

Capitolo 0 – Concetti Generali di reti di calcolatori.

L'argomento di questo capitolo è essenzialmente quello che la gente dell'istituto tecnico fa già prima di noi, e che quindi è avvantaggiata siccome non ha bisogno di studiarla da capo (come noi scemi dello scientifico).

Prestazioni

Nelle reti di calcolatori, le prestazioni sono per lo più dovute da due fattori:

- La **capacità di trasmissione**, che è determinata dalle capacità delle tecnologia usata in quel momento.
- Il **ritardo di collegamento**, determinato da una serie di fattori (come distanza, tempi di gestione, protocolli usati etc...)

Con **throughput** si intende la quantità di bit effettivamente trasmessi, ovvero la quantità dei messaggi inviati con successo attraverso un canale di comunicazione (maggiori info [qui](#)). La capacità effettiva non è da confondersi con la "capacità" del link: sia la capacità che il throughput si esprimono in bit/s, ma, mentre la prima esprime la frequenza trasmissiva massima alla quale i dati possono viaggiare, il throughput è un indice dell'"effettivo" utilizzo della capacità del link.

Per calcolarlo, si usa la formula

$$TP = \text{capacità di trasmissione} - \text{overhead dei protocolli} - \text{eventuali errori di comunicazione}.$$

(Il prof qui ha confuso goodput con throughput, nel linguaggio tecnico il goodput != throughput).
Il throughput è anche influenzato dal jitter.

Con **jitter**, si intende la variazione nel tempo del ritardo di rete, in questo modo è facile vedere se il ritardo della rete è stabile o meno. In caso si ha un jitter elevato, il flusso di dati non è stabile, quindi avrà momenti in cui si trasmettono molti bit e momenti in cui invece ne trasmetto davvero pochi. Il problema di un jitter elevato è che le applicazioni devono **bufferizzare** la comunicazione per ridarla all'utente in modo costante. Inoltre, il jitter elevato richiede una grande quantità di buffer. Se non ci fosse il jitter, il mondo sarebbe un posto migliore e non ci sarebbe alcun tipo di ritardo nella comunicazione.

Definiamo Round Trip Time (**RTT**) il tempo impiegato da un piccolo pacchetto per andare da mittente a destinatario (e con cui ritorna l'ACK) e viceversa.

Componenti necessari per le reti di calcolatori

Tra i componenti necessari per collegarsi a internet, abbiamo:

- **Scheda di rete**: dispositivo collegato direttamente alla scheda madre della calcolatore, che ha il compito di trasmettere e ricevere i dati oltre che di codificarli e decodificarli nel formato che supporta il link trasmissivo. Ad ogni scheda di rete è associato un indirizzo MAC univoco a 48 bit. La scheda di rete è amministrata dal driver del sistema operativo, che permettono di fare delle specifiche chiamate di funzione.
- **Connettore di rete**: interfaccia standard che permette di collegare la scheda di rete al link trasmissivo.
- **Link trasmissivo**: mezzo fisico nel quale viaggiano (e si propagano) le informazioni sotto forma di segnale analogico. Possono essere fili di rame, cavi coassiali, fibre ottiche o l'ambiente nel caso delle onde radio.

Infrastruttura di rete

L'infrastruttura di rete è la struttura dei collegamenti di una rete.

- **Punto-punto:** connessione diretta tra due host
- **Connessioni multiple:**
 - **Completamente connessa:** esiste un collegamento diretto da ad ogni host. Massimo grado di ridondanza, molta tolleranza ai guasti ma alti costi.
 - **Minimamente connessa:** esiste un cammino da ogni host ad ogni host che eventualmente passa per nodi intermedi. Costi minimi ma grande rischio di partizionamenti di rete visto l'assenza di ridondanza.
 - **Partizione di rete:** due isole di host non connesse (ho quindi 2 gruppi isolati).

Topologia delle infrastrutture di rete

Le infrastrutture di rete possono avere diversi schemi di connessione dei dispositivi:

- **Ad anello:** ogni host è connesso direttamente ad ogni host adiacente. Esiste quindi un cammino da ad ogni host, ed i cammini girano nella rete in senso orario od antiorario. La struttura presenta un grado di ridondanza, se cade un collegamento la rete risulta comunque minimamente connessa.
- **A stella:** rete minimamente connessa, ma gli host non sono connessi direttamente, bensì si connettono ad un dispositivo centrale (uno switch o un hub) che propaga il segnale in tutte le direzioni. Questa topologia presenta un alto rischio di partizioni di rete ed il dispositivo centrale è un single point of failure.
- **A bus:** dispositivo connessi ad un bus comune a tutti. Anche in questo caso non c'è ridondanza inoltre si presenta il problema di gestire l'accesso al mezzo di comunicazione condiviso.
- **Ad albero:** struttura gerarchica in cui ogni nodo può comunicare direttamente con i suoi figli diretti.

Mezzo di trasmissione

- **Cavi o fili metallici:** permettono di codificare l'informazione digitale mediante variazioni della corrente elettrica. Possono essere doppini in rame intrecciati o cavi coassiali. Prestazioni (1-2 Gbit/s), basso costo ma suscettibili ad interferenze.
- **Fibra ottica:** fili sottili in vetro nei quali passano onde luminose che vengono codificate in base a variazioni della frequenza. Presentano le prestazioni migliori (migliaia di Gbit/s), subiscono poca interferenza (questo perché luce le frequenze della luce e delle onde radio hanno di grandezza estremamente differenti) ma hanno alti costi di giunzione.
- **Segnali radio:** trasmissione mediante onde elettromagnetiche nello spazio. Hanno il vantaggio di permettere una comunicazione in mobilità, ma presentano problemi circa le interferenze e gli errori, il problema dell'attenuazione del segnale con la distanza e la minor capacità di trasmissione.

Scheda di rete

La scheda di rete è un componente hardware dotato di PCI (che è l'interfaccia di collegamento al PC) e di connettore di rete. Questo è in grado di codificare e trasmettere, oppure ricevere e decodificare i dati inviati dal calcolatore alla rete, e dalla rete al calcolatore. La tipologia di scheda di rete dipende molto anche dal mezzo di trasmissione dei dati. La scheda di rete presenta anche un identificativo unico, ovvero l'indirizzo MAC.

Canali di comunicazione

Un singolo mezzo fisico trasmissivo può essere visto come un insieme di canali logici virtuali.

- **Canale punto-punto:** un canale logico viene riservato per far comunicare direttamente due host.
- **Canale ad accesso multiplo (broadcast):** un unico canale viene utilizzato da più dispositivi in modo che ciò che trasmette uno arriva a tutti. Presenta la problematica **delle collisioni** dovute alla sovrapposizione di segnali che transitano nello **stesso tempo sul canale**. Questa problematica è molto più semplice da gestire nei canali punto-punto.

Reti a commutazione di circuito e reti a commutazione di pacchetto.

Nelle reti a **commutazione a circuito**, per far comunicare due host si riserva un circuito di canali logici per tutta la durata della comunicazione.

Prima di iniziare la comunicazione bisogna quindi creare il circuito ed una volta che inizia tutte le risorse sono dedicate per la comunicazione end-to-end. Il vantaggio è un minor ritardo, maggiore affidabilità visto che le risorse sono allocate per tutto il tempo, inoltre la connessione è sicura.

Tuttavia presenta diverse problematiche come il fatto che **bisogna riservare risorse di fatto anche se non si sta sfruttando il circuito riservato**, quindi questo risulterà in maggiori costi di comunicazione (non è, quindi, molto utilizzato). In questo tipo di comunicazione, si paga per tempo di comunicazione.

Nelle reti a **commutazione di pacchetto**, invece, i dati vengono spezzati in pacchetti, immessi nei canali ad accesso pubblico ed affidati a dispositivi chiamati **commutatori di pacchetto** che li faranno arrivare a destinazione, seguendo gli indirizzi del mittente e del destinatario. In questo modo la rete risulta utilizzata al meglio, si trasmette solo quando si ha necessità di farlo e non si allocano risorse ma si può utilizzare la rete solo se è disponibile. Questo si traduce in **maggiori ritardi di comunicazione** da un lato (es. Ritardo di accodamento) ma anche in minori costi dall'altro, in quanto si possono allocare piu' utenti allo stesso canale condiviso. In questo tipo di comunicazione, l'ISP cerca di saturare quasi al 100% il canale di comunicazione allocando gli utenti in modo dinamico. In questo tipo di comunicazione, si paga per qta di dati trasmessi.

Servizi

I servizi offerti dalle reti nella commutazione a pacchetto dipendono dalle proprietà dei protocolli utilizzati:

- **Connection oriented:** i dati vengono spediti e ricevuti in ordine, inoltre arrivano senza errori e vengono rispediti i pacchetti andati perduti (possiamo metaforicamente associarlo al servizio telefonico). In generale, con connection oriented si intende un servizio o comunicazione in cui prima di inviare i dati si stabilisce una connessione logica fra i due utenti (es. Il 3-way handshake nel caso della TCP). Un protocollo Connection oriented si può basare su un protocollo connection-less (come nella caso della TCP).
- **Connection-less:** i dati vengono affidati alla rete che per definizione è un servizio non orientato alla connessione, perciò può perderli, farli arrivare sbagliati o non in ordine (associabile al servizio postale).

Capitolo 0.5 – Introduzione a STACK ISO/OSI e Protocolli

Protocolli

Un **protocollo** è un insieme di regole sintattiche e semantiche, che permettono la compatibilità tra dispositivi e sistemi che seguono lo stesso standard. Devono definire il formato dei messaggi scambiati e le regole comportamentali per decidere quando e come inviare i messaggi (o i dati). In generale, lo scopo di un protocollo è quello di risolvere una problematica di rete.

Le classi di protocolli sono raggruppate in livelli, ogni livello gestisce una fase della comunicazione. **Un livello x fornisce servizi al livello x+1 e richiede servizi offertogli dal livello x-1.**

La struttura a livelli standard per le reti di calcolatori è chiamata modello **ISO/OSI RM (Open System Interconnection Reference Model)**:

1. **Livello applicazione:** è la sede dei protocolli che permettono di far comunicare applicazioni remote. A questo livello le applicazioni si scambiano dati tramite i protocolli di questo livello, e questi pacchetti di informazione sono detti messaggi.
2. **Livello presentazione:** risolve eventuali eterogeneità del formato dei dati tra i nodi della rete. Questo livello si occupa anche di convertire i dati da ascii ad unicode e viceversa, siccome si occupa di presentare i dati a livello utente.[NON TRATTATO]
3. **Livello sessione:** mantiene e gestisce lo stato attuale del collegamento tra due applicazioni remote. [NON TRATTATO]
4. **Livello trasporto:** fornisce alle applicazioni i servizi orientati alla connessione e non, controlla la congestione della rete. In questo livello i messaggi vengono spezzati in segmenti. Il pacchetto qui prende il nome di **SEGMENTO**.
5. **Livello rete:** si occupa dell'instradamento dei datagrammi nella rete e di scrivere gli indirizzi di mittente e destinatario.
6. **Livello data link (o MAC/LLC):** si occupa della corretta trasmissione dei pacchetti tra nodi adiacenti della rete, in particolare in modo da evitare il jamming e collisioni. A questo livello i datagrammi sono chiamati **FRAME**. (l'acronimo MAC sta per Medium Access Control, mentre LLC sta per Logical Link Control). La parte LLC permette di gestire la correttezza della trasmissione della comunicazione, controlla gli errori, e ritrasmette i pacchetti in caso di errori.
7. **Livello fisico:** si occupa di definire le tecniche di codifica e decodifica dei dati e la loro trasmissione sul canale trasmissivo.

Ad ogni livello i protocolli aggiungono informazioni, andando ad incapsulare il messaggio dentro delle “buste”. Queste informazioni verranno poi analizzate in parte dai nodi intermedi della rete ed infine dal destinatario. Ogni livello si occupa di analizzare e scrivere nelle buste i dati relativi al suo livello.

Livello Fisico

Il livello fisico interessa la trasmissione, ricezione e codifica dei dati da bit a **segnali analogici** (e di conseguenza corrente elettrica). La velocità di propagazione dei segnali e circa quella della luce, mentre la velocità di trasmissione dipende dal numero di bit/secondo che si riescono a trasmettere. Possiamo definire la capacità del canale di trasmissione come la quantità massima di bit trasmessi al

secondo. La capacità a sua volta dipende dal tempo che ho per interpretare ogni singolo segnale analogico, e dipende da quando dura la finestrella attraverso la quale possiamo leggere ed interpretare i segnali. Dunque capacità alta → bit descritti in tempo breve, mentre capacità bassa → bit descritti in molto tempo.

Livello MAC

Il livello MAC si occupa di capire quando inviare il segnale, in modo da evitare collisioni, in poche parole, arbitra il canale.

Ogni calcolatore è dotato di una scheda di rete capace di codificare e decodificare i segnali. Ad ogni scheda di rete è associato un identificativo chiamato **indirizzo MAC**. Quando si inviano i frame sul link comunicativo si aggiunge l'indirizzo MAC del mittente e del destinatario.

In un canale condiviso, tutti riceveranno il messaggio, ma questo verrà preso e passato ai livelli superiori solo da chi possiede l'indirizzo MAC corrispondente a quello del destinatario del pacchetto.

A questo livello si vuole rendere il canale sicuro ed affidabile, e qui entra in gioco il **protocollo LLC**, che si occupa di rendere il canale affidabile, evitando errori di trasmissione. Per fare ciò, il frame viene arricchito con campi utili per identificare eventuali errori. Dopo la trasmissione il mittente fa partire un **timer**, se il pacchetto arriva al destinatario errato, esso non trasmetterà l'ACK di conferma.

Di conseguenza, se il mittente non lo riceve entro un certo lasso di tempo procederà a tentativi rispedendo il pacchetto finché riuscirà a trasmettere correttamente il messaggio.

Alcuni protocolli di questo livello sono:

- **Ethernet**: è un protocollo usato nelle reti cablate. Questo protocollo obbliga la scheda di rete ad ascoltare il canale, e se nessuno trasmette (ovvero, c'è silenzio) allora trasmetto. In caso di collisione (che posso individuare grazie alla **collision detection** garantita da questo protocollo, o CSMA/CD), smetto subito di trasmettere. Dopo aver atteso un certo lasso di tempo, ritrasmetto. Quando un frame arriva a destinazione, si manda immediatamente l'ACK di conferma così da ottimizzare l'utilizzo della rete (**piggy backing**).
- **IEEE 802.11 (o Wi-Fi)**: è il protocollo utilizzato per le reti wireless, permette di ascoltare il canale, ma non permette la Collision Detection. Dunque, usa la **collision avoidance**, ovvero appena mi accorgo che c'è "silenzio", trasmetto il pacchetto, mentre se non c'è ancora silenzio, aumento gradualmente il tempo di attesa per il tentativo di ricezione. (Più informazioni [qui](#)).
- **Token ring**: utilizzato per le reti a topologia ad anello. In base a questo protocollo, può trasmettere solo chi possiede un frame speciale chiamato token, che viene fatto passare di volta in volta a tutti gli host della rete. In questo modo si annullano le collisioni, inoltre i messaggi vengono ricevuti da tutti in senso orario od antiorario, quindi se chi riceve un messaggio possiede il token, esso può interpretare questo segnale come ACK implicito. Un problema di questo protocollo è la perdita del token. Infatti, se questo viene perso, **bisogna fare in modo che la rigenerazione del token impedisca che se ne crei più di uno** (es. se uno non sa che un altro lo sta creando).

Livello trasporto

Qui avviene anche la frammentazione dei dati in pacchetti.

Livello applicazione

Traceroute è un programma applicativo che sfrutta il protocollo di livello rete **ICMP**.

ICMP (Internet Control Message Protocol) è un protocollo del livello rete usato da host e router per scambiare messaggi attraverso il livello rete utilizzando il protocollo IP, e riguarda sia router che host. Se l'invio avviene con successo si riceve una conferma, altrimenti riceviamo un messaggio d'errore.

I messaggi scambiati hanno come contenuto informazioni sulla rete, ad esempio:

- *Rete di destinazione non raggiungibile* (possibile interruzione di rete?)
- *Rete di destinazione sconosciuta* (indirizzo di rete male specificato?)
- *Host destinazione non raggiungibile* (host spento o scollegato?)
- *Host destinazione sconosciuto* (indirizzo di host male specificato?)
- *Protocollo richiesto non disponibile* (servizi non previsti)
- *Ricerca di un cammino alternativo per la destinazione* (se esiste)

Il programma **traceroute** agisce mandando una serie di pacchetti per la rete verso una destinazione. Ognuno di questi pacchetti ha un **TTL (time to live)** che indica il numero massimo di hop che quel pacchetto può compiere. Il primo pacchetto viene mandato con un TTL pari ad 1, quindi il primo router lo decrementerà a 0 ed invierà il suo pacchetto echo di risposta che contiene il suo indirizzo IP e il RTT, ovvero il tempo di andata e ritorno. Dopodiché verrà mandato un secondo pacchetto con TTL = 2 e così via ad aumentare finché si spera di giungere alla destinazione. Router successivi possono avere RTT minore perché nel mentre le condizioni della rete possono cambiare, o perché può essere cambiato anche il percorso. Quando un pacchetto ICMP non torna entro un tempo limite (generalmente 3 secondi, ma è modificabile dall'utente), il comando traceroute sostituisce il tempo mancante con un asterisco. Generalmente vengono inviati 3 pacchetti di fila non uno solo. Un'altra applicazione è **PING**. Ping serve per verificare l'esistenza di una connessione tra due host. Viene mandato un pacchetto di tipo echo request e si fa partire un timer. Quando arriva al destinatario questo manda un messaggio echo reply al mittente che serve per calcolare il RTT, e in caso di insuccesso viene indicato che il timer per la richiesta inviata è scaduto senza ottenere risposta. Al termine dei tentativi, viene mostrato un elenco di statistiche sul numero di richieste andate a buon fine e i tempi medi stimati di andata e ritorno dei pacchetti.

Alcuni dispositivi di rete di livello 1 e 2

Livello 1:

- **Repeater**: amplifica il segnale di arrivo e lo rigenera, tuttavia unisce segmenti al livello uno, quindi non puo collegare segmenti con tecnologie diverse ed inoltre non gestisce l'accesso al canale quindi aumenta il dominio di collisione. Siccome i segnali emessi su qualsiasi mezzo fisico si degradano al crescere della distanza percorsa, esiste un limite massimo per la lunghezza di un segmento di rete. Ad esempio, un segmento Ethernet, può variare dai 100 ai 200 metri. Dunque, se volessimo avere un segmento di rete più lungo, sarà necessario un repeater.
- **Hub**: è un semplice repeater multiporta. Esso realizza il punto centrale di connessione, detto concentratore, dei segmenti di una rete locale con topologia a stella. In pratica si tratta di un ripetitore (repeater) con tante connessioni entranti e uscenti.

Livello 2:

- **Bridge**: collega segmenti di rete anche con tecnologie diverse lavorando a livello 2, ovvero traduce i frame in una tecnologia in frame di un'altra (es. Wireless e cablato), usando il protocollo MAC adatto. I bridge fanno quindi da traduttori dei frame nei formati richiesti dal livello MAC.
- **Switch**: permette anch'esso di collegare più segmenti di tecnologie diverse, traducendo i frame e usando il protocollo MAC adatto. Possiede anche una switching table, con la quale "impara" dove stanno i vari dispositivi, in quali porte. Al contrario del bridge, esso permette di connettere un numero maggiore di segmenti diversi (fino a 10 o 12). I Bridge (penso il prof intendesse gli Switch in questo caso) sono dotati della capacità di filtrare e instradare opportunamente i frame di dati sul segmento opportuno, osservando sui frame le informazioni di indirizzo MAC del dispositivo destinatario.

Livello 3:

- **Router**: i router sono dei dispositivi di livello rete che si occupano di eseguire il routing e il forwarding. Più info [qui](#).

Capitolo 1 – Application Layer – Livello 7

The Internet. LOL, the internet (ISP, Network Core, Network Edge)

Internet è una rete di reti, in cui ci sono tanti dispositivi che sono interconnessi grazie reti wireless o cablate. In questi dispositivi, detti **host** (o end-systems), girano le applicazioni di rete.

Internet è strutturata secondo un sistema gerarchico che vede gli **ISP regionali** che offrono servizi agli end-system e i **global ISP** che, a loro volta, offrono servizi agli ISP regionali (I regional ISP sono quindi connessi con I global ISP). Essendo internet una rete a comunicazione di pacchetto, l'unità di informazione base che viaggia su internet è, appunto, **il pacchetto**, e i dispositivi che si occupano dell' inoltro dei pacchetti su internet sono i commutatori di pacchetto, router o switch.

I protocolli si trovano nelle **RFC** (request for comments) che vengono creati dalla **IETF** (Internet Engineering Task Force), che sono scritti in testo ASCII e che originariamente erano necessari per chiedere appunto commenti circa la creazione di un nuovo protocollo.

La struttura di internet è costituita da:

- **Network Edge:** è composto dagli host, sia client che server. Spesso i regional ISP sono considerati edge.
- **Network Core:** sono router che cumincano tra di loro, e che trasmettono sulla dorsale di Internet (ovvero la backbone).
- **Access Networks:** rete regionale, che ci collega all'internet core, composta dagli edge routers.

Per collegare gli end-systems all'edge router, possiamo fare uso di reti istituzionali, access point domestici oppure reti di accesso per dispositivi mobili. Quando ci connettiamo al router di una rete locale, solo se il router di questa rete locale è connesso tramite un regional ISP, allora abbiamo accesso ad internet.

Network Edge (DSL, DSLAM, DMZ, ritardo di trasmissione, mezzi guidati e non)

Con il termine **bandwidth** indichiamo la larghezza di banda nominale di una rete. Tuttavia tale larghezza di banda non può essere dedicata ad ogni singolo utente, ma al contrario potrebbe essere condivisa e di conseguenza l'effettiva velocità di trasmissione potrebbe essere inferiore. Questa è una problematica delle **reti edge**, mentre le **reti core** sono tarate per permettere una comunicazione sempre efficiente e veloce.

Una rete di accesso residenziale è generalmente costituita da un **modem DSL** (che sta per Digital Subscriber Line), che spesso possiede anche funzionalità di access point wireless e di router/switch e che è connesso ad una presa telefonica.

Possiamo trovare 2 tipi principali di DSL, l'ADSL (che è asimmetrica, e quindi la velocità del download è diversa da quella dell'upload) e il VDSLAM (che invece è molto veloce, e spesso viene usato nelle connessioni in fibra ottica).

Il **modem** permette di trasmettere i dati in arrivo lungo la rete telefonica, producendo variazione di corrente a determinate frequenze. Ad esempio la voce viaggia su frequenze da 0 a 4 KHz, da 4 a 50 KHz i segnali di upload e da 50 KHz in su i segnali in download. In questo modo è possibile distinguere i segnali delle chiamate telefoniche da quelli destinati ad internet, pur avendo un unico mezzo trasmissivo (come abbiamo visto nel).

I dati in arrivo alla centrale operativa dell'ISP finiscono in un **DSLAM** (Digital Subscriber Line Access Multiplexer, che si trova nella centrale di commutazione dell'ISP, ovvero in un POP dell'ISP) che instrada i pacchetti verso la rete internet, mentre le chiamate vanno nella rete telefonica.

Abbiamo poi le **cable networks**, in cui essenzialmente si ha un unico cavo coassiale che viene condiviso da più host. Per trasmettere tutti questi segnali mantenendoli distinti, si fa uso di canali: abbiamo infatti diversi canali, che sono caratterizzati da bande di frequenze diversi. Questo viene chiamato **FDM** (Frequency Division Multiplexing).

La rete infrastrutturale di un'azienda è più complessa: il **router istituzionale** analizza i dati in arrivo. Infatti bisogna sempre distinguere gli host degli impiegati e gli **host nella DMZ** (zona demilitarizzata). In questa zona (DMZ) vi si collocano i server aziendali che devono poter essere raggiunti dall'esterno, mentre gli host privati aziendali non devono essere raggiungibili dall'esterno, di conseguenza se il router vede arrivarsi dall'esterno pacchetti destinati ad host privati aziendali, li bloccherà per ragioni di sicurezza.

Uno dei parametri che ci permettono di misurare la bontà di una rete è **il ritardo di trasmissione**. Il ritardo di trasmissione di un pacchetto è il rapporto tra gli L bit del pacchetto e la velocità del link in cui viaggia R bit/s.

$$\text{Ritardo} = L/R$$

I link di trasmissione possono essere **guidati**, se l'informazione viaggia vincolata nel mezzo fisico come ad esempio nel filo in rame, mentre quelli **non guidati** non sono vincolati e viaggiano nell'ambiente, come ad esempio le onde radio.

Il vantaggio dei mezzi non guidati rispetto a quelli guidati è che permettono una maggiore mobilità, tuttavia i segnali in questo modo sopravvivono meno e subiscono più interferenza, di conseguenza necessitano di essere trasmessi a maggior intensità consumando più energia e, inoltre, viaggiando nell'ambiente arrivano dappertutto, potenzialmente anche a malintenzionati.

I **cavi ethernet** sono costituiti da fili di rame intrecciati schermati per ridurre l'interferenza esterna. I cavi categoria 5 permettono di viaggiare a 100 Mbps, mentre i categoria 6 sono schermati meglio e raggiungono anche 10 Gbps.

I **cavi coassiali** sono due conduttori di rame concentrici isolati tramite un isolante in mezzo, e la differenza di potenziale che si crea tra i due conduttori di rame genera sinusoidi che possono essere differenziate in base alla frequenza.

Network Core (S&F, buffer, FDM, TDM)

L'architettura core è una mesh di router interconnessi. I router della rete core prendono i pacchetti che arrivano dalle reti di accesso e li inoltrano, attraverso un meccanismo di **Store&Forward**, che consiste nel conservare i nei router intermedi finché l'intero pacchetto non è stato ricevuto, per poi inoltrarlo al router successivo o al destinatario. I link che connettono i router core sono fibre ottiche ad alte prestazioni.

Quando abbiamo un file da trasmettere lo dividiamo in pacchetti di dimensione. Tenendo a mente che i router eseguono lo S&F, per spedire un pacchetto il ritardo end to end è:

$$\text{Ritardo} = (N * L) / R$$

dove N è il numero di collegamenti e, come prima, R è la velocità del link in cui viaggiano i dati e L è il numero dei bit.

Generalmente i pacchetti vengono accumulati in una coda in un buffer. Vi è un buffer per ogni porta di ingresso ed ogni porta di uscita. Quando il pacchetto viene messo in una coda di uscita deve aspettare finché il canale non è libero, e questo risulta in perdita di pacchetti (o packet loss) nel caso le code siano piene.

Ogni router deve incarnare due aspetti principali:

1. i **routing protocol**, ovvero come scrivere (e creare) le tabelle di routing attraverso algoritmi di routing, in modo da stabilire quale linea di uscita far prendere ad ogni pacchetto in arrivo.
2. il **forwarding**, ovvero analizzare l'intestazione di ogni pacchetto in ingresso e destinarlo a una linea di uscita, instradandolo fisicamente. Per fare l'inoltro, si usano le tabelle di routing scritte dall'algoritmo di routing.

[Quindi: routing → scrivere tabelle di routing, e forwarding → usare le tabelle di routing]

I routing protocol devono essere coerenti e basati su regole comuni tra i router. Un **autonomous system** è un'isola di router che è gestita tramite le medesime regole. Il ritardo di trasmissione (o

tempo di push-out) indica il tempo che ci mette un pacchetto ad essere trasmesso (o meglio, buttato fuori).

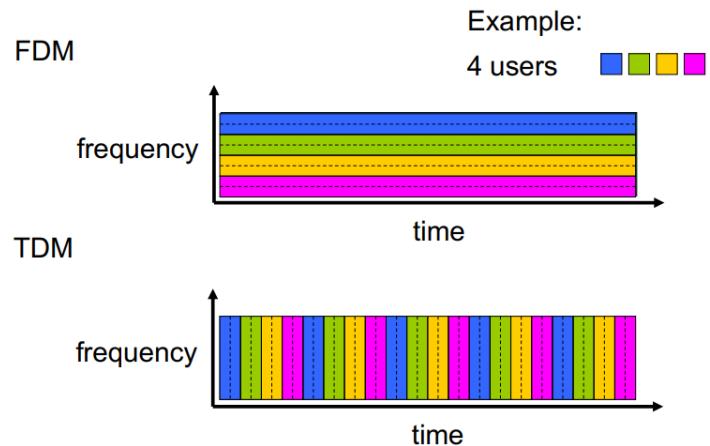
I router possono anche usare un altro modo di operare, attraverso l'implementazione del circuit switching su una rete internet. Per fare ciò, come abbiamo visto, normalmente bisogna allocare risorse end-to-end, che a loro volta non potranno essere usate da altri, e perciò saranno solo mie (come nella rete telefonica). Per evitare di dover creare più link (ad es. Creare un link per l'upload e uno per il download), è possibile dividere lo stesso link in più canali logici virtuali in due modi:

1. tramite suddivisione di frequenze

(**FDM**), con questo metodo, divido il link in tanti "intervalli" di frequenze, e in questo modo ogni user avrà la sua banda sulla quale trasmettere. In questo modo, i possono essere riconosciuti dai vari destinatari distinguendo le frequenze.

2. suddivisione di tempo (TDM)

Il tempo viene suddiviso in frame e ad ogni comunicazione viene riservato uno slot all'interno del frame.



Nonostante ciò, per una questione di probabilità¹, è difficile che molte persone siano connesse contemporaneamente, per questo è preferibile utilizzare packet switching siccome per la stessa quantità di risorse disponibili permette di accomodare più utenti assicurando prestazioni simili. Inoltre nel packet switching **non c'è il call setup**, ovvero la creazione del circuito tra mittente e destinatario, che può farci perdere del tempo. I downside dell'utilizzo del packet-switching sono il **congestionamento**, i **ritardi di accodamento** e il **possibile packet loss** (che sono tutte cose che dovremo gestire attraverso protocolli).

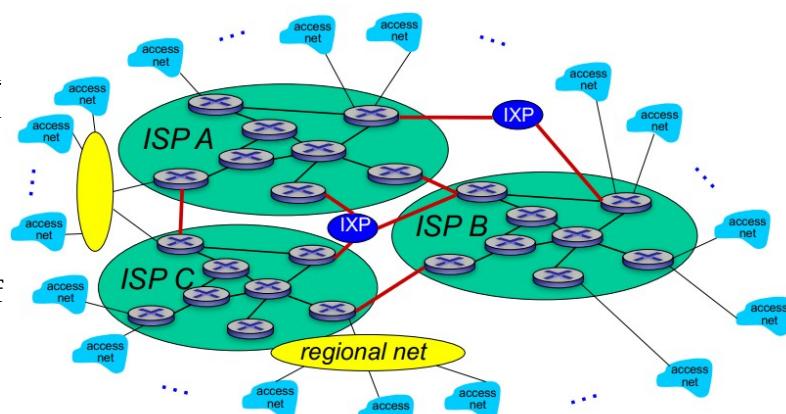
Network Core – Struttura di Internet (gISP, rISP, IXP, POP)

Per consentire una comunicazione globale bisogna interconnettere le reti di accesso tra di loro. Una soluzione ingenua sarebbe connettere ogni access ISP ad ogni access ISP, ma in questo modo la rete non scala in quanto avremmo bisogno di N connessioni (che sono davvero tanti).

Per risolvere questo problema, Internet possiede una architettura gerarchica scalabile, attraverso l'uso di una rete di **global ISP** che connettono tra di loro i vari **ISP regionali**, che sono connessi fra loro mediante l'uso di **peering link e IXP** (Internet eXchange Points, punti di scambio ad altissime prestazioni).

I **client ISP** si connettono ai fornitori (che sono in questo caso gli ISP globali) mediante i Point of Presence (**POP**), ovvero un gruppo di router dell'ISP tramite i quali i client ISP si connettono al fornitore e che trasmettono i dati attraverso la backbone di internet.

Un meccanismo parallelo è costituito dalle **reti dei fornitori di contenuti** (ad esempio Google), in cui un'azienda crea la sua rete privata che ha come scopo principale quello di connettere i suoi data centers, e che allo stesso tempo li connette ad internet spesso bypassando i regional ISP e i tier-1



1. Ad esempio se abbiamo utenti attivi il 10%, che usano 100kbps su un link di 1 mbps ne possiamo accomodare al massimo 10 con circuit switching. Usano la distribuzione binomiale secondo la formula $(n k)(p)^k (q)^{n-k}$ si ha lo 0.0004 di probabilità che con 35 persone accomodate più di 10 stiano trasmettendo.

ISP, e collegandosi agli ISP di livello ancora più basso o tramite peering (operazione di collegamento fra host che appartengono ad ISP diversi) oppure collegandosi ai loro IXP (anche se comunque è obbligata a collegarsi ad alcuni tier-1 ISP (sinonimo di global ISP) siccome è l'unico modo con cui può accedere ad alcune access networks). In questo modo, Google ha un maggiore controllo del suo servizio e riduce i pagamenti agli upper-tier ISP (es. Regionali e tier-1).

Network Core – Ritardi

Gli elementi che producono ritardi dovuti ad un router sono:

- **Nodal processing:** tempo che un nodo della rete impiega per processare il pacchetto, dovuto al controllo di errori e scelta del link di uscita
- **Ritardo di accodamento:** dovuto al tempo che un pacchetto deve aspettare in coda (nel buffer) prima di poter essere trasmesso sul link fisico (è dipendente dalla congestione del router);
- **Ritardo di trasmissione:** tempo di trasmissione di un pacchetto, dipende dalla larghezza di banda e dalla dimensione del pacchetto;
- **Ritardo di propagazione:** tempo che ci mette il singolo bit ad attraversare il link, dipende dalla distanza fisica tra il router e il nodo successivo.

Il fattore **La/R** (o **intensità di traffico** in un nodo/router, in cui **L** sono i bit si ogni pacchetto, **a** i pacchetti in arrivo ogni secondo ed **R** la velocità di trasmissione), rappresenta il rapporto di bit in arrivo e bit in uscita. Se questo rapporto è minore di 1 vuol dire che quello che arriva è minore di quello che parte, quindi siamo ancora al di sotto del livello di congestione. Tuttavia quando questo rapporto è maggiore di 1, il ritardo di accodamento inizia a tendere ad infinito.

$$[\text{INTENSITÀ DI TRAFFICO} = L \cdot A/R]$$

L=bit per pacchetto

R=la velocità di trasmissione

A=pkgs in arrivo

Per misurare questi fattori si usa il programma **Traceroute**. **[L'INTENSITÀ DI TRAFFICO È UN PARAMETRO MOLTO IMPORTANTE, L'HA CHIESTO E NON SAPEVO RISONDERE :c].**

Il **throughput** è la capacità di consegna dei dati in bit al secondo da mittente a destinatario in quantità effettive (quindi senza headers, o altri overhead). Il **throughput istantaneo** è la velocità con cui si stanno ricevendo i dati, il **throughput medio** è la velocità di ricezione dei dati in un certo lasso di tempo. Il **throughput end-to-end** dipende dal link più lento nel percorso mittente destinatario, e questo link è detto bottleneck link. Generalmente il collo di bottiglia si ha nei link della rete edge.

Si definisce invece **goodput** la quantità di dati utili nell'unità di tempo del throughput trasmesso scartando la extrainformazione o informazione di overhead associata ai protocolli durante la trasmissione.

Capitolo 2 – Application Layer – Livello 7

Applicazioni Client-Server e P2P, Socket

I programmi applicativi di rete devono essere fatti per essere eseguiti su diversi dispositivi attraverso Internet.

L'architettura **client-server** si basa sui concetti di client e di server.

- Un **server** è un host sempre connesso, sempre raggiungibile, che non muta il suo indirizzo IP e che offre determinati servizi.
- Un **client** è un fruitore di servizi, può mutare il suo indirizzo IP e può connettersi solo quando gli è necessario. I client non comunicano direttamente tra di loro.

Nell'**architettura P2P** una macchina può svolgere sia il ruolo di client che di server. Quando un peer ha bisogno di un file effettua una richiesta e il primo peer che può risolvere tale richiesta fungerà da server per inviare quel file. Quindi i peer comunicano direttamente tra loro, e ogni richiedente diventa anche un potenziale fornitore, di conseguenza l'architettura P2P è fortemente scalabile. Il problema principale di questo paradigma è che non essendoci un dispositivo fisso a cui fare richiesta, tutti i peer possono cambiare il loro indirizzo IP e la gestione della rete diventa complicata quando bisogna determinare a chi richiedere un determinato file. Oltre a ciò, è necessario gestire il problema del capire quale client può farmi da server e quale no, e a chi mandare la richiesta per capire ciò. Possiamo fare uso o di un server “di indirizzamento”, che però genera un single point of failure, oppure possiamo usare un *shortlist* di utenti che mi dice che in quella determinata fascia oraria ci sono determinate macchine attive che possono condividere dati.

Le applicazioni comunicano tramite **processi che si scambiano messaggi**. Se due processi appartengono allo stesso host, allora questi possono comunicare mediante l'inter-process communication (/DAVOLI) definito dal sistema operativo o potrebbero utilizzare lo stack ISO/OSI per inviare e ricevere il messaggio sulla stessa macchina.

Se la comunicazione avviene tra host diversi si deve stabilire una connessione tra socket. Sia chi trasmette che chi riceve utilizza un socket in modo analogo ad una porta: chi trasmette spinge i dati verso il socket mentre chi riceve preleva i dati dal suo socket (possiamo vedere i socket come tante caselle di posta). Nell'architettura ISO/OSI i primi quattro livelli vengono gestiti dal sistema operativo, mentre l'utente può creare le applicazioni utilizzando le API del livello applicazione. Per scambiare messaggi, i processi devono mantere degli identificatori, in modo da capire a quale processo sulla macchina dobbiamo mandare i dati. Un identificatore deve includere sia l'indirizzo IP della macchina sia il numero di porta, un numero da 16 bit che funziona come un identificatore di socket e che ci permette di stabilire la buchetta della posta di un processo. Un esempio di porta è la famosa porta 80, usata dal protocollo HTTP.

I protocolli del livello applicazione devono definire il tipo dei messaggi scambiati, la sintassi di questi messaggi e la loro semantiche e le regole comportamentali per decidere quando e come inviare o rispondere a questi messaggi. I protocolli possono essere **open**, nel senso che chiunque può vedere la definizione del protocollo, di conseguenza studiarlo e sfruttarlo al meglio, o **proprietari** per questioni economiche (e tal volta di sicurezza), in modo che non si possano eseguire “copie” della stessa applicazione ma fatte da altri.

settare sempre dimensioni, immagini

overhead = contenuto di giubilo, rispetto al contenuto effettivo

Livello trasporto al supporto del livello applicativo, TCP e UDP nelle app

I servizi che le app necessitano a **livello trasporto** sono:

- **Data Integrity**, certe applicazioni necessitano che i dati siano trasmessi con successo e senza errori. Alcune app sono tolleranti ad un certo margine di errore.
- **Timing**, alcune applicazioni necessitano bassi ritardi e bassa variabilità del ritardo (jitter) per tarare nel modo giusto la connessione (es. Giochi online o online calls).
- **Throughput**: alcune app possono richiedere che un minimo throughput sia garantito per consentire una comunicazione efficiente (es. App multimediali), altre app ‘elastiche’ possono funzionare anche se in alcuni momenti il throughput subisce dei cali significativi.
- **Security**: alcune app necessitano che la comunicazione sia sicura, che posso fare attraverso la crittazione dei dati.

I protocolli del livello trasporto che offrono servizi alle applicazioni sono **TCP e UDP**.

TCP	UDP
<p>TCP gestisce la comunicazione in modo che:</p> <ul style="list-style-type: none"> • <u>tutti i dati siano recepiti senza errori</u> • regola il congestionamento della rete (<i>Congestion Control</i>) • <u>non spedisce i dati più velocemente di quando il destinatario possa riceverli</u>. (<i>Flow Control</i>) <p>NON garantisce, invece:</p> <ul style="list-style-type: none"> • timing • il throughput • la sicurezza (anche se una connessione TCP può essere comunque resa sicura tramite SSL) <p>La connessione TCP è connection oriented, ovvero viene stabilita una connessione prima che avvenga la trasmissione effettiva dei dati (3-way handshake).</p>	<p>UDP <i>no connection establishment</i> <i>no connection control for receiver</i></p> <p>UDP non fornisce alcun servizio, NON ASSICURA SU NULLA!!</p> <p>È un protocollo più leggero di TCP e viene utilizzato da quelle app che non hanno bisogno di molte garanzie per quanto riguarda la comunicazione.</p> <p>In poche parole, serve per bypassare il livello trasporto.</p> <p>(Es. Di servizi che possono fare uso di UDP sono: <u>DNS</u>, <u>DHCP</u>, <u>ICMP</u> siccome se trovano un errore nel pacchetto possono rispedirlo senza alcun problema, e anche video-streaming siccome <u>se ci sono degli errori sono comunque piccoli e non troppo “notabili”</u>, e in questo modo <u>il video arriva più velocemente dato che non devo gestire ritrasmissione</u>: rinuncio a un po’ di qualità per fluidità);</p> <p>Dunque, nei protocolli in cui non ho una “serie di pacchetti” da inviare (a parte nello streaming) posso usare UDP senza problemi.</p>

Per rendere TCP sicuro bisogna aggiungere il protocollo **SSL**, che, **aggiendo a livello applicazione**, serve a rendere cifrate le connessioni TCP, senza agire sulle connessioni in sé.

Web e HTTP

L’obiettivo del servizio Web è condividere documenti. I documenti web sono scritti nel linguaggio HTML. Quando un browser scarica una pagina Web (da un **web server**), esso mostra tutto il suo contenuto testuale ed il suo formato. Eventuali altri oggetti che sono inclusi (linkati) nel documento HTML vengono rilevati dal browser quando esso ne fa il parsing del documento, e successivamente verranno scaricati e disposti nel riquadro apposito secondo quanto definito dal contenuto del documento HTML. Ogni oggetto possiede un indirizzo univoco chiamato **URL**. Un **URL** è composto dal hostname e dal percorso in cui questo oggetto si trova all’interno del sistema.

HTTP è il protocollo che definisce lo standard di come effettuare le richieste e inviare le risposte nel web (ovvero ai web server). **HTTP si basa su TCP**. Il client inizializza una connessione TCP sulla **porta 80** verso il server. Se il server accetta la connessione può iniziare lo scambio dei

messaggi. I messaggi scambiati sono messaggi HTTP che il client invia per mezzo di un **browser** ad un **Web Server**.

Quando il client ha ricevuto la pagina web, la connessione TCP cade (questo nella versione HTTP/1.1 e HTTP/1.0), e se nel documento vi era un link ad un altro oggetto bisogna ristabilire un'altra connessione TCP. Questo viene fatto per evitare di lasciare allocate inutilmente risorse da parte del Web Server (Connessione NON Persistente). Tuttavia, vista l'evoluzione delle pagine web, chiudere ogni volta la connessione può essere sconveniente, per questo nella release 2.0 del protocollo HTTP è il client a decidere quando buttare giù la connessione (Connessione Persistente). Nel caso di connessione NON persistente (HTTP/1.1 in giù), ogni volta bisogna aprire la connessione (1 RTT) e poi richiedere il file (1 RTT + file transfer). Quindi per ogni oggetto il tempo è di 2 RTT + file transfer. Nelle connessioni persistenti (HTTP/2.0), la connessione TCP viene lasciata aperta quindi per ogni oggetto il tempo è 1 RTT + file transfer.

HTTP è senza stato (**stateless/state-less**), quindi i server non mantengono informazioni delle richieste effettuate dai client.

Possiamo distinguere i messaggi HTTP in messaggi di richiesta o di risposta.

Un **messaggio HTTP di richiesta** include sempre una riga di richiesta, che è suddivisa in un campo metodo, un campo pathname e un campo versione HTTP. Poi si trovano le **header lines**. Ogni header line è nel formato campo : valore, alla fine di ogni riga si aggiungono i simboli `\r\n` (a capo e nuova linea). Per terminare si aggiunge una riga vuota con solo `\r\n`, poi c'è il corpo del messaggio. I metodi possono essere GET, POST, HEAD, PUT, DELETE:

- **GET** serve per richiedere pagine web.
- **POST** per caricare sul server informazioni come ad esempio quando si digita sul form di ricerca di un motore di ricerca. Le informazioni che scriviamo nel form sono immesse nel campo body del messaggio HTTP. Tuttavia questo è possibile farlo anche con il metodo GET estendendo l'URL con le informazioni che immettiamo.
- Il metodo **HEAD** è simile a GET ma non preleva gli oggetti richiesti.

Questi tre metodi sono inclusi in HTTP\1.0, HTTP\1.1 include anche i metodi **PUT** e **DELETE**.

- **PUT**, mette la risorsa dentro a un web server.
- **DELETE**, cancella la risorsa da un web server.

Non tutti i campi delle header lines sono obbligatori, tranne il campo Host, che indica a chi fare richiesta. Altri campi possono riguardare il formato in cui ricevere i messaggi, il browser utilizzato e se lasciare aperta la connessione o no. Una tipica risposta HTTP inizia con una linea di stato che inizia con la versione di HTTP usata e un numero che indica lo stato della richiesta. Le header lines includono informazioni circa il documento richiesto come la data dell'ultima modifica, importante perché se il client ha in cache una copia del file con data successiva all'ultima modifica, allora non è necessario il trasferimento dello stesso file. Poi il server che ha soddisfatto la richiesta ed altre informazioni sul formato dei dati e sulla connessione.

I codici di stato della prima linea possono essere:

- 200 OK, ovvero richiesta soddisfatta;
- 301 Moved Permanently, ovvero il file non si trova più in quel server e la nuova locazione viene indicata nel campo Location;
- 400 Bad Request, il server non ha compreso il messaggio;

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

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

- 404 Not Found, il documento richiesto non è presente su quel server;
- 505 HTTP Version Not Supported.

Cookies

non ci si può basare completamente sui cookie

Diversi Web Server utilizzano i **cookies** per tenere traccia di chi siano i client che fanno richiesta e quale stato della comunicazione è stato raggiunto. I cookies non sono altro che una riga di header nel messaggio HTTP. Un cookie è un valore intero numerico. La prima volta che un client contatta una piattaforma, il server crea un ID del client con un numero e lo inserisce nel suo database. L'ID viene inserito anche nella risposta HTTP nel campo set-cookie. A questo punto il browser associa quel server a quel cookie all'interno di una cache, e quando contatta nuovamente quel server va settato il valore cookie. I cookie servono per mantenere lo stato anche se la connessione TCP cade nel frattempo o cambiamo indirizzo IP. Da browser diversi si avranno dei cookie diversi, ma volendo il server può tenere traccia di entrambi i cookies ed identificare il client con entrambi. Spesso i cookie possono rappresentare anche un rischio di sicurezza, siccome possono essere usati per identificarmi come qualcun altro.

Server Proxy

Il **proxy server** è una macchina che si frappone tra chi fa le richieste e il server a cui vengono effettuate. Se una pagina richiesta si trova all'interno della **cache** del proxy server questo restituisce la pagina senza contattare direttamente il server di origine, in caso contrario il proxy web richiede la pagina al server e se la salva in cache e poi la inoltra al client. I proxy server vengono installati all'interno di reti locali dove generalmente la velocità di connessione è molto elevata.

Il **web caching** è utilizzato dagli ISP che conoscono le richieste che gli utenti effettuano su internet e di conseguenza possono salvarsi le pagine che richiediamo per ridurre il traffico di dati su internet (e quindi alleggerire il carico ai router) e aumentare la responsività della comunicazione. Se un proxy server o il nostro browser hanno nella cache una versione obsoleta della pagina web da noi richiesta, vorremmo trovare il modo di ricevere la versione aggiornata. In questo caso, si aggiunge nelle header lines del metodo GET il campo If-modified-since<date>. Se il file non è stato modificato dopo la data specificata allora il server produce un messaggio 304 Not Modified, altrimenti arriva la pagina aggiornata che va sovrascritta a quella in cache.

I proxy server possono rappresentare un rischio per la sicurezza, siccome possono essere usati con funzione di spyware.

E-mail (normalmente porta 25)

Il **protocollo SMTP** è il protocollo principale per la posta elettronica, ed assieme alle applicazioni utente e ai mail servers, compongono il **servizio di posta elettronica**.

[?/]Lo User Agent è lo strumento utilizzato per comporre o leggere le e-mail[?/]. I messaggi vengono inviati tramite il protocollo SMTP dal lato client di un server mail al mail server di destinazione. Chi ha un account e-mail ha una **quota di disco** in un server SMTP (o meglio, mail server) del dominio di posta elettronica (**Mailbox**). I messaggi che non possono essere inoltrati al mail server di destinazione vengono immagazzinati in una **coda**. **SMTP usa una connessione TCP persistente**. I metodi di cui SMTP fa uso sono:

- **HELO**, che indica l'host da cui proviene la mail
- **MAIL FROM** che indica l'indirizzo mail del mittente
- **RCPT TO** che indica a chi inviare la mail
- **DATA** che serve per indicare il messaggio che si vuole inviare (si indica la fine del messaggio con i caratteri “.” poi a capo e “...”)
- **QUIT** per chiudere la connessione TCP.

La differenza tra HTTP e SMTP è che il primo è un **protocollo PULL** perché tende a tirare a sé gli oggetti che vengono prelevati dai server, mentre SMTP è un **protocollo PUSH** in quanto si inviano i messaggi verso i mail server.

Lo standard per i messaggi di posta elettronica contenuto nel RFC 822 include 3 righe di header che sono From, To e Subject e che sono differenti da quelli di SMTP e in seguito il body del messaggio separato da una riga vuota.

I protocolli usati per prelevare la posta dal proprio mail server sono **POP3, IMAP e HTTP**. Nel protocollo **POP3** (porta 110) vi è una fase di autenticazione tra client e server in cui vengono inviati username e password. Se le autenticazioni vanno a buon fine il server risponde con OK altrimenti ERR. Poi è possibile scaricare i propri messaggi con i comandi `list` che genera una lista dei nuovi messaggi, `retr` che ritira un messaggio, `dele` che lo elimina e `quit` per terminare la sessione. Il POP3 ha due limitazioni, l'autenticazione avviene in chiaro e quando ritiriamo un messaggio nella modalità scarica e cancella, la copia sul server viene eliminata, quindi non si può scaricare la posta da un altro client. **IMAP** presenta numerosi vantaggi rispetto a **POP3** come la possibilità di accedere da remoto alle proprie mail che si trovano tutte sul server, la possibilità di mettere le mail in diverse cartelle e il fatto che IMAP mantenga lo stato tra una sessione e l'altra, come ad esempio i nomi delle cartelle e l'associazione tra messaggi e cartelle. Infine si può usare un browser come client mail e quindi sfruttare il protocollo **HTTP** per inviare o ricevere messaggi. Tuttavia i server mail che comunicano utilizzano ancora SMTP per recapitare i messaggi.

DNS (normalmente porta 53)

Il DNS è un servizio che permette di associare a un indirizzo IP un nome che può essere facilmente ricordabile da una persona (al contrario dell'indirizzo IP, che sono essenzialmente una serie di numeri). Il DNS è implementato tramite una gerarchia di Nameservers che possono essere contattati tramite un protocollo per risolvere un nome di dominio in un indirizzo IP. Il servizio DNS è considerato una funzionalità stessa di internet, e i server sono distribuiti nella parte edge di internet, perché applicare una politica distribuita permette di gestire a livello locale il lavoro che i server DNS devono svolgere. Altre funzionalità del servizio DNS sono la possibilità di fare **aliasing**, ovvero ad uno stesso indirizzo IP possono essere associati più nomi e anche la possibilità di associare più indirizzi IP ad uno stesso nome, in modo da **distribuire** le richieste che vengono effettuate ad un unico web server (ad esempio www.google.com).

[pi/] Il motivo per cui i DNS non vengono gestiti in un unico server DNS è che il sistema non scalerebbe [/pi]. Il servizio DNS fa uso anche di un procollo dell'application layer. Questo protocollo stabilisce la **sintassi dei messaggi di richiesta e risposta** che appunto permettono di implementare questo servizio.

Il DNS funziona con un meccanismo ad albero: la richiesta dell'IP associato al nome parte dal basso (ovvero, dalle foglie) e se il server DNS (ovvero il DNS recursive resolver) non sa risolvere tale richiesta la manda verso l'alto fino ad arrivare ai **Root DNS** server che fa ridiscendere le richieste nel lato giusto dell'albero (solitamente, un Root DNS ridireziona a un TLD, il quale invierà la risposta all'host, che lo direzionerà ad un authoritative DNS). Nel livello più basso dell'albero ci sono gli authoritative DNS come amazon o yahoo (in particolare, gli URL di questa piattaforma/e). Al livello più alto ci sono i domini come .com, .org o .it (Top level domain, o TLD) ed infine la radice. Un **TLD nameserver** contiene tutte le informazioni sui domain name che hanno lo stesso suffisso (es. Tutti i .org, .com...). A volte ci possono essere DNS ancora più locali rispetto agli authoritative DNS.

Ogni ISP possiede il suo DNS server e quando si effettuano richieste DNS si contatta prima questo. Generalmente questo server ha una cache in cui contiene tutti i mapping nome-indirizzo IP di richieste precedentemente effettuate. Se non conosce l'associazione dovrà propagare la richiesta verso l'alto.

La richiesta DNS può essere di tipo **iterativo**, in questo caso si contatta il proprio DNS locale. Se esso sa la risposta la darà al client, altrimenti contatterà il server DNS di livello superiore che gli dirà il DNS da contattare per ottenere la risposta. Quindi ad ogni iterazione il local DNS otterrà il nome del server successivo da contattare. Tutto il lavoro in questo caso viene fatto dal local server DNS. In questo caso, però, l'effetto del caching sarebbe inutile. Questo metodo viene usato soprattutto per i domini che hanno un unico DNS server, in modo che si riduca il carico di lavoro.

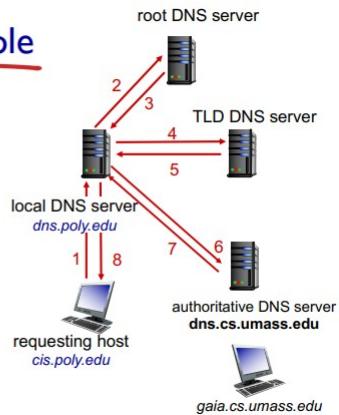
Un'alternativa consiste nell'implementare la richiesta DNS in modo **ricorsivo**. Ad ogni passo il server interpellato diventa il client che fa la richiesta al server successivo in maniera ricorsiva. In questo modo il local DNS avrà meno carico di lavoro a discapito del root DNS.

DNS name resolution example

- host at cis.poly.edu wants IP address for gaia.cs.umass.edu

iterated query:

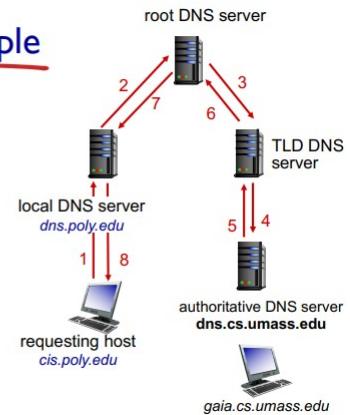
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example

recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



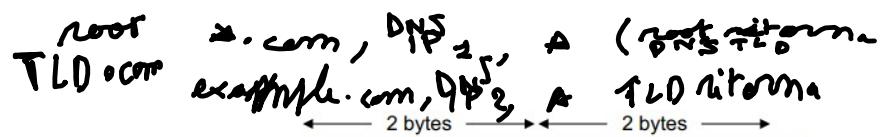
Il protocollo DNS implementa il protocollo **UDP**, siccome se non vede ritornare la risposta entro un certo limite la rimanda lui stesso, rifà la richiesta senza problemi (senza bisogno di un sistema di TCP di rinvio separato. In sostanza, ci pensa già il DNS. In poche parole, **gestisce lui stesso il packet-loss**).

Parliamo ora delle **cache del DNS**. Una cache non può essere valida indefinitamente perché le associazioni potrebbe cambiare. Tutte le entry a livello cache hanno un time to live (ttl nei Resource Records). Inoltre, possono essere invalidate le tabelle di cache degli altri DNS NameServers se bisogna aggiornare un'informazione (DNS Poisoning).

Il servizio DNS, oltre al protocollo di cui abbiamo parlato prima, ha bisogno anche di un database di associazioni nome – IP. Queste entry sono chiamate **DNS record (o Resource Records (RR))** e sono nel formato (name, value, type, ttl). A seconda del valore di type gli altri campi assumono diversi significati:

- **type = A** (Quello principale)
 - name è l'*hostname*
 - value è l'*indirizzo IP*;
- **type = NS** (Name Server)
 - name è il *dominio*
 - value è l'*hostname* dell'**authoritative DNS** per questo dominio.
 - Per esempio, (foo.com, dns.foo.com, NS, ttl) è un record di tipo NS;
name value
- **type = CNAME**,
 - name è l'alias per qualche nome canonico
 - value è il nome canonico.
- **type = MX (Mail Exchange)**,
 - name dominio (es. gmail)
 - value è il mail server associato al dominio scritto in name.

Il processo di "query" per ottenere l'ind. IP del dominio sarà prima di andare a cercare il record di tipo NS per trovare il suo authoritative DNS, e poi un'altra query di tipo "A" che avrà associato il nome che cercavo.



Sia richieste che risposte DNS hanno lo stesso formato.

- I primi 16 bit sono dedicati all'identificazione in cui viene indicato il numero della domanda che effettuiamo, e la risposta userà lo stesso identification number.
- Altri 16 bit vengono dedicati al **flags** in cui si specifica che si tratta di una query o di una risposta, se la query è ricorsiva o meno, se la ricorsione è disponibile o se la risposta arriva da un DNS authoritative.
- Nelle quattro righe successive troviamo i campi che indicano le quantità di occorrenze delle quattro righe che seguono l'intestazione.
- Dopo l'intestazione troviamo nella prima riga la query che include il nome che si sta cercando e il tipo di richiesta (A, MX, ecc).
- Il campo **risposta** include gli RR in risposta alla query (alla richiesta), il campo **authority** contiene i record di altri server authoritative, infine il campo **addizionale** che include informazioni aggiuntive.

identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
<u>questions (variable # of questions)</u>	
	answers (variable # of RRs)
	authority (variable # of RRs)
	additional info (variable # of RRs)

Per registrare un nuovo dominio bisogna fornire ad un **DNS registrar** (no, non è un errore di battitura, si chiama proprio così) il nome e gli indirizzi IP degli authoritative name server primario e secondario. Il registrar inserisce due RRs nel TLD server, un record type NS e un RR type A per associare il dominio all'autoritative DNS e quest'ultimo al suo indirizzo IP.

all'interno dei parametri dell'autoritative server

1 es. (example.com, dns1.example.com, NS) → type "NS"
 2 (dns1.example.com, 212.212.212.1, A) → type "A"

mail.pippo.it => it
 pippo.it => mail.pippo.it
 mail.pippo.it => pippo.it
 pippo.it => dominio.pippo.it
 dominio.pippo.it => mail.pippo.it

A questo punto è possibile creare nell'autoritative DNS server del nostro dominio tutti gli RR di tipo A per associare ogni nome ad un indirizzo IP, fare alias o mail server. Nell'ambito DNS un attacco **DDoS** (Distributed Denial of Service) è il più pericoloso. Attaccare un root server non è molto efficace perché applicano filtraggio del traffico. Per questo si bombardano i TLD server.

Altri attacchi possibili sono:

- Man in the middle: si intercettano le query
- DNS poisoning: si immettono delle informazioni sbagliate nella cache di un nameserver (ad esempio attraverso un DNS server maligno), e poi questa "cache" verrà copiata da tutti gli altri DNS server, diffondendo il "veleno".
- Exploit DNS for DDoS: essenzialmente metto dietro a un "nome comune" un IP address di uno sfortunato che si penderà tutte le richieste.

P2P "file", myIP, numero Parte

Il protocollo P2P, come abbiamo già visto, permette a un host di comportarsi sia da client che da server per lo scambio di dati (es. La copia di un file). La scelta principale fra l'utilizzo di una architettura basata su P2P rispetto a una client/server sta nel tempo per distribuire un file.

- Nella architettura client/server, se un server deve mandare un file ad N client sulla rete, dovrà inviare N copie di quel file. Il tempo per distribuire le N copie è il massimo tra il tempo che impiega il server per trasferire gli N file ed il tempo che impiega il client più lento a scaricare la sua copia. Il primo valore incrementa linearmente in N.

Nel caso di architettura P2P, un server deve immettere almeno una copia del file, dopodichè potenzialmente non è più necessario. Appena un client ha la copia potrebbe diventare server. Tutti I client nel complesso devono scaricare NF bits, dove F è la dimensione di una copia del file. Tuttavia il max upload rate è dato non solo dal server ma anche dalla somma degli upload dei client che hanno acquisito una copia (gli altri ci aiutano, evvai!!!!). Quindi con un approccio P2P il tempo per distribuire le N copie è il massimo tra il tempo che il server impiega per inviare una copia, il tempo che il client più lento impiega a scaricarla e il tempo che impiega la rete a trasmettere le N copie al

*Proprietà della somma
e commutatività \Rightarrow
facilità di calcolo*

$$\begin{array}{r} 1 \\ 1 \\ \hline 0 \end{array}$$

*checksum con Wray e Rowland
(complementari)*

massimo upload rate, che aumenta linearmente in N ma scala meglio visto che potenzialmente aumentano i server che possono fornire il file ad altri. Ad esempio l'architettura BitTorrent funziona che i file vengono divisi in chunks da 256Kb e i peer si scambiano questi chunks.

Vi sono poi dei nodi speciali chiamati **trackers** che registrano i nodi attivi in rete che posseggono la copia di un determinato file.

In questa architettura il collo di bottiglia è la velocità di download del client. Quando un client ottiene un chunks potenzialmente lui può diventare fornitore registrandosi nella rete, ovvero dicendo al trackers il proprio indirizzo e la lista dei chunks del file da uploadare che possiede.

Un problema di questa rete sono i nodi (**selfishly**) che richiedono servizi ma non forniscono servizi nella rete (scaricano ma non fanno upload). Per questo esistono meccanismi che permettono di inviare chunks solo se chi gli richiede ha fornito servizio all'interno della rete, tra cui:

- Metodo dei crediti: ottengo dei crediti ogni volta che faccio upload, e che possono consumare se scarico.
- Metodo di strozzatura: posso scaricare solo se condivido.

$$D_{P2P} \geq \max\{F/u_s, F/d_{min}, NF/(u_s + \sum u_i)\}$$

increases linearly in N ...
... but so does this, as each peer brings service capacity

• Video streaming and content distribution network (CDN)

Dalla crescita a livello di internet dei contenuti video streaming si è arrivati alla conclusione che la scalabilità dei sistemi riguardava sia i protocolli che l'architettura di rete. Vista

l'eterogeneità dei client bisogna anche ottimizzare il modo in cui vengono codificati i file per impiegare meno rete possibile e fornire a tutti i contenuti che richiedono (inoltre, si ottimizza anche in base alle specifiche del client: che senso ha fare streaming di un video a 1080p a un telefono a 720p?). La codifica delle informazioni delle immagini video deve eliminare la ridondanza dell'immagine e di quelle successive, ovvero eliminare l'informazione che può essere codificata implicitamente, così da diminuire il numero di bit da inviare.

Si fa uso di:

- **CBR** (constant bit rate) se il video è codificato sempre con lo stesso bit rate, e si usa **spatial coding** (ovvero, si riduce il numero dei pixel diversi in un'immagine, frame dopo frame)
- **VBR** (variable bit rate) se il flusso dei bit rate è variabile. Si usa un buffer per sostenere la differenza dei bit, e nel buffer andrà a salvare l'immagine prima di mostrarla. In questo modo, non ci sarà un distacco di tempo (almeno che la connessione non sia davvero bassa, nel caso si fa buffering). Fa uso di **temporal coding**, ovvero viene passata unicamente la differenza fra il frame precedente e quello successivo.

Una delle tecnologie streaming più utilizzate oggi è il **DASH** (Dynamic Adaptive Streaming over HTTP) che si basa sul protocollo HTTP per rendere il browser l'interfaccia di fruizione dei contenuti multimediali su internet. Al posto di chiedere pagine html, il browser chiederà delle immagini/frame.

I server dividono i video in **chunks**. Ogni chunks è memorizzato e codificato a diversi rate. Poi c'è un file chiamato **Manifest File** che fornisce una URL diversa per ognuno dei diversi chunks, quindi per ogni frame abbiamo una URL per ogni formato.

Il client periodicamente misura (**benchmark**) la quantità di bit che vengono effettivamente trasmessi dal server al client, così da analizzare il funzionamento della rete e intervenire di conseguenza, adattandosi ai chunks di qualità migliore possibile. Quindi il client in maniera intelligente determina quando richiedere i chunks, a quale encoding rate (da qui deriva "Dynamic" e "Adaptive").

Per realizzare questo servizio utilizzare un unico mega server è impossibile perché sarebbe un unico punto di fallimento, i client potrebbero trovarsi troppo distanti dal server e questo lo sovraccaricherebbe di lavoro. In poche parole non scala. Un'alternativa è generare copie multiple dei video distribuite geograficamente su determinati siti (**CDN**, Content Distribution Networks).

Esistono due filosofie di posizionamento delle CDN:

1. **Enter Deep** consiste nel mettere un gran numero di CDN server nelle reti di accesso, così da avvicinarsi il più possibile agli utenti finali garantendo un alto throughput e una bassa latenza.
2. **Bring Home** consiste nel posizionare grandi cluster in meno punti e fuori (anche se comunque vicino) le reti di accesso. Questi cluster vengono posizionati vicino i POP di Tier-1. In questo modo ci sono minori costi di manutenzione. Si sfrutta la authoritative DNS del fornitore di contenuti per mappare la richiesta verso il CDN da cui scaricheremo il nostro video, e l'authoritative DNS del CDN ci dirà da quale specifico server della CDN scaricarlo.

Come vengono gestite le richieste nelle CDN?

Nelle CDN si usa un approccio OTT (che sta per Over The Top), che indica appunto il fatto che forniscono contenuti e servizi SENZA avere una infrastruttura propria (come nel caso della TV), bensì si trovano “sopra la rete”, nel senso che usano la rete di reti (Internet). In questo modo, vengono risparmiati i costi di gestione della propria infrastruttura.

Come capire da quale CDN Server prendere I chunks del file che vogliamo visualizzare?

Dunque, come sappiamo, un DNS sever locale riceve la richiesta dell’utente, finché non viene inoltrata all’authoritative DNS del dominio. Questo traduce l’URL mettendo, al posto del dominio, **l’indirizzo del CDN server** che lo inoltrerà al DNS server locale.

Il DNS server locale così farà la richiesta all’indirizzo del CDN che ha appena ricevuto, e così si instaurerà una connessione col DNS server.

Capitolo 3 - Transport Layer – Livello 4

Il livello trasporto implementa un servizio di comunicazione logico end-to-end tra i due processi del livello applicazione che stanno comunicando (e quindi, permette anche di mandare i dati al processo corretto). A differenza del livello rete, che si trova più in basso e tratta della comunicazione fra i due processi *su internet*, il livello trasporto tratta in particolare delle regole da implementare per “capiere” la comunicazione.

In questo livello, dal lato mittente si spezzano i messaggi delle app in segmenti che vengono passati alla rete sottostante, mentre dal lato ricevente si riassemblano i **SEGMENTI** per ricostruire i messaggi e inviarli alle app in esecuzione (avviene dunque, la pacchettizzazione e riassemblamento/spacchettizzazione dei dati).

Quando necessitiamo di un protocollo di livello trasporto affidabile, connection oriented, che gestisce il traffico e come i segmenti vengono inviati utilizziamo **TCP** (Transmission Control Protocol). TCP è quindi il protocollo principale del livello trasporto.

UDP (User Datagram Protocol) non implementa nulla, ed è infatti un’estensione del livello rete. Per questo motivo, viene spesso visto come un modo per “*bypassare*” il livello trasporto in sé.

Nonostante ciò, TCP è costruito sopra UDP.

Il livello trasporto NON garantisce il minimo ritardo garantito e throughput garantito (ovvero la quantità dei dati arrivata con successo).

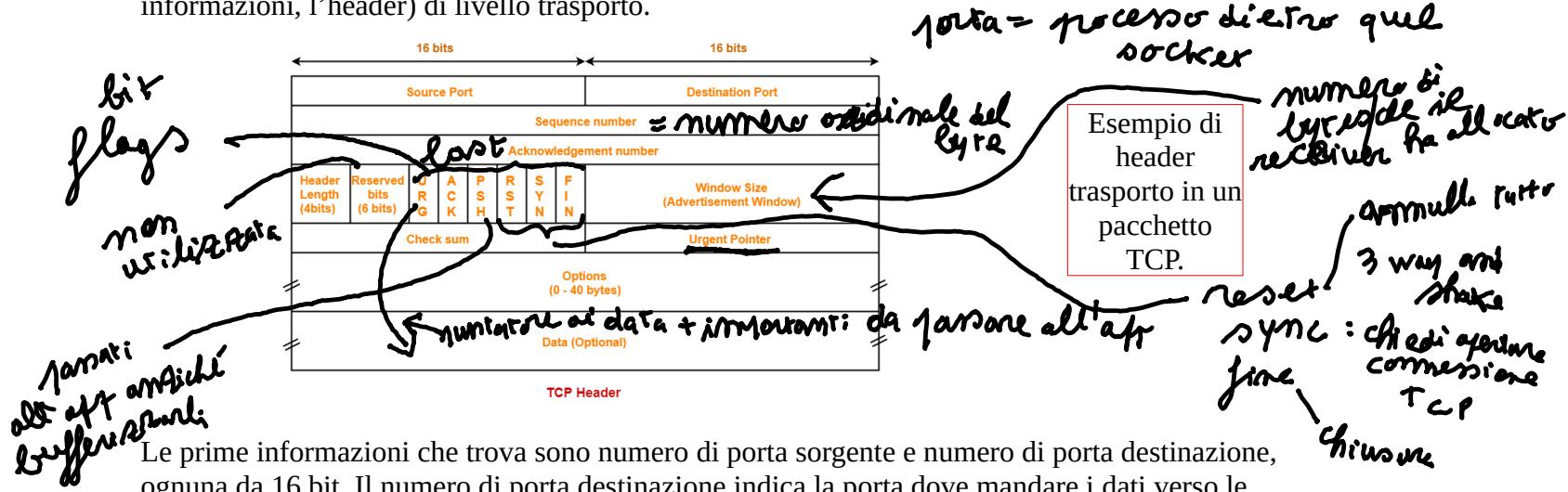
Multiplexing & Demultiplexing

Se abbiamo più applicazioni in esecuzione in un host, sia client che server, a livello trasporto possono esserci più flussi in arrivo/partenza che devono arrivare ognuno in una specifica applicazione, e perciò dovranno essere smistati. Il concetto di “imbuto” (o filtro) verso le applicazioni è ricreato tramite il **multiplexing** (nel mittente, tanti input dall’host e un output verso la rete) e **demultiplexing** (nel ricevente, un input dalla rete e un output verso l’host).

- Il **demultiplexing** prende i dati dalla rete usa le informazioni di header per indirizzare i segmenti verso il giusto socket.

- Il **multiplexing** prende i dati da più applicazioni, e aggiunge le informazioni di header nei **SEGMENTI**, e che saranno poi usate durante il demultiplexing.

Una volta che il pacchetto è arrivato a destinazione il livello trasporto esamina la busta (ovvero le informazioni, l'header) di livello trasporto.



Le prime informazioni che trova sono numero di porta sorgente e numero di porta destinazione, ognuna da 16 bit. Il numero di porta destinazione indica la porta dove mandare i dati verso le applicazioni.

Nel caso di **UDP** quando si riceve un segmento si definisce la porta di destinazione e si mandano i dati nel socket corrispondente a quella porta. Se arrivano dati da diversi IP ma verso la stessa porta **UDP li manda alla stessa applicazione** (Questo perché il socket UDP è identificato solo da IP e port number di "destinazione"). Tra l'altro, nell'UDP colui che manda il messaggio deve sapere in primis a quale porta mandare i dati (es. una well known port number, oppure quella descritta nella documentazione. Nel caso del video streaming, il web server dell'applicazione fornisce le informazioni sul port number al quale il video verrà mandato).

Nel caso di **TCP** bisogna specificare attraverso il socket la sorgente e la destinazione, e poi bisogna mantenere lo stato di trasmissione tra i due estremi, quindi i segmenti in arrivo vanno mantenuti e gestiti. Un socket deve essere definito in modo univoco sia sul client che sul server in modo che si conosca sia la porta sorgente che destinazione. Questo permette a un server di controllare più connessioni TCP verso client diversi. Nel caso di UDP invece quello che arriva ad una porta viene mandato verso la porta da cui è stato ricevuto. Quindi, un normale **socket TCP è definito dalla coppia IP:Porta**.

Il **TCP** fa uso di un **welcoming socket** (tipicamente la porta 80, detta anche well known port number), dove essenzialmente vengono inviate le richieste di apertura di una connessione fra due host. Dopo che si è ricevuta una richiesta di apertura di connessione, il ricevente crea un socket e manda anch'esso una richiesta di apertura di connessione (che viene considerata come un ack della richiesta). Il **welcoming socket** esiste siccome il server può effettuare più operazioni in parallelo (**multithread**).

Al contrario dell'UDP, in caso si abbiano due pacchetti con la stessa porta ma IP diversi, verranno mandati a socket diversi. Questo perché un socket **TCP è identificato da IP e port number di mittente e IP e port number del destinatario**.

UDP un po' più in dettaglio

UDP è un servizio **best effort**, ovvero fa del suo meglio ma non offre garanzie circa il corretto trasferimento dei dati (in particolare, riguardo il ricevimento e l'integrità). In alcuni casi, può essere reso un poco più affidabile aggiungendo l'affidabilità a livello applicazione (**reliable UDP**).

Un segmento UDP è costituito da:

- 16 bit per la **porta sorgente**
- 16 bit per la **porta destinazione**
- 16 bit per dire la **lunghezza** del segmento in bytes (che include l'header).

- il **checksum** che permette di verificare la correttezza delle informazioni che sono state ricevute (nel caso fosse sbagliato provvederà a cestinare il contenuto).

Source Port (2 bytes)	Destination Port (2 bytes)
Length (2 bytes)	Checksum (2 bytes)

UDP Header

Il meccanismo di **checksum** funziona in questo modo: il mittente (sender) tratta il segmento intero in sequenze di 16 bit ed effettua la somma delle colonne e poi il complemento ad uno. Il ricevente effettua il checksum sul segmento ricevuto e se non coincide con quella calcolata dal sender vuol dire che c'è un bit errato. UDP presenta dunque un meccanismo di identificazione degli errori.

● Costruire TCP: come rendere UDP più reliable

livello Rete inaffidabile \Rightarrow livello applicazioni

A livello rete abbiamo una rete inaffidabile, e vogliamo colmare costruendo un protocollo che renda il canale tra sender e receiver affidabile.

Se un'applicazione richiede il trasferimento dei dati affidabile il livello trasporto deve offrire una primitiva che svolga questo lavoro (`rdt_send()`, dove rdt sta per reliable data transfer). Questa primitiva dovrà generare dei pacchetti che spedirà tramite una primitiva offertagli dal livello sottostante (`udt_send()`, che manda informazioni tramite UDP). Quando si riceve il pacchetto il livello trasporto provvederà a ricevere i dati dal livello sottostante e riordinarli con una primitiva apposita (`rdt_recv()`) e poi spedirli all'applicazione che li ha richiesti. Dobbiamo creare un algoritmo che implementi tutto ciò, e per useremo una macchina a stati per rappresentarlo. Per descrivere per bene ciò, useremo un approccio in cui esamineremo i casi aumentandone gradualmente la difficoltà.

supponiamo che
Passo 1: caso in cui abbiamo un canale affidabile. *\Rightarrow no bit errors* *no loss of packets*
Se supponiamo che sotto abbiamo già un canale affidabile (quindi senza perdita di pacchetti e senza errori nei bit), il protocollo non dovrà fare molto lavoro.

- Lato sender**, si aspetta la chiamata da sopra, quando viene chiamata la `rdt_send(data)` il protocollo provvederà a dividere con una primitiva `make_pkt(data)` il messaggio in pacchetti e ad usare `udt_send(packet)` per spedire i pacchetti.
- Lato ricevente**, il protocollo è in attesa all'infinito. Quando viene chiamata la `rdt_rcv(packet)` viene estratto il campo data con `extract(packet, data)` e viene passato all'applicazione il messaggio tramite `deliver_data(data)`.



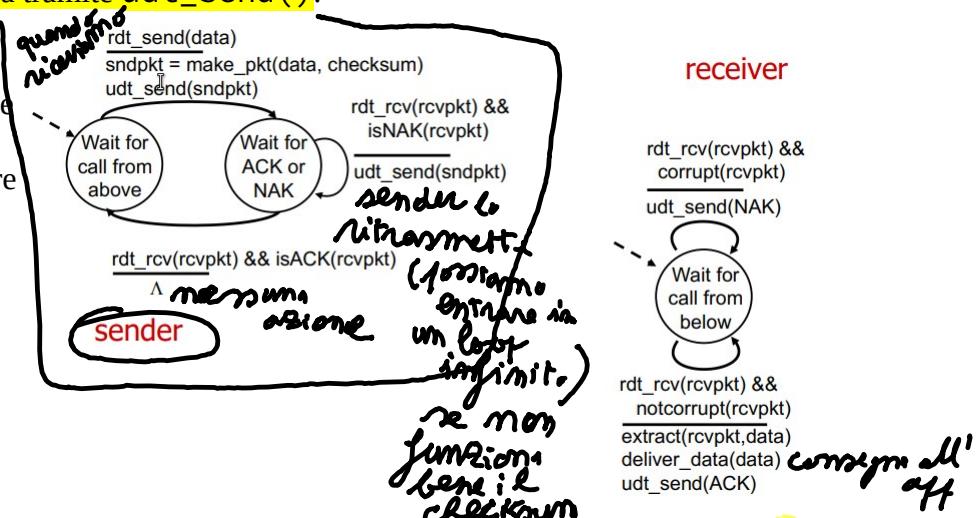
Passo 2: caso in cui abbiamo un canale in cui potrebbero esserci errori sui bit.

Ora per avvicinarci alla situazione reale, immaginiamo che nel canale rete ci possano essere errori sui bit. Il **checksum** rileva gli errori e **vogliamo capire come risolvere gli errori**.

Se non si rilevano errori il receiver invia al sender un messaggio di acknowledgement (ACK) per dire esplicitamente che il pacchetto è arrivato senza errori. In caso contrario si manda un negative acknowledgement (NAK) se il pacchetto arriva con errori. Se il sender riceve un NAK manderà di nuovo il pacchetto.

In questo caso il protocollo rdt_send 2.0 resta sempre in attesa di chiamate dall'alto (nel senso, dalla applicazione) sia nel caso del ricevente che in quello del mittente. Quando crea il pacchetto aggiunge un campo checksum e lo invierà tramite `udt_send()`.

A questo punto però aspetta la ricezione del messaggio di conferma. In questo stato, se dal basso si riceve un pacchetto e questo pacchetto è un NAK richama un'altra volta `udt_send()` per rispedire il pacchetto arrivato con errori. Se riceve un pacchetto e questo è un ACK allora torna nello stato di attesa dall'alto.



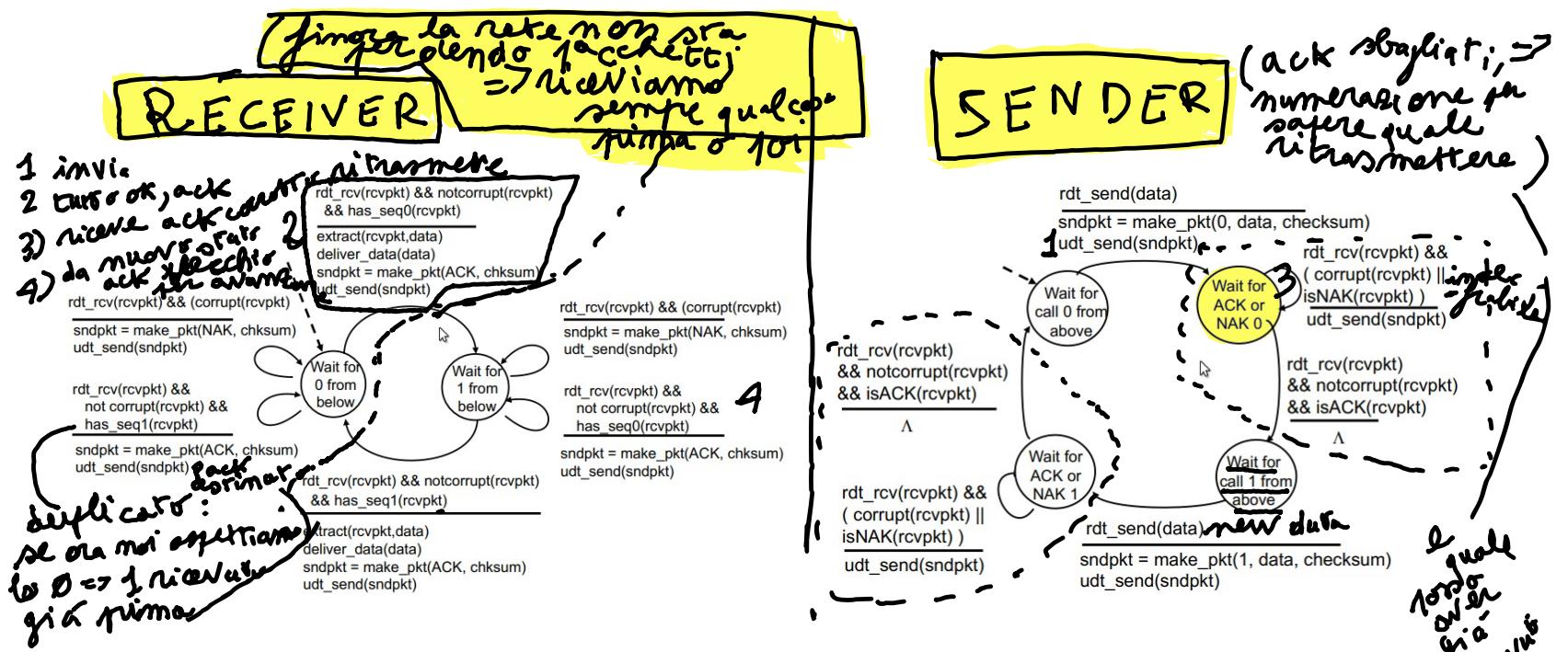
Dunque: dal lato receiver, se riceve dal basso un pacchetto e questo pacchetto è corrotto, allora invia un NAK usando udt_send(), altrimenti se non è

corrotto estrae dal pacchetto i dati, li manda all'applicazione e manda indietro un ACK. Il problema principale è se l'ACK o il NAK arrivano con errori al sender. Nel caso in cui il sender rispedisca il pacchetto quando arriva un ACK/NAK sbagliato, potrebbero generarsi duplicati. Quindi il sender deve aggiungere nei numeri di sequenza ad ogni pacchetto così che il receiver scarti tutti i pacchetti duplicati.

Passo 2.1: gestione della duplicazione dei pacchetti.

In questa nuova versione del protocollo rdt che chiamiamo rdt_send 2.1, inizialmente il protocollo è sempre in attesa di dati dall'alto. In questo caso però differenziamo le chiamate dall'alto in chiamata 0 e chiamata 1. Quindi si numera il primo pacchetto come pacchetto 0 e lo si spedisce come prima. Poi si aspetta la risposta dal receiver del pacchetto 0, se questa è corrotta o è un NAK allora rispedisce il pacchetto, altrimenti se non è corrotta ed è un ACK non si fa niente e si va nello stato “aspetta chiamata 1” e si fa lo stesso ma applicato al caso pacchetto 1. Quindi si cicla all’infinito alternando i pacchetto 0 e 1.

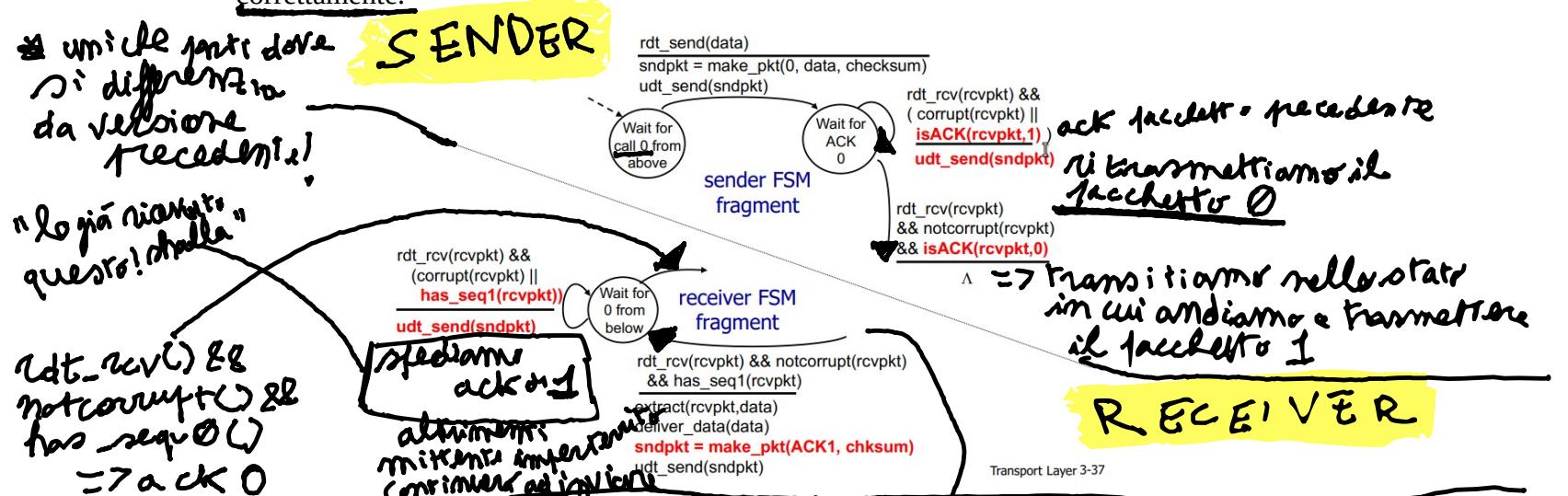
A questo punto lato receiver il protocollo aspetterà i pacchetti 0 e 1. Se è nello stato “aspetta pacchetto 0” e gli arriva un pacchetto non corrotto ma con numero 1, allora questa è la ritrasmissione di un pacchetto precedentemente arrivato, quindi si ritrasmette un **ACK e NON un NAK!** (a cui si associa anche un checksum), siccome probabilmente il nostro ACK non è arrivato correttamente a destinazione. Se arriva un pacchetto corrotto si spedisce esplicitamente un NAK sempre con checksum. Se invece arriva un pacchetto corretto con numero di sequenza 0 si trasmette l'ACK e lo si spedisce con deliver data all'applicazione. A questo punto si passa nello stato “aspetta pacchetto 1” che è la versione duale del caso precedente. Bastano solo due bit perché è sufficiente differenziare i pacchetti tra quello attuale e quello precedente, in quanto il sender è definito nella modalità **stop and wait** (ovvero rispedisce il pacchetto successivo solo dopo che ha ricevuto l'ACK del precedente), di conseguenza può trasmettere o il pacchetto attuale o quello precedente nel caso non sia arrivato un ACK corretto.



Passo 2.2: eliminazione del NAK, non ci serve lol.

Nella versione 2.2 del rdt vogliamo **eliminare il NAK**. Questo è possibile facendo sì che il receiver invii solamente l'**ACK dell'ultimo pacchetto ricevuto correttamente**, quindi si deve esplicitamente dire il numero di sequenza del pacchetto nell'**ACK che inviamo**. In questo caso sono possibili duplicati del ACK, che lato sender verranno interpretati come NAK. Quindi:

- Lato sender, si aggiorna il controllo dicendo che se siamo nello stato 0 e ci arriva l'**ACK** identificato con numero 1, allora vuol dire che l'ultimo pacchetto corretto arrivato era il pacchetto 1 e quindi dobbiamo ritrasmettere.
- Lato receiver, se ci arriva un pacchetto corrotto oppure un pacchetto di una sequenza errata, allora basterà inviare l'**ACK** con numero di sequenza dell'ultimo pacchetto arrivato correttamente.



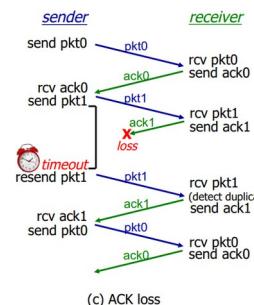
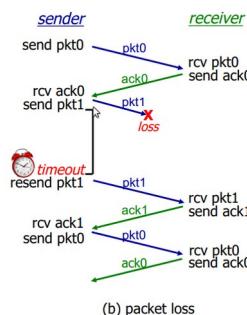
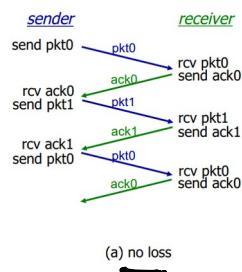
Passo 3: canale con possibile packet-loss.

Se ora il canale può avere **può perdere anche i pacchetti**, allora progettiamo il protocollo rdt 3.0. Teniamo a mente che in questo caso **possiamo perdere sia i dati che l'ACK**. In questo caso, bisogna stabilire un intervallo di attesa temporale, ovvero un **timeout**, per decidere se rispedire un pacchetto che è andato perduto. Dobbiamo cercare di tarare il timeout in modo che, se un pacchetto impiega molto tempo per arrivare ma non è andato perduto, questo non sia ritrasmesso. Quindi in questo caso, se si riceve un pacchetto (o ACK) sbagliato (o un ACK fuori sequenza) non bisognerà fare nulla, perché non appena scatterà il timeout il pacchetto verrà rinviato e sarà fatto ripartire il tempo di attesa. Se arriva l'**ACK** corretto allora non faremo altro che passare allo stato successivo, e se dovessero arrivare pacchetti mentre stiamo aspettando un ACK, allora non faremo nulla perché si riferiscono allo stato precedente. Questo caso particolare potrebbe accadere, per esempio, perché si

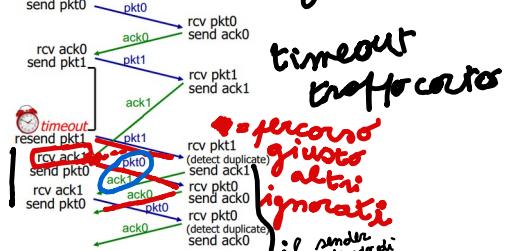
il dest non manca due volte l'ACK
è già passato allo stato dove aspetta il pacchetto successivo

pepi setta un tempo di timeout che è troppo breve, e dunque potrebbero esserci pacchetti "pre-ACK" nella rete, ovvero pacchetti inviati prima di ricevere l'ACK (anche se la ricezione era senza intoppi).

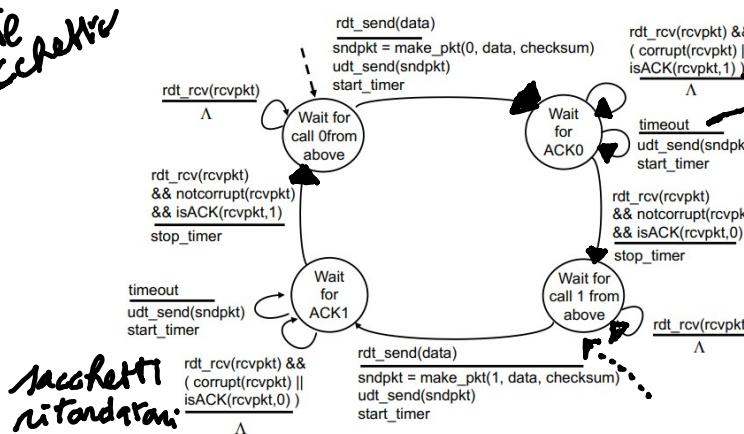
Alcuni esempi di casistiche:



(esempio di mal gestione)



bisogna mandare l'ACK tutte le volte che si riceve un pacchetto



timeout troppo corto
= percorso giusto altri ignorati
il sender non si accorgere che l'onda sta andando
per capire se c'è o no una perdita di un pacchetto => Timer

Il problema del protocollo rdt 3.0 è che, anche se rende la rete sicura, utilizza una frazione piccolissima delle sue potenzialità.

Facciamo un esempio per chiarire: se definiamo **l'utilizzo del mittente (o del canale)** come la frazione di tempo in cui il mittente è stato effettivamente occupato nell'invio di bit sul canale, se il RTT è molto più grande del ritardo di trasmissione L/R, rischieremmo di sprecare troppo tempo a causa dei protocolli di rete (la quantità L/R diviso (RTT + L/R) è l'effettivo utilizzo della rete).

$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027 \quad 8000 \text{ bit per secondo per ricevere l'ACK completamente}$$

Questo deriva dal fatto che il nostro protocollo fa uso di **stop&wait** (stop and wait), ovvero i pacchetti vengono trasmessi uno alla volta, e solo dopo che quello precedente sia stato ack-ato. La rete così fatta trasporta solo un pacchetto alla volta end-to-end. Quindi dobbiamo utilizzare dei protocolli che permettono di trasmettere più dati in parallelo (**pipelining**).

Pipelining (timer dedicato per ogni pacchetto)

Il **pipelining** consiste nel fare in modo che il mittente possa inviare più pacchetti non ancora ackati. Le due forme di pipelining implementabili sono **go-Back-N** e **selective repeat**, che si differenziano per la possezione di un buffer e per il modo con cui si gestisce la ritrasmissione dei pacchetti.

Attraverso il pipelining, possiamo ottimizzare l'utilizzo delle reti: se spediamo 3 pacchetti alla volta ad esempio, l'utilizzo effettivo della rete lato sender diventa $3L/R$ diviso $RTT + L/R$ che è tre volte maggiore del precedente!

- Il protocollo **go-Back-N** permette di inviare N pacchetti alla volta prima di ricevere l'acknowledgment. Il ricevente non invia un ACK alla volta ma invia un **ACK cumulativo**. Il sender **setta il timer** solo per il pacchetto più vecchio che non ha ancora ricevuto l'ACK. Quando questo timer scade vengono rinvolti tutti i pacchetti ancora unacked. Questo è dovuto al fatto che **go-back-N non usa buffer**, e quindi non può bufferizzare i pacchetti

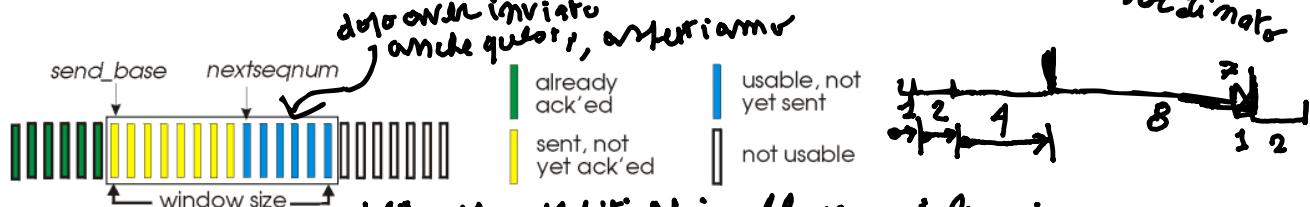
successivi (che tra l'altro richiederebbe una richiesta grossa quantità di memoria da parte del receiver se fatto per ogni socket).

- Nel protocollo **Selective Repeat** possono essere mandati N pacchetti la volta ma vengono mandati un ACK per ogni singolo pacchetto e quindi va mantenuto un timer per ogni pacchetto che non è arrivato. In questo caso **serve un buffer** per mantenere tutti i pacchetti che sono arrivati prima del primo pacchetto della sequenza.

avanza a blocco

Go-back-N *per evitare di saturare code router intermedi
(nel frattempo le code si dilatano)*

Go-Back-N funziona in questo modo: nell'header del pacchetto si aggiunge un **numero di sequenza**. Abbiamo una **finestra scorrevole (sliding window)** di dimensione N che indica il massimo numero di pacchetti di cui non abbiamo ancora ricevuto l'ACK che possiamo inviare in parallelo.



si prende solo se si fa ricevere l'ACK del più vecchio, altrimenti se si riconosce un timeout ritrasmette

Di questi pacchetti di alcuni abbiamo già ricevuto l'ACK e quindi sono già stati trasmessi (quelli verdi). Poi abbiamo alcuni pacchetti, al più N che sono stati trasmessi ma di cui non abbiamo ancora ricevuto l'ACK (quelli gialli). Potenzialmente nella finestra scorrevole ci potrebbero essere degli spazi vuoti, ovvero ci potrebbero essere dei pacchetti che possono essere potenzialmente inviabili, ma che non lo sono siccome l'applicazione non li ha inviati (quelli azzurri)! I pacchetti oltre la finestra scorrevole non possono essere inviati (quelli bianchi).

Se arriva un ACK(n), questo significa un **ACK cumulativo che include implicitamente l'ACK di tutti i pacchetti fino ad n**.

Come abbiamo già detto, si mantiene il timer per il primo pacchetto nella finestra scorrevole. Se non arriva l'ACK di conferma per il pacchetto più vecchio e finisce il timer, **tutti i pacchetti dal primo fino all'ultimo nella finestra scorrevole vengono ritrasmessi**.

Con **send_base** indichiamo il primo elemento inviato di cui non abbiamo ancora ricevuto l'ACK, con **nextseqnum** indichiamo il primo elemento dello spazio ancora disponibile nella sliding window.

Inizialmente il protocollo è nello stato wait.

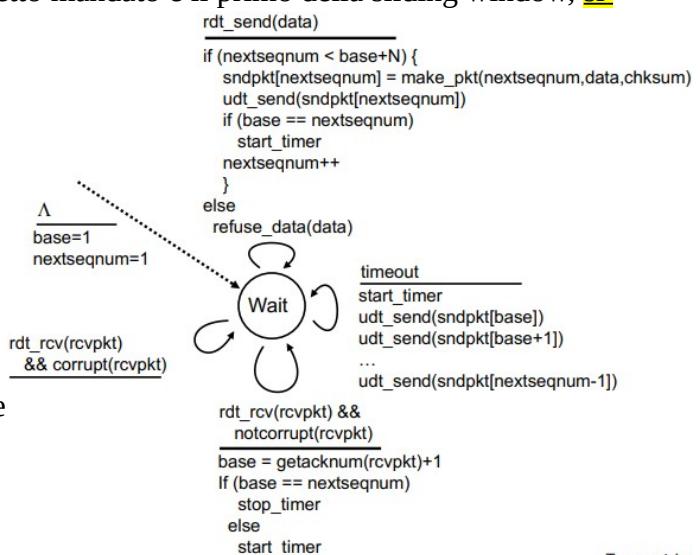
Quando dall'applicazione viene chiamata **rdt_send**, lui controllerà se c'è spazio nella sliding window, ovvero se il valore nextseqnum è minore di base + N. In tal caso, il protocollo creerà un pacchetto associandogli il numero di sequenza dettato da **nextseqnum**.

Se **base==nextseqnum**, ovvero se il pacchetto mandato è il primo della sliding window, **si inizierà il timer, ed infine si aggiornerà il nextseqnum**.

Se la **sliding window è piena**, si rifiutano momentaneamente i dati in ingresso provenienti dalla applicazione.

Se **scade il timeout** che abbiamo assegnato al primo pacchetto, tutti i pacchetti da base fino a nextseqnum-1 vengono rinviati.

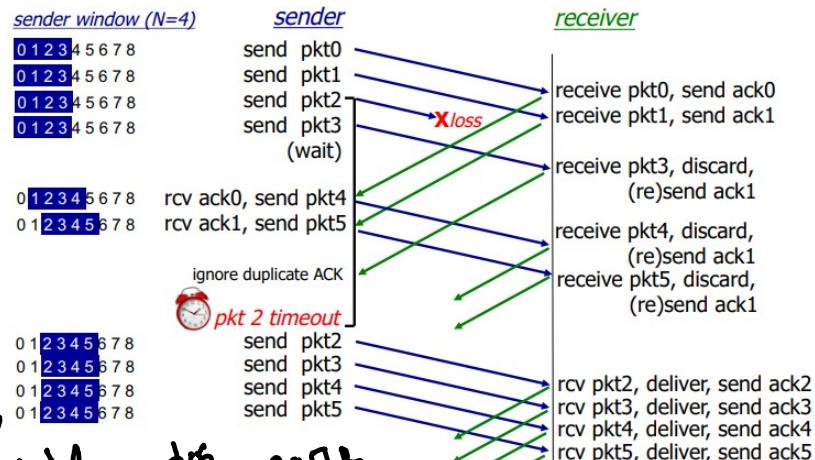
Quando si riceve un ACK corretto, si estrae il numero di ACK e si fa scorrere la finestra incrementando la **base**, ovvero ponendo come primo elemento della sliding window il pacchetto con numero di sequenza **extractACK+1** (ovvero il numero di seq. dell'ACK incrementato di 1). Se in seguito a



quest'operazione la sliding window risulta vuota (ovvero base = nextseqnum) si stoppa il timer, ovvero l'operazione è quindi finita, altrimenti si fa ripartire da capo. Nel caso di ACK corrotti non si fa nulla e si aspetta il termine del timer per ritrasmettere.

Dal **lato receiver**, invece, durante il three way handshake viene impostato il valore `expectedseqnum`, ovvero il numero di sequenza atteso come primo pacchetto. A questo punto, quando si riceve un pacchetto con numero di sequenza identico a quello aspettato, corretto senza errori, allora lo si spedisce all'applicazione (non c'è buffering) e si manda l'ACK indietro, per poi incrementare l'`expectedseqnum`.

In caso contrario si manda indietro sempre l'ACK dell'ultimo pacchetto correttamente ricevuto. In questo modo il mittente capisce che il problema è che il primo pacchetto della finestra scorrevole non è arrivato e quindi bisogna rimandare tutta la finestra scorrevole (gli ACK arrivano in "ordine").



*Non ha uno ad una,
solo quello ritrasmesso,
ni serve
immodificare
(per reti labili)*

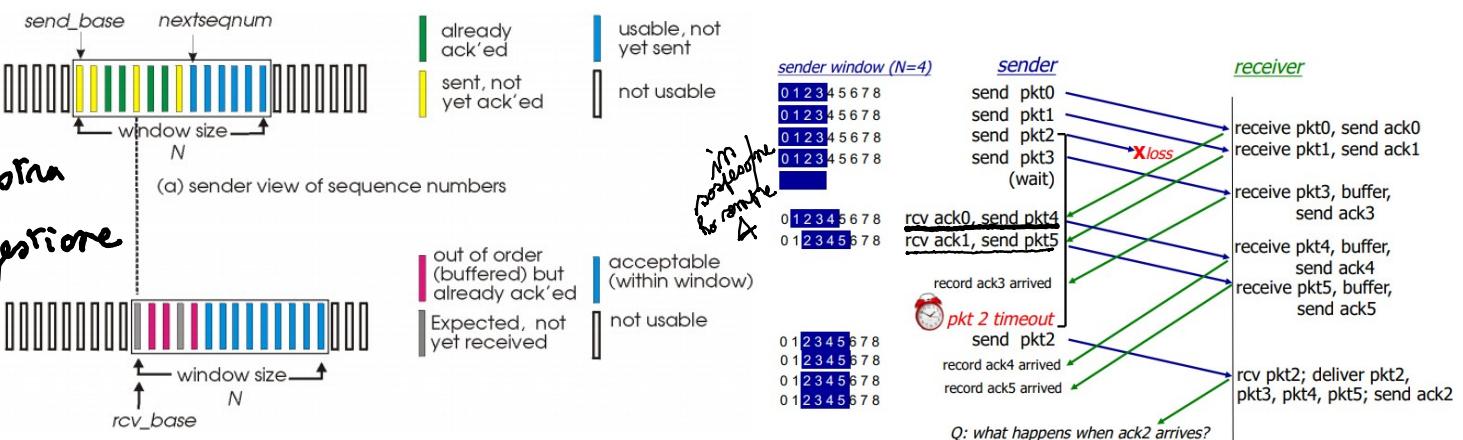
Selective Repeat

Nel caso di **selective repeat** si inviano gli ACK per ogni pacchetto giunto a destinazione.

Quindi il sender dovrà impostare un timer per ogni pacchetto inviato, così da rinviare solo quelli di cui non si riceve l'ACK. Il receiver invece avrà di un buffer per memorizzare i pacchetti arrivati fuori ordine, mentre quelli che arrivano in ordine possono essere mandati verso l'applicazione.

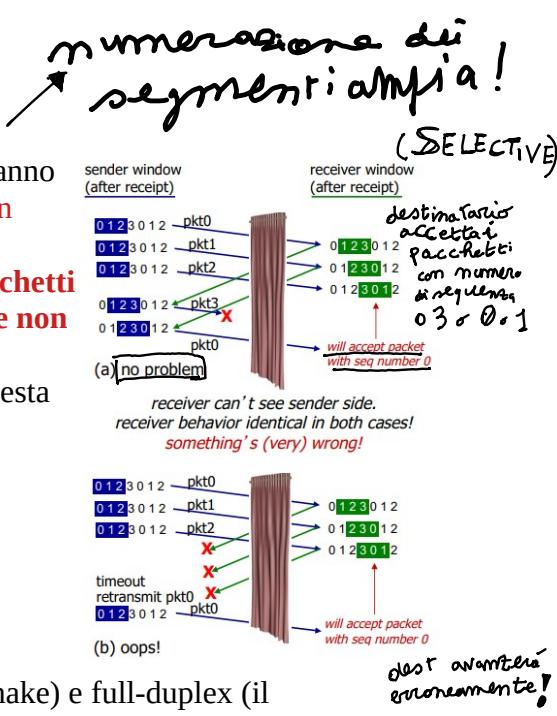
Dunque nella sliding window del mittente potremo avere dei pacchetti di cui abbiamo ricevuto l'ACK intervallati da pacchetti inviati senza ACK ricevuto. A differenza del GBN, la finestra scorrevole scorre solo se si riceve l'ACK del primo pacchetto unACKed nella finestra scorrevole. Se arriva un pacchetto in ordine lo si manda all'applicazione, altrimenti se è fuori sequenza viene memorizzato in un buffer.

Quando si riceve il pacchetto corrispondente alla posizione `rcv_base`, che indica il primo pacchetto nell'ordine non ancora ricevuto, tutti i pacchetti successivi a `rcv_base` già arrivati (e consecutivi fra loro) vengono spediti all'applicazione e si scorre la finestra fino al primo pacchetto non ancora arrivato. Se arriva un pacchetto precedentemente inviato all'applicazione può significare che il relativo ACK è andato perduto, quindi si rimanda l'ACK del rispettivo pacchetto.



Un problema del selective repeat è che, siccome il sender e il receiver non sanno cosa succede dal lato dell'altro, se i numeri di sequenza vengono impostati in modo scorretto, potrebbe succedere che dal lato receiver scorre la finestra scorrevole, tuttavia gli ACK vengono perduti. Dunque, il sender invierà pacchetti con un numero di sequenza che può essere accettato dal receiver, ma che non corrisponde al reale pacchetto atteso, bensì ad uno già arrivato.

Questa situazione non può essere identificata dal receiver non può sapere questa cosa. Per evitare questo, bisogna usare numeri di sequenza molto maggiori rispetto alla dimensione della sliding window.



TCP in dettaglio

TCP è un protocollo pipelined, connection-oriented (grazie al 3-way-handshake) e full-duplex (il flusso dei dati è bidirezionale). Il TCP inoltre possiede anche un controllo di flusso dei dati, in modo che il mittente non sommerga il destinatario di informazioni. Il controllo del flusso inoltre stabilisce anche a quale valore settare la dimensione della sliding-window. TCP è inoltre un protocollo punto a punto, dunque non permette il multicasting (es. un socket comunica solo con un socket).

Un pacchetto TCP è strutturato in questa maniera ([vai qui per vedere l'immagine](#)):

- 16 bit per **source port** e 16 bit per **destination port**.
- 32 bit utilizzati per il **numero di sequenza**, ovvero il primo byte scambiato nel segmento (ad esempio se un file è di 50000 byte e MSS=1000 [MSS sta per Maximum Segment Size], il primo segmento ha numero 0, il secondo 1000 e così via) [può anche essere visto come il numero del primo byte nel segmento rispetto alla grandezza totale del file]. [Normalmente, vi è un offset random impostato per la grandezza in modo da non confonderli con pacchetti vecchi].
- **Acknowledgment number** che contiene il primo byte atteso dall'altro lato. Ad esempio che A ha ricevuto da B i bytes da 0 a 535 in questo campo si scriverà 536. Significa che i bytes da 0 a 535 sono stati ricevuti e quindi è un ACK cumulativo.
- Altri 16 bit sono usati per indicare la **grandezza dell'header**, alcuni non sono utilizzati e poi ci sono 6 bit di cui il primo indica se il pacchetto contiene **dati urgenti (URG)**, il secondo indica se il pacchetto contiene un **ACK (ACK)**, il terzo è il **push bit (PSH)** e significa che sono dei dati necessari all'applicazione e quindi vanno spediti all'applicazione immediatamente, anche se fuori ordine, poi gli ultimi 3 bit servono per l'handshake:
 - il primo (**RST**) se è ad uno significa che il server non vuole accettare o vuole buttare giù la connessione, il secondo (**SYN**) per inizializzare una nuova connessione o la risposta ad una richiesta (in questo caso va ad 1 anche il bit dell'ACK), il terzo (**FIN**) si setta ad uno per richiedere di finire una connessione che va confermata anche dall'altro lato.
- 16 bit che indica il numero di bytes che il ricevente è disponibile ad accettare, ovvero il buffer residuo (**receive window**).
- 16 bit di **checksum** e altri 16 che indicano nel caso in cui vi siano dati urgenti (URG=1), indica la posizione dei dati importanti in quel pacchetto (urgent data pointer).
- Campo di **options** di dimensione variabile ed infine **i data**.

Il protocollo TCP non presenta un modo predefinito di gestire i pacchetti fuori ordine, quindi la gestione spetta al programmatore che crea l'implementazione del TCP. Potrà infatti decidere di

usare un sistema pipelined attraverso **go-back-N** oppure **selective repeat**, oppure potrà ignorarli direttamente e considerarli come pacchetti persi.

Come settare il timeout in TCP?

TCP, come il nostro protocollo ideato per il reliable UDP, presenta un valore di timeout per il recupero dei pacchetti in ritardo. Ma come settare questo timeout? Questo valore è importante, siccome se è troppo corto ci saranno dei timeout prematuri e delle ritrasmissioni inutili di pacchetti, mentre se è troppo lungo, la reazione alla perdita dei pacchetti sarà troppo lenta.

In generale, dovremo settare il valore di timeout in modo che sia **più grande del RTT**, tuttavia quest'ultimo è variabile. Quindi si fa un *campionamento del RTT*, che chiameremo **sampleRTT**, (che corrisponde all'ultimo RTT registrato) tenendo però a mente che questo può variare da un test all'altro. Dunque, introduciamo un **estimatedRTT** che essenzialmente fa una media pesata fra il nostro sampleRTT che abbiamo trovato all'istante t (quindi quello più nuovo, del pacchetto inviato più recente) e dell'**estimatedRTT calcolato all'istante precedente**. In questo modo ottengo:

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

dove **alpha** è una costante di controllo della media pesata, spesso settata pari a 0.125, in questo modo il test attuale influenzerà solo per $\frac{1}{8}$ il tempo di RTT stimato.

Il timeout viene impostato maggiore rispetto al tempo all'estimatedRTT appena calcolato.

In particolare, sarà in particolare di un certo margine, che chiamiamo **devRTT** (ovvero, deviazione dell'RTT).

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot | \text{SampleRTT} - \text{EstimatedRTT} |$$

Le sbarrette indicano il valore assoluto.

Dove il devRTT è quello dell'istante precedente, **beta** anche qui è un parametro di controllo, spesso settato a 0.25.

Il valore del timeout finale sarà:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

All'inizio, ovvero quando non si hanno abbastanza dati per stabilire il valore del timeout, un valore pari a 1 è accettabile.

application	application layer protocol	underlying transport protocol
remote terminal access	e-mail	SMTP [RFC 2821]
		Telnet [RFC 854]
	Web	HTTP [RFC 2616]
	file transfer	FTP [RFC 959]
streaming multimedia	HTTP (e.g., YouTube),	TCP or UDP
	RTP [RFC 1889]	
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

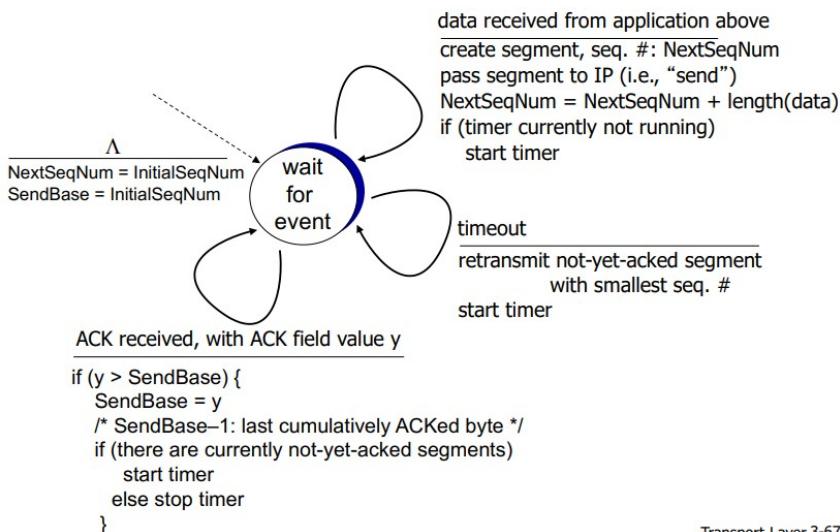
application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video: 10kbps-5Mbps msec	yes, 100's
stored audio/video	loss-tolerant	same as above	
interactive games	loss-tolerant	few kbps up	yes, few secs
text messaging	no loss	elastic	yes, 100's msec yes and no

Meccanismi importanti del TCP – gestione ACK e ritrasmissione.

Il TCP crea un servizio reliable di trasferimento dei dati, costruito sopra un servizio di trasferimento unreliable dato dal protocollo IP (e in generale del network layer, livello rete). Normalmente, la ritrasmissione si ha in due casistiche principali, ovvero in caso di **timeout** del timer di un segmento e in caso di **ack duplicati**.

LATO SENDER

I passi principali che il **mittente** esegue, sono la creazione di un segmento (dai dati provenienti dall'applicazione) con il giusto numero di sequenza (ovvero il numero del primo byte nel segmento rispetto alla grandezza intera del file). Prima di inviare, si inizia un timer (per esempio, possiamo immaginare il timer come il timer per il pacchetto più vecchio non ancora ackato come in GBN). In caso di timeout, si ha una ritrasmissione del segmento e si ricomincia il timer, mentre se ricevo l'ack di quel segmento (quindi con ACKnumber = ultimo_byte_segmento + 1).



Transport Layer 3-67

Il sender inizialmente è in attesa di dati dall'alto. Si inizializzano i parametri NexSeqNumber e SendBase, e questo viene deciso durante il 3-way-handshake. Quando arrivano dati all'applicazione (come abbiamo visto prima) e si aggiornano il seqNum del segmento e il nextSeqNum. Se il timer è fermo, viene fatto partire.

In caso di timeout, si ritrasmettono i pacchetti non ackati con il num di seq. più basso, e il timer ricomincia.

Se si riceve un ACK con valore y nel campo ACK, allora se questo è maggiore di sendBase, aggiorniamo sendBase a y. Se ci sono segmenti non ackati, resetta il timer, altrimenti ferma il timer e congratulati per un buon lavoro svolto.

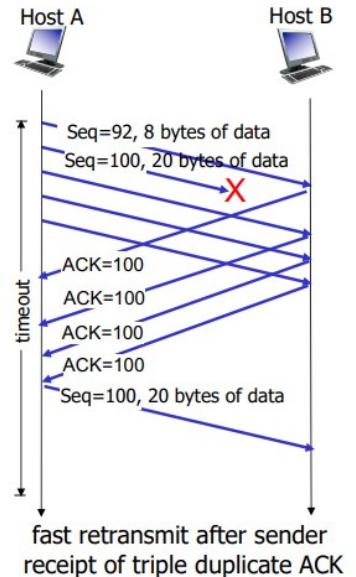
Le politiche di generazione di ACK lato ricevente dipendono da diverse casistiche, che si possono riassumere in questa tabella:

EVENTO NEL RICEVENTE	AZIONE DA SVOLGERE SECONDO TCP
Arrivo in ordine di un segmento, con il giusto numero di sequenza. Inoltre, tutti i dati fino al numero di sequenza che stavamo aspettando sono già stati ACK-ati. (Dunque, mi è arrivata una mole di dati corretta).	Allora mando un ack con un ritardo: aspetto 500 ms per il prossimo segmento. Se non arriva, invia l'ack (In questo modo posso fare un ack comulativo di una intera sequenza).
Arrivo in ordine di un segmento, con il giusto numero di sequenza. Tuttavia, prima di questo è arrivato un altro segmento con un ACK non ancora risolto, in attesa.	Manda immediatamente un ACK comulativo, quindi solo dell'ultimo segmento, in modo che possiamo ack-are entrambi i segmenti in ordine.
Arrivo di un segmento fuori ordine, con un numero di sequenza maggior di quello atteso. Dunque, ho un buco , con dei segmenti mancanti.	Manda immediatamente un ACK duplicato , con numero di ACK pari al numero di sequenza (ovvero primo byte) del segmento atteso.
Arrivo di un segmento che riempie completamente o parzialmente il buco.	Manda immediatamente un ACK, a patto che il segmento riempia il GAP partendo da sinistra. Altrimenti, ack duplicato.

Meccanismi importanti del TCP – FAST RETRANSMITTING

Quando un segmento è stato perso, il timeout dei pacchetti potrebbe generare problemi in quanto rappresenta un altro parametro che aumenta il ritardo in aggiunta agli altri. Fortunatamente, il mittente può capire quando avviene un packet-loss grazie agli **ack duplicati inviati dal ricevitore**. Dunque, se mandiamo una serie di pacchetti ma riceviamo 3 volte un ACK duplicato, possiamo capire che appunto uno dei nostri segmenti che abbiamo inviato è andato perduto (in particolare, il segmento inviato prima di questi altri 3 pacchetti che hanno causato l'invio degli ACK duplicati).

L'arrivo degli ACK duplicati spinge il sender ad inviare subito il pacchetto che si suppone di aver perso, **senza aspettare il timeout e soprattutto senza diminuire il flusso/ritrmo di invio dei dati**, siccome l'arrivo dei 3 ack ci dimostra che la rete non è congestionata. Questo meccanismo, viene detto **fast retransmitting**.



Meccanismi importanti del TCP – Flow Control (controllo di flusso)

Come sappiamo, ogni socket ha allocato un buffer in cui vengono memorizzati i segmenti in arrivo dalla rete. I segmenti in arrivo vengono poi letti e passati all'applicazione e quindi il buffer verrà successivamente sovrascritto. Tuttavia, se l'applicazione richiede i dati troppo lentamente mentre arrivano molto velocemente dalla rete, e non abbiamo un buffer molto grosso potremmo perdere i pacchetti.

Dunque, per risolvere questo problema il TCP implementa un meccanismo di controllo del flusso (**flow control**). Per ogni socket dobbiamo allocare un buffer (che chiamiamo `rcvbuffer`) di dimensione tipicamente di 4096 bytes. Questo buffer al momento t avrà una certa quantità di memoria occupata da dati ancora non letti e uno spazio libero che chiamiamo receive window (rwnd, paramentro che abbiamo visto essere presente anche nell'header TCP). Il ricevente deve comunicare la grandezza di questa finestra al sender così che esso regoli la grandezza della finestra scorrevole (e questo viene comunicato ad ogni invio di pacchetto dal receiver al sender es. ACK, essendo un parametro dell'header) e non la faccia più grande di quanto il buffer possa sopportare, questo garantisce che non vi sia overflow nel buffer del ricevente. In particolare, dovrà limitare il numero di pacchetti “in volo” inviati e non ancora ackati a questa quantità fornita dal receiver.

Connection Establishment in TCP

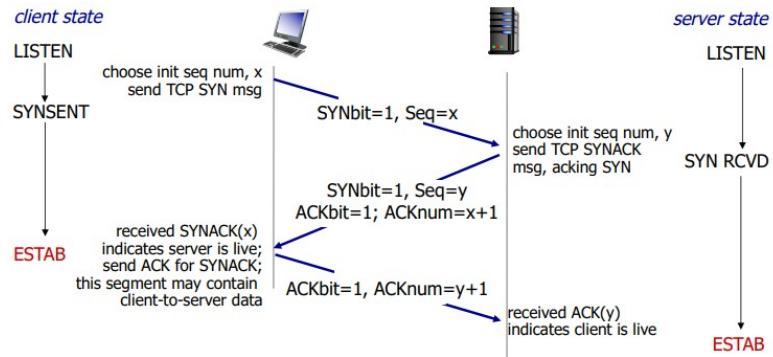
Prima di scambiare dati attraverso TCP, bisogna prima **stabilire una connessione** effettuando un **handshake**. Durante questa fase si determinano informazioni importanti come i numeri di sequenza e le dimensioni dei buffer.

Se usassimo un 2-way-handshake, avremo diversi problemi, ad esempio se effettuiamo la richiesta di connessione ma la conferma arriva troppo tardi dal server, potremmo nel mentre mandare un'altra richiesta inutilmente. Ma se arriva la conferma della prima e nel mentre finiamo la connessione, e al server arriva anche l'altra richiesta, questo potrebbe accettarla ma sarebbe una connessione aperta solo lato server. Ancor peggio se nel mentre inviamo dei dati che vengono accettati ma li ritrasmettiamo perché scade il timeout e nel mentre finiamo la connessione ma arriva l'altra richiesta inviata, questa potrebbe essere accettata e potrebbero persino arrivare i dati che avevamo ritrasmesso due volte.

Per **instaurare una connessione**, il protocollo TCP procede in questo modo:

Prima di tutto, l'applicazione lato client dice al protocollo TCP di voler iniziare una connessione verso un server. Questo fa sì che TCP mandi un segmento contenente **SYN=1** al server con numero di sequenza x ed entra nella fase **SYN_SENT**. In questa fase, si attende un ACK dal server che certifichi che la connessione che la richiesta di connessione è arrivata. Il server infatti, dopo aver ricevuto il segmento (**SYN_RCVD**), per rispondere al client genererà un pacchetto con SYN = 1,

$ACK = x+1$ e $seq = y$. Questo pacchetto è chiamato anche **SYNACK** e quando arriva al client esso va in stato **ESTABLISHED**, in cui la connessione viene dichiarata stabilita. Il client a questo punto in via un ACK, e questo segmento ACK potrebbe anche contenere dei dati applicazione. A questo punto anche il server entra in in stato **ESTABLISHED**.

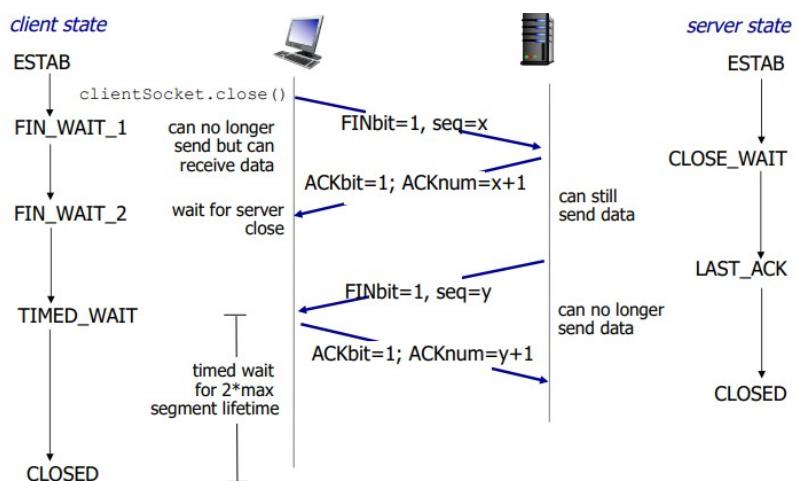


Supponiamo che il client adesso voglia chiudere la connessione. Questo spinge il client ad inviare un segmento con **FIN = 1** ed entra nella fase **FIN_WAIT1**. In questa fase, il client non può più inviare dati, ma può riceverli.

Appena il server ha ricevuto il segmento, entra in **CLOSE_WAIT** e manda al client un ACK. Il client dunque attenderà questo ACK e quando lo riceve entra nella fase **FIN_WAIT2**, in cui attende nuovamente un segmento dal server, ma sta volta semplicemente sarà un pacchetto con **FIN = 1** (indicando così che anche il server è disposto a chiudere la connessione), ed entra in modalità **LAST_ACK**.

Dopo questo momento il client manda un ACK ed entra in **TIMED_WAIT** in modo da poter rimandare eventualmente questo ACK.

Questa fase dura (ovvero **TIMED_WAIT**) dai 30 secondi fino ai 2 minuti generalmente, dopodiché la connessione è chiusa definitivamente. Anche nel caso in cui sia client che server mandino **FIN** simultanei la situazione può essere gestita correttamente. Dopo la chiusura, buffer e socket sono eliminati.



Controllo della congestione

La causa principale della congestione è data da troppi host che provano a mandare un flusso di dati ad un ritmo troppo alto, che provoca un overflow nei buffer dei router intermedi (oppure accodamento se la congestione è meno intensa).

Il **packet-loss** è un sintomo della congestione (e ritardi nel caso dell'accodamento).

Il controllo della congestione è ben diverso dal controllo del flusso! (infatti, il primo interessa i buffer degli host, mentre il secondo interessa la rete in sé).

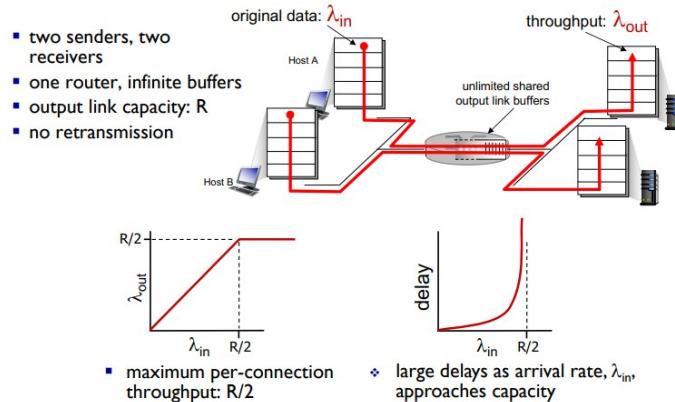
Per capire meglio come funziona il controllo della congestione, vediamo tre scenari in cui si verifica la congestione.

SCENARIO 1 – DUE MITTENTI, DUE RICEVENTI E UN ROUTER CON BUFFER INFINITO

Supponiamo di avere due sender **A e B** che condividono un link ed inviano dati a due receiver **C e D** passando per un unico router con capacità del buffer infinita. La velocità dell'output link è di R e sia A che B producono pacchetti alla velocità di V bit/s.

Il throughput cresce linearmente in funzione di V , tuttavia quando V arriva ad $R/2$, ovvero la velocità massima di connessione condivisa, il throughput non può crescere più siccome è il massimo che l'output link può supportare.

Il problema sta nel fatto, che man mano che V si avvicina ad $R/2$, il ritardo nel router inizia ad aumentare e quando V tende ad $R/2$, il ritardo tende ad infinito.



SCENARIO 2 – DUE MITTENTI, DUE RICEVENTI E UN ROUTER CON BUFFER FINITO

Ipotizziamo che ora il router abbia buffer **finito**.

Sia il parametro V con lo stesso significato di prima, e introduciamo invece V' che è il tasso di invio di **pacchetti originali + pacchetti ritrasmessi**, visto che ora possono andare perduti a causa del buffer finito.

Vogliamo fare in modo che non ci siano ritrasmissioni, ovvero vogliamo trovare un punto in cui inviamo più dati possibili senza congestionare il router. Infatti se ci sono ritrasmissioni la rete viene usata non solo per spedire dati originali ma anche dati ritrasmessi, quindi aumentando V , non è detto che il throughput aumenti linearmente con V' , perché parte dei dati inviati potrebbero essere pacchetti andati precedentemente perduti e che quindi facciano sprecare risorse (questo nel caso in cui $V' > V$). In un caso ideale noi dovremmo sempre avere spazio libero sul buffer dei router, e vorremmo che la somma dei dati inviati dagli host sia inferiore del buffer intermedio. Purtroppo non conosciamo né quanto buffer il router intermedio abbia né quanti pacchetti perde.

Controllo della congestione in TCP (Slow start)

Nel congestion control di TCP, i sender partono da un carico di dati (**transmission rate**) molto basso, dunque il valore della finestra di congestione (**cwnd**, solitamente è corrisponde alla finestra scorrevole, e corrisponde anche alle quantità di dati massimi che possiamo tenere in sospeso) è inizialmente basso (pari a 1 MSS). All'aumentare del tempo, il valore di **cwnd** sarà regolato in questo modo:

- Aumentiamo il valore di cwnd di 1 MSS (ovvero di un segmento) ad ogni RTT, e quindi ad ogni ACK ricevuto per pacchetto (**additive increase**)
- Se invece perdiamo un pacchetto, si divide il cwnd a metà (**multiplicative decrease**)

Grazie a questo crollo della velocità di invio permettiamo al router di liberare il buffer che si era riempito.

Dunque, il controllo della congestione consiste nel mandare un numero di byte pari alla finestra scorrevole, aspettare un RTT e poi mandare ancora più bytes, quindi il sending rate è approssimativamente pari alla dimensione finestra scorrevole in bytes/RTT in secondi.

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

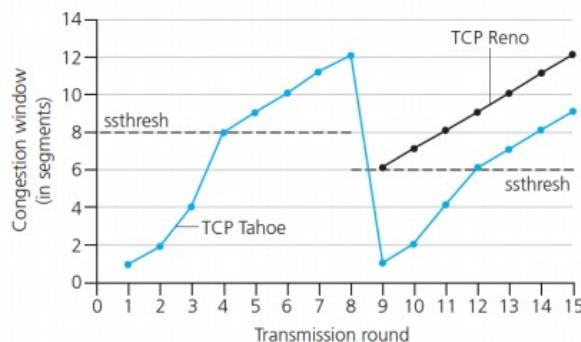
Un'altra tecnica di aumento della cwnd è la tecnica **slow start**, che consiste nel raddoppiare ad ogni RTT la dimensione della finestra.

Possiamo avere più modi di gestire/riconoscere la perdita dei pacchetti e regolare la finestra in base ad essi:

- Potremo avere riconoscere come un packet-loss lo scattare del timeout. In questo caso, il valore della finestra torna ad 1 MSS, poi cresce in modo esponenziale fino a threshold (soglia) e poi aumenta in modo lineare.
- Potremo riconoscere 3 ACKs duplicati come un indicatore di packet-loss, infatti se ricevo gli ACK sicuramente non ho congestione. Dunque, riparto da metà cwnd in modo lineare. (approccio adottato da TCP Reno).
- Potremo usare entrambi i precedenti come indicatori di packet-loss. In questo caso, settiamo la dimensione della finestra ad 1 e la aumentiamo esponenzialmente fino a una soglia, dal quale poi aumenterà in modo lineare. (approccio adottato da TCP Tahoe)

Il valore della soglia, in tutti e 3 i casi (anche in TCP Reno, dove la soglia è appunto metà cwnd) è dettato da una variabile chiamata **ssthresh** (slow start threshold). Il valore di sshtresh è di metà della cwnd al momento del packet loss.

Inizialmente, l'aumentare della quantità dei dati è esponenziale (a causa del slow start), ma quando si raggiunge questa soglia, si ha un incremento di 1 MSS alla volta (quindi lineare). Questa fase lineare è detta **congestion avoidance**.



TCP Fairness

TCP presenta un problema di **fairness**, ovvero dati due host che condividono risorse (in questo caso, i buffer dei router) essi iniziano nella fase slow start a mandare pacchetti sempre più numerosi in modo esponenziale. Poi il primo host che perde un pacchetto (e dunque ricomincia in slow start) prova a riaumentare il ritmo, tuttavia è bloccato in questa impresa dall'altro host che non ha mai fermato il suo ritmo di invio e che anzi continua ad aumentarlo, consumando la maggior parte delle risorse.

Teniamo a mente che il ritmo di invio, fuori dal slow-start, è controllato dall'applicazione.

Un altro problema è che pacchetti che utilizzano UDP **ignorano la congestione della rete**, quindi l'applicazione non fa nulla per rallentare i pacchetti UDP, mentre quelli TCP devono gestire questa problematica.

Una delle funzioni implementate a livello rete per gestire il congestionamento è l'**Explicit Congestion Notification (ECN)**, che prevede che nei pacchetti a livello rete sia presente un campo detto ToS (Type Of Service, è un campo nell'header IP) che, se marcato da un router, notifica ai router vicini (in particolare, quello successivo), che quel router è congestionato. **Tuttavia, queste informazioni vengono inserite dai router PER I ROUTER, quindi le sorgenti non sanno nulla della congestione e possono comunque inviare ad un ritmo molto elevato.** In poche parole, non si fa altro che spostare il punto di congestione dal router congestionato a quello prima di esso, che invierà a sua volta dati a un ritmo inferiore. Per questo motivo, non è molto usato. [In realtà, se il ricevente vede che il campo ToS dell'header del livello rete è stato marcato, invia al sender un ACK con il campo ECE (che sta per ECN-Echo) del header di livello trasporto per avvisare il sender del congestionamento.]

Alcune considerazioni su UDP

Solitamente, sappiamo che dobbiamo usare UDP quando il nostro servizio può tollerare packet-loss, in quanto UDP presenta un checksum, quindi identifica se ci sono errori, ma non fa nulla al riguardo (o riguardo al packet Loss per esempio).

Capitolo 4 – Network Layer – livello 3 – Data Plane

Introduzione all'IP (Internet Protocol)

A livello tre, la rete diventa una rete di reti gestita gerarchicamente.

Il protocollo importante di questo livello è il **protocollo IP**, che fornisce un nuovo metodo di indirizzamento alternativo all'indirizzo MAC. Grazie all'indirizzo IP, si può astrarre la struttura di una rete locale, e per l'instradamento dei pacchetti basta un rappresentante, ovvero un **router**, che ne fa le veci dall'esterno.

L'indirizzo IP è formato da una quadrupla di 8 bit, dunque ciascun elemento della quadrupla può assumere valori da 0 a 255.

Mentre una scheda di rete può assumere un solo indirizzo IP alla volta, un indirizzo IP può essere usato da più schede di rete alla volta (vedremo più avanti perché). Se ad un indirizzo MAC viene associato sempre lo stesso IP si chiama **indirizzo IP statico**, altrimenti **dinamico**.

Un indirizzo IP può essere diviso in due componenti logiche, una che individua il **numero di rete** (network number), una il numero di **host nella rete** (host number), questo grazie alla **maschera di rete**.

Attraverso l'indirizzo IP possiamo distinguere 3 classi di reti:

- **Le reti di classe A** sono al massimo 126 e ognuna può contenere fino a oltre 16 milioni di host. Per le reti di classe A, **il bit più a sinistra di tutti comincia sempre è sempre 0**, e per questo, il **network number** può assumere i valori da 1 a 126. I tre byte rimanenti possono assumere oltre 16 milioni di combinazioni, ognuna associabile a un host della rete (in particolare, va da 0.0.1 a 255.255.255, quest'ultimo indica l'indirizzo di broadcast).
- **Le reti di classe B** sono al massimo 16.382 e ognuna può contenere fino a oltre 64.000 host. Per le reti di classe B, il network number è dato dai **due byte di indirizzo più significativi** (a sinistra), che hanno sempre **i primi due bit a sinistra uguali ai bit 10** (uno acceso e uno spento). I network number di classe B possono assumere i valori da 128.0. a 191.255. I due byte rimanenti (host number) possono assumere oltre 64.000 combinazioni, ognuna associabile a un host della rete (in particolare, va da 0.1 a 255.255, dove anche qui quest'ultimo indica l'indirizzo di broadcast).
- **Le reti di classe C** sono oltre 2 milioni, e ognuna può contenere fino a 254 host. Per le reti di classe C, i tre byte di indirizzo più significativi (a sinistra) rappresentano il network number, e **hanno sempre i primi tre bit uguali ai bit 110**. I network number di classe C possono assumere i valori da 192.0.0 a 223.255.255. **Il byte rimanente (host number) può assumere 254 combinazioni utili**, su 256 possibili, ognuna associabile a un host della rete. (*Il prof identifica come reti di classe C anche gli IP con numero superiore a 223 fino 255*).

Nei 3 casi visti, non posso assegnare come host number i numeri 0 o 1 (es. 126.0.0.0 e 126.0.0.1, 192.255.255.0 e 192.255.255.1) in quanto rappresentano l'indirizzo della rete e l'indirizzo del router rispettivamente. A livello MAC, il broadcast è dato dall'indirizzo fisico da F-F-F-F-F-F, mentre nell'indirizzo IP l'indirizzo di broadcast è dato dall'host "massimo", ovvero quello con tutti i bit dell'host number settati ad 1 (es. 192.0.0.255, oppure 128.0.255.255). Anche questo, non è un numero assegnabile agli host.

Per istruire ogni router subordinato sulla dimensione e sull'interpretazione degli indirizzi IP da amministrare, **ogni router subordinato deve essere fornito di una maschera di rete (netmask)**. Questa maschera di rete può essere rappresentata o con

IP/[bit_dedicated_to_the_network_number] (vedi [CIDR](#)) oppure indicando con una quadrupla bit dedicati alla maschera (es. IP/24 è 255.255.255.0).

Per ogni dispositivo di rete o calcolatore connesso a una rete IP (es. Internet), la maschera di rete, l'indirizzo del default router e l'indirizzo IP, **costituiscono i tre parametri fondamentali di configurazione del protocollo IP**, e devono essere forniti, al momento della connessione, al livello IP.

Per stabilire se il router deve mandare il pacchetto all'interno della rete locale oppure all'esterno, **questo esegue una operazione di AND** tra l'indirizzo del mittente e del destinatario e in più la **netmask del router**, e controlla se il risultato corrisponde allo stesso numero di rete.

La trasmissione **su un segmento di rete locale** avviene mediante frame di livello MAC/LLC, indirizzati mediante gli **indirizzi MAC** dei dispositivi, **e non attraverso gli indirizzi IP!!**.

Un router deve quindi saper gestire **l'associazione tra indirizzo IP di un dispositivo a livello rete e il suo indirizzo MAC a livello MAC/LLC**. (Maggiori info sull'IP, tra cui header IP, [qui](#)).

ARP (Address Resolution Protocol)

Il protocollo **Address Resolution Protocol (ARP)** risponde in modo standard a questa esigenza. In altre parole il protocollo **ARP** è il protocollo **che lega l'indirizzamento a livello MAC con l'indirizzamento a livello IP**. Il router spedisce sui segmenti della rete locale **un frame in broadcast MAC** (cioè ricevuto da tutti i dispositivi) contenente il codice di richiesta ARP (il pacchetto prende il nome di **ARP Request**), e **l'indirizzo IP del destinatario del pacchetto**. Tale frame equivale quindi al rivolgere a tutti i dispositivi la domanda: "quale indirizzo MAC ha il dispositivo corrispondente al seguente indirizzo IP"? Il dispositivo in questione, se esiste, risponde con un frame indirizzato **all'indirizzo MAC del router**, **contenente il codice di risposta ARP (ARP response)**, e **con allegato l'indirizzo MAC richiesto**. Esiste anche una versione analoga del protocollo ARP, detta **Reverse-ARP**, che risponde alla domanda: "quale indirizzo IP corrisponde al dispositivo con questo indirizzo MAC"?. Una volta che ha il router riceve le informazioni richieste (indirizzo MAC o indirizzo IP che sia), il router aggiorna la sua **ARP table** con le informazioni appena ricevute.

Routing e Forwarding, ovvero le funzioni principali del livello di rete.

Le funzioni del livello rete sono il **forwarding** e il **routing**. Come abbiamo già visto:

- Il **forwarding** è la pratica di inoltrare i pacchetti al giusto router, dunque corrisponde all'azione di "usare le tabelle di routing". Può anche essere visto come il processo dei dati di passare per un router.
- Il **routing** è la procedura di creare ed aggiornare le tabelle di routing. Per fare ciò, si usano diversi algoritmi di routing. Può anche essere visto come la pianificazione del percorso che dovranno eseguire i dati.

Control Plane e Data Plane

Il livello di rete è composto da due parti fondamentali che interagiscono fra loro. Queste sono il Control Plane e il Data Plane.

Il **data plane** consiste nell'insieme di funzioni che vengono implementate *localmente* in ogni router. Queste funzioni devono determinare **le destinazioni di ogni pacchetto**, e devono fare in modo che ogni pacchetto che arriva in una porta di input sia inoltrato in una porta di output il più **velocemente possibile**. Dunque, il forwarding è una funzione del data plane. Un router quindi, una volta che riceve un pacchetto, ne estrae le informazioni dall'header, e in base a questo deciderà dove instradarlo (sfruttando le tabelle di routing).

Il **control plane** è una logica del livello rete, quindi non di un singolo router ma dell'intera rete.

Determina **come scrivere le tabelle dei router in modo che siano il più efficienti possibile**. Ci sono due approcci per realizzare le funzioni di control plane, uno **che prevede di implementarle in ogni router**, dotando ogni router di un algoritmo di routing (**algoritmi di routing tradizionali**), un altro

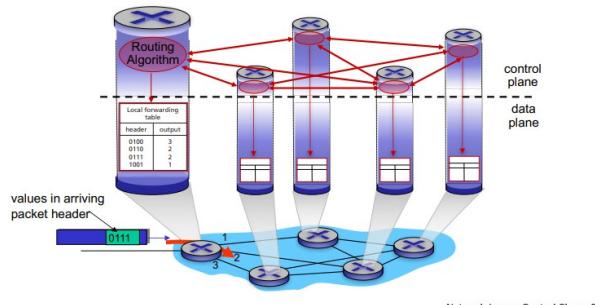
di implementarle a livello software gestito da un server remoto (**SDN, Software Defined Networks**).

Nella versione **decentralizzata**, in cui tutti i router fanno uso di un algoritmo di routing, ogni router conosce la situazione della rete attorno a se, quindi puo usare questi algoritmi **per scrivere le tabelle di inoltro localmente**.

Nella versione **centralizzata**, ovvero nelle SDN, i routing algorithm non vengono eseguiti da ogni router intermedio, ma piuttosto ogni router è dotato di un control agent che comunica con un server remoto che conosce lo stato generale della rete e di conseguenza comunica con i router costantemente per tenere aggiornate le tabelle di inoltro.

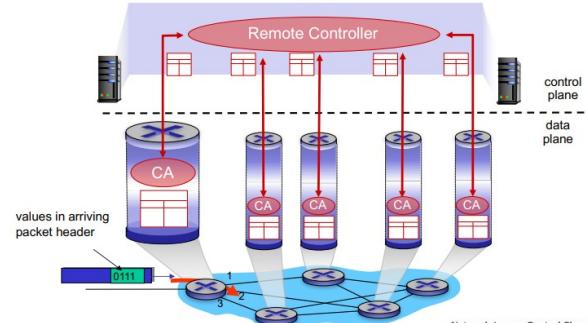
Per-router control plane

Individual routing algorithm components *in each and every router* interact in the control plane



Logically centralized control plane

A distinct (typically remote) controller interacts with local control agents (CAs)



Architettura di un router

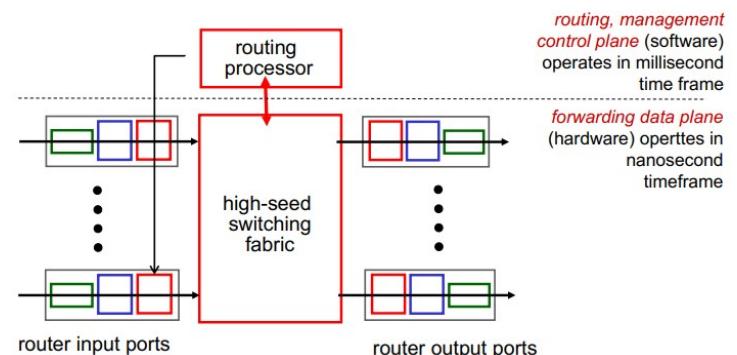
Un router prevede più **input/output port**, una **switching fabric** ad alta velocità, ovvero una centrale di commutazione che connette le porte di input con quelle di output, e un **routing processor**, sul quale girano gli algoritmi di routing (o in generale, nel quale il forwarding viene implementato).

La switching fabric impiega nanosecondi mentre il routing processor opera nell'ordine di millisecondi. Una **porta di input** di un router è composta da questi elementi:

- Un **line terminator**, che riceve i segnali fisici, e li passa al link layer protocol.
- Un **link layer protocol** che legge l'header di livello 2 (ovvero livello MAC) del pacchetto. “Spacchettando” l'header di livello 2, otteniamo il pacchetto vero e proprio. (es. Ethernet) (usando il protocollo MAC necessario). (*Politica di scaricamento*).
- Un **buffer** che riceve i dati dal link layer protocol e conserverà il pacchetto accodandolo finché non verrà instradato.

Dunque, per inoltrare un pacchetto, un router:

1. legge l'header
2. usando le tabelle di forwarding, fa un “look-up” velocissimo per capire dove mandarlo e usando quale porta.
3. Appena fa un “match” con il destinatario (o meglio, nell'intervallo del destinatario) nella tabella di routing, fa l'azione associata al campo “action” relativo a quell'indirizzo (questo sarà più chiaro quando vedremo le tabelle di routing qui)
4. passa al prossimo pacchetto nella coda del buffer.



Vorremmo che queste operazioni siano fatte alla velocità del collegamento, ovvero come se il pacchetto viaggiasse sulla linea mentre è elaborato. Infatti, se arrivano più velocemente di quanto

forwarding table
prefix matching 1 1 0 0 0 0 3 \leftarrow 10010000000000000000000000000000
all'interno del router
possano essere elaborati si accodano e di conseguenza possono andare perduti se c'è un buffer

possano essere elaborati si accodano e di conseguenza possono andare perduti se c'è un buffer overflow.

Il forwarding può essere fatto in due modi:

- **Destination-based forwarding:** si guarda solo l'indirizzo IP del destinatario. È molto veloce, ed è il metodo predefinito di cui fa uso internet.
 - **Generalized forwarding:** si fanno ulteriori operazioni, si controllano se ci sono bit urgenti etc, dunque si tiene conto anche di altre informazioni di header. Questo è molto più lento.

Forwarding Table

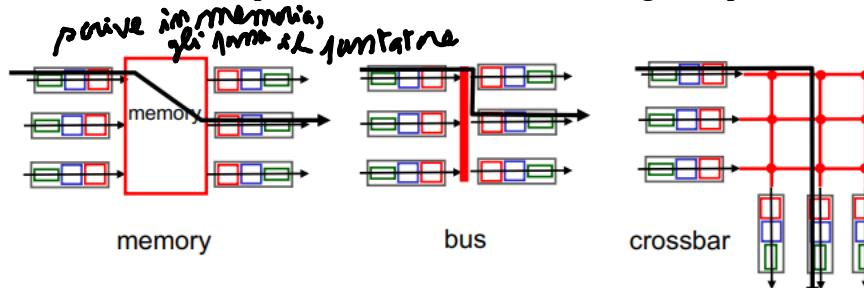
Una tabella di forwarding possibile, ma troppo lenta è costituita da righe che contengono un range di indirizzi IP a cui corrisponde un destination port.

Un’alternativa è non scrivere esplicitamente tutti i bit di una riga (lasciando asterischi), confrontando i prefissi e facendo match con la riga che ha il prefisso più lungo che faccia match (**longest prefix matching**). Dovrò fare una operazione di & con i bit del mio indirizzo e poi ottenere così solamente i bit che fanno match. Se il nostro indirizzo non fa match con nessun altro indirizzo, allora andiamo nel campo “altrimenti”. Esistono memorie che permettono di confrontare in un unico ciclo di clock tutte le righe della tabella (stile TLB di sistemi operativi), e questo tipo di memoria viene chiamata TCAM (Ternary Content Addressable Memories).

Switching Fabric

La **switching fabric** realizza il trasferimento da un input buffer ad un output buffer. Può essere implementata tramite una **memoria**, **un bus** o **a crossbar** con circuiti elettrici.

Lo **switching rate** è il ritmo con il quale i pacchetti possono essere trasferiti da input ad output. Se la switching fabric ha un rate lento può diventare un collo di bottiglia, e provocare ritardi e packet loss.



I router di prima generazione erano computer con funzioni di switch, quindi avevano **una memoria interna (Switching via memory)** in cui venivano messi i pacchetti interni che venivano letti dal processo che gestiva i pacchetti in uscita. Questo sistema ha un collo di bottiglia nella fase di lettura/scrittura.

Switching via bus utilizza un bus condiviso tra le varie porte di input e le varie porte di output. Il problema è la contesa per utilizzare il bus che può essere utilizzato uno alla volta. Inoltre, la velocità della switching fabric sarà dipendente dalla “banda” del bus.

Nello **switching via crossbar** vi è una rete programmabile che viene comandata similmente a dei ponti elevatoi che si alzano o abbassano in base a dove devono andare i bit. Il vantaggio è che permette un grado di parallelismo se non si incrociano i dati in transito. [?] Inoltre permettono di frammentare i datagram in pezzetti chiamati celle che quindi possono essere trasferiti a gruppi anche contemporaneamente [/?].

Un problema che si può verificare all'interno della coda di input (nel buffer si intende) è il cosiddetto **head-of-line blocking**: in poche parole, un pacchetto che avrebbe la porta di uscita "libera" non esce dal router siccome è bloccato da un altro pacchetto che si trova in coda davanti ad esso e che non viene inviato. Inoltre, se due pacchetti all'inizio di due code di ingresso sono destinati verso la stessa porta di uscita se la contendono, siccome solo uno dei due può essere trasferito.

(line terminata = segnale elettrico a cavo)

Architettura delle porte di Output

Poco prima, 2 o 3 paragrafi fa, abbiamo trattato delle porte di Input. È ora il turno delle **porte di output**.

Una porta di output ha una architettura **speculare** a una porta di input, infatti c'è sempre un buffer per l'accodamento dei pacchetti, un link layer protocol e un terminatore di linea (questa volta che "invia" i segnali al posto che riceverli). Anche il buffer delle code di output può causare **packet-loss**, in particolare se la switching fabric commuta i pacchetti più velocemente del ritmo di invio (ovvero, di trasmissione sul mezzo), allora appunto li dovrò conservare in questi buffer, che se non sono abbastanza capienti possono provocare **congestione** e conseguentemente, packet-loss.

Le domande che sorgono spontanee sono quindi:

- Come calcolare la quantità di **buffer** da usare? Per calcolare il buffer, ci sono due regole che possiamo applicare: possiamo usare la formula $BUFFER = RTT \text{ "tipico"} (solitamente 250 ms) per C (ovvero la capacità del link)$. Un nuovo metodo è invece quello di usare la formula:

$$\frac{RTT \cdot C}{N}$$

Numero di "flussi" del router.

buffering necessario
per gestire traffico
di rete

- Come decidere in che modo/ordine inviare e ricevere i pacchetti (**scheduling**)? Siccome la risposta a questa domanda è abbastanza lunga, rispondiamo qua sotto:

Tra le politiche di scheduling dei pacchetti, la più famosa è la **FIFO**, che essenzialmente consiste nell'ordinare i pacchetti attraverso una coda, rimuovendo gradualmente i pacchetti in testa ed inserendo nella tail della coda i pacchetti in arrivo. Se il buffer è pieno, possiamo decidere di "droppare" un pacchetto in base a 3 regole:

- **tail drop**: il pacchetto in arrivo e che trova il buffer pieno viene "droppato".
- **priority**: droppa il pacchetto con priorità più bassa,
- **random**: rimuove un pacchetto scelto casualmente.

Un altro metodo di scheduling dei pacchetti è attraverso il **priority scheduling**, in cui il primo pacchetto ad essere inviato è il pacchetto con priorità maggiore. Per implementare questo metodo di scheduling dobbiamo fare uso di due code per ogni output, una per i pacchetti a priorità alta, e un'altra per quelli a priorità bassa (generalizzando, sarebbe una coda per ogni classe di priorità). Dunque, se c'è almeno un pacchetto nella coda di priorità alta si sceglie lui, altrimenti uno dalla coda normale. Il rischio del priority scheduling è che quelli a bassa priorità subiscano grossi ritardi, dunque ci sarebbe starvation nelle code a basse priorità, in caso le code ad alta priorità siano sempre piene (per questo motivo esistono le leggi sulla Net Neutrality).

Un'altra politica è la **Round Robin** (RIP Corso di Sistemi Operativi). Anche qui, abbiamo più classi di pacchetti, e ciclicamente si scansionano le code di ogni classe e si manda, se possibile, un pacchetto di ognuna. Il vantaggio è rispetto la priority scheduling sta nella assenza di starvation dei pacchetti di bassa priorità.

Infine, abbiamo la politica **Weighted Fair Queuing (WFQ)**, basata su Round Robin, dà diverse priorità per ogni coda/classe, quindi magari si può dare il doppio di priorità ad una coda rispetto alle altre e mandare i pacchetti con un rapporto 2:1:1 se abbiamo tre classi (per una data classe, quindi, si mandano più pacchetti alla volta).

Il protocollo IP – Elementi di un header IP

- I primi 4 bit indicano la **versione del protocollo** (se IPv4 o IPv6).
- Il campo successivo la **lunghezza dell'header** in bytes
- Type of service**: campo non molto usato, eventualmente usato per indicare la priorità di un pacchetto (oppure per le ECN).
- Poi 16 bit per la **lunghezza totale**, compreso l'header, del pacchetto in bytes (massimo 2 alla 16 bytes).
- 16 bit per l'**identificatore del pacchetto** (una specie di targa). *Non chi può essere frammentato via*
- poi i **flags** (che se a 1 ci dice che questo pacchetto è un frammento di un pacchetto più grande).
- Il **fragment offset**: numero del frammento del pacchetto spezzettato, in questo modo possiamo capire a che parte del pacchetto completo corrisponde. I pacchetti con lo stesso identifier sono pezzi dello stesso pacchetto. [SE DOVESSI PERDERE UN FRAMMENTO DEL PACCHETTO PIÙ GRANDE, DEVO RIMANDARLO TUTTO!!!].
- Poi 8 bit per il **time to live (TTL)**, che rappresenta il numero massimo di router attraverso cui può passare.
- L'**upper layer** è il byte che indica il protocollo di livello trasporto a cui deve essere passato il pacchetto (TCP o UDP).
- Poi 16 bit di **header checksum** che controlla Solo se l'header è giusto. (Il controllo della correttezza dei dati applicativi viene fatto solo dal checksum del livello trasporto).
- Poi 32 bit **per source IP** e 32 bit **per destination IP**, che onestamente penso sia abbastanza ovvio che cosa siano.
- Infine il campo option (permesso al programmatore di implementare diversi modi di fare forwarding) e poi data.

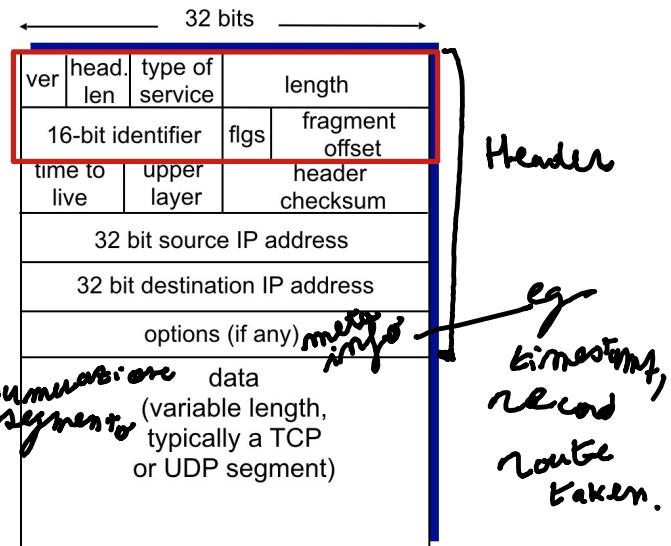
L'overhead totale causato dagli header, escludendo i campi options, sono 20 byte per TCP + 20 byte per IP + overhead del livello applicazione. Dunque il "goodput" (o throughput netto) totale sarà composto dai [dati trasmessi – overhead vari]. Normalmente, in realtà, la grandezza dell'header IP può andare dai 20 ai 60 bytes, questo a causa dell'opzione options, che è variabile.

Frammentazione dei pacchetti

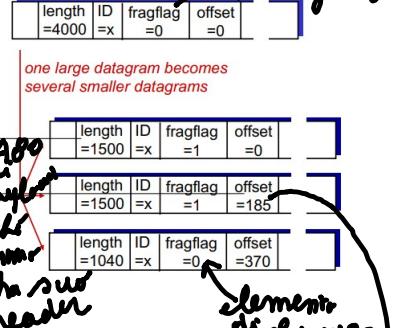
I network links (in realtà sono i protocolli a livello MAC ad avere il MTU) hanno un max transfers size, diverso per ogni tipo di link, di conseguenza se arriva un pacchetto di grandi dimensioni dal livello applicazione, questo va spezzato in frammenti più piccoli, che avanzeranno in modo autonomo e saranno riassemblati nel datagram originale solo a destinazione, che quindi necessita un buffer per questo lavoro.

Per esempio, immaginiamo di spedire un pacchetto da 3980 (+20 header), utilizzando il protocollo Ethernet a livello MAC, dove il **Maximum Transfer Unit (MTU)** è =1500.

Quindi il pacchetto va spezzato in 2 frammenti da 1480 (+20 header) ed uno da 1020 (+20 header). Come abbiamo visto, nell'header IP l'identificativo per i frammenti è lo stesso, e il flag va ad 1. **L'offset** rappresenta un puntatore al primo byte del frammento ma va misurato a blocchi di 8 bytes, quindi bytes/8. L'ultimo frammento ha flag zero. Per distinguere i pacchetti unici dall'ultimo frammento di un pacchetto basta vedere se sia flag che offset sono 0.



example:
♦ 4000 byte datagram
♦ MTU = 1500 bytes



IPv6 cancella frammentazione per agevolare

gestione buffer e riordinamento

IP Addressing e Subnetting

L'indirizzo IP, come abbiamo visto, è un identificatore da 32 bit. Ogni interfaccia di rete ha associato il suo indirizzo IP, gli host hanno generalmente un paio di interfacce, Ethernet e WI-FI, mentre un router ha *varie interfacce*.

Se in una rete è presente uno switch, a livello 3 questo può essere ignorato perché non lavora a livello 3, quindi in una rete si vedono il router e gli host connessi a questo router.

Le reti sono organizzate secondo una gerarchia, e come abbiamo visto, possiamo dividere l'indirizzo IP in network number e host number. Grazie alla maschera di rete (e a dei router ausiliari) possiamo dividere una singola rete più sottoreti o subnet.

Una **subnet** è un insieme di dispositivi che si connettono tra loro senza passare da un router (infatti, come dicevamo qui), che appunto distingue le varie subnet.

Possiamo identificare le subnet usando la notazione **CIDR** (**Classless Interdomain Routing**), che permette identifica porzioni di subnet di dimensione arbitraria con il formato **a.b.c.d/x**, dove **x** indica il numero di bit dedicati alla subnet.

ICMP

ICMP è un protocollo del livello rete, che invia messaggi riguardo lo stato della rete attraverso **UDP**. Più info qui.

DHCP e configurazione dell'IP Address

Un indirizzo IP può essere configurato sia manualmente ma anche automaticamente.

Il protocollo **DHCP** (Dynamic Host Configuration Protocol) permette di ottenere gli indirizzi dinamicamente quando un host si collega ad una rete, in maniera “plug and play”, in modo che chiunque ne faccia richiesta (al server DHCP) possa collegarsi alla rete. Questo metodo viene usato sia nelle reti Wireless che nelle LAN.

Ci sono 4 messaggi/fasi DHCP:

1. il primo detto **DHCP discover** viene mandato in broadcast da un client su tutta la LAN (o meglio su tutta la rete) e sta ad indicare che il client (o host) vuole connettersi alla rete.
2. Tutti i DHCP server che lo ricevono rispondono con un **DHCP offer** in cui offrono un indirizzo IP e la configurazione per la rete, e mandano questo pacchetto in **unicast** verso il client (si può vedere con)
3. A questo punto il client se riceve più offer effettua un messaggio **DHCP request** indicando il server di cui ha scelto l'indirizzo IP offertogli...
4. ...e quest'ultimo manda un **DHCP ack** per confermare l'assegnazione dello stesso.

Come abbiamo mezionato in questi 4 passi, una buona parte del lavoro è svolta dal Server DHCP (che spesso è integrato nei router moderni e non è separato). Il server DHCP possiede un blocco di host number e di IP address assegnabili. Periodicamente, il DHCP eseguirà il cosiddetto **leasing**, ovvero controllerà se l'host a cui aveva assegnato l'indirizzo IP è ancora connesso o meno, e in caso contrario, il DHCP server riterrà tale indirizzo IP come “libero” e volendo lo potrà assegnare ad un altro host. Un DHCP server può ritornare più del solo indirizzo IP della subnet, ad esempio l'indirizzo del primo router per quel client (**Default Gateway**), un nome e l'indirizzo IP del DNS server e la network mask.

Una richiesta DHCP viene incapsulata utilizzando UDP (siccome non può usare TCP dato che richiede l'indirizzo IP di dest. e mitt. oltre che alla porta, cosa che invece non è così in UDP. Inoltre, se riceve una richiesta con errori, semplicemente la può ritrasmettere facendo una nuova richiesta), poi in un pacchetto IP, in un pacchetto livello 2 (es. in 802.1 Ethernet) e viene mandata in **broadcast MAC** (quindi indirizzo MAC FF:FF:FF:FF:FF:FF) sulla LAN. Quando raggiunge il DHCP server, “spacchetta” tutti i pacchetti (ethernet → IP → UDP → DHCP) e lo manda sulla porta 67/68 (del router). Durante una comunicazione DHCP viene assegnato un *transaction ID* per identificare quella specifica trasmissione e differenziarla dalle altre DHCP.

ISP e comprare IP addresses

Un ISP, normalmente, compra un blocco di indirizzi IP che può assegnare ai suoi clienti. Un ISP che compri un indirizzo di rete, può vendere a delle organizzazioni delle sottoreti specificando la netmask. Se queste sottoreti non presentano buchi, ovvero sono **indirizzi contigui**, può fornire al control plane di Internet tutti gli indirizzi IP appartenenti alla sua rete e sottoreti. Se poi una di queste organizzazioni si spostasse sotto un altro ISP (quindi le sottoreti non hanno più indirizzi contigui), quando si fa richiesta per il control plane bisogna specificare di fornire le informazioni non solo delle sottoreti di quell'ISP ma anche dell'organizzazione che si è spostata. Quindi idealmente bisogna che gli ISP gestiscono blocchi contigui di indirizzi, in modo che il trasferimento fra ISP sia facile e poco complesso.

Un ISP compra un "blocco" di indirizzi dall'ICANN (Internet Corporation for Assigned Names and Numbers). Questa società alloca gli indirizzi, gestisce i DNS e assegna i nomi di dominio, risolvendo eventuali controversie.

NAT (Network Address Translation), ovvero perché abbiamo indirizzo 10.0.0.1

Un router ha almeno due interfacce, una verso la rete privata ed una verso internet.

Verso la rete internet, il router ha un suo indirizzo IP che gli fornisce l'ISP. A questo punto il router è un client dell'ISP su internet. La nostra rete locale sarà quindi definita dall'IP del router.

Dall'altro lato la rete privata cambia. Qualunque pacchetto che viaggia solo all'interno della rete privata senza dover andare all'esterno viene gestito dal router senza andare fuori.

Se invece la destinazione è esterna verso internet, il router fa evadere il pacchetto dalla rete locale, ed è facilmente gestibile dal router attraverso il suo indirizzo IP. Tuttavia quando arriva un pacchetto dall'esterno verso uno dei nostri host della rete privata, l'host "esterno" usa l'indirizzo IP del router come indirizzo di destinazione. Come farà dunque ad arrivare all'host della nostra rete privata?

A questo punto entra in gioco il **NAT** che, creando sul router un numero di porta e associandolo ad uno degli host della rete privata, definisce chi fa la richiesta e di conseguenza chi deve ricevere la risposta dentro la rete privata, quindi definisce un host della rete privata su internet.

Ogni router ha una **tabella di traduzione NAT** che associa ad ogni **indirizzo virtuale** un **indirizzo reale di internet** (questo indirizzo non è comunque un indirizzo reale dell'host, bensì è l'indirizzo del router+numero di porta).

Dunque, un router con NAT distingue i vari host associando a ciascuno di esso una porta.

Il limite è il numero di porte del NAT server. In questo modo si possono avere fino a 64000 connessioni con un unico "rappresentante" a livello di LAN, tuttavia in questo caso i router che implementano NAT devono prendere pacchetti fino al livello 4. Inoltre il NAT, grazie all'azione di camuffamento della sottorete, protegge gli host che rappresenta in quanto l'unico modo per indovinare un host è cercare la riga corrispondente nella tabella del NAT, che però non è accessibile a nessuno. Il NAT è anche abbastanza controverso, siccome secondo la filosofia iniziale dell'architettura iniziale di internet, i router dovevano solo agire fino al livello 3, e non anche a livello 4! (che è necessario, appunto, per capire a quale porta del router è destinato il pacchetto).

IPv6

Le motivazioni per l'utilizzo di IPv6 sono che i 32 bit per l'indirizzamento sono pochi visti tutti i dispositivi connessi ad internet. Inoltre l'overhead dovuto alla frammentazione dei pacchetti IPv4 è elevato e non fornisce qualità del servizio in IPv4.

I vantaggi di IPv6 sono che l'header ha una dimensione fissa di 40 bytes (grazie alla rimozione del campo options) e non c'è frammentazione dei pacchetti. I primi 4 bit sono sempre per la versione, poi in 4 bit IPv6 fornisce un campo addizionale per la **priorità** dei pacchetti all'interno dei flussi. I

ver	pri	flow label	payload len	next hdr	hop limit
source address (128 bits)					
destination address (128 bits)					
data					

no checksum perché il livello transport già ce l'ha sopra

gestione di flussi

2 flussi paralleli da questi 2 clienti sono possibili in modo + comodo

ce ne vuole per un flusso + anche same IP different port

restanti 24 bit sono la **flow label** che identifica un flusso di comunicazione, poi il **payload len** ovvero la dimensione del campo data, il **next header** indica il punto in cui iniziano i bit appartenenti ai protocolli di livello superiore all'interno del campo data. Il campo **hop limit** è equivalente al **TTL** e infine source address e destination address da 128 bit.

Non c'è più il checksum, il campo options è gestito fuori dall'header di IPv6 dal next header.

ICMPv6 è una nuova versione di ICMP che prevede un messaggio di errore che notifica la dimensione troppo grande del pacchetto nel caso in cui superi la dimensione massima del data link.

Infatti, non essendoci frammentazione bisogna gestire la dimensione dei datagrammi. In questo caso viene notificata la dimensione massima alla sorgente originale che comincerà a creare pacchetti di quella dimensione, evitando così la frammentazione.

Un indirizzo IPv6 è composto da 128 bit, con 6 campi da 16 bit (al posto di 4 per l'IPv4).

Tunneling IPv6

Il **tunneling** permette di incapsulare un pacchetto IPv6 in un pacchetto IPv4 trattandolo come payload (ovvero, inserendolo nel campo data di un pacchetto IPv4). Questo è necessario perché durante il tragitto si potrebbe dover passare per router IPv4. Esistono router speciali capaci di comprendere sia IPv4 che IPv6, così da fare da intermediari tra i due mondi internet.

SDN (software defined network) e OpenFLOW

In un approccio **SDN**, i processi di routing sono controllati da un'unità centralizzata, quindi è prevista la separazione netta tra il control plane ed il data plane.

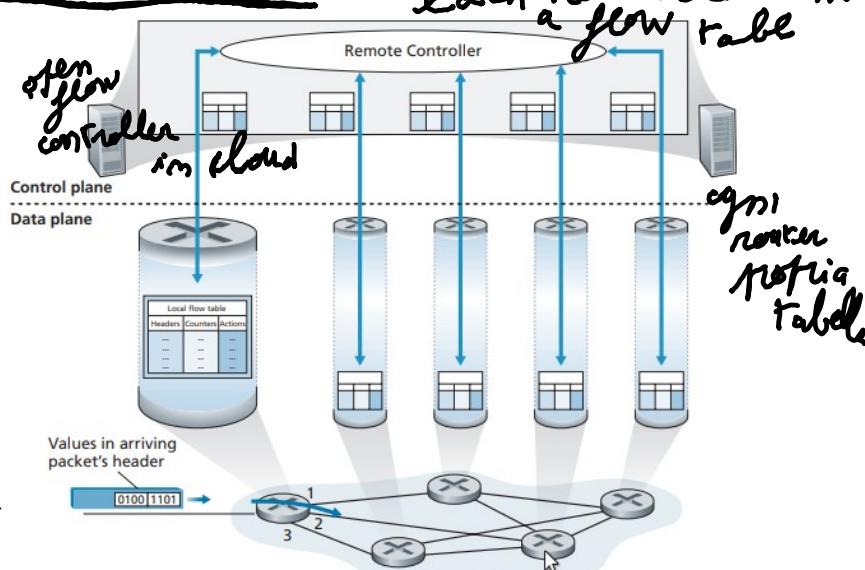
Il networking viene quindi gestito da un software, che scrive le **tabelle di flusso locali (flow table)** all'interno di ogni router. In altre parole, il controllo del traffico viene trapiantato su uno (o più) calcolatori che, mediante applicazioni specifiche, sono in grado di amministrare dinamicamente la rete.

La porzione logica della rete si accentra quindi in un'unica entità (il calcolatore di cui stavamo parlando prima, o network controller) dotata di una visuale panoramica di tutta la struttura. Queste flow table funzionano in modo simile a delle routing table. Grazie a questo metodo (in particolare, grazie alle flow table), gestire router, switch etc diventa molto semplice, in quanto vengono visti solo come semplici hardware di inoltro (descrivibili come "packet switches").

Ma come mai le flow table ci permettono di fare così tanto? La risposta sta nel fatto che le flow table hanno come campi delle **regole generali** da intraprendere in base alla casistica. Normalmente, infatti, il router analizza gli header del pacchetto e in base a queste informazioni decide l'azione da intraprendere.

Openflow è una implementazione di una metodologia di definizione delle flow table basate su SDN. In base a questa implementazione, le regole di forwarding (o in generale, le regole della tabella) vengono applicate in base alla corrispondenza con i valori dell'header di un pacchetto. Quindi, si potrebbe dire che i flussi sono definiti dai campi dell'header. In generale, il generalized forwarding usa una tabella di flusso (o match-plus-action tables) per stabilire come inoltrare tali pacchetti. I campi di queste match-plus-action tables prevodono:

- Un **pattern** col quale fare match con le informazioni nell'header del pacchetto
- Un **azione** corrispondente a tale pattern, che può essere di **doppiare**, **inoltrare** o **modificare** tale pacchetto. (Oppure, essere mandati al controller centrale magari per notificare al server che c'è una nuova condizione da introdurre)

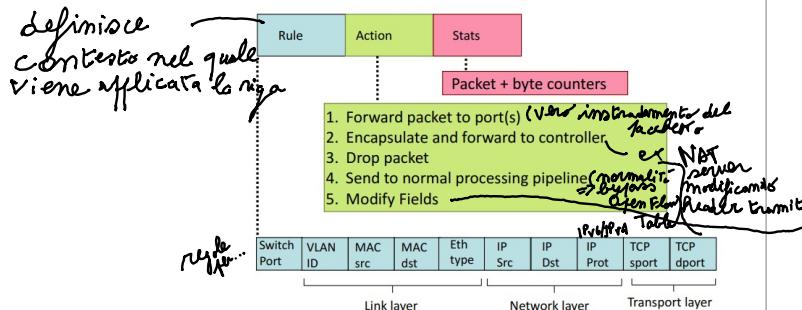


- Dei **contatori** che vengono aggiornati ogni volta che esce inoltre un pacchetto o che si svolge una determinata azione.

[Le slides del prof indicano anche un campo priority, che indica come comportarsi in caso di pacchetti con lo stesso match, ovvero disambiguare overlapping patterns].

In OpenFlow, le tabelle di flusso presentano tutti questi parametri.

OpenFlow: Flow Table Entries



Examples

Destination-based forwarding: *indirizzo in uscita sulla porta 6*

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Action
*	*	*	*	*	*	51.6.0.8	*	*	*	port6

IP datagrams destined to IP address 51.6.0.8 should be forwarded to router output port 6

Firewall:

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Forward
*	*	*	*	*	*	*	*	*	*	22 drop

do not forward (block) all datagrams destined to TCP port 22

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport	Forward
*	*	*	*	*	128.119.1.1	*	*	*	*	drop

do not forward (block) all datagrams sent by host 128.119.1.1

Le regole possono essere definite sul livello 2, 3 e 4. Quindi ci possono essere regole definite ad esempio a livello 2 che determinano le azioni da intraprendere quando arrivano pacchetti da un determinato MAC address, oppure regole di livello 3 simili alle classiche forwarding table, oppure regole per firewall che droppano tutti i pacchetti in arrivo da una sorgente o destinati ad una determinata porta. Quindi, attraverso il paradigma match+action si unificano differenti tipi di dispositivi: *=> settando opportunamente...*

- router tramite match con il longest destination IP prefix e le azioni di inoltrare i pacchetti in un output link
- switch tramite il match tra MAC address di destinazione e azione di inoltro verso una porta o tutte le porte di uscita
- firewall facendo match con IP address o TCP/UDP port e azione di concedere o negare
- NAT facendo match tra IP e porta e azione di riscrivere un indirizzo e una porta.
-

Le SDN però presentano anche una serie di problemi, principalmente il single point of failure e congestione.

La vera domanda ora rimane: come facciamo a computare le flow table e le forwarding table dei router? Per rispondere a questo, dobbiamo vedere il CONTROL PLANE, WOHOOO!!!

[RICORDA: nel destination based forwarding usiamo le forwarding table, nel generalized forwarding si usano le flow table].

Capitolo 5 – Network Layer – livello 3 – Control Plane

Come abbiamo visto all'inizio del capitolo 4, mentre il data plane implementa le funzioni di Forwarding attraverso l'applicazione delle tabelle di forwarding, il **control plane** determina il cammino da seguire, quindi in sostanza **si occupa della costruzione delle tabelle di forwarding**, determinando così il cammino che un certo pacchetto dovrà seguire.

Inoltre, abbiamo anche visto che esistono due metodi per strutturale il control plane:

- uno tradizionale, in cui il routing è calcolato localmente su ogni router
- uno globale, in cui il routing viene controllato da un'unità centrale (SDN).

Quest'ultima è preferibile perché le decisioni prese da singoli router possono entrare in contrasto, mentre l'utilizzo di un'unica unità centrale permette una migliore **coordinazione**.

Protocolli di routing

La funzione dei protocolli di routing è determinare il **path migliore** da un host sorgente ad un host destinazione attraverso una rete di routers. Un cammino buono può essere quello più veloce o meno costoso (dipende da diverse metriche).

routing tradizionale *link state = globale*
distanza vettoriale = locale, temporanea
della rete

sdn

~~rete con topologia complessa -> link state~~

~~conosciuto vicini ma non i vicini dei vicini -> distant vector~~

Dinamica - last so far

Una rete può essere vista come un grafo pesato, dove il peso può dipendere dal grado di congestione o da altri parametri, e i cui nodi sono i router.

Possiamo classificare gli algoritmi di routing in due modi:

- In un approccio **centralizzato** (centralized routing algorithm), si computa il percorso di minor costo usando una conoscenza globale e completa della rete. Il calcolo può essere fatto da un singolo computer (es. SDN) oppure da tutti i router della rete. Questo tipo di algoritmi, in cui la conoscenza della rete è globale sono detti **link state algorithms**. Per questo tipo di algoritmi, il cammino viene determinato attraverso l'algoritmo di **Djikstra** per gli shortest-path. Quando l'algoritmo link state termina, abbiamo per ciascun nodo il suo predecessore lungo il percorso a costo minimo dal nodo origine, in questo modo riusciamo a costruire l'intero percorso dall'origine a tutte le destinazioni. La complessità di questi algoritmi, a causa dell'uso di Dijkstra, è solitamente di $O(n^2)$.

A causa delle possibili oscillazioni di congestione nella rete, potrebbe essere che il costo del calcolo del percorso sia così grande da creare lui stesso congestione (in particolare, se devo fare dei calcoli continui del percorso). Inoltre, il continuo ricalcolo e scelta dei nuovi percorsi può essere causa di disordine dei pacchetti.

- In un approccio **decentralizzato** (decentralized routing algorithm), si computa il percorso di minor costo in modo iterativo e distribuito dai router. I router conoscono la situazione della rete solo per lo stato dei loro router vicini (siccome periodicamente viene mandata la propria tabella di forwarding ai router vicini), quindi il calcolo del percorso è iterativo, e in questo caso un nodo calcola in modo graduale il percorso di costo minimo. Questo tipo di algoritmi viene anche chiamato **distance-vector algorithms**. Questo tipo di algoritmi fanno uso dell'algoritmo di **Bellman-Ford** per calcolare il cammino. Questo tipo di algoritmi sono:
 - distribuiti**, nel senso che ciascun nodo riceve parte dell'informazione da uno o più dei suoi vicini direttamente connessi a cui.
 - iterativi**, nel senso che questo processo si ripete fino a quando non avviene ulteriore scambio informativo tra vicini
 - asincroni** siccome non richiedono che tutti i nodi operino al passo con gli altri.

Il motivo dell'uso di BF è che in questo modo si può, gradualmente, migliorare il cammino intrapreso. Infatti, ogni volta che un router si accorge di un miglioramento, lo notifica ai vicini, e ciò porta a dei cambiamenti a cascata, che a loro volta causano delle notifiche a cascata ai router vicini.

A causa di questi continui cambiamenti, però, non arriverò mai ad uno stato finale e ci saranno continui ricalcoli del percorso. Dunque, non otterrò mai una soluzione veramente ottima.

I cammini possono essere **statici**, ovvero cambiare poco in funzione del tempo o **dinamici**, ovvero soggetti a continui aggiornamenti, siccome i pesi dei link possono cambiare in continuazione.

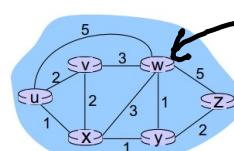
Come abbiamo visto, la velocità di convergenza dei link state è quadratica ma è preferibile a quella dei distance vector in quanto questi potrebbero entrare in routing loops.

D_{ij} : least path from one source to all other nodes

$D(v)$ - cost minimum per reaching dest so far

```

1 Initialization:
2   N' = {u}
3   for all nodes v
4     if v adjacent to u
5       then D(v) = c(u,v)
6     else D(v) = ∞
7
8 Loop
9   find w not in N' such that D(w) is a minimum
10  add w to N'
11  update D(v) for all v adjacent to w and not in N' :
12     $D(v) = \min(D(v), D(w) + c(w,v))$ 
13  /* new cost to v is either old cost to v or known
14  shortest path cost to w plus cost from w to v */
15 until all nodes in N'
  
```



non è graf. diretto
ma bidirezionale

clearly, $d_v(z) = 5$, $d_x(z) = 3$, $d_w(z) = 3$

B-F equation says:

$$d_u(z) = \min \{ c(u,v) + d_v(z), c(u,x) + d_x(z), c(u,w) + d_w(z) \}$$

$$= \min \{ 2 + 5, 1 + 3, 5 + 3 \} = 4$$

node achieving minimum is next hop in shortest path, used in forwarding table

Nel caso di link state inoltre, anche se un nodo calcola un costo sbagliato, ogni nodo calcola solo la sua tabella, mentre in distance vector se un nodo sbaglia il calcolo l'intera rete può essere compromessa (**routing table poisoning**).

Inter-AS e intra-AS

isola di internet sotto il controllo di un'autorità amministrativa

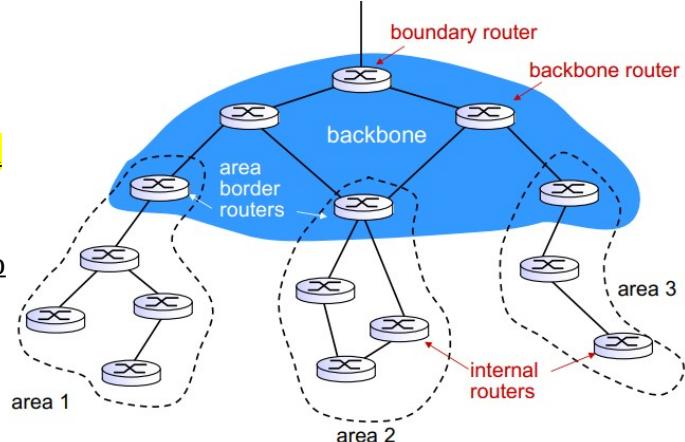
Nella realtà dei fatti vi sono distinzioni gerarchiche che rendono il calcolo degli algoritmi di routing più agevole. Queste distinzioni gerarchiche vengono fatte organizzando i router in aggregati detti **Autonomous Systems (AS)** (gli stessi AS visti prima). Tutti i router nello stesso AS utilizzano lo stesso algoritmo di routing, e se ad un certo punto i pacchetti entrano in un altro AS, allora verranno gestiti da altri algoritmi. Dunque, avremo bisogno di protocolli che permettono di intraprendere la comunicazione fra AS diversi. Quindi gli algoritmi di routing che sono in esecuzione *all'interno* di uno stesso AS sono detti **intra-AS routing protocols**, mentre quelli che permettono l'inoltro fra AS diversi sono detti **inter-AS routing protocols** (Nel caso di inter-AS le metriche del costo possono anche includere eventuali costi sul traffico di rete o accordi politici).

IGP e OSPF

Con **IGP (Interior Gateway Protocol)** si intende una categoria di protocolli intra-AS. I più famosi sono:

- RIP (routing information protocol)
- **OSPF** (Open Shortest Path First) questo è il più importante. Fa uso di Dijkstra e i messaggi scambiati con OSPF sono autenticati (ovvero mi garantisce l'identità del router che me lo invia) e quindi sicuri.
- IGRP (Interior Gateway Routing Protocol)

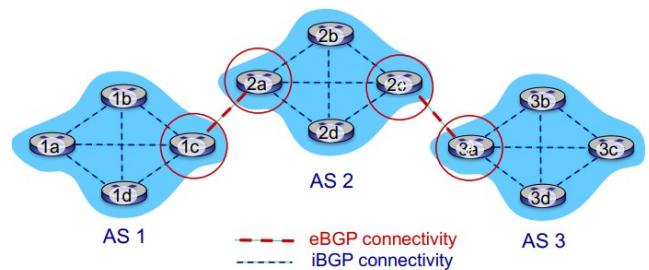
Anche all'interno dello stesso AS si possono suddividere aree e gerarchie. In ogni area, uno o più router di confine d'area (**area border router**) si fa carico dell'instradamento dei pacchetti indirizzati all'esterno. Un'area di un sistema autonomo OSPF è configurata per essere l'area di dorsale (**backbone area**), il cui ruolo principale è quello di instradare il traffico tra le altre aree nel sistema autonomo. La dorsale contiene sempre tutti i router di confine del sistema autonomo, ma non necessariamente soltanto quelli. L'instradamento nell'area di un sistema autonomo richiede che i pacchetti siano instradati dapprima verso il proprio router di confine (instradamento intra-area) e da questi, attraverso la dorsale, al router di confine dell'area di destinazione, che provvede a farli pervenire alla destinazione finale. OSPF fa uso del protocollo IP.



BGP (border gateway protocol) *(per aggiornare)*

È il protocollo utilizzato per connessioni **inter-AS**. Viene considerato il collante di internet. Il BGP implementa due funzioni:

- fornisce ad ogni AS le informazioni per la raggiungibilità dagli AS vicini (**eBGP**, che viene implementato solo dai boundary router, connessione che riguarda router appartenenti a diversi AS).
- propaga ad ogni router all'interno dell'AS le informazioni circa la raggiungibilità dei router interni all'AS (**iBGP**, comunicazione che riguarda router nello stesso AS).



gateway routers run both eBGP and iBGP protocols

I **gateway router** (ovvero i router di un AS che comunicano con altre AS) usano sia eBGP che iBGP. BGP include nei percorsi “buoni” non solo le metriche prestazionali ma anche le **policy**.

Il BGP permette inoltre di a una sottorete di notificare Internet della sua esistenza. Infatti, ogni volta che c’è una nuova rete in un AS viene fatta un “advertising” che informa, attraverso pacchetti eBGP dal border router di questo AS al border router di tutti gli altri AS.

Inoltre c’è un meccanismo che permette alle sottoreti di notificare al resto della rete la sua esistenza. [/?] I router di una BGP si scambiano messaggi (sono chiamati BGP peer) in cui pubblicizzano che sono i router gateway di un AS e se un nuovo router si aggiunge al suo AS direbbero anche che sono i router da contattare per inviare.[?/]

Le informazioni vengono propagate all’interno dell’AS poi usando iBGP.

BGP usa TCP per scambiarsi informazioni. I messaggi che utilizza sono:

- OPEN per aprire una connessione
- UPDATE per avvertire di nuovi percorsi (o rimovere quelli vecchi)
- KEEPALIVE per mantenere connessioni aperte anche in assenza di UPDATE
- NOTIFICATION per segnalare errori in precedenti messaggi e per chiudere le connessioni.

Se noi volessimo decidere il percorso che i dati devono compiere all’interno di un cammino, non possiamo semplicemente cambiare il valore dei costi dei collegamenti, ma dobbiamo usare degli algoritmi di routing come ad esempio OpenFlow così che noi possiamo decidere le regole. Quindi questo è possibile solo con SDN. Il problema è che questo paradigma è molto suscettibile ad errori. Quindi più trattabile ma anche più pericoloso.

Con **Traffic Engineering** si intende la possibilità di programmare il traffico per determinati dati, andando a modificare le tabelle di instradamento, ovvero programmare la rete. Dunque, in questo modo ho controllo sul control plane. Anche questo è possibile solo con le SDN.

Capitolo 6 – Cenni di sicurezza – livello bho

Garantire sicurezza, in generale, implica garantire:

- **Confidenzialità (confidentiality)**, che significa che solo chi manda e chi riceve il messaggio possono comprenderne il contenuto. Questo si può ottenere se il sender modifica il messaggio in un messaggio criptato, il receiver deve decriptarlo.
- **Autenticazione (authentication)**, sender e receiver vogliono confermare l’identità dell’uno e dell’altro, dunque vogliono essere sicuri che sia proprio il sender a mandare il messaggio e proprio e che il receiver sia quello che lo riceve. L’autenticazione dev’essere **mutua**, quindi si autenticano l’uno verso l’altro.
- **Integrità (message integrity)**, sender e receiver vogliono assicurarsi che il messaggio venga ricevuto senza alterazioni durante il transito. A
- **Accesso e disponibilità (access and availability)**, in quanto noi vogliamo che i dati siano accessibili e disponibili immediatamente.

Alcuni degli attacchi che un malintenzionato potrebbe fare sono:

- **Eavesdrop**: ovvero intercettazione di messaggi
- **Impersonification**: potrebbe far finta di essere qualcun’altro, modificando il source address nei pacchetti
- **Hijacking**: potrebbe intercettare la connessione e dirottarla a se, “sostituendosi” a uno dei estremi della conversazione
- **Denial of service**: impedire ad altri di fare un di un servizio, per esempio tramite un overloading di risorse.
- Potrebbe inserire attivamente dei messaggi nella connessione.

Crittografia

Per crittografare un messaggio, si dà come input un *messaggio in chiaro* (in inglese **plaintext**) a un **algoritmo di cifratura**. Questo lo trasforma in un **messaggio cifrato**, che viaggerà in rete come tale. Una **chiave di decifratura** poi permetterà al receiver di trasformare il testo cifrato nella sua controparte in chiaro, **decifrando**.

Possiamo rappresentare questi algoritmi attraverso una funzione non invertibile, in cui $m = K_b(K_a(m))$.

Nonostante la metodologia algoritmica sia nota, la sicurezza è garantita dalla chiave di cifratura.

Dunque, l'unico modo per smascherare il messaggio cifrato senza la chiave di cifratura è applicare tutte le chiavi possibili usando l'algoritmo noto (bruteforce), o anche utilizzare modelli statistici.

Un altro modo con cui si potrebbe craccare il messaggio si ottiene se l'intruso conosce una parte del messaggio in plaintext e quindi può ridurre drasticamente lo spazio delle chiavi di ricerca (**known-plaintext attack**).

Tipi di algoritmi per la crittografia – chiave simmetrica

Una **chiave** è detta **simmetrica** quando essa viene applicata sia per cifrare che per decifrare il messaggio. Per fare che con questo metodo la comunicazione sia effettivamente sicura, devo trovare un modo per comunicare la chiave simmetrica in modo sicuro.

Una semplice algoritmo di cifratura è la **cifratura di cesare**, che consiste nel mappare ogni lettera dell'alfabeto in un'altra (cifrario monoalfabetico), e quindi applicarlo alle lettere del messaggio. Nella sicurezza informatica si prende l'intera sequenza di bit e la si trasforma in un'altra sequenza alla quale poi applicare la chiave di cifratura.

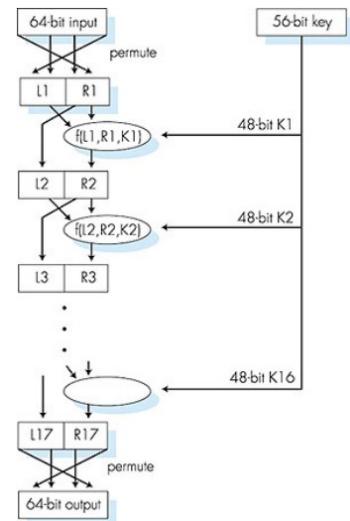
Uno degli algoritmi di cifratura informatica a chiave simmetrica più famosi è il **DES (Data Encryption Standard)**, che genera chiavi a 56-bit simmetriche e viene applicato su messaggi da 64-bit. In particolare, l'algoritmo viene applicato su blocchi da 64bit consecutivi e concatenati. Oggi per decifrare DES basta meno di un giorno, anche se non si conoscono ancora backdoor di questo algoritmo (ovvero che inverte la funzione). Un'evoluzione più sicura è chiamata 3DES che applica 3 volte di fila l'algoritmo DES, con 3 chiavi diverse.

Un algoritmo di cifratura a chiave simmetrica più solido è **AES (Advanced Encryption Standard)**. È un algoritmo di cifratura diverso e la chiave può variare in quanto dimensione, da 128, 192 o 256 bit. I blocchi da processare sono da 128 bit. Per craccarlo, normalmente ci si impiega un mese. Gli algoritmi con chiave simmetrica sono più efficienti e sprecano meno risorse.

Symmetric key crypto: DES

DES operation

initial permutation
16 identical "rounds" of function application, each using different 48 bits of key
final permutation



Tipi di algoritmi per la crittografia – chiave pubblica

Un'alternativa alla chiave simmetrica è la **chiave pubblica**. Sia mittente che destinatario hanno una chiave pubblica (che conoscono a tutti) ognuno ed una chiave privata ognuno.

Supponiamo di avere due innamorati di nome Alice e Bob che vogliono inviare un messaggio in sicurezza. Alice manda il messaggio m e tramite l'algoritmo applica la chiave pubblica di Bob al messaggio e lo spedisce. Dopo che il messaggio è finalmente arrivato a destinazione, Bob applica la sua chiave privata (l'unica che permette di decrittare un messaggio criptato con la sua chiave pubblica) e in questo modo potrà leggere il messaggio. In altre parole, data la chiave pubblica $K+$ applicata al messaggio m , grazie alla chiave privata $K-$ è possibile ottenere m applicando $K-(K+(m))$. Anche qui, la funzione di cifratura dev'essere non invertibile, e spesso si impiega la fattorizzazione dei numeri primi e le curve ellittiche per assicurare l'invertibilità. Un messaggio è

un pattern di bit, ed ogni pattern di bit può essere visto come un numero tramite conversione in binario. Crittare un messaggio è uguale a crittare un numero.

Teniamo a mente questa formula: $(a \text{ mod } n)^d \text{ mod } n = a^d \text{ mod } n$

Tra gli algoritmi a chiave pubblica, uno dei più famosi è l'algoritmo **RSA**, che funziona in questo modo: Prendiamo due numeri primi p, q rappresentabili in 1024 bit, tali che $n=p \times q$. L'unico modo per scomporre n è trovare p e q . Sia poi $z=(p-1)(q-1)$, che è un numero pari siccome p e q sono dispari. Scegliamo poi un numero e tale che sia minore di n e che non abbia divisori comuni a z ovvero sono primi tra loro (e dev'essere sicuramente dispari). Scegliamo d tale che $e \times d - 1$ è divisibile per z , quindi $(e \times d) \% z = 1$.

La chiave pubblica è $K_+ = (n, e)$ e la chiave privata $K_- = (n, d)$.

Per cifrare m (che è $< n$) si fa $c = m^e \% n$.

Per decifrare m , si fa l'operazione $m = c^d \% n$.

La cosa interessante è che vale la proprietà commutativa, ovvero $K_-(K_+(m)) = m = K_+(K_-(m))$, quindi il sender può certificare che m arriva da una certa persona se e solo se applicando la sua chiave pubblica otteniamo il messaggio m .

DES è 100 volte più veloce di RSA, quindi un metodo intelligente è scambiarsi chiavi simmetriche tramite chiavi private. RSA è sicuro perché anche se si conosce la chiave pubblica (n, e) , l'unico modo per trovare d è cercare i fattori di n senza conoscere p e q che è un problema difficile.

Authentication

Il problema ora è **garantire l'identità**. Per fare ciò, non basta dichiarare il proprio IP perché un intruso potrebbe usare un IP fittizio per fingersi qualcun altro. Anche usare una password non è abbastanza sicuro, siccome un intruso potrebbe copiare il pacchetto contenente la password e rimandarlo (**replay attack**).

Per rendere sicura la sessione di autenticazione, si usa quindi un **nonce**, un numero once-in-a-lifetime. Così, se qualcuno dovesse mandare tale pacchetto più di una volta, l'autenticazione fallirebbe. Per capire meglio facciamo un esempio:

Alice manda a Bob un messaggio. Allora Bob, per autenticare, manda ad Alice un **nonce R** unico, in modo che Alice ritorni il messaggio crittato contenente R utilizzando **una chiave simmetrica** che solo Alice e Bob conoscono. Quindi, per essere sicuri che la chiave simmetrica non venga rubata tramite una intercettazione, la si spedisce crittandola prima con la chiave privata. [DISCLAIMER: IL NONCE VA SEMPRE PRIMA CHIESTO AL DESTINATARIO, NON POSSO GENERARLO IO COSÌ DAL NULLA].

Facciamo un esempio solo con la chiave pubblica/privata. Ancora, Alice manda un messaggio, e Bob manda R e chiede ad Alice di cifrarlo con la sua chiave privata, così che possa essere sicuro dell'identità di Alice, dato che solo la chiave pubblica di Alice riesce a ritornare il messaggio originale. Tuttavia, se Bob non conosce la sua chiave pubblica, lui dovrà richiederla ad Alice.

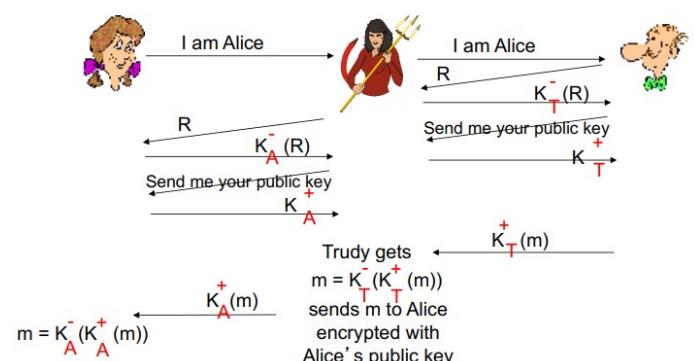
Questo genera un problema di sicurezza non banale, infatti se un intruso intercetta la richiesta di Bob, potrebbe inviare il messaggio cifrato con la sua chiave privata, e quando Bob richiederà la chiave pubblica, l'intruso invierà la sua chiave pubblica. In questo modo, Bob crederà di

comunicare con Alice, ma invece starà comunicando con la perfida Trudy (**man in the middle**). (Trudy volendo potrà anche ridare il controllo ad Alice e modificare i messaggi, che sarà ancora peggio!!!).

Quindi, nonostante questo schema impedisca un replay attack, c'è comunque la possibilità di un man in the middle.

Come abbiamo visto, la debolezza di questo schema sta nel momento in cui si richiede la chiave pubblica del secondo interlocutore, in cui Trudy può mettersi

man (or woman) in the middle attack: Trudy poses as Alice (to Bob) and as Bob (to Alice)



in mezzo. Dunque, un modo per certificare che la chiave pubblica sia effettivamente di Alice, è attraverso le **certification authority**. Dalle CA (da non confondere con control agent dei router) possiamo ricavare la chiave pubblica di Alice, e ci garantiscono che sia effettivamente quella "giusta". Per garantire che le certification authority siano effettivamente "sicure" e non un altro che si pone come tale, queste sono a loro volta sono certificate da una rete di CA (su internet dice inoltre che per verificare l'identità delle CA, si può far uso della sua chiave privata). Questi sono i famosi "**certificati di sicurezza**" usati dai browser, e ogni volta hanno una scadenza, in modo che rinnovando i certificati un possibile intruso avrebbe la vita troppo difficile. In poche parole quindi, usare la chiave privata è come "firmare il messaggio", mentre usare la pubblica permette di verificare che la firma sia corretta.

Message Integrity (o come essere sicuri che il messaggio non sia stato modificato).

Per garantire l'**integrità** del messaggio, si usa un metodo simile alla firma a mano, così da non rendere discutibile l'autore del messaggio. Per ottenerlo basta cifrarlo con la propria chiave privata. Si manda m sia in chiaro sia crittato, e se i due messaggi si equivalgono, ovvero $m = K+ (K- (m))$, allora si arriva alla conclusione che Bob è l'effettivo autore e il messaggio è arrivato con successo. Il motivo per cui mandiamo anche il messaggio in chiaro è perché se si manda solo il messaggio crittato, potrebbero comunque essere stati cambiati bit a caso (magari da Trudy o da un'interferenza) e non avremmo nulla con cui confrontarli. Ci serve, appunto, per valutare l'**integrità del messaggio**. L'intruso, infatti, potrebbe ricevere $K- (m)$ e modificarlo a caso, poi usare $K+$ ed ottenere m' , così da sostituirlo anche ad m e mandare ad Alice questi due messaggi. Però appunto m' contiene valori casuali, che non c'entrano nulla con il vero valore di m , e quindi l'autenticazione fallisce.

Se m è un messaggio molto lungo, $K- (m)$ richiede molto lavoro. Per evitare ciò creiamo un **message digest**, ovvero calcoliamo, con una funzione di hash H , un valore hash $H(m)$ di dimensione costante. In questo modo, se anche cambiasse un bit tra m ed m' , $H(m)$ e $H(m')$ saranno completamente diversi. Le funzioni hash sono anche sicure, siccome sono difficilmente invertibili, quindi non riusciremo MAI a ricavare m da $H(m)$. Gli m che generano collisione sono inoltre diversissimi tra di loro, quindi raramente avremo due m diversi con lo stesso hash. [/?] Il checksum soddisfa la proprietà commutativa, quindi da un m possiamo facilmente ricavare gli m' con lo stesso valore di checksum[?]. Quindi adesso, quando dobbiamo spedire il messaggio m , oltre a criptare il plaintext, inviamo anche un message digest $H(m)$, ovvero $K- (H(m))$ per certificare che siamo noi gli autori del messaggio, e usiamo l'algoritmo di RSA solo ad una piccola parte rispetto ad m in modo da rendere il processo di autenticazione più efficiente.

Quindi, riassumendo: Bob invia ad Alice il **messaggio criptato** e il **message digest** (anch'esso crittato), dopodichè quando Alice riceve il messaggio lo decripta, ed applica $H(m)$ al messaggio decriptato. Dopodiché, decripta $H(m)$ ricevuto da Bob e confronta il digest ricevuto con quello calcolato. Se i due valori ottenuti si equivalgono, allora è certificato (il messaggio è integro e sono certo che sia Bob ad avermelo inviato).

Alcune funzioni di hashing sono **MD5** che genera un valore di 128 bit in 4 step. **SHA-1** genera message digest da 160 bit.

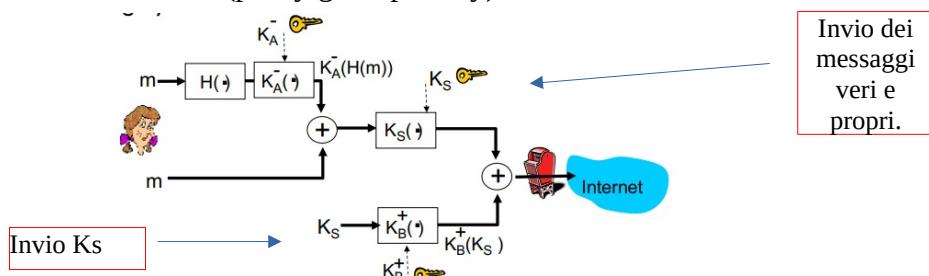
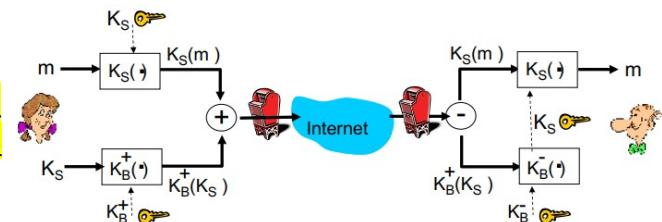
Come registrarsi ad una Certificatoin Authority?

Per registrare una chiave pubblica, in una certification authority bisogna innanzitutto provare la propria identità alla CA. Dopodiché, la CA crea un certificato digitale in cui associa/lega l'identità del richiedente alla sua chiave pubblica, firmandola con la chiave privata della CA. Chiunque volesse garantire che la chiave sia autentica, ovvero derivi effettivamente dalla CA, può applicare la chiave pubblica della CA (K_{CA}), che si può richiedere ad un'altra CA. (Inoltre, la CA manda anche un digest, per verificarne l'integrità del messaggio).

Secure e-mail (o come assicurare confidenzialità, ovvero segretezza).

Supponiamo che Alice voglia mandare una mail confidenziale a Bob. Se m è molto grande, conviene fare uso di chiavi simmetriche, dunque Alice genera $K_s(m)$ e spedirà questo messaggio crittato. Dopodiché, Alice passa anche la **chiave simmetrica** K_s , in modo che Bob possa leggere effettivamente il messaggio. Per spedire questa chiave, e garantire la sicurezza, prima di spedirla si critta la chiave con la **chiave PUBBLICA** K_B^+ di Bob ottenuta da una CA, in modo da spedire un messaggio tale che sia $K_B^+(K_s)$. L'unico che potrà leggere il messaggio (o meglio, la chiave) sarà così Bob, che quindi può estrarre K_s e applicarla ad $K_s(m)$ per rigenerare e leggere il messaggio m originale. **A questo punto può continuare ad usare K_s nella stessa connessione TCP, e Alice e Bob potranno continuare a scambiarsi messaggi crittati.**

Se Alice volesse fornire anche l'autenticazione e l'integrità del messaggio, può prendere m e calcolarne il digest, che verrà poi firmato con la chiave privata di Alice $K_A^- (H(m))$ e lo invia insieme al plaintext crittato nel modo spiegato prima. A questo punto Bob chiede K_B^+ di Alice ad una CA, confronta i due $H(m)$ e quindi se sono uguali **m è autentico ed integro**. Se volessimo comporre le due cose creiamo il digest e prendiamo anche m . Poi questi due messaggi vengono spediti in sequenza tramite K_s e quest'ultima viene spedita con $K_B^+(K_s)$. Questo è detto **schema PGP** (pretty good privacy).



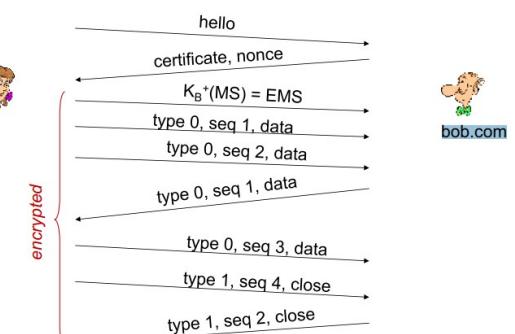
Alice uses three keys: her private key, Bob's public key, newly created symmetric key

Secure TCP (SSL, Secure Socket Layers)

SSL è un protocollo largamente utilizzato, supportato da praticamente tutti i browser e utilizzato da http nella sua versione https. Questo servizio viene applicato a livello applicazione. Una variazione si chiama TLS.

SSL fornisce **confidenzialità, integrità e autenticazione**. Per implementare una conversazione sicura, si devono redere sicuri alcuni passi:

1. HANDSHAKE: Bob inizia una connessione **TCP** con Alice. Bob manda un messaggio "SSL" con cui dichiara di voler usare SSL, al cui Alice risponde con il proprio certificato contenente la chiave pubblica. Bob usa la chiave pubblica di Alice per generare un **Master Secret (MS)** che poi Bob critta con la chiave pubblica di Alice (**encrypted master secret EMS**). Alice riceve l'**EMS** e ricava, con la sua chiave privata, il **MS**.
2. A questo punto Bob ed Alice possono scambiare messaggi sicuri generando chiavi ottenute grazie al **segreto condiviso (key derivation)**.
3. I dati vengono trasmessi in blocchi critptati, con grandezza minore del messaggio originale.
4. La connessione viene chiusa tramite **speciali messaggi (autenticati)** e divisi in tipi (quindi si usa type 1 per messaggi normali, type 0 per messaggi di chiusura). In questo modo ci si assicura che non sia un intruso a voler chiudere la connessione (**truncation attack**), e a questo punto le **chiavi vengono distrutte**.



Anche in questo caso, per evitare reply attack si usano **nonce** per ogni messaggio di andata e ritorno. Gli elementi di cifratura utilizzati da SSL sono gli algoritmi di chiave pubblica, di chiave simmetrica e i **MAC algorithm** (Message Authentication Code). I MAC (ovviamente diversi dall'indirizzo MAC di livello 2) sono l'equivalente di una hash function, che ci garantisce l'integrità del messaggio. Un tipico header SSL (o SSL record) è composto da:

- 1 byte di content type che specifica il contenuto
- 2 bytes per la versione di SSL utilizzata
- 3 bytes per la lunghezza del campo dati e poi i dati (crittati)
- ed infine il MAC che è l'equivalente dell'hash function.

SSL usa RSA per chiavi pubbliche, AES per simmetriche e per hash MD5. Inoltre, la length permette anche di stabilire il numero del pacchetto inviato (come in TCP).

Sicurezza nel livello rete (3) (IPsec)

Se volessimo garantire sicurezza anche al livello rete, dobbiamo rendere sicuro lo scambio di pacchetti attraverso IPsec. Garantendo sicurezza al livello trasporto, possiamo rendere segreti le informazioni nei segmenti TCP o UDP, i messaggi ICMP o i messaggi BGP.

L'uso predominante di **IPsec** è per la creazione di **VPN (Virtual Private Network)**. Le VPN sono reti private virtuali alle quali possiamo accedere da remoto come se fossimo fisicamente connessi in quella rete, per fare questo dobbiamo spedire i pacchetti a livello IP in modo che sia impossibile capire da dove stiamo comunicando (crittando quindi le informazioni del livello trasporto).

I pacchetti mandati su internet hanno un **IPsec header** oltre al normale header IP che viene letto da tutti router su internet, e i router vedono solo le reti di destinazione e mittente. Mentre l'header **IPsec** crittato che contiene le informazioni su chi è il reale host destinazione può essere analizzato solo dal router certificati che implementano il protocollo IPsec. Quando il pacchetto IPsec arriva al router dell'azienda, viaggia all'interno della rete aziendale come se fosse stato generato da un host interno. Le VPN vengono usate anche dalle società quando devono usare l'internet pubblico, siccome permette di comunicare in segretezza e siccome è molto sicuro.

La cosa bella dell'IPsec è che automaticamente protegge anche le informazioni del layer più alti di quello di rete, in particolare garantisce il **data integrity, l'autenticazione dell'origine, previene replay attacks, e assicura confidenzialità**.

Firewall

Un **firewall** server viene utilizzato per isolare la rete interna di un'azienda dal resto di internet e filtrare i pacchetti che passano attraverso. Serve inoltre per prevenire un DoS attack o modifiche a dati della rete interna (infatti il firewall contiene una lista di host, una whitelist, in cui mostra gli utenti con i vari permessi).

I firewall possono essere suddivisi in tre categorie:

- **stateless packet filter**: filtri che non considerano il contesto.
- **stateful packet filter**: filtri che considerano il contesto (context-sensitive).
- **application gateways**.

Un **firewall stateless** (o stateless packet filter) è un classico firewall con funzioni di filtraggio dei pacchetti. Viene installato a livello del router o vicino ad esso, e può decidere se inoltrare i pacchetti o dropparli basandosi su IP address e IP destination, TCP/UDP source e destination port number, richieste ICMP e richieste TCP SYN e ACK (per DoS).

La tabella delle regole viene chiamata ACL (Access Control Line) ed è simile alle tabelle di OpenFlow.

action	source address	dest address	protocol	source port	dest port	flag bit
allow	222.22/16	outside of 222.22/16	TCP	> 1023	80	any
allow	outside of 222.22/16	222.22/16	TCP	80	> 1023	ACK
allow	222.22/16	outside of 222.22/16	UDP	> 1023	53	---
allow	outside of 222.22/16	222.22/16	UDP	53	> 1023	----
deny	all	all	all	all	all	all

corrente elettrica =
movimento di elettroni che si
muovono

+ V, + cariche libere in grado di
spostarsi
elettroni (resiste dall'accanire
negativo)

Lo stateful packet filter tiene traccia dello stato di ogni connessione TCP, quindi mantiene più informazioni. Per questo motivo, analizza se i pacchetti in arrivo hanno senso nel contesto (es. ACK o FIN), e chiude le connessioni TCP inattive, impedenendo l'arrivo dei pacchetti da essi. Nelle ACL di questi firewall c'è un campo in più, chiamato "**check connection**" (ovvero check connection). Questo parametro, se contrassegnato (ovvero posto a 1) indica che deve controllare la tabella dello stato di connessione prima di accettare un pacchetto, in caso appunto l'host sia stato inattivo per troppo tempo.

Le application gateways sono dei server specifici per l'applicazione che permettono di bypassare il controllo del packet filter per una determinata applicazione. Ad esempio se volessimo usare un telnet verso l'esterno ma il firewall blocca le connessioni telnet, possiamo connetterci all'application gateway tramite SSH e fare un telnet da lì che sarà fatto passare dal packet filter.

La DMZ è un'area che da servizi a chi fa richiesta (Web server, FTP server ecc.). Questi servizi devono essere raggiungibili da internet, quindi il firewall non deve bloccarne i pacchetti. Tuttavia, il firewall dovrà mettersi fra gli host dipendenti dell'azienda e il DMZ, in modo che eventuali attackers non possano entrare con gli host di questi. Il firewall deve quindi filtrare i pacchetti che arrivano alla DMZ facendoli passare e impedire il passaggio dei pacchetti verso la rete interna aziendale.

~~cosa sono i generi progettare un suo radio come vengono nell'ambiente come gestione~~
Capitolo 7 – Physical Layer (nelle reti wireless) – livello 1

Concetti chiave della trasmissione radio (e sulle onde)

Le onde radio sono onde elettromagnetiche generate da correnti alternate (AC) all'interno di un circuito.

Le antenne convertono la corrente elettrica in segnali a radiofrequenza e viceversa.

Alcuni concetti delle onde EM:

~~creiamo di una variazione / t~~
L'ampiezza (A) rappresenta la distanza fra il picco positivo e negativo nell'onda del segnale, che viene rappresentata come una sinusoide. Tanto più la sinusoide è ampia, tanto più il segnale è ampio. L'ampiezza della sinusoide corrisponde alla quantità di potenza energetica della trasmissione, che si misura in watt (ed è direttamente prop. alla diff. di potenziale). ($P = V \cdot I$)

~~cicli sinusoidati comparsi~~
La frequenza (f) è il numero di cicli che vengono effettuati in ogni unità di tempo.

• La lunghezza dell'onda radio (λ) è data dalla formula c/f , dove c è la velocità della luce ed è la frequenza (in Hz). Le antenne con una ricezione migliore sono quelle lunghe $1, \frac{1}{2}, \frac{1}{4}$ della lunghezza d'onda. (Per questo motivo, le antenne dei dispositivi IEEE 802.11 sono lunghe 12.5 cm).

In generale, possiamo descrivere un'onda con l'equazione (o formula):

$$y = A \sin(f \cdot x + p)$$

Dove A è l'ampiezza, f è la frequenza e p è la fase. Se avessimo un coseno, basta aggiungere una fase di $\pi/2$ per riportarlo alla formula del seno.

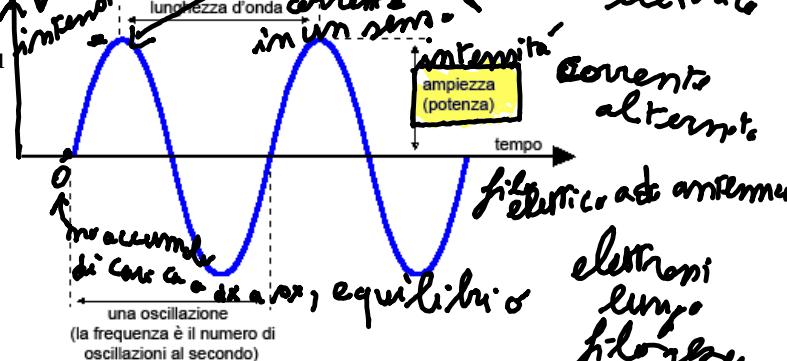
~~corrente che l'antenna trae~~

Uno dei problemi che si incontra dei segnali radio è che con l'aumentare della distanza, la potenza del segnale decade in modo esponenziale, dunque dopo una certa distanza il nostro segnale non sarà più percepibile. In particolare, una volta che la potenza del segnale scende sotto il signal detection limit, non riesco più a decifrare i bit che sono stati trasmessi. Siccome l'andamento è esponenziale, per raddoppiare la distanza di ricezione di un fattore 2 bisogna almeno

energia e frequenza onda radio \leftrightarrow corrente con
propagazione nel vuoto

$$\text{Wavelength} = \frac{\text{Velocità luce}}{\text{Cicli/sec}}$$

quanti è
= mili
mili
mili
mili



quadruplicare la potenza trasmissiva (infatti, l'area di una sfera è $4\pi r^2$, dunque la potenza di ricezione sarà proporzionale a $1/d^2$, dove d è la distanza fra receiver e sender).

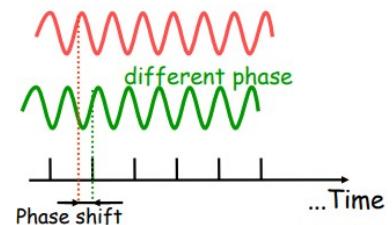
In base alle distanza dalla sorgente, possiamo dividere gli intervalli di ricezione in 3 “range”:

- **Transmission range** è l'intervallo in cui la comunicazione è possibile e tutti ricevono dati con basso error rate. Questo è anche l'intervallo di “copertura” del segnale.
- Il **detection range** è l'intervallo in cui posso riconoscere, recepire la trasmissione, ma siccome il segnale è troppo debole, non riesco a decifrare i dati.
- Il **range di interferenza** è dove i segnali sono incredibilmente deboli e potrebbero non essere rivelati o riconosciuti.

Questi 3 intervalli dipendono sia dalla potenza di trasmissione del sender che dalla sensibilità di ricezione del ricevente. Inoltre, le onde radio ad alte frequenze sono buone per corte distanze ma subiscono la resistenza degli ostacoli, mentre le basse frequenze si comportano meglio su lunghe distanze e subiscono meno la resistenza degli ostacoli.

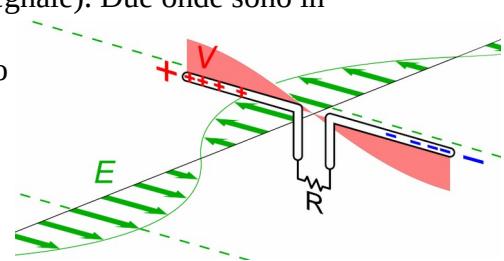
Fase e polarizzazione

La **fase (ϕ)** è lo spostamento nel tempo della sinusoide rispetto ad un segnale di riferimento, e che si misura in gradi o radianti. Una **fase positiva** significa che l'onda è “in anticipo” (left-shift), mentre una **fase negativa** significa che l'onda è “in ritardo” (right-shift). Più vado lontano dall'origine più il segnale originale subisce uno sfasamento.



Se però due onde sfocate collidono, vanno in **interferenza**, e la differenza di fase può avere un **effetto additivo oppure distruttivo** nei confronti dell'ampiezza d'onda (e quindi anche nei confronti dell'intensità di segnale). Due onde sono in opposizione di fase quando lo sfasamento è di 180° .

Le onde em sono delle perturbazioni del campo magnetico ed elettrico che si propagano nello spazio, e in particolare il campo magnetico ed elettrico si propagano su piani ortogonali tra loro, quello elettrico è parallelo all'antenna. La differenza di potenziale (ovvero il voltaggio), a valore diversi genera onde di ampiezza diversa (in particolare, in cui il campo elettrico è diverso).



La **polarizzazione** indica l'inclinazione dell'antenna, si parla di horizontal e vertical polarization nel caso in cui il campo elettrico sia parallelo o perpendicolare al terreno. La polarizzazione verticale è tipicamente usata nelle WLAN. Quando le due antenne (sender e receiver) sono polarizzate allo stesso modo, allora si cattura il maggior segnale possibile.

Nei luoghi chiusi non dovrò preoccuparmi troppo della polarizzazione, siccome le onde “rimbalzano” sui muri e quindi la polarizzazione si auto-risolve. Al contrario, è un bel problema negli spazi ampi e aperti.

Amplificazione di segnale

Per aumentare il segnale, e quindi prolungare la trasmissione, possiamo fare uso degli **amplificatori radio**. Questi prendono in input segnali radio e ne aumentano l'ampiezza in due modi:

- con **guadagno attivo**, attraverso l'uso di amplificatori, abbiamo una sorgente di energia addizionale che aumenta l'ampiezza del segnale in input e lo inoltra.
- con **guadagno passivo**, con l'uso di parabole si catturano molte quantità di energia, e la fanno rimbalzare tutta al centro della parabola, dove ci sarà l'antenna.

Nel caso in cui il guadagno sia negativo, abbiamo delle **perdite**. Queste perdite possono essere:

- intenzionali (causate da resistenze che trasformano l'energia in calore)
- da ostacoli (muri, acqua e distanza).

Effetti che un'onda può subire

L'onda può quindi subire vari "effetti":

- **shadowing**, che impedisce che l'onda prosegua
- **reflection**, se rimbalza su una superficie
- **refraction**, quando cambia direzione
- **scattering** quando rimbalza in maniera casuale colpendo un angolo spigoloso
- **diffraction** quando colpisce bordi a curvatura molto ampia.

Impedenza e VSWR

Quando creiamo un sistema di comunicazione con il trasmettitore che manda un segnale grazie a dei connettori fino ad un'antenna, ogni volta che collegiamo due dispositivi dobbiamo far attenzione al valore di **impedenza** (ovvero di resistenza nel mezzo), perché se hanno impedanze diverse si verifica il **VSWR (Voltage Standing Wave Ratio)** che causa una dissipazione di energia. Dunque, l'impedenza idealmente deve avere sempre un rapporto 1:1.

Intentional Radiator (IR), radiatore isotropico e l'EIRP

Con **Intentional Radiator (IR)** si intende tutto l'insieme di dispositivi (cavi, connettori..) dal trasmettitore fino all'antenna esclusa. L'**energia di un sistema radio**, ovvero l'energia che arriva all'antenna, è calcolata come la potenza in uscita dall'IR meno le tutte le perdite nel percorso, ed è anche chiamata **intentional radiator power output**.

Un **radiatore isotropico** è un modello ideale di un antenna, ed emette l'energia omogeneamente in tutte le direzioni, al contrario di un'antenna normale, che invece concentra la potenza in una direzione piuttosto che in un'altra.

Il valore **EIRP (Equivalent Isotropically Radiated Power)** rappresenta l'energia che dovrei dare ad un radiatore isotropico per produrre lo stesso segnale di un'antenna direzionale.

Decibel, ovvero come misurare le variazioni di potenza di un'onda

Il **decibel** è un'unità di misura per esprimere le perdite di potenza, e misura le differenze di potenza tra due segnali in **scala logaritmica**. Ad esempio il rapporto tra 1 e 1000 è 1:1000, ma $1=10^0$ e $1000=10^3$, quindi ci sono 3 bel di differenza ovvero **30 dB** (1 bel = 10 decibel). Quindi nella scala esponenziale cresciamo ogni volta di un valore 10, mentre nella scala logaritmica aggiungiamo sempre 1.

Per indicare la forza di un segnale in dB bisogna avere un segnale di riferimento. **10dB** significa moltiplicare per 10, **20db** significa moltiplicare per 100, **30dB** per 1000 ecc. Se il valore in dB è positivo, abbiamo un guadagno; altrimenti una perdita di potenza. Per misurare la differenza di potenza in dB tra il segnale trasmesso Tx e ricevuto Rx si utilizza la formula $10 \cdot \log(Rx/Tx)$. (In generale, per misurare la differenza fra due segnali, siccome come abbiamo detto prima c'è bisogno di un segnale di riferimento, si usa la formula $10 \cdot \log(P_a/P_b)$). Sommare 10dB equivale a moltiplicare per 10, -10 equivale a dividere per 10, sommare 3 dB equivale a moltiplicare per 2 e sottrarre per 3 equivale a dividere per 2.

DecibelMilliWatt (dBm) *scala assoluta*

Il **dBm** indica una misura assoluta di energia. Per permettere una conversione facile da dB a dBm, assumiamo che

$$\underline{1 \text{ mW (milliWatt)} = 0 \text{ dBm.}}$$

Per convertire dalla potenza in mW a potenza in dBm, si usa la formula:

$$P_{\text{dBm}} = 10 \cdot \log(P_{\text{mW}})$$

e la formula inversa è, invece, $P_{\text{mW}} = 10^{(P_{\text{dBm}} / 10)}$.

DecibelIsotropic (dBi)

Un radiatore isotropico, come sappiamo, è utopico. L'antenna che gli assomiglia di più è il **dipolo**, che spara onda elettromagnetica forte in orizzontale ma quasi nulla in verticale. I **dipoli a basso guadagno** producono più energia in verticale rispetto all'orizzontale, questo rispetto ai **dipoli ad alto guadagno**, che producono molta energia in orizzontale e pochissima in verticale. La direzione nella quale un'antenna spara il massimo dell'energia si chiama **direzione preferenziale**.

Il **dBi** misura la variazione di energia di un dipolo rispetto ad un radiatore isotropico. Si calcola quindi in questo modo: $\text{dBi} = 10 \times \log(\text{P}_{\text{antenna}}/\text{P}_{\text{antenna isotropica}})$.

Vari tipi di antenna

Le **antenne omnidirezionali** sono quelle che si avvicinano di più ad un radiatore isotropico, ovvero un dipolo (infatti sono più o meno sinonimi).

Un **antenna tilt** si ottiene inclinando la direzione dell'antenna rispetto ad un asse perpendicolare. Le **antenne semidirezionali** concentrano l'energia verso una determinata **direzione a cono**. Alla base del cono, se prendiamo le estremità misuriamo una variazione di -3dB rispetto all'energia al centro, ovvero la metà.

Un'**antenna altamente direzionale** concentra tutta l'energia lungo un cono con un'ampiezza molto piccola.

Un'**antenna a settore** è un particolare tipo di antenna che spara segnali in un settore circolare per coprire più spazio possibile. È costituita da diverse antenne semidirezionali, che sono usate per dividere il segnale in diversi settori/canali logici.

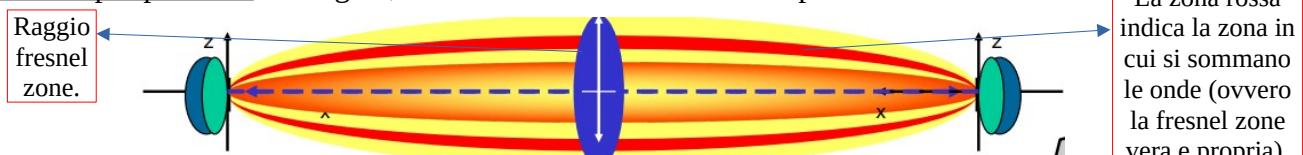
In generale, le antenne non sono MAI da sole, infatti ogni dispositivo ha almeno due antenne per sfruttare il principio di **diversity**. Questo perché se ho due antenne, riceveranno lo stesso segnale con un grado di rumore diverso, quindi in questo modo potò scandire più facilmente il fattore comune di questi segnali, in modo da isolarlo dagli altri rumori.

Come rappresentare un'antenna

- Posso usare l'azimuth chart, che rappresenta il "pattern" (ovvero, il modo in cui si propagano i segnali nelle direzioni) di un'antenna dall'alto.
- Posso usare l'elevation chart, che rappresenta il "pattern" di un'antenna guardandolo di fronte in varie direzioni.

Fresnel zone

La **fresnel zone** è un'area centrale tra due antenne che comunicano, che va lasciata libera da ostruzioni perché vi passano più segnali di tipo additivo sul ricevente. Il raggio della fresnel zone non dipende dal tipo di antenna **ma solo dalla distanza e dalla frequenza**, quindi è lo stesso sia per dipoli che per parabole. Di regola, la fresnel zone non va ostruita più del 20%.



Consideriamo il blocco causato dai rialzi della terra, quindi se usassimo delle antenne che hanno una forte distanza fra loro. La fresnel zone sarebbe occupata al più del 40%! Perciò, abbiamo bisogno di una distanza minima fra le antenne. L'altezza minima di un'antenna dovrà essere quindi

$$h = (43.3 \frac{d}{4f}) + \frac{d^2}{8}$$

Path Loss – perdita di segnale e Link budget

La **perdita di segnale** in funzione della distanza dalla sorgente si misura in decibel con la formula

$$\text{Perdita (in dB)} = 36.6 + 20 \cdot \log(f \text{ in MHz}) + 20 \cdot \log(d \text{ in miglia}).$$

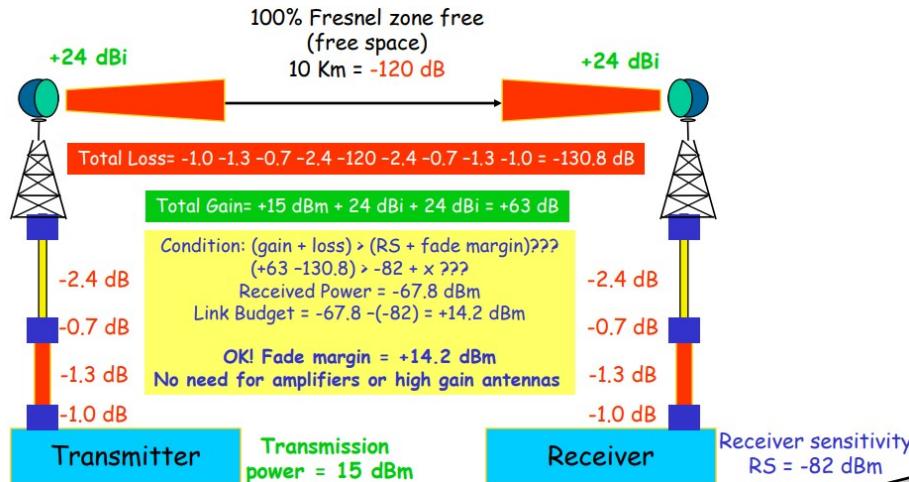
Nella realtà, quando noi raddoppiamo la distanza il segnale decade di circa $\frac{1}{4}$ (per le ragioni viste in precedenza).

Il **link budget** indica l'eccesso di segnale tra trasmittente e ricevente (dalla soglia minima di ricezione), ovvero la distanza tra l'energia in un certo punto e la soglia di sensibilità.

Se il **link budget** è **positivo** in un certo punto, significa che arriva segnale al ricevente, altrimenti non riuscirà a ricevere la trasmissione. In generale, il link budget si ottiene con la formula:

$$LB = \text{Potenza ricevuta (in dBm)} - \text{RecevierSensitivity (in dBm)}$$

La sola positività del link budget, però, non garantisce l'efficienza della comunicazione, siccome se sono poco di sopra alla soglia minima basterà un piccolo ostacolo per interferire nella comunicazione. Dunque, il link budget dev'essere superiore ad un **system operating margin (o fade margin)** che è solitamente tra 10 e 20 dB.



Trasmissione dati wireless

Le reti wireless, sorprendentemente, possiedono una larghezza di banda: la **larghezza di banda** di un canale wireless dipende da quanto tempo è necessario per rappresentare un bit.

Ora, supponiamo di avere due host A e B, uno low Transmission Power e l'altro High Tx Power. Se la copertura di B raggiunge A, ma quella di A non raggiunge B, allora ho un **link unidirezionale**, in cui B potrà inviare messaggi ad A, ma non viceversa.

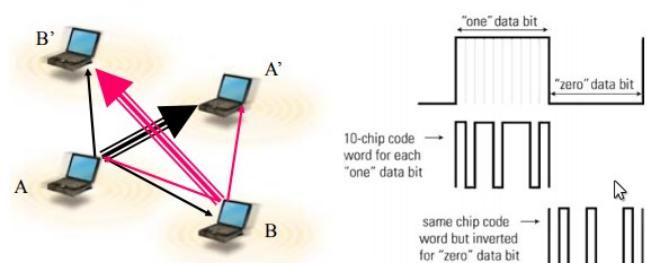
Se invece si raggiungono a vicenda, il link sarà bidirezionale e simmetrico, ed entrambi potranno comunicare.

Una **tecnologia wireless narrowband** (banda stretta), fa uso di una sola, determinata frequenza, e perciò potrà coesistere con gli altri segnali. Quando avremo tante, diverse frequenze narrowband, per distinguere l'una dall'altra dovremo fare uso di **filtri passabanda**, che permettono di isolare una sola frequenza.

Frequency hopping spread spectrum si differenzia dalla precedente in quanto due host, per comunicare, non fanno uso sempre della stessa frequenza, bensì effettua salti in maniera pseudocasuale tra una frequenza e l'altra (spesso con un pattern generato da una funzione hash).

Questo metodo, però, richiede che i due host siano perfettamente sincronizzati.

Direct Sequence Spread Spectrum (DSSS) rappresenta ogni bit con un pattern di chip (ovvero valori binari). Quindi se 1 equivale ad un chip, 0 è lo stesso chip capovolto. Si usa questa tecnica perché ogni chip viene trasmesso su tutto lo spettro della banda a disposizione. Le interferenze che si creano si annullano grazie alle differenze di pattern, così il chip (o meglio, i dati in formato chip) di un host appare ad un altro come semplice rumore.



Struttura delle reti wireless e gestione dei canali logici wireless

Una **rete ad hoc** è una rete senza un'infrastruttura predisposta, come quella bluetooth, e che si instaura su richiesta e solo per quel tipo di comunicazione. Si contrappongono alla rete **ad infrastruttura**, che richiedono cablaggio, access point e altro.

Di norma un access point deve implementare sia stack per connessioni con filo sia per connessioni wireless, e permette di far comunicare questi due mondi tramite **le bridging function**. Come abbiamo già visto [qui](#), per creare più canali sulle stesse risorse possiamo suddividere in base allo spazio, al tempo, alle frequenze o al codice, in particolare:

- **Frequency multiplex** significa che ogni canale utilizza un dato range di frequenze.
- **Time multiplex** si utilizzano tutte le frequenze ma per un periodo limitato di tempo.
- **Time e frequency multiplex** è una fusione delle due tecniche sopra, in cui si fanno salti di frequenza sincroni nel tempo (una sola frequenza per un periodo limitato).
- **Code multiplex** tutti utilizzano lo stesso range di frequenza nello stesso tempo, ma con diversi codici di chipping sequence (in modo simile a [DSSS](#)).
- **Frequency planning** si riusano le stesse frequenze ad una certa distanza tale da non creare interferenza.

Modulazione e codifica dei dati in modo wireless

Con modulazione, si intende la trasformazioni delle informazioni logiche, convertite in segnale elettrico, in un segnale trasmissibile nel mezzo.

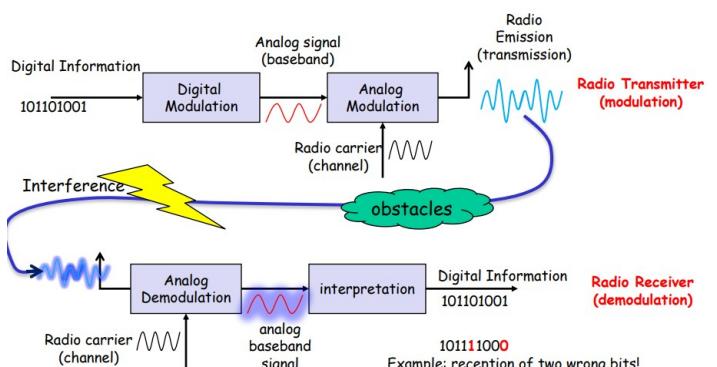
Usando le onde radio vogliamo riuscire a codificare dei valori digitali. Facciamo distinzione fra:

- **Analogic Modulation**, che prende il segnale analogico e ne copia le caratteristiche sul nostro canale radio.
- **Digital Modulation**, vogliamo trasmettere attraverso onde radio dei bit digitali, 1 e 0.

Il processo di modulazione e demulazione avviene in questo modo:

Lato sender	Lato receiver
<ol style="list-style-type: none"> 1. Do la mia sequenza di bit al modulatore digitale, che crea una sinusoide <u>che rappresenti la mia sequenza</u>. 2. Do la sinusoide appena creata al modulatore analogico, che <u>genera un segnale</u> alla giusta frequenza (ovvero quella del canale su cui vogliamo trasmettere). 3. Mando questa trasmissione radio all'antenna (per esempio, dell'access point). 	<ol style="list-style-type: none"> 1. L'antenna cattura il segnale radio, che sarà un po' <u>disturbato</u> da fattori naturali/ostacoli ed eventuali interferenze. 2. A questo punto entra in gioco il demulatore analogico, che <u>cercherà la frequenza giusta</u> (ovvero quella corrispondente al canale assegnato) <u>ed estrarrà da essa i dati inviati</u>. 3. <u>La sinusoide finale</u> che otterrò non sarà molto chiara e definita, ma nonostante ciò <u>verrà interpretata dall'interprete</u>, che la trasforma in una sequenza di bit.

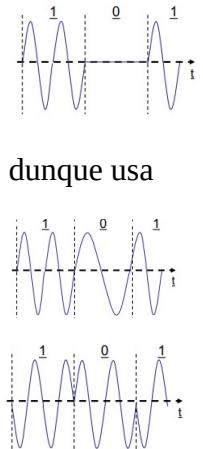
Ci sono vari modi di gestire gli eventuali errori: potrei per esempio gestirli come segmenti MAC, ovvero facendoli ritrasmettere, ma la procedura sarebbe davvero troppo costosa. Se invece capissi quali sono i bit errati, possono correggerli senza il rinvio del pacchetto.



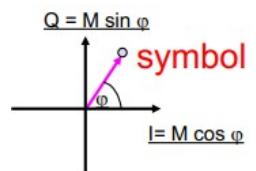
Tecniche di modulazione analogica.

Esistono molte tecniche per convertire una informazione digitale in bit in una sinusoide. Tra queste, le più importanti sono:

- **ASK (Amplitude Shift Keying)**, in questo essenzialmente uso la frequenza del canale su cui voglio trasmettere e in base all'ampiezza della mia onda, indico se il bit è 0 o 1. In particolare, se la ampiezza della sinusoide è piatta, allora il mio bit sarà 0, altrimenti il mio bit sarà 1. **Questo metodo è molto soggetto ad interferenze**. Questa tecnica fa uso solo di una data frequenza definita del canale, dunque usa poche risorse dello spettro.
- **FSK (Frequency Shift Keying)**, uso due diverse frequenze, una di frequenza bassa, che indica lo 0, e l'altra di alta frequenza. **Questo metodo fa uso di uno spettro più grande di frequenze, dunque usa più risorse del canale**.
- **PSK (Phase Shift Keying)**, la sinusoide del bit 1 e quella del bit 0 hanno la stessa frequenza, e varia esclusivamente la fase. In base quindi al ritardo o all'anticipo della sinusoide, codifico a 0 o ad 1. **Questo è poco soggetto ad interferenza**, ma è più complesso da implementare.

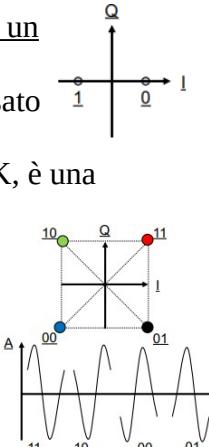


Possiamo usare un tipo di codifica diversa, più complessa. Ci sono infatti modi differenti di rappresentare le caratteristiche di un'onda em. Oltre a classici modi (usando la sinusoida per esempio) abbiamo il metodo che consiste nel rappresentarlo con un **diagramma di fase e ampiezza**, un cui essenzialmente usiamo la fase e l'ampiezza come coordinate polari. Un **simbolo**, ovvero uno dei punti in questo diagramma, rappresenterà uno dei possibili stati in cui si può trovare il nostro segnale radio.



Possiamo sfruttare questa rappresentazione per codificare il nostro segnale. Inoltre, con questo metodo, la frequenza non inciderà sulla rappresentazione del dato, siccome sarà solo un valore in Input al modulatore che ci indica su quale canale stiamo comunicando. Esistono vari modi per applicare questo concetto:

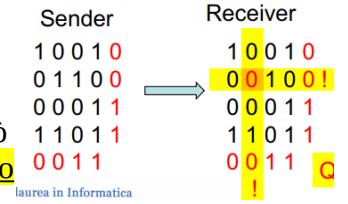
- **BPSK (Binary Phase Shift Keying)**, in questa codifica, ogni simbolo rappresenta un bit. In particolare, se la fase è 0 (e quindi l'angolo è 0), allora il bit sarà 0, mentre invece se la fase sarà di 180° , allora il bit associato sarà 1. Questo metodo viene usato nelle comunicazioni satellitari, tuttavia ha una "efficienza spettrale" bassa, ovvero abbiamo pochi bit per codifica. In generale, comunque, essendo un derivato di PSK, è una tecnica molto solida e molto resistente alle interferenze.
- **QPSK (Quadrature Phase Shift Keying)** è l'evoluzione di BPSK, in cui ogni simbolo rappresenta due bit, e non più solo 1, e avrà così un totale di 4 simboli e 2 bit per segnale. I simboli vengono interpretati associandoli allo stato più simile rispetto a quando arrivano. Con questo metodo, infatti, la tolleranza all'errore (fault tolerance) sarà molto minore rispetto al BPSK, siccome possiamo facilmente confondere un segnale per un altro.



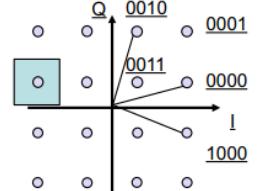
Dunque, a seconda della situazione in cui mi trovo, mi converrà fare un cambiamento della tecnica di codifica. In particolare, se avrò molti ostacoli e molte interferenze, mi converrà usare la BPSK, in caso contrario userò la QPSK, spendendo così più dati alla volta. (In generale però, si parte sempre con una BPSK ed eventualmente si cambia a QPSK).

Per scegliere come posizionare le "etichette" dei bit associati ai simboli, conviene usare delle etichettature con una distanza di hamming che sia di massimo 1, siccome l'errore capita solitamente perché confondo una etichetta con l'adiacente. Infatti, se le etichette adiacenti sono il più simili possibile, abbiamo anche il minor numero di bit errati possibile (quindi, minimizziamo l'errore di interpretazione). Se riusciamo ad identificare un errore, possiamo correggerlo

con il bit di parità, ovvero aggiungiamo un bit per fare in modo che il numero di 1 sia pari, e se il receiver riceve una sequenza con un numero dispari di bit (nonostante il bit di parità sia ad 1) si accorge della presenza di un errore. Il bit di parità però non riconosce se 2 bit sono errati, dunque si può fare uso di una **matrice di parità**, che organizza i bit in una matrice, mettendo alla fine di ogni riga e ogni colonna un bit di parità, così se un bit cambia riusciamo non solo ad identificarlo ma anche a correggerlo, guardando se uno dei due bit sia “scorretto”.

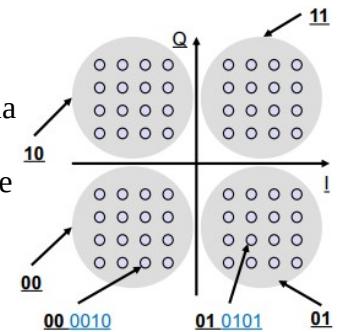


Se la nostra connessione va benissimo, possiamo usare una codifica ancora più densa: la **Quadrature Amplitude Modulation (QAM)** ha 16 simboli, ed ognuno codifica 4 bit in cui ogni stato cambia sia fase che ampiezza. Posso mettere anche più simboli, ma in questo modo la tolleranza agli errori diminuisce gradualmente. Inoltre, la distanza di hamming fra un simbolo e l’altro deve comunque essere di 1 al massimo. (Notare come nella figura, 0011 e 0001 hanno stessa fase ma diversa ampiezza).



Il caso nella figura qui a fianco viene chiamato QAM-16 siccome ha 16 simboli.

Un altro modo di codificare i dati è attraverso **Hierarchical Modulation**, in cui ho 4 “nuvole di bit” ognuna contenente 16 simboli e ogni nuvola ha associati 2 bit, quindi ogni simbolo rappresenta 6 bit. Questa struttura dei simboli a gerarchia permette di differenziare due flussi (o meglio, tipi) di dati, ad esempio in una video call, in cui vengono trasmessi più dati video che audio, possiamo associare 2 bit di audio a 4 bit di video. In questo modo se non c’è interferenza trasmitteremo ad alta velocità e in modo corretto, mentre se c’è rumore, difficilmente un errore sarà tale da far confondere una nuvola con un’altra, quindi i dati video potrebbero arrivare sbagliati ma la voce sarà trasmessa correttamente. (Questo siccome è difficile che ci sia confusione fra “nuvole” diverse). È MOLTO IMPORTANTE NON CONFONDERE QAM CON HM E VICEVERSA!!!!!!



Protocolli MAC (Medium Access Control) nelle reti wireless

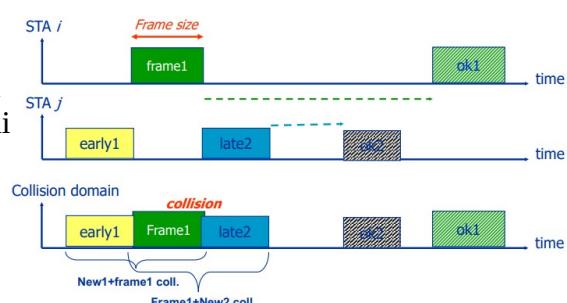
Per le reti wireless, non possiamo usare gli stessi protocolli MAC delle reti cablate. Questo per una serie di motivi, tra cui:

- Il fatto che le reti wireless siano dei **broadcast naturali**.
- Nelle reti wireless, ho il problema delle collisioni dovuto non solo alla comunicazione nello stesso momento, ma anche alla comunicazione nello stesso luogo. Dunque la collision detection dovrà essere gestita in modo diverso.
- Mentre Ethernet, in caso di collisione, ferma tutti gli host e li fa partire in modo sfasato, ciò non è possibile nelle reti wireless, siccome con le onde radio porterei la connessioni all’infinito. Infatti, non posso fermare le onde radio. Avrò quindi bisogno di **collision avoidance**, NON collision detection. La collision detection nei sistemi wireless infatti causa uno spreco di potenza e di canale.

I protocolli wireless si suddividono in due categorie: in base a *quando* devo trasmettere (**time domain**) e in base a *dove* devo trasmettere (**space domain**).

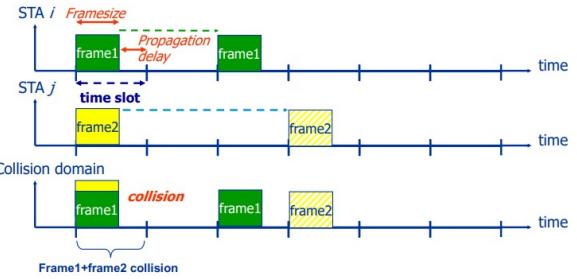
Time domain wireless protocols

1. Uno dei primi protocolli wireless mai creati è il **protocollo Aloha**, che aspetta fino a che c’è un pacchetto pronto, e appena questo è pronto, lo invia subito. Dopodiché, aspetta un RTT, e se non riceve l’ACK, aspetta un tempo casuale di attesa nel quale ritenta a mandare il pacchetto, altrimenti manda quello successivo. Con questo schema, il rischio di

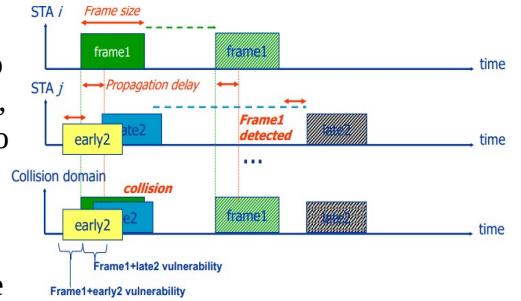


collisione è molto alto, e la **vulnerabilità di un frame** (ovvero il tempo durante il quale nessun'altro oltre a me deve trasmettere) è pari a due volte la sua dimensione.

- Il protocollo **Slotted Aloha** introduce la suddivisione del tempo in slot ed il frame può essere trasmesso solo all'inizio di un nuovo slot di tempo. Quindi la vulnerabilità diventa pari alla grandezza del frame, siccome c'è il rischio di collisione solo se arriva un segmento all'inizio dello slot corrente.



- Il protocollo **CSMA** può ascoltare il canale grazie a dei chip, e trasmette solo se nessuno sta trasmettendo. Nel caso il canale sia occupato, calcolo un tempo random e riascolto, questo finché non “sentirò silenzio”. Se dopo che ho inviato il pacchetto non arriva un ACK entro un RTT, ascolto e ritrasmetto come all'inizio. In questo modo, c'è un basso rischio di collisione, e la vulnerabilità è pari al due volte il propagation delay (ovvero il tempo che impiega il segnale radio ad arrivare da una stazione all'altra)



- Se usiamo **slotted CSMA**, che fa uso anche questo della divisione del tempo in slot, la vulnerabilità diventa una sola volta il propagation delay.

Space domain wireless protocols

Il problema però è che un ricevitore può avere segnali da più parti, e quindi bisogna decidere non solo quando parlare ma anche chi deve parlare da dove. Quindi, devo capire se il destinatario è libero da altri segnali. Per evitare una collisione useremo dei messaggi **CTS (Clear To Send)** e **RTS (Request To Send)** in questo modo:

- Il mittente manda un messaggio RTS al destinatario
- Se il messaggio arriva al destinatario, risponde con un CTS al mittente, e questo indica che sarà libero.
- Il CTS però non arriverà solo al mittente iniziale, ma anche a tutti gli altri host che sono nel raggio di trasmissione del destinario. E siccome loro, al contrario del primo mittente, non hanno mandato un RTS, “intuiranno” che quell’host sta comunicando con qualcuno e staranno in silenzio.

Il protocollo **RTS/CTS** conviene quando ho molti dati da trasmettere, siccome se ne ho pochi non mi conviene sprecare questo scambio di messaggi (in generale, si usa solo se il pachetto è di dimensione maggiore di 2KB).

Lo standard 802.11 funziona con due schemi MAC preesistenti:

- il **DCF (Distributed Coordination Function)**, questo fa uso di CSMA/CA con l'uso di Binary Exponential Backoff come metodo di randomizzazione dei tentativi di trasmissione. Questo è usato soprattutto nelle reti di tipo ad hoc, siccome non necessita di una base centrale/controllore centrale. Questo metodo non ha QoS, e l'accesso alla rete è conteso.
- il **PCF (Point Coordination Function)** usa una base station o un coordinatore centrale, che essenzialmente decide il turno di comunicazione (“di chi parla”). Di base (livello 0) ha anche lui il DCF, ma in presenza del coordinatore esso prende il controllo del canale. L'accesso è atteso.

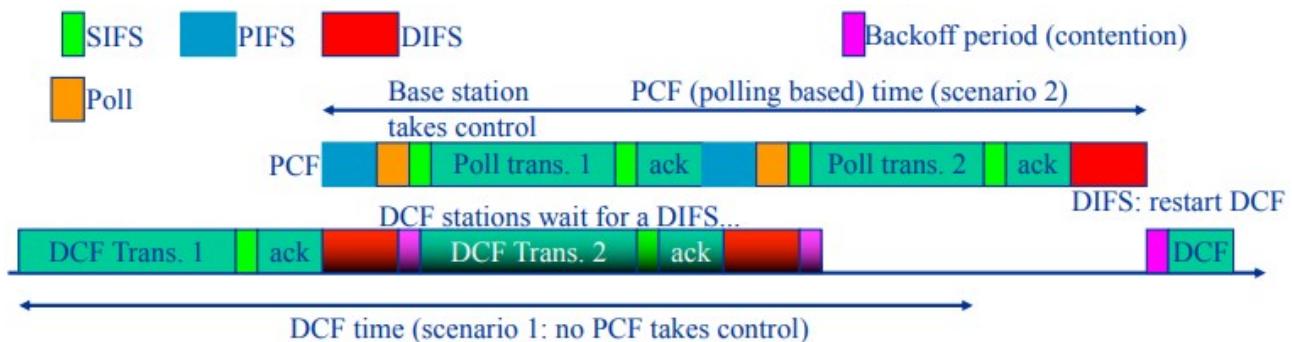
Questi due schemi si attuano in congiunta in questo modo:

Prima di tutto, un access point trasmette un pacchetto chiamato Beacon che contiene le informazioni necessarie (es. rete, canale virtualizzato...), e che permette di schedulare chi parla.

Quando finisce questa fase (**PCF**), fino al prossimo beacon si entra nella fase decentralizzata (o distribuita) (**DCF**) in cui tutti provano a parlare. Se il beacon non arriva, si continua ad usare la DCF. Dopo una trasmissione DCF, c'è un intervallo detto **SIFS** (Short Inter Frames Space) durante il quale il destinatario può inviare l'ACK.

A questo punto trascorre un altro intervallo di tempo, detto **PIFS** (Point IFS), durante il quale l'access point (o base station) se vuole può prendere il controllo del canale che pù generare dei poll e chi viene chiamato può fare la trasmissione, poi si aspetta un altro **PIFS** e così via. Il controllo quindi è in mano all'access point finché ha host da chiamare.

Quando l'access point ha finito di chiamare le stazioni, si aspetta un **DIFS** (Distributed IFS, intervallo di valore massimo), dopo un DIFS chiunque può trasmettere, e si ritornerà in fase DCF (quindi con slotted CSMA). In questa fase, ogni host avrà un numero random di slot da aspettare prima di trasmettere (**random backoff**). Se dopo la fine di uno slot, un host non potrà parlare (perché il canale è occupato, ricordiamo che si usa CSMA) allora si duplica il range del random backoff (se prima era da 0 a 5, ora sarà da 0 a 10), e ciò sarà un sintomo di troppa gente che sta trasmettendo. Se si arriva al settimo tentativo di trasmissione, il MAC protocol si arrende e notifica al piano di sopra che la comunicazione non è possibile.



BONUS: reti satellitari e cellulari

Nelle reti satellitari, i satelliti vanno lanciati alla giusta velocità. In base alla loro velocità, possono avere un'orbita bassa (in se sono più veloci del moto rotatorio terrestre) oppure un'orbita alta (se sono più lenti del moto rotatorio terrestre). I satelliti ad orbita stazionaria, invece, hanno una velocità pari a quella della terra.

Nelle reti cellulari (o multi-wlan), abbiamo una griglia di Access Point, che coprono tutto il territorio e che permettono ai terminali mobili di connettersi ovunque essi si trovino. Se cambieranno posizione, ambieranno anche frequenza di comunicazione.

Capitolo Bonus – consigli utili per gli esercizi

Esercizi di congestione dei router

Per questo tipo di esercizi, solitamente ci sarà un tot. di host (es. 8 host), e una probabilità di scegliere un certo link di output di un router, sapendo che la banda massima dell'output link è di un certo dato. Per questo tipo di esercizi, è fondamentale sapere la formula del intensità di traffico, ovvero:

$$[\text{INTENSITÀ DI TRAFFICO} = L \cdot A / R]$$

L=bit per pacchetto

R=la velocità di trasmissione

A=pkgs in arrivo

Sapendo questo, bisognerà anche considerare la probabilità della scelta di un link. In quel caso, quindi, bisognerà moltiplicare $L \cdot A$ per la probabilità della scelta del link con la banda limitata. Ricorda, se l'intensità è > 1 , allora C'È CONGESTIONAMENTO.

Link Budget

Per questo tipo di esercizi, è fondamentale ricordarsi la [formula della perdita, la formula del link budget](#), e sapere che il LB deve essere maggiore di un system operating margin, che solitamente è tra 10 e 20 [dB](#).

Subnetting

Oooohohoh, questa è lunghina. Prima di tutto dobbiamo vedere l'IP della nostra rete (ovvero il network number iniziale). Questo avrà anche una netmask in CIDR solitamente, e in base a quella possiamo calcolare i parametri.

Il primo host è dato dal network number + 1, l'IP del router è dato ponendo tutti i bit del nostro host number a 1 (oppure comando il valore massimo dell'host number - 2 (es. 255 - 2, 63 - 2 etc...)) e l'ultimo host è l'IP del router se si intende quello come l'ultimo host (di solito dice proprio "ultimo IP assegnabile", intende esattamente quello), oppure è l'IP del router - 1.

Adesso arriva la parte brutta. Bisogna ora passare alle sottoreti.

Prima di tutto, ti conviene partire a fare il calcolo partendo dalla [sottorete della sottorete con più host](#) (e se ha fratelli, ancora quella con più host). Supponiamo che questa sottorete (B) abbia una sola sottorete senza fratelli (che chiamiamo B1). Dopodiché, dobbiamo guardare il numero di host che dobbiamo assegnare. Se sono 143, allora avremo bisogno di 8 bit per l'host number, stessa cosa se sono 128 host, o 127 host, oppure 126 host se devo escludere il router (in modo da poter assegnare 254 host. 7 bit non bastavano, dato che grazie a quelli potevamo assegnare solo 127 - 1 host).

Dopodiché, possiamo passare ai calcoli. La netmask sarà data da (32 - numero di bit dell'host number), quindi se 8 bit sono 24 bit per il network number, e la netmask in CIDR sarà /24.

A questo punto, il primo host sarà dato dal network number + 1. Siccome siamo al primo calcolo, o alla prima sotterete, come [network number si usa lo stesso della rete grande](#), con una netmask CIDR diversa però.

Possiamo poi passare alla rete sopra alla nostra, ovvero la rete B. Da qui, usiamo lo stesso esatto metodo usato prima, con lo stesso network number. Inoltre, aggiorniamo il default gateway di B1, che corrisponde al router di B. Il DG di B sarà il router di N.

Benissimo, primo branch fatto.

Passiamo ora al fratello di B, A, che anch'essa ha un solo figlio A1.

Partendo sempre dal basso, dobbiamo procedere esattamente come abbiamo fatto prima, se non per il fatto che il network number da cui dovremo partire (sia per A che per A1) non sarà più il NetNum di N, bensì sarà il router di B + 2! E quindi il primo host, sia di A che di A1, sarà Router di B + 3.

Per facilitare la risoluzione degli esercizi, è bene ricordarsi queste corrispondenze.

1000 0000	128	11 (2)	3
1100 0000	192	111 (3)	7
1110 0000	224	1111 (4)	15
1111 0000	240	11111 (5)	31
1111 1000	248	111111 (6)	63
1111 1100	252	1111111 (7)	127
1111 1110	254	11111111 (8)	255
1111 1111	255	111111111 (9)	511

velocità della luce (c) = 300.000 Km/sec

Per il procedimento: al prof basta che tu scriva numero di bit, con host. E che faccia vedere PERCHÈ l'ultimo host a tale IP.

NAT

In questo tipo di esercizi abbiamo una tabella da compilare, e abbiamo un router NAT e degli host. In pratica, ci viene dato l'IP e la porta del Server di destinazione e dell'host dal quale parte il pacchetto. Dobbiamo tenere a mente che il NAT crea una porta (al quale gli diamo il numero che vogliamo noi, 1000 è un buon numero) dedicata a questo host. Il WAN side address allora sarà l'IP_del_router:1000 e il LAN side address sarà l'IP_dell_host:porta_host. L'address di destinazione del pacchetto sarà l'IP_del_server:porta_del_server.

Decodifica di un'onda sinusoidale

Questo tipo di esercizi è abbastanza lunghetto. Prima di tutto, ci viene chiesta cosa indicano le lettere nella [formula dell'onda sinusoidale](#). Per questo, conviene guardare la formula [qui](#).

Dopodiché, ci viene chiesta quale fra le due codifiche utilizzate è la migliore, e quale nome dobbiamo dargli. Possiamo trovare la risposta spiegazione [qui](#), tuttavia solitamente basta scegliere quella in cui i simboli adiacenti hanno distanza di hamming 1, e il nome della codifica sarà QAM se si fa anche distinzione di ampiezza (con numero dei simboli vicino), oppure PSK se c'è solo disintinzione di fase fra i simboli.

Ora arriva la parte bella, ovvero la decodifica. Solitamente, nello schema della codifica viene specificato come si indica l'angolo del ritardo e l'anticipo, tuttavia è bene tenere a mente che l'anticipo è dato da un'onda che si trova più "avanti" nel tempo rispetto il riferimento, mentre il ritardo è dato da un'onda che si trova più indietro nel tempo rispetto al riferimento (possiamo vedere le due cose immaginando di visualizzare l'andamento delle onde nel tempo).

Se abbiamo QAM, dobbiamo anche usare la decodifica in base all'ampiezza dell'onda, quindi se l'onda sarà grande allora dobbiamo usare il simbolo più lontano dal centro, viceversa se sarà piccola dobbiamo usare il simbolo più vicino al centro. È MOLTO IMPORTANTE CHE PER L'AMPIEZZA NON SI FACCIA IL CONFRONTO CON L'ONDA DI RIFERIMENTO! Altrimenti sei fregato. Devi letteralmente guardare l'ampiezza dell'onda che hai, senza fare confronti, tutto qui. Per fare la codifica, è bene inoltre distinguere ampiezza grande con ampiezza piccola usando A() e a(). Struttare bene gli esercizi è sempre e comunque molto utile.

Dopo aver fatto la codifica, abbiamo un esercizio sulla checksum molto semplice: se il bit è 1 e la somma degli 1 è un numero pari, allora la sequenza è errata. Altrimenti è giusta.

Dijkstra

Per risolvere questo esercizio, ovviamente dobbiamo tenere a mente come funziona l'algoritmo di dijkstra. Essenzialmente, partiamo da nodo di partenza, e simuliamo di avere una coda priorità, in cui ogni adiacente non esplorato o che è ancora nella queue verrà aggiunto/aggiornato. Poi, al passo successivo prendiamo il nodo con il valore del costo totale più basso. Nel mentre, manteniamo una tabella con un numero di colonne pari al numero dei nodi, e poi scriviamo il nodo “padre” che ci porta a quel nodo. Quando abbiamo finito, dovremo scegliere il nodo con costo minimo. Per ogni colonna (che rappresenta anche il path di costo minimo). E niente, poi è finito sostanzialmente. Possiamo vedere l'algoritmo completo [a questa immagine](#), nella parte sui link state.

Sicurezza e crittazione dei messaggi

Per fare questo tipo di esercizio, è necessario tenere bene a mente le richieste e strutturare per bene la risposta. Siccome ci saranno sempre almeno due scenari da orchestrare, dividi il primo scenario dal secondo. Usa inoltre il linguaggio simbolico (es. Kb- per chiave privata di Bob, Ks per chiave simmetrica, H(m) per messaggio hashato...). Ricorda che puoi anche inviare tutto in una volta, usando per esempio un pacchetto $P = []$.

Dunque, in base alle richieste, dovrà garantire queste cose:

- **Non ripudiabilità (o integrità)**: intende che il messaggio deve avere un Hash. L'hash inoltre garantisce anche se il pacchetto è proprio quello.
- **Autenticazione (o firma, garanzia mittente)**, ovvero essere sicuri che l'abbia mandato proprio Alice o Bob o Claire. Per garantire ciò, basta firmare con la propria chiave privata K- il messaggio (se abbastanza piccolo) oppure l'hash con K-(H(m)) (in particolare se m è grande).
- **Privacy (o confidenzialità)**: non bisogna sapere il contenuto del messaggio. È quindi importante che sia firmato con una chiave simmetrica Ks oppure con la **chiave pubblica K+** della seconda parte (NON CON LA PROPRIA CHIAVE PRIVATA). In caso si usa la chiave simmetrica, RICORDATI DI CRITTARLA CON LA CHIAVE PUBBLICA DELLA SECONDA PARTE!
- **Non replay attack**: per farlo, bisogna prima richiedere un **nonce**, e poi crittarlo (magari con la propria chiave privata per dimostrare che l'hai inviato proprio tu). Il prof ha dato buono anche generare il nonce a caso, senza richiederlo, ma comunque dovrà essere crittato. Se ci sono più segmenti, dovrà garantire il non replay anche in quelli, quindi 2 nonce!

In ogni caso, dai MOLTA importanza all'output finale, e controlla se tutte le richieste sono state rispettate.

Dimensione cwnd, sending rate, valore cwnd per massimizzare prestazioni di rete.

Per fare questo esercizio, dobbiamo tenere a mente alcune delle cose imparate nel capitolo del [controllo della congestione in TCP](#) (ovviamente UDP non possiede alcun tipo di controllo della congestione). Ovvero $\text{rate} = \text{cwnd}/\text{RTT}$. Da qui, possiamo trarre la formula

$$\text{cwnd} = \text{RTT} * \text{rate}$$

Dunque, solitamente il testo indica che un host A e un host B devono usare contemporaneamente un link da un tot di capacità (ad esempio, 64 kB/s) e che un pacchetto ha un certo RTT (supponiamo $\text{RTT} = 250 \text{ ms}$), e che un ogni pacchetto ha un ack che pesa un tot% di un pacchetto (che anche lui ha un valore, solitamente 1kB). Dunque, a questo punto dobbiamo calcolare il valore del cwnd.

Dunque, siccome il link dev'essere diviso in modo equo, supponiamo che entrambi debbano rispettivamente usare al massimo 32 kB/s, che è il nostro valore **rate**. Dopodiché, abbiamo RTT che vale 0.25 s, e quindi possiamo già stabilire che la dimensione del cwnd dovrebbe essere di 8 kB ($32\text{kB/s} * 0.25\text{s}$). C'è però un problema che dobbiamo considerare: ogni pacchetto, che pesa un 1 kB, avrà (se tutto va bene) un ACK, che pesa 12.5% di un pacchetto normale nel nostro esempio. Quindi, allora, avremo un 1kB totale di ACK da usare nella rete. Il motivo per cui consideriamo questi ACK è perché dobbiamo settare la cwnd da entrambi le parti, ovvero sia per A che per B. Quindi, il valore ideale della cwnd sarà 7 kB.

Fusione delle reti/sottoreti

Tipicamente un esercizio di questo tipo richiede il “numero di reti di [classe B](#) che si fondono tra loro se ho netmask /13”, e richiede il numero di reti che si fondono in una super rete. Per quanto sia fuorviante la consegna, risolvere questo esercizio è molto semplice. In pratica, bisogna solo fare la differenza fra una netmask di classe B (ovvero /16) con la netmask data, quindi, $16 - 13 = 3$. Da qui, usiamo il risultato ottenuto come l'esponente di 2, e otterremo $2^3 = 8$ reti che si fondono in una super rete.

Numero di rete, sottereete e numero di host

Per questo esercizio, è necessario prima valutare se il tipo di rete è [di classe A, B o C](#). Dopodiché, bisogna osservare la netmask data. Essenzialmente, bisognerà contare quanti bit della netmask non sono nella parte che definisce la classe di rete. A questo punto, il numero delle reti sarà $2^{(\text{bit contati})}$. Per capire il numero di rete, dobbiamo semplicemente vedere che valore assumono questi bit nel nostro IP dato (magari anche usando una calcolatrice di bit) e quello sarà il nostro network number. Infine, per il numero di host bisognerà fare l'esatta stessa cosa di prima, ma per il numero di host. Quindi osserviamo i valori che assume l'host number nella netmask.

I tre parametri fondamentali di configurazione del protocollo IP

La maschera di rete, l'indirizzo del default router e l'indirizzo IP, costituiscono i tre parametri fondamentali di configurazione del protocollo IP, e devono essere forniti, al momento della connessione, al livello IP.