Corso di Laurea Triennale in Informatica
Università degli Studi della Basilicata

Reti di Calcolatori

Docente: Ugo Erra
ugo.erra+reti@unibas.it

7° Lezione – Livello di trasporto – I° parte

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

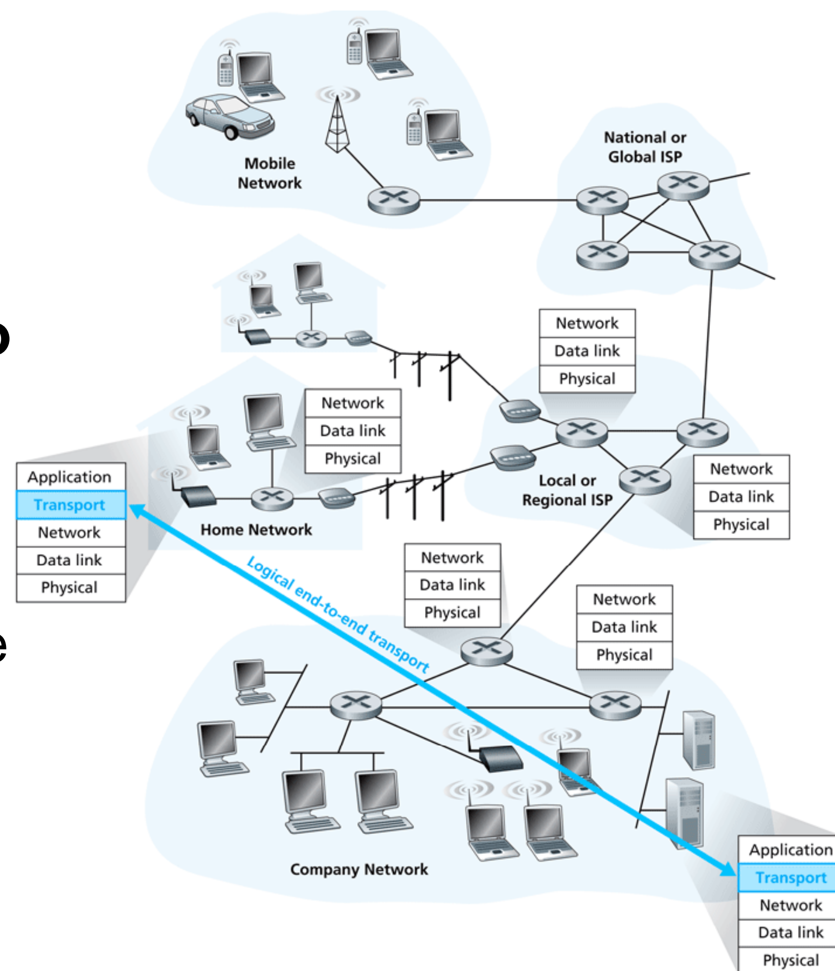
Sommario

- **Servizi a livello di trasporto**
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP
- Principi del trasferimento dati affidabile
- Protocolli affidabili con pipeling

Dove ci troviamo?

Servizi e protocollo di trasporto

- Forniscono la **comunicazione logica** tra processi applicativi di host differenti
- Il livello trasporto converte i messaggi ricevuti dal livello applicazione in **segmenti a livello di trasporto**
- 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



Relazione tra il livello di rete ed il livello di trasporto

- Livello di rete: comunicazione logica tra host
- Livello di trasporto: comunicazione logica tra processi
  - Si basa sui servizi del livello di rete

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

- 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 a massimo sforzo
- Servizi non disponibili:
  - Garanzia su ritardi
  - Garanzia su ampiezza di banda

Gestione dei socket

- Un processo può presentare una o più socket attraverso le quali i dati fluiscono dalla rete al processo o viceversa
- Il livello di trasporto trasferisce i dati al socket non al processo
  - Poiché può esserci più di una socket nell'host ricezione ogni socket deve essere indentificato univocamente
- Il multiplexing ed il demultiplexing sono due compiti dello strato trasporto per la gestione corretta dei dati da/verso i socket

- Servizi a livello di trasporto
- **Multiplexing e demultiplexing**
- Trasporto senza connessione: UDP
- Principi del trasferimento dati affidabile
- Protocolli affidabili con pipeling

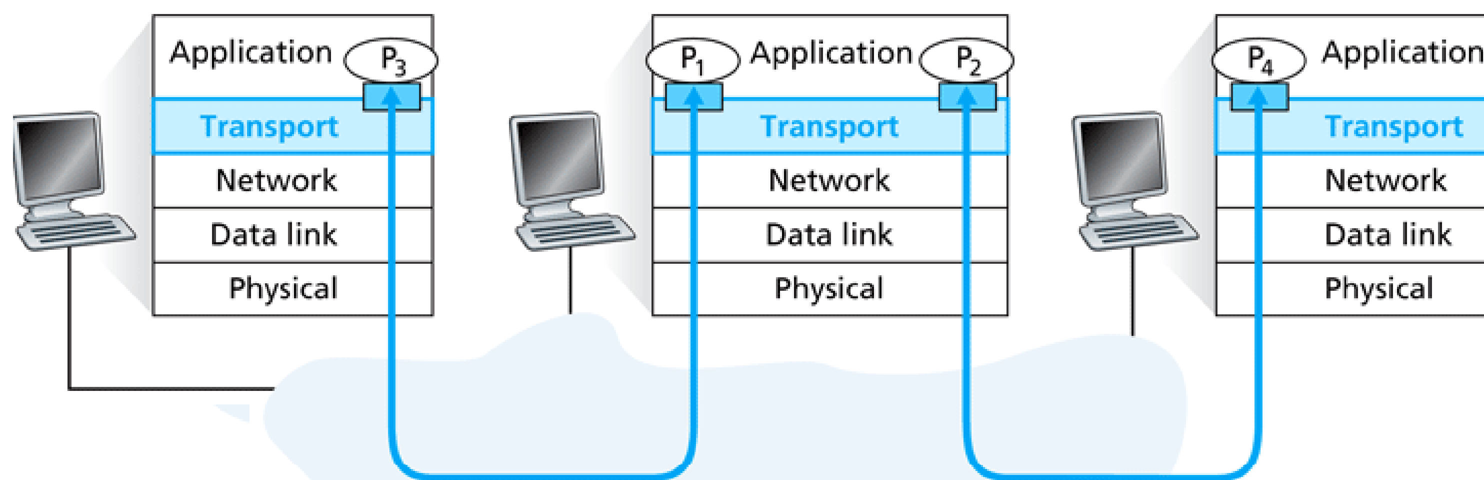
Multiplexing/Demultiplexing

Demultiplexing nell'host ricevente:

consegnare i segmenti ricevuti alla socket appropriata

Multiplexing nell'host mittente:

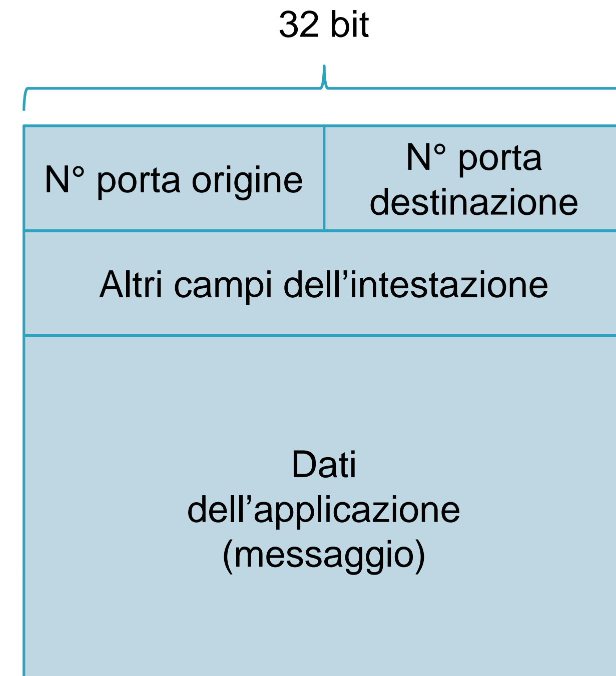
raccogliere i dati da varie socket, incapsularli con l'intestazione (utilizzati poi per il demultiplexing)



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

- Come identifichiamo un socket in UDP e nel TCP?



Struttura del segmento TCP/UDP

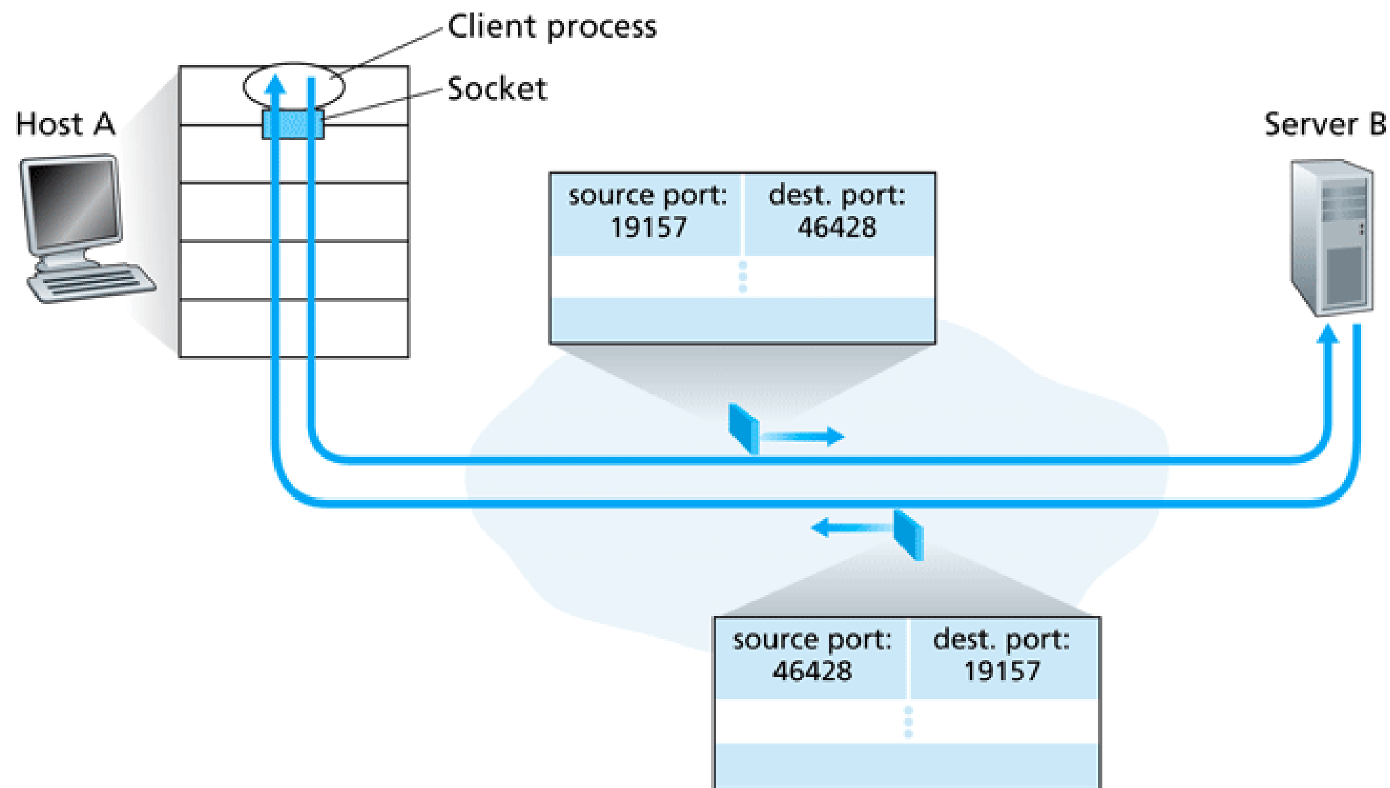
Demultiplexing senza connessione:UDP

- La socket UDP è identificata da 2 parametri:
  - **Indirizzo IP di destinazione**
  - **Numero della porta di destinazione**
- In UDP possiamo creare una socket specificando i numeri di porta oppure no

```
DatagramSocket mySocket1 = new DatagramSocket();
DatagramSocket mySocket2 = new DatagramSocket(99157);
```

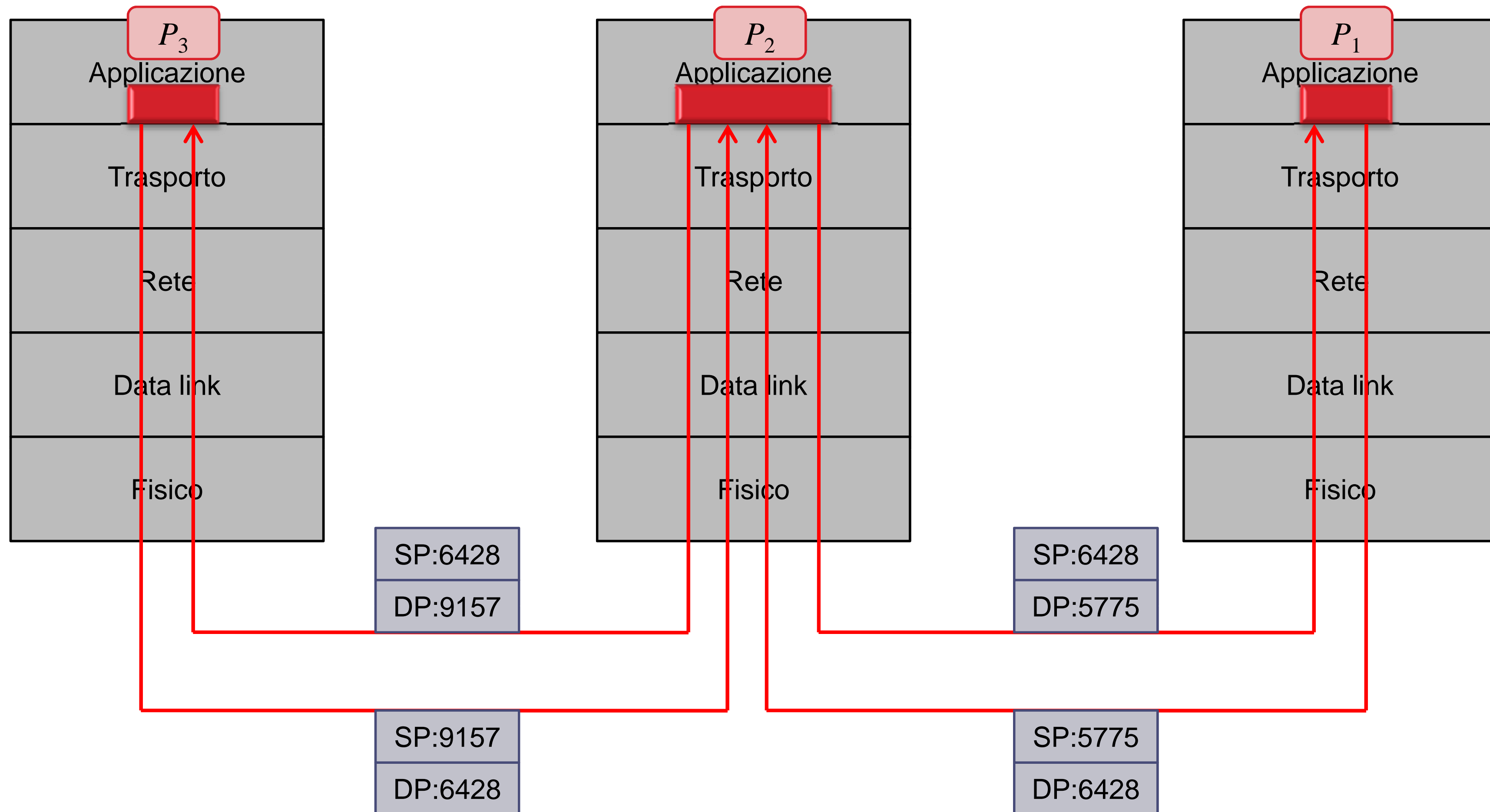
- Quando l'host riceve il segmento UDP:
  - Controlla il numero della porta di destinazione nel segmento
  - Invia il segmento UDP alla socket con quel numero di porta
- <u>I datagrammi IP con indirizzi IP di origine e/o numeri di porta di origine differenti vengono inviati alla stessa socket</u>

Demultiplexing senza connessione:UDP



Esempio

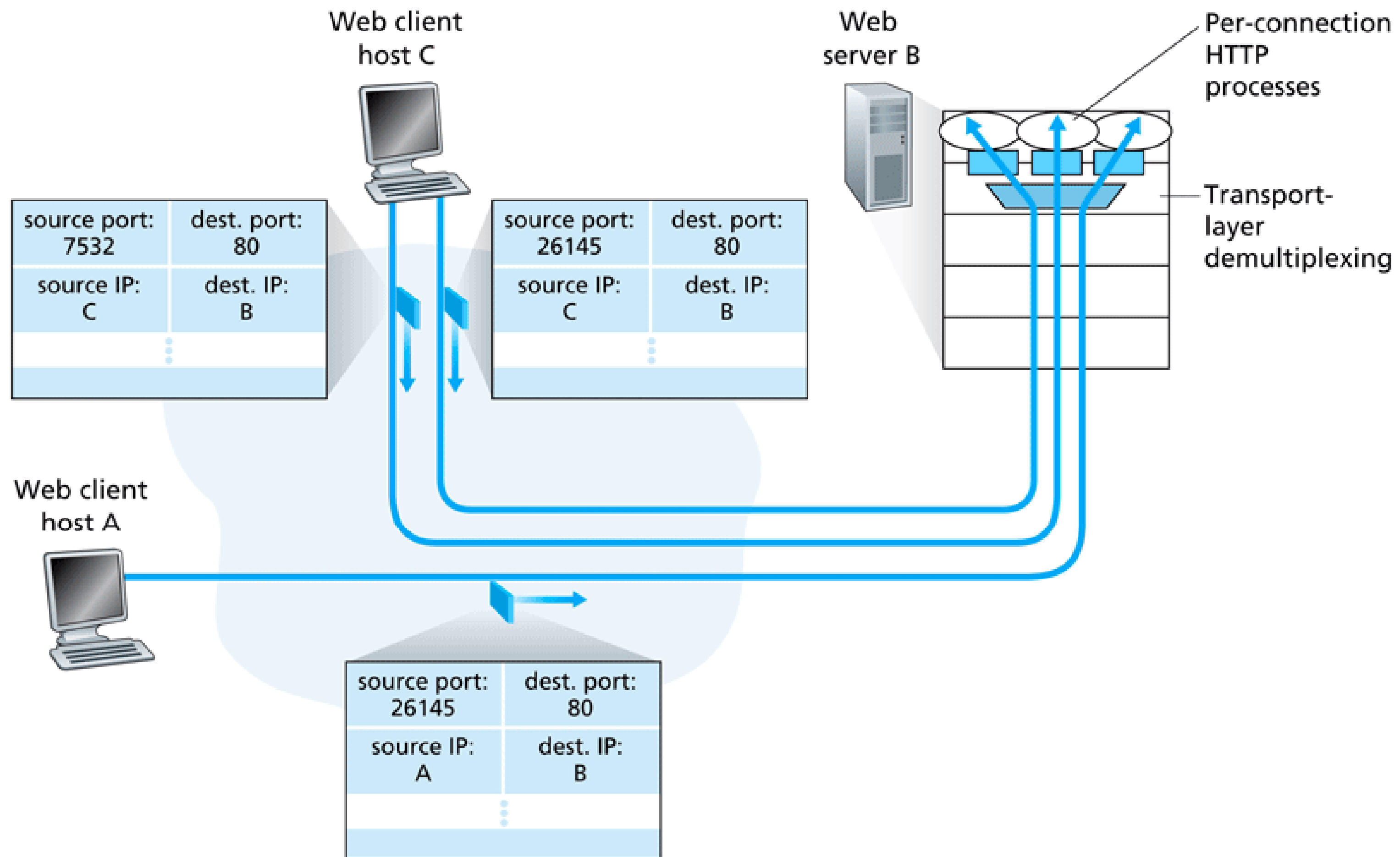
```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



Demultiplexing orientato alla connessione: TCP

- Nel protocollo TCP la socket 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: TCP



Differenze socket TCP e UDP

- La socket UDP è identificata da 2 parametri:
  - Indirizzo IP di destinazione
  - Numero della porta di destinazione
- 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
- In UDP due segmenti in arrivo con indirizzo IP e/o porta d'origine diversa ma con lo stesso numero di porta destinazione saranno smistati alla stessa applicazione
- In TCP due segmenti in arrivo con indirizzi IP o porta d'origine diversi saranno diretti a due socket differenti

- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- **Trasporto senza connessione: UDP**
- Principi del trasferimento dati affidabile
- Protocolli affidabili con pipeling

UDP: User Data Protocol

- UDP è un protocollo di trasporto “senza fronzoli”
  - Definito nell’RFC 768
- Offre un servizio di consegna “a massimo sforzo” (best effort)
- Non c’è garanzia sulla consegna dei segmenti
  - Possono perdersi
  - Possono essere consegnati fuori sequenza all’applicazione
- Senza connessione:
  - No handshaking tra mittente e destinatario UDP
  - Ogni segmento UDP è gestito indipendentemente dagli altri

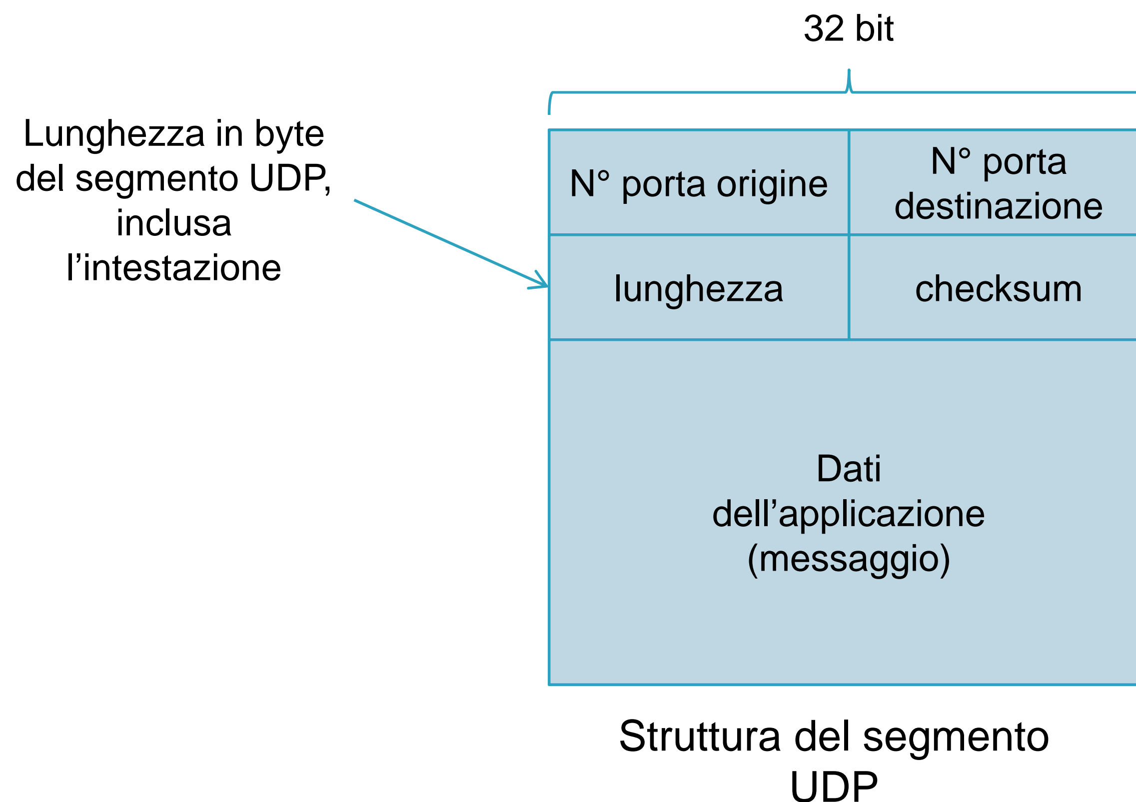
Ma allora perché usare UDP?

- Nessuna connessione stabilita
  - UDP non introduce ritardo nello stabilire una connessione
- Nessuno stato di connessione nel mittente e destinatario
  - Non ci sono buffer di ricezione e di invio, parametri per il controllo della congestione, sequenze di ACK
- Intestazioni di segmento corte
  - L'intestazione dei pacchetti TCP è 20 byte mentre in UDP 8 byte
- Senza controllo di congestione: UDP può sparare dati a raffica
  - DNS non produce ritardo nello stabilire una connessione

C'è un motivo...

- Utilizzato spesso nelle applicazioni multimediali poiché
  - Tollera piccole perdite
  - Sensibile alla frequenza
- Altri impieghi di UDP
  - DNS
  - SNMP
- Se pretendiamo un trasferimento affidabile con UDP allora bisogna aggiungere affidabilità al livello di applicazione
  - Recupero degli errori nelle applicazioni! Ma ci conviene?

Struttura dei segmenti UDP



Struttura del segmento UDP

Checksum UDP

- UDP fornisce un semplice meccanismo per rilevare gli errori nel segmento trasmesso (“bit alterati”)

Mittente:

- Tratta il contenuto del segmento come una sequenza di interi da 16 bit
- Il checksum si calcola come somma (complemento a 1) dei 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?* Altro più avanti …

Esempio di calcolo di checksum in UDP

- Supponiamo che il mittente deve spedire le tre parole da 16 bit

  0110011001100000
  0101010101010101
  1000111100001100

- Somma le prime due è

  0110011001100000
  0101010101010101
  ―――――――――――――――――
  1011101110110101

- Somma la terza al risultato ottenuto:

  1000111100001100
  1011101110110101
  ―――――――――――――――――
  10100101011000001

- Il riporto dell'ultima somma è sommato al primo bit

  0100101011000001
  0000000000000001
  ―――――――――――――――――
  0100101011000010

Esempio di calcolo di checksum in UDP

- Il valore finale è complementato bit a bit ottenendo

  1011010100111101

- In ricezione si sommano le tre parole iniziali e la checksum
- Se non ci sono errori allora l'addizione darà come risultato

  1111111111111111

- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP
- **Principi del trasferimento dati affidabile**
- Protocolli affidabili con pipeling

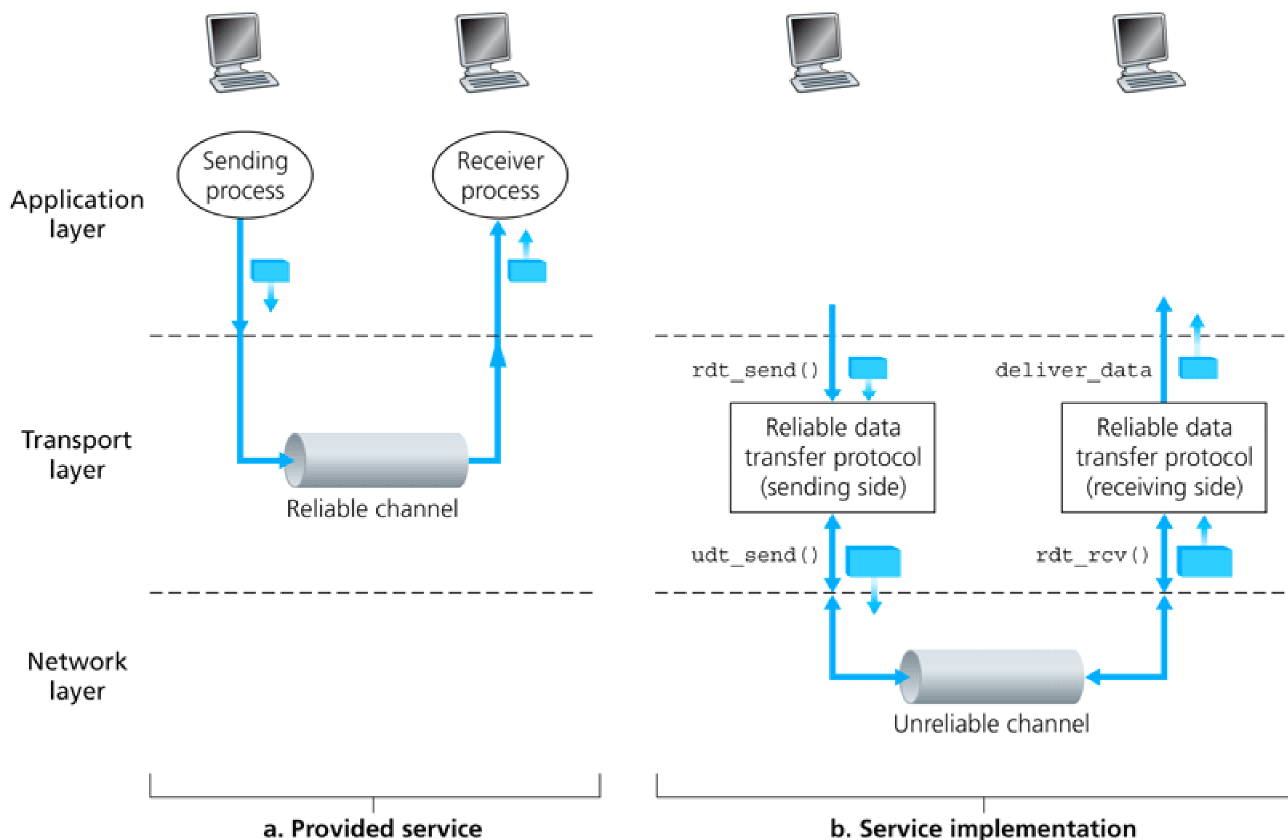
Protocolli di trasferimento dati affidabile

- Implementare un servizio di trasferimento affidabile considerando l'inaffidabilità del livello "al di sotto"
- TCP è un protocollo di trasferimento dati affidabile implementato in cima al livello di rete punto-punto non affidabile (IP)
- Lo scopo è sviluppare un protocollo dati affidabile considerando modelli via via più complessi

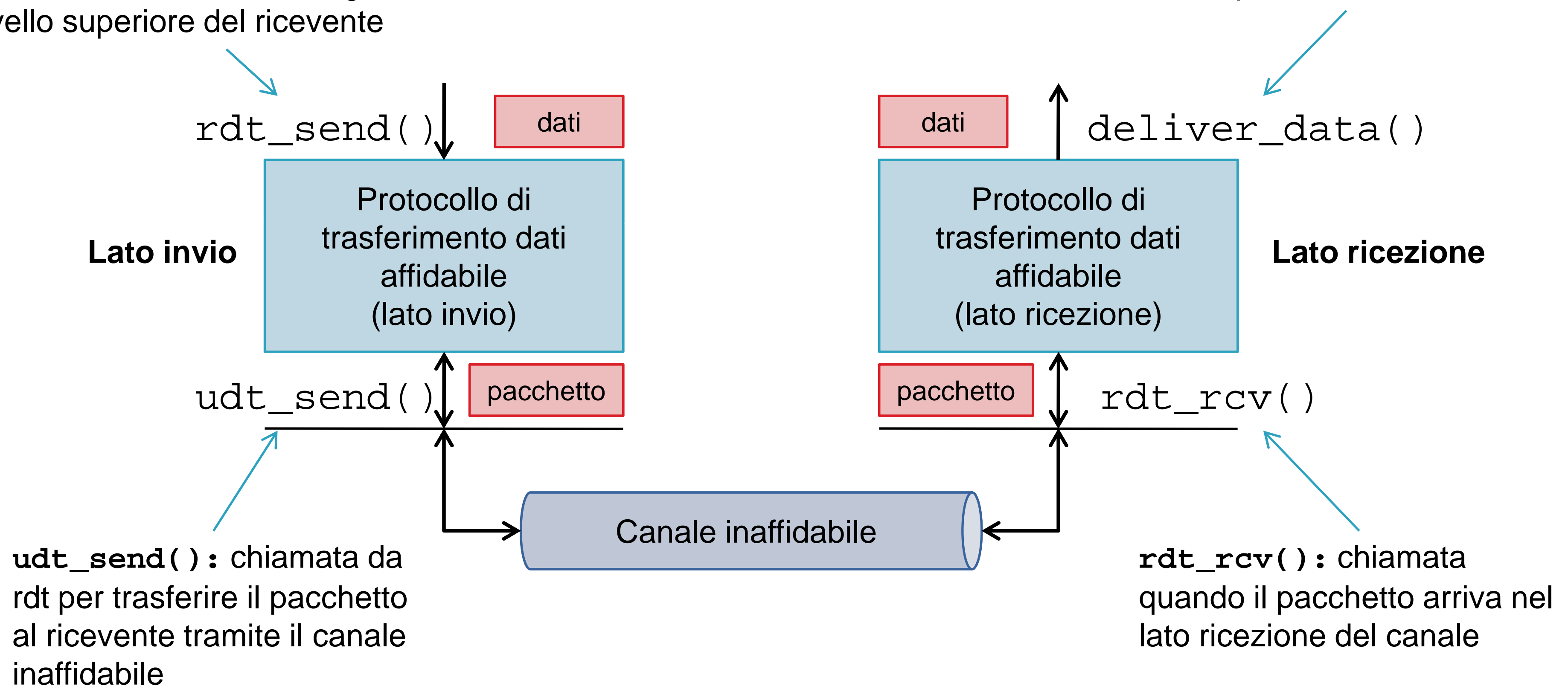
Un protocollo Reliable Data Transfer

- Vogliamo sviluppare un protocollo affidabile chiamato *reliable data transfer* (RDT)
- Il protocollo ha le seguenti interfacce:
  - `rdt_send():` chiamata dall'alto, (ad es. dall'applicazione). Trasferisce i dati da consegnare al livello superiore del ricevente
  - `udt_send():` chiamata da RDT per trasferire il pacchetto al ricevente tramite il canale inaffidabile
  - `deliver_data():` chiamata da RDT per consegnare i dati al livello superiore
  - `rdt_rcv():` chiamata quando il pacchetto arriva nel lato ricezione del canale

Trasferimento dati affidabile

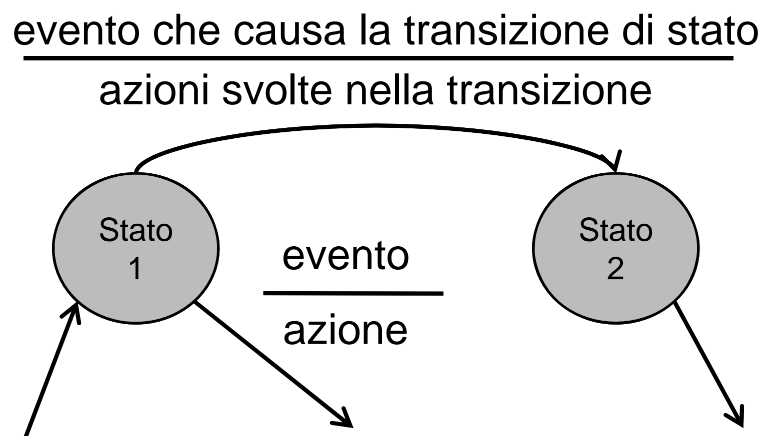


Trasferimento dati affidabile: preparazione



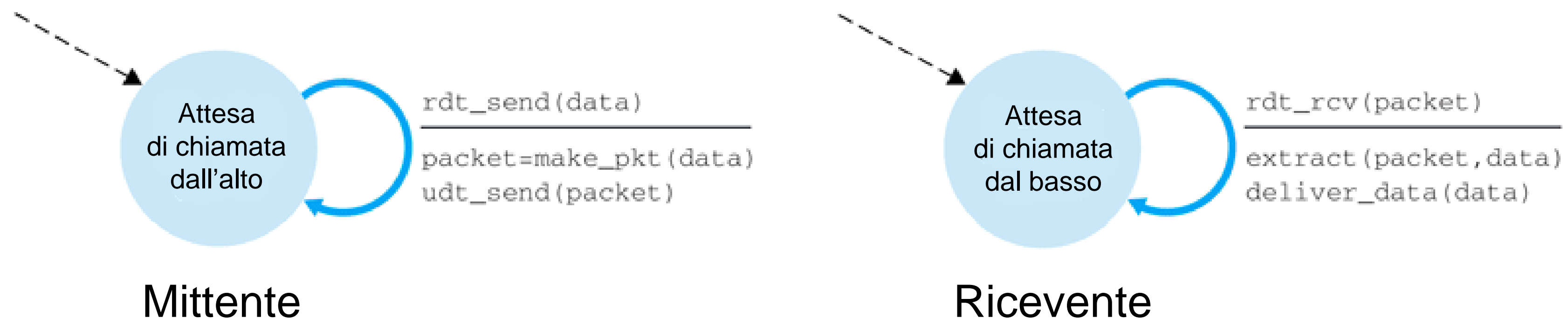
Automi a stati finiti

- 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

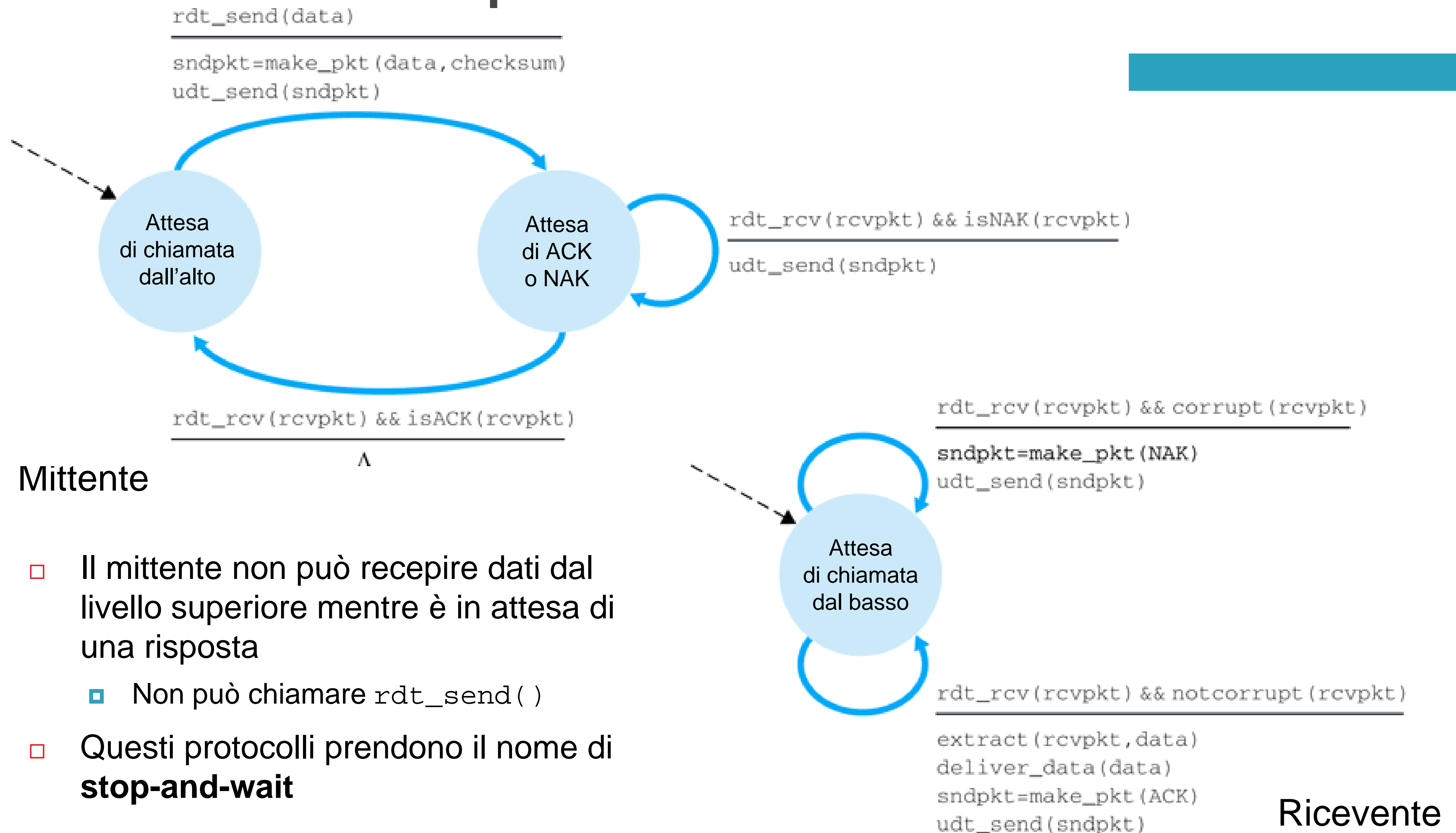
- Canale sottostante perfettamente affidabile
  - Nessun errore nei bit
  - Nessuna perdita di pacchetti
- Automa distinto per il mittente e per il ricevente:
  - Il mittente invia i dati nel canale sottostante
  - Il ricevente legge i dati dal canale sottostante



RDT2.0: canale con errori nei bit

- Il canale sottostante potrebbe confondere i bit nei pacchetti
  - Checksum per rilevare gli errori nei bit
- In che modo possiamo 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
- Abbiamo bisogno di nuovi meccanismi in RDT2.0 (oltre a RDT1.0):
  - Rilevamento di errore
  - Feedback del destinatario: messaggi di controllo (ACK, NAK) dal ricevente al mittente

RDT2.0: Operazioni senza errori



- Il mittente non può recepire dati dal livello superiore mentre è in attesa di una risposta
  - Non può chiamare `rdt_send()`
- Questi protocolli prendono il nome di **stop-and-wait**

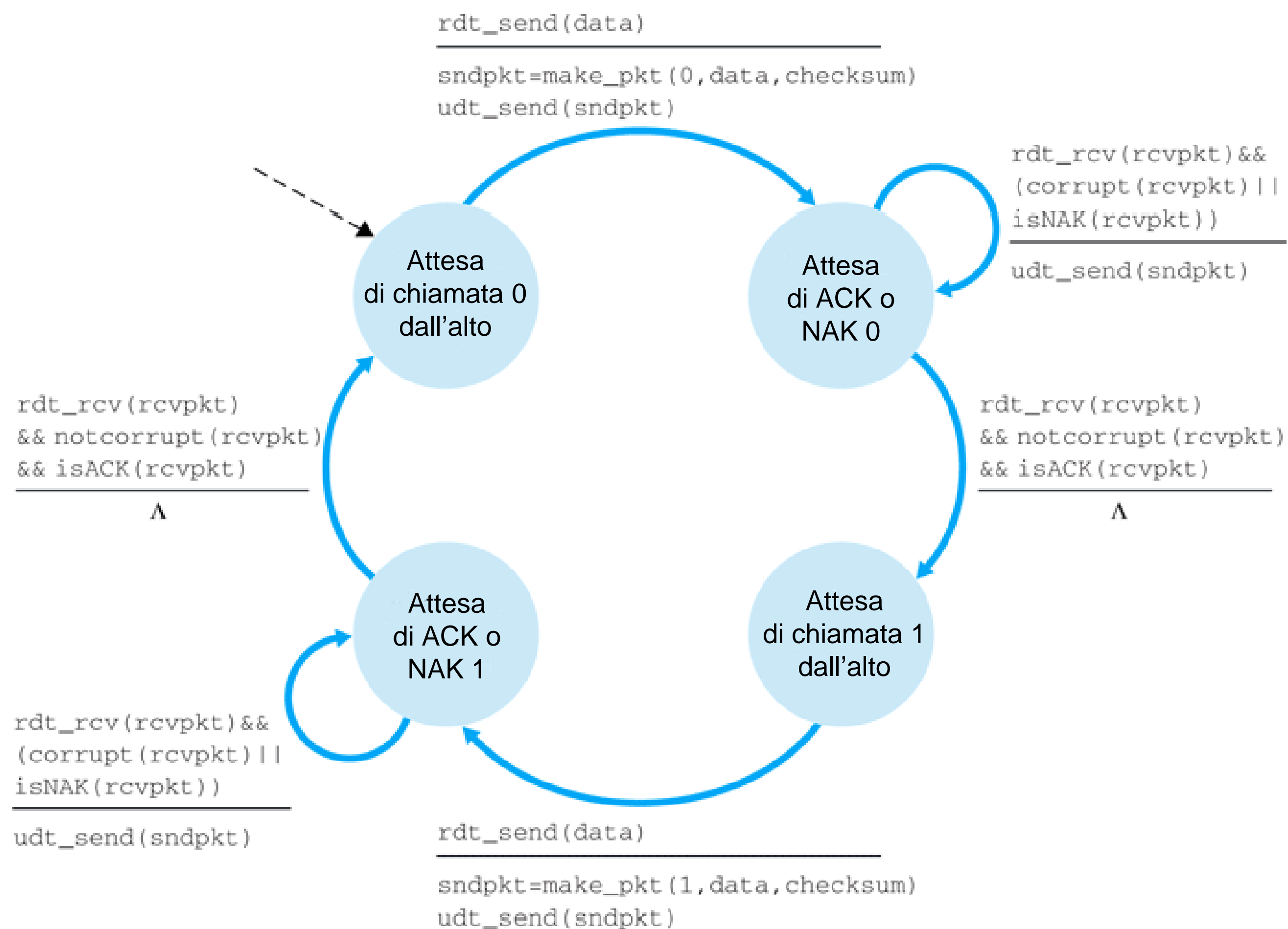
RDT2.0: Difetti

- 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
    - E' sufficiente un contatore ad 1 bit
  - Il ricevente scarta il pacchetto duplicato

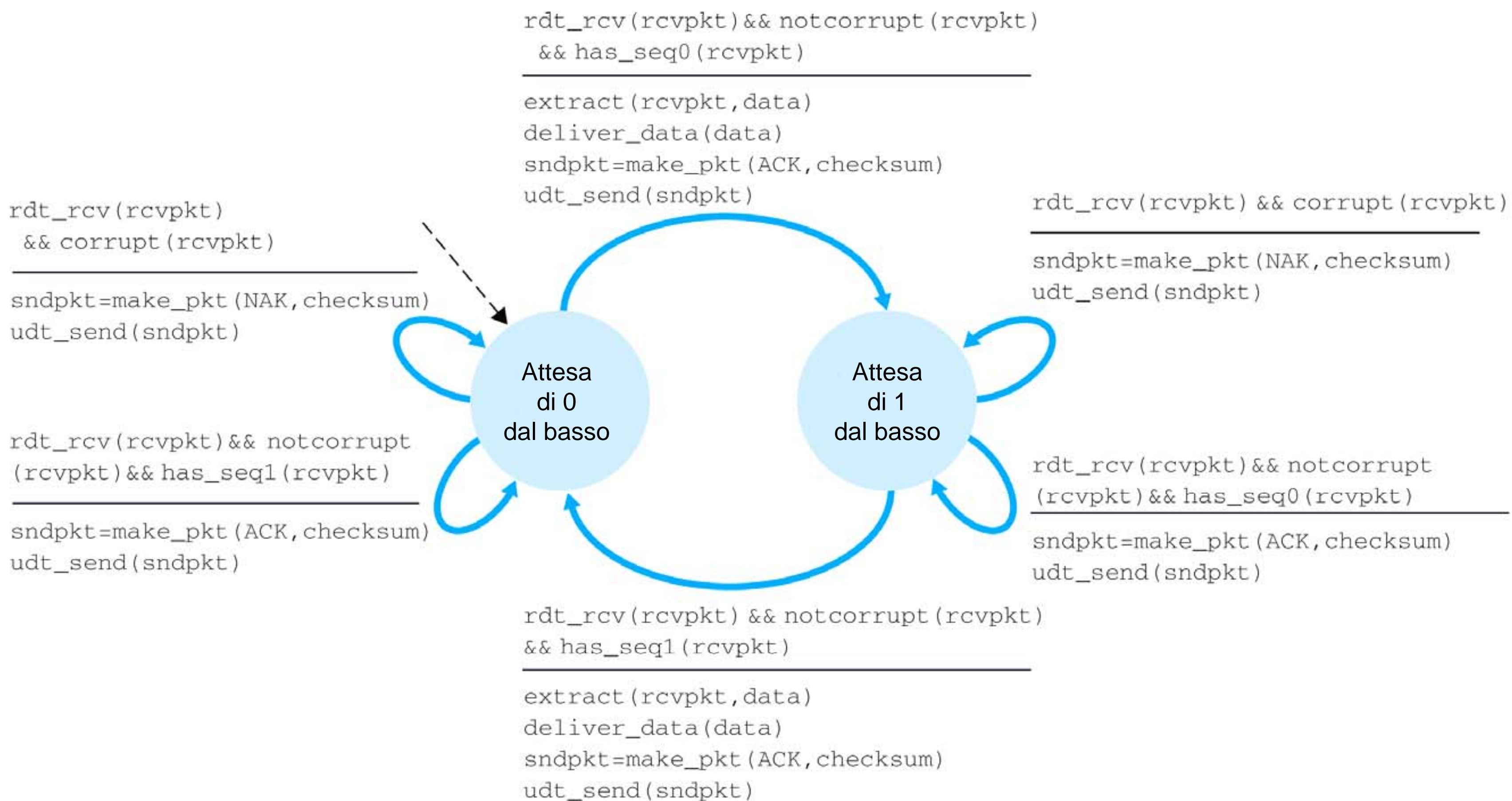
RDT2.1

- Mittente:
  - Aggiunge il numero di sequenza al pacchetto
  - Sono sufficienti due numeri di sequenza 0 e 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
- Il ricevente non può sapere se il suo ultimo ACK/NAK è stato ricevuto correttamente dal mittente

RDT2.1:Il mittente gestisce gli ACK/ NAK alterati



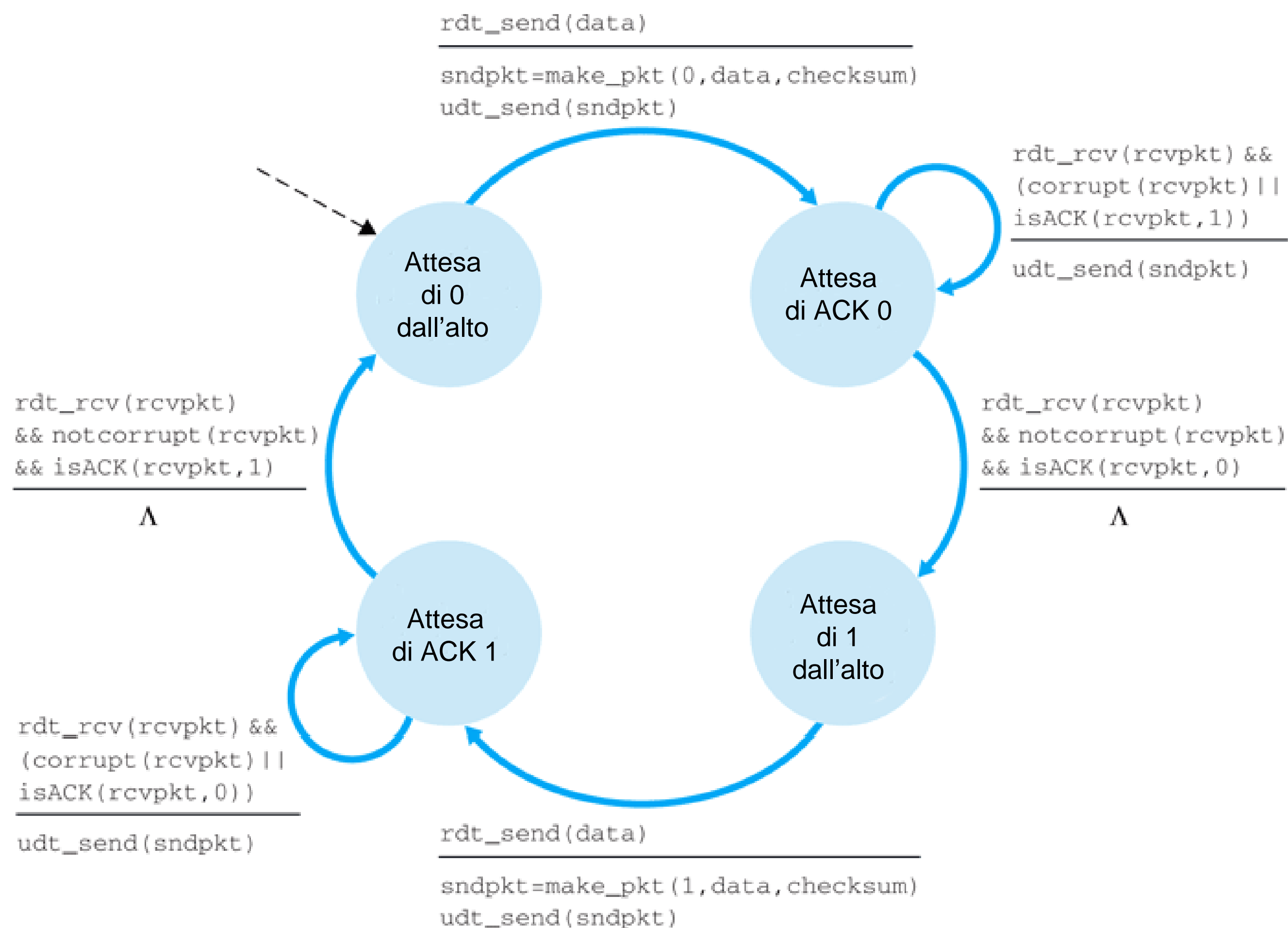
RDT2.1: il ricevente gestisce gli ACK/NAK alterati



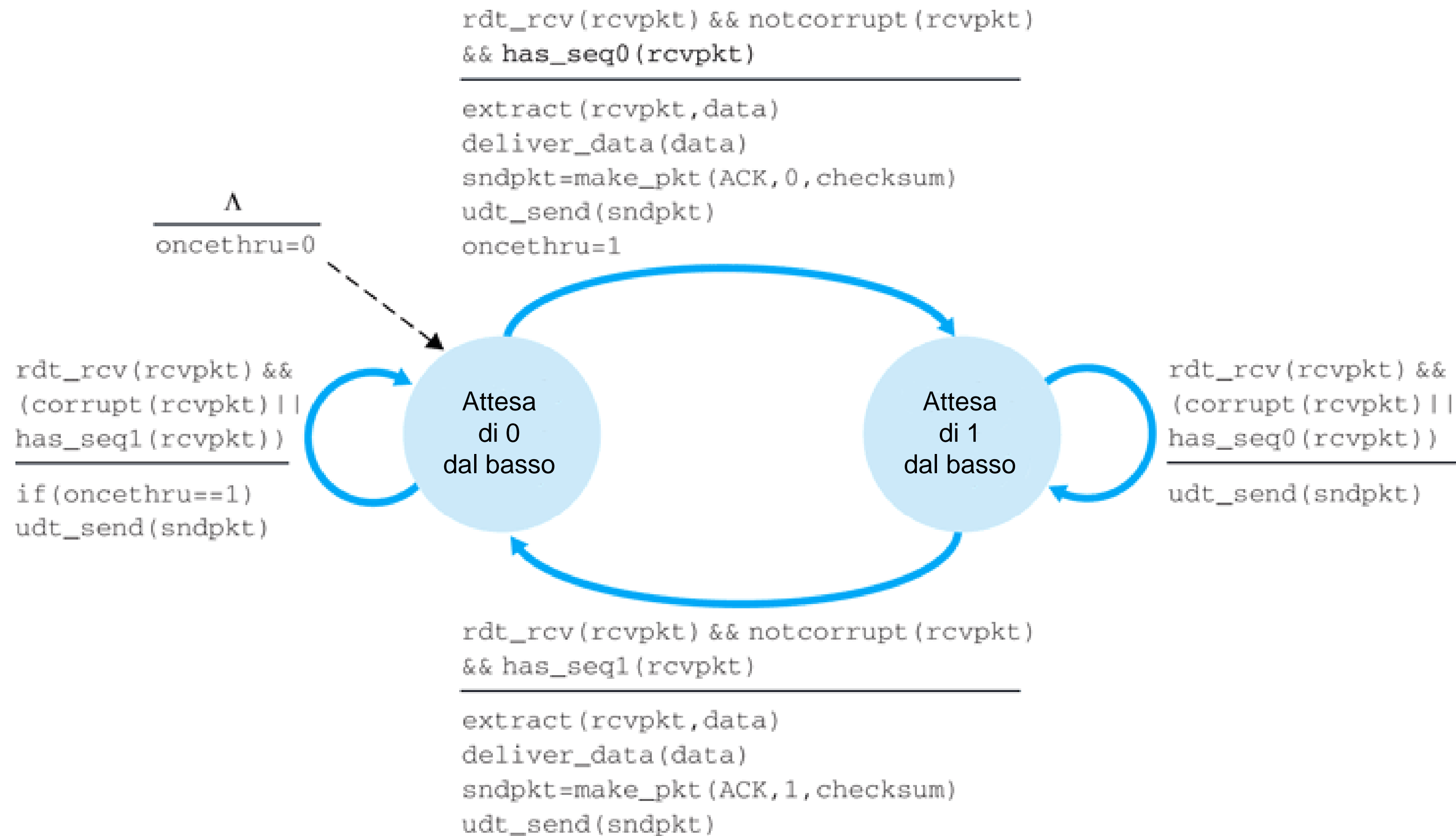
RDT2.2: un protocollo senza NAK

- 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: Mittente privo di NAK



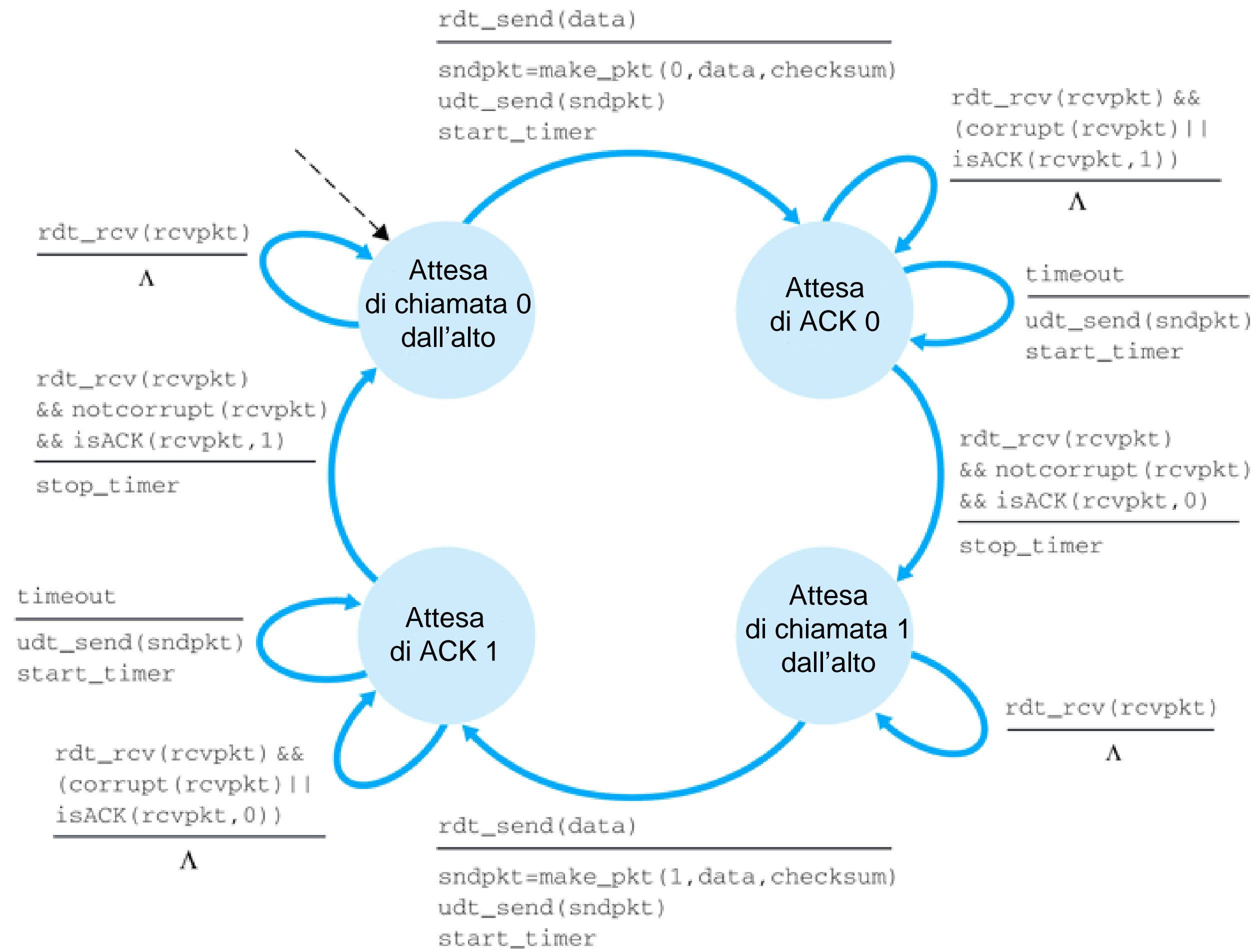
RDT2.2: Destinatario privo di NAK



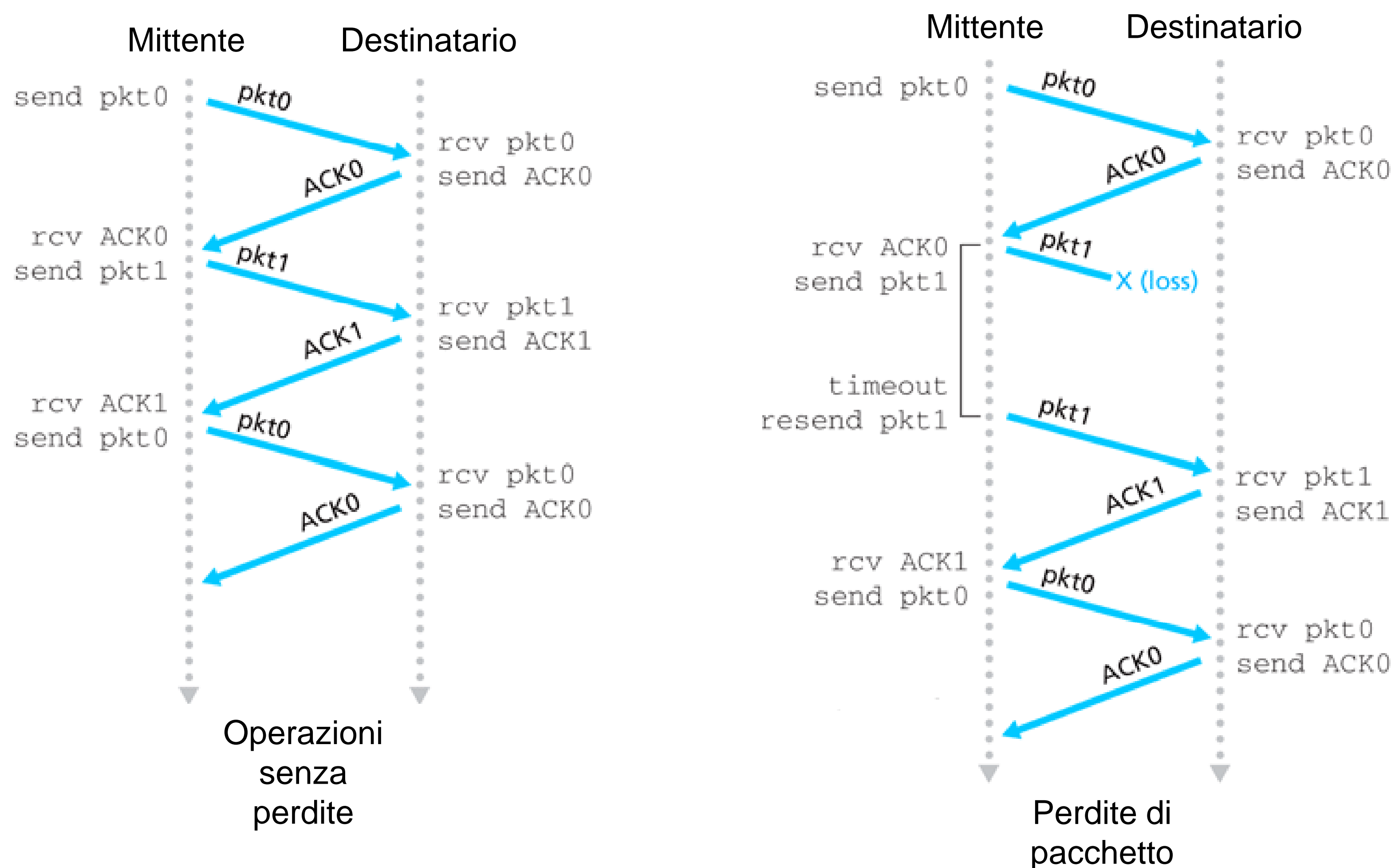
RDT3.0: canali con errori e perdite

- Il canale sottostante può anche smarrire i pacchetti (dati o ACK)
  - checksum, numero di sequenza, ACK e ritrasmissioni aiuteranno, ma non saranno sufficienti
- Il mittente attende un ACK per un tempo "ragionevole"
- Se non riceve un ACK in questo periodo ritrasmette
- 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
- E' necessario un contatore (countdown timer)

RDT3.0



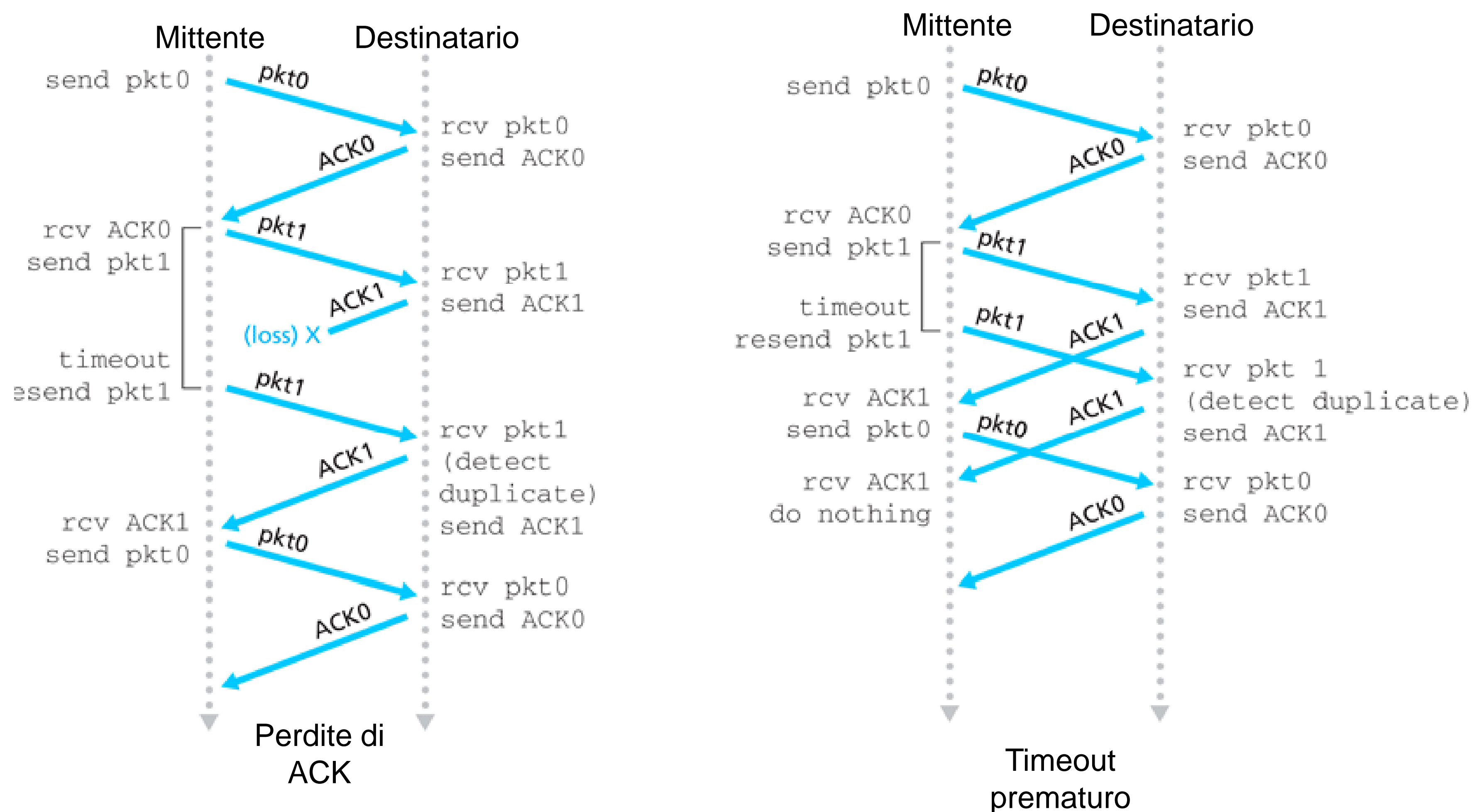
RDT3.0 in azione



Operazioni senza perdite

Perdite di pacchetto

RDT3.0 in azione



Prestazioni di RDT3.0 - 1

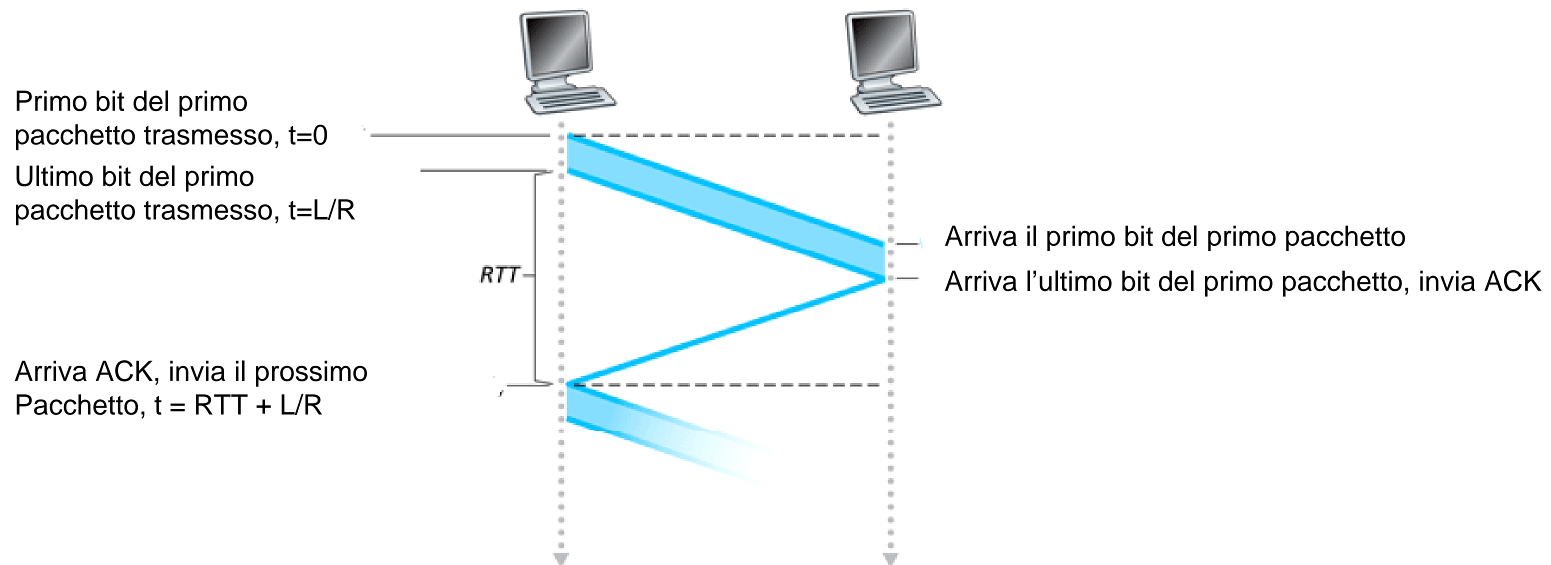
- Il protocollo RDT3.0 è corretto ma non è efficiente
- Supponiamo di avere:
  - Un collegamento di R=1Gbps
  - Un ritardo di propagazione di 15ms
  - Pacchetti di L=1KB
- Il tempo di andata e ritorno (round trip time – RTT) è di circa 30 millisecondi
- In un protocollo stop-and-wait un pacchetto giunge al destinatario all'instante

$$t = \mathrm{RTT}/2 + \mathrm{L}/\mathrm{R} = 15{,}008\mathrm{ms}$$

Prestazioni di RDT3.0 - 2

- Il tempo impiegato per trasmettere un pacchetto è di

$$t_{trasm} = \frac{L}{R} = \frac{8000 \text{ bit/pacchetto}}{10^9 \text{ bit/sec}} = 8 \text{ microsecondi}$$



- In un arco di tempo di 30,008ms il mittente ha trasmesso solo oer 0,008ms

- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP
- Principi del trasferimento dati affidabile
- **Protocolli affidabili con pipeling**

Prestazioni di RDT3.0 - 3

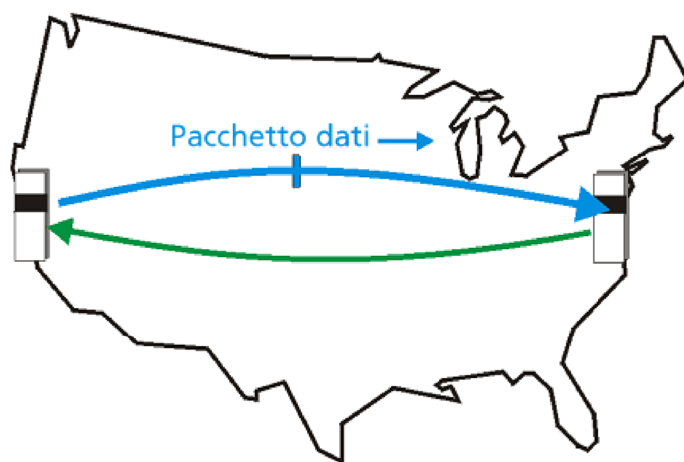
- L'**utilizzo** è la frazione di tempo in cui il mittente effettivamente è occupato nell'invio di bit sul canale ed è dato da

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

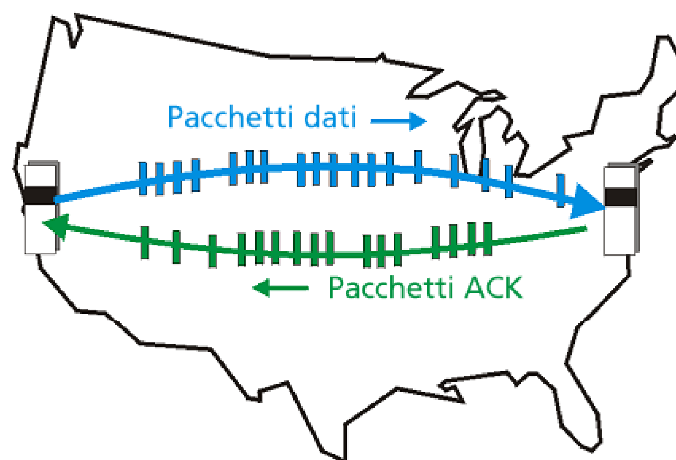
- Solo 2,7 centesimi dell'1% del tempo è stato utilizzato dal mittente
- Il mittente ha spedito solo 1000 byte in 30,008ms con un throughput di 267Kbps…ma il collegamento era di 1Gbps!
- Il protocollo di rete limita l'uso delle risorse fisiche

Protocolli con pipeline

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

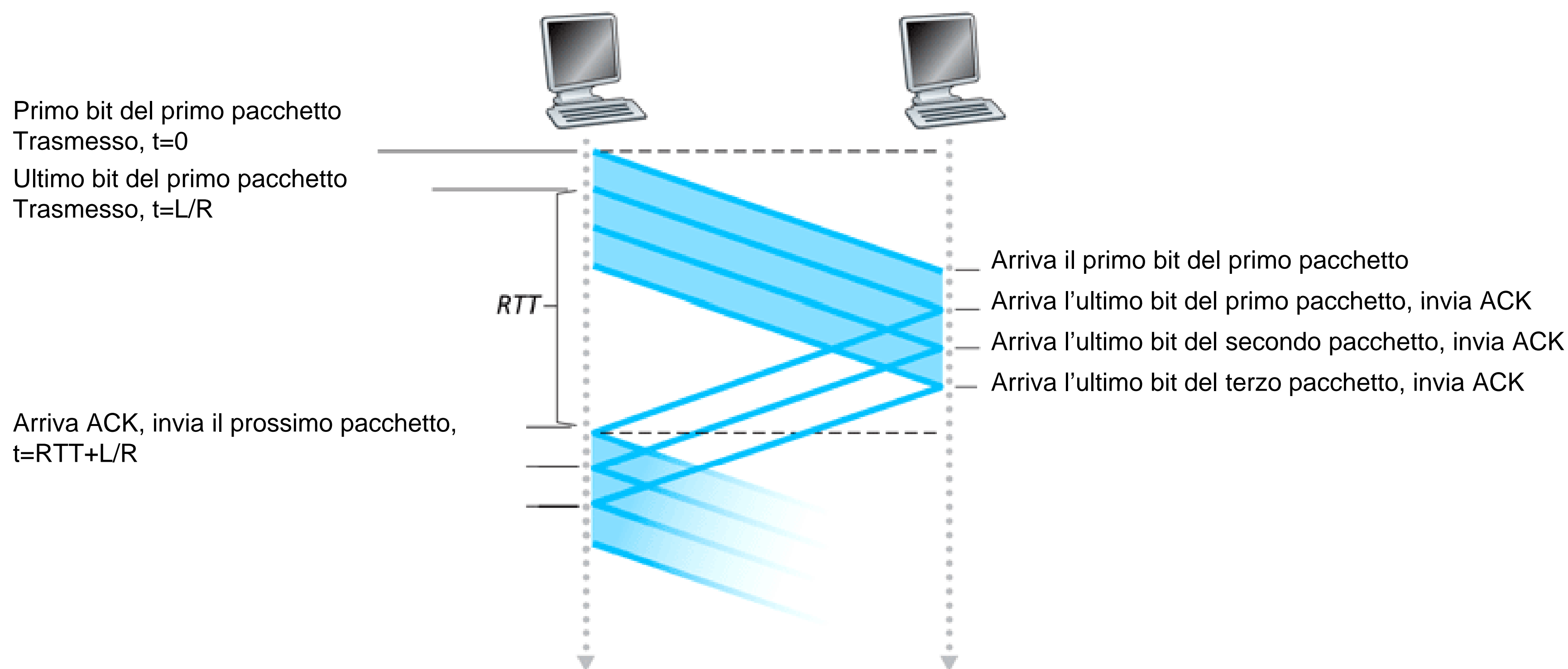


a) Protocollo stop-and-wait all'opera



b) Protocollo con pipeline all'opera

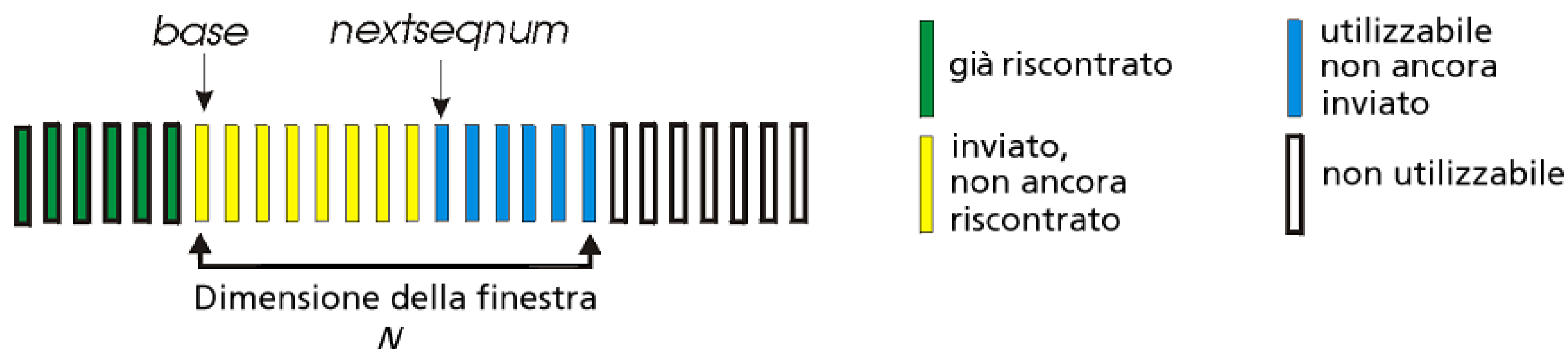
Pipeling: aumento dell’utilizzo



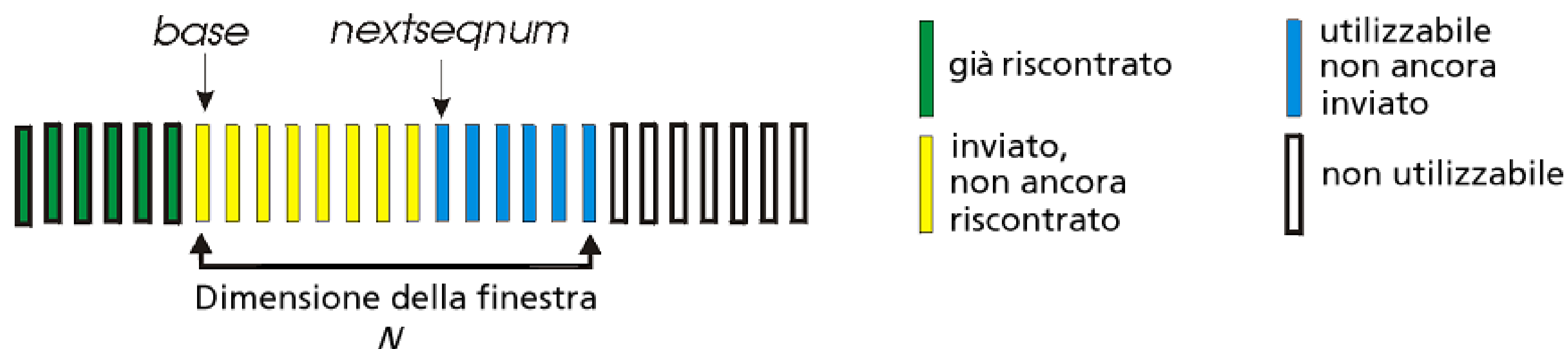
$$U_{mit} = \frac{3 \times L/R}{RTT + L/R} = \frac{3 \times 0{,}008}{30{,}008} = 0{,}0008 \text{ microsecondi}$$

Go-Back-N

- Il protocollo **Go-Back-N** (GBN) permette al mittente di inviare più pacchetti senza attendere la notifica di ricezione
- Per ogni pacchetto nell'intestazione c'è un numero di sequenza a k bit
- La "finestra" contiene fino a *N* pacchetti consecutivi non riscontrati
  - *base* il numero di sequenza più vecchio
  - *nextseqnum* il più piccolo numero di sequenza inutilizzato



Go-Back-N

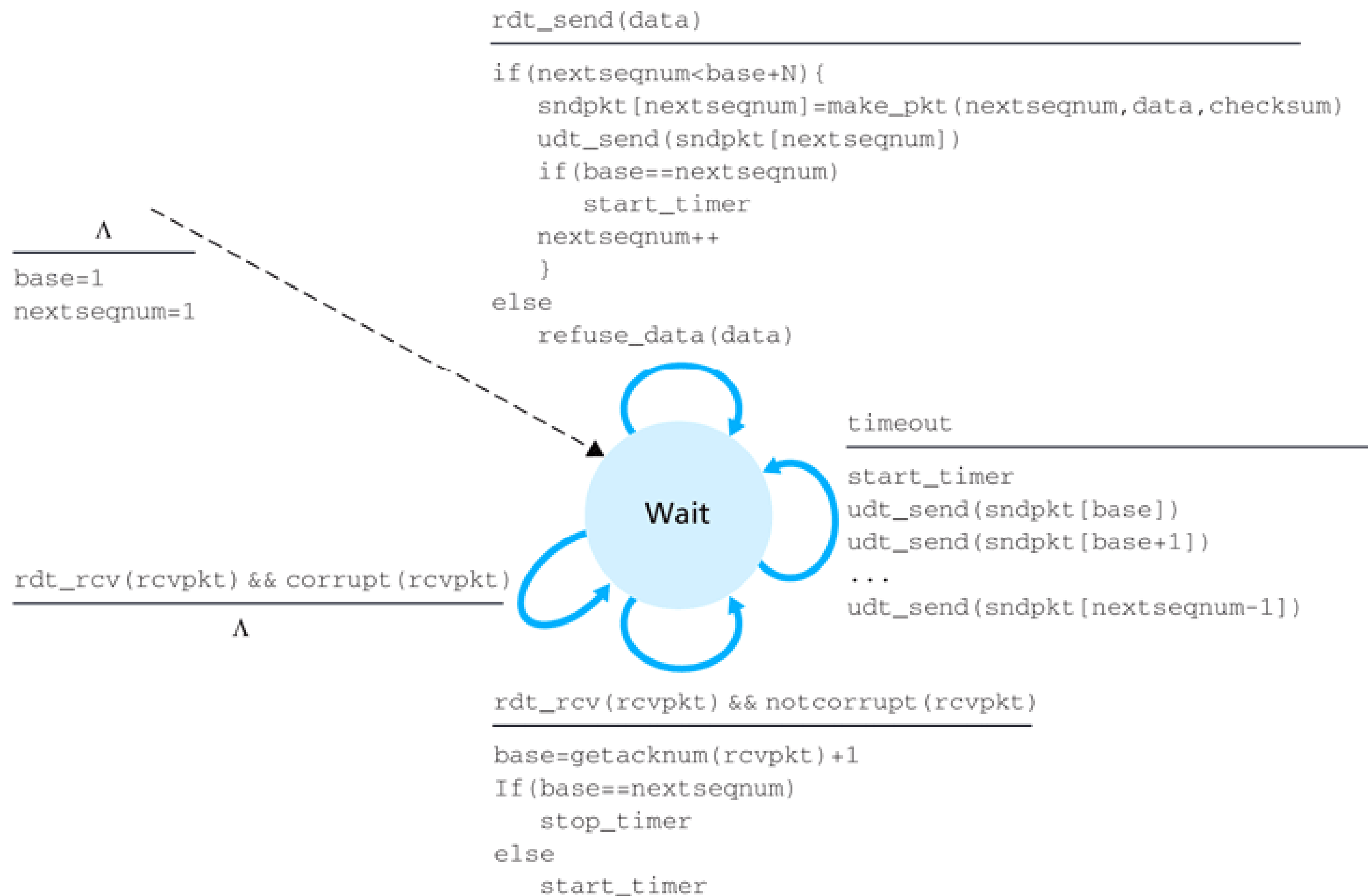


- [0, *base*-1]
  - Pacchetti trasmessi e riscontrati
- [*base*, *nexseqnum*-1]
  - Pacchetti trasmessi e non riscontrati
- [*nexseqnum*, *base*+*N*-1]
  - Slot disponibili
- >*base* + *N*
  - Non disponibili

Go-Back-N: mittente

- Il mittente risponde a tre tipi di eventi:
  - **Invocazione dall'alto**: per ogni chiamata di `rtd_send()` si verifica che la finestra non sia piena, se è piena l'applicazione deve attendere.
  - **Ricezione di un ACK**: Riceve un ACK(*n*) come riscontro per tutti i pacchetti con numero di sequenza minore o uguale a *n* – (**riscontri cumulativi**)
    - Pacchetti duplicati potrebbero essere scartati (vedere il ricevente)
  - **Evento di timeout**: ritrasmette il pacchetto *n* e tutti i pacchetti con i numeri di sequenza più grandi nella finestra

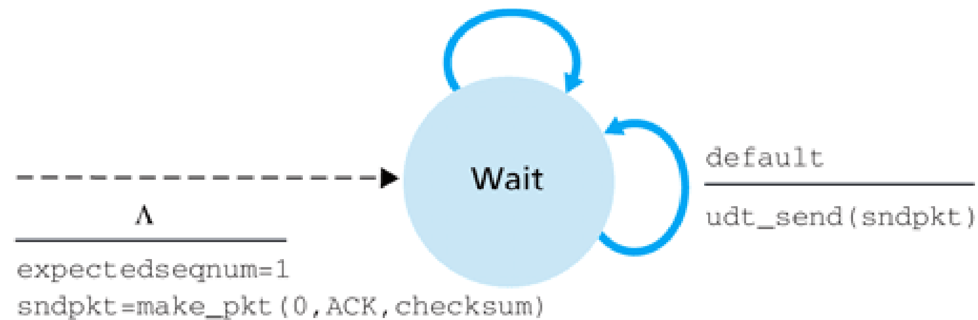
Go-Back-N: automa esteso del mittente



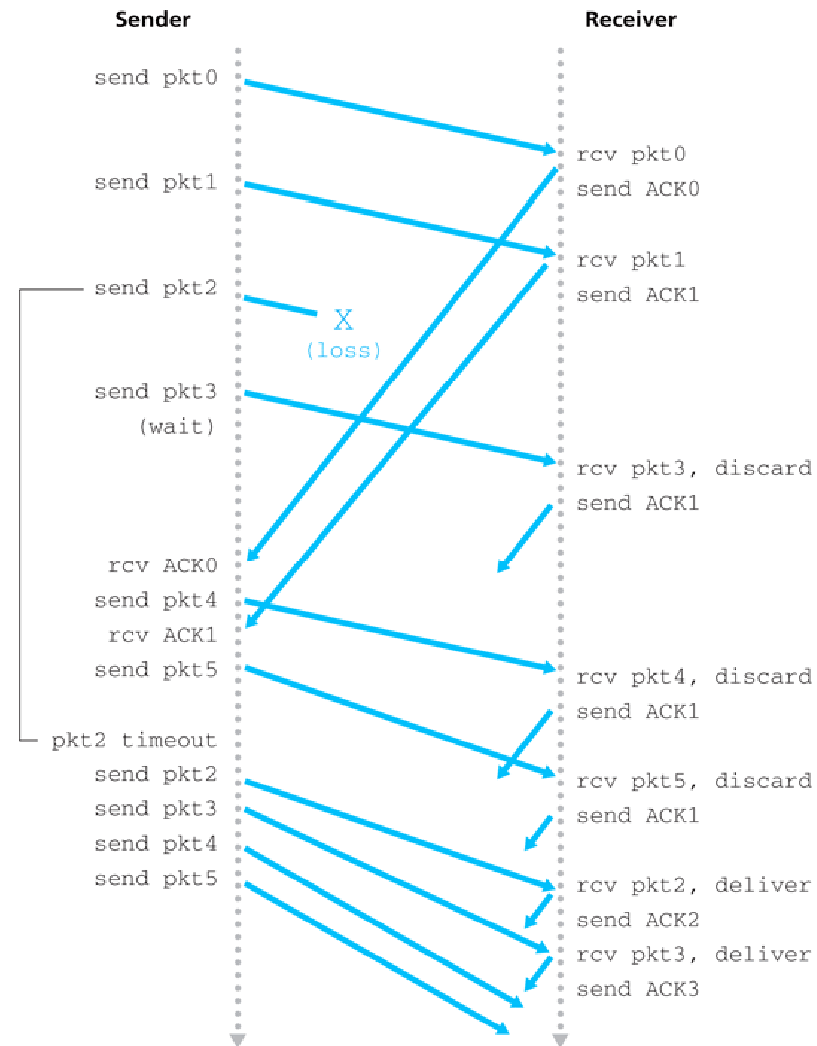
Go-Back-N: destinatario

- Il destinatario si aspetta di ricevere i pacchetti corretti ed in ordine
- Per ogni pacchetto ricevuto:
  - Se è in sequenza
    - 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`**
  - Se è fuori sequenza
    - Scartato (non è salvato) $\Rightarrow$ senza buffering del ricevente!
    - Rimanda un ACK per il pacchetto con il numero di sequenza più alto *in sequenza*

GBN: automa esteso del ricevente



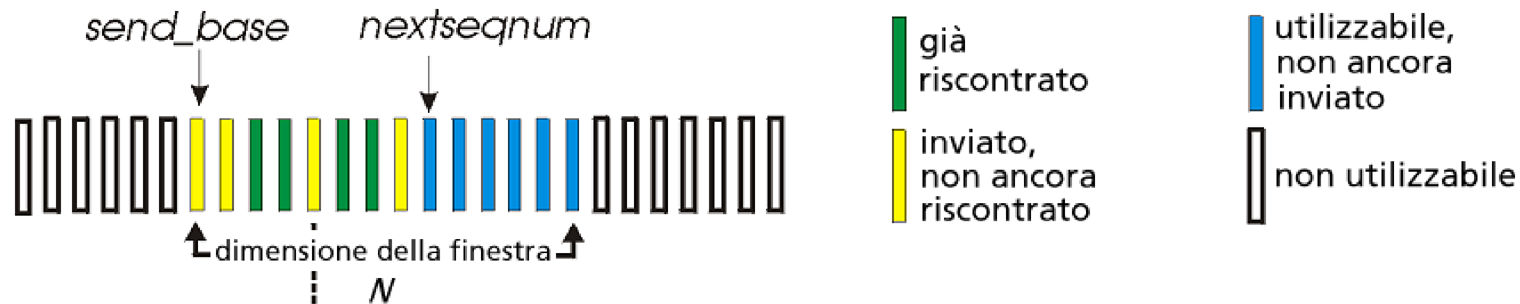
GBN in azione



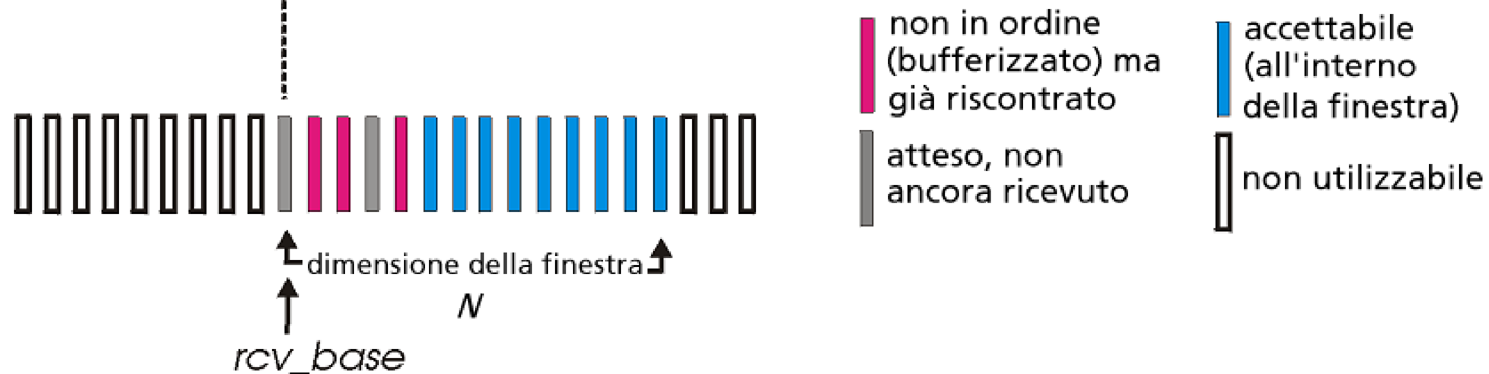
Ripetizione selettiva

- Nei protocolli a **ripetizione selettiva** si evitano le ritrasmissioni non necessarie
- 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 e Destinatario

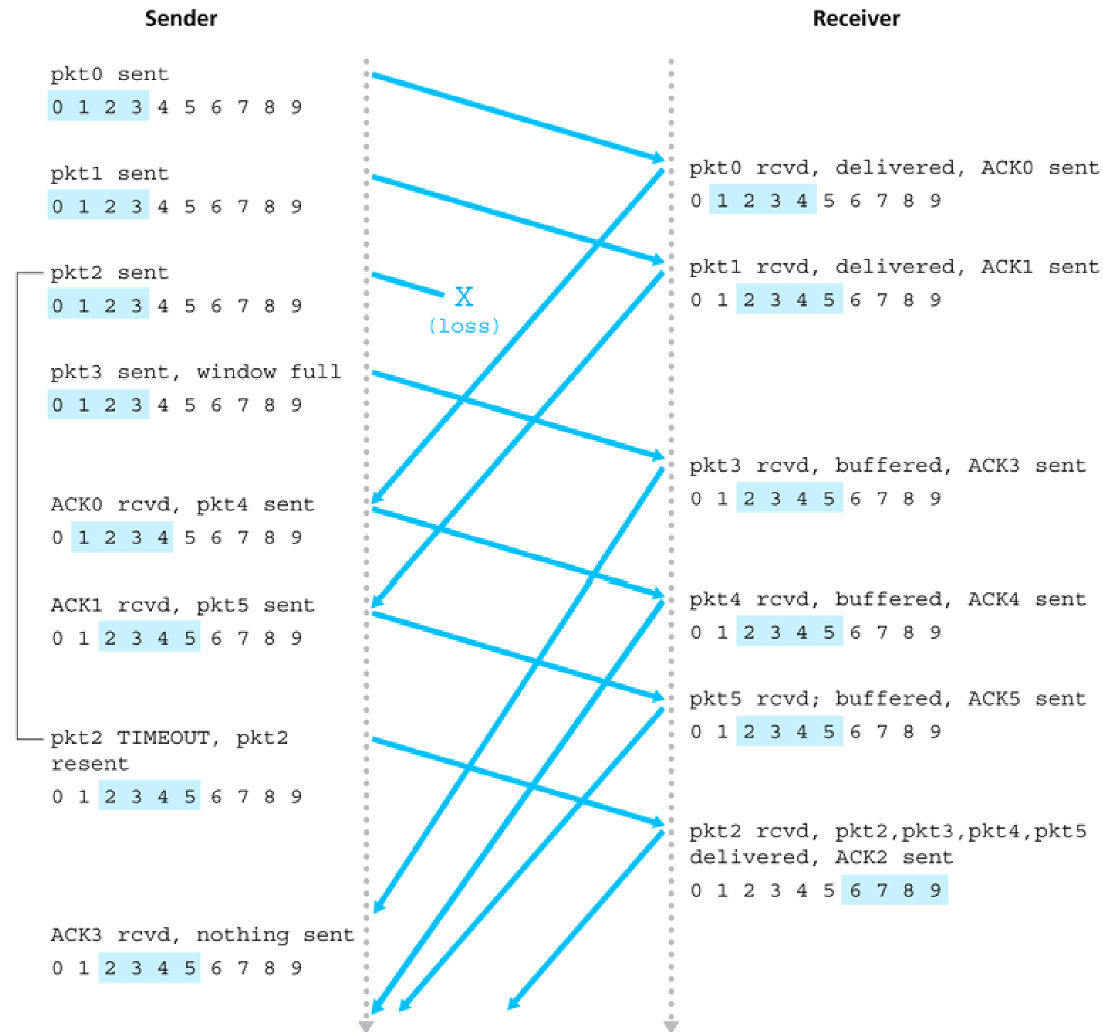
Mittente

- Dati dall'alto:
  - Se nella finestra è disponibile il successivo numero di sequenza, invia il pacchetto
- Timeout(n):
  - Ogni pacchetto n trasmesso il ha un 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

Destinatario

- Pacchetto n in [rcvbase, rcvbase+N-1]
  - Invia ACK(n)
  - Fuori sequenza: buffer
  - In sequenza: 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

- Assumiamo che:
  - Numeri di sequenza: 0, 1, 2, 3
  - Dimensione della finestra = 3
- Supponiamo che il mittente invii tra pacchetti numerati 0,1,2. Due casi:
  - 1° caso: i riscontri di perdono il pkt0 viene rispedito
  - 2° caso: i riscontri arrivano ed il mittente invia il pkt3 e ptk0 ma pkt3 si perde e ptk0 arriva
- Il ricevente non vede alcuna differenza fra i due scenari!
- Passa erroneamente i dati duplicati come nuovi in (a)