

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

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2024/2025

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



UDP: User Datagram Protocol

- protocollo di trasporto di Internet "senza fronzoli"
- servizio di consegna "best effort" (massimo sforzo), i segmenti UDP possono essere:
 - perduti
 - consegnati fuori sequenza all'applicazione
- *senza connessione*
 - no handshaking tra mittente e destinatario UDP
 - ogni segmento UDP è gestito indipendentemente

Perché esiste UDP?

- nessuna connessione stabilita (che potrebbe aggiungere ritardo)
- semplice: nessuno stato di connessione nel mittente e nel destinatario (perciò un server può gestire più client)
- intestazioni di segmento corte
- senza controllo della congestione
 - UDP può sparare dati a raffica!
- controllo più preciso a livello di applicazione su quali dati sono inviati e quando (utile per applicazioni con requisiti di velocità di invio minima e ritardi limitati, ma che sono tolleranti alle perdite)

UDP: User Datagram Protocol

- utilizzo di UDP:
 - applicazioni per lo streaming multimediale (tolleranti alle perdite, sensibili alla banda)
 - DNS
 - SNMP
 - HTTP/3
- trasferimento affidabile con UDP (ad esempio, HTTP/3):
 - aggiungere affidabilità a livello di applicazione
 - aggiungere controllo della congestione a livello di applicazione

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768

J. Postel

ISI

28 August 1980

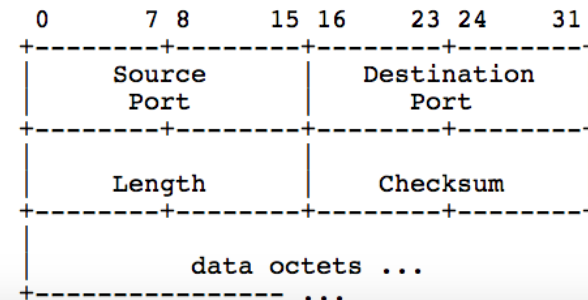
User Datagram Protocol

Introduction

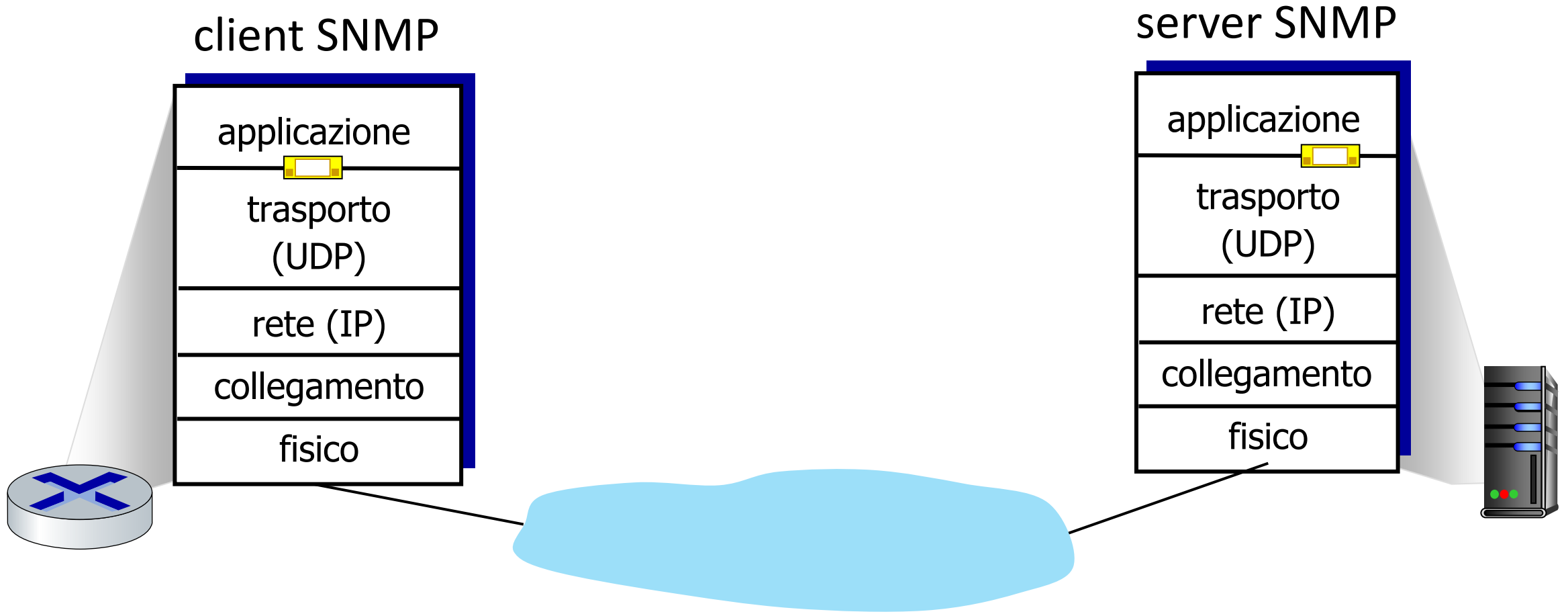
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

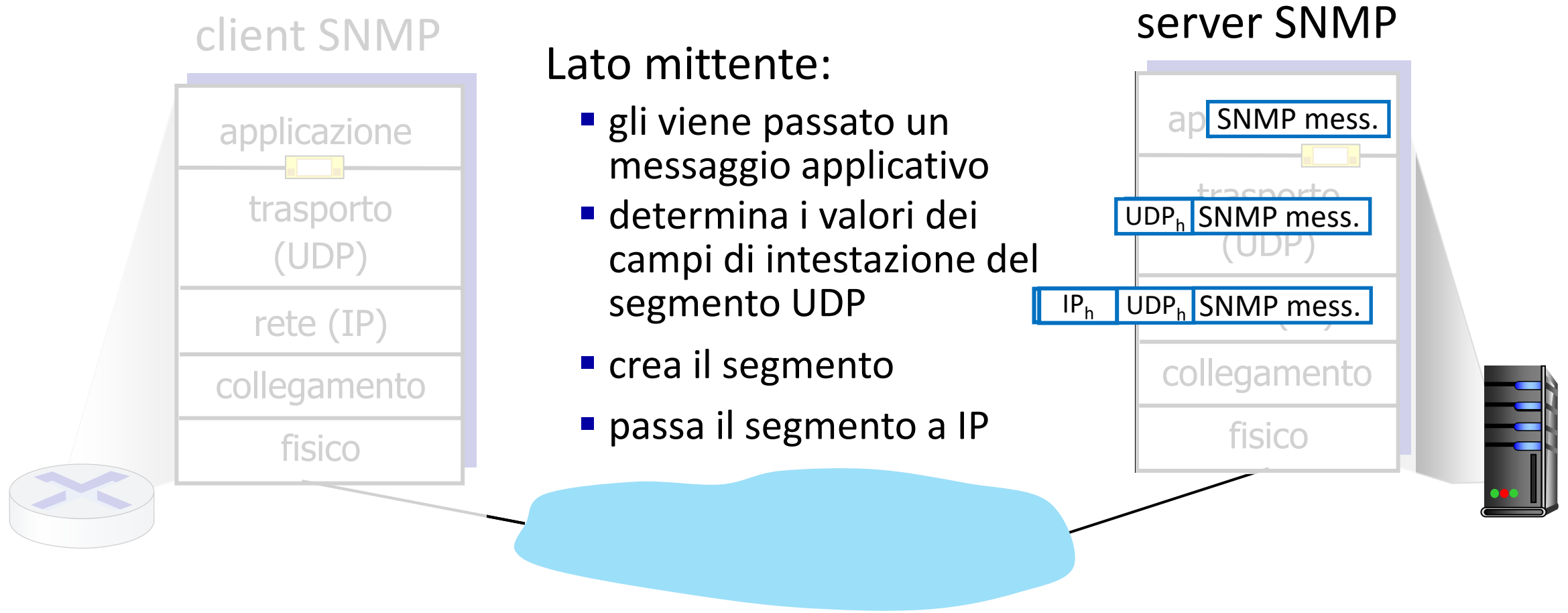
Format



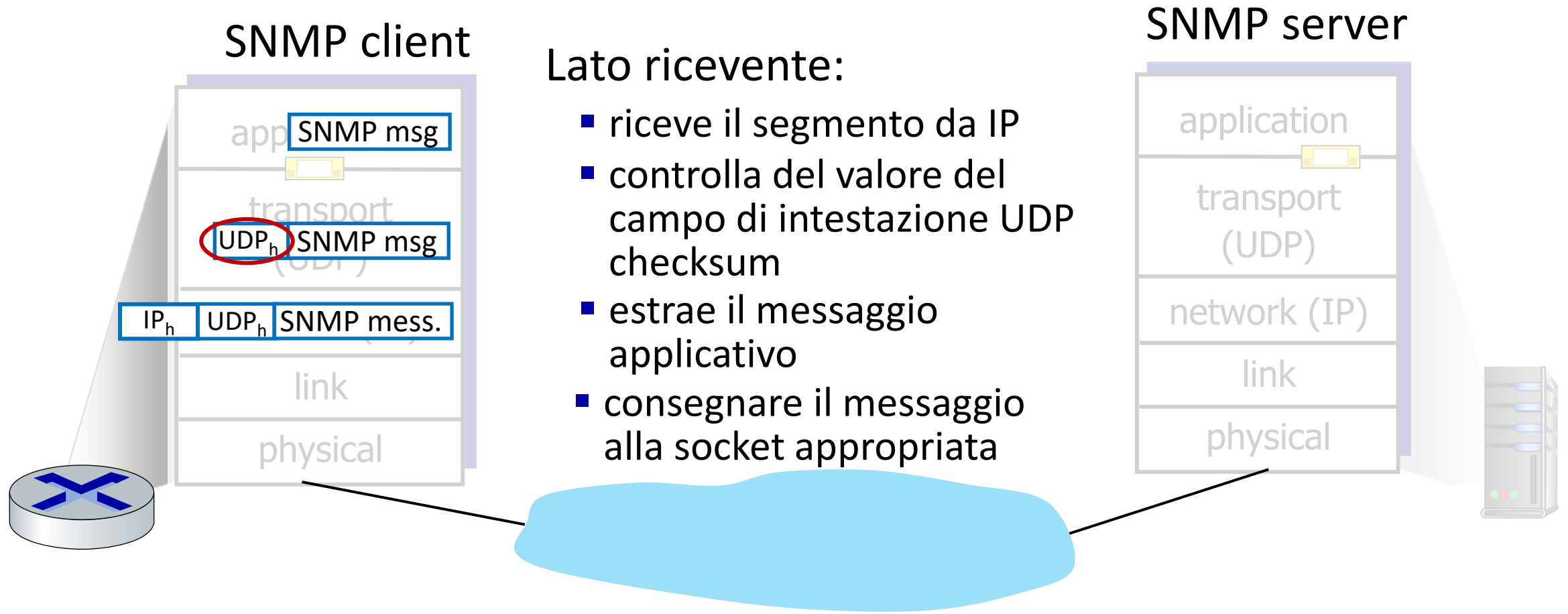
UDP: azioni del livello di trasporto



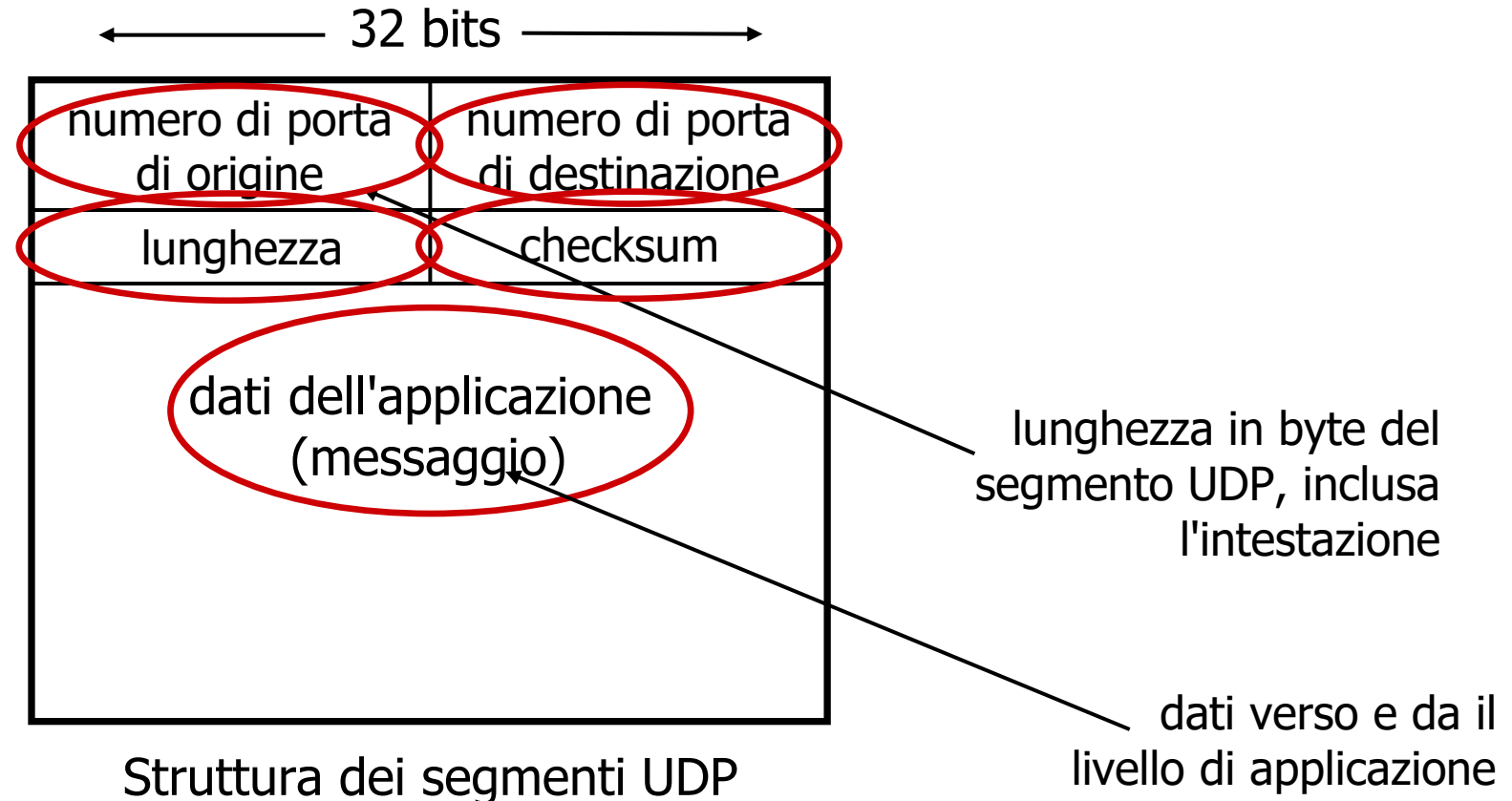
UDP: azioni del livello di trasporto



UDP: azioni del livello di trasporto

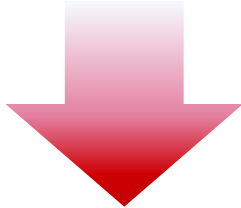


Struttura dei segmenti UDP



Checksum UDP

Obiettivo: rilevare gli "errori" (*bit alternati*) nel segmento trasmesso

	1° numero		2° numero		somma	
messaggio:	5		6		11	
						
trasmesso:	5	+	6	+	-11	= 0
						≠
ricevuto:	4	+	6	+	-11	= -1

errore rilevato
dal ricevente

Checksum Internet [RFC 1071, RFC 1624]

Obiettivo: rilevare gli "errori" (*bit alternati*) nel segmento trasmesso

mittente:

- tratta il contenuto del segmento come una sequenza di interi da 16 bit (inclusi i campi dell'intestazione UDP e gli indirizzi IP)
- **checksum:** complemento a 1 della somma (in complemento a 1) della sequenza di interi a 16 bit (considerando il campo checksum uguale a 0)
- pone il valore della checksum nel campo checksum del segmento UDP

ricevente:

- calcola la checksum allo stesso modo del mittente (ma sommando anche il valore ricevuto del checksum)
- Il risultato è costituito da tutti bit 1 (-0 nell'aritmetica del complemento a 1)?
 - Sì - nessun errore rilevato. Ma potrebbero esserci errori nonostante questo? Lo scopriremo più avanti....
 - No - errore rilevato
- Altre implementazioni verificano la checksum calcolandola (allo stesso modo del mittente) e confrontandola col valore ricevuto

Checksum Internet: un esempio

esempio: sommare due interi da 16 bit

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
<hr/>																	
a capo	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
<hr/>																	
somma	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0	
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1

Nota: quando si sommano i numeri, un riporto dal bit più significativo deve essere sommato al risultato

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

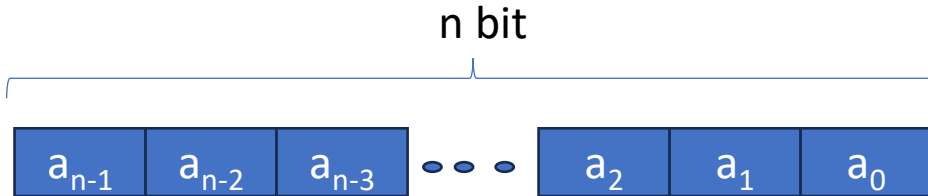
Checksum Internet: protezione debole

esempio: sommare due interi da 16 bit

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	<hr/>															
a capo	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
somma	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Nonostante i numeri siano cambiati (bit alternati), *nessun* cambiamento nella checksum!

Rappresentazione interi relativi in complemento a 1



Se $a_{n-1} = 0$, rappresenta un numero positivo (effettivamente di $n-1$ bit):

$$\sum_{i=0}^{n-2} a_i 2^i$$

Se $a_{n-1} = 1$, rappresenta un numero negativo, ottenuto invertendo tutti i bit:

$$-\sum_{i=0}^{n-2} \overline{a_i} 2^i$$

Possiamo rappresentare i numeri x nell'intervallo

$$-(2^{n-1} - 1) \leq x \leq 2^{n-1} - 1$$

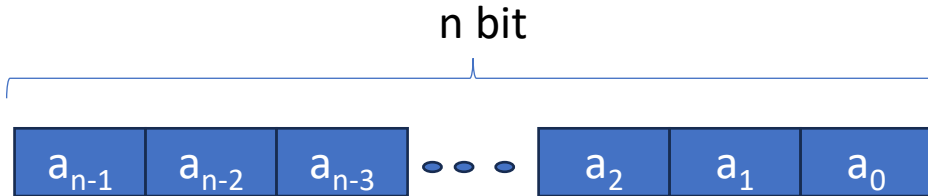
due rappresentazioni dello zero

a_2	a_1	a_0	Valore rappresentato in complemento a 1	Valore rappresentato senza segno
0	0	0	0	0
0	0	1	1	1
0	1	0	2	2
0	1	1	3	3
1	0	0	-3	4
1	0	1	-2	5
1	1	0	-1	6
1	1	1	-0	7

provate a sommare -2 e -1 per capire l'importanza del bit di riporto!

Se sommiamo un numero e il suo complemento a 1, otteniamo sempre tutti 1

Rappresentazione interi relativi in complemento a 1



Se $a_{n-1} = 1$, rappresenta un numero negativo, ottenuto invertendo tutti i bit:

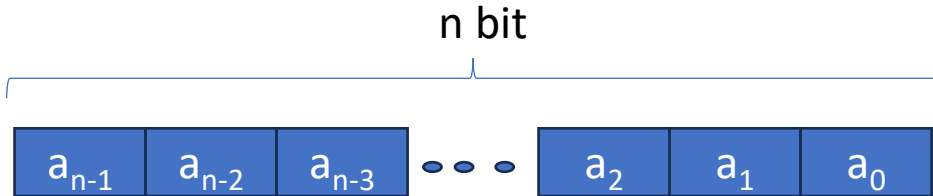
$$-\sum_{i=0}^{n-2} \overline{a_i} 2^i$$

Cioè

$$\begin{aligned} -\sum_{i=0}^{n-2} \overline{a_i} 2^i &= -\sum_{i=0}^{n-2} (1 - a_i) 2^i \\ &= -\sum_{i=0}^{n-2} 2^i + \sum_{i=0}^{n-2} a_i 2^i \\ &= (-2^{n-1} + 1) + \sum_{i=0}^{n-2} a_i 2^i \end{aligned}$$

a_2	a_1	a_0	Valore rappresentato in complemento a 1	Valore rappresentato senza segno
0	0	0	0	0
0	0	1	1	1
0	1	0	2	2
0	1	1	3	3
1	0	0	-3	4
1	0	1	-2	5
1	1	0	-1	6
1	1	1	-0	7

Rappresentazione interi relativi in complemento a 1



Rappresenta il numero relativo

$$a_{n-1}(-2^{n-1} + 1) + \sum_{i=0}^{n-2} a_i 2^i$$

a_2	a_1	a_0	Valore rappresentato in complemento a 1	Valore rappresentato senza segno
0	0	0	0	0
0	0	1	1	1
0	1	0	2	2
0	1	1	3	3
1	0	0	-3	4
1	0	1	-2	5
1	1	0	-1	6
1	1	1	-0	7

Riassunto: UDP

- protocollo “senza fronzoli” :
 - i segmenti possono essere persi, consegnati fuori sequenza
 - servizio best effort: “invia e spera per il meglio”
- UDP ha i propri vantaggi:
 - nessun bisogno di setup/handshaking (non si incorre in alcun RTT)
 - può funzionare quando il servizio di rete è compromesso
 - aiuta con l'affidabilità (checksum)
- costruire funzionalità aggiuntive su UDP nel livello di applicazione (ad esempio, HTTP/3)

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

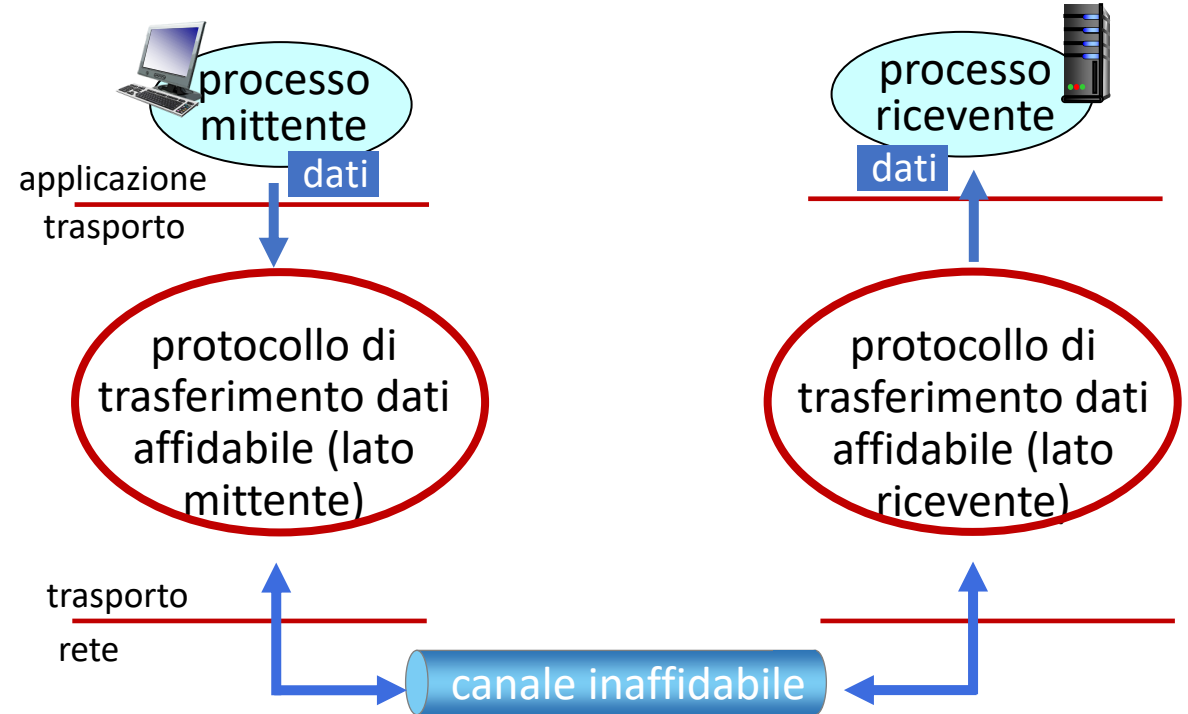
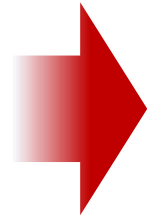
per il momento ci concentriamo su un canale unidirezionale, per l'invio affidabile di dati da un mittente a un destinatario.

Principi del trasferimento dati affidabile



astrazione di un servizio affidabile

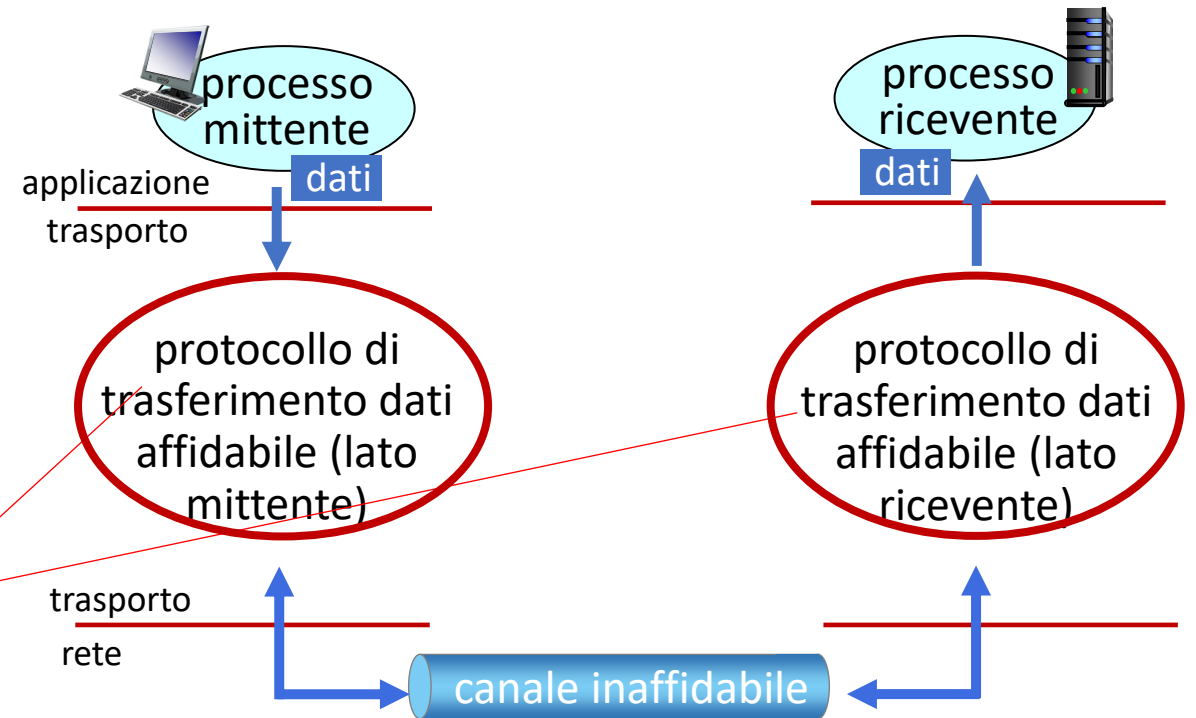
nonostante il nostro sia a livello astratto un canale di comunicazione unidirezionale, le due parti dentro al livello di trasporto devono comunicare in entrambi i versi usando il canale *inaffidabile* fornito dal livello di rete.



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?)

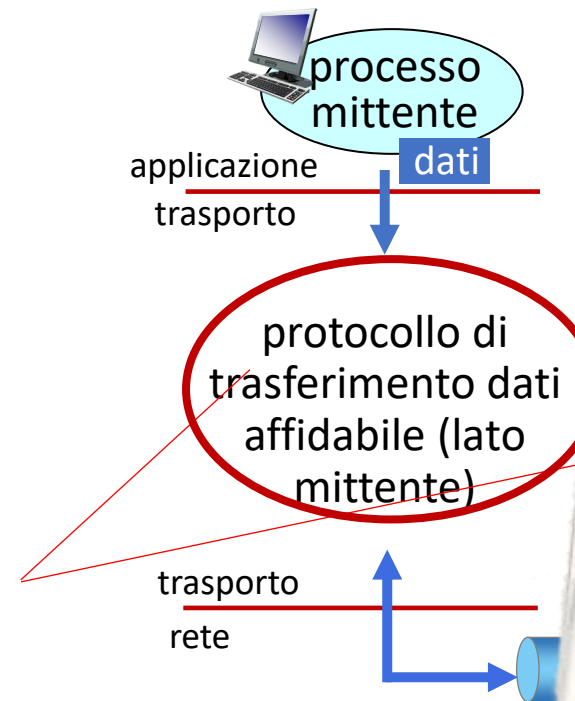


Implementazione di un servizio affidabile

Principi del trasferimento dati affidabile

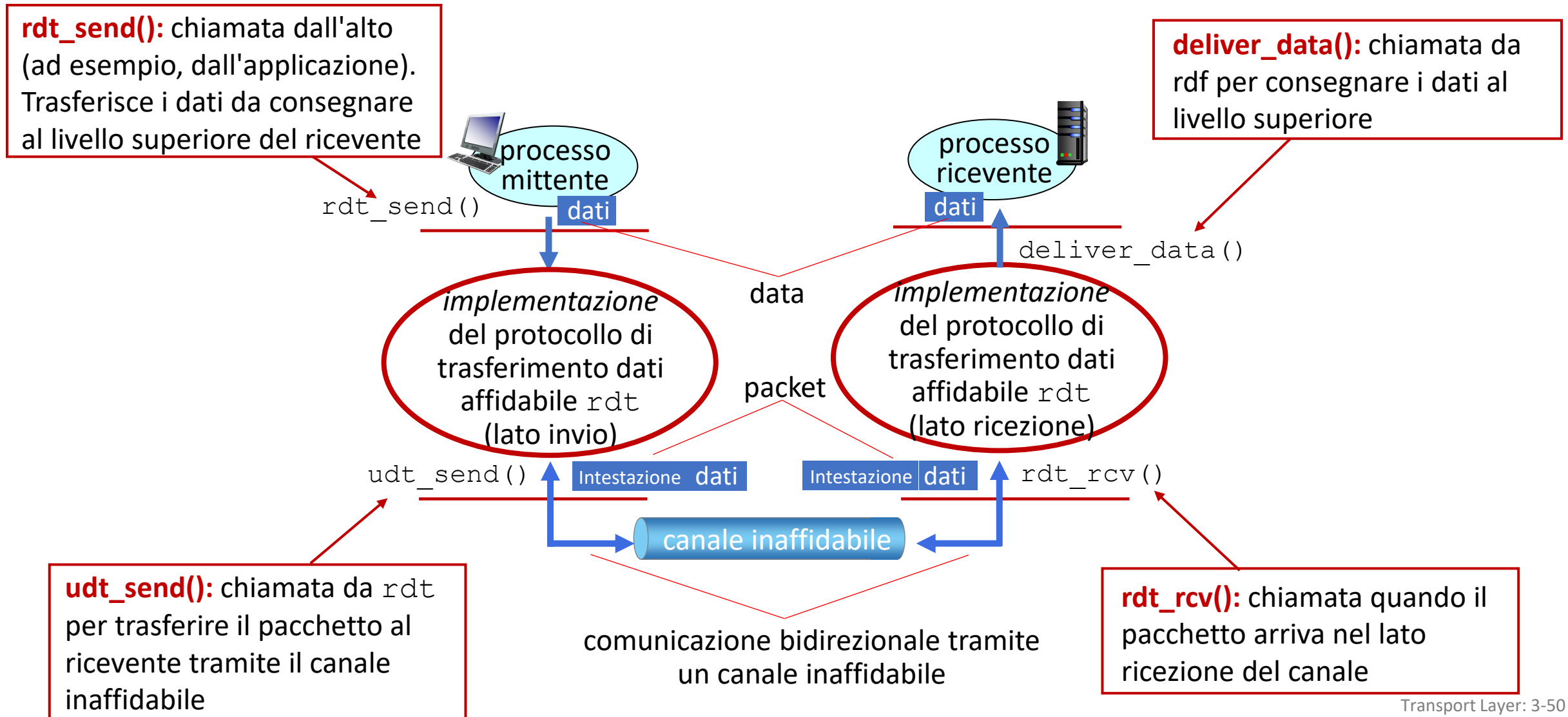
Il mittente e il ricevente *non* conoscono ciascuno lo "stato" dell'altro, ad esempio: il messaggio è stato ricevuto?

- a meno che non venga comunicato attraverso un messaggio



Implementazione di un servizio affidabile

Trasferimento dati affidabile (rdt): interfacce



Trasferimento dati affidabile: come iniziare

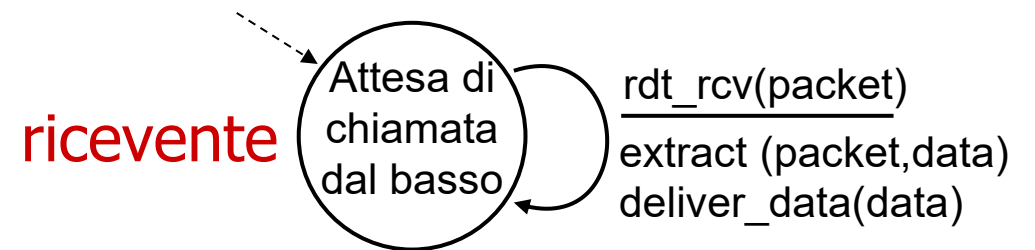
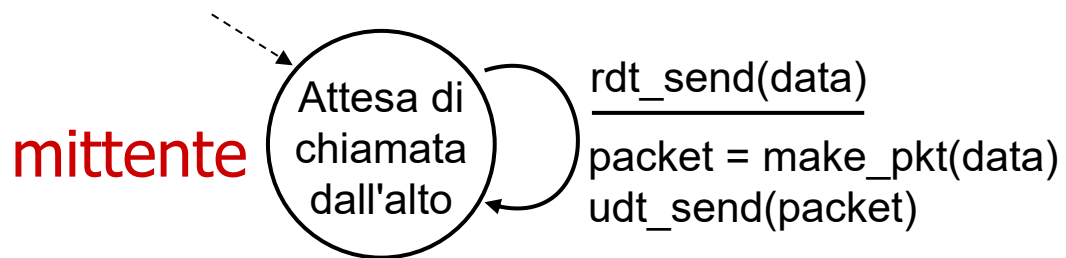
- svilupperemo progressivamente i lati d'invio e di ricezione di un protocollo di trasferimento dati affidabile, reliable data transfer protocol (rdt)
- considereremo soltanto i trasferimenti dati unidirezionali
 - ma le informazioni di controllo fluiranno in entrambe le direzioni!
- utilizzeremo macchine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



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?

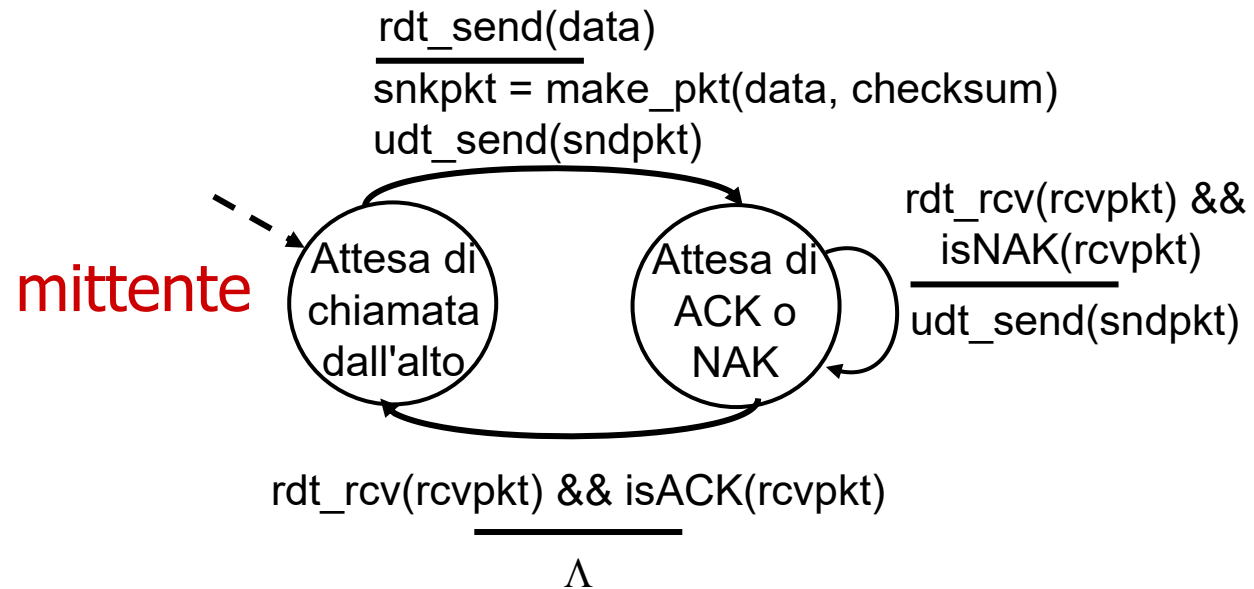
rdt2.0: canale con errori nei bit

- 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

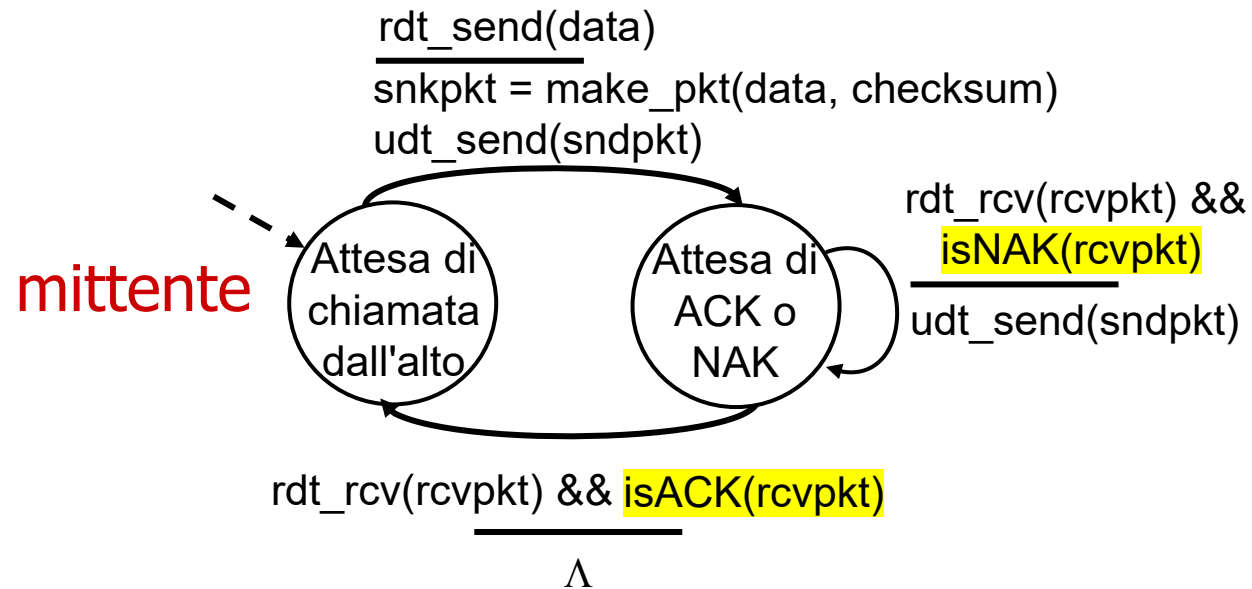
stop and wait

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

rdt2.0: specifica delle FSM



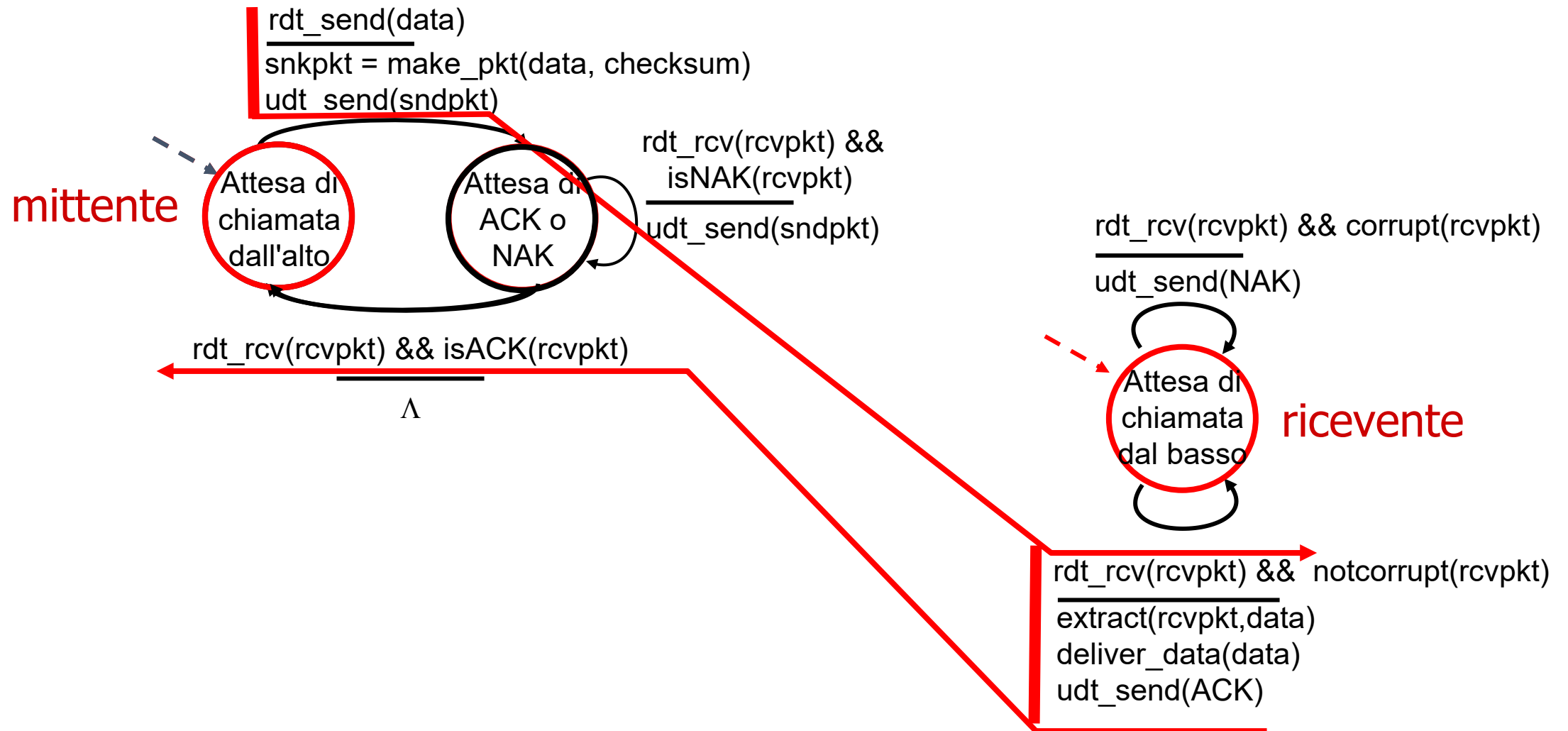
rdt2.0: specifica delle FSM



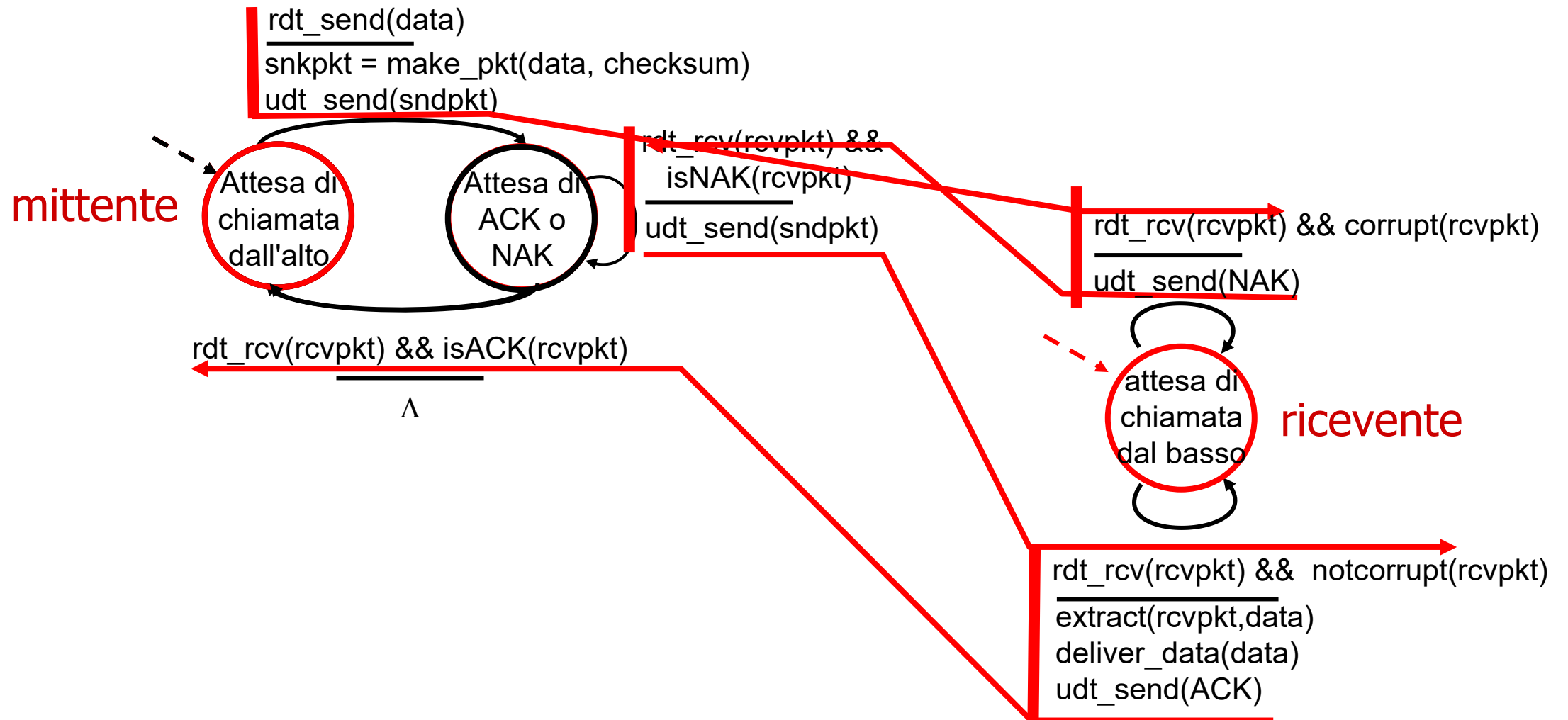
Nota: lo "stato" del destinatario (ha ricevuto correttamente il mio messaggio?) non è noto al mittente a meno che non venga comunicato in qualche modo dal destinatario al mittente.

- 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

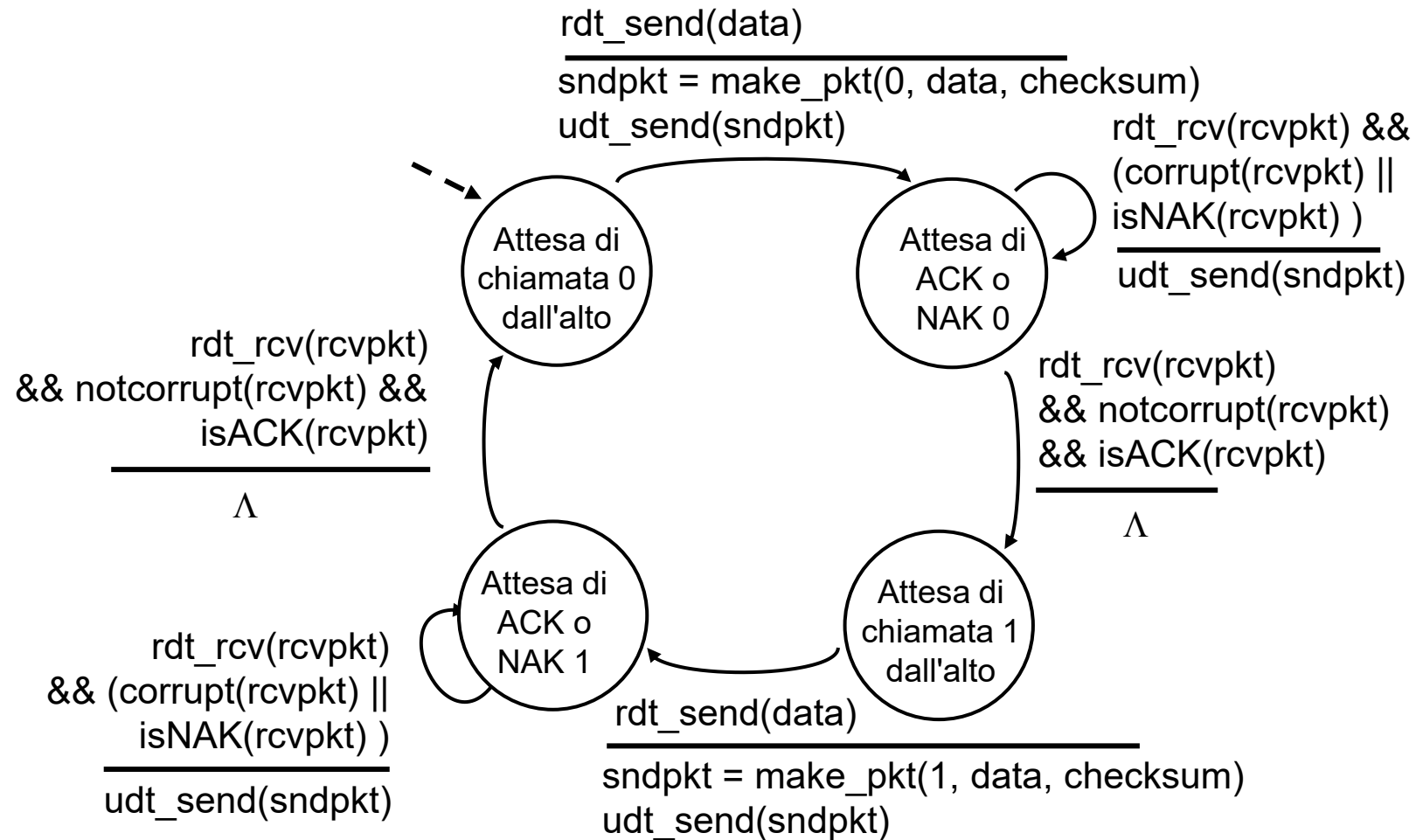
gestione dei duplicati

- 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

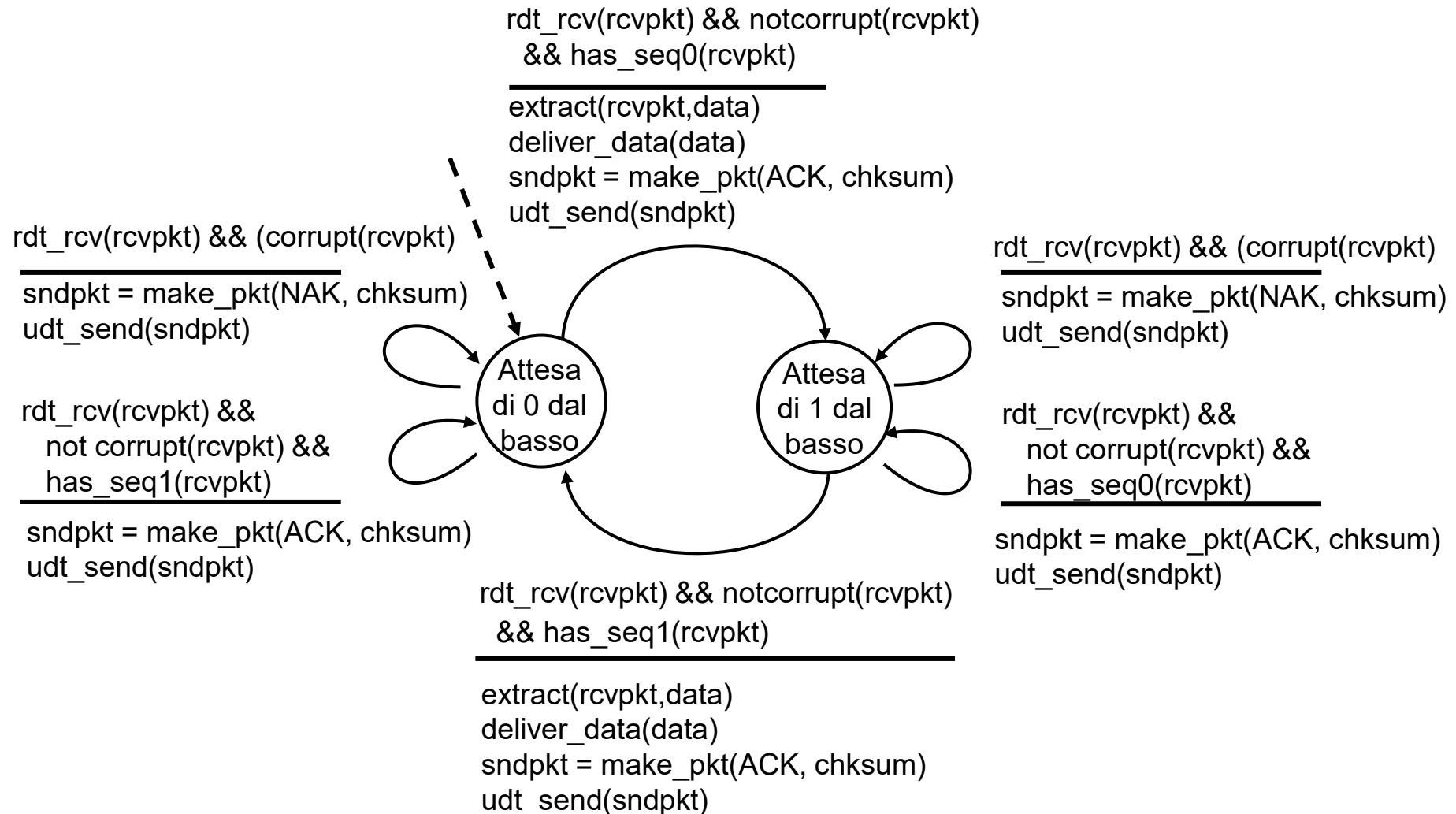
— stop and wait —

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:

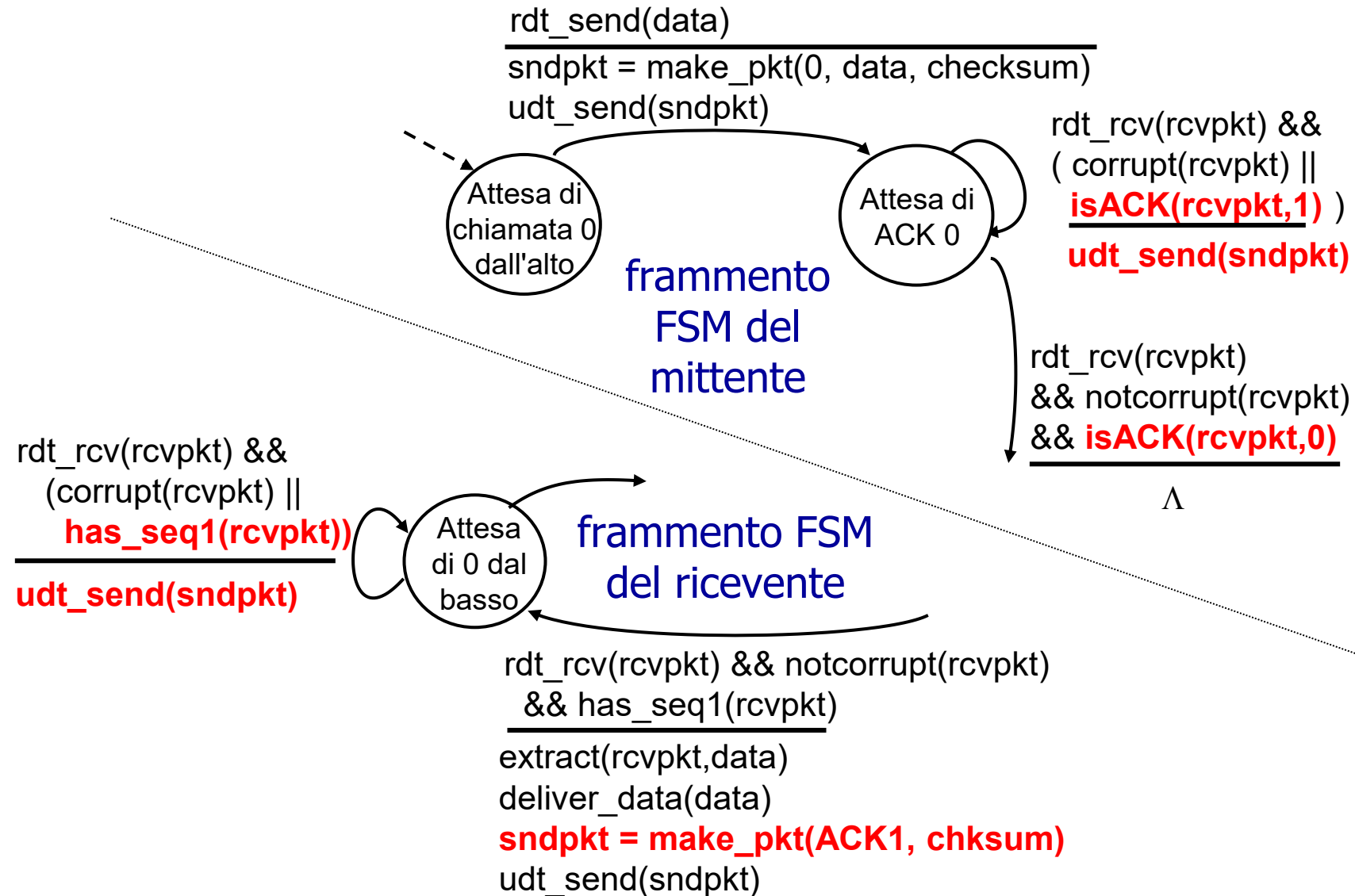
- deve controllare se il pacchetto ricevuto è duplicato
 - lo stato indica se il numero di sequenza previsto è 0 o 1
- nota: il ricevente *non* può sapere se il suo ultimo ACK/NAK è stato ricevuto correttamente dal mittente

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

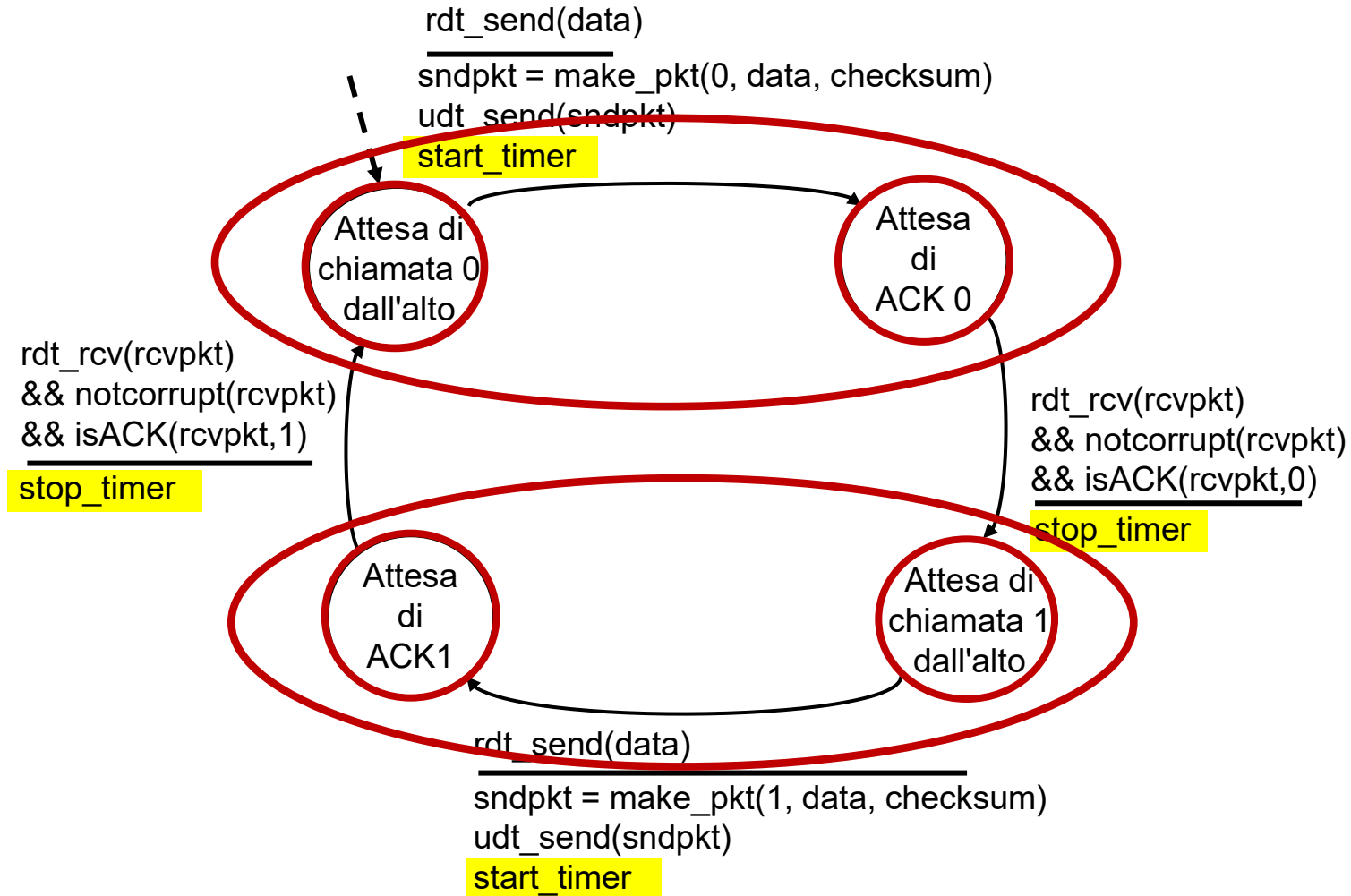
Approccio: il mittente attende un ACK per un tempo "ragionevole"

- 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"

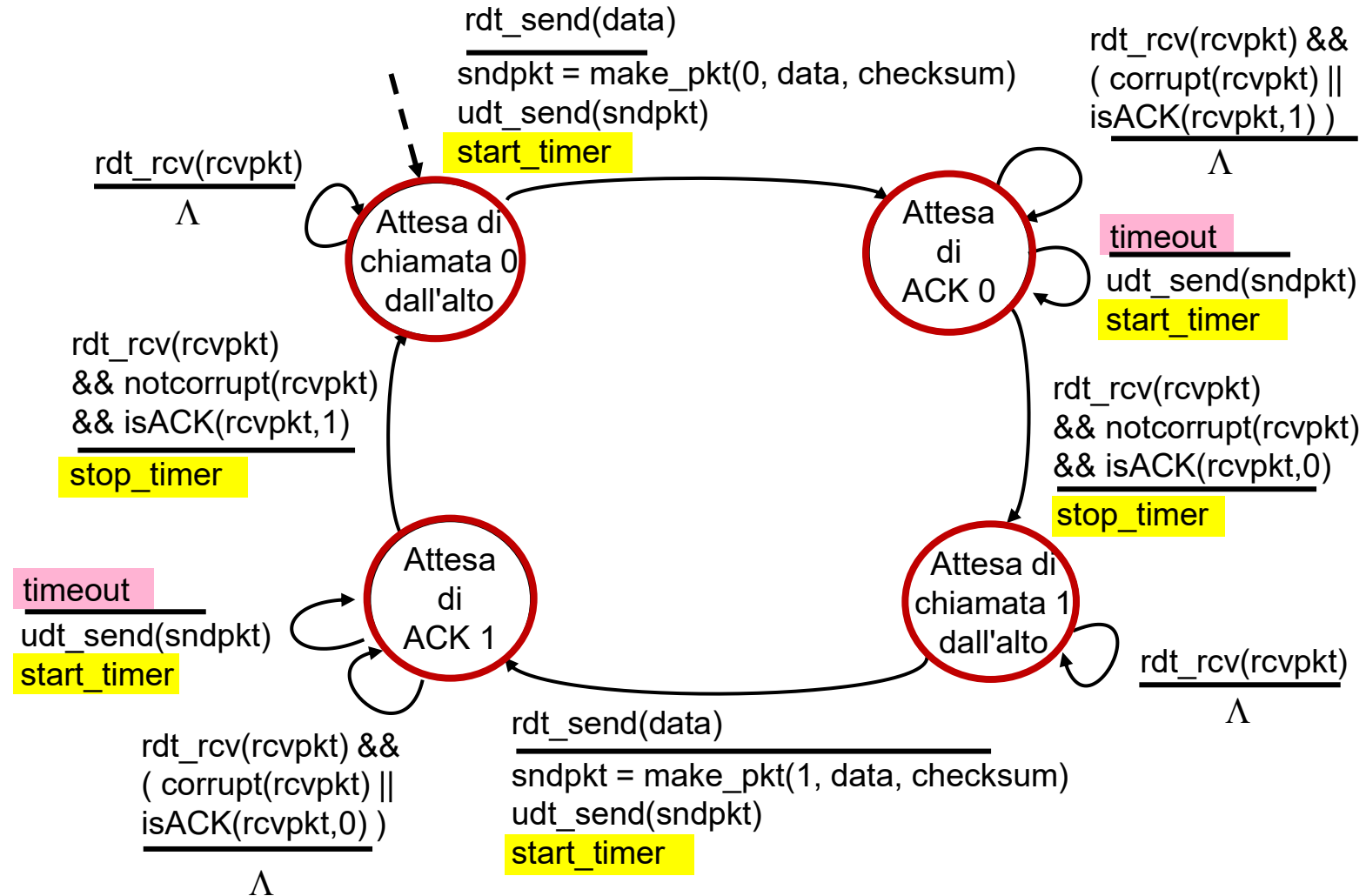


timeout

rdt3.0 mittente



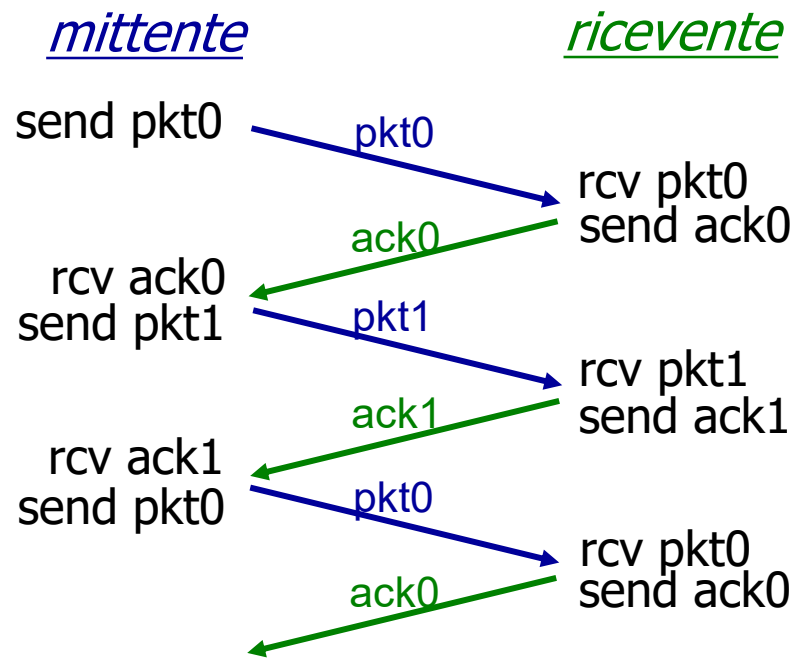
rdt3.0 mittente



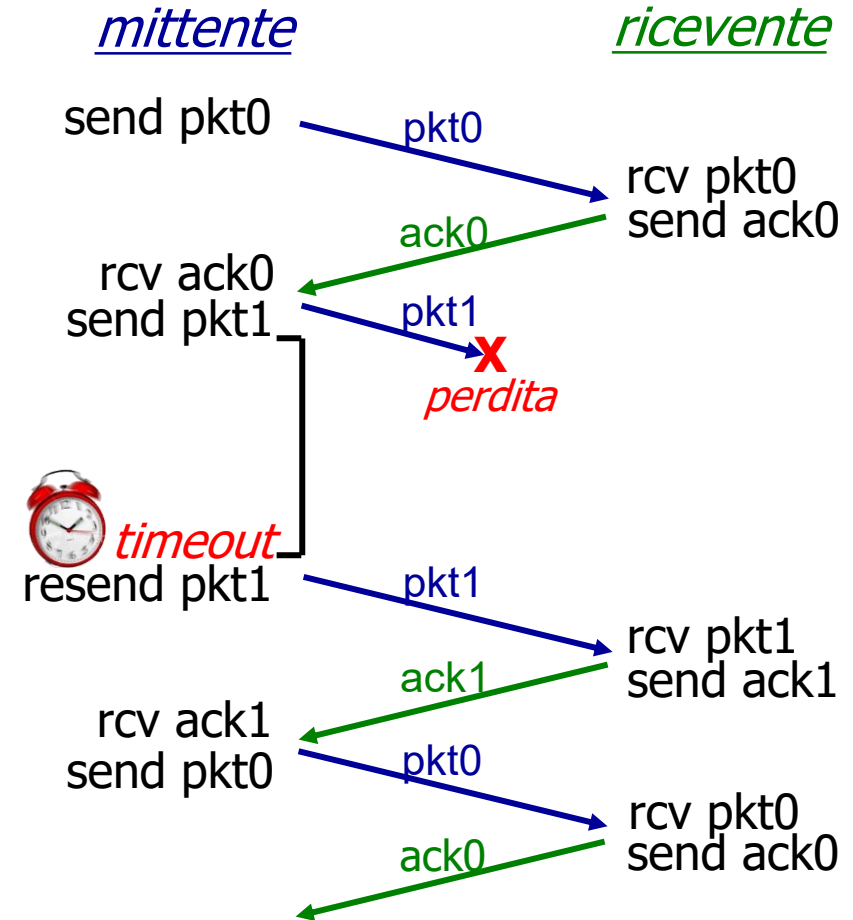
rdt3.0 ricevente

- vedi rdt 2.1

rdt3.0 in azione

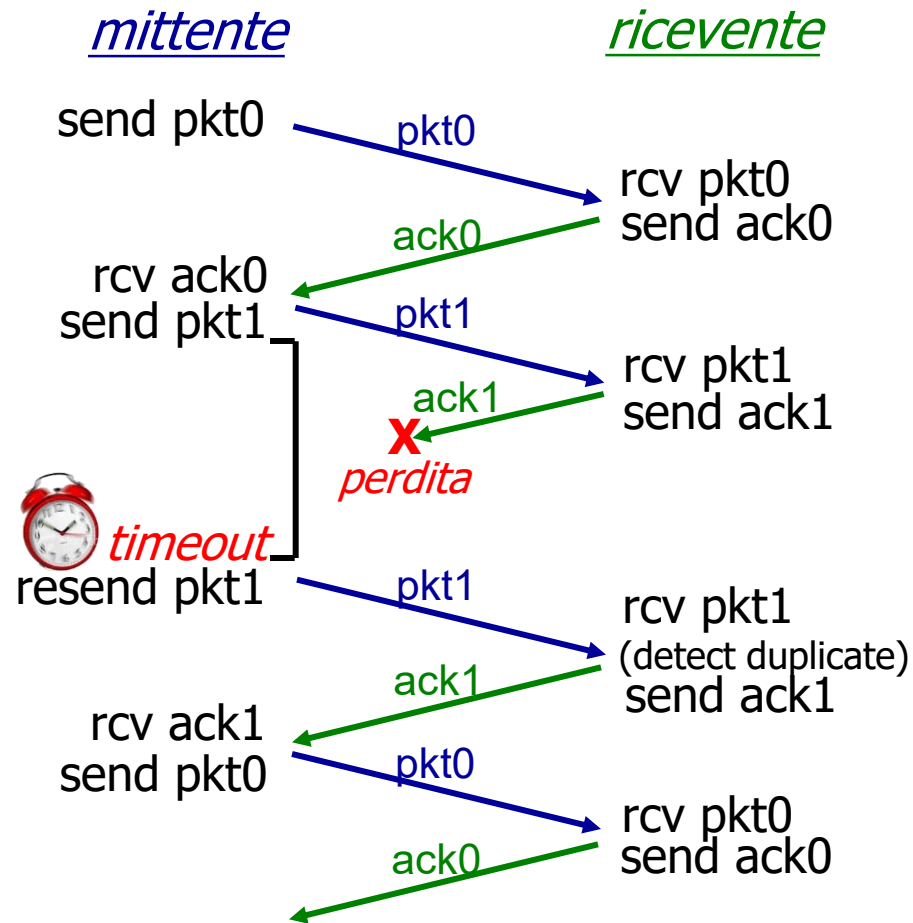


(a) operazioni senza perdita

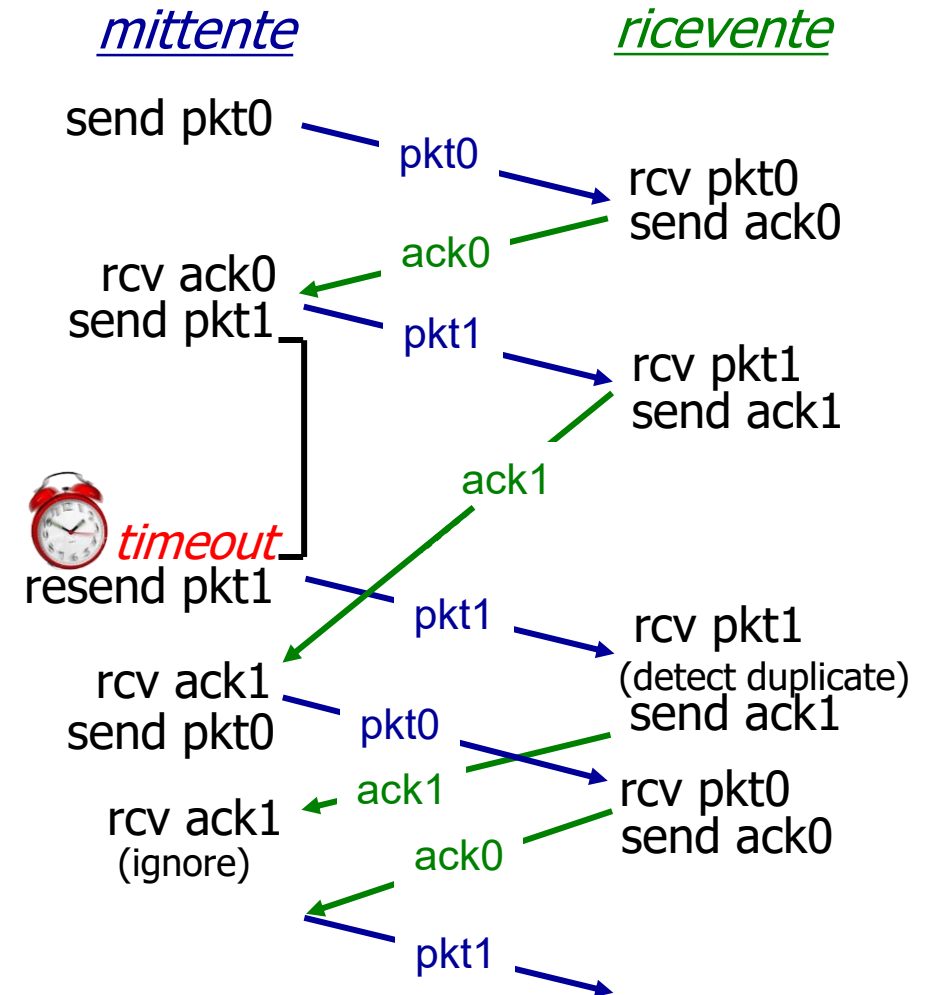


(b) perdita di pacchetto

rdt3.0 in azione



(c) perdita di ACK



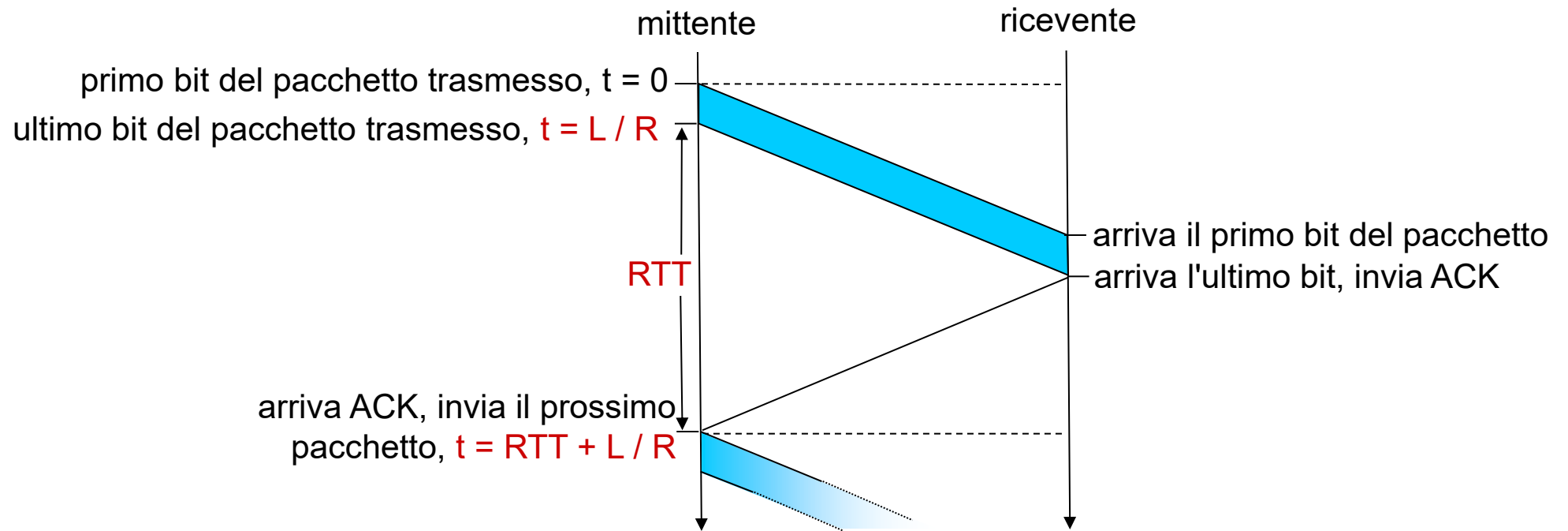
(d) timeout prematuro / ACK ritardato

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:

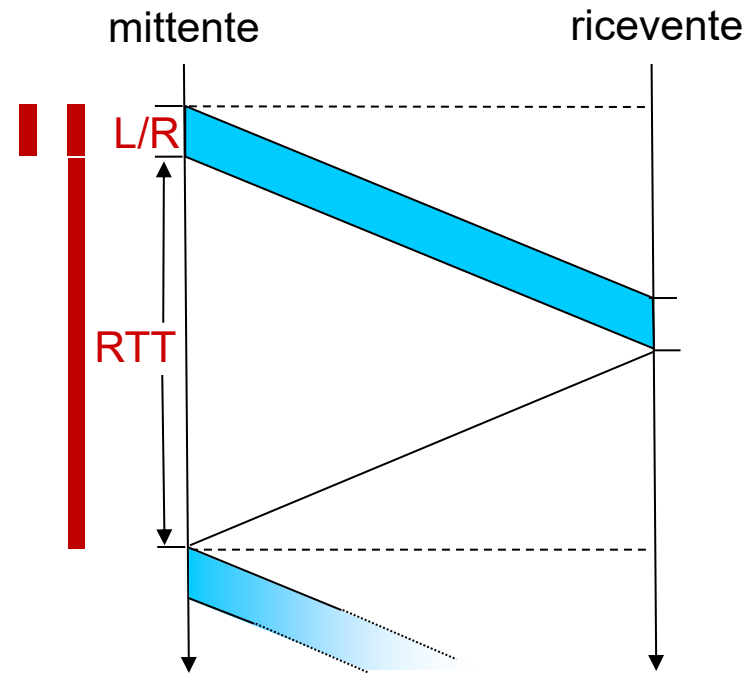
$$D_{trasm} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \mu s$$

rdt3.0: funzionamento con stop-and-wait



rdt3.0: funzionamento con stop-and-wait

$$\begin{aligned}U_{\text{mittente}} &= \frac{L / R}{RTT + L / R} \\&= \frac{.008}{30.008} \\&= 0.000267\end{aligned}$$

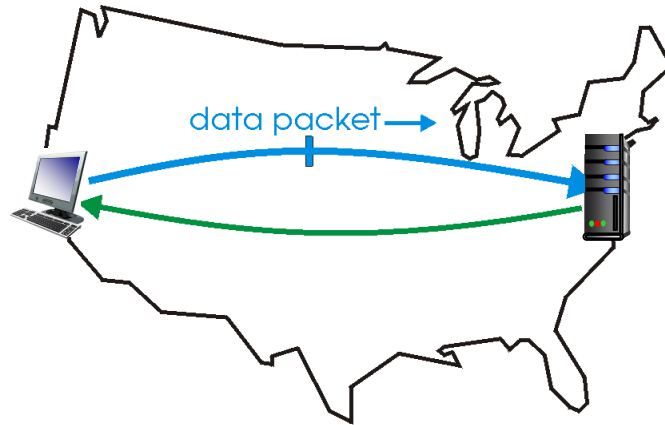


- il throughput effettivo generato dal mittente:
 $L / (RTT + L/R) = U_{\text{mittente}} \cdot R = 267 \text{ kbps}$
- le prestazioni del protocollo rdt 3.0 fanno schifo!
- il protocollo limita le prestazioni dell'infrastruttura sottostante (canale)

rdt3.0: funzionamento con pipeline

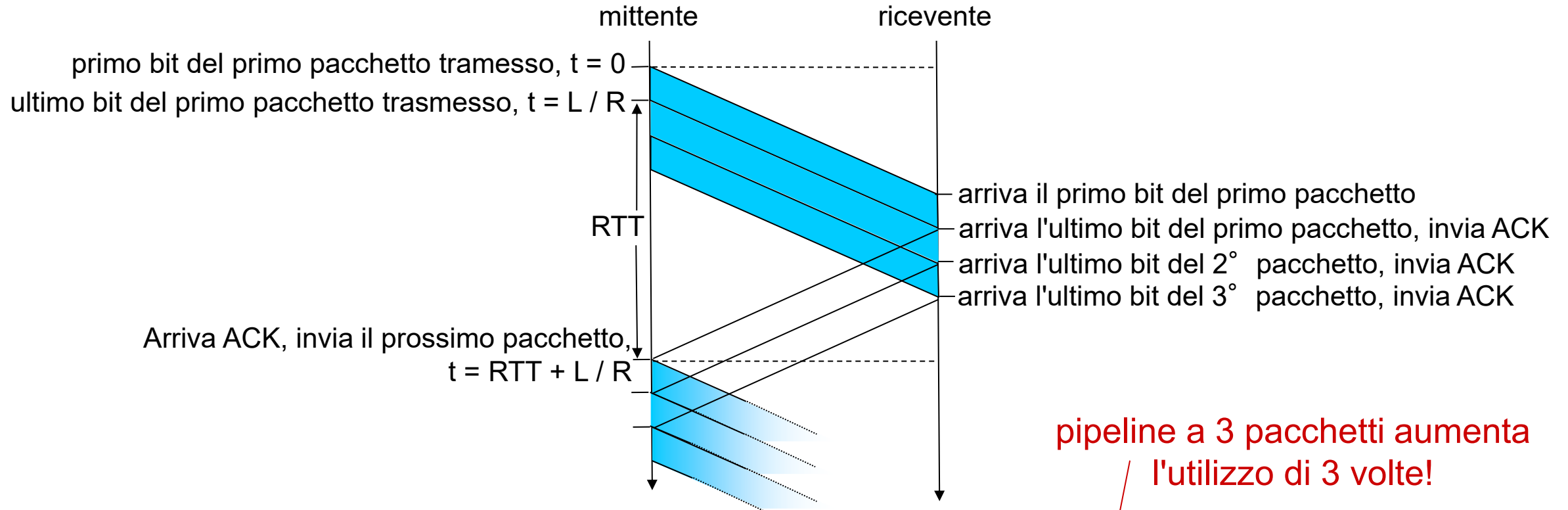
pipelining: il mittente ammette più pacchetti in transito, ancora da notificare

- 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

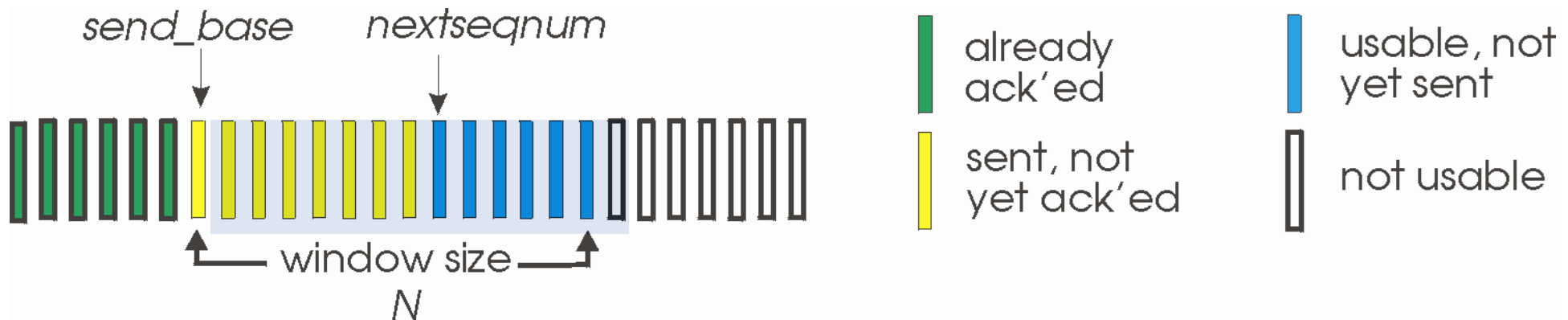


pipeline a 3 pacchetti aumenta
l'utilizzo di 3 volte!

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

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

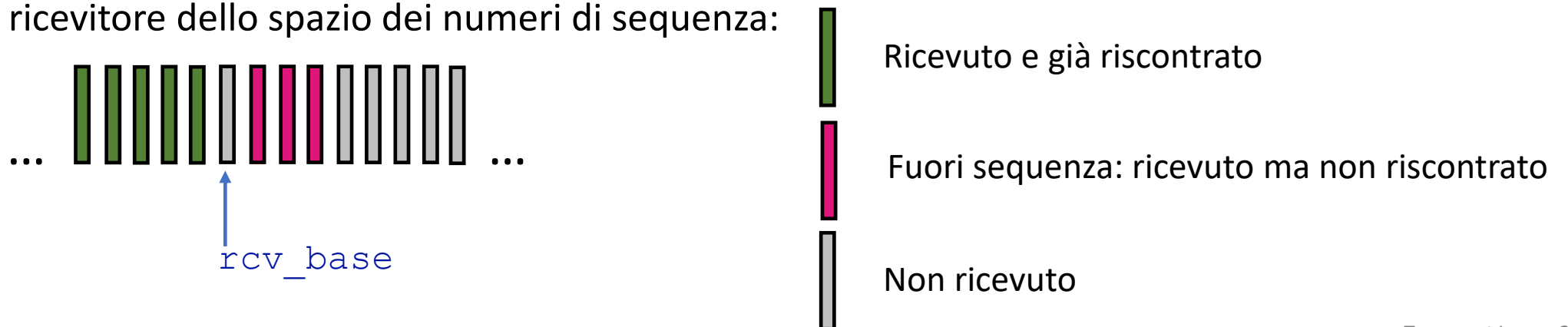


- **Riscontro cumulativo:** $ACK(n)$: riscontro di tutti i pacchetti con numero di sequenza minore o uguale a n
 - alla ricezione dell' $ACK(n)$: sposta la finestra in avanti per iniziare da $n+1$
- timer per il pacchetto più vecchio in transito
- $timeout(n)$: ritrasmette il pacchetto n e tutti i pacchetti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

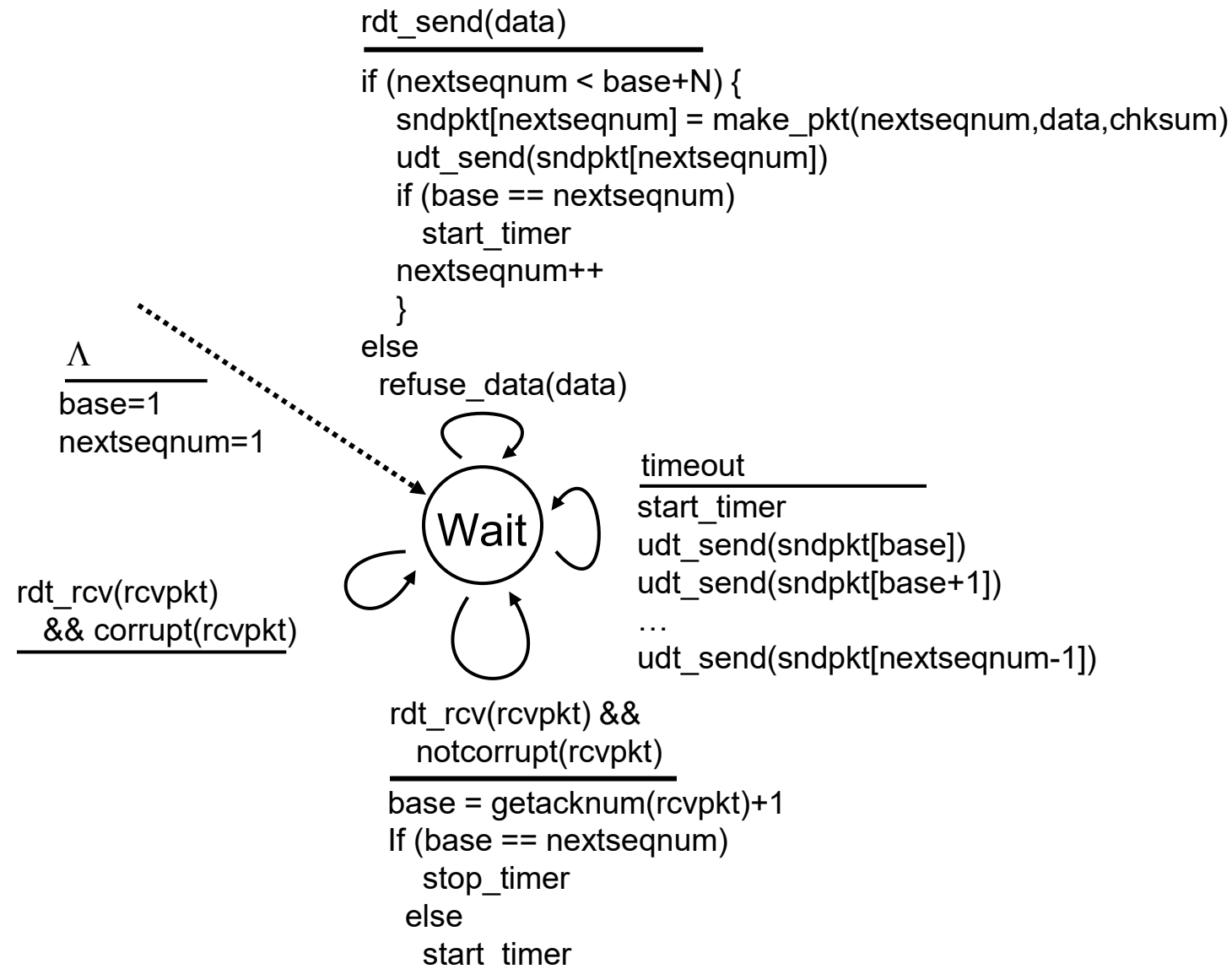
Protocolli con pipeline: Go-Back-N (ricevente)

- 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: la implementazione che discuteremo scarta i pacchetti fuori sequenza e deve solo gestire la variabile `rcv_base`.
 - rimanda un ACK per il pacchetto con il numero di sequenza più alto in sequenza

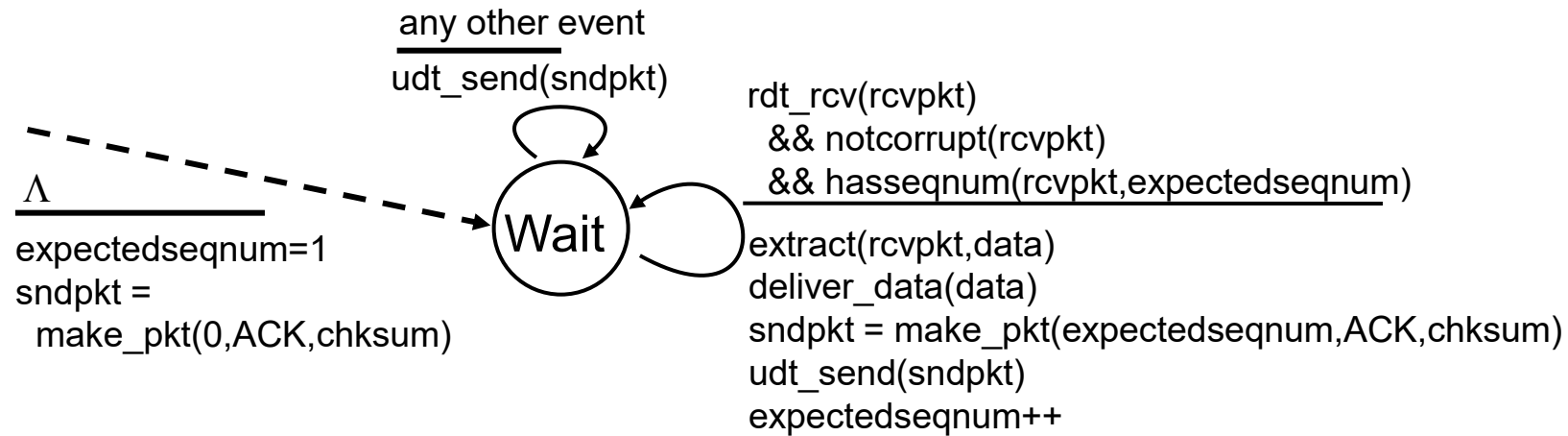
Vista del ricevitore dello spazio dei numeri di sequenza:



Go-Back-N: 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

finestra (N=4)

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

mittente

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

ignore duplicate ACK



pkt 2 timeout

send pkt2
send pkt3
send pkt4
send pkt5

ricevente

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, discard,
(re)send ack1

receive pkt4, discard,
(re)send ack1

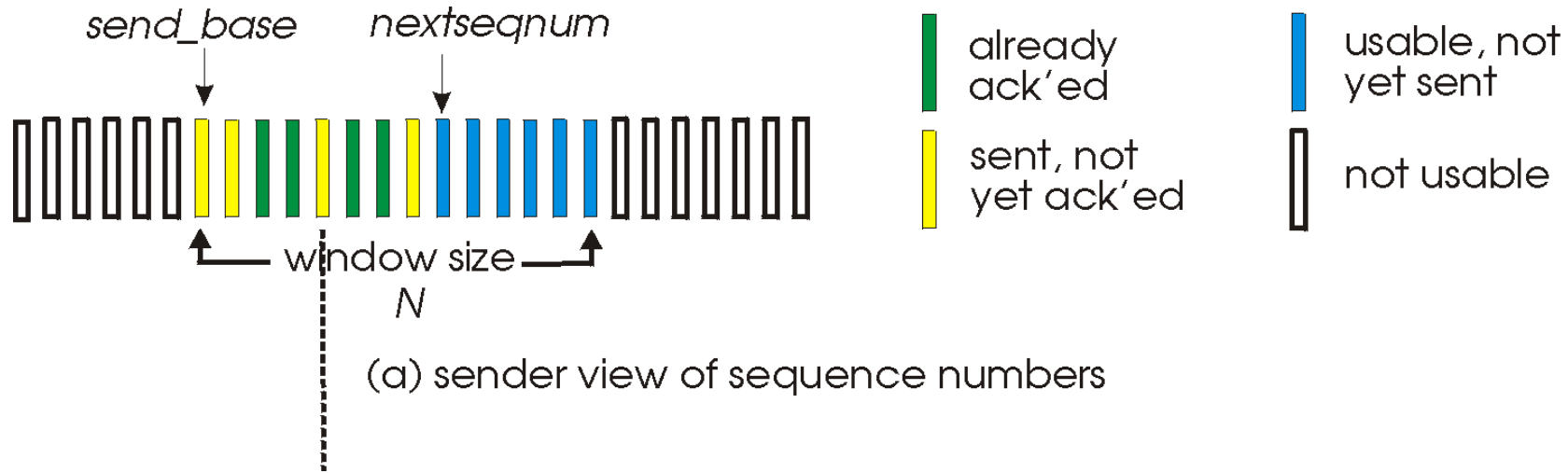
receive pkt5, discard,
(re)send ack1

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

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 $[\text{sendbase}, \text{sendbase}+N-1]$:

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

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

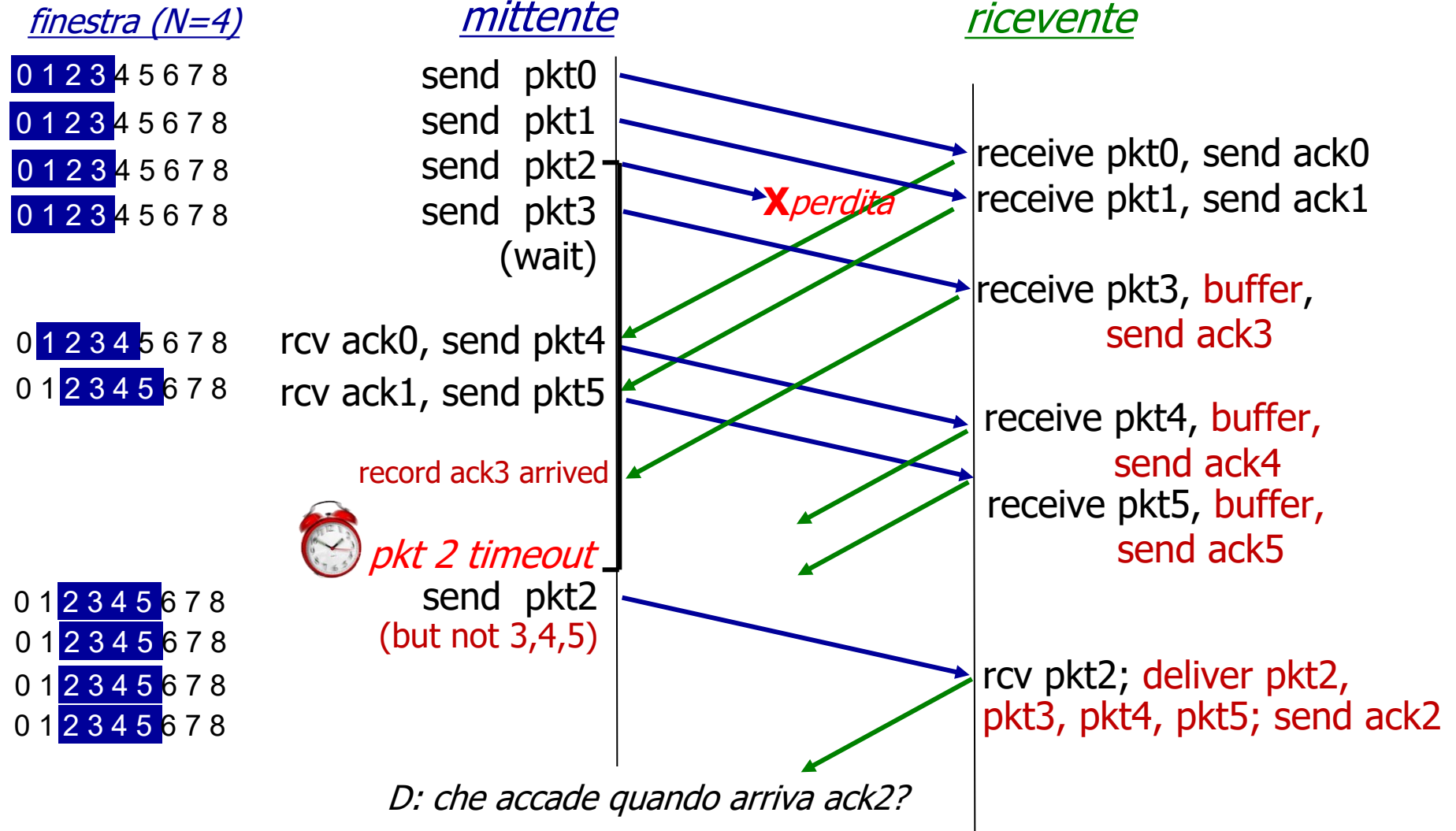
- ACK(n)

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

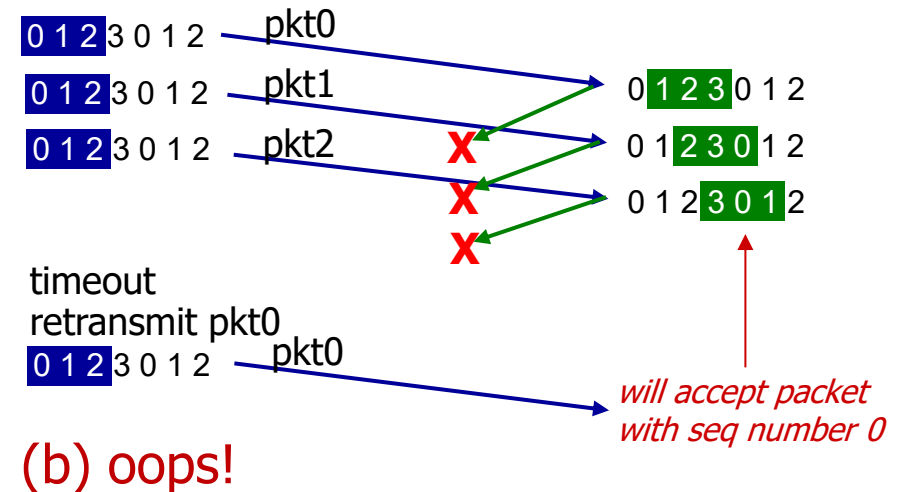
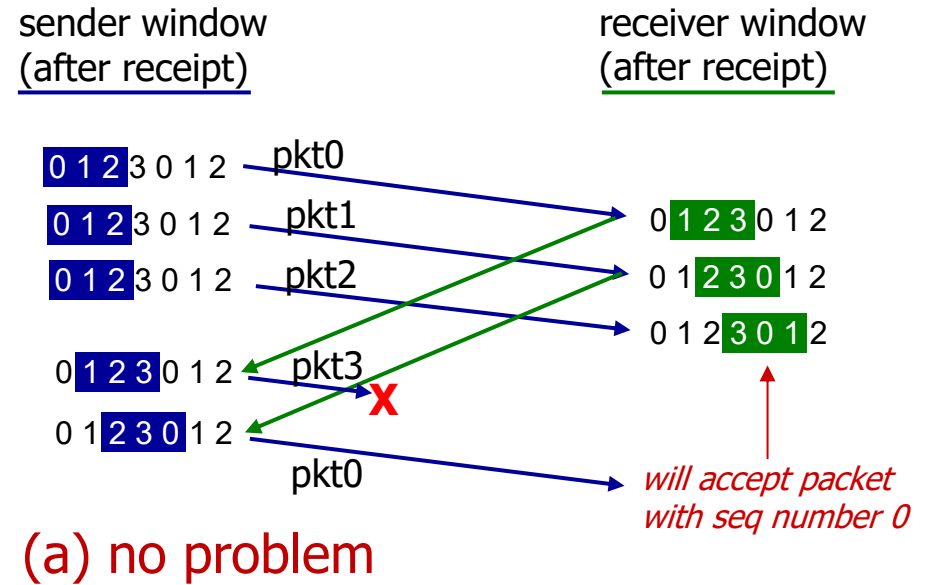


Selective repeat: dilemma!

esempio:

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

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



Selective repeat: dilemma!

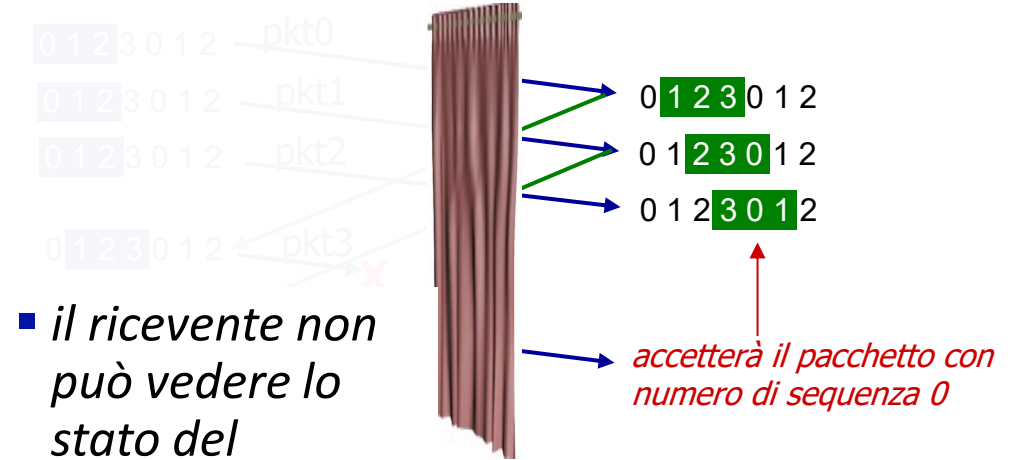
esempio:

- 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)?

Finestra del mittente
(dopo la ricezione)

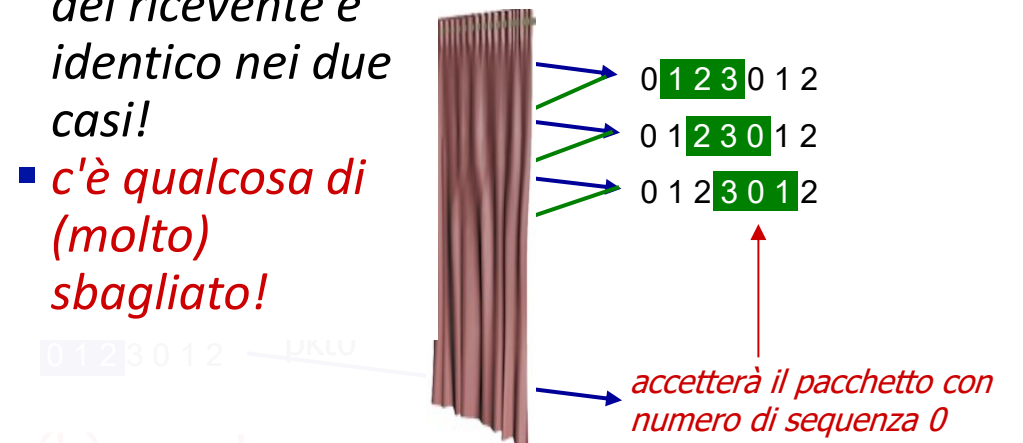
finestra del ricevente
(dopo la ricezione)



- *il ricevente non può vedere lo stato del mittente*

■ //
comportamento del ricevente è identico nei due casi!

- *c'è qualcosa di (molto) sbagliato!*



(b) oops!

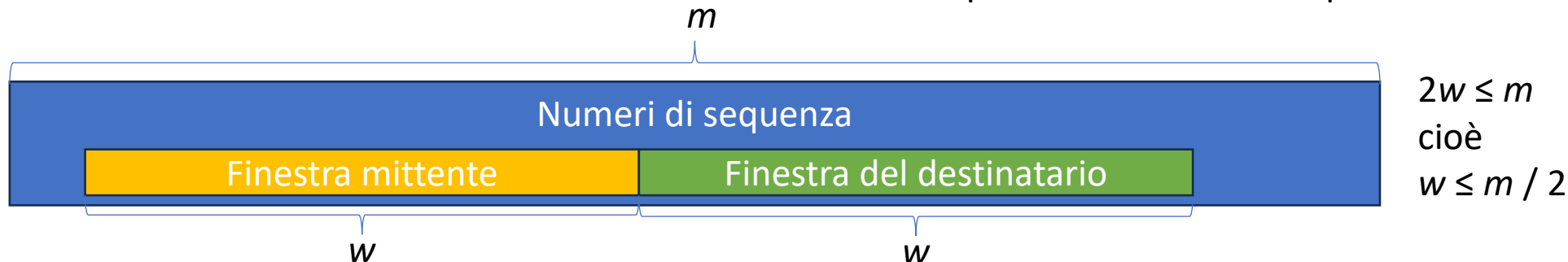
Relazione tra dimensione della finestra e dei numeri di sequenza (sia SR sia GBN)

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.

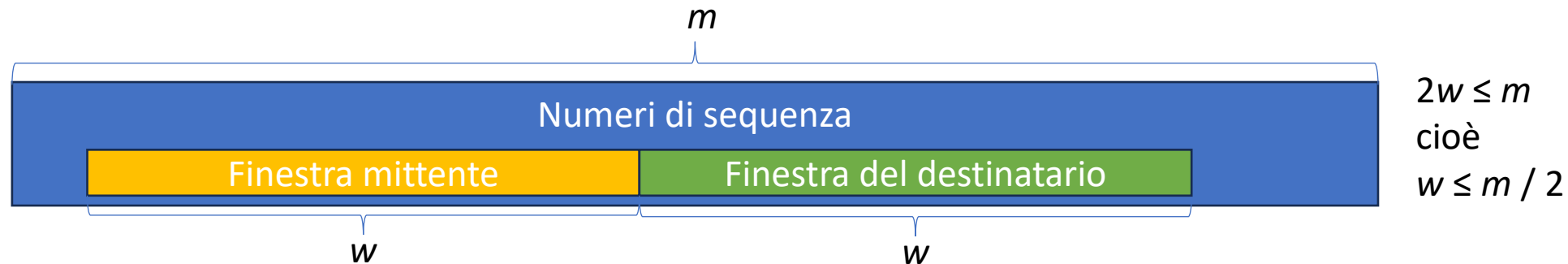
È necessario che lo spazio dei numeri di sequenza sia sufficientemente grande da contenere sia la finestra del mittente sia la finestra del destinatario senza che si sovrappongano (considerando l'operazione di modulo)

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



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)?



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