

Capitolo 3

Livello di trasporto

All material copyright 1996-2007
J.F Kurose and K.W. Ross, All Rights Reserved



*Reti di calcolatori e Internet:
Un approccio top-down*

4ª edizione
Jim Kurose, Keith Ross

Pearson Paravia Bruno Mondadori Spa
©2008

Capitolo 3: Livello di trasporto

Obiettivi:

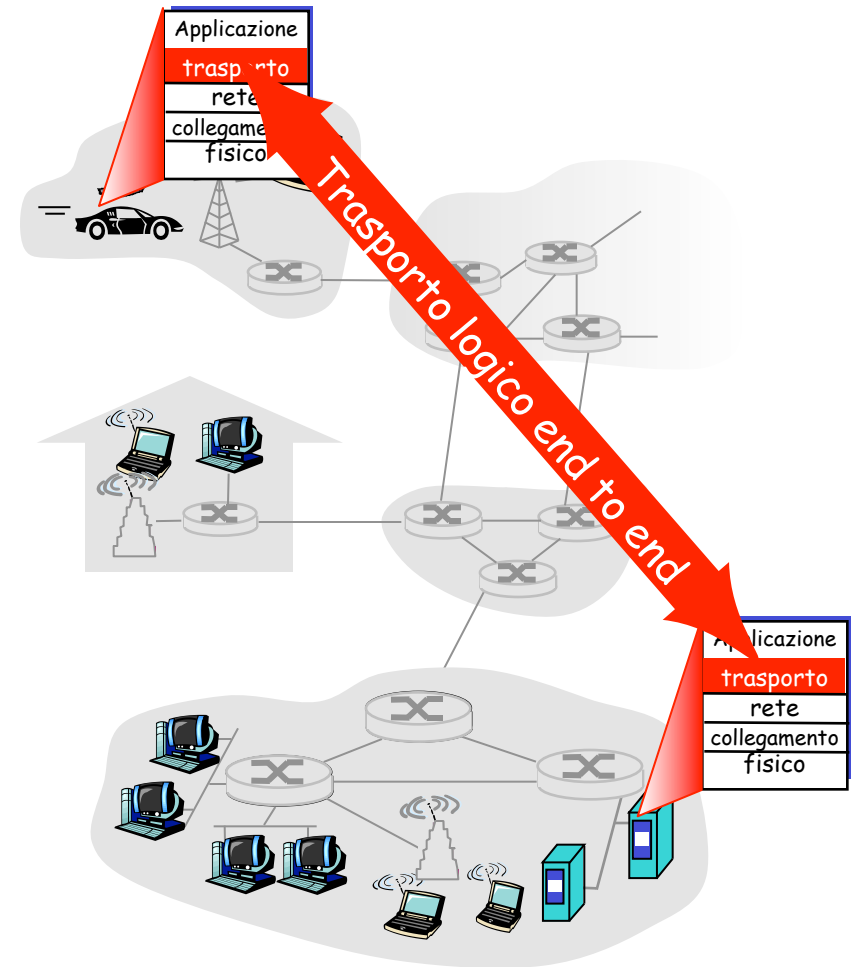
- ❑ Capire i principi che sono alla base dei servizi del livello di trasporto:
 - multiplexing/demultiplexing
 - trasferimento dati affidabile
 - controllo di flusso
 - controllo di congestione
- ❑ Descrivere i protocolli del livello di trasporto di Internet:
 - UDP: trasporto senza connessione
 - TCP: trasporto orientato alla connessione
 - controllo di congestione TCP

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

Servizi e protocolli di trasporto

- ❑ Forniscono la *comunicazione logica* tra processi applicativi di host differenti
- ❑ I protocolli di trasporto vengono eseguiti nei sistemi terminali
 - lato invio: scinde i messaggi in *segmenti* e li passa al livello di rete
 - lato ricezione: riassembla i segmenti in messaggi e li passa al livello di applicazione
- ❑ Più protocolli di trasporto sono a disposizione delle applicazioni
 - Internet: TCP e UDP



Relazione tra livello di trasporto e livello di rete

- ❑ *livello di rete:*
comunicazione logica tra host
- ❑ *livello di trasporto:*
comunicazione logica tra processi
 - si basa sui servizi del livello di rete e li potenzia

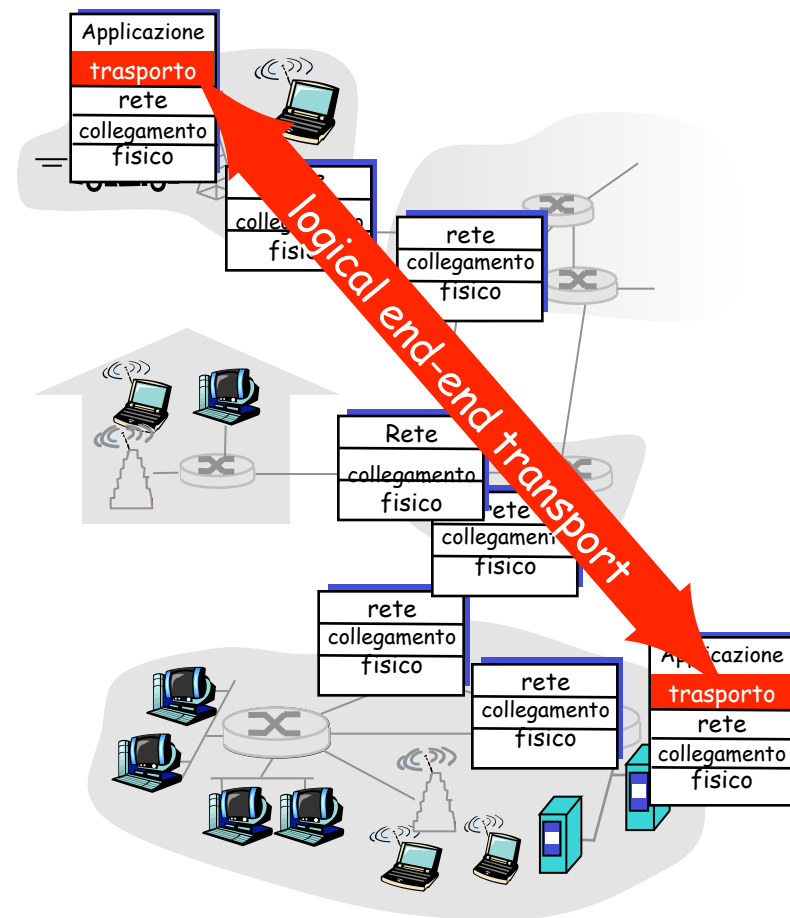
Analogia con la posta ordinaria:

12 ragazzi inviano lettere a 12 ragazzi

- ❑ processi = ragazzi
- ❑ messaggi delle applicazioni = lettere nelle buste
- ❑ host = case
- ❑ protocollo di trasporto = Anna e Andrea
- ❑ protocollo del livello di rete = servizio postale

Protocolli del livello di trasporto in Internet

- ❑ Affidabile, consegne nell'ordine originario (TCP)
 - controllo di congestione
 - controllo di flusso
 - setup della connessione
- ❑ Inaffidabile, consegne senz'ordine: UDP
 - estensione senza fronzoli del servizio di consegna best effort
- ❑ Servizi non disponibili:
 - garanzia su ritardi
 - garanzia su ampiezza di banda



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

Multiplexing/demultiplexing

Demultiplexing

nell'host ricevente:

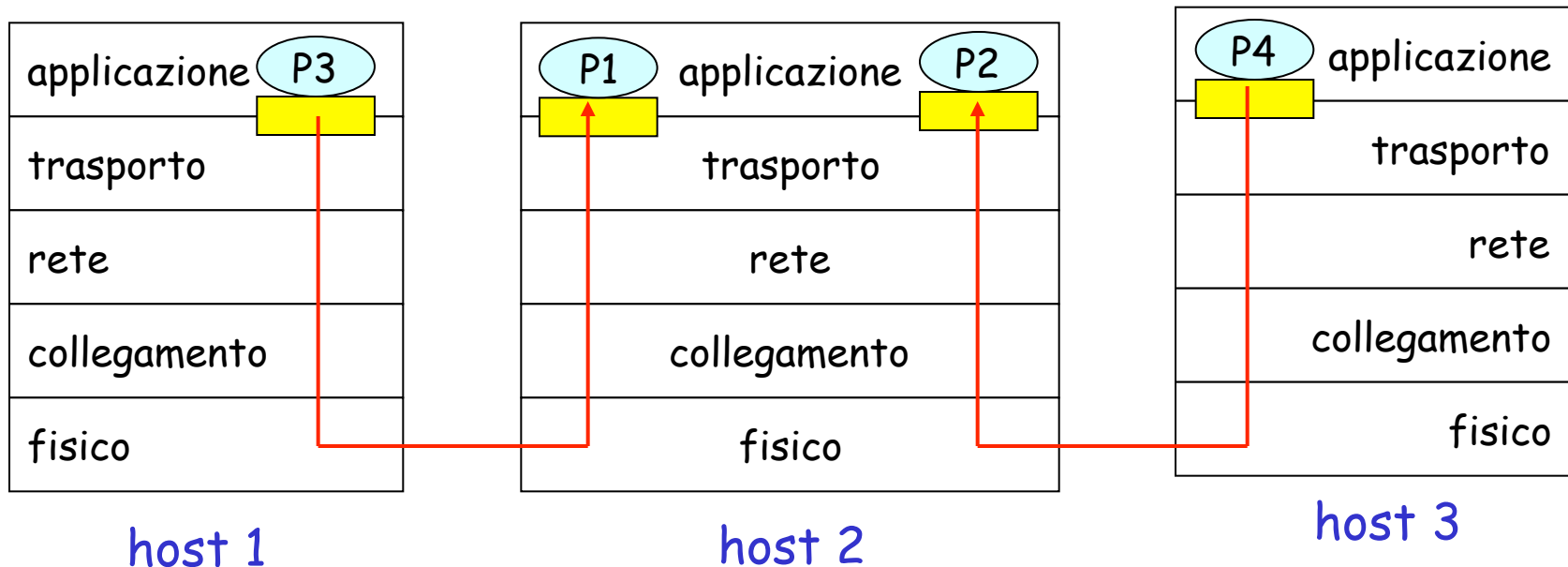
consegnare i segmenti ricevuti alla socket appropriata

Multiplexing

nell'host mittente:

raccogliere i dati da varie socket, incapsularli con l'intestazione (utilizzata poi per il demultiplexing)

 = socket  = processo

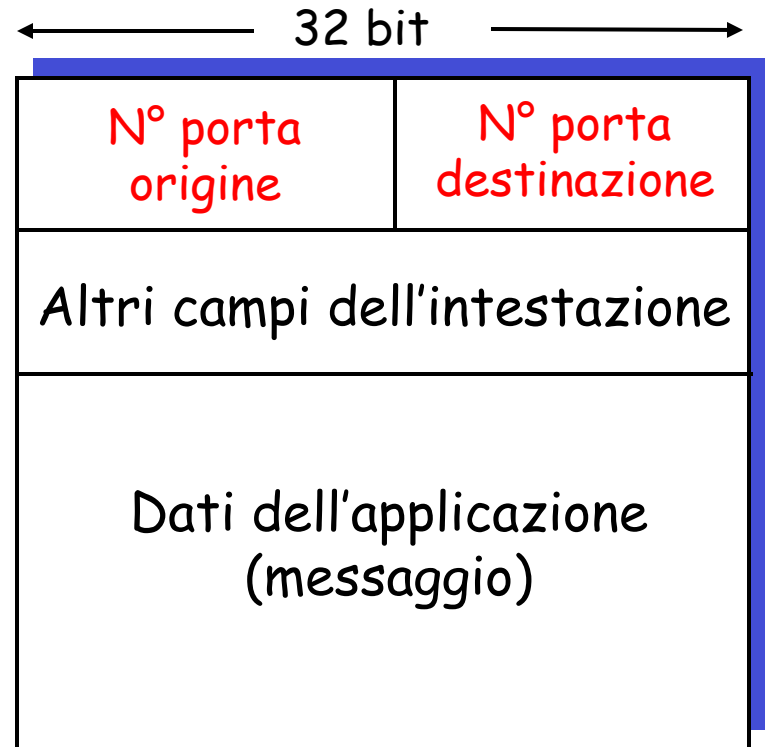


Come funziona il demultiplexing

L'host riceve i datagrammi IP

- ogni datagramma ha un indirizzo IP di origine e un indirizzo IP di destinazione
- ogni datagramma trasporta 1 segmento a livello di trasporto
- ogni segmento ha un numero di porta di origine e un numero di porta di destinazione

L'host usa gli indirizzi IP e i numeri di porta per inviare il segmento alla socket appropriata



Struttura del segmento TCP/UDP

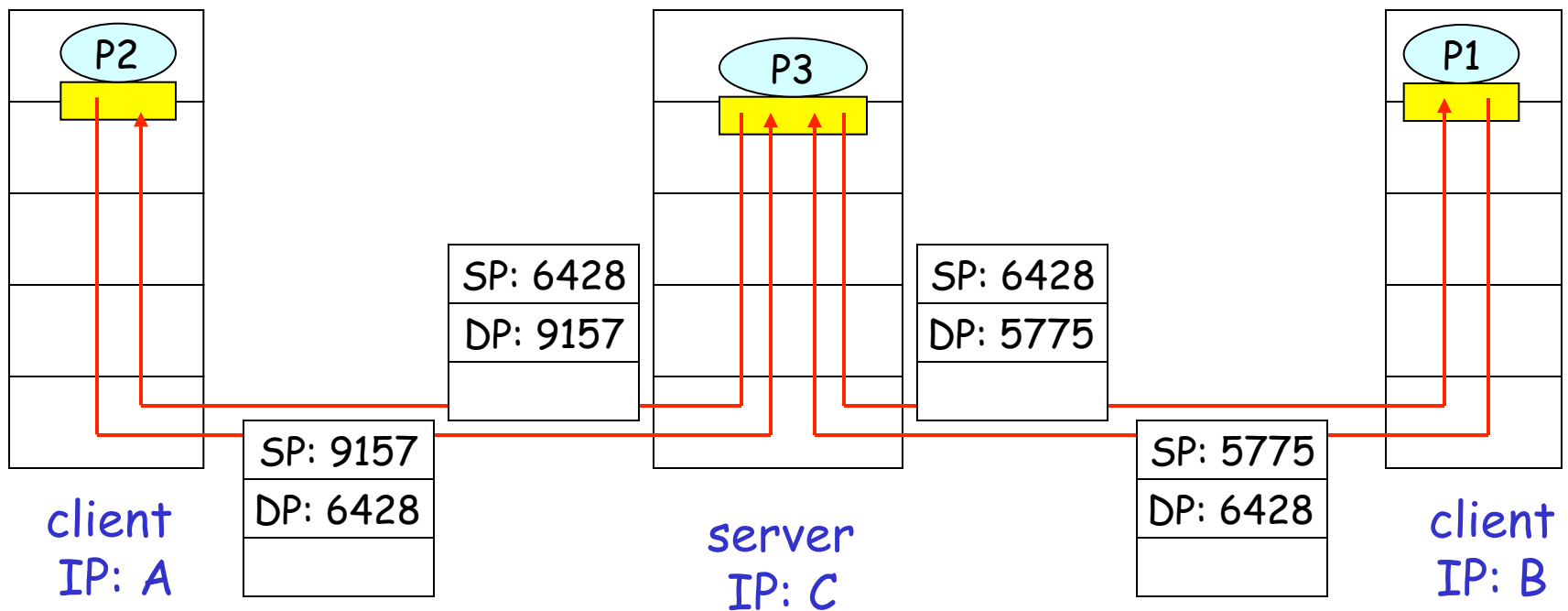
Demultiplexing senza connessione

- ❑ Crea le socket con i numeri di porta:
 - ❑ `DatagramSocket mySocket1 = new DatagramSocket(12534);`
 - ❑ `DatagramSocket mySocket2 = new DatagramSocket(12535);`
- ❑ La socket UDP è identificata da 2 parametri:
(indirizzo IP di destinazione,
numero della porta di destinazione)
- ❑ Quando l'host riceve il segmento UDP:
 - controlla il numero della porta di destinazione nel segmento
 - invia il segmento UDP alla socket con quel numero di porta
- ❑ Datagrammi IP con indirizzi IP di origine e/o numeri di porta di origine differenti vengono inviati alla stessa socket

Demultiplexing senza connessione

(continua)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

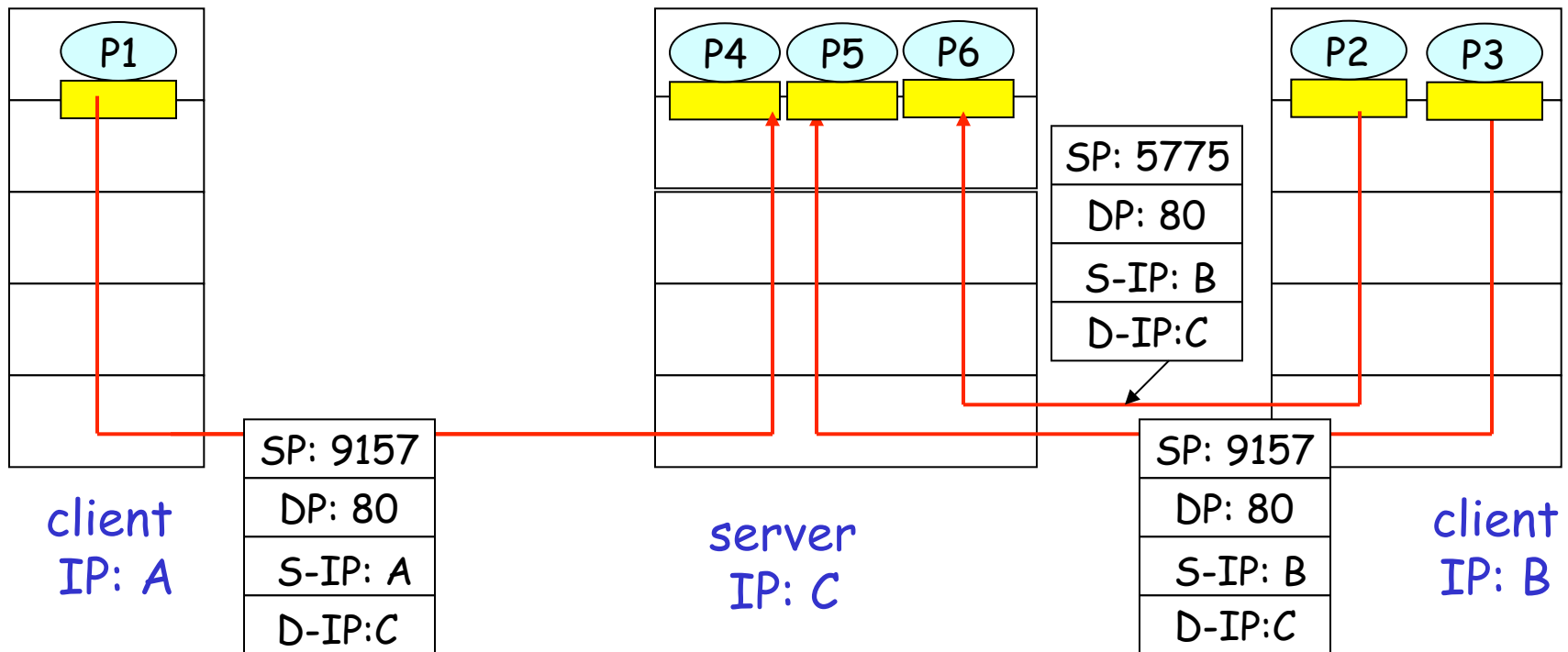


SP fornisce "l'indirizzo di ritorno"

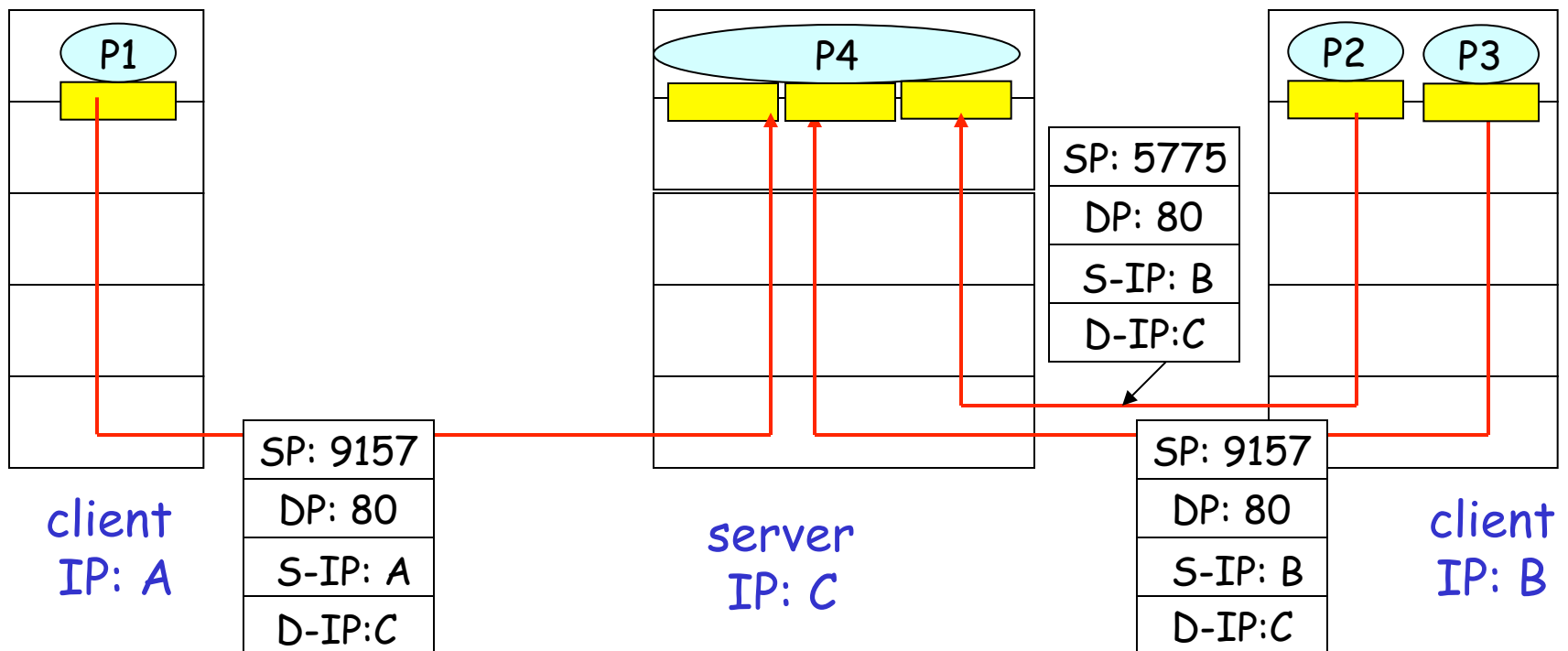
Demultiplexing orientato alla connessione

- ❑ La socket TCP è identificata da 4 parametri:
 - indirizzo IP di origine
 - numero di porta di origine
 - indirizzo IP di destinazione
 - numero di porta di destinazione
- ❑ L'host ricevente usa i quattro parametri per inviare il segmento alla socket appropriata
- ❑ Un host server può supportare più socket TCP contemporanee:
 - ogni socket è identificata dai suoi 4 parametri
- ❑ I server web hanno socket differenti per ogni connessione client
 - con HTTP non-persistente si avrà una socket differente per ogni richiesta

Demultiplexing orientato alla connessione (continua)



Demultiplexing orientato alla connessione: thread dei server web



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

UDP: User Datagram Protocol [RFC 768]

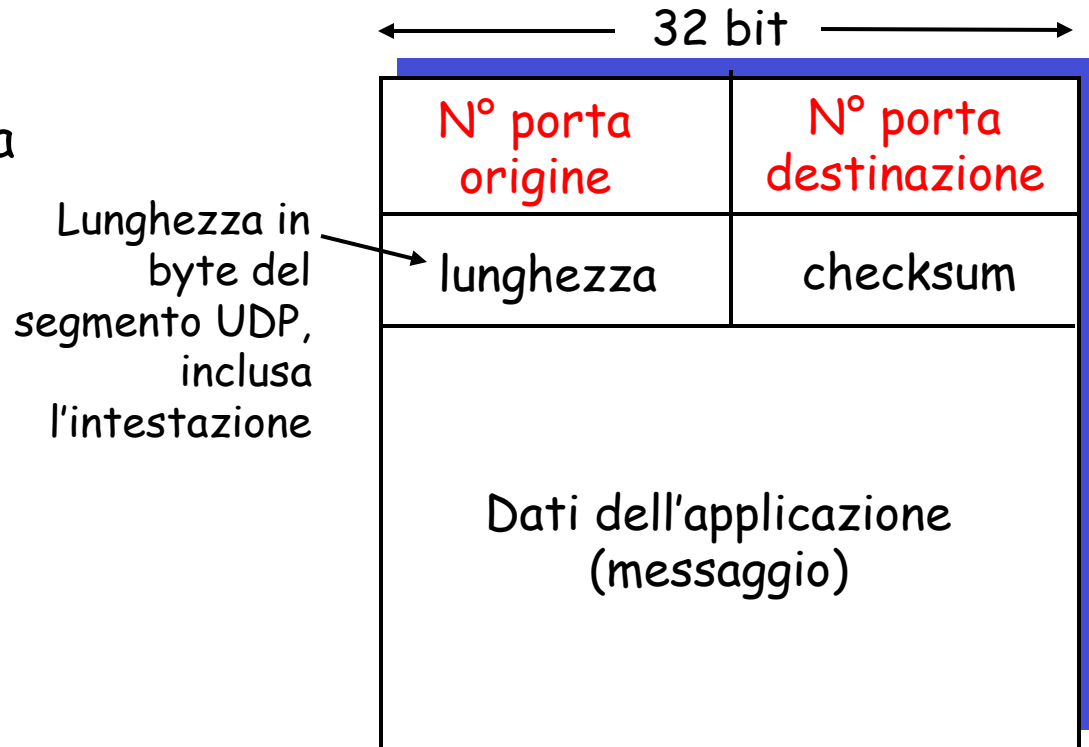
- ❑ Protocollo di trasporto "senza fronzoli"
- ❑ Servizio di consegna "a massimo sforzo", i segmenti UDP possono essere:
 - perduti
 - consegnati fuori sequenza all'applicazione
- ❑ *Senza connessione:*
 - no handshaking tra mittente e destinatario UDP
 - ogni segmento UDP è gestito indipendentemente dagli altri

Perché esiste UDP?

- ❑ Nessuna connessione stabilita (che potrebbe aggiungere un ritardo)
- ❑ Semplice: nessuno stato di connessione nel mittente e destinatario
- ❑ Intestazioni di segmento corte
- ❑ Senza controllo di congestione: UDP può sparare dati a raffica

UDP: ulteriori informazioni

- ❑ Utilizzato spesso nelle applicazioni multimediali
 - tollera piccole perdite
 - sensibile alla frequenza
- ❑ Altri impieghi di UDP
 - DNS
 - SNMP
- ❑ Trasferimento affidabile con UDP: aggiungere affidabilità al livello di applicazione
 - Recupero degli errori delle applicazioni!



Struttura del segmento UDP

Checksum UDP

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

Mittente:

- ❑ Tratta il contenuto del segmento come una sequenza di interi da 16 bit
- ❑ checksum: somma (complemento a 1) i contenuti del segmento
- ❑ Il mittente pone il valore della checksum nel campo checksum del segmento UDP

Ricevente:

- ❑ calcola la checksum del segmento ricevuto
- ❑ controlla se la checksum calcolata è uguale al valore del campo checksum:
 - No - errore rilevato
 - Sì - nessun errore rilevato. *Ma potrebbero esserci errori nonostante questo? Lo scopriremo più avanti ...*

Esempio di checksum

□ Nota

- Quando si sommano i numeri, un riporto dal bit più significativo deve essere sommato al risultato

□ Esempio: sommare due interi da 16 bit

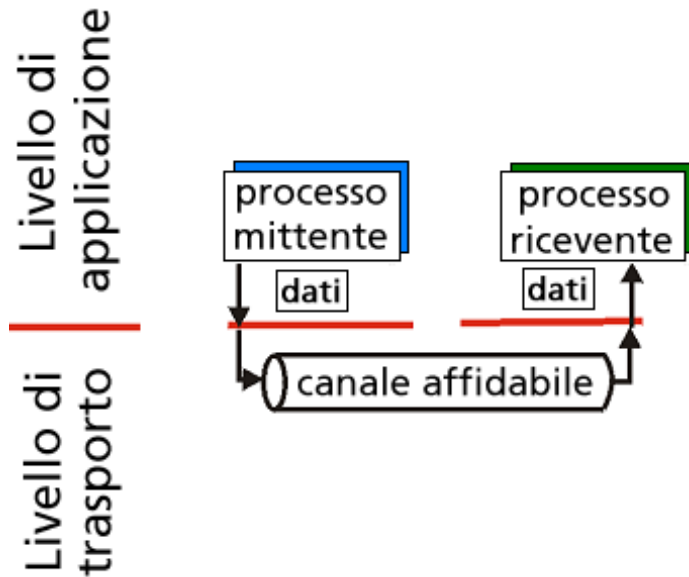
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																	
a capo	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
somma		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

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!

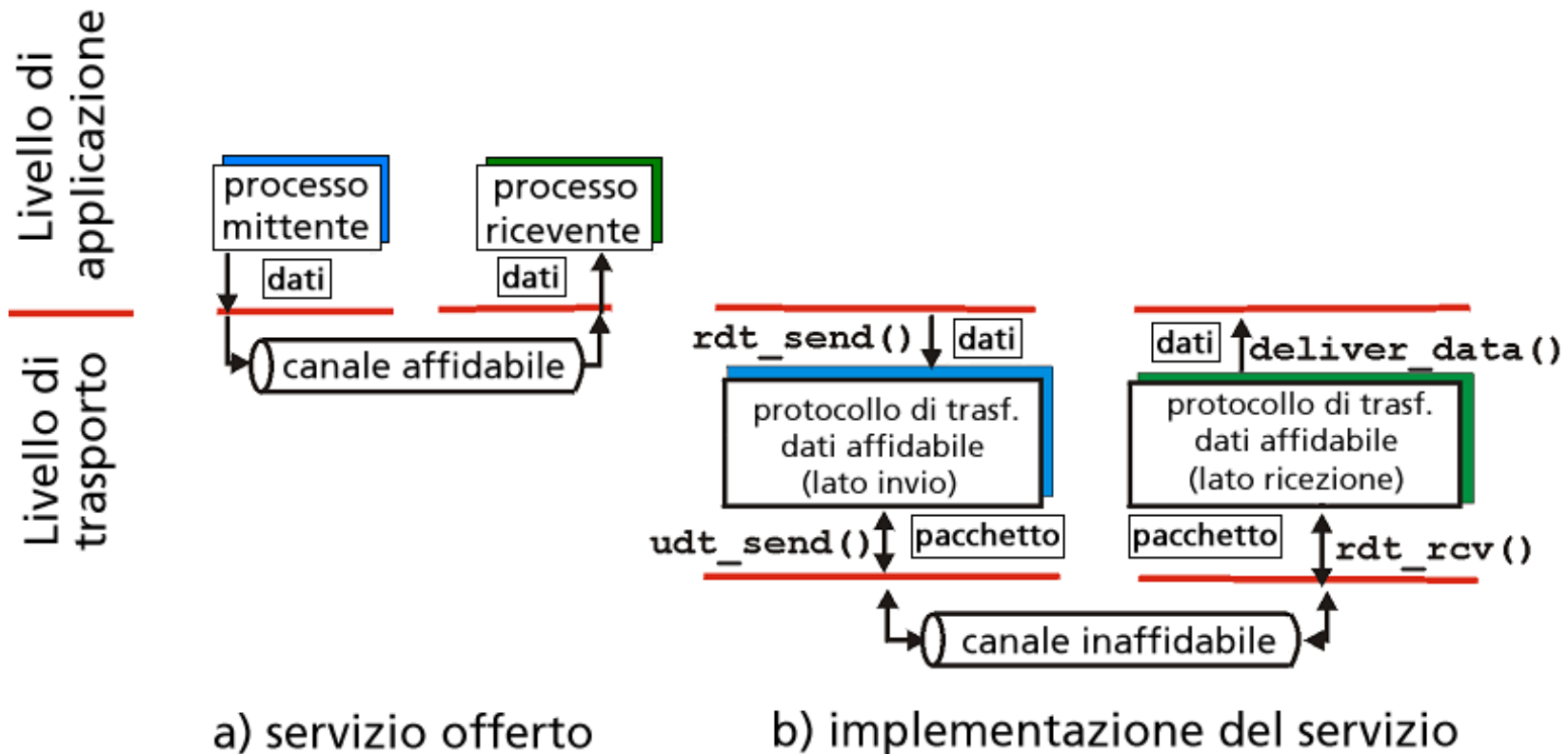


a) servizio offerto

- ❑ 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!

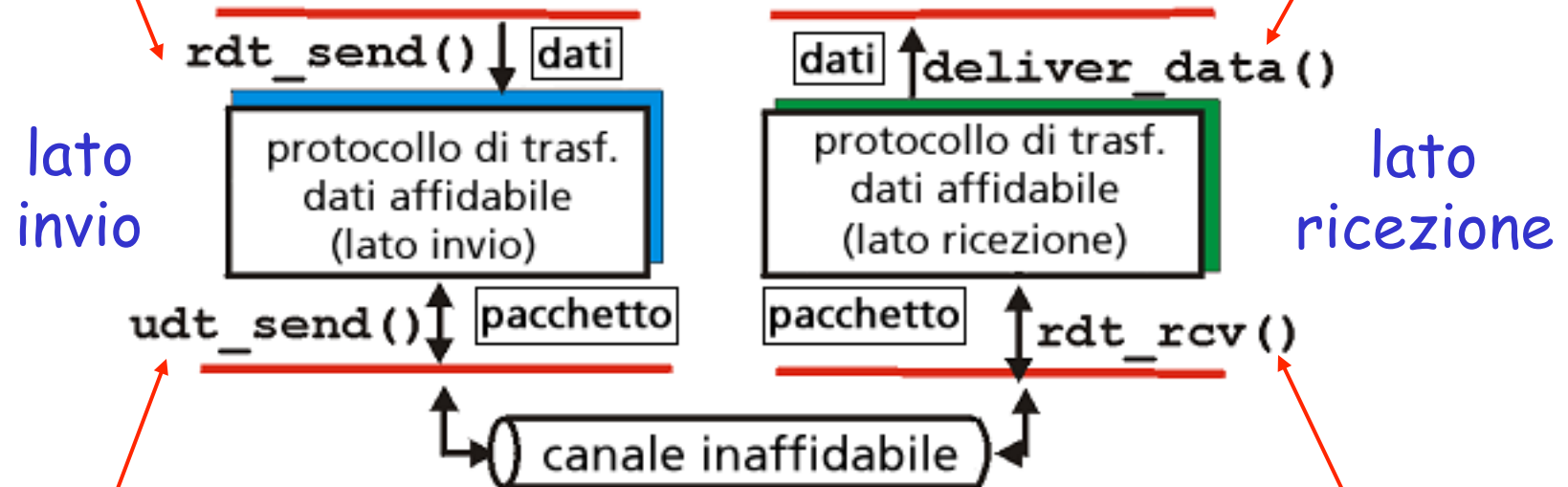


- ❑ Le caratteristiche del canale inaffidabile determinano la complessità del protocollo di trasferimento dati affidabile (reliable data transfer o rdt)

Trasferimento dati affidabile: preparazione

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

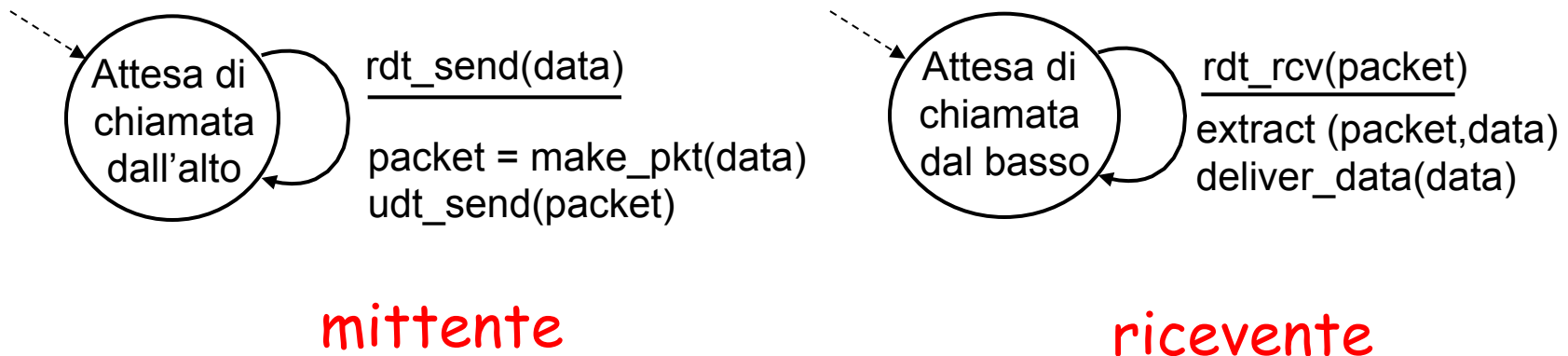
Trasferimento dati affidabile: preparazione

- ❑ 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



Rdt1.0: trasferimento affidabile su canale affidabile

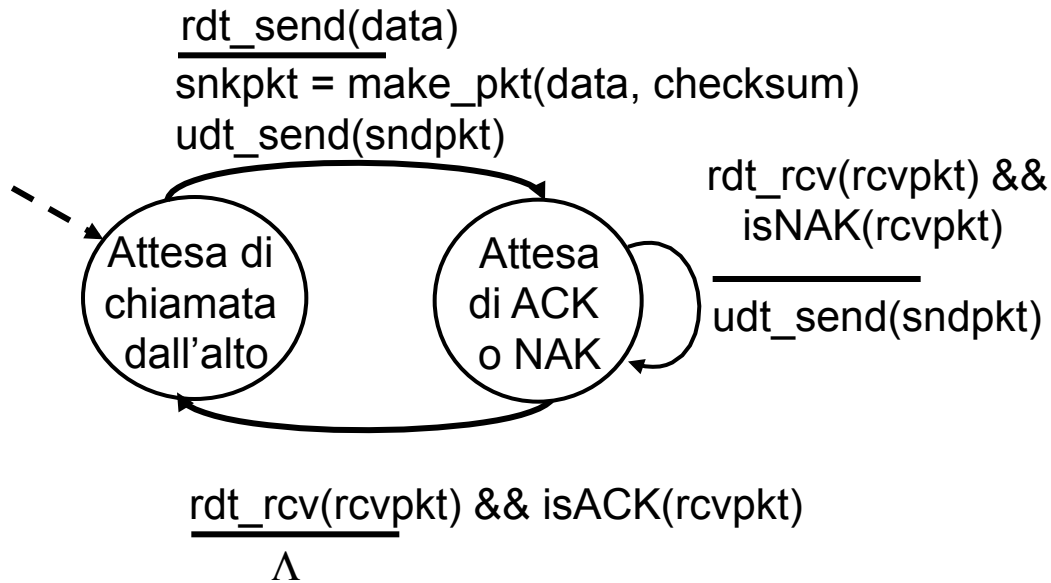
- ❑ Canale sottostante perfettamente affidabile
 - Nessun errore nei bit
 - Nessuna perdita di pacchetti
- ❑ Automa distinto per il mittente e per il ricevente:
 - il mittente invia i dati nel canale sottostante
 - il ricevente legge i dati dal canale sottostante



Rdt2.0: canale con errori nei bit

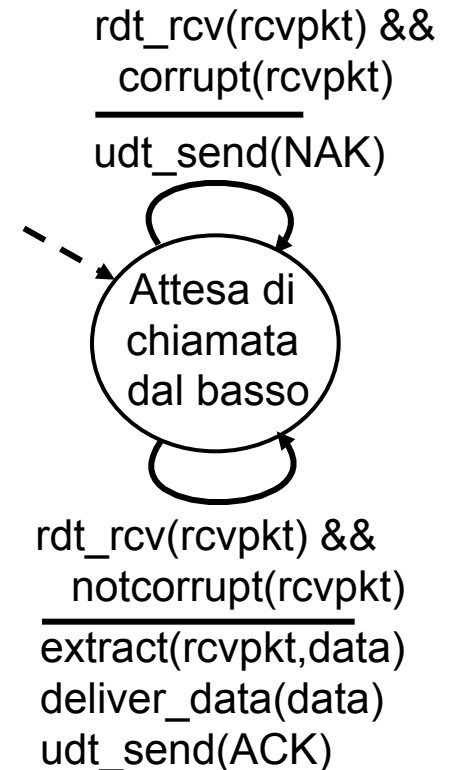
- ❑ Il canale sottostante potrebbe confondere i bit nei pacchetti
 - checksum per rilevare gli errori nei bit
- ❑ *domanda*: come correggere gli errori:
 - *notifica positiva (ACK)*: il ricevente comunica espressamente al mittente che il pacchetto ricevuto è corretto
 - *notifica negativa (NAK)*: il 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):
 - rilevamento di errore
 - feedback del destinatario: messaggi di controllo (ACK, NAK) ricevente→mittente

rdt2.0: specifica dell'automa

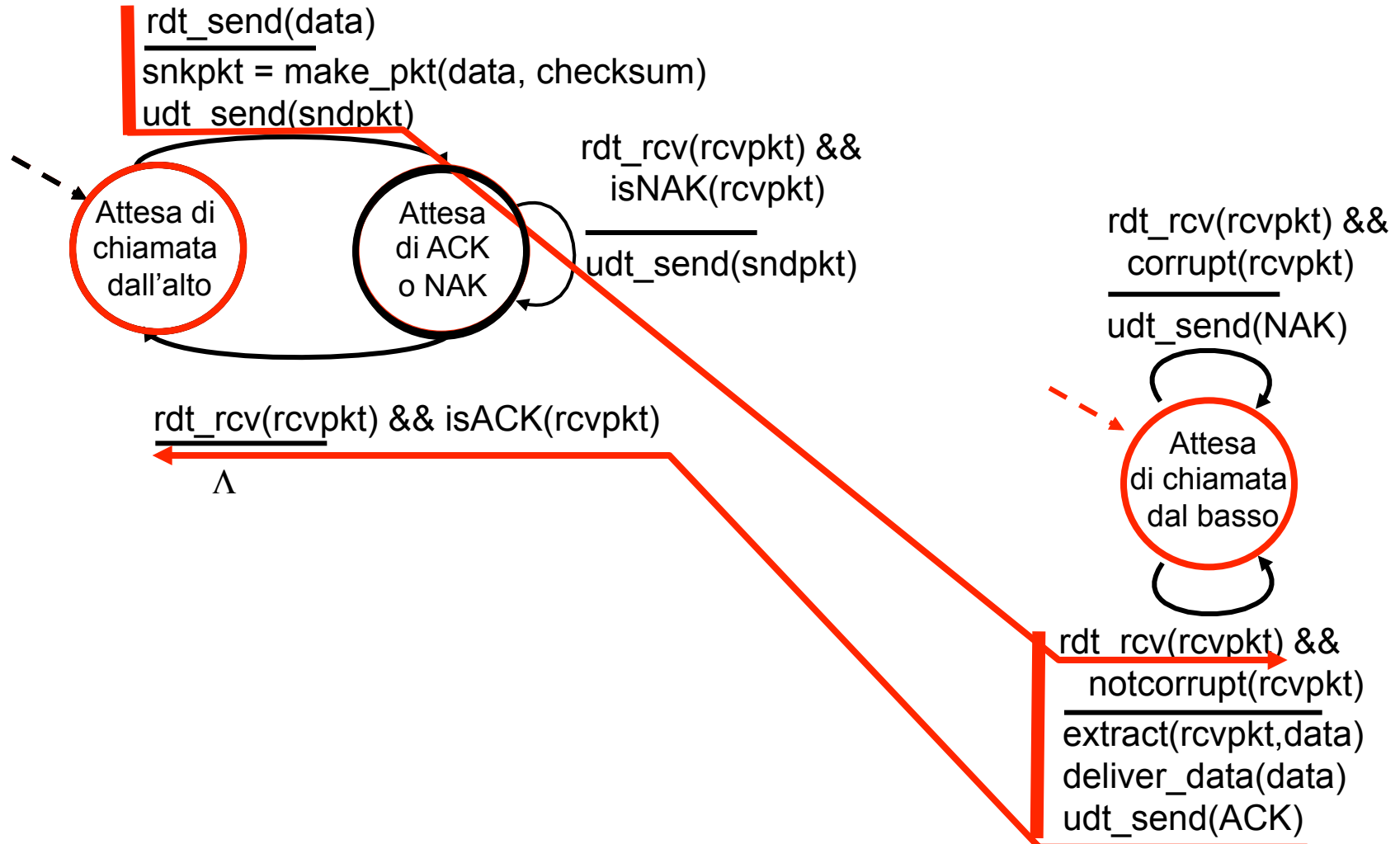


mittente

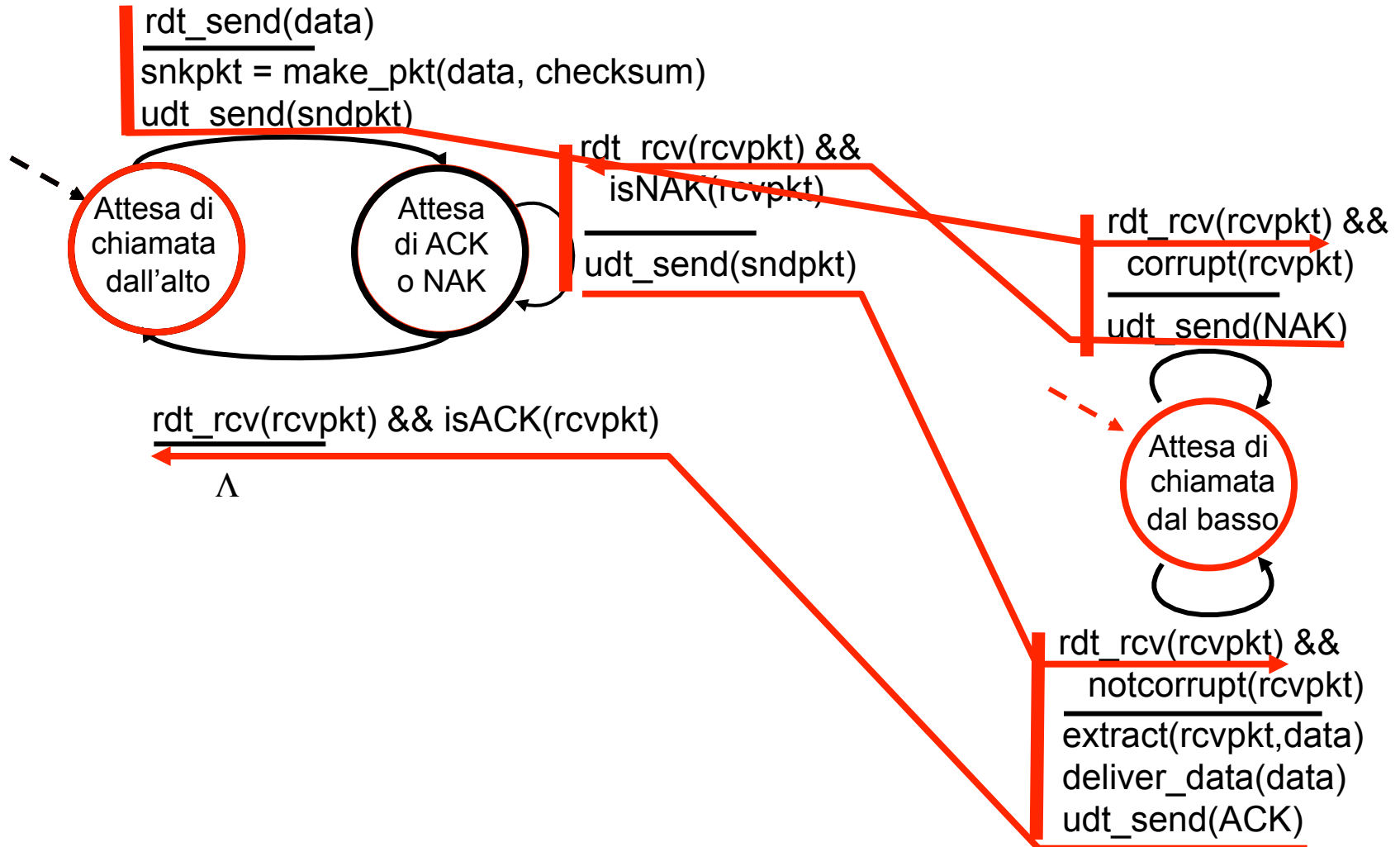
ricevente



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

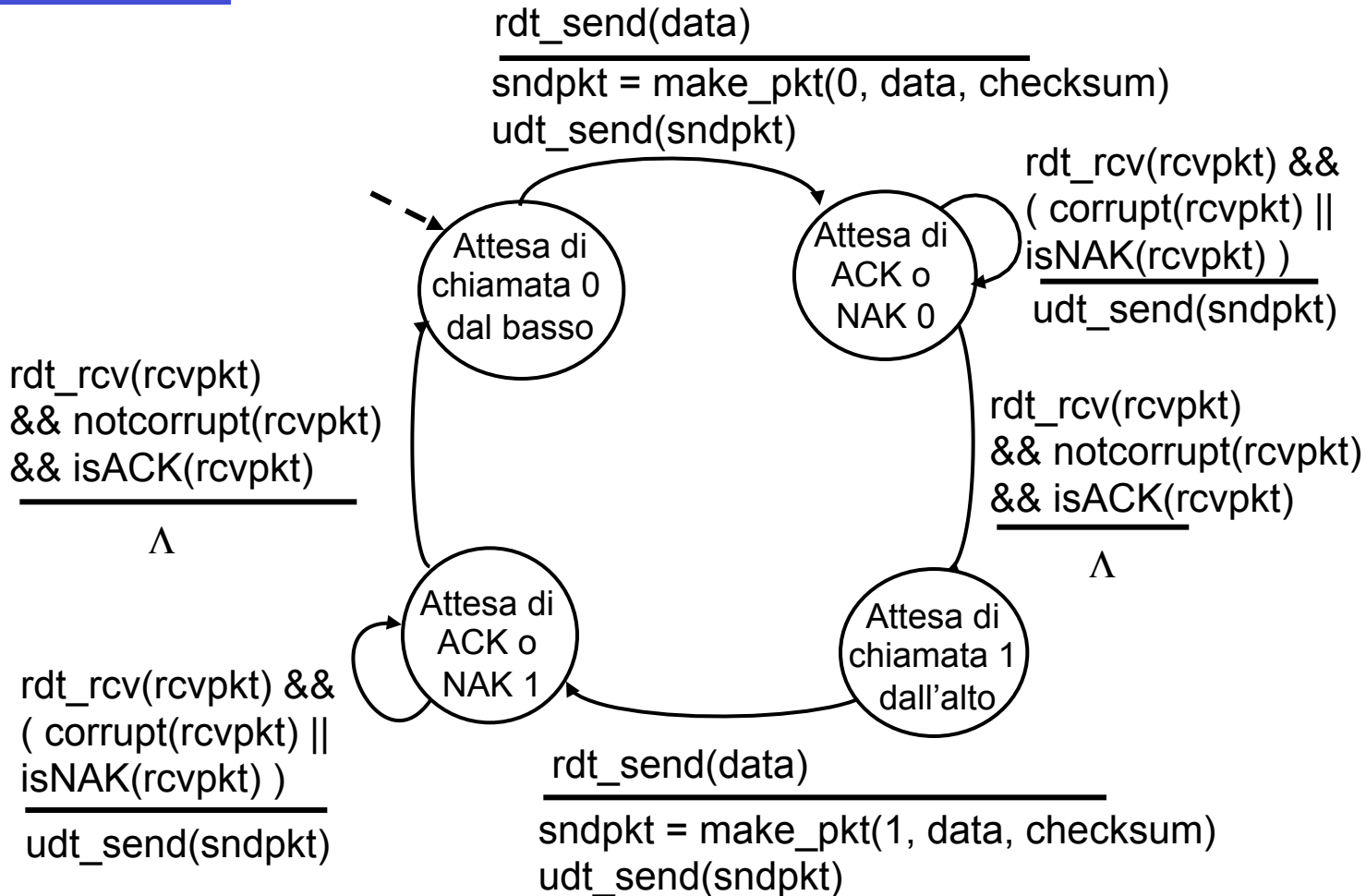
Gestione dei duplicati:

- ❑ 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

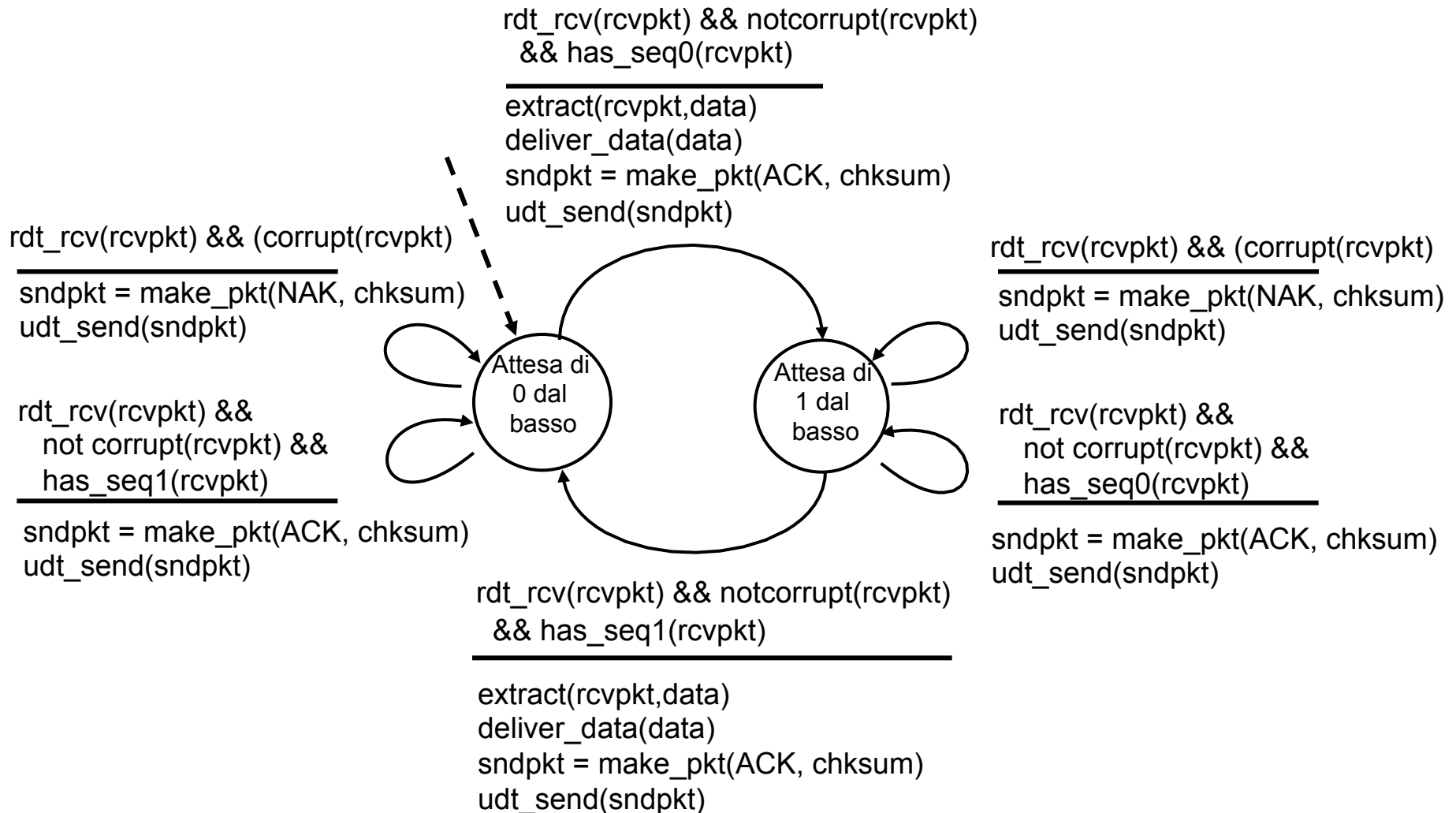
— stop and wait —

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). 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

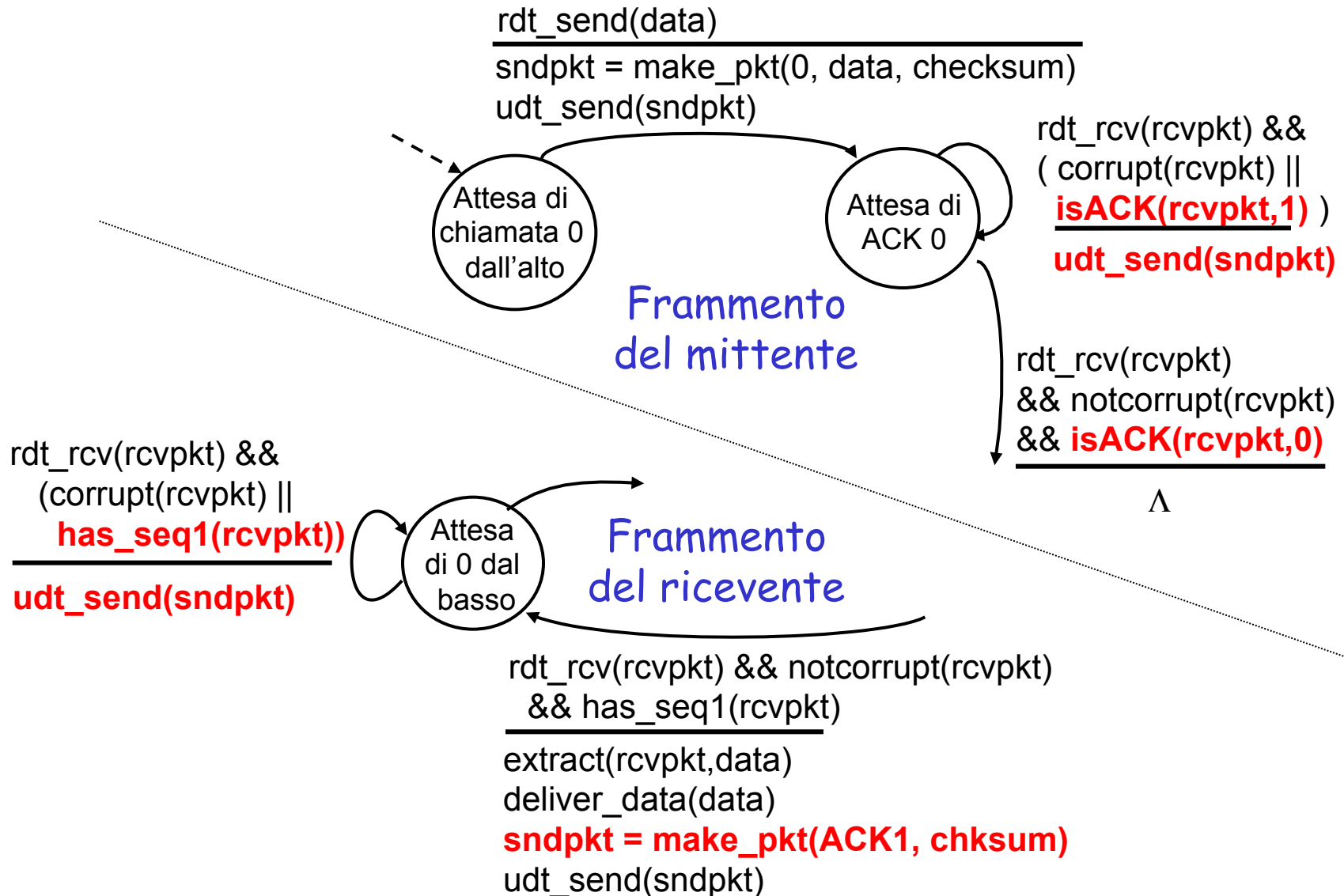
Ricevente:

- ❑ 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

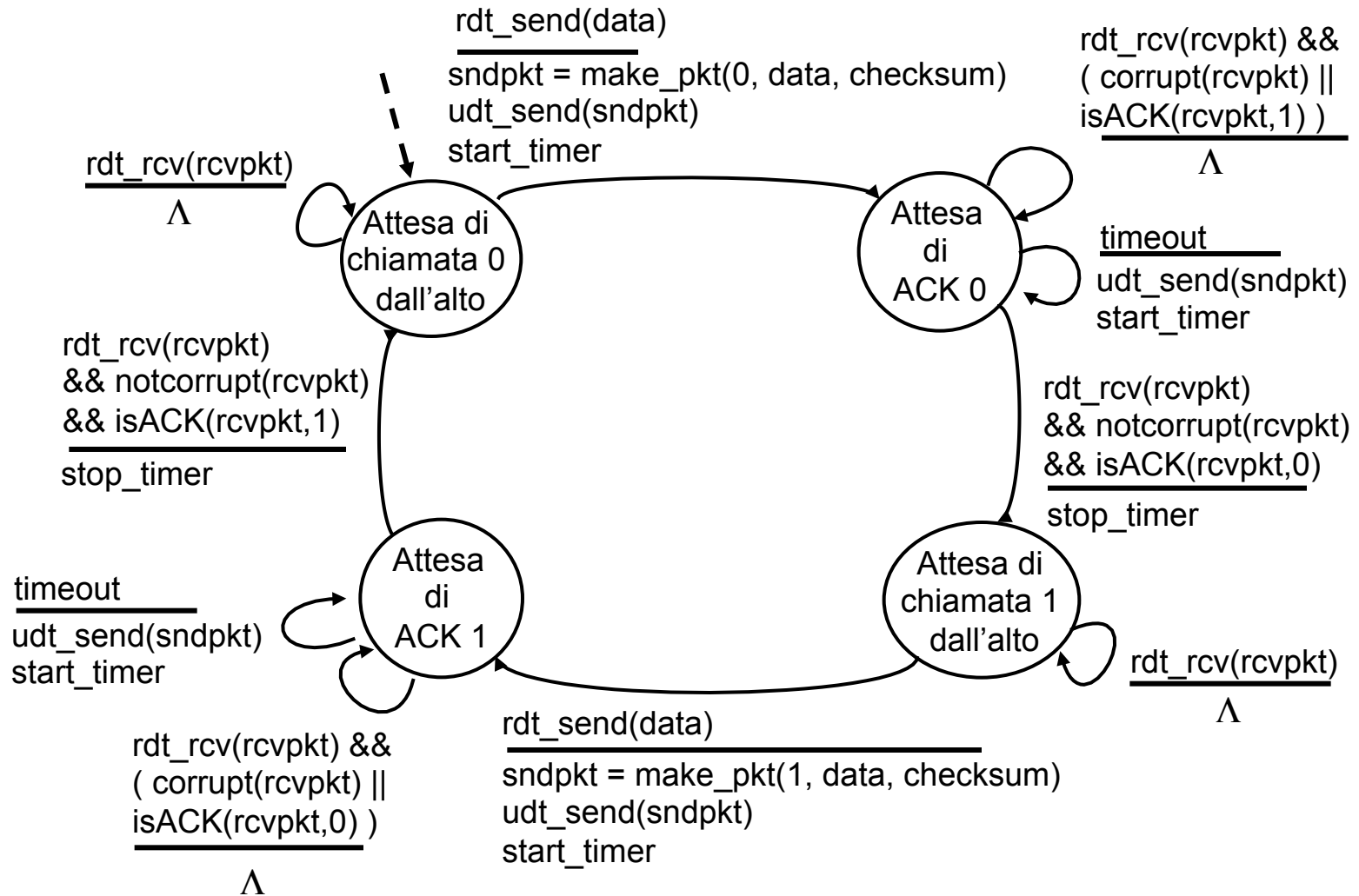
Nuova ipotesi: il canale sottostante può anche smarrire i pacchetti (dati o ACK)

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

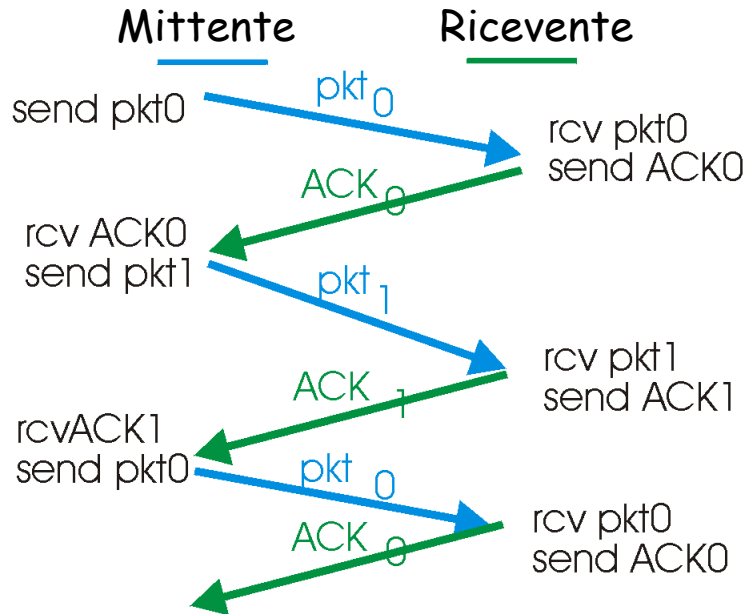
Approccio: il mittente attende un ACK per un tempo "ragionevole"

- 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*)

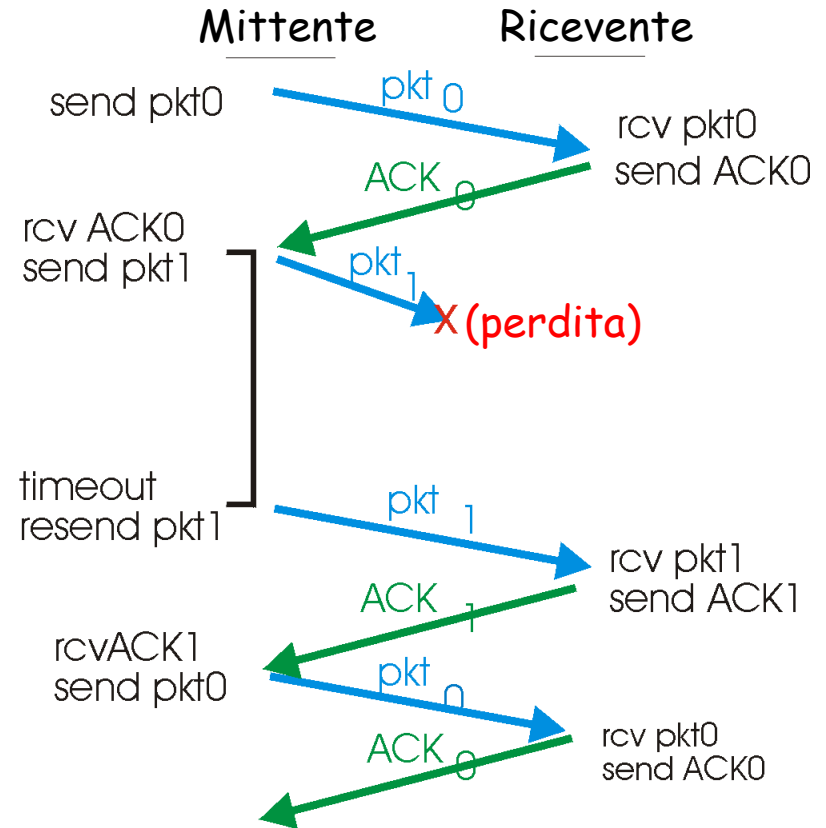
rdt3.0 mittente



rdt3.0 in azione

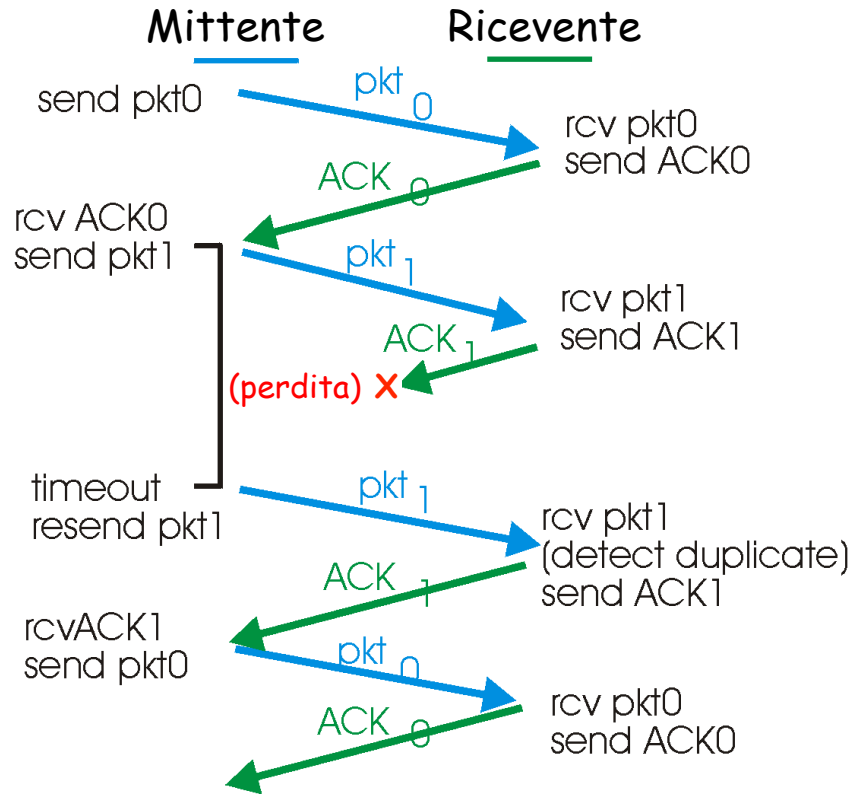


a) Operazioni senza perdite

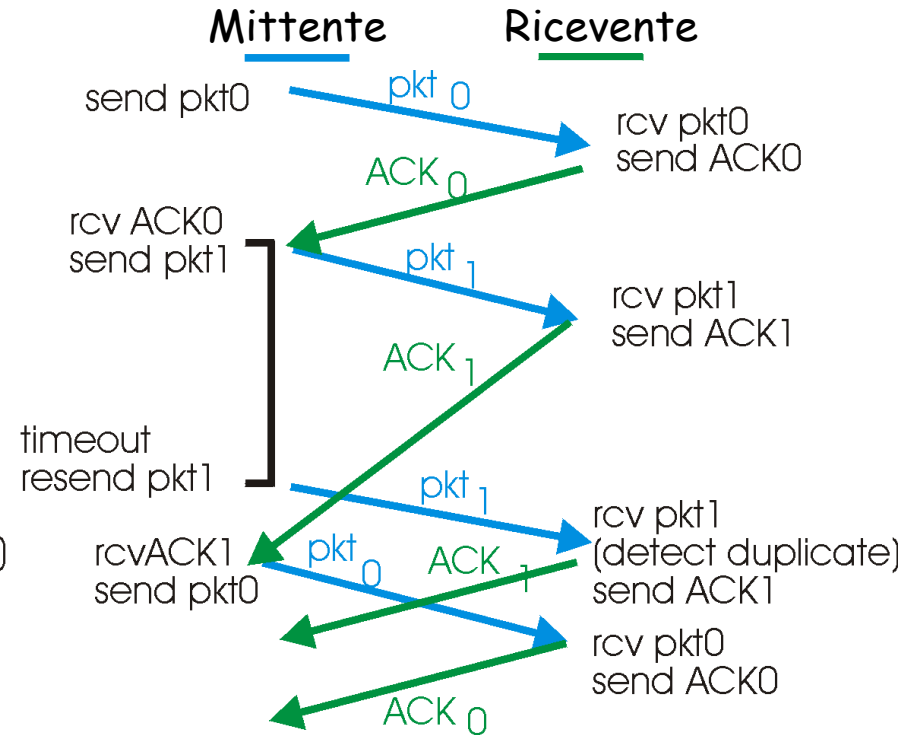


b) Perdita di pacchetto

rdt3.0 in azione



c) Perdita di ACK



d) Timeout prematuro

Prestazioni di rdt3.0

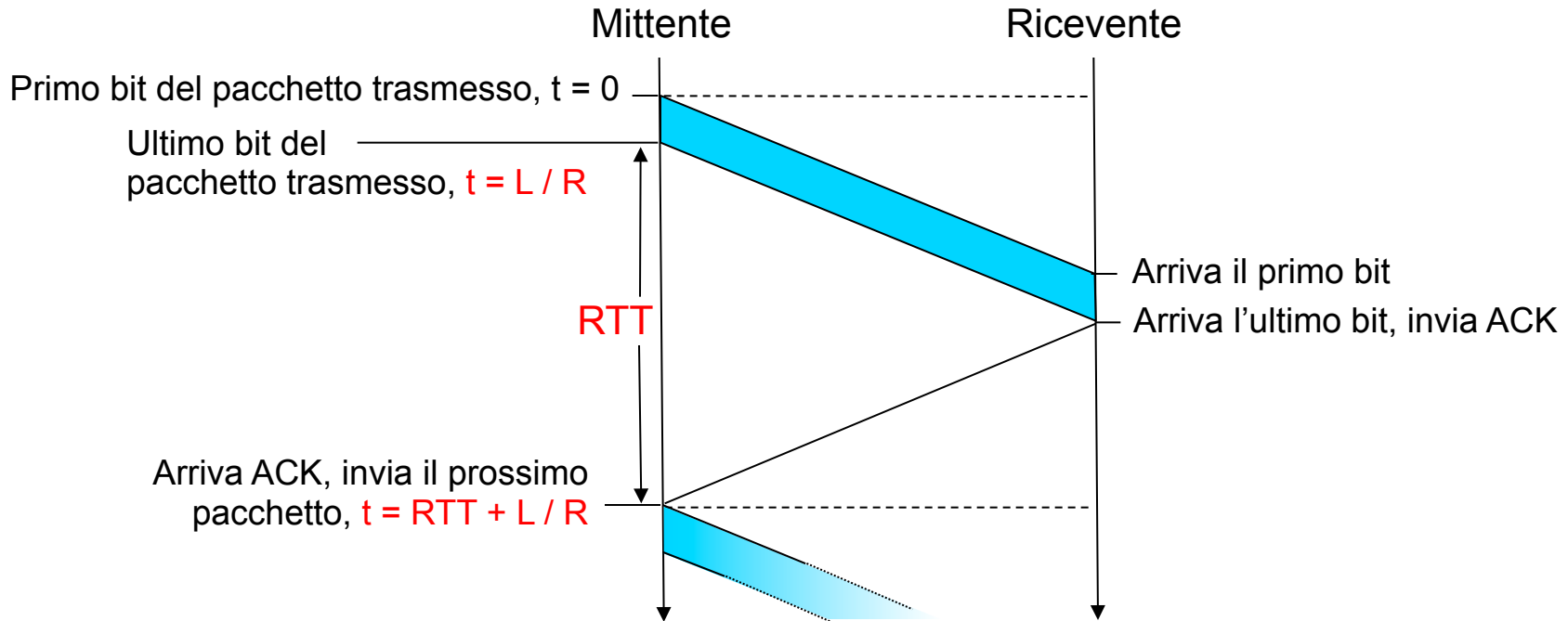
- ❑ rdt3.0 funziona, ma le prestazioni non sono apprezzabili
- ❑ esempio: collegamento da 1 Gbps, ritardo di propagazione 15 ms, pacchetti da 1 KB:

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

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

- 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

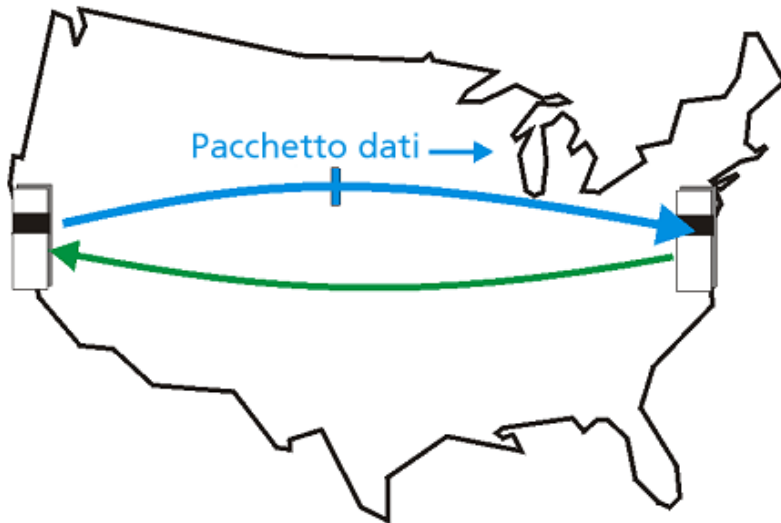


$$U_{\text{mitt}} = \frac{L / R}{RTT + L / R} = \frac{0,008}{30,008} = 0,00027 \text{ microsec}$$

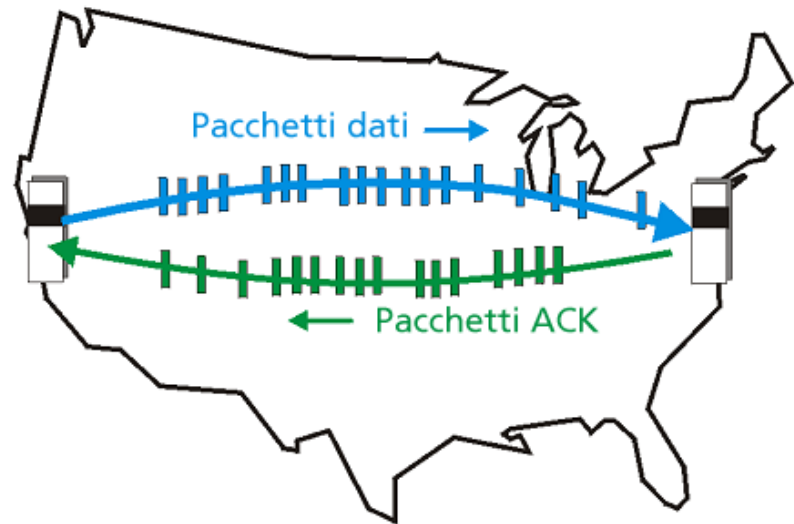
Protocolli con pipeline

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



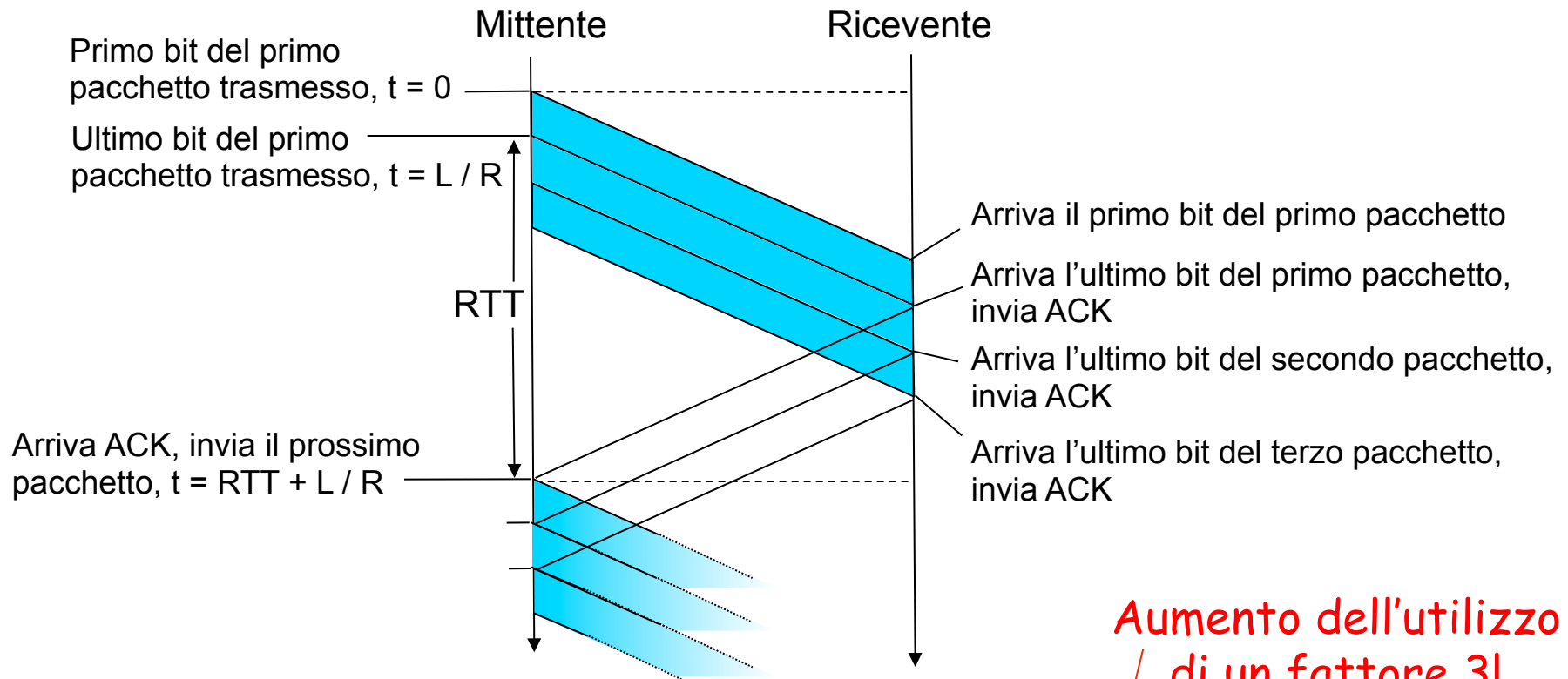
a) Protocollo stop-and-wait all'opera



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!**

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

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 solo 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

Ripetizione selettiva

- ❑ Il mittente può avere fino a N pacchetti nella pipeline che non hanno ancora ricevuto ACK
- ❑ Il ricevente accusa ricevuta di ciascun singolo pacchetto
- ❑ Il mittente mantiene un timer per ciascun pacchetto che non ha ancora avuto ACK
 - Quando il timer scade, ritrasmette solo i pacchetti non riscontrati

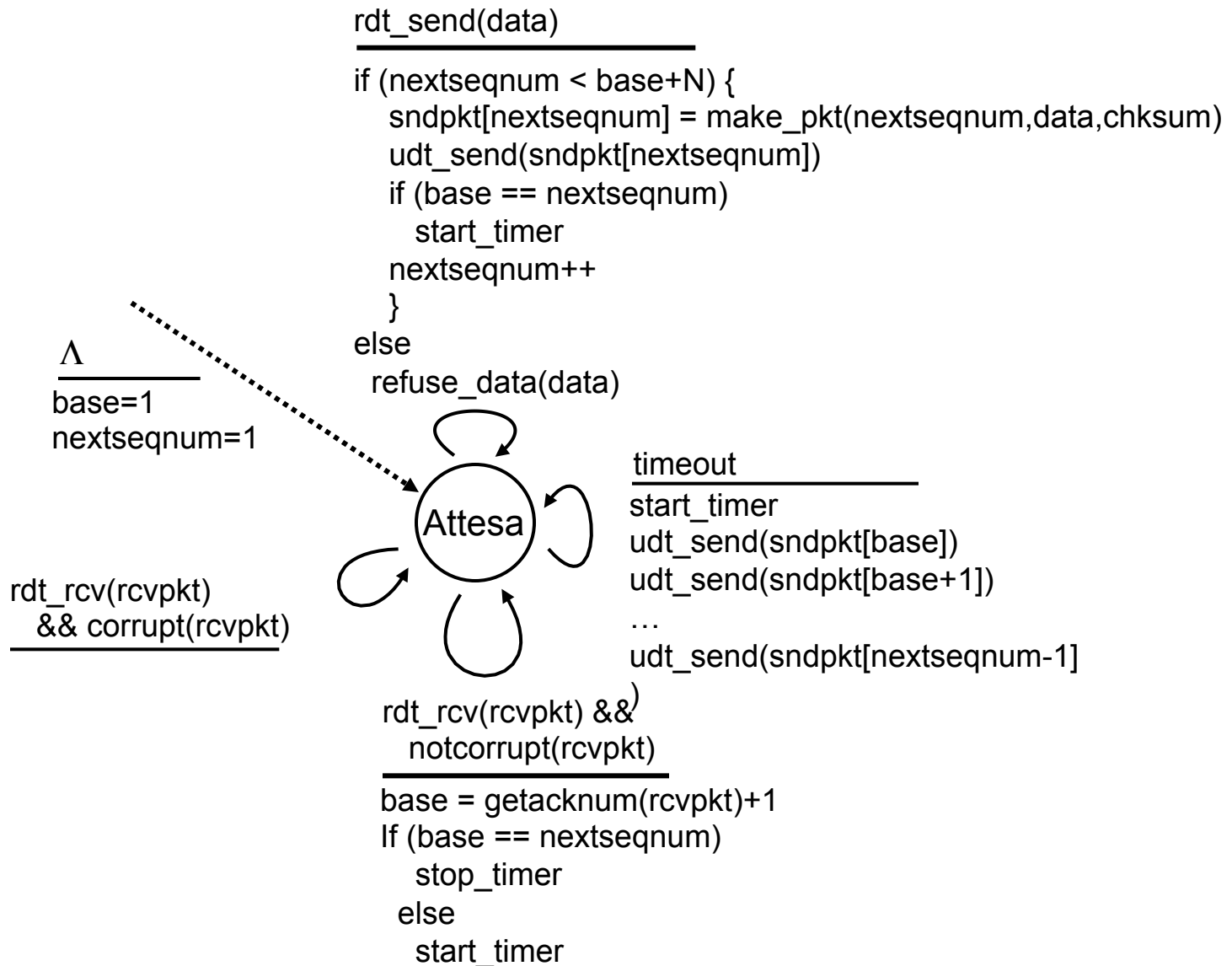
Go-Back-N

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

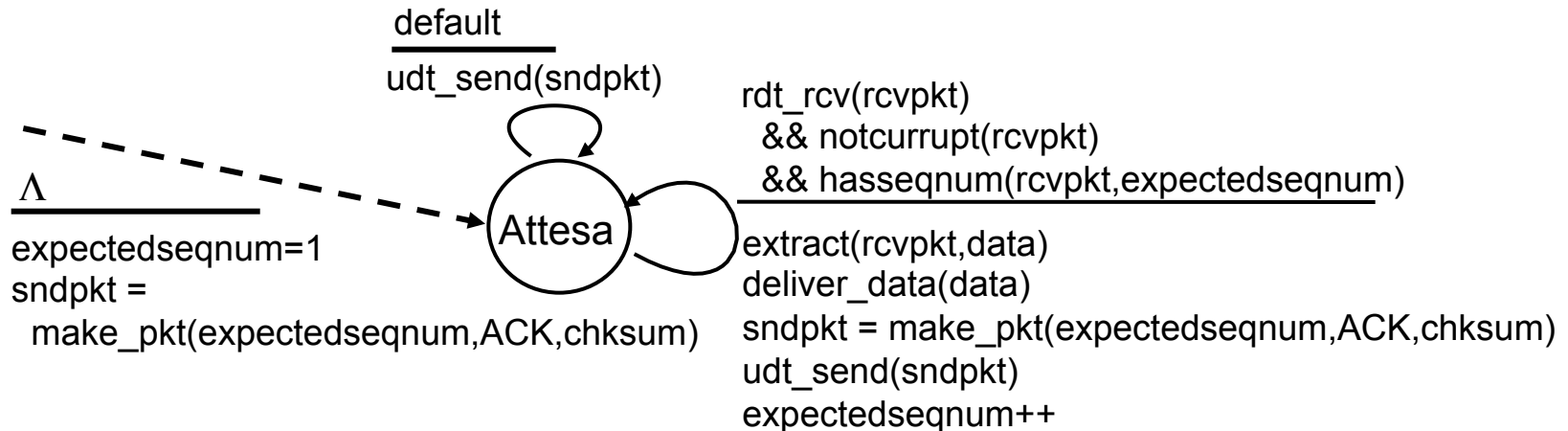


- ❑ $ACK(n)$: riscontro di tutti i pacchetti con numero di sequenza minore o uguale a n - "riscontri cumulativi"
 - pacchetti duplicati potrebbero essere scartati (vedere il ricevente)
- ❑ timer per ogni pacchetto in transito
- ❑ $timeout(n)$: ritrasmette il pacchetto n e tutti i pacchetti con i numeri di sequenza più grandi nella finestra

GBN: automa esteso del mittente



GBN: automa esteso del ricevente



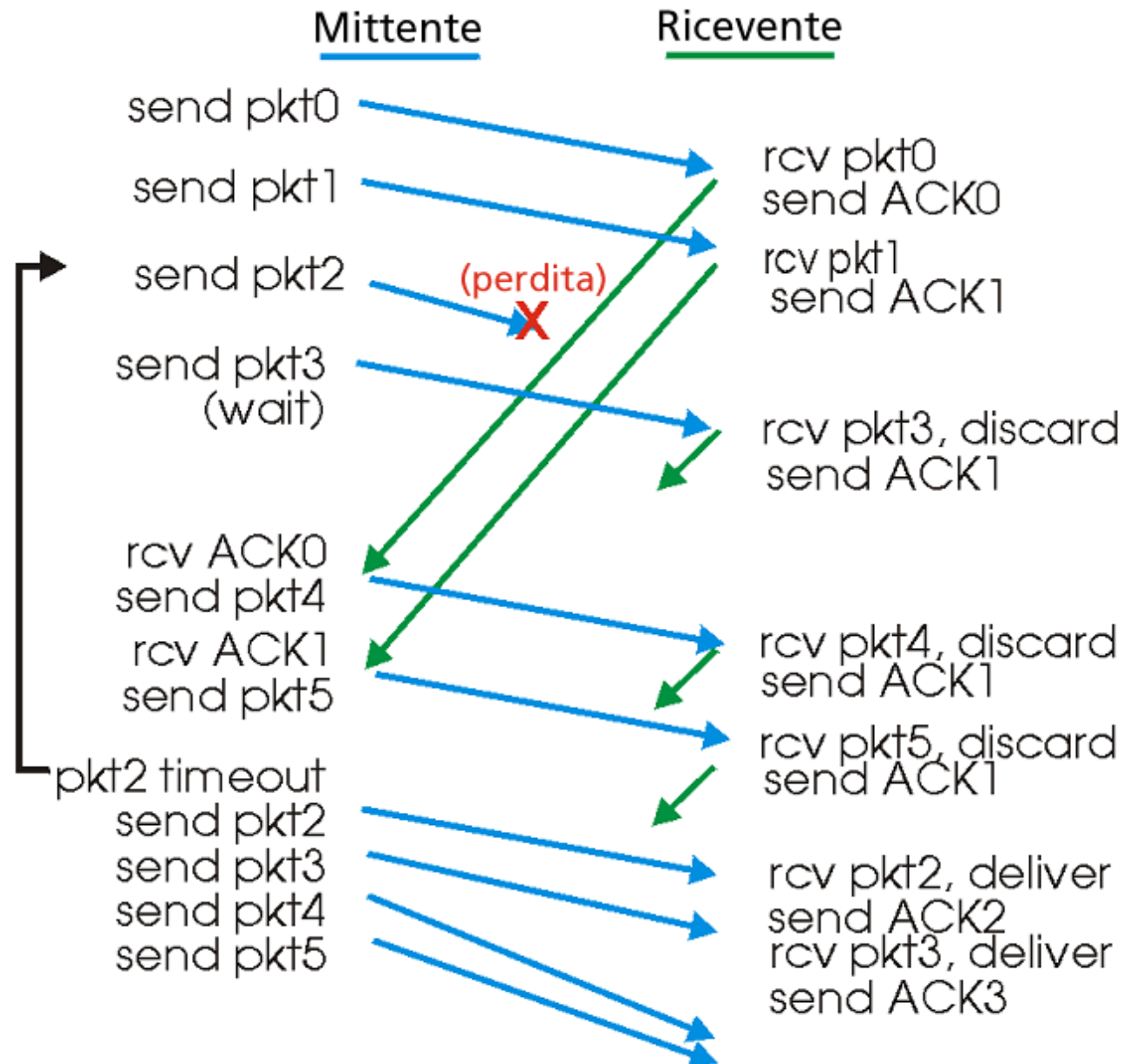
ACK-soltanto: invia sempre un ACK per un pacchetto ricevuto correttamente con il numero di sequenza più alto *in sequenza*

- potrebbe generare ACK duplicati
- deve memorizzare soltanto **expectedseqnum**

❑ 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*

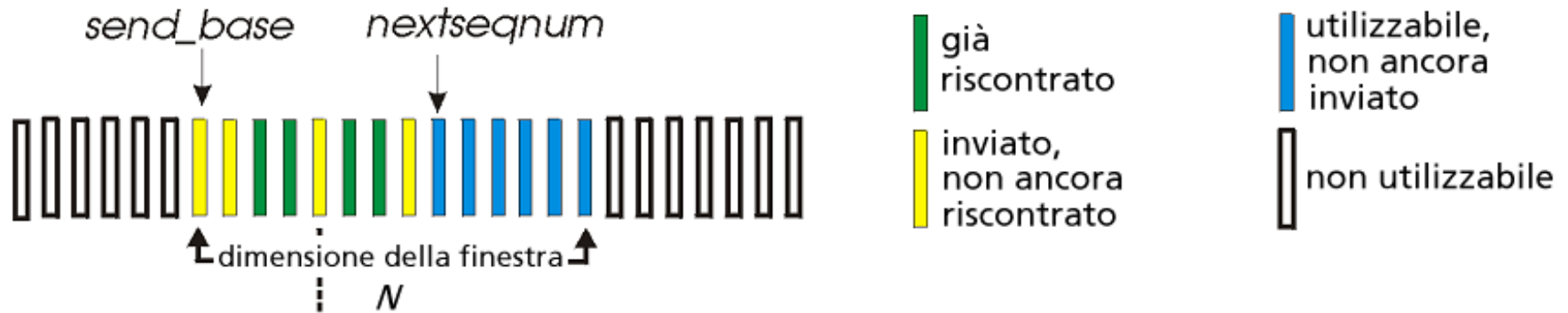
GBN in azione



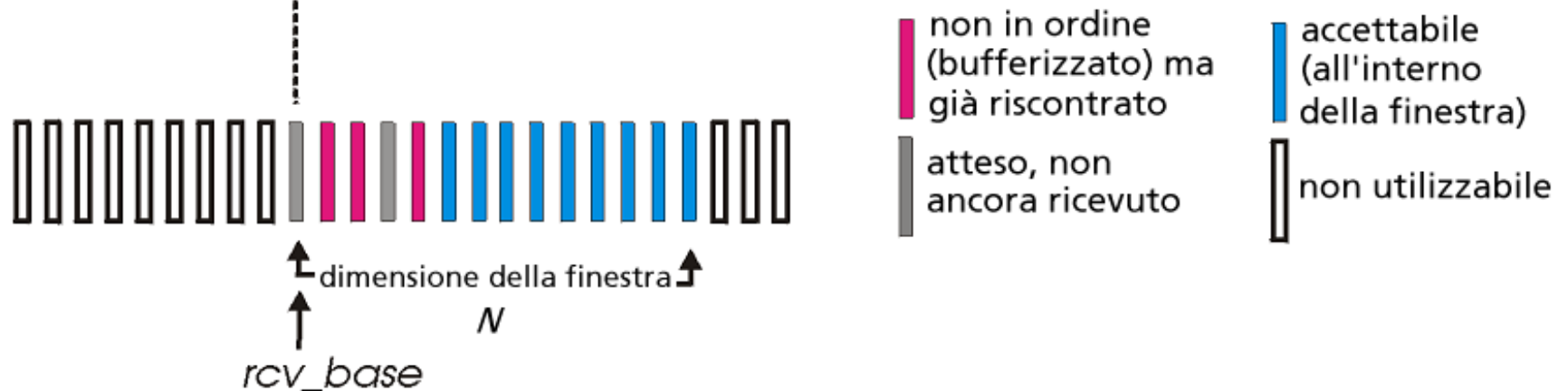
Ripetizione selettiva

- ❑ 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



a) Visione del mittente sui numeri di sequenza



b) Visione del ricevente sui numeri di sequenza

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

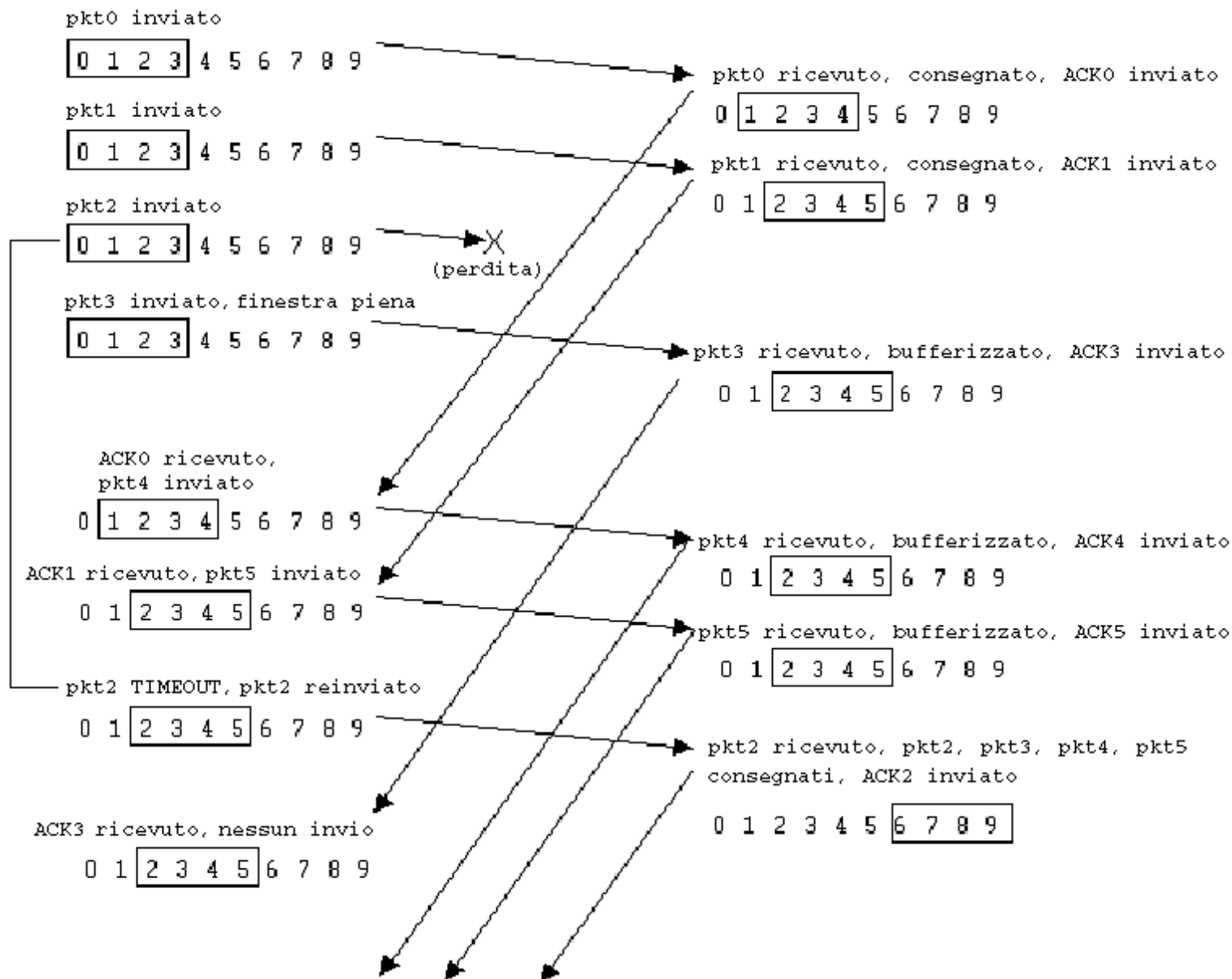
Pacchetto n in [rcvbase-N, rcvbase-1]

- ACK(n)

altrimenti:

- ignora

Ripetizione selettiva in azione



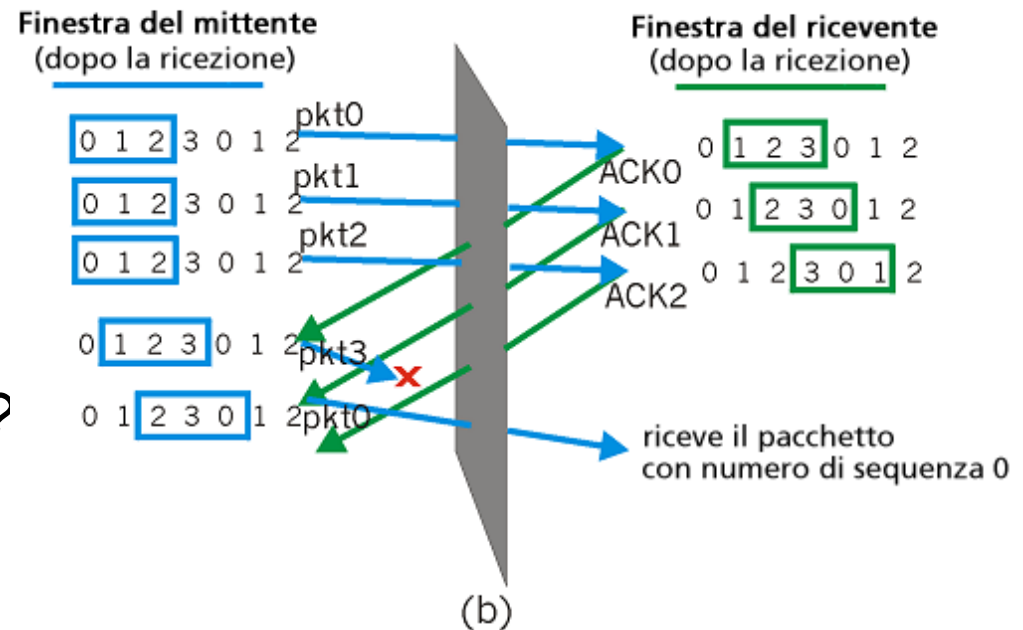
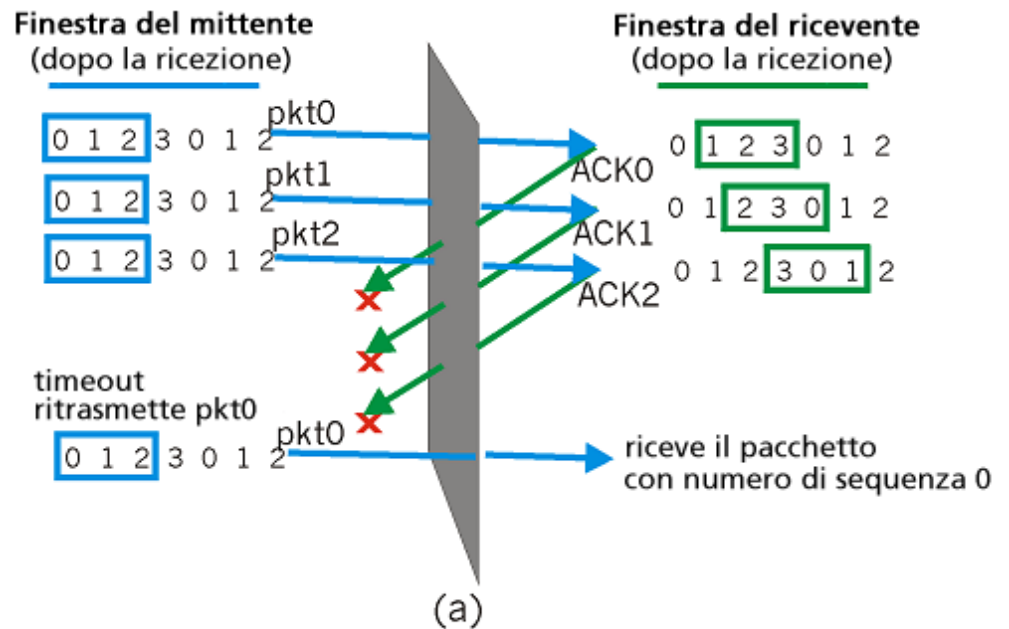
Ripetizione selettiva: dilemma

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?



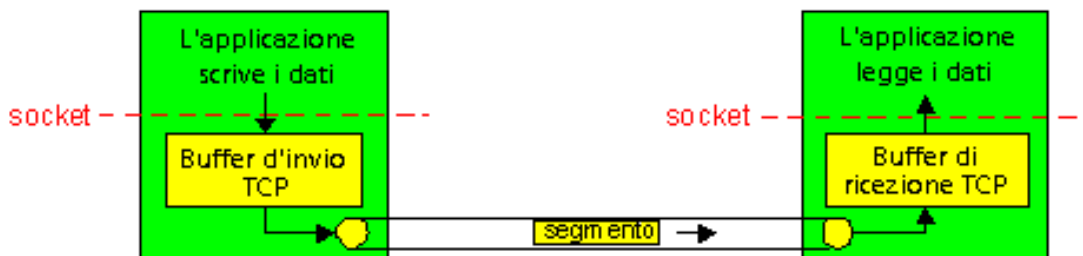
Capitolo 3: Livello di trasporto

- ❑ 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

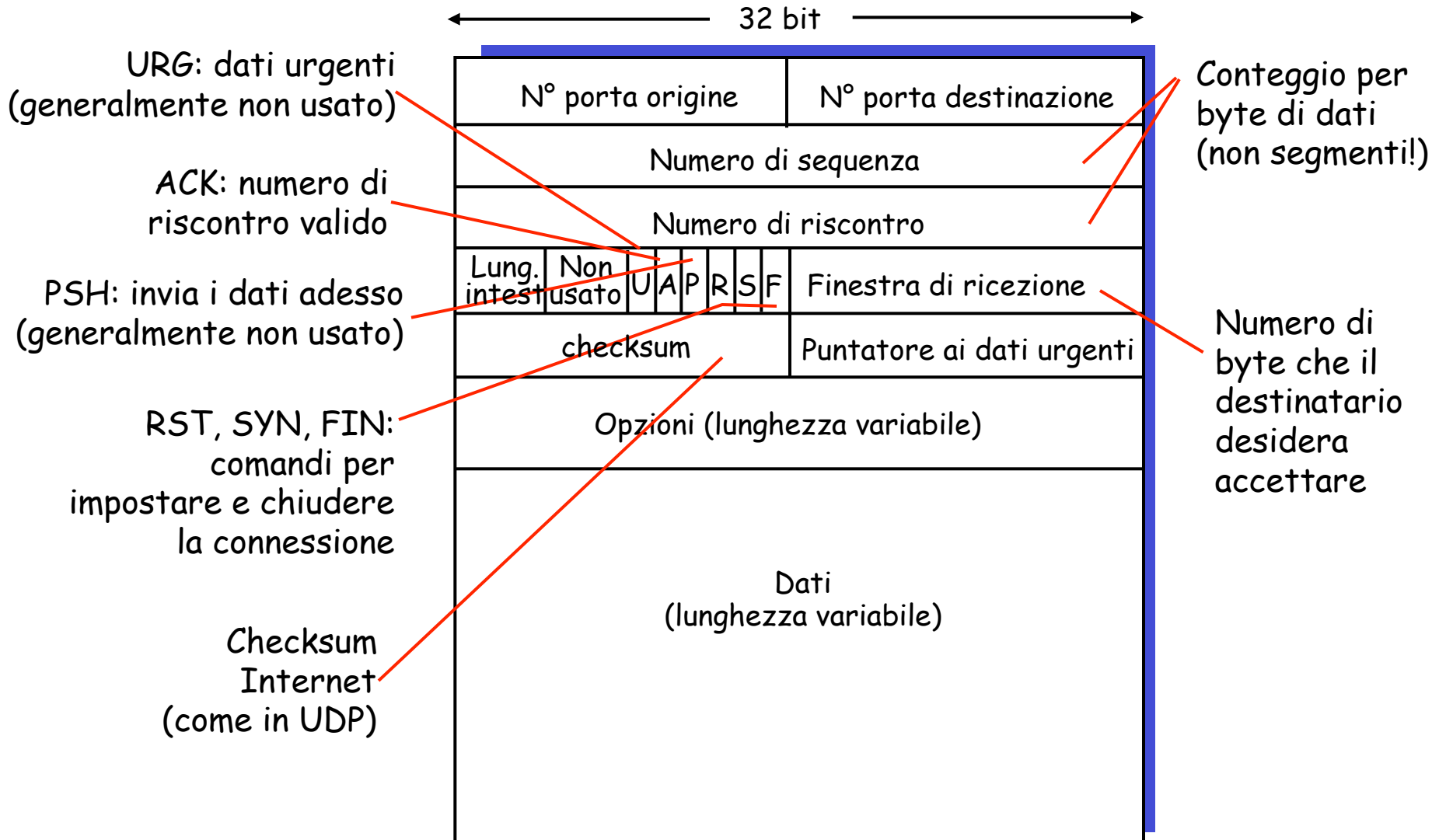
TCP: Panoramica

RFC: 793, 1122, 1323, 2018, 2581

- ❑ **punto-punto:**
 - un mittente, un destinatario
- ❑ **flusso di byte affidabile, in sequenza:**
 - nessun "confine ai messaggi"
- ❑ **pipeline:**
 - il controllo di flusso e di congestione TCP definiscono la dimensione della finestra
- ❑ **buffer d'invio e di ricezione**
- ❑ **full duplex:**
 - flusso di dati bidirezionale nella stessa connessione
 - MSS: dimensione massima di segmento (maximum segment size)
- ❑ **orientato alla connessione:**
 - l'handshaking (scambio di messaggi di controllo) inizializza lo stato del mittente e del destinatario prima di scambiare i dati
- ❑ **flusso controllato:**
 - il mittente non sovraccarica il destinatario



Struttura dei segmenti TCP



Numeri di sequenza e ACK di TCP

Numeri di sequenza:

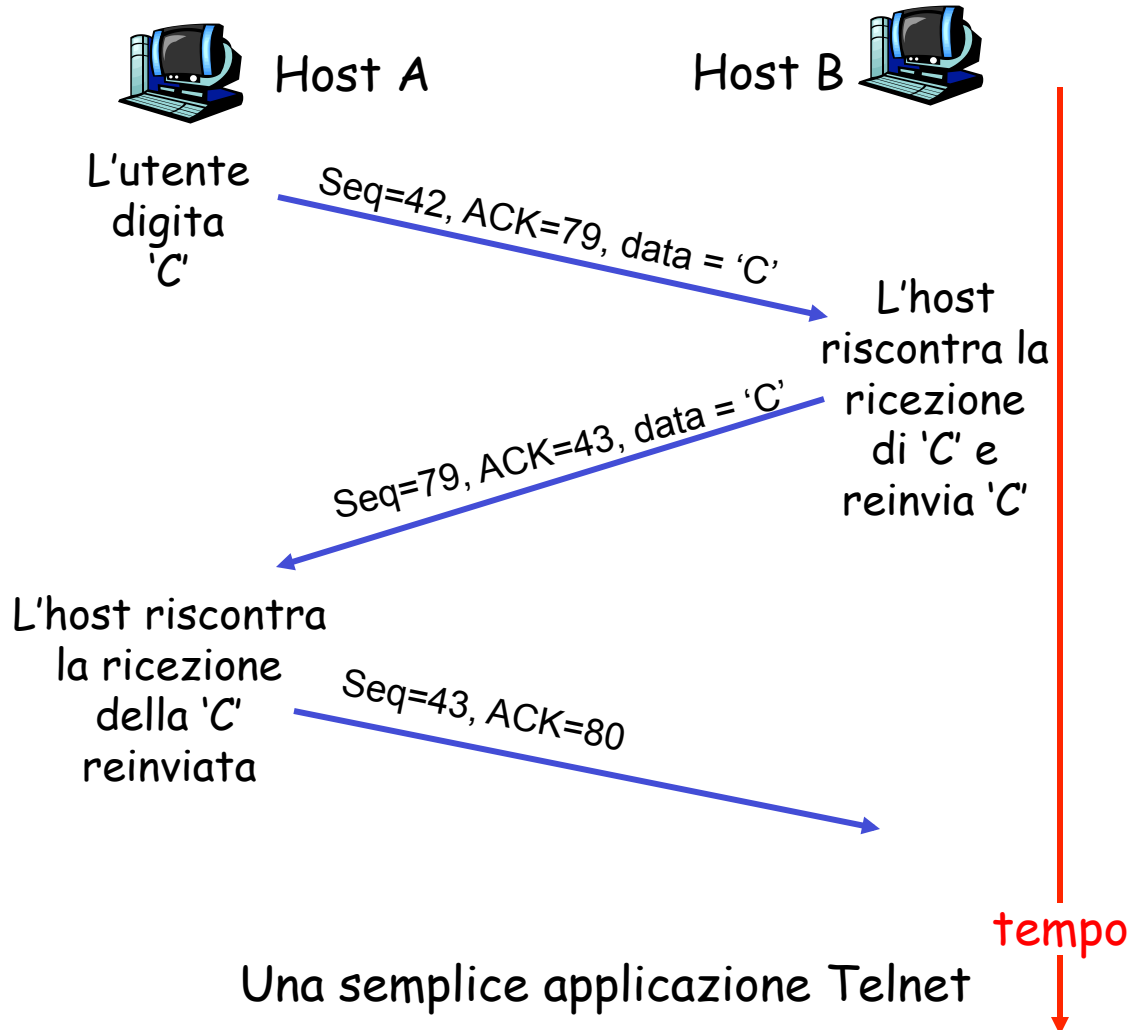
- "numero" del primo byte del segmento nel flusso di byte

ACK:

- numero di sequenza del prossimo byte atteso dall'altro lato
- ACK cumulativo

D: come gestisce il destinatario i segmenti fuori sequenza?

- R: la specifica TCP non lo dice - dipende dall'implementatore



TCP: tempo di andata e ritorno e timeout

D: come impostare il valore del timeout di TCP?

- ❑ Più grande di RTT
 - ma RTT varia
- ❑ Troppo piccolo: timeout prematuro
 - ritrasmissioni non necessarie
- ❑ Troppo grande: reazione lenta alla perdita dei segmenti

D: come stimare RTT?

- ❑ **SampleRTT**: tempo misurato dalla trasmissione del segmento fino alla ricezione di ACK
 - ignora le ritrasmissioni
- ❑ **SampleRTT** varia, quindi occorre una stima "più livellata" di RTT
 - media di più misure recenti, non semplicemente il valore corrente di **SampleRTT**

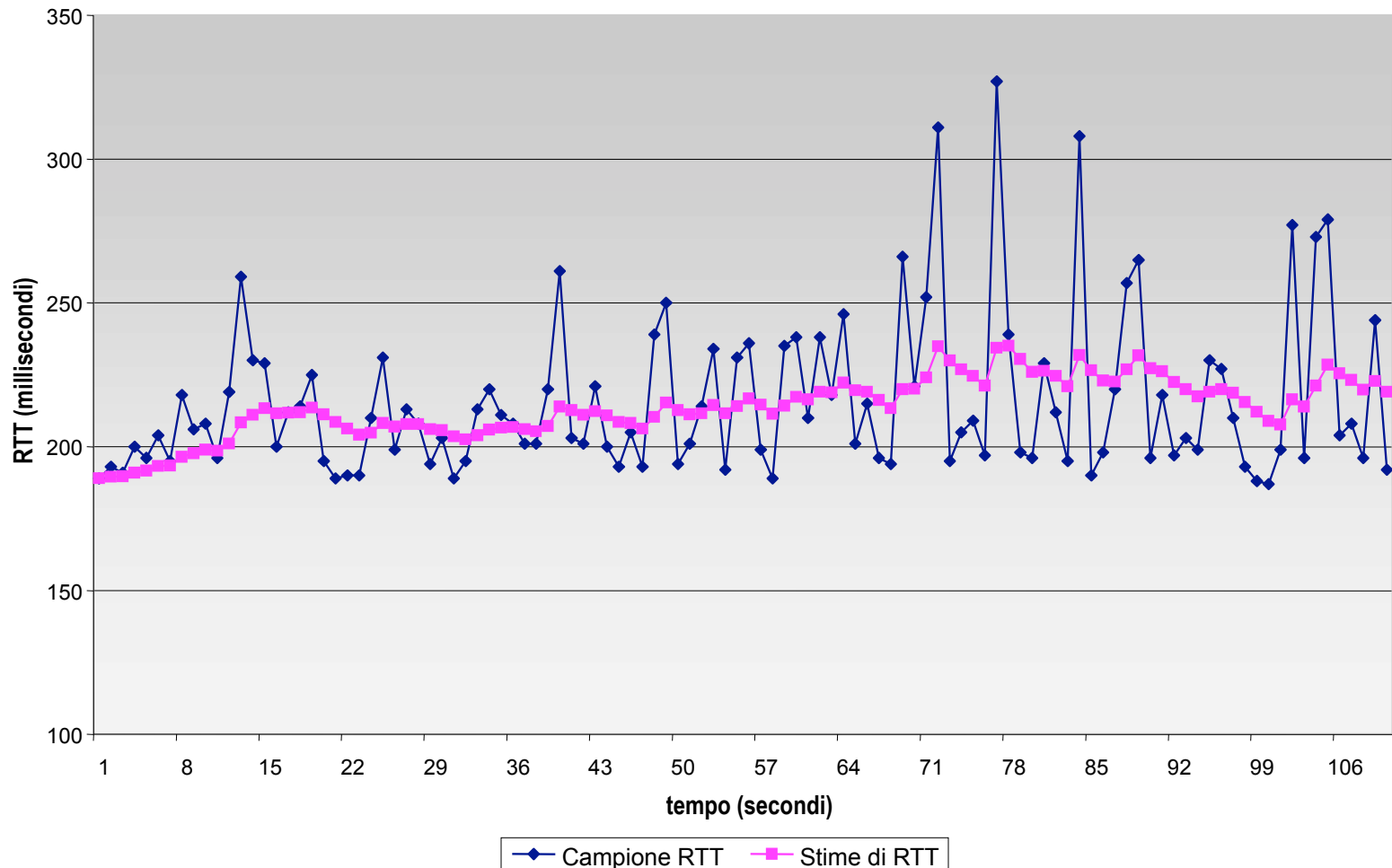
TCP: tempo di andata e ritorno e timeout

$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- ❑ Media mobile esponenziale ponderata
- ❑ L'influenza dei vecchi campioni decresce esponenzialmente
- ❑ Valore tipico: $\alpha = 0,125$

Esempio di stima di RTT:

RTT: gaia.cs.umass.edu e fantasia.eurecom.fr



TCP: tempo di andata e ritorno e timeout

Impostazione del timeout

- ❑ EstimatedRTT più un "margine di sicurezza"
 - grande variazione di EstimatedRTT -> margine di sicurezza maggiore
- ❑ Stimare innanzitutto di quanto SampleRTT si discosta da EstimatedRTT:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(tipicamente, $\beta = 0,25$)

Poi impostare l'intervallo di timeout:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

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

TCP: trasferimento dati affidabile

- ❑ TCP crea un servizio di trasferimento dati affidabile sul servizio inaffidabile di IP
- ❑ Pipeline dei segmenti
- ❑ ACK cumulativi
- ❑ TCP usa un solo timer di ritrasmissione
- ❑ Le ritrasmissioni sono avviate da:
 - eventi di timeout
 - ACK duplicati
- ❑ Inizialmente consideriamo un mittente TCP semplificato:
 - ignoriamo gli ACK duplicati
 - ignoriamo il controllo di flusso e il controllo di congestione

TCP: eventi del mittente

Dati ricevuti dall'applicazione:

- ❑ Crea un segmento con il numero di sequenza
- ❑ Il numero di sequenza è il numero del primo byte del segmento nel flusso di byte
- ❑ Avvia il timer, se non è già in funzione (pensate al timer come se fosse associato al più vecchio segmento non riscontrato)
- ❑ Intervallo di scadenza:
TimeoutInterval

Timeout:

- ❑ Ritrasmette il segmento che ha causato il timeout
- ❑ Riavvia il timer

ACK ricevuti:

- ❑ Se riscontra segmenti precedentemente non riscontrati
 - aggiorna ciò che è stato completamente riscontrato
 - avvia il timer se ci sono altri segmenti da completare

NextSeqNum = InitialSeqNum

SendBase = InitialSeqNum

```
loop (sempre) {  
    switch(evento)
```

evento: i dati ricevuti dall'applicazione superiore
creano il segmento TCP con numero di sequenza NextSeqNum
if (il timer attualmente non funziona)
 avvia il timer
passa il segmento a IP
NextSeqNum = NextSeqNum + lunghezza(dati)

evento: timeout del timer
ritrasmetti il segmento non ancora riscontrato con
 il più piccolo numero di sequenza
avvia il timer

evento: ACK ricevuto, con valore del campo ACK pari a y
if (y > SendBase) {
 SendBase = y
 if (esistono attualmente segmenti non ancora riscontrati)
 avvia il timer
}

```
} /* fine del loop */
```

Mittente TCP (semplificato)

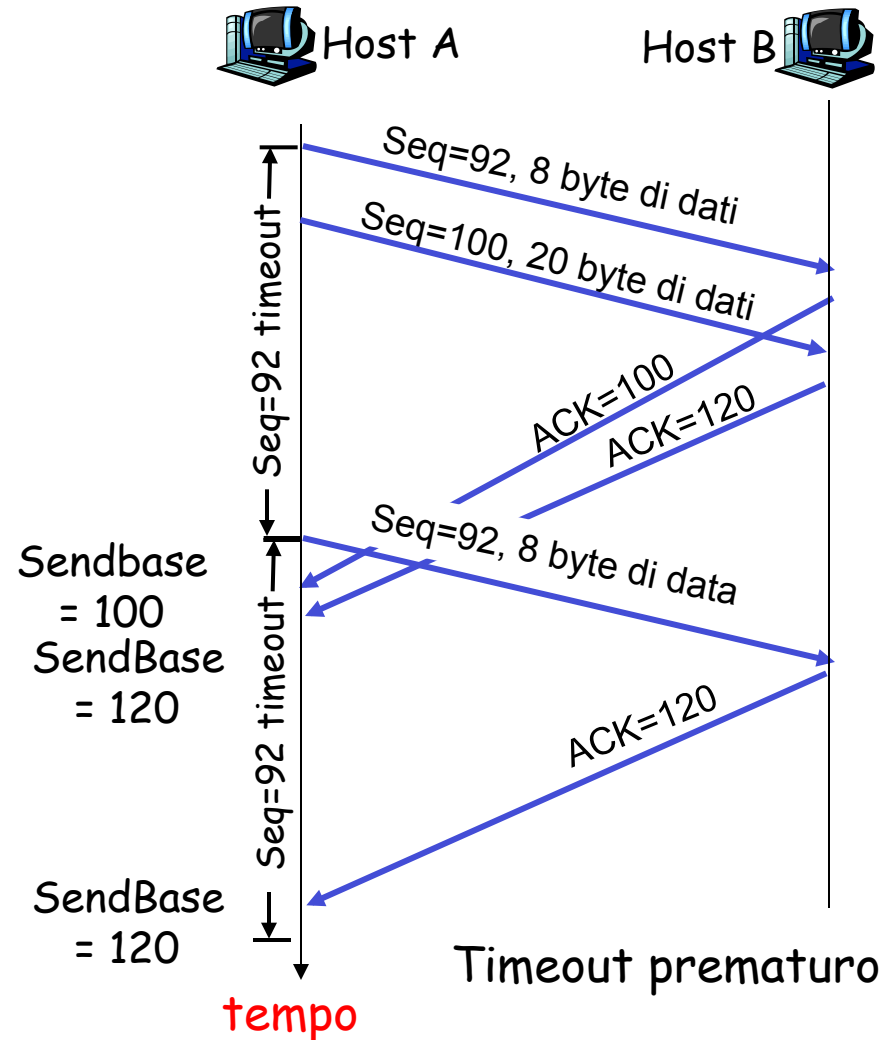
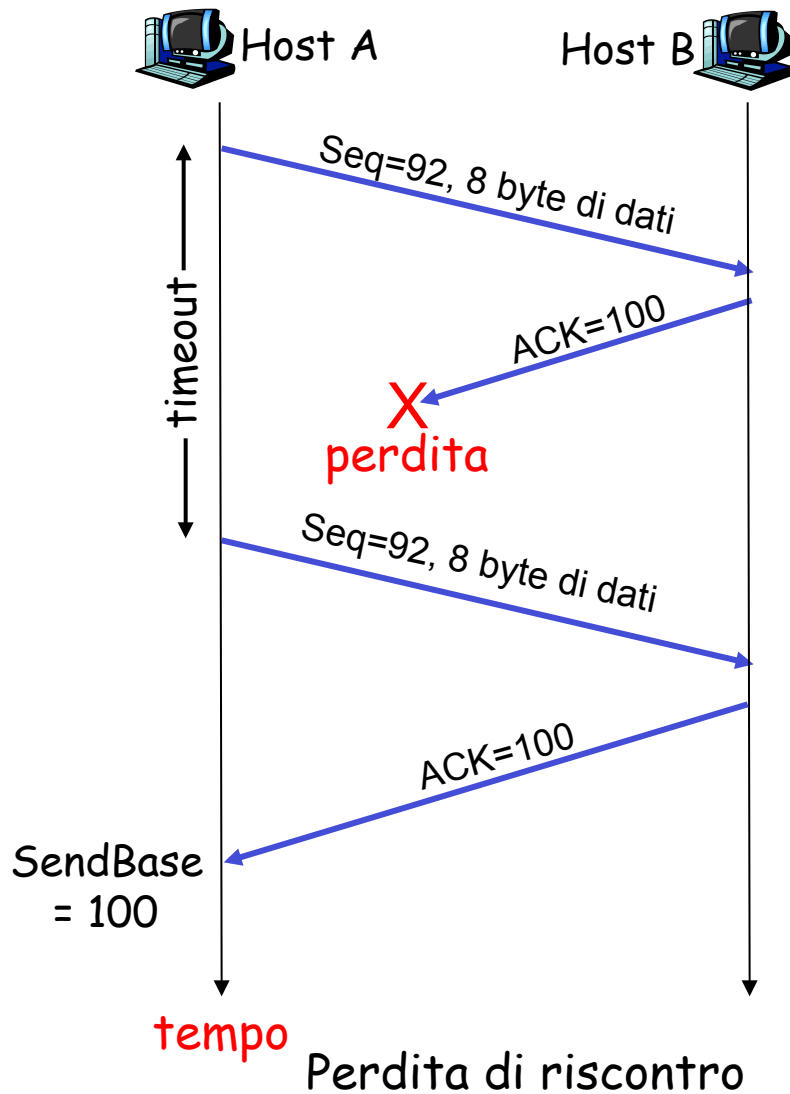
Commento:

- SendBase-1: ultimo byte cumulativamente riscontrato

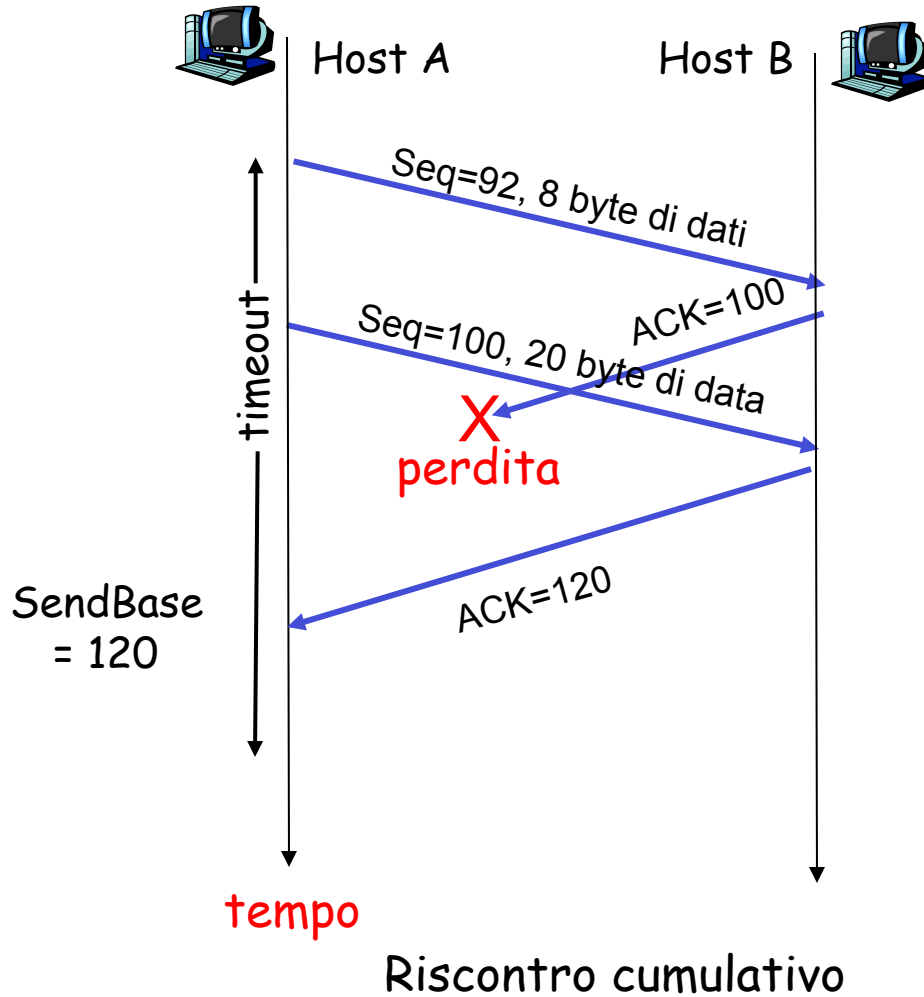
Esempio:

- SendBase-1 = 71;
y = 73, quindi il destinatario vuole 73+ ;
y > SendBase, allora vengono riscontrati tali nuovi dati

TCP: scenari di ritrasmissione



TCP: scenari di ritrasmissione



TCP: generazione di ACK [RFC 1122, RFC 2581]

Evento presso il destinatario

Azione del ricevente TCP

Arrivo ordinato di un segmento con numero di sequenza atteso. Tutti i dati fino al numero di sequenza atteso sono già stati riscontrati.

ACK ritardato. Attende fino a 500 ms l'arrivo del prossimo segmento. Se il segmento non arriva, invia un ACK.

Arrivo ordinato di un segmento con numero di sequenza atteso. Un altro segmento è in attesa di trasmissione dell'ACK.

Invia immediatamente un singolo ACK cumulativo, riscontrando entrambi i segmenti ordinati.

Arrivo non ordinato di un segmento con numero di sequenza superiore a quello atteso. Viene rilevato un buco.

Invia immediatamente un ACK duplicato, indicando il numero di sequenza del prossimo byte atteso.

Arrivo di un segmento che colma parzialmente o completamente il buco.

Invia immediatamente un ACK, ammesso che il segmento cominci all'estremità inferiore del buco.

Ritrasmissione rapida

- ❑ Il periodo di timeout spesso è relativamente lungo:
 - lungo ritardo prima di ritrasmettere il pacchetto perduto.
- ❑ Rileva i segmenti perduti tramite gli ACK duplicati.
 - Il mittente spesso invia molti segmenti.
 - Se un segmento viene smarrito, è probabile che ci saranno molti ACK duplicati.
- ❑ Se il mittente riceve 3 ACK per lo stesso dato, suppone che il segmento che segue il dato riscontrato è andato perduto:
 - ritrasmissione rapida: rispedisce il segmento prima che scada il timer.

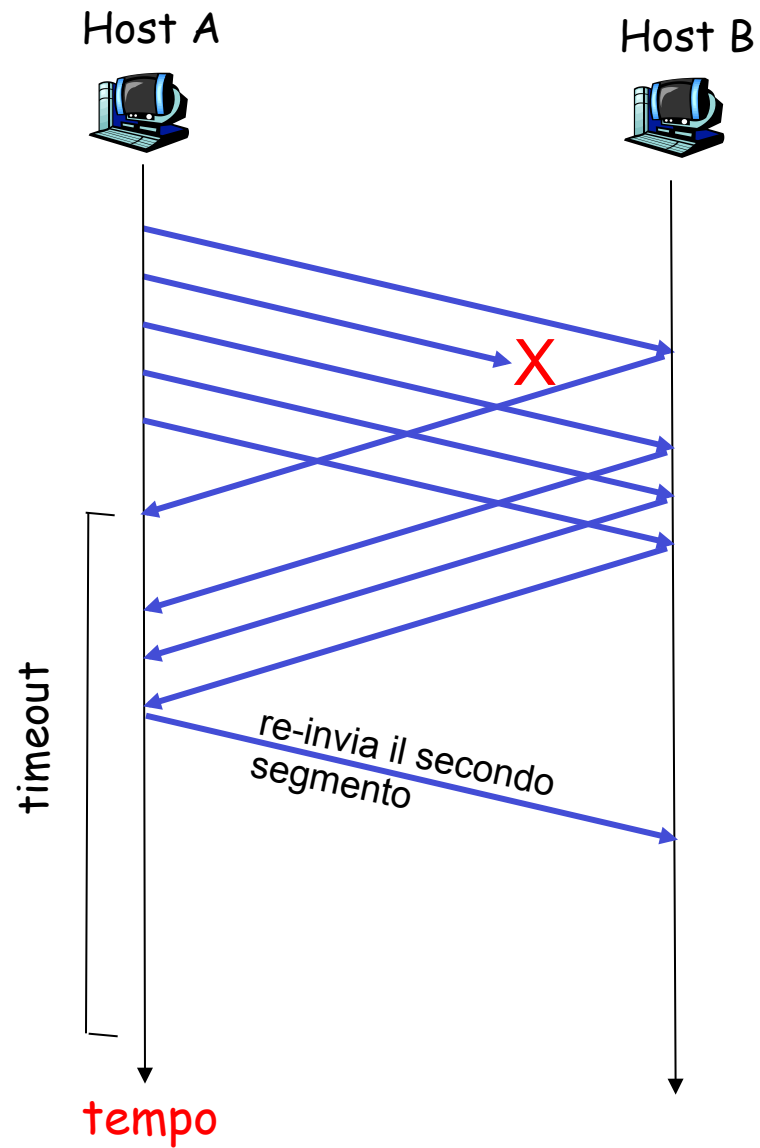


Figure 3.37 Re-invio di un segmento dopo un triplice ACK

Algoritmo della ritrasmissione rapida:

```
evento: ACK ricevuto, con valore del campo ACK pari a y
    if (y > SendBase) {
        SendBase = y
        if (esistono attualmente segmenti non ancora riscontrati)
            avvia il timer
    }
    else {
        incrementa il numero di ACK duplicati ricevuti per y
        if (numero di ACK duplicati ricevuti per y = 3) {
            rispeditisci il segmento con numero di sequenza y
        }
    }
```

un ACK duplicato per un
segmento già riscontrato

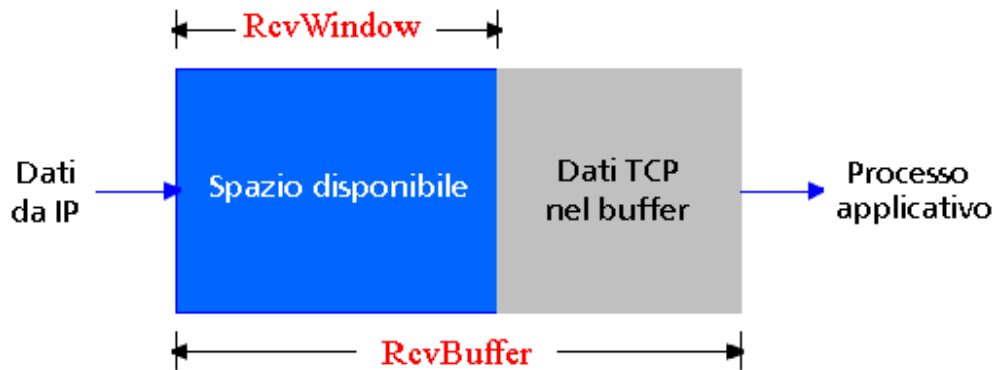
ritrasmissione rapida

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

TCP: controllo di flusso

- Il lato ricevente della connessione TCP ha un buffer di ricezione:



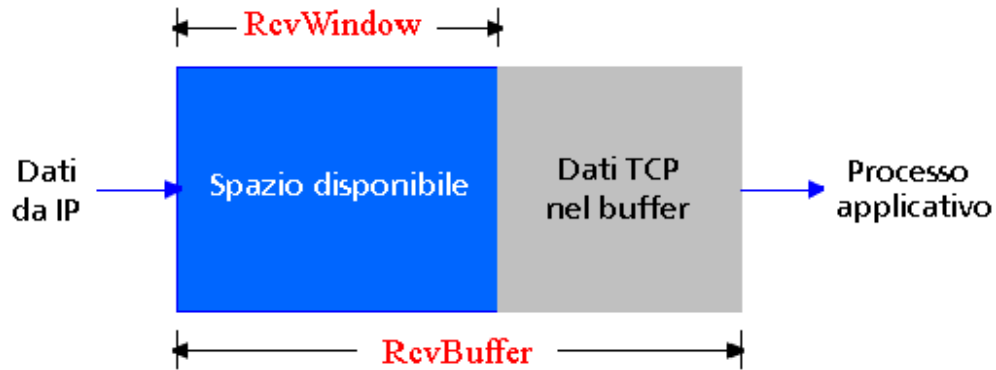
- Il processo applicativo potrebbe essere rallentato dalla lettura nel buffer

Controllo di flusso

Il mittente non vuole sovraccaricare il buffer del destinatario trasmettendo troppi dati, troppo velocemente

- Servizio di corrispondenza delle velocità: la frequenza d'invio deve corrispondere alla frequenza di lettura dell'applicazione ricevente

TCP: funzionamento del controllo di flusso



(supponiamo che il destinatario TCP scarti i segmenti fuori sequenza)

□ Spazio disponibile nel buffer

= $RcvWindow$

= $RcvBuffer - [LastByteRcvd - LastByteRead]$

- Il mittente comunica lo spazio disponibile includendo il valore di $RcvWindow$ nei segmenti
- Il mittente limita i dati non riscontrati a $RcvWindow$
 - garantisce che il buffer di ricezione non vada in overflow

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

Gestione della connessione TCP

Ricordiamo: mittente e destinatario TCP stabiliscono una "connessione" prima di scambiare i segmenti di dati

- ❑ inizializzano le variabili TCP:
 - numeri di sequenza
 - buffer, informazioni per il controllo di flusso (per esempio, RcvWindow)
- ❑ *client*: avvia la connessione
- ❑ `Socket clientSocket = new Socket("hostname", "portnumber");`
- ❑ *server*: contattato dal client
- ❑ `Socket connectionSocket = welcomeSocket.accept();`

Handshake a tre vie:

Passo 1: il client invia un segmento SYN al server

- specifica il numero di sequenza iniziale
- nessun dato

Passo 2: il server riceve SYN e risponde con un segmento SYNACK

- il server alloca i buffer
- specifica il numero di sequenza iniziale del server

Passo 3: il client riceve SYNACK e risponde con un segmento ACK, che può contenere dati

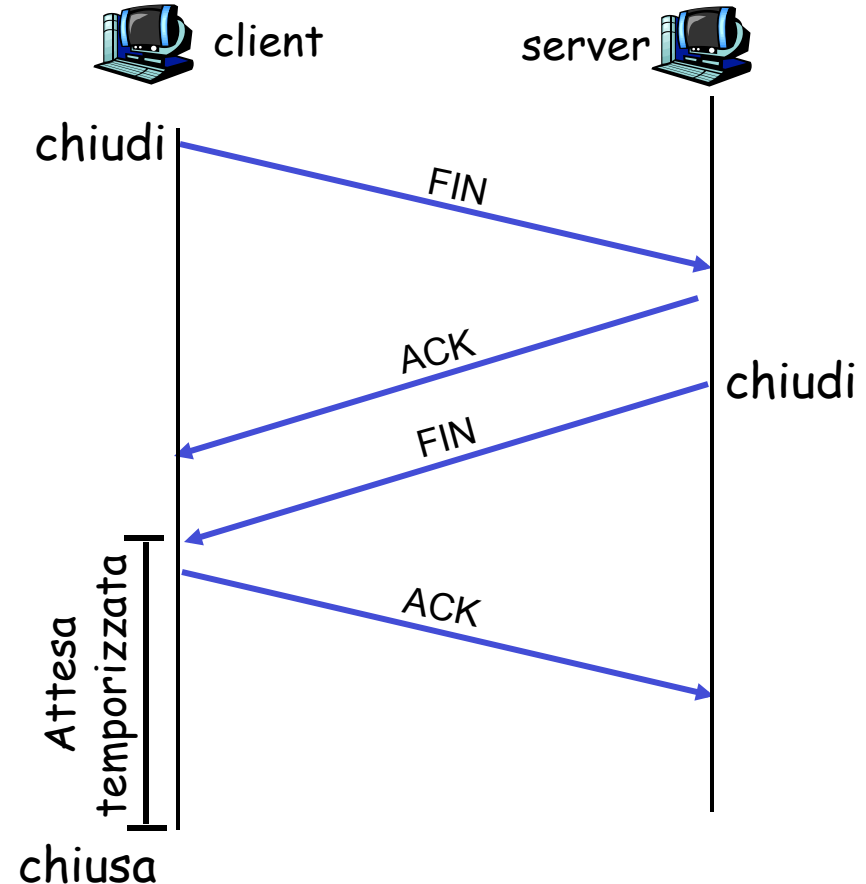
Gestione della connessione TCP (continua)

Chiudere una connessione:

Il client chiude la socket:
`clientSocket.close()` ;

Passo 1: il **client** invia un segmento di controllo FIN al server.

Passo 2: il **server** riceve il segmento FIN e risponde con un ACK. Chiude la connessione e invia un FIN.



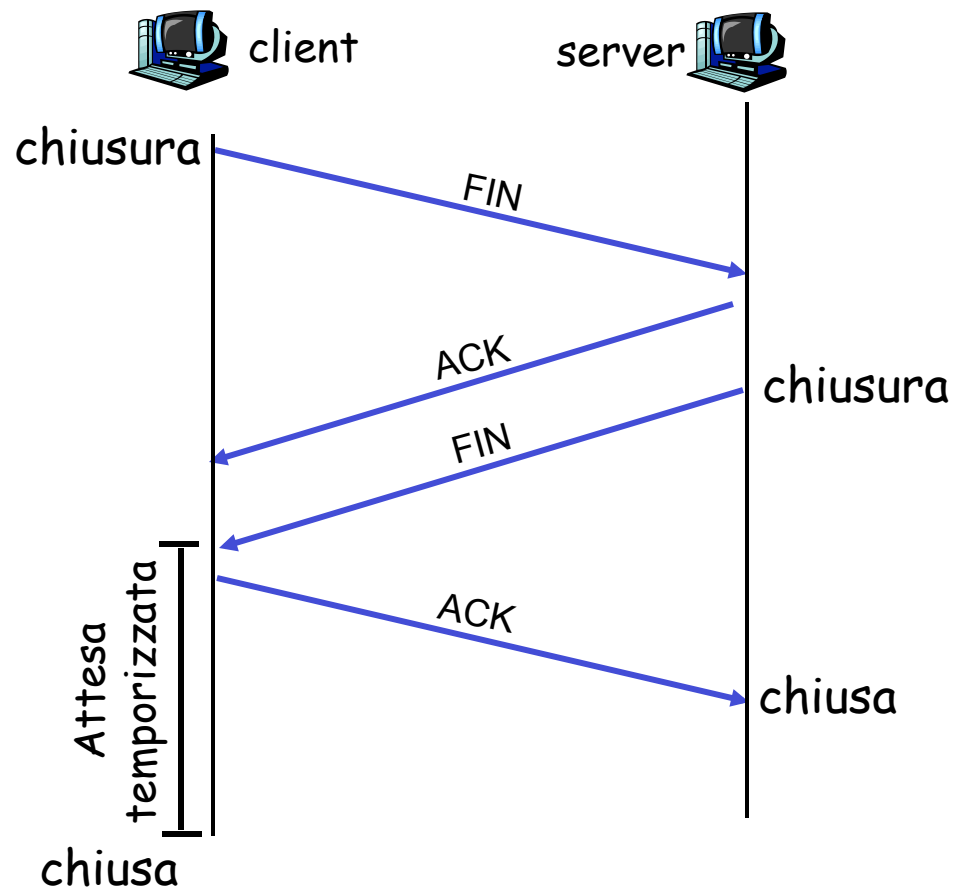
Gestione della connessione TCP (continua)

Passo 3: il **client** riceve FIN e risponde con un ACK.

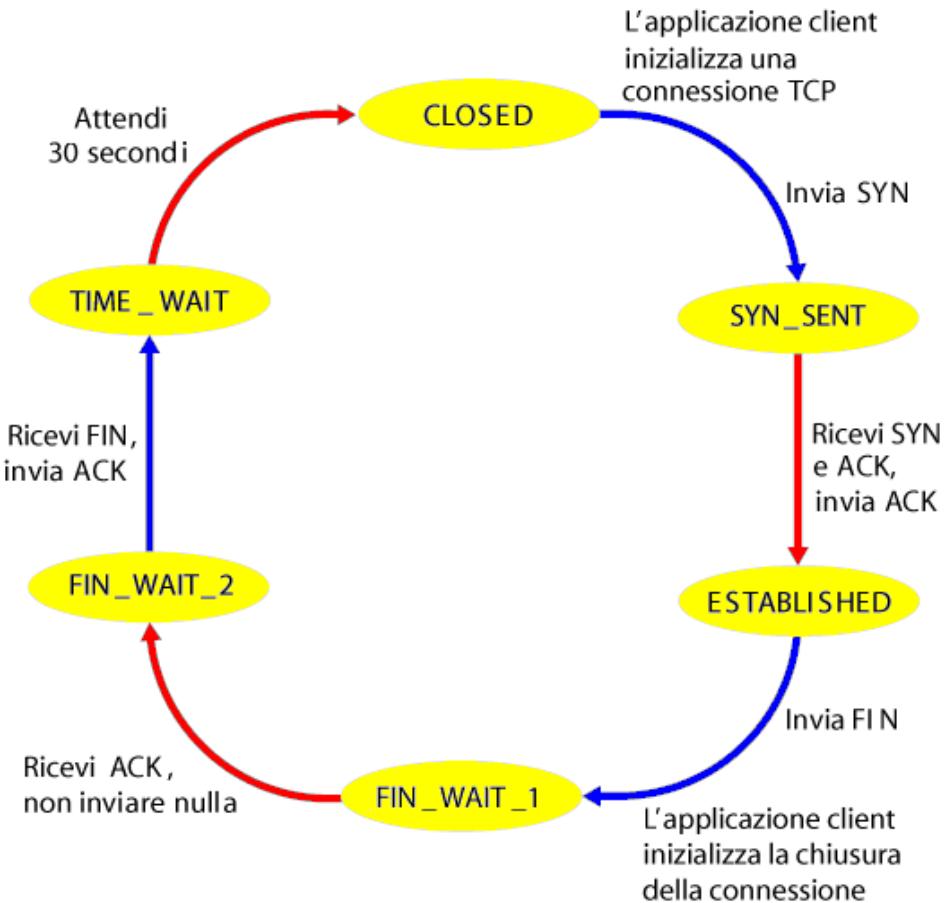
- inizia l'attesa temporizzata - risponde con un ACK ai FIN che riceve

Passo 4: il **server** riceve un ACK. La connessione viene chiusa.

Nota: con una piccola modifica può gestire segmenti FIN contemporanei.

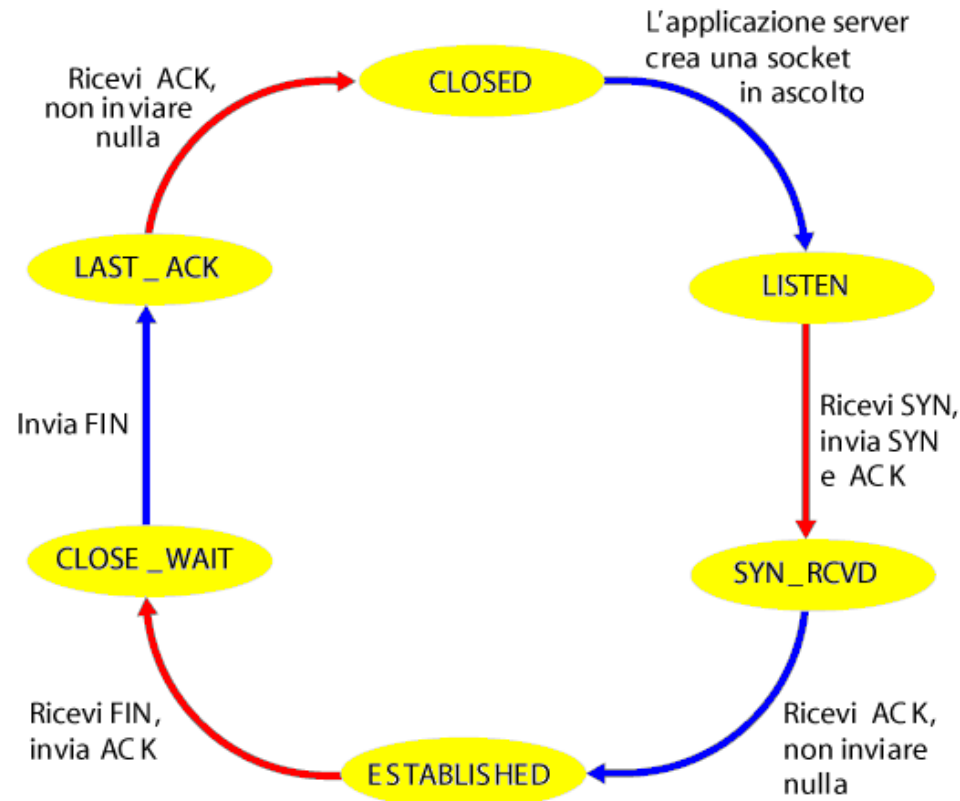


Gestione della connessione TCP (continua)



Sequenza di stati
visitati da un client TCP

Sequenza di stati
visitati dal TCP
sul lato server



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

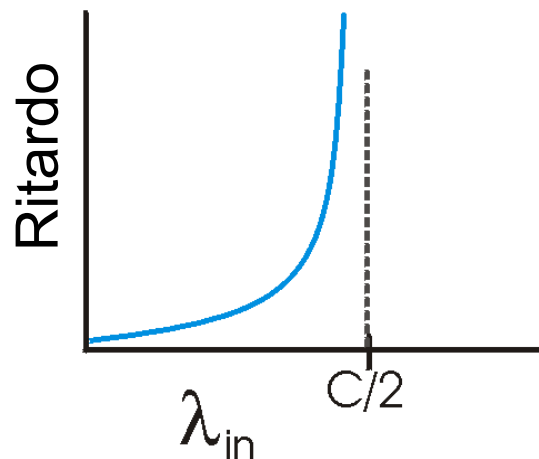
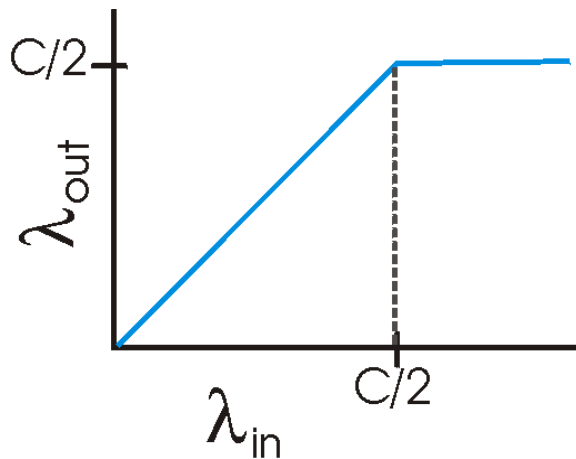
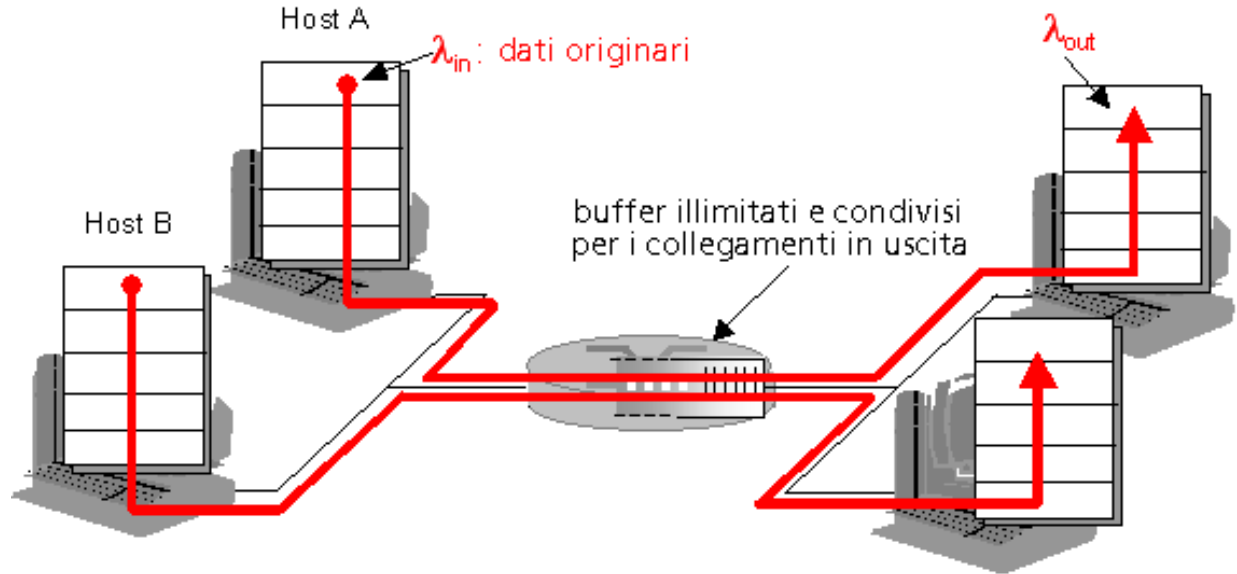
Principi del controllo di congestione

Congestione:

- ❑ informalmente: "troppe sorgenti trasmettono troppi dati, a una velocità talmente elevata che la *rete* non è in grado di gestirli"
- ❑ differisce dal controllo di flusso!
- ❑ sintomi:
 - pacchetti smarriti (overflow nei buffer dei router)
 - lunghi ritardi (accodamento nei buffer dei router)
- ❑ tra i dieci problemi più importanti del networking!

Cause/costi della congestione: scenario 1

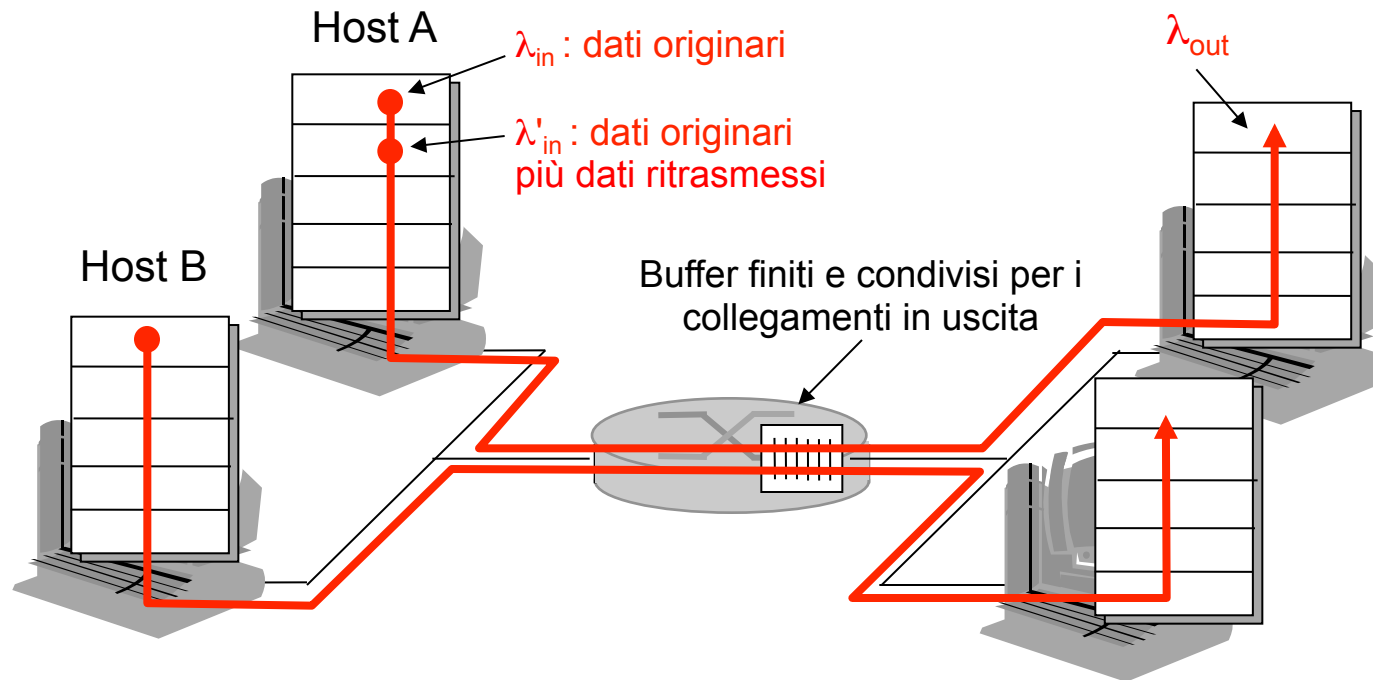
- ❑ due mittenti, due destinatari
- ❑ un router con buffer illimitati
- ❑ nessuna ritrasmissione



- ❑ grandi ritardi se congestionati
- ❑ throughput massimo

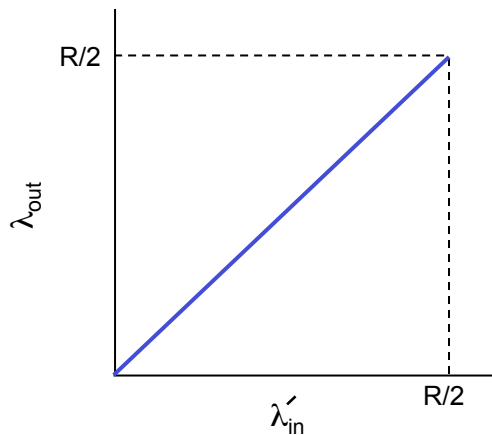
Cause/costi della congestione: scenario 2

- ❑ un router, buffer *finiti*
- ❑ il mittente ritrasmette il pacchetto perduto

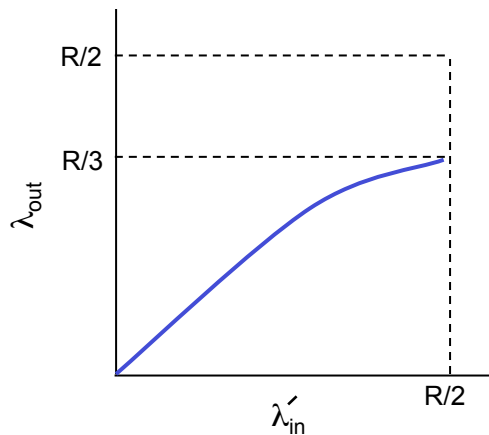


Cause/costi della congestione: scenario 2

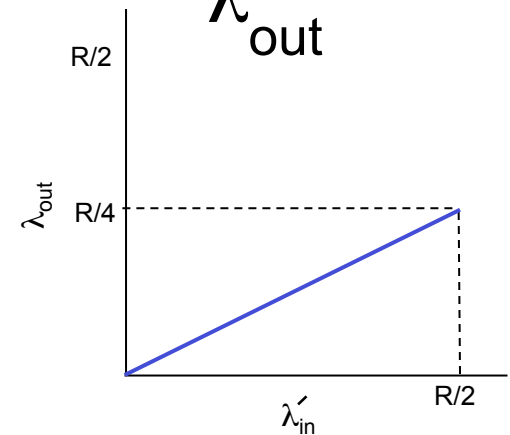
- Sempre: $\lambda_{in} = \lambda_{out}$ (goodput)
- Ritrasmissione "perfetta" solo quando la perdita: $\lambda'_{in} > \lambda_{out}$
- La ritrasmissione del pacchetto ritardato (non perduto) rende λ'_{in} più grande (rispetto al caso perfetto) per lo stesso λ_{out}



a.



b.



c.

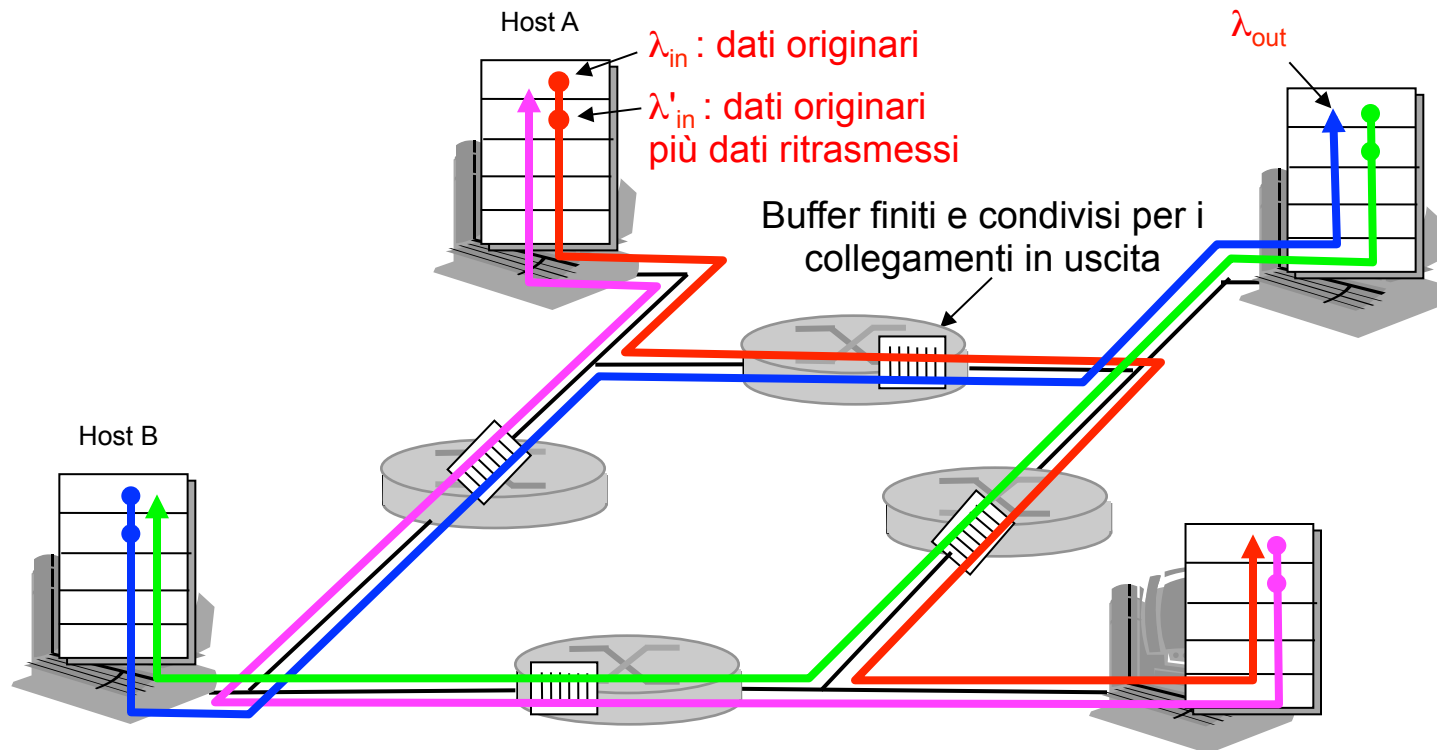
"Costi" della congestione:

- Più lavoro (ritrasmissioni) per un dato "goodput"
- Ritrasmissioni non necessarie: il collegamento trasporta più copie del pacchetto

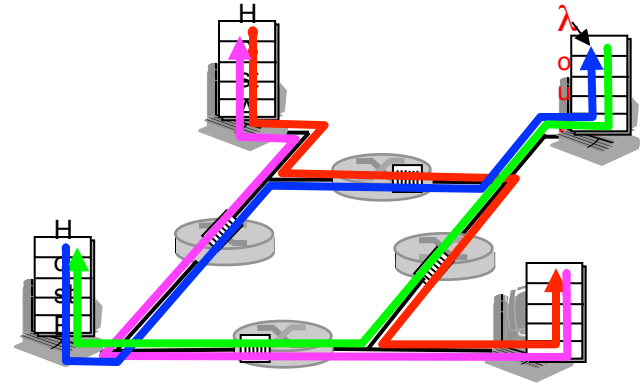
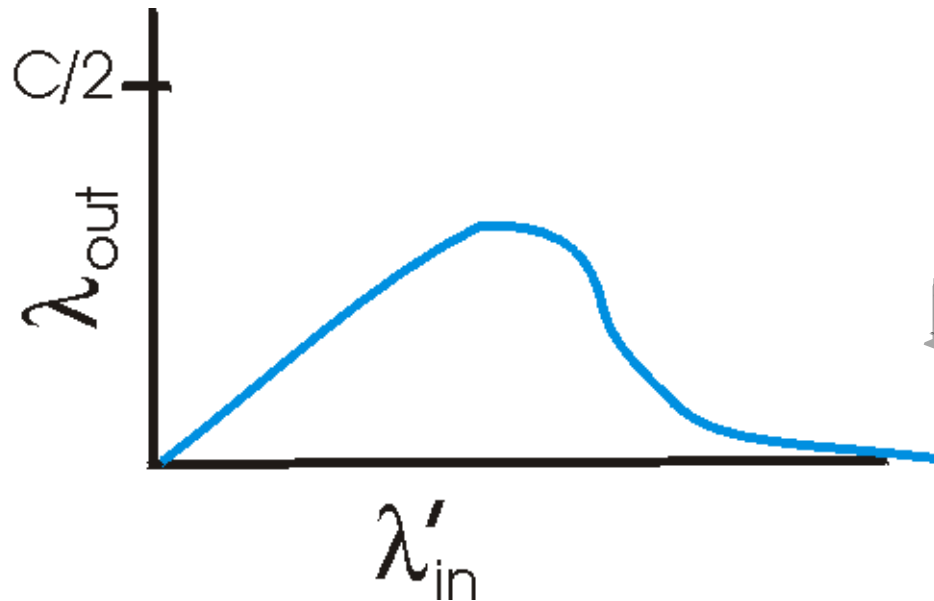
Cause/costi della congestione: scenario 3

- ❑ Quattro mittenti
- ❑ Percorsi multihop
- ❑ timeout/ritrasmissione

D: Che cosa accade quando λ_{in} e λ'_{in} aumentano?



Cause/costi della congestione: scenario 3



Un altro "costo" della congestione:

- Quando il pacchetto viene scartato, la capacità trasmissiva utilizzata sui collegamenti di upstream per instradare il pacchetto risulta sprecata!

Approcci al controllo della congestione

I due principali approcci al controllo della congestione:

Controllo di congestione punto-punto:

- ❑ nessun supporto esplicito dalla rete
- ❑ la congestione è dedotta osservando le perdite e i ritardi nei sistemi terminali
- ❑ metodo adottato da TCP

Controllo di congestione assistito dalla rete:

- ❑ i router forniscono un feedback ai sistemi terminali
 - un singolo bit per indicare la congestione (SNA, DECbit, TCP/IP ECN, ATM)
 - comunicare in modo esplicito al mittente la frequenza trasmissiva

Un esempio: controllo di congestione ATM ABR

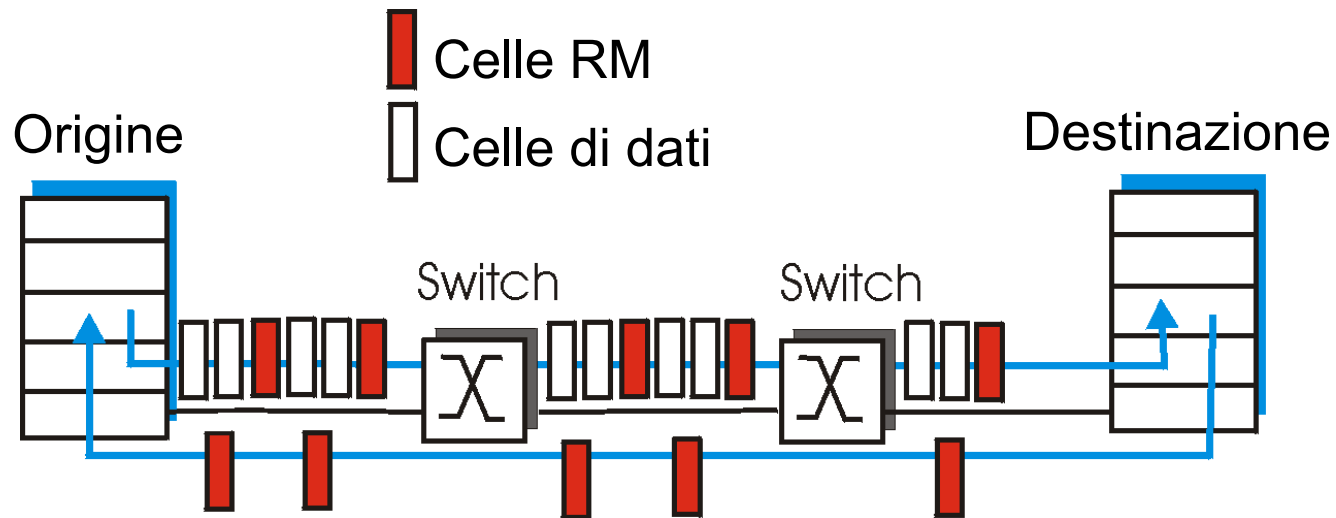
ABR: available bit rate:

- ❑ "servizio elastico"
- ❑ se il percorso del mittente è "sottoutilizzato":
 - il mittente dovrebbe utilizzare la larghezza di banda disponibile
- ❑ se il percorso del mittente è congestionato:
 - il mittente dovrebbe ridurre al minimo il tasso trasmissivo

Celle RM (resource management):

- ❑ inviate dal mittente, inframmezzate alle celle di dati
- ❑ i bit in una cella RM sono impostati dagli switch (*"assistenza dalla rete"*)
 - bit **NI**: nessun aumento del tasso trasmissivo (congestione moderata)
 - bit **CI**: indicazione di congestione (traffico intenso)
- ❑ il destinatario restituisce le celle RM al mittente con i bit intatti

Un esempio: controllo di congestione ATM ABR



- ❑ Campo esplicito di frequenza (ER, explicit rate) in ogni cella RM
 - lo switch congestionato può diminuire il valore del campo ER
 - in questo modo, il campo ER sarà impostato alla velocità minima supportabile da tutti gli switch sul percorso globale
- ❑ Ogni cella di dati contiene un bit EFCI: impostato a 1 nello switch congestionato
 - se la cella di dati che precede la cella RM ha impostato il bit EFCI, il mittente imposta il bit CI nella cella RM restituita

Capitolo 3: Livello di trasporto

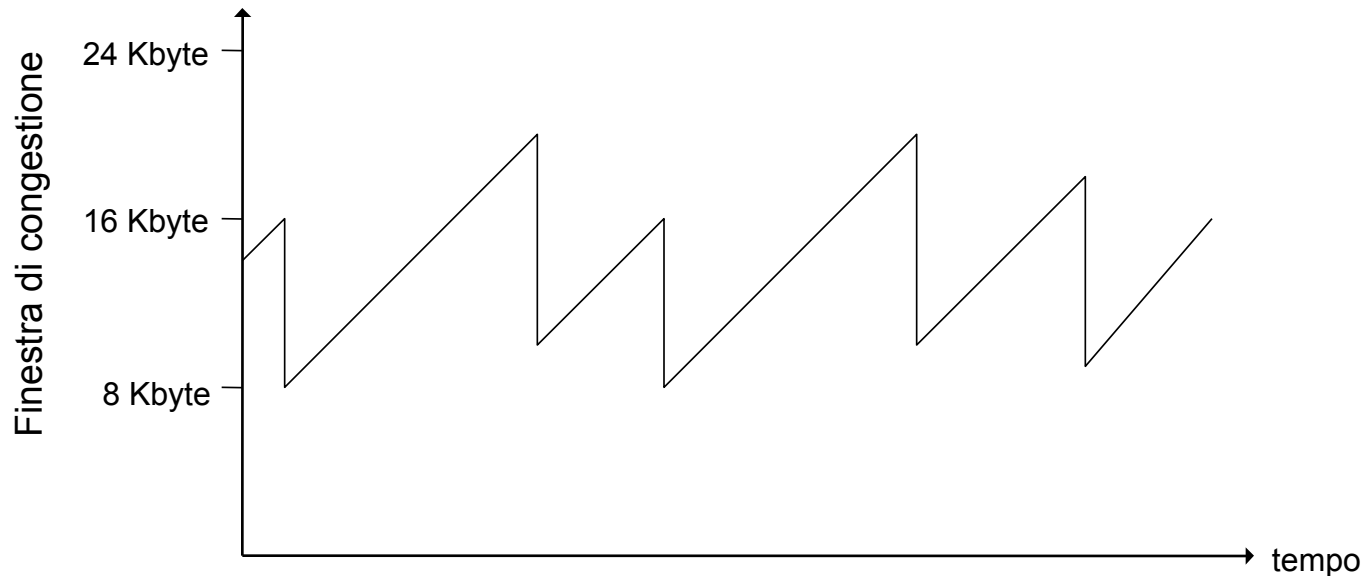
- ❑ 3.1 Servizi a livello di trasporto
- ❑ 3.2 Multiplexing e demultiplexing
- ❑ 3.3 Trasporto senza connessione: UDP
- ❑ 3.4 Principi del trasferimento dati affidabile
- ❑ 3.5 Trasporto orientato alla connessione: TCP
 - struttura dei segmenti
 - trasferimento dati affidabile
 - controllo di flusso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

Controllo di congestione TCP: incremento additivo e decremento moltiplicativo (AIMD)

Approccio: aumenta il tasso trasmissivo sondando la rete, fino a quando non si verifica una perdita

Incremento additivo: aumenta CongWin di 1 MSS a ogni RTT in assenza di eventi di perdita: *sondaggio*

Decremento moltiplicativo: riduce a metà CongWin dopo un evento di perdita



Controllo di congestione AIMD

Controllo di congestione TCP

- Il mittente limita la trasmissione:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$$

- Approssimativamente:

$$\text{Frequenza d'invio} = \frac{\text{CongWin}}{\text{RTT}} \text{ byte/sec}$$

- CongWin è una funzione dinamica della congestione percepita

In che modo il mittente percepisce la congestione?

- Evento di perdita = timeout o ricezione di 3 ACK duplicati
- Il mittente TCP riduce la frequenza d'invio (CongWin) dopo un evento di perdita

tre meccanismi:

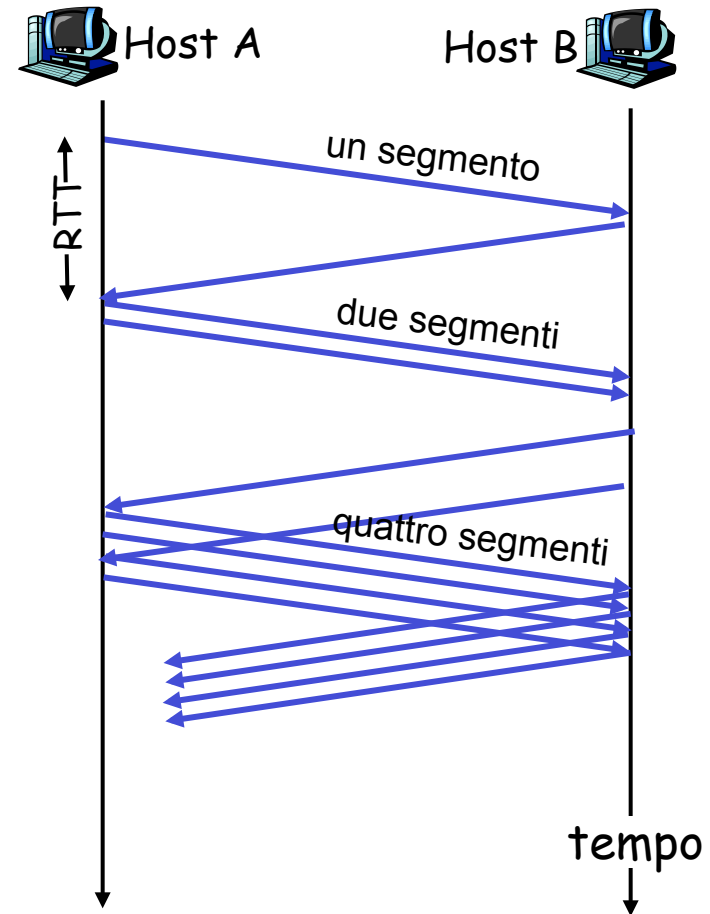
- AIMD
- Partenza lenta
- Reazione agli eventi di timeout

Partenza lenta

- ❑ Quando si stabilisce una connessione,
 $\text{CongWin} = 1 \text{ MSS}$
 - Esempio: $\text{MSS} = 500 \text{ byte}$
 $\text{RTT} = 200 \text{ msec}$
 - Frequenza iniziale = 20 kbps
- ❑ La larghezza di banda disponibile potrebbe essere $\gg \text{MSS}/\text{RTT}$
 - Consente di raggiungere rapidamente una frequenza d'invio significativa
- ❑ Quando inizia la connessione, la frequenza aumenta in modo esponenziale, fino a quando non si verifica un evento di perdita

Partenza lenta (continua)

- Quando inizia la connessione, la frequenza aumenta in modo esponenziale, fino a quando non si verifica un evento di perdita:
 - raddoppia CongWin a ogni RTT
 - ciò avviene incrementando CongWin per ogni ACK ricevuto
- **Riassunto:** la frequenza iniziale è lenta, ma poi cresce in modo esponenziale



Affinamento

- Dopo 3 ACK duplicati:
 - CongWin è ridotto a metà
 - la finestra poi cresce linearmente
- Ma dopo un evento di timeout:
 - CongWin è impostata a 1 MSS;
 - poi la finestra cresce in modo esponenziale
 - fino a un valore di soglia, poi cresce linearmente

Filosofia:

- 3 ACK duplicati indicano la capacità della rete di consegnare qualche segmento
- un timeout prima di 3 ACK duplicati è "più allarmante"

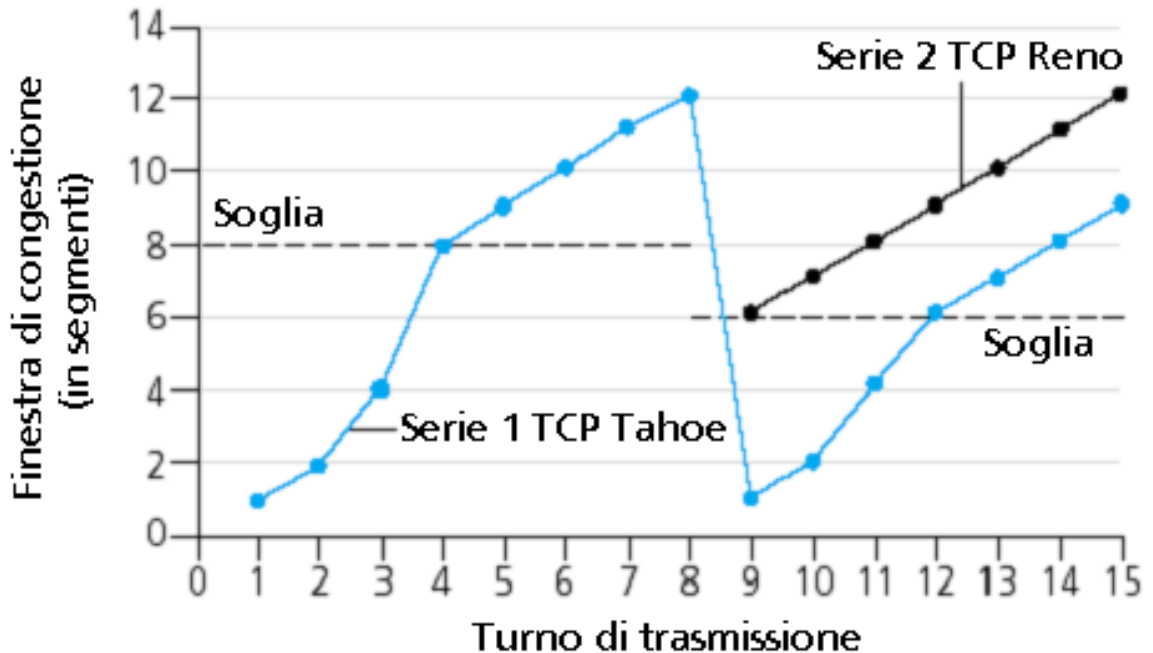
Affinamento (continua)

D: Quando l'incremento esponenziale dovrà diventare lineare?

R: Quando CongWin raggiunge 1/2 del suo valore prima del timeout.

Implementazione:

- ❑ Soglia variabile
- ❑ In caso di evento di perdita, la soglia è impostata a 1/2 di CongWin, appena prima dell'evento di perdita



Riassunto: il controllo della congestione TCP

- ❑ Quando CongWin è sotto la soglia (Threshold), il mittente è nella fase di **partenza lenta**; la finestra cresce in modo esponenziale.
- ❑ Quando CongWin è sopra la soglia, il mittente è nella fase di **congestion avoidance**; la finestra cresce in modo lineare.
- ❑ Quando si verificano **tre ACK duplicati**, il valore di Threshold viene impostato a $\text{CongWin}/2$ e CongWin viene impostata al valore di Threshold.
- ❑ Quando si verifica un **timeout**, il valore di Threshold viene impostato a $\text{CongWin}/2$ e CongWin è impostata a 1 MSS.

Controllo di congestione del mittente TCP

Stato	Evento	Azione del mittente TCP	Commenti
Slow Start (SS)	Ricezione di ACK per dati precedentemente non riscontrati	$\text{CongWin} = \text{CongWin} + \text{MSS}$, If ($\text{CongWin} > \text{Threshold}$) imposta lo stato a "Congestion Avoidance"	CongWin raddoppia a ogni RTT
Congestion Avoidance (CA)	Ricezione di ACK per dati precedentemente non riscontrati	$\text{CongWin} = \text{CongWin} + \text{MSS} * (\text{MSS} / \text{CongWin})$	Incremento additivo: CongWin aumenta di 1 MSS a ogni RTT
SS o CA	Rilevato un evento di perdita da tre ACK duplicati	$\text{Threshold} = \text{CongWin} / 2$, $\text{CongWin} = \text{Threshold}$, imposta lo stato a "Congestion Avoidance"	Ripristino rapido con il decremento moltiplicativo. CongWin non sarà mai minore di 1 MSS
SS o CA	Timeout	$\text{Threshold} = \text{CongWin} / 2$, $\text{CongWin} = 1 \text{ MSS}$, imposta lo stato a "Slow Start"	Entra nello stato "Slow Start"
SS o CA	ACK duplicato	Incrementa il conteggio degli ACK duplicati per il segmento in corso di riscontro	CongWin e Threshold non variano

Throughput TCP

- ❑ Qual è il throughput medio di TCP in funzione della dimensione della finestra e di RTT?
 - Ignoriamo le fasi di partenza lenta
- ❑ Sia W la dimensione della finestra quando si verifica una perdita.
- ❑ Quando la finestra è W , il throughput è W/RTT
- ❑ Subito dopo la perdita, la finestra si riduce a $W/2$, il throughput a $W/2RTT$.
- ❑ Throughout medio: $0,75 W/RTT$

Futuro di TCP

- ❑ Esempio: segmenti da 1500 byte, RTT di 100 ms, vogliamo un throughput da 10 Gbps
- ❑ Occorre una dimensione della finestra pari a $W = 83.333$ segmenti in transito
- ❑ Throughput in funzione della frequenza di smarrimento:

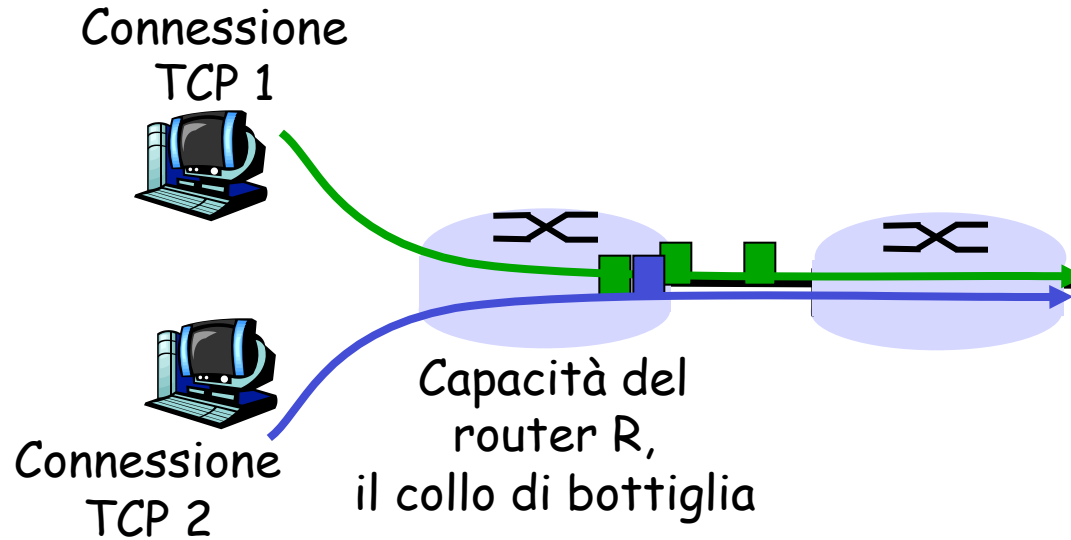
$$\rightarrow L = 2 \times 10^{-10} \frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

Wow

- ❑ Occorrono nuove versioni di TCP per ambienti ad alta velocità!

Equità di TCP

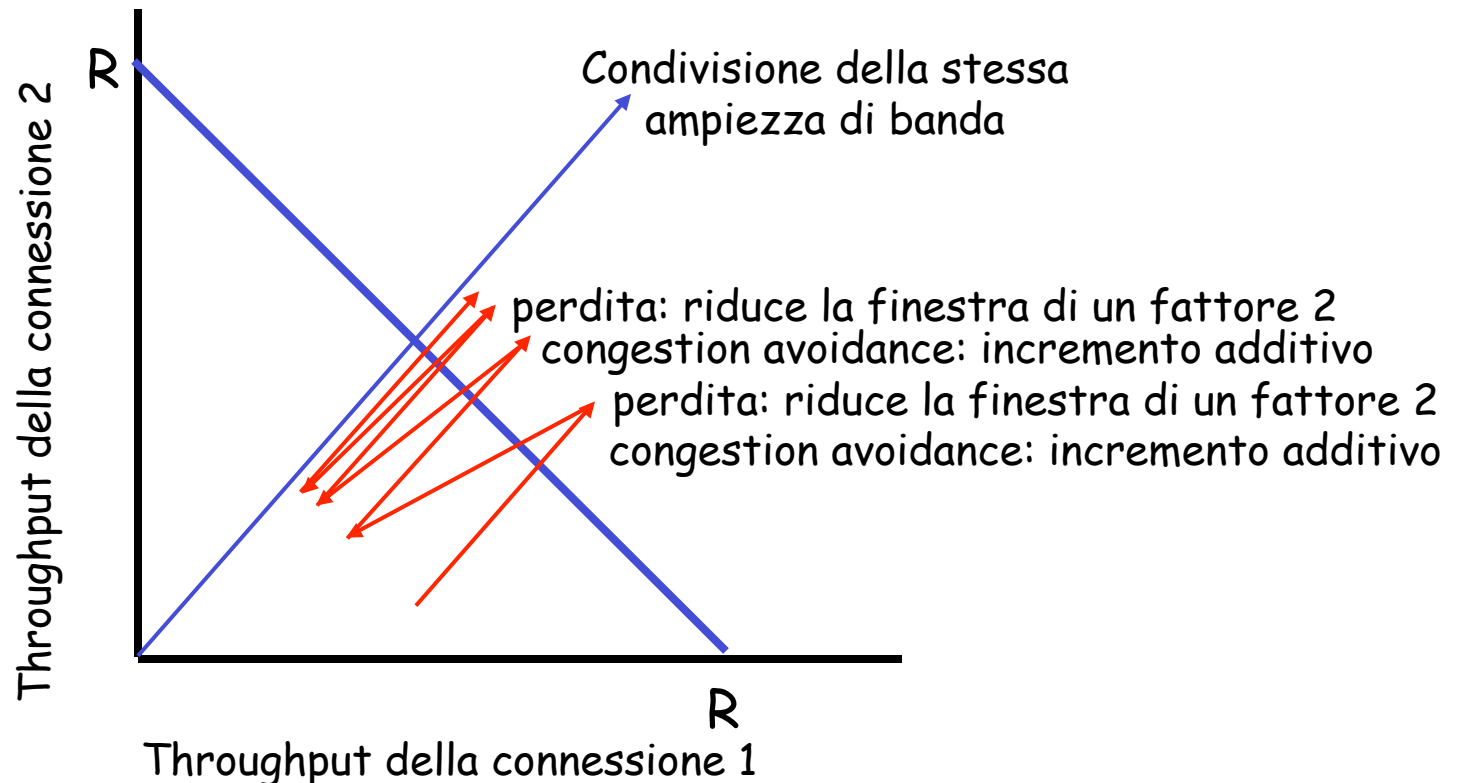
Equità: se K sessioni TCP condividono lo stesso collegamento con ampiezza di banda R , che è un collo di bottiglia per il sistema, ogni sessione dovrà avere una frequenza trasmissiva media pari a R/K .



Perché TCP è equo?

Due connessioni in concorrenza tra loro:

- L'incremento additivo determina una pendenza pari a 1, all'aumentare del throughput
- Il decremento moltiplicativo riduce il throughput in modo proporzionale



Equità

Equità e UDP

- ❑ Le applicazioni multimediali spesso non usano TCP
 - non vogliono che il loro tasso trasmissivo venga ridotto dal controllo di congestione
- ❑ Utilizzano UDP:
 - immettono audio/video a frequenza costante, tollerano la perdita di pacchetti
- ❑ Area di ricerca: TCP friendly

Equità e connessioni TCP in parallelo

- ❑ Nulla può impedire a un'applicazione di aprire connessioni in parallelo tra 2 host
- ❑ I browser web lo fanno
- ❑ Esempio: un collegamento di frequenza R che supporta 9 connessioni;
 - Se una nuova applicazione chiede una connessione TCP, ottiene una frequenza trasmissiva pari a $R/10$
 - Se la nuova applicazione chiede 11 connessioni TCP, ottiene una frequenza trasmissiva pari a $R/2$!

Capitolo 3: Riassunto

- ❑ principi alla base dei servizi del livello di trasporto:
 - multiplexing, demultiplexing
 - trasferimento dati affidabile
 - controllo di flusso
 - controllo di congestione
- ❑ implementazione in Internet
 - UDP
 - TCP

Prossimamente:

- ❑ lasciare la “periferia” della rete (livelli di applicazione e di trasporto)
- ❑ entrare nel “cuore” della rete