# RETI DI CALCOLATORI

Autunno 2018

docente: Laura Ricci

laura.ricci@unipi.it

# Lezione 9:
# COMUNICAZIONE AFFIDABILE: PRINCIPI
# 05/11/2018

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

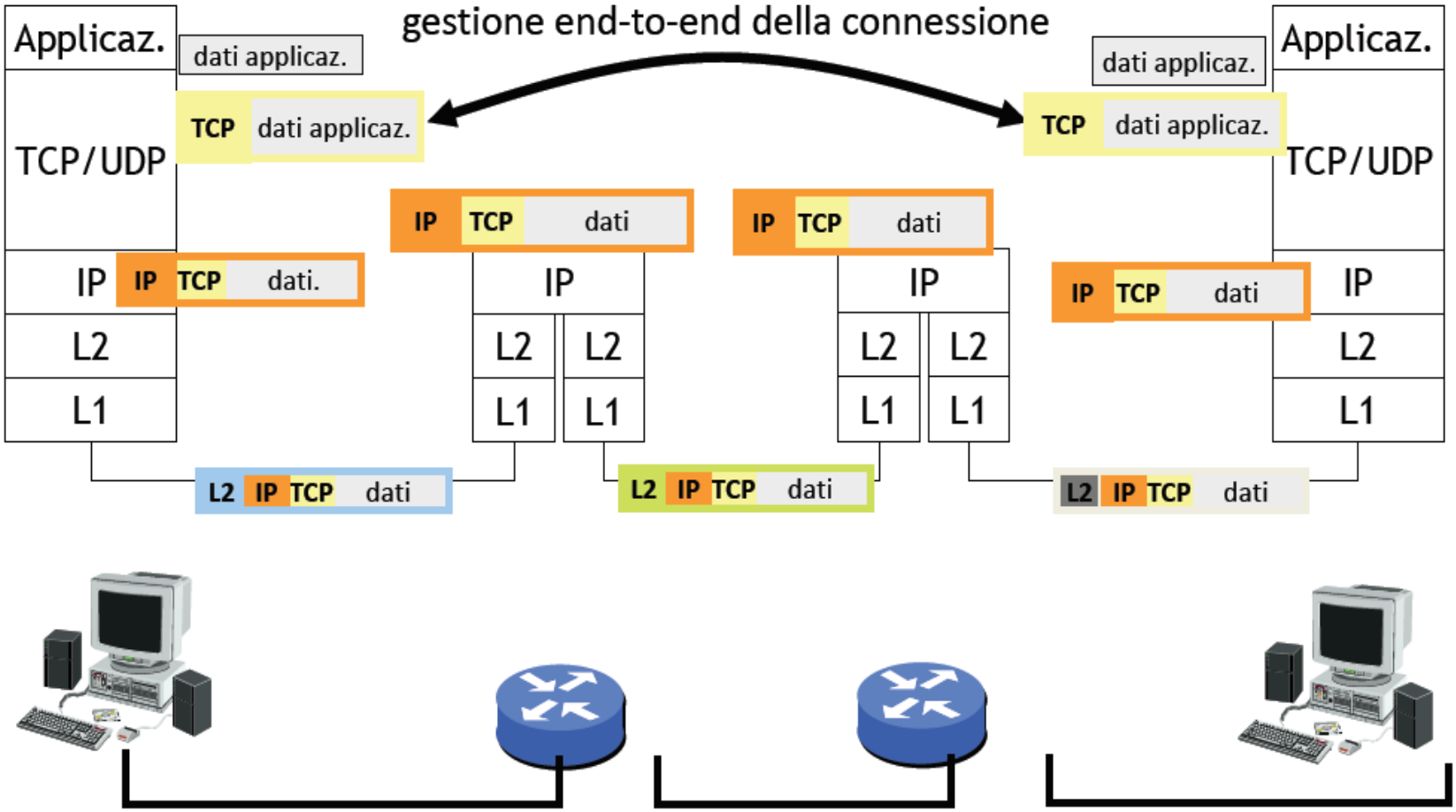
# MATERIALE DIDATTICO

- Forouzan, capitolo 3
  - paragrafo 3.1: Introduzione
  - paragrafo 3.2: Protocolli di livello Trasporto

# PROTOCOLLI E SERVIZI DI TRASPORTO

- forniscono la comunicazione logica tra processi applicativi di host diversi
- sono eseguiti negli end host
  - lato mittente: organizzare i messaggi applicativi in segmenti e passarli al livello di rete
  - lato ricevente: riassemblare i segmenti in messaggi e passarli al livello di applicazione
- Il livello trasporto, come quello di applicazione, non ha visibilità dei dettagli della rete
  - la semantica del protocollo è end-to-end e la rete viene trattata come una scatola nera
- Lo scopo "generale" del protocollo è fornire un'astrazione di comunicazione ai protocolli di livello applicativo
  - attraverso astrazioni chiamate socket
  - socket: varie librerie JAVA per la loro gestione

# ARCHITETTURA A LIVELLI: LIVELLO TRASPORTO



# IL LIVELLO TRASPORTO

- fornisce un canale di trasporto end-to-end "astratto" tra due utenti, indipendentemente dalla rete.
- per raggiungere questo obiettivo, come tutti i livelli, il livello di trasporto offre, attraverso delle primitive, dei servizi al livello superiore e svolge una serie di funzioni
- servizi offerti al livello applicativo:

| | |
|---|---|
| connection-oriented affidabile<br>• per il trasferimento dei dati viene attivata una connessione<br>• ogni pacchetto (→ segmento) inviato viene "riscontrato" in modo individuale | TCP |
| connectionless non affidabile<br>• non viene attivata nessuna connessione<br>• invio delle trame senza attendere alcun feedback dalla destinazione sulla corretta ricezione<br>– se una trama viene persa non ci sono tentativi per recuperarla | UDP |

# IL LIVELLO DI TRASPORTO

- IP fornisce un modello di servizio "best effort"
  - i pacchetti possono essere corrotti, ritardati, persi, riordinati, duplicati a causa di
    - overflow delle code nei router a causa della congestione
    - collisioni ripetute su un mezzo di trsmssione condiviso
    - fallimenti del processo di routing
    - inoltro su rotte diverse, caratterizzate da ritardi diversi
  - non permette di guidare quando e quanto traffico inoltrare
- gestire tutti questi problemi a livello applicazione sarebbe molto oneroso per lo sviluppatore che, invece, si deve occupare solo della logica della applicazione
- Il livello trasporto offre servizi/protocolli per gestire i precedenti problemi.

# IL LIVELLO DI TRASPORTO: FUNZIONI

- multiplexing e demultiplexing dei messaggi diretti verso le applicazioni
- scarto segmenti malformati o con errori nell'header
- instaurazione, gestione e rilascio delle connessioni
  - gestione dello scambio di informazioni necessarie per concordare l'attivazione di un canale di comunicazione
- ritrasmissione segmenti persi
- consegna ordinata dei segmenti
- controllo di flusso
  - azione preventiva finalizzata a limitare l'immissione di dati in rete a seconda della capacità end-to-end degli host
- controllo della congestione
  - azioni da intraprendere come reazione alla congestione di rete

UDP

TCP

# IL PROBLEMA DELLA AFFIDABILITA'

problema della affidabilità della comunicazione:

- data una sequenza di pacchetti P1, P2, . . . , PN, generati da un mittente
- il mittente deve consegnare questi pacchetti al destinatraio nello stesso ordine con cui vengono generati
- nonostante tutto quello che accade in una rete "best effort", dove un pacchetto può
  - essere corrotto (bit corrotti)
  - essere perso
  - subire ritardi consistenti
  - essere duplicato
  - essere riordinato rispetto ad un altro pacchetto della stessa sequenza

# RELIABLE TRANSPORT

- Il livello trasporto deve assicurare che una sequenza di pacchetti sia consegnata
  - completa ed in ordine
  - senza errori
  - senza duplicazioni

- Protocolli ARQ (Automatic Repeat ReQuest): il destinatario individua gli errori nel pacchetto e richiede al mittente di ripetere la trasmissione di quei pacchetti.
  - Stop-and-Wait
  - Go-Back N
  - Selective Repeat

- Elementi di base per la definizione di un protocollo ARQ:
  - error-detecting code che permetta un'alta copertura di errori
  - ACKs (positive acknowledgments) (eventuali NACK)
  - meccanismi di time-out
  - numeri di sequenza

# ERROR DETECTING CODES: INTENET CHECKSUM

- metodo di individuazione degli errori usato da IP, TCP, UDP.
- calcolo della checksum:
  - il pacchetto IP/TCP/UDP è diviso in sezioni di n-bits
  - le sezioni sono sommate usando l'aritmetica in "complemento ad 1"
    - la somma è ancora lunga n bits!
  - la somma è complementata per produrre la checksum (il complemento di un numero nell'aritmetica ad 1 bit è il negativo del numero)
- vantaggi
  - segmenti relativamente piccoli, overhad controllato:
  - facile implementazione
- svantaggi
  - minore protezione rispetto a metodi più complessi, ad esemio CRC
  - non individua bytes/words riordinati
  - individuano tutti gli errori che coinvolgono un numero dispari di bits
  - e la maggior parte degli errori che coinvolgono un numero pari di bit

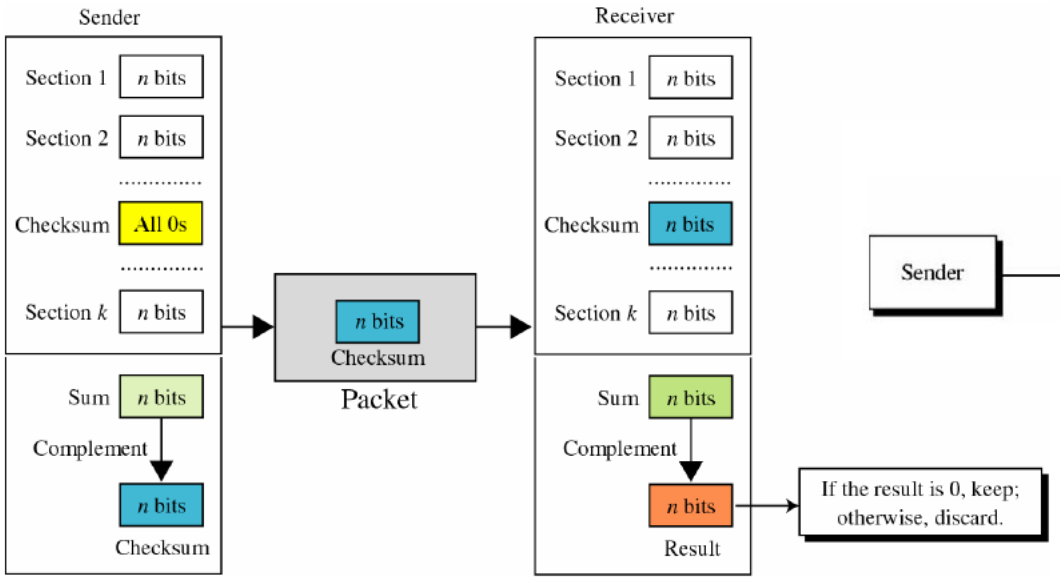
# INTERNET CHECKSUM

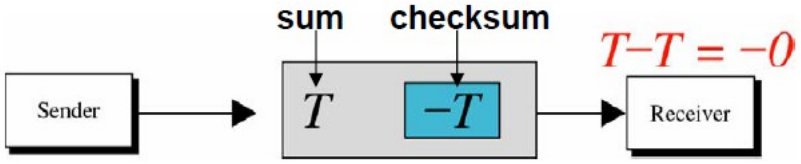
Mittente:

- i dati sono divisi in k sezioni, di n bit
- le sezioni sono sommate in complemento a 1
- la somma risultante è complementata bit a bit
  - il risultato è la checksum
- la checksum è inviata con i dati

Ricevente:

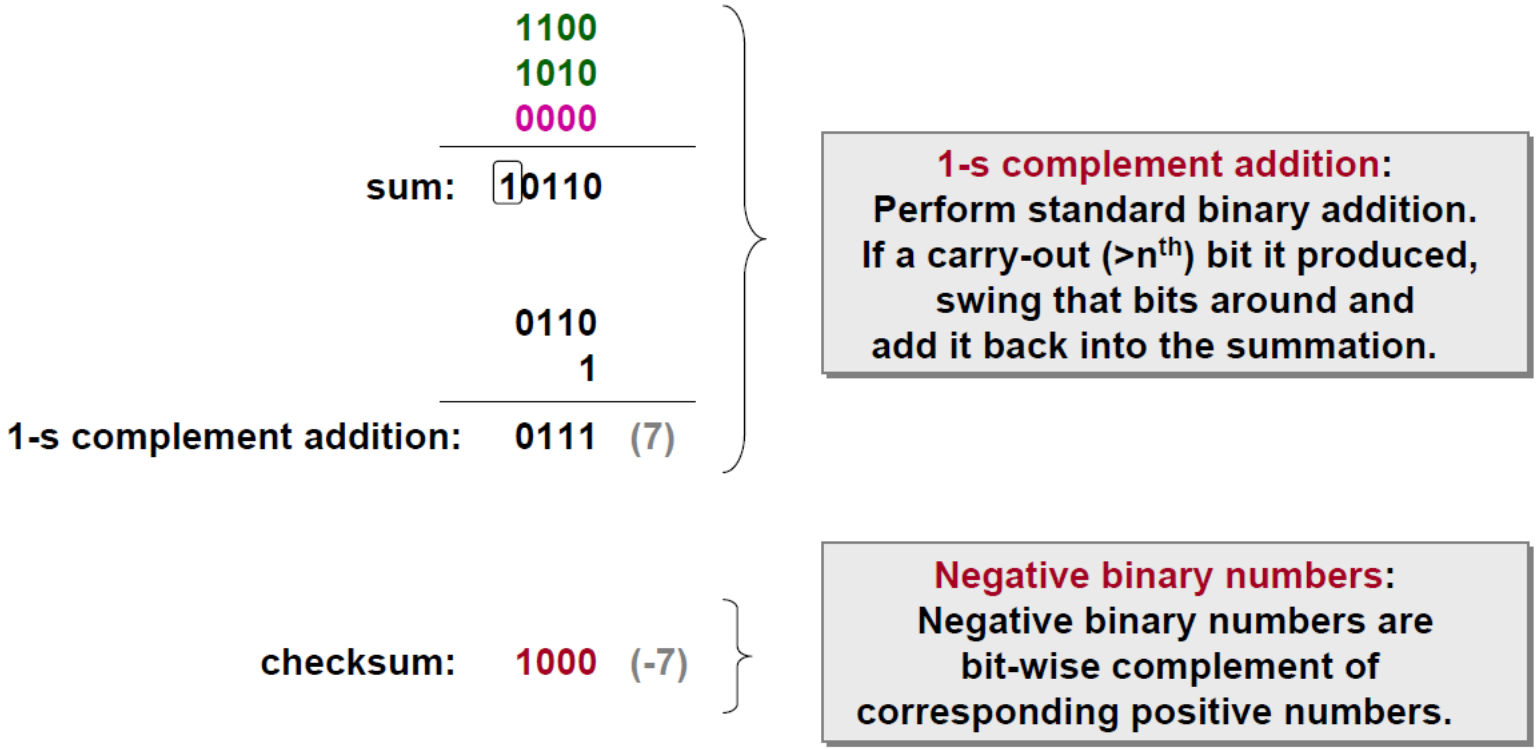
- i dati sono divisi in k sezioni di n bit
- le sezioni sono sommate in complemento a 1
- la somma risultante è complementata bit a bit
- se il risultato è 0, il dato è accettato, altrimenti viene rifiutato



# INTERNET CHECKSUM

- supponiamo che il seguente blocco di 8 bits 1100 1010 sia inviato insieme con una checksum di 4 bit
- calcolare la checksum



# INTERNET CHECKSUM

- supponiamo che il ricevente riceva la sequenza di bit e la checksum non corrotti

| | |
|---|---|
| | 1100 |
| | 1010 |
| | 1000 |
| sum: | 11110 |
| 1-s complement addition: | 1111 |
| bit-wise complement: | 0000 |

- quando il ricevente somma i tre blocchi ottiene una sequenza di 1 che, complementati, danno una sequenza di 0, mostrando che non ci sono stati errori

If one or more bits of a segment are damaged, and the corresponding bit of opposite value in a second segment is also damaged, the sums of those columns will not change and the receiver will not detect the problem. ☹

# INTERNET CHECKSUM

- supporre ora che un blocco di 16 bits 10101001 00111001 venga inviato usando una checksum di 8 bits.
- supporre anche che si sia verificato un errore che ha corrotto 5 bit consecutivi
- il ricevente somma le tre sezioni, ottenendo:

10101111 11111001 00011101

| | |
|---|---|
| | 10101111 |
| | 11111001 |
| | 00011101 |
| Partial Sum | 1 11000101 |
| Checksum | 11000110 |
| Complement | 00111001 the pattern is corrupted. |

# IL PROTOCOLLO STOP AND WAIT: IDEA DI BASE

- come suggerisce il nome, il mittente nel protocollo Stop-And-Wait
  - invia un pacchetto al ricevente
  - quindi si ferma e aspetta di ricevere un acknowledgement
  - quindi passa al prossimo pacchetto

- si assicura che ogni pacchetto sia stato ricevuto correttamente prima di iniziare la trasmissione del pacchetti successivo

- protocollo semplice, ma adatto come punto iniziale per la comprensione dei protocolli sliding window, simili al protocollo reale adottato da TCP

# IL PROTOCOLLO STOP AND WAIT: IDEA DI BASE

- Protocolli ARQ (Automatic Repeat ReQuest): il destinatario individua gli errori nel pacchetto e richiede al mittente di ripetere la trasmissione di quei Pacchetti.

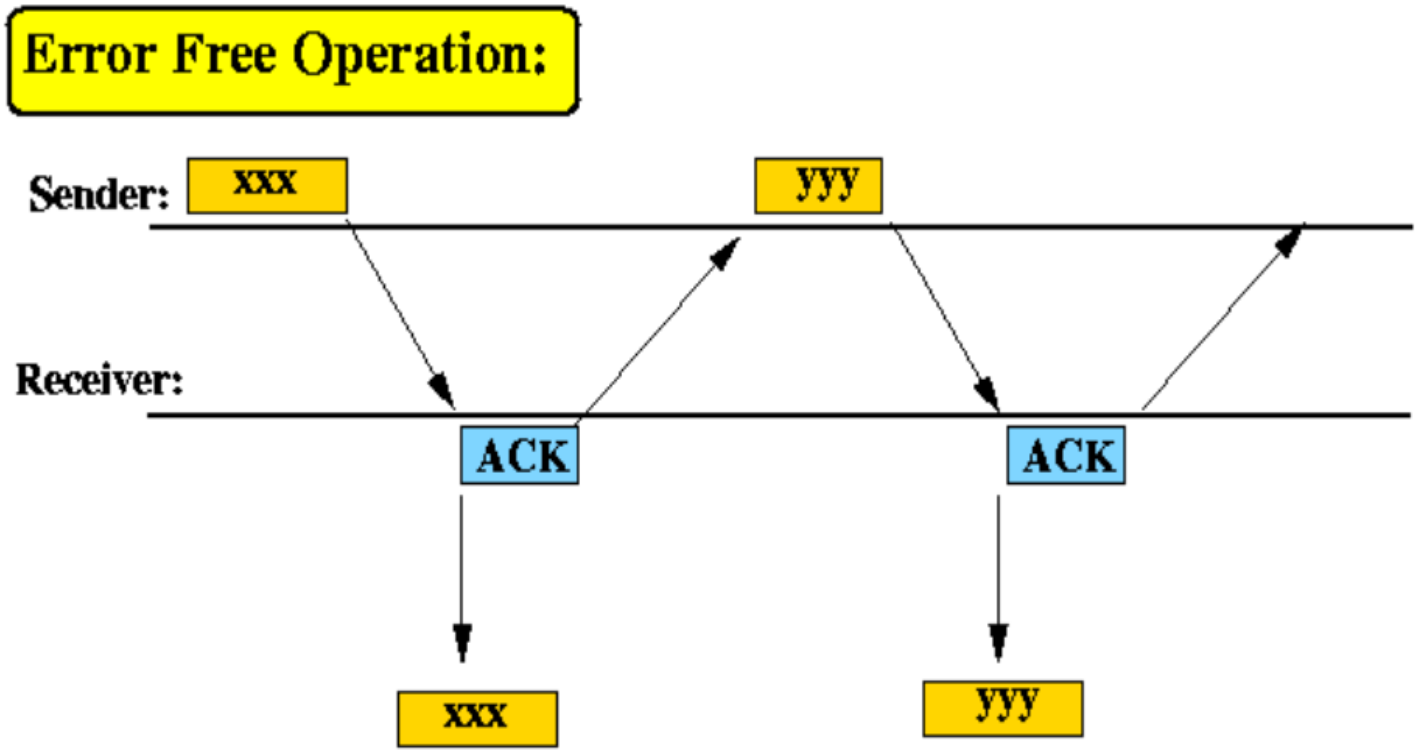
- in questa lezione:
  - Stop-and-Wait
  - Go-Back N
  - Selective Repeat

- Il protocollo implementato in TCP ha aspetti ripresi sia da Go-Back N che da Selective Repeat

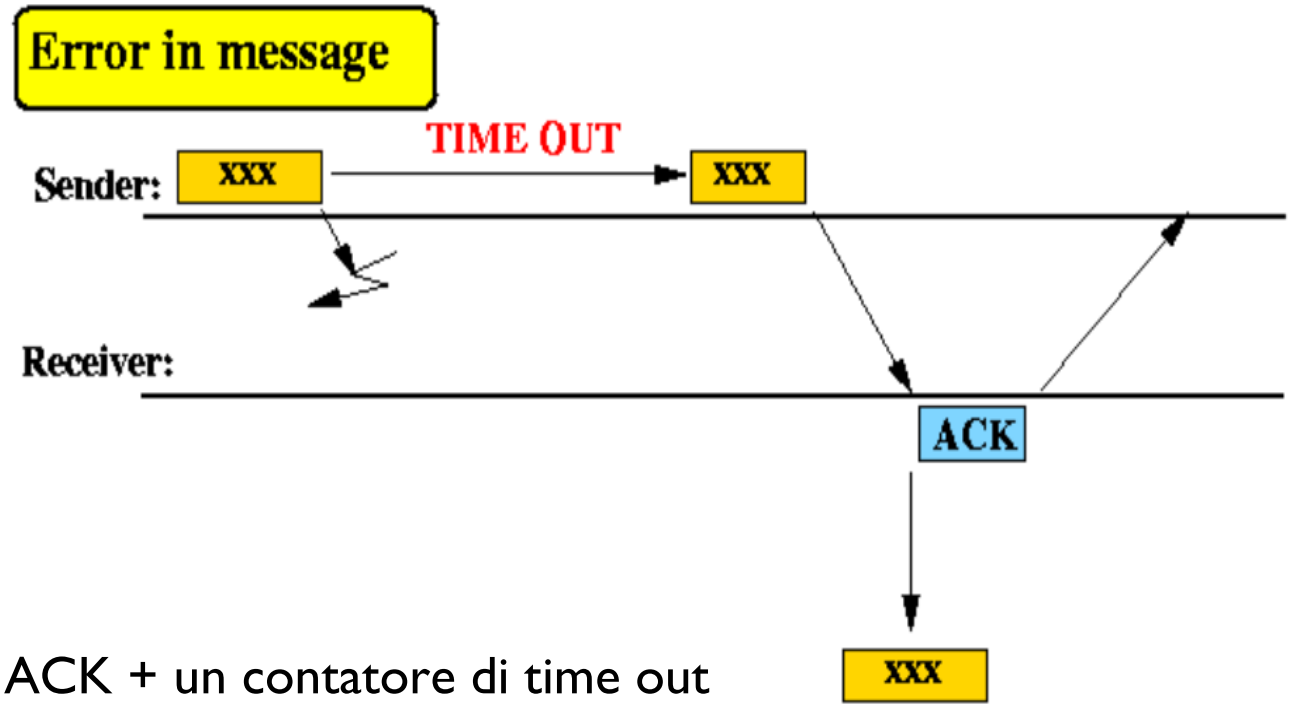
# IL PROTOCOLLO STOP AND WAIT

- utilizza riscontri ACK per assicurare la corretta di ricezione del pacchetto
  - sia il pacchetto contenente i dati che quello contenente i riscontri possono essere corrotti
  - richiede un meccanismo di controllo degli errori nelle due direzioni
- per garantire l'affidabilità, non può utilizzare meccanismi basati sulla analisi del contenuto del pacchetto
  - non può analizzare se due pacchetti successivi sono identici per individuare i duplicati
    - violazione del layering principle
    - non possibile affidarsi ai livelli più alti (ad esempio il livello applicazione) per garantire proprietà del livello inferiore.
- utilizza:
  - ACK
  - Time-out
  - Numeri di sequenza

# STOP AND WAIT SENZA ERRORI

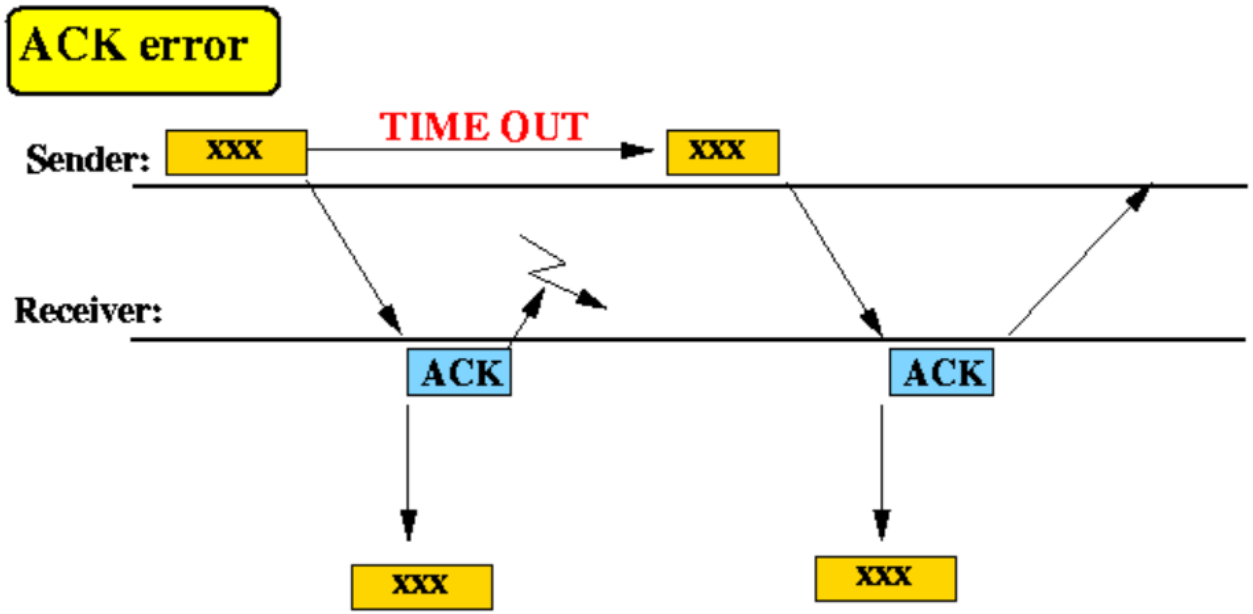


# STOP AND WAIT: PERDITA PACCHETTO



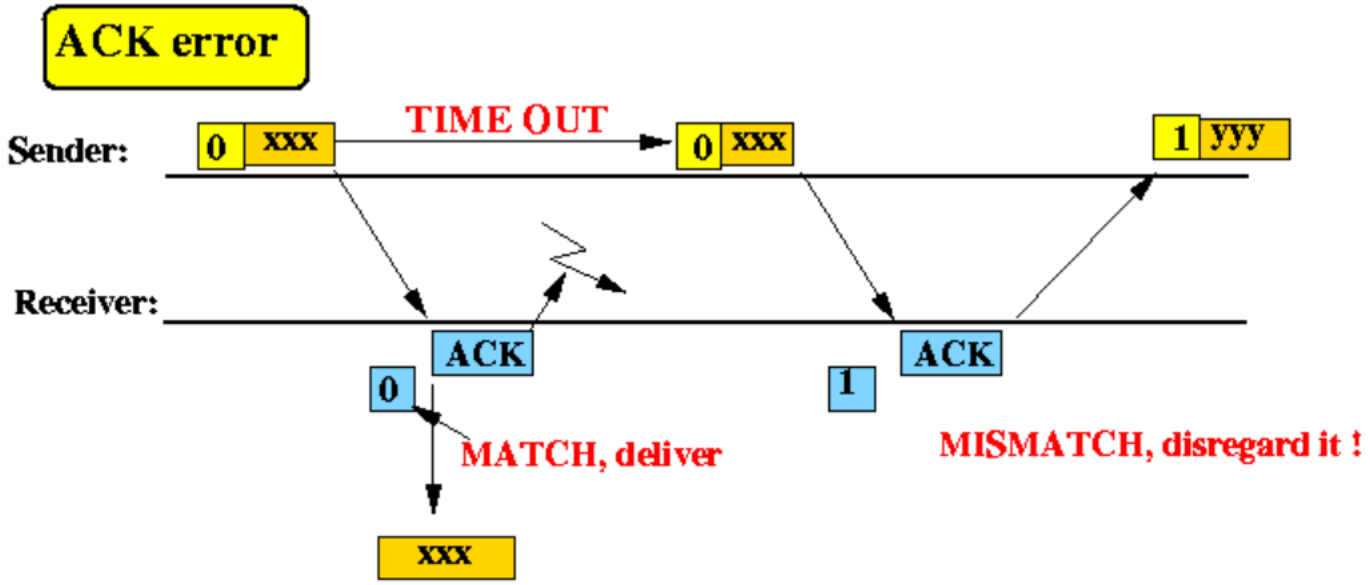

- utilizza ACK + un contatore di time out
- il mittente trasmette un pacchetto e inizializza un contatore (time-out), quindi si ferma in attesa di un ACK:
- se il primo evento successivo è
  - un ACK (corretto), trasmette il pacchetto successivo
  - il time-out, ritrasmette il pacchetto corrente
- il protocollo è in grado di "riparare gli errori" lato mittente

# STOP AND WAIT: PERDITA ACK



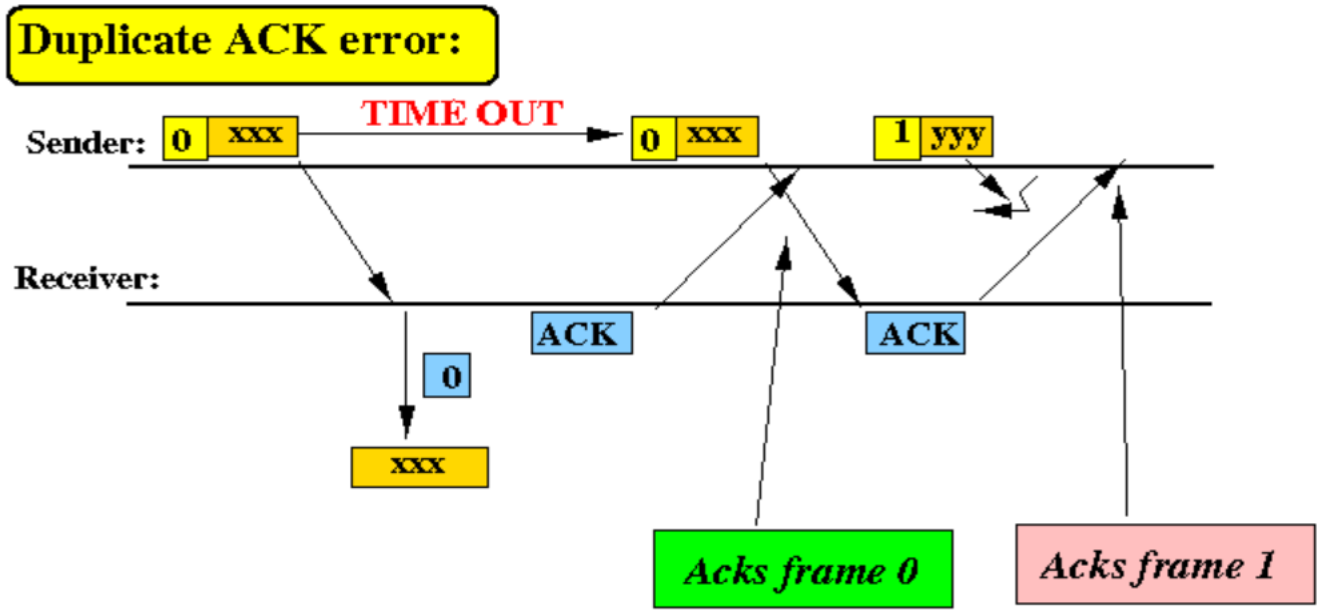

- il destinatario accetta il messaggio due volte di seguito!
- quale è il problema?
  - il mittente si comporta nello stesso modo in questo caso e nel caso precedente: ritrasmette il pacchetto
- il ricevente:
  - nel caso precedente riceve il pacchetto una sola volta
  - in questo caso lo riceve due volte
- il protocollo non è capace di "riparare" errori lato destinatario

# NUMERI DI SEQUENZA: PERCHE'?



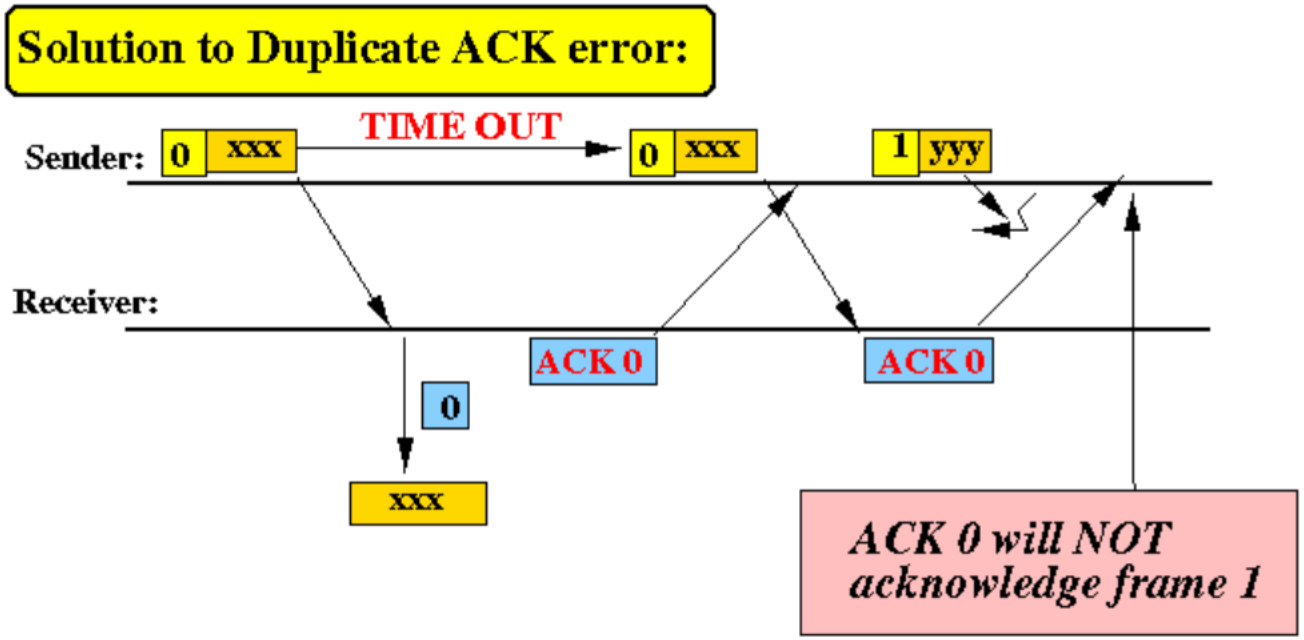

- numeri di sequenza:
  - numeri di sequenza aggiunti nell'header del pacchetto spedito dal mittente
  - il ricevente ha anche esso un numero di sequenza, che identifica il prossimo messaggio che il ricevente si aspetta
  - un messaggio con numero di sequenza sbagliato NON viene accettato
- questo meccanismo consente di distinguere i due casi: errore lato mittente o destinatario
- questo rende il protocollo affidabile? ...*non ancora*.....

# STOP AND WAIT: ACK DUPLICATI




- time-out prematuro nel mittente (o ACK ritardato): può comportare interpretazione errata degli ACK
- scade time out: mittente rimanda il pacchetto 0 perchè pensa che il primo pacchetto sia andato perso
- il pacchetto 1 viene perso
- se non ci sono numeri di sequenza nell'ACK, il mittente pensa che il secondo pacchetto 0 sia perso ed attribuisce il secondo ACK al pacchetto 1
  - timeout prematuri ( o ACK ritardati) combinati con la perdita di pacchetti possono produrre mancata consegna di pacchetti al destinatario

# STOP AND WAIT: ACK DUPLICATI




Soluzi

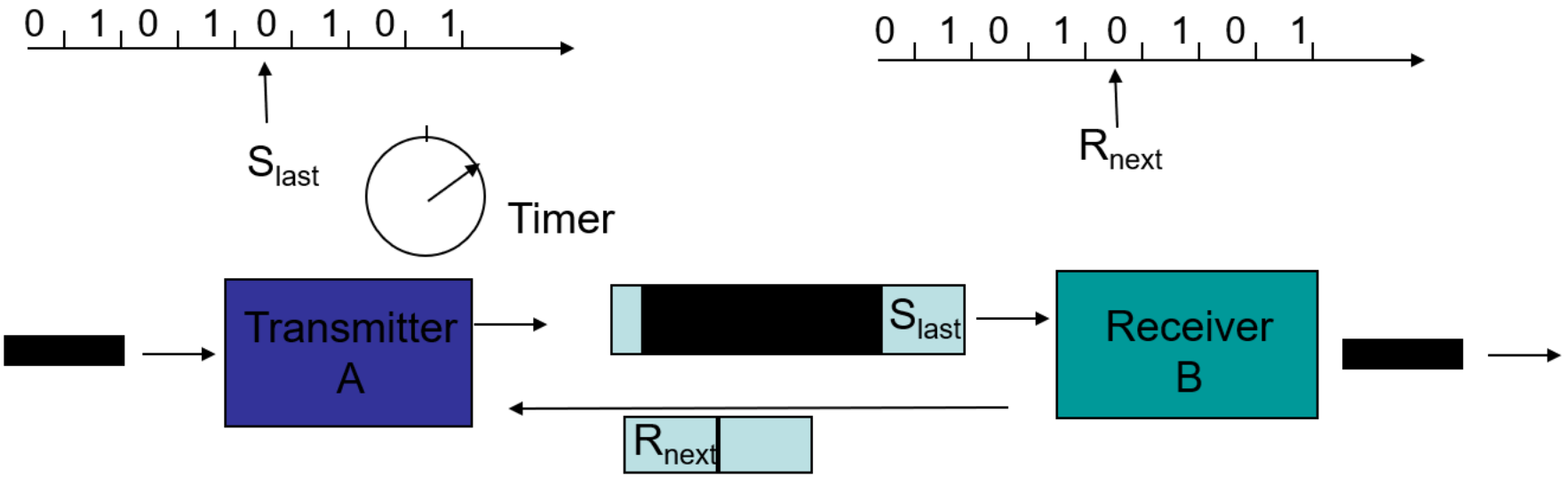
- aggiungere un numero di sequenza anche agli ACK:
  - numero di sequenza del pacchetto ricevuto
  - A riceve due ACK per il pacchetto 0, scarta il secondo
  - A non attribuisce il secondo ACK al pacchetto 1

# STOP AND WAIT: NUMERI DI SEQUENZA

Numerazione dei pacchetti (S) e degli ACK (R)

- se un pacchetto viene trasmesso più volte (anche erroneamente)
  - il ricevente può riconoscere la duplicazione perché i pacchetti hanno lo stesso valore di S
- quando il mittente riceve un ACK lo assegna al pacchetto corretto
  - considerando R può assegnarlo al pacchetto con il rispettivo S
- quanti bit per i numeri di sequenza?
  - un numero di sequenza non può essere arbitrariamente grande, perchè deve essere inserito nell'header del frame
  - nel caso dello Stop&Wait
    - se la rete mantiene l'ordinamento
    - è sufficiente un solo bit per il numero di sequenza, perchè?

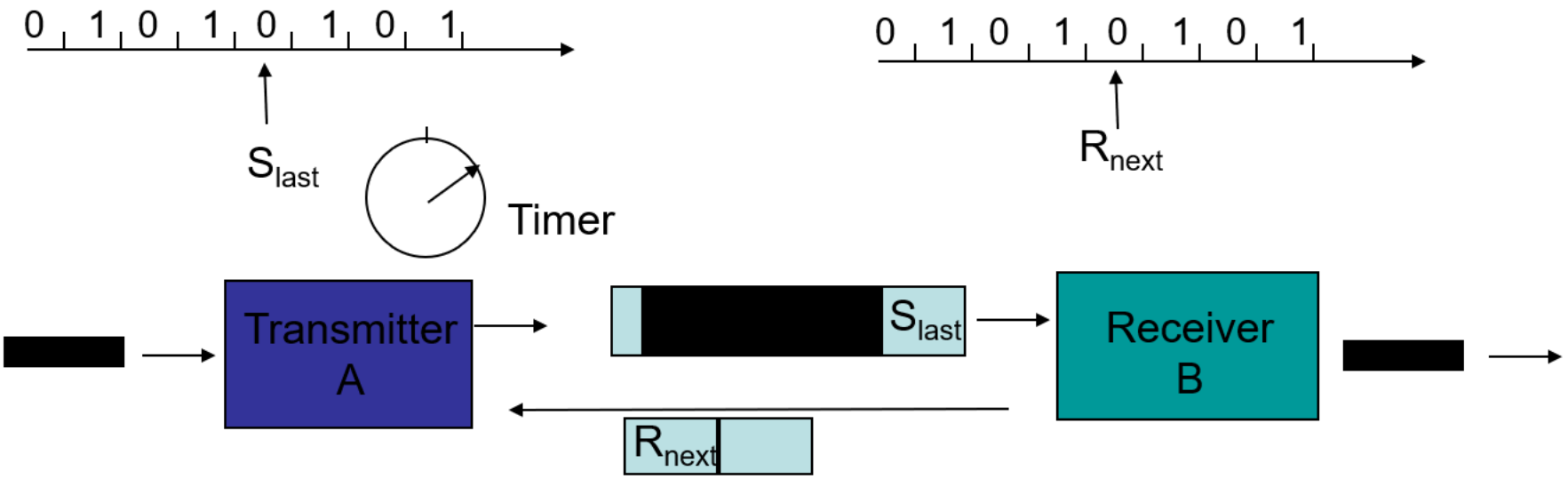
# NUMERI DI SEQUENZA DI UN BIT SONO SUFFICIENTI....



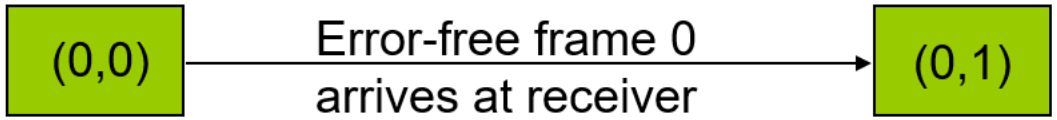

Global State:
($S_{last}$, $R_{next}$)

(0,0)

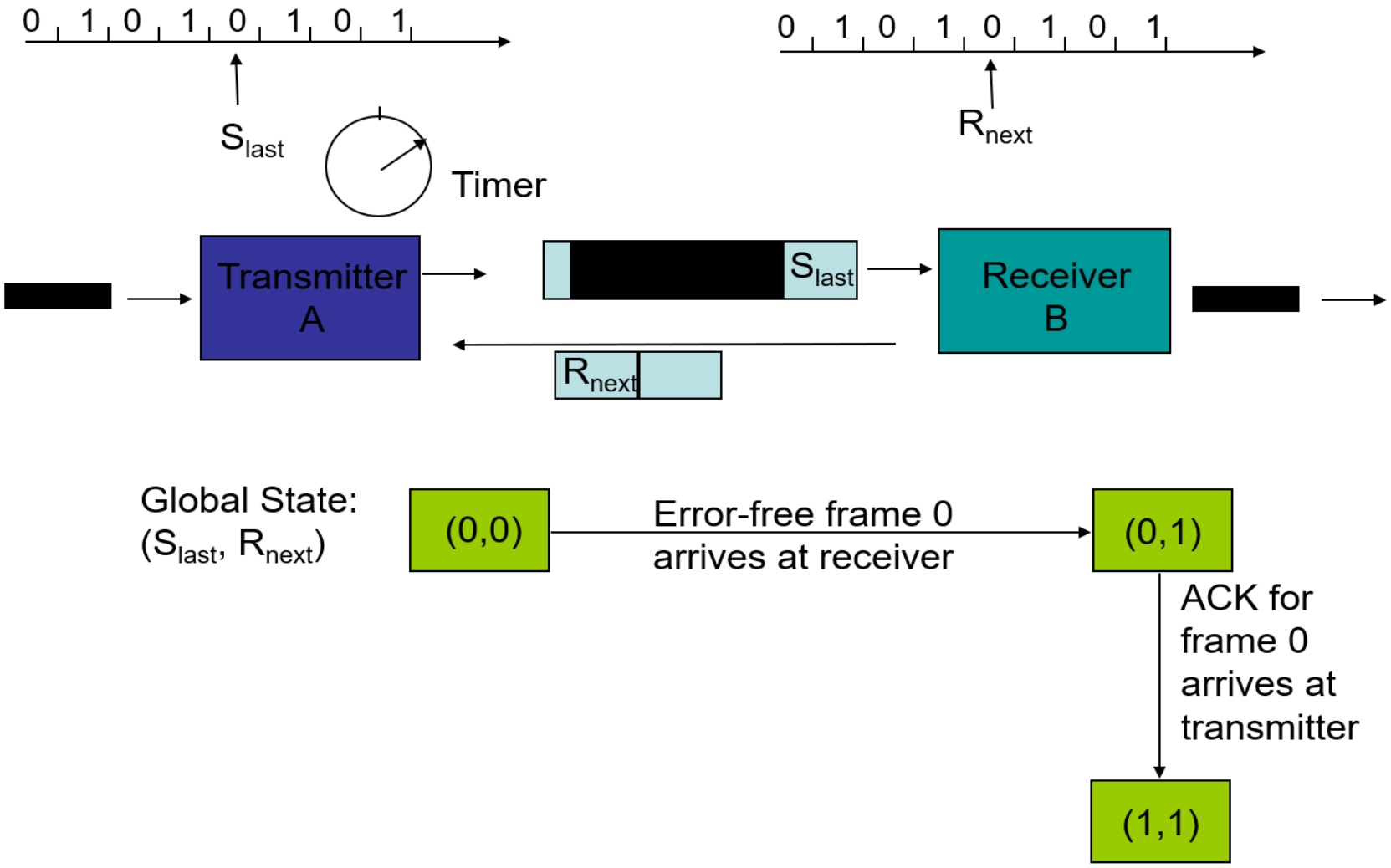
# NUMERI DI SEQUENZA DI UN BIT SONO SUFFICIENTI....



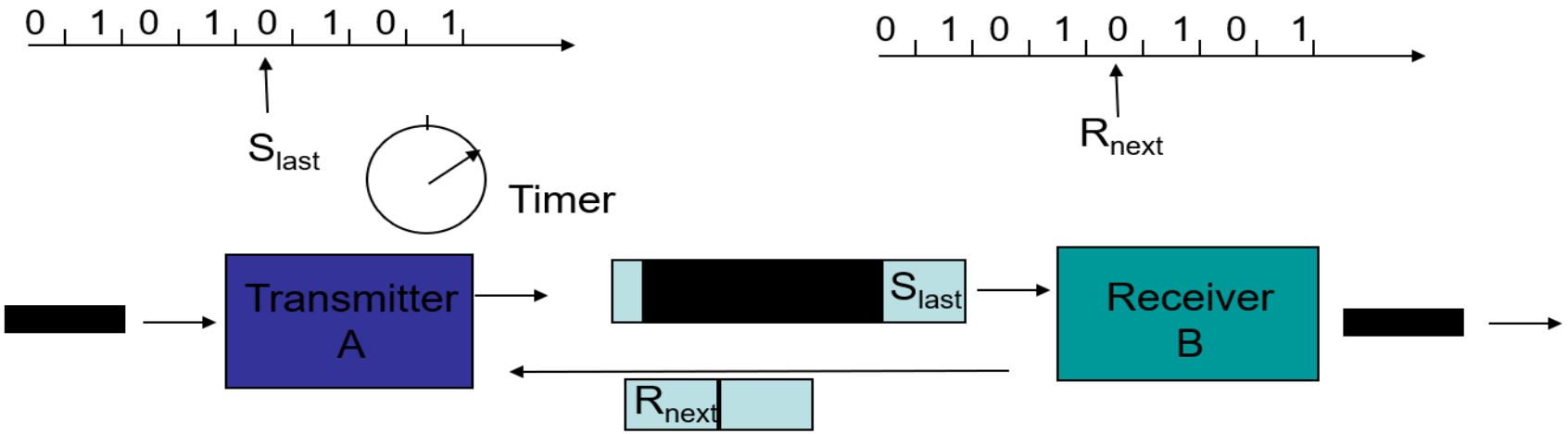

Global State: ($S_{last}$, $R_{next}$)

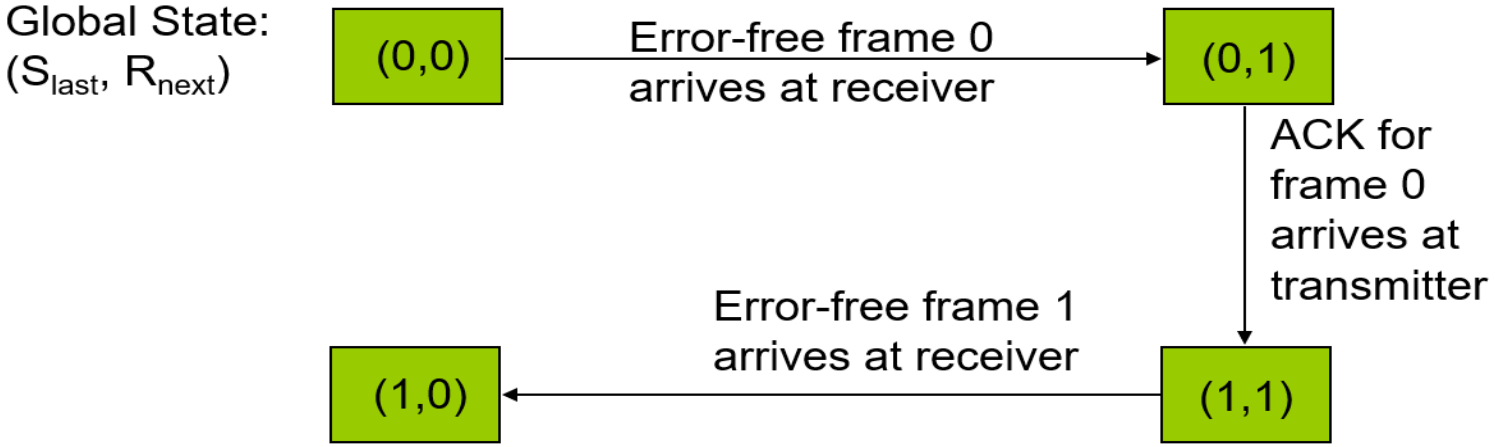


# NUMERI DI SEQUENZA DI UN BIT SONO SUFFICIENTI....

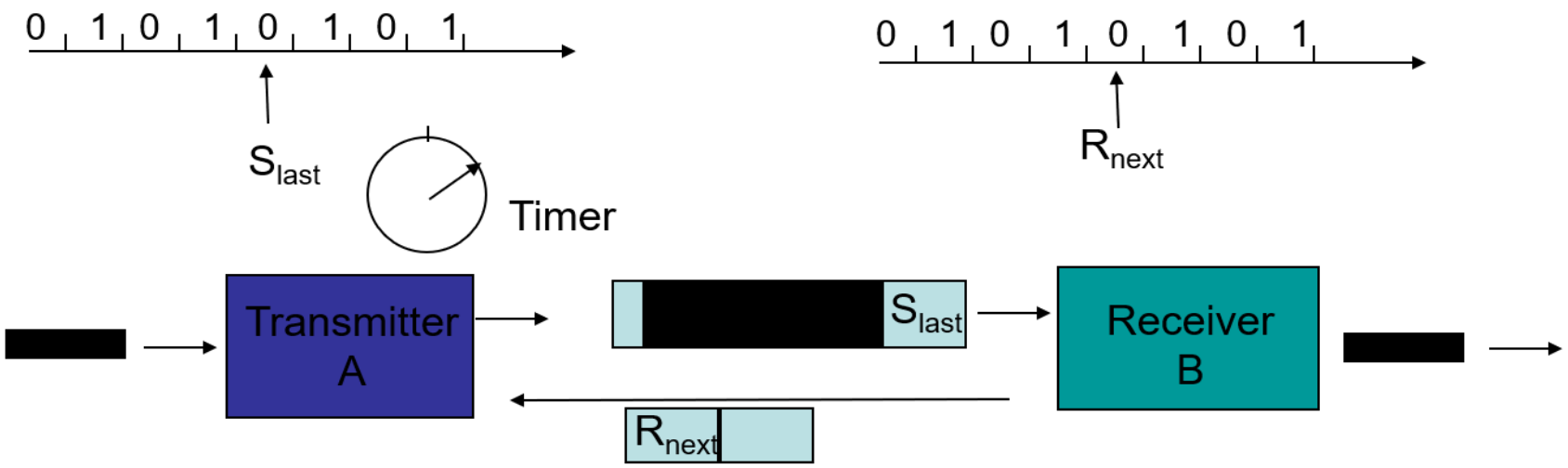


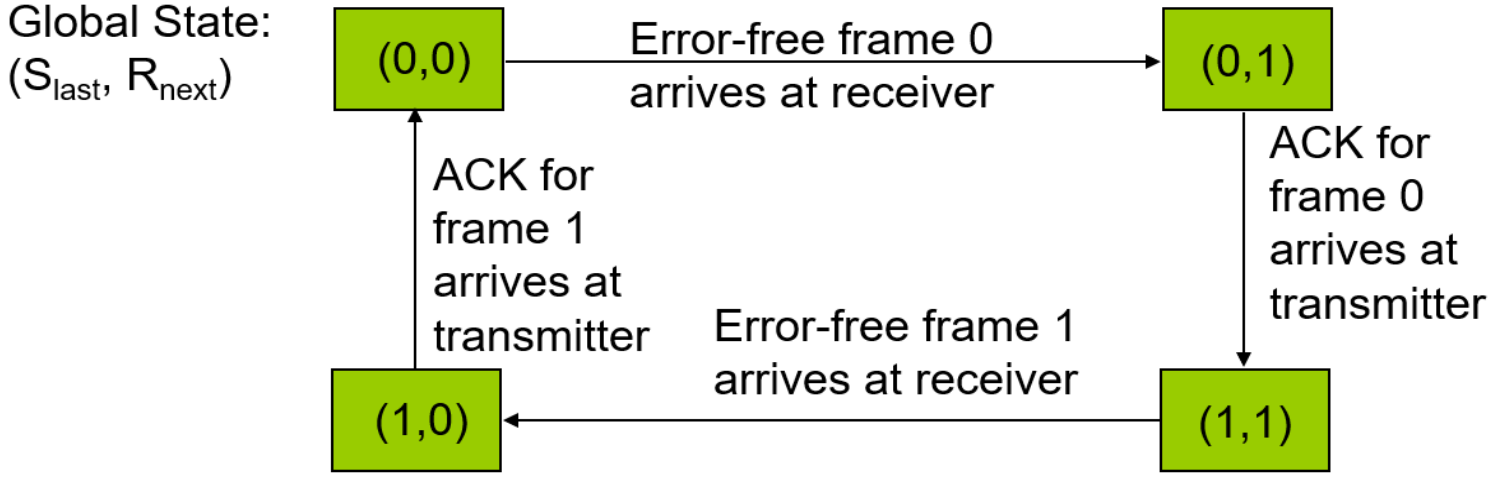
# NUMERI DI SEQUENZA DI UN BIT SONO SUFFICIENTI....



# NUMERI DI SEQUENZA DI UN BIT SONO SUFFICIENTI....



# STOP AND WAIT ARQ (MITTENTE)

Stato **Ready**

- attesa di una richiesta di invio di un pacchetto dallo strato superiore
- quando arriva una richiesta, trasmette il pacchetto con numero di sequenza $S_{last}$
- transizione nello stato Wait

Stato **Wait**

- attesa del riscontro del pacchetto emesso o dell'esaurimento del timeout (la ricezione delle richieste dallo strato superiore sono bloccate)
- se il timeout scade viene ritrasmesso il pacchetto e riavviato il timer
- se viene ricevuto un ACK
  - se il numero di sequenza non è corretto oppure il pacchetto è corrotto, l'ACK è ignorato
  - se il numero di sequenza (dell'ACK) è corretto ($R_{next} = S_{last} + 1$), il pacchetto è accettato, il timer arrestato e $S_{last} = S_{last} + 1$. Si torna nello Stato Ready.

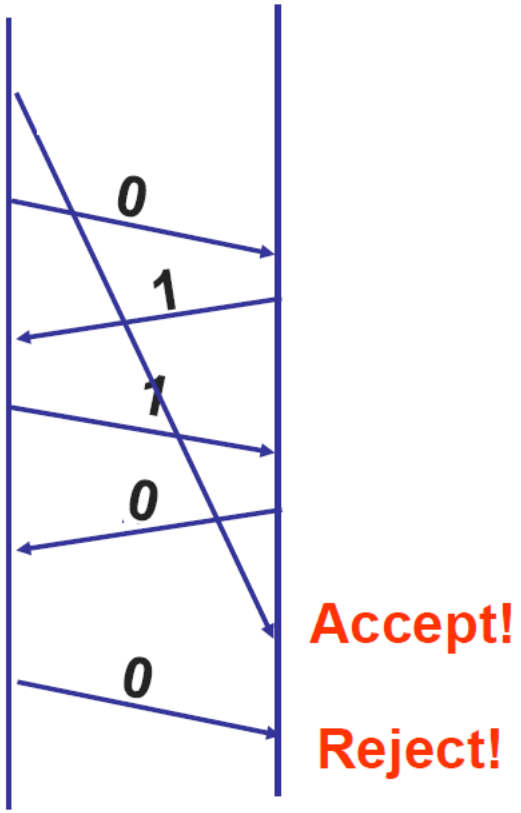
# STOP AND WAIT ARQ (DESTINATARIO)

sempre nello stato **Ready**

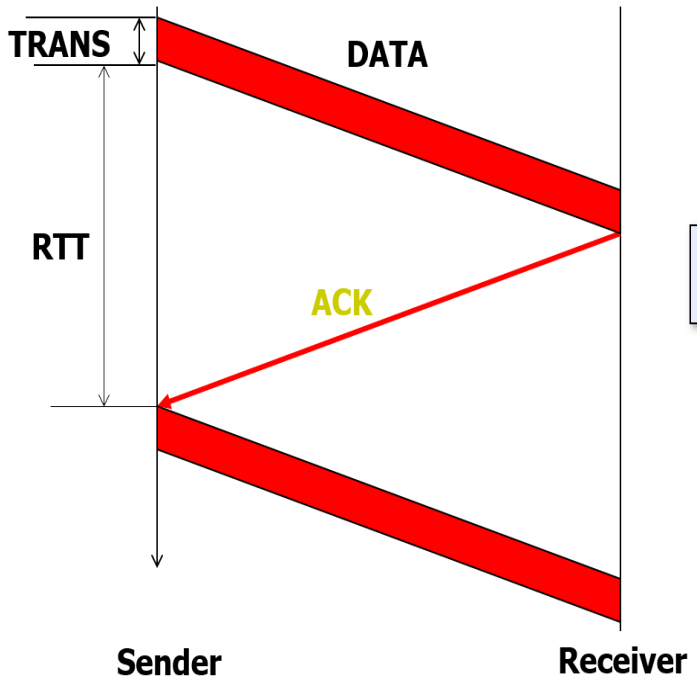
- attesa dell'arrivo di un nuovo pacchetto
- quando arriva un pacchetto viene eseguito il controllo d'errore
- se non sono rivelati errori e il numero di sequenza è corretto ($S_{last} = R_{next}$),
  - il pacchetto è accettato e consegnato allo strato superiore
  - viene aggiornato il valore di $R_{next} = R_{next} + 1$
  - viene inviato l'ACK con valore $R_{next}$
- se non sono rivelati errori, ma il numero di sequenza non è corretto il pacchetto è scartato
  - viene emesso un ACK with $R_{next}$ (ACK duplicato)
- se sono stati rivelati errori, il pacchetto è scartato

# STOP AND WAIT: RIORDINAMENTO

- i pacchetti possono subire larghi ritardi
  - ordinamento non garantito
  - un solo bit non è più sufficiente
- soluzioni possibili:
  - usare numeri di sequenza #seq con un range di valori più ampio
  - non riusare mai lo stesso #seq
  - range molto ampio per #seq approssima la generazione di numeri di sequenza sempre nuovi



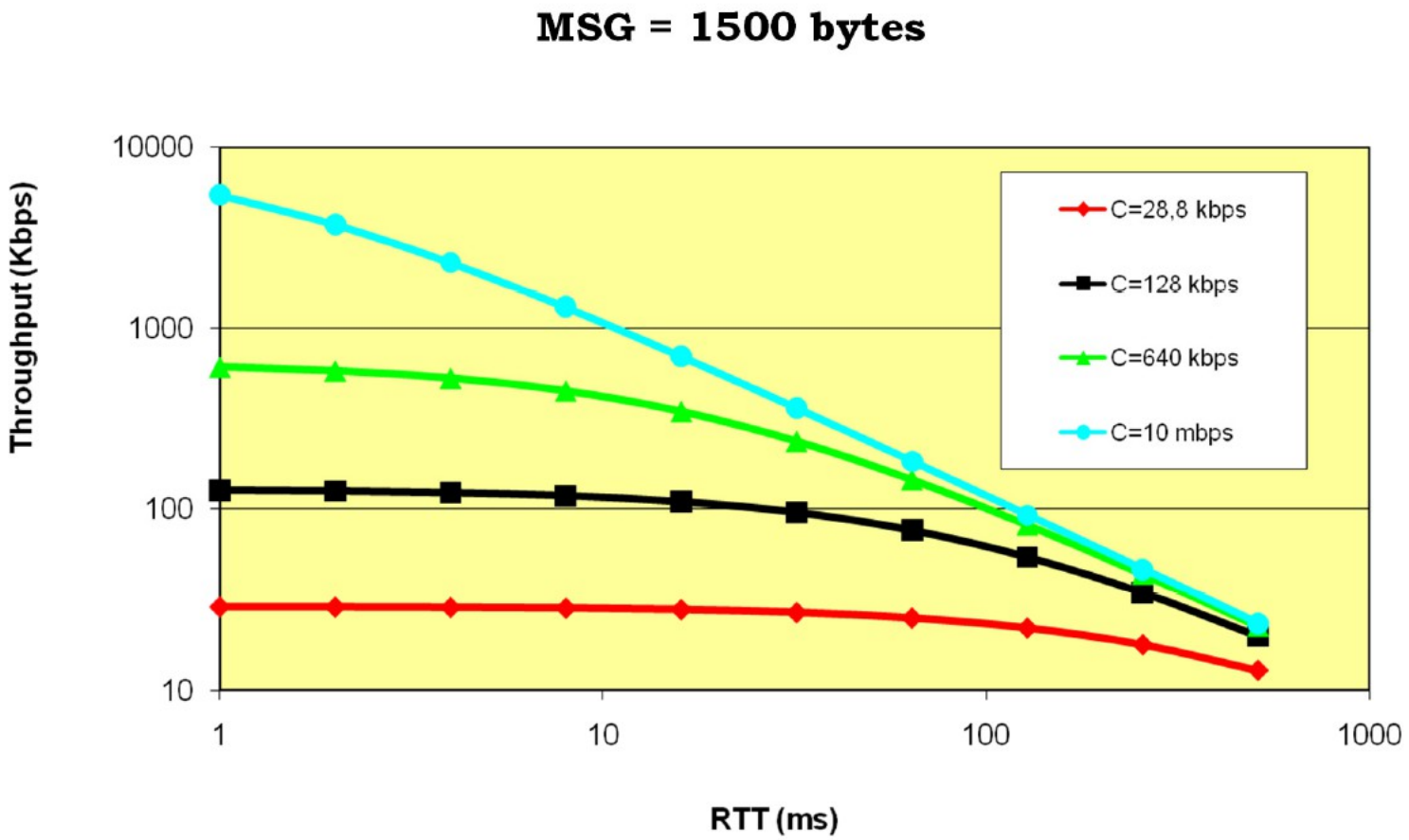
# STOP AND WAIT: THROUGHPUT




Inefficient if TRANS << RTT

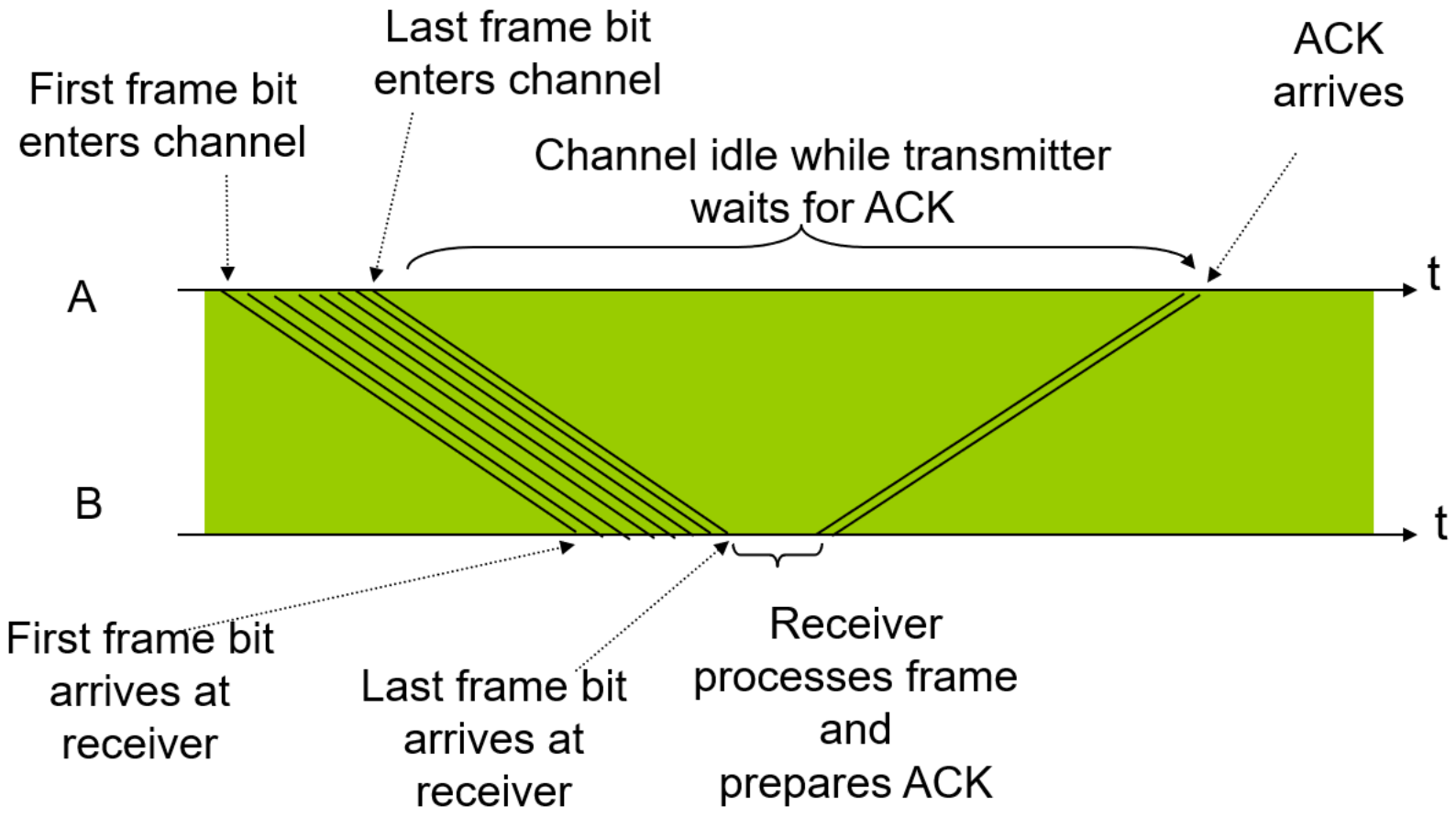
- un pacchetto per ogni "round" invio-acknowledgment
- se si assume che le perdite di pacchetti siano relativamente rare
  - un round impiega circa un RTT
- consideriamo un Wi-Fi link con capacità 10 Mbit/s. Supponiamo:
  - che trasporti 1460 bytes di dati TCP in ogni pacchetto.
  - che il round-trip time tra qui e Milano sia circa 100 ms.
- throughput: 1460 bytes ogni 100 ms, 116 kbit/s.
  - si utilizza circa l'1% della capacità del link

# STOP AND WAIT: THROUGHPUT



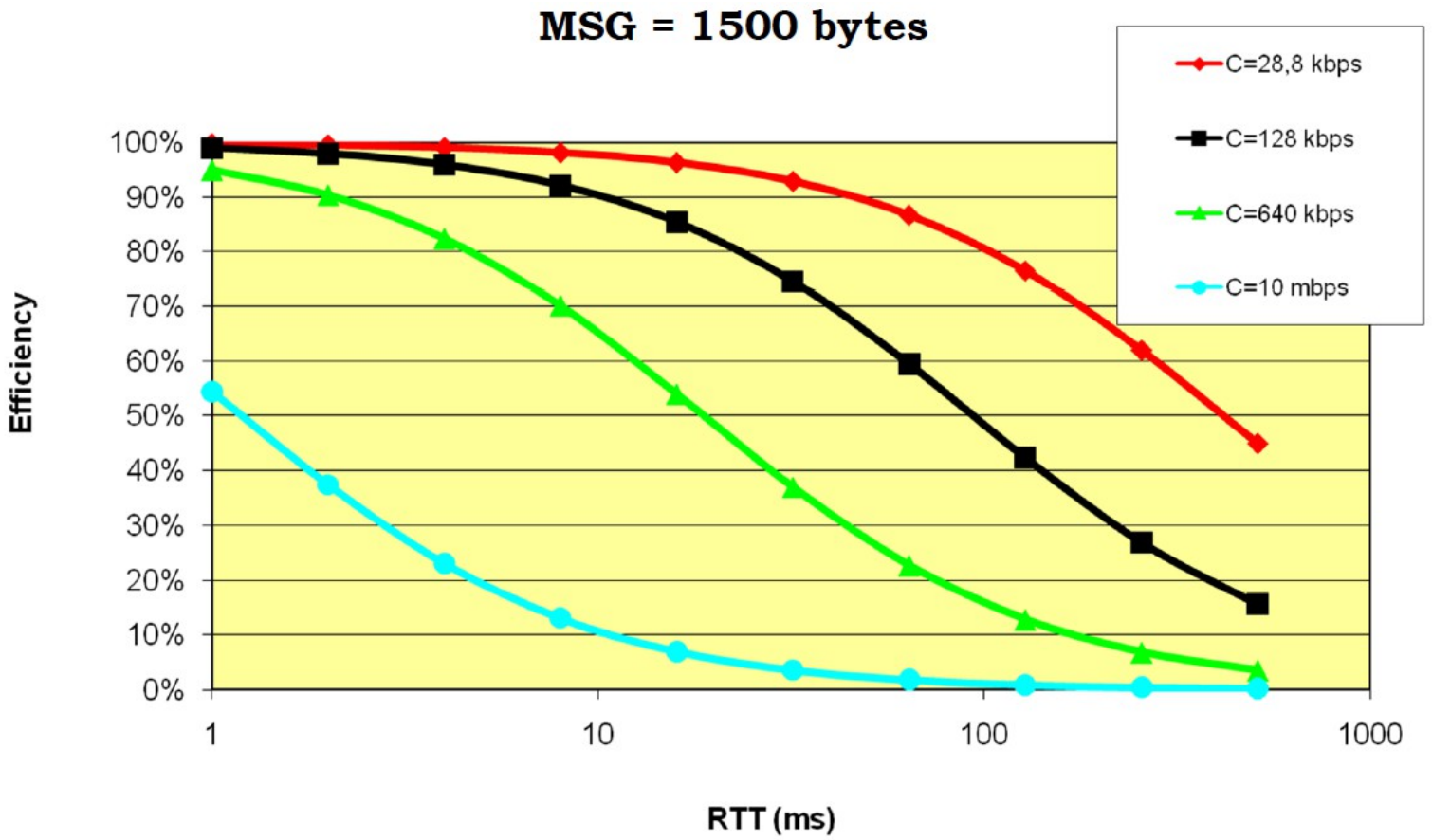

Under-utilization with: 1) high capacity links, 2) large RTT links

# STOP AND WAIT: EFFICIENZA




- pacchetto di 10000 bit @1Mbps : 10 ms per la trasmissione del pacchetto
- se latenza spedizione ACK= 1ms, efficienza 10/11= 91%
- se latenza spedizione ACK=20ms, efficienza 10/30=33%

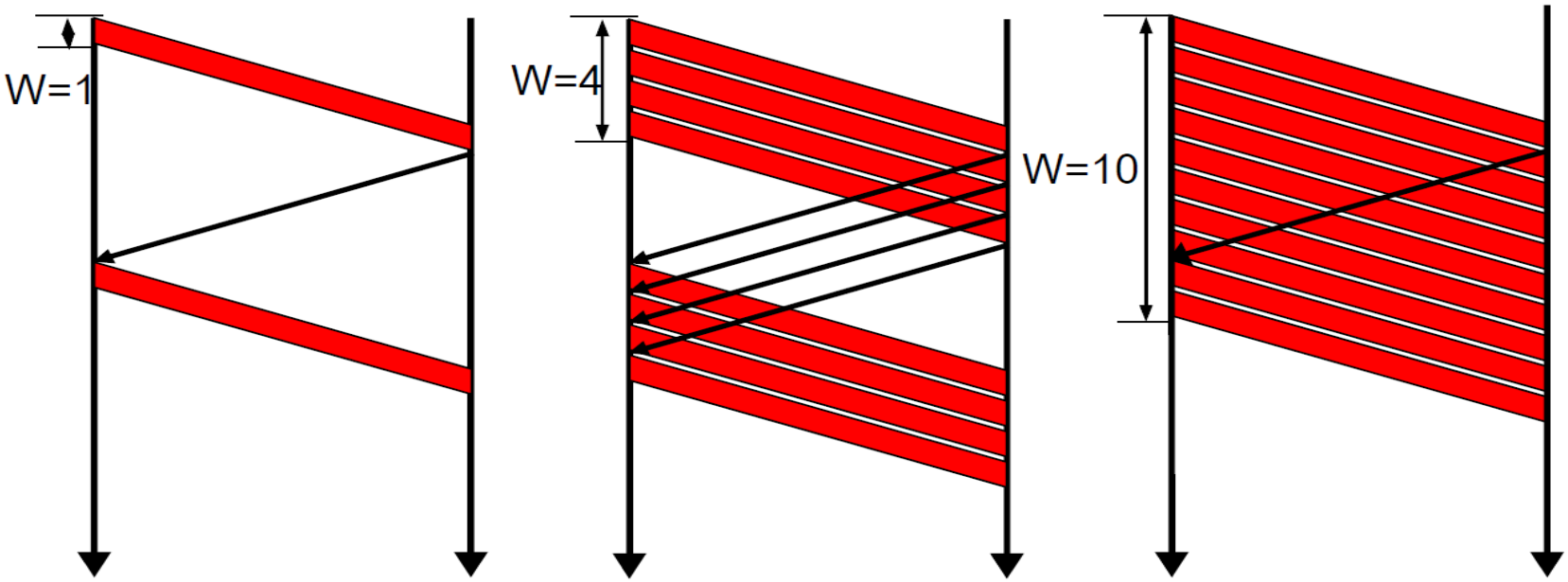
# STOP AND WAIT: EFFCIENCY




Under-utilization with: 1) high capacity links, 2) large RTT links

# PIPELINING

- una procedura in cui un task viene iniziato prima che il task precedente sia completato




- per superare l'inefficienza di Stop&Wait ARP:
  - inviare fino a W pacchetti prima di ricevere il primo ACK!
  - overlap tra trasmissione di pacchetti e ricezione di ack
  - evitare periodi di attesa dei riscontri, in cui non si ha alcuna trasmissione
  - mantenere il canale sempre occupato

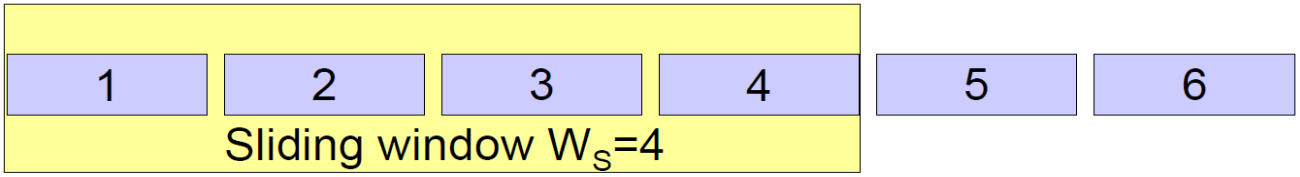
# PIPELINING

- sliding window protocols:
  - i più importanti e generali
  - basati su pipelining
  - utilizzati da TCP

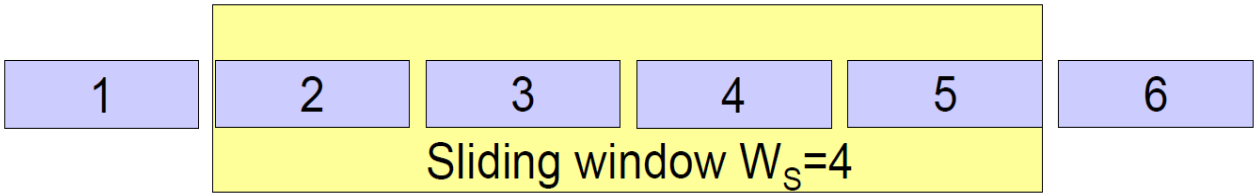
- approcci esistenti
  - Go-Back-N
    - Ack cumulativi
  - Selective Repeat
    - ACK selettivi
  - TCP utilizza un sistema ibrido
  - + mille altre varianti che differiscono nei dettagli implementativi
    - se fate una ricerca su Google ve ne potete rendere conto!

# SLIDING WINDOW: PRINCIPIO BASE

- in tutti i protocolli basati su sliding window:
  - window: intervallo di numeri di sequenza consecutivi
  - contiene tutti i pacchetti spediti, ma non ancora riscontrati + eventuali numeri di sequenza disponibili
- esempio:
  - il mittente trasmette consecutivamente i pacchetti numerati da 1 a 4




  - la finestra "scorre" (sliding window) quando il mittente ottiene uno o più riscontri positivi dal destinatario




    - $W_S$: dimensione finestra($W_S$ numeri di sequenza adiacenti)
    - scopo: cercare di mantenere la finestra sempre piena

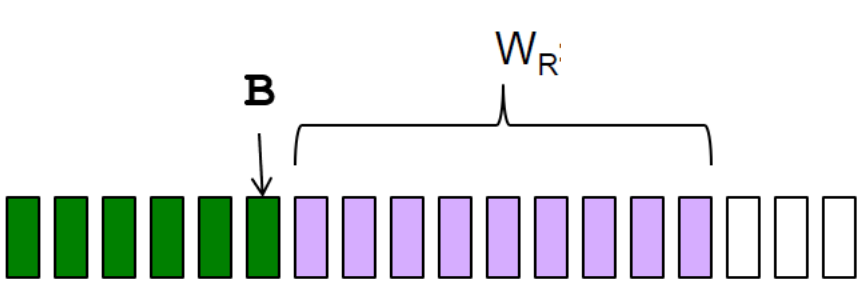
# SLIDING WINDOW

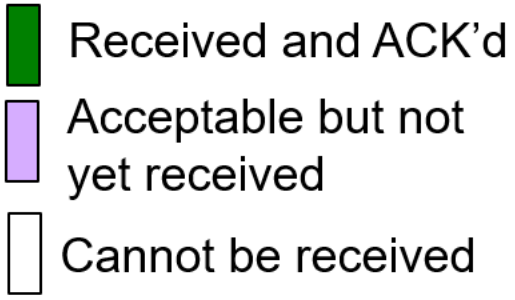
- il mittente ed il destinatario mantengono delle finestre, eventualmente di dimensione diversa
- i pacchetti e gli ack vengono numerati progressivamente
- se i numeri di sequenza sono di k bits
  - i pacchetti sono numerati 0, 1, 2, …, $2^k$-1, 0, 1, … (modulo $2^k$)
  - gli ACK sono numerati 0, 1, 2, …, $2^k$-1, 0, 1, … (modulo $2^k$)
  - la dimensione della finestra deve essere minore o uguale a $2^k$-1
- se la dimensione della finestra del mittente è W, il mittente può trasmettere fino a W pacchetti, senza attendere un ACK
- gestione degli ACK:
  - Cumulativi: ACK J, dove 0<=J<= $2^k$-1, implica che il destinatario ha ricevuto tutti i pacchetti fino al pacchetto J-1 ed è pronto per ricevere il pacchetto J
  - Selettivi: ACK J, dove 0<=J<= $2^k$-1, implica che il destinatario ha ricevuto il pacchetto J, ma non da indicazione sui pacchetti precedenti

# ACK CUMULATIVI

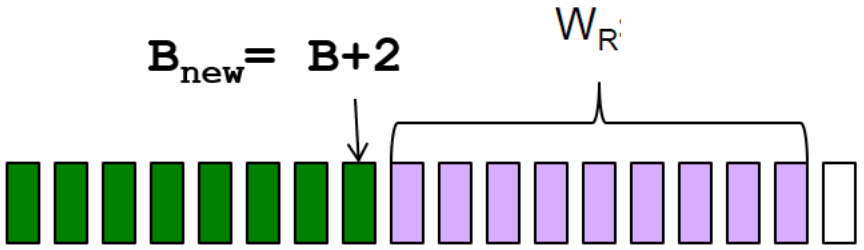
L'ACK inviato dal destinatario contiene il numero di sequenza del prossimo pacchetto "in ordine" atteso (senza "buchi" precedenti).

NEL DESTINATARIO




DOPO AVER RICEVUTO B+1, B+2



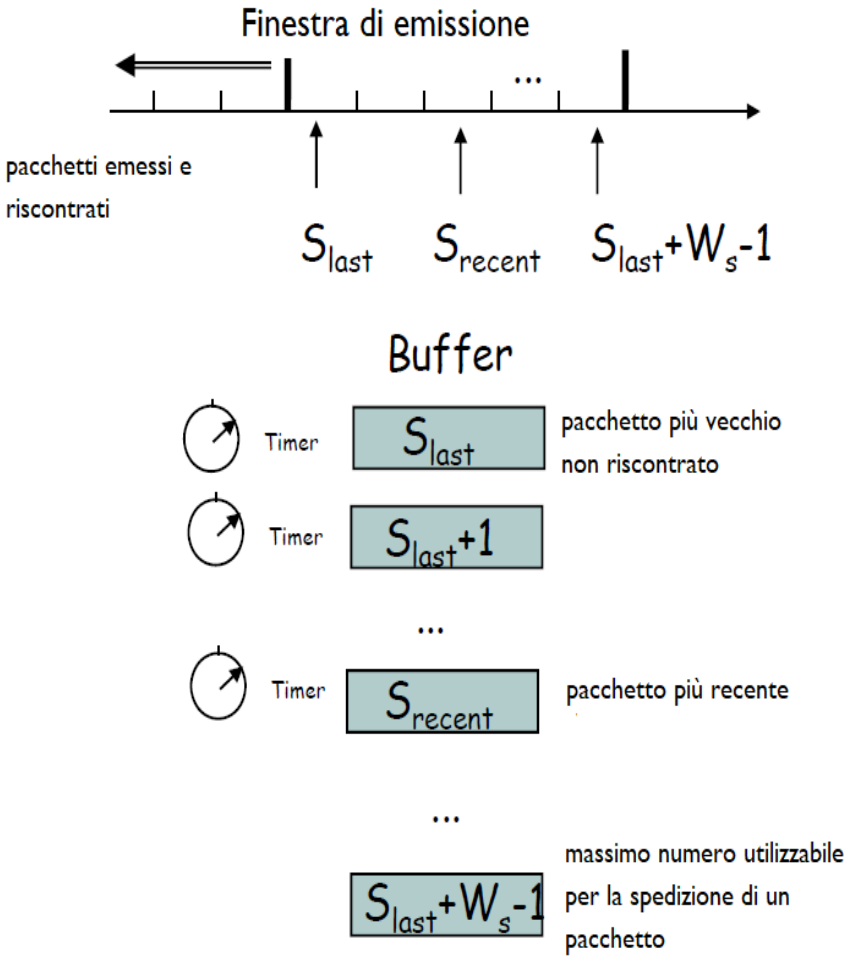

IL DESTINATARIO INVIA $ACK(B_{new}+1)$

# GO-BACK-N IN BREVE

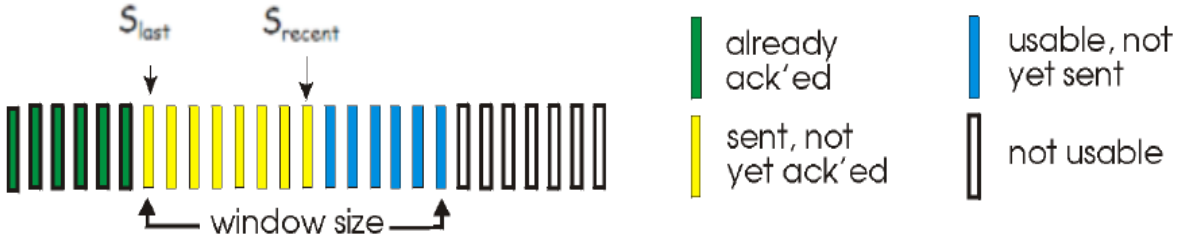
- Il mittente può inviare in pipeline fino ad n pacchetti, dove n è la dimensione della sua window
- Il ricevente invia ack cumulativi
  - non riscontra un pacchetto se c'è un "buco" prima di quel pacchetto
  - scarta quel pacchetto, non lo memorizza
  - il meccanismo dei cumulative ack presenta un meccanismo di correzione degli errori "built in"
    - se si perde l'ACK N, ma l'ACK N+1 è ricevuto, il mittente viene informato dall'ACK ricevuto che il pacchetto N è stato ricevuto
    - Importante in ambienti con alto tasso di errori.
- Il mittente setta un timer associato al pacchetto non riscontrato più vecchio
  - se il timer si esaurisce rinvia tutti i pacchetti non riscontrati

# GO-BACK-N: IL MITTENTE

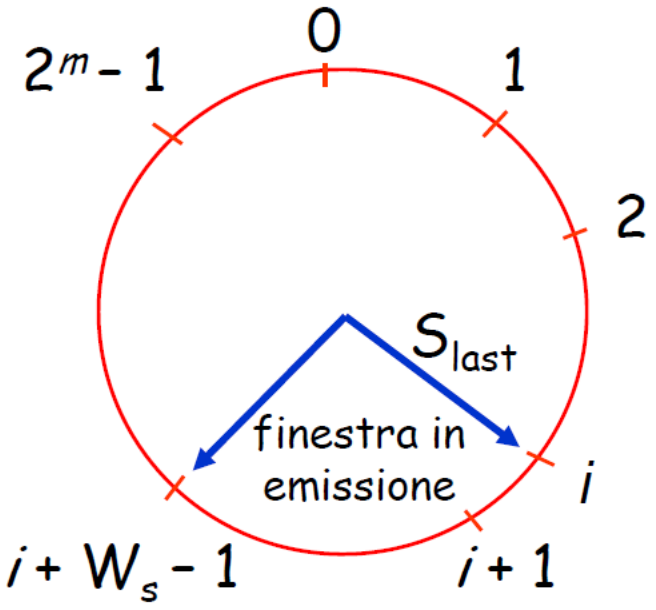
- mantiene una finestra di dimensione $W_s$: può inviare fino a $W_s$ pacchetti senza attendere i rispettivi riscontri
- Finestra gestita mediante tre puntatori
  - $S_{last}$
  - $S_{recent}$
  - + un puntatore calcolato sulla base dei precedenti, che indica la fine della finestra
- se $S_{recent} = S_{last} + W_s - 1$
  - la finestra è esaurita
  - il mittente rimane in attesa degli ACK



# GO-BACK-N: IL MITTENTE

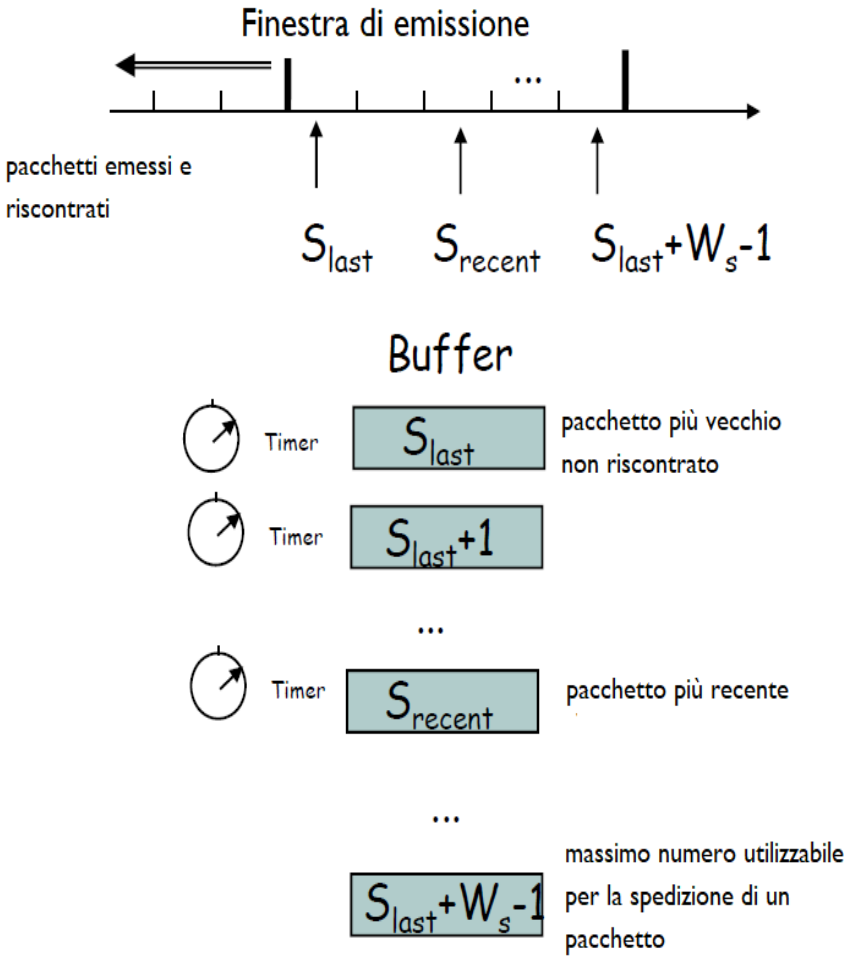



- attende gli ACK con numero di sequenza $S \geq S_{last}$
- quando si riceve un ACK S (cumulativo), la finestra scorre, $S_{last} = S$
- l'estremo superiore della finestra è $S_{last} + W_s + 1$



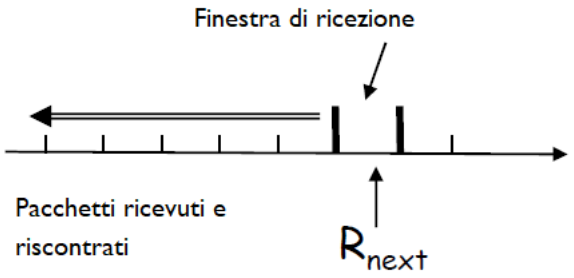
# GO-BACK-N: IL TIMER

- il mittente imposta un timer per ogni pacchetto inviato
  - sufficiente anche un solo timer associato a $S_{last}$
  - il timer del primo pacchetto scatta sempre per primo
  - se il pacchetto i non è stato riscontrato non è stato riscontrato neppure il pacchetto i-1 (ack cumulativi)
- se non si ricevono ACK ed timeout scatta, ritrasmette tutti i pacchetti non ancora riscontrati

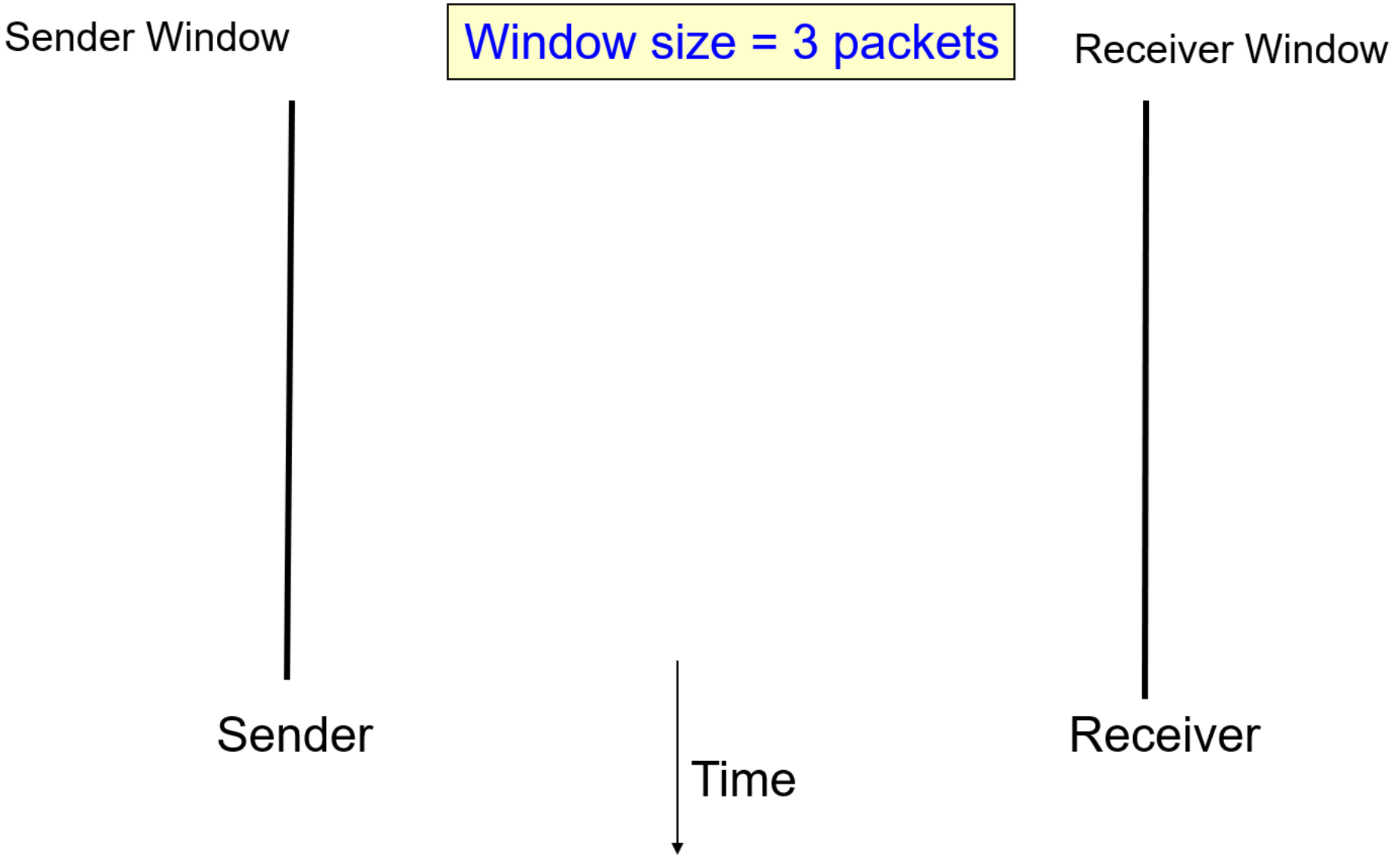


# GO-BACK-N: IL RICEVENTE

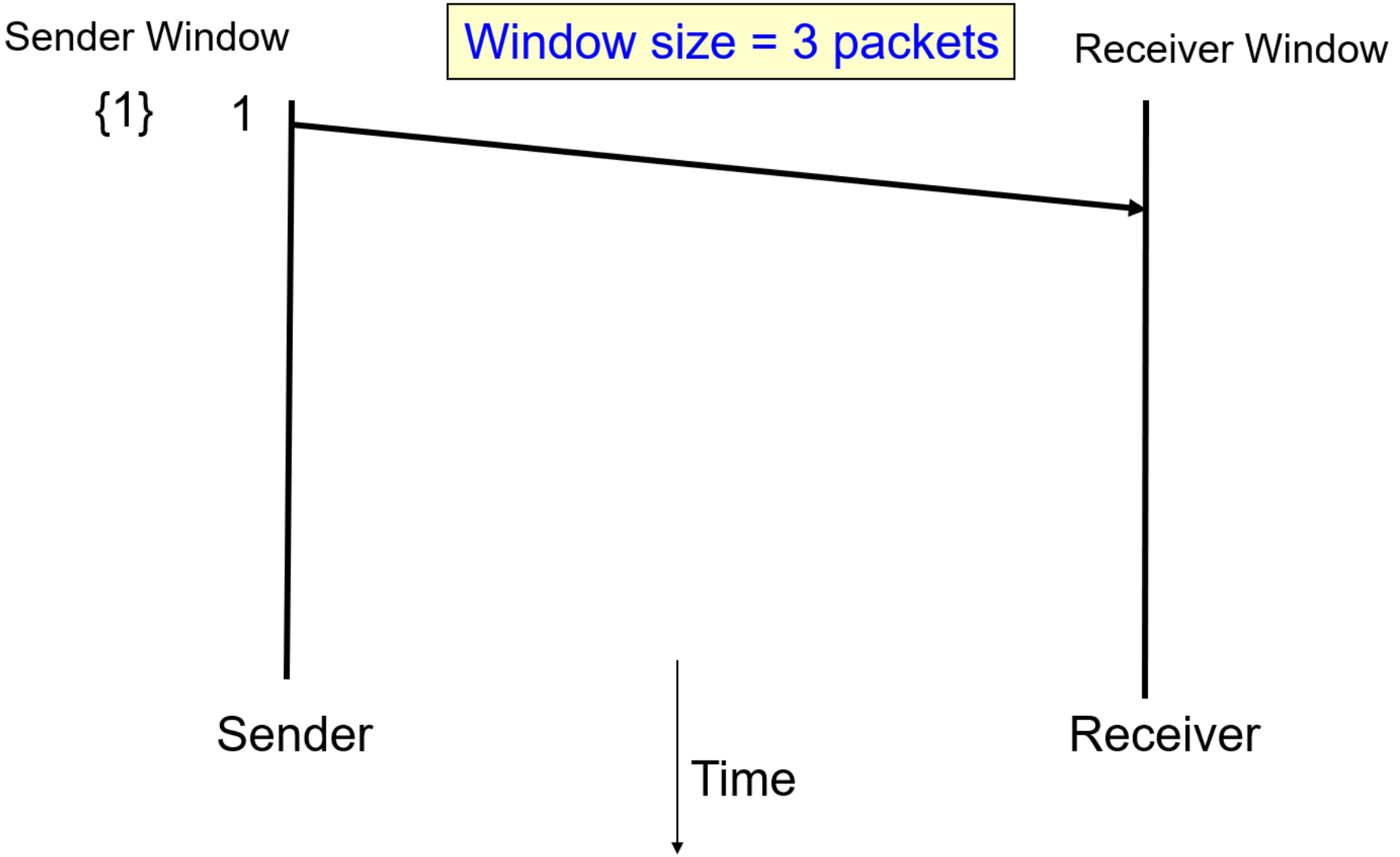
- accetta solo pacchetti corretti ed in sequenza (con numero di sequenza $R_{next}$)
- quando arriva un nuovo pacchetto in sequenza, incrementa $R_{next}$ di 1
  - la finestra di ricezione slitta di una unità
- scarta tutti i pacchetti fuori ordine
- utilizza gli ack cumulativi:
  - # sequenza spedito con l'ACK = # del prossimo pacchetto atteso in ordine



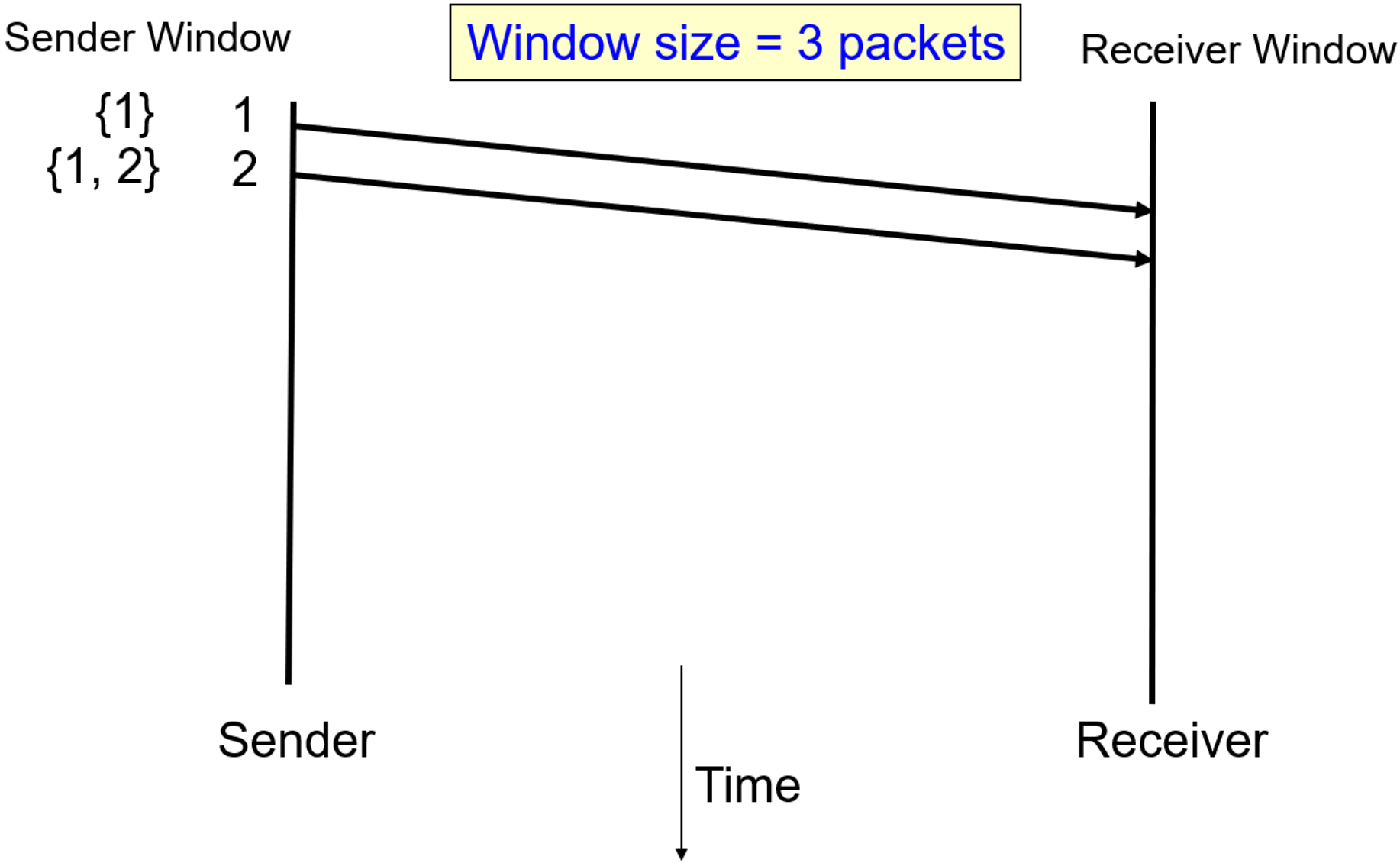
# GBN SENZA ERRORI



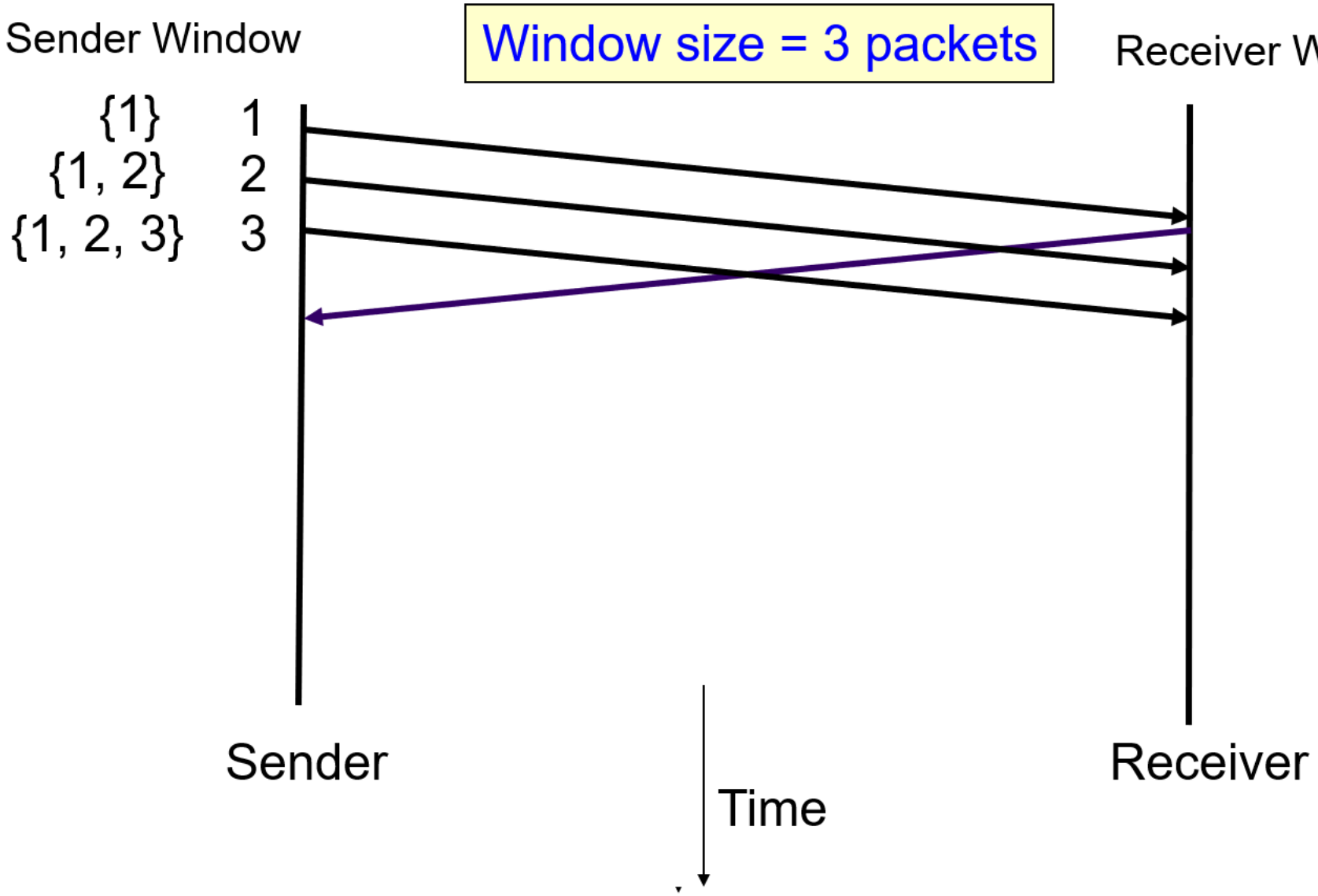
# GBN SENZA ERRORI



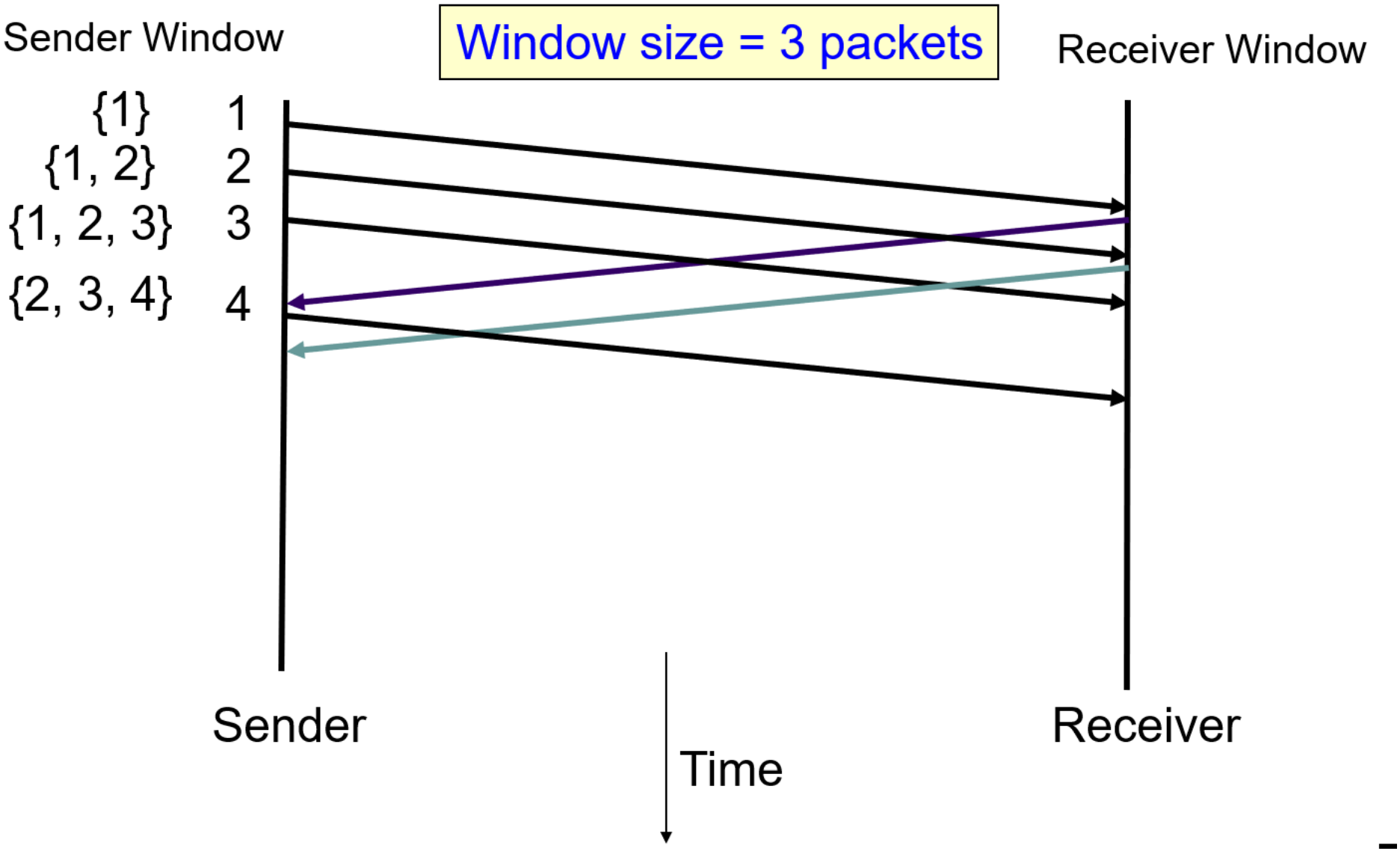
# GBN SENZA ERRORI



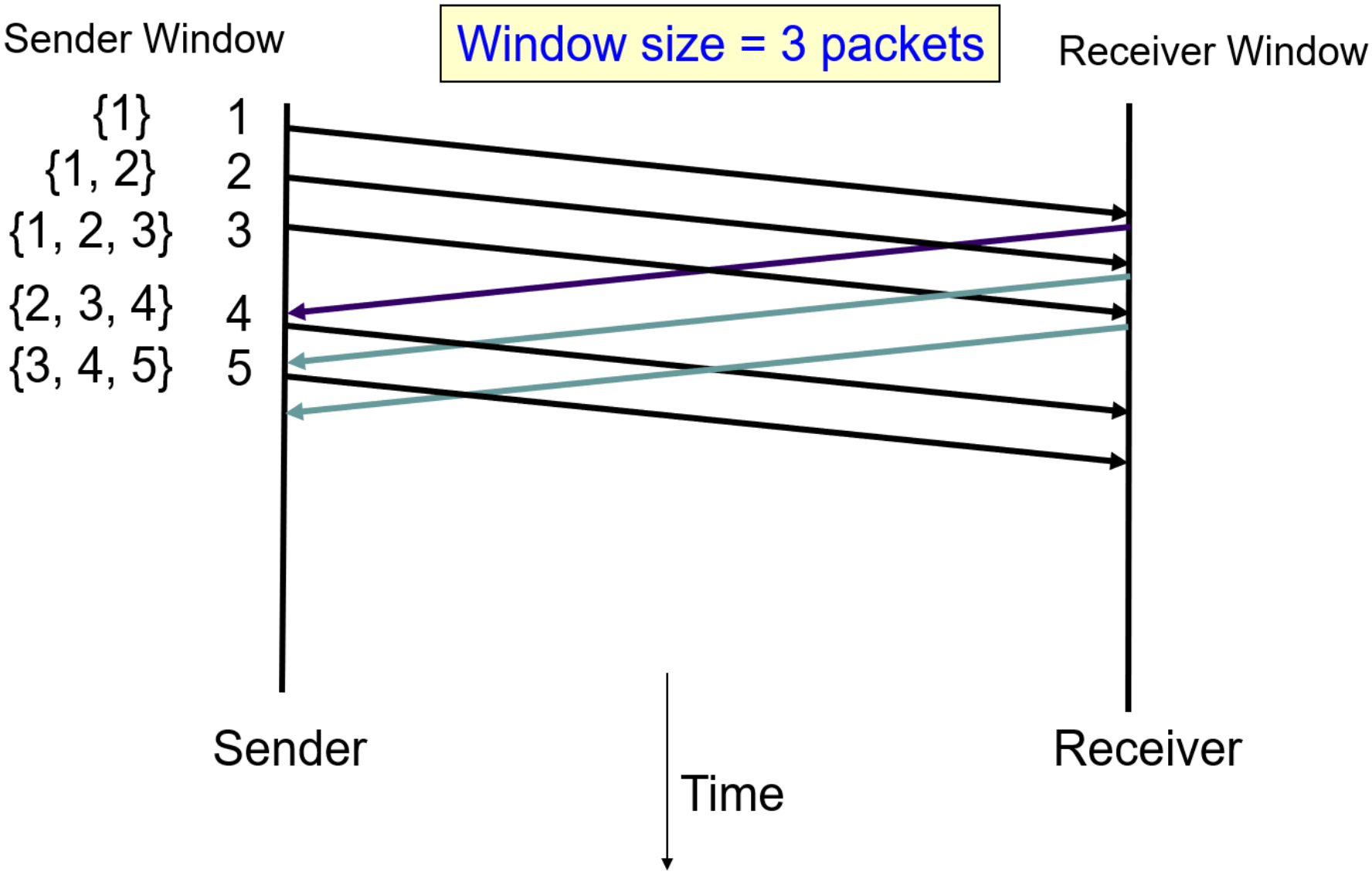
# GBN SENZA ERRORI



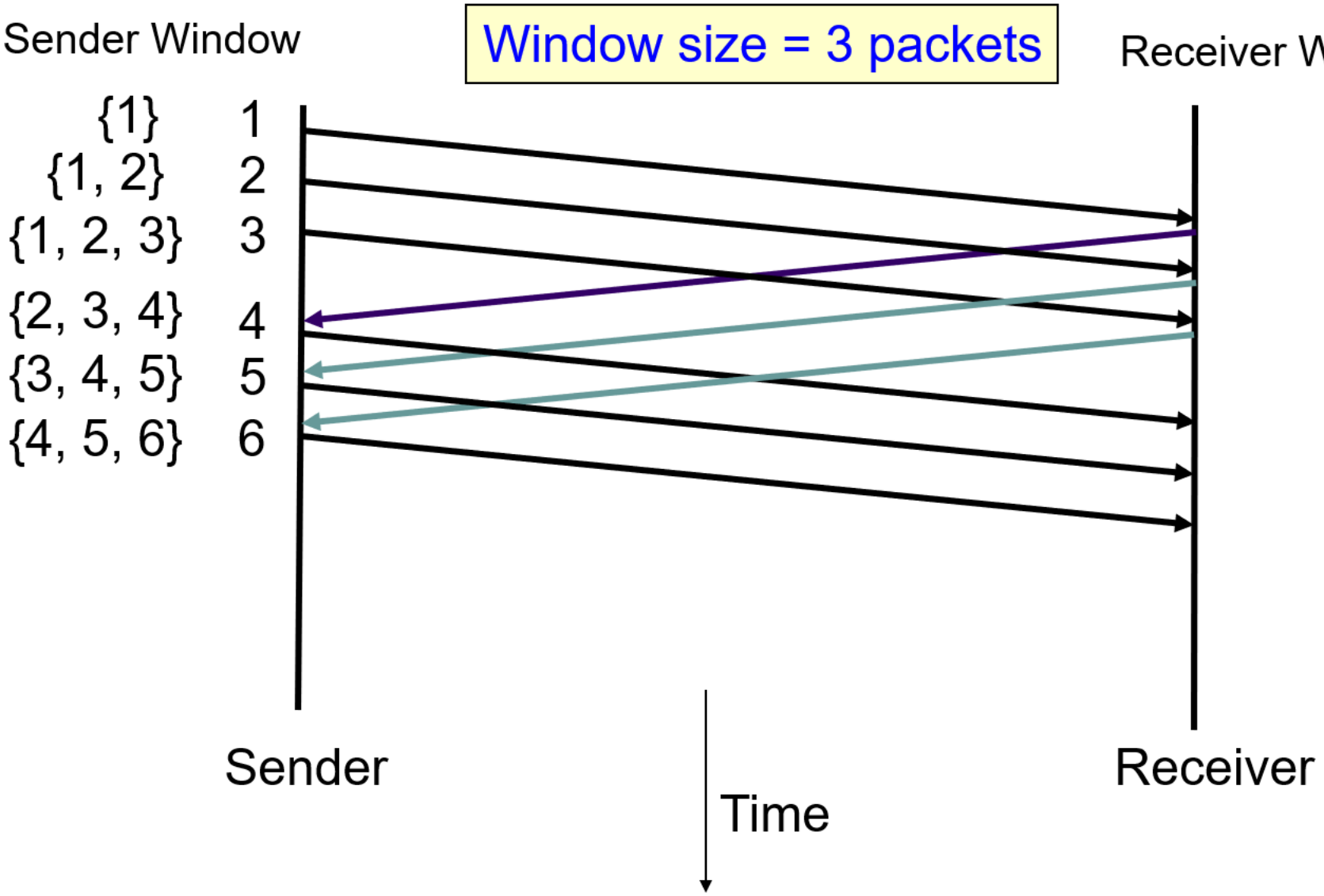
# GBN SENZA ERRORI



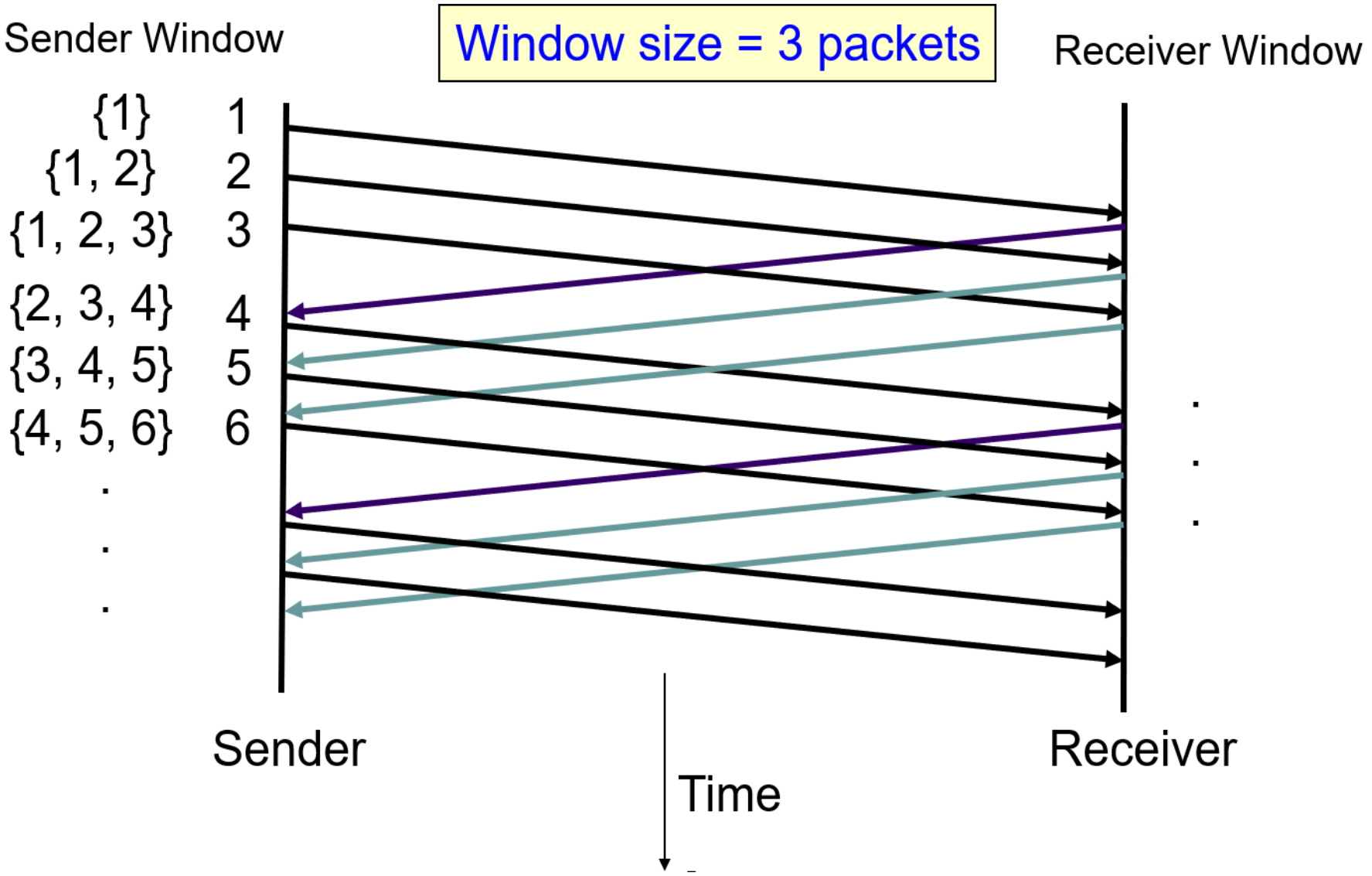
# GBN SENZA ERRORI



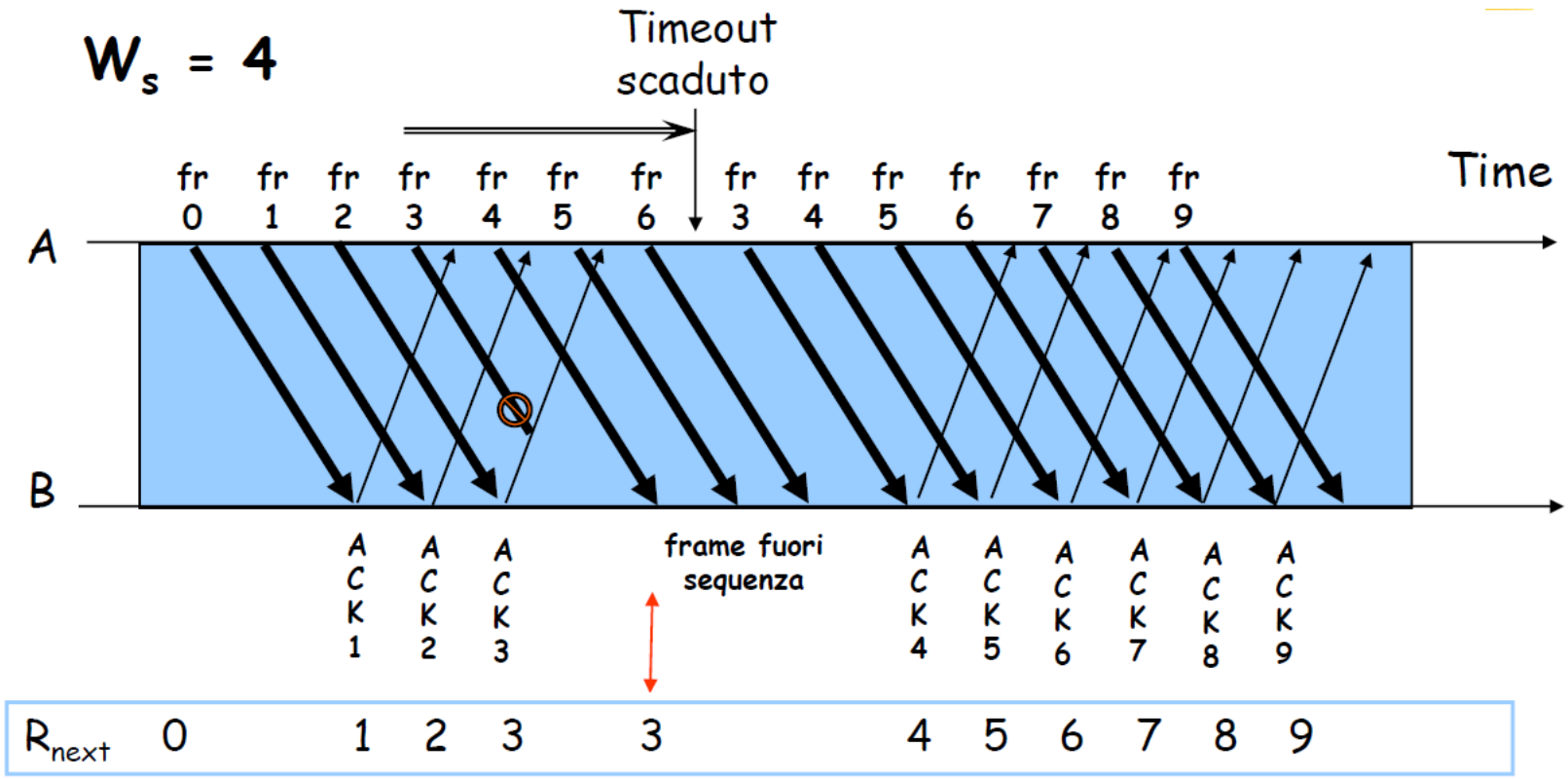
# GBN SENZA ERRORI



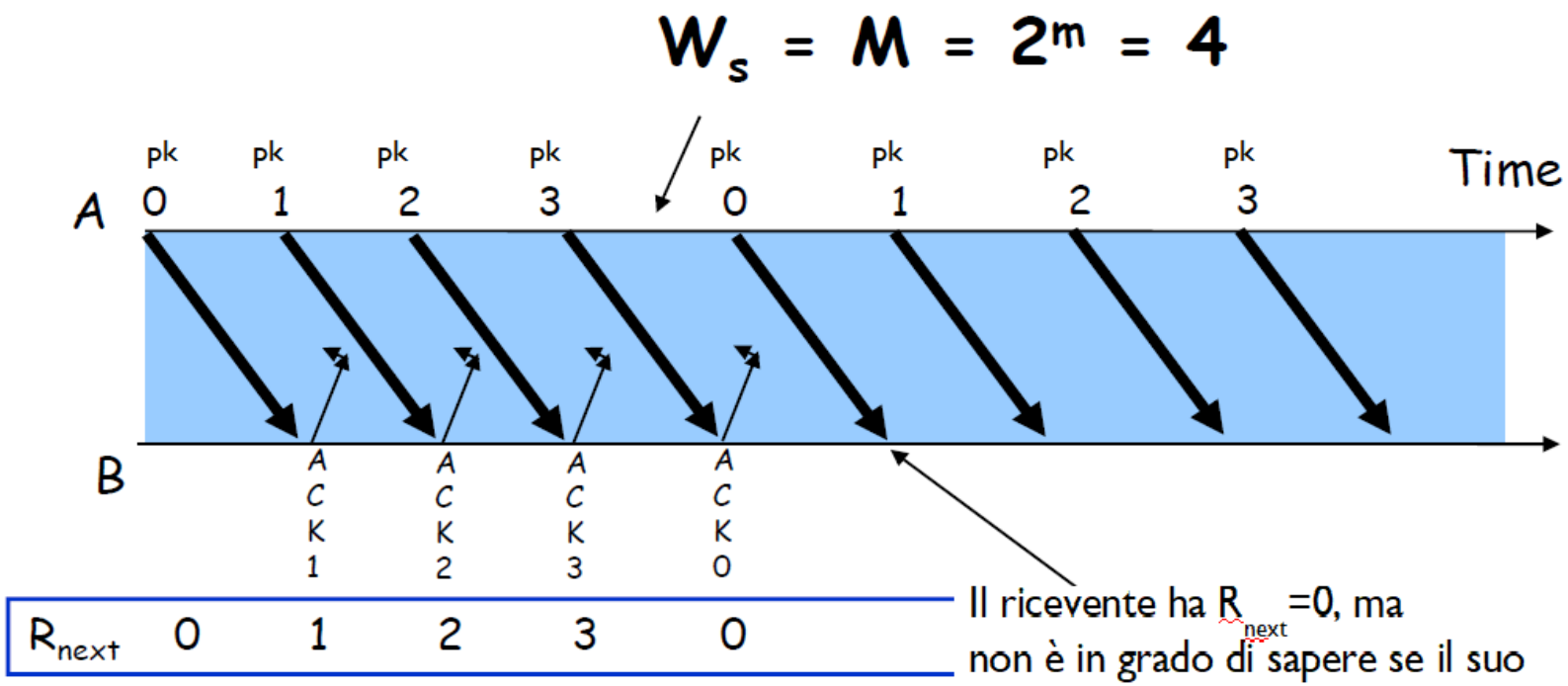
# GBN SENZA ERRORI



# GBN CON ERRORI



# DIMENSIONE MASSIMA DELLA FINESTRA

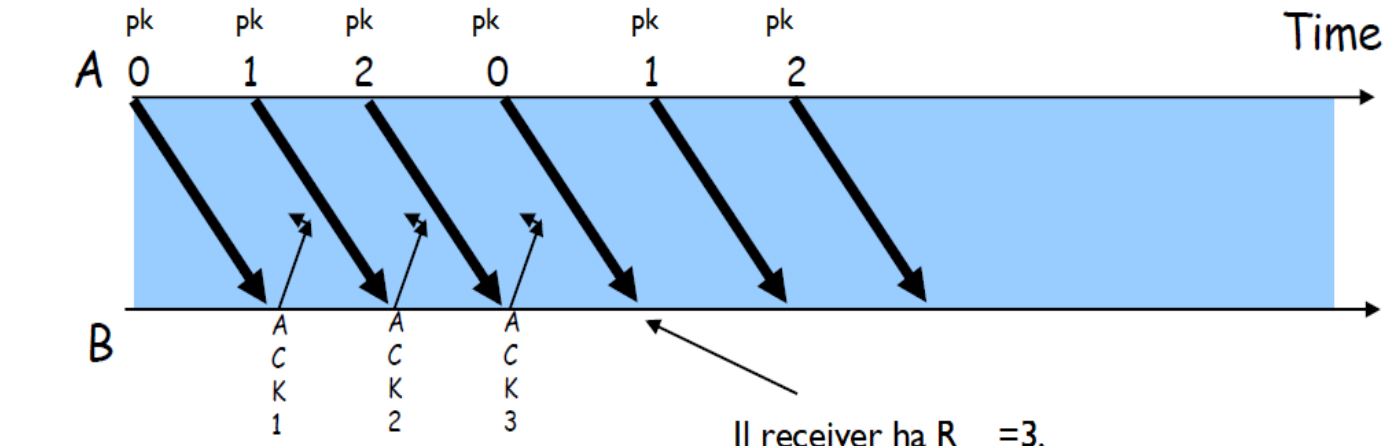



Il ricevente ha $R_{next}$=0, ma non è in grado di sapere se il suo ACK per il pacchetto 0 è stato ricevuto e quindi il pacchetto arrivato è nuovo oppure si tratta della ritrasmissione del vecchio pacchetto 0

Il massimo valore della finestra è uguale a $W_s$ = = $M - 1 = 2^m - 1$

# DIMENSIONE MASSIMA DELLA FINESTRA

$$W_s = M = 2^m - 1 = 3$$




| $R_{next}$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|

Il receiver ha $R_{next}$=3, quindi è sicuro che il pacchetto 0 arrivato è una duplcazione e quindi lo scarta

Il massimo valore della finestra è uguale a $W_s = M - 1 = 2^m - 1$

# PIGGYBACKING


"A" finestra in ricezione

$R^A_{next}$

"A" Finestra in trasmissione

$S^A_{last}$ $S^A_{last}+W^A_s-1$

Timer $S^A_{last}$

Timer $S^A_{last}+1$

Timer $S^A_{recent}$

Timer $S^A_{last}+W^A_s-1$

Buffer

"B" finestra in ricezione

$R^B_{next}$

"B" Finestra in trasmissione

$S^B_{last}$ $S^B_{last}+W^B_s-1$

Timer $S^B_{last}$

Timer $S^B_{last}+1$

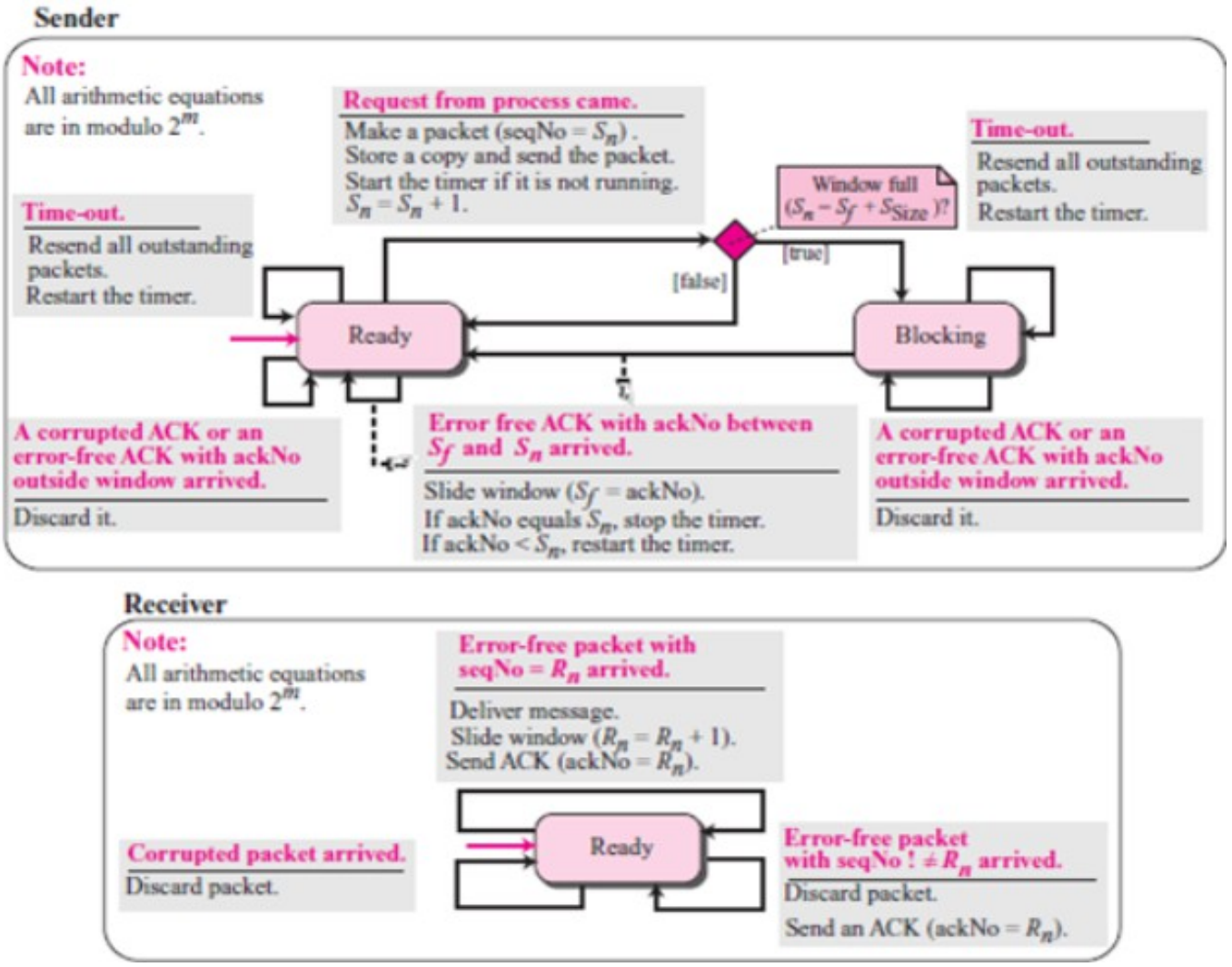
Timer $S^B_{recent}$

Timer $S^B_{last}+W^B_s-1$

Buffer

Le frame fuori sequenza sono scartate

# STATE MACHINE PER GO-BACK-N



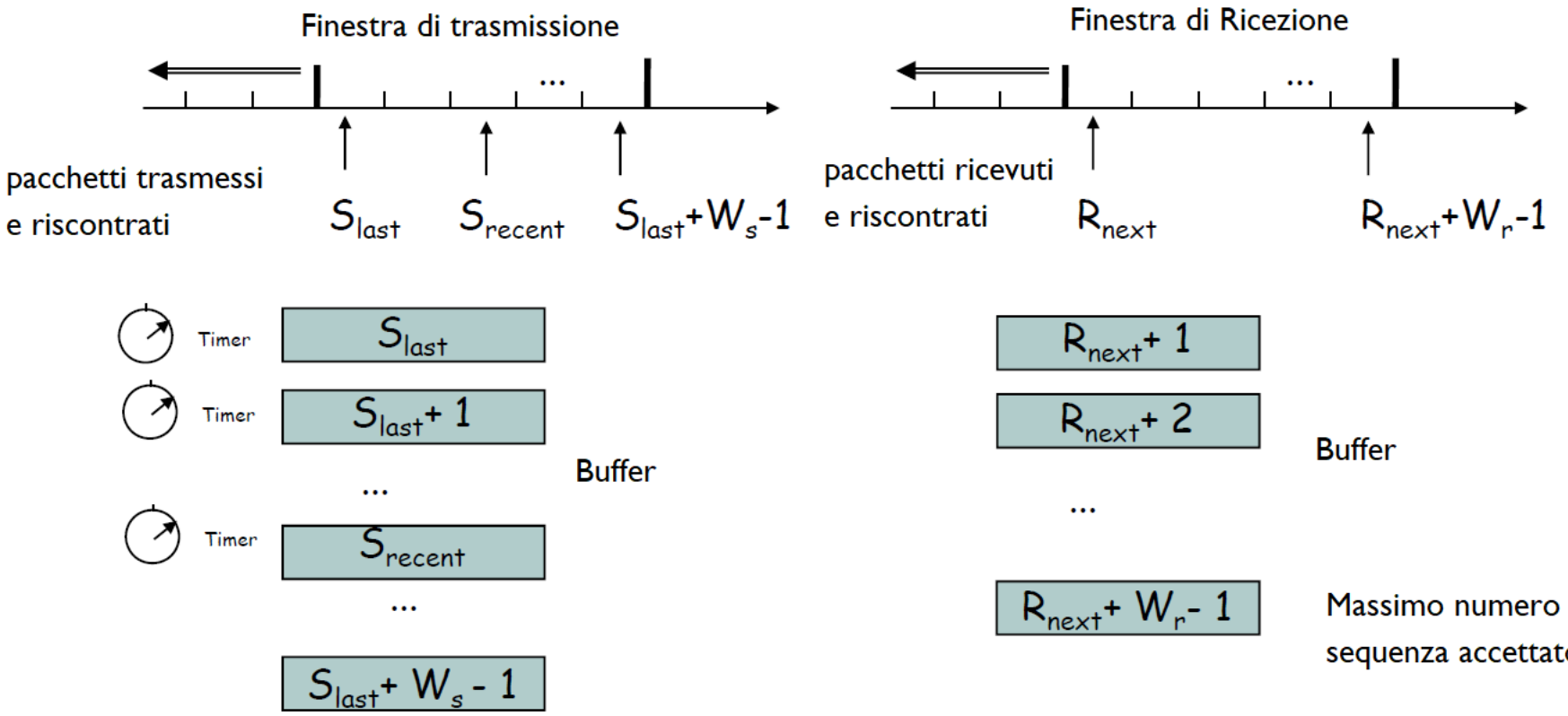
# SELECTIVE REPEAT

- Go-Back-N ARQ è inefficiente poichè, in caso di ritrasmissione, è riemesso un numero elevato di pacchetti, anche se già ricevuti correttemente dal receiver
- Selective Repeat:
  - ritrasmette solo I pacchetti che sono state persi
  - l'esaurimento del timeout determina la ritrasmissione solo del pacchetto corrispondente
- Il Receiver
  - gestisce una finestra in ricezione che indica i numeri di sequenza che possono essere accettati
  - pacchetti corretti, ma fuori sequenza con numero di sequenza compreso nella finestra in ricezione non sono scartati, ma sono bufferizzati
  - un arrivo di un pacchetto corrispondente ad $R_{next}$, determina lo scorrimento della finestra in trasmissione

# SELECTIVE REPEAT

- mittente: trasmette fino a n pacchetti non riscontrati
- se il pacchetto k è perso, mentre non lo è il pacchetto k+1
  - ricevente: manda un ack per il pacchetto k+1
  - mittente: ritrasmette solo il pacchetto k, allo scadere del timeout
- vantaggi: utilizzo efficiente delle risorse, in particolare della banda
- gestione complessa
  - un timer per ogni pacchetto

# SEECTIVE REPEAT: WINDOWS



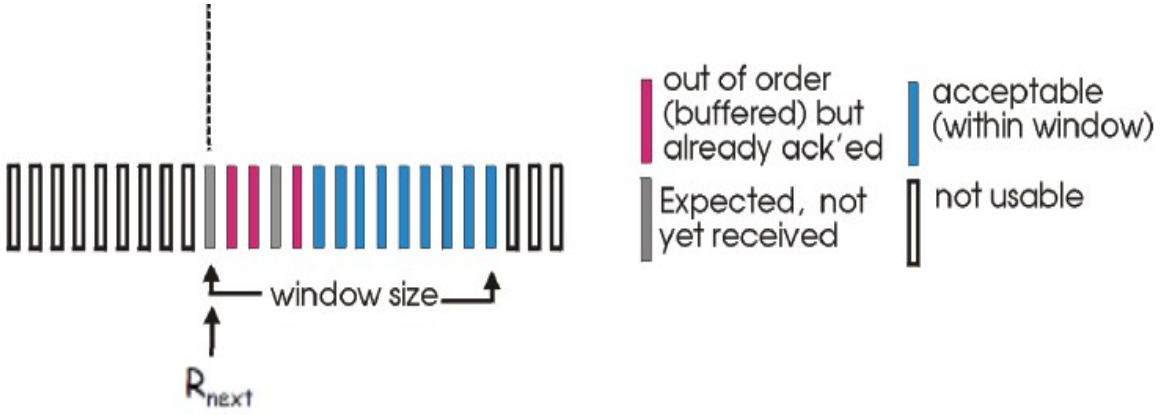
# WINDOW LATO MITTENTE

- finestra attuale del mittente (di dimensione $W_s$) contiene sia elementi riscontrati che non riscontrati


- $S_{last}$ -1: indice ultimo pacchetto riscontrato tale che tutti i pacchetti spediti precedentemente sono stati anche essi riscontrati

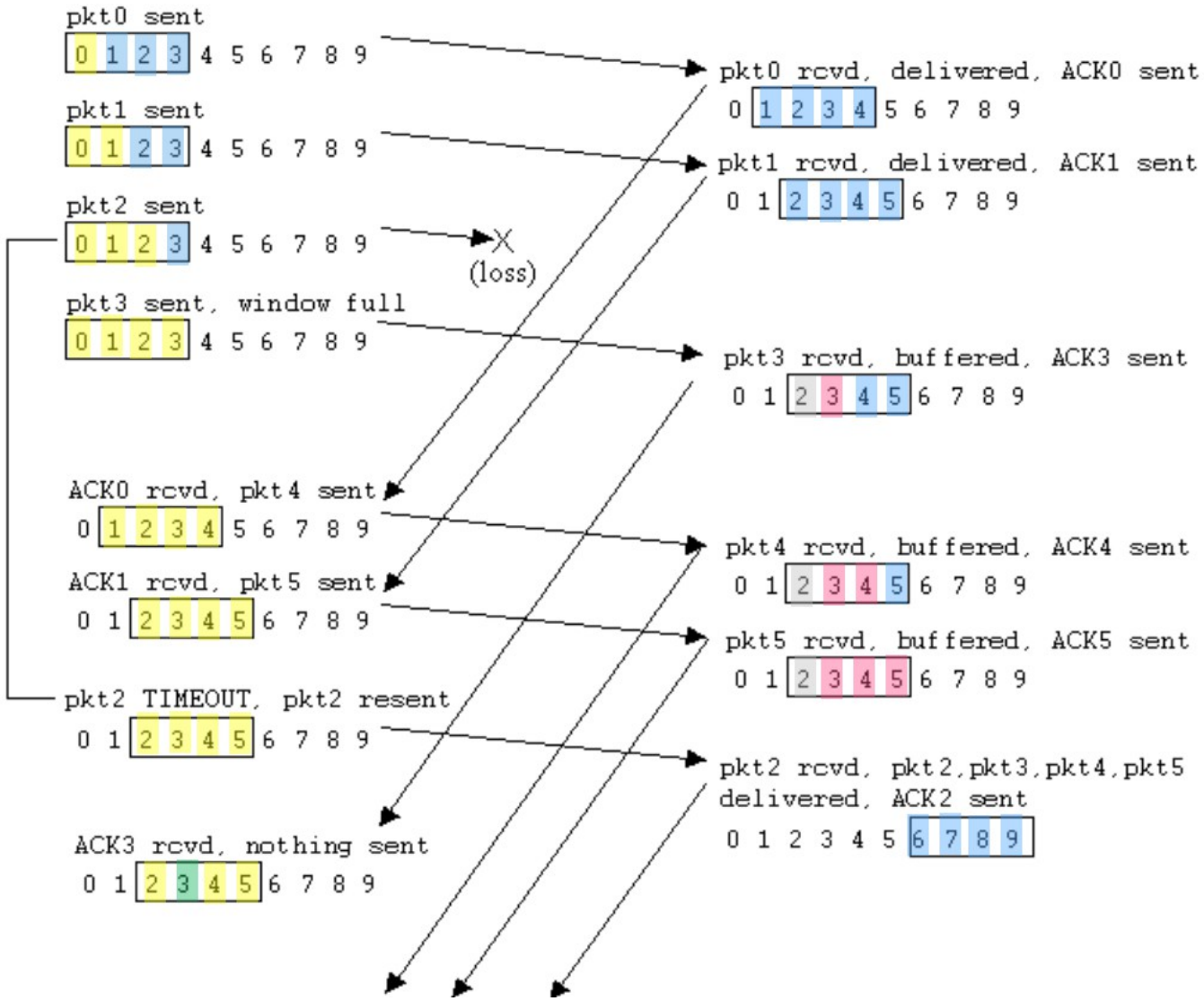
# WINDOW LATO DESTINATARIO

- la finestra in ricezione permette al destinario di considerare "buoni" anche gli eventuali pacchetti giunti fuori sequenza.
  - dimensione della finestra di ricezione: $W_R$
  - $W_R \leq W_S$, in genere $W_R = W_S$
  - scorre quando si riceve un prefisso di numeri in sequenza "in ordine"
- in caso di errore, consente al mittente di ritrasmettere solo i pacchetti errati (selective reject)

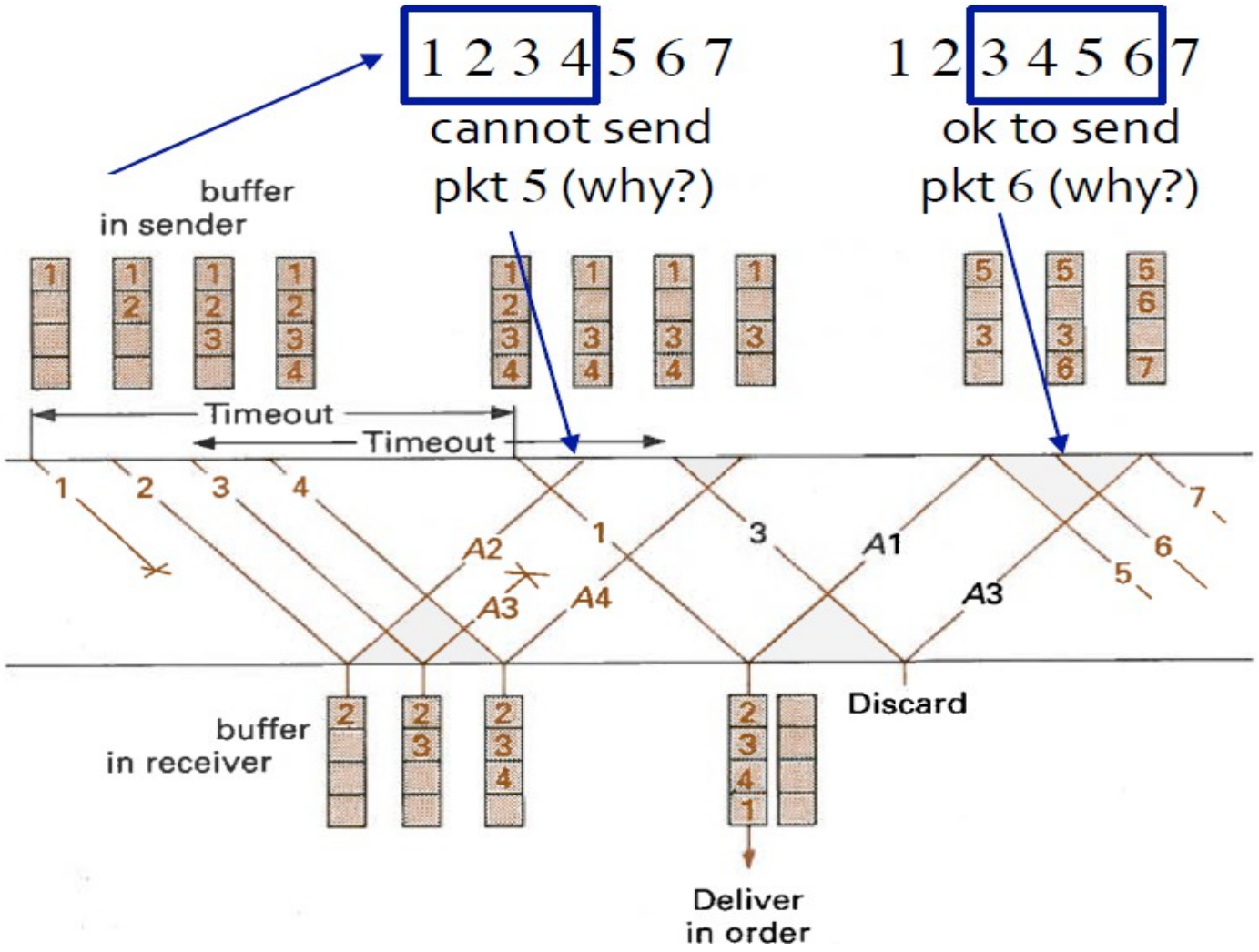



- $R_{next}$ -1: indice dell'ultimo pacchetto ricevuto in ordine dal mittente

# SELECTIVE REPEAT: UN ESEMPIO

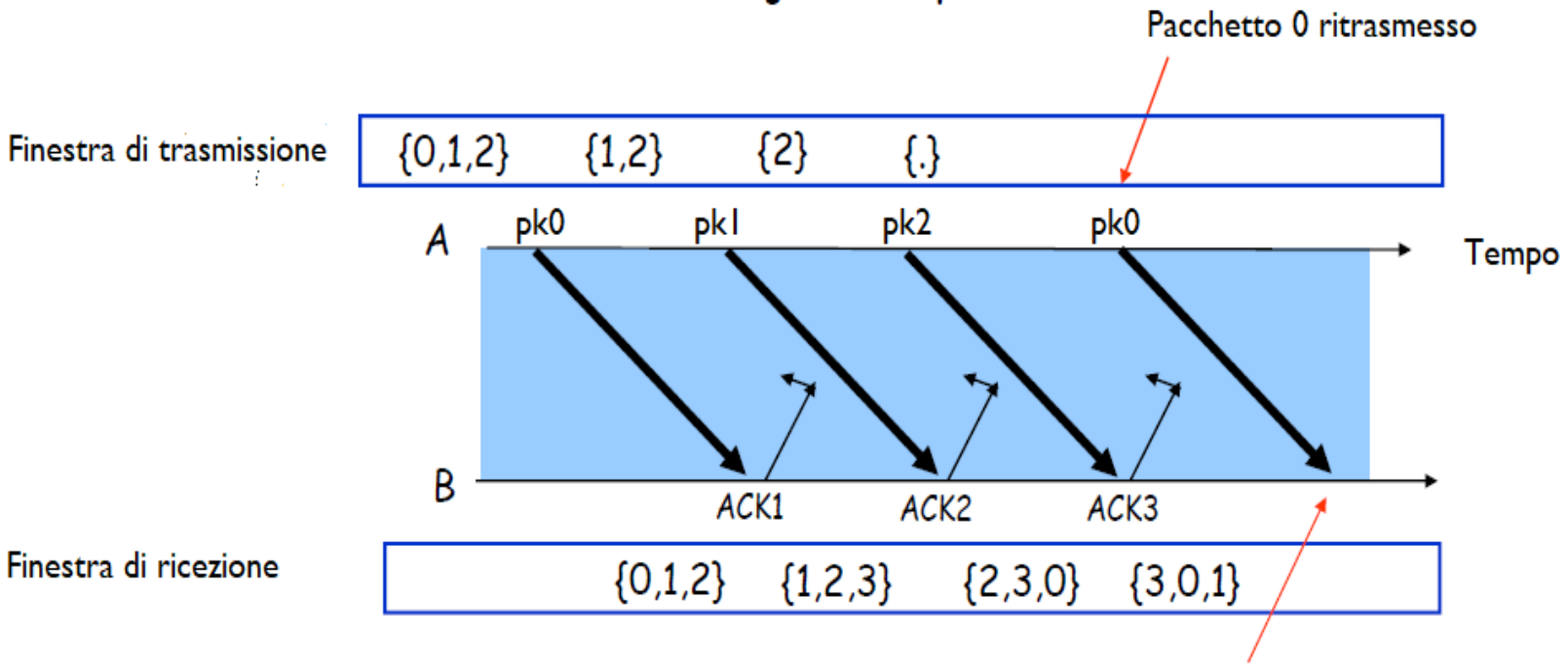


# SELECTIVE REPEAT: UN ESEMPIO



# DIMENSIONE MASSIMA DELLE FINESTRE ?

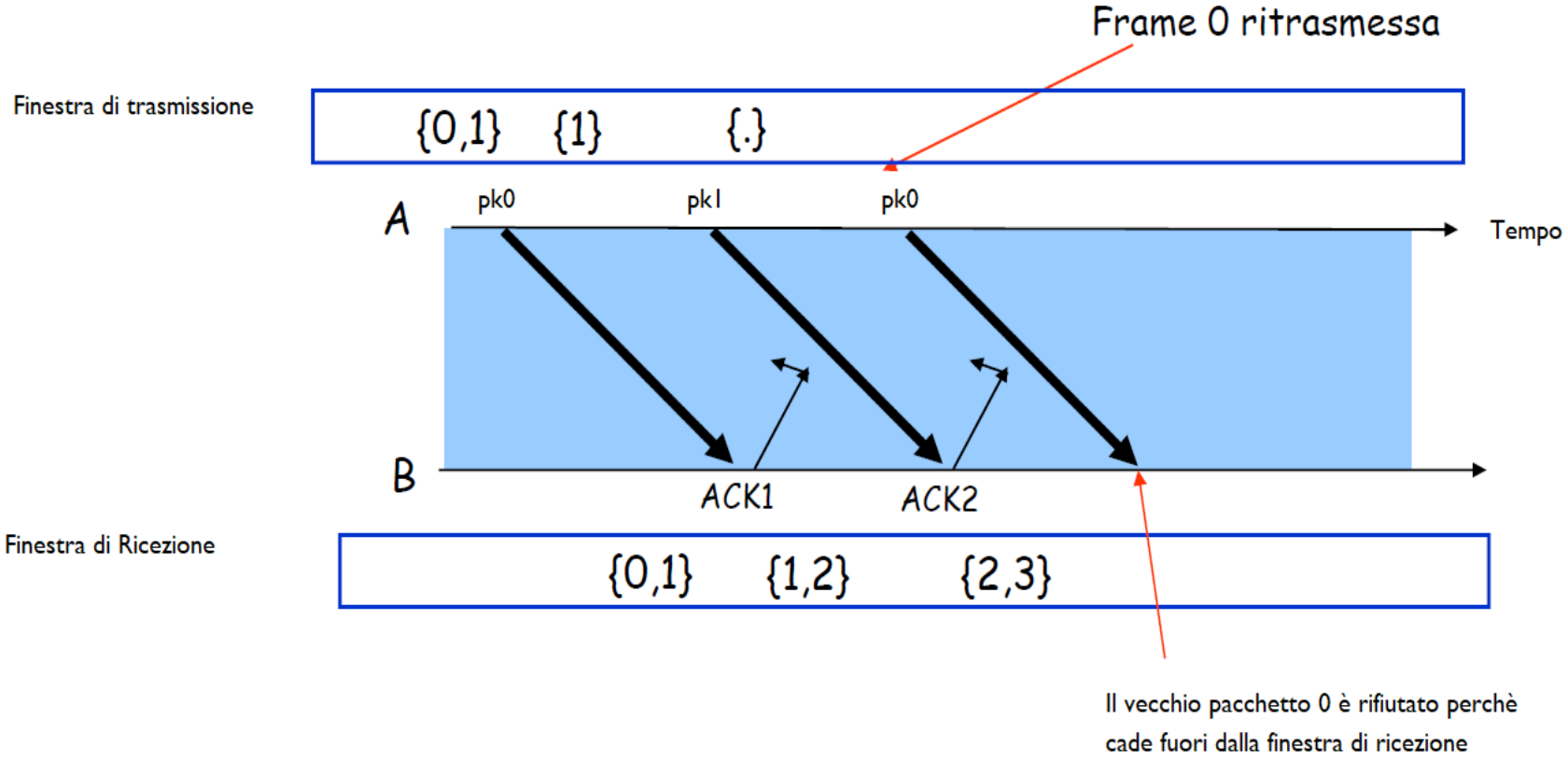
2 bit per i numeri di sequenza $2^2=4$, $W_s=3$, $W_r=3$




Il vecchio pacchetto 0 è accettato perchè ricade nell'intervallo di ricezione.

# DIMENSIONE MASSIMA DELLE FINESTRE?

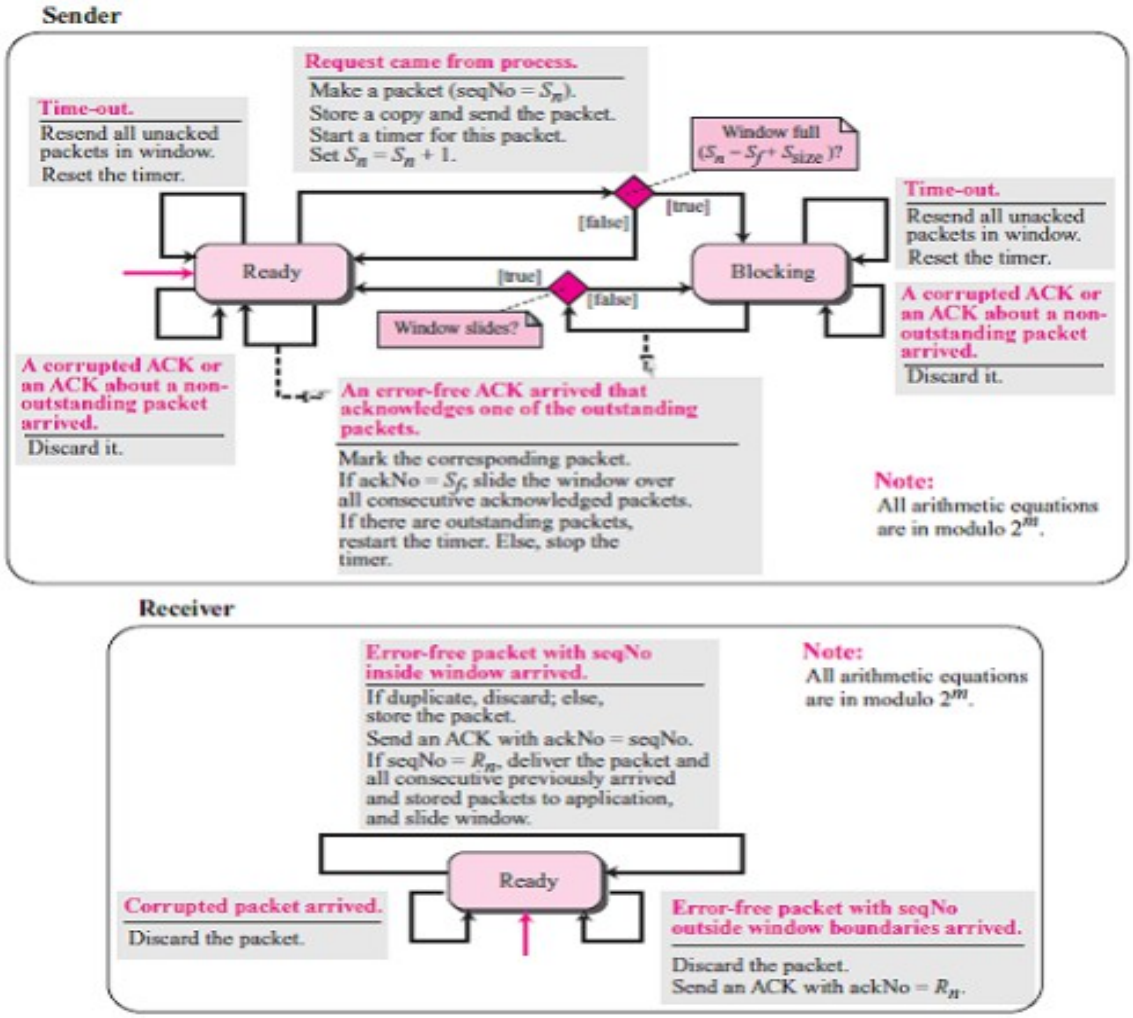
2 bit per i numeri di sequenza $2^2=4,\ W_s=2,\ W_r=2$



# DIMENSIONE MASSIMA DELLE FINESTRE?

- Se si utilizzano m bit per i numeri di sequenza
- $2^m$ identificatori
- La dimensione massima delle finestre di invio e di ricezione può essere al massimo $2^{m-1}$

# STATE MACHINE PER SELECTIVE REPEAT



# E IL TCP?

Usa un "mix" dei concetti visti, con alcune differenze:

- i numeri di sequenza sono byte offsets
- il mittente ed il ricevente mantengono una sliding window
- il ricevente invia ACK cumulativi (come GBN)
- il mittente mantiene un solo timer di ritrasmissione
- il ricevente non scarta I pacchetti ricevuti (come SR)
- introduce il meccanismo del fast retransmit : ottimizzazione che usa gli ACK duplicati per accellerare la ritrasmissione di pacchetti
- introduce algoritmi per la stima dei timeout