Reti di Comunicazioni e Internet

Protocolli di trasporto (TCP)

Massimo Tornatore

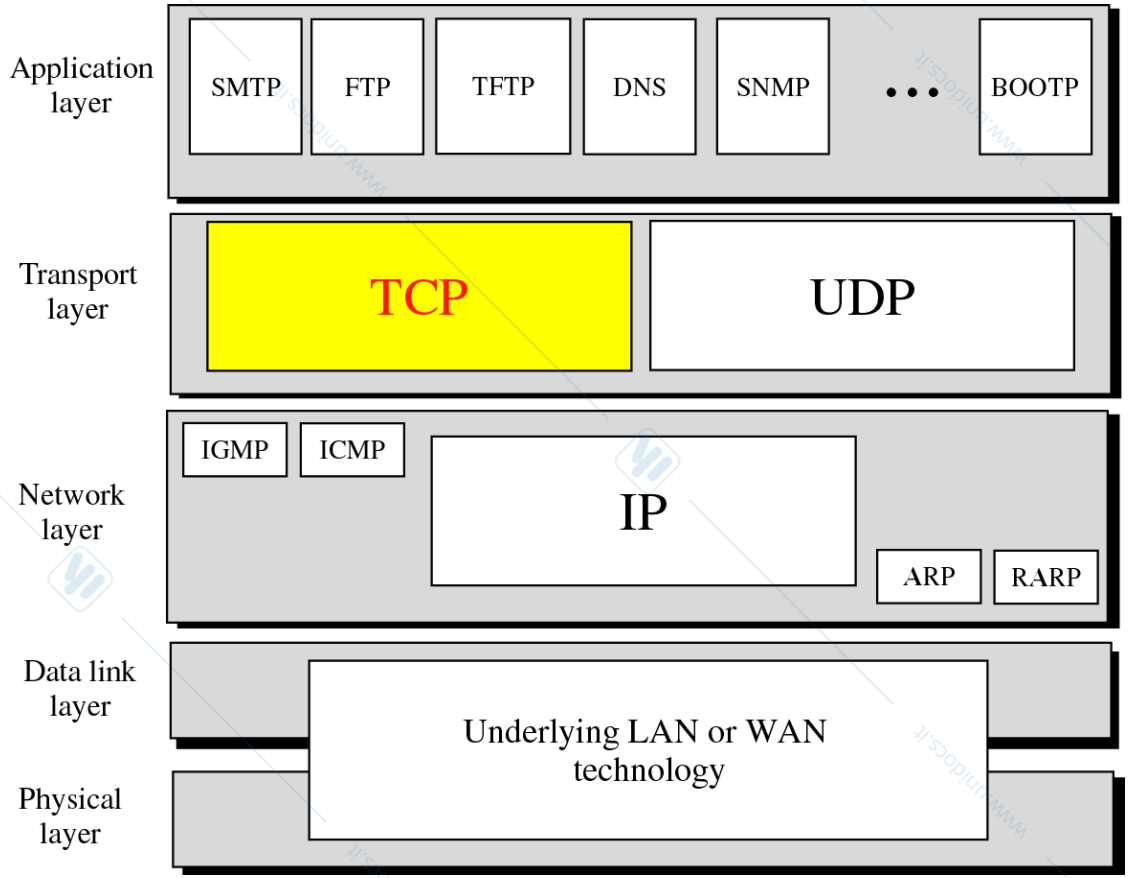
Politecnico di Milano

Dip. di Elettronica, Informazione e Bioingegneria (DEIB)

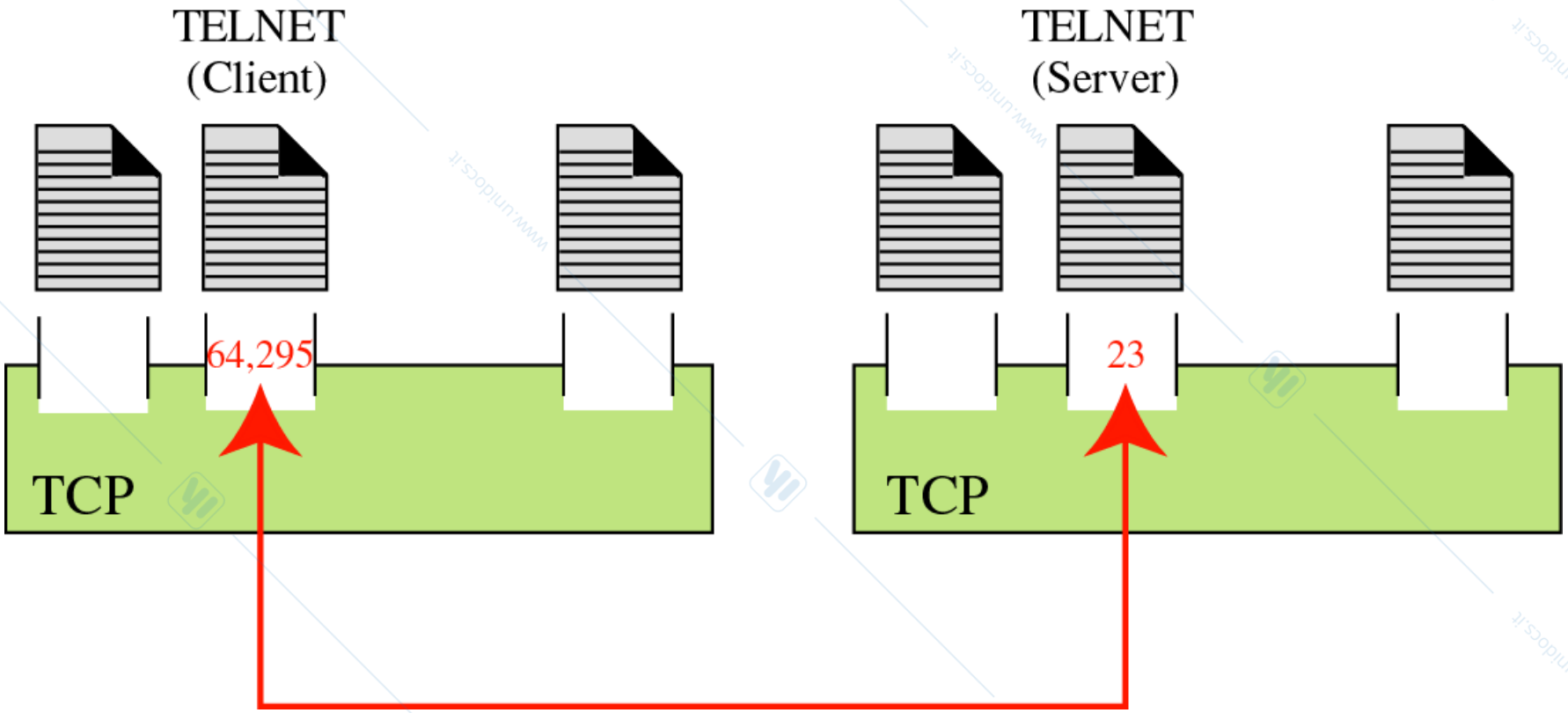
Via Ponzio, 34/5 – 20133 Milan, Italy
massimo.tornatore@polimi.it

Outline

- **Funzioni**
- **Protocollo UDP**
- **Protocollo TCP**
  - Generalità
  - Gestione delle connessioni
  - Controllo di flusso e trasporto dei dati
  - Meccanismo di ritrasmissione
  - Controllo di congestione

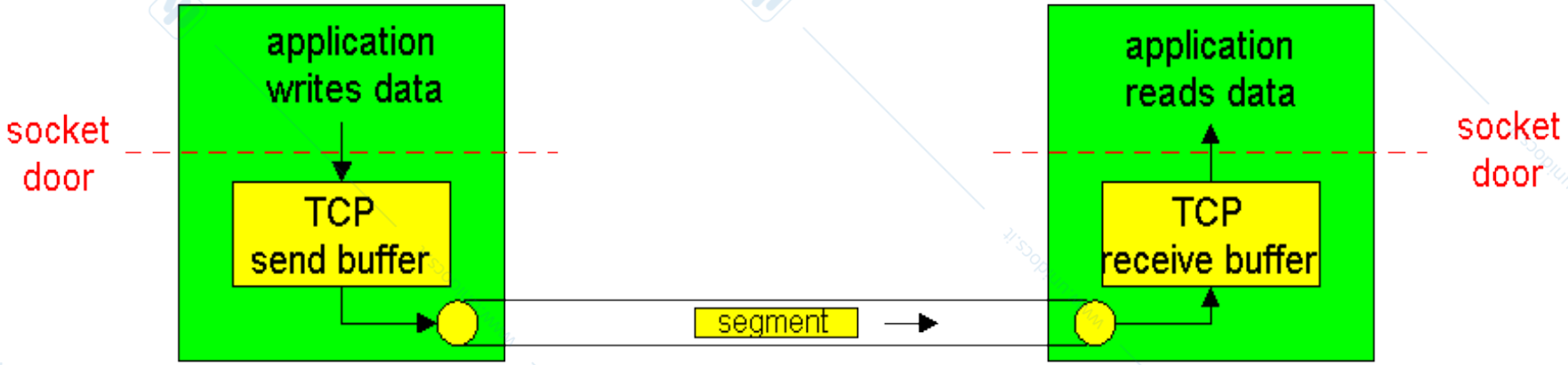


Protocolli di trasporto
TCP

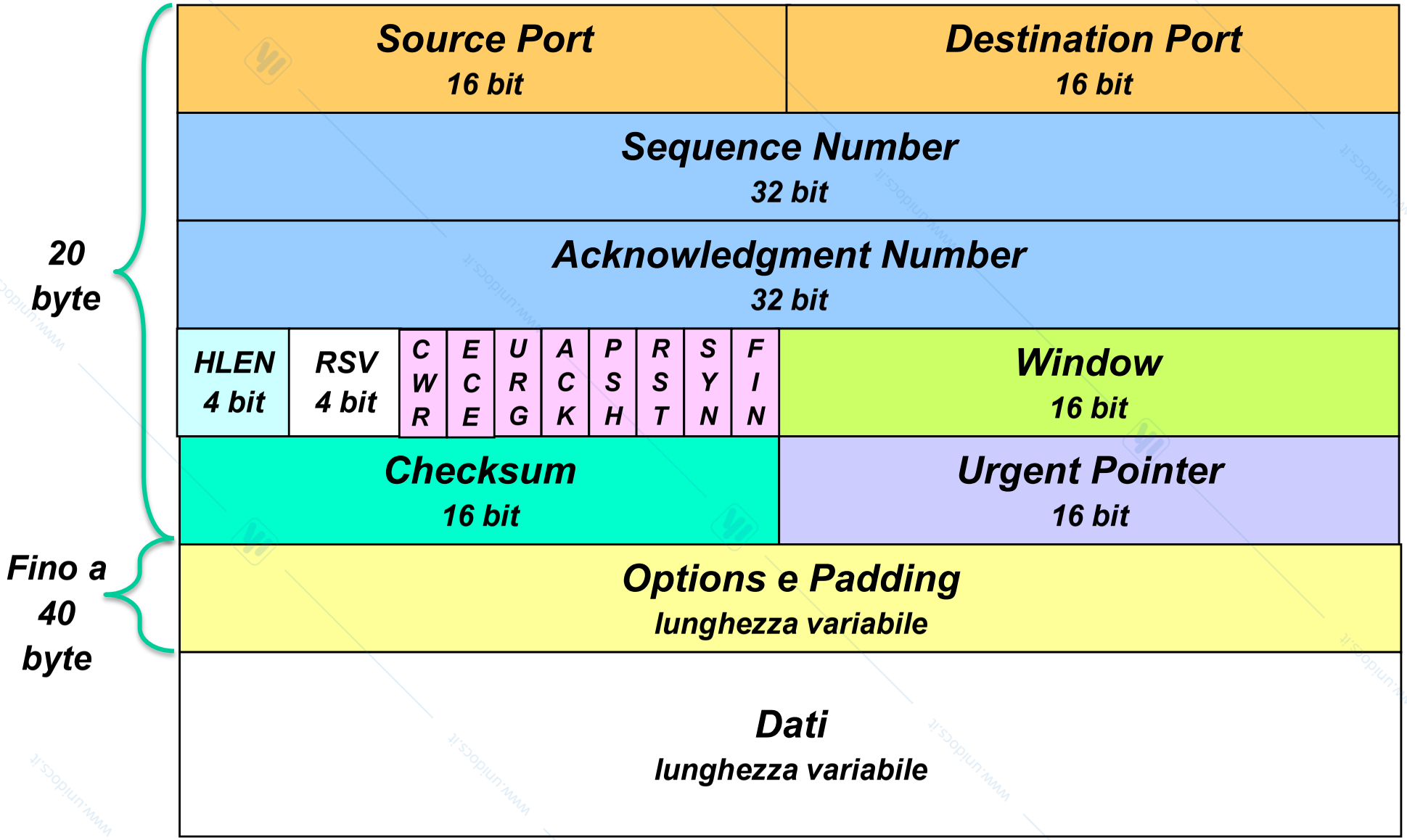


TCP
Generalità

- **Transmission Control Protocol (TCP): RFC 793, RFC 2581, ecc.**
  - Protocollo connection-oriented (full duplex connections)
  - Servizio di trasporto affidabile
  - Flusso di dati organizzato in *gruppi di byte* a loro volta organizzati in *segmenti* di dimensione variabile
  - Fornisce
    - Multiplazione per mezzo dei socket
    - Consegna in sequenza (UI ricevuti anche fuori sequenza)
    - Controllo di flusso e di perdita
    - Controllo di congestione



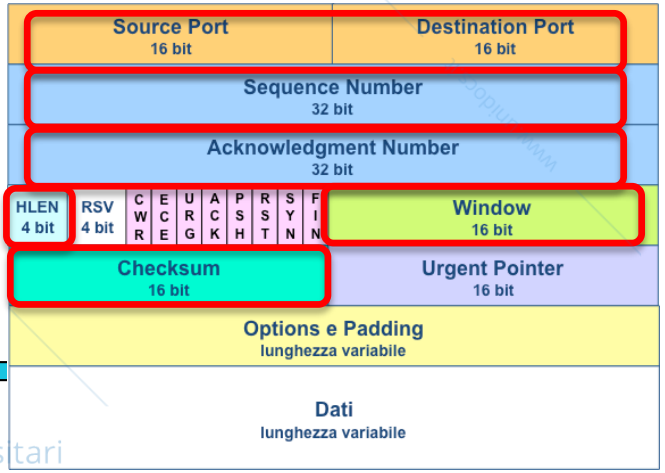
TCP
Formato del segmento



TCP

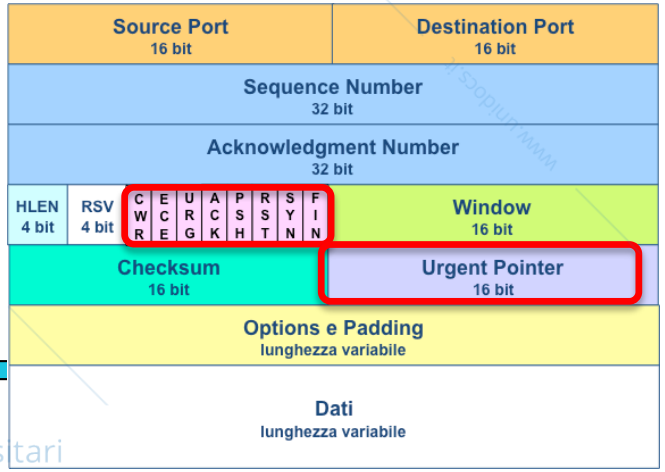
Header segmento TCP (1)

- **Source port, Destination port** (16 bit ciascuno): indirizzi di porta sorgente e porta destinazione
- **Sequence Number** (32 bit): numero di sequenza del primo byte nel payload ≈ N(S) di HDLC 32 bit → numerazione max modulo 2^{32}
- **Ack Number** (32 bit): numero di sequenza del **prossimo byte che si intende ricevere** (numero valido solo se flag ACK valido) ≈ N(R) di HDLC
- **HLEN** (4 bit): contiene la lunghezza complessiva dell'**header** TCP, espressa come numero di «parole» da 32 bit Max HLEN = 5 (base) + 10 (opt) = 15
  - eventualmente si usa *padding*
- **Window** (16 bit): contiene il valore della finestra di ricezione come comunicato dal ricevitore al trasmettitore
- **Checksum** (16 bit): calcolato come in UDP
  - include *pseudoheader*



TCP
Header segmento TCP (2)

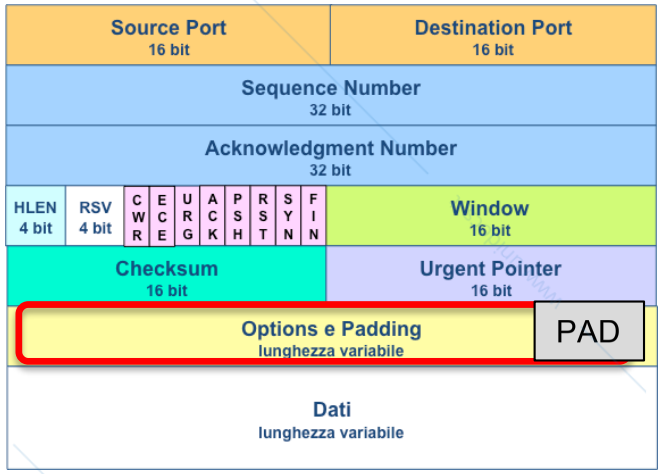
- **Flags** (1 bit ciascuno)**:** 6 campi indipendenti, ciascuno assume valore 0 o 1
  - **CWR** (*congestion window reduced*): settato quando si notifica al TCP remoto che si è ridotta la *finestra di congestione* (vedremo dopo cos'è)
  - **ECE** (*Explicit Congestion Notification Echo*): serve a segnalare che si è rilevata una congestione e serve a chiedere al TCP remoto di rallentare la trasmissione
  - **URG**: vale 1 se vi sono dati urgenti e quindi il TCP deve passare in modalità urgente; in questo caso ***urgent pointer*** punta all'ultimo byte dei dati all'interno del flusso oltre il quale TCP può tornare in modalità normale
    - **ACK**: vale 1 se il pacchetto è un ACK valido; in questo caso *l'acknowledge number* contiene un numero valido
  - **PSH:** vale 1 quando il trasmettitore intende usare il comando di PUSH; il ricevitore può anche ignorare il comando (dipende dalle implementazioni)
    - **RST:** per resetta la connessione
    - **SYN:** *synchronize*; usato durante il *setup* per comunicare i numeri di sequenza iniziale
    - **FIN:** usato per la chiusura di connessione esplicita



TCP

Header segmento TCP (3)

- **Options: informazioni di stato facoltative**
- **Padding: garantisce che il TCP header sia multiplo di 32 bit**
  - **N.B.** Padding nel payload: considerato SOLO nel calcolo del checksum, come in UDP, ma **non trasmesso veramente** (multiplo 16 bit)

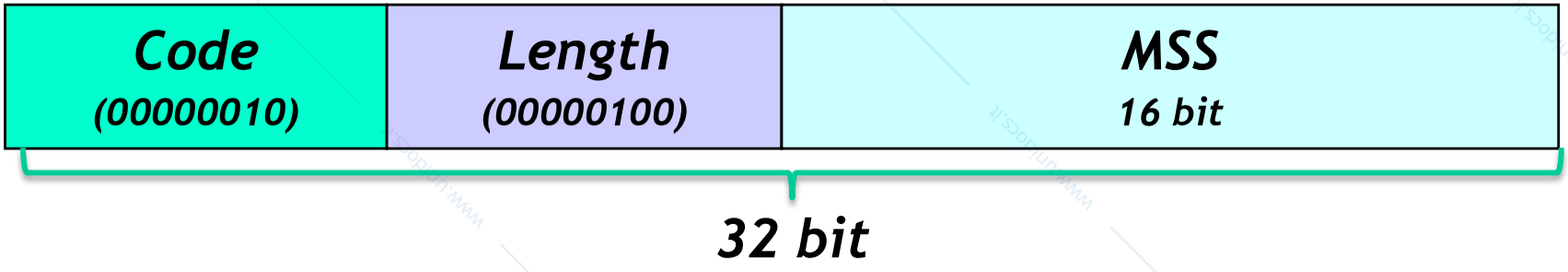


Opzioni

- Le opzioni sono aggiunte all'header TCP
- Diversi tipi di opzioni
  - Opzioni di 1 byte: servono come riempimento per avere un header multiplo di 32 bit
  - Opzioni lunghe:
    - Possiedono un campo «length» che indica la dimensione dell'intero campo option (in byte)
    - Maximum segment size (MSS) – 32 bit
    - Fattore di scala della finestra – 24 bit
    - …
    - http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml

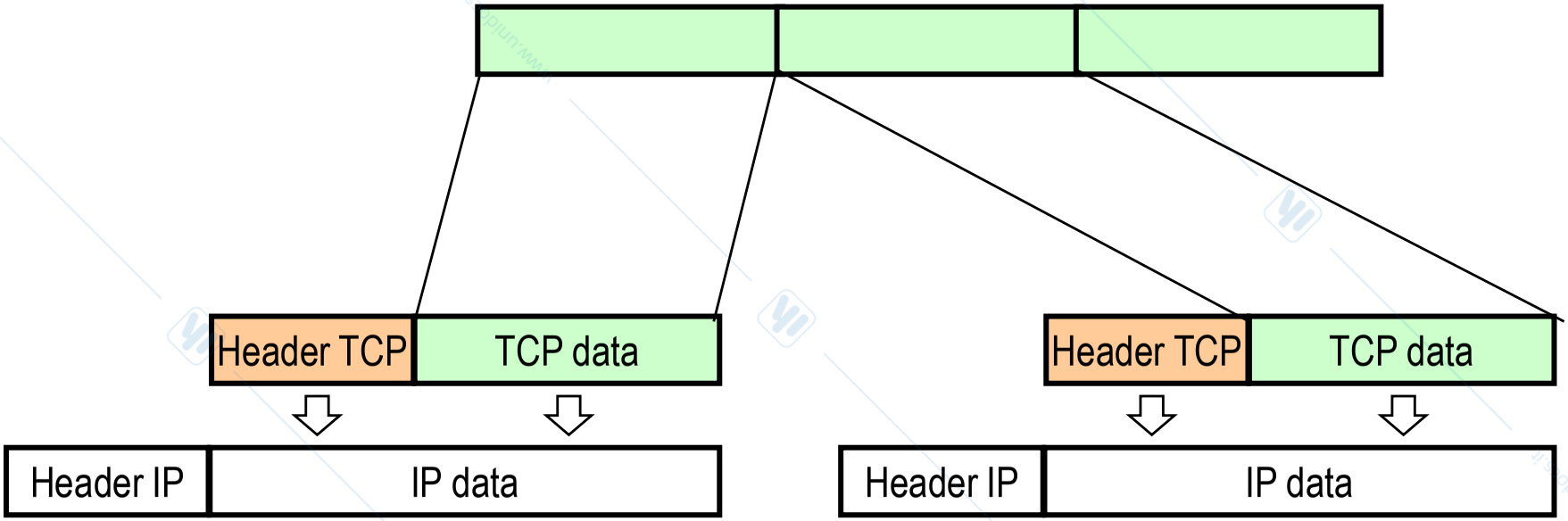
Opzioni
Maximum Segment Size (MSS)

- Definisce la dimensione **massima** del segmento (solo i DATI, non include l'header) che verrà usata nella connessione TCP
- La dimensione è decisa da ciascuna delle due parti durante la fase di *setup* (ciascuna delle parti decide il MSS dei segmenti che riceverà)
- Non si modifica durante la connessione
- Il valore di *default* è 536 byte, il valore massimo **teorico** è 65535 (=2^{16}-1) byte



TCP
Encapsulation e Frammentazione

- Il messaggio dell'applicazione può essere frammentato in più segmenti TCP
- Da ciascun segmento si deve "produrre" esattamente un datagramma IP (anche se IP può operare frammentazione all'occorrenza)

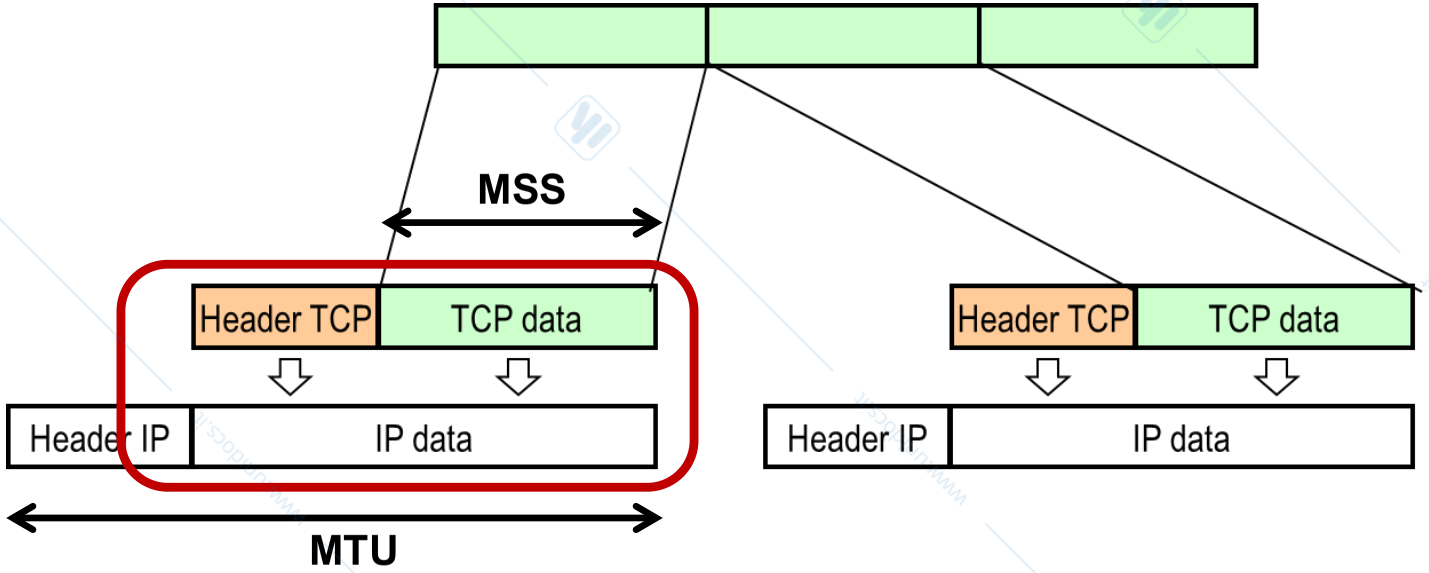


TCP
Encapsulation e Frammentazione

- MSS deve essere scelto in modo da far corrispondere un segmento ad un datagramma, considerando la max dimensione del datagramma (che deriva dalla MTU di livello 2)
  - Interferenza tra layer diversi, che non dovrebbe esserci [RFC 879]

MSS = MTU – size{IP_header} – size {TCP_header}

  - Gli header hanno dimensione variabile → si fanno delle stime
    - Stima "ottimistica": size{IP_header} = size{TCP_header} = 20 byte
    - Stima conservativa: size{IP_header} = size{TCP_header} = 60 byte

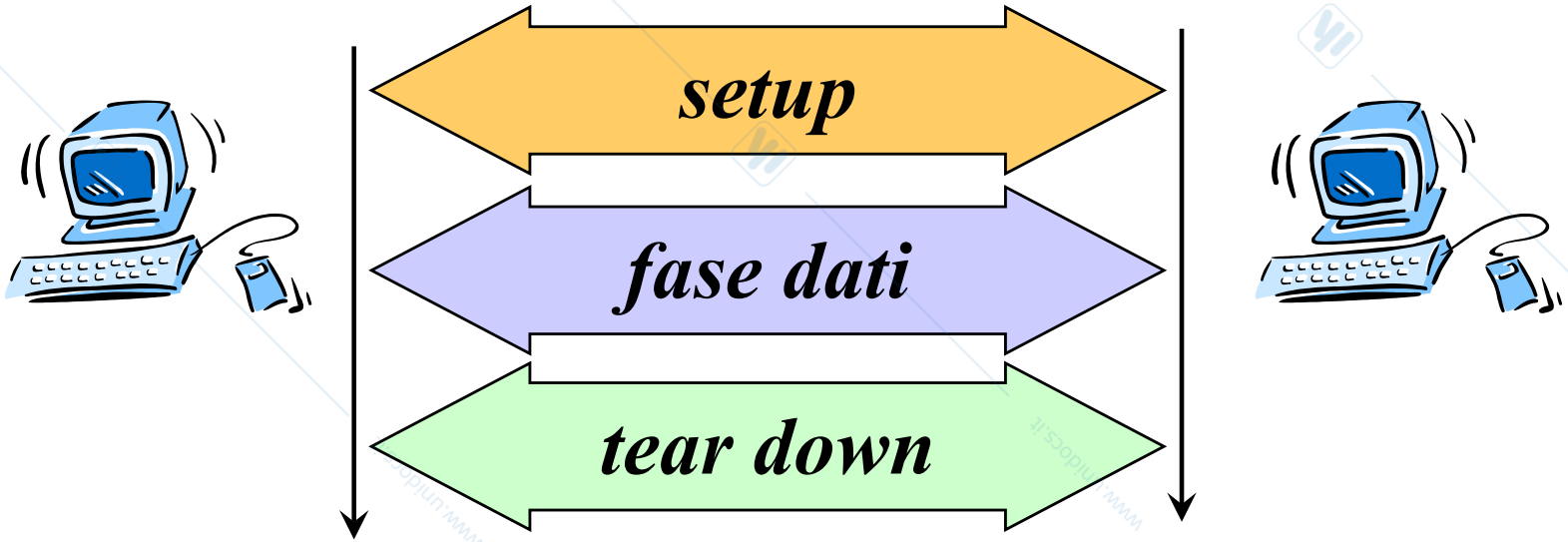


Outline

- **Funzioni**
- **Protocollo UDP**
- **Protocollo TCP**
  - Generalità
  - Gestione delle connessioni
  - Controllo di flusso e trasporto dei dati
  - Meccanismo di ritrasmissione
  - Controllo di congestione

TCP: *connection oriented*

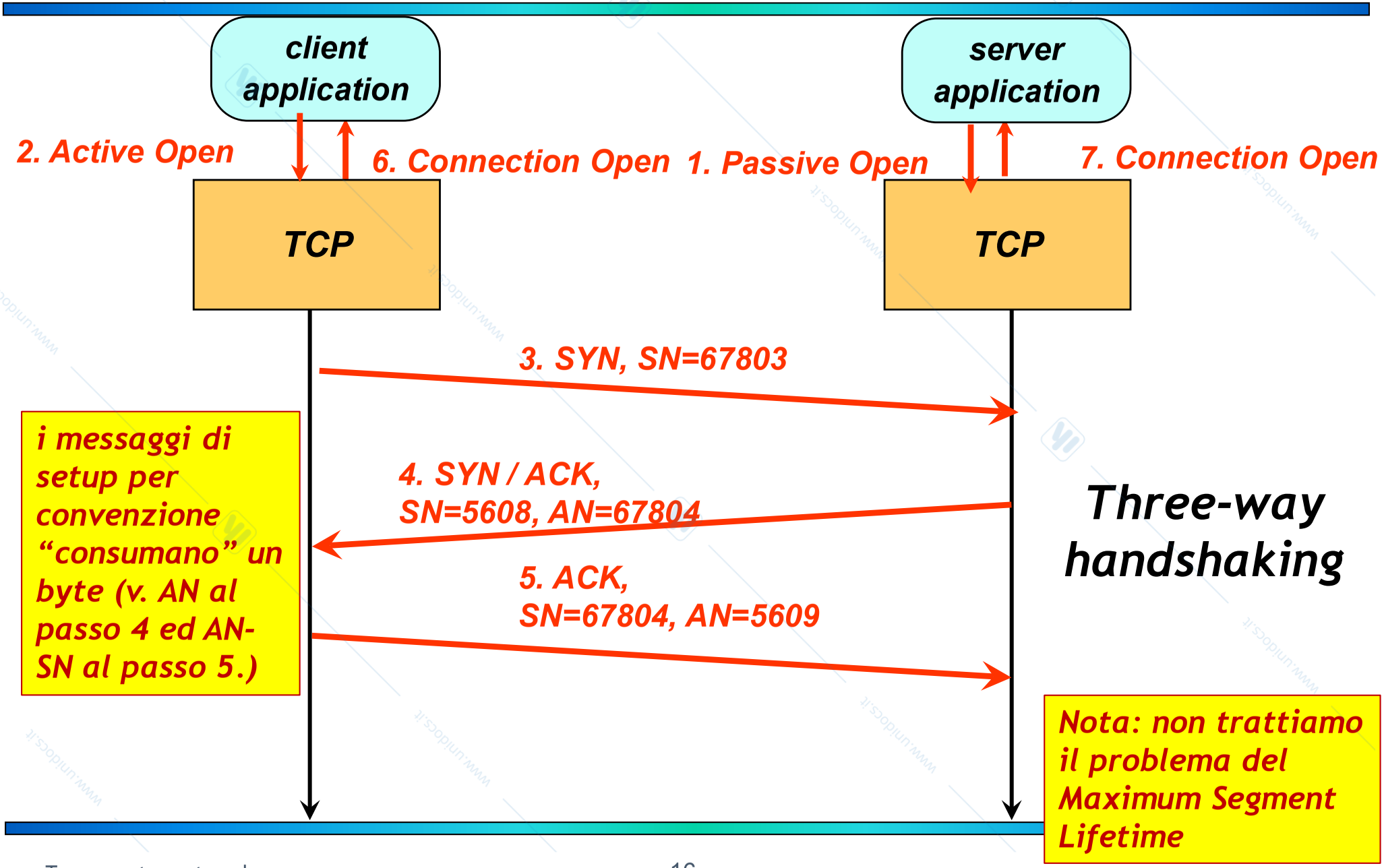
- **Il TCP è orientato alla connessione (*connection oriented*):**
  - prima del trasferimento di un flusso dati occorre instaurare una connessione mediante segnalazione (setup)
  - la connessione TCP è di tipo *full-duplex*



TCP
Gestione della connessione

- **Instaurazione («setup») della connessione TCP**
  - Attuata mediante la procedura del "three-way handshake"
  - Utilizza i bit SYN e ACK

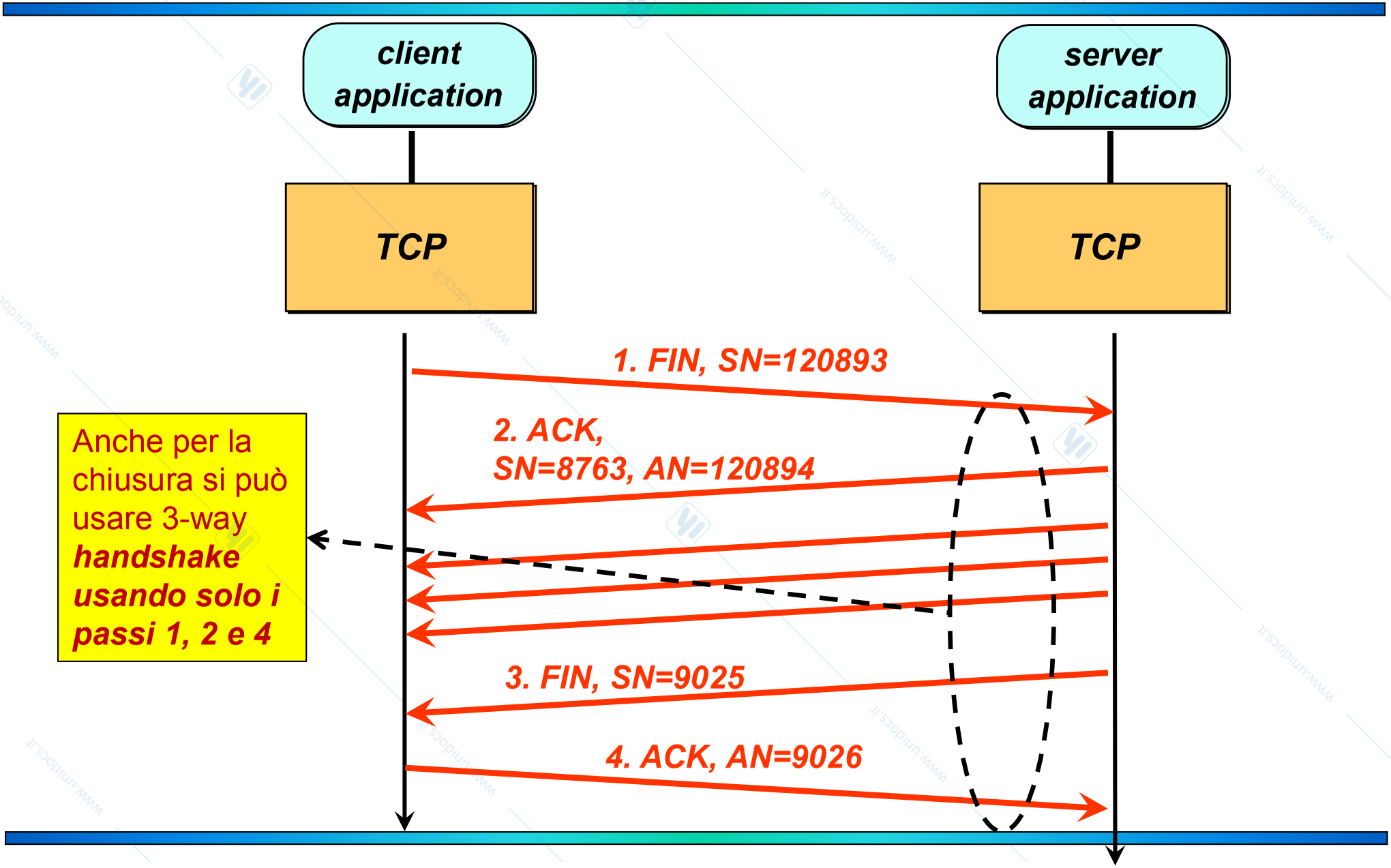
Setup della connessione (sommario)



TCP
Gestione della connessione

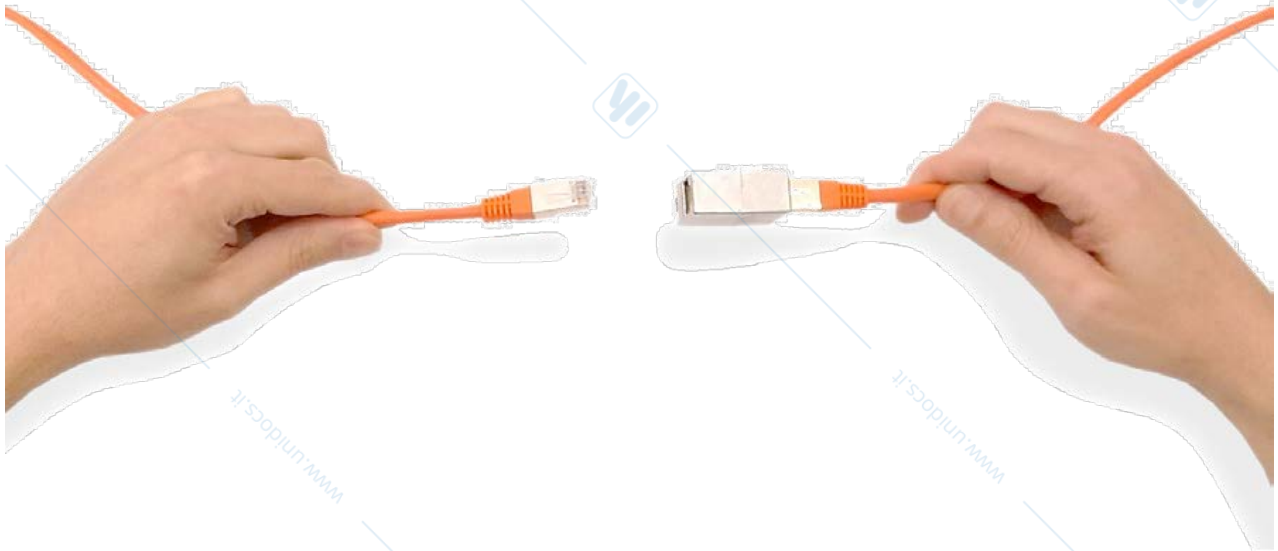
- **Rilascio delle connessione TCP**
  - Iniziato da una qualunque delle due entità TCP
  - Attuato mediante "two-way handshake" per ognuna delle due direzioni
  - Utilizza i bit FIN e ACK

Tear down delle connessioni (sommario)



Reset della connessione

- La connessione può anche essere chiusa senza scambio di messaggi nei due versi
- E' possibile infatti settare il flag di RESET nel segmento e interrompere la connessione in entrambe le direzioni
- Il TCP che riceve un RESET chiude la connessione interrompendo ogni invio di dati



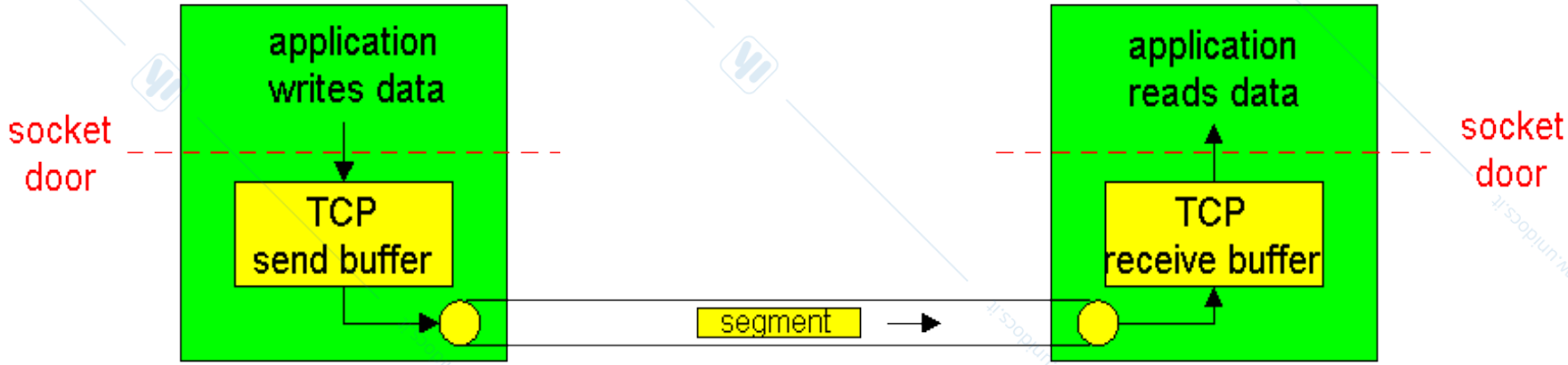
Outline

- **Funzioni**
- **Protocollo UDP**
- **Protocollo TCP**
  - Generalità
  - Gestione delle connessioni
  - Controllo di flusso e trasporto dei dati
  - Meccanismo di ritrasmissione
  - Controllo di congestione

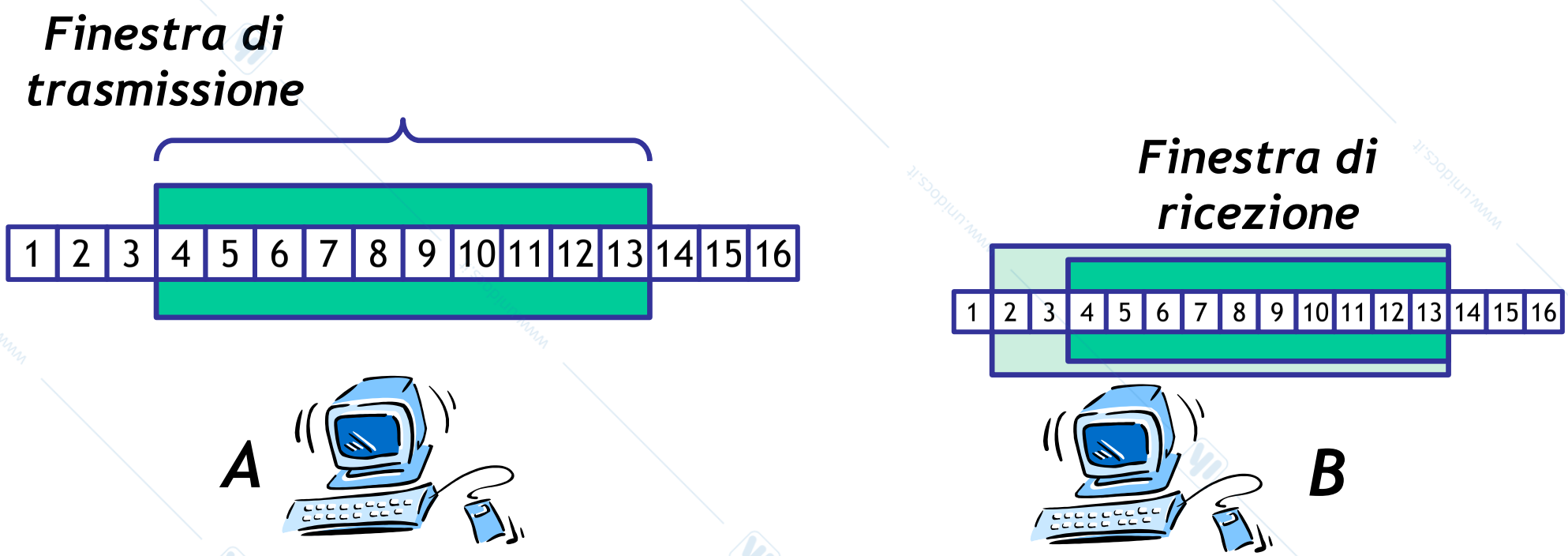
- **Iniziamo a vedere il meccanismo di controllo di flusso a sliding window nel caso senza errori/perdite**

Implementazione del controllo di flusso

- TCP usa due buffer (trasmissione/ricezione) per **controllare il flusso** di quello trasmittente.
- Lato **trasmettitore**:
  - Buffer di trasmissione: accumula i byte in attesa di essere trasmessi (ovvero, in attesa che il ricevitore dia il “consenso” a trasmetterli
- Lato **ricevitore**:
  - Buffer di ricezione: accumula i byte ricevuti e non ancora assorbiti dall’applicazione

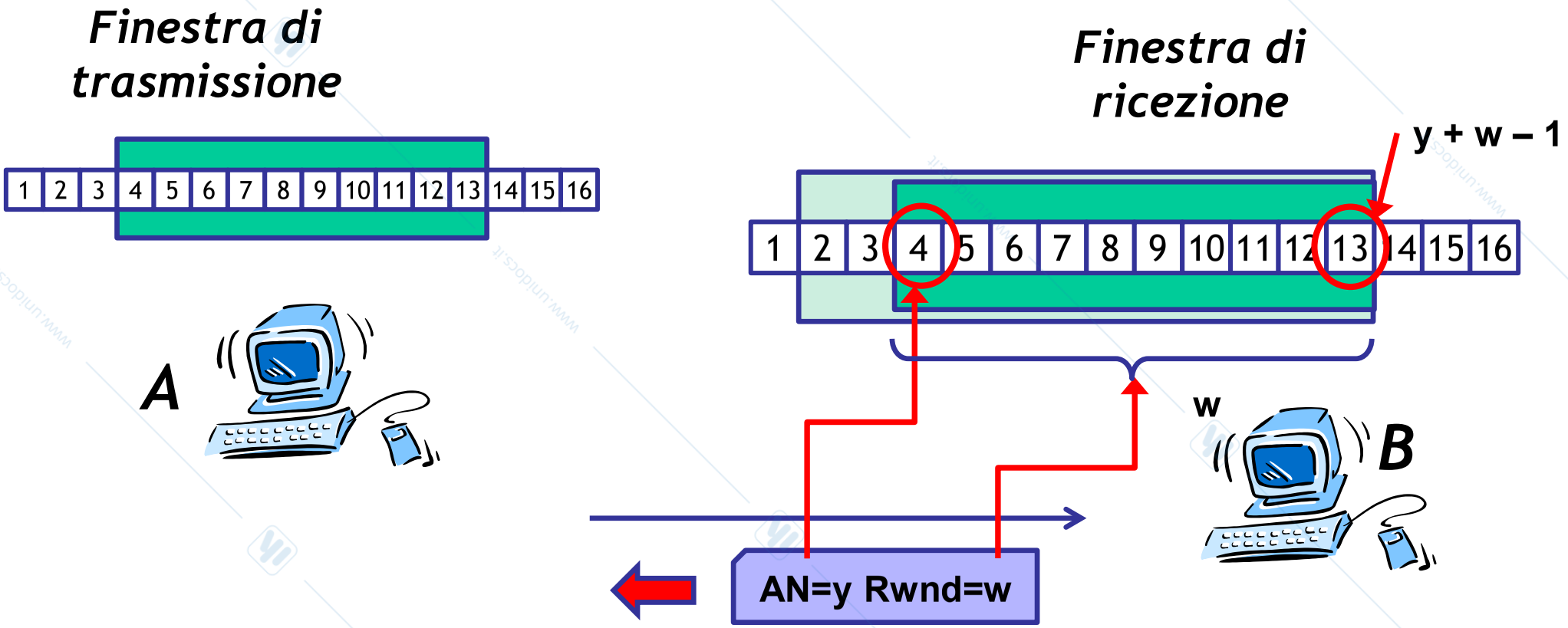


Finestra di *trasmissione e ricezione* e campi TCP

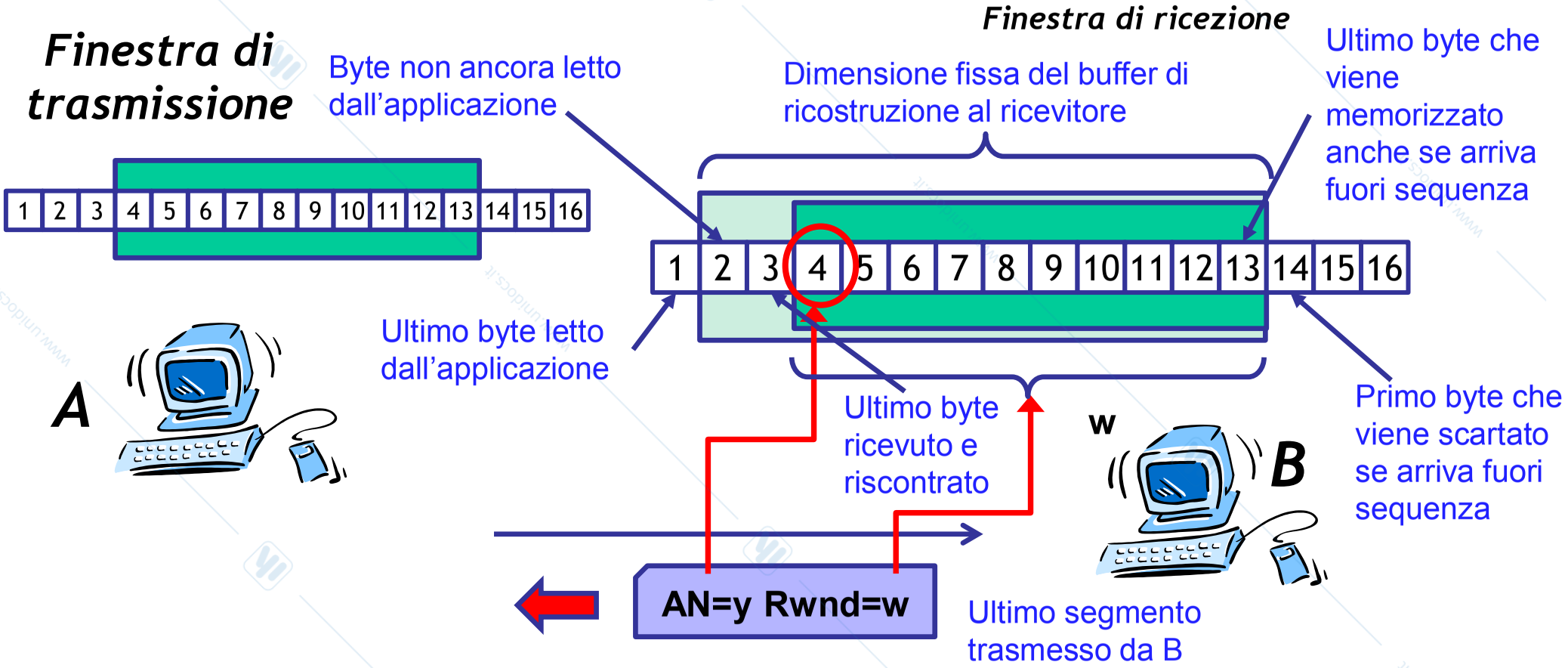


- **Si evidenzia solo la connessione A→B (analogo per B→A)**
- **La finestra di ricezione di B (in assenza di errori e a meno del ritardo di trasferimento) si sposta "in sincronia" con la finestra di trasmissione**
  - B setta il campo «Window» in base alla occupazione corrent del proprio buffer di ricezione
  - A adatta l'ampiezza della finestra di trasmissione in base al valore «window» ricevuto da B

Finestra di *ricezione* e campi TCP

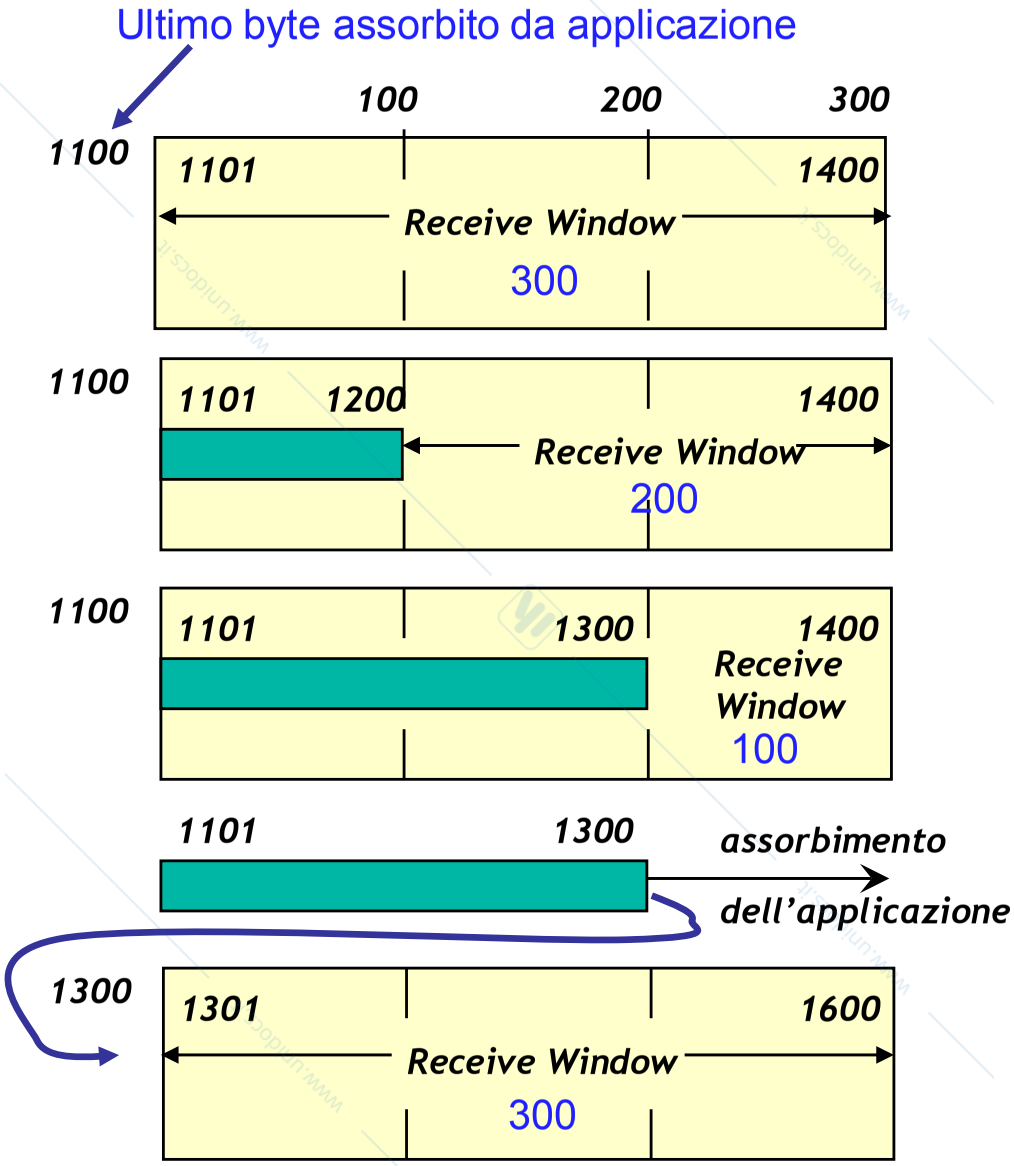


Finestra di *ricezione* e campi TCP



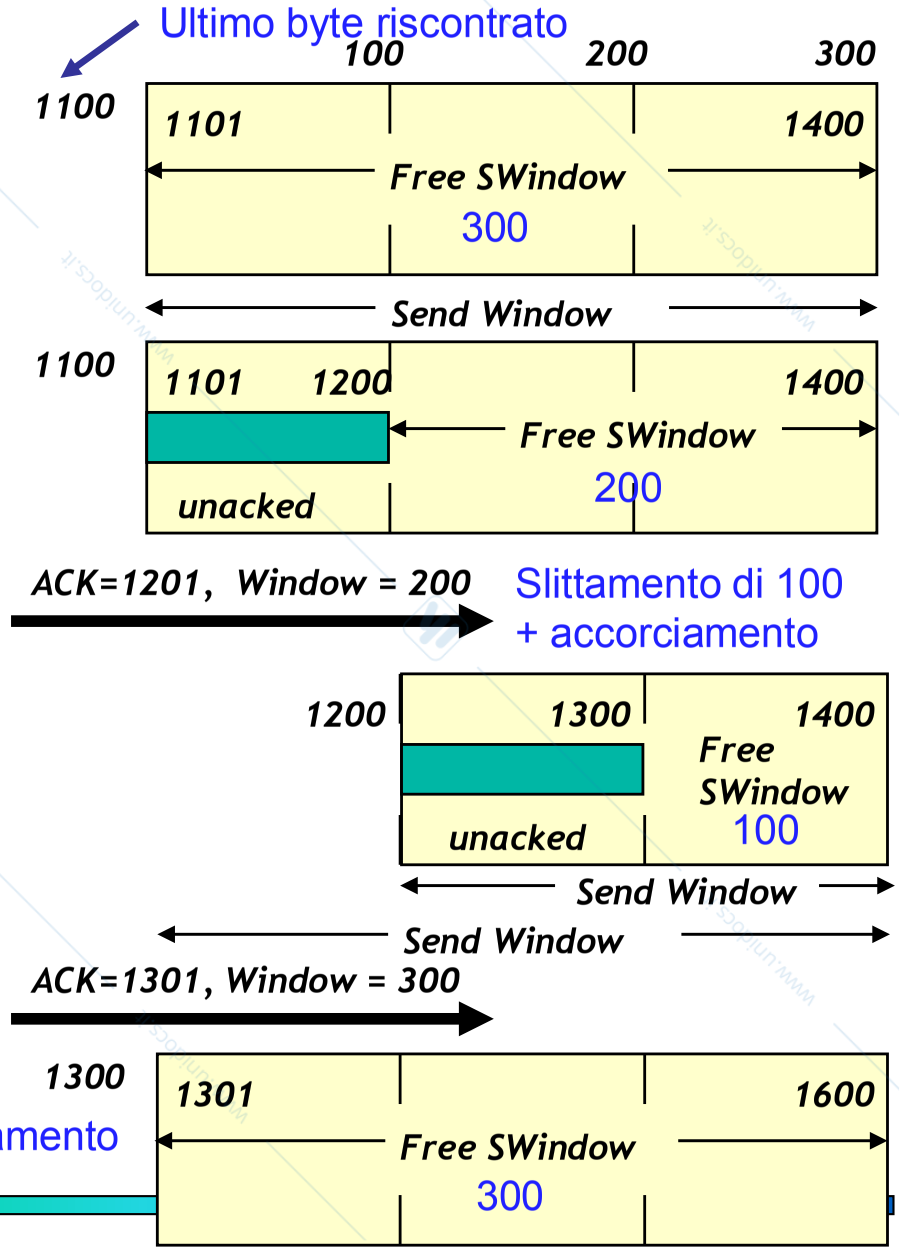
Controllo di flusso: lato ricevitore

- *Receiver Window (Rwnd)*
  - **spazio del buffer in ricezione disponibile per ricevere dati (buffer≠Rwnd)**
- I byte della Rwnd si estendono:
  - dall'ultimo byte ricevuto (e riscontrato), ma non necessariamente assorbiti dalla applicazione
  - fino alla fine del buffer
- La dimensione di *Rwnd* è segnalata in ogni segmento inviato dal ricevitore al trasmettitore



Controllo di flusso: lato trasmettitore

- Il trasmettitore mantiene un buffer di trasmissione per i **dati che sono stati trasmessi ma non ancora riscontrati**
- Se: *Sender Window (Swnd) = Rwnd* → il buffer di trasmissione contiene i byte dal primo byte non riscontrato all'estremo a destra della finestra di ricezione del ricevitore
- La parte inutilizzata del buffer (gialla), rappresenta i byte che possono essere trasmessi senza attendere ulteriori riscontri



* Può anche verificarsi il caso in cui Swnd ≠ Rwnd (in particolare Swnd <Rwnd).
Ciò avviene quando si ha un vincolo più stringente per effetto del controllo di congestione (v. dopo)

- **Non copriamo il problema della «silly window» (algoritmi di Nagle e Clark)**

Outline

- **Funzioni**
- **Protocollo UDP**
- **Protocollo TCP**
  - Generalità
  - Gestione delle connessioni
  - Controllo di flusso e trasporto dei dati
  - Meccanismo di ritrasmissione
  - Controllo di congestione

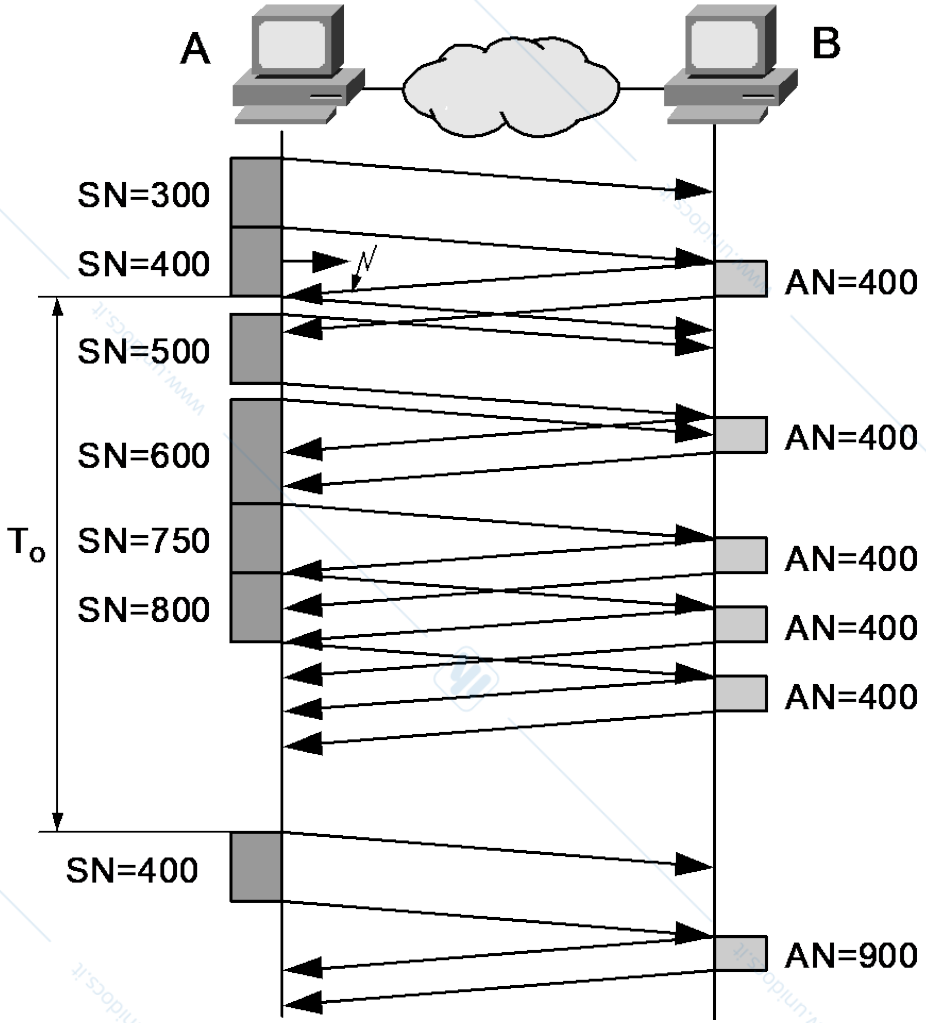
- **Vediamo ora come il meccanismo di controllo di flusso a sliding window funziona nel caso con errori/perdite**

TCP error control: un misto di GBN e SR

Ritrasmissione

- **TCP non usa NACK** (vedi slide seguente)
  - Le ritrasmissioni avvengono solo per scadenza di timeout

Esempio di gestione dell'errore con soli riscontri positivi e scadenza del time out

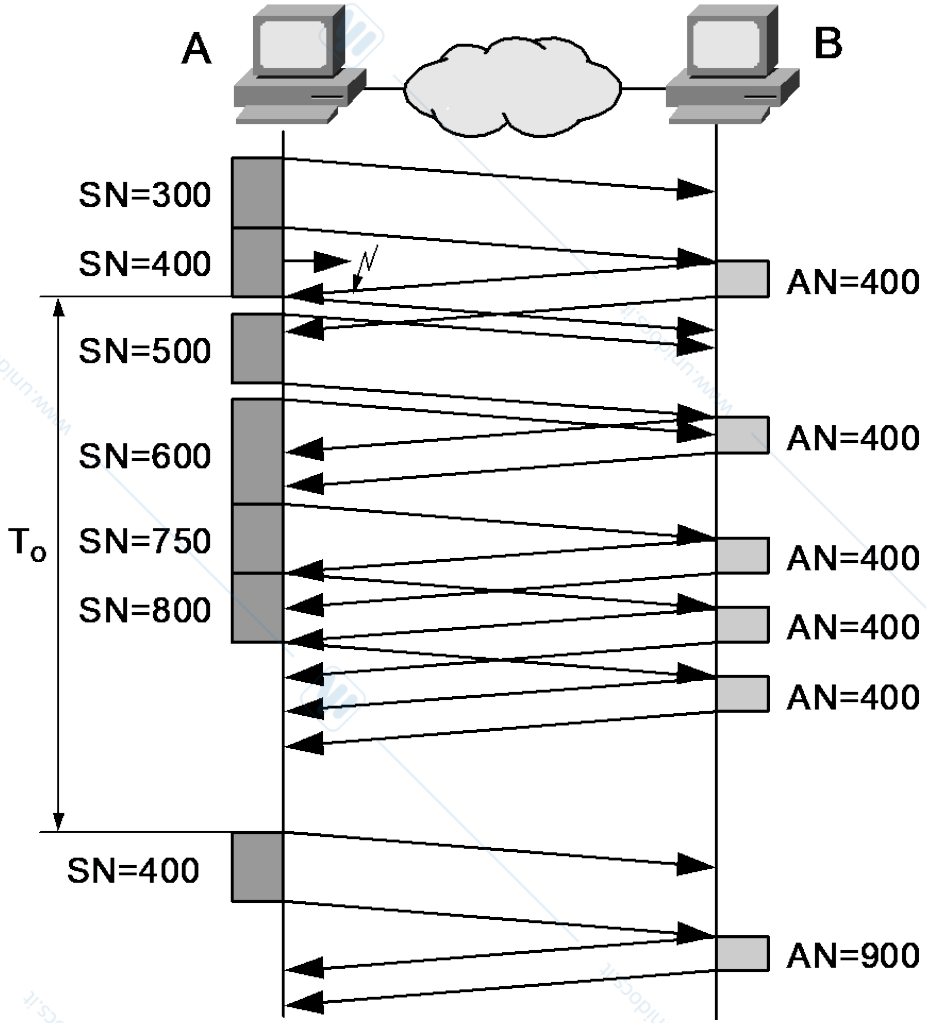


TCP
Fast retransmit

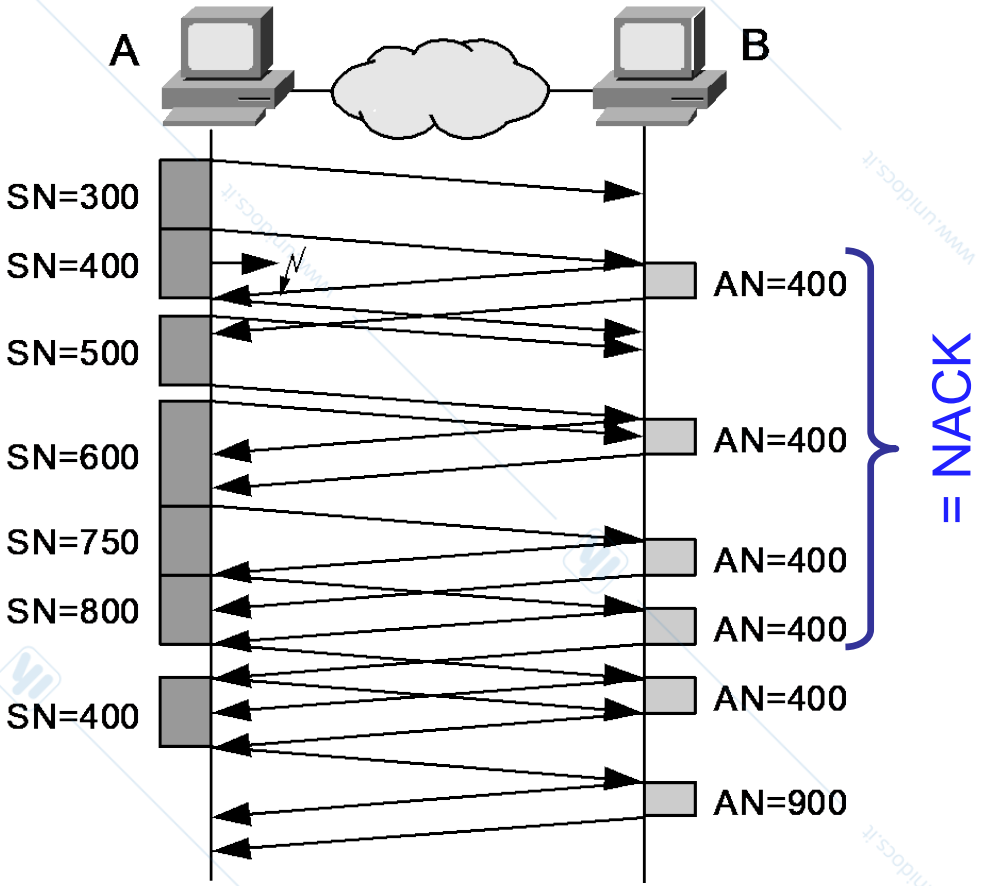
- **Ritrasmissione «standard»**
  - A ritrasmette il segmento perso solo dopo la scadenza del timeout
    - Se un segmento (es. il primo) di una sequenza va perso, il ricevitore B può mantenere (TCP Reno, simile a GBN) o meno (TCP Tahoe, simile a SR) i seguenti nel buffer
    - Un ACK cumulativo verrà trasmesso da B solo dopo aver ricevuto il segmento mancante

  → sarebbe più efficiente che A lo ritrasmettesse più velocemente
- **Upgrade del meccanismo di ritrasmissione: *Fast retransmit* (RFC 2001)**
  - Alla ricezione di 3 ACK consecutivi "ripetuti", si procede a ritrasmissione anche prima che scatti il time-out (equivale a un NACK implicito)

Fast retransmit
Esempio



Senza Fast retransmit



Si noti che non esiste NACK

Con Fast retransmit

TCP

Controllo di flusso con errori (ritrasmissioni)

- **Perché non usiamo Go-Back-N o SR esattamente come in HDLC?**
- **In altre parole, perché non usiamo il NACK?!**
  - Perché c'è una differenza sostanziale tra livello 2 e livello 4
    - Livello 2: Round trip time (RTT) fisso (circa 2τ)
    - Livello 4: Round trip time (RTT) variabile
      - Poiché i segmenti TCP attraversano una RETE, essi subiscono ritardi variabili → RTT non è fisso
      - Un NACK non necessariamente indica un pacchetto perso (è più probabile che sia un fuori sequenza)!
      - Non ci resta che usare il T_{out}

Ma come scegliamo il T_{out}?

- **E' necessario che il timeout sia adattato continuamente al RTT variabile**
  - Se T_o < RTT → ritrasmissioni inutili
  - Se T_o » RTT → protocollo inefficiente
- Il TCP misura continuamente l'RTT per effettuare delle ***stime*** utili a impostare un valore "appropriato" di timeout

TCP
Stima di RTT

- Utilizzo di un *filtro a media mobile*
- RTT viene misurato per **ogni segmento** *al ricevimento del rispettivo riscontro*
  - La misura dell'RTT dell'ultimo segmento è detta RTT_{last}

- **Stima corrente (Smoothed RTT) data da**

$$RTT_{av} = (1-\alpha)RTT_{av} + \alpha \cdot RTT_{last} \qquad 0 < \alpha \leq 1$$

- **Stima della variabilità di *RTT***

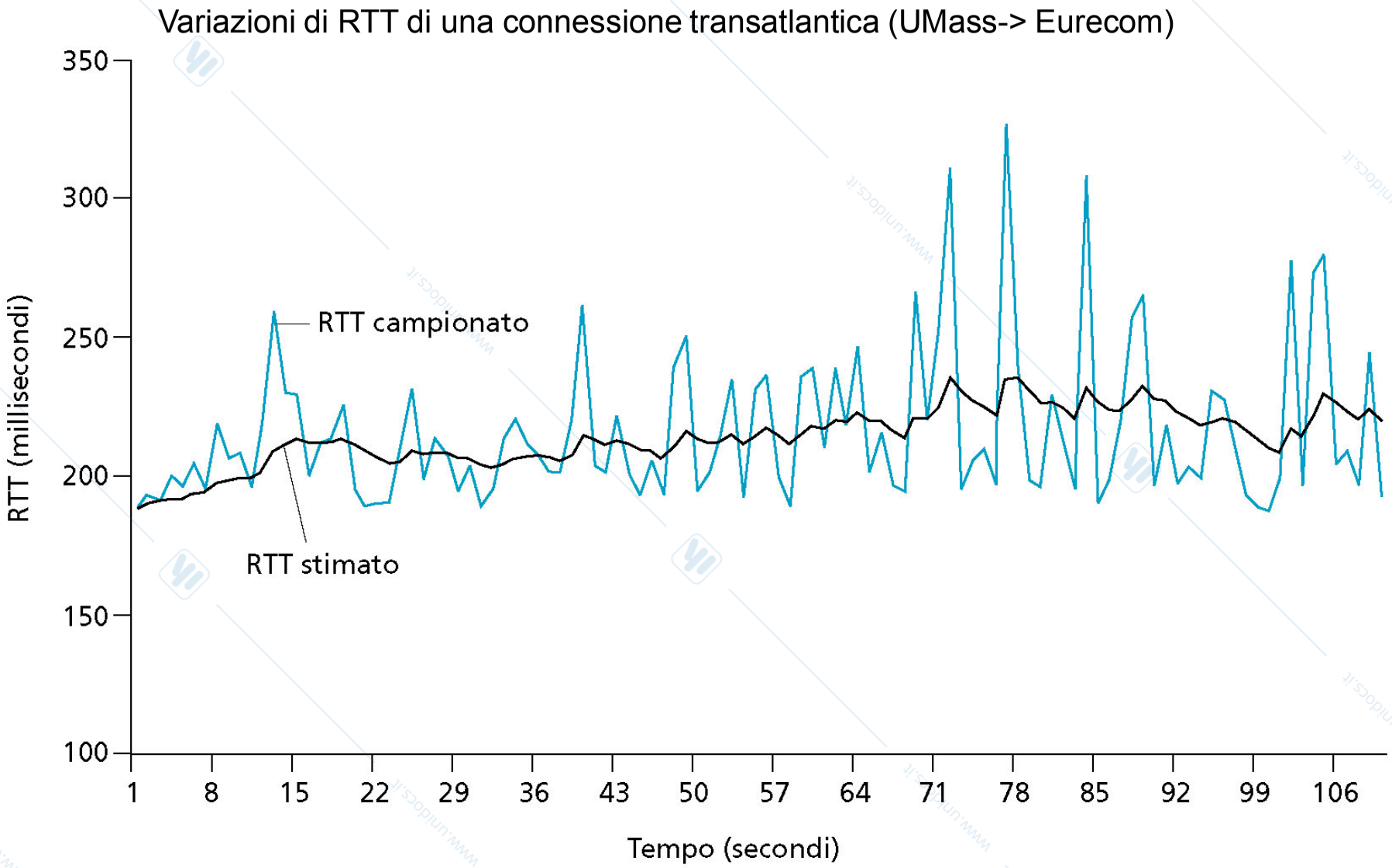
$$RTT_{dev} = (1-\beta)RTT_{dev} + \beta \cdot |RTT_{last} - RTT_{av}| \qquad 0 < \beta \leq 1$$

- **Valore tipico** $\alpha = 0.125$, $\beta = 0.25$
- **Attenzione! Step 0 e 1 sono trattati come casi particolari**

  Step 0: $RTT_{av} = 0$; $RTT_{dev} = 1.5s$

  Step 1 (ricezione primo ack): $RTT_{av} = RTT_{last}$; $RTT_{dev} = RTT_{last}/2$

TCP
Stima di RTT



TCP
Timeout

- **Normalmente il timeout viene determinato sulla base del RTT stimato (Jacobson)**

$$T_o = RTT_{av} + n \cdot RTT_{dev}$$

Valori di n tipici: 2, 3, 4

(T_o iniziale = 3s con n = 2)

- **Esercizio: scrivere i valori di T_{out} per una serie di tre pacchetti ricevuti con RTT_1, RTT_2, RTT_3 pari a 5, 10, 6 secondi. Si consideri $\alpha=\beta=0.4$.**

Un esempio di calcolo del T_{out} su 3 campioni

- **Due computer A e B si scambiano informazione**
- **Computer A sperimenta i seguenti RTT (in s): RTT_1=5, RTT_2=10, RTT_3=6**
- **Media matematica (5+10+6)/3=7**
- **α=0.4, β=0.4, n=2**

$$RTT_{av} = (1-\alpha)RTT_{av} + \alpha \cdot RTT_{last} \qquad RTT_{dev} = (1-\beta)RTT_{dev} + \beta \cdot |RTT_{last} - RTT_{av}| \qquad T_o = RTT_{av} + n \cdot RTT_{dev}$$

Step	RTT_{av}	RTT_{dev}	T_o
0	$RTT_{av}(0)$=0 [input]	$RTT_{dev}(0)$=1,5 [input]	$T_o(0)$=3
1	$RTT_{av}(1)=RTT_{last}=RTT_1$=5	$RTT_{dev}(1)=RTT_{last}$/2=2.5	T_o=5+2*2.5=10
2	$RTT_{av}(2)$=0.6*5+0.4*10=7	$RTT_{dev}(2)$=0.6*2.5+0.4*|10-7| 7| =2,7	T_o=7+2*2.7=12,4
3	$RTT_{av}(3)$=0.6*7+0.4*6=6.6	$RTT_{dev}(3)$=0.6*2.7+0.4*|6-6.6| =1.38	T_o=6.6+2*1.38=9.36

Algoritmo di Karn

- **OSS:** Non si valuta *RTT* per i segmenti che vengono ritrasmessi
- **Problema: Con rete improvvisamente congestionata quasi tutti i segmenti vengono ritrasmessi → ma, siccome non si valuta RTT per i segmenti che vengono ritrasmessi, il time-out non verrebbe più aggiornato!**
  - Circolo vizioso: mancati aggiornamenti, aumentano le ritrasmissioni
- **Soluzione: Algoritmo di Karn (timer backoff): ad ogni ritrasmissione dovuta a scadenza di timeout, il timeout viene aumentato**
  - Timeout aumentato con tecnica moltiplicativa (tipicamente × 2)
  - Aumento avviene fino a una soglia (per esempio 60 s)
  - Al primo segmento ricevuto senza scadenza del time-out si torna alla procedura standard

Outline

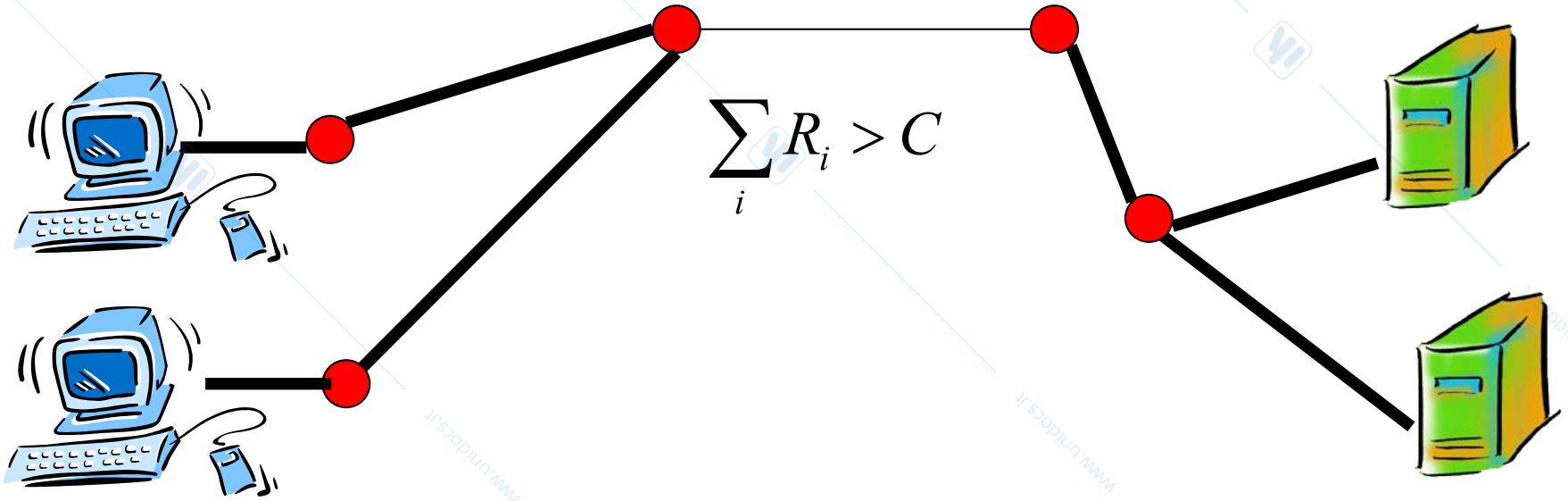
- **Funzioni**
- **Protocollo UDP**
- **Protocollo TCP**
  - Generalità
  - Gestione delle connessioni
  - Controllo di flusso e trasporto dei dati
  - Meccanismo di ritrasmissione
  - Controllo di congestione

Controllo di congestione

- Il controllo di flusso
  - dipende solo dalla ***capacità del ricevitore***
  - non è sufficiente ad evitare la ***congestione nella rete***
- ***Il controllo di congestione è delegato al TCP!!!***
  - Essendo il TCP implementato solo negli *host*, il controllo di congestione è di tipo *end-to-end*

Controllo di congestione

- Un **evento di congestione** si verifica quando il rate di trasmissione porta in congestione un link sul percorso in rete verso la destinazione
- Un link è congestionato quando la somma dei rate di trasmissione dei flussi che lo attraversano è maggiore della sua capacità

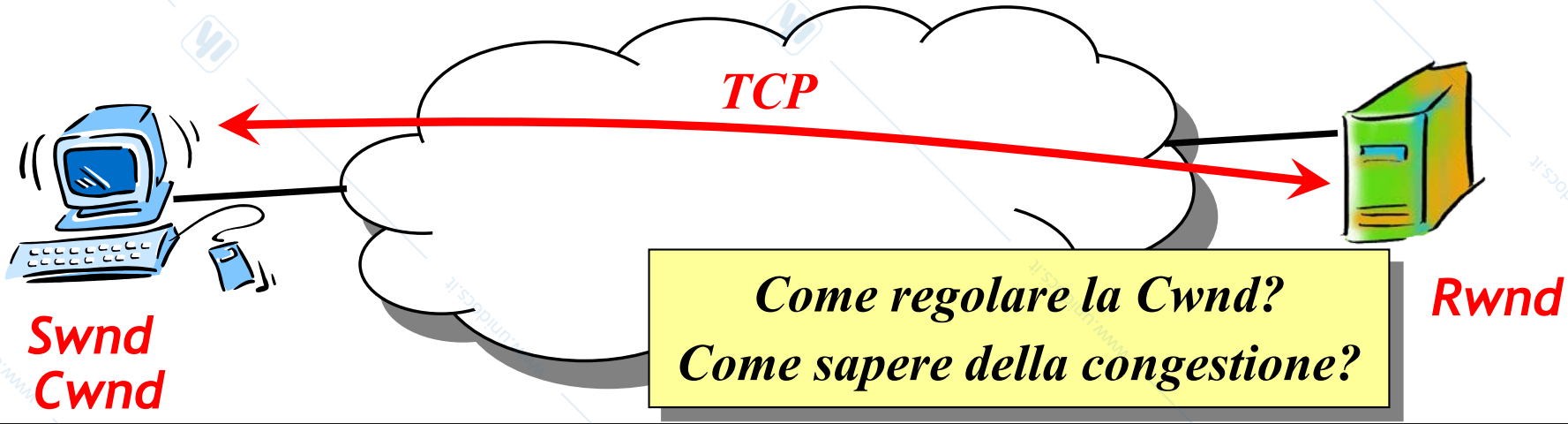


Controllo di congestione

- Per controllare il ritmo di immissione in rete dei dati, il TCP può ***regolare la finestra di trasmissione (Swnd)***

Swnd = min (Rwnd, Cwnd)

- In ogni istante la finestra del trasmettitore *Swnd* (=W_s) è dimensionata al minimo tra *Rwnd* e *Cwnd*
- ***Congestion Window*** (**Cwnd**) varia in base allo stato della rete

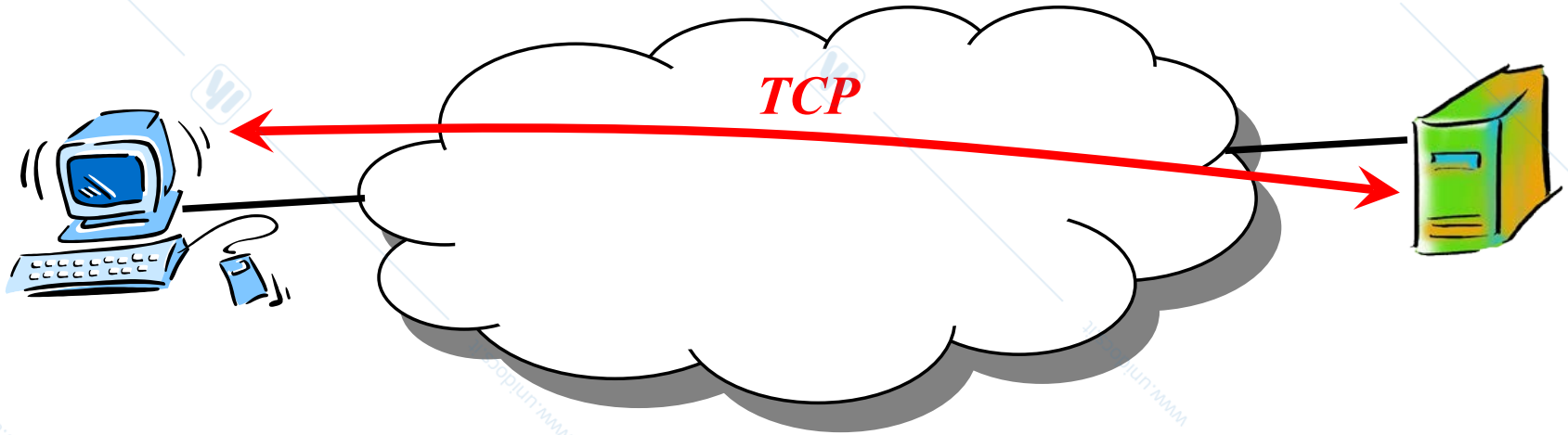


Controllo di congestione

- L'idea base del controllo di congestione del TCP è:

  perdita di un segmento (scadenza T_o)
  → significa «evento di congestione»

- La reazione ad un evento di congestione è quella di **ridurre l'ampiezza della finestra di congestione *(Cwnd)***



Slow start & Congestion avoidance

- Il valore della finestra *Cwnd* viene aggiornato dal trasmettitore TCP in base ad un algoritmo in due fasi
  - ***Slow start***
  - ***Congestion avoidance***

- Per distinguere tra le due fasi si usa una ***soglia***, ***Ssthresh***, che è gestita dall'entità TCP al trasmettitore:
  - ➔ se *Cwnd* < *Ssthresh* si è in *Slow start*
  - ➔ se *Cwnd* ≥ *Ssthresh* si è in *Congestion avoidance*

Aggiornamento della Finestra di Congestione

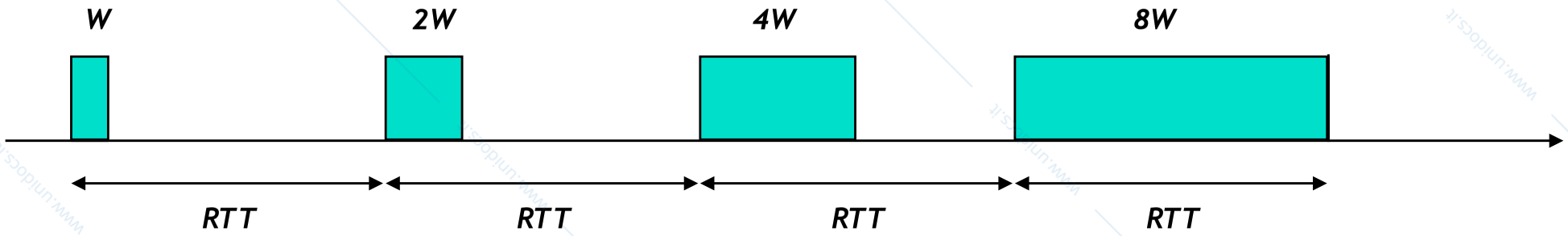
- Vediamo ora come viene aggiornato il valore della finestra *Cwnd* dal trasmettitore TCP in queste due fasi
  - *Slow start*
  - *Congestion avoidance*

Slow start

- All'inizio di una connessione, il trasmettitore pone la *Cwnd* a 1 segmento (MSS) e la Ssthresh ad un valore di *default* molto elevato
- Essendo *Cwnd* < *Ssthresh* si inizia in *Slow start*

- *Regola di incremento della Cwnd in Slow Start:*
  - *la Cwnd viene incrementata di 1 MSS per ogni ACK ricevuto*

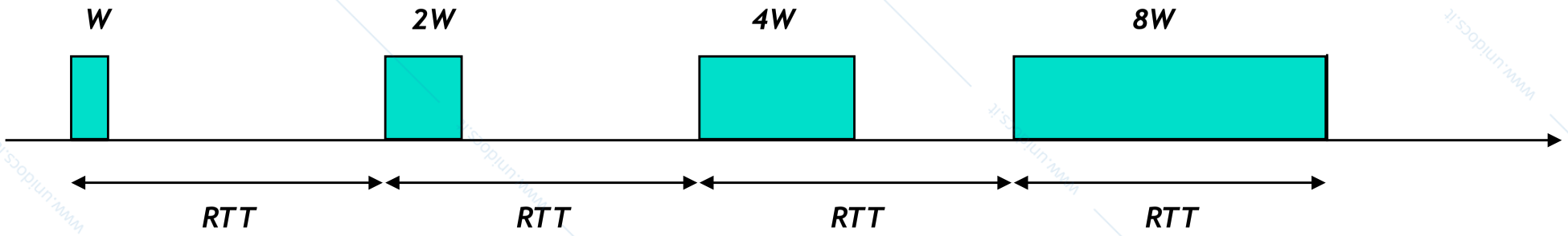
- Si invia un segmento e dopo un intervallo = RTT si riceverà l'ACK; quindi si pone *Cwnd* a 2 e si inviano 2 segmenti, si ricevono 2 ACK, si pone *Cwnd* a 4 e si inviano 4 segmenti, ...



Slow start

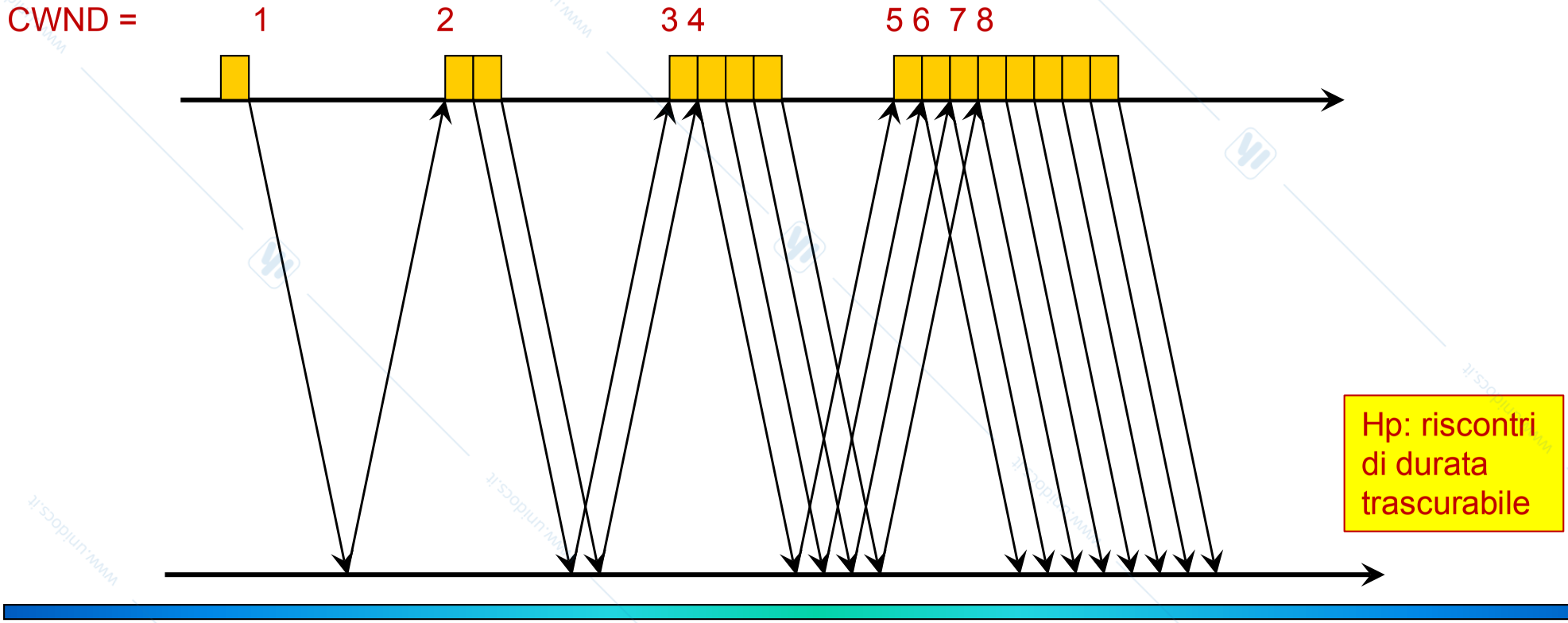
- All'inizio di una connessione, il trasmettitore pone la *Cwnd* a 1 segmento (MSS) e la Ssthresh ad un valore di *default* molto elevato
- Essendo *Cwnd* < *Ssthresh* si inizia in *Slow start*
- *Regola di incremento della Cwnd in*
  - *la Cwnd viene incrementata di 1*
- Si invia un segmento e dopo un intervallo = RTT si riceverà l'ACK; quindi si pone *Cwnd* a 2 e si inviano 2 segmenti, si ricevono 2 ACK, si pone *Cwnd* a 4 e si inviano 4 segmenti, ...

N.B. In realtà, la finestra di congestione è espressa in **numero di byte** (così come la finestra di ricezione vista nel controllo di flusso). Per semplicità, facciamo riferimento all'unità di misura *MSS*.



Slow start

- Al contrario di quanto il nome faccia credere, l'incremento della finestra avviene in modo esponenziale (raddoppia ogni RTT)

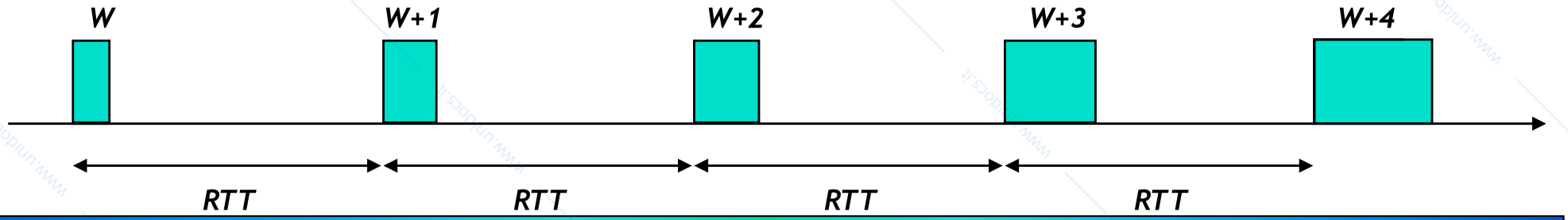


Aggiornamento della Finestra di Congestione

- Vediamo ora come viene aggiornato il valore della finestra *Cwnd* dal trasmettitore TCP in queste due fasi
  - *Slow start*
  - *Congestion avoidance*

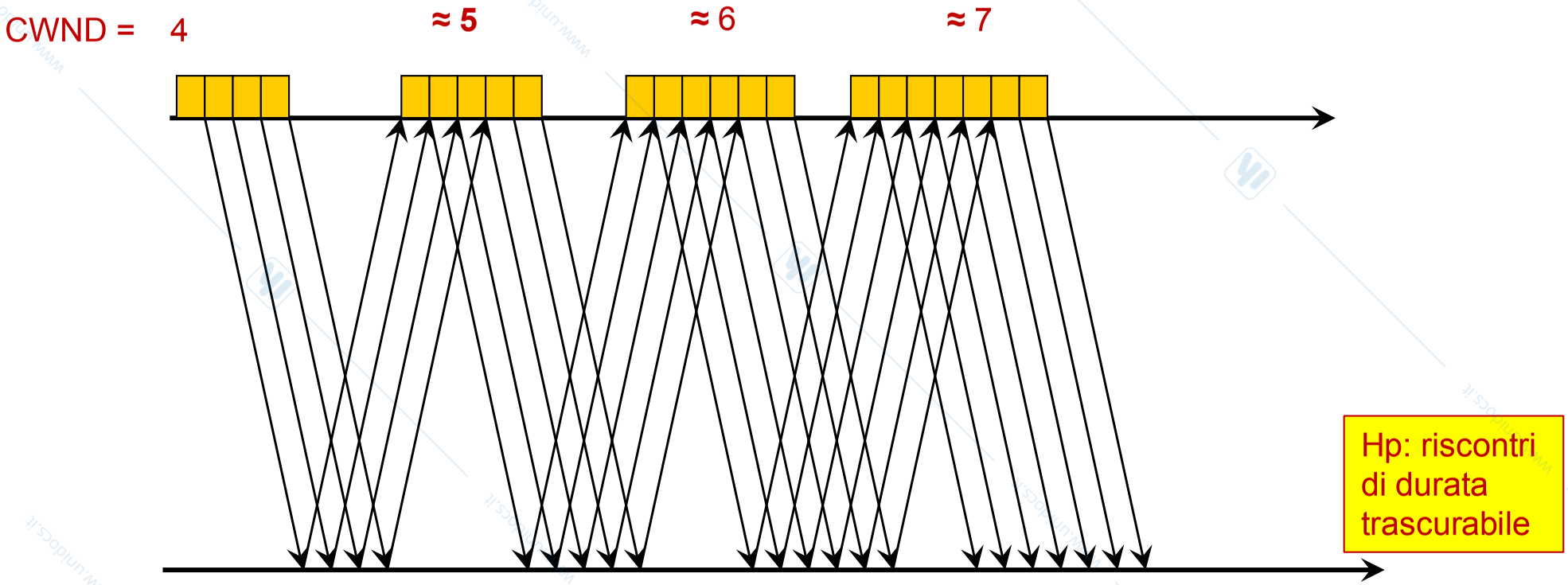
Congestion avoidance

- Slow start continua fino a che *Cwnd* non eguaglia *Ssthresh* e poi parte la fase di *Congestion avoidance*
- *Regola di incremento della Cwnd in Congestion Avoidance:*
  - *la Cwnd viene incrementata di 1/Cwnd ad ogni ACK ricevuto*
  - In altre parole, in Congestion avoidance si attua un ***incremento lineare*** della finestra di congestione



Congestion avoidance

- Dopo aver raggiunto *Ssthresh* la finestra continua ad aumentare ma molto più lentamente



Evento di congestione

- Se avviene congestione, ovvero se scade un timeout (riscontri ritardati, persi o mai trasmessi), sia in Slow start che in Congestion avoidance
  - la finestra non cresce più nel momento in cui smetto di ricevere i riscontri
  - agisco sulla *Ssthresh* e sulla *Cwnd* come mostrato nelle slide seguenti

Evento di congestione

- Scade un timeout di ritrasmissione
  1. il TCP reagisce ponendo *Ssthresh* uguale alla metà dei “byte in volo” (byte trasmessi ma non riscontrati); più precisamente

$$Ssthresh = \max\left(2MSS, \frac{\text{FlightSize}}{2}\right)$$

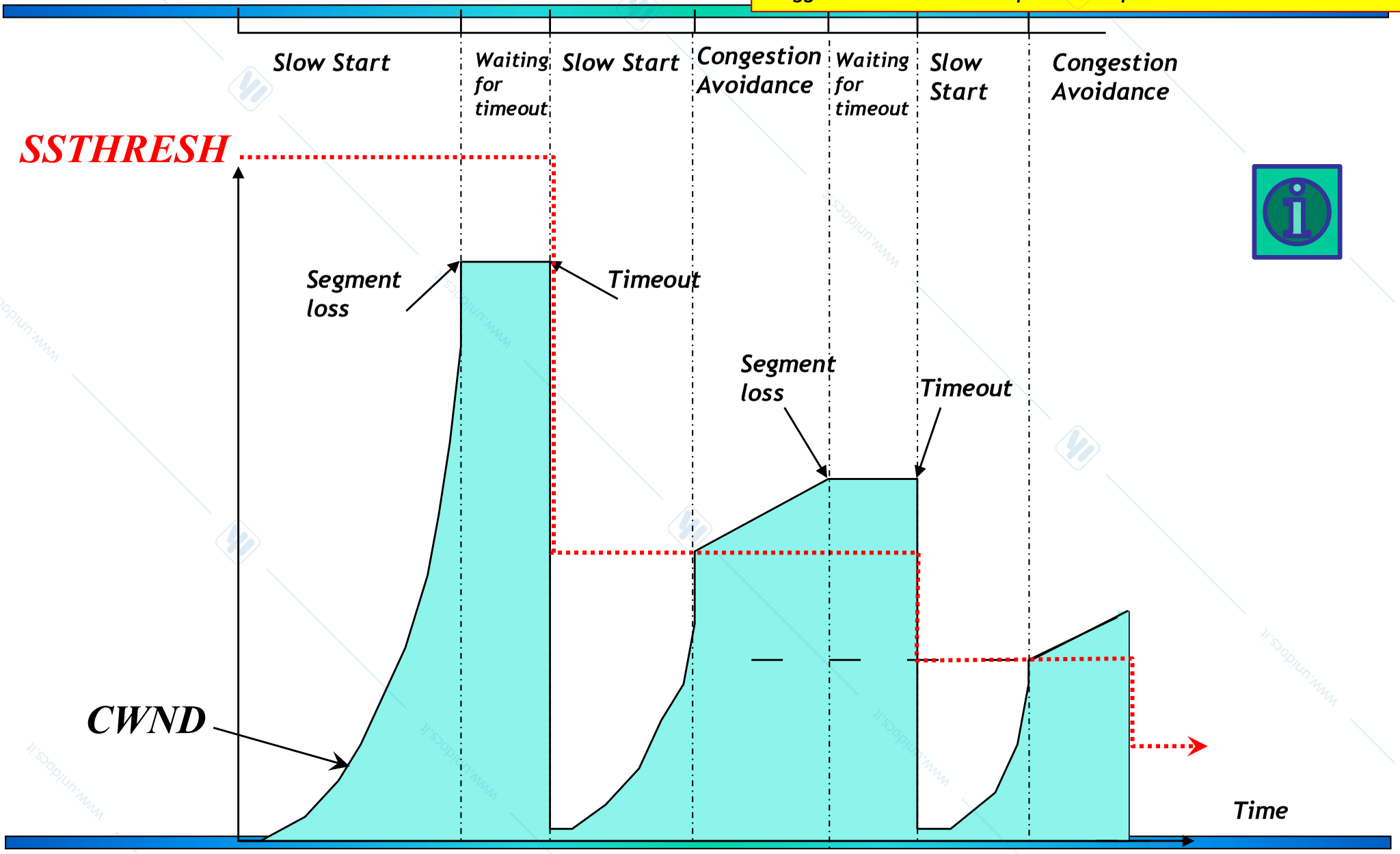
  2. e ponendo *Cwnd* a 1 (TCP «flavor» Tahoe)

Evento di congestione

- Osservazione 1
  - TCP base (Tahoe): il trasmettitore ritrasmette tutti i segmenti successivi già trasmessi (il ricevitore li ha eliminati → scadranno i timeout loro associati). Analogo a GBN
  - TCP avanzato (Reno): il trasmettitore invia nuovi segmenti (il ricevitore può risequenziare quelli già ricevuti → invia ACK cumulativo con “delayed ACK”). Analogo a SR
- Osservazione 2
  - Il valore a cui è posta la *Ssthresh* corrisponde ad una stima (conservativa) della finestra ottimale che eviterebbe futuri eventi di congestione

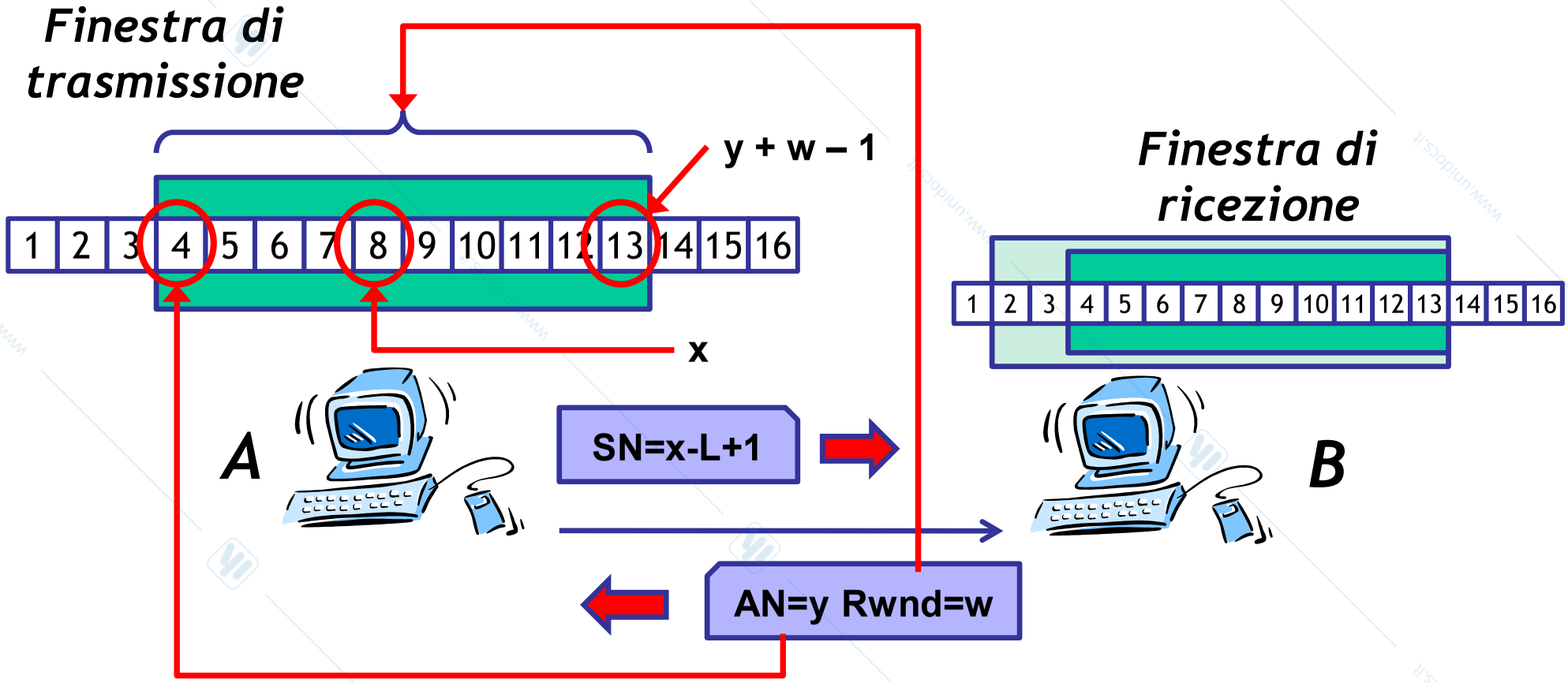
Esempio di funzionamen

Attenzione a non pensare che SSTHR sia destinata solo a ridursi. Se la trasmissione continua per molto tempo senza eventi di congestione, CWND cresce. Quindi, al momento del successivo evento di congestione il nr dei pacchetti in volo sarà alto (paragonabile a Cwnd), pertanto il nuovo valore di SSTHR sarà riaggiornato ad un valore più alto di prima

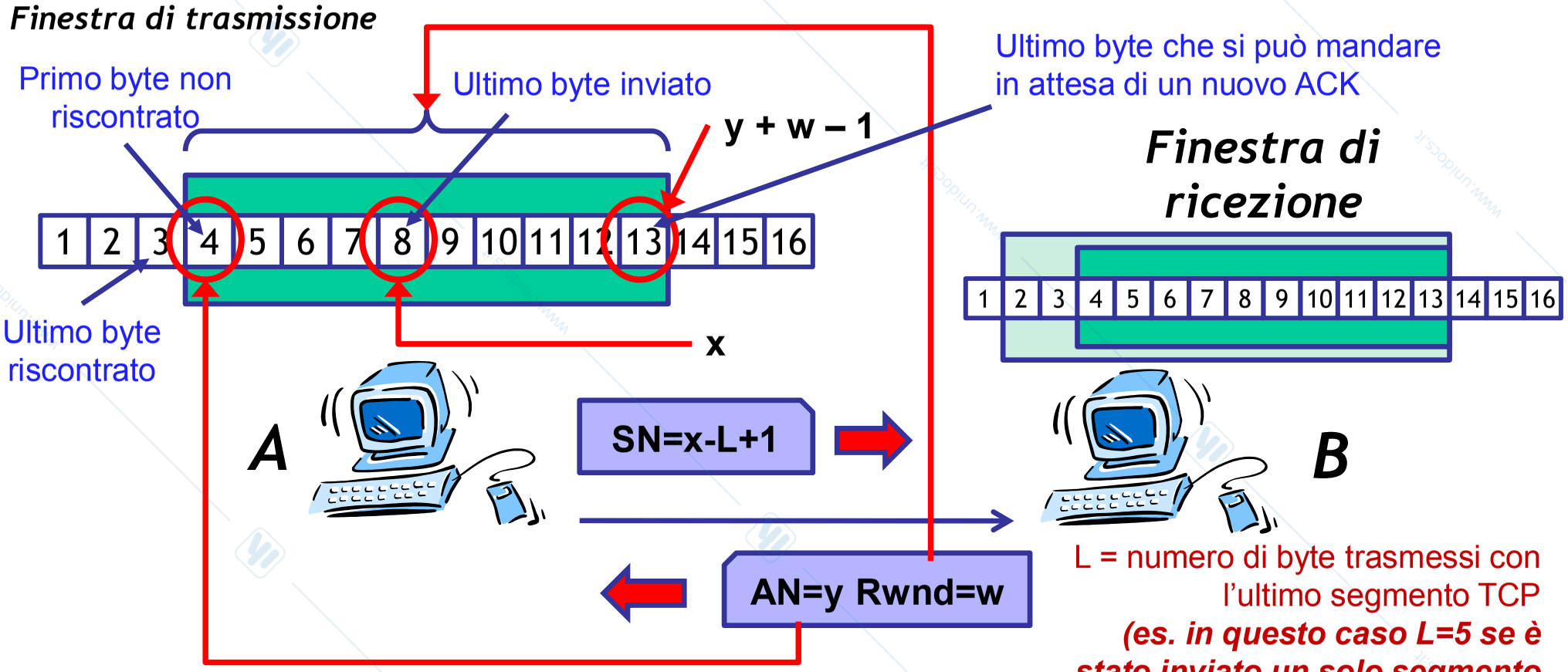


FINE

Finestra di *trasmissione* e campi TCP



Finestra di *trasmissione* e campi TCP



L = numero di byte trasmessi con l'ultimo segmento TCP
(es. in questo caso L=5 se è stato inviato un solo segmento con i byte da 4 a 8)

I segmenti che contengono i byte da y a x vengono copiati in un altro buffer presente al trasmettitore (buffer delle ritrasmissioni). Quando un segmento viene riscontrato, i relativi byte vengono eliminati dal buffer delle ritrasmissioni. Se scade il timeout di ritrasmissione, si ritrasmette il primo segmento presente nel buffer delle ritrasmissioni. Dunque, le ritrasmissioni non hanno alcun effetto sul pointer x (RFC 379)