Livello di trasporto: meccanismi trasferimento dati affidabile

Gaia Maselli
maselli@di.uniroma1.it

Livello di trasporto

Lezione precedente

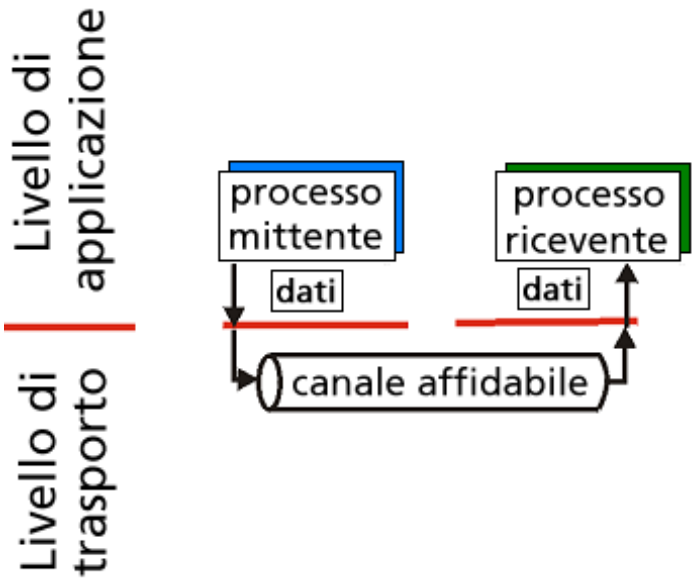
- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP

Oggi

- Principi del trasferimento dati affidabile

Principi del trasferimento dati affidabile

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

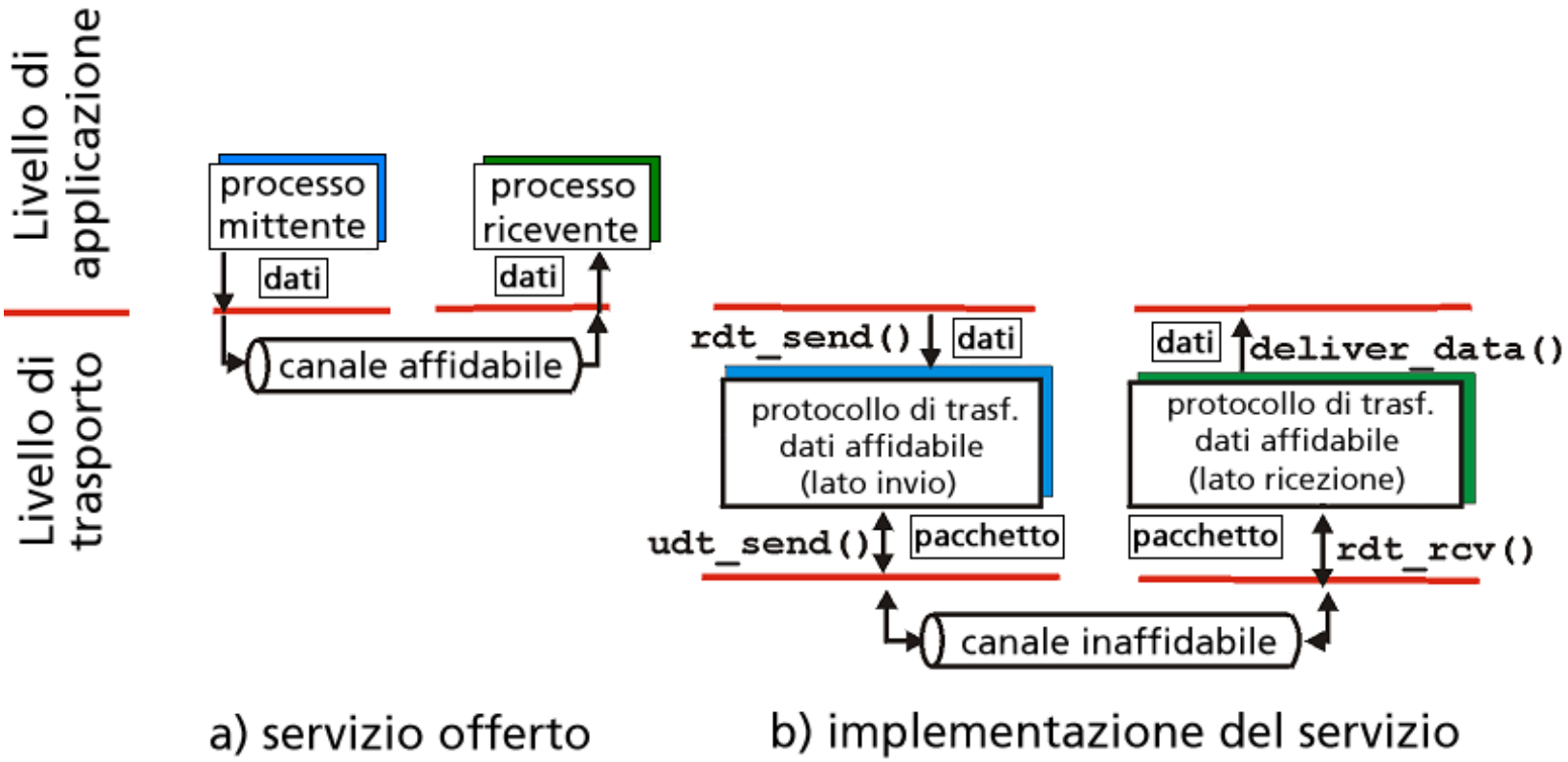


a) servizio offerto

- 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!

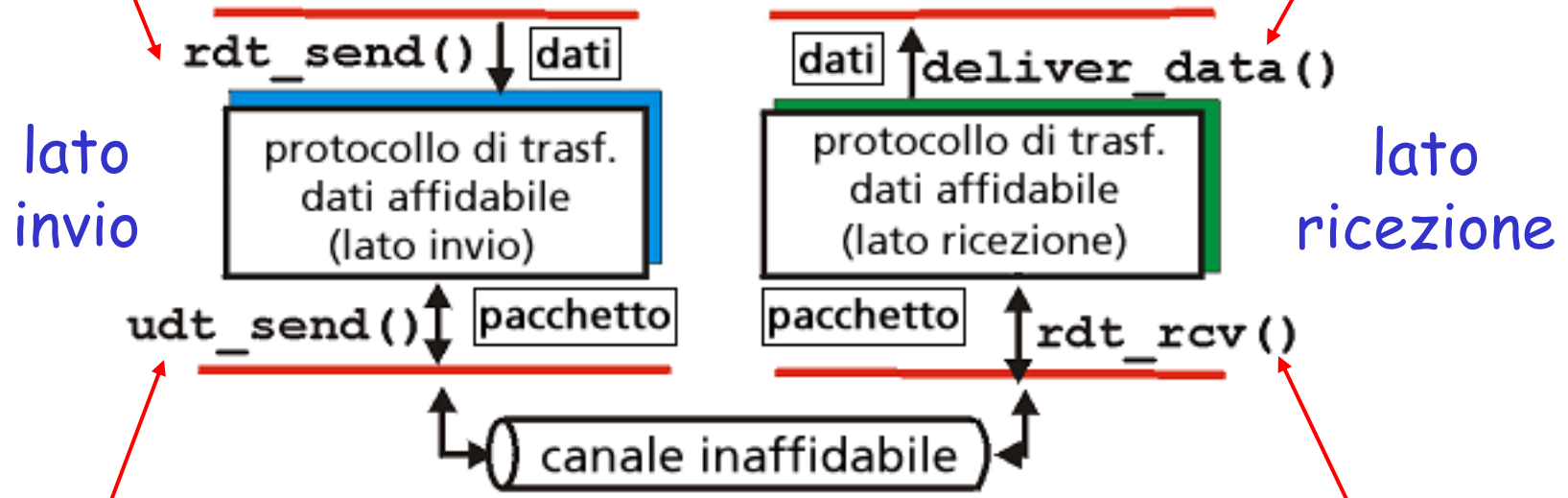


- Le caratteristiche del canale inaffidabile determinano la complessità del protocollo di trasferimento dati affidabile (reliable data transfer o rdt)

Trasferimento dati affidabile: preparazione

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

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



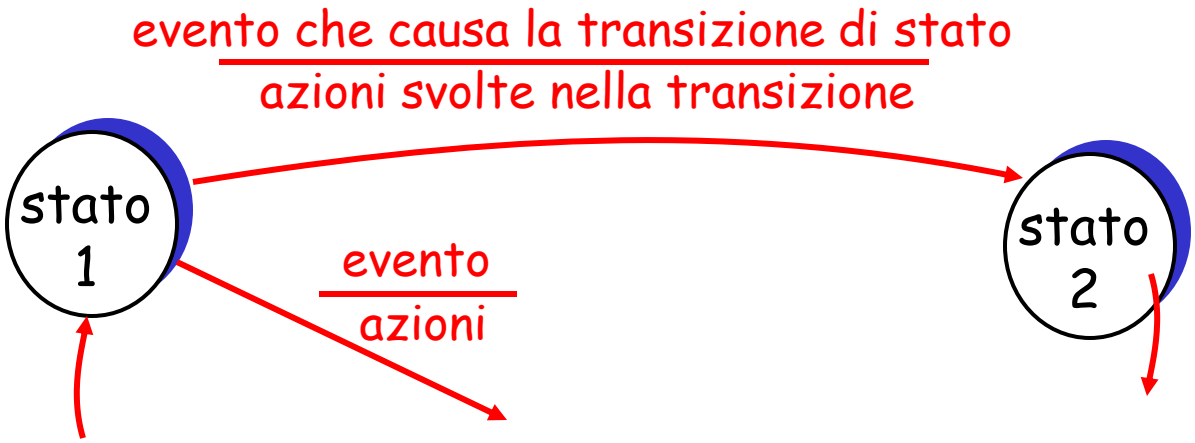
`udt_send()`: chiamata da rdt per trasferire il pacchetto al ricevente tramite il canale inaffidabile

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

Trasferimento dati affidabile: preparazione

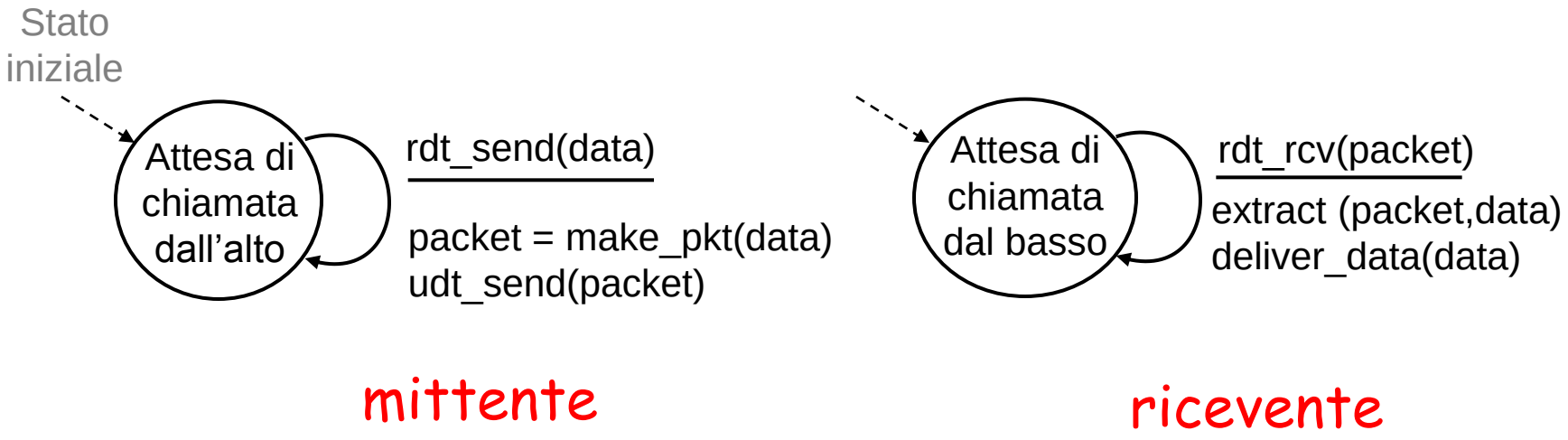
- Svilupperemo progressivamente i lati d'invio e di ricezione di un protocollo di trasferimento dati affidabile (rdt)
- Considereremo soltanto i trasferimenti dati unidirezionali
  - ma le informazioni di controllo fluiranno in entrambe le direzioni!
- Utilizzeremo automi a stati finiti per specificare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

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



Rdt1.0: trasferimento affidabile su canale affidabile

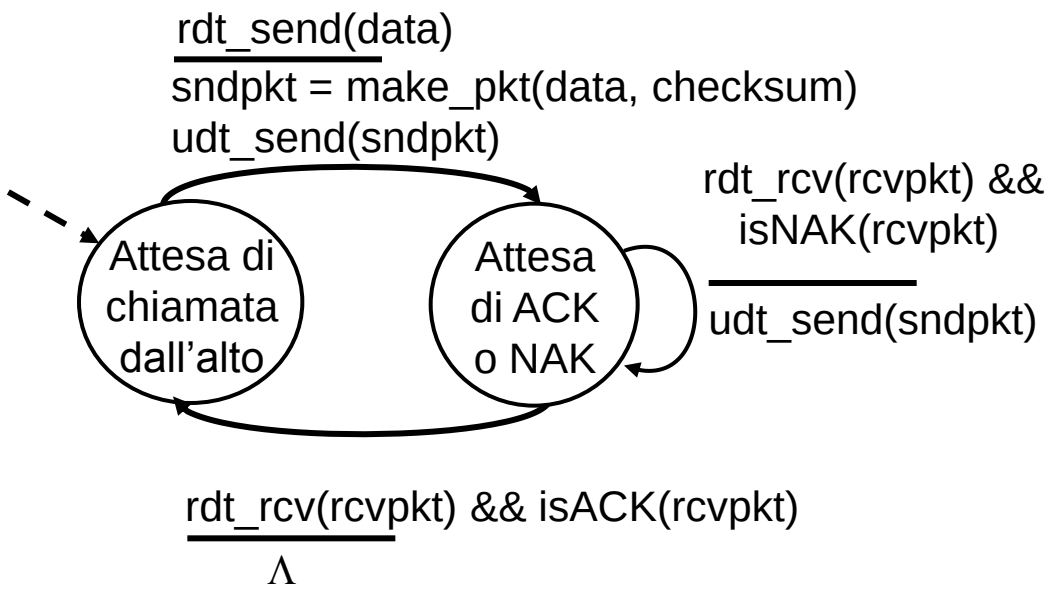
- Canale sottostante perfettamente affidabile
  - Nessun errore nei bit
  - Nessuna perdita di pacchetti
- Automa distinto per il mittente e per il ricevente:
  - il mittente invia i dati nel canale sottostante
  - il ricevente legge i dati dal canale sottostante



Rdt2.0: canale con errori nei bit

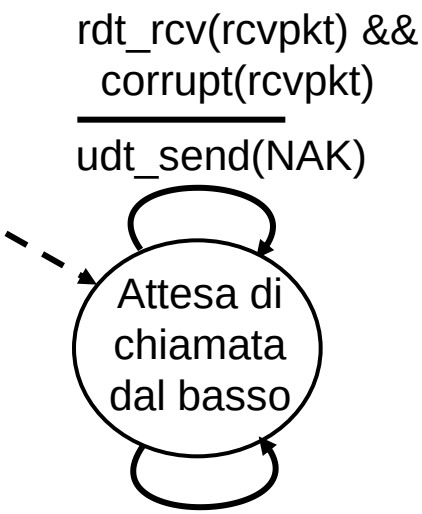
- Il canale sottostante potrebbe confondere i bit nei pacchetti
  - checksum per rilevare gli errori nei bit
- "Puoi ripetere per favore ?"
- *Protocolli ARQ (Automatic Repeat reQuest)*:
  - *notifica positiva (ACK):* il ricevente comunica espressamente al mittente che il pacchetto ricevuto è corretto
  - *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`):
  - Rilevamento di errore (es. tramite checksum)
  - Feedback del destinatario: messaggi di controllo (ACK, NAK) ricevente->mittente
  - Ritrasmissione

rdt2.0: specifica dell'automa



mittente

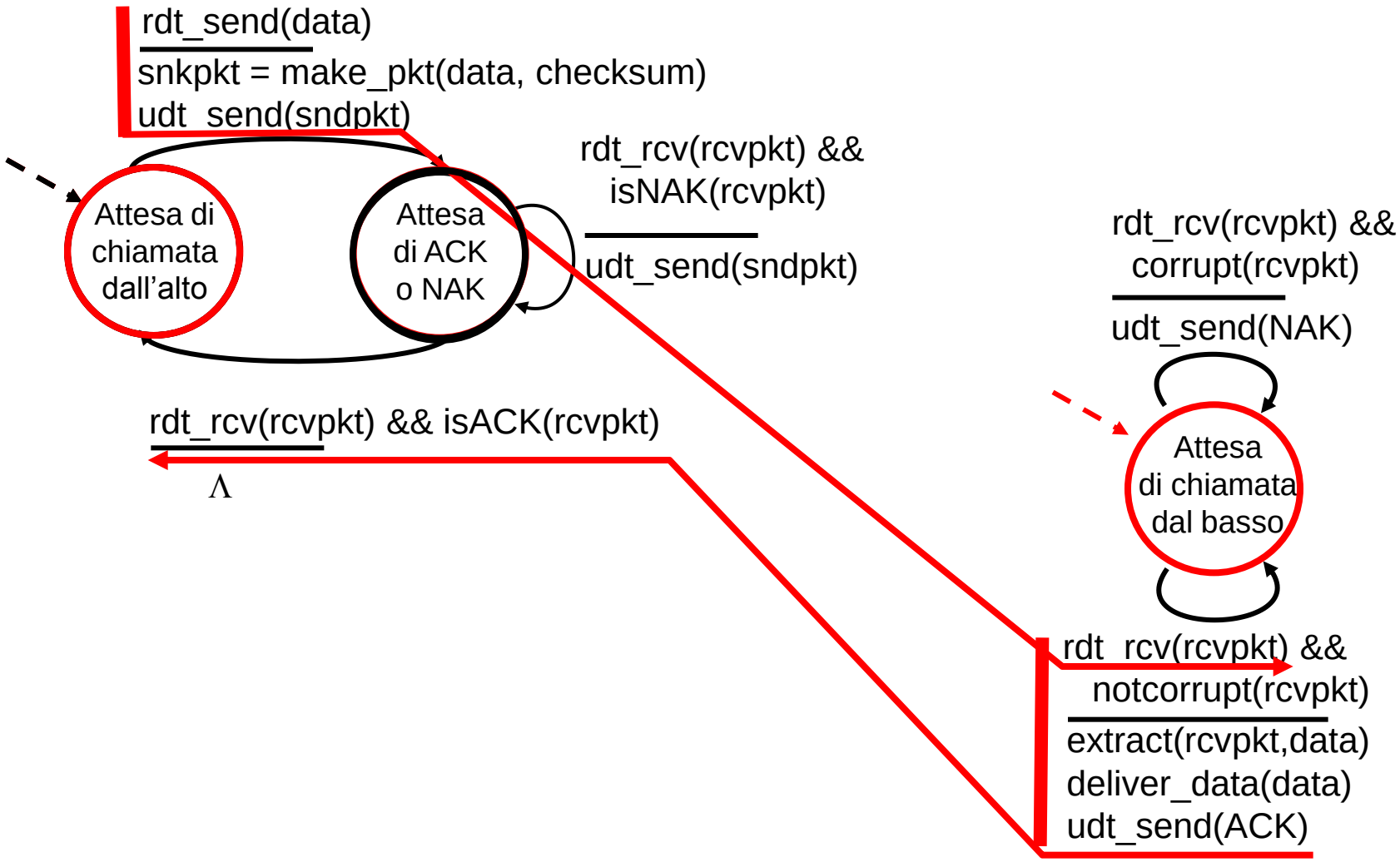
ricevente



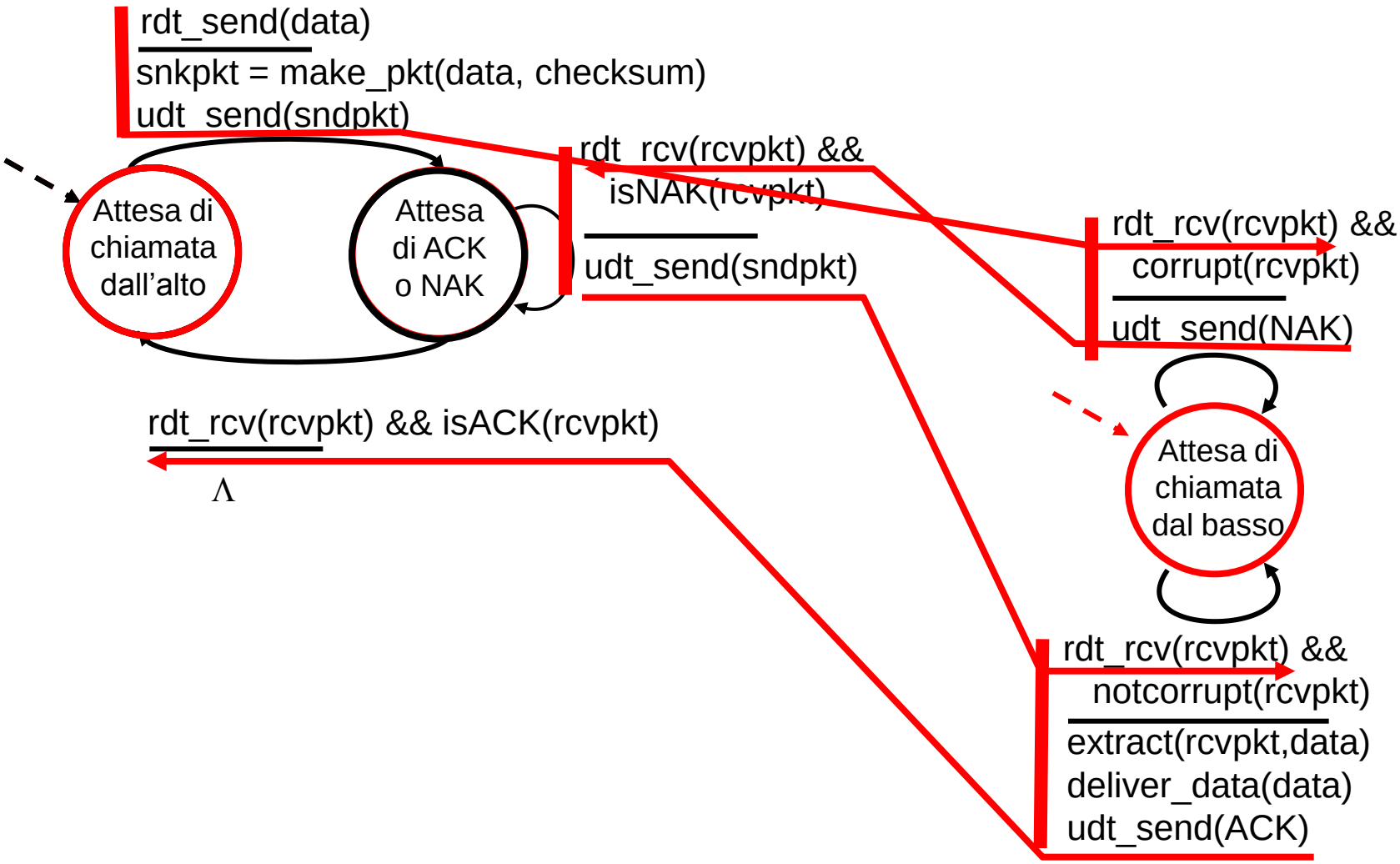
stop and wait

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

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!
- Può ritrasmettere: possibili duplicati (da gestire!!!)

Come fa l'entità di trasporto ricevente a sapere che gli è arrivato un duplicato? (non guarda i dati dell'applicazione)

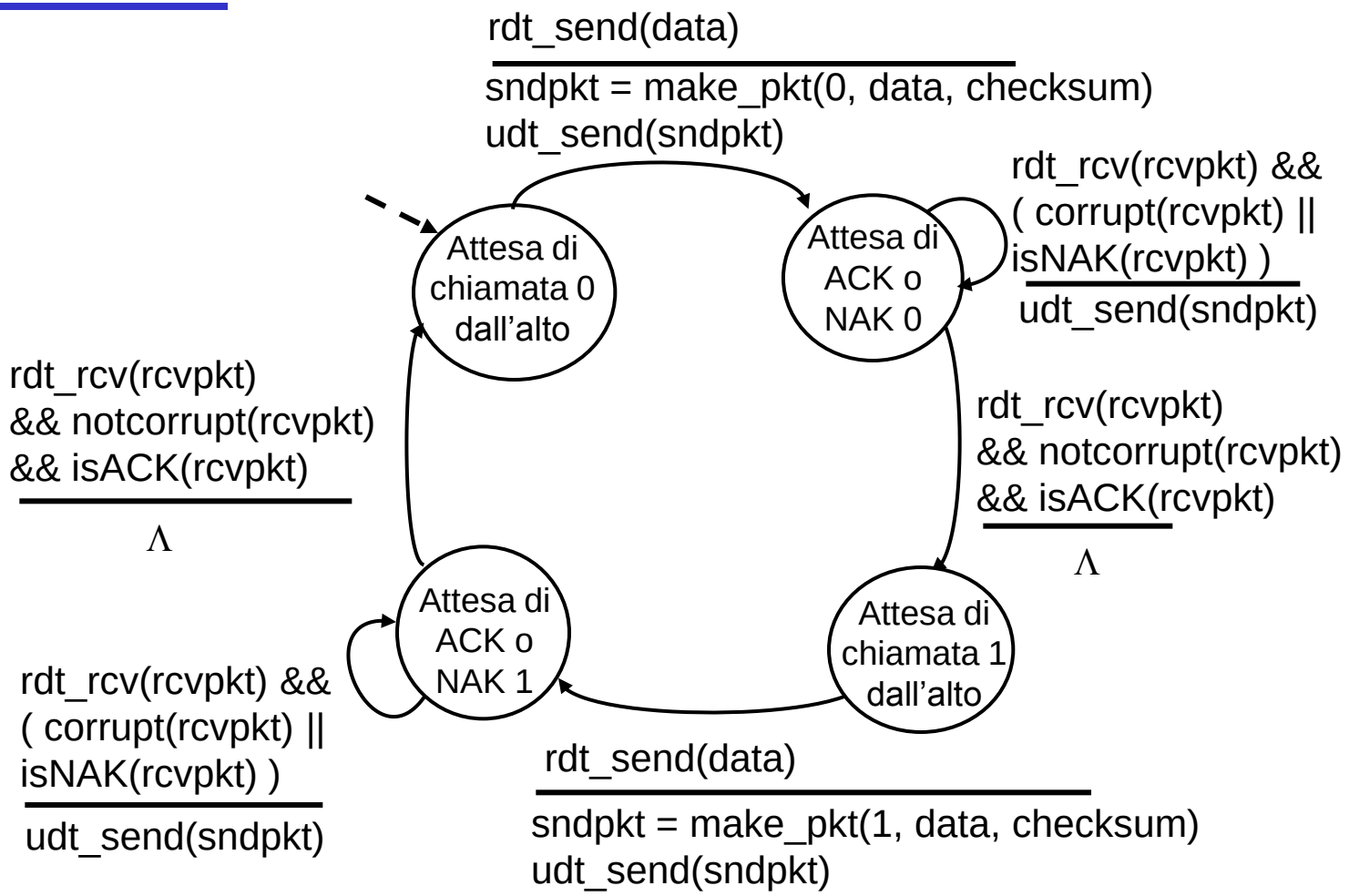
Gestione dei duplicati:

- Il mittente ritrasmette il pacchetto corrente se ACK/NAK è alterato
- Il mittente aggiunge un *numero di sequenza* (di un bit, 0 - 1) a ogni pacchetto
- Il ricevente scarta il pacchetto duplicato

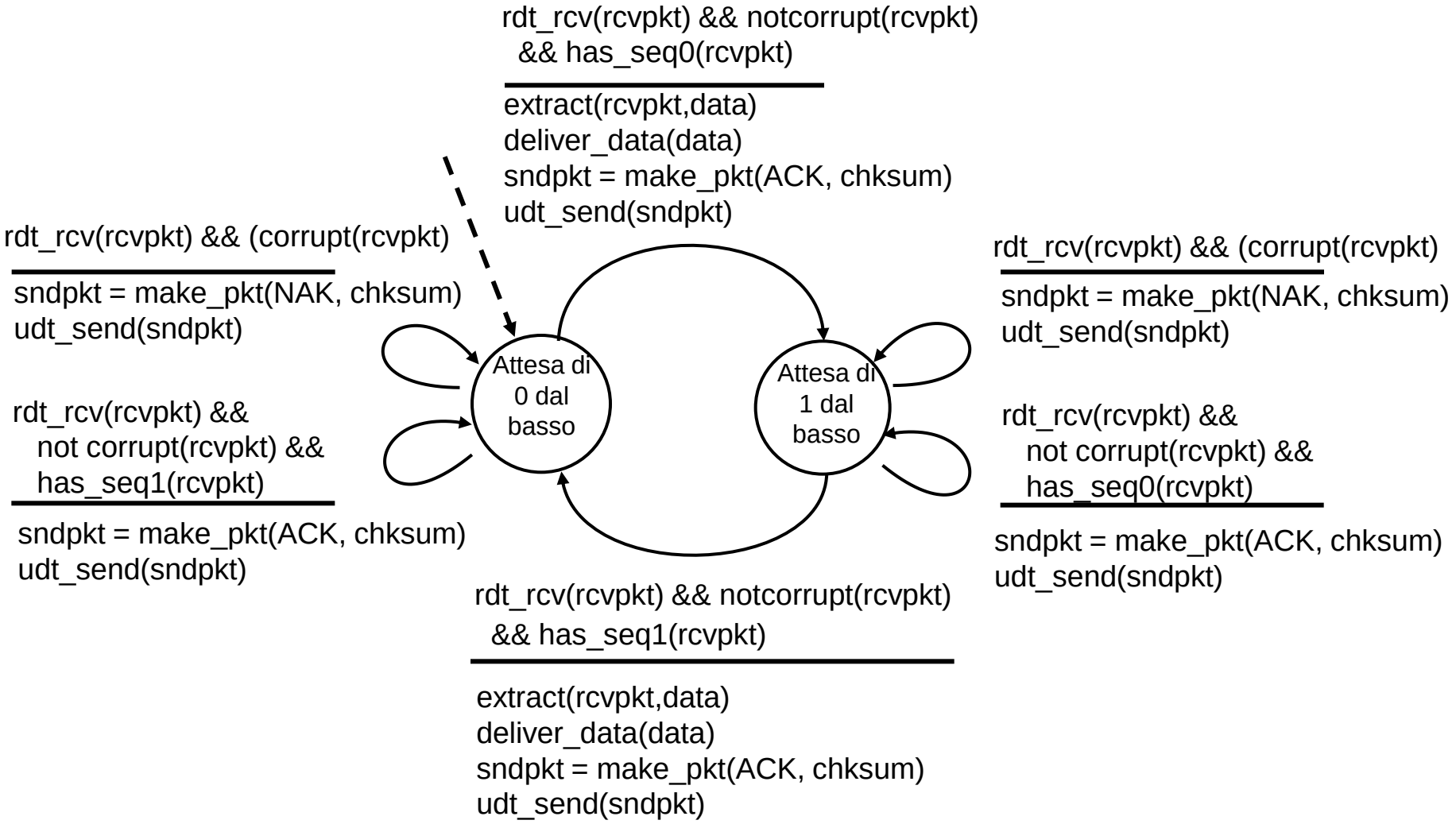
stop and wait

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
- Sono sufficienti due numeri di sequenza. 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

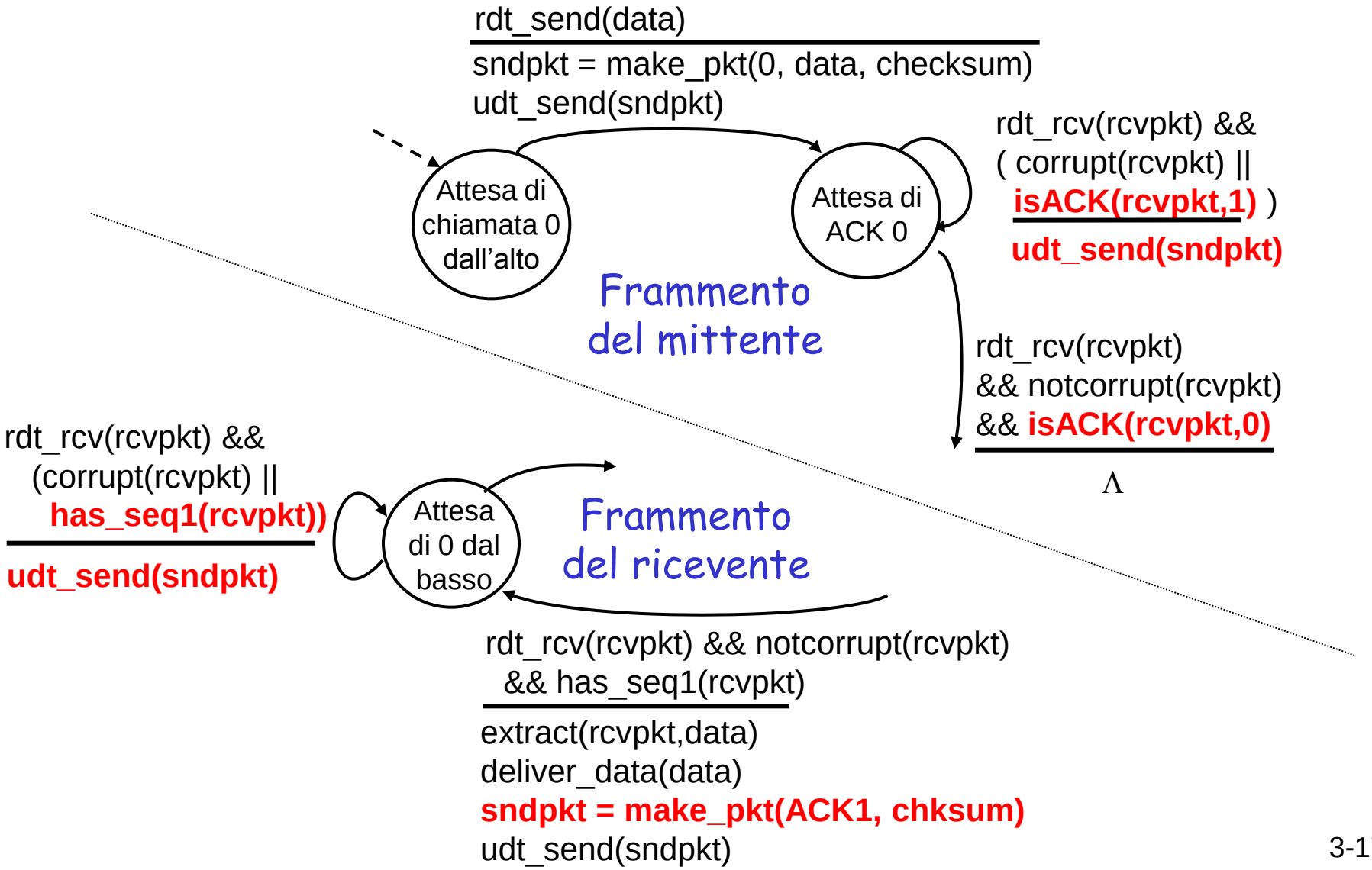
Ricevente:

- 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*

rdt2.2: frammenti del mittente e del ricevente



rdt3.0: canali con errori *e* perdite

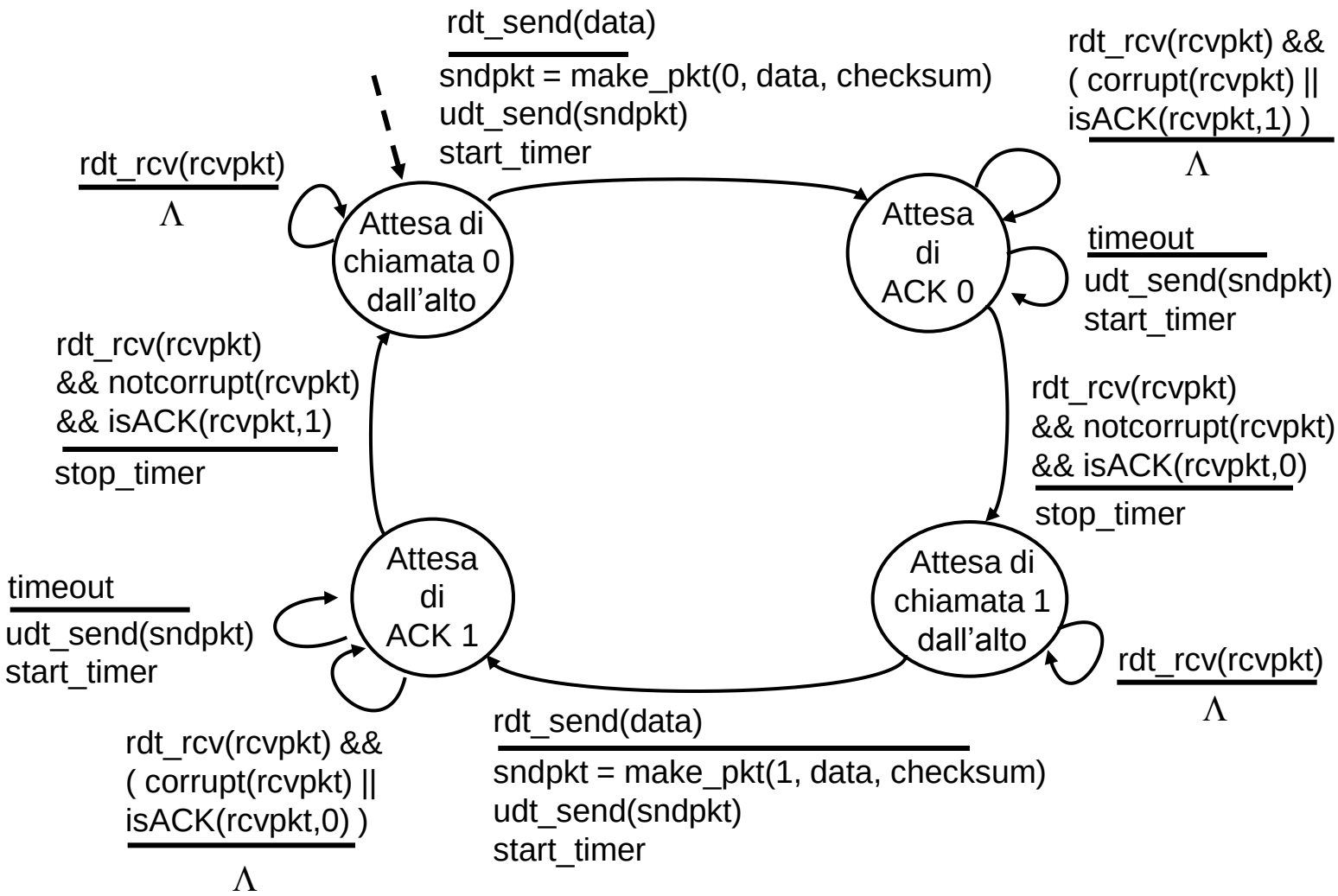
Nuova ipotesi: il canale sottostante può anche smarrire i pacchetti (dati o ACK)

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

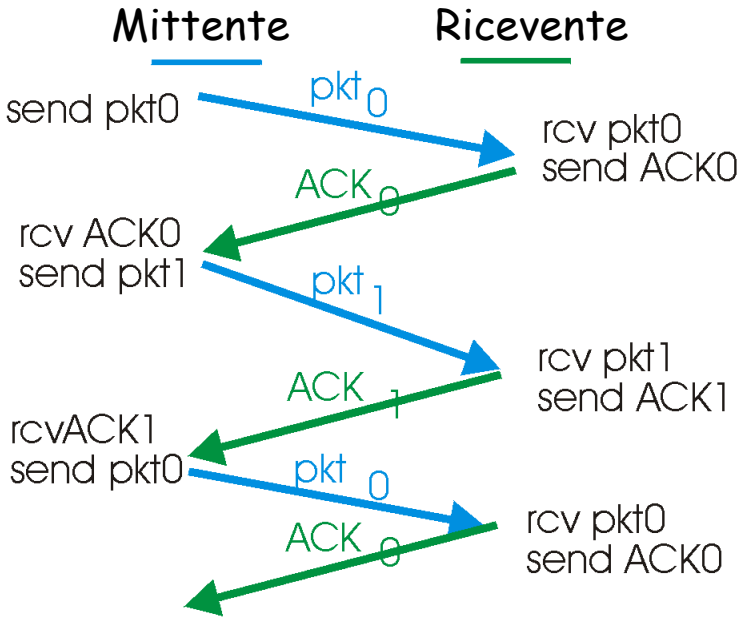
Approccio: il mittente attende un ACK per un tempo "ragionevole"

- 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*)

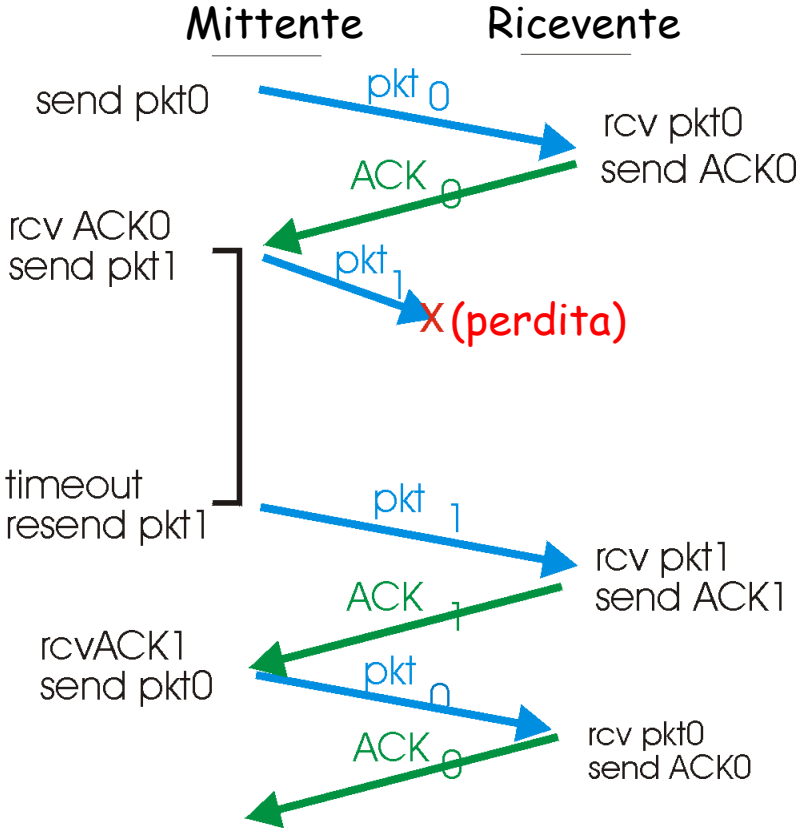
rdt3.0 mittente



rdt3.0 in azione

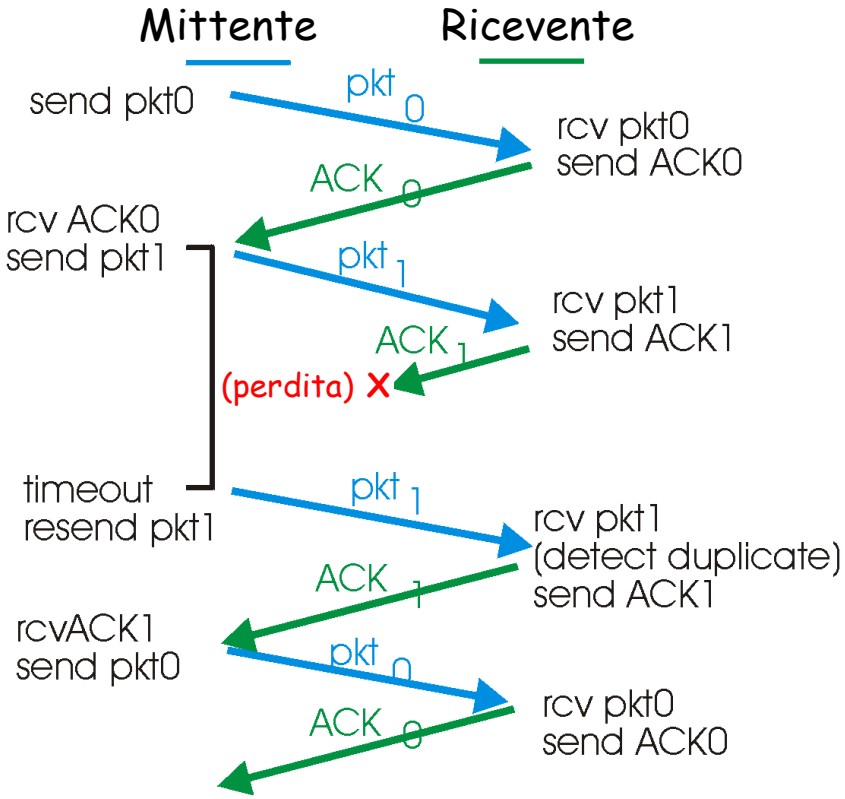


a) Operazioni senza perdite

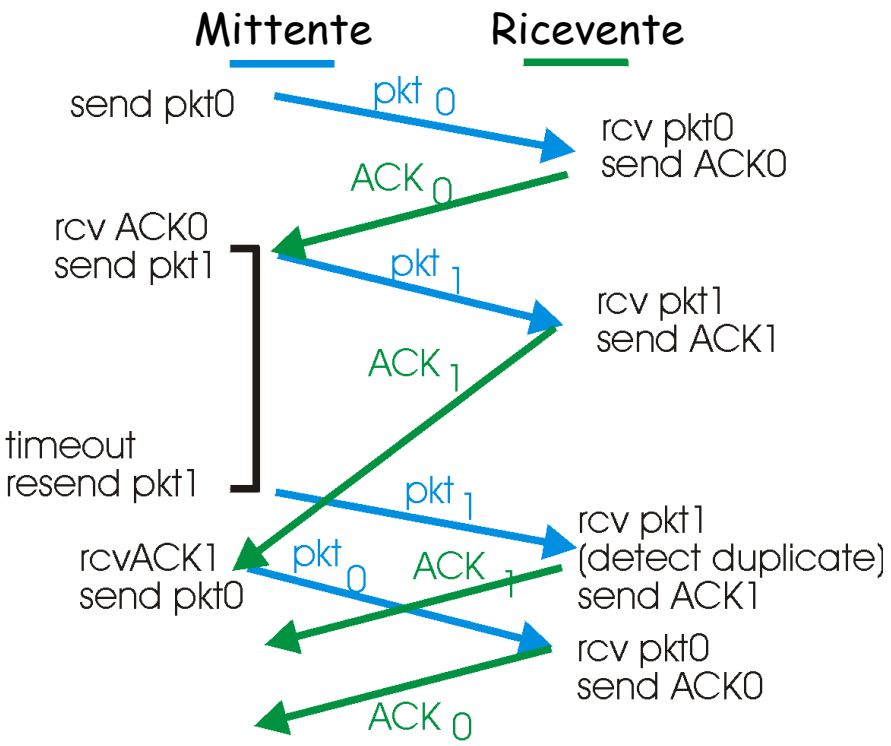


b) Perdita di pacchetto

rdt3.0 in azione



c) Perdita di ACK



d) Timeout prematuro

Prefissi metrici

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	,000,000,000,000,000,000,000	Yotta

- Es. una linea di comunicazione da 1Mbps trasmette 10^6 bit al secondo
- N.B. la memoria è misurata diversamente (potenze di 2, Kilo significa 2^{10} ovvero 1024)

Prestazioni di rdt3.0

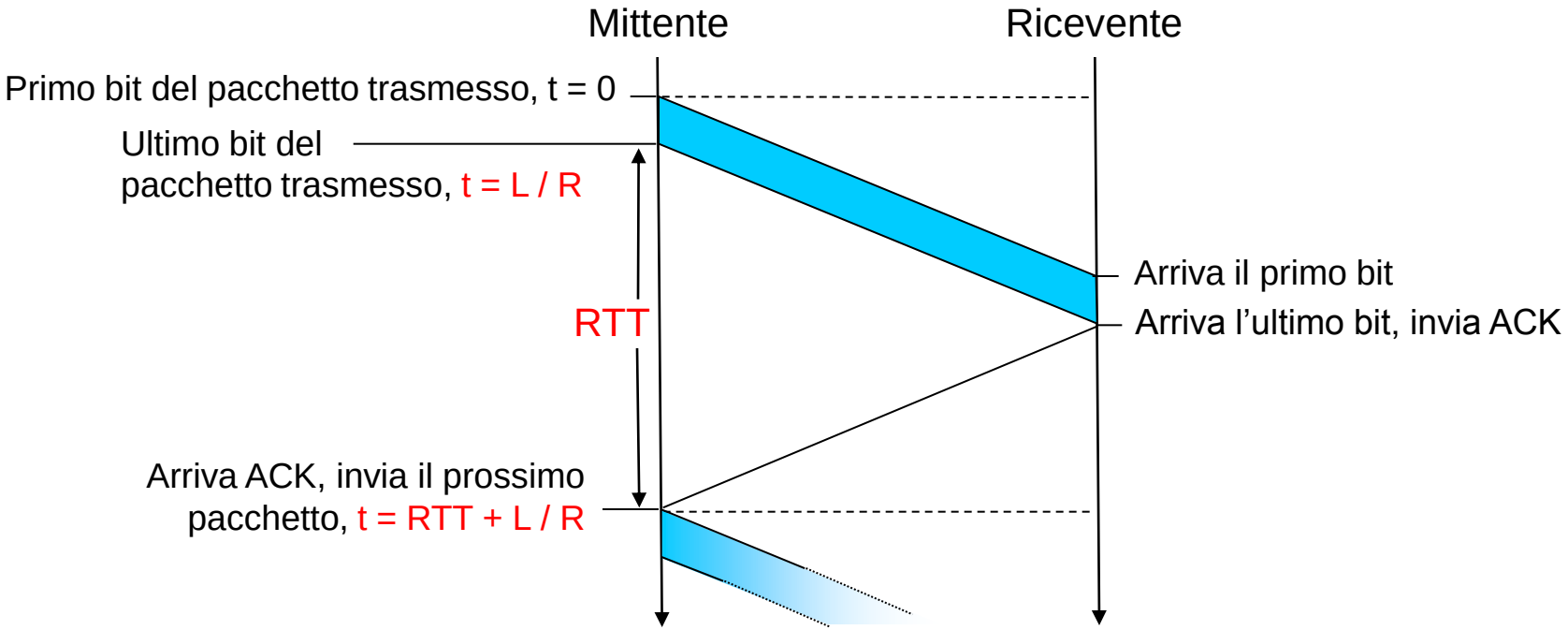
- rdt3.0 funziona (bit corrotti, perdita pacchetti), ma le prestazioni non sono apprezzabili
- esempio: collegamento da 1 Gbps, ritardo di propagazione 15 ms, pacchetti da 1 KB:

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

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

  - U_{mitt} : utilizzo è la frazione di tempo in cui il mittente è occupato nell'invio di bit
  - Un pacchetto da 8 kb ogni 30 msec -> throughput di 267 Kbps in un collegamento da 1 Gbps
  - Il protocollo di rete limita l'uso delle risorse fisiche!

rdt3.0: funzionamento con stop-and-wait

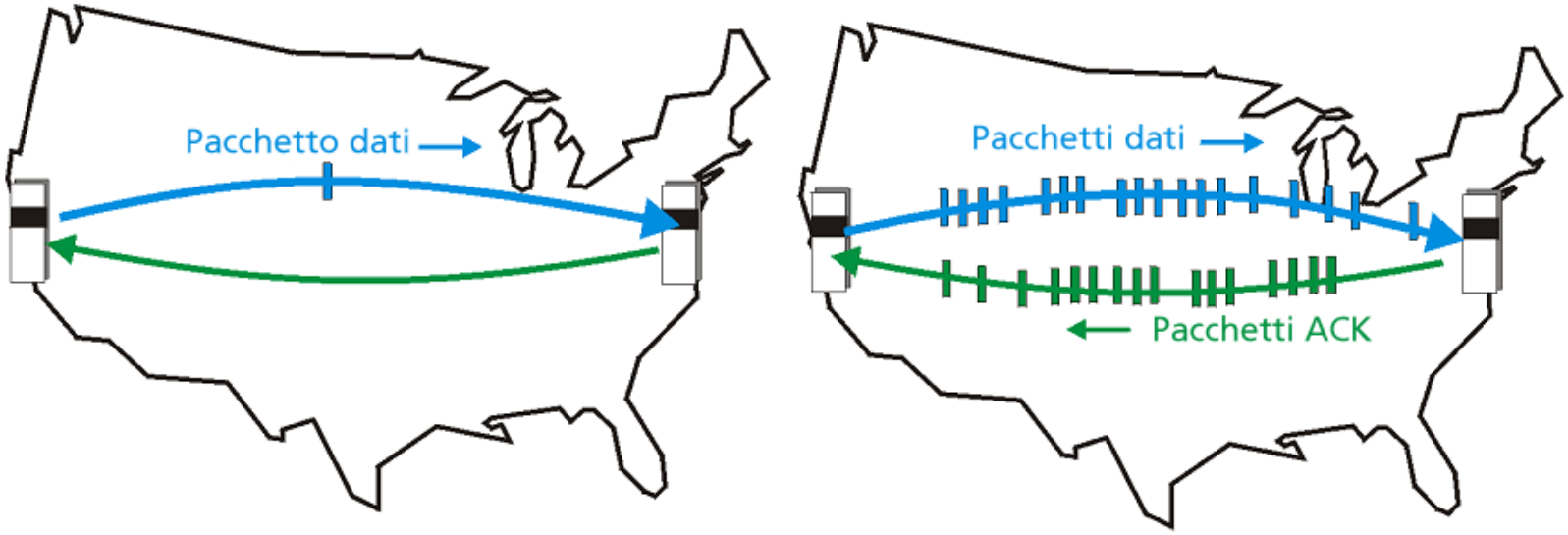


$$U_{mitt} = \frac{L / R}{RTT + L / R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

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

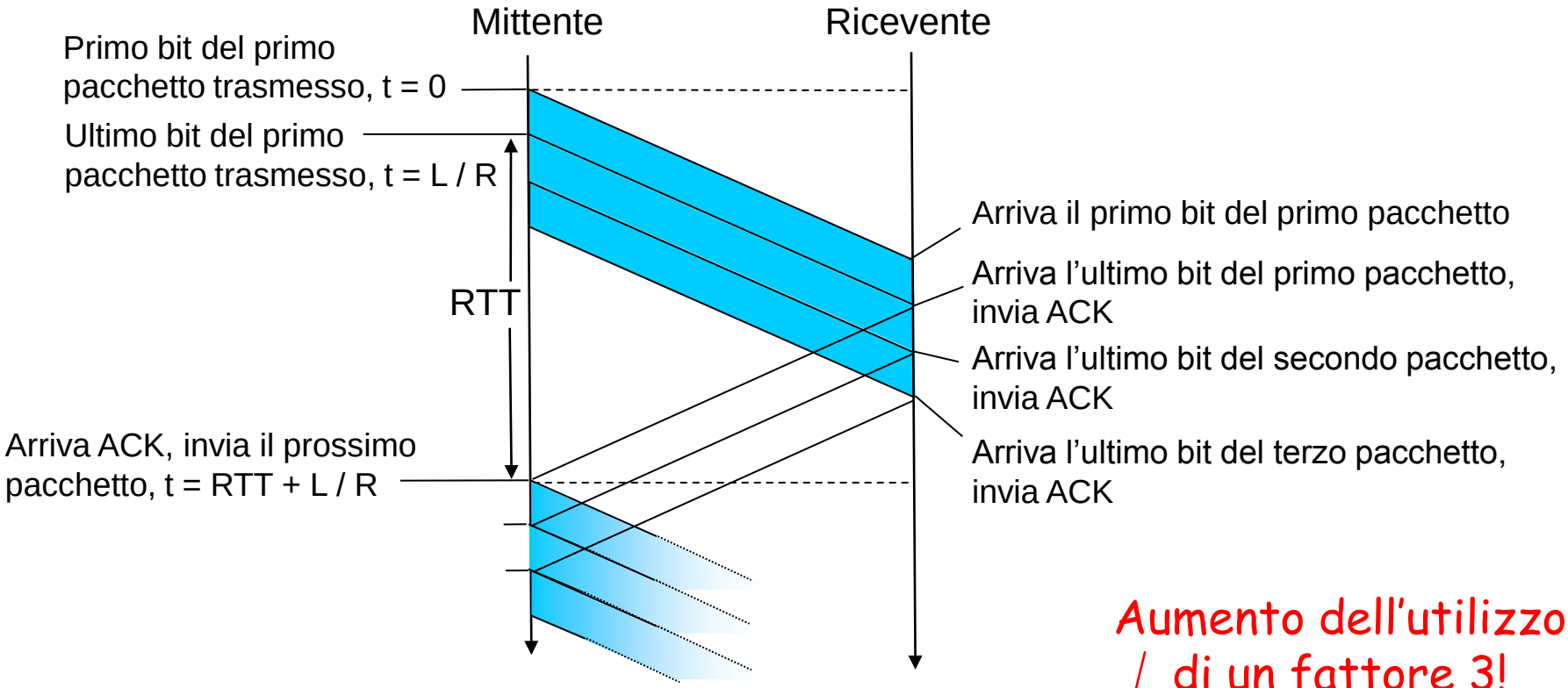


a) Protocollo stop-and-wait all'opera

b) Protocollo con pipeline all'opera

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

Pipelining: aumento dell'utilizzo



Aumento dell'utilizzo di un fattore 3!

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