
Reti di Elaboratori

Introduzione alle reti

Capitolo 1

Enrico Mensa,

Basato sulle lezioni del prof. [Franco Sirovich](#)

Introduzione

1) *Perché le reti?* 1

1.1) La rete di calcolatori

1.2) Scopi delle reti di calcolatori

1.3) Sviluppo del wireless e della portabilità

Hardware e topologia di rete

1) *Nozioni introduttive* 2

1.1) L'hardware: un primo approccio

2) *Le tecnologie di trasmissione* 2

2.1) Tecnologia trasmittiva broadcast

Le tipologie di indirizzi

2.2) Tecnologia trasmittiva punto-punto

3) *Estensione geografica* 4

3.1) LAN: reti locali

Dimensioni

Tecnologia di trasmissione

Topologia

3.2) MAN: reti metropolitane

3.3) MAN: reti geografiche (WAN)

4) *Ancora tecnologie: le reti wireless e le altre reti* 6

4.1) Le LAN wireless

4.2) Le reti "domestiche"

4.3) Le Internetwork

Software di rete e layers

1) *Il concetto di livello* 7

1.1) I livelli come astrazione

1.2) La metafora della libreria	
2) Il software nelle reti di comunicazione	7
3) I protocolli	8
3.1) Un protocollo banale	
3.2) Il protocollo di comunicazione	
3.3) Due filosofi che vogliono comunicare	
3.4) L'architettura di rete	
3.5) Progettare un'architettura di rete: alcuni problemi da risolvere	
Indirizzamento	
Controllo degli errori	
Controllo di flusso	
Ordinamento	
Frammentazione e riassemblaggio	
Multiplexing e demultiplexing	
Routing	
4) I servizi	11
4.1) Servizi orientati alla connessione / senza connessione	
Servizi orientati alla connessione (CO)	
Servizi senza connessione (CL)	
4.2) Servizi affidabili / non affidabili	
Tasso di errore e tasso di errore residuo	
5) Servizi e protocolli: due concetti da non confondere	12
5.1) Un'analogia chiarificatrice	

Modelli di riferimento (OSI, TCP/IP)

1) Il modello di riferimento ISO/OSI	13
1.1) I principi fondamentali	
1.2) I livelli del modello	
Livello fisico	
Livello Trasferimento Dati	

Livello di Rete (Network)
Livello di Trasporto
Livello di Presentazione
Livello Applicazione

2) *Il modello di riferimento TCP/IP* 14

Livello di Internet
Livello di Trasporto

3) *TCP/IP vs ISO/OSI: chi ha vinto la battaglia?* 15

Esempi di rete

1) *Esempi di reti* 16

1.1) ARPANET

1.2) Reti CO: X.25, Frame Relay e ATM

1.3) Ethernet

1.4) Wireless LAN

Gli standard

1) *I protocolli standard* 17

2) *Organismi di standardizzazione* 17

2.1) Settore delle telecomunicazioni (guidata dai service provider)

2.2) Standard tecnologici (guidata dai costruttori)

2.3) Nel settore di Internet (inizialmente guidata dagli utilizzatori)

Introduzione

1) Perché le reti?

1.1) La rete di calcolatori

Una **rete di calcolatori** è un insieme di **dispositivi** in grado di **trasmettere** e **ricevere** informazioni. I calcolatori sono fra loro collegati tramite una rete di comunicazione.

La **rete di comunicazione** è una infrastruttura **hardware/software** che permette la **comunicazione** tra un insieme di calcolatori.

È importante notare la differenza fra la rete di calcolatori e la rete di comunicazione.

L'obbiettivo della prima è quello di fornire un servizio per gli utenti, mentre invece la seconda ha il solo obbiettivo di fornire un mezzo di connessione fra i calcolatori.

Spesso le due reti sono addirittura gestite da soggetti differenti che quindi avranno anche obbiettivi differenti.

1.2) Scopi delle reti di calcolatori

Vi sono diversi scopi per cui creare una rete di calcolatori:

- Condivisione di risorse (file, stampanti) per ridurre i costi di servizio
- Meccanismi di comunicazione tra impiegati (o per assistenza)
- Commercio Elettronico
- Accesso ad informazioni remote (i siti Web)
- Comunicazione da persona a persona (MSN, Peer to Peer, Social Network)
- Intrattenimento (Videogiochi)

1.3) Sviluppo del wireless e della portabilità

Queste due caratteristiche, seppur spesso presenti sullo stesso dispositivo, sono totalmente scollegate nella loro crescita/sviluppo. Tant'è che esistono tutte le combinazioni fra portabilità e reti wireless.

Wireless	Mobile	Utilizzo
No	No	Computer desktop (scollegato dalla rete, non portatile)
No	Sì	Notebook in albergo (portatile ma connesso alla rete via cavo)
Sì	No	Rete wireless sfruttata su calcolatori fissi (edifici storici dove non si tirano cavi)
Sì	Sì	Notebook in albergo (portatile e connesso alla rete wireless)

Hardware e topologia di rete

1) Nozioni introduttive

Parlando di hardware ci troveremo spesso ad avere a che fare con unità di misura (per velocità, dimensione, ecc.) Per queste unità si adoperano i suffissi classici del sistema "mks". Si applichi l'esponente a 10, in notazione scientifica.

Exp	Prefix	Exp	Prefix	Exp	Prefix	Exp	Prefix
-15	Femto	-3	Milli	3	Kilo	15	Peta
-18	Atto	-6	Micro	6	Mega	18	Exa
-21	Zepto	-9	Nano	9	Giga	21	Zetta
-24	Yocto	-12	Pico	12	Tera	24	Yotta

NOTA: Si noti che nel gergo delle reti, quando si parla di Mbit si parla di 10^6 bit, ovvero 1.000.000 bit.

Ma invece, quando si parla di Mbyte (ad esempio in relazione ad hard disk), si parla di potenze di due e quindi di 1024×1024 byte di memoria, ovvero 1.048.576 byte (leggermente di più di 10^6 quindi!).

1.1) L'hardware: un primo approccio

Le reti di calcolatori sono catalogabili secondo due grandi categorie:

- **Tecnologia di trasmissione** adottata (wifi, rame, ecc.)
- **L'estensione geografica** della rete stessa

Abbiamo poi caratteristiche *macroscopiche* quali:

- **Velocità di trasmissione**, calcolata in bit/s. Si tratta della quantità di dati che vengono accettati/consegnati e NON la velocità che i dati hanno durante la trasmissione poiché quella velocità è sempre prossima a quella della luce. Possiamo immaginare i dati come acqua in un tubo: più il tubo è grande più acqua riuscirà ad immettersi nel tubo in meno tempo e viceversa. Ma le molecole d'acqua, all'interno del tubo, viaggeranno sempre a velocità costante (prossima a quella della luce).
- **Affidabilità**, si tratta della probabilità che la rete sia **funzionante** (trasmetta qualcosa) e che i dati trasmessi siano **corretti** ovvero non vengano intaccati durante il tragitto. Questa probabilità, per la costituzione stessa della rete (l'errore è sempre presente) non può essere pari a 1 ma l'obiettivo è quello di alzarla il più possibile.
- **Costo**.

2) Le tecnologie di trasmissione

Vediamo le reti basandoci sul primo dei due aspetti sopra nominati. Abbiamo due tipologie di tecnologie di trasmissione:

- Broadcast
- Punto-punto

Vediamole nel dettaglio.

2.1) Tecnologia trasmissiva broadcast

La caratteristica fondamentale di tale tecnologia è il fatto che abbiamo un **solo** canale di trasmissione, il quale è quindi **condiviso** fra tutti i calcolatori connessi.

Classici esempi di broadcasting sono la radio, la voce, ecc.

Esaminiamo la situazione della voce. Una persona parla e tutti coloro che sono intorno a lui entro un certo raggio sono in grado di sentirla. Pertanto il messaggio viene "spedito" una sola volta, ed ogni persona "connessa" alla rete (ovvero collegata al canale, in questo caso, l'aria, che trasporta le onde sonore) è in grado di ascoltare il messaggio. Così come nel nostro esempio una frase viene pronunciata, nelle reti si ha un **pacchetto di dati** che viene trasmesso da un calcolatore e che è visibile a tutti gli altri. Questo tipo di trasmissione è detta **trasmissione a pacchetto**. Possiamo facilmente immaginare il perché di questa scelta implementativa: se ogni calcolatore dovesse richiedere la disponibilità della rete per ogni bit che deve inviare, ci sarebbero moltissime richieste e ciò intaserebbe la comunicazione. In questo modo vengono invece passati pacchetti di diversi byte tutti in una volta.

Bisogna però tener conto del fatto che ogni calcolatore è identificato da **uno o più indirizzi**. Quando noi vogliamo parlare ad una data persona, includiamo magari il suo nome nel nostro messaggio, e così accade nelle reti.

Viene quindi inserito nel pacchetto l'indirizzo del destinatario, in questo modo ogni calcolatore "in ascolto" non dovrà far altro che verificare se il pacchetto in arrivo è diretto a lui o meno (ricordiamo che ogni calcolatore riceve tutti i pacchetti), ed in tal caso **accettarlo** (o **riceverlo**) (ovvero prenderlo in carico e spostarlo su un buffer, ad esempio).

Abbiamo però detto che ad ogni calcolatore può corrispondere uno o più indirizzi, pertanto esso accetterà un pacchetto se esso contiene uno dei suoi indirizzi.

Le tipologie di indirizzi

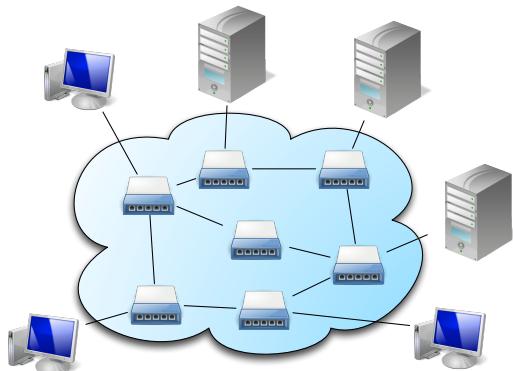
Vi sono tre tipologie differenti di indirizzi:

- **Broadcast**: potrebbe essere utile recapitare un messaggio a tutti i calcolatori, per questo viene riservato **un particolare indirizzo** che è assegnato a tutti i calcolatori della rete. Il pacchetto è spedito una volta soltanto ma è accettato da tutti.
- **Multicast**: un indirizzo viene assegnato a **molti** calcolatori, formanti così il "gruppo multicast". Si potrebbe vedere come un sottoinsieme del broadcast, ma in realtà è il broadcast ad essere visibile come un sottocaso del multicast, ovvero un multicast che ha come gruppo tutti i calcolatori della rete. La radio è un ottimo esempio, poiché una stazione radio trasmette a una certa frequenza e chi è interessato si sintonizza, anche se ogni radio riceve le onde di tutte le stazioni radio.
- **Unicast**: un indirizzo che è posseduto da **al più** un solo calcolatore della rete, quindi, solamente quel calcolatore sarà l'unico in grado di ricevere un messaggio a lui diretto.

Pertanto, data la tecnologia broadcast, avremo un traffico broadcast, multicast oppure unicast.

2.2) Tecnologia trasmissiva punto-punto

Si tratta in pratica di un unicasting tra coppie di macchine. Anche qui, si può vedere come un broadcast fra tutte le macchine disponibili, cioè solamente due (la coppia). Come si può immaginare, la situazione è più complessa. Abbiamo qualcosa del genere:



L'infrastruttura consta di dispositivi che collegano fra loro le varie coppie di macchine. Quando un pacchetto deve essere inviato da una macchina all'altra, inizia il processo di **intradamento** che viene operato da questi dispositivi, i quali, passandosi il pacchetto, lo portano a destinazione.

Questa configurazione porta ad una distinzione fondamentale: il contenuto della 'nuvola' e ciò che sta fuori dalla 'nuvola'. La "rete" (il contenuto) è spesso pubblica mentre invece i "dispositivi utente" esterni sono privati.

Si noti inoltre che, grazie a questa tecnica, le reti sono in grado di ricoprire vaste zone geografiche e di connettere terminali anche geograficamente distanti (infatti i singoli dispositivi di intradamento sono in grado di far rimbalzare il messaggio e così facendo lo "rigenerano" risolvendo il problema del "rovinarsi" dei dati, ne parleremo meglio in seguito).

3) Estensione geografica

Secondo criterio di classificazione di cui abbiamo parlato è l'estensione geografica.

Aiutiamoci con la distinzione operata dal Tanenbaum.

Distanza Coperta	Locazione degli altri dispositivi	Esempio
1 metro	Un metro quadrato	Personal area network
10 metri	Una stanza	Local area network (LAN)
100 metri	Un edificio	
1 kilometro	Campus	
10 kilometri	Città	Metropolitan area network
100 kilometri	Paese	Wide area network
1.000 kilometri	Continente	

3.1) LAN: reti locali

Si tratta di reti private a **ridotta** estensione geografica. Le peculiarità di questa rete sono la dimensione, la tecnologia di trasmissione e la topologia.

Dimensioni

La dimensione di questo tipo di reti è piuttosto ridotta e per questo motivo è possibile sapere qual'è il tempo massimo di trasmissione, quindi, è possibile applicare tecniche di trasmissione ad hoc che altrimenti non potremmo utilizzare.

Inoltre, la gestione è abbastanza semplificata dato che la rete è molto "familiare".

Le LAN operano tra i 10 ed i 100 Mbit/s, fino a toccare i 10 Gbit/s nelle reti particolarmente avanzate.

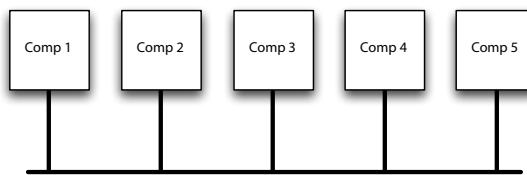
Tecnologia di trasmissione

Dato che le LAN utilizzano un cavo unico (**fisico o concettuale**) a cui tutti i calcolatori sono collegati (prendiamo ad esempio il cavo Ethernet), la tecnologia di trasmissione che più si confà a questa struttura fisica è il broadcasting.

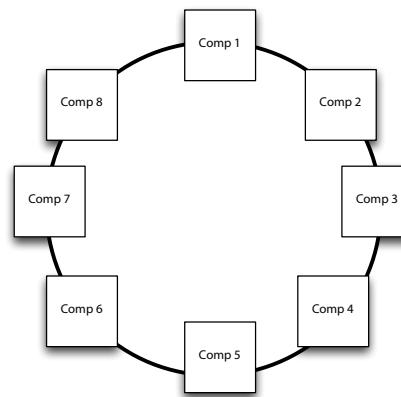
Topologia

Innanzitutto a seconda della topologia si utilizzano meccanismi per decidere quale sia il calcolatore a poter "parlare" sulla rete e che quindi abbia diritto di mandare i suoi pacchetti.

Abbiamo topologie a **bus**:



Oppure a **stella**, oppure ancora ad **anello**:



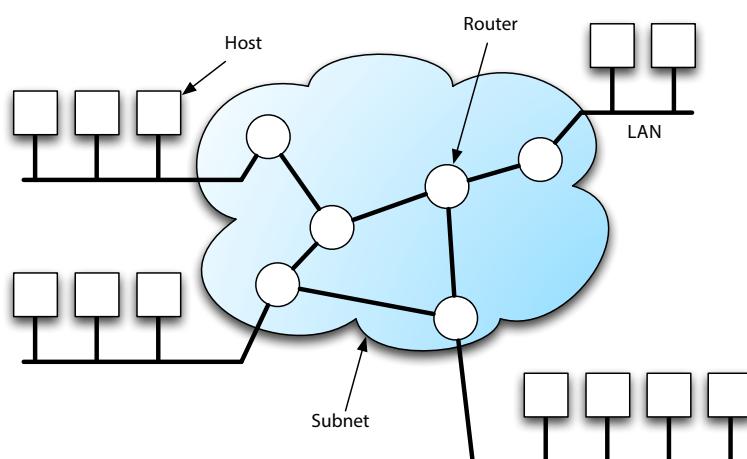
3.2) MAN: reti metropolitane

Abbiamo una rete che ricopre un'area metropolitana: in questo caso è possibile avere trasmissioni sia broadcast e sia punto-punto. Grande utilizzo della fibra ottica con topologia ad anello.

Si hanno sia reti pubbliche che reti private che però devono aver avuto le autorizzazioni per utilizzare il suolo pubblico per "tirare i cavi".

3.3) MAN: reti geografiche (WAN)

L'area geografica ricoperta è molto vasta. Ogni calcolatore (detto **host**) è collegato ad una **communication subnet** che svolge la mansione di trasportare i pacchetti tra i vari calcolatori. Generalmente si tratta di rete pubbliche, dato l'alto costo di costruzione/mantenimento.



Gli host sono di proprietà dei clienti mentre invece la subnet è gestita da aziende che vendono questo servizio. Tali aziende, chiamate Service Provider, operano in concessione dello stato.

Come si può vedere la **subnet** è costituita da linee di trasmissione che portano i bit da una macchina all'altra e da commutatori, chiamati **router** (instradatori).

Tali router prendono in consegna un pacchetto da uno degli host, dopo di che spediscono il pacchetto al router successivo (designato tramite un **algoritmo di routing**) fino a raggiungere l'host destinatario.

Vi sono due approcci differenti:

- reti **connection oriented (CO)**
- reti **connection less (CL)**

Nel primo caso quando un messaggio viene "spedito" viene designata una strada fra i router: quella strada verrà adoperata per tutti i pacchetti componenti il messaggio.

Nel caso invece di una CL, ad ogni pacchetto costituente il messaggio verrà applicato l'algoritmo di routing alla ricerca della strada migliore per portare il singolo pacchetto a destinazione.

4) Ancora tecnologie: le reti wireless e le altre reti

La grande differenza fra reti cablate e reti wireless è il fatto che il mezzo di trasporto sia infinitamente meno costoso (seppur più pericoloso dal punto di vista della salute). Tendenzialmente, più il collegamento è "poco distante" (ad esempio l'auricolare bluetooth) e meno ci farà male.

4.1) Le LAN wireless

Esistono anche LAN wireless (WLAN) che hanno portata di alcune decine di metri, con ottime velocità. Si stanno ora sviluppando reti a larga banda basate sul wireless, WiMAX (che risolverebbe il problema dell'ultimo miglio, che in Italia impedisce ai concorrenti della Telecom di farsi strada nel settore).

4.2) Le reti "domestiche"

Vengono sempre più spesso adottate reti domestiche per collegare TV, DVD, VRC, videocamere, elettrodomestici.

Tali reti hanno però requisiti non facili da soddisfare:

- Si vuole una rete sicura ma semplice
- Costi ridotti
- Grandi capacità per supportare il multimedia

4.3) Le Internetwork

Trattasi di una tecnologia software che permette la connessione fra più reti diverse.

Non si tratta quindi di una tecnologia di rete di comunicazione vera e propria, ma bensì di una tecnologia per **interconnettere reti di comunicazione**.

Si noti che la parte interessante è il fatto che le reti interconnesse possano essere di geografia diversa (LAN, MAN, WAN) e possano addirittura sfruttare mezzi di trasporto differenti.

Le diverse reti sono collegate fisicamente tramite l'ausilio di **router** (anche detti **gateway**).

Tutto questo è reso possibile grazie al **software di rete**.

Si viene quindi a creare una vera e propria **rete virtuale**: il massimo esempio è Internet.

Software di rete e layers

1) Il concetto di livello

1.1) I livelli come astrazione

Prendiamo un elaboratore.

Esso funziona "a livelli", ovvero a successive "zone" mano a mano più complesse che cooperano per il funzionamento del calcolatore stesso.

Ogni livello **nasconde** i dettagli del livello sottostante, ma **fornisce** delle nuove funzioni per il livello successivo.

All'interno del livello abbiamo quindi delle strutture e dei componenti che vengono aggregati per costituire nuove entità utilizzabili poi dai livelli sovrastanti.

1.2) La metafora della libreria

Un'ottima metafora è quella della libreria di funzioni.

Ogni livello di macchine virtuali è in sostanza una libreria che prende funzioni precedenti e le aggredisce per crearne di nuove, utilizzabili a loro volta da chi implementa la libreria, ovvero da altre librerie, ovvero da altri livelli.

La libreria, quindi, offre un servizio che consta di una serie di funzioni.

Si noti che la libreria è totalmente trasparente a chi la utilizza, e così per i livelli: al loro interno possono essere implementati in maniera procedurale piuttosto che a scambio di messaggi (dipende dalla scelta implementativa) ma questo non tocca l'utilizzatore poiché esso adopera le funzioni tramite una **interfaccia**.

Quindi i livelli comunicano fra loro tramite interfacce.

2) Il software nelle reti di comunicazione

Il software di rete è estremamente complesso poiché deve svolgere mansioni parecchio complicate e deve farlo in maniera sicura e chiara. Si rende quindi necessaria un'attenta organizzazione (a livelli, appunto!) ed è oltremodo necessario specificare in maniera **non ambigua** quali **convenzioni** i dispositivi in comunicazione debbano rispettare per poter dialogare. Questo passaggio è fondamentale se consideriamo il fatto che spesso dispositivi diversi provengano da aziende diverse, le quali senz'altro non possono "mettersi d'accordo" fra loro ma hanno bisogno di una chiara standardizzazione per poter mettere in comunicazione le macchine.

Il software di comunicazione ci permette quindi di:

- scambiare dati tra calcolatori/dispositivi in rete
- controllare i dispositivi di rete (i router, per intenderci)

Tale software è quindi presente sia sui dispositivi di rete che sui calcolatori.

Possiamo pensare ad una banca che voglia interconnettere i suoi calcolatori. Non solo il programma che gestisce i conti correnti dovrà funzionare sul server centrale, ma dovrà anche funzionare su tutti i Bancomat e in tutte le filiali della banca.

3) I protocolli

Vediamo ora il fondamentale concetto di **protocollo**.

3.1) Un protocollo banale

L'esempio forse più banale di protocollo è quello che adoperiamo ogni giorno durante la comunicazione verbale. Consideriamo questo dialogo:

- A: "Salve."
- B: "Buongiorno."
- A: "Che ore sono?"
- B: "Sono le 2:00."
- A: "Grazie."
- B: "Prego."

Il protocollo **specificava le regole che governano le interazioni** e definisce **azioni da intraprendere** al verificarsi di certi eventi.

Come vediamo, all'inizio A cerca l'attenzione di B introducendosi con un messaggio. Dopodiché effettua la sua richiesta e B, basandosi sul protocollo che "alla domanda -che ore sono?- bisogna guardare l'orologio e rispondere con l'ora corretta", fornisce l'orario ad A. Infine A chiude la conversazione, per educazione, ringraziando. Useremo questo esempio per fare un parallelo con i calcolatori.

3.2) Il protocollo di comunicazione

Del tutto simile al protocollo sopracitato, abbiamo il protocollo di comunicazione.

Tale protocollo riguarda chiaramente i calcolatori e governa le comunicazioni fra **entità** della rete (dove con entità si intendono dispositivi piuttosto che processi di calcolatori). Le entità possono essere diverse ma devono tutte (ovviamente) sottostare al protocollo.

Sono poi definite le **funzioni** (anche dette servizio) che il protocollo offre al livello superiore, oltre che il formato e la sequenza di messaggi scambiate fra entità paritarie.

Sono quindi definite le **azioni** da esse intraprese in conseguenza della ricezione di messaggi o da altri eventi.

3.3) Due filosofi che vogliono comunicare

Cerchiamo di capire perché si renda necessaria una classificazione a livelli.

Prendiamo in considerazione due filosofi che hanno pari conoscenze (diciamo su Kant), ma uno parla inglese e l'altro francese. Inoltre i due filosofi sono geograficamente distanti.

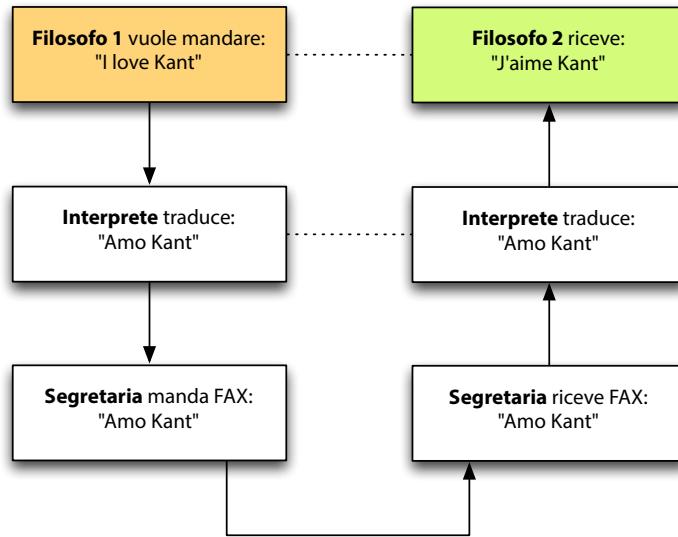
Si rende quindi necessario:

- Apprendere una lingua comune
- Trovare un meccanismo per comunicare

Ma invece di mettersi a studiare una nuova lingua e poi spostarsi da dove si trovano, i due filosofi scelgono di delegare questi compiti ad altri.

Ognuno di loro assume **localmente** un interprete che è in grado di tradurre il messaggio in una lingua comune (diciamo in italiano). Dopodiché ogni interprete, che ha la commissione di spedire il messaggio, assume a sua volta una **segretaria** che tramite FAX invierà il messaggio tradotto alla segretaria dell'altro filosofo.

Possiamo visualizzare tutta la vicenda in questo modo:



Le segretarie utilizzano un **mezzo di trasmissione** (in questo caso il FAX è a sua volta una macchina, ma possiamo immaginarlo come 'atomico') ma sono gli unici "livelli" che sono realmente in comunicazione fisica fra loro. Le altre entità (interpreti e filosofi) sono comunicazione, ma solamente "teorica". Tale comunicazione è possibile poiché ognuno di essi comunica con i livelli sottostanti.

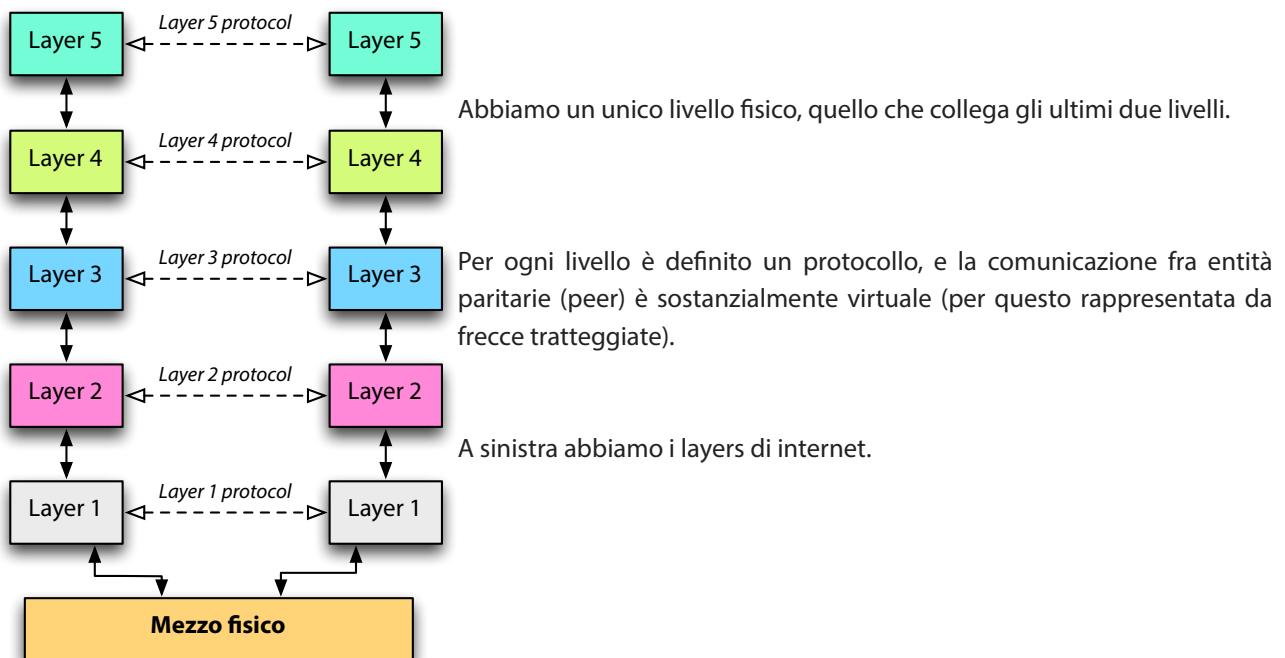
3.4) L'architettura di rete

Il software di comunicazione è quindi organizzato a **strati (layer)** ciascuno dei quali offre servizi di comunicazione allo strato superiore, sfruttando però i servizi del livello sottostante.

Si noti che ogni entità **coopera** con le entità **paritarie** (dello stesso livello).

La comunicazione verticale su un dispositivo (fra livelli diversi adiacenti) avviene per mezzo di interfacce (come detto prima) mentre la cooperazione fra entità di diversi dispositivi (paritari) è possibile grazie all'utilizzo dei servizi dei livelli sottostanti più le funzioni offerte dal protocollo.

L'insieme dei livelli e dei protocolli prende il nome di **architettura di rete**.



3.5) Progettare un'architettura di rete: alcuni problemi da risolvere

In molti degli strati possiamo occorrere in questi problemi (ma non necessariamente in tutti gli strati vi sono tutti):

Indirizzamento

Come sappiamo gli indirizzi servono per identificare un certo calcolatore. Ma in questo caso dobbiamo identificare le entità operanti allo stesso livello ed ogni calcolatore ha più applicazioni con più livelli e quindi con più entità! Supponiamo di avere due browser aperti, ed uno di loro effettua una richiesta ad un server. Quale dei due browser deve ricevere risposta? L'indirizzamento è quindi un aspetto fondamentale che non può essere dimenticato.

Controllo degli errori

Le informazioni trasmesse possono subire errori (per bug del programma, piuttosto che per difetti fisici del mezzo di trasmissione o semplicemente per motivi di dispersione) e quindi ciascun livello deve prevedere dei meccanismi per il rilevamento degli errori commessi dal livello inferiore ed eventualmente la loro correzione (o quantomeno la segnalazione al livello paritario).

Controllo di flusso

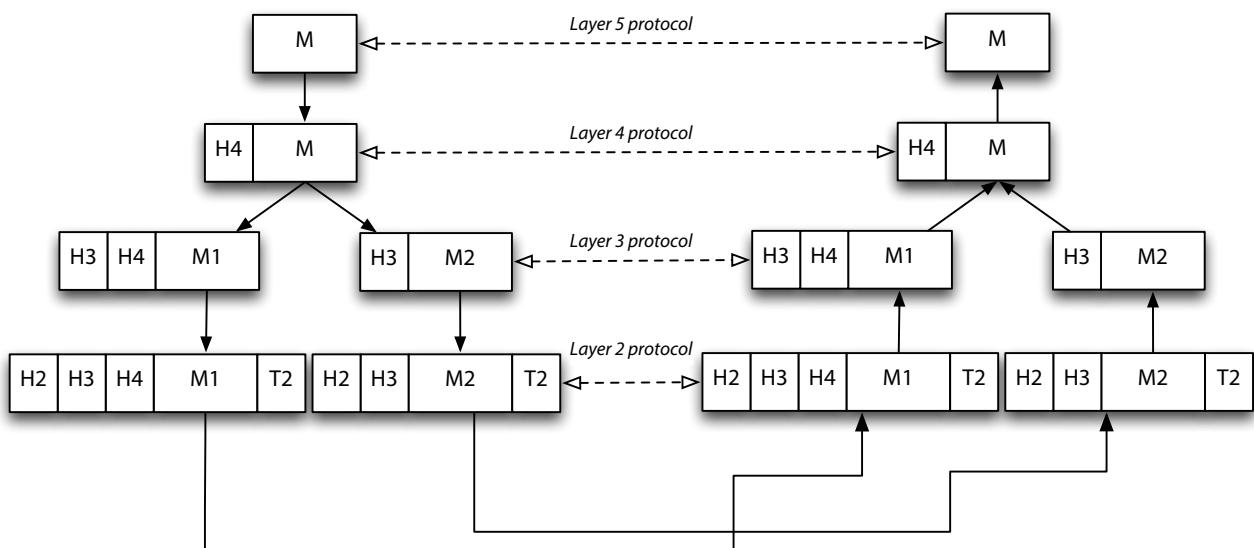
Calcolatori diversi in comunicazione significa macchine che vanno a velocità diversa che si passano informazioni. I buffer non possono contenere tutti i dati in attesa che il computer più lento li riceva. Per questo è necessario controllare il flusso (la velocità) con cui i pacchetti vengono trasmessi.

Ordinamento

Talvolta può accedere che i messaggi giungano a destinazione in ordine corretto. Nel caso in cui il livello inferiore non garantisca l'ordine ed il livello superiore richieda l'ordinamento dei dati, allora il livello frapposto dovrà prendersi carico di riordinare il messaggio.

Frammentazione e riassemblaggio

Non tutti i livelli sono in grado di ricevere messaggi della stessa dimensione, talvolta può essere necessario frammentare il messaggio. Se il livello 2, ad esempio, può leggere solo messaggi più piccoli rispetto al livello 3, sarà compito di quest'ultimo suddividere il messaggio e sarà compito del peer del livello 3 riassembrarlo affinché al livello 4 arrivi un messaggio unico (così come il livello 4 del mittente aveva inviato al livello 3 del mittente).



Come si può notare vengono aggiunti ad ogni livello **headers** e **trailers** (nel livello fisico) che contengono informazioni aggiuntive utili per il riassemblaggio piuttosto che per la correzione degli errori.

Multiplexing e demultiplexing

Per risparmiare risorse può essere utile accoppare più connessioni in una sola (fase di multiplexing) che poi devono essere nuovamente separate una volta giunte a destinazione (demultiplexing).

È un po' come avere quattro persone che vogliono parlare l'una con l'altra. Invece di fare più telefonate separate, è meglio chiamare una sola volta e poi passare il telefono agli altri.

Routing

La scelta del miglior percorso da seguire per un pacchetto in rete dipende da svariati fattori, che quindi a seconda del livello possono essere diversamente gestiti. Si noti che l'algoritmo viene poi eseguito un numero differente di volte a seconda che la rete sia CO (connection oriented) oppure CL (connection less).

4) I servizi

I servizi sono insiemi di primitive che un livello offre a quello superiore attraverso un service access point.

4.1) Servizi orientati alla connessione / senza connessione

Vi sono due tipologie differenti di servizi che un livello può fornire al suo livello superiore.

Servizi orientati alla connessione (CO)

L'analogia più semplice è la chiamata telefonica. È necessario comporre il numero ed attendere la risposta dall'altro capo prima di iniziare la comunicazione. In questa fase è possibile effettuare una **negoziazione**, ovvero i quattro soggetti coinvolti (le due entità che offrono il servizio e i due utilizzatori) possono mettersi d'accordo su alcune caratteristiche della connessione (velocità, ecc.).

Il mezzo trasmissivo è interpretabile come un tubo ai quali capi si trovano i due interlocutori. Uno inserisce i bit, l'altro li preleva. Per questo motivo tipicamente l'ordine di arrivo è lo stesso di quello di invio.

È quindi in generale richiesta molta **coordinazione fra le quattro entità coinvolte**.

Servizi senza connessione (CL)

L'esempio questa volta è la posta. Il mittente invia il suo messaggio al servizio dopodiché non se ne cura. Invia il messaggio apponendo l'indirizzo del destinatario. Ogni messaggio è singolo e viene gestito indipendentemente, potrebbe esserci un messaggio ora e poi uno dopo molto tempo: per questo motivo i percorsi di rete di due messaggi differenti ma tra lo stesso mittente/destinatario potrebbero anche non corrispondere.

4.2) Servizi affidabili / non affidabili

Per poter parlare di servizi affidabili dobbiamo innanzi tutto introdurre i concetti di **tasso di errore** e di **tasso di errore residuo**.

Tasso di errore e tasso di errore residuo

Un errore, quando avviene, può essere rilevato oppure non rilevato. Ovviamente se non è rilevato siamo nella "casistica peggiore" poiché potremmo non accorgerci di avere dati inconsistenti in arrivo.

- **tasso di errore** = tasso di errori *rilevati* + tasso di errori *non rilevati*

Il tasso di "errori non rilevati" è anche detto **tasso di errore residuo**.

Un servizio è **affidabile** quando garantisce che ogni errore sarà certamente rilevato, ovvero ha tasso di errore residuo tendente a zero. Per quanto riguarda il tasso di errore, esso non sarà necessariamente nullo (potrebbero essercene tantissimi, purché siano tutti rilevati!), ma molto probabilmente sarà basso.

Un servizio è **non affidabile** quando il suo tasso di errore residuo è diverso da zero. È possibile che si abbia un tasso di errore estremamente basso, anche più basso di quello di un servizio affidabile, ma noi ci stiamo basando sul concetto di errore residuo per definire l'affidabilità!

I servizi CO possono essere affidabili, poiché con l'allocazione statica delle risorse è più agilmente possibile verificare la correttezza dei messaggi. Viene stipulato un **contratto** al momento della connessione.

I servizi CL invece non possono dare questa garanzia (non sono affidabili), poiché non stipulano un contratto e l'allocazione dinamica delle risorse impedisce un controllo preciso. I servizi CL sfruttano però al meglio le risorse e quindi sono più performanti.

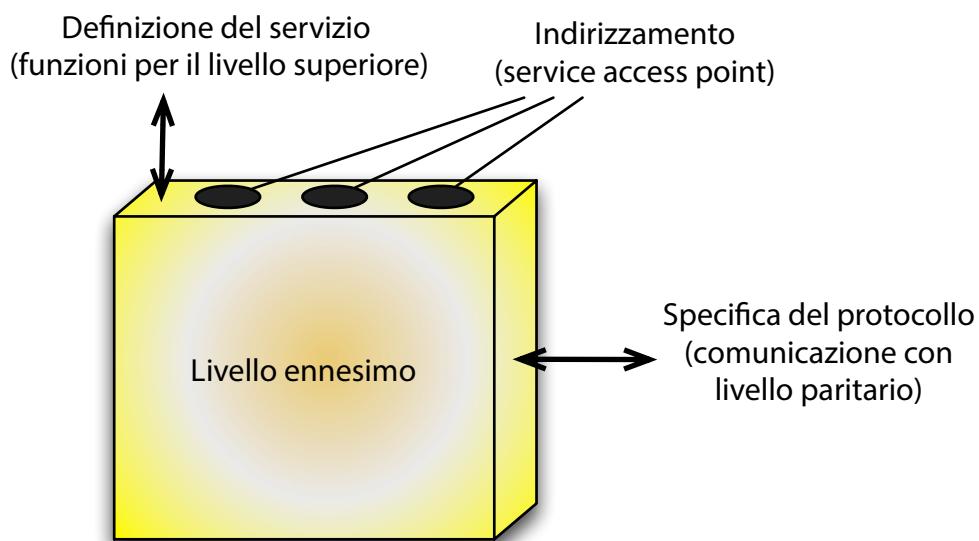
5) Servizi e protocolli: due concetti da non confondere

Ricordiamo che:

Un **servizio** è un insieme di operazioni che un livello offre al suo superiore tramite un service access point.

All'interno di un servizio troviamo la definizione delle operazioni disponibili e della loro semantica, le quali operazioni sono adoperabili tramite interfacce.

È essenziale capire che non è definito come vadano implementate le operazioni e le loro interfacce, rimane quindi tutto "all'interno".



Un **protocollo**, invece, è un insieme di regole che specificano il significato ed il formato dei messaggi scambiati tra le entità di pari livello. Specifica (più o meno formalmente) come implementare il servizio.

Il cambio di un protocollo può essere totalmente trasparente al servizio purché ovviamente non si cambino le operazioni utilizzabili, le interfacce e la loro semantica (dato che potrebbero essere impiegati al livello superiore).

5.1) Un'analogia chiarificatrice

Prendiamo come esempio lo sviluppo di un software.

Il servizio sarebbe l'interfaccia di una classe, il protocollo è la specifica più o meno formale della classe, mentre il codice è una delle possibili implementazioni del servizio.

Modelli di riferimento (OSI, TCP/IP)

1) Il modello di riferimento ISO/OSI

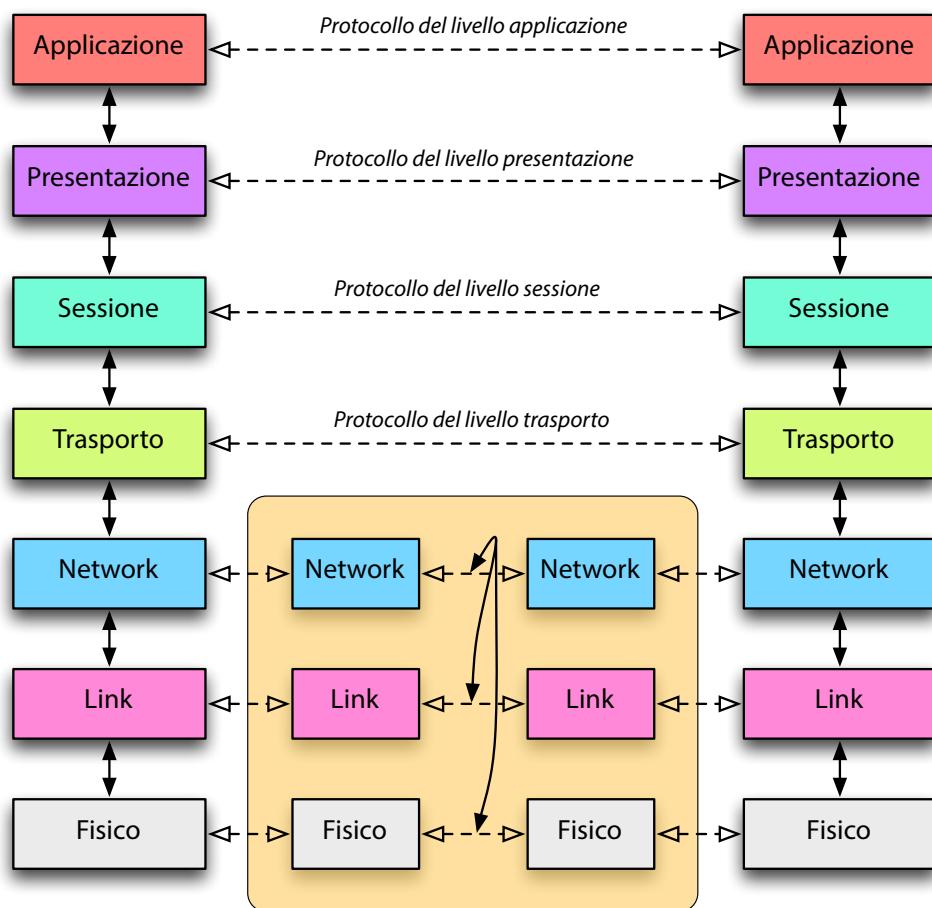
L'ISO è l'International Standard Organization che ha deciso di creare il modello di riferimento OSI (Open System Interconnection), essendosi infatti posta il problema di standardizzare Internet, un nuovo mondo "dilagante" che necessitava, appunto, di standard.

1.1) I principi fondamentali

Nel momento della realizzazione del modello si sono decisi alcuni punti cardine:

- Ogni astrazione diversa deve avere un suo livello.
- Ogni livello deve avere una funzione ben chiara e definita (ma attenzione alla realizzabilità!).
- Le interfacce fra livelli devono cercare di minimizzare il flusso di dati tra un livello e l'altro.
- La definizione degli stati deve permettere un cambiamento di hardware e protocolli senza sconvolgere l'intero sistema.
- Ogni stato può comunicare solamente con il livello sottostante e quello soprastante.

1.2) I livelli del modello



Questo schema descrive il modello ISO/OSI, notiamo che al fondo gli ultimi tre stati sono stati ripetuti poiché come sappiamo i pacchetti passano fra molti router e quindi ripercorrono questi livelli molte volte. Vediamo ora nel dettaglio ogni livello del modello per capire cosa implementa.

Livello fisico

- I bit vengono trasmessi sul canale fisico:
- Codifica dei dati tramite segnali (i bit, appunto)
 - Interfacciamento fra il calcolatore, l'interfaccia di rete (scheda hardware) e la rete

Livello Trasferimento Dati

Avviene l'invio su collegamenti punto-punto:

- Sincronizzazione
- Rilevamento e correzione degli errori
- Controllo del flusso
- Accesso al mezzo trasmissivo

Livello di Rete (Network)

Meccanismi per la comunicazione tra entità collegate rete commutata (quindi tra calcolatori ma anche router):

- Indirizzamento
- Instradamento (routing)

Livello di Trasporto

I calcolatori comunicano in modo sicuro ed affidabile:

- Creazione, mantenimento e terminazione di connessioni
- Controllo sugli errori

Livello di Presentazione

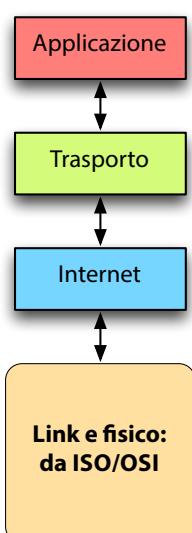
Vengono gestiti i diversi tipi dei dati scambiati fra le applicazioni (compressione, crittografia).

Livello Applicazione

I programmi che interagiscono con l'utente (client di posta elettronica, browser, ecc.).

2) Il modello di riferimento TCP/IP

Premettiamo che il modello TCP/IP nasce prima del modello ISO/OSI: esso è più "immediato" e creato da coloro i quali lo utilizzano, a differenza dell'ISO/OSI creato a tavolino dai produttori. Capiremo meglio dopo le implicazioni di questo fatto.



Il modello TCP/IP non prevede i due livelli **Presentazione** (incorporato nel livello **Applicazione**) e Sessione (ritenuto inutile). Non sono inoltre inclusi i livelli **Link** e **Fisico** poiché vengono sfruttati quelli del modello ISO/OSI.

Livello di Internet

Viene introdotto invece il livello Internet (al posto del Network) il quale offre un **particolare servizio di rete**:

- senza connessione
- non vi sono meccanismi per la garanzia di recapito (quindi dovranno occuparsene i livelli sopra)
- è possibile l'instradamento di pacchetti dello stesso "traffico" su reti fisiche diverse (ordine non garantito)
- si cerca il **massimo utilizzo** delle risorse a discapito delle garanzie
- il protocollo utilizzato per questo servizio è l'**Internet Protocol** (IP).

Livello di Trasporto

Ritroviamo poi il livello trasporto che questa volta offre due servizi di comunicazione differenti:

- TCP: affidabile ed orientato alla connessione. Controllo di flusso con mantenimento dell'ordine di spedizione.
- UTP: non affidabile e senza connessione. Servizio che punta tutto sull'efficienza ma privo di controllo di flusso e garanzia di recapito.

3) TCP/IP vs ISO/OSI: chi ha vinto la battaglia?

TCP/IP nasce dagli **utilizzatori** della rete, quindi è molto più orientato all'utilizzo semplice e veloce, allo sfruttare le risorse.

ISO/OSI, invece, arrivando successivamente, porta un grande concetto ben formato e ben pensato ma perde il punto di vista dell'utilizzatore e punta al vantaggio del **produttore**.

Ci sono poi grande differenze fra i livelli e fra i servizi di trasporto (CO e CL in TCP/IP, solo CO in OSI) e di network (solo CL in TCP/IP, CO e CL in OSI).

Per questi motivi, per quanto riguarda Internet, vince la battaglia TCP/IP.

Esempi di rete

1) Esempi di reti

Vediamo ora alcuni esempi di reti, propedeutici per fissare i concetti appena esposti.

1.1) ARPANET

ARPANET è la "mamma" di internet, non è una vera e propria rete, infatti, bensì un'unione di altre reti.

ARPANET nasce come rete dell'agenzia DARPA del DoD (Departement of Defence) degli USA.

Essa sfrutta la commutazione di pacchetto (quindi il loro instradamento), con alta affidabilità (si può capire dal fatto che le informazioni trasportate fossero essenziali - si dice fosse in grado di resistere ad attacchi nucleari).

ARPANET si basava su una rete telefonica poco affidabile.

Giacché i protocolli usati in ARPANET non erano adatti a collegare reti di tipologia differente (si pensi che non erano ammesse reti LAN!) all'inizio degli anni '70 si sviluppa il protocollo TCP/IP e all'inizio degli anni '80 si vede l'arrivo dei DNS.

Le prime applicazioni reali sono state le email, i news feed, il remote login ed il file transfer. Segue dagli anni '90 il WWW, applicazione definitiva di Internet. Chiaramente gli ISP assumono un ruolo fondamentale, fornendo la funzione di instradatori per i pacchetti della rete.

1.2) Reti CO: X.25, Frame Relay e ATM

Internet, per motivi di prestazioni, è CL.

Ma le compagnie telefoniche hanno sempre preferito CO, poiché permettono la creazione di contratti con allocazione statica delle risorse, fornendo grande qualità di servizio e permettendo quindi di fatturare e calcolare precisamente la quantità di risorsa utilizzata.

Le reti CO quindi hanno seguito un loro sviluppo, come le reti X.25 (commutazione di pacchetto) e le reti frame relay (per il collegamento LAN quindi prive di gestione di flusso).

Altro esempio sono le reti ATM che sono nate negli anni '90 per il trasporto di dati multimediali. I pacchetti sono molto piccoli (detti celle), viene garantito l'ordine ma non la delivery (non un granché...). Oggi vengono usate per le reti telefoniche atte al trasporto IP.

1.3) Ethernet

Ne parleremo meglio in seguito: sono reti basati sull'idea: "effettua una richiesta per un servizio, se non ti viene dato prova più tardi". La connessione è effettuata tramite cavo coassiale, con un massimo di 256 macchine. Nate negli anni '70, sono standardizzate come IEEE 802.3

Sono usatissime in ambito LAN.

1.4) Wireless LAN

Vi sono due approcci, le reti "Wireless networking with a base station" e "ad hoc networking". Nel primo caso abbiamo una base che fornisce la rete a diversi terminali, nell'altro uno dei terminali funge da fornitore della rete.

La prima tecnica è la più usata poiché risolve problemi di copertura radio. Le varie basi sono poi collegate con cavi alla rete.

Le onde radio soffrono però di problemi diversi, come ad esempio i solidi che bloccano la trasmissione delle onde stesse.

Gli standard

1) I protocolli standard

Come è immaginabile esistono moltissimi produttori di software ed hardware, ma l'intento è quello di poter utilizzare i prodotti di diversi produttori sia simultaneamente che fra loro. Si ricerca quindi l'**interoperatività**.

Per realizzare questo compito però si richiede la presenza di organi "superiori" e fuori dagli interessi che possano imporre degli standard che i produttori devono rispettare. Vi sono due tipi di standard:

- **de iure**: definiti e formalizzati da organismi internazionali di standardizzazione di riconosciuta autorità (ad esempio ISO/OSI).
- **de facto**: divenuti tali in virtù in una loro rapida diffusione in comunità molto vaste di produttori/utenti (ad esempio il protocollo TCP/IP).

2) Organismi di standardizzazione

2.1) Settore delle telecomunicazioni (guidata dai service provider)

- ITU: International Telecommunication Union

2.2) Standard tecnologici (guidata dai costruttori)

- ISO: Internaional Standards Organization
- IEEE: Institute of Electrical and Eletronic Engineers

2.3) Nel settore di Internet (inizialmente guidata dagli utilizzatori)

- Internet Research Task Force (IRTF)
- Internet Engineering Task Force (IETF)
- Si lavora con il meccanismo degli RFC (request for comments)

Reti di Elaboratori

Il livello fisico

Capitolo 2

1) Dati, segnali e mezzi trasmissivi 1

1.1) Il dato

1.2) Il segnale

1.3) Il mezzo trasmissivo

2) La trasmissione tramite segnali 1

2.1) Dai dati al segnale

2.2) Come trasmettere i segnali

2.3) La scelta del digitale

3) I segnali periodici 3

3.1) Il segnale periodico e le sue grandezze

L'ampiezza

La frequenza

La fase

3.2) Relazione fra data rate e frequenza

Lo spettro

Caso 1

Caso 2

Caso 3

Conclusione

3.3) Segnali a banda limitata

L'attenuazione non uniforme ed il ritardo

Il rumore

3.4) Capacità di un canale trasmissivo

Concetti importanti per capire la capacità

Teorema di Nyquist

Teorema di Shannon

Esempio tipico

4) I mezzi trasmissivi 7

4.1) I mezzi guidati

Il doppino

Il cavo coassiale

La fibra ottica

Mezzi guidati: riassunto

4.2) I mezzi non guidati

- Le onde radio
- Le microonde
- Infrarossi e onde millimetriche
- Onde luminose
- Comunicazioni satellitari
- Fibra vs Satellite

4.3) La rete telefonica

- Evoluzione storica del sistema
- Il canale telefonico
- La modulazione/demodulazione
- Il baud
- Tecnica QPSK (quadrature phase shifting)
- Evoluzioni dell'QPSK: QAM-16 e QUAM-64
- Una linea in due direzioni
- Linee a 56kbit/sec: oltre il teorema di Shannon?
- Le DSL
- Le DMT
- La Asymmetric DSL (ADSL)
- Il NID (Network Interface Device)
- xDSL asimmetriche
- Il tratto a lunga distanza
- Approccio al FDM e al TDM
- Il FDM (Frequency Division Multiplexing)
- Il TDM (Time Division Multiplexing)
- La codifica dei segnali binari
- Ancora sul TDM (Time Division Multiplexing)

4.4) La rete telefonica: la commutazione

- Commutazione di circuito
- Commutazione di pacchetto

1) Dati, segnali e mezzi trasmissivi

1.1) Il dato

Un dato è un'entità a cui è associato un significato.

Vi sono due tipologie di dati:

- **analogici**: i valori assunti sono in un insieme continuo (la voce ad esempio).
- **digitali**: i valori assunti sono in un insieme discreto (sequenze di bit, l'alfabeto).

1.2) Il segnale

Il segnale è un'indicazione convenzionale che è utilizzata per la comunicazione di dati.

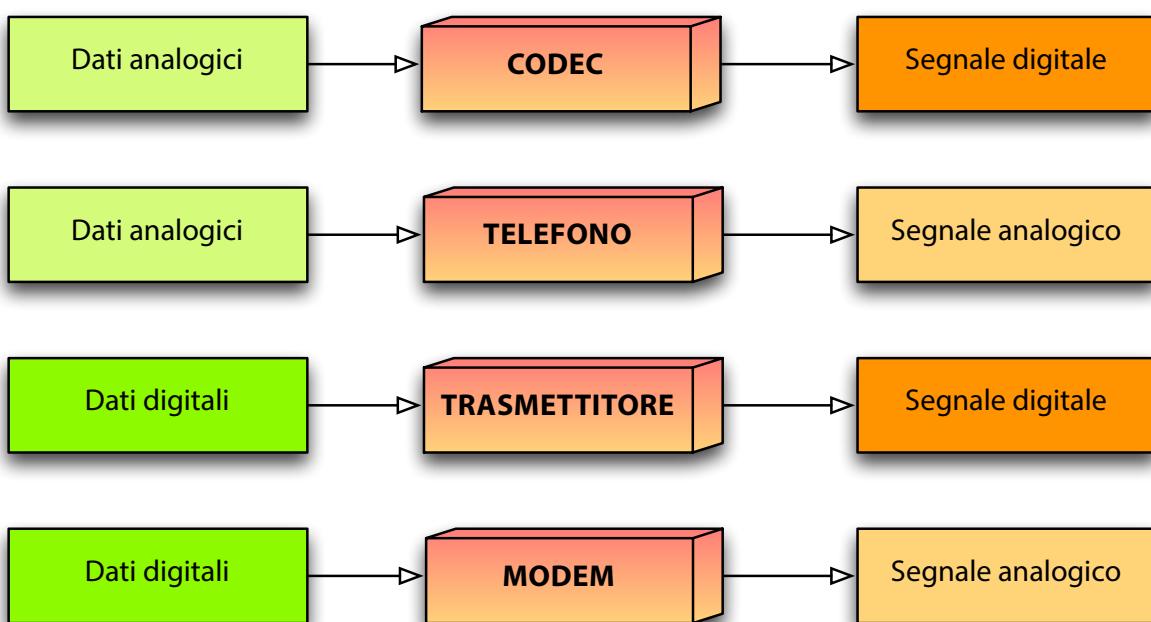
Il segnale è una variazione temporale di una grandezza misurabile (ad esempio il voltaggio). Chiaramente i segnali, così come i dati, si dividono in analogici e digitali (a seconda del dato che comunicano).

1.3) Il mezzo trasmissivo

È il supporto fisico che permette la propagazione del segnale. Anche la trasmissione può essere analogica oppure digitale.

2) La trasmissione tramite segnali

2.1) Dai dati al segnale



Il **CODEC** trasforma i dati analogici in segnale digitale, ovviamente con perdita di informazioni (minima). Questa fase è detta COdificazione. Quando il segnale arriva a destinazione viene DECOdificato e quindi torna ad essere dato analogico.

Il **MODEM** MOdula il dato digitale rendendolo analogico, ed una volta giunto a destinazione viene DEModulato per tornare digitale.

2.2) Come trasmettere i segnali

Una volta che il dato è trasformato in segnale, esso viene trasmesso. Esaminiamo le varie casistiche come fatto prima. La trasmissione tiene sempre conto del fatto che il segnale **mano a mano si affievolisce**.

Segnale analogico --> Trasmissione analogica

Il segnale viene amplificato (l'onda aumenta la sua ampiezza) in maniera "stupida", affinché possa propagarsi per altro spazio. Quindi nel caso in cui al **trasmettitore** arrivasse un segnale sporco, anche l'errore verrebbe amplificato.

Segnale analogico --> Trasmissione digitale

Il segnale è **codificato** in **dati digitali**, e vengono propagati grazie a **ripetitori** che a differenza dei trasmettitori **riconoscono** i dati e logicamente li "ricostruiscono" in maniera esatta, quindi ogni errore viene ripulito ad ogni ritrasmissione.

Segnale digitale --> Trasmissione digitale

Come sopra, ma viene evitato il passaggio della codifica.

Segnale digitale --> Trasmissione analogica

Non si usa.

2.3) La scelta del digitale

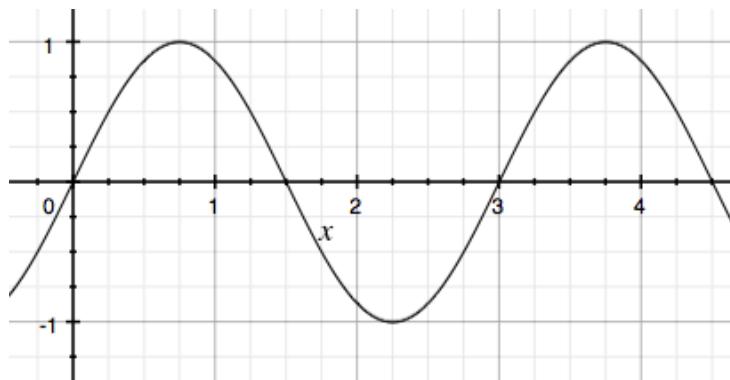
La trasmissione digitale è la più usata fra tutte, poiché ha un basso costo di implementazione e permette l'uso di ripetitori intelligenti che non propagano l'errore.

Risulta inoltre possibile l'utilizzo di crittografia e l'implementazione di un'unica rete di trasmissione per voce e per dati.

3) I segnali periodici

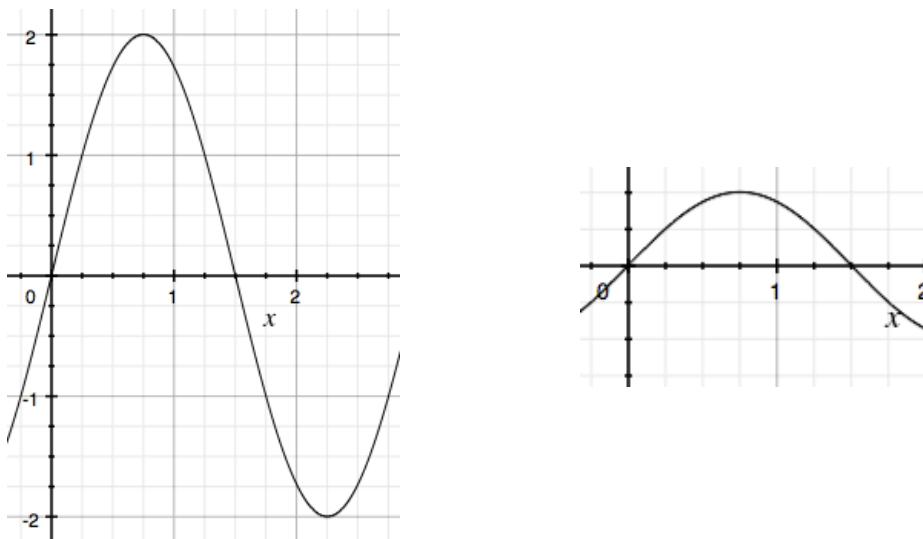
3.1) Il segnale periodico e le sue grandezze

Un segnale è periodico solo se $s(t + T) = s(t)$. T viene detto **periodo**. Ecco una funzione sinusoidale:



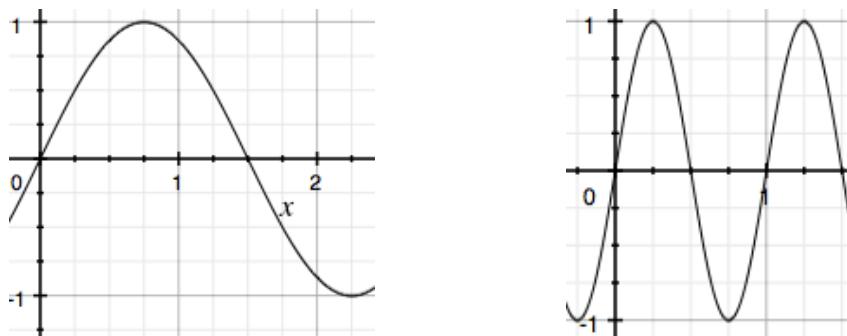
L'ampiezza

L'ampiezza è il valore **massimo** che un segnale può assumere. Siamo interessati all'ampiezza poiché c'è una relazione tra l'energia del segnale (quanto si propaga) e la sua ampiezza $E \propto A^2$. Vediamo sotto una curva con ampiezza 2 e poi una curva con ampiezza 0.5.



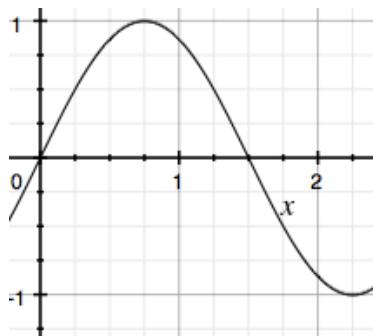
La frequenza

La frequenza è $f = 1/T$ cioè è **inversamente proporzionale al periodo**. Rappresenta quanto frequentemente si ripete il segnale. Vediamo sotto un segnale con frequenza 1 e poi con frequenza 1/3.



La fase

La fase di un segnale è l'istante di tempo (dall'inizio di un periodo) in cui il segnale assume il valore zero. L'unità di misura sono i radianti. Qui sotto, ad esempio, la fase è $\Phi = 0$ (poiché nel punto 0 la curva è a $x = 0$).



3.2) Relazione fra data rate e frequenza

Sappiamo bene che il nostro obiettivo è quello di trasmettere i bit. Per fare questo, possiamo pensare di rappresentarli con una curva quadrata. Un impulso positivo rappresenta il bit 1, un impulso negativo il bit 0.

Avendo quindi T come **periodo** posso trasmettere due bit in un solo periodo. Ogni impulso (bit) dura quindi $T/2$ e dato che $T = 1/f$ (la frequenza) ogni impulso ha la durata di $1/(2*f)$.

Giacché il **data rate** è "il numero di bit per secondo" (**bps**) ed avendo definito la frequenza come "variazioni al secondo", possiamo sostenere che il data rate è **2f bps** (ogni T abbiamo due bit, quindi se $T = 1/f$, $T/2 = 1/2f$ ovvero $2f = 1/T$)

Sfruttando il teorema di Fourier ("qualunque funzione sufficientemente periodica può essere ottenuta sommando un numero idealmente infinito di seni e coseni di opportuna frequenza, ampiezza e fase che chiamiamo **componenti armoniche**") possiamo aggiungere le componenti armoniche 1, 3, 5 (dispari) all'onda e ottenere così, approssimativamente, un'onda quadra.

Chiaramente all'aumentare delle funzioni sovrapposte, aumenta la precisione all'onda quadra.

Lo spettro

Quando il segnale è periodico, lo **spettro** (ovvero l'energia fornita da ogni frequenza) è composta da righe: una per ogni componente.

Quando il segnale non è periodico, lo spettro è continuo quindi l'energia è fornita da tutte le infinite frequenze.

Dunque, volendo ottenere un'onda quadra, dobbiamo sommare segnali a diversa frequenza. Ma se non riesco a trasmettere tutti le infinite componenti, potrò trasferirne solo un certo intervallo, chiamato **banda**. E data una certa banda (ovvero un certo range di frequenze che riesco a trasmettere), qual'è il data rate che riesco ad ottenere?

La risposta è dipendente da quanto vogliamo precisa l'onda quadra.

Supponiamo di usare le frequenze f , $3f$ e $5f$ (armoniche). Vediamo alcuni casi.

Caso 1

Consideriamo un'onda quadra con frequenza **f = 1MHz** (quindi $T = 1/f = 1/1.000.000 = 10^{-6} = 1\mu s$).

Per ottenere la **banda** (cioè il range trasmisibile di frequenze) sottraiamo dalla frequenza maggiore la frequenza minore (ovvero $5f - f$, cioè $5\text{MHz} - 1\text{MHz}$) e otteniamo così **4MHz**. Questo accade poiché non vogliamo trasmettere nulla di minore di 1MHz e maggiore di 5MHz.

Dato il periodo $1\mu s$, ogni $0,5\mu s$ c'è un bit, e quindi il data rate è di **2Mbps** ($2*f$).

Se ne deduce quindi che con una banda di 4MHz possiamo trasmettere a 2Mbps.

Caso 2

Proviamo a raddoppiare la frequenza.

f = 2MHz (quindi T = 0,5μs).

Per ottenere la **banda** calcoliamo $2*5*10^6 - 2*1*10^6$ e otteniamo così **8MHz**.

Dato il periodo 0,5μs, ogni 0,25μs c'è un bit, e quindi il data rate è di **4Mbps** ($2*f$).

Caso 3

E se riducessimo a due il numero di componenti? Cioè usassimo solo f e 3f? (supponendo che il ricevitore sia in grado di distinguere i bit).

f = 2MHz (quindi T = 0,5μs).

Dato il periodo 0,5μs, ogni 0,25μs c'è un bit, e quindi il data rate è di **4Mbps** ($2*f$).

Per ottenere la **banda** calcoliamo $2*3*10^6 - 2*1*10^6$ e otteniamo così **4MHz**.

Conclusione

- A seconda della capacità del ricevente possiamo scegliere data rate differenti pur mantenendo la banda costante.

- A parità di banda, possiamo adoperare ricevitori più o meno costosi per diminuire od aumentare le componenti armoniche.

3.3) Segnali a banda limitata

È necessario tenere presente due fenomeni naturali sempre presenti: l'**attenuazione** ed il **ritardo**.

L'attenuazione non uniforme ed il ritardo

Mentre il segnale si propaga, si ha una perdita di potenza causata dal mezzo trasmittivo.

Il vero problema è che l'attenuazione **non è uniforme** ciò significa che ogni componente si attenua in modo diverso nel tempo!

L'intervallo di frequenze che sono trasmesse con attenuazione abbastanza uniforme è chiamata **banda passante**.

Esiste poi un valore di soglia, f_c , che corrisponde all'intervallo di frequenze nel quale l'ampiezza è maggiore del 50% del massimo. Più la banda è ampia, più è facile individuare il segnale!

Il segnale arriva **distorto**, ovvero la forma del segnale ricevuto è differente rispetto alla forma del segnale nel momento della trasmissione. Individuiamo diverse cause:

- **Ritardo**: ogni componente ha differente frequenza e quindi si propaga più o meno lentamente nel mezzo di trasmissione. Quindi ogni componente arriverà "sfalsata" (con fase non uguale) al ricevitore. Per risolvere il problema si adoperano tecniche di **equalizzazione** che permettono di rimettere in fase le componenti.
- **Attenuazioni**: maggiore è la frequenza, maggiore è l'attenuazione. Il segnale cambia forma a causa del fatto che ogni componente si attenua in tempi differenti. Anche qui si equalizza uniformando l'attenuazione su tutte le componenti.

L'attenuazione, in particolare, si calcola in **decibel** (dB). $N_{dB} = 10 \log(P_R / P_T)$ dove con P_R si rappresenta la potenza ricevuta e con P_T quella trasmessa dal segnale (in Watt).

Il rumore

Il **rumore** è costituito da segnali estranei che si sommano al segnale originario durante la trasmissione. Ne abbiamo di diversi tipi:

- rumore termico (elettroni agitati all'interno del mezzo trasmittivo).
- rumore di intermodulazione (più segnali con frequenza differente vengono trasmessi: se uno di essi è troppo forte, intacca l'altro ottenendo la somma o la differenza fra i due).
- crosstalk (segnali estranei da altri mezzi trasmittivi).

- rumore a impulsi (impulsi elettrici di breve durata e grande ampiezza che intaccano il sistema trasmissivo).

3.4) Capacità di un canale trasmissivo

La capacità di un canale (anche chiamata date rate, tasso, velocità) si misura in bps (bit per second). Ci poniamo ovviamente l'obiettivo di avere massima data rate sempre rimanendo sotto la soglia di errore per cui il segnale viene intaccato in maniera irreparabile.

Concetti importanti per capire la capacità

- **Bandwidth:** ampiezza di banda (misurata in Hz al secondo), ed è limitata dal mezzo trasmissivo o dal trasmettitore o dal ricevitore
- **Rumore:** quantità media di rumore nel mezzo
- **Error rate:** tasso con cui si verificano gli errori (rilevati e residuali)
- **Costo:** I mezzi trasmissivi aumentano di costo quando aumenta la banda supportata e diminuiscono di costo quando aumenta la quantità di rumore.

Teorema di Nyquist

“Se un canale ha banda B, allora il massimo data rate (chiamato signale rate) è $2B$ ”. Cioè **$C = 2B \text{ bps}$** .

Volendo ottenere una capacità di $2B$ allora un segnale con frequenze non superiori a B è sufficiente per realizzare la trasmissione.

Grazie al teorema sappiamo di certo che trasmettendo un segnale qualsiasi attraverso un filo low-pass (che ignora i valori più piccoli di f) con banda pari a B Hz, il segnale filtrato è ricostruibile prendendo solo $2B$ campioni ogni secondo. Il teorema vale **in assenza di rumore**.

Pertanto, **raddoppiando la banda raddoppia il data rate**.

La versione generale è però questa (con V come numero di livelli): **$C = 2B \log_2 V \text{ bps}$** .

Ogni **livello** è un valore considerabile dal ricevitore. Quindi invece di avere 0 e 1 ($V = 2$) possiamo avere ($V = 4$) e avere quindi 00, 01, 10, 11. In buona sostanza si fanno “gruppetti” di bit e si codificano in livelli.

In questo caso avremmo $C = 2 * 2B$ bps. Avremmo quindi raddoppiato la capacità rispetto a prima!

Teoricamente parlando, sarebbe quindi sufficiente aumentare i livelli V per ottenere capacità grande a piacere. Ma il teorema di Nyquist si basa sul fatto che non vi sia rumore e questo non è evitabile. Il rumore, con l'aumentare dei livelli, diventa sempre più critico poiché riuscirebbe a confondere un livello con un altro se la differenza fra i livelli stessi non fosse sufficientemente ampia. **Non si può ignorare il rumore**.

Teorema di Shannon

È il teorema di Shannon a venirci in aiuto, affermando che la capacità massima teorica (al di là dell'implementazione) di un canale indipendentemente dal numero di livelli è pari a **$C = B \log_2(1+S/N)$** chiamando N la potenza del rumore e S la potenza del segnale. S/N è detto rapporto “segnale-rumore”.

Come già detto nel paragrafo 3.3 il rapporto S/N si misura in dB e si può calcolare come $10 \log_{10} S/N$.

Perciò un canale che ha una banda passante di 3kHz ed un rapporto segnale-rumore di 30dB non può trasmettere più di $3.000 * \log_2(1+1000)$ bps = 30.000 bps.

ATTENZIONE AI LOGARITMI! Il calcolo S/N è in \log_{10} , il teorema di Shannon è in \log_2 .

Esempio tipico

Abbiamo uno spettro fra i 3MHz e i 4MHz ($B = 4 - 3 = 1\text{MHz}$).

$S/N = 251$ (quindi il canale è a **24 dB** poiché $10 * \log_{10} 251 = 24\text{dB}$).

Quindi con Shannon otteniamo che:

$$C = 10^6 * \log_2(1+251) \approx 10^6 * 8 \approx 8 \text{ Mbps.}$$

E se ci chiedessimo quanti livelli ci servono per arrivarci? Applichiamo Nyquist.

$$C = 2B * \log_2(V)$$

$$8 * 10^6 = 2 * 10^6 \log_2 (V) \rightarrow V = 16$$

4) I mezzi trasmissivi

Definiamo due macro categorie di mezzi trasmissivi. I mezzi **guidati** ed i mezzi **non guidati**.

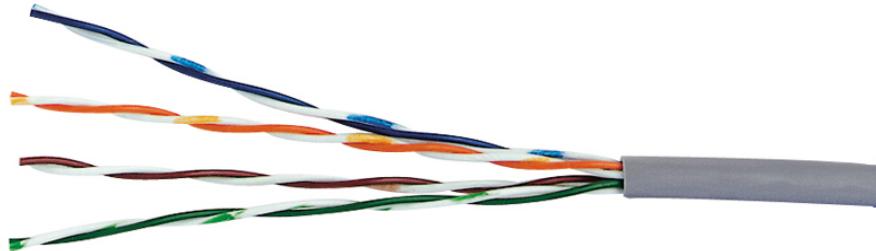
4.1) I mezzi guidati

La qualità di trasmissione è direttamente collegata al tipo di cavo utilizzato. Vediamoli.

Il doppino

Il mezzo più classico, composto da gruppi di coppie di **cavetti di rame** intrecciati fra loro (per evitare l'effetto antenna). I segnali trasportati sono di tipo elettrico e possono essere **digitali** o **analogici** con velocità di alcuni Mbps. L'attenuazione è molto alta, e quindi è necessario un ripetitore ogni 2.3 Km ed un amplificatore ogni 5-6 Km. Dato il suo costo molto ridotto è spesso adoperato nel cosiddetto **ultimo miglio**. Esistono tipi diversi di doppini, con diverse categorie. In gergo tecnico sono detti UDP (Unshilded Twisted Pair).

Il doppino è **poco resistente al rumore**.



Il cavo coassiale

È un **conduttore cilindrico** ricoperto da un isolante di plastica, a sua volta avvolto in una schermatura metallica e infine rivestito da una copertura di gomma per proteggere il cavo.

È un cavo più delicato ma decisamente più **efficiente**. Abbiamo un'ampiezza di banda di circa 1GHz. La **resistenza al rumore** è molto **grande** (molto schermato).



La fibra ottica

Mezzo trasmissivo molto veloce, si basa sulla luce che passa all'interno di un cavo di silicio rivestito di altro silicio (con capacità rifrangenti molto diverse). Si possono raggiungere anche i 10Gbps.

È necessaria una **sorgente luminosa** ed un **rilevatore**: il segnale di luce è codificato come 1, l'assenza di luce come 0. La trasmissione è **unidirezionale** (quindi ci vogliono due cavi).

La luce continua a "rimbalzare" dentro al cavo grazie all'effetto della rifrazione.

Vi sono fibre **multimodali** entro le quali viaggiano a frequenze diverse diversi raggi di luce, e **monomodali** entro le quali viaggia un solo raggio di luce che è praticamente parallelo al cavo (i cavi sono molto sottili). Quindi una fibra ottica monomodale sarà più sottile, delicata e costosa ma l'effetto di attenuamento sarà molto meno presente.

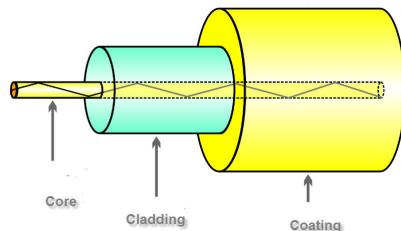
Il grande fattore positivo della fibra ottica è il fatto che, appunto, si attenui molto poco (ogni rimbalzo costa pochissima energia). Un trasmettitore è installabile anche a 50 km dalla fonte e il segnale sarebbe ancora sufficientemente buono da essere recuperato.

Il segnale luminoso può essere **emesso** tramite LED (poco costoso, vita lunga, trasmette a corta distanza) oppure tramite ILD (molto costoso, vita breve, trasmette a lunga distanza).

La fase di **collegamento con l'esterno** è molto importante e delicata. È possibile:

- Adoperare connettori in apposite prese: dispersione del 20% della luce.
- Adoperare connettori meccanici: dispersione del 10% della luce.
- Fondere le due estremità dei cavi: dispersione minima.

Si noti che facendo uso di fibra ottica **non è possibile intercettare** il segnale e si hanno solo **collegamenti punto-punto**.



Mezzi guidati: riassunto

	Fibra ottica	Cavo coassiale	Doppino
Capacità	Centinaia di Gbps su decine di Km	Centinaia di Mbps su 1Km	Qualche Mbps su 1km o 100 Mbps su decine di metri
Peso e dimensioni	Le fibre ottiche sono 10 volte più sottili dei cavi coassiali e di gruppi di doppini		
Attenuazione	Bassa e costante per un ampio spettro	Meglio del doppino, si può usare per frequenze più elevate	Dipende dalle frequenze
Isolamento elettromagnetico	Non necessario	Meglio del doppino	Pessimo
Distanza ripetitori ed amplificatori	Decine/Centinaia di Km	Analogico: 2-3 Km Digitale: ogni Km (dipende dalla frequenza)	Analogico: 5-6 Km Digitale: 2-3 Km
Frequenze utilizzabili	180-370 TeraHz	0-500 MHz	0-3.5 MHz

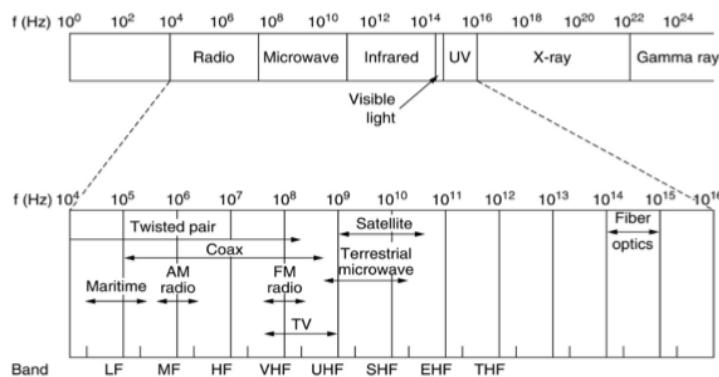
4.2) I mezzi non guidati

Possiamo emettere segnali elettromagnetici nell'aria.

Di queste onde dobbiamo tener conto della frequenza **f** e della loro lunghezza d'onda **λ** (la distanza fra due massimi o due minimi consecutivi).

Nel vuoto, le onde si propagano alla velocità della luce **c = 3x10⁸ m/s**.

Per la relazione di **Le Broglie** si ha quindi che **$\lambda f = c$** . Per i calcoli, si tende ad utilizzare la **f** in MHz, così si ha che **$\lambda f = 300$** . Vediamo ora i vari mezzi irradiati, partendo dalle più basse frequenze e seguendo questo schema:



Le frequenze sono preziose e quindi il governo ne riserva alcune per canali come quelli della polizia, dei telefoni cellulari, ecc. In teoria ogni stato dovrebbe riservare bande equivalenti per servizi equivalenti così da avere un sistema globalmente ordinato (la maggior parte dei governi riservano bande **ISM** (Industrial, Scientific, Medical)).

Un approccio alternativo potrebbe essere quello di imporre una riduzione della potenza: ognuno potrebbe propagare le sue onde in totale libertà poiché la portata sarebbe ridottissima e quindi non potrebbe generare disturbo.

Le onde radio

Le onde radio sono semplici da generare e vengono irradiate in **ogni direzione**.

Sono estremamente correlate alla loro **frequenza**: onde radio di **bassa** frequenza sono in grado di passare oltre gli ostacoli ma si attenuano molto velocemente, viceversa onde radio di **alta** frequenza non passano attraverso gli ostacoli (o addirittura vi rimbalzano) e viaggiano in linea retta. Possono percorrere maggiori distanze. In ogni caso le onde radio sono soggette a **interferenze** da parte di motori e fonti di energia elettrica.

Nelle bande VLF, LF e MF (che variano da 1KHz fino a 1MHz) le onde seguono la linea curva della terra e sono in grado di percorrere 1000 km senza ricevitore. Ignorano gli ostacoli.

Nelle bande HF e VFH invece le onde sono in linea retta, ma rimbalzano contro la ionosfera quindi calcolando correttamente gli angoli è possibile percorrere distanze ancora maggiori rispetto alle basse frequenze.

Le microonde

Le microonde viaggiano praticamente in **linea retta**, pertanto le antenne devono essere correttamente allineate per poter comunicare. Chiaramente con l'aumentare dell'altezza delle antenne, aumenta la distanza ricoperta (perché così si evita la curvatura terrestre), ma ogni 80 km è necessario un ricevitore. Ottime per le **grandi distanze**.

Un grande problema sono gli edifici: le onde rimbalzano e arrivano al ricevitore, ma avendo percorso strade diverse le varie fasi sono sfalsate. Questo problema è detto **multipath fading**.

Se si va sopra i 4GHz, la lunghezza d'onda diventa paragonabile alla dimensioni delle gocce, le quali, quindi, assorbono l'onda e creano grande interferenza.

Infrarossi e onde millimetriche

Piccole onde, usate con **minima potenza** e quindi per distanze estremamente ravvicinate (telecomandi, periferiche).

Non possono attraversare muri.

Onde luminose

Vengono impiegate **sorgenti luminose** che inviano **laser**. Sarebbe l'equivalente delle fibre ottiche, ma con minore sicurezza e soprattutto sono più soggette a interferenze (nebbia, pioggia, ecc.).

Il canale è intrinsecamente univoco quindi c'è bisogno di una coppia di ricettori/trasmettitori in ognuno dei due punti di connessione.

Comunicazioni satellitari

I satelliti sono molto utilizzati per connessioni a **lunghe distanze**. Dalla terra vengono inviate microonde che vengono intercettate dal satellite e vengono quindi restituite sulla terra su una frequenza diversa (altrimenti il satellite interferirebbe con se stesso) ovviamente amplificate.

Vi sono diverse "posizioni" in cui il satellite può essere posizionato. A seconda della distanza, il moto del satellite rispetto alla rotazione terrestre sarà minore, maggiore o uguale e questo comporta una sincronizzazione differenziata fra satellite e sorgente del dato. Più i satelliti sono piazzati "in alto" più sono in grado di coprire superficie terrestre, ma comportano anche un rallentamento della comunicazione (il delay per inviare il dato fino al satellite!).

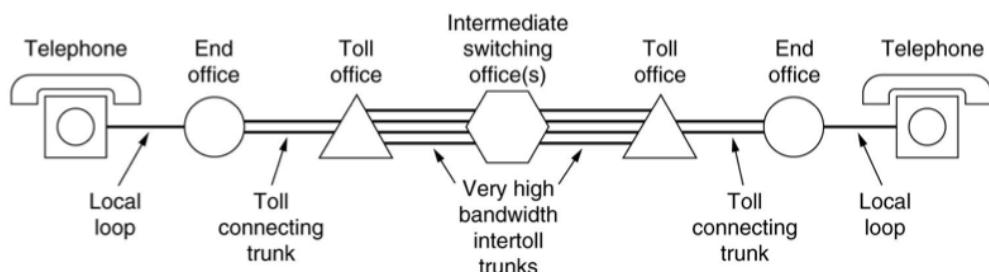
Un satellite può restituire un segnale in due modi: creando un **ponte**, quindi una connessione uno a uno tra due parti del globo (ad esempio per le chiamate internazionali vengono creati ponti fra le connessioni delle due reti telefoniche), oppure può **illuminare** un'intera zona (ad esempio la TV satellitare).

Fibra vs Satellite

Ad una rapida occhiata notiamo subito che il satellite con la sua capacità di illuminare può coprire anche le zone rurali con agilità, cosa che invece non accade con la fibra ottica che dovrebbe essere "portata" direttamente in ogni casa dislocata sul territorio impervio.

4.3) La rete telefonica

La rete telefonica (PSTN - Public Switched Telephone Network) è adoperata, oggi, per la trasmissione dati. Ma nasce come mezzo trasmissivo per dati di tipo analogico (la voce).



Come visibile dall'immagine sopra, la rete telefonica è organizzata in modo gerarchico, cioè ad albero. Ogni nodo deve essere in grado di servire tutti i sottodonodini fino alle foglie.

Evoluzione storica del sistema

L'idea iniziale era un'**architettura totalmente connessa** in cui i due interlocutori comunicavano tramite una centralina con un addetto che collegava direttamente i due cavi.

Si è poi passati ad un'**architettura con un centro di commutazione**. Infine i vari centri sono stati uniti da altri centri di commutazioni creando così un'**architettura gerarchica**.

Il canale telefonico

Abbiamo i collegamenti locali (il famoso **ultimo miglio**) che sono interconnessi tramite le **linee** che portano infine alle **centrali di commutazione**. Sono quindi queste tre le componenti che permettono alla rete telefonica di funzionare.

Sono i doppini a trasportare la voce. I doppini hanno una buona banda, ma in realtà essa viene limitata. Perché? Lo spettro della voce ha un range che va dai 100 Hz ai 3000 Hz e quindi tutto il resto della banda sarebbe solamente rumore.

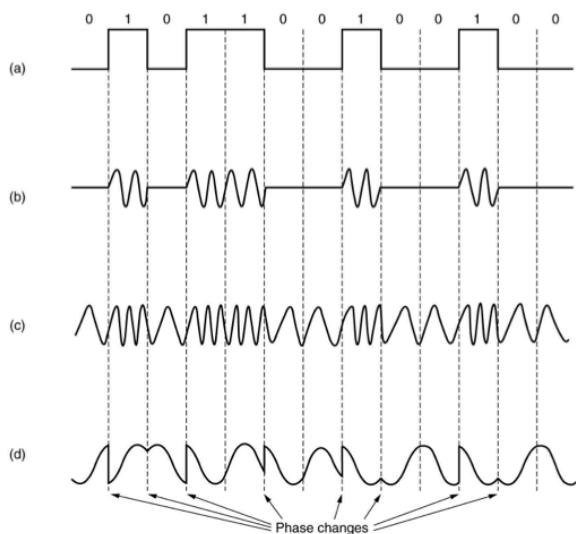
Ma tutto questo era sensato finché la rete telefonica non è stata adoperata per la trasmissione di dati digitali oltreché analogici! Come ovviare a questa situazione?

La modulazione/demodulazione

Dobbiamo immettere in una rete analogica dei dati digitali. Si rende necessario quindi passare attraverso una fase di **modulazione** in cui viene sfruttata un'**onda portante sinusoidale** (con spettro fra i 1000 e i 2000 Hz).

Il segnale di input $m(t)$ viene pertanto combinato con una portante a frequenza f_c generando così un segnale $s(t)$.

Vi sono tre tipologie diverse di modulazione.



a) Il nostro segnale $m(t)$ detto **modulante**.

b) Si applica una **modulazione di ampiezza**, ovvero viene variata l'ampiezza del segnale portante e viene portata a zero dove il segnale modulante è zero.

c) Si applica una **modulazione di frequenza**, ovvero viene variata la frequenza codificando con (ad esempio) il tratto a bassa frequenza il bit zero e il tratto ad alta frequenza il bit uno.

d) Si applica una **modulazione di fase**, ovvero viene variata la fase del segnale portante in maniera "improvvisa". Diversi valori di fase, indicano diversi bit.

Più formalmente:

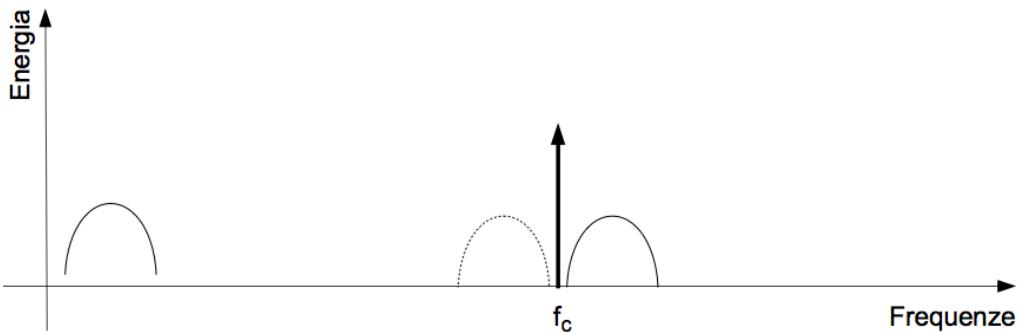
$$\text{Ampiezza: } s(t) = \begin{cases} A \cos(2\pi f_c t), & \text{binary 1} \\ 0, & \text{binary 0} \end{cases}$$

$$\text{Frequenza: } s(t) = \begin{cases} A \cos(2\pi f_1 t), & \text{binary 1} \\ A \cos(2\pi f_2 t), & \text{binary 0} \end{cases}$$

$$\text{Fase: } s(t) = \begin{cases} A \cos(2\pi f_c t), & \text{binary 1} \\ A \cos(2\pi f_c t + \pi), & \text{binary 0} \end{cases}$$

Prendiamo la modulazione d'ampiezza per esempio. L'ampiezza di banda del segnale modulante è solitamente

centrata su una certa frequenza f_c . Pertanto lo spettro del segnale modulato creerà una forma a farfalla, ma con il doppio di banda richiesta... da 0 - 3KHz si sposta a una banda $f_c + 3\text{KHz}$.



Ma non è necessario raddoppiare la banda, poiché, pur tagliando "un ala" della farfalla, è ancora recuperabile il dato (e quindi la banda rimane a 0 - 3KHz).

Il baud

Per ricostruire la sequenza di bit che viene trasmessa, il modem deve **campionare** il segnale ricevuto con una certa frequenza.

Dato il teorema di Nyquist, è inutile avere un campionamento a frequenza maggiore di 6.000 Hz (data la banda che è 3.000 Hz).

I vecchi modem campionavano 2.400 volte al secondo, ovvero venivano letti 2.400 simboli al secondo. Ogni simbolo può assumere uno dei **k** valori assumibili (nel caso dei bit i valori sono due, 0 e 1).

Un **baud = 1 simbolo / secondo**.

Pertanto, una linea a 2.400 baud trasmette/riceve 2.400 simboli al secondo: se ogni simbolo può essere uno zero oppure un uno (due livelli di voltaggio) allora significa che la linea è a 2400 bit/sec.

Se implementassimo altri due livelli (per un totale di quattro) allora avremmo quattro valori assumibili: 00, 01, 10, 11 e quindi con 2.400 baud (un simbolo/secondo) si ha una linea da 4800 bit/sec (ogni s rappresenta due bit!).

Quindi attenzione, un baud non è un bps!

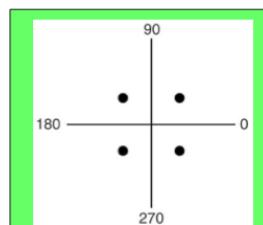
Tecnica QPSK (quadrature phase shifting)

$$s(t) = \begin{cases} A \cos(2\pi f_c t + 45^\circ) & \text{codifica 11} \\ A \cos(2\pi f_c t + 135^\circ) & \text{codifica 10} \\ A \cos(2\pi f_c t + 225^\circ) & \text{codifica 00} \\ A \cos(2\pi f_c t + 315^\circ) & \text{codifica 01} \end{cases}$$

Per codificare più bit per ogni simbolo si usa la modulazione di fase adoperando quattro sfasamenti di $\pi/2$ invece che due di π .

Così si ottengono quattro livelli.

In questo modo si ottiene il **diagramma a costellazione**:



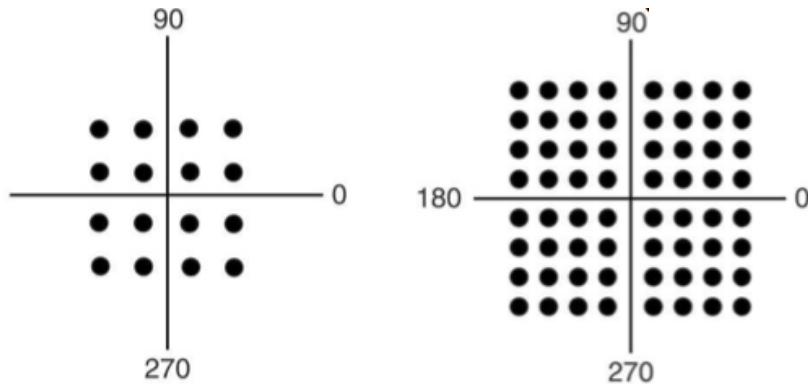
L'**ampiezza** rappresenta la distanza dall'**origine**, mentre la **fasa e l'angolo** ha l'asse X e la linea che collega il punto all'origine. In questo caso le ampiezze rimangono uguali ma la modulazione di fase cambia.

Evoluzioni dell'QPSK: QAM-16 e QUAM-64

Si possono anche estendere le QPSK aggiungendo variazioni di ampiezza e fase in modo da codificare più bit per ogni simbolo.

Con QAM-16 si hanno 16 combinazioni, quindi 16 diversi simboli, quindi si possono codificare 4 bit per ogni simbolo (2^4) pertanto con 2.400 baud si raggiungono i 9.600 bit/sec.

Con QAM-64 si hanno 8 ampiezze e 8 sfasature, quindi 64 combinazioni, quindi 64 diversi simboli, quindi si possono codificare 6 bit per ogni simbolo (2^6) pertanto con 2.400 baud si raggiungono i 14.400 bit/sec.



Ovviamente col crescere del numero di punti aumenta anche la probabilità che ci sia un errore poiché i simboli sono sempre più simili fra loro.

Una linea in due direzioni

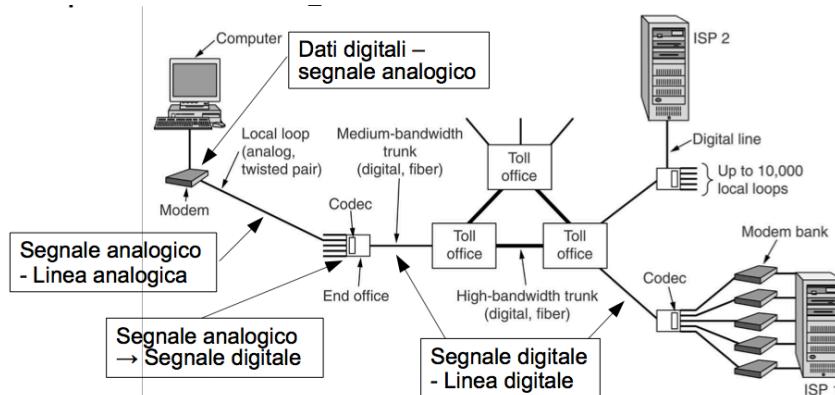
Tutti i modem moderni permettono di trasmettere in due direzioni. Come è possibile?

- Connessione **full duplex** (connessione in entrambe le direzioni simultaneamente)
- Connessione **half duplex** (comunicazione bidirezionale, ogni interlocutore spedisce i suoi dati, uno per volta)
- Connessione **simplex** (comunicazione unidirezionale)

Linee a 56kbit/sec: oltre il teorema di Shannon?

Stando al teorema di Shannon, avendo una banda di 3100Hz e 30dB la velocità limite per il sistema telefonico dovrebbe essere 35kbit/s. Ma allora perché esistono reti a 56kbit/s?

I motivo è che prima della necessità del trasporto di dati (digitali) sulla rete, essa funzionava in maniera interamente analogica. Ma con la necessità di trasportare bit, si è reso necessario un meccanismo di traduzione analogico-digitale-analogico: "l'interno" della rete è fatto interamente da sistemi digitali!



I 35kbit/sec dipendono quindi dalla tratta media in analogico che i dati devono percorrere. Per quanto riguarda la parte digitale si potrebbe arrivare anche a 70kbit/sec!

La linea viene infatti **aumentata** a circa 4.000Hz (senza guardie, cioè intervalli di frequenza lasciati liberi agli estremi).

Ciò significa incrementare la velocità di campionamento a 8.000Hz e quindi **8.000 baud!**

Negli USA vengono adoperati 8 bit/baud, ma solo 7 sono dati mentre uno è di controllo: questo implica una velocità massima di $8.000 * 7 = \mathbf{56.000 \text{ bit/sec.}}$

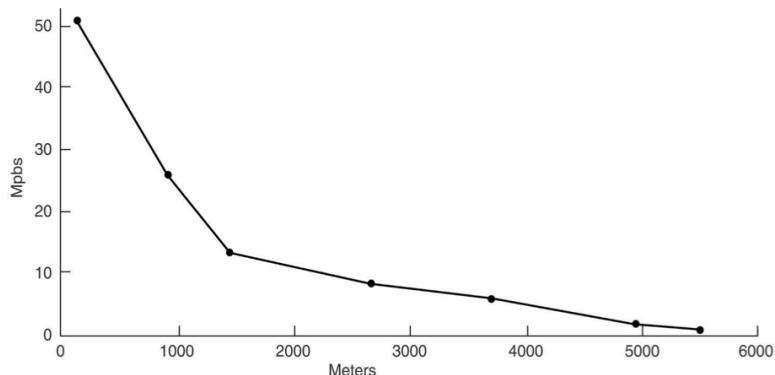
La scelta in Europa, invece è stata quella di adoperare tutti gli 8 bit quindi è possibile trasmettere a 64.000 bit/sec.

Ma per avere uno standard, anche in Europa si è deciso di trasmettere a 56 kbit/sec.

Le DSL

DSL sta per Digital Subscriber Line: una tecnologia concepita per portare la "banda larga" alle utenze residenziali utilizzando il local loop esistente.

L'idea è quella di **eliminare il filtro a 4.000Hz** (usato per la voce) sfruttando di più le capacità del doppino che ha un'ampiezza massima pari a circa 16MHz, permettendo così velocità ancora maggiori.



Si nota subito come la velocità sia dipendente dalla lunghezza del doppino: è molto **sensibile** al **rumore**.

Le velocità previste dallo standard DSL sono in genere più basse del massimo, poiché devono poter funzionare su ogni doppino (anche quelli di bassa categoria) e non devono interferire con le comunicazioni dei fax.

La prima versione (AT&T) prevedeva una suddivisione dello spettro del doppino in tre bande differenti:

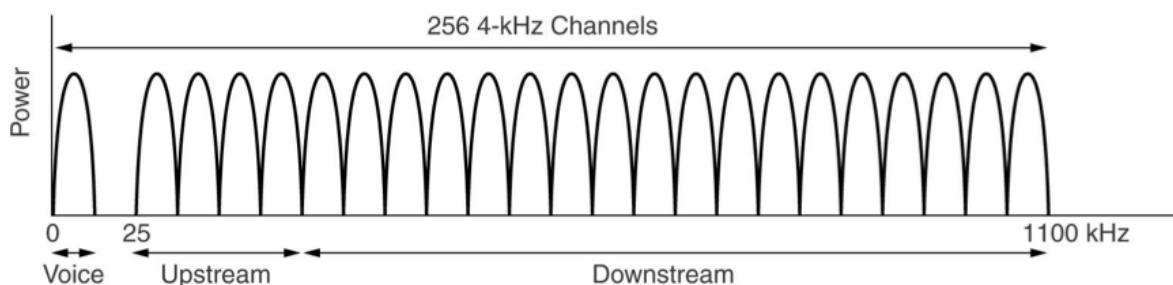
- **POTS** (Plain Old Telephone Service)
- **Upstream**: dall'utente alla centrale
- **Downstream**: dalla centrale all'utente

Le DMT

DMT sta Discrete Multi Tone, una tecnica alternativa alla DSL.

Lo spettro è suddiviso in 256 canali con ampiezza di banda di 4.312 Hz (312 Hz di guardia).

Il canale 0 è usato da POTS, i canali 1-5 non sono usati, due canali sono di controllo, i rimanenti 248 sono a disposizione dell'utente.



La Asymmetric DSL (ADSL)

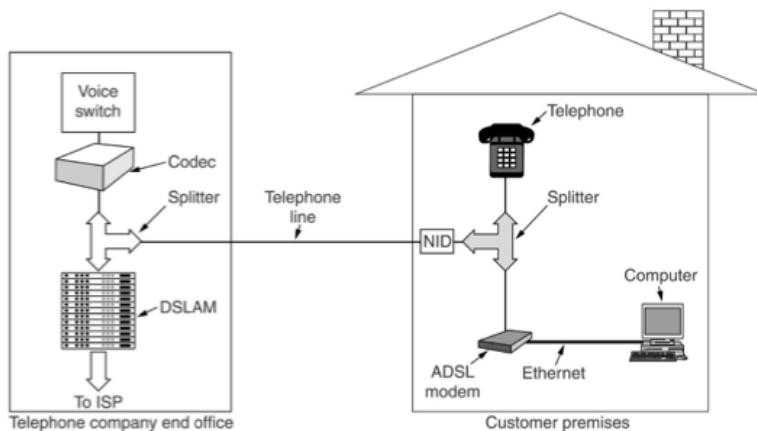
Vengono forniti 32 canali di upstream e 224 di downstream, supporta la velocità sino a 8 Mbit/s in ricezione ed 1 Mbit/s in trasmissione.

Su ogni canale si utilizza uno schema di modulazione QAM con 15 bit per baud, con velocità di campionamento di 4.000 baud.

Perciò, 224 canali, 15 bit per baud -> 13,44 Mbit/s teorici.

Il NID (Network Interface Device)

Presso la residenza viene installato un NID ed uno **splitter** che separa il traffico voce (e fax) da quello dati.



Il modem ADSL è utilizzato per convertire i dati digitali in segnale analogico e viceversa.

Nella centrale, invece, è presente il **DSL - Access Multiplexer**.

Il NID e lo splitter hanno dei costi di costruzione e di installazione notevoli. Per evitarli è stata sviluppata una versione di ADSL che non richiede tali dispositivi. Tale versione è comunque meno efficiente ed adopera dei microfiltrati per simulare l'effetto dello splitter.

xDSL asimmetriche

La velocità in upstream è inferiore a quella downstream. Coesistenza di trasmissione vocale e dati.

Rate-adaptive ADSL (RADSL)

Il modem modifica continuamente la velocità di trasmissione per adattarsi al rapporto S/N della linea, che è variabile da 1 a 12 Mbit/s. Downstream da 128 kbit/s e upstream a 1 Mbit/s.

Very-Hight-Bit-Rate DSL (VDSL)

Da 13 a 52Mbit/s il downstream e da 1.5 a 2.3 Mbit/s l'upstream.

Il tratto a lunga distanza

Le linee a lunga distanza devono anche trasportare i dati trasferiti fra le centrali di commutazione.

Per fare questo (per far colimare tutti insieme i dati in una stessa linea, per altro con comunicazioni di dati differenti) si adopera un sistema di multiplexing. Abbiamo due tecniche principali:

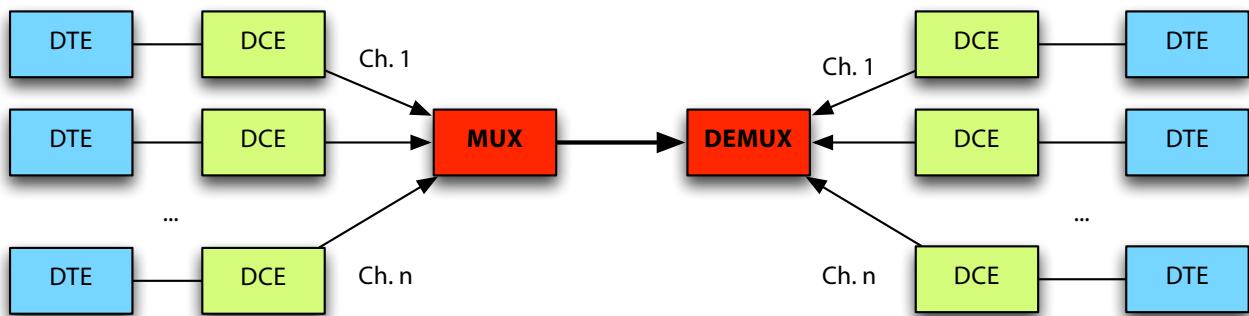
- Frequency Division Multiplexing (**FDM**)
- Time Division Multiplexing (**TDM**)

Per la fibra ottica si usa un tipo di FDM chiamato Wavelength Division Multiplexing (WDM).

Storicamente, prima si è adottata la FDM e successivamente la TDM.

Un canale fisico avente una data (alta a sufficienza) capacità può essere condiviso da più canali dati.

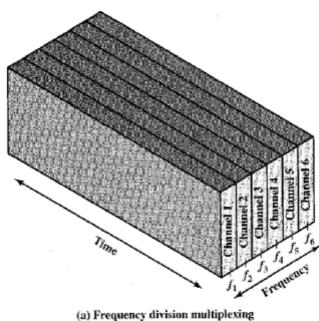
Il **multiplexing** è quella tecnica che permette di suddividere/condividere lo stesso canale fisico tra più canali dati.



Ad un estremo si esegue la funzione di multiplexing (MUX), mentre dall'altro si esegue il demultiplexing (DEMUX). DCE (Data circuit terminating equipment), MUX e DEMUX appartengono al **fornitore di rete** mentre il DTE (Data Terminal Equipment) appartiene al **cliente**.

Quindi il fornitore "arriva in casa", portando il collegamento DTE-DCE localmente!

Approccio al FDM e al TDM

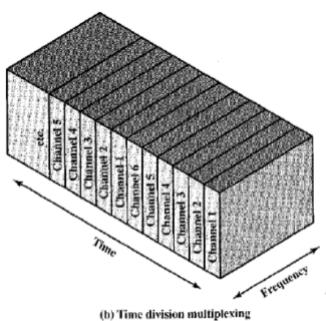


Come il nome suggerisce, la **Frequency Division Multiplexing** si basa sul concetto della suddivisione della frequenza.

Ogni canale che necessita di trasmissione prende una "fetta" della banda fornita dal mezzo trasmittivo. La fetta è definita da un range di frequenza.

Tale banda deve essere quindi ampia a sufficienza per ospitare i range.

Il risultato è che **simultaneamente** vengono inviati dati di **tutti i canali**.



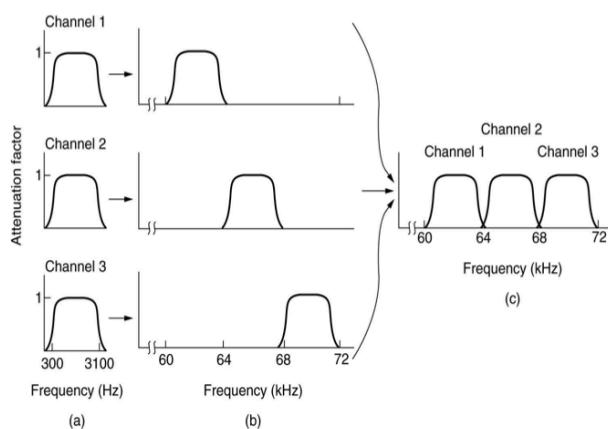
La **Time Division Multiplexing** si basa invece sul concetto della suddivisione del tempo.

Ad ogni canale viene fornito un certo quantitativo temporale entro il quale può occupare tutta la banda per inviare i suoi dati.

Il risultato è che **ciclicamente** vengono inviati dati di **tutti i canali**.

Vediamoli ora nel dettaglio.

Il FDM (Frequency Division Multiplexing)



Ogni canale che vogliamo trasmettere ha una un suo segnale. Il nostro obiettivo è quello di spostare ogni segnale in una zona precisa, in modo da, sommandoli tutti insieme, farli coesistere senza che interferiscano l'uno con l'altro.

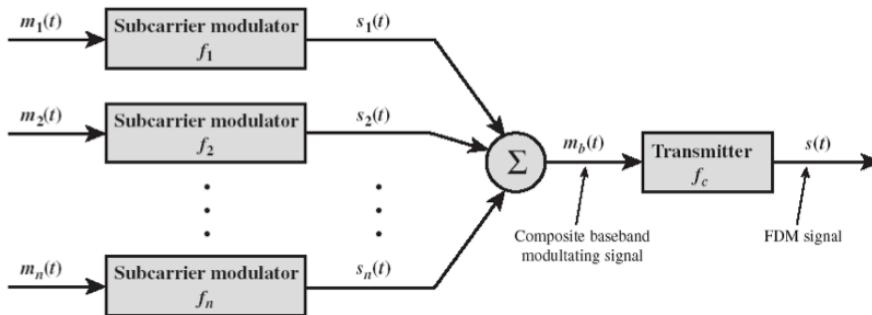
Per fare questo si prendono diverse frequenze f_c portanti e ogni canale viene modulato su una f_c diversa creando così un effetto di "spostamento" del segnale.

Questo sistema è adoperato per la trasmissione **analogica**.

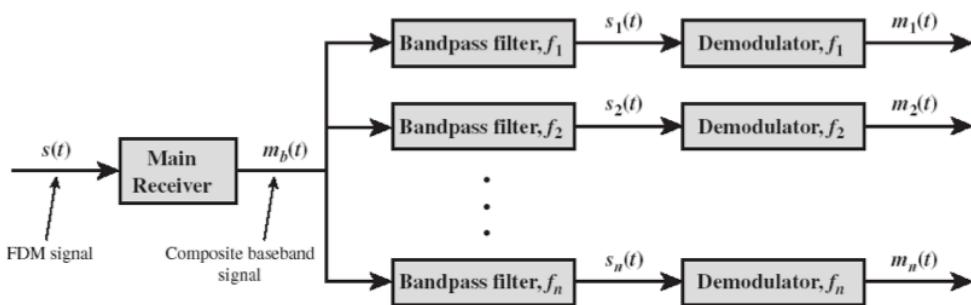
Ogni canale ha un segnale $m_i(t)$ che viene, tramite la portante f_i , trasformato in un segnale $s_i(t)$.

Tutti i segnali $s_i(t)$ vengono fisicamente sommati e trasmessi, ottenendo un segnale totalitario $s(t)$.

Il segnale composto $s(t)$ avrà una banda $B \geq \sum B_i$ dove B_i è la banda di ogni canale iniziale. La somma potrebbe risultare maggiore per via delle bande di **guardia** che sono piccole frazioni di frequenza mantenute vuote per distinguere "meglio" ogni $s_i(t)$.



Nel momento della ricezione, invece, il segnale viene nuovamente scomposto con l'ausilio di filtri **passa-banda** che "ritagliano" ogni $s_i(t)$. Ognuno di questi viene poi demodulato per riottenere la $m_i(t)$.



Il sistema Wavelength Division Multiplexing è analogo nell'ambiente delle fibre ottiche, con la differenza che si adopera la lunghezza d'onda della luce (quindi il colore) per discernere i vari canali.

Il TDM (Time Division Multiplexing)

Questa tecnica è utilizzata nell'universo del **digitale** e ora capiremo perché.

Ciascun canale ha la disponibilità dell'intero canale per un periodo di tempo chiamato **slot**. I dati che devono essere inviati vengono bufferizzati e poi raccolti in **frame** per essere inviati tutti in una volta. Questo è utile soprattutto per i dispositivi che sono lenti, poiché hanno tempo di accumulare i dati prima di inviarli.

Come sappiamo, però, la rete telefonica deve trasportare dati analogici. Si richiede quindi una fase di trasformazione del dato analogico in dato digitale.

Il campionamento: PAM e PCM

Il primo passaggio è il **Pulse Amplitude Modulation**, che campiona il valore del segnale a intervalli regolari. È un campionamento in **tempo**, che viene effettuato a frequenza doppia rispetto a quella della componente più alta del segnale (dal teorema di Nyquist, ad esempio avendo frequenza 4.000Hz si campiona a 8.000 Hz).

Questo tipo di campionamento trasforma il segnale analogico in una "gradinata", cioè a tratti in cui il valore è costante nel tempo e poi salta direttamente al valore successivo.

Viene poi effettuato il **Pulse Code Modulation**, un'approssimazione in ampiezza, che assegna ad ogni gradino ottenuto un livello, codificabile in n bit.

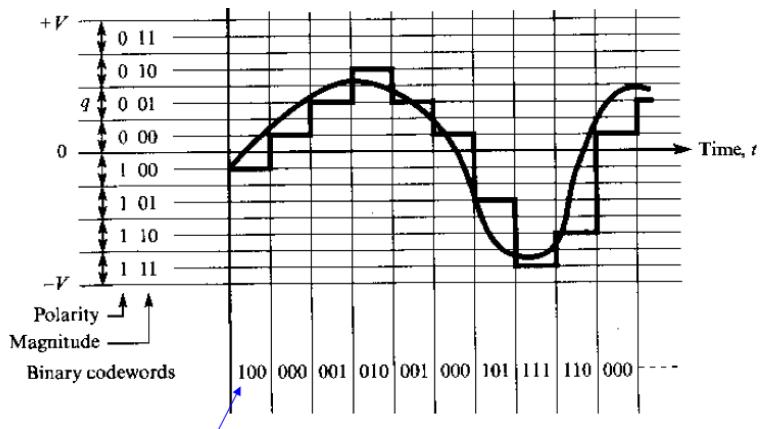
Più esattamente, si prendono 2^n livelli dove con n si intende il numero di bit con cui vogliamo codificare ogni livello.

Quindi si suddivide, in sostanza, tutta l'ampiezza in 2^n zone e per ogni gradino si vede in che zona si ricade.

Ogni zona è numerabile tramite n bit, quindi ogni gradino che ricade in una certa zona è definibile con n bit, quindi alla fine si otterrà una serie di serie di n bit. Ogni n bit definiscono un singolo campione, cioè un singolo gradino.

Tipicamente si adoperano n = 8 bit, quindi si hanno 256 livelli differenti in cui i gradini possono ricadere.

Vediamo un esempio semplificato con n = 3.



Come si vede, ogni blocchetto da tre bit rappresenta un campione (più precisamente in quale degli 8 livelli quel campione ricade).

Il punto fondamentale della trasmissione digitale è il fatto che il dato analogico venga approssimato "di base", ma poi non rischia eventuali errori, poiché il segnale è facilmente rigenerabile.

La codifica dei segnali binari

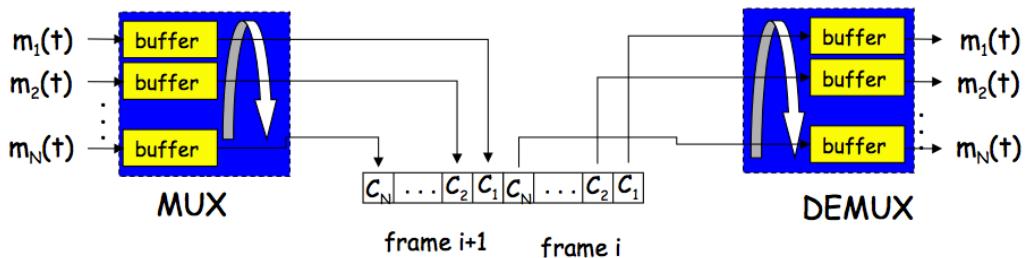
Ma come viene trasmesso sulla rete il segnale binario? Si tengano presenti due fattori: il primo è che i circuiti sono "bravi" nel notare il cambiamento fra livelli di tensione più che la costanza del livello, il secondo è che mantenendo un livello alto di tensione per lungo tempo le cariche si accumulano creando quindi una costante sulla linea che è fonte di disturbo.

NRZ-L		NZR - L: metodo classico, con un certo livello di tensione di identifica l'1, con un altro livello si identifica lo 0.
NRZI		NZRI: con 0 si rappresenta l'assenza di transizione a inizio intervallo, con 1 si rappresenta la transizione all'inizio dell'intervalllo.
Bipolar-AMI (most recent preceding 1 bit has negative voltage)		Bipolar: se non c'è segnale si sta rappresentando lo 0, se invece c'è alternanza fra due valori, allora si rappresenta l'1.
Pseudoternary (most recent preceding 0 bit has negative voltage)		Pseudoternary: duale del bipolare.
Manchester		Manchester: con 0 transizione HL a metà intervallo, con 1 transizione LH a metà intervallo.
Differential Manchester		Differential Manchester : con 0 transizione a inizio intervallo, con 1 nessuna transizione ai inizio intervallo.

ATTENZIONE: come si nota, con il metodo Manchester e Differential Manchester il numero di transizioni (di simboli) aumenta molto, è infatti il doppio: ad ogni intervallo vengono adoperati due simboli. Ma ne vale lo stesso la pena, per agevolare i chip che riconoscono più facilmente la variazione di tensione.

Quindi dato 1 baud si può codificare solamente "mezzo" bit, pertanto il bit rate è la metà del baud rate! Si adopera quindi nel caso in cui la banda sia abbondante.

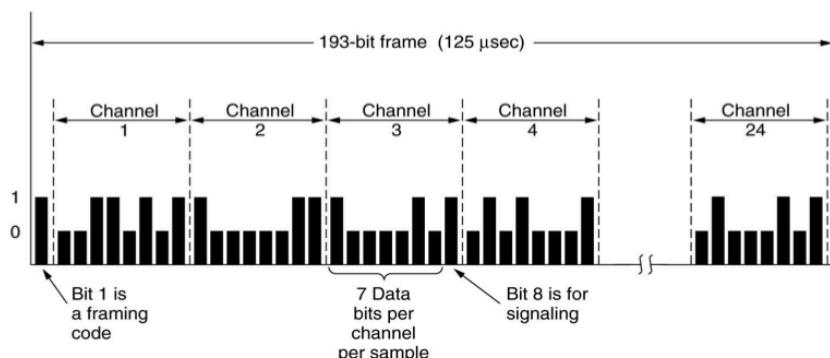
Ancora sul TDM (Time Division Multiplexing)



Come vediamo ogni segnale (suddiviso in bit) viene bufferizzato e viene così raccolto un certo numero di bit.

I buffer vengono quindi svuotati ciclicamente e riempiono il canale, per poi essere riletti dall'altra parte e quindi interpretati.

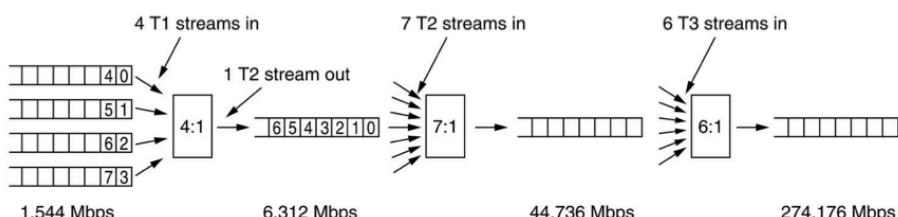
La costituzione di un singolo frame (supponendo che un frame contenga 193 bit) è questa:



Negli USA si adotta la portante T1, composta da 24 canali vocali ognuno dei quali trasporta 8 bit (7 dati e uno di controllo). Quindi, in totale abbiamo 56kbps (campionamento a 8.000Hz, $8.000 \times 7 = 56.000$).

Viene poi aggiunto un ulteriore bit di controllo, perciò, $(24 \times 8) + 1$ bit ogni frame, ovvero 193 bit.

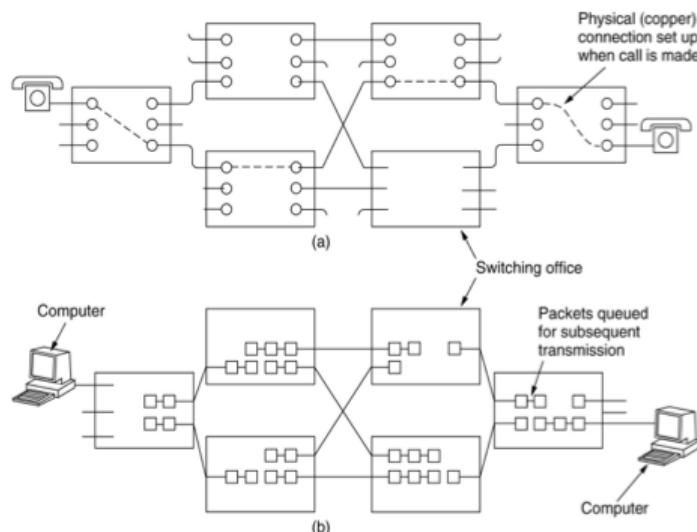
Ognuna di queste portanti T1 viene poi a sua volta multiplexata per generare portanti più ampie.



T1	T2	T3	T4
Sigla	Numero di canali vocali	Data rate (Mbit/sec.)	
DS-1 (T1)	24	1,544	
DS-1C (T1-C)	48	3,152	
DS-2 (T2)	96	6,312	
DS-3 (T3)	672	44,736	
DS-4 (T4)	4032	274,176	

4.4) La rete telefonica: la commutazione

I commutatori (anche detti switch) trasportano i dati permettendo la comunicazione fra gli utenti. Vi sono due tecniche principali, la **commutazione di pacchetto** e la **commutazione di circuito**.



Commutazione di circuito

Le risorse per le chiamate sono **riservate**, allocate quindi nel momento della necessità.

Non esiste il concetto di condivisione poiché le risorse sono dedicate, ovvero si ha una prestazione simile ad un collegamento end-to-end fra i due terminali.

A inizio connessione v'è una fase di **call setup** che verifica la presenza delle risorse necessarie e se disponibili le riserva.

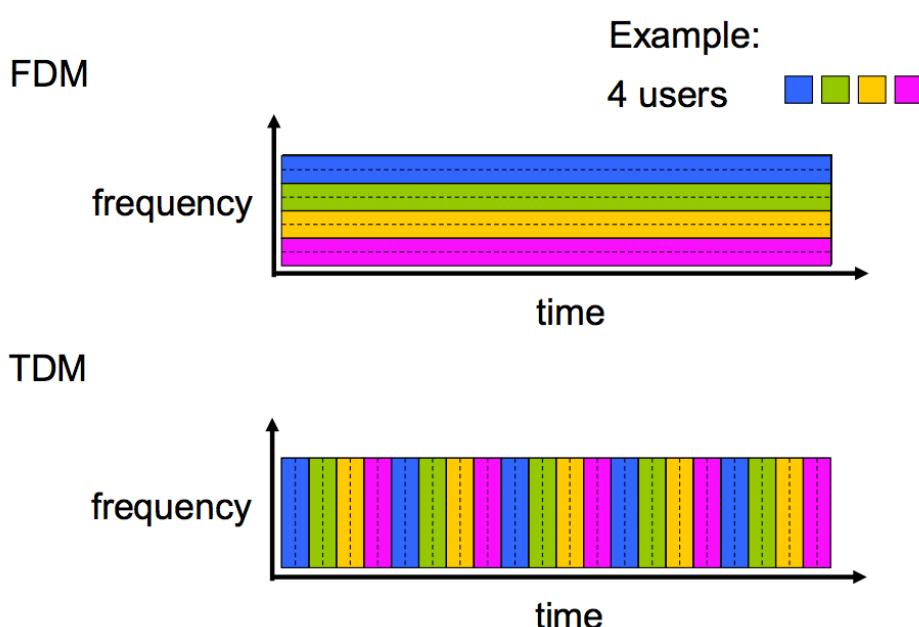
Intuitiva quindi la necessità di una procedura di chiusura della chiamata per rilasciare le risorse.

Le risorse di rete sono suddivise in **trance** (che possono essere bande oppure slot di TDM) che vengono allocate (riservate) in caso di necessità.

Una volta che la risorsa è riservata, anche se non utilizzata, rimane in uno stato di **idle** ma non viene effettuata condivisione!

La qualità è **garantita** poiché, se la comunicazione parte, le risorse ci sono necessariamente.

Qui sotto vediamo l'applicazione delle trance nel caso di FDM e nel caso di TDM (più usato).

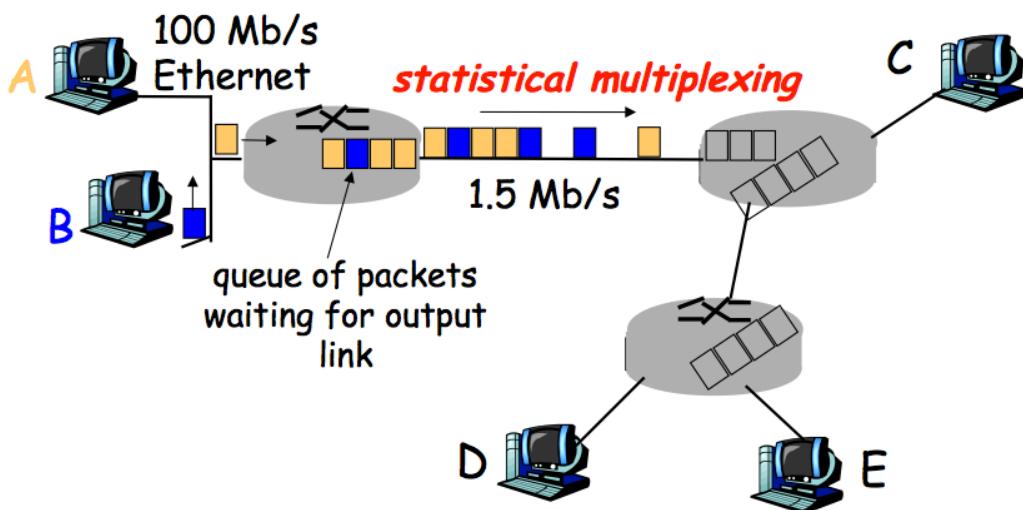


Commutazione di pacchetto

Il flusso dei dati viene **suddiviso** all'interno della rete. Gli utenti **condividono**, ovvero competono, per le risorse della rete. Le risorse sono usate solamente se necessarie, ciò significa poterle sfruttare al 100% senza sprechi.

La qualità non può essere garantita poiché una certa centrale potrebbe essere **congestionata**, quindi i pacchetti potrebbero essere messi in coda in attesa di link.

Deve poi essere applicata la tecnica dello **store and forward**: ad ogni router i pacchetti devono essere ricevuti interamente, immagazzinati e poi rispediti (questo per poter inviare i dati tutti insieme).



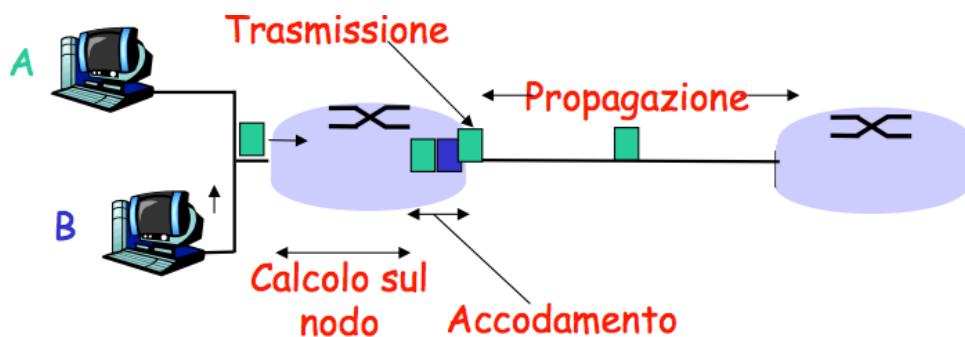
La sequenza che i pacchetti percorreranno per raggiungere il destinatario non è nota.

Nel FDM si dice quindi che si ha un **multiplexing statistico**, ovvero l'intera capacità trasmissiva viene messa a disposizione per la commutazione di ciascun pacchetto.

La commutazione di pacchetto porta con se due problemi intrinseci: le **perdite** ed i **ritardi**. Perché?

I pacchetti devono essere accodati nei buffer del dispositivo di rete, ma il buffer ha una certa capienza, quindi a un certo punto saremo costretti a **scartare i pacchetti** (qualità non garantita!).

Inoltre il tasso di arrivo di pacchetti nel dispositivo di rete può essere maggiore del tasso di partenza. Possiamo in totale delineare quattro fonti differenti di ritardo:



- **Calcolo sul nodo**: controllo degli errori, calcolo del link di uscita, inserimento del pacchetto nella coda di uscita

- **Accodamento**: attesa del proprio turno sul link di uscita (dipende dalla coda, cioè dal livello di congestione)

- **Trasmissione**: Delay (D) = R (banda del link - bps) / L (lunghezza del pacchetto - bits)

- **Propagazione**: Propagazione (p) = d (lunghezza fisica del link) / s (velocità del pacchetto $\sim 2 \times 10^8$ m/s)

Nota: R e s sono chiamate ambedue velocità, ma rappresentano cose molto diverse!

Quindi il *tempo di ritardo su un nodo* è:

$$d_{\text{nodo}} = d_{\text{cal}} + d_{\text{coda}} + d_{\text{trasm}} + d_{\text{prop}}$$

Infine possiamo dire che con frequenza di arrivo di pacchetti a , si ha che l'**intensità del traffico $I = L*a / R$** cioè la lunghezza di ogni pacchetto, per il numero di pacchetti in arrivo, fratto la banda del link (quanti pacchetti può trasportare).

Con $I \sim 0$ abbiamo un ritardo accettabile, con I maggiore o uguale a 1 abbiamo un forte ritardo, eventualmente infinito (significa che arrivano più pacchetti di quanti possano essere smaltiti!).

Si noti che i pacchetti non vengono persi in causa di un errore, ma vengono oculatamente scartati, poiché non riescono a stare nel buffer!

Se ne deduce quindi che, in totale, la commutazione di pacchetto è ottima per dati a **burst** (molti dati in poco tempo) poiché sfrutta al massimo le abilità di non spreco. Inoltre l'assenza di call setup semplifica notevolmente la gestione della connessione. È sempre da tenere da conto, però, il problema della congestione.

Reti di Elaboratori

Il livello data link

Capitolo 3

Enrico Mensa,

Basato sulle lezioni del prof. [Franco Sirovich](#)

1) La mansione del livello data link	1
1.1) L'incapsulamento (header e trailer)	
1.2) Il flusso di dati	
1.3) Tre tipi di servizi	
1.4) Gli errori percepiti dal network	
1.5) La posizione fisica del livello data link	
2) La suddivisione in frame	2
2.1) Conteggio dei caratteri (character counting)	
2.2) Flag Byte	
Stuffing	
2.3) Bit stuffing	
3) La gestione degli errori	4
3.1) Il NACK e l'ACK + Timer	
3.2) Rilevare gli errori: distanza di Hamming	
Esempio: il bit di parità	
3.3) Correggere gli errori: distanza di Hamming	
Il minimo numero di bit necessari	
3.4) Correggere gli errori: codice di Hamming	
Generare la codeword	
Decodificare la codeword	
Gli errori a burst	
3.6) Rilevare gli errori: tecniche di parità	
3.7) Rilevare gli errori: Cyclic Redundancy Check	
La codifica polinomiale	
4) Protocolli data link	8
4.2) Il simplex elementare	
Il codice	
I problemi del simplex elementare	
Tutta questione di controllo di flusso	
4.3) Protocollo simplex stop-and-wait	
Ipotesi per il corretto funzionamento	

Il codice

Valutare l'efficienza: l'utilizzazione del canale

Conclusioni sul simplex stop-and-wait

4.4) Protocollo stop-and-wait per canali rumorosi

I numeri di sequenza

Alcune dichiarazioni di supporto

Il codice

Conclusioni sul stop-and-wait per canali rumorosi

4.5) Protocolli sliding window: introduzione

L'idea di base

Un semplice esempio

La finestra del sender

La finestra del receiver

Gestione del rumore: go back N vs selective repeat

I tipi di frame di riscontro

4.6) Protocolli sliding window: 1-bit sliding window

Il codice

Un protocollo robusto

Pipelining

4.7) Protocolli sliding window: go back N con controllo del livello network

Il controllo del network

Il codice

La dimensione della finestra

Piggybacking obbligato

4.8) Protocolli sliding window: selective repeat con controllo del livello network

I numeri di sequenza rispetto ai buffer

Corretta gestione del piggybacking

Il codice

Nessuno è perfetto

4.10) Attesa non deterministica - metodi alternativi alla wait

5) Esempio di protocollo data link: HDLC

31

5.1) La struttura del frame: introduzione

Architettura master - slave

5.2) La struttura del frame: il campo control

Frame di informazioni

Frame di supervisione

Frame senza numero

6) *Esempio di protocollo data link: PPP* 34

6.1) PPP: LCP e NCP

6.2) PPP: i requisiti

6.3) PPP: i non requisiti

6.4) Scenario di funzionamento

6.5) Struttura dei frame

6.5) Diagramma di flusso: la vita del protocollo

7) *Esercizi sul livello data link* 36

7.1) Esercizi sul framing

7.2) Esercizi sul codice di Hamming

7.3) Esercizi sulle velocità di trasmissione

1) La mansione del livello data link

Si vuole creare una comunicazione che sia efficiente ed affidabile tra due macchine.

I **servizi** il livello datalink offre sono:

- Un'interfaccia utile e ben fatta per il livello network
- Gestire gli errori di trasmissione (**error control**)
- Regolare il flusso di dati (**flow control** - se non ci sono abbastanza buffer, i pacchetti vengono scartati e devono essere ritrasmessi!)

L'illusione è quella di fornire un collegamento diretto con l'altro livello network, ma in realtà sappiamo bene che passiamo tra vari livelli fra cui il fisico.

1.1) L'incapsulamento (header e trailer)

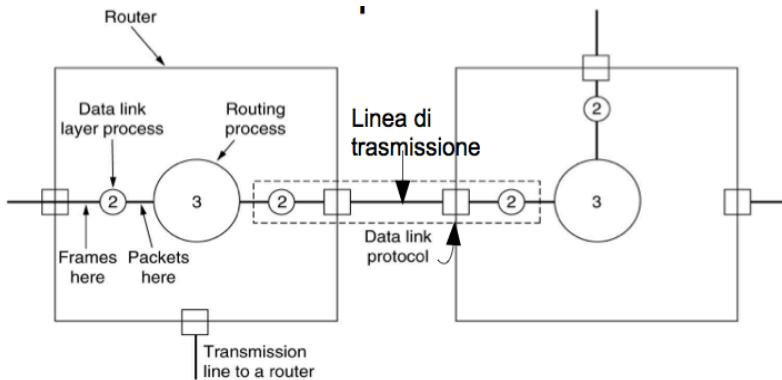
L'header e il trailer vengono aggiunti per fornire informazioni extra al messaggio. L'**header** contiene informazioni che il ricevitore deve avere prima di leggere i dati, mentre il **trailer** trasporta informazioni che sono utili solamente in seguito alla lettura del dato (ad esempio codice per correzione degli errori).

Quindi ogni pacchetto che passa per il livello data link viene munito di header e trailer, venendo così rinominato **"frame"**.

1.2) Il flusso di dati

Il flusso di dati può avvenire fra:

- Entità di livello diverso (sulla stessa macchina)
- Entità dello stesso livello (su macchine diverse)
- Trasferimento di dati "continuo" nel tempo



1.3) Tre tipi di servizi

Il livello data link può adottare tre diverse politiche per fornire il suo servizio.

In sostanza è scelta dell'implementatore come distribuire i compiti "da fare" fra livello network e livello data link. Possiamo quindi distinguere tre varianti:

- **unacknowledged senza connessione**, ciascun frame viene inviato al destinatario in maniera indipendente dagli altri (non c'è ordine quindi) ed il destinatario non invia alcuna conferma del fatto di aver ricevuto un frame.
- **acknowledged senza connessione**, ciascun frame viene inviato al destinatario in maniera indipendente dagli altri (non c'è ordine quindi) ma viene aggiunto un **frame di riscontro** durante la trasmissione per la correzione degli errori (ovviamente questo frame non verrà passato al livello network).
- **acknowledged con connessione**, i frame vengono inviati in maniera ordinata (dopo averli numerati). Viene inoltre aggiunto il frame di riscontro.

Non esiste un "migliore" fra i tre, dipende dall'uso che vogliamo fare!

1.4) Gli errori percepiti dal network

Come abbiamo detto è possibile che vi sia un fallimento durante la trasmissione. Il livello data link comunicherà l'errore al livello network. Possiamo distinguere quattro tipologie di errore:

- **Omission failure** - il frame non è ricevuto
- **Value failure** - il frame ricevuto contiene errori
- **Replication failure** - il frame è stato ricevuto più volte
- **Timing failure** - il frame è ricevuto con ritardo o in ordine erroneo (magari l'ordine è errato per via di una ritrasmissione)

Relativamente a questi errori le tre filosofie di servizio viste sopra si comportano in questo modo:

- unacknowledged senza connessione: fast and dirty (veloce ma con molti errori)
- acknowledged senza connessione: viene ridotta l'omission (c'è il frame di riscontrol) ma aumentano i timing e i replication failure (vengono effettuate più ritrasmissioni)
- acknowledged con connessione: l'ordine è garantito!

1.5) La posizione fisica del livello data link

Questo livello si trova frapposto fra il livello fisico ed il resto della macchina. I dispositivi di controllo del livello fisico sono ovviamente molto vicini al mezzo trasmittivo, ed è qui che si piazza un chip per la gestione del livello data link. Perché questa scelta? Proprio perché abbiamo bisogno di componenti hardware per gestire i servizi forniti dal data link.

I livelli superiori fino all'application sono gestiti invece via software dal sistema operativo, tranne il livello application che è gestito al di sopra del SO.

2) La suddivisione in frame

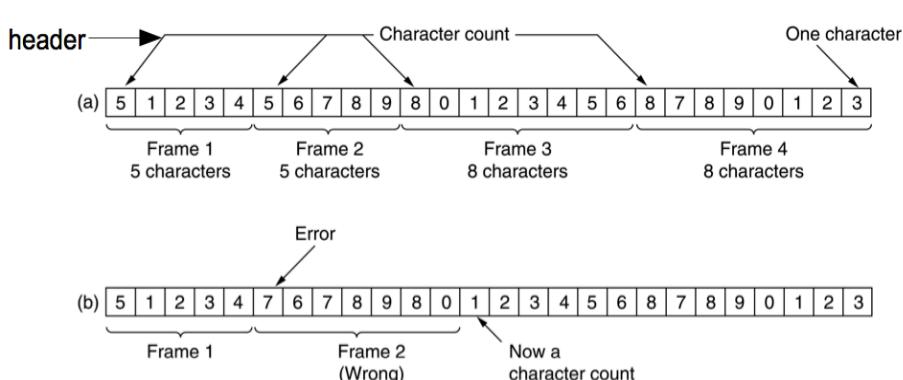
Il livello data link riceve un flusso di bit direttamente dal livello fisico. Dopodiché viene effettuata l'importantissima operazione di suddivisione in frame, ovvero i bit vengono "raggruppati" in sezioni precise.

La scelta dell'utilizzo dei frame è dettata dal fatto che è più **facile capire gli errori**, e che pacchetti troppo grossi rischierebbero di **"affogare" il ricevitore**.

Non resta che capire come si definisce un frame. Potremmo pensare di inserire una pausa fra un frame e l'altro, ma poi sarebbe complicato definire il concetto di "pausa" stesso. Vediamo quindi delle tecniche ad hoc per suddividere i frame.

2.1) Conteggio dei caratteri (character counting)

Si riservano alcuni bit dell'header per dire quanto sarà lungo il frame. Il numero in realtà non rappresenta il numero di bit che seguono, bensì il numero di caratteri rappresentabili con i bit che seguono, quindi i bit sono otto volte il numero indicato, dato che un carattere è codificabile in 8 bit.



ATTENZIONE

Erroneamente potremmo pensare che nell'head la lunghezza venga rappresentata sotto forma di caratteri, ma invece si tratta di una rappresentazione a 8 bit senza segno.

Con 8 bit rappresentiamo numeri fino a 255. Se invece rappresentassimo il numero '255' in caratteri, avremmo otto bit per rappresentare '2', altri otto per il secondo '2' e poi altri otto per il '5'. Per un totale di ben 24 bit per rappresentare la lunghezza!

Non resta che, come convenzione, scegliere se il numero includa o meno l'head.

Notiamo subito una problematica: un **errore** nella zona indicante la lunghezza del frame potrebbe essere tragicamente **irreparabile** (si iniziano a interpretare male i frame)!

Pertanto questa tecnica è adottata solamente con un livello fisico perfetto.

2.2) Flag Byte

L'idea è quella di introdurre un byte predefinito all'interno del flusso di dati, uno a inizio frame e uno a fine frame. Tale byte viene detto **flag byte**.

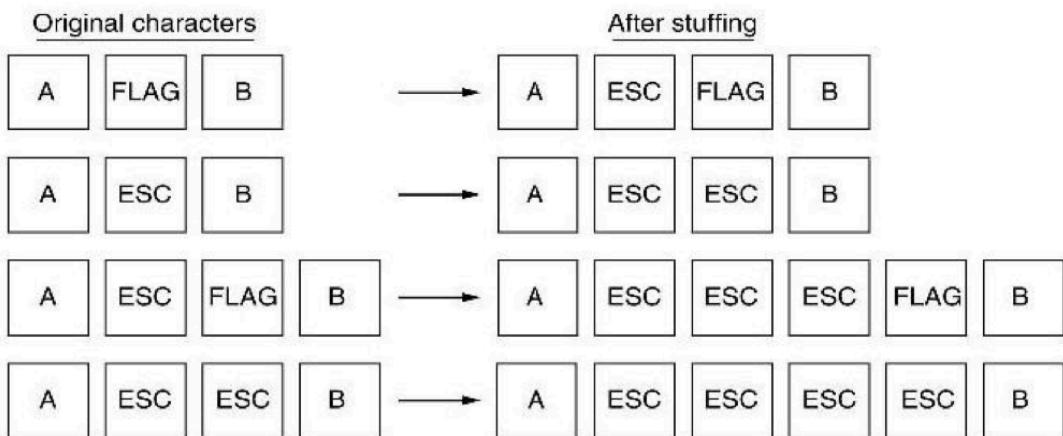
Il punto debole è subito evidente: in caso di trasmissione di bit "sciolti", potremmo trovare la flag generando così un misunderstanding sulla fine del frame! Se stiamo trasmettendo caratteri, invece, il problema non si pone poiché solamente alcune delle sequenze di otto bit rappresentano caratteri quindi basta sceglierne una non utilizzabile per fare da flag. Ma negli altri casi il problema persiste.

Vediamo quindi una tecnica risolutiva per questo problema.

Stuffing

Il termine significa "farcire", che è proprio quello che viene fatto con il frame.

Viene definito un carattere speciale di **escape**, che, se posto davanti ad una flag o davanti ad un altro carattere di escape indica di "leggerlo normalmente" e di non interpretarlo. Vediamo un esempio di stuffing:



Si noti che questo tipo di operazioni vengono effettuate da chip hardware quindi il livello data link, logicamente parlando, potrebbe anche non accorgersi di tutto questo.

2.3) Bit stuffing

Quando trasmettiamo a bit sciolti, il byte stuffing può diventare davvero costoso. Si introduce così il bit stuffing!

Prendiamo ad esempio la flag **0111110** (sei uni di seguito attorniati da due zeri).

Il bit stuffing consiste nell'inserire nell'interezza del frame (tranne ovviamente nella flag) uno zero dopo ogni serie di cinque uni, evitando così la riproduzione della flag (ogni cinque, anche se dopo non c'è un uno)!

In fase di ricezione ovviamente gli uni verranno tolti per leggere il frame normalmente.
Anche qui, il tutto verrà gestito da un chip dedicato.

- (a) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0
- (b) 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0 1 0 0 1 0

 Stuffed bits
- (c) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

3) La gestione degli errori

Prima di tutto è necessario tenere a mente l'enorme differenza fra **riconoscere** un errore e **correggere** un errore.
Una volta riconosciuto l'errore, è possibile cercare di correggere l'errore o almeno di segnalare l'errore alla sorgente per la ritrasmissione.

3.1) Il NACK e l'ACK + Timer

Tipicamente si tende a dare qualche feedback alla sorgente dei dati.

Il **NACK** (acknowledgement negativo) segnala quando un dato ricevuto è errato. Il NACK, però, può essere perso a sua volta, oppure ci può essere troppo rumore per trasmettere il NACK stesso.

Pertanto viene adottato anche l'**ACK** (acknowledgement) con timer, ovvero ogni volta che la trasmissione ha successo viene inviato un ACK di risposta alla sorgente. Se l'ACK non è ricevuto allora la sorgente ritrasmette.

3.2) Rilevare gli errori: distanza di Hamming

Dati **m** bit di dati, se ne vogliono aggiungere **r** di controllo per gli errori.

Perciò la **codeword** sarà lunga $n = m + r$. Con gli **n** bit della codeword, avremmo 2^n possibili parole. Di queste, solo un sottoinsieme lo interpreteremo come **legale**.

Se, nel momento della trasmissione, il ricevitore ottiene una sequenza **non legale** allora c'è stato un errore.

La **distanza di Hamming** fra due codeword legali (**d**) è il "numero di bit che devono essere cambiati per passare da una codeword ad un 'altra", calcolabile tramite lo XOR fra le due codeword (a bit uguali da 0, a bit diversi da 1). Il numero di uno ottenuto dallo XOR è la distanza di Hamming.

$$\begin{array}{r}
 10001000 \\
 10110001 \\
 \hline
 00111001
 \end{array} \quad \left\{ \quad \text{distanza} = 4$$

Se due codeword hanno distanza **d**, allora ci vogliono **d errori** sui singoli bit per renderle identiche. La distanza di Hamming dell'intera codifica è la distanza di Hamming minima fra due codeword (che comunque sono distribuite regolarmente).

Quindi, per rilevare **d errori** sui singoli bit è necessaria una distanza di almeno **d + 1**.

Esempio: il bit di parità

Il bit di parità è un bit aggiunto. La distanza è $d = 2$, cioè le codeword legali differiscono di almeno due bit. Ovvero ci vogliono due errori per far passare una codeword legale in un'altra codeword legale (e quindi non notare l'errore!).

Si potrebbe pensare "allora mettiamo tantissimi bit r!", ma questa sarebbe una deduzione parecchio stupida: aggiungendo molti bit di controllo, si aumentano i bit soggetti ad errore!

3.3) Correggere gli errori: distanza di Hamming

Potremmo pensare... una volta rilevato l'errore, perché non guardare la parola più vicina a quella che ci è arrivata e correggere così l'errore?

Questo è possibile se, anche dopo **d errori**, la codeword legale a **distanza minima** a quella illegale è unica ed è quella di partenza.

Volendo correggere d errori, è necessaria una distanza di Hamming di **$2d + 1$** .

Consideriamo ad esempio una codifica a quattro codeword legali con distanza cinque. Inviamo 00000 11111

0000000000
0000011111
1111100000
1111111111



$$d = 5$$

Qualora il destinatario ricevesse 00000 00111 (errata a causa di due errori), troveremmo che la parola a distanza minima (2) sarebbe 00000 11111, che è la nostra parola inviata, quindi ok!
Ma se invece ricevessimo 00000 00011 (3 errori) correggeremmo con 00000 00000, ed avremmo sbagliato miseramente!

Il minimo numero di bit necessari

Esiste dunque un numero di bit minimi ottimali con cui correggere gli errori? Ovvero, quanti bit di controllo **r** sono necessari volendo correggere **un solo bit** in messaggi da **m** bit?

Per ciascuno dei 2^m messaggi ci sono n codeword **illegali** (infatti è sufficiente, data la parola lunga n, invertire un bit per ottenerne una illegale. Ma essendo appunto lunga n bit, è possibile invertire ciascuno di quegli n bit avendo quindi n diverse combinazioni di codeword illegali).

Quindi, a ciascun messaggio legale corrispondono **n + 1** messaggi (n illegali e uno legale) quindi ogni messaggio necessita di $n + 1$ codeword dedicate.

Poiché fornendo n bit esistono in totale 2^n sequenze di bit generali, si ha che: $(n + 1) * 2^m \leq 2^n$ ovvero tutte le sequenze generabili devono essere maggiori delle codeword dedicate per ognuno delle 2^m parole che vogliamo rappresentare. Sostituendo poi con $n = m + r$: $(m + r + 1) * 2^m \leq 2^{m+r}$ e quindi $(m + r + 1) \leq (2^{m+r} / 2^m)$ e perciò $2^r \geq (m + r + 1)$. Passando ai logaritmi: $\log_2(2^r) \geq \log_2(m + r + 1)$ cioè $r > \log_2(m)$ (per limite superiore).

3.4) Correggere gli errori: codice di Hamming

Sì ok, tutto chiaro, ma esiste questo numero minimo?

Hamming è riuscito a trovarlo ed ha creato un codice che sfrutta il minor valore di r .

Il **codice(n, r)** è un codice di n bit complessivi di cui r di controllo.

Generare la codeword

Prendiamo la parola che vogliamo trasmettere.

- 1) Numeriamo le **posizioni** dei bit da **1 ad n** partendo da destra e andando verso sinistra.
- 2) I bit la cui posizione è una **potenza di due** (1, 2, 4, 8, 16, 32, ecc.) sono i bit di controllo, quindi lasciamo una "x", e distribuiamo i bit della parola da trasmettere nelle altre postazioni.
- 3) Un bit in posizione **k** è controllato dai bit di controllo presenti nell'espansione di k come **somma di potenze di due**. Quindi è controllato da più bit!
- 4) I bit di controllo nelle posizioni di potenza di due fungeranno da bit di parità per quelle posizioni che controllano.
- 5) Ogni bit di controllo è lo **XOR** dei bit che lui controlla.

Esempio

Vogliamo trasmettere, dato un codice di Hamming (11, 4) la parola **1001101**.

- Posizioniamo i dati riservando lo spazio ai bit di controllo

Bit Position	11	10	9	8	7	6	5	4	3	2	1
Bit value	1	0	0	x	1	1	0	x	1	x	x

- Calcoliamo lo XOR fra tutte le posizioni rappresentate in binario che hanno valore "uno" (il risultato è il valore dei bit di controllo in ordine da destra a sinistra)

$$\begin{array}{rcl}
 11 & = & 1011 \\
 7 & = & 0111 \\
 6 & = & 0110 \\
 3 & = & 0011 \\
 & & 1001
 \end{array}$$

- Piazziamo i bit ottenuti nella codeword

Bit Position	11	10	9	8	7	6	5	4	3	2	1
Bit value	1	0	0	1	1	1	0	0	1	0	1

Decodificare la codeword

Supponiamo di ricevere la sequenza **10011100101**.

Calcoliamo lo XOR fra tutte le posizioni rappresentate in binario che hanno valore "uno" corrispondente.

11 = 1011

8 = 1000 La somma è 0000 quindi non ci sono errori.

7 = 0111 Con somma diversa da zero, diciamo 1010 allora avremmo capito che l'errore si trovava nel bit in posizione 10 (1010 è la rappresentazione di 10 in binario).

6 = 0110

3 = 0011

1 = 0001

Gli errori a burst

Spesso gli errori però non si presentano a singoli bit, ma più probabilmente si presentano in larga scala in un breve periodo di tempo.

Char.	ASCII	Check bits	
H	1001000	00110010000	Ma allora con un piccolo marcheggiò possiamo sfruttare il codice di Hamming per correggere gli errori.
a	1100001	10111001001	
m	1101101	11101010101	Raduniamo in blocchi un po' di frame, e poi li trasmettiamo per colonne invece che per righe!
m	1101101	11101010101	
i	1101001	01101011001	
n	1101110	01101010110	Così il frame rovinato dal burst sarà in realtà distribuito fra gli altri.
g	1100111	01111001111	
	0100000	10011000000	
c	1100011	11111000011	
o	1101111	10101011111	
d	1100100	11111001100	
e	1100101	00111000101	

Order of bit transmission

3.5) Rilevare vs Correggere

La correzione viene effettuata su mezzi trasmissivi altamente rumorosi. Perché?

Se ad esempio volessimo trasmettere blocchi da 1000 bit (supponiamo una probabilità di errore molto bassa, tipo 10^{-6}), con il codice di Hamming avremmo 10 bit di controllo quindi in realtà trasmetteremmo 1010 bit. Con i bit di parità, invece, per il semplice rilevamento dell'errore, avremmo 1001 bit da trasmettere.

Volendo trasmettere 1Mbit invece (cioè mille blocchi da mille) avremmo un totale di 1.010.000 bit nel caso del codice di Hamming e 1.001.000 bit nel caso del solo rilevamento.

Ma considerando la probabilità di 1/1.000.000 allora vorrebbe dire che per ogni milione di bit trasmessi viene perso un blocco da mille, e quindi il costo di overhead nel caso di bit di parità sarebbe di 1000 (i mille bit di parità di ogni blocco) + 1001 (il blocco ritrasmesso) = 2001.

2001 bit sono comunque meno dei 10.000 necessari per il codice di Hamming!

Ovviamente con l'aumentare della probabilità i due valori si avvicinano fino a che il codice di Hamming diventa più conveniente.

3.6) Rilevare gli errori: tecniche di parità

La tecnica più semplice per rilevare gli errori è quella del bit di parità anche se la probabilità che un errore passi inosservato è del 50% (poiché non rileva errori su un numero pari di bit caduti in errore).

Tale tecnica può essere migliorata riunendo in blocchi i frame e generando così una matrice di **n colonne e k righe**.

Per ogni colonna viene calcolato un bit di parità e viene aggiunto al fondo, generando così una nuova riga.

Inoltre ogni riga dispone del suo bit di parità.

In ricezione si verificano i bit, se anche solo uno è rovinato, allora il blocco intero viene ritrasmesso.

3.7) Rilevare gli errori: Cyclic Redundancy Check

Le sequenze di bit vengono interpretati come polinomi.

Dati n bit si ha un polinomio di grado n -esimo i cui coefficienti possono essere solamente zero oppure uno.

Ad esempio 10011 è rappresentato da $x^4 + x^1 + x^0$.

L'aritmetica dei polinomi è gestita in modulo 2 (XOR), quindi somma e sottrazione sono la stessa operazione e la divisione è gestita "normalmente" come abbiamo appreso alle elementari.

La codifica polinomiale

Sorgente e destinazione si accordano su un polinomio generatore $G(x)$ avente bit più significativo e meno significativo pari a uno.

Per calcolare il **checksum** (ovvero il risultato finale di somma fra le varie parti di bit di messaggio e bit di rilevazione) si prende il polinomio $M(x)$ rappresentante il messaggio di m bit e, combinato con la checksum (vedi sotto come) si ottiene un polinomio tale che esso **sia divisibile per $G(x)$** . Se così è, non c'è stato errore, altrimenti l'errore è rilevato.

Per fare questa operazione si adotta l'algoritmo:

- posto il grado r di $G(x)$ si aggiungono r bit a valore "0" agli m di $M(x)$. Otteniamo quindi il polinomio $M(x)x^r$ cioè $M(x)$ viene shiftato a sinistra di r posizioni
- **dividere $M(x)x^r$ per $G(x)$ in aritmetica modulo 2**
- la sequenza $T(x)$ da trasmettere si ottiene quindi sommando il resto $R(x)$ con $M(x)x^r$ cioè $T(x) = M(x)x^r + R(x)$ dove $R(x)$ è ottenuto dalla divisione fra $M(x)x^r$ e $G(x)$ (punto sopra).

Se $T(x)$ è divisibile per $G(x)$ anche in ricezione, allora non c'è stato errore.

La presenza di errori è rilevabile tramite un polinomio $E(x)$ i cui coefficienti valgono uno se c'è stato un errore su quel bit.

Perciò nel caso di errore il frame ricevuto sarà $T'(x) = T(x) + E(x)$ e poiché $T'(x)/G(x) = T(x)/G(x) + E(x)/G(x)$, sarà solamente il fattore $E(x)/G(x)$ a indicare un'eventuale errore e perciò se anche $E(x)$ fosse divisibile per $G(x)$, l'errore passerebbe **inosservato**.

A seconda della $G(x)$ scelta (che avrà quindi determinate proprietà) possiamo avere una $G(x)$ in grado di correggere tutti gli errori di un solo bit, piuttosto che tutti gli errori di due bit, oppure ancora tutti gli errori su un numero dispari di bit, oppure errori su burst di lunghezza $k < r$.

Alcune delle $G(x)$ più usate:

- * **CRC-12**: $x^{12} + x^{11} + x^3 + x^2 + x + 1$
- * **CRC-16**: $x^{16} + x^{15} + x^2 + 1$
- * **CRC-CCIT**: $x^{16} + x^{12} + x^5 + 1$
- * **CRC-32**: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^2 + 1$

4) Protocolli data link

I protocolli data link devono svolgere mansioni non facili. Per poter studiare il loro funzionamento, partiremo da un modello molto semplificato (**simplex elementare**) che fa assunzioni pesanti (così pensanti che rendono il simplex una soluzione non reale, un'utopia). Dopodiché, mano a mano, complicheremo la soluzione rendendoci sempre più conto delle problematiche da risolvere.

L'obiettivo è, come sempre, quello del controllo di flusso e della gestione degli errori di omissione.

4.1) Definizioni preliminari

Per scrivere gli algoritmi adotteremo una scrittura C-like, con una gestione ad eventi. Introduciamo quindi un po' di codice che utilizzeremo nei prossimi sviluppi degli algoritmi per i protocolli.

```
#define MAX_PKT 1024                                //Dimensione max del pacchetto

typedef enum {false, true} boolean;                 //Tipo booleano
typedef unsigned int seq_nr;                      //Sequenza o numero di ACK
typedef enum {data, ack, nack} frame_kind;        //Definizione di tipo di frame

typedef struct {                                     //Definizione di pacchetto
    unsigned char data[MAX_PKT];                  //come array di char
}

typedef struct {                                     //Definizione di frame
    frame_kind kind;                            //Tipo del frame
    seq_nr seq;                                //Numero di sequenza
    seq_nr ack;                                //Numero di ACK
    packet info;                               //Pacchetto da livello network
} frame;
```

Abbiamo poi i metodi:

```
//Attende l'avvenire di un certo evento
void wait_for_event(event_type *event);

//Prende un pacchetto dal network e lo fa passare in basso verso il canale
void from_network_layer(packet *p);

//Manda un pacchetto al network
void to_network_layer(packet *p);

//Costruisce un frame prendendo i bit dal livello fisico
void from_physical_layer(frame *r);

//Inserisce nel livello fisico un frame
void to_physical_layer(frame *r);
```

4.2) Il simplex elementare

Le assunzioni che poniamo sono:

- La comunicazione è **simplex** (una entità trasmette sempre, una entità riceve sempre) e non half-duplex / duplex.
- La sorgente non deve **mai attendere di avere dati** da inviare (ne ha infiniti e sempre pronti) e la destinazione ha **buffer infiniti**.
- Le macchine **non si guastano mai**.
- I tempi di elaborazioni sono pari a **zero**.
- Non ci sono **errori** o perdita di frame.

Il codice

Vediamo ora l'algoritmo per il protocollo simplex. Possiamo immaginare due procedure separate (si tratta di una trasmissione simplex).

Questo codice è siffatto solamente perché ci possiamo basare sulle supposizioni sopracitate! Altrimenti non funzionerebbe. Ma lo esamineremo meglio dopo.

```

void sender1(void) {
    frame s;
    packet buffer;

    while(true) {
        from_network_layer(&buffer);
        s.info = buffer;
        to_physical_layer(&s);
    }
}

void receiver1(void) {
    frame r;
    event_type event;

    while(true) {
        wait_for_event(&event);
        from_physical_layer(&r);
        to_network_layer(&r.info);
    }
}

```

I problemi del simplex elementare

Vediamo alcune casistiche che possono creare problemi in questo sistema utopico (iniziamo quindi a smontare parte delle assunzioni utopiche).

- **receiver1 più veloce di sender1** --> attesa in `wait_for_event(&event)`. Ma l'attesa è busy? Il processo tiene la CPU per se durante l'attesa? L'attesa potrebbe essere infinita!
- **receiver1 più lento di sender1** --> in `from_physical_layer(&r)` ci devono essere buffer per accumulare temporaneamente i messaggi in sovraccarico. Ma se la differenza di velocità fosse costante e non momentanea, allora i buffer dovrebbero crescere all'infinito! Questo è ovviamente improponibile, il data link è implementato vicino al livello fisico e quindi ha una scheda dedicata che è certamente povera di buffer. I dati rischiano di essere scartati per assenza di buffer!
- **sender1 è più lento del network sorgente** --> in `from_network_layer(&buffer)` abbiamo che questa volta è il network ad avere bisogno di buffer. L'implementazione è fatta nel sistema operativo quindi abbiamo in effetti più spazio, ma comunque l'effetto potrebbe propagarsi anche a livello transport ed application.
- **sender1 è più veloce del network sorgente** --> anche qui il problema del busy waiting si ripropone (per il data link).
- **il network di destinazione è più veloce del receiver1** --> il network attenderà bloccato nella fase di trasmissione del pacchetto al receiver1.
- **il network di destinazione è più lento del receiver1** --> buffer richiesti in `to_network_layer(&r.info)`, anche questi non possono essere infiniti, e una volta pieni verranno sfruttati i buffer del livello data link il quale non potrà svuotare il suo buffer perdendo così i frame sul mezzo fisico!

Tutta questione di controllo di flusso

Tutte le problematiche viste sopra rientrano nella problematica del controllo di flusso.

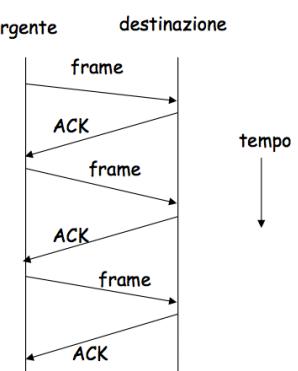
È semplice, il produttore ed il consumatore devono essere regolati: se la velocità del produttore supera un certo limite (prima del limite è possibile attutire con l'aiuto dei buffer) esso deve essere rallentato.

4.3) Protocollo simplex stop-and-wait

Una prima soluzione ad intuito potrebbe essere quella di calcolare il **tempo massimo** che il ricevitore impiega a **processare** un frame, e il produttore, dopo ogni invio, attende quel tempo per inviare un altro frame.

Sembra una buona soluzione ma sarebbe attuabile solamente se il ricevente fosse sempre regolare nella lettura del frame. Ma se pensiamo che talvolta è addirittura il livello applicativo a dover far uso dei frame (quindi l'utente umano che banalmente potrebbe essere andato a prendere un caffè) capiamo che questa soluzione non è realizzabile.

Si sceglie perciò di adottare un sistema ad **acknowledge esplicito**, ovvero dopo ogni trasmissione di frame il ricevente manda un ACK per dire "ho ricevuto e sono



pronto a ricevere ancora".

Questo approccio risolve senza dubbio il problema sopra posto ma certo porta alla luce alcune problematiche di tipo prestazionale.

La **comunicazione** diventa, prima di tutto, **bidirezionale** e questo genera anche **ritardi accumulati** fra le trasmissioni.

Ipotesi per il corretto funzionamento

Affinché questo metodo funzioni sempre, dobbiamo dare per scontate alcune cose:

- Il **canale è perfetto** (non c'è rumore)
- Il **network è sempre pronto** (a dare e ricevere dati)
- La **capacità** della destinazione può essere **finita** (basta un buffer)

Eventualmente se il network dovesse essere intasato, con la funzione `to_network_layer(&r.info)` bloccante possiamo anche risolvere la casistica del network a capacità limitata.

Questo approccio per risolvere il problema del controllo di flusso è chiamato **back pressure**, ovvero è il più lento a costringere la comunicazione al suo livello, quindi, metaforicamente, per evitare di essere spintonati in coda siamo noi a dare una spallata all'indietro.

La tecnica back pressure fa quindi sì che la velocità di trasmissione sia quella del più lento e quando uno dei due in attesa, quell'attesa impedisce all'individuo di compiere azioni. L'utente quindi si rende conto che la macchina è "piantata".

Il codice

```
void sender2(void) {
    frame s;
    packet buffer;
    event_type event;

    while(true) {
        from_network_layer(&buffer);
        s.info = buffer;
        to_physical_layer(&s);
        wait_for_event(&event); //Attendo un ACK
    }
}

void receiver2(void) {
    frame r,s;
    event_type event;

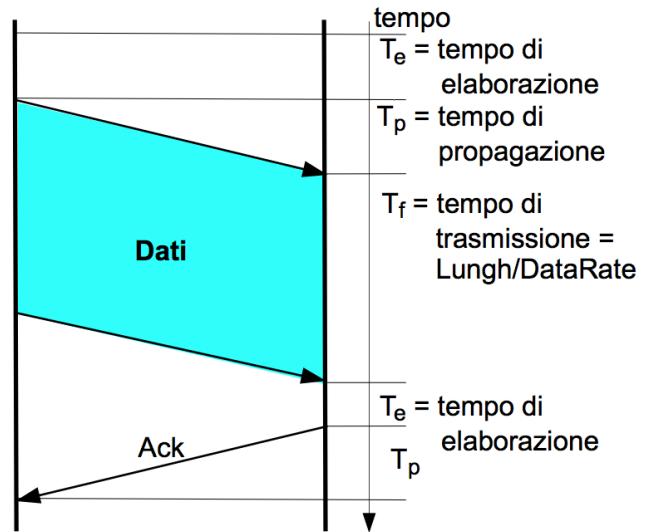
    while(true) {
        wait_for_event(&event);
        from_physical_layer(&r);
        to_network_layer(&r.info);
        to_physical_layer(&s); //Mando l'ACK
    }
}
```

Come vediamo è stata introdotta una linea per ogni codice. Nel caso del sender in seguito all'invio ci metteremo in attesa di un ACK, nel caso del receiver in seguito alla ricezione inviamo un ACK.

Valutare l'efficienza: l'utilizzazione del canale

Per cercare di capire quanto le nostre soluzioni siano più o meno efficienti, è necessario introdurre alcuni concetti per valutare l'utilizzazione del canale.

- Con **utilizzazione del canale U** si intende il rapporto di tempo in cui il mezzo fisico è usato per la trasmissione dei dati (solo dei dati!).
- Con **tempo di lavoro T_e** si intende il tempo necessario affinché un calcolatore elabori un frame.
- Con **tempo di trasferimento T_f** si intende il tempo impiegato dal frame per passare fra le due parti, quindi è il tempo che intercorre tra la partenza del primo bit dal sender e la ricezione dell'ultimo bit dal receiver.
- Abbiamo poi che $T_t = T_p + T_f$ con **T_p tempo di propagazione e T_f tempo di trasmissione**.
- T_f è calcolabile come L/R cioè come (lunghezza del frame) / (data rate).



Nel caso quindi del protocollo simplex stop-and-wait abbiamo che il tempo totale di una interazione è (non consideriamo il T_f dell'ACK essendo praticamente nullo):

$$T_{\text{tot}} = T_e + T_p + T_f + T_e + T_p = 2T_e + 2T_p + T_f$$

Ma l'unico tempo adoperato per la trasmissione è T_f e quindi abbiamo che (tempo utile / tempo totale):

$$U = T_f / (2T_e + 2T_p + T_f)$$

Conclusioni sul simplex stop-and-wait

In conclusione, quindi, questa tecnica è realizzabile ma inefficiente e viaggia alla velocità del più lento fra gli attori, senza contare che ignora totalmente i problemi del livello fisico (rumore) e pertanto è **irrealistica**.

4.4) Protocollo stop-and-wait per canali rumorosi

Cercando di rendere più realizzabile il protocollo stop-and-wait incappiamo immediatamente nel problema del rumore: possono esserci errori di trasmissione.

Un primo approccio può essere quello di **introdurre un timer**, che scatta nel momento dell'invio del frame. Se il frame è danneggiato il destinatario non invierà l'ACK e dopo un certo tempo il timer scatterà avvertendo il sender del problema di trasmissione appena occorso.

A questo punto il sender provvederà a ritrasmettere il frame.

Attenzione! Se l'errore di trasmissione avesse colpito l'ACK allora il receiver avrebbe un duplciato... se ne deve rendere conto!

I numeri di sequenza

Per far sì che i duplicati si "notino" viene introdotto un numero di sequenza. Nel caso di stop-and-wait abbiamo solamente un tipo di ambiguità: quella fra un frame ed il suo successore. Quindi ci bastano due valori per identificare un'ordine. Ovviamente i valori che sceglieremo sono quelli che ci occupano meno spazio e quindi '0' e '1'. Quindi con **un bit nell'head** si identifica il numero di sequenza.

Questi protocolli sono detti PAR (Positive Acknowledgement with Retransmission).

Alcune dichiarazioni di supporto

Introduciamo alcuni metodi che adopereremo in seguito.

```
//Fa partire il timer e abilita l'evento di timeout
void start_timer(seq_nr k);

//Ferma il timer e disabilita l'evento di timeout
void stop_timer(seq_nr k);

//Fa partire un timer ausiliare e abilita l'evento di ack_timeout
void start_ack_timer(seq_nr k);

//Ferma un timer ausiliare e disabilita l'evento di ack_timeout
void stop_ack_timer(seq_nr k);

//Permette al livello network di generare un evento network_layer_ready
void enable_network_layer(void);

//Vieta al livello network di generare un evento network_layer_ready
void disable_network_layer(void);

//Una macro che incrementa un valore k di in maniera circolare
//(una volta arrivata a MAX_SEQ torna da capo)
#define inc(k) if(k < MAX_SEQ) k = k + 1; else k = 0
```

Il codice

```
void sender3(void) {
    seq_nr next_frame_to_send;
    frame s;
    packet buffer;
    event_type event; // (a)
    next_frame_to_send = 0;
    from_network_layer(&buffer);
    while(true) {
        s.info = buffer;
        s.seq = next_frame_to_send;
        to_physical_layer(&s);
        start_timer(s.seq);
        wait_for_event(&event); // (b)
        if(event == frame_arrival) {
            from_physical_layer(&s);
            if(s.ack == next_frame_to_send) {
                stop_timer(s.ack);
                from_network_layer(&buffer);
                inc(next_frame_to_send);
            }
        } // (c)
    }
}
```

Vediamo il codice del sender: non è così banale.

Come possiamo vedere, al punto **(a)** abbiamo l'init del numero di sequenza e del primo frame da inviare (tutto ciò accade fuori dal while!). Dopo di che vengono effettuate le solite operazioni, il timer viene avviato e poi ci si mette in attesa di un evento nel punto **(b)**. Qui l'evento che può arrivare è di tre tipi: frame_arrival, cksum_err oppure timeout. L'if successivo gestisce in realtà tutte e tre le casistiche!

Se arriva un frame_arrival allora l'ACK è stato ricevuto, lo si preleva e se ne verifica il numero di sequenza per capire se è proprio l'ACK che stavamo attendendo (relativo al frame appena inviato). Se così è, allora fermiamo il timer prendiamo un nuovo frame, torniamo a monte del frame, lo inviamo e via così.

Nel caso di cksum_err oppure di timeout (punto **(c)**), dovremo in entrambi i casi inviare nuovamente il frame, quindi saltiamo l'if e torniamo direttamente a inizio while... ottimo! Era proprio quello che volevamo poiché il while reinizializza il frame e lo invia!

```
void receiver3(void) {
    seq_nr frame_expected;
    frame r,s;
    packet buffer;
    event_type event;

    frame_expected = 0;
    while(true) {
        wait_for_event(&event);
        if(event == frame_arrival) {
            from_physical_layer(&r);
            if(r.seq == frame_expected) {
                to_network_layer(&r.info);
                inc(frame_expected);
            }
            s.ack = 1 - frame_expected;
            to_physical_layer(&s);
        }
    }
}
```

Anche qui inizializziamo il frame_expected a zero, in concordanza con quanto fatto con il sender.

L'evento che può arrivare è un frame_arrival oppure un cksum_err. Nel primo caso verifichiamo che non sia un duplicato controllando il numero di sequenza. Se dunque è quello corretto, lo inviamo al network e incrementiamo il numero di sequenza che attendiamo. Se il numero di sequenza invece non è corretto, allora inviamo al sender un ACK con il numero precedente (il sender potrebbe aver inviato nuovamente un frame perché non ha ricevuto l'ACK precedente!), e quindi lo inviamo al fisico.

Nel caso di cksum_err, invece, torniamo semplicemente al while attendendo un altro evento (e lasciando immutato il numero di sequenza atteso).

Conclusioni sul stop-and-wait per canali rumorosi

Abbiamo trattato il problema dei canali rumorosi... ma abbiamo ancora molto da fare!

Il canale viaggia sempre alla velocità del più lento, non sono implementate funzioni di accelerazione in seguito ad un rallentamento (magari vorremmo poter recuperare il lavoro accumulato nei buffer). Si manifestano quindi due necessità:

- Il produttore, per un breve periodo, deve poter andare più veloce del consumatore.
- Bisogna poter inviare un certo numero di frame prima di aver ricevuto l'ACK di quelli già inviati.

4.5) Protocolli sliding window: introduzione

Avendo noi un livello fisico full-duplex, passiamo ad un protocollo data link full-duplex!

Ogni entità è in grado di ricevere ed inviare simultaneamente. Introduciamo ora diversi concetti prima di vedere le implementazioni dei singoli protocolli sliding window.

Abbiamo quindi che ACK e frame viaggiano sullo stesso canale nella stessa direzione... a questo punto potremmo pensare di unirli! Inviamo un frame con l'ACK del frame precedente incorporato.

Introduciamo quindi il **piggybacking**, ovvero si ritarda leggermente la spedizione degli ACK attendendo che il network fornisca qualche frame da trasmettere con cui accoppare l'ACK, e si ritarda anche la trasmissione di dati network attendendo di avere un ACK da accoppare.

La tecnica di piggybacking è ottima per quanto riguarda l'utilizzo del canale, inoltre dimezza il numero di interrupt che arrivano alla macchina receiver.

Per migliorare l'utilizzazione del canale si deve quindi **inviare più di un frame** senza attendere l'ACK.

L'idea di base

Iniziamo a esaminare la questione.

Numeriamo i frame tramite i **numeri di sequenza** i quali saranno lunghi **n** bit (e quindi avremo numeri da 0 a 2^{n-1}).

Possiamo subito notare che quindi lo stop-and-wait è un protocollo sliding window con n pari a uno.

L'idea di base è fondata su questi punti:

- Ogni **sorgente** tiene traccia di quali numeri di sequenza è **autorizzata ad inviare** (l'autorizzazione arriva ovviamente dalla destinazione, che, dovendo ricevere i frame, potrebbe avere non sufficienti buffer e quindi impone un limite massimo al numero di frame che può mantenere contemporaneamente).
- I numeri di sequenza autorizzati sono quindi compresi in un range, detto **finestra di invio**.
- Non tutti i numeri di sequenza nella finestra di invio sono già stati adoperati, potremmo infatti avere frame che il network non ha ancora trasmesso al data link oppure frame che si trovano già sui buffer ma devono ancora essere spediti.
- La **destinazione** ha invece un range detto **finestra di ricezione** che è sempre un range di numeri di frequenza ma questa volta comprende quelli che la destinazione accetterà (anche in questo caso, evidentemente, sarà la destinazione a decidere il range basandosi sulla quantità dei suoi buffer).
- I limiti superiori ed inferiori delle finestre non devono essere necessariamente coincidenti in un certo istante, tranne nel caso in cui ci sia una pausa nella trasmissione: in quel caso le finestre si risistemerebbero sugli stessi limiti.
- Le dimensioni delle finestre sono sempre più piccole dei numeri di sequenza possibili (capiremo meglio perché in seguito).

Per far sì che tutto questo funzioni sono necessari due vincoli fondamentali:

- I pacchetti devono essere trasmessi al network nell'ordine esatto in cui sono stati inviati dal data link sorgente (dall'altra parte) pertanto è necessario **mantenere l'ordine**.
- Il canale fisico trasferisce ottimamente i pacchetti in maniera ordinata ma soffre purtroppo di **perdite e corruzioni**.

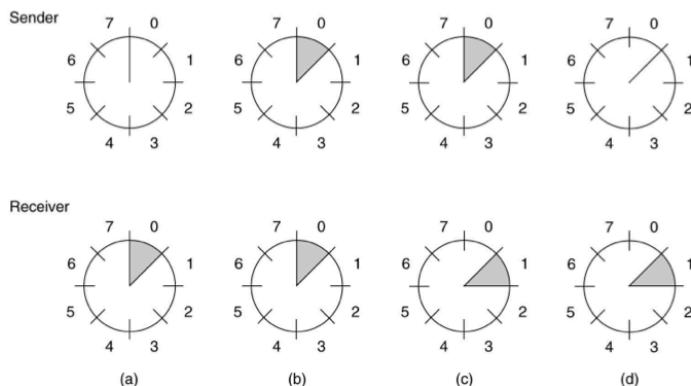
Un semplice esempio

Per capire a pieno il funzionamento del sistema adoperiamo un esempio.

Supponiamo di avere n = 3, quindi numeri di sequenza da 0 a 7.

Supponiamo poi che la send e la receive window abbiano entrambe dimensione 1.

Notiamo che qui la finestra è molto "castrata", infatti il limite superiore ed inferiore saranno sempre coincidenti. Siamo in pratica nel caso di uno stop-and-wait (prestazionalmente parlando) ma implementato secondo questo protocollo.



Il sender

L'immagine sopra descrive lo stato dei buffer.

Il sender inizialmente ha finestra con limiti (0,0) ovvero è autorizzato ad inviare solamente un frame con numero 0.

(a) Il sender ha i buffer vuoti con finestra (0,0)

(b) Arriva un frame dal livello network, viene numerato come 0, e viene inviato. La finestra è completa, (spazio = 1), quindi non si può ricevere altro dal livello network.

(c) Il frame è ricevuto dal receiver il quale invia un ACK, ma questo non cambia lo stato della sorgente.

(d) La sorgente riceve l'ACK e capisce che la trasmissione è andata a buon fine ma soprattutto che è autorizzato ad inviare un altro frame. Inoltre non ha più bisogno di mantenere il frame etichettato come 0 nei buffer, quindi lo rimuove, dopodiché dato che il frame di cui ha ricevuto l'ACK corrispondeva (il suo numero di sequenza) all'attuale inferiore della finestra, quest'ultima viene fatta ruotare cambiandone il range in (1,1).

Il receiver

L'immagine sopra descrive lo stato della finestra.

Il receiver imposta la sua finestra a (0,0).

(a) Il receiver accetta frame che vanno da 0 a 0 (banalmente solo quello che corrisponde esattamente a 0).

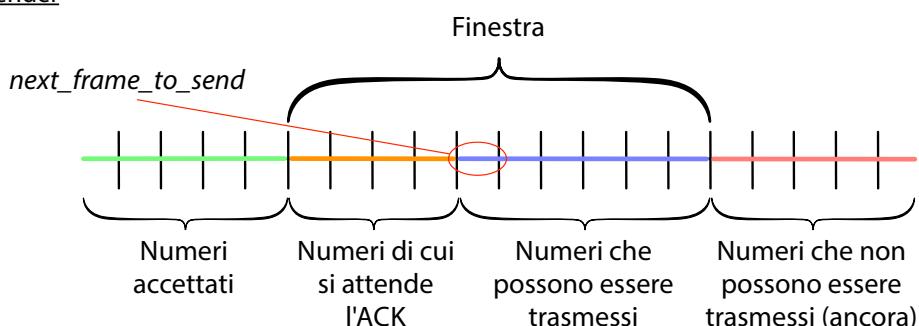
(b) Il frame viene inviato ma non è ancora ricevuto, nulla accade per il receiver.

(c) Il receiver ottiene il frame con numero di sequenza 0, quindi lo accetta, lo prende in carico e lo trasmette al livello network. Invia poi l'ACK al sender ed infine fa ruotare la sua finestra spostandola al range (1,1) attendendo così un nuovo frame.

(d) L'ACK è ricevuto dal sender, nulla accade per il receiver.

Sembra abbastanza regolare, ma questo esempio non mette in evidenza il problema principale dell'implementazione sliding window: se n fosse maggiore di uno allora **potremmo accettare frame non in ordine** corretto, supponendo che nel frattempo alcuni se ne siano persi. Ad esempio con una finestra di dimensione 2 potrebbe andar perso il frame 0 ed arrivare correttamente il frame 1. Il frame 1 sarebbe nel range della finestra e quindi verrebbe accettato... è un problema! L'ordine viene perso.

La finestra del sender

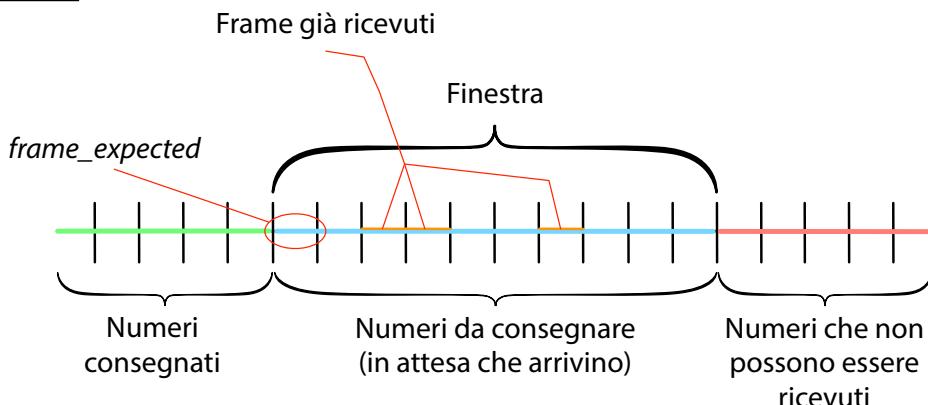


Come vediamo la finestra del sender ha, all'inizio, alcuni numeri di sequenza che sono stati già accettati e che quindi ovviamente non ci aspettiamo di ricevere.

Vi sono poi numeri di cui si attende l'ACK e numeri che possono essere trasmessi (ma i frame magari sono ancora nel network piuttosto che sui buffer in attesa di invio). Tutti questi sono inclusi nella finestra.

Infine, numeri che non possiamo accettare poiché sono oltre il range e non siamo autorizzati ad inviarli.

La finestra del receiver



Ecco che si presenta il problema di cui abbiamo parlato sopra. Abbiamo frame già ricevuti "a groviera" rispetto alla finestra. Che fare? Come facciamo ad avvertire che ci sono stati dei problemi e vogliamo prima gli altri frame?

Gestione del rumore: go back N vs selective repeat

Come gestiamo il rumore? Potremmo perdere frame durante la trasmissione.

La tecnica **go back N**, di più semplice implementazione, prevede che venga preso ed accettato un singolo frame alla volta (il *frame_expected*).

La tecnica **selective repeat**, invece, prevede la ricezione di frame anche non contigui (che dovranno quindi essere immagazzinati in appositi buffer) ma poi l'invio ordinato al network.

Più nel dettaglio:

Dal lato del **trasmettitore** attiviamo un timer ogni qualvolta viene inviato un frame (quindi ogni frame ha un timer proprio). Se non riceviamo l'ACK per un certo frame entro lo scadere del timer, allora procediamo alla ritrasmissione. Ma cosa ritrasmettiamo?

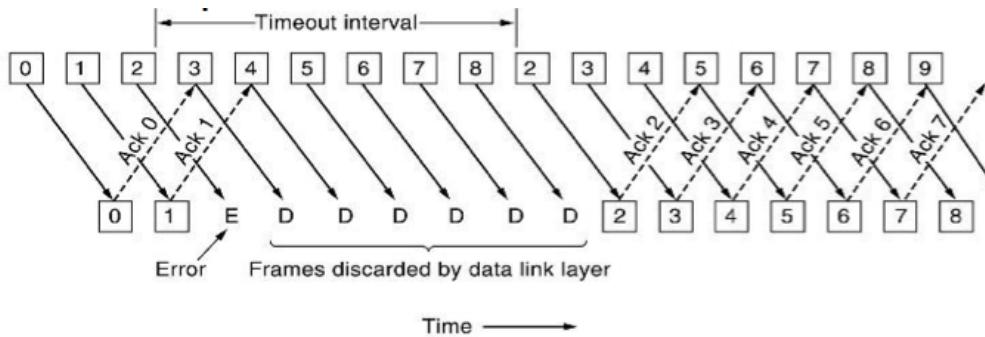
- Se adottiamo una tecnica **go back N** allora ritrasmitteremo tutti i frame inviati a partire da quello di cui non abbiamo ricevuto l'ACK.
- Se invece implementiamo una tecnica **selective repeat** allora ritrasmitteremo solamente il frame di cui è scaduto il timer.

Mettendoci nei panni del **ricevitore** invece, una volta ricevuto un frame:

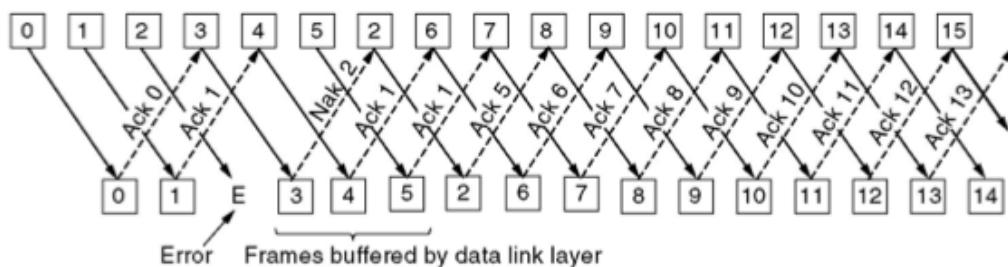
- Nel caso di **go back N** controlliamo che il frame sia il primo atteso, se lo è viene mandato al network e la finestra ruota di uno in avanti.
- Nel caso di **selective repeat** invece verifichiamo che il frame sia contenuto nella finestra e se ci sono frame contigui ricevuti e vengono spediti tutti al network (tutti quelli contigui fino al primo non pervenuto ovviamente!).

Visualizziamo meglio i due sistemi tramite questi schemi (destinatario sotto, mittente sopra):

Il **go back N**, in seguito all'errore smette di inviare ACK. Allo scattare del timeout del mittente, egli ripete tutti i frame a partire da 2, il primo di cui non ha avuto ACK.



Il **selective repeat** si comporta in modo totalmente diverso: il destinatario riceve il frame numero 3 senza aver ricevuto il 2. Si rende quindi conto dell'errore e lo notifica tramite un NACK al mittente il quale provvederà a rispedire (dato che scatterà il suo timer) solamente il 2. Il NACK non è l'unico modo di notificare al mittente la situazione, infatti verrà ripetuto l'ACK dell'ultimo frame ricevuto a oltranza, finché non se ne riceverà uno nuovo. Questa tecnica porta con sé alcune problematiche... tra cui la suddetta "groviera". Che fare dei frame 3, 4, 5, ecc.? Fino a quando non ricevo il frame mancante non posso trasmettere al network... Ci servono dei buffer. L'implementazione sarà trattata nell'ultimo protocollo.



I tipi di frame di riscontro

Esaminiamo ora quali frame di riscontro posso inviare "indietro" per comunicare informazioni differenti (a seconda dell'implementazione verrà specificato quale adoperiamo).

- **ACK individuale - ack(n)**: un singolo frame n è stato correttamente ricevuto e mandato al network.
- **ACK cumulativo - ack(n)**: tutti i frame fino a (n-1) sono stati correttamente ricevuti e mandati al network mentre n è il primo non ricevuto.
- **ACK negativo (NACK) - nak(n)**: il singolo frame n deve essere ritrasmesso.

Si ricorda che si cercherà di adottare l'uso di piggybacking.

4.6) Protocolli sliding window: 1-bit sliding window

Iniziamo con qualcosa di semplice. Abbiamo una window di un bit soltanto, quindi in realtà siamo in un caso stop-and-wait prestazionalmente parlando.

Il codice

```

00 | #define MAX_SEQ 1
01 | typedef enum {frame_arrival, cksum_err, timeout} event_type;
02 |
03 | void protocol4 (void) {
04 |     seq_nr next_frame_to_send;
05 |     seq_nr frame_expected;
06 |     frame r,s;
07 |     packet buffer;
08 |     event_type event;
09 |
10 |     /* inizializzo e mando il primo frame*/
11 |     next_frame_to_send = 0;
12 |     frame_expected = 0;
13 |     from_network_layer(&buffer);
14 |     s.info = buffer;
15 |     s.seq = next_frame_to_send;
16 |     s.ack = 1 - frame_expected;
17 |     to_physical_layer(&s);
18 |     start_timer(s.seq);
19 |
20 |     while(true) {
21 |
22 |         wait_for_event(&event);
23 |         if(event == frame_arrival) {
24 |             from_physical_layer(&r);
25 |
26 |             if(r.seq == frame_expected) {
27 |                 to_network_layer(&r.info);
28 |                 inc(frame_expected);
29 |             }
30 |
31 |             if(r.ack == next_frame_to_send) {
32 |                 stop_timer(r.ack);
33 |                 from_network_layer(&buffer);
34 |                 inc(next_frame_to_send);
35 |             }
36 |         }
37 |         s.info = buffer;
38 |         s.seq = next_frame_to_send;
39 |         s.ack = 1 - frame_expected;
40 |         to_physical_layer(&s);
41 |         start_timer(s.seq);
42 |
43 |     }//fine while
44 | }
```

La prima cosa che notiamo che è che ora il metodo è unico, poiché siamo in un protocollo full-duplex.

In [0] vediamo che il **massimo numero di sequenza** è 1. Perciò i numeri sono {0, 1}.

In [4] e [5] troviamo due **variabili fondamentali**, il primo frame da inviare ed il primo frame che ci aspettiamo.

Seguono poi le solite dichiarazioni di variabili di servizio.

Le righe da [10] a [18] prevedono l'**inizializzazione** delle variabili e l'**invio del primo frame**. Questo evidenzia già un primo problema: se il network non ha pacchetti rimaniamo fermi in [13].

Notiamo che l'ack che viene inviato è compilato in maniera ciclica basandoci sul frame atteso alla riga [16].

Entriamo poi nel while e prima di tutto attendiamo l'arrivo di un evento. A seconda di quale evento abbiamo eseguiremo codice diverso.

Se l'evento è l'**arrivo** di un **frame** [23] allora lo andiamo a prelevare dal livello fisico e ne controlliamo il numero di sequenza [25]. Se il frame arrivato è proprio quello che stavamo aspettando allora lo inviamo al network e ruotiamo la finestra incrementando il valore di frame_expected.

In ogni caso guardiamo l'ACK [31]: dato che stiamo applicando piggybacking vediamo se quell'ACK è quello del frame che stavamo aspettando (negli altri casi è un ACK fuori dalla finestra dato che next_frame_to_send è il limite superiore della finestra di dimensione uno). Se l'ACK è proprio quello corretto, allora stoppiamo il timer e preleviamo un nuovo frame da spedire [33] (la finestra è ora più larga dato che abbiamo ricevuto l'ACK e siamo autorizzati a mandare un nuovo frame) concludendo con l'incremento del next_frame_to_send.

Ad ogni giro **spediamo** qualcosa [37]-[40] (che se è stato ricevuto un ACK sarà un nuovo frame -vedi la variabile buffer- altrimenti sarà la ritrasmissione del precedente).

Un protocollo robusto

In [16] settiamo l'ACK del primo frame della comunicazione a 1, questa cosa ovviamente non ha senso (è il primo frame, non posso mandare ACK di nulla!) ma verrà scartata dalla controparte poiché non facente parte della finestra. Se supponiamo che le due controparti partano in sincronia estrema, avremo che entrambi rifiuteranno l'ACK impostato a 1 ma entrambi manderanno i dati al rispettivo network (aggiungendo questa volta l'ACK corretto). Dunque al giro successivo entrambi invieranno i dati, ma questa volta verranno scartati poiché l'if in [26] risulterà falso ma invieranno l'ACK corretto. A questo punto una delle due controparti riceverà l'ACK del primo frame e procederà con la sua finestra, sbloccando la situazione.

Questo protocollo si dice pertanto **robusto**.

Uso del canale

Prendiamo un canale satellitare con round trip da A a B i 500ms e con capacità di 50Kbps. Vogliamo inviare frame da 1.000 bit.

A inizia a trasmettere a t = 0 e data la velocità del canale, finisce di inviare i dati a t = 20ms (1.000 bit a 50Kbps).

B, dall'altro lato, riceverà l'ultimo bit del frame inviato da A a t = 270ms (250 di metà round trip più i venti di trasmissione) e l'ACK inviato in risposta tornerà indietro non prima di t = 520ms (conclude il round trip, essendo l'ACK minimo non consideriamo i tempi di trasmissione... anche se ci sarebbero i dati nel caso di piggybacking!).

Se ne deduce che A ha trasmesso per 20ms ma è rimasta bloccata per 500ms!

Utilizzazione = 20/520 = 3,8% --> PESSIMO

Nel tempo in cui A aspetta l'ACK del primo frame, potrebbe trasmettere ben altri 25 frame!

Pipelining

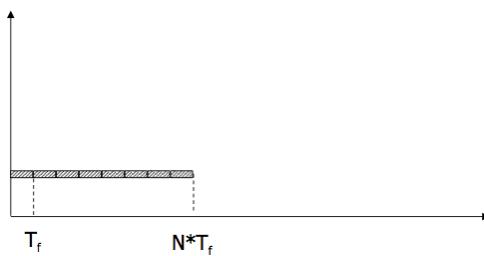
Come abbiamo visto il canale è adoperato in maniera pessima dal protocollo 4.

In realtà il vero ritardo consta nell'attesa del primo ACK, dopodiché la linea sarebbe sempre occupata.

Quindi tanto più la **banda è larga** (piccolo tempo di trasmissione) con **alto ritardo**, tanto più dovrà spedire frame per saturare il ritardo. Quindi l'ampiezza della finestra è proporzionale a **banda * ritardo**.

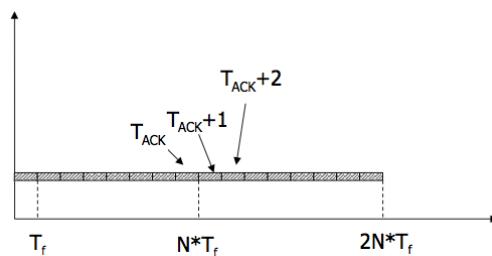
Definiamo con T_f il tempo di trasmissione di un singolo frame e con T_p il suo tempo di propagazione.

Avendo un sender con N frame pronti, finirà di trasmetterli a tempo $T_{TOT} = N \cdot T_f$.



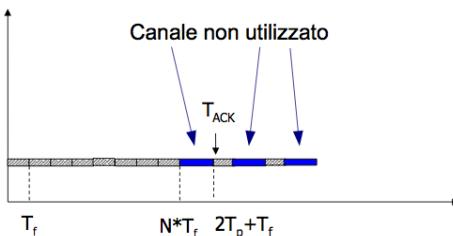
Il primo ACK (per il frame 0) verrà ricevuto a $T_{ACK} = 2T_p + T_f$ (la propagazione del frame, quella dell'ACK e il tempo iniziale da cui è partito).

Se $T_{TOT} \geq T_{ACK}$ allora il mittente riceverà il primo ACK prima di aver terminato di trasmettere tutti i suoi frame e quindi il canale sarà stato occupato al 100%, con **$U = 1$** .



Ma se T_{TOT} non fosse maggiore di T_{ACK} ?

Allora, (ricordando che $T_{ACK} = 2T_p + T_f$ e $T_{TOT} = N \cdot T_f$) il mittente dovrà attendere (facendo nulla) un ACK per l'intervallo di tempo che va da $N \cdot T_f$ a $2T_p + T_f$.



La soluzione? **Aumentare T_f** .

4.7) Protocolli sliding window: go back N con controllo del livello network

Iniziamo a preoccuparci seriamente dei problemi che il protocollo 4 porta intrinsecamente nella sua implementazione. Le ultime due versioni che vedremo supporteranno una la go back N, l'altra la selective repeat (trattati nel capitolo 4.5).

Il controllo del network

Introduciamo inoltre in questa versione un controllo per quanto riguarda il network: se prima lo strato di network non avesse avuto dati da inviare, supponendo un'attesa sospensiva, avremmo avuto uno stallo fra le due parti che non è affatto positivo.

Introducendo l'evento **network_layer_ready** ovviamo a questa situazione: quando il network ha qualcosa da inviare, senda un evento **network_layer_ready**.

Quando i buffer di invio sono però pieni, allora questo evento deve essere **disabilitato**.

Il codice

```

00 | #define MAX_SEQ 7
01 | typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
02 | #include "protocol.h"
03 |
04 | static boolean between(seq_nr a, seq_nr b, seq_nr c) {
05 |     if(((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
06 |         return true;
07 |     else
08 |         return false;
09 | }
10 |
11 | static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[]) {
12 |     frame s;
13 |
14 |     s.info = buffer[frame_nr];
15 |     s.seq = frame_nr;
16 |     s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
17 |     to_physical_layer(&s);
18 |     start_timer(frame_nr);
19 | }
20 |
21 | void protocol5(void) {
22 |     seq_nr next_frame_to_send;
23 |     seq_nr ack_expected;
24 |     seq_nr frame_expected;
25 |     frame r;
26 |     packet buffer[MAX_SEQ + 1];
27 |     seq_nr nbuffered;
28 |     seq_nr i;
29 |     event_type event;
30 |
31 |     enable_network_layer();
32 |     ack_expected = 0;
33 |     next_frame_to_send = 0;
34 |     frame_expected = 0;
35 |     nbuffered = 0;
36 |

```

```

37 |         while(true) {
38 |             wait_for_event(&event);
39 |
40 |             switch(event) {
41 |                 case network_layer_ready:
42 |                     from_network_layer(&buffer[next_frame_to_send]);
43 |                     nbuffered = nbuffered + 1;
44 |                     send_data(next_frame_to_send, frame_expected, buffer);
45 |                     inc(next_frame_to_send);
46 |                     break;
47 |
48 |                 case frame_arrival:
49 |                     from_physical_layer(&r);
50 |
51 |                     if(r.seq == frame_expected) {
52 |                         to_network_layer(&r.info);
53 |                         inc(frame_expected);
54 |                     }
55 |
56 |                     while(between(ack_expected, r.ack, next_frame_to_send)) {
57 |                         nbuffered = nbuffered - 1;
58 |                         stop_timer(ack_expected);
59 |                         inc(ack_expected);
60 |                     }
61 |                     break;
62 |
63 |                 case cksum_err: break;
64 |
65 |                 case timeout:
66 |                     next_frame_to_send = ack_expected;
67 |                     for(i = 1 ; i <= nbuffered; i++) {
68 |                         send_data(next_frame_to_send, frame_expected, buffer);
69 |                         inc(next_frame_to_send);
70 |                     }
71 |
72 |             } //fine switch
73 |
74 |             if(nbuffered > MAX_SEQ)
75 |                 enable_network_layer();
76 |             else
77 |                 disable_network_layer();
78 |
79 |         } //fine while
80 |     } //fine protocollo

```

Come vediamo il codice è parecchio più complesso.

Abbiamo innanzi tutto numeri di sequenza da 0 a 7 (8 totali) ma ne useremo solamente 7. In seguito capiremo perché.

Inizialmente definiamo la funzione di **between** che ci dice se un certo $a \leq b < c$ in modo circolare, lo sfrutteremo per sapere se un sequence number è nella finestra oppure no.

Poi definiamo la **send_data**, funzione che dato un certo numero di frame (lo vogliamo inviare) e quale frame ci aspettiamo (il precedente è quello di cui vogliamo inviare l'ACK) oltre che al buffer da cui prendere il frame procede all'invio al livello fisico del dato più l'ACK effettuando quindi un piggybacking obbligato.

Entrando nel vivo del codice, abbiamo le solite fasi di dichiarazione e inizializzazione, poniamo l'attenzione in particolare a [27] dove dichiariamo un contatore di un array parzialmente riempito che è buffer: quando nbuffed è maggiore della length di buffer, allora dobbiamo dire al network di smettere di inviarci frame poiché non siamo più in grado di mantenerli nei nostri buffer (questo accadrà in [74]).

Come sempre, aspettiamo un evento [38]. Nel caso in cui si tratti di un evento che ci comunica che il **network** è pronto a **consegnare** un **frame** [41], procediamo a metterlo nel corretto buffer (il `next_frame_to_send`) e aumentiamo il numero di buffer occupati. Mandiamo quindi il dato passando come parametro alla `send` il `next_frame_to_send` in modo che il metodo ripeschi proprio il frame che abbiamo appena bufferizzato, dopodiché mandiamo anche l'ACK per il frame precedente al `frame_expected` (di questo si occupa la `send`). Non resta che incrementare il `next_frame_to_send` [45] poiché abbiamo appena inviato il valore attuale.

Se invece l'evento è di tipo **arrivo** di un **frame** [48] allora lo preleviamo dal livello fisico e se la sequenza arrivata è proprio quella che ci aspettavamo allora lo inviamo al network e incrementiamo il lower bound della finestra [51] - [54].

C'è poi un nuovo pezzo di codice che tratta gli ACK.

Gli ACK di cui parliamo sono cumulativi quindi il fatto che `r.ack` sia arrivato indica che tutti gli ACK da `ack_expected` a `r.ack` sono da considerarsi validi. Ecco quindi che [56] svolge proprio questa funzione: finché `r.ack` è nella finestra (eventualmente come lower bound) facciamo avanzare `ack_expected` [59] e fermiamo i timer per quegli ACK. Intanto ovviamente liberiamo i buffer [57].

Nel caso di **errore** [63] semplicemente ignoriamo: il sistema go back N provvederà a farci riavere il frame rovinato dato che non passeremo mai per l'`if` in [51] e quindi non aumenteremo mai il `frame_expected`, ovvero non manderemo mai ACK per i pacchetti successivi, ovvero il timer dall'altra parte scatterà e quindi provvederà ad ad inviare nuovamente i frame.

Nel caso di **timeout** [65] abbiamo che il prossimo frame che dovremo inviare sarà quello di cui aspettiamo l'ACK (ovviamente, dato che è il primo di cui non abbiamo avuto riscontro!), da lì dovremo ritrasmetterli tutti fino a quanto i nostri buffer contenevano (ovvero fino all'interezza della finestra). Ed è quello che accade con [67].

Infine, verifichiamo che se abbiamo ancora spazio per nuovi dati possiamo abilitare il network, altrimenti ne disattiviamo l'evento.

La dimensione della finestra

Come abbiamo detto prima, non possiamo usare l'interezza della finestra. Perché?

Mettiamoci nel caso in cui l'invio di otto frame (da 0 a 7) abbia successo. A questo punto un ACK per il frame 7 (cumulativo!) tornerebbe indietro via piggybacking. La sorgente procede all'invio di altri 8 bit, e riceve nuovamente un ACK per 7.

Tale ACK va interpretato come ACK del nuovo gruppo (cumulativo) o è la ripetizione del vecchio ACK?

Ovvero il destinatario ha ricevuto tutti i frame o nessuno del secondo gruppo?

Dobbiamo quindi tenere uno spazio vuoto per permettere alla finestra di ruotare su un numero in più. Cioè al primo giro la finestra sarà da 0 a 6 (salta il 7), poi dopo un intero giro andrà da 7 a 5 (ma salterà il 6).

Piggybacking obbligato

Un'altra problematica di questo protocollo è il fatto che il piggybacking sia costretto: non è possibile inviare un ACK da solo! Quindi nel caso in cui il traffico non sia bidirezionale, il sistema si impalla (no ACK = ritrasmissioni).

4.8) Protocolli sliding window: selective repeat con controllo del livello network

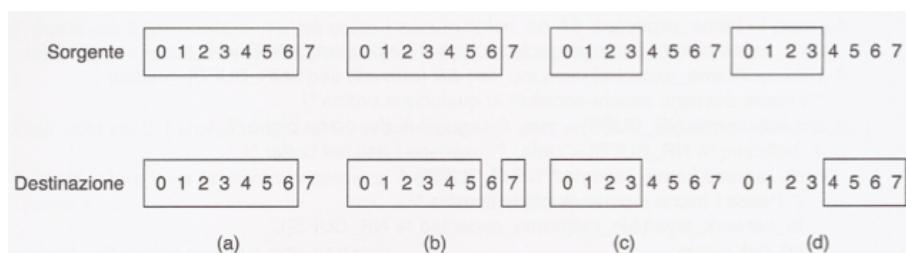
Siamo giunti all'ultimo passaggio. Adesso implementeremo il selective repeat, ricordando che questo significa avere anche dei buffer in ricezione (ai quali va associato un array di bit per dare significato al contenuto!) per poter completare la "groviera" e attendere i frame mancanti. Vengono rispediti solo i frame di cui scatta il timer.

Nel caso in cui riceviamo un frame successivo al frame_expected, sappiamo di certo che almeno il frame_expected è andato perso quindi possiamo inviare un NACK per comunicare al sender la mancata ricezione di quel frame. Stessa cosa vale nel caso di checksum error. Attenzione: il NACK deve essere inviato una sola volta per evitare ripetizioni inutili (ad ogni NACK corrisponde una ritrasmissione!).

I numeri di sequenza rispetto ai buffer

Come abbiamo visto nel go back N per evitare ambiguità si è reso necessario avere più numeri di sequenza rispetto alla dimensione effettiva della finestra.

Nel selective repeat il problema è ancora più estremo, infatti, come vediamo qui, non può bastare un solo numero in più.



Usiamo 7 numeri su 8: la sorgente invia 7 frame e il destinatario li riceve tutti e manda un ACK fino a 6 spostando così la sua finestra da 7 a 5 avendo tutti i buffer vuoti (intanto avrà consegnato al network) (b).

Purtroppo la sorgente non riceve alcun ACK (un fulmine!) e così il primo timer a scattare è ovviamente quello per il frame 0. Viene ritrasmesso il frame 0.

Il destinatario riceve un frame 0, che è nella sua finestra ma lo interpreta come nuovo --> PROBLEMA.

Il selective repeat richiede quindi più numeri di sequenza rispetto ai buffer: **NR_BUFS = (MAX_SEQ + 1) / 2**

Corretta gestione del piggybacking

Introduciamo anche un nuovo timer per poter gestire l'invio di ACK unitari.

Come abbiamo detto sopra, non possiamo costringere la rete ad avere un continuo scambio per poter inviare ACK... quindi dobbiamo introdurre un nuovo timer (ack_timer) che nel caso in cui non siano pronti dati proceda all'invio del singolo ACK. Si noti che la scelta del tempo di durata di questo ACK è molto delicata, poiché deve essere senz'altro più breve del timer di non ricezione (altrimenti scatterebbe in ogni caso il timer di non ricezione dell'ACK) ma se lo lasciamo troppo breve allora ricadiamo nella situazione del piggybacking forzato.

Quindi i timer totali saranno: NR_BUFS + 1.

Il codice

```

00 | #define MAX_SEQ 7
01 | #define NR_BUFS ((MAX_SEQ + 1) / 2)
02 | typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout}
03 |                                         event_type;
04 |
05 |
06 | static boolean between(seq_nr a, seq_nr b, seq_nr c) {
07 |     return (((a <= b) && (b < c))
08 |             || ((c < a) && (a <= b))
09 |             || ((b < c) && (c < a)));
10 |
11 |     }
12 |
13 |     static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected,
14 |                                         packet buffer[]) {
15 |         frame s;
16 |
17 |         s.kind == fk;
18 |         if(fk == data) s.info = buffer[frame_nr % NR_BUFS];
19 |         s.seq = frame_nr;
20 |         s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
21 |         if(fk == nak) no_nack = false;
22 |         to_physical_layer(&s);
23 |         if(fk == data) start_timer(frame_nr % NR_BUFS);
24 |         stop_ack_timer();
25 |     }
26 |
27 |     void protocol6(void) {
28 |         seq_nr next_frame_to_send;
29 |         seq_nr ack_expected;
30 |         seq_nr frame_expected;
31 |         seq_nr too_far;
32 |         int i;
33 |         frame r;
34 |         packet out_buf[NR_BUFS];
35 |         packet in_buf[NR_BUFS];
36 |         boolean arrived[NR_BUFS];
37 |         seq_nr nbuffered;
38 |         seq_nr i;
39 |         event_type event;
40 |
41 |         enable_network_layer();
42 |         ack_expected = 0;
43 |         next_frame_to_send = 0;
44 |         frame_expected = 0;
45 |         nbuffered = 0;
46 |         too_far = NR_BUFS;
47 |         for(i = 0; i < NR_BUFS; i++)
48 |             arrived[i] = false;

```

```

45 |     while(true) {
46 |         wait_for_event(&event);
47 |
48 |         switch(event) {
49 |             case network_layer_ready:
50 |                 from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]);
51 |                 nbuffered = nbuffered + 1;
52 |                 send_frame(data, next_frame_to_send, frame_expected, out_buf);
53 |                 inc(next_frame_to_send);
54 |             break;
55 |
56 |             case frame_arrival:
57 |                 from_physical_layer(&r);
58 |
59 |                 if(r.kind == data) {
60 |                     if((r.seq != frame_expected) && no_nack)
61 |                         send_frame(nack, 0, frame_expected, out_buf);
62 |                     else
63 |                         start_ack_timer();
64 |
65 |                     if(between(frame_expected, r.seq, too_far) &&
66 |                         (arrived[r.seq % NR_BUFS] == false)) {
67 |                         arrived[r.seq % NR_BUFS] = true;
68 |                         in_buf[r.seq % NR_BUFS] = r.info;
69 |
70 |                         while(arrived[frame_expected % NR_BUFS]) {
71 |                             to_network_layer(
72 |                                 &in_buf[frame_expected % NR_BUFS]
73 |                             );
74 |                             no_nack = true;
75 |                             arrived[frame_expected % NR_BUFS] = false;
76 |                             inc(frame_expected);
77 |                             inc(too_far);
78 |                             start_ack_timer();
79 |                         } //fine while
80 |                     } //fine if
81 |                 } // fine if
82 |
83 |                 if((r.kind == nak) &&
84 |                     between(ack_expected,
85 |                             (r.ack + 1) % MAX_SEQ + 1),
86 |                             next_frame_to_send)) {
87 |                     send_frame(data, (r.ack + 1) % (MAX_SEQ + 1),
88 |                             frame_expected, out_buf);
89 |
90 |                     while(between(ack_expected, r.ack, next_frame_to_send)) {
91 |                         nbuffered = nbuffered - 1;
92 |                         stop_timer(ack_expected % NR_BUFS);
93 |                         inc(ack_expected);
94 |                     }
95 |                 }
96 |             break;
97 |         }
98 |     }
99 |
100 |     if(nbuffers == 0)
101 |         stop_timer();
102 |
103 |     if(nbuffers > 0)
104 |         start_timer();
105 |
106 |     if(nbuffers > 0)
107 |         inc(nbuffers);
108 |
109 |     if(nbuffers <= 0)
110 |         dec(nbuffers);
111 |
112 |     if(nbuffers <= 0)
113 |         stop_timer();
114 |
115 |     if(nbuffers > 0)
116 |         start_timer();
117 |
118 |     if(nbuffers > 0)
119 |         inc(nbuffers);
120 |
121 |     if(nbuffers <= 0)
122 |         dec(nbuffers);
123 |
124 |     if(nbuffers <= 0)
125 |         stop_timer();
126 |
127 |     if(nbuffers > 0)
128 |         start_timer();
129 |
130 |     if(nbuffers > 0)
131 |         inc(nbuffers);
132 |
133 |     if(nbuffers <= 0)
134 |         dec(nbuffers);
135 |
136 |     if(nbuffers <= 0)
137 |         stop_timer();
138 |
139 |     if(nbuffers > 0)
140 |         start_timer();
141 |
142 |     if(nbuffers > 0)
143 |         inc(nbuffers);
144 |
145 |     if(nbuffers <= 0)
146 |         dec(nbuffers);
147 |
148 |     if(nbuffers <= 0)
149 |         stop_timer();
150 |
151 |     if(nbuffers > 0)
152 |         start_timer();
153 |
154 |     if(nbuffers > 0)
155 |         inc(nbuffers);
156 |
157 |     if(nbuffers <= 0)
158 |         dec(nbuffers);
159 |
160 |     if(nbuffers <= 0)
161 |         stop_timer();
162 |
163 |     if(nbuffers > 0)
164 |         start_timer();
165 |
166 |     if(nbuffers > 0)
167 |         inc(nbuffers);
168 |
169 |     if(nbuffers <= 0)
170 |         dec(nbuffers);
171 |
172 |     if(nbuffers <= 0)
173 |         stop_timer();
174 |
175 |     if(nbuffers > 0)
176 |         start_timer();
177 |
178 |     if(nbuffers > 0)
179 |         inc(nbuffers);
180 |
181 |     if(nbuffers <= 0)
182 |         dec(nbuffers);
183 |
184 |     if(nbuffers <= 0)
185 |         stop_timer();
186 |
187 |     if(nbuffers > 0)
188 |         start_timer();
189 |
190 |     if(nbuffers > 0)
191 |         inc(nbuffers);
192 |
193 |     if(nbuffers <= 0)
194 |         dec(nbuffers);
195 |
196 |     if(nbuffers <= 0)
197 |         stop_timer();
198 |
199 |     if(nbuffers > 0)
200 |         start_timer();
201 |
202 |     if(nbuffers > 0)
203 |         inc(nbuffers);
204 |
205 |     if(nbuffers <= 0)
206 |         dec(nbuffers);
207 |
208 |     if(nbuffers <= 0)
209 |         stop_timer();
210 |
211 |     if(nbuffers > 0)
212 |         start_timer();
213 |
214 |     if(nbuffers > 0)
215 |         inc(nbuffers);
216 |
217 |     if(nbuffers <= 0)
218 |         dec(nbuffers);
219 |
220 |     if(nbuffers <= 0)
221 |         stop_timer();
222 |
223 |     if(nbuffers > 0)
224 |         start_timer();
225 |
226 |     if(nbuffers > 0)
227 |         inc(nbuffers);
228 |
229 |     if(nbuffers <= 0)
230 |         dec(nbuffers);
231 |
232 |     if(nbuffers <= 0)
233 |         stop_timer();
234 |
235 |     if(nbuffers > 0)
236 |         start_timer();
237 |
238 |     if(nbuffers > 0)
239 |         inc(nbuffers);
240 |
241 |     if(nbuffers <= 0)
242 |         dec(nbuffers);
243 |
244 |     if(nbuffers <= 0)
245 |         stop_timer();
246 |
247 |     if(nbuffers > 0)
248 |         start_timer();
249 |
250 |     if(nbuffers > 0)
251 |         inc(nbuffers);
252 |
253 |     if(nbuffers <= 0)
254 |         dec(nbuffers);
255 |
256 |     if(nbuffers <= 0)
257 |         stop_timer();
258 |
259 |     if(nbuffers > 0)
260 |         start_timer();
261 |
262 |     if(nbuffers > 0)
263 |         inc(nbuffers);
264 |
265 |     if(nbuffers <= 0)
266 |         dec(nbuffers);
267 |
268 |     if(nbuffers <= 0)
269 |         stop_timer();
270 |
271 |     if(nbuffers > 0)
272 |         start_timer();
273 |
274 |     if(nbuffers > 0)
275 |         inc(nbuffers);
276 |
277 |     if(nbuffers <= 0)
278 |         dec(nbuffers);
279 |
280 |     if(nbuffers <= 0)
281 |         stop_timer();
282 |
283 |     if(nbuffers > 0)
284 |         start_timer();
285 |
286 |     if(nbuffers > 0)
287 |         inc(nbuffers);
288 |
289 |     if(nbuffers <= 0)
290 |         dec(nbuffers);
291 |
292 |     if(nbuffers <= 0)
293 |         stop_timer();
294 |
295 |     if(nbuffers > 0)
296 |         start_timer();
297 |
298 |     if(nbuffers > 0)
299 |         inc(nbuffers);
300 |
301 |     if(nbuffers <= 0)
302 |         dec(nbuffers);
303 |
304 |     if(nbuffers <= 0)
305 |         stop_timer();
306 |
307 |     if(nbuffers > 0)
308 |         start_timer();
309 |
310 |     if(nbuffers > 0)
311 |         inc(nbuffers);
312 |
313 |     if(nbuffers <= 0)
314 |         dec(nbuffers);
315 |
316 |     if(nbuffers <= 0)
317 |         stop_timer();
318 |
319 |     if(nbuffers > 0)
320 |         start_timer();
321 |
322 |     if(nbuffers > 0)
323 |         inc(nbuffers);
324 |
325 |     if(nbuffers <= 0)
326 |         dec(nbuffers);
327 |
328 |     if(nbuffers <= 0)
329 |         stop_timer();
330 |
331 |     if(nbuffers > 0)
332 |         start_timer();
333 |
334 |     if(nbuffers > 0)
335 |         inc(nbuffers);
336 |
337 |     if(nbuffers <= 0)
338 |         dec(nbuffers);
339 |
340 |     if(nbuffers <= 0)
341 |         stop_timer();
342 |
343 |     if(nbuffers > 0)
344 |         start_timer();
345 |
346 |     if(nbuffers > 0)
347 |         inc(nbuffers);
348 |
349 |     if(nbuffers <= 0)
350 |         dec(nbuffers);
351 |
352 |     if(nbuffers <= 0)
353 |         stop_timer();
354 |
355 |     if(nbuffers > 0)
356 |         start_timer();
357 |
358 |     if(nbuffers > 0)
359 |         inc(nbuffers);
360 |
361 |     if(nbuffers <= 0)
362 |         dec(nbuffers);
363 |
364 |     if(nbuffers <= 0)
365 |         stop_timer();
366 |
367 |     if(nbuffers > 0)
368 |         start_timer();
369 |
370 |     if(nbuffers > 0)
371 |         inc(nbuffers);
372 |
373 |     if(nbuffers <= 0)
374 |         dec(nbuffers);
375 |
376 |     if(nbuffers <= 0)
377 |         stop_timer();
378 |
379 |     if(nbuffers > 0)
380 |         start_timer();
381 |
382 |     if(nbuffers > 0)
383 |         inc(nbuffers);
384 |
385 |     if(nbuffers <= 0)
386 |         dec(nbuffers);
387 |
388 |     if(nbuffers <= 0)
389 |         stop_timer();
390 |
391 |     if(nbuffers > 0)
392 |         start_timer();
393 |
394 |     if(nbuffers > 0)
395 |         inc(nbuffers);
396 |
397 |     if(nbuffers <= 0)
398 |         dec(nbuffers);
399 |
400 |     if(nbuffers <= 0)
401 |         stop_timer();
402 |
403 |     if(nbuffers > 0)
404 |         start_timer();
405 |
406 |     if(nbuffers > 0)
407 |         inc(nbuffers);
408 |
409 |     if(nbuffers <= 0)
410 |         dec(nbuffers);
411 |
412 |     if(nbuffers <= 0)
413 |         stop_timer();
414 |
415 |     if(nbuffers > 0)
416 |         start_timer();
417 |
418 |     if(nbuffers > 0)
419 |         inc(nbuffers);
420 |
421 |     if(nbuffers <= 0)
422 |         dec(nbuffers);
423 |
424 |     if(nbuffers <= 0)
425 |         stop_timer();
426 |
427 |     if(nbuffers > 0)
428 |         start_timer();
429 |
430 |     if(nbuffers > 0)
431 |         inc(nbuffers);
432 |
433 |     if(nbuffers <= 0)
434 |         dec(nbuffers);
435 |
436 |     if(nbuffers <= 0)
437 |         stop_timer();
438 |
439 |     if(nbuffers > 0)
440 |         start_timer();
441 |
442 |     if(nbuffers > 0)
443 |         inc(nbuffers);
444 |
445 |     if(nbuffers <= 0)
446 |         dec(nbuffers);
447 |
448 |     if(nbuffers <= 0)
449 |         stop_timer();
450 |
451 |     if(nbuffers > 0)
452 |         start_timer();
453 |
454 |     if(nbuffers > 0)
455 |         inc(nbuffers);
456 |
457 |     if(nbuffers <= 0)
458 |         dec(nbuffers);
459 |
460 |     if(nbuffers <= 0)
461 |         stop_timer();
462 |
463 |     if(nbuffers > 0)
464 |         start_timer();
465 |
466 |     if(nbuffers > 0)
467 |         inc(nbuffers);
468 |
469 |     if(nbuffers <= 0)
470 |         dec(nbuffers);
471 |
472 |     if(nbuffers <= 0)
473 |         stop_timer();
474 |
475 |     if(nbuffers > 0)
476 |         start_timer();
477 |
478 |     if(nbuffers > 0)
479 |         inc(nbuffers);
480 |
481 |     if(nbuffers <= 0)
482 |         dec(nbuffers);
483 |
484 |     if(nbuffers <= 0)
485 |         stop_timer();
486 |
487 |     if(nbuffers > 0)
488 |         start_timer();
489 |
490 |     if(nbuffers > 0)
491 |         inc(nbuffers);
492 |
493 |     if(nbuffers <= 0)
494 |         dec(nbuffers);
495 |
496 |     if(nbuffers <= 0)
497 |         stop_timer();
498 |
499 |     if(nbuffers > 0)
500 |         start_timer();
501 |
502 |     if(nbuffers > 0)
503 |         inc(nbuffers);
504 |
505 |     if(nbuffers <= 0)
506 |         dec(nbuffers);
507 |
508 |     if(nbuffers <= 0)
509 |         stop_timer();
510 |
511 |     if(nbuffers > 0)
512 |         start_timer();
513 |
514 |     if(nbuffers > 0)
515 |         inc(nbuffers);
516 |
517 |     if(nbuffers <= 0)
518 |         dec(nbuffers);
519 |
520 |     if(nbuffers <= 0)
521 |         stop_timer();
522 |
523 |     if(nbuffers > 0)
524 |         start_timer();
525 |
526 |     if(nbuffers > 0)
527 |         inc(nbuffers);
528 |
529 |     if(nbuffers <= 0)
530 |         dec(nbuffers);
531 |
532 |     if(nbuffers <= 0)
533 |         stop_timer();
534 |
535 |     if(nbuffers > 0)
536 |         start_timer();
537 |
538 |     if(nbuffers > 0)
539 |         inc(nbuffers);
540 |
541 |     if(nbuffers <= 0)
542 |         dec(nbuffers);
543 |
544 |     if(nbuffers <= 0)
545 |         stop_timer();
546 |
547 |     if(nbuffers > 0)
548 |         start_timer();
549 |
550 |     if(nbuffers > 0)
551 |         inc(nbuffers);
552 |
553 |     if(nbuffers <= 0)
554 |         dec(nbuffers);
555 |
556 |     if(nbuffers <= 0)
557 |         stop_timer();
558 |
559 |     if(nbuffers > 0)
560 |         start_timer();
561 |
562 |     if(nbuffers > 0)
563 |         inc(nbuffers);
564 |
565 |     if(nbuffers <= 0)
566 |         dec(nbuffers);
567 |
568 |     if(nbuffers <= 0)
569 |         stop_timer();
570 |
571 |     if(nbuffers > 0)
572 |         start_timer();
573 |
574 |     if(nbuffers > 0)
575 |         inc(nbuffers);
576 |
577 |     if(nbuffers <= 0)
578 |         dec(nbuffers);
579 |
580 |     if(nbuffers <= 0)
581 |         stop_timer();
582 |
583 |     if(nbuffers > 0)
584 |         start_timer();
585 |
586 |     if(nbuffers > 0)
587 |         inc(nbuffers);
588 |
589 |     if(nbuffers <= 0)
590 |         dec(nbuffers);
591 |
592 |     if(nbuffers <= 0)
593 |         stop_timer();
594 |
595 |     if(nbuffers > 0)
596 |         start_timer();
597 |
598 |     if(nbuffers > 0)
599 |         inc(nbuffers);
600 |
601 |     if(nbuffers <= 0)
602 |         dec(nbuffers);
603 |
604 |     if(nbuffers <= 0)
605 |         stop_timer();
606 |
607 |     if(nbuffers > 0)
608 |         start_timer();
609 |
610 |     if(nbuffers > 0)
611 |         inc(nbuffers);
612 |
613 |     if(nbuffers <= 0)
614 |         dec(nbuffers);
615 |
616 |     if(nbuffers <= 0)
617 |         stop_timer();
618 |
619 |     if(nbuffers > 0)
620 |         start_timer();
621 |
622 |     if(nbuffers > 0)
623 |         inc(nbuffers);
624 |
625 |     if(nbuffers <= 0)
626 |         dec(nbuffers);
627 |
628 |     if(nbuffers <= 0)
629 |         stop_timer();
630 |
631 |     if(nbuffers > 0)
632 |         start_timer();
633 |
634 |     if(nbuffers > 0)
635 |         inc(nbuffers);
636 |
637 |     if(nbuffers <= 0)
638 |         dec(nbuffers);
639 |
640 |     if(nbuffers <= 0)
641 |         stop_timer();
642 |
643 |     if(nbuffers > 0)
644 |         start_timer();
645 |
646 |     if(nbuffers > 0)
647 |         inc(nbuffers);
648 |
649 |     if(nbuffers <= 0)
650 |         dec(nbuffers);
651 |
652 |     if(nbuffers <= 0)
653 |         stop_timer();
654 |
655 |     if(nbuffers > 0)
656 |         start_timer();
657 |
658 |     if(nbuffers > 0)
659 |         inc(nbuffers);
660 |
661 |     if(nbuffers <= 0)
662 |         dec(nbuffers);
663 |
664 |     if(nbuffers <= 0)
665 |         stop_timer();
666 |
667 |     if(nbuffers > 0)
668 |         start_timer();
669 |
670 |     if(nbuffers > 0)
671 |         inc(nbuffers);
672 |
673 |     if(nbuffers <= 0)
674 |         dec(nbuffers);
675 |
676 |     if(nbuffers <= 0)
677 |         stop_timer();
678 |
679 |     if(nbuffers > 0)
680 |         start_timer();
681 |
682 |     if(nbuffers > 0)
683 |         inc(nbuffers);
684 |
685 |     if(nbuffers <= 0)
686 |         dec(nbuffers);
687 |
688 |     if(nbuffers <= 0)
689 |         stop_timer();
690 |
691 |     if(nbuffers > 0)
692 |         start_timer();
693 |
694 |     if(nbuffers > 0)
695 |         inc(nbuffers);
696 |
697 |     if(nbuffers <= 0)
698 |         dec(nbuffers);
699 |
700 |     if(nbuffers <= 0)
701 |         stop_timer();
702 |
703 |     if(nbuffers > 0)
704 |         start_timer();
705 |
706 |     if(nbuffers > 0)
707 |         inc(nbuffers);
708 |
709 |     if(nbuffers <= 0)
710 |         dec(nbuffers);
711 |
712 |     if(nbuffers <= 0)
713 |         stop_timer();
714 |
715 |     if(nbuffers > 0)
716 |         start_timer();
717 |
718 |     if(nbuffers > 0)
719 |         inc(nbuffers);
720 |
721 |     if(nbuffers <= 0)
722 |         dec(nbuffers);
723 |
724 |     if(nbuffers <= 0)
725 |         stop_timer();
726 |
727 |     if(nbuffers > 0)
728 |         start_timer();
729 |
730 |     if(nbuffers > 0)
731 |         inc(nbuffers);
732 |
733 |     if(nbuffers <= 0)
734 |         dec(nbuffers);
735 |
736 |     if(nbuffers <= 0)
737 |         stop_timer();
738 |
739 |     if(nbuffers > 0)
740 |         start_timer();
741 |
742 |     if(nbuffers > 0)
743 |         inc(nbuffers);
744 |
745 |     if(nbuffers <= 0)
746 |         dec(nbuffers);
747 |
748 |     if(nbuffers <= 0)
749 |         stop_timer();
750 |
751 |     if(nbuffers > 0)
752 |         start_timer();
753 |
754 |     if(nbuffers > 0)
755 |         inc(nbuffers);
756 |
757 |     if(nbuffers <= 0)
758 |         dec(nbuffers);
759 |
760 |     if(nbuffers <= 0)
761 |         stop_timer();
762 |
763 |     if(nbuffers > 0)
764 |         start_timer();
765 |
766 |     if(nbuffers > 0)
767 |         inc(nbuffers);
768 |
769 |     if(nbuffers <= 0)
770 |         dec(nbuffers);
771 |
772 |     if(nbuffers <= 0)
773 |         stop_timer();
774 |
775 |     if(nbuffers > 0)
776 |         start_timer();
777 |
778 |     if(nbuffers > 0)
779 |         inc(nbuffers);
780 |
781 |     if(nbuffers <= 0)
782 |         dec(nbuffers);
783 |
784 |     if(nbuffers <= 0)
785 |         stop_timer();
786 |
787 |     if(nbuffers > 0)
788 |         start_timer();
789 |
790 |     if(nbuffers > 0)
791 |         inc(nbuffers);
792 |
793 |     if(nbuffers <= 0)
794 |         dec(nbuffers);
795 |
796 |     if(nbuffers <= 0)
797 |         stop_timer();
798 |
799 |     if(nbuffers > 0)
800 |         start_timer();
801 |
802 |     if(nbuffers > 0)
803 |         inc(nbuffers);
804 |
805 |     if(nbuffers <= 0)
806 |         dec(nbuffers);
807 |
808 |     if(nbuffers <= 0)
809 |         stop_timer();
810 |
811 |     if(nbuffers > 0)
812 |         start_timer();
813 |
814 |     if(nbuffers > 0)
815 |         inc(nbuffers);
816 |
817 |     if(nbuffers <= 0)
818 |         dec(nbuffers);
819 |
820 |     if(nbuffers <= 0)
821 |         stop_timer();
822 |
823 |     if(nbuffers > 0)
824 |         start_timer();
825 |
826 |     if(nbuffers > 0)
827 |         inc(nbuffers);
828 |
829 |     if(nbuffers <= 0)
830 |         dec(nbuffers);
831 |
832 |     if(nbuffers <= 0)
833 |         stop_timer();
834 |
835 |     if(nbuffers > 0)
836 |         start_timer();
837 |
838 |     if(nbuffers > 0)
839 |         inc(nbuffers);
840 |
841 |     if(nbuffers <= 0)
842 |         dec(nbuffers);
843 |
844 |     if(nbuffers <= 0)
845 |         stop_timer();
846 |
847 |     if(nbuffers > 0)
848 |         start_timer();
849 |
850 |     if(nbuffers > 0)
851 |         inc(nbuffers);
852 |
853 |     if(nbuffers <= 0)
854 |         dec(nbuffers);
855 |
856 |     if(nbuffers <= 0)
857 |         stop_timer();
858 |
859 |     if(nbuffers > 0)
860 |         start_timer();
861 |
862 |     if(nbuffers > 0)
863 |         inc(nbuffers);
864 |
865 |     if(nbuffers <= 0)
866 |         dec(nbuffers);
867 |
868 |     if(nbuffers <= 0)
869 |         stop_timer();
870 |
871 |     if(nbuffers > 0)
872 |         start_timer();
873 |
874 |     if(nbuffers > 0)
875 |         inc(nbuffers);
876 |
877 |     if(nbuffers <= 0)
878 |         dec(nbuffers);
879 |
880 |     if(nbuffers <= 0)
881 |         stop_timer();
882 |
883 |     if(nbuffers > 0)
884 |         start_timer();
885 |
886 |     if(nbuffers > 0)
887 |         inc(nbuffers);
888 |
889 |     if(nbuffers <= 0)
890 |         dec(nbuffers);
891 |
892 |     if(nbuffers <= 0)
893 |         stop_timer();
894 |
895 |     if(nbuffers > 0)
896 |         start_timer();
897 |
898 |     if(nbuffers > 0)
899 |         inc(nbuffers);
900 |
901 |     if(nbuffers <= 0)
902 |         dec(nbuffers);
903 |
904 |     if(nbuffers <= 0)
905 |         stop_timer();
906 |
907 |     if(nbuffers > 0)
908 |         start_timer();
909 |
910 |     if(nbuffers > 0)
911 |         inc(nbuffers);
912 |
913 |     if(nbuffers <= 0)
914 |         dec(nbuffers);
915 |
916 |     if(nbuffers <= 0)
917 |         stop_timer();
918 |
919 |     if(nbuffers > 0)
920 |         start_timer();
921 |
922 |     if(nbuffers > 0)
923 |         inc(nbuffers);
924 |
925 |     if(nbuffers <= 0)
926 |         dec(nbuffers);
927 |
928 |     if(nbuffers <= 0)
929 |         stop_timer();
930 |
931 |     if(nbuffers > 0)
932 |         start_timer();
933 |
934 |     if(nbuffers > 0)
935 |         inc(nbuffers);
936 |
937 |     if(nbuffers <= 0)
938 |         dec(nbuffers);
939 |
940 |     if(nbuffers <= 0)
941 |         stop_timer();
942 |
943 |     if(nbuffers > 0)
944 |         start_timer();
945 |
946 |     if(nbuffers > 0)
947 |         inc(nbuffers);
948 |
949 |     if(nbuffers <= 0)
950 |         dec(nbuffers);
951 |
952 |     if(nbuffers <= 0)
953 |         stop_timer();
954 |
955 |     if(nbuffers > 0)
956 |         start_timer();
957 |
958 |     if(nbuffers > 0)
959 |         inc(nbuffers);
960 |
961 |     if(nbuffers <= 0)
962 |         dec(nbuffers);
963 |
964 |     if(nbuffers <= 0)
965 |         stop_timer();
966 |
967 |     if(nbuffers > 0)
968 |         start_timer();
969 |
970 |     if(nbuffers > 0)
971 |         inc(nbuffers);
972 |
973 |     if(nbuffers <= 0)
974 |         dec(nbuffers);
975 |
976 |     if(nbuffers <= 0)
977 |         stop_timer();
978 |
979 |     if(nbuffers > 0)
980 |         start_timer();
981 |
982 |     if(nbuffers > 0)
983 |         inc(nbuffers);
984 |
985 |     if(nbuffers <= 0)
986 |         dec(nbuffers);
987 |
988 |     if(nbuffers <= 0)
989 |         stop_timer();
990 |
991 |     if(nbuffers > 0)
992 |         start_timer();
993 |
994 |     if(nbuffers > 0)
995 |         inc(nbuffers);
996 |
997 |     if(nbuffers <= 0)
998 |         dec(nbuffers);
999 |
1000 |     if(nbuffers <= 0)
1001 |         stop_timer();
1002 |
1003 |     if(nbuffers > 0)
1004 |         start_timer();
1005 |
1006 |     if(nbuffers > 0)
1007 |         inc(nbuffers);
1008 |
1009 |     if(nbuffers <= 0)
1010 |         dec(nbuffers);
1011 |
1012 |     if(nbuffers <= 0)
1013 |         stop_timer();
1014 |
1015 |     if(nbuffers > 0)
1016 |         start_timer();
1017 |
1018 |     if(nbuffers > 0)
1019 |         inc(nbuffers);
1020 |
1021 |     if(nbuffers <= 0)
1022 |         dec(nbuffers);
1023 |
1024 |     if(nbuffers <= 0)
1025 |         stop_timer();
1026 |
1027 |     if(nbuffers > 0)
1028 |         start_timer();
1029 |
1030 |     if(nbuffers > 0)
1031 |         inc(nbuffers);
1032 |
1033 |     if(nbuffers <= 0)
1034 |         dec(nbuffers);
1035 |
1036 |     if(nbuffers <= 0)
1037 |         stop_timer();
1038 |
1039 |     if(nbuffers > 0)
1040 |         start_timer();
1041 |
1042 |     if(nbuffers > 0)
1043 |         inc(nbuffers);
1044 |
1045 |     if(nbuffers <= 0)
1046 |         dec(nbuffers);
1047 |
1048 |     if(nbuffers <= 0)
1049 |         stop_timer();
1050 |
1051 |     if(nbuffers > 0)
1052 |         start_timer();
1053 |
1054 |     if(nbuffers > 0)
1055 |         inc(nbuffers);
1056 |
1057 |     if(nbuffers <= 0)
1058 |         dec(nbuffers);
1059 |
1060 |     if(nbuffers <= 0)
1061 |         stop_timer();
1062 |
1063 |     if(nbuffers > 0)
1064 |         start_timer();
1065 |
1066 |     if(nbuffers > 0)
1067 |         inc(nbuffers);
1068 |
1069 |     if(nbuffers <= 0)
1070 |         dec(nbuffers);
1071 |
1072 |     if(nbuffers <= 0)
1073 |         stop_timer();
1074 |
1075 |     if(nbuffers > 0)
1076 |         start_timer();
1077 |
1078 |     if(nbuffers > 0)
1079 |         inc(nbuffers);
1080 |
1081 |     if(nbuffers <= 0)
1082 |         dec(nbuffers);
1083 |
1084 |     if(nbuffers <= 0)
1085 |         stop_timer();
1086 |
1087 |     if(nbuffers > 0)
1088 |         start_timer();
1089 |
1090 |     if(nbuffers > 0)
1091 |         inc(nbuffers);
1092 |
1093 |     if(nbuffers <= 0)
1094 |         dec(nbuffers);
1095 |
1096 |     if(nbuffers <= 0)
1097 |         stop_timer();
1098 |
1099 |     if(nbuffers > 0)
1100 |         start_timer();
1101 |
1102 |     if(nbuffers > 0)
1103 |         inc(nbuffers);
1104 |
1105 |     if(nbuffers <= 0)
1106 |         dec(nbuffers);
1107 |
1108 |     if(nbuffers <= 0)
1109 |         stop_timer();
1110 |
1111 |     if(nbuffers > 0)
1112 |         start_timer();
1113 |
1114 |     if(nbuffers > 0)
1115 |         inc(nbuffers);
1116 |
1117 |     if(nbuffers <= 0)
1118 |         dec(nbuffers);
1119 |
1120 |     if(nbuffers <= 0)
1121 |         stop_timer();
1122 |
1123 |     if(nbuffers > 0)
1124 |         start_timer();
1125 |
1126 |     if(nbuffers > 0)
1127 |         inc(nbuffers);
1128 |
1129 |     if(nbuffers <= 0)
1130 |         dec(nbuffers);
1131 |
1132 |     if(nbuffers <= 0)
1133 |         stop_timer();
1134 |
1135 |     if(nbuffers > 0)
1136 |         start_timer();
1137 |
1138 |     if(nbuffers > 0)
1139 |         inc(nbuffers);
1140 |
1141 |     if(nbuffers <= 0)
1142 |         dec(nbuffers);
1143 |
1144 |     if(nbuffers <= 0)
1145 |         stop_timer();
1146 |
1147 |     if(nbuffers > 0)
1148 |         start_timer();
1149 |
1150 |     if(nbuffers > 0)
1151 |         inc(nbuffers);
1152 |
1153 |     if(nbuffers <= 0)
1154 |         dec(nbuffers);
1155 |
1156 |     if(nbuffers <= 0)
1157 |         stop_timer();
1158 |
1159 |     if(nbuffers > 0)
1160 |         start_timer();
1161 |
1162 |     if(nbuffers > 0)
1163 |         inc(nbuffers);
1164 |
1165 |     if(nbuffers <= 0)
1166 |         dec(nbuffers);
1167 |
1168 |     if(nbuffers <= 0)
1169 |         stop_timer();
1170 |
1171 |     if(nbuffers > 0)
1172 |         start_timer();
1173 |
1174 |     if(nbuffers > 0)
1175 |         inc(nbuffers);
1176 |
1177 |     if(nbuffers <= 0)
1178 |         dec(nbuffers);
1179 |
1180 |     if(nbuffers <= 0)
1181 |         stop_timer();
1182 |
1183 |     if(nbuffers > 0)
1184 |         start_timer();
1185 |
1186 |     if(nbuffers > 0)
1187 |         inc(nbuffers);
1188 |
1189 |     if(nbuffers <= 0)
1190 |         dec(nbuffers);
1191 |
1192 |     if(nbuffers <= 0)
1193 |         stop_timer();
1194 |
1195 |     if(nbuffers > 0)
1196 |         start_timer();
1197 |
1198 |     if(nbuffers > 0)
1199 |         inc(nbuffers);
1200 |
1201 |     if(nbuffers <= 0)
1202 |         dec(nbuffers);
1203 |
1204 |     if(nbuffers <= 0)
1205 |         stop_timer();
1206 |
1207 |     if(nbuffers > 0)
1208 |         start_timer();
1209 |
1210 |     if(nbuffers > 0)
1211 |         inc(nbuffers);
1212 |
1213 |     if(nbuffers <= 0)
1214 |         dec(nbuffers);
1215 |
1216 |     if(nbuffers <= 0)
1217 |         stop_timer();
1218 |
1219 |     if(nbuffers > 0)
1220 |         start_timer();
1221 |
1222 |     if(nbuffers > 0)
1223 |         inc(nbuffers);
1224 |
1225 |     if(nbuffers <= 0)
1226 |         dec(nbuffers);
1227 |
1228 |     if(nbuffers <= 0)
1229 |         stop_timer();
1230 |
1231 |     if(nbuffers > 0)
1232 |         start_timer();
1233 |
1234 |     if(nbuffers > 0)
1235 |         inc(nbuffers);
1236 |
1237 |     if(nbuffers <= 0)
1238 |         dec(nbuffers);
1239 |
1240 |     if(nbuffers <= 0)
1241 |         stop_timer();
1242 |
1243 |     if(nbuffers > 0)
1244 |         start_timer();
1245 |
1246 |     if(nbuffers > 0)
1247 |         inc(nbuffers);
1248 |
1249 |     if(nbuffers <= 0)
1250 |         dec(nbuffers);
1251 |
1252 |     if(nbuffers <= 0)
1253 |         stop_timer();
1254 |
1255 |     if(nbuffers > 0)
1256 |         start_timer();
1257 |
1258 |     if(nbuffers > 0)
1259 |         inc(nbuffers);
1260 |
1261 |     if(nbuffers <= 0)
1262 |         dec(nbuffers);
1263 |
1264 |     if(nbuffers <= 0)
1265 |         stop_timer();
1266 |
1267 |     if(nbuffers > 0)
1268 |         start_timer();
1269 |
1270 |     if(nbuffers > 0)
1271 |         inc(nbuffers);
1272 |
1273 |     if(nbuffers <= 0)
1274 |         dec(nbuffers);
1275 |
1276 |     if(nbuffers <= 0)
1277 |         stop_timer();
1278 |
1279 |     if(nbuffers > 0)
1280 |         start_timer();
1281 |
1282 |     if(nbuffers > 0)
1283 |         inc(nbuffers);
1284 |
1285 |     if(nbuffers <= 0)
1286 |         dec(nbuffers);
1287 |
1288 |     if(nbuffers <= 0)
1289 |         stop_timer();
1290 |
1291 |     if(nbuffers > 0)
1292 |         start_timer();
1293 |
1294 |     if(nbuffers > 0)
1295 |         inc(nbuffers);
1296 |
1297 |     if(nbuffers <= 0)
1298 |         dec(nbuffers);
1299 |
1300 |     if(nbuffers <= 0)
1301 |         stop_timer();
1302 |
1303 |     if(nbuffers > 0)
1304 |         start_timer();
1305 |
1306 |     if(nbuffers > 0)
1307 |         inc(nbuffers);
1308 |
1309 |     if(nbuffers <= 0)
1310 |         dec(nbuffers);
1311 |
1312 |     if(nbuffers <= 0)
1313 |         stop_timer();
1314 |
1315 |     if(nbuffers > 0)
1316 |         start_timer();
1317 |
1318 |     if(nbuffers > 0)
1319 |         inc(nbuffers);
1320 |
1321 |     if(nbuffers <= 0)
1322 |         dec(nbuffers);
1323 |
1324 |     if(nbuffers <= 0)
1325 |         stop_timer();
1326 |
1327 |     if(nbuffers > 0)
1328 |         start_timer();
1329 |
1330 |     if(nbuffers > 0)
1331 |         inc(nbuffers);
1332 |
1333 |     if(nbuffers <= 0)
1334 |         dec(nbuffers);
1335 |
1336 |     if(nbuffers <= 0)
1337 |         stop_timer();
1338 |
1339 |     if(nbuffers > 0)
1340 |         start_timer();
1341 |
1342 |     if(nbuffers > 0)
1343 |         inc(nbuffers);
1344 |
1345 |     if(nbuffers <= 0)
1346 |         dec(nbuffers);
1347 |
1348 |     if(nbuffers <= 0)
1349 |         stop_timer();
1350 |
1351 |     if(nbuffers > 0)
1352 |         start_timer();
1353 |
1354 |     if(nbuffers > 0)
1355 |         inc(nbuffers);
1356 |
1357 |     if(nbuffers <= 0)
1358 |         dec(nbuffers);
1359 |
1360 |     if(nbuffers <= 0)
1361 |         stop_timer();
1362 |
1363 |     if(nbuffers > 0)
1364 |         start_timer();
1365 |
1366 |     if(nbuffers > 0)
1367
```

```

89 |         case cksum_err:
90 |             if(no_nak) send_frame(nack, 0, frame_expected, out_buf);
91 |             break;
92 |
93 |         case timeout:
94 |             send_frame(data, oldest_frame, frame_expected, out_buf);
95 |             break;
96 |
97 |         case ack_timeout:
98 |             send_frame(ack, 0, frame_expected, out_buf);
99 |             break;
100|
101|     } //fine switch
102|
103|     if(nbuffed > MAX_SEQ)
104|         enable_network_layer();
105|     else
106|         disable_network_layer();
107|
108| } //fine while
109| } //fine protocollo

```

Come è chiaro, questa volta ci troviamo ad affrontare un codice ancora più complesso.

A inizio codice oltre alla famigliare define del MAX_SEQ [0] vediamo ora un **NR_BUFS** [1] poiché questa volta i numeri di sequenza ed il numero di buffer sono ben differenti (anche se legati dalla correlazione $NR_BUFS = (MAX_SEQ + 1) / 2$ come spiegato nel paragrafo precedente).

Abbiamo una nuova tipologia di evento, l'**ack_timeout** [2]: non faremo piggybacking in ogni caso e quindi talvolta, più precisamente allo scadere del timer per gli ACK, sarà necessario interpretare l'evento per mandare un ACK singolo senza i dati annessi.

Vediamo poi una variabile booleana **no_nack** [3] che ci dice se abbiamo già spedito un NACK oppure no. Ne capiremo meglio l'uso in seguito.

La variabile **oldest_frame** [4] è una variabile globale usata dalle funzioni di timing che ci dice di quale è stato il frame di cui è scaduto il timer.

Ritroviamo poi la stessa funzione between del protocollo precedente ma implementata in modo ancora più criptico.

Cambia invece la **send_frame** [10]: abbiamo ora diverse tipologie di frame: ACK, NACK, ACK con dati. A seconda quindi del parametro fk che rappresenta il tipo (uno fra ACK, NACK, data) di frame che vogliamo inviare eseguiremo parti differenti del metodo.

Se il frame è di tipo dati, copiamo i dati dal buffer (ottenuto circolarmente sulla variabile NR_BUFS) e li mettiamo nella variabile che poi spediremo [14]. Andiamo poi a riempire il campo numero di sequenza [15] e successivamente il campo ACK [16]. Nel caso in cui il tipo sia NACK allora verrà comunque usato il campo ACK per passare il NACK. Quindi questo campo è bivalente, interpretato a seconda del frame.

Dopodiché nel caso in cui stiamo costruendo un NACK, ci teniamo segnato il fatto che un NACK l'abbiamo mandato per questo frame e quindi settiamo la variabile no_nack a false [17]. Non resta che inviare al livello fisico il frame [18] e poi nel caso di dati, far partire un timer [18] (che come sempre verrà bloccato all'arrivo dell'ACK per quel determinato frame).

Fermiamo poi l'ack_timer [19] in ogni caso, poiché abbiamo appena mandato un ACK, oppure data (che contengono comunque un ACK) oppure un NACK (e ci va bene lo stesso: comunque abbiamo dato un feedback).

Entriamo nel vivo del codice: fra le inizializzazioni troviamo le **definizioni delle due finestre**: in [24] abbiamo il limite inferiore della finestra sorgente, in [25] il suo limite superiore + 1, in [26] il limite inferiore della finestra di ricezione e in [37] il suo limite superiore più uno (che infatti inizializziamo poi al numero di buffer di ricezione di cui disponiamo [42]).

I limiti superiori sono definiti con una unità in più per comodità sui controlli (più precisamente della funzione `between`).

In [30] e [31] abbiamo i buffer di entrata e di uscita, con l'array di supporto **arrived** [32] che valorizzerà i buffer in ricezione contenenti effettivamente dati utili da quelli invece vuoti (bit a 1, buffer pieno, bit a 0, buffer vuoto). Tale array è inizializzato totalmente a 0 [43] (non abbiamo dati all'inizio!).

Come sempre attendiamo un evento, dopodiché lo interpretiamo con uno switch.

Nel caso in cui il **network abbia frame da spedire** [49] allora preleviamo il dato e lo mettiamo nel buffer corretto rispettivamente al suo numero di sequenza (il prossimo che dovremo inviare è quello che stiamo prelevando, lo modularizziamo circolarmente su NR_BUFS). Incrementiamo quindi il numero di buffer di uscita occupati [51] (ci basta la variabile `nbuffered`), spediamo il frame [52] e incrementiamo la variabile `next_frame_to_send` [53].

Nel caso in cui **arrivi un frame** [56] invece vediamo subito quale sia il tipo del frame .

Nel caso di un **frame data** ne guardiamo come al solito il numero di sequenza [60]: se tale numero è diverso dal `frame_expected` e non abbiamo ancora inviato alcun NACK, allora inviamo un NACK per il `frame_expected` [61]. Questo perché, qualsiasi sia il numero di sequenza, se non è il `frame_expected` allora sicuramente non abbiamo ricevuto il `frame_expected` (poiché i frame sono sempre in ordine) e quindi mandiamo un NACK per lui, poiché è il primo frame che non abbiamo.

Se invece è il `frame_expected`, allora facciamo partire un `ack_timer` [63] poiché abbiamo ricevuto il frame giusto e ne dovremo spedire l'ACK: ma se il network non dovesse avere pacchetti da mandare non passeremo mai nel pezzo di codice che ha la `send` ([52]) e quindi facciamo partire questo timer che eventualmente ci porterà un nuovo evento (esamineremo meglio dopo).

Verifichiamo dunque che il frame appena arrivato sia compreso nella finestra e non sia una ripetizione (non sia ancora arrivato) [65] e nel caso in cui sia così, lo impostiamo come arrivato e lo copiamo nel buffer alla posizione corretta (sempre con la solita tecnica della circolarità).

Entriamo poi in un while molto importante [69]: controlliamo dal `frame_expected` in poi quali frame sono arrived, quindi stiamo in pratica prendendo tutti i frame "contiguamente arrivati" a partire dal `frame_expected` e li inviamo al network [70]. Ristabiliamo la variabile `no_nack` (abbiamo appena ricevuto un frame buono e l'abbiamo trasmesso quindi di certo se avevamo mandato NACK per qualcosa, oramai è passato), svuotiamo il buffer (abbiamo inviato al network!) indicandone il false su `arrived` [72] e poi incrementiamo la finestra. Dopodiché facciamo partire l'`ack_timer` [75], poiché manderemo ACK per questi frame in piggybacking se possibile, altrimenti li invieremo singolarmente (si veda dopo la parte del codice che gestisce l'`ack_timeout`). Notiamo che [75] "refresherà" se stesso molte volte durante il while, precisamente ad ogni giro. Ma non importa poiché gli ACK sono cumulativi! Ricevere ACK per il frame 4 è come riceverlo per il frame 0, 1, 2, 3 e infine 4.

Nel caso di un **frame NACK** [80] allora dobbiamo procedere alla ritrasmissione del frame (nel caso in cui esso sia nella finestra, come giustamente verifica l'`if`) [81].

Altrimenti abbiamo un **ACK** [83]. quindi, se è un ACK sensato (nella finestra) possiamo svuotare i buffer di invio [84] e stoppare i timer per i frame di cui abbiamo appena ricevuto l'ACK [85]. Dopodiché aumentiamo l'`ack_expected` (la finestra).

Nel caso di un **cksum_err** [89] allora se ancora non è stato fatto, inviamo un NACK.

Nel caso di un **timeout** [93] applichiamo pedestremente la selective repeat e quindi ritrasmettiamo solamente il frame di cui è scaduto il timer (sfruttando la variabile globale `oldest_frame`) [94].

Se abbiamo un **ack_timeout** [97] allora dobbiamo inviare un ACK senza far uso di piggybacking e quindi lo facciamo specificando il kind ACK [98].

Come sempre, al fondo, gestiamo il livello network [103]: se ci sono ancora buffer di invio disponibili siamo disposti ad accettare dal network dei frame da inviare, altrimenti disabilitiamo l'evento.

Nessuno è perfetto

Allora, finalmente abbiamo ottenuto un protocollo funzionale e funzionante al 100%?

No.

La procedura **to_network_layer** che invia un pacchetto potrebbe rovinarci i piani! Infatti se il network non avesse buffer per ricevere il dato, allora saremmo ancora una volta in una situazione di attesa attiva!

Il problema è che qui il concetto di "ho ricevuto, tutto bene" e di "puoi ancora inviare" sono fusi insieme nel singolo messaggio di ACK, intrinsecamente.

4.10) Attesa non deterministica - metodi alternativi alla wait

Oltre che utilizzare la `wait(&event)` - che attende un evento e se ve ne sono di multipli li mette in coda e li esamina uno alla volta - potremmo voler adottare una tecnica di **polling**.

Si ha un array dei possibili eventi: viene effettuato ciclicamente un controllo su ogni evento e si verifica quindi se è accaduto o meno. Se l'evento è occorso, allora si esegue la parte di codice relativa (per gestirlo), se non è occorso allora si passa a controllare l'evento successivo. Questo significa avere un **busy waiting**: la CPU sarebbe sempre impegnata a ruotare sugli eventi, quindi questa soluzione è ottimale solo in quei casi in cui la scheda di rete ha una piccola unità di calcolo dedicata, appunto, a questo tipo di operazioni.

5) Esempio di protocollo data link: HDLC

Come primo esempio di protocollo prendiamo l'**HDLC** (High-Level Data Link Control) che nasce dall'evoluzione del protocollo **SDLC** (Synchronous Data Link Control) della IBM.

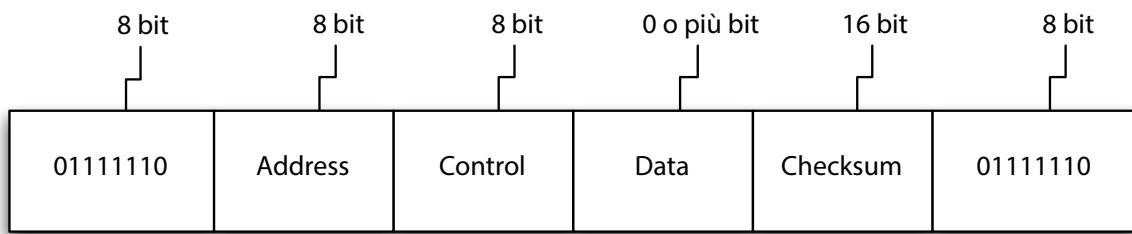
È un protocollo standardizzato da ANSI e da ISO, quindi possiamo dire essere il protocollo "ufficiale" di data link (ma non quello adoperato in Internet, in realtà!).

Il CCITT ne crea una versione modificata, il LAPB, che poi lo modifica nuovamente per riavvicinarlo a HDLC.

Si tratta di un **protocollo orientato ai bit** (non si ipotizza la trasmissione di byte), facente uso di **bit stuffing**.

5.1) La struttura del frame: introduzione

Vediamo ora come sono strutturati i frame nella trasmissione con protocollo HDLC.



All'inizio e alla fine troviamo le **flag** così come le abbiamo studiate nel capitolo 2.3 (bit stuffing).

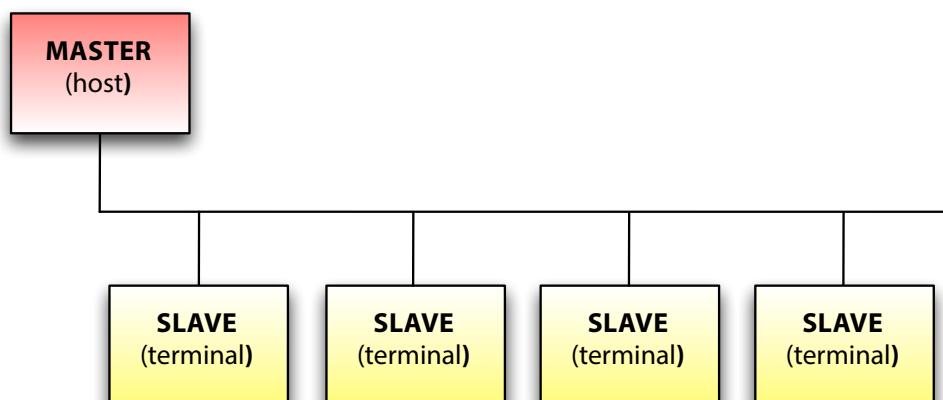
Abbiamo poi:

- un campo **address** che è adoperato nelle linee con architettura master-slave (trattate nel paragrafo successivo) per definire a quale slave stiamo parlando,
- un campo **control** che differenzia le varie classi di frame,
- un campo **data** che contiene i dati da trasferire,
- un campo **checksum** che contiene il codice CRC per il rilevamento degli errori (come visto nel capitolo 3.7).

Quando non ci sono dati, vengono inviati frame "vuoti" cioè solo sequenze di flag, per mantenere le parti **sincronizzate** (infatti il protocollo è detto sincrono!).

Architettura master - slave

Questo protocollo, per via delle sue caratteristiche intrinseche (anche se poi in seguito è stato adattabile per altre architetture), si presta molto bene per funzionare un'architettura **master slave**.



Una macchina funge da master, quindi controlla tutti i terminali.

La macchina master effettua **polling** continuo tra i terminali chiedendo se hanno qualcosa da trasmettere. Se la macchina master deve trasmettere, allora trasmette immediatamente facendo uso del campo address per indirizzare un terminale preciso.

I terminali fra loro non possono comunicare.

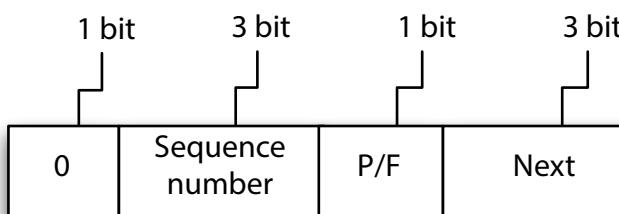
5.2) La struttura del frame: il campo control

Il campo control trasporta ulteriori informazioni e definisce la tipologia di frame. A seconda del campo control abbiamo tre frame diversi:

- **frame di informazioni**
- **frame di supervisione**
- **frame senza numero**

Vediamoli nel dettaglio.

Frame di informazioni



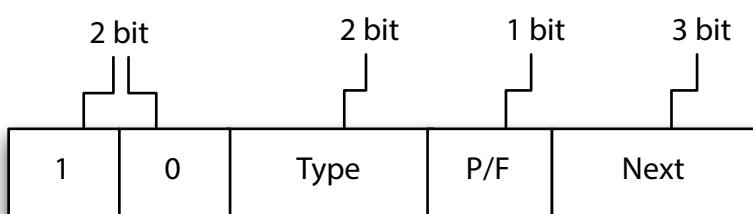
Questo frame trasporta **dati**.

Viene adottata una sliding window con 3 bit per i numeri di sequenza (che sono contenuti nel campo **sequence number**).

Il campo **next** è invece sfruttato per fare piggybacking (contiene quindi un ack). Come detto sopra, nell'architettura master-slave avviene polling continuo, quindi il campo P/F (Poll/Final) verrà sfruttato dal master per dire allo slave (tramite un messaggio **poll**) che può iniziare a trasmettere.

Dal canto suo, il terminale vorrà tendenzialmente mandare messaggi più grandi di un frame, quindi manderà tutti i frame con poll tranne l'ultimo che avrà **final** a segnalare che il messaggio è concluso.

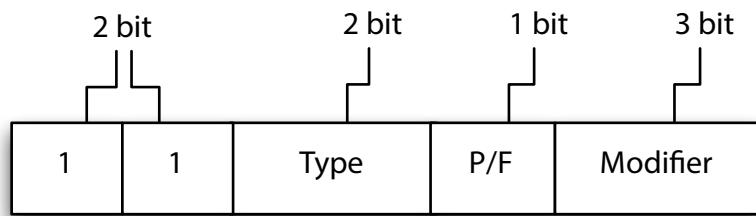
Frame di supervisione



Questo frame è utilizzato per scambiare **informazioni di controllo**.

A seconda del campo type differenziamo quale informazione stiamo trasportando.

- Type = 0 (in bit: 00): **frame di ACK (receive ready)** che segnala il prossimo frame atteso, viene adoperato quando non si riesce a fare piggybacking. Il campo next contiene l'ACK.
- Type = 1 (in bit: 01): **frame di NACK (reject)** sfruttato nei protocolli go back N che comunica la necessità di ritrasmettere tutto a partire da next.
- Type = 2 (in bit: 10): **frame di ACK (receive not ready)** funziona allo stesso modo dell'ACK ma risolve il problema esposto nel paragrafo "Nessuno è perfetto" ovvero si conferma la ricezione di un certo frame ma non si permette l'invio di altri (tendenzialmente perché i buffer sono pieni). La situazione verrà sbloccata con un *receive ready* o un *reject*, a seconda dei casi. Il campo next contiene l'ACK.
- Type = 3 (in bit: 11): **frame di NACK (selective reject)** sfruttato nei protocolli selective repeat, comunica quale frame ritrasmettere nel next.

Frame senza numero

Questo frame è utilizzato per il trasporto di **comandi**.

I comandi servono per portare informazioni generiche ma essenziali per il funzionamento delle varie macchine insieme.

- **DISC** (*Disconnect*): la macchina segnala che si sta per disconnettere.
- **SNRM** (*Set Normal Response Mode*): in architettura master - slave un terminale segnala il suo rientro.
- **SABM** (*Set Asynchronous Balanced Mode*): in architettura bilanciata e asincrona, segnala il rientro di una macchina (quindi in un sistema paritario) e quindi avviene un reset della linea.
- **FRMR** (*FRaMe Rejected*): segnala la ricezione di un frame che non ha avuto errori sul canale ma risulta no-sense rispetto alla logica del protocollo (ad esempio un frame con tipo non previsto dallo standard). Per l'ACK su questo tipo di frame si adopera un tipo di ACK speciale (detto UA - Unnumbered Acknowledgement).
- **UI** (*Unnumbered Information*): scambio di informazioni fra data link.

6) Esempio di protocollo data link: PPP

PPP (Point to Point Protocol) è un protocollo adoperato in Internet.

In Internet si utilizza una connessione punto-punto quando si ha una connessione fra router oppure quando c'è una connessione tra PC e ISP (adoperando il modem come intermediario).

6.1) PPP: LCP e NCP

PPP nasce dall'RFC 1661 come un meccanismo di framing multi protocollo adatto alla trasmissione dati via modem, alle linee seriali ed ad altri livelli fisici.

Vengono forniti: negoziazione, rilevazione degli errori, supporto per più protocolli, negoziazione IP, autenticazione e framing di inizio/fine.

Parte integrante del protocollo sono **LCP** (Link Control Protocol) che si occupa della connessione, test di linea, negoziazione e opzioni di collegamento e disconnessione e l'**NCP** (Network Control Protocol) che si occupa delle negoziazioni relative allo strato di network.

6.2) PPP: i requisiti

Nella definizione di PPP ci si è posti alcuni obiettivi:

- Il **framing** consiste nella creazione di un frame su un mezzo fisico che non dispone di questa funzionalità (quella dei frame), ovvero si cerca da una linea seriale di ottenere il concetto di frame che altrimenti non esisterebbe.
- Permette il **multiplexing di strati di network differenti** (e quindi il demultiplexing relativo).
- C'è **error detection** ma non correction (il livello fisico non è rumoroso)
- C'è **connection liveness** per rilevare/segnalare (al network) la caduta del link
- È disponibile la possibilità di **negoziare l'indirizzo del network**.

6.3) PPP: i non requisiti

È esplicitamente richiesto che il protocollo non abbia:

- Correzione degli errori
- Controllo di flusso
- Consegnata ordinata (se ne occupa il livello fisico)
- Link multipunto (è point to point! - no polling)

Tutte queste funzionalità sono lasciate ai livelli superiori. C'è da porre particolare attenzione al fatto che mancando il flow control, esiste la concreta possibilità di perdere frame per mancanza di buffer.

6.4) Scenario di funzionamento

Vediamo per punti cosa accade nel momento della connessione e della disconnessione:

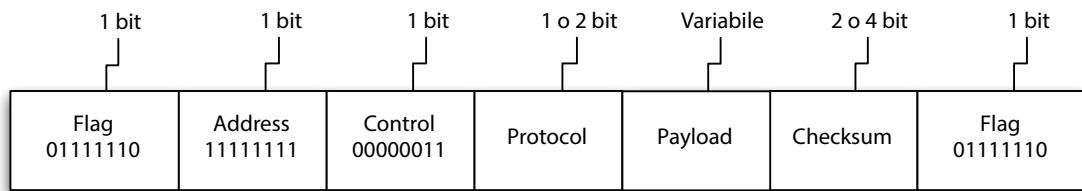
Connessione:

- Il PC chiama il router del provider tramite modem.
- Il provider risponde.
- Il PC manda al provider una serie di pacchetti LCP nel payload di uno o più frame PPP (si stabiliscono i parametri PPP)
- Il PC invia una serie di pacchetti NCP per configurare il network, si configurano gli indirizzi IP degli endpoint.
- Connessione effettuata , il PC è connesso alla rete IP.

Disconnessione:

- NCP termina la connessione liberando l'indirizzo IP dalle due parti
- LCP chiude la connessione datalink
- Il computer chiude il modem

6.5) Struttura dei frame

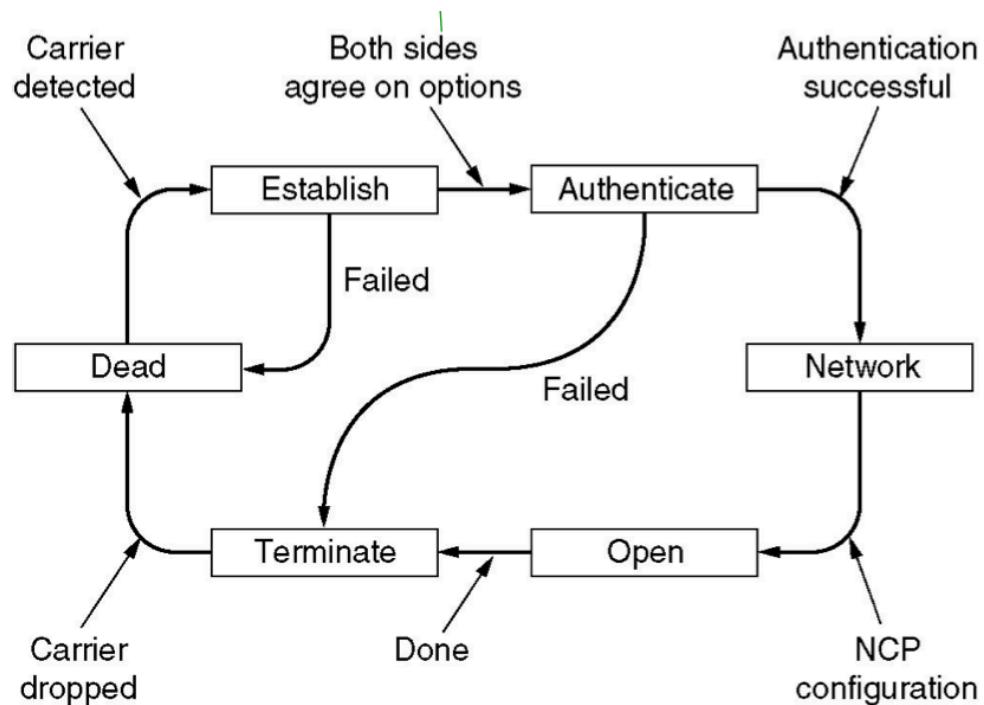


Dato che PPP nasce quando HDLC era già in auge, il frame è molto simile a quello già esaminato per lo scorso protocollo. La flag è corrispondente, il campo address e control sono inutili (valore fisso) ma vengono tenuti per non mutare troppo la struttura del flag: si può negoziare il loro non utilizzo (in fase di connessione).

Il campo protocol esprime su quale protocollo si baserà il payload.

Il payload ha dimensione massima 1500 byte (come il limite di trasmissione di ethernet) e contiene il dato. Checksum e flag si comportano allo stesso modo dell'HDLC.

6.5) Diagramma di flusso: la vita del protocollo



7) Esercizi sul livello data link

7.1) Esercizi sul framing

Esercizio 1

Un protocollo data link usa la seguente codifica di caratteri (esadecimale):

- A = 47,
- B = E3,
- FLAG = 7E,
- ESC = E0.

Trovare la sequenza di bit trasmessi, in binario e in decimale per il frame di 4 caratteri: **A B ESC FLAG** per ognuno dei seguenti metodi di framing:

1. Conteggio caratteri
2. Flag byte con byte stuffing
3. Flag byte di inizio e fine con bit stuffing

Es. 1 - Risposta 1

La tecnica del conteggio di caratteri prevede di inserire nell'head un certo quantitativo di bit per indicare il numero di bit che sono contenuti nel frame (precisando, a seconda del protocollo, se quel numero comprende anche l'head stesso oppure no).

Quindi introduciamo nell'head un byte (decidiamo noi la dimensione, ma con un byte possiamo delimitare frame lunghi 255, poiché con 8 bit contiamo fino a 255) dal valore: **0000 0100** che in esadecimale è **04** (abbiamo optato per non tener conto dell'head nel conteggio, quindi {A, B, ESC, FLAG} sono quattro caratteri).

In totale abbiamo (esadecimale): **04 47 E3 E0 7E**.

Equivalenti al binario: **0000 0100 ~ 0100 0111 ~ 1110 0011 ~ 1110 0000 ~ 0111 1110**.

(NB: per convertire da binario ad esadecimale basta prendere ogni cifra e trasformarla in binario e poi affiancarle).

Es. 1 - Risposta 2

La tecnica del flag byte con byte stuffing prevede di inserire una flag all'inizio e una alla fine del frame, anteponendo, all'interno del frame, ad ogni carattere di flag o di escape un carattere di escape.

Abbiamo quindi (esadecimale): **7E 47 E3 E0 E0 E0 7E 7E**.

Equivalenti al binario:

0111 1110 ~ 0100 0111 ~ 1110 0011 ~ 1110 0000 ~ 1110 0000 ~ 0111 1110 ~ 0111 1110.

Es. 1 - Risposta 3

La tecnica del flag byte con bit stuffing prevede di inserire una flag all'inizio e una alla fine del frame, facendo seguire, all'interno del frame, ad ogni sequenza di cinque bit a 1 uno zero.

Abbiamo quindi: **0111 1110 0100011111010001111100000011111010 0111 1110.**

Sono stati applicati tre stuffing (gli zeri sottolineati).

Esercizio 2

La seguente sequenza si trova nel mezzo di un flusso di dati da trasmettere usando l'algoritmo di byte stuffing:

A B ESC C ESC FLAG FLAG D

Qual è l'output dopo il byte stuffing?

E se invece fosse un flusso in ricezione, che frame (o porzione di frame) sto ricevendo? Che analisi fareste del flusso?

Es. 2 - Risposta 1

Dato che siamo nel mezzo di un flusso di dati, non mettiamo le flag all'inizio e alla fine del frame.

Introduciamo però un carattere di escape davanti a ogni ESC e FLAG che troviamo.

Abbiamo quindi: **A B ESC ESC C ESC ESC ESC FLAG ESC FLAG D.**

Es. 2 - Risposta 2

In lettura, il carattere di escape ci dice: "prendi il carattere successivo così com'è, senza interpretarlo".

Quindi abbiamo:

A B ESC C ESC FLAG FLAG D --> A B C FLAG e poi l'altra flag viene letta come chiusura del frame, perciò viene inviato al livello superiore "**A B C FLAG**".

7.2) Esercizi sul codice di Hamming**Esercizio 1**

Qual è il valore in binario dopo la codifica di Hamming con parità pari per il singolo byte AF (esadecimale)?

Es. 1 - Risposta

Innanzi tutto, trasformiamo AF in binario: **1010 1111**.

Dopodiché li sistemiamo partendo da destra, contando da uno, lasciando libere le posizioni che sono potenze di due.

Bit	1	0	1	0	x	1	1	1	x	1	x	x
<i>Posizione</i>	12	11	10	9	8	7	6	5	4	3	2	1

Prendiamo quindi le posizioni con bit a 1 e le sommiamo in binario (XOR):

Pos.	Binario	Riempiamo quindi le "x" leggendo da sinistra verso destra :													
3	0011 +														
5	0101 +	<table border="1"> <tr> <td>Bit</td> <td>1</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>1</td> <td>1</td> <td>1</td> <td>0</td> <td>1</td> <td>0</td> <td>1</td> </tr> </table>	Bit	1	0	1	0	0	1	1	1	0	1	0	1
Bit	1	0	1	0	0	1	1	1	0	1	0	1			
6	0110 +														
7	0111 +														
10	1010 +														
12	1100 =														
#	0001														

7.3) Esercizi sulle velocità di trasmissione

Esercizio 1

Abbiamo tre nastri da 7GB da trasmettere con una rete a 150 Mbps.

In alternativa possiamo usare un trasporto terreno a 18 km/h.

A quale distanza conviene usare uno piuttosto che l'altro sistema?

Es. 1 - Risposta

Passiamo da GB a Byte e quindi a bit per poterli confrontare coi Mbps.

$$7 \cdot 3 \text{ (nastri)} \cdot 10^9 \text{ Byte} = 21 \cdot 10^9 \text{ Byte} = 21 \cdot 10^9 \cdot 8 \text{ Bit} = 16,8 \cdot 10^{10} \text{ bit.}$$

Li trasmettiamo a 150 Mbps = $15 \cdot 10^7$ bps e quindi abbiamo un tempo di trasmissione totale
 $t = 16,8 \cdot 10^{10} / 15 \cdot 10^7 = 1,12 \cdot 10^3$ secondi.

Se usassimo il mezzo, in quello stesso tempo trasporteremmo ($18 \text{ km/h} = 0,005 \text{ km/sec}$) $0,005 \cdot 1,2 \cdot 10^3 = \mathbf{5,6 \text{ km.}}$

Con una tratta di 5,6 Km i due metodi sono esattamente equivalenti. Se aumentiamo la distanza, il trasporto fisico diventa meno conveniente, mentre se la diminuiamo esso diventa più conveniente.

Reti di Elaboratori

Il sotto-livello MAC

Capitolo 4

Enrico Mensa,

Basato sulle lezioni del prof. [Matteo Sereno](#)

1) MAC - Medium Access Control

1

1.1) Canali multiaccesso

Il caro vecchio multiplexing?

1.2) Effetto contesa per il canale

Calcolo dell'inefficienza del multiplexing

2) Protocolli MAC

2

2.1) Il protocollo ALOHA

Probabilità di non collisione

2.2) Il protocollo slotted ALOHA

2.3) Rilevare la portante: CSMA 1-persistente

2.4) Rilevare la portante: CSMA non persistente

2.5) Rilevare la portante: CSMA p-persistente

2.6) I CSMA: riassunto delle efficienze

2.7) I CSMA con Collision Detection (CSMA/CD)

Rilevare una collisione

Nessuna garanzia di consegna

3) LAN Ethernet (IEEE 802.3)

6

Nozioni storiche

Le connessioni

Topologie

Hub e ripetitori

La codifica Manchester

3.2) Gli indirizzi: MAC

3.3) La struttura del frame

3.4) L'algoritmo CSMA/CD

L'exponential backoff

3.5) Analisi delle prestazioni

Formule per valutare l'efficienza

3.6) Interconnessioni

Gli hub

Gli switch

3.7) Le versioni successive

Fast Ethernet (IEEE 802.3u)

Gigabit Ethernet (IEEE 802.3ab)

3.8) Il perché della vittoria di Ethernet

4) LAN Wireless (IEEE 802.11)

14

4.1) Introduzione

I protocolli

L'architettura

I canali

La giungla Wi-Fi

L'associazione (con scansione)

4.2) Architettura e protocolli

Il CSMA/CA (Collision Avoidance)

Un nuovo modo di operare: DCF (distributed coordination function)

La stazione nascosta e la stazione esposta

La rilevazione: fisica e virtuale

4.3) Il protocollo MACAW

Meccanismi aggiuntivi a CSMA/CA

La frammentazione

La qualità del servizio

4.4) Il frame 802.11

4.5) Servizi

Servizio di associazione

Servizio di riassociazione

Servizio di autenticazione

Servizio di distribuzione

Servizio di integrazione

5) WiMAX - Wireless a banda larga (IEEE 802.16)

22

5.1) Le differenze rispetto a 802.11

5.2) Lo stack

5.3) Il livello fisico

5.4) Il frame 802.16

6) Il Bluetooth

24

6.1) Introduzione

6.2) "Una" pila complicata

6.2) La pila generica

Livello fisico

Livello baseband

Livello L2CAP

6.3) Il frame 802.15

7) Il problema della commutazione

27

7.1) Il bridge

L'apprendimento ed il problema dei cicli

7.2) I bridge remoti

7.3) Tutto questo commutare: bridge, switch, hub o router?

7.4) Le Virtual LAN (VLAN)

Il funzionamento della VLAN

8) Esercizi finali

31

8.1) Problema su un canale di trasmissione: teoria delle code

8.2) Calcolo del numero di stazioni Aloha supportabili

8.3) Calcolo del padding del frame Ethernet

8.4) Trasmissione simultanea fra reti 802.11

8.5) Acquisizione degli indirizzi MAC da parte di un bridge

1) MAC - Medium Access Control

Il livello MAC, seppur venga trattato in seguito del livello data link, ne fa invece parte e si frappone tra l'LLC (la parte del data link appena vista) ed il livello fisico (quindi "temporalmente" viene prima di tutte le altre operazioni viste nel capitolo precedente).

1.1) Canali multiaccesso

Il problema che il livello MAC si prefigge di risolvere è l'accesso al canale da parte di multiple entità. Ad esempio, come usiamo il **broadcast**? Bisogna **coordinare** ed è di questo che si occupa il MAC.

Il caro vecchio multiplexing?

Perché non adoperare multiplexing per coordinare le varie entità?

Per via dell'assegnazione statica adottata dal multiplexing, avremmo delle gravi **inefficienze intrinseche**.

La banda viene suddivisa fra N calcolatori ma dato che il traffico è irregolare, in un momento magari sarebbero utili tutte e N, mentre in un altro ne sarebbero utili di meno: viene sprecata della banda.

1.2) Effetto contesa per il canale

Introduciamo qualche concetto della teoria delle code per quantificare l'inefficienza e per capire di più il problema. Adotteremo la metafora dell'ufficio postale.

Vogliamo calcolare il ritardo medio **T** (dall'istante in cui si entra in posta a quello in cui si esce) per un canale di capacità **C** bit/sec su cui arrivano **λ** frame/sec di dimensione **D**.

Consideriamo D variabile casuale a distribuzione esponenziale con parametro μ , quindi si ha un numero medio di frame al secondo pari a λ e la dimensione media dei frame è $1/\mu$.

Con la teoria delle code dimostriamo che:

$$T = \frac{1}{\mu C - \lambda}$$

Quindi, con un canale $C = 100 \text{ Mbit/s}$ e $1/\mu = 10^4 \text{ e } \lambda = 5 * 10^3$ abbiamo che:

$$T = \frac{1}{\frac{1}{10^4} * 10^8 - 5 * 10^3} = \frac{1}{10^4 - 5 * 10^3} = \frac{1}{5 * 10^3} \text{ sec} = \frac{10^6}{5 * 10^3} \mu\text{s} = 200 \mu\text{s}$$

Quindi il tempo effettivo senza contesa è di 100μ viene raddoppiato dal problema della contesa!

La coda è **limitata** (non infinita) solo se $\lambda / (\mu * C) < 1$ (cioè la coda viene smaltita). [Nel paragrafo 8, un esercizio simile].

Calcolo dell'inefficienza del multiplexing

Se adottassimo multiplexing, avremmo:

$$T_{FDM} = \frac{1}{\mu(C/N) - (\lambda/N)} = \frac{N}{\mu C - \lambda} = NT$$

N volte il T di prima dove N è il numero di suddivisioni che facciamo della banda (la banda va a C/N con numero di frame N-mezzati ovvero λ/N).

2) Protocolli MAC

I protocolli MAC si suddividono tre tipologie a seconda di come si voglia regolare l'accesso alle stazioni:

- **Round robin**: viene dato un quanto di tempo in cui una sola entità può comunicare
- **Reservation**: ogni calcolatore deve prenotare uno slot di tempo
- **Contention**: un calcolatore che deve trasmettere lo fa senza curarsi degli altri (con eventuale gestione delle collisioni)

2.1) Il protocollo ALOHA

È stato il primo protocollo a nascere, è molto molto semplice. È di tipo contention.

Tutte le stazioni inviano senza curarsi degli altri: se avviene una collisione il frame viene ritrasmesso a un tempo randomico (per evitare che avvenga una ritrasmissione simultanea).

Se abbiamo infinite stazioni e complessivamente il fenomeno di generazione dei frame è in accordo con una distribuzione di Poisson con media N frame ogni "tempo di frame" (tempo necessario alla trasmissione di un frame). Se N è maggiore di uno allora avremo certamente più frame inviati di quanti il canale possa ospitare e quindi la collisione sarà sicura. Bisogna avere un **throughput** ragionevole con $0 < N < 1$.

Quando avviene una collisione, le trasmissioni si adoperano per effettuare una ritrasmissione.

Supponiamo che vengano trasmessi k frame ogni tempo di frame (ogni istante t), dove k è una variabile aleatoria con distribuzione di Poisson e media G .

Chiaramente $G \geq N$ poiché vi sono le ritrasmissioni. Con bassi livelli di carico, ovvero basse collisioni, G ed N saranno molto vicini, mentre per alti livelli di carico G si discosterà molto da N .

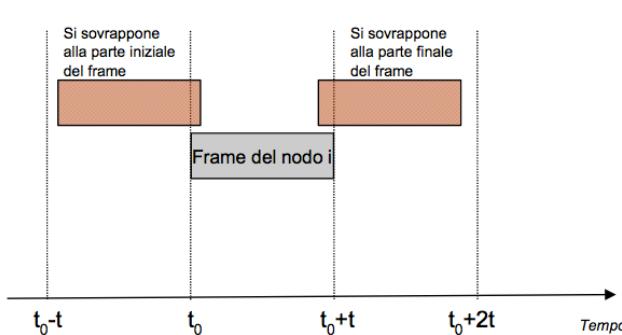
Il **throughput S** è pari al numero di frame (da G) che non subiscono collisioni cioè quelli effettivamente arrivati.

Quindi avendo come P_0 la probabilità che un frame non subisca collisioni, $S = G * P_0$.

Quanto vale P_0 ?

Probabilità di non collisione

Banalmente, un frame non subirà collisioni solamente se durante la sua trasmissione nessun altro frame verrà trasmesso.



Il **periodo di vulnerabilità** è quello che va da t_0-t fino a t_0+t .

Questo poiché se un frame inizia a trasmettere in un istante qualsiasi compreso nell'intervallo, allora sicuramente entrerà in collisione con il frame i -esimo.

Quindi il periodo è $2t$.

Considerando la formula data dalla distribuzione di Poisson $P(k) = G^k e^{-G} / k!$ quindi la probabilità che non vi siano collisioni è pari a $P(0) = e^{-G}$. Dato che non vogliamo collisioni per due tempi di frame, $S = G * e^{-2G}$

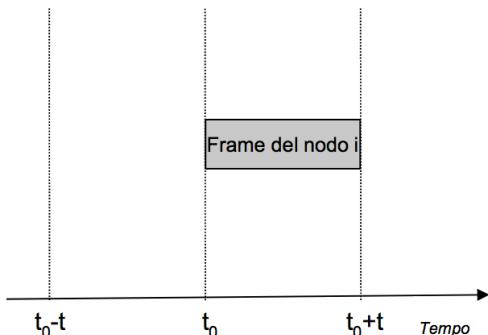
Quindi si può sperare di utilizzare al massimo **il diciotto per cento** del canale! Pochissimo!

Come fare per migliorarlo? Potremmo discretizzare il tempo... vediamolo.

2.2) Il protocollo slotted ALOHA

Possiamo raddoppiare S semplicemente discretizzando il tempo.

Il frame vengono inviati a intervalli regolari, ad ogni scattare di t . Se un frame arriva per essere inviato tra un t ed un'altro, attenderà lo scattare del prossimo clock.

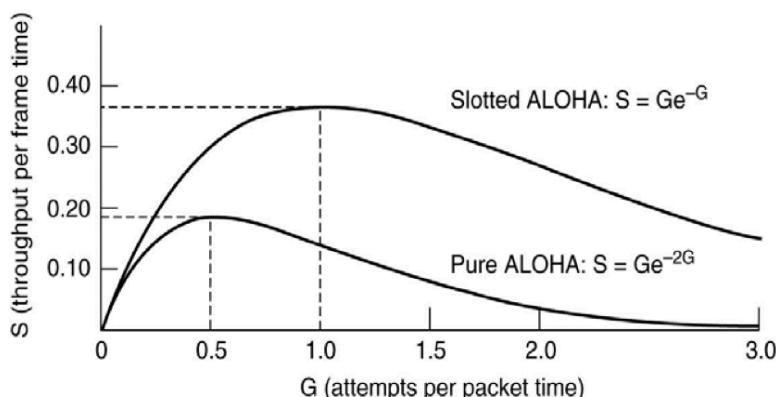


Adesso il **periodo di vulnerabilità** è dimezzato: va da t_0-t fino a t .

Infatti l'arrivo di un frame da t_0 a t_0+t non è un problema poiché tale frame verrà inviato al clock successivo (t_0+t).

Questo ci porta ad un **$S = 36\%$**

Un aumento di G porta in ogni caso ad un aumento esponenziale delle collisioni, come illustra questo grafico:



Ma non ci basta. Vediamo cos'altro potremmo fare.

2.3) Rilevare la portante: CSMA 1-persistent

Se prima di procedere all'invio si potesse ascoltare cosa sta passando sul mezzo, allora potremmo cercare di ottimizzare le nostre azioni.

La prima tecnica adottabile è quella di stare **continuamente in ascolto** sul canale e appena c'è un "buco", **trasmettiamo** (1-persistent, appunto, trasmette con **probabilità 1**). Come possiamo immaginare è una tecnica non molto efficiente infatti le collisioni sono ancora molto frequenti, vengono gestite proprio come nell'ALOHA (ritrasmissione a tempo casuale).

Si noti che in questo protocollo sono molto importanti le dimensioni della rete! Con una rete molto grande, potremmo non renderci conto che c'è qualcosa sul mezzo anche se in effetti "sta arrivando" e quindi trasmettere ugualmente.

2.4) Rilevare la portante: CSMA non persistente

Se invece di inviare in ogni caso decidessimo, a fronte di aver incontrato il mezzo fisico vuoto, di far scorrere un quantitativo di tempo **random prima di trasmettere**, allora potremmo diminuire certamente il numero di collisioni. Non staremo più continuamente in ascolto del canale ma controlleremo ogni t . In questo caso però aumenteremmo sicuramente i ritardi.

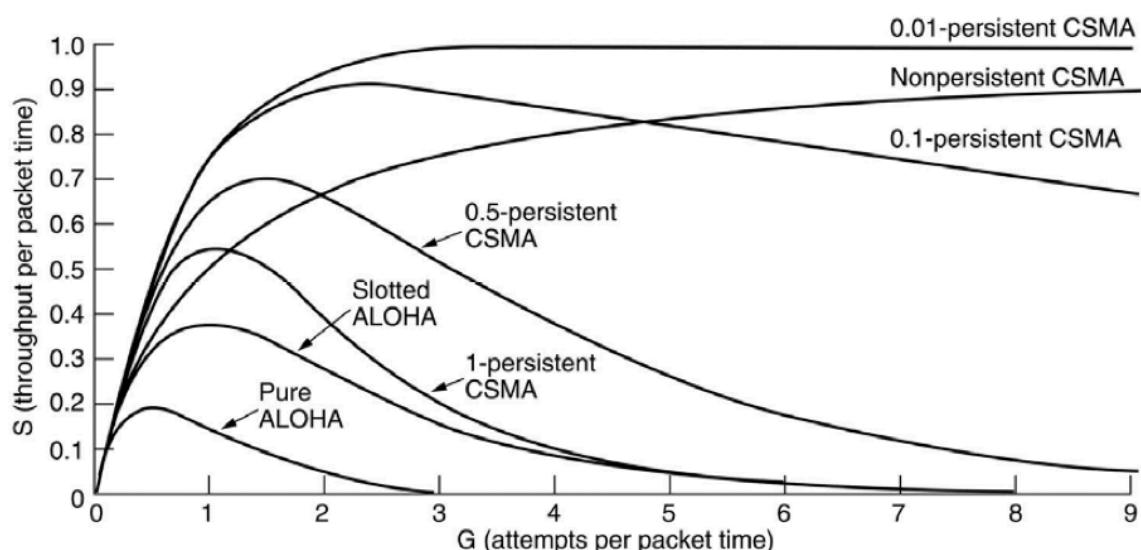
2.5) Rilevare la portante: CSMA p-persistente

La tecnica non persistente introduce degli inevitabili ritardi dettati dal fatto che non entriamo nel canale appena è libero.

Allora possiamo parlare della tecnica p-persistent, in cui se una stazione trova **libero il canale**, allora **trasmette con probabilità p** e rimanda la trasmissione con probabilità q (1-p). Questo controllo viene effettuato ad oltranza finché il canale non viene trovato libero.

Durante i tentativi successivi, nel caso di canale occupato, si attende un tempo casuale prima di controllare nuovamente se sia possibile procedere con l'invio a probabilità p.

2.6) I CSMA: riassunto delle efficienze



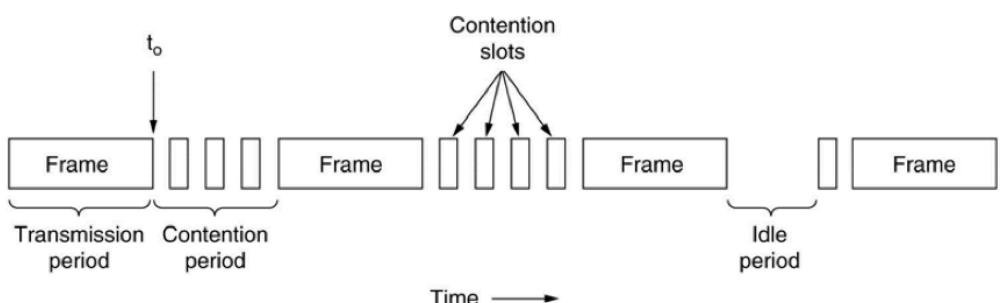
Come vediamo è molto semplice aumentare l'efficienza anche se questo **non ci garantisce** affatto di tenere il **canale occupato**. Dobbiamo considerare anche quell'aspetto!

2.7) I CSMA con Collision Detection (CSMA/CD)

Cerchiamo di risolvere il problema delle collisioni interrompendo immediatamente la trasmissione quando ne rileviamo una. È importante effettuare questo tipo di operazione per riportare il canale ad uno stato di quiete il più in fretta possibile.

Rilevare una collisione

Punto fondamentale per rilevare una collisione è la possibilità di **ascoltare ed inviare** insieme sul canale. Senza questa prerogativa, non possiamo parlare di rilevazione di collisione.



In seguito ad ogni invio segue un periodo di contesa, durante il quale può avvenire la collisione.

Per rilevare la collisione si effettua un controllo tra il segnale in entrata e quello in uscita: se c'è differenza fra i due, allora evidentemente qualche segnale si è sommato al proprio (c'è stata collisione) quindi si deve avvisare l'altro interlocutore.

Quanto tempo ci vuole per rilevare una collisione?

Se τ è il tempo che il segnale ci mette ad essere propagato fra i due punti più distanti della rete, allora 2τ sarà la dimensione del contention slot (il tempo di contesa - pericoloso!) poiché la collisione deve essere prima rilevata (τ al massimo) e poi deve essere inviata notifica all'altro interlocutore (un altro τ).

Se il frame non fosse sufficientemente lungo, allora colui che trasmette finirebbe di inviare il proprio frame prima che arrivi dall'altra parte, quindi, potenzialmente, prima che accada una collisione rendendola così irrilevabile.

In buona sostanza il frame deve "durare tutto il percorso" affinché il primo bit arrivi a destinazione prima che l'ultimo sia già stato messo sul livello fisico dal sender di modo che questo possa ancora sentire ciò che invia e ciò che riceve, potendo così fare il confronto.

Teniamo bene a mente questa problematica perché sarà motivo della conformazione del frame nel protocollo 802.5.

Nessuna garanzia di consegna

Si noti che non c'è garanzia di consegna, infatti non esiste controllo di flusso e i due interlocutori potrebbero non avere abbastanza buffer!

3) LAN Ethernet (IEEE 802.3)

3.1) Introduzione

La tecnologia LAN Ethernet è la più diffusa attualmente per quanto riguarda le reti LAN.

Adotta un protocollo MAC CSMA/CD 1 - persistente. La versione attuale è 10Mbit/sec.

Le versioni vengono standardizzate sul nome:

<Velocità di trasmissione>, <Modalità di trasmissione>, <Lunghezza del segmento>.

Abbiamo quattro tipologie di mezzi trasmissivi (**cable**):

Name	Cable	Max. seg.	Nodes/seg.	Advantages
10Base5	Thick coax	500 m	100	Original cable; now obsolete
10Base2	Thin coax	185 m	30	No hub needed
10Base-T	Twisted pair	100 m	1024	Cheapest system
10Base-F	Fiber optics	2000 m	1024	Best between buildings

Annotations:

- A green box labeled "Con spine a vampiro" (With vampire teeth) points to the "Max. seg." column of the 10Base5 row.
- A green box labeled "Con connessioni BNC (più affidabili e semplici)" (With BNC connections (more reliable and simple)) points to the "Cable" column of the 10Base2 row.
- A green box labeled "Basati su hub e doppini telefonici" (Based on hubs and telephone jacks) points to the "Cable" column of the 10Base-T row.

Nozioni storiche

La prima Ethernet nasce con cavo coassiale da mezzo pollice (giallo!) con spine a vampiro che bucavano il mantello protettivo ed entravano direttamente sul cavo. Il brevetto era DIX, Digital Intel Xerox.

Le connessioni

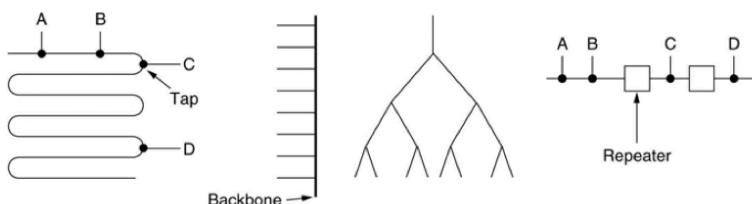
Le connessioni sono effettuate tramite l'ausilio di un chip, detti **NIC (Network Interface Card)** che constano di:

- Una piccola RAM
- un DSP (digital signal processor)
- un'interfaccia con il BUS dell'host e con la rete

Tramite il NIC è possibile effettuare, quindi, connessioni Ethernet.

Topologie

Le reti LAN assumono le seguenti topologie:



La massima distanza fra due **transceiver** è di 2.5Km con un numero massimo di ripetitori attraversabili pari a quattro.

Hub e ripetitori

Introduciamo gli importanti concetti di **hub** e di **ripetitore**.

Un **hub** unisce più segmenti di cavo: ciò che arriva da un cavo viene trasmesso a tutti gli altri. Non effettua amplificazione. L'utilizzo di hub genera spesso collisioni.

Un **ripetitore** invece unisce due (o tre) segmenti di cavo: ciò che arriva da un cavo viene amplificato (senza essere esaminato) e trasmesso all'altro segmento.

La codifica Manchester

Come visto nel capitolo 2 al paragrafo 2.4 possiamo sfruttare una codifica Manchester per codificare i dati durante la trasmissione (si ricorda che si trasmette con un baud mezzo bit!).

3.2) Gli indirizzi: MAC

Dato che le trasmissioni che effettuiamo non sono punto-punto è necessario poter identificare una macchina. Ci serve un indirizzo fisico (**MAC**) e un indirizzo logico (**IP**).

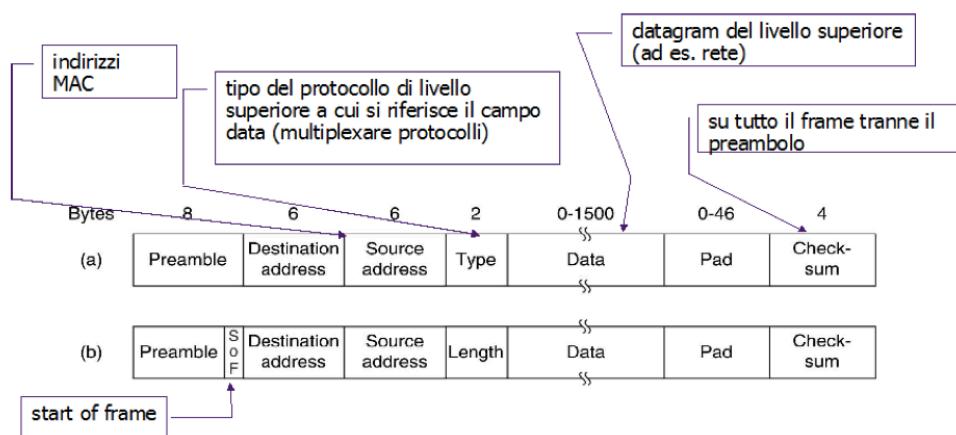
Il primo ci servirà sul livello datalink: si tratta di indirizzi a 48 bit (12 cifre esadecimali per comodità di scrittura). Il MAC è scritto sulla ROM della scheda di rete, quindi è unico ed inequivocabile. Il MAC è adoperato all'interno di una sottorete.

L'indirizzo logico è invece necessario per **astrarre dal MAC** e quindi far comunicare diverse sottoreti fra loro.

Metaforicamente il MAC è il codice fiscale di una persona mentre l'IP è il suo codice postale. Si noti infine che i MAC sono portabili (sono sulla ROM, cambiando posizione della macchina posso cambiare posizione del MAC) mentre gli IP non sono portabili, dipendono dalla sottorete a cui la macchina è connessa.

3.3) La struttura del frame

Vediamo ora la struttura di due frame, uno secondo la prima versione di Ethernet (DIX) e l'altra secondo la definizione 802.3 (standard IEEE).



Il **preamble** consta di alcuni byte adoperati per sincronizzare le due macchine (quindi sono una serie fissa). I primi sette sono: 01010101 mentre l'ultimo (SoF) è 10101011.

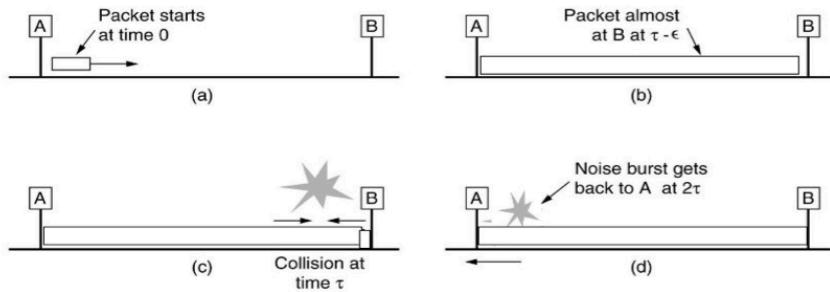
Il **destination address** serve per identificare il destinatario del frame mentre il **source address** è la fonte (sono entrambi indirizzi MAC).

Nel frame DIX abbiamo poi il campo **Type**, che serve a distinguere quale sovra-protocollo datalink dovrà gestire il frame. Infatti questo protocollo MAC è preposto per il funzionamento con diversi protocolli sovrastanti!

Nella definizione dell'IEEE viene invece impostato il campo **Length** che fornisce la lunghezza del frame.

I due protocolli sono comunque “compatibili” per via del fatto che i valori assunti da Type nel protocollo DIX sono tutti maggiori di 1536 e quindi le lunghezze dei frame saranno minori: sarà sufficiente interpretare il dato per capire se si tratta di una lunghezza oppure di un tipo.

Come annunciato precedentemente, il campo **Pad** serve per riempire il frame e renderlo di lunghezza minima 512 bit. Questo per il problema del non poter rilevare le collisioni se il frame è troppo piccolo.



Stiamo cercando di avere questa situazione: con una collisione poco prima di B, il frame è ancora in trasmissione (sufficientemente lungo) e quindi è possibile rilevare la collisione (d).

Infine il campo **checksum** contiene il solito codice CRC per la correzione degli errori.

Dopo il frame viene trasmesso silenzio per comunicare la fine del frame stesso (**non c'è trail**).

3.4) L'algoritmo CSMA/CD

Abbiamo detto che viene applicato un CSMA 1-persistente slotted. Vediamone le caratteristiche più dettagliatamente:

- L'adattatore non trasmette se sente che c'è una trasmissione in corso (funzionalità di **carrier sense**).
- Nel caso di collisione, il trasmettitore blocca la sua trasmissione (funzionalità di **collision detection**).
- La ritrasmissione avviene dopo un tempo casuale (**random access**).

L'algoritmo risulta quindi essere:

- 1) L'adapter riceve un datagram (da sopra) e crea un frame.
- 2) Se il canale è **idle** (libero), inizia la trasmissione del frame se è occupato, attende a tempo randomico e riprova.
- 3) Mentre la trasmissione avviene, l'adapter è in ascolto per ricevere notifica di un'eventuale collisione. Se non ci sono collisioni la trasmissione è andata a buon fine.
- 4) Se viene percepita una collisione, la trasmissione è interrotta e viene mandato un segnale di collision enforcement (detto **jamming**, verso tutte le stazioni).
- 5) Una volta terminata la trasmissione del collision enforcement, l'adapter entra in una fase di **exponential backoff** (metodo per tentare di ridurre le collisioni future): trovandosi all' m -esimo tentativo attende un tempo R compreso fra 0 e 2^{m-1} per ritrasmettere.
- 6) L'adapter attende $R*512$ tempi di slot e ritorna al passo 2.

Precisando la terminologia, un **datagram** è tale solamente se è passato da un livello IP superiore appartenente allo stack TCP/IP.

Il **jam** è un pattern di bit prefissato che avverte tutti quanti che c'è stata una collisione.

L'exponential backoff

Può risultare strana la scelta di adoperare un exponential backoff, ma noi stiamo cercando un protocollo che funzioni bene con un qualsiasi numero di macchine connesse. Quindi ci serve qualcosa di dinamico e non di statico!

Consideriamo ad esempio di avere un intervallo statico di R (0, 100). Avendo 100 macchine, allora per poter trasmettere ognuna delle 100 macchine dovrebbe estrarre un R differente, evento che ha probabilità di 0,01 per ogni macchina (parecchio bassa quindi!).

Se invece avessimo 2 macchine, allora potremmo estrarre 90 e 100 come R , e quindi attenderemmo ben 90 tempi di slot (512×90) assolutamente inutili.

Da qui si deduce che un intervallo statico non può funzionare e quindi l'expontential backoff è un'ottima soluzione. Dopo la **decima collisione** si ha un intervallo **{0, 1, 2, ..., 1023}** e si procede con questo intervallo per altri sei tentativi. Quindi, al **sedicesimo tentativo**, si suppone che ci siano problemi nel livello fisico e viene restituito un **errore di trasmissione**.

3.5) Analisi delle prestazioni

Possiamo ora ad un'analisi prestazionale.

I ritardi dovuti all'expontential back-off sono ovviamente da considerarsi negativi.

Su Ethernet a 10Mbps il tempo di contesa risulta essere $50\mu\text{s}$ (2 volte τ), quindi con R peggiore (1023) il ritardo inserito nel prossimo tentativo di ritrasmissione è di 50msec.

Formule per valutare l'efficienza

Come possiamo valutare l'efficienza del canale?

Se chiamiamo **P** il tempo che mediamente un frame ci impiega per arrivare a destinazione (quindi in cui occupa il canale), e **A** la probabilità che un canale acquisca il controllo del canale durante una contesa allora abbiamo che: $1/A$ è il numero medio di intervalli ogni contesa.

$$\frac{P}{P + 2\tau/A}$$

Con l'aumento di τ (la distanza massima fra due punti della rete) l'efficienza diminuisce e lo stesso vale se P è particolarmente piccolo.

Se ne deduce che **Ethernet funziona molto bene con frame grandi e distanze piccole**.

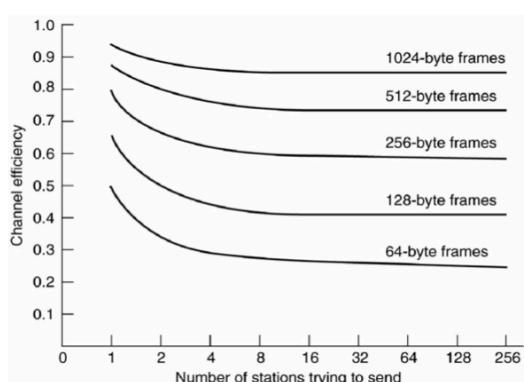
Possiamo anche valutare l'efficienza basandoci sul concetto di banda.

Definiamo con **F** la dimensione del frame in bit e con **B** la banda in bps, allora $P = F/B$.

Definiamo poi con **L** la massima distanza fra due nodi e con **c** la velocità di propagazione del segnale.

$$\frac{1}{1 + 2BL/cF}$$

Ancora una volta, aumentare la dimensione della rete diminuisce l'efficienza così come l'aumento della banda.



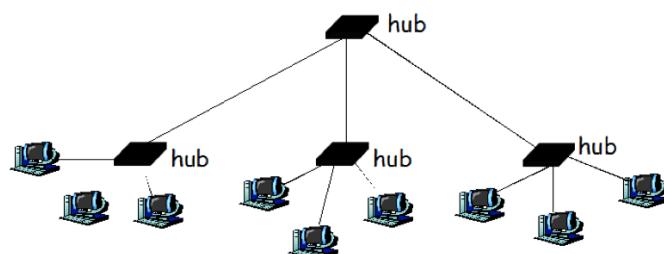
A fianco abbiamo un sorprendente riassunto delle efficienze, basate sulle dimensione dei frame.

3.6) Interconnessioni

Abbiamo un chiaro problema: le reti LAN che ci servono devono avere dimensioni grandi ma è controproducente rispetto alle prestazioni. Abbiamo quindi bisogno di dispositivi di interconnessione. Ve ne sono principalmente due, **hub e switch**.

Gli hub

L'hub non è null'altro che una prolungazione del cavo. Ha capacità di amplificazione ma **non riduce il dominio di collisione**.



Nell'esempio sopra esposto, i tre hub non confinano assolutamente il dominio di collisione: abbiamo sempre le nove macchine che possono collidere fra loro. Gli hub portano solamente **comodità di costruzione**.

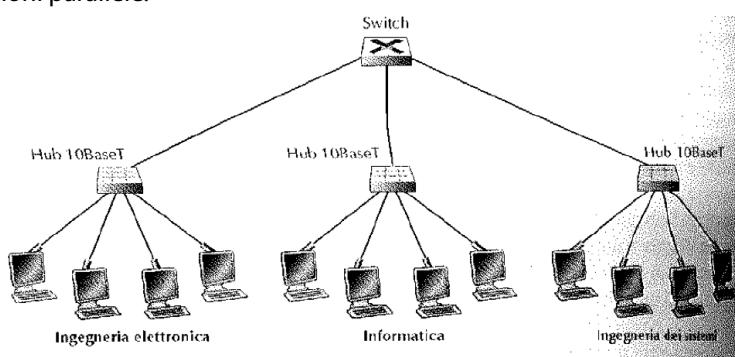
Gli switch

Ben diverso è invece il risultato ottenuto applicando degli switch.

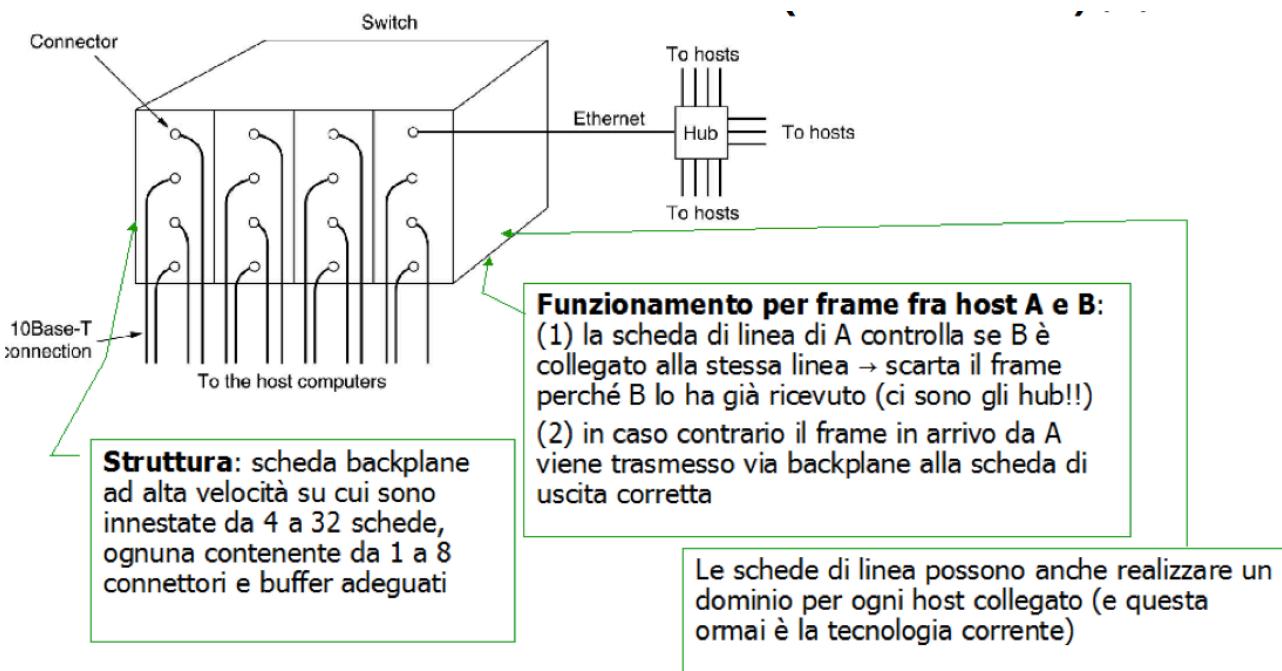
Gli switch sono sistemi dotati di capacità di calcolo e buffer. Hanno più porte e funzionano grazie a tre diverse capacità:

- sono in grado di **apprendere i singoli indirizzi MAC collegati ad una porta** leggendo il frame.
- possono **inviare un solo frame** ad una certa porta (ovviamente quella cui è collegata la macchina che specifica il MAC che è contenuto come destinatario nel frame).
- possono **bufferizzare** i frame limitando di per sé le collisioni.

Le collisioni sono ridotte drasticamente: accadono solamente più nei singoli domini di collisione e non fra tutte le macchine collegate allo switch. Inoltre, se le connessioni sfruttano porte diverse (avvengono fra macchine diverse) è possibile gestire connessioni parallele.



In questa immagine vediamo tre diversi domini di collisione. Abbiamo quindi limitato le collisioni, il che è ottimo per le prestazioni. Ma come funziona questo sistema?



Come vediamo lo switch è suddiviso in **schede**. Macchine dello stesso dominio di collisione sono connesse alla stessa scheda. Questa tecnologia permette anche di associare reti diverse su schede diverse e farle comunicare fra loro grazie ad uno switch.

Al suo interno, lo switch tiene in memoria una **tabella** che correla le porte ad ogni MAC con relativo timestamp. Quando il timestamp scade, lo switch rieffettua un controllo delle porte e (eventualmente) cambia gli indirizzi MAC associati alle porte.

Lo switch è in grado di effettuare:

- **filtraggio** ovvero il messaggio viene scartato poiché il destinatario e il mittente si trovano entrambi sulla stessa scheda, ciò significa che il messaggio è già stato ricevuto dal mittente all'interno della rete (che funziona col solito metodo di broadcast).
- **inoltro** il destinatario è mappato su una scheda differente rispetto al mittente, quindi il frame viene "girato" sulla scheda che contiene il MAC del destinatario. Questa operazione viene effettuata solitamente tramite store and forward (cioè il frame è salvato su buffer e in seguito inviato) ma talvolta è anche applicabile la tecnica del cut-through che procede all'inoltro appena possibile, senza salvare prima il frame.

Ethernet è intrinsecamente half-duplex, ma con l'introduzione degli switch la connessione diventa **punto-punto** tra due nodi e avendo due canali separati può anche diventare full-duplex. In tal caso non è più necessario rilevare le collisioni (ma lo switch deve essere bufferizzato, per gestire al meglio i messaggi tra i nodi).

Inutile dire come i messaggi di **broadcast** siano dannosi per lo switch: moltissimi inoltri da gestire, cosa che può affaticare notevolmente il dispositivo.

3.7) Le versioni successive

Vediamo ora una rapida carrellata delle versioni successive, evidenziandone i punti di differenza rispetto all'Ethernet classico.

Fast Ethernet (IEEE 802.3u)

Sviluppata nel 1992, è una versione di Ethernet che opera a 100Mbit/sec.

Le specifiche a livello MAC sono invariate, la topologia adottata è esclusivamente a stella.

Lo slot time (tempo massimo tra i due nodi più distanti) è ridotto da 100 a 10 nsec, con distanza massima fra due nodi nel dominio di collisione a 200 metri (10 volte di meno rispetto a Ethernet classico).

Name	Cable	Max. segment	Advantages
100Base-T4	Twisted pair	100 m	Uses category 3 UTP
100Base-TX	Twisted pair	100 m	Full duplex at 100 Mbps (Cat 5 UTP)
100Base-FX	Fiber optics	2000 m	Full duplex at 100 Mbps; long runs

Viene **abbandonata la codifica Manchester** poiché sarebbero necessari dispositivi troppo sofisticati in grado di produrre ben 200Mbaud (nella codifica un bit corrisponde a due baud).

Più nel dettaglio:

Nella versione 100Base-T4:

Sono adoperati quattro link, (due doppini). In fase di trasmissione si usano i link 2, 3 e 4 per trasmettere mentre sul link 1 si ascolta. In fase di ricezione invece i link 1, 2, 4 sono adoperati per ricevere mentre il link 2 rimane utilizzato. La codifica Manchester è abbandonata a favore della codifica **8B6T**, cioè si codificano otto simboli binari in sei simboli ternari. I simboli ternari vengono poi inviati sui link tramite tecnica round robin.

Nella versione 100Base-TX:

Al posto della tecnica Manchester viene adottata la tecnica **4B/5B** che scomponete la sequenza di bit da trasmettere in gruppi da quattro bit, ciascuno dei quali è codificato mediante un **code group** di cinque bit.

I code group sono costruiti in modo che non vi siano più di tre zeri di seguito nella sequenza e che vi siano almeno due uno.

Si noti che solamente 16 delle 32 code group sono adoperate per codificare dati, le altre sono usate per il controllo del livello di collegamento.

Nella versione 100Base-FX:

Dato che adoperiamo fibra ottica, non è possibile utilizzare CSMA/CD quindi ogni calcolatore costituisce un dominio di collisione a se stante (ciò implica nessun hub ma solo switch).

Gigabit Ethernet (IEEE 802.3ab)

Sviluppata nel 1999, è una versione di Ethernet che opera ad 1 Gbit/sec.

Tutte le connessioni di tipo Gigabit sono punto-punto.

Ne esistono versioni half-duplex (se si usano hub, è necessario però introdurre CSMA/CD, ma un frame è trasmesso 100 volte più velocemente rispetto all'Ethernet classica, con conseguenza il fatto che la distanza massima sia di soli 25 metri... pochissimo!) e full-duplex (si adoperano switch, quindi la distanza non è limitativa poiché non è implementato il CSMA/CD).

Sono state proposte delle soluzioni per risolvere il problema delle distanze massime che sono davvero limitanti.

Soluzione 1: carrier extension

Si fa uso massiccio del padding all'interno dei frame, allungandoli fino a 512 byte (distanza massima: 200 metri). Questa tecnica è poco efficiente per frame piccoli, infatti, con un frame di 46 byte dati (Ethernet classica) avremmo una percentuale di efficienza di soli 8,9% (46/512).

Soluzione 2: frame bursting

Si attende che il mittente abbia generato frame a sufficienza per raggiungere i 512 byte. Sono evidenti i problemi di latenza con piccoli frame.

Entrambe le soluzioni non sono un granché, quindi è chiaro che nessuno adoperi reti Gigabit con hub (che hanno il problema della distanza massima), ma solamente con gli switch.

Anche qui non viene più adottata la codifica Manchester (ci vorrebbero segnali a 2Gbaud!) ma si adotta una tecnica **8B/10B** che funziona in modo simile alla **4B/5B** con la differenza che non si vogliono più di 5 cinque 1, più di cinque 0, ma abbiano lo stesso numero di 0 e di 1.

Un'altra problematica è costituita dalla memorizzazione dei frame: 1Gbit è davvero una grande velocità, in un solo secondo possono essere ricevuti fino a 1953 frame con chiari problemi di overflow dei buffer. Per questo è stata implementata una sorta di **controllo di flusso** (che invece non è presente in tutte le altre versioni di Ethernet), ovvero, nel campo type del frame si introduce il codice 0x8808 che sta a significare "non ho più spazio, smetti di inviare".

3.8) Il perché della vittoria di Ethernet

Il protocollo Ethernet ha vinto sugli altri perché è semplice e flessibile.

Economicamente parlando richiede pochi costi, ed è pienamente in grado di supportare TCP/IP.

L'evoluzione inoltre è stata intelligente, sviluppata in maniera retrocompatibile e con incremento graduale delle velocità.

4) LAN Wireless (IEEE 802.11)

Le tecnologie wireless non fanno uso di cablaggio e per questo sono molto vantaggiose nei casi di:

- Edifici dove non è possibile effettuare cablaggio (edifici storici, ad esempio)
- Connettere più edifici che sono divisi da suolo pubblico
- Permettere l'accesso "nomade" degli utenti
- Avere reti temporanee (**ad hoc**)

Chiaramente non avere i cavi implica anche dei lati negativi, ovvero una grande interferenza oltre che problemi dal lato della sicurezza e del consumo.

4.1) Introduzione

La componente fondamentale delle reti 802.11 è l'**Access Point** (AP): ogni client è associato ad un AP connesso a sua volta alla rete.

Quando più AP sono connessi fra loro (generalmente in rete cablata), allora la rete prende il nome di **sistema di distribuzione** mentre invece se più terminali sono connessi fra loro senza far uso di un AP abbiamo una **rete ad hoc**.

I protocolli

Vi sono molteplici protocolli per questo livello, **802.11a** (da 5.1 a 5.8 Ghz, fino a 54 Mbps), **802.11b** (da 2.4 a 2.485 Ghz, fino a 11 Mbps), **802.11c** (da 2.4 a 2.485 Ghz, fino a 54 Mbps).

Tutti e tre presentano caratteristiche comuni:

- Sfruttano il protocollo **CSMA/CA**
 - Stessa struttura del frame
 - È possibile ridurre la frequenza per raggiungere distanze minori (**rate adaption**)
 - Possono funzionare sia in modalità ad hoc che in modalità infrastruttura
- Ci sono però delle differenze a livello fisico.

Ne esiste poi una versione ulteriore, **802.11n** (> 100 Mbps) detta MIMO, cioè multiple input/multiple output, che sfrutta più antenne di ricezione e di invio simultaneamente.

L'architettura

Il componente di base è il **set di servizio base** (BBS) che contiene una o più stazioni, insieme all'AP.

Le stazioni gli AP hanno indirizzi MAC a 6 byte che sono cablati nelle schede di rete.

Prima di procedere all'invio di pacchetti, le stazioni necessitano di essere associate ad un certo AP.

Ogni AP dispone di un **SSID** (*service set identifier*). In pratica tramite i SSID si identificano le reti, quindi quando su un sistema operativo selezioniamo "visualizza reti" visualizziamo gli SSID degli AP attualmente raggiungibili.

I canali

L'amministratore deve, poi, assegnare il numero del canale su cui la rete dovrà comunicare. La banda in cui opera 802.11 va dai 2.400 Mhz ai 2.485 Mhz. In questi 85 Mhz sono definiti **undici canali** che "frazionano" la banda in 11 parti parzialmente sovrapposta. Per non essere sovrapposti, due canali devono avere quattro canali di differenza (pertanto 1, 6, 11 costituiscono l'unica terna priva di sovrapposizione).

La giungla Wi-Fi

Se in un certo luogo ci sono più BBS appartenenti a sottoreti differenti, esse dovrebbero viaggiare su canali differenti. Una volta individuato l'AP di interesse, è necessario effettuare una **associazione** cioè creare un cavo virtuale tra la nostra macchina ed il particolare AP.

L'associazione (con scansione)

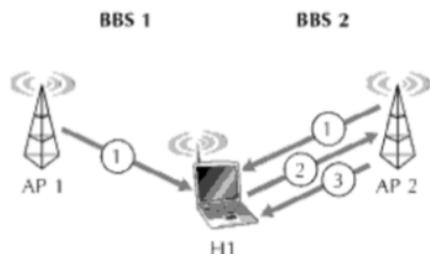
Lo standard 802.11 prevede che gli AP inviano periodicamente dei **frame** detti **beacon** contenenti il SSID e l'indirizzo MAC dell'AP.

La BBS scannerizza gli 11 canali alla ricerca di AP disponibili, legge i frame di beacon e seleziona l'AP secondo un algoritmo (che dipende dal progettatore della BBS, cioè del sistema operativo, ma tendenzialmente si seleziona quella con segnale più forte).

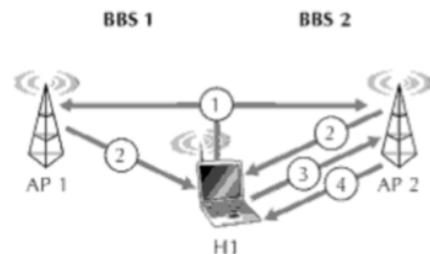
Vi sono due tipologie di scan che si possono effettuare:

Scansione passiva:

- 1 - Il frame di beacon viene inviato dall'AP
- 2 - H1 risponde ad AP2 con un frame di richiesta di associazione
- 3 - AP2 risponde con un frame di risposta di associazione

*Scansione attiva:*

- 1 - H1 manda frame in broadcast di richiesta a tutti gli AP
- 2 - Tutti gli AP rispondono ad H1
- 3 - H1 risponde ad AP2 con un frame di richiesta di associazione
- 4 - AP2 risponde con un frame di risposta di associazione

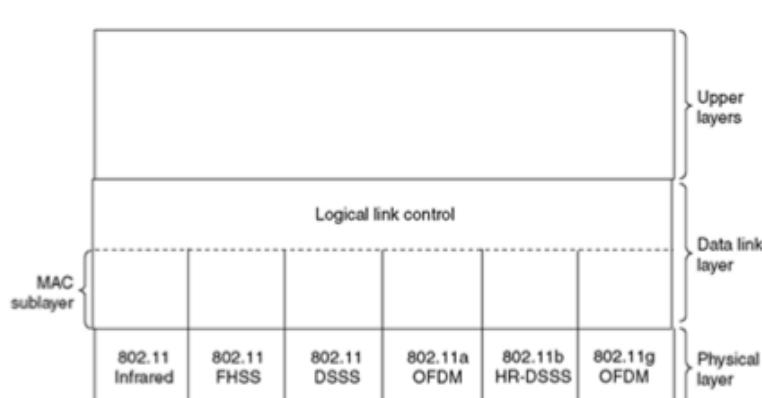


Chiaramente durante l'associazione può essere richiesto di **autenticarsi**. Questo può avvenire sulla base del MAC dell'host, oppure tramite l'inserimento di utente e password. In ogni caso gli AP sono in connessione con un server di autenticazione, col quale comunicano tramite appositi protocolli.

4.2) Architettura e protocolli

Rivediamo quanto introdotto prima in maniera più dettagliata.

Tutti i protocolli 802.11 hanno caratteristiche comuni, con una struttura simile.



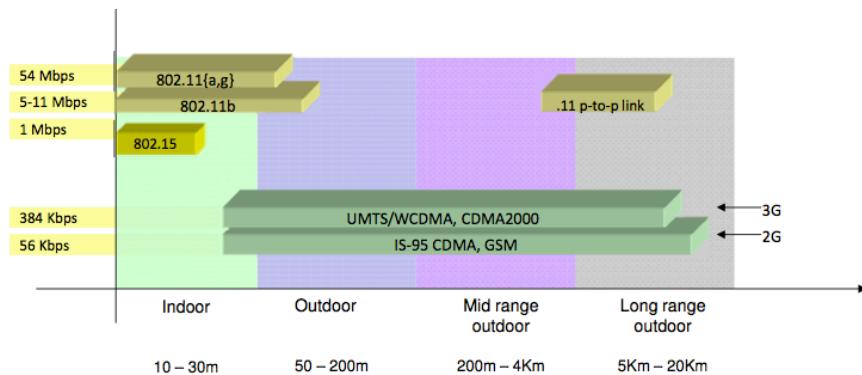
Il livello fisico è ereditato dallo standard ISO/OSI, mentre il datalink presenta due o più suddivisioni.

Mentre il MAC si occupa della gestione del canale, il livello sovrastante LLC nasconde le differenze fra tutti i protocolli 802 e li rende indistinguibili a livello di rete.

I protocolli che abbiamo sono:

- **802.11a** (da 5.1 a 5.8 Ghz, fino a 54 Mbps),
- **802.11b** (da 2.4 a 2.485 Ghz, fino a 11 Mbps),
- **802.11c** (da 2.4 a 2.485 Ghz, fino a 54 Mbps).
- **802.11n** (> 100Mbps) detta MIMO, cioè multiple input/multiple output, che sfrutta più antenne (fino a quattro) di ricezione e di invio simultaneamente.

Riassumendo:



Il CSMA/CA (Collision Avoidance)

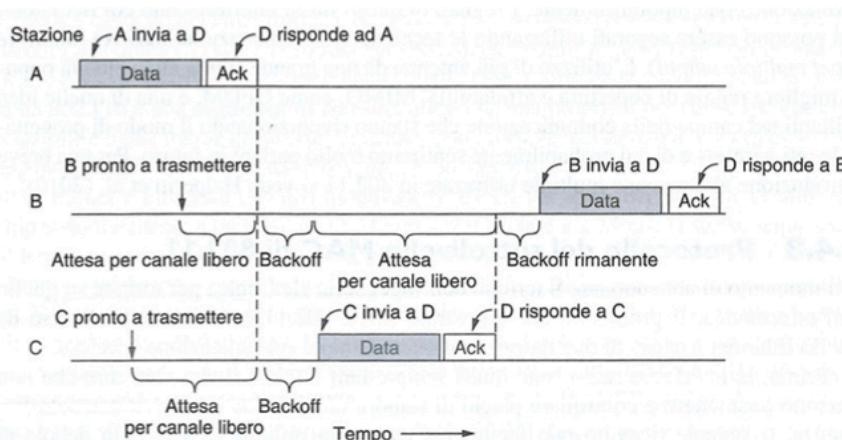
Anche qui si presenta il problema delle collisioni. Potremmo pensare applicare una tecnica di CSMA/CD ma questo non è attuabile poiché non è possibile ascoltare e inviare simultaneamente sul canale (manca la tecnologia).

Si passa quindi ad un'altra tecnica, **CSMA/CA** che mira a precedere ed evitare la collisione, invece che ascoltarla ed avvertire gli altri coinvolti.

Quando una stazione deve spedire un frame, inizia con un backoff casuale (numero di slot di backoff compreso fra 0 e 15).

Dopodiché la stazione attende finché il canale è libero (il canale viene considerato libero se lo è per un certo tempo) e conta (alla rovescia) gli slot inattivi (cioè i tempi t da 15 fino a 0) sospendendo il conteggio quando qualcuno invia un frame. Riprende poi a contare, finché arriva a zero. Quando una stazione arriva a zero, allora invia il suo frame. Appena una stazione riceve un frame, invia immediatamente un piccolo ACK la cui mancanza indica un errore, che sia stata una collisione o altro. In questo caso il mittente raddoppia il suo periodo di backoff e ricomincia con il count-down, continuando con backoff esponenziali come già è stato in Ethernet.

Esempio:



A invia a **D** un frame. A riceve l'ACK. A questo punto **B** e **D**, che era pronti a trasmettere, iniziano con il loro backoff, ma **C** ha un tempo minore quindi conclude il countdown e inizia a trasmettere. Una volta che **C** ha terminato, **B** riprende il suo conteggio e al termine del backoff invia.

Un nuovo modo di operare: DCF (distributed coordination function)

Emergono subito due differenze fra Ethernet e 802.11. Prima di tutto la partenza preventiva dei backoff fa evitare le collisioni. Questo tipo di approccio è molto conveniente quando le collisioni sono costose (si ritrasmette tutto il frame).

Secondo, gli ACK vengono utilizzati per dedurre le collisioni.

Questo modo di operare è detto **DFC** poiché ogni stazione agisce indipendentemente senza avere un controllo centrale.

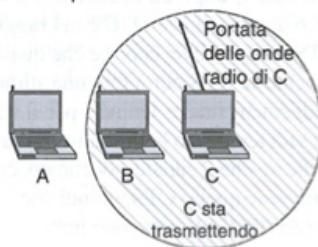
Ben diverso sarebbe l'approccio **PFC** (point coordination function) in cui l'AP governa le attività della sua cella, tecnica non utilizzata nella pratica poiché non è possibile governare il traffico proveniente da reti adiacenti.

La stazione nascosta e la stazione esposta

L'Ethernet è progettato in modo che tutte le stazioni siano in grado di sentirsi a vicenda.

Cosa ben diversa è invece il sistema Wi-Fi, che, per via dei segnali elettromagnetici, non garantisce che ogni stazione possa sentire tutte le stazioni vicine, ovvero, possono esserci stazioni fuori portata. Questo genera due problematiche differenti:

A vuole trasmettere dati a B
ma non è in grado di scoprire
se B è occupato

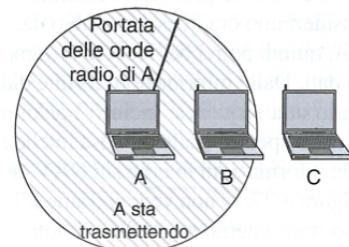
Stazione nascosta

Qui si ha, ad esempio, una trasmissione tra B e C.

Se A volesse trasmettere non sarebbe in grado di recepire la trasmissione tra C e B!

Stazione esposta

B vuole trasmettere dati a C
ma erroneamente pensa
che la trasmissione non avrà buon fine



In questo caso, invece, B vorrebbe trasmettere a C ma percepisce la trasmissione di A, che eventualmente sta inviando ad un'altra stazione D. Quindi non trasmette, sprecando così un'opportunità di trasmissione.

La rilevazione: fisica e virtuale

Per ovviare ai due problemi sopraesposti si adotta un tipo di rilevazione composita, ovvero, per rilevare il canale si tiene in considerazione una **rilevazione fisica** ed **una virtuale**.

La rilevazione fisica controlla la presenza di un segnale valido. Con la rilevazione virtuale, invece, ogni stazione tiene traccia dei **NAV** (network allocation vector).

Ogni frame possiede un campo NAV che indica quanto tempo sia necessario per completare la sequenza di cui il frame fa parte. Le stazioni che vedono il NAV, sapranno che il canale sarà certamente occupato per tutto il tempo indicato dal NAV. Il NAV include il tempo necessario alla spedizione dell'ACK (quindi tutte le stazioni ritarderanno l'invio durante il periodo di ACK, anche se non lo vedranno).

Hai fatto un mix del protocollo MACA e del protocollo MACAW. Sono due cose diverse. Quello che tu hai descritto è il funzionamento di MACAW (MACA for Wireless), ma il titolo del capitolo l'hai chiamato MACA.

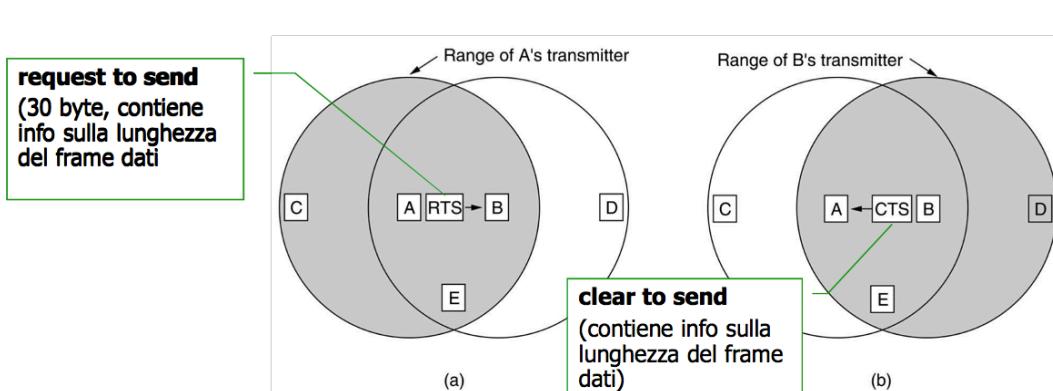
Il MACA originario era mirato alle connessioni wireless progettato appositamente per evitare il problema delle stazioni nascoste/esposte, ma il canale NAV non funzionava esattamente come l'hai descritto, poiché chi riceveva l'RTS ma non il CTS poteva comunque trasmettere, tanto il suo segnale non sarebbe arrivato a chi stava ricevendo in quel momento. (Risoluzione della stazione esposta).

Con il protocollo MACAW invece, anche chi riceve l'RTS sta zitto per tutto il tempo (non risolve quindi il problema della stazione nascosta, ma qui si assume che nelle reti moderne wireless questo tipo di problema sia molto limitato).

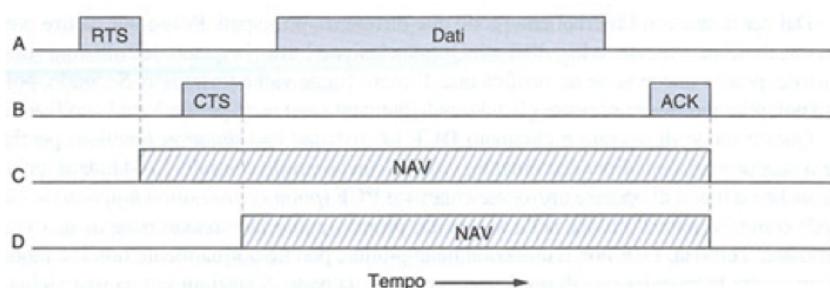
Nel MACAW inoltre ci sono altre ottimizzazioni che lo rendono migliore.

4.3) Il protocollo MACAW

Il MACAW (*Multiple Access With Collision Avoidance for wireless*) implementa un sistema molto simile allo stop-and-wait. Infatti, il trasmettitore sollecita il ricevitore a rispondere tramite un frame di controllo, in modo che tutte le altre stazioni nelle vicinanze capiscano che è imminente una trasmissione.



Vediamo come i frame di controllo RTS e CTS regolino il traffico e ci dicono che tra A e B stia intercorrendo una trasmissione.



Nella fattispecie, A dice che è pronto a inviare (**RTS** - tutti lo sentono, non trasmettono per evitare collisioni), dopodiché B con un **CTS** conferma che si può procedere alla trasmissione.

Gli altri attenderanno un tempo sufficiente affinché la trasmissione

avvenga: il tempo è contenuto all'interno del CTS.

Si noti che questa tecnica non garantisce al 100% l'assenza di collisione: pensiamo a due basi che inviano contemporaneamente un frame RTS!

Inoltre RTS/CTS:

- Non è efficace per frame piccoli (hanno la stessa dimensione dei frame di controllo...!)
- Rallentano molto: ogni frame RTS o CTS ricevuto blocca per un tempo fisso, sufficiente per trasmettere l'ACK.
- Funziona bene per la questione dei terminali nascosti, ma non sono poi così frequenti queste situazioni...

Meccanismi aggiuntivi a CSMA/CA

Nel protocollo MACAW vengono inoltre introdotte altre tecniche per risolvere il problema dell'**affidabilità**: le reti senza fili sono estremamente soggette a disturbi, quindi, il meccanismo degli ACK è troppo debole per poterci garantire un certo livello di sicurezza. Pertanto:

- Se vengono persi troppi frame, la stazione abbassa il rate di invio.
- Se invece la situazione è ottimale, viene aumentato il rate di invio.
- Si **divide in frammenti** un singolo frame per diminuire la probabilità che questo si scontri con un altro frame oppure che venga rovinato (meno bit, minore probabilità!).

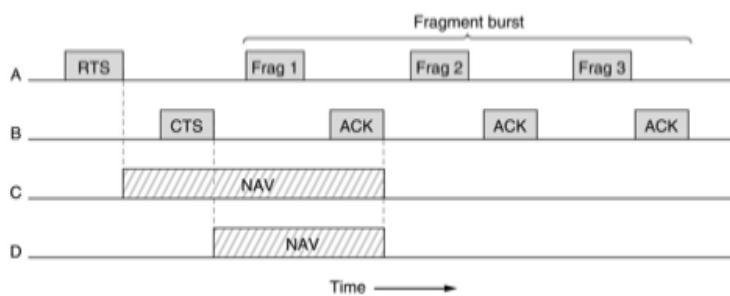
Un'altra problematica è quella rappresentata dal **consumo** di energia elettrica.

Sempre più spesso i terminali sono laptop dotati di batteria, quindi, il problema del consumo dei client non è così banale. Il client può, infatti, mettersi in "stop" e l'AP accumulerà il traffico per lui. Il client si risveglierà ad ogni beacon (inviai in broadcast) per verificare se "c'è qualcosa per lui".

La frammentazione

Come detto sopra, un frame può essere scomposto in più frammenti (la dimensione è a discrezione della cella).

- Ciascun frammento dispone di un proprio checksum,
- Ciascun frammento è numerato e gestito in stop-and-wait
- In seguito all'acquisizione del canale (solita, gestita RTS-CTS), viene inviato un insieme di frammenti
- Il canale risulta più utilizzato, in effetti vengono ridotte le collisioni e gli errori ma vengono anche trasmessi molti più ACK

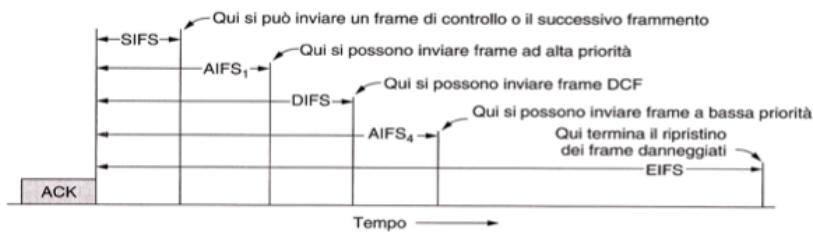


La qualità del servizio

Un ulteriore aspetto da considerarsi è la **qualità del servizio**.

Consideriamo ad esempio un traffico VoIP (Skype, per dire) ed un traffico peer-to-peer: nel caso di competizione tra i due traffici il primo ne risentirà certamente poiché il peer-to-peer a banda larga nonostante la banda del VoIP sia bassa. Ma questo significherebbe avere delle chiamate di scarsa qualità, quindi, è necessario trovare una soluzione. Banalmente, si può pensare di assegnare **priorità differenti a differenti frame** che rappresentano differenti traffici. IEEE 802.11 fornisce un meccanismo intelligente (formalizzato come IEEE 802.11e) per risolvere questa problematica.

Una volta che un frame è spedito, è richiesto un certo intervallo di tempo libero prima che una stazione possa spedire un altro frame. Il trucco sta nel definire differenti intervalli per differenti tipi di frame!



Viene detto **DIFS** (*DCF interframe spacing*) l'intervento di tempo fra frame di dati.

Definiamo poi differenti intervalli di tempo. Il più breve è l'**SIFS** che viene adoperato per esempio per inviare frammenti ravvicinati: tra un frammento e l'altro intercorrerà solo un tempo SIFS, questo significa che tutte le altre stazioni (che invece attendono DIFS) non potranno mai interrompere la sequenza di frammenti "a raffica".

I due livelli **AIFS** definiscono due livelli priorità: con AIFS₁ possiamo ad esempio inviare il traffico voce poiché più prioritario del normale DIFS, mentre con AIFS₄ trasmettiamo invece il traffico di background, meno rilevante, differibile finché non viene trasmesso il traffico regolare.

4.4) Il frame 802.11

Vediamo ora la struttura del frame.

802.11 definisce tre differenti classi di frame, uno dati, uno di controllo e uno di gestione.



Abbiamo una prima sezione **frame control** che è suddivisa in undici parti:

- **Version** (di default a 00, permette interoperabilità con le versioni successive di 802.11)
- **Type** (dati, controllo o gestione)
- **Subtype** (RTS o CTS)
- **To DS / From DS** indica se il frame sta arrivando o vendendo dall'AP
- **More Fragment (MF)** indica se seguiranno altri frammenti oppure no
- **Retry** indica se si tratta della ritrasmissione di un frame già spedito
- **Power (PWR)** indica se è attivato qualche sistema di power management
- **More** indica che il mittente ha altri dati per il destinatario
- **Protected frame (W)** indica se il frame è criptato
- **Order (O)** ci dice se il livello superiore si aspetta i frame in ordine oppure no

Abbiamo poi il campo **duration** che indica per quanto tempo il frame ed il suo ACK occuperanno il canale (servono al NAV).

Abbiamo poi gli indirizzi (rispettivamente **destinatario** e **sorgente**). Il terzo indirizzo è il reale punto finale della trasmissione (gli AP sono solamente posizioni di "staffetta" per trasmettere distante un messaggio).

Abbiamo poi il campo **sequence** (per eliminare i duplicati), il campo **dati** e il solito checksum.

4.5) Servizi

Vediamo ora i servizi che lo standard 802.11 gestisce per permettere ai client, agli AP e alla rete di cooperare insieme.

Servizio di associazione

Usato dalle stazioni mobili per connettersi agli AP.

Tipicamente quando il terminale entra nel raggio di azione dell'AP, riceve un frame di beacon contenente le informazioni necessarie per identificare l'AP.

Le funzionalità comprendono le modalità di sicurezza, la gestione energetica, il supporto per la qualità del servizio e altro ancora.

Servizio di riassociazione

Permettono a una stazione di cambiare il suo AP preferenziale all'interno della stessa rete LAN, talvolta garantendo che non venga perso nessun dato.

Servizio di autenticazione

Direttamente collegato con l'associazione è il servizio di autenticazione. Vi sono differenti schemi di sicurezza, dal WEP al WPA2. L'AP è sempre in collegamento con un server di autenticazione che possiede il database degli utenti e delle password per determinare se la stazione abbia il permesso di accesso alla rete.

Servizio di distribuzione

Vengono definiti i sistemi con cui i dati vengono inoltrati dalla stazione base.

Servizio di integrazione

Invio e ricezione di frame che non fanno parte della rete 802.11 (servizio Internet, ad esempio).

5) WiMAX - Wireless a banda larga (IEEE 802.16)

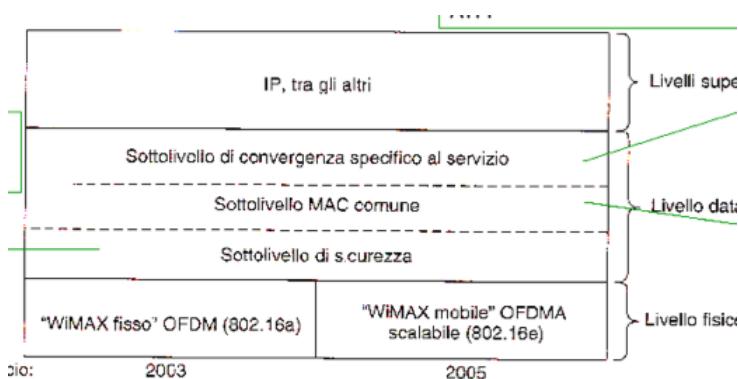
L'obiettivo delle reti wifi a banda larga è quello di eliminare definitivamente il problema dell'ultimo miglio collegando quindi più **edifici** in rete. Il protocollo è più complesso rispetto a 802.11.

5.1) Le differenze rispetto a 802.11

Innanzitutto 802.16 mira a unire edifici, quindi stazioni ferme. Inoltre i "terminali" sono ora gruppi di terminali (di ogni edificio, appunto).

Date le grandi distanze, il rumore è possibilmente più variabile o più forte e la banda risulta essere più grande poiché la quantità di dati da trasportare è maggiore (ma sforando nelle onde millimetriche, c'è il problema della propagazione nell'acqua, quindi è richiesta maggior efficienza nel controllo degli errori).

5.2) Lo stack



Vediamo come lo stack presenti elementi simili rispetto al classico 802.11 anche se figurano dei nuovi sottolivelli, come quello di sicurezza.

Questo livello è decisamente cruciale, poiché le reti pubbliche (tra edifici) soffrono molto di più il problema della privacy!

5.3) Il livello fisico

Viene adottato un sistema chiamato **OFDMA** (*orthogonal frequency division multiple access*) che permette a gruppi diversi di sottoportanti di essere assegnati a stazioni distinte, in modo che più di una stazione possa inviare o ricevere contemporaneamente.

Oltre che questo traffico simmetrico, le stazioni sono solite alternare invii e ricezioni. Così facendo applicano una tecnica **TDD** (*time division duplex*) mentre invece potrebbero anche adottare un metodo **FDD** (*frequency division duplex*) in cui la stessa stazione invia e riceve contemporaneamente (ma ovviamente su due sottoportanti differenti).

La stazione base fornisce ai terminali in rete mappe di trasmissione e mappe di ricezione che indicano quali sottoportanti sono libere per la trasmissione. Sono anche riportate le tempistiche di burst in cui assegnare i vari frame.

A differenza del livello 802.11 abbiamo il concetto di **connessione** a livello fisico.

La **ricezione** è abbastanza semplice: la stazione base controlla gli invii a livello fisico usati per mandare informazioni ai vari abbonati nella rete. Tutto quello che fa il sottolivello MAC è infilare i frame nella struttura.

L'**invio** è invece più complesso, poiché c'è il problema della concorrenza da gestire. Gli abbonati non sono coordinati nell'invio. Vengono definite quattro classi di servizio:

- constant bit rate
- real-time variable bit rate
- non-real-time variable bit rate
- best-effort

Il primo servizio (**constant bit rate**) è usato quando si vogliono inviare quantità di dati predefiniti ad intervalli temporali predefiniti (detti burst). Una volta creata la connessione, non sarà necessario richiedere slot temporali, ma verranno automaticamente forniti quando utili (nei momenti di burst, quindi). Viene usato per trasmettere voce non compromessa.

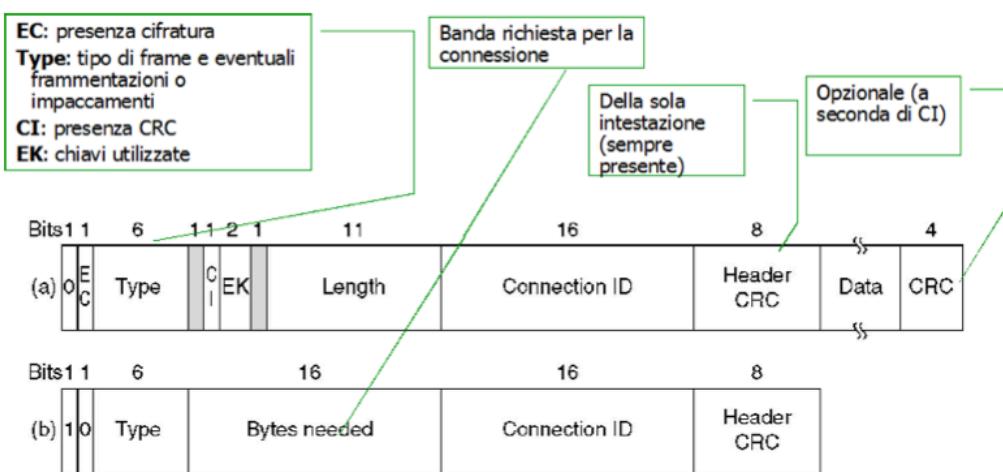
Il secondo servizio (**real-time variable bit rate**) è usato per i dati multimediali compressi e per trasmissione dati real-time (quindi la quantità di dati è altamente variabile ad ogni istante). La stazione base interroga periodicamente l'abbonato per sapere di quanta banda ha bisogno in quel momento.

Il terzo servizio (**non-real-time variable bit rate**) è usato per trasmissioni pesanti e non in tempo reale (ad esempio il download di grossi file). La base interroga a intervalli meno rigidi la base per sapere di quanta banda ha bisogno.

Il quarto servizio (**best-effort**) è usato per il resto del traffico. Non c'è alcuna interrogazione e l'abbonato deve competere per la banda insieme a tutti gli altri abbonati best-effort. Per limitare il numero di collisioni di adopera un algoritmo di backoff esponenziale.

5.4) Il frame 802.16

Il protocollo 802.16 ha molti tipi di frame. Ne vediamo uno generico (a), e poi uno particolare (b), che si adopera per richiedere banda.



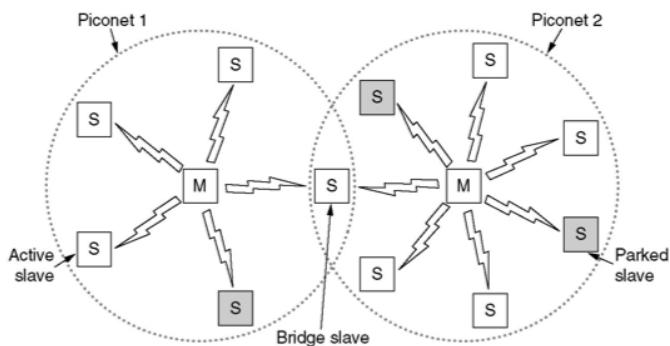
6) Il Bluethoot

Le reti bluethoot, dette **piconet**, sono nate principalmente con l'obiettivo di mettere in connessione dispositivi spazialmente vicini quali cellulari e computer con chip di poco costo.

6.1) Introduzione

I dispositivi bluethoot comunicano fra loro seguendo tre fasi succedanee:

- fase di ricerca
- fase di peering (e associazione)
- fase di comunicazione



La scelta di un chip economico ha portato alla scelta di un'architettura **master/slave**. Per ogni rete bluethoot (la cui portata massima è 10 metri) si possono avere al massimo un master (ovviamente) e sette slave attivi (anche se è possibile introdurre fino a 255 slave congelati). Come tutte le architetture master/slave, gli slave non possono comunicare tra loro ma fanno riferimento solamente al master.

Il bluethoot funziona sulle stesse frequenze del 802.11 (questo ovviamente crea spesso problemi di interferenze).

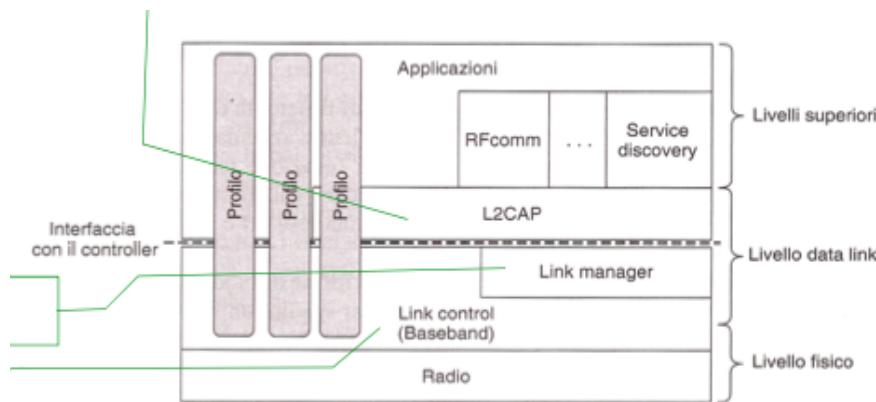
6.2) "Una" pila complicata

Giacché l'architettura bluethoot è stata pensata e implementata da produttori, non sono state affatto rispettate le metodologie di progettazione classiche di coloro i quali puntano ad una architettura chiara e ben modellata.

Abbiamo quindi il concetto di **profili**, ovvero, l'architettura specifica 25 diverse (qui riportate le prime 13) applicazioni. Ognuna di esse ha una pila differente! Si ha quindi un'enorme complessità per un risultato che non è neanche così eccezionale. Questo è una delle differenze fondamentale che c'è fra la 802.11 e l'architettura bluethoot.

Name	Description
Generic access	Procedures for link management
Service discovery	Protocol for discovering offered services
Serial port	Replacement for a serial port cable
Generic object exchange	Defines client-server relationship for object movement
LAN access	Protocol between a mobile computer and a fixed LAN
Dial-up networking	Allows a notebook computer to call via a mobile phone
Fax	Allows a mobile fax machine to talk to a mobile phone
Cordless telephony	Connects a handset and its local base station
Intercom	Digital walkie-talkie
Headset	Intended for hands-free voice communication
Object push	Provides a way to exchange simple objects
File transfer	Provides a more general file transfer facility
Synchronization	Permits a PDA to synchronize with another computer

6.2) La pila generica



Vediamo quindi una pila generica, considerando la problematica dei profili che sono trasversali rispetto alla pila, cioè, che talvolta re-implementano parti della pila in maniera personalizzata ed eventualmente ridondante.

Il livello **fisico** è preso dallo standard ISO/OSI, il **baseband** è l'analogo del livello MAC, l'**L2CAP** corrisponde all'LLC (si occupa di astrarre i protocolli superiori rispetto alla trasmissione inferiore).

Livello fisico

Abbiamo più in basso il **livello fisico**, del tutto simile a quello proposto dallo standard ISO/OSI.

Il range è quello dei 10 metri, con frequenze a 2.4GHz, suddivisa in 79 canali da 1MHz. Attenzione a non confondere il concetto di canale fisico con quello logico che viene stabilito in seguito alla connessione!

La scelta del canale viene effettuata evitando accuratamente quei canali in cui c'è già una trasmissione di altro tipo (per evitare interferenze).

Il bluetooth viaggia a 1.0 Mps (ma ha una resa bassa, del 13-15%).

Livello baseband

Il livello baseband si occupa principalmente di suddividere il flusso di bit trasformandolo in un flusso di frame.

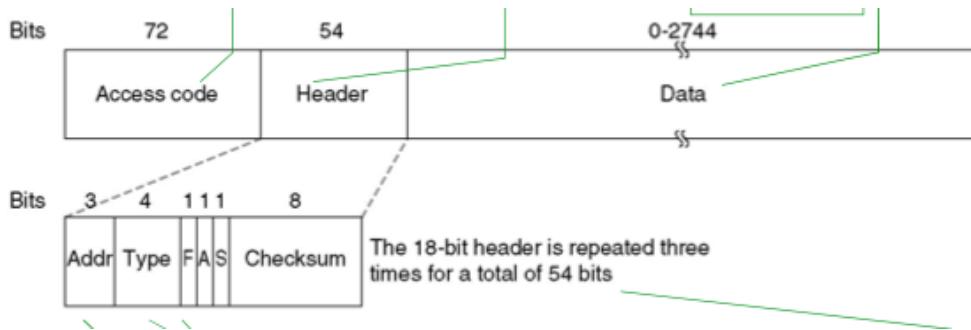
Inoltre supporta due tipi di link logici: **ACL** (Asynchronous Connectionless) che è un servizio best efforts (fa quel che riesce), che, insieme a un protocollo stop and wait può essere implementato per il trasferimento di files. L'altro tipo di link logico è **SCO** (Synchronous Connection Oriented) è invece un servizio a slot temporanei preimpostati, utilissimi per il trasferimento ad esempio tra una webcam e il computer (che quindi deve garantire una certa QoS). SCO non implementa alcun metodo per recuperare i bit persi, sarebbe inutile recuperare una parola che durante una chiamata è stata detta un minuto prima. Ben diverso è il caso del trasferimento file, dove la mancanza di anche solo un byte corromperebbe il file!

Livello L2CAP

Equivalenti all'LLC, prende i pacchetti dal livello superiore e li suddivide in frame. Inoltre permette ai protocolli superiori di non preoccuparsi di quelli inferiori e viceversa, astraendoli avvicendativamente (multiplexing/demultiplexing).

6.3) Il frame 802.15

Data la problematica dei profili, esistono molti frame differenti per l'architettura bluetoot. Vediamone uno generico.



L'**access code** contiene un indirizzo di riconoscimento per il master dell'architettura.

All'interno dell'header, abbiamo il campo **addr** che indica quale degli otto dispositivi coinvolti sia il destinatario.

Abbiamo poi il **type** che distingue fra ACL oppure SCO.

I tre bit successivi **F A S** indicano rispettivamente l'**overflow** (fornisce una piccola funzione di controllo di flusso, indica che il ricevente non può più ricevere a causa dei buffer pieni) l'**ack** e il numero di **sequenza**. Abbiamo poi il solito checksum.

Per questioni di sicurezza, l'header è ripetuto tre volte (occupando così 54 bit totali).

C'è poi la classica sezione **data**.

7) Il problema della commutazione

Il concetto di trasmissione è fondamento e costituzione del concetto di rete.

Ma come possiamo mettere in comunicazioni due reti LAN separate fra loro?

Supponiamo di avere una rete LAN e di volerla scindere in due reti diverse, pur mantenendole collegate. Perché optare per questa architettura?

- Gestioni differenziate per ogni LAN
- Sicurezza incrementata
- Dominio di collisione ridotto
- Capacità della rete aumentata (relativo al dominio di collisione)

Come fare?

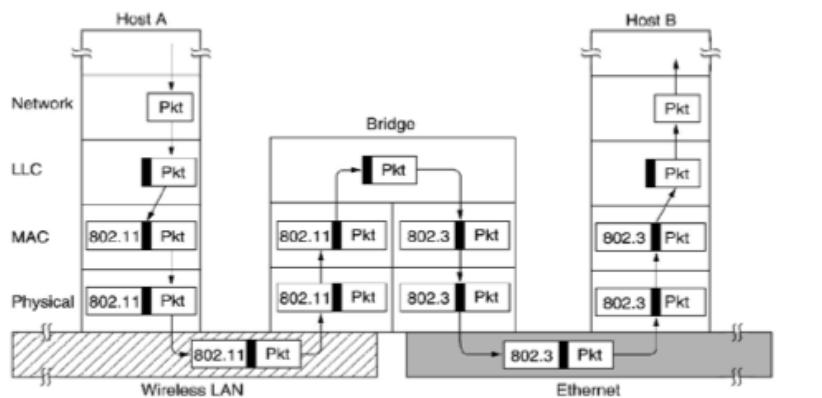
7.1) Il bridge

Il bridge è quello strumento che permette questo tipo di connessione.

Dal funzionamento simile allo switch, instrada i frame che devono essere inviati alle LAN che altrimenti sarebbero scollegate, e non instrada quelli che invece sono già arrivati poiché inviati da una LAN all'interno della LAN stessa.

Il bridge ha una fase di apprendimento iniziale (che può essere tramite **spanning tree** oppure tramite **backward learning**) grazie al quale mappa le LAN per poter lavorare al meglio.

In modalità promisqua, inoltre, permette di collegare reti che sfruttano tecnologie diverse (ethernet con 802.11 ad esempio).



L'apprendimento ed il problema dei cicli

Esaminiamo con attenzione il problema dell'apprendimento degli indirizzi.

Vogliamo un bridge che, una volta acquistato e montato nella rete "capisca da solo" quali macchine sono parte di una LAN e quali di un'altra, così da compiere correttamente le operazioni di inoltro/scarto.

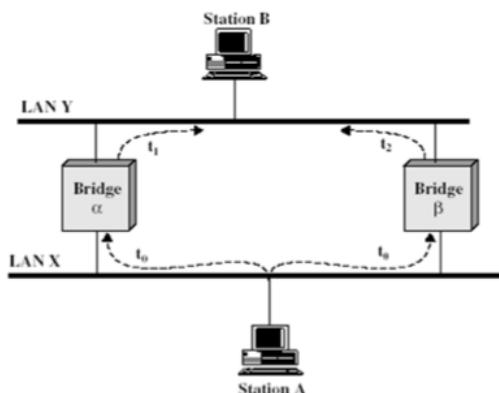
Come fare?

Prima di tutto si applica il **backward learning**, ovvero, il bridge memorizza i mittenti dei frame che gli "passano per le mani": memorizza l'indirizzo dai frame che gli arrivano e li inserisce in una tabella che correla ogni porta ai MAC raggiunti (esattamente come faceva lo switch). Viene inoltre inserito un timestamp, per avere un meccanismo di refresh dei dati su base temporale.

Quando non si conosce il destinatario (l'indirizzo MAC del destinatario non è ancora stato aggiunto alla tabella) allora il bridge ritrasmette il frame su tutte le porte tranne quella da cui ha inviato il mittente (ovviamente! Se il destinatario si trovasse su quella stessa LAN, allora avrebbe già ricevuto il messaggio via broadcast).

Questo tipo di approccio che sembrerebbe ottimo, ha un grosso difetto: si possono generare cicli di invii di frame "eterni" ed esponenziali che possono far collassare la rete in pochi secondi.

Capiamo meglio la problematica:

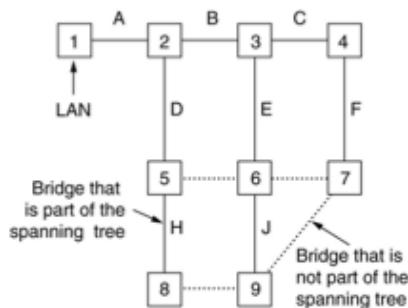


Come si vede dalla figura, un singolo frame t_0 viene inviato sia al bridge alfa che a quello beta.

Entrambi trasmetteranno il frame alla stazione B poiché la invieranno su tutte le porte tranne quella del mittente. Ora, i due frame t_1 e t_2 raggiungeranno nuovamente i bridge i quali, non conoscendo ancora B, reinvieranno a loro volta un t_3 ed un t_4 e così via!

Come risolvere la problematica?

Si implementa uno **spanning tree**, ovvero un albero disegnato **sopra alla topologia** della rete. Essendo un albero, per definizione non può contenere cicli, quindi alcune delle linee topologiche verranno ignorate. I bridge effettueranno ritrasmissione basandosi sullo spanning tree logico e non sulla rete reale, quindi.



Come si vede in figura, le linee tratteggiate rappresentano cicli dannosi e quindi non verranno effettuate ritrasmissioni su quelle linee (poiché non sono nello spanning tree).

7.2) I bridge remoti

Esistono poi bridge inventati apposta per le grandi distanze. In questo caso si parla di bridge remoti, che sono collegati con lunghe connessioni punto-punto via cavo. Ogni LAN deve avere un proprio bridge, quindi, per permettere queste connessioni.

7.3) Tutto questo commutare: bridge, switch, hub o router?

Abbiamo parlato ampiamente di questi quattro strumenti (meno del router, ma lo tratteremo meglio a livello network). Con l'avanzamento della tecnologia, switch e bridge sono sempre più simili (ma lo switch non offre opportunità di creare "switch-remoti", mentre i bridge sì).

Gli **hub** operano a **livello fisico**

Bridge, switch sono dispositivi che operano a livello **data link** mentre il **router** si occupa dell'instradamento a livello **network**!

Ovvero, una volta che abbiamo collegato le LAN, come facciamo a collegare tutti gli agglomerati di LAN in una rete unica? Se ne occupa il router.

Ne parleremo meglio in seguito.

7.4) Le Virtual LAN (VLAN)

Supponiamo ora di voler creare un livello astratto sopra alla rete: una organizzazione logica che non rispetta la **geografia** della rete.

Magari vogliamo gestire delle sottoreti per utenti di un certo livello, e quindi con **accesso ristretto** rispetto agli altri. Inoltre il problema dell'**efficienza** non è da dimenticare: una LAN normale non può differenziare il traffico in nessun modo.

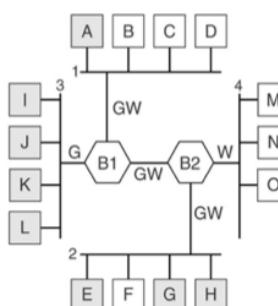
Un altro problema da considerarsi è il **broadcast storm**, ovvero l'invio di messaggi di broadcast con eventuali risposte in broadcast. Se le macchine interessate al messaggio sono in numero ridotto rispetto a quelle connesse alla LAN, allora per tutte le altre il broadcast continuo sarà solo fonte di disturbo!

Il funzionamento della VLAN

Come possiamo però "montare" questo sistema sulla rete?

Il concetto è quello di attribuire ad ogni LAN virtuale un colore differente. Gli utenti dello stesso colore si potranno scambiare frame fra loro mentre macchine con colore diverso non vedranno (reciprocamente) i loro frame.

Per poter attuare un sistema del genere, è necessario usare bridge che supportino lo smistamento dei frame e quindi **bridge VLAN-aware**.



Come potremmo implementarlo?

Etichettare le porte? ma allora tutti i calcolatori connessi a quella porta dovrebbero essere della stessa VLAN...

Etichettare i MAC?

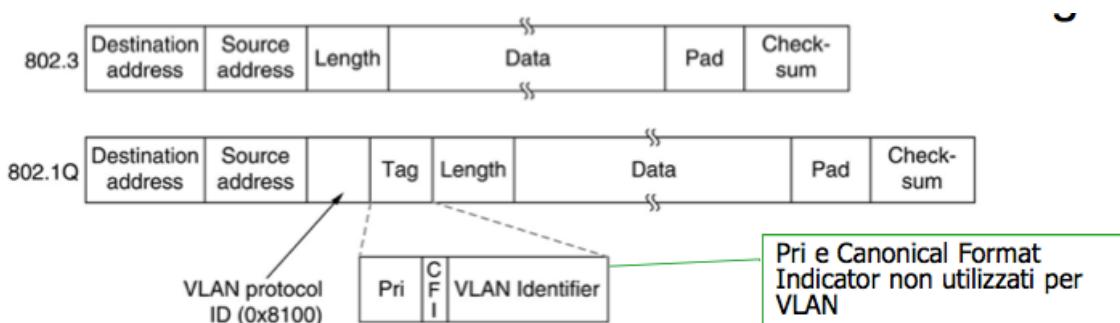
Etichettare gli indirizzi del livello network (IP)?

La soluzione più semplice sembrerebbe quella di "prendere" un pezzo del campo dati e inserirvi all'interno un flag indicante il colore.

Quest'ultima soluzione ha però un'effetto terribile: modifica il protocollo! Lo standard non può essere mai cambiato, significherebbe dover ridistribuire tutte le schede Ethernet in circolazione (sono milioni!).

Allora si è pensato di effettuare una continua operazione di "taglia e cuci". I bridge, quando arriva un frame, lo "tagliano" e vi inseriscono i flag di colorazione. I bridge successivi sapranno quindi dove instradare il frame.

Dopodiché, una volta giunti "a fine" tragitto, l'ultimo bridge toglierà le informazioni relative alla VLAN e instraderà correttamente il frame.



Questo metodo risulta totalmente trasparente per le macchine all'interno della rete! È perfetto!

Vengono inoltre inseriti due campi, non relativi alla VLAN, ma che "già che ci siamo" hanno pensato di inserire.

Abbiamo quindi questo funzionamento:

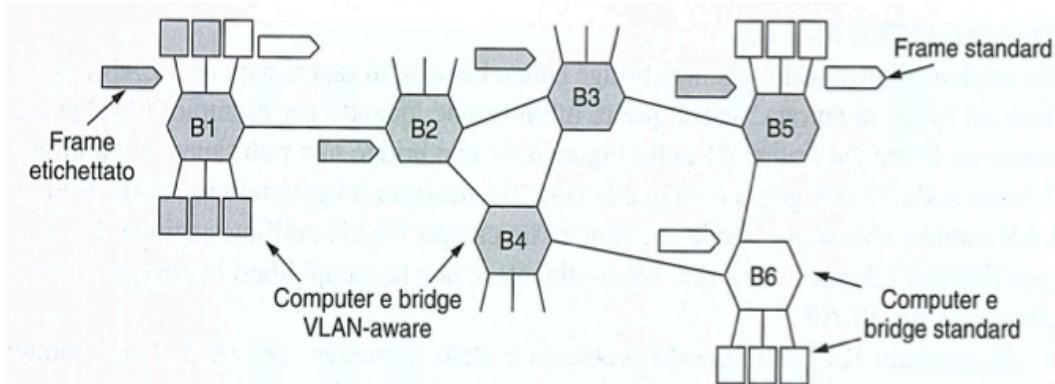


Figura 4.48 Una LAN che fa uso di bridge solo parzialmente VLAN-aware. I simboli evidenziati sono VLAN-aware, gli altri non lo sono.

I bridge che non sono in grado di capire le VLAN (più vecchi) verranno trattati alla stregua di macchine.

8) Esercizi finali

8.1) Problema su un canale di trasmissione: teoria delle code

Su un canale di trasmissione arrivano frame in maniera casuale.

La capacità del canale è **C = 100 Mbit/s**. Quando un frame trova il canale occupato, viene messo in coda.

Ogni frame è lungo **D = 10.000 bit**.

Supponendo che le frequenze di arrivo siano (1) 90 frame/sec (2) 900 frame/sec (3) 9.000 frame/sec, *calcolare il ritardo medio T* dato da $T = T_{\text{coda}} + T_{\text{trasmissione}}$.

Si nota qualcosa di strano?

Risposta

$$T = \frac{1}{\mu \cdot C - \lambda} \quad \mu = \frac{1}{D}$$

È necessario utilizzare la formula dove $\mu = \frac{1}{D}$ e lambda rappresenta i frame/sec (l'unità di misura è proprio frame per secondi! Anche se il canale e la dimensione del frame sono espressi in bit!

$$\mu = \frac{1}{10^4} = 10^{-4}$$

Perciò, nei tre casi (abbiamo sempre che: $\mu = \frac{1}{10^4} = 10^{-4}$) e $C = 100 \cdot 10^6 = 10^8$):

$$(1): T = \frac{1}{10^{-4} \cdot 10^8 - 90} = \frac{1}{10^4 - 90} = 0,1m\sec$$

$$(2): T = \frac{1}{10^{-4} \cdot 10^8 - 900} = \frac{1}{10^4 - 900} = 0,1m\sec$$

$$(3): T = \frac{1}{10^{-4} \cdot 10^8 - 9.000} = \frac{1}{10^4 - 9.000} = 1m\sec$$

Notiamo che nel caso (3) influiamo molto di più sul ritardo rispetto a quanto facciano il caso (1) e (2). Questo perché

$$\rho = \frac{\lambda}{\mu \cdot C}$$

il parametro $\rho = \frac{\lambda}{\mu \cdot C}$ è minore di uno nei primi due casi (0,009 e 0,09) ma è molto vicino a uno nell'ultimo (0,9). Ricordiamo che tale parametro ci dice quando la coda è satura (con 1 o maggiore la coda è satura ovvero non può essere smaltita).

8.2) Calcolo del numero di stazioni Aloha supportabili

Abbiamo una rete con N stazioni che supportano **56 kbit/s**. Viene inviato, mediamente, un frame ogni **100 secondi** e un frame ha lunghezza pari a **10.000 bit**.

Qual'è il valore massimo di N?

Risposta

Aloha [semplice] ha una resa del 18%. Quindi dei 56 kbit/s di capacità sono usabili solo 10,08 kbit/s ovvero 10.080 bit/s

Mediamente, ogni secondo vengono trasmessi 10 bit (poiché abbiamo 10.000 bit ogni 100 secondi).

$$N = \frac{\text{capacità_effettiva}}{\text{frequenza}} = \frac{10.800}{10} = 1.080$$

Pertanto, calcolando abbiamo che possiamo avere 1.080 stazioni insieme.

8.3) Calcolo del padding del frame Ethernet

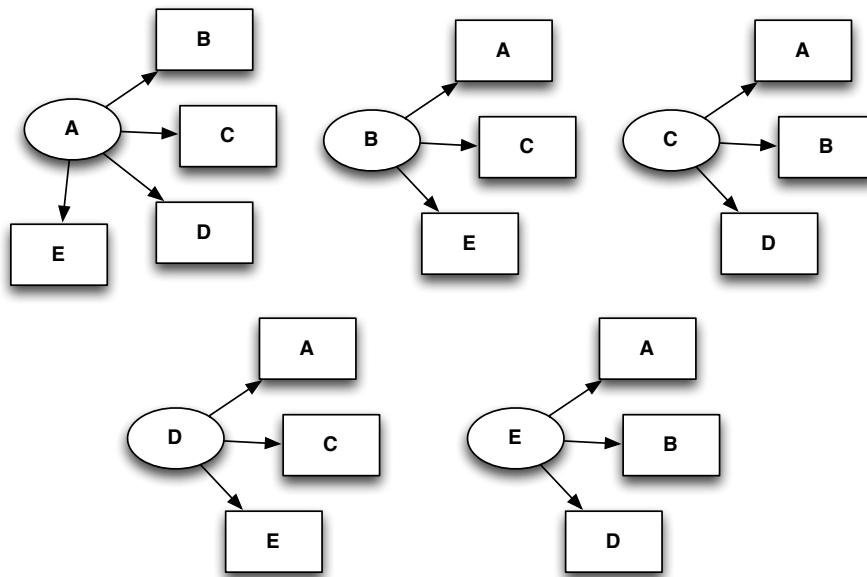
Abbiamo un frame ethernet con 60 byte di dati da trasmettere. *Quanto padding dobbiamo inserire affinché il frame venga trasmesso?*

Risposta

Sappiamo che la dimensione minima del frame Ethernet deve essere 512 bit ovvero 64 byte.

Dato le informazioni del frame aggiunte a livello Ethernet constano di 18 byte [non si conta il preambolo], abbiamo che $60 + 18 = 78$ byte, già maggiore di 64 byte. Quindi il padding da aggiungere è pari a zero.

8.4) Trasmissione simultanea fra reti 802.11



Abbiamo queste quattro stazioni A B C D E. Nel grafico sopra è riportato quali stazioni possono "vedere" le altre.

Supponendo le interazioni (-> significa invia un pacchetto):

- 1) A -> B
- 2) B -> A
- 3) B -> C

Quali altri invii sono legali (non creano collisione)?

Risposta:

Caso 1:

Dato che A -> B, chiunque si trovi nel range di A non potrà inviare. Dato che A possiede tutte le altre stazioni nel range, nessuna altra trasmissione sarà possibile.

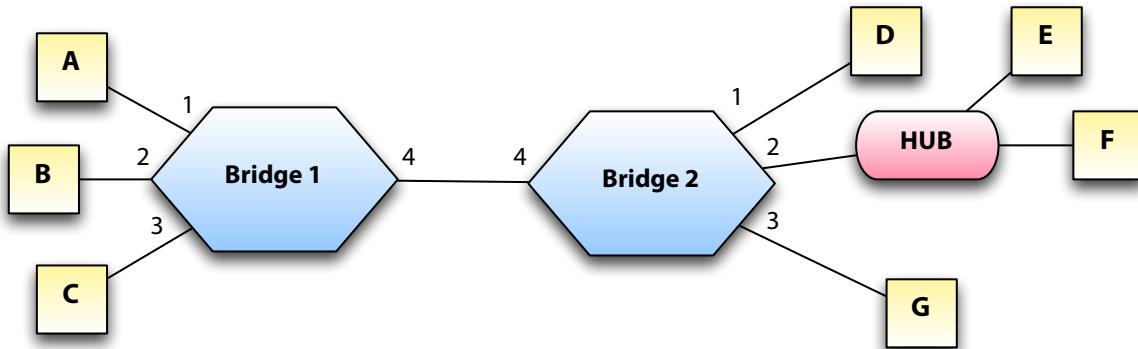
Caso 2: B -> A, vediamo che D non è nel range di B. Allora vediamo cosa è nel range di D, e abbiamo {A, C, E}.

Se D trasmettesse, raggiungerebbe anche A, disturbando l'arrivo del pacchetto che B sta inviando! Quindi, nessuna altra trasmissione sarà possibile. Viceversa, se A oppure C oppure E trasmettessero, raggiungerebbero allo stesso modo A, creando interferenza.

Caso 3: B -> C, vediamo che D non è nel range di B. Allora vediamo cosa è nel range di D, e abbiamo {A, C, E}.

D non può trasmettere poiché la sua trasmissione intaccherebbe C, che sta ricevendo. Però notiamo che, mentre A e C non posso trasmettere perché intaccherebbero C (C sta proprio ricevendo, addirittura!) E non ha C nel suo range, quindi può trasmettere. Pertanto **E -> D** è legale.

8.5) Acquisizione degli indirizzi MAC da parte di un bridge



Abbiamo questa rete. Data la seguente sequenza di invio di pacchetti, indicare come i due bridge acquisiscono gli indirizzi MAC delle stazioni. (-> rappresenta l'invio di un frame). Consideriamo i primi sei invii.

1) A → C

A invia un frame a **C** [mappa **A** su **1**], il **Bridge 1** non conosce **C** quindi procede ad un floating sulle porte **2**, **3** [**C** riceve il pacchetto] e **4** (tutte meno quella da cui è arrivato il frame).

Bridge 2 riceve quindi dalla porta **4** il frame inviato da **A** [mappa **A** su **4**] per **C**. Non conosce **C**, quindi effettua floating sulle porte **1**, **2** e **3**.

Bridge 1		Bridge 2	
Porta	MAC	Porta	MAC
1	A	4	A

2) E → F

E invia un frame ad **F** [mappa **E** su **2**] [**F** riceve il pacchetto, stessa LAN!], il **Bridge 2** non conosce **F** quindi procede ad un floating sulle porte **1**, **3** e **4**.

Bridge 1 riceve quindi dalla porta **4** il frame inviato da **E** [mappa **E** su **4**] per **F**. Non conosce **F**, quindi effettua floating sulle porte **1**, **2** e **3**.

Bridge 1		Bridge 2	
Porta	MAC	Porta	MAC
1	A	4	A
4	E	2	E

3) F → E

F invia un frame ad **E** [mappa **F** su **2**] [**E** riceve il pacchetto, stessa LAN!], il **Bridge 2** conosce **E** quindi dato che la porta di **F** è uguale a quella di **E**, scarta il pacchetto.

Bridge 1		Bridge 2	
Porta	MAC	Porta	MAC
1	A	4	A
4	E	2	E
		2	F

4) G -> E

G invia un frame ad **E** [mappa **G** su **3**], il **Bridge 2** conosce **E** quindi inoltra il pacchetto sulla porta **2** [**E** riceve il pacchetto].

Bridge 1		Bridge 2	
Porta	MAC	Porta	MAC
1	A	4	A
4	E	2	E
		2	F
		3	G

5) D -> A

D invia un frame ad **A** [mappa **D** su **1**], il **Bridge 2** conosce **A** quindi inoltra il pacchetto sulla porta **4**.

Bridge 1 riceve quindi dalla porta **4** il frame inviato da **D** [mappa **D** su **4**] per **A**. Conosce **A**, quindi inoltra il pacchetto sulla porta **1** [**A** riceve il pacchetto].

Bridge 1		Bridge 2	
Porta	MAC	Porta	MAC
1	A	4	A
4	E	2	E
4	D	2	F
		3	G
		1	D

6) B -> F

B invia un frame ad **F** [mappa **B** su **2**], il **Bridge 1** non conosce **F** quindi procede ad un flooding sulle porte **1, 3 e 4**.

Bridge 2 riceve quindi dalla porta **4** il frame inviato da **B** [mappa **B** su **4**] per **F**. Conosce **F**, quindi inoltra il pacchetto sulla porta **2** [**F** riceve il pacchetto].

Bridge 1		Bridge 2	
Porta	MAC	Porta	MAC
1	A	4	A
4	E	2	E
4	D	2	F
2	B	3	G
		1	D
		4	B

Reti di Elaboratori

Il livello di rete

Capitolo 5

Enrico Mensa,

Basato sulle lezioni del prof. [Daniele Manini](#)

Introduzione al livello di rete

1) *Tante reti sconnesse* 1

1.1) Una scelta fondamentale: applicazione contro sistema operativo

- Interconnessione a livello applicativo
- Interconnessione a livello network

2) *L'Internet* 2

2.1) Indirizzare una macchina

2.2) Calcolatori per inoltrare: i router

2.3) L'architettura di una Internet

- Conoscere la topologia?
- Gli elementi della rete
- Il punto di vista dell'utente

3) *Indirizzi Internet basati su classe* 4

3.1) Gli identificatori universali

3.2) Gli indirizzi IP

- Netid univoci

3.3) Il meccanismo di indirizzamento

3.4) Come dividere l'indirizzo: le classi

3.5) Indirizzi come specificatori di connessioni di rete

3.6) Il broadcast e gli indirizzi speciali

- Subnet e Supernet
- Il multicast

3.7) I difetti/conseguenze dell'indirizzamento IP

3.8) I vantaggi dell'indirizzamento IP

3.9) Notazione dotted quad

3.10) Indirizzo di loopback

3.11) Ottenere un netid: ICANN e ISP

4) *Traduzione da indirizzi di rete ad indirizzi fisici: ARP* 8

4.1) Uno sguardo al problema

4.2) Due tipi di indirizzi fisici

4.3) Risoluzione mediante mapping diretto

- Con indirizzi fisici configurabili
- Con indirizzi fisici legati all'hardware

4.4) Risoluzione mediante mapping dinamico

- Funzionamento di ARP
- Richieste ARP: quanto mi costate!
- La cache di risoluzione
- Raffinamenti di ARP
- ARP soft stale: il timeout della cache

4.5) Implementazione di ARP

- Determinare l'indirizzo fisico per il pacchetto IP uscente
- L'incapsulamento di ARP

4.6) Due macchine con lo stesso indirizzo IP

4.7) RARP: conoscere il proprio indirizzo IP

- Se il RARP non va: server primari e di backup
- DHCP e BOOTP: la vera soluzione

Il protocollo IP

1) *IP: filosofia connection-less* 13

1.1) Caratteristiche e scopi

2) *L'unità di trasferimento: il datagram* 13

2.1) La struttura del datagram

2.2) Dimensione del datagram: MTU e frammentazione

- Riassemblaggio dei frammenti
- Il campo FLAG
- Il campo Time to Live (TTL)
- I campi PROTOCOL & HEADER CHECKSUM
- Esempio di frammentazione

3) *L'instradamento IP* 17

3.1) Consegnna diretta e consegna indiretta

3.2) Table-driven IP routing

- Vantaggi e svantaggi
- Il default router
- Strade specifiche per gli host

L'algoritmo di routing
Esempi di routing
Gestione datagram entranti
Impostare le tabelle di routing

4) Gestione degli errori: ICMP 22

4.1) Introduzione a ICMP: un protocollo indipendente

4.2) Formato dei messaggi ICMP

L'uso di ICMP

4.3) Messaggio ICMP: Echo Request/Echo Reply

Formato

4.4) Messaggio ICMP: destinazione non raggiungibile

4.5) Messaggio ICMP: controllo della congestione e del flusso dei datagram

Cosa c'è sotto davvero

4.6) Messaggio ICMP: richieste di cambiamento di route

4.7) Messaggio ICMP: Time To Leave scaduto

4.8) Messaggio ICMP: richiesta di subnet mask

4.9) Messaggio ICMP: annuncio dei router

4.10) Messaggio ICMP: richiesta di router

5) Subnetting e indirizzi senza classe 28

5.1) Ciò che sappiamo fin ora

5.2) Proxy ARP

Esempio di comunicazione
Tutta questione di fiducia

5.3) Il subnetting

Reminder: l'indirizzamento a classi
Gli indirizzi di sottorete
Esempi di subnetting
La flessibilità nell'assegnamento dell'indirizzamento a sottoreti
Impostare le subnet mask e gli indirizzi non utilizzabili
Esempio (1) di subnetting
Routing in presenza di sottoreti
Esempio (1) di routing in presenza di sottoreti: pacchetto dall'esterno
Esempio (2) di routing in presenza di sottoreti: consegna diretta

Esempio (3) di routing in presenza di sottoreti: consegna indiretta

Esempi di configurazione non valida di mask

5.4) Il supernetting

Supernet: una sola rete per controllarle tutte

Supernet: oltre il concetto di classe

Primi esempi di supernet e lista di mask CIDR

Strutture dati e algoritmi per la ricerca di indirizzi senza classi

Evoluzione dell'addressing

Altri esempi di supernetting

Le migliori ottenute grazie al supernetting

Introduzione al livello di rete

1) Tante reti sconnesse

Abbiamo profusamente parlato di decine di reti diverse ma nessuna di esse ci permette di collegarle tutte quante in una rete sola.

Vogliamo collegare tutti i calcolatori del mondo, abbiamo bisogno di un sistema che ci permetta di farlo. Sfruttiamo il modello già usato, ad esempio, dalle poste ed otteniamo il risultato tanto anelato:

- Tutte le reti fisiche sono connesse fra loro (ricordiamo che hanno tecnologie diverse fra loro!) [grazie ai **router**]
- Tutti i calcolatori sono raggiungibili (**indirizzabili**)

Si richiede quindi l'uso di **una tecnologia comune** che permetta a questo sistema di funzionare.

Possiamo, come sempre, fornire un servizio di rete di due tipi: connection less o connection oriented (quindi con QoS garantita).

Tale tecnologia mira ad unire le reti fondate su tecnologie diverse richiedendo semplicemente un livello datalink che sia in grado di spedire/ricevere pacchetti (come il PPP che abbiamo visto).

Sono essenzialmente tre i punti di cui dobbiamo occuparci:

- Fornire uno schema di astrazione dal livello sottostante, fornendo così dei **servizi di comunicazione universali**.
- Dettagliare come questa astrazione possa **costruire i necessari livelli software** per effettuare la **comunicazione** e nascondere invece i meccanismi sottostanti propri delle reti interconnesse.
- Mostrare come le **applicazioni** possano sfruttare l'astrazione.

1.1) Una scelta fondamentale: applicazione contro sistema operativo

La prima scelta che si è dovuta effettuare è stata quella di definire chi si sarebbe dovuto accollare il ruolo di gestore della astrazione.

Solo due scelte sono parse plausibili: o il sistema si implementa all'interno dell'applicazione oppure si implementa all'interno del sistema operativo.

Si ricorda che le proprietà desiderate sono l'**universalità**, la **connettività end-to-end** e la **trasparenza**.

Dunque per ottenere quel risultato ci si è resi conto di dover:

- Definire i dati da passare (**pacchetti**)
- Definire le procedure di scambio di dati.
- Definire un meccanismo di identificazione della singola macchina all'interno della rete (**indirizzamento** e **naming**).

Interconnessione a livello applicativo

Iniziamo col varare l'idea di introdurre nell'applicazione la parte di codice in grado di risolvere il nostro problema.

I programmi applicativi devono conoscere le caratteristiche di tutte le reti a cui possono essere interconnesse ed è in grado di stabilire una connessione con un'applicazione paritaria al di là della rete che si sta utilizzando "sotto".

Sembrerebbe fare al caso nostro, ma riflettendo un attimo riusciamo subito a delineare parecchie problematiche:

- Ogni nuova tecnologia di rete che viene introdotta richiede un'aggiornamento di tutti gli applicativi.
- In ogni applicativo è presente il codice in grado di gestire la connessione: ciò implica il codice venga ripetuto moltissime volte (rispetto alla scala della rete mondiale)!
- Ancor più grave, gli utenti devono specificare l'intero percorso di rete da un'applicazione all'altra.

Interconnessione a livello network

Innanzitutto si è notato che è più intelligente far viaggiare piccole unità di dati (i **pacchetti**) universalmente riconosciute e stabilite.

Questo porta alcuni vantaggi:

- È possibile inviare un pacchetto direttamente sfruttando la tecnologia di rete pre-esistente.
- L'attività di trasmissione dati è separata dall'elaborazione dell'applicazione.
- Il sistema è più flessibile poiché è possibile sviluppare sistemi "generici" giacché il pacchetto è un'unità sempre fissa.
- La parte di codice a gestire la connessione è unica ed è introdotta nel sistema operativo: più manutenibile e fa sì che i programmi applicativi non debbano essere modificati.

Il concetto che si vuole sviluppare di cui abbiamo già parlato sopra è l'**Internetworking**. La comunicazione deve essere totalmente scorrelata dal servizio fisico che l'effettua, fornendo all'utente totale trasparenza rispetto alla tecnologia di rete.

Si vogliono quindi interconnettere **migliaia di computer** con **decine di tecnologie diverse**, permettendo la cooperazione delle suddette affinché ogni computer possa comunicare con ogni altro computer sul globo. Vogliamo creare una **rete Internet**, ovvero una rete di reti.

Vogliamo assolutamente che il servizio sia **nascosto all'utente** affinché il livello applicazione non debba prendersi carico della mansione di connettere i calcolatori ma soprattutto non debba conoscere la topologia della rete. Così facendo otterremo un sistema flessibile che, in caso di cambiamento, non impatterà gravemente su tutte le applicazioni.

2) L'Internet

Eccoci quindi a sviluppare l'Internet basandoci sull'interconnessione a livello network.

2.1) Indirizzare una macchina

Le macchine all'interno dell'Internet devono essere tutte identificabili univocamente, non importa a quale tecnologia di rete siano connesse o quale sia la struttura dell'Internet stessa.

Chiaramente l'indirizzo deve essere **univoco**.

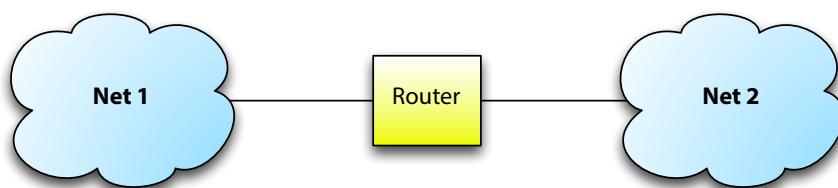
2.2) Calcolatori per inoltrare: i router

Come effettuare la connessione fra due sottoreti? È necessario un calcolatore che implementi il nostro sistema e cioè sia in grado di **ricevere** ed **inoltrare** pacchetti fra le due reti (alle quali deve essere ovviamente connesso).

Inoltrare è la parola chiave quando si parla di router!

2.3) L'architettura di una Internet

Vediamo la rappresentazione della più piccola Internet possibile:



Come si vede abbiamo due reti separate connesse fra loro grazie ad un router. Ovviamente il router deve disporre delle due interfacce di rete corrispondenti per comunicare con Net1 e Net2. Il router conosce inoltre tutte le macchine connesse alle due Net e si occupa di inoltrare i pacchetti tra una rete e l'altra. Il calcolo per verificare se un pacchetto debba essere inoltrato o meno (e se sì a chi) deve essere implementato in maniera molto snella poiché si tratta di un calcolo che verrà eseguito di continuo.

Le reti devono **collaborare con il router** affinché questo sistema funzioni. Ovvero, ogni rete dovrà passare al router i pacchetti (e quindi dovrà conoscerne l'indirizzo) e all'interno di essi vi saranno i dati necessari affinché R capisca cosa fare.

Conoscere la topologia?

E se avessimo una situazione del genere?



A questo punto come può Net1 comunicare con Net3? Router1 dovrebbe conoscere la topologia della rete, ovvero la lista di calcolatori appartenenti ad ogni rete. Ma questo è chiaramente impossibile poiché la memoria richiesta per memorizzare l'intera rete Internet sarebbe spropositata.

L'importantissima scelta implementativa che è stata fatta è quindi quella che **ogni router fa riferimento alle sottoreti e non alle macchine connesse nelle sottoreti**. Perciò i router "ragionano" solamente a sottoreti!

Si parla perciò di sottorete di destinazione anziché macchina di destinazione.

Così facendo abbiamo già ridotto notevolmente la quantità di informazioni che devono essere conosciute dai router, ma, come vedremo successivamente, i router dovranno disporre di ancora meno informazioni per inoltrare un pacchetto: ogni router dovrà solamente conoscere le reti alle quali è connesso.

Gli elementi della rete

Parlando di reti avremo:

- **macchine sorgenti** che sono coloro le quali generano i pacchetti e li spediscono. Se una macchina (sorgente) A deve comunicare con una macchina B sulla sua stessa rete, allora inoltrerà il pacchetto direttamente a B senza coinvolgere alcun router.
- **router intermedi** non hanno a disposizione la sottorete in cui si trova la macchina destinataria (quindi non sono connessi alla sottorete destinataria) pertanto inoltrano il pacchetto.
- **router finali** consegnano il pacchetto alla sottorete che contiene la macchina destinazione.

L'operazione di **instradamento** viene sempre effettuata basandosi sull'indirizzo della rete di destinazione e non sull'indirizzo del calcolatore.

Il punto di vista dell'utente

L'utente deve essere all'oscuro della topologia della rete, non ne ha bisogno: può immaginare di trovarsi in una enorme rete "unica" alla quale sono collegati tutti i calcolatori del mondo.

3) Indirizzi Internet basati su classe

Iniziamo a smembrare il problema e a trovare qualche soluzione pratica. Iniziamo dall'**indirizzamento**.

Come già detto, vogliamo un sistema che possa identificare ogni macchina in maniera univoca.

3.1) Gli identificatori universali

Come possiamo identificare una macchina? Distinguiamo alcuni concetti; secondo Shoch abbiamo:

- **nome**: cosa è un oggetto, è pronunciabile, ed è facile da ricordare.
- **indirizzo**: indica dove è un oggetto, è un numero, difficile da ricordare.
- **route**: indica come arrivare all'oggetto, non si vuole costringere i router a conoscerla poiché è pesante da memorizzare.

Nel **livello IP** abbiamo indirizzi sotto forma di numeri binari che sono facili da trasmettere grazie ai quali è possibile reperire la route. Gli indirizzi sono traducibili in nomi grazie al **naming** (ad opera dei DNS), che studieremo in seguito. Ma è importante ricordare che il livello IP ragiona **soltamente tramite indirizzi**.

3.2) Gli indirizzi IP

Non possiamo basarci sugli indirizzi fisici per via del fatto che non sono sempre univoci, non sono tutti chiari o definiti e con tecnologie diverse abbiamo indirizzi fisici diversi.

Scegliamo perciò di implementare un sistema logico arbitrario per definire gli indirizzi: un indirizzo è una **serie di 32 bit** univoco su tutta la rete. Tale indirizzo è detto anche **indirizzo IP**.

Una macchina può avere più indirizzi IP (potrebbe ad esempio interfacciarsi su tecnologie di rete diverse e quindi avere più indirizzi).

Abbiamo detto che l'indirizzo deve essere in grado di definire la route che il pacchetto deve seguire. Come fare?

La sotto rete deve essere identificata all'interno dell'indirizzo!

Suddividiamo quindi l'indirizzo IP in due parti: **netid** e **hostid**. Il netid definisce l'indirizzo della sottorete del destinatario mentre l'hostid definisce l'indirizzo della macchina destinataria **relativamente a quella specifica sottorete!** È un concetto molto importante.

Netid univoci

A questo punto è facile immaginare che **il netid debba essere univoco** mentre l'hostid debba essere univoco solamente nel caso in cui il netid sia già uguale. Come è possibile garantire tutto questo? Gli indirizzi netid debbono essere **amministrati** da una qualche autorità: gli **ISP (Internet service provider)** che affittano indirizzi IP ai loro clienti (come vedremo in seguito, nel tempo gli indirizzi IP hanno subito diverse modifiche quindi per ora accontentiamoci di sapere che sono gli ISP ad aiutare molto nella gestione degli indirizzi).

3.3) Il meccanismo di indirizzamento

La decisione della strada da seguire deve essere snella e veloce poiché viene effettuata molte volte. Si vuole quindi ridurre il più possibile l'informazione che deve essere esaminata.

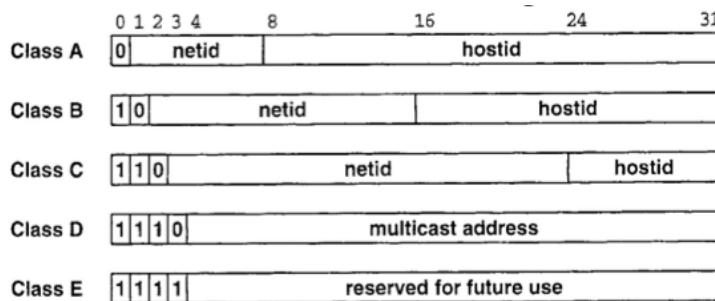
Il netid è leggibile in sole cinque istruzioni macchina e ci fornisce moltissime informazioni! Dividere in due l'indirizzo IP è quindi un'ottima idea.

3.4) Come dividere l'indirizzo: le classi

Non rimane che scegliere come suddividere i 32 bit. Con un grande spazio per le netid possiamo avere molte reti con pochi host, viceversa, poche reti con molti host.

Ma il nostro obiettivo è sempre stata la flessibilità... non possiamo imporre rigidamente il numero di bit riservati al netid (e quindi quelli riservati all'hostid).

Introduciamo perciò delle **classi**.



Sfrutteremo i bit iniziali di un indirizzo per capire di che classe sono (si veda il disegno sopra).

A seconda delle nostre esigenze potremo quindi avere indirizzi di classe A, B o C. Traducendo in numeri ciò che ogni classe ci offre abbiamo:

	Numero di reti	Numero di macchine
Classe A	126	16.777.214
Classe B	16.384	65.534
Classe C	2.097.152	254

Come vediamo i numeri non sono potenze di due (alcune configurazioni sono riservate, vedremo meglio in seguito).

Questo sistema offre una via di mezzo fra un sistema variabile e uno fisso! Abbiamo comunque il rischio di sprecare indirizzi. Si potrebbe pensare di aumentare le classi ma questo vorrebbe dire sprecare altri bit per codificarle...

3.5) Indirizzi come specificatori di connessioni di rete

Giacché ogni sottorete deve avere un netid univoco, allora i router (connessi a molteplici reti) dovranno avere molti indirizzi IP (uno per ogni sottorete) per far parte della sottorete stessa!

Quindi un indirizzo IP su un router specifica non una singola macchina, bensì una interfaccia che dà su una sottorete. Questo è uno svantaggio introdotto dal fatto di suddividere in due l'indirizzo IP.

3.6) Il broadcast e gli indirizzi speciali

Come detto sopra, alcune configurazioni di IP non sono utilizzabili poiché riservate ad altri scopi. Alcune di queste configurazioni sono adoperate per implementare il broadcast (si veda in seguito la dotted quad notation per capire come sono espressi gli indirizzi IP).

- Con **32 bit a 0** si indica "questo host su questa rete". È un indirizzo usato ad esempio da un calcolatore nell'istante in cui si collega alla rete e ha bisogno di conoscere il suo indirizzo IP.
- Gli indirizzi con **hostid tutto a 0** sono indirizzi che indicano "questo host" ma la rete è identificata dal netid che invece è valorizzato. Ad esempio avremo l'indirizzo 128.0.10.0 che identifica "questo host" nella rete 128.0.10.
- Per comunicare con tutte le macchine di una sottorete è sufficiente impostare l'**hostid tutto a 1**. Stiamo parlando del cosiddetto **broadcast diretto**. Qualora la sottorete fisica non supporti il broadcast, il pacchetto viene scartato. Ad esempio un indirizzo 128.0.10.255 fa sì che si effettui broadcast sulla sottorete 128.0.10.x ed è un indirizzo specificabile dall'esterno della sottorete 128.0.10.x.
- Con **32 bit a 1** otteniamo l'effetto broadcast sulla sottorete in cui il pacchetto appare, il **broadcast limitato**. Giacché non è possibile effettuare broadcast a livello di netid, un pacchetto con tutti 1 vivrà solo all'interno di una sottorete.

sottorete e verrà automaticamente scartato dai router. Pertanto il broadcast limitato si fa dall'interno di una sottorete per la sottorete stessa. Avremo quindi (ad esempio) un pacchetto 255.255.255.255.

Tendenzialmente con zero si indica "questo" e con uno si indica "tutti".

Subnet e Supernet

Ritorniamo sulla problematica della distribuzione di bit fra netid e hostid. Lo sviluppo delle reti ha generato un'enorme spreco di slot per host che sono rimasti inutilizzati.

Sono stati inventati sistemi di subnetting e supernetting che fanno sì che reti fisiche possano condividere lo stesso indirizzo IP di classe A, B o C risparmiando così slot per gli host. Ne parleremo in dettaglio in seguito.

Il multicast

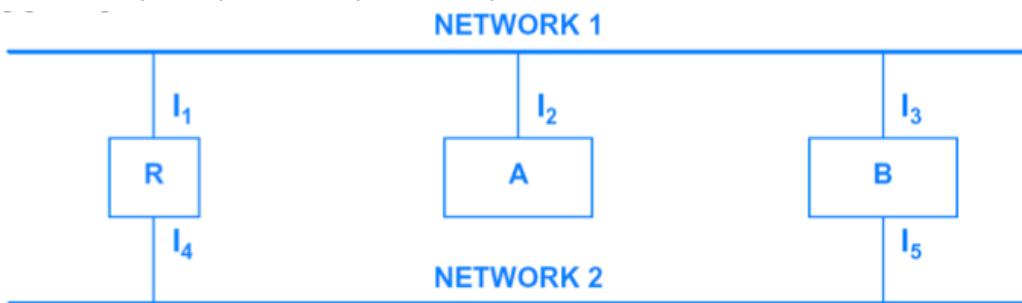
Volendo inviare uno stesso pacchetto a più macchine, si può adoperare un indirizzo di multicast che dovrà essere condiviso da tutte le macchine che sono in multicast.

3.7) I difetti/conseguenze dell'indirizzamento IP

Facciamo un'analisi per capire cosa abbiamo ottenuto.

- Lo spostamento di una macchina implica il cambio di indirizzo IP (pessimo per portatili, se una rete raggiunge il numero massimo di host si richiede una grande operazione di riconfigurazione).
- Il cammino seguito dai pacchetti inviati da una macchina A ad una macchina B sarà tendenzialmente sempre lo stesso, poiché l'indirizzo IP definisce anche il routing.

Riguardo al secondo punto, prendiamo questo esempio:



Supponendo che B voglia parlare con A, può farlo tramite l'interfaccia I₃. Ma se I₃ si guastasse, allora B non potrebbe più comunicare con A!

Ma in realtà esiste un'altra interfaccia, I₅, con la quale sarebbe possibile contattare A: purtroppo B non ne è a conoscenza e non possiamo pretendere che i router conoscano tutte le interfacce possibili di ogni macchina.

- Pacchetti inviati ad un IP con stesso netid seguono la stessa strada: talvolta però ci sono pacchetti speciali che hanno bisogno di seguire una strada diversa. Bisogna trattare il caso negli algoritmi di routing.
- La lunghezza dell'IP porta ad un limite superiore di host connessi alla rete.
- Il netid provoca comunque uno spreco di spazio se la rete fisica non sfrutta tutti gli host assegnati.

3.8) I vantaggi dell'indirizzamento IP

- Mixing di reti di dimensioni diverse.
- Lunghezza fissa degli indirizzi implica minore costo di elaborazione per trattare gli indirizzi.
- Netid facilita l'instradamento.
- Netid facilita la consegna diretta.
- Netid facilita l'amministrazione degli indirizzi.

3.9) Notazione dotted quad

Come è facile immaginare, non è comodo comunicare fra "umani" fornendo 32 bit. Pertanto esiste la dotted quad notation con la quale rappresentare gli indirizzi IP.

Ogni byte viene rappresentato con un numero decimale. I quattro byte (o meglio i quattro numeri decimali ottenuti dai quattro byte) vengono scritti affiancati e separati da un punto.

Quindi, 10000000 00001010 00000010 00011110 diventa 128.10.2.30.

Si noti che si tratta di una notazione solo ad alto livello, mentre le macchine lavorano ovviamente coi bit.

La dotted quad non permette facilmente di capire quale sia la classe di un indirizzo IP, poiché il valore dei primi bit dell'indirizzo è "nascosto" all'interno del primo decimale. Vediamo la lista degli indirizzi IP in dotted quad suddivisi per classi:

Classe	Minimo	Massimo
A	1.0.0.0	126.0.0.0
B	128.1.0.0	191.255.0.0
C	192.0.1.0	223.255.255.0
D	224.0.0.0	239.255.255.255
E	240.0.0.0	255.255.255.254

3.10) Indirizzo di loopback

Dall'immagine sopra potremmo notare un fatto strano: mancano gli indirizzi 127.0.0.0. Perché?

Tali indirizzi sono riservati per il loopback, ovvero per l'invio di un pacchetto da una macchina alla macchina stessa. Il pacchetto non tocca neanche il livello fisico ma "torna su" direttamente.

Come possiamo immaginare la funzionalità è molto utile in fase di testing.

Inoltre, non si deve neanche conoscere l'hostid della macchina.

3.11) Ottenere un netid: ICANN e ISP

Dato che i netid devono essere univoci, allora avremo un'autorità che li distribuisce: ICANN.

Al giorno d'oggi, se si desidera una rete, si contatta il proprio ISP che genererà una subnet della rete dell'ISP stesso con un netid univoco.

4) Traduzione da indirizzi di rete ad indirizzi fisici: ARP

Abbiamo ormai chiaro il concetto che due applicazioni si possono parlare solamente se conoscono reciprocamente gli indirizzi IP. Ma due macchine possono comunicare solo conoscendo reciprocamente l'indirizzo fisico... abbiamo un evidente problema di traduzione.

4.1) Uno sguardo al problema

Il problema è formalizzabile come segue:

A e B sono due macchine nella stessa sottorete fisica e vogliono comunicare. A avrà un indirizzo IP I_A e un indirizzo fisico P_A mentre B avrà un indirizzo IP I_B e un indirizzo fisico P_B .

Le applicazioni useranno I_A ed I_B ma la rete fisica necessita dell'utilizzo di P_A e P_B . È quindi necessario tradurre, **mappare**, un indirizzo sull'altro.

Questa traduzione deve essere effettuata ad ogni passaggio del pacchetto tra un router e l'altro e ovviamente alla fine. Questo è necessario poiché i router sono elementi fisici e hanno bisogno dell'indirizzo fisico del prossimo router per instradare il pacchetto.

4.2) Due tipi di indirizzi fisici

Dobbiamo permettere che esistono due tipologie di indirizzi fisici:

- **Indirizzi fisici fissati all'hardware** che sono tendenzialmente più grandi degli indirizzi IP (es. Ethernet). Fondamentalmente soffrono della problematica di non essere configurabili dall'amministratore di rete.
- **Indirizzi fisici configurabili dall'hardware** e quindi manipolabili dall'amministratore di rete.

Chiaramente prediligeremmo il secondo caso, poiché avremmo la possibilità di configurare l'indirizzo di rete in modo che la risoluzione dell'indirizzo IP sia il più facile possibile.

4.3) Risoluzione mediante mapping diretto

Supponiamo di avere una sottorete i cui indirizzi fisici sono piccoli interi (diciamo minori di 254 bit): potremmo forzare l'indirizzo fisico uguale all'hostid dell'indirizzo IP.

In questo modo la risoluzione sarebbe banale e richiederebbe pochissime istruzioni macchina.

Più in generale la traduzione è effettuata tramite una funzione $P_A = f(I_A)$ dove tendenzialmente f vuole essere il più semplice possibile, altrimenti non vale la pena.

Con indirizzi fisici configurabili

Con un indirizzo fisico configurabile, è un'ottima idea applicare $P_A = \text{hostid}_A$.

Con indirizzi fisici legati all'hardware

Qualora l'indirizzo fisico fosse non configurabile oppure molto grande si può ricorrere ad una **tavella hash di traduzione** e quindi f viene definita in **modo tabellare**.

Il problema della tabella non è tanto l'accesso che è molto veloce e largamente usato in moltissime applicazioni, ma è bensì la **manutenzione della tabella**: essa deve essere tenuta **allineata** in **tutte le macchine** della rete e richiede **aggiornamenti** ogni volta che una macchina entra o esce dalla rete stessa.

L'approccio ISO/OSI è stato quello di avere un server che distribuisce la tabella alle macchine mantenendola così aggiornata (le macchine chiederanno la tabella al server ogni tot tempo).

E se trovassimo invece un sistema con cui creare la tabella senza averla dal server e così facendo mantenerla sempre aggiornata? Avremmo solo più il costo dell'accesso, che, come detto sopra, nelle tabelle di hash è molto ridotto.

4.4) Risoluzione mediante mapping dinamico

Ethernet è uno di quei casi in cui abbiamo difficoltà nel mappaggio dell'indirizzo di rete su quello IP.

Al cambio di una scheda di rete **non vogliamo** dover ri-compilare il software, cambiare indirizzo IP e allo stesso tempo non vogliamo avere una gestione della tabella centralizzata.

Si richiede quindi la **cooperazione fra le macchine** invece dell'impiego di un server... ma la cooperazione è il problema stesso, come possono comunicare le macchine se non conoscono gli indirizzi fisici reciproci?

Si potrebbe sfruttare il **broadcast** di Ethernet, ma con un uso attento: abusare di questa funzionalità significherebbe congestionare la rete.

Di tutto questo si occupa ARP (**address resolution protocol**).

Funzionamento di ARP

Quando A vuole risolvere l'indirizzo I_B allora invia in broadcast sulla rete un pacchetto ARP tramite il quale chiede all'host di indirizzo I_B di rispondere fornendo il suo indirizzo fisico P_B . Nel pacchetto, A includerà P_A affinché B possa rispondere direttamente ad A evitando così il costo di elaborazione di ogni macchina della rete per scartare questa risposta (non è il problema il broadcast in sé poiché sfrutta la banda come pacchetto in unicast).

In questo modo, A otterrà P_B e quindi potrà comunicare con B.

Risulta evidente che ogni macchina debba conoscere il proprio indirizzo IP poiché solamente B dovrà raccogliere il pacchetto inviato da A.

Richieste ARP: quanto mi costate!

Come abbiamo detto prima, il costo delle richieste ARP in broadcast è l'elaborazione di ogni macchina che deve scartare la maggior parte dei pacchetti che viaggiano in rete più che il broadcast stesso.

Abbiamo quindi limitato il broadcast a quello iniziale ma in realtà anche questo è troppo per le macchine: per il momento effettuiamo un broadcast ogni volta che vogliamo inviare un pacchetto! Come possiamo **ridurre le richieste ARP**? Sono le uniche a darcì informazioni!

La cache di risoluzione

Si introduce una cache di risoluzione per aiutarci in questa problematica: la famosa tabella.

Ogni corrispondenza ottenuta viene messa in cache: al momento dell'invio di un pacchetto verrà interrogata la cache e se l'indirizzo fisico richiesto non dovesse essere presente, verrà inviato il pacchetto di richiesta ARP. In questo modo abbiamo ridotto le richieste ARP ad una per ogni "inizio-comunicazione" con una certa macchina.

Ma possiamo fare di meglio.

Raffinamenti di ARP

Possiamo attuare una serie di perfezionamenti al protocollo per "risparmiare" richieste ARP:

- Giacché A deve inviare una richiesta a B, allora probabilmente B dovrà rispondere ad A. Così, **A includerà il suo P_A** per agevolare B nella risposta.
- Quando B riceve una richiesta ARP **inserisce** il binding del mittente nella sua cache anche se **non è il destinatario**: avrà comunque acquisito una corrispondenza IP/indirizzo fisico eventualmente utile.
- Quando una macchina entra in una rete, una volta ottenuto il suo indirizzo IP, manda un pacchetto ARP **"presentandosi"** fornendo quindi il suo binding IP/indirizzo fisico (e tutti, stando al secondo punto, lo mettono in cache).

Stando a queste migliorie parrebbe davvero un'evento raro l'invio di un pacchetto ARP.

Ma attenzione! Abbiamo dimenticato un problema: le informazioni, dopo un certo tempo, diventano inconsistenti poiché la rete muta (le macchine se ne vanno, possono cadere senza preavviso, possono cambiare IP). Le righe della tabella devono quindi essere trattate come in una cache, cioè, eliminate di quando in quando. ARP viene quindi detta **soft stale** poiché a rischio di **stale** (ammuffimento).

ARP soft stale: il timeout della cache

Introduciamo quindi un timeout per ogni record della cache.

Un record della cache può diventare **invalido** quando scatta il suo timeout (in ARP, 20 minuti) oppure quando all'utilizzo dell'informazione contenuta nel record di ottiene un errore.

Chiaramente il timeout di ogni record può essere **ricaricato** quando avviene una comunicazione con la macchina di cui il record specifica l'indirizzo fisico oppure quando riceviamo direttamente un pacchetto ARP dalla macchina: in questo modo eviteremo ritrasmissioni inutili.

4.5) Implementazione di ARP

Parliamo ora dell'implementazione: aspetto da non trascurare ovviamente.

Stando a quanto detto prima dobbiamo risolvere due questioni:

- Come determinare l'indirizzo fisico per un pacchetto IP uscente.
- Come gestire i pacchetti ARP ricevuti.

Determinare l'indirizzo fisico per il pacchetto IP uscente

Se l'indirizzo richiesto è in cache si risolve, altrimenti si procede con una richiesta ARP. Tale richiesta può, per diversi motivi, fallire:

- La macchina target è down.
- La macchina target è molto lenta.
- Il pacchetto ARP può andare perso.

Inoltre:

- Altre macchine potrebbero chiedere lo stesso mappaggio dell'IP.
- Per un po' di tempo si deve mantenere il pacchetto in memoria in attesa del binding.
- Occorre cancellare la entry nella cache dopo un po' di tempo che non viene refreshata o in seguito ad un fallimento.

Tenendo conto dei punti appena espressi, agiremo in questo modo:

- Prima di tutto **salviamo dall'ARP il binding del mittente**.
- Se si ha un ARP request, allora si prepara il pacchetto di risposta e lo spediamo.
- Se si ha un ARP reply, si inviano i pacchetti IP in attesa (che possono essere più di uno o anche nessuno).

L'incapsulamento di ARP

Come per tutti i protocolli, le informazioni portate dai pacchetti ARP sono incapsulate. Nel frame Ethernet, useremo il campo **frame type** con valore (in esadecimale) 0806 per contraddistinguere il tipo "ARP" affinché il destinatario sappia capire quali informazioni gli stiamo dando.

I pacchetti ARP hanno forme diverse, vediamo a scopo didattico il pacchetto ARP per Ethernet.

0	8	16	24	31		
HARDWARE TYPE		PROTOCOL TYPE				
HLEN	PLEN	OPERATION				
SENDER HA (octets 0-3)						
SENDER HA (octets 4-5)		SENDER IP (octets 0-1)				
SENDER IP (octets 2-3)		TARGET HA (octets 0-1)				
TARGET HA (octets 2-5)						
TARGET IP (octets 0-3)						

Giacché i pacchetti ARP hanno forme diverse essi sono **autodescritti** ovvero al loro interno specificheranno la lunghezza stessa del pacchetto (non abbiamo allineamento con i 32-bit). Vediamo ora il significato dei singoli campi:

- **HARDWARE TYPE**: il mezzo fisico.
- **PROTOCOL TYPE**: il protocollo usato (in IP, ARP).
- **HLEN**: quantità di ottetti di cui è composto l'indirizzo fisico (6 nel nostro caso).
- **PLEN**: quantità di ottetti di cui è composto l'indirizzo IP (4 nel nostro caso).
- **OPERATION**: operazione da effettuarsi (1 significa richiesta ARP, 2 significa risposta ARP, 3 significa richiesta RARP, 4 significa risposta RARP).
- **SENDER HA**: indirizzo hardware del mittente.
- **SENDER IP**: indirizzo IP del mittente.

Quando l'operazione è di richiesta, avremo anche i campi **TARGET IP** (se in ARP) e **TARGET HA** (se in RARP) e gli altri riempiti di zero.

Se l'operazione è invece di risposta, avremo tutti e quattro gli indirizzi valorizzati.

4.6) Due macchine con lo stesso indirizzo IP

Occupiamoci ora di questa problematica.

Quando una macchina si presenta annuncia il suo IP: la macchina che riceve tale annuncio e si accorge che il suo IP è uguale a quello appena annunciato, si annuncia a sua volta.

Ci si rende così conto dell'errore e questo viene segnalato tramite una finestra d'avviso del sistema operativo.

Dipenderà poi dal sistema operativo il seguito (ad esempio in Windows XP la seconda macchina perde la connettività e ottiene indirizzo 0.0.0.0 mentre la prima rimane intatta).

Un'altro caso è quando la seconda macchina che ha preso l'indirizzo IP si accorge di averlo copiato e ri-annuncia se stessa. In questo modo avremo un ciclo di annunci fra le due macchine: le altre macchine in rete se ne renderanno conto e smetteranno di inviare pacchetti alle due macchine "litiganti" e segnaleranno il problema con una finestra.

4.7) RARP: conoscere il proprio indirizzo IP

Come può una macchina conoscere il suo indirizzo IP?

Si potrebbe usare un file di configurazione, ma:

- Ci sono macchine senza dischi
- Grande costo di gestione
- Cambi di configurazione sono costosi
- Ci sono macchine portatili che non sono sempre nella rete

La soluzione primordiale proposta da Internet è **RARP**: che risolve solo i primi tre punti.

L'indirizzo IP di una macchina è tenuto da un server di rete col quale comunicheremo tramite l'indirizzo hardware (unica conoscenza della macchina).

Possiamo vedere RARP come estensione di ARP, infatti adotta la stessa struttura (ma il campo type del frame Ethernet varrà, in esadecimale, 8035).

La macchina manda in broadcast una richiesta in cui appare sia come SENDER che come TARGET. Al posto dell'IP del mittente si indica un indirizzo di tutti zeri ("questa macchina"). Solo i server RARP risponderanno e forniranno sia la coppia HA-IP del server e sia il nuovo indirizzo IP che dovrà essere adoperato dalla macchina.

I server RARP sapranno quale indirizzo IP assegnare poiché dispongono di tabelle di configurazione centralizzate. Vi sono altri servizi più sofisticati (**DHCP**) che permettono di assegnare dinamicamente l'indirizzo IP in un range dato. Inoltre, DHCP fornisce anche altri tipi di configurazioni alla macchina (e risolve anche il quarto problema).

Se il RARP non va: server primari e di backup

Qualora un server RARP fosse guasto, la situazione sarebbe chiaramente critica. Introduciamo più server RARP. A questo punto abbiamo due approcci.

Il primo è quello di **assegnare ad ogni macchina un server primario** e lasciare gli altri come secondari. Quest'ultimi risponderanno solamente se il server primario è guasto.

Il secondo consta invece nel far sì che **ogni server risponda con un tempo casuale** evitando così collisioni inutili.

Chiaramente le informazioni dei server RARP devono essere condivise per rispondere tutti allo stesso modo.

DHCP e BOOTP: la vera soluzione

RARP porta con sé alcune problematiche:

- Tanto spazio viene sprecato all'interno del pacchetto Ethernet: un IP soltanto è davvero poco e occorre ben più di un indirizzo IP per configurare una macchina nella rete.
- Il software di rete deve gestire tanti protocolli diversi con tanti costruttori di schede e con tanti modelli di schede: un protocollo di livello fisico è troppo poco performante...

Meglio un protocollo applicativo come lo sono DHCP e BOOTP.

Il funzionamento di questi protocolli è il seguente:

La macchina invia in broadcast una richiesta al server DHCP (oppure il server DHCP è configurato sulla macchina) e così ottiene: indirizzo IP, indirizzo del DNS, indirizzo del gateway (si capiranno meglio tutti questi termini in seguito).

Il protocollo IP

1) IP: filosofia connection-less

Entriamo quindi nel dettaglio di IP.

La sua filosofia ci è oramai chiara, si tratta di un protocollo **connection-less** che permette l'astrazione fra la rete fisica e la rete virtuale. Ora ci occuperemo proprio di come questa astrazione venga effettuata.

Si ricordino tutti i concetti espressi nel primo capitolo "*Introduzione alle reti*" (protocolli, comunicazione verticale, comunicazione fra pari, ecc.).

1.1) Caratteristiche e scopi

IP, servizio di consegna pacchetti di Internet, è un servizio **connection-less**, **best-effort** e quindi **non affidabile**. Si noti che i pacchetti vengono talvolta scartati appositamente dai router e non persi casualmente, poiché non sempre le risorse delle macchine sono sufficienti per gestire le comunicazioni (buffer pieni, ad esempio).

Gli obiettivi di IP sono:

- Definire il formato dell'unità base di comunicazione (il pacchetto).
- Realizzare funzioni di routing e forwarding per l'interconnessione delle varie sottoreti.

Inoltre, seguendo la filosofia connection-less, dovremmo definire:

- Come processare i pacchetti.
- Come e quando generare messaggi di errore.
- Sotto quali condizioni sia possibile scartare pacchetti.

Avremo quattro fondamentali passi: l'**indirizzamento** (di cui abbiamo già parlato), il datagram e la sua **frammentazione**, l'**intradamento** e l'**ICMP** per l'invio di messaggi di errore. Gli ultimi tre sono trattati nei prossimi paragrafi.

2) L'unità di trasferimento: il datagram

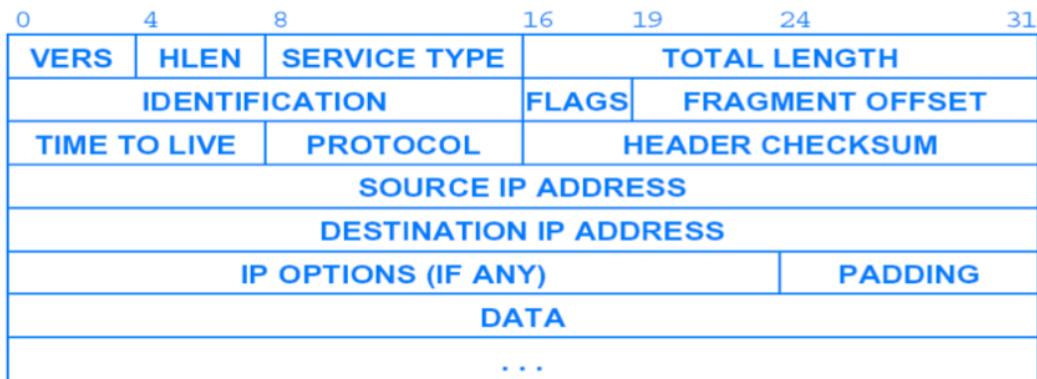
All'interno del protocollo IP chiamiamo l'unità di trasferimento **datagram**.

2.1) La struttura del datagram

Tale datagram sarà siffatto:



La seconda parte, data area, non è definita da IP bensì dai livelli superiori. Più nel dettaglio, il datagram ha la seguente forma:



Tutti i campi hanno lunghezza fissa tranne il campo **IP OPTIONS** che può esserci oppure no. Segue infatti un campo di padding per allinearsi ai 32 bit. Senza opzioni l'header è lungo 5 parole di 32 bit, 20 ottetti, 160 bit.

Esaminiamo con la dovuta calma tutti i campi del pacchetto.

- **VERS**: versione del protocollo adottato.
- **HLEN**: lunghezza dell'header in parole di 32 bit.
- **TOTAL LENGTH**: lunghezza massima di tutto il pacchetto in ottetti (header incluso) (perciò si possono avere **datagram lunghi massimo 64KB**).

Ci servono entrambe le lunghezze poiché il livello hardware non sempre fornisce informazioni corrette sulla lunghezza della DATA AREA di IP.

- **SERVICE TYPE**: è un campo talvolta ignorato dai router che definisce la qualità del servizio ovvero la precedenza che un certo pacchetto dovrebbe avere sugli altri. Abbiamo tre bit per esprimere la precedenza, poi, altri tre per definire la QoS (basso ritardo, alto throughput, alta affidabilità) ed infine due bit inutilizzati. La struttura verrà poi cambiata nel 1990 per permettere anche altre tipologie di servizio: si hanno quindi 6 bit per il codepoint e 2 bit non utilizzati.

Per comprendere appieno gli altri campi è necessario essere ben consci del funzionamento dell'**incapsulamento**. Qual è la relazione che lega i frame fisici con i pacchetti IP? I datagram IP non sono trattati direttamente dall'hardware e quindi hanno dimensione massima a discrezione del progettista. In IP la lunghezza è pari a 16 bit, cioè fino a 64KB informazioni rappresentabili. Qualora fosse possibile, gradiremmo incapsulare tutti i 16 bit in un solo frame fisico. Ma questa è una pretesa molto grossa: **ogni mezzo fisico ha una massima dimensione del frame**.

2.2 Dimensione del datagram: MTU e frammentazione

Ogni mezzo fisico dispone quindi di un certo valore di MTU (**Maximum Transfer Unit**). Nel caso di Ethernet, 1500 Bytes.

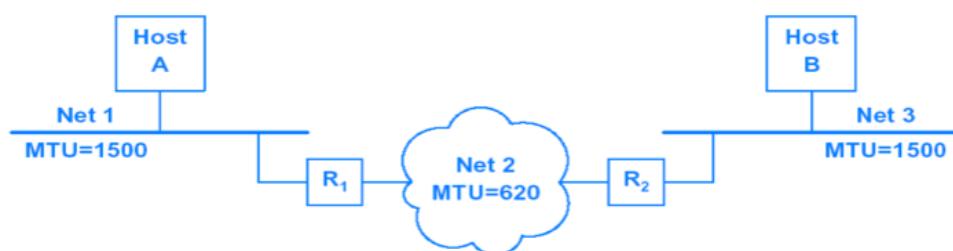
L'idea migliore è quella di creare comunque datagram di dimensioni ottimali per il livello superiore ad IP e che poi, se necessario per via della rete fisica, IP stesso si occuperà di **frammentare**. Ovviamente sarà il protocollo IP paritario dall'altro lato a dover **riassemblare i frame** riottenendo così il datagram integro.

La frammentazione, quando avviene, avviene in un qualche router fra la rete origine e quella destinazione oppure sull'host mittente.

La **dimensione** del frammento è quella **massima permessa dall'MTU** della rete fisica, ma con il vincolo che sia **multiplo di otto ottetti**.

Ciascun frammento avrà la stessa struttura del datagram iniziale ovvero disporrà di header (con alcuni campi diversi, dopo vedremo) e della parte dei datagram.

Vediamo un esempio:



Il router R₁ si occuperà di frammentare le comunicazioni da A verso B ed R₂ si occuperà di frammentare le comunicazioni da B verso A.

Le deframmentazioni, invece, avverranno sulla macchina ricevente (su A i messaggi inviati da B e su B i messaggi inviati da A).

Un singolo datagram come questo:



verrà quindi suddiviso in tre frame come questi:



Fragment 1 (offset 0)



Fragment 2 (offset 600)



Fragment 3 (offset 1200)

Il **FRAGMENT OFFSET** indicato nell'head del datagram (paragrafo 2.1) va moltiplicato per otto e indica a che "punto" del dato siamo arrivati.

Capiremo se siamo a fine datagram (è arrivato l'ultimo frame) grazie a un bit contenuto nella parte **FLAGS**, il terzo precisamente, che se è a 1 indica che ci sono ancora frame a seguire mentre se è a 0 indica che siamo all'ultimo.

Questi due campi, oltre al campo **TOTAL LENGTH** (che indica la lunghezza del frame) sono gli unici tre a cambiare da un frame all'altro.

Questo tipo di procedimento fa sì che il riassemblaggio venga effettuato solo sull'host destinario: ci sono vantaggi e svantaggi intrinseci.

Il vantaggio più evidente è che evitiamo di riassemblare e frammentare di continuo, mentre lo svantaggio altrettanto evidente è il fatto che frame con MTU molto bassa possono viaggiare anche su reti con MTU invece alta.

Sono possibili frammentazioni in cascata (ma l'offset tenuto sarà sempre quello del frammento originale da cui è partita la seconda frammentazione ovvero non riframmenteremo da 0 ma suddivideremo a partire dall'offset padre).

Riassemblaggio dei frammenti

Il riassemblaggio dei frammenti è basato su un reassembly timer che il ricevente fa partire quando ricevere il primo frammento. Qualora il timer scadesse prima della totale ricezione del datagram, verranno scartati tutti i frame ricevuti relativi al datagram. Il motivo dell'esistenza del timer è il fatto che frame correlati fra loro hanno il campo **IDENTIFICATION** in comune, che, però, essendo di soli 16 bit non può essere "trattenuto" troppo a lungo.

Come facciamo a sapere se **abbiamo ricevuto tutti i frame** (non è sufficiente attendere l'arrivo dell'ultimo poiché i frame potrebbero essere instradati in ordine diverso)?

Semplicemente si calcola la somma dei byte ricevuti e si confronta con l'offset dell'ultimo frammento più la parte dati dell'ultimo frammento stesso. Se sono lo stesso valore, allora disponiamo di tutti i frame e possiamo procedere al riassemblaggio.

Il campo FLAG

Abbiamo parlato del terzo bit del campo **FLAGS**: gli altri?

Il primo bit è riservato (deve essere 0).

Il secondo invece è settato a 1 se è **vietata la frammentazione** (se si finisce su reti con MTU troppo basso, il datagram è scartato).

Il campo Time to Live (TTL)

Abbiamo ancora il campo **TIME TO LIVE** da descrivere.

Esso specifica il tempo in secondi per cui un datagram può stare nella rete.

Chi tratta il pacchetto deve decrementare il TTL con il tempo impiegato per trattarlo (compresa la coda di spedizione).

Quando il TTL è negativo, il pacchetto deve essere scartato.

Anche i router, al passaggio del frame, devono decrementare di uno almeno il TTL (per evitare che vi siano pacchetti vaganti per la rete in eterno a causa di instradamenti errati).

I campi PROTOCOL & HEADER CHECKSUM

Mancano all'appello solo più i campi **PROTOCOL** e **HEADER CHECKSUM**.

- **PROTOCOL**: è analogo al campo frame type di Ethernet: vi possono essere più protocolli utenti di IP e in questo modo l'IP ricevente sa a chi passare i dati ricevuti (ogni protocollo utente formatta i dati in maniera diversa!).

- **HEADER CHECKSUM**: controlla l'integrità dell'header (somma in complemento a 1 delle parole di 16 bit che costituiscono l'header) ed infine si prende il complemento a 1 del risultato. Questo campo viene ricalcolato ad ogni hop (è possibile calcolare un checksum incrementale). I **dati non vengono controllati**, sono i protocolli a livello superiore ad occuparsi di questo.

Esempio di frammentazione

Supponiamo di avere un datagram lungo 1.400 byte. Una prima frammentazione fa sì che il datagram sia suddiviso in tre frame, rispettivamente di 600 byte, 300 byte e 200 byte. Mentre il primo frame giunge a destinazione così com'è, gli altri due sono costretti a passare in due reti con MTU rispettivamente di 300 Byte e 100 Byte.

I frame risultati con relativi offset saranno quindi (a sinistra l'offset, a destra la lunghezza del frame):

Offset: 0	600 Byte
Offset: 600	300 Byte
Offset: 900	300 Byte
Offset: 1200	100 Byte
Offset: 1300	100 Byte

3) L'instradamento IP

Veniamo ora alla terza caratteristica fondamentale di IP: l'instradamento.

Dobbiamo poter "far viaggiare" i pacchetti da un host all'altro in modo veloce e intelligente. Sfrutteremo gli elementi hardware di cui abbiamo parlato a lungo, i router.

I router e gli host saranno quindi capaci di **intradare** (scegliere un path da far seguire al pacchetto) ma solamente i router sono capaci di **inoltrare** (far passare un pacchetto da una rete ad un'altra rete).

3.1) Consegnna diretta e consegna indiretta

Lo scenario è abbastanza semplice, un host vuole inviare un pacchetto: se il pacchetto è diretto a qualcuno all'interno della stessa rete fisica (**consegnna diretta**) allora verrà inoltrato dall'host direttamente (conoscerà l'indirizzo fisico della macchina grazie ad una richiesta ARP). La consegna diretta, quindi, non coinvolge i router!

Se invece l'host vuole inviare un pacchetto "fuori" (**consegnna indiretta**), lo invierà al router il quale lo invierà ad un altro router e così via finché non si giungerà alla rete di destinazione e quindi al destinatario.

L'host può rendersi conto se il pacchetto che sta inviando è per qualcuno all'interno della sua rete grazie al netid (parleremo meglio di questo in seguito).

Chiaramente il caso più interessante è quello della consegna indiretta che coinvolge i router.

Il datagram composto dall'host avrà come **indirizzo destinatario quello dell'altro host** con cui vuole dialogare, ma verrà **invia**to (in seguito alla risoluzione dell'indirizzo fisico del router) **al router!**

Sorgono spontanee alcune domande:

- Come fa il mittente a sapere quale router scegliere?
- Come fa un router a scegliere un altro router?
- Come fa un router ad apprendere nuove strade o a correggere quelle strade che sono divenute inconsistenti?

3.2) Table-driven IP routing

È questo sistema a rispondere alle tre domande di cui sopra.

La **tavella di instradamento IP** contiene le informazioni sulle **possibili destinazioni note** e sul modo di raggiungerle.

Chiaramente la tavella non può contenere tutte le reti attualmente collegate ad Internet e quindi si limita a contenere le informazioni necessarie per effettuare un **hop** cioè, un salto successivo.

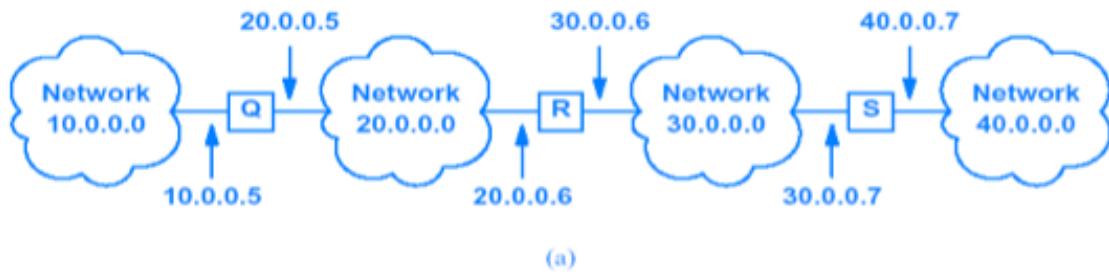
La tavella di instradamento contiene coppie (N, R) dove N è l'**indirizzo di una rete destinazione** (32 bit) e dove R è l'**indirizzo IP** (32 bit) **del router** che permette la connessione alla rete N.

Quindi, nella colonna R avremo solamente i router direttamente raggiungibili dal router/host possessore della tavella!

Si noti che sono contenuti gli indirizzi IP dei router e non quelli fisici (poiché in questo modo astraiamo maggiormente la rete logica da quella fisica, prevenendo problematiche causate dal cambio fisico dei router, ecc.).

Pertanto la tavella è un array a due colonne con tante righe quante sono le destinazioni direttamente raggiungibili.

Vediamo ora un esempio per chiarificare il concetto.



TO REACH HOSTS ON NETWORK	ROUTE TO THIS ADDRESS
20.0.0.0	DELIVER DIRECTLY
30.0.0.0	DELIVER DIRECTLY
10.0.0.0	20.0.0.5
40.0.0.0	30.0.0.7

La tabella sotto presentata è quella del router R, poiché è in grado di raggiungere direttamente i network 20.0.0.0 e 30.0.0.0 mentre deve instradare sull'indirizzo IP 20.0.0.5 (appartenente al router Q) per andare al network 10.0.0.0 e deve instradare sull'indirizzo IP 30.0.0.7 (appartenente al router S) per andare sul network 40.0.0.0.

Vantaggi e svantaggi

Abbiamo due grandi vantaggi:

- Le tabelle di routing si **basano sulle reti** e non sugli **host** quindi la loro dimensione dipende dalla quantità di reti raggiungibili: il numero di record è, così, drasticamente minore rispetto alla soluzione di mantenere gli indirizzi IP degli host.
- Le tabelle, quindi, aumentano di dimensione solamente quando viene aggiunta una nuova rete.

Purtroppo sono presenti alcuni svantaggi:

- Il traffico verso una rete segue sempre lo stesso percorso (dettato dalle tabelle).
- Se vi sono più strade con cui raggiungere una stessa rete, verrà usata solo una strada. Pensando ad un guasto sulla strada designata, la connessione cade mentre invece si potrebbe ancora usare la strada alternativa. Si è quindi costretti a modificare la tabella di routing.
- I cammini fra due reti nelle due direzioni possono essere diversi (i router devono cooperare affinché la comunicazione bidirezionale funzioni sempre).

Inoltre questa impostazione non permette di instradare basandosi sul TOS (Type of Service) né di dividere il traffico su due strade equivalenti.

Il default router

Per contenere il numero di righe di una tabella è stato introdotto il **default router**. Quando un router non trova una corrispondenza tra il pacchetto che deve inoltrare e la sua tabella (quindi non esiste record nella tabella per il destinatario espresso nel pacchetto) allora tale pacchetto viene inviato al default router.

I default router sono "più informati" dei router classici, cioè sono, in sostanza, router "a più alto livello" che hanno una conoscenza maggiore della rete. Non è escluso che a sua volta un default router abbia un default router e così via.

Strade specifiche per gli host

È possibile, in alcuni casi, definire strade particolari per alcuni host: questo poiché può essere di nostro interesse far seguire una strada meno trafficata a pacchetti destinati ad un certo host di una certa rete.

L'algoritmo di routing

Quanto detto è esplicitato dal codice che esegue il routing:

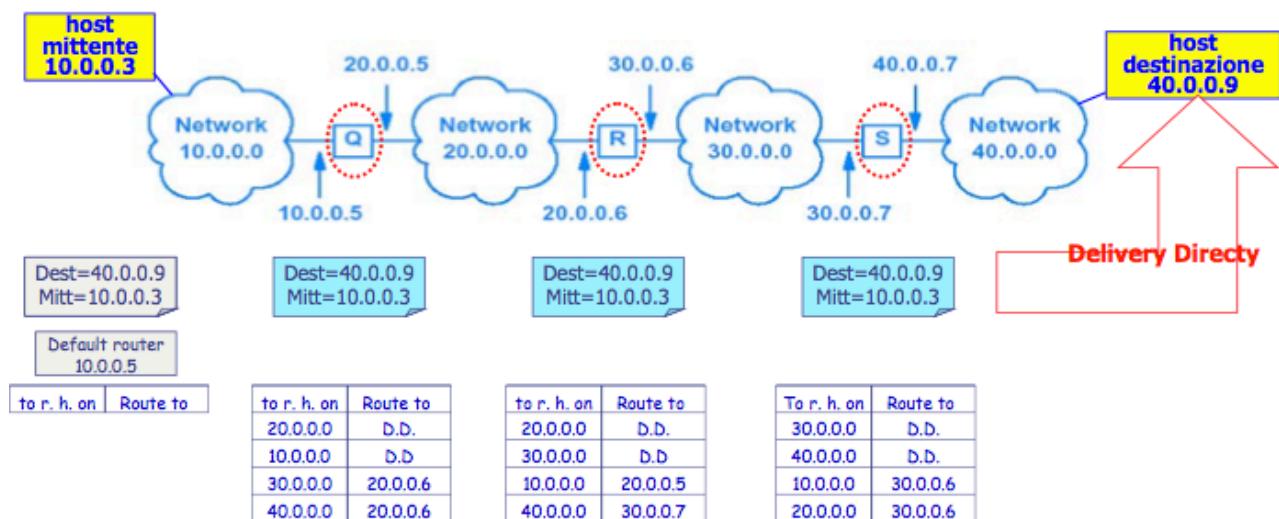
ForwardDatagram (Datagram , RoutingTable)

```

Extract destination IP address, D, from the datagram;
if the table contains a host-specific route for D
    send datagram to next-hop specified in table and quit;
compute N, the network prefix of address D;
if N matches any directly connected network address
    deliver datagram to destination D over that network;
    (This involves resolving D to a physical address,
     encapsulating the datagram, and sending the frame.)
else if the table contains a route for network prefix N
    send datagram to next-hop specified in table;
else if the table contains a default route
    send datagram to the default router specified in table;
else declare a forwarding error;

```

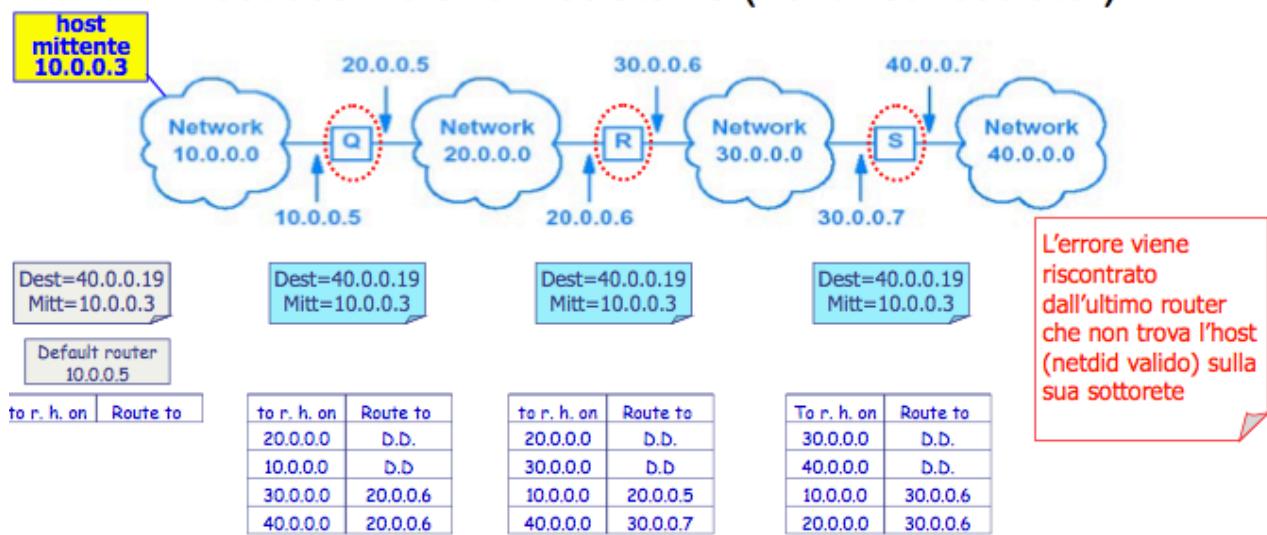
Come si vede è possibile che avvenga un *forwarding error*: questo solamente se la conoscenza del router è incompleta oppure la destinazione non esiste. Se c'è un default router, non è possibile che accada un forwarding error.

Esempi di routing

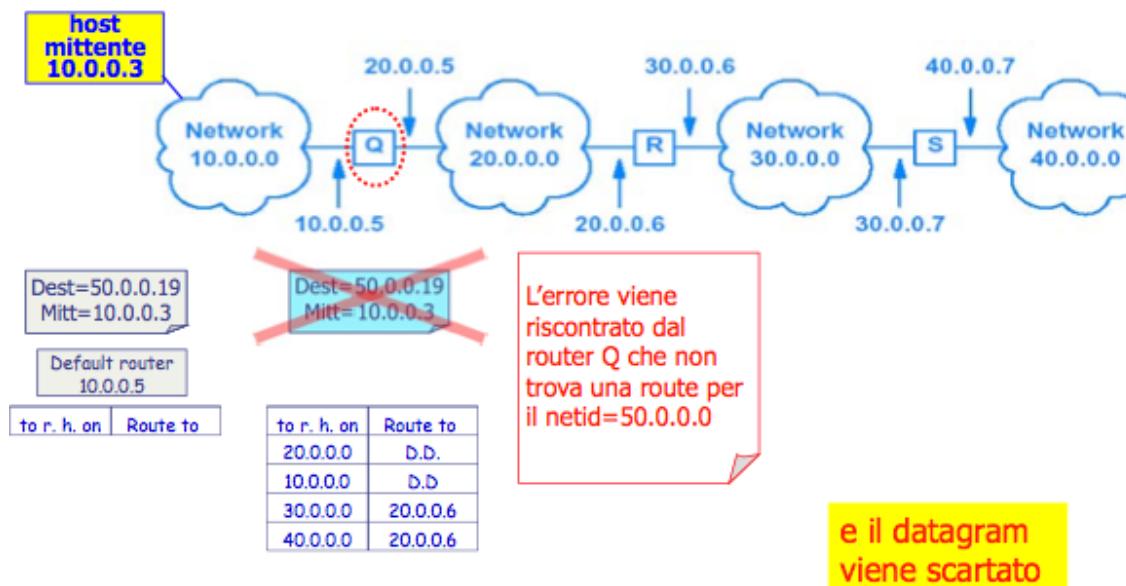
Vogliamo inviare da 10.0.0.3 a 40.0.0.9.

Al primo passo la tabella è vuota quindi scegliamo il default router (che è presente). Tale router risulta essere Q, il quale scopre che per inviare un pacchetto a 40.0.0.0 l'hop deve essere per R, su 20.0.0.6. Così via fino al delivery.

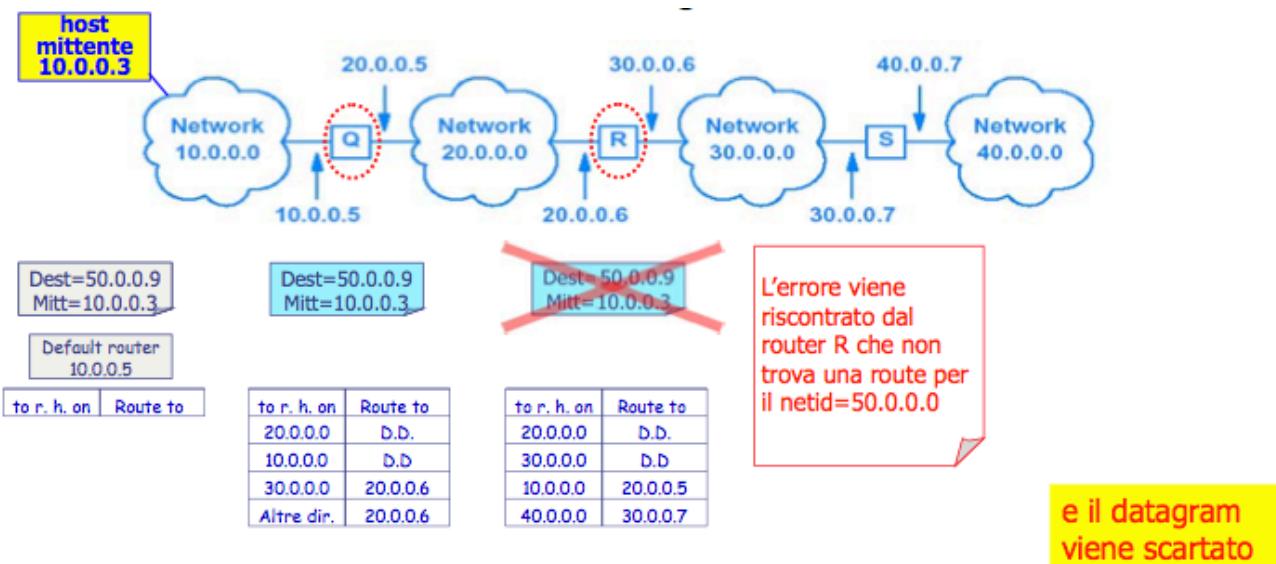
Chiaramente al passo finale verrà effettuata la traduzione da indirizzo IP ad indirizzo fisico tramite ARP.



In questo esempio vogliamo inviare a 40.0.0.19. Tale macchina non è però presente all'interno della rete 40.0.0.0. L'errore verrà riscontrato solo alla fine.



Qui, invece, il netid non esiste all'interno della rete (50.0.0.0) e quindi al secondo hop il router Q non trova alcuna corrispondenza nella tua tabella per quel netid. Non disponendo di default router, riscontra un errore di trasmissione e scarta il datagram.



In quest'ultimo esempio, la seconda tabella ha un indirizzo di routing predefinito dove, qualora non ci fossero corrispondenze, il pacchetto viene inviato. R, che non ha nulla del genere e non ha un default router, sarà costretto a scartare il datagram (la rete 50.0.0.0 non esiste).

Gestione datagram entranti

I datagram in arrivo vengono gestiti nel seguente modo:

- Se si è **host** si verifica che siano stati ricevuti sull'interfaccia corretta, se così è allora si procede all'inoltro al livello superiore (bisogna capire quale sia quello giusto!). Se invece l'interfaccia su cui ha ricevuto il datagram non è corretta, esso va scartato.
- Se si è **router** bisogna allo stesso modo controllare la coerenza delle interfacce. Se il datagram è destinato al router che l'ha ricevuto si sfrutta il campo PROTOCOL per capire a quale livello superiore inoltrare il pacchetto. Se invece il datagram non è per quel router, si procede all'instradamento decrementando il TTL del datagram.

Le questioni relative al riconoscimento della consistenza delle interfacce non è così banale: bisogna trattare i casi di limited broadcast/direct broadcast, subnetting e multicast che rendono più complicato l'intero controllo.

Perché l'host deve scartare i pacchetti che riceve se non sono a lui destinati? La funzione di inoltro non è implementata nella macchina per i seguenti motivi:

- Utilizzo della CPU dell'host (che non è detto sia sufficiente e comunque non è stata adibita a quell'uso).
- I router inviano anche errori: se gli host ne fossero in grado allora potrebbero inondare il mittente!
- Errori semplici possono generare grande caos (pensiamo al broadcast: ogni host potrebbe inoltrare un pacchetto ricevuto in broadcast! Terribile!).
- I router inoltre si scambiano messaggi per mantenere le tabelle corrette: gli host sono troppi per poter cooperare.

Impostare le tabelle di routing

Tutto si basa sulle tabelle di routing. Esse devono essere **consistenti** e **corrette**: IP non può occuparsi anche di questo.

Sarà un **protocollo di livello applicativo** a mantenere tutte le tabelle aggiornate. Ne parleremo, quindi, a tempo debito.

4) Gestione degli errori: ICMP

Non rimane che parlare dell'ultima parte di IP, il protocollo che tratta gli errori, l'**Internet Control Message Protocol**. Ci si prepone l'obiettivo di segnalare eventuali errori/anomalie alla sorgente.

Come sappiamo nessun sistema funziona al 100%, è sempre possibile un fallimento delle linee di comunicazione piuttosto che una disconnessione improvvisa della macchina destinatario o ancora i router intermedi troppo congestionati.

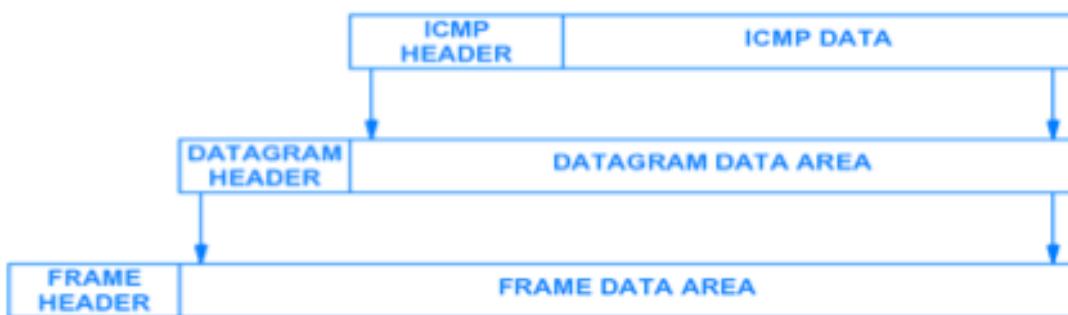
ICMP è quel protocollo che permette la **segnalazione** di errori **da parte di un router ad altri router o host**, oltre che a fornire le funzionalità per permettere ai **router di comunicare fra loro**.

4.1) Introduzione a ICMP: un protocollo indipendente

ICMP è incluso in IP ma è in realtà un protocollo indipendente.

Semanticamente lo vediamo incluso all'interno di IP e quindi non è un livello a se stante ma in realtà lavora per conto suo. Allo stesso tempo, però, non può esistere IP senza ICMP!

Perciò i messaggi ICMP sono **inclusi nel campo data del datagram IP**! Ma allora come possiamo distinguere un messaggio che deve essere recapitato a ICMP invece che a IP? Sfrutteremo il campo PROTOCOL (settato a 1, significa che il datagram IP è in realtà per ICMP).



I messaggi ICMP sono inviati tipicamente da router ma possono anche essere generati da host, con alcune restrizioni. Questo perché gli host, come sempre, sono meno controllabili e si vuole evitare una congestione dettata dall'invio superfluo di messaggi ad opera degli host.

Un utente IP riceve un ICMP solamente se può essere interessato all'errore o ne è il fautore. Pensiamo ad un forwarding error: non verranno avvertiti tutti i router intermedi ma solamente la sorgente del datagram (la quale, comunque, non è detto che sia in grado di risolvere il problema... proprio come accade nella casistica del forwarding error). Questo accade perché i router non sono facilmente identificabili nella rete, essi sono in grado di definire e cambiare le proprie tabelle di routing e quindi risalire al percorso "all'indietro" è molto complesso.

Infine, ICMP è un meccanismo di **segnalazione e non di correzione** degli errori!

4.2) Formato dei messaggi ICMP

Il messaggio ICMP è siffatto:

0	8	16	31
TYPE	CODE	CHECKSUM	

Il campo TYPE (8 bit) specifica il tipo di messaggio mentre CODE (8 bit) specifica un'ulteriore definizione della problematica. Il CHECKSUM (16 bit), invece, riguarda il solo messaggio ICMP.

Se il messaggio ICMP è relativo ad un datagram, allora **all'interno del messaggio avremo anche l'header IP e i primi 64 bit** del datagram che ha generato/è coinvolto nell'errore.

L'uso di ICMP

Onde evitare un'abuso di ICMP, non si generano messaggi ICMP in risposta altri messaggi ICMP, quando l'errore è causato da un messaggio IP multicast o broadcast oppure quando l'errore è generato da un frammento diverso dal primo.

Vediamo ora un po' di messaggi ICMP, i più usati/interessanti.

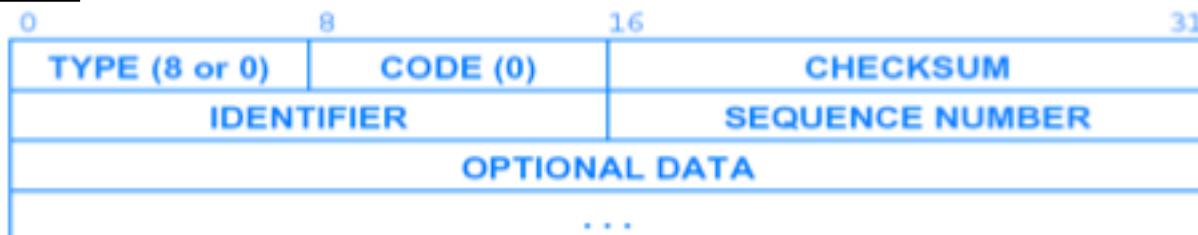
4.3) Messaggio ICMP: Echo Request/Echo Reply

È il messaggio generato dal famoso comando **ping** da terminale.

Si risponde alla Echo Request con una Echo Reply. Questo sistema è sfruttato per i seguenti motivi:

- Verificare se il software IP della macchina da cui si fa request funziona correttamente.
- Verificare se i router intermedi funzionino correttamente (da sorgente a destinazione).
- Verificare se il software IP della macchina a cui si fa request (che prepara la reply) funziona correttamente, sa gestire gli interrupt ed è in grado di rispondere.
- Verificare se i router intermedi funzionino correttamente (da destinazione a sorgente).
- Il software IP della macchina da cui si fa request è anche in grado di ricevere.

Formato



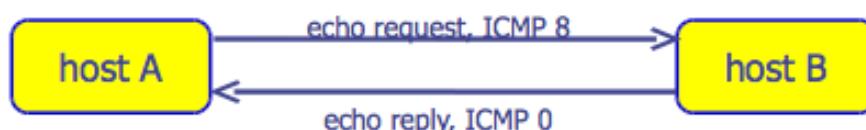
Il type vale 8 se effettuiamo una request e 0 se effettuiamo una reply.

I tre campi IDENTIFIER, SEQUENCE NUMBER e OPTIONAL DATA permettono al mittente di calcolare statistiche sul **round trip time** e sui tassi di perdita.

Si può effettuare una request per datagram di diverse dimensioni, quindi la reply sarà lunga quanto la request per soddisfare la richiesta del mittente.

Nonostante l'utilizzo di request e reply sia sempre correlato, i due messaggi ICMP sono diversi poiché hanno type diverso.

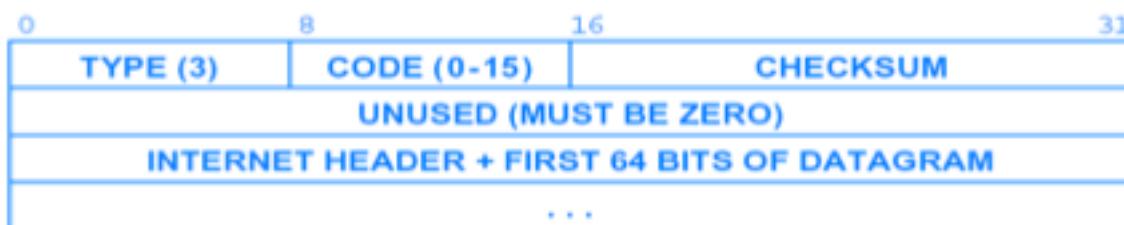
Ecco il normale svolgimento di una sequenza request/reply.



4.4) Messaggio ICMP: destinazione non raggiungibile

Se un router non riesce a raggiungere una destinazione allora invia un ICMP al mittente con type = 3.

Formato



A seconda del CODE, avremo il motivo per cui l'instradamento è fallito:

Code Value	Meaning
0	Network unreachable
1	Host unreachable
2	Protocol unreachable
3	Port unreachable
4	Fragmentation needed and DF set
5	Source route failed
6	Destination network unknown
7	Destination host unknown
8	Source host isolated
9	Communication with destination network administratively prohibited
10	Communication with destination host administratively prohibited
11	Network unreachable for type of service
12	Host unreachable for type of service
13	Communication administratively prohibited
14	Host precedence violation
15	Precedence cutoff in effect

4.5) Messaggio ICMP: controllo della congestione e del flusso dei datagram

IP è connection less e quindi i router non possono riservare memoria/CPU/risorse di rete.

Parliamo di **congestione** quando mancano le risorse necessarie per gestire i datagram in arrivo. La congestione può accadere per tre motivi:

- Una macchina può generare più traffico di quanto la sottorete sappia assorbire.
- Molte macchine creano un traffico aggregato che la sottorete non sa assorbire.
- Il traffico che arriva ad un router è superiore a quello che il router riesce ad elaborare.

Ogni casistica crea uno scenario a se:

- 1) **Singola sorgente**: una macchina è connessa ad una LAN veloce e si affaccia ad una rete più lenta tramite un router.
Il router ha, così, più datagram in entrata che in uscita: **congestione in uscita**.
- 2) **Multi sorgente**: anche se ogni macchina è sotto la soglia del traffico smaltibile dalla rete più lenta, il traffico di più macchine sommato può riportarci al punto 1: **congestione in uscita**.
- 3) **In ingresso**: il router non è abbastanza potente da gestire tutto il traffico in entrata dalla/dalle sue sottoreti: **congestione in entrata**.

Nei primi due casi, il router **si accorge** della situazione mentre invece nella terza non se ne rende neanche conto! Pertanto la segnalazione avverrà solamente nei primi due casi e non nel terzo.

Cosa c'è sotto davvero

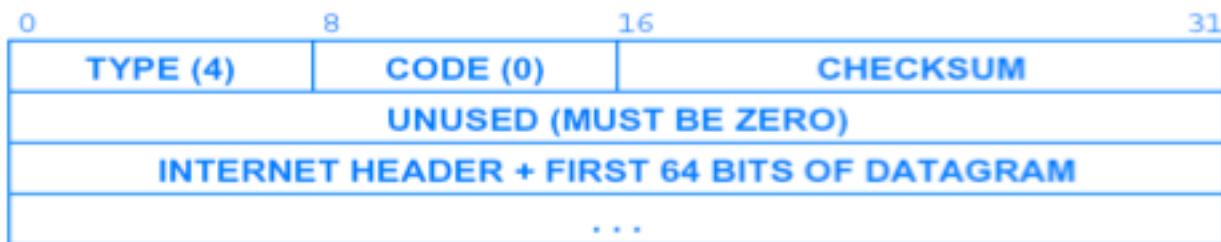
Sembrerebbe un banale problema di velocità... ma cosa c'è sotto davvero?

Prima di tutto le macchine potrebbero condurre **riscontro** (se i messaggi non vengono recapitati, allora la macchina può diminuire il suo throughput) oppure **controllo di flusso** (ci sono elementi di protocollo che riescono ad informare la macchina del fatto che il ricevente è in grado di ricevere solo un certo numero di messaggi).

Da cosa dipende la **velocità di una rete**? Ovvero, quanti messaggi si possono inviare su quella sottorete?

- Velocità del mezzo fisico
- Velocità di gestione hardware/software
- Protocollo di riscontro
- Protocollo di controllo di flusso

Qualora il traffico da inviare fosse più grande del traffico che la sottorete di destinazione è in grado di ricevere, IP accoda i messaggi in appositi buffer. Se i buffer finiscono, però, si procede ad inviare un **ICMP source quench** ovvero si invita la sorgente a diminuire la frequenza di messaggi inviati.

Formato

Il tipo corrispondente è il 4.

Normalmente, quando le code diventano troppo lunghe (RED = Random Early Discard) oppure sono piene si inizia ad inviare questo messaggio: la sorgente diminuirà gli invii finché non smetterà di ricevere messaggi source quench.

Si noti che, spesso, la sorgente smorzata non è la sola causa della congestione (caso 2 esposto prima) e la sorgente non è detto sia IP bensì i livelli superiori (applicativo, TCP, UDP).

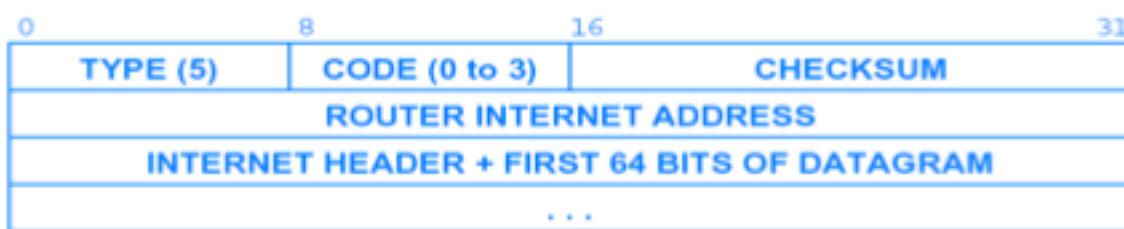
Come si può intuire, questo metodo non permette una gestione sicura e pulita del controllo di flusso.

4.6) Messaggio ICMP: richieste di cambiamento di route

I router sono in grado di scambiarsi informazioni per far sì che le tabelle di routing siano sempre corrette.

Al boot, i router si suppone sappiano le strade corrette e gli host iniziano con informazioni minimali, apprendendo poi dai router le varie strade. Gli host iniziano quindi, tipicamente, con il solo indirizzo del default router.

Quando un router si rende conto che un **host sta usando una strada errata o poco conveniente** per raggiungere una destinazione allora invia un messaggio ICMP di **redirect** all'host stesso.

Formato

Il codice definisce se il redirect deve essere effettuato per la Net, per l'host, per il service type e il net, per il service type e host.

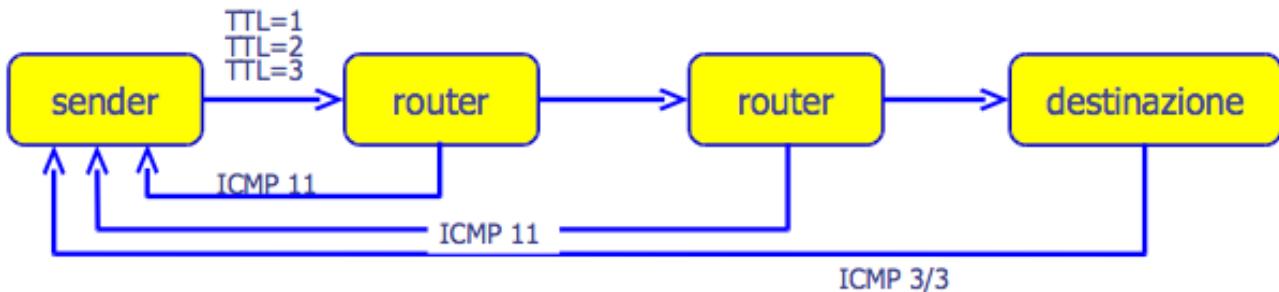
Se le situazioni sono troppo complesse, però, non può bastare un messaggio di redirect per ristabilire gli ordini delle tabelle di routing. Sarà un protocollo di livello applicativo a gestire il tutto.

4.7) Messaggio ICMP: Time To Leave scaduto

Come sappiamo i router devono decrementare di uno il TTL dei datagram IP ogni volta che ne trattano uno.

Se durante questa operazione il TTL diventa zero, allora si procede con l'invio di un messaggio ICMP time exceeded.

Il programma applicativo **traceroute** usa TTL crescenti per conoscere i router sul cammino verso la destinazione.



Formato

0	8	16	31
TYPE (11)	CODE (0 or 1)	CHECKSUM	
UNUSED (MUST BE ZERO)			
INTERNET HEADER + FIRST 64 BITS OF DATAGRAM			
...			

Il tipo assegnato è l'11, mentre il codice serve a capire se abbiamo un **time exceed** (come descritto sopra) oppure un **fragment reassembly time exceed** (i frame di un datagram vengono messi in attesa finché non ci sono tutti prima di procedere al riassemblaggio. Se passa troppo tempo, allora viene inviato questo tipo di messaggio).

4.8) Messaggio ICMP: richiesta di subnet mask

È necessario sapere cosa siano le subnet per capire l'utilità di questo ICMP message. In ogni caso, quando un host ha bisogno di una subnet mask può richiederla ad un singolo host oppure in broadcast e riceverà il **subnet mask address** in risposta.

Formato

0	8	16	31
TYPE (17 or 18)	CODE (0)	CHECKSUM	
UNUSED (MUST BE ZERO)			
INTERNET HEADER + FIRST 64 BITS OF DATAGRAM			
...			

4.9) Messaggio ICMP: annuncio dei router

Inviato dai router, permette agli host di sapere quali router sono presenti sulla rete in seguito al loro boot. Una alternativa sono i protocolli BOOTP e DHCP. In pratica i router, periodicamente, inviano questa informazione in broadcast permettendo agli host di conoscere i nuovi router. Le informazioni inviate hanno una LIFETIME di 30 minuti (dopodiché vengono eliminate) e l'annuncio viene spedito ogni 10 minuti.

Formato

0	8	16	31
TYPE (9)	CODE (0)	CHECKSUM	
NUM. ADDRS	ADDR SIZE (1)	LIFE TIME	
	ROUTER ADDRESS 1		
	PREFERENCE LEVEL 1		
	ROUTER ADDRESS 2		
	PREFERENCE LEVEL 2		
		...	

4.10) Messaggio ICMP: richiesta di router

Chiaramente gli host non devono attendere 10 minuti per sapere quali sono i router connessi alla sottorete e quindi è possibile compilare direttamente un messaggio di richiesta: riceveranno quindi un annuncio da parte dei router.

Formato

0	8	16	31
TYPE (9)	CODE (0)	CHECKSUM	
	RESERVED		

5) Subnetting e indirizzi senza classe

Fin ora abbiamo sempre detto che **una rete fisica = almeno un indirizzo IP**.

Si tratta di un limite pesante dal quale vogliamo slegarci.

5.1) Ciò che sappiamo fin ora

Ogni rete fisica ha assegnato un **netid**: la prima parte dell'indirizzo IP (che è comune a tutti gli host di una stessa rete fisica). A seconda della classe dell'IP (A, B, C), il netid è più o meno esteso.

Abbiamo già parlato del fatto che questo rappresenti un vasto spreco di indirizzi per indirizzi classe A e B (che hanno troppi host assegnati, decisamente troppo difficili da gestire).

Lo schema originale di indirizzamento IP (anni '70) non prevedeva la vasta crescita di utenze che negli anni successivi la rete Internet avrebbe avuto e quindi gli indirizzi IP, ora come ora, sono troppo pochi rispetto alle reti che ci sono e le reti presenti sprecano indirizzi IP (saturando invece le tabelle di routing con costi in termini di efficienza).

Vorremmo invece una **struttura a siti**, dove ogni sito può, al suo interno, gestire i suoi indirizzi IP ma che esce verso la rete con un IP soltanto (e quindi come una rete unica). Un sito può essere ad esempio l'Università degli Studi di Torino.

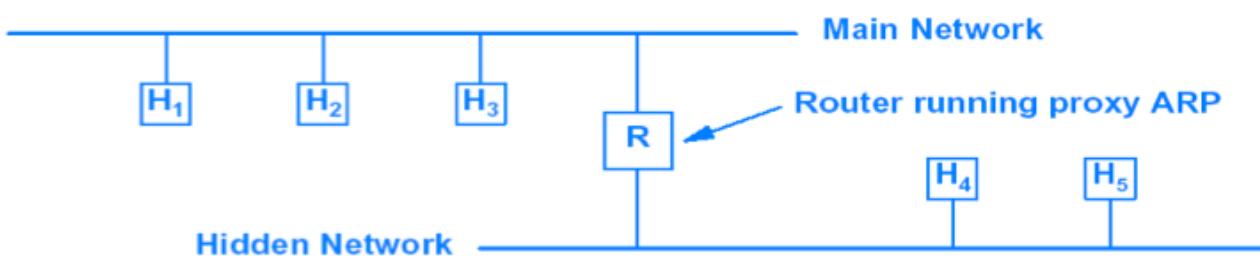
Le proposte effettuate storicamente per la risoluzione del problema sono state:

- **Proxy ARP**
- **IP subnet**
- Reti punto-punto anonime
- **Indirizzamento senza classi (CIDR)**

Ne vedremo solamente tre (le reti punto-punto anonime saranno trattate solo superficialmente).

5.2) Proxy ARP

Adoperabile solamente fra due reti collegate fra loro da un router appositamente configurato per il Proxy ARP.



Abbiamo un Main Network che supponiamo esistere da un po'. Per qualche necessità vogliamo ampliarlo con un altro network, il nostro Hidden Network. Tradizionalmente dovremmo prendere un altro indirizzo IP (quindi un altro netid) e creare una nuova rete, connettere le due tramite un router e compilare debitamente la tabella del router. Grazie al proxy ARP invece, impostiamo lo stesso netid per le due reti. Come può funzionare? Il router R sa quali host appartengono all'Hidden Network (H_4 e H_5) e risponderà alle richieste ARP di H_1 , H_2 ed H_3 fingendosi l'host a cui il richiedente sta parlando. Ovviamente R svolgerà la stessa funzione speculare per gli host sulla rete nascosta.

Esempio di comunicazione

Supponiamo che H_1 voglia parlare con H_4 . H_1 farà il match del netid con H_4 e noterà che sono uguali: H_1 si illude che H_4 sia sulla sua stessa rete. Quindi, fa una richiesta ARP per ottenere la risoluzione dell'indirizzo IP in indirizzo fisico.

A questo punto sarà il router a giocare la sua parte: essendo collegato al Main Network riceverà la richiesta ARP e invierà una risposta, fornendo come indirizzo il suo indirizzo fisico: si sta quindi fingendo H₄.

H₁, ignaro che sia stato il router a rispondergli, prenderà l'indirizzo ricevuto come quello fisico di H₄ inserendolo nella sua cache ARP. A questo punto invierà i pacchetti al router credendo che questo sia H₄ ed il router li inoltrerà ad H₄.

Tutta questione di fiducia

Il proxy ARP si basa sul concetto di fiducia: gli host devono fidarsi di ciò che ricevono come risposta all'ARP request. Tecnicamente, una macchina potrebbe sapere se si trova coinvolta in una rete che fa uso di proxy ARP controllando la sua cache ARP: se più entry corrispondono ad uno stesso indirizzo fisico, quello sarà l'indirizzo del router che fa proxy. Quindi le macchine non devono effettuare questo controllo per far sì che il sistema di proxy ARP funzioni e non venga bloccato.

Si noti che la situazione in cui una macchina fa finta di essere un'altra, detta **spoofing**, è un problema di sicurezza non da poco. Perciò se si vuole usare proxy ARP si devono evitare blocchi per lo spoofing dato che la finzione è la base di questo meccanismo.

Si noti che **i router Cisco effettuano proxy ARP di default** (informazione utile in fase di testing delle mask come vedremo dopo).

Il proxy ARP è andato bene come palliativo ma la limitazione di poter coinvolgere solo due reti e di costringere l'amministratore di rete in una configurazione manuale sono problemi che ne hanno impedito lo sviluppo su larga scala.

5.3) Il subnetting

Vediamo una tecnica diversa per risolvere il problema dei pochi indirizzi IP.

Reminder: l'indirizzamento a classi

Ricordiamo come funziona l'indirizzamento a classi introducendo il concetto di mask.

L'indirizzo IP è suddiviso in una parte di netid ed una parte di hostid. A seconda di dove si trova la "linea" separatrice fra le due parti siamo in grado di avere più network o più host. Ogni classe definisce una separazione diversa.

Come sappiamo un host prima di inviare un pacchetto verifica se il destinatario si trovi o meno sulla sua stessa rete confrontando il proprio netid con quello del destinatario: ma come fa l'host a sapere il netid di un certo indirizzo IP?

Viene applicata la **mask** ovvero un indirizzo che avrà tanti 1 quanti sono i bit di netid: effettuando quindi un AND a livello macchina fra l'indirizzo IP e la mask otterremo il netid (affinché l'host lo possa confrontare col proprio).

Non abbiamo mai usato un granché la mask poiché le classi fornivano la loro definizione sfruttando i primi bit del netid (che erano fissi) e quindi non era necessaria la mask. Ma d'ora in poi sarà indispensabile.

- Reti di classe A

- Range del primo ottetto [1 - 126].
- Il netid inizia con un bit 0.
- Network mask con i primi 8 bit ad 1 (scritta come /8 o in dot notation 255.0.0.0).
- Reti con indirizzi da 1.0.0.0 a 126.0.0.0, ognuna con 16.777.214 host.

- Reti di classe B

- Range del primo ottetto [128 - 191].
- Il netid inizia con due bit 10.
- Network mask con i primi 16 bit ad 1 (scritta come /16 o in dot notation 255.255.0.0).
- Reti con indirizzi da 128.0.0.0 a 191.0.0.0, ognuna con 65.534 host.

- Reti di classe C

- Range del primo ottetto [192 - 223].
- Il netid inizia con due bit 110.
- Network mask con i primi 24 bit ad 1 (scritta come /24 o in dot notation 255.255.255.0).
- Reti con indirizzi da 192.0.0.0 a 223.255.255.0, ognuna con 254 host.

- Reti di classe D

- Range del primo ottetto [224 - 239].
- Il netid inizia con due bit 1110.
- Indirizzi di multicast.

- Reti di classe E

- Classe sperimentale per usi futuri.

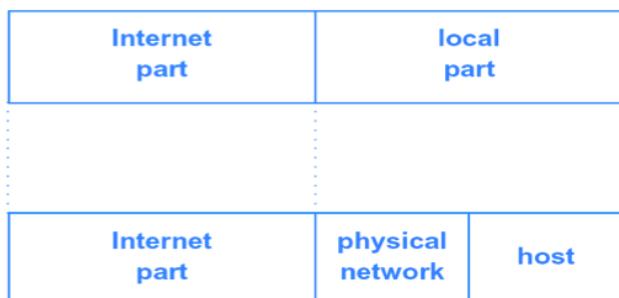
- Indirizzi riservati

- 0.0.0.0 indirizzo di startup.
- Indirizzi che iniziano con 127, sono di loopback.

Gli indirizzi di sottorete

Adottiamo l'idea del sito di cui abbiamo parlato prima. Vogliamo che all'interno di un sito gli indirizzi IP siano configurabili a piacere senza alcuna limitazione questo purché gli host siano d'accordo nel rispettare quello schema di indirizzamento.

Il subnetting si basa su uno sfruttamento più attento della **mask**.



In alto vediamo l'indirizzo IP così come è visto dall'esterno, cioè una parte netid ed una parte hostid.

Ma con il subnetting noi decidiamo di sfruttare una parte dell'hostid per creare un'ulteriore suddivisione all'interno della rete (immagine in basso).

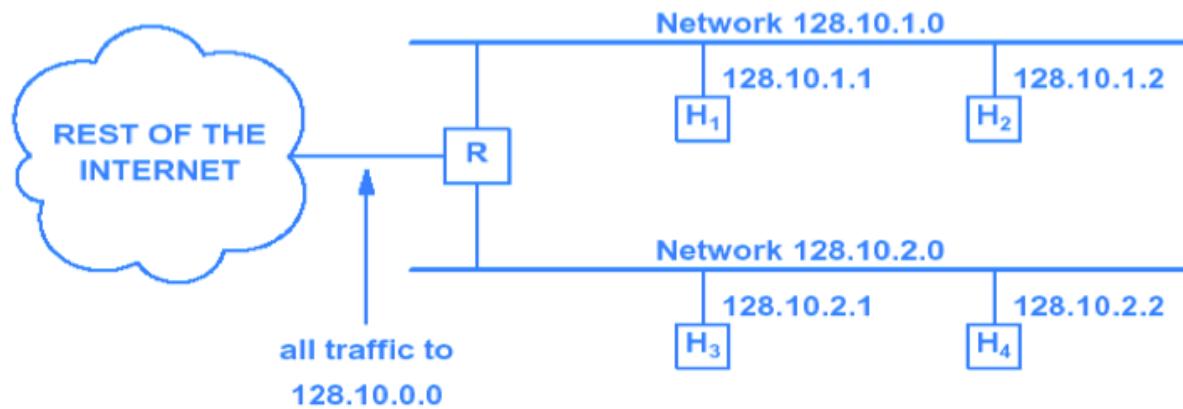
Esempi di subnetting

Ad esempio un indirizzo di classe A 10.0.0.0 con mask 255.0.0.0 può avere delle nuove sottoreti se cambiassimo la sua subnet mask in 255.255.0.0, infatti, le nuove sottoreti sarebbero 10.1.0.0, 10.2.0.0, ..., 10.255.0.0 con 65.534 host per ogni sottorete.

Ovviamente possiamo creare subnet anche su indirizzi di classe B, infatti, se avessimo un indirizzo 172.16.0.0 con mask 255.255.0.0 potremmo impostare la mask a 255.255.255.0 e ottenere così sottoreti: 172.16.1.0, 172.16.2.0, ecc. con 254 host ogni sottorete.

Vediamo un esempio in modo più dettagliato.

Abbiamo un indirizzo di classe B: 128.10.0.0 gli diamo una mask 255.255.255.0. Possiamo quindi avere 254 nuove sottoreti.



In questo caso ne abbiamo due, la 128.10.1.x (x sono gli host) e la 128.10.2.y (y sono gli host). Le due reti sono **fisicamente scollegate** ma sfruttano uno stesso indirizzo IP di classe B. Ecco il vantaggio!

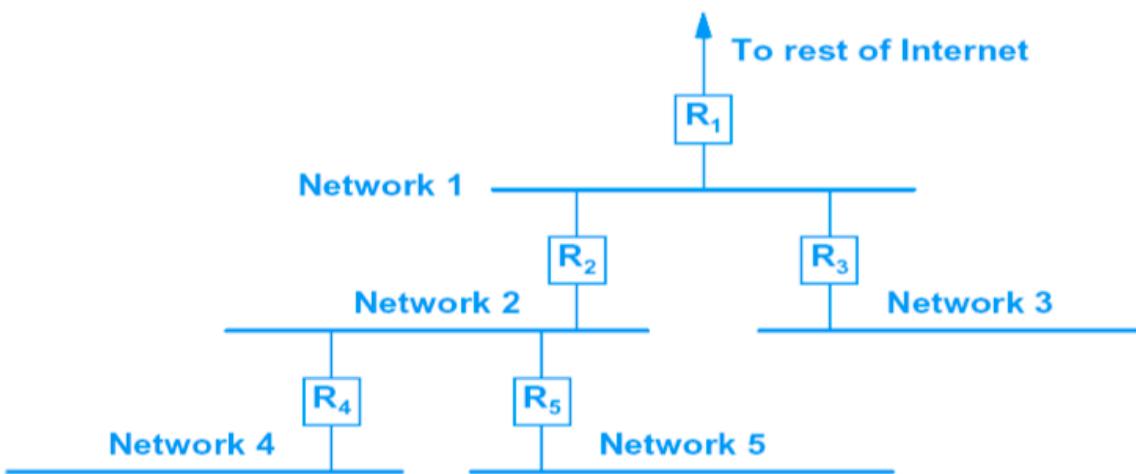
Dall'immagine vediamo come dall'esterno si vedano solamente IP 128.10.0.0 e questo ha un buon impatto sulle tabelle di routing dell'Internet.

Quando H_1 vorrà comunicare con H_3 userà la mask e noterà che c'è differenza fra i due netid+subnetid (su H_1 avremmo 128.10.1 e sull'indirizzo di H_3 avremmo 128.10.2) e quindi invieremo al router il quale si occuperà di instradare ad H_3 il pacchetto.

Si noti che anche gli host devono sapere del subnetting poiché devono instradare ad R i pacchetti diretti all'altra rete fisica, seppur logicamente uguale.

La flessibilità nell'assegnamento dell'indirizzamento a sottoreti

Non tutti i siti hanno le stesse esigenze e quindi la struttura dell'albero di assegnazione deve poter essere decisa localmente. Ad esempio scegliere in una classe B una suddivisione fra subnetid e hostid di 8+8 ci obbligherà ad avere al massimo 254 host per ognuna delle 254 reti possibili: ma se avessimo bisogno nel futuro di più reti di quelle disponibili? Viceversa, se dedicassimo tre bit per la parte di sottorete potremmo avere 6 reti con 8190 host... ma magari, potremmo aver bisogno di più reti nel futuro.



Possiamo quindi effettuare:

- **Subnetting a lunghezza fissa:** tutte le sottoreti del sito hanno la stessa dimensione, è il modo più semplice e che induce a non fare errori. Adeguato nella maggior parte dei casi.
- **Subnetting a lunghezza variabile:** in sostanza si applica la definizione di sottorete ricorsivamente: le sottosottoreti avranno una loro struttura a loro volta ma verranno visti dal resto del sito come sottorete fisica unica. Questa tecnica è altamente sconsigliata poiché induce facilmente in errore: uno stesso indirizzo può apparire abbinato a due sottoreti diverse.

Per i motivi descritti sopra, tendenzialmente si adopera la subnetting a lunghezza fissa in cui la mask è uguale per tutte le sottoreti: vantaggiosa dal punto di vista della facilità di mantenimento e svantaggiosa dal punto di vista della flessibilità.

Impostare le subnet mask e gli indirizzi non utilizzabili

Le subnet mask sono quindi fondamentali. Lo standard non guida in nessun modo l'assegnamento dei bit a 1, quindi volendo potremmo avere una mask come questa: 11111111 11111111 00011100 00001000 con degli 1 non contigui... ma la lettura di indirizzi fatti in questo modo è davvero pesante e complica anche l'assegnamento degli indirizzi di hosting e la compilazione delle tabelle di routing. Perciò, si considera come raccomandazione quasi obbligata **l'implementazione di maschere con bit a 1 contigui**.

Inoltre, come si sarà notato, la rappresentazione delle maschere in binario puro risulta parecchio. Per questo si adotta la dot notation o eventualmente una tripletta { <Network Number>, <Subnet Number>, <Host Number> } (ambigua, perché non indica con precisione quanto è lungo ogni componente).

A seconda dei bit che riserviamo alla subnet avremo più o meno subnet e più o meno host. Qui vediamo alcune soluzioni:

Subnet Bits	Number of Subnets	Hosts per Subnet
0	1	65534
2	2	16382
3	6	8190
4	14	4094
5	30	2046
6	62	1022
7	126	510
8	254	254
9	510	126
10	1022	62
11	2046	30
12	4094	14
13	8190	6
14	16382	2

Un occhio attento potrà notare che c'è qualcosa di strano. Perché con 2 bit abbiamo solamente due subnet? Teoricamente le combinazioni generabili con due bit sono quattro e quindi dovremmo avere quattro subnet! Non è così. L'**RFC 950** infatti proibisce l'uso degli indirizzi "tutti zero" e "tutti uno" (ecco quindi due possibilità in meno).

Questo perché l'indirizzo composto di "tutti zero" serve per identificare la sottorete stessa, mentre l'indirizzo con "tutti uno" è necessario per il broadcast sull'intera subnet.

Questa limitazione verrà poi tolta nel futuro, dall'**RFC 1812** che valuterà le due reti come "praticamente non pericolose". Infatti, l'unico vero problema poteva essere il broadcast, ma le richieste di broadcast prima di arrivare vengono certamente scartate dal router più esterno (poiché sono lette come broadcast su tutta la classe A B o C che sia!).

Esempio (1) di subnetting

Abbiamo un'azienda che ha un ip 193.1.1.0/24 (quindi una classe C).

Desidera definire **sei sottoreti** e la sottorete con maggior numeri di host deve avere **25 host al massimo**.

Definiamo prima di tutto la subnet mask. Per avere 6 sottoreti abbiamo bisogno di tre bit ($2^3 = 8$) e quindi avremo 5 bit rimanenti per gli host. La nostra mask sarà quindi **255.255.255.224**.

Ovvero, da questa mask: 11111111.11111111.11111111.00000000 (classe C)

otteniamo questa mask: 11111111.11111111.11111111.11100000 (tre bit per subnet id, 5 per gli hostid)

Perciò, avendo come indirizzo di base: $11000001.00000001.00000001.00000000 = 193.1.1.0/24$
avremo le seguenti **sottoreti**:

Subnet #0: $11000001.00000001.00000001.\textbf{00000000} = 193.1.1.0/27$

Subnet #1: $11000001.00000001.00000001.\textbf{00100000} = 193.1.1.32/27$

Subnet #2: $11000001.00000001.00000001.\textbf{01000000} = 193.1.1.64/27$

Subnet #3: $11000001.00000001.00000001.\textbf{01100000} = 193.1.1.96/27$

Subnet #4: $11000001.00000001.00000001.\textbf{10000000} = 193.1.1.128/27$

Subnet #5: $11000001.00000001.00000001.\textbf{10100000} = 193.1.1.160/27$

Subnet #6: $11000001.00000001.00000001.\textbf{11000000} = 193.1.1.192/27$

Subnet #7: $11000001.00000001.00000001.\textbf{11100000} = 193.1.1.224/27$

Allo stesso modo, definiremo invece gli host (per ogni sottorete!) ad esempio per la subnet #2:

Subnet #2: $11000001.00000001.00000001.010\textbf{00000} = 193.1.1.64/27$

Host #1: $11000001.00000001.00000001.010\textbf{00001} = 193.1.1.65/27$

Host #2: $11000001.00000001.00000001.010\textbf{00010} = 193.1.1.66/27$

Host #3: $11000001.00000001.00000001.010\textbf{00011} = 193.1.1.67/27$

Host #4: $11000001.00000001.00000001.010\textbf{00100} = 193.1.1.68/27$

Host #5: $11000001.00000001.00000001.010\textbf{00101} = 193.1.1.69/27$

...

Host #27: $11000001.00000001.00000001.010\textbf{11011} = 193.1.1.91/27$

Host #28: $11000001.00000001.00000001.010\textbf{11100} = 193.1.1.92/27$

Host #29: $11000001.00000001.00000001.010\textbf{11101} = 193.1.1.93/27$

Host #30: $11000001.00000001.00000001.010\textbf{11110} = 193.1.1.94/27$

Avremo poi un indirizzo di broadcast come per ogni sottorete: $11000001.00000001.00000001.010\textbf{11111} = 193.1.1.95$

Routing in presenza di sottoreti

È chiaro che la presenza delle sottoreti debba in qualche modo **cambiare l'algoritmo di routing**. Tutti gli host ed i router del sito devono ora adottare il **subnet routing**.

Come sempre, cercheremo di ottenere il cammino ottimo e questo non è così banale. Infatti:

- Il cammino ottimo cambia in presenza di **guasti hardware**.
- Il cammino ottimo cambia in presenza di **congestione**.
- L'indirizzamento su sottorete non considera i confini dei sistemi autonomi, si **ignora il problema di rendere note le maschere di sottorete**: i vecchi protocolli di routing non sapevano propagare il concetto di sottorete.

Per questi motivi i siti devono tenere il subnetting più semplice possibile e confinarlo il più possibile in una zona ristretta:

- **Tutte le sottoreti di una rete IP devono essere contigue** (non possiamo assegnare due host di una certa subnet in una rete fisica e poi altri due host della stessa rete fisica in un'altra subnet dislocata altrove).
- **Le maschere di sottorete devono essere uniformi su tutte le sottoreti**.
- **Tutte le macchine devono partecipare all'instradamento su sottorete**.

A parte queste linee guida, le tabelle di routing dovranno variare il loro contenuto. Dal <netid, indirizzo router> avremo ora <**subnet mask, indirizzo di rete, indirizzo router**>. All'arrivo di un pacchetto il router leggerà il destinatario e applicherà la subnet mask all'indirizzo per verificare la corrispondenza del netid+subnetid. Si noti che questo meccanismo è retrocompatibile per l'inoltro senza uso di subnet: semplicemente le mask saranno adeguate all'indirizzo di classe A, B o C.

Ultima questione a cui porre attenzione è il fatto che gli indirizzi diretti per gli host all'interno delle routing table dovranno essere messi a inizio tabella mentre i default gateway andranno messi al fondo (ovvero andiamo da indirizzi più specializzati a indirizzi meno specializzati).

Ecco l'algoritmo di routing:

Algorithm:

Forward_IP_Datagram (datagram, routing_table)

```

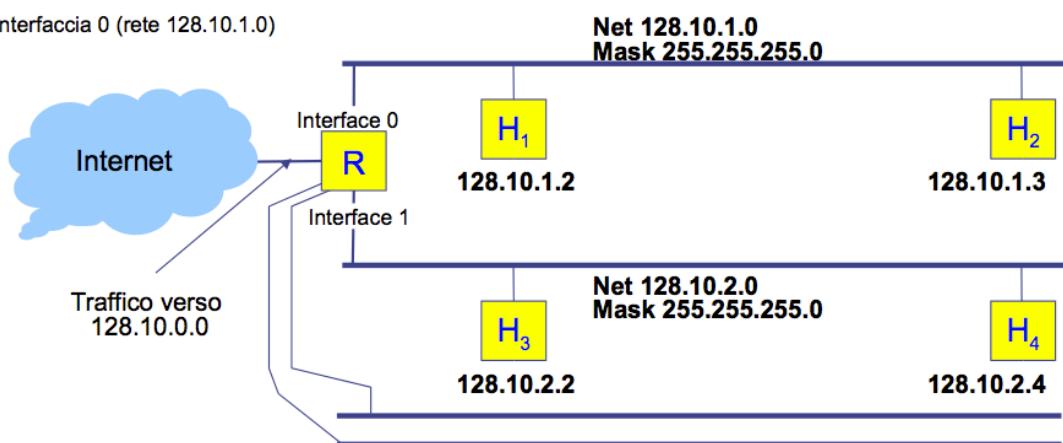
Extract destination IP address,  $I_D$ , from datagram;
If prefix of  $I_D$  matches address of any directly connected
network send datagram to destination over that network
(This involves resolving  $I_D$  to a physical address,
encapsulating the datagram, and sending the frame.)
else
    for each entry in routing table do
        Let N be the bitwise-and of  $I_D$  and the subnet mask
        If N equals the network address field of the entry then
            forward the datagram to the specified next hop address
        endforloop
    If no matches were found, declare a forwarding error;

```

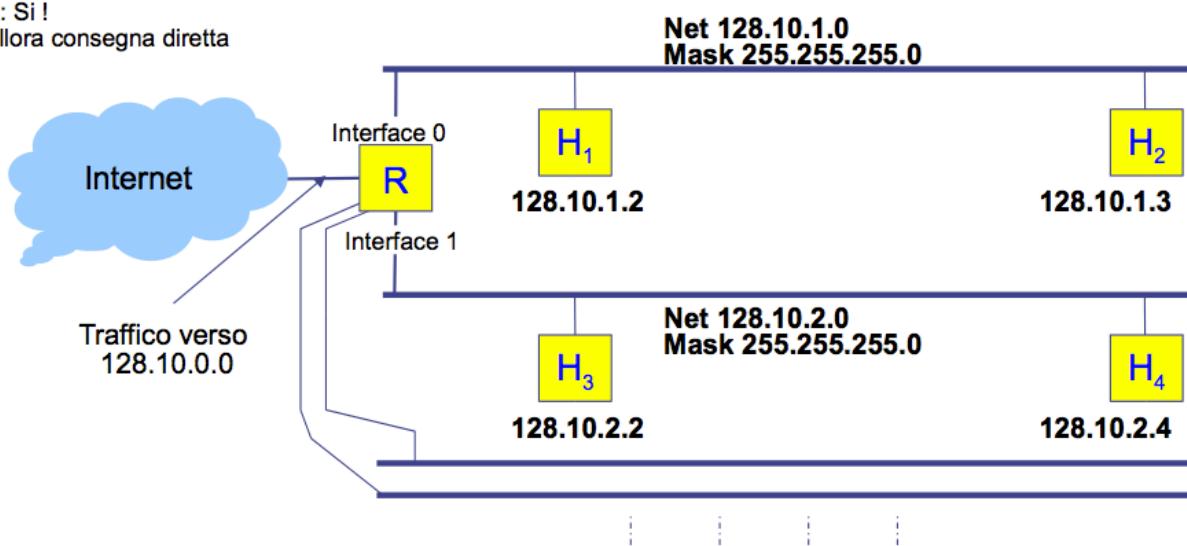
Esempio (1) di routing in presenza di sottoreti: pacchetto dall'esterno

A R arriva un datagram per
 128.10.1.3 b/ 10000000.00001010.00000001.00000011
 Netmask
 255.255.255.0 b/ 1111111.1111111.1111111.00000000
AND
 D: Il risultato è uguale ad una delle righe ?
 R: si alla prima
 Allora instrada sull'interfaccia 0 (rete 128.10.1.0)

Routing table for R		
Subnet Number	Subnet Mask	Next hop
128.10.1.0	255.255.255.0	Interface 0
128.10.2.0	255.255.255.0	Interface 1

Esempio (2) di routing in presenza di sottoreti: consegna diretta

A H₁ deve mandare un datagram per ad H₂
 128.10.1.3 b/ 10000000.00001010.00000001.00000011
 Netmask
 255.255.255.0 b/ 1111111.1111111.1111111.00000000
AND
 D: Il risultato è uguale al netid di H₁ (128.10.1.0) ?
 R: Si !
 Allora consegna diretta



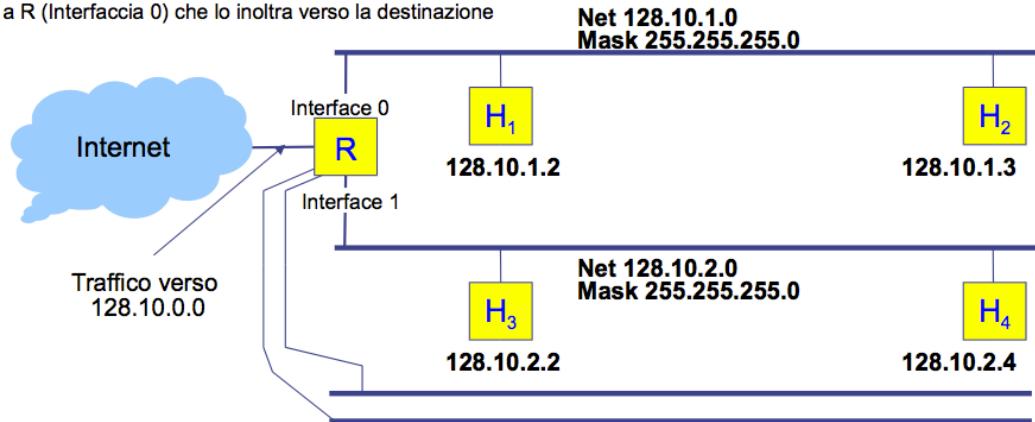
Esempio (3) di routing in presenza di sottoreti: consegna indiretta

Il default router sulla rete 128.10.1.0 è R (Interfaccia 0)
 A H₁ deve mandare un datagram per ad H₄
 128.10.2.4 b/ 10000000.00001010.00000010.00000100
 Netmask
 255.255.255.0 b/ 1111111.1111111.1111111.00000000

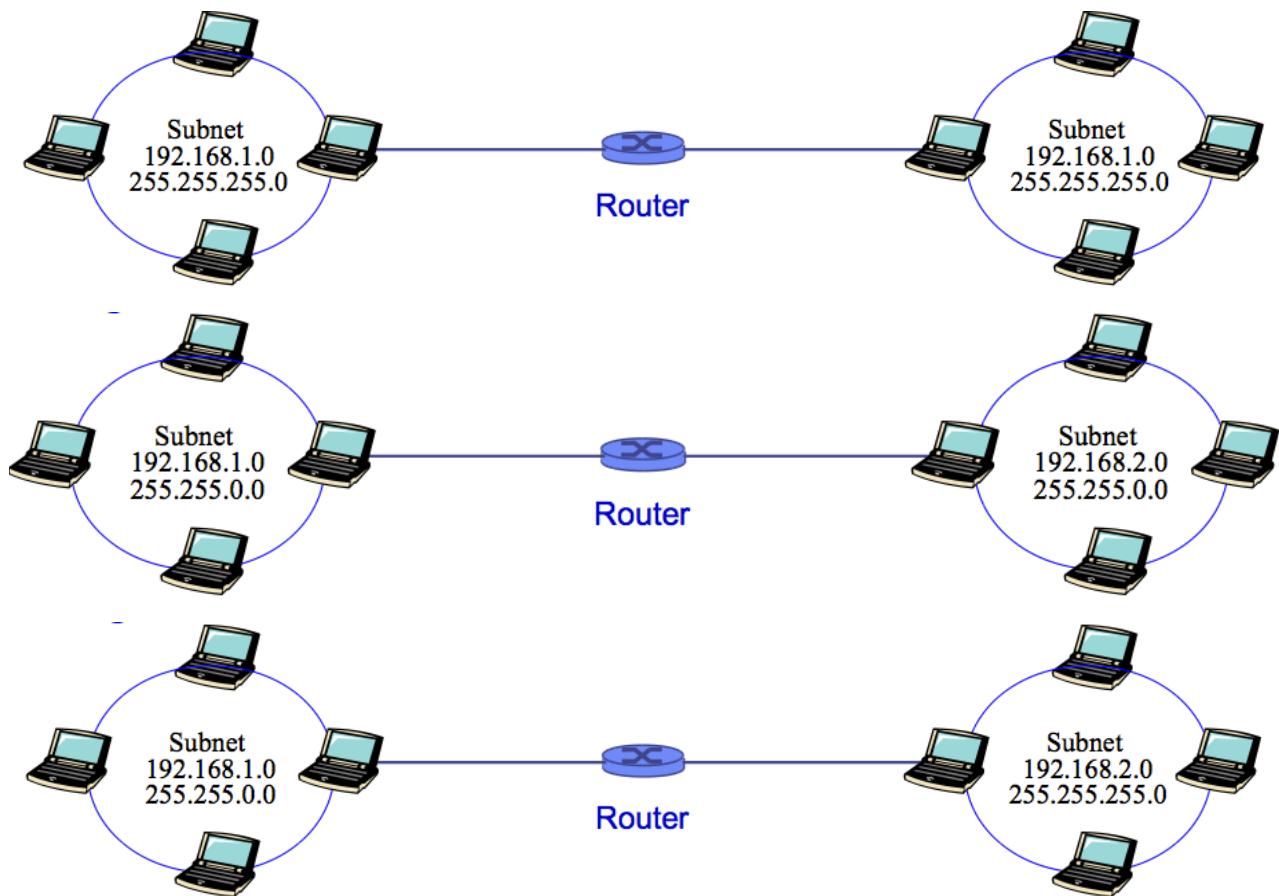
ANDD: Il risultato è al netid di H₁ (128.10.1.0) ?

R: No !

Allora invio del datagram a R (Interfaccia 0) che lo inoltra verso la destinazione

Esempi di configurazione non valida di mask

Alcuni esempi di configurazioni non valide per delle masks.



5.4) Il supernetting

Tutti i problemi di cui abbiamo parlato fin ora hanno, storicamente, trovato soluzione grazie al subnetting. Ma la miglioria di cui stiamo per parlare è stata a dir poco rivoluzionaria ed ha permesso alla rete Internet di non collassare anni fa.

Il supernetting (anche detto CIDR), consiste, sostanzialmente, nel **raggruppare le reti**. Questo tipo di funzionalità è stata molto utile per permettere l'avvento degli ISP e per far sì che le reti classe C non venissero più sprecate.

Per esempio due reti con indirizzi di classe C contigui 192.168.32.0, 192.168.33.0 possono essere unificate in una rete unica 192.168.32.0/23! Vediamo nel dettaglio di cosa stiamo parlando.

Supernet: una sola rete per controllarle tutte

Il sistema a classi ci è troppo stretto, vogliamo poter parlare di **blocchi**. Quindi vogliamo che le tabelle di instradamento dei router crescano in relazione ai **blocchi assegnati** e non più alle classi C assegnate.

Un blocco è un insieme di reti, ed è specificato dalla coppia (**network address, count**) dove network address è il più piccolo indirizzo del blocco e count è il numero di reti di classe C incluse nel blocco stesso.

Quindi la coppia (192.5.48.0, 3) identifica il blocco costituito dalle reti di classe C: 192.5.48.0, 192.5.49.0, 192.5.50.0.

Immaginiamo delle entità particolari, gli ISP, che sono i "fornitori della rete": questi, essendo pochi, potrebbero acquistare dei blocchi di indirizzi enormi e così facendo le tabelle di routing fra gli ISP sarebbero di ridotte dimensioni! Capiamo bene quanto sia **fondamentale che le reti siano contigue**.

Supernet: oltre il concetto di classe

Ma perché limitarsi a concedere multipli di classi C? Potremmo pensare di assegnare un range **arbitrario** purché la dimensione dei blocco sia in potenza di due. Sfrutteremo al massimo la bit mask: indicherà i bit che restano costanti dall'indirizzo minimo a quello massimo (per questo la dimensione risulta essere potenza di due).

Ci correrà quindi in aiuto, ancora una volta, la notazione **slash notation** già usata per le subnet. Quindi i bit a 1 saranno a partire da sinistra e saranno contigui. All'aumentare il numero indicato a fianco della slash (cioè il numero di bit a 1) diminuiremo il numero di indirizzi appartenenti al blocco.

Per ottenere il numero di indirizzi utilizzabili data un certo indirizzo espresso in slash notation è sufficiente calcolare $x = 32$ (tutti i bit dell'indirizzo IP) - numero a fianco della slash. Non resta che calcolare 2^x per avere quanti indirizzi sono utilizzabili. Questo calcolo è banale, stiamo in pratica contando quanti bit non a 1 abbiamo e quindi ne consideriamo le combinazioni (2 alla numero di bit).

Primi esempi di supernet e lista di mask CIDR

Ad esempio un range di 2048 indirizzi:

	Dotted Decimal	32-bit Binary Equivalent
lowest	128.211.168.0	10000000 11010011 10101000 00000000
highest	128.211.175.255	10000000 11010011 10101111 11111111

Abbiamo questa supernet che ha come indirizzo più basso 128.211.168.0 e come più grande 128.211.175.255. La sua mask sarà quindi di 21 bit uguali a 1, 1111111.1111111.1111000.0000000 infatti indicheremo l'indirizzo come 128.211.168.0/21 che col calcolo detto prima ci da $(32 - 21) = 11$, $2^{11} = 2048$ reti.

Oppure, in questo esempio:

	Dotted Decimal	32-bit Binary Equivalent
lowest	128.211.176.212	10000000 11010011 10110000 11010100
highest	128.211.176.215	10000000 11010011 10110000 11010111

Avremo una mask di 30 bit ad 1, quindi 128.211.176.212/30 (che ci da $2^2 = 4$ reti).

Ecco una lista delle mask CIDR in dotted quad:

CIDR Notation	Dotted Decimal	CIDR Notation	Dotted Decimal
/1	128.0.0.0	/17	255.255.128.0
/2	192.0.0.0	/18	255.255.192.0
/3	224.0.0.0	/19	255.255.224.0
/4	240.0.0.0	/20	255.255.240.0
/5	248.0.0.0	/21	255.255.248.0
/6	252.0.0.0	/22	255.255.252.0
/7	254.0.0.0	/23	255.255.254.0
Classe A		/24	255.255.255.0
/8	255.0.0.0	/25	255.255.255.128
/9	255.128.0.0	/26	255.255.255.192
/10	255.192.0.0	/27	255.255.255.224
/11	255.224.0.0	/28	255.255.255.240
/12	255.240.0.0	/29	255.255.255.248
/13	255.248.0.0	/30	255.255.255.252
/14	255.252.0.0	/31	255.255.255.254
/15	255.254.0.0	/32	255.255.255.255
Classe B			
/16	255.255.0.0		

Un ulteriore esempio:

Un ISP di proprietà di mr. Polly riceve un blocco di indirizzi 210.20.128.0/17 (cioè 32.768 indirizzi).

Decide di creare 128 reti da 256 indirizzi ciascuna (cioè delle classi C):

- 210.20.128.0/24
- 210.20.129.0/24
- ...

La maschera di rete per ogni sottorete sarà 255.255.255 mentre la mask del blocco è 255.255.128!

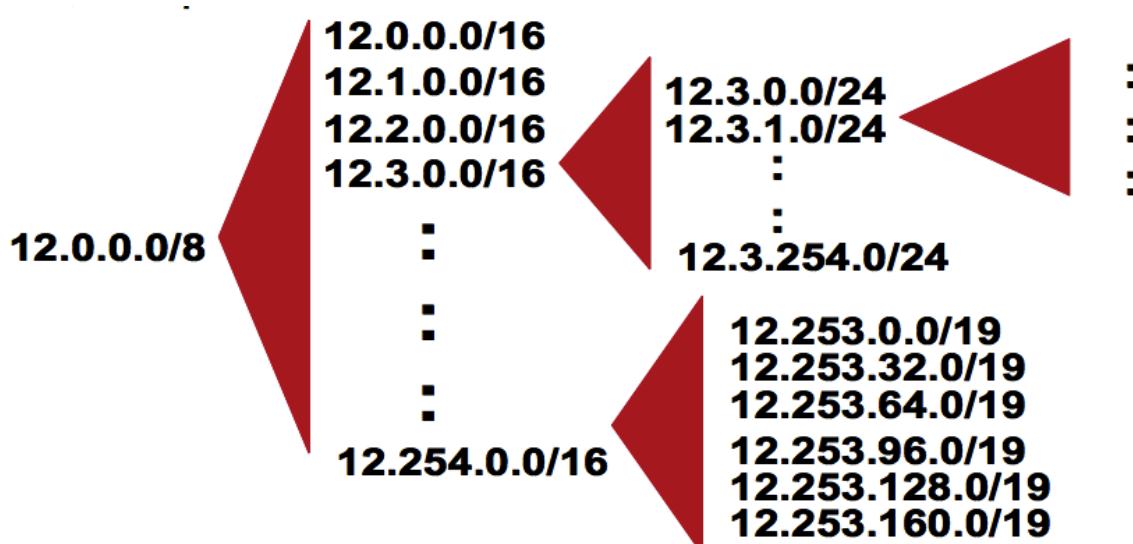
Da questo esempio capiamo come le mask siano "incrementali", nozione che ci sarà utile dopo.

Strutture dati e algoritmi per la ricerca di indirizzi senza classi

Come sempre, il nostro obiettivo è la velocità/efficienza.

La rapidità nel trovare il next hop purtroppo ha subito un colpo: gli indirizzi CIDR non sono più **auto-identificanti** come erano le classi e quindi è necessario l'utilizzo costante delle mask per ottenere il netid.

Nelle tabelle di routing applicheremo un principio di specificità ovvero gli indirizzi con mask più lunghe verranno messi a inizio tabella mentre quelli meno precisi saranno più in fondo.



Quanto detto sopra è abbastanza ovvio: se dobbiamo arrivare a Torino sarà più interessante l'informazione "vai in Piemonte" piuttosto che "vai in Italia".

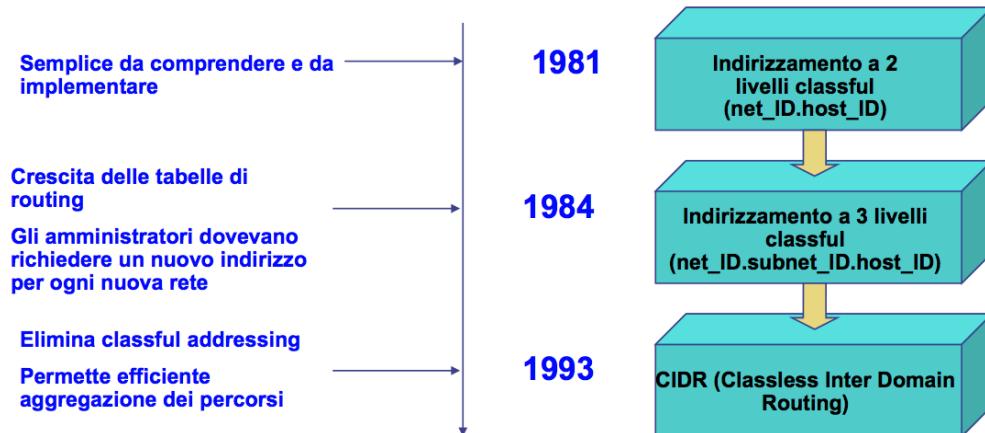
Per esempio, avendo come destinazione 11.1.2.5 (come indicato sotto) sceglieremo la route più precisa, cioè per la quale corrispondono più bit (11.1.2.0).

Destination	11.1.2.5	= 00001011.00000001.00000010.00000101
* Route #1	11.1.2.0/24	= 00001011.00000001.00000010.00000000
Route #2	11.1.0.0/16	= 00001011.00000001.00000000.00000000
Route #3	11.0.0.0/8	= 00001011.00000000.00000000.00000000

L'algoritmo che effettua questa operazione testerà prima se c'è una corrispondenza esatta con 32 bit, poi se ne esiste una a 31, poi a 30, ecc.

Questo algoritmo è molto inefficiente per questo vengono adottate strutture ad hoc (alberi binari) per minimizzare il costo computazionale.

Evoluzione dell'addressing



Altri esempi di supernetting

IP Address: 192.168.0.4

Subnet Mask: /31 (255.255.255.254)

2 indirizzi IP: 192.168.0.4 e 192.168.0.5. L'indirizzo 192.168.0.4/31 è il primo indirizzo della rete

IP Address: 192.168.0.4

Subnet Mask: /30 (255.255.255.252)

4 indirizzi IP: da 192.168.0.4 a 192.168.0.7. L'indirizzo 192.168.0.4/30 è il primo indirizzo della rete

IP Address: 192.168.0.4

Subnet Mask: /29 (255.255.255.248)

8 indirizzi IP: da 192.168.0.4 a 192.168.0.7. L'indirizzo 192.168.0.4/29 è il primo indirizzo della rete

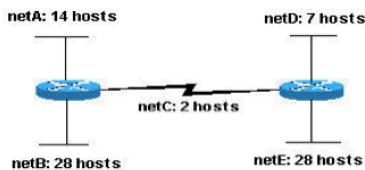
IP Address: 192.168.0.4

Subnet Mask: /32 (255.255.255.255)

1 solo indirizzo 192.168.0.4. Primo ed ultimo della rete

Le migliori ottenute grazie al supernetting

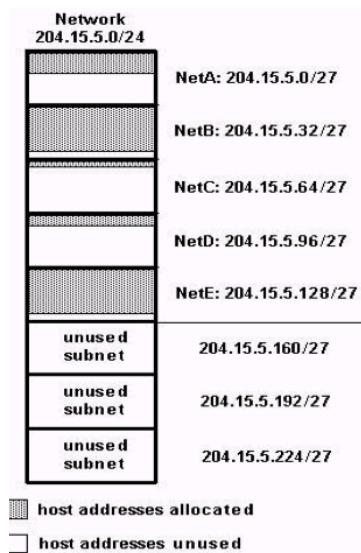
Abbiamo questa situazione:



- Rete di classe C 204.15.5.0/24
- Soluzione con maschere a lunghezza fissa
- $2^5 = 32$ host (30 usabili)

Subnet	Mask	Host Range	Broadcast
NetA: 204.15.5.0	255.255.255.224	204.15.5.1 to 204.15.5.30	204.15.5.31
NetB: 204.15.5.32	255.255.255.224	204.15.5.33 to 204.15.5.62	204.15.5.63
NetC: 204.15.5.64	255.255.255.224	204.15.5.65 to 204.15.5.94	204.15.5.95
NetD: 204.15.5.96	255.255.255.224	204.15.5.97 to 204.15.5.126	204.15.5.127
NetE: 204.15.5.128	255.255.255.224	204.15.5.129 to 204.15.5.158	204.15.5.159

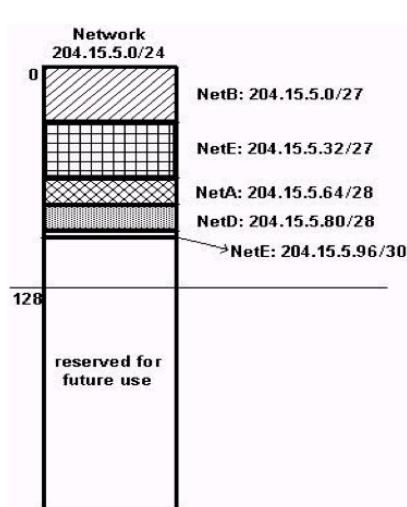
Potendo solamente usare maschere a lunghezza fissa, otterremmo quindi i seguenti sprechi (immagine a destra). A sinistra invece il calcolo se volessimo usare delle maschere diverse:

**NetA e soprattutto NetC sprecano indirizzi**

netA: must support 14 hosts
 netB: must support 28 hosts
 netC: must support 2 hosts
 netD: must support 7 hosts
 netE: must support 28 host
 netA: /28 (255.255.255.240) to support 14 hosts
 netB: /27 (255.255.255.224) to support 28 hosts
 netC: /30 (255.255.255.252) to support 2 hosts
 netD*: /28 (255.255.255.240) to support 7 hosts
 netE: /27 (255.255.255.224) to support 28 hosts

* a /29 (255.255.255.248) would only allow 6 usable host addresses therefore netD requires a /28 mask.

Vediamo come le maschere diverse aumentano lo spazio utilizzabile!

**Il modo più semplice è di assegnare prima le sottoreti con più host**

netB: 204.15.5.0/27 host address range 1 to 30
 netE: 204.15.5.32/27 host address range 33 to 62
 netA: 204.15.5.64/28 host address range 65 to 78
 netD: 204.15.5.80/28 host address range 81 to 94
 netC: 204.15.5.96/30 host address range 97 to 98

Reti di Elaboratori

Il livello di trasporto

Capitolo 6

Enrico Mensa,

Basato sulle lezioni del prof. [Matteo Sereno](#)

Il livello di trasporto

1) La comunicazione fra le applicazioni

1

1.1) Un nuovo concetto di comunicazione

1.2) Servizi forniti dal livello transport

1.3) Primitive del livello di trasporto [connection-oriented]

La richiesta di connessione: un'approccio differente

I messaggi scambiati

1.3) Identificare un'applicazione: le porte

1.4) Stabilire una connessione

Con segmenti duplicati/ritardati

1.5) Chiudere una connessione

Il problema dei due eserciti

1.6) Controllo degli errori e controllo di flusso

La differenza fra lavorare a livello transport ed a livello data link

L'allocazione dei buffer

I colli di bottiglia

1.7) Ripristino dopo un crash

1.8) Controllo della congestione

Allocare la banda staticamente: max-min fairness

Allocare la banda dinamicamente: AIMD

Regolare i tassi di invio

TCP e le reti Wifi

2) Il protocollo UDP

12

2.1) Le caratteristiche basilari

2.2) Il segmento UDP

Un particolare checksum: lo pseudo-header UDP

2.3) Le porte

Due tipologie di porte

2.4) Le remote procedure call (RPC)

Gli stub

Operazioni idempotenti

2.5) Trasporto real-time (protocollo RTP)

Header RTP

Protocollo RTCP

3) Il protocollo TCP

16

3.1) Realizzare l'affidabilità

Stop & wait: uno scarso utilizzo della banda

L'idea della finestra scorrevole

I numeri di sequenza: una precisazione

Probing e finestra chiusa

Migliorare l'efficienza della sliding window: algoritmo di Neagle

La silly window syndrome

3.2) Porte, endpoint e connessioni

Alcuni esempi di endpoint/connessioni

Apertura della connessione (attiva e passiva)

Le well-known ports

3.3) Il modello di servizio (socket)

Messaggi urgenti: PUSH e URGENT

3.4) Il segmento TCP

I campi del segmento

I riscontri in TCP

Un particolare checksum: lo pseudo-header TCP

3.5) Setup di una connessione in TCP

Il problema del syn flood

3.6) Rilascio di una connessione in TCP

3.7) TCP come macchina a stati

3.8) La gestione dei timer

I timer di TCP

Il timer RTO

3.9) La gestione della congestione

Le cause della congestione

La collaborazione di TCP

La finestra di congestione

Percepire la congestione

Lo slow-start: non accontentarsi di AIMD

Fast-retransmission: ACK ripetuti, un segnale di congestione

Fast-recovery: l'ultima ottimizzazione

3.10) Problemi di prestazioni nelle reti

Protocolli per reti ad alta capacità

4) Il DNS

30

4.1) Gli obiettivi del DNS

4.2) I nomi

Spazio dei nomi piatto

Spazio dei nomi piatto globale

Nomi gerarchici

4.2) Internet Domain Names

Nomi di dominio Internet (ufficiali/non ufficiali)

4.3) Nomi e proprietà

Lista dei tipi di resource

4.4) Il mapping fra nomi ed elementi informativi

DNS: un'applicazione client-server

L'albero dei name server

4.5) Risolvere un nome di dominio (primo approccio e concetti)

Due modi di risolvere: traduzione completa e traduzione incompleta

Ottenerne il primo name server da cui iniziare la ricerca

Le informazioni a disposizione di un name server

4.6) Risolvere un nome di dominio (primo passo d'efficienza: bottom up)

4.7) Risolvere un nome di dominio (la soluzione definitiva: il name caching)

Come avviene la risoluzione

La durata della cache

4.8) Il formato dei messaggi DNS

Formato compresso dei nomi e abbreviazione dei nomi di dominio

Il mapping inverso (query PTR)

4.9) Resources ed esempi

Risorse A

Risorse CNAME

Risorse HINFO e MINFO

Risorse MX

Risorse NS

Risorse PTR

Risorse SOA

4.10) Richieste al DNS: server primari e server secondari

4.11) DNS: UDP o TCP?

4.12) Ottenerne l'autorità per un sottodomainio

Studio delle tracce Wireshark

45

5.1) Richieste generiche

- Ping (request)
- Traceroute (sequenza di ping)
- Il framing

5.2) ARP

- ARP request
- ARP reply

5.3) DNS

- Richiesta NS

I socket

1) Introduzione ai socket

49

1.1) Le funzionalità delle interfacce

1.2) Creare un socket

- Creare un endpoint

1.3) System call server side

- bind()
- listen()
- accept()

1.4) System call client side

- connect()
- send()
- sendto() e sendmsg()
- recv()
- recvfrom() e recvmsg()
- close() e shutdown()
- Interfacciarsi col DNS: gethostbyname() e gethostbyaddr()
- Impostare parametri dei socket: setsockopt() e getsockopt()
- Convertire gli indirizzi: inet_ntoa() e inet_addr
- select(): attesa non deterministica
- Esempio di utilizzo delle system call: Client
- Esempio di utilizzo delle system call: Server

Il livello di trasporto

1) La comunicazione fra le applicazioni

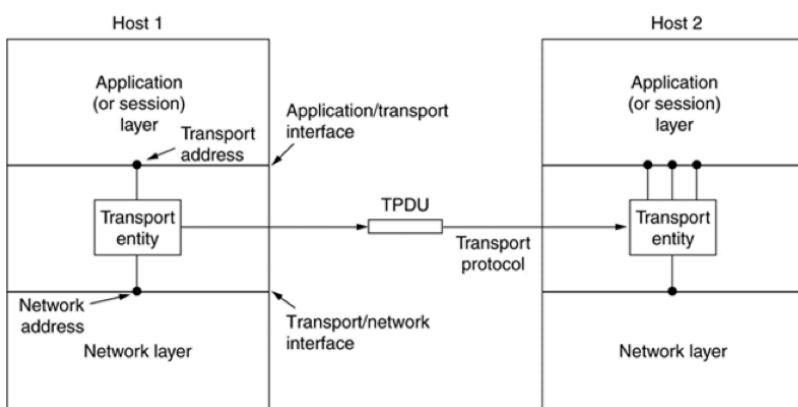
Insieme al livello di rete (network) il livello di trasporto è **il cuore della gerarchia dei protocolli di Internet**, che, non per altro, è chiamato stack TCP/IP (TCP è un protocollo di livello transport).

Qual è il nostro obiettivo? Vogliamo fornire la **comunicazione fra due processi** di due macchine diverse connesse alla rete. Fino ad ora abbiamo sempre parlato di connessione fra host ma capiamo bene che questo non può bastare: ogni macchina ha più processi concorrenti in esecuzione!

In ogni caso, anche se una macchina fosse “mono task” il problema sussisterebbe ugualmente poiché ogni processo può sfruttare un protocollo diverso di livello transport. Dobbiamo quindi risolvere (ancora una volta, come già fatto nei livelli precedenti) il problema del multiplexing/demultiplexing dei messaggi destinati ad entità differenti.

1.1) Un nuovo concetto di comunicazione

La comunicazione che vogliamo fornire è ovviamente appoggiata al network ma è concettualmente distante da quella studiata fin ora: infatti mentre prima avevamo una comunicazione da host mittente a router, da router a router e quindi da router a host destinatario (cioè il livello di rete era implementato anche sulle macchine intermedie, i router, appunto) questa abbiamo una connessione **end to end**: il livello transport è **implementato solamente sugli host**.



Come vediamo l'host 1 comunica tramite il suo livello transport con l'host 2 appoggiandosi al livello di network.
Tra i due vengono scambiati i messaggi del livello transport che si chiamano **TPDU** (*transport protocol data unit*).

1.2) Servizi forniti dal livello transport

Il livello di transport offre sostanzialmente la possibilità a due applicazioni di poter comunicare. Questo può accadere con un servizio di trasporto connection-less (chiamato **UDP**) che è sostanzialmente ciò che offre IP più, ovviamente, la parte che permette la comunicazione fra due applicazioni.

Esiste anche un servizio di trasporto connection-oriented (chiamato **TCP**) che offre molte altre funzionalità (affidabilità, gestione di flusso, gestione degli errori, gestione dei ritardi, ecc.). Se ci pensiamo TCP è una miglioria incredibile poiché è basato su un servizio che non è affidabile (il livello di rete).

Una grande differenza fra il livello di rete e quello di transport è che il secondo è **accessibile agli utenti**, che, tramite interfacce (una delle quali sono i socket di cui parleremo in seguito) possono “lavorare con la rete” e rendere quindi le loro applicazioni dei servizi connessi in rete.

1.3) Primitive del livello di trasporto [connection-oriented]

Nel caso di TCP disponiamo di alcune primitive che sono, come detto sopra, utilizzabili dalle applicazioni tramite un'interfaccia.

Ecco la lista delle primitive con relativa spiegazione:

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

Nota bene: le primitive hanno nomi simili a quelli che riscontreremo nell'interfaccia socket. Ma questo è un caso, o meglio, sono i socket che hanno attinto dalle primitive e non il viceversa (ovviamente, poiché i socket sono un'interfaccia per usare le primitive e non il contrario).

La richiesta di connessione: un'approccio differente

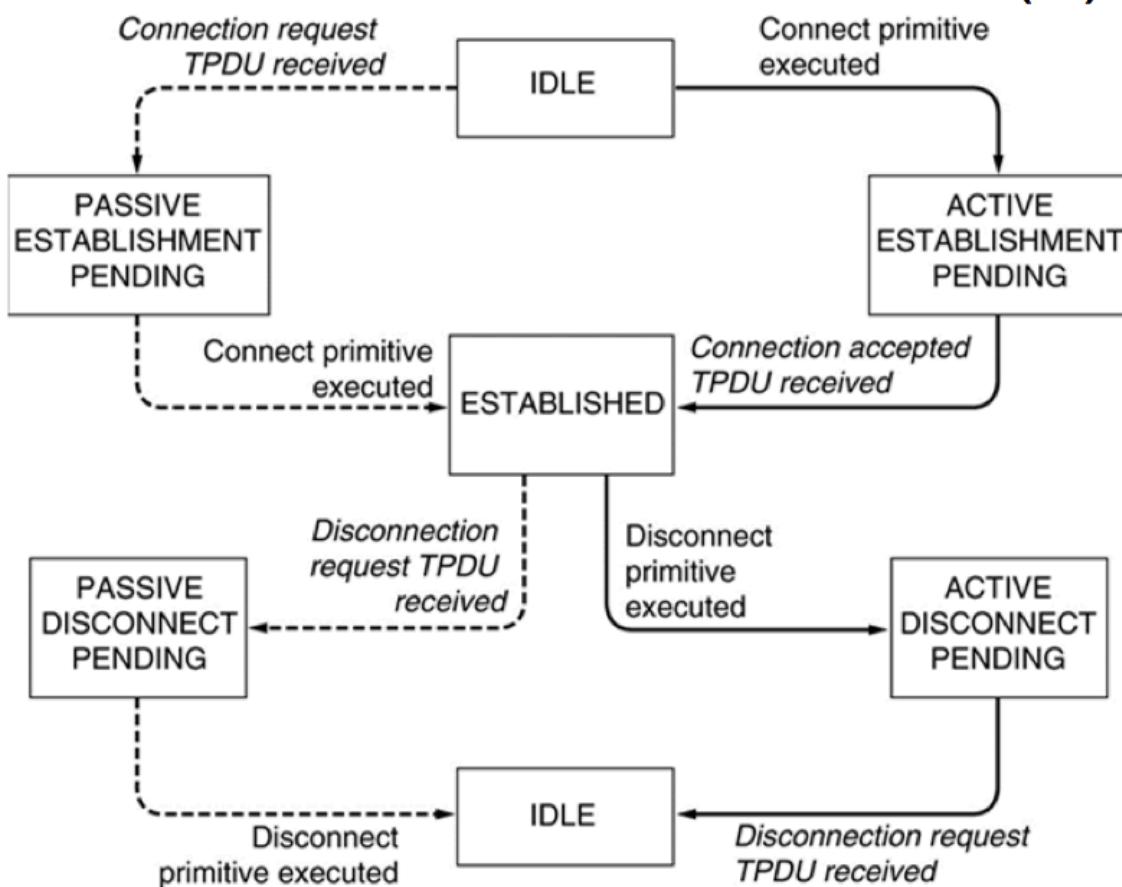
Facciamo una metafora fra la linea telefonica e la rete Internet.

Quando una persona ne chiama un'altra, il destinatario riceve una sorta di interrupt (la suoneria) e quindi va a rispondere. Questa è una **richiesta di sincronizzazione**. Entrambi gli umani vivono le loro vite, poi, a un certo punto, uno interrompe l'altro e si sincronizzano per comunicare.

Nel caso dei calcolatori non accade così: abbiamo infatti l'architettura **client-server** che modifica totalmente l'idea (anche se non si direbbe). Abbiamo infatti un'entità che è sempre ad aspettare (il server) e una che invece chiede la connessione. Abbiamo un paradigma asimmetrico!

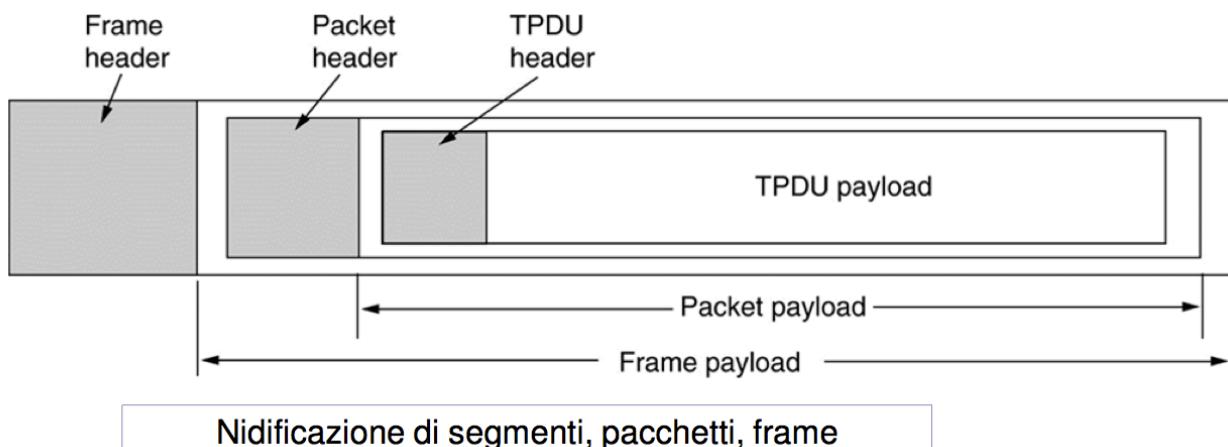
Vediamo infatti fra le primitive la possibilità di richiedere una connessione (e anche di toglierla, liberando le risorse allocate per effettuare la comunicazione).

Ecco quindi come si svolge un ciclo di connessione:



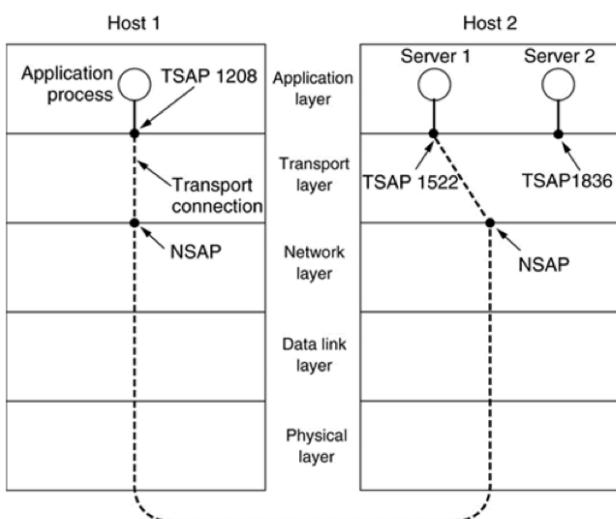
I messaggi scambiati

I messaggi transport sono, come gli altri, incapsulati all'interno dei messaggi dei livelli sottostanti. Come già detto il messaggio "generico" di un livello transport è chiamato TPDU ma nel caso di TCP parleremo di **segmenti**, nel caso di UDP (con scarsa fantasia) parleremo di **datagram UDP**.

**1.3) Identificare un'applicazione: le porte**

Come possiamo, dunque, identificare una singola applicazione su una macchina? Per arrivare alla macchina abbiamo l'indirizzo IP mentre per identificare l'applicazione sfrutteremo la **porta**. Il termine "porta" è usato solamente nello stack Internet, esse sono infatti genericamente chiamate **TASP** (transport service access point) mentre i punti di accesso al livello network sono detti **NASP** (network service access point).

Quindi quando un host vuole stabilire una connessione dovrà inviare una request ad un indirizzo IP fornendo anche la porta sulla quale vuole comunicare.



Una coppia <IP, porta> è detta **endpoint**. Come può un client generico conoscere come sono distribuiti i servizi (le applicazioni cioè) sulla macchina server? Si adottano delle **porte standard** per i servizi più famosi (e per gli altri è possibile specificare la porta). Ad esempio:

- http, porta 80
- ftp, porta 21
- client di posta, porta 25
- ssh, porta 22

Chiaramente non è possibile adottare la stessa porta per due applicazioni differenti.

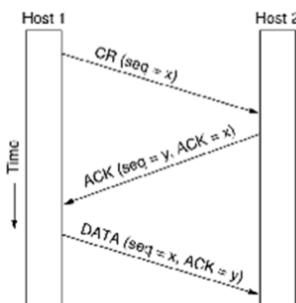
Come vediamo dall'immagine sopra, **anche il client dovrà scegliere una porta**: questa sarà scelta a caso fra quelle libere e verrà inclusa nel messaggio affinché il server sappia rispondere al client (conosca la sua porta sulla macchina mittente).

Nell'esempio sopra abbiamo Host 1 che comunica grazie alla porta 1208 con Host 2 che ospita due applicazioni diverse, una sulla porta 1522 e una sulla porta 1836.

Avendo Host 1 specificato l'indirizzo IP di Host 2 e la porta 1522 (cioè come end point Server 1) ora è in comunicazione proprio con Server 1.

1.4) Stabilire una connessione

Per effettuare una connessione i due partecipanti si devono ovviamente mettere d'accordo. Più precisamente, devono sincronizzare i loro numeri di sequenza. Per fare questo viene adottato un protocollo 3-way handshake:



L'host 1 inizia inviando una CR (CONNECTION REQUEST), e fornendo il numero di sequenza dal quale intende partire (x).

L'host 2 invia un ACK per confermare la ricezione ed annuncia il suo numero di sequenza y.

Quindi l'host 1 inizierà la sua trasmissione dal numero di sequenza x mandando un ACK del numero di sequenza y: ora i numeri di sequenza sono sincronizzati.

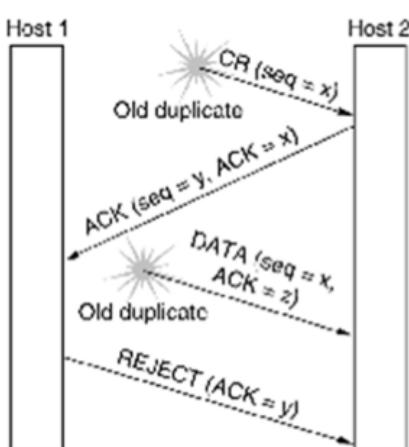
Con segmenti duplicati/ritardati

Nel caso in cui accadano errori durante la trasmissione 3-way handshake:

Abbiamo che il primo segmento di CR è un duplciato, cioè si riferisce ad una vecchia connessione. Quindi l'host 1 non è a conoscenza della richiesta che sta inviando.

L'host 2 riceve una richiesta e risponde come visto sopra.

L'host rifiuta inviando un REJECT e così l'host 2 non lascia la connessione pendente.

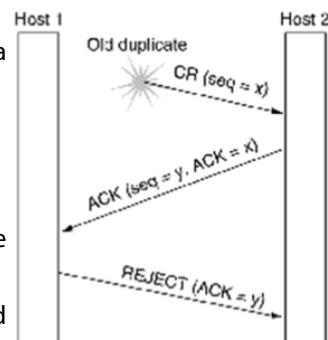


Questo caso più complesso vede sia la CR che altri dati ritardati.

L'host 2 riceve un CR ritardato e risponde ad esso come conforme al protocollo.

Arrivavano ad host 2 alcuni dati (anch'essi ritardati) ma hanno un ACK non conforme a quanto appena comunicato da host 2 (l'ACK è per il numero di sequenza z e non per il numero di sequenza y) quindi host 2 si rende conto che non gli interessa e lo butta via.

Intanto host 1 invia anche un REJECT della connessione causata dal ritardo del CR (come nel caso visto prima).



Questa tecnica funziona bene ma ha un rischio: il **wrapping dei numeri di sequenza**. Se i numeri di sequenza dovessero essere troppo pochi allora si adotta un'opzione aggiuntiva detta PAWS.

Abbiamo anche un **problema di sicurezza**: un avversario può prevedere i numeri di sequenza ed infilarsi nella connessione fingendosi l'interlocutore che aveva richiesto la connessione. Per risolvere questo problema si adottano sequence number iniziali **pseudocasuali**.

1.5) Chiudere una connessione

Vi sono due modi di chiudere una connessione:

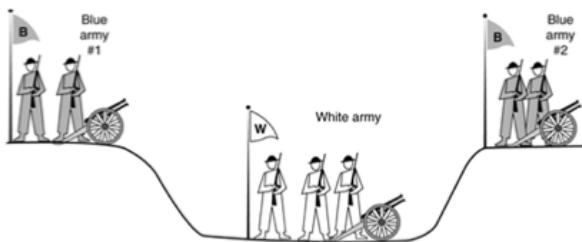
- Simmetrico: usato da TCP, richiede una chiusura "concorde" dalle due parti per evitare perdita di dati.
- Asimmetrico: non si cura dello stato del peer e "trancia" la connessione di netto.

Dato che è usato da TCP concentriamoci sul caso **simmetrico**.

Il rilascio simmetrico funziona al 100% se la quantità di dati da trasmettere è conosciuta ma questa pre-condizione, ovviamente, nella rete Internet non c'è.

Il problema dei due eserciti

Rappresentiamo la problematica tramite il **problema dei due eserciti**.



L'esercito bianco (nella valle) è più numeroso di quello grigio ma meno numeroso della somma fra i due.

L'esercito grigio può vincere la battaglia solo radunandosi e attaccando nello stesso momento l'esercito bianco.

Come potrebbe funzionare la comunicazione fra le due parti dell'esercito grigio?

- L'esercito grigio a destra manda un messaggero all'esercito grigio di sinistra col messaggio: "Attacchiamo domani a mezzogiorno. Che ne dite?"

- Il messaggio viene recapitato e il secondo comandante risponde affermativamente.

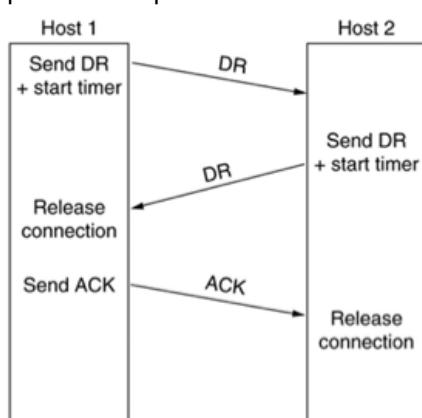
L'attacco sarà vincente in ogni caso? No.

Il secondo comandante non è sicuro che la sua **risposta sia arrivata**. Si può quindi pensare che una volta ricevuto il secondo messaggio il primo comandante mandi un ACK, ma a questo punto egli non sarà sicuro che l'ACK sia arrivato.

È dimostrabile che **non esiste protocollo in grado di garantire la common knowledge**.

Si noti che se entrambe le parti richiedono che anche l'altra sia pronta, non avremo mai una situazione di terminazione.

Questo problema è evitabile non imponendo la necessità di un accordo e demandando il problema ad un protocollo superiore che utilizza il livello transport. Perciò, abbiamo un protocollo **non infallibile ma adeguato**.



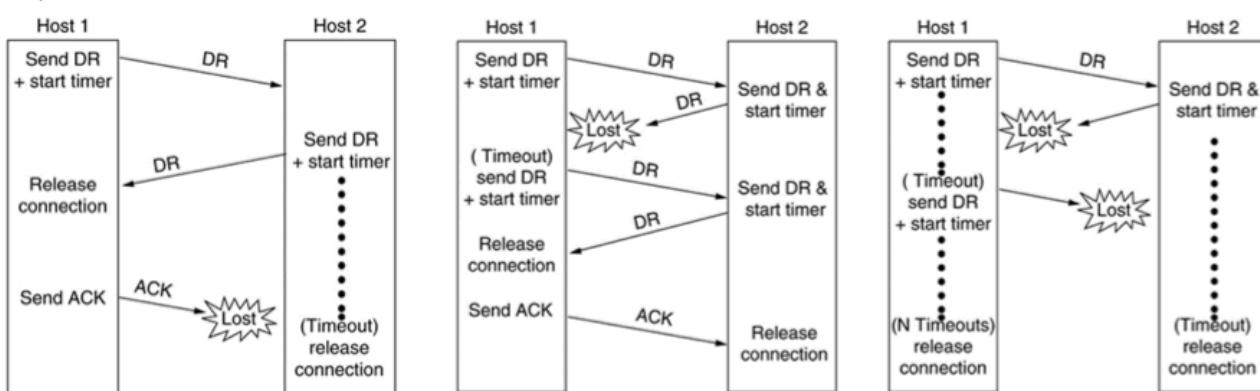
Solitamente accade che l'Host 1 che vuole terminare la connessione invii un DR e faccia partire un timer dalla sua parte.

Dunque, l'Host 2 risponde con un'altro DR e fa partire anch'esso un timer.

A questo punto, quando l'Host 1 riceve il DR rilascia la sua connessione e manda un ulteriore ACK di conferma a Host 2.

Quando l'ACK arriva a destinazione, anche l'Host 2 rilascerà la sua connessione.

Se qualcosa non dovesse funzionare saranno i timer a "salvare" la situazione, ci troveremo in una delle tre casistiche:



Come detto prima, teoricamente questo protocollo può fallire. Se viene perso il DR iniziale inviato da Host 1 (N volte) il mittente rilascerà la connessione e l'Host 2 non potrà saperlo.

Perciò, se non arrivano pacchetti entro un certo tempo una connessione viene chiusa. Per questo talvolta capita che vengano inviati dei pacchetti di **keep-alive** che servono a tenere la connessione aperta anche se sono privi di dati.

1.6) Controllo degli errori e controllo di flusso

Vogliamo applicare, con TCP, anche controllo degli errori e controllo di flusso. Le soluzioni sono quelle già viste a livello data link:

- CRC & ARQ per il controllo degli errori
- Stop-and-wait & sliding window per il controllo di flusso

La differenza fra lavorare a livello transport ed a livello data link

Essere a livello transport ci proietta in un contesto differente rispetto a quello data link.

Il checksum di livello data link si occupa di proteggere il frame durante un solo collegamento, mentre invece il checksum del livello transport protegge il segmento mentre attraversa tutta la rete: il data link è utile ma non basta.

Inoltre abbiamo il concetto di connessione e quindi l'allocazione di risorse può essere effettuato per una singola connessione (ad esempio).

L'allocazione dei buffer

La gestione dei buffer è effettuabile in tre modi possibili:

- Buffer di **dimensione costante** per ogni segmento (ma ogni segmento può avere dimensione diversa)
- Buffer di **dimensione variabile** (gestione assai complessa)
- Buffer **circolare di grande dimensione** per ogni connessione (spazio sprecato se la connessione è poco utilizzata)

In ogni caso ci sarà un momento di trattazione in cui il mittente richiederà un certo numero di buffer e il destinatario allocherà la quantità di buffer più prossima alla richiesta (considerando però i suoi limiti).

Ogni segmento che il mittente invia dovrà bufferizzarlo per effettuare ritrasmissione nel caso in cui venga perso (eventualmente, se non ha più spazio, attenderà).

Allo stesso modo il destinatario, quando invia degli ACK, fornisce anche informazioni sullo stato di allocazione dei buffer.

Illustriamo ora una situazione in cui accade un potenziale deadlock a causa di un messaggio perso (e anche a causa delle ritrasmissioni effettuate da A).

A	Message	B	Comments
1	→ < request 8 buffers>	→	A wants 8 buffers
2	← <ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→ <seq = 0, data = m0>	→	A has 3 buffers left now
4	→ <seq = 1, data = m1>	→	A has 2 buffers left now
5	→ <seq = 2, data = m2>	•••	Message lost but A thinks it has 1 left
6	← <ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7	→ <seq = 3, data = m3>	→	A has 1 buffer left
8	→ <seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9	→ <seq = 2, data = m2>	→	A times out and retransmits
10	← <ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11	← <ack = 4, buf = 1>	←	A may now send 5
12	← <ack = 4, buf = 2>	←	B found a new buffer somewhere
13	→ <seq = 5, data = m5>	→	A has 1 buffer left
14	→ <seq = 6, data = m6>	→	A is now blocked again
15	→ <ack = 6, buf = 0>	→	A is still blocked
16	••• <ack = 6, buf = 4>	→	Potential deadlock

Inizialmente A richiede 8 buffer

B ne alloca solamente 4

Per evitare questa situazione è necessario inviare costanti informazioni relative ai buffer (anche senza l'ACK di un qualche messaggio).

I colli di bottiglia

Abbiamo due possibili colli di bottiglia:

- **Pochi buffer** (memoria piena, i due sono in attesa).
- **Capacità di trasporto della rete** (se i router sul percorso mittente-destinatario possono al massimo scambiare x pacchetti nell'unità di tempo ed il mittente invia pacchetti con una velocità maggiore di x, la rete si congestionata poiché non è in grado di consegnare i segmenti che riceve nello stesso tempo in cui li riceve).

Il primo problema si verifica sempre meno, le memorie costano poco. È quindi necessario un sistema per far sì che la velocità di throughput sia regolata dalla capacità della rete!

Lo schema di controllo di flusso si basa sulla finestra scorrevole (sliding window) che riesce così a effettuare sia controllo di flusso che controllo della congestione.

Se la rete può gestire c segmenti al secondo ed il round trip time è di r , allora la dimensione della finestra mittente sarà $c \cdot r$. Chiaramente la finestra sarà adattabile a seconda delle condizioni della rete.

1.7) Ripristino dopo un crash

Durante una connessione può ovviamente accadere un crash. Come gestiamo la situazione? Supponiamo di avere un crash durante la trasmissione di un grosso file... Ecco lo scenario:

- L'host sta inviando il file mediante protocollo s-&-w
- Il server prende i pacchetti e li passa al transport (che poi li passerà all'applicazione)
- Il server crasha improvvisamente, si riaccende e tutte le sue tabelle sono ovviamente re-inizializzate: a che punto era il trasferimento?

Un primo approccio potrebbe essere chiedere in broadcast al client "a che punto ero arrivato?"...

In ogni caso, il client si può trovare in due stati differenti:

- S1, il segmento che ha inviato è in circolazione sulla rete
- S0, non vi sono segmenti in circolazione sulla rete

Chiaramente starà a lui decidere se ritrasmettere o meno:

Ad esempio, se si trova in stato S1 e non ha avuto un ACK, ritrasmetterà. Ma se il protocollo applicativo del server invia prima l'ACK e poi tratta il segmento, supponendo un crash proprio in mezzo alle due operazioni, l'host riceverà un ACK per un segmento che il server in realtà non possiede!

Allo stesso modo se dal server avviene prima la scrittura e poi l'invio dell'ACK, l'host invierà un segmento due volte (duplicato). **Il protocollo fallisce sempre.**

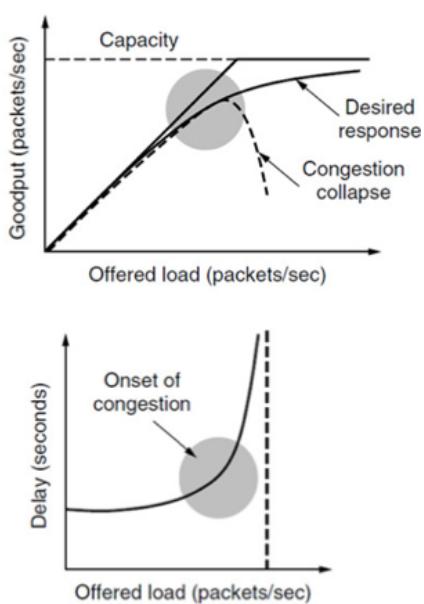
Il concetto che si vuole esprimere è che il ripristino di un crash a livello N può essere svolto solamente a livello N+1 e solo se il livello superiore dispone di informazioni sufficienti per effettuarlo. Infatti il transport può risolvere crash a livello di rete ma sarà l'application a dover risolvere i crash a livello transport.

Il server non può memorizzare tutte le informazioni relative alle connessioni! Sta al client farlo: ecco perché l'introduzione dei cookies, delle variabili di sessione, ecc.

1.8) Controllo della congestione

Abbiamo accennato prima al dimensionamento della finestra per controllare la congestione sulla rete. Ma questo non può bastare poiché ogni host può inviare pacchetti e quindi è necessario un **meccanismo congiunto** di controllo poiché la congestione dipende da molti attori.

Si deve effettuare una allocazione della banda che rispetti tre principi: **efficienza** (usare tutta la banda disponibile), **equità** (ogni processo deve avere una parte equa della banda rispetto agli altri processi), **dinamicità** (l'allocation deve cambiare a seconda dei cambiamenti della rete) e tutto questo **evitando le congestioni**.



I due grafici a fianco ci mostrano una problematica:

Nel primo grafico abbiamo il goodput (pacchetti inviati utili e quindi non le ritrasmissioni) in relazione ai pacchetti al secondo trasmessi: ci aspetteremmo un andamento asintotico rispetto alla capacità, ma invece abbiamo un **crollo terribile del goodput!** Questo per un effetto del protocollo di trasporto: quando un pacchetto si perde il suo timer scatta e questo genera ritrasmissione. Ma nel caso in cui il pacchetto non si fosse perso ma bensì si trovasse in coda su qualche router (congestionato, quindi) allora l'invio di un duplicato non farà altro che peggiorare la situazione senza portare alcuna miglioria.

Ovviamente anche i ritardi crescono in relazione a quanto detto (come ci mostra il secondo grafico).

Quindi i ritardi generano congestione.

Abbiamo che il goodput è il carico/ritardo: questa quantità aumenterà con il carico (inizialmente) ma con l'aggiungersi del ritardo crollerà terribilmente. Avere goodput maggiore implica (ovviamente) una gestione efficiente della rete.

Perciò i timer dovranno essere regolati dinamicamente per evitare questo problema.

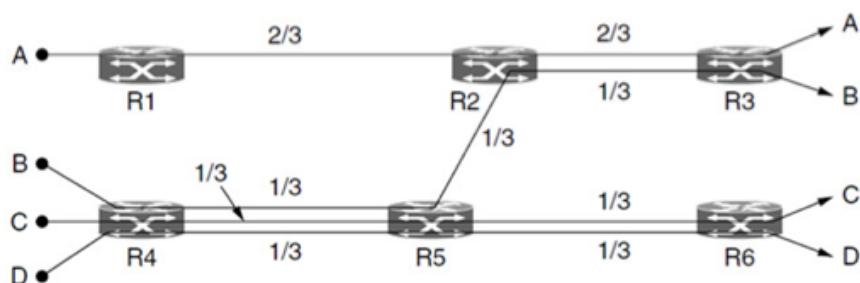
Allocare la banda staticamente: max-min fairness

Dunque, come allochiamo la banda?

Se N flussi usano un solo collegamento, l'allocazione è abbastanza semplice: 1/N della banda totale ad ogni flusso. Ma se i flussi hanno **percorsi di rete sovrapposti** allora la situazione si fa più complicata: potremmo avere un flusso che attraversa tre collegamenti e altri che invece ne attraversano uno solo. Diamo minore banda ai singoli flussi? C'è un conflitto fra equità ed efficienza.

La nozione di equità è detta **max-min fairness** e porta ad una conseguenza ovvia:

Aumentare la banda di un flusso porta alla diminuzione della banda di un altro flusso con allocazione minore o uguale: perciò aumentare la banda di un flusso peggiora la situazione per i flussi meno abbienti.



Possiamo fare questo tipo di riflessioni sullo schema appena riportato:

- Se allocassimo a B un flusso maggiore sul collegamento R₂ - R₃, ce ne sarà di meno per A. Ma questo non sarebbe un problema per A che già possiede 2/3 della banda, bensì per C e D che nel tratto R₄ - R₅ avrebbero meno di 1/3 di banda allocata!
- Notiamo subito che ci sono degli "sprechi" ad esempio nel collegamento R₅ - R₂ o R₅ - R₆, ma questi sono **inevitabili** per far sì che i flussi meno abbienti non rimangano "senza banda".

Abbiamo quindi capito che conoscendo globalmente la rete sarebbe possibile ottenere le giuste allocazioni di banda: perfetto. Dopodiché i vari flussi dovranno (dobbiamo ancora vedere come) lentamente aumentare fino al limite imposto dal min-max. Ogni flusso aumenterà "per gli affari suoi" fino al suo limite e poi continuerà a trasmettere a quella velocità.

Tutto perfetto! Ma questa è una **allocazione statica**. E la rete è estremamente dinamica: connessioni vengono create e lasciate continuamente.

Allocare la banda dinamicamente: AIMD

Il protocollo di trasporto regola il tasso di spedizione grazie ai **feedback** inviati dalla rete. Ve ne sono di diversi tipi:

- Feedback **esplicito**:

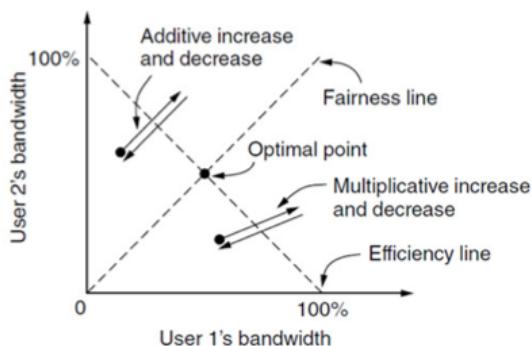
- I router comunicano direttamente ai client la velocità a cui spedire (esempio: ECN in TCP). È un metodo impreciso.

- Feedback **implicito**:

- FAST TCP misura il round time trip e lo usa come segnale per evitare una congestione
- La perdita di pacchetti è indice di congestione

È possibile combinare queste tecniche.

L'**AIMD** (additive increase multiplicative decrease) è una proposta di legge di controllo appropriata per arrivare ad un punto di funzionamento equo ed efficiente.



Sulle x abbiamo la percentuale di banda data all'utente 1, mentre sulle y la percentuale di banda data all'utente 2.

Su tutta la diagonale fra i due punti del 100% abbiamo il massimo dell'**efficienza** poiché (con percentuale diversa fra i due utenti) assegnamo tutta la banda.

Invece sulla diagonale opposta abbiamo la linea dell'**equità** che distribuisce la banda in modo equo fra i due user.

Il punto di ottimo si trova, ovviamente, nel mezzo.

Perché la soluzione che adottiamo è quella di avere un incremento additivo ed un decremento moltiplicativo?

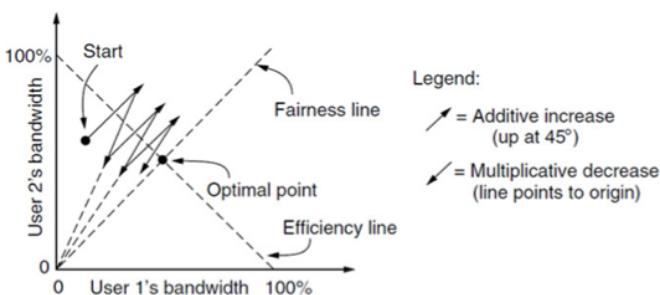
Supponiamo le altre casistiche:

- **Incremento e decremento additivo:** i due user aumentano incrementalmente l'utilizzo della rete (ad esempio 1Mbps ogni secondo) fino a che non giunge loro un feedback (non ci interessa di che tipo) che li avverte che la rete è prossima alla congestione. Dunque, diminuiscono l'uso delle risorse (sempre in modo additivo). Come vediamo dall'esempio riportato in alto, i due utenti non faranno altro che oscillare su una linea additiva.

- **Incremento e decremento moltiplicativo:** i due utenti aumentano moltiplicativamente il loro utilizzo delle risorse (ad esempio del 10% ogni secondo) fino a che non otterranno un feedback che li avverte che la rete è prossima alla congestione. Dunque entrambi diminuiranno in modo moltiplicativo l'utilizzo delle risorse. Come vediamo dall'esempio in alto, i due utenti non faranno altro che oscillare su una linea moltiplicativa (inclinata rispetto a quella additiva).

In entrambi i casi abbiamo sempre (mediamente) **efficienza** ma mai equità!

Ma se noi combiniamo insieme gli effetti dell'additività e della moltiplicatività otteniamo invece qualcosa di simile:



Abbiamo esattamente quello che ci serviva!

L'inclinazione del decrease fa sì che lentamente si raggiunga l'equilibrio fra la fairness e la efficency.

La politica di incremento deve quindi essere **graduale** mentre invece la politica di decremento deve essere **aggressiva**.

TCP utilizza la strategia sopra descritta e per quanto concerne gli altri protocolli, anche loro devono sottostare a queste regole per evitare "prepotenze" che renderebbero sia vani gli sforzi di TCP e che comunque non garantiscono ai suddetti protocolli la garanzia di "funzionare meglio": anche loro subiscono le congestioni, come tutti.

Questi protocolli sono detti **TCP-friendly**.

Si noti che questo "altalenare" è sia necessario per via delle caratteristiche dinamiche della rete e sia perché **non esiste messaggio di ripresa di velocità**, ovvero, mentre con ICMP possiamo dire "rallenta", non esiste modo per dire "vai più veloce". Quindi questa tecnica permette ai singoli processi di "provarci" e vedere se è possibile aumentare il proprio throughput.

Regolare i tassi di invio

Per modificare il tasso di invio si opera sulla finestra (o almeno così fa TCP). Se la dimensione della sliding window è W e il round trip time è RTT, allora la dimensione ottimale della finestra sarà W/RTT .

La finestra è quindi **sia funzionale per il controllo di flusso e sia funzionale per il controllo della congestione.** Ottimo!

TCP e le reti Wifi

Come sappiamo il protocollo di trasporto dovrebbe essere indipendente rispetto alle tecnologie di rete. Ma questo non è così facile da ottenere se parliamo di reti wifi.

Giacché TCP sfrutta le perdite come segnale di congestione, nel caso delle reti wifi le perdite sono notevoli e quindi TCP non riesce a comportarsi adeguatamente. Data anche la **legge di Padhaye** che dice che il throughput aumenta come l'inverso della radice quadrata del tasso di perdita dei pacchetti, capiamo che non è possibile far lavorare TCP con le reti wifi! Infatti TCP è tanto più veloce quando la rete perde meno pacchetti. TCP lavora bene con un tasso di perdita dell'1% e nelle reti wifi un tasso accettabile è il 10% delle perdite. È un problema che dobbiamo risolvere.

La soluzione consta nel **mascherare le perdite wireless usando ritrasmissioni sul livello wireless stesso**. Ad esempio il protocollo 802.11 prova molte volte ad ottenere una ritrasmissione del pacchetto prima di comunicare al livello superiore che il pacchetto è, a tutti gli effetti, andato perso.

Ma il livello transport del mittente non può sapere se ci sono linee wifi sul tragitto che il pacchetto farà e quindi starà al datalink occuparsi della questione: datalink e transport devono cooperare. La perdita di un pacchetto deve essere interpretata come perdita oppure come segnale di congestione ma non entrambe!

Comunque i due meccanismi agiscono su scale temporali differenti: microsecondi/millisecondi per quanto riguarda il datalink mentre i timer del livello transport si attivano nell'ordine dei millisecondi/secondi perciò il livello datalink ha "tempo" di provare le varie ritrasmissioni prima che il transport si accorga della perdita e lo interpreti come una congestione (rallentando la rete).

La strategia di mascheramento è sufficiente a risolvere il problema nella maggior parte delle reti wifi.

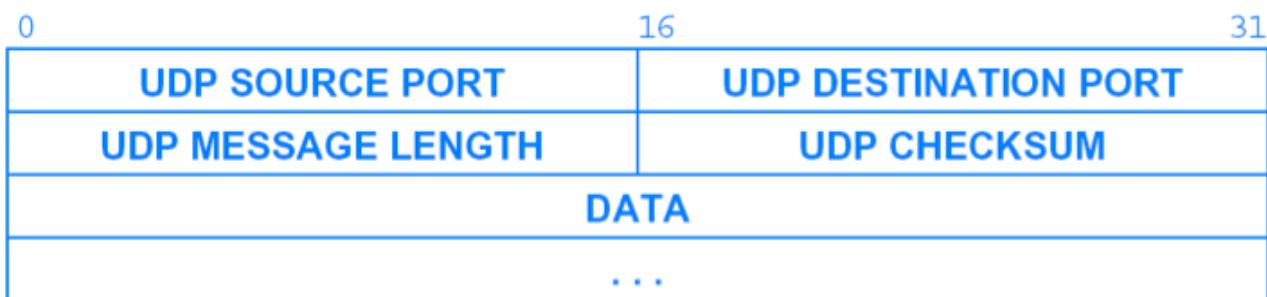
2) Il protocollo UDP

Il protocollo UDP è il più semplice protocollo transport. Non offre, sostanzialmente, alcuna nuova funzionalità rispetto al livello IP a parte la possibilità di poter identificare un endpoint (in sostanza fornisce la gestione delle porte) e quindi dare un servizio **da-applicazione-a-applicazione**. Fa quindi da tramite fra IP ed il livello applicativo (IP non può servire un application).

2.1) Le caratteristiche basiliari

UDP è un protocollo **connection-less** con servizio **best-effort** e perciò i pacchetti possono essere persi, arrivare in ordine errato, ecc. Non c'è handshaking fra i due interlocutori. Perciò le applicazioni che sfruttano a livello trasporto UDP dovranno effettuare una loro gestione degli eventuali errori/pacchetti mancanti o disordinati.

2.2) Il segmento UDP

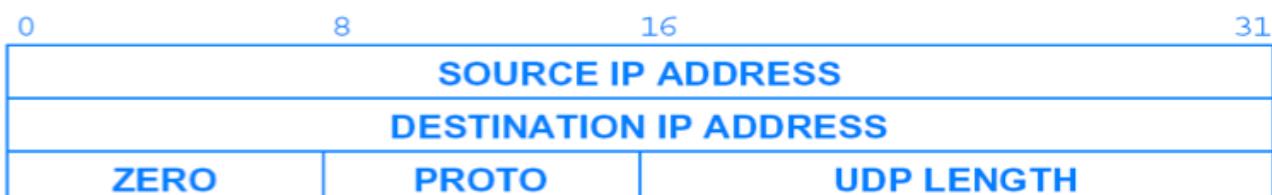


Le porte sono **interi a 16 bit** e abbiamo:

- UDP SOURCE PORT: porta sorgente (eventualmente vuota)
- UDP DESTINATION PORT: porta di destinazione
- UDP MESSAGE LENGTH: lunghezza del messaggio comprensiva dell'head
- UDP CHECKSUM: checksum sul messaggio
- DATA: il payload

Un particolare checksum: lo pseudo-header UDP

Il checksum UDP viene calcolato in modo alquanto particolare. Non vogliamo limitarci a controllare la correttezza del pacchetto UDP ma bensì che la coppia <indirizzo IP, porta> sia quella giusta. Perciò includiamo nel checksum anche l'indirizzo IP del destinatario. UDP si costruisce questo particolare pseudo-header:



Questo header **non viene inviato** ma viene incluso nel CHECKSUM.

- SOURCE IP ADDRESS: IP della fonte
- DESTINATION IP ADDRESS: IP del destinatario
- ZERO: serie di zeri
- PROTO: protocollo (per UDP si ha 17)
- UDP LENGTH: lunghezza del pacchetto UDP senza considerare lo pseudo-header

Abbiamo quindi una violazione della stratificazione! Al livello transport parliamo di indirizzi IP. Si tratta di un'ottimizzazione sporca ma molto utile poiché l'endpoint è definitibile da <IP, porta>; non usare una delle due è insensato.

Tutte queste informazioni non vengono inviate perché sono recuperabili dall'altra parte dal destinatario il quale potrà facilmente ricalcolare il CHECKSUM.

2.3) Le porte

Una porta può essere vista come una **coda realizzata** dal **sistema operativo**.

Se arriva un datagram quando la porta non è in uso, allora verrà dato un messaggio **port unreachable**. Se la coda è piena, allora il SO scarterà il datagram.

Due tipologie di porte

Come può un client sapere su che porta contattare un server? Semplice: i servizi dei server vengono messi su porte **well-known**. Tali porte sono degli "standard", basti pensare ad http sulla porta 80 (che i browser usano di default).

Le porte well-known hanno la loro definizione all'interno del file /etc/services (su sistemi UNIX).

Esistono poi altre porte (dalla 1024 in su) dette porte **dinamiche** che vengono usate dai client per essere contattati (sono porte scelte randomicamente e on fly).

Chiaramente le porte devono essere assegnate in modo univoco per evitare applicazioni che "si mangino i messaggi a vicenda".

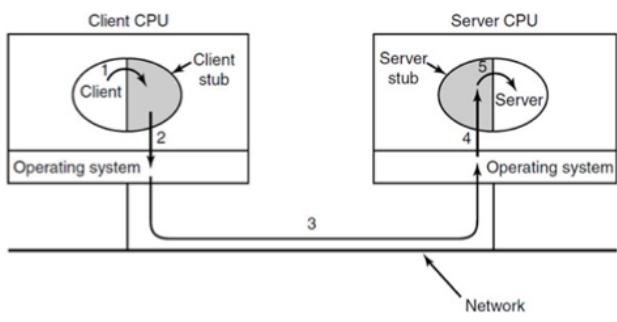
2.4) Le remote procedure call (RPC)

L'invio di un messaggio ad un host remoto è simile ad una chiamata di funzione all'interno di un linguaggio di programmazione. Questa è, almeno, l'illusione che si vuole dare al programmatore del livello applicativo. Perciò il passaggio di parametri ed il ritorno sono gli strumenti di comunicazione fra il livello transport e quello application.

Ad esempio la funzione `get_IP_address(host_name)` invia un pacchetto UDP ad un server DNS ed attende una risposta (inviata come valore di ritorno). Nel caso in cui la risposta non arrivi entro lo scadere del timer, allora verrà effettuata una ritrasmissione.

Abbiamo quindi la possibilità di **eseguire qualcosa su un host remoto**. Quando un processo su un certo host 1 viene eseguito su un host 2, il primo host sarà in attesa mentre il secondo host eseguirà. Abbiamo quindi una netta divisione dei ruoli: l'host 1 sarà il **client** l'host 2 sarà il **server**.

Gli stub



Come funziona tutto questo? Il client ed il server dispongono dei rispettivi **stub**. Lo stub funge da "parte locale" ed è interpellabile direttamente dall'application.

Guardando una comunicazione da client a server vediamo che ad esso vengono inviati i parametri, il SO effettua **marshalling** (i parametri vengono elaborati e preparati per l'invio) e quindi vengono inviati tramite la pila TCP/IP.

Nel momento della ricezione avverrà il processo inverso, il SO tradurrà i parametri per lo stub il quale comunicherà con l'application del server.

Da ciò se ne deduce che certe operazioni siano impossibili: il **passaggio di puntatori non ha senso** (non c'è memoria condivisa), abbiamo **problemi relativi alla tipizzazione** (bisogna passare anche la dimensione degli oggetti), e **le variabili globali non hanno senso**.

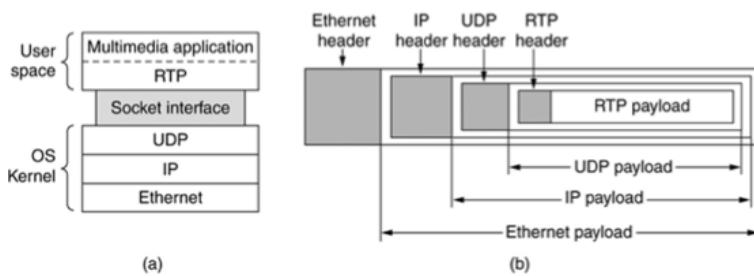
Operazioni idempotenti

UDP, come sappiamo, effettua ritrasmissione in caso di perdita. Ma se la perdita non ci fosse stata, potremmo avere due volte lo stesso messaggio e quindi compiere una certa operazione più di una volta (ovvero più del richiesto). Per questo si effettua RPC tipicamente con operazioni **idempotenti** ovvero che, eseguite più volte, non cambiano il risultato. Ad esempio una richiesta al DNS è idempotente, un'aggiornamento di un database invece no (vi sono protocolli appositi per gestire la problematica).

2.5) Trasporto real-time (protocollo RTP)

UDP è molto utilizzato per la trasmissione real-time (streaming, telefonia, musica, ecc). Il protocollo RTP mira proprio a questo.

RTP è eseguito sullo spazio utente ma è un protocollo di transport.



è eseguito a livello utente e non a livello transport.

Ancora una volta ci troviamo in una situazione di violazione del concetto di livelli per motivi di ottimizzazione.

Le funzioni fornite da RTP sono tipicamente di trasporto (non ha un "vantaggio" di per sé e lavora per un protocollo sovrastante, l'application) ma

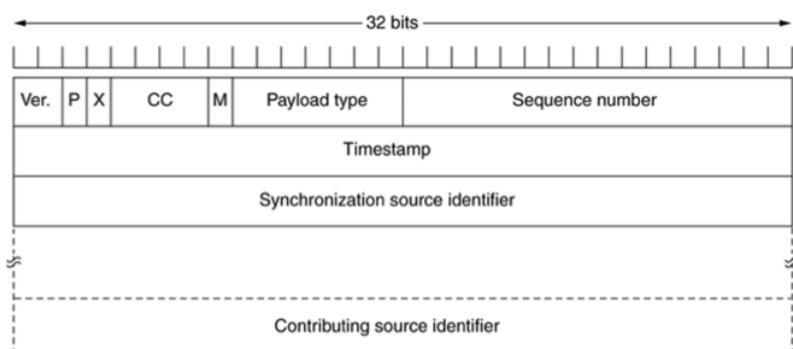
In un'applicazione multimediale flusso video e flusso audio (ad esempio) sono sconnessi e diversi: sarà compito dell'RTP prenderli ed effettuare **multiplexing** per inviarli poi ad UDP. Il processo inverso avverrà, ovviamente, per il ricevente. I pacchetti RTP non vengono trattati in modo differente dai router (seppur sia possibile specificare nel campo TYPE OF SERVICE una direttiva per la QoS) e quindi hanno lo stesso rischio di perdita/duplicazione di un normale pacchetto UDP.

Ad ogni pacchetto viene assegnato un numero incrementale e nel caso di perdite la gestione dipende dal protocollo RTP in uso.

La **ritrasmissione non è importante** poiché se abbiamo un flusso continuo di dati non ci servirà sapere un'informazione (un fotogramma piuttosto che un suono) relativa ad alcuni attimi prima.

Il payload RTP può contenere più informazioni codificate diversamente: questo sarà indicato nell'intestazione dello stesso affinché il livello applicativo possa effettuare una decodifica corretta dell'informazione (altrimenti illeggibile).

Viene spesso usato un **timestamp** per sincronizzare i flussi di diversa natura di una stessa trasmissione (audio e video) così da "astrarli" dall'ordine effettivo di arrivo.

Header RTP

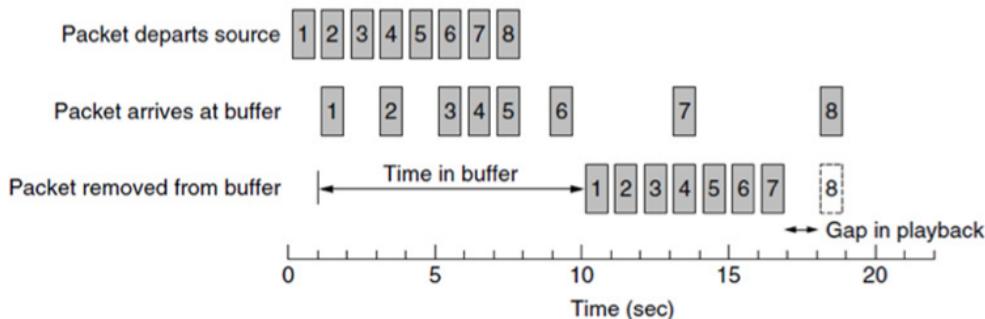
Abbiamo un campo VERSION per indicare la versione di RTP, il bit P che indica la presenza di padding (per ottenere un multiplo di 4 byte), il bit X per indicare la presenza di un head esteso (implementato per futuri scopi attualmente non usati). Il campo CC indica il numero di sorgenti attive da cui si attinge la trasmissione in real-time. Il bit M specifica l'applicazione che deve ricevere il pacchetto, il campo Payload Type indica invece quale codifica è stata adottata per il flusso trasmesso. Infine abbiamo il sequence number ed il timestamp di cui abbiamo già parlato. Synchronization source identifier è utile per quei flussi che hanno bisogno di altri flussi (es: audio e video) oppure per discernere più flussi RTP che sono stati multiplexati in un solo pacchetto UDP e i Contributing source identifier sono facoltativi.

Protocollo RTCP

Definito nello stesso RFC di RTP abbiamo il protocollo RTCP. Tale protocollo gestisce le interazioni verso la sorgente (ritardi, jitter, banda, congestione), la sincronizzazione (fra flussi) e l'interfaccia con l'utente (denominazione delle sorgenti).

Chiamiamo **jitter** la variazione del ritardo: tale fattore è ben più fastidioso del ritardo stesso nel caso di trasmissioni real-time. Abbiamo infatti il classico effetto "singhiozzo" durante la riproduzione di un video, ad esempio, mentre sarebbe preferibile attendere in modo "costante" così da avere un flusso continuo.

La soluzione per la riduzione del tasso di jitter è **il buffering** delle informazioni.



Nell'esempio sopra abbiamo che i pacchetti fino al settimo arrivano in tempo utile rispetto alla riproduzione mentre il pacchetto otto arriva troppo tardi creando un singhiozzo nella riproduzione.

3) Il protocollo TCP

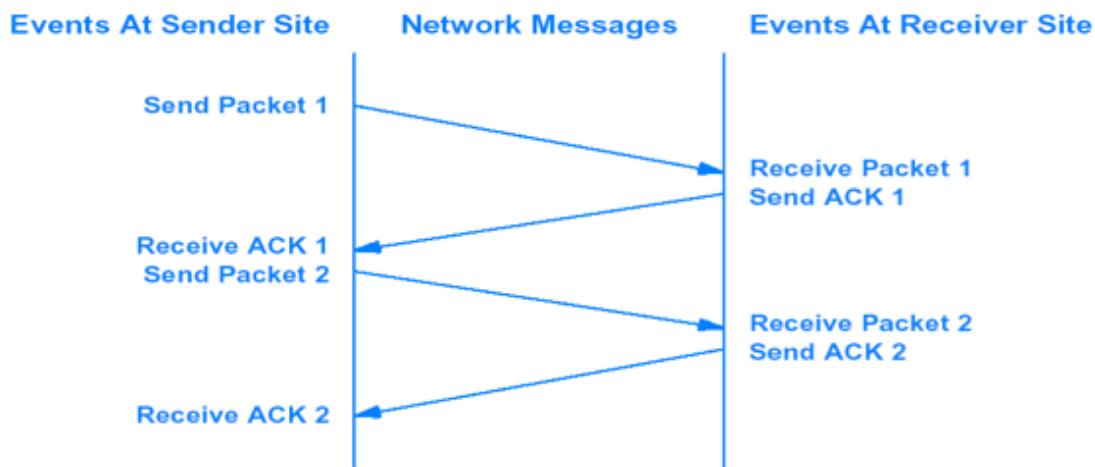
Il protocollo TCP ovvia a ciò che IP non può fare: controllo di flusso, affidabilità, controllo della congestione.

La prima applicazione pensata per TCP fu il terminale remoto.

3.1) Realizzare l'affidabilità

Come possiamo realizzare l'affidabilità? Usando un **riscontro positivo e ritrasmissione**. Si tratta di una tecnica già vista molte volte in precedenza:

- Il TCP mittente invia un datagram e fa partire un timer. Il mittente deve tenere traccia del pacchetto appena inviato (per eventuali ritrasmissioni).
- Il TCP destinatario riceve il datagram, lo inoltra a livello applicativo, crea un ACK e lo spedisce sulla rete.
- Il TCP mittente riceve l'ACK e ripulisce il buffer dal datagram che stava serbando per la ritrasmissione.



Nel caso in cui il timer scatti (l'ACK o il datagram sono andati persi) il mittente effettua una ritrasmissione. Da ciò se ne deduce che sia necessario un controllo dei duplicati.

Il compito di spedire riscontri, notare la duplicazione di un pacchetto, ritrasmettere, ecc. appartengono a TCP.

Nel caso in cui nonostante tutti i tentativi di ritrasmissione di TCP non si riesca ad inviare un pacchetto, TCP considera la connessione **abortita** (morta).

Stop & wait: uno scarso utilizzo della banda

Come sappiamo da quanto visto nel protocollo datalink, un approccio stop & wait è molto poco performante. Il motivo è che il mittente aspetta di ricevere un ACK prima di proseguire il suo lavoro. C'è una fase in cui il tempo è sprecato. Non sfruttiamo, quindi, il fatto che la rete è full-duplex (possiamo trasmettere in entrambe le direzioni contemporaneamente).

Volendo fare una valutazione pratica dell'effettiva utilizzazione del canale, consideriamo due utenti che si trovano a una distanza tale per cui il RTT è di 30 millisecondi ed il canale trasmissivo usato dai due (R) è pari a 1 Gbps (cioè 10^9 bit al secondo). I pacchetti sono di $L = 1.000$ byte (8.000 bit).

Il tempo richiesto per trasmettere un pacchetto è quindi:

$$T_{\text{trasm}} = \frac{L}{R} = \frac{8000 \text{ bit/pacchetto}}{10^9 \text{ bit/sec}} = 8 \text{ microsecondi}$$

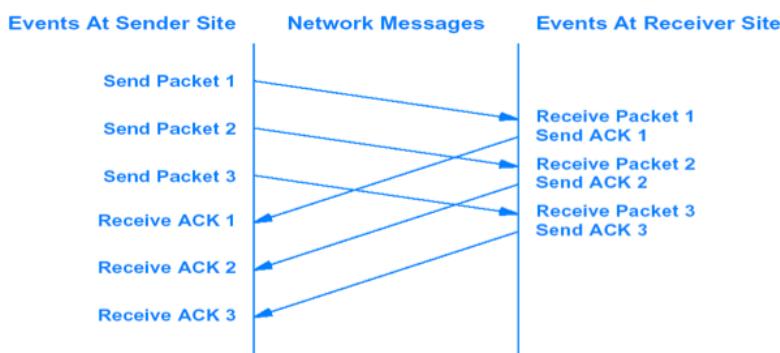
Tale pacchetto viaggia per 15 millisecondi e quindi l'ultimo bit arriva al destinatario ad un tempo $t_1 = \text{RTT}/2 + L/R$ e cioè 15,008 ms. Viene subito generato l'ACK e spedito, in un certo istante $t_2 = \text{RTT} + L/R = 30,008$ ms.

Perciò avendo come concetto di utilizzazione del canale solo il momento in cui c'è qualcosa in trasmissione sul canale, abbiamo:

$$U_{\text{canale}} = \frac{L/R}{\text{RTT} + L/R} = \frac{0,008}{30,008} = 0,00027$$

Stiamo usando il canale a meno dell'1%! Terribile.

La soluzione è quindi quella di **spedire "tutto quello che ho"**: mi segnerò quali riscontri ho ottenuto e quali no, bufferizzerò ogni pacchetto inviato in attesa dell'ACK ma non attenderò mai un riscontro prima di spedire qualcosa. La rete è, così, totalmente utilizzata.



Sembrerebbe perfetto! Ma, il ricevitore, dove mette tutti i pacchetti che gli arrivano? Dovrà sfruttare dei buffer.

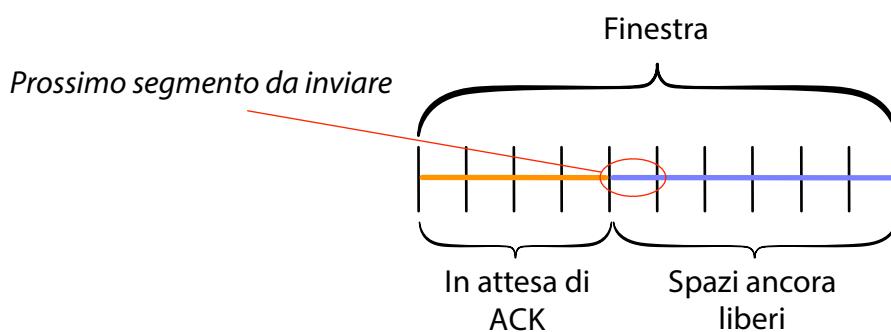
Ecco un problema di **affidabilità**: se i buffer sono pieni il ricevitore è costretto a scartare i pacchetti senza avvertire il mittente.

Non si può quindi puntare allo stesso istante ad avere la rete usata al massimo garantendo l'affidabilità del servizio. È necessario un **sistema di controllo di flusso**.

L'idea della finestra scorrevole

Come detto sopra la finestra viene implementata per effettuare controllo di flusso ma anche per effettuare controllo della congestione. Per il momento ci limiteremo a studiare come la finestra scorrevole permetta di effettuare controllo di flusso.

La dimensione della finestra di un host rappresenta la capacità recettiva dell'altro host. Quindi la finestra è dimensionata sui buffer del ricevitore! Abbiamo tre puntatori: l'inizio della finestra (in attesa di ACK), il prossimo datagram da inviare, la fine della finestra.



Il ricevitore dovrà mandare degli ACK per dire di aver ricevuto dei dati e, intanto, **fornire informazioni relative alla propria finestra**.

Ad esempio con finestra di dimensione 3 abbiamo:



La nuova finestra va da 4 a 6 poiché sono stati ricevuti gli ACK.

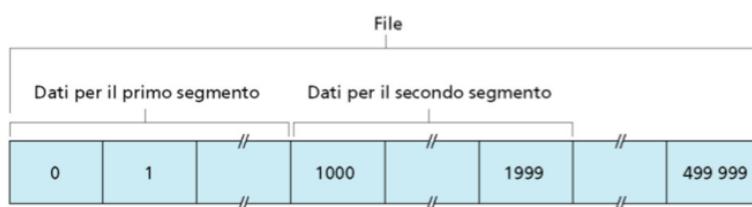
Se ci sono molti dati da inviare il ricevitore riempirà i suoi buffer riducendo a 0 la dimensione della sua finestra (la cosa verrà notificata al mittente che quindi sarà **in attesa** che la finestra si liberi).

Avendo una finestra opportunamente ampia (dipende dalla capacità del ricevitore) allora siamo in grado di saturare la rete e quindi di sfruttarla al massimo. Qualora non ci fossero datagram da inviare questo fatto dipende dal livello applicativo e non dal livello TCP: non è imputabile a TCP il non utilizzo della rete.

I numeri di sequenza: una precisazione

Parliamo spesso di sequence number/numeri di sequenza. Tali numeri vengono spesso indicati come interi incrementali ma in realtà per ragioni tecniche (TCP non sarebbe altrimenti in grado di ricostruire lo stream) un numero di sequenza è l'indice dell'ottetto successivo nello stream. Perciò se un interlocutore deve inviare 5000 Byte in segmenti da 1500 Byte (numerandoli da 0), avremo che il primo pacchetto avrà sequence number 1500 (i sequence number contenuti vanno da 0 a 1499, il prossimo è 1500). Il secondo avrà indice 3000 (per la stessa ragione).

Per ragioni prettamente didattiche però non rappresentiamo i sequence number per ciò che sono ma come cifre sequenziali che identificano l'intero datagram. Si tenga sempre presente questa astrazione che, nella realtà, non esiste.



Probing e finestra chiusa

Come sappiamo le finestre sono dinamiche: al terminare dei buffer da parte del ricevente il mittente avrà finestra zero (l'informazione gli giunge insieme ad un ACK di un qualche pacchetto inviato precedentemente). Dunque come può il mittente sapere che la finestra è nuovamente libera? Vengono inviati dei pacchetti di **probe** (pacchetti vuoti) dal mittente al destinatario che il destinatario riscontra ed intanto fornisce informazioni sulla sua finestra.

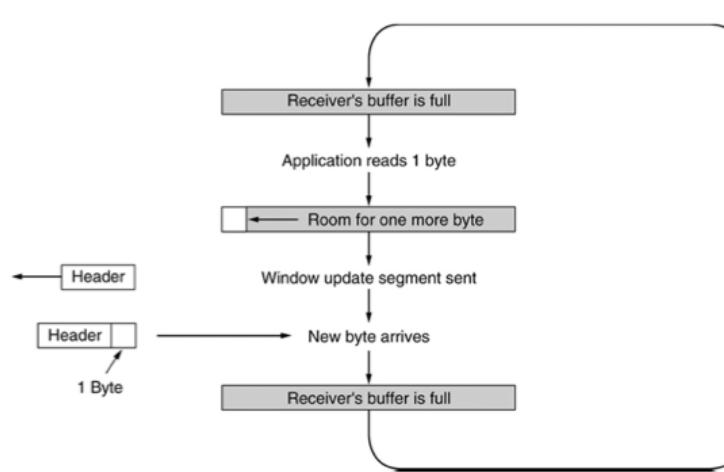
Il probe è anche utile per svolgere funzioni di keep-alive della connessione (TCP ha dei timeout che chiudono la connessione se non c'è attività, come vedremo meglio in seguito).

Migliorare l'efficienza della sliding window: algoritmo di Neagle

Il mittente non deve necessariamente trasmettere immediatamente i segmenti del livello applicativo ma potrebbe anzi decidere di ritardare leggermente l'invio per ragioni di ottimizzazioni (questo approccio può essere bypassato da particolari pacchetti come vedremo meglio in seguito).

Vengono effettuati due tipi di accorgimenti (oltre a quelli già visti):

- **Delayed acknowledgment:** viene ritardato l'invio di un ACK di 500ms sperando di ottenerne altri (si ricorda che gli ACK sono cumulativi).
- **Algoritmo di Neagle:** quando i dati arrivano al mittente a piccoli gruppi è sufficiente inviare il primo gruppo e bufferizzare il resto (in attesa di comporre un gruppo più sostanzioso ed evitare così l'overhead) finché non giunge l'ACK per il gruppetto precedente (e così via). L'algoritmo di Neagle non viene usato quando è richiesta particolare interattività.

La silly window syndrome

Questo problema si presenta quando i dati vengono passati a TCP in grandi blocchi ma il ricevente ha buffer molto piccoli (ad esempio 1 byte alla volta).

L'overhead della trasmissione è particolarmente alto poiché si possono trasmettere solamente dati grandi 1 byte (il destinatario continua a comunicare una finestra = 1).

La soluzione proposta è semplicemente quella di rifiutare una dimensione di finestra così piccola ma impone una soglia minima.

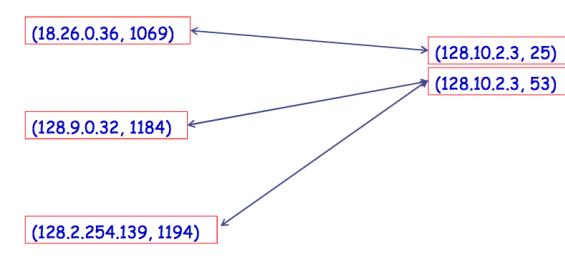
3.2) Porte, endpoint e connessioni

Anche TCP sfrutta il concetto di porta sebbene in modo differente rispetto a quanto visto in UDP. Le porte non sono buffer ma l'istanziazione delle risorse è effettuata a livello di **connessione**.

Le porte vengono assegnate a UDP così come a TCP, non c'è rischio di ambiguità perché è sempre il kernel del SO ad assegnarle (e quindi ad assegnarle se non in uso).

Una connessione è caratterizzata dai suoi **due endpoint**, coppie <Indirizzo IP, Porta>. Questo implica che siano **quattro numeri** a caratterizzare una connessione e quindi che più processi possano sfruttare la stessa porta TCP poiché gli interlocutori (gli altri endpoint) saranno sempre diversi: questo implica la possibilità di offrire servizi concorrenti su una stessa porta.

Alcuni esempi di endpoint/connessioni

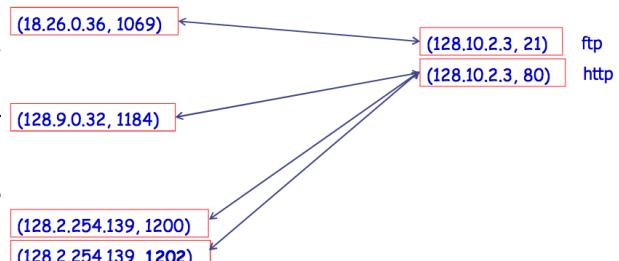


Come vediamo abbiamo dal lato server (a destra) due porte aperte (la 25 e la 53).

La prima connessione è con un endpoint soltanto, nessun dubbio.

Sotto abbiamo due connessioni da due client diversi ma non c'è ambiguità poiché la prima connessione è costituita fra <128.9.0.32, 1184> e <128.10.2.3, 53> mentre la seconda fra <128.2.254.139, 1194> e <128.10.2.3, 21>.

Nell'immagine a destra abbiamo un caso ancora più sottile, uno stesso client (128.2.254.139) ha aperto due istanze di un browser ed effettua due richieste http allo stesso server (128.10.2.3, 80). Non c'è ambiguità dato che il sistema operativo del client ha assegnato due porte diverse (1200 e 1202) alle due istanze del browser.



Apertura della connessione (attiva e passiva)

Abbiamo due diversi tipi di apertura di connessione. Una prima apertura è detta **passiva** cioè si comunica al SO che si vuole stare in attesa su una certa porta.

Altra tipologia di apertura è quella **attiva** che invece implica l'invio di una richiesta di connessione ad un certo host su una certa porta. TCP assegna una porta privata (dalla 1024 in su) a questo processo affinché possa stabilire la connessione e ricevere i suoi messaggi su quella porta.

Un server può ricevere quante connessioni vuole (finché ha sufficiente capacità) purché **descriva ogni connessione** ovvero mantenga la correlazione fra gli endpoint (ad esempio in una tabella).

Le well-known ports

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Nel momento in cui si richiede un servizio conosciuto si adopera una porta che è universalmente conosciuta come la porta di quel servizio. Le porte well-known sono scritte nel file etc/services (per sistemi unix).

Nel momento dell'avvio del sistema operativo viene attivato un demone che è in attesa su tutte quelle porte e lancia il demone della gestione della singola porta quando è richiesta una connessione inerente.

3.3) Il modello di servizio (socket)

Dal punto di vista dell'utente, TCP è accessibile grazie a particolari punti terminali detti **socket**. Un socket può essere usato per più connessioni contemporaneamente.

I socket sono visti dal punto di vista del sistema operativo come veri e propri file sui quali vengono effettuate operazioni di I/O. Questo è il grande vantaggio dei socket: creano astrazione fra la rete ed il sistema operativo.

Parleremo in seguito in maniera esplicita e dettagliata dei socket.

Messaggi urgenti: PUSH e URGENT

Nel momento in cui un'applicazione passa dei dati a TCP questo può decidere se bufferizzarli oppure inviarli direttamente. Talvolta l'applicazione richiede una particolare velocità di invio poiché si tratta di applicazioni real-time (terminali interattivi, desktop remoti, ecc.). In questi casi si sfrutta un flag contenuto all'interno del pacchetto TCP detto **PUSH**. Il flag PUSH attivo indica a TCP di non ritardare in nessun modo il pacchetto. Metaforicamente è come se consegnassimo il pacco direttamente sotto casa dell'altro TCP, con alta priorità.

Ben diverso è attivare il flag **URGENT** che invece, nella nostra metafora, fa sì che il postino suoni anche il campanello del destinatario. Dal punto di vista tecnico viene lanciato un interrupt dal sistema operativo verso l'applicazione: un **signal**. Se l'applicazione è progettata in maniera coerente (come ci si augura) l'applicazione sul destinatario raccoglierà il signal e compirà le azioni che ritiene opportuna.

Quando descriveremo tutti i campi del datagram TCP capiremo meglio anche l'implementazione di questa parte.

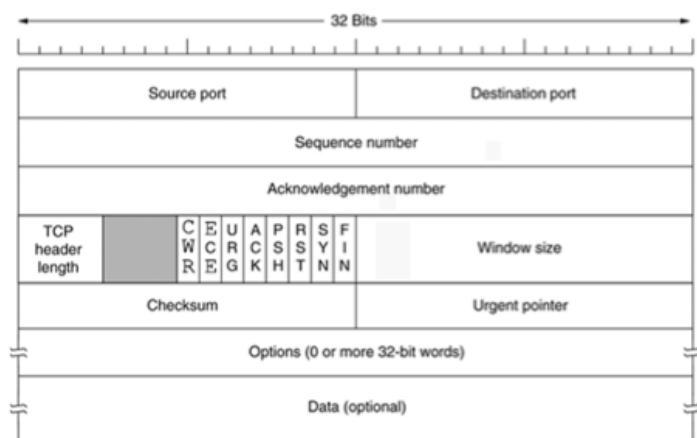
3.4) Il segmento TCP

Come detto in precedenza ogni ottetto di connessione ha un suo numero di sequenza (a 32 bit). Un segmento TCP inviato dal mittente al destinatario avrà un certo numero di sequenza x che verrà riscontrato nell'ACK inviato dal destinatario al mittente (infatti il significato dell'ACK è: tutto bene fino a x-1, attendo x). Il che è assolutamente coerente con quanto spiegato nel paragrafo *"i numeri di sequenza: una precisazione"*.

Un segmento consiste di **5 header di 32 bit** e TCP decide la dimensione dei segmenti con due limitazioni imposte da livelli sottostanti:

- Il pacchetto TCP deve essere contenuto nel payload IP (65.515 byte al massimo)
- Ogni collegamento ha una sua MTU (minimum transfer unit): quella di Ethernet ricordiamo essere 1.500 byte.

I campi del segmento



Quali campi abbiamo?

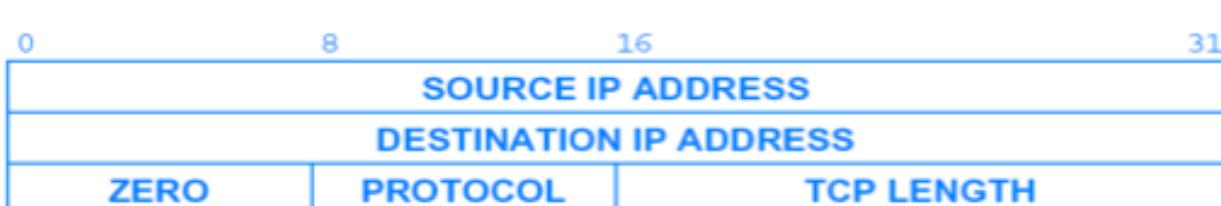
- **Source Port:** la porta della sorgente.
- **Destination Port:** la porta del destinatario.
- **Sequence number:** numero di sequenza univoco e identificativo del pacchetto (ne abbiamo parlato profusamente sopra). Ricordiamo solo che si tratta del **prossimo byte previsto**.
- **ACK number:** il pacchetto prevede sia il campo ACK che il campo sequence number poiché le comunicazioni sono spesso in due direzioni quindi diamo la possibilità alle due parti di inviare insieme ai dati il riscontro per i messaggi dell'altro peer. Questa operazione è detta **piggybacking**. Il campo ACK potrebbe anche non essere "sensato" nel caso in cui non vi siano messaggi da riscontrare: si userà una flag apposita per capire se il campo ACK è valorizzato o meno.
- **TCP Header Length:** la lunghezza dell'header di TCP (necessario poiché il campo Option è di lunghezza variabile).
- Otto flag:
 - **CWR:** usato per costringere l'interlocutore a diminuire la finestra di congestione (di cui parleremo dopo).
 - **ECN:** usato per segnalare all'interlocutore una congestione di rete.
 - **PSH:** presenza di dati push (come abbiamo descritto in precedenza). TCP è libero (entro certi limiti) di ritardare l'invio di un segmento per poter compiere ottimizzazioni e massimizzare l'utilizzo della rete (ad esempio accorpando più messaggi applicativi in un unico segmento). Se però il flag PSH è attivo questo sistema viene bypassato e il segmento viene buttato direttamente in rete. Allo stesso modo verrà trattato dal ricevente TCP che non lo bufferizzerà ma lo inoltrerà direttamente al livello applicativo. Attenzione però: nel caso di PSH non si forza in alcun modo l'applicazione a ricevere il pacchetto! Si velocizza solamente l'operazione di trasferimento.
 - **URG:** talvolta è necessario che certi messaggi viaggino **out of band**, fuori dal normale percorso dei pacchetti. Il trattamento è simile a quanto visto per PSH fino al momento dell'arrivo all'applicazione: in questo caso viene anche effettuato l'invio di un interrupt (signal) all'applicazione. Tale applicazione andrà a leggere quanto scritto nel pacchetto TCP fino ad un certo indice numerico: tale indice è scritto nel campo Urgent Pointer (di cui parleremo meglio dopo). Basti sapere che l'applicazione leggerà on the fly tutti i dati inviati da TCP fino all'Urgent Pointer.
 - **ACK:** bit che se settato a uno indica l'utilizzo di piggybacking e che quindi il campo ACK deve essere interpretato.
 - **RST:** viene usato per re-inizializzare una connessione (in seguito a qualche problema).
 - **SYN:** bit fondamentale usato per instaurare una connessione. Tale bit vale 1 nei primi tre messaggi di handshake e poi vale sempre 0.
 - **FIN:** usato per rilasciare la connessione.
- **Window Size:** informazioni sulle dimensioni della finestra. Una finestra a 0 indica che tutti i pacchetti fino ad ACK-1 sono stati correttamente ricevuti. Il ricevente può indicare che può nuovamente ricevere indicando un valore della window size maggiore di 0. Si noti la dimensione del campo è 16 bit quindi la finestra è al massimo di 2^{16} bit. Vi sono opzioni aggiuntive che permettono di allungare la finestra fino a 2^{32} (usata anche per gestire numeri di sequenza maggiori di 2^{32} : PAWS).
- **Checksum:** solito algoritmo per il controllo degli errori. Diremo alcune parole in seguito su questo campo.
- **Urgent pointer:** l'urgent pointer è usato nel caso in cui il pacchetto abbia abilitato il campo URG. Se così è, l'urgent pointer serve per indicare all'application fin quanto leggere della parte dati (qual è, cioè, la parte urgente dei dati). Questo numero è rappresentato come offset rispetto al **sequence number** corrente.
- **Options:** campo che permette l'utilizzo di funzionalità aggiuntive come il **window scale** che permette al mittente ed al destinatario di negoziare un fattore di scala per la finestra ad inizio connessione.

I riscontri in TCP

I riscontri di TCP sono considerati come **cumulativi** ma TCP deve sempre riscontrare ogni pacchetto arrivato. Perciò mandare un ACK per il sequence number x è come dire "fino a $x-1$ ho tutto". Ma se per caso ci arrivano i datagram successivi ad x saltando x stesso (quindi $x, x+1, x+2, x+3$, ecc.) per ognuno di essi manderemo comunque un ACK (TCP deve riscontrare ogni pacchetto) ma sarà un ACK sempre con x , cioè continueremo a dire "fino a $x-1$ ho tutto". I riscontri così organizzati hanno chiari vantaggi di facilità nell'essere generati e possibilità di risparmio (magari un ACK cumulativo riesce ad evitare la ritrasmissione di un datagram perché il suo ACK era andato perduto). Purtroppo però il punto negativo è che non abbiamo informazioni riguardo a tutto quel che ha il destinatario ma solo tutto quel che ha di **contiguo** il destinatario!

Un particolare checksum: lo pseudo-header TCP

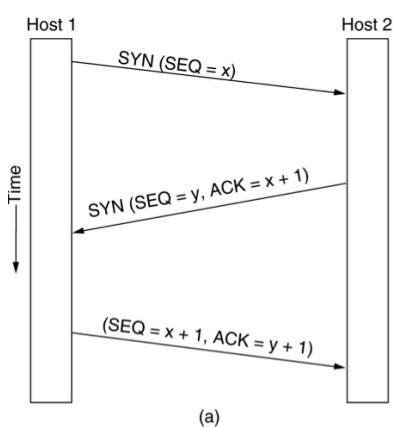
Anche qui si usa uno pseudo-header durante il calcolo del checksum per poter avere anche l'indirizzo IP innestato nel controllo.



Lo pseudo-header è del tutto simile ma il campo PROTOCOL vale ora 6.

3.5) Setup di una connessione in TCP

Come abbiamo già visto, per stabilire una connessione a livello transport viene adottato un 3 way handshake.



L'host 2 (in questo esempio il server) sarà in attesa passiva (avrà effettuato una **LISTEN**) quando un host 1 (il client) effettuerà una richiesta **CONNECT** mandando il suo numero di sequenza di partenza, x .

Tale numero, per motivi di sicurezza, (IP spoofing) deve essere necessariamente un numero pseudocasuale imprevedibile.

Nella CONNECT il flag di **SYNC** varrà 1 e il flag di **ACK** varrà 0. Si noti che sarà l'unico frame dell'intera connessione che avrà contemporaneamente questi due parametri settati in questo modo (questo è molto usato dai firewall per bloccare connessioni dall'esterno).

Il server, ricevuta la CONNECT, risponderà inviando il suo numero di sequenza iniziale e riscontrando il numero di sequenza inviato dal client (con la solita tecnica del "+1" e quindi dicendo che si aspetta $x + 1$). Il flag SYNC sarà ancora 1 ma il frame ACK sarà invece 1.

Infine, il client riceve la risposta e conferma definitivamente concludendo il protocollo 3 way handshake e instaurando la connessione.

Se qualcosa dovesse andare storto, si ricorrerà al flag RST.

Nella maggior parte delle applicazioni, comunque, è il SO a gestire l'intera fase di handshaking: avverte TCP solamente una volta che le due parti si sono accordate e la connessione è definitivamente stabilita. Anche i buffer vengono allocati solamente quando la connessione è "up": il SO si limita a definire i descrittori (le tabelle di correlazione fra gli endpoint) della connessione.

Il problema del syn flood

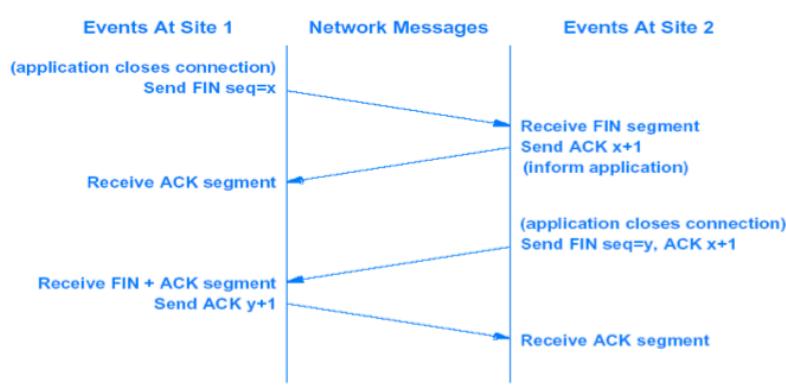
Questa struttura ha un problema di base: il server deve mantenere "pendenti" tutte le richieste che riceve finché non si conclude il 3 way handshake. Quindi un avversario potrebbe decidere di inviare (eventualmente con IP spoofati) moltissime richieste di connessione ad un server il quale, per definizione, dovrà rispondere. Un numero troppo alto di richieste può però, come è prevedibile che sia, portare il server al collasso.

Questo tipo di attacco è detto **syn flood**.

Nel caso vi siano troppe connessioni entranti il SO potrebbe decidere di rifiutarle direttamente senza neanche far interagire TCP.

3.6) Rilascio di una connessione in TCP

Giacché le connessioni TCP sono full-duplex, ogni direzione deve essere rilasciata indipendentemente dall'altra.



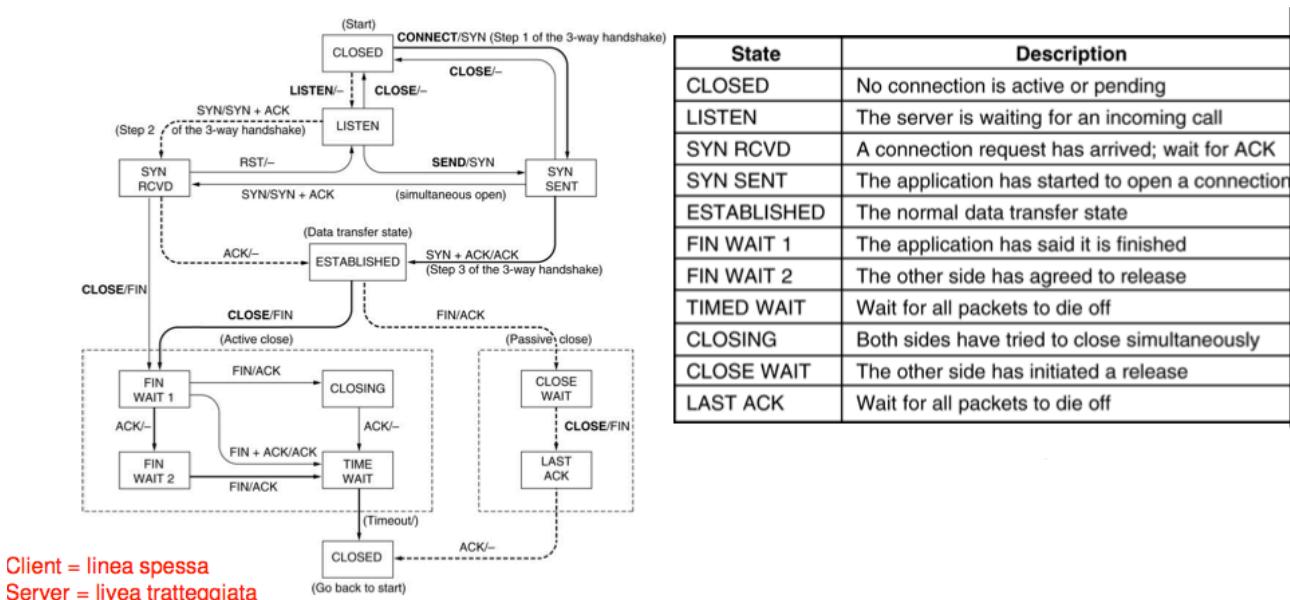
Per effettuare questa operazione viene sfruttato il flag **FIN**: il significato di questo flag a 1 è: "non ho più nulla da trasmettere, attendo riscontro".

Nel momento in cui TCP riceve l'ACK per il segmento che conteneva il flag FIN a 1 la connessione viene chiusa definitivamente ma quel TCP per un certo periodo di tempo continuerà a riscontrare i pacchetti dell'altro

interlocutore (il quale potrebbe avere ancora qualcosa da mandare).

Come abbiamo già discusso (mentre affrontavamo il problema per un generico protocollo transport) non possiamo garantire che i dati non vengano persi nel 100% dei casi.

3.7) TCP come macchina a stati



3.8) La gestione dei timer

Una delle parti più interessanti (ed innovative) di TCP è la gestione dei timer. Tale gestione è stata migliorata nel tempo quindi anche noi vedremo diverse soluzioni mano a mano sempre più fini e mirate al problema, ripercorrendo, ove possibile, l'evoluzione storica.

I timer di TCP

TCP ha diversi timer che assolvono diverse funzioni (tutte decisamente importanti).

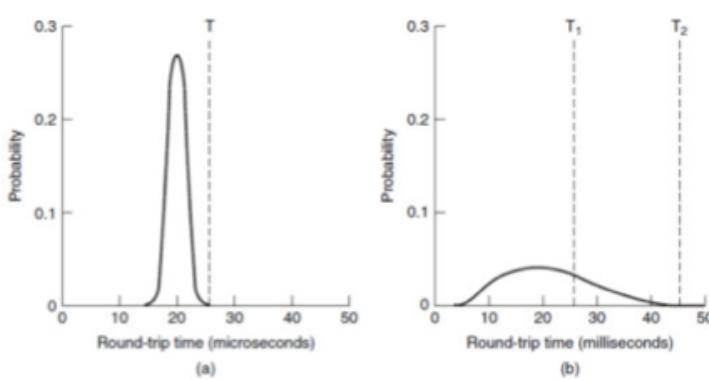
- **Timer di persistenza:** intervallo di tempo che trascorre fra due window probe (per sapere la dimensione della finestra quando l'ultima informazione era window = 0). I tempi di window probe sono abbastanza statici, il timer viene raddoppiato ogni probe (per evitare traffico inutile).
- **Timer di keep alive:** viene usato dal mittente per mantenere "viva" una connessione che per un periodo di tempo non ha avuto alcun flusso di dati in trasmissione.
- **Timer di chiusura di connessione:** usato nella chiusura di una connessione (se non si riceve riscontro per il pacchetto con FIN = 1): il tempo è pari al doppio della lunghezza di vita massima di un pacchetto.
- **Timer di retransmission timeout (RTO):** sicuramente il timer più complesso ed interessante di TCP. È fortemente dinamico e modificato dalle informazioni che la rete fornisce a proposito della congestione.

Il timer RTO

Mentre i primi tre timer sono abbastanza intuitivi, studiamo nel dettaglio il quarto.

L'**RTO** viene attivato nel momento in cui viene spedito un segmento mentre viene disattivato nell'istante in cui si riceve un ACK per il segmento corrispondente.

Se invece il timer scatta avviene una ritrasmissione per il segmento cui il timer è associato e il conteggio viene riavviato.



Abbiamo già parlato di come l'RTO sia una lancetta molto **sensibile** di TCP che può facilmente far scoppiare una congestione sulla rete. La scelta dell'intervallo di timeout è un chiaro "**rubinetto**" di pacchetti sulla rete: un timer troppo lungo ritarderebbe inutilmente una ritrasmissione necessaria, un timer troppo corto genererebbe ritrasmissioni per pacchetti che in realtà non sono andati persi ma bensì sono solamente accodati in qualche router sulla rete.

L'RTO è molto legato all'RTT: un RTO perfetto sarebbe poco dopo l'RTT medio (come indicato in T nella prima immagine) laddove la distribuzione dei valori dell'RTT sia "favorevole" (capiremo sotto cosa significa).

Le due figure rappresentano rispettivamente un RTT "che ci piace", con una **varianza molto piccola** (poca dispersione di valori). Impostando l'RTO al punto T avremo un numero irrisorio di casi in cui un pacchetto ancora in coda venga ritrasmesso.

La seconda figura invece è un caso terribile! La varianza è molto grande ed è difficile stabilire un punto di ottimalità dove mettere l'RTO.

È anche necessario ricordare che i **ritardi sono variabili per via della congestione** (i timer [basati sui ritardi] agiscono sulla congestione e la congestione sui ritardi).

Un primo approccio è stato quello di usare una variabile che stimasse l'RTT, chiamata **SRTT** (smoothed RTT) così definita:

$$SRTT = SRTT + (1 + \alpha) \cdot R$$

Ogni connessione tiene conto di un SRTT (è un bel costo!).

In sostanza per ogni segmento inviato (quindi ogni timer attivato) se l'ACK arriva in tempo, si tiene conto di quanto tempo è trascorso (ovvero dell'RTT per quel segmento) e chiamiamo questo tempo R. Dopodiché si stabilisce un α che pesa il vecchio SRTT rispetto alla nuova informazione acquisita (R). Tendenzialmente $\alpha = 7/8$ (ovviamente viene considerato maggiormente il dato passato).

Una volta acquisito l'SRTT si stabilisce la RTO in base ad esso: si è deciso (così, "a braccio") di impostare:

$$RTO = SRTT \cdot 2$$

Col passare del tempo, però, ci si è resi conto che è molto più interessante osservare anche la **varianza** (basti osservare le figure di cui abbiamo parlato precedentemente). Si definisce così una nuova variabile RTTVar:

$$RTTVar = RTTVar + (1 - \beta) \cdot |SRTT - R|$$

Con un β che ha la stessa applicazione dell' α di prima. Tendenzialmente $\beta = 3/4$. L'RTO viene così ri-definito:

$$RTO = SRTT + 4 \cdot RTTVar$$

Anche qui, la scelta del numero 4 è arbitrario e derivante da sperimentazioni.

Abbiamo trovato un bel modo di regolare l'RTO. Perfetto! Non rimane che decidere che fare di quei timer che sono scattati in mancanza di un ACK per il segmento associativo.

Supponiamo che un datagram con numero di sequenza x venga spedito e perso. Dopo un certo tempo scatterà il timer e quindi si procederà con una ritrasmissione dello stesso. A questo punto viene ricevuto un ACK per x: si riferisce alla prima copia del segmento o alla seconda? Magari il primo pacchetto era semplicemente rimasto bloccato nella coda di qualche router!

L'**algoritmo di Karn** risolve il problema alla radice suggerendo di ignorare nel nostro calcolo i valori dei timer che sono coinvolti in ritrasmissioni. Inoltre suggerisce di:

- Raddoppiare il timeout per ogni ritrasmissione successiva.
- Calcolare l'RTO diminuendone continuamente il valore stesso finché non si ha una buona stima con l'RTTVar.

3.9) La gestione della congestione

Abbiamo visto come regolare l'RTO per evitare di generare inutili segmenti e quindi generare congestione. Ma la congestione non è solo causata da pacchetti inutili, bensì anche da trasmissioni utili: se molti calcolatori sfruttano la rete, la congestione esiste e quindi deve essere gestita in modo più preciso e sicuro.

Nell'affrontare l'argomento vedremo mano a mano delle migliorie apportate nelle varie versioni di TCP.

Le cause della congestione

La congestione della rete può verificarsi per l'effetto combinato di una o più di queste situazioni:

- Più interfacce del router mandano pacchetti sulla stessa interfaccia di rete.
- Il numero di pacchetti in arrivo ad un router è maggiore del tasso di uscita sulla rete degli stessi (i pacchetti vengono accodati).
- Al termine dei buffer, il router inizia a scartare pacchetti.

La collaborazione di TCP

Gli endpoint non sono interessati a conoscere la situazione di congestione della rete (si tratta di un problema di livello inferiore). Ma è pur vero che TCP non può ignorare totalmente la questione poiché **il suo aiuto è prezioso** per evitare che la situazione peggiori. Ricordiamo che non è TCP a generare traffico ma sempre il livello applicativo! Quindi TCP "fa quel che può".

La finestra di congestione

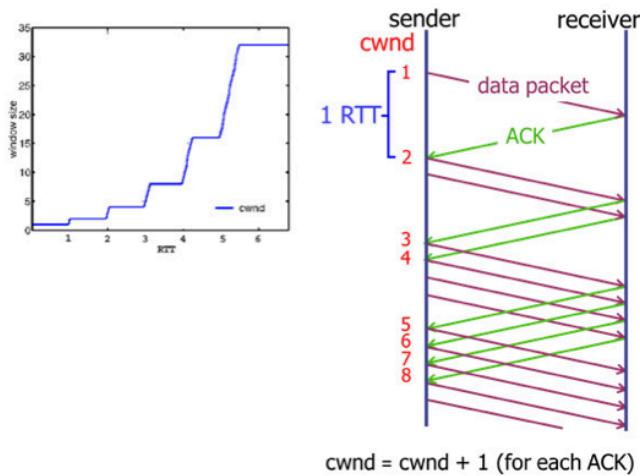
Abbiamo oramai acquisito il concetto di finestra scorrevole: ne abbiamo usata una per risolvere il problema del controllo di flusso. Si adotta la stessa idea anche qui (quindi in realtà TCP implementa due finestre, una per il controllo di flusso e una per la congestione: le due **finestre** sono **sovraposte** e chiaramente la finestra adottata sarà la **minore** fra le due).

Percepire la congestione

Come fa il nostro TCP a sapere che c'è congestione? Vi sono diverse tecniche ed è a seconda del metro scelto che avremo versioni di TCP diverse. La più accreditata è quella che si basa sull'idea di Jacobson: **la perdita di un pacchetto è segnale di una congestione**.

Lo slow-start: non accontentarsi di AIMD

Abbiamo già studiato AIMD e sappiamo bene come funziona. Ma una regola AIMD ci può mettere anche parecchi secondi a stabilirsi e a trovare un punto ottimale. Chiaramente non è risolutivo adottare una finestra gigantesca fin da subito poiché genereremmo una congestione immediata.



Perciò viene usata una tecnica chiamata **slow-start** (slow rispetto alla finestra enorme citata sopra) che si basa su un concetto semplice: facciamo sì che la finestra cresca esponenzialmente fino a un certo limite e poi aggiustiamo il valore tramite AIMD.

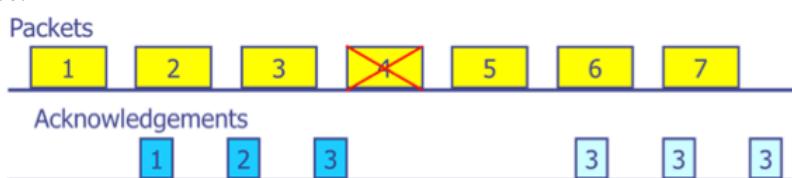
Per ogni segmento che riceve un ACK viene incrementata di 1 la finestra: questo significa che ogni volta che la finestra viene riempita avremo il doppio dello spazio nella finestra rispetto a prima.

La soglia imposta è detta **slow-start threshold** e TCP continua a far crescere la finestra finché non c'è un timeout oppure la soglia viene superata. **Ogni volta che avviene un timeout** il valore della slow-start threshold è dimezzato.

Fast-retransmission: ACK ripetuti, un segnale di congestione

Un'altra grande ottimizzazione apportata a TCP nel corso del suo sviluppo (versione **Tahoe** 1988) è stata quella di considerare la ricezione di più ACK uguali come un messaggio di congestione (non così grave) in aumento.

Invece di attendere il timeout si cerca di **anticipare la ritrasmissione** una volta che ci si è resi conto che un segmento è stato perso.

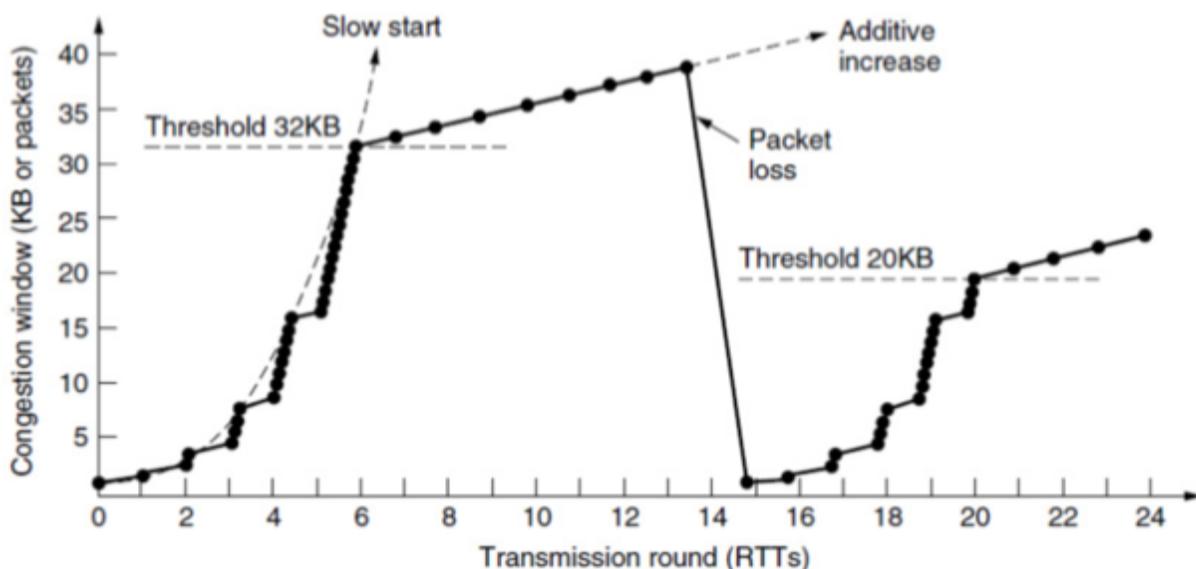


Dopo **tre ACK uguali ricevuti per lo stesso segmento**, viene ritrasmesso il segmento richiesto (ricordiamo la questione che un ACK = x significa avere tutto fino a $x - 1$ e ci aspetta x: ricevere quindi tre ACK per x, significa che x è andato perso).

I pacchetti potrebbero anche arrivare in ordine sbagliato generando così degli ACK ripetuti allo stesso modo (si inviano pacchetti da 1 a 5, il 3 viene ritardato e arrivano 1, 2, 4, 5: a questo punto sarà ripetuto l'ACK per 2 fino all'arrivo di 3) ma questo evento è abbastanza raro e la soglia dei tre ACK riesce ad arginare leggermente questa situazione già di per sé rara.

Si noti che la ripetizione di tre ACK è sì indice di congestione ma meno "allarmante" di quanto sia lo scattare di un timeout: nel primo caso almeno gli ACK arrivano (quindi la perdita potrebbe essere stata un evento sporadico) mentre nel secondo ci sono probabilmente problemi più grossi sulla rete poiché l'interlocutore non riesce neanche ad inviare ACK!

Una volta effettuata la fast-retransmission allora si tornerà ad eseguire la tecnica slow-start fino al threshold (che sarà stato dimezzato). La figura sotto riassume quanto detto fin ora:



Abbiamo un andamento con segmenti di 1KB. Abbiamo una crescita iniziale con slow-start fino a giungere al threshold (finestra a 32 KB) dopodiché si procede con un additive increase fino a 40KB.

A questo punto si perde un pacchetto, esso viene ritrasmesso (fast-retransmission) e si ricomincia con lo slow-start ma con un threshold della metà ($40\text{KB} / 2 = 20\text{KB}$).

Fast-recovery: l'ultima ottimizzazione

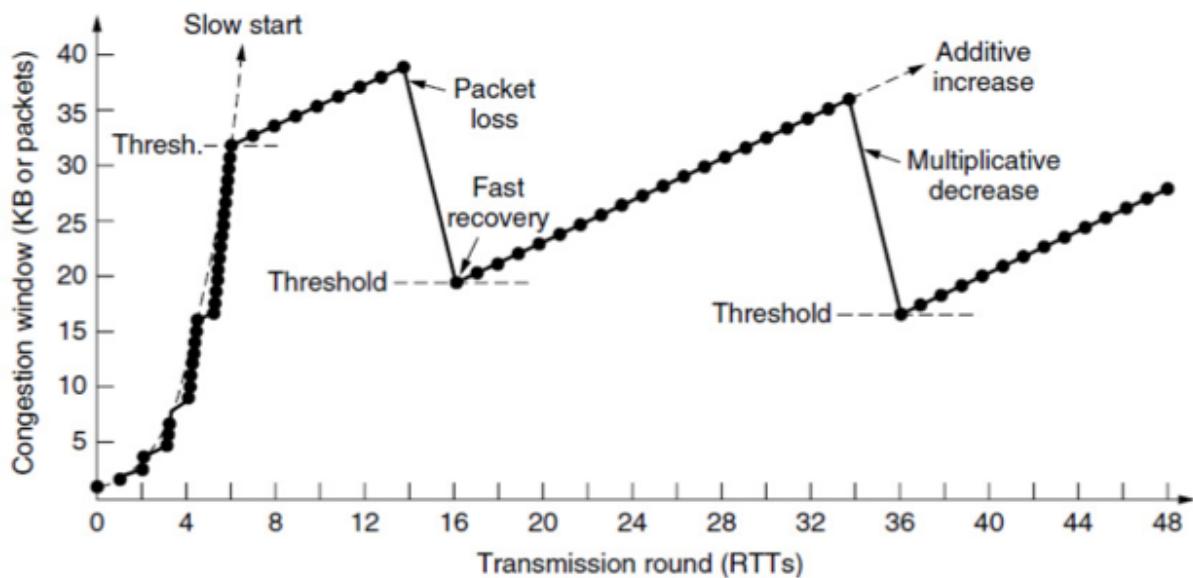
Abbiamo già fatto enormi progressi: la versione Tahoe di TCP porta con sé slow-start, collision avoidance, algoritmo di Karn, RTO basato sulla varianza dell'RTT, fast-retransmission.

Jacobson pensa però che sia possibile fare ancora di meglio. In seguito alla perdita di un pacchetto, perché ridurre a 0 la finestra per poi tornare velocemente al threshold? Sarebbe più sensato, invece, **bloccare** la finestra **direttamente al threshold!**

Questa è l'idea del fast-recovery. In sostanza si cerca di non svuotare totalmente la pipe e quindi di non sprecare lo sforzo fatto dalla prima fase di slow-start.

L'immagine a seguire illustra tutte le tecniche di cui abbiamo parlato: l'andamento a dente si sega ci fa capire come le tecniche siano davvero ottimizzanti e migliorino la regolazione della finestra di congestione in modo veloce e dinamico.

Sarà la versione Reno (1990) a implementare la fast-recovery.



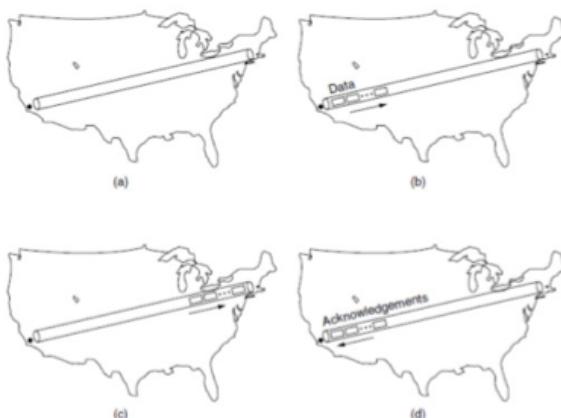
3.10) Problemi di prestazioni nelle reti

Il nostro discorso su TCP si conclude qui: le versioni successive hanno provato a cambiare indice per cogliere la congestione ma con scarsi risultati.

Inoltre le reti così come tutto il mondo tecnologico dovranno svilupparsi nel tempo: il problema delle prestazioni è generato tipicamente da una mancanza di risorse. Non è neanche così facile capire dove sia il problema prestazionale all'interno di una singola rete: **la misurazione delle prestazioni è complessa**.

Protocolli per reti ad alta capacità

Vi sono poi problematiche relativamente ai protocolli per le reti ad alta capacità (sono necessarie grandi dimensioni per la finestra di controllo di flusso).



Ad esempio, si voglia inviare un pacchetto di 64KB da San Diego a Boston con lo scopo di riempire il buffer del ricevente.
(si ha un collegamento a 1Gpbs e un ritardo su una direzione di 20 ms).

A $t = 0$, abbiamo che il canale è vuoto (a).

Dopo 500 μ s (b) tutti i segmenti sono sulla linea e il ricevente attende l'arrivo degli ACK (la finestra dall'altro è piena).

Dunque, dopo 20ms il primo segmento arriva (c) e il primo ACK viene generato. Ci metterà altri 20ms (40 totali) per tornare al mittente (d).

La linea di trasmissione è stata usata per 1,25 ms su 100, con un

efficienza pari al 1,25%.

Decisamente poco! Questo perché abbiamo inviato pochi dati, non abbiamo riempito la pipe!

Bisogna considerare il prodotto **banda-ritardo** ottenuto moltiplicando la banda per l'RTT. In questo caso avremmo $b \times r = 40$ milioni di bit. Ecco quindi che un blocco di soli 64KB (che è solo mezzo milione di bit) è troppo piccolo e quindi non sfrutta a pieno l'efficienza della rete.

Ci vorrebbero almeno finestre da 5MB per reti come quella appena esposta.

4) Il DNS

Affrontiamo ora un argomento importante, il **domain name system** (DNS).

Abbiamo oramai capito come gli indirizzi IP siano fondamentali e molto "parlanti" per le macchine: fanno sì che tutto il sistema di indirizzamento funzioni a dovere.

Ma mettendoci nei panni dell'utente, possiamo subito capire che gli indirizzi IP non sono altrettanto comodi per definire un certo calcolatore.

4.1) Gli obiettivi del DNS

Il DNS si prende carico di **associare un nome ad un insieme di attributi/proprietà conoscendo il nome ed il tipo di attributo desiderato**. Quindi il DNS non è solamente "quella cosa che associa un indirizzo IP ad un nome", sarebbe riduttivo e non corretto, ma realizza i seguenti obiettivi:

- **Associa nomi a proprietà** (l'indirizzo IP è soltanto un esempio di proprietà, ve ne sono altri!).
- Dato un **nome** ed un **tipo** fornisce il **valore** di un attributo.
- Ci dice se a un **indirizzo IP è associato un certo nome**.
- Associa un **nome ad un servizio** così che gli utenti possano ritrovare il servizio anche se hanno cambiato locazione (ad esempio www.educ.unito.it).
- Permettere ad ogni amministratore di una rete di dare nomi a **indirizzi, servizi e proprietà** che sono sotto il suo controllo.
- Permette ad un amministratore di far sapere agli altri che si possono **fidare** di certe macchine sotto il suo controllo.
- Fornisce un **meccanismo** che sia **scalabile** su tutta Internet.

4.2) I nomi

Quello di attribuire un nome a qualcosa è un problema che affrontiamo fin dalle scuole elementari. Un nome permette l'identificazione di un qualcosa ed ha caratteristiche comuni e differenti rispetto ad un indirizzo.

Un nome è **costruito su un alfabeto finito** (come l'indirizzo), ha una **struttura dettata** da regole ben definite (come l'indirizzo) ed è **semplice** per chi lo pronuncia (a differenza dell'indirizzo che invece è semplice per chi deve localizzare qualcosa!).

Spazio dei nomi piatto

I nomi sono utili se sono arbitrari, ovvero se "posso scegliermeli io". Questo approccio sembrerebbe sensato ma solamente nell'ambiente di una piccola LAN: quando i nomi sono **globali** invece la situazione si fa più complessa...

Due **nomi diversi** devono identificare **entità diverse**! Per questo il controllo dei nomi sarà "centralizzato" da una **autorità centrale**. Appena Internet ha iniziato ad espandersi l'idea di uno spazio di nomi piatto è fallita poiché ingestibile.

Spazio dei nomi piatto globale

La soluzione che è stata invece introdotta è la seguente:

- Su ogni macchina è presente un file che mappa la corrispondenza fra nomi ed indirizzi (/etc/hosts nei sistemi Unix).
- Tali file sono interrogabili per eseguire il mapping.
- Esiste una macchina centrale in cui è serbata una copia master di questo file.
- Nel momento in cui il file necessita di modifiche, queste avvengono sulla copia master.
- Colui che opera sul file è l'autorità centrale di cui abbiamo parlato prima.
- Per aggiungere un nome questo non deve esistere all'interno del file (univocità!).
- Periodicamente la copia master viene trasferita su ogni macchina dell'Internet affinché venga aggiornata.

Questa soluzione **non** si è rivelata **vincente** poiché la gestione dell'aggiornamento dei file è troppo complessa e costosa. Il file /etc/hosts esiste ancora ed è un "rimasuglio storico", viene comunque interrogato prima di interpellare il DNS (infatti ad esempio troviamo localhost).

Nomi gerarchici

L'idea è quella **decentralizzare** ancora di più la gestione dei nomi. Abbiamo diversi siti che amministrano lo spazio dei nomi ottenendo così una responsabilità distribuita per il mapping fra nomi e proprietà. Non sono sufficienti copie master parziali, è necessario anche **decentralizzare il controllo** sull'univocità dei nomi.

Avremo perciò una moltitudine di autorità che non solo si prenderanno la responsabilità di gestire il dominio dei nomi ma anche quello di effettuare il mapping fra nomi e proprietà.

Come è possibile tutto ciò?

Ogni dominio dispone di un nome univoco ed ogni altro nome sotto quel dominio conterrà il nome "padre" in una posizione ben stabilita così da essere riconoscibile ed univoco. Da ciò deduciamo che:

- Il nome è composto da parti.
- Il nome è in qualche modo strutturato.

Se pensiamo a questo sistema come continua suddivisioni di domini in sotto-domini abbiamo ottenuto una struttura ad albero che può propagarsi per tutta la rete Internet.

Quindi riassumendo abbiamo che lo spazio di nomi è partizionato, **ogni partizione** è individuata **da un nome** e tutti i nomi di una partizione contengono il nome della partizione stessa. Perciò da un nome possiamo risalire alla sua partizione, a chi è assegnata la delega di autorità e la responsabilità amministrativa.

Stiamo però chiedendo ancora un po' troppo: pensare che ogni nodo sia una autorità è una richiesta troppo stretta. Ci piacerebbe poter avere **alcuni nodi che sono autorità ed altri che invece non lo sono**. Ma sicuramente **ogni nodo che è autorità deve anche eseguire il mapping** (responsabilità di traduzione).

Come partizioniamo?

Una buona base potrebbe essere quella geografica. Chiaramente vi sono più siti nell'Internet ed ogni sito avrà un nome univoco sitoX e potrà assegnare solamente nomi di questa forma:

nome-locale.sitoX.

Il punto separa i componenti del nome, quindi all'amministratore non resta che scegliere come impostare/suddividere la parte "nome-locale".

Capiamo subito che c'è un problema: la granularità dei siti è troppo fine per creare un'astrazione utile, dobbiamo avere un livello in più. Ciascun amministratore sarà libero di assegnare nomi nella forma:

nome-locale.gruppoY.sitoX.

Comunque, in Internet la metodologia più usata è il **partizionamento per linee organizzative** e non geografiche.

4.2) Internet Domain Names

La struttura che gestisce tutto questo sistema gerarchico è il DNS. Esso ha due aspetti importanti: uno più **astratto** (specifica la sintassi dei nomi, le informazioni associabili ai nomi, le regole per delegare l'autorità) e uno più **concreto** (specifica l'implementazione di un sistema di calcolo concreto per il mapping).

Un **nome di dominio** è costituito da componenti separate da un punto dette **labels**. Ogni label può essere lungo al massimo 63 caratteri e l'intero nome 255. I caratteri accettati sono le lettere, le cifre decimali, il "-" e il ".".

Ad esempio

cs.purdue.edu.

dispone di tre etichette (cs, purdue ed edu). Al suo interno troviamo tre diversi **domini** (cs.purdue.edu., purdue.edu. e .edu.) Come spiegato sopra i domini sono in ordine (da destra verso sinistra) di "località" incrementale. I nomi **non sono case sensitive**.

Nomi di dominio Internet (ufficiali/non ufficiali)

Come abbiamo detto ogni organizzazione può definire i propri nomi come preferisce. Ma in realtà viene sempre adottato uno schema "standard".

Il dominio top level (quello più ad alto livello) è stato definito in questo modo:

Domain Name	Meaning
aero	Air transport industry
arpa	Infrastructure domain
biz	Businesses
com	Commercial organization
coop	Cooperative associations
edu	Educational institution (4-year)
gov	United States government
info	Information
int	International treaty organizations
mil	United States military
museum	Museums
name	Individuals
net	Major network support centers
org	Organizations other than those above
pro	Credentialed professionals
country code	Each country (geographic scheme)

Osservando la tabella notiamo che vi sono due tipologie di domini top level. Quelli **organizzativi** (edu, gov, ecc.) e quelli **geografici** (it, fr, ecc.)

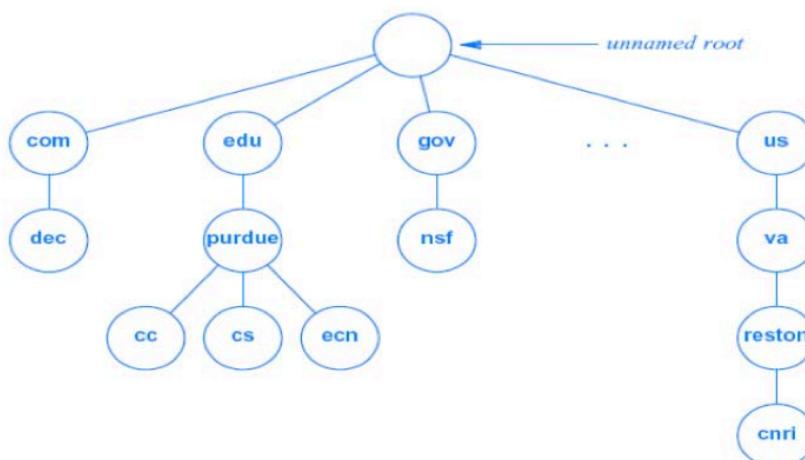
Per quanto detto fin ora se vediamo un indirizzo così costituito:

xINU.cs.purdue.edu.

possiamo immaginare che l'amministratore dell'università di Purdue abbia delegato una divisione cs (probabilmente il dipartimento Computer Science) di gestire quel dominio. Tale dipartimento avrà deciso di assegnare ad un particolare gruppo qualche risorsa, identificata col nome xINU.

L'abbiamo già detto, ma non dobbiamo pensare che ad un nome corrisponda una macchina!

Si ricordi inoltre che un'autorità più top level può decidere di assegnare il nome ad un'altra autorità ma riservarsi il diritto della gestione dei nomi su quel sottodominio. Ecco quindi come appare parte della gerarchia dei nomi di dominio di Internet:



4.3) Nomi e proprietà

Chiariamo meglio il punto già più volte citato: un nome non corrisponde necessariamente ad un indirizzo IP bensì ad una generica risorsa che ha un tipo (ed l'indirizzo IP è una di queste, ma solo un esempio).

Il DNS offre un servizio generico di **mapping fra nomi e proprietà** le quali hanno un **tipo**. Ogni elemento informativo è chiamato **resource** ed è mantenuto in **resource record** da parte del DNS.

Viene da sé che quando un client effettua una richiesta al DNS non si limiterà a fornire il nome ma dovrà anche dire il tipo della resource alla quale è interessato! Questo è ovvio poiché il nome, di per sé, non ci dice se ci riferiamo ad una macchina o meno. Ad esempio se:

gwen.purdue.edu.

corrisponde ad una macchina allora avrà un indirizzo IP, mentre invece il nome:

cs.purdue.edu.

potrebbe riferirsi solamente ad un dominio e quindi non avere alcun indirizzo IP assegnato.

Lista dei tipi di resource

Dunque quali resource possono essere assegnate ad un nome?

Tipo	Significato	Contenuto
A	Host Address	Indirizzo IP di 32 bit
CNAME	Canonical Name	Nome canonico per un alias
HINFO	Hardware Info	Nome della CPU e del sistema operativo
MINFO	Mailbox Info	Informazioni sulla mailbox o su una mail list
MX	Mail Exchanger	Nome dell'host che funge da mailbox per il dominio
NS	Name Server	Nome del server che ha autorità sul dominio
PTR	Pointer	Traduce da indirizzo IP a nome
SOA	Start Of Authority	Campi multipli che specificano quali parti della gerarchia dei nomi un server implementa, cioè ad esempio se specifico di.unito.it otterrò tutti i server che hanno autorità per quel tipo di dominio.
TXT	Arbitrary Text	Testo ASCII non interpretato

4.4) Il mapping fra nomi ed elementi informativi

Abbiamo compreso che esiste quindi una correlazione fra un nome e la (le) sue proprietà che hanno tipo differente fra loro. Ma il DNS specifica anche un sistema per effettuare a tutti gli effetti il mapping fra i nomi e gli elementi informativi, tale sistema è inoltre:

- **Efficiente**: la maggior parte dei nomi viene mappata localmente senza generare traffico Internet.
- **Affidabile**: il fallimento di un server non impedisce al sistema di operare correttamente.
- **General purpose**: non è ristretto al funzionamento per certi nomi di macchine o per indirizzi IP.
- **Distribuito**: consta di un insieme di server che operano in siti diversi e sotto la responsabilità di gestori diversi ma che cooperano per risolvere il problema del mapping fra un nome ed i suoi elementi informativi.

DNS: un'applicazione client-server

Ogni server che coopera per effettuare il mapping è detto **name server** mentre l'applicazione che vi gira sopra è detto DNS (è quindi un'applicazione client-server).

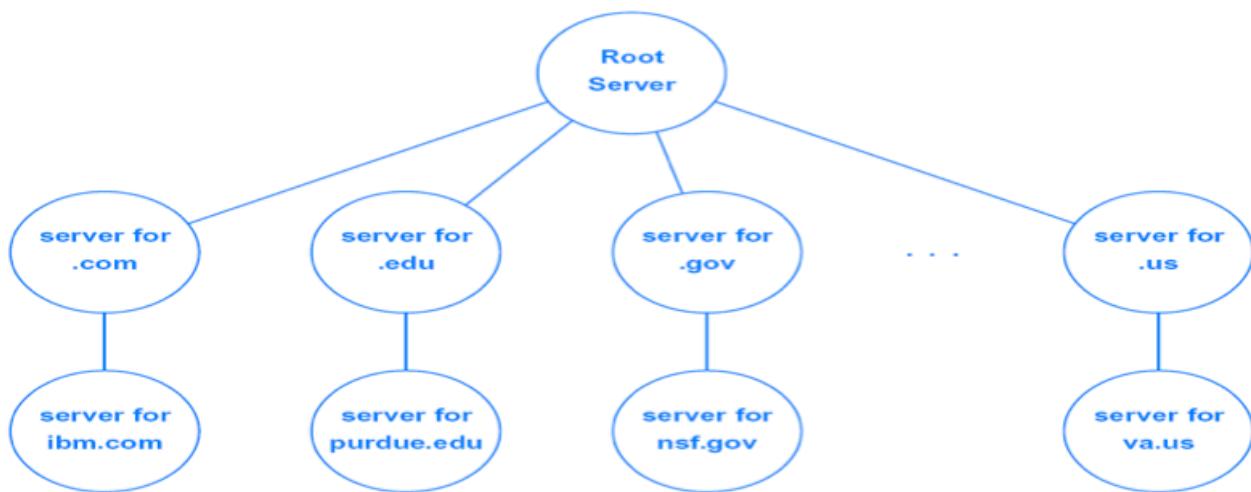
Il client è quindi il **name resolver** ed è pensabile come una libreria (una system call) chiamata dalle applicazioni permettendo così la comunicazione con un name server (al fine di ottenere una risorsa correlata ad un nome di dominio).

Ad esempio un browser sfrutterà un name resolver per tradurre ciò che l'utente scrive all'interno della sua barra dell'URL (chiederà chiaramente una risorsa di tipo A correlata al nome espresso nella barra) al fine di ottenere l'indirizzo IP del server dove risiede la pagina richiesta.

L'albero dei name server

Ci chiediamo dunque come faccia un name server a trovare la giusta traduzione per un nome di dominio.

Come prima approssimazione vediamo l'insieme dei name server organizzati ad albero. Tale albero corrisponderà (**in parte**) alla gerarchia dei nomi, ovvero non vi sarà un server per ogni nome di dominio esistente nel web.



La radice dell'albero conosce tutti i domini di primo livello e sa quale server è autorevole (cioè in grado di risolvere) rispetto ad una certa gamma di domini. Sicuramente nei livelli più alti dell'albero avremo un server per ogni dominio ma scendendo invece la situazione cambierà: un server potrà gestire più nomi di dominio contemporaneamente (ecco perché l'albero corrisponde solo in parte alla gerarchia dei nomi).

Questo accade per due motivi:

- Alle grandi organizzazioni fa comodo avere un server solo in grado di gestire tutti i sottodomini (si evitano conflitti fra nomi).
- Nel caso di domini molto piccoli (molto in basso nell'albero) si potrebbe non disporre delle risorse necessarie per amministrare un name server.

Ovviamente il root server non sarà unico.

4.5) Risolvere un nome di dominio (primo approccio e concetti)

Nonostante l'albero dei server non sia profondo come quello dei domini, abbiamo comunque una struttura mastodontica da gestire. Il processo di risoluzione di un nome sarebbe quindi molto pesante se non lo trattassimo in modo intelligente.

Il primo approccio che verrebbe in mente a chiunque è quello di partire, per ogni traduzione, dal root server e leggendo il nome da destra verso sinistra risalire al server più in basso nell'albero in grado di tradurre, finalmente, il nome in risorsa. Più formalmente:

- Se il server ha la risposta lo ritorna al client.
- Se il server non ha la risposta ma la responsabilità della traduzione è di uno dei server sottostanti inoltra la richiesta a quel server.
- Se il server non ha la risposta e non esistono server sottostanti che abbiano la responsabilità di tradurre quel nome viene ritornato errore.

Questa tecnica è evidentemente poco performante e discuteremo questo fatto a breve. Intanto facciamo alcune riflessioni importanti.

Due modi di risolvere: traduzione completa e traduzione incompleta

Esistono due modi per risolvere un nome:

- **Traduzione completa** (ricorsiva): una volta che il client contatta un name server sarà il name server stesso a occuparsi di **tutta la traduzione** ovvero contatterà lui direttamente il server figlio chiedendo la traduzione (e così via fino all'ultimo server autorevole in merito alla traduzione) dopodiché una volta ottenuta la risposta la invia al client.
- **Traduzione parziale** (iterativa): il server (se non dispone della traduzione) risponde al client fornendo il nome di un altro name server da contattare per ottenere la traduzione.

Chiaramente nel caso in cui un server sia autorevole per il nome, non dispone della traduzione e non ha più altri sotto-server autorevoli per quel nome (siamo in fondo alla ricerca) verrà ritornato un errore.

Dunque quando un client contatta un name server deve specificare:

- Il nome da tradurre
- La classe del nome (i DNS non lavorano solo per la traduzione di domini Internet, ma noi trattiamo solo questo caso)
- Il tipo di resource che si desidera
- La tipologia di traduzione desiderata: ricorsiva o iterativa (il name server potrebbe anche non accondiscendere al desiderio espresso dal richiedente)

Ottenere il primo name server da cui iniziare la ricerca

Come può un client conoscere il suo primo name server di contatto per iniziare la ricerca?

L'indirizzo del primo DNS viene fornito al client insieme al gateway ed al proprio indirizzo IP nel momento in cui si contatta il server **DHCP**, oppure è possibile specificare manualmente i server DNS.

In Unix il file che contiene queste configurazioni è **etc/resolv.conf**.

È possibile specificare diversi name server: se il primo non dovesse rispondere possiamo interpellare gli altri.

Fra questi server non compare mai un root server: ecco che l'idea avuta in precedenza di far partire la ricerca dall'inizio dell'albero non viene usata. Vedremo dopo come funziona il sistema.

Le informazioni a disposizione di un name server

Mantenendo il modello (che non è quello usato in realtà) di ricerca che parte dalla root e scende fino al name server che dispone della traduzione, possiamo intuire che ogni name server deve disporre di un collegamento rispetto ai nodi figli. Più precisamente un name server deve conoscere:

- Tutti i nomi dei suoi sottodomini che esistono. Dunque, per ciascuno di questi nomi:
 - I nomi di tutti i server che traducono i nomi di ciascun sottodominio
 - L'indirizzo IP di ogni server assegnato a ciascun sottodominio

Quindi ad esempio se considerassimo il name server pianeta che è autorevole per `di.unito.it` avremo una configurazione del genere:

```
educ.di.unito.it IN NS albert.educ.di.unito.it
albert.educ.di.unito.it IN A 130.192.241.8
```

Questo ci dice che il figlio di pianeta (`di.unito.it`) è il dominio `educ.di.unito.it` il cui name server autorevole corrisponde al nome `albert.educ.di.unito.it`. Dopodiché abbiamo un altro record che ci fornisce l'indirizzo IP corrispondente al name server che ha nome `albert.educ.di.unito.it`.

In altre parole sappiamo che se pianeta dovesse essere interpellato per tradurre `xy.educ.unito.it` (supponiamo in modalità ricorsiva) allora si prenderà in carico la richiesta e contatterà `albert.educ.di.unito.it` tramite l'indirizzo IP 130.192.241.8 per avere risposta riguardo alla richiesta.

4.6) Risolvere un nome di dominio (primo passo d'efficienza: bottom up)

Come abbiamo già accennato più volte partire dalla root per effettuare la traduzione è decisamente inefficiente, questo perché:

- La maggior parte delle richieste si riferisce a nomi locali che si trovano sullo stesso sottodominio in cui si trova la macchina che fa la richiesta
- Le macchine che ospitano la radice dell'albero e le loro interfacce diventerebbero rapidamente un collo di bottiglia
- I root server dovrebbero essere enormemente ridondati (difficile mantenere la consistenza)

Anche se i nomi sono definiti dall'alto verso il basso ci sarebbe molto più utile poterli tradurre dal basso verso l'alto! Perciò la traduzione inizia contattando un server locale, cioè uno più in basso nell'albero.

Questo ci porta ad una nuova problematica: **un server contattato potrebbe non essere autorevole** per la richiesta che gli è appena giunta! Questo non poteva succedere prima poiché partendo dalla root (che è autorevole su tutta l'Internet) si scendeva "mano a mano" ma sempre su server autorevoli.

Ecco quindi che è necessario passare la richiesta ad un **server a livello superiore dell'albero**. Dobbiamo scegliere fra:

- Un root server
- Un name server del dominio padre
- Il name server dell'ISP che mi sta offrendo il servizio di connessione

È quindi necessario che ogni name server conosca l'indirizzo IP di almeno un root server (che sono replicati per assicurare affidabilità ed effettuare load balancing) e poi (come già detto prima) i nomi dei sottodomini dei domini di cui è autorevole e per ciascuno di essi gli indirizzi IP dei server autorevoli per tali sottodomini.

Inoltre, optionalmente, un name server può conoscere il nome di suo padre e cioè il nome di quel name server che è autorevole per il dominio superiore al suo.

Anche con questa tecnica, però, abbiamo ancora bisogno di troppi accessi!

4.7) Risolvere un nome di dominio (la soluzione definitiva: il name caching)

La soluzione che risulta essere quella definitiva è il **name caching**, ovvero dotare di cache più o meno ampie i name server (e anche i client).

Una cache contiene:

- I nomi usati recentemente
- Gli attributi ad esso associati (i record)
- Il nome e l'indirizzo del/dei server autorevole/i da cui è stata ottenuta l'informazione
- Un Time To Leave (di cui parleremo meglio dopo)

Come avviene la risoluzione

Riassumiamo quindi come avviene una risoluzione: un name server X riceve una richiesta per tradurre un nome D.

X è autorevole per D?

1. Se **X è autorevole**, l'informazione è reperibile in diversi modi:
 - 1.1. Usando il database
 - 1.2. Interrogando i server dei sottodomini (richiesta ricorsiva)
 - 1.3. Ritornando il nome dei sottodomini (richiesta iterativa)
 - 1.4. Ritornando un errore se X non ha altri sottodomini autorevoli per D e non ha la traduzione
2. Se **X non è autorevole**
 - 2.1. X controlla la cache, contiene D?
 - 2.1.1. **La cache di X contiene D** quindi la traduzione viene restituita ma etichettata come *non autorevole* riportando il nome e l'IP del server autorevole per l'informazione richiesta (dal quale ha ottenuto l'informazione precedentemente)
 - 2.1.2. **La cache di X non contiene D**
 - 2.1.2.1. X contatta il server superiore (richiesta ricorsiva)
 - 2.1.2.2. X contatta server root (richiesta ricorsiva)
 - 2.1.2.3. X fornisce l'indirizzo di un server superiore/root (richiesta iterativa)

In questo modo i client possono scegliere a loro piacimento se "fidarsi" oppure contattare nuovamente il server autorevole a seconda della priorità (in termini di velocità) che hanno.

La durata della cache

Come abbiamo detto sopra ogni entry della cache dispone di un TTL. Questo è utile perché ovviamente i messaggi non possono avere durata infinita: sarà l'amministratore dei name server a decidere il valore del TTL (è l'unico in grado di poterlo assegnare in maniera sensata).

4.8) Il formato dei messaggi DNS

I messaggi DNS viaggiano su UDP (sono richieste idempotenti). Il formato dei messaggi DNS è molto utile per capire appieno il funzionamento di questo sistema.

Teniamo a mente un buon esempio: un browser chiede la traduzione di un nome di dominio per ottenere l'indirizzo IP (resource A, quindi) del server che contiene la pagina web richiesta.

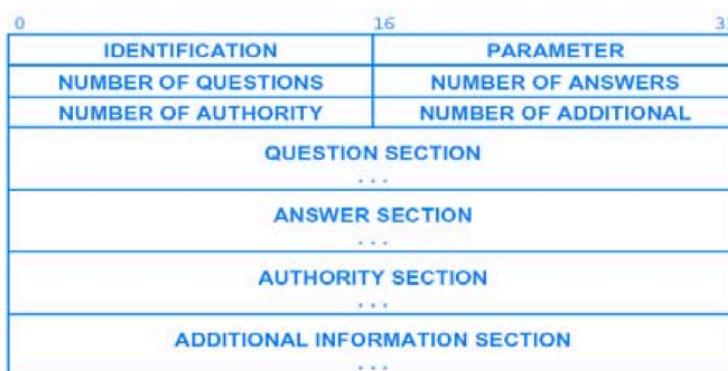
Ogni richiesta consta di:

- un *nome di dominio*
- la specifica di una *query class* (nel nostro caso sempre Internet)
- Il *tipo* dell'oggetto/risorsa richiesto

Dunque il server risponderà con:

- Le *risorse* richieste oppure, se queste non erano disponibili al server, informazioni sui *server autorevoli* corredate da *ulteriori informazioni* per poterli contattare.
- Informazioni relative all'*autorità* dell'*informazione* inviata

Ecco il formato:



Bit of PARAMETER field	Meaning
0	Operation: 0 Query 1 Response
1-4	Query Type: 0 Standard 1 Inverse 2 Server status request 4 Notify 5 Update
5	Set if answer authoritative
6	Set if message truncated
7	Set if recursion desired
8	Set if recursion available
9	Set if data is authenticated
10	Set if checking is disabled
11	Reserved
12-15	Response Type: 0 No error 1 Format error in query 2 Server failure 3 Name does not exist 5 Refused 6 Name exists when it should not 7 RR set exists 8 RR set that should exist does not 9 Server not authoritative for the zone 10 Name not contained in zone

L'header è fisso.

- IDENTIFICATION serve per accoppiare domande e risposte (è possibile effettuare più richieste ad un DNS quindi devono essere correlate le domande e le risposte per evitare misunderstanding). La domanda è comunque sempre restituita insieme alle risposte.

- PARAMETER permette di interpretare il messaggio (si veda la tabella a fianco). Come si nota non è così facile carpire il significato del campo PARAMETER e perciò è suggeribile usare una tabella come quella riportata qui:

Pos	Peso Hex	Significato
0	8	0=Query; 1=Response
1	4	0 =Standard; 1=Inverse;
2	2	2=Completion 1
3	1	3=Completion 2
4	8	
5	4	Set if answer authoritative
6	2	Set if message truncated
7	1	Set if recursion desired
8	8	Set if recursion available
9	4	Reserved
10	2	
11	1	
12	8	0=No error; 1= Format error;
13	4	2=Server failure;
14	2	3=Name does not exist
15	1	

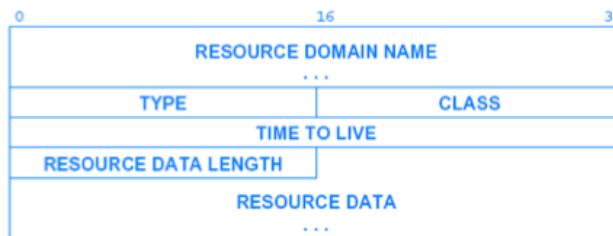
Come vediamo abbiamo alcuni modi per esprimere il fatto che il client desidera una traduzione ricorsiva (ma il server può rispondere anche che non è disponibile).

- NUMBER OF fornisce un controllo delle entry che sono incluse nel messaggio.
- QUERY SECTION conterrà query nel seguente formato:



Viene fornito il domain name (ha lunghezza variabile, il server potrà determinarla in un modo che vedremo in seguito) la QUERY TYPE (il tipo di risorsa desiderato) e la QUERY CLASS (Internet per noi).

- ANSWER SECTION, AUTORITY SECTION e ADDITIONAL INFORMATION SECTION sono invece compilate dal server e hanno questa struttura:



Il nome dei campi è esplicativo.

Formato compresso dei nomi e abbreviazione dei nomi di dominio

I nomi di dominio vengono trasmessi come sequenza di etichette (separate fra loro da punti). Ogni etichetta è codificata con un otetto iniziale che rappresenta la lunghezza dell'etichetta stessa e la fine dell'etichetta è marcata da un otetto vuoto. Le etichette possono essere al massimo lunghe 64 caratteri (6 bit).

Spesso i server ritornano risposte relative a nomi che hanno un suffisso comune. Per ragioni di ottimizzazione nelle risposte questa parte comune può essere inviata una volta sola e nei nomi successivi si punterà a quel suffisso comune. Come?

Se il byte che indica la lunghezza di un'etichetta ha i primi due bit a uno allora i successivi 14 bit indicheranno un intero a 14 bit che punta alla posizione nel messaggio DNS in cui inizia la stringa che termina il nome di dominio.

Inoltre, i nomi di dominio sono abbreviabili: nella configurazione di un client sarà possibile introdurre un suffisso da apporre alle risposte di un certo DNS:

```
domain di.unito.it.
nameserver 130.192.239.1
nameserver 130.192.119.1
nameserver 193.205.245.8
nameserver 193.205.245.5
```

Si noti che il completamento viene applicato prima che la query venga inviata al server (il DNS mappa solo nomi completi!).

Il mapping inverso (query PTR)

Fino ad ora abbiamo pensato a mapping fra nomi di dominio verso resource record e non il contrario.

Ma come abbiamo detto qualche paragrafo sopra, è possibile anche ottenere un nome di dominio dato un indirizzo IP. Dato che però le risposte inverse non sono uniche ed è molto complesso individuare un server che sia in grado di dare la risposta, le *inverse query* non sono implementate nonostante il formato del messaggio DNS sia in grado di esprimerele.

L'unica inverse query che è possibile è quella da indirizzo IP a nome di dominio, ma in realtà questa viene fatta "mascherata" da query diretta.

Innanzitutto: perché ci possono servire query PTR?

I manager di rete tendenzialmente assegnano nomi di dominio solo per macchine fidate: disporre di uno di questi è quindi **indice di "fiducia"**.

Il messaggio di richiesta dovrebbe contenere un indirizzo IP ma dato che quest'ultimo non ha la forma di un nome di dominio ci troviamo in difficoltà!

È stato allora ideato un **dominio DNS all'interno del quale sistemare gli indirizzi IP** in forma dotted quad (ovviamente scritti al contrario cioè con i byte del netID "in alto" sull'albero!).

Dunque prima di tutto trasformiamo l'IP in forma dotted quad (aaa.bbb.ccc.ddd).

Apponiamo poi il dominio **in-addr.arpa**. (quel dominio DNS ideato ad hoc di cui parlavamo poco fa) all'indirizzo ma scritto al contrario poiché l'indirizzo IP ha più precisione da sinistra verso destra (mentre i nomi di domini sono più precisi da destra verso sinistra). Otteniamo quindi ddd.ccc.bbb.aaa.in-addr.arp.

A questo punto il name server locale di primo contatto non avrà sicuramente autorità né per arpa né per in-addr e dovrà contattare obbligatoriamente il root server. Chiaramente non ci si può aspettare che vi sia un unico name server ad effettuare le traduzioni inverse... ogni amministratore di rete dovrà registrare (se interessato) un name server autorevole per xxx.in-addr.arpa. in modo da poter effettuare le traduzioni inverse per la propria rete.

4.9) Resources ed esempi

A monte di tutte le riflessioni fatte fin ora rivediamo le risorse ottenibili tramite query a name server corredate di esempi.

Risorse A

Il tipo di risorse più chieste, viene fornito l'indirizzo IP corrispondente al nome di dominio.

```
C:\>nslookup webmail.libero.it
Server: ns.iunet.it
Address: 192.106.1.1
Risposta da un server non di fiducia:
Nome: webmail.libero.it
Addresses: 193.70.192.64, 193.70.192.41,
           193.70.192.42, 193.70.192.61,
           193.70.192.62, 193.70.192.63
C:\>nslookup webmail.libero.it
Server: ns.iunet.it
Address: 192.106.1.1
Risposta da un server non di fiducia:
Nome: webmail.libero.it
Addresses: 193.70.192.61, 193.70.192.62,
           193.70.192.63, 193.70.192.64,
           193.70.192.41, 193.70.192.42
```

In questo esempio vediamo che vengono restituiti più indirizzi IP: perché?

Se parliamo di grossi servizi (con grossa utenza) non vi sarà una macchina sola a gestire tutte le richieste ma ve ne saranno una moltitudine. Il DNS viene, in questo caso, sfruttato in modo leggermente improprio: si sfrutta per fare load balancing. Quando si effettua una richiesta ad un name server autoritario per un certo dominio il quale ha delle corrispondenze di tipo A ed è gestito da più macchine verranno restituiti tutti gli IP **ma in ordine diverso ogni volta**: questo fa sì che le cache dei vari name server contengano sempre gli stessi indirizzi IP ma in ordine diverso uno dall'altro. Giacché i client interrogano il primo IP che trovano, le richieste verranno spartite fra i vari indirizzi che gestiscono il servizio.

Si noti che usando il DNS in questo modo il caching (ed il TTL correlato) diventa pericoloso: infatti se abbiamo un TTL molto lungo non verranno effettuate nuove richieste DNS e quindi in realtà il name server chiamato per effettuare la richiesta sarà sempre lo stesso e non avremo il load balacing che desideravamo.

Risorse CNAME

Quando un nome di dominio dispone di CNAME significa che è un alias e il CNAME ci restituisce il vero nome della macchina alla quale l'alias punta (e gli altri eventuali alias). Gli alias sono molto usati per dare il nome ad un servizio (un classico esempio è **www** che è spesso l'alias di una macchina che dispone di un servizio HTTP attivo).

```
C:\>nslookup
Server predefinito: ns.iunet.it
Address: 192.106.1.1
> www.di.unito.it
Server: ns.iunet.it
Address: 192.106.1.1
Risposta da un server non di fiducia:
Nome: pianeta.di.unito.it
Address: 130.192.239.1
Aliases: www.di.unito.it
>
```

Risorse HINFO e MINFO

Raramente utilizzate, forniscono informazioni su hardware e mailbox.

Risorse MX

Il valore di queste risorse è un nome di dominio. In sostanza un client di posta elettronica si trova a dover inviare una mail a abc@gmail.com (ad esempio). Verrà inviata una richiesta al name server di primo contatto per il nome 'gmail.com'. La risposta sarà il nome di dominio del server che dispone di un servizio di posta elettronica (che quindi il client di posta elettronica provvederà a contattare per inviarvi l'email).

```
> set q=mx
> di.unito.it
Server: ns.iunet.it
Address: 192.106.1.1
Risposta da un server non di fiducia:
di.unito.it      MX preference = 10, mail
exchanger = pianeta.di.unito.it
di.unito.it      MX preference = 40, mail
exchanger = albert.unito.it
```

In questo caso abbiamo due macchine che gestiscono il dominio di posta di.unito.it (pianeta.di.unito.it e anche albert.unito.it).

Risorse NS

Resource molto importante poiché informa nella risposta quale sia il name server autorevole per la domanda (così il client può contattare direttamente il server autorevole se non si dovesse fidare).

```
> set q=ns
> di.unito.it
Server: ns.iunet.it
Address: 192.106.1.1
Risposta da un server non di fiducia:di.unito.it
nameserver = albert.unito.it
di.unito.it      nameserver = amleto.di.unito.it
di.unito.it      nameserver = pianeta.di.unito.it
albert.unito.it internet address = 130.192.119.1
amleto.di.unito.it internet address =
130.192.239.30
pianeta.di.unito.it internet address =
130.192.239.1
```

Risorse PTR

Ne abbiamo parlato profusamente prima, viene restituito il nome di dominio di una macchina dato il suo indirizzo IP.

```
> set q=ptr
> 130.192.239.1
Server: ns.iunet.it
Address: 192.106.1.1
Risposta da un server non di fiducia:
1.239.192.130.in-addr.arpa name = pianeta.di.unito.it
239.192.130.in-addr.arpa nameserver =
pianeta.di.unito.it
239.192.130.in-addr.arpa nameserver = itaca.di.unito.it
pianeta.di.unito.it internet address = 130.192.239.1
itaca.di.unito.it   internet address = 130.192.239.182
```

Risorse SOA

Vengono fornite informazioni riguardanti la gerarchia dei nomi a cui un nome di dominio appartiene.

```

> set q=soa
> di.unito.it
Server: ns.iunet.it
Address: 192.106.1.1
Risposta da un server di fiducia:
di.unito.it primary name server =
    pianeta.di.unito.it
    responsible mail addr =
    root.pianeta.di.unito.it
    serial = 2001050901
    refresh = 86400 (1 day)
    retry = 3600 (1 hour)
    expire = 302400 (3 days 12 hours)
    default TTL = 43200 (12 hours)
di.unito.it      nameserver = albert.unito.it
di.unito.it      nameserver = amleto.di.unito.it
di.unito.it      nameserver = pianeta.di.unito.it
amleto.di.unito.it internet address = 130.192.239.30
pianeta.di.unito.it internet address = 130.192.239.1

```

4.10) Richieste al DNS: server primari e server secondari

Abbiamo più volte accennato al fatto che spesso i server vengono replicati per motivi computazionali: le richieste sono molte e un server solo (soprattutto se si trova ad alto livello sull'albero) non può prendersi carico di tutte le traduzioni.

I nostri obiettivi sono due:

- Evitare single points of failure (se si rompe un name server il dominio per cui è autorevole e tutti i sottodomini non sono più raggiungibili?)
- Dividere il carico fra numerosi name server

Vengono perciò introdotti i **server secondari** che sono server DNS che prendono le informazioni su una intera zona di autorità per cui sono server secondari di un server primario: in altre parole sono "sostituti" dei server primari in caso di necessità. Il server primario comunicherà col server secondario per mandare le informazioni (che possono essere molto grosse) utili per effettuare la traduzione: questo tipo di operazione è detto **zone transfer** (trasferimento della zona di autorità) e viene effettuata tramite TCP (ne parleremo brevemente in seguito).

I server secondari devono trovarsi ovviamente su reti diverse (altrimenti al crash della rete perdiamo entrambi i server) ed essere indipendenti rispetto alle risorse che usano (ad esempio l'alimentazione).

I client locali sfruttano i server secondari solamente se i server primari non rispondono.

4.11) DNS: UDP o TCP?

Il protocollo DNS è uno dei pochi che sfrutta sia le connessioni TCP che quelle UDP. TCP viene usato **solamente** durante le zone transfer tra server primari e server secondari, mentre invece UDP è il protocollo solitamente usato per le richieste DNS. **Entrambi i protocolli comunicano sulla porta 53.**

4.12) Ottenere l'autorità per un sottodomainio

Se volessimo creare un nostro server DNS e quindi avere autorità per un certo sottodomainio, come potremmo fare?

Dobbiamo sottostare a certe **condizioni**:

- Essere disposti a gestire un server DNS che soddisfa gli standard di Internet (verrà effettuata una verifica di compatibilità e correttezza).
- Si deve dimostrare che il server conosce l'indirizzo di almeno un root server.
- Si deve dimostrare che il server conosce (nel caso in cui ve ne siano) tutti i server che gestiscono gli eventuali sottodomini.
- Bisogna avere almeno un server secondario ragionevolmente separato dal server primario.

Si consideri comunque che gestire un server DNS è tutt'altro che semplice:

- Bisogna saper gestire più domini (anche se indipendenti).
- Bisogna essere capaci di gestire numerose operazioni in parallelo (performance, richieste ricorsive, richieste iterative, ecc.).
- Supportare la replica tramite zone transfer.
- Gestire in modo efficiente cache anche molto grandi.

5) Studio delle tracce Wireshark

Vediamo ora alcune tracce di Wireshark per contestualizzare quanto fatto fin ora.

5.1) Richieste generiche

Ping (request)

```

> Frame 46272: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
  ▾ Ethernet II, Src: Apple_b9:0b:59 (10:dd:b1:b9:0b:59), Dst: Netgear_3b:e1:5b (10:0d:7f:3b:e1:5b)
    ▾ Destination: Netgear_3b:e1:5b (10:0d:7f:3b:e1:5b)
      Address: Netgear_3b:e1:5b (10:0d:7f:3b:e1:5b)
      .... ..0. .... .... .... = LG bit: Globally unique address (factory default)
      .... ..0. .... .... .... = IG bit: Individual address (unicast)
    ▾ Source: Apple_b9:0b:59 (10:dd:b1:b9:0b:59)
      Address: Apple_b9:0b:59 (10:dd:b1:b9:0b:59)
      .... ..0. .... .... .... = LG bit: Globally unique address (factory default)
      .... ..0. .... .... .... = IG bit: Individual address (unicast)
    Type: IP (0x0800)
  ▶ Internet Protocol Version 4, Src: 192.168.0.15 (192.168.0.15), Dst: 173.194.40.20 (173.194.40.20)
  ▾ Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x9edf [correct]
    Identifier (BE): 47368 (0xb908)
    Identifier (LE): 2233 (0x08b9)
    Sequence number (BE): 4 (0x0004)
    Sequence number (LE): 1024 (0x0400)
    [Response frame: 46273]
    Timestamp from icmp data: Jun 7, 2013 14:14:00.431312000 CEST
    [Timestamp from icmp data (relative): 0.000044000 seconds]
  ▾ Data (48 bytes)
    Data: 08090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f...
    [Length: 48]
```

Un classico ping a www.google.it (indirizzo IP 173.194.40.20): notiamo che nel frame ethernet la destinazione indicata è il Netgear_3b:e1:5b mentre nel pacchetto IP la destinazione è l'indirizzo IP del server di Google.

Traceroute (sequenza di ping)

Filter:	icmp	Time	Source	Destination	Protocol	Info
0.		928 129.373008	192.168.0.15	74.125.91.147	ICMP	Echo (ping) request (id=0x0001, seq(be/le)=4/1024, ttl=1)
1.		929 129.413927	192.168.0.1	192.168.0.15	ICMP	Time-to-live exceeded (Time to live exceeded in transit)
2.		930 129.415644	192.168.0.15	74.125.91.147	ICMP	Echo (ping) request (id=0x0001, seq(be/le)=5/1280, ttl=1)
3.	931 129.417250	192.168.0.1	192.168.0.15	ICMP	Time-to-live exceeded (Time to live exceeded in transit)	
4.	932 129.418859	192.168.0.15	74.125.91.147	ICMP	Echo (ping) request (id=0x0001, seq(be/le)=6/1536, ttl=1)	
5.	933 129.429882	192.168.0.1	192.168.0.15	ICMP	Time-to-live exceeded (Time to live exceeded in transit)	
6.	936 130.450735	192.168.0.15	74.125.91.147	ICMP	Echo (ping) request (id=0x0001, seq(be/le)=7/1792, ttl=1)	
7.	937 130.588477	10.39.112.1	192.168.0.15	ICMP	Time-to-live exceeded (Time to live exceeded in transit)	
8.	938 130.590271	192.168.0.15	74.125.91.147	ICMP	Echo (ping) request (id=0x0001, seq(be/le)=8/2048, ttl=2)	
9.	939 130.714645	10.39.112.1	192.168.0.15	ICMP	Time-to-live exceeded (Time to live exceeded in transit)	
10.	940 130.716295	192.168.0.15	74.125.91.147	ICMP	Echo (ping) request (id=0x0001, seq(be/le)=9/2304, ttl=2)	
11.	943 130.836414	10.39.112.1	192.168.0.15	ICMP	Time-to-live exceeded (Time to live exceeded in transit)	
12.	983 136.283932	192.168.0.15	74.125.91.147	ICMP	Echo (ping) request (id=0x0001, seq(be/le)=10/2560, ttl=3)	
13.	984 136.312163	67.231.221.149	192.168.0.15	ICMP	Time-to-live exceeded (Time to live exceeded in transit)	
14.	985 136.313958	192.168.0.15	74.125.91.147	ICMP	Echo (ping) request (id=0x0001, seq(be/le)=11/2816, ttl=3)	


```

> Frame 929: 134 bytes on wire (1072 bits), 134 bytes captured (1072 bits)
  ▾ Ethernet II, Src: SmcNetwo_9a:b5:af (00:26:f3:9a:b5:af), Dst: LiteonTe_c2:4f:21 (70:f1:a1:c2:4f:21)
  ▾ Internet Protocol, Src: 192.168.0.1 (192.168.0.1), Dst: 192.168.0.15 (192.168.0.15)
  ▾ Internet Control Message Protocol
    Type: 11 (Time-to-live exceeded)
    Code: 0 (Time to live exceeded in transit)
    Checksum: 0xf4ff [correct]
```

Vediamo come vengano inviati ping con **TTL crescente** che ricevono risposte di **TTL exceeded**.

Il framing

```
▽ Internet Protocol Version 4, Src: 192.168.0.15 (192.168.0.15), Dst: 173.194.40.16 (173.194.40.16)
  Version: 4
  Header length: 20 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
  Total Length: 1500
  Identification: 0xb61e (46622)
  ▷ Flags: 0x01 (More Fragments)
    0.... .... = Reserved bit: Not set
    .0.... .... = Don't fragment: Not set
    ..1.... .... = More fragments: Set
  Fragment offset: 0
  Time to live: 64
  Protocol: ICMP (1)
```

Vediamo che il flag "More Fragments" è stato settato: implica che il frame IP è frammentato.

Si noti che la frammentazione può essere sfruttata per **scoprire l'MTU** di una linea: si mandano ping con payload incrementali ma con l'opzione **don't fragment settata**: in questo modo quando riceveremo un failure dal ping sapremo quale sarà l'MTU poiché un router avrà scartato il pacchetto dato che era necessaria la frammentazione (ma dato che il flag don't fragment è a 1, il pacchetto è stato scartato).

```
230506 1794.755253000 192.168.0.1           192.168.0.15           ICMP          590 Destination unreachable (Fragmentation needed)
```

5.2) ARP

ARP request

```

> Frame 830: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
  ▼ Ethernet II, Src: HonHaiPr_a2:32:c6 (78:e4:00:a2:32:c6), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
    ▼ Destination: Broadcast (ff:ff:ff:ff:ff:ff)
      Address: Broadcast (ff:ff:ff:ff:ff:ff)
      .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory default)
      .... ..1. .... .... .... .... = IG bit: Group address (multicast/broadcast)
    ▼ Source: HonHaiPr_a2:32:c6 (78:e4:00:a2:32:c6)
      Address: HonHaiPr_a2:32:c6 (78:e4:00:a2:32:c6)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ..0. .... .... .... .... = IG bit: Individual address (unicast)
    Type: ARP (0x0806)
    Padding: da5bd4a3e0946da80ef4ffd170302002cf4
  ▼ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: HonHaiPr_a2:32:c6 (78:e4:00:a2:32:c6)
    Sender IP address: 192.168.0.9 (192.168.0.9)
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.0.1 (192.168.0.1)

```

Vediamo una ARP request da parte del calcolatore con MAC NonHaiPr_a2:32:c6. Ci rendiamo conto che si tratta di una ARP request poiché il **target MAC address è vuoto**. Inoltre, è una richiesta **in broadcast**.

ARP reply

```

> Frame 12549: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
  ▼ Ethernet II, Src: Apple_b9:0b:59 (10:dd:b1:b9:0b:59), Dst: Netgear_3b:e1:5b (10:0d:7f:3b:e1:5b)
    ▼ Destination: Netgear_3b:e1:5b (10:0d:7f:3b:e1:5b)
      Address: Netgear_3b:e1:5b (10:0d:7f:3b:e1:5b)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ..0. .... .... .... .... = IG bit: Individual address (unicast)
    ▼ Source: Apple_b9:0b:59 (10:dd:b1:b9:0b:59)
      Address: Apple_b9:0b:59 (10:dd:b1:b9:0b:59)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ..0. .... .... .... .... = IG bit: Individual address (unicast)
    Type: ARP (0x0806)
  ▼ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: Apple_b9:0b:59 (10:dd:b1:b9:0b:59)
    Sender IP address: 192.168.0.15 (192.168.0.15)
    Target MAC address: Netgear_3b:e1:5b (10:0d:7f:3b:e1:5b)
    Target IP address: 192.168.0.1 (192.168.0.1)

```

Ecco invece un'ARP reply: notiamo che è in **unicast** ed il campo target MAC address è **compilato**. Troveremo nel sender MAC address la risposta che interessava (in questo caso) al Netgear_3b:e1:5b.

5.3) DNS

Richiesta NS

```
Frame 1: 69 bytes on wire (552 bits), 69 bytes captured (552 bits)
Ethernet II, Src: EdimaxTe_00:d3:1e (00:50:fc:00:d3:1e), Dst: Cisco_cf:fc:a1 (00:07:50:cf:fc:a1)
Internet Protocol Version 4, Src: 130.192.239.253 (130.192.239.253), Dst: 195.210.91.100 (195.210.91.100)
User Datagram Protocol, Src Port: bnetgame (1119), Dst Port: domain (53)
    Source port: bnetgame (1119)
    Destination port: domain (53)
    Length: 35
    Checksum: 0xb745 [validation disabled]
Domain Name System (query)
    [Response In: 2]
    Transaction ID: 0x0021
    Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
    Queries
        libero.it: type NS, class IN
            Name: libero.it
            Type: NS (Authoritative name server)
            Class: IN (0x0001)
```

Una semplice query di tipo NS, notiamo i campi della query, il numero di domande, la porta 53, ecc.

I socket

1) Introduzione ai socket

Il software relativo a TCP/IP risiede all'interno del sistema operativo.

Si deve quindi dare la possibilità alle applicazioni (tramite un'interfaccia) di sfruttare la rete. L'interfaccia più usata sono i **socket**.

1.1) Le funzionalità delle interfacce

Quali sono le funzionalità che un'interfaccia dovrebbe fornire?

- Allocare risorse necessarie per la comunicazione
- Specificare gli endpoint della comunicazione (locale e remoto) iniziare una connessione (lato client)
- Attendere una connessione (lato server)
- Inviare e ricevere dati
- Determinare quando i dati arrivano
- Generare dati urgent
- Gestire l'arrivo di dati urgent
- Terminare una connessione (gracefully)
- Gestire la terminazione che arriva dalla controparte
- Abort di una comunicazione
- Gestire condizioni di errore oppure abort di connessione rilascio risorse locali quando la comunicazione termina

Per poter svolgere tutte queste funzioni il protocollo TCP/IP, implementato nel sistema operativo, permette alle applicazioni di interagire con esso tramite le **system call**.

La comunicazione con i socket avviene sfruttando le stesse system call che si usano normalmente per l'I/O su file, infatti (su unix), i socket sono a tutti gli effetti dei file particolari che permettono l'interazione con la rete sfruttando il protocollo TCP/IP.

1.2) Creare un socket

Per dichiarare un socket si usa la system call:

socket = (protofamily, type, protocol)

Dove con *protofamily* definiamo la famiglia di protocollo, e quindi abbiamo:

- **PF_INET** per TCP/IP.
- **PF_PUP** per protocolli Xerox.
- **PF_APPLETALK** per i protocollli AppleTalk.
- **PF_UNIX** per specificare il file system di Unix.

Con *type* definiamo la modalità di connessione:

- **SOCK_STREAM** per connection-oriented (TCP).
- **SOCK_DGRAM** per connection-less (UDP).
- **SOCK_RAW** per usi con privilegi particolari.

Con *protocol* si definisce il particolare protocollo all'interno della famiglia e tipo.

Quindi, per esempio, istanzieremo un socket eseguendo: `socket(PF_INET, SOCK_STREAM, 0);`

Una volta creato, il socket viene usato per comunicare (**active socket**) oppure per attendere una comunicazione (**passive socket**).

Creare un endpoint

La creazione di un socket non ne comporta l'utilizzo immediato! Esso è privo di tutte le informazioni relative ai numeri di porta o agli IP (locali e remoti). Dobbiamo quindi esplicitare un **endpoint** cioè una coppia (IP, #porta) oltre alla famiglia di indirizzi (per TCP/IP è **AF_INET**).

I socket permettono di definire una forma di endpoint generico oppure un formato specifico per una certa famiglia di protocolli. Gli endpoint TCP/IP hanno formato specifico e sono così composti:

```
struct sockaddr_in {
    u_char sin_len;                      /* contiene un indirizzo */
    u_short sin_family;                  /* tipo dell'indirizzo */
    u_short sin_port;                   /* porta del protocollo */
    struct in_addr sin_addr;            /* IP address, si veda sotto */
    char sin_zero[8];                  /* inutilizzato */

}

struct in_addr {                         /* struttura per l'indirizzo IP */
    u_long s_addr;                      /* indirizzo IP */
}
```

Chiaramente tutti questi metodi lavorano in interi mentre TCP/IP utilizza la rappresentazione network byte order (interi con il byte più significativo per primo): perciò si adoperano funzioni di conversione quali:

- htons() - host to network short
- htonl() - host to network long
- ntohs() - network to host short
- nrhol() network to host long

1.3) System call server side

Vediamo prima tutte le system call che vengono usate dal server.

bind()

La bind **collega il socket all'endpoint locale** ovvero viene usata dal server per "esporre" il suo socket sulla rete, fornendo infatti un indirizzo ed un socket.

bind(int sockfd, struct sockaddr *my_addr, int addrlen)

- *sockfd*: descrittore di un socket
- *my_addr*: endpoint (come visto sopra)
- *addrlen*: dimensione del parametro *my_addr*

Esempio

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#define MYPRT 3490

main() {
    int sockfd;
    struct sockaddr_in my_addr;
    sockfd = socket(PF_INET, SOCK_STREAM, 0); my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(MYPRT);
    my_addr.sin_addr.s_addr = inet_addr("132.241.5.10");
    bzero(&(my_addr.sin_zero), 8);
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

    ...
}
```

La bind restituisce -1 se c'è un errore (si cerca di assegnare al socket una porta minore di 1024 oppure una porta già assegnata).

listen()

Un socket a cui è assegnato un endpoint viene **reso passivo**, ovvero, viene messo in ascolto e reso in grado di accettare una connessione (ma non l'accetta ancora!).

listen(int sockfd, int qlen)

- *sockfd*: descrittore di un socket (passivo)
- *qlen*: numero massimo di connessioni in attesa (sul socket passivo)

accept()

Un server **accetta una richiesta di connessione**.

accept(int sockfd, sockaddr_in *addr, int addrlen)

- *sockfd*: descrittore di un socket (passivo)
- *addr*: puntatore ad una struttura *sockaddr_in* dove verranno memorizzate le informazioni sulla connessione (IP e porta del chiamante)
- *addrlen*: dimensione del parametro *addr*

Quando la accept raccoglie una richiesta, viene creato un nuovo socket ed il server utilizzerà il nuovo socket per connessione appena aperta mentre manterrà il vecchio socket passivo per riceverne delle altre.

Esempio

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#define MYPORt 3490      /* porta alla quale si connette il client */
#define BACKLOG 10        /* quante connessioni pendenti possiamo tenere */

main() {
    int sockfd, new_fd, sin_size;
    struct sockaddr_in my_addr, their_addr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(MYPORt);
    my_addr.sin_addr.s_addr = INADDR_ANY; bzero(&(my_addr.sin_zero), 8);

    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    listen(sockfd, BACKLOG);
    new_fd = accept(sockfd, &their_addr, &sin_size);

    ...
}
```

Stando a quanto detto sopra, l'accept può essere gestita sia concorrentemente che iterativamente. Quando un'accept riceve una richiesta termina e quindi:

- **Concorrentemente:** dopo la terminazione dell'accept, il master server crea uno slave server per gestire la richiesta, chiude sia la listen che il socket e poi lo riapre (con l'accept conseguente) se richiesto.
- **Iterativamente:** dopo la terminazione dell'accept il server stesso gestisce la richiesta, quindi, chiude il socket (cioè lo toglie dallo stato di listen), serve il client e poi riabilita l'accept se richiesto.

Vi saranno esempi esplicativi in seguito.

1.4) System call client side

Vediamo ora quali sono le system call che il client può adoperare per interagire con il server.

connect()

La connect è eseguita dal client per **richiedere una connessione** ad un server.

```
connect(int sockfd, struct sockaddr *serv_addr, int addrlen)
```

- *sockfd*: descrittore di un socket
- *serv_addr*: endpoint remoto al quale ci si vuole connettere
- *addrlen*: dimensione del parametro *serv_addr*

La connect esegue quattro compiti:

- Controlla che il socket (espresso come parametro) non sia già connesso ad un altro endpoint.
- Memorizza l'endpoint remoto nella struttura connessa al socket.
- Sceglie un endpoint locale (cioè **quello che faceva la bind dal server!** Viene fatta direttamente dalla connect).
- Inizia una connessione TCP e restituisce un valore che il indica successo o il fallimento dell'operazione.

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#define DEST_IP "132.241.5.10"
#define DEST_PORT 23

main() {
    int sockfd;
    struct sockaddr_in dest_addr;
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    dest_addr.sin_family = AF_INET; /* ordine dei byte dell'host */
    dest_addr.sin_port = htons(DEST_PORT);
    dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
    bzero(&(dest_addr.sin_zero), 8);
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));

    ...
}
```

send()

La send è eseguita da un client per **inviare un pacchetto**.

```
send(int sockfd, char * buff, int nbytes, int flags)
```

- *sockfd*: descrittore di un socket
- **buff*: indirizzo di memoria contenente il dato da spedire
- *nbytes*: dimensione del messaggio da spedire
- *flags*: può essere 0 oppure:
 - MSG_OOB: invia dati urgenti (PSH)
 - MSG_DONTROUTE: bypass il routing

Restituisce il numero di byte effettivamente inviati. La send viene usata con un socket sul quale è anche stata applicata la connect (per un approccio connection-oriented)

sendto() e sendmsg()

Analoghi della send, ma non richiedono la connect, abbiamo infatti che l'endpoint è specificato direttamente nella chiamata. Sono usati in un approccio connection-less.

```
sendto(int sockfd, char * buff, int nbytes, int flags,
       (struct * sockaddr_in) destaddr, int addrlen)
```

- *sockfd*: descrittore di un socket
- **buff*: indirizzo di memoria contenente il dato da spedire
- *nbytes*: dimensione del messaggio da spedire
- *flags*: può essere 0 oppure:
 - MSG_OOB: invia dati urgenti (PSH)
 - MSG_DONTROUTE: bypass il routing
- *destaddr*: endpoint remoto al quale ci si vuole connettere
- *addrlen*: dimensione del parametro destaddr

```
sendmsg(int sockfd, const struct mssghdr * msg, int flags)
```

- *sockfd*: descrittore di un socket
- *mssghdr*: struttura che contiene sia il messaggio da inviare che l'endpoint di destinazione
- *flags*: può essere 0 oppure:
 - MSG_OOB: invia dati urgenti (PSH)
 - MSG_DONTROUTE: bypass il routing

Quindi, come è chiaro, sendto e sendmsg cambiano solo per il modo di incapsulare il parametri.

[recv\(\)](#)

La recv è eseguita da un client per **ricevere un pacchetto**.

```
recv(int sockfd, char * buff, int nbytes, int flags)
```

- *sockfd*: descrittore di un socket
- **buff*: indirizzo di memoria dove verrà memorizzato il dato ricevuto
- *nbytes*: dimensione del messaggio da ricevere
- *flags*: può essere 0 oppure:
 - MSG_OOB: riceve dati urgenti (PSH)
 - MSG_PEEK: legge i dati senza rimuoverli dal buffer di sistema

Restituisce il numero di byte effettivamente letti. La recv viene usata con un socket sul quale è anche stata applicata la connect (per un approccio connection-oriented).

[recvfrom\(\) e recvmsg\(\)](#)

Analoghi della recv, ma non richiedono la connect, abbiamo infatti che l'endpoint è specificato direttamente nella chiamata. Sono usati in un approccio connection-less.

```
recvfrom(int sockfd, char * buff, int nbytes, int flags,
         (struct * sockaddr_in) sndaddr, int addrlen)
```

- *sockfd*: descrittore di un socket
- **buff*: indirizzo di memoria dove verrà memorizzato il dato ricevuto
- *nbytes*: dimensione del messaggio da ricevere
- *flags*: può essere 0 oppure:
 - MSG_OOB: riceve dati urgenti (PSH)
 - MSG_PEEK: legge i dati senza rimuoverli dal buffer di sistema
- *sndaddr*: endpoint remoto del mittente
- *addrlen*: dimensione del parametro sndaddr

```
sendmsg(int sockfd, const struct mssghdr * msg, int flags)
```

- *sockfd*: descrittore di un socket
- *mssghdr*: struttura che contiene sia il messaggio ricevuto che l'endpoint del mittente
- *flags*: può essere 0 oppure:
 - MSG_OOB: invia dati urgenti (PSH)
 - MSG_PEEK: legge i dati senza rimuoverli dal buffer di sistema

Quindi, come è chiaro, recvfrom e recvmsg cambiano solo per il modo di encapsulare il parametri.

close() e shutdown()

La close **chiude la connessione** (se presente) **e rimuove il socket**.

close(int sockfd)

- *sockfd*: descrittore di un socket

La shutdown **chiude selettivamente un socket**.

shutdown(int sockfd, int direction)

- *sockfd*: descrittore di un socket

- *direction*: a seconda del suo valore specifica la direzione di shutdown

- 0: impedisce ulteriori recezioni
- 1: impedisce ulteriori spedizioni
- 2: simile alla close

Interfacciarsi col DNS: gethostbyname() e gethostbyaddr()

La gethostbyname restituisce l'indirizzo IP di un calcolatore chiamato come la stringa passata come parametro.

Questa funzione è diretta ai DNS.

gethostbyname(char * hostname)

- *hostname*: stringa contenente l'hostname

Esempio

```
hp = gethostbyname("pianeta.di.unito.it");
bcopy(hp->h_addr, (char*)&serv_addr.sin_addr, hp->h_length); /* copio */
```

La gethostbyid restituisce l'host name di un calcolatore al quale corrisponde l'IP passato come parametro.

gethostbyaddr(char *IPAddr, int sizeOfIp, sin_family)

- *IPAddr*: stringa contenente l'indirizzo IP

- *sizeOfIp*: dimensione del parametro IPAddr

- *sin_family*: famiglia dell'indirizzo

Esempio

```
hostent *hp;
struct in_addr IPAddr;

IPAddr.s_addr = inet_addr("130.192.239.1");
hp = gethostbyaddr((char*)&IPAddr, sizeof(IPAddr), AF_INET);
```

Impostare parametri dei socket: setsockopt() e getsockopt()

Permettono le impostazioni di alcuni parametri di un socket.

setsockopt(int sockfd, int level, int oname, char *oval, int olen)
getsockopt(int sockfd, int level, int oname, char *oval, int olen)

Convertire gli indirizzi: inet_ntoa() e inet_addr

Con **inet_ntoa()** convertiamo un indirizzo IP in notazione dotted decimal e con **inet_addr** convertiamo un indirizzo da notazione dotted decimal in formato IP (già in network order).

select(): attesa non deterministica

La funzione select consente ad un processo di bloccarsi in attesa di dati su più socket simultaneamente.

```
int select(int maxfd, fd_set *readfs, fd_set *writefs,  
          fd_set *exc, struct timeval *timeout)
```

- *writefs*: socket pronti per la scrittura
- *exc*: socket su cui si verifica un errore
- *timeout*: tempo massimo di attesa

I possibili valori di ritorno sono:

- ▶ -1 se vi è un errore
- ▶ 0 se si verifica un timeout
- ▶ n>0 se n descrittori diventano pronti

Esistono poi alcune macro per elaborare i socket, che si usano tramite fd_set (fd_set è un array di bit dove l'i-esimo elemento rappresenta l'i-esimo socket manipolabile tramite macro):

- FD_ZERO(fd_set *fdset): imposta a 0 tutti i bit di fdset
- FD_SET(int fd, fd_set *fdset): imposta ad 1 il bit corrispondente ad fd
- FD_CLR(int fd, fd_set *fdset): imposta a 0 il bit corrispondente ad fd
- FD_ISSET(int fd, fd_set *fdset): testa il bit corrispondente ad fd

Chiaramente bisogna porre attenzione al fatto che fd_set è un array e quindi il suo primo elemento (0) sarà il descrittore del primo socket.

Esempio di utilizzo delle system call: Client

```

#include <netinet/in.h>
#include <unistd.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>

#define MAXLINE 256
#define PORT 2000

int main(int argc, char **argv)
{
    int sockfd, n;
    char recvline[MAXLINE + 1];
    struct sockaddr_in servaddr;
    struct hostent *hp;
    struct in_addr **pptr;

    if (argc != 2){
        fprintf(stderr,"usage: %s <IPaddress>\n", argv[0]);
        exit (1);
    }

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0){
        perror("socket error");
        exit (1);
    }

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);

    /* Map host name to IP address, allowing for dotted decimal */
    if ((hp = gethostbyname(argv[1])) == NULL)
        fprintf(stderr, "hostname error for %s", argv[1]);

    pptr = (struct in_addr **) hp->h_addr_list;
    memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));

    if (connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr)) < 0){
        perror("connect error");
        exit (1);
    }

    while ((n = recv(sockfd, recvline, MAXLINE, 0)) > 0) {
        recvline[n] = 0;
        if (fputs(recvline, stdout) == EOF){
            perror("fputs error");
            exit (1);
        }
    }

    if (n < 0){
        perror("read error");
        exit (1);
    }
    close(sockfd);
    exit(0);
}// of main

```

Esempio di utilizzo delle system call: Server

```

#include <netinet/in.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <errno.h>

#define MAXLINE 256
#define PORT 2000

int main(int argc, char **argv)
{
    int
        listenfd, connfd, addrlen;
    struct sockaddr_in servaddr;
    char buff[MAXLINE];
    time_t ticks;

    < INIZIALIZZAZIONE>
    < CICLO DI ATTESA ED ELABORAZIONE RISPOSTE>
}

listenfd = socket(AF_INET, SOCK_STREAM, 0);
if (listenfd < 0){
    perror("opening socket");
    exit(1);
}

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(PORT);
a=bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
if (a < 0){
    perror("Error in binding");
    exit(1);
}

listen(listenfd, 5);

```

Distinguiamo poi la **versione iterativa**:

```
for ( ; ; ) {
    connfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
    ticks = time(NULL);
    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
    send(connfd, buff, strlen(buff), 0);
    close(connfd);
}
}
```

Da quella **concorrente**:

```
while(1) {
    addrlen = sizeof(claddr);
    connfd = accept(listenfd, (struct sockaddr *)&claddr,&addrlen);
    if(connfd>0) {
        if(fork()>0) close(connfd); //processo padre
        else {
            close(listenfd);
            ticks = time(NULL);
            snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
            send(connfd, buff, strlen(buff), 0);
            close(connfd);
            exit(0);
        }
        while(waitpid(-1,NULL,WNOHANG) > 0);
    }
    else printf("Errore in accept\n");
}
```