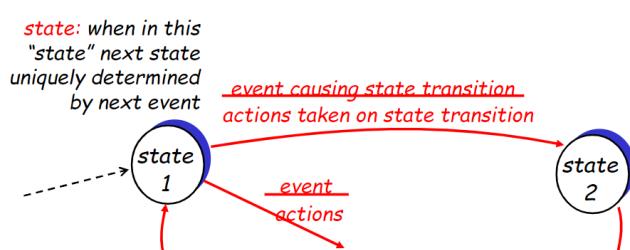


Diamo qualche parola chiave:

- Quando il livello più alto vuole inviare un pacchetto sfruttando un canale sicuro, chiama una `rdt_send()` (che sta per **reliable data transfer**, lato "send" del protocollo).
- Il più basso deve poi invocare una `udt_send()` (che sta per **unreliable data transfer**, ed è dovuto al fatto che si appoggia su un canale che per sua natura non può essere affidabile al 100%)
- Quando il ricevitore vorrà ricevere qualcosa dal canale (non affidabile), dovrà invocare una `rdt_recv()` e, qualora sia possibile spedire il pacchetto ai livelli superiori, si esegue una `deliver_data()`. Il pacchetto si spedisce in due situazioni: **nessun errore, ed è il pacchetto che mi aspettavo**.

Creiamo dunque il protocollo, cercando di aumentarne la complessità volta per volta e interpretandone i vantaggi e i problemi che ne derivano.

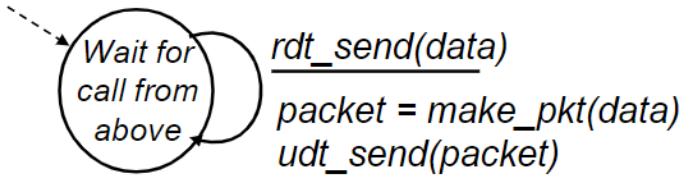
Rappresenteremo il protocollo mediante l'utilizzo di una macchina a stati, sia per il trasmettitore che per il ricevitore. In una rappresentazione di questo tipo nei cerchi ci sarà una descrizione dello stato, le frecce (che corrispondono a transizioni di stato), avranno sopra di una linea. Al di sopra della linea ci sono gli **eventi che causano la transizione**, mentre al di sotto della linea stanno le azioni prese durante la transizione. È possibile avere frecce che si chiudono sullo stesso stato, che rappresentano un paradigma *evento - azione*.



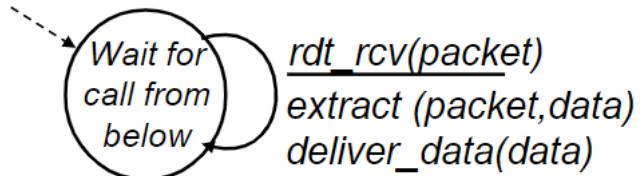
### RDT Protocol 1.0 : Il caso ideale

Nel caso ideale, dove il trasferimento avviene su un canale sicuro che non produce errori, e dove il ricevitore non deve dare alcun segnale di feedback o alcun segnale di "slow - down", ciò che devono fare i due attori del nostro protocollo è super intuitivo:

- **Trasmettitore:** aspetta i pacchetti dall'alto e, quando arrivano, li **inpacchetta**. A questo livello di idealità il pacchetto creato è esattamente uguale ai dati che si stanno trasmettendo! Non ci sono, ad esempio, bit di ridondanza. Dopodiché il trasmettitore fa *udt\_send(packet)* e manda sul canale (**affidabile**) il pacchetto.



- **Ricevitore:** aspetta i pacchetti dal basso (cioè dal canale affidabile). Quando riceve un dato, estrae dal pacchetto i dati (**in questa operazione pacchetto e dati sono la stessa cosa, non c'è nulla da estrarre**), dopodiché fa *deliver\_data(data)* e spedisce i dati sui livelli più alti dello stack protocollare.



Si noti che entrambi gli attori in gioco hanno un solo stato possibile. Questo perché non possono verificarsi errori, attese, o altro. Si manda e basta, si riceve e basta, e si è certi che tutto sia andato bene.

Nella realtà però non è così, **i pacchetti possono corrompersi**.

### RDT Protocol 2.0 : I pacchetti che si corrompono

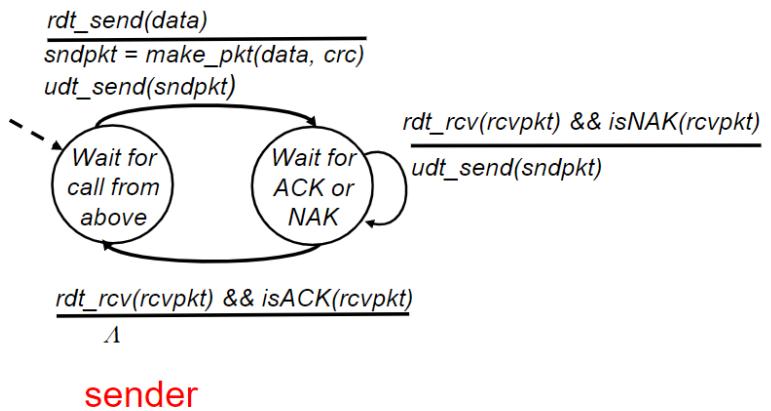
Considerato che nel mondo reale i canali non sono davvero affidale e possono corrompersi, c'è bisogno di inventarsi una sorta di comunicazione con il ricevitore, per capire quando ha capito o quando non ha capito perché ha rilevato degli errori.

Dobbiamo quindi modificare il nostro protocollo inserendo tre elementi fondamentali:

1. **Error Detection:** introduciamo dei bit di ridondanza in fondo al pacchetto che stiamo mandando (assumeremo siano i bit CRC, ma potrebbero essere i bit checksum, ecc.). In questo modo il ricevitore sarà in grado di capire quando c'è stato un errore durante la trasmissione.
2. **Feedback:** c'è bisogno che il ricevitore sia in grado di mandare dei segnali che rappresentino "ho capito" o "non ho capito". Servono questa funzione gli **Acknowledge Positivi (ACK)** e gli **Acknowledge Negativi (NAK)**. Quando il ricevitore riesce a rilevare una corruzione, manda un NAK, viceversa manda un ACK.
3. **Retrasmissione:** c'è bisogno di modificare il trasmettitore dicendogli di ritrasmettere l'ultimo pacchetto inviato nel caso in cui riceva il NAK. Attenzione: stiamo assumendo che i pacchetti arrivino in ordine e che non si perdino, e dunque è immediato per il trasmettitore capire quale pacchetto deve ritrasmettere (l'ultimo che ha inviato).

Ecco come si modificano le due macchine a stati di trasmettitore e ricevitore, implementando un protocollo ARQ (**Automatic Repeat reQuest**). Il protocollo che segue il trasmettitore è del tipo **stop - and - wait**, ovvero non manda nulla finché non è sicuro che il ricevitore abbia ricevuto correttamente l'ultimo pacchetto trasmesso.

- **Trasmettitore:** ha ora due stati in più. Dopo aver trasmesso sul canale unreliable (come nella versione 1.0, ma stavolta con i bit crc in fondo), si porta in uno stato in cui **aspetta un ACK o un NAK**. Se riceve un ACK, transisce nello stato iniziale e torna ad aspettare dati dall'alto. Se riceve un NAK, ripete l'invio dell'ultimo pacchetto trasmesso. **Se il trasmettitore è nello stato di wait, non accetta dati dai livelli superiori (stop - and - wait).**



- **Ricevitore:** il ricevitore rimane con un solo stato. Se il pacchetto gli arriva integro, fa deliver e trasmette un ACK, altrimenti scarta il pacchetto e trasmetta un NAK.

Questo sistema funziona, ma introduce un problema. **Il segnale di ACK o NAK potrebbe corrompersi!** Se questo succede è un serio problema, perché il trasmettitore deve fare in modo di capire quand'è che il pacchetto è stato ricevuto correttamente o meno, ed eventualmente ritrasmetterlo.

Una soluzione possibile è quella di interpretare l'ACK / NAK corrotto come un NAK. In questo modo il trasmettitore reinvia lo stesso pacchetto quando non riesce a capire se è stato ricevuto correttamente. Ciò introduce un nuovo problema: se il segnale corrotto fosse un ACK, il ricevitore non riesce a capire che il prossimo pacchetto sarà un duplicato del precedente!

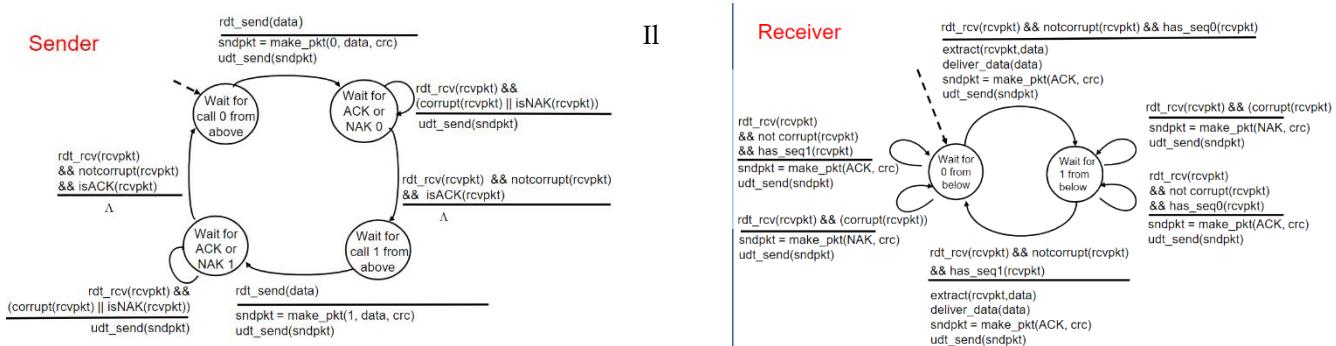
Bisogna dunque implementare un sistema per aiutare a far capire al ricevitore *quale pacchetto aspettarsi*.

### RDT Protocol 2.1 : I Numeri di Sequenza

Per risolvere il problema accenato sopra si inserisce in testa ad ogni pacchetto un **numero di sequenza** fatto da un solo bit che aumenta circolarmente (0, 1, 0, 1, 0, ....).

Perché un bit basta? Al ricevitore serve capire solo due informazioni: “*è lo stesso del precedente, o è diverso?*”. Al ricevitore basta dunque sbirciare il numero di sequenza, se è uguale “a quello precedente”, allora scarta il pacchetto, altrimenti lo prende e si porta in uno stato in cui aspetta il prossimo numero di sequenza.

Proprio perché trasmettitore e ricevitore devono portarsi in uno stato in cui mandano o ricevono rispetto al numero di sequenza, per entrambi il numero di stati è raddoppiato!



trasmettitore è “trasparente” rispetto a quello che vive il ricevitore, e sa solo che deve confezionare dei numeri di sequenza (e ritrasmettere lo stesso in caso di NAK o ACK/NAK corrotto). Per il ricevitore la situazione cambia leggermente. Conserva l'informazione “*Mi aspetto questo numero di sequenza*” nello stato in cui si trova. Dopodiché se il pacchetto arriva corrotto manda un bel NAK. Se il pacchetto arriva integro, **ma**

**non è il numero di sequenza atteso in quello stato, si manda un ACK, ma non si fa deliver e non avanza lo stato!**

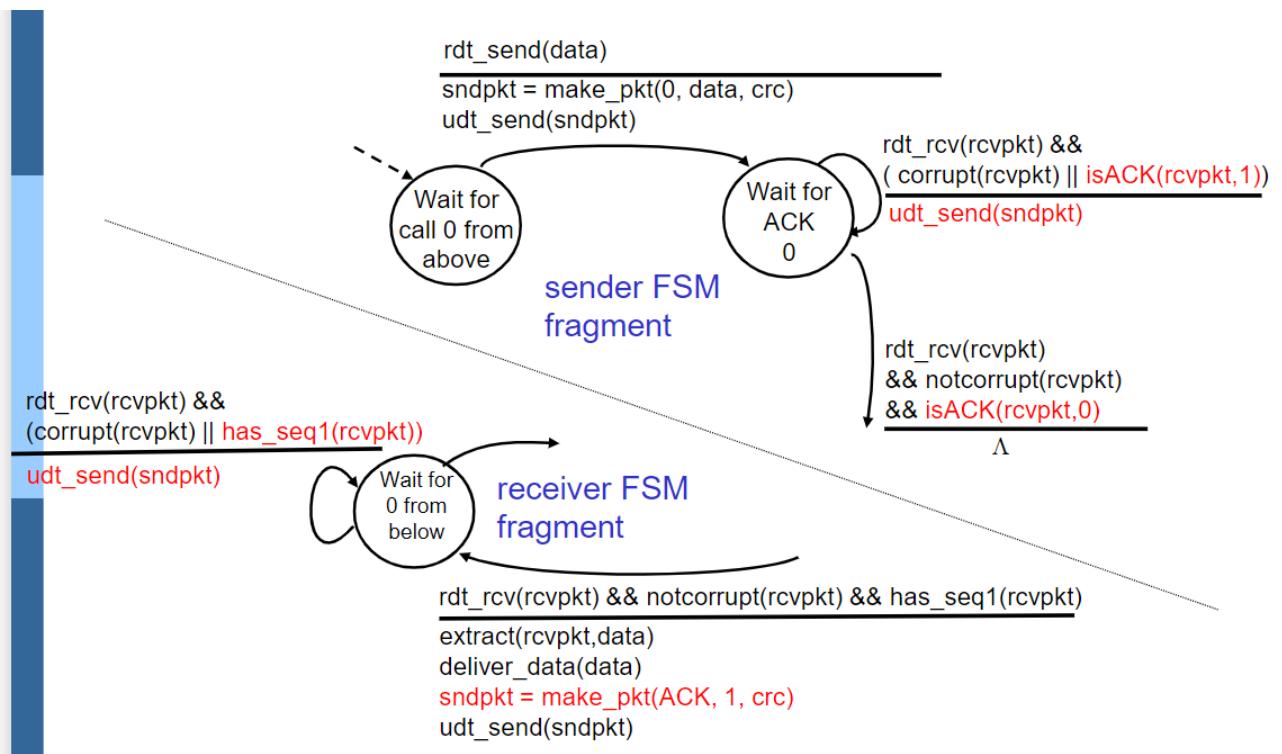
Questa implementazione va già meglio, ma può essere resa *maggiormente elegante*. Si vorrebbe utilizzare una sola forma di feedback, e non due. Pertanto ci si chiede, *esiste un modo di raggiungere lo stesso risultato usando solo ACK o NAK?*

### RDT Protocol 2.2 : La versione NAK - Free

Utilizzando solo ACK, potremmo decidere di ragionare in questo modo:

- Il ricevitore invia il numero di sequenza il cui ACK sta facendo riferimento.
- Se il trasmettitore riceve questo ACK in maniera integra, ma si sta aspettando un altro ACK, allora capisce che il ricevitore non sta capendo e rimanda il pacchetto

Essendo la sequenza lunga un solo bit, quando il ricevitore manda "un ACK sbagliato", è come se dicesse al trasmettitore "*vedi che ho capito fino a quello precedente*". In questo meccanismo (degli ACK duplicati) non è dunque affatto necessario utilizzare NAK, ma è necessario che il ricevitore riesca a trasmettere un'informazione in più (il numero di sequenza). Viene tuttavia **semplificata l'implementazione del ricevitore**, che ora fa la stessa cosa sia che il pacchetto sia corrotto, sia che il pacchetto sia di una sequenza inattesa. Nell'RDT 2.1 avrebbe inviato un NAK e un ACK (per sbloccare il trasmettitore), mentre adesso manderà sempre un ACK [*Numeros Seq Precedente*].



C'è però un ultimo problema, ovvero che il nostro protocollo non prevede **Packet Loss**:

- *Cosa deve fare il trasmettitore se l'ACK non arriva (perché si è perso il pacchetto trasmesso)*
- *Cosa deve fare il ricevitore se il pacchetto non arriva (perché si è perso l'ACK)*

Dobbiamo implementare sul trasmettitore un sistema per fare in modo che il ricevitore riceva comunque tutti i pacchetti, e dobbiamo fare in modo che il ricevitore continui a identificare eventuali duplicati.

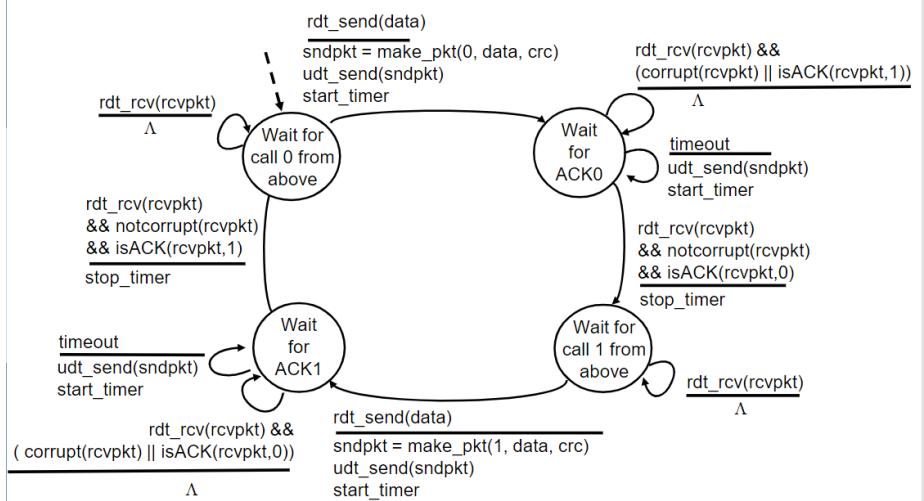
## RDT Protocol 3.0 : I Packet Loss

Come riconosco il Packet Loss, e come reagisco?

La soluzione è molto semplice: **Metto un Timeout, oltre il quale ritrasmetto il pacchetto.**

Dal punto di vista del ricevitore, lui deve solo "ricevere pacchetti", e sa già gestire i pacchetti duplicati. Di conseguenza il problema è solo del trasmettitore. Dovrà gestire un timer che, allo scadere del countdown, dovrà lanciare un'interruzione che **imponga la retrasmessione**.

Ovviamente se il timeout è scelto troppo prematuro, la risposta alla fine arriverà e si crea l'invio di un pacchetto duplicato (caso che però sappiamo il ricevitore sa già gestire). Viceversa se il timeout è scelto troppo "largo", allora è possibile che si intervenga troppo tardi su un packet loss prematuro.



Ciò che è vero è che **il timeout**

**deve essere paragonabile al RTT**, che a livello data Link è calcolabile con una certa precisione (a causa dell'assenza del passaggio nei router e, quindi, del ritardo di accodamento imprevedibile).

**Perché il trasmettitore semplicemente ignora i pacchetti ricevuti corrotti o con ACK errato?**

Il motivo consiste nell'evitare di generare troppe copie nel caso di timeout troppo prematuro.

Il protocollo potrebbe dichiararsi concluso, ma ci sono ancora dei problemi, e stavolta sono **di efficienza**. L'RDT Protocol 3.0 è infatti ancora di tipo stop - and - wait, e il trasmettitore non manda **nulla** se il precedente pacchetto non è stato ricevuto correttamente (eventualmente lo ritrasmette allo scadere del timeout se non riceve feedback positivo).

L'approccio stop - and wait è davvero limitante. Se il trasmettitore potesse inviare anche altro mentre aspetta un ACK le cose sarebbero migliori. Questo però introduce nuovi problemi:

- Come fa il trasmettitore a tenere traccia dei pacchetti spediti correttamente?
- Come può ritrasmettere esattamente solo i pacchetti che sono stati recepiti male?
- Come può il ricevitore garantire in ogni caso il deliver dei pacchetti in ordine?

## Trasferimento in PipeLine : Il Go - Back - N (GBN)

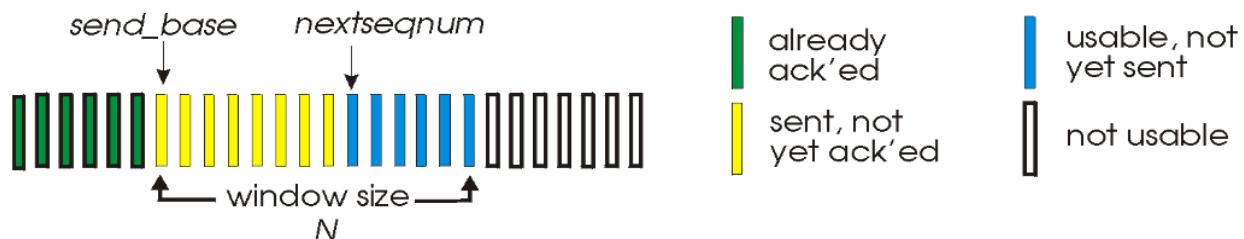
Il protocollo GBN si basa su un concetto fondamentale: il trasmettitore può mandare al più N pacchetti senza aver bisogno di ricevere alcun ACK. È quindi come se ci fosse una *finestra* sui pacchetti e che N rappresenti la grandezza su questi pacchetti "trasmissibili". Per questo motivo N è detto anche **dimensione della finestra** (window - size) e il protocollo è chiamato anche sliding - window - protocol.

I numeri di sequenza ai pacchetti non possono più stare su un bit, ma c'è bisogno di rappresentarli in accordo con la grandezza della finestra. In generale:

- Chiamarò **base** il più piccolo numero di sequenza di cui non ho ricevuto alcun ACK

- Chiamo **nextseqnum** il più piccolo numero di sequenza di pacchetti che non ho ancora inviato, ma che stanno nella finestra. In sostanza questo è il numero di sequenza del prossimo pacchetto che mando.

Individuo dunque ben quattro segmenti di numeri di sequenza utilizzabili:



- **Segmento [0, Base - 1]**: è l'intervallo dei numeri di sequenza corrispondenti a pacchetti che sono riuscito correttamente ad inviare al ricevitore (e di cui ho ricevuto l'ACK)
- **Segmento [Base, NextSeqNum - 1]**: rappresenta l'intervallo di quei numeri di sequenza corrispondente a tutti i pacchetti di cui devo ricevere l'ACK.
- **Segmento [NextSeqNum, Base + N - 1]**: rappresenta l'intervallo dei numeri di sequenza corrispondenti a pacchetti "inviabili", che *non escono fuori dalla finestra*
- **Segmento [Base + N, ... ]**: rappresentano pacchetti che però non sono ancora inviabili, perché la finestra scorre solo quando viene consegnato il pacchetto numero *base*.

Il protocollo si basa su due concetti fondamentali:

- C'è un ACK cumulativo. Fare ACK(n) significa "ho capito tutto fino al numero di sequenza n".
- C'è un solo Timer: è relativo al pacchetto mandato per primo e di cui non si è ancora ricevuto l'ACK. Nel nostro contesto, corrisponde al pacchetto di numero *base*.

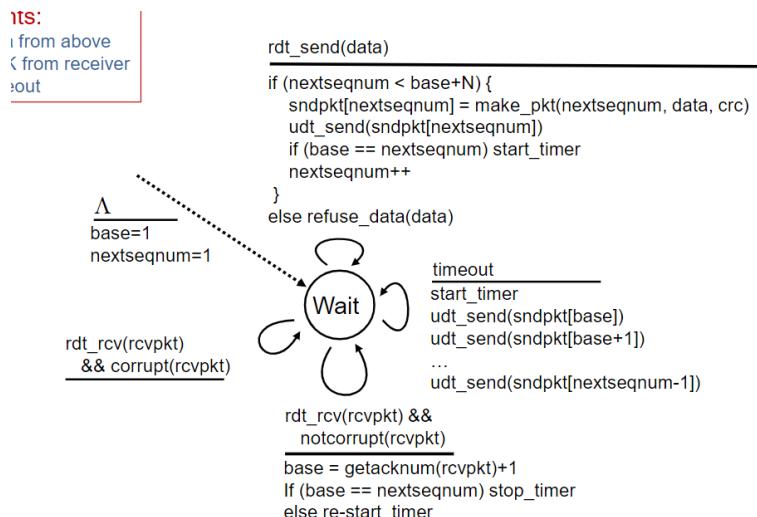
Il trasmettitore avrà il compito di inserire tra le linee di intestazione il numero di sequenza, incrementandolo circolarmente basandosi sui K bit sul quale è rappresentato.

Dopodiché, il trasmettitore deve stare attento a 3 eventi:

1. **Riceve dati dall'alto** : quando qualcuno invoca una *rdt\_send()*, il trasmettitore deve chiedersi se può inviare, ovvero deve controllare che la finestra non sia piena. La finestra è piena quando  $nextseqnum = base + N$ . Per uscire da questo stallo, è necessario incrementare *base*, e questo succede quando il ricevitore fa un ACK.

Se la finestra è piena, il trasmettitore solitamente *rimanda indietro i dati*, e ci si aspetta che l'upper layer riprovi dopo un po'. Oppure si può implementare un meccanismo di **sincronizzazione**, dove la *rdt\_send()* viene effettivamente chiamata solo se la finestra ha un posto libero.

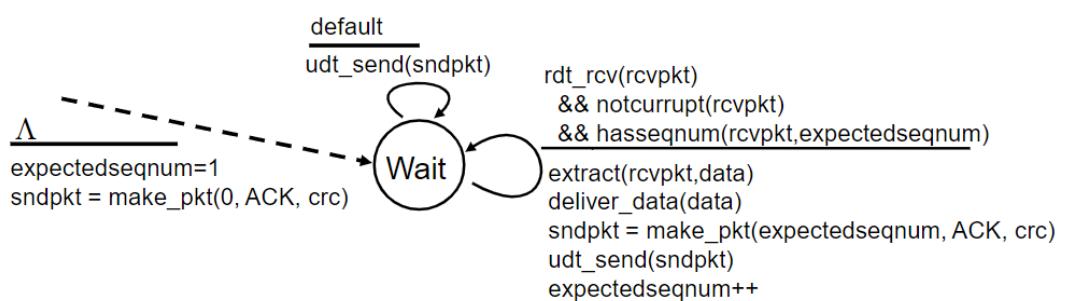
Se la finestra non è piena, il pacchetto viene inviato al ricevitore con numero di sequenza *nextseqnum*, e questo stesso indice viene poi incrementato.



2. **Riceve un ACK**: quando riceve un ACK(n), il trasmettitore fa scorrere *base* fino a n, essendo l'ACK cumulativo. Quando questo scorrimento fa scorrere *base* fino a *nextseqnum*, allora il timer viene fermato, altrimenti viene solo fatto ripartire.
3. **Rispondere al Timeout**: come abbiamo detto prima, il timer viene applicato sul pacchetto più vecchio di cui non si ha ancora l'ACK, e questo è indicato da *base*. Quando scatta il timeout, il trasmettitore **ritrasmette tutti i pacchetti da base ad nextseqnum - 1**.

Il ricevitore, d'altra parte, manda degli ACK cumulativi, e come informazione memorizza unicamente il prossimo numero di sequenza che si aspetta di ricevere. In particolare lui consegna un pacchetto per volta, e deve consegnarli tutti in ordine. Se un pacchetto arriva integro a *out-of-order*, lui **lo scarta e fa l'ACK dell'ultima cosa che**

**ha capito**. In questo modo il trasmettitore rischia di dover reinviare dei pacchetti che erano persino arrivati correttamente, ma perlomeno il ricevitore è molto semplice, non deve bufferizzare nulla!



Si noti che in questo approccio (ACK cumulativi) non avrebbe senso che il ricevitore bufferizzi. Se memorizzasse un *out-of-order*, finerebbe comunque per fare gli ACK in modo da segnalare "ho capito tutto fino a questo punto". L'*out-of-order*, a rigor di logica, non rientrerebbe mai in questo ACK, e pertanto sarebbe in ogni caso ritrasmesso!

Purtroppo un grosso difetto di questo sistema è che **il trasmettitore potrebbe finire per ritrasmettere pacchetti inutilmente**, quando in realtà non sarebbe servito! Per questo motivo si pensa ad un'altra metodologia.

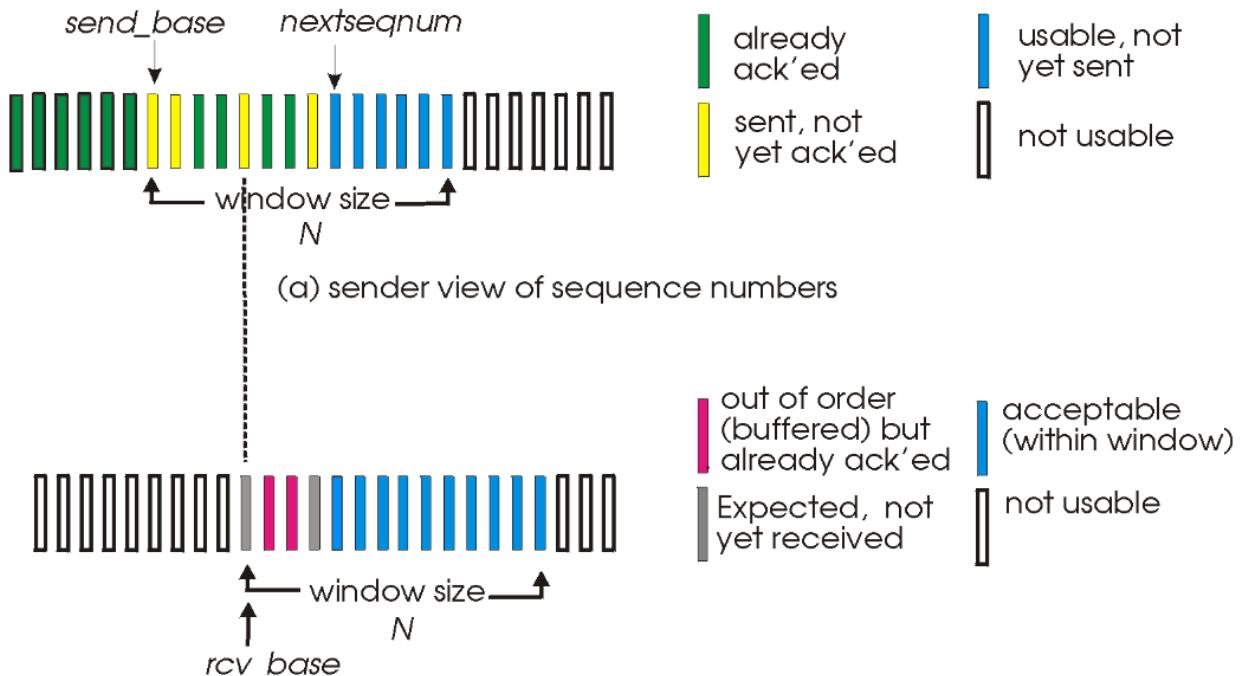
### L'ultimo Approccio : il Selective Repeat (SR)

Il protocollo SR cerca di eliminare il limite del GBN che consiste nel fatto che quando la finestra è molto grande se *Base* non viene ricevuto si finisce per rimandare l'intera finestra alla peggio, e questo è un problema che vogliamo non avere.

*Obiettivo : Eliminare la inutile retrasmessione introducendo degli ACK individuali per singolo pacchetto*

Vogliamo dunque che il ricevitore **bufferizzi** i pacchetti *out-of-order* che gli arrivino, e che quando faccia *deliver* spedisca eventualmente anche questi pacchetti (se siamo nel caso in cui ad esempio ha fatto l'ACK di *n+1*, quando arriva *n* spedisce entrambi!).

Ed ecco come cambiano i modi di ragionare circa i numeri di sequenza:



Eventi del Trasmettitore:

1. **Riceve Dati dall'Alto :** come in GBN, se il numero di sequenza passato è fuori finestra il dato viene rigettato (si deve scorrere la finestra, ovvero aspettare l'ACK di  $send\_base$ )
2. **Timeout del singolo pacchetto :** diversamente da GBN, c'è un Timer *per ogni pacchetto*. In questo modo quando scade questo timer è possibile ritrasmettere solo quel pacchetto. Si noti che è possibile generare diversi timer logici con un solo timer hardware, quindi tutto ok!
3. **Arrivo dell'ACK :** se mi arriva l'ACK di un pacchetto che sia diverso da  $send\_base$ , ma che comunque sta nella finestra, faccio in modo di memorizzare il fatto che quel pacchetto è stato ricevuto correttamente. Se arriva invece l'ACK di  $send\_base$ , scorro la finestra in avanti fino al primo di cui non ho ricevuto l'ACK.

Eventi del Ricevitore:

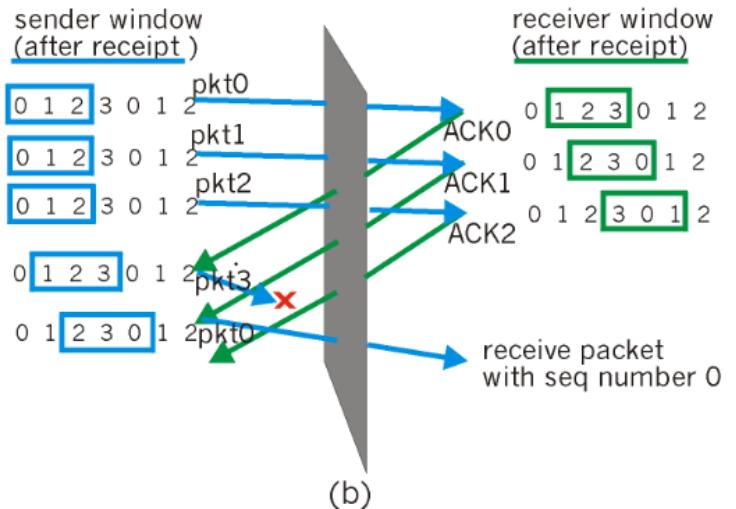
1. **Riceve Correttamente un Pacchetto nella sua finestra :** se il pacchetto ricevuto è diverso da  $rcv\_base$  si limita a bufferizzare il pacchetto, altrimenti aumenta  $rcv\_base$  fino al prossimo non ricevuto e si fa la deliver di tutti quelli mandabili (in ordine).
2. **Riceve Correttamente un Pacchetto precedente alla finestra :** se il pacchetto è "leggermente indietro", fino a  $rcv\_base - N$ , allora significa sicuramente che il ricevitore l'ha già spedito all'upper layer. È comunque necessario fare l'ACK di questi pacchetti più vecchi, perché il sender potrebbe averne bisogno per muovere in avanti la sua finestra (magari gli ACK precedenti non gli sono arrivati). Non è necessario fare l'ACK oltre  $rcv\_base - N$  perché tanto la finestra del sender è sicuramente più avanti (avendomi mandato  $rcv\_base - 1$ ).

*Il problema dei numeri di sequenza finiti:*

Nella vita reale il numero di sequenza è finito ed aumenta in maniera circolare. Se la grandezza della finestra è comparabile col numero di sequenza il ricevitore potrebbe non avvertire alcune differenze in alcune situazioni:

Nella figura a destra, il ricevitore vede il pacchetto 0 come un nuovo pacchetto e se lo prende, ma se il trasmettitore non avesse ricevuto gli ACK e avesse reinviato il pacchetto 0 iniziale (una copia), il ricevitore se lo sarebbe preso comunque!

In generale la grandezza della finestra deve essere <= della grandezza della sequenza!



## Il protocollo PPP: trasporto a livello Data Link

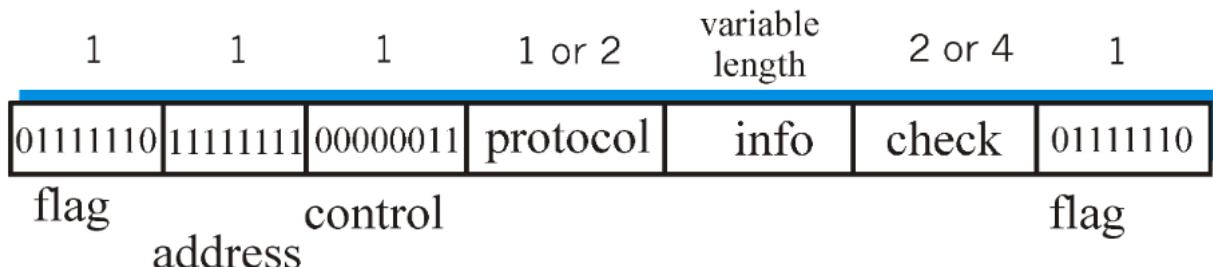
Nella pratica il trasporto a livello Data Link si ottiene mediante il protocollo **PPP (Point - To - Point Protocol)**. Il protocollo offre diversi servizi, e il primo fra tutti è consegnare un frame garantendo che possa essere usato anche da un piano più alto dello stack protocollare.

*Elenco di alcuni servizi :*

- **Framing** : il pacchetto che proviene dall'upper - layer è diviso in Frame
- **Error Detection** : per ogni frame si include un campo (*check*), in cui vengono inseriti dei bit CRC per poter permettere al ricevitore di fare detection dell'errore e richiedere in tal caso una retransmissione
- **Bit Transparency** : si vuole che all'interno del campo dati del frame si possa trasmettere *qualunque sequenza di bit*.
- **Configurare il Network Layer** : è possibile grazie a questo protocollo impostare le cose che servono per i protocolli di livello più alto (si può negoziare, ad esempio, un indirizzo IP), ecc.

È molto importante notare che al protocollo *non importa assicurare alcune cose*. In particolare non fornisce error correction e necessità di retransmissione. Inoltre non ha interessa nel fare flow control(ovvero il ricevitore non ha modo di dire al trasmettitore "rallenta") né tantomeno assicura che i pacchetti arrivino in ordine. Questi compiti sono delegati ai protocolli di livello più alto (come ad esempio il TCP!).

Il formato del frame che viene generato dal Protocollo è il seguente:



- Si apre e chiude con un **flag fissato a 01111110**, per aiutare il ricevitore a capire dove inizia e dove finisce la trasmissione
- Ha un campo **address** statico, 11111111, che ricorda la comunicazione in broadcast e nasce per supportare la comunicazione punto - a - multipunto.

- C'è un byte **control** (anch'egli fissato), che non serve a niente, ma è lì per possibili utilizzi futuri.
- Ci sono un paio di byte dedicati al **protocollo di livello più alto** a cui il frame è destinato
- C'è il campo *check* per fare error detection mediante i bit CRC
- C'è il campo *info* che contiene i dati veri e propri

La storia è tutta qui, non c'è niente da dire. Nasce però un problema, **come può il protocollo assicurare la bit transparency se c'è un bit costante di flag?**

Detto in soldoni, se l'utente vuole trasmettere 01111110 come fa a far capire al ricevitore che non è il flag di fine trasmissione?

La soluzione è raggiunta mediante un flag di escape, 01111101. Nello specifico:

- L'utente scriva tutti i dati che vuole
- Il protocollo, se fa lo "scan" di un byte di flag 01111110, lo codifica come 01111101 01111110. Se invece il protocollo fa lo "scan" di un byte di escape 01111101, allora viene codificato come 01111101 01111101
- In questo modo per il ricevitore è facile: se vede 01111101 01111101, traduce 01111101. Se invece vede 01111101 01111110, traduce 01111110

### Sender (byte stuffing)

- 01111110 → 01111101 01111110
- 01111101 → 01111101 01111101

### Receiver (byte unstuffing)

- 01111101 01111110 → 01111110
- 01111101 01111101 → 01111101

## I link ad Accesso Multiplo e relativi Protocolli

Finora abbiamo sempre trattati collegamenti **Point - To - Point**, in cui al più due host sono collegati in maniera diretta. In realtà esiste un altro modo di collegare host direttamente, ed è utilizzare un link condiviso, un **mezzo di broadcast**.

I mezzi di broadcast fanno in modo che tutti vedano tutto ciò che è trasmesso, e sono regolati dai **MAC Protocols** (Multiple Access Control). Questi protocolli devono assicurare che le trasmissioni vadano a buon fine.

*Perché, possono fallire?*

Essendo il collegamento diretto, tutti possono trasmettere a tutti, e di conseguenza se non si trasmette uno alla volta si crea **collisione**. Quando due nodi (*Nodo : Entità Ricevente o Trasmettente*) trasmettono un frame nello stesso momento, si crea una collisione e nessuno dei riceventi capisce nulla. Si spreca tempo e banda.

È necessario un tipo di politica, appunto garantita dai protocolli d'accesso multipla, che facciano agire in qualche modo i nodi *in sincronia*, in modo tale che le collisioni siano poche, rilevabili, e correggibili.

*Idealmente, su un mezzo di capacità R si vorrebbero 4 proprietà:*

1. **Proprietà della Piena Utilizzazione** : se c'è una sola persona che trasmette, allora questa deve essere capace di trasmettere a pieno bitrate R
2. **Caratteristica "Fairness"** : se ci fossero M nodi intenzionati a trasmettere, si vorrebbe che possano trasmettere tutti con rate  $R/M$
3. **Proprietà della "Decentralizzazione"** : non deve essere presente alcun elemento di centralizzazione. Ovvero tutti i nodi devono essere uguali, e non c'è nessun elemento specifico a cui devono fare riferimento (come un arbitro ad esempio)
4. **Semplicità** : semplice è bello e più efficiente

I protocolli MAC sono divisibili in tre grandi *categorie* (nonostante ce ne sia un'infinità in giro, il template di ragionamento è sempre lo stesso):

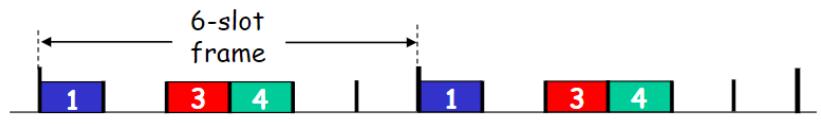
- I protocolli **a partizione di risorse** : sono protocolli che suddividono il link secondo regole logiche per permettere a più nodi di trasmettere contemporaneamente. Fanno parte di questa famiglia i protocolli TDMA, FDMA e CDMA
- I protocolli **ad accesso casuale** : sono protocolli in cui tutti trasmettono quando vogliono e deve essere prevista una corretta gestione della collisione, che avviene con probabilità molto elevata in situazioni di questo tipo
- I protocolli "*Taking Turns*" : sono protocolli in cui si prevede un meccanismo di turnazione dei nodi, in modo che capiscano quando possono trasmettere senza collidere con qualcun altro

### I protocolli a Partizione di Risorse : TDMA, FDMA e CDMA

Questo tipo di protocolli, come detto sopra, si basano sulla possibilità di suddividere la risorsa di trasmissione (il mezzo di broadcast), e lo fanno secondo diverse politiche. Ne analizziamo uno, il **TDMA** (**Time Division Multiple Access**).

Il TDMA divide l'intero asse temporale in frame temporali, e in ogni frame associa un certo numero di slot. Ad ogni slot è associato uno e un solo nodo, secondo diverse possibilità:

- Ogni nodo può avere a disposizione un solo slot *per ogni* frame
- Ogni nodo può avere a disposizione più slot *per ogni* frame
- Ogni nodo può avere un numero variabile di slot *ogni qualche* frame
- Una combinazione dei tre modi precedenti



Un metodo di questo tipo ha il vantaggio di **rendere nulla la possibilità di collisione**. Infatti se tutti sanno qual è il proprio slot, trasmettono solo in quel tempo. È sicuramente un approccio semplice, e sicuramente se M nodi vogliono trasmettere e assegno uno slot per ogni frame ad ogni nodo, tutti trasmettono a un rate pari a  $R/M$ . Ciò significa che da questo punto di vista il protocollo è altamente fair.

*Il protocollo non è però davvero fair*, e mostra le sue lacune quando non tutti i nodi sono attivi, ovvero quando non tutti i nodi vogliono trasmettere.

- Se in un frame c'è un solo nodo interessato a trasmettere, questo potrà trasmettere *comunque solo per il suo / i suoi slot assegnati*. Di conseguenza **non c'è la proprietà di Piena Utilizzazione**.
- Un frame che ha trasmesso e vuole ritrasmettere è **costretto ad aspettare il proprio Slot**, anche quando gli altri non vogliono trasmettere

Questi due svantaggi che si basano entrambi sugli slot *wasted* (sprecati) rendono il protocollo poco efficiente, e limitante dal punto di vista della piena utilizzazione.

Si può dimostrare che FDMA è equivalente a TDMA, e pertanto ne condivide pregi e difetti.

**NOTA STORICA : IL CDMA (Code Division Multiple Access)** è un protocollo in cui a dividersi è il codice di comunicazione. In particolare se ogni coppia di nodi trasmettente – ricevente utilizza una “lingua” diversa, allora tutti possono trasmettere contemporaneamente ed è garantito il fatto che il ricevitore “scarti le altre lingue” e capisce solo quella a lui riservata.

### I protocolli ad Accesso Casuale (1) : L'ALOHA

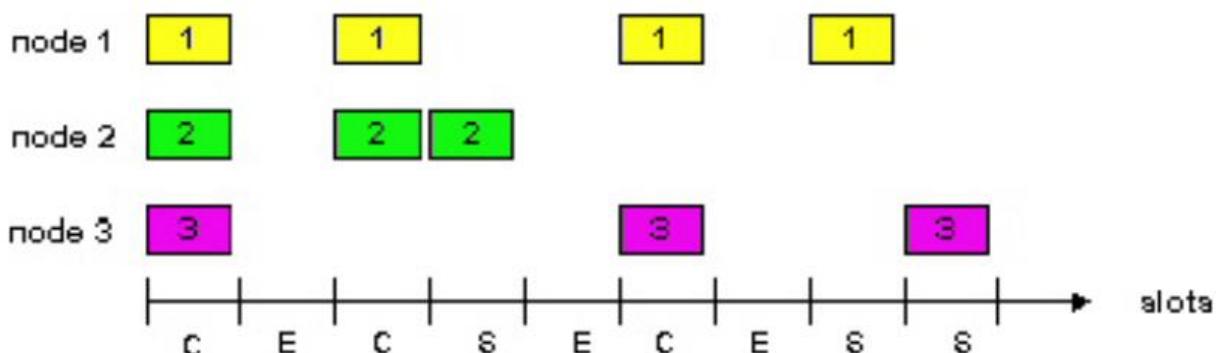
L'obiettivo del protocollo ALOHA è permettere ad ognuno di trasmettere con rate massimo R. Purtroppo questa possibilità è un'utopia perché, essendo un protocollo ad accesso casuale, **si presta molto bene alla possibilità di collisione**. L'idea di questi protocolli è *aspettare un tempo casuale* quando si fa detection di una collisione. Se tutti aspettano un *tempo casuale indipendente*, c'è la possibilità che alla fine si riesca a trasmettere. Vediamo come il protocollo ALOHA è implementato!

*Alcune premesse :*

- Tutti i nodi devono trasmettere pacchetti della stessa dimensione, L
- Il tempo è *diviso in slot* abbastanza grandi da permettere una trasmissione del pacchetto e una ricezione di un ACK. In questo modo la collisione è rilevata *prima del prossimo slot*
- Se si riceve un ACK corrotto, o se non si riceve, si assume conservativamente che ci sia stata una collisione
- I nodi sono sincronizzati e tutti sanno quando comincia il prossimo slot

Dopodiché il protocollo è semplice: quando un trasmettitore ha un pacchetto pronto, *lo trasmette nello slot successivo* allo slot corrente. Dopodiché :

1. Se non c'è collisione, va tutto bene e può preparare un pacchetto già per lo slot successivo (se ce l'ha già pronto).
2. Se c'è collisione, deve decidere se rimandare il pacchetto nel prossimo slot o meno. In particolare **avrà una probabilità P di ritrasmettere**, e una probabilità  $(1 - P)$  di non riprovare la trasmissione. Se vince la probabilità P, e nello slot successivo c'è di nuovo collisione, si ripete lo stesso e identico procedimento.



*Vantaggi dell'ALOHA*

- Sicuramente gode della proprietà della **Piena Utilizzazione**: quando un nodo trasmette, trasmette a full rate
- È semplice!
- È quasi completamente Decentralizzato: l'unico elemento di sincronia che devono avere i nodi è il clock, così che tutti sappiano correttamente quando inizia il prossimo slot

### Svantaggi dell'ALOHA

- **Non è fair**: essendo un gioco di probabilità, lo stesso nodo può trasmettere più volte prima di qualcun altro
- **Wasted Slots**: ci possono essere momenti in cui nessun nodo becca la probabilità P!
- **Alta probabilità di collisione**
- **La collisione deve essere rilevata velocemente**, prima dell'inizio dello slot successivo

Si può anche fare un ragionamento circa l'efficienza del protocollo, ovvero posso ricavare una percentuale di slot utili in cui qualcosa viene trasmesso correttamente

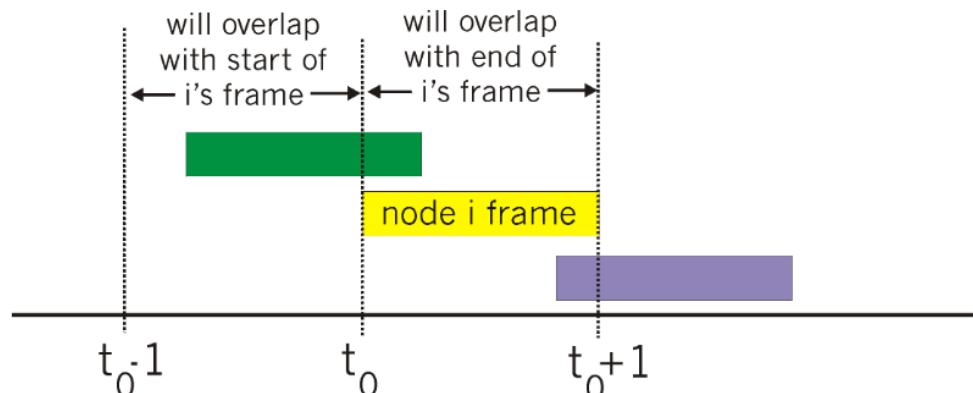
1. Sia  $N$  il numero dei nodi *attivi*, che hanno qualcosa da trasmettere. La probabilità che un nodo in fase di collisione decida di trasmettere è  $P$ . La probabilità che nessun altro trasmetta, e che quindi non ci sia collisione, è  $(1 - P)^{N-1}$ . Dunque la probabilità di riuscita è  $(\text{Trasmetto}) * (\text{Nessun Altro Trasmette}) = P(1 - P)^{N-1}$ .
2. La probabilità che **un qualunque nodo** trasmetta è che succeda quanto detto sopra *per ogni nodo*, e dunque è pari a  $NP(1 - P)^{N-1}$ .
3. Posso trovare  $P^*$  che massimizza questa probabilità, che diventa dunque  $NP * (1 - P^*)^{N-1}$
4. Portando questa probabilità al limite, quando  $N$  diventa molto grande (tende all'infinito), scopriamo che la probabilità è  $1/e = 0.37$

In conclusione c'è il 37% di possibilità che si trasmetta correttamente, ovvero che solo **il 37% di slot sia "utile"**! È davvero poco!

Prima dell'ALOHA slotted esisteva un **ALOHA PURO**, non slotted. Questo si basava su due concetti:

- Era davvero decentralizzato: non c'era alcuna forma di sincronizzazione, e tutti mandavano appena il frame raggiungeva il data link – layer.
- Se ci fosse stata collisione, il frame sarebbe stato rimandato con probabilità  $P$  dopo un tempo uguale al tempo che ci vuole a trasmettere un frame.

Purtroppo la centralizzazione pura introduce una efficienza peggiore. Se infatti i frame possono essere mandati a un qualunque istante, sia  $T_0$ , allora bisogna sperare che **nessuno stia trasmettendo nell'intero**



**intervallo  $[T_0 - 1, T_0 + 1]$** , se 1 è il tempo di trasmissione di un frame. Ciò significa che la probabilità è (*Trasmetto*) \* (*Nessun Altro Trasmette prima di me*) \* (*Nessun Altro Trasmette Dopo di me*). La sincronia dello slotted ALOHA garantiva invece che all'inizio dello slot non ci fossero problemi da "ciò che c'era prima".

Dovendo preoccuparmi del doppio delle cose, la probabilità che un nodo target ce la faccia è  $P(1 - P)^{2(N-1)}$ .

Portando al limite questa situazione, l'efficienza è  $1/2e = 0.18$ , l'esatta metà dello slotted ALOHA. L'approccio è orrendo!

### I protocolli ad Accesso Casuale (2) : CSMA e CSMA/CD

Il difetto dei precedenti protocolli (ALOHA puro e slotted) è che sono **incivili**. Parlano anche mentre qualcun altro sta parlando, e non gli importa nulla di rovinare la trasmissione di qualcun altro, incrementando il numero di slot wasted.

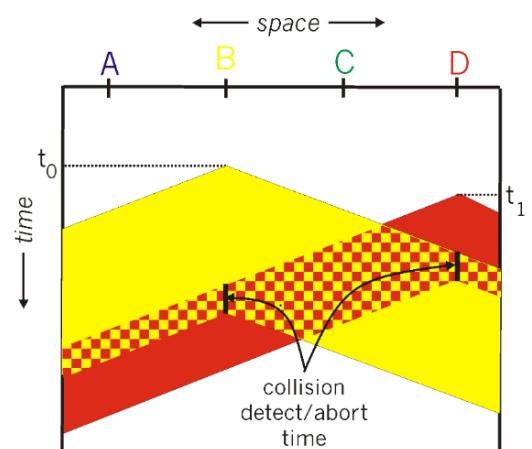
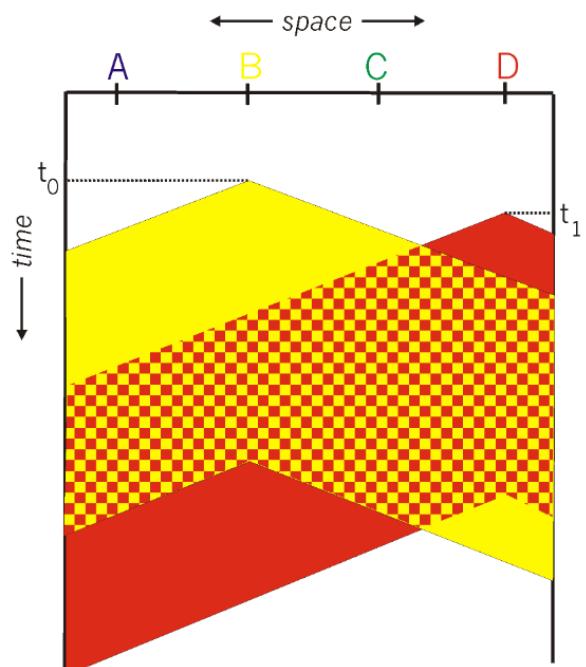
Se noi umani parlassimo continuamente a raffica, nessuno capirebbe mai nulla (se non appunto con una bassa probabilità come in ALOHA). Ma allora *come facciamo noi umani?*

- **Ascoltiamo prima di parlare** : se quando vogliamo parlare qualcuno sta parlando, evitiamo di parlare (o almeno dovremmo !!!). A livello network, significa che un nodo trasmette solo quando non vede trasmissioni per un certo tempo. Questa proprietà è chiamata **carrier - sensing**, e sta alla base dei protocolli della famiglia CSMA (**Carrier - Sensing Multiple Access**)
- **Se quando cominciamo a parlare, qualcuno comincia insieme a noi, smettiamo di parlare** : a livello network significa che i nodi che, non sentendo nessuno, hanno cominciato a parlare devono essere in grado di riconoscere una collisione, e questo sta alla base dei protocolli della famiglia CSMA / CD (ovvero **con Collision Detection**)

*Ma davvero quindi i CSMA possono incontrare collisioni così spesso? Non è rarissimo che due comincino a parlare allo stesso esatto momento?*

Purtroppo esiste **il ritardo di propagazione**, ovvero ci vuole un tempo (piccolo, ma non nullo) per raggiungere i nodi, in base a quanto distanti sono.

Nel diagramma spazio - tempo qui a fianco, il nodo B comincia a trasmettere a un tempo  $T_0$  perché il mezzo trasmittivo è libero, ma non riesce a raggiungere in tempo il nodo D che, al tempo  $T_1$ , vede il mezzo trasmittivo libero perché B non l'ha ancora raggiunto. Ciò significa che dopo un breve tempo i pacchetti di B e D faranno interferenza, e **si perderà tutto il tempo relativo alla trasmissione dell'intero pacchetto**, perché B e D non sono in grado, in questa forma di protocolli, di rilevare la collisione.



*E se invece fossero in grado di capire che hanno fatto interferenza?*

Se riuscissero a capire di aver fatto interferenza, i due interromperebbero il prima possibile la trasmissione,

rendendo minore il tempo sprecato a trasmettere roba danneggiata!

**Quando ci riprovano?** Dipende dal protocollo, ma potremmo assumere che ci riprovano dopo un tempo casuale (indipendente l'uno dall'altro).

Esamineremo come funziona CSMA/CD più in là, quando parleremo dell'Ethernet, enjoy the future!

Purtroppo i protocolli ad Accesso Casuale, per loro natura, non sono affatto Fair.

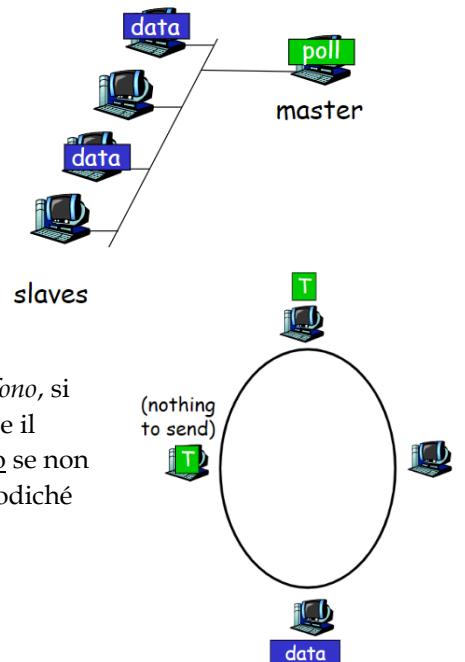
### I Protocolli "Taking Turns" : I più fair sulla piazza

I protocolli Taking Turns vogliono rendere l'accesso multiplo il più *fair* possibile, garantendo, appunto a turni, la trasmissione.

Ce ne sono tantissimi, ma ne ricordiamo due in particolare.

#### Polling Protocol

In questo protocollo, esiste un Nodo Master che dice a tutti quando possono trasmettere, e gli dice anche quant'è il numero massimo di frame che possono inviare sul canale di broadcast. Dopodiché, quando il master non vede più traffico sul canale, dà il permesso ad un altro nodo di inviare il / i suo / suoi frame, e così via, **con politica Round - Robin** così che tutti possano prima o poi trasmettere.



#### Token - Passing Protocol

In questo protocollo, che è l'equivalente umano di *parla solo se hai un microfono*, si passa a giro un **Token** che rappresenta il permesso di parlare. In particolare il nodo 1 quando riceve il token lo passa immediatamente al nodo successivo se non deve trasmettere, altrimenti trasmette un numero massimo di frame e dopodiché passa il token.

#### Purtroppo l'essere fair si paga:

- In entrambi i protocolli c'è la mancanza di piena utilizzazione : se un solo nodo vuole trasmettere, deve aspettare alla peggio N-1 passaggi del token o N-1 ordini del master (in particolare il master risveglia a giro e se non hanno intenzione di trasmettere allora risveglia qualcun altro, ma questo richiede del tempo)
- In entrambi i protocolli è **presente un single - point of failure** : nel caso del master è il master stesso, nel caso del token è il token stesso (se si rompe il nodo che ha il token, bisogna perdere tempo a prevedere operazioni di recovery del token).

## Le Reti Locali : Local Area Network (LAN)

Cerchiamo adesso di capire come si struttura una LAN, ovvero un rete locale a un edificio, un condominio, una scuola, ecc.. È un mezzo di broadcast condiviso tra tutti? Se si com'è questo mezzo? Si usano gli indirizzi IP per comunicare?

Ci sono un po' di questioni da capire per bene, e la prima è l'**addressing**, ovvero come i nodi di una rete locale si devono riferire l'un l'altro.

### Gli indirizzi MAC

L'idea è che ogni Host abbia un **adattatore** che chiameremo NIC (Network Interface Card). Queste schede, o meglio, queste **schede di rete**, sono indirizzate con un numero binario su 48 bit detto Indirizzo MAC.

Questi indirizzi MAC, normalmente espressi come 6 coppie di numeri esadecimale, operano a livello Data Link, e sono diversi dagli indirizzi IP. In particolare:

- Se un'interfaccia riceve un Frame, riesce a capire dall'Header l'indirizzo MAC del destinatario
- Confronta questo indirizzo MAC con il proprio. Se i due coincidono, **spedisce il datagram** a livello Network, altrimenti *scarta il frame*.

Non esistono due NIC con lo stesso indirizzo MAC nel mondo. In particolare quando un costruttore vuole realizzare un adapter chiede all'IEEE di fissare i 24 bit più significativi del suo indirizzo MAC. È poi libero di fissare i restanti 24 bit come vuole.

Un indirizzo MAC **non cambia se la NIC si sposta**. Viceversa un indirizzo IP dipende dalla rete a cui è collegato l'Host, ed è dunque soggetto a variazione se si sposta.

All'interno di una rete locale LAN, tutti i nodi hanno dunque un certo indirizzo MAC diverso da tutti gli altri. *Ma è davvero necessario avere un indirizzo MAC? Non si possono usare gli indirizzi IP e basta?*

Ci sono diverse ragioni per cui è comodo tenere distinte le due cose:

1. Le reti LAN sono pensate per essere qualcosa di livello Data Link, una struttura di nodi che riesce ad accedere all'internet mediante un router. Tuttavia utilizzare gli indirizzi MAC aumenta la versatilità della trasmissione, che può accettare in questo modo protocolli di livello Network diverso da quello IP. In particolare nel frame di livello Data Link sarà possibile inserire un campo *type* che specifica il protocollo di livello network a cui il frame è diretto. Se questo protocollo è diverso da IP, allora significa che un indirizzo MAC *neutro rispetto ai protocolli di livello Network* è essenziale!
2. Se non esistesse un indirizzo MAC, ma gli adapter fossero identificati dall'indirizzo IP, allora questo indirizzo dovrebbe essere messo all'interno della memoria interna dell'interfaccia. **Questo significa che l'interfaccia andrebbe riconfigurata ad ogni spostamento, e questo non lo vogliamo!**
3. Si potrebbe pensare che gli indirizzi MAC potrebbero non esistere affatto. In particolare quando il frame giunge a un nodo, questo scapsula le informazioni di livello Data Link e spedisce il datagram a livello Network. Sarà poi compito del livello Network confrontare l'indirizzo IP e decidere se il datagram è davvero per quel nodo o meno. Un approccio di questo tipo sarebbe **totalmente inefficiente**, in quanto presupporrebbe che deve essere spedito al livello network qualsiasi frame passi sulla LAN, il che è inutilmente dispendioso.

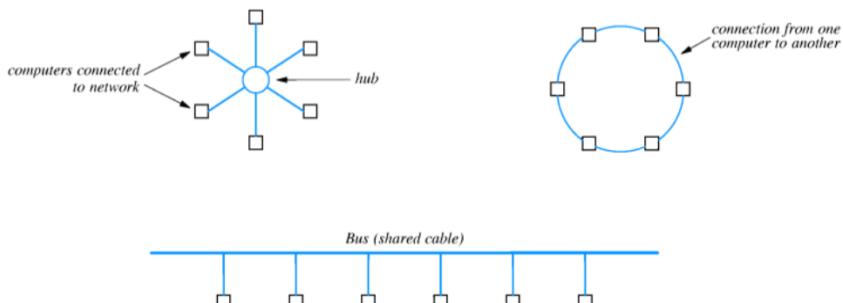
Non si dimentichi mai che gli indirizzi MAC sono qualcosa di livello Data Link. Due Nodi che vogliono comunicare utilizzando gli indirizzi MAC devono essere **direttamente collegati**.

Esiste infine l'indirizzo MAC *di broadcast*, che è sempre FF-FF-FF-FF-FF-FF, che permette di trasmettere il frame a tutti quelli che stanno sulla rete locale.

## Ethernet(1) : Introduzione

Prima di parlare di Ethernet, che definisce uno standard di formato dei frame, pensiamo prima a *come* sono collegate le reti in LAN. Esistono principalmente tre tipi di collegamento : a *bus*, a *stella*, e ad *anello*.

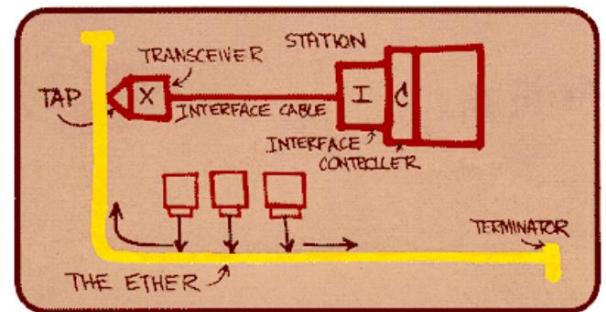
- **Il collegamento a Bus** è quello tipico dei mezzi trasmissivi di tipo broadcast. Ogni nodo è collegato allo stesso filo. La possibilità di collisione esiste coerentemente col fatto che il filo è purtroppo uno solo. Inoltre il bus ha due terminazioni. Se permettiamo al segnale di "rimbalzare" sulla terminazione, avviene un collisione anche quando a trasmettere è un solo nodo. Per questo motivo è necessario fare in modo che le terminazioni "assorbano" il



segnaletico, così che non rimbalzi o comunque che ne rimbalzi così poco da non avere abbastanza potenza da essere ascoltato. Nei fili in cui il segnale trasmissivo è elettrico, si pongono alle terminazioni alte resistenze, così da dissipare molta potenza.

- **Il collegamento a Stella** è equivalente a un *bus arrotolato*. Il “centro dell’arrotolamento” è spesso un dispositivo detto Hub. L’Hub è in generale un dispositivo **trasparente**, e pertanto non abbassa, né tantomeno fa scomparire, il rischio di collisione.
- **Il collegamento ad Anello** è una tipologia di bus in cui ogni nodo è collegato a due nodi adiacenti, a mo’, appunto, di anello. Se ogni ramo dell’anello è full – duplex (doppio anello) effettivamente non ci sono più collisioni, altrimenti purtroppo restano.

Ethernet è una tecnologia che prevedeva che ogni Nodo avesse un **adapter** con un certo MAC address, e che tutti gli adapter fossero collegati mediante bus di tipo broadcast in cavo coassiale (con terminazioni a resistenza alta). Il bus veniva chiamato **Ether (Etere)** da cui il nome della tecnologia. L’adapter era direttamente collegato all’etere, e poi questi era collegato all’interfaccia mediante il cavo di interfaccia (cavo di rete).



### Ma perché Ethernet è così famoso?

- Ethernet fu la prima tecnologia Wired – LAN ampiamente diffusa sul mercato. Ciò portò gli amministratori di reti a studiarla e a focalizzarsi su di esse, così da rendere più difficile un “cambio di rotta”
- Le altre tecnologie (come ATM) erano decisamente più costose e complesse dell’Ethernet, che invece era molto semplice
- Un motivo per abbandonare Ethernet sarebbe stato un lento bit rate, ma questo si è evoluto, da 10Mbps a 100Mbps, fino a 10Gbps!!

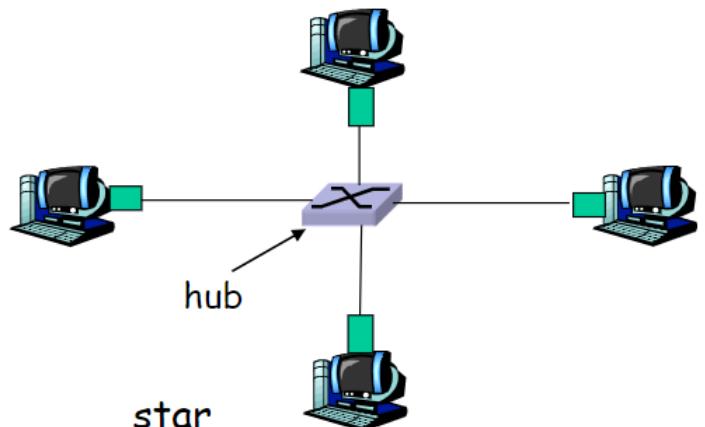
Col tempo Ethernet si è evoluto in una struttura *a stella* che prevedeva un **Hub**.

Che vantaggi ci sono ad utilizzare un Hub?

L’Hub è un dispositivo piuttosto stupido. È un amplificatore che lavora sul bit. Quando gliene arriva uno, il suo lavoro è quello di *amplificarlo in potenza* (rigenerarlo) e trasmettere **su tutte le interfacce di uscita** questo bit.

L’Hub dunque non elimina affatto la possibilità di collisione (lavora sul bit, non sul frame, e inoltre su tutti i link di uscita).

Inoltre, l’Hub non riesce nemmeno a rilevare collisione, quello è compito degli End – System che si collegano all’Hub con doppino telefonico.



Ciò fa sì che sia nella struttura a stella con Hub, sia nella struttura a basi condivise, i nodi vivono nello **stesso unico dominio di collisione**. Se collidono in due, collidono tutti!

### **Ethernet(2) : Il protocollo CSMA/CD**

Abbiamo prima lasciato in sospeso come un host potesse fare rilevazione di collisione. Se infatti il nodo ci riesce, si risparmia un sacco di tempo (i nodi smettono di trasmettere e si torna a una situazione di idle bus).

Cerchiamo di Descrivere cosa deve succedere:

- Il nodo attende che la linea sia idle. Se l'adapter non rileva "incoming energy" per un certo tempo, che nell'ethernet corrisponde a 96 bit - time, allora comincia a trasmettere, altrimenti si ferma ad aspettare l'idle per questo tempo. (Nota : Sull'ethernet a 10Mbps,  $96 \text{ bit} = 9,6 \times 10^{-6}$  microsecondi).
- Se il nodo riesce a cominciare la sua trasmissione, fa comunque un monitoring del canale. **Se rileva energia in ingresso rileva una collisione**, altrimenti se riesce a terminare la sua trasmissione senza aver rilevato energia dichiara conclusa la situazione.
- Se ha rilevato una collisione, deve **aspettare per un tempo casuale**, e riprovare la trasmissione.

*Come lo scelgo questo tempo casuale?*

L'intervallo di tempo che devo aspettare prima di ritrasmettere non può essere né troppo grande né troppo piccolo. Se l'intervallo è troppo piccolo, ma ci sono tanti nodi che vogliono trasmettere, è più probabile che i nodi collidano di nuovo! Viceversa, se l'intervallo è troppo grande, e ci sono pochi nodi che vogliono trasmettere, allora è probabile che vengano creati momenti in cui si crea dell'idle indesiderato!

L'ideale sarebbe trovare una soluzione che **aumenti l'intervallo insieme al numero dei nodi!**

L'algoritmo utilizzato da Ethernet è **Exponential Backoff**. Il nome deriva dal fatto che il tempo di backoff (ovvero il tempo che aspetto prima ritrasmettere) è esponenziale nel numero dei nodi. Vediamo perché:

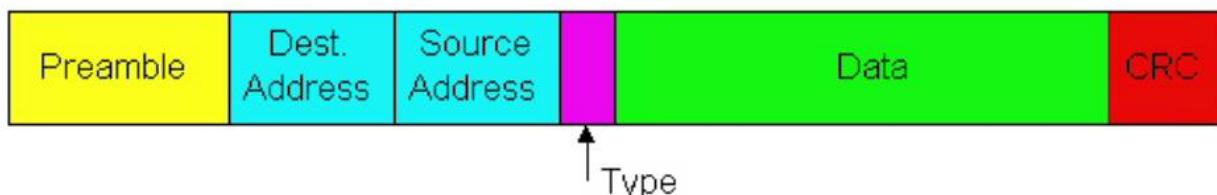
Sia N il numero di collisioni che il nodo sta vivendo **consecutivamente**. All'N-esima collisione, deve scegliere con uguale probabilità un numero K appartenente all'intervallo  $\{0, 1, 2, \dots, 2^N - 1\}$ .

- Alla prima collisione deve scegliere con probabilità 0.5 un numero tra  $\{0, 1\}$
- Alla seconda collisione ( $N = 2$ ) deve scegliere con probabilità 0.25 tra  $\{0, 1, 2, 3\}$
- Alla decima collisione ( $N = 10$ ) deve scegliere con ugual probabilità tra  $\{0, 1, 2, \dots, 1023\}$

NOTA : In ogni caso, l'intervallo non può diventare più grande di  $\{0, 1, 2, \dots, 1023\}$

NOTA2 : Quando il nodo fa esperienza della sua 17esima collisione (numero della sfortuna), ci rinuncia.

### Ethernet (3) : Struttura del Frame



La struttura del frame Ethernet è stata modificata negli anni, ma vediamola nel suo aspetto originale :

- **Campo "Data" (46 - 1500 byte)** : contiene il datagram che è stato incapsulato. Normalmente è un datagram IP, ma l'Ethernet supporta anche altri protocolli di livello network. Il campo data ha dimensione minima (46 byte) e dimensione massima (1500 byte). Ciò significa che se un datagram eccede queste dimensioni, deve essere frammentato in più datagram (ma questo lo vedremo più avanti). Se un datagram non arriva alle dimensioni mimime, si devono mettere dei bit in più per arrivare a 46, e sarà poi il protocollo di livello network, guardando la lunghezza effettiva del datagram, a fare l'unstuff dei bit aggiunti.
- **Destination Address (6 byte)** : è il MAC address del destinatario. Tutti ricevono il frame nella versione originale di Ethernet. Un nodo accetta il frame se Destination Address = Proprio MAC address **oppure** se Destination Address = Indirizzo MAC di broadcast

- **Source Address (6 byte)** : è l'indirizzo MAC del nodo che sta trasmettendo il frame. Si presta particolare attenzione a questo campo, che è necessario per permettere il self - learning degli Switch (vedi avanti)
- **Campo Type (2 byte)** : è la dimostrazione della versatilità della tecnologia, che si adatta a diversi protocolli di livello network. Il campo type serve infatti a specificare a quale protocollo di livello network spedire il datagram (IP, ARP, ...).
- **CRC (4 byte)** : questi sono 4 byte di ridondanza per permettere al ricevitore di fare error detection. Tuttavia **il ricevitore si limita a buttare via il frame se corrotto**, non dà alcun segnale di risposta! È il non aver spedito il frame ai livelli superiori che fa accorgere il TCP (livello 2) che qualcosa non sta funzionando. Per questo motivo il TCP può decidere di far inviare un "NAK". Non si confonda tuttavia la questione: **Ethernet non fornisce alcun tipo di trasferimento affidabile**, sono protocolli superiori a doversene preoccupare!
- **Preambolo (8 byte)** : il preambolo serve al ricevitore a sincronizzarsi col trasmettitore. C'è bisogno di sincronizzarsi perché i due clock hanno un **drift** (ovvero sono sfasati anche se di poco). Ciò significa che, anche campionando all'esatta metà dell'impulso, è possibile che lo sfasamento corrompa i dati al crescere della lunghezza del frame. Per questo motivo il preambolo serve al ricevitore per *ricevere informazioni sul clock del trasmettitore*. Il preambolo è costituito da 7 byte 10101010 e da un ottavo byte 10101011. Questa sequenza manda dunque esattamente il clock del trasmettitore (l'ottavo byte che finisce con due bit uguali serve a segnalare che "la roba importante sta arrivando").

*NOTA : Non era necessario mandare dei byte con tutti i bit alternati, è possibile fare uno XOR col clock del trasmettitore per ottenere comunque informazioni (Manchester Encoding)!*

Il servizio Ethernet è un servizio di tipo **connectionless**, ovvero non ha bisogno di instaurare connessioni preliminari e ogni frame viaggia in maniera indipendente dagli altri della stessa sequenza di frame.

Per indicare il tipo di tecnologia Ethernet che si sta implementando, che si è ampliata nel corso degli anni, si sono creati degli acronimi diversi, tutti del tipo XBASE-Y. BASE indica il fatto che la trasmissione supporta solo comunicazione di tipo Ethernet, X indica la velocità (10, 100, 1000, 10G), e Y indica il mezzo trasmisivo di livello fisico (T per doppino telefonico, F per fibra, ecc.).

## CAPITOLO 4 : LA NASCITA DELLE RETI SWITCHED

Cerchiamo di capire in questa parte in cosa differisce una rete a connessione diretta da una rete switched.

Restiamo nell'ambito delle LAN : l'ethernet con Hub è davvero la possibilità migliore che abbiamo? In realtà no, esiste un dispositivo più intelligente : lo **switch**.

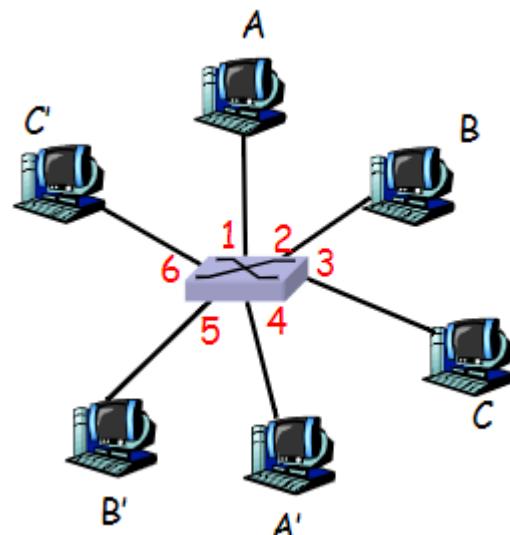
### Dallo switch singolo alle reti VLAN

#### Link - Layer Switch

Ogni nodo ha un collegamento dedicato con lo switch, e tutti possono trasmettere contemporaneamente a questo.

Lo switch è dotato di una **switch table** che collega gli indirizzi MAC a una certa interfaccia d'uscita. In particolare, la tabella ha record del tipo ( *MAC Address | Interfaccia di Uscita | Timestamp* ). Dopodiché le azioni che uno switch fa si riducono a due : **Filtering e Forwarding**.

1. Lo switch è in grado di **bufferizzare** i Frame. Quando ne riceve uno, guarda il MAC address del destinatario e consulta la sua tabella
2. Se nel record relativo a quel MAC trova come



interfaccia di uscita la stessa da cui è arrivato il frame, opera un'operazione di *filtering*, e scarta il frame.

3. Se c'è un'interfaccia di uscita diversa opera il *forwarding*, e instrada il frame su quell'interfaccia di uscita
4. **Se non esiste un Record Relativo a quel MAC Address destinatario, lo switch manda in broadcast a tutte le interfacce di uscita.**

Tutti i collegamenti sono in full duplex, ovvero non c'è collisione se il nodo trasmette e contemporaneamente riceve dal frame. **Ogni nodo ha il suo dominio di collisione**, e pertanto le collisioni sono totalmente eliminate!

Abbiamo già accennato il fatto che lo switch è un dispositivo intelligente, ma non ci riferivamo alle sole operazioni di Filtering e Forwarding, ma anche al fatto che lo switch è Self - Learning. In particolare è in grado di riempire la sua tabella autonomamente, senza alcun bisogno di far intervenire un amministratore di rete. Questo rende il dispositivo di tipo **plug - and - play**, in quanto l'unica cosa che bisogna fare è connettere i nodi della LAN allo switch. Se un Host viene rimosso, pace.

Un host potrebbe cambiare posizione o addirittura essere rimosso dalla LAN, e qui entra in gioco il *timestamp* inserito nei vari record delle tabelle. Se il timestamp indica il momento in cui quel record è stato inserito, si possono implementare delle politiche di **aging** secondo le quali dopo un certo periodo in cui non si riceve un frame da un certo MAC address si elimina la riga relativa dalla tabella.

Come è riempita la tabella in maniera Self - Learning?

1. Quando arriva un MAC address source che non è registrato in tabella, si registra il record relativo nella tabella (so da dove mi è arrivato il frame)
2. Quando non trovo la riga relativa al MAC address destination faccio il flooding (**mando in broadcast**)
3. Ciò implica che la tabella riesce a registrare le locazioni di tutti i nodi se e solo se tutti trasmettono.
4. Vale sempre l'operazione di **filtering** secondo la quale il frame viene scartato se è destinato alla stessa interfaccia che lo ha mandato

Infine, esaminiamo i vantaggi che gli switch hanno portato alla luce :

1. **Abbiamo eliminato le Collisioni** : non si spreca banda inutilmente, il throughput che si viene a creare è la **somma dei rate dei link nodo - switch**.
2. **Possibilità di introdurre Link Eterogenei** : i link che connettono il nodo allo switch possono essere di diverso tipo (sempre rispettando lo standard che descrive Ethernet). Questa possibilità non era possibile nell'Ethernet originale a bus condiviso.
3. **Management Semplificato** : i nodi malfunzionanti possono essere semplicemente disconnessi dallo switch
4. **Sicurezza Aumentata** : anche se esiste la possibilità che si riesca a prendere il controllo dello switch, sniffare i pacchetti è diventato molto più difficile, in quanto non si possono ricevere i frame se lo switch trova l'interfaccia di uscita del giusto destinatario. È comunque possibile l'attacco di **avvelenamento dello Switch**, in cui l'attaccante manda tanti frame con diversi Source Address, in modo da riempire la switch table. In questo modo lo switch sarà costretto a trasmettere in broadcast, e questo tipo di trasmissione è comunque sniffabile!

## Interconnettere Switch

Se l'obiettivo è creare una LAN più o meno estesa, magari perché si vuole offrire il servizio a un intero palazzo o addirittura a un campus, uno switch potrebbe non bastare. In particolare potremmo trovarci

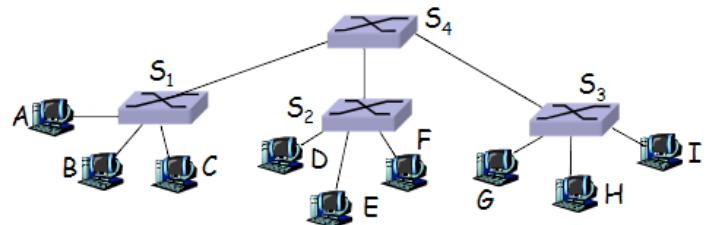
davanti all'esigenza di comprare uno switch con davvero troppo che, ammesso anche esista, finirebbe per costare così tanto da contrastare l'economicità della tecnologia Ethernet.

Si può implementare per risolvere questo problema una **struttura gerarchica**, con switch a più livelli.

*C'è bisogno di modificare il modo di riempire Switch Table?*

Inaspettatamente, la risposta è **no!** Immaginiamo che l'Host A debba mandare un messaggio all'Host I :

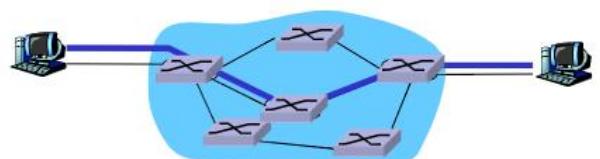
1. Il frame giunge in prima battuta allo Switch S<sub>1</sub> : lui instrada il frame sull'interfaccia di uscita che porta allo Switch S<sub>4</sub>.
2. Lo switch S<sub>4</sub>, che come source address continua a vedere quello di A e come destination continua a vedere quello di I, instrada sul link di uscita che porta ad S<sub>3</sub>
3. Lo switch S<sub>3</sub> vede il MAC address di I e capisce che il link di uscita su cui instradare è proprio quello che porta ad I



Seguendo lo stesso ragionamento, si può intuire che la Switch Table è popolato seguendo gli stessi meccanismi di quando lo switch era solo uno. I record continuano a riferirsi ai MAC address dei nodi, con la differenza che ora l'interfaccia di uscita può portare a uno switch che "fa avvicinare il frame alla sua destinazione finale".

La struttura gerarchica sembra dunque aumentare potenzialmente illimitatamente il numero di nodi che posso connettere in una rete LAN, sfruttando i **Tier - 1 Switch (quelli di livello più basso)** per separare in maniera "logica" gruppi di Nodi appartenenti alla stessa sub - organizzazione. Facciamo però alcune considerazioni :

- Il guasto del collegamento a uno switch di livello più alto impedisce la trasmissione con un grosso numero di nodi, che aumenta esponenzialmente nel numero del livello di switch che non posso più raggiungere. Quindi *non è sempre una buona idea* aumentare indefinitivamente il numero dei livelli degli switch. Per risolvere questo problema si può pensare a una struttura a **maglia**, in cui gli switch possono essere raggiunti con più percorsi.
- Una comunicazione di tipo **broadcast** raggiunge l'intera LAN. Questo è in generale un errore, in quanto le persone della stessa sub - organizzazione vorrebbero utilizzare una comunicazione di tipo broadcast che però si ferma alla loro sub - organizzazione. Un **singolo dominio di broadcast**, infatti, porta problemi di sicurezza e privacy.



## Rotta verso le VLAN (Virtual Local Area Network)

Precedentemente abbiamo chiarito che allo switch di livello più basso ci sono persone appartenenti a una certa organizzazione. Per queste persone, che vorrebbero vivere nel loro gruppo ristretto, esiste già il problema del singolo dominio di broadcast. Il problema del broadcast si potrebbe risolvere ponendo come switch di livello superiore un **router**, che garantisce l'isolamento del traffico. Vedremo che le VLAN risolvono comunque questo problema utilizzando solo Link - Layer Switches. In realtà ci sono anche altre due problematiche :

- Nell'approccio gerarchico con switch a più livelli, diverse porte Switch vengono sprecate. Infatti una divisione logica sempre più specifica porta ad avere gruppi sempre più piccoli, con switch di livello 1 dedicati che avranno poche interfacce utilizzate. Inoltre, alla nascita di un gruppo dovrebbe essere sempre comprato un nuovo First – Level Switch!
- **Un user non può muoversi nella zona dell'altro gruppo facilmente.** In particolare è impossibile che un utente cambi gruppo e riesca comunque ad accedere allo switch di livello 1 della sua precedente sub – organizzazione, a meno che non venga cambiato il cablaggio della situazione. In condizioni normali, se si va “fisicamente” nell’altro gruppo, si sarà connessi allo switch di quell’altro gruppo, e questo crea problemi alle persone del nuovo gruppo, che vedono il nuovo tipo come un estraneo!

Per questi motivi nascono le **VLAN**, un meccanismo per utilizzare switch in cui ogni porta di ingresso si trova assegnata a un certo gruppo di Nodi.

In una VLAN, ogni porta dello switch è assegnata a un certo gruppo di nodi. È compito degli amministratori configurare le porte in modo tale che i nodi appartenenti a una VLAN comunichino solo e soltanto con porte dello stesso tipo.

- È immediato il fatto che non si sprecano porte dello Switch, in quanto ogni gruppo può far riferimento allo stesso switch
- È garantito che ci sia un **diverso dominio di broadcast per ogni VLAN**, in quanto ogni nodo può comunicare solo con nodi connessi a porte della stessa VLAN
- È possibile cambiare facilmente gruppo modificando la configurazione della porta alla quale sono collegati!

Con le VLAN creo *virtualmente* diversi switch, anche se in realtà ne sto usando solo uno!

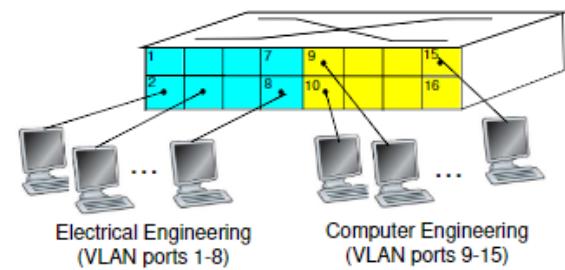
C'è però un problema, **come fanno organizzazioni diverse a comunicare?** Il problema è risolto introducendo insieme allo switch un router, che permette lo scambio di messaggi tra switch diversi.

Essendo la VLAN la virtualizzazione di più switch diversi, un router farà il suo dovere anche se collegato a un solo switch.

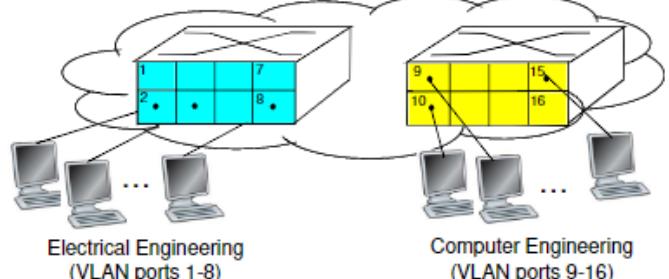
È per questo motivo che, nella pratica, il dispositivo che ci viene venduto è **switch e router insieme**, in modo tale che sia possibile configurare delle VLAN all'interno del nostro dispositivo!

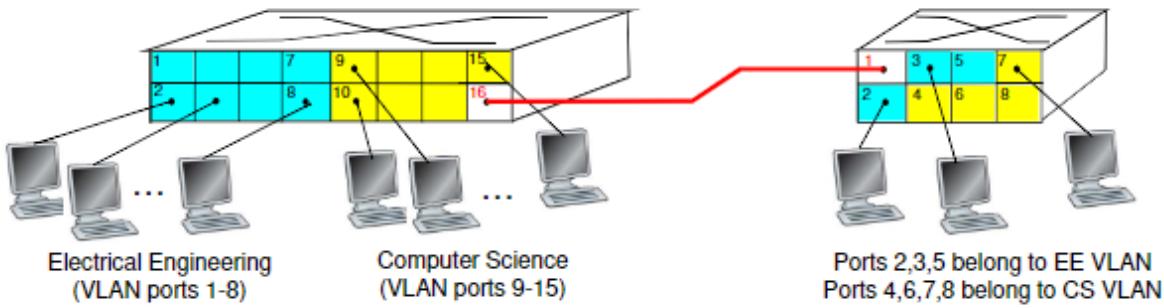
*Ma se una persona dell'organizzazione è fisicamente distante dallo switch che ospita le diverse VLAN?*

  
**Port-based VLAN: switch ports grouped (by switch management software) so that *single* physical switch .....**



*... operates as *multiple* virtual switches*





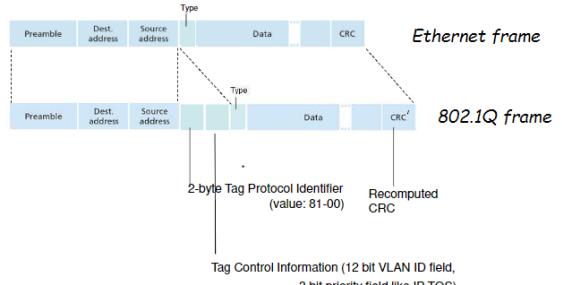
Anche questo non è un problema, **basta collegare due switch insieme**, utilizzando due porte (una su uno switch e una sull'altro). Questo tipo di meccanismo è detto **VLAN Trunking** (perché sfrutta appunto porte "trunk", o porte "ponte").

Le due porte trunk devono essere utilizzate indifferentemente da tutte le VLAN. Per questo motivo abbiamo introdotto un nuovo problema: una volta "atterrati" sull'altro switch, come si fa a capire a quale VLAN appartiene il messaggio?

Ci viene in aiuto una modifica nell'**Ethernet Frame**.

Introducendo infatti dei byte che identificano la VLAN, si possono modificare i protocolli in modo tale che :

- La trunk port di invio incapsuli aggiungendo questo campo
- La trunk port di ricezione decapsuli rimuovendo queste informazioni e **leggendole**, così da reindirizzare il traffico sulla VLAN corretta



## Trasferimento Asincrono : le reti ATM

Le reti ATM (**Asynchronous Transfer Mode**) nascono per garantire servizi circa la trasmissione di traffico audio e voce. Solo successivamente vengono usate anche per il trasferimento dati.

- Si impongono nel mercato come le reti in grado di incontrare dei requisiti minimi di timing, in contrasto al QoS offerto da Internet (best - effort)
- Si presentano come l'evoluzione delle reti telefoniche, in quanto sono **packet switching** su pacchetti di lunghezza fissa, detti **Celle**, che si spostano su un circuito virtuale (**Virtual Circuit** VC)

### Caratteristiche del Circuito Virtuale VC

Il circuito Virtuale riprende i servizi che erano offerti inizialmente dal circuito fisico che si stabiliva in ambito telefonico, e coinvolge azioni che devono essere intraprese nel Core Network.

- C'è bisogno di *instaurare una connessione e di chiuderla* (setup & teardown). Il setup va fatto **prima che i dati possano fluire**
- È necessario che lungo il path mittente - destinatario i dispositivi del core mantengano delle informazioni di stato del circuito
- È possibile che vengano allocati link o risorse sui dispositivi del core (buffer, banda) per quel circuito. In generale, se alcune risorse sono preallocata, si possono fare stime piuttosto accurate su alcuni requisiti minimi che saranno soddisfatti

Un circuito virtuale è implementato con tre meccanismi :

1. Si sceglie un **path** che tutti i pacchetti appartenenti al circuito virtuale devono seguire

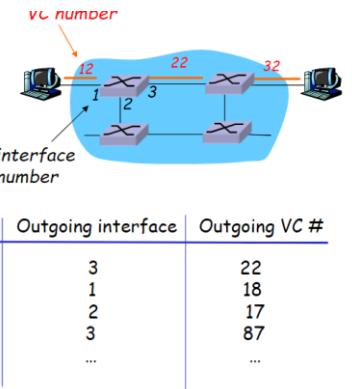
2. Si scelgono dei VC Number su ogni link attraversato nel percorso (i VC number possono cambiare da link a link) e le celle ATM contengono **il VC number, non il destination address**
3. Le tabelle di forwarding dei dispositivi del core devono contenere quadruple (*Interfaccia di Ingresso, VC number di ingresso | Interfaccia di Uscita | Nuovo VC Number*). Mantengono dunque **più informazioni di stato di un classico switch!**

Analizzeremo di nuovo il discorso più in là nel corso.

Forwarding table in A switch

Incoming interface	Incoming VC #	Outgoing interface	Outgoing VC #
1	12	3	22
2	63	1	18
3	7	2	17
1	97	3	87
...	...	...	...

Switches maintain connection state information!



## CAPITOLO 5 : LE INTER-RETI, IL NETWORK LAYER

Applicheremo nello studio delle “reti di reti” (il network layer) una separazione di concetti : il **Data Plane** e il **Control Plane** : il primo riferito a come un datagram viene instradato da un link di ingresso a un link di uscita di un router (forwarding), e il secondo riferito al percorso che i vari datagram devono percorrere end – to – end (routing).

Si ricordi che l’obiettivo finale è permettere la comunicazione tra più reti diverse tra di loro, che sfruttano **diverse tecnologie** e diversi protocolli.

### Introduzione al Layer : Terminologia e Servizi Offerti

*Nota Iniziale :* Internet (con la I maiuscola) è solo una possibile implementazione di una internet (che, con la i minuscola, sta per “inter-rete”, ovvero un sistema che riesce a collegare reti di tipo diverso).

Quando si parla di Network – Layer, ci sono due termini principali che bisogna tenere a mente, e che spesso sono trattati come se fossero la stessa cosa : **Forwarding e Routing**.

- Con il termine **Forwarding** intendiamo il meccanismo secondo il quale il router riceve un datagram da una delle sue interfacce di ingresso, e lo **instrada** (“fa Forward”) verso un’interfaccia di uscita. Il forwarding è raggiunto mediante la consultazione di una **Tabella di Forwarding**, grazie alla quale il router sceglie sempre un link di uscita che “avvicina” il datagram alla sua destinazione finale. In questo senso quello che fa un router è dunque analogo a quello che fa uno switch, però non c’è il self – learn, vedremo che esistono più modi per riempire la tabella, e tutti sono fatti “da umani”. L’implementazione del Forwarding in senso stretto, inteso come **instradamento da interfaccia di ingresso a interfaccia di uscita**, è solitamente gestita via Hardware
- Con il termine **Routing** intendiamo il meccanismo utilizzato per determinare il percorso intero che il datagram deve fare partendo dall’Host mittente all’Host destinatario. In particolare il calcolo di questo percorso avviene spesso via Software, mediante gli **algoritmi di Routing**. Questi algoritmi sono in grado di scegliere il percorso “più breve” (vedremo che il termine “breve” è dinamico nella scelta del criterio utilizzato) che il datagram deve fare per giungere a destinazione. Una volta che l’algoritmo è riuscito a determinare il percorso più breve, questo deve essere messo in testa ai router. Ciò significa che il meccanismo di routing è completato solo quando **si è riempita la Tabella di Forwarding dei router coinvolti**. Dato il percorso, inserisco i record giusti nei router giusti, affinché poi il forwarding in ogni router realizzzi il percorso “progettato”.

*Quali servizi riesce a fornire Internet combinando Forwarding e Routing?*

Chiediamoci cosa potrebbe fare :

1. **Consegna Garantita** : si assicura che i datagram giungano a destinazione

2. **Consegna Garantita con Ritardo Limitata** : si assicura che arrivino a destinazione con un ritardo massimo sperimentato lungo il percorso
3. **Consegna In Ordine** : si assicura che i datagram arrivino nello stesso ordine in cui sono stati mandati
4. **Minima Banda Garantita** : si assicura che i pacchetti viaggino “virtualmente” in un link point – to – point con un certo rate
5. **Sicurezza** : si assicura che i pacchetti siano criptati al sending – side e decriptati al receiver – side.

Di questi servizi, nessuno è assicurato da Internet. L'unico servizio che offre è il **Best Effort**, ovvero “fa del suo meglio, e devi pure accettarlo”. Anche se “Best Effort” sembra quasi un servizio vero, è l'equivalente elegante di dire “non faccio nessun servizio”.

Il servizio è di tipo **connectionless**, nel senso che ogni datagram viaggia individualmente dal resto della sequenza di datagram coinvolti nella singola trasmissione (ciò significa che i datagram diretti allo stesso host dallo stesso host possono fare strade molto diverse tra di loro).

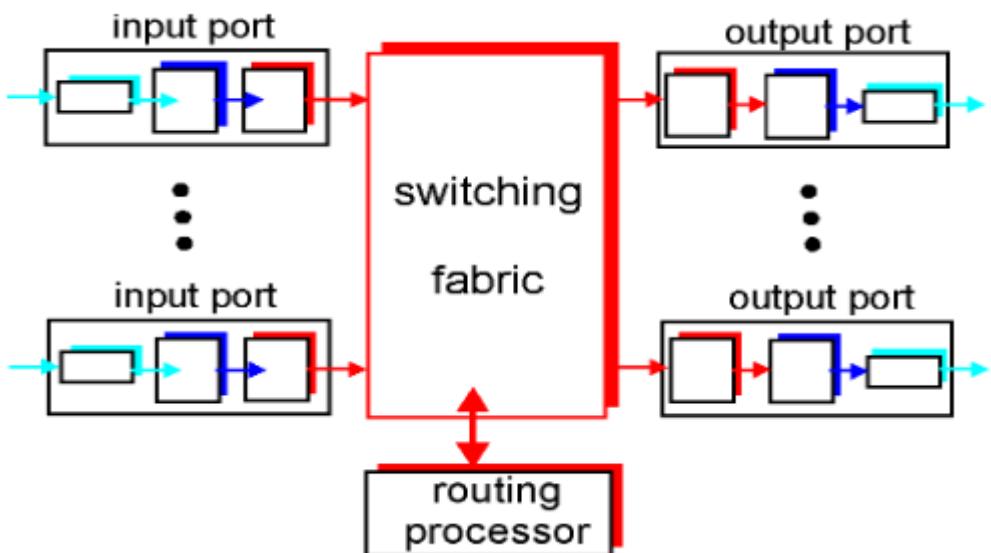
Nonostante Internet sia dunque così “scadente” rispetto ad altri tipi di implementazione che invece garantivano un certo numero di servizi (vedi ATM), ha preso comunque piedi ed è ancora oggi il modo di implementare inter – reti utilizzato!

Il motivo per cui Internet e ATM sono così diversi è dovuto a due fattori :

- **Il motivo per cui sono nati** : ATM nasce per trasmissione voce (e in seguito video), mentre Internet nasce principalmente per permettere il trasferimento di dati.
- **Gli End – System a cui si riferiscono** : ATM nasce con l'intento di servire degli End – System “stupidi”, come i vecchi telefoni. Per questo motivo implementa tutta la complessità all'interno del Core Network. Viceversa, Internet collega computer che in generale sono piuttosto intelligenti (e in grado di implementare da soli un servizio affidabile grazie ad ACK/Retrasmit). È per questo che nell'approccio di Internet si tiene tutta la complessità alla frontiera, mantenendo invece un Core Network molto semplice.

## Com'è fatto un Router?

Un router è composto principalmente da 4 componenti :

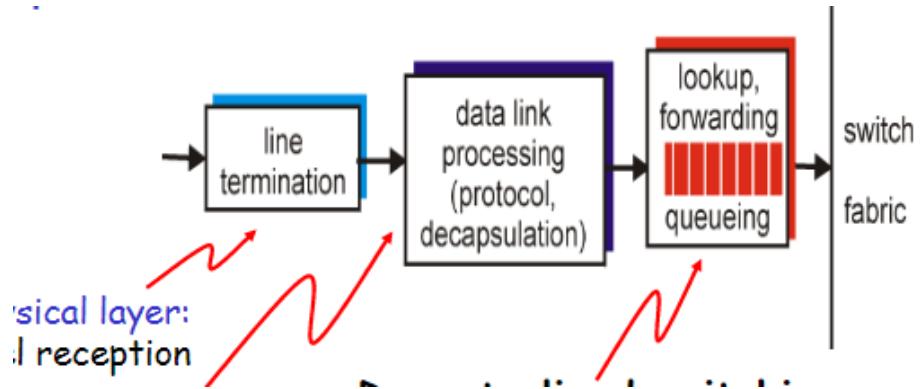
1. Le **Input Port**, dove vanno performate le azioni di livello fisico e Data – Link, e dove fa effettuata la “scelta di dove spedire il datagram”
  2. Le **Output Port**, dove si spedisce il datagram e viene *incapsulato* per poter permettere al prossimo router / switch di poter utilizzare le informazioni di livello Data – Link. Se un link è full duplex input port e output port sono realizzate in “parallelo” sulla stessa linea.
- 
- The diagram illustrates the internal structure of a router. It features two input ports at the top, each with three rectangular components and a red arrow pointing to a central 'switching fabric'. Below the input ports are two sets of three dots, indicating multiple input and output paths. To the right of the switching fabric is an output port with three rectangular components and a red arrow pointing away. Below the output port are two sets of three dots. A red arrow points from the bottom of the switching fabric to a 'routing processor' box at the bottom. Another red arrow points from the top of the routing processor back up to the switching fabric.

3. Una **switching Fabric**, dove avviene l'instradamento da una input a una output port
4. Un **processore** che deve eseguire gli algoritmi di routing e compilare la tabella di forwarding

## Il Viaggio del Datagram : Le Porte di Ingresso

La porta di ingresso di un router è divisa in 3 parti:

1. Una **Terminazione** dove viene recepito il singolo bit trasmesso a livello fisico
2. Un blocco dedicato alle operazioni di **livello Data - Link**. In particolare quello che arriva al router è un frame, che deve essere decapsulato per restituire il datagram oppure scartato se fatta Error Detection
3. Un blocco dedicato alla funzione di **Forwarding** del network - layer. In particolare è qui che si fa *lookup* (si guarda) della tabella di forwarding, ed è sempre in questo blocco che il processore compone la tabella stessa, così che l'instradamento possa avvenire localmente su ogni porta di ingresso.



Una volta che la porta di ingresso ha capito qual è il link di uscita, deve mandare il datagram nella switch fabric. Serve una gestione di **coda** perché la fabric potrebbe essere occupata da altri datagram inviati da altre porte di ingresso, o perché il forwarding potrebbe essere stato più lento dell'invio di un altro datagram sulla stessa porta.

Proprio per necessità di velocità, le funzioni di forwarding devono essere implementate in hardware. Questo perché su un link di, ad esempio, 10Gbps, un datagram di 64 byte (512 bit) deve essere processato in al più 51.2 ns prima che arrivi il prossimo datagram delle stesse dimensioni!

*Come si può rendere efficiente la funzione di Forwarding?*

Anche se sappiamo che in linea teorica "basterebbe" fare un record per ogni possibile indirizzo di destinazione, è totalmente fuori questione. L'indirizzo IP è su 32 bit, e dunque non è possibile fare 4G record!

La soluzione che si adotta è basata su **Range**.

Si possono instradare sulla stessa interfaccia di uscita tutti gli IP che hanno lo stesso prefisso.

In questo modo si **riduce drasticamente** il numero di entrate necessarie.

Se un indirizzo IP fa match su più prefissi, si utilizza una politica secondo la quale l'interfaccia di uscita scelta è quella con il prefisso più lungo matchato (**longest prefix matching rule**).

<i>Destination Address Range</i>	<i>Link Interface</i>
<i>11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111</i>	0
<i>11001000 00010111 00011000 00000000 through 11001000 00010111 00011100 11111111</i>	
<i>11001000 00010111 00011001 00000000 through 11001000 00010111 00011101 11111111</i>	1
<i>11001000 00010111 00011100 00000000 through 11001000 00010111 00011111 11111111</i>	
<i>otherwise</i>	3

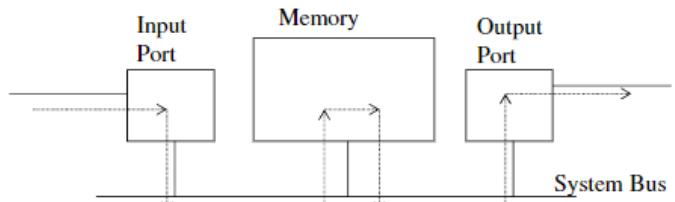
## Il Viaggio del Datagram : La switching Fabric

Il ruolo della switching Fabric è implementare una logica secondo la quale si trasmette un pacchetto da una porta di ingresso a una di uscita.

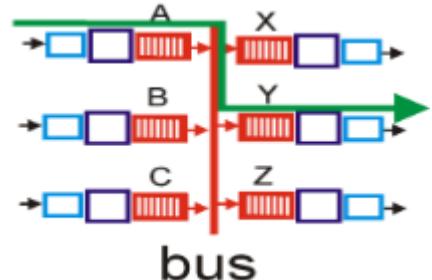
Le possibili implementazioni sono racchudibili in tre categorie:

1. **Switching secondo Memoria** : tempo fa le input port comunicavano direttamente con la CPU del router, e questa era incaricata di copiare il pacchetto dalla input port alla memoria, riempire la tabella di forwarding, consultarla, e prelevare il datagram dalla memoria e copiarlo nel buffer della giusta output port. Questa soluzione

implementa un overhead significativo, e nei router moderni è stata resa più efficiente facendo in modo che sia la input port a consultare la tabella di forwarding e a copiare il datagram nella giusta porzione della memoria. In questo modo ciò che resta da fare è un trasferimento da precise porzioni della memoria a precise interfacce di uscita.

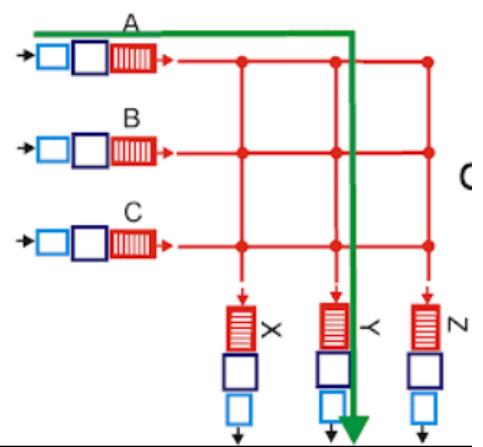


2. **Switching secondo Bus condiviso** : in questa soluzione (molto più efficiente della precedente) le porte di ingresso tentano di spedire immediatamente il loro datagram su un bus condiviso. L'unica regola da rispettare è che sul bus viaggi un solo datagram per volta. Un'altra cosa che è necessario che le input port facciano è inserire una **breve intestazione nel datagram** dove specificano a quale output port è destinato. In questo modo il bus condiviso fa in modo che il datagram arrivi a tutte le output port, ma che venga bufferizzato solo da quella che trova l'intestazione che si riferisce a lei (ovviamente è compito della output port rimuovere questa informazione prima di proseguire).



3. **Switching secondo Rete di Bus** : secondo questo tipo di switching è possibile interconnettere  $2N$  bus per connettere  $N$  porte di ingresso ad  $N$  porte di uscita. I bus orizzontali si incrociano coi bus verticali, creando dei punti di intersazion, che un controllore può decidere di chiudere impedendone l'attraversamento.

Grazie a questa soluzione è possibile spedire in parallelo più datagram da certe porte di ingresso a certe porte di uscita. Se ad esempio A sta spedendo a Y, nulla vieta il trasferimento B - X (o C - X). Ciò che importa è che "il percorso delimitato dai punti di intersezione" sia percorso da un datagram alla volta. Se ad esempio C vuole spedire ad Y mentre lo sta già facendo A, dovrà aspettare al crosspoint che la parte di bus crosspoint - Y si liberi. Una politica così fatta è detta **non bloccante**, ovvero il datagram non è mai bloccato se non quando un altro datagram tenta di raggiungere la stessa interfaccia di uscita!



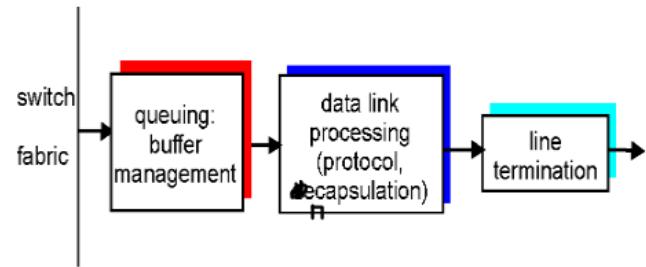
## Il viaggio del Datagram : le porte di Uscita

Le porte di uscita devono, come le porte di ingresso, effettuare operazioni di Data Link (incapsulare in frame) e di livello fisico.

A differenza delle porte di ingresso sono meno complesse, in quanto non devono consultare la tabella di forwarding o copiare datagram in memoria (qualora la switching fabric sia costituita da una memoria).

Come le porte di ingresso, però, si possono trovare a gestire una **coda**, in quanto potrebbero arrivare datagram più velocemente di quanto la porta riesca a spedirne.

In conclusione, il fenomeno del *packet loss* può verificarsi sia in ingresso che in uscita a un router!



## Il protocollo IPv4

Ci sono ad oggi due versioni del protocollo IP che si sono affermate : **IPv4** e **IPv6**. Esaminiamo il primo dei due con grande dettaglio, e poi cerchiamo di capire cosa cambia nell'altro.

### IPv4 : Formato del Datagram

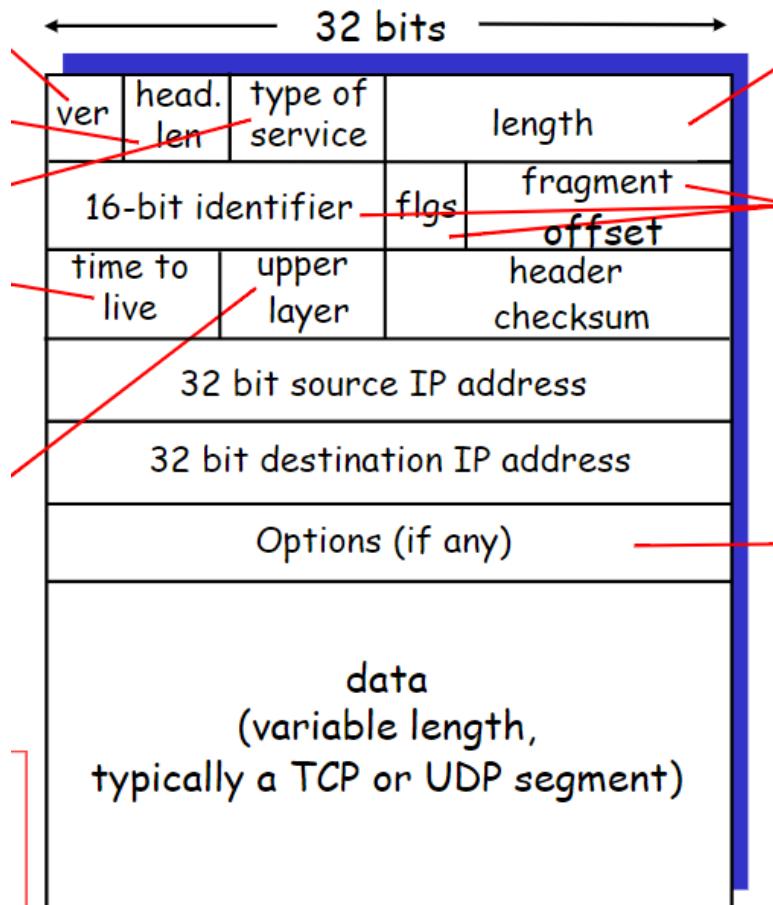
Il datagram ha un'intestazione composta generalmente di **20 byte** (cinque righe da 32 bit).

**Prima Riga :** Contiene 4 bit per esprimere la versione del protocollo IP che si sta utilizzando (**Version Number**), così che il router sappia come interpretare il resto dell'intestazione. In questo caso c'è dunque scritto "4" : 0100. Dopodiché segue un campo di 4 bit **Header Length**, che esprime la lunghezza (in parole da 32 bit) dell'Header. Questo può essere più grande di 20 byte quando si aggiungono delle *opzioni*, anche se di solito non ce ne sono. Seguono 8 bit per il **Type of Service**, dove si specifica il tipo di datagram (lasciar perdere). Seguono 16 bit dedicati al Datagram **length**, che esprime la grandezza dell'intero datagram (Header + Dati). Avere 16 bit significa che esiste un limite teorico di oltre 65000 Byte. Capiremo in seguito che i datagram sono *molto più piccoli*.

**Seconda Riga :** Lasciamola perdere per ora

**Terza Riga :** ha un intero byte dedicato al **TTL (Time To Live)**, dove viene conservato il numero massimo di Hop rimanenti oltre il quale il datagram sarà buttato via. C'è poi un altro byte dedicato al protocollo di livello superiore a cui la destinazione finale del datagram dovrà spedire lo stesso. Ci sono poi 16 bit di **Checksum** per fare detection *sulla sola intestazione*. In particolare ogni coppia di byte è interpretata come un numero. Si esegue la somma in complemento a 1, e si complementa il risultato (questo sarà il checksum). Se si fa error detection, normalmente si butta il datagram. Si consideri che non si fa error detection sui dati, in quanto non si deve dimenticare che Internet offre un servizio di tipo **best Effort**!

**Quarta e Quinta Riga :** contengono gli indirizzi IP del mittente e del destinatario del datagram



**Campo Data** : è il cuore del datagram, ed è normalmente un Segmento di livello trasporto (TCP o UDP), ma non si esclude sia un messaggio di livello network (come quelli che si scambiano in fase di routing).

## IPv4 : Necessità di Frammentare un Datagram, la MTU!

I Datagram non possono essere lunghi a piacere. Questo non è un limite espresso dal network – layer, ma piuttosto un limite dei protocolli di livello Data Link. Ogni protocollo, ogni tecnologia, richiede che il frame abbia un payload di dimensione massima. Chiamiamo questa dimensione massima **MTU (Maximum Transmission Unit)**.

Se tutti a livello Data Link avvessero lo stesso MTU, si potrebbe semplicemente fare in modo che anche i router rispettino un MTU. Il problema è che il network – layer serve a connettere reti di tipo diverso, che possono differire proprio nella dimensione del payload (e dunque nell'MTU).

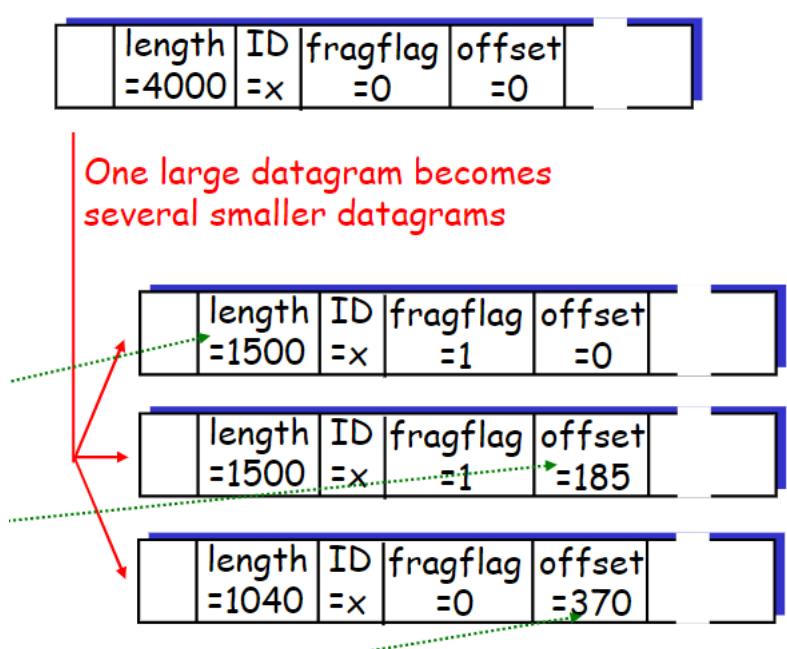
Il router si può trovare dunque con un datagram (che è il payload del link – layer) **che supera in dimensione l'MTU**. Per risolvere questo problema, i designer di IPv4 hanno deciso che la complessità del problema si spostasse a livello End - System. Il protocollo prevede la possibilità di dividere un datagram in datagram più piccoli, detti **frammenti**. La ricomposizione in ordine dei frammenti avviene tuttavia a livello più alto, e sono stati pensati dei campi di intestazione che possano aiutare a ricomporre i frammenti. Questi campi costituiscono la seconda riga di ogni header IP:

1. **Identification Number** : ogni volta che il sending Host spedisce un datagram “diverso”, incrementa questo numero. In questo modo ci si aspetta che l’Host ricevente sia in grado di capire quali sono i frammenti dello stesso datagram : quelli che hanno identification number uguale!
2. **FragFlag** : con la frammentazione si introduce il problema del “non so se questo datagram è finito oppure mancano dei frammenti!” Per risolvere questo problema si introduce un flag che vale 0 se siamo all’ultimo frammento, 1 altrimenti.
3. **Fragment Offset** : questo campo garantisce che i frammenti siano riordinati in ordine. In particolare esprimono l’offset, 8 byte alla volta , in cui il campo dati di quel frammento trova posto nel campo dati del datagram originale.

Facciamo un esempio per capire meglio

Supponiamo si voglia mandare un datagram di 4000 byte (20 byte di intestazione, 3980 di dati), e che però si incontri un MTU di 1500 byte. Sono costretto a spezzare il datagram in tre frammenti :

- **Il primo frammento** sarà lungo 1500 byte e avrà fragflag = 1 (è il primo, non l’ultimo) e offset = 0 (appunto, è il primo). I 1480 byte di dati, se raggruppati in righe da 8 byte, raggiungono l’offset 185
- **Il secondo frammento** sarà lungo anche lui 1500 byte e trasporterà 1480 dati utili. Ciò significa che “incollato questo frammento” nel datagram ci saranno 2960/3980 byte di dati ricomposti. L’offset è 185, per i motivi



detti sopra.

- Il terzo e ultimo frammento ha appunto fragflag = 0, e ha un offset = 370 perché  $2960/8 = 370$ , ovvero il campo dati di questo frammento va incollato 370 byte  $\times 8$  dopo l'inizio del campo dati del datagram originale. La lunghezza è 1040, 20 della sua intestazione e 1020 di dati utili. Notiamo che  $2960 + 1020 = 3980$ , i conti tornano!

Si è scelto di dividere per 8 ( $2^3$ ) nel calcolo dell'offset per ovviare al problema che i bit dedicati all'offset sono solo 13 (e non 16 come il campo length!).

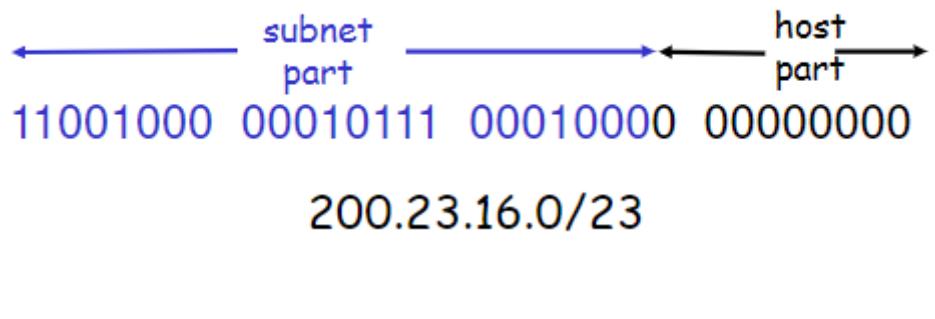
## Indirizzamento nel protocollo IPv4

Organizzazione "strutturata gerarchica" dell'indirizzo IP

Il compito di ogni indirizzo IP è quello di identificare **un'interfaccia**, ovvero il meccanismo che collega l'host al livello fisico. Generalmente un Host ha una sola interfaccia, e quindi si dice che "*l'indirizzo IP identifica l'host*" mentre un router ha un'interfaccia per ogni link a cui è collegato (un router avrà dunque sempre almeno due indirizzi IP).

Ogni indirizzo IP :

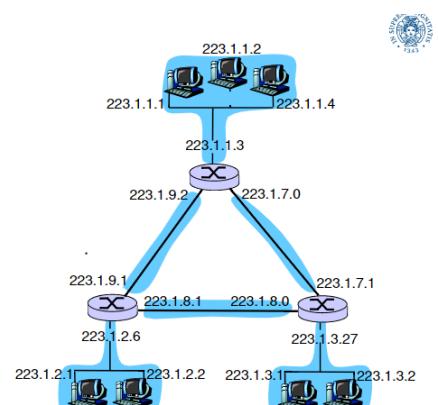
- Ha un **indirizzo di sottorete** identificabile dal gruppo di bit più significativi
- Ha l'**indirizzo dell'host in quella sottorete** identificabile dal gruppo di bit meno significativi.



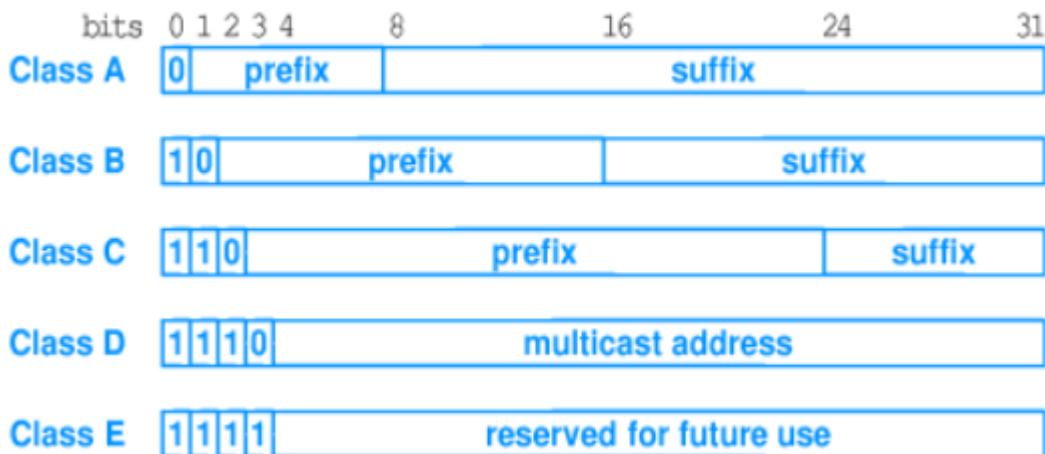
Una sottorete è un insieme di Host localmente collegati (ovvero che possono comunicare direttamente senza utilizzo di un router in mezzo).

Nella figura qui a destra esistono in particolare ben 6 sottoreti :

- 3 di queste sono evidenti (perché raggruppano un certo numero di Host e un'interfaccia del router).
- Anche se i collegamenti tra router sono punto punto, questi collegamenti costituiscono ognuna una sottorete, portando il totale a 6!



In generale quando due host hanno la stessa parte significativa è probabile che appartengano alla stessa sottorete. Ma cerchiamo di capire meglio come funziona. Ci sono due approcci possibili, il primo è quello basato su **Classi di Indirizzi**.



- **Indirizzi di Classe A :** sono indirizzi che cominciano tutti per 0 e che hanno 8 bit dedicati all'indirizzo di sottorete, e 24 dedicati agli Host. Questi indirizzi costituiscono dunque pochi sottoreti con un enorme numero di Host.
- **Indirizzi di Classe B :** sono indirizzi che cominciano tutti per 10 e hanno 16 bit di sottorete e 16 di host. Sembrano sottoreti equilibrate.
- **Indirizzi di Classe C :** sono gli indirizzi che cominciano per 110 e hanno 24 bit per la sottorete e solo 8 per gli host (massimo 256 host). Sono dunque tante sottoreti con un piccolo numero di Host.
- Gli indirizzi di classe D servono per la comunicazione multicast (non ho idea di cosa sia)
- Gli indirizzi di classe E erano pensati per essere riservati per un futuro prossimo.

C'è però un problema serio nell'organizzazione degli indirizzi in classi. Se una persona ha un numero di Host > 256, deve decidere se prendere più spazi di classe C o uno solo di classe B :

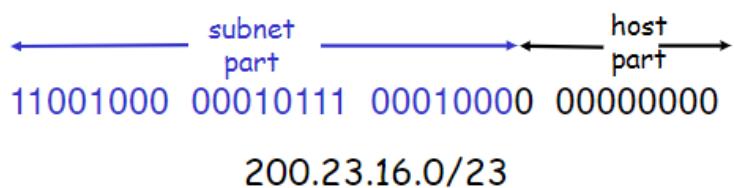
- Nel caso ne scegliesse tanti di classe C, avrebbe bisogno di un router per ogni sottorete presa
- Nel caso scegliesse un unico spazio di classe B spreca un numero enorme di indirizzi

Mettiamoci nel secondo caso, dove spreco un sacco di indirizzi. Se tutti ragionano così (perché conviene), allora gli spazi di classe B finiscono rapidamente!

Per questo motivo si abbandonò presto l'approccio basato su classi con l'aumentare delle piccole - medie organizzazioni che avevano bisogno di una sottorete.

Il nuovo approccio fu quello **CIDR : Classless InterDomain Routing**. Come funziona?

Si alloca un indirizzo di sottorete a lunghezza variabile. Dopodiché l'indirizzo IP si scrive nella forma A.B.C.D/X dove X rappresenta il numero di bit dedicati alla sottorete. In questo modo le piccole - medie organizzazioni possono rispondere ai loro bisogni di avere un numero di Host medio, e non si spreca più nulla (o quasi).



Prima di esaminare come questo tipo di pensare l'indirizzamento aiuta i router nelle loro funzioni di forwarding, fissiamo bene il fatto che ci sono alcuni **indirizzi riservati**.

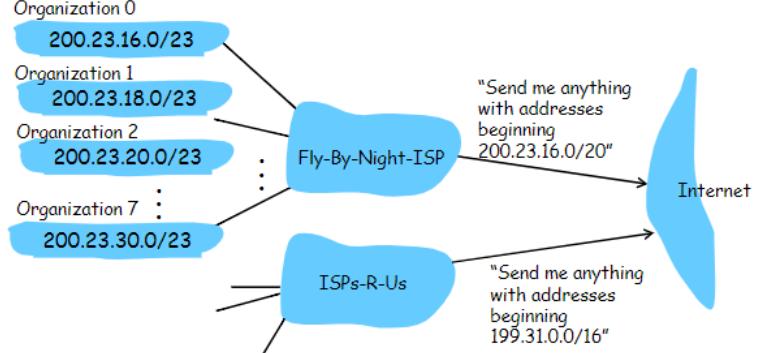
- L'indirizzo a tutti 0 è l'indirizzo che rappresenta un nodo che non ha indirizzo.
- Gli indirizzi che cominciano con l'indirizzo di sottorete e poi sono tutti 0 o tutti 1 rappresentano l'indirizzo della sottorete e l'indirizzo di broadcast della sottorete (ovvero l'indirizzo che un nodo, eventualmente esterno alla sottorete, inserisce per mandare in broadcast a tutti i nodi della sottorete, **broadcast orientato**).
- L'indirizzo a tutti 1 rappresenta **l'indirizzo di broadcast ristretto**, mediante il quale il datagram è spedito a tutti i nodi della rete locale a cui il nodo sorgente appartiene. È dunque utilizzato quando ad esempio il nodo non conosce l'indirizzo della sua sottorete, e dunque usa questo.
- L'indirizzo di sottorete 127 è riservato per fare **loopback** quando si sviluppano applicazioni distribuite. Quando a livello network si riconosce questo 127, il pacchetto è rimandato allo stesso Host (utile per simulare il funzionamento della propria applicazione).

Network Number	Host Number	Description	Notes
all 0s	all 0s	"this node"	Used at startup
x	All 0s	Network Address	Identify network x
x	all1s	Broadcast Address	datagram sent to all nodes of network x
all 1s	all1s	Restricted Broadcast Address	datagram sent to all nodes of the local network
127	--	Loopback Address	Used when developing applications

### Qual è il grosso vantaggio di organizzare una inter-rete in sottoreti?

Il motivo principale è snellire le tabelle di forwarding dei router che stanno nel core. Se infatti sottoreti fanno riferimento a sottoreti più grandi, come nell'esempio qui a fianco :

- I router dell'Internet interno conserveranno nelle loro tabelle di forwarding solo una entrata per tutti gli indirizzi IP dell'ISP (che nell'esempio gestisce 8 organizzazioni).
- È poi il router dell'ISP che dovrà contenere 8 record, **sposto la complessità esternamente**.



Un ISP può richiedere il suo indirizzo di sottorete all'ICANN (**Internet Corporation for Assigned Names and Numbers**), che gestisce le sottoreti e il sistema DNS (conosce i nomi di dominio disponibili).

Se un'organizzazione conosce il suo personale /X, può risalire all'indirizzo della sua sottorete guardando semplicemente il suo indirizzo IP, e conoscere anche la sottorete dell'ISP conoscendo lo /X dell'ISP (semplicemente shifta a sinistra l'osservazione del suo indirizzo IP).

ISP's block	11001000 00010111 00010000 00000000	200.23.16.0/20
Organization 0	11001000 00010111 00010000 00000000	200.23.16.0/23
Organization 1	11001000 00010111 00010010 00000000	200.23.18.0/23
Organization 2	11001000 00010111 00010100 00000000	200.23.20.0/23
...	.....	....
Organization 7	11001000 00010111 00011110 00000000	200.23.30.0/23

C'è però un problema più generale : come fa il singolo Host a ricevere il suo indirizzo IP?

## RICEVERE L'INDIRIZZO IP : PERMANENT O TEMPORARY ADDRESS

Una valida soluzione è di fissare **permanente** un indirizzo IP a un Host. Ovviamente per eseguire questo compito serve qualcuno che conosca quali indirizzi IP sono disponibili ad essere assegnati (è il ruolo, ad esempio, di un amministratore di sistema, non sono scelte libere dell'utente).

Nonostante fosse comunque un'opzione possibile, c'è lo svantaggio enorme che non ha senso che sia riservato un indirizzo IP anche quando un Host non è permanentemente 24/7 connesso in rete.

Per questo motivo nasce l'esigenza di "**ricevere un indirizzo IP al bisogno e restituirlo quando non si usa più**". Questo meccanismo (*temporary addressing*) potrebbe anche essere fatto a mano, ma ovviamente sarebbe meglio sfruttare un meccanismo "plug - and - play" che funzioni in maniera automatica.

Questo meccanismo sfrutta un'applicazione "**Client - Server**": il client effettua una richiesta "voglio un indirizzo IP!" e un server risponde. Ovviamente come tutte le applicazioni client - server c'è bisogno di un protocollo, e questo protocollo si chiama **DHCP : Dynamic Host Configuration Protocol**

### DHCP e NAT : "I router che implementano oltre il layer - 3"

Analizziamo anzitutto il protocollo DHCP. Questo permette agli host di ricevere dinamicamente un indirizzo IP. In particolare :

- È plug - and - play, le persone comuni non devono nemmeno conoscere l'esistenza di un DHCP Server all'interno della loro rete locale
- Permette una **migliore efficienza nell'utilizzo degli indirizzi**, in quanto questi vengono assegnati con un lifetime e non rinnovati a meno che richiesti nuovamente.
- È automatico, ovviamente è lavoro in meno per amministratori di rete che devono ora solo creare un "pool" di indirizzi da inserire all'interno del DHCP

Cerchiamo ora di capire come fa il server DHCP ad assegnare un **indirizzo IP temporaneo ad un Host**. Ipotizzeremo che nella subnet sia presente un server di questo tipo (altrimenti si dovrebbe demandare al router il compito di cercarne uno), e supponiamo di non sapere *quanti server DHCP ci sono*. Il sistema è complessivamente detto **a quadrupliche scambio di messaggi**, e si articola appunto in quattro fasi :

- 1) **Messaggio DHCP Discover** : il client che vuole ricevere un indirizzo IP cerca di raggiungere un DHCP server compilando un messaggio così fatto :

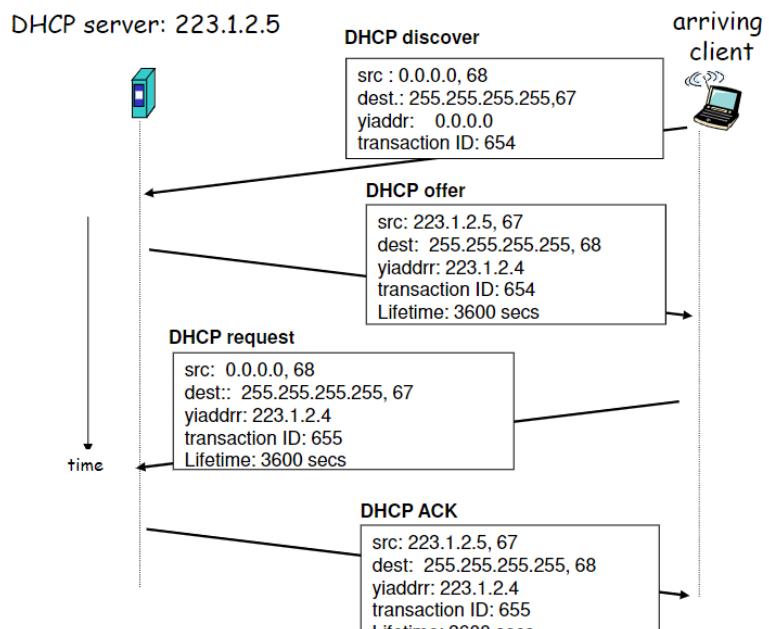
Source : Tutti 0, per rappresentare il fatto che è senza indirizzo IP

Destination : Tutti 1 per comunicare in broadcast con la rete locale a cui si è connesso

Yiaddr : Che sta per "*your internet address*" è il campo che sarà riempito con l'IP offerto

Transaction ID : Rappresenta l'ID della richiesta (ipotizziamo un numero casuale). Se un DHCP risponde con lo stesso transaction ID, il client capisce che sta parlando con lui

- 2) **Messaggio DHCP Offer** : il DHCP server che è in ascolto sulla porta 67 riceve la richiesta di aiuto e rimanda in broadcast



(perché appunto il sorgente non aveva indirizzo) un IP disponibile, rispondendo con lo stesso transaction ID così da far capire al client che si sta parlando con lui. Si introduce anche un lifetime con cui si vuole comunicare "avrai l'indirizzo IP per questo tempo"

- 3) **Messaggio DHCP Request**: il client comunica in broadcast che vuole accettare l'offerta dell'IP che specifica nel campo *yiaddr*. Il client potrebbe parlare direttamente col server che gli ha fatto l'offerta perché conosce il suo source address adesso, ma comunica in broadcast per **segnare agli altri DHCP server** che sta accettando l'offerta di qualcun altro.
- 4) **Messaggio DHCP ACK**: il DHCP client comunica la buona riuscita del protocollo. Se questo messaggio viene ricevuto correttamente, l'IP è consegnato e può essere utilizzato per un *lifetime*. Il client può comunque sfruttare il protocollo per **rinnovare lo stesso indirizzo IP** per altro tempo.

In realtà il server DHCP comunica insieme all'indirizzo IP anche altre cose : Maschera di Sottorete, il router di default e il servizio DNS primario e secondario (parleremo più avanti di queste componenti).

I server DHCP possono essere implementati direttamente nel router di frontiera. Questa è una strana contraddizione, perché se fosse così i router dovrebbero implementare anche il livello trasporto e il livello applicazione (UDP per trasporto, e protocollo DHCP a livello applicazione).

Questa è una eccezione, la visione funzionale di router resta comunque **fino al layer - 3** (non lasciatevi confondere dal titolo del paragrafo, che ho messo fuorviante apposta).

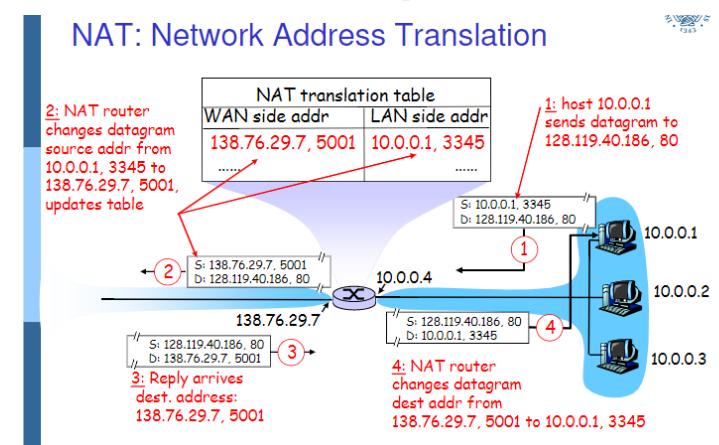
Parliamo ora del meccanismo **NAT – Network Address Translation**. L'idea di base è poter associare una multitudine di Host allo stesso indirizzo IP. Per far questo introduciamo anzitutto il concetto di **indirizzi privati**.

Ci sono certi indirizzi IP, come quelli del tipo 10.x.x.x o 192.x.x.x che sono dichiarati privati, ovvero possono essere utilizzati per comunicare nella propria rete locale. In particolare all'interno di una rete locale, posso utilizzare un indirizzo privato come destinatario anche senza avere un IP pubblico :

- Se il router vede un indirizzo non privato come destinatario, allora rigetta il datagram, dichiarandolo **non routable** perché ha un indirizzo sorgente privato e destinatario non privato.
- Se il router vede invece un IP privato instrada nella stessa rete locale verso quell'IP privato.

L'idea è quella di implementare un NAT all'interno del router di frontiera delle reti locali. Ipotizziamo che un client NATTATO (ovvero "coperto da un router NAT – Enabled") voglia comunicare con un server esterno :

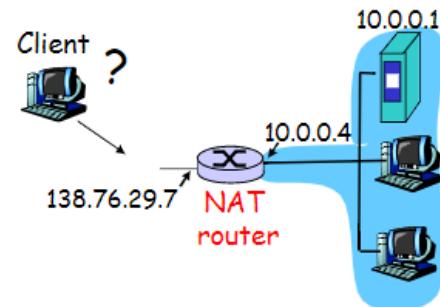
- 1) Il client inserisce nel datagram l'indirizzo destinatario del server, che è un IP pubblico. Specifica anche una **porta di ascolto**, e manda tutto al router.
- 2) Il router, che normalmente dichiarerebbe questo datagram not routable, fa agire il NAT : questo cambia l'indirizzo sorgente del datagram nell'**indirizzo IP del NAT stesso** (che è un IP pubblico), e mette un numero di porta eventualmente diverso. Memorizza la coppia (IP Pubblico NAT | Porta Corrispondente - IP Privato Host | Porta Client) all'interno di una tabella, detta **NAT translation table**.
- 3) Quando il server pubblico risponde, questo risponde inserendo come destinatario l'IP pubblico del NAT e la porta indicata dal NAT stesso.



- 4) Quando il NAT riceve qualcosa dall'internet esterno, consulta i record della sua tabella basandosi sulla porta sulla quale gli è arrivato il messaggio: quando trova la corrispondenza, sa a quale host privato instradare il datagram.

Il router di frontiera NAT - Enabled può ottenere il suo indirizzo IP mediante protocollo DHCP con l'internet esterno, e dopodiché può distribuire lui stesso gli indirizzi privati con lo stesso protocollo.

**Problema :** ma se fosse il Server ad essere Nattato, come fanno i client a riferirsi a lui? I client possono infatti conoscere l'indirizzo del Server come l'indirizzo del NAT, ma poi quando le richieste arrivano al NAT, lui non trova nessuna entry in tabella e non riesce a spedire nulla al server.  
Una semplice soluzione a questo problema consiste nell'inserire a priori manualmente dei record in tabella, così che il NAT possa redigere il traffico correttamente sul server che ha IP privato.



**Enorme Vantaggio :** posso "mascherare" dietro un solo indirizzo IP un numero di Host pari al massimo numero di porta possibile. Essendo il numero di porta su 16 bit, si parla in linea teorica di oltre 65 mila Host dietro un solo indirizzo IP!

**Critiche al meccanismo :** ne sono state fatte diverse. Prima fra tutte, si è contestato il fatto che il numero di porta non dovrebbe servire a identificare gli Host, ma i processi all'interno degli Host. Nel meccanismo NAT le porte servono invece a "identificare gli Host" dal punto di vista del NAT, che trova corrispondenze basandosi sui numeri di porta!

Un'altra critica fu il fatto che i Router che fanno girare DHCP e NAT vanno troppo oltre la loro pura visione funzionale di routing e forwarding, sembrano dei dispositivi di livello layer - 5 in questo modo! Per questo motivo, nonostante il NAT risolve il problema di "carenza di indirizzi", gli esperti credono che il problema deve invece essere risolto da una "maggiore disponibilità di indirizzi in generale". Questo sarà risolto con **IPv6**, che fornirà uno spazio di indirizzamento di ben 128 bit (ne parliamo dopo).

## Il Forwarding dei Datagram : I Parametri di Configurazione

Come detto sopra, il protocollo DHCP consegna, oltre all'indirizzo IP, anche altri parametri. Oltre all'indirizzo del DNS, che sappiamo a cosa serve, ci sono altri due parametri :

- **Maschera di Sottorete :** è una stringa di 32 bit con tutti i bit a 1 nella parte dedicata alla sottorete, e tutti i bit a 0 nella parte dedicata agli Host. Ad esempio, se l'indirizzo IP è a.b.c.d/**20**, allora la maschera di sottorete sarà 255.255.240.0. Poiché la maschera di sottorete (come vedremo) è qualcosa che serve più al router che a noi, è comodo anche esprimere per intero ( 11111111 11111111 11110000 00000000 ).
- **Indirizzo del Gateway di Default :** è l'indirizzo del router di default. In effetti questo parametro serve perché il primo forwarding lo fa proprio l'Host. Questo deve infatti decidere se spedire in locale il proprio pacchetto o se spedirlo alla propria rete locale.

Vediamo come possiamo incastrare queste informazioni nel forwarding del router e dell'Host iniziale.

### Router

Il router consulta (al limite) tutte le sue entrate nella tabella di Forwarding. Per ogni entrata ha coppie (indirizzo di sottorete | interfaccia di uscita), e altre informazioni come la maschera di sottorete relative alla sottorete di riferimento e l'identificativo del router verso cui si va seguendo quell'interfaccia.

Il compito del router è prendere **l'indirizzo IP del destinatario**, e ricavarne l'indirizzo di sottorete. Se poi questo indirizzo fa matching con una delle sue entry, instrada sulla coerente interfaccia di uscita. A questo punto l'unica cosa da capire è come, dato un indirizzo IP, si può ottenere il suo indirizzo di rete, o meglio come si può capire se il suo indirizzo di sottorete coincide con qualcun altro.

SubnetNumber	SubnetMask	NextHop	Interface
128.96.34.0	255.255.255.128	Router R1	interface 0
128.96.34.128	255.255.255.128	Router R3	interface 1
128.96.33.0	255.255.255.0	Router R3	interface 1
...	...	...	...

```
DHost=Destination IP Address
For each entry [i] in Table {
    DNet=(SubnetMask[i] & Dhost)
    If(DNet==SubnetNumber[i])
        then deliver datagram to NextHop[i] through Interface[i]
}
```

La soluzione è un **and logico con la maschera di sottorete**. La presenza di tutti i bit a 0 nella parte di Host farà in modo che l'AND resetti tutti i bit dedicati agli Host, restituendo appunto l'indirizzo di sottorete del destinatario. Se questo indirizzo restituito non coincide con nessuna entry (si fa l'and con la i-esima maschera ogni volta!), allora il datagram si butta, altrimenti si instrada coerentemente con l'interfaccia indicata dal record in cui c'è stato matching.

### Sending Host

Il sending Host non ha propriamente una tabella di forwarding, ma conosce la sua maschera di sottorete e il default gateway, perché sono parametri che gli sono stati dati dal DHCP server.

Quando vuole inviare qualcosa, dunque, si prende l'IP del destinatario e se ne fa l'AND logico con la maschera di sottorete del sending Host. Se il risultato coincide con l'indirizzo di sottorete dell'host stesso, allora significa che il destinatario sta nella sua stessa rete.

```
SubnetNum=MySubnetMask & Dest_IP_Addr
If(SubnetNum == MySubnetNum)
    then deliver datagram to Dest_IP_Addr directly
else forward datagram to default router
```

Se il risultato è diverso, allora significa che appartiene all'“Internet Esterno”, e pertanto si **spedisce al Default Gateway**.

## I router che si scambiano messaggi : Il protocollo ICMP

Il protocollo ICMP (**Internet Control Message Protocol**) nasce per permettere ai router di scambiarsi principalmente messaggi di errore. Il protocollo è di livello **superiore** all'IP, in quanto i messaggi di questo tipo sono incapsulati nel payload del datagram. Se nel campo *Upper - Layer* risulta ICMP, allora il contenuto viene interpretato come messaggio ICMP.

Ogni messaggio ICMP è strutturato in tre parti : **Tipo, Codice**, e i l'intestazione + i primi 8 byte del datagram che sta causando l'errore. Quest'ultima aggiunta serve a chi riceve il messaggio ICMP a riconoscere il datagram che ha causato problemi, o comunque il datagram a cui si sta riferendo.

Type	Code	description
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	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

Per quanto riguardo Tipo e Codice, sono l'equivalente di <Classe, Sottoclasse> di messaggio. Ad esempio troviamo nel tipo 3 diversi codici riferiti a "cose non trovate" (Port unreachable, Host unreachable, ...).

Ci sono due applicazioni di uso comune implementate mediante questo protocollo.

La prima di queste è il **PING**, utilizzato per calcolare un RTT da un host a un altro (utilizzato per capire se i server sono svegli e, qualora lo fossero, quanto tempo ci vuole a raggiungerlo). In questo caso :

- L'Host mittente invia un messaggio ICMP con <Tipo, Codice> (8, 0), a rappresentare una Echo Request. Incapsula questo messaggio in un datagram IP e lo spedisce. *Parte un Timer*.
- L'Host destinatario riceve il messaggio ICMP (vede *Upper Layer : ICMP*) e risponde con un altro messaggio ICMP con <Tipo, Codice>, (0, 0), a rappresentare una Echo Reply. Quando il mittente riceve questo messaggio *interrompe il Timer*, ha calcolato l'RTT!

L'altro meccanismo è quello del **TRACEROUTE**, utilizzato per calcolare una serie di router attraversati (e per ognuno indicando l'RTT) da un Host che vuole raggiungerne un altro. In questo caso :

- L'Host mittente prepara una serie di pacchetti ICMP con **TTL Crescente**. Il primo con TTL = 1, il secondo con TTL = 2, e così via. I pacchetti sono spediti in maniera *UDP con una porta Improbabile* (si vuole che nessuno sia in ascolto su quella porta).
- Quando il primo router riceve il primo pacchetto (quello con TTL = 1) decremente TTL e si accorge che si è azzerato. Il router allora spedisce un messaggio ICMP di <Tipo, Codice> (11, 0), *TTL Expired* e spedisce anche il suo indirizzo IP. In questo modo il mittente è riuscito a capire qual è il primo router che il pacchetto ha attraversato
- Quando il secondo router riceve il secondo pacchetto (quello con TTL = 2) si accorge che TTL si è azzerato e reagisce come sopra. Scopre così le indicazioni su tutti i router!
- Il traceroute si ferma quando l'ultimo router, che collega alla destinazione finale, trova che non esiste la porta specificata e spedisce un ultimo messaggio ICMP di <Tipo, Codice> (3, 3), *Destination Port Unreachable*.

```
Microsoft Windows 2000 [Versione 5.0.2195]
(C) Copyright 1985-1999 Microsoft Corp.

C:\>tracert www.unipi.it

Rilevazione instradamento verso www.unipi.it [131.114.190.24]
su un massimo di 30 punti di passaggio:
1 <10 ms <10 ms <10 ms r150.univ.trieste.it [140.105.50.254]
2 <10 ms <10 ms <10 ms 140.105.150.13
3 <10 ms <10 ms <10 ms utsgw48.univ.trieste.it [140.105.48.231]
4 31 ms 31 ms 47 ms rc-unit2.ts.garr.net [193.206.132.29]
5 31 ms 62 ms 47 ms mi-ts-2.garr.net [193.206.134.53]
6 47 ms 47 ms 47 ms bo-mi-2.garr.net [193.206.134.61]
7 125 ms 125 ms 125 ms pi-bo-1.garr.net [193.206.134.82]
8 * 200 ms 281 ms unipi-rc.pi.garr.net [193.206.136.18]
9 219 ms 312 ms 250 ms eth03-gw.unipi.it [131.114.188.61]
10 219 ms 187 ms 204 ms 131.114.186.1
11 250 ms 266 ms 266 ms solaria.adm.unipi.it [131.114.190.24]

Rilevazione completata.
C:\>
```

## Internet sta Colllassando : IPv6

Il protocollo IPv4 ha indirizzamento su 32 bit. Poiché il meccanismo del NAT è tanto criticato si è preparato un nuovo protocollo, **IPv6**, principalmente per risolvere il problema della scarsità di indirizzi a cui si stava velocemente arrivando.

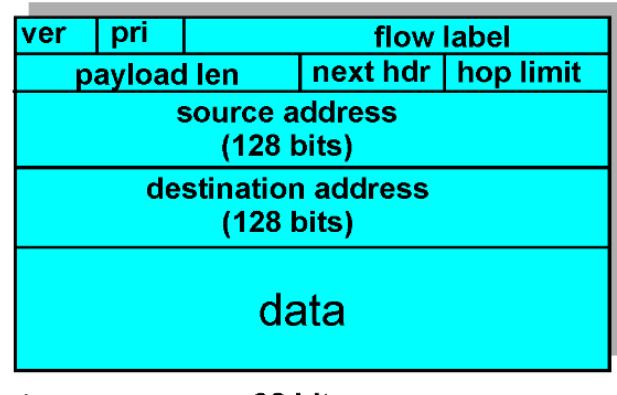
### L'intestazione del Datagram IPv6

Si è voluto creare, a differenza di IPv4, un header che risparmiasse ai router alcune operazioni fastidiose che ritardassero il processing del Datagram. L'header IPv4 sembrava infatti pensato per far perdere tempo, con il checksum che a causa del TTL decrementale doveva essere ricalcolato in ogni router.

La principale differenza è che ora **l'Header ha una lunghezza fissata di 40 byte**, perché sono state eliminate le opzioni. Analizziamo nel dettaglio i campi :

- **Version e pri** : sono bit dedicati alla versione (sta scritto il numero 6, se ci scrivi 4 non significa che è un datagram IPv4 ovviamente) e alla **priorità** del Datagram.

- **Flow Label** : è un nuovo concetto non descritto benissimo dall'RFC, che dice “è possibile identificare con una Flow Label quei datagram per cui il sorgente vuole un trattamento particolare da parte del destinatario”. Nella realtà la flow label è utilizzata, ad esempio, quando si vuole richiedere una QoS diversa in base ai dati che si stanno trasmettendo (potrebbe essere utile, ad esempio, identificare il traffico come voce / audio, così da cercare di richiedere a livello superiore una certa responsiveness).
- **Payload Length** : indica la dimensione del campo *data* (in byte). È bello notare come in IPv6 sia necessario indicare solo questa lunghezza, in quanto l'Header ha ora una lunghezza fissata
- **Next Header** : analogo di *Upper Layer* di IPv4
- **Hop Limit** : analogo di *TTL* di IPv4
- **Source e Destination Address** : sono ora su ben 128 bit, il problema dell'indirizzamento non è più un nostro problema!



## Le differenze da IPv4

- Ovviamente **lo spazio di indirizzamento** : questo non è più su 32 bit, ma su 128! Con questa enormità di indirizzi sicuramente non ci saranno problemi per le prossime generazioni. Se anche questi non basteranno più in futuro, sicuramente non sarà un nostro problema.
- **Non c'è più il campo Checksum** : questo campo, che richiedeva di essere calcolato ad ogni router perché il campo TTL decrementava, è stato dichiarato ridondante poiché la detection di errori avviene sia a livello data link, sia a livello trasporto!
- **Rimossa la possibilità di Frammentare i datagram** : se i router vedono un datagram troppo grande da instradare su un link con un certo MTU, si limitano a mandare un messaggio ICMPv6 (si, ICMP versione 6) con cui mandano un “*Packet Too Big*”: è dunque compito del sorgente e del destinatario svolgere questi compiti.
- **“Rimosse” le opzioni** : sono comunque utilizzabili, ma devono essere raggiunte mediante il campo *Next Header* dell'Header di lunghezza fissata.

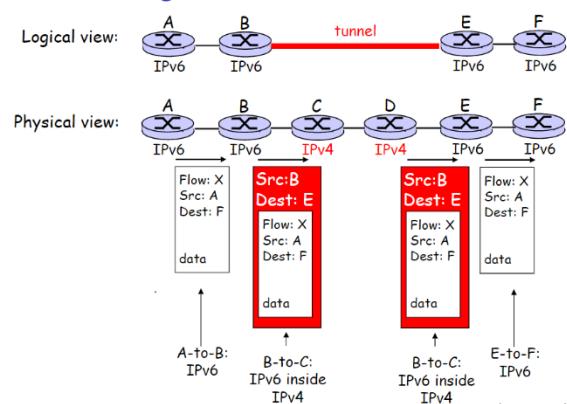
## Transizione da IPv4 a IPv6 : il Tunneling

Poiché ci si rese conto che non era possibile istituire un **Flag Day**, ovvero un giorno in cui tutti i dispositivi cambiassero da IPv4 a IPv6, (perché i dispositivi erano davvero troppi), si dovette ideare un sistema per permettere la coesistenza dei due tipi di protocollo.

- I dispositivi IPv6 possono essere dotati di *retrocompatibilità* grazie alla quale possono trasportare anche messaggi IPv4 (in particolare possono costruire autonomamente un Header IPv6 a partire dall'Header IPv4).
- Ma i dispositivi IPv4 come fanno a trasportare messaggi IPv6?

In particolare se un Host deve comunicare con un altro mediante una serie di router, qualcuno dei quali abilitato solo ad IPv4, ma il messaggio viene preparato in IPv6, come giunge a destinazione?

Il meccanismo ideato è quello del **Tunneling** :



- Il passaggio da router IPv6 a router IPv4 avviene incapsulando l'intero datagram IPv6 in un datagram IPv4, indicando come destinazione il primo router IPv6 più in la nel percorso. (Nel nostro esempio a lato, si crea un datagram IPv4 B – to – E).
- Il passaggio da router IPv4 a IPv6 avviene decapsulando il datagram IPv6 e riprendendo a spedirlo normalmente come se nulla fosse successo. In particolare se un router IPv6 riceve un datagram IPv4 con version **41** destinato a lui sa che all'interno del payload c'è un datagram IPv6, e che essendo lui il destinatario è tempo di decapsularlo.

## Risoluzione degli Indirizzi IP : il protocollo ARP

Abbiamo già detto che l'Host sorgente, e poi il router su ogni porta di ingresso, deve operare una *incapsulazione in frame* del datagram che prepara.

I frame però si spediscono in base a un indirizzo MAC, mentre i datagram seguono un indirizzamento basato su IP. Ma allora come si fa a decidere che indirizzo MAC scrivere quando si vuole spedire qualcosa a un certo indirizzo IP?

*Non esiste una funzione di traduzione X -> Y con X IP e Y MAC*, in quanto il MAC viene assegnato dai costruttori e l'IP viene assegnato non in base al MAC ottenuto. Non c'è dunque modo di trovare una funzione che traduca l'uno nell'altro.

Il compito del protocollo **ARP (Address Resolution Protocol)** è quello di compiere questa traduzione, similmente a come il DNS traduce i nomi in indirizzi IP.

L'idea è che ogni host e ogni router mantenga almeno una ARP Table, che conserva triple (IP, MAC, TTL), con TTL tempo di vita del record, oltre il quale viene dichiarato obsoleto (tipicamente 20 minuti).

*Scenario 1 : Invio nella propria rete locale*

Se l'host A deve inviare un datagram a un host nella propria rete locale consulta la sua tabella ARP e trova la corrispondenza IP -> MAC. Sarà il MAC corrispondente che sarà inserito nel campo *Source Address* del frame.

Come fa l'Host a riempire la sua Tabella ARP?

È semplice. Quando la traduzione non è risolvibile mediante tabella (per via di record mancante) si invia in broadcast (MAC FF:FF:FF:FF:FF:FF) un pacchetto ARP in cui si specifica il proprio indirizzo MAC e il proprio indirizzo IP, e si specifica inoltre l'indirizzo IP dell'Host di cui si vuole conoscere il corrispondente indirizzo Fisico.

Quando l'host locale destinatario riconosce l'indirizzo IP come proprio, risponde inviando al mittente il proprio indirizzo MAC.

*In questa fase sia l'Host mittente che il destinatario aggiungono eventualmente una riga all'interno della propria tabella ARP, poiché il destinatario riceve la coppia (IP\_Mittente, MAC\_Mittente) nel pacchetto ARP, e quindi può salvare il record nella propria tabella.* È per questo motivo che risponde in maniera *unicast*, direttamente al mittente giusto.

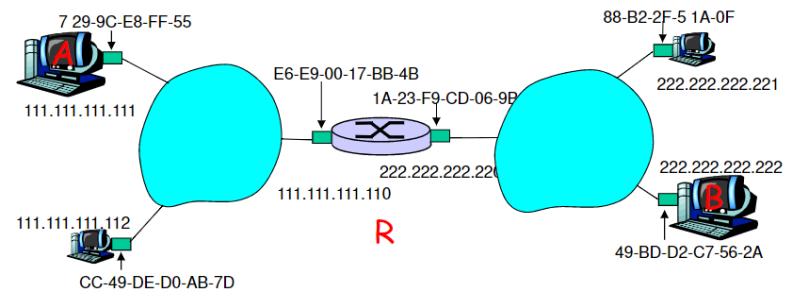
È dunque semplice capire come fanno gli Host in locale a riempire le proprie tabelle : "basta chiedere".

*Scenario 2 : Datagram destinato al di fuori della sottorete locale*

Secondo i meccanismi spiegati prima non sembra esserci modo per l'ARP table dell'host di contenere un record associato ad host che sono al di fuori della propria rete locale. In effetti è così, non si può.

**È qui che entrano in gioco i Router, che hanno più tabelle ARP, una per ogni interfaccia di uscita.**

Nell'esempio a lato, se A vuole comunicare con B inserisce nel source address del frame che prepara l'indirizzo MAC del router. Al router arriva correttamente il frame, che viene decapsulato. Il router consulta la sua tabella di forwarding basandosi sull'indirizzo IP contenuto nel datagram e sceglie un'interfaccia di uscita. A questo punto il router inserirà come MAC destinatario il destinatario finale (se è collegato alla sottorete finale) o quello del prossimo router consultando la sua seconda tabella ARP.



## CAPITOLO 6 : IL TRANSPORT - LAYER

Nonostante Internet provveda un servizio di tipo *Best Effort*, si potrebbe concludere che, in realtà, basta quello per terminare un servizio di comunicazione **Host - To - Host**. Il motivo per cui esiste anche il protocollo di trasporto è che questa comunicazione non basta, in quanto sugli Host sono in esecuzione, in generale, diversi processi e c'è dunque bisogno di instaurare una comunicazione **Process - To - Process**.

Nelle applicazioni Client - Server, ad esempio, è il processo Client che esegue il browser che fa richiesta al processo Server che gira sui server di Google e risponde alle richieste sulla porta 80.

Il compito principale del livello trasporto è dunque quello di instaurare una comunicazione **Process - To - Process**, ovvero deve implementare **Multiplexing** e **Demultiplexing** dei pacchetti.

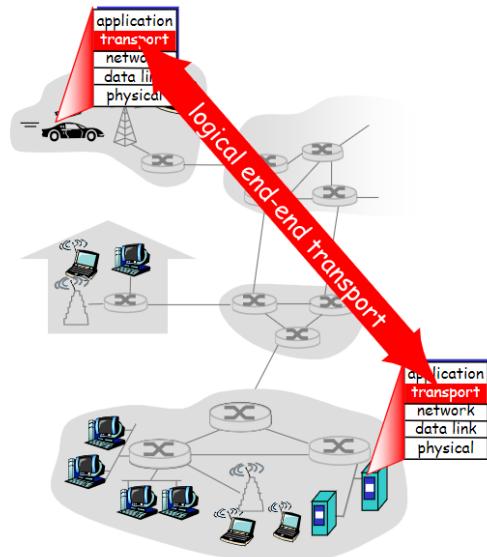
I due protocolli di livello trasporto principali esistenti sono TCP e UDP. UDP fa il minimo, ovvero implementa mux/demux. TCP fornisce servizi in più : trasporto affidabile, controllo di flusso e controllo della congestione.

Ma andiamo con ordine.

Il compito del livello di trasporto è instaurare una **connessione logica** tra i due host che vogliono comunicare. Sulla sending - side si prende il messaggio di livello applicazione e si suddivide in **segmenti**. Questi segmenti sono poi riassemblati nel messaggio originale nella receiving - side del livello di trasporto.

L'idea è di creare un'astrazione nella quale si ha l'impressione che esista un link punto - punto tra i due livelli di trasporto, mittente e ricevente.

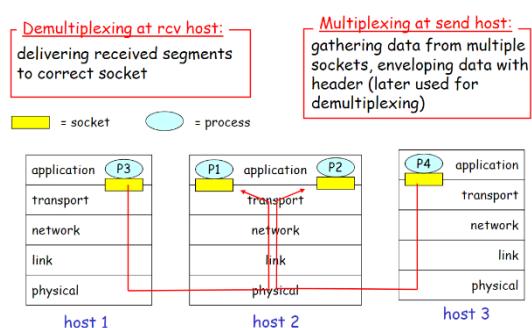
Sappiamo che in realtà non è così, ma questo le applicazioni non devono saperlo per forza.



## Il comune Denominatore : il (De)Multiplexing

Sia il protocollo TCP che UDP devono implementare questi due servizi. Cerchiamo di capire cosa sono nel dettaglio e cerchiamo di capire come sono implementati :

L'operazione di **Multiplexing** consiste nel raggruppare dati da Socket diversi (che coinvolgono processi diversi) in un unico flusso di segmenti. Ogni segmento deve però essere incapsulato mediante un header che deve fornire informazioni per le operazioni che devono essere fatte quando viene raggiunto l'host destinatario.



Grazie al livello network l'host destinatario viene raggiunto, ed è compito dell'operazione di **Demultiplexing** decapsulare il segmento e mettere i dati nel socket giusto. Per svolgere questa operazione è necessario inserire nell'header del segmento informazioni sulla porta del processo mittente e sulla porta del processo destinatario.

### Demultiplexing nei protocolli Connectionless

Il protocollo UDP è di tipo connectionless, non richiede che si instauri una connessione mediante handshaking degli host che vogliono comunicare. Il demultiplexing è così implementato :

- Arriva il Datagram all'Host, l'indirizzo IP viene riconosciuto come proprio, siamo sull'host giusto
- Il segmento (datagram decapsulato) viene visto dal protocollo UDP, che esamina l'intestazione
- Legge la **porta del destinatario** (destination port) nell'header e consegna il messaggio al socket corrispondente

Una proprietà importante è che se due IP sorgenti diversi mandano segmenti allo stesso IP destinatario alla stessa porta, il segmento arriva **allo stesso Socket**.

Formalmente si dice che il Socket UDP è identificato da una coppia (IP Destinatario, Porta Destinatario)

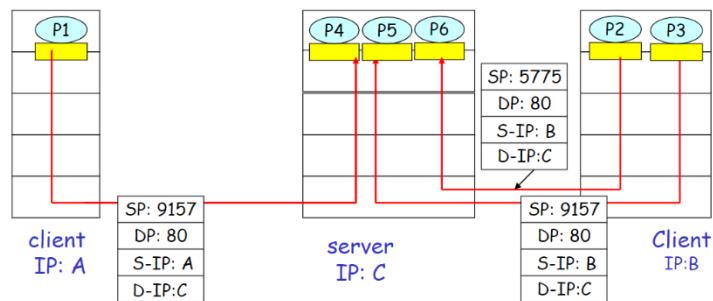
### Demultiplexing nei protocolli Connection - Oriented

Il **Socket TCP** è identificato non da una coppia, ma da una quadrupla :

(Source IP, Dest IP, Source Port, Dest Port)

Avere una quadrupla di questo tipo rende il concetto di connessione tra i due host che vi stanno partecipando.

Ha dei vantaggi mantenere anche Source IP e Source Port?



SI! In questo modo i Server Host possono creare un socket in ascolto sulla stessa porta per ogni client che vuole richiedere qualcosa. Nel caso di UDP questo non sarebbe stato possibile, in quanto a stessi Dest IP (Server Host) e Dest Port sarebbe corrisposto sempre lo stesso socket!

È dunque possibile **duplicare processi che servono le stesse richieste, e metterli in attesa su Socket diversi!**

In HTTP in modalità non - persistente si crea un nuovo server socket per ogni richiesta, anche se il Client è lo stesso.

Ai giorni d'oggi non è più buona norma creare un processo diverso per ogni richiesta e assegnarli un socket, è molto più efficiente fare un solo processo **multi - thread**, con ogni thread che gestisce un socket diverso.

## Il protocollo UDP : User Datagram Protocol

Il protocollo UDP (**User Datagram Protocol**) non estende la qualità del servizio offerta da Internet, nel senso che continua a garantire la non affidabilità della comunicazione.

L'unico servizio che offre è quello necessario al livello di trasporto, demultiplexing e multiplexing.

Il servizio è **connectionless**, ogni segmento UDP è trattato in maniera indipendente dagli altri e non c'è bisogno di una fase di handshaking iniziale tra i due Host che vogliono comunicare.

Perché esiste UDP se tanto è peggio di TCP?

UDP è in generale più veloce ed economico, in quanto non c'è bisogno di perdere tempo a stabilire connessioni point - to - point e non è neanche necessario conservare informazioni di stato grosse quanto il TCP (ricordiamo che UDP necessita di doppie, TCP di quadruple).

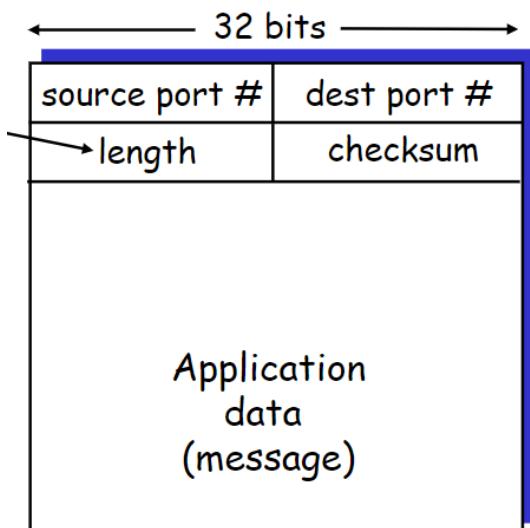
In generale il protocollo UDP è utilizzabile da quelle applicazioni che richiedono **rate sensitivity** (ovvero vogliono una certa risposta velocemente) e sono **loss tolerant** (ovvero non si aspettano il 100% dell'affidabilità). È il caso delle trasmissioni multimediali in streaming, dove se si rovina un po' il segnale il complesso è interpretabile in ogni caso. (Anche il DNS usa il protocollo UDP!).

### L'Header del protocollo UDP

L'header di UDP è grosso solo 8 byte. Servono solo alcune informazioni necessarie alla fase di demultiplexing (source port e dest port), e alcune informazioni di controllo :

- **Length** : rappresenta la lunghezza, in byte, dell'intero segmento UDP, Header + Messaggio di livello applicazione
- **CheckSum** : questo campo serve a permettere error detection. Se il checksum non torna, il segmento viene buttato via e non spedito a livello applicazione (anche se non c'è affidabilità e non è dunque prevista alcuna operazione di recovery).

Il checksum è calcolato dividendo l'intero segmento in interi di 16 bit, e sommandoli tutti a due a due, comando eventuale riporto che si genera dalla somma. Il risultato finale viene complementato e piazzato all'interno del campo checksum. Quando il destinatario rifà lo stesso conto (stavolta includendo il checksum) il suo risultato dovrà essere 1 --- 1, o l'errore sarà rilevato e il segmento scartato via!



## Il protocollo TCP : l'affidabilità in un colpo solo

Prima di addentrarci nell'argomento ricapitoliamo velocemente l'astrazione realizzata da questo protocollo :

- Il collegamento che si crea richiama quello **point - to - point**, come se i due host fossero direttamente collegati e l'unico servizio da realizzare fosse quello di mux/demux.
- Il protocollo è **connection - oriented** : c'è una fase di handshaking (detto *triplice handshake* perché richiede 3 messaggi) che deve precedere la fase di trasmissione dati
- La trasmissione è **full duplex**, ovvero A può trasmettere a B e viceversa contemporaneamente, senza che nulla si corrompa o faccia interferenza
- Implementa un **reliable data transfer** basato sul concetto di ACK, Retrasmissione e timeout
- Sono allocati sia sul mittente che sul destinatario dei **buffer** di dimensione variabile grazie ai quali è possibile bufferizzare pacchetti di cui non si può ancora procedere all'invio sul livello applicazione. I buffer sono necessari per via del fatto che i messaggi dell'applicazione non devono essere mandati così come sono ma vanno spezzati secondo un **MSS (Maximum Segment Size)**. Essendo l'header TCP + IP tipicamente di un totale di 40 byte, un MSS tipico è 1460 byte per rispettare l'MTU tipico di 1500 byte.
- Il TCP permette **flow control**, ovvero c'è rallentamento sulla trasmissione quando questa avviene troppo più velocemente della ricezione dei messaggi. In particolare se il buffer del ricevitore è pieno si vuole impedire che segmenti vengano inviati e subiscano buffer overflow, che è una situazione che fa sprecare solo risorse

- C'è anche un **controllo di congestione**, in cui si riducono o si aumentano le finestre dei vari buffer in base alla congestione sulla rete (*gestione dinamica di una pipeline*).

## Il formato del Segmento TCP

Il formato si mostra decisamente più articolato e complesso del segmento UDP, proprio perché il TCP deve garantire dei servizi in più (primo fra tutti l'affidabilità) e ha dunque bisogno di più informazioni.

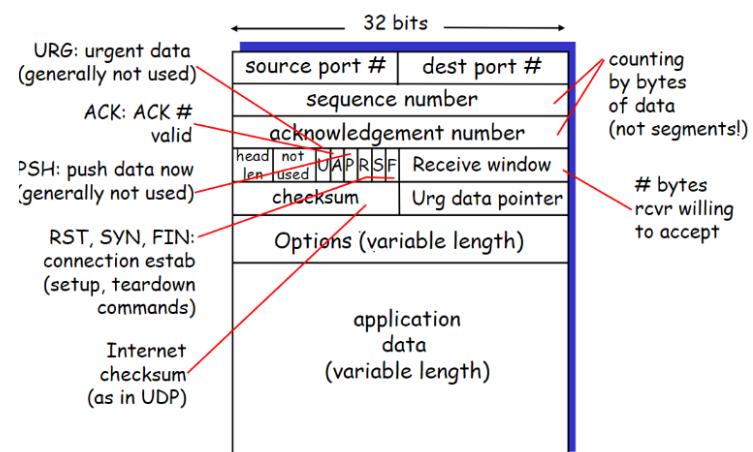
**Prima Riga :** (*Source Port | Dest Port*) sono, analogalmente al segmento UDP, il numero di porta del sorgente e del destinatario.

**Seconda Riga :** *Sequence Number* è il numero di sequenza e rappresenta il numero del **primo byte del campo dati nell'intera sequenza di byte in cui è stato suddiviso il messaggio di livello applicazione**. Questo

significa che se il messaggio era grande 3000 byte e c'era un MSS di 1000 byte, il primo segmento avrà un numero di sequenza 0, mentre il secondo avrà un numero di sequenza 1000, perché il primo byte del campo dati è il millesimo byte del messaggio originale.

**Terza Riga :** *Acknowledgement Number* rappresenta il numero di sequenza relativo **al prossimo byte che si aspetta di ricevere**. Se ad esempio, con riferimento all'esempio di prima, ho ricevuto un segmento con ACK number 1000, significa che ho ricevuto correttamente fino al byte 999, ovvero fino **all'ultimo byte del campo data del segmento, in ordine, precedente**.

Questa riga è valida, ovvero il segmento è un segmento di ACK, se e solo se il bit A è settato.

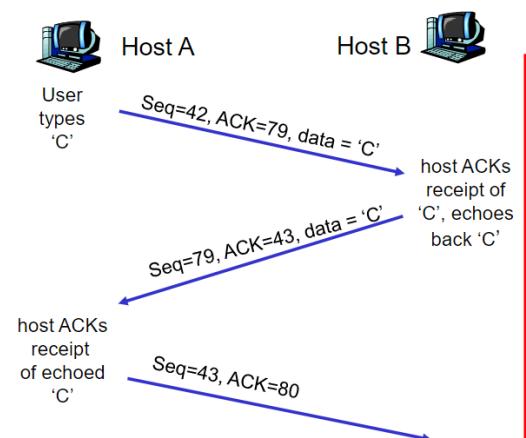


**Quarta Riga :** (*Header Length | Not Used | Flag | Receive Window*). Il campo header length rappresenta la lunghezza dell'intestazione del segmento, che è variabile per colpa della presenza delle opzioni. Il campo "not used" è, per l'appunto, riservato ad utilizzi futuri. Il campo flag descrive alcuni bit importanti, di cui mettiamo l'accento su A, R, S, F. A, come detto sopra, serve a determinare se la terza riga è o meno significativa. I bit R, S, F servono invece per la gestione della connessione (vedi avanti). La **receive Window** ci dice, in caso di messaggio di ACK, *quanti byte il ricevitore può ancora accettare*, e ovviamente trova utilizzo nel flow control fornito dal protocollo.

**Quinta Riga :** (*Checksum | Urg Data Pointer*) sono campi di cui non dobbiamo preoccuparci molto. Il checksum permette l'error detection così come nell'UDP (solo che nel caso del TCP una detection richiede retransmissione come vedremo). Il campo urg data pointer serve a marchiare un certo byte come "punto di inizio di una sezione urgente".

Nell'esempio a lato, in cui A chiede a B di fare la *echo* di un char (1 byte), cerchiamo di capire come funzionano numeri di sequenza e ACK :

- L'host A spedisce un numero sequenza 42 con ACK 79. Ciò significa che il primo byte del campo data è il 42-esimo nel flusso di byte che si stanno mandando (**il flusso non parte necessariamente da 0**). Inoltre significa che l'ultimo byte recepito dal flusso B - To - A è quello con numero 78, e ora aspettiamo il 79.
- Similmente, quando l'host B risponde ad A per fare l'ACK del messaggio inserirà numero di sequenza 79 (coerentemente con quello che A si aspetta) e ACK 43, per notificare che ha recepito il messaggio con sequenza 42 (che consisteva solo di un byte essendo un char) e ora si aspetta il byte con numero di sequenza 43.



- L'host A reagisce di nuovo, stavolta con sequenza 43 (quella che B si aspetta visto l'ACK precedente) e ACK 80 (visto che anche B ha mandato ad A un solo byte)

È importante notare che il protocollo TCP non specifica come gestire i pacchetti fuori ordine :

- Questi possono essere buttati via, e ci sarà la retransmissione degli stessi
- Questi possono essere **bufferizzati** e spediti *in burst*. Questo comporterà al destinatario di mandare due ACK consecutivi che però possono essere molto diversi tra di loro (in quanto tra il primo e il secondo sono stati spediti a livello applicazione più segmenti).

La scelta, in generale, è demandata al programmatore che è libero di implementare una o l'altra scelta liberamente.

## Gestione della Connessione TCP

Come si è detto più volte il TCP richiede che si stabilisca una connessione iniziale prima che i byte comincino a fluire. Durante la connessione :

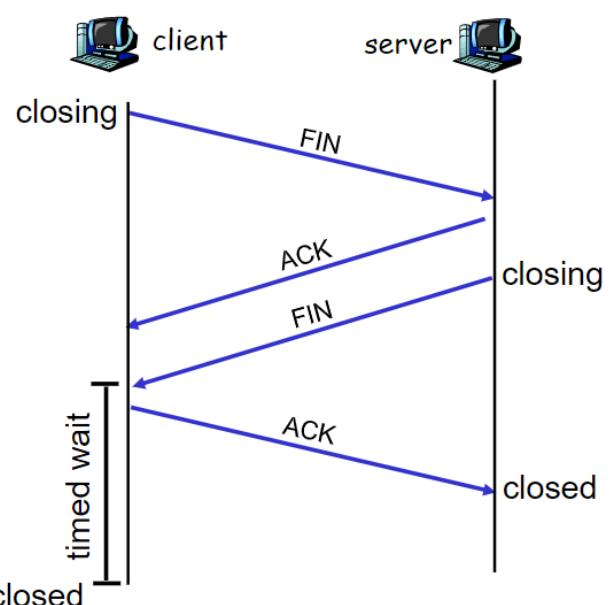
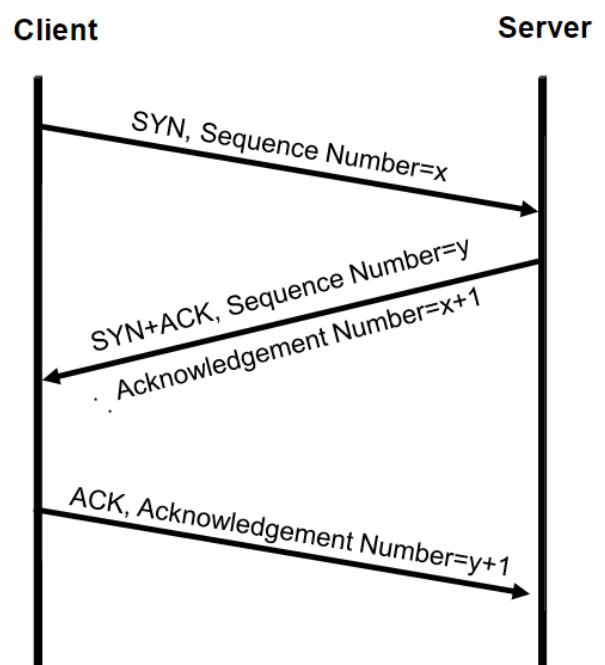
- Si devono stabilire e inizializzare alcune **variabili** : numero di sequenza iniziale, rcv window iniziale, ...
- Il client contatta il server, e il server è contattato

La connessione avviene mediante quello che si chiama **Triplice Handshake** (perché coinvolge tre messaggi)

1. **Il client** manda un segmento TCP col bit SYN alzato (a indicare che vuole fare setup della connessione) e comunica il suo numero di sequenza iniziale ( $x$ , generato randomicamente)
2. **Il server** risponde coi bit SYN e ACK alzati (per dire che ha capito che stanno stabilendoi una connessione). Anche lui dovrà comunicare il suo numero di sequenza iniziale ( $y$ , generato randomicamente) e manderà un ACK  $x + 1$ , coerentemente con quanto dice il protocollo
3. Il client risponde con un ACK number  $y + 1$ , la connessione è stabilita e il client può anche inserire dei dati in quest'ultimo segmento per cominciare la comunicazione.

Anche per quanto riguarda la chiusura della connessione è necessario che i due comunicano e "si dicano che la connessione sta finendo".

1. **Il client** comunica le sue intenzioni settando il bit FIN
2. **Il server** dice di aver capito mandando un ACK, aspetta un po'
3. **Il server** manda anche lui un segmento col bit FIN settato
4. **Il client** aspetta una timed wait prima di chiudere definitivamente la connessione. Durante quest'attesa manda l'ACK al FIN del server.



## Reliable Data Transfer nel protocollo TCP

Abbiamo già detto che il reliable data transfer si basa su tre concetti : ACK, Retrasmissione e Timeout.

Il primo scoglio da superare è appunto **decidere questo timeout** :

- Se troppo lungo, rischio di perdere troppo tempo ad aspettare pacchetti che tanto sono andati perduti
- Se troppo corto, rischio di ritrasmettere un enorme numero di duplicati

Si è detto a suo tempo che una buona scelta del timeout è “*intorno al valore di RTT*”. Nel caso del livello Data – Link era possibile stimare il RTT con una buona precisione, in quanto questo era determinato dalla somma dei ritardi di elaborazione, trasmissione e propagazione di un ACK in andata e ritorno.

Adesso non è più possibile, perché si implementano ritardi variabili all’interno di *ogni router* che il segmento attraverserà.

Per questo motivo è necessario fare una **stima del RTT**, basata sul concetto di Media esponenziale.

Qui a lato vediamo come funziona la media esponenziale, che si basa sul fatto che “i valori passati misurati nella storia abbiano un peso esponenzialmente diverso”. In base al valore scelto di  $\alpha$ , che deve essere sempre compreso tra 0 e 1, si sceglierà di dare più peso alla misurazioni passate o alle misurazioni recenti.

Al limite, se è 0, la stima è **immutata**, e coincide sempre con la stima iniziale. Se, viceversa, è 1, la **stima successiva coincide con la misurazione immediatamente**

**precedente**. Questo non è comunque un buon indicatore, c’è bisogno di tenere conto di una “media delle cose passate”. Raccogliendo algebricamente, ci rendiamo conto che la formula può essere notevolmente semplificata dal punto di vista **delle risorse che bisogna allocare**. Nel primo caso (sopra la freccia) era necessario *conservare l’intera storia delle misurazione degli RTT*. Nel secondo caso (sotto la freccia) bastano due valori, l’RTT misurato precedente e la stima corrispondente del passo precedente.

Si è notato che un valore buono per il parametro è 0.125 (grande importanza alle misurazioni recenti!).

Ma una volta calcolata una stima del RTT, come imposto il timeout? Mi basta metterlo uguale alla stima? Dovrei farlo leggermente più grande? Se sì, di quanto?

Una cosa su cui tutti sono d’accordo è che la stima del RTT non deve essere calcolata sui pacchetti che sono in ritrasmissione. Se calcolassimo la stima e arrivasse un ACK per quel segmento, non riusciremmo a capire se l’ACK si riferisce al segmento mandato ora (quello ritrasmesso) o quello mandato all’inizio (che ha fatto scattare il timeout).

Dunque una prima regola d’oro : non si calcolano i valori di RTT sui segmenti ritrasmessi.

*Una prima Soluzione : l’algoritmo di Karn – Partridge*

Questo algoritmo è molto semplice : calcolata una stima del RTT con la media esponenziale mobile poniamo :

$$\text{Timeout} = 2 \cdot \text{StimaRTT}$$

Notiamo subito una falla : un fattore moltiplicativo sulla sola stima non basta a mettere un timeout affidabile. Se la stima è ad esempio troppo larga per via delle misurazioni recenti, il timeout potrebbe finire per essere davvero troppo gigante.

## Il fattore Additivo : l'Algoritmo di Van Jacobson – Karel

Si è dunque pensato : “invece di moltiplicare la stima”, posso aggiungere *qualcosa di variabile in base alla situazione?* La ricerca di un fattore additivo ha portato a studiare la **Deviazione del RTT**, ovvero una proprietà di scostamento del valore reale rispetto alla stima effettuata sullo stesso.

Si vuole che la deviazione sia appunto tanto più alta quanto è stato più alto, nel tempo, lo scostamento tra stime e misurazioni. Ciò a cui si è arrivato è :

$$DevRTT_{n+1} = (1 - \beta) \cdot DevRTT_n + \beta \cdot |StimaRTT_n - MisuraRTT_n|$$

Si può scegliere un opportuno valore di beta per dare appunto più valore alla deviazione immediatamente precedente o in generale alla storia della stessa (valore tipico : 0.25).

Ora che abbiamo un valore che dipende dalle oscillazioni delle misurazioni rispetto alla stima, sappiamo che dobbiamo creare per il timeout un “margine di sicurezza” tanto più alto quanto DevRTT è alto. Secondo Jacobson :

$$\text{Timeout} = \text{StimaRTT} + 4 \cdot \text{DevRTT}$$

Ora che abbiamo una buona infarinatura di come il timeout può essere scelto, addentriamoci nel come il sender utilizza questa informazione? Quanti timeout ci sono? Come si reagisce a un timeout?

Il TCP utilizza un meccanismo *a singolo timer*, poiché introdurre un timer per ogni pacchetto not – ACKed richiede una certa quantità di overhead che vogliamo evitare.

Inizialmente consideriamo un sender TCP molto semplificato : assumiamo non ci sia bisogno di controllo di flusso e di congestione, che la comunicazione sia unilaterale e che i dati che arrivano da sopra rispettano MSS.

### Descrizione ad Eventi

**Ricevuti dati dal Livello Applicazione** : in questo caso il TCP sender crea un segmento TCP con numero di sequenza coerente e lo fa scorrere in base alla lunghezza (in byte, ricorda che il numero di sequenza è orientato al byte!) del messaggio. Dopodiché **se non c'è un timer già attivo**, fa partire il timer appena prima di passare il segmento a livello IP. Il fatto che il timer parte solo se non ce n'è un altro, significa che il timer attivo a un certo istante si riferisce **al segmento più “antico” di cui non c'è ancora ACK**.

**Timeout** : in caso di timeout è necessario che si ritrasmetta il segmento che non ha ACK con numero di sequenza più piccolo. Gli ACK sono cumulativi, ed è dunque necessario fare così, altrimenti un ACK potrebbe far intendere che sono state capite anche cose antiche che in realtà non sono state capite. Ovviamente si fa partire il timer di nuovo.

**ACK ricevuto** : se arriva un segmento col bit A settato, e quindi c'è il campo ACK valido, dobbiamo guardarne il valore, diciamo Y. Ogni TCP sender ha una **SendBase**, ovvero l'indice del primo byte di cui non si è ancora ricevuto ACK. In particolare, SendBase – 1 rappresenta dunque l'ultimo byte di cui è stato fatto ACK. Se Y è < SendBase, allora l'ACK è ridondante e non succede nulla. Se Y = SendBase, ricordando come il ricevitore invia l'ACK (ovvero invia il numero di sequenza del prossimo byte che si aspetta), **non succede nulla comunque**, perché quel byte è in invio ed è essendo quello a SendBase è quello a cui il timer sta facendo riferimento.

Infine, se Y > SendBase, allora significa che posso porre SendBase = Y per via di ACK cumulativo. Inoltre faccio ripartire il timer se ci sono segmenti not – ACKed da inviare / inviati ma senza ACK ricevuti.

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
    switch(event)
        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum = NextSeqNum + length(data)

        event: timer timeout
            retransmit not-yet-acknowledged segment with
            smallest sequence number
            start timer

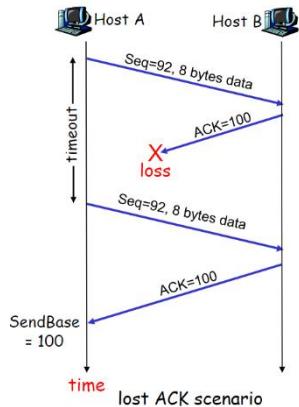
        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
            }
    } /* end of loop forever */
```

Come prima forma di controllo di congestione è possibile **raddoppiare il Timeout interval** ogni volta che questo scatta. Questa è solo una prima forma, molto meno articolata di quella che vedremo successivamente.

### *Scenari di Esempio*

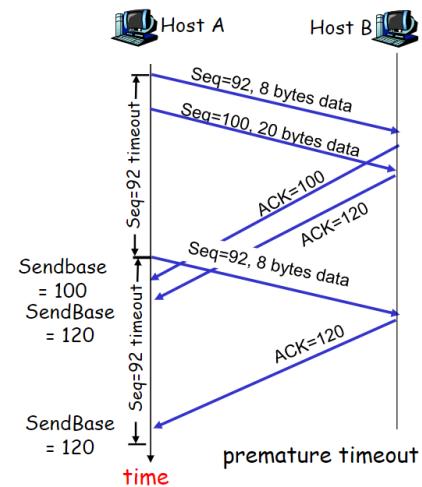
Nel primo scenario qui a lato vediamo l'Host A che vuole inviare 8 byte di dati all'host B. Il numero di sequenza utilizzato è il 92. Per questo motivo l'host B, che riesce a ricevere correttamente il segmento, invia ACK = 100 perché ha ricevuto correttamente gli 8 byte [92, 99] e il prossimo che si aspetta è il byte 100.

**L'ACK viene perduto**, il timeout scade e l'host A ritrasmette. Il segmento arriva all'host B anche questa volta. Vedendo un numero di sequenza inferiore a quello che si aspetta, rispedisce l'ACK = 100, e scarta il segmento. Quando l'host A stavolta riceve successivamente l'ACK, vede che  $100 > 92$ , e dunque avanza **SendBase** fino al numero di ACK ricevuto. Il primo byte di cui non ha ancora l'ACK diventerà dunque il byte con numero di sequenza 100.

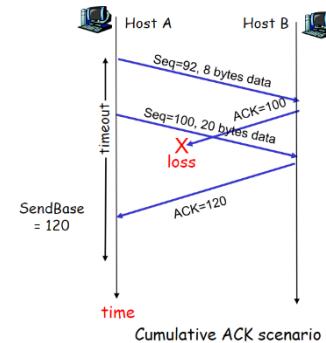


In questo secondo esempio l'host A invia due segmenti (uno di 8 byte e il secondo di 20 byte). Partendo dal numero di sequenza 92, il secondo ha numero di sequenza 100. I segmenti sono correttamente ricevuti da B, che invia due ACK, uno = 100 per notificare la ricezione del primo segmento, e uno = 120 per notificare la ricezione del secondo. **Il timeout, prematuro, scatta prima della ricezione di questi ACK**. Parte la ritrasmissione del segmento più vecchio in attesa di ACK, che è quello con numero di sequenza 92, e riparte il timer.

Mentre il timer scorre, arrivano i due ACK all'Host A. Il secondo (quello = 120) impedisce la ritrasmissione del segmento con numero di sequenza 100, che diventa "automaticamente ACKed". Il SendBase diventa dunque magicamente 120, anche se c'è stata la ritrasmissione solo del numero di sequenza 92.



In questo terzo scenario vediamo l'host A inviare ancora gli stessi due segmenti in conseguenza. Stavolta vediamo la **potenza dell'ACK cumulativo**. Essendo i segmenti correttamente ricevuti, vengono spediti, come prima, due ACK. Il primo ACK si perde, mentre il secondo arriva. Non ci sarà nessuna retransmissione, perché ACK = 120 confermerà la ricezione di *entrambi* i segmenti. È una bella cosa!

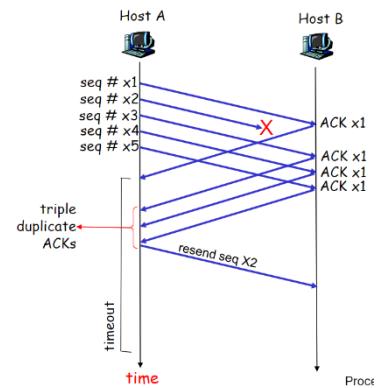


### *Il concetto di Fast – Retransmission*

Con il termine di **Ritrasmissione Veloce** intendiamo un altro modo di determinare la retransmissione. A volte potrebbe essere utile "dichiarare il pacchetto perso" ancora prima dello scadere del timeout, così da ritrasmettere prima e risparmiare tempo.

Il sender performa una ritrasmissione veloce quando riceve tre ACK duplicati per lo stesso segmento. Ma chiediamoci piuttosto **quando si genera un ACK duplicato**.

Quando il sender invia una serie di segmenti l'host B potrebbe non riceverne qualcuno (packet loss). Ma se ne perde qualcuno, non è detto che gli altri si perdano uguali. Questo crea il fenomeno della ricezione fuori ordine, in cui il receiver riceve dei segmenti "che non si aspettava", perché non corrispondono al "punto in cui è arrivato". Quando gli arriva un pacchetto fuori ordine, l'Host B rimanda un ACK relativo all'ultimo segmento in ordine che gli è arrivato, per segnalare "**guarda che io ho capito fino a qua non è che se mi mandi altre cose mi zittisco**". Si noti che questo meccanismo non è in contrasto con il fatto che l'host B potrebbe comunque bufferizzare i segmenti fuori ordine e spedirli "in burst" a livello applicazione quando fa *fill del gap che ha rilevato*.



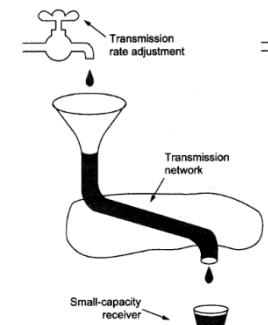
Proprio per il fatto che si potrebbe perdere un segmento ma gli altri no, si è deciso che il segnale che triggerà la fast retransmission sia un **triplice ACK duplicato**.

### IL TCP è GoBackN o SelectiveRepeat?

**Nessuno dei Due!** Avendo ACK cumulativo e non individuale sul singolo segmento anche out-of-order non è sicuramente SR (anche se esiste una versione, **TCP a selective acknowledgement** che rende il protocollo quasi uguale a SR). Non è tuttavia nemmeno GoBackN, in quanto in caso di ritrasmissione non si ritrasmette "dalla perdita fino a N", ma si ritrasmette **sempre al più un segmento**. Il receiver può infatti bufferizzare quelli out-of-order, rispondere alla ritrasmissione del segmento con una ACK che fa capire al trasmettitore che in realtà "grazie a questo filling è andato molto più avanti", e il trasmettitore non ritrasmette altro.

## Controllo di Flusso nel TCP

Il **controllo di flusso** (diverso dal controllo di congestione) consiste nell'evitare che il **buffer del ricevitore si riempia** e che si creino segmenti che si buttano per colpa del sender che invia troppo velocemente rispetto a quanto il receiver legga dal suo buffer. Situazioni di questo tipo si possono creare perché, ad esempio, l'applicazione legge dal buffer lentamente, perché ha altre cose da fare (non come il TCP sender, che deve fare solo quello).



Diversamente dal controllo di congestione che dipende dalla struttura del core quanto è trafficata, questo riguarda l'**end-to-end**.

Per inviare informazioni al sender, il ricevitore ha a disposizione (nei messaggi di ACK), il campo **receive window**, con cui cerca di comunicare lo spazio disponibile nel suo buffer.

Non si deve mai creare confusione sul fatto che la receive window è una fotografia che quando arriva al sender potrebbe essere già datata!

### Il punto di Vista del Ricevitore

Nei panni del ricevitore, immaginiamo che questo abbia un buffer di dimensione *RcvBuffer*. Facciamo in modo che il ricevitore mantenga due variabili di stato :

- **LastByteRead** : rappresenta l'ultimo byte che è stato letto dall'applicazione, e dunque "cancellato dal buffer" (o comunque sovrascrivibile).
- **LastByteRcvd** : rappresenta l'ultimo byte che è stato ricevuto e al momento è bloccato nel buffer.



La differenza tra i due valori può coinvolgere dei gap (perché magari il ricevitore ha bufferizzato dei pacchetti fuori ordine), ma sicuramente è minore della grandezza del buffer stesso.

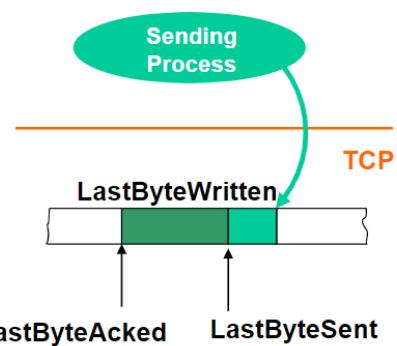
In un certo senso, possiamo concludere che la differenza rappresenta "lo spazio occupato" nel buffer del ricevitore. Ciò significa che la finestra annunciata, ovvero il valore che il ricevitore invierà nei suoi messaggi di ACK, sarà pari a :

$$\text{AdvertisedRcvWindow} = \text{RcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead})$$

### Il punto di vista del sender

Dal punto di vista del sender, ci sono due variabili di stato anche qui. Queste variabili servono perché **non è detto che si possa inviare esattamente tanti byte quanti indicati dalla finestra annunciata**. Quello, ricordiamo, è una fotografia di uno stato che potrebbe essere già cambiato. Per questo motivo il send memorizza :

- **LastByteAcked** : ultimo byte per cui è stato ricevuto l'ACK, ovvero che è giunto correttamente nel buffer del ricevitore
- **LastByteSent** : ultimo byte che è stato inviato ma non si è ricevuto l'ACK, e dunque sicuramente non è roba nel buffer del ricevitore



La differenza tra questi due valori esprime una porzione di dati che "è in viaggio", arriva sicuramente dopo che il ricevitore manda la finestra annunciata al trasmettitore. Per questo motivo il sender deve mantenersi, ovviamente, sotto la finestra annunciata, ma deve sottrargli la differenza tra questi due valori, per avere un margine di sicurezza che descriva il fatto che c'è roba in viaggio che presto arriverà (o è già arrivata ma lui ancora non lo sa) nel buffer del ricevitore.

$$\text{RcvWindow} = \text{AdvertisedRcvWindow} - (\text{LastByteSent} - \text{LastByteAcked})$$

**SITUAZIONE DI POSSIBILE STALLO** : se il buffer di ricezione si riempie, e viene comunicata una finestra annunciata pari a 0, il sender non invia più nulla. Tuttavia, se poi l'applicazione lato receiver preleva dati dal buffer (che dunque "si libera anche parzialmente"), il receiver non ha più modo di dire al sender che si è liberato, perché riesce a comunicare solo tramite messaggi di ACK, che però smette di inviare poiché il sender non invia più nulla. La soluzione a questo problema è fare in modo che il sender, periodicamente, mandi un segmento *di 1 byte* per stimolare la reazione del receiver. Se il buffer resta pieno, ok. Ma se il buffer si è svuotato il byte riesce ad essere bufferizzato e il receiver manda un ACK, comunicando l'informazione che manda al sender per sbloccarsi!

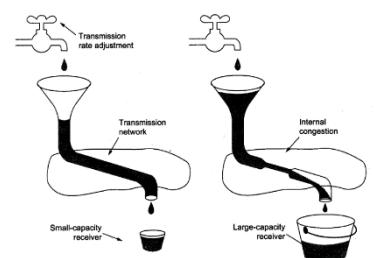
## CAPITOLO 7 : IL CONTROLLO DI CONGESTIONE

### Introduzione alla Congestione

La congestione è un fenomeno che si verifica quando tante sorgenti mandano diversi pacchetti sul network. Se tanti lo fanno anche a una discreta velocità, si cominciano a riempire le code dei router. Se queste si riempiono, si forma il fenomeno di **packet loss** a livello network. Anche se le code non si riempissero, la congestione ha anche come conseguenza quello di aumentare significativamente il queueing delay, che in generale rappresenta la maggior parte del ritardo end - to - end qualora la comunicazione coinvolgesse il network.

Il controllo di congestione **non coincide con il flow control**. Il secondo è un accortezza che il trasmettitore prende quando il buffer del ricevitore rischia di traboccare. Il primo è invece un controllo che si fa quando è il "percorso" end - to - end ad essere intasato!

Principalmente per risolvere la congestione ci sono due approcci :



- **Network - Assisted** : in questo caso sono gli stessi router che danno dei segnali di *feedback* agli end - system. Quando gli host ricevono queste informazioni dal network, allora aggiustano il rate di conseguenza. Il feedback potrebbe essere semplice, ad esempio ponendo un solo bit a segnalare la presenza o meno di congestione, o più articolato, con un feedback che indica anche **il rate al quale regolarsi** (valore preciso, oppure solo se aumentare / diminuire).
- **End - To - End** : in questo caso gli host mittente e destinatario non ricevono alcun feedback da parte del network, e dunque devono cercare di rilevare la congestione e aggiustare il proprio rate secondo proprie regole. È chiaro che la cosa si risolve solo se **tutti ragionano così!**

Un esempio di approccio Network - Assisted è realizzato nelle reti ATM, mentre un approccio End - To - End è implementato nel TCP. C'è poi chi questi controlli non li fa proprio, come **UDP**, che ha bisogno di un traffico il più continuo possibile e dunque non se ne importa nulla di nessuno, continua a mandare.

Nota : il TCP si appoggia sul protocollo IP, che è inaffidabile. Dunque non ha altra scelta di scegliere l'approccio end - to - end.

## Controllo di Congestione nelle reti ATM

Analizziamo uno dei servizi delle reti ATM, detto **ABR (Available Bit Rate)** e cerchiamo di capire come un controllo di congestione si rende necessario per rendere questo servizio effettivo.

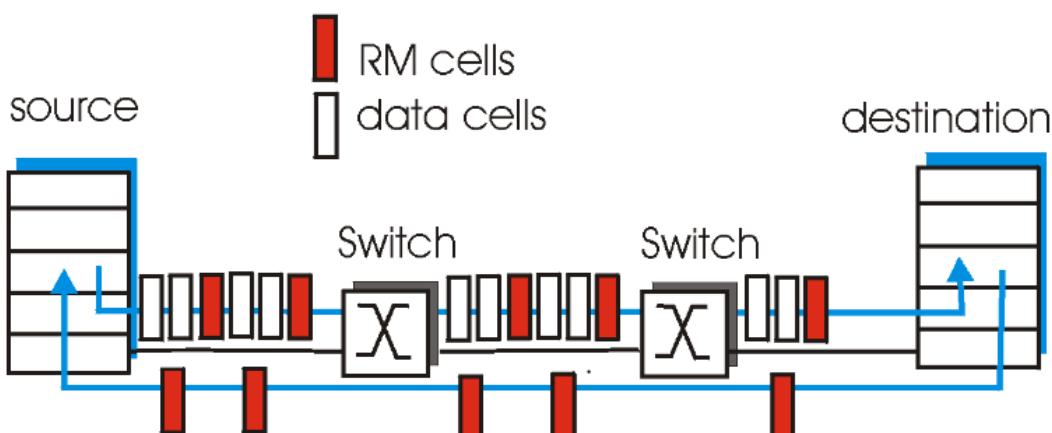
ABR è un servizio elastico, in cui :

- Se il link collegato al sender è libero, allora il sender manda a un rate più elevato
- Se il link collegato al sender è in congestione, allora **manda alla minima banda garantita** (servizio unico delle reti ATM!)

Il controllo di congestione delle reti ATM è **network - assisted**, ovvero sono i dispositivi del Core a dare informazioni al sender sullo stato di congestione della rete.

In particolare, insieme alle celle di dati il sender manda periodicamente una **Cella RM (Resource Management)**. È compito del dispositivo del core settare opportunamente dei bit di questa cella, e fare in modo che il destinatario finale rispedisca la cella al mittente per fargli ottenere quelle informazioni. I bi di cui stiamo parlando sono principalmente :

- **NI : No Increase In Rate**, ovvero si vuole segnalare al Sender di "non andare più veloce" perché siamo vicini a una possibile congestione
- **CI : Congestion Indication**, ovvero si vuole segnalare al Sender che c'è congestione, e dunque ci si deve abbattere alla minima banda garantita



In particolare, è compito del Core settare un bit speciale EFCI per segnalare la congestione. Quando il destinatario prenderà in mano la cella RM e vedrà il bit EFCI settato, dovrà rispedire la cella RM al mittente col bit CI settato!

È possibile utilizzare altri due byte ER (Explicit Rate) all'interno della cella RM, e questi due bit sono sempre manipolati dagli switch per segnalare al mittente un tetto massimo oltre il quale non deve andare (durante il percorso lo switch congestionato abbassa dunque questo valore). È di nuovo compito del destinatario finale rispedire queste informazioni al mittente!

## Controllo di Congestione nel TCP

Per capire come fa il TCP ad operare un controllo di congestione dobbiamo anzitutto capire cosa dovrebbe fare in linea di massima. Ricordiamo che l'obiettivo è che il sender “aggiusti il suo rate”, e che lo alzi o lo abbassi a seconda di una congestione rilevata. Detto a parole è bello ma :

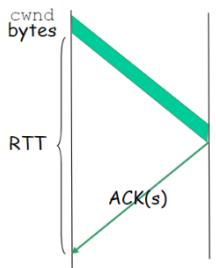
- 1) *Come fa il Sender a limitare il suo rate?* Come può porre un tetto massimo più alto o più piccolo?
- 2) *Come fa a rilevare una congestione?* Siamo in un approccio non network – assisted, quindi in realtà nessuno gli dice che nel network c'è congestione
- 3) *Secondo quale algoritmo regola il suo rate?* Lo abbassa? E se si, di quanto?

### Come fa il Sender a limitare il rate e a rilevare una congestione

Il Sender memorizza una variabile di stato oltre a quelle nominate durante i discorsi sul controllo di flusso : **la Congestion Window cwnd**.

In particolare il sender limiterà i byte che può inviare sia basandosi sulla receive window del buffer del ricevitore, sia sulla sua congestion window.

Assumiamo che il ricevitore non abbia una receive Window problematica. Allora significa che, mediamente, si possono mandare **cwnd / RTT** byte al secondo. Questa rappresenta una finestra massima entro la quale il sender “tollerà” byte non ACKed.



Questo significa che per limitare il proprio rate il sender deve limitarsi a diminuire la quantità **cwnd** (i discorsi sono molto semplificativi, ma il concetto è che il sender è capace di autolimitare il proprio rate senza aver bisogno di ricevere informazioni di receive window dal receiver).

Dunque il trasmettitore diminuisce la propria cwnd ogni volta che rivela congestione, ma riesce davvero a rilevarne una?

Il TCP assume che sia stata congestione **ogni volta che si verifica un fenomeno di Loss**. Questo, ricordiamo, può avvenire in due situazioni differenti :

- Perché si è verificato un timeout. Questo è un sinonimo molto forte di congestione, perché sembra davvero che il pacchetto si sia perso nei vari router. **Non è sicurezza di congestione**, in quanto il timeout potrebbe essere semplicemente prematuro.
- Perché si è verificato un triplice ACK duplicato. Anche questa **non è sicurezza di congestione**, in quanto il pacchetto potrebbe semplicemente star sperimentando un ritardo nei router e magari gli sono arrivati prima altri segmenti. Anzi, un fenomeno del genere è sinonimo di congestione debole, in quanto il fatto che ci sia triplice ACK duplicato significa che qualcosa al receiver sta arrivando, semplicemente non è ancora arrivato il prossimo in-order-segment!

Non perdiamo di vista la differenza tra congestione debole e forte, ci sarà utile fra poco. Per adesso, ci basti sapere

### L'algoritmo di controllo del Rate del sender

Il sender controlla il suo sending – rate basandosi su dei “feedback impliciti” che gli danno “informazioni” sul grado di congestione della rete. Esistono feedback negativi (come detto prima, i fenomeni di loss) e feedback positivi (ACK corretti ricevuti!). Il sender regola il suo rate in tre fasi :

- 1) ***Slow Start***
- 2) ***Congestion Avoidance***
- 3) ***Fast Recovery*** (questa è opzionale, ma vedremo che è utile)

Il primo scoglio da superare è “come parto”?

Nella fase di slow start (che poi tanto slow non è), si sceglie di :

- Porre inizialmente **cwnd = 1 MSS**, e dunque trasmettere a rate circa uguale a MSS/RTT. Questo significa.
- Per ogni ACK ricevuto, incremento cwnd di un numero pari agli ACK correttamente ricevuti. Ciò significa che al primo ACK ricevuto cwnd = 2MSS, dopodiché arriveranno verosimilmente due ACK (ho mandato due segmenti MSS) e dunque sarà cwnd = 4MSS. Questo significa che ogni volta **raddoppio la mia finestra di congestione**. Dunque la crescita è esponenziale.

Purtroppo non sempre è privamerà : *prima o poi ci sarà congestione*. Quando la congestione è rilevata, il sender comincia a memorizzare un’altra variabile di stato, la **threshold (soglia)**. Questa soglia viene posta a  $cwnd/2$  quando viene rilevata congestione.

Per un primo esempio, consideriamo che rivelata la congestione si abbassi cwnd = 1MSS e si setti la soglia. Una volta che cwnd si avvicina alla soglia se continuiamo a raddoppiarla ogni volta significa che *la congestione è dietro l’angolo*. Non è una scelta saggia continuare a raddoppiare ogni volta dopo la soglia visto che la soglia è stata settata visto che da un raddoppiamento dalla stessa è stata rilevata congestione.

Di conseguenza si sente l’esigenza di “cambiare comportamento” al raggiungimento della threshold. Si dice che quando si raggiunge la soglia si passa alla fase di **congestion avoidance** (avoidance detto un po’ male, visto che in realtà siamo qui perché siamo vicini alla congestione, ma vabbè, è a interpretazione).

In questa fase si cambia modo di aumentare la finestra di congestione : si passa da una crescita esponenziale a una crescita lineare, ovvero la crescita avviene facendo  $cwnd = cwnd + 1MSS$ .

Questo sembra aver attenuato i problemi, ma è tempo di rinfrescare il discorso su congestioni “serie” e “meno serie”.

Un approccio possibile è il seguente :

- Aumento esponenzialmente in slow start fino al raggiungimento della soglia
- Aumento linearmente dalla soglia in poi, fino a un packet loss (congestion avoidance)
- **Atterro cwnd a 1MSS, e riparto da slow start**

Questo è l’approccio seguito quando non esiste la fase di Fast Recovery, ovvero quando trattiamo allo stesso modo sia il Loss provocato da timeout sia il Loss provato da triplice ACK duplicato.

In realtà sappiamo che nella seconda situazione **la congestione è meno grave**, perché per mandare ACK duplicati significa che qualcosa al ricevitore sta arrivando.

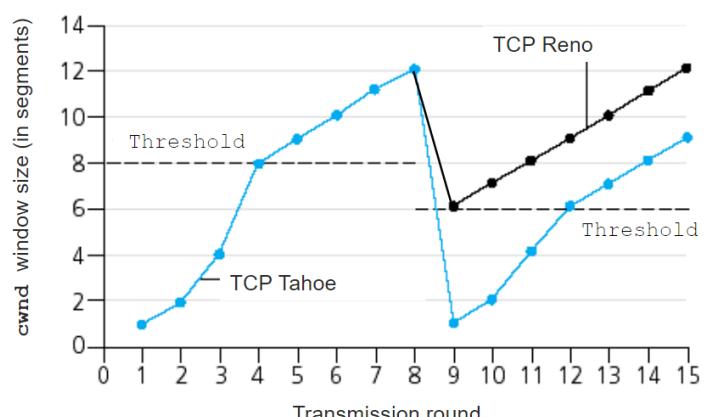
Nelle versioni TCP che implementano il **Fast Recovery** si evita di atterrare cwnd a 1MSS in caso di triplice ACK duplicato, ma si pone uguale a threshold + 3MSS (con i 3MSS che si riferiscono ai tre ACK duplicati che sono

### □ 3 Duplicate ACKs

- **Threshold=Cwind/2**
- **Cwind=Cwind/2 + 3 MSS**
- **Congestion avoidance (cwind increases linearly)**
  - Fast Recovery

### □ Timeout

- **Threshold=Cwind/2**
- **Cwind=1**
- **Slow Start (cwind increases exponentially)**



arrivati, e che segnalano "cose che sono arrivate al ricevitore").

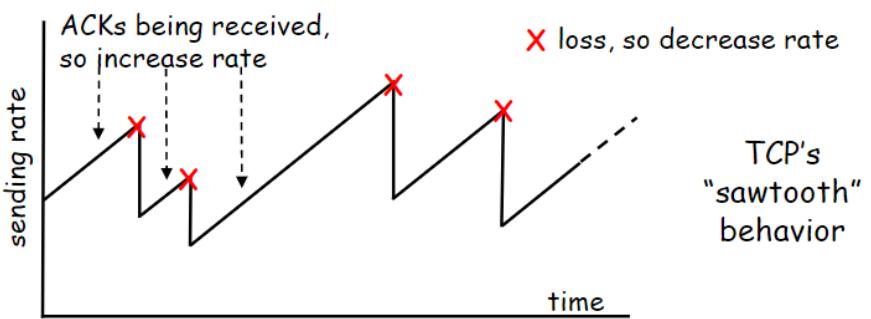
Siano ad esempio TCP Tahoe una versione del TCP che non implementa fast recovery e sia invece TCP Reno una versione che lo implementa. Nel caso di triplice ACK duplicato a cwnd = 12MSS :

- La soglia si abbassa in entrambi i casi a 6MSS
- Per il TCP Tahoe si ripone cwnd = 1MSS (in azzurro)
- Per il TCP Reno si pone invece cwnd = 6MSS + 3MSS = 9MSS (in nero). Nella figura sembra che riparta da 6 MSS, ma è sbagliata.

## La fairness del TCP

È il TCP un protocollo equo? Se ci sono K connessioni TCP che vogliono comunicare utilizzando un link R, comunicano a un rate medio di K/R?

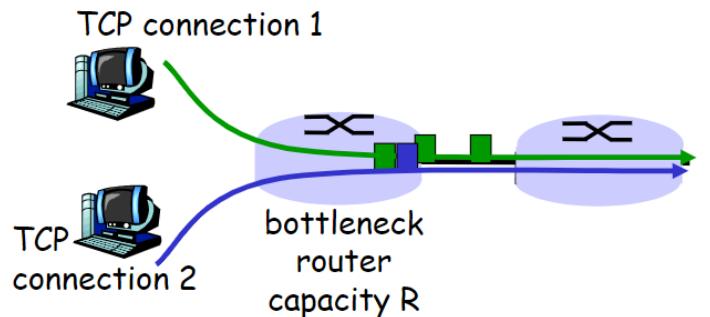
Analizzando il sending rate del mittente TCP, questo cresce linearmente (a partire dalla fase di slow start), e poi si dimezza in caso di triplice ACK duplicato (o si atterra in caso di timeout). In generale, l'andamento è a **dente di sega** (più evidente se, come a lato, il loss accade solo per triplice ACK duplicato).



Per capire se è fair, facciamo un esempio pratico.

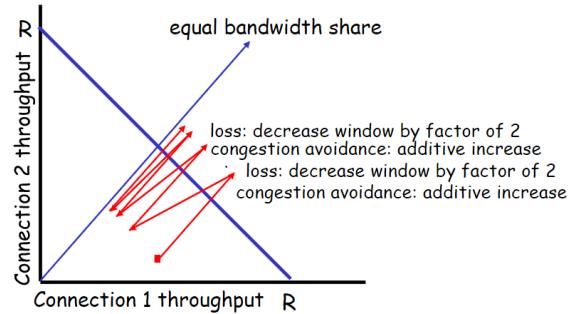
Facciamo un po' di assunzioni :

- Sulla rete esistono due (e solo due) connessioni TCP che attraversano lo stesso **bottleneck link** di capacità  $R$ . Questo significa che stiamo assumendo che su tutti gli altri link ci sia abbastanza banda da non rilevare congestione prima che sul link di capacità  $R$
- Assumiamo che le due connessioni TCP abbiano lo stesso RTT e lo stesso MSS. Questo significa che la loro finestra di congestione aumenta nello stesso modo (MSS uguale) e il rate dipende dagli stessi fattori (Finestra / RTT).
- Assumiamo che abbiano grossi dati da mandare (e che quindi proveranno ad abusare la banda) e che **agiscano sempre in congestion avoidance**, e non arrivino mai a slow start (che poi è un altro modo di dire che i loss capitano solo per triplice ACK duplicato)



Sotto queste assunzioni, possiamo concludere che nessuna delle due connessioni raggiunge mai un throughput maggiore di  $R$ , e che il protocollo sarebbe fair se le due connessioni avessero un throughput medio di  $R/2$ . Questo è un caso ideale, ora vediamo come vanno le cose :

- Partiamo da un generico punto (quello più in basso ad esempio) in cui la **somma dei throughput** è < R. Se è così, visto le assunzioni precedenti, significa che entrambi possono aumentare il proprio rate. Essendo in fase di congestion avoidance, e avendo lo stesso MSS, la linea cresce parallela alla bisettrice del primo quadrante (45 gradi). Arriva un punto in cui la somma dei due throughput eccede R e, dunque, arriva la congestione.
- Alla detection della congestione entrambi (assunto che il loss, come detto sopra, avviene sempre per triplice ACK duplicato) dimezzano la propria finestra di congestione, e dunque il loro throughput.
- Da lì in poi cominciano di nuovo a crescere linearmente, e il giro si ripete.
- **In conclusione, il throughput in ogni istante fluttua in maniera parallela alla bisettrice, che è la linea che rappresenta un "ugual utilizzo di banda".** Questo è, in altri termini, una dimostrazione che il protocollo è fair.



Le linee sono più “dense” al centro della X perché i dimezzamenti coinvolgono numeri più piccoli, ciò significa che anche facendo avanti e indietro spesso (ovvero che si fluttua sia sopra che sotto la linea dell’equal bandwidth) in realtà si sta più tempo vicino al centro!

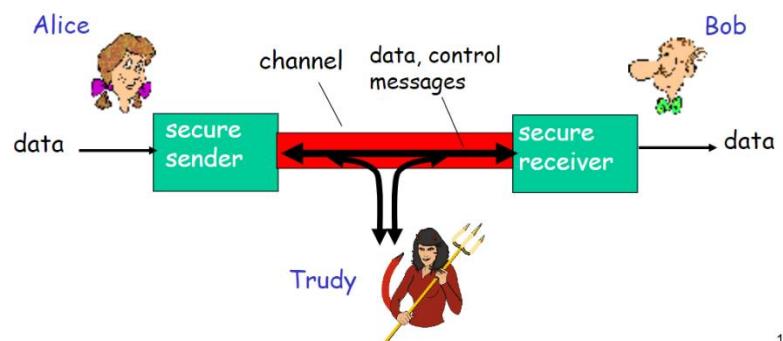
**Buchi del ragionamento :** la fairness si rompe se ci sono degli host che stanno trasmettendo secondo il protocollo UDP (perché quelli non fanno controllo di congestione e mandano quando vogliono) oppure se la stessa applicazione *apre più connessioni TCP in parallelo*, sfruttando il fatto che ogni connessione TCP prende grossolanamente la stessa banda e quindi appropriandosi di una fetta maggiore della stessa.

## CAPITOLO 8 : LA SICUREZZA IN RETE

### Introduzione : Il bisogno della sicurezza

La prima cosa che dobbiamo sapere prima di parlare di sicurezza è la **consapevolezza che la rete non è un posto sicuro**. Esistono diversi tipi di attacchi, quale più dannoso e quale un po’ meno, a cui ci esponiamo ogni volta che siamo collegati. Prima ancora di chiederci “come ci proteggiamo” è necessario sapere quello che stiamo rischiando :

- **Ci sono i packet sniffer, i cocainomani della rete** : queste persone si introducono sui canali di comunicazione ricevono i pacchetti indirizzati alle altre persone. L’attacco può diventare del tipo *“record and playback”* quando il bad guy riesce a memorizzare alcune informazioni sensibili (come una password) e a riutilizzarle a proprio piacimento (come appunto utilizzare la password per accedere a cose di qualcun altro).
- **Ci sono i masquerade**, ovvero quegli host a cui piace inserire nel campo *source address* l’indirizzo di qualcun altro, così da provare a mascherare la propria identità e contemporaneamente a spacciarsi per un’altra persona (**attacco IP Spoofing**)
- Ci sono gli attacchi **DoS (Denial of Service)** in cui più host cercano di



impiegare inutilmente le risorse di un certo server, così da negare i servizi a chi realmente ne ha bisogno. Attacchi di questo tipo sono spesso performati mediante una rete di host infettati precedentemente mediante malware o worm, creando una **botnet**, una rete di host inconsapevoli che il bad guy utilizza per mascherare la sua reale identità

Gli attacchi non coinvolgono necessariamente sempre un intermediario su un mezzo comune. Molte volte è lo stesso utente che inconsapevolmente "si lascia attaccare da un bad guy". È il caso in cui l'utente scarica inavvertitamente un **Trojan Horse**, un contenuto malevolo nascosto all'interno di software apparentemente innocuo e utile per l'utente che vuole farne uso. Potrebbe involontariamente ricevere degli **Spyware** **Malware**, dei software spia che sono in grado di copiare e trasmettere informazioni sensibili sulla vittima all'attaccante. Inoltre l'host può essere infettato quando esegue inavvertitamente un **virus**, che riesce a replicarsi da solo (e a creare dunque una botnet), ma **che ha bisogno comunque della vittima che lo esegue inavvertitamente**, o da un **worm** che, in maniera ancora peggiore, **non ha bisogno di nessuno che lo esegua**.

Tutti questi discorsi ci portano a una sola conclusione : abbiamo bisogno di un meccanismo superiore che ci faccia sentire sicuro, e che si basi principalmente su 4 principi fondamentali :

- **Confidenzialità** : due persone che vogliono comunicare (in generale due host, si parla anche di due server, di una persona con un server, ecc.) vogliono assicurato il fatto che i fatti loro non li legga qualcun altro. Nasce l'esigenza della **crittografia**, ovvero un meccanismo secondo il quale se proprio qualcuno deve sniffare i dati non deve poterne decifrare il contenuto.
- **Integrità** : non vogliamo che un intruso possa modificare, inserire dei messaggi o addirittura impedirne la ricezione.
- **Autenticazione End - To - End** : i due che hanno intenzione di comunicare devono essere in grado di riconoscere l'altro come tale. C'è bisogno che dunque i due siano in grado di assicurarsi che il messaggio arrivi esattamente dall'altra persona, e non da un intruso che fa IP spoofing
- **Accesso e Disponibilità dei Servizi** : non si vuole che, per colpa dei bad guy, dei servizi siano resi non disponibile. È necessario introdurre dei sistemi di protezione (come i *firewall*) che impediscono attacchi di tipo DoS.

## La Confidenzialità e la nascita della Crittografia

La crittografia si basa sul principio della confidenzialità. *Esiste un modo per far sì che due persone possano scambiarsi messaggi comprensibili solo a loro due?*

Introduciamo un po' di terminologia :

- Chiameremo **messaggio in chiaro** (*plaintext*) il messaggio originale non cifrato che A vuol mandare a B
- Chiameremo **crittogramma, o messaggio cifrato**, (*ciphertext*) il messaggio che viaggia sul canale di comunicazione e che risulta illegibile ad ogni intruso.
- Ovviamente le due parti devono riuscire a mettersi d'accordo su un certo **algoritmo di cifratura e decifratura**. Al giorno d'oggi questo algoritmo è pubblico.

Ma se gli algoritmi sono pubblici dove sta la segretezza? La magia sta nell'esistenza delle **chiavi**, degli speciali valori di input per l'algoritmo che decifra il messaggio cifrato.

Diremo dunque che se A vuole comunicare con B e vuole spedire un messaggio m :

- A applica l'algoritmo di cifratura utilizzando una chiave, e spedisce sul canale il ciphertext  $K_A(m)$
- B applica l'algoritmo di decifratura utilizzando un'altra chiave, e riceve  $K_B(K_A(m))$
- Dunque le cose devono essere costruite in modo che  $K_B(K_A(m)) = m$ !

Nella crittografia a **chiave simmetrica**, ogni coppia di host ha la stessa chiave. Nella crittografia a **chiave pubblica**, ogni utente ha una coppia di chiavi : la chiave pubblica di un certo utente è nota al mondo esterno, mentre la chiave privata è nota solo a lui.

## Crittografia a Chiave Simmetrica

Nella crittografia a chiave simmetrica i due host che hanno intenzione di comunicare conoscono entrambi la stessa chiave per codificare / decodificare il messaggio cifrato.

La versione più nota storicamente è il **cifrario di Cesare**, che consisteva nello shiftare in avanti (in senso circolare) le lettere del messaggio di una quantità costante "K".

Se ad esempio K = 3, 'A' diventa 'D', 'Z' diventa 'C', e così via.

Ovviamente un cfrario del genere non è robusto, in quanto le possibili K sono 26 (se è maggiore si ottiene lo stesso risultato prendendo K % 26 quindi siamo sempre lì). Un attacco a forza bruta impiegherebbe davvero poco tempo a rompere il messaggio!

Nascono per questo motivo altri cfrari leggermente più sofisticati, i **cfrari monoalfabetici**. In questo tipo di meccanismo gli host avevano una tabella di corrispondenza lettera - lettera. Ad ogni lettera, in qualunque posizione questa si trovi, ne viene tradotta un'altra indipendentemente da tutto, senza seguire pattern (cosa che succedeva nel cfrario di Cesare).

Ovviamente un cfrario monoalfabetico aumenta esponenzialmente la dimensione delle chiavi possibili. Infatti una chiave possibile ora è una **qualsiasi permutazione dell'alfabeto**, e dunque abbiamo 26! Chiavi possibili. Con l'attacco a forza bruta sarebbe dabbero difficile forzare un meccanismo del genere.

Ma gli attaccanti sono astuti, e sanno che la stessa lettera è sempre tradotta nella solita lettera. Ciò significa che, specie se si intuisce la tipologia del messaggio che si sta inviando, si possono provare a intuire alcune lettere, riducendo significativamente la complessità dell'algoritmo a forza bruta.

Nascono dunque i cfrari **polialfabetici**, che funzionano in questo modo :

- Si raccolgono N cfrari monoalfabetici
- Si introduce un **pattern ciclico** di utilizzo dei suddetti cfrari
- La prima lettera sarà cifrata usando il primo cfrario del pattern, la seconda usando il secondo, e così via a rotazione

Nei cfrari polialfabetici la chiave è costituita dalla conoscenza dei cfrari utilizzati e dal pattern di utilizzo scelto. Questo risolve eventualmente il problema degli attaccanti che riescono a intuire lettere, perché **ora la stessa lettera è cifrata in modo diverso a seconda della sua posizione nella frase**.

Esistono in questo tipo di crittografia due tipi principali di cfrari :

- 1) **Cfrari di tipo "Stream"** : che si occupano di cifrare bit per bit le informazioni
- 2) **Cfrari a Blocchi** : che dividono il messaggio in blocchi di *k bit* e cifrano blocco per blocco

Concentriamoci sui cfrari a blocchi. Ce ne sono diversi, e la prima cosa da fare è capire "come valuto se un cfrario è robusto?".

plaintext: abcdefghijklmnopqrstuvwxyz  
ciphertext: mnbvcxzasdfghjklpoiuytrewq  
  
E.g.: Plaintext: bob. i love you. alice  
ciphertext: nkn. s gktc wky. mgsbc

n monoalphabetic cyphers,  $M_1, M_2, \dots, M_n$

Cycling pattern:

o e.g., n=4,  $M_1, M_3, M_4, M_3, M_2; M_1, M_3, M_4, M_3, M_2;$

For each new plaintext symbol, use subsequent monoalphabetic pattern in cyclic pattern

o dog: d from  $M_1$ , o from  $M_3$ , g from  $M_4$

La soluzione sta nel **provare a forzarlo**. Fu il caso di **DES (Data Encryption Standard)**, basato su cifratura a blocchi di 64 bit con una chiave stringa da 56 bit. Ci fu una competizione, e con l'algoritmo a forza bruta si riuscì a forzare il cifrario in meno di un giorno.

Nacque così **AES (Advanced Encryption Standard)**, che era ora basato su blocchi di 128 bit cifrati usando chiavi di 128, 192 o 256 bit. Fu dimostrato che se per forzare DES ci voleva 1 giorno, per forzare AES ci sarebbero voluti 149 trilioni di anni.

Ma aldi là di questioni che poi non ci interessano molto, dobbiamo porre il focus su una domanda :

*“Come fanno i due host a scambiarsi la chiave simmetrica?”*

## Lo scambio della chiave

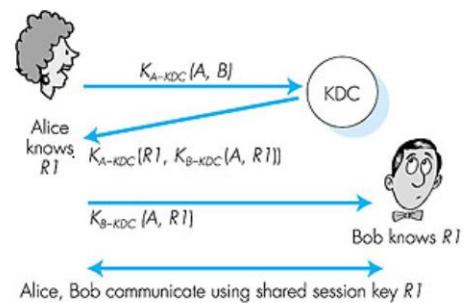
Senza perderci in chiacchere, gli approcci possibili sono 3 :

- **Scambio diretto** : le due persone si incontrano direttamente di persona e si scambiano la chiave. Questo approccio è talvolta il più sicuro, ma non è sempre possibile, perché due persone potrebbero vivere in luoghi davvero troppo distanti per potersi incontrare
- **Scambio utilizzando la Crittografia a Chiave Pubblica** : vedremo dopo che il meccanismo della chiave pubblica permette di comunicare senza scambi di chiavi. Poiché però ha complessità maggiore della comunicazione a chiave simmetrica, si farà in modo di utilizzare questo meccanismo solo per scambiarsi la chiave stessa, e poi passare alla comunicazione basata sulla simmetrica

Un altro approccio, interessante da studiare, è quello che coinvolge delle entità mediatici, dette **KDC (Key Distribution Center)** ai quali gli host possono rivolgersi per agevolare lo scambio della chiave.

Vediamo come funziona :

- 1) Alice e Bob vanno di persona a un KDC fisico e si registrano. Il KDC consegna a loro una chiave che permette la comunicazione in chiave simmetrica col KDC stesso. Chiameremo la chiave Alice - To - KDC  $K_{A-KDC}$ , mentre chiameremo la chiave Bob - To - KDC  $K_{B-KDC}$ .
- 2) Utilizzando una comunicazione a chiave simmetrica, Alice comunica a KDC che vuole parlare con Bob
- 3) Il KDC, utilizzando la stessa comunicazione (e quindi utilizzando la chiave  $K_{A-KDC}$ ) manda ad Alice una chiave di sessione **R1**. Questa sarà la chiave che Alice dovrà utilizzare per parlare con Bob
- 4) Ma come fa Bob a ricevere questo R1? Nel messaggio precedente il KDC invia un messaggio cifrato **utilizzando la chiave di Bob** che quindi Alice non riesce a comprendere. Lei prende questo messaggio e lo spedisce a Bob. Bob, che viceversa questo messaggio può leggerlo, scopre che è un messaggio del KDC che dice “per parlare con Alice usa R1”
- 5) Ora Alice e Bob possono comunicare usando la chiave simmetrica di sessione R1

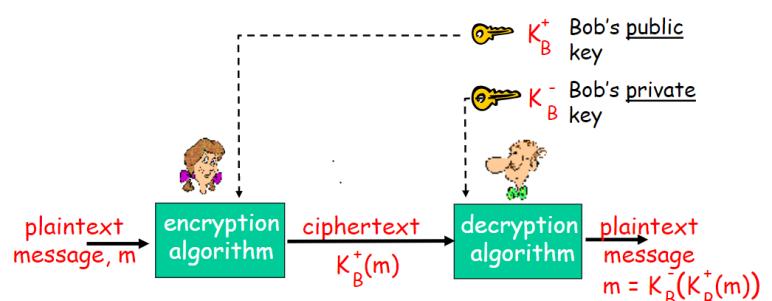


## Crittografia a Chiave Pubblica

L'idea della chiave pubblica risolve in generale il problema dello scambio della chiave, perché permette una comunicazione sicura senza alcuno scambio. In particolare **ogni host ha una coppia di chiavi** :

- 1) Una **chiave pubblica**, che è nota a tutti, anche agli intrusi, in generale a tutto il mondo
- 2) Una **chiave privata**, che è nota solo all'host

Quando Alice vuole mandare un messaggio  $m$  a Bob, applica come chiave di cifratura di un algoritmo eventualmente pubblico la chiave



pubblica di Bob. Dopodiché Bob decifra il messaggio utilizzando invece la sua chiave privata.

La coppia di chiavi è pensata per essere applicata in qualsiasi ordine per criptare / decriptare un messaggio.

Anche se ora riusciamo ad apprezzare solo la prima proprietà, capiremo l'utilità della seconda più in seguito (si pensi a cosa vuol dire che si applica prima la chiave privata di Bob e poi si utilizza la chiave pubblica dello stesso Bob per rivelare il vero messaggio ... forse autenticazione?)

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

use public key  
first, followed  
by private key

use private key  
first, followed  
by public key

In generale il meccanismo della chiave pubblica potrebbe essere utilizzato per comunicarsi qualsiasi cosa, ma ha dei problemi :

- **Chiunque** può mandare messaggi criptati a Bob pretendendo di essere Alice, perché sia Alice che gli intrusi utilizzerebbero la stessa chiave pubblica di Bob
- Il sistema è soggetto ad attacchi di tipo chosen - plain - text, ovvero l'attaccante può decidere una frase in chiaro e criptarla nello stesso modo di come Alice criptrebbe la stessa frase. Ciò significa che se Trudy (l'intrusa) è la moglie di Bob, e Alice l'amante, e Alice manda "Ti Amo" a Bob, se Trudy manda sul canale un "Ti Amo" e ne scopre la codifica, quando vedrà la stessa codifica passare sul canale saranno "lacrime amare per il povero Bob!".
- È **computazionalmente più complicato comunicare in questo modo**, per questo motivo di solito si utilizza il meccanismo della chiave pubblica per scambiarsi la chiave simmetrica (detta **chiave di sessione**) e comunicare poi secondo l'altro meccanismo

## Integrità dei Messaggi

Con il termine **Integrità**, ci riferiamo ad alcune proprietà che vogliamo vedere soddisfatte :

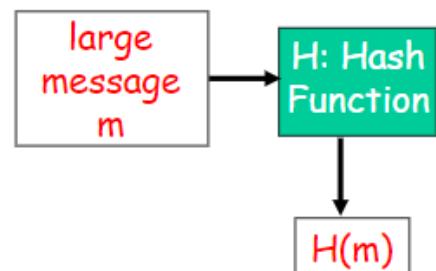
- 1) Vogliamo essere sicuri che il messaggio sia inviato **da chi pensiamo noi**. Se Alice manda un messaggio a Bob, Bob deve essere certo che il messaggio provenga davvero da Alice.
- 2) Vogliamo che il messaggio non sia un **playback** : i packet sniffer possono infatti fare "record - and - playback" e reiterare, ad esempio, transizioni monetarie non desiderate.
- 3) Vogliamo che il messaggio sia a tutti gli effetti **integro**, ovvero che non sia alterato da Trudy durante la trasmissione
- 4) Vogliamo anche che la sequenza dei messaggi sia la stessa che ha mandato Alice, e non una sequenza alterata da un intruso

## Il Digest, la "firma del messaggio"

Poiché i messaggi possono essere arbitrariamente grandi, è utile introdurre il concetto di **digest**, ovvero di una stringa di lunghezza fissata che in qualche modo dipende dal messaggio.

Quello che vorremmo è una **funzione Hash** che, preso in input un grosso messaggio m restituisca una stringa di lunghezza fissata H(m). Ovviamente il meccanismo non funziona o si rende inutile computazionalmente parlando se ci aspettiamo che le cose vadano lisce come l'olio. Ci sono proprietà che vogliamo H soddisfi :

- **Deve essere una funzione da calcolare.** Non vogliamo che garantire l'integrità del messaggio richieda troppe risorse computazionali.



- **Irreversibile** : dato un digest  $H(m)$ , non bisogna essere in grado di risalire al messaggio originale  $m$ . In questo senso vogliamo che  $H(m)$  appaia come un “output casuale”, che non ha alcun riferimento ad  $m$ .
- **Resistenza alle Collisioni** : vogliamo che sia computazionalmente difficile trovare un messaggio  $m'$  diverso da  $m$  tale che risulti  $H(m) \neq H(m')$ . Non possiamo rendere questa situazione *impossibile* poiché la funzione Hash è **many - to - one**, le collisioni esistono. Quello che ci aspettiamo è che per Trudy sia davvero difficile creare un messaggio  $m'$  tale che la firma risulti uguale a quella che ha sniffato.

Un primo esempio di funzione Hash che sembra possa funzionare è il checksum.

Il checksum infatti produce un risultato **su un numero fissato di bit**, ed è ovviamente many - to - one.

Il motivo per cui non è un buon modo di creare un digest è che è troppo facile creare due messaggi diversi che abbiano appunto lo stesso digest (come si vede qui a fianco).

<u>message</u>	<u>ASCII format</u>	<u>message</u>	<u>ASCII format</u>
I O U 1	49 4F 55 31	I O U 9	49 4F 55 31
0 0 . 9	30 30 2E 39	0 0 . 1	30 30 2E 39
9 B O B	39 42 D2 42	9 B O B	39 42 D2 42
B2 C1 D2 AC		B2 C1 D2 AC	

different messages  
but identical checksums!

Funzioni Hash attualmente largamente utilizzate sono la **MD5** (che produce un digest su 128 bit) e la **SHA-1** (che produce un digest su 160 bit).

### Una prima soluzione di integrità : il Message Authentication Code (MAC)

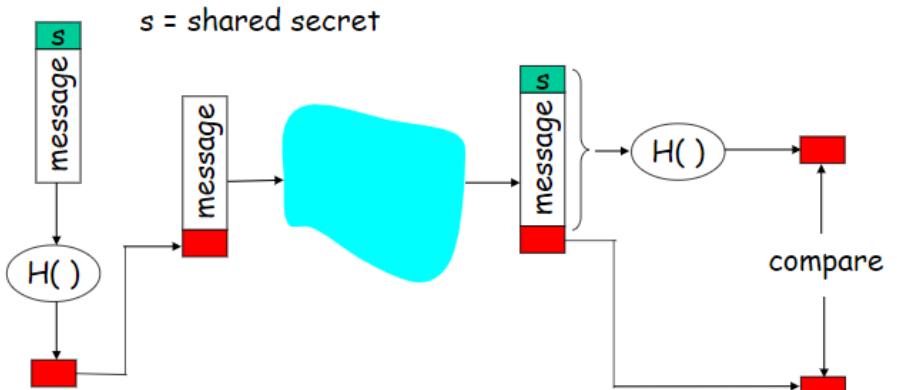
Mettiamo sotto il presupposto in cui Alice e Bob condividono un **segreto** (diremo da ora segreto condiviso s). Sia Alice e Bob conoscono la stessa funzione  $H$  (anzi, supponiamo che questa sia nota a tutto il mondo, Trudy inclusa).

L'idea è di creare un **MAC**, una sorta di “firma” che possa portare a due risultati :

- Bob può concludere che è proprio Alice a mandargli il messaggio
- Bob può dare per certo che il messaggio arrivi integro, ovvero che non sia replicato e/o alterato

Il meccanismo diventa ora semplice :

- Alice prepara il messaggio  $m$  e calcola un digest risolvendo  $H(s | m)$ . Dopotiché invia il messaggio senza codice segreto. Invia dunque  $[m | H(s | m)]$ .
- Quando Bob riceve il messaggio, lui conosce  $s$ , e dunque può calcolare a sua volta  $H(s | m)$
- Se il digest da lui calcolato è uguale a quello che gli è arrivato, può concludere che m non si è alterato, perché è arrivato allo stesso risultato di Alice, e che il mittente sia proprio Alice, perché “solo lei conosce il segreto condiviso  $s$ ”.
- Chiamiamo il digest  $H(s | m)$  **MAC, Mac Authentication Code**.



Una versione più moderna di generazione del MAC utilizzerebbe un “doppio hashing”, e consisterebbe in  $H(H(s | m) | m)$ .

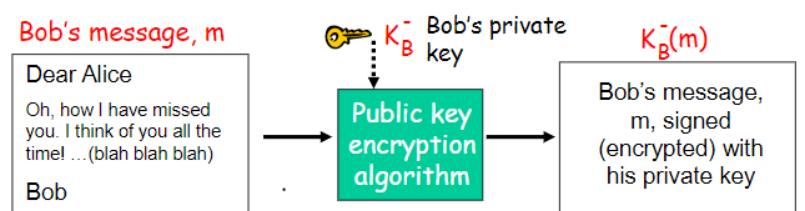
Un vantaggio enorme di questo sistema è che non richiede crittografia e dunque è possibile tenere il concetto di riservatezza e integrità totalmente separati. È buono perché Alice e Bob potrebbero fregarsene che qualcuno legga le loro cose, ma gli importa solo dell'integrità della comunicazione.

Vediamo ora di interpretare alcune caratteristiche del MAC . Mettiamoci nel caso in cui Bob vuole inviare un messaggio  $m$  ad Alice, e vuole certificare che lui ha firmato il messaggio  $m$ , **che è stato lui e che ha firmato esattamente quel messaggio**. Vedremo fra poco che queste sono proprie le caratteristiche di una firma digitale e che il MAC non si presta affatto bene a questo concetto :

- Può Bob dimostrare che lui ha "firmato" il messaggio M e non il messaggio M' ? Sì, perché il messaggio M' non restituirebbe H(s | m) essendo H resistente alle collisioni.
  - Può Alice **verificare** che è stato Bob a firmare quel documento, lui e nessun altro? No, perché il segreto è condiviso ed esistono almeno 2 persone che possono generare quella firma
  - Può Bob **dimostrare che la sua firma è stata falsificata?** No, perché, come sopra, almeno due persone possono generare la stessa e identica firma senza alcuna ambiguità
  - Può Alice **dimostrare che Bob ha firmato M e non M'?** No, perché potrebbe essere stata la stessa Alice a firmare M' e apparirebbe come Bob!

## Firma digitale utilizzando la Chiave Pubblica

Cerchiamo di capire se la crittografia a chiave pubblica può esserci d'aiuto nell'implementare una firma digitale. Come accennato prima, una firma digitale deve assicurare 4 parametri :



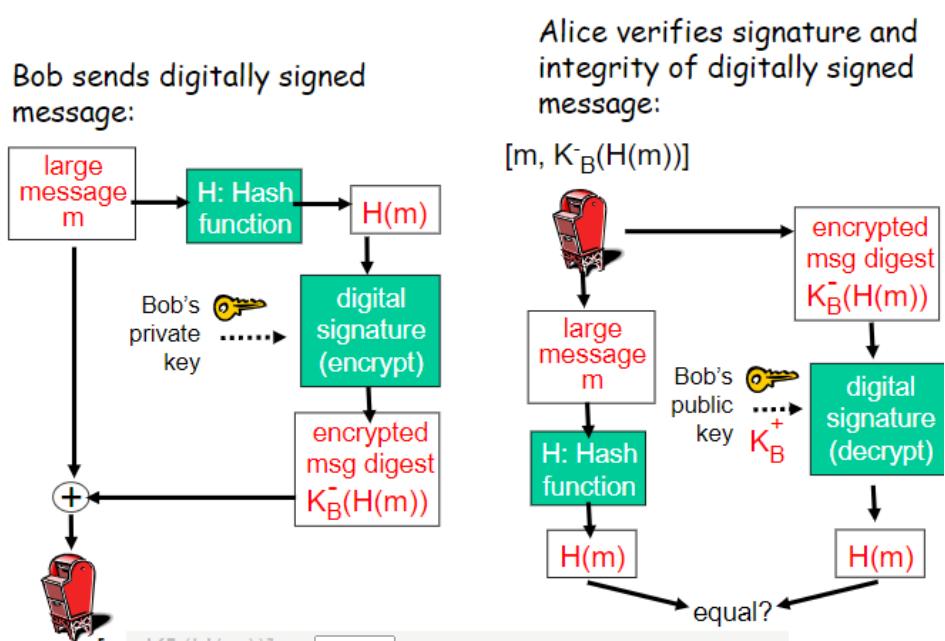
- 1) **Integrità del Messaggio**
  - 2) **Verificabilità** : si può verificare a chi appartiene la firma
  - 3) **Non Falsificabile** : non deve essere possibile per nessuno firmare al posto di qualcun altro
  - 4) **Non Ripudiabile** : si può provare che il messaggio firmato è M e non M'

L'idea che è venuta in mente è quella di cifrare il messaggio con la propria chiave privata. Quando Alice riceve il messaggio, applica la chiave pubblica di Bob. Valendo la proprietà  $\text{ChiavePubblica}(\text{ChiavePrivata}(m)) = m$ , se Bob invia  $[m \mid \text{ChiavePrivata}(m)]$ , allora Alice può assicurare l'integrità del messaggio applicando la chiave pubblica e confrontando gli  $m$ . In particolare Alice riceve  $[m \mid \text{ChiavePrivata}(m')] :$

- Alice applica la chiave pubblica di Bob su ChiavePrivata( $m'$ ). Si vede dunque restituita  $m'$ . Se  $m' = m$ , allora sicuramente quel messaggio è stato inviato da chi conosceva la chiave privata di Bob, ovvero da Bob Stesso.
  - Nessuno può falsificare la firma di Bob, perché per farlo dovrebbe conoscere la sua chiave privata
  - Alice non può dimostrare che Bob ha firmato  $m'$  e non  $m$ , perché a Bob basta applicare la sua chiave pubblica su ChiavePrivata( $m$ ), rivelando che quello che aveva firmato era proprio  $m$ , e non  $m'$ .
  - Viceversa, Bob non può tirarsi indietro da un messaggio che ha firmato, perché Alice può fare la stessa cosa di sopra, rivelando che il messaggio che ha firmato è  $m$  e non  $m'$ .

Visto che crittografare l'intero messaggio m può essere oneroso, si cerca di far entrare il concetto di digest all'interno di questo meccanismo :

- 1) Quando Bob vuole mandare un messaggio firmato digitalmente, performa  $H(m)$  e applica la sua chiave privata su questo digest. La sua firma diventa dunque ChiavePrivata( $H(m)$ ). A questo punto concatena e



manda  $[m, \text{ChiavePrivata}(H(m))]$ . In questo modo la crittografia entra in scena solo su un digest di lunghezza fissata

- 2) Quando alice vuole verificare la firma digitale di un messaggio che le arriva, separa ciò che le è arrivato e applica  $H(m)$  sul messaggio  $m$ . Dopodiché applica la chiave pubblica di Bob su  $\text{ChiavePrivata}(H(m))$  e si vede ritornare un digest che dovrebbe essere appunto  $H(m)$ . Se i due digest così calcolati coincidono, la verifica è stata effettuata correttamente.

Il sistema richiede crittografia, ed è dunque un metodo più pesante computazionalmente parlando. Tuttavia ci sono casi in cui basta l'integrità del messaggio, e dunque basta utilizzare MAC, e casi in cui c'è necessità di firma digitale, e dunque si usa questo meccanismo.

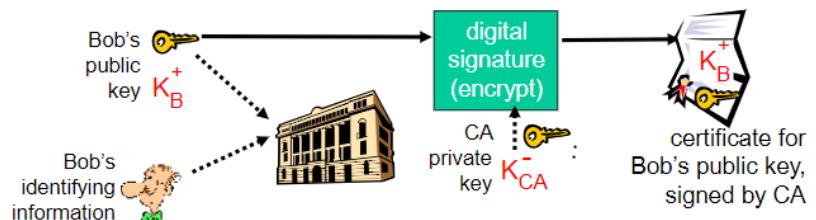
Il sistema, così descritto, ha una falla. Stiamo infatti presupponendo che Alice conosca a priori la Chiave Pubblica di Bob.

Se Alice non conoscesse la chiave pubblica di Bob, e dovrebbe chiederla in giro, allora Trudy si può intromettere. In particolare Trudy può firmare un messaggio utilizzando la sua chiave privata, e spedire la sua chiave pubblica, ma spacciandosi per Bob. Alice utilizzerebbe la chiave pubblica Trudy per confermare l'identità di qualcosa che è stato firmato da Trudy stessa, ma che si sta spacciando per Bob.

In conclusione, Alice confermerebbe che il messaggio è di Bob, ma in realtà gliel'ha mandato Trudy, e la falla è derivata dal fatto che non è riuscita ad ottenere la giusta chiave pubblica di Bob.

## La giusta Chiave Pubblica : I Certificati

Il problema nasce dal bisogno di ricevere la chiave pubblica da qualcuno di *fidato*. La soluzione coinvolge un **ente centralizzato, CA (Certification Authority)** in cui le persone vanno a registrarsi ottenendo una chiave pubblica.



Dopodiché quando Alice ha bisogno della chiave pubblica di Bob, richiede un **Certificato**, ovvero un messaggio criptato con la chiave privata della certification authority (firmato quindi da questa!). Dopodiché Alice deve "trovare la chiave pubblica nel certificato", e per farlo utilizza la chiave pubblica della CA.

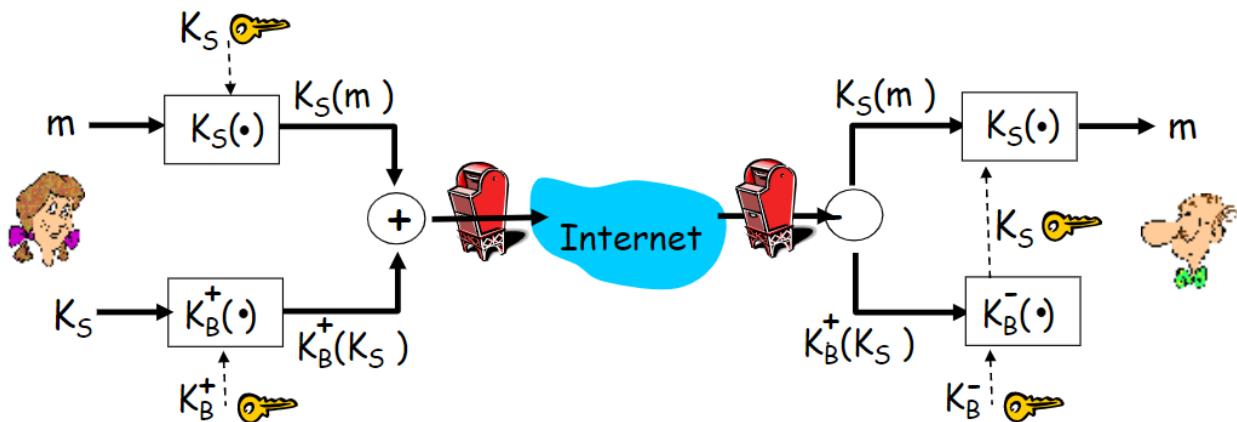
Ma così diventa un loop? No, bisogna semplicemente assumere che la chiave pubblica della CA sia nota a tutti (website fidati, oppure è conosciuta a priori).

È importante il concetto dei certificati nel problema di **identificazione end - to - end**. Se Alice volesse essere sicura di parlare con Bob e viceversa, questi si scambiano una chiave di sessione (simmetrica) mediante la crittografia a chiave pubblica. Quello che però è essenziale che si mandino i certificati con le loro rispettive chiavi pubbliche, così da evitare che Trudy operi un attacco di tipo **man - in - the - middle**, in cui impersona Bob per Alice o viceversa.

## Implementare la sicurezza in Rete

I discorsi che abbiamo fatto sopra sono tutti molto interessanti, ma resta un quesito : a che livello si mette la sicurezza?

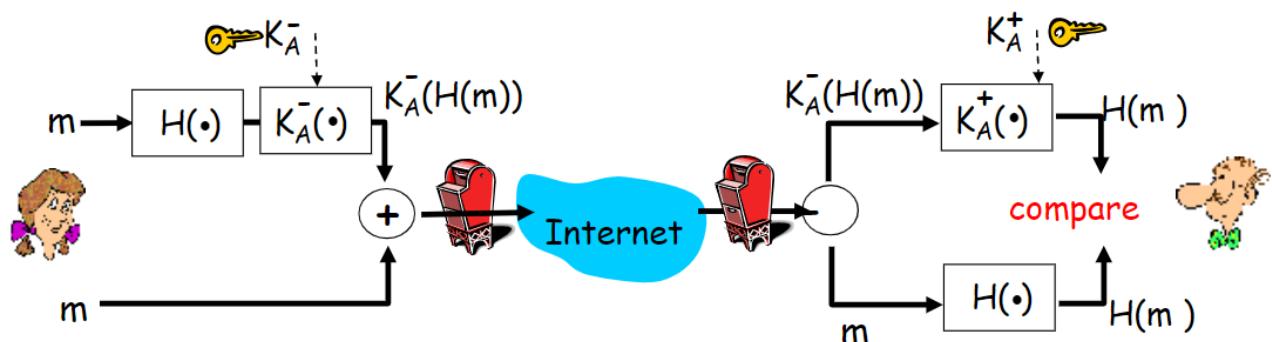
## Sicurezza nel livello Applicazione : Pretty Good Privacy (PGP)



Ipotizziamo di voler rendere sicuro un servizio di scambio di messaggi mediante e-mail. Vogliamo che sia possibile l'autenticazione end – to - end, l'integrità e la confidenzialità del messaggio. Nello schema sopra, vediamo una piccola Alice che vuole mandare un messaggio a Bob :

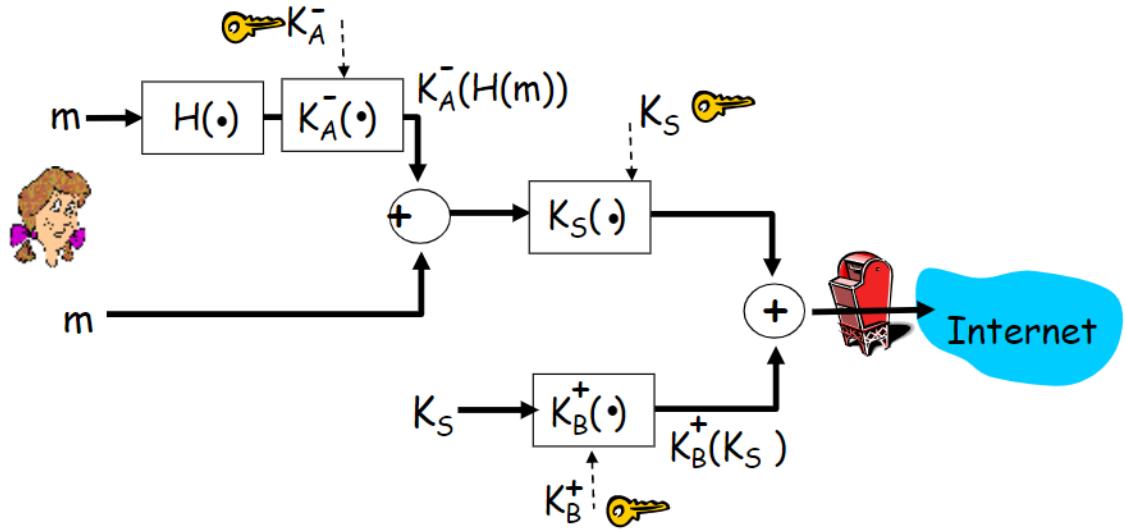
- Alice cripta il messaggio con una chiave simmetrica  $K_S$ . Questo assicura la confidenzialità della comunicazione.
- Alice concatena a questo messaggio criptato la chiave  $K_S$  criptata utilizzando la chiave pubblica di Bob.
- In questo modo Bob può ricevere il messaggio criptato ed è l'unico che (mediante la sua personale **chiave privata**) può scoprire come decriptarlo.

Notiamo già che c'è un problema, ovvero che questo sistema **non assicura alcuna autenticazione di nessun tipo**.



Analizziamo ora questa situazione. Alice vuole sempre mandare un messaggio a Bob, e stavolta quello che vuole ottenere è l'**autenticazione del messaggio**, ovvero vuole far capire a Bob che è proprio lei che sta parlando e che il messaggio è integro.

- Alice prepara il messaggio  $m$  e lo contatena al digest di  $m$  cifrato con la sua chiave privata. A tutti gli effetti, dunque, **Alice pone la sua firma digitale sul messaggio**.
- Bob utilizza la chiave pubblica di Alice per ottenere il digest. Calcola poi il digest sul messaggio ricevuto confronta i risultati ottenuti. Per i discorsi fatti precedentemente, se il confronto è positivo Bob può concludere che il messaggio è di Alice ed è integro



A questo punto è chiaro che dobbiamo semplicemente **concatenare i due meccanismi**. Prima si crea la coppia ( $m$ , firmaDigitale), e poi si cifra il tutto con la chiave simmetrica. Dopodiché si concatena la stessa chiave simmetrica cifrata con la chiave pubblica del destinatario.

In generale **abbiamo bisogno di tre chiavi**:

1. **La chiave Simmetrica**, per garantire la riservatezza del messaggio. Si usa chiave simmetrica e non chiave pubblica per leggerezza computazionale
2. **La chiave Pubblica di Bob**, per permettere lo scambio della chiave simmetrica
3. **La chiave Privata di Alice**, per assicurare la firma digitale

Il meccanismo appena descritto va sotto il nome di **Pretty Good Privacy (PGP)**.

Il problema dell'implementazione della sicurezza a questo livello è che va fatto per ogni applicazione. Se fosse possibile scendere al livello più basso, ad esempio nel TCP, tutte le applicazioni che userebbero il TCP godrebbero della proprietà di sicurezza. Per questo bisogna pensare a qualcosa di simile al PGP, ma che coinvolga **flussi di byte, intere connessioni**, e non si limitino al singolo utilizzo di una singola applicazione.

### Sicurezza a Livello Trasporto : Secure Sockets Layer (SSL)

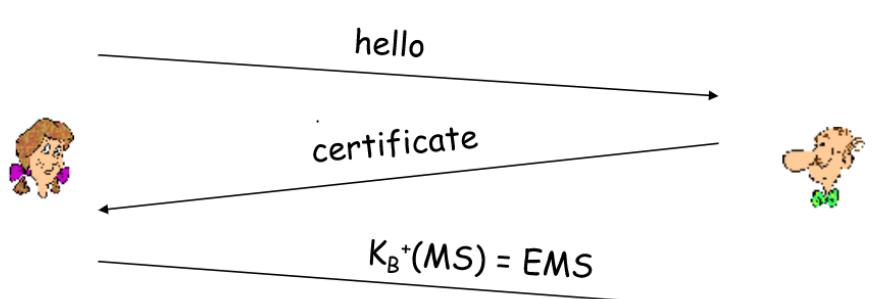
Una possibile implementazione della sicurezza a livello trasporto è fornita dal Secure Sockets Layer, che consiste nel fornire un flusso sicuro di byte mediante lo scambio di un intero set di chiavi che supporti l'intera connessione.

Il protocollo si divide in **quattro fasi**:

- Una fase di Handshake, in cui c'è l'autenticazione della connessione e lo scambio di un *segreto*
- Una fase di Generazione delle Chiavi, in cui si produce un set di chiavi valido per l'intera connessione
- Una fase di Scambio dei messaggi, dove si descrivono le regole che i messaggi SSL devono rispettare (i messaggi di questo protocollo si dicono **record**)
- Una fase di Chiusura della connessione, che è in sostanza un messaggio particolare

### SSL : VERSIONE SEMPLIFICATA

Nella fase di Handshake il client (Alice) manda un messaggio di *inizio connessione* al server (Bob).



Il server risponde **autenticandosi**, ovvero manda un certificato con la propria chiave pubblica.

A questo punto il Client genera un **Master Secret**, e lo *cifra* utilizzando la chiave pubblica appena ottenuta

Nella fase di *Generazione delle Chiavi*, Bob e Alice si scambiano 4 chiavi che sono generate utilizzando il MS, e se lo scambiano mediante la crittografia a chiave pubblica.

Le chiavi che si scambiano sono 4 :

- $K_c$  = encryption key for data sent from client to server
- $M_c$  = MAC key for data sent from client to server
- $K_s$  = encryption key for data sent from server to client
- $M_s$  = MAC key for data sent from server to client

- Due che servono per la riservatezza, e sono le due *chiavi simmetriche* che il client e il server, rispettivamente, utilizzeranno per criptare i messaggi di uscita e decriptare i messaggi di ingresso. In particolare quando il server invia un messaggio lo critta con  $K_s$ , e quando ne riceve uno lo decripta con  $K_c$ , viceversa per il client
- Due servono per l'integrità del messaggio, e costituiscono il **segreto condiviso s** di cui si parlava ai tempi del MAC (e per questo sono indicate come MAC key). Quando il Client manda un messaggio  $m$ , ad esempio, appenderà alla fine  $H(M_c \mid m)$ , mentre quando riceve un messaggio dal server dovrà computare  $H(M_s \mid m)$  e confrontarlo con il MAC ricevuto.

Una volta superata questa fase, si può procedere allo scambio dei messaggi veri e propri, quelli che per SSL, come già detto, prendono il nome di **record**.

La prima domanda che ci poniamo è dove mettere il MAC. Se lo mettiamo solo alla fine, stiamo costringendo le due parti ad aspettare l'intero messaggio prima di poterne verificare l'integrità. Per questo motivo si inserisce il campo MAC **all'interno di ogni Record**.

length	data	MAC
--------	------	-----

Poiché i due devono essere in grado di discriminare cosa è un dato e cosa è il MAC, e per questo viene inserito anche il campo Length all'interno di ogni record.

Notiamo che questo sistema **non è robusto agli attacchi di tipo record and playback**, in cui Trudy rimanda lo stesso messaggio. Se però ad ogni MAC fosse in qualche modo associato un *numero di sequenza ignoto a Trudy*, perché "nascosto dentro il MAC e noto in chiaro solo ai due soggetti", allora questo problema sarebbe risolto.

La soluzione è "semplice" : Alice ad ogni nuova connessione genera un numero di sequenza che parte sempre da 0 e aumenta in maniera incrementale. Anche Bob sa che si inizia sempre da 0. Dopodiché il MAC è ora composto da  $H(M_x \mid \text{n}^{\circ}\text{Sequenza} \mid m)$ . Si noti anche che non esiste scritto da nessuna parte in maniera esplicita il numero di sequenza corretto corrente, e che questo è noto solo a Bob e ad Alice. Una soluzione di questo tipo difende i due, dunque da attacchi di tipo playback.

vers	type	length	data	MAC
------	------	--------	------	-----

Trudy può ancora performare un attacco di tipo **Truncation**, ovvero può costringere Bob a chiudere la connessione. Per evitare questo problema si aggiunge un campo **Type** (0 per la chiusura della connessione 1 per il semplice trasferimento dati).

Il protocollo ha però dei buchi, analizziamo dunque il procollo reale.

### SSL : VERSIONE REALE

Quello che più cambia nella versione reale rispetto alla semplificata proposta sopra è la fase di Handshaking. Notiamo infatti che manca una **negoziazione degli algoritmi di crittografia**. Sia il Client che il Server potrebbero decidere di utilizzare uno o l'altro algoritmo, ma in ogni caso c'è bisogno che i due utilizzino lo stesso medesimo algoritmo! Gli algoritmi sui quali si devono mettere d'accordo sono tre :

- L'algoritmo di chiave pubblica (normalmente RSA)
- L'algoritmo di crittografia a chiave simmetrica (ce ne sono diversi)
- L'algoritmo di generazione del MAC (in sostanza devono comunicarsi la funzione Hash)

Durante la fase di Handshake il client manda una lista di algoritmi da lui supportati, e il server è costretto a scegliere uno solo tra quelli indicati!

E' dunque necessario introdurre i punti 5 e 6 (integrità della fase di handshake) per evitare attacchi di tipo **tampering**, in cui Trudy elimina dalla lista di algoritmi forniti dal client quelli più robusti e per lei difficili da rompere.

Nei primi due punti Client e Server si scambiano anche un **nonce**, **Number Used Once**, che generato pseudo - randomicamente permette ai due di difendersi da attacchi di tipo **Connection Replay**, in cui Trudy replica l'intera connessione sniffata precedentemente. Notiamo infatti che se non ci fossero dei nonce :

- I messaggi di Trudy supererebbero il controllo di integrità perché manderebbe l'intera sequenza di messaggi (non dovrebbe preoccuparsi dei numeri di sequenza sul singolo record)
- I messaggi di Trudy sarebbero criptati utilizzando le chiavi simmetriche giuste in quanto ha generato lo stesso Master Secret utilizzato da Alice e sta comunicando utilizzando le chiavi utilizzate da Alice il giorno prima

Con l'introduzione dei nonce invece i **messaggi di Trudy falliscono il controllo di integrità da parte di Bob**, perché le 4 chiavi simmetriche vengono generate sulla base di MS | nonceServer | nonceClient e dunque saranno diverse da quelle usate il giorno prima. Se Trudy emula i messaggi vecchi, lo starà facendo con chiavi sbagliate, e quando Bob andrà a verificare l'integrità con le chiavi giuste dovrà dichiarare inaffidabile la comunicazione!

### Sicurezza a Livello Network : Ipsec e VPN

Implementare la sicurezza a livello rete permette di implementare le tanto citate **VPN** : **Virtual Private Network**.

Le VPN permettono alle organizzazioni di implementare una propria "rete privata". Questo sarebbe altresì complesso e costoso, perché richiederebbe l'aggiunta di router e link dedicati alla sola organizzazione (davvero troppo costoso).

Se un'organizzazione è in un certo istante "sparsa per il mondo" (perché magari ci sono dei rappresentanti in Hotel di diverse zone



del continente), si può creare qualcosa che dia l'impressione di rete privata?

La risposta è sì, e consiste nell'utilizzare i meccanismi di crittografia all'interno dell'Internet pubblica. Il tutto è realizzato mediante il protocollo **IPsec**, che critta un datagram fino alla destinazione finale.

Il meccanismo di criptazione non deve essere necessariamente noto agli host finali, ma può essere implementato sui due router di frontiera di sorgente e destinazione. Nell'esempio del rappresentante in Hotel, il quartier generale comunica di voler comunicare con lui, il router di frontiera critta il datagram e lo spedisce nella internet pubblica. Quando il destinatario riceve il pacchetto si rende conto che deve decriptarlo in qualche modo (**esempio di Ipsec implementato sulla coppia Router | HostDestinatario**).

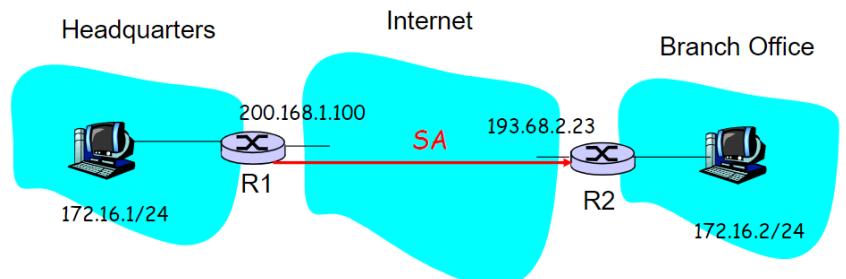
Ci sono due protocolli che implementano questo meccanismo :

- **Autentication Header (AH)** che assicura l'autenticazione dei dati e l'integrità dei messaggi
- **Encapsulation Security Protocol (ESP)** che assicura, oltre che come sopra, anche la riservatezza della comunicazione

Ci concentreremo sull'ESP nel seguito della trattazione.

Immaginiamo che il meccanismo Ipsece sia implementato sui due router di cui a lato R1 e R2.

I due router devono stabilire una o più **Security Associations SA** che è un meccanismo *simplex* (ovvero funziona solo in una direzione, e ne servono due se si vuole anche il viceversa).



Mettiamoci nel caso che ci sia una SA tra R1 ed R2. I due router devono memorizzare delle informazioni :

- Un identificatore a 32 bit detto SPI : Security Parameter Index. Questo sarà l'indice che useranno per sfogliare una tabella interna e ricavare tutte le seguenti informazioni
- Origine della SA (R1 nell'esempio)
- Destinazione della SA (R2 nell'esempio)
- Tipo di algoritmo crittografico utilizzato per la riservatezza della comunicazione
- Chiave simmetrica per criptare / decriptare i dati
- Tipo di algoritmo di integrità (ad esempio HMAC con funzione hash MD5)
- Chiave di autenticazione (il segreto condiviso dell'algoritmo di integrità)

Queste informazioni, come già accennato, sono memorizzate in un Database interno ai router detto **SAD**, **Security Associations Database**. Quando R1 deve comunicare con R2, guarda il Database per decidere come processare il datagram. Viceversa quando R2 riceve un messaggio da R1 guarda il campo SPI per accedere al database e processare il datagram di conseguenza.

Qualcuno deve però dire a R1 cosa fare, e questo è compito del **mittente originale**. La politica che si utilizza è la **SPD, Security Policy Database**, secondo la quale il mittente deve indicare, tra le altre cose :

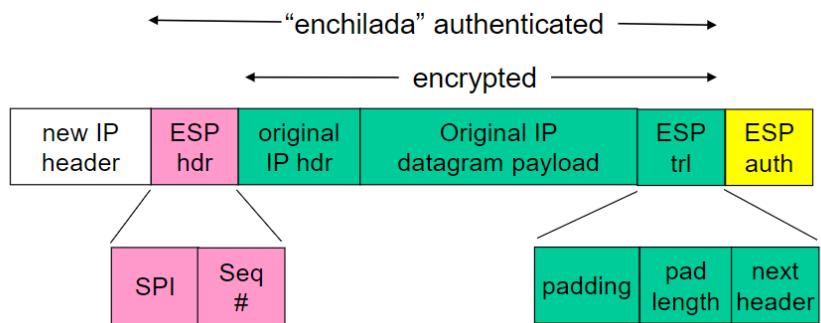
1. Se si intende utilizzare Ipsec
2. Quale SA intende utilizzare

Dunque SPD fa sì che il router sappia "cosa fare" (in termini di quale SA guardare), mentre SAD fa sì che il router sappia "come fare".

Scendiamo ora nel livello di dettaglio e analizziamo come funzionano i campi del "datagram Ipsec" :

La prima cosa da fare è aggiungere **in coda al datagram originale** un ESP trailer che contenga un eventuale *padding* (e se si deve indicare anche quanto è lungo) perché gli algoritmi di cifratura spesso ragionano *a blocchi* e quindi si richiede che la roba da criptare sia multiplo di un certo valore.

Si critta poi la coppia (Datagram Originale + ESP Trailer) in accordo all'algoritmo di cifratura trovato nel SAD.



Si mette davanti poi un **ESP Header**, che deve contenere :

- Lo SPI così che R2 possa capire cosa fare (R1 riceve questa informazione tramite SPD)
- Un **numero di sequenza** che viene inizializzato a 0 per ogni SA e incrementato ad ogni comunicazione che coinvolga la SA. Questa è una misura di sicurezza che protegge da attacchi di tipo replay

La struttura ottenuta, detta "**enchilada**" viene poi autenticata e si pone il MAC alla fine dell'intera sequenza. Il MAC, anche stavolta, è generato utilizzando la chiave di autenticazione ottenuta guardando il SAD. Viene comunque creato un IP Header (IPv4) in testa a tutto così che il datagram possa viaggiare correttamente sulla rete.

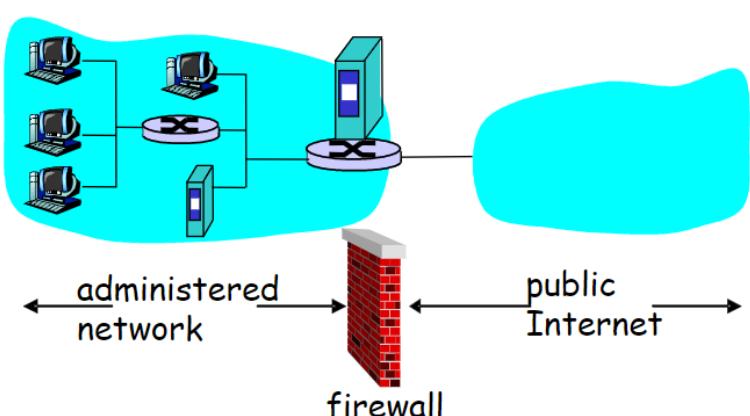
## Le mura della sottorete : i Firewall e gli IDS

Presi coscienza del fatto che la rete non è un posto sicuro e che le persone cattive esistono, abbiamo analizzato come possiamo rendere le nostre comunicazioni "sicure", ovvero assicurarci che nessuno le legga e che stiamo parlando con chi ci aspettiamo di parlare.

**Ma i dati che stiamo spedendo o che stiamo ricevendo sono sicuri?** Chi ci assicura di non ricevere un messaggio *malevolo*? Supposti di essere gli amministratori di una sottorete, come tuteliamo i nostri Host a non ricevere danni dall'esterno e, non meno importante, a non mandare danni all'esterno?

La soluzione è recintare la sottorete con un muro. Ovviamente il muro deve essere dotato di porte attraversabili secondo certe condizioni (perché altrimenti si avrebbe l'isolamento totale e non ha molto senso).

Questa è l'idea di base dei **firewall** : introdurre politiche e meccanismi secondo le quali operare una scelta : faccio passare un pacchetto in ingresso / uscita o non lo faccio



passare?

Un meccanismo del genere avrebbe sicuramente dei vantaggi non indifferenti :

- Un firewall potrebbe fare detection di un attacco **DoS** e bloccare le richieste al server da parte di una certa sottorete (botnet)
- Potrebbe evitare che **dall'interno** si mandino dati falsi all'esterno (se ci fosse una spia nella CIA e una persona si connettesse alla loro homepage, questa spia potrebbe trasmettere qualcosa di molto diverso e dannoso)
- Un firewall può implementare anche un set di utenti autenticati, e fare in modo che siano gli unici a poter addentrarsi in una sottorete

Esistono principalmente **tre tipi di Firewall** :

1. I firewall a filtraggio di pacchetto stateless
2. I firewall a filtraggio di pacchetto stateful
3. Gli **application gateway**

### I **firewall a filtraggio di Pacchetto** e gli **Application Gateway**

#### STATELESS

Questo tipo di Firewall sono pensati per prendere una scelta sul singolo pacchetto, indipendentemente dagli altri. La scelta può essere presa su diversi parametri :

- Source and Destination IP address
- TCP/UDP Source and Destination Port
- Tipo del messaggio ICMP
- Pacchetti che trasportano un segmento con il bit di SYN o il bit di ACK significativo

Riportiamo nel seguito alcune regole intriganti che si possono implementare sfruttando uno o più parametri tra quelli sopra indicati :

<u>Policy</u>	<u>Firewall Setting</u>
No outside Web access.	Drop all outgoing packets to any IP address, port 80
No incoming TCP connections, except those for institution's public Web server only.	Drop all incoming TCP SYN packets to any IP except 130.207.244.203, port 80
Prevent Web-radios from eating up the available bandwidth.	Drop all incoming UDP packets - except DNS and router broadcasts.
Prevent your network from being used for a smurf DoS attack.	Drop all ICMP packets going to a "broadcast" address (eg 130.207.255.255).
Prevent your network from being tracerouted	Drop all outgoing ICMP TTL expired traffic

Queste regole si implementano facendo in modo che il router consulti una **ACL (Access Control List)**. Questa lista, che possiamo vedere come una tabella, descrive le coppie (**Azione | Condizione**) che sono

analizzate dal router dall'alto verso il basso, e viene scelta semplicemente **sempre la prima che matcha la condizione, e basta!**

Tutte le ACL devono avere una **regola di default**, ovvero qualcosa che si applichi a tutti i pacchetti che non rispettano le condizioni sopra di essa. Questa regola di default può essere :

- **DENY ALL** : è un approccio molto restrittivo e molto sicuro, e costringe l'amministratore a indicare **tutte le cose consentite**, aumentando il rischio di dimenticarsi qualcosa di "consentito" e bloccare troppe cose (**Firewall Inclusivo**)
- **ALLOW ALL** : è un approccio molto liberale e poco sicuro e costringe l'amministratore a indicare **tutte le cose vietate**, aumentando il rischio di dimenticarsi qualcosa di "dannoso" e non bloccare abbastanza (**Firewall Esclusivo**)

action	source address	dest address	protocol	source port	dest port	flag bit
allow	222.22/16	outside of 222.22/16	TCP	> 1023	80	any
allow	outside of 222.22/16	222.22/16	TCP	80	> 1023	ACK
allow	222.22/16	outside of 222.22/16	UDP	> 1023	53	---
allow	outside of 222.22/16	222.22/16	UDP	53	> 1023	----
deny	all	all	all	all	all	all

#### STAFEUL

Cosa significa rendere "stateful" un firewall? Cos'è che dovrebbe ricordare? Nell'approccio stateless qualcuno potrebbe mandare un segmento TCP con bit di ACK settato e, se permesso, passerebbe sempre, **anche quando non c'è una connessione TCP stabilita**. Una prima cosa che possiamo pensare di far ricordare è dunque se è stata creata una connessione TCP prima dell'arrivo di quel messaggio, altrimenti rischiamo di accettare cose che semplicemente "**non hanno senso**".

Nei Firewall staful si aggiunge un campo alla tabella che determini "quando c'è bisogno che la connessione sia attiva" (connection flag), e questo risultato viene raggiunto tenendo traccia dei SYN e FIN trasmessi. Nel caso di timeout raggiunti e nessun FIN rilevato, il firewall può decidere di comportarsi rifiutare tutti i dati!

Gli application Gateway sono dei "server speciali" che implementano la scelta "PASSA / NON PASSA" decidendo in base all'**applicazione che si sta utilizzando**, piuttosto che sui parametri di intestazione dei protocolli TCP/UDP/IP.

Si ragiona nello stesso modo di un Proxy Server : quando un Host vuole utilizzare un'applicazione, redirige invece la richiesta sull'application gateway. **Se l'utente può utilizzare l'applicativo**, sarà il gateway a fare richieste al posto suo, altrimenti la richiesta sarà rigettata.

A differenza dei Firewall, questo è l'unico sistema che permette di discriminare per utenti, e non per pacchetti!

Il problema del Gateway è che **ne serve uno per ogni applicazione che richiede questo "trattamento speciale"**, e che chi vuole comunicare ha bisogno di sapere come contattare il gateway.

Un limite di tutti i meccanismi sopracitati è che non sono in grado, da soli, di riconoscere **IP spoofing**. Un attaccante, sapendo che il suo IP è bloccato dalle ACL, potrebbe inserire un campo diverso nell'IP source address e risolvere la questione!

#### I paladini della giustizia : Intrusion Detection System (IDS)

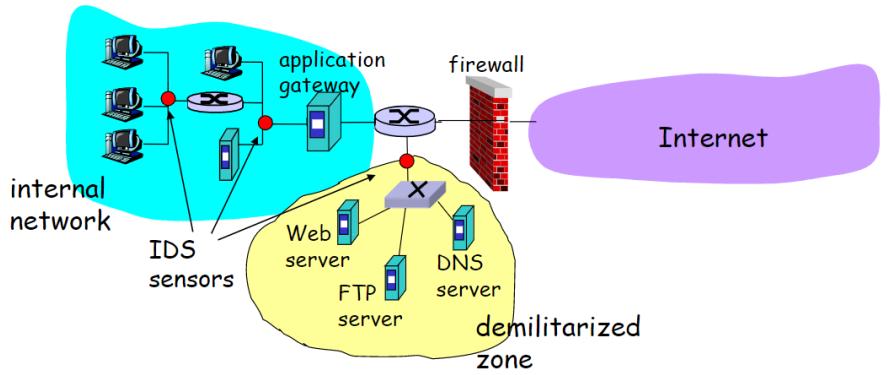
Per risolvere i limiti di Firewall e Application Gateway sono stati introdotti dei meccanismi più intelligenti che fossero capaci di due operazioni significative :

- **Deep Packet Inspection** : analizza anche il contenuto dati del pacchetto, così da rivelare se sono presenti alcune stringhe malevoli che potrebbero danneggiarre l'host destinatario
- **Examine Packet Relation** : invece di analizzare pacchetto per pacchetto, si cerca una correlazione (ad esempio stesso IP source address) per evitare, ad esempio, che ci siano attacchi di tipo DoS a un server

## LO SCENARIO FINALE

Combinando, infine, i meccanismi di Firewall (stateless and stateful) e i meccanismi IDS. Si creano all'interno di internet delle zone che sono chiamate :

- **Delimitarizzate**, dove si predilige la velocità di comunicazione (e dunque è inserito un solo IDS come nell'immagine in figura)
- **Internal Secured Network** dove i controlli si fanno più frequenti e c'è un numero di IDS considerevole, così da bloccare pericoli sia "all'interno" che "all'esterno".



## CAPITOLO 9 : Reti Wireless e Mobili

Oggi le reti non si presentano all'utente finale come abbiamo detto finora. Siamo pieni di dispositivi IoT, laptop, smartphone, che hanno bisogno della connettività, ma che non si possono collegare fisicamente ad alcun dispositivo del network. Questo può essere dovuto a limiti del dispositivo che si vuole connettere, o al fatto che il dispositivo è in continuo movimento.

La sfida è dunque creare un internet **onnipresente**, e questo richiede il superamento di due diverse sfide :

- **Wireless** : connettere tutti i dispositivi che lo volessere attraverso un mezzo "wire - less"
- **Mobility** : accettare e risolvere il fatto che un utente potrebbe muoversi durante la sua esperienza di connettività

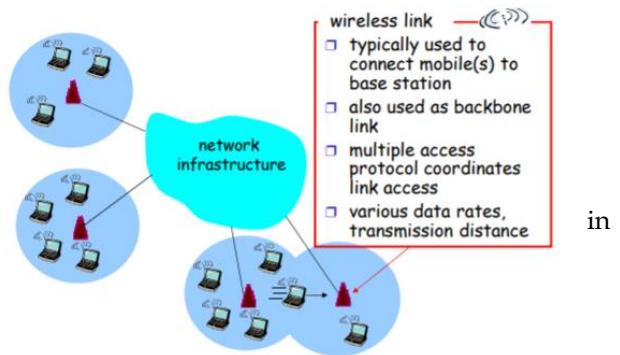
## Le reti Wireless

### Elementi costitutivi di una rete Wireless

Tra gli elementi costitutivi ci sono ovviamente i **wireless Host**, ovvero tutti gli host (o più in generale i processi) che hanno bisogno di connettività senza avere a disposizione un link fisico.

Gli host Wireless sono tipicamente laptop o applicazioni distribuite. Possono essere o meno stazionari. Questo significa che wireless non necessariamente implica mobility!

Gli host wireless fanno riferimento a una **base Station**, un dispositivo che tipicamente è connesso maniera wired al network ed è responsabile dello scambio di pacchetti tra il network a cui è collegato e gli host wireless che stanno nella sua area.



Per connettersi, gli host usano dei **wireless Link**, che tipicamente consistono in un'antenna che hanno al loro interno in grado di propagare il segnale fino alla base station. Notiamo già che c'è sicuramente bisogno di un multiple access protocol per coordinare i vari accessi. I wireless Link hanno caratteristiche differenti dai link wired :

- La potenza del segnale diminuisce man mano che si propaga nello spazio (fenomeno di path loss)

- **Ci sono interferenze con le altre sorgenti**: ci sono delle frequenze standard che sono utilizzate nelle reti wireless, ma queste possono essere utilizzate da device diversi, e dunque si può creare interferenza
- **Il segnale si propaga su più percorsi**: il segnale rimbalza e si riflette sugli oggetti, e arriva a destinazione con tempi variabili

Anche una comunicazione punto - punto in wireless diventa dunque decisamente più complessa!

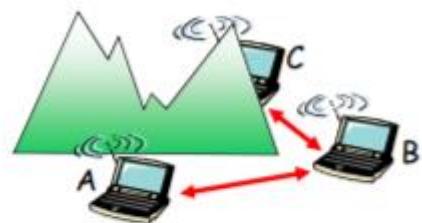
Il problema del **nodo nascosto**, ad esempio, è tipico delle reti wireless, e non può presentarsi nelle reti wired. Questo tipo di problema fa in modo che due host non riescano a rendersi conto di un'interferenza che stanno creando per via del fatto che *"non riescono a sentirsi"*.

Analizziamo un caso in cui A e C sono separati da un'enorme montagna che fa in modo che il segnale di C non arrivi mai ad A e viceversa.

È chiaro che entrambi possono decidere di comunicare con B, che però sicuramente rileverà un'interferenza. Il problema è che A e C non lo sapranno (a meno che non sia B a dirglielo) perché entrambi credono di essere **in due a comunicare** (A - B e C - B).

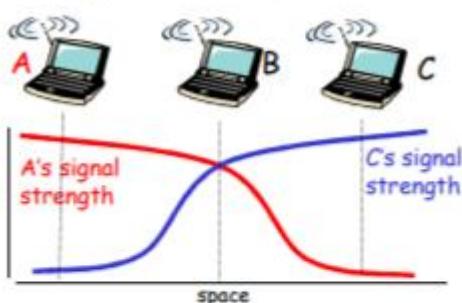
Ma c'è davvero bisogno di una montagna?

Ricordando che il segnale wireless diminuisce in potenza all'aumentare della distanza, in realtà basta che A e C siano abbastanza distanti, affinché i loro segnali appaiano all'altro come rumore, appunto segnale da non considerare come interferenza.



### Hidden terminal problem

- B, A hear each other
- B, C hear each other
- A, C can not hear each other  
means A, C unaware of their interference at B



### Signal attenuation:

- B, A hear each other
- B, C hear each other
- A, C can not hear each other  
interfering at B

## Wireless LAN (Wi - Fi)

Le reti wireless sono reti **con infrastruttura a Basic Service Set (BSS)**. Ogni BSS ha :

- Gli Host Wireless che si vogliono connettere
- Un **access point (AP)** costituito dalla Base Station

La rete Wi - Fi è per natura di tipo infrastructure - based, per via di questa divisione in BSS, ma esiste la modalità **ad hoc** in cui all'interno di un BSS sono presenti solo gli host wireless.

Ogni AP Admin deve decidere una frequenza per il proprio AP scegliendo tra 11 canali diversi in cui lo spettro delle frequenze 2.4GHz - 2.485GHz viene suddiviso.

Un Host Wireless deve essere poi in grado di **associarsi** alla Base Station, rilevando all'interno delle comunicazioni un beacon contenente il nome dell'AP e il suo MAC address. In realtà un Host ne rileva più di uno (fase di **scanning**), ma ne sceglie uno e soltanto uno (con il quale può performare anche un'operazione di autenticazione a seconda che l'AP la preveda o meno).

### Il protocollo MAC sulle reti Wi - Fi : CSMA/CA

All'interno delle Reti Wi - Fi si può utilizzare sicuramente il protocollo **CSMA (Carrier Sense Multiple Access)**, in cui i nodi evitano di trasmettere se non sentono nessuno.

Ma abbiamo già detto che ci sono dei problemi in più nelle reti wireless, e questi si traducono nell'impossibilità di fare detection delle collisioni. Infatti un host wireless :

- **Ha in genere una sola antenna**, e questo significa che non può ascoltare il canale mentre sta trasmettendo (e quindi in generale non può detectare la collisione)
- Esiste il problema del **nodo nascosto**. Anche se l'host avesse più antenne, ci sono interferenze (collisioni) che l'host non riuscirebbe a rilevare a causa del fatto che un Host sta trasmettendo a una distanza considerevole oppure sono separati da un ostacolo che scherma il segnale.

Per questo motivo si è pensato a un altro protocollo, il **CSMA/CA** (CSMA / Congestion Avoidance). In questo protocollo, il sender :

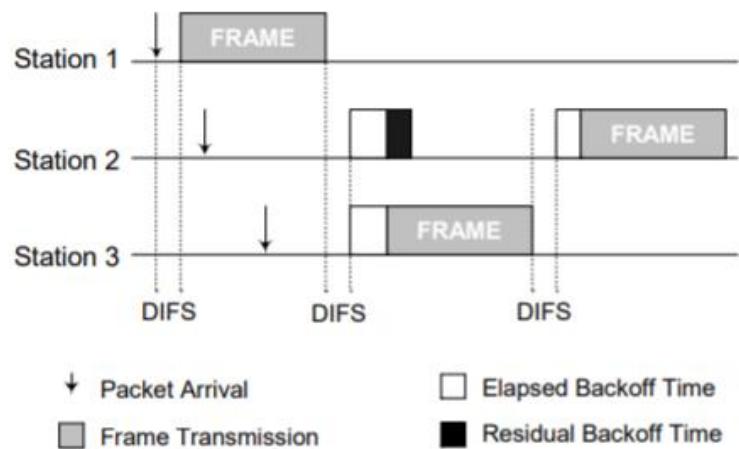
- Ascolta il canale e se lo vede libero per un tempo DIFS, decide di **trasmettere l'intero frame** (senza possibilità di fare collision detection).
- Se lo vede occupato :
  1. Comincia un **tempo casuale di backoff**
  2. Il timer scorre solo quando il canale è libero (tempo DIFS escluso)
  3. Trasmetti quando il timer scade
  4. Se non arriva l'ACK, incrementa il tempo di backoff e ripete dal punto 1

Per quanto riguarda il ricevitore, questo deve limitarsi a mandare un ACK dopo un tempo SIFS.

Gli ACK sono necessari per rilevare errori nel canale e per accorgersi che probabilmente si è perso il frame per la strada.

Nell'esempio a fianco :

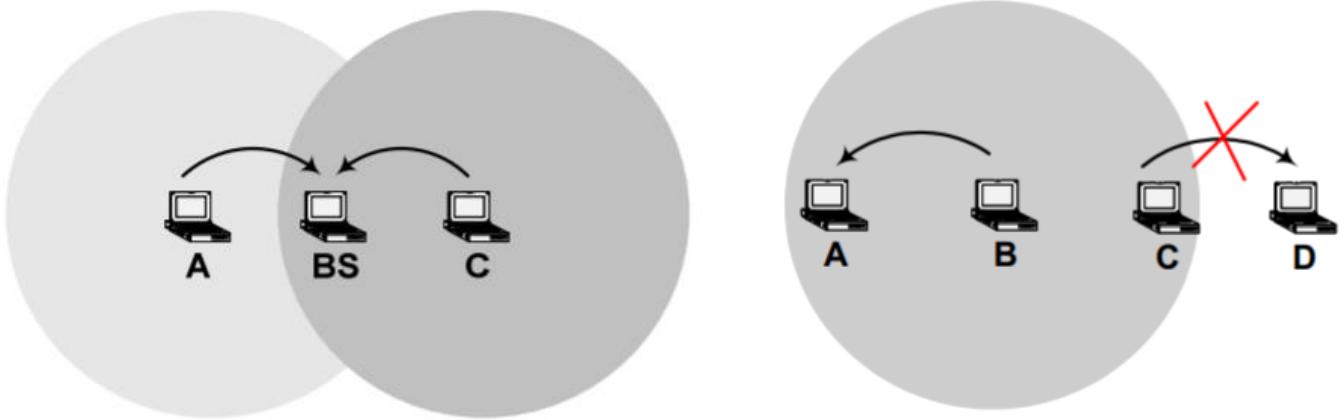
- La prima stazione sente libero per DIFS e trasmette il frame.
- Nel frattempo la stazione 2 è intenzionata a trasmettere, aspetta che la prima abbia finito, aspetta DIFS e fa partire un timer.
- La terza stazione fa la stessa cosa, ma sceglie un timer più corto e trasmette il frame. In questo momento **il timer della stazione 2 si interrompe**
- Quando la stazione 3 finisce, la 2 aspetta DIFS e poi aspetta il tempo residuo di attesa, oltre il quale può trasmettere.



Il protocollo non elimina le collisioni : queste possono continuare ad esistere sia se due stazioni scegliersero lo stesso timer di backoff, sia se due stazioni scelgono una un timer classico che finisce per coincidere con il residuo dell'altra.

Il tempo di backoff è scelto randomicamente nell'intervallo  $[0, CW - 1]$ , con CW che raddoppia ad ogni missed ACK e che sta sempre nell'intervallo  $[CW_{\min}, CW_{\max}]$ , i cui valori dipendono dal physical layer di riferimento.

### PROBLEMA DEL NODO NASCOSTO E DEL NODO ESPOSTO



Il secondo caso (quello di destra), è un problema detto di **nodo esposto**. Se C volesse trasmettere a D mentre B sta trasmettendo ad A teoricamente non succederebbe nulla. Il problema è che per come è fatto il protocollo C si trattiene, e sceglie di non trasmettere.

Nel primo caso (di sinistra) rivediamo il problema del **nodo nascosto** già visto precedentemente. Se A e C vogliono trasmettere a BS, entrambi vedono il channel idle per via del fatto che A non si accorge della trasmissione di C, e viceversa. Quando si arriva in prossimità di BS, tuttavia, la collisione avviene.

È chiaro che il nodo esposto è, se vogliamo, un problema di "ritardo", di "prestazioni che potrebbero essere più performanti". Il vero problema è dunque quello del nodo nascosto.

### C'è modo di risolverlo?

#### **Un nuovo meccanismo : Il Virtual Carrier Sensing**

Ricordando che Carrier Sensing significa "ascoltare il mezzo misurando il livello di potenza", questo meccanismo ha l'etichetta "**virtual**" perché la decisione non si prende in base a una misurazione fisica.

Questo meccanismo permette a una coppia di prenotare il canale per un certo intervallo di tempo, come se si chiedesse "*per favore*" di non trasmettere. In particolare si vorrebbe evitare che i nodi vicini trasmettino **anche se vedono il canale libero** (perché magari lo vedono libero per il problema del nodo nascosto!).

Come posso "prenotare" e "annunciare" la prenotazione?

Il meccanismo è implementato **ampliando il protocollo CSMA/CA**, aggiungendo due frame particolari :

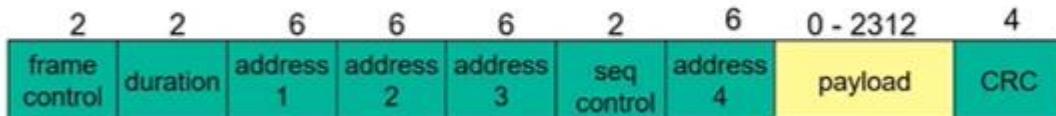
- **RTS : Request - To - Send**, è un frame che viene inviato quando si trova il canale libero per DIFS. RTS è un frame che viene mandato alla base station, ma essendo il mezzo wireless questo viene propagato in broadcast a tutti. Il Frame contiene al suo interno, in campo detto duration, l'intervallo di tempo corrispondente alla **durata della trasmissione** (inteso come il tempo che intercorre dalla fine della trasmissione di RTS alla fine della trasmissione dell'ACK). Questo tempo può essere calcolato dal sorgente (SIFS + CTS + SISF + FRAME + SISF + ACK).
- **CTS : Clear - To - Send**, è l'"ack" che la base station manda a chi ha mandato l'RTS, convalidando la prenotazione del canale. Anche il campo CTS ha un campo duration, che è lo stesso di RTS diminuito di SISF + CTS.

La Base Station manda un CTS, che sicuramente arriva a tutti perché tutti arrivano alla base station. Questo significa che **il problema del nodo nascosto si è risolto**, perché anche se C non vede RTS di A, riceve comunque il CTS!

Quando una stazione vede un RTS o un CTS si parte un timer detto NAV (NAV RTS e NAV CTS rispettivamente) e tutti i nodi seguono la regola di “non trasmettere fino a quando c’è un NAV attivo”.

**La dimensione dell’RTS deve essere decisamente più piccola del Frame**, perché altrimenti è meglio mandare il frame stesso. Se la loro dimensione fosse paragonabile le probabilità di collisione si avvicinerebbero e si rende inutile l’overhead provocato da RTS. Questo rende questo meccanismo opzionale, in base alla dimensione del frame che si vuole mandare.

### Formato del Frame, Mobilità e Power Management



- Il campo **Duration** contiene il tempo (2 byte) per cui il canale deve essere riservato
- I campi **address** (1, 2, 3) rappresentano : indirizzo MAC del destinatario (1), indirizzo MAC del sorgente (2), indirizzo MAC dell’**interfaccia router** alla quale la base station è collegata.
- Il campo **address 4** serve per la modalità ad hoc, in cui la base station non è presente

Compito dell’Access Point è operare una **traduzione di formato**, passando da un frame Wi – Fi a un frame Ethernet (mantiene come source address l’address 2 del frame Wi – Fi, e come destination l’address 3).

L’Host wireless può spostarsi tra le varie BSS, perlomeno allo stato dei fatti se resta all’interno della stessa **sottorete IP**.

Se l’Host cambia BSS, ma entrambi i BSS sono collegati a un link – layer switch, lo switch ricorda qual è la porta per raggiungere Host.

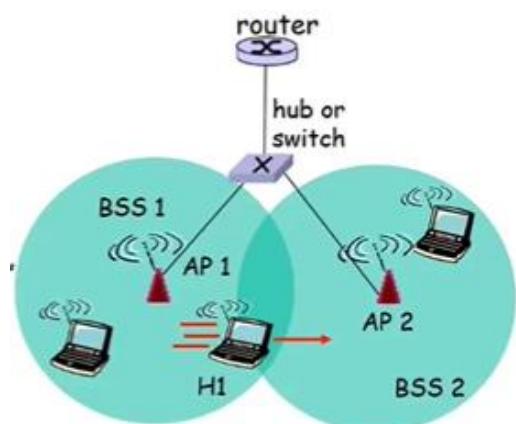
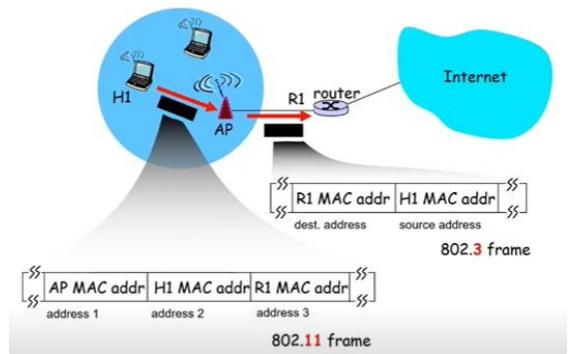
Sembrerebbe essere un problema che lo Switch mandi il frame verso AP1 quando l’host è in realtà andato ora su AP2, ma in realtà non è niente di critico in quanto sappiamo che lo switch periodicamente opera il flushing dei propri record. Dimenticando che l’Host era con AP1, riuscirà a capire, mediante il meccanismo di flooding, che si è spostato!

C’è un ultimo discorso interessante da trattare che è il Power Management. Nel caso delle reti Wired il problema dell’elettricità non è un problema. Ma i nodi che si muovono hanno un’energia limitata, che dipende dalla batteria!

L’idea è fare in modo che l’Host Wireless “dormi”, risparmiando energia. Bisogna introdurre un sistema per “svegliarsi periodicamente”. Questo è possibile grazie all’AP che emette dei Beacon ogni 100ms. Questi di base servono per la sincronizzazione con l’AP, ma se ne fa anche un uso di Power Management : l’Access Point fa da “segretario” al nodo Wireless, bufferizzando ciò che gli altri utenti vogliono inviare.

Nello specifico :

- Un nodo segnala all’AP “vedi che vado a dormire”



- L'AP allora bufferizza i messaggi a lui destinati
- L'AP include nei beacon la lista degli Host con messaggi che non sono stati recapitati, perché i destinatari dormivano.
- Il nodo si sveglia ad ogni beacon : se la lista contiene qualcosa a lui destinato resta sveglio e ne chiede l'invio, altrimenti **torna a dormire**.

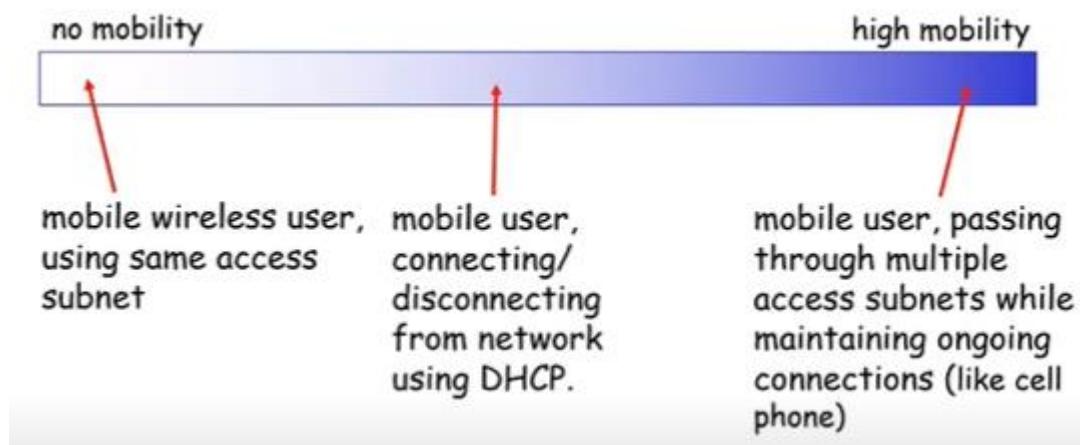
## La Mobilità in Internet

Il Protocollo IP assume, implicitamente, che gli host siano **stazionari**, ovvero che non cambino perlomeno la propria sottorete di appartenenza.

Oggi però le cose sono cambiate, ci sono smartphone e laptop che si muovono continuamente!

Il discorso di assegnazione dell'indirizzo IP sembra risolto con il **DHCP**. Ci sono però dei casi, vedremo, in cui il DHCP non va bene.

### Cos'è la Mobilità dal punto di vista di Internet



IP conclude che **non c'è mobilità** quando gli utenti si collegano sempre dallo stesso punto di accesso. La mobilità nasce quando cambio punto di accesso. Dal punto di vista di IP il punto di accesso non è l'AP di cui abbiamo finora parlato, ma **una intera sottorete di accesso**.

In conclusione cambio punto di accesso, e quindi sono mobile, se e solo se cambio **sottorete di accesso**. Se invece cambio AP, ma resto sempre all'interno della stessa sottorete, il mio IP non cambia!

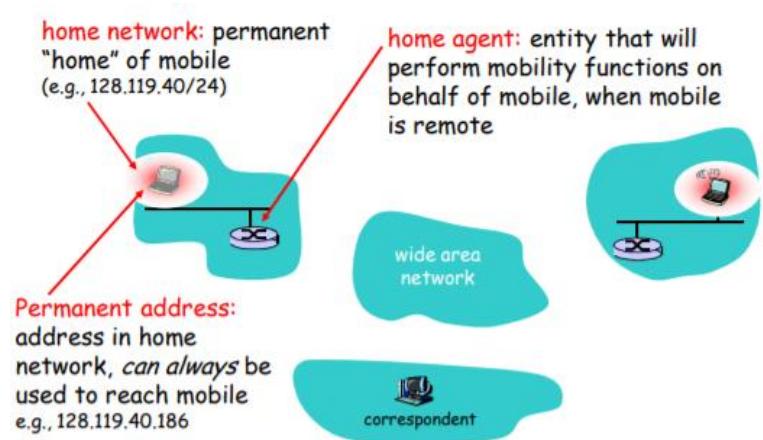
L'utente è in **alta mobilità** se cambia il suo punto di accesso molto frequentemente (magari perché sta viaggiando).

Poiché se l'utente cambiisse IP la connessione TCP cadrebbe, ci sarebbe bisogno di "mantenere il vecchio indirizzo IP anche se mi sono mosso". La soluzione è **estendere il protocollo IP, cercando di fargli comprendere il senso di mobilità!**

### L'idea di base e la possibile implementazione

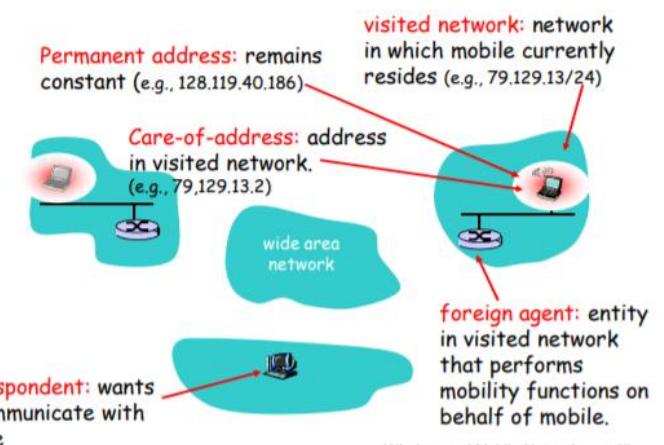
Innanzitutto un po' di terminologia :

- **Home Network** : è la rete "permanente" dell'host che si muove (nel senso di sottoreti IP)
- **Permanent Address** : è l'address preciso dell'host che si muove all'intero dell'Home Network.  
L'obiettivo è fare in modo che tutti



possano raggiungere l'host mobile utilizzando questo indirizzo!

- **Home Agent** : è il router che cercherà (secondo il meccanismo descritto sotto) di comportarsi come l'host mobile quando lui è via
- **Visited Network** : è la sottorete IP in cui il nodo mobile decide di spostarsi
- **Care-of-address** : è l'indirizzo specifico che l'host ha all'interno della visited network
- **Foreign Agent** : è l'entità che, comunicando con l'home agent riesce a far comunicare gli altri con l'host mobile
- **Correspondent** : l'host che vuole comunicare con l'host mobile



L'idea è molto semplice : mettiamoci in uno scenario in cui l'host sia arrivato nella *Visited Network* e si sia registrato al foreign agent.

Compito del foreign agent è vedere il permanent address dell'host mobile e comunicare all'home agent il "nuovo indirizzo IP dell'host mobile" (come a dirgli : "vedi che lui è qui da me").

Alla fine di questa fase il foreign agent è a conoscenza che c'è un host mobile, e l'home agent conosce **dove l'host mobile si trova in quel momento!**

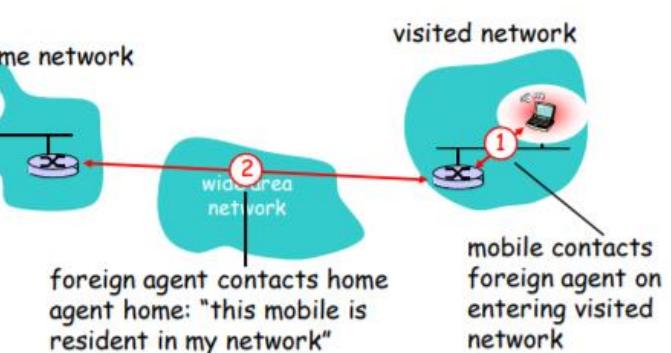
Adesso bisogna pensare a "chi deve curarsi della mobilità" : i router potrebbero ipoteticamente aggiornare le proprie tabelle coerentemente con l'indirizzo IP che l'host guadagna all'interno della visited network, ma questa soluzione **non può scalare considerato l'elevatissimo numero di host mobili!**

L'idea è fare in modo che siano gli end - system a curarsi di questo problema. Ci sono due possibilità :

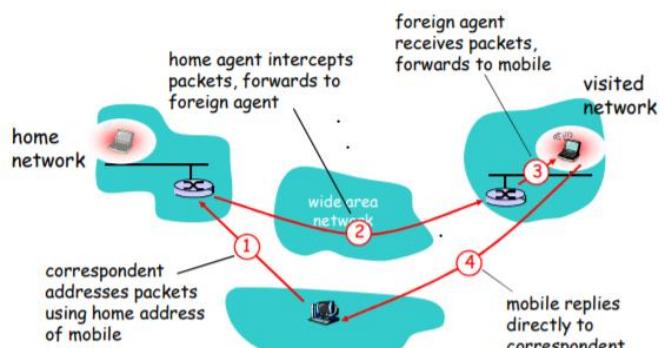
- **Indirect Routing** : il correspondent comunica con l'home agent, che poi provvede a inoltrare la richiesta all'host mobile contattando il foreign agent
- **Direct Routing** : il correspondent ottiene il nuovo indirizzo dell'host mobile, e comunica direttamente con lui

Analizziamo il primo meccanismo :

1. Il correspondent utilizza **il permanent address** per indirizzare il suo messaggio. Questo messaggio è indirizzato dall'home agent, poiché la home network è di sua responsabilità
2. L'home agent **manda i pacchetti al foreign agent**, e può farlo perché in fase di registrazione il foreign ha comunicato con l'home circa la propria ubicazione
3. Il foreign agent **manda i pacchetti al mobile Host**
4. L'host mobile riceve come source address l'indirizzo del correspondent, e dunque può rispondere direttamente a lui!



**C'è un problema :** è necessaria una triangolazione per comunicare, e questa non è proprio la migliore delle cose in termini di overhead! Tra i vantaggi troviamo sicuramente il fatto che la mobilità dell'host appare



*trasparente al correspondant*, e che il compito del foreign agent potrebbe essere svolto proprio dall'host mobile, risparmiando un passaggio.

Nota : C'è triangolazione anche quando correspondant e host mobile sono nella stessa rete, e quindi si rischia che i datagram facciano un lungo viaggio anche quando la distanza fisica è minima!

Se l'Host si sposta in una nuova Visited Network, durante tutto il tempo in cui l'host fa una nuova registrazione con il foreign agent l'home agent non è ancora a conoscenza dello spostamento e tutti i datagram mandati in quel tempo saranno persi! ("Piccola Disconnessione del Servizio").

Nel meccanismo a **Direct Routing**, il correspondant richiede e riceve (1, 2) l'indirizzo "nuovo" dell'Host mobile o meglio del **foreign agent** in cui l'host si trova (e questo indirizzo è conosciuto a causa della registrazione).

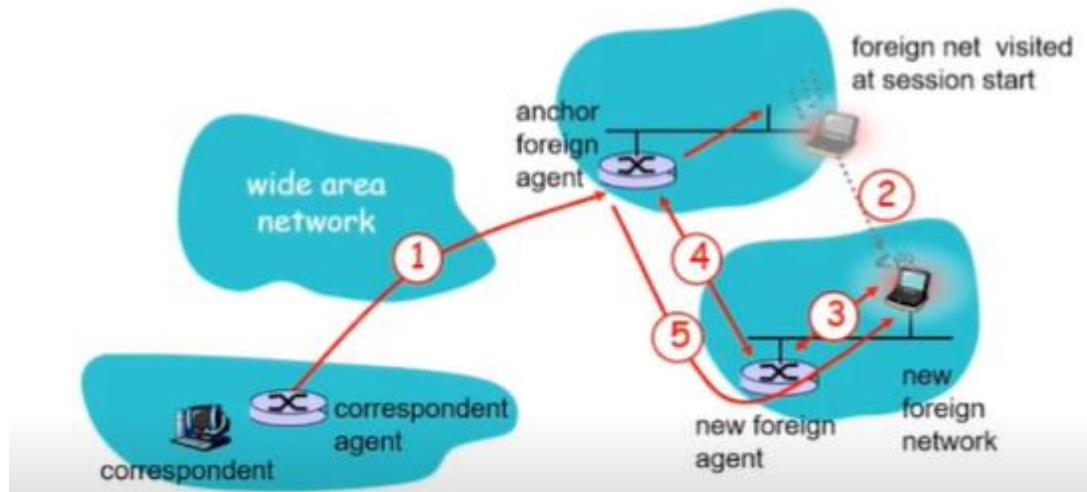
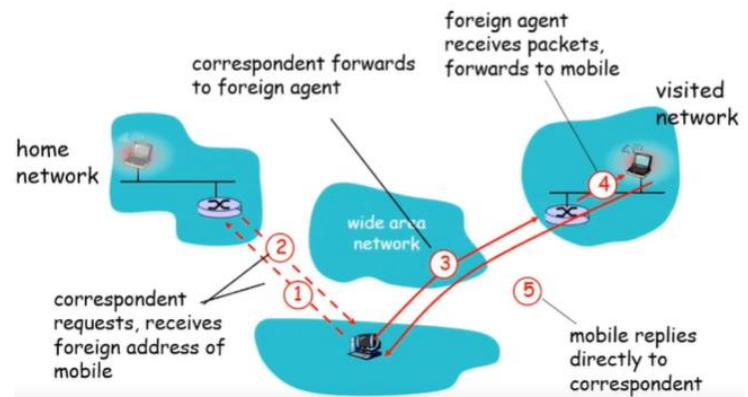
La richiesta del correspondant è "non voluta", ovvero è l'home agent che si accorge di dover comunicare un nuovo indirizzo!

A questo punto il correspondant parla direttamente col foreign agent, e riceve le risposte dall'host mobile.

Non c'è più triangolazione, ma nemmeno trasparenza al correspondant (il che potrebbe creare problemi di privacy).

Cosa succede se invece l'Host mobile cambia rete? Il correspondant non comunica più con l'home agent!

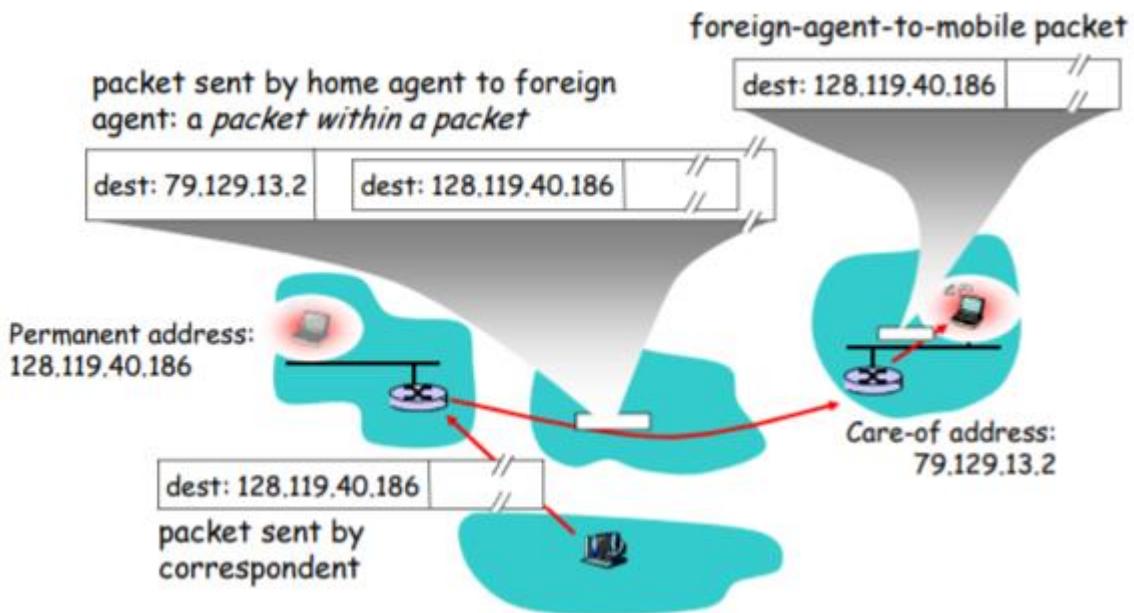
Per garantire la continuità, il nuovo FA (Foreign Agent) deve dire al vecchio voreign agent **la nuova ubicazione dell'host mobile**.



Si può creare così una catena : il correspondant parla con il FA Ancora, che poi comunica col nuovo FA in cui si trova l'Host.

## Quale implementazione ha scelto Mobile IP?

Mobile IP è l'estensione di IP che tiene conto della mobilità degli host e sfrutta il meccanismo dell'**indirect routing**!

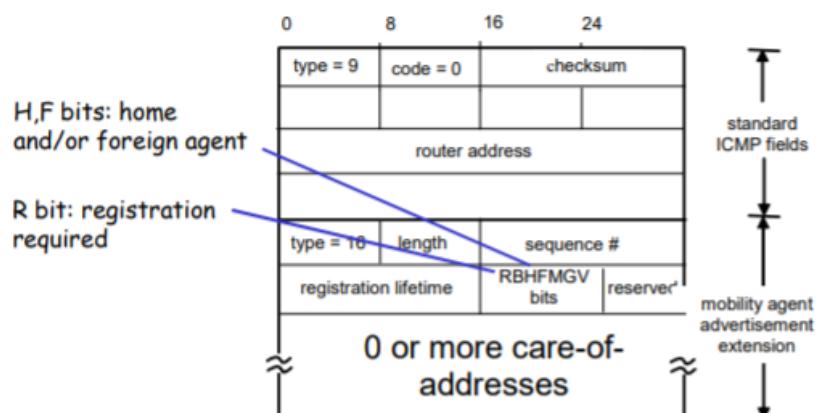


Ci rendiamo conto che l'home agent effettua un'operazione di **"Encapsulation"** quando arriva un messaggio del correspondent, in cui incapsula il pacchetto in un altro che spedirà lui stesso, inserendo come destination address il care - of - address comunicatogli dal foreign agent in fase di registrazione.

**Ma come fa l'host mobile a scoprire il foreign agent, e come fa questi a comunicargli un care - of - address?**

I foreign / home agent mandano periodicamente dei messaggi ICMP di tipo 9 con i quali comunicano:

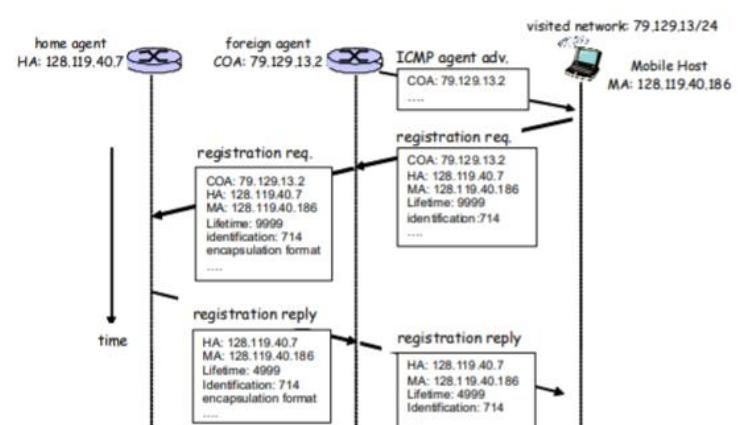
- Se si comportano da Home , Foreign o da entrambi
- Settano un bit R se la registrazione è richiesta (e di solito lo è)
- Comunicano, in caso di foreign, una serie di **care - of - address (COA)** che l'host mobile può utilizzare.



Capiamo ora come funziona la fase di registrazione, che si articola in 4 fasi.

**Fase 1 :** l'host mobile, venuto a sapere del FA presente, formula una registrazione request dove indica il suo Home Agent, e il suo permanent address, e richiede un certo COA advertisato prima

**Fase 2 :** la richiesta di registrazione è propagata dal foreign agent all'home agent. Grazie ai dati inseriti dall'host mobile,



l'home agent può essere contattato e può capire di chi si parla

**Fase 3 :** l'HA prepara un messaggio di registration reply mantenendo lo stesso identificativo trovato prima e dove può decidere, eventualmente, di modificare il lifetime richiesto dal mobile host

**Fase 4:** il FA propaga il risultato della registrazione al mobile host, che ora può utilizzare il COA per il lifetime indicato.

### Cosa succede ai protocolli di livello superiore?

Ci sono delle piccole conseguenze :

- Il TCP direbbe che c'è congestione anche quando il pacchetto si è perso a causa della **mobilità**. La congestion window sarebbe decrementata non necessariamente, col throughput che diminuisce!

Anche nei casi wireless ci sono perdite di pacchetti dovute ad errori di trasmissione, che il TCP continua a interpretare come congestione.

Sembra esserci bisogno di una piccola modifica al TCP :

Nelle reti Wireless un'opportunità è implementare un **local recovery**, in cui si implementa un ARQ scheme o un Forward Error Correction a livello data link.

Un'altra opportunità è fare in modo che il TCP distingua i loss dovuti a **errori del canale** dai loss dovuti alla congestione.

Un'altra ancora è spezzare la connessione TCP in due tronconi : una parte totalmente wired e una parte totalmente wireless!

## Reti Wireless Senza Infrastruttura

Se mi trovo in un posto in cui non c'è alcuna infrastruttura, posso mettere su una **rete Ad Hoc** fatta solo di link wireless, e di nodi (statici e mobili)?

Un esempio di questo tipo sono le **reti Bluetooth**

### Bluetooth (IEEE 802.15.1)

La stringa 802.15 è una famiglia di standard PAN (**Personal Area Network**). Bluetooth è una tecnologia WPAN (Wireless PAN), in cui :

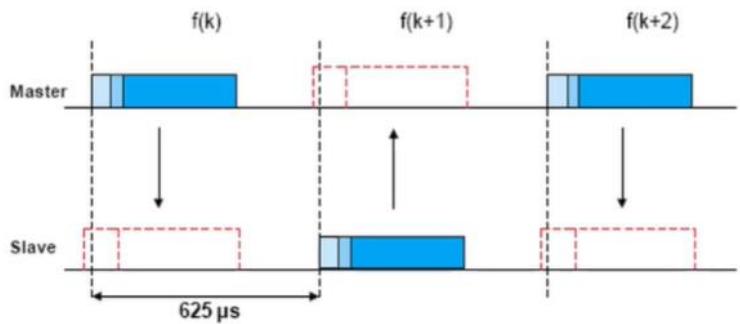
- Il dispendio energetico è minimo, e anche il costo deve essere basso (infatti ora lo mettono ovunque)
- Il raggio è molto corto (di solito una decina di metri)
- Nasce come meccanismo di **cable replacement** dei dispositivi di uso personale, per fare in modo che non ci fosse bisogno di collegare tutti i dispositivi di un utente attraverso dei cavi, e per permettere l'automatica sincronizzazione con i dispositivi in range.
- Visto che c'era, si è pensato di implementarlo anche per la comunicazione telefonica in locale, permettendo, ad esempio, agli utenti in auto di utilizzare cuffie bluetooth per parlare al telefono.
- Un'ultima applicazione pensata è creare **ad hoc networking** (quello che facevamo da piccoli per scambiarci musica e file).

Nel linguaggio dello standard la rete così creata si chiama **Piconet**, in cui ci sono tre tipi di attori : slave, master e parked.

Il protocollo si basa sul cambiare frequenza ad ogni Time Slot, e fare in modo che tutti i nodi siano coordinati in questo cambio di frequenza. Ma perché c'è bisogno di cambiare frequenza e non tenere una delle 78 disponibili? Bluetooth opera con potenze molto piccole (dell'ordine dei milliWatt) sia per motivi di salute e sia per il fatto che la distanza da coprire è molto piccola.

Se si incontrano un segnale a potenza 1000 volte superiore (come Wi-Fi rispetto a bluetooth) il bluetooth sarebbe soppresso! Per questo motivo cambiare ogni volta frequenza permette di fare in modo che le interferenze con le altre reti siano minime!

La sequenza ciclica delle frequenze è conosciuta da **tutti i nodi che stanno all'interno della PicoNet**, ed è comunicata dal Master. La durata dello slot è 625 microsecondi. Durante ogni slot un host può ricevere un pacchetto o inviarne uno. Il protocollo è un adattamento del TDMA ed è detto TDD (**Time Division Duplexing**) e non è molto fair perché metà degli slot è associata al master, mentre l'altra metà a tutti gli altri slave.



Nella metà degli slot degli slave, c'è bisogno di scegliere chi sarà a trasmettere. Questo è possibile grazie ai pacchetti di **poll** che il master manda negli slot pari. Chi riceve il poll si sveglia e trasmette durante gli slot dispari. Non è detto che lo slave scelto abbia qualcosa da mandare, ma questo il master non lo sa!