Reti di calcolatori

VR443470

gennaio 2023

Indice

1 Introduzione

Internet è una rete di calcolatori che interconnette miliardi di dispositivi di calcolo in tutto il mondo. Gli strumenti in una rete, per esempio cellulari o computer, vengono chiamati **host** (*ospiti*) o **sistemi periferici** (*end system*). Essi sono connessi tra di loro tramite una **rete di collegamenti** (*communication link*) e **commutatori di pacchetti** (*packet switch*). I collegamenti possono essere di vario tipo: cavi coassiali, fili di rame, fibre ottiche e onde elettromagnetiche.
Ogni collegamento detiene una sua **velocità di trasmissione** (*transmission rate*), ovvero la velocità di trasmissione dei dati. L'**unità di misura** è il bit per secondo (bit/secondo, *bps*).

L'insieme delle informazioni, o dati, che vengono inviati o ricevuti prendono il nome di **pacchetto**. L'**obbiettivo** di un commutatore di pacchetti è quello di ricevere un pacchetto che arriva da un collegamento in ingresso e di ritrasmetterlo su un collegamento d'uscita. I due principali commutatori di internet sono: *router* e i commutatori a livello di collegamento (*link-layer switch*). La sequenza di collegamenti e di commutatori di pacchetto attraversata dal singolo pacchetto è nota come **percorso** o **cammino** (*route* o *path*).

Quindi, in sintesi, le definizioni più rilevanti sono:

- **Internet.** Rete di calcolatori che interconnette i dispositivi di calcolo di tutto il mondo.
- **Host (*o* sistemi periferici).** Strumenti in una rete, per esempio computer.
- **Rete di collegamenti (*communication link*) e commutatori di pacchetto (*packet switch*).** Collega vari *host*, per esempio cavi coassiali o fili di rame.
- **Velocità di trasmissione (*transmission rate*).** È la velocità di trasmissione dei dati e solitamente la sua **unità di misura** è il bit per secondo, cioè *bps*.
- **Pacchetto.** Insieme delle informazioni che vengono inviate e ricevute.
- **Obbiettivo** commutatore di pacchetti. Ricevere un pacchetto proveniente da un collegamento in ingresso e ritrasmetterlo su un collegamento d'uscita. Per esempio i *router*.
- **Percorso (*route*) o cammino (*path*).** Sequenza di collegamenti e di commutatori di pacchetto attraversata dal singolo pacchetto.

2 ISP, TCP/IP, commutazione dei pacchetti e ritardi

2.1 ISP

I sistemi periferici accedono ad Internet tramite un servizio chiamato **Internet Service Provider** (ISP). Con **provider** si intende un insieme di commutatori di pacchetto e di collegamenti. Gli **obbiettivi** degli ISP è fornire ai sistemi periferici svariati tipi di accesso alla rete, come quello residenziale a larga banda (e.g. DSL), quello in rete locale ad alta velocità, quello senza fili (*wireless*) e in mobilità.
Esistono 3 **tipi** di livelli di ISP:

Livello 1. *Internazionale* (Telecom, TIM, ...);

Livello 2. *Nazionale* (Fastweb);

Livello 3. *Locale* (solitamente per professionisti).

Più è basso il livello, più gli ISP sono costituiti da *router* ad alta velocità interconnessi tipicamente tramite fibra ottica.

2.2 TCP/IP

I sistemi periferici, i commutatori di pacchetto e altre parti di Internet fanno uso di **protocolli** che controllano l'invio e la ricezione di informazioni all'interno della rete. Esistono **due principali protocolli** Internet: ***Transmission Control Protocol*** (TCP) e ***Internet Protocol*** (IP). In particolare, l'IP specifica il formato dei pacchetti scambiati tra router e sistemi periferici. Generalmente ci si riferisce a questi due protocolli tramite il nome collettivo TCP/IP.

2.3 Commutazione dei pacchetti

Esistono due diversi approcci per spostare quantità di dati all'interno di una rete: la **commutazione di circuito** e la **commutazione di pacchetto**.

Commutazione di circuito

Nella **commutazione di circuito** le risorse richieste lungo un percorso (buffer e velocità di trasmissione sui collegamenti) sono **riservate** per l'intera durata della sessione di comunicazione.
Vantaggi:

- ✓ **Velocità costante** durante il collegamento poiché le risorse sono riservate e non condivise. Questo si traduce in un **ritardo contenuto**.

Svantaggi:

- ✗ **Spreco di risorse** poiché i circuiti sono inattivi durante i periodi di silenzio, ovvero nei periodi in cui non c'è comunicazione;
- ✗ **Complicazioni** nello stabilire circuiti e nel riservare larghezza di banda *end-to-end*.

In questo contesto, i ritardi possono essere causati solamente per tre motivi: (1) a causa dell'instaurazione del circuito, (2) a causa della distanza tra sorgente e destinazione, (3) a causa della trasmissione vera e propria.

Commutazione di pacchetto

Nella **commutazione di pacchetto** la sorgente divide i messaggi in parti più piccole, ovvero in **pacchetti** assegnando a ciascuno un'intestazione. I pacchetti viaggiano attraverso collegamenti e commutatori di pacchetto dalla sorgente alla destinazione.
Vantaggi:

- ✓ **Ottimizzazione** delle risorse poiché c'è una condivisione di esse nei momenti di inattività.

Svantaggi:

- ✗ **Possibile perdita** di pacchetti nel caso in cui un buffer di un nodo sia saturo. Questo comporta un buffer overflow e una conseguente perdita;
- ✗ **Ritardo dovuto a *store and forward* e numero di nodi intermedi.** A causa dello *store and forward*, ogni nodo deve attendere di ricevere l'intero pacchetto prima di ritrasmetterlo. Inoltre, con l'aumentare dei nodi intermedi, il ritardo aumenta.
  (approfondimento *store and forward*)

2.4 Tipologie di ritardi

Esistono diverse tipologie di ritardo perché quando un pacchetto parte da un *host* (sorgente), passa attraverso una serie di *router* e conclude il viaggio in un altro *host* (destinazione). Questo comporta un ritardo in ciascun nodo (*host* o *router*). I principali ritardi sono: **ritardo di elaborazione**, **ritardo di accodamento**, **ritardo di trasmissione** e **ritardo di propagazione**. L'insieme di questi ritardi è chiamato **ritardo totale di nodo** (*nodal delay*).

Ritardo di elaborazione

Il tempo richiesto per esaminare l'intestazione del pacchetto e per determinare dove dirigerlo fa parte del **ritardo di elaborazione** (*processing delay*). Per dirigere si intende il tempo che impiega il *router* a determinare la sua parte di uscita.

Ritardo di accodamento

Una volta in coda, il pacchetto subisce un **ritardo di accodamento** (*queuing delay*) mentre attende la trasmissione sul collegamento. La lunghezza di tale ritardo dipenderà dal numero di pacchetto precedentemente arrivati, accodati e in attesa di trasmissione sullo stesso collegamento. In altre parole, è il tempo speso nel *buffer* prima che il pacchetto venga ritrasmesso.

Ritardo di trasmissione

Data L la lunghezza del pacchetto, in bit, e R *bps* la velocità di trasmissione del collegamento dal *router* A al *router* B, il **ritardo di trasmissione** (*transmission delay*) sarà $L \div R$. Questo è il tempo richiesto per trasmettere tutti i bit del pacchetto sul collegamento.
Più semplicemente, dipende dalla velocità di trasmissione e dalla dimensione del pacchetto ed è possibile sintetizzarlo con la formula:

$$t_{\text{trasm}} = \frac{\text{dim\_pacchetto}}{\text{velocità\_trasmissione}}$$

Ritardo di propagazione

Una volta immesso sul collegamento, un bit deve propagarsi fino al *router* B. Il tempo impiegato è il **ritardo di propagazione** (*propagation delay*). In altre parole è il tempo impiegato per percorrere la distanza verso il *router* successivo.

Strumenti di misurazione

Esistono diversi **strumenti per misurare il ritardo**:

- **PING.** Dato un indirizzo di destinazione, il calcolatore manda una serie di messaggi e misura il tempo che intercorre tra l'invio e la ricezione della risposta, chiamato anche *Rount Trip Time* (RTT).
- **TRACEROUTE.** Misura il *Round Trip Time* tra la sorgente e **tutti** gli apparati di rete intermedi.

2.5 Sintesi

- ***Internet Service Provider* (ISP).** Strumento utilizzato dai sistemi periferici per accedere ad Internet.
- ***Provider*.** Insieme di commutatori di pacchetto e di collegamenti, solitamente è un'azienda che fornisce servizi.
- **Obbiettivi ISP.** Fornire vari tipi di accesso alla rete ai dispositivi che si collegano (e.g. DSL, *wireless*, ecc.).
- **Tipi di ISP:**
  - **Livello 1.** *Internazionale* (Telecom, TIM, ...);
  - **Livello 2.** *Nazionale* (Fastweb);
  - **Livello 3.** *Locale* (solitamente per professionisti).
- **Definizione TCP/IP.** Protocolli più famosi utilizzati dai sistemi periferici, i commutatori di pacchetto e altre parti di Internet. N.B. il protocollo IP specifica il formato dei pacchetti scambiati tra *router* e sistemi periferici.
- **Definizione commutazione di circuito.** Le risorse sono riservate per l'intera comunicazione.
  - **Vantaggio commutazione di circuito.** Velocità costante grazie ad un canale dedicato e quindi ritardo contenuto.
  - **Svantaggio commutazione di circuito.** Spreco di risorse in caso di silenzi durante la comunicazione.
  - **Causa dei ritardi nella commutazione di circuito.** I motivi possono essere tre:
    I Instaurazione del circuito;
    II Distanza tra sorgente e destinazione;
    III Trasmissione vera e propria della comunicazione.
- **Definizione commutazione di pacchetto.** La sorgente divide i messaggi in parti più piccole chiamate **pacchetti**.
  - **Vantaggio commutazione di pacchetto.** Ottimizzazione delle risorse poiché c'è una condivisione durante l'inattività.
  - **Svantaggi commutazione di circuito.** Eventuale perdita di pacchetti nel caso in cui un nodo intermedio abbia il *buffer* saturo (generazione di *buffer overflow*); ritardo causato da *store and forward* poiché ogni pacchetto per essere inoltrato deve essere completamente trasmesso; all'aumentare dei nodi intermedi, il ritardo aumenta.
- **Ritardo di elaborazione (*processing delay*).** Tempo impiegato dal *router* per esaminare l'intestazione del pacchetto e determinare l'uscita.
- **Ritardo di accodamento (*queuing delay*).** Tempo impiegato dal pacchetto all'interno della coda del buffer del *router*.

- **Ritardo di trasmissione (*transmission delay*).** Tempo che dipende dal rapporto tra la dimensione del pacchetto e la velocità di trasmissione.
- **Ritardo di propagazione (*propagation delay*).** Tempo impiegato per percorrere la distanza verso il *router* successivo.
- **Strumenti per la misurazione del ritardo.** I due strumenti sono "PING" e "TRACEROUTE". La differenza è che PING misura il RTT tra sorgente e destinazione, mentre il TRACEROUTE misura il RTT tra sorgente e ogni nodo intermedio.

3 Tecnica di load balancing, Throughput e collo di bottiglia

3.1 Tecnica di load balancing

Nel momento in cui il **mittente** (sorgente) calcola il **percorso migliore** per inviare i suoi dati al destinatario, può accadere che **trovi due o più strade identiche**. Con quest'ultimo termine si intende che i percorsi con il costo minimo, e quindi i più efficienti, siano due o più. In questo caso, viene applicata la tecnica di load balancing.

La tecnica di **load balancing** prevede di suddividere il carico dei pacchetti in tutti i percorsi migliori trovati. In questo modo, la comunicazione non avrà un unico percorso sovraccaricato, ma il carico sarà diviso tra più percorsi.

3.2 Throughput

Un'altra misura che influisce sulle prestazioni in una rete di calcolatori è il throughput *end-to-end*. Esistono **due tipi di throughput**:

- **Throughput istantaneo**, in ogni istante di tempo p, è la velocità (misurata in bit per secondo, *bps*) alla quale il destinatario B sta ricevendo il file.
- **Throughput medio** è dato da una formula specifica. Se l'oggetto da inviare è formato da F bit e il trasferimento richiede T secondi affinché il destinatario B riceva tutti gli F bit, allora il throughput medio del trasferimento dell'oggetto da inviare è di

$$\frac{F}{T} \textit{ bit per secondo}$$

3.3 Collo di bottiglia

Quindi, la connessione *end-to-end* presenta criticità nel momento in cui più dispositivi dividono la strada tra sorgente e destinazione. Si parla, infatti, di **collo di bottiglia** (*bottleneck link*), nel momento in cui la velocità di trasferimento viene diminuita a causa di un canale più piccolo o a causa di un dispositivo con banda minore.

4 Architettura a livelli e incapsulamento

4.1 Architettura a livelli

Un'**architettura a livelli** consente di manipolare una parte specifica e ben definita di un sistema articolato e complesso.

Questa struttura è data dal fatto che fin quando ciascun **livello** (*layer*, o strato) fornisce lo stesso servizio allo strato superiore e utilizza gli stessi servizi dello stato inferiore, la parte rimanente del sistema rimane invariata al variare dell'implementazione a quel livello.

I **servizi** vengono offerti da un determinato livello a quello superiore, ovvero si tratta del **modello di servizio** (*service model*) di un livello. Più in generale, **ogni livello fornisce il suo servizio** effettuando determinate azioni all'interno del livello stesso e utilizzando i servizi del livello immediatamente inferiore.

Nel caso di sistemi grandi e complessi, che vengono costantemente aggiornati, la capacità di cambiare l'implementazione di un servizio senza coinvolgere altre componenti del sistema costituisce un ulteriore importante vantaggio legato alla stratificazione. Quindi, i pro e i contro di questa architettura sono:

- **Vantaggio**
  - Il sistema è strutturato e dunque permette il trattamento dei componenti senza stravolgere l'intera architettura o struttura.
- **Svantaggi**
  - Possibilità di duplicazione delle funzionalità tra due o più livelli, ovvero che un livello cloni le caratteristiche del livello inferiore;
  - Possibilità che la funzionalità presente ad un livello possa richiedere informazioni presenti solo ad un altro livello.

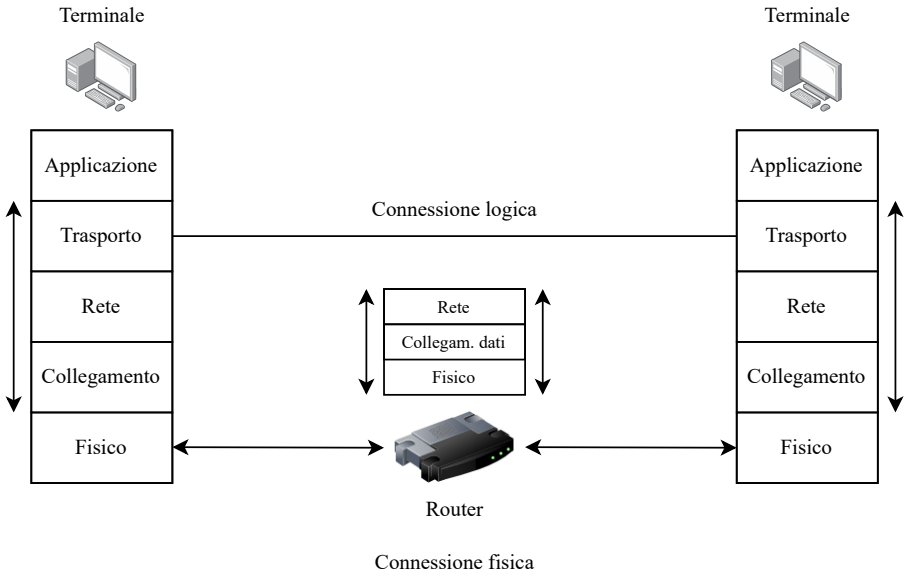
Ogni livello ha un **protocollo** e l'insieme dei protocolli vengono definiti **pila di protocolli** (*protocol stack*). La pila di protocolli di Internet consiste di cinque livelli:

1. Fisico
2. Collegamento
3. Rete
4. Trasporto
5. Applicazione

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

4.2 Incapsulamento

L'**incapsulamento** (o imbustamento) è un modus operandi applicato nel momento in cui si deve inviare un messaggio.



La comunicazione avviene nel seguente modo:

1. Parte nel **livello di applicazione** del host mittente il quale crea un **messaggio a livello di applicazione** (*application-layer message*) concatenando informazioni aggiuntive, o meglio le informazioni di intestazione. Alla fine del processo di creazione, il messaggio viene passato al livello inferiore, quello di trasporto;

2. A **livello di trasporto** vengono aggiunte altre informazioni di intestazione. Le intestazioni di applicazione e trasporto formano il **segmento a livello di trasporto** (*transport-layer segment*) che incapsula il messaggio a livello di applicazione. Infine, il livello di trasporto passa il messaggio al livello di rete;

3. A **livello di rete** vengono aggiunte informazioni come gli indirizzi dei sistemi periferici di sorgente e di destinazione. Facendo così viene creato un **datagramma a livello di rete** (*network-layer datagram*). Infine, il messaggio viene passato al livello collegamento (*link*);

4. A **livello di collegamento** le informazioni aggiuntive creano un **frame a livello di collegamento** (*link-layer frame*);

5. A **livello fisico** vengono inviati i dati al router e qui termina l'incapsulamento.

Per cui ad ogni livello, il pacchetto ha due tipi di campi: l'intestazione e **payload** (il carico utile trasportato). Il payload è tipicamente un pacchetto proveniente dal livello superiore.

5 Indirizzi IP

5.1 Indirizzi IP

Un indirizzo IP consente di rendere **identificativo** e **univoco** un host all'interno della rete. Gli indirizzi IP vengono **rappresentati** con 32 bit e utilizzando una **notazione decimale puntata**. Prendendo in considerazione l'architettura di rete spiegata nel capitolo precedente, l'IP si posiziona al livello di rete, nel quale viene aggiunto al messaggio da inviare.

La **notazione decimale puntata** è una rappresentazione degli indirizzi IP che facilita la lettura. Il modus operandi per ottenere tale notazione è il seguente:

1. Dividere i bit in 4 gruppi, ovvero 8 bit per ciascun gruppo;
2. Traduzione di ogni gruppo da binario a decimale;
3. Divisione di ogni gruppo da un punto.

Negli indirizzi IP è importante dividere il prefisso dal suffisso poiché ogni parte ha un significato diverso:

- **Prefisso**, identifica una rete all'interno di Internet;
- **Suffisso**, identifica un host all'interno della rete.

Non esiste un numero specifico di indirizzi IP per il prefisso e per il suffisso. Questo perché dipendono entrambi dalla grandezza della rete; più è grande la rete e meno bit ha di prefisso.

Un ***esempio***: 157.27.12.63/16, dove 157.27 identifica il prefisso e 12.63 il suffisso.

5.2 Maschera e blocco CIDR

Per identificare il numero di bit presenti nel prefisso, il calcolatore utilizza una sequenza di 32 bit in cui i bit del prefisso sono posti tutti a uno e i restanti a zero. Questo metodo si chiama maschera e un esempio di **maschera** 16 (notazione: /16):

11111111111111110000000000000000

Che rappresenta l'indirizzo: 255.255.0.0

Per cui si può affermare che la maschera **identifica** la grandezza della rete. Pensandoci, più è grande la maschera e più piccola è la rete visto che c'è una stretta relazione con il prefisso di un indirizzo IP.

Un **blocco CIDR** (*Classless Inter-Domain Routing*) è un intervallo di indirizzi IP che sono disponibili nella propria rete.

5.3 Esercizio di traduzione e numero host

Dato il seguente indirizzo in notazione binaria:

$$11100111 \quad 11011011 \quad 10001011 \quad 01101111$$

Si rappresenta in notazione decimale puntata.

Per **prima cosa** si esegue la traduzione di ogni gruppo da binario a decimale:

- $11100111 \longrightarrow 2^7 \cdot 1 + 2^6 \cdot 1 + 2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 0 + 2^2 \cdot 1 + 2^1 \cdot 1 + 2^0 \cdot 1 = 231$
- $11011011 \longrightarrow 2^7 \cdot 1 + 2^6 \cdot 1 + 2^5 \cdot 0 + 2^4 \cdot 1 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 219$
- $10001011 \longrightarrow 2^7 \cdot 1 + 2^6 \cdot 0 + 2^5 \cdot 0 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 139$
- $01101111 \longrightarrow 2^7 \cdot 0 + 2^6 \cdot 1 + 2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 1 + 2^1 \cdot 1 + 2^0 \cdot 1 = 111$

E infine si riscrive la notazione in notazione decimale puntata:

$$231.219.139.111$$

Adesso viene eseguita la conversione binaria della notazione decimale puntata del seguente indirizzo:

$$221.34.255.82$$

Per **prima cosa** si esegue la traduzione di ogni gruppo. La traduzione non è banale, difatti si prenderà ciascun gruppo, si dividerà per due e nella colonna di destra verranno scritti i riporti. Infine, il numero binario sarà scritto dal numero di riporto del numero più basso, fino al numero più alto:

$221 \div 2$	1
$110 \div 2$	0
$55 \div 2$	1
$27 \div 2$	1
$13 \div 2$	1
$6 \div 2$	0
$3 \div 2$	1
$1 \div 2$	1

E così via. Unica accortezza da **ricordare** che se il numero di bit fosse meno di 8, si aggiungono zeri nella parte più significativa.

Dopo alcuni calcoli l'indirizzo IP in binario è:

$$11011101 \quad 00100010 \quad 11111111 \quad 01010010$$

Una rete con un suffisso di /20, quanti host contiene? Dato un indirizzo IP da 32 bit, se il suffisso ha 20 bit, allora: $2^{32} \div 2^{20} = 2^{12}$. Una rete con un suffisso di 20 bit, avrà un prefisso di 12 bit, ovvero $2^{12} \longrightarrow 4096$ indirizzi.

6 Indirizzi IP privati

6.1 Indirizzi IP privati

Gli indirizzi IP riservati sono indirizzi IP che non possono essere assegnati a nessun host. In particolare, sono:

- **Indirizzo di rete**: tutti i bit del suffisso sono posti a zero;
- **Indirizzo di Direct Broadcast**: tutti i bit del suffisso sono posti a uno;
- **Indirizzo con tutti i bit a zero**;
- **Indirizzo con tutti i bit a uno**;

6.2 Esercizio subnetting, creazione sottoreti partendo da un blocco di indirizzi

La tecnica di ***subnetting*** consiste nel suddividere una rete in più rete, chiamate sottoreti. L'esercizio fornisce un indirizzo IP:

180.190.0.0/16

E l'obbiettivo è quello di creare due sottoreti di grandezza identica.

Per **prima cosa** si deve effettuare la conversione da notazione decimale puntata a notazione binaria:

10110100 10111110 00000000 00000000

A questo punto è possibile creare due sottoreti come richiesto dall'esercizio.

Per farlo si identifica il primo bit utile nell'indirizzo binario. Esso è possibile trovarlo escludendo il prefisso (/16). Quindi, si conta dal bit più significativo 16 bit, escluso sé stesso, arrivando al bit numero 17, ovvero il più significativo del terzo gruppo:

10110100 10111110 ***x***0000000 00000000

Al posto della ***x*** verrà sostituito lo zero per creare la prima rete e l'uno per creare la seconda rete. Gli indirizzi saranno:

Ind. bin. della **prima rete**:	10110100	10111110	**0**0000000	00000000
Ind. bin. della **seconda rete**:	10110100	10111110	**1**0000000	00000000

Il **prefisso** delle sottoreti sarà aumentato di un solo bit, per cui saranno /17.

La **traduzione in decimale** sarà:

- **Prima rete:** 180.190.0.0/17
- **Seconda rete:** 180.190.128.0/17

La **maschera** delle sottoreti sarà formata da 17 bit posti a 1 partendo dal più significativo:

$$11111111 \quad 11111111 \quad 10000000 \quad 00000000$$

E la **traduzione della maschera** in notazione decimale puntata sarà:

$$255.255.128.0$$

Infine, la **dimensione dei blocchi** di rete:

$$\textbf{Pre-subnetting}: \quad 2^{32} \div 2^{16} = 2^{16} \quad \longrightarrow \quad 65'536 \text{ indirizzi}$$

$$\textbf{Post-subnetting}: \quad 2^{32} \div 2^{17} = 2^{15} \quad \longrightarrow \quad 32'768 \text{ indirizzi}$$

7 Socket e protocolli a livello di trasporto: TCP e UDP

7.1 Socket

I sistemi periferici collegati a Internet forniscono un'**interfaccia socket** (*socket interface*), che specifica come un programma eseguito su un sistema periferico possa chiedere a Internet di recapitare dati a un programma eseguito su un altro sistema periferico.

In altre parole, l'interfaccia socket è un insieme di regole che il programma mittente deve seguire in modo che i dati siano recapitati al programma di destinazione.

È importante anche sapere l'esistenza delle **API** (*Application Programmin Interface*) tra l'applicazione e la rete, poiché l'interfaccia socket non è altro che un'interfaccia di programmazione con cui le applicazioni di rete vengono costruite.

Infine, esistono due tipi di processi:

- Il **client**, ovvero colui che avvia la comunicazione e ha un IP dinamico;
- Il **server**, ovvero colui che attende di essere contattato per iniziare la sessione e ha un IP statico.

7.2 Protocolli nel livello di trasporto

Dopo il livello applicativo, si trova il livello di trasporto. Esso può utilizzare due possibili **protocolli** nelle reti TCP/IP: **TCP** e **UDP**.

7.2.1 Protocollo TCP

Il protocollo TCP prevede la fornitura di un servizio **orientato alla connessione** (*Connection oriented communication*) e il **trasporto affidabile dei dati**.

Le **caratteristiche** di questo protocollo sono:

- **Servizio orientato alla connessione**. Questa caratteristica deriva dal fatto che vengono scambiate delle informazioni di controllo prima che i messaggi veri e propri vengano processati dal livello applicativo. Questo scambio di messaggi prende il nome di **handshaking**.
  Successivamente, *se tutto è andato a buon fine*, si instaura una **connessione TCP** tra le socket dei due processi. La connessione viene chiamata ***full-duplex*** ovvero i due processi possono scambiarsi i messaggi contemporaneamente sulla connessione. Infine, per terminare la connessione, il mittente e il destinatario si **scambiano** alcuni **messaggi**.
- **Servizio di trasferimento affidabile**. Il protocollo è affidabile poiché i vari controlli effettuati permettono di:
  - Non perdere dati;
  - Inviarli nel giusto ordine;
  - Evitare duplicazioni.

Inoltre, questo protocollo implementa un meccanismo di **controllo della congestione**. Nel caso in cui si manifestasse, eseguirebbe una "strozzatura" del processo d'invio.

Infine, il protocollo negli ultimi anni implementa anche il ***secure sockets layer*** (**SSL**). Ovvero un servizio aggiuntivo che consente la cifratura dei messaggi. Attenzione che questo servizio deve essere attivo sia lato client che lato server, altrimenti si rischia che una delle due parti non possa decifrare il messaggio ricevuto.

7.2.2 Protocollo UDP

Al contrario del protocollo TCP, UDP è un protocollo di trasporto **leggero**, **rapido** e **minimalista**. Esso ***non*** necessita di connessione, per cui non esegue alcun handshaking e di conseguenza **non** fornisce neanche un **trasferimento dati affidabile**.

Per cui, quando processo invia un messaggio tramite un socket, UDP **non garantisce la consegna del messaggio**. Inoltre, i messaggi potrebbero giungere a destinazione non in ordine.

La **rapidità del protocollo** è dovuta anche al fatto che i pacchetti vengono inviati direttamente **senza** utilizzare un sistema di **controllo della congestione**. Tuttavia, il reale throughput end-to-end potrà essere inferiore a causa della banda limitata dei collegamenti coinvolti o a causa della congestione. Si ricorda che con il termine throughput ci si riferisce al tasso con quale il processo mittente può inviare i bit al processo ricevente.

- **Vantaggi**
  - **Leggero** poiché non ha bisogno di controllare che la connessione sia instaurata, ovvero il fenomeno di *handshaking* viene eliminato;
  - **Rapido** poiché non esiste nessun sistema di controllo della congestione, per cui i pacchetti vengono mandati uno dopo l'altro. Talvolta il throughput potrebbe essere minore a causa di eventuali colli di bottiglia;
  - **Minimalista** poiché non implementa tecniche particolari come detto in precedenza.
- **Svantaggi**
  - **Nessuna affidabilità** a causa della mancanza del fenomeno di *handshaking*. Quindi, **nessuna garanzia di consegna** del messaggio;
  - **Alta probabilità di congestione** dovuta alla mancanza di controllo di essa. Quindi, il buffer potrebbe riempirsi rapidamente;
  - **Messaggi non in ordine**.

7.3 Esercizio sull'indirizzamento

Dato il seguente IP:

$$140.120.84.20/20$$

Determinare l'indirizzo di rete.

Il **primo passo** è la traduzione dell'IP da notazione decimale puntata a notazione binaria:

$$140_{10} \longrightarrow 10001100$$

$$120_{10} \longrightarrow 01111000$$

$$84_{10} \longrightarrow 01010100$$

$$20_{10} \longrightarrow 00010100$$

Scrivendo l'indirizzo esteso:

$$10001100 \quad 01111000 \quad 01010100 \quad 00010100$$

Il **secondo passo** è l'azzeramento del suffisso (bits in rosso) dato che l'indirizzo di rete ha il prefisso *non* nullo e il suffisso con tutti i bit a zero. Il prefisso viene dato dall' esercizio, ovvero /20:

$$10001100 \quad 01111000 \quad 0101\mathbf{0000} \quad \mathbf{00000000}$$

Il **terzo** e ultimo **passo** è la conversione in decimale e scrivere l'indirizzo di rete in notazione decimale puntata:

$$10001100_2 \longrightarrow 140_{10}$$

$$01111000_2 \longrightarrow 120_{10}$$

$$01010000_2 \longrightarrow 80_{10}$$

$$00000000_2 \longrightarrow 0_{10}$$

E l'indirizzo di rete sarà:

$$140.120.80.0/20$$

7.4 Esercizio subnetting - Avanzato

Date 3 reti LAN, viene assegnato il blocco di rete 165.5.1.0/24. Creare 3 sottoreti per ogni rete LAN in modo da avere lo stesso numero di host.

Il **primo passo** è la classica traduzione in binaria. Si deve tradurre l'indirizzo da notazione decimale puntata a binario:

$$\begin{aligned} 165_{10} &\longrightarrow 10100101_2 \\ 5_{10} &\longrightarrow 00000101_2 \\ 1_{10} &\longrightarrow 00000001_2 \\ 0_{10} &\longrightarrow 00000000_2 \end{aligned}$$

Il **secondo passo** è quello di creare delle sottoreti. Per farlo si deve prendere in considerazione il suffisso. Prima di tutto, si scrive l'indirizzo in notazione binaria:

10100101 00000101 00000001 00000000

La creazione di 3 sottoreti prevede almeno due bit. Difatti, se venisse scelto un bit, si potrebbe creare al massimo 2 sottoreti. Per cui, dall'indirizzo in notazione binaria, si riservano (x in rosso) due bit nel suffisso per creare le sottoreti:

10100101 00000101 00000001 ***xx***000000

Riservando questi due bit, adesso si sono create quattro sottoreti:

LAN1.	10100101	00000101	00000001	**00**000000
LAN2.	10100101	00000101	00000001	**01**000000
LAN3.	10100101	00000101	00000001	**10**000000
LAN4.	10100101	00000101	00000001	**11**000000

La prima sottorete è assegnata alla LAN1, la seconda alla LAN2, la terza alla LAN3 e la quarta è considerata libera. Essa potrà essere utilizzata in futuro.

Il **terzo passo** è calcolare il nuovo prefisso delle sottoreti. Ogni sottorete è formata dal prefisso della rete originaria (/24), più due bit che sono serviti per creare le 3 reti. Quindi, il nuovo prefisso delle 3 reti è diventato /26.

Infine, il **quarto passo** è la scrittura in notazione decimale puntata delle tre reti e il calcolo degli indirizzi disponibili.

La conversione è semplice e si omettono i passaggi e le spiegazioni:

LAN1.	165.5.1.0/26
LAN2.	165.5.1.64/26
LAN3.	165.5.1.128/26
LAN4.	165.5.1.192/26

E la rete è passata da avere ($2^{32} \div 2^{24} = 2^8$) 256 indirizzi disponibili a ($2^{32} \div 2^{26} = 2^6$) 64 indirizzi.

7.4.1 Domanda bonus

Se la LAN1 avesse dovuto avere il doppio degli indirizzi rispetto alla LAN2 e alla LAN3, l'esercizio come si sarebbe svolto?

In questo caso specifico, partendo dal **passo numero due**, ovvero alla creazione delle sottoreti, si sarebbe proceduti in maniera diversa.

Invece di prendere due bit, si prende un solo bit che ci permette di creare due sottoreti: nella prima sottorete si assegna la LAN1 e nella seconda sottorete si assegnano LAN2 e LAN3.

10100101 00000101 00000001 x0000000

Creando le due sottoreti:

10100101 00000101 00000001 **0**0000000
10100101 00000101 00000001 **1**0000000

Il **terzo passo** è quello di creare le due sottoreti all'interno del secondo indirizzo. Due sottoreti necessitano di due bit, per cui la divisione sarà:

LAN2. 10100101 00000101 00000001 1**0**000000
LAN3. 10100101 00000101 00000001 1**1**000000

Infine, il **quarto passo** è quello di calcolare la maschera delle reti, riscrivere gli indirizzi in notazione decimale puntata e calcolare il numero di indirizzi possibili.

La maschera per la LAN1 è aumentata di 1 dalla rete di partenza, per cui /25. Mentre per la LAN2 e LAN3 è aumentata di 2 bit, ovvero /26.

La scrittura in notazione decimale puntata sarà:

LAN1. 165.5.1.0/25
LAN2. 165.1.128/26
LAN3. 165.1.192/26

E il numero di indirizzi della LAN1 saranno ($2^{32} \div 2^{25} = 2^7$) 128, mentre quelli della LAN2 e della LAN3 rimarranno a 64 indirizzi.

8 Protocollo HTTP

8.1 Protocollo HTTP

HTTP (*Hypertext Transfer Protocol*), protocollo a livello di applicazione del Web, costituisce il cuore del Web. Questo **protocollo** è implementato nei programmi in esecuzione su sistemi periferici diversi che comunicano tra loro scambiandosi messaggi HTTP.

Una **pagina web** (*web page*), detta anche documento, è costituita da oggetti. Un **oggetto** è semplicemente un file indirizzabile tramite un URL. La maggioranza delle pagine web consiste di un **file HTML principale** e diversi oggetti referenziati da esso.

Un **browser web** implementa il lato *client* di HTTP, ovvero l'utente stesso. Mentre il **web server** implementa il lato server di HTTP, ospita oggetti web, indirizzabili tramite URL.

HTTP utilizza il protocollo **TCP come protocollo di trasporto**. Il client HTTP per prima cosa inizia una connessione TCP con il server. Una volta stabilita, i processi client e server accedono a TCP attraverso le proprie socket.

Dato che i server HTTP non mantengono informazioni sui client, HTTP è classificato come **protocollo senza memoria di stato** (*stateless protocol*). Un **web server** è **sempre attivo**, ha un indirizzo IP fisso e risponde potenzialmente alle richieste provenienti da milioni di diversi browser.

8.2 HTTP con connessioni (non) persistenti

Sia i client che i server possono avere due diverse configurazioni: le connessioni persistenti e le connessioni non persistenti. Prima di iniziare la spiegazione, si introduce il concetto di **round-trip time** (**RTT**), che rappresenta il **tempo impiegato da un pacchetto per viaggiare dal client al server e poi tornare al client**. Esso include i **ritardi di propagazione**, di **accodamento** nei **router** e nei **commutatori intermedi** nonché di elaborazione del pacchetto.

8.2.1 Connessioni non persistenti

Per spiegare le connessioni non persistenti, si faccia riferimento ad un esempio in cui c'è il trasferimento di una pagina web dal server al client. Si supponga che la pagina consista di un file HTML principale e di 10 immagini, e tutti gli oggetti risiedano sullo stesso server.

1. Il processo client HTTP inizializza una connessione di tipo TCP con il server sulla porta 80, ovvero la porta di default per HTTP.
2. Il client HTTP, tramite la socket, invia al server un messaggio di richiesta del file principale.
3. Il processo server HTTP riceve il messaggio di richiesta attraverso la socket, recupera l'oggetto richiesto dalla propria memoria, lo incapsula in un messaggio di risposta che viene inviato al client attraverso la socket.
4. Il processo server HTTP comunica a TCP di concludere la connessione. Tuttavia, il protocollo TCP garantisce la consegna del messaggio, per cui non termina la connessione finché il client non ha ricevuto l'intero messaggio.
5. Il client HTTP riceve il messaggio di risposta. La connessione TCP termina a questo punto e il messaggio ricevuto indica che l'oggetto incapsulato è un file HTML. Il client estrae il file del messaggio di risposta, esamina il file HTML e trova i riferimenti ai 10 oggetti.
6. Infine, vengono ripetuti tutti i primi quattro passi per ciascuno degli oggetti referenziati.

Si conclude che l'utilizzo di connessione non persistenti, consente di chiudere ogni connessione aperta dopo l'invio dell'oggetto da parte del server e dopo aver ricevuto una conferma dal client.

Nell'esempio vengono create 11 connessioni TCP, una per ogni oggetto richiesto.

Come si può notare, queste connessioni presentano alcuni **limiti**:

1. Per ogni oggetto richiesto occorre stabilire e mantenere una nuova connessione.
2. Per ogni connessione si deve allocare buffer e mantenere variabili TCP sia nel client che nel server.
3. Ciascun oggetto subisce un ritardo di consegna di due RTT: uno per stabilire la connessione TCP e uno per richiedere e ricevere un oggetto.

Questi limiti richiedono un grande onere sul web server che deve aprire nuove connessioni ogni qualvolta ci deve essere uno scambio di dati. Il grande onere riguarda soprattutto il momento in cui il server riceve centinaia di richieste da parte dei client.

8.2.2 Connessioni persistenti

La particolarità delle connessioni persistenti è la possibilità da parte del server di lasciare aperta la connessione TCP dopo l'invio di una risposta. Così facendo, le richieste/risposte future da parte degli stessi client e server potranno essere trasmesse sulla stessa connessione.

Difatti, il server può inviare un'intera pagina web **sfruttando** solamente **una connessione** TCP permanente, oppure inviare più pagine web allo stesso client.

Le **richieste** possono essere effettuate **una dopo l'altra senza attendere** le risposte delle eventuali richieste pendenti (*pipelining*).

In generale, il server HTTP **chiude la connessione** quando essa rimane inattiva per un lasso di tempo arbitrario.

8.3 Formato dei messaggi HTTP

Esistono due tipi di formati di questo protocollo: il messaggio di richiesta HTTP e messaggio di risposta HTTP.

8.3.1 Messaggio di richiesta HTTP

I messaggi di richiesta possono avere un numero indefinito di righe, anche una sola.

Ogni riga alla fine ha un **carattere di ritorno a capo** (*carriage return*) e un **carattere di nuova linea** (*line feed*).

L'ultima riga è seguita da una coppia di caratteri di ritorno a capo e nuova linea aggiuntivi.

In generale, la **prima riga** è la **riga di richiesta** (*request line*) e quelle successive si chiamano **righe di intestazione** (*header lines*).

La **riga di richiesta** è formata da tre campi:

- Campo **metodo** (GET, POST, HEAD, PUT, DELETE)
- Campo **URL**
- Campo **versione HTTP**

Le **righe di intestazione** possono essere di molti tipi. La riga **Host** specifica l'host su cui risiede l'oggetto, la riga **Connection** indica se il server deve effettuare una connessione di tipo persistente o non persistente, la riga **User-agent** specifica il tipo di browser che sta effettuando la richiesta al server e, infine, la riga **Accept-language** indica se l'utente preferisce ricevere una versione dell'oggetto nella lingua specificata; in caso di mancanza, il server provvederà a fornire la versione di default.

8.3.2 Messaggio di risposta HTTP

Nei messaggi di risposta ci sono tre sezioni importanti: una **riga di stato iniziale**, le **righe di intestazione** e il **corpo**.

La **riga di stato iniziale** presenta tre campi:

- **Versione del protocollo**
- **Codice di stato**
- **Messaggio di stato** (approfondimento alla fine di questo paragrafo)

Le **righe di intestazione** possono essere di tipo **Connection** per comunicare al client la gestione della connessione, per esempio "Connection: close" per indicare l'intenzione di chiudere la connessione TCP dopo l'invio del messaggio; di tipo **Date** per indicare l'ora e la data di creazione e invio, da parte del server, della risposta HTTP; di tipo **Server** per indicare che il messaggio è stato generato da un determinato tipo di web server, simile alla riga "User-agent" nel messaggio di richiesta; di tipo **Last-Modified** per indicare l'istante e la data in cui l'oggetto è stato creato o modificato per l'ultima volta; di tipo **Content-Length** per indicare il numero di byte dell'oggetto inviato; di tipo **Content-Type** per indicare il tipo di oggetto specificato nel corpo.

Le **righe di corpo** sono il fulcro del messaggio. Esse contengono l'oggetto richiesto.

Nella **riga di stato iniziale**, i due campi **codice** e **messaggio di stato** indicano operazioni importanti. Qui di seguito si elencano i codici con i relativi messaggi, sono riportati solo i più importanti:

- 200 OK: la richiesta ha avuto successo e in risposta si invia l'informazione.
- 301 Moved Permanently: l'oggetto richiesto è stato trasferito in modo permanente; il nuovo URL viene specificato nell'intestazione, campo Location, del messaggio di risposta.
- 400 Bad Request: codice di errore generico che indica che la richiesta non è stata compresa dal server.
- 404 Not Found: il documento richiesto non esiste sul server.
- 505 HTTP Version Not Supported: il server non dispone della versione di protocollo HTTP richiesta.

8.4 Cookie

I ***cookie*** è un meccanismo utilizzato dal server per sapere se ha interagito precedentemente con un determinato client.

La **prima volta** che il *client* interagisce con un *server*, inviando una richiesta GET, il server crea un identificativo (codice) associato all'utente (e.g. 1234).

Il server **risponde** con una richiesta REPLY HTTP, impostando nell'intestazione il valore ***set-cookie*** al valore impostato al passaggio prima. Alla ricezione della risposta, il *client* memorizza l'indirizzo della pagina usato per fare la prima richiesta GET e il codice fornito dal server (*set-cookie*).

Dopo una serie di interazioni, quando il *client* effettuerà di nuovo una richiesta GET al server specifico, esso dovrà indicare come codice *cookie*, il codice salvato la prima volta. Invece, il server controllerà la storia associata all'utente con il codice fornito e restituisce una risposta specifica all'utente.

8.5 Cache di rete

La **cache di rete** intercetta i messaggi e memorizza le risposte. I vantaggi riguardano principalmente la velocità. Il suo funzionamento è il seguente:

1. Invio della richiesta di un determinato dato da parte del *client*;
2. **Cache di rete** intercetta il messaggio e controlla se ha il dato richiesto:
   - Se il **dato è presente**:
     (a) La cache di rete invia una richiesta di tipo **GET condizionale** per verificare che il contenuto del dato richiesto non sia stato modificato nel server (campo nell'intestazione: *if-modified-since*);
     (b) Il server risponde inviando un messaggio con scritto nell'intestazione *not modified*;
     (c) Cache di rete invia direttamente il dato al client diminuendo il tempo d'attesa del mittente (lo scambio di messaggi piccoli tra cache di rete e server è rapido!).
   - Se il **dato non è presente**:
     (a) La cache di rete invia una richiesta di tipo **GET condizionale** per verificare che il contenuto del dato richiesto non sia stato modificato nel server (campo nell'intestazione: *if-modified-since*);
     (b) Il server di destinazione risponde inviando i dati richiesti (operazione lenta);
     (c) La cache di rete intercetta la risposta salvando una copia dei dati al suo interno. La risposta non viene bloccata, quindi arriva anche al client,

9 DNS

9.1 DNS

Gli host Internet possono essere identificati in vari modi. I **nomi degli host** (*hostname*), per esempio www.facebook.com o www.google.com, risultano abbastanza appropriati per l'uomo, ma forniscono ben poca informazione sulla loro collocazione all'interno di Internet. Per cui, i nomi vengono utilizzati dagli utenti finali, mentre i calcolatori utilizzando gli **indirizzi IP** degli host.

Gli utenti finali inseriscono i nomi degli host e il calcolatore riesce a ottenere l'indirizzo IP grazie ad un servizio in grado di tradurre i nomi in indirizzi IP. Il suo nome è **Domain Name System** (DNS).

Il **DNS** è un **database** distribuito implementato in una **gerarchia** di **DNS server** e un protocollo a livello di applicazione che consente agli **host** di **interrogare** il **database**. Generalmente i DNS server sono macchine UNIX che eseguono un software chiamato BIND (*Berkeley Internet name domain*). Il protocollo DNS utilizza **UDP** e la **porta 53**.

Nel dettaglio quello che succede nel momento in cui un utente digita un host name:

1. La macchina utente (*client*) esegue l'applicazione DNS;
2. Il browser estrae il nome del host dall'URL inserito e lo passa all'applicazione DNS;
3. L'applicazione DNS esegue una interrogazione (*query*), sul database di una DNS server, contenente l'hostname;
4. L'applicazione DNS prima o poi riceve una risposta che include l'indirizzo IP corrispondente al host dell'hostname fornito;
5. Una volta ricevuto l'indirizzo IP dal DNS, il browser può dare inizio a una connessione di tipo TCP verso il processo server HTTP collegato alla porta 80 di quell'indirizzo IP.

9.2 Approfondimento sul database distribuito e gerarchico

Il DNS utilizza un grande numero di server, organizzati in maniera gerarchica e distribuiti nel mondo. Nessun DNS server ha le corrispondenze per tutti gli host in Internet, che sono invece distribuite tra tutti i DNS server.

Esistono **tre classi** di DNS server:

- **Root server**. In Internet esistono 400 root server circa, dislocati in tutto il mondo. Essi sono gestiti da 13 diverse organizzazioni.
  I root server forniscono gli indirizzi IP dei server TLD.
- **Top-level domain (TLD) server**. Questi server si occupano dei domini di primo livello quali *com*, *org*, *net*, *edu* e *gov*, e di tutti i domini di primo livello relativi ai vari paesi, come *uk*, *fr*, *ca* e *jp*. Per esempio, l'azienda "Verisign Global Registry Services" gestisce i TLD server per il dominio com.
  I server TLD forniscono gli indirizzi IP dei server autoritativi.
- **Server autoritativi**. Ogni organizzazione dotata di host pubblicamente accessibili tramite Internet (quali web server e e-mail server) deve fornire record DNS pubblicamente accessibili che associno i nomi di tali host a indirizzi IP. Il DNS server autoritativo dell'organizzazione ospita questi record.
  Un'organizzazione può scegliere di implementare il proprio server autoritativo oppure di pagare un fornitore di servizi per ospitare questi record su un suo server.

Un **esempio** riepilogativo può essere del tipo: si supponga che un client DNS voglia determinare l'indirizzo IP relativo all'hostname www.amazon.com.
Per fare ciò, il client dapprima contatta uno dei root server che gli restituisce uno o più indirizzi IP relativi al server TLD per il dominio *com* (dominio gestito dall'azienda Verisign Global Registry Services).
Per cui, contatta uno di questi server TLD che gli restituisce uno o più indirizzi IP del server autoritativo per amazon.com.
Infine, contatta uno dei server autoritativi per amazon.com che gli restituisce l'indirizzo IP dell'hostname www.amazon.com.

9.3 Esercizio - Determinare CIDR e creare subnetting con condizioni

Si consideri la seguente rete formata da 5 sottoreti. Due indirizzi sono già assegnati alla rete:

101.75.79.255 101.75.80.0

Determinare il blocco CIDR più piccolo che contiene tali indirizzi. Per "più piccolo" si intende con numero minore di indirizzi.

Una volta trovato il blocco CIDR, creare 5 sottoreti con i seguenti vincoli:

- LAN1: Deve avere un prefisso di /21
- LAN2: Deve ospitare fino a 1000 host
- LAN3: Deve avere un prefisso di /23
- LAN4: Deve ospitare fino a 400 host
- LAN5: Deve avere la metà degli indirizzi a disposizione del blocco di partenza

Inizialmente si traducono gli indirizzi che si hanno in binario:

$$\begin{array}{lcl} 101.75.79.255 & \longrightarrow & 01100101\,.\,01000011\,.\,01001111\,.\,11111111 \\ 101.75.80.0 & \longrightarrow & 01100101\,.\,01000011\,.\,01010000\,.\,00000000 \end{array}$$

Per **calcolare il prefisso** basta prendere in considerazione i bit identici nei due indirizzi. Il conteggio prosegue finché non si trova un bit che differisce. Per cui, in questo esercizio il bit numero 20 differisce, infatti nel primo indirizzo è 0, mentre nel secondo è 1. In conclusione, il prefisso è /19. Ponendo i restanti $(2^{32} \div 2^{19} = 2^{13})$ 13 bit a zero, si ottiene l'indirizzo di rete:

101.75.64.0/19

Che avrà a disposizione 2^{13} indirizzi, ovvero 8192 il quale rappresenta il **blocco CIDR**.

Adesso inizia il calcolo dei vari indirizzi delle LAN. La LAN numero 5 deve avere la metà degli indirizzi disponibili messi a disposizione dalla rete di partenza, ovvero quella appena trovata. Per cui si creano le prime sottoreti (*subnetting*) assegnandone una alla LAN5 che avrà per cui il prefisso maggiorato di un bit e un numero di indirizzi disponibili pari a: $2^{32} \div 2^{20} = 2^{12} \rightarrow 4096$.

Quindi si creano le due sottoreti:

$$\begin{array}{lcl} 01100101\,.\,01000011\,.\,0100\mathbf{0}0000\,.\,00000000 & \longrightarrow & 101.75.64.0/20 \\ 01100101\,.\,01000011\,.\,0101\mathbf{1}0000\,.\,00000000 & \longrightarrow & 101.75.80.0/20 \end{array}$$

E si afferma che l'indirizzo 101.75.64.0/20 è assegnato alla LAN5.

Dal secondo indirizzo si creano altre sottoreti per soddisfare il criterio della prima LAN:

$$\begin{aligned} &01100101\,.\,01000011\,.\,0101\mathbf{0}000\,.\,00000000 \quad \longrightarrow \quad 101.75.80.0/21 \\ &01100101\,.\,01000011\,.\,0101\mathbf{1}000\,.\,00000000 \quad \longrightarrow \quad 101.75.88.0/21 \end{aligned}$$

E si afferma che l'indirizzo 101.75.80.0/21 è assegnato alla LAN1.

A questo punto si può notare che la seconda LAN richiede almeno 1000 host. Per soddisfare questa richiesta si può fare un calcolo rapido e confermare il fatto che se venissero create altre sottoreti dall'indirizzo disponibile qui sopra (101.75.88.0/21), si verrebbe a creare una rete con un prefisso di 22, un suffisso di $32 - 22 = 10$ bit, ovvero $2^{10} = 1024$ indirizzi. Quindi:

$$\begin{aligned} &01100101\,.\,01000011\,.\,0101\mathbf{0}000\,.\,00000000 \quad \longrightarrow \quad 101.75.88.0/22 \\ &01100101\,.\,01000011\,.\,0101\mathbf{1}000\,.\,00000000 \quad \longrightarrow \quad 101.75.92.0/22 \end{aligned}$$

E si afferma che l'indirizzo 101.75.88.0/22 è assegnato alla LAN2.

Infine, costruendo altre due sottoreti dall'indirizzo disponibile si troverebbe il nuovo indirizzo disponibile per la rete LAN numero 3 e 4. La numero 3 sarebbe rispettata perché il prefisso passerebbe da 22 a 23, mentre la numero 4 avrebbe $32 - 23 = 9$ bit di suffisso, ovvero $2^9 = 512$ indirizzi disponibili.
Per concludere quindi:

$$\begin{aligned} &01100101\,.\,01000011\,.\,0101110\mathbf{0}\,.\,00000000 \quad \longrightarrow \quad 101.75.92.0/23 \\ &01100101\,.\,01000011\,.\,0101111\mathbf{0}\,.\,00000000 \quad \longrightarrow \quad 101.75.94.0/23 \end{aligned}$$

E si afferma che l'indirizzo 101.75.92.0/23 è assegnato alla LAN3, mentre l'indirizzo 101.75.94.0/23 è assegnato alla LAN4.

10 Posta elettronica SMTP e livello di trasporto

10.1 Posta elettronica

L'e-mail rappresenta un mezzo di comunicazione asincrono: le persone inviano e leggono messaggi nel momento per loro più opportuno, senza doversi coordinare con altri utenti. I grandi vantaggi della posta elettronica sono la velocità, la facilità di distribuzione e il prezzo, ovvero gratuito.

In questo argomento ci sono tre **componenti principali**:

1. **User agent** (o agenti utente) che sono per esempio Microsoft Outlook e Apple Mail, i quali consentono agli utenti di leggere, rispondere, inoltrare, salvare e comporre i messaggi.

2. **Server di posta** (o *mail server*) costituiscono la parte centrale dell'infrastruttura del servizio di posta elettronica. Ciascun destinatario ha una **casella di posta** (*mailbox*) collocata in un mail server.
   Può capitare che una volta inviato il messaggio, il server non possa consegnare la posta. In tal caso esso la trattiene in una **coda di messaggi** e cerca di trasferirla in un secondo momento. In caso di mancata consegna dopo alcuni giorni dall'invio del messaggio, il server rimuove il messaggio e avvisa il mittente con un messaggio di posta elettronica.

3. **Protocollo SMTP** (*Simple Mail Transfer Protocol*) rappresenta il principale protocollo a livello di applicazione per la posta elettronica su Internet. Utilizza TCP, quindi viene considerato un servizio di trasferimento dati affidabile.
   Questo protocollo viene eseguito sia lato client, quando è in esecuzione sul server di posta del mittente, sia lato server, quando è in esecuzione sul server del destinatario.

10.2 Protocollo SMTP

Il **protocollo SMTP** si occupa di **trasferire i messaggi dal mail server del mittente a quello del destinatario**. Nonostante la sua fama, SMTP rappresenta una tecnologia ereditata con caratteristiche "arcaiche".
Per esempio, il corpo dei messaggi di posta li tratta come semplice codice ASCII a 7 bit. Questa restrizione aveva senso nei primi anni '80, quando la capacità trasmissiva era scarsa e nessuno inviava per posta elettronica grandi allegati quali immagini, audio o video, ma oggi la restrizione all'ASCII a 7 bit è piuttosto penalizzante, in quanto richiede che i dati multimediali binari vengano codificati in ASCII prima di essere inviati e che il messaggio venga nuovamente decodificato in binario dopo il trasporto.

Il protocollo SMTP è uno dei protagonisti principali nella comunicazione tra client e server. Per esempio, supponiamo che Alice voglia inviare a Bob un semplice messaggio ASCII:

1. Alice invoca il proprio *user agent* per la posta elettronica, fornisce l'indirizzo di posta di Bob, compone il messaggio e dà istruzione allo *user agent* di inviarlo.
2. Lo *user agent* di Alice invia il messaggio al suo *mail server*, il quale colloca esso in una coda di messaggi.
3. Il lato client di SMTP, eseguito sul server del mittente (Alice), vede il messaggio nella coda dei messaggi e apre una connessione TCP verso un server SMTP in esecuzione sul *mail server* del destinatario (Bob).
4. Dopo un *handshaking* SMTP, il client SMTP invia il messaggio di Alice sulla connessione TCP.
5. Il *mail server* di Bob, il lato server di SMTP riceve il messaggio, che viene posizionato nel *mail server* della casella postale di Bob.
6. Bob, quando ritiene opportuno, invoca il proprio user agent per leggere il messaggio.

Si noti che solitamente SMTP non usa *mail server* intermedi. Ovvero, se il server del mittente si trova a Hong Kong e quello del destinatario a St. Louis, la connessione TCP ha luogo direttamente tra le due città. In particolare, se il *mail server* del destinatario è spento, il messaggio rimane nel *mail server* del mittente e attende un nuovo tentativo. Questo è il significato dell'affermazione **"il messaggio non viene posizionato in alcun *mail server* intermedio"**.

Nell'esempio si è visto come il *client* SMTP (in esecuzione sul *mail server* del mittente) fa stabilire a TCP una **connessione sulla porta 25** verso il server SMTP (in esecuzione sul *mail server* del destinatario).
Una volta stabilita la connessione, il *server* e il *client* effettuano una qualche forma di *handshaking* a livello applicativo, ovvero il *client* inizialmente invia prima l'e-mail del mittente e solo dopo aver ricevuto un messaggio di avvenuta ricezione invia il secondo messaggio contenente l'e-mail del destinatario.
Dopo l'*handshaking*, il *client* invia il messaggio e ripete il processo sulla stessa connessione TCP se ha altri messaggi da inviare al *server*, altrimenti ordina a TCP di chiudere la connessione.

10.3 Livello di trasporto

10.3.1 Definizione

Un **protocollo a livello** di trasporto mette a disposizione una **comunicazione logica** tra processi applicativi di host differenti.

Per **comunicazione logica** si intende che tutto procede come se gli host che eseguono i processi fossero direttamente connessi. In realtà, gli host si possono trovare agli antipodi del pianeta ed essere connessi da numerosi router e da svariati tipi di collegamenti. Infatti, i processi applicativi usano la comunicazione logica fornita dal livello di trasporto per scambiare messaggi, senza preoccuparsi dei dettagli dell'infrastruttura fisica utilizzata per trasportarli.

Il **compito** del **livello di trasporto** è il seguente:

- **Lato mittente:**
  a) Il livello di trasporto **converte i messaggi** che riceve da un processo applicativo in pacchetti a livello di trasporto, noti secondo la terminologia Internet come **segmenti** (*transport-layer segment*);
  b) Successivamente il livello di trasporto passa il segmento al livello di rete, dove viene **incapsulato all'interno di un pacchetto** a livello di rete (si chiamerà datagramma);
  c) Infine, il livello di rete lo invia a destinazione.
- **Lato destinatario:**
  a) Il livello di rete **estrae il segmento** dal datagramma e lo passa al livello superiore, quello di trasporto;
  b) Il livello di trasporto **elabora il segmento** ricevuto, rendendo disponibili all'applicazione destinataria i dati del segmento.

10.3.2 Protocolli usati

I protocolli utilizzati a livello di trasporto sono i due modelli già visti in passato: **UDP** (*user datagram protocol*, capitolo 7.2.2), che fornisce un servizio non affidabile e non orientato alla connessione, e **TCP** (*transmission control protocol*, capitolo 7.2.1), che offre un servizio affidabile e orientato alla connessione.

10.3.3 Struttura dei segmenti nei protocolli

Al livello di trasporto i due protocolli hanno due strutture diverse.

Protocollo UDP

Nel protocollo UDP l'intestazione presenza solo quattro campi di due byte ciascuno, ovvero **16 bit ciascun campo**. L'intestazione occupa 8 byte.

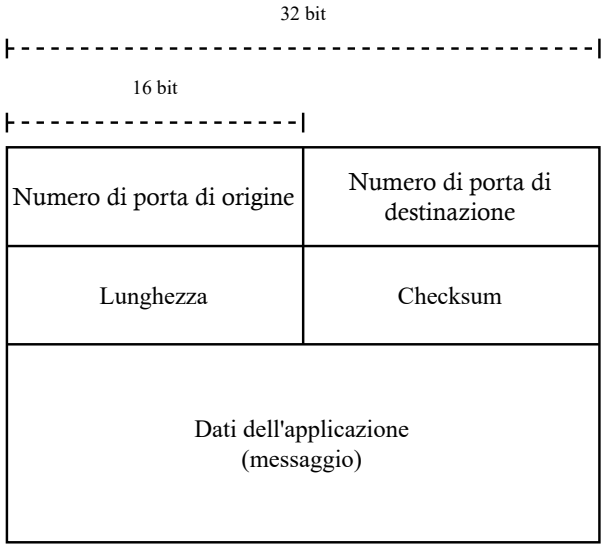


Figura 1: Struttura segmento nel protocollo UDP.

I **numeri di porta** consentono all'host di destinazione di trasferire i dati applicativi al processo corretto.

Il **campo lunghezza** specifica il numero di byte del segmento UDP (intestazione più dati). Un valore esplicito di lunghezza è necessario perché la grandezza del campo dati può essere diversa tra un segmento e quello successivo.

L'host ricevente utilizza il **checksum** per verificare se sono avvenuti errori nel segmento.

Protocollo TCP

Nel protocollo TCP il segmento è formato da campi intestazione e di un campo contenente un blocco di dati proveniente dall'applicazione. I campi intestazione formano in totale 20 byte, ovvero 12 in più rispetto a UDP.

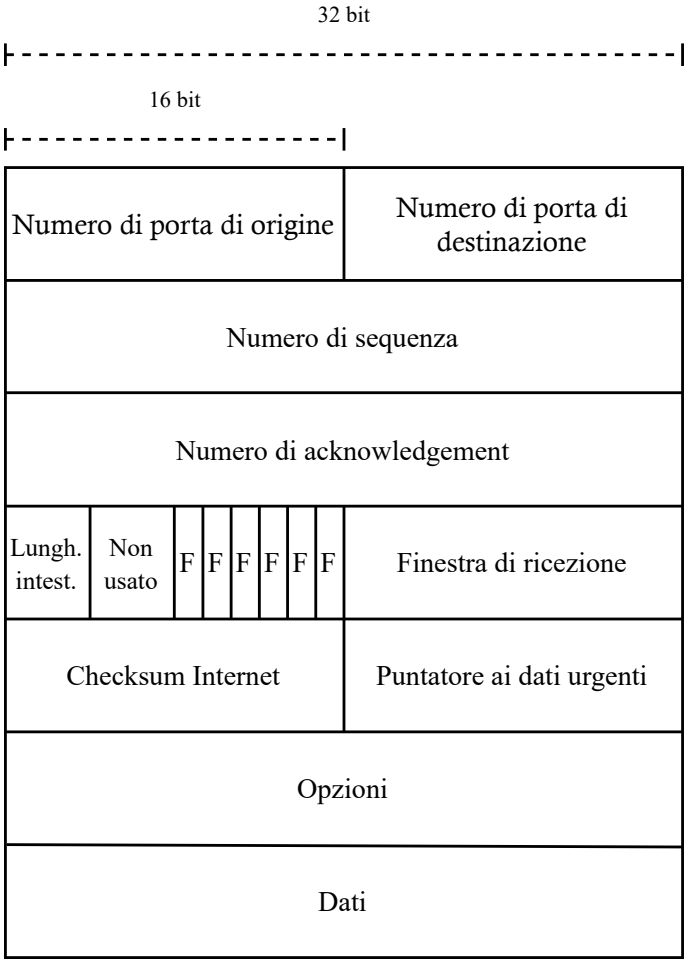


Figura 2: Struttura segmento nel protocollo TCP.

L'**intestazione** include:

- **Numeri di porta di origine** e **di destinazione**, utilizzati per il *multiplexing* (processo di costruzione del messaggio) e *demultiplexing* (processo di scomposizione del messaggio).
- Il campo **numero di sequenza** (*sequence number*) e il campo **numero di acknowledgement** (*acknowledgment number*), entrambi di 32 bit, vengono utilizzati dal mittente e dal destinatario TCP per implementare il trasferimento dati affidabile.

- Il campo **lunghezza dell'intestazione** (*header length*), di 4 bit, specifica la lunghezza dell'intestazione TCP in multipli di 32 bit. L'intestazione TCP ha lunghezza variabile a causa del tempo delle opzioni TCP. Generalmente, il campo delle opzioni è vuoto e, pertanto, la lunghezza consueta è di 20 byte.
- Il campo **flag** (indicato con le lettere F nella figura) è di 6 bit:
  - **ACK** viene usato per indicare che il valore trasportato nel campo di acknowledgement è valido, ovvero il segmento contiene un acknowledgement per un segmento che è stato ricevuto con successo.
  - **RST, SYN, FIN** vengono utilizzati per impostare e chiudere la connessione.
  - **CWR, ECE** sono usati nel controllo di congestione esplicito.
  - **PSH** se ha il valore 1 vuol dire che il destinatario dovrebbe inviare immediatamente i dati al livello superiore.
  - **URG** indica che il segmento presenta dati che sono stati marcati "urgenti" dal livello superiore (solo il mittente ha questa possibilità).
- Il campo **finestra di ricezione** (*receive window*), di 16 bit, viene utilizzato per il controllo di flusso, ovvero per indicare il numero di byte che il destinatario è disposto ad accettare.
- Il campo **checksum** di 16 bit è utilizzato per rilevare errori sui bit in un pacchetto trasmesso.
- Il campo **puntatore ai dati urgenti** è di 16 bit. Quando ci sono dati urgenti, TCP deve informare l'entità destinataria al livello superiore e passarle un puntatore alla fine dei dati urgenti.
- Il campo **opzioni** (*options*) è facoltativo e di lunghezza variabile. Viene utilizzato quando mittente e destinatario negoziano la dimensione massima del segmento (*Maximum Segment Size*) o come fattore di scala per la finestra nelle reti ad alta velocità.

Cos'è MSS?
La massima quantità di dati prelevabili e posizionabili in un segmento viene limitata dalla **dimensione massima di segmento** (MSS, *maximum segment size*). Questo valore viene generalmente impostato determinando prima la lunghezza del frame più grande che può essere inviato a livello di collegamento dall'host mittente locale, ovvero la cosiddetta **unità trasmissiva massima** (MTU, *maximum transmission unit*).

I **campi** presenti all'interno del TCP hanno delle porte. I numeri di porta sono di 16 bit e vanno da 0 a 65535, quelli che vanno da 0 a 1023 sono chiamati **numeri di porta noti** (*well-know port number*) e sono riservati per essere usati da protocolli applicativi ben noti quali HTTP (porta 80) e FTP (porta 21). (l'elenco dei numeri di porta noti è fornito nell'RFC 1700, la sua versione aggiornata si trova al link: iana.org)

Il **numero di sequenza per un segmento** è il numero nel flusso di byte del primo byte di segmento. Per esempio, supponiamo che un processo nell'Host A voglia inviare un flusso di dati a un processo sull'Host B su una connessione TCP. TCP sull'Host A numera implicitamente ogni byte del flusso di dati. Ipotizziamo che il flusso di dati consista in un file da 500'000 byte, che la MSS valga 1000 byte e che il primo byte del flusso sia numerato con 0. TCP costruisce 500 segmenti per questo flusso di dati ($500'000 \div 1000$). Al primo segmento viene assegnato numero di sequenza 0, al secondo 1000, al terzo 2000 e così via. Ogni numero di sequenza viene inserito nel campo numero di sequenza dell'intestazione del segmento TCP appropriato. Il numero di acknowledgement che l'Host A scrive nei propri segmenti è il numero di sequenza del byte successivo che l'Host A attende dall'Host B. Facciamo alcuni esempi.

Supponiamo che l'Host A abbia ricevuto da B tutti i byte numerati da 0 a 535 e che A stia per mandare un segmento all'Host B. L'Host A è in attesa del byte 536 e dei successivi byte nel flusso di dati di B. pertanto, l'Host A scrive 536 nel campo del numero di acknowledgement del segmento che spedisce a B.

Un ulteriore esempio. Supponiamo che l'Host A abbia ricevuto un segmento dall'Host B contenente i byte da 0 a 535 e un altro segmento contenente i byte da 900 a 1000. Per qualche motivo l'Host A non ha ancora ricevuto i byte da 536 a 899. In questo esempio, l'Host A sta ancora attendendo il byte 536 (e i successivi) per ricreare il flusso di dati di B. perciò il prossimo segmento di A destinato a B conterrà 536 nel campo del numero di acknowledgement. Dato che TCP effettua l'acknowledgement solo dei byte fino al primo byte mancante nel flusso, si dice che tale protocollo offre **acknowledgement cumulativi** (*cumulative acknowledgement*).

11 Fasi di connessione TCP, RTO e RTT

11.1 Fasi di connessione TCP

In questo capitolo si vedranno tre scenari: l'instaurazione di una connessione TCP, quindi la fase iniziale, la chiusura di una connessione TCP, quindi la fase finale, e la perdita di un pacchetto.

Nella **fase iniziale** (**handshake a tre vie**), ovvero nella fase di instaurazione della comunicazione, c'è uno scambio di messaggi:

1. [**Segmento SYN**] TCP lato client invia uno speciale segmento al TCP del server.
   Tale segmento è privo di dati, quindi è in **assenza** del campo **Dati**.
   Esso contiene solamente il bit **SYN** a 1 e un **numero di sequenza iniziale** (*client initial sequence number*). Quest'ultimo valore viene generato casualmente dal client e posto nel campo **numero di sequenza** (*sequence number*).
   Infine, viene incapsulato in un datagramma IP e inviato al server.

2. [**Segmento SYNACK**] Una volta che il datagramma IP arriva all'host server (ipotizzando che arrivi), il server estrae il segmento dal datagramma, alloca i buffer e le variabili TCP alla connessione e invia un segmento di connessione approvata al client TCP.
   Il segmento è privo di dati, quindi anch'esso è in **assenza** del campo **Dati**.
   Esso contiene il campo **SYN** a 1, il campo **ACK** e un **numero di sequenza iniziale** (*server initial sequence number*).
   SYN indica che il segmento è ancora nella fase iniziale.
   Numero di acknowledgement acquisisce il valore del *client initial sequence number* incrementandolo di 1.
   Nel campo **numero di sequenza** viene inserito il numero di sequenza iniziale generato casualmente dal server (*server initial sequence number*).

3. [**Segmento ACK**] All'arrivo del segmento SYNACK, anche il client alloca buffer e variabili alla connessione. L'host client invia quindi al server un altro segmento in risposta al segmento di connessione approvata dal server.
   Il segmento questa volta **può** contenere informazioni che vanno dal client verso il server.
   Esso contiene solamente il campo **numero di acknowledgement** con valore *server initial sequence number* incrementato di 1.
   Il bit **SYN** è posto a zero poiché la connessione è stata stabilita.

Una volta completati i tre passi, gli host client e server possono scambiarsi segmenti contenenti i dati con bit **SYN** a zero.

Nella **fase finale**, ovvero nella fase di conclusione e chiusura della connessione, supponendo che il client voglia chiudere la connessione, i messaggi sono:

1. Il processo applicativo client invia un comando di chiusura, che forza il client TCP a inviare un segmento TCP speciale al processo server.
   L'intestazione presenta solo il campo **FIN** ad 1.

2. Quando il server riceve il segmento di chiusura, risponde inviando un acknowledgement al client. Quindi il server invia un vero e proprio segmento di shutdown.
   Ancora una volta, l'intestazione presenta solamente il campo **FIN** ad 1.

3. Infine, come accade nella fase iniziale, il client conclude la fase finale inviando un ultimo messaggio di acknowledgement come risposta.

Alla fine di questo scambio, tutte le risorse degli host risultano deallocate.

11.2 Caso di perdita: RTO e calcolo RTT

Prima di parlare del caso in cui un pacchetto viene perso nella connessione di tipo TCP, è doveroso introdurre l'indice RTO e RTT.

L'**indice RTT** (*Round Trip Time*) è la quantità di tempo che intercorre tra l'istante di invio del segmento (ossia quando viene passato a IP) e quello di ricezione dell'acknowledgement del segmento.

Invece di misurare per ogni segmento il suo RTT, il protocollo TCP valuta il *round trip time* solo per uno dei segmenti trasmessi e per cui non si è ancora ricevuto acknowledgement. Questo comporta la misurazione di un nuovo valore a ogni RTT, circa.

Ovviamente, tali valori variano a seconda del segmento, a seconda della congestione nei router e a seconda del carico sui sistemi periferici. A causa di questa fluttuazione, ogni valore RTT può essere atipico. Per effettuare una stima naturale, si calcola una media di valori del RTT, ovvero ***Estimated*** RTT. Tale valore è una media ponderata dei valori del RTT ed attribuisce maggiore importanza ai valori recenti rispetto a quelli vecchi. Tuttavia, è del tutto normale questa valutazione poiché i primi riflettono meglio la congestione attuale della rete.

L'**indice RTO** (*Retrasmission Time-Out*) è il tempo di ritrasmissione di un pacchetto. All'invio di un pacchetto, parte un timer che indica il range di tempo in cui il server deve rispondere. Nel caso in cui il server non riesca a rispondere entro questo intervallo, viene inviato nuovamente lo stesso pacchetto al server.

Questo indice viene impostato prendendo il *Estimated* RTT e aggiungendo un certo margine che dovrebbe essere grande quando c'è molta fluttuazione dei valori RTT e piccolo in caso contrario. Inizialmente il valore è pari a 1 secondo.

In caso di perdita, TCP **ritrasmette il segmento con il più basso numero di sequenza che non abbia ancora ricevuto acknowledgement**. Tuttavia, ogni volta che questo si verifica, TCP imposta il successivo RTO al doppio del valore precedente. In caso di perdite continue, RTO continua a raddoppiare.

Uno dei casi in cui RTO aumenta è dovuto al fatto che la rete potrebbe essere congestionata, ovvero troppi pacchetti arrivano presso una (o più) code dei router nel percorso tra l'origine e la destinazione, provocando l'eliminazione dei pacchetti e/o lunghi ritardi di accodamento.

11.3 Esame - Blocco CIDR, subnetting e broadcast

Tema d'esame - 05/07/2013, esercizio 2

Si determini il blocco CIDR più piccolo della rete sapendo che si devono creare 3 reti LAN con i seguenti vincoli:

- LAN1 con 300 host
- LAN2 con 40 host
- LAN3 con 90 host

Sapendo che l'indirizzo di broadcast della LAN3 è: 148.12.79.255.

Prima di tutto, si ricorda che l'indirizzo broadcast è un indirizzo in cui il suffisso, nella forma binaria, è formato da soli bit posti a 1.

Il **primo passo** degli esercizi con il blocco CIDR e il *subnetting* è stabilire il numero di bit da assegnare a ciascuna rete seguendo i vincoli imposti:

- LAN1 necessita di 300 host e l'unica potenza del 2 che copre tale numero è $2^9 = 512$;
- LAN2 necessita di 40 host e l'unica potenza del 2 che copre tale numero è $2^6 = 64$;
- LAN3 necessita di 90 host e l'unica potenza del 2 che copre tale numero è $2^7 = 128$.

La somma dei risultati delle potenze, ovvero il numero di indirizzi che dovrà almeno avere questa rete è:

$$512 + 64 + 128 = 704$$

L'unica potenza del 2 che riesce a coprire tale valore è 2^{10}. Per cui servono 10 bit di suffisso per ricoprire almeno 704 host.

Grazie a questo dato si può ricavare il prefisso di rete: $2^{32} \div 2^{10} = 2^{22}$. A questo punto ci sono tutti i dati per costruire l'indirizzo di rete partendo dall'unico indirizzo derivato da esso, ovvero l'indirizzo di broadcast.

Si effettua la traduzione dell'indirizzo di broadcast (148.12.79.255) in binario:

$$10010100\,.\,00001100\,.\,01001111\,.\,11111111$$

Sapendo che i primi 22 bit sono di prefisso, poniamo i restanti bit di suffisso a zero e otteniamo il blocco di rete CIDR più piccolo:

$$10010100\,.\,00001100\,.\,01001100\,.\,00000000$$

Convertendo nuovamente l'indirizzo in notazione decimale puntata, il **blocco CIDR più piccolo** è: 148.12.76.0/22.

Per trovare le 3 LAN si procede con la creazione di sottoreti partendo dall'indirizzo di rete appena trovato e aumentando il prefisso:

$$10010100\,.\,00001100\,.\,01001100\,.\,00000000/23$$
$$10010100\,.\,00001100\,.\,01001110\,.\,00000000/23$$

Il primo indirizzo trovato si può subito assegnare alla LAN1 che richiedeva almeno 300 host e con un indirizzo del genere ne vengono messi a disposizione massimo (2^9) 512. Quindi, l'indirizzo della **LAN1 in notazione decimale puntata**: 148.12.76.0/23.

Con il secondo indirizzo si costruiscono nuovamente altre due sottoreti per cercare di rispettare i vincoli imposti:

$$10010100\,.\,00001100\,.\,01001110\,.\,00000000/24$$
$$10010100\,.\,00001100\,.\,01001111\,.\,00000000/24$$

Gli indirizzi trovati non soddisfano nessun vincolo. Precisamente entrambi potrebbero contenere sia la LAN numero 2 e 3, ma non sarebbero ottimizzate perché con il suffisso di 7 bit si avrebbero 128 indirizzi contro gli 8 bit con 256 indirizzi.

Per cui, si utilizza il primo indirizzo per suddividere ancora la rete e il secondo indirizzo si lascia libero per operazioni future. Quindi, un'altra suddivisione utilizzando solamente il primo indirizzo porterebbe a:

$$10010100\,.\,00001100\,.\,01001110\,.\,00000000/25$$
$$10010100\,.\,00001100\,.\,01001110\,.\,10000000/25$$

Il primo indirizzo soddisfa il vincolo imposto dalla terza LAN, per cui scrivendo l'indirizzo della **LAN3 in notazione decimale puntata**: 148.12.78.0/25.

Per l'ultima LAN serve creare un'altra sottorete partendo dal secondo indirizzo, quindi:

$$10010100\,.\,00001100\,.\,01001110\,.\,10000000/26$$
$$10010100\,.\,00001100\,.\,01001110\,.\,11000000/26$$

Il primo indirizzo rispetta i vincoli imposti dalla seconda LAN, per cui scrivendo l'indirizzo della **LAN2 in notazione decimale puntata**: 148.12.78.128/26. Infine, il secondo indirizzo rimane libero per il futuro.

12 Controllo di flusso (TCP), finestra e ripetizione selettiva

12.1 Controllo di flusso (TCP)

In una comunicazione con connessione di tipo TCP, gli host posseggono un buffer (registro temporaneo, chiamato anche *buffer di ricezione*) in cui vengono salvati i *byte* corretti e in sequenza. Ovviamente tale registro ha una capienza limitata e l'applicazione mittente potrebbe mandare in overflow il buffer di ricezione del destinatario inviando molti dati troppo rapidamente.

Per risolvere l'overflow del buffer di ricezione, il protocollo TCP offre un **servizio di controllo di flusso** (*flow-control service*). Esso è un servizio di confronto sulla velocità poiché paragona la frequenza di invio del mittente con quella di lettura dell'applicazione ricevente.

***Attenzione!* Flusso vs Congestione**

Esiste una sostanziale **differenza** tra il **controllo di flusso** e il **controllo di congestione**.
Nel **controllo di flusso**, il protocollo evita che il buffer di ricezione del destinatario vada in overflow.
Nel **controllo di congestione** (*congestion control*), il protocollo evita che ci siano carichi troppi elevati e code troppo consistenti sui buffer, causa principale di rallentamenti nella rete.

TCP garantisce il controllo di flusso grazie all'obbligo di mantenimento di una variabile chiamata **finestra di ricezione** (*receive window*) gestita dal mittente. Essa fornisce al mittente un'indicazione dello spazio libero disponibile nel buffer del destinatario. Per approfondire l'argomento e vedere la sua applicazione, nella prossima pagina verrà effettuato un esempio di trasferimento di file.

Esempio

Supponiamo che l'Host A stia inviando un file di grandi dimensioni all'Host B su una connessione di tipo TCP, quindi affidabile e con controllo di flusso.

TCP obbliga l'Host B ad allocare un **buffer di ricezione** per la connessione, la dimensione di tale buffer sarà denotata con **RcvBuffer** (*Receive Buffer*) ed ogni tanto il processo applicativo dell'Host B legge dal buffer.

Si definiscono le seguenti variabili:

- **LastByteRead**: numero dell'**ultimo byte**, nel flusso di dati, che il processo applicativo in B ha **letto dal buffer**.
- **LastByteRcvd**: numero dell'**ultimo byte**, nel flusso di dati, che proviene dalla rete e che è stato **copiato nel buffer di ricezione** di B.

Dato che il protocollo TCP **evita** che il buffer allocato vada in overflow, si può affermare con certezza la seguente disuguaglianza:

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

La **finestra di ricezione**, indicata con **rwnd**, viene impostata alla quantità di spazio disponibile nel buffer, ovvero:

$$rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

Dato che lo spazio disponibile varia col tempo, rwnd è dinamica.

Adesso che sono state definite le basi, come viene usata la variabile rwnd per offrire il servizio di controllo di flusso?

L'Host B comunica all'Host A quanto spazio disponibile è presente nel buffer della connessione, scrivendo il valore corrente di rwnd nel campo apposito dei segmenti che manda ad A. L'Host B inizializza rwnd con il valore RcvBuffer e, ovviamente, deve tenere traccia di variabili specifiche per ogni connessione.

A sua volta, l'Host A tiene traccia di due variabili, **LastByteSent** e **LastByteAcked**, il cui significato è rispettivamente "**ultimo byte mandato**" e "**ultimo byte per cui si è ricevuto acknowledgement**".

Si noti che la differenza tra i valori di queste due variabili esprime la **quantità di dati spediti da A per cui non si è ancora ricevuto un acknowledgement**. Mantenendo la quantità di dati senza acknowledgement sotto il valore di rwnd, si garantisce che l'Host A non mandi in overflow il buffer di ricezione dell'Host B. Quindi, l'Host A si assicura che per tutta la durata della connessione sia rispettata la disuguaglianza:

$$LastByteSent - LastByteAcked \leq rwnd$$

Tuttavia, questo esempio presenta un problema. Si supponga che l'Host B si riempia, quindi con rwnd uguale a zero, notifichi l'Host A e, infine, non abbia più nulla da inviare ad A. Quello che succede è che TCP non invia nuovi segmenti con nuovi valori di rwnd poiché l'Host B non ha più dati da inviare o acknowledgement da mandare. Questo comporta che l'Host A non venga notificato nel momento in cui l'Host B svuoti il buffer. Quindi, si crea un'inabilitazione dell'Host A!

Per **risolvere tale problema** basta che l'Host A continui a inviare periodicamente segmenti con un byte di dati quando la finestra di ricezione di B è zero. Il destinatario, ovvero l'Host B, risponderà a questi segmenti con un acknowledgement. Prima o poi il buffer inizierà a svuotarsi e i riscontri conteranno un valore non nullo per rwnd.

In sintesi, c'è un continuo scambi di messaggi per controllare che la finestra sia sempre uguale a zero.

12.2 Ripetizione selettiva

I **protocolli a ripetizione selettiva** (SR, *selection-repeat protocol*) *evitano* le ritrasmissioni non necessarie facendo ritrasmettere al mittente solo quei pacchetti su cui esistono sospetti di errore, ovvero smarrimento o alterazione.

Questo protocollo **obbliga** il destinatario ad inviare acknowledgement specifici per i pacchetti ricevuti in modo corretto.

Per esempio, se un segmento non ricevesse l'acknowledgement relativo dal destinatario, allo scadere del RTO (*Retrasmission Time-Out*) entrerebbe in gioco il protocollo di ripetizione selettiva, il quale invierebbe al destinatario solamente quel pacchetto che non ha ricevuto il suo acknowledgement.

13 Algoritmo di controllo di congestione di TCP

Generalmente TCP capisce quando aumentare e diminuire la velocità di trasmissione dei dati da parte del mittente grazie ad alcune linee guida:

- Un segmento perso implica congestione, quindi i tassi di trasmissione del mittente TCP dovrebbero essere decrementati quando un segmento viene perso.
- Un acknowledgement indica che la rete sta consegnando i segmenti del mittente al ricevente e quindi il tasso di trasmissione del mittente può essere aumentato quando arriva un acknowledgement non duplicato.
- Rilevamento della larghezza di banda tramite alcune tecniche.

Nonostante tali linee guida, esiste un **algoritmo di congestione di TCP** per controllare la congestione. Esso presenta tre componenti o fasi principali: ***slow start***, ***congestion avoidance*** e ***fast recovery***. I primi due componenti sono obbligatori di TCP e differiscono nel modo in cui aumentano la grandezza di cwnd[1] in risposta agli acknowledgement ricevuti. Al contrario, la *fast recovery* è suggerita, ma non obbligatoria, per i mittenti TCP.

[1] *congestion window*, finestra di congestione, ovvero la finestra di dati trasmessi e non ancora riscontrati dal mittente poiché risposta acknowledgement mancante

13.1 Slow start

Quando si stabilisce una connessione TCP, il valore di cwnd viene in genere inizializzato a 1 MSS (*Maximum Segment Size*). Questo comporta una velocità di invio iniziale di circa MSS/RTT.

Per esempio, se $MSS = 500$ Byte e $RTT = 200$ ms, la velocità iniziale è solo di circa 20 kbps. Tuttavia, la banda disponibile inizialmente potrebbe essere molto più grande. Per questo motivo, durante la fase iniziale, il valore della ***congestion window*** (cwnd) **parte da 1 MSS e si incrementa di 1 MSS ogni volta che un segmento trasmesso riceve un acknowledgement** (detta ***slow start*** per l'inizio adagio).

Per esempio, nel momento di scambio di dati tra due host dopo l'handshaking, accade che:

1. Il mittente invia il **primo segmento** nella rete e attende un riscontro:

   a. Se il mittente riceve un acknowledgement del segmento, quindi **senza** avere nessuna **perdita**, esso incrementa la finestra di congestione di 1 MSS.
   Adesso il mittente ne potrà inviare due di segmenti poiché la sua cwnd sarà di 2 MSS.
   Assumendo che vengano inviati i due segmenti e si ricevano i due acknowledgement rispettivi, il destinatario, come risposta, comunica la grandezza della cwnd a 4 poiché per ogni segmento ricevuto correttamente ha incrementato la finestra di 1.
   E così via.

   b. Se il mittente non riceve un acknowledgement del segmento appena inviato, si manifesta un evento di **perdita** (e quindi di congestione). In questo caso, il mittente pone il valore della cwnd pari a 1 e inizia nuovamente il processo di *slow start* (punto a).
   Oltre ad iniziare da capo, **modifica il valore della variabile ssthresh** (*slow start threshold*, in italiano "soglia di *slow start*") a **cwnd** $\div$ 2: ovvero metà del valore che aveva la finestra di congestione quando è stata rilevata una perdita, o meglio una congestione.

La conseguenza dell'evento "1.*a*", cioè senza perdita, ha come effetto il raddoppio della velocità trasmissiva a ogni RTT. È possibile affermare che nel protocollo TCP, la **velocità di trasmissione parte lentamente, ma cresce in modo esponenziale** durante la fase di *slow start*.

La fase di *slow start* può **terminare** con una perdita: come spiegato al punto *b*, oppure quando il valore della cwnd raggiunge o supera il valore della ssthresh. Il motivo è dovuto al fatto che ssthresh è il limite e il suo superamento aumenta le probabilità di congestione, ovvero di perdita.

Una volta terminata la *slow start*, TCP entra nella fase di ***congestion avoidance***.

13.2 Congestion avoidance

Quando TCP entra nello stato di ***congestion avoidance*** adotta un approccio più conservativo. Invece di raddoppiare il valore di cwnd ogni RTT come nello *slow start*, l'aumento della cwnd avviene di 1 MSS ogni RTT.

L'**aumento** può avvenire tramite il seguente metodo (più comune).
Il mittente incrementa la propria **cwnd** di $MSS \times \left(\frac{MSS}{\text{cwnd}}\right)$ byte ogni qualvolta riceva un nuovo acknowledgement.
Per esempio, supponendo che MSS vale $1'460$ byte e cwnd $14'600$ byte, allora in un RTT vengono spediti dieci segmenti. Questo calcolo è possibile eseguirlo mentalmente poiché se i dati da trasmettere devono essere $14'600$, ma ogni segmento può avere al massimo una grandezza di $1'460$, allora considerando un segmento come $1'460$ byte, si dovranno inviare 10 segmenti per avere il pacchetto completo.
In questo esempio, la cwnd aumenterà di 1/10 della MSS ogni qualvolta riceverà 1 acknowledgement di un segmento inviato. Quindi una volta ricevuti tutti i 10 acknowledgement dei 10 segmenti inviati, la cwnd sarà aumentata di 1 MSS, ovvero di 1460 byte.

Al contrario, la fase di incremento si **interrompe** quando si esaurisce il tempo del RTO (*Retrasmission Time-Out*). Il comportamento successivo all'interruzione è identico a quello dello *slow start*:

- Il valore di **cwnd** è posto uguale a 1 MSS;
- Il valore di **ssthresh** viene impostato alla metà del valore di cwnd al momento del time-out.

Infine, TCP entra nella fase di *fast recovery*.

13.3 Fast recovery

Una volta terminato il periodo di attesa del RTO si passa dallo stato di ***fast recovery*** a quello di *slow start* dopo avere effettuato le stesse azioni presenti sia in *slow start* che in *congestion avoidance*. Queste azioni sono: il valore di cwnd posto a 1 MSS e il valore di ssthresh è impostato a metà del valore di cwnd nel momento in cui si è riscontrato l'evento di perdita.

Attenzione! Questo metodo **non** è obbligatorio, viene solo raccomandato. Una prima versione di TCP, nota come **TCP Tahoe**, portava la finestra di congestione a 1 MSS ed entrava nella fase di *slow start* dopo qualsiasi tipo di evento di perdita. La versione più recente, **TCP Reno**, adotta invece *fast recovery*.

14 Dettaglio dell'algoritmo di controllo di congestione di TCP

14.1 Algoritmo

Nello scorso capitolo è stato introdotto l'algoritmo di controllo di congestione del protocollo TCP. Al contrario, in questo capitolo l'algoritmo verrà approfondito e utilizzato per svolgere esercizi.

Prima di iniziare la descrizione dell'algoritmo, è necessario tenere in **considerazione quattro variabili**:

- **CWND** (*Congestion Window*), ovvero la dimensione della finestra della trasmissione.
- **RTO** (*Retrasmission Time-Out*), calcolato dinamicamente, è il tempo che intercorre dall'invio del segmento al momento in cui il protocollo realizza che c'è un evento di congestione (paragrafo 11.2).
- **RCV WND** (*Receive Window*), ovvero la dimensione massima della finestra di ricezione.
- **SSTHRESH** (*Slow Start Threshold*), ovvero la soglia per capire se utilizzare l'algoritmo *slow start* o passare all'algoritmo *congestione avoidance*.

E nella fase di **inizializzazione**:

- CWND = 1;
- RCV WND = comunicato dalla destinazione la quale inserisce il valore di tale finestra nel header TCP, precisamente il campo *window*;
- SSTHRESH = $RCV\ WND$ oppure $RCV\ WND \div 2$.

Ora che sono state effettuate delle doverose spiegazioni delle variabili necessarie e sono state inizializzate, è possibile **descrivere l'algoritmo di controllo della congestione**:

1. **Effettuare la numerazione dei segmenti** finché non si raggiunge la dimensione delle CWND (*congestion window*) ed **inviare i segmenti** al destinatario;
2. **Quando arrivano tutti i riscontri** relativi ai segmenti inviati al punto precedente, si controlla **se il valore della CWND è minore** al valore del limite, ovvero **della SSTHRESH**:
   (a) Se CWND è minore della SSTHRESH allora si avvia la fase di *slow start* e la CWND diventa uguale al valore **minimo** tra:
      i. CWND_{old} + numero di ACK ricevuti
      ii. SSTHRESH
      iii. RCV WND
   (b) Se CWND è maggiore della SSTHRESH allora si avvia la fase di *congestion avoidance* e il valore della CWND diventa uguale al valore **minimo** tra:
      i. $\text{CWND}_{old} + \frac{\text{numero di ACK ricevuti}}{\text{CWND}_{old}}$
      ii. RCV WND
3. Al contrario, in caso di **mancanza di un riscontro** di un segmento inviato al punto uno:
   (a) Si aspetta la fine del tempo di *time-out* scandito dal RTO;
   (b) Una volta finito il tempo della RTO, la SSTHRESH acquisisce il valore della CWND diviso 2 al momento esatto della perdita;
   (c) Il nuovo valore della CWND assumerà un valore diverso a seconda dell'algoritmo applicato:
      i. Con la versione **TCP Tahoe**, ovvero la meno recente e quella utilizzata durante il corso, il nuovo valore sarà:
      $$\text{CWND} = 1$$
      ii. Con la versione **TCP Reno**, ovvero la più recente che utilizza la *fast recovery*:
      $$\text{CWND} = \text{SSTHRESH}$$
   (d) Una volta cambiati i valori necessari, i segmenti vengono ritrasmessi e il nuovo valore RTO diventa il valore precedente moltiplicato per due:
   $$\text{RTO}_{new} = \text{RTO} \times 2$$
   (e) Infine, si torna al punto *a*.

14.2 Sintesi algoritmo

Data l'eventuale difficoltà nel leggere lo schema, si lascia qui di seguito un diagramma di flusso:

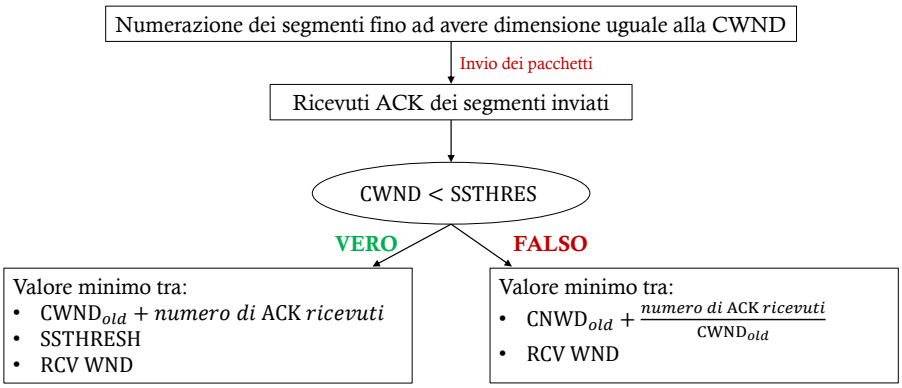


Figura 3: Algoritmo di congestione del protocollo TCP.

E uno aggiuntivo che si riferisce alle scelte da applicare nel momento in cui vengano persi dei segmenti:

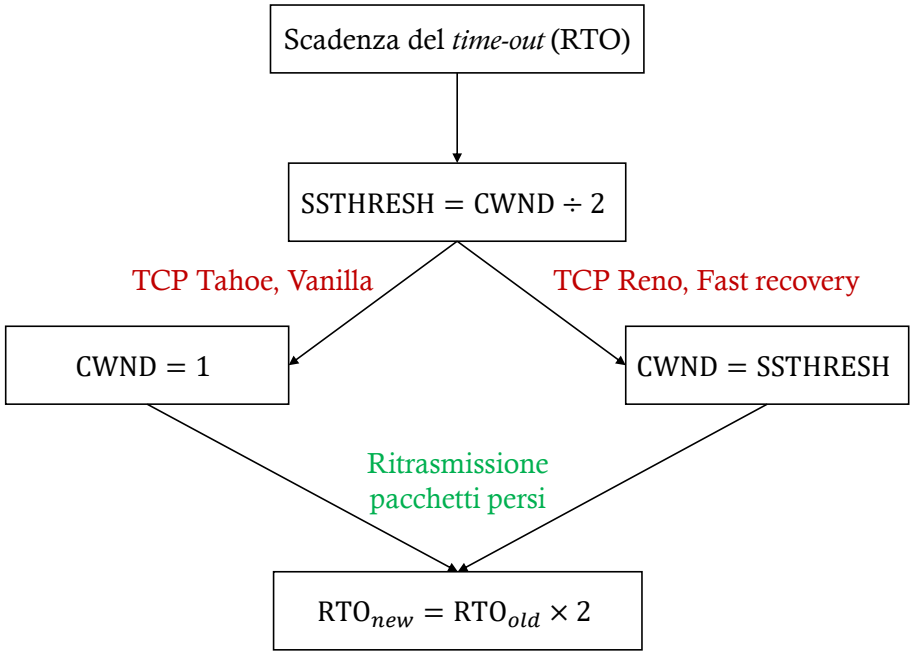
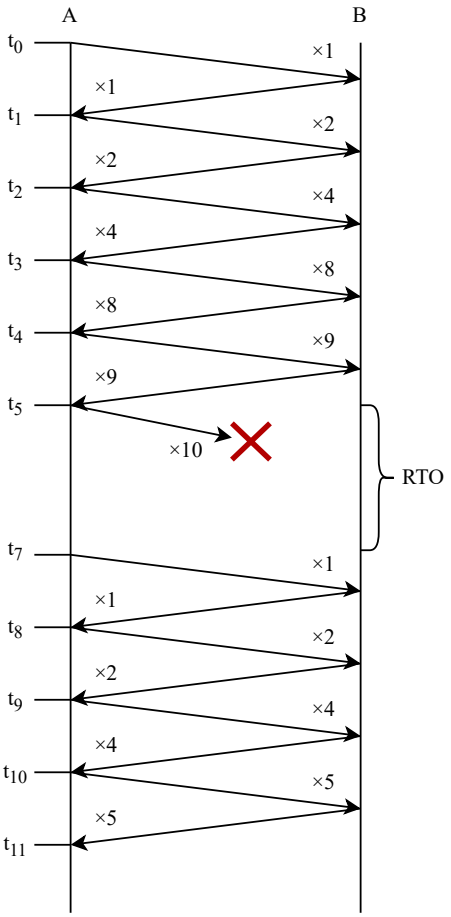


Figura 4: Algoritmo di congestione del protocollo TCP in **caso di perdita**.

14.3 Esempio di congestione

In questo esempio, si simula l'invio di una serie di pacchetti per studiare lo *slow start* e la *congestion avoidance*. Il mittente A invia dei segmenti al destinatario B; il protocollo TCP inizia con l'algoritmo *slow start*. La RCV WND viene posta e rimane a 16 durante tutto l'esempio, mentre la SSTHRESH parte con il valore 8.

Al tempo t_0 viene inviato il primo segmento dato che la CWND è posta a 1.



All'arrivo del primo ACK, tempo t_1, l'algoritmo confronta i valori della CWND e della SSTHRESH: la prima è minore della seconda poiché $1 < 8$; per cui si confrontano i tre valori scegliendo il minore. Tra 2 (CWND + #ack ricevuti), 8 (SSTHRESH) e 16 (RCV WND), il valore più piccolo è 2. Quest'ultimo valore diventerà il nuovo valore della *congestion window*: $\text{CWND}_{new} = 2$.

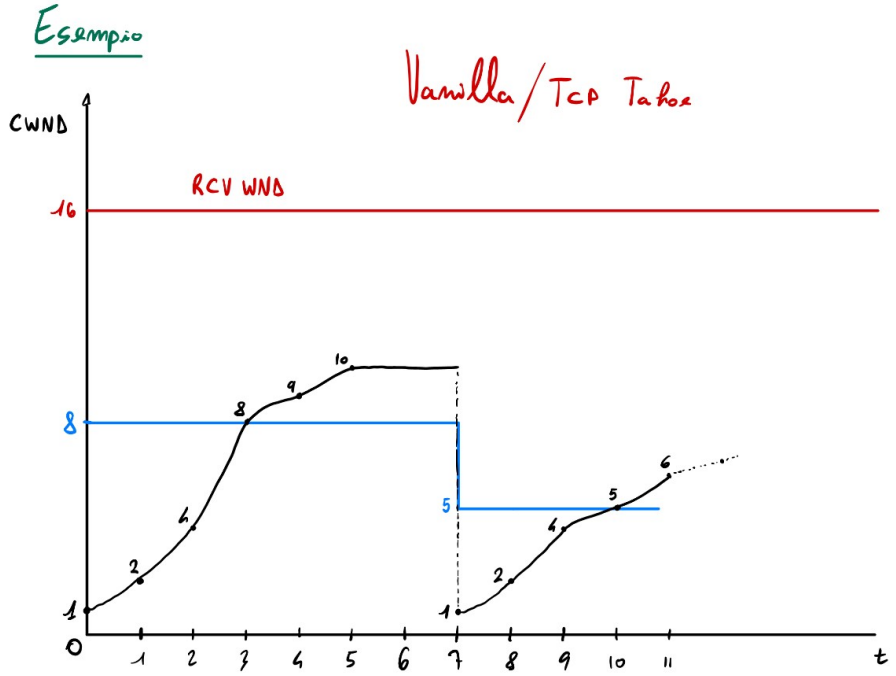
Al tempo t_2, vengono inviati due segmenti e il processo spiegato precedentemente si ripete.

Il cambiamento degno di nota è al tempo t_4 quando la CWND passa dal valore 8 al valore 9. Questo accade poiché al tempo t_4 la CWND è uguale alla SSTHRESH e, seguendo l'algoritmo, si dovrà quindi scegliere il valore minimo tra: 9 ($\frac{\text{CWND}_{old}+(\#\text{ack ricevuti})}{\text{CWND}_{old}}$) e 16 (RCV WND). Il valore minimo è 9, la CWND_{new} diventa 9 e vengono trasmessi i segmenti.

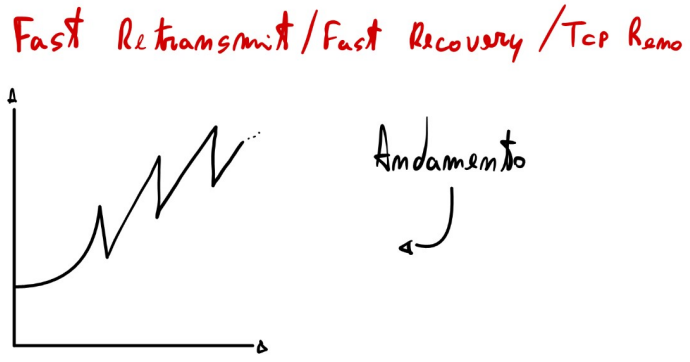
Un'altra osservazione doverosa è quello che succede al tempo t_5. Il TCP rileva una perdita poiché alla fine del RTO non sono stati ancora ricevuti gli ACK. In quel momento entra in gioco la fase di "mancanza di riscontro", ovvero: il valore della SSTHRES diventa il valore della CWND, al momento della perdita, diviso 2; la CWND_{new} diventa pari a 1 (uso del TCP Tahoe o Vanilla); RTO dei pacchetti ritrasmessi diventa $\text{RTO} \times 2$.

Infine, al tempo t_7 si inizia nuovamente con lo *slow start*.

Qui di seguito si lascia lo schema utilizzato anche per la risoluzione degli esercizi di questo tipo. Esso presenta sull'asse delle ordinate (y) il valore della CWND, mentre sull'asse delle ascisse (x) il tempo. Con la linea blu viene rappresentato lo SSTHRESH, mentre con il rosso la RCV WND costante per tutto l'esempio.



È interessante notare la differenza nel caso in cui si utilizzi la versione di TCP Reno, ovvero la più recente. Essa utilizza la *Fast Recovery*, una tecnica spiegata nel capitolo precedente. L'andamento è molto variabile e qui di seguito si rappresenta con un grafico approssimativo.



14.4 Esercizio sul controllo della congestione TCP

14.4.1 Esercizio 1

L'applicazione A deve trasferire all'applicazione B 96′000 Byte. La connessione è già stata instaurata. I dati disponibili sono:

- MSS = 1′000 Byte
- RCV WND = 32′000 Byte costante
- SSTHRESH = RCV WND $\div$ 2
- RTT = 0, 50 secondi costante
- RTO = RTT $\times$ 2 e raddoppia in caso di perdite sequenziali

Infine, negli intervalli aperti $3 - 3.5$ e $7 - 7.5$ è presente un down di rete.

Il **primo passo** per risolvere l'esercizio è la numerazione dei segmenti. Per farlo basta prendere il numero dei Byte da trasferire e dividere il valore per la MSS:

$$\textbf{Numero di segmenti da inviare} \longrightarrow 96'000 \div 1'000 = 96 \text{ segmenti}$$

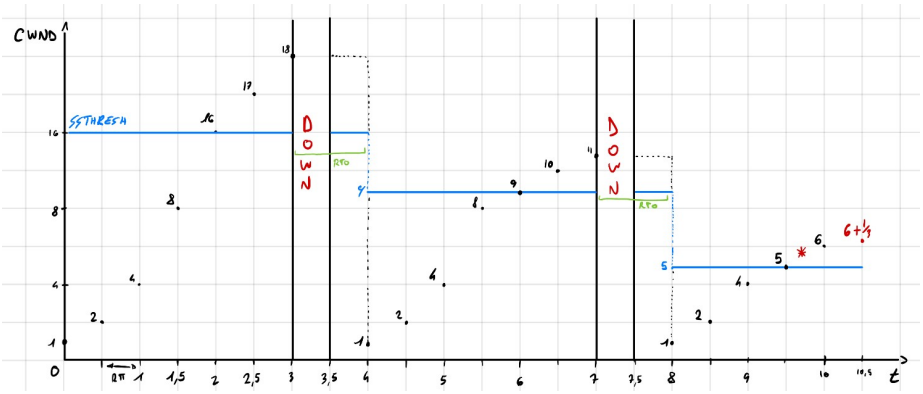
Prima di passare al secondo passaggio e osservare il numero di segmenti da inviare, è necessario ottenere il valore della RCV WND (finestra massima di ricezione) per avere conseguentemente anche la CWND. Essa è possibile calcolarla facendo una semplice divisione tra il numero di Byte della RCV WND e della MSS:

$$\textbf{RCV WND} \longrightarrow 32'000 \div 1'000 = 32 \text{ segmenti}$$

E la SSTHRESH, come scritto nei dati del testo dell'esercizio, è ottenibile dividendo per due la *receive window*:

$$\textbf{SSTHRESH} \longrightarrow 32 \div 2 = 16 \text{ segmenti}$$

Il **secondo passo** necessita di uno schema. Esso ha nell'asse delle ascisse (x) il tempo e nell'asse delle ordinate (y) la CWND.



All'inizio i pacchetti trasmessi seguono l'algoritmo slow start e c'è poco da dire. La somma dinamica, ovvero la somma dei segmenti inviati con successo necessaria per capire quando concludere la connessione, al tempo $t_{1,5}$: $1+2+4+8 = 15$.

Successivamente la CWND diventa come la SSTHRESH e l'avanzata diventa più cauta, notabile all'aumento di 1 valore per volta dal tempo t_2 fino al tempo t_3.

A questo punto, l'**evento down** non permette ai 18 *segmenti* di giungere correttamente a destinazione. Il protocollo TCP, alla fine del RTO entra nella fase di recupero del segmento perso: la SSTHRESH viene posta a 9 ($\mathsf{CWND} \div 2 \rightarrow 18 \div 2$), la nuova $\mathsf{CWND}_{new} = 1$ e il nuovo $\mathsf{RTO}_{new} = \mathsf{RTO} \times 2$. Dato che l'RTO è ad 1 perché $0,5 \cdot (\mathsf{RTT}) \times 2$, all'incirca al tempo t_4 riparte la fase di *slow start*.

Attenzione! Nella somma dinamica, i 18 *segmenti* persi non vengono contati e quindi il risultato fino al tempo t_3 è $1 + 2 + 4 + 8 + 16 + 17 \cancel{+18} = 48$.

Questo evento di *down* si ripete anche al tempo t_7.

L'**ultimo passo** è la fase di chiusura della connessione. Al tempo t_{10} la CWND viene posta a 6 come impone l'algoritmo. Effettuando la somma dinamica si ha come risultato:

$$1 + 2 + 4 + 8 + 16 + 17 + 1 + 2 + 4 + 8 + 9 + 10 + 1 + 2 + 4 + 5 = 94$$

Se i segmenti inviati sono 94, ne servono 2 per giungere a 96, ovvero all'obbiettivo dell'esercizio. Quindi, al tempo t_{10} vengono inviati 2 segmenti nonostante la grandezza della CWND è posta a 6. A questo punto, una volta ricevuti gli ultimi ACK dei segmenti inviati, al tempo $t_{10,5}$ (dopo mezzo secondo perché RTT è di $0,50$ secondi) è possibile chiudere la connessione.

La chiusura avviene dunque al tempo $t_{10,5}$ con l'aggiornamento della CWND: la CWND è maggiore della SSTHRESH, il valore minimo da scegliere sarà $\frac{\mathsf{CWND}_{old} + (\#\text{ack ricevuti})}{\mathsf{CWND}_{old}}$ (e non la RCV WND), ovvero $6 + \frac{2}{6}$. **Solo dopo aver aggiornato la CWND, la connessione si può dire conclusa**.

14.4.2 Esercizio 2

L'applicazione A trasferisce 46′500 Byte all'applicazione B. I dati sono:

- MSS = 1′500 Byte
- RCV WND = 24′000 Byte costante
- $\text{SSTHRESH}_{iniziale} = \text{RCV WND}_{iniziale} \div 2$
- RTT = 0, 50 secondi costante
- RTO = RTT × 2 e raddoppia in caso di perdite consecutive

Infine, gli eventi che si manifestano sono due *down* di rete al tempo $1,5 - 3,5$ e $7 - 7,5$.

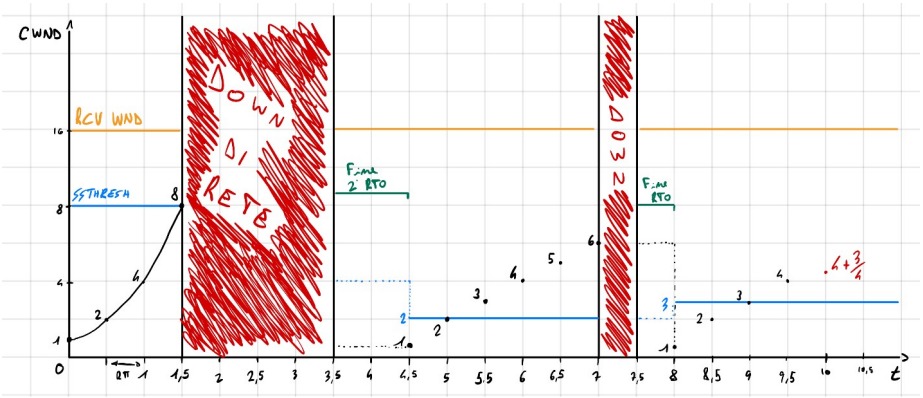
Il **primo passo** è l'enumerazione dei segmenti da inviare:

$$\text{dati da inviare} \div \text{MSS} = 46'500 \div 1'500 = 31 \text{ segmenti}$$

E il calcolo della **SSTHRESH** partendo dalla RCV WND:

$$\begin{array}{lcl} \textbf{RCV WND} & = & 24'000 \div 1'500 = 16 \text{ segmenti} \\ \textbf{SSTHRESH} & = & \text{RCV WND} \div 2 = 16 \div 2 = 8 \text{ segmenti} \end{array}$$

E adesso si lascia qui di seguito il grafico contenente l'andamento della CWND e del resto:



Ci sono solo due osservazioni importanti degne di nota. La prima riguarda il primo down di rete. Al tempo $t_{2,5}$, il TCP del mittente A non riceve i riscontri dei segmenti inviati e viene dunque dimezzata la SSTHRESH (CWND $\div$ 2), la CWND riparte da 1 e RTO diventa il doppio, quindi da 1 passa a 2. La parte interessante è questa: alla fine del secondo RTO (tempo $t_{4,5}$), il TCP del mittente non ha ancora ricevuto i pacchetti, per cui dovrebbe diminuire *di nuovo* la SSTHRESH dividendo la CWND per due. Tuttavia, dato che la CWND è al valore minimo, non avrebbe senso portare la SSTHRESH a $0,5$, per cui si imposta il **valore minimo** della SSTHRESH che corrisponde a 2.

A questo punto inizia l'algoritmo di *slow start* e l'invio dei pacchetti.

La seconda osservazione riguarda la chiusura della connessione che non necessita di commenti particolari ma di una piccola precisazione per comprendere meglio. Al tempo $t_{9,5}$ vengono inviati 3 segmenti nonostante la CWND sia di 4. Questo accade poiché ne mancavano soltanto 3 al conteggio finale.