

Reti di Elaboratori

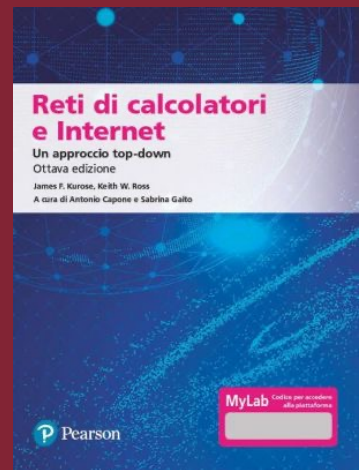
Livello di Trasporto: trasferimento affidabile dei dati



SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Checco

alessandro.checco@uniroma1.it



Capitolo 3

Livello di trasporto: sommario

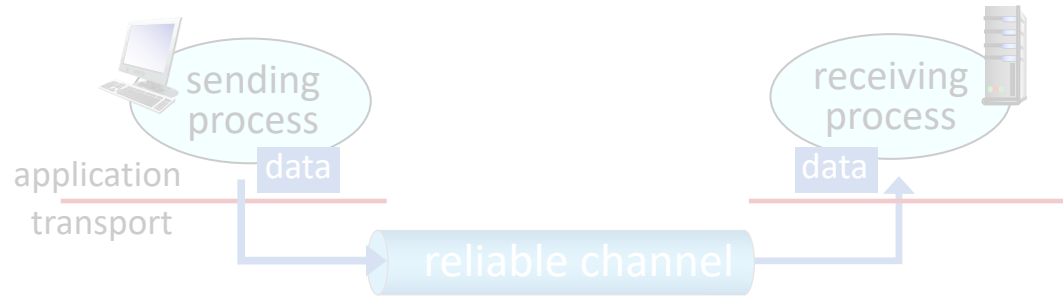
- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP
- **Principi di trasferimento affidabile dei dati**
- Trasporto orientato alla connessione: TCP
- Principi di controllo della congestione
- Controllo della congestione TCP
- Evoluzione della funzionalità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

Principi del trasferimento affidabile dei dati

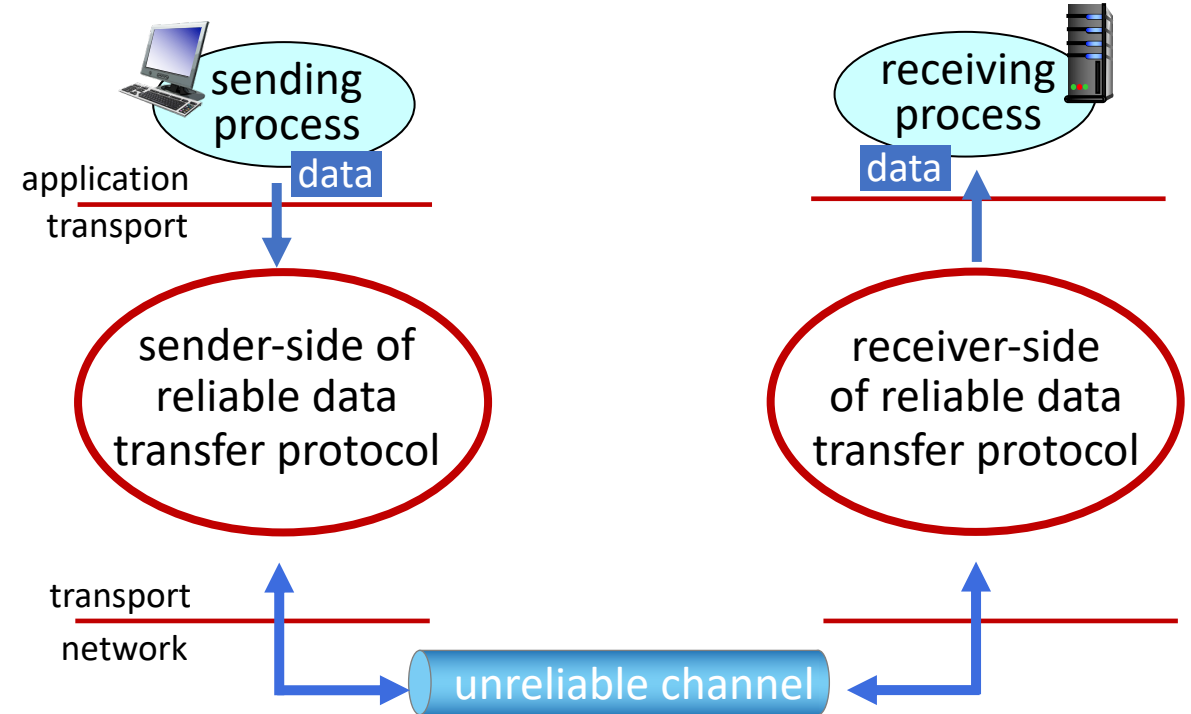
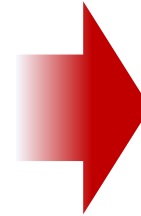


reliable service *abstraction*

Principi del trasferimento affidabile dei dati



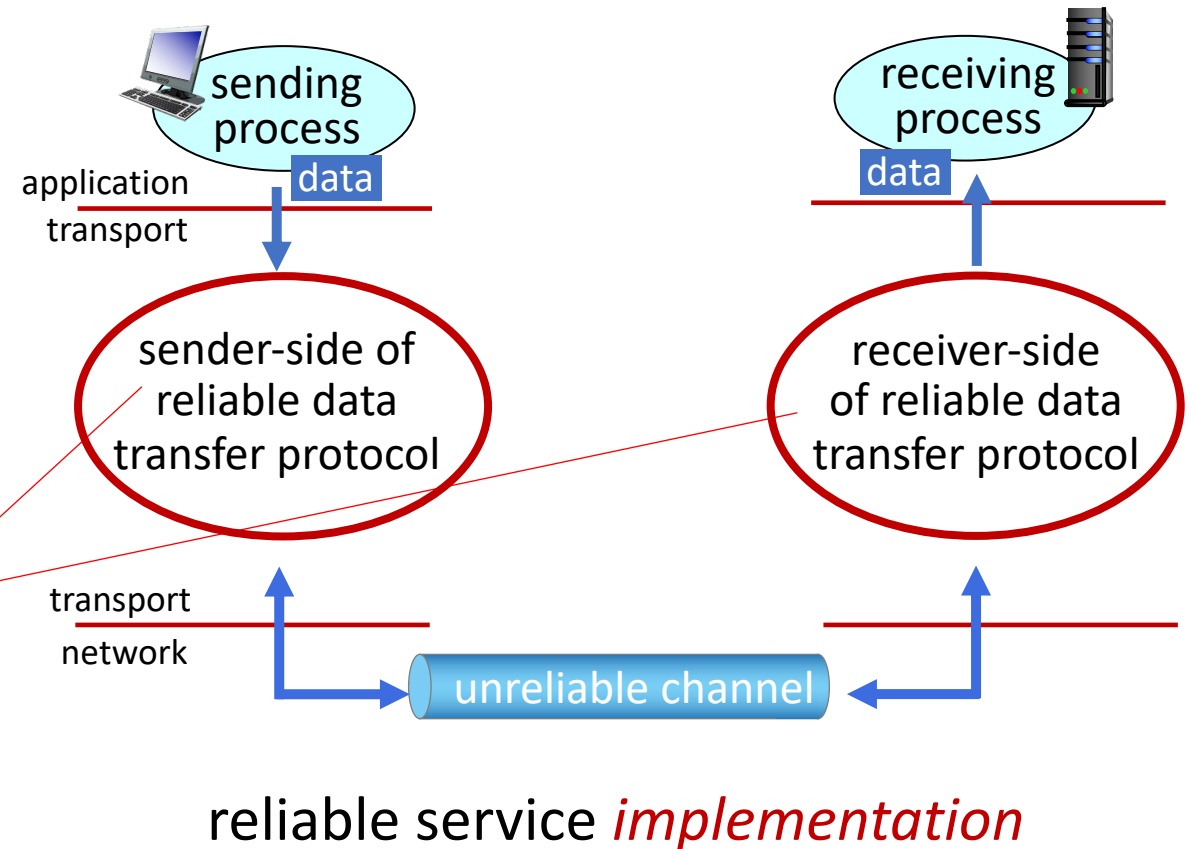
reliable service *abstraction*



reliable service *implementation*

Principi del trasferimento affidabile dei dati

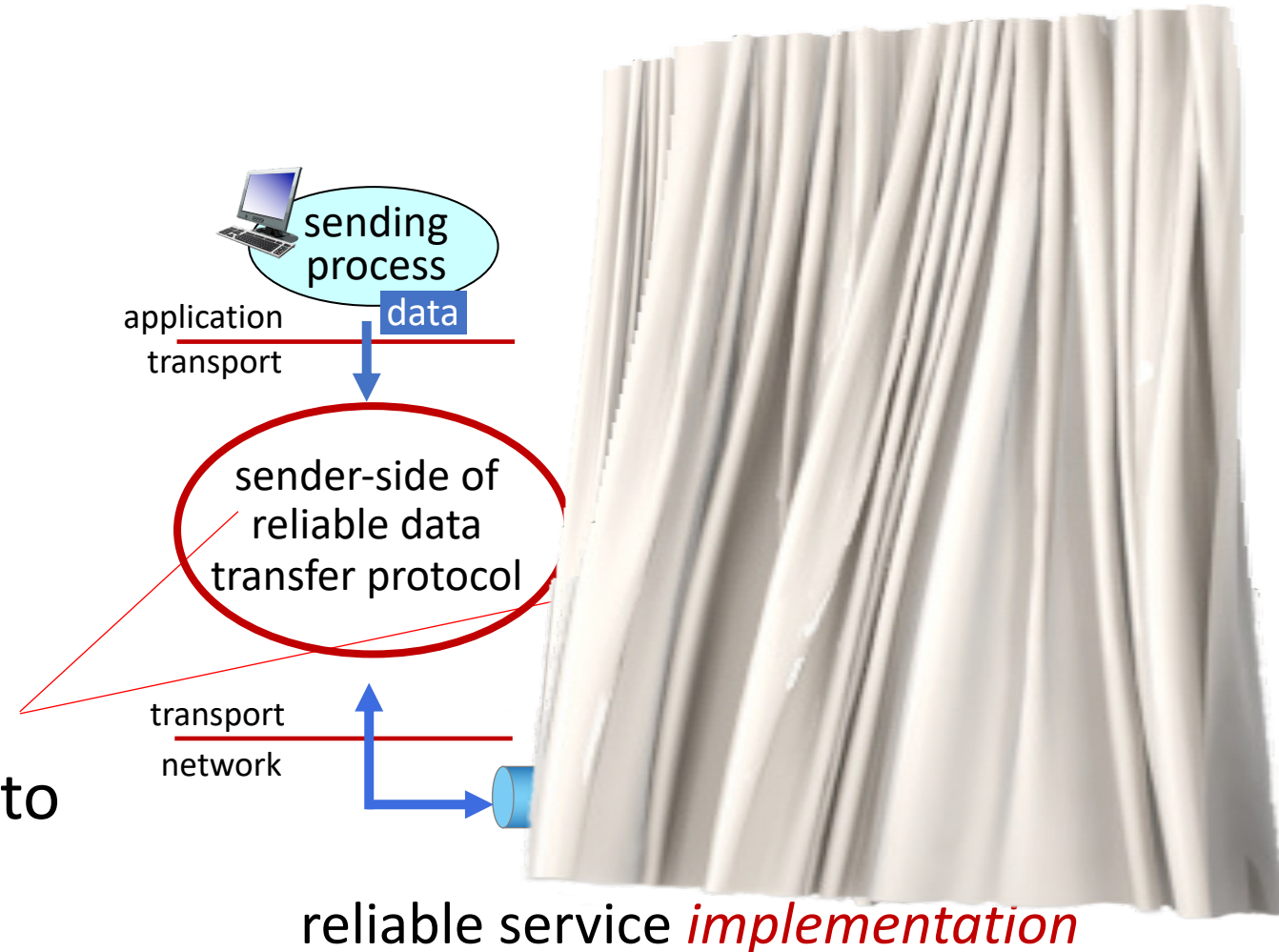
La complessità del protocollo di trasferimento dati affidabile dipenderà (fortemente) dalle caratteristiche del canale inaffidabile (perdita, corruzione, ordine di ricezione)



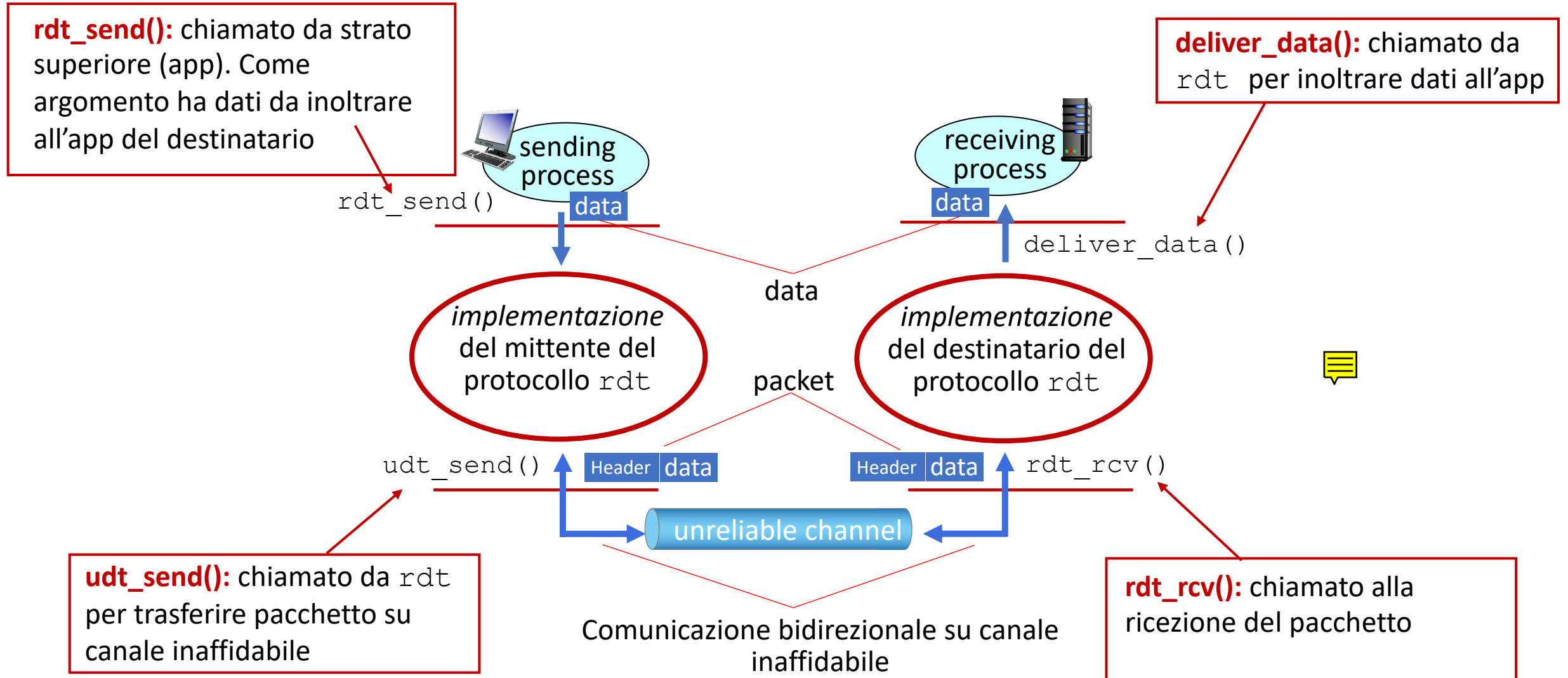
Principi del trasferimento affidabile dei dati

mittente e il destinatario *non* conoscono lo "stato" l'uno dell'altro (ad. es messaggio ricevuto?)

- a meno che non venga comunicato tramite messaggio



Protocollo di trasferimento dati affidabile (rdt): interfacce



Trasferimento affidabile dei dati: come iniziare

Obiettivo:

- sviluppare in modo un protocollo affidabile per il trasferimento dei dati (rdt)
- considerando trasferimento di dati unidirezionale
 - ma le informazioni di controllo viaggeranno in entrambe le direzioni!
- utilizzeremo macchine a stati finiti (FSM) per specificare mittente, destinatario



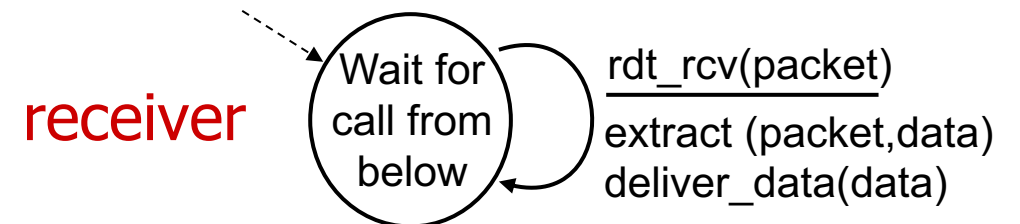
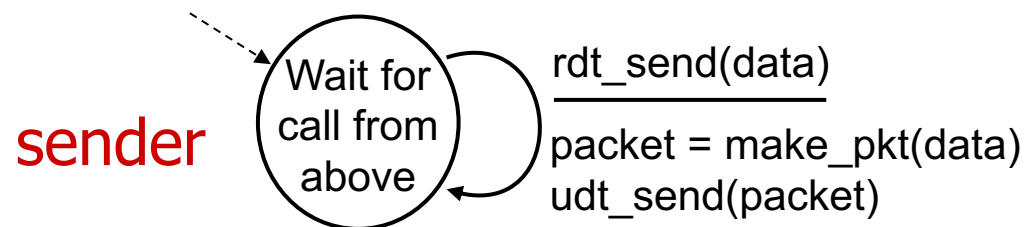
Lampadina e interruttore

- Stati?
- Evento che causa il cambio di stato?
- Azioni intraprese al cambio di stato?



rdt1.0: trasferimento affidabile su un canale affidabile

- canale sottostante perfettamente affidabile
 - nessun errore di bit
 - nessuna perdita di pacchetti
- *una FSM separata* per mittente e destinatario:
 - mittente invia i dati nel canale sottostante
 - il ricevitore legge i dati dal canale sottostante



rdt2.0: canale con bit errors

- il canale sottostante può capovolgere i bit nel pacchetto
 - checksum (ad es., checksum Internet) per rilevare errori di bit
- come recuperare dagli errori?

In che modo gli esseri umani recuperano da "errori" durante la conversazione?

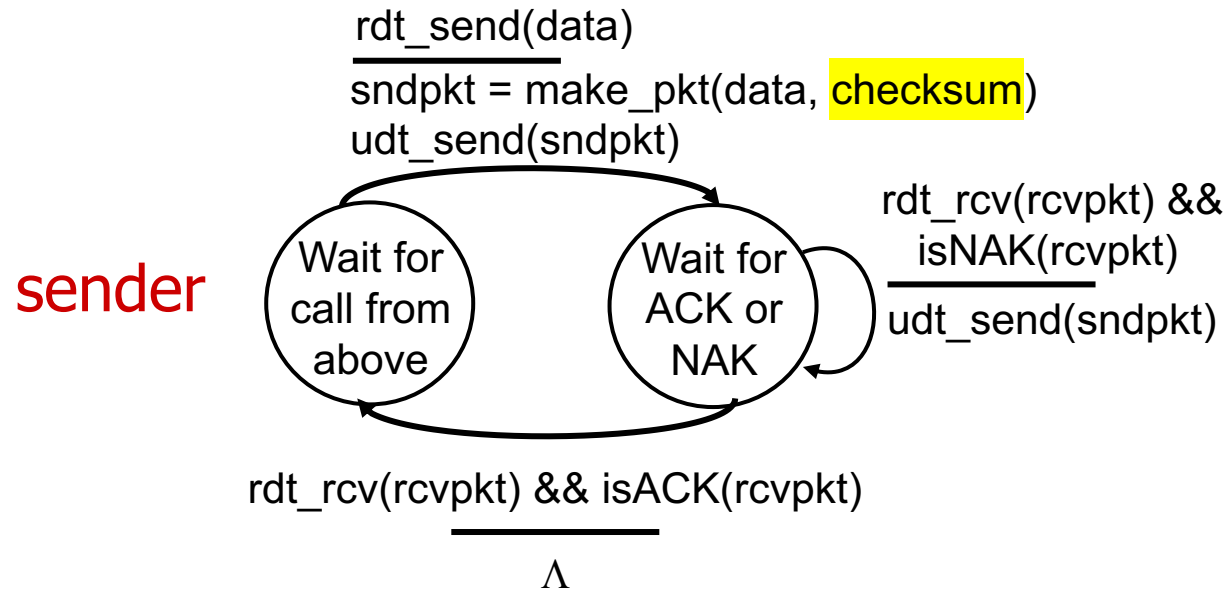
rdt2.0: canale con bit errors

- il canale sottostante può invertire dei bit nel pacchetto
 - checksum per rilevare errori di bit
- come recuperare dagli errori?
 - *acknowledgements (ACK)*: il destinatario dice esplicitamente al mittente che il pacchetto è stato ricevuto senza problemi
 - *negative acknowledgement (NAK)*: il destinatario dice esplicitamente al mittente che il pacchetto conteneva degli errori
 - mittente *ritrasmette* il pacchetto al ricevimento di NAK

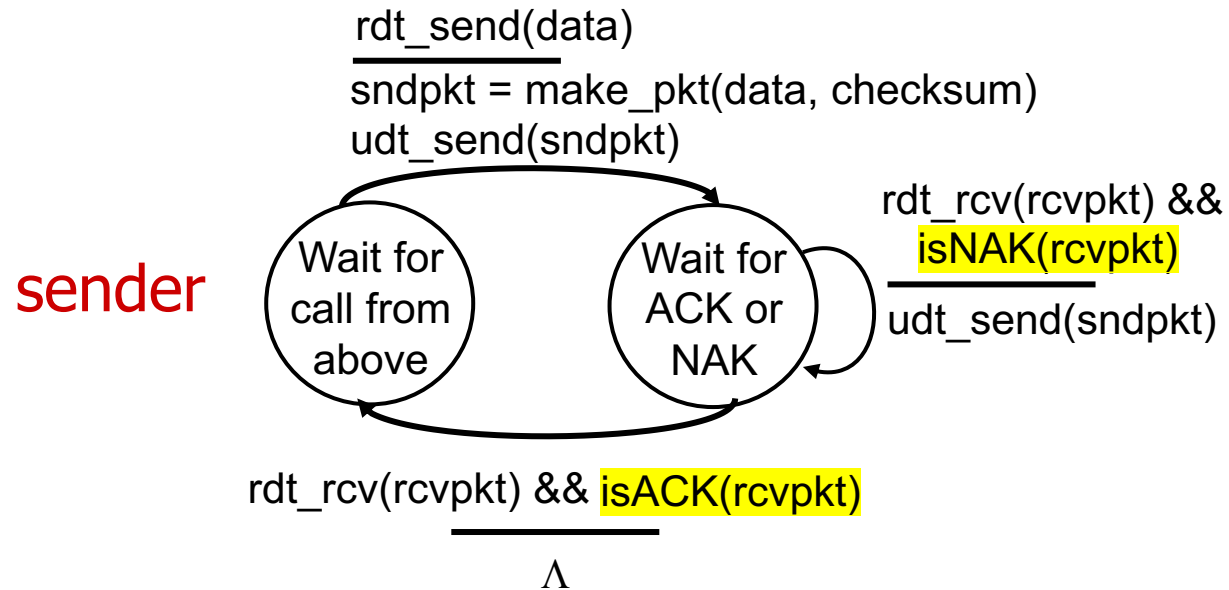
stop and wait

il mittente invia un pacchetto e si ferma attendendo la risposta del destinatario

rdt2.0: specifica delle FSM



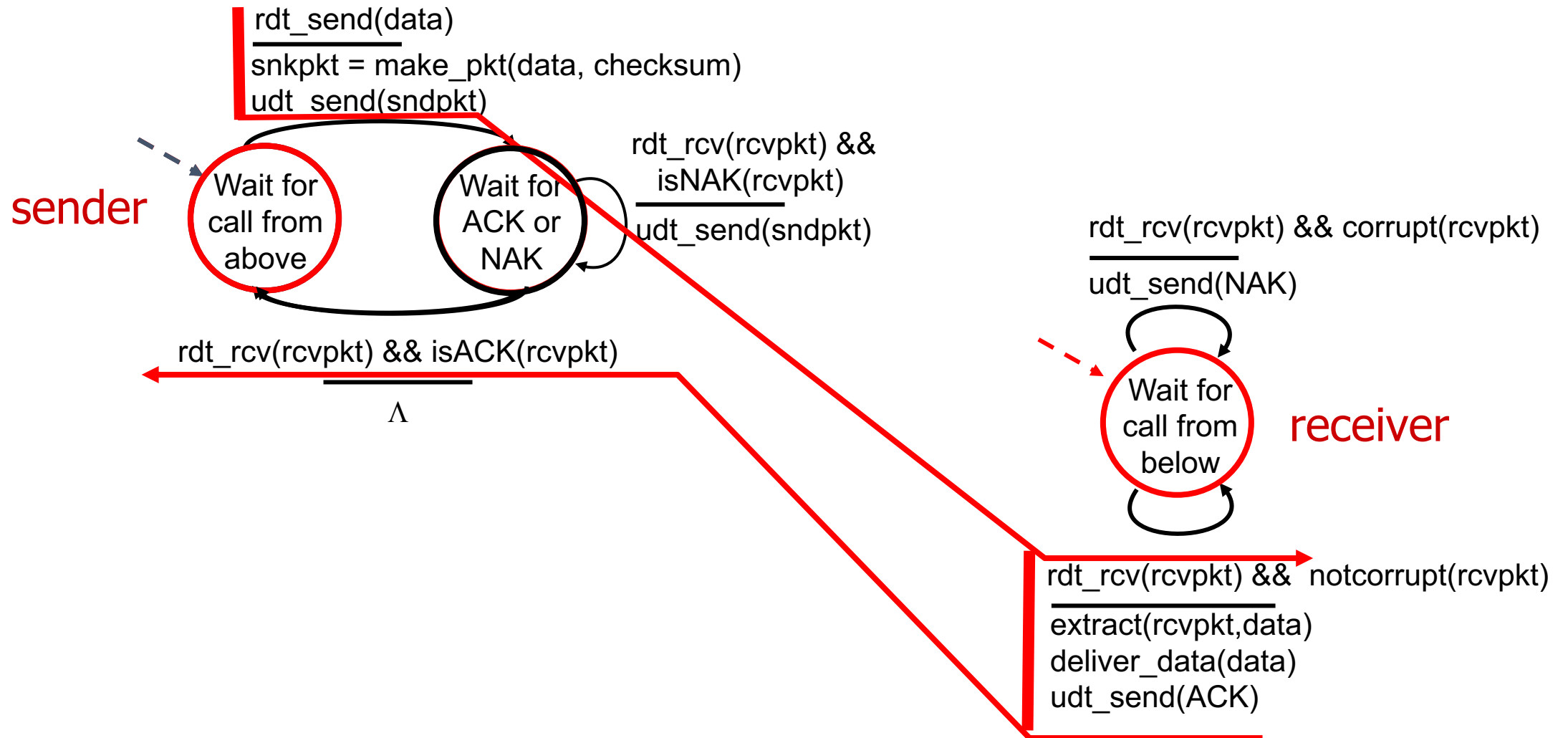
rdt2.0: specifica delle FSM



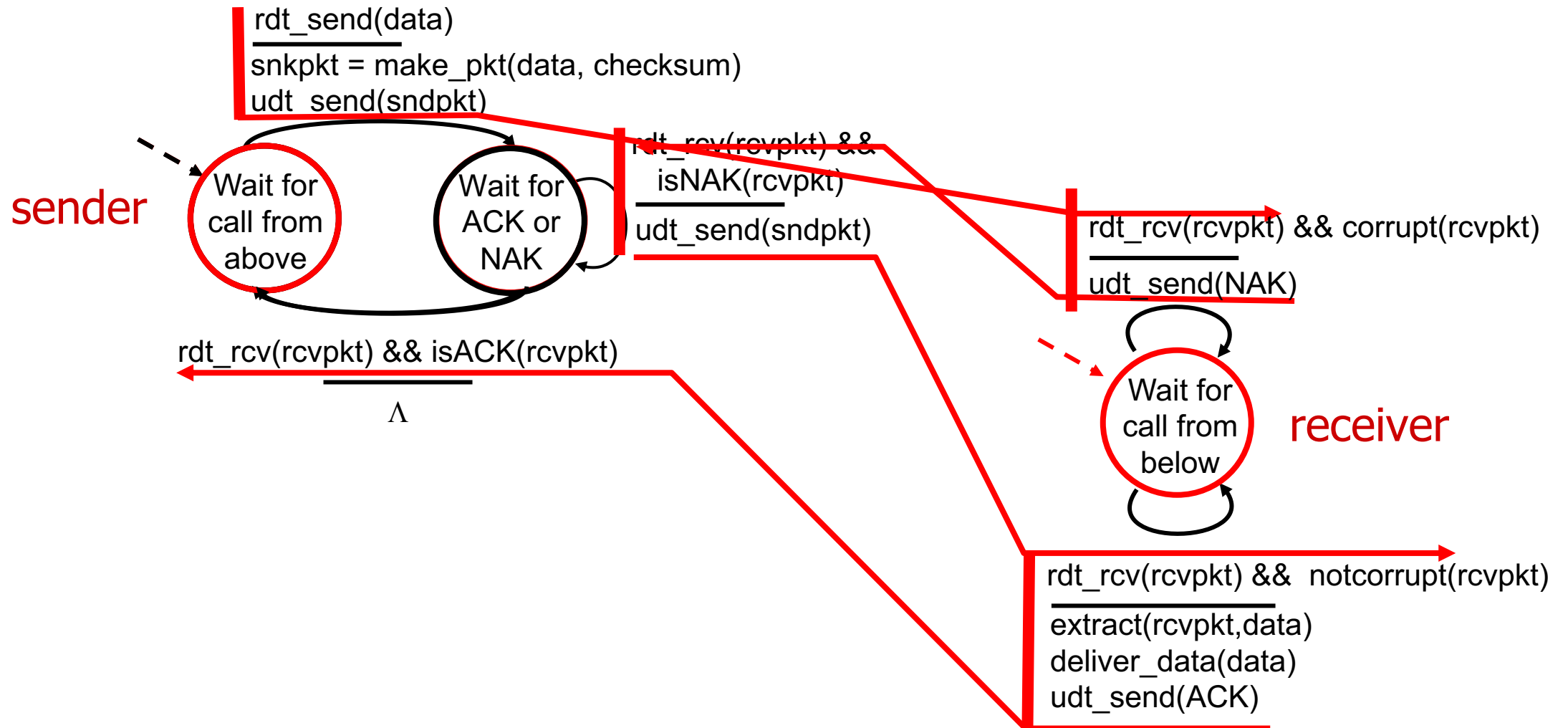
- Note:** lo "stato" del destinatario (messaggio ricevuto?) non è noto al mittente a meno che non sia comunicato in qualche modo dal destinatario
- per questo serve un protocollo!



rdt2.0: operazione senza errori



rdt2.0: scenario corruzione pacchetto



rdt2.0 ha un difetto fatale!

cosa succede se ACK/NAK è corrotto?

- il mittente non sa cosa è successo al destinatario!
- non basta ritrasmettere: potenziale pacchetto duplicato inoltrato all'app

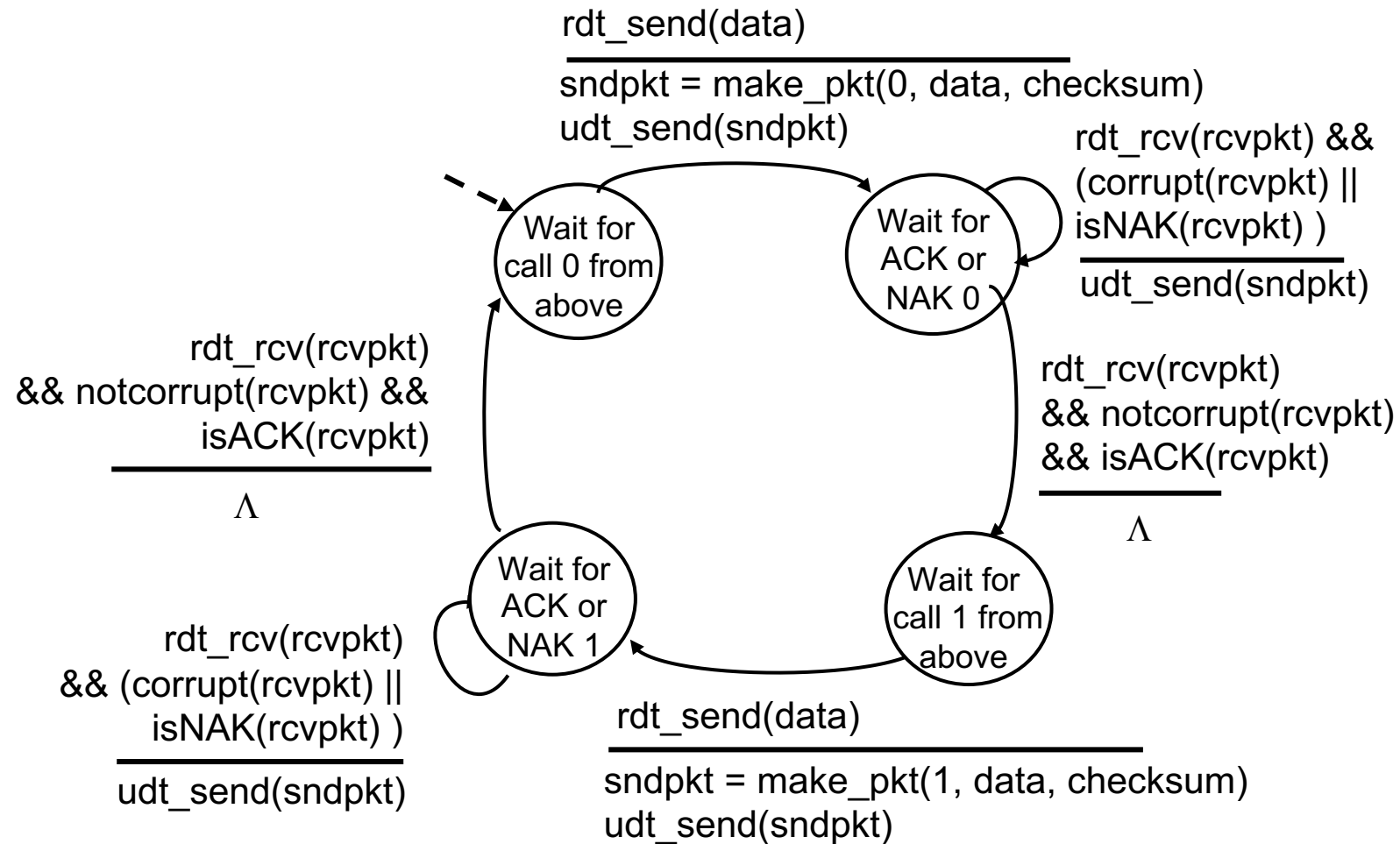
gestione dei duplicati:

- il mittente ritrasmette il pacchetto corrente se ACK/NAK è corrotto
- il mittente aggiunge un *numero di sequenza* a ciascun pkt
- il ricevitore scarta (non consegna all'app) eventuali pacchetti duplicati

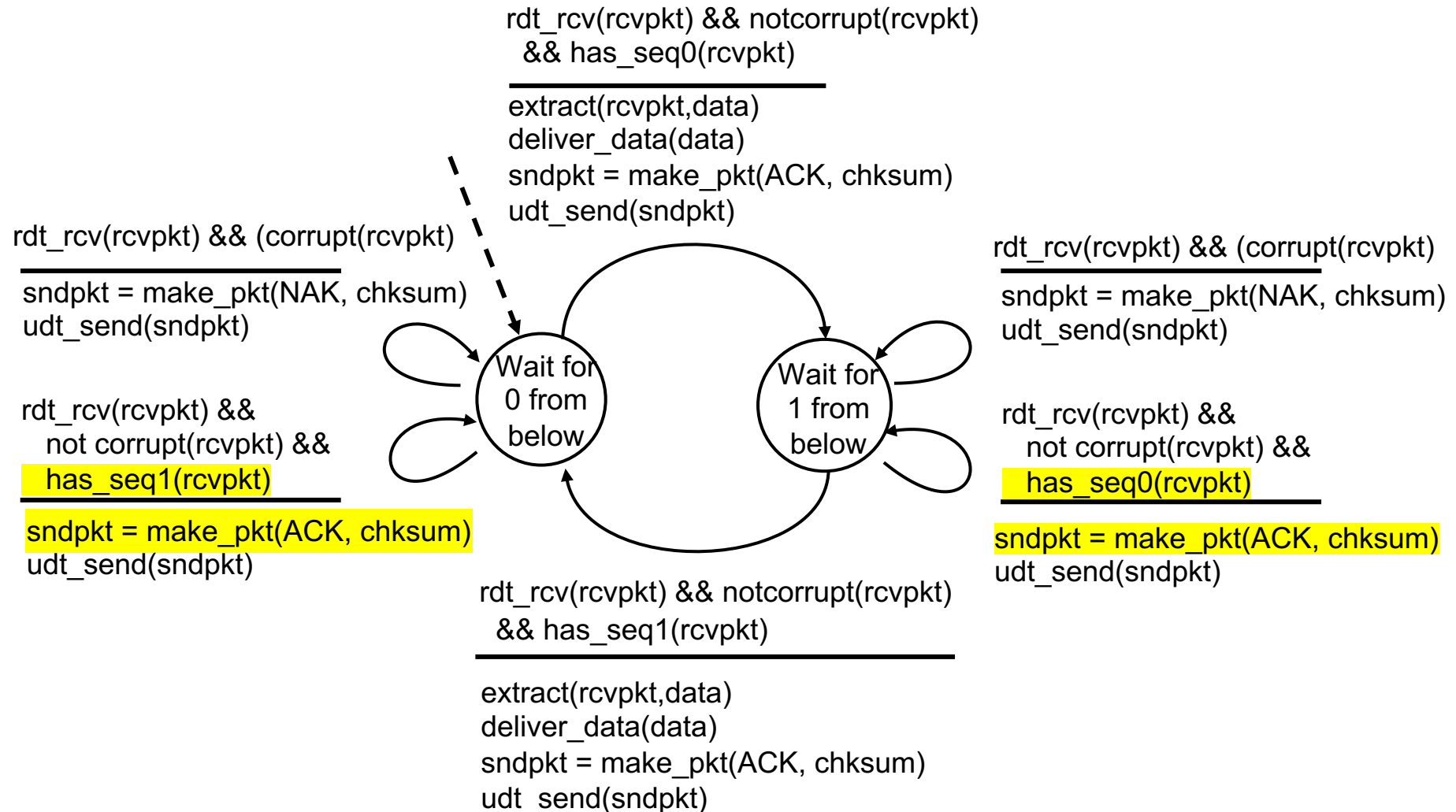
stop and wait

il mittente invia un pacchetto e attende la risposta del destinatario

rdt2.1: gestione di ACK/NAKs corrotti lato mittente



rdt2.1: gestione di ACK/NAKs corrotti lato destinatario



rdt2.1: osservazioni

mittente:

- aggiunge # sequenza al pacchetto
- Basta usare # seq {0,1}. Perché?
- È necessario controllare che ACK/NAK non siano corrotti
- Il doppio degli stati di rdt2.0
 - lo stato deve sapere se il pacchetto atteso ha # seq 0 o 1

destinatario:

- deve controllare che il pacchetto ricevuto non sia un duplicato
 - lo stato deve tener traccia del numero di sequenza atteso
- Il destinatario **non sa** se l'ultimo ACK/NACK inviato è stato ricevuto correttamente dal mittente

Numeri di sequenza e ACK nello stop and wait

- I numeri di sequenza 0 e 1 sono sufficienti per il protocollo stop and wait
- Convenzione: il numero di riscontro (ack) indica sempre il numero di sequenza del prossimo pacchetto atteso dal destinatario
 - Se il destinatario ha ricevuto correttamente il pacchetto 0 invia un riscontro con valore 1 (che significa che il prossimo pacchetto atteso ha numero di sequenza 1)

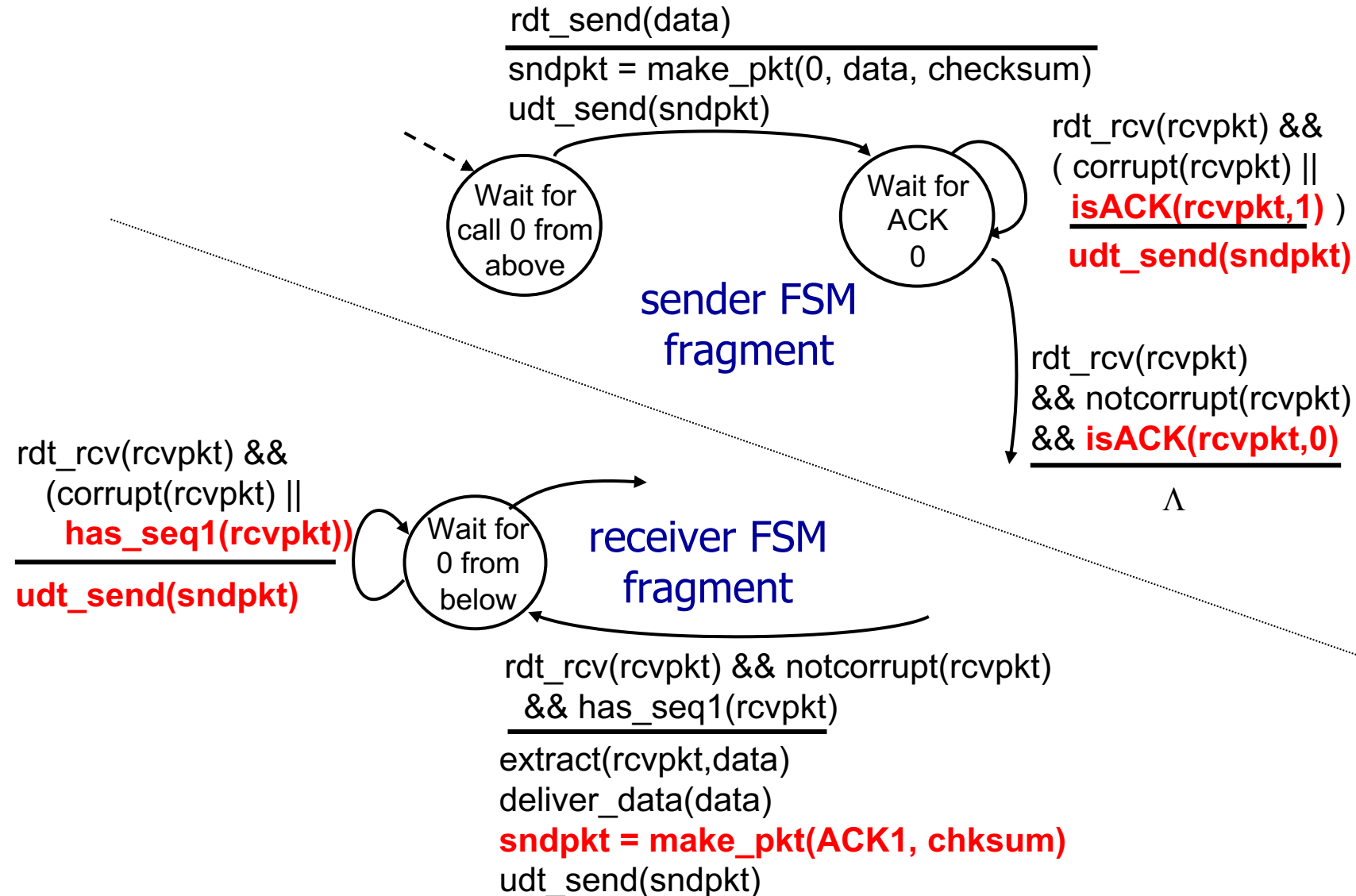
Nel meccanismo stop and wait il numero di riscontro indica, in aritmetica modulo 2, il numero di sequenza del prossimo pacchetto atteso dal destinatario

rdt2.2: un protocollo privo di NAK

- stessa funzionalità di rdt2.1, utilizzando solo ACK
- invece di NAK, il ricevitore invia ACK per l'ultimo pacchetto ricevuto OK
 - ricevitore deve includere in *modo esplicito* # seq del pacchetto per il quale si sta mandando un ACK
- ACK duplicato al mittente comporta la stessa azione di NAK: *ritrasmettere il pacchetto corrente*

Come vedremo, TCP utilizza questo approccio per essere privo di NAK

rdt2.2: parte della FSM



Cosa manca?

- Questo modello considera solo bit error come possibile problema
- Il pacchetto può essere perso (molto comune)!

rdt3.0: canali con errori e perdite

Nuova assunzione del canale: il canale sottostante può *perdere* pacchetti (dati o ACK)

- checksum, numeri di sequenza, ACK, ritrasmissioni possono aiutare... ma non completamente

D: In che modo gli *esseri umani* gestiscono il caso in cui una parte di conversazione viene persa?

rdt3.0: canali con errori e perdite

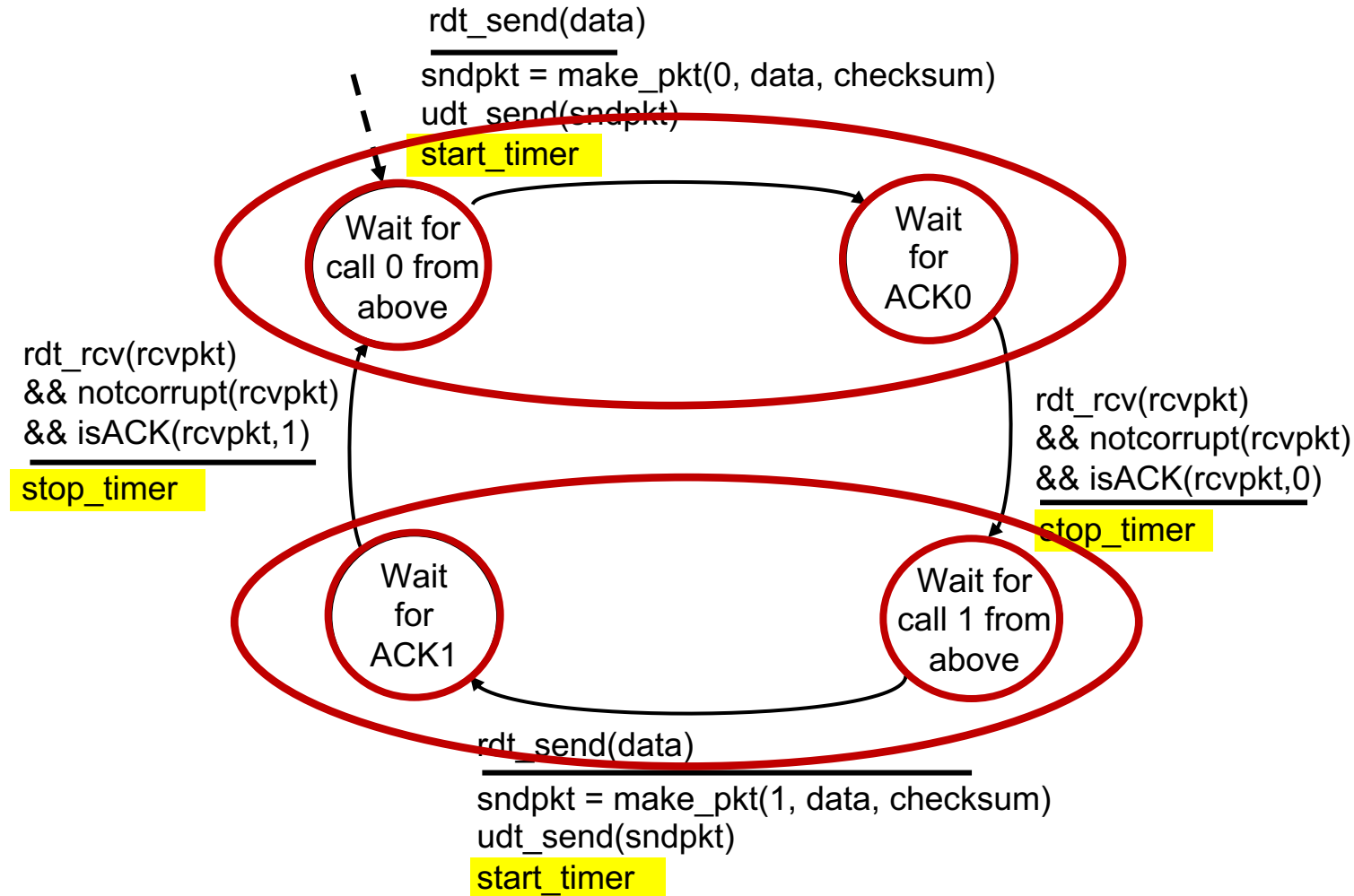
Approccio: il mittente attende un periodo di tempo "ragionevole" per l'ACK

- ritrasmette se nessun ACK ricevuto in questo tempo
- se pkt (o ACK) arriva dopo (quindi non era perso):
 - la ritrasmissione sarà duplicata, ma #seq già gestisce questo problema!
 - destinatario deve specificare il numero di sequenza del pacchetto per il quale sta mandando un ACK
- uso di un timeout per interrompere dopo un periodo di tempo "ragionevole"

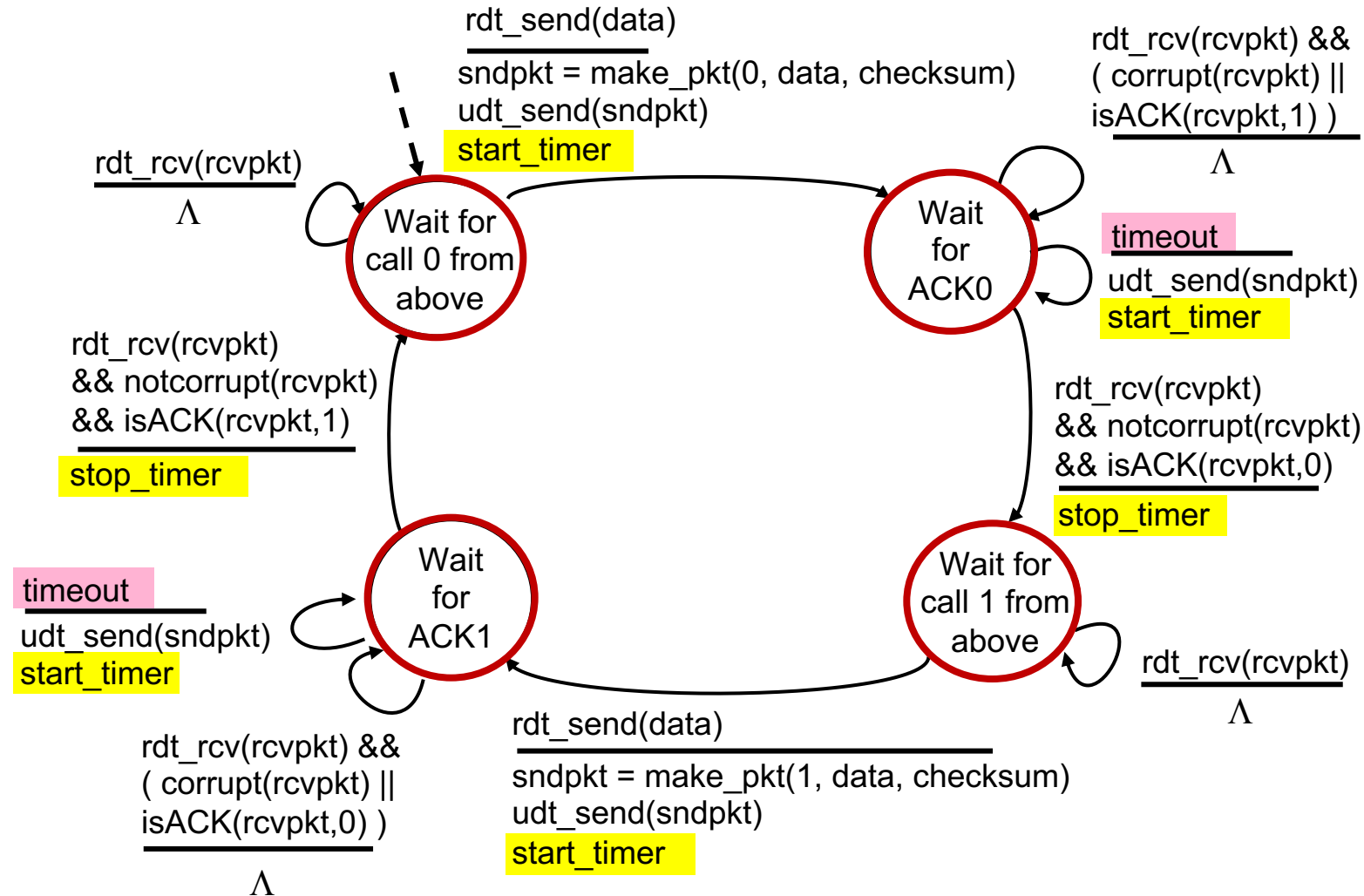


timeout

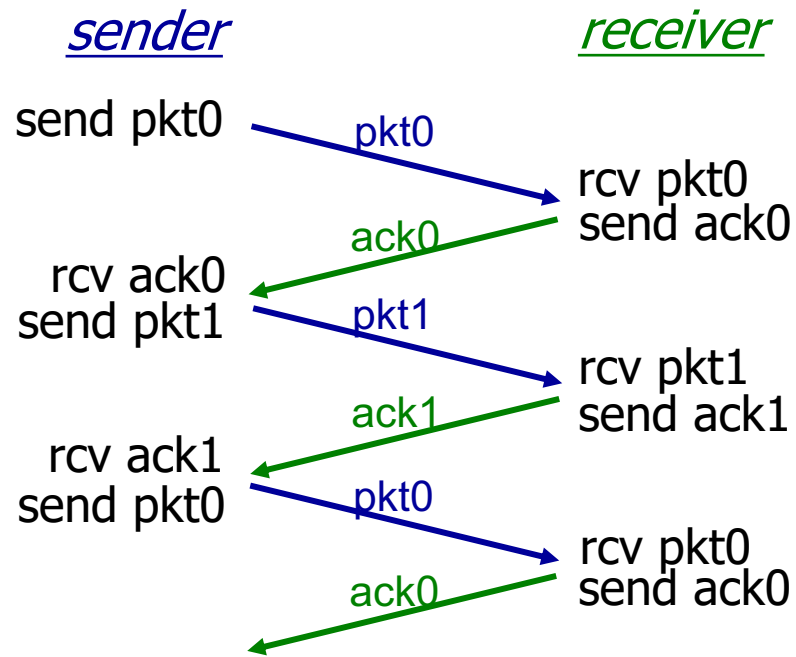
rdt3.0 FSM del mittente



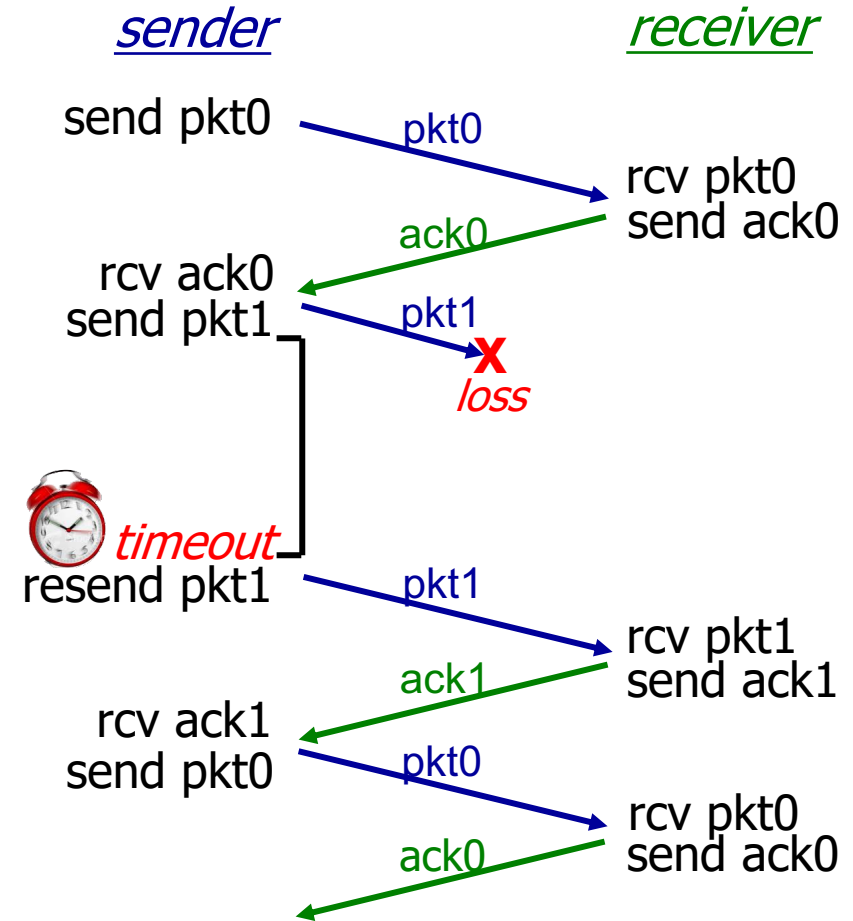
rdt3.0 FSM del mittente



rdt3.0 in azione

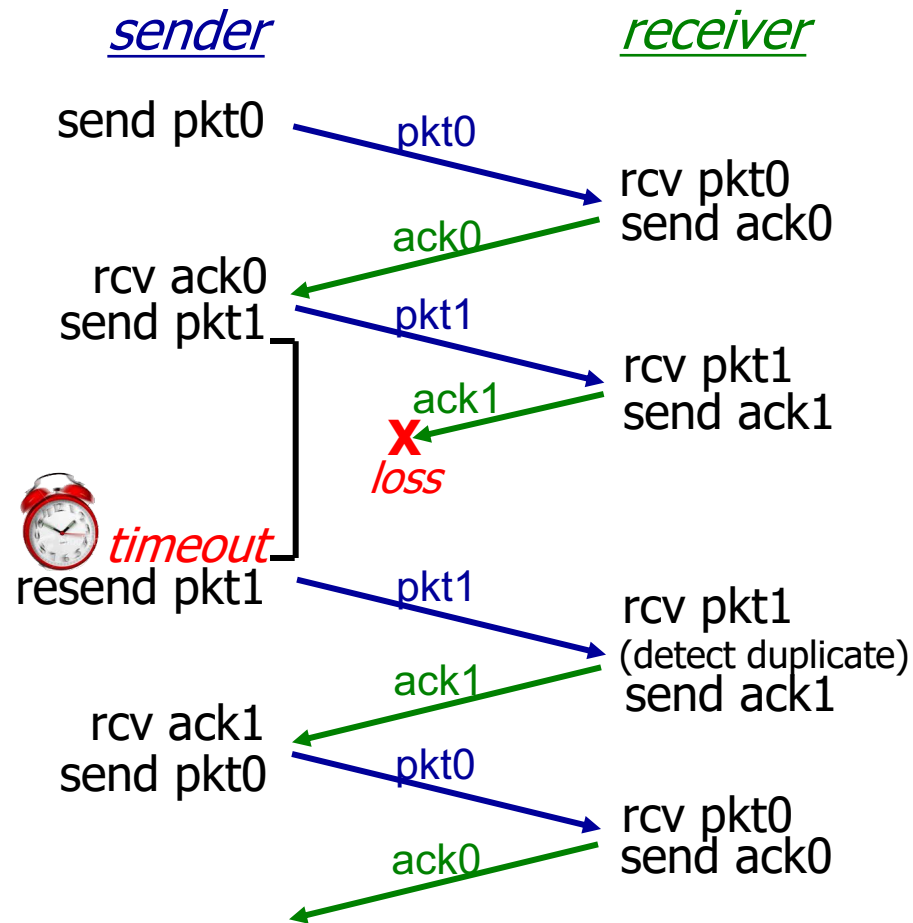


(a) no loss

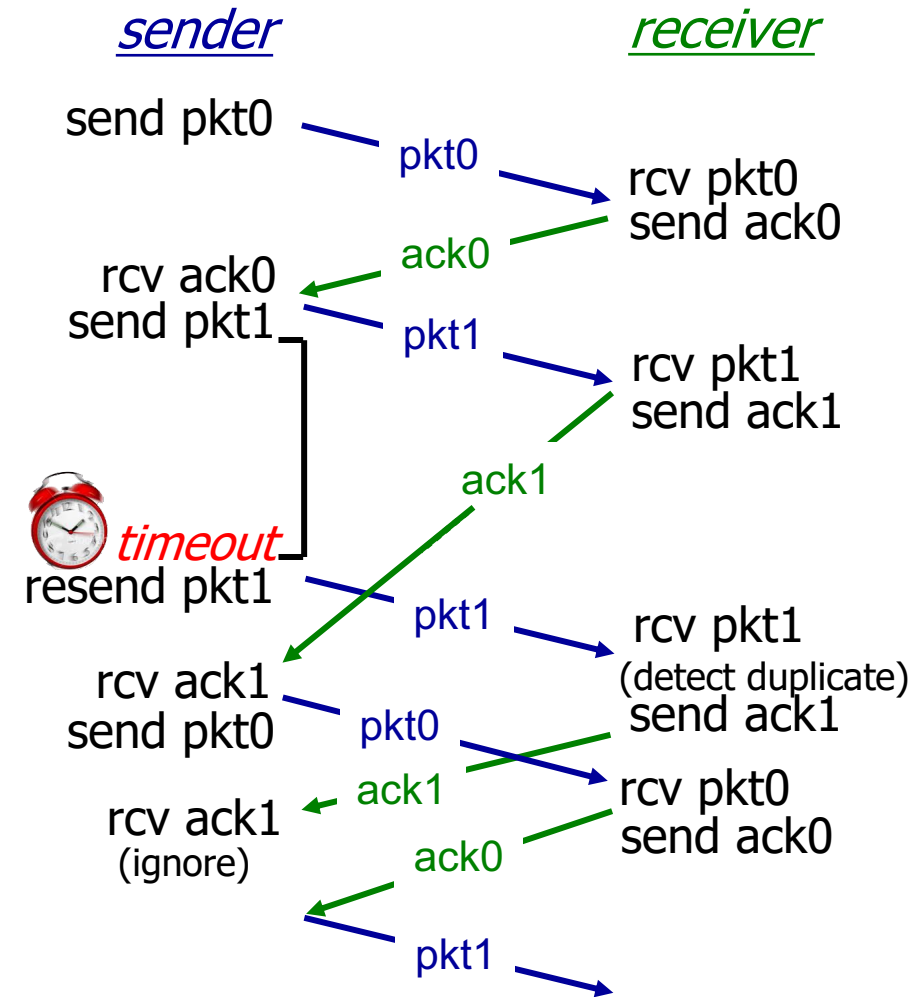


(b) packet loss

rdt3.0 in azione



(c) ACK loss



(d) premature timeout/delayed ACK

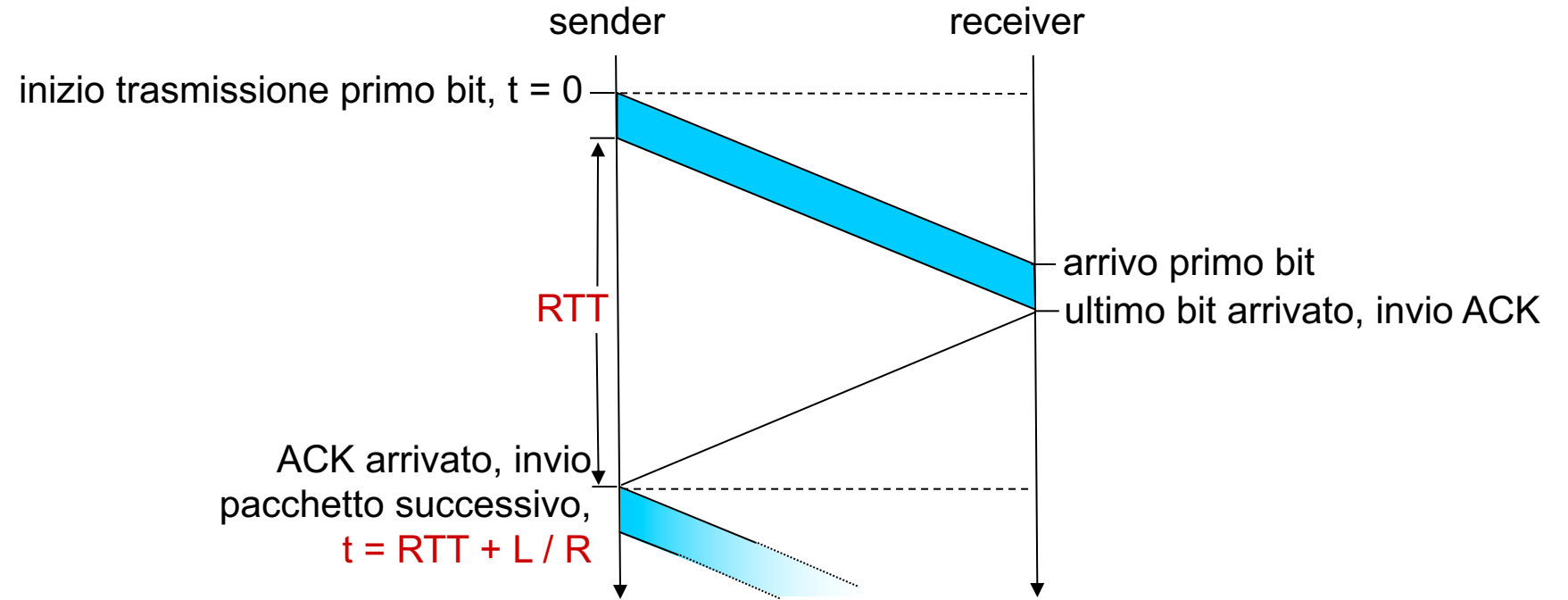


Prestazioni di rdt3.0 (stop-and-wait)

- $U_{mittente}$: *utilizzo* – frazione di tempo in cui il mittente è impegnato nell'invio
- esempio: collegamento da 1 Gbps, 15 ms di ritardo di propagazione, pacchetto da 8000 bit
 - tempo per trasmettere il pacchetto nel canale:

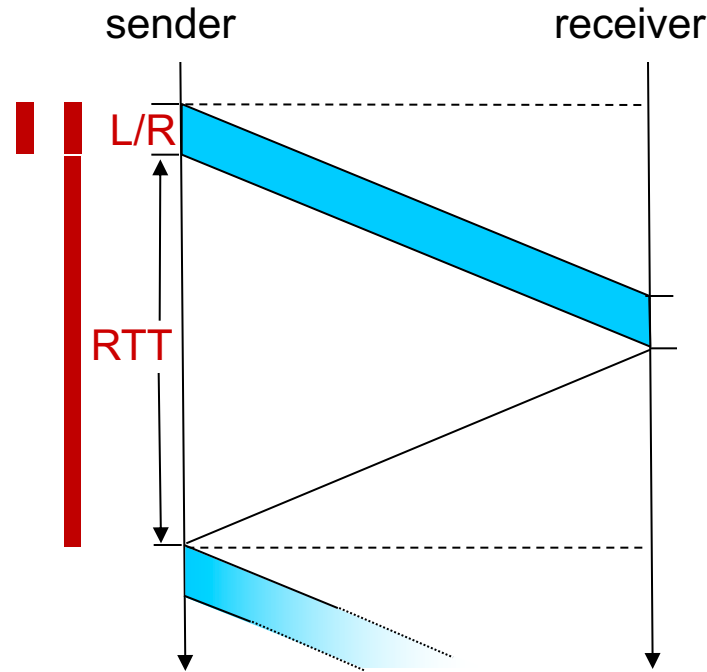
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bit}}{10^9 \text{ bit/s}} = 8 \mu s$$

rdt3.0: stop-and-wait



rdt3.0: stop-and-wait

$$\begin{aligned}U_{\text{sender}} &= \frac{L / R}{RTT + L / R} \\&= \frac{.008}{30.008} \\&= 0.00027\end{aligned}$$

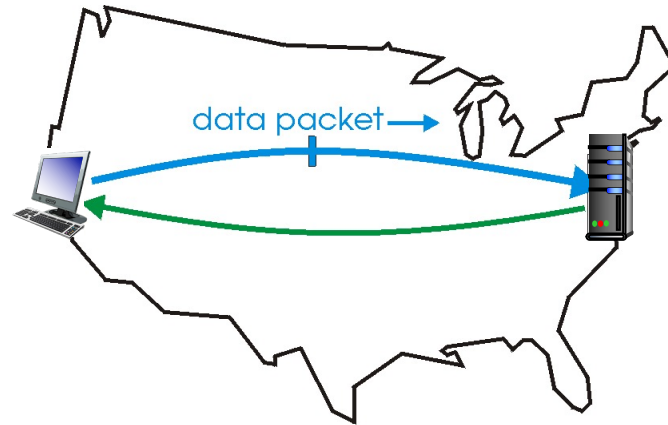


- il protocollo rdt3.0 ha prestazioni infime!
- Il protocollo limita le prestazioni dell'infrastruttura sottostante (canale)

rdt3.0: protocolli con pipeline

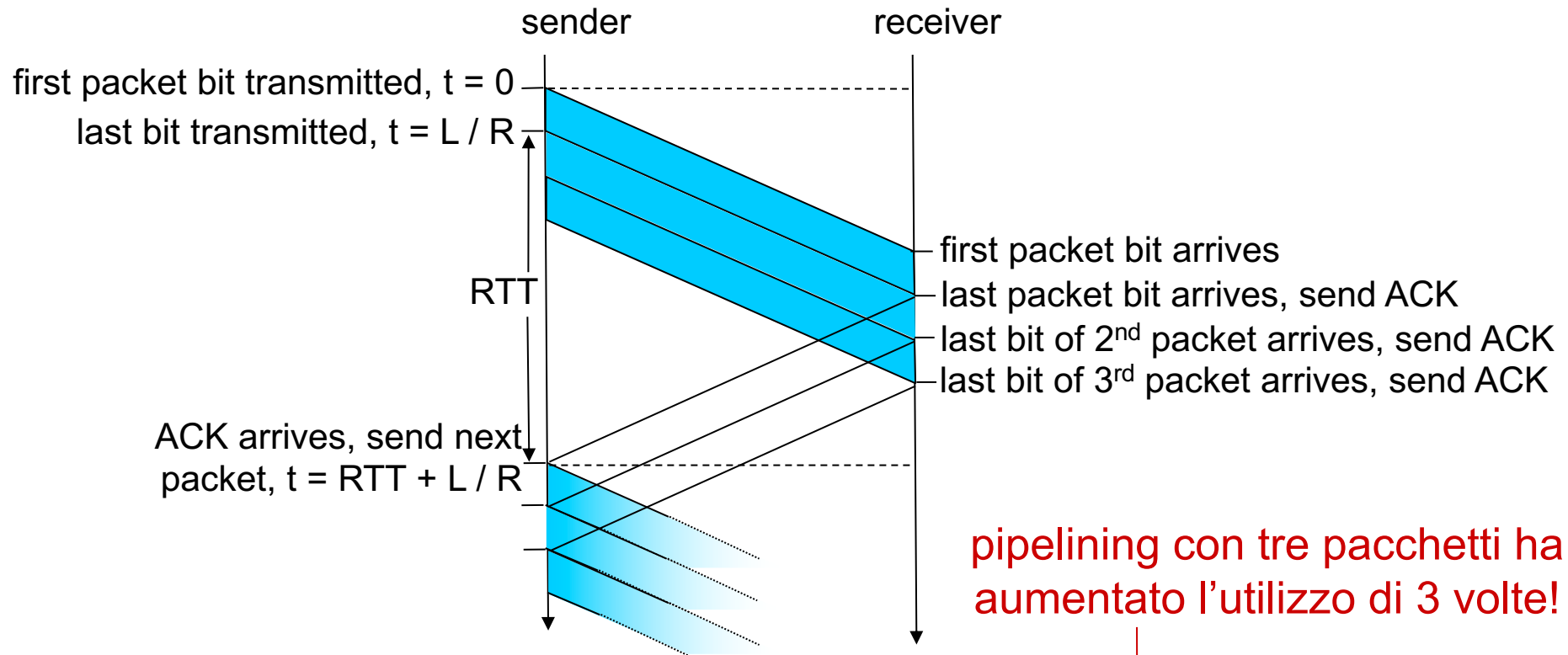
pipelining: il mittente consente molteplici pacchetti "in volo", inviati senza aver ricevuto un ACK per i precedenti

- l'intervallo di numeri di sequenza deve essere aumentato
- buffering al mittente e al destinatario



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

Pipelining: aumento dell'utilizzo



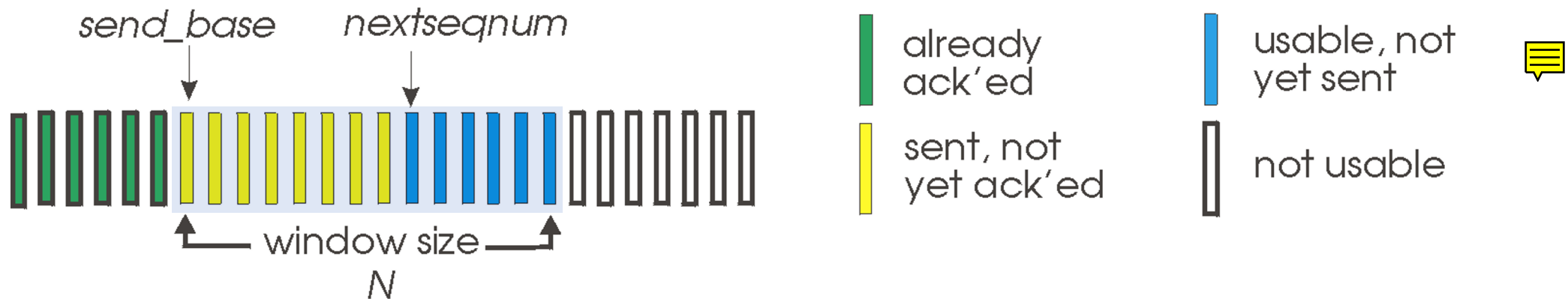
pipelining con tre pacchetti ha aumentato l'utilizzo di 3 volte!

$$U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$



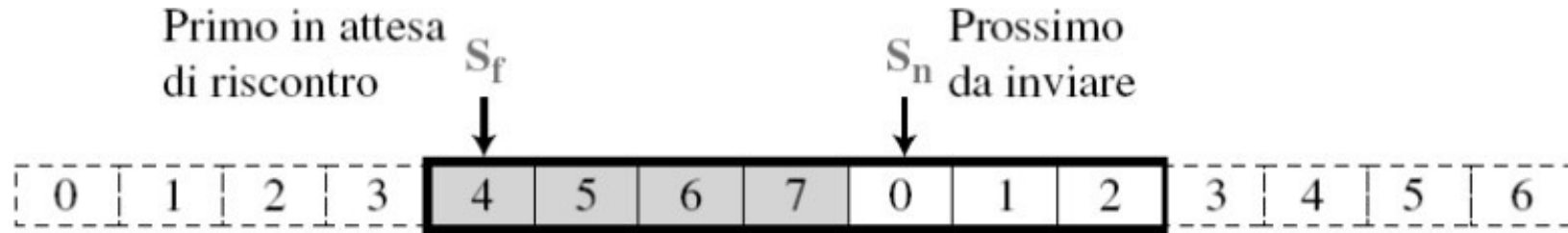
Go-Back-N: mittente

- mittente: ha una "finestra" fino a N pacchetti consecutivi trasmessi senza ACK
 - numero di sequenza di k-bit > 2 nell'header del pacchetto

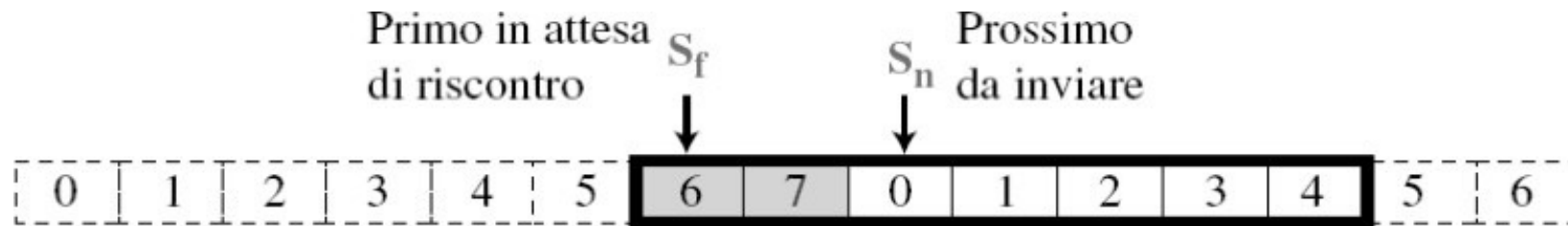


- **ACK cumulativo:** $ACK(n)$: ACK di tutti i pacchetti fino a n
 - alla ricezione di $ACK(n)$: sposta la finestra in avanti per iniziare da $n+1$
- timer per il pacchetto in volo più vecchio (primo giallo)
- **timeout(n):** ritrasmette il pacchetto n e tutti i pacchetti con # seq superiore nella finestra

Esempio di scorrimento della finestra di invio



a. Finestra prima dello scorrimento.

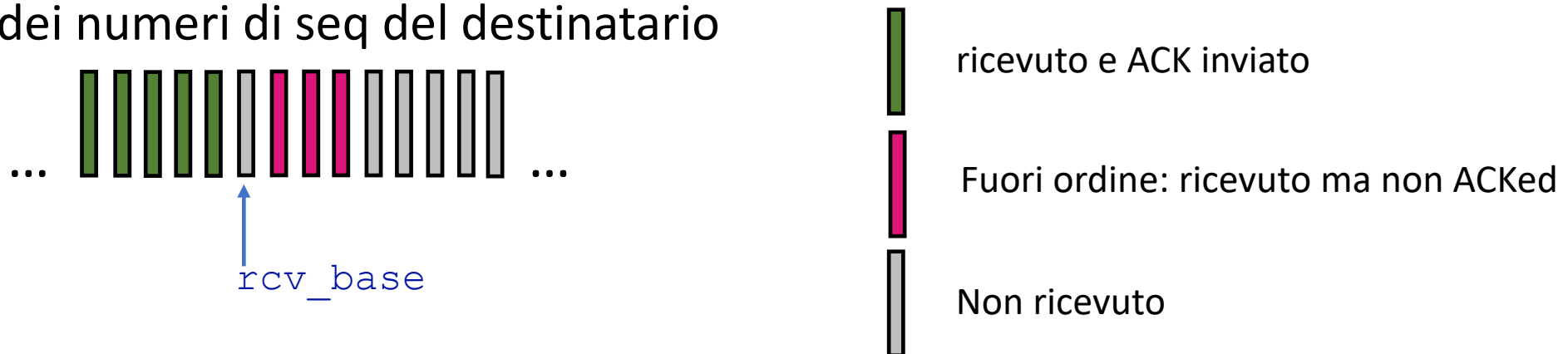


b. Finestra dopo lo scorrimento (è arrivato un ACK con ackNo = 6).

Go-Back-N: destinatario

- Solo ACK: invia sempre ACK per i pacchetti ricevuti correttamente fino a quel momento, con il # seq più alto *in ordine* (per cui non ci sono pacchetti di seq minore mancanti)
 - potrebbe generare ACK duplicati
 - deve solo ricordare `rcv_base` (# seq dove sono arrivato)
- al ricevimento del pacchetto fuori ordine:
 - può scartarlo (don't buffer) o buffer: una decisione di implementazione (**default don't**)
 - Manda un ACK con il più alto # seq in ordine (possibilmente duplicato)

Spazio dei numeri di seq del destinatario



FSM mittente

Mittente

Nota:

tutte le equazioni aritmetiche sono modulo 2^m .

Time-out

Rispedisci tutti i pacchetti in attesa di riscontro.
Riavvia il timer.

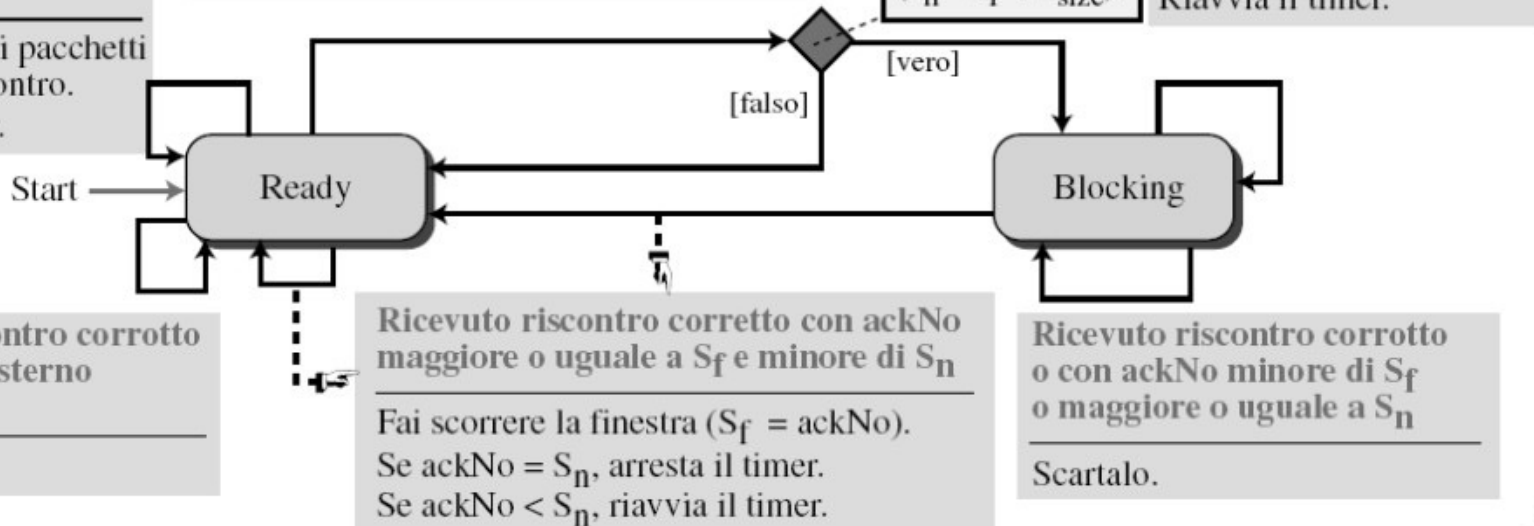
Ricevuto messaggio dal processo

Costruisci un pacchetto ($\text{seqNo} = S_n$).
Memorizzane una copia e invia il pacchetto.
Avvia il timer se non è già attivo.
 $S_n = S_n + 1$.

Time-out

Rispedisci tutti i pacchetti in attesa di riscontro.
Riavvia il timer.

Finestra piena
($S_n = S_f + S_{\text{size}}$)?



FSM destinatario

Destinatario

Nota:

tutte le equazioni
aritmetiche
sono modulo 2^m .

Ricevuto pacchetto corrotto

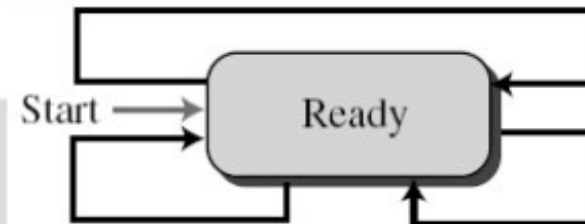
Scarta il pacchetto.

**Ricevuto pacchetto integro con
 $\text{seqNo} = R_n$**

Consegna il messaggio.

Fai scorrere la finestra ($R_n = R_n + 1$).

Invia un ACK ($\text{ackNo} = R_n$).

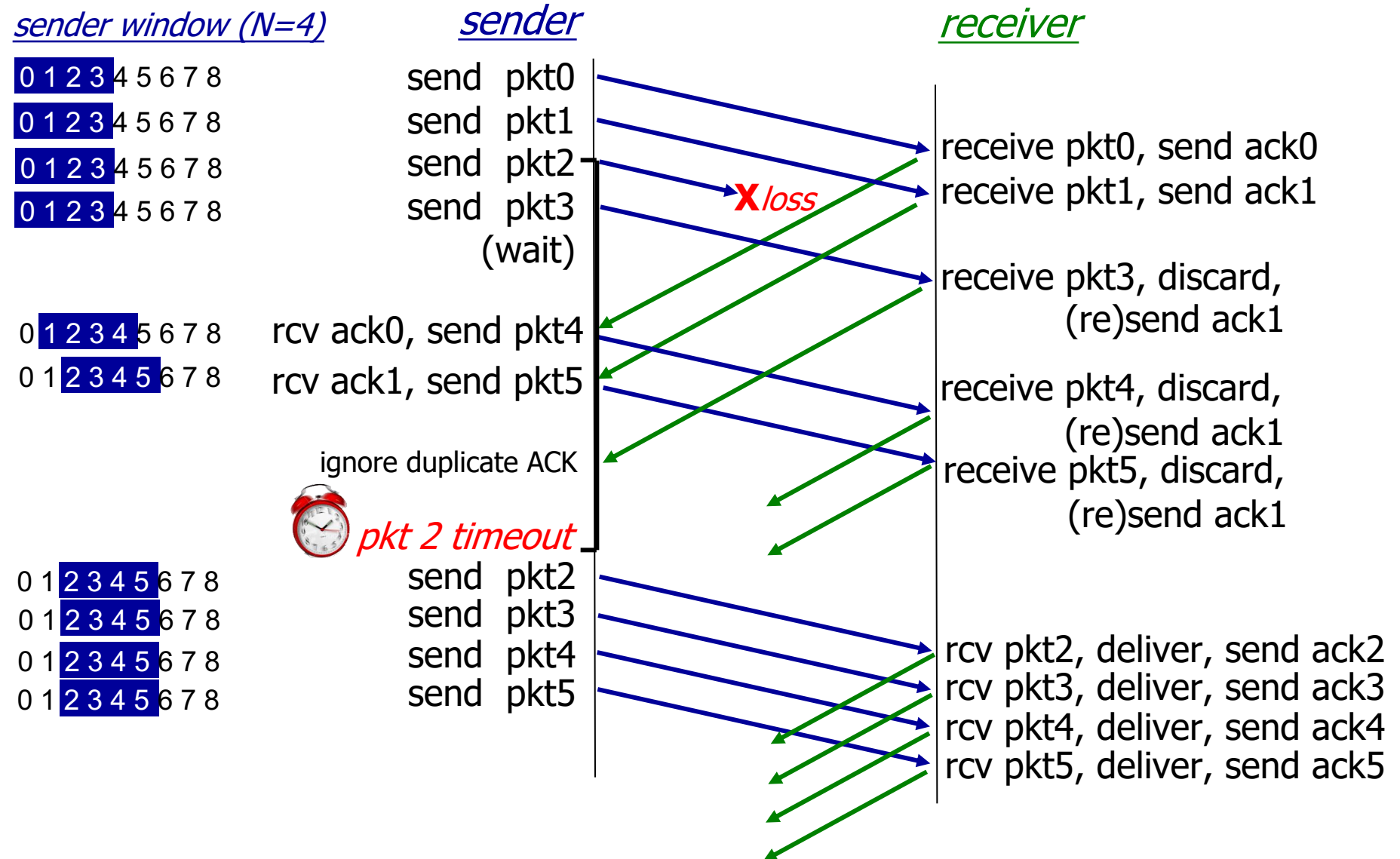


**Ricevuto pacchetto integro
con $\text{seqNo} \neq R_n$**

Scarta il pacchetto.

Invia un ACK ($\text{ackNo} = R_n$).

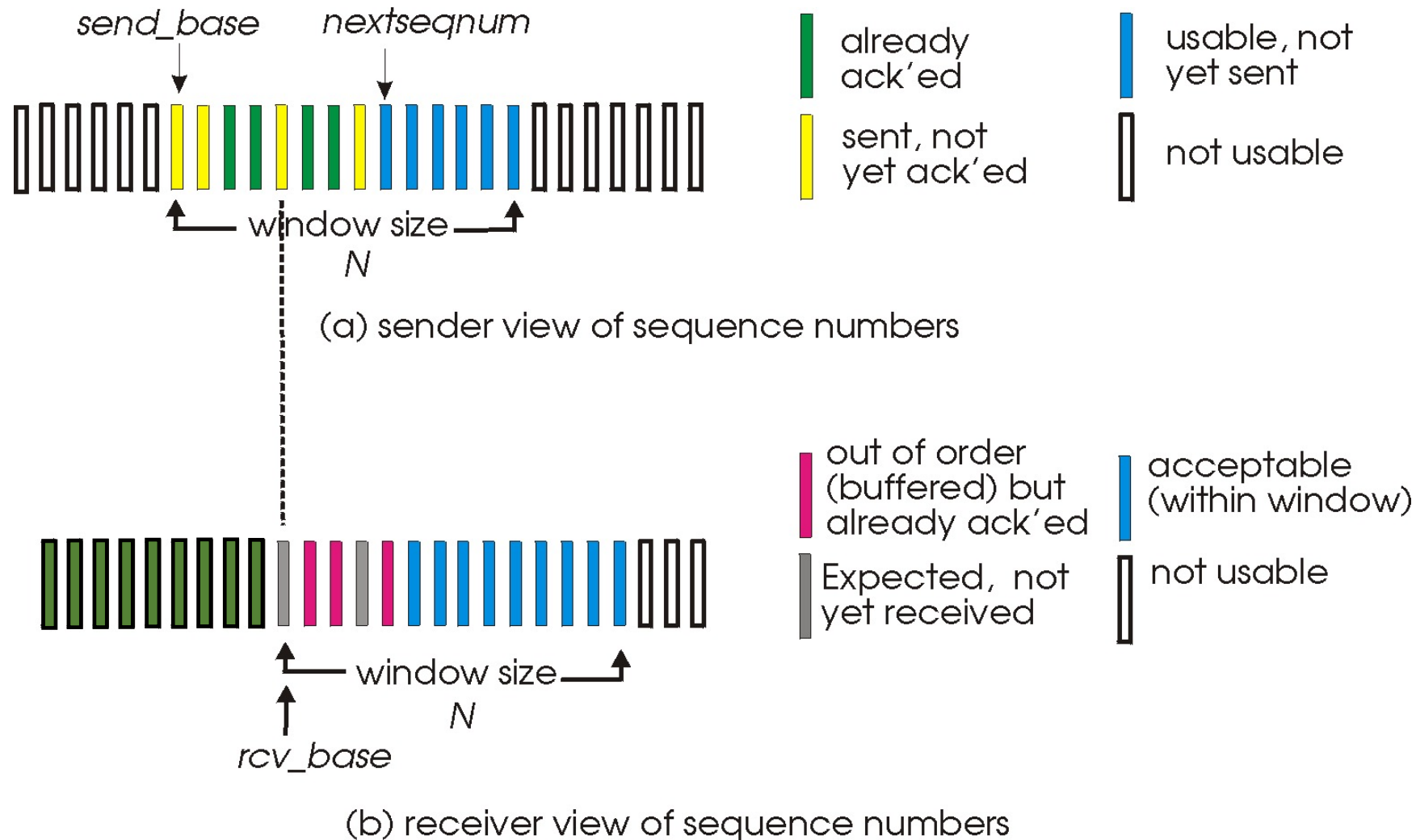
Go-Back-N in azione (without receive buffer)



Ripetizione selettiva (Selective repeat)

- il destinatario conferma *individualmente* tutti i pacchetti ricevuti correttamente (anche se fuori sequenza)
 - bufferizza i pacchetti, se necessario, per l'eventuale consegna in ordine al livello superiore
- il mittente re-invia ogni pacchetto alla scadenza del timeout individualmente
 - il mittente mantiene un timer per ogni pacchetto non ACKed
- La finestra del mittente scorre a partire dal pacchetto più alto confermato in ordine (senza pacchetti non confermati prima di esso)
- Perché?
 - In GBN per un solo pacchetto perso si ritrasmettono tutti i successivi già inviati nella pipeline
 - Se in rete c'è congestione, la rispedizione di tutti i pacchetti peggiora la congestione

Selective repeat: finestra di mittente e dest



Ripetizione selettiva: mittente e destinatario

mittente

dati dal livello superiore:

- se ci sono slot disponibili nella finestra, invia il pacchetto

timeout(n):

- invia nuovamente il pacchetto n , riavvia il timer per quel pacchetto

ACK(n) in $[\text{sendbase}, \text{sendbase}+N]$:

- contrassegna il pacchetto n come ricevuto
- se n era il pacchetto più vecchio ancora non confermato, avanza la finestra al prossimo pacchetto non confermato

destinatario

pacchetto n in $[\text{rcvbase}, \text{rcvbase}+N-1]$

- invia ACK(n)
- fuori ordine: buffer
- in ordine: consegna al livello superiore (consegna anche pacchetti bufferizzati contigui), avanza la finestra al successivo pacchetto non ancora ricevuto

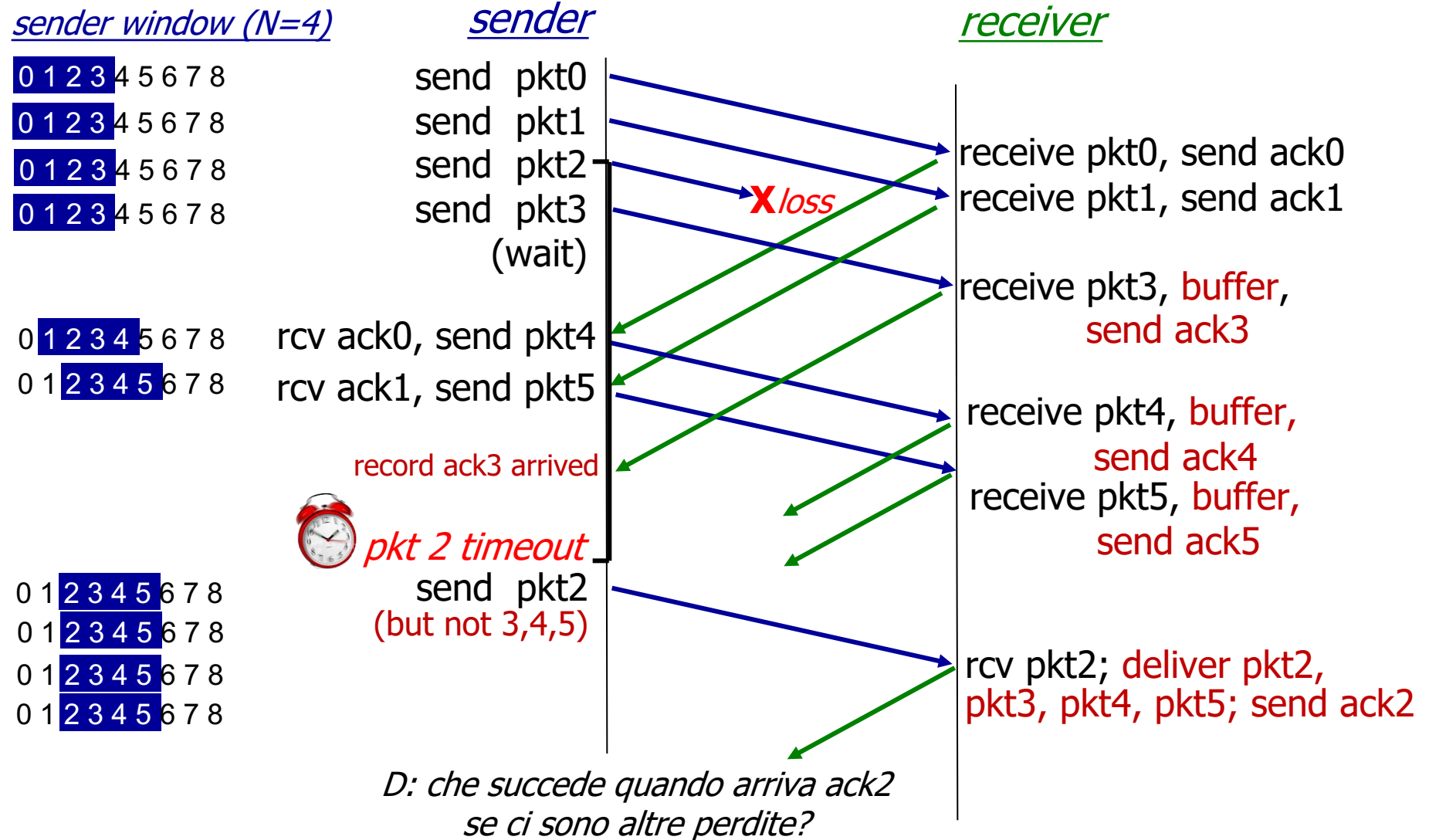
pacchetto n in $[\text{rcvbase}-N, \text{rcvbase}-1]$

- ACK(n)

Altrimenti:

- ignora

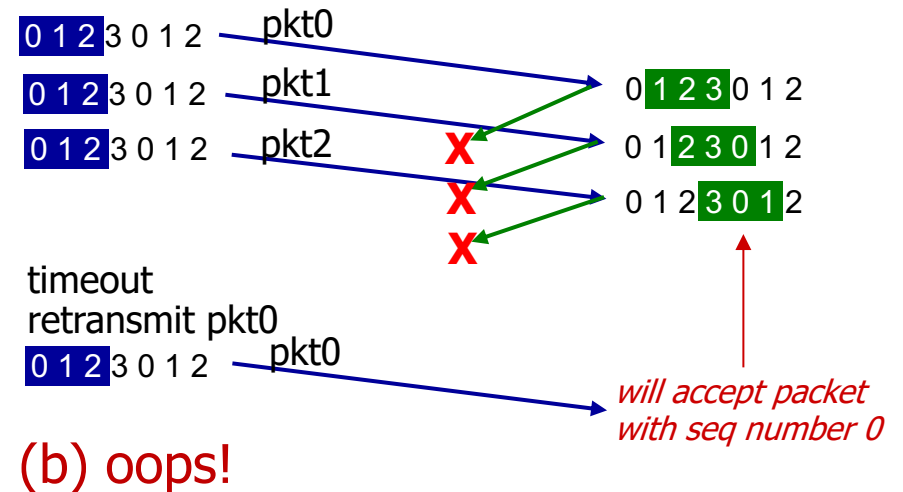
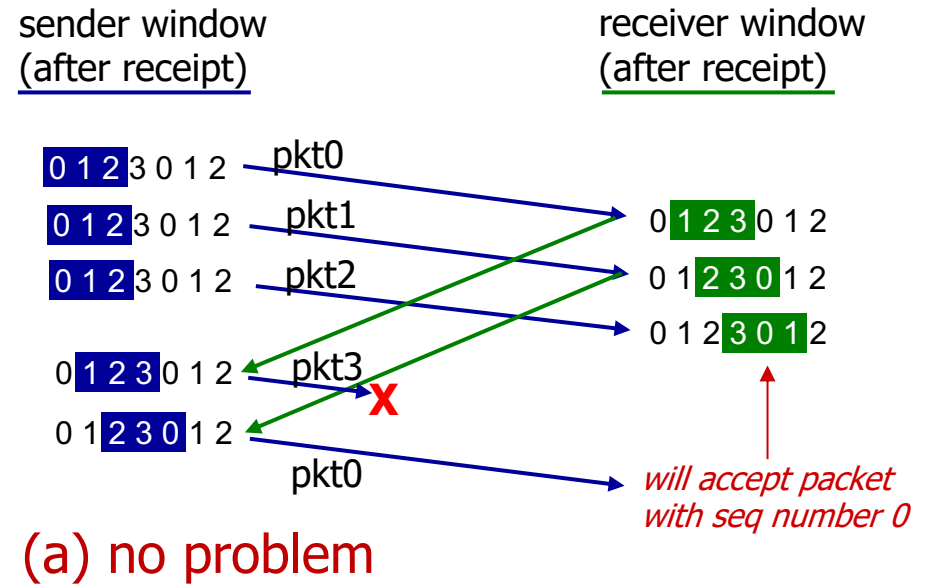
Selective Repeat in azione



Selective repeat: un dilemma!

esempio:

- numeri di seq: 0,1,2,3 (base 2^2)
- dimensione finestra=3 = $(2^2 - 1)$



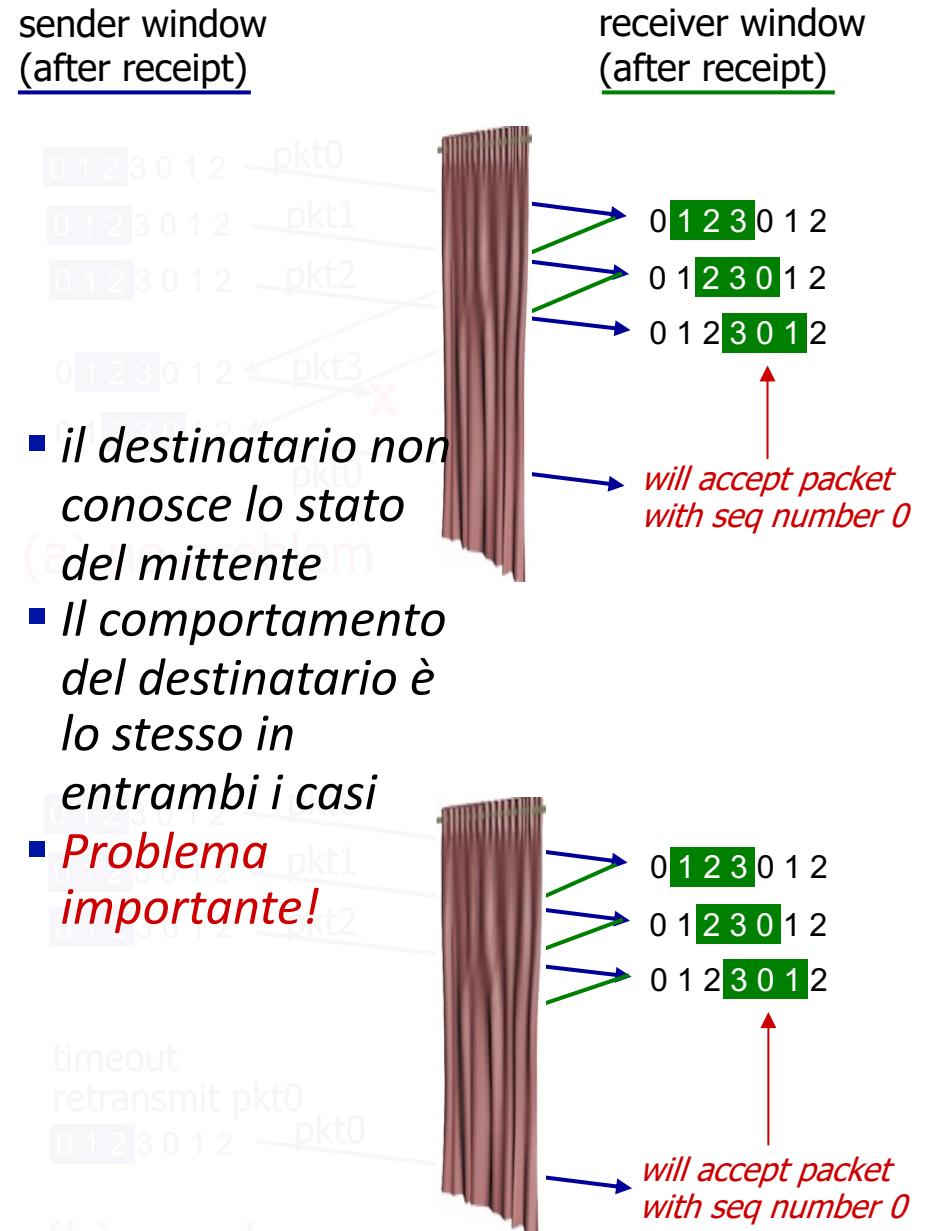
Selective repeat: un dilemma!

esempio:

- numeri di seq: 0,1,2,3 (base 2^2)
- dimensione finestra=3 = ($2^2 - 1$)

Q: qual è la relazione tra dimensione finestra e numeri di sequenza per evitare questo problema?

se la sequenza è in base 2^m , la
finestra può essere al più grande 2^{m-1}



Riepilogo

- Stop-and-wait (rdt3.0): timeout per gestione delle ritrasmissioni e dei pacchetti corrotti
- Go-back-N: il mittente ha una finestra di trasmissione, il destinatario (di default) scarta pacchetti fuori ordine. ACK cumulativo, al timeout il mittente reinvia tutti i pacchetti dall'ultimo ACK in su
- Selective repeat: mittente e destinatario hanno una finestra di trasmissione. ACK selettivo, buffer dei pacchetti fuori ordine. Re-Invio al timeout (individuale di ogni pacchetto)

Esercizio 1

Usando numeri di sequenza a $m=5$ bit, qual è la dimensione massima delle finestre di invio e di ricezione per ciascuno dei meccanismi seguenti?

a) Stop-and-Wait b) Go-Back-N c) Selective-Repeat

a) Stop-and-Wait:

MaxSendWsize = 1

MaxReceiveWsize=1

b) Go-Back-N:

MaxSendWsize= $2^5 - 1 = 31$

MaxReceiveWsize=1

c) Selective-Repeat:

Max Send Wsize = $2^{5-1} = 16$

Max Receive Wsize = $2^{5-1} = 16$

Esercizio 2

In una rete con un valore fisso $m > 1$ (numero di bit della sequenza), è possibile utilizzare entrambi i meccanismi Go-Back-N e Selective-Repeat

1. Indicare i vantaggi e gli svantaggi dell'impiego di ciascuno di essi
2. Quali altre considerazioni si devono fare per decidere quale meccanismo utilizzare?

Confronto

GO-BACK-N

- Ritrasmette tutti i frame inviati dopo il frame che si sospetta essere danneggiato o perso
- Se il tasso di errore è alto, spreca molta larghezza di banda
- Meno complicato
- Window size $N-1 = 2^m-1$
- Riordinamento non è richiesto né lato mittente né lato destinatario
- Il destinatario non memorizza i frame ricevuti dopo il frame corrotto finché esso non viene ritrasmesso (dipende implem)
- Non è richiesta alcuna ricerca di frame né lato mittente né destinatario



SELECTIVE REPEAT

- Ritrasmette solo i frame sospettati di essere persi o danneggiati
- Comparativamente meno larghezza di banda viene sprecata nella ritrasmissione
- Più complesso in quanto richiede l'applicazione di logica aggiuntiva, ordinamento e archiviazione, lato mittente e destinatario
- Window size $(N+1)/2 = 2^{m-1}$
- Il destinatario deve essere in grado di ordinare in quanto deve mantenere la sequenza dei frame
- Il destinatario memorizza i frame ricevuti dopo il frame danneggiato nel buffer finché il frame danneggiato non viene sostituito
- Il mittente deve essere in grado di cercare e selezionare il frame richiesto

Protocolli bidirezionali: piggybacking

- Abbiamo mostrato meccanismi unidirezionali: pacchetti dati in una direzione e ack nella direzione opposta
- In realtà entrambi viaggiano nelle due direzioni (due protocolli rdt in parallelo)
- Per migliorare l'efficienza dei protocolli bidirezionali viene utilizzata la tecnica del ***piggybacking***: quando un pacchetto trasporta dati da A a B, può trasportare anche i riscontri relativi ai pacchetti ricevuti da B e viceversa

Riassunto dei meccanismi di trasferimento dati affidabile e loro utilizzo

Meccanismo	Uso	
Checksum	Per gestire errori nel canale	 Gestione inaffidabilità della rete
Acknowledgment	Per gestire errori nel canale	
Numero di sequenza	Ack con errori Perdita pacchetti	
Timeout	Perdita pacchetti	
Finestra scorrevole, pipeling	Maggior utilizzo della rete	 Miglioramento prestazioni

- ☐ Meccanismi per realizzare un trasferimento dati affidabile in un contesto generale
- ☐ TCP come e quali meccanismi usa per realizzare un trasferimento affidabile?

Socket programming con TCP

Il client contatta il server

- il processo del server deve essere già attivo
- il server deve avere un socket attivo per il primo contatto

Procedura di contatto server

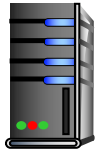
- Il client crea un socket TCP, specificando l'indirizzo IP e numero di porta del server
- *quando il client crea il socket:* TCP del client stabilisce una connessione con TCP del server

- quando contattato dal client, il *server TCP crea un nuovo socket* specifico per quella connessione (IP del client, porta locale del client)
 - il server può comunicare con più client

Applicazione

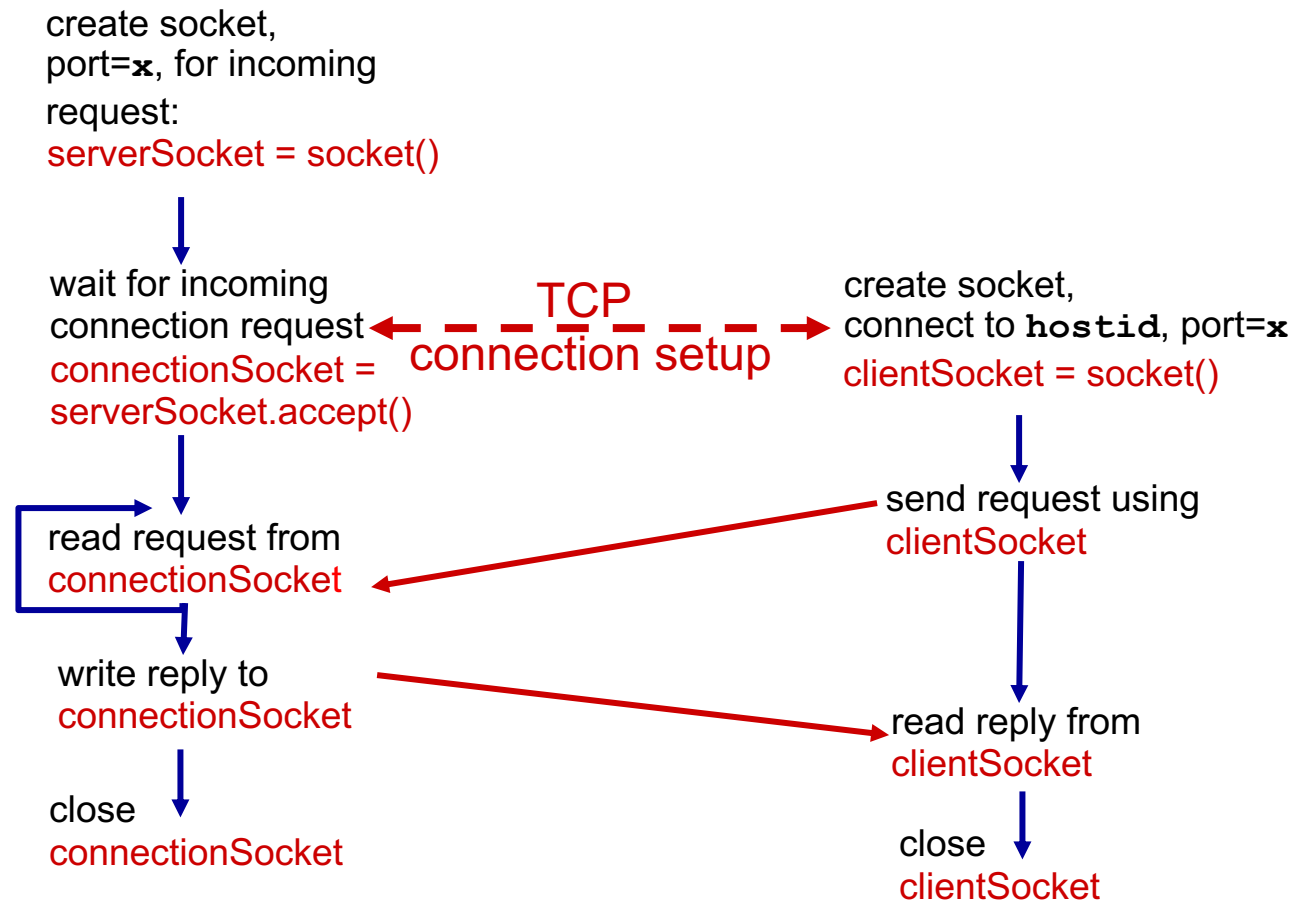
TCP fornisce trasmissione di dati affidabile e in ordine di byte-stream (flussi) tra client e server

Interazione TCP client/server



server (running on `hostid`)

client



Esempio: client TCP

Python TCPClient

create TCP socket for server,
remote port 12000

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

#use pickle.dumps() to serialize a variable

Esempio: server TCP

Python TCP Server

	from socket import *
	serverPort = 12000
create TCP welcoming socket →	serverSocket = socket(AF_INET,SOCK_STREAM)
	serverSocket.bind(('',serverPort))
server begins listening for incoming TCP requests →	serverSocket.listen(1)
	print 'The server is ready to receive'
loop forever →	while True:
server waits on accept() for incoming requests, new socket created on return →	connectionSocket, addr = serverSocket.accept()
read bytes from connection socket (but not address as in UDP) →	sentence = connectionSocket.recv(1024).decode()
	capitalizedSentence = sentence.upper()
	connectionSocket.send(capitalizedSentence.encode())
close connection to this client (but <i>not</i> welcoming socket) →	connectionSocket.close()

#use pickle.loads() to deserialize a bytestream