# RETI DI CALCOLATORI

Autunno 2018

docente: Laura Ricci

laura.ricci@unipi.it

# Lezione 11:
# TCP: THREE WAY HANDSHAKE
# FLOW CONTROL
# 12/11/2018

parte di queste slides sono ricavati da slides pubblicate in corsi universitari tenuti da colleghi italiani e stranieri

# MATERIALE DIDATTICO

- Fourozan
  - paragrafo 3.4.4
  - paragrafo 3.4.5
  - paragrafo 3.4.6
  - paragrafo 3.4.7

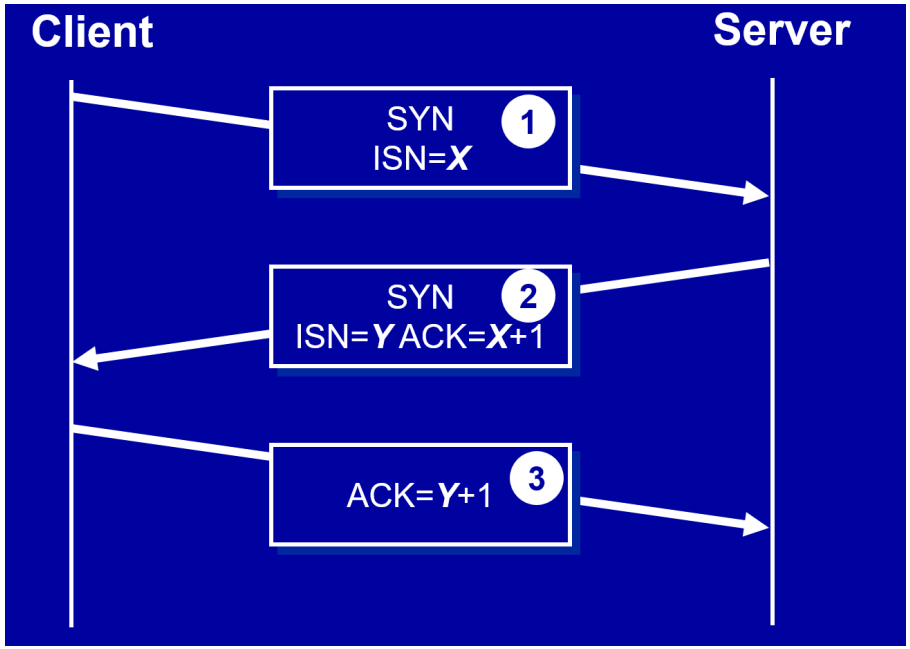
# CONNESSIONI TCP

- il protocollo TCP è un protocollo del tipo con connessione:
  - mittente e destinatario aprono una "connessione" prima di scambiarsi qualsiasi tipo di dati
- In questa lezione analizziamo:
  - apertura di una connessione TCP
  - chiusura della connessione
  - scenari speciali
  - diagramma degli stati

# TCP: CONNESSIONE

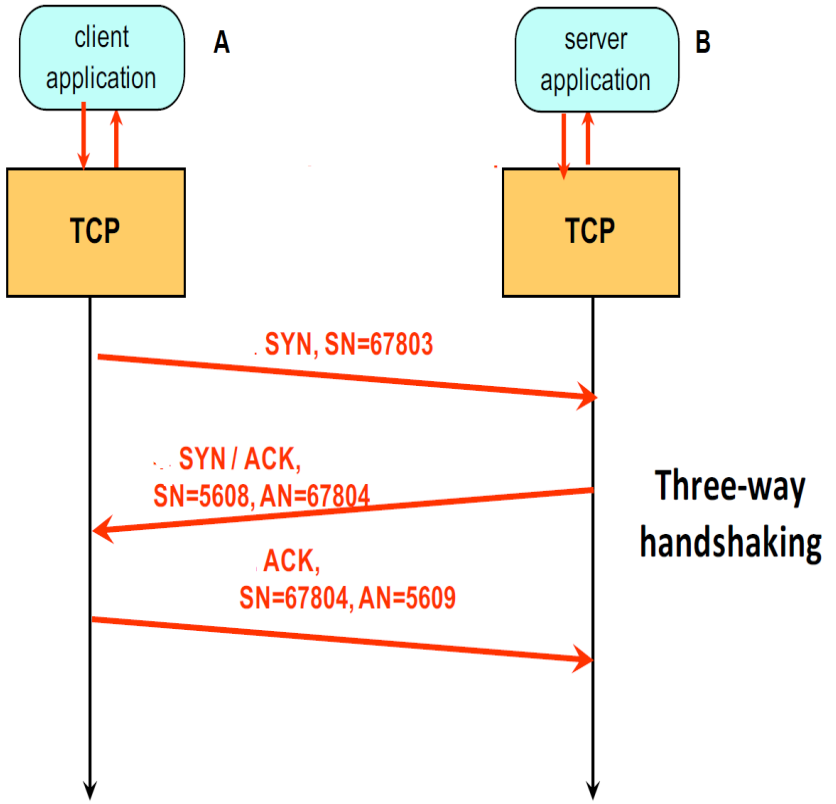
- La fase di apertura di una connessione è asimmetrica:
  - uno dei due nodi è in stato listen (server)
  - l'altro, invece, è attivo nel richiedere la apertura della connessione
- fase di connessione:
  - client "Voglio connettermi, e il mio numero di sequenza attuale è X"
  - server "OK, sono disponibile. L'identificatore del mio numero di sequenza attuale è Y.

    So che l'identificatore del primo byte che invierai sarà X+1
  - client OK, "so che l'identificatore del primo byte che mi invierai è Y+1"



# THREE WAY HANDSHAKE

- passo 1: l'host A invia un segmento SYN all'host B in cui indica:
  - il suo numero di sequenza iniziale (SN)
  - altri parametri della connessione (MSS, dimensione finestra)
  - nessun dato in questo segmento
- passo 2: l'host B riceve SYN
  - alloca i buffer
  - Riscontra con il segmento SYN ACK ilil segmento ricevuto ed invia proprio numero di sequenza iniziale
- passo 3: l'host A riceve il riscontro
  - riscontra il riscontro con un ACK,
  - Il segmento inviato può contenere dati

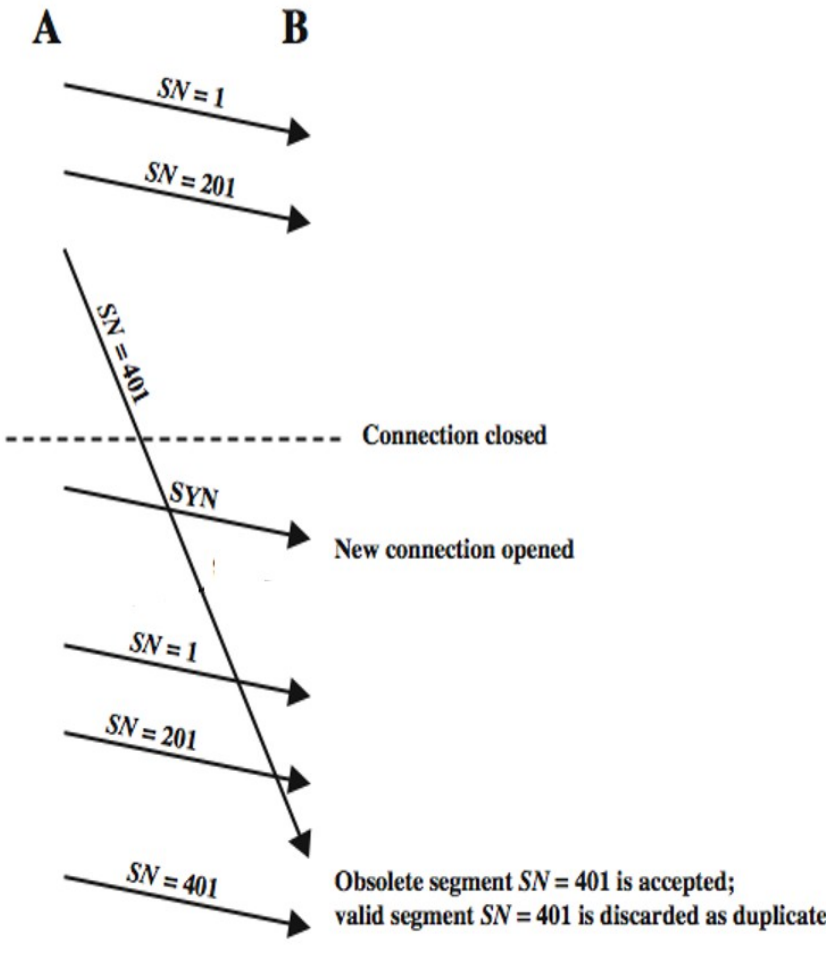


# INITIAL SEQUENCE NUMBER (ISN)

- l'informazione più rilevante trasmessa in fase di handshake:
  - numero di sequenza (ISN)
  - indica il primo byte che verrà trasmesso sulla connessione, dalle due parti
  - scelto come numero casuale tra 0 e $2^{32}$-1 = 4294967295
- perchè questa fase è necessaria? perchè i due host non usano semplicemente ISN=0?
- scelta casuale di ISN consente:
  - riconoscimento dati obsoleti
  - misura contro spoofing TCP
    - ancora possibile consniffing pacchetti IP

# ISN: PACCHETTI OBSOLETI

- la copia duplicata di un segmento spedito da A
  - può provenire da una vecchia connessione
  - arrivare al destinatario durante una nuova connessione
  - il segmento duplicato può arrivare prima di quello con lo stesso numero di sequenza, generato nella nuova connessione
- per diminuire la probabilità di questi casi
  - usare numeri di sequenza diversi per ogni connessione
  - scelta random



# ISN: SPOOFING

- spoofing: forgiare una identità falsa
- attaccante può forgiare un pacchetto SYN con l'indirizzo IP falsificato
  - il pacchetto è inviato al server
  - il server cercha di portare a termine l'handshake rispondendo con un pacchetto SYN/ACK.
  - questo pacchetto non può essere ricevuto dall'attaccante perchè riporta l'indirizzo IP falsificato
  - l'attaccante  deve inviare il secondo pacchetto, senza aver visto il secondo
  - deve forgiare un pacchetto ACK per il server che
    - riporti nuovamente l'indirizzo IP falsificato,
    - riporti il sequence number che il server ha inserito nel pacchetto SYN/ACK.
    - molto improbabile individuare il numero di sequenza se generato casualmente

# THREE WAY HANDSHAKE: AFFIDABILITA'

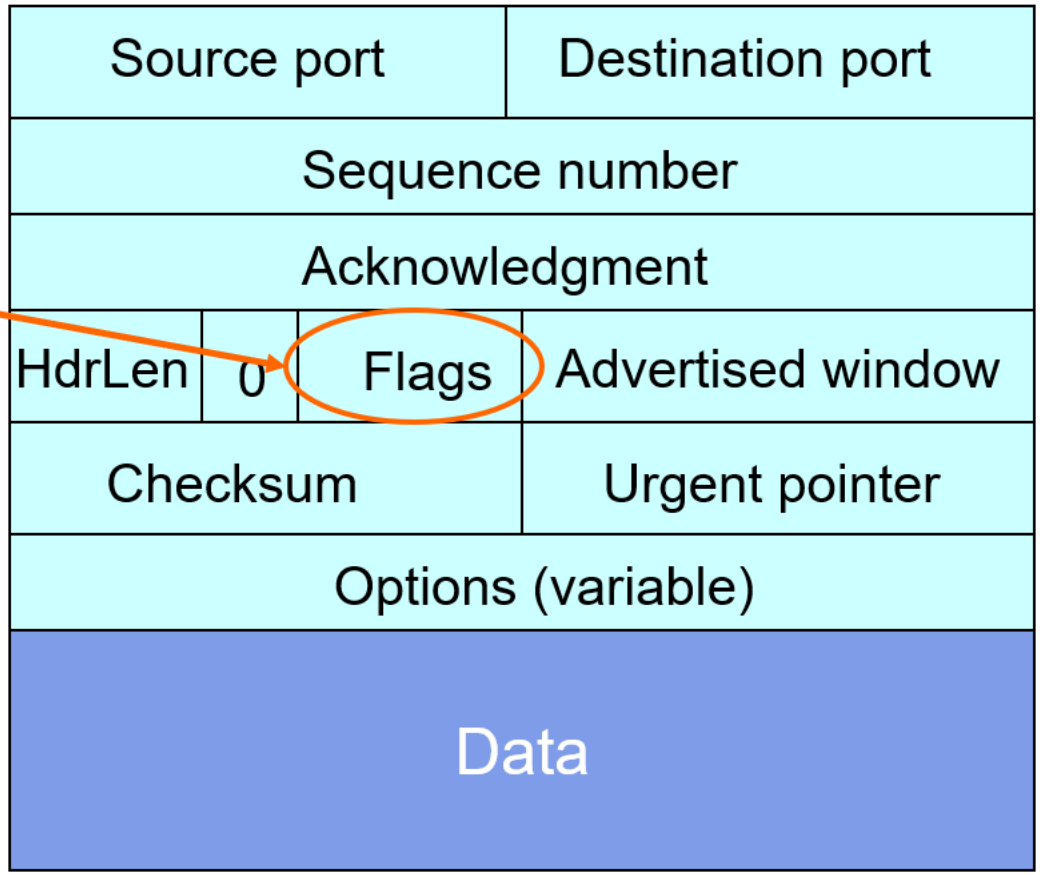
- come per altri aspetti del protocollo, anche nella fase di connection establishment bisogna tenere di conto che la rete IP è best effort
- cosa implica, in questo scenario?
  - A invia SYN a B
    - SYN può essere perso.
    - il riscontro (SYNACK) può essere perso.
- utilizzo di un retransmit-SYN timer
  - reinvio di SYN allo scadere del timer
  - possibilità di SYN duplicati (se è perso il riscontro)
  - nessuna possibilità di conoscere il RTT, perchè la connessione è in fase di apertura
  - in genere settato a 3 secondi
- scartare tutti I SYN pervenuti dopo l'apertura di una comunicazione

# RETRANSMIT SYN E WEB DOWNLOAD

- l'utente clicca su un link ipertestuale
  - Il browser crea una socket e richiede una connessione al server HTTP
  - Il SO (livello TCP) trasmette un SYN
- se il SYN è perso....
  - più di 3 secondi di ritardo: può essere percepito come attesa non accettabile da parte dell'utente
  - ...l'utente può cliccare il link di nuovo, oppure cliccare "reload"
- l'azione dell'utente scatena un "abort" della connessione
  - il browser crea una nuova socket e tenta un'altra connessione
  - in pratica, forza un invio più veloce di un nuovo pacchetto SYN
  - può essere molto efficace, la pagina viene caricata rapidamente

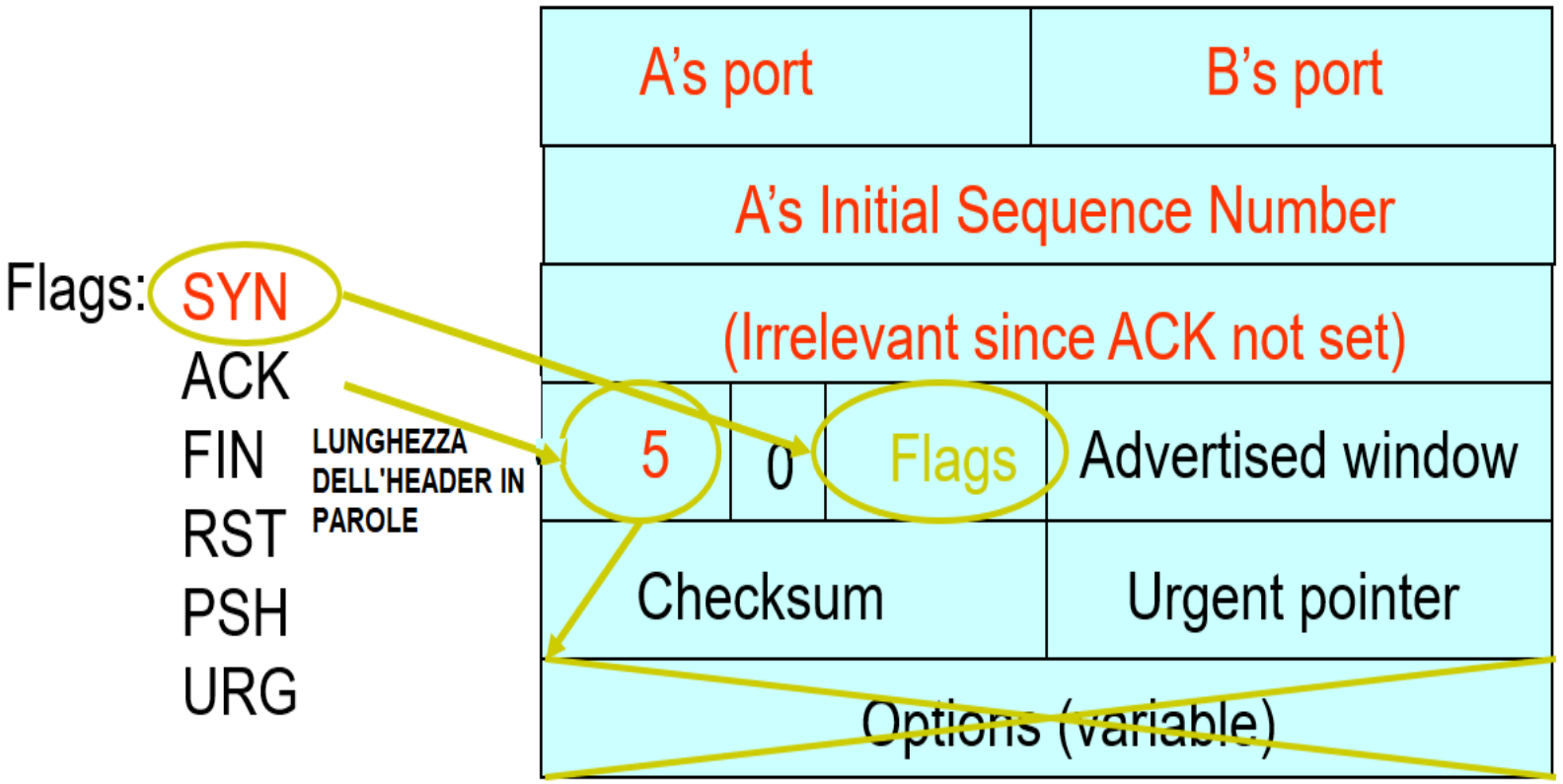
# HEADER: FLAG SYN ED ACK

Flags: **SYN**
**ACK**
FIN
RST
PSH
URG



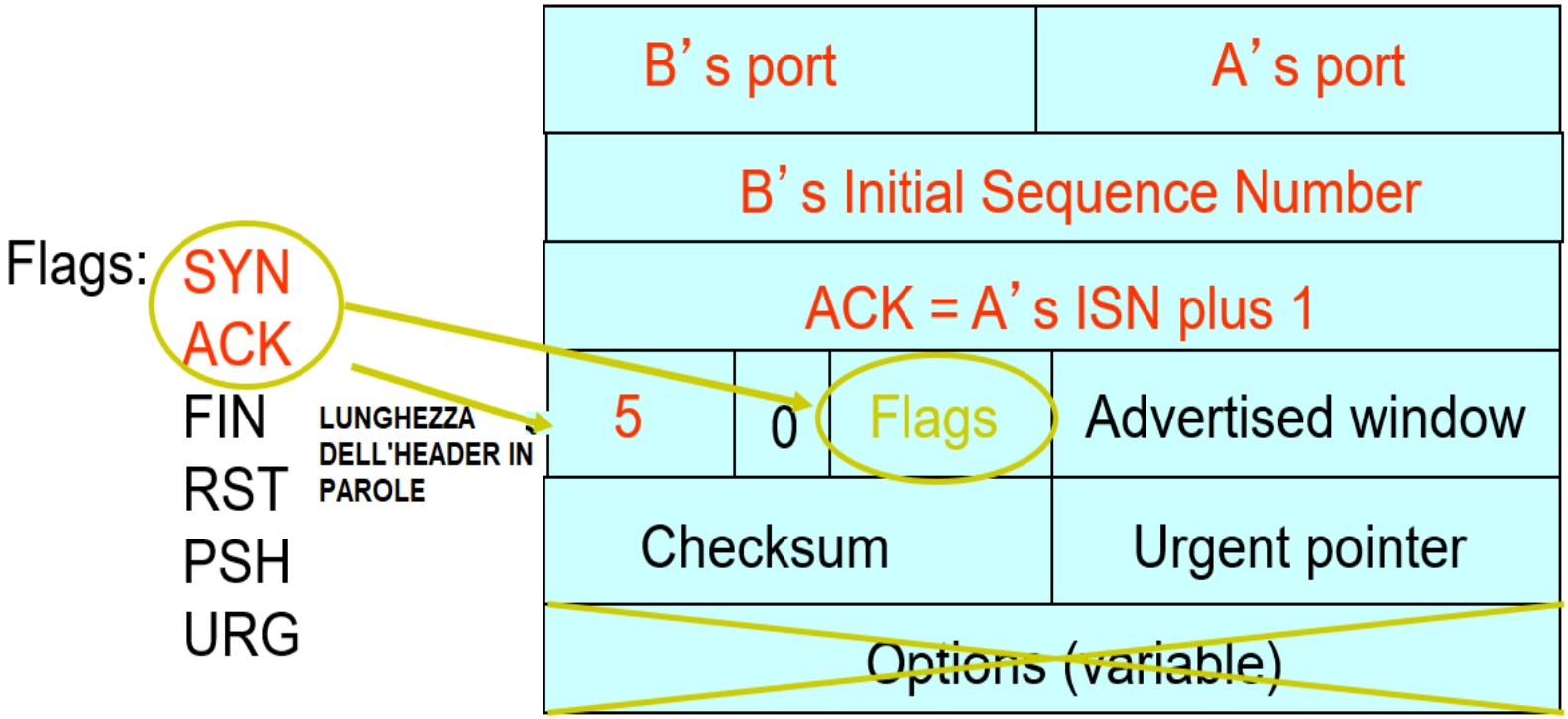

Nelle prossime slide evidenziati in rosso i campi interessati dal three way handshake nelle diverse fasi

# PASSO I: A INVIA IL SYN




A DICE A B CHE VUOLE APRIRE UNA CONNESSIONE

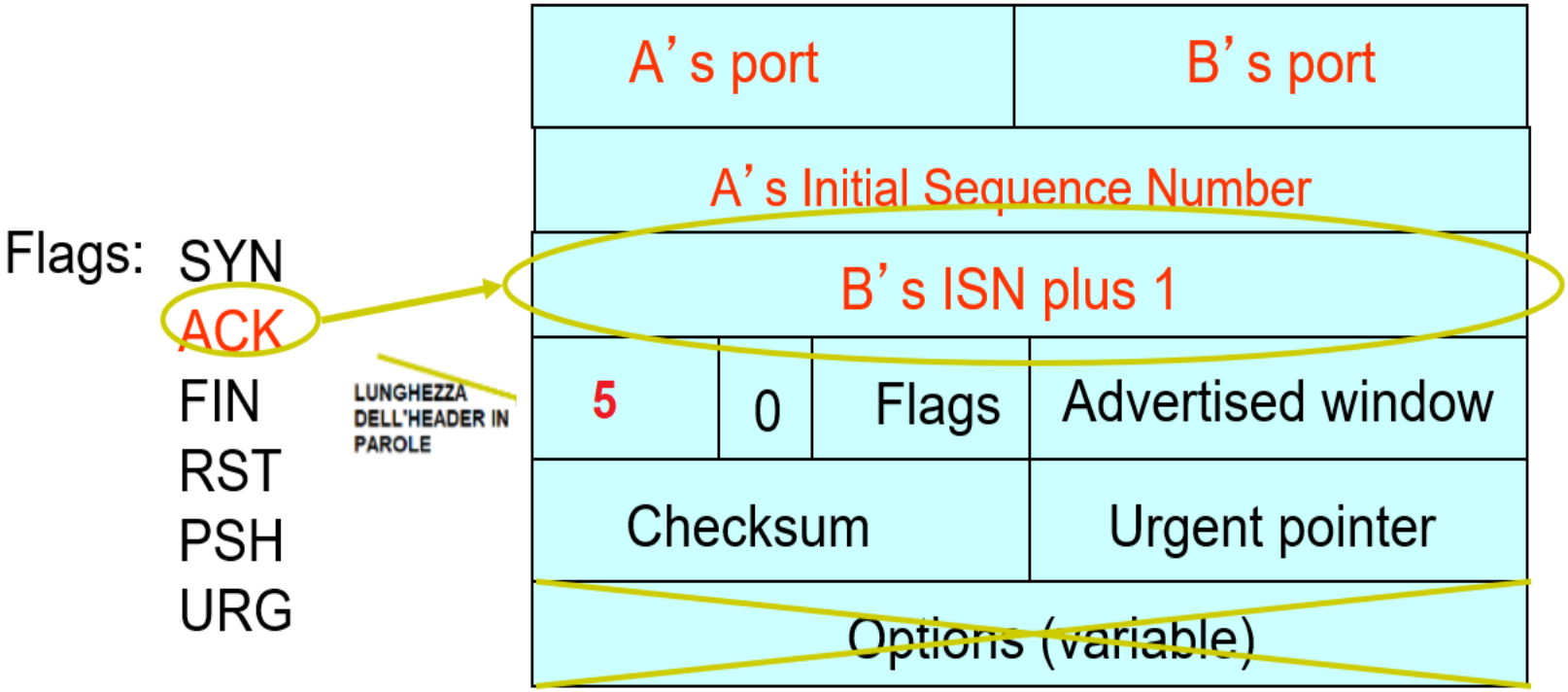
# PASSO 2: B INVIA IL SYN-ACK




B DICE AD A CHE ACCETTA LA CONNESSIONE ED E' PRONTO AD

ACCETTARE IL PROSSIMO BYTE

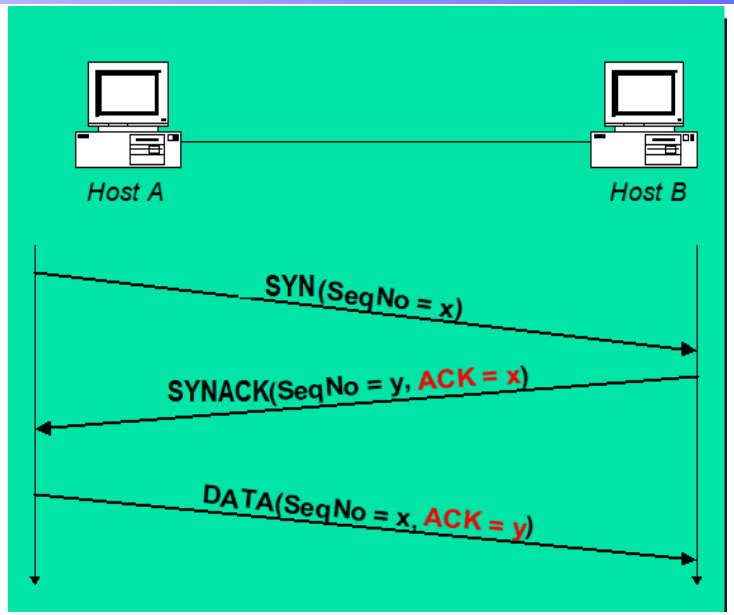
# PASSO 3: ACK DEL SYN-ACK SPEDITO DA A




A DICE A B CHE PUO' INIZIARE A SPEDIRE PACCHETTI

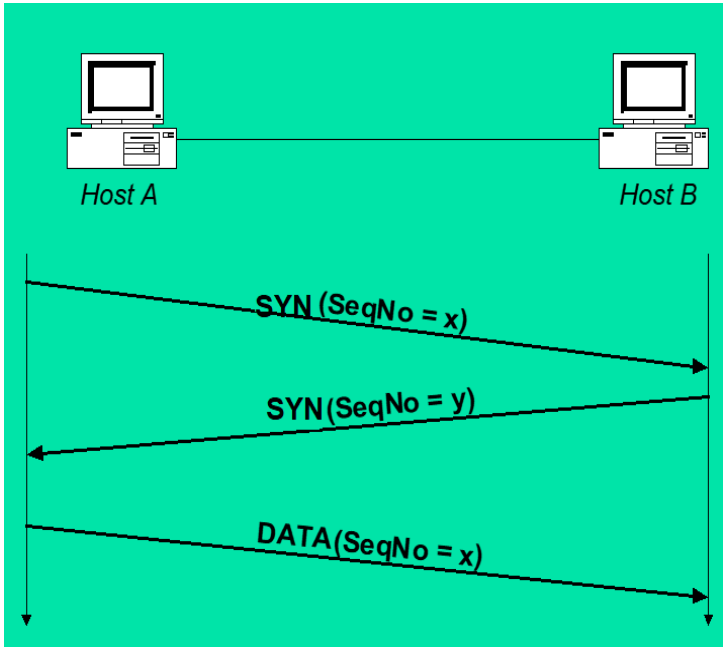
....DOPO AVER RICEVUTO IL PACCHETTO, B PUO' INIZIARE A SPEDIRE

# PERCHE' THREE-WAY HANDSHAKE?

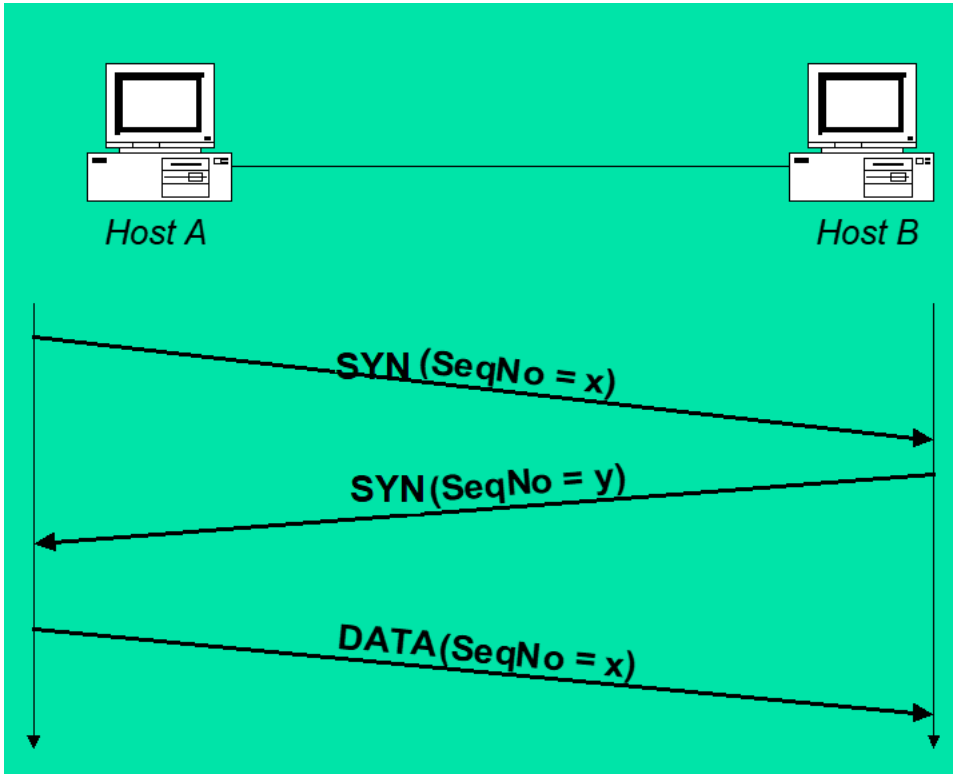



- immaginiamo un two-way handshake
  - i due nodi si scambiano i numeri di sequenza
  - non riscontrano i messaggi ricevuti
  - iniziano la trasmissione usando gli ISN comunicati
  - questa soluzione non è corretta, perchè?

- sia A che B riscontrano il numero di sequenza inviato dall'altro nodo
  - perchè il terzo ack?
  - non è sufficiente un three-way-handshake?

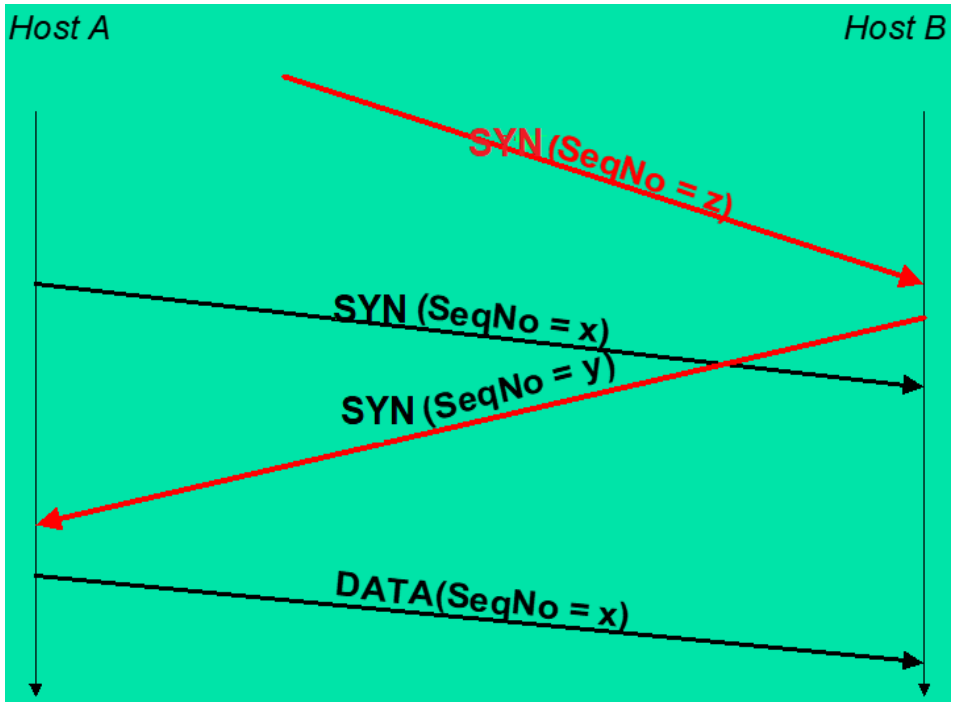


# TWO-WAY HANDSHAKE




- **SYN (SeqNo = x)**
  - A vuole iniziare una richiesta di connessione con SeqNo = x
- **SYN (SeqNo = y)**
  - indica che, B inizierà la connessione con numero di sequenza SeqNo = y
- **DATA (SeqNo = x)**
  - data transmission con SeqNo= x
- nessun riscontro dei messaggi SYN ricevuti

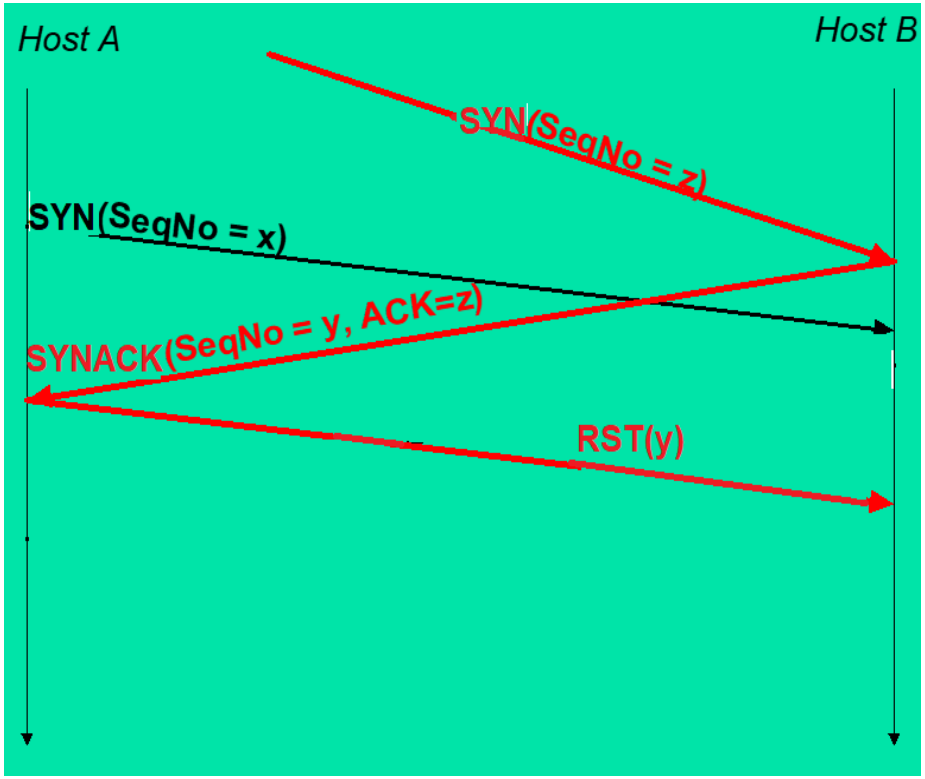
# TWO-WAY HANDSHAKE: SYN DUPLICATI




- SYN possono essere duplicati
  - SYN(SeqNo = z) proviene da una vecchia connessione
- B risponde al vecchio SYN, inviando il

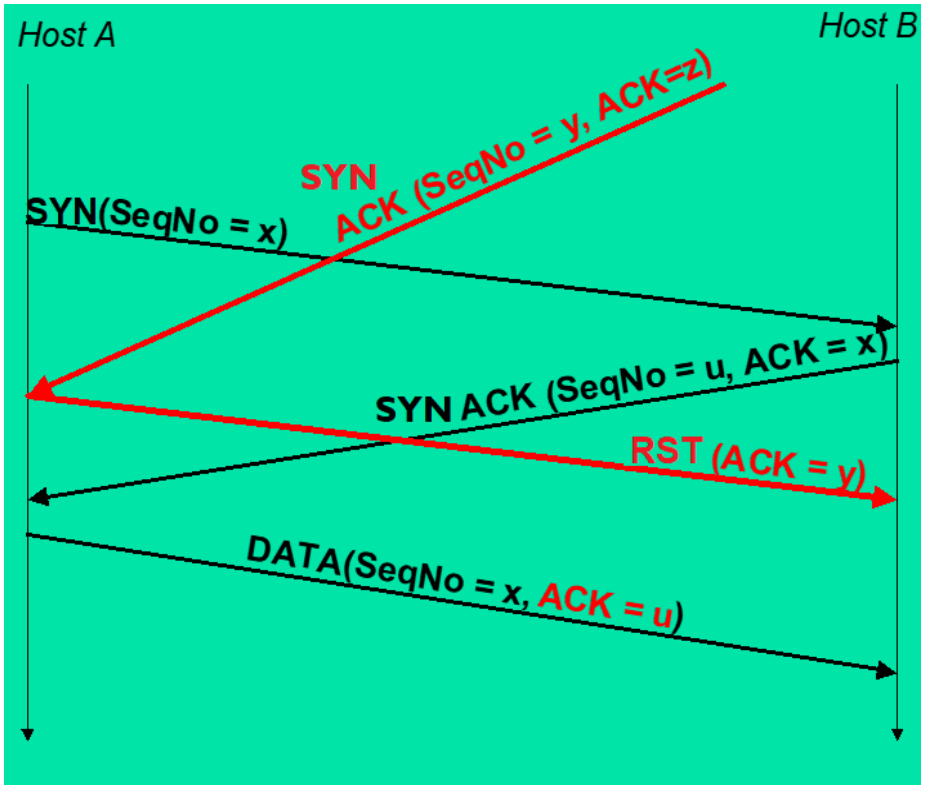
  suo ISN
- A riceve il messaggio (SeqNo = y) e crede che sia una risposta alla sua richiesta di connessione
- risultato:
  - A inizia ad inviare dati con SeqNo=x
  - B scarta i dati, perchè che si aspetta un numero di segmento SeqNo = z

# THREE-WAY HANDSHAKE: SYN DUPLICATI




- Il three-way handshake risolve il problema degli ACK duplicati
- A rifiuta il SYNACK che riscontra il segmento con ISN=z
  - Il SYNACK non si riferisce all'ultimo segmento inviato
  - si aspetta x, arriva z
- A invia un pacchetto di reset (RST) della connessione
- successivamente, quando il time out scade, A potrà riniviare la richiesta per la nuova connessione (seqNo=x)

# TWO-WAY HANDSHAKE: ACK DUPLICATI




- l'host A
  - rifiuta il SYNACK invalido, inviando un messaggio di reset (RST)
  - riscontra il SYNACK valido
- la richiesta di connessione (SeqNo=x) viene accettata con successo
- esempio analogamente si può fare riguardo all'ultimo aCK
- conclusione:
  - i riscontri di entrambe i SYN sono fondamentali come protezione da segmenti obsoleti

# TCP: MAXIMUM SEGMENT SIZE

- Quando si stabilisce una connessione TCP tra due host, essi si scambiano il valore di MSS (Maximum Segment Size)
- il TCP che manda il SYN annuncia MSS,
  - il valore più piccolo viene usato come dimensione massima del segmento per quella connessione
  - inserito nel campo Options dell'header in fase di apertura di una connessione
  - se MSS non è presente nell'header
    - il mittente usa l'MTU della LAN locale
    - valore di default

      536 bytes= 576 (min.dim.pacchetto)-20B(IP Header)-20 (TCPHeader)
- scopo: tentativo di limitare la fragmentazione.
- come si determina MSS
  - se gli host sono collegati alla stessa rete locale si può derivarlo dall'MTU

# TCP: MAXIMUM SEGMENT SIZE

- ogni endpoint mantiene anche un Path MTU (PMTU)
  - rappresenta il minimo MTU lungo un cammino verso una certa destinazione
- si invia un pacchetto con Don't Fragment (DF) flag ad 1 nell' IP header.
  - vieta la frammentazione del pacchetto sul cammino tra gli host
- se un link intermedio ha un MTU più piccolo della dimensione del pacchetto, il router informa l'host sorgente
  - invia un messaggio ICMP che indica che il segmento è troppo grande per essere spedito su quel tratto di rete
  - quando il TCP dell'host mittente riceve il messaggio ICMP, il TCP può definire il MSS usando l'MTU specificato nel messaggio ICMP
  - processo iterativo, necessari più probe.

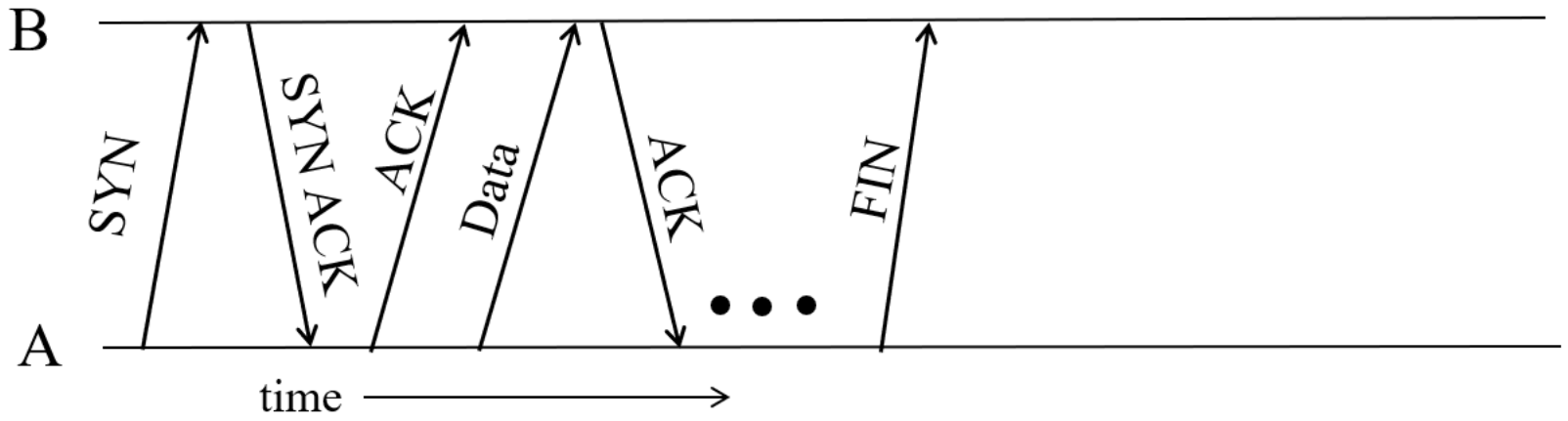
# TERMINAZIONE DELLA CONNESSIONE

- poiché la connessione è bidirezionale, la terminazione deve avvenire in entrambe le direzioni

- "gentle termination":
  - l'host che non ha più dati da trasmettere e decide di chiudere la connessione invia un segmento FIN
    - segmento con il campo FIN posto a 1 e campo dati vuoto
  - la stazione che riceve il segmento FIN lo riscontra e indica all'applicazione che la comunicazione è stata chiusa nella direzione entrante

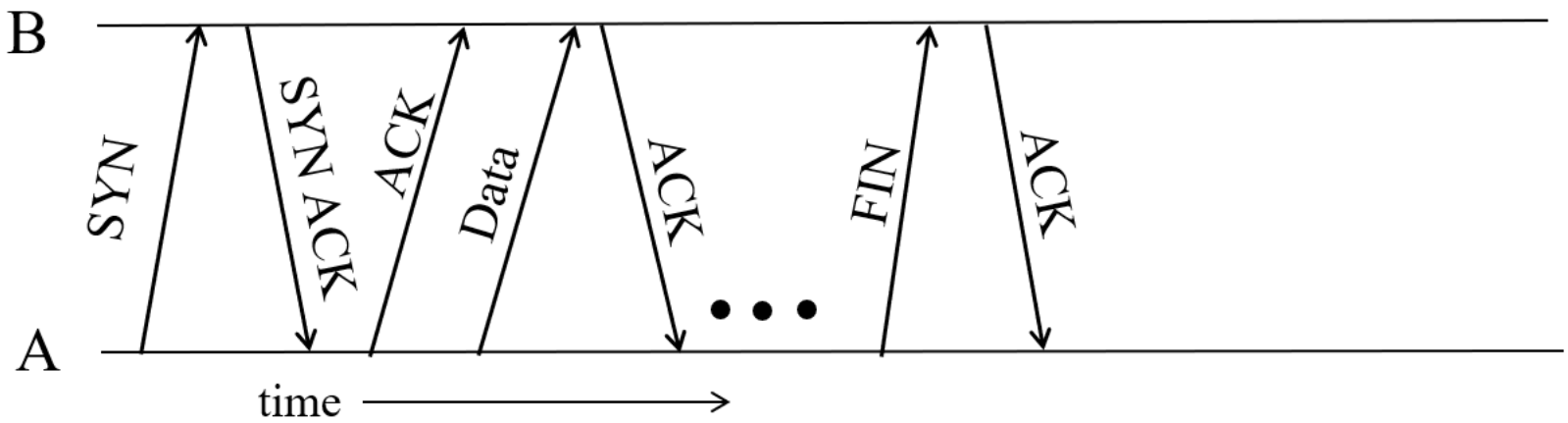
- se questa procedura avviene solo in una direzione (half close), nell'altra il trasferimento dati può continuare
  - gli ACK spediti dall'host che ha chiuso la connessione non sono considerati come traffico originato, ma come risposta al traffico originato dall'altro end host
  - per chiudere completamente la connessione, la procedura di half close deve avvenire anche nell'altra direzione

# CHIUSURA REGOLARE: HALF CLOSE



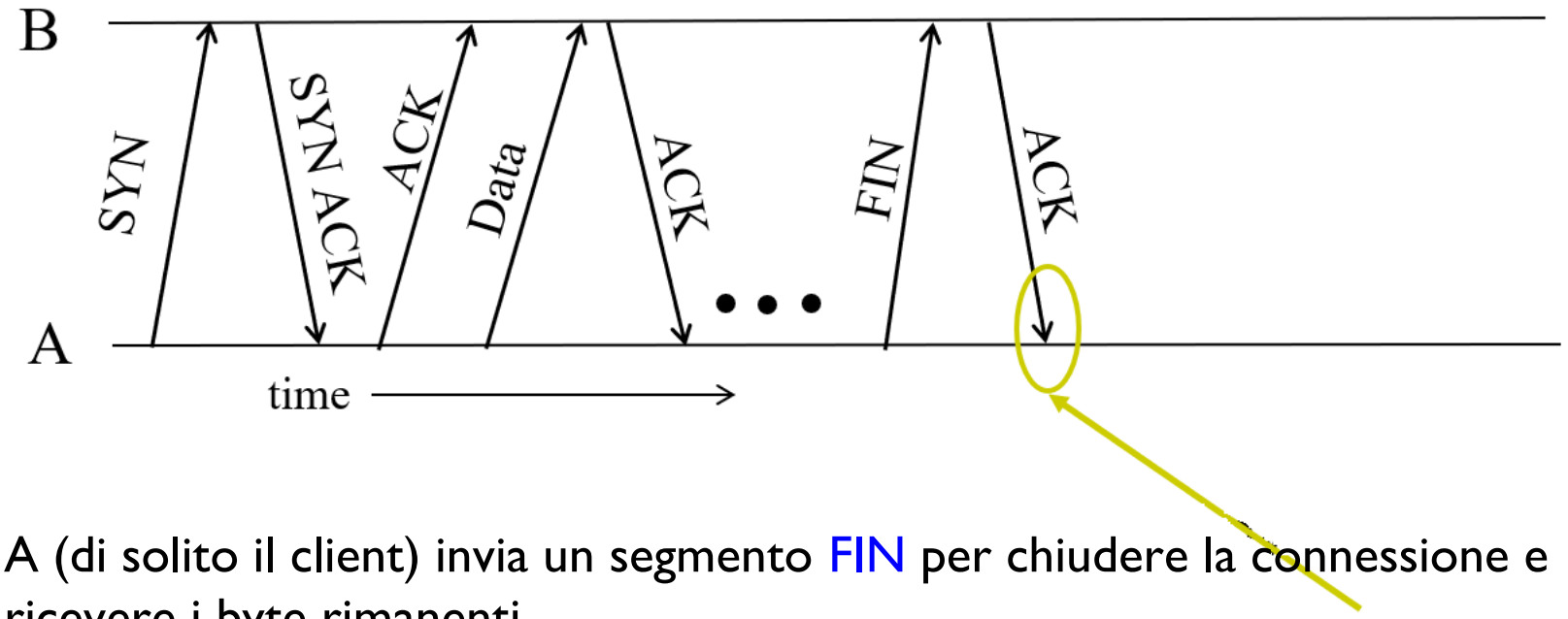

- A (di solito il client) invia un segmento FIN per chiudere la connessione e ricevere i byte rimanenti
  - flag FIN=1 nell'header
  - Il lato TCP che chiude la connessione(A) invia, insieme al FIN, gli ultimi dati

# CHIUSURA REGOLARE: HALF CLOSE



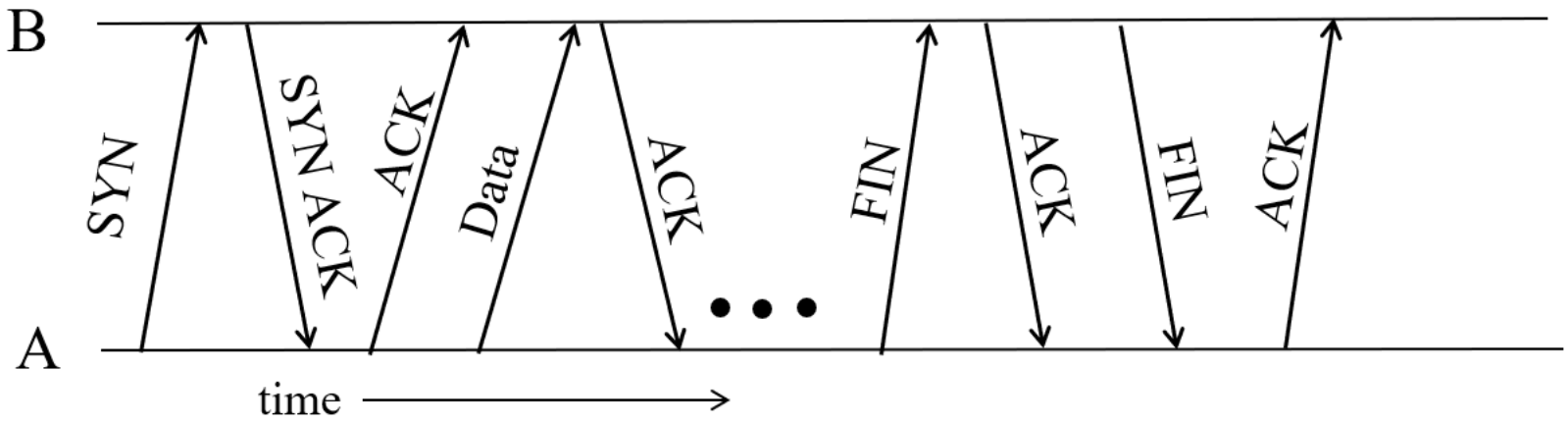

- A (di solito il client) invia un segmento FIN per chiudere la connessione e ricevere i byte rimanenti
  - flag FIN=1 nell'header
  - Il lato TCP che chiude la connessione(A) invia, insieme al FIN, gli ultimi dati
  - l'altro host riscontra il FIN

# CHIUSURA REGOLARE: HALF CLOSE



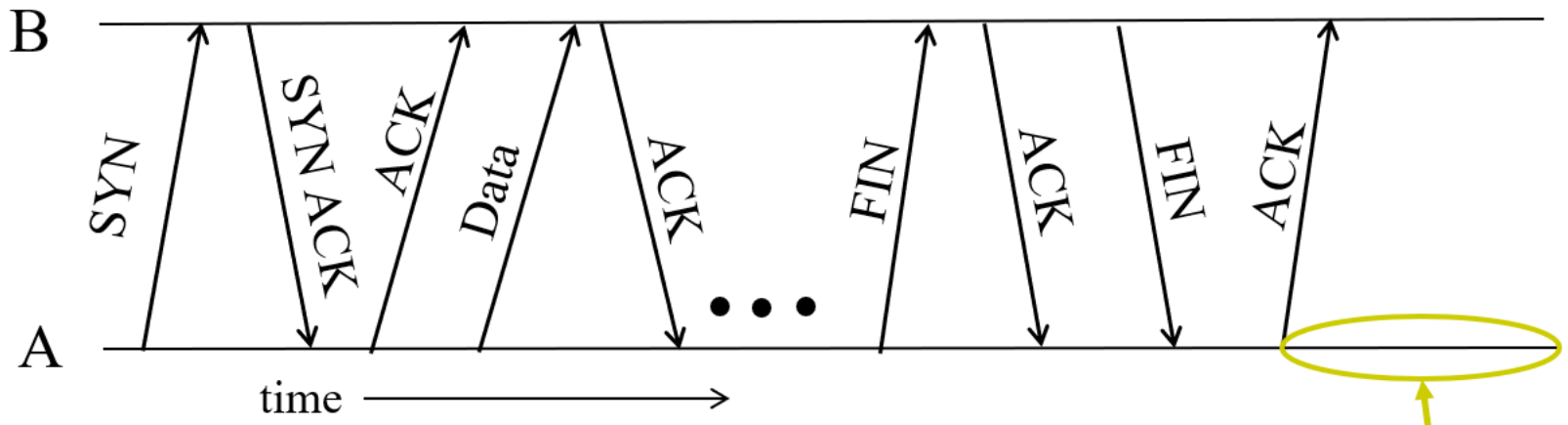

- A (di solito il client) invia un segmento FIN per chiudere la connessione e ricevere i byte rimanenti
  - flag FIN=1 nell'header
  - Il lato TCP che chiude la connessione(A) invia, insieme al FIN, gli ultimi dati
  - l'altro host riscontra il FIN
  - half closure:
    - questo chiude il collegamento da A a B, ma non il viceversa

# CHIUSURA REGOLARE: HALF CLOSE




- A (di solito il client) invia un segmento FIN per chiudere la connessione e ricevere i byte rimanenti
  - flag FIN=1 nell'header
  - Il lato TCP che chiude la connessione (A) invia, insieme al FIN, gli ultimi dati
  - l'altro host riscontra il FIN
  - half closure:
    - questo chiude il collegamento da A a B, ma non il viceversa
  - alla fine B chiude, a sua volta, il collegamento
    - che viene riscontrato da A

# CHIUSURA REGOLARE: HALF CLOSE



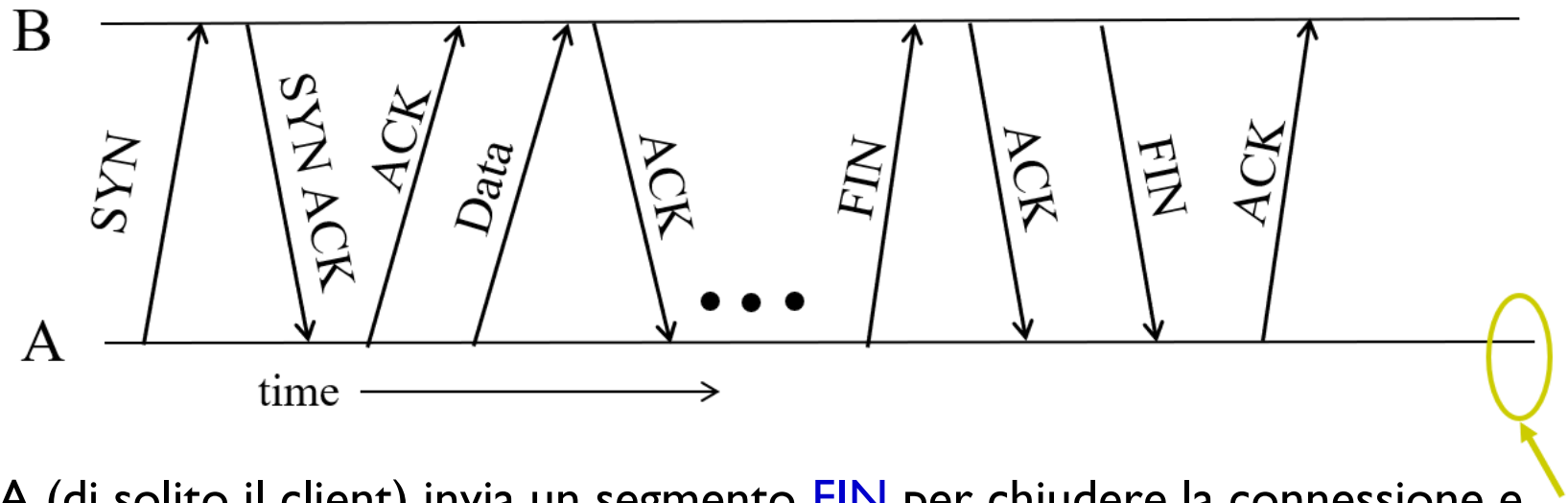

- A (di solito il client) invia un segmento FIN per chiudere la connessione e ricevere i byte rimanenti
  - flag FIN=1 nell'header
  - Il lato TCP che chiude la connessione invia, insieme al FIN, gli ultimi dati
  - l'altro host riscontra il FIN
  - half closure:
    - questo chiude il collegamento da A a B, ma non il viceversa
    - fino a che B chiude a sua volta il collegamento
  - alla fine B chiude, a sua volta, il collegamento
    - che viene riscontrato da A

**TIME_WAIT**:

Avoid reincarnation

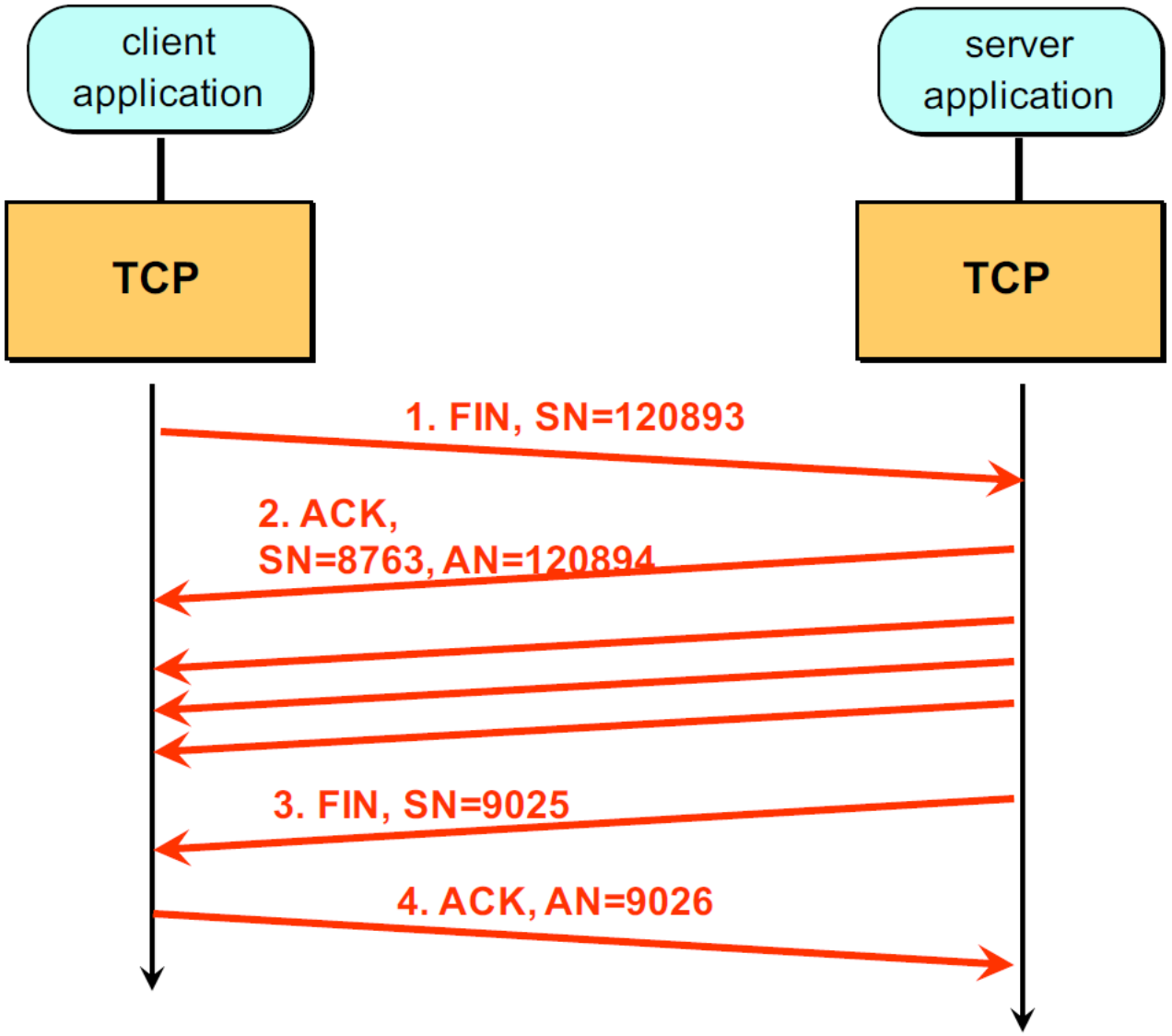
B will retransmit FIN if ACK is lost

# CHIUSURA REGOLARE: HALF CLOSE

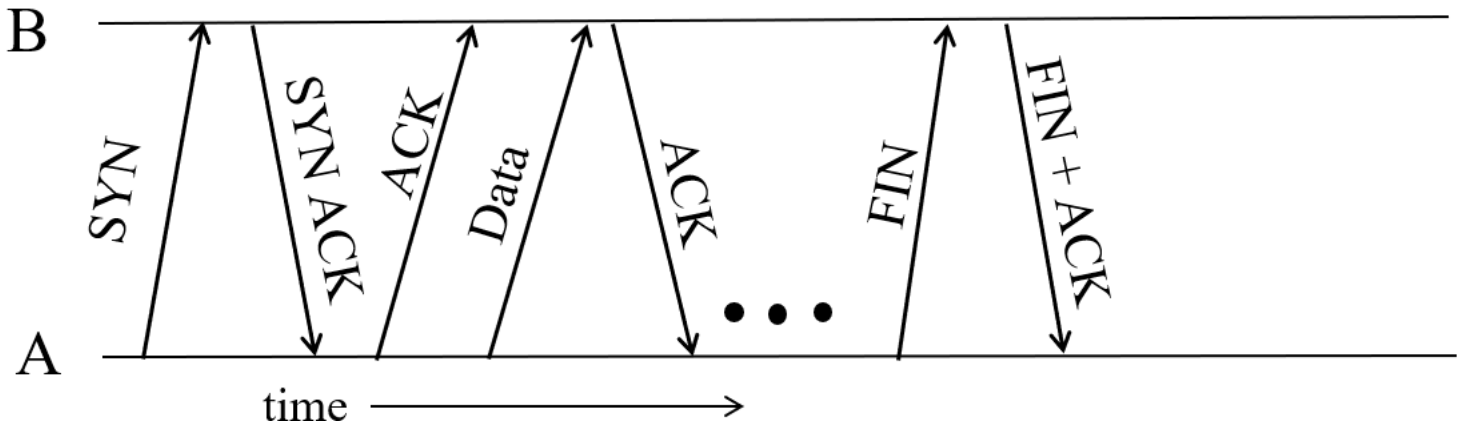



- A (di solito il client) invia un segmento FIN per chiudere la connessione e ricevere i byte rimanenti
  - flag FIN=1 nell'header
  - Il lato TCP che chiude la connessione(A) invia, insieme al FIN, gli ultimi dati
  - l'altro host riscontra il FIN
  - half closure:
    - questo chiude il collegamento da A a B, ma non il viceversa
    - fino a che B chiude a sua volta il collegamento
    - che viene riscontrato da A
  - segue una fase di TIME_WAIT

# HALF CLOSE: RIASSUNTO

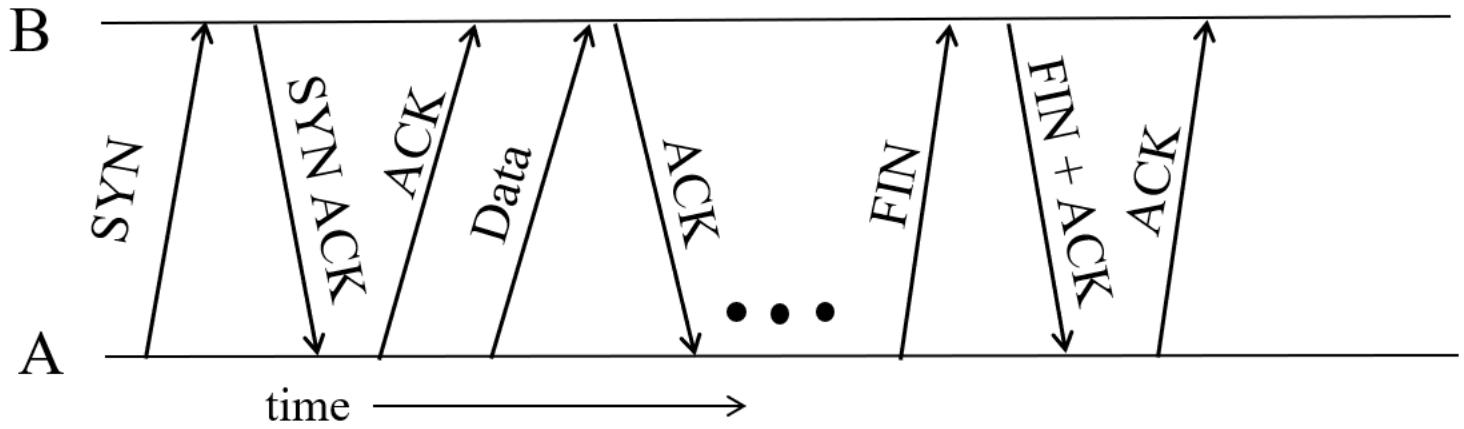


# CHIUSURA REGOLARE: ENTRAMBE I NODI



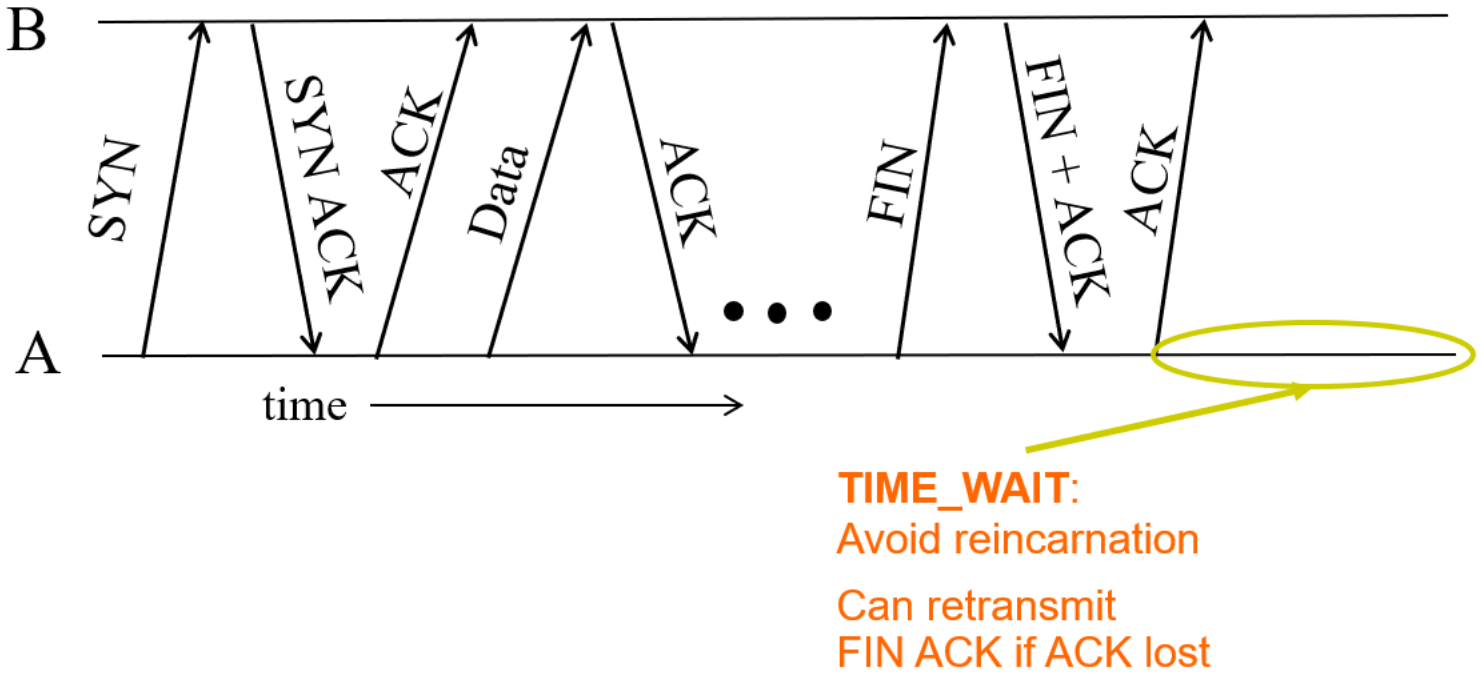

- i nodi decidono contemporaneamente la connessione
- come in precedenza, ma B setta il FIN nel segmento di ACK al FIN di A

# CHIUSURA REGOLARE: ENTRAMBE I NODI



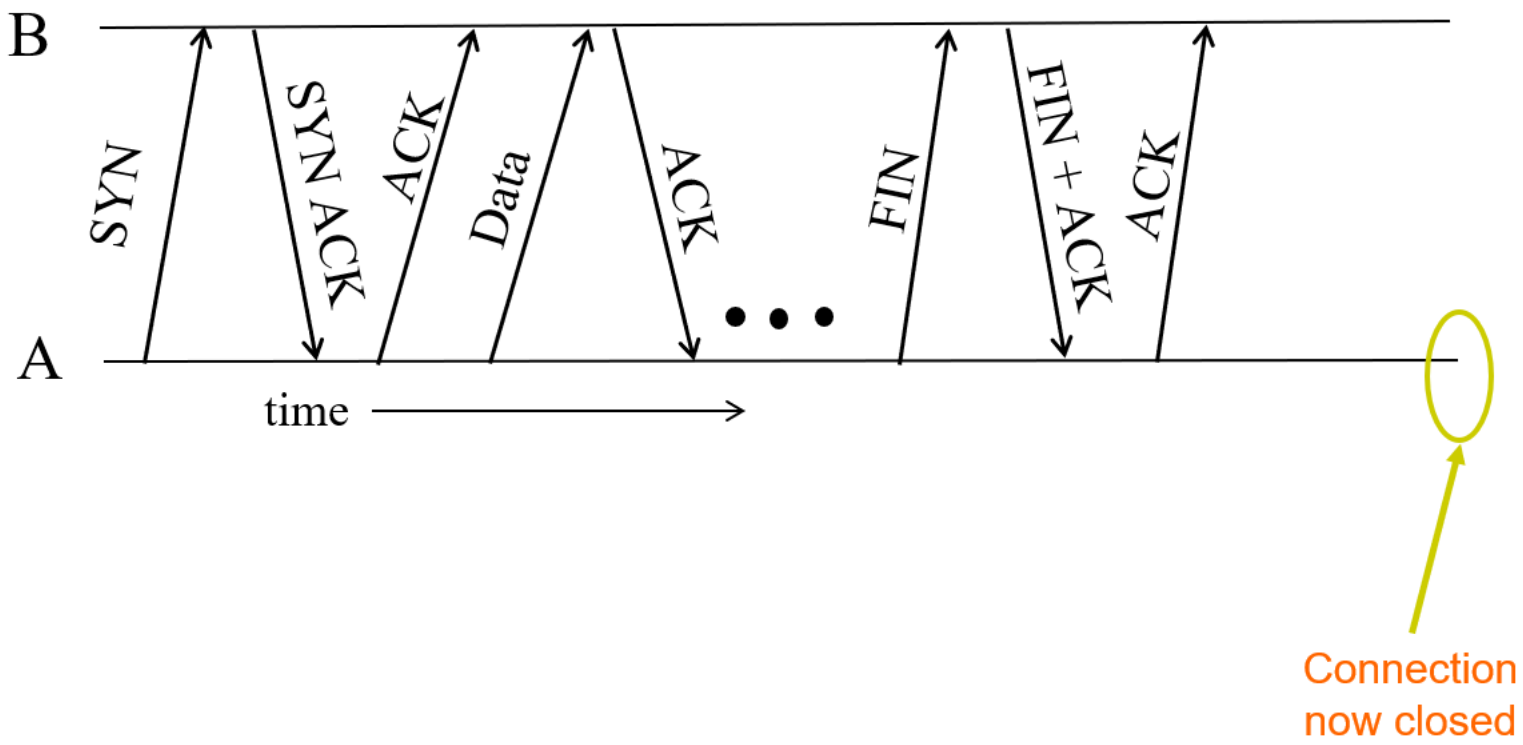

- i nodi decidono contemporaneamente la connessione
- come prima, ma B setta il FIN nel segmento di ACK al FIN di A

# CHIUSURA REGOLARE: ENTRAMBE I NODI




- i nodi decidono contemporaneamente la connessione
- come prima, ma B setta il FIN nel segmento di ACK al FIN di A

# CHIUSURA REGOLARE: ENTRAMBE I NODI



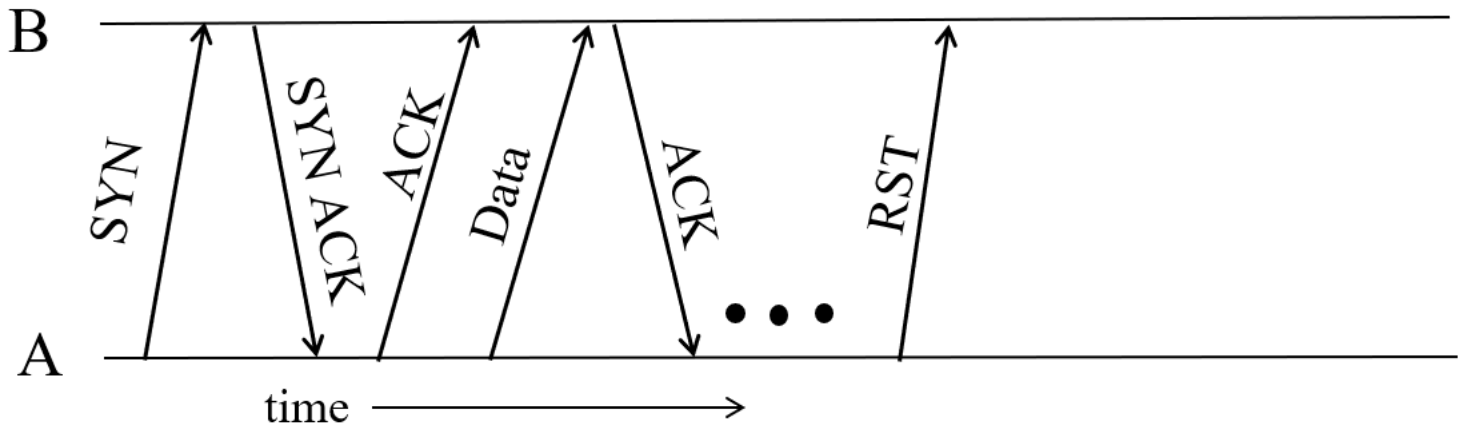

- i nodi decidono contemporaneamente la connessione
- come prima, ma B setta il FIN nel segmento di ACK al FIN di A

# LO STATO TIME WAIT

- una connessione che si trova nello stato time wait non si sposta nello stato closed fino a quando non ha aspettato due volte la massima quantità stimata di tempo di vita di un datagramma IP (ovvero, 120 secondi)

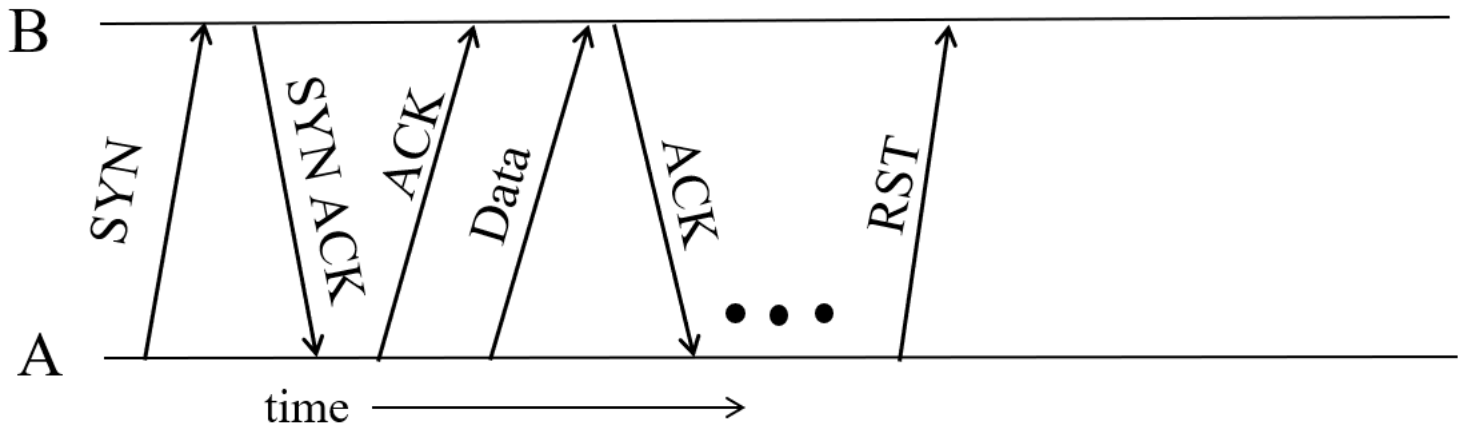
- Motivazione
  - "rincarnazione" di una connessione tra i soliti end-host, sulle stesse porte
  - il segmento con il flag ACK impostato ad 1 in risposta al segmento dell'altro lato con il flag FIN impostato ad 1, potrebbe essere perso
  - l'altro nodo potrebbe ritrasmettere un altro segmento con il flag FIN impostato ad 1 e questo secondo segmento potrebbe essere ritardato nella rete.
  - se la connessione si sposta direttamente nello stato closed, un'altra coppia di processi applicativi potrebbe aprire la stessa connessione (ovvero, usare la stessa coppia di porte)
  - il segmento con il flag FIN, impostato ad 1 ed in ritardo causerebbe la terminazione della seconda "incarnazione della connessione"

# TERMINAZIONE IMPROVVISA




- A invia un RESET (RST) a B
  - ad esempio perchè il processo applicativo A ha subito un crash

# TERMINAZIONE IMPROVVISA

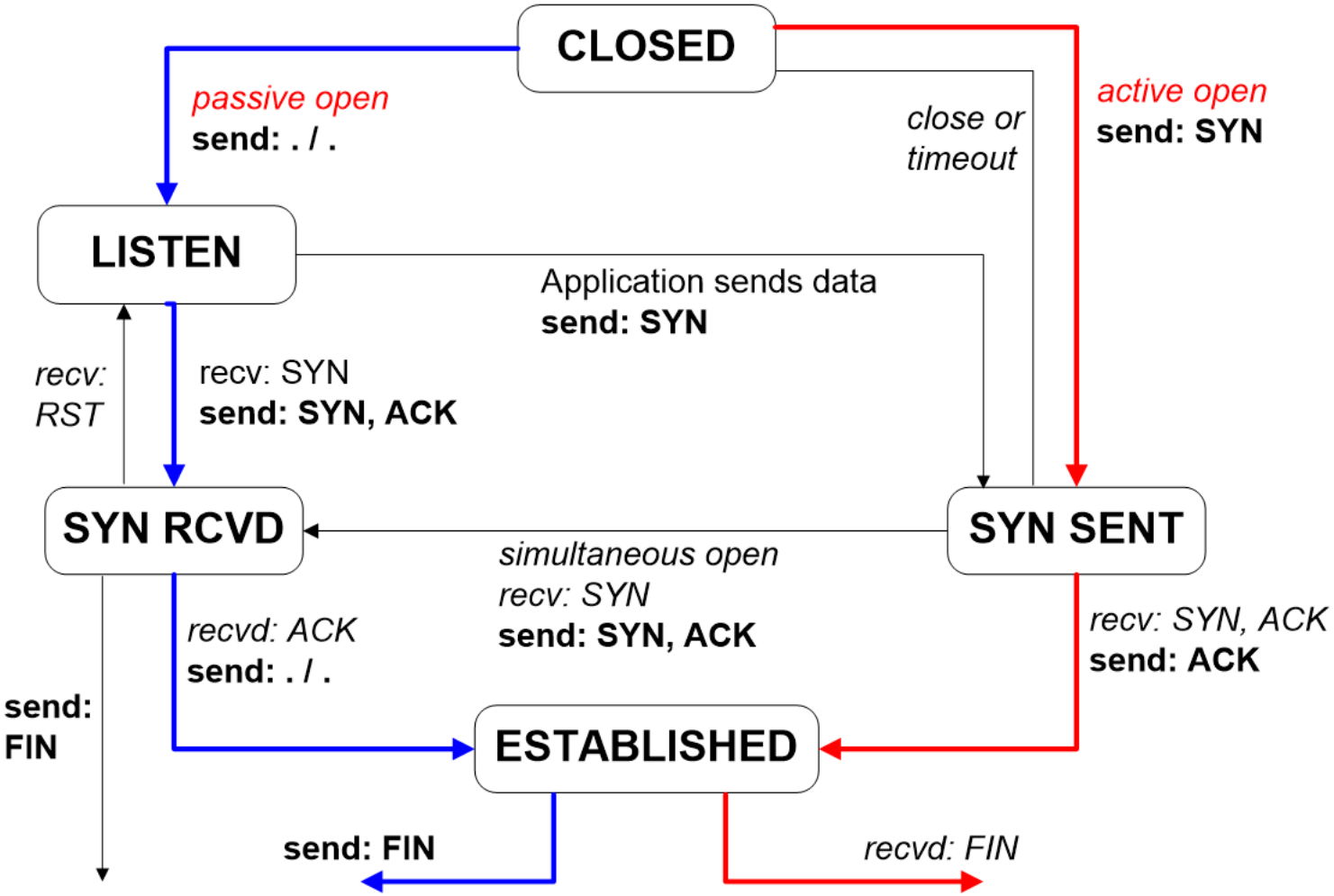



- A invia un RESET (RST) a B
  - ad esempio, perchè il processo applicativo su A ha subito un crash
- Il destinatarioB
  - non riscontra il RST
  - termina la connessione
  - informa la applicazione del RESET
  - ogni data in corso di trasmissione viene perso

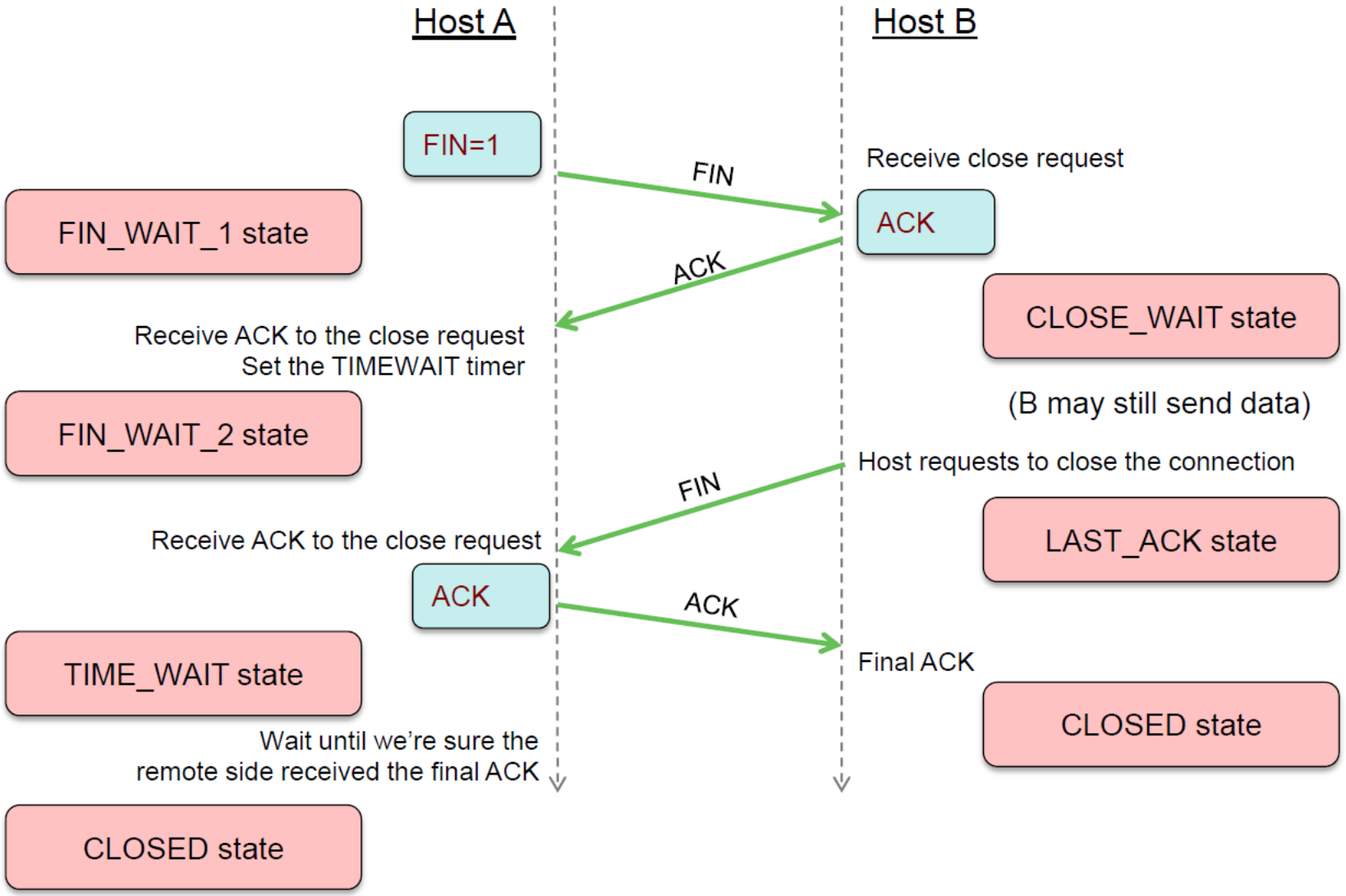
# STATI PROTOCOLLO TCP

| State | Description |
|---|---|
| CLOSED | No connection is active or pending |
| LISTEN | The server is waiting for an incoming call |
| SYN RCVD | A connection request has arrived; wait for Ack |
| SYN SENT | The client has started to open a connection |
| ESTABLISHED | Normal data transfer state |
| FIN WAIT 1 | Client has said it is finished |
| FIN WAIT 2 | Server has agreed to release |
| TIMED WAIT | Wait for pending packets (“2MSL wait state”) |
| CLOSING | Both Sides have tried to close simultanesously |
| CLOSE WAIT | Server has initiated a release |
| LAST ACK | Wait for pending packets |

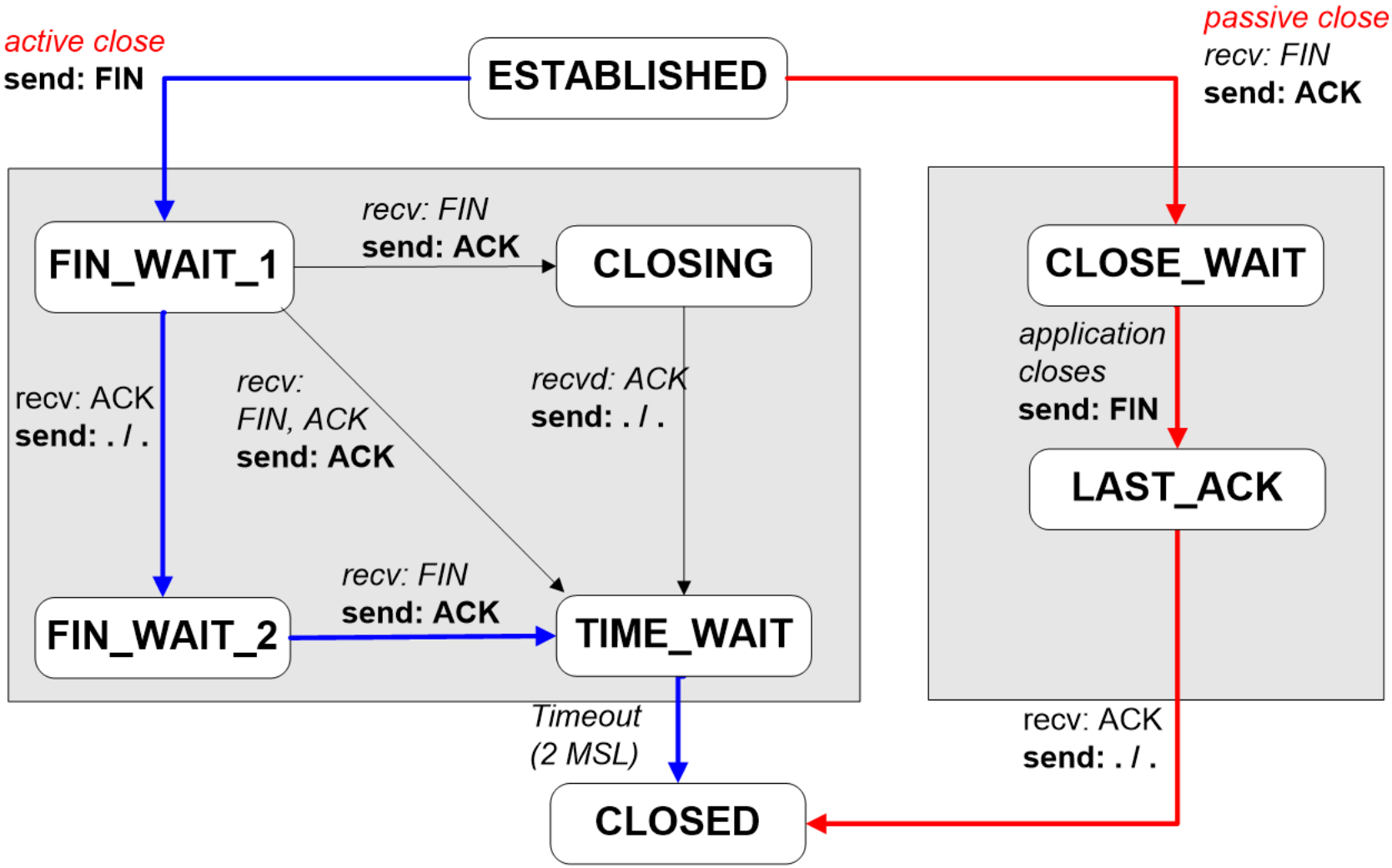
# TCP STATE TRANSITION DIAGRAM: OPENING CONNECTION



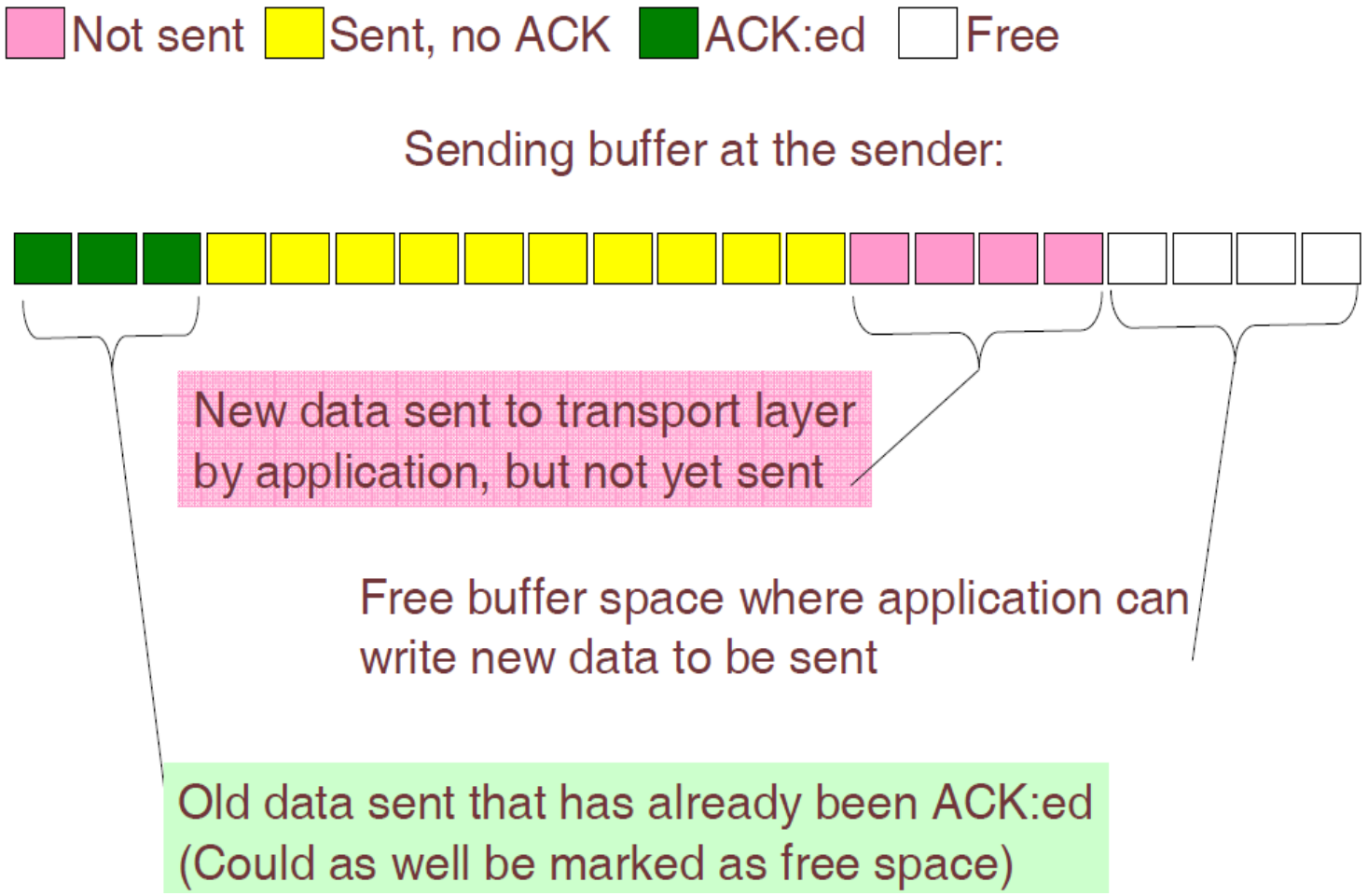
# CHIUSURA REGOLARE: DIAGRAMMA DEGLI STATI



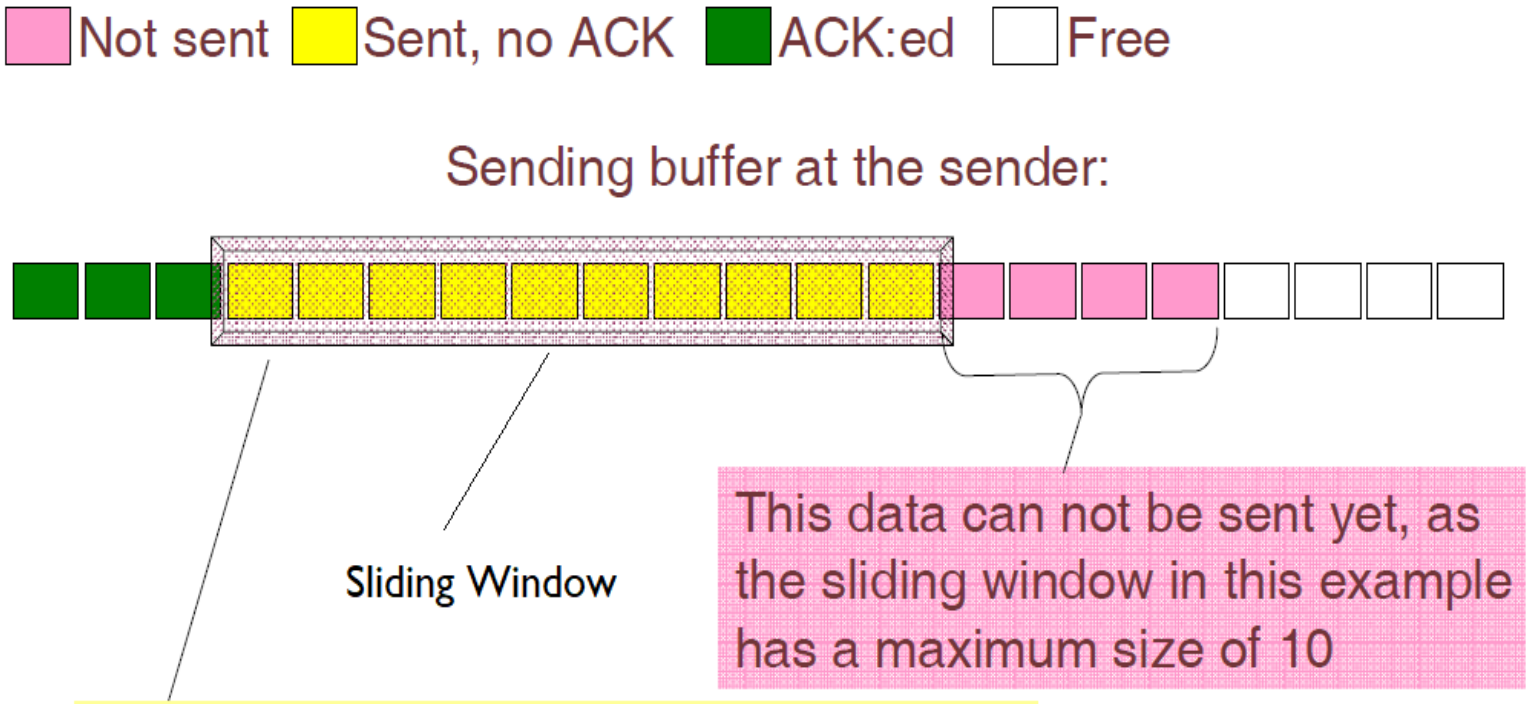
# TCP STATE TRANSITION DIAGRAM: CLOSING CONNECTION



# SLIDING WINDOWS: RICHIAMI



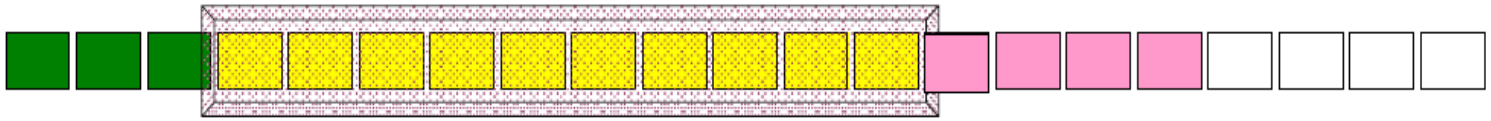
# SLIDING WINDOW: RICHIAMI



# SLIDING WINDOW MITTENTE: RICHIAMI

Not sent | Sent, no ACK | ACK:ed | Free

Sending buffer at the sender:



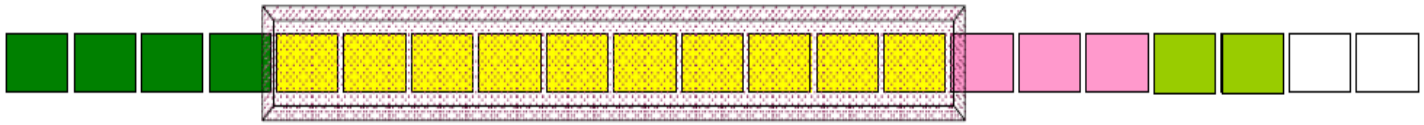
**ACTION**: An ACK of the oldest sent packet arrives

- The window *slides* so that the left border is in line with the oldest outstanding ACK
- The unsent segments that fit within the window are sent

# SLIDING WINDOW MITTENTE: RICHIAMI

Not sent Sent, no ACK ACK:ed Free

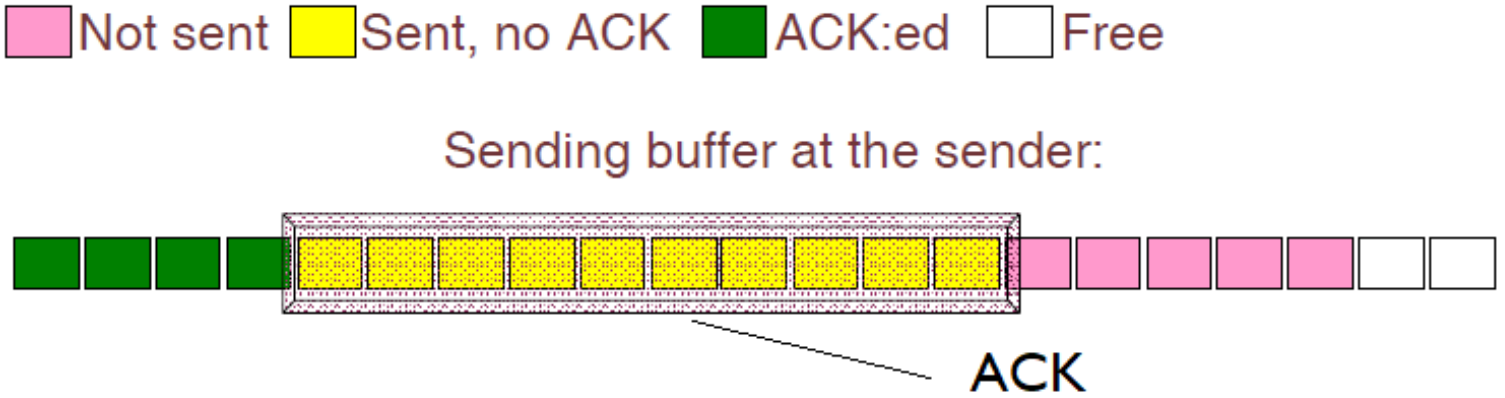
Sending buffer at the sender:



**ACTION**: The application has more data to send

- The data is placed in free buffer slots

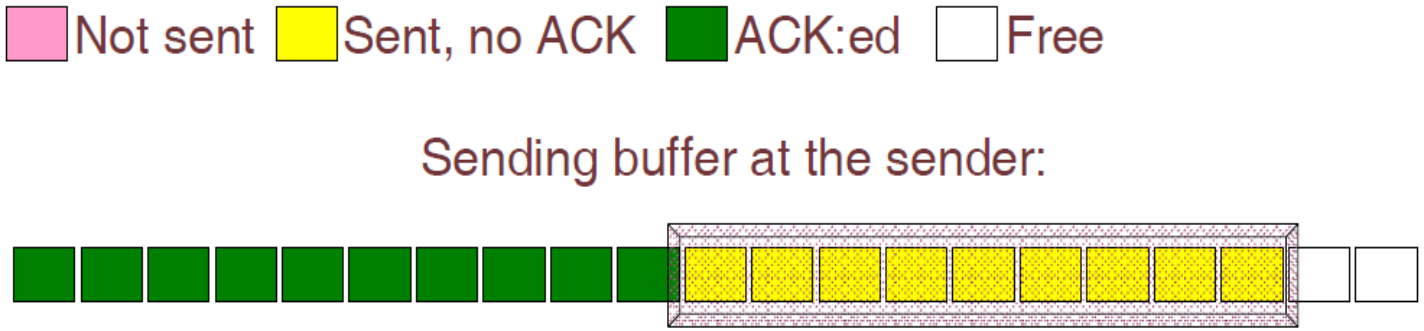
# SLIDING WINDOW MITTENTE : RICHIAMI




**ACTION**: An ACK arrives in the middle of the window

- Older sent but un-ACK:ed segments are now considered to be ACK:ed
- The window slides and unsent segments within the window are sent
- The window shrinks by one segment as there is no more than 9 segments outstanding

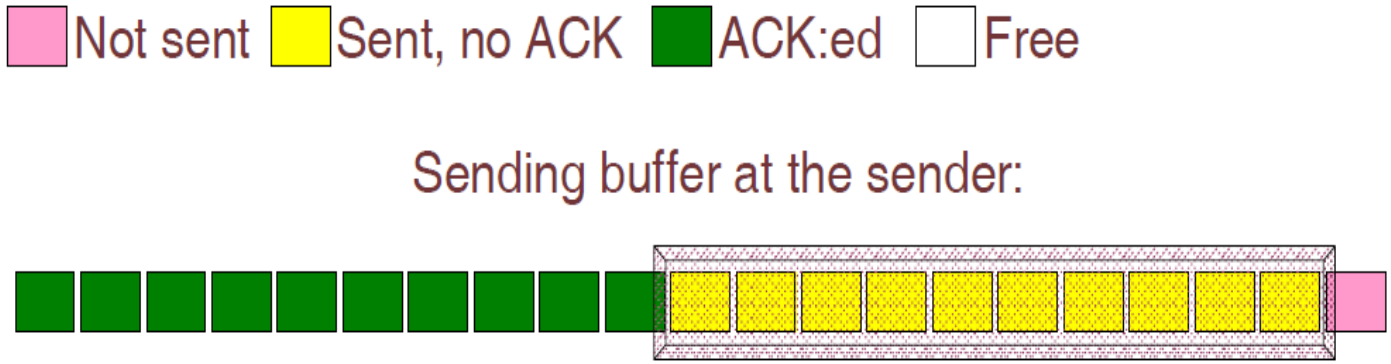
# SLIDING WINDOW MITTENTE: RICHIAMI




**ACTION**: The application has more data to send

- The data is placed in free buffer slots
- As the window is currently 9 segments wide, it can grow by one segment
- The new data that fits within the window is sent

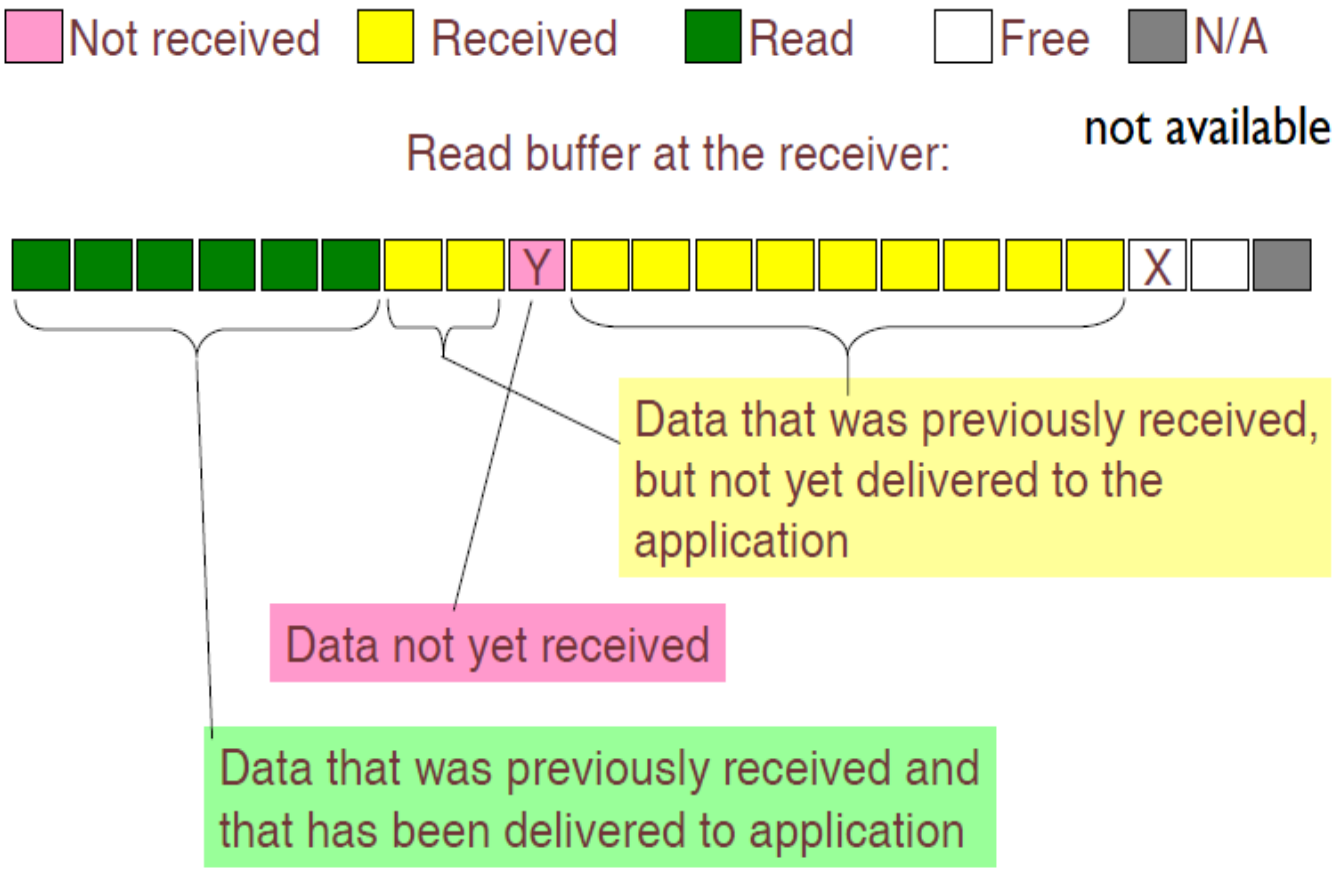
# SLIDING WINDOW MITTENTE: RICHIAMI



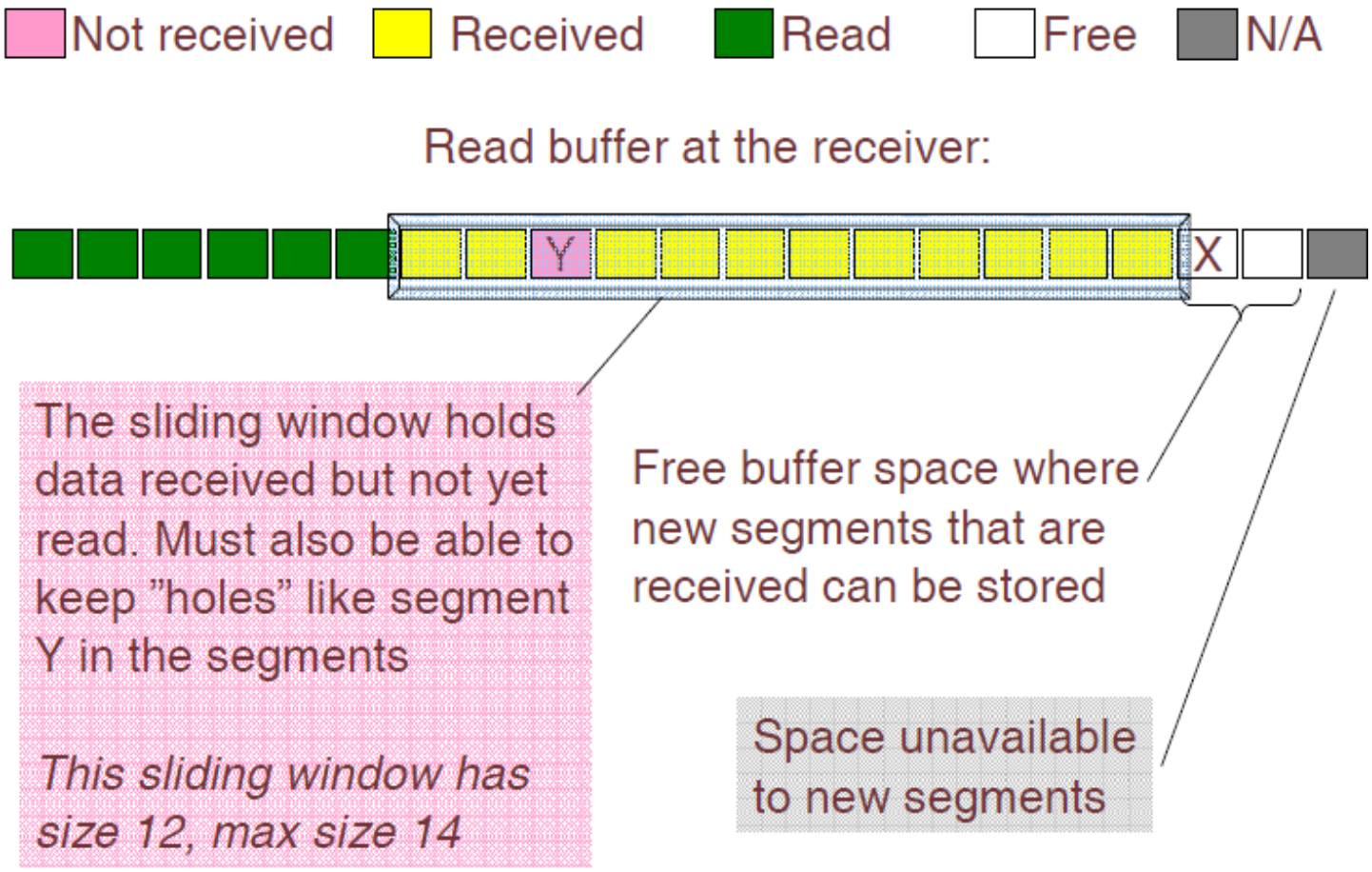

**ACTION**: An ACK of already ACK:ed segments arrives

- The ACK is silently ignored

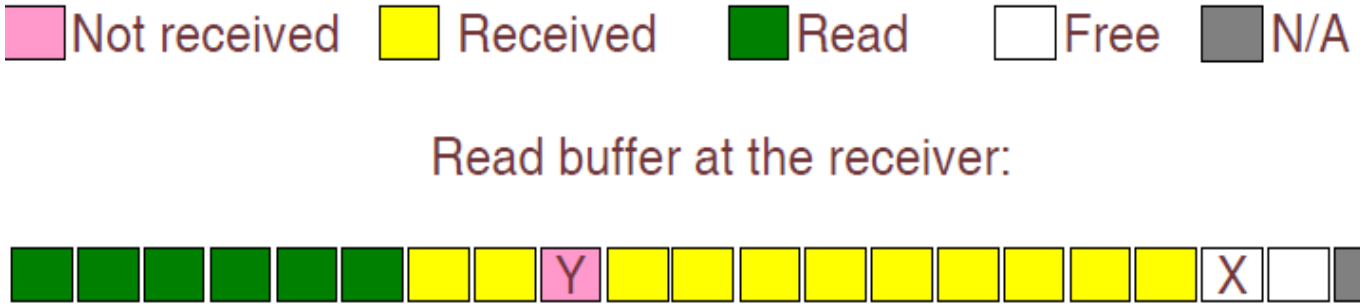
# SLIDING WINDOWS DESTINATARIO: RICHIAMI



# SLIDING WINDOWS DESTINATARIO: RICHIAMI



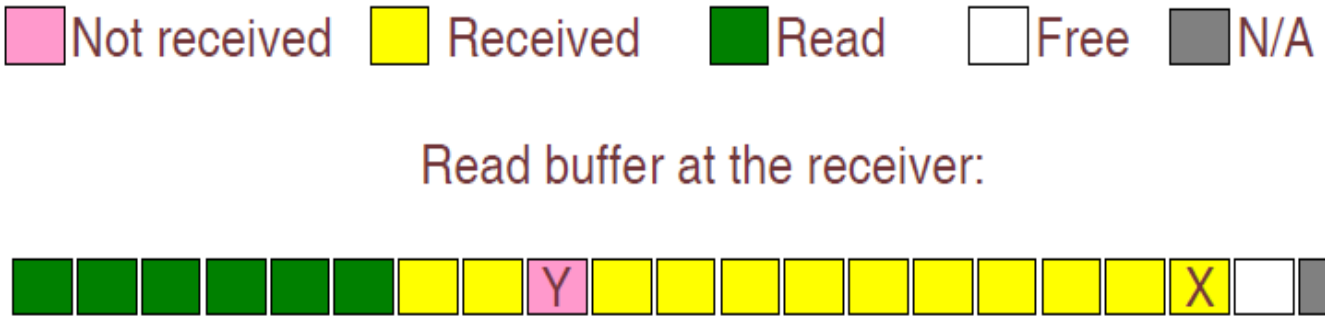
# SLIDING WINDOWS DESTINATARIO: RICHIAMI




**ACTION**: Segment X arrives

- Store in read buffer, register as received
- Send cumulative ACK Y to indicate that receiver is waiting for Y

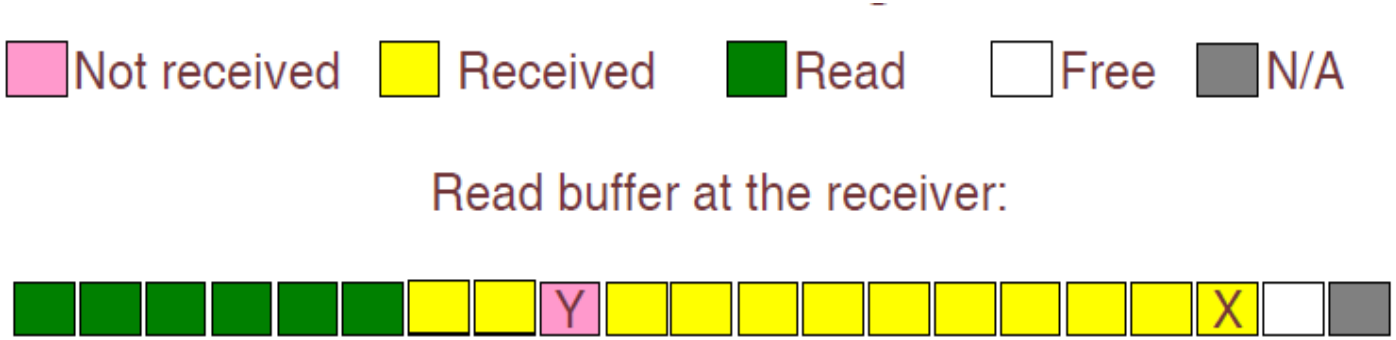
# SLIDING WINDOWS DESTINATARIO: RICHIAMI




**ACTION**: Segment X+2 arrives

- Can not fit into the buffer, must be discarded
- Send cumulative ACK Y to indicate that receiver is waiting for Y

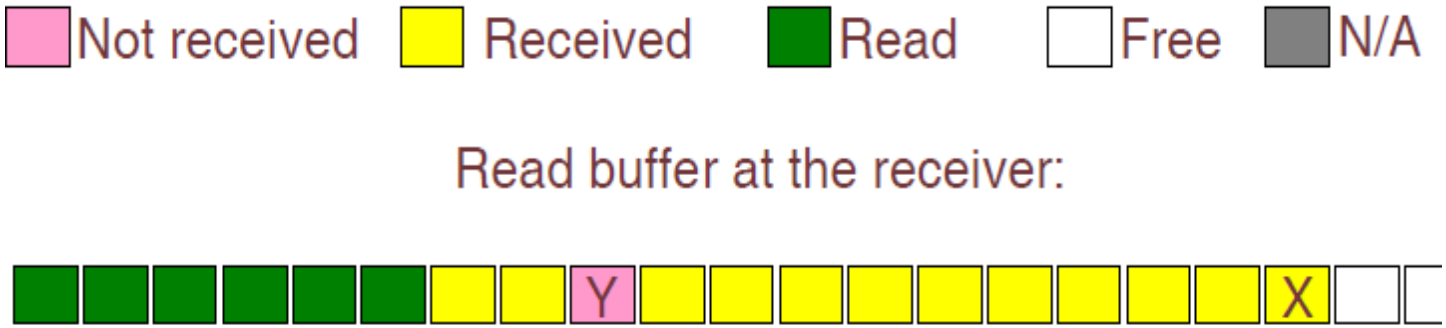
# SLIDING WINDOWS DESTINATARIO: RICHIAMI




**ACTION**: Applications try to read 5 segments

- Only two segments are returned, still waiting for Y
- Application is informed of how much data was read
- The unavailable segment at the end of the buffer becomes available

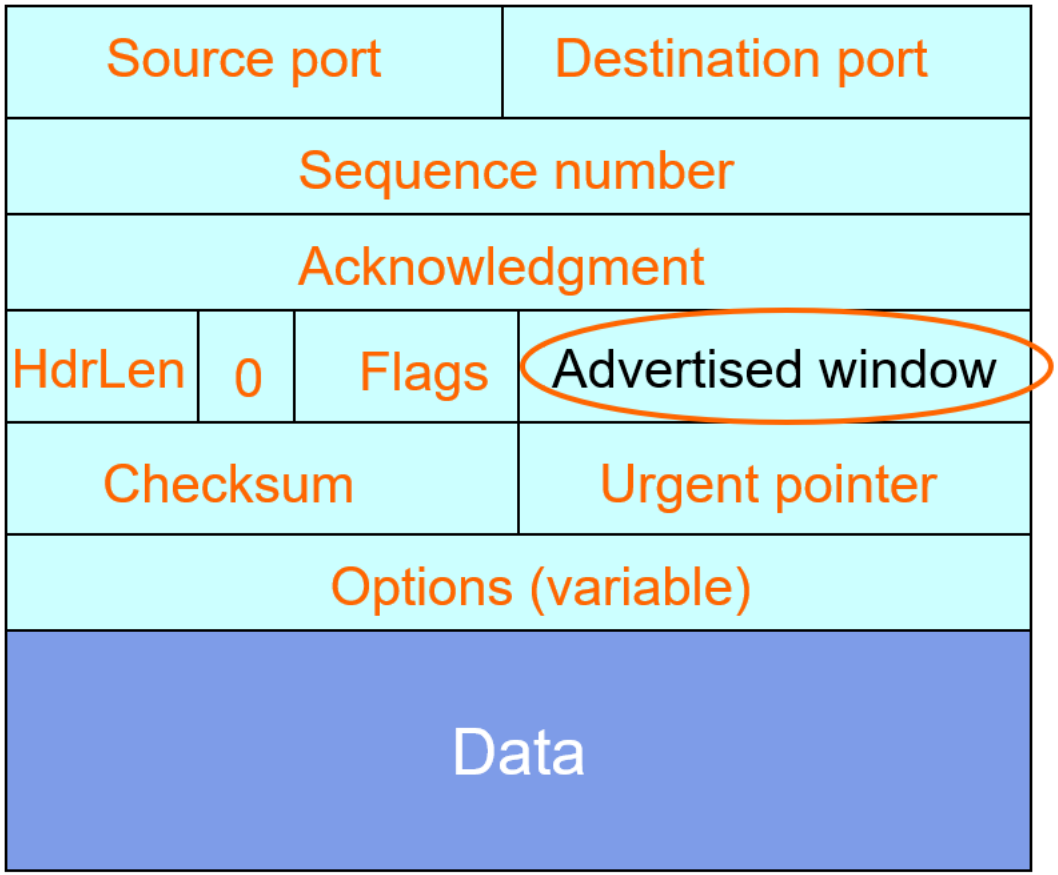
# SLIDING WINDOWS DESTINATARIO: RICHIAMI




**ACTION**: Segment Y arrives

- Store in read buffer, register as received
- Send cumulative ACK (X+1) to indicate that receiver is waiting for (X+1)

# IL CONTROLLO DEL FLUSSO



# TCP FLOW CONTROL E SLIDING WINDOWS

- Il controllo di flusso ha lo scopo di limitare il ritmo di emissione dei dati da parte di un end-host per evitare la saturazione della capacità del buffer di ricezione

- TCP utilizza il concetto di sliding window, sia lato mittente che lato destinatario, per diversi scopi:
  - garantire l'affidabilità dei dati
  - assicurare che i dati vengano consegnati in ordine
  - supportare il controllo del flusso tra mittente e destinatario

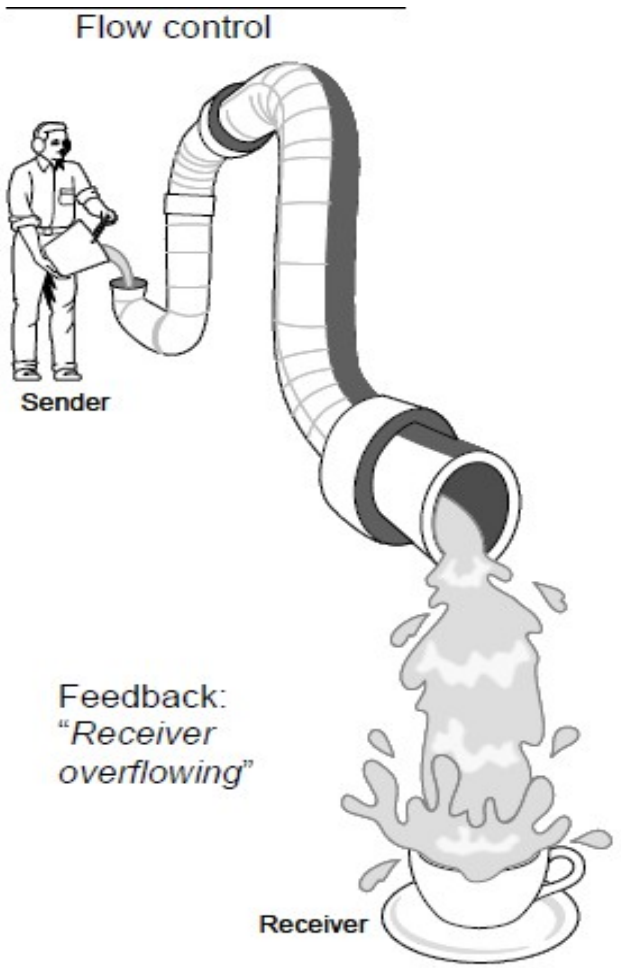
- TCP definisce il controllo del flusso basato su una sliding window di larghezza variabile

  - lo scorrimento e la larghezza della finestra sono controllati dall'entità TCP ricevente
  - Il controllo di flusso opera a livello di byte

# TCP FLOW CONTROL

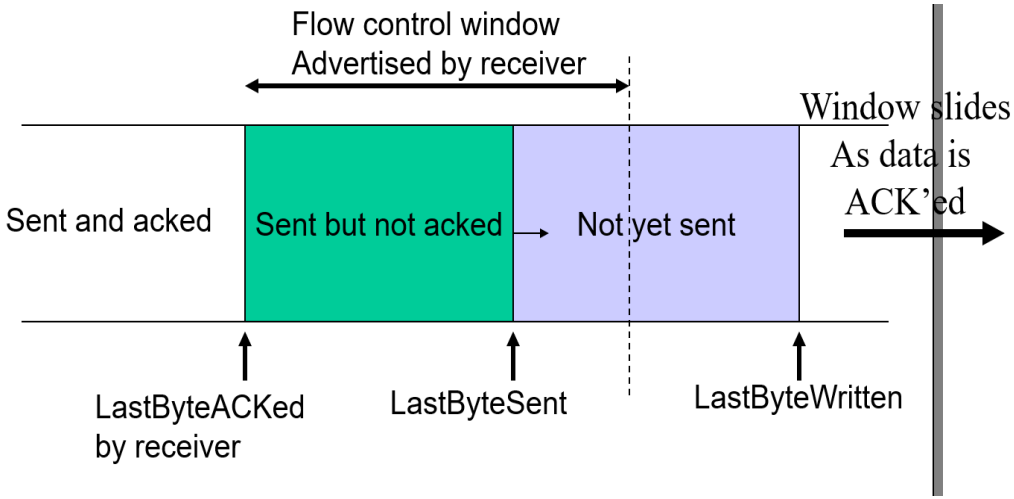
- controllo del flusso: meccanismo di "speed matching" tra mittente e destinatario
  - confronta la frequenza di invio con quella di ricezione
  - regola la quantità di dati che il mittente può inviare al destinatario prima di ricevere un ACK.

- due casi estremi
  - attesa dell'ACK di ogni byte spedito: troppo lento
  - invio di tutti i dati disponibili senza attandere ACK: overflow nel destinatario?



# SLIDING WINDOWS LATO MITTENTE

- mittente: lato sinistro della finestra:
  - inizio dei dati non riscontrati
  - lato sinistro della finestra + costante: spazio a disposizione fornito dal SO
  - il controllo del flusso determina dinamicamente il lato destro della finestra

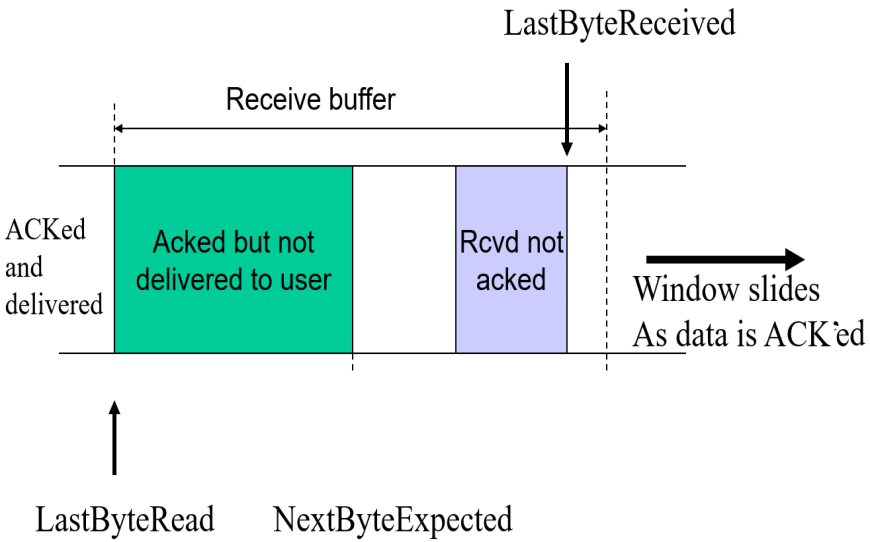



Sender side

- LastByteAcked <= LastByteSent
- LastByteSent <= LastByteWritten

# SLIDING WINDOW LATO DESTINATARIO

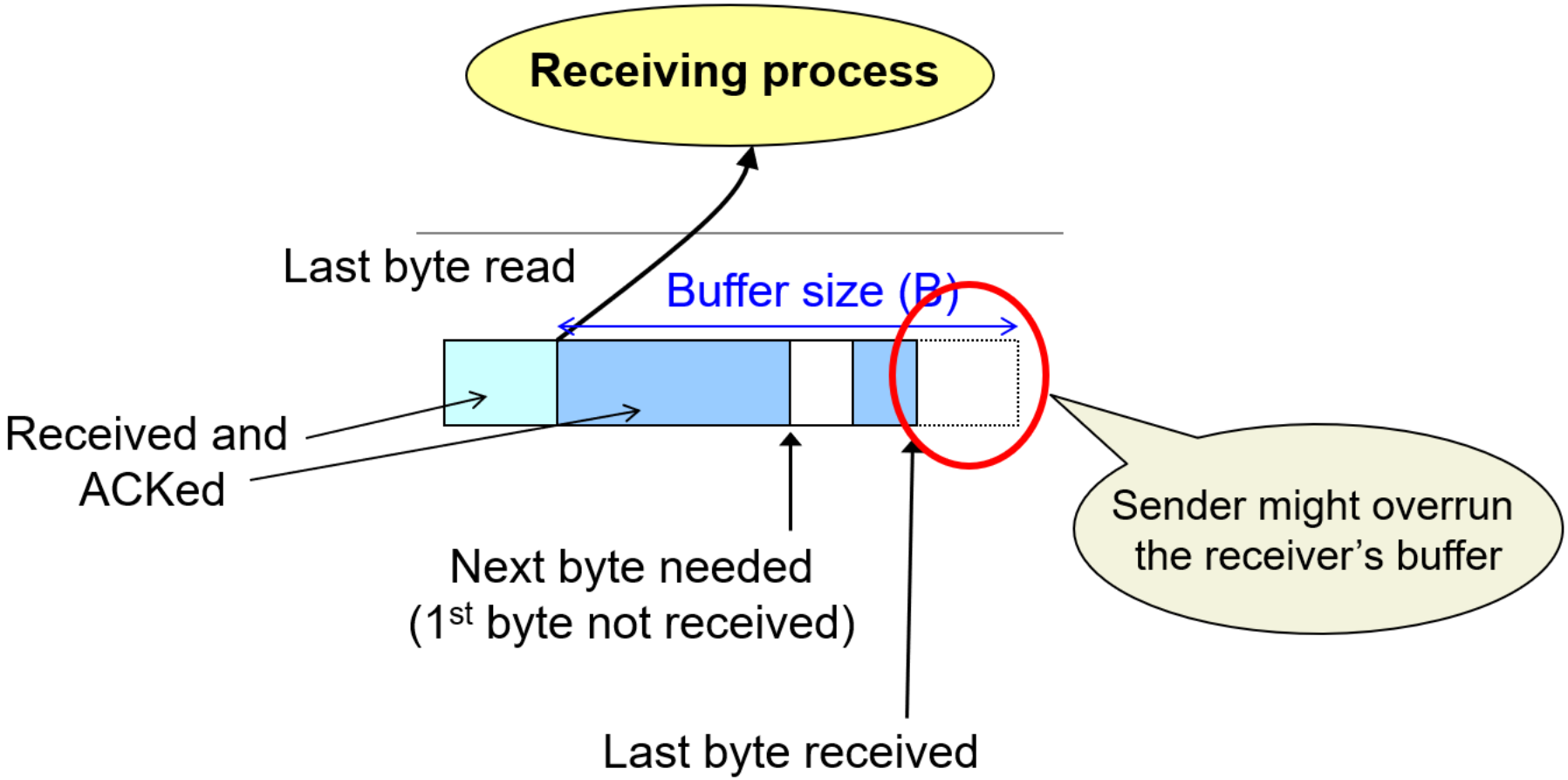
- destinatario: lato sinistro della finestra:
  - inizio dei dati non riscontrati
  - lato sinistro della finestra + costante: spazio a disposizione fornito dal SO
  - il controllo del flusso determina dinamicamente il lato destro della finestra




LastByteRead < NextByteExpected

NextByteExpected <= LastByteRcvd +1

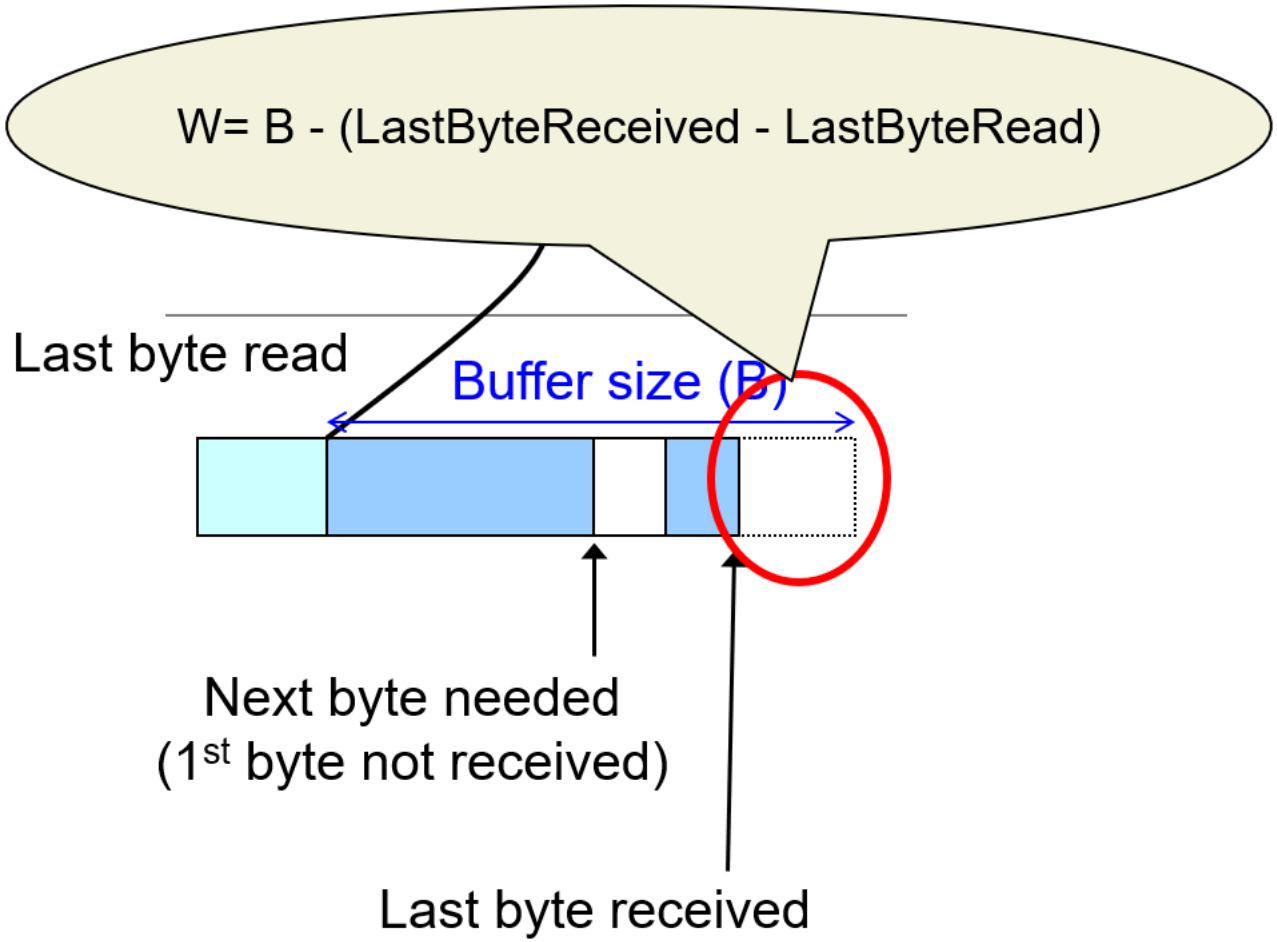
# SLIDING WINDOW DESTINATARIO: OVERFLOW



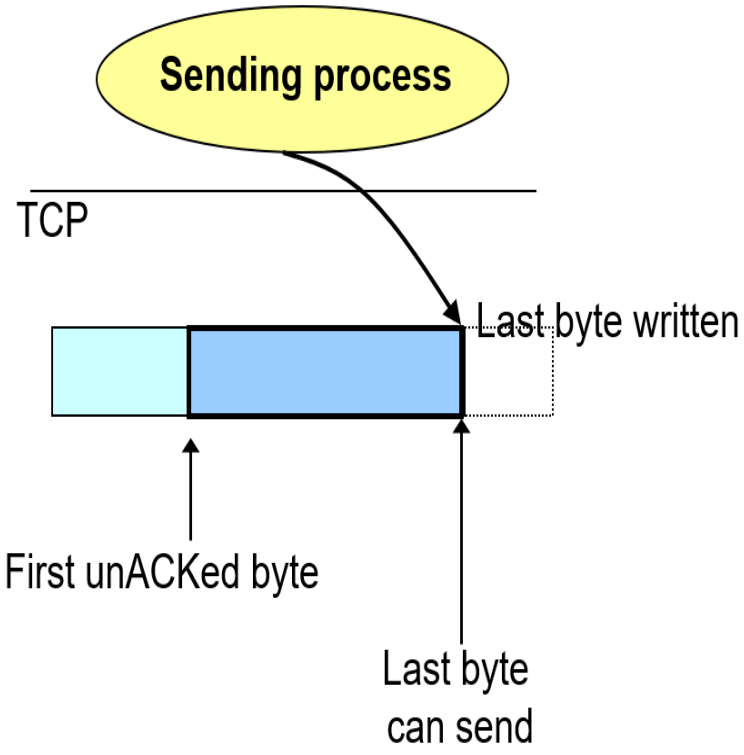
# TCP: CONTROLLO DEL FLUSSO

- il ricevente
  - usa una “Advertised Window” (W) per evitare che il mittente “inondi” la sua finestra di ricezione
  - indica il valore di W nell'ACK che invia
- Il mittente
  - limita il numero di bytes non riscontrati ad essere <= W

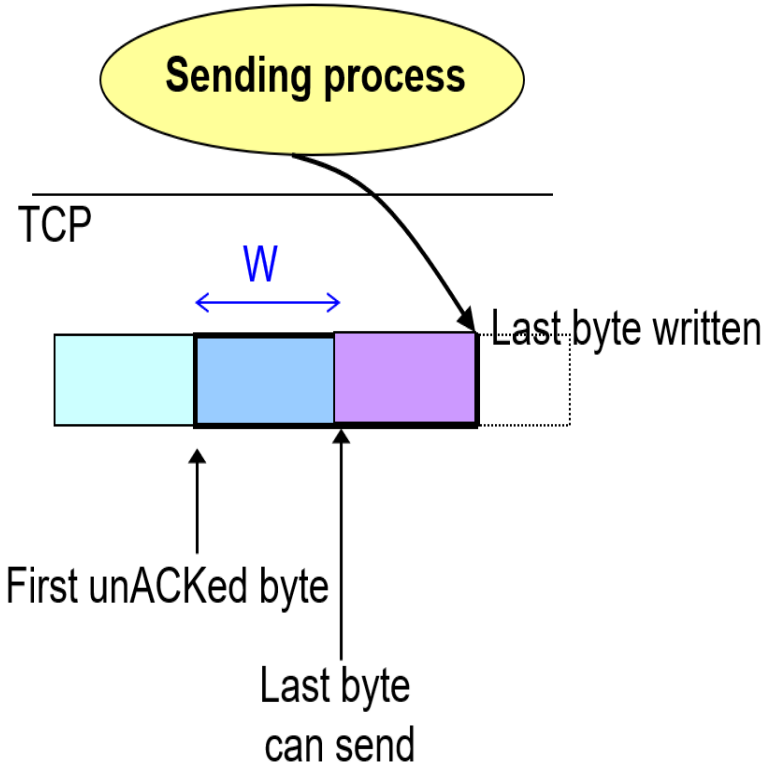
# ADVERTISED WINDOW: DIMENSIONE



# ADVERTISED WINDOW (W)

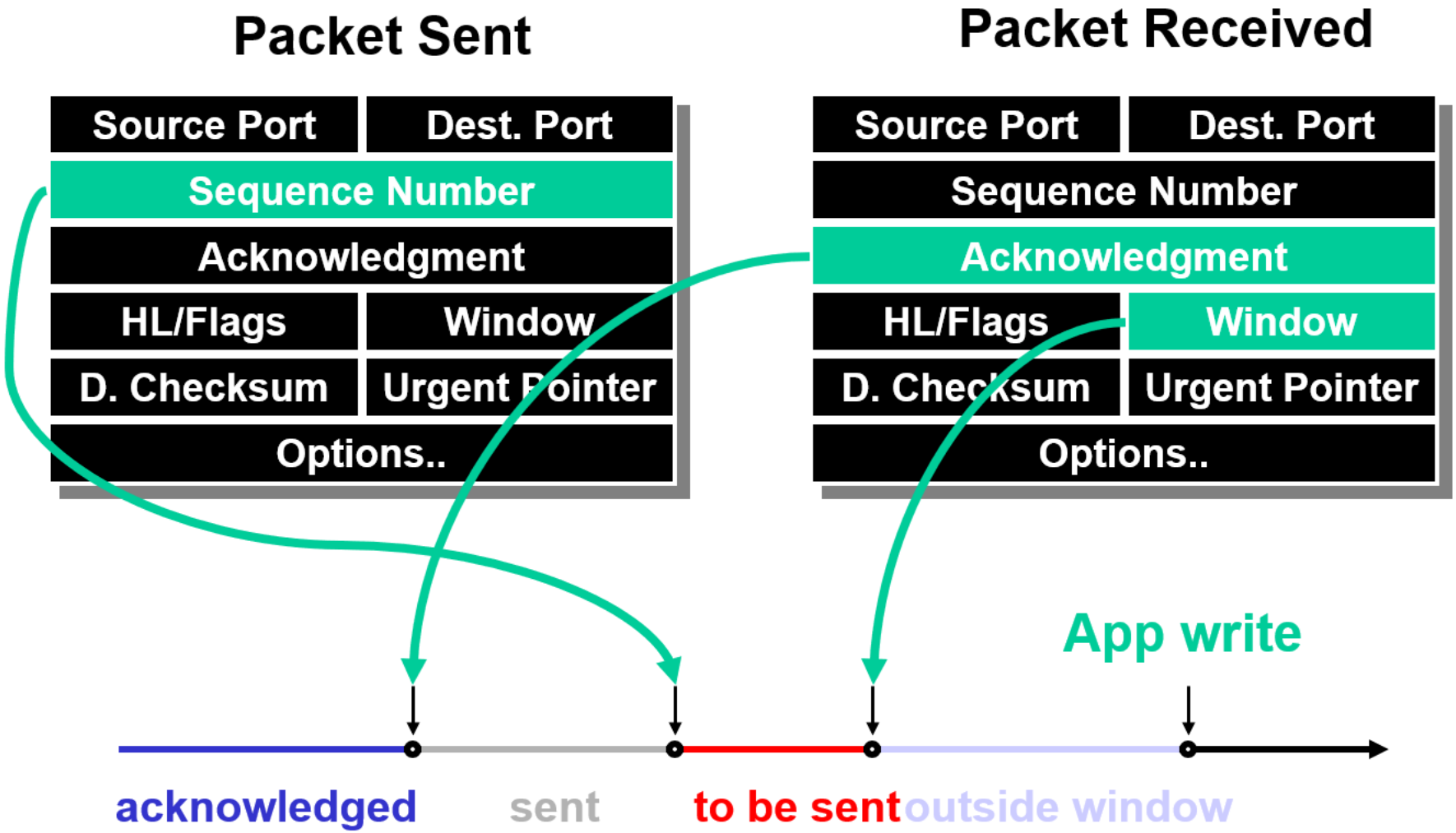



Prima di ricevere W




Dopo aver ricevuto W

# CONTROLLO DEL FLUSSO: CAMPI DELL'HEADER



# SILLY WINDOW SYNDROME : LATO DESTINATARIO

- lato destinatario:
  - il ricevitore svuota lentamente il buffer di ricezione
  - invia segmenti con finestra molto piccola
  - il mittente invia segmenti corti con molto overhead
- soluzione (algoritmo di Clark)
  - il ricevitore "mente" al trasmettitore indicando una finestra nulla sino a che il suo buffer di ricezione non si è svuotato per metà o per una porzione almeno pari al MSS

# SILLY WINDOW SYNDROME: LATO MITTENTE

- L'applicazione genera dati lentamente:
  - invia segmenti molto piccoli man mano che vengono prodotti

- Soluzione (algoritmo di Nagle)
  - Il TCP sorgente invia la prima porzione di dati anche se corta
  - gli altri segmenti vengono generati e inviati solo se
    - Il buffer d'uscita contiene dati sufficienti a riempire un MSS
    - oppure, quando si riceve un acknowledgement per un segmento precedente.

# CLASSE SOCKET DI JAVA: OPZIONI

- la classe Socket offre la possibilità di impostare diverse proprietà del socket
- opzioni
  - SO_TIMEOUT
  - SO_RCVBUF
  - SO_SNDBUF
  - SO_KEEPALIVE
  - TCP_NODELAY
  - SO_LINGER

  .....

# CLASSE SOCKET: TCP_NODELAY

Algoritmo di Nagle:

- introdotto per evitare che il TCP spedisca una sequenza di piccoli segmenti, quando la frequenza di invio dei dati da parte della applicazione è molto bassa
- riduce il numero di segmenti spediti sulla rete fondendo in un unico segmento più dati
- applicazione originaria dell'algoritmo
  - sessioni Telnet, in cui è richiesto di inviare i singoli caratteri introdotti, mediante keyboard, dall'utente
  - se l'algoritmo di Nagle non viene applicato, ogni carattere viene spedito in un singolo segmento, (1 byte di data e decine di byte di header del messaggio)
- Motivazioni per disabilitare l'algoritmo di Nagle: trasmissioni di dati in 'tempo reale', ad esempio movimenti del mouse per un'applicazione interattiva come un gioco multiplayer

# CLASSE SOCKET: TCP_NODELAY

Algoritmo di Nagle:

- in generale, per default, l'algoritmo di Nagle risulta abilitato
- tuttavia alcuni sistemi operativi disabilitano l'algoritmo di default
- per disabilitare l'algoritmo di Nagle

sock.setTcpNoDelay(true)

disabilita la bufferizzazione (no delay= non attendere, inviare subito un segmento, non appena l'informazione è disponibile)

- JAVA RMI, ad esempio, disabilita l'algoritmo di Nagle
  - lo scopo è quello di inviare prontamente il segmento contenente i parametri di una call remota oppure il valore restituito dall'invocazione di un metodo remoto

# FLAG DI CONTROLLO DELL'HEADER: RIASSUNTO

| Flag | Description |
|---|---|
| URG | If this bit field is set, the receiving TCP should interpret the urgent pointer field. Used when a section of data should be read out by the receiving application quickly. The rest of the segment is processed normally. |
| ACK | If this bit field is set, the acknowledgement field is valid. |
| PSH | If this bit field is set, the sender/receiver should deliver this segment to the TCP module/receiving application as soon as possible, without waiting for *receive window* to get filled. |
| RST | If this bit is present, it signals the receiver that the sender is aborting the connection and all queued data and allocated buffers for the connection can be freely relinquished. |
| SYN | When present, this bit field signifies that sender is attempting to "synchronize" sequence numbers. This bit is used during the initial stages of connection establishment between a sender and receiver. |
| FIN | If set, this bit field tells the receiver that the sender has reached the end of its byte stream for the current TCP connection. |