Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2023/2024

Livello di trasporto (parte2)

dr. Manuel Fiorelli

manuel.fiorelli@uniroma2.it

https://art.uniroma2.it/fiorelli

Basate sulle slide del libro di testo:
https://gaia.cs.umass.edu/kurose_ross/ppt.php

Capitolo 3: tabella di marcia

- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP
- Principi del trasferimento dati affidabile
- Trasporto orientato alla connessione: TCP
- Principi del controllo della congestione
- Controllo della congestione TCP
- Evoluzione della funzionalità del livello di trasporto

Principi del trasferimento dati affidabile

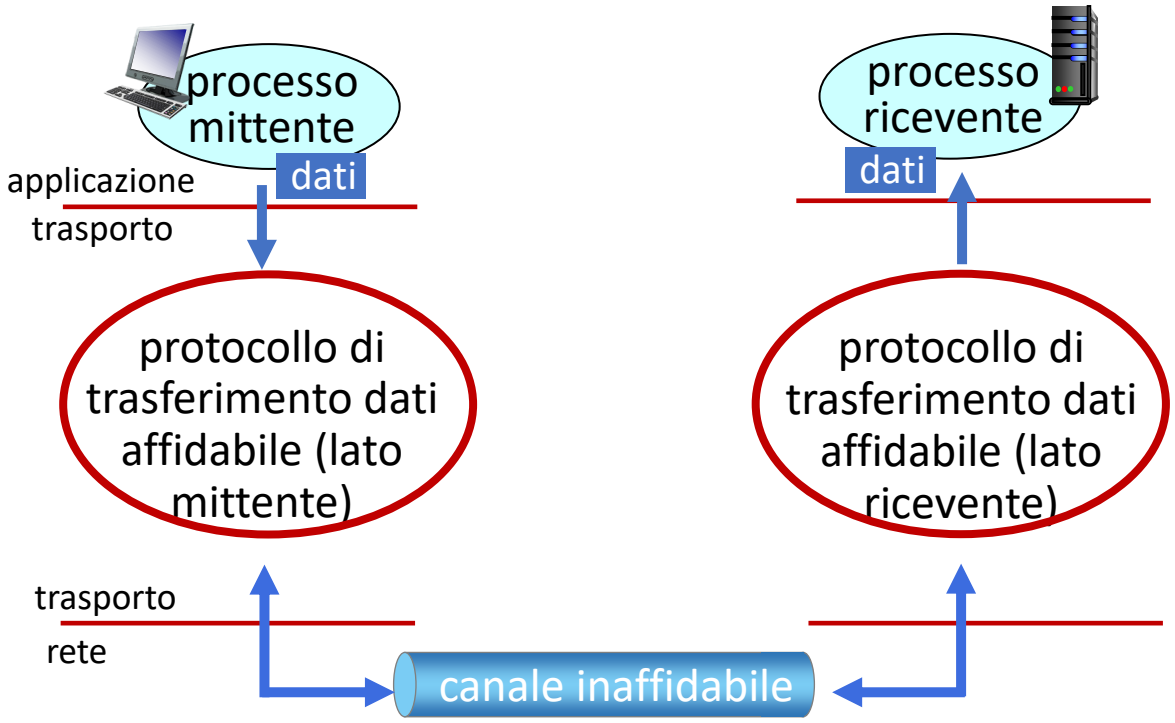
astrazione di un servizio affidabile

per il momento ci concentriamo su un canale unidirezionale, per l'invio affidabile di dati da un mittente a un destinatario.

Principi del trasferimento dati affidabile

astrazione di un servizio affidabile

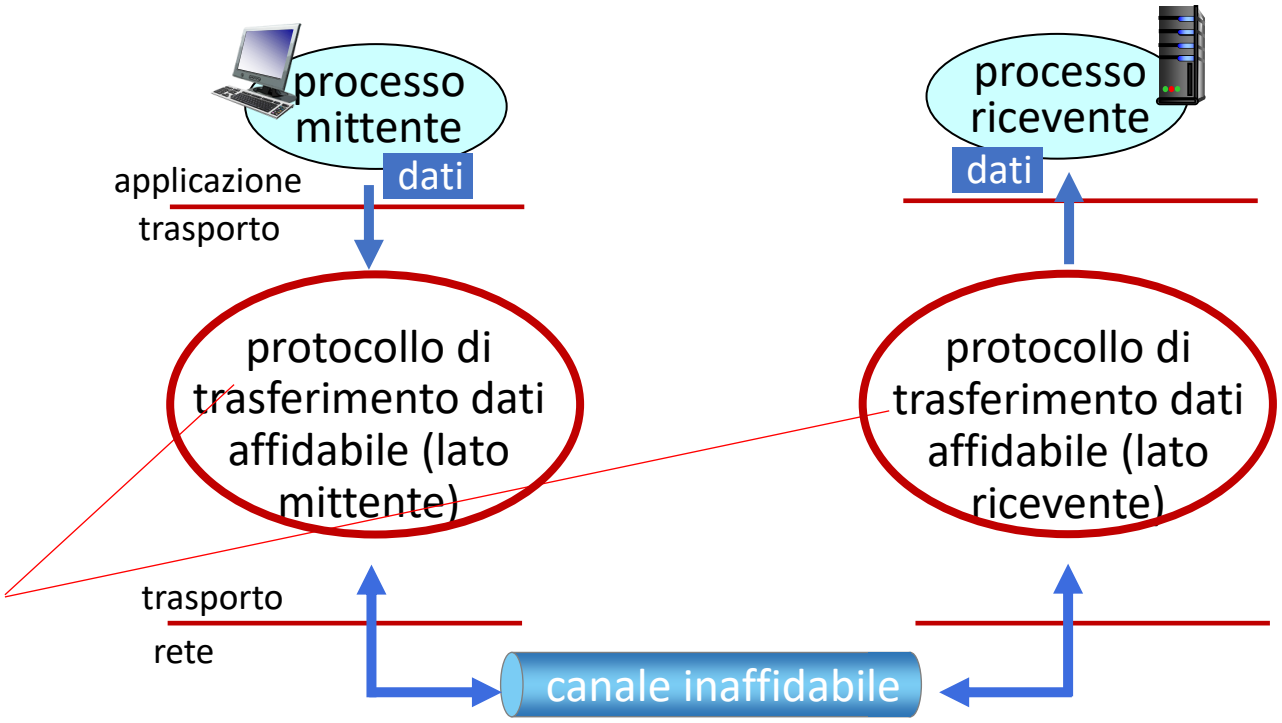
nonostante il nostro sia a livello astratto un canale di comunicazione unidirezionale, le due parti dentro al livello di trasporto devono comunicare in entrambi i versi usando il canale *inaffidabile* fornito dal livello di rete.



Implementazione di un servizio affidabile

Principi del trasferimento dati affidabile

la complessità del protocollo di trasferimento dati affidabile (reliable data transfer, rdt) dipende (fortemente) dalle caratteristiche del canale inaffidabile (perdite, corruzione, riordino dei dati?)

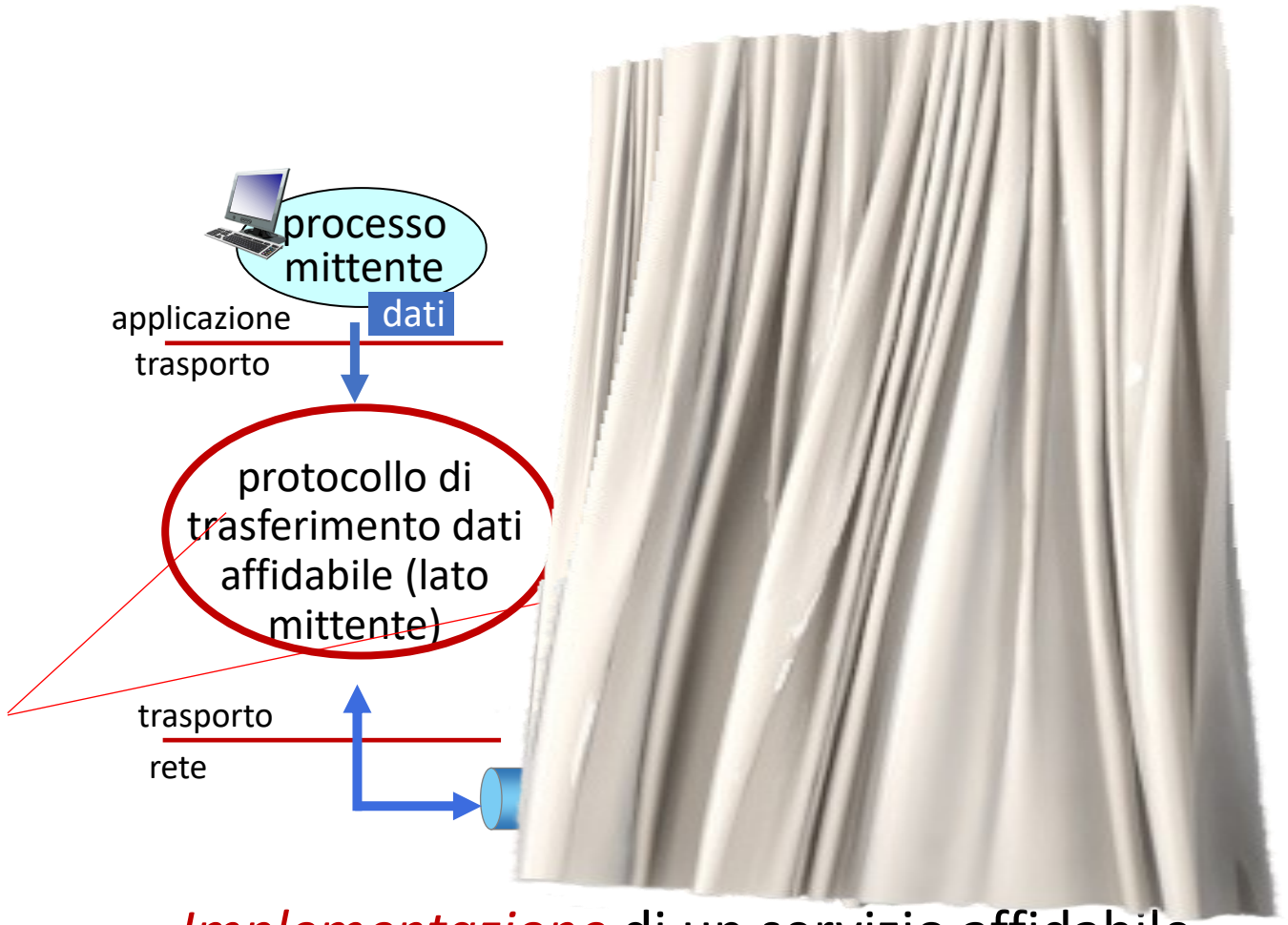


Implementazione di un servizio affidabile

Principi del trasferimento dati affidabile

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

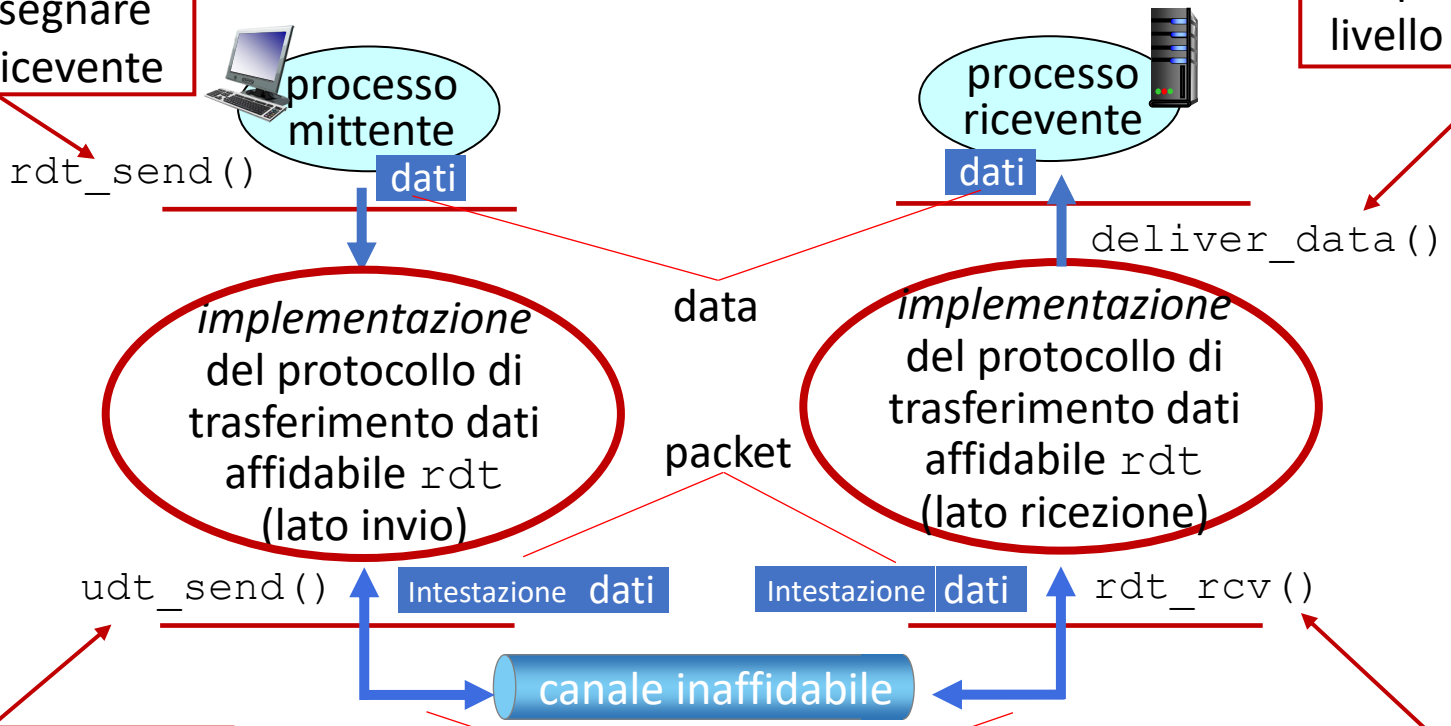


Implementazione di un servizio affidabile

Trasferimento dati affidabile (rdt): interfacce

rdt_send(): chiamata dall'alto (ad esempio, dall'applicazione). Trasferisce i dati da consegnare al livello superiore del ricevente

deliver_data(): chiamata da rdf per consegnare i dati al livello superiore



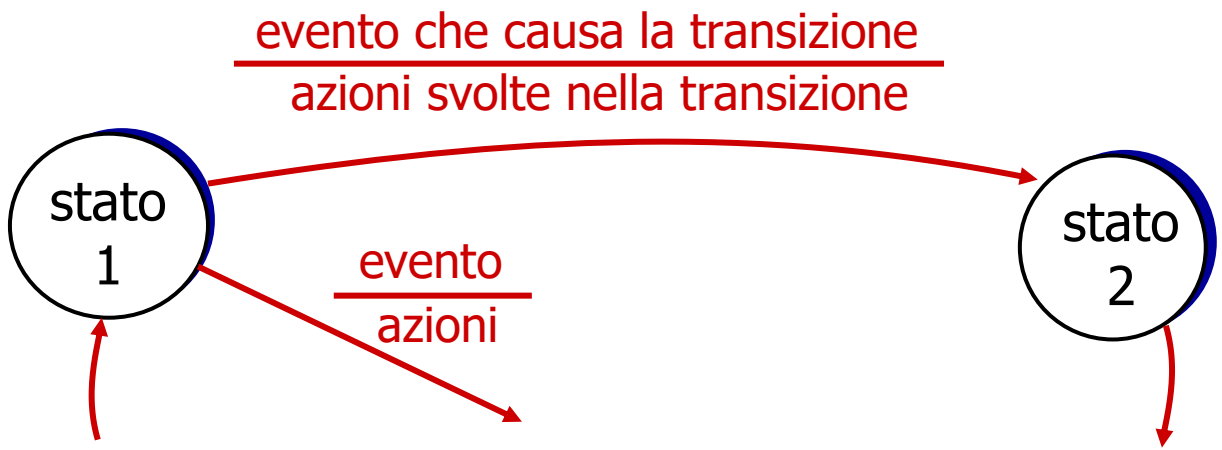
udt_send(): chiamata da rdt per trasferire il pacchetto al ricevente tramite il canale inaffidabile

comunicazione bidirezionale tramite un canale inaffidabile

rdt_rcv(): chiamata quando il pacchetto arriva nel lato ricezione del canale

Trasferimento dati affidabile: come iniziare

- svilupperemo progressivamente i lati d'invio e di ricezione di un protocollo di trasferimento dati affidabile, reliable data transfer protocol (`rdt`)
- considereremo soltanto i trasferimenti dati unidirezionali
  - ma le informazioni di controllo fluiranno in entrambe le direzioni!
- utilizzeremo macchine a stati finiti, finite state machines (FSM) per specificare il mittente e il ricevente

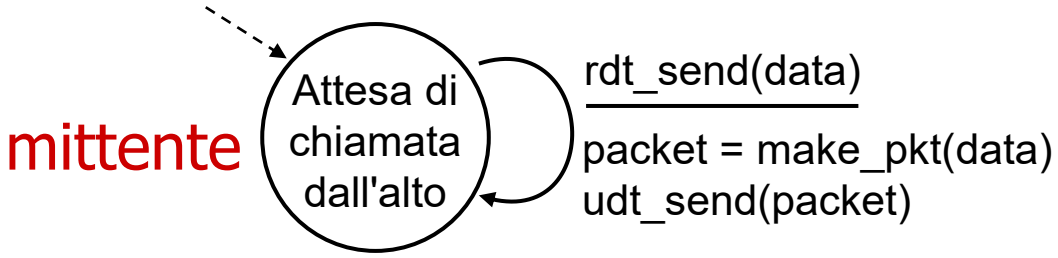


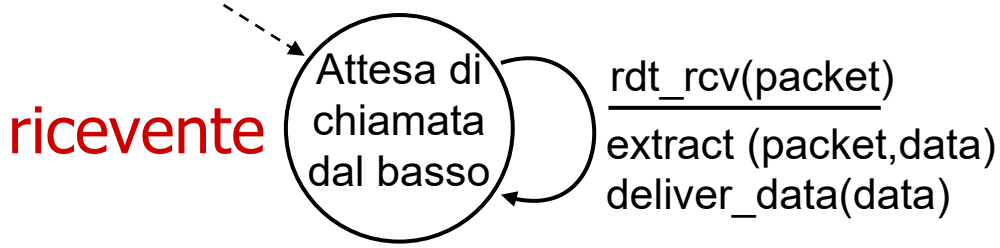
stato: lo stato successivo a questo è determinato unicamente dall'evento successivo

Una specifica formale di un protocollo è meno soggetta a fraintendimenti e incompletezza rispetto a una specifica testuale. Sulla base di una specifica formale è inoltre possibile **provare** che il protocollo specificato abbia determinate proprietà.

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?

Come fanno gli esseri umani a recuperare dagli "errori" durante la conversazione?

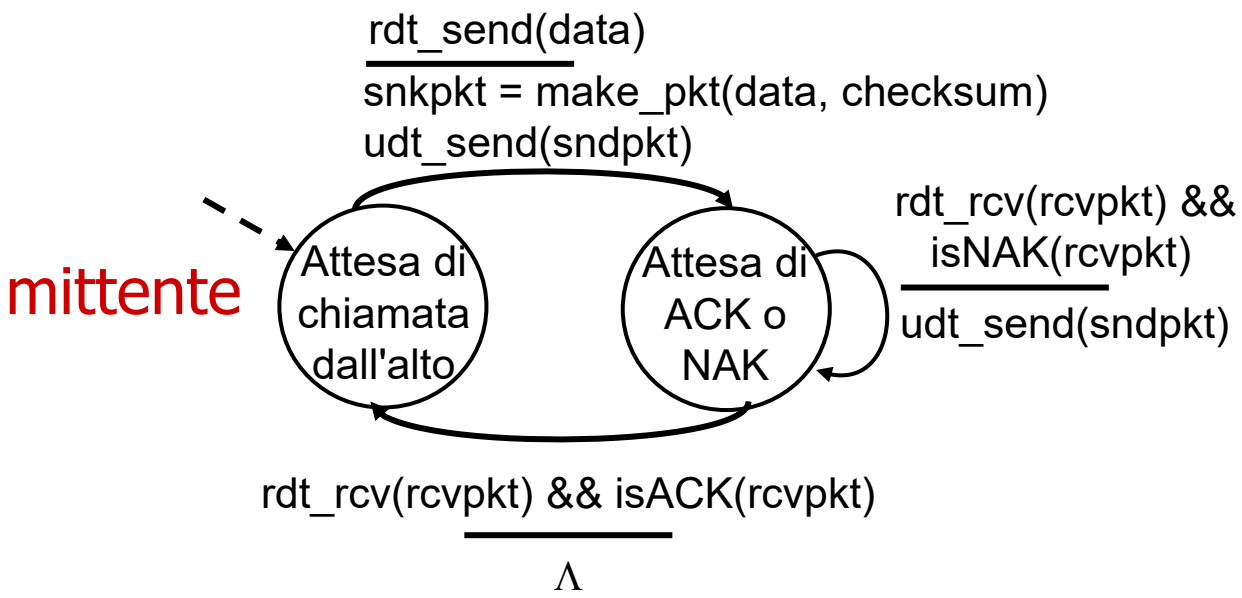
rdt2.0: canale con errori nei bit

- il canale sottostante può invertire (*flip*) i bit nel pacchetto
  - checksum per rilevare gli errori nei bit (*error detection*)
- *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 *ritrasmette* il pacchetto se riceve un NAK

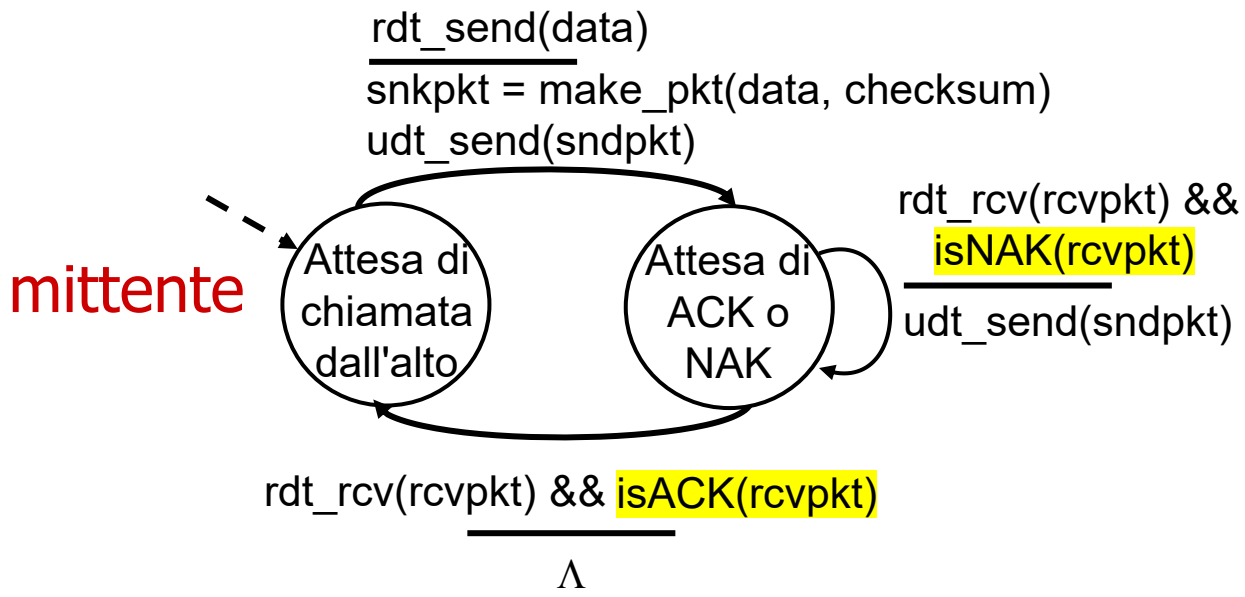
stop and wait

il mittente invia un pacchetto, quindi attende la risposta del destinatario

rdt2.0: specifica delle FSM



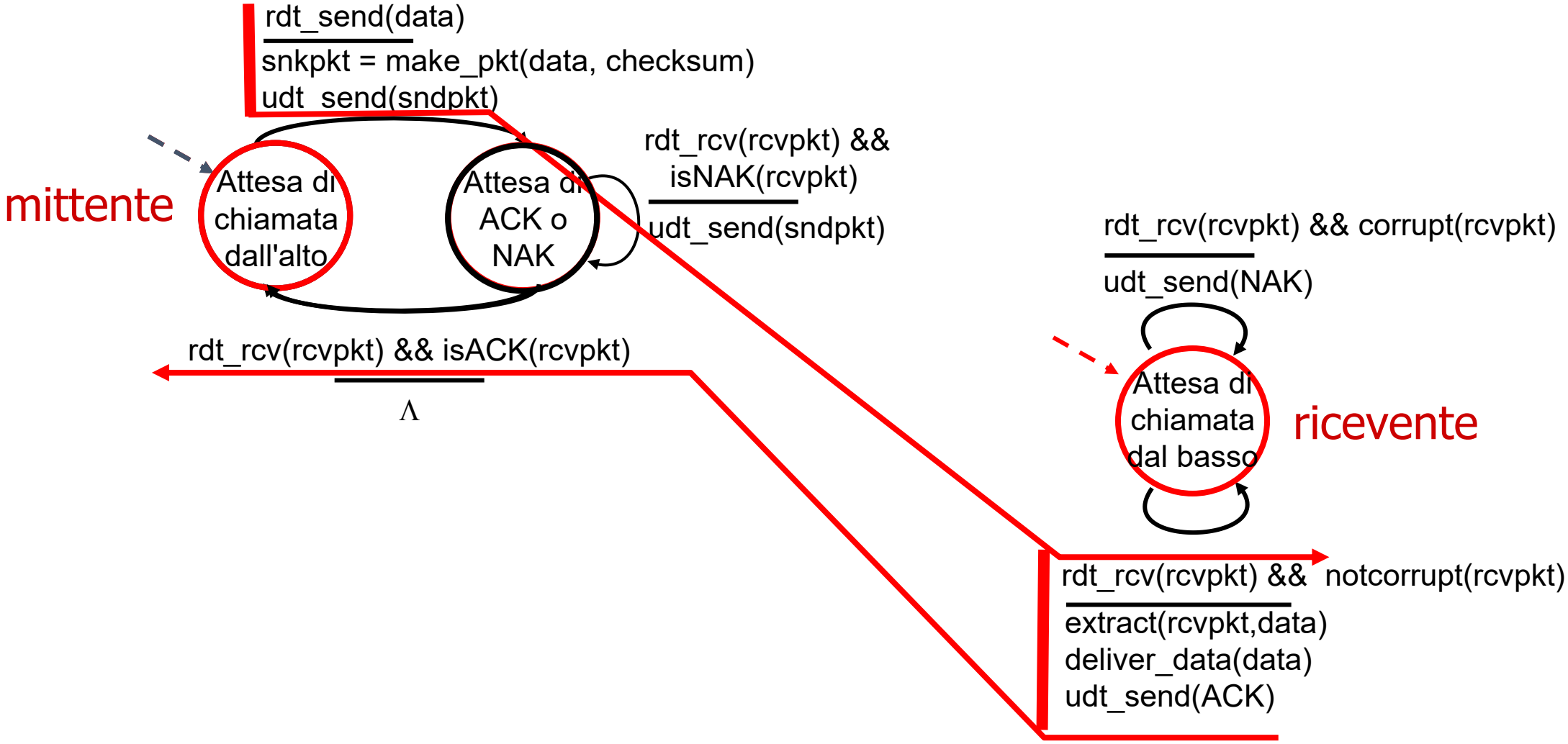
rdt2.0: specifica delle FSM



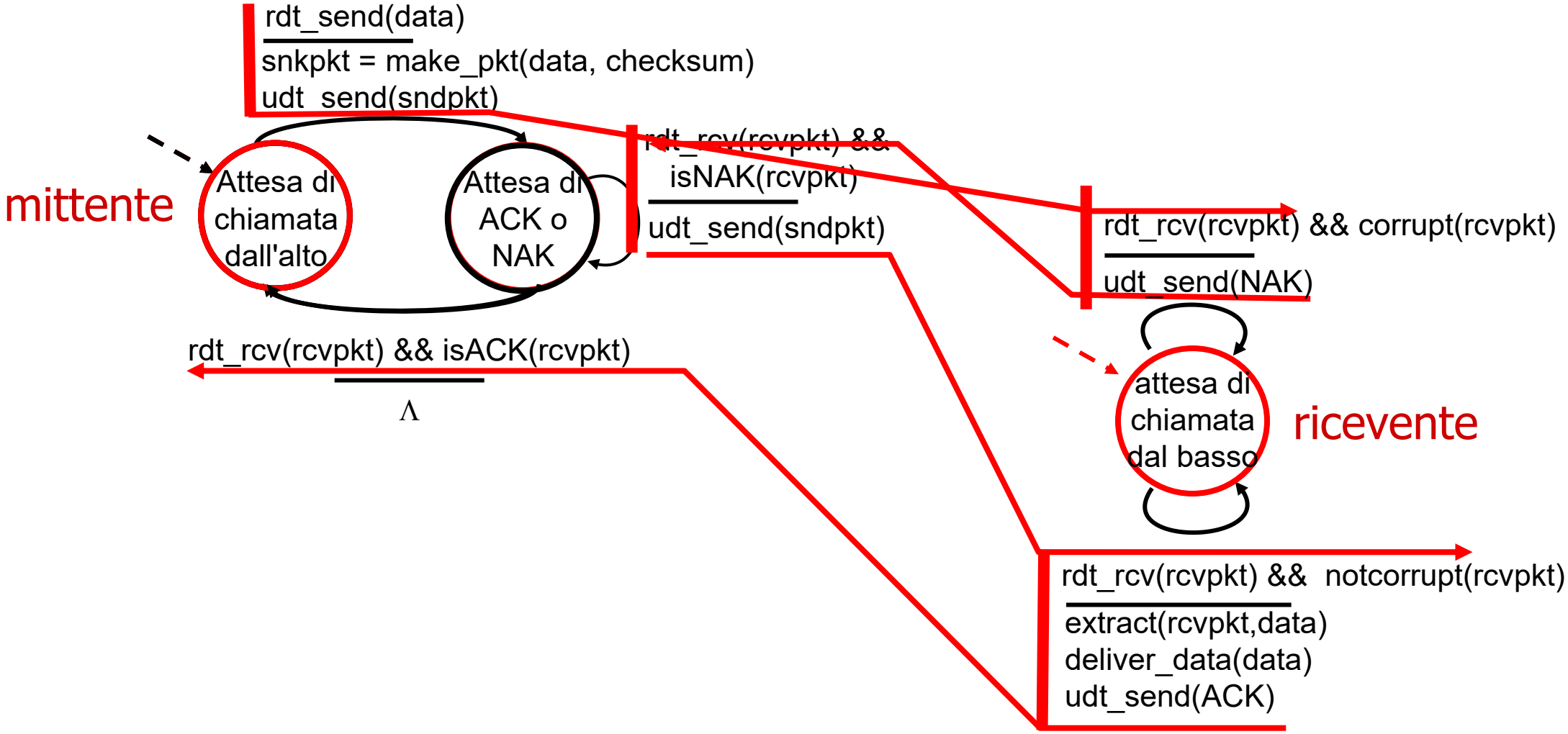
Nota: lo "stato" del destinatario (ha ricevuto correttamente il mio messaggio?) non è noto al mittente a meno che non venga comunicato in qualche modo dal destinatario al mittente.

- ecco perché abbiamo bisogno di un protocollo!

rdt2.0: operazione 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

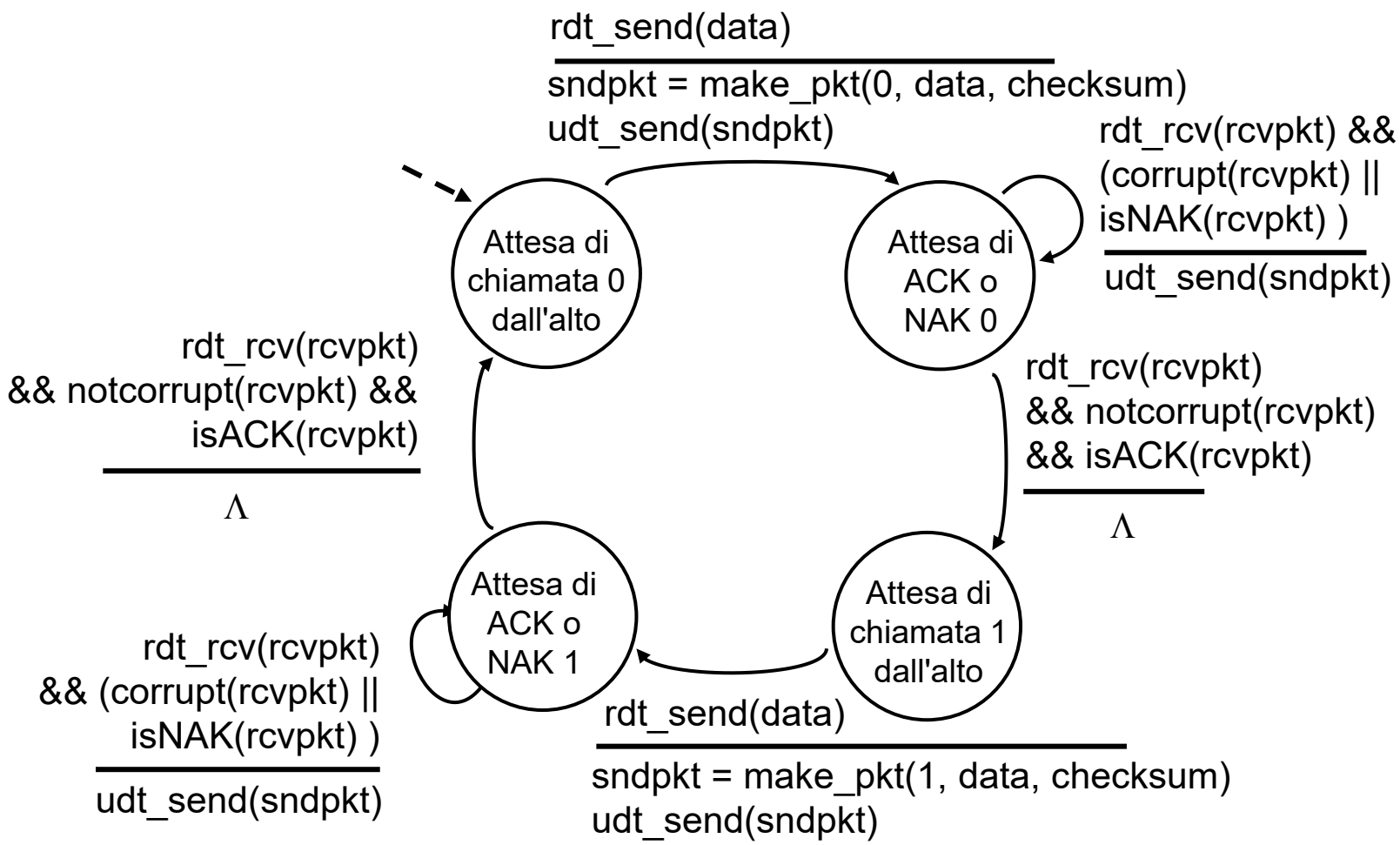
gestione dei duplicati

- 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

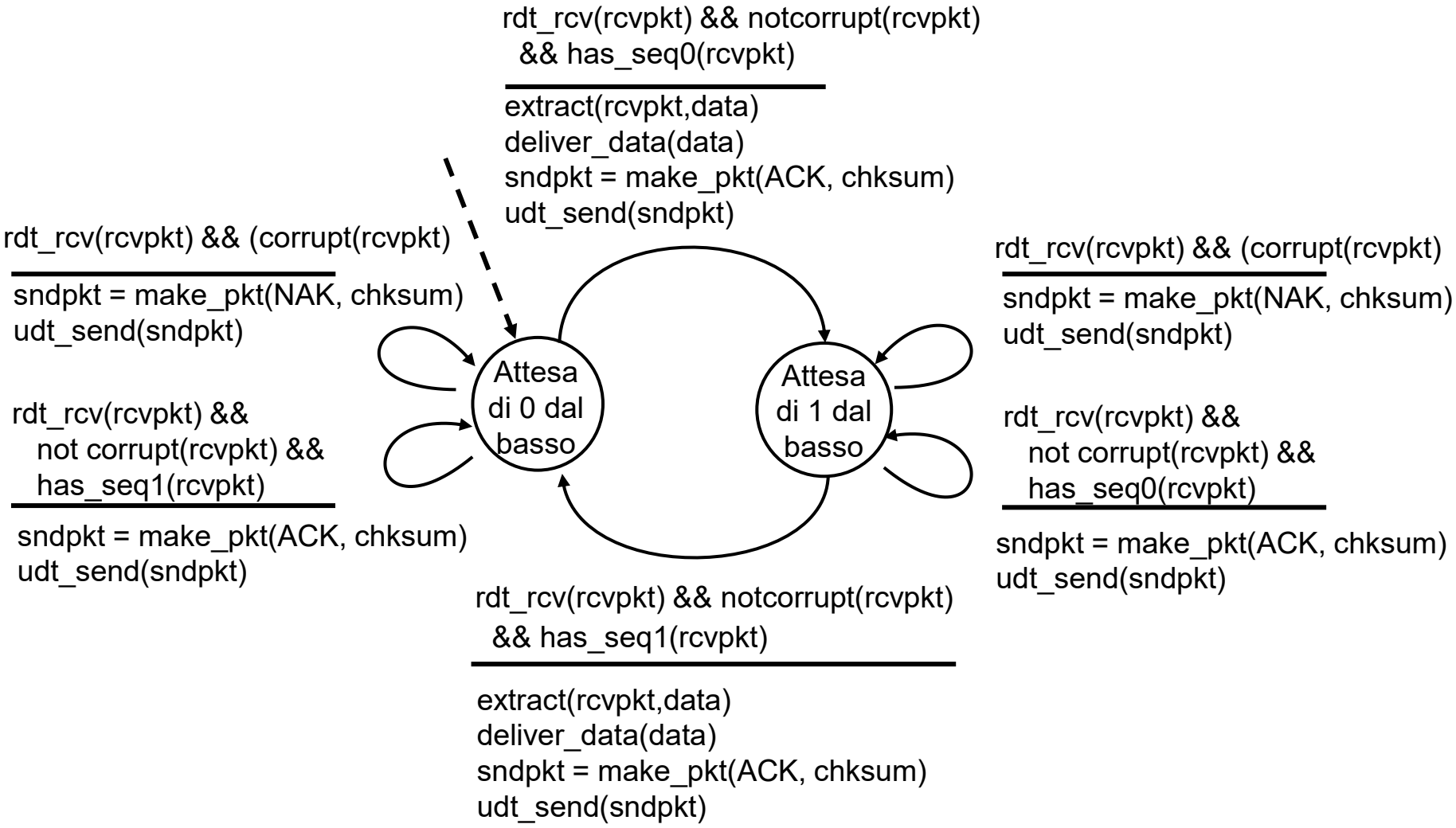
stop and wait

Il mittente invia un pacchetto, poi aspetta la risposta del destinatario

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



rdt2.1: il ricevente 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:

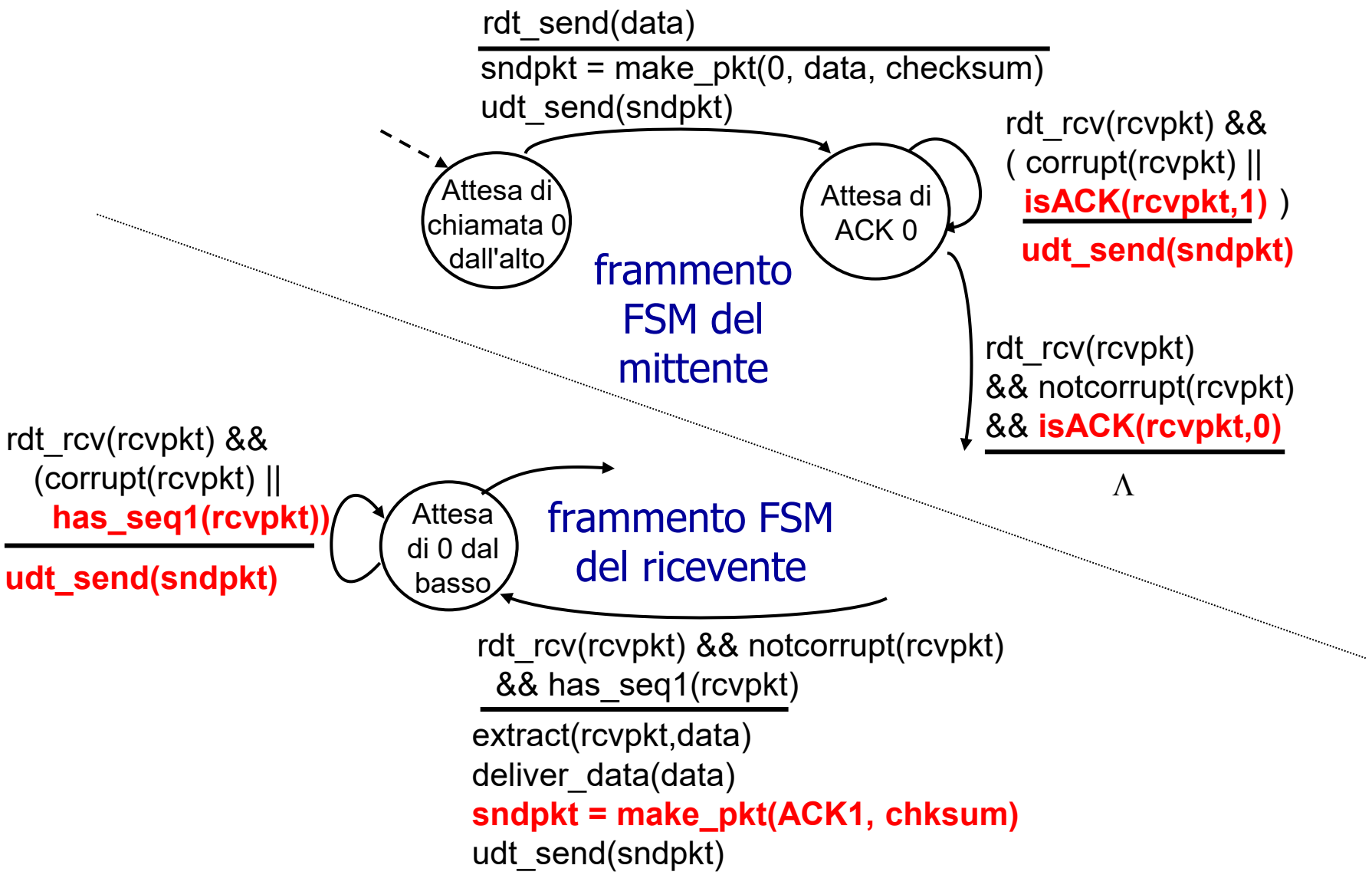
- deve controllare se il pacchetto ricevuto è duplicato
  - lo stato indica se il numero di sequenza previsto è 0 o 1
- nota: il ricevente *non* può sapere se il suo ultimo ACK/NAK è stato ricevuto correttamente dal mittente

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.

rdt2.2: frammenti del mittente e del ricevente



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

D: Come gestiscono gli *esseri umani* le parole perse da mittente a destinatario nelle conversazioni?

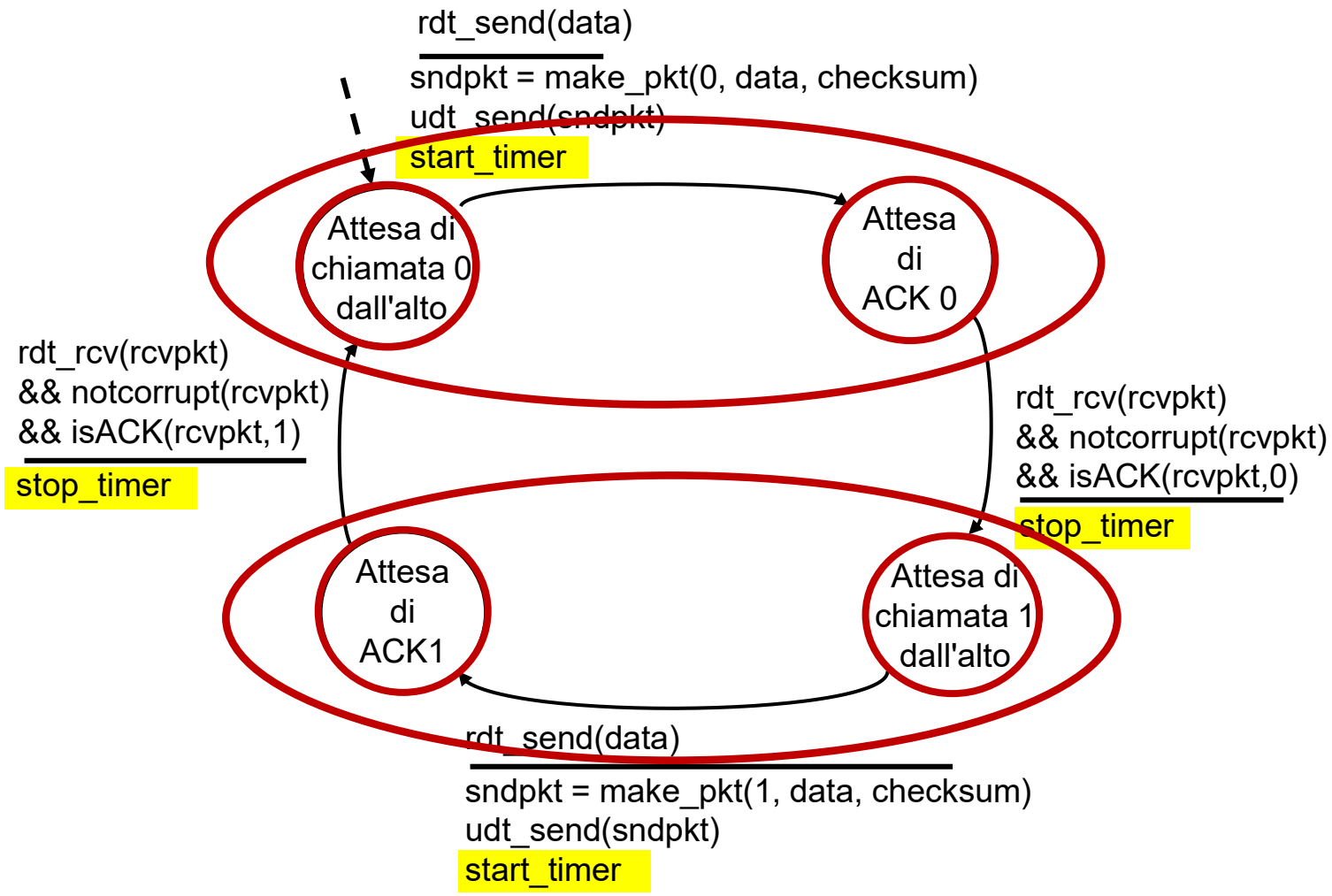
rdt3.0: canali con errori *e perdite*

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

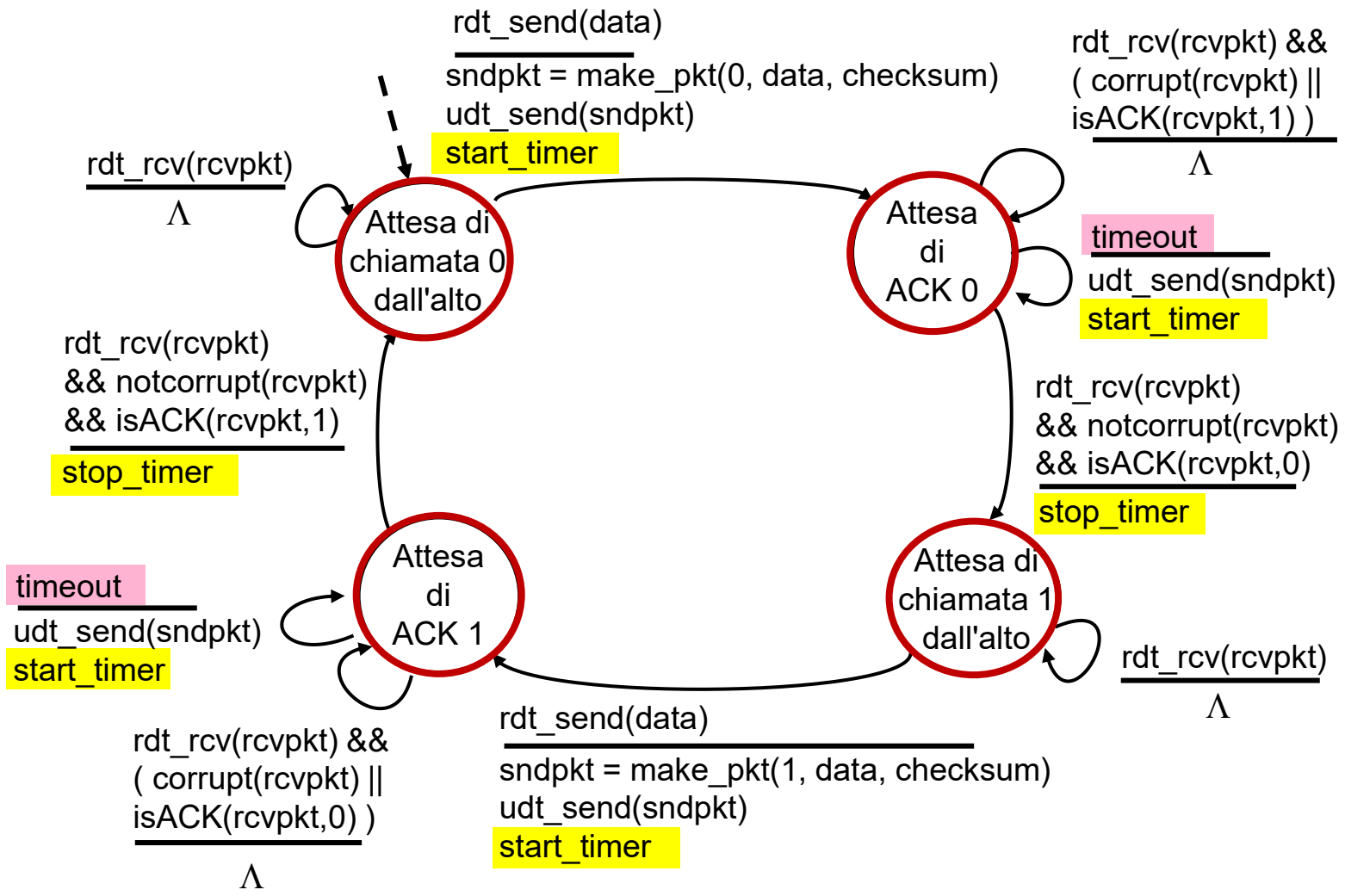
- ritrasmette se non riceve un ACK in questo periodo
- se il pacchetto (o l'ACK) è soltanto in ritardo (non perso)
  - la ritrasmissione sarà duplicata, ma l'uso dei numeri di sequenza gestisce già questo
  - il destinatario deve specificare il numero di sequenza del pacchetto da riscontrare
- utilizzare un timer per il conto alla rovescia (countdown timer) per interrompere (interrupt) dopo un periodo di tempo "ragionevole

timeout

rdt3.0 mittente



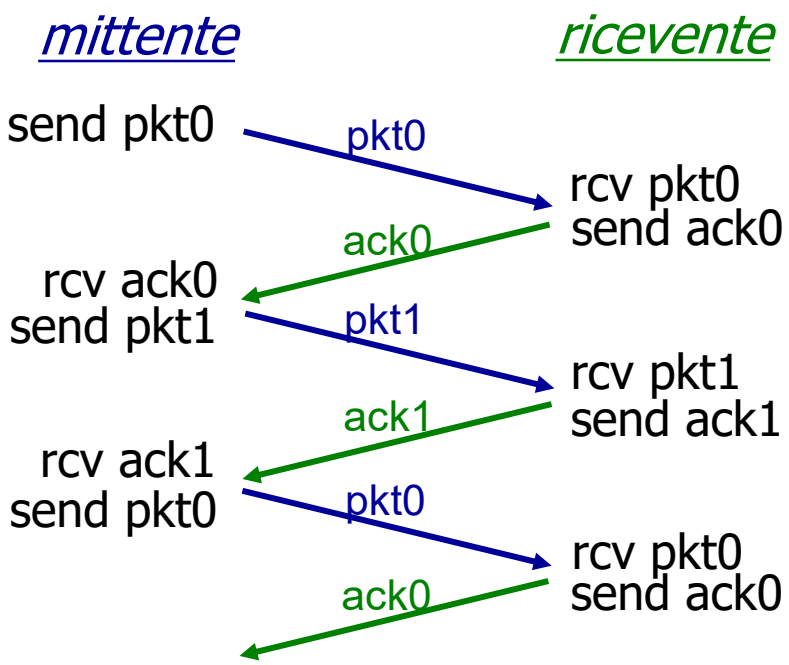
rdt3.0 mittente



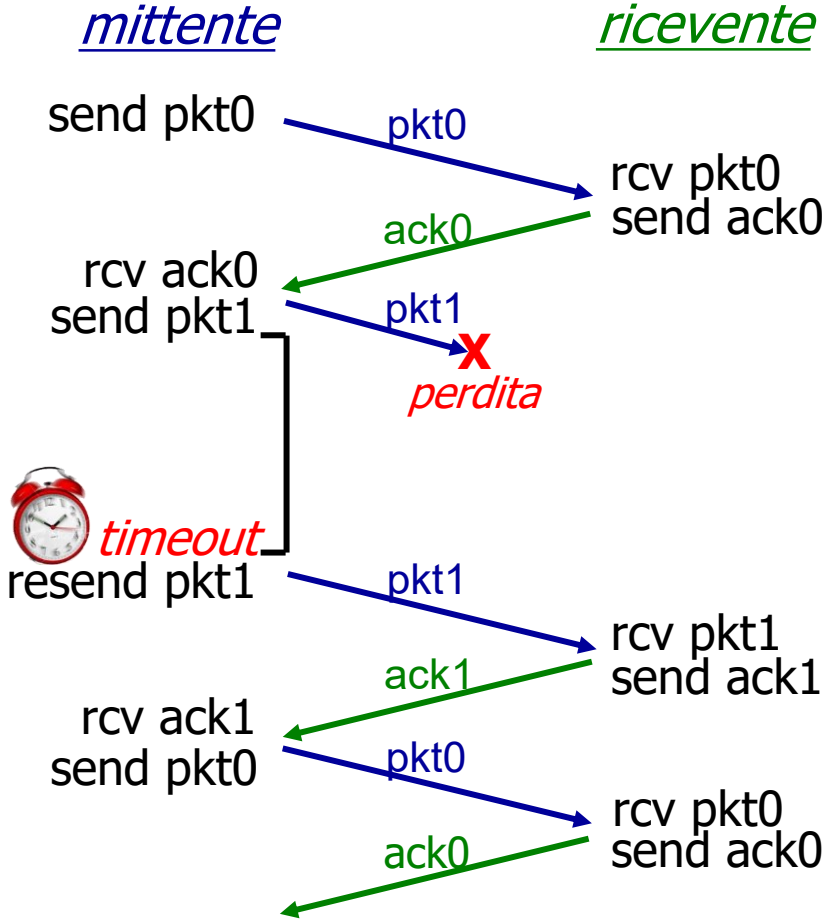
rdt3.0 ricevente

- vedi rdt 2.1

rdt3.0 in azione

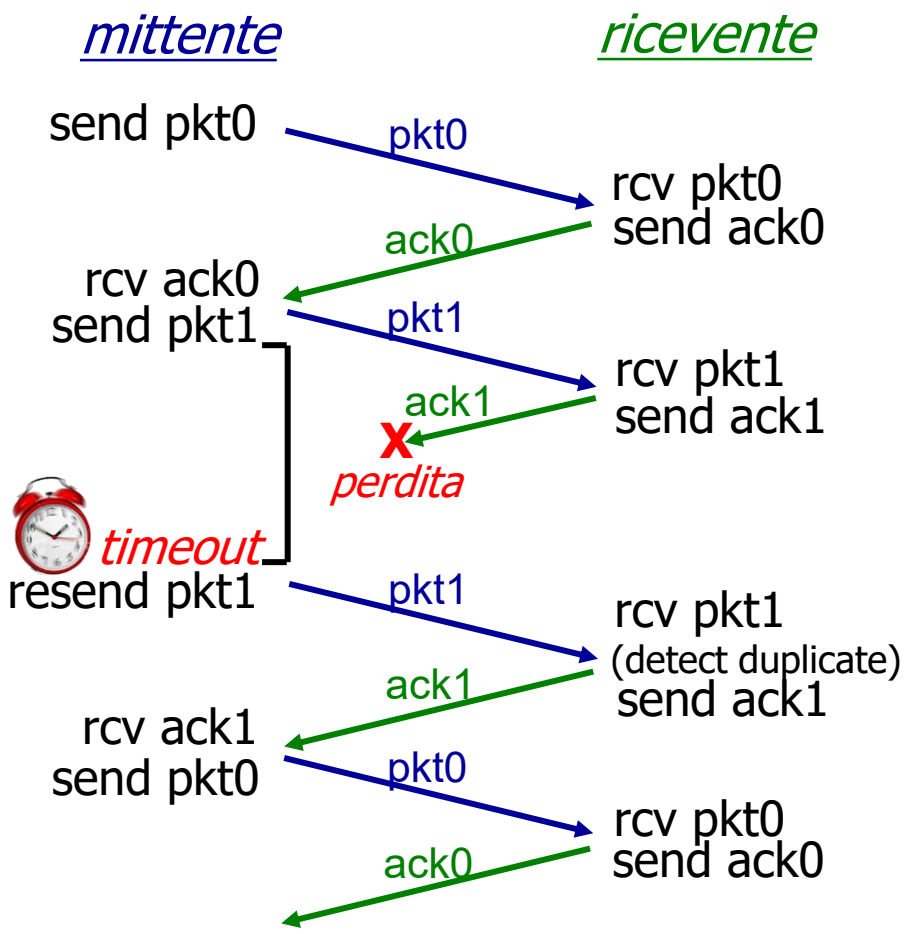


(a) operazioni senza perdita

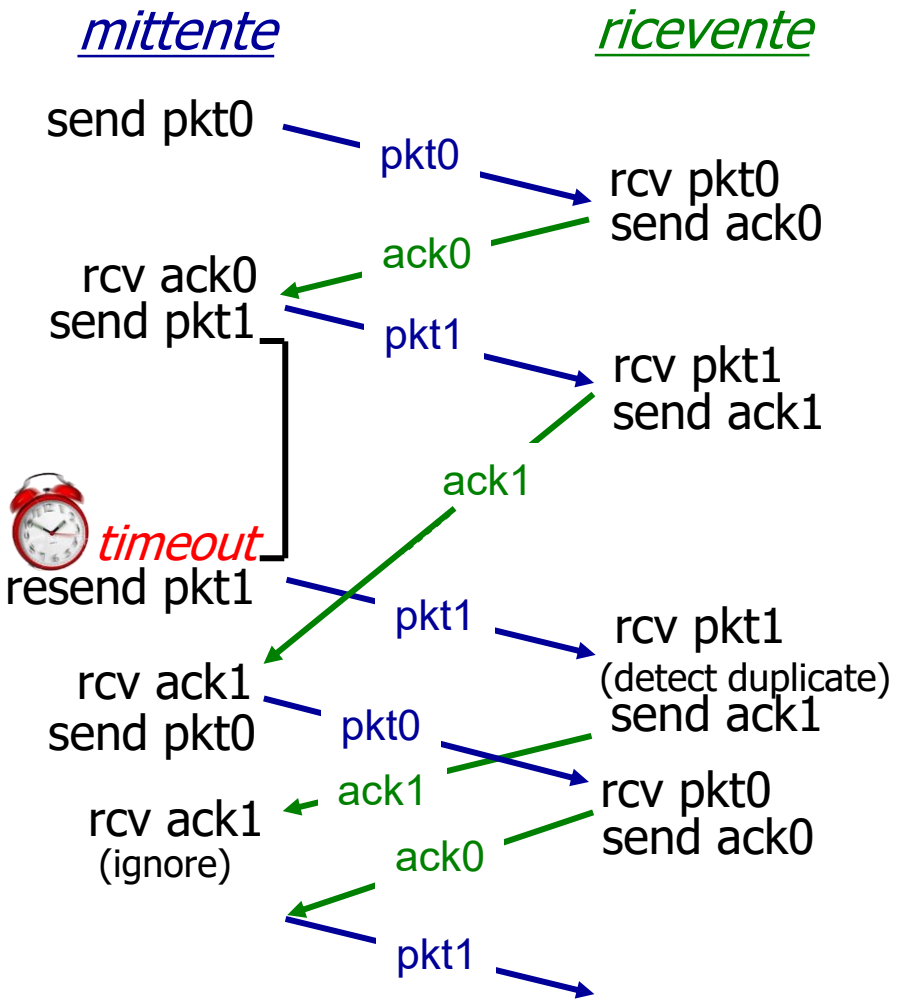


(b) perdita di pacchetto

rdt3.0 in azione



(c) perdita di ACK



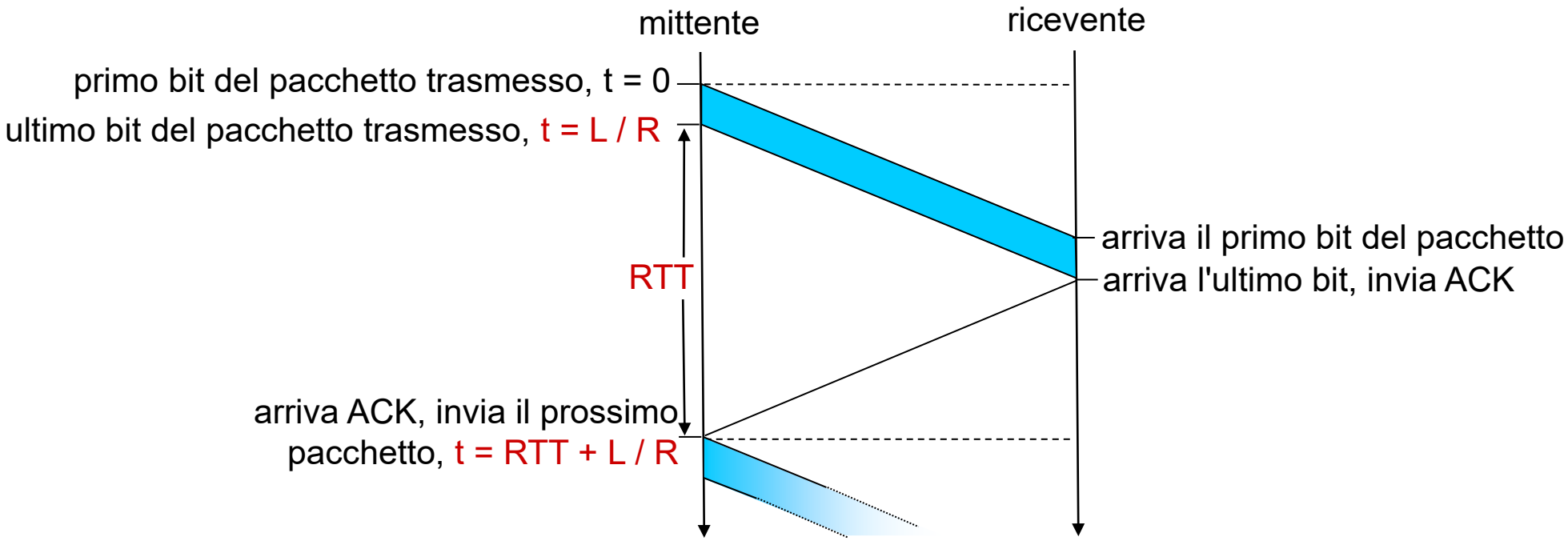
(d) timeout prematuro / ACK ritardato

Prestazioni di 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:

$$D_{trasm} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ \mu s$$

rdt3.0: funzionamento con stop-and-wait

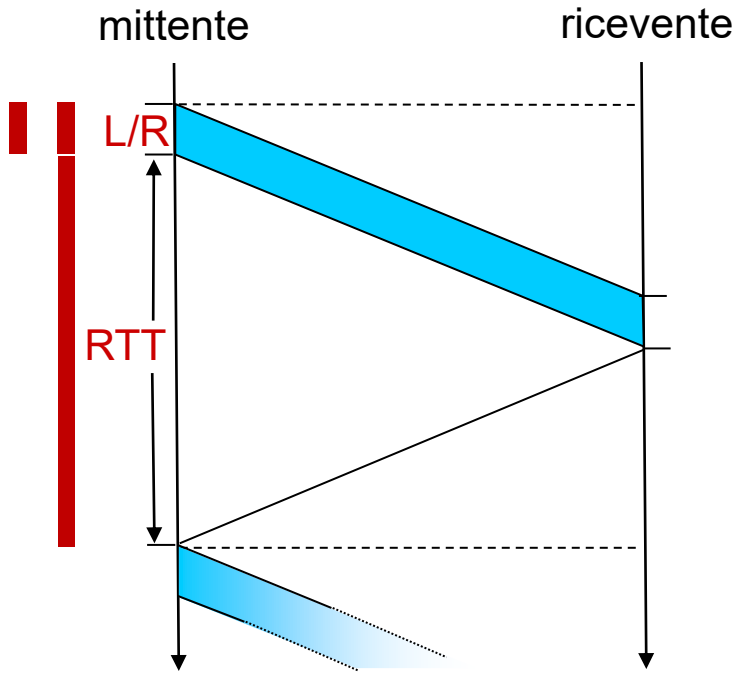


rdt3.0: funzionamento con stop-and-wait

$$U_{mittente} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.000267$$



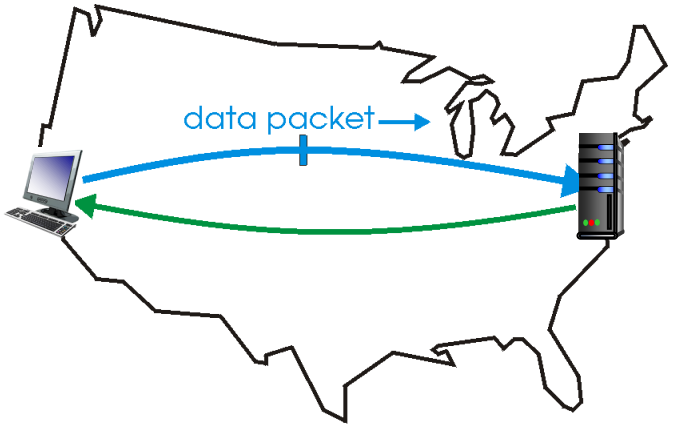
- il throughput effettivo generato dal mittente:

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

rdt3.0: funzionamento con pipeline

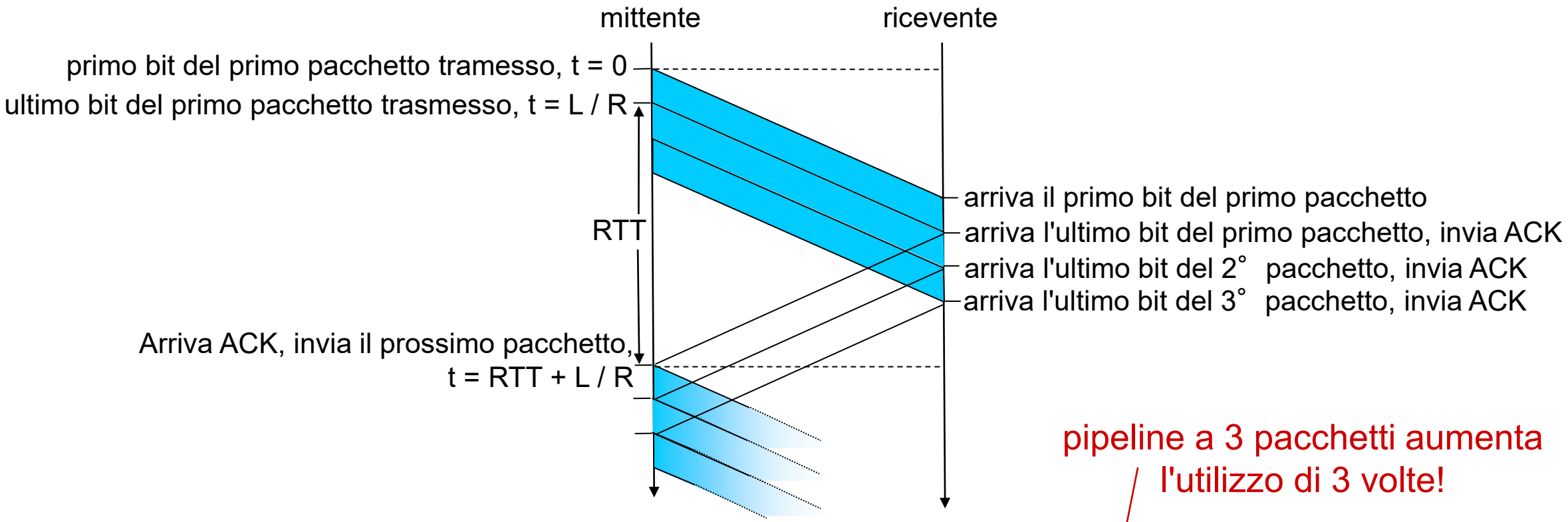
pipelining: il mittente ammette più pacchetti in transito, ancora da notificare

- l'intervallo dei numeri di sequenza deve essere incrementato
- buffering dei pacchetti presso il mittente e/o ricevente



(a) a stop-and-wait protocol in operation

Pipelining: aumento dell'utilizzo

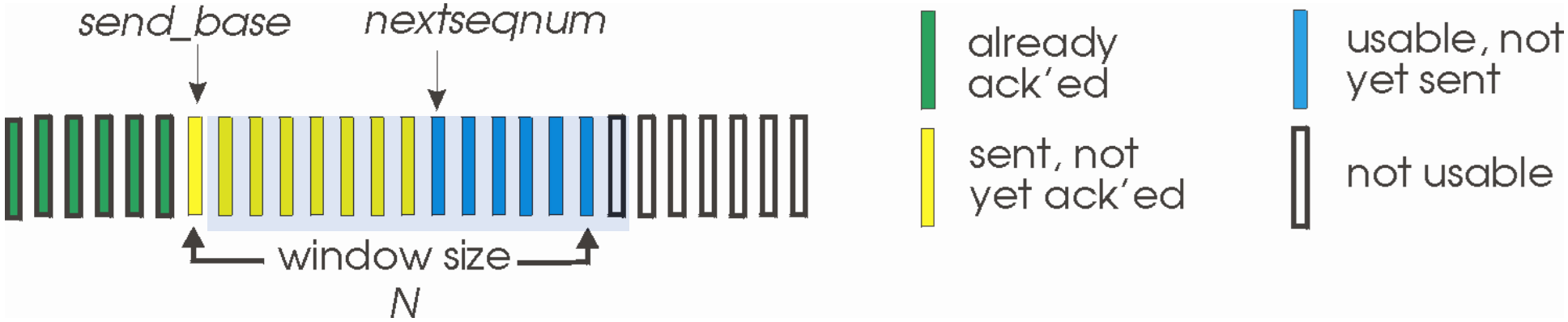


pipeline a 3 pacchetti aumenta l'utilizzo di 3 volte!

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

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

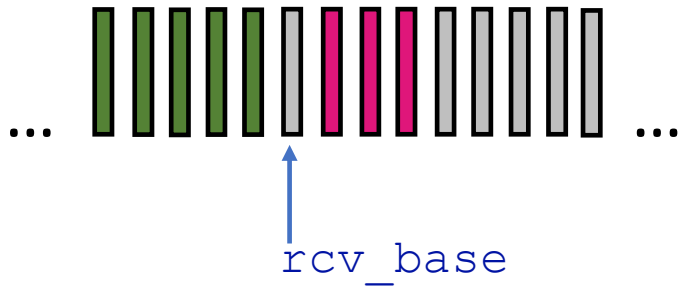


- *Riscontro cumulativo:* ACK(*n*): riscontro di tutti i pacchetti con numero di sequenza minore o uguale a *n*
  - alla ricezione dell'ACK(n): sposta la finestra in avanti per iniziare da n+1
- timer per il pacchetto più vecchio in transito
- *timeout(n):* ritrasmette il pacchetto n e tutti i pacchetti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

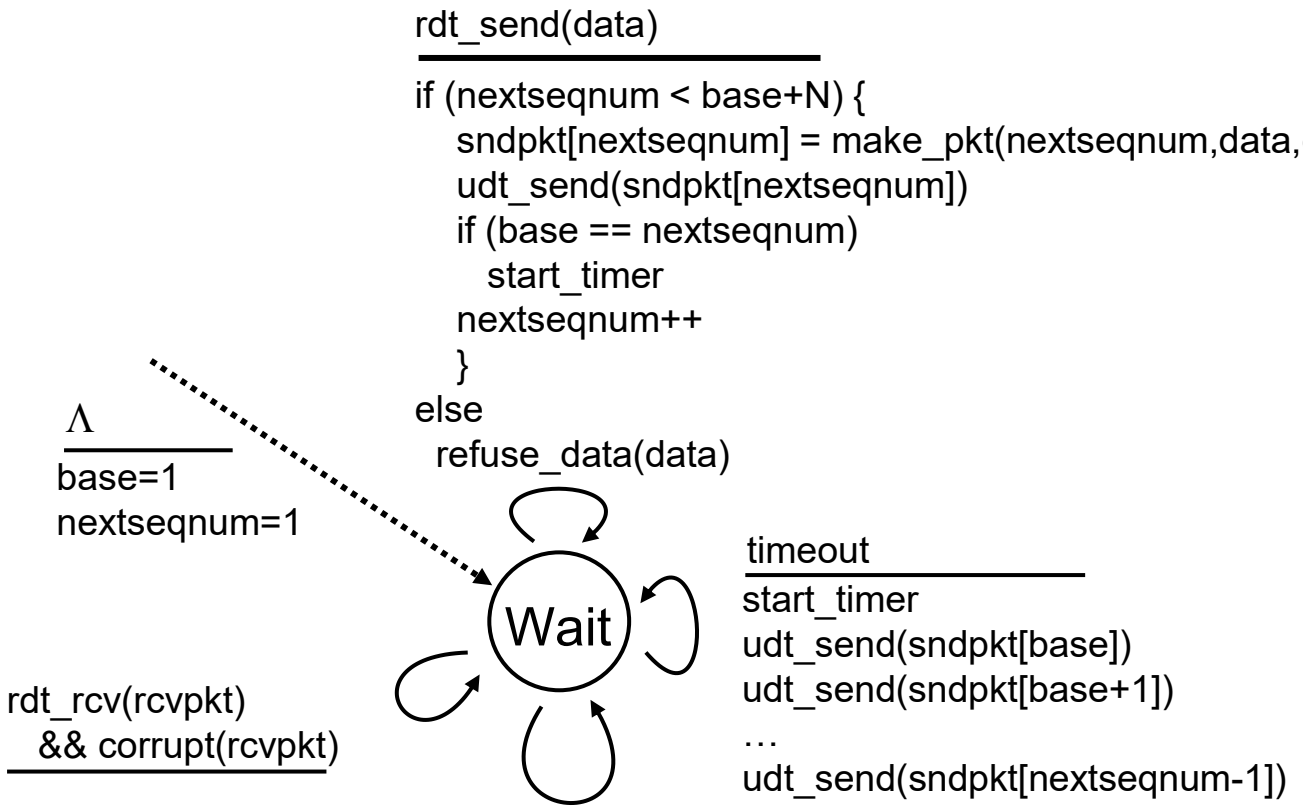


Ricevuto e già riscontrato

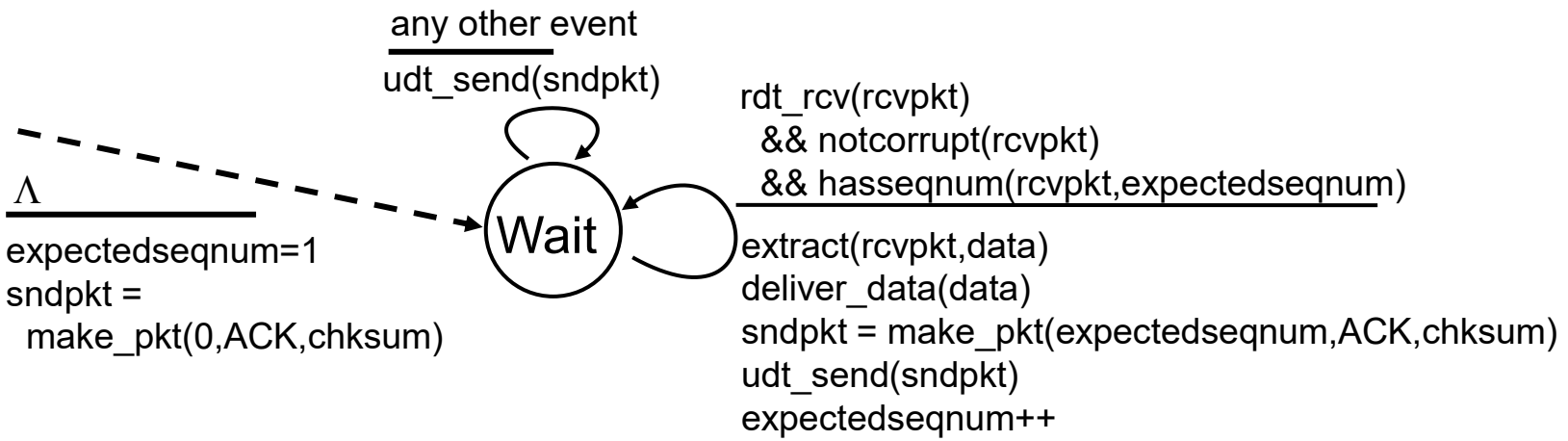
Fuori sequenza: ricevuto ma non riscontrato

Non ricevuto

Go-Back-N: FSM esteso del mittente



Go-Back-N: FSM esteso del rivente

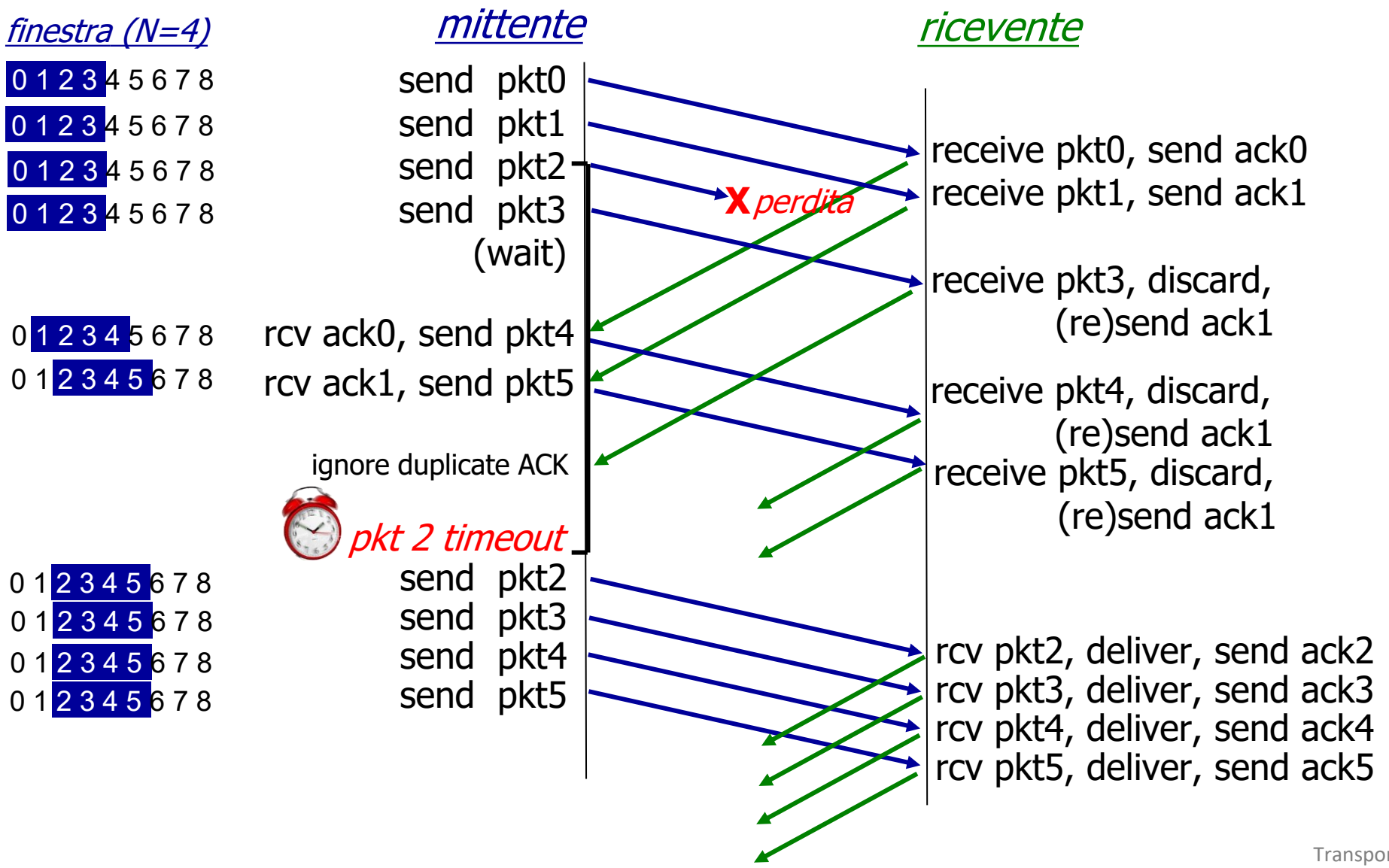


Solo ACK: invia sempre un ACK per pacchetti ricevuti correttamente col più alto numero di sequenza *in-ordine*

- può generare ACK duplicati
- deve solo ricordare **`expectedseqnum`**

- pacchetti fuori ordine:
  - scarta (non salva nel buffer): *nessun buffering lato ricevente!*
  - Re-invia ACK per il pacchetto con il più alto numero di sequenza in ordine

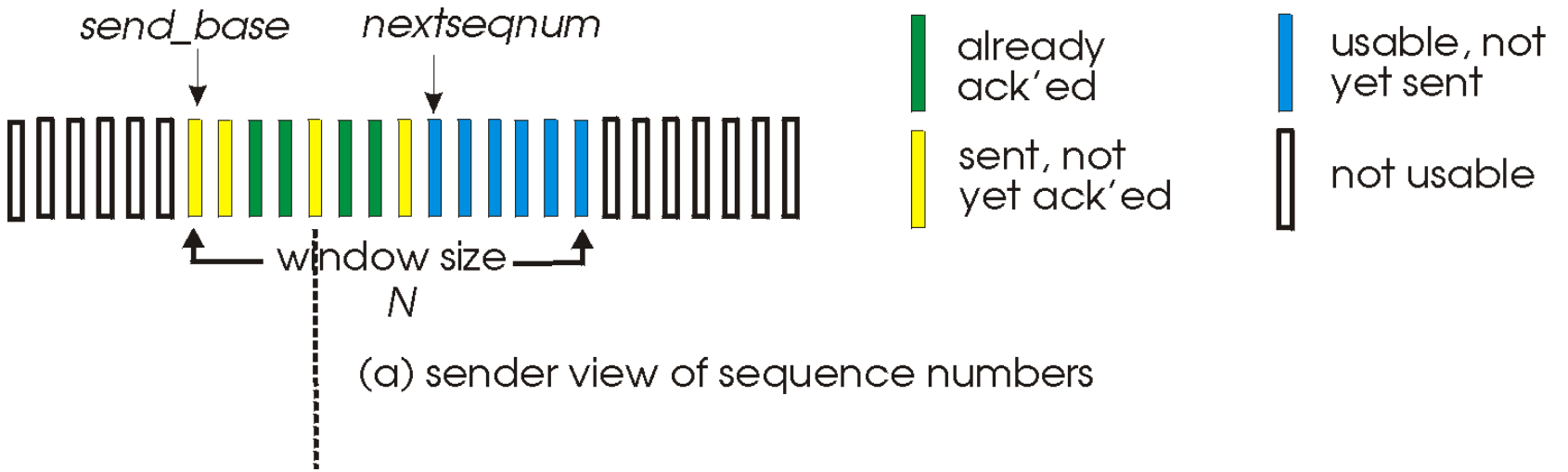
Go-Back-N in azione



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



Selective repeat: mittente e 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, rcvbase+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,rcvbase-1]

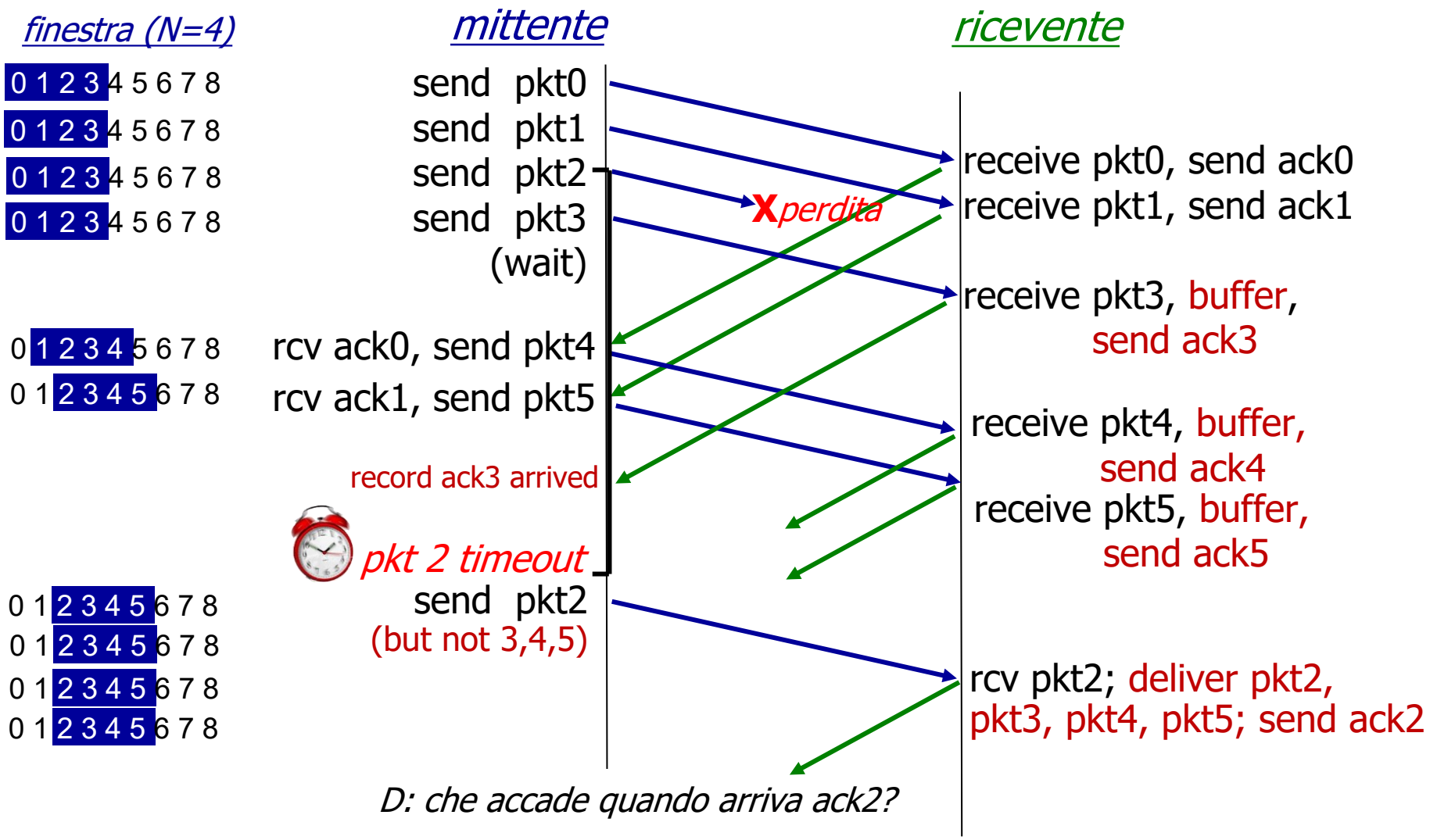
- ACK(*n*)

altrimenti:

- ignora

Il ricevente deve riscontrare pacchetti con numero di sequenza al di sotto di `rcvbase` (già consegnati all'applicazione), per far avanzare la finestra del mittente (rimasta indietro perché gli ACK non sono stati ricevuti)

Selective Repeat in azione

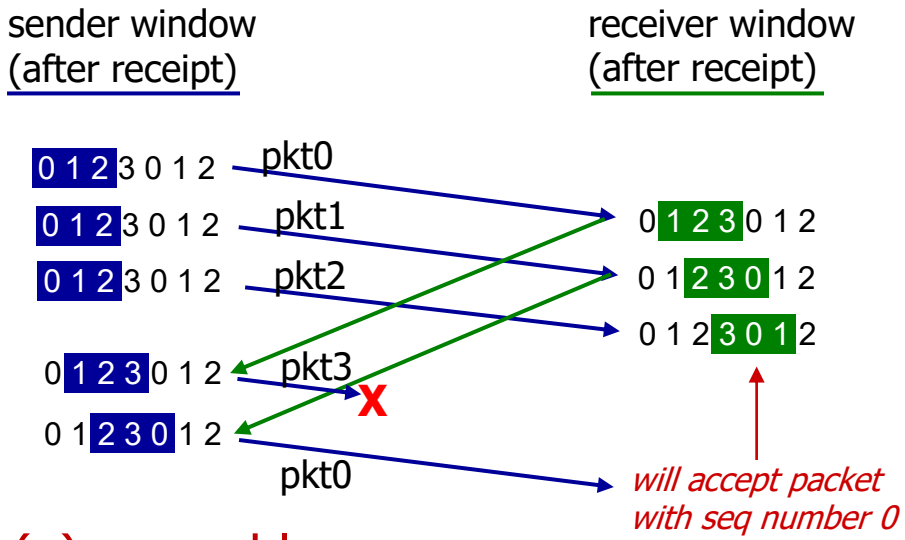


Selective repeat: dilemma!

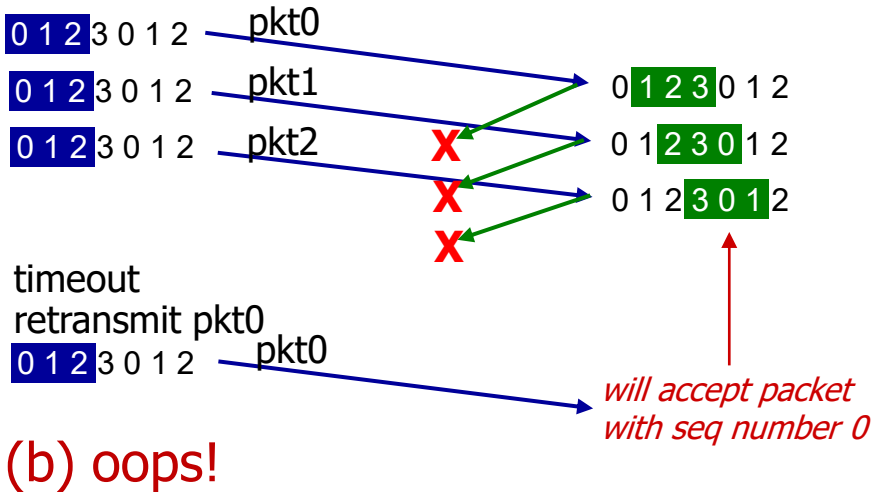
esempio:

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

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



(a) no problem



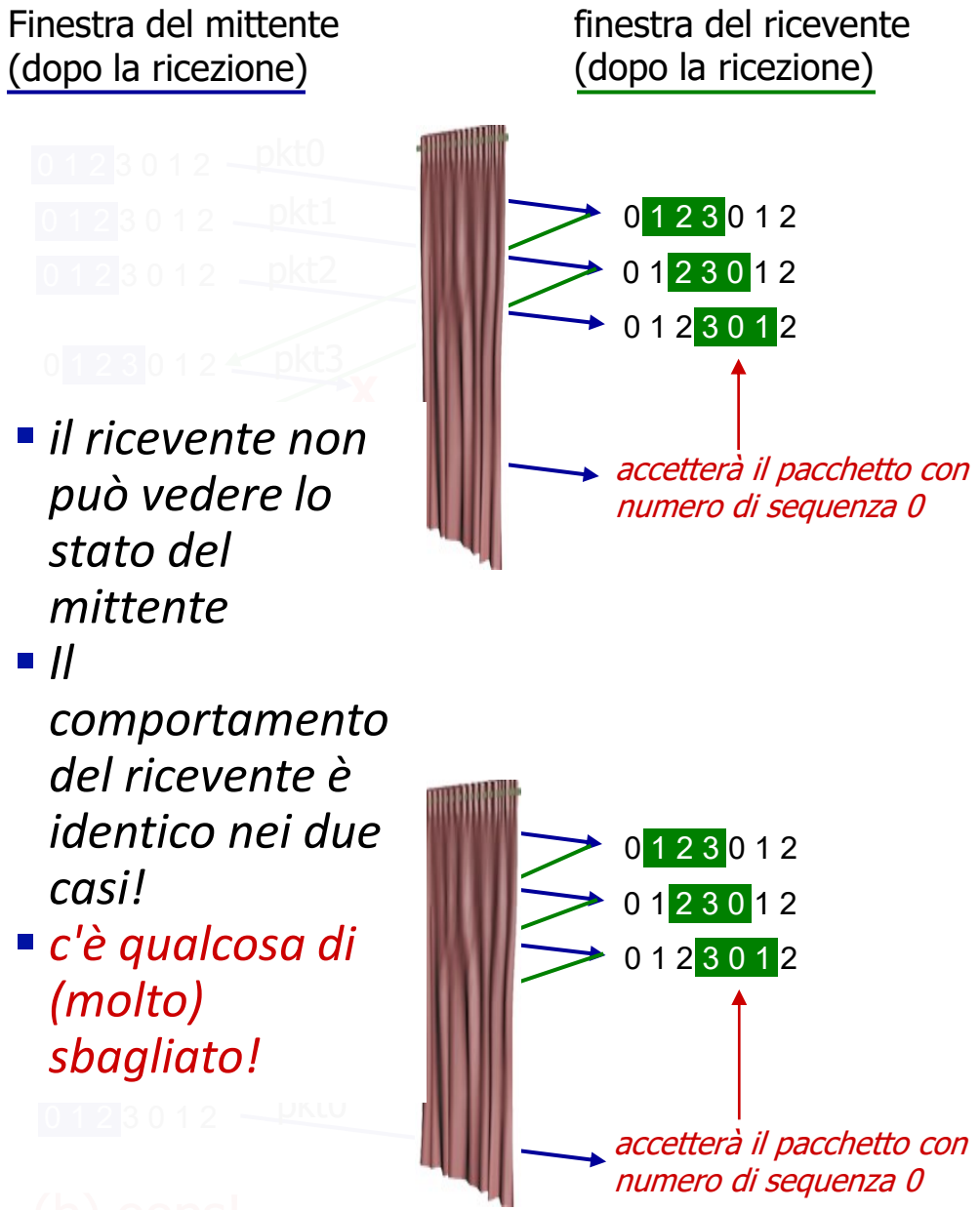
(b) oops!

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)?



- *il ricevente non può vedere lo stato del mittente*
- *Il comportamento del ricevente è identico nei due casi!*
- *c'è qualcosa di (molto) sbagliato!*

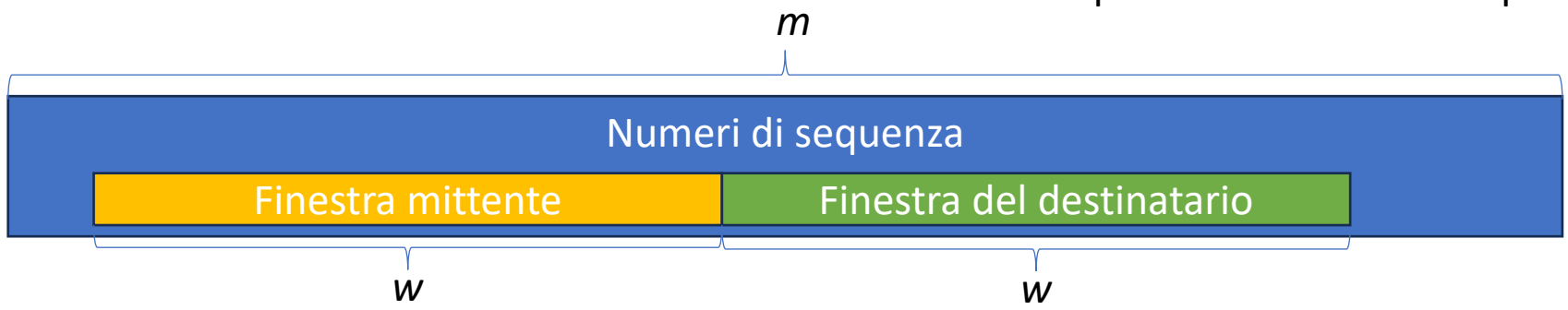
Relazione tra dimensione della finestra e dei numeri di sequenza (sia SR sia GBN)

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:



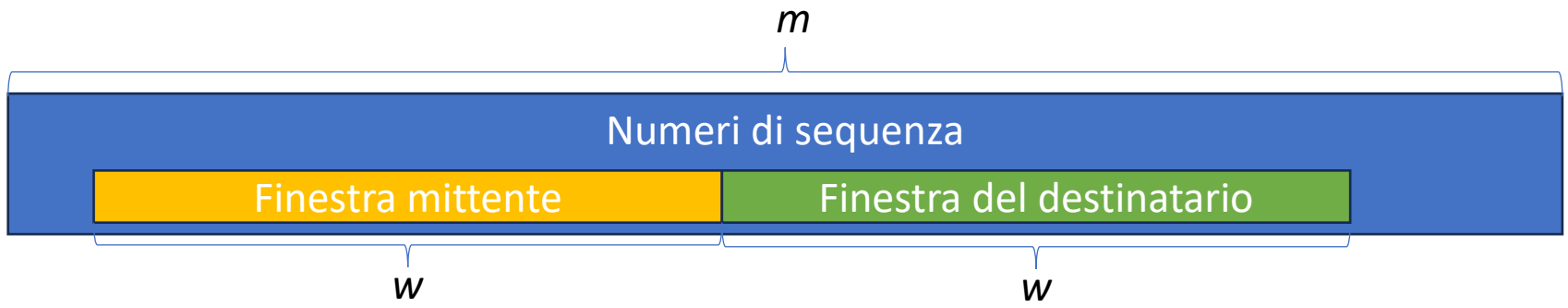
$2w \leq m$

cioè

$w \leq m / 2$

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)?



$2w \le m$
cioè
$w \le m / 2$

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