Reti di calcolatori e Internet: Un approccio top-down

7ª edizione
Jim Kurose, Keith Ross

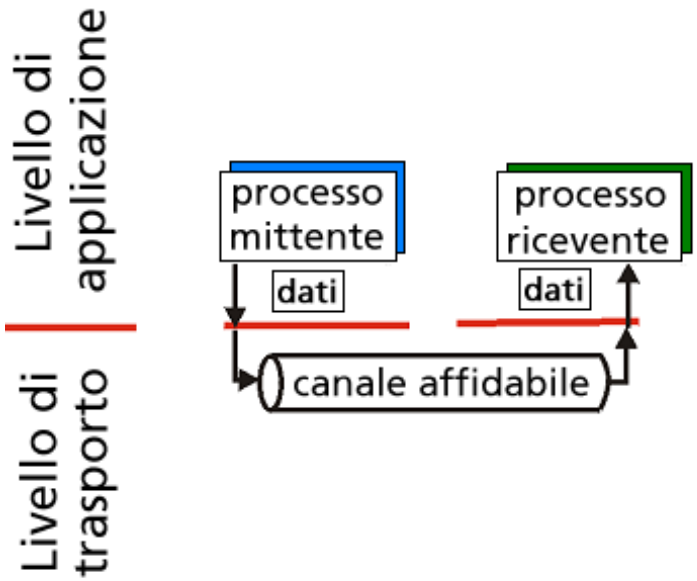
Pearson Paravia Bruno Mondadori Spa

Capitolo 3: Livello di trasporto

- 3.1 Servizi a livello di trasporto
- 3.2 Multiplexing e demultiplexing
- 3.3 Trasporto senza connessione: UDP
- 3.4 Principi del trasferimento dati affidabile
- 3.5 Trasporto orientato alla connessione: TCP
  - struttura dei segmenti
  - trasferimento dati affidabile
  - controllo di flusso
  - gestione della connessione
- 3.6 Principi del controllo di congestione
- 3.7 Controllo di congestione TCP

Principi del trasferimento dati affidabile

- Importante nei livelli di applicazione, trasporto e collegamento
- Tra i dieci problemi più importanti del networking!

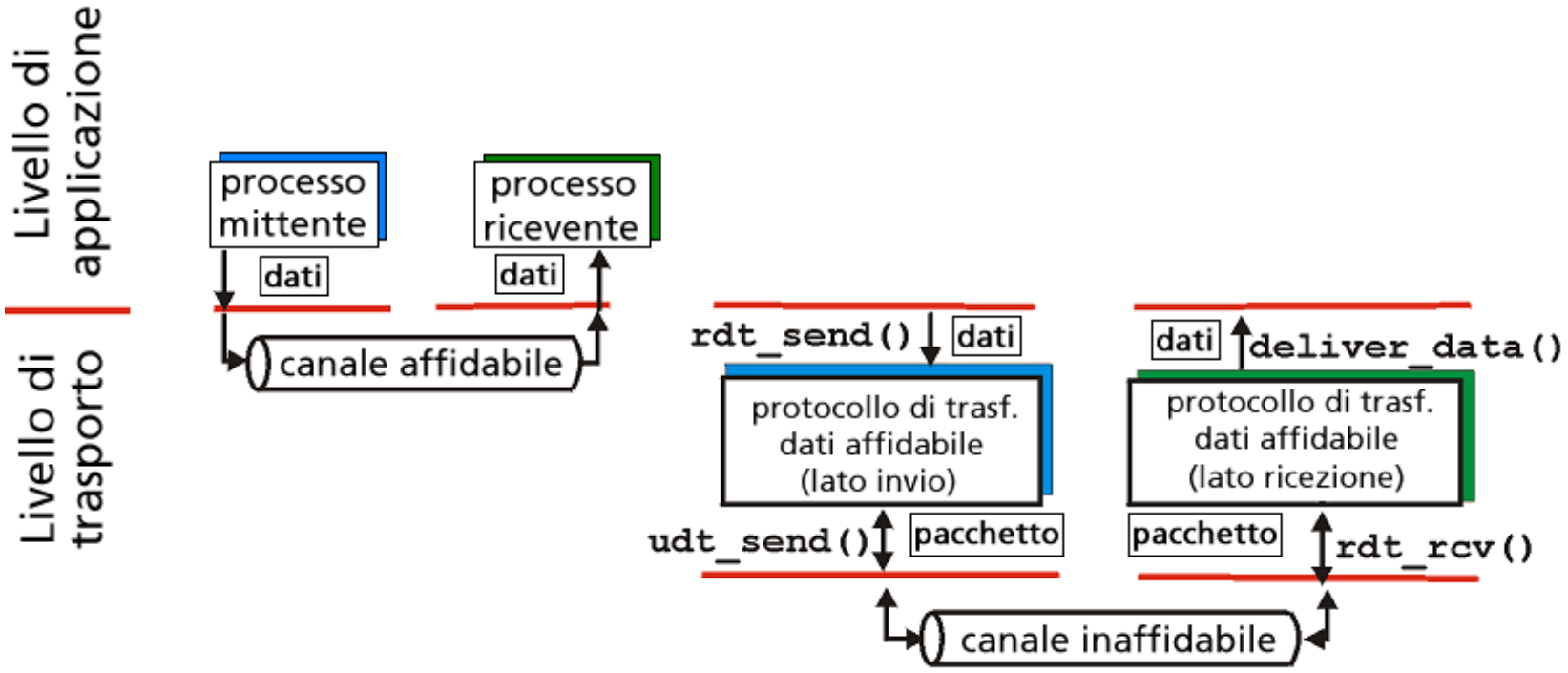


a) servizio offerto

- Le caratteristiche del canale inaffidabile determinano la complessità del protocollo di trasferimento dati affidabile (reliable data transfer o rdt)

Principi del trasferimento dati affidabile

- Importante nei livelli di applicazione, trasporto e collegamento
- Tra i dieci problemi più importanti del networking!

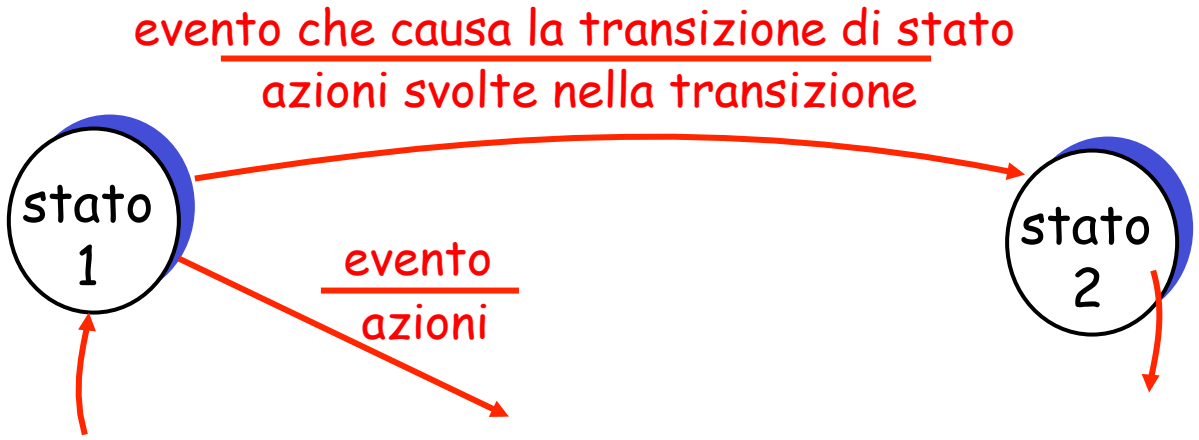


- Le caratteristiche del canale inaffidabile determinano la complessità del protocollo di trasferimento dati affidabile (reliable data transfer o rdt)

Trasferimento dati affidabile: preparazione

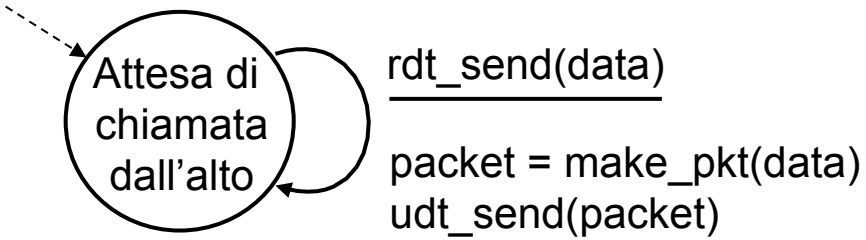
- Svilupperemo progressivamente i lati d'invio e di ricezione di un protocollo di trasferimento dati affidabile (rdt)
- Considereremo soltanto i trasferimenti dati unidirezionali
  - ma le informazioni di controllo fluiranno in entrambe le direzioni!
- Utilizzeremo automi a stati finiti per specificare il mittente e il ricevente

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

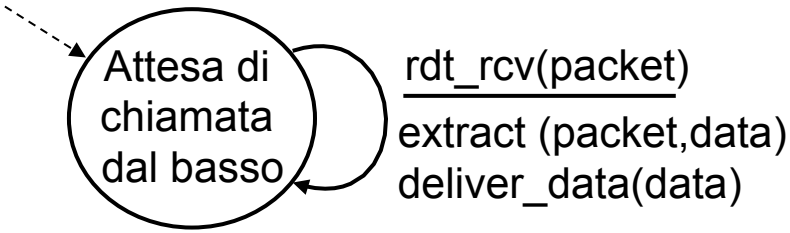


Rdt1.0: trasferimento affidabile su canale affidabile

- Canale sottostante perfettamente affidabile
  - Nessun errore nei bit
  - Nessuna perdita di pacchetti
- Automa distinto per il mittente e per il ricevente:
  - il mittente invia i dati nel canale sottostante
  - il ricevente legge i dati dal canale sottostante



mittente

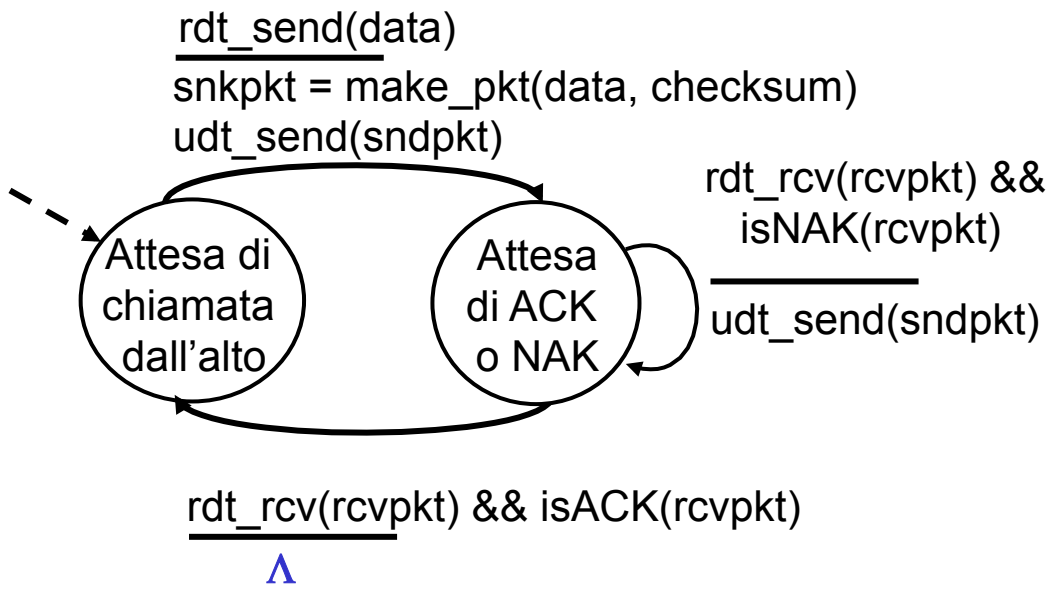


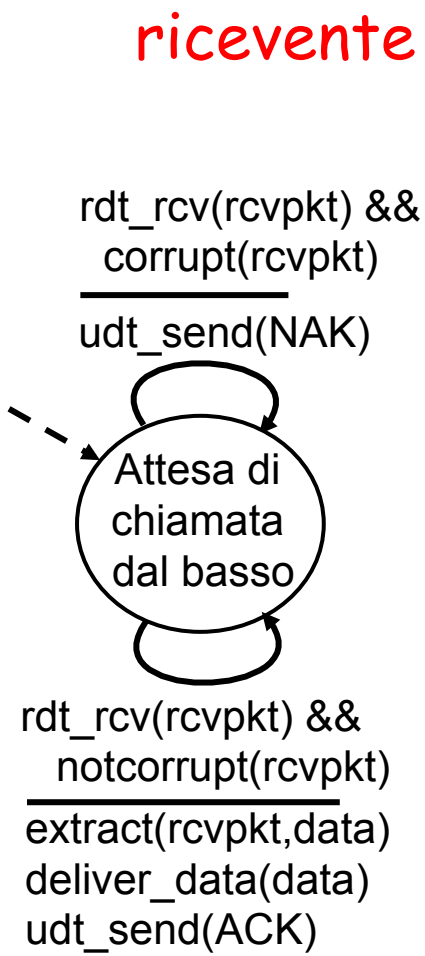
ricevente

Rdt2.0: canale con errori nei bit

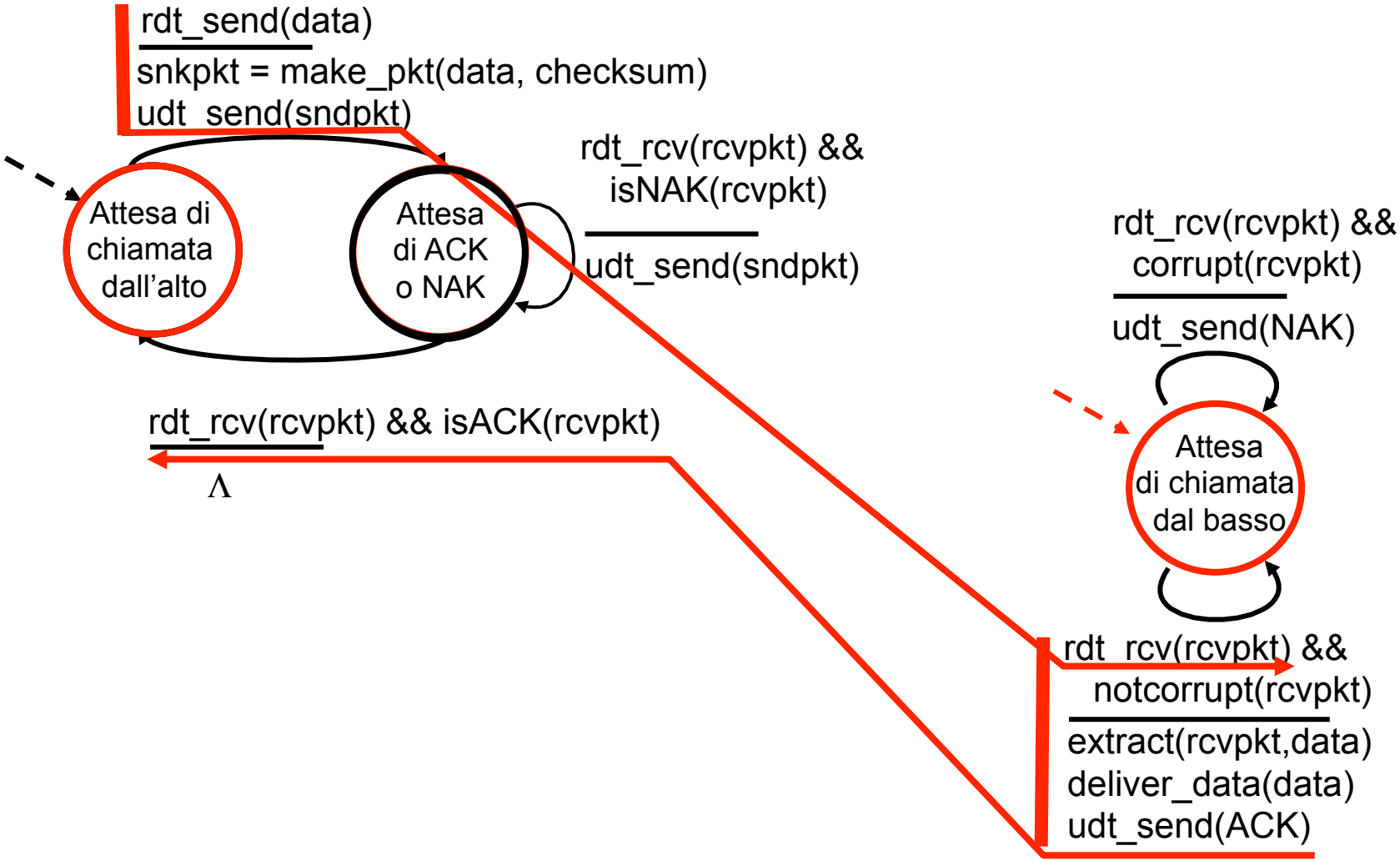
- Il canale sottostante potrebbe confondere i bit nei pacchetti, ma i pacchetti non vengono persi e sono ricevuti in ordine
  - checksum per rilevare gli errori nei bit
- *domanda*: come correggere gli errori:
  - *notifica positiva (ACK):* il ricevente comunica espressamente al mittente che il pacchetto ricevuto è corretto
  - *notifica negativa (NAK):* il ricevente comunica espressamente al mittente che il pacchetto contiene errori
  - il mittente ritrasmette il pacchetto se riceve un NAK
- nuovi meccanismi in **`rdt2.0`** (oltre a **`rdt1.0`**)
  [ARQ = Automatic Repeat reQuest]
  - rilevamento di errore
  - feedback del destinatario: messaggi di controllo (ACK, NAK) ricevente->mittente
  - ritrasmissione

rdt2.0: specifica dell'automa

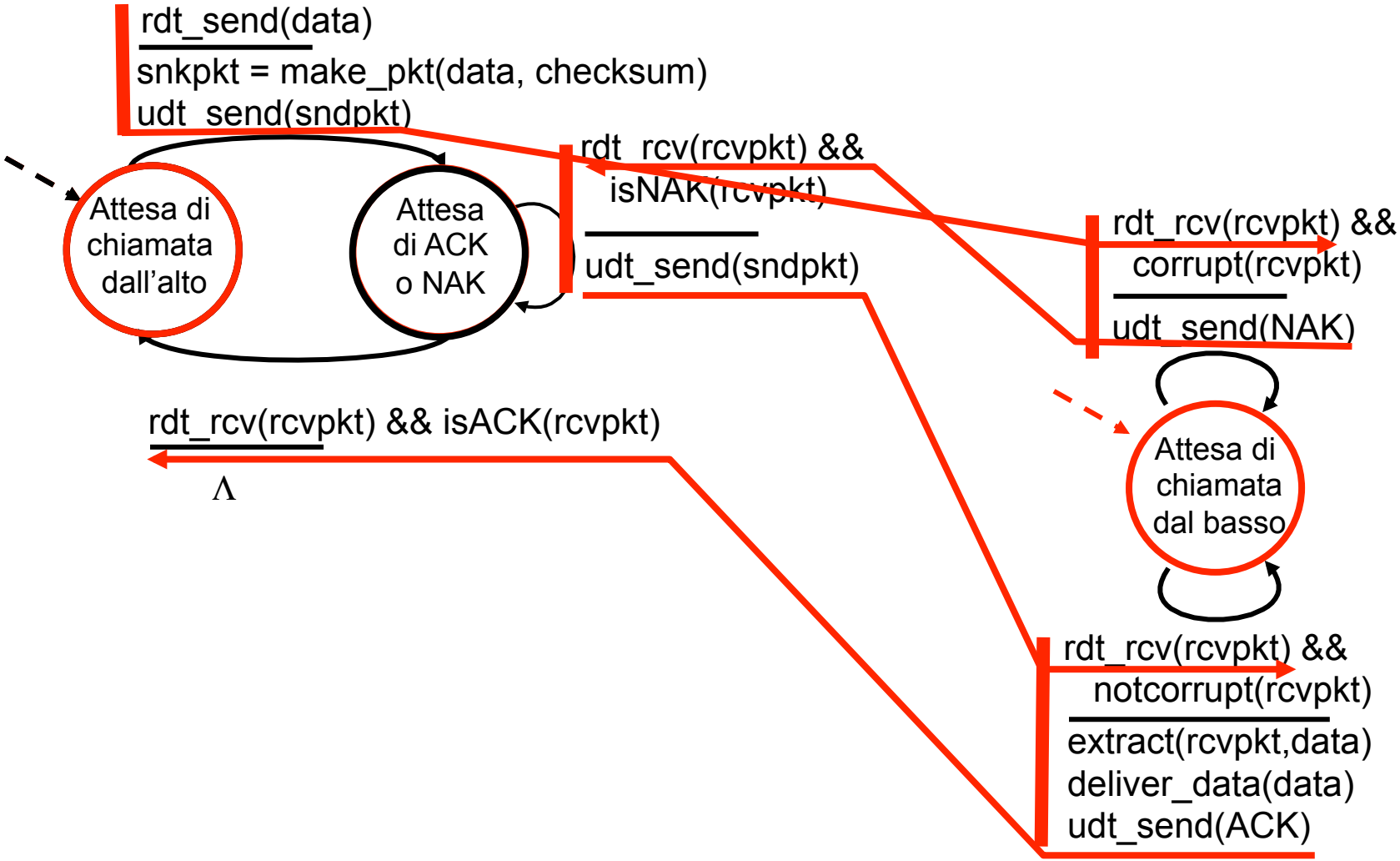




rdt2.0: operazione senza errori



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 al destinatario!
- Non basta ritrasmettere: possibili duplicati

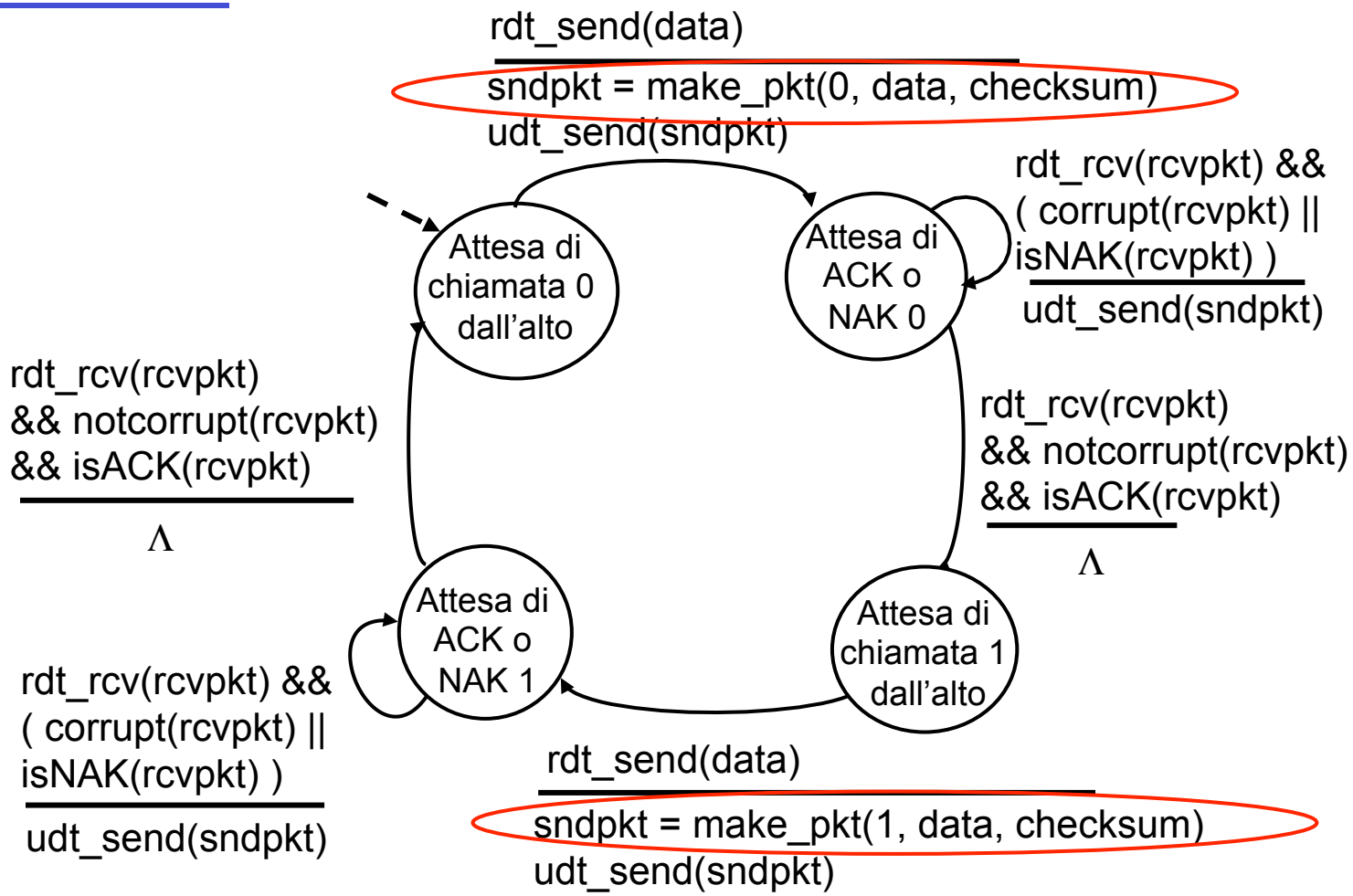
Gestione dei duplicati:

- Il mittente ritrasmette il pacchetto corrente se ACK/NAK è alterato
- Il mittente aggiunge un *numero di sequenza* a ogni pacchetto
- Il ricevente scarta il pacchetto duplicato

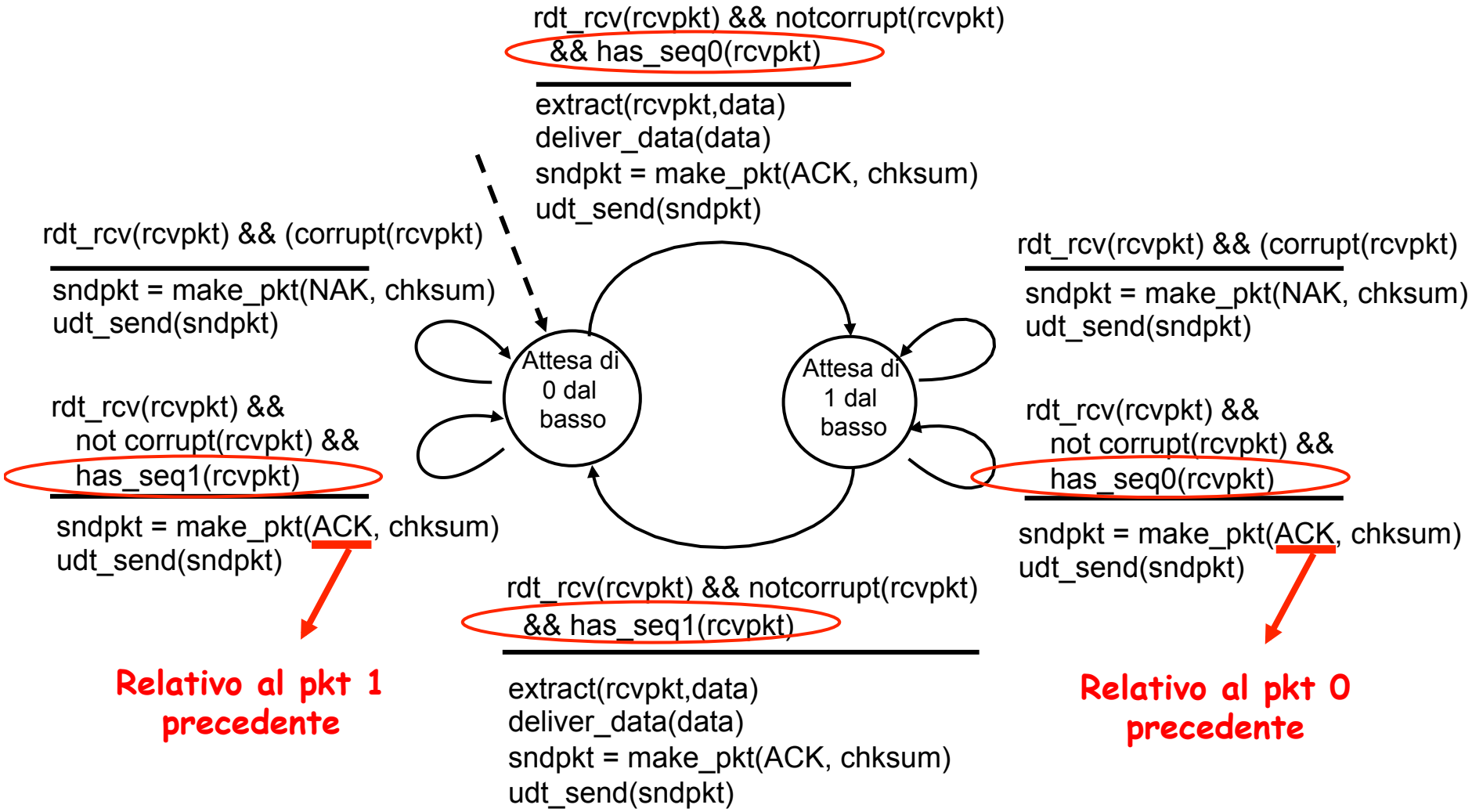
stop and wait

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: discussione

Mittente:

- Aggiunge il numero di sequenza al pacchetto
- Saranno sufficienti due numeri di sequenza (0,1)
- 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

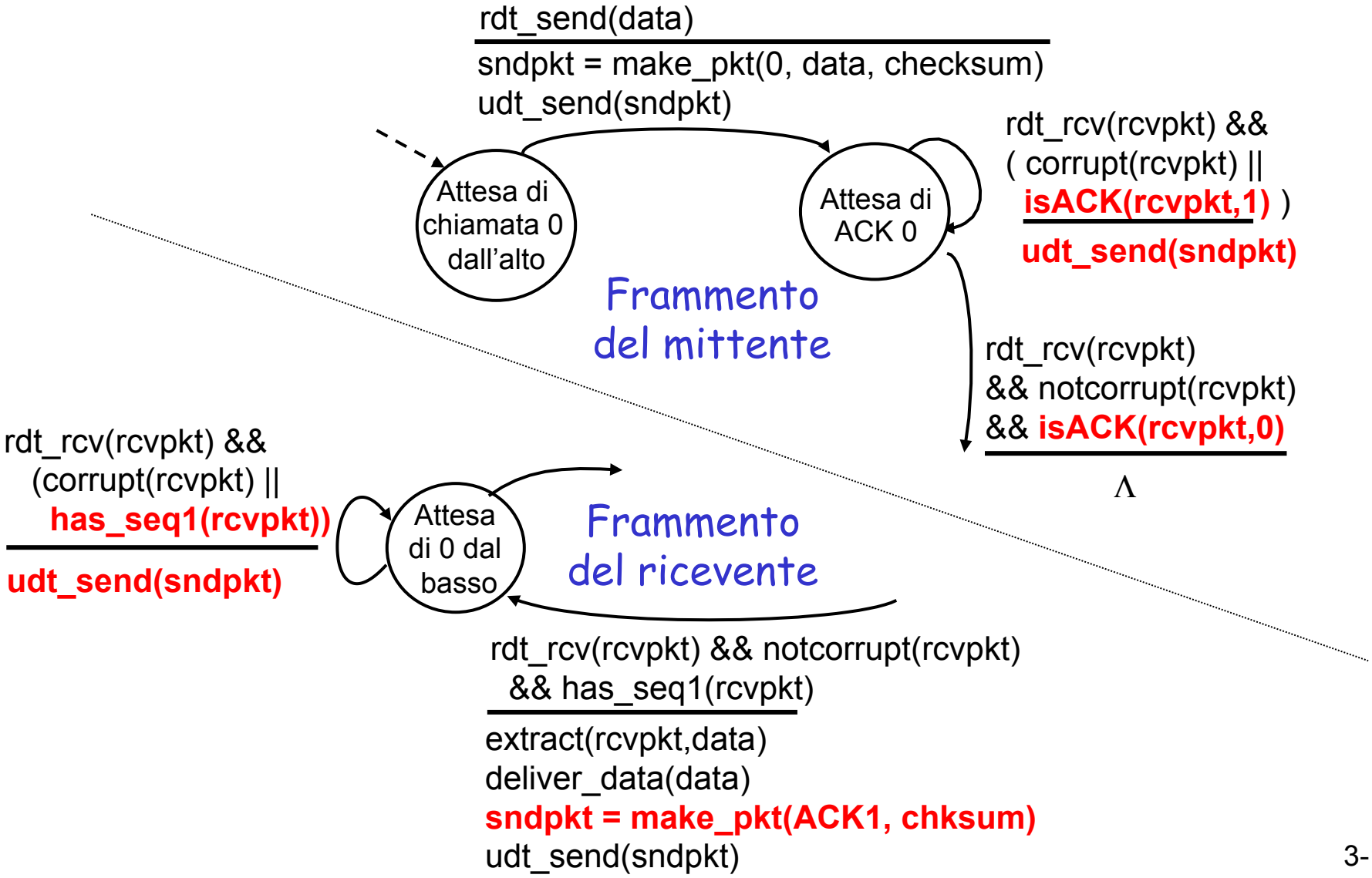
Ricevente:

- 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*

rdt2.2: frammenti del mittente e del ricevente



rdt3.0: canali con errori *e* perdite

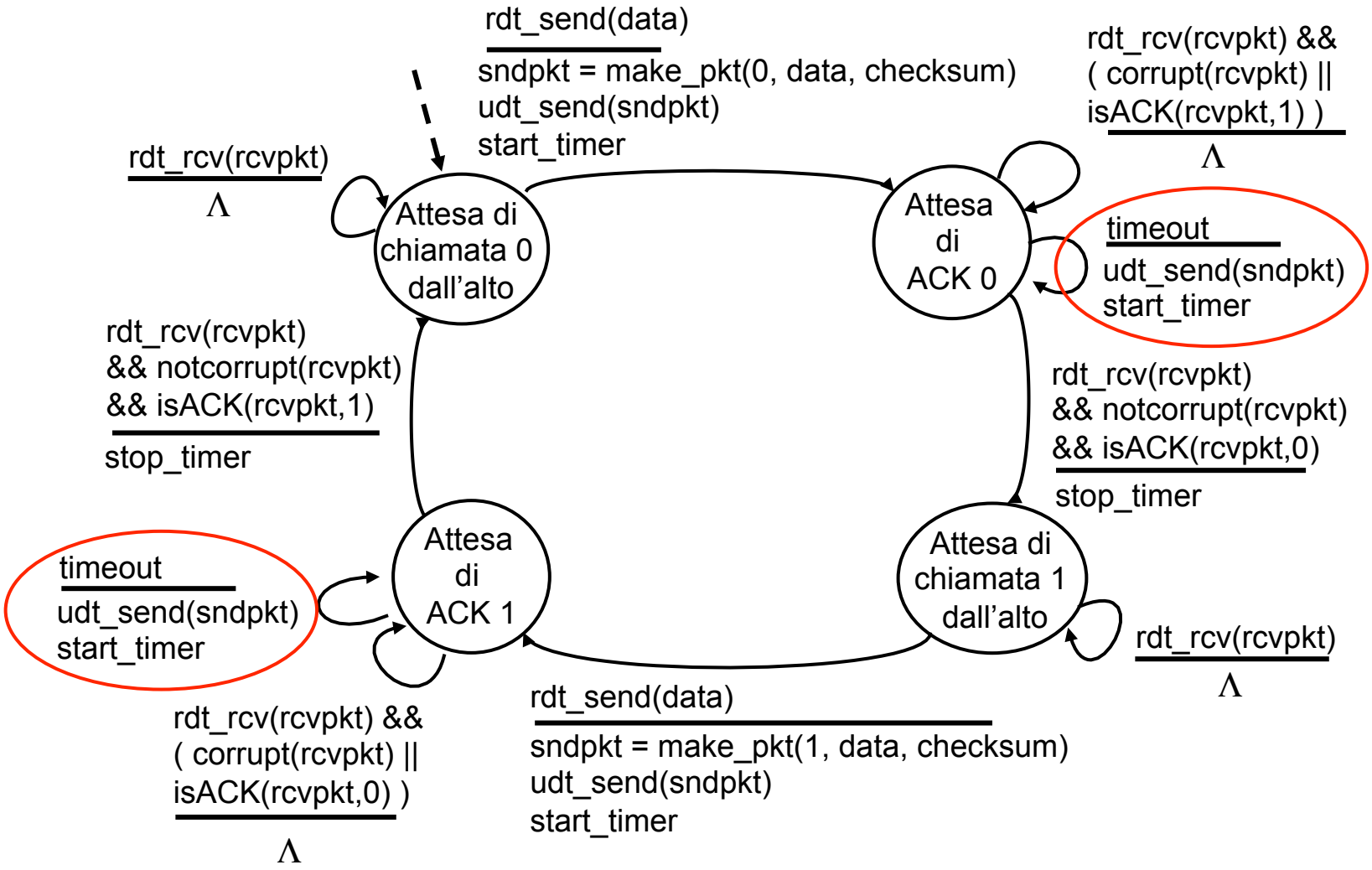
Nuova ipotesi: il canale sottostante può anche smarrire i pacchetti (dati o ACK)

- checksum, numero di sequenza, ACK e ritrasmissioni aiuteranno, ma non saranno sufficienti

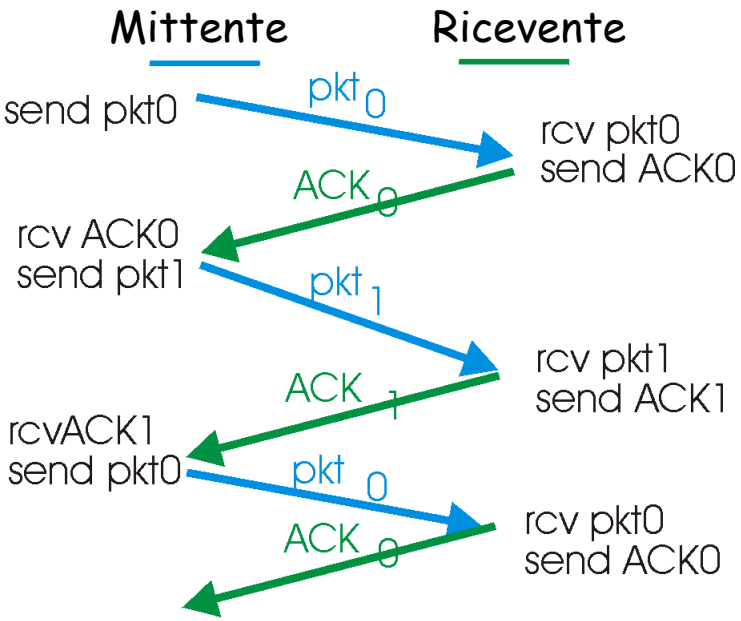
Approccio: il mittente attende un ACK per un tempo "ragionevole"

- 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
- occorre un contatore (*countdown timer*)

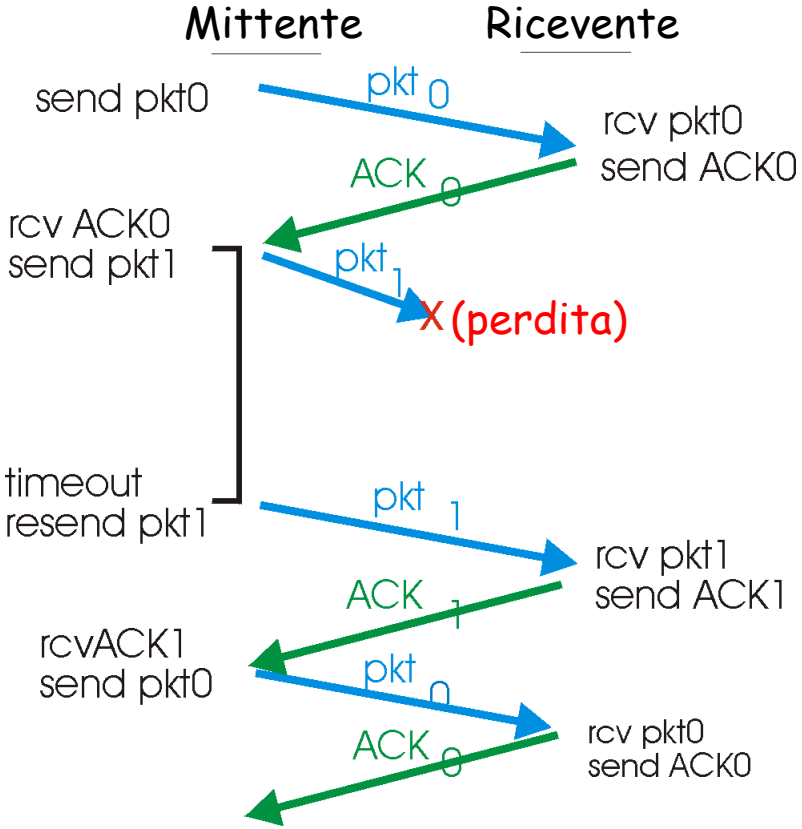
rdt3.0 mittente



rdt3.0 in azione

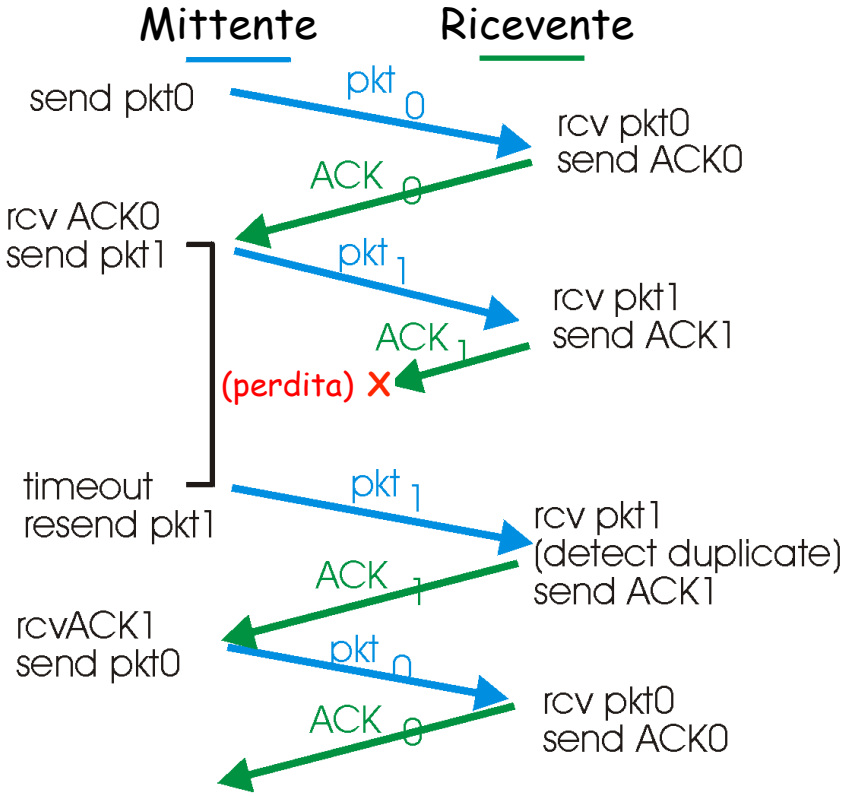


a) Operazioni senza perdite

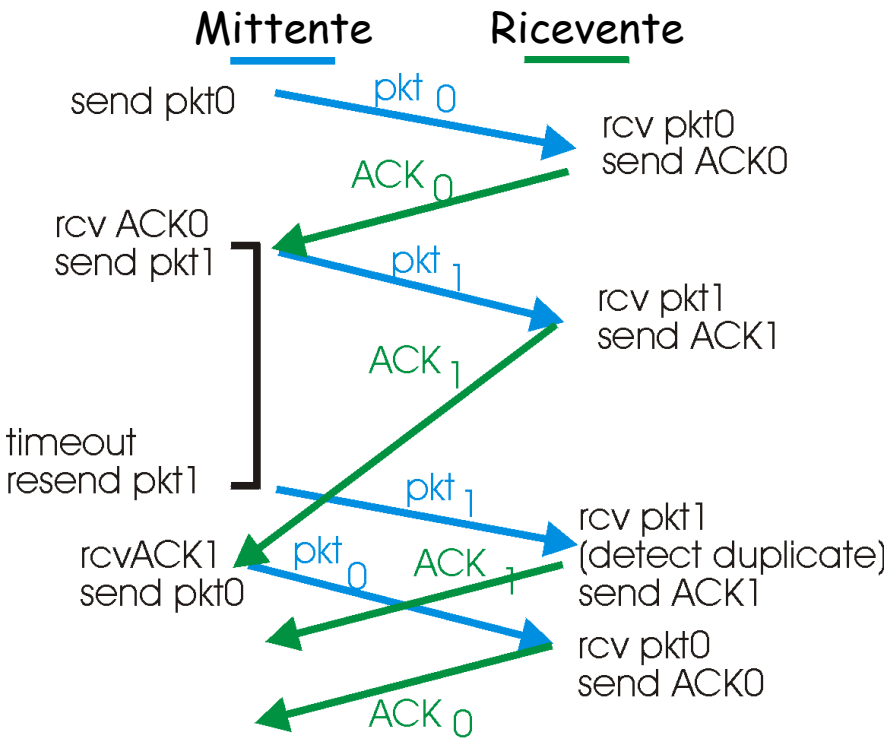


b) Perdita di pacchetto

rdt3.0 in azione



c) Perdita di ACK



d) Timeout prematuro

Prestazioni di rdt3.0

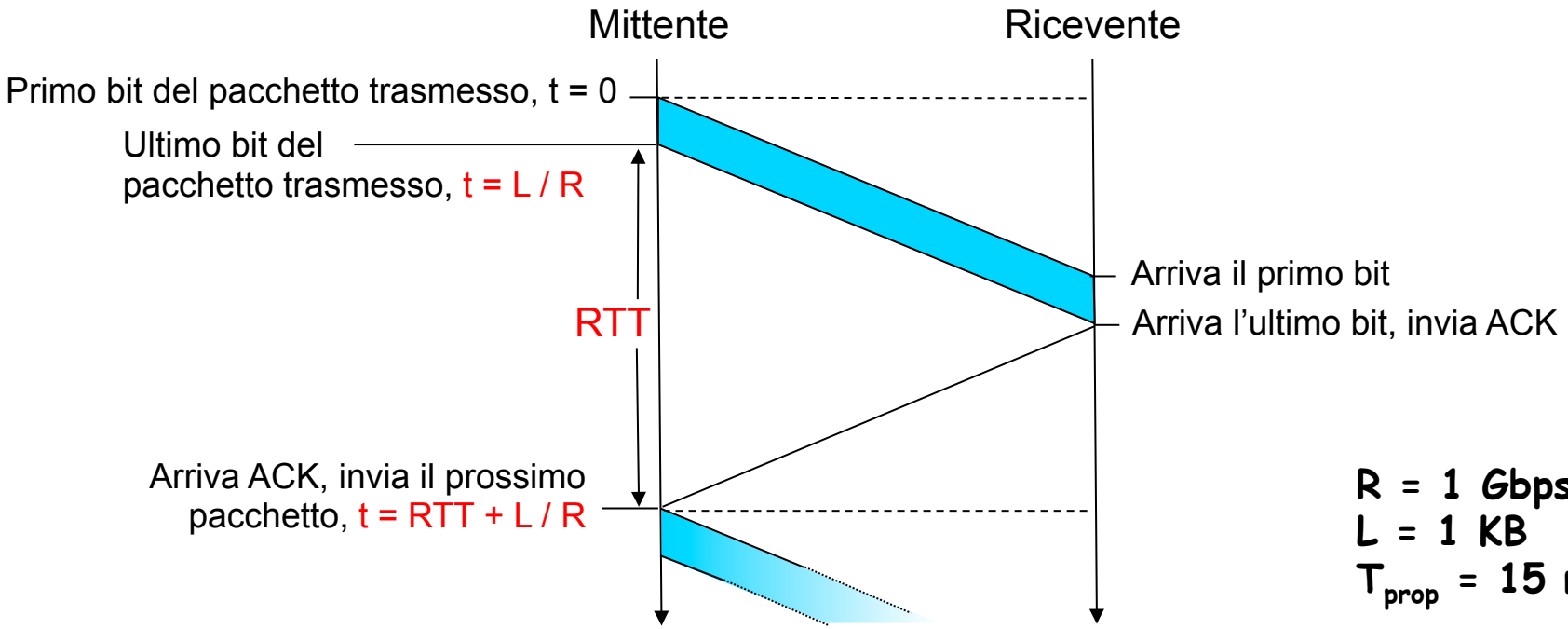
- rdt3.0 funziona, ma le prestazioni non sono apprezzabili
- esempio: collegamento da 1 Gbps, ritardo di propagazione 15 ms, pacchetti da 1 KB:

$$T_{trasm} = \frac{L \text{ (lunghezza del pacchetto in bit)}}{R \text{ (tasso trasmissivo, bps)}} = \frac{8 \text{ kb/pacc}}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

$$U_{mitt} = \frac{L / R}{RTT + L / R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

  - U_{mitt} : utilizzo è la frazione di tempo in cui il mittente è occupato nell'invio di bit
  - Un pacchetto da 1 KB ogni 30 msec -> throughput di 33 kB/sec in un collegamento da 1 Gbps
  - Il protocollo di rete limita l'uso delle risorse fisiche!

rdt3.0: funzionamento con stop-and-wait



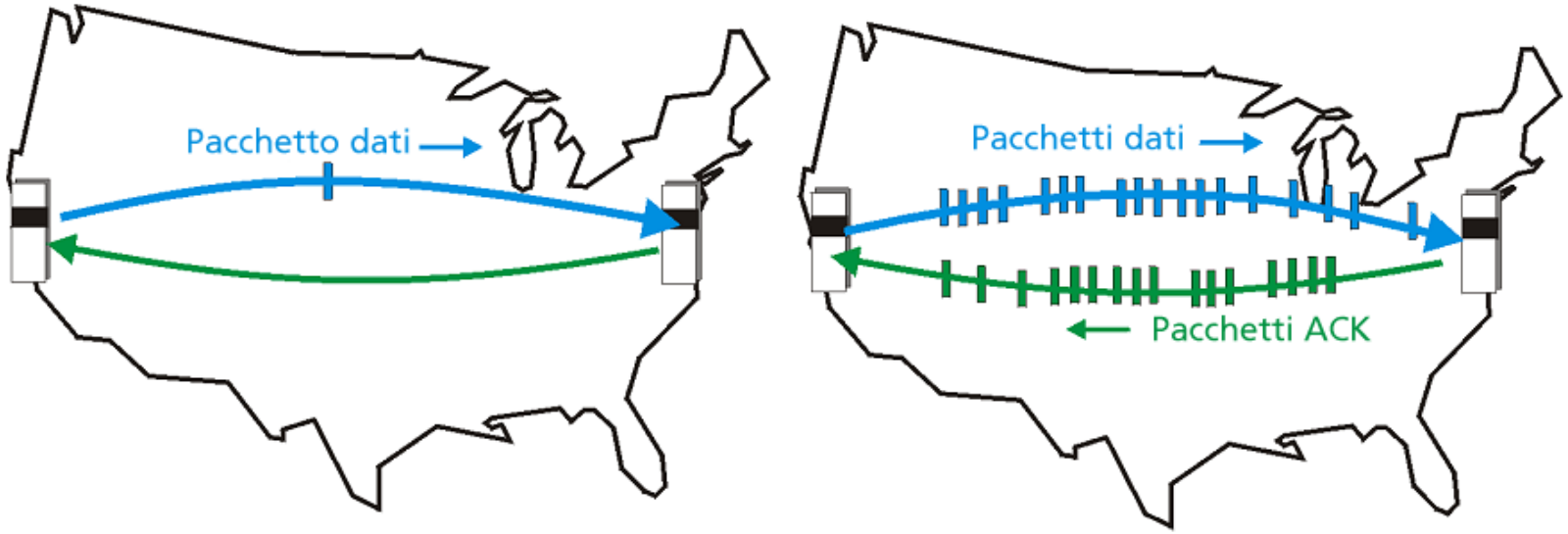
R = 1 Gbps
L = 1 KB
T_{prop} = 15 ms

$$\text{Percentuale di utilizzo del canale} = \frac{L / R}{RTT + L / R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

Protocolli 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

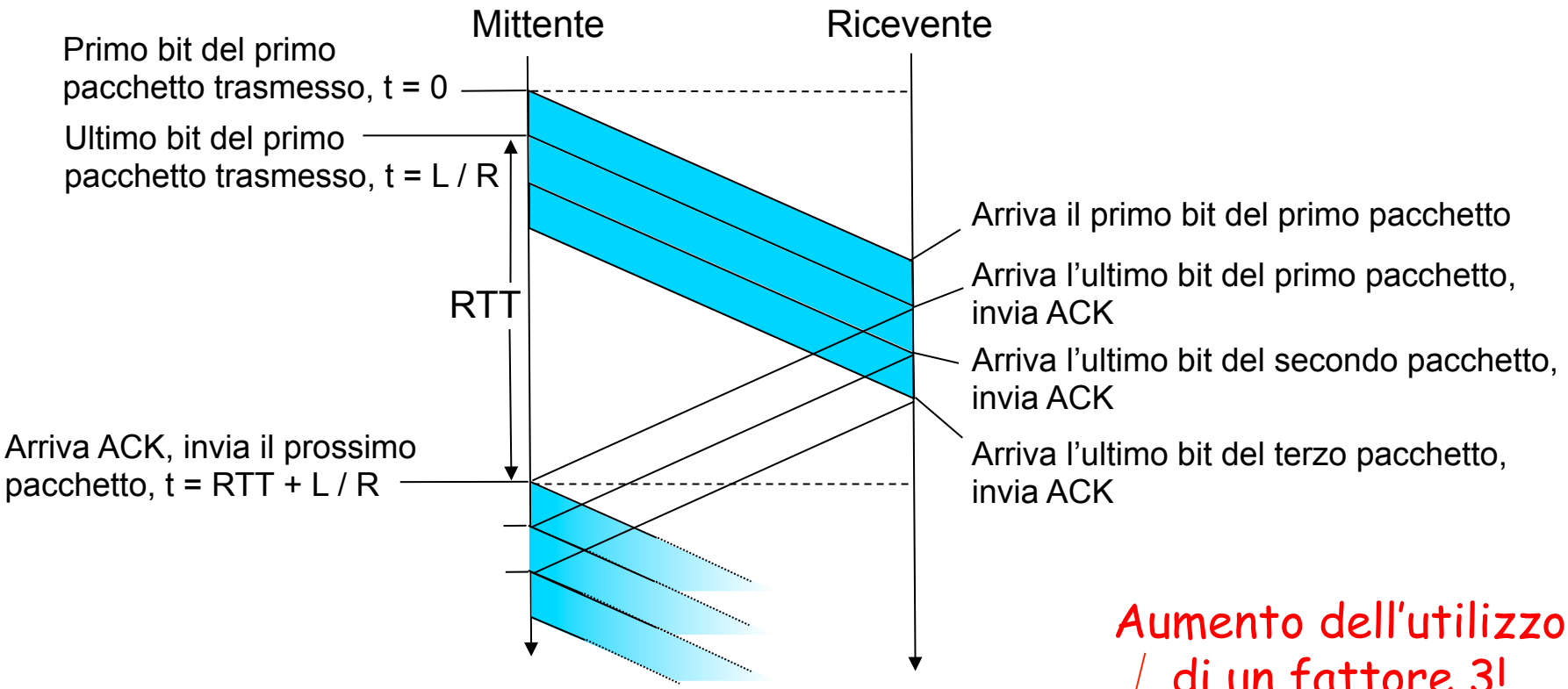


a) Protocollo stop-and-wait all'opera

b) Protocollo con pipeline all'opera

- Due forme generiche di protocolli con pipeline: *Go-Back-N* e *ripetizione selettiva*

Pipelining: aumento dell'utilizzo



$$U_{mitt} = \frac{3 * L / R}{RTT + L / R} = \frac{0{,}024}{30{,}008} = 0{,}0008$$

Aumento dell'utilizzo di un fattore 3!

Protocolli con pipeline

Go-back-N:

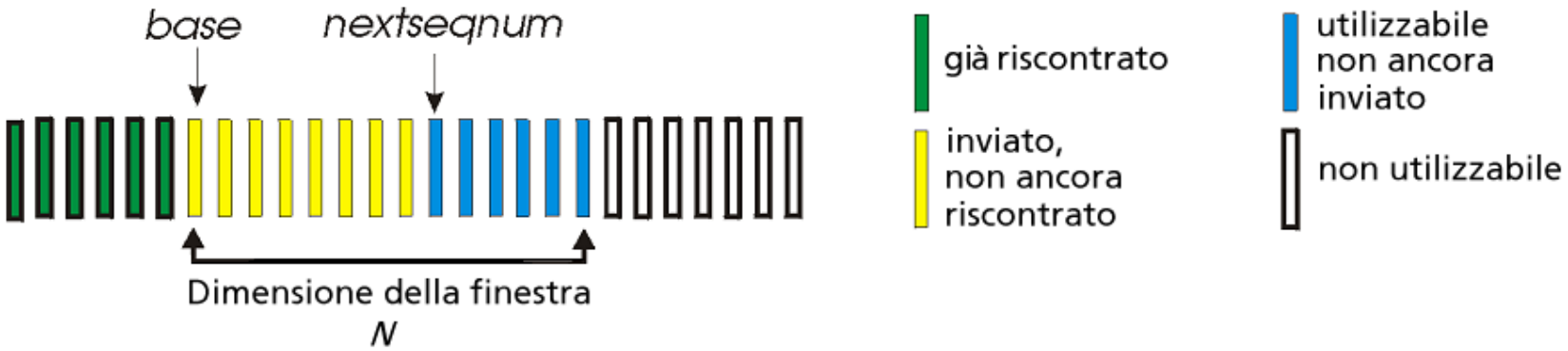
- Il mittente può avere fino a *N* pacchetti senza ACK in pipeline
- Il ricevente invia solo ACK cumulativi
  - Non dà l'ACK di un pacchetto se c'è un gap
- Il mittente ha un timer per il più vecchio pacchetto senza ACK
  - Se il timer scade, ritrasmette tutti i pacchetti senza ACK

Ripetizione selettiva

- Il mittente può avere fino a *N* pacchetti senza ACK in pipeline
- Il ricevente dà l'ACK ai singoli pacchetti
- Il mittente mantiene un timer per ciascun pacchetto che non ha ancora ricevuto ACK
  - Quando il timer scade, ritrasmette solo i pacchetti che non hanno avuto ACK

Go-Back-N

- Mittente:
- Numero di sequenza a k bit nell'intestazione del pacchetto
- "Finestra" contenente fino a N pacchetti consecutivi non riscontrati



- ACK(n): riscontro di tutti i pacchetti con numero di sequenza minore o uguale a n - "riscontri cumulativi"
- timer per il più vecchio pacchetto non riscontrato
- *timeout:* ritrasmette tutti i pacchetti non riscontrati

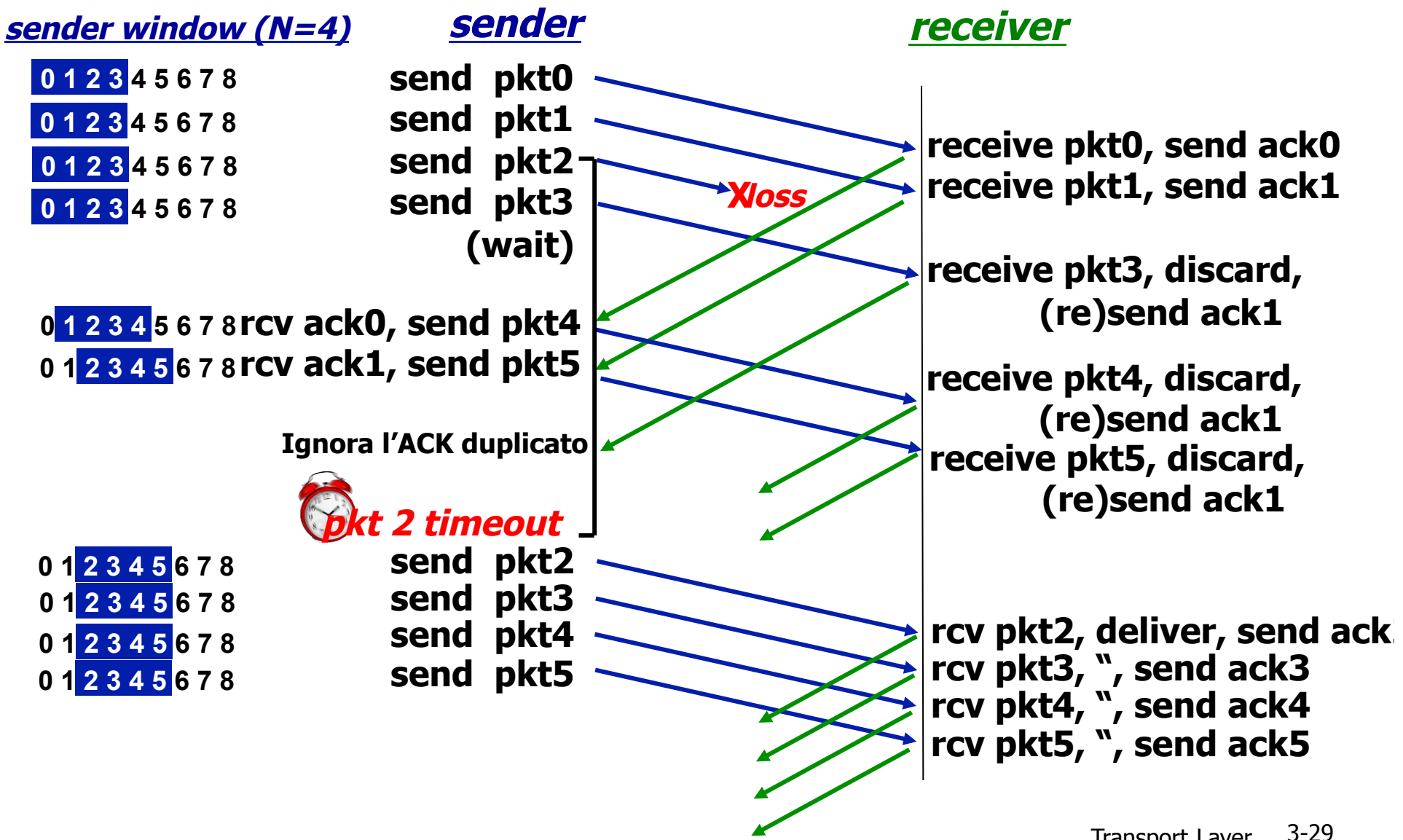
GBN: automa esteso del ricevente

ACK-soltanto: 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 `il` numero di sequenza atteso

- Pacchetto fuori sequenza:
  - scartato (non è salvato) -> senza buffering del ricevente!
  - rimanda un ACK per il pacchetto con il numero di sequenza più alto *in sequenza*

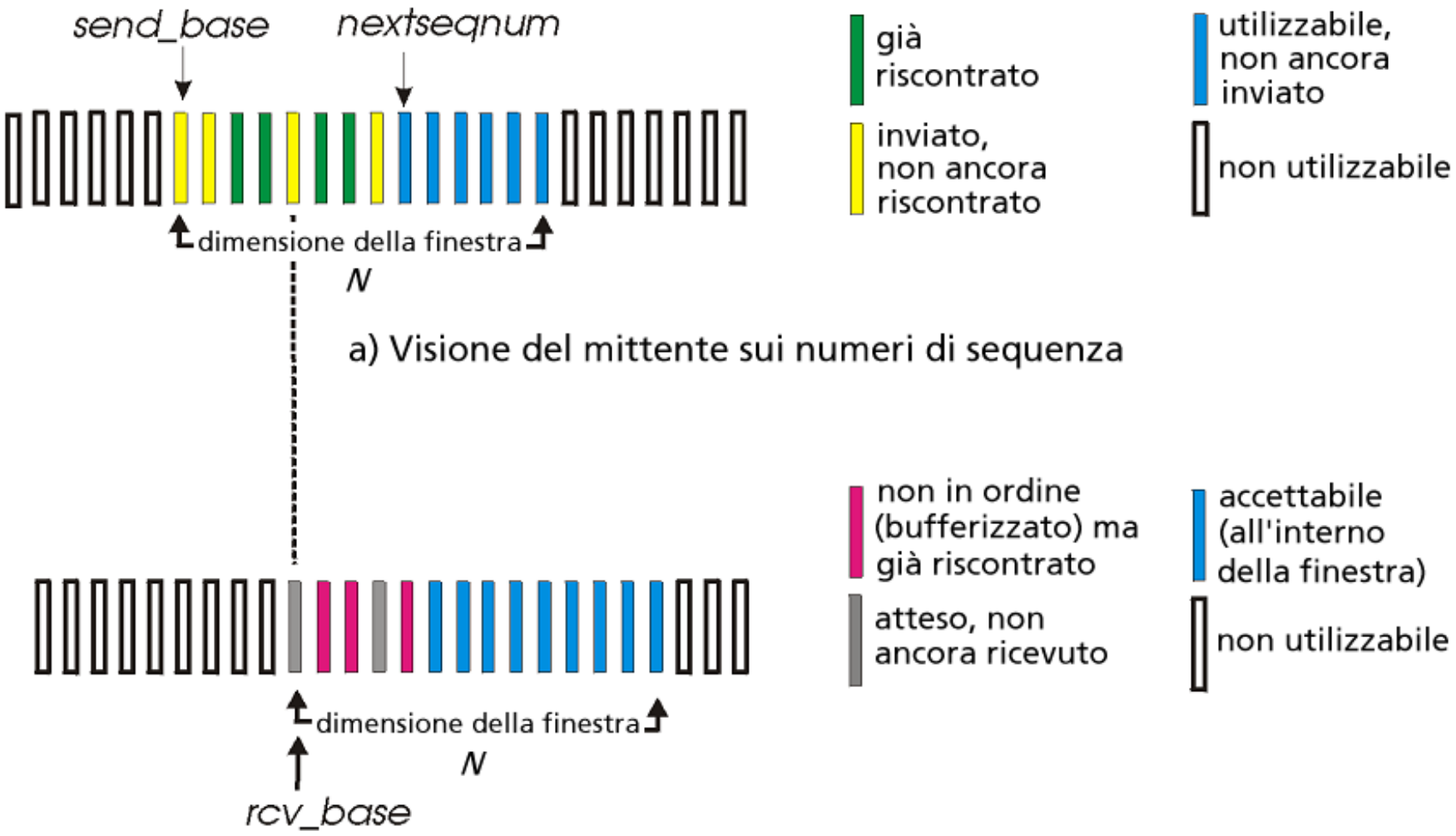
GBN in action



Ripetizione selettiva

- Il ricevente invia riscontri *specifici* per tutti i pacchetti ricevuti correttamente
  - buffer dei pacchetti, se necessario, per eventuali consegne in sequenza al livello superiore
- Il mittente ritrasmette soltanto i pacchetti per i quali non ha ricevuto un ACK
  - timer del mittente per ogni pacchetto non riscontrato
- Finestra del mittente
  - N numeri di sequenza consecutivi
  - limita ancora i numeri di sequenza dei pacchetti inviati non riscontrati

Ripetizione selettiva: finestre del mittente e del ricevente



Ripetizione selettiva

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]:

- 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

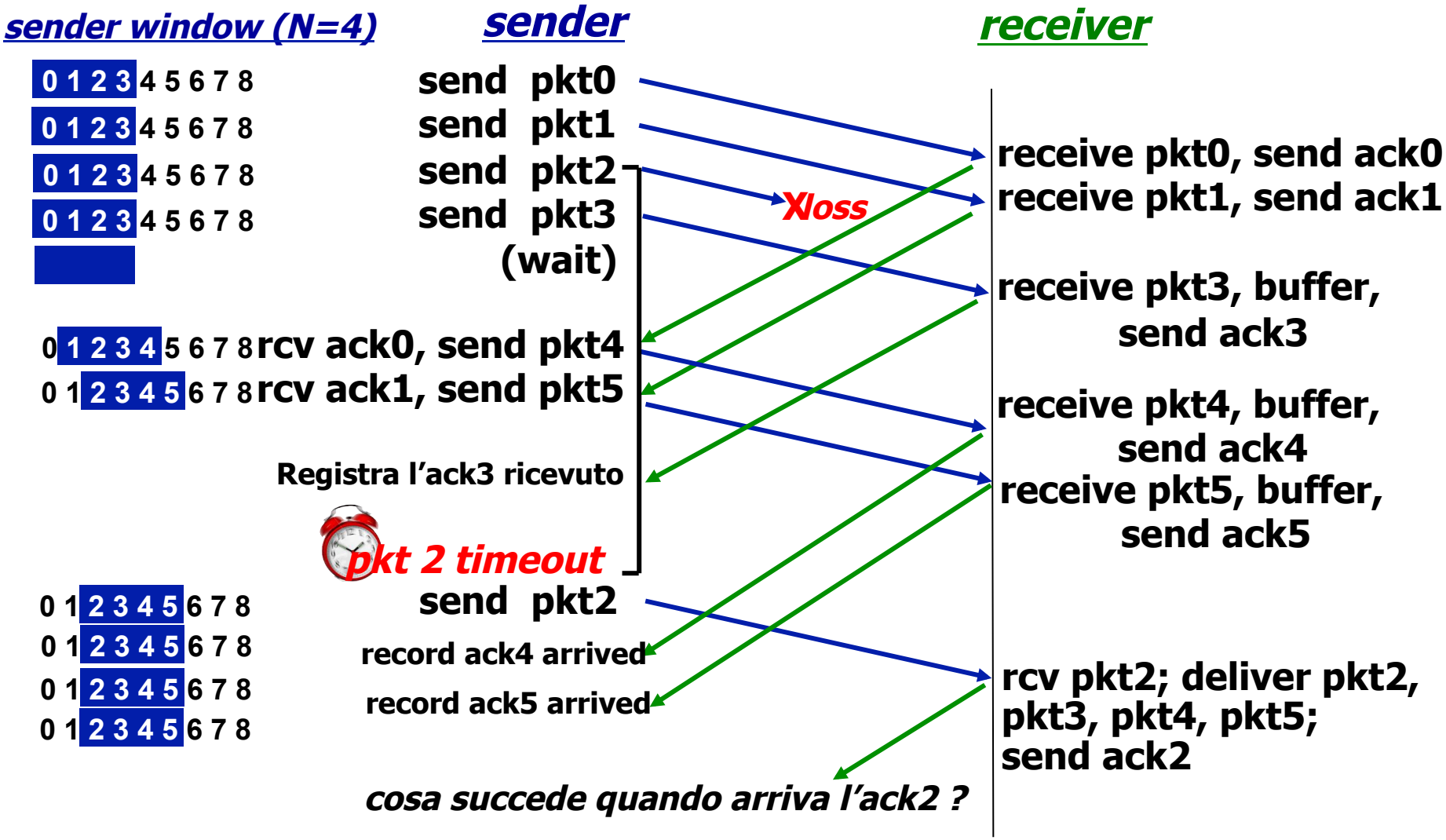
Pacchetto n in [rcvbase-N, rcvbase-1]

- ACK(n)

altrimenti:

- ignora

Selective repeat in action



Ripetizione selettiva: dilemma

Esempio:

- Numeri di sequenza: 0, 1, 2, 3
- Dimensione della finestra = 3

- Il ricevente non vede alcuna differenza fra i due scenari!
- Passa erroneamente i dati duplicati come nuovi in (a)

D: Qual è la relazione fra lo spazio dei numeri di sequenza e la dimensione della finestra?