# 1 Il Livello Trasporto

I protocolli di livello Trasporto forniscono la comunicazione logica tra processi applicativi, detta anche comunicazione *end-to-end*.

Il suo obiettivo é quello di frammentare e riordinare i segmenti inviati e ricevuti, fornendo un livello di astrazione superiore che permetta ai processi applicativi di usare la rete sottostante senza conoscerne ne la topologia ne la tecnologia sottostante effettivamente utilizzata. Basti pensare che una data applicazione non interessa generalmente se i dati vengono inviati tramite frequenze radio o via cavo, a lei interessa che i dati arrivino alla destinazione.

Questo servizio non puó essere fornito dai livelli sottostanti perché non prevedono lo stesso livello di affidabilitá.

Gli obiettivi del livello di trasporto sono stati implementati in due protocolli di base: **TCP** e **UDP**. La loro caratteristica comune é quella di eseguire il **multiplexing** *(multiplazione)* e il **demultiplexing** *(demultiplazione)* dei dati inviati e ricevuti dalle applicazioni del livello superiore, ovvero riconoscere a quale applicazione consegnare, o inviare, il flusso dati in ingresso o in uscita, in presenza di una gestione concorrente delle risorse di rete.

A questo livello protocollare la PDU[1] prende il nome di **segmento**, mentre i livelli superiori lo utilizzano tramite la SAP[2] appropriata, chiamata **TSAP** *(Transport Service Access Point)*.

## 1.1 mux/demux

Per eseguire la multiplazione e la demultiplazione dei flussi di dati, il livello di trasporto identifica ogni processo tramite un numero intero senza segno su a 16 bit, chiamato **porta**. L'associazione tra processo e numero di porta viene gestito dal sistema operativo, dove risiede l'implementazione del livello trasporto.

Per convenzione i numeri di porta da `0-1023` vengono chiamate **Well-known Port** *(Porte ben conosciute)*, ovvero numeri riservati per applicazioni standard, mentre i numeri di porta da `1024-65535` sono lasciati per tutte le altre applicazioni.

Ovviamente la porta del client, nel momento di stabilire una connessione, puó essere una porta scelta a caso dal sistema operativo, cosa non vera per la porta di destinazione, che deve essere appropriata per identificare l'applicazione remota.

[1] *PDU*: Package Data Unit
[2] *SAP*: Service Access Point

# 2 ARQ - Automatic Retransmission reQuest

Vengono definiti **ARQ** *(Automatic Retransmission reQuest)* tutti quei protocolli che consentono di richiedere in modo automatico la trasmissione dei segmenti persi o non consegnati.

Ogni ARQ fa uso della tecnica chiamata a **Sliding Windows** *(Finestre a Scorrimento)*, ovvero usano un buffer di PDU inviate e ricevute, e dei segmenti particolari per il riscontro dell'avvenuta ricezione della PDU da parte del destinatario, chiamati **Acknowledgement** *(Riscontro)*; piú formalmente si definisce:

**Finestra di Trasmissione** $W_t$ come la quantitá massima di PDU in sequenza che il trasmettitore é autorizzato ad inviare in rete senza averne ricevuto nessun riscontro.

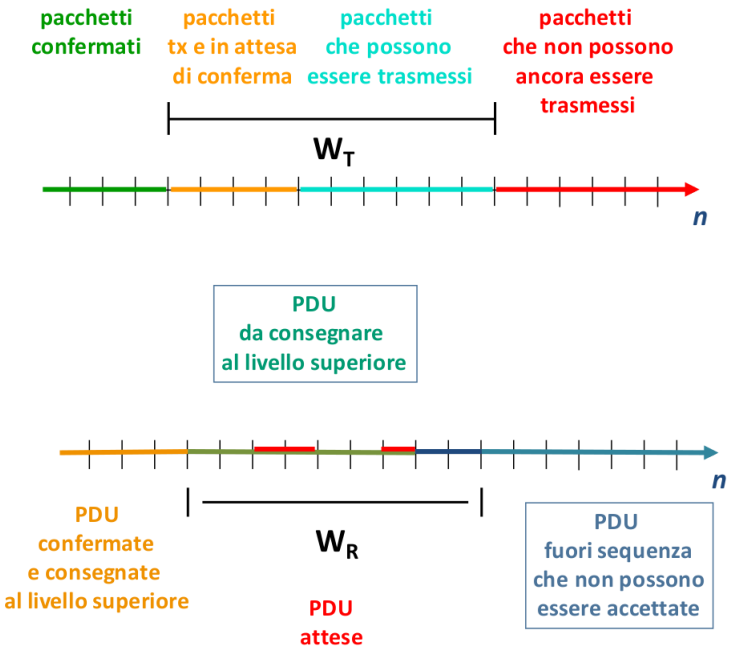
**Finestra di Ricezione** $W_r$ come la quantitá massima di PDU che il ricevitore puó memorizzare dal trasmettitore.

**Acknowledgement** $ACK(n)$ come il riscontro per il segmento con numero di sequenza uguale a $n$.

L'utilizzo delle finestre scorrevoli comporta anche il problema dell'inizializzazione delle stesse ad una dimensione appropriata, risolto da TCP al momento della connessione tramite un'apposita procedura.

L'impostazione di una dimensione consona é cruciale, in quanto: se troppo grande si rischia di sprecare memoria, altrimenti il ricevitore rischia di inviare ACK per ogni pacchetto ricevuto; questo vanifica concettualmente tutti i benefici della tecnica a finestre scorrevoli. La dimensione solitamente adottata é quindi

$$W_r = W_t.$$



Ogni PDU inviata all'interno di una finestra a scorrimento ha un numero di sequenza a $k$ bit. La numerazione é necessaria per tenere traccia dell'ordinamento dei dati in fase di ricezione e serve al destinatario per sapere qual'é il prossimo segmento da ricevere: nel caso in cui il destinatario riceva un segmento fuori dalla sua finestra di ricezione, il segmento verrá scartato.

La numerazione segue la numerazione modulo $2^k$ ovvero viene eseguita ciclicamente su k bit.

Attraverso il sistema di ACK é possibile implementare il controllo di flusso: basta inserire un campo nell'intestazione dei segmenti ACK dove il ricevente specifica la dimensione corrente della sua finestra di ricezione, in unitá di PDU del protocollo in uso; in questo modo il trasmettitore non invierá piú di quello che il ricevente puó gestire.

## 2.1 Semantica dei pacchetti di riscontro

Ogni volta che si instaura una connessione, le due entitá devono accordarsi sul significato semantico del pacchetti di riscontro ACK, che puó essere:

- **ACK individuale o selettivo:** assegna ad ogni pacchetto di riscontro un valore,ovvero il numero di sequenza del segmento riscontrato. Quindi ACK(n) viene inteso come *"ho ricevuto il pacchetto N"*. Questo prevede che il destinatario mantenga traccia del numero di sequenza del successivo frame che si aspetta di ricevere, e spedisce tale numero, ogni volta che manda un segnale ACK
- **ACK cumulativi:** assegna ad ogni pacchetto di riscontro un valore ACK(N) che viene inteso come *"ho ricevuto tutti i pacchetti fino a N escluso"*. In caso di errore peró e' necessario ritrasmettere l'intera finestra in quanto il trasmettitore non ha idea di quale pacchetto sia andato perso
- **ACK negativo o NAK:** assegna ad ogni pacchetto di riscontro un valore NAK(N) che viene inteso come *"ritrasmetti il pacchetto N"*

## 2.2 Piggybacking

La tecnica Piggybacking é utile nel caso in cui vi siano comunicazioni bidirezionali al fine di ridurre i pacchetti che viaggiano sulla rete e, di conseguenza, aumentare le prestazioni del protocollo.

Con questa tecnica gli ACK, non vengono spediti come un segmento singolo, ma vengono inseriti nell'intestazione delle PDU che viaggia nella direzione opposta.

## 2.3 Prestazioni dei Protocolli a Finestre

Si definisce **round trip time** come il doppio del tempo di propagazione:

$$RTT = 2 * T_{prop} = 2 * [s] \tag{1}$$

Si definisce **tempo di trasmissione** come il rapporto tra lunghezza del pacchetto $L$ e la capacitá della rete $R$:

$$\begin{aligned} T_{trasm} &= \frac{L}{R} = \frac{[bit]}{[bit/s]} = [s] \\ T_{trasm} &= \frac{L * 8}{R} = \frac{[byte] * 8}{[bit/s]} = \frac{[bit]}{[bit/s]} = [s] \end{aligned} \tag{2}$$

Si definisce **efficienza** o **utilizzo** come il tempo in cui il mittente é occupato nell'invio di bit ($W_t$ si intende in numero di PDU inviate):

$$U_{mitt} = \frac{W_t * T_{trasm}}{RTT + T_{trasm}} = \frac{[s]}{[s]} \tag{3}$$

Si definisce **throughput** come la capacitá di un algoritmo dell'utilizzo effettivo del mezzo trasmissivo. Nel caso dell'utilizzo di finestre statiche si definisce throughput:

$$\begin{aligned} Thr &= \frac{W_t}{RTT} = \frac{[bit]}{[s]} = \text{ bps} \\ Thr &= \frac{W_t * 8}{RTT} = \frac{[byte] * 8}{[s]} = \frac{[bit]}{[s]} = \text{ bps} \end{aligned} \tag{4}$$

Nel caso invece di finestre di dimensione variabile si ottiene:

$$Thr = \int_{T_{trasm}} W_t(t)dt = \frac{[bit]}{[s]} = \text{ bps} \tag{5}$$

**Esempio:** collegamento a 1 Gbps, pacchetti da 8 kbit, ritardo di propagazione di 15 ms e finestra scorrevole pari a 1 PDU, quindi:

$$\begin{aligned} T_{trasm} &= \frac{8\,\text{kbit}}{10^9\,\text{bps}} = 0.008 \\ Thr &= \frac{8\,\text{kbit}}{0.03\,\text{s}} = 266\,\text{kbps} \\ U_{mitt} &= \frac{0.008}{(15\,\text{s} * 2) + 0.008} = 0.00027 \end{aligned} \tag{6}$$

Nel caso in cui si abbia una finestra scorrevole piú grande, ad esempio $W_r = 3$ si ottiene:

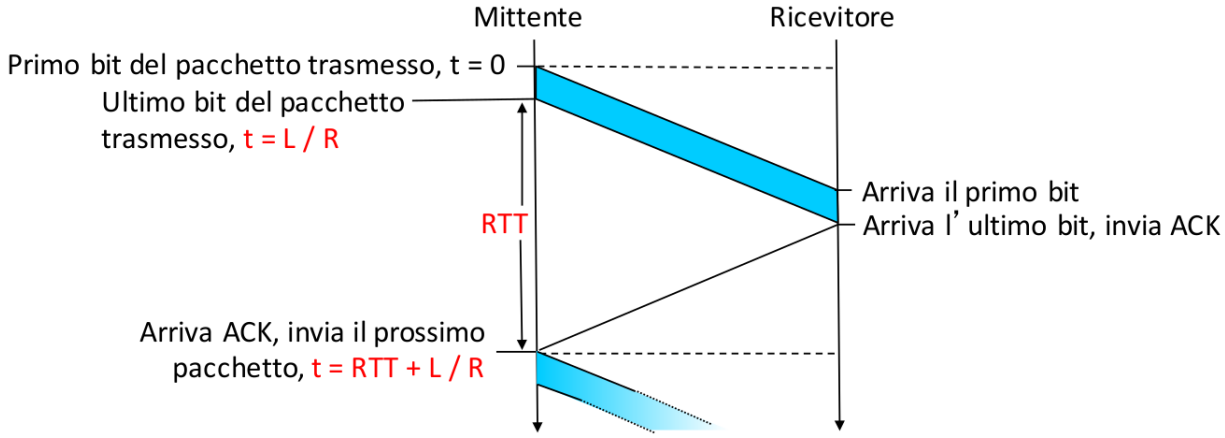
$$U_{mitt} = \frac{3 * 0.008}{(15\,\text{s} * 2) + 0.008} = 0.0008 \tag{7}$$

Aumentando la finestra scorrevole si ottiene un aumento del tempo in cui il mittente é occupato nell'invio dei bit, quindi un aumento dell'efficienza.

## 2.4 Stop and Wait

I protocolli di trasporto a finestra scorrevole di tipo Stop and Wait funzionano nel seguente modo:

- Trasmettitore:
  - invia una PDU[$SEQ$[3] $= X$] e ne salva una copia. $X$ é il numero di SEQ dell'ultimo ACK
  - attiva un timer ad un tempo standard
  - si pone in attesa della ricezione del ACK[$SEQ = X + 1$]
  - se riceve l'ACK[$SEQ = X + 1$], la procedura ricomincia
  - se scade il timer, ritrasmette la PDU
- Ricevitore
  - riceve una PDU
  - calcola il checksum e lo confronta con quello nell'intestazione della PDU; se fallisce scarta la PDU
  - se $SEQ == X$[4] allora é la PDU attesa, altrimenti la scarta
  - invia ACK[$SEQ = X + 1$] e la procedura ricomincia




Il protocollo é poco efficiente perché per ogni PDU inviata dal trasmettitore é richiesto un riscontro con un ACK. La soluzione é l'invio in pipeline delle PDU, ovvero una trasmissione in sequenza.

[3] Il campo *Sequence Number* nell'intestazione di TCP
[4] Il numero di sequenza viene accordato in fase di connessione

## 2.5 Go Back N

Questa particolare istanza di ARQ permette di inviare una serie di segmenti, presenti nella finestra dei trasmettitore, senza averne prima ricevuto un ACK.

Nel caso in cui il destinatario non riceva un segmento, continuerá a rispondere con ACK duplicati. Una volta che il mittente ha esaurito la sua finestra di invio, controlla gli ACK e, se non sono in sequenza, riparte a trasmettere dall'ultimo segmento non pervenuto.

- Trasmettitore:
  - invia $N = W_t$ PDU facendo di ognuna una copia
  - attiva un solo timer per N PDU
  - si pone in attesa della ricezione degli ACK
  - se il timer scade, ritrasmette tutte le PDU nella finestra di cui non ha ricevuto un nessun ACK
  - se riceve un set di ACK con $SEQ = X$, dove X é il piú piccolo numero di sequenza dei segmenti che sono stati trasmessi, allora procede alla ritrasmissione di tutti i segmenti con campo $SEQ \geq X$
  - al contrario, se riceve un ACK per ogni segmento che ha nella finestra, la procedura ricomincia
- Ricevitore
  - riceve una PDU
  - calcola il checksum e lo confronta con quello nell'intestazione della PDU; se fallisce scarta la PDU
  - se la PDU contiene il primo numero di SEQ non ancora ricevuto, allora risponde con un ACK contenente il successivo numero di sequenza atteso, altrimenti scarta l'intera PDU. Ricomincia la procedura

## 2.6 Selective-Repeat

Il protocollo Selective Repeat sopperisce alle mancanze di Go-Back-N in termini di prestazioni. Prevede infatti che il destinatario accetti le PDU fuori sequenza, ma comunque all'interno della finestra di ricezione, e ne invii i rispettivi riscontri.

- Trasmettitore:
  - invia $N = W_t$ PDU facendo di ognuna una copia
  - attiva un timer per ogni PDU
  - quando riceve un ACK relativo ad una PDU la toglie dalla finestra di trasmissione e trasmette una nuova PDU
  - se scade il timer, ritrasmette la PDU relativa, facendo partire un nuovo timer
- Ricevitore
  - riceve una PDU
  - calcola il checksum e lo confronta con quello nell'intestazione della PDU; se fallisce scarta la PDU
  - se il numero di sequenza SEQ della PDU é contenuto all'interno della finestra di ricezione, allora risponde con i relativi riscontri, altrimenti la scarta.

# 3 TCP

TCP *(Transmission Control Protocol)* é stato progettato appositamente per fornire un servizio affidabile su una rete chiaramente inaffidabile. É compito di TCP stabilire una connessione full duplex[5], spedire i dati, riscontrare l'avvenuta ricezione, ritrasmettere i dati in caso di perdita e riordinarli nel caso in cui non arrivino in ordine.

Le entitá TCP scambiano dati sotto forma di **segmenti**, che sono composti da: 20 byte di intestazione *(header)* e da 0 a 1460 byte di dati *(payload)*, chiamata **MSS** *(Maximun Segment Size)*. Quest'ultima dimensione é vincolata per convenzione dalla dimensione massima dei segmenti che possono essere trasmessi su una rete ethernet, ovvero 1500 byte[6]. Ulteriormente esiste un vincolo che riguarda l'incapsulamento dei segmenti nei pacchetti IP, che deve rispettare le dimensioni del payload, chiamato **MTU** *(Maximun Transfer Unit)*. In Pratica ogni MTU definisce il limite superiore per la dimensione di un segmento che puó circolare su una data rete. Puó succedere che delle reti, connesse tra di loro, abbiano MTU differenti, di conseguenza c'é bisogno di scoprire la minima MTU sul tratto di rete. Questa limitazione é quindi una chiara violazione del principio di indipendenza e stratificazione del modello protocollare.

A questo proposito si utilizzano protocolli di **MTU path discovery** attraverso messaggi di controllo **ICMP** *(Internet Control Message Protocol)* i quali testano la rete sottostante con segmenti di dimensione incrementale fino a quando i segmenti vengono persi.

TCP implementa anche un meccanismo di controllo degli errori sull'intero segmento (piú 8 byte dell'header del livello sottostante, contenenti l'indirizzo sorgente e destinazione a livello IP) tramite il calcolo di un *checksum* a 16 bit. Il mittente, una volta formato il segmento, aggiunge all'intestazione il checksum relativo nell'apposito campo e procede all'invio; il destinatario invece, calcolerá il checksum nello stesso modo e lo confronterá con quello contenuto nell'intestazione del segmento appena ricevuto. Se il confronto porta delle differenze tra i codici di controllo TCP scarta completamente il segmento e ne chiede la ritrasmissione.

## 3.1 TCP in breve

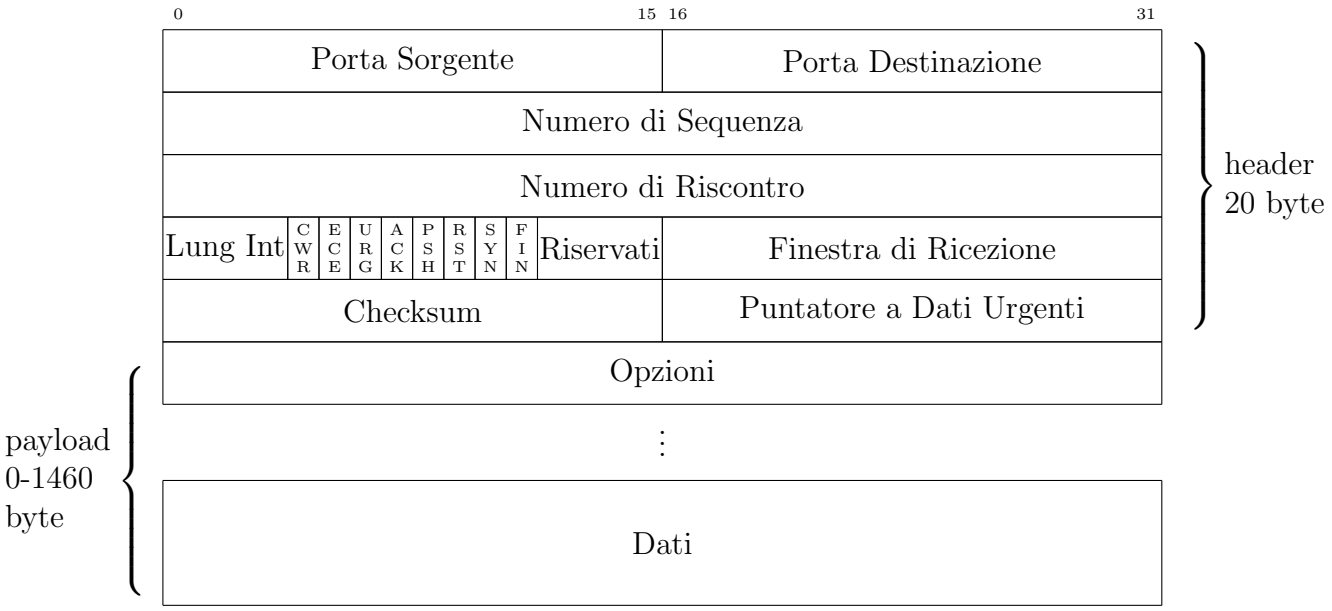
- mantiene lo stato della connessione: si occupa dell'instaurazione e del rilascio
- recupero degli errori sul segmento
- consegna ordinata dei segmenti all'applicazione
- controllo di flusso: controlli per evitare che il *destinatario* non riesca a immagazzinare tutti i dati che sto trasmettendo
- controllo di congestione: controlli per evitare che la *rete* non riesca a consegnare tutti i dati che sto trasmettendo

[5] full duplex: vengono aperte due "mezze" connessioni in entrambe le direzioni

[6] 1500 byte trama ethernet = 20 header IP + 20 byte header TCP + 1460 byte payload (default MSS)

## 3.2 Formato del Segmento

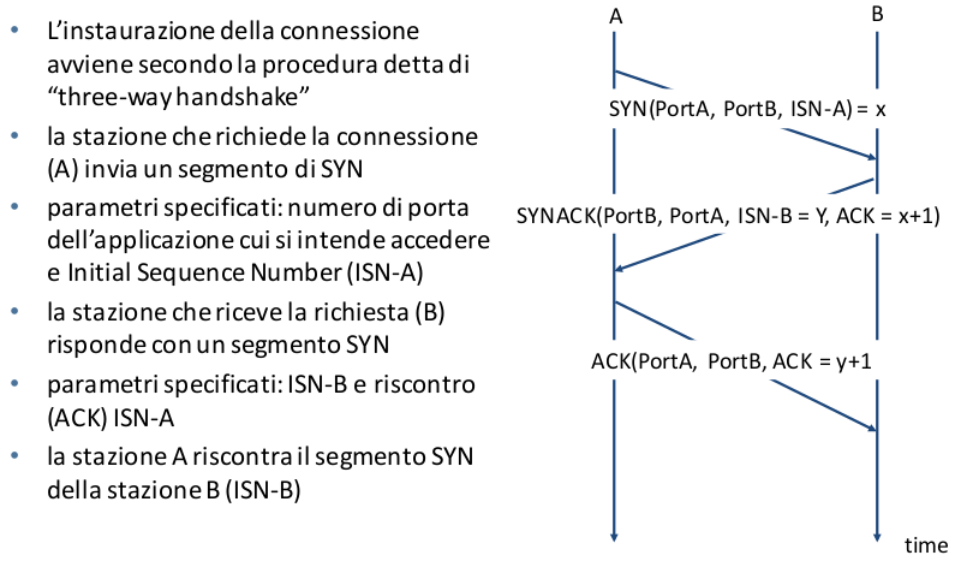
Il segmento TCP é cosí composto:




- **Porta Sorgente e Destinazione**: numero di porta sorgente e destinazione.
- **Numero di Sequenza**: numero di sequenza a numerazione ciclica su 32 bit.
- **Numero di Riscontro**: numero usato per il riscontro del segmento, ha significato se il bit `ACK` é impostato.
- **Lunghezza dell'Intestazione**: numero che rappresenta quanti gruppi di word a 32 bit sono presenti nell'header, in quanto sono presenti opzioni facoltative.
- Serie di bit:
  - `CWR`: *Congestion Windows Reduce*, se impostato indica di ridurre la congestione
  - `ECE`: *Explicit Congestion Notification*, se impostato indica che l'host supporta questa funzione durante il three-way handshake
  - `URG`: *Urgent Pointer*, se impostato abilita il campo Urgent Pointer dell'header
  - `ACK`: *Acknowledgement*, se impostato abilita il campo Acknowledgement dell'header
  - `PSH`: *Push*, se impostato indica di passare i dati direttamente all'applicazione senza usare buffer di lettura. Usato principalmente nelle applicazioni di terminali remoti
  - `RST`: *Reset*, usato principalmente per rilasciare la connessione in modo netto, nel caso in cui il peer non risponda piú
  - `SYN`: *Synchronize*, flag utilizzato per sincronizzare i numeri di sequenza durante l'apertura di una connessione.
  - `FIN`: *Fine*, usato per la chiusura della connessione
- **Finestra di Ricezione**: indica la dimensione della finestra di ricezione del mittente.
- **Checksum**: campo per il controllo degli errori.
- **Dati Urgenti**: rappresenta un offset sui dati del segmento ed indica che quel set di dati deve essere consegnato all'applicazione non in ordine con gli alti.
- **Opzioni**: opzioni aggiuntive in word da 32 bit.

## 3.3 Instaurazione di una Connessione

L'instaurazione della connessione tra entitá TCP avviene tramite un processo chiamato **Three-way Handshake** *(Stretta di mano a tre)*.

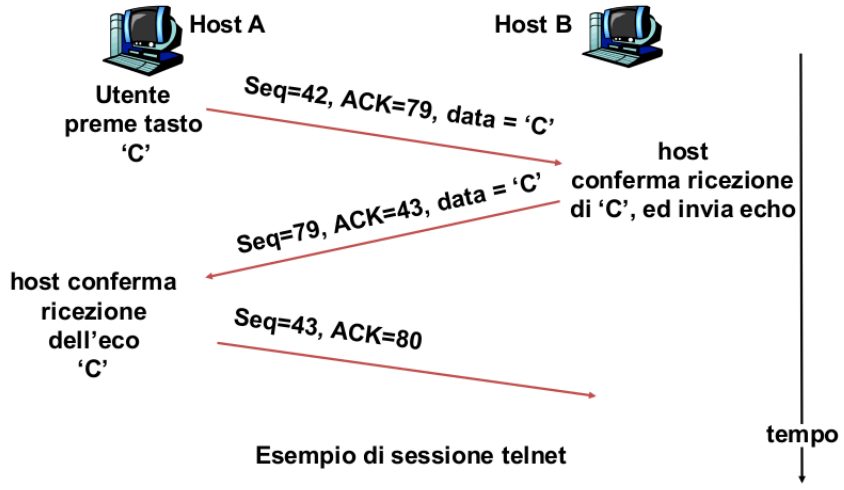



Il protocollo di connessione Three-way Handshake prevede lo scambio di segmenti chiamati: `SYN`, `SYNACK` e `ACK`, ovvero normali segmenti TCP con i bit `SYN` e `ACK` attivi.

Con questa procedura é possibile che si verifichino delle situazioni indesiderate, dove o il mittente o il destinatario ricevono segmenti duplicati.

Nel caso in cui l'host A riceva un segmento con i bit `SYNACK` ma, senza aver inviato nessun segmento `SYN` verso l'host mittente B, allora A provvederá ad inviare un segmento con il flag `RST` impostato per terminare la connessione. Lo stesso comportamento si verifica con la ricezione di segmenti `ACK` senza prima aver inviato segmenti con numero di sequenza $SEQ = ACK(SEQ - 1)$, ovvero senza aver inviato prima il relativo segmento.

Nell'ultimo caso, invece, se le due entitá tentano di creare una connessione simultaneamente, ne viene creata solo una.



## 3.4 Chiusura di una Connessione

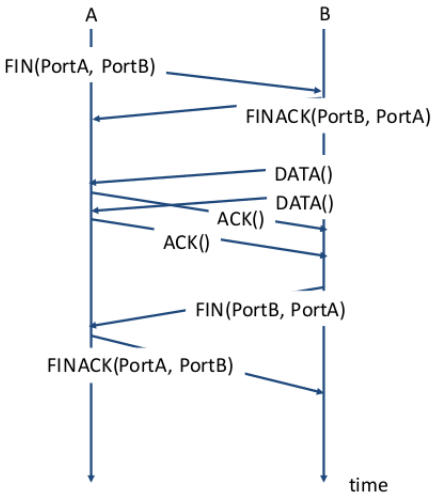
Per rilasciare una connessione entrambe le entitá devono inviare e riscontrare un segmento TCP con il flag di `FIN` impostato, anche chiamato segmento **DR** *(Disconnection Request)*.

Al momento della ricezione del relativo `ACK` la direzione viene chiusa per i nuovi dati, ma possono comunque arrivare dati nella direzione opposta. Con questo meccanismo le due parti coinvolte dovranno inviare segmenti `DR` e attendere l'`ACK` per terminare la propria parte di connessione.

In altre parole: quando un host riceve un segmento `DR` sá che la connessione nel lato opposto é terminata (non riceverá piú dati in quella direzione), ma potrá comunque finire di trasmettere i propri dati e terminarli con l'invio di un segmento `DR` a sua volta

Nel caso un cui venga perso il segmento `DR`, l'host in attesa del riscontro, andrá in timeout e procederá con il ri-invio del segmento.

Nel grafico seguente sono riportati i segmenti `FIN` (corrispondenti ai `DR`) e i `FINACK`, che sono i riscontri dei segmenti `FIN`.




É possibile terminare la connessione anche impostando il flag `RST`. In questo modo la chiusura avviene unilateralmente, senza necessitá di conferme da nessun lato. É molto utilizzata dai server che hanno bisogno di liberare le risorse usate dalla connessione in modo rapido, ad esempio per prevenire attacchi di *Denial of Service*: l'entitá attaccante, dopo aver creato una connessione col server, non risponderá mai con i riscontri ai segmenti `DR`, quindi il server non riusciá mai a chiudere il suo lato di connessione e, di conseguenza, non libererá mai le risorse a lei assegnate.

## 3.5 Gestione dei Timer - Timeout

In TCP sono implementati molti meccanismi che permetto la ritrasmissione dei segmenti nel caso non vengano consegnati; ma l'ultima tecnica che possiede (ed é l'unica che funziona sempre) é quella di utilizzare un timer chiamato **RTO** *(Retransmission Time Out)* che, alla scadenza, ri-invia il segmento perso.

Il RTO indica, quindi, il tempo entro il quale la sorgente si aspetta di ricevere un ACK. Questo tempo non puó essere impostato staticamente, per via della differenza delle reti fisiche su cui TCP deve basarsi, ma deve tenere conto di molti fattori come: la distanza tra sorgente e destinazione, dalla condizione della rete, dalla disponibilitá dell'host ecc.

Quindi l'unico modo é quello di stimare il RTO misurando il tempo trascorso tra l'invio di un segmento e il suo riscontro; questo tempo viene chiamato **RTT** *(Round Trip Time)*.

Il RTT varia da pacchetto a pacchetto, quindi adattare staticamente il RTO a questi cambiamenti improvvisi non é la scelta migliore. A questo scopo si utilizza una media mobile esponenziale chiamata **SRTT** *(Smoothed RTT)*, calcolata e tenuta costantemente aggiornata nel seguente modo:

$$SRTT = (1 - \alpha) \cdot SRTT + \alpha \cdot 1RTT \tag{8}$$

dove $\alpha$ é tipicamente impostata a $1/8$ [7].

In generale, il problema di usare una media come "rappresentante" di un insieme di valori é che si perde l'informazione di quanto i dati si discostano tra di loro, quindi in questo caso la media non segue fedelmente il RTO.

A questo proposito si utilizza la varianza della madia, chiamata **RTTVAR** *(Varianza della media SRTT)* calcolata nel seguente modo:

$$RTTVAR = (1 - \beta) \cdot RTTVAR + \beta \cdot \mid SRTT - RTT \mid \tag{9}$$

dove $\beta$ é tipicamente impostata a $1/4$ [7].

Conoscendo lo SRTT e l'RTTVAR si puó impostare correttamente il RTO come segue:

$$RTO = SRTT + 4 \cdot RTTVAR \tag{10}$$

Da notare che é data molta importanza alla variabilitá della media. Esiste comunque un valore sotto il quale non é possibile scendere, ovvero $\approx 200ms$

[7] Questa scelta é dovuta ad una serie di test che hanno riscontrato che é un valore appropriato.

## 3.6 Congestione

Uno dei compiti del livello di Trasporto é quello di rilevare e reagire alle congestioni che si verificano sulla rete. Le cause di una congestione possono essere molteplici: la velocitá di elaborazione dei segmenti di alcuni apparati di rete, oppure un eccessivo traffico generato dal trasmettitore che satura la capacitá della rete.

L'effetto principale é la perdita di segmenti lungo la rete, che quindi, ne causa la ritrasmissione da parte dei terminali, innescando un circolo vizioso che, se non gestito, porta inevitabilmente la rete a consegnare solo informazioni ritrasmesse, quindi al collasso[8].

Questa situazione vista in termini di throughput é la seguente: in accordo con la sua definizione (equazione 4) all'aumento del RTT segue un calo del throughput, quindi TCP deve trovare il modo di adattare $W_t$ dinamicamente in modo da non ridurlo a zero (nel caso in cui scade il RTO).

L'implementazione adottata é quella di creare una finestra logica che condiziona la dimensione di $W_t$ in funzione dello stato attuale della rete, tale meccanismo prende il nome di **CWND** *(Congestion Window)*.

La CWND indica, quindi, il numero di segmenti che il mittente puó inviare senza creare congestione sulla rete; nel caso un cui ci siano comunque delle perdite di segmenti, TCP interviene variandone la dimensione e, di conseguenza, varia anche la finestra di trasmissione che riduce il tasso di invio dei segmenti.

In TCP la finestra di congestione viene gestita tramite due algoritmi che ne garantiscono una crescita controllata nel caso non ci siano perdite, **Slow Start** e **Congestion Avoidance**, mentre nel caso contrario sono stati ideate tecniche che permettono di recuperare selettivamente i segmenti senza creare congestione e lasciando il meccanismo di RTO come ultima risorsa, si tratta di: **Fast Retransmit** e **Fast Recovery**.

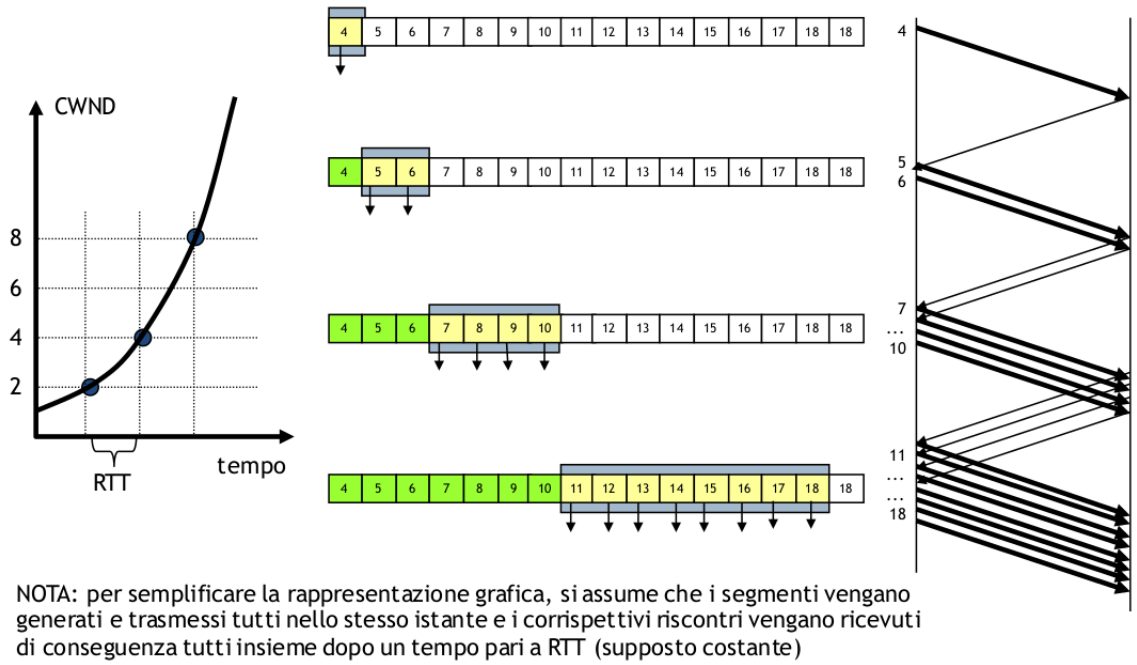
Il recupero allo scadere del RTO viene lasciato come ultima risorsa, da usare solo nel caso in cui le altre tecniche hanno fallito. Il motivo principale di questa scelta é dettata dal fatto che la ritrasmissione con il RTO é sempre attuabile e perché riduce drasticamente le prestazioni del trasferimento.

All'inizio della connessione TCP imposta la CWND ad un valore molto basso, tipicamente ad 1 MSS, poi l'aumenta secondo le tecniche di Slow Start, fino ad una certa soglia chiamata **SSTRESH** *(Slow Start Threshold)*, poi continua con Congestion Avoidance, fino ad un massimo che é dato dal mezzo trasmissivo o dalla RWND *(Receiver Window)* del destinatario.

[8]Questo é accaduto molte volte ai tempi di ARPANET, richiedendo che l'intera rete venisse spenta e progressivamente riaccesa.

### 3.6.1 Slow Start

Di seguito viene riportato l'immagine dell'andamento di crescita della CWND con l'algoritmo di Slow Start: i segmenti verdi sono stati inviati e riscontrati, quelli gialli sono stati inviati ma non ancora riscontrati, quelli bianchi invece non sono stati ancora inviati, la finestra di congestione e' raffigurata in blu.



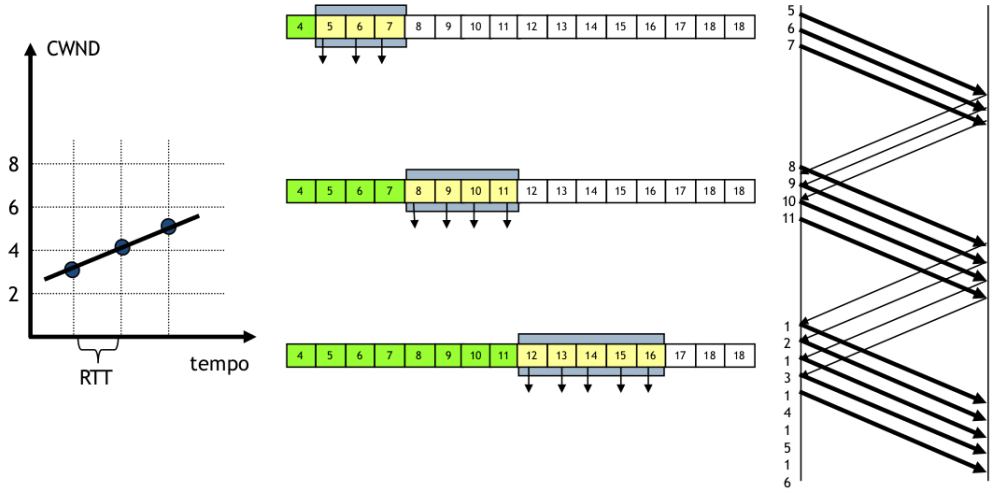

Slow Start inizia con la CWND inizializzata ad 1 MSS. Ad ogni ACK (non duplicato) ricevuto (ovvero ad ogni RTT) incrementa la finestra di congestione sempre di 1 MSS; questo implica la trasmissione sempre di due nuovi segmenti per ACK ricevuto. In altri termini, ad ogni RTT la CWND viene aggiornata nel seguente modo:

$$CWND = CWND + MSS \tag{11}$$

Questa peculiare caratteristica rende l'evoluzione della dimensione *esponenziale*.

L'inevitabile conseguenza di questo andamento é che arriva a saturare il mezzo trasmissivo o la destinazione in poco tempo; quindi Slow Start arriva fino alla SSTHRESH dove l'incremento passa a lineare.

### 3.6.2 Congestion Avoidance




L'algoritmo Congestion Avoidance subentra dopo la soglia SSTRESH, per evitare tutti i problemi di Slow Start e il suo andamento esponenziale.

Congestion Avoidance, ad ogni ACK (non duplicato) ricevuto aumenta la finestra di congestione di un fattore, quindi assume un andamento *lineare* ad ogni RTT; ovvero:

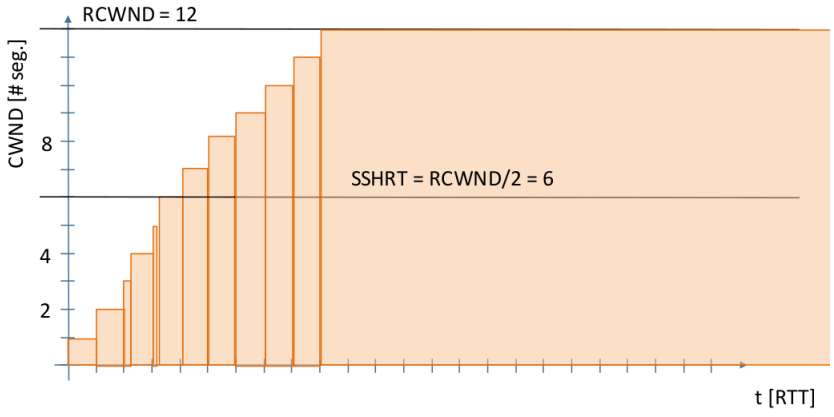
$$CWND = CWND + MSS/CWND \tag{12}$$

In altri termini: deve ricevere un numero di ACK non duplicati pari ai segmenti presenti nella finestra di congestione per aumentarla di 1 MSS.

### 3.6.3 Riassumendo

TCP utilizza il seguente algoritmo per evitare la congestione:

- inizio della trasmissione, si pone
  - CWND = 1 segmento, ovvero 1 MSS espresso in byte
  - SSTRESH = RWND/2
- la CWND evolve secondo l'algoritmo **Slow Start** fino a SSTRESH
- superata la SSTHRESH continua con **Congestion Avoidance**
- la CWND cresce fino al raggiungimento della RWND



# 4 Recupero delle Perdite

Durante il normale trasferimento dati é possibile che alcuni segmenti vengano scartati dagli apparecchi di rete, come router, firewall e gateway, oppure dagli stessi host, per molteplici motivi, il piú frequente é l'accumulo nei buffer del ricevitore in quanto non riesce a svuotarlo a consegnarli all'applicazione abbastanza velocemente per riceverne dei nuovi.

TCP, una volta rilevato le perdite, utilizza varie tecniche per il recupero di dati; possono variare dall'utilizzo di del RTO, oppure da algoritmi piú complessi come Fast Retransmit e/o Fast Recovery che tengono conto dello stato della congestione della rete oppure dell'host di destinazione.

Questi algoritmi implementano la specifiche ARQ descritte dei paragrafi precedenti.

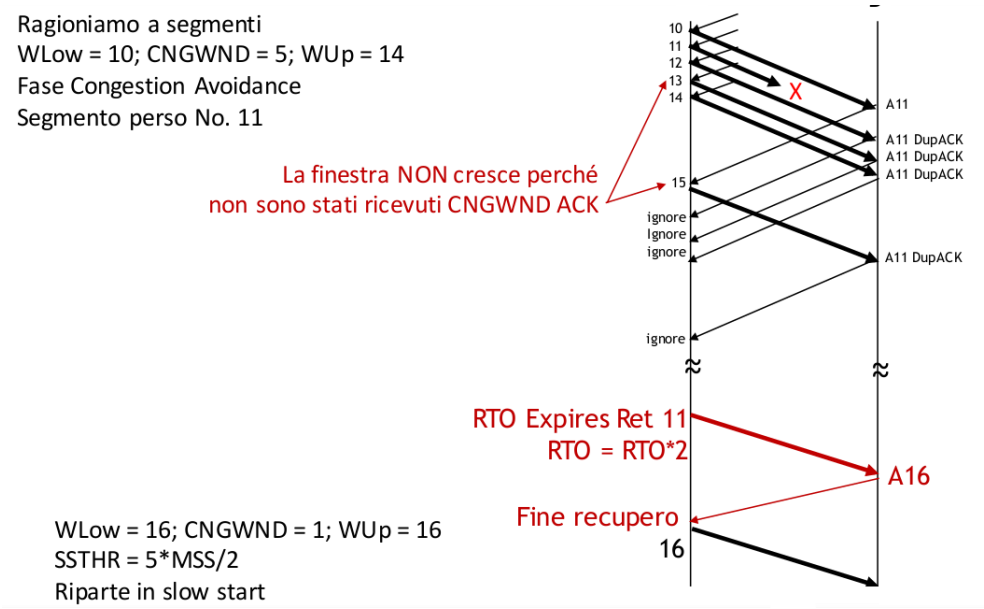
## 4.1 Recupero con RTO

Il meccanismo piú semplice per il recupero di un segmento non riscontrato é quello di lasciar scadere un timer, il RTO, e poi procedere alla ritrasmissione. Ad ogni scadenza il RTO viene raddoppiato, lasciando la possibilitá che la congestione si risolva da sola: si pensi ad un host particolarmente congestionato che deve svuotare i propri buffer. La tecnica di raddoppiare il RTO ogni volta che scade viene chiamata *Back-off Esponenziale*. Se scade il RTO per 16 volte, TCP dichiara che non é piú connesso alla rete.

Andando in dettaglio, alla scadenza di un RTO avvengono le seguenti azioni:

1. $RTO = RTO \cdot 2$: raddoppio del valore del precedente RTO, per le ragioni spiegate sopra
2. $SSTRESH = CWND/2$: viene impostata la soglia di Slow Start a metá della finestra di congestione
3. $CWND = MSS$: viene impostata la finestra di congestione alla dimensione di un segmento
4. Ritrasmette il primo segmento della finestra: per costruzione, il primo segmento della finestra di congestione é il segmento non riscontrato
5. Attende il riscontro: in questo caso puó ricevere i seguenti tipi di ACK:
   - se $ACK(SEQ) = W_{low}$ significa che é stato perso ancora il segmento appena ritrasmesso, quindi la procedura riparte dal punto 4
   - se $W_{low} < ACK(SEQ) < W_{up}$ allora significa che ha riscontrato il segmento perso ed ha richiesto la ritrasmissione di un altro segmento della finestra anch'esso perso; quindi viene spostato il bordo inferiore della finestra di trasmissione al numero di SEQ ricevuto: $W_{low} = ACK(SEQ)$
   - se $ACK(SEQ) > W_{up}$ allora la ritrasmissione é andata a buon fine, tutti i segmenti persi sono stati ricevuti e riscontrati quindi puó riprendere la normale trasmissione
6. Se scade il RTO: ricomincia la procedura dal punto 1 per 10 volte, poi altre 6 senza aumentare il RTO, poi si arrende.

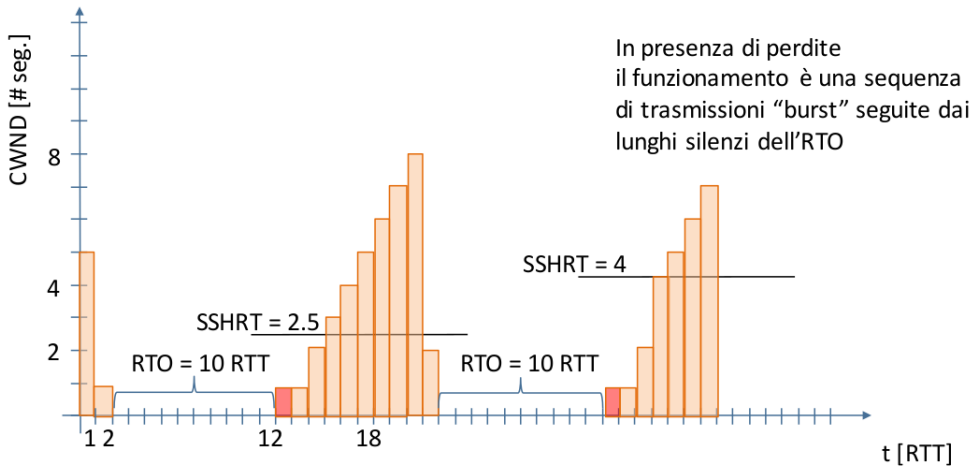
Da notare che in questo caso si utilizzano ACK comulativi come semantica dei riscontri.




Come si nota dall'esempio, alla perdita del segmento 11, il trasmettitore riceverá una serie di ACK duplicati con $SEQ = 11$ che vengono ignorati; allo scadere del RTO ritrasmetterá il segmento 11 ricevendo come $ACK(SEQ) = 16$, ovvero il prossimo numero di sequenza che il ricevitore si aspetta di ricevere.

Di seguito vengono riportati i grafici dell'invio effettivo dei segmenti ad ogni RTT. Il segmento in rosso é dove avviene la perdita.

Si supponga che: $RTT = 20ms, RTO = 200ms, RCWND = 40$ segmenti:




Come si nota dal grafico, con il solo utilizzo del RTO, non é affatto efficiente, in quanto c'é bisogno di 18 RTT prima che la CWND ritorni ad avere un numero di segmenti pari a prima della perdita e, nel caso di perdite multiple, si notano molti momenti di silenzio, dovuti al RTO, e a momenti di forte trasmissione a *"burst"*.

Ovviamente questo andamento non é ottimale e non usa al meglio il mezzo trasmissivo.

## 4.2 Recupero con Fast Retransmit

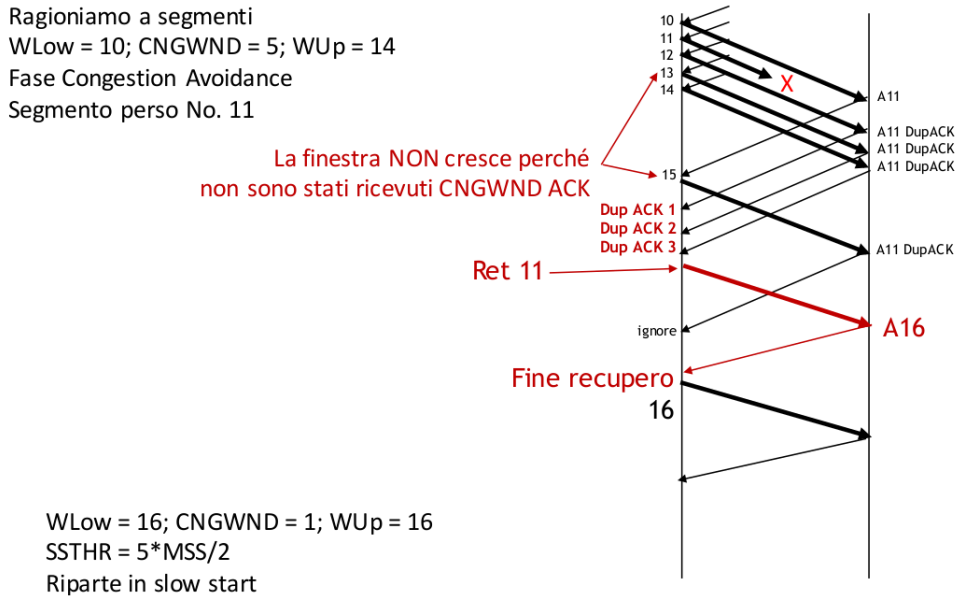
Il secondo algoritmo usato da TCP é chiamato Fast Retransmit che, a differenza del precedente, utilizza la seguente l'euristica: se si ricevono piú di tre ACK duplicati, ovvero con numero di SEQ uguali, allora qualche segmento é andato perso e si puó procedere alla ritrasmissione senza che scada nessuno timer. In effetti questa semplice idea migliora notevolmente l'utilizzo del mezzo trasmissivo e le prestazioni del trasferimento rispetto all'utilizzo puro del RTO.

La tecnica di recupero tramite RTO peró non é mai abbandonato del tutto, in quanto viene sempre inizializzato e, alla scadenza, si comporta esattamente come spiegato precedentemente.

L'algoritmo si comporta nel seguente modo quando riceve 3 ACK con lo stesso numero di sequenza:

1. $SSTRESH = CWND/2$: dimezza il valore della soglia di Slow Start
2. $CWND = MSS$: viene impostata la finestra di congestione alla dimensione di un segmento
3. Ritrasmette il primo segmento della finestra di congestione: per costruzione, il primo segmento é il segmento non riscontrato
4. Attende il riscontro: in questo caso puó ricevere i seguenti tipi di ACK:
   - se $ACK(SEQ) = W_{low}$ significa che é stato ricevuto ancora un ACK duplicato, quindi lo scarta
   - se $W_{low} < ACK(SEQ) < W_{up}$ allora significa che ha riscontrato il segmento perso ed ha richiesto la ritrasmissione di un altro segmento della finestra, anch'esso perso; quindi viene spostato il bordo inferiore della finestra di trasmissione al numero di SEQ ricevuto: $W_{low} = ACK(SEQ)$ e $W_{up} = W_{low} + CWND$
   - se $ACK(SEQ) > W_{up}$ allora la ritrasmissione é andata a buon fine, tutti i segmenti persi sono stati ricevuti e riscontrati quindi puó riprendere la normale trasmissione
5. se scade il RTO si comporta come un recupero con puro RTO

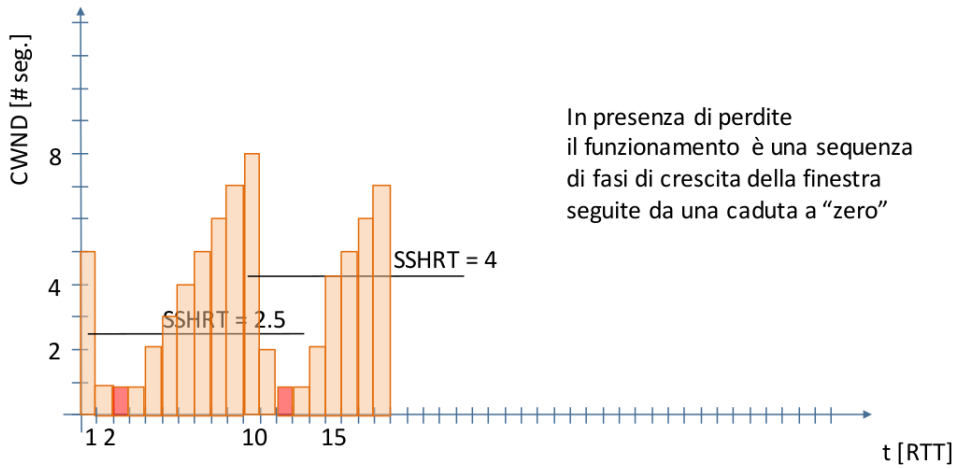
Nel momento in cui Fast Retransmit recupera l'informazione persa, la trasmissione riprende in Slow Start, con i parametri impostati nel corso dell'algoritmo (SSTRESH al punto 1); limitando molto le prestazioni dell'algoritmo. Questi semplici meccanismi si implementando di fatto la ritrasmissione selettiva dell'informazione persa.




Nell'esempio riportato si nota come il segmento con $SEQ = 11$ venga perso, e alla ricezione dei 3 ACK duplicati, si provveda alla sua riconsegna; ignorando l'ACK duplicato dopo la ritrasmissione, dovuto al segmento con $SEQ = 15$.

Importante e sostanziale differenza rispetto ad RTO é che la ritrasmissione del segmento avviene dopo un solo RTT: si noti che, nel grafico sottostante, tra la perdita del segmento (barra rossa) e la ritrasmissione, non é passato nessun RTT *"vuoto"*, migliorando le prestazioni.

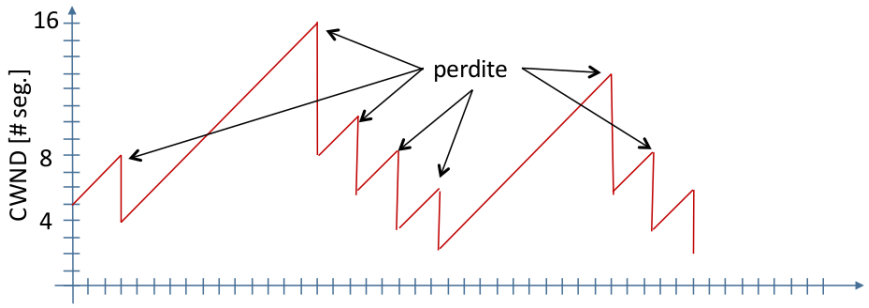
Si supponga che: $RTT = 20ms, RTO = 200ms, RCWND = 40$ segmenti:




Come si nota dal grafico, la finestra di congestione ritorna al suo valore precedente alla perdita solo dopo 8 RTT, rispetto a 18 con RTO puro, con conseguente miglioramento del throughput.

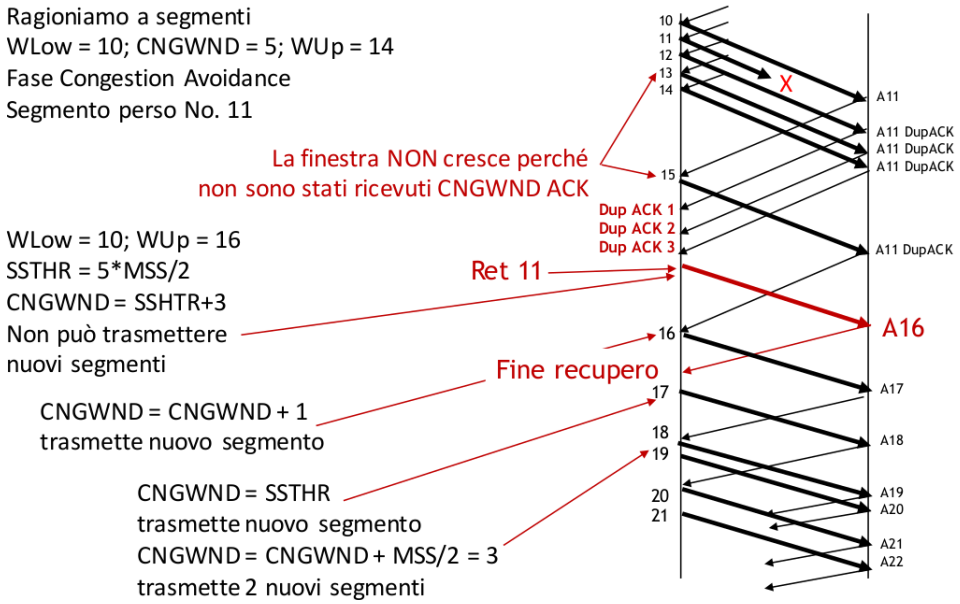
## 4.3 Recupero con Fast Recovery

L'idea di base per ottimizzare Fast Recovery é quella di non ricominciare da capo ogni volta, ma di impostare la $CWND = SSTRESH$ e ricominciare in Congestion Avoidance; in questo modo si cerca di ottenere un oscillazione a *"dente di sega"* dimezzandola in caso di perdite e incrementandola linearmente altrimenti.




L'algoritmo usato da Fast Recovery é uguale a Fast Retransmit per le modalitá in cui viene attivato (dopo 3 ACK duplicati) ma é piú efficiente perché consente di continuare l'invio di segmenti mentre stá ancora recuperando i segmenti persi:

1. $SSTRESH = CWND/2$: imposta la soglia di Slow Start a metá finestra di congestione
2. Ritrasmette il primo segmento della finestra di congestione: per costruzione, il primo segmento é il segmento non riscontrato
3. $CWND = SSTRESH + 3 \cdot MSS$: imposta la finestra di congestione al valore della soglia di Slow Start aggiungendo 3 segmenti, che sono i 3 ACK duplicati ricevuti all'inizio della perdita. Questo é dovuto dal fatto che il ricevitore ha ricevuto altri pacchetti (non quello che si aspettava) e gli ha riscontrati con ACK duplicati
4. Attende il riscontro: in questo caso puó ricevere i seguenti tipi di ACK:
   - se $ACK(SEQ) = W_{low}$ significa che é stato ricevuto ancora un ACK duplicato, quindi $CWND = SSTRESH + MSS$, ovvero aumenta la finestra di congestione di un segmento, per i motivi del punto 3
   - se $W_{low} < ACK(SEQ) < W_{up}$ allora significa che ha riscontrato il segmento perso ed ha richiesto la ritrasmissione di un altro segmento della finestra, anch'esso perso; quindi viene spostato il bordo inferiore della finestra di trasmissione al numero di SEQ ricevuto: $W_{low} = ACK(SEQ)$ e $W_{up} = W_{low} + CWND$
   - se $ACK(SEQ) > W_{up}$ allora la ritrasmissione é andata a buon fine, tutti i segmenti persi sono stati ricevuti e riscontrati quindi $CWND = SSTRESH$ e si puó riprendere in Congestion Avoidance
5. se scade il RTO si comporta come un recupero con puro RTO

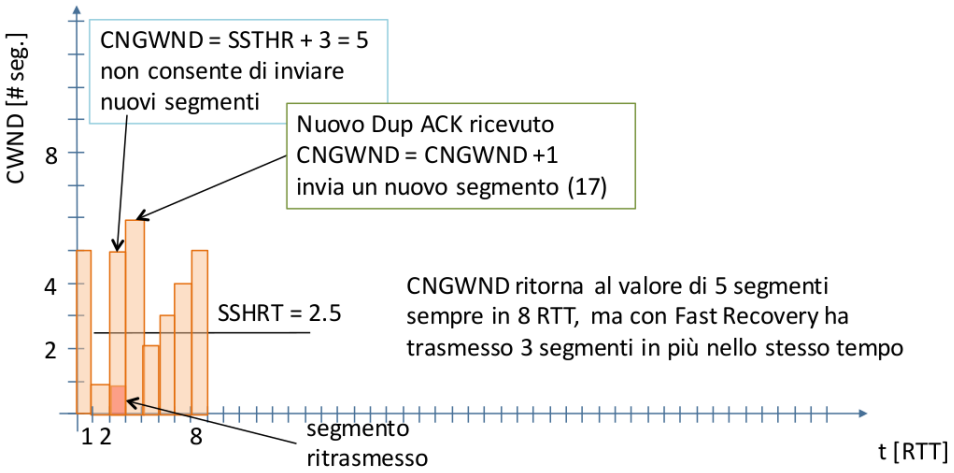



Come si vede dall'esempio, dopo la ricezione di 3 ACK duplicati si procede alla ritrasmissione del segmento perso, numero 11, ma quando ri riceve per la quarta volta ancora un riscontro duplicato, si fa crescere la finestra di congestione e si cerca di continuare a trasmettere segmenti (trasmissione del segmento numero 16).

Una volta ricevuto un ACK non duplicato che non é contenuto nell'attuale finestra di trasmissione, il che significa che ha riscontrato tutti i segmenti precedenti, si continua in Congestion Avoidance, incrementando linearmente la dimensione della finestra.

Di seguito vengono riportati i grafici dell'invio effettivo dei segmenti ad ogni RTT. Il segmento in rossoé dove avviene la perdita.

Si supponga che: $RTT = 20ms, RTO = 200ms, RCWND = 40$ segmenti:




Come si nota dal grafico soprastante, la situazione migliora molto rispetto a Fast Retransmit, in quanto in 8 RTT per recuperare la perdita si riesce ad inviare 3 nuovi segmenti. L'unico problema di Fast Recovery é quello della scadenza del RTO; infatti se la rete, o il ricevitore, sono particolarmente congestionate, il RTO continuerá a scadere, degradando le prestazioni.

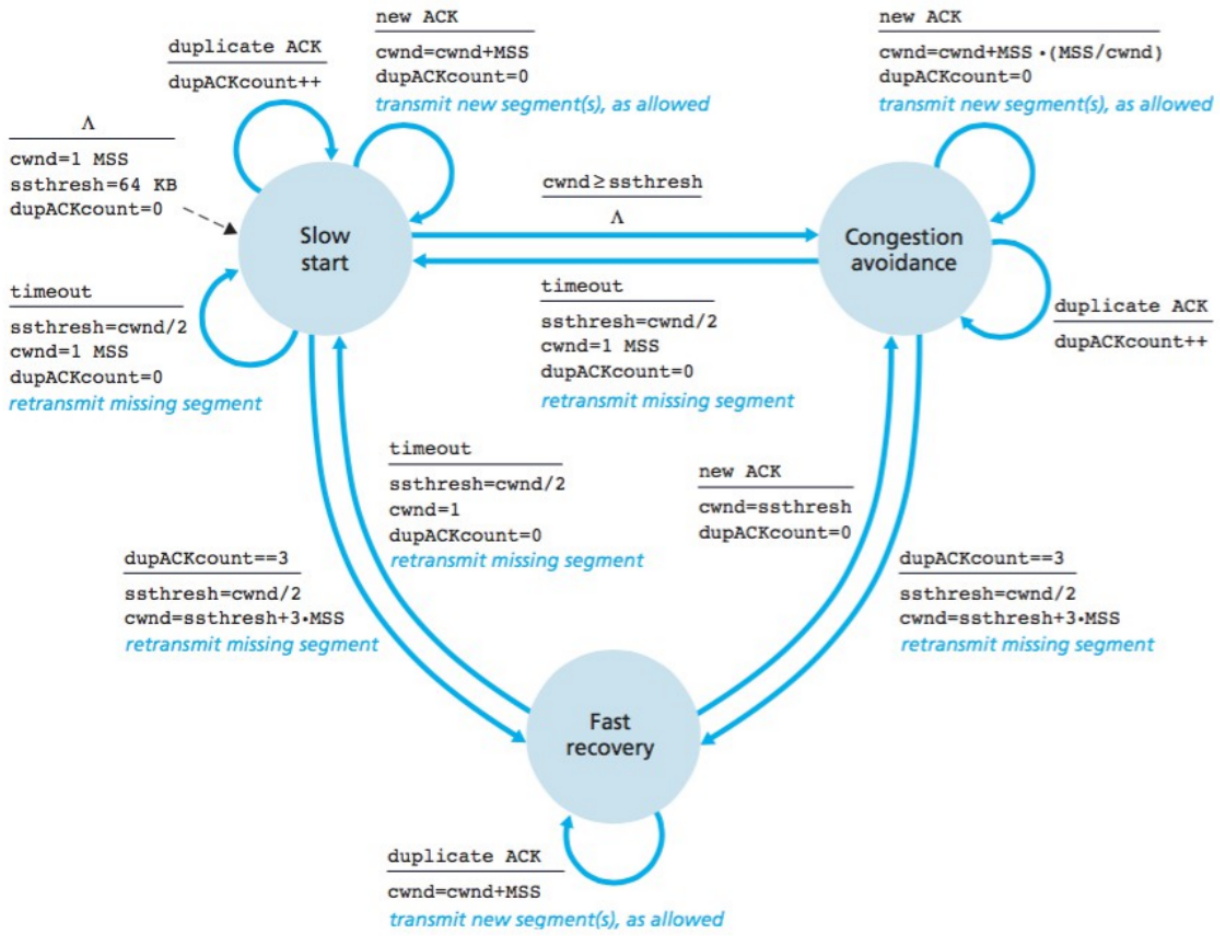
## 4.4 Riassumendo

Riassumendo i meccanismi di recupero delle perdite:

- **RTO:** recupera un segmento alla volta. Funziona sempre ma spreca moltissime risorse
- **Fast Retransmit:** abbatte il tempo di recupero, consente di recuperare un segmento per RTT, anziché per RTO, ma é ancora inefficiente
- **Fast Recovery:** é il piú efficiente ma, nel caso di perdite multiple, ricade nel caso di RTO

## 4.5 Diagramma a Stati

Di seguito viene rappresentato il diagramma a stati di TCP in tutte le fasi della trasmissione dati:



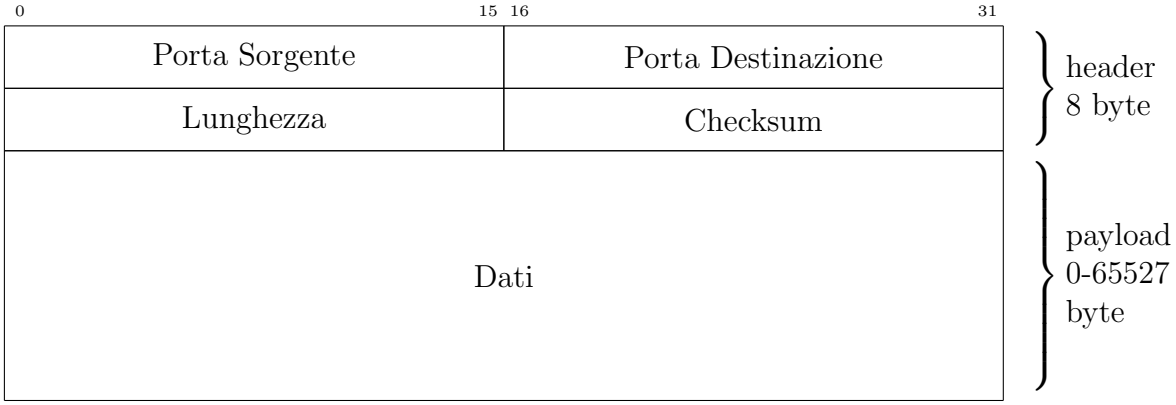
# 5 UPD

UDP *(User Datagram Protocol)* é un protocollo leggero e veloce, in quanto fornisce un servizio di solo invio di segmenti senza conferma di ricezione e senza bisogno di instaurare una connessione tra le entitá coinvolte; per questo viene definito *non connesso-non confermato.*

Questo porta alcuni svantaggi, ad esempio non gestisce il controllo di flusso o la ritrasmissione dei segmenti corrotti o mai arrivati, ma demanda alle applicazioni che lo utilizzano l'implementazione di tali meccanismi.

D'altro canto peró é l'unico protocollo che permette a tutte le applicazioni in real-time di funzionare: basti pensare alla comunicazione audio/video, in questo caso non e' importante la consegna in sequenza e completa dei segmenti, in quanto si tende ad accettare una certa tolleranza di perdita a fronte di una comunicazione il piú possibile fluida e in tempo reale.

Il segmento UDP é composto:




- **Porta Sorgente e Destinazione**: numero di porta sorgente e destinazione
- **Lunghezza**: valore in byte che identifica la lunghezza dell'intero segmento
- **Checksum**: campo per il controllo degli errori per l'intestazione e i dati
- **Dati**: i dati che trasporta il segmento

**Perché non usare direttamente il livello di Rete sottostante ?** Non é possibile in quanto non gestisce la comunicazione end-to-end, infatti nell'intestazione del pacchetto IP non é presente la porta sorgente e destinazione.