RETI INFORMATICHE

DOMANDE ORALI CON RISPOSTE

Lorenzo Mancinelli

SOMMARIO

DOMANDE ANASTASI

1 APPLICAZIONI DI RETE

1.1 SISTEMA DNS

Il DNS è un protocollo di **risoluzione degli indirizzi**: offre un mapping che traduce l'hostname nell'indirizzo IP corrispondente.

Mapping: traduzione di indirizzi simbolici nel loro indirizzo IP corrispondente.

Le richieste http verso il server hanno nel messaggio di richiesta l'hostname: sarà il servizio DNS a far arrivare la richiesta nel posto giusto sfruttando il **protocollo di trasporto UDP**.

Si preferisce la reattività all'affidabilità del servizio.

Per richiedere la registrazione di un indirizzo ci si rivolge ad un **register**, che aggiungerà i record necessari al suo database.

I servizi offerti sono quindi:

1. **TRADUZIONE** da hostname ad indirizzo IP.
2. **HOST ALIASING:** definito **nome canonico** l'hostname originale che il DNS mette in correlazione con l'indirizzo IP dell'host, chiameremo **alias** un altro nome mnemonico.

   Quando l'utente digiterà un alias, questo sarà tradotto nel nome canonico e poi sarà il nome canonico ad essere tradotto nell'indirizzo IP.
3. **MAIL SERVER ALIASING:** traduce gli indirizzi mail negli hostname relativi a quell'indirizzo.
4. **DISTRIBUZIONE DI CARICO** che permette di dirottare alcune richieste ad un indirizzo ad un suo **mirror**, ovvero un server identico a quello a cui è stata mandata la richiesta, gestito dalla stessa organizzazione ma su un indirizzo diverso, per non congestionare un solo server.

   Per fare ciò ruota semplicemente la lista di indirizzi IP che corrispondono al nome simbolico cercato, in modo da distribuire le richieste.

Non esiste un solo server DNS, per diversi motivi:

1. **SINGLE POINT OF FAILURE:** se il server fosse solo uno, nel caso in cui crashi, manderebbe in tilt l'intero sistema Internet.
2. **TRAFFICO:** se il server fosse solo uno, avrebbe da gestire le richieste DNS di tutti gli host, che comprendono tutte le richieste http e tutte le richieste a un mail server.
3. **DISTANZA DAL SERVER:** introduce ritardi non indifferenti e possibile congestione dei bottleneck lungo il percorso.
4. **MANUTENZIONE:** essendo un solo server, deve essere molto grande, e più è grande il server, più necessita di manutenzione e di sistemi di aggiornamento.

Abbiamo quindi una **struttura gerarchica:**

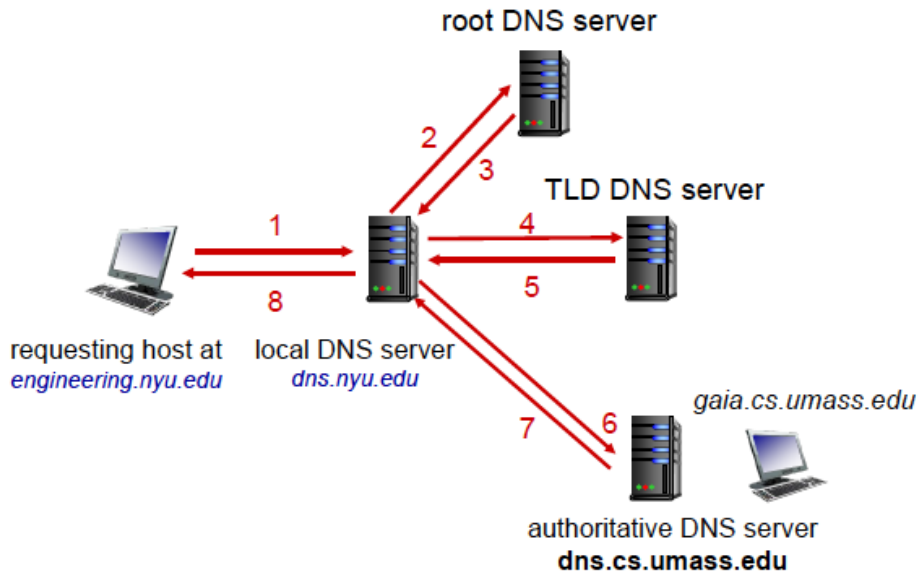
1. **13 ROOT SERVER**, che a loro volta hanno dei server regionali per servire meglio le richieste in giro per il mondo.
2. **TLD SERVER (Top-Level-Domain Server)** che gestiscono tutti i domini di primo livello e quelli relativi ai paesi.
3. **AUTHORATIVE SERVER** che si occupano di risolvere tutti i sottodomini di un dominio e sono comuni all'interno di organizzazioni e server provider.

Nella realtà i client contattano i **local DNS server**, i quali risolvono la richiesta contattando altri server DNS agendo come proxy e implementando un sistema di cache.

La risoluzione può avvenire iterativamente o ricorsivamente.

1.1.1 RISOLUZIONE ITERATIVA

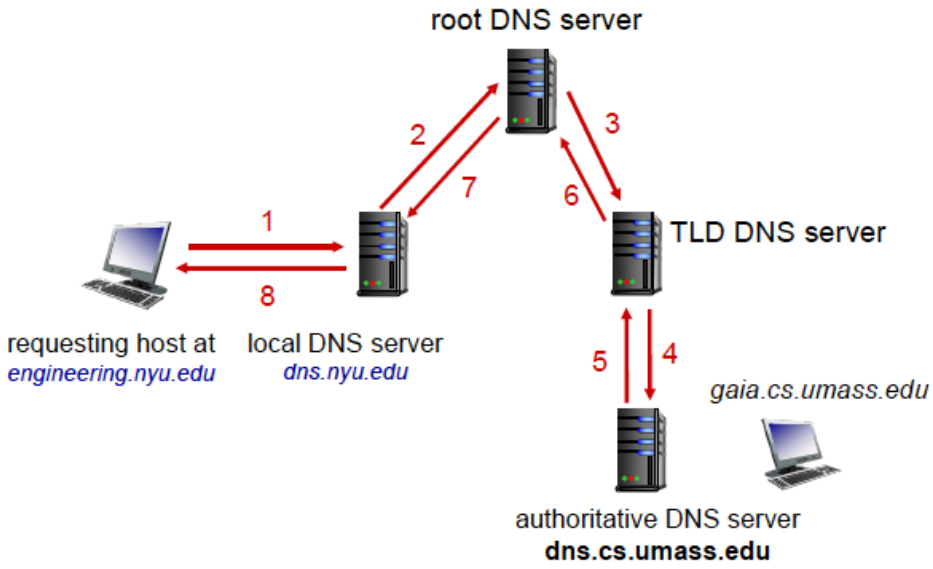
1. Il client chiede al **local server DNS** la risoluzione di un indirizzo.
2. Il local DNS server contatta il **root DNS** per ottenere l'indirizzo del TLD server responsabile del dominio richiesto.
3. Il local DNS contatta il **TLD server** che risponde con l'IP dell'authorative server che gestisce l'indirizzo.
4. Il local DNS server contatta quest'ultimo **authorative server** che risolve infine l'indirizzo richiesto dal client.



1.1.2 RISOLUZIONE RICORSIVA

Le singole richieste non vengono effettuate tutte dal local DNS server, ma dal **server contattato che a sua volta restituisce il risultato a chi ha fatto la richiesta**, fino a tornare al client.

Il carico sui server intermedi aumenta.



Ogni server, compreso l'host stesso, ha la propria cache, la cui vita utile è di solito 2 giorni.

I TLD sono cachati nei local DNS, dato che sono tutte le richieste verso lo stesso TLD DNS server.

La struttura di un **record DNS (RR: Resource Record)** è del tipo

$$< Nome, Value, Type, TTL >$$

TTL è il **Time-To-Live**, il tempo per il quale la entry è valida. Il campo Type assume diversi significati:

- **Type = A** → il campo *Nome* contiene l'hostname e *Value* l'indirizzo IP.
- **Type = NS** → il campo *Nome* contiene il dominio e *Value* l'hostname dell'authorative server per quel dominio.

  I record di tipo NS sono sempre accompagnati da un record di tipo A che traduce l'indirizzo IP del nome dell'authorative server.
- **Type = CNAME** → il campo *Nome* contiene l'alias e *Value* il nome canonico.
- **Type = MX** → il campo *Nome* contiene il dominio e *Value* il nome canonico del mail server associato al dominio.

  Quando ci si riferisce al Web si cercherà tra i record di tipo CNAME, mentre quando ci si vorrà riferire al Mail si cercherà tra i record di tipo MX.

1.2 HTTP

È il protocollo usato per la navigazione sul web, che è un'applicazione **on demand**: tutti vedono quello che vogliono al momento che vogliono.

Ha un approccio **Client-Server**, dove il Client è il browser di un utente che comunica attraverso la porta 80.

- Il Client fa un certo numero di richieste.

- Il Server risponde alle richieste.

Desiderio del Client è quello di ricevere Web File che consistono di oggetti.

Una pagina web è formata da alcuni elementi, viene scaricata quando richiesta e al suo interno avrà dei collegamenti detti URL per accedere alle risorse presenti nel server contattato.

Si appoggia su **TCP**, dunque si deve prima aprire una connessione di questi tipo. Dopodiché il Client formula un messaggio di richiesta, chiedendo cosa vuole, e il Server risponde.

L'oggetto che il client può chiedere è nella forma

$$\boldsymbol{IndirizzoServer \mid PathOggetto}$$

e gli indirizzi del server sono fissati.

Il Client chiede sulla connessione TCP un documento HTML, e dopo che il browser ne fa parsing scopre che ha bisogno di altri oggetti. Allora il Client si mette e comincia a chiedere al Server questi oggetti.

Abbiamo due approcci per le connessioni TCP:

1. **A CONNESSIONE NON PERSISTENTE:** si deve creare una connessione TCP nuova per ogni oggetto che si vuole richiedere.
2. **A CONNESSIONE PERSISTENTE:** si mantiene la connessione aperta per lo scambio di più oggetti.

Facciamo una distinzione pratica tra le due. Chiamiamo **RTT (Round-Trip-Time)** il tempo che impiega un piccolo pacchetto a fare il percorso

$$Client \rightarrow Server \rightarrow Client$$

<u>CONNESSIONE NON PERSISTENTE</u>

1. Il client apre la connessione TCP e il server chiude l'handshake, aprendo ufficialmente la connessione.
   Un piccolo segmento TCP è andato dal client al server e un altro dal server al client: siamo a

   $$1\ RTT$$

2. Il client formula il messaggio di richiesta e il server risponde con il messaggio di risposta più l'oggetto richiesto. Si consuma

   $$1\ RTT + Tempo\ di\ trasmissione\ del\ file$$

3. Il server chiude la connessione TCP solo se si è sicuri che il client abbia effettivamente ricevuto il messaggio. In totale abbiamo usato

   $$\boldsymbol{2\ RTT + Tempo\ di\ trasmissione\ del\ file}$$

   per ogni oggetto richiesto.

CONNESSIONE PERSISTENTE

È solo il primo oggetto a impiegare $2\ RTT + Tempo\ di\ trasmissione\ del\ file$, gli altri impiegano

$$1\ \boldsymbol{RTT + Tempo\ di\ trasmissione\ del\ file}$$

Il protocollo è **stateless**, quindi ogni richiesta è "nuova" e a sé stante.

1.2.1 FORMATO DEL MESSAGGIO HTTP

MESSAGGI DI RICHIESTA

- **REQUEST LINE:** <metodo><URL><Versione HTTP>
  - → GET: richiesta di un oggetto
  - → POST: invio di un oggetto
- **HEADER LINE:** <intestazione>

  Linee che trasmettono informazioni aggiuntive al server:
  - **Host** in cui si trova l'oggetto da recuperare.
  - **Connection:** se al termine della gestione di questa richiesta si può chiudere la connessione TCP.
  - **User-Agent:** browser che si vuole utilizzare per leggere l'oggetto.
  - **Accept-Language:** si comunica al server che si vorrebbe ricevere l'oggetto in una determinata lingua.
- **LINEA VUOTA.**
- **ENTITY BODY:** <corpo>

MESSAGGI DI RISPOSTA

- **STATUS LINE:** <Versione protocollo><Code Status><Status Description>
  - → Risultato dell'operazione (es: error 404)
- **HEADER LINE:** <Keyword><':'><Value>
- **LINEA VUOTA.**
- **ENTITY BODY.**

1.2.2 COOKIE

Metodo utilizzato per rendere **statefull** il protocollo HTTP. Il Server può inviare un cookie al Client, che lo memorizza e lo invia al server con ogni richiesta.

Il cookie si compone di:

- **Header Cookie** nei messaggi di richiesta.
- **Header Set Cookie** nei messaggi di risposta.
- File sul Client che mantiene i cookie in memoria.
- Database sul Server che mantiene i cookie in memoria.

Il meccanismo è il seguente:

1. La prima volta che eseguo una richiesta HTTP su un sito mi viene **associato un ID.**
2. Il Server risponde con un header di tipo **Set Cookie:ID**, dove comunica l'ID al Client. Il browser vede quell'ID e crea una nuova linea nel file Cookie dove annota la coppia

$$Hostname \mid ID$$

3. Alla richiesta successiva il Client invia un header di tipo **Cookie:ID** per far capire al Server che si tratta di lui.
   Il Server risponderà performando azioni rivolte a quel client e alla storia delle sue richieste consultando il back-end DB.

1.3 WEB CACHING (PROXY SERVER)

Un proxy è un **server** che si mette tra client e server e che **memorizza le risorse scaricate**, in modo da poterle servire in caso di richieste future.

Questo permette di ridurre il carico sul server e di velocizzare la navigazione.

Il proxy server si comporta **sia come client**, quando chiede al server se le sue risorse sono aggiornate, **sia come server**, quando serve le risorse ai client.

Vediamo cosa succede:

1. Il client fa una richiesta HTTP. Il messaggio passa prima dal proxy server: la **connessione TCP** viene creata **con il proxy**.
2. Se il proxy ha l'oggetto desiderato, allora è lui che **spedisce l'oggetto al client**.
3. Se il proxy non ha l'oggetto desiderato, **apre una connessione TCP con il server originale** e costruisce un messaggio di richiesta.
4. Il server risponde inviando l'oggetto desiderato. Il proxy server **ne fa una copia e restituisce l'oggetto al client**.

Abbiamo tuttavia un problema: la copia dell'oggetto nel proxy potrebbe non essere aggiornata.

Introduciamo quindi in meccanismo del **conditional GET:** il client sfrutta il fatto che il server risponde con il campo **Last-Modified:** per dire al proxy che vuole una copia dell'oggetto più o meno recente.

- Il client inserisce nella richiesta HTTP un campo

$$If\ Modified\ Since: Data$$

  con cui dice al proxy di mandargli l'oggetto che ha lui, **a meno che il server non gli dica che il file è stato modificato da *Data* in poi.**
- Il proxy manda la richiesta al server, che risponderà sì o no: se risponde positivamente **il proxy farà una nuova richiesta e si farà mandare la copia aggiornata.**

1.4 CLIENT-SERVER (CS) E PEER-TO-PEER (P2P)

1.4.1 DIFFERENZE, PRO E CONTRO, CARATTERISTICHE DEL SERVER E CARATTERISTICHE DEL PEER

Il **paradigma client-server** si serve di una macchina nella quale gira il processo server (detta anch'essa server) che fornisce i servizi ai dispositivi che li richiedono, i client.

I client non possono comunicare tra di loro, e il servizio è erogabile fintantoché il server è **raggiungibile**.

Per questo motivo sono nati i DataCenter, ovvero complessi di macchine server che creano virtualmente una macchina server unica.

Il **paradigma peer-to-pee**r (P2P) invece fa affidamento ad una rete in cui i dispositivi possono fungere sia da client che da server, detti appunto peer, che comunicano tra di loro senza bisogno di un server centrale.

Le principali **differenze** emergono in due aspetti:

1. **SCALABILITA':** un'architettura **P2P** è **più facilmente scalabile**, poiché sarebbe sufficiente aggiungere dei peer alla rete, che sono di fatto dispositivi da normali prestazioni, mentre nell'altro caso aumentare la potenza di calcolo, la memoria di un server o aggiungerne di nuovi può essere impegnativo.
2. **GESTIONE:** i peer non garantiscono la loro presenza costante, una velocità di connessione adeguata e generalmente non implementano misure di sicurezza molto robuste, al contrario dei **server** che invece sono macchine **performanti, sempre online e con importanti misure di sicurezza.**

1.4.2 ESEMPIO DI FILE GRANDE F, CALCOLARE L'ASINTOTO ORIZZONTALE NEL CASO P2P (ANCHE CON SCHEMA DI CONNESSIONE)

Dobbiamo inviare un file di **dimensione F** ad **N dispositivi**. Abbiamo:

- U_s → velocità di upload del server.
- U_i → velocità di upload del peer i-esimo.
- D_i → velocità di download del peer i-esimo.

CLIENT-SERVER

Il tempo impiegato dal **server** per **inviare a tutti i client** è pari a

$$\frac{NF}{U_s}$$

mentre il tempo necessario ad ogni **singolo client** per **scaricare il file** è

$$\frac{F}{D_i}$$

Di conseguenza, il tempo necessario per la **distribuzione del file ad N client** è pari a

$$\max\left\{\frac{NF}{U_s}, \frac{F}{D_i}\right\}$$

che, per $N \to \infty$, è **lineare**.

P2P

Abbiamo:

- $\frac{F}{U_s}$ → tempo che impiega il server a inviare una copia.
- $\frac{F}{\min\{D_i\}}$ → tempo per scaricare il file nel nodo con velocità minore.
- $\frac{NF}{(U_s+\sum_1^N U_i)}$ → tempo per caricare N copie usando upload massimo. Il denominatore può essere assunto come

  $$\mathrm{U_s} + N \cdot U_{medio}$$

  nel caso ideale.

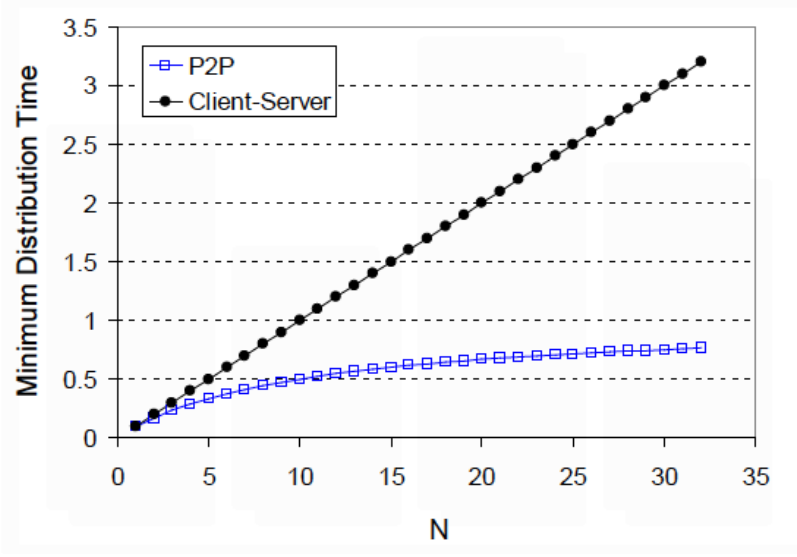
Dunque, il tempo necessario per la **distribuzione del file** è

$$\max\left\{\frac{F}{U_s}, \frac{F}{\min\{D_i\}}, \frac{NF}{(U_s + \sum_1^N U_i)}\right\}$$

che, per $N \to \infty$, è pari a

$$\frac{F}{U_{medio}}$$

Dunque, P2P è **asintoticamente migliore** di CS per l'invio.



1.4.3 IN CHE SENSO NOI INTENDIAMO CHE UN DISPOSITIVO PUO' ESSERE CLIENT O SERVER?

Un dispositivo si dice che è server se in esso gira il processo server, e lo stesso vale per il client.

Nel paradigma peer-to-peer i peer possono fare da server quando devono distribuire un file e da client quando devono richiederlo.

1.4.4 I PEER DEVONO ESSERE SEMPRE ATTIVI?

No, perché non abbiamo un approccio centralizzato: i vari peer si incaricano della distribuzione dei file.

1.4.5 È PIU' VANTAGGIOSO UTILIZZARE UN APPROCCIO CS O P2P PER INVIARE UN FILE? QUAL È L'ISTANTE INIZIALE? E L'ISTANTE FINALE?

Come visto prima, l'approccio client-server fa crescere linearmente, per $N \to \infty$, il tempo di distribuzione di un file ad N dispositivi, mentre l'approccio P2P lo fa crescere asintoticamente.

L'istante iniziale è quando il file viene messo a disposizione, mentre quello finale è quando anche il più lento dei dispositivi lo ha scaricato.

1.4.6 QUANDO N TENDE AD INFINITO COSA SUCCEDE? SIA NEL CASO CLIENT-SERVER CHE NEL CASO PEER-TO-PEER

Nel caso client-server il tempo di trasferimento

$$\max\left\{\frac{NF}{U_s}, \frac{F}{D_i}\right\}$$

cresce linearmente, mentre nel caso peer-to-peer il tempo di trasferimento

$$\max\left\{\frac{F}{U_s}, \frac{F}{\min\{D_i\}}, \frac{NF}{(U_s + \sum_1^N U_i)}\right\}$$

cresce asintoticamente al valore

$$\frac{F}{U_{medio}}$$

1.5 TIPI DI APPROCCIO PER STRUTTURARE IL DATABASE P2P PER DISTRIBUZIONE DI INDICI E CONTENUTI

La filosofia del paradigma P2P è la **struttura decentralizzata**, che potrebbe incontrare delle difficoltà in fase di individuazione delle risorse.

L'idea è quella di creare un database contenente delle tuple dove la **chiave** rappresenta il **contenuto messo a disposizione o da individuare** mentre il **valore** rappresenta **l'indirizzo IP del nodo che possiede tale risorsa**, in modo che i peer possano consultare questo database e richiedere direttamente la risorsa.

1.5.1 INDICE CENTRALIZZATO

Prevede l'esistenza di un **server centrale** che possiede il database e risponda alle query dei peer per la localizzazione delle risorse, mentre lo scambio avviene tra i due peer. Sono presenti alcune criticità:

1. Il server costituisce un **single point of failure** per il servizio di ricerca della risorsa.
2. **Performance** del server che potrebbero essere ridotte, dato che deve rispondere a tutte le richieste dei peer.
3. **Responsabilità legali** del gestore del server.

1.5.2 QUERY FLOODING

È un approccio **completamente decentralizzato**: l'indice è distribuito su tutti i nodi connessi tra di loro a formare un **grafo**.

La richiesta viene quindi forgiata e inviata da un nodo ai suoi adiacenti, che a loro volta la invieranno ai loro adiacenti, per poi tornare indietro una volta che la risorsa è stata individuata.

Il numero di richieste all'interno di una rete è molto elevato: per evitare problemi di prestazioni si può far ricorso alle **limited-scope query**. Questo metodo fa si che la richiesta venga propagata un numero limitato di volte, introducendo però la possibilità di incorrere in falsi negativi.

1.5.3 RICERCA GERARCHICA

Soluzione ibrida tra il query flooding e l'indice centralizzato: sono presenti dei **supernodi**, dei peer con elevata banda e lunghi tempi di permanenza online, che formano una seconda rete tra di loro.

Ogni supernodo possiede inoltre una porzione dell'indice.

Un peer può connettersi ad uno di questi supernodi per informarli del loro contenuto e per effettuare query di ricerca che viene propagata eventualmente all'interno della sottorete.

Una volta ricevuta risposta, viene instaurata una connessione P2P tra i due peer interessati.

1.6 BIT TORRENT

1.6.1 COME FUNZIONA

Il trasferimento di un file avviene grazie alla collaborazione di più peer.

Dividendo il file in **chunk**, l'obiettivo è quello di farsi mandare diversi chunk da diversi peer contemporaneamente in modo da aumentare il tempo di distribuzione del file.

Tutta la comunità che partecipa alla trasmissione del file partecipa a un "torrent di bit".

- **TORRENT SERVER:** mantiene in memoria i file .torrent e permette a chiunque di scaricarli.
- **TRACKER:** server che traccia i peer che fanno parte della rete di distribuzione.
- **SEEDER:** peer che ha terminato il download del file e continua a distribuirlo.
- **LEECHER:** peer che ancora deve completare il download e che contemporaneamente carica i chunk in suo possesso.

Chiamiamo **TAGGED PEER** il peer interessato a ricevere il file:

1. Richiedo al torrent server il .torrent.
2. Richiedo al tracker (il cui indirizzo è nel .torrent) la lista dei peer che stanno partecipando al torrent.
3. Instauro un **vicinato**: instauro connessioni TCP con discreto numero di peer.
   In generale non si comunica con l'intera comunità: chiamo vicini i peer con cui si è instaurata con successo una connessione TCP.
4. Richiedo secondo la politica **rarest first** i chunk che mi mancano e condivido copie dei chunk ricevuti.

Rarest first: si fa un'operazione di richiesta circa i chunk mancanti e si vede quanti peer hanno quel chunk. Si chiede il trasferimento del chunk più raro, perché se i peer che ce l'hanno si disconnettessero, potrei non riuscire ad ottenerlo.

Per decidere a chi mandare i chunk (e quanto frequentemente) si usa la politica **tit-for-tat**: instauro una lista **top4** dei peer che mi inviano più chunk/sec e rivaluto questa lista a intervalli regolari secondo l'**optimistically unchoke.** Invierò i miei chunk ai miei top4.

- **Se sei troppo passivo non ti tornerà niente**: chi decide di non condividere i chunk che ha ottenuto finirà sempre in fondo alla lista dei top4 degli altri peer, e si rallenterà la sua velocità di ricezione.
- **Non possiamo essere sicuri che i top4 siano davvero la scelta migliore**: potrebbe esserci qualche altro peer che potrebbe mandarmi le cose più velocemente, ma non posso scoprirlo perché non gli ho mai inviato chunk.
  A intervalli regolari manderò un chunk a un peer a caso: se questo mi aiuterà a ricevere una risposta da lui, potrò valutare di inserirlo nella mia top4.

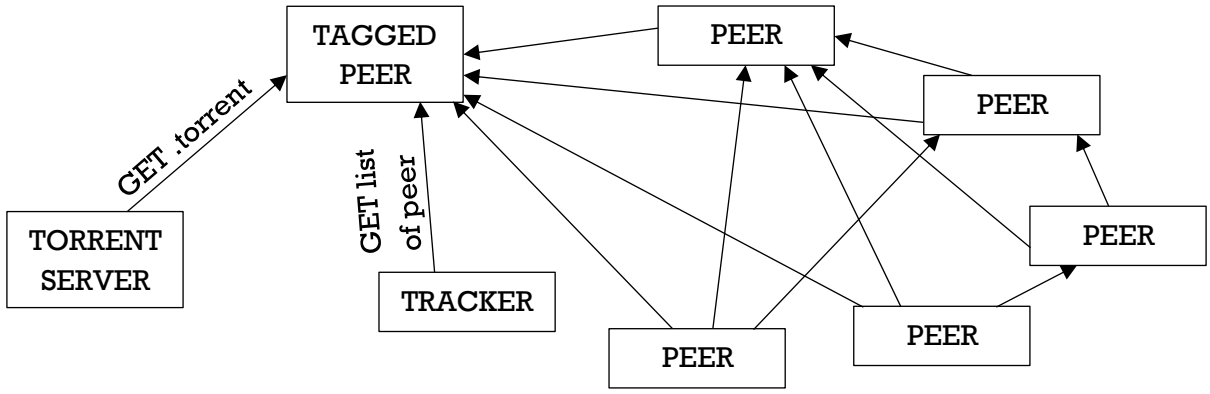
1.6.2 COME SI FA A SAPERE QUALI PEER FANNO PARTE DEL TORRENT

Si chiede al tracker, che tiene traccia di tutti i peer della rete di distribuzione.

1.6.3 COME FA UN PEER PER ENTRARE IN UN TORRENT E SCARICARE UN FILE?

Chiede il .torrent, chiede al tracker una lista dei peer che partecipano al torrent, instaura un vicinato con alcuni di questi e richiede secondo la politica rarest first i chunk che servono al peer per scaricare il file.

1.6.4 DISEGNO ECOSISTEMA BIT TORRENT



1.6.5 SE UNO SI REGISTRA CON IL TORRENT SERVER, COME FA IL TRACKER A SAPERE CHE IL NUOVO PEER SI È REGISTRATO?

Quando un nuovo peer vuole inserirsi nel torrent, richiede innanzitutto l'indirizzo del tracker al torrent server, che sarà contenuto nel file .torrent.

A questo punto, il nuovo peer (che chiameremo Tagged Peer) può contattare il tracker, che gli fornirà la lista dei peer che stanno partecipando al torrent e che aggiungerà il Tagged Peer nella rete di peer che partecipano al torrent.

1.6.6 IN GENERALE, RAPPORTO TRA PEER NUOVO E TRACKER?

Vedi domanda 1.6.5.

1.6.7 IL TAGGED PEER COME MISURA LA VELOCITA' DI UPLOAD DEGLI ALTRI PEER VERSO DI LUI?

Calcola il numero di byte al secondo che un altro peer gli invia.

1.6.8 COME FACCIO A SAPERE QUALI SONO I CHUNK PIU' RARI?

Il Tagged Peer richiede ai vicini la lista dei chunk che hanno a disposizione. Quindi, tra quelli che gli servono per ricevere il file completo, vede quali di questi sono meno presenti nelle liste che gli sono state inviate dai vicini.

1.6.9 COME STILO LA CLASSIFICA DEI PEER?

Ad intervalli regolari di tempo (secondo l'optimistically unchoke) faccio una classifica TOP4 dei vicini che mi inviano più chunk al secondo e, per essere sicuro di non lasciare fuori dalla classifica peer che potrebbero inviarmi chunk più velocemente, ma con cui non ho mai scambiato dati, sempre ad intervalli regolari invierò ad un peer a caso dei chunk, e in base alla sua risposta valuto se inserirlo nella TOP4.

1.6.10 UNA VOLTA FATTA LA CLASSIFICA?

Vedi domanda 1.6.9.

1.6.11 PERCHE' FAVORISCO I PEER CHE HANNO UNA VELOCITA' DI UPLOAD MAGGIORE?

Perché posso ricevere più velocemente i chunk che mi servono per scaricare il file, e a sua volta posso ricondividerli subito, e se un peer ha velocità di upload maggiore potrebbe essere un peer poco attivo, che invia pochi chunk e non partecipa attivamente al torrent.

1.6.12 NOMI DEL MECCANISMO DELLA GRADUATORIA E DELLA SCELTA CASUALE?

Tit-for-tat e optimistically unchoke.

2 DATA LINK LAYER

2.1 IL PROTOCOLLO PPP

È un protocollo del livello Data-Link che permette la comunicazione tra due elementi, un mittente e un destinatario.

Il servizio principale è consegnare un frame garantendo che possa essere usato anche da un piano più alto dello stack protocollare, quindi prevede:

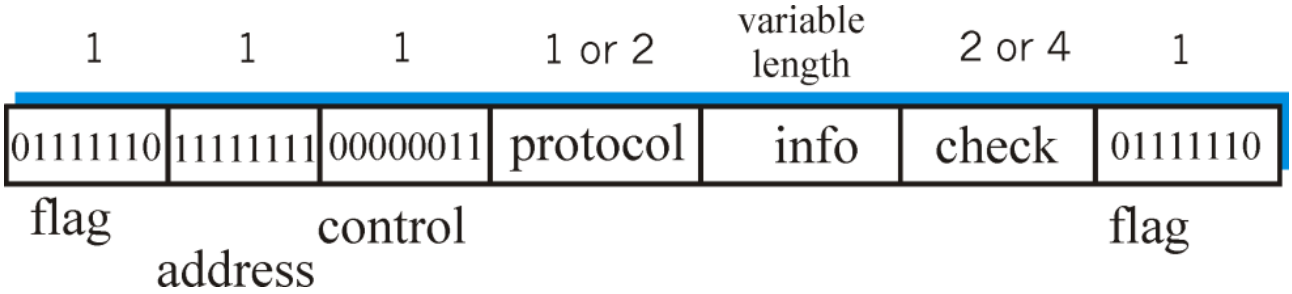
- **Packet framing:** permette di incapsulare i datagram del livello di rete nel livello Data-Link, in modo da implementare tutti i protocolli del livello di rete.
- **Error detection:** per ogni frame include un campo *check* con dei bit CRC per poter permettere al ricevitore di fare error detection.
- **Bit transparency:** è possibile inviare bit arbitrari, senza che questi vengano modificati o male interpretati.
- **Connection liveness:** trova e segnala problemi di connessione tra i vari network layer.
- **Network layer address negotiation:** gli endpoint possono accordarsi su un indirizzo di rete da utilizzare.

Il protocollo non implementa:

- **Error correction.**
- **Flow control:** non è possibile controllare la quantità di dati che possono essere inviati.
- **Error recovery:** non ritrasmette.
- **Point-to-multipoint:** non è possibile inviare dati a più destinatari.
- **In-order delivery:** non è possibile garantire l'ordine di consegna dei dati.

2.1.1 PPP FRAME

È un datagram che viene inviato dal livello di rete al livello Data-Link.



- **Flag:** 8 bit a inizio e fine del datagram che segnalano l'inizio e la fine del frame.
- **Address:** 8 bit per la comunicazione punto-a-multipunto, attualmente non utilizzati.
- **Control:** 8 bit disponibili per utilizzi futuri (campi di controllo), ma attualmente non utilizzati.
- **Protocol:** 8 o 16 bit che contengono l'identificativo del protocollo di livello più alto a cui il frame è destinato.
- **Info:** contiene i dati veri e propri.
- **Check:** contiene il checksum del frame.

2.1.2 BYTESTUFFING

È una tecnica che permette di inserire nel frame dei bit di controllo, senza che questi vengano interpretati come tali. In particolare, questo viene fatto con i bit di flag di inizio e fine frame.

La soluzione è aggiungere un **flag di escape 01111101.** Il trasmettitore fa byte stuffing:

1. **Byte di flag 01111110 → 01111101 01111110.**
2. **Byte di escape 01111101 → 01111101 01111101.**

In questo modo il ricevitore può fare byte unstuffing:

1. **01111101 01111101 → 01111101.**
2. **01111101 01111110 → 01111110.**

Il destinatario rimuove il primo byte e mantiene il secondo.

2.1.3 COME SI NEGOZIANO I CAMPI INUTILIZZATI (LINK CONTROL PROTOCOL)

Utilizzato per permettere agli endpoint di negoziare alcune configurazioni, come

1. Massima dimensione del frame.
2. Un metodo di autenticazione.
3. Scambio e configurazione di indirizzi IP.

In particolare, le prime due configurazioni vengono negoziate tramite il Link Control Protocol (LCP), mentre l'ultimo tramite l'Internet Protocol Control Protocol (IPCP).

2.1.4 A COSA SERVONO I BYTE DI FLAG

Servono a delimitare inizio e fine del frame.

2.1.5 QUALE PROTOCOLLO DI NEGOZIAZIONE DEI PARAMETRI DEL PROTOCOLLO IP SI USA?

Si utilizza l'ICMP.

2.1.6 BIT TRANSPARENCY

È possibile inviare qualsiasi sequenza di byte, senza che venga modificata o male interpretata.

2.2 SLOTTED E UNSLOTTED ALOHA

2.2.1 SLOTTED ALOHA

Il slotted aloha è un protocollo di accesso al mezzo che permette di gestire la collisione tra più trasmissioni.

L'idea alla base è quella di aspettare un tempo casuale quando si fa detection di una collisione: se tutti aspettano un tempo casuale indipendente, c'è la possibilità che alla fine si riesca a trasmettere.

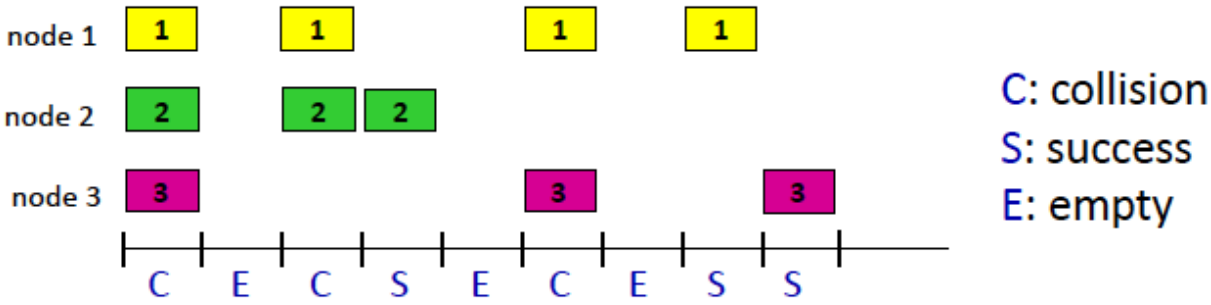
Facciamo alcune premesse:

1. Tutti i nodi devono trasmettere pacchetti della **stessa dimensione L**.
2. Il tempo è **diviso in slot** abbastanza grandi da permettere una trasmissione del pacchetto e la ricezione di un ACK → La collisione è rilevata prima del prossimo slot.

3. Se si riceve un ACK corrotto, o se non si riceve, **si assume conservativamente che ci sia stata una collisione** → Tutti i nodi individuano la collisione.
4. I nodi sono sincronizzati → Tutti sanno quando comincia il prossimo slot.

Il protocollo è implementato nel seguente modo:

1. Il nodo che vuole trasmettere un frame lo fa il prima possibile (nello slot successivo allo slot in cui il pacchetto è pronto).
2. Se uno o più nodi trasmettono contemporaneamente, si verifica una collisione.
3. I nodi che hanno colliso scelgono se provare a reinviare subito il frame secondo una **probabilità *p***, che viene calcolata finché il frame non viene inviato con successo, e ciò per tutti i nodi.



I vantaggi di questo protocollo sono:

1. **Semplicità:** è molto semplice da implementare.
2. **Piena utilizzazione del mezzo:** è possibile utilizzare il mezzo al massimo delle sue capacità.
3. **Alta decentralizzazione:** le uniche informazioni da negoziare sono la durata dello slot e il suo inizio.

Gli svantaggi sono:

1. **Alto numero di collisioni.**
2. Alcuni slot vengono **sprecati**.
3. **Detenction delle collisioni non immediata**.
4. **Sincronizzazione** dei nodi.

L'efficienza di questo protocollo è probabilistica, e dipende dalla probabilità che tutti i nodi inviino correttamente il frame.

Questa probabilità è pari alla probabilità che un nodo invii correttamente il frame (che abbiamo visto essere p) e alla probabilità che nessun nodo trasmetta, e quindi che non ci siano collisioni $(1-p)^{n-1}$. Otteniamo quindi

$$p_{eff} = (Trasmetto) \cdot (Nessun\ altro\ trasmette) = p(1-p)^{n-1}$$

Quanto appena detto vale per **un singolo nodo**. La probabilità che **un qualunque nodo** trasmetta è che succeda quanto detto sopra per ogni nodo:

$$np(1-p)^{n-1}$$

Per avere la **massima efficienza**, devo trovare un p^* che massimizzi questa probabilità:

$$\boldsymbol{np^*(1-p)^{n-1}}$$

Asintoticamente, per $n \to \infty$, otteniamo che la massima efficienza è

$$\boldsymbol{\frac{1}{e} \approx 0.37}$$

Abbiamo quindi il 37% di possibilità che si trasmetta correttamente.

2.2.2 UNSLOTTED ALOHA

Viene eliminata la sincronizzazione tra i nodi: tutti mandano appena il frame raggiunge il data-link layer.

Ciò comporta una maggior probabilità di collisione, in quanto un frame inviato all'istante t_0 può collidere con i frame inviati nella finestra temporale $[t_0 - 1, t_0 + 1]$.

La sua efficienza è quindi:

$$\boldsymbol{p_{eff} = (Trasmetto) \cdot (Nessun\ altro\ trasmette\ prima\ di\ me) \cdot (Nessun\ altro\ trasmette\ dopo\ di\ me) = p(1-p)^{2(n-1)}}$$

Facendo il limite come prima, ottengo un'efficienza peggiore rispetto allo slotted aloha; infatti, ho

$$\boldsymbol{p_{eff} = \frac{1}{2e} \approx 0.18}$$

Ovvero un'efficienza del 18%.

2.3 CSMA/CD

Il CSMA/CD è un protocollo di accesso al mezzo che permette di gestire la collisione tra più trasmissioni, derivata dal CSMA puro.

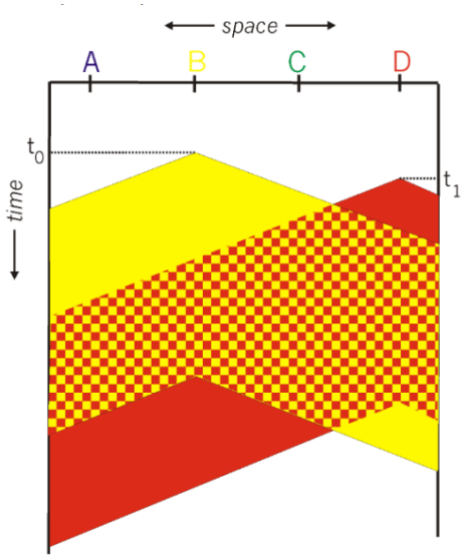
Mentre quest'ultimo inviava tutto il frame se trovava il canale libero, senza alcuna tecnica per la verifica del mezzo durante la trasmissione, il CSMA/CD invia un frame **solo se il canale è libero per tutta la durata della trasmissione.**

Se si verifica una collisione, il canale viene liberato immediatamente in modo da non sprecare tempo.

La detenzione della collisione può essere effettuata confrontando la **potenza ricevuta** con la **potenza emessa:** se ascolto una potenza **maggiore** allora c'è collisione, perché due o più segnali si sono sovrapposti.

Facciamo due esempi, con il CSMA puro e con il CSMA/CD, tenendo conto del **ritardo di propagazione**, ovvero il tempo che impiega un segnale a raggiungere tutti gli altri nodi.

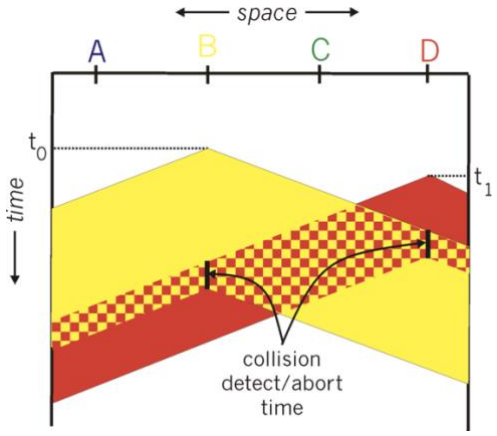
CSMA PURO



Il nodo B inizia a trasmettere all'istante t_0 ma non riesce a raggiungere in tempo il nodo D che, all'istante t_1, vede il mezzo trasmissivo libero.

Dopo un breve tempo i pacchetti B e D faranno interferenza, e si perderà tutto il tempo relativo alla trasmissione dell'intero pacchetto, perché B e D non sono in grado di rilevare la collisione.

CSMA/CD



I due nodi interrompono il prima possibile la trasmissione.

Quando ci si accorge di una collisione, si aumenta la potenza di trasmissione per rendere noto a tutti dell'avvenuta collisione: il segnale ora inviato viene definito **jam signal**.

I vantaggi del CSMA/CD sono:

- Garantisce la **fullfilness** del mezzo.
- Garantisce la **fairness** del mezzo: ogni nodo attende un tempo casuale prima di ritrasmettere, garantendo a tutti la possibilità di ritrasmettere.
- È **completamente decentralizzato.**
- È molto efficiente per **bassi carichi di traffico**, ma non per carichi elevati, in quanto la probabilità di collisione aumenta con il numero di nodi.

2.3.1 PERCHE' SI USA QUESTO RAPPORTO ADATTIVO?

Aumentando la potenza del segnale, tutti i nodi possono rendersi conto che c'è stata una collisione: se non stavano trasmettendo, ora ascoltando il mezzo si troveranno ad ascoltare qualcosa, mentre se stavano trasmettendo ascolteranno un segnale più potente di quello che hanno inviato.

2.4 ETHERNET

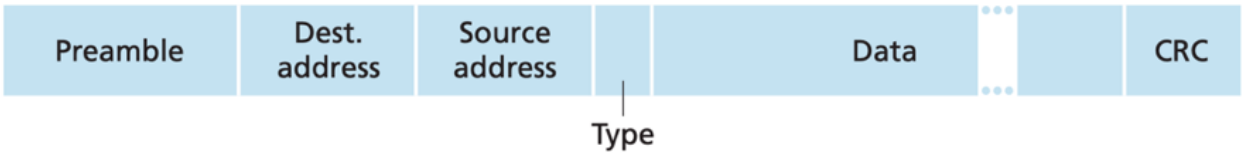
L'Ethernet è un protocollo connectionless di livello 2, non affidabile, che permette di trasmettere frame tra nodi collegati tramite un cavo.

Con il tempo si è evoluto in una struttura a stella che prevede un hub, il cui vantaggio principale era che l'hub è un amplificatore che lavora sul bit. Quando gliene arriva uno, lo amplifica in potenza e lo trasmette su tutte le interfacce di uscita.

Ci sono però delle criticità:

1. L'hub non elimina la possibilità di collisione.
2. L'hub non riesce a rilevare collisione: sarà compito degli end-systems che si collegano all'hub.

2.4.1 FORMATO FRAME ETHERNET



- **PREAMBLE:** sequenza di 8 byte (i primi 7 sono 10101010 e l'ultimo è 10101011) che serve a ricevitore e trasmettitore per sincronizzarsi.
  I due clock hanno un drift, ovvero sono sfasati, ed è possibile che lo sfasamento corrompa i dati al crescere della lunghezza del frame. Con il preambolo il ricevitore riceve informazioni sul clock del trasmettitore.
- **DESTINATION ADDRESS:** è un indirizzo MAC (6 byte) che permette di identificare il destinatario.
  Viene inviato subito dopo il preambolo in modo che i nodi non interessati possano ignorare il frame.
  Se $Destination\ address = Proprio\ MAC\ address$ oppure $Destination\ address = Indirizzo\ MAC\ address$ il MAC accetta il frame.
- **SOURCE ADDRESS:** è un indirizzo MAC (6 byte) che permette di identificare il nodo mittente.
- **TYPE:** 2 byte che indicano il protocollo usato per il livello superiore a cui spedire il datagram.
- **DATA:** contiene il datagram che è stato incapsulato, per il livello superiore. È di lunghezza variabile, e varia da un minimo di 46 byte a un massimo di 1500 byte.
  Se un datagram eccede queste dimensioni, deve essere frammentato in più datagram, mentre se non arriva alle dimensioni minime, devono essere aggiunti dei bit in più, e sarà il livello superiore a fare unstuff di questi byte.
- **CRC:** 4 byte che permettono di fare error detenction, ma non error correction: il ricevitore si limita a buttare via il frame se corrotto.

2.5 CSMA/CD IN ETHERNET

Si compone dei seguenti passaggi:

1. La scheda di rete riceve i dati dal network layer e crea il frame.
2. Se il canale è rilevato come libero per un tempo di 96 bit, allora inizia la trasmissione del frame. Se il canale è occupato, si aspetta che sia libero per 96 bit, e poi si ripete il passaggio 2.
   Il bit time dipende dalla velocità di trasmissione del cavo.
3. Se il frame viene inviato correttamente, allora passo al frame successivo.
4. Se viene rilevata una collisione, si abortisce la trasmissione e si invia un **jam signal**, per un tempo di 48 bit.
5. Dopo aver interrotto una trasmissione, attendo per un **exponential backoff:** all'N-esima collisione si sceglie un K appartenente all'intervallo
   $$\{0,1,2,\dots,2^M-1\}$$
   con $M = \min\{N, 10\}$, e si attende
   $$K \cdot 512\ bit\ (grandezza\ minima\ di\ un\ frame\ ethernet)$$
   prima di tornare al punto 1.

L'exponential backoff è un modo per adattarsi al carico di trasmissione del mezzo:

- Se il carico è alto, si attende più tempo prima di riprovare a trasmettere, in modo da non avere collisioni.
- Se il carico è basso, si attende meno tempo in modo da perderne il meno possibile.

Dopo 17 collisioni, il frame viene comunque scartato.

3 INTERNETWORKING

3.1 ASSEGNAZIONE DI INDIRIZZI IP

3.1.1 INDIRIZZAMENTO NEL PROTOCOLLO IPv4

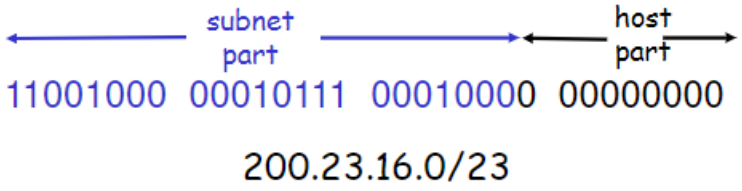
L'organizzazione dell'indirizzo IP segue una **struttura gerarchica**.

Il compito di ogni indirizzo IP è quello di identificare **un'interfaccia** (il meccanismo che collega l'host al livello fisico).

Generalmente un host ha una sola interfaccia, e quindi **l'indirizzo IP identifica l'host**, mentre un router ha un'interfaccia per ogni link a cui è collegato.

Ogni indirizzo IP ha:

1. **Un indirizzo di sottorete** identificabile dal gruppo di bit più significativi.
2. **L'indirizzo dell'host in quella sottorete** identificabile dal gruppo di bit meno significativi.

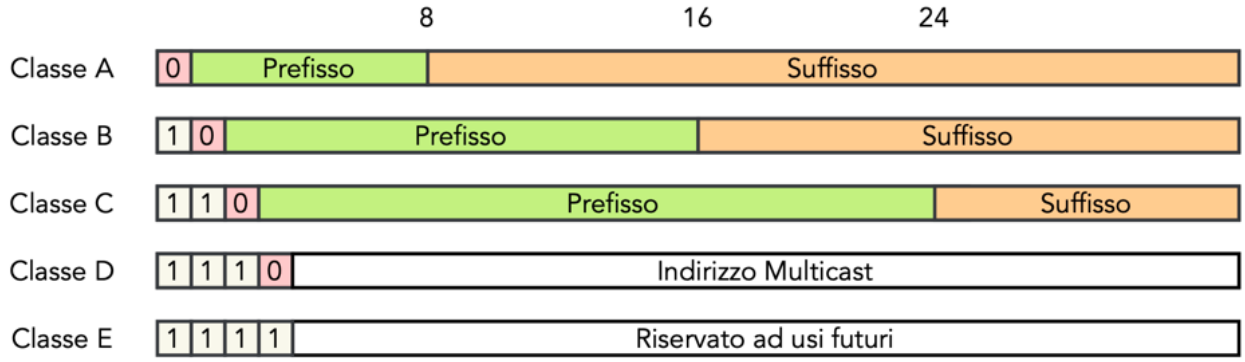


Una sottorete è un insieme di host **localmente collegati**, che possono comunicare direttamente senza l'utilizzo di un router in mezzo.

Quando due host hanno la stessa parte significativa è probabile che appartengano alla stessa sottorete.

Abbiamo due approcci per strutturare un indirizzo IP.

CLASSI DI INDIRIZZI



- **Indirizzi di Classe A:** cominciano tutti per 0 e hanno 8 bit dedicati all'indirizzo di sottorete, e 24 dedicati agli host.
  Costituiscono poche sottoreti con un enorme numero di host.

- **Indirizzi di Classe B:** cominciano tutti per 10 e hanno 16 bit di sottorete e 16 di host.
- **Indirizzi di Classe C:** cominciano tutti per 110 e hanno 24 bit di sottorete e solo 8 bit per gli host.
  Sono dunque tante sottoreti con un piccolo numero di host.
- **Indirizzi di Classe D:** comunicazione multicast.
- **Indirizzi di Classe E:** riservati per un futuro prossimo.

Ci sono delle criticità: ad esempio, se si hanno più di 256 host:

1. Scegliendo tanti spazi di classe C, servirebbe un router per ogni sottorete presa.
2. Scegliendo un unico spazio di classe B, si spreca un enorme numero di indirizzi.

Se tutti scegliessero la seconda opzione, gli spazi di classe B finirebbero rapidamente.

CIDR: CLASSLESS INTERDOMAIN ROUTING

- Si alloca un **indirizzo di sottorete a lunghezza variabile**.
- L'indirizzo IP si scrive nella forma A.B.C.D/X dove X rappresenta il numero di bit dedicati alla sottorete.

Dobbiamo tenere conto del fatto che ci sono degli indirizzi riservati:

- L'indirizzo a **tutti 0** rappresenta un nodo che **non ha indirizzo**.
- Gli indirizzi che cominciano con l'indirizzo di sottorete e poi sono **o tutti 0 o tutti 1** rappresentano l'indirizzo della sottorete e **l'indirizzo di broadcast della sottorete**.
- L'indirizzo a **tutti 1** rappresenta **l'indirizzo di broadcast ristretto**, mediante il quale il datagram è spedito a tutti i nodi della rete locale a cui il nodo sorgente appartiene.
- L'indirizzo di **sottorete 127** è riservato per fare **loopback** quando si sviluppano applicazioni distribuite: il pacchetto è rimandato allo stesso host.

QUALE È IL VANTAGGIO DI ORGANIZZARE UNA INTER-RETE IN SOTTORETI?

Snellire le tabelle di forwarding dei router che stanno nel core: se tante sottoreti fanno riferimento a sottoreti più grandi:

1. I router dell'Internet interno conserveranno nelle loro tabelle di forwarding solo un'entrata per tutti gli indirizzi IP dell'ISP.
2. È poi il router dell'ISP che dovrà contenere 8 record, quindi **sposto la complessità esternamente**.

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

Se un'organizzazione conosce il suo personale /X, può risalire all'indirizzo della sua sottorete guardando semplicemente il suo indirizzo IP, e conoscere anche la sottorete dell'ISP conoscendo lo /X dell'ISP.

Però, come fa il singolo host a ricevere il suo indirizzo IP?

3.1.2 DIFFERENZA TRA ASSEGNAZIONE STATICA E DINAMICA

ASSEGNAZIONE STATICA: una soluzione è **fissare permanentemente** un indirizzo IP a un host.

Serve qualcuno che conosca quali indirizzi IP sono disponibili ad essere assegnati.

Un enorme **svantaggio** è che non ha senso che sia riservato un indirizzo IP anche quando un host non è permanentemente 24/7 connesso in rete.

ASSEGNAZIONE DINAMICA: nasce l'esigenza di **ricevere un indirizzo IP al bisogno e restituirlo quando non si usa più**.

Questo meccanismo sfrutta un'applicazione client-server: il client effettua una richiesta di indirizzo IP e un server risponde.

Il protocollo che regola questo meccanismo è il protocollo DHCP (Dynamic Host Configuration Protocol).

3.1.3 VANTAGGI DELL'ASSEGNAZIONE DINAMICA

1. Riutilizzo degli indirizzi IP.
2. Comodità nell'assegnazione automatica.
3. Cambio di rete implica cambio di indirizzo (non lo potrei fare con assegnazione statica).

3.2 PROTOCOLLO DHCP

Il protocollo DHCP è un protocollo applicativo di livello 5 (applicazione) **basato su UDP** che permette agli host di ricevere **dinamicamente** un indirizzo IP.

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

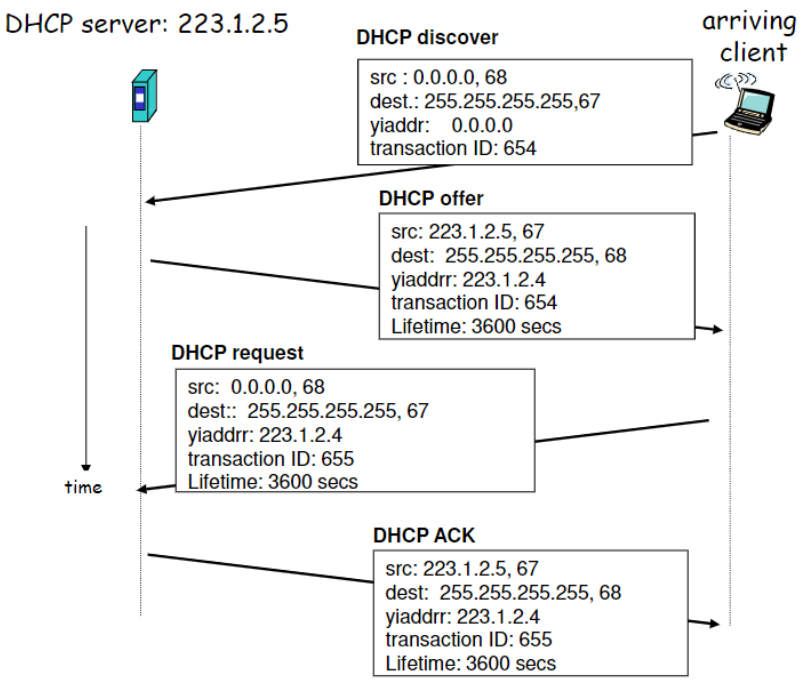
Ipotizziamo che nella subnet ci sia almeno un server DHCP e di non sapere quanti ce ne siano effettivamente.

Il sistema viene realizzato tramite il **four-way-handshake**, ovvero vengono inviati in tutto 4 messaggi:

1. **DHCP DISCOVER:** inviato dal client (che vuole ricevere un indirizzo IP) dalla porta 68 in broadcast alla porta 67. Il messaggio sarà composto da:
    a. *Source* – 0.0.0.0:68, per rappresentare il fatto che è senza indirizzo IP e che sta inviando dalla porta 68.
    b. *Destination* – 255.255.255.255:67, comunica in broadcast sulla porta 67 con la rete locale a cui è connesso.
    c. *yiaddr* – "your internet address", è il campo che sarà riempito con l'IP offerto (inizialmente posto a 0.0.0.0).
    d. *Transaction ID* – generato casualmente, è l'ID della richiesta.

Dato che è possibile avere più server DHCP nella stessa rete, inviando i messaggi in broadcast tutti questi server possono vedere la richiesta, offrire un proprio indirizzo ed eventualmente procedere con il protocollo.

2. **DHCP OFFER:** il DCHP server che è in ascolto sulla porta 67 riceve la richiesta e rimanda **in broadcast** un IP disponibile.
   In questo modo anche il client che ha fatto richiesta, che ancora non possiede un indirizzo, è in grado di ricevere il messaggio monitorando tale porta.
   Stavolta il campo *yiaddr* contiene un indirizzo, che è quello che il server DHCP sta offrendo, e il campo *Transaction ID* è uguale al precedente, in modo che il client capisca che si sta parlando con lui.
   Si introduce anche un *lifetime* che contiene la durata temporale dell'associazione host-IP.
3. **DHCP REQUEST:** il client comunica **in broadcast** che vuole accettare l'offerta dell'IP che specifica nel campo *yiaddr*.
   Source conterrà 0.0.0.0:68, *Destination* conterrà 255.255.255.255:67 e *yiaddr*, appunto, conterrà l'indirizzo che ha ricevuto e che sta accettando.
   Riporterà inoltre il campo *lifetime* e il *Transaction ID* incrementato di 1.
   Perché la comunicazione avviene in broadcast? In questo modo tutti i server DHCP riceveranno *yiaddr*, e se vedono che *yiaddr* è diverso da quello da loro proposto capiranno che il client ha accettato la richiesta di un altro server.
4. **DHCP ACK:** il DCHP client comunica la buona riuscita del protocollo.
   Se questo messaggio viene ricevuto correttamente, l'IP è consegnato e può essere utilizzato per un *lifetime*.



Il campo *lifetime* viene utilizzato per permettere il recupero, e quindi il riutilizzo, degli indirizzi non più utilizzati.

Prima della scadenza un host può "rinnovare" la propria presenza e mantenere l'assegnazione host-IP. In caso contrario, tale indirizzo potrà nuovamente essere riassegnato, il tutto senza bisogno che il client avvisi il server della disconnessione.

Oltre all'indirizzo IP, il protocollo si occupa anche di configurare tutto il necessario per la connessione, come il server DNS, il gateway e la subnet mask.

3.2.1 COME AVVIENE LA COMUNICAZIONE? IL PROTOCOLLO UDP

La comunicazione si basa sul **protocollo UDP**. Il protocollo UDP non estende la qualità del servizio offerta da Internet; infatti, l'unico servizio che offre è quello necessario al livello di trasporto, ovvero multiplexing e demultiplexing.

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

I vantaggi dell'UDP rispetto al TCP sono che è più veloce ed economico, dato non c'è bisogno di perdere tempo a stabilire connessioni point-to-point e non è neanche necessario conservare informazioni di stato grosse quanto il TCP.

In generale, UDP è utilizzabile da quelle applicazioni che richiedono **rate sensitivity** (vogliono una risposta velocemente) e sono **loss tollerant** (non si aspettano il 100% dell'affidabilità).

Andiamo a vederne il funzionamento: in generale, il compito del livello di trasporto è quello di instaurare una connessione logica tra due host che vogliono comunicare.

- Sulla sending-side si prende il messaggio di livello applicazione e si divide in **segmenti**.
- Questi segmenti sono poi riassemblati nel messaggio originale nella receiving-side del livello di trasporto.

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

MULTIPLEXING: consiste nel raggruppare dati da socket diversi in un unico flusso di segmenti.

Ogni segmento deve però essere incapsulato mediante un header che fornisce informazioni per le operazioni che devono essere fatte quando viene raggiunto l'host destinatario.

DEMULTIPLEXING: con questa operazione l'host destinatario decapsula il segmento e mette i dati nel socket giusto.

Si inseriscono nell'header del segmento informazioni sulla porta del processo mittente e sulla porta del processo destinatario.

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

Se due IP sorgenti diversi mandano segmenti allo stesso IP destinatario alla stessa porta, il segmento arriva allo stesso socket.

Il socket UDP è identificato da una coppia $(ID\ Destinatario, Porta\ Destinatario)$.

3.3 NAT

3.3.1 A COSA SERVE?

Il NAT (Network Address Translation) è un meccanismo implementato nei router che permette di usare **un solo IP pubblico per un'intera sottorete di host privati**, e dunque per tutti i dispositivi ad essa connessi, fungendo da intermediario tra una rete locale e Internet.

3.3.2 PER QUALE SCOPO È STATO PENSATO?

Lo scopo principale è quello di **ridurre il numero di indirizzi IP necessari** per far comunicare host e Internet esterno.

3.3.3 COME SI IMPLEMENTA? DOVE SI TROVA IL NAT SERVER? DISEGNO DI COSA SUCCEDE QUANDO SI UTILIZZA IL NAT

Immaginiamo di avere una rete composta da alcuni dispositivi che posseggono tutti un proprio **indirizzo privato**, quindi valido solo all'interno della propria rete.

La comunicazione tra tali dispositivi non rappresenta un problema, facendo parte della stessa rete, ma la situazione cambia quando devono comunicare con dispositivi al di fuori della sottorete: se il router vede un indirizzo **non privato** come destinatario del pacchetto, si rifiuta di instradarlo.

L'idea è quindi quella di implementare un NAT **all'interno del router di frontiera delle reti locali**.

Il NAT in sostanza, quando un client vuole inviare un datagram (specificando l'indirizzo destinatario pubblico e la porta sul quale è in ascolto), modifica l'indirizzo privato sorgente di questo client con **quello del NAT stesso**, ed eventualmente anche la **porta**, creando una entry in una speciale tabella di traduzione, detta **NAT Translation table**, all'interno della quale associa la tupla

$$(\boldsymbol{IP\ Privato\ Host} \mid \boldsymbol{Porta\ Host})$$

alla tupla

$$(\boldsymbol{IP\ Pubblico\ NAT} \mid \boldsymbol{Porta\ Router})$$

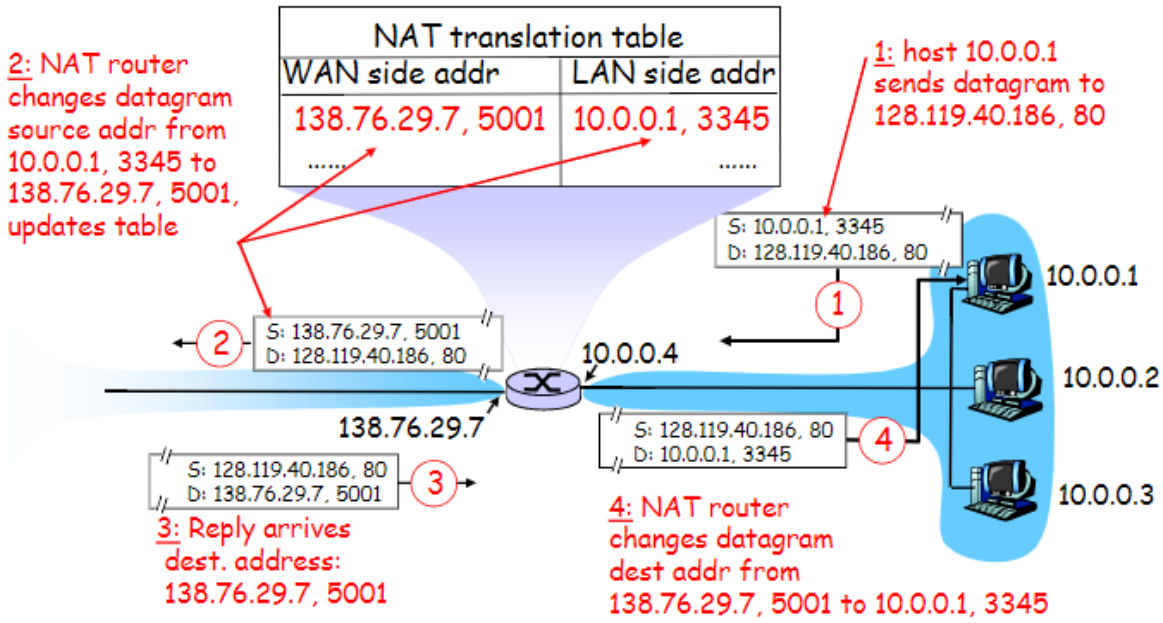
L'entry avrà quindi la forma

$$(IP\ Pubblico\ NAT \mid Porta\ Router - IP\ Privato\ Host \mid Porta\ Host)$$

Quando il server pubblico risponde, risponde inserendo come destinatario l'IP pubblico del NAT e la porta indicata dal NAT stesso.

In questo modo, quando il router riceve un pacchetto dall'esterno su una determinata porta, potrà consultare la tabella, verificare se il pacchetto è destinato ad un host sulla rete privata (ovvero se nella tabella c'è una corrispondenza), e modificherà eventualmente l'IP e la porta

con quelle corrispondenti nella tabella di traduzione, per poi instradare correttamente il pacchetto nella rete locale.



3.3.4 SITUAZIONE PARTICOLARE IN CUI QUESTO MECCANISMO POTREBBE NON BASTARE E COME SI RISOLVE

Se la richiesta parte dall'esterno, con destinatario un server NATato della rete locale, potrebbe mancare l'associazione porta-IP privato, dunque il NAT non riesce a spedire nulla al richiedente.

Si potrebbe risolvere inserendo manualmente, a priori, dei record nella tabella, in modo che il NAT possa redigere il traffico correttamente sul server che ha IP privato.

Il comando per poter aggiungere tale regola è iptables.

3.3.5 VANTAGGI E LIMITI

I principali vantaggi sono:

1. **Economicità:** consente l'utilizzo di un solo indirizzo IP pubblico per un'intera sottorete, indirizzando un numero di host pari al numero di porte possibili.
2. **Sicurezza:** gli indirizzi privati vengono nascosti da quello pubblico condiviso dalla rete. Questo permette di non instradare pacchetti direttamente verso gli host; quindi, è più difficile per eventuali attacchi individuare e attaccare i dispositivi nella sottorete.
3. **Scalabilità:** l'aggiunta di nuovi dispositivi è semplice, non richiede sforzi onerosi e utilizza protocolli già eventualmente implementati, come il DHCP, che **non va in conflitto con il NAT**.

I principali limiti sono:

1. **Difficoltà in connessioni P2P:** potrebbero non essere presenti entry nella tabella di traduzione nel caso di connessioni "in entrata". Una soluzione potrebbe essere quella vista prima, di creare manualmente una entry permanente nella tabella, oppure sfruttare un ibrido CS/P2P dove il server fornisce ai client tutte le informazioni per instaurare una connessione P2P.

2. **Numero di porte limitato:** esse sono poco più di 60.000, quindi in ogni caso non sarà possibile avere più di quel numero di connessioni simultanee dalla stessa sottorete.
3. **Il numero di porta non dovrebbe servire a identificare gli host, ma i processi all'interno degli host.**
4. **Layer di lavoro:** di fatto il NAT lavora nel router, che è un dispositivo di livello 3, e va a modificare informazioni del layer di livello 4, violando la connessione end-to-end. Questo di fatto non è uno svantaggio, infatti non presenta potenziali problemi in fase di utilizzo, ma risulta essere una controversia riguardante i vari layer e la loro "autonomia". La soluzione a tale controversia sarebbe l'implementazione del protocollo IPv6.

4 TRANSPORT LAYER

4.1 TCP

Il protocollo TCP (**Transmission Control Protocol**) ha le seguenti caratteristiche:

1. Il collegamento che si crea richiama quello **point-to-point**, come se i due host fossero direttamente collegati e l'unico servizio da realizzare fosse quello di mux/demux.
2. Il protocollo è **connection-oriented:** c'è una fase di handshaking chiamata triplice handshake che deve precedere la fase di trasmissione dei dati.
3. La trasmissione è **full duplex:** A può trasmettere a B e viceversa contemporaneamente.
4. Implementa un **reliable data transfer** basato sul concetto di ACK, Retrasmissione e timeout.
5. Sono allocati sia sul mittente che sul destinatario dei **buffer** di dimensione variabile grazie ai quali è possibile bufferizzare pacchetti di cui non si può ancora procedere all'invio sul livello applicazione.
   I messaggi dell'applicazione non possono essere mandati così come sono ma vanno spezzati secondo un **MSS (Maximum Segment Size)**. Tipicamente, essendo l'header TCP+IP di 40 byte, un MSS è 1460 byte, per rispettare l'MTU di 1500 byte.
6. Permette **flow control:** c'è rallentamento sulla trasmissione quando questa avviene troppo più velocemente della ricezione dei messaggi.
   Se il buffer del ricevitore è pieno si vuole impedire che segmenti vengano inviati e subiscano **buffer overflow**.
7. C'è **controllo di congestione:** si riducono o si aumentano le finestre dei vari buffer in base alla congestione sulla rete (gestione dinamica di una pipeline).

4.1.1 FORMATO DEL FRAME TCP

PRIMA RIGA: <source port | dest port>

- Source port: numero di **porta del sorgente**, 16 bit.
- Dest port: numero di **porta del destinatario**, 16 bit.

SECONDA RIGA: <sequence number>

- Numero di sequenza: usato per dare un ordine ai pacchetti, il valore iniziale è scelto all'apertura della connessione ed è la **posizione del primo byte del segmento di dati nello stream**, 32 bit.

TERZA RIGA: <acknowledgement number>

- Usato per confermare la ricezione dei pacchetti, è il numero di sequenza relativo al **prossimo byte che si aspetta di ricevere**.
- Significa che ho ricevuto correttamente **fino all'ultimo byte del campo data del segmento precedente**.
- È valido solo quando il flag ACK è impostato a 1.
- 32 bit.

QUARTA RIGA: <header lenght | reserved | flag | receive window>

- **Header lenght:** lunghezza dell'intestazione del segmento, 4 bit.
- **Reserved:** riservato per utilizzi futuri, 4 bit a 0.
- **Flag:** descrive alcuni bit importanti, quali
  - **ACK:** acknowledgement, 1 bit.
  - **URG:** urgente, 1 bit.
  - **PSH:** push, 1 bit.
  - **RST:** reset, 1 bit.
  - **SYN:** synchronize, 1 bit.
  - **FIN:** finish, 1 bit.
- **Receive window:** ci dice, in caso di ACK, quanti byte il ricevitore può ancora accettare, 16 bit.

QUINTA RIGA: <checksum | urg data pointer>

- **Checksum:** permette l'error detection, 16 bit.
- **Urg data pointer:** serve a marchiare un certo byte come "punto di inizio di una sezione urgente", 16 bit.

SESTA RIGA: <options>

- Lunghezza variabile, poco usato.

SETTIMA RIGA: <application data>

- Dati da trasferire, lunghezza variabile da 20 a 60 byte.

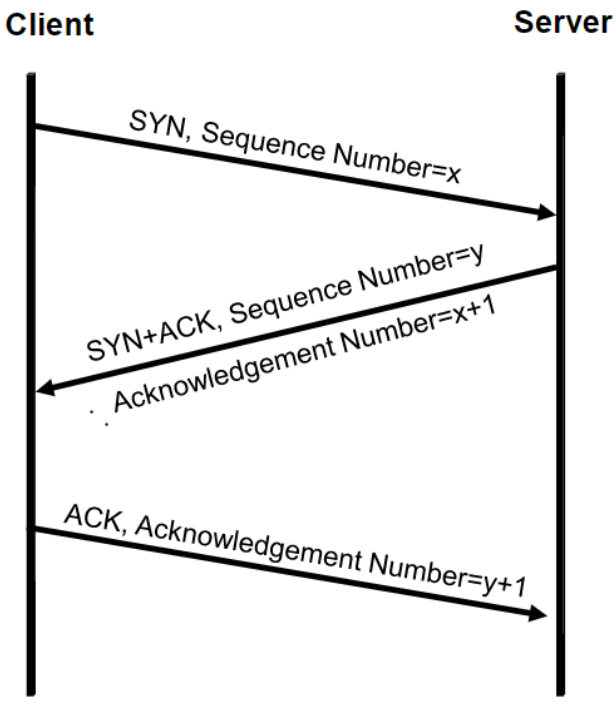
4.1.2 APERTURA/CHIUSURA CONNESSIONE TCP

APERTURA CONNESSIONE TCP: durante la fase di connessione vengono create e impostate tutte le strutture dati necessarie alla connessione TCP, usando ad esempio le primitive di sistema, con le impostazioni desiderate.

Il **3-way handshake** si compone delle seguenti fasi:

1. Il **client** invia un pacchetto TCP con flag SYN impostato a 1 (per dire che vuole fare setup della connessione) e comunica il suo numero di sequenza iniziale.
   Non vengono inviati dati.
2. Il **server** risponde con un pacchetto TCP con i flag SYN e ACK impostati a 1 (per dire che ha capito che stanno stabilendo una connessione). Anche lui dovrà comunicare il suo numero di sequenza iniziale.
   Il campo ACK contiene il numero di sequenza del pacchetto ricevuto dal client + 1.
   In questa fase il server alloca i buffer, e non vengono inviati dati.
3. Il **client** riceve il segmento SYN-ACK e risponde con il solo ACK attivo e con il numero di ACK pari al numero di sequenza ricevuto dal server + 1.
   La connessione è stabilita e il client può anche inserire dei dati in quest'ultimo segmento per cominciare la comunicazione.

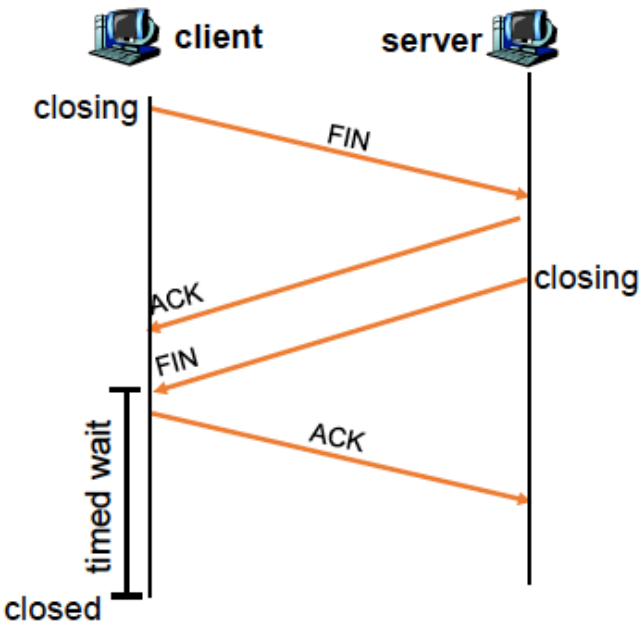
I numeri di sequenza vengono generati **casualmente** per evitare che il client, connettendosi, chiudendo la connessione e riaprendola immediatamente, possa ritrovarsi dati di una connessione precedente nei buffer.



CHIUSURA DELLA CONNESSIONE TCP: avviene in tali fasi.

1. Il **client** invia un pacchetto TCP con il flag impostato a 1, e il numero di sequenza a caso. Non vengono inviati dati.
2. Il **server** risponde con ACK attivo ed eventualmente i dati che deve ancora inviare. Successivamente chiude la connessione e invia un pacchetto TCP con il flag FIN impostato a 1.
3. Il **client**, ricevuto quest'ultimo pacchetto, risponde con ACK attivo, e chiude la connessione.

Se il FIN del server non riceve risposta, lo stesso provvede a rinviarlo: sia il client che il server rimangono dunque in attesa della conferma della chiusura della connessione.



PERCHE' IL NUMERO DI SEQUENZA INIZIALE È CASUALE? Per evitare che il client si connetta, chiuda la connessione e poi la riapra subito dopo ed allora nei buffer interni vi rimangano ancora vecchi dati.

COSA SUCCEDEREBBE SE INVECE VENISSERO USATI GLI STESSI NUMERI DI SEQUENZA? Se un numero di sequenza non venisse inviato nella "vecchia" connessione, potrebbe essere interpretato come pacchetto dati nella "nuova" connessione.

USO DEL TIMER NELLA CHIUSURA DELLA CONNESSIONE TCP: il client aspetta una **timed wait** prima di chiudere definitivamente la connessione. Durante questa attesa manda l'ACK al FIN del server.

4.1.3 AFFIDABILITA' PROTOCOLLO TCP: COME VIENE GARANTITA

Il TCP crea un servizio affidabile usando uno schema ARQ: ACK, Retransmission e Timeout. Quest'ultimo elemento è il più complesso da considerare, perché ovviamente non è possibile calcolarlo a priori, specialmente se esso dipende dai router intermedi che sono fuori dal controllo degli host.

Ci si basa dunque su delle **stime** che considerano l'RTT degli ultimi pacchetti inviati, e l'ACK relativo agli stessi. Si usa, in sostanza, una media esponenziale mobile:

$SampleRTT \coloneqq RTT$

$EstimateRTT \coloneqq ERTT \qquad \alpha < 1$

$ERTT_1 = RTT_0$

$ERTT_2 = \alpha \cdot RTT_1 + (1-\alpha) \cdot RTT_0$

$ERTT_3 = \alpha \cdot RTT_2 + \alpha(1-\alpha) \cdot RTT_1 + (1-\alpha)^2 \cdot RTT_0$

...

$ERTT_{n+1} = \alpha \cdot RTT_n + \alpha(1-\alpha) \cdot RTT_{n-1} + \alpha(1-\alpha)^2 \cdot RTT_{n-2} + \cdots + (1-\alpha)^n \cdot RTT_0$

Raccogliendo, otteniamo

$$ERTT_{n+1} = \alpha \cdot RTT_n + (1-\alpha) \cdot [\alpha \cdot RTT_{n-1} + \alpha(1-\alpha) \cdot RTT_{n-2} + \cdots + (1-\alpha)^{n-1} \cdot RTT_0] \Rightarrow$$

$$\boldsymbol{ERTT_{n+1} = \alpha \cdot RTT_n + (1-\alpha) \cdot ERTT_n}$$

dove:

- RTT_n è il Round Trip Time **misurato** del pacchetto n.
- $ERTT_n$ è il Round Trip Time **stimato** del pacchetto n.

Facciamo alcune osservazioni su α, in base al quale decidiamo se dare più importanza alle misurazioni recenti o alle misurazioni passate.

- $\alpha = 0 \rightarrow$ la stima è immutata e coincide con la stima iniziale.
- $\alpha = 1 \rightarrow$ la stima successiva coincide con la misurazione immediatamente precedente.

Tipicamente si usa $\alpha = 0.125$, dando maggiore importanza alle misurazioni recenti.

4.1.4 TIMEOUT SU RETI TCP

COME SI CALCOLA: innanzitutto, dobbiamo considerare che **la stima del RTT non deve essere calcolata sui pacchetti che sono in ritrasmissione**.

- Se calcolassimo la stima e arrivasse un ACK per quel segmento, non riusciremmo a capire se l'ACK si riferisce al segmento mandato ora o quello mandato all'inizio.

Abbiamo quindi due algoritmi per calcolare il timeout:

1. **ALGORITMO DI KARN-PARTRIDGE:** calcolata una stima del RTT con la media esponenziale mobile abbiamo

   $$\boldsymbol{Timeout = 2 \cdot ERTT}$$

   I pacchetti trasferiti ma persi vengono esclusi dal calcolo.
2. **ALGORITMO DI VAN JACOBSON-KAREL:** vogliamo aggiungere un fattore additivo, che si trova andando a studiare la **deviazione del RTT**, ovvero una proprietà di scostamento del valore reale rispetto alla stima effettuata sullo stesso.

   $$\boldsymbol{DevRTT_{n+1} = (1-\beta) \cdot DevRTT_n + \beta \cdot |ERTT_n - RTT_n|}$$

   Scegliendo un opportuno valore di β possiamo dare più importanza alla deviazione immediatamente precedente o alla storia della stessa.
   Tipicamente si sceglie $\beta = 0.25$.
   Otteniamo quindi

   $$\boldsymbol{Timeout = ERTT + 4 \cdot DevRTT}$$

Chiediamoci ora: come il sender utilizza questa informazione? Quanti timeout ci sono? Come si reagisce a un timeout?

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

PERCHE' NON SI TIENE CONTO DEI PACCHETTI PERDUTI?

La stima del RTT non deve essere calcolata sui pacchetti che sono in ritrasmissione, perché se calcolassimo la stima e arrivasse un ACK per quel segmento, non riusciremmo a capire se l'ACK si riferisce al segmento mandato ora o quello mandato all'inizio.

4.1.5 TCP SEMPLIFICATO

Ignoriamo per un attimo la possibilità del TCP di poter controllare il flusso e gestire gli ACK duplicati per vedere una versione semplificata del protocollo. Il trasmittente deve essere in grado di:

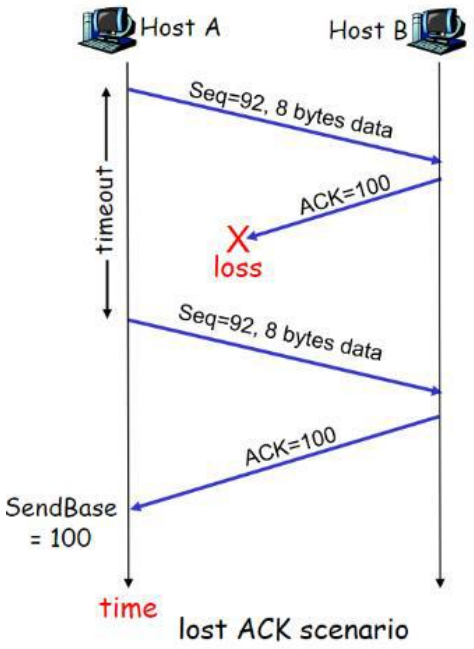
1. **Ricevere i dati dall'applicazione**, creare un segmento TCP con numero di sequenza opportuno, farlo scorrere in base alla lunghezza del messaggio e, **se non c'è già un timer attivo**, far partire il timer appena prima di inviare il segmento.
   Il timer attivo a un certo istante si riferisce al **segmento più antico di cui non c'è ancora ACK.**
2. **Segnalare il timeout**, e dunque ritrasmettere il segmento che non ha ACK con numero di sequenza più piccolo.
   Gli ACK sono cumulativi, quindi se non si facesse così un ACK potrebbe far intendere che sono state capite cose antiche che in realtà non sono state capite.
3. **Ricevere l'ACK**, verificare a quale pacchetto fa riferimento e gestire timer e/o ritrasmissione di conseguenza.
   Ogni trasmettitore ha una **SendBase**, ovvero l'indice del primo byte di cui non si è ancora ricevuto ACK.
   SendBase – 1 rappresenta dunque l'ultimo byte di cui è stato fatto ACK.
   Se Y (valore del segmento) < SendBase, allora l'ACK è ridondante e non succede nulla.
   Se Y = SendBase non succede nulla comunque, perché quel byte è in invio ed è quello a cui il timer fa riferimento.
   Se Y > SendBase, allora significa che posso porre Y = SendBase per via dell'ACK cumulativo.
   Inoltre, faccio ripartire il timer se ci sono segmenti not-ACKed da inviare o inviati senza ACK ricevuti.

Quindi, abbiamo diversi scenari:

1. L'host A vuole inviare 8 byte di dati all'host B. Il numero di sequenza utilizzato è 92. L'host B invia ACK=100 perché ha ricevuto correttamente gli 8 byte [92,99] e il prossimo che si aspetta è il byte 100.

   L'ACK viene perduto: il timeout scade e A ritrasmette. Il segmento arriva all'host B anche questa volta, però l'host B, vedendo un numero di sequenza minore rispetto a quello che si aspetta, rispedisce ACK=100 e scarta il segmento.

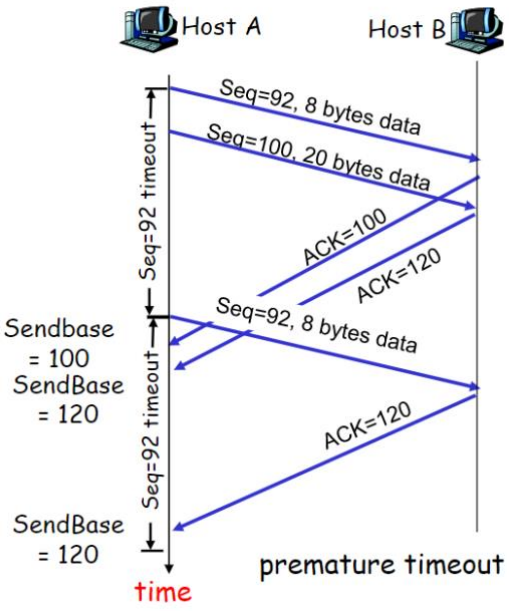
   Quando l'host A riceve successivamente l'ACK, vede che 100>92 e dunque avanza SendBase fino al numero di ACK ricevuto. Il primo byte di cui non ha ancora l'ACK diventerà dunque il byte con numero di sequenza 100.



2. L'host A invia due segmenti. Partendo dal numero di sequenza 92, il secondo ha numero di sequenza 100.

   I segmenti sono correttamente ricevuti da B, che invia due ACK, uno 100 per la ricezione del primo segmento e uno 120 per la ricezione del secondo segmento.

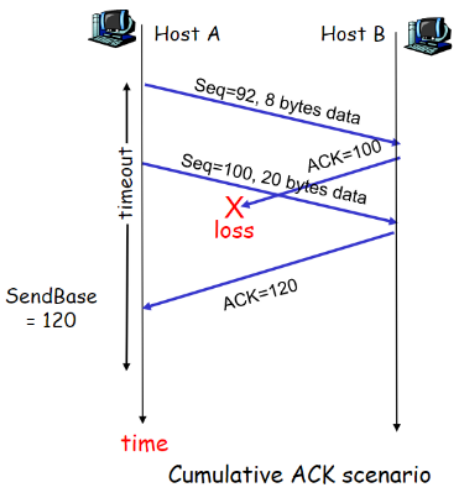
   Il timeout scatta prematuramente prima della ricezione di questi ACK: parte la retrasmissione del segmento più vecchio in attesa di ACK, con numero di sequenza 92, e riparte il timer.



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

3. L'host A invia gli stessi due segmenti, ma stavolta vediamo la potenza dell'**ACK cumulativo.**

   Essendo i segmenti correttamente ricevuti, vengono spediti due ACK. Il primo ACK si perde, mentre il secondo arriva. Non ci sarà nessuna ritrasmissione, perché ACK=120 confermerà la ricezione di entrambi i segmenti.



Cumulative ACK scenario

4.1.6 FAST RETRANSMISSION

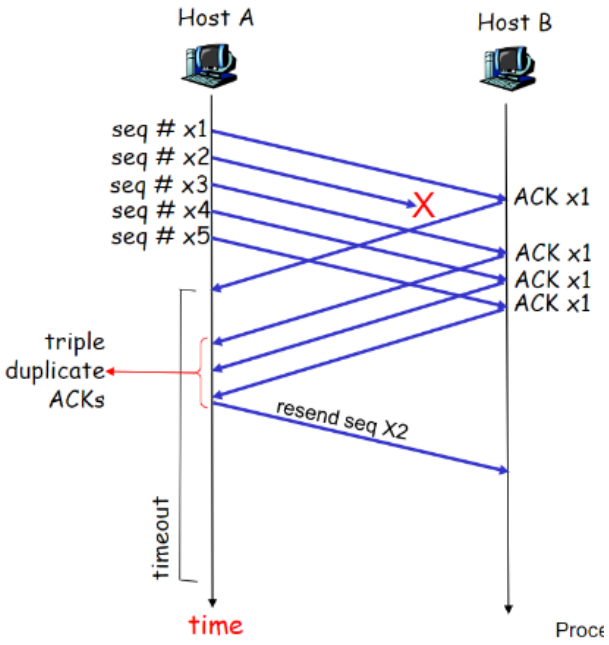
In caso di ACK cumulativi di cui però uno è mancante, allora possiamo implementare una politica che ci permette di reinviare il pacchetto mancante, senza attendere il timeout.

Questo avviene in particolare alla ricezione di 3 ACK duplicati (ovvero 4 ACK dello stesso segmento).

Vediamo come fa a generarsi un ACK duplicato: quando il trasmettitore invia una serie di segmenti il ricevitore potrebbe non riceverne qualcuno.

Ma se ne perde qualcuno, non è detto che gli altri si perdano. Questo crea il fenomeno della **ricezione fuori ordine**, in cui il ricevitore riceve dei segmenti che non si aspettava, perché non corrispondono al punto in cui è arrivato.

Il ricevitore rimanda quindi un ACK relativo all'ultimo segmento in ordine che gli è arrivato.



Il ricevitore potrebbe comunque bufferizzare i segmenti fuori ordine e spedirli "in burst" a livello applicazione quando fa fill del gap che ha rilevato.

Ricapitolando, ci sono 4 diversi scenari per il ricevitore:

1. Arrivo di un segmento con numero di sequenza uguale a quello atteso, con tutti i segmenti precedenti ACKati. Attende 500ms per eventuali altri pacchetti in arrivo e invia ACK cumulativo.
2. Arrivo di un segmento con numero di sequenza uguale a quello atteso, ma con segmenti precedenti non ACKati. Invia ACK cumulativo senza attendere i 500ms.
3. Arrivo di un segmento out-of-order. Invia ACK duplicato indicando il numero di sequenza atteso, senza aspettare i 500ms.
4. Arrivo di un segmento che riempie un gap formato da un segmento del tipo precedente. Invia ACK cumulativo senza aspettare i 500ms.

4.1.7 CONTROLLO DI FLUSSO NEL PROTOCOLLO TCP: A COSA SERVE

Il controllo di flusso consiste nell'evitare che **il buffer del ricevitore si riempia** e che si creino segmenti che si buttano per colpa del trasmettitore che invia troppo velocemente rispetto a quanto il ricevitore riesca a leggere dal suo buffer.

Per inviare informazioni al trasmettitore, il ricevitore ha a disposizione il campo **receive window**, con cui cerca di comunicare lo spazio disponibile nel suo buffer.

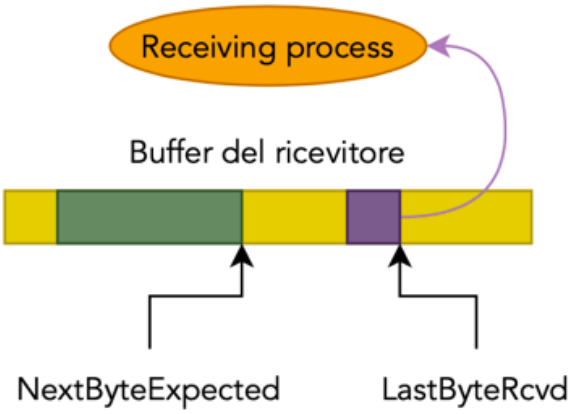
RICEVITORE: mantiene in memoria

- ***RcvBuffer:*** dimensione del buffer.
- ***LastByteRead:*** ultimo byte che è stato letto dall'applicazione e dunque "cancellato" dal buffer (o comunque sovrascrivibile).
- ***LastByteRcvd:*** ultimo byte che è stato ricevuto e al momento è bloccato nel buffer.
- ***NextByteExpected:*** indice dell'eventuale buco mancante in caso di ricezione out-of-order.

All'interno del buffer avremo un po' di segmenti memorizzati, un eventuale gap (dovuto ai segmenti out-of-order) e altri byte ricevuti: lo spazio libero sarà dato da

$$RcvBuffer - OccupiedSpace$$

Il protocollo **non** considera come spazio libero eventuali buchi causati dalla ricezione out-of-order (tale gap prima o poi sarà colmato da segmenti che potrebbero essere stati inviati).



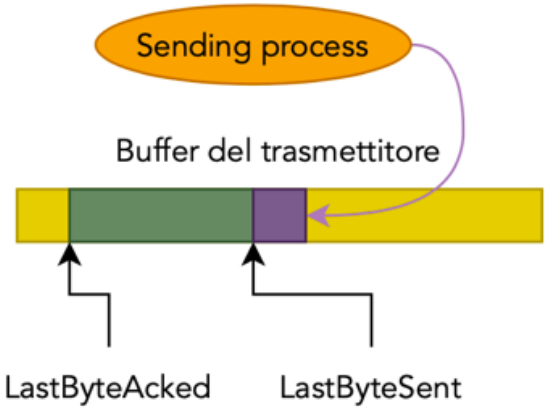
Possiamo quindi concludere che la dimensione libera, e quindi la finestra annunciata, sarà

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

Tale valore si inserisce nel campo *ReceiveWindow* nell'header TCP.

TRASMETTITORE: mantiene in memoria

- Dimensione del buffer.
- ***LastByteAcked:*** ultimo byte per cui è stato ricevuto l'ACK.
- ***LastByteSent:*** ultimo byte che è stato inviato ma di cui non si è ricevuto l'ACK; dunque, che non si troverà nel buffer del ricevitore.



Le due variabili di stato sono fondamentali perché non è detto che si possa inviare esattamente tanti byte quanti indicati dalla finestra annunciata.

Il trasmettitore riceverà il feedback della situazione corrente dal ricevitore, che gli comunicherà la dimensione libera del suo buffer tramite il campo *ReceiveWindow* del segmento TCP, e invierà una quantità di dati pari a

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

Se il buffer di ricezione si riempie, il trasmettitore **smette di inviare dati** e invia periodicamente un segmento con un byte di dati, finché il ricevitore non ha spazio libero, in modo che il ricevitore possa aggiornarlo della situazione corrente.

Se infatti il trasmettitore non gli inviasse tale messaggio, non potrebbe conoscere la situazione corrente e continuerebbe a trasmettere dati, causando una perdita degli stessi.

4.1.8 CONGESTIONE TCP

COS'E' LA CONGESTIONE: è un fenomeno che si verifica quando tante sorgenti mandano velocemente diversi pacchetti sul network e si cominciano a riempire le code dei router.

COSA COMPORTA PER GLI HOST?

1. Se le code si riempiono, si forma il fenomeno di **packet loss** a livello network.
2. Se le code non si riempiono, aumenta significativamente il queueing delay, che rappresenta la maggior parte del ritardo end-to-end se la comunicazione coinvolge il network.

COME E DOVE SI VERIFICA? Si verifica nei router intermedi che smistano il traffico.

COME SI GESTISCE? Per risolvere la congestione ci sono due approcci:

1. **Network-assisted:** i router forniscono feedback al trasmettitore, che può quindi adattare la sua trasmissione.
   Il feedback può essere di due tipi:
   a. Semplice: un solo bit a segnalare la presenza o meno di congestione.
   b. Articolato: indica anche il rate al quale regolarsi.
2. **End-to-end:** il trasmettitore adatta la sua trasmissione solo quando si hanno perdite di pacchetti e rallentamenti.
   Questo approccio conservativo è quello adottato dal TCP.

END-TO-END CONGESTION CONTROL IN TCP: l'obiettivo della tecnica è fare in modo che tutte le sorgenti inviino dati il più velocemente possibile, senza che la rete si congestioni.

Per limitare il rate di trasmissione possiamo diminuire il numero di byte inviati nella stessa finestra temporale, abbassando il numero di segmenti non ACKati ad un valore detto ***cwnd*** **(congestion window)** tale che

$$LastByteSent - LastByteAcked \leq cwnd$$

Il valore è in realtà il minimo tra la dimensione della *congestion window* e la dimensione della *receiver window*. Possiamo dunque affermare che il *cwnd* è un valore dinamico che cambia a seconda della congestione della rete.

Il TCP, per accorgersi della presenza di una congestione, fa uso di ACK e dei segmenti perduti:

1. Se ho ottenuto ACK allora non c'è congestione, e posso continuare ad inviare a un ritmo sostenuto, provando anche ad aumentarlo.
2. Se ci sono segmenti persi, si assume che ciò sia dovuto alla congestione della rete e quindi si abbassa il ritmo di trasmissione.

Ricordiamo che la perdita di pacchetti può avvenire in due casi:

1. **Si è verificato un timeout:** sembra davvero che il pacchetto si sia perso nei vari router. Non è sicurezza di congestione, perché il timeout potrebbe essere semplicemente prematuro.
2. **Si è verificato un triplice ACK duplicato:** anche questo non è sicurezza di congestione, perché il pacchetto potrebbe semplicemente star sperimentando un ritardo nei router e gli sono arrivati prima altri segmenti. Un fenomeno del genere è sinonimo di **congestione debole**, in quanto significa che al ricevitore qualcosa sta arrivando, ma non è ancora arrivato il prossimo in-order segment.

L'algoritmo di congestione ha come idea quella di fare **probing** della bandwidth:

- Quando **ricevo ACK**, incremento il rateo della trasmissione.
- Quando **perdo segmenti**, abbasso il rateo della trasmissione.

L'intero protocollo si divide in 3 fasi:

1. **SLOW START:** pongo inizialmente

$$\boldsymbol{cwnd = MSS\ (MaxSegmentSize)}$$

e dunque posso trasmettere a

$$\boldsymbol{rate = \frac{MSS}{RTT}}$$

Il rate viene raddoppiato ad ogni ACK ricevuto, quindi **raddoppio la mia finestra di congestione** con una crescita esponenziale.
Quando la congestione sarà rilevata, il trasmettitore imposta un valore di soglia detto **treshold**, impostato inizialmente a

$$\frac{cwnd}{2}$$

Supponiamo ad esempio che venga rilevata congestione e, dopo aver impostato il treshold, si abbassi cwnd a 1MSS. D'ora in poi, ogni volta che verrà superata tale soglia si passerà alla fase di congestion avoidance.

2. **CONGESTION AVOIDANCE:** si incrementa linearmente il *cwnd* ad ogni ACK, fino a che non si verifica una congestione. Abbiamo due casi possibili di congestione, uno più grave e uno meno grave:
   a. 3 ACK duplicati (meno grave): impostiamo
   $$\boldsymbol{threshold = \frac{cwnd}{2}}$$
   e passiamo alla fase successiva di **fast recovery**.
   b. Timeout (più grave): impostiamo
   $$\boldsymbol{cwnd = 1\ MSS}$$
   $$\boldsymbol{threshold = \frac{cwnd}{2}}$$
   e ripartiamo dalla fase di slow start.
3. **FAST RECOVERY:** a seconda della causa che ha determinato l'entrata in questa fase, e dal tipo di TCP usato, vengono scelti dei valori diversi per il *cwnd*.
   a. La causa è la ricezione di 3 ACK duplicati: il valore di *cwnd* viene impostato a

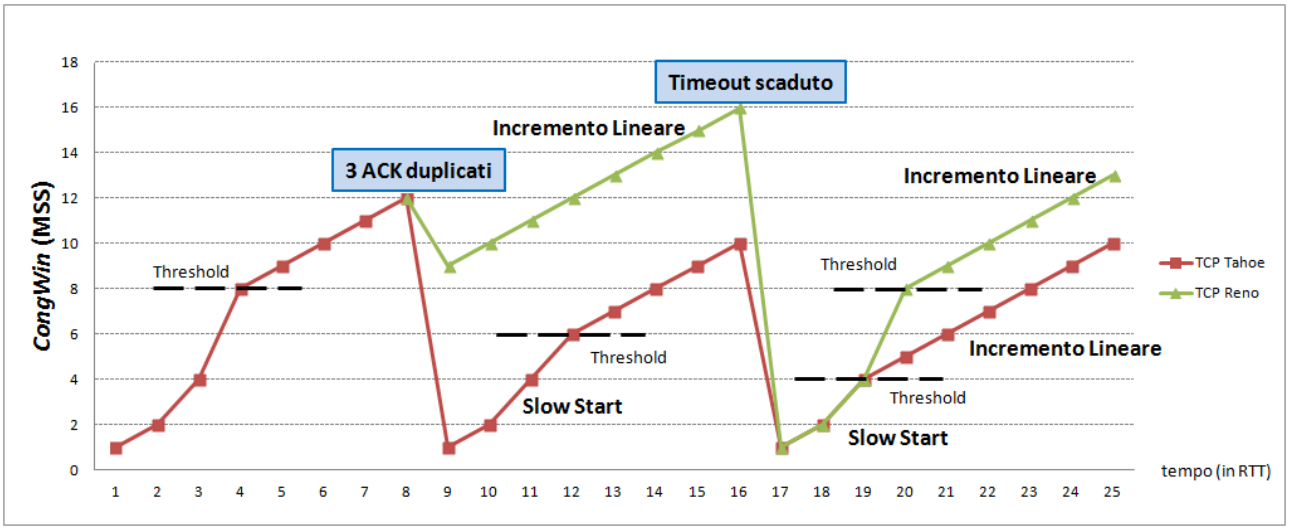
$$\boldsymbol{cwnd = \frac{cwnd}{2} + 3 \cdot MSS}$$

e si procede aumentando linearmente *cwnd*.

b. La causa è la perdita dedotta da timeout: il valore di *cwnd* viene impostato pari a 1MSS e si riprende con la slow start.

In questo modo è possibile distinguere quando si è in *fast recovery* per una perdita o per un ACK duplicato e agire di conseguenza, senza abbassare il rate anche se non ce n'è effettivamente bisogno.

Vediamo un esempio grafico.



Siano TCP Tahoe una versione del TCP che non implementa fast recovery e sia invece TCP Reno una versione che lo implementa:

- All'inizio, ad ogni ricezione di ACK viene raddoppiato il valore di *cwnd*.
- Raggiunto *threshold*, si inizia la fase di congestion avoidance e la crescita di *cwnd* diventa lineare.
- A $cwnd = 12MSS$ si rileva un triplice ACK duplicato:
  - In entrambi i casi, la soglia si abbassa a
    $$threshold = \frac{cwnd}{2} = \frac{12MSS}{2} = 6MSS$$
  - **TCP Tahoe:** si abbatte *cwnd* a 1MSS e si riprende con la fase di slow start.
  - **TCP Reno:** si pone
    $$cwnd = threshold + 3MSS = 6MSS + 3MSS = 9MSS$$
    e si riparte con una crescita lineare.
- Dopo un po' si verificherà una condizione di timeout scaduto: il comportamento dei due TCP è lo stesso per quando riguarda *cwnd*, in quanto essendo congestione grave si riparte da 1MSS in entrambi i casi, tuttavia il *threshold* sarà diverso, perché la *cwnd* al momento del timeout era diversa nei due casi.

4.1.9 DIFFERENZA TRA CONTROLLO DI FLUSSO E CONTROLLO DI CONGESTIONE

Il controllo di flusso avviene tra trasmettitore e ricevitore, ed è un'accortezza che il trasmettitore prende quando il buffer del ricevitore rischia di traboccare.

Il controllo di congestione invece riguarda i router intermedi, ed è un controllo che si fa quando è il percorso end-to-end ad essere intasato.

4.1.10 PERCHE' SI FA UN TRATTAMENTO DIVERSO TRA IL TIMEOUT E IL TRIPLICE ACK?

Nel caso di triplice ACK abbiamo una congestione molto meno grave, perché per mandare ACK duplicati significa che qualcosa al ricevitore sta arrivando.

5 SECURITY

5.1 MAC (MESSAGE AUTHENTICATION CODE)

5.1.1 A COSA SERVE

È un meccanismo usato per garantire l'**integrità** di un messaggio.

Quando si parla di integrità, vogliamo che siano rispettate alcune proprietà:

1. Vogliamo essere sicuri che il messaggio sia **inviato da chi pensiamo noi**.
2. Vogliamo che il messaggio non sia un **playback** (ovvero che viene inviato ripetutamente).
3. Vogliamo che il messaggio sia a tutti gli effetti **integro**.
4. Vogliamo che la sequenza dei messaggi sia **la stessa che ha mandato il mittente**, e non una sequenza alterata da un intruso.

5.1.2 PROPRIETA' SUE E DELLA FUNZIONE HASH UTILIZZATA

Il MAC è un valore, chiamato **digest**, che viene aggiunto al messaggio e che viene calcolato in base al messaggio stesso, ad una chiave segreta e ad una funzione hash.

- Il ricevente, per verificare l'integrità del messaggio, calcola il MAC del messaggio ricevuto e lo confronta con il MAC ricevuto. Se i due valori coincidono, allora il messaggio è integro.
- In questo modo, essendo il segreto condiviso da solo due persone, viene garantita anche l'autenticità del messaggio, in quanto siamo sicuri che il destinatario sia chi affermi di essere.
- Il messaggio **non** viene però criptato, quindi non viene garantita la confidenzialità dei messaggi: è tuttavia possibile abbinare a questa soluzione un algoritmo a chiave simmetrica che permette di criptare il messaggio stesso, e risultare del tutto trasparente al meccanismo del MAC.

  Una sua applicazione è quella detta HMAC (Hash-based Message Authentication Code), che è un MAC basato su SHA-1 e MD5.

Vediamo ora le proprietà della funzione hash, che preso in input un messaggio m restituisce una stringa di lunghezza fissata $H(m)$.

- La funzione H **deve essere una funzione da calcolare**, che non richieda troppe risorse computazionali.
- **Irreversibile:** dato un digest $H(m)$, non bisogna essere in grado di risalire al messaggio originale m.
- **Resistenza alle collisioni:** vogliamo che sia computazionalmente difficile trovare un messaggio m' **diverso da** m tale che risulti

$$H(m) = H(m')$$

  Ovviamente non possiamo rendere questa situazione impossibile, perché la funzione hash è many-to-one e quindi c'è la possibilità di collisioni.

5.1.3 COSA SIGNIFICA CHE UN MESSAGGIO È CORROTTO

Un messaggio è corrotto se, ricalcolando il MAC del messaggio, questo è diverso dal MAC ricevuto.

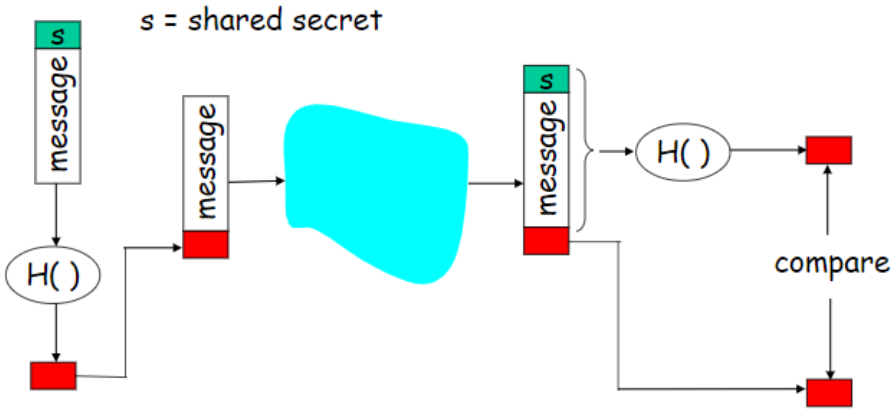
5.1.4 QUALI ALTRI SERVIZI, OLTRE ALL'INTEGRITA', OFFRE?

Vedi domanda 5.1.2.

5.1.5 COME SI IMPLEMENTA?

Il MAC viene calcolato tramite i seguenti passaggi:

1. Il mittente prepara il messaggio m e lo concatena alla chiave segreta s.
2. Calcola un digest risolvendo
$$H(s \mid m)$$
3. Concateno il risultato al messaggio e invio la stringa risultante (che quindi non avrà al suo interno la chiave segreta)
$$[m \mid H(s \mid m)]$$
4. Il ricevente, conoscendo la chiave segreta, può a sua volta calcolarsi il digest della combinazione ricevuta:
$$H(s \mid m)$$
5. Se il digest da lui calcolato è uguale a quello che gli è stato mandato, può concludere che m non si è alterato e che il mittente sia proprio quello che si aspetta, dato che solo lui conosce la chiave segreta s.



5.1.6 PREVENIRE RECORD & PLAYBACK

Il **record & playback** è un attacco che consiste nel registrare un messaggio e riprodurlo in seguito, per poi inviarlo al destinatario.

Il problema è che il messaggio viene inviato con la stessa data e ora di quando viene registrato; quindi, il destinatario non può capire se il messaggio è stato registrato e riprodotto o se è stato inviato in tempo reale.

Per ovviare a questo problema, si può aggiungere al messaggio una data e un'ora di invio, che viene calcolata dal mittente e che viene aggiunta al messaggio prima di calcolare il MAC. In questo modo può verificare se il messaggio è stato inviato in tempo reale (decidendo anche un'eventuale tolleranza) o se è stato registrato e riprodotto.

In alternativa, si può aggiungere al messaggio un **nonce**, ovvero un numero casuale che viene generato dal mittente, sempre diverso per ogni messaggio, e che viene aggiunto al messaggio

prima di calcolare il MAC. In questo modo non si possono usare messaggi registrati in precedenza, perché il nonce è diverso per ogni messaggio.

5.2 FIRMA DIGITALE

5.2.1 PROPRIETA'

Si tratta del corrispondente digitale della firma cartacea. Deve **necessariamente** rispettare tutte queste proprietà:

1. **Verificabilità:** chi riceve il messaggio deve poter verificare che sia stato firmato dal mittente.
2. **Non ripudiabilità:** il mittente non può negare di aver di aver firmato il messaggio.
3. **Non falsificabilità:** nessuno può firmare al posto di un altro.
4. **Integrità del messaggio:** il messaggio non può essere alterato senza che la firma venga invalidata.

5.2.2 PERCHE' IL MAC NON È DEFINIBILE UNA FIRMA DIGITALE?

Il MAC non può essere usato come meccanismo di firma digitale perché la chiave segreta è conosciuta da almeno due persone, e non si potrebbe accertare la paternità di una firma, in quanto più persone hanno a disposizione la chiave. Facciamo un esempio: Bob vuole inviare un messaggio M ad Alice:

1. **Bob può dimostrare che lui ha firmato il messaggio M e non il messaggio M'?** Sì, perché il messaggio M' non restituirebbe $H(s \mid M)$.
2. **Alice può verificare che è stato Bob a firmare il messaggio M, e nessun altro?** No, perché la chiave segreta è condivisa ed esistono almeno 2 persone che possono generare quella firma.
3. **Bob può dimostrare che la sua firma è stata falsificata?** No, perché almeno due persone possono generare la stessa e identica firma.
4. **Alice può dimostrare che Bob ha firmato M e non M'?** No, perché potrebbe essere stata la stessa Alice a firmare M' e apparirebbe come Bob.

Viene quindi usato un algoritmo a chiave pubblica, per permettere ai destinatari di verificare l'autore della firma, che ha ovviamente firmato con la sua chiave segreta che appartiene a lui e a nessun altro ed è quindi verificabile.

5.2.3 IMPLEMENTAZIONE DI UN PROTOCOLLO DI FIRMA DIGITALE

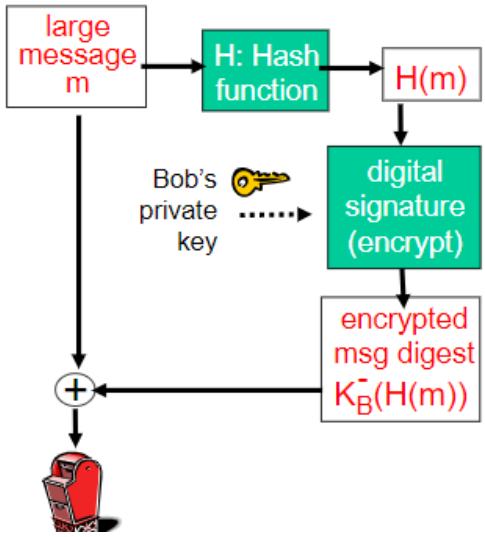
Il protocollo di firma digitale è composto dalla parte di firma e dalla parte di verifica. La parte di firma è composta da:

1. **Generazione della chiave pubblica e privata:** la chiave pubblica viene inviata al destinatario (vedremo più avanti come viene gestita la cosa), mentre la chiave privata viene tenuta segreta.
2. **Generazione del messaggio:** il mittente genera il messaggio che vuole firmare.
3. **Generazione hash del messaggio:** il mittente calcola l'hash del messaggio.
4. **Firma dell'hash:** il mittente firma il digest appena calcolato con la sua chiave privata. La sua firma diventa quindi

$$\boldsymbol{ChiavePrivata(H(m))}$$

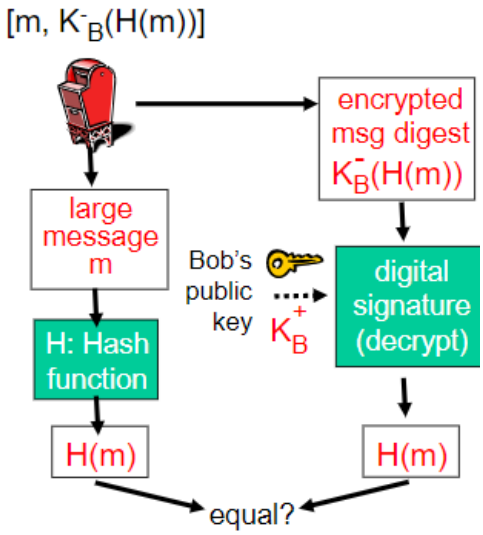
Si fa così perché sarebbe troppo dispendioso calcolare l'hash dell'intero messaggio.

5. **Concatenazione:** il mittente concatena il messaggio con la firma e invia
$$[m, ChiavePrivata(H(m))]$$



La parte di verifica è composta da:

1. **Ricezione del messaggio:** il destinatario riceve il messaggio $[m, ChiavePrivata(H(m'))]$.
2. **Estrazione della firma:** il destinatario estrae la firma dal messaggio ricevuto.
3. **Decrittazione della firma:** il destinatario applica la chiave pubblica del mittente alla firma che ha appena estratto, da cui ricava $H(m')$.
4. **Confronto:** il destinatario confronta l'hash del messaggio m con l'hash ottenuto decrittografando la firma. Se i due valori coincidono, allora il messaggio è integro e il mittente è autentico.



5.2.4 COME FACCIO AD ESSERE SICURO CHE QUELLA CHE RICEVO SIA EFFETTIVAMENTE LA CHIAVE DEL MITTENTE?

Il protocollo è sensibile agli attacchi **man-in-the-middle**: un attaccante può firmare un messaggio utilizzando la sua chiave privata, e sostituire la chiave pubblica del mittente con la sua.

Per ovviare a questo problema, si può usare un protocollo di firma digitale a chiave pubblica **con certificato**:

- Il mittente invia al destinatario il suo certificato, che contiene la sua chiave pubblica e la firma digitale del certificatore.
- Il certificato viene quindi inviato al destinatario, che può verificare la firma digitale del certificatore, e quindi accertare che la chiave pubblica sia autentica.

Il certificatore è una **terza parte** che ha la responsabilità di certificare la validità di una chiave pubblica.

5.3 METODI PER INVIARE UNA CHIAVE DI SESSIONE CON RISERVATEZZA

Gli approcci possibili sono 3:

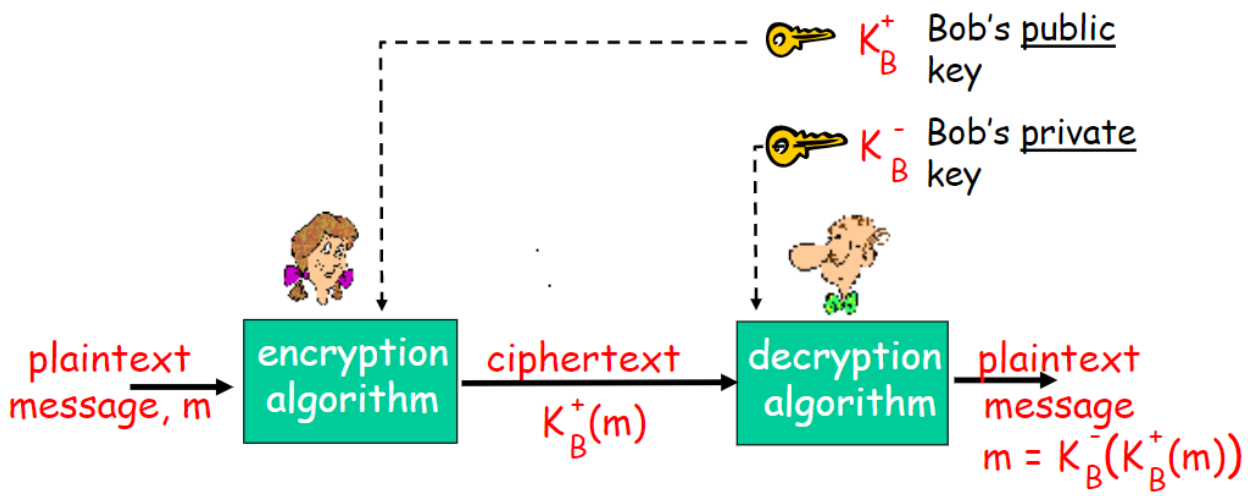
1. **Scambio diretto:** le due persone si incontrano direttamente di persona e si scambiano la chiave.
2. **Scambio utilizzando la crittografia a chiave pubblica:** permette di comunicare senza scambio di chiavi → Dato che ha complessità maggiore della comunicazione a chiave simmetrica, si farà in modo di utilizzare questo meccanismo solo per scambiarsi la chiave stessa.
3. **KDC (Key Distribution Center):** gli host si rivolgono a queste entità mediatrici per agevolare lo scambio della chiave.

5.3.1 METODO CON CHIAVE PUBBLICA

Permette una comunicazione sicura **senza alcuno scambio**. In particolare, **ogni host ha una coppia di chiavi:**

1. Una **chiave pubblica** che è nota a tutti.
2. Una **chiave privata** che è nota solo all'host.

Quando il mittente vuole mandare un messaggio *m*, applica come chiave di cifratura di un algoritmo (eventualmente pubblico), la **chiave pubblica del destinatario**. Il destinatario, quindi, decifra il messaggio utilizzando la sua chiave privata.



La coppia di chiavi è pensata per essere applicata in qualsiasi ordine per criptare/decriptare un messaggio.

$$\underbrace{K_B^-(K_B^+(m))}_{\text{use public key first, followed by private key}} = m = \underbrace{K_B^+(K_B^-(m))}_{\text{use private key first, followed by public key}}$$

Questo metodo ha dei problemi:

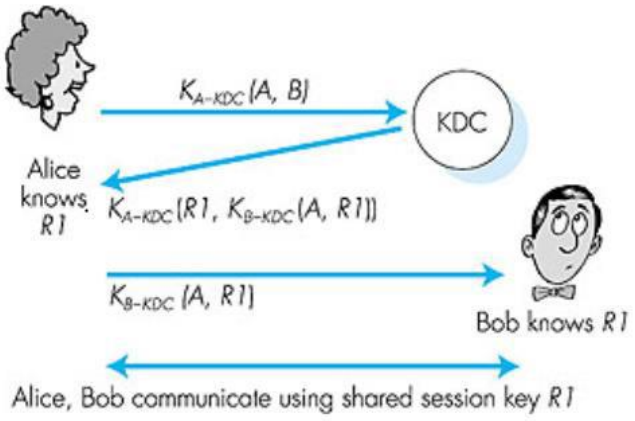
- **Chiunque** può mandare dei messaggi criptati al destinatario, pretendendo di essere il reale mittente, perché entrambi utilizzerebbero la stessa chiave pubblica del destinatario.
- Il sistema è esposto ad attacchi di tipo **chosen-plain-text**, ovvero l'attaccante può decidere una frase in chiaro e criptarla nello stesso modo di come il mittente cripterebbe la stessa frase.
- **È computazionalmente più complicato comunicare in questo modo**.

5.3.2 METODO CON CHIAVE SIMMETRICA E KDC

- Mittente (Alice) e destinatario (destinatario) vanno di persona da un KDC fisico e si registrano. Il KDC consegna loro una chiave che permette la comunicazione in chiave simmetrica con il KDC stesso.

$$K_{A-KDC}: chiave\ da\ Alice\ a\ KDC$$
$$K_{B-KDC}: chiave\ da\ Bob\ a\ KDC$$

- Alice comunica a KDC che vuole parlare con Bob.
- Il KDC, usando la stessa comunicazione (K_{A-KDC}) manda ad Alice una chiave di sessione **R1**, che sarà la chiave che dovrà usare per parlare con Bob.
- **Come fa Bob a ricevere questo R1?** Il KDC, nel messaggio precedente, invia un messaggio cifrato utilizzando la chiave di Bob, che quindi Alice non riesce a comprendere. Alice prende quindi questo messaggio e lo spedisce a Bob, che scopre quindi che è un messaggio del KDC che dice "per parlare con Alice usa R1".
- **Ora Alice e Bob possono parlare usando la chiave simmetrica di sessione R1.**

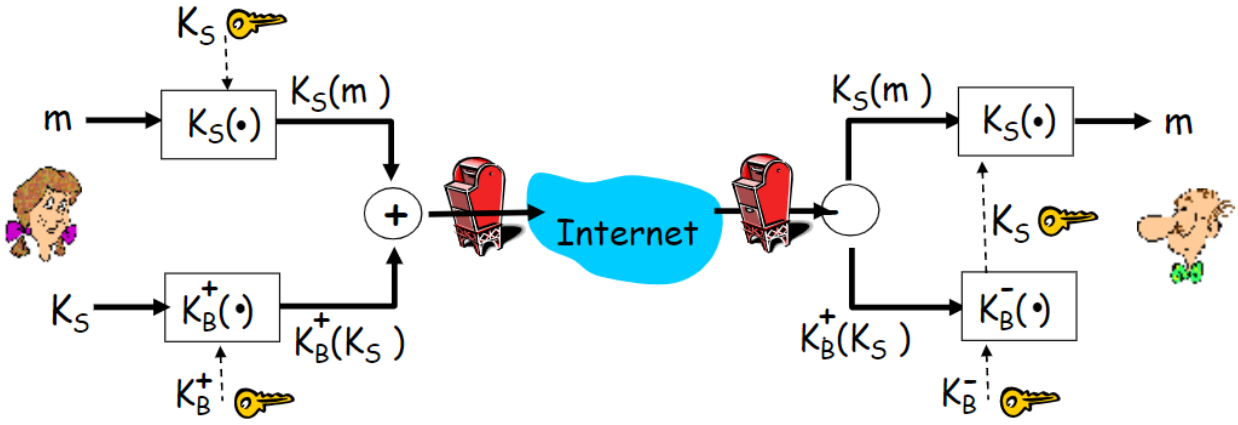


5.4 PGP (PRETTY GOOD PRIVACY)

Il PGP è un metodo di implementazione della sicurezza **a livello applicazione**.

Ipotizziamo di avere un servizio di scambio di messaggi mediante e-mail. Vogliamo che siano possibili:

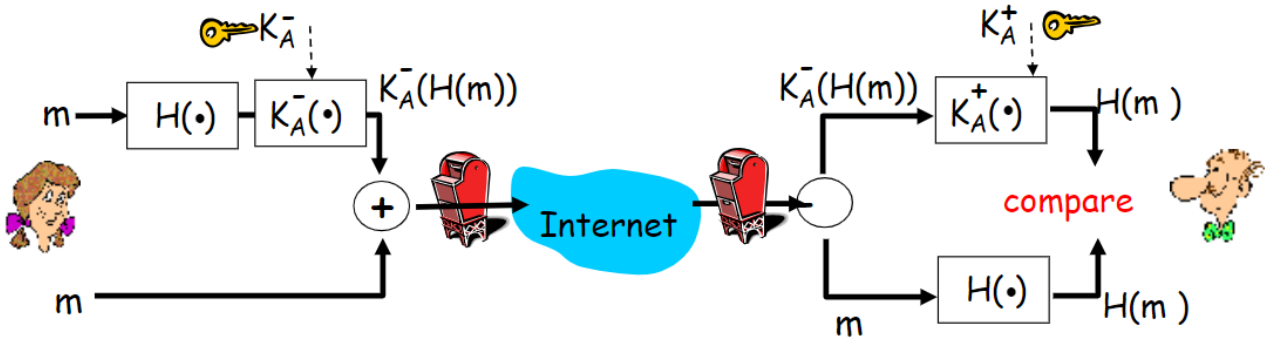
1. Autenticazione end-to-end.
2. Integrità.
3. Confidenzialità del messaggio.



- Alice **cripta il messaggio con una chiave simmetrica K_S** → Assicura la **confidenzialità** della comunicazione.
- Alice concatena a questo messaggio criptato la **chiave K_S criptata utilizzando la chiave pubblica di Bob K_B^+.**
- Bob può ricevere il messaggio criptato **ed è l'unico che, tramite la sua chiave privata, può scoprire come decriptarlo.**

C'è un problema: questo sistema non assicura alcuna autenticazione.

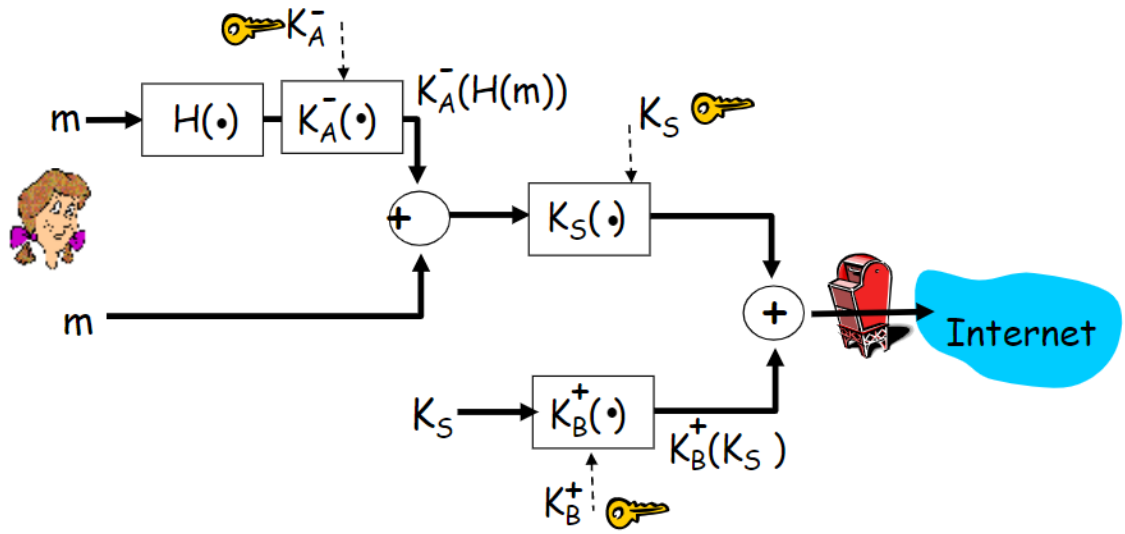
Alice vuole mandare un messaggio a Bob con l'obiettivo di ottenere **l'autenticazione del messaggio.**



- Alice prepara il messaggio *m* e lo concatena al **digest di *m* cifrato con la sua chiave privata** → Alice pone la sua firma digitale sul messaggio.

- Bob utilizza la **chiave pubblica di Alice per ottenere il digest**. Calcola poi il digest sul messaggio ricevuto per confrontare i risultati ottenuti: se il confronto è positivo Bob può concludere che il messaggio è di Alice ed è integro.

Quello che dobbiamo fare per ottenere tutte e tre le proprietà è **concatenare i due meccanismi**.



Prima si crea la coppia $[m, FirmaDigitale]$, poi si cifra il tutto con la chiave simmetrica. Infine, si concatena la stessa chiave simmetrica cifrata con la chiave pubblica del destinatario.

Quindi, in generale, abbiamo bisogno di tre chiavi:

1. La **chiave simmetrica** per garantire la riservatezza del messaggio.
2. La **chiave pubblica di Bob** per permettere lo scambio della chiave simmetrica.
3. La **chiave privata di Alice** per assicurare la firma digitale.

Il problema dell'implementazione della sicurezza a questo livello è che va fatto **per ogni applicazione**.

5.5 SSL (SECURE SOCKET LAYER)

5.5.1 A CHE LIVELLO È IMPLEMENTATA?

Secure Socket Layer (SSL), vecchio nome del protocollo Transport Layer Security (TLS), è un protocollo di sicurezza posto tra il livello di trasporto e il livello applicazione.

5.5.2 CHE SERVIZI IMPLEMENTA QUESTA LIBRERIA?

Mette a disposizione delle API per fornire crittografia a livello applicazione, permettendo di concentrarsi solo sulla logica della stessa.

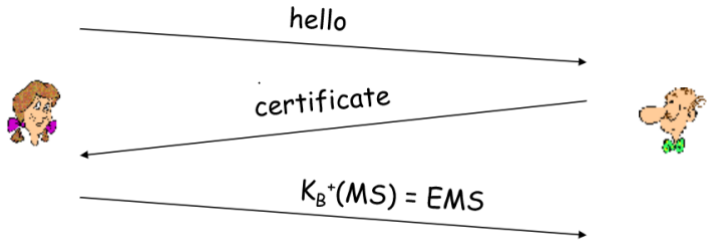
Viene garantita la confidenzialità, l'integrità dei dati e l'autenticazione degli endpoint.

5.5.3 SSL: VERSIONE SEMPLIFICATA

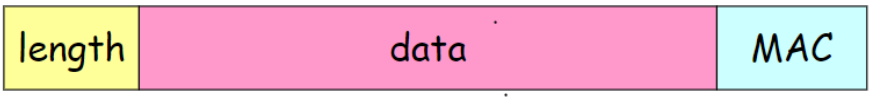
Il protocollo consiste nel fornire un flusso sicuro di byte mediante lo scambio di un intero set di chiavi che supporti l'intera connessione.

Si divide in quattro fasi:

1. **HANDSHAKE:** c'è l'autenticazione della connessione e lo scambio di una chiave segreta.
   Il client manda un messaggio di *inizio connessione* al server, mentre il server risponde autenticandosi, ovvero mandando un certificato con la propria chiave pubblica. A questo punto il client genera un **master secret**, e lo *cifra* utilizzando la chiave pubblica del server.



2. **GENERAZIONE DELLE CHIAVI:** il client e il server, mediante la crittografia a chiave pubblica, usano la chiave condivisa per generare 4 chiavi di sessione. In particolare, vengono generate:
   a. K_C – chiave per cifrare i messaggi dal client al server.
   b. M_C – chiave per generare il MAC dal client al server.
   c. K_S – chiave per cifrare i messaggi dal server al client.
   d. M_S – chiave per generare il MAC dal server al client.
3. **DATA TRANSFER:** i messaggi vengono divisi in blocchi, detti **record**, ai quali vengono concatenati un **MAC** e un campo **length** per discernere tra MAC e record e per permettere record di dimensioni diverse.



   In questa fase si inserisce anche un **numero di sequenza** usato per calcolare il MAC, che sarò ora composto da $H(M_C \mid NumeroSequenza \mid m)$, in modo da proteggersi dagli attacchi *record&playback*.
   Inoltre, si usa un campo **type** per evitare il *truncation attack*, dove si costringe il destinatario a chiudere la connessione.
   Il record sarà quindi composto da:
   - ***Version:*** versione del protocollo.
   - ***Type:*** tipo di record.
   - ***Length:*** lunghezza del record.
   - ***Data:*** dati.
   - ***MAC:*** $H(M_C \mid NumeroSequenza \mid Type \mid m)$.

5.5.4 SSL: VERSIONE REALE

Al protocollo semplificato mancano delle funzionalità, come ad esempio la **negoziazione degli algoritmi di crittografia**.

Sia il client che il server potrebbero decidere di utilizzare un algoritmo piuttosto che un altro; tuttavia, c'è bisogno che entrambi utilizzino lo **stesso** algoritmo.

Gli algoritmi su cui si devono mettere d'accordo sono tre:

1. L'algoritmo di **chiave pubblica (solitamente RSA).**
2. L'algoritmo di **crittografia a chiave simmetrica.**
3. L'algoritmo di **generazione del MAC.**

Nella fase di handshake dobbiamo quindi prevedere:

1. L'autenticazione del server.
2. La negoziazione degli algoritmi di crittografia.
3. Lo stabilire delle chiavi.
4. L'autenticazione del client.

NEGOZIAZIONE DEGLI ALGORITMI DI CRITTOGRAFIA

1. Il client invia al server la lista di algoritmi da lui supportati, assieme ad un nonce.
2. Il server sceglie dalla lista un algoritmo a chiave simmetrica (es. AES), uno a chiave pubblica (es. RSA con una specifica lunghezza di chiave) e un algoritmo MAC. Restituisce al client le proprie scelte, insieme ad un certificato e al nonce del server.
3. Il client verifica il certificato, estrae la chiave pubblica del server, genera un **pre-master secret (PMS)** e lo cifra con la chiave pubblica del server per poi mandarlo cifrato al server.
4. Usando la stessa funzione di derivazione della chiave, come specificato dallo standard SSL, il client e il server calcolano indipendentemente il master secret partendo da PMS e nonce. Il master secret viene poi suddiviso per generare le due chiavi di cifratura e le due chiavi MAC.
5. Il client invia al server un MAC di tutti i messaggi di handshake.
6. Il server invia al client un MAC di tutti i messaggi di handshake.

Dato che la lista di algoritmi viene inviata in chiaro, i passaggi 5 e 6 servono a prevenire attacchi **man-in-the-middle**, nei quali un attaccante può modificare la lista di algoritmi, togliendo quelli più robusti e sostituendoli con alcuni più deboli.

- Se il client invia al server un MAC di tutti i messaggi inviati e ricevuti, il server può confrontare questo MAC con quello dei messaggi da lui inviati e ricevuti: se c'è un'inconsistenza, può chiudere la connessione.
- Lo stesso vale per il server, permettendo quindi al client di eventualmente chiudere la connessione.

I nonce sono utilizzati per evitare attacchi di tipo **reply**: cambiando il nonce cambieranno anche le chiavi crittografiche, ed essendo i nonce casuali e diversi per ogni sessione, non è possibile usare i vecchi messaggi per instaurare una nuova sessione.

5.6 IPSEC E VPN

IPsec è un servizio di sicurezza che opera a livello di rete e permette di implementare le **VPN (Virtual Private Network)**: una VPN sfrutta l'Internet pubblica per creare una rete privata tra alcuni nodi (i meccanismi di sicurezza vengono quindi implementati all'interno dell'Internet pubblica).

Si basa su un insieme di protocolli che permettono di garantire la sicurezza dei pacchetti IP.

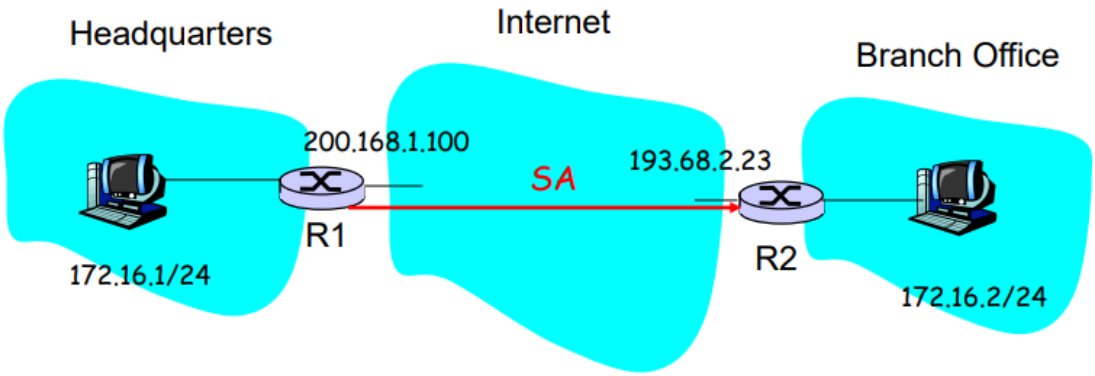
Questi offrono **tutti** autenticazione e integrità (un esempio è l'Autentication Header) mentre la confidenzialità dipende dallo specifico protocollo.

ENCAPSULATION SECURITY PROTOCOL (ESP)

L'esempio che prendiamo in analisi è l'**Encapsulation Security Protocol (ESP)**, un protocollo di sicurezza che si basa su autenticazione e cifratura.

Il meccanismo di criptazione non deve essere necessariamente noto agli host finali, ma può essere implementato sui due router di frontiera di sorgente e destinazione.

Immaginiamo che il meccanismo IPsec sia implementato sui due router R1 e R2.



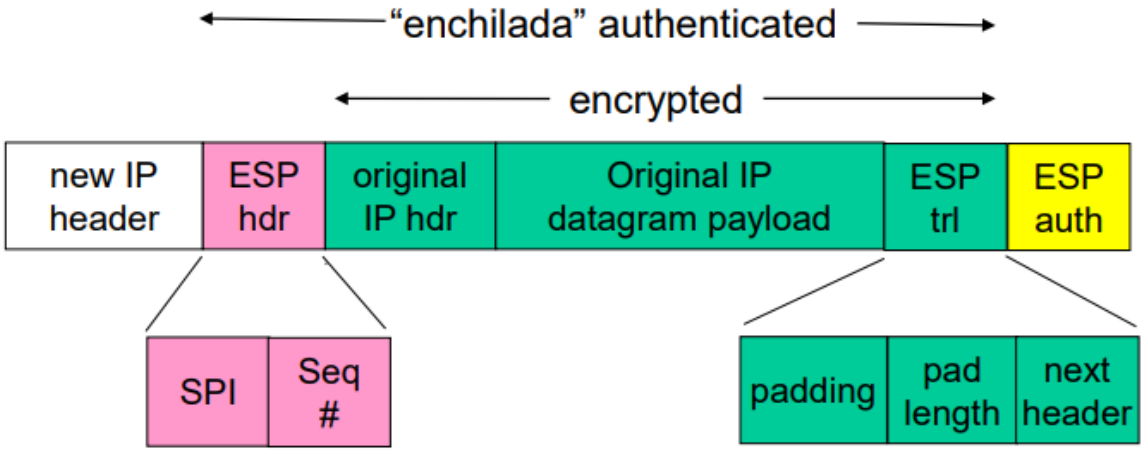
I due router devono stabilire una o più connessioni dette **Security Associations (SA)**: è un meccanismo simplex, cioè funziona solo in una direzione, e ne servono due se si vuole anche il viceversa.

Se ci sono N nodi nella rete, ci saranno 2 + 2N Security Association.

Ogni nodo, per far funzionare il meccanismo, mantiene in memoria, dentro un database interno ai router chiamato **SAD (Security Associations Database)**, le seguenti informazioni:

- **SPI (Security Parameter Index):** identificatore a 32 bit che individua la SA che l'host sta gestendo.
- **IP dell'interfaccia di origine** (nell'esempio R1).
- **IP dell'interfaccia di destinazione** (nell'esempio R2).
- **Tipo di cifratura utilizzata.**
- **Chiave simmetrica di cifratura.**
- **Tipo di algoritmo di integrità** (ad esempio HMAC con funzione hash MD5).
- **Chiave di autenticazione per il controllo di integrità.**

La VPN in sostanza implementa IPsec sui router e/o sui singoli host, rimpiazzando il payload del pacchetto IP con un pacchetto IPsec e un nuovo IP Header (IPv4). Il datagram IPsec che utilizza il protocollo ESP sarà formato da:



1. **new IP header:** con questo nuovo header il datagram può viaggiare correttamente sulla rete.
2. **ESP hdr (ESP header):** contiene il **SPI** e un **numero di sequenza** inizializzato a 0 per ogni SA e incrementato ad ogni comunicazione che coinvolga la SA.
   Questa è una misura di sicurezza che protegge da attacchi di tipo replay.
3. **Original IP hdr (original IP header):** header originale del pacchetto IP.
4. **Original IP datagram payload:** payload originale del pacchetto IP.
5. **ESP trl (ESP trailer):** che contiene il padding, la lunghezza del padding e il next header, perché gli algoritmi di cifratura spesso ragionano a blocchi e quindi si richiede che la roba da criptare sia multipla di un certo valore.
6. **ESP auth (ESP authentication data):** contiene il MAC, generato utilizzando la chiave di autenticazione ottenuta guardando il SAD.

AUTENTICATI

AUTENTICATI E CRITTOGRAFATI

Così facendo abbiamo eseguito una encapsulazione del pacchetto IP originale, che è stato quindi crittografato e autenticato.

Il nuovo pacchetto IP è quindi pronto per essere inviato sulla rete pubblica e viene instradato tramite un **Security Police Database (SPD)** che contiene le regole di routing per la VPN.

Il pacchetto IPsec viene quindi decapsulato dal router di destinazione e il pacchetto originale viene rilasciato verso il destinatario.

6 WIRELESS AND MOBILE NETWORK

6.1 PROBLEMA DEL NODO NASCOSTO E DEL NODO ESPOSTO

Il problema del nodo nascosto e del nodo esposto derivano dalle caratteristiche di una rete wireless (infatti, questi due problemi non si verificano nelle reti wired):

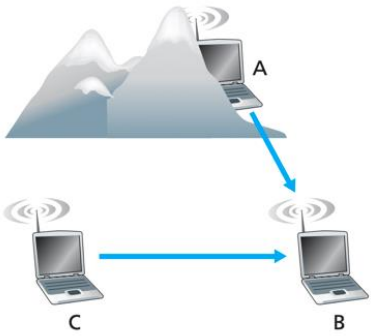
1. La potenza del segnale diminuisce man mano che si propaga nello spazio.
2. Ci sono interferenze con le altre sorgenti.
3. Il segnale si propaga su più percorsi, dato che il segnale rimbalza e si riflette su eventuali ostacoli.

6.1.1 PROBLEMA DEL NODO NASCOSTO

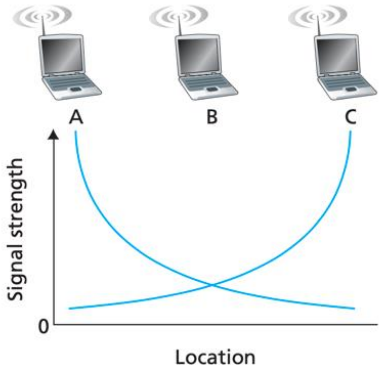
DA COSA DERIVA IL NOME: il nome deriva dal fatto che, se abbiamo tre ricetrasmettitori (A, B, C) e, ad esempio, A e C che vogliono comunicare con B non si rendono conto delle interferenze che stanno generando perché "non riescono a sentirsi".

Abbiamo due possibili cause:

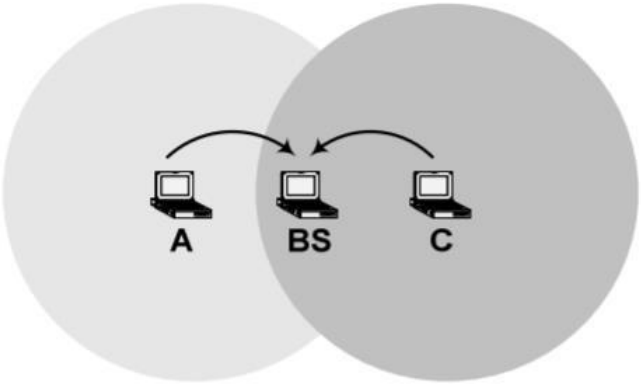
1. A e C sono separati da un ostacolo che fa in modo che il segnale di C non arrivi mai ad A, e viceversa.
   Entrambi possono decidere di comunicare con B, che però sicuramente rileverà un'interferenza. **A e C, però, non se ne renderanno mai conto, perché entrambi crederanno di essere in due a comunicare.**



2. A e C sono abbastanza distanti affinché i loro segnali appaiano all'altro come rumore, e quindi da non considerare come interferenza.



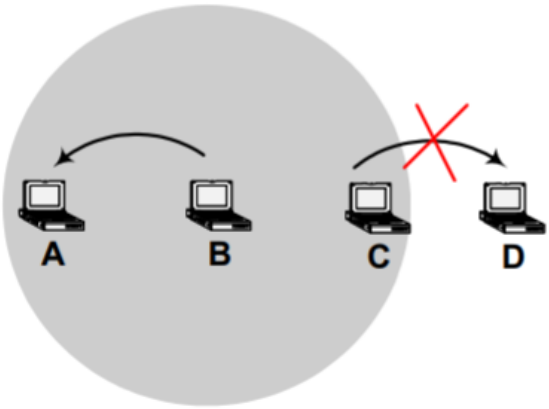
In generale, possiamo schematizzare i due casi in questo modo: se A e C vogliono trasmettere a BS, entrambi vedranno il canale idle per via del fatto che A non si accorge della trasmissione di C, e viceversa. Quando si arriva in prossimità di BS la collisione avviene.



6.1.2 PROBLEMA DEL NODO ESPOSTO

Se C volesse trasmettere a D mentre B sta trasmettendo ad A, in linea teorica non dovrebbe succedere nulla.

Tuttavia, C vede che il nodo B sta trasmettendo, e quindi interpreta il canale di comunicazione come occupato, e decide quindi di non trasmettere, in linea con il protocollo CSMA/CA.



6.1.3 COME SI RISOLVE IL PROBLEMA?

Il problema si risolve con il **Virtual Carrier Sensing**, un'estensione del CSMA/CA, nel quale la decisione di trasmettere oppure non trasmettere non si prende in base a una misurazione fisica.

Viene eseguita una sorta di prenotazione per un certo intervallo di tempo del mezzo di comunicazione.

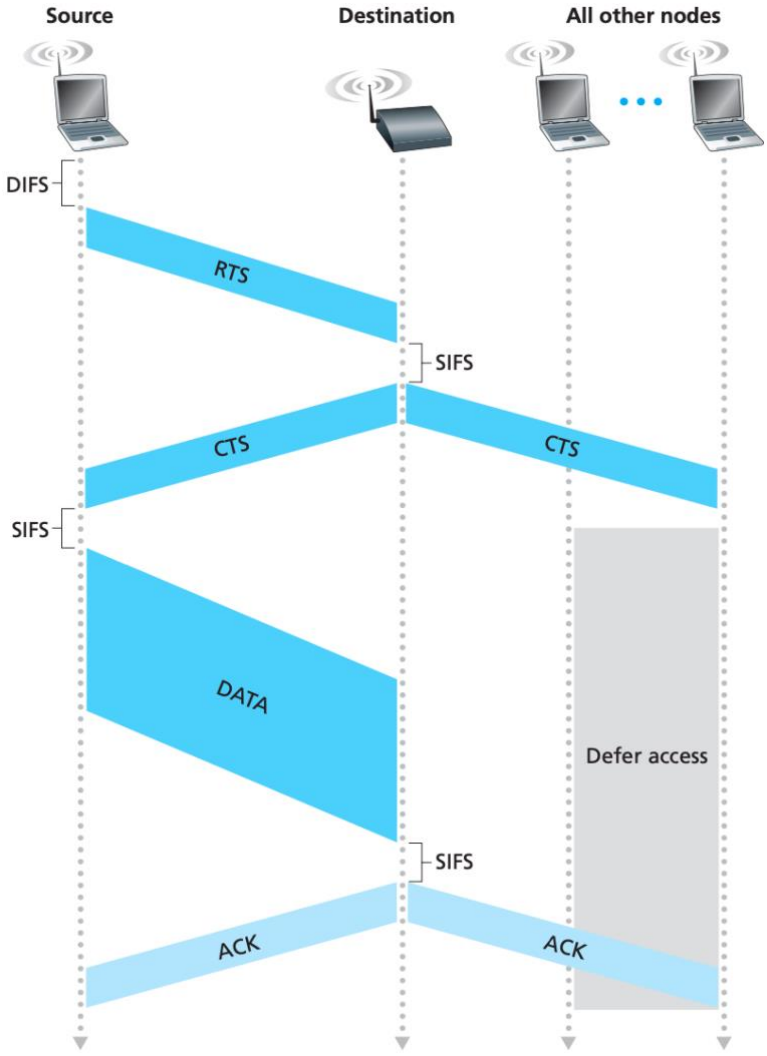
Supponiamo che un nodo voglia trasmettere: andrà ad ascoltare il canale.

- Nel caso in cui esso sia libero per un tempo **DIFS**, allora invierà un pacchetto **RTS (Request To Send)**, che viene ricevuto sia dalla stazione base che da tutti gli altri nodi, essendo il mezzo wireless.

Chiunque ascolti tale pacchetto imposterà un timer, detto **NAV RTS (Network Allocation Vector RTS)**, per un tempo pari a quello specificato nel campo **duration** del pacchetto RTS.

- Dopo un tempo **SIFS** la stazione base procederà a rispondere con un pacchetto **CTS (Clear To Send)**: anche questo pacchetto contiene un campo **duration**, minore del valore contenuto in RTS.
  Esso farà partire il timer **NAV CTS**, anche a coloro che non avevano sentito in precedenza il NAV RTS.
- Quando la sorgente recepisce il CTS, fa **ACK della trasmissione**, attende un tempo SIFS e successivamente inizia a trasmettere il frame.
- Una volta terminata la trasmissione, attende un tempo SIFS e fa **ACK della ricezione**. Tutti i nodi vedranno quindi il termine della trasmissione, attenderanno un tempo DIFS e successivamente un altro timer di backoff scelto casualmente.

Il timer NAV serve a tutti i nodi per non trasmettere fintantoché c'è un NAV attivo.



La collisione è ancora presente, nel caso specifico in cui due nodi inviano RTS nello stesso momento. Esso, tuttavia, è pensato per essere abbastanza piccolo da rendere minore possibile la probabilità di collisione e soprattutto, anche se dovesse accadere, verrebbe perso solo il tempo relativo a RTS, che è appunto abbastanza piccolo.

Tra i contro di questo protocollo c'è sicuramente l'aumento di overhead in preparazione alla trasmissione vera e propria: per tale motivo questo non viene usato nel caso in cui la dimensione del frame sia paragonabile a quella del pacchetto RTS → Verrà infatti inviato direttamente il frame.

6.2 PROTOCOLLO CSMA/CA

All'interno delle reti Wi-Fi si può utilizzare il protocollo **CSMA (Carrier Sense Multiple Access)**, tuttavia abbiamo detto che non abbiamo la possibilità di fare **collision detection.** Infatti:

- Il dispositivo Wi-Fi che sta trasmettendo **non può ascoltare il canale**, perché sta trasmettendo. Infatti, solitamente **è dotato di una sola antenna:** una soluzione potrebbe essere usare due antenne.
- Il problema del **nodo nascosto** non viene risolto: anche se l'host avesse più antenne, ci sono interferenze che il dispositivo non riuscirebbe a rilevare a causa del fatto che un altro dispositivo sta trasmettendo a una distanza considerevole, oppure perché sono separati da un ostacolo che scherma il segnale.

Il punto del protocollo **CSMA/CA (CSMA/Collision Avoidance)** è quello di prevenire le collisioni (che possono comunque avvenire) usando uno schema **ARQ (Automatic Repeat reQuest)** con una divisione in piccoli slot, usati comunque solo per dare inizio alle operazioni.

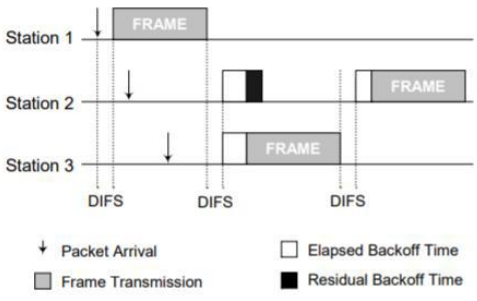
6.2.1 SPECIFICARE I PASSAGGI

Queste, in ordine, sono:

1. La sorgente che vuole trasmettere ascolta il canale e se lo vede libero per un tempo **DIFS**, decide di **trasmettere l'intero frame.**
2. Subito dopo la ricezione del frame, il destinatario muta da ricezione a trasmissione entro un tempo **SIFS** (switching time) tale che $SIFS < DIFS$.
3. Invia un **ACK** al mittente.

Inoltre, il protocollo permette di prevenire la collisione tra due frame che vengono inviati dopo aver osservato il mezzo libero per un tempo DIFS grazie a un **backoff time** scelto casualmente, da aggiungere al tempo DIFS di attesa del punto 1. Se infatti viene rilevato l'uso del mezzo durante il tempo di backoff, il timer viene congelato e verrà risvegliato dopo la fine della trasmissione e l'attesa del nuovo periodo di backoff. Quindi, se il canale è occupato:

1. Comincia un tempo casuale di backoff.
2. Il timer scorre solo quando il canale è libero (tempo DIFS escluso).
3. Trasmetti quando il timer scade.
4. Se non arriva l'ACK, incrementa il tempo di backoff e torna al punto 1.



6.2.2 SPECIFICARE PERCHE' SIFS E DIFS HANNO DURATA DIFFERENTE

SIFS deve **necessariamente** essere minore di DIFS, altrimenti un nodo potrebbe rilevare il mezzo come libero, e potrebbe trasmettere un frame mentre il mittente originale sta ancora aspettando l'ACK.

6.2.3 CALCOLO DEL TEMPO DI BACKOFF

Come tempo di backoff si sceglie un numero random di slot da aspettare in

$$[\mathbf{0}, \boldsymbol{CW} - \mathbf{1}]$$

Inizialmente CW vale $\boldsymbol{CW_{MIN}}$ (standard del protocollo) e ogni volta che abbiamo un ACK mancante raddoppia fino ad arrivare a $\boldsymbol{CW_{MAX}}$.

CW è quindi sempre compreso nell'intervallo $[CW_{MIN}, CW_{MAX}]$, i cui valori dipendono dal livello fisico di riferimento.

Questo protocollo, tuttavia, non risolve il problema del nodo nascosto: se i due nodi che vogliono comunicare non si vedono tra di loro, ognuno vedrà il canale libero e andrà a trasmettere causando una collisione in ricezione sull'access point.

6.3 MOBILE IP

Introduciamo il concetto di **mobilità:** un host è in grado di spostarsi mentre è connesso ad una rete, ha delle connessioni TCP attive ecc…

Abbiamo, nello specifico, tre tipi di mobilità:

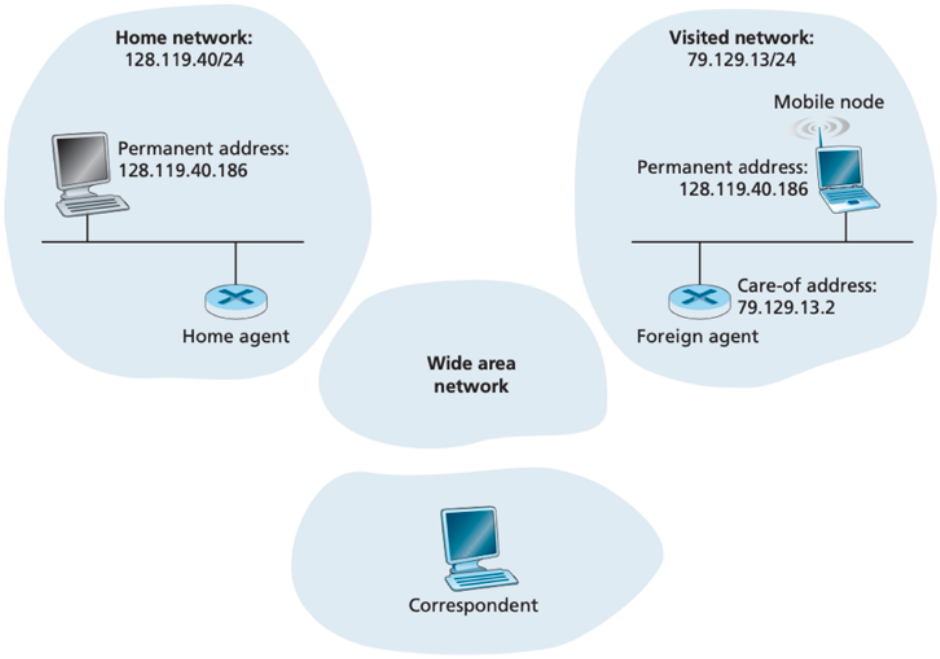
1. **Nessuna:** l'host cambia punto di accesso, ma rimane sempre all'interno della stessa sottorete. In questo caso, non è necessario cambiare l'indirizzo IP.
2. **Media:** l'host cambia rete, e con essa anche l'indirizzo IP tramite DHCP. Si parla di mobilità senza continuità, ma non rappresenta un problema.
3. **Alta:** l'host cambia diverse reti in poco tempo. La gestione di tale situazione diventa necessaria per un corretto funzionamento dell'host che vuole comunicare con altri host (infatti, cambiando IP la connessione TCP con un eventuale server cadrebbe).

Nell'ultimo caso non possiamo pensare di gestire il tutto tramite una continua connessione/riconnessione, altrimenti avrei un servizio intermittente. Peraltro, questa possibilità non è neanche prevista dal protocollo IP.

6.3.1 COSA VUOL DIRE CHE CAMBIA IL PUNTO DI ACCESSO?

Cambio punto di accesso, e quindi sono mobile, se e solo se cambio sottorete di accesso. Se invece cambio access point, ma resto sempre all'interno della stessa sottorete, il mio IP non cambia.

6.3.2 SCHEMA E SIGNIFICATO DEI SUOI ELEMENTI



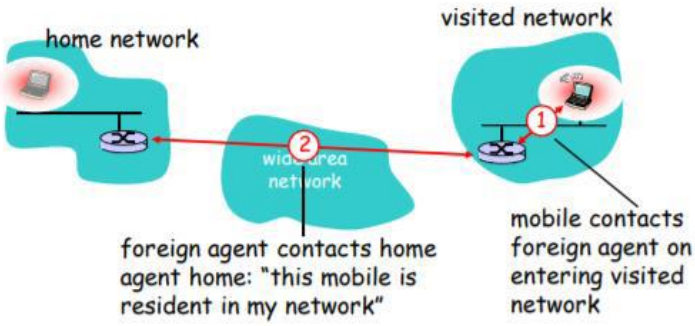
- **HOME NETWORK:** è la rete in cui si trova inizialmente l'host mobile.
- **PERMANENT ADDRESS:** è l'address preciso dell'host che si muove all'interno della home network. L'obiettivo è fare in modo che tutti possano raggiungere l'host mobile utilizzando questo indirizzo.
- **HOME AGENT:** è il router che cercherà di comportarsi come l'host mobile quando lui è via.
- **VISITED NETWORK:** è la sottorete IP in cui il nodo mobile decide di spostarsi.
- **CARE-OF-ADDRESS:** è l'indirizzo specifico che l'host ha all'interno della visited network.
- **FOREIGN AGENT:** è il router che riesce a far comunicare gli altri con l'host mobile.
- **CORRESPONDENT:** l'host che vuole comunicare con l'host mobile.

6.3.3 COME POSSIAMO PERMETTERE LA CONTINUITA' DEL SERVIZIO IN CASO DI MOBILITA' (FARE ANCHE IL DISEGNO DELLE PARTI COINVOLTE)?

Mettiamoci in uno scenario in cui l'host sia arrivato nella visited network e si sia registrato al foreign agent.

Compito del foreign agent è vedere il permanent address dell'host mobile e comunicare all'home agent il nuovo indirizzo IP dell'host mobile.

- Alla fine di questa fase, il foreign agent è a conoscenza del fatto che c'è un host mobile nella sua sottorete, e l'home agent sa dove si trova l'host mobile in quel momento.



Chi si occupa della mobilità? I router potrebbero aggiornare le proprie tabelle coerentemente con l'indirizzo IP che l'host guadagna all'interno della visited network, ma questa soluzione non può scalare considerato l'elevatissimo numero di host mobili.

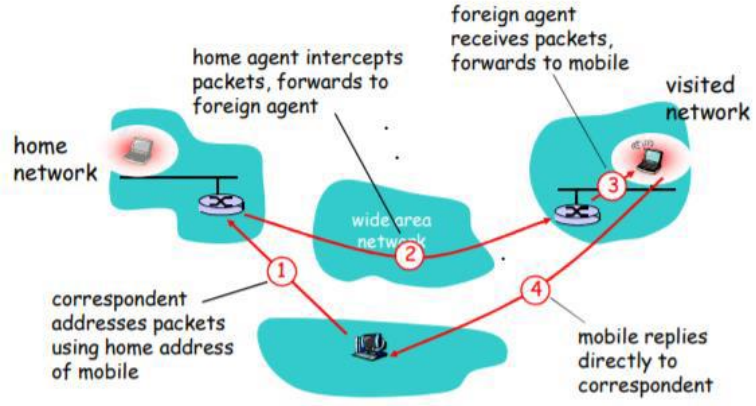
L'idea è quindi che siano gli end-system a curarsi di questo problema, in due modi:

1. **Indirect routing:** il correspondent comunica con l'home agent, che poi provvede a inoltrare la richiesta all'host mobile contattando il foreign agent.
2. **Direct routing:** il correspondent ottiene il nuovo indirizzo dell'host mobile, e comunica direttamente con lui.

6.3.4 COME VIENE GESTITO LO SPOSTAMENTO NELLE DUE MODALITA' DI ROUTING INDIRETTO E DIRETTO?

INDIRECT ROUTING: intanto, diciamo che non possiamo annunciare attraverso i protocolli di routing lo spostamento dell'host, perché il routing funziona sulla base della rete e non del singolo host. La situazione deve essere inevitabilmente gestita dall'home agent e dall'host.

1. Il correspondent utilizza il **permanent address** per indirizzare il suo messaggio. Questo messaggio sarà indirizzato dall'home agent.
2. L'home agent **manda i pacchetti al foreign agent**, e può farlo perché in fase di registrazione il foreign ha comunicato con l'home circa la propria ubicazione.
   Quando l'host mobile si sposta nella foreign network, contatta il foreign agent e gli dice di registrarlo presso l'home agent. Il foreign agent contatta quindi l'home agent, avvisandolo che se dovessero arrivare pacchetti diretti verso l'host mobile, essi dovranno essere reindirizzati verso la nuova rete.
3. Il foreign agent manda i pacchetti al mobile host.
4. L'host mobile riceve come source address l'indirizzo del correspondent, e dunque può rispondere direttamente a lui.



Tutti coloro che inviano all'host pacchetti non sono a conoscenza dello spostamento; quindi, continueranno ad inviare pacchetti allo stesso indirizzo.

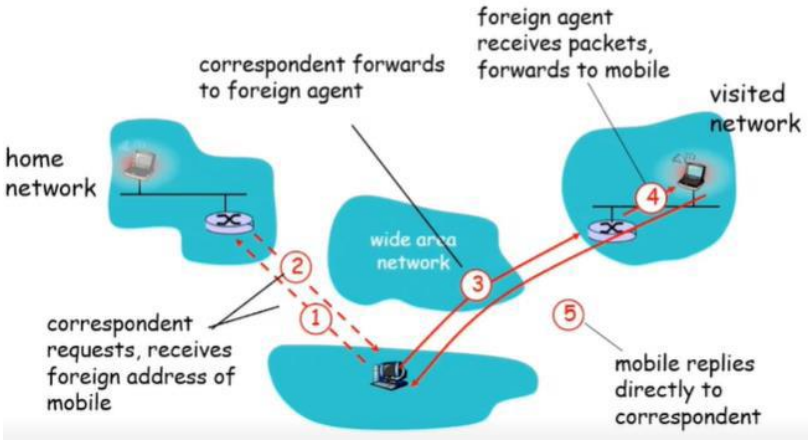
La tecnica è trasparente rispetto ai mittenti dei pacchetti verso l'host mobile.

In caso di nuovo spostamento, c'è ovviamente bisogno di una nuova registrazione: il rischio è che se gli spostamenti fossero molto frequenti, sarebbe più il tempo utilizzato per notificare lo spostamento rispetto a quello di permanenza nella rete stessa. Inoltre, durante tutto il tempo in cui l'host fa una nuova registrazione, l'home agent non è ancora a conoscenza dello spostamento, e tutti i pacchetti mandati in quel tempo saranno persi.

Il lato negativo di questa strategia è che è necessaria una triangolazione per comunicare, andando ad aumentare di molto l'overhead.

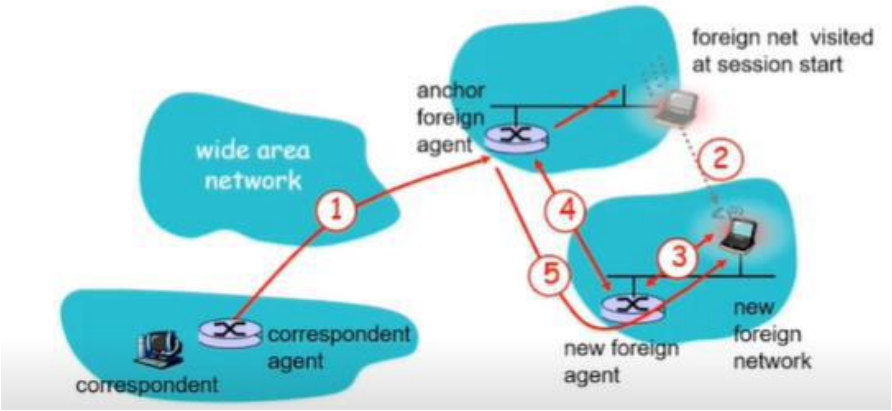
DIRECT ROUTING: gli elementi in gioco sono gli stessi dell'indirect routing ma, al contrario di quest'ultimo, riusciamo ad evitare la triangolazione dei dati, perché il mittente **invierà direttamente** al dispositivo mobile i pacchetti.

- Quando il dispositivo arriverà in una nuova rete, farà in modo di notificare all'home agent del suo spostamento.
- L'home agent, quando vedrà arrivare un pacchetto destinato all'host mobile, avviserà il mittente che il destinatario non si trova più nella home network.
- A questo punto, sarà il mittente stesso ad occuparsi di reinviare il pacchetto al nuovo indirizzo del dispositivo mobile.



Cosa succede se l'host mobile cambia nuovamente rete? Il correspondant non avrà bisogno di comunicare nuovamente con l'home agent, ma comunicherà con il foreign agent **ancora**, che a sua volta comunicherà con nuovo foreign agent in cui si trova l'host.

Affinché tutto funzioni, il nuovo foreign agent deve dire al vecchio foreign agent la nuova ubicazione dell'host mobile.



I principali problemi di questa strategia sono due:

1. **Tecnico:** abbiamo un aumento di overhead per la notifica del nuovo indirizzo, per la notifica del suo spostamento e il tempo di reinvio del pacchetto.
2. **Etico:** possono sorgere dei problemi di privacy in quanto il mittente sarà a conoscenza di tutti gli spostamenti dell'host mobile di rete in rete.

6.3.5 QUALE APPROCCIO UTILIZZA IL MOBILE IP?

Il mobile IP sfrutta il meccanismo dell'**indirect routing.**

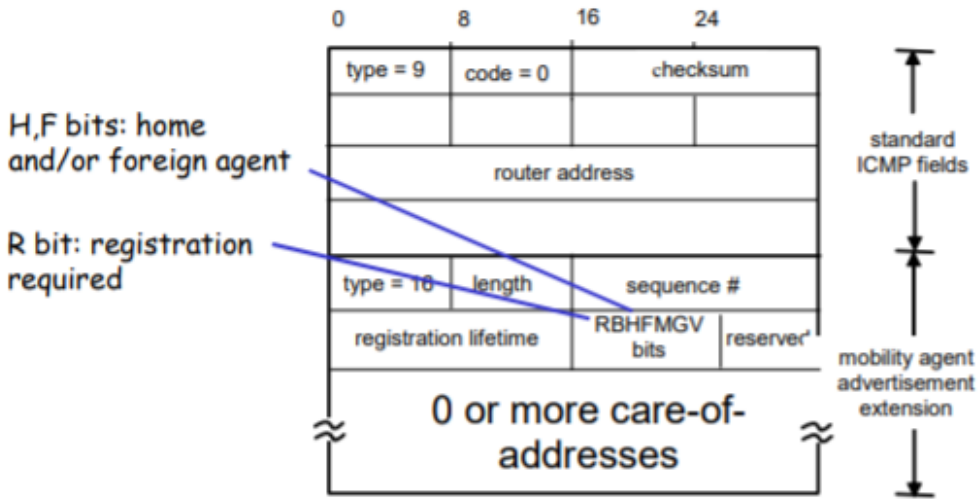
FORMATO DEL PACCHETTO MOBILE IP: l'home agent effettua un'operazione di "**encapsulation**" quando arriva un messaggio del correspondent, in cui incapsula il pacchetto in un altro che spedirà lui stesso, inserendo come destination address il care-of-address comunicatogli dal foreign agent in fase di registrazione.

Come fa il mobile host a scoprire il foreign agent, e come fa questi a comunicargli un care-of-address? I foreign/home agent mandano periodicamente dei messaggi ICMP di tipo 9. Il pacchetto completo sarà formato da questi campi standard dell'ICMP…

- **Type:** 8 bit, indica il tipo di pacchetto. Per il mobile IP il valore standard è 9.
- **Code:** 8 bit, indica il codice del pacchetto. Per mobile IP il valore standard è 0.
- **Checksum:** 16 bit, è il checksum del pacchetto.
- **Router address:** 32 bit, è l'indirizzo del router che ha ricevuto il pacchetto.

…e da altri campi tipici del mobile IP:

- **Type:** 8 bit, indica il tipo di pacchetto mobile IP. Il valore di default è 16.
- **Length:** 8 bit, indica la lunghezza del pacchetto.
- **Sequence number:** 16 bit, è il numero di sequenza del pacchetto.
- **Registration lifetime:** 16 bit, indica la durata della registrazione.
- **RBHFMGV bit:** 8 bit di informazioni aggiuntive.
  - **H/F:** indica se si tratta di foreign o home agent.
  - **R:** indica se è necessaria la registrazione.
- **Care-of-adress:** 32 bit, è l'indirizzo del dispositivo mobile.



REGISTRAZIONE.

1. L'host mobile aspetta di ricevere un ICMP dall'agent discovery.
2. Venuto a sapere del foreign agent presente, formula una **registration request** dove indica:
    a. **COA:** care-of-address, l'indirizzo el dispositivo mobile scelto.
    b. **HA:** home agent, l'indirizzo del router che si trova nella home network.
    c. **MA:** mobile agent, l'indirizzo del router che si trova nella foreing network.
    d. **Lifetime:** durata della registrazione.
    e. **Sequence number:** numero di sequenza del pacchetto.
3. Il foreign agent riceve il pacchetto di richiesta e lo inoltra all'home agent, aggingendo una nota sul tipo di encapsulation da usare.
4. L'home agent riceve il pacchetto, abbassa il lifetime richiesto e risponde con un pacchetto di **registration reply**, identico al registration request ma nel quale viene tolto il campo COA, come una sorta di ACK.
5. Il foreign agent riceve il pacchetto e lo inoltra al dispositivo mobile. Da questo momento in poi il mittente può inviare il traffico, e questo verrà reindirizzato al dispositivo mobile.

7 TEORIA LABORATORIO

7.1 QUALI SONO I REQUISITI PER ESSERE CONNESSI AD INTERNET?

I requisiti sono 4:

1. **INDIRIZZO IP:** sequenza di **32 bit** che identifica un dispositivo all'interno della rete. Per poter essere connessi ad Internet, questo indirizzo deve essere **pubblico** (i router hanno questo indirizzo pubblico perché devono decidere a quale dispositivo inviare i pacchetti).
   Si usa la notazione decimale puntata: i primi **k bit** a sinistra identificano la **rete** (indirizzo di rete), mentre i restanti **32 – k bit** a destra identificano l'**host** all'interno della rete (indirizzo dell'host).
2. **MASCHERA DI RETE (NETMASK):** sequenza di **32 bit** con **tanti bit a 1 quanti sono i bit che identificano la rete**, i restanti 32 – k bit sono a 0 (in corrispondenza dell'indirizzo dell'host).
   Tramite IP e maschera di rete possiamo ricavare:
   a. **Indirizzo di rete:** AND bit a bit tra IP e maschera di rete.
   b. **Indirizzo di broadcast:** OR bit a bit tra IP e maschera di rete negata.
3. **INDIRIZZO DEL GATEWAY.**
4. **INDIRIZZO DEL SERVER DNS.**

7.2 ip addr show e ip addr add

In generale, il comado `ip` serve per **visualizzare** e **modificare le impostazioni di rete**.

7.2.1 ip addr show

Mostra tutte le interfacce di rete.

- Il comando `ip addr show up` invece mostra solo le interfacce **accese**.

Un esempio di output è

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
state UP group default qlen 1000
  link/ether 08:00:27:9b:5c:4d brd ff:ff:ff:ff:ff:ff
  inet 192.168.1.52/24 brd 192.168.1.255 scope global eth0
    valid_lft forever preferred_lft forever
  inet6 fe80::ccbe:5117:5e99:2b07/64 scope link
  valid_lft forever preferred_lft forever
```

Gli elementi che ci interessano sono:

- `2: eth0` – indicano il numero di interfaccia e il nome dell'interfaccia.

- **<BROADCAST, MULTICAST, UP, LOWER_UP>** sono i flag dell'interfaccia. In questo caso l'interfaccia è in broadcast (invia a tutti), è in multicast (invia a più destinatari), è abilitata e in funzione.
- **mtu 1500** – Maximum Transmission Unit, è la dimensione massima in byte del pacchetto IP che deve essere spedito in rete. In questo caso è 1500 byte.
- **qdisc pfifo_fast** – Queuing Discipline, è il tipo di coda utilizzata per la gestione dei pacchetti in ingresso e in uscita. In questo caso si usa una coda FIFO.
- **state UP** – è lo stato dell'interfaccia. Può essere UP, ovvero abilitata (come in questo caso), oppure DOWN, ovvero disabilitata.
- **qlen 1000** – è la lunghezza della coda. In questo caso è di 1000 pacchetti.
- **link/ether 08:00:27:9b:5c:4d** – è il tipo di interfaccia (Ethernet) e l'indirizzo MAC.
- **brd ff:ff:ff:ff:ff:ff** – è l'indirizzo MAC di broadcast.
- **inet 192.168.1.52/24** – indica che stiamo utilizzando il protocollo IPv4 a livello network e ci dice che l'indirizzo IP è **192.168.1.52**, mentre la subnet mask è **/24**.
- **brd 192.168.1.255** – è l'indirizzo di broadcast.
- **scope global eth0** – è lo scope dell'indirizzo IP. In questo caso è globale.

7.2.2 ip addr add

Permette di **aggiungere** un indirizzo IP all'interfaccia di rete. Ad esempio

```
ip addr add 192.168.1.42/24 broadcast 192.168.1.255 dev eth0
```

aggiunge una configurazione specificando indirizzo IP, subnet mask, indirizzo di broadcast e interfaccia di rete.

Per rimuovere un indirizzo IP da un'interfaccia possiamo usare

```
ip addr del 192.168.1.42/24 dev [nome_interfaccia]
```

oppure

```
ip addr flush dev [nome_interfaccia]
```

Ad esempio, `ip addr flush dev eth0` rimuove l'indirizzo IP dell'interfaccia eth0.

La configurazione fatta con il comando `ip` viene annullata al riavvio della macchina, quindi l'interfaccia è resettata.

Per rendere permanente la configurazione usiamo il file di configurazione **/etc/network/interfaces** e i comandi **ifup** e **ifdown**.

Il file di configurazione si presenta così:

```
auto lo

iface lo inet loopback

iface eth0 inet static
      address 192.168.1.2
      netmask 255.255.255.0
```

```
broadcast 192.168.1.255
```

All'avvio viene inizializzata l'interfaccia di loopback, ovvero `lo`. Questa è necessaria per poter comunicare con sé stessi, e non ha indirizzo IP. Inoltre, è sempre UP e non ha coda. Viene usata per eseguire test di connessione e ping.

Con la parola `static` indichiamo che i parametri di rete specificati non cambiano, e quindi possiamo rendere la configurazione permanente.

Se al file aggiungiamo la riga **`auto eth0`** impostiamo che l'interfaccia `eth0` sia abilitata automaticamente all'avvio.

A questo punto, con i comandi `ifup` e `ifdown` possiamo:

- **`ifup eth0`** – abilitiamo l'interfaccia (in questo caso `eth0`) con la configurazione specificata in `/etc/network/interfaces`.
- **`ifdown eth0`** – disabilitiamo l'interfaccia (in questo caso `eth0`).
- **`ifup -a`** – abilitiamo tutte le interfacce della sezione auto nel file di configurazione, nello stesso ordine.
  È eseguito all'avvio.

7.3 DOVE VEDIAMO SE ABBIAMO ACCESSO AL ROUTER DI DEFAULT? QUAL È IL SUO INDIRIZZO IP?

Per vedere se abbiamo accesso al router (o gateway) di default, abbiamo due modi:

1. **`ip route show`**, che ci darà un output del tipo

   ```
   default via 192.168.1.1 …
   ```

2. **Aprire il file di configurazione** e vedere se è presente la riga

   ```
   …
   gateway 192.168.1.1
   ```

In entrambi i casi, vediamo che l'indirizzo IP del gateway di default è 192.168.1.1.

Per aggiungere un accesso al router di default si usa il comando

```
ip route add default via 192.168.1.1
```

7.4 DNS

Il Domain Name System si occupa della **risoluzione dei nomi**, ovvero si occupa di tradurre nomi di host in indirizzi IP.

Per farlo utilizza un **database distribuito e gerarchico** su più server DNS.

Il client effettua una richiesta ad un server DNS, che può comportarsi in due modi:

1. **Conosce la risposta:** risponde al client con l'indirizzo IP.
2. **Non conosce la risposta:** inoltra la richiesta ad un server DNS più grande.

È quindi sufficiente conoscere almeno un indirizzo IP di un server DNS.

Gli IP dei server DNS che l'host può contattare sono scritto nel file di configurazione `/etc/resolv.conf`.

Un esempio di contenuto è:

```
nameserver 8.8.8.8

nameserver 8.8.4.4
```

che sono i DNS primario e secondario di Google.

7.5 DA DOVE SI VEDE SE IL DNS È CONFIGURATO?

Possiamo leggere il file di configurazione tramite il comando

```
cat /etc/resolv.conf
```

che produrrà un output del tipo

```
nameserver 8.8.8.8

nameserver 8.8.4.4
```

Per testare un singolo dominio usiamo invece il comando

```
nslookup nome_dominio
```

7.6 NSS (NAME SERVER SWITCH)

Il Name Server Switch è il meccanismo che i sistemi Unix usano per ricavare nomi di host da diverse fonti.

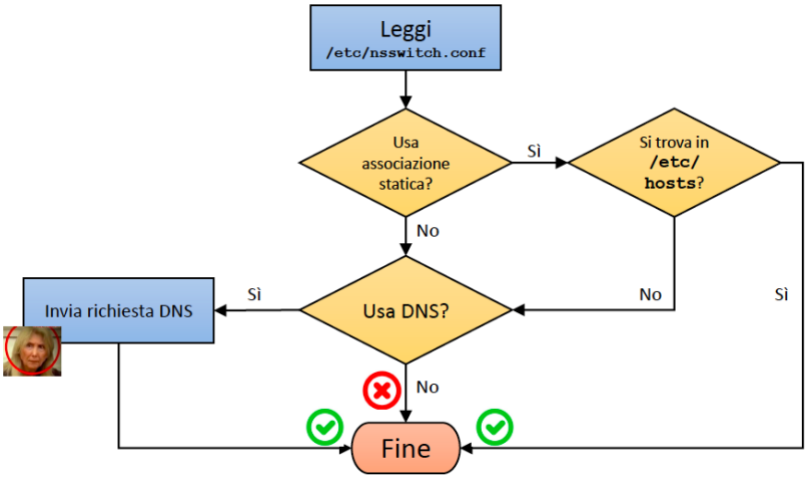
Il file di configurazione `/etc/nsswitch.conf` specifica le fonti da usare e in che ordine usarle. Un esempio di contenuto è:

...

```
hosts:      files dns
```

...

Il meccanismo è il seguente:



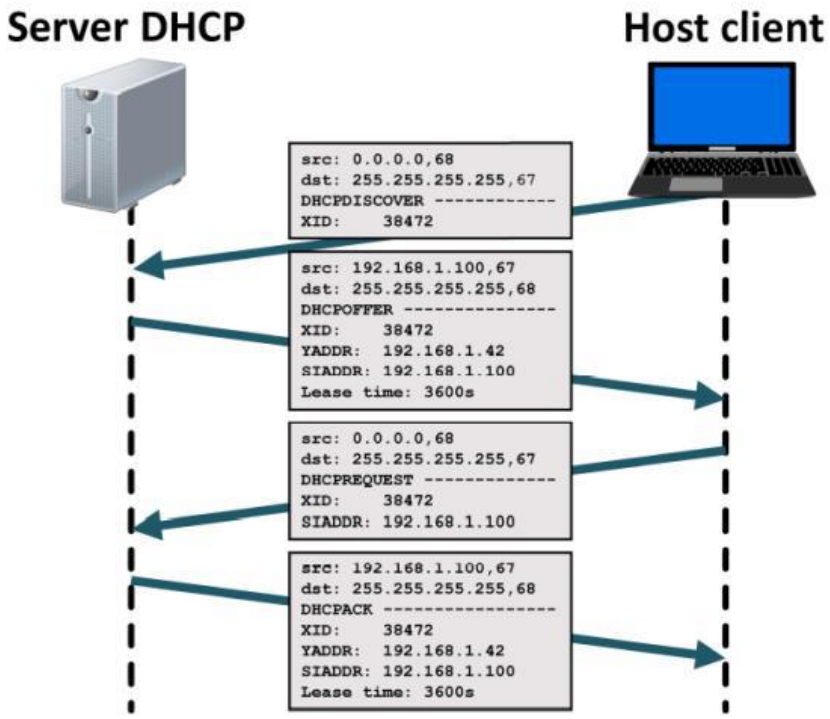
7.7 DHCP

Il DHCP permette la configurazione **automatica e dinamica** dei parametri TCP/IP degli host.

All'interno della rete c'è un server DHCP che configura i parametri di rete degli host: il server/router rileva i nuovi dispositivi che vogliono connettersi alla rete e gli assegna i 4 parametri necessari per la connessione.

7.7.1 COME FUNZIONA

1. Il dispositivo manda un segnale **DHCPDISCOVER** che verrà rilevato da più server DHCP, dato che `dst: 255.255.255.255`. Il sorgente è 0.0.0.0, 68 perché
   a. `0.0.0.0` è un indirizzo fittizio che indica che il dispositivo non ha ancora un indirizzo IP all'interno della rete.
   b. `68` è la porta riservata al server DHCP.
2. Il dispositivo riceve dal server la risposta **DHCPOFFER**, che sarà composta da:
   a. src: 192.168.1.100, 67 → Indirizzo IP del server DHCP e porta sulla quale è in ascolto il client.
   b. dst: 255.255.255.255, 68 → Il server manda il messaggio in broadcast perché l'host non può ancora essere indirizzato.
   c. YADD: 192.168.1.42 → Indirizzo IP offerto.
   d. Lease time: 3600s → TTL (Time To Live), è il tempo di vita dell'IP, ovvero la durata dei parametri di rete a partire dal momento in cui il server capisce che l'host ha accettato l'IP offerto.
      Serve per poter riutilizzare gli indirizzi IP inutilizzati.
   e. Il client accetta l'indirizzo IP proposto con **DHCPREQUEST**.
   f. Il server da l'ok con **DHCPACK** inviando le stesse informazioni del punto 2.



Il client manda DHCPREQUEST in broadcast perché, essendoci più server DHCP, ognuno di questi manderà la propria offerta di indirizzo IP. In questo modo, tutti i server sanno quale IP ha accettato l'host e possono controllare se corrisponde a quello che hanno inviato loro.

7.7.2 CONFIGURAZIONE SERVER DHCP

È possibile installare in una macchina UNIX un server DHCP, che si occuperà di assegnare indirizzi IP alle macchine.

1. Installiamo il server tramite il comando

   ```
   apt-get install isc-dhcp-server
   ```

2. Nel file di configurazione `/etc/default/isc-dhcp-server` scriviamo

   ```
   INTERFACES="eth0"
   ```

   per stabilire l'interfaccia sulla quale agisce il server DHCP.
3. Nel file di configurazione `/etc/dhcp/dhcpd.conf` andiamo a scrivere

   ```
   # server DNS
   option domain-name-servers 192.168.0.2, 8.8.8.8;
   default-lease-time 3600;
   # subnet -> sottorete; netmask -> maschera di sottorete.
   # Range di indirizzi IP dinamici gestiti dal server DHCP.
   # Tutti gli altri sono indirizzi statici o non gestiti dal server
   # DHCP in questione.
   subnet 192.168.0.0 netmask 255.255.255.0 {
        range 192.168.0.10 192.168.0.100
   }
   ```

4. Dopo le modifiche eseguiamo il comando

   ```
   systemctl restart isc-dhcp-server.service
   ```

7.7.3 CONFIGURAZIONE CLIENT DHCP

È sufficiente andare a modificare il file di configurazione `/etc/network/interfaces` sostituendo in particolare la parola `static` con la parola `dhcp` e facendo in modo che il file risulti

```
auto lo eth0

iface lo inet loopback

iface eth0 inet dhcp
```

7.8 COME VEDERE SE UNA MACCHINA HA INDIRIZZO STATICO O DINAMICO

Controllo nel file di configurazione `/etc/network/interfaces`:

1. Se l'interfaccia è settata in `static`, allora l'indirizzo è statico.
2. Se l'interfaccia è settata in `dhcp`, allora l'indirizzo è dinamico.

7.9 COME VEDERE SE LA MACCHINA È CONNESSA AD INTERNET? COME VERIFICO LA CONFIGURAZIONE?

Innanzitutto, bisogna controllare che sia presente ed attiva una configurazione tramite il comando

```
ip addr show
```

Dopodiché, per verificare la connessione possiamo usare il comando

```
ping nome_dominio
```

oppure il comando

```
traceroute nome_dominio
```

7.9.1 INTERNET CONTROL MESSAGE PROTOCOL

Sia `ping` che `traceroute` si basano su questo protocollo di servizio che **rileva malfunzionamenti, scambia informazioni di controllo e messaggi di errore.**

È incapsulato nel protocollo IP, nel campo ICMP dove troveremo i messaggi di errore.

7.9.2 PING

Testa la connettività tra l'host che lo esegue e un host remoto, contattandolo con brevi messaggi per assicurarsi che sia "still alive".

Iterativamente, l'host mittente invia ad un host destinatario un pacchetto ICMP **Echo Request**, attende in risposta un pacchetto ICMP **Echo Reply** e misura il tempo in millisecondi che il pacchetto ha impiegato ad arrivare al destinatario e tornare indietro.

Questo comando non è eseguibile verso reti private, perché sono protette da firewall e per comunicare con l'esterno usano una porzione di rete chiamata DMZ, dove rimangono eventuali problemi ed errori, senza propagarsi all'interno/esterno della rete.

Il **RTT (Round Trip Time)** cambia da pacchetto a pacchetto, per due motivi:

1. Cambia il traffico all'interno del percorso.
2. Il percorso seguito dai pacchetti non è sempre lo stesso.

Alla fine del comando, vengono mostrate delle statistiche sulla connessione, come il numero di pacchetti persi, il tempo di risposta ecc…

Inoltre, si possono aggiungere delle opzioni per specificare quanti pacchetti inviare, il timeout ecc…

Può capitare che al mittente non arrivi nulla (packet loss). Si può stabilire se il pacchetto è stato perso con Echo Request o con Echo Reply? No, possiamo solo stabilire che non c'è stata risposta.

PROVO A FARE PING AD UN DOMINIO MA NON RILEVO RISPOSTA. QUALI POSSONO ESSERE LE CAUSE?

Le cause possono essere:

1. **Network unreachable:** l'host locale non ha route valide per raggiungere l'host remoto.
2. **100% packet loss:** l'host locale non ha ricevuto alcun pacchetto in risposta.
3. **Unknown host:** non è stato possibile risolvere il nome dell'host remoto.

Vanno quindi controllare le seguenti cose:

1. **Presenza di una configurazione** tramite il comando

   ```
   ip addr show
   ```

2. **Accesso al gateway** controllando il file di configurazione

   ```
   /etc/network/interfaces
   ```

   oppure con il comando

   ```
   ip route show
   ```

   che ci mostra l'indirizzo del gateway.
3. **Accesso al DNS** controllando il file di configurazione

   ```
   /etc/resolv.conf
   ```

   oppure con il comando

   ```
   cat /etc/resolv.conf
   ```

   che restituisce gli indirizzi dei server DNS presenti. Senza DNS non possiamo risolvere i nomi degli host, e quindi non possiamo fare ping ad alcun dominio.
4. **Controllo del firewall** esaminando i file di configurazione

   ```
   /etc/hosts.allow, /etc/hosts.deny
   ```

   oppure tramite il comando

   ```
   iptables -L
   ```

   che restituisce le regole del firewall. Il firewall potrebbe avere una specifica regola per l'indirizzo IP richiesto, oppure potrebbe avere come regola di default il blocco di tutti i pacchetti in ingresso.
   La soluzione sarebbe quella di aggiungere una regola per consentire il traffico in ingresso, oppure rimuovere la regola di default.

7.10 traceroute

Oltre a dire se il destinatario ha risposto, mostra il **percorso** che un pacchetto IP effettua per raggiungere un host destinatario.

Il percorso è composto di fatto dagli indirizzi IP dei router attraversati.

7.10.1 COME FUNZIONA

Intanto, specifichiamo che TTL è il numero di apparati di rete che un pacchetto può attraversare prima di "scadere".

Ogni router che riceve un pacchetto, prima di inoltrarlo **decrementa di 1 TTL**. Quando si raggiungerà TTL = 0, il router invierà al mittente un messaggio ICMP **Time Exceeded** con l'indirizzo del router che ha portato TTL a 0.

1. Sfrutta la gestione del TTL, inviando all'host destinatario una serie di **terne di pacchetti UDP con TTL crescente.**
2. Tiene traccia degli indirizzi IP dei router che ha attraversato, mostrandoli in ordine all'interno dei messaggi ICMP **Time Exceeded** ricevuti, finché l'ultimo pacchetto non raggiunge il destinatario.
3. Il destinatario, quando viene raggiunto, risponde con un pacchetto ICMP **Destination Unreachable:** UDP ha bisogno di porte per inoltrare i pacchetti, e nello specifico i pacchetti verso il destinatario hanno bisogno di essere inoltrati tramite **porte di ascolto**. Scegliendone una a caso per far continuare il tragitto del pacchetto, quando si arriva al destinatario è improbabile trovarne una che sia effettivamente in ascolto, e quindi l'host invia Destination Unreachable con **code = 3**, ovvero **port unreachable** (l'host non sa a chi inoltrare il pacchetto).
4. Il comando ordina gli IP in base al TTL crescente e ricostruisce il percorso, fornendo anche il round-trip time dei pacchetti.

Nel caso in cui l'output siano 3 asterischi, vuol dire che è scattato il **timeout**, ovvero l'host mittente non ha ricevuto alcuna risposta per nessun pacchetto della terna. Le possibili cause sono:

1. Device non configurato per rispondere a traffico ICMP/UDP.
2. Pacchetti scartati per motivi di rete, ad esempio traffico bloccato da un firewall per non subire attacchi dall'esterno (DoS).

Uno dei principali **vantaggi** è che `traceroute` può essere usato per la diagnostica di problemi della rete, in quanto se ci sono problemi nei router intermedi questi sono facilmente rintracciabili (ogni volta che il mittente riceve un messaggio ICMP sa anche da chi proviene).

Ha tuttavia anche degli **svantaggi**:

1. I pacchetti possono seguire diversi percorsi, dunque gli indirizzi IP ottenuti possono riferirsi a più percorsi.
2. Se i pacchetti e i messaggi ICMP seguono percorsi diversi, il calcolo del round-trip time è inaffidabile.

7.11 PROBLEMA DELL'ENDIANNESS

Il problema riguarda l'ordine dei byte all'interno di una word: infatti, calcolatori diversi possono **posizionare in modi diversi i byte all'interno di una word.**

Esistono due tipi di memorizzazione:

1. **Big-endian:** per primo il byte più significativo (MSB).
2. **Little-endian:** per primo il byte meno significativo (LSB).

Il formato usato in rete è **big-endian**, mentre il formato usato dagli host **dipende dall'host**.

Abbiamo quindi bisogno di funzioni di conversione.

7.12 FUNZIONI DI CONVERSIONE

Per la conversione **da host a network**:

- `uint32_t htonl(uint32_t hostlong);`
- `uint16_t htons(uint16_t hostshort);`

Per la conversione **da network a host**:

- `uint32_t ntohl(uint32_t netlong);`
- `uint16_t ntohs(uint16_t netshort);`

I tipi `uint32_t` e `uint16_t` sono **interi senza segno** che occupano sempre 32 o 16 bit, a prescindere dall'host usato per compilare.

7.13 TIPI CERTIFICATI, FUNZIONI inet_pton E inet_ntop

7.13.1 TIPI CERTIFICATI

I tipi certificati, anche detti tipi **timbrati**, sono i tipi di dato segnati con `_t` e valgono come se fossero dei typedef: sono garantiti in termini di dimensioni e numero di byte, in modo da poter essere utilizzati su tutte le architetture.

COSA FARE SE NON LI USO? Dovrei far comunicare i 2 host con un protocollo di testo per scambiarsi i dati: convertendo i numeri in testo il char è sempre rappresentato con 1 byte, qualsiasi sia l'architettura usata.

7.13.2 inet_pton e inet_ntop

Queste due funzioni servono per rappresentare gli **indirizzi**. Abbiamo due formati:

- **Formato numerico**: 32 bit.
- **Formato presentazione:** stringa in notazione decimale puntata.

Per la conversione da presentazione a numerico abbiamo

```
int inet_pton(int af, const char* src, void* dst)
```

- `af`: famiglia dell'indirizzo (AF_INET).
- `src`: stringa del tipo "ddd.ddd.ddd.ddd".
- `dst`: puntatore ad un'istanza di struttura `in_addr`.

Per la conversione da numerico a presentazione abbiamo

```
const char* inet_ntop(int af, const void* src, char* dst, socklen_t
                      size)
```

- `af`: famiglia dell'indirizzo (AF_INET).
- `src`: puntatore ad un'istanza di struttura `in_addr`.
- `dst`: puntatore ad un buffer di caratteri di lunghezza `size`.

- `size`: dimensione dell'area di memoria che conterrà l'indirizzo in formato presentazione.

7.14 socket(), listen(), bind(), accept(), connect()

Un socket è un'**estremità di un canale di comunicazione** fra due processi in esecuzione su macchine connesse in rete.

7.14.1 socket()

Creazione di un socket: viene chiamata quando dobbiamo creare un punto di connessione tra un processo e l'altro.

```
int socket(int domain, int type, int protocol)
```

- `domain`: famiglia di protocolli da usare (AF_LOCAL – comunicazione locale, AF_INET – protocolli IPv4, TCP e UDP).
- `type`: tipologia di socket (SOCK_STREAM – connessione TCP, SOCK_DGRAM – connectionless, invio di pachetti UDP).
- `protocol`: sempre a 0.

Restituisce un **descrittore di file**, -1 in caso di errore, cioè un numero che identifica un socket aperto al processo e sul quale il processo può effettuare operazioni.

Il socket, dopo la chiamata, non è ancora associato né a un indirizzo IP né a una porta.

7.14.2 bind()

Assegna un indirizzo IP a un socket: specifica IP e **porta dove il server riceve richieste di connessione.**

Il client non ha bisogno di fare la bind: mentre il server ha bisogno di avere associata una porta "fissa" per poter essere contattato dai client, questi ultimi, facendo molto più spesso login e logout potrebbero avere ogni volta una porta diversa assegnatagli dal sistema operativo, che appunto in assenza di bind assegnerà ai client automaticamente una porta casuale.

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- `sockfd`: descrittore del socket restituito dalla `socket()`.
- `addr`: puntatore alla struttura `sockaddr` che si vuole associare al socket.
- `addrlen`: dimensione di `addr`.

Restituisce 0 se ha successo, -1 se si verifica un errore.

Il socket non è ancora utilizzabile, perché non è in ascolto.

7.14.3 listen()

Mette in **ascolto** il socket e crea una **coda di attesa** con un numero massimo di richieste: ascolta le richieste ma non può soddisfarle finché non viene chiamata la `accept()`.

Se arriva una richiesta in più rispetto al massimo, questa viene scartata.

```
int listen(int sockfd, int backlog)
```

- **`sockfd`**: descrittore del socket.
- **`backlog`**: dimensione della coda.

Specifica che il socket è **usato per ricevere richieste di connessione** (socket passivo).

È possibile mettere in attesa solo socket SOCK_STREAM: i socket UDP non devono instaurare alcuna connessione.

La funzione restituisce 0 se ha successo, -1 altrimenti.

7.14.4 accept()

Viene creato un ulteriore socket di comunicazione per collegare le richieste in coda con il server: è a questo socket che si collegherà il client.

La primitiva, quindi, **stoppa il processo fino all'arrivo di una richiesta di connessione** sul socket di ascolto creato con la bind, e accetta la prima in caso di eventuale coda di connessioni pendenti.

```
int accept(int sockfd, struct sockaddr* addr, socklen_t* addrlen)
```

- **`sockfd`**: descrittore del socket.
- **`addr`**: puntatore a struttura (vuota) dove viene salvato l'indirizzo del client.
- **`addrlen`**: puntatore alla dimensione di `addr`.

Anche in questo caso, questa primitiva ha senso solo per i socket SOCK_STREAM.

La funzione restituisce il **descrittore del nuovo socket** che sarà usato per lo scambio di dati con il client. Altrimenti, restituisce -1 in caso di errore.

7.14.5 connect()

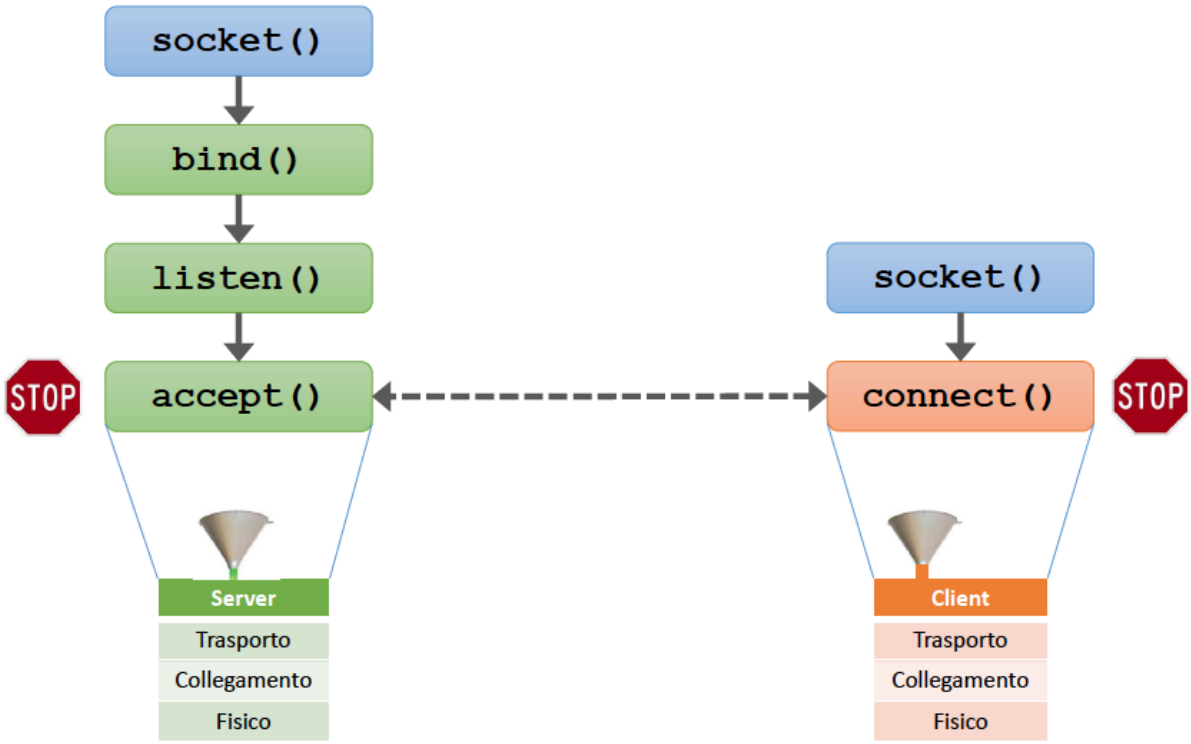
Permette a un socket locale di inviare una **richiesta di connessione** a un socket remoto.

int connect(int sockfd, struct sockaddr* addr, socklen_t *addrlen)

- **`sockfd`**: descrittore del socket.
- **`addr`**: puntatore alla struttura contenente l'indirizzo del socket remoto.
- **`addrlen`**: dimensione dell'indirizzo del socket remoto.

È bloccante: il programma si ferma in attesa che la richiesta di connessione sia accettata.

Restituisce 0 se ha successo, -1 se si verifica un errore.



7.15 send(), receive()

Usate per inviare e ricevere dati tramite un socket (il socket di comunicazione).

7.15.1 send()

Invia un messaggio attraverso un socket connesso.

```
ssize_t send(int sockfd, const void* buf, size_t len, int flags)
```

- `sockfd`: descrittore del socket.
- `buf`: puntatore al buffer contenente il messaggio da inviare.
  Area di memoria dove il programmatore salva il messaggio da inviare.
- `len`: dimensione del messaggio in byte.
- `flags`: opzioni di invio (per ora è a 0).

La funzione restituisce il **numero di byte inviati**, oppure -1 se c'è stato un errore.

È bloccante: il programma si ferma finché non ha scritto il messaggio (tutto o in parte) nel buffer di invio.

Se il numero restituito dalla funzione è diverso dal numero di byte da inviare, il programmatore deve richiamare nuovamente la primitiva a partire dall'ultimo byte inviato. In generale, dato che **a una `send()` corrisponde necessariamente una `recv()`, andrebbero messe in un loop che ha come condizione di uscita `ritorno della send() == dimensione memoria`.**

7.15.2 recv()

Riceve un messaggio da un socket connesso.

```
ssize_t recv(int sockfd, const void* buf, size_t len, int flags);
```

- **`sockfd`**: descrittore del socket.
- **`buf`**: puntatore al buffer in cui salvare il messaggio.
  Area di memorizzazione del socket.
- **`len`**: dimensione del messaggio desiderato in byte.
- **`flags`**: opzioni di ricezione.

La funzione restituisce il **numero di byte ricevuti**, -1 se ci sono errori oppure 0 se il socket remoto si è chiuso.

Il valore di ritorno 0 quindi non significa "ho ricevuto 0 byte".

È bloccante: il programma si blocca finché **non riceve almeno un byte**.

7.16 DIFFERENZE TRA SOCKET BLOCCANTI E SOCKET NON BLOCCANTI

I socket bloccanti sono quelli che si comportano come le funzioni di I/O standard, ovvero si bloccano fino a quando non hanno finito di eseguire l'operazione.

I socket non bloccanti invece non si bloccano, fanno altro mentre aspettano la risposta e ritornano un errore se non possono eseguire l'operazione.

Le azioni che possono essere svolte non devono coinvolgere i dati richiesti o l'interazione con un altro client nella stessa rete.

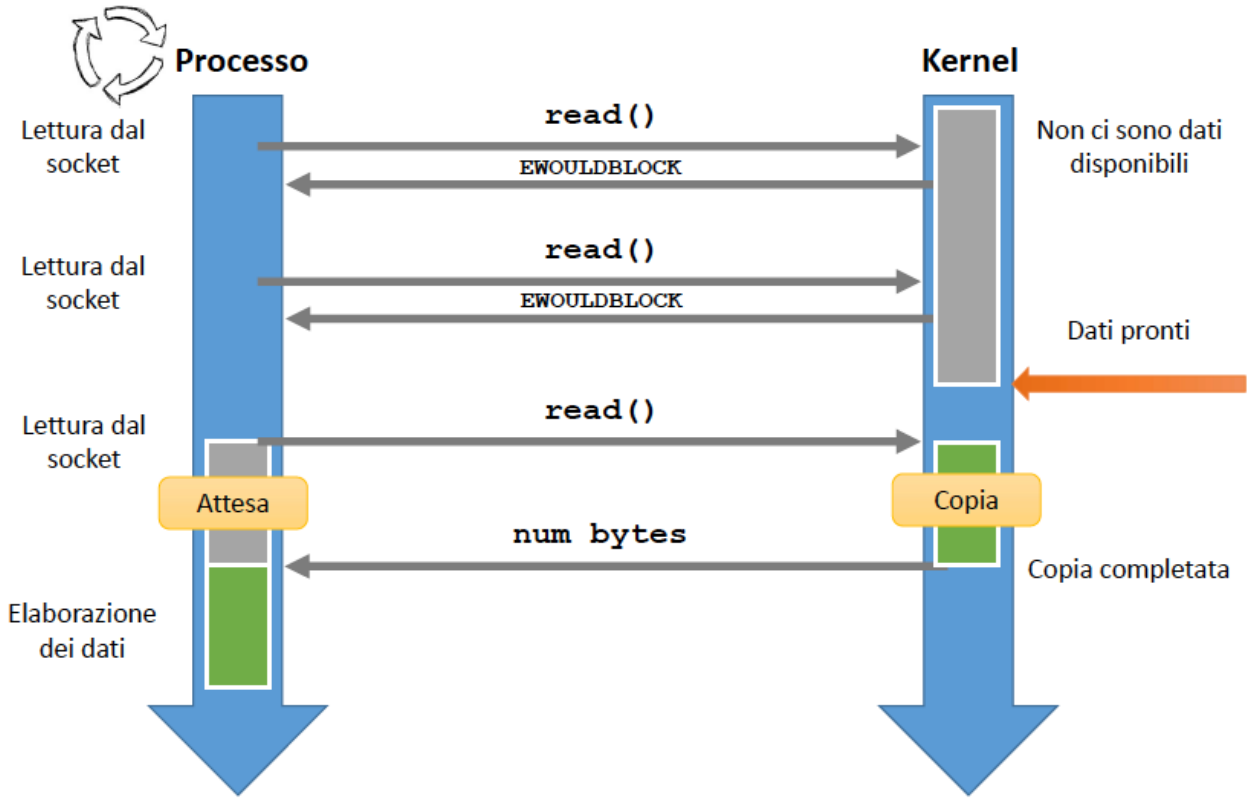
Per rendere un socket non bloccante si usa il parametro SOCK_NONBLOCK all'interno della `socket()`.

I cambiamenti sono i seguenti:

1. **`connect()`** – se non può connettersi restituisce -1 e imposta `errno` a EINPROGRESS.
2. **`accept()`** – se non ci sono richieste, restituisce -1 e imposta `errno` a EWOULDBLOCK.
3. **`send()`** – se non può inviare il messaggio, restituisce -1 e imposta `errno` a EWOULDBLOCK.
4. **`recv()`** – se non ci sono messaggi, restituisce -1 e imposta `errno` a EWOULDBLOCK.

In questo modo quando dovremo leggere dei dati eseguiremo una serie di `read()` fino a quando non ci verranno effettivamente consegnati, ma nel frattempo possono essere svolte altre operazioni.

Il tempo di attesa si riduce solamente all'attesa che i dati vengano copiati nel buffer e trasferiti.



7.17 fork()

Primitiva usata su server concorrenti per far creare al **sistema operativo** dei processi figli che si occuperanno di gestire le connessioni con i client. Il processo creato è un perfetto clone del padre, ed esegue il suo stesso codice. Il valore di ritorno è:

- **0** se siamo nel **processo figlio.**
- **Il PID del processo figlio** se siamo nel **processo padre.**
- **-1 in caso di errore.**

Possiamo utilizzare il valore di ritorno per differenziare il codice eseguito dal padre e dal figlio.

Visto che entrambi i processi condivideranno gli stessi descrittori di socket, è opportuno chiudere il **socket di comunicazione** nel processo **padre**, e chiudere il **socket di ascolto** nel processo **figlio**.

7.18 DIFFERENZE TRA SERVER CONCORRENTE E MULTIPLEXATO

Un server concorrente è un server che crea un processo figlio per ogni connessione, mentre un server multiplexato è un server che usa la primitiva `select()` per gestire più connessioni contemporaneamente.

Bisogna prestare attenzione all'uso dei processi figli perché essi hanno in comune con il padre tutte le strutture dati, e quindi possono accedere a risorse condivise: una gestione non ottimale del codice potrebbe portare a inconsistenze.

7.19 PROTOCOLLI TEXT E BINARY

Abbiamo due modi di inviare dati e messaggi:

1. **Protocolli text:** inviano messaggi in formato testo, ovvero prendo il testo e tutti i dati che devo inviare, li converto in una enorme stringa e la invio in un formato concordato tra mittente e destinatario.
2. **Protocolli binary:** invio direttamente le struttura dati, quindi invio i dati "grezzi", binari, così come sono rappresentati nella macchina, senza convertirli in testo.

I protocolli text sono più facili da implementare, ma sono più lenti e difficili da debuggare. I procotolli binary sono più veloci e facili da debuggare, ma sono più difficili da implementare, dal momento che abbiamo bisogno di convertire ogni volta i dati in base all'architettura in cui ci troviamo.

7.19.1 PROTOCOLLO TEXT

Dobbiamo innanzitutto convertire **da struttura a stringa** usando la funzione `sprintf()` per scrivere tutti i dati in un'unica stringa.

```
char buffer[1024];

struct temp{
      int a;
      char b;
}

struct temp t;

sprintf(buffer, "%d %c", t.a, t.b);
```

Il ricevente deve fare **parsing** del messaggio per passare **da stringa a struttura**: per farlo usiamo la funzione `scanf()` per leggere i dati.

```
struct temp t;

sscanf(buffer, "%d %c", &t.a, &t.b);
```

7.19.2 PROTOCOLLO BINARY

Dobbiamo definire messaggi con una struttura fissata, con campi che rappresentano l'informazione da scambiare.

- Ogni campo ha una lunghezza e un tipo che possa essere trasferito.
- Alcuni protocolli di tipo binario possono contenere campi di lunghezza variabile, in quel caso il protocollo definisce comunque una lunghezza massima dei messaggi.

Usiamo la funzione send() per l'invio dei dati, e la funzione recv() per ricevere i dati. Il protocollo ha bisogno dell'universalità dei dati, ovvero che i dati siano rappresentati allo stesso modo su tutti i sistemi. Per questo motivo utilizzo il tipo opaco uintN_t, che rappresenta un intero a N bit (8, 16, 32) senza segno. Se non lo utilizzassi andrei incontro a:

- **Endianness:** il formato dei dati è diverso su sistemi big-endian e little-endian (dovrei ogni volta convertire i dati con le opportune funzioni di conversione).

- **Padding:** il padding è diverso su sistemi a 32 bit e sistemi a 64 bit, potrebbero variare anche da compilatore a compilatore.
- **Dimensione dei dati:** la dimensione dei dati potrebbe essere diversa a seconda dell'architettura del sistema.

Un vantaggio dei protocolli text è che se devo lavorare principalmente con stringhe, non devo fare conversione di alcun tipo e non ho problemi di endianness, dato che lavoro con char. Tuttavia, con numeri grandi ho molta occupazione di memoria e i messaggi viaggiano in chiaro, quindi non ho sicurezza.

I vantaggi del protocollo binary invece sono che non devo fare parsing, ho meno occupazione di memoria con numeri grandi e ho molta più sicurezza.

7.20 SOCKET UDP

Non c'è differenza tra socket di ascolto e socket di comunicazione: il protocollo è di tipo connectionless; quindi, viene meno tutta la fase di instaurazione della connessione. I dati vengono inviati direttamente, senza alcuna garanzia della consegna e della correttezza dei dati ricevuti.

Il socket UDP è più veloce del socket TCP, ma è meno affidabile.

Il socket UDP è utilizzato per applicazioni che non necessitano di affidabilità, ma hanno bisogno di una risposta molto rapida (ad esempio giochi o il protocollo DNS), mentre il socket TCP deve essere usato per quelle applicazioni che hanno bisogno di elevata affidabilità, ma possono permettersi ritardi (ad esempio applicazioni bancarie o servizi di messaggistica).

Nelle due system call usate per i socket UDP si deve ogni volta specificare chi sia il destinatario e quale sia il socket dall'altro capo della connessione.

7.20.1 PRIMITIVA sendto()

Invia un messaggio attraverso un socket all'indirizzo specificato.

```
ssize_t sendto(int sockfd, const void* buf, size_t len, int flags, const struct sockaddr* dest_addr, socklen_t addrlen)
```

- `sockfd`: il descrittore del socket.
- `buf`: puntatore al buffer contenente il messaggio da inviare.
- `len`: dimensione in byte del messaggio.
- `flags`: opzioni di invio (a 0).
- `dest_addr`: puntatore alla struttura contenente l'indirizzo del destinatario.
- `addrlen`: lunghezza di `dest_addr`.

Restituisce il **numero di byte inviati**, oppure -1 in caso di errore.

Si suppone che con una chiamata si riesca ad inviare tutto il messaggio.

Volendo inviare un file, dobbiamo dividerlo in chunk e caricarli sul buffer di invio del socket: la funzione restituisce un valore inferiore di quello inserito.

Inviare qualcosa significa riversare il contenuto del buffer applicazione nel buffer kernel. Se si verifica un errore, a livello applicazione non ci interessa, perché ci pensa il protocollo di trasporto.

È bloccante: il programma si ferma finché non ha scritto tutto il messaggio.

7.20.2 PRIMITIVA recvfrom()

Riceve un messaggio attraverso un socket.

```
ssize_t recvfrom(int sockfd, const void* buf, size_t len, int flags,
struct sockaddr* src_addr, socklen_t addrlen)
```

- **`sockfd`**: descrittore del socket.
- **`buf`**: puntatore al buffer contenente il messaggio da ricevere.
- **`len`**: dimensione in byte del messaggio.
- **`flags`**: opzioni di ricezione (a 0).
- **`src_addr`**: puntatore a struttura vuota per salvare l'indirizzo del mittente.
- **`addrlen`**: lunghezza di `src_addr`.

Restituisce il **numero di byte ricevuti**, -1 in caso di errore, 0 se il socket remoto si è chiuso.

È bloccante: il programma si ferma finché non ha letto qualcosa.

7.21 I/O MULTIPLEXING: COME FUNZIONA

Il problema che si pone è il seguente: come gestire più connessioni contemporaneamente?

Il meccanismo dell'I/O multiplexing permette di **gestire più socket contemporaneamente** e di sapere quando uno di questi è pronto per essere letto o scritto.

Il socket può essere **pronto in lettura** se:

- C'è almeno un byte da leggere.
- Il socket è stato chiuso: `read()` restituirà 0.
- C'è un errore: `read()` restituirà -1.
- È un socket in ascolto e ci sono connessioni attive.

Il socket può essere **pronto in scrittura** se:

- C'è spazio per scrivere nel buffer.
- Il socket è stato chiuso.
- C'è un errore: `write()` restituirà -1.

Un **descrittore** di socket è un intero che va da 0 a `FD_SETSIZE`.

Un **insieme di descrittori**, detto **set**, si rappresenta con una variabile di tipo **`fd_set`** e si manipola con delle macro.

- **`FD_SET(int fd, fd_set* set)`** aggiunge il descrittore `fd` all'insieme `set`.
- **`FD_ISSET(int fd, fd_set* set)`** controlla se il descrittore `fd` è nell'insieme `set`.
- **`FD_CLR(int fd, fd_set* set)`** rimuove il descrittore `fd` dall'insieme `set`.

- **`FD_ZERO(fd_set* set)`** svuota l'insieme `set`.

7.21.1 PRIMITIVA select()

Controlla più socket contemporaneamente, rilevando quelli pronti. In particolare, capisce quali descrittori sono pronti in un insieme di descrittori e **toglie da questo insieme i descrittori che non sono pronti.**

Prende in input:

- **`int nfds`** – numero massimo di descrittori da controllare (numero del descrittore più altro tra quelli da monitorare + 1).
- **`fd_set* readfds`** – puntatore all'insieme di descrittori da controllare in lettura.
- **`fd_set* writefds`** – puntatore all'insieme di descrittori da controllare in scrittura.
- **`fd_set* exceptfds`** – puntatore all'insieme di descrittori da controllare in caso di errore.
- **`struct timeval* timeout`** – puntatore alla struttura del timeout.

Per quanto riguarda il timeout, abbiamo diversi casi:

- **`timeout = NULL`** → Attesa indefinita, fino a quando un socket non è pronto.
- **`timeout = { 10; 5; }`** → Attesa massima di 10 secondi e attesa minima di 5 microsecondi.
- **`timeout = { 0; 0; }`** → Attesa nulla, controlla i descrittori ed esce subito.

La funzione restituisce il **numero di descrittori pronti** (-1 in caso di errore) e **non il numero di uno dei descrittori pronti.**

È **bloccante:** si blocca finché uno dei descrittori controllati non è pronto o finché non scade il timeout.

COMPORTAMENTO DELLA select()

- **Prima** di chiamare la `select()`, occorre inserire i descrittori da monitorare nei set di lettura e scrittura.
- **Dopo** l'esecuzione di `select()`, i set di lettura e scrittura contengono i descrittori pronti.

UTILIZZO DI select(): per utilizzare la `select()` abbiamo bisogno di due set di descrittori.

- **`fd_set master`** – set principale con i descrittori che voglio monitorare, gestito dal programmatore con le macro.
- **`fd_set read_fds`** – set di lettura gestito dalla `select()`. È una copia del master sulla quale agisce la `select()`, perché elimina dal set i descrittori non pronti e quindi non sarei in grado di monitorarli in futuro se non ne avessi una copia.

7.22 FIREWALL

7.22.1 COS'E' UN FIREWALL E A COSA SERVE

Un firewall è un sistema hardware o software, o più in generale un **meccanismo di protezione**, che controlla le connessioni in ingresso e in uscita, analizzando delle **regole** che il pacchetto deve soddisfare per passare.

7.22.2 DOVE SI TROVA

Può essere implementato su due livelli:

1. **NETWORK LAYER:** packet filtering che filtra i pacchetti in ingresso/uscita operando a livello di **TCP/IP**, analizzando gli header IP, TCP e UDP.
2. **APPLICATION LAYER:** opera a livello applicazione, facendo **deep packet inspection**. Può comprendere i dati a livello applicazione, dunque può essere utilizzato per filtrare i pacchetti in base al contenuto dei dati, tenendo dunque conto del contesto del pacchetto.
   Più efficace ma fa uso di maggiori risorse computazionali.

7.22.3 TIPI DI FIREWALL

Il firewall può essere:

1. **STATEFUL:** controlla il flusso dei pacchetti e tiene traccia delle connessioni TCP e degli scambi UDP in corso, dunque controlla anche il loro ordine.
   Più efficace, ma più complesso e pesante rispetto al firewall stateless.
2. **STATELESS:** non controlla il flusso dei pacchetti, e ogni pacchetto viene analizzato in base a campi statici come indirizzo sorgente o destinazione. La decisione se far passare o meno un pacchetto è quindi presa in base al suo contenuto.

7.22.4 COME FUNZIONA

Il firewall contiene una **tabella di regole**. Ogni regola sta su una riga, ha un indice progressivo che parte da uno e contiene:

- **Criteria:** caratteristiche del pacchetto (IP Sorgente, Porta Sorgente, IP Destinatario, Porta Destinatario).
- **Target:** azione da eseguire su quel pacchetto, ovvero scarta (DROP) o accetta (ACCEPT).

Indice	IP sorgente	Porta sorgente	IP destinatario	Porta dest.	Azione
1	131.114.0.0/16		131.114.54.4	80	SCARTA
2	0.0.0.0	23	112.143.2.2		ACCETTA

Per ogni pacchetto, **scorre le regole** in ordine crescente di indice e quando ne trova una i cui criteria corrispondono ai criteria del pacchetto, **applica quella sola regola**, poi va ad analizzare il pacchetto successivo.

Se nessuna regola soddisfa i criteri, viene applicata la **regola di default**. La regola di default determina il tipo di firewall:

1. **INCLUSIVO:** la regola di default **blocca tutto**.
   È sicuro ma scomodo, dato che se non si definiscono altre regole non si può accedere a nulla.
2. **ESCLUSIVO:** la regola di default **consente tutto**.
   È comodo ma non sicuro, perché devo prevedere e inserire manualmente tutte le regole che ritengo utili.

7.22.5 COME SI CONFIGURA

Si configura tramite il comando **`iptables`**.

7.23 iptables

`iptables` è un programma da linea di comando che permette di configurare il **`netfilter`**. Quest'ultimo è il componente del kernel di Linux che si occupa del firewall e che offre le funzionalità di:

1. **Stateless/stateful packet filtering:** permette di filtrare i pacchetti in ingresso/uscita.
2. **NA[P]T/PAT:** permette di fare NAT e port forwarding.
3. **Packet mangling:** permette di modificare i pacchetti in ingresso e uscita.

`iptables` lavora su diverse tabelle, ognuna dedicata a una funzionalità. Le tabelle che ci interessano sono:

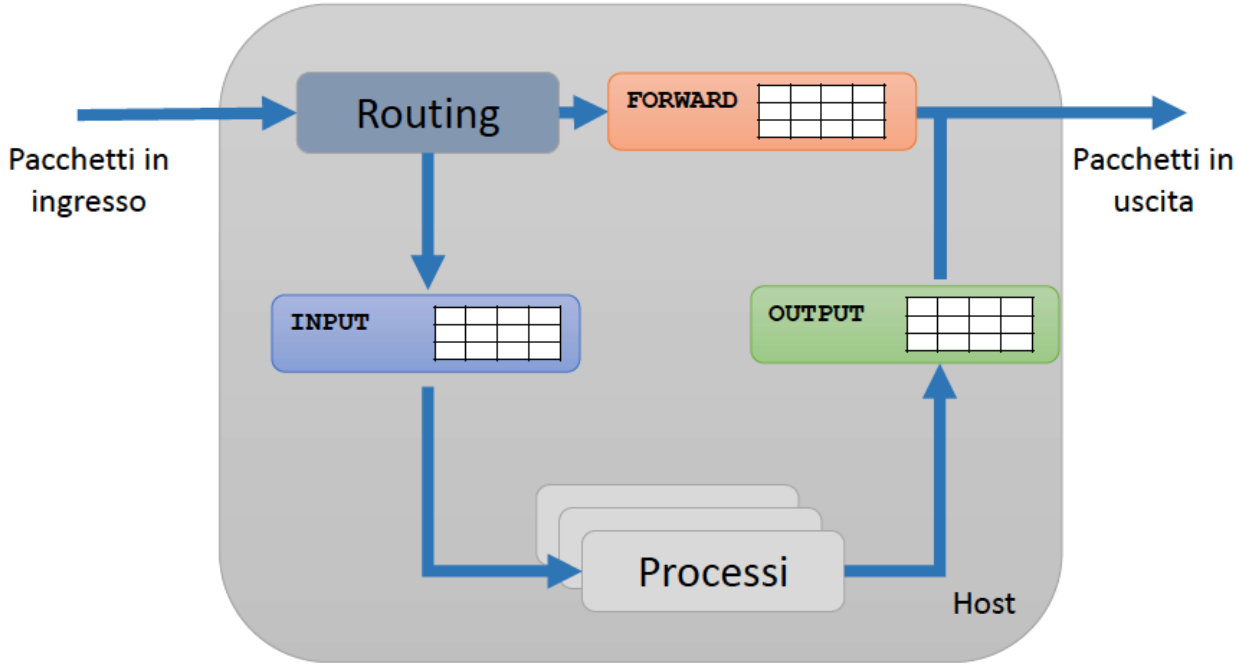
- **`filter`:** packet filtering.
- **`nat`:** si occupa del NAT e del port forwarding.

Ogni tabella contiene diverse catene (**chain**), e ognuna di esse contiene una lista di regole da applicare ad una determinata categoria di pacchetti, in modo da raggruppare le regole in dipendenza di cosa il pacchetto sta facendo.

7.23.1 TABELLA filter

La tabella `filter` ha 3 chain:

1. **INPUT:** per i pacchetti in ingresso destinati ai processi locali.
2. **OUTPUT:** per i pacchetti in uscita dai processi locali.
3. **FORWARD:** per i pacchetti in transito, da inoltrare ad altri host.



7.23.2 VISUALIZZARE LE REGOLE

Per visualizzare le regole usiamo il comando

```
iptables [-t table] -L [chain]
```

- Se la tabella non viene specificata, la tabella selezionata è `filter`.
- Se la catena non viene specificata, vengono elencate tutte le catene.

Per vedere le regole di una tabella diversa da `filter` si usa il flag `-t`, mentre `-L` serve per vedere tutte le regole rispetto alle catene scelte.

7.23.3 AGGIUNGERE LE REGOLE

Per l'aggiunta di una regola si usa il comando

```
iptables [-t table] -A chain rule-specification
```

La regola, descritta da `rule-specification`, viene aggiunta in fondo alla catena. Per aggiungere una regola in una posizione specifica, si usa il comando

```
iptables [-t table] -I chain [num] rule-specification
```

Se `num` non è specificato, la regola è posta per prima nella catena.

7.23.4 RIMUOVERE LE REGOLE

Per rimuovere una regola dalla catena si usa il comando

```
iptables [-t table] -D chain rule-specification
```

oppure

```
iptables [-t table] -D chain num
```

Per rimuovere tutte le regole da una o più catene si usa il comando

```
iptables [-t table] -F [chain]
```

Per cambiare la regola di default si usa il comando

```
iptables [-t table] -P target
```

7.23.5 ESEMPIO: BLOCCARE CONNESSIONI VERSO SITI WEB

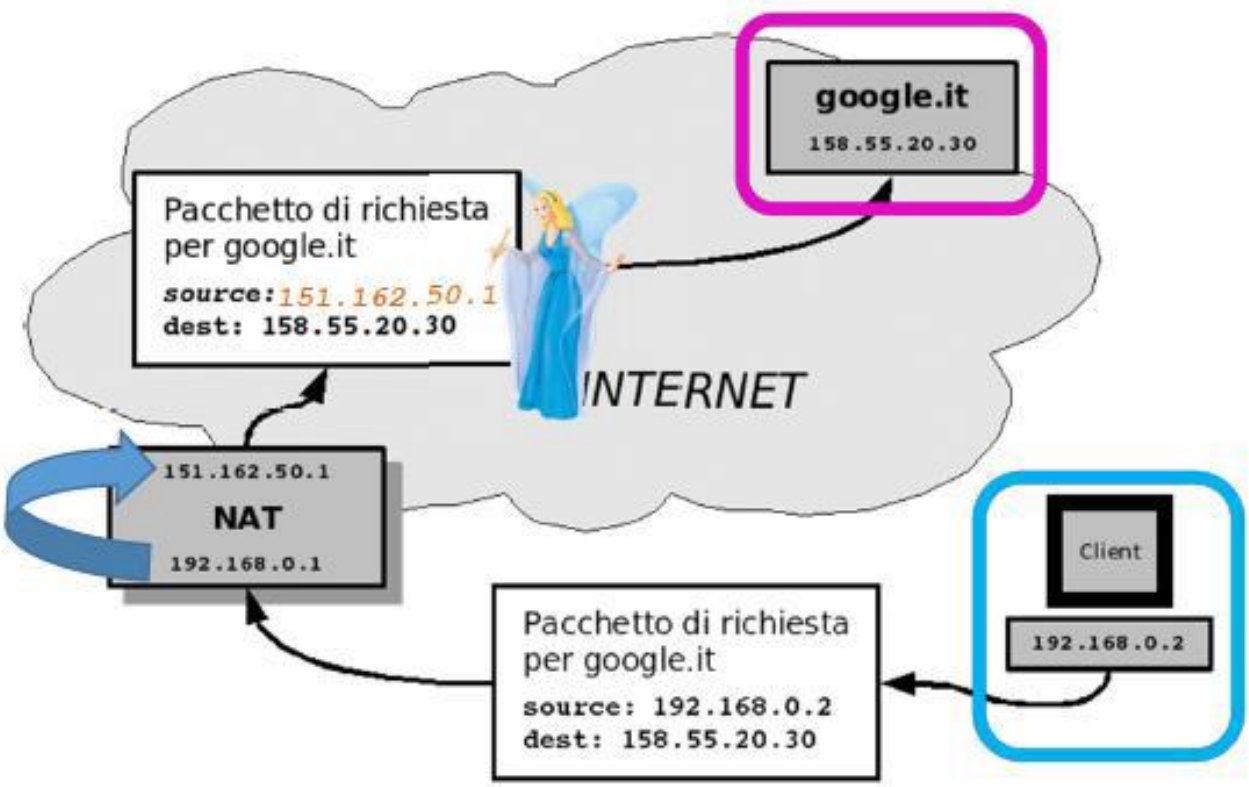
```
# iptables -A OUTPUT -p tcp -d 10.0.5.4 --dport 80 -j DROP
```

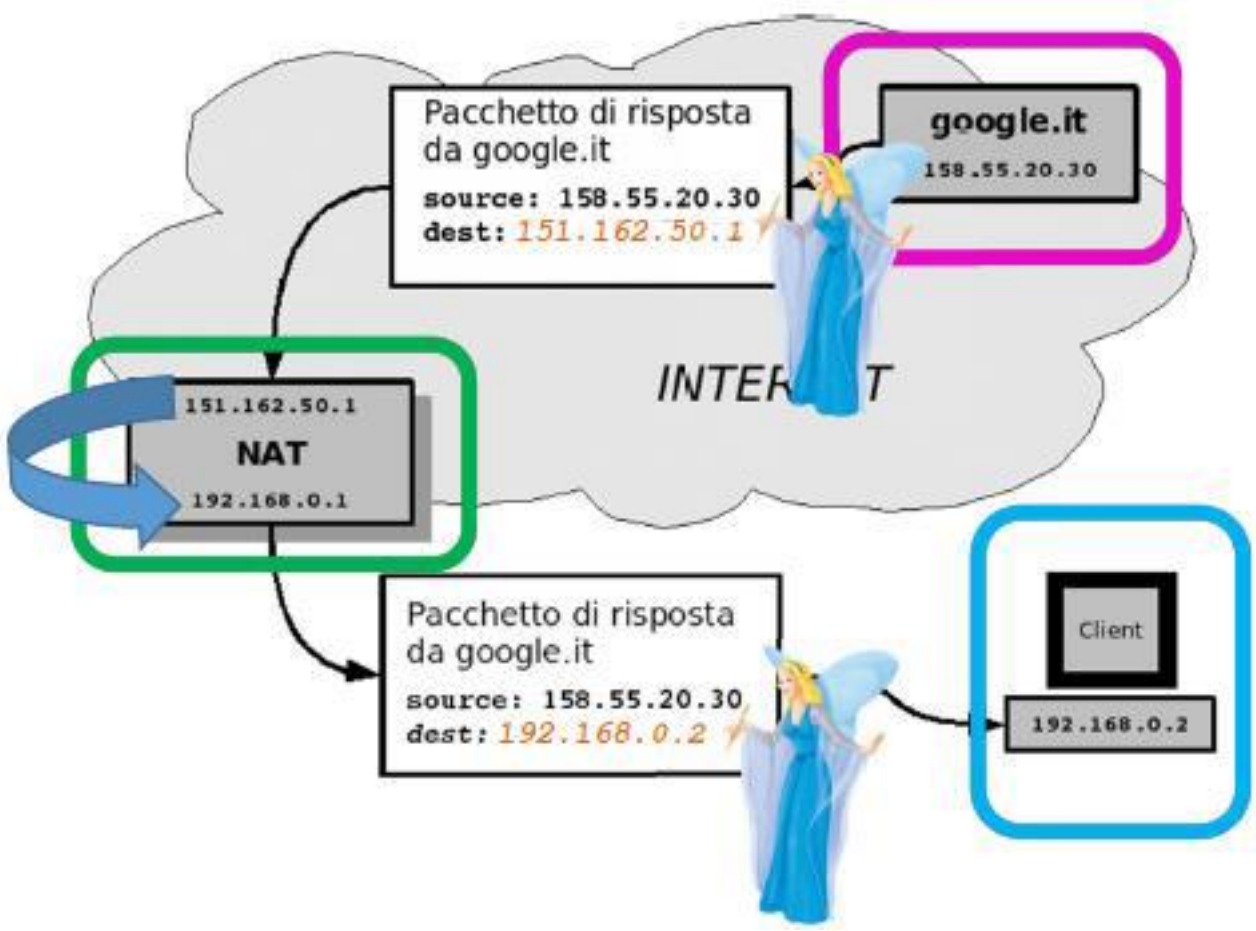
Aggiunge in fondo alla catena OUTPUT della tabella filter una regola che scarta tutti i pacchetti TCP destinati alla porta 80 dell'host 10.0.5.4.

7.24 NAT E PAT

Il NAT (Network Address Translation) è un meccanismo che permette di tradurre un indirizzo IP in un altro indirizzo IP. In particolare, vista la scarsità di indirizzi IP a disposizione si è cercato un modo per "fare economia":

- Viene assegnato a ciascun cliente **un unico IP per il traffico Internet.**
- Nella rete del cliente, ogni host riceve un IP unico, usato per instradare il traffico interno.
- Quando un pacchetto sta per lasciare la rete locale, **si esegue la traduzione** dall'indirizzo interno privato a quello pubblico.



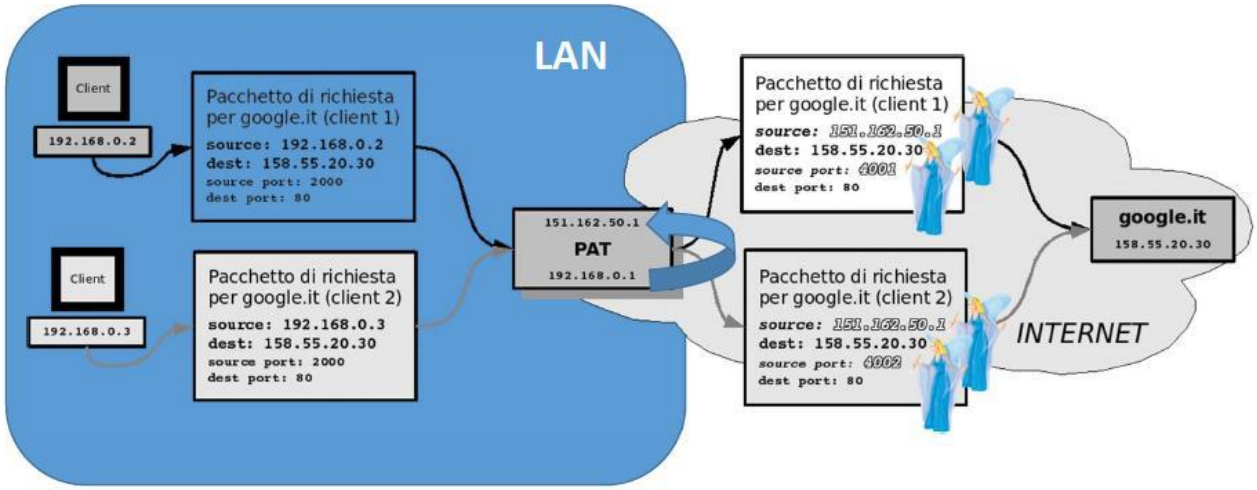


Il PAT (Port Address Translation) permette di tradurre una porta in un'altra porta. In particolare, permette di **mappare più host della LAN in un solo IP**.

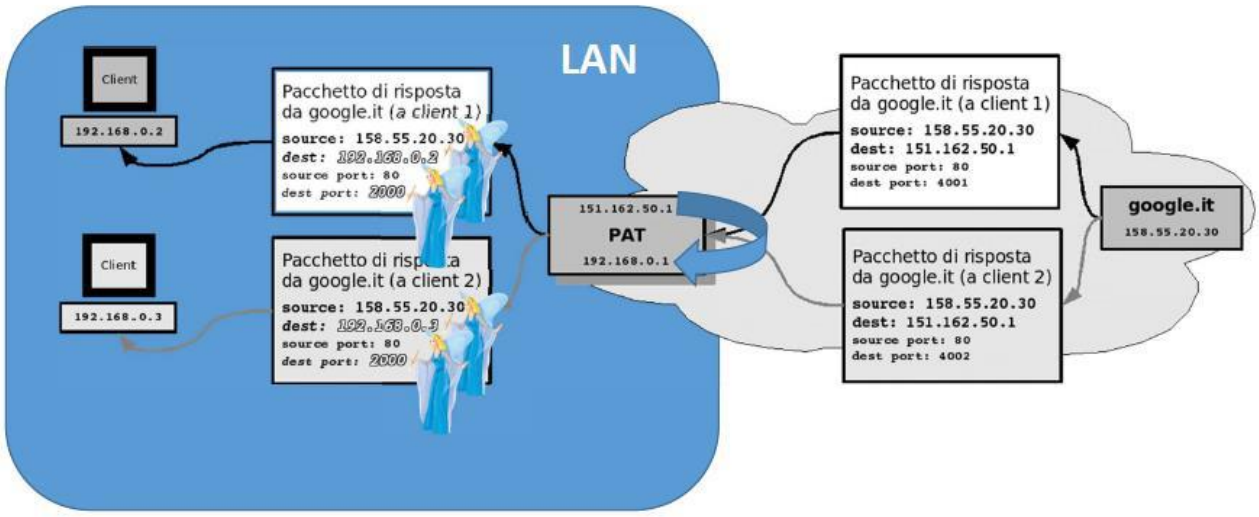
Perché si è reso necessario introdurre il meccanismo del PAT? Con più host dentro la stessa rete, se più host identificati dalla stessa porta vogliono inviare un messaggio allo stesso server web, quando quest'ultimo risponde il router deve sapere a chi mandare le informazioni, e senza il PAT non avrebbe modo di distinguere i due host.

Con PAT, un pacchetto avente **porta pubblica** è tradotto in un pacchetto avente una **porta privata**.

La porta sorgente interna (2000) viene cambiata con due nuove porte esterne. La 4001 identifica il client 192.168.0.2, mentre la 4002 identifica il client 192.168.0.3.



Quando Google invierà le due risposte all'unico IP pubblico da cui vede provenire le richieste il router potrà capire a quale il client della LAN deve inoltrare ciascuna risposta.



Il meccanismo può essere gestito tramite `iptables`, che lavora sulla tabella `nat`. La tabella `nat` ha 3 chain:

1. **PREROUTING:** i pacchetti in ingresso vengono modificati prima di essere processati.
2. **OUTPUT:** i pacchetti uscenti dai processi locali vengono modificati prima di essere inviati.
3. **POSTROUTING:** i pacchetti in uscita vengono modificati dopo essere stati processati.

Un esempio di regola di NAT è

```
iptables -t nat -A POSTROUTING -s 192.168.0.2 -j SNAT --to-source
151.162.50.1
```

- `-t nat` → Si seleziona la tabella `nat`.
- `-A POSTROUTING` → Si aggiunge una regola alla chain `POSTROUTING`.
- `-s 192.168.0.2` → Si applica la regola solo ai pacchetti che hanno come sorgente l'indirizzo IP scritto.

- `-j SNAT` → Si esegue una SNAT.
- `--to-source 151.162.50.1` → Si traduce l'indirizzo IP sorgente in quello scritto.

7.25 apache2

Apache HTTP Server è un web server, implementato tramite il programma **apache2**, che permette di gestire più richieste HTTP e HTTPS contemporaneamente.

7.25.1 COME FARLO PARTIRE

Per farlo partire possiamo usare due comandi:

1. `apache2ctl <comando>` dove comando può essere `start, stop, restart, status e configtest`.
2. `service apache2 <comando>` dove comando può essere `start, stop, restare e reload`.

7.25.2 FILE DI CONFIGURAZIONE

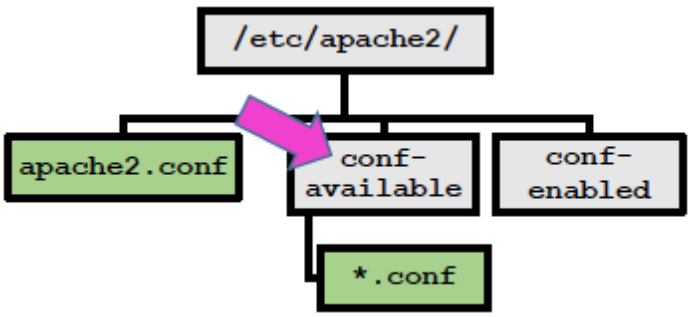
Nella directory principale `/etc/apache2/` abbiamo il file di configurazione

`/etc/apache2/apache2.conf`

che è un file modulare: la configurazione si fa tramite **direttive**, eventualmente raggruppate in **direttive contenitore.**

Il file di configurazione recupera le varie **parti di configurazione** (in formato .conf) da altri file, tramite la direttiva `Include`**.**

Le **parti di configurazione** si mettono nella directory `/etc/apache2/conf-available`:



Un file si abilita con

`a2enconf <nome_file>`

andando a creare un soft link nella directory `/etc/apache2/conf-enabled`.

Un file si disabilita con

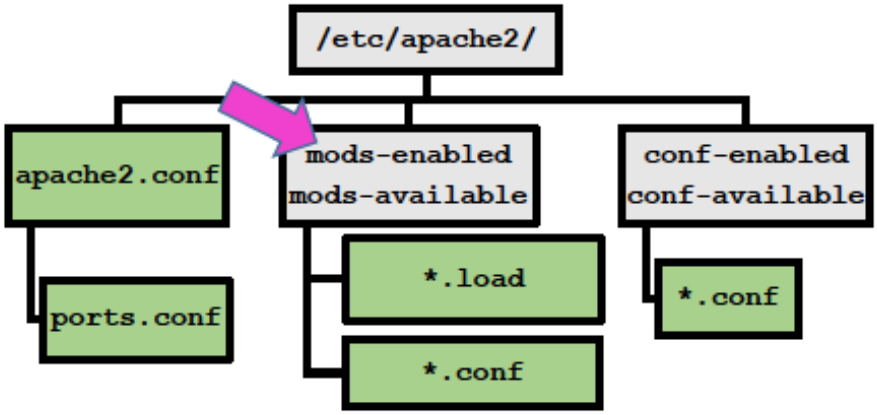
`a2disconf <nome_file>`

Ogni volta che si modifica il file di configurazione di apache, **per rendere le modifiche effettive bisogna riavviare il server**.

7.25.3 STRUTTURA

La struttura di Apache è basata su moduli, che si trovano nella directory `/etc/apache2/mods-available/` e si abilitano e disabilitano tramite i comandi

- `a2enmod <nome_file>`
- `a2dismod <nome_file>`



Un modulo abilitato ha un soft link in `mods-enabled` e, come prima, per ricaricare la configurazione è necessario **riavviare il server.**

7.25.4 DIRETTIVE

Le direttive servono a specificare **opzioni globali**, valide per l'intero server.

- `ServerRoot /etc/apache2` – specifica la directory principale dei file di configurazione di Apache.
  Su Debian è configurata automaticamente all'avvio del servizio.
- `KeepAlive on` – specifica se offrire o meno le connessioni persistenti di HTTP 1.1.
- `KeepAliveTimeout 5` – specifica quanti secondi attendere la successiva richiesta dal client su una stessa connessione prima di chiuderla.
- `Listen 80` – specifica le porte su cui Apache si mette in ascolto di connessioni.
  È obbligatoria ed è possibile specificare più porte.
- `ErrorLog /var/log/apache2/error.log` – specifica il file di log degli errori.

7.25.5 VIRTUAL HOST

Con il Virtual Hosting si possono configurare più siti sullo stesso server web, sulla stessa macchina, con lo stesso indirizzo IP.

Il server discrimina le richieste dei client in base al campo `Host` della richiesta HTTP.

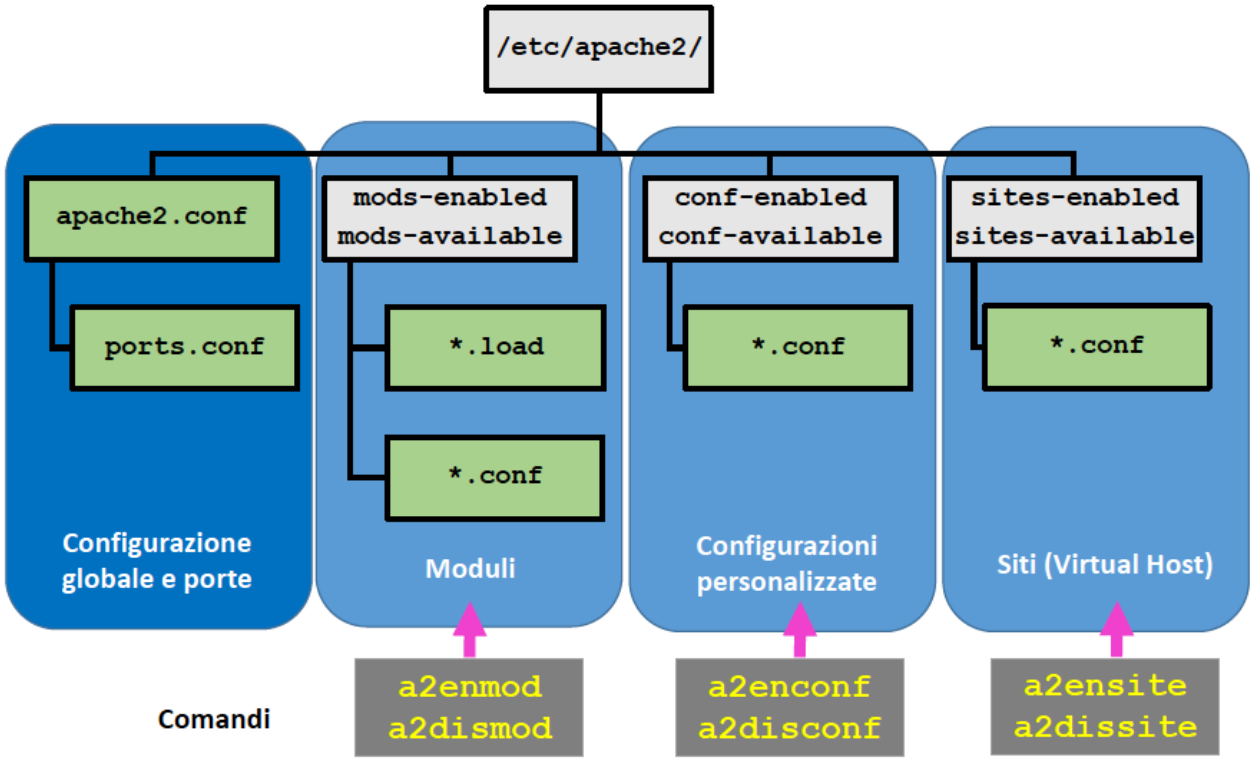
Questa configurazione può essere eseguita nei file contenuti nella directory `/etc/apache2/sites-available`. I siti web possono essere abilitati/disabilitati con i comandi

- `a2ensite <nome_file>`
- `a2dissite <nome_file>`

Nel caso più semplice Apache ha un **default Virtual Host** abilitato in /etc/apache2/sites-available/000-default.conf, dove troviamo delle direttive contenitori

```
<VirtualHost *:80>
        ServerName www.example.com
        …
        DocumentRoot /var/www/html
</VirtualHost>
```

- `<VirtualHost ip:porta>` … `</VirtualHost>` serve per definire un virtual host.
- `ServerName www.example.com` specifica il nome simbolico del sito. Serve a discriminare le richieste fatte dai client ai vari virtual host del server.
- `DocumentRoot /var/www/html` specifica la directory dei file del sito.



7.25.6 MULTI-PROCESSING MODULES

Apache accetta e serve più richieste contemporaneamente tramite i **moduli Multi-Processing Module (MPM).** Questo viene fatto tramite

1. Gestione dei socket.
2. Binding delle porte.
3. Processazione delle richieste usando processi figli e thread.

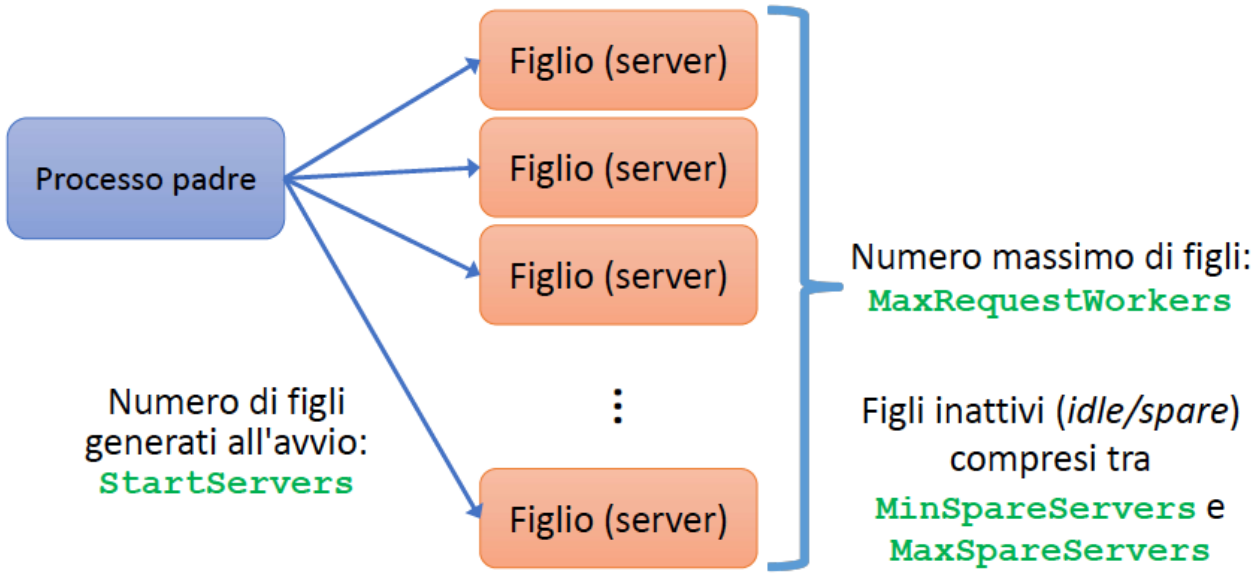
MPM PREFORK: implementa un server multi-processo senza thread.

All'avvio, un processo padre lancia un certo numero di processi figli (procedimento chiamato **preforking**):

- I figli, chiamati **worker**, restano in ascolto, accettano le connessioni e le servono.
- Dopo aver servito una connessione, il worker torna disponibile.

Il padre gestisce il pool dei figli cercando di mantenerne sempre alcuni disponibili. Il preforking all'avvio e il riutilizzo dei vari worker evitano l'overhead della `fork()` a ogni connessione.

Ogni figlio viene riciclato per **`MaxConnectionsPerChild`** connessioni, poi viene terminato, per evitare memory leak accidentali.



I vantaggi sono:

1. **Massima compatibilità:** funziona anche con i moduli e le librerie che non supportano il multithreading.
2. **Massima stabilità:** alla caduta di un worker, cade una sola connessione.

Gli svantaggi sono:

1. **Grande occupazione di memoria:** ogni worker ha una copia di tutto il codice del padre, essendo processi.
2. **Complessità nel regolare le impostazioni:** troppi processi inattivi occupano inutilmente memoria, troppi pochi causano più overhead da `fork()` in caso di picchi di richieste.

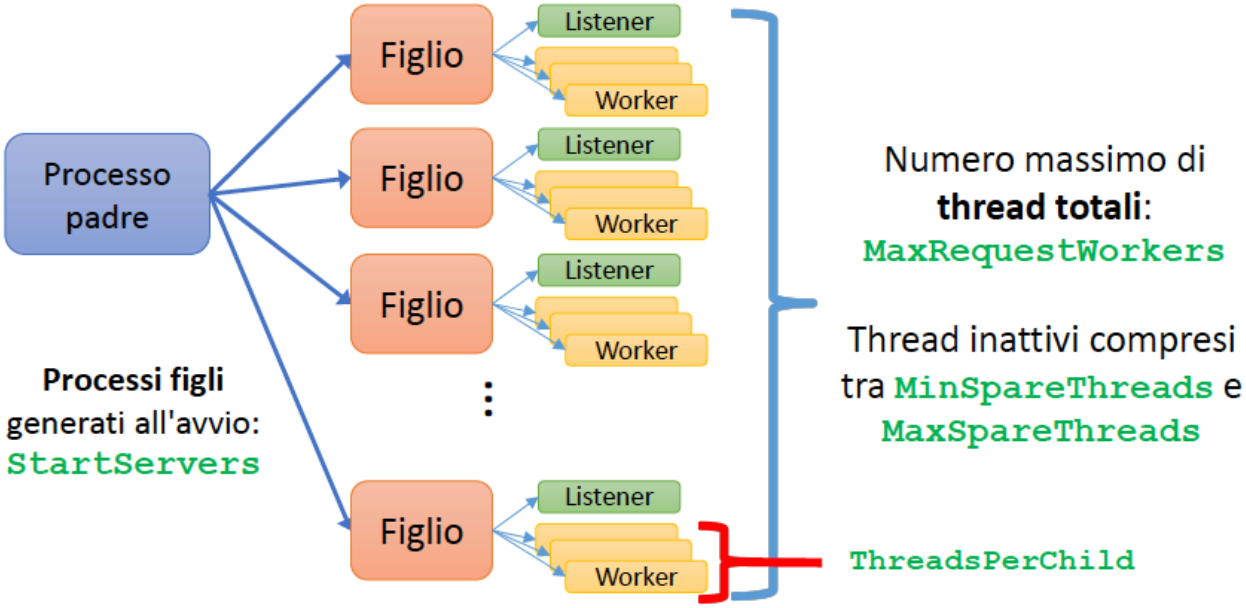
MPM WORKER: rende il server sia multiprocesso che multithread.

Il porcesso padre genera un certo numero di processi figli (prefork). **Ogni processo figlio genera:**

1. Un thread **listener** che accetta/smista le connessioni.
2. Un certo numero di thread **worker** che servono le richieste.

Abbiamo un overhead ridotto (grazie al preforking) e un risparmio di energia (grazie ai thread in caso di autotuning, ovvero quando c'è bisogno di aprire altri thread in caso quelli presenti non siano sufficienti).

- Ogni processo figlio viene riciclato per `MaxConnectionsPerChild` connessioni.
- Il numero massimo di figli è necessariamente `MaxRequestWorkers/ThreadsPerChild`.
- È un metodo stabile perché alla caduta di un worker cade una sola connessione.



MPM EVENT: versione migliorata di mpm worker, ed è il metodo di default in apache 2.4.

Oltre ad accettare connessioni, il **listener** è in grado di gestire le connessioni **temporaneamente inattive:**

- Se un worker è connesso a un client che **tarda ad inviare una richiesta**, invece di attendere restituisce il controllo del socket al listener e passa a seguire un altro client. Quando il primo client invierà la richiesta, il listener la assegnerà a un altro worker libero.
- Se un worker sta servendo un client con una **connessione lenta** e il buffer di invio del socket si riempie, invece di attendere restituisce il controllo del socket al listener che lo assegnerà ad un altro worke non appena sarà di nuovo libero il buffer.

Con questa politica il numero di connessioni servite contemporaneamente, a parità del numero di thread, aumenta, perché eliminiamo i tempi morti.