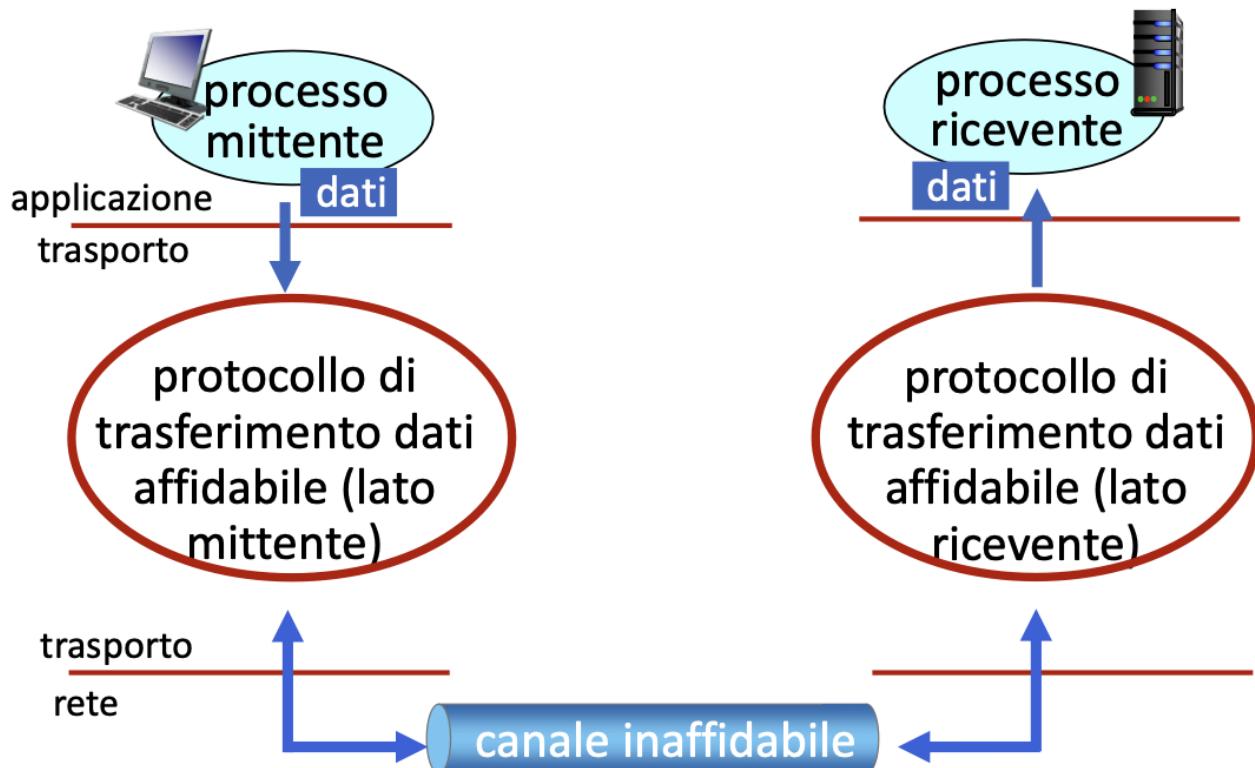


Lezione 8 - Livello di Trasporto

Principi del trasferimento dati affidabile



- **Astrazione** di un servizio affidabile



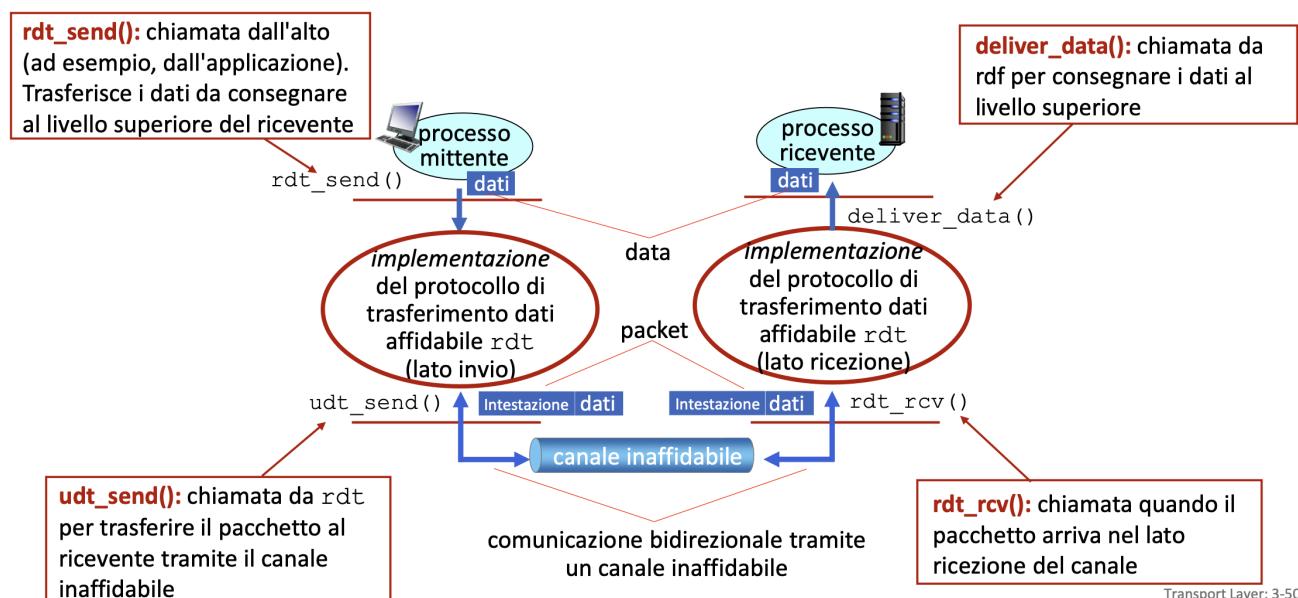
- **Implementazione** di un servizio affidabile

La complessità del protocollo di trasferimento dati affidabile dipende dalle caratteristiche del canale inaffidabile.

Il mittente e il ricevente non conoscono ciascuno lo "stato" dell'altro , ad esempio : il messaggio è stato ricevuto ?

- A meno che non venga comunicato attraverso un messaggio

Trasferimento dati affidabile (rdt) : interfacce



- **rdt_send()** : chiamata dall'alto (ad esempio, dall'applicazione). Trasferisce i dati da consegnare al livello superiore del ricevente.
- **udt_send()** : chiamata da rdt per trasferire il pacchetto al ricevente tramite il canale inaffidabile.
- **deliver_data()** : chiamata da rdf per consegnare i dati al livello superiore.
- **rdt_rcv()** : chiamata quando il pacchetto arriva nel lato ricezione del canale.

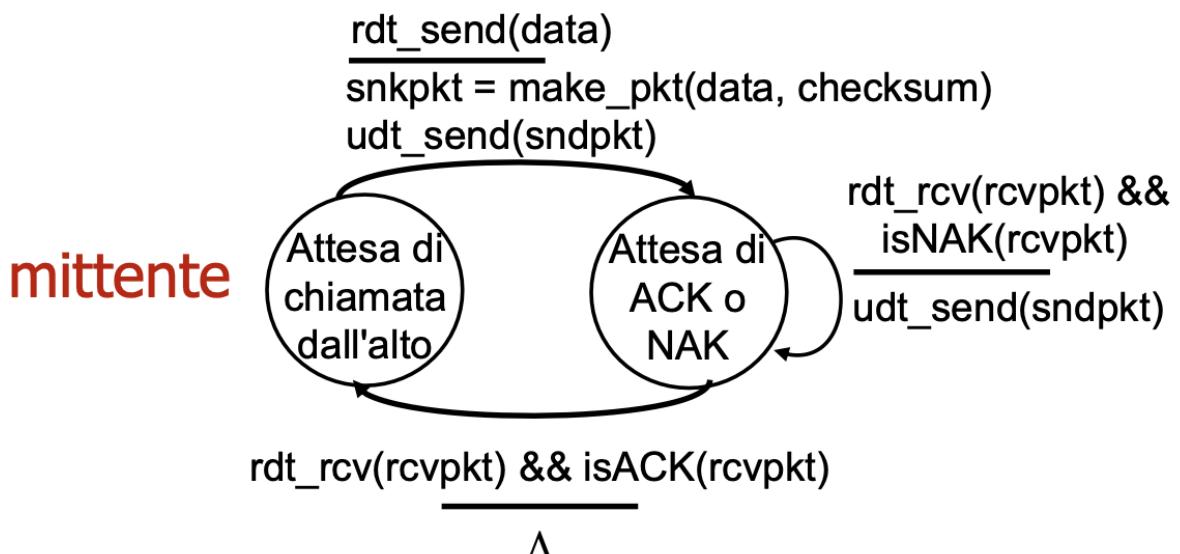
rdt1.0 : trasferimento affidabile su canale affidabile

- Canale sottostante perfettamente affidabile
 - nessun errore nei bit
 - nessuna perdita di pacchetti
- FSM distinto per il mittente e il ricevente:
 - il mittente invia I dati nel canale sottostante
 - il ricevente legge I dati dal canale sottostante

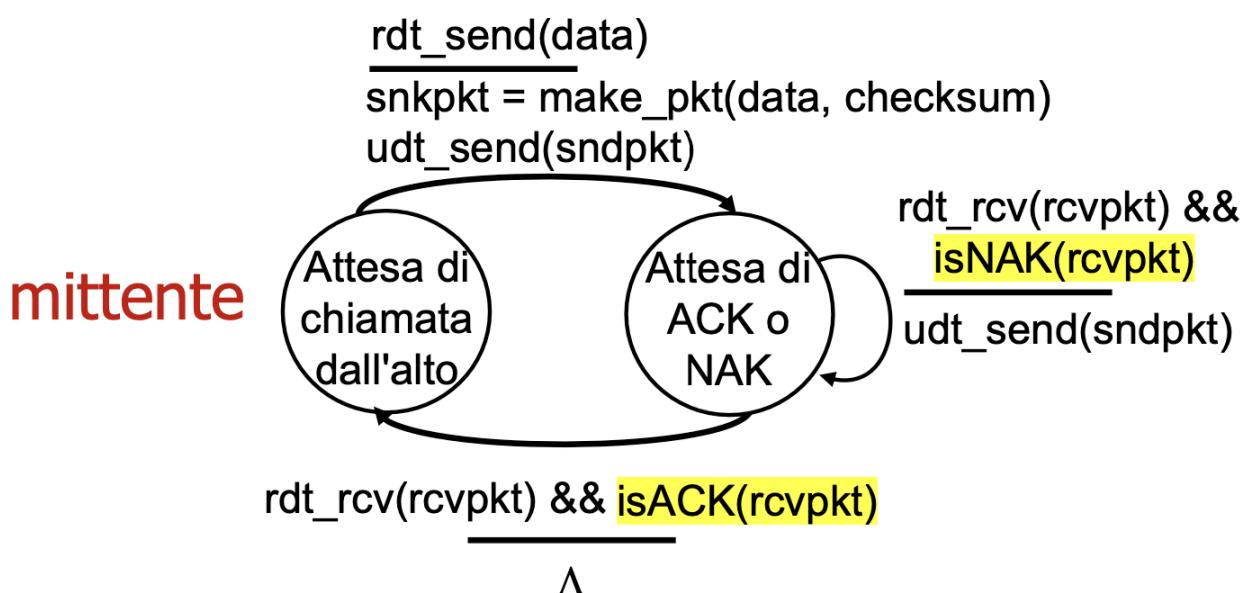


rdt2.0 : canale con errori nei bit

- il canale sottostante può invertire (flip) i bit nel pacchetto
 - checksum (ad esempio, Internet checksum) per rilevare gli errori nei bit
- la domanda: come recuperare (recover) dagli errori?
- **notifica positiva, acknowledgements (ACKs)**: il ricevente comunica espressamente al mittente che il pacchetto ricevuto è corretto.
- **notifica negativa, negative acknowledgements (NAKs)**: il ricevente comunica espressamente al mittente che il pacchetto contiene errori
- il mittente **rtrasmette** il pacchetto se riceve un NAK.
- **Stop and wait** : il mittente invia un pacchetto , quindi attende la risposta del destinatario.



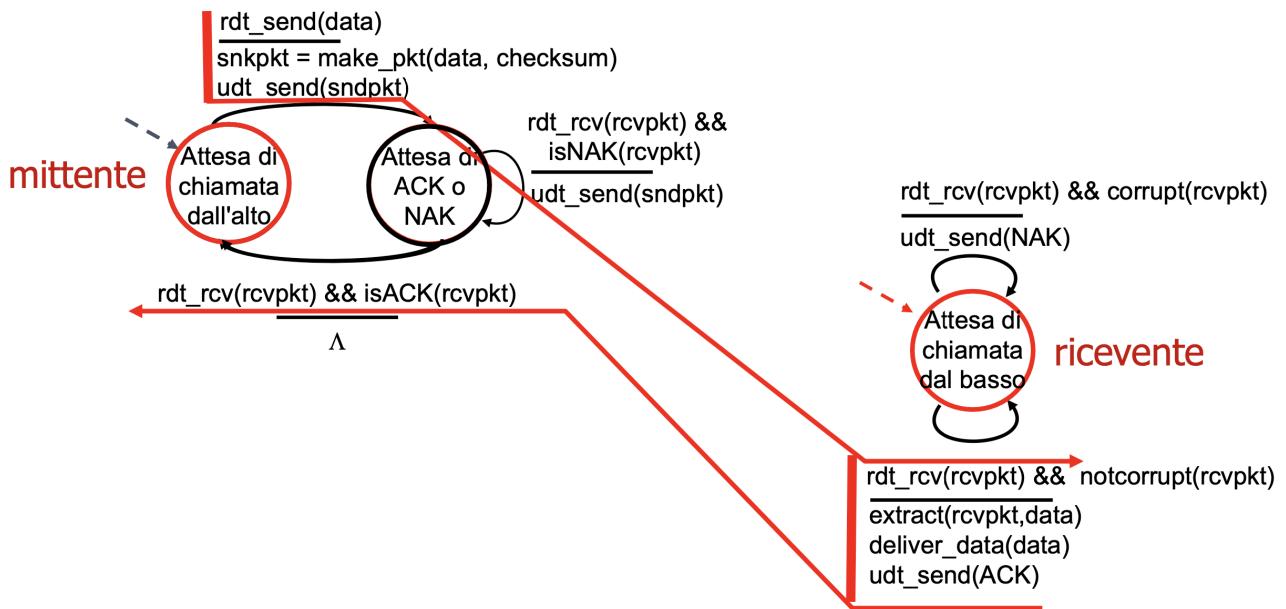
rdt2.0 : specifica delle FSM



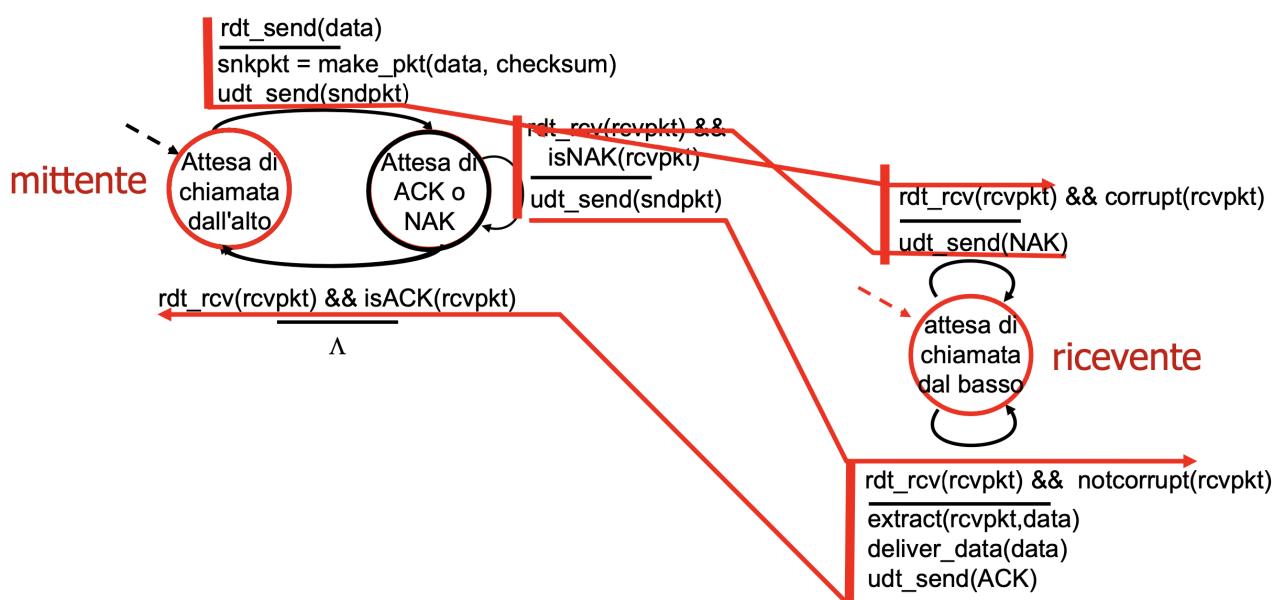
Nota : lo "stato" del destinatario (ha ricevuto correttamente il mio messaggio?) non è noto al mittente a meno che non venga comunicato in qualche modo dal destinatario al mittente.

- ecco perché abbiamo bisogno di un protocollo!

rdt2.0 : operazioni senza errori



rdt2.0 : scenario di errore



rdt2.0 ha un difetto fatale!

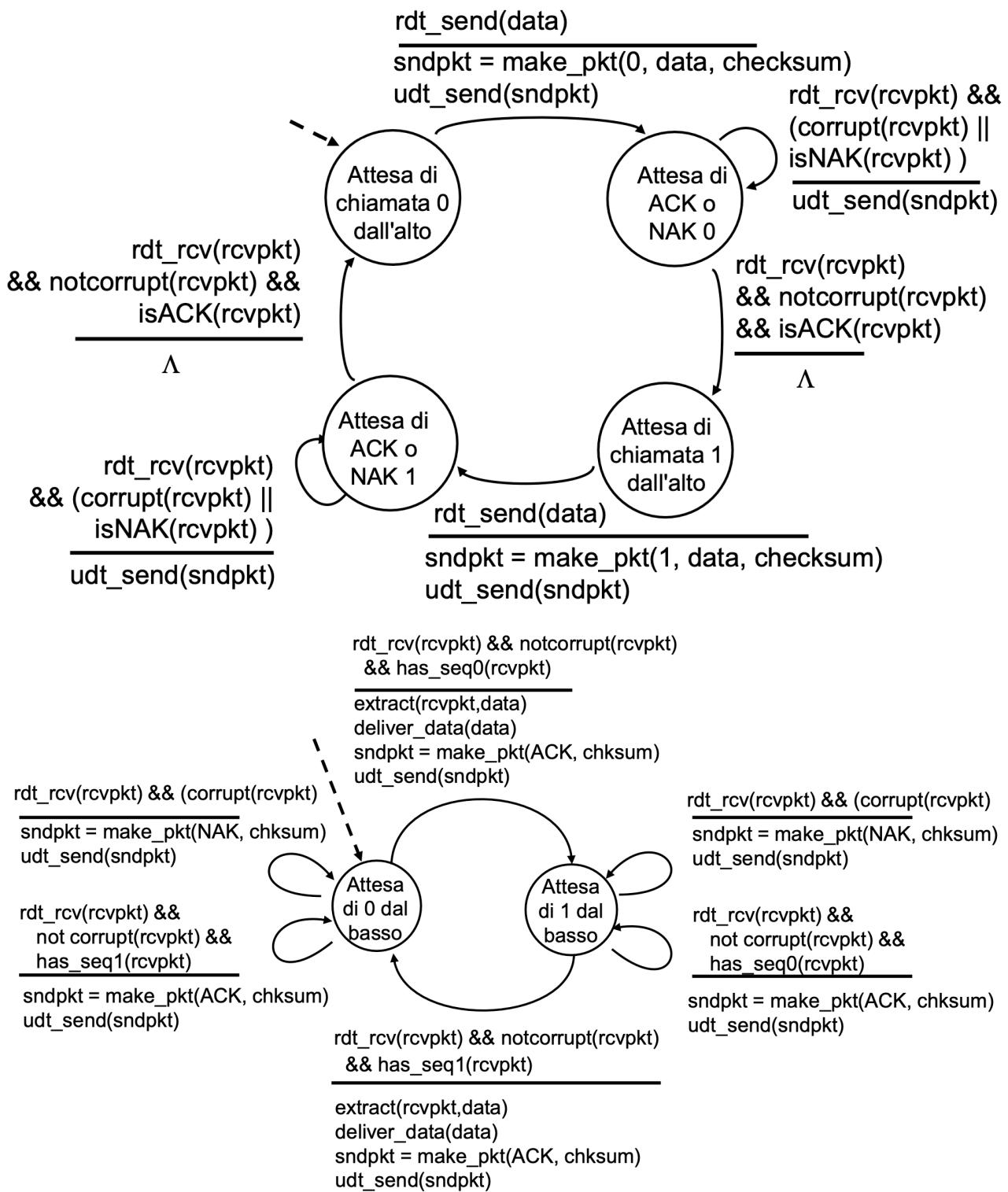
Che cosa accade se i pacchetti ACK/NAK sono danneggiati?

- il mittente non sa che cosa sia accaduto
- non basta ritrasmettere: possibili duplicati

Gestione dei duplicati

- il mittente ritrasmette il pacchetto corrente se ACK/NAK è alterato
- il mittente aggiunge un numero di sequenza a ogni pacchetto
- il ricevitore scarta (non consegna) il pacchetto duplicato

rdt2.1 : il mittente gestisce gli ACK/NAK alterati



rdt2.1 : discussion

Mittente :

- aggiunge il numero di sequenza al pacchetto
- saranno sufficienti due numeri di sequenza (0,1). Perché?

- deve controllare se gli ACK/NAK sono danneggiati
- il doppio di stati
 - lo stato deve "ricordarsi" se il pacchetto "corrente" ha numero di sequenza 0 o 1

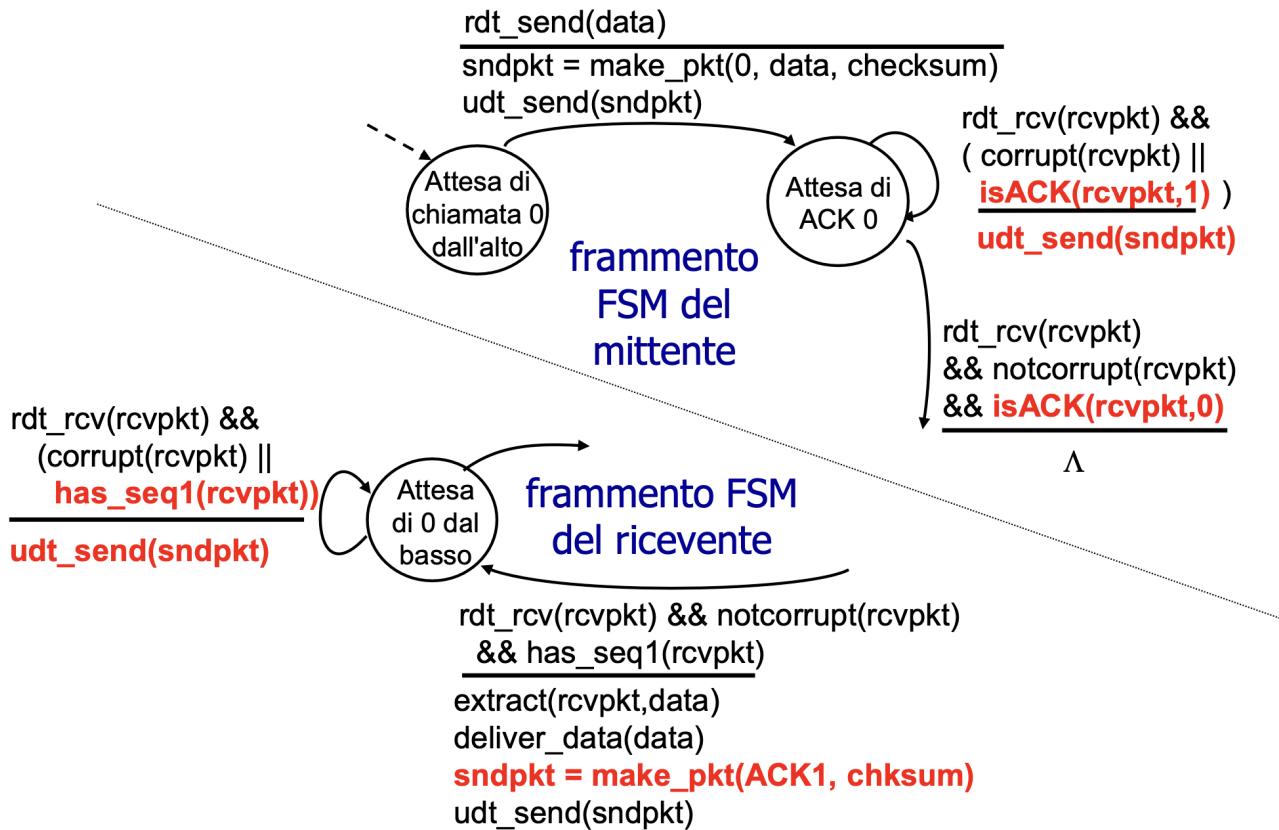
Ricevente :

- deve controllare se il pacchetto ricevuto è duplicato
 - lo stato indica se il numero di sequenza previsto è 0 o 1
- nota: il ricevente non può sapere se il suo ultimo ACK/NAK è stato ricevuto correttamente dal mittente

rdt2.2 : un protocollo senza NAK

- stessa funzionalità di rdt2.1, utilizzando soltanto gli ACK
- al posto del NAK, il destinatario invia un ACK per l'ultimo pacchetto ricevuto correttamente
 - il destinatario deve includere esplicitamente il numero di sequenza del pacchetto con l'ACK
- Un ACK duplicato presso il mittente determina la stessa azione del NAK: ritrasmettere il pacchetto corrente

Come vedremo, il protocollo TCP utilizza questo approccio senza NAK.



rdt3.0 : canali con errori e perdite

Nuova ipotesi : il canale sottostante può anche smarrire pacchetti (dati o ACK)

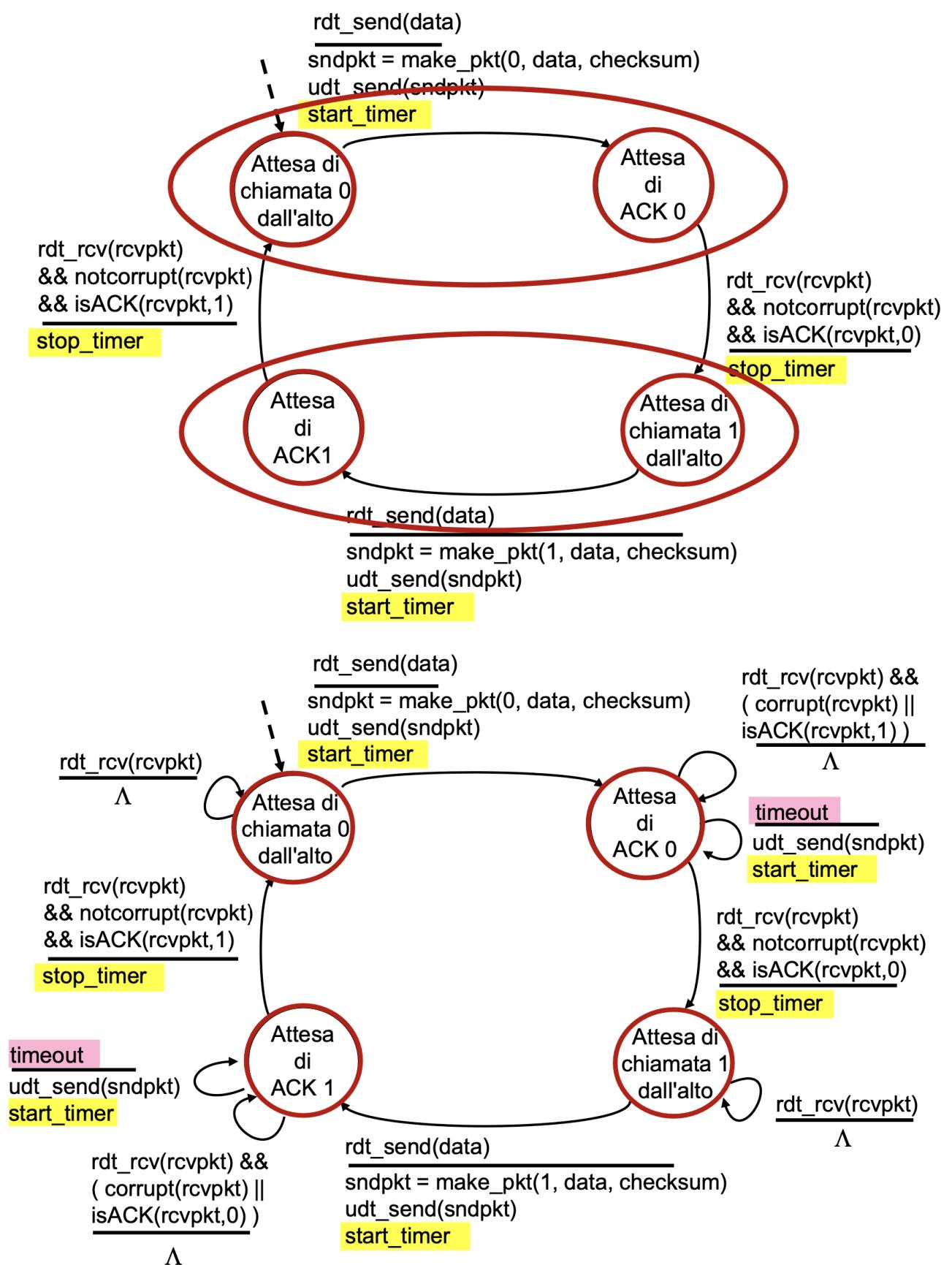
- checksum, numeri di sequenza, ACK, ritrasmissioni aiuteranno...
ma non sono sufficienti

Approccio : il mittente attende un ACK per un tempo "ragionevole"

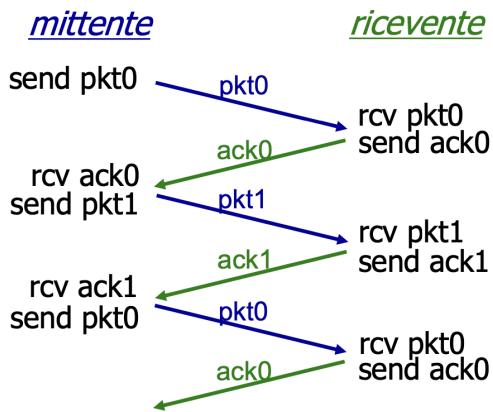
- ritrasmette se non riceve un ACK in questo periodo
- se il pacchetto (o l'ACK) è soltanto in ritardo (non perso)
 - la ritrasmissione sarà duplicata, ma l'uso dei numeri di sequenza gestisce già questo
 - il destinatario deve specificare il numero di sequenza del pacchetto da riscontrare

- utilizzare un timer per il conto alla rovescia (countdown timer) per interrompere (interrupt) dopo un periodo di tempo "ragionevole"

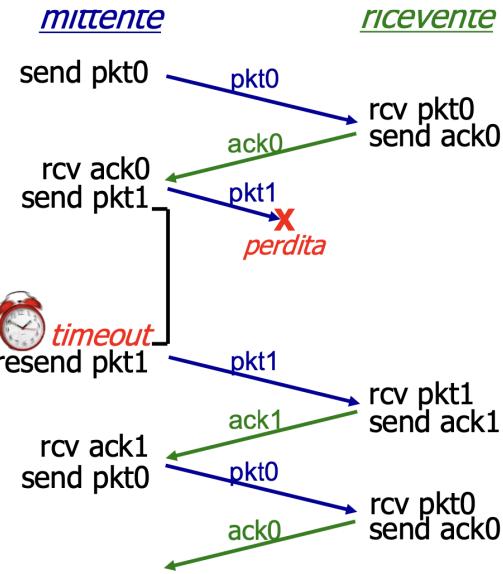
rdt3.0 : mittente



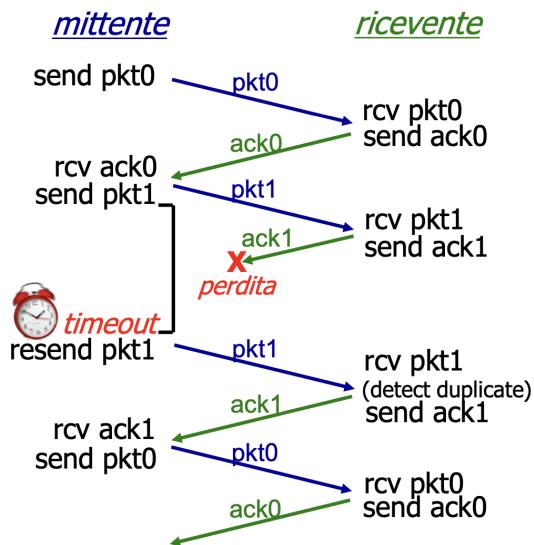
rdt3.0 in azione



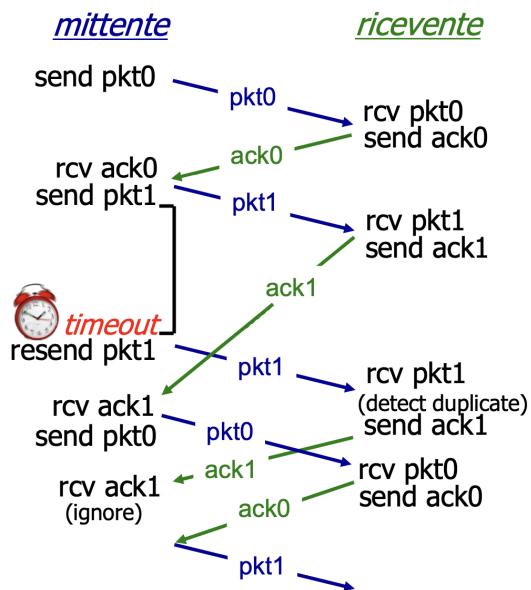
(a) operazioni senza perdita



(b) perdita di pacchetto



(c) perdita di ACK

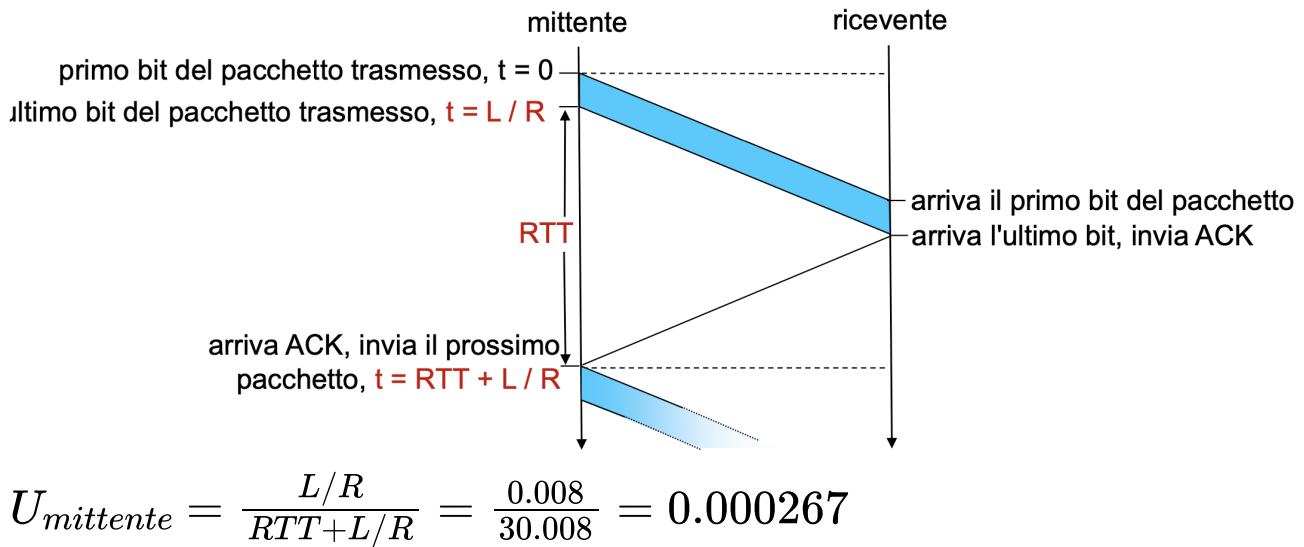


(d) timeout prematuro / ACK ritardato

Prestazione rdt3.0 (stop and wait)

- $U_{mittente}$: **utilizzazione** - la frazione di tempo in cui il mittente è stato effettivamente occupato nell'invio di bit sul canale
- esempio: collegamento da 1 Gbps, ritardo di propagazione 15 ms, pacchetti da 1000 byte (8000 bit)
 - Tempo per trasmettere un pacchetto sul collegamento:
$$D_{trasm} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bits/sec}} = 8\mu\text{s}$$

rdt3.0 : funzionamento con stop-and-wait



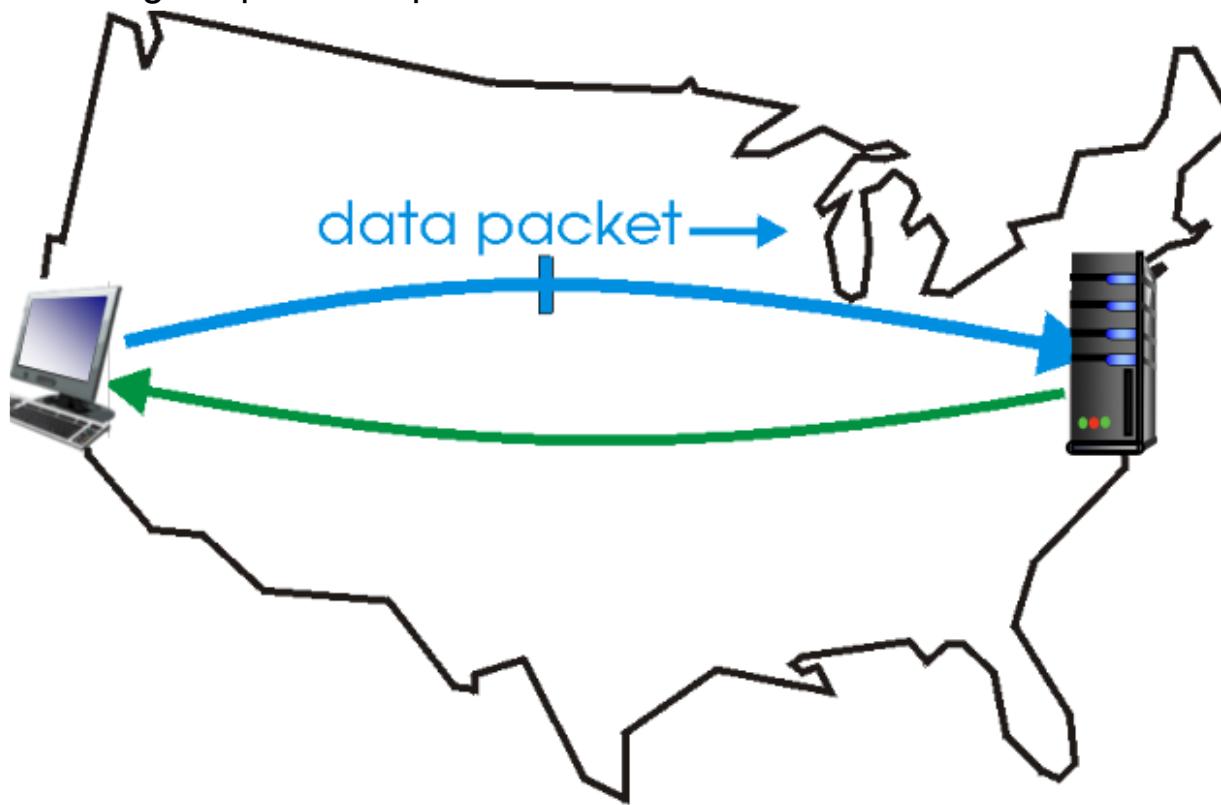
- il throughput effettivo è
 $L/(RTT + L/R) = U_{mittente} \bullet R = 267 kbps$
- le prestazioni del protocollo rdt 3.0 fanno schifo!
- il protocollo limita le prestazioni dell'infrastruttura sottostante (canale)

rdt3.0 : funzionamento con pipeline

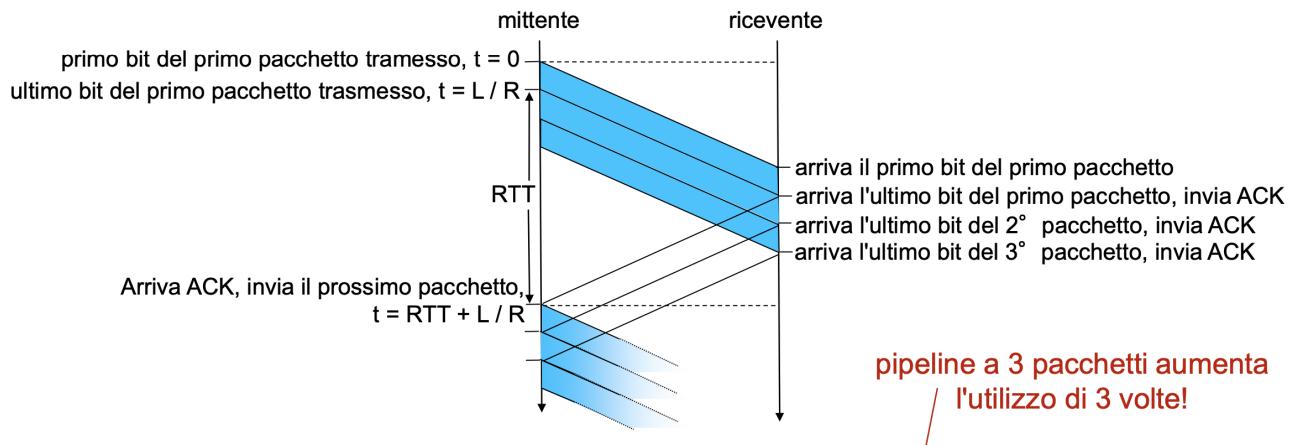
Pipelining : il mittente ammette più pacchetti in transito, ancora da notificare

- l'intervallo dei numeri di sequenza deve essere incrementato

- buffering dei pacchetti presso il mittente e/o ricevente



Pipelining : aumento dell'utilizzo



$$U_{mittente} = \frac{3L/R}{RTT+L/R} = \frac{0.0024}{30.008} = 0.00081$$

Protocolli con pipeline : Go-Back-N (mittente)

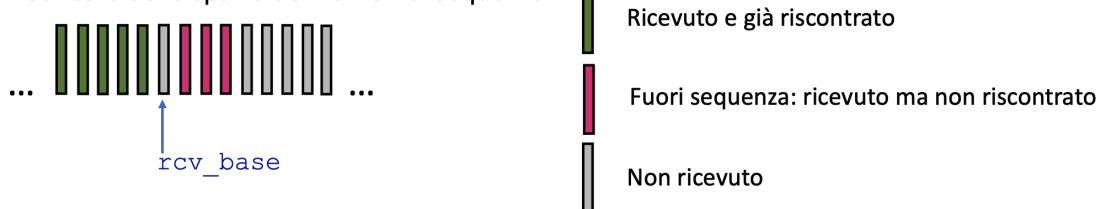
- mittente: "finestra" contenente fino a N pacchetti consecutivi trasmessi ma non riscontrati

- numero di sequenza a k bit nell'intestazione del pacchetto
- **Riscontro cumulativo** : ACK(n): riscontro di tutti i pacchetti con numero di sequenza minore o uguale a n
 - alla ricezione dell'ACK(n): sposta la finestra in avanti per iniziare da $n+1$
- timer per il pacchetto più vecchio in transito
- $timeout(n)$: ritrasmette il pacchetto n e tutti i pacchetti con i numeri di sequenza più grandi

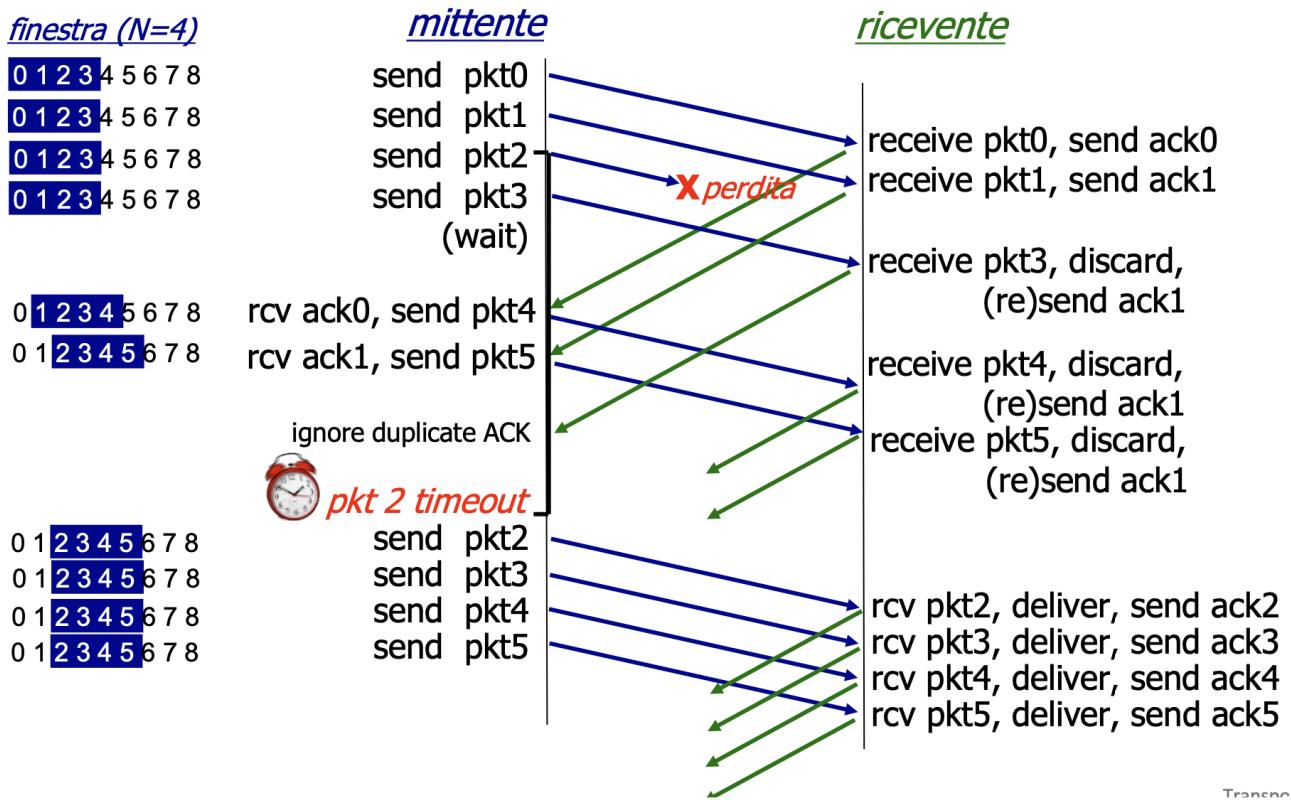
Protocolli con pipeline : Go-Back-N (ricevente)

- Solo ACK: invia sempre un ACK per un pacchetto ricevuto correttamente con il numero di sequenza più alto **in sequenza**
 - potrebbe generare ACK duplicati
 - deve memorizzare soltanto `rcv_base`
- Alla ricezione di un pacchetto fuori sequenza:
 - può scartarlo (non è salvato) o inserirlo in un buffer: una decisione implementativa
 - rimanda un ACK per il pacchetto con il numero di sequenza più alto **in sequenza**

Vista del ricevitore dello spazio dei numeri di sequenza:



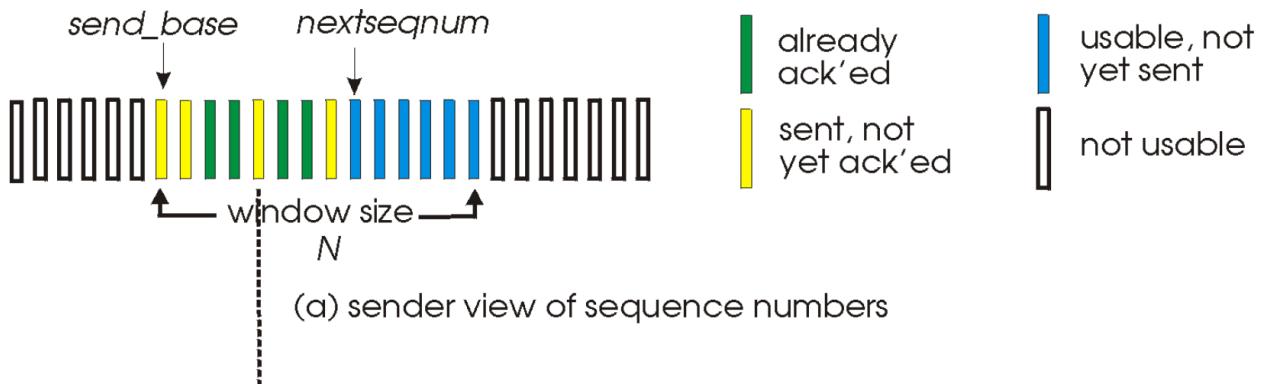
Go-Back-N in azione



Protocolli con pipeline : selective repeat

- **Pipelining** : più pacchetti in transito
- il ricevente riscontra individualmente tutti i pacchetti ricevuti correttamente
 - buffer dei pacchetti , se necessario , per eventuali consegna in sequenza al livello superiore
- mittente :
 - mantiene (concettualmente) un timer per ogni pacchetto non riscontrato
 - timeout : ritrasmette il singolo pacchetto non riscontrato associato al timeout
 - mantiene (concettualmente) una "finestra" su N numeri di sequenza consecutivi
 - limita i pacchetti in pipeline , "in transito" , per rientrare in questa finestra

Selective repeat : finestre del mittente e del ricevente



- **Mittente**

- **Dati dall'alto :**

- se nella finestra è disponibile il successivo numero di sequenza , invia il pacchetto

- **Timeout(n) :**

- ritrasmette il pacchetto n, riparte il timer

- **ACK(n)** in $[sendbase, sendbase + N - 1]$:

- marca il pacchetto n come ricevuto
 - se n è il numero di sequenza più piccolo, la base della finestra avanza al successivo numero di sequenza del pacchetto non riscontrato

- **Ricevente**

- **pacchetto n in $[rcvbase, rcbase+N-1]$**

- invia ACK(n)

- fuori sequenza: buffer

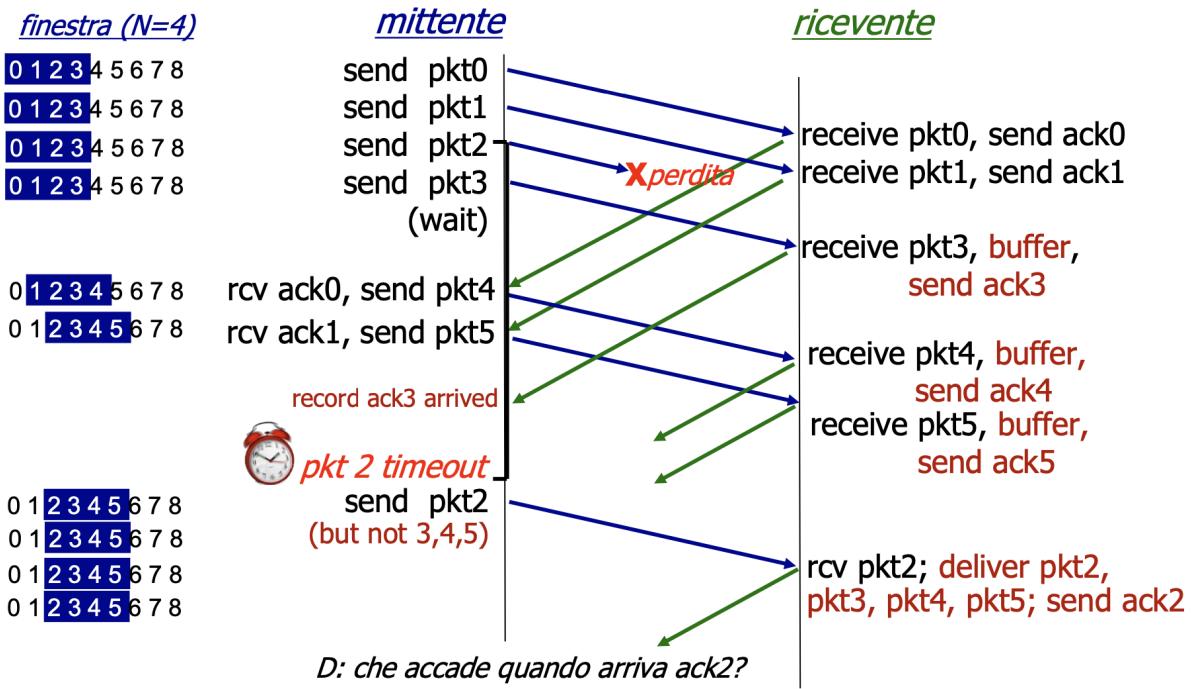
- in sequenza: consegna (vengono consegnati anche i pacchetti bufferizzati in sequenza), la finestra avanza al successivo pacchetto non ancora ricevuto

- **pacchetto n in $[rcvbase-N, rcbase-1]$**

- ACK(n)

- **altrimenti :**

- ignora

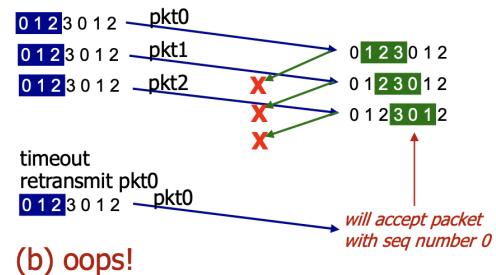
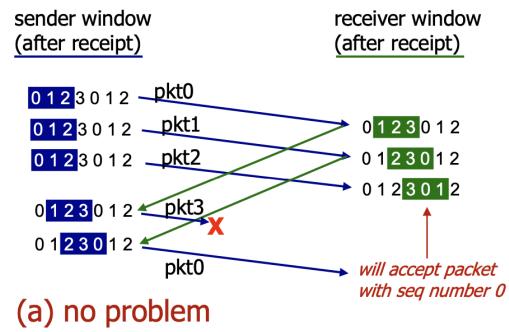


Selective repeat: dilemma!

esempio:

- numeri di sequenza: 0, 1, 2, 3
- dimensione della finestra = 3

I numeri di sequenza sono usati *ciclicamente* -> aritmetica modulare

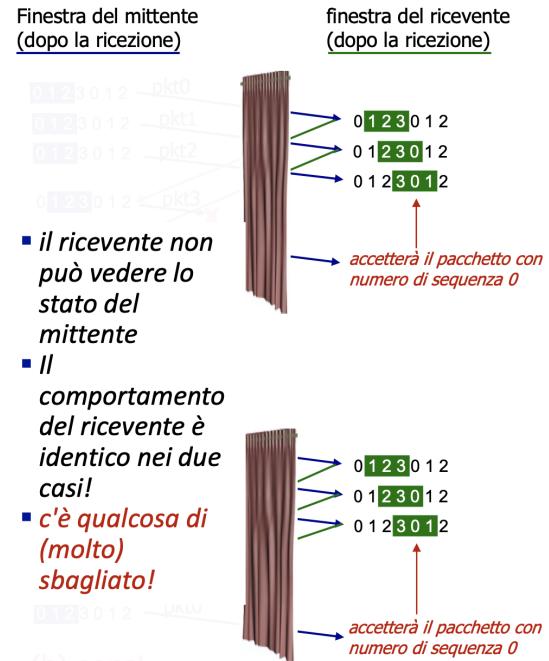


Selective repeat: dilemma!

esempio:

- numeri di sequenza: 0, 1, 2, 3
- dimensione della finestra = 3

Q: Quale relazione è necessaria tra la dimensione dei numeri di sequenza e la dimensione della finestra per evitare il problema nello scenario (b)?



Relazione tra dimensione della finestra e dei numeri di sequenza (sia SR sia GBN)

Q : Quale relazione è necessaria tra la dimensione dei numeri di sequenza e la dimensione della finestra per evitare il problema nello scenario (b)?

Il problema è che a causa della mancata ricezione degli ACK la finestra del ricevente può andare avanti rispetto a quella del mittente: si consideri il caso peggiore associato all'invio di un'intera finestra di pacchetti e la mancata ricezione di tutti gli ACK.

È necessario che lo spazio dei numeri di sequenza sia sufficientemente grande da contenere sia la finestra del mittente sia la finestra del destinatario senza che si sovrappongano (considerando l'operazione di modulo)

Sia w la dimensione della finestra e m la dimensione dello spazio dei numeri di sequenza:



Se i numeri di sequenza sono espressi tramite k bit, allora $m = 2^k$
 $w \leq 2^k/2$ cioè $w \leq 2^{k-1}$