Il Livello Trasporto

Obiettivi:

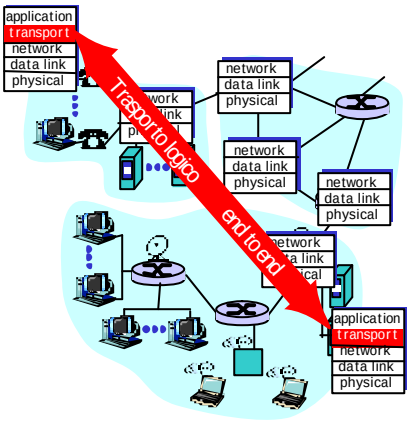
- Comprendere i principi costitutivi dei servizi del lievello trasporto:
  - multiplexing/demultiplexing
  - Trasf. dati affidabile
  - controllo flusso
  - controllo congestione
- Realizzazione in Internet

Panoramica:

- Servizi del livello trasporto
- multiplexing/demultiplexing
- trasporto connectionless : UDP
- principi del trasferimento dati affidabile
- Trasporto connection-oriented: TCP
  - trasferimento affidabile
  - controllo del flusso
  - Management della connessione
- principi controllo congestione
- controllo congestione del TCP

Servizi e protocolli di Trasporto

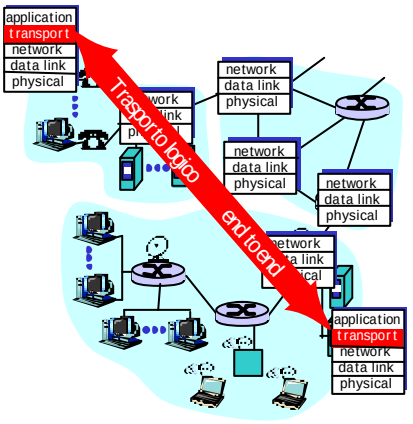
- Forniscono una *comunicazione logica* tra i processi applicativi in esecuzione su host differenti
- I protocolli di trasporto agiscono sugli end systems
- Servizi di trasporto e di rete:
- *Livello network:* trasferimento dati tra end systems
- *Livello trasporto:* trasferimento dati tra processi
  - Si appoggia su, e migliora, i servizi di livello network



Protocolli del Livello Trasporto

Servizi di trasporto Internet:

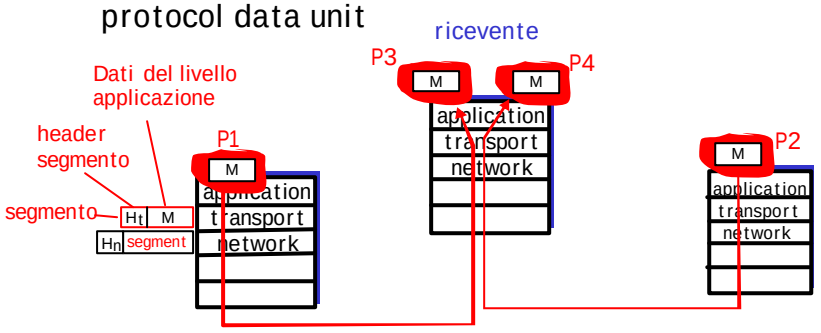
- Consegna affidabile e ordinata di tipo unicast (TCP)
  - congestione
  - Controllo del flusso
  - setup della connessione
- Consegna inaffidabile ("best-effort"), disordinata di tipo unicast o multicast: UDP
- servizi non disponibili:
  - real-time
  - garanzia sulla banda
  - multicast affidabile



Multiplexing/demultiplexing

Segmento – unità di dati scambiati tra entità di livello trasporto
  - TPDU: transport protocol data unit

Demultiplexing: consegna dei segmenti ricevuti ad opportuni processi di livello applicazione

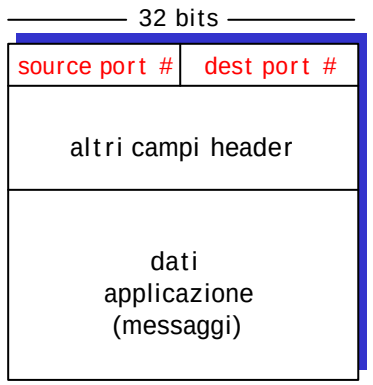


Multiplexing/demultiplexing

Multiplexing:
raccolta dati dai processi di applicazione, imbustamento dati con header (poi usati per il demultiplexing)

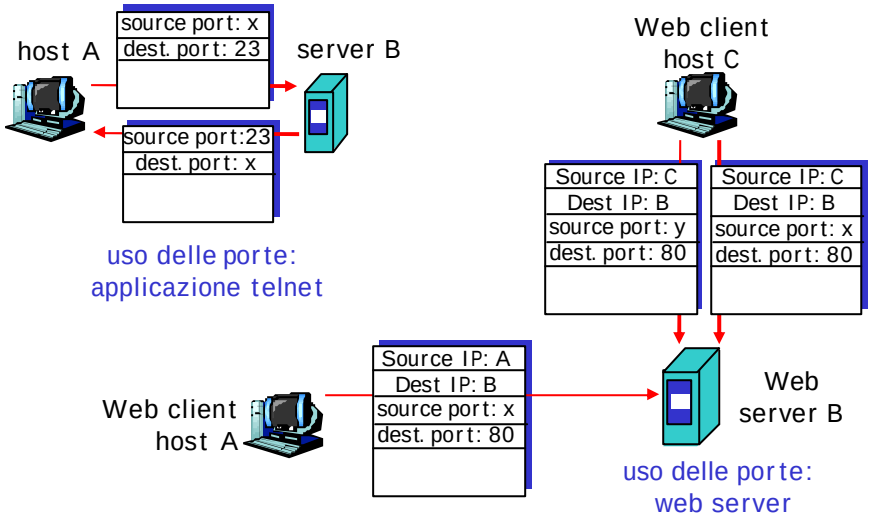
multiplexing/demultiplexing:

- Basati sui numeri di porta e gli indirizzi IP del mittente e del destinatario
  - porte di source, dest in ogni segmento
  - Porte well-known per applicazioni specifiche



Formato del segmento TCP/UDP

Multiplexing/demultiplexing: esempi



UDP: User Datagram Protocol [RFC 768]

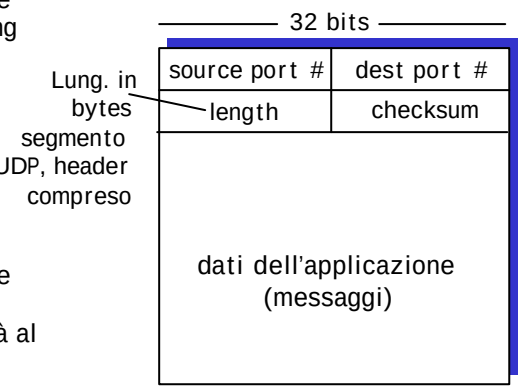
- Il protocollo di trasporto di Internet "senza fronzoli"
- Servizio "best effort", i segm. UDP possono essere:
  - persi
  - Consegnati all'appl. fuori ordine
- *connectionless:*
  - non c'è handshaking tra mittente e destinatario UDP
  - ogni segmento UDP viene trattato separatamente dagli altri

Perchè esiste UDP?

- Non occore stabilire una connessione (meno ritardi)
- semplicità: non occorre gestire la connessione
- L'header del segmento è piccolo
- Non c'è controllo della congestione: l'UDP può spedire dati tanto velocemente quanto lo si desidera

UDP (2)

- Spesso è utilizzato per le applicazioni con streaming multimedia che sono
  - loss tolerant
  - rate sensitive
- Altri usi UDP:
  - DNS
  - SNMP
- Trasferimento affidabile con UDP: aggiungere il controllo dell'affidabilità al livello applicazione
  - Recupero dell'errore specifico per l'applicazione!



formato segmento UDP

UDP checksum

Obiettivo: rilevare "errori" (per es., bit invertiti) nei segmenti trasmessi

Mittente:

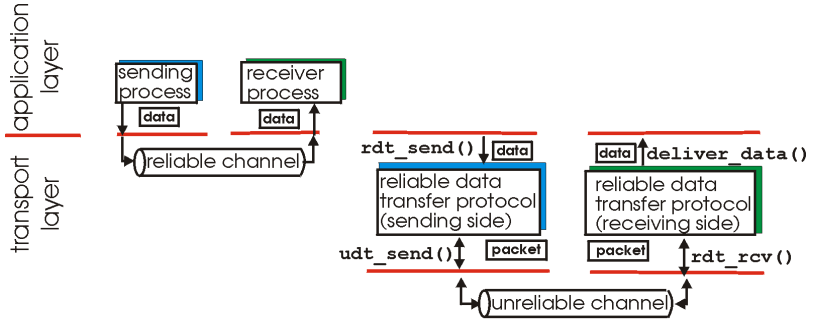
- contenuti del segmento trattati come sequenze di interi a 16-bit
- checksum: somma (in complemento a 1) dei contenuti del segmento
- Il mittente inserisce il valore del checksum nel campo checksum del pacchetto UDP

Destinatario:

- Calcola il checksum del segmento ricevuto
- Verifica se il checksum calcolato è uguale al valore del campo checksum:
  - NO – errore rilevato
  - YES - nessun errore *rilevato* (non vuol dire che non ci siano errori …)

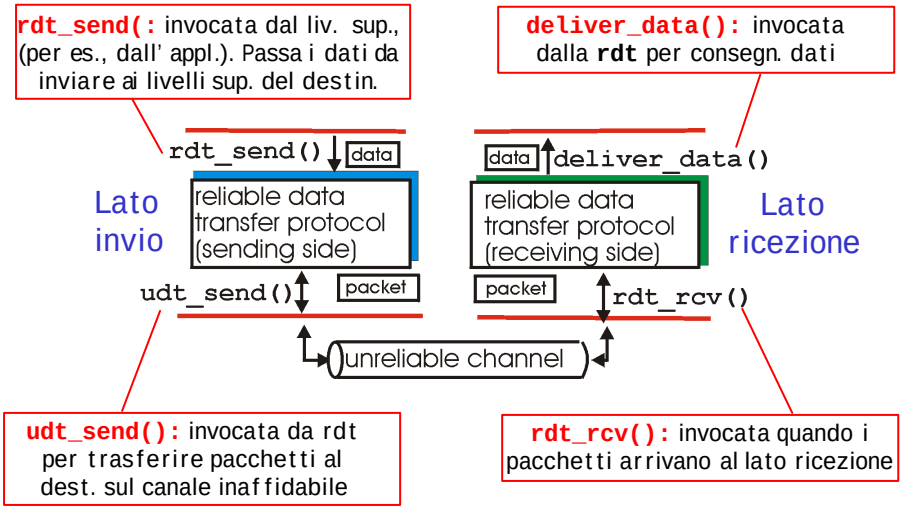
Principi di trasferimento affidabile

- Importante per i livelli applicazione, trasporto, data link
- È nella top-10 delle problematiche di rete importanti!



- Le caratteristiche di un canale inaffidabile determinano la complessità di un protocollo per il trasferimento affidabile dei dati (rdt – reliable data transfer)

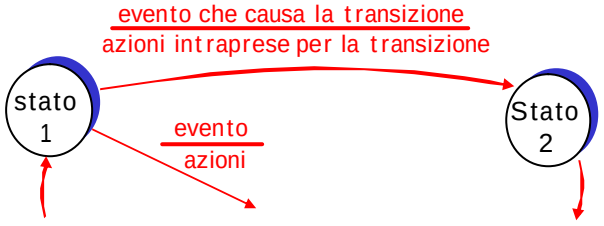
Reliable data transfer (1)



Reliable data transfer: Introduzione

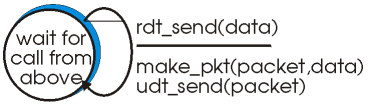
- Sviluppiamo incrementalmente le parti mittente e destinatario di un protocollo rdt
- Consideriamo solo trasferimenti monodirezionali
  - ma le info di controllo vanno in ambedue le direzioni!
- mittente e destinatario come macchine a stati finiti

stato: se siamo in questo "stato" il prossimo stato è univocamente determinato dal prossimo evento

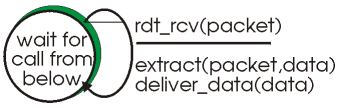


Rdt1.0: trasferim. affidabile su canale affidabile

- Il canale sottostante è perfettamente affidabile
  - Non ci sono errori nei bit
  - Non si perdono pacchetti
- FSMs separate per mittente e destinatario:
  - Il mittente invia dati nel canale sottostante
  - Il destinatario legge dati dal canale sottostante



(a) rdt1.0: sending side

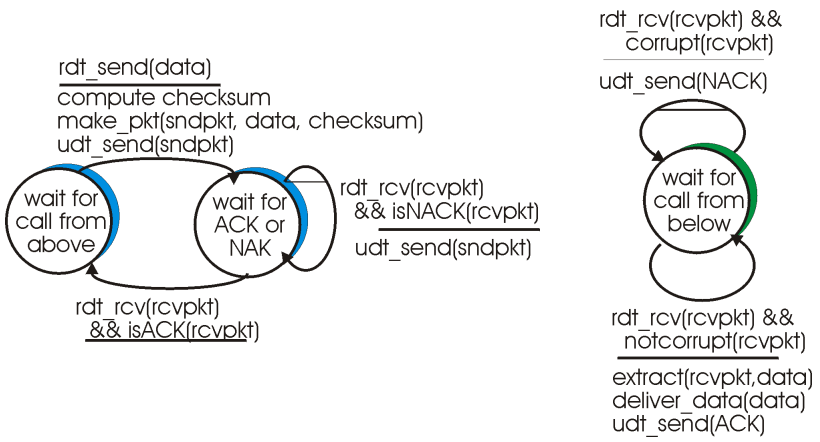


(b) rdt1.0: receiving side

Rdt2.0: canale con errori sui bit

- Il canale può alterare i bits del pacchetto
  - il checksum UDP usato per rilevare gli errori sui bit
- *la* questione: come recuperare gli errori:
  - *acknowledgements (ACKs):* il destinatario dice esplicitamente al mittente che il pkt ricevuto è OK
  - *negative acknowledgements (NAKs):* il destinatario dice esplicitamente al mittente che il pkt ricevuto ha errori
  - Il mittente ritrasmette i pkt se riceve un NAK
  - Un esempio umano che usa ACK e NAK?
- Nuovi meccanismi nel **`rdt2.0`** (rispetto a **`rdt1.0`**):
  - Rilevamento errori
  - Feedback del destinatario: msg di controllo (ACK,NAK) dal dest->mitt

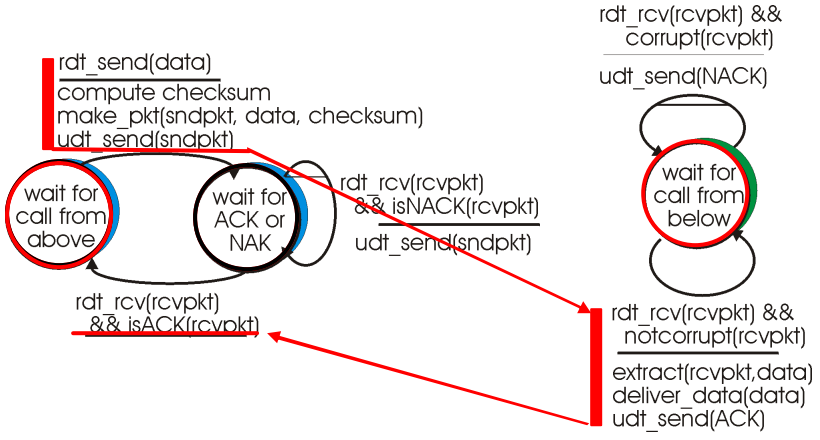
rdt2.0: specifica delle FSM



FSM del mittente

FSM del destinatario

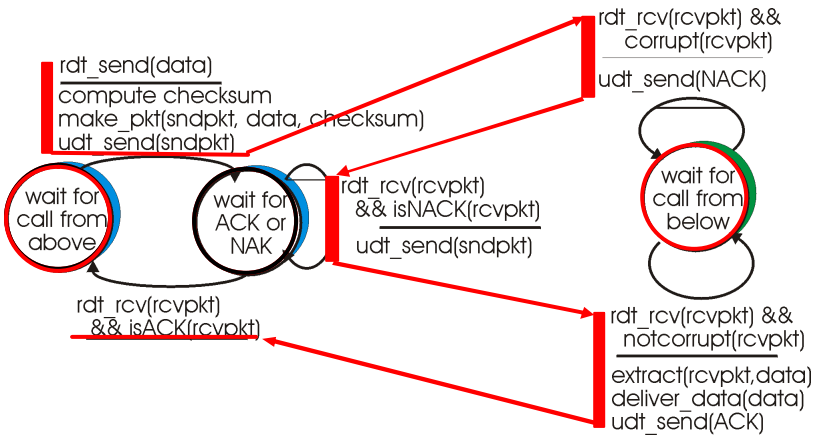
rdt2.0: in azione (nessun errore)



FSM del mittente

FSM del destinatario

rdt2.0: in azione (scenario errori)



FSM del mittente

FSM del destinatario

rdt2.0 ha un buco fatale!

Che succede se si altera un ACK/NAK?

- Il mitt. non sa cosa è accaduto al dest.!
- Non può ritrasmettere e basta: possibili duplicati

Che fare?

- ACK/NAK del mittente e ACK/NAK del dest? Che succede se si perde un ACK/NAK del mittente?
- ritrasmettere, ma si possono ritrasmettere pacchetti ricevuti correttamente!

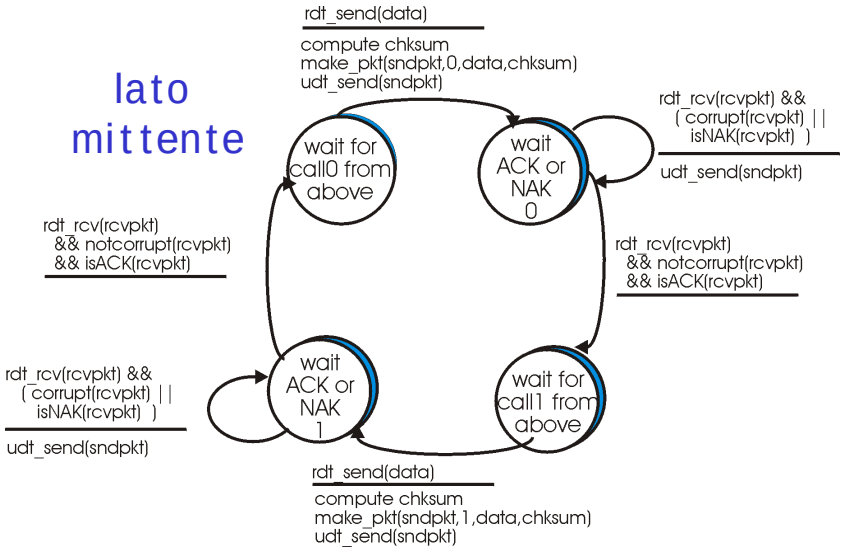
Trattamento duplicati:

- il mitt. aggiunge *sequence number* a ciascun pkt
- il mitt. ritrasmette il pkt corrente se ACK/NAK corrotti
- dest. scarta (non consegna ai livelli superiori) i pkt duplicati

stop and wait
il mittente invia un solo pacchetto e poi attende la risposta del destinatario

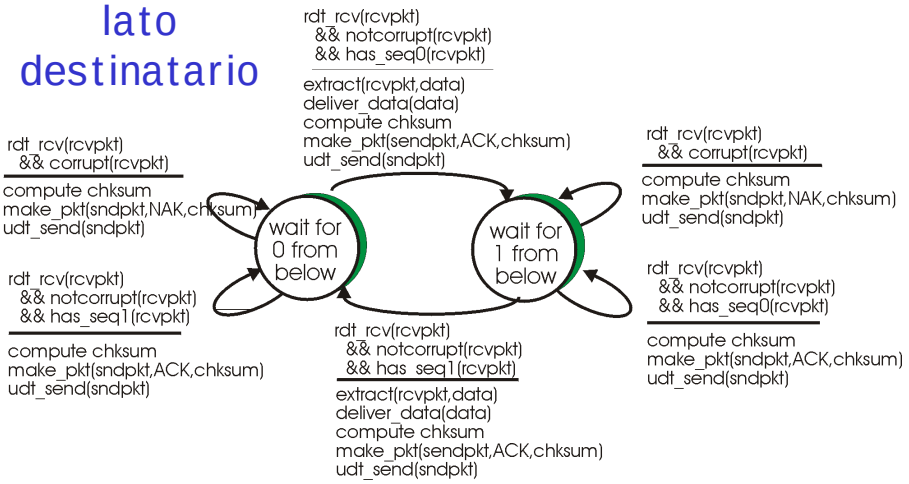
rdt2.1: gestione ACK/NAK corrotti

lato mittente



rdt2.1: gestione ACK/NAK corrotti

lato destinatario



rdt2.1: discussione

Mittente:

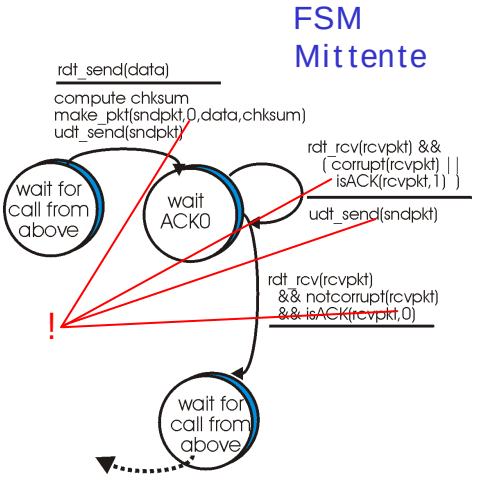
- aggiunta di seq# al pkt
- bastano due seq# (0,1) Perchè?
- deve controllare se vengono ricevuti ACK/NAK corrotti
- Il doppio degli stati
  - lo stato deve "ricordare" se il pkt "corrente" ha seq# 0/1

Destinatario:

- Deve controllare se il pacchetto ricevuto è un duplicato
  - lo stato indica se il seq # del pacchetto atteso deve essere 0 o 1
- nota: il dest. non può sapere se il suo ultimo ACK/NAK è stato ricevuto correttamente dal mittente

rdt2.2: un protocollo NAK-free

- stesse funzionalità di rdt2.1, usando solo ACK
- invece di NAK, il dest. invia ACK per l'ultimo pkt ricevuto OK
  - il dest. deve includere *esplicitamente* il seq # del pkt di cui fa l'ACK
- un ACK duplicato provoca dal mittente la stessa azione di un NAK: *ritrasmetti il pkt*



rdt3.0: canali con errori e perdite

Nuova ipotesi: il canale sottostante può anche smarrire pacchetti (sia dati che ACK)

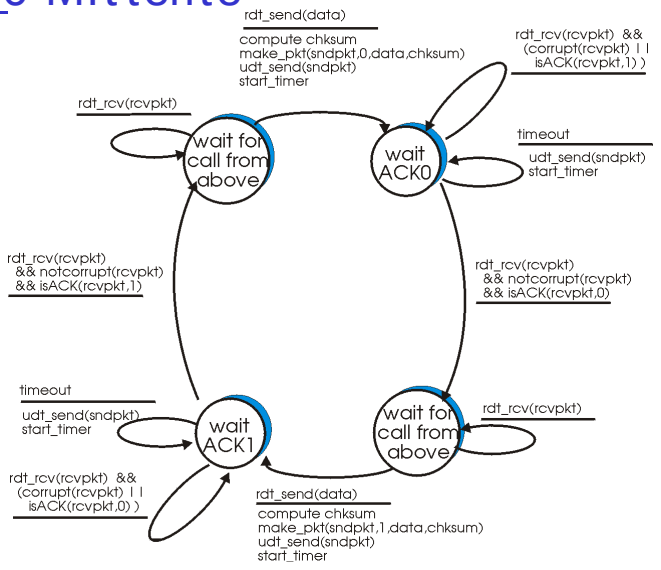
- checksum, seq. #, ACK, ritrasmissioni sono di aiuto ma non bastano

D: come gestire gli smarrimenti?

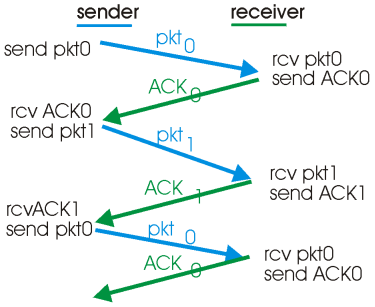
Un approccio: il mittente attende un "ragionevole" ammontare di tempo per un ACK

- Ritrasmette se non riceve ACK entro questo tempo
- se il pkt (o l'ACK) era solo in ritardo (non smarrito):
  - la ritrasmissione genera un duplicato ma i seq. # sono già in grado di gestirlo
  - il dest. Deve specificare il seq # del pkt di cui fa ACK
- Ci vuole un countdown timer

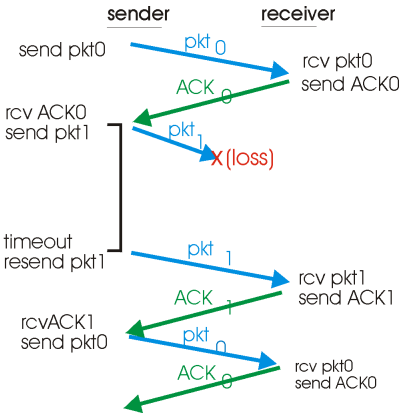
rdt3.0 Mittente



rdt3.0 in azione

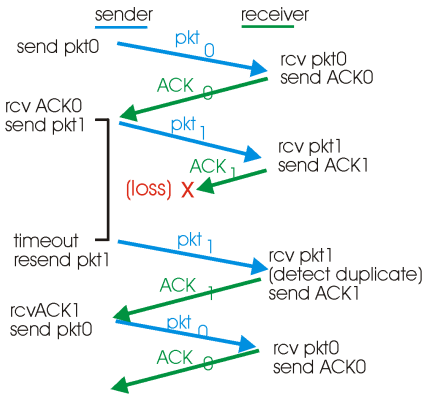


(a) operation with no loss

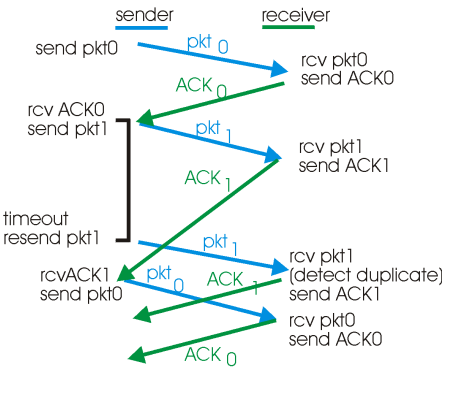


(b) lost packet

rdt3.0 in azione



(c) lost ACK



(d) premature timeout

Performance del rdt3.0

- rdt3.0 funziona, ma le prestazioni fanno schifo!
- esempio: link da 1 Gbps, ritardo prop. 15 ms e-e, pacchetto 1KB :

$$T_{Trasm.} = \frac{8kb/pkt}{10**9\ b/sec} = 8\ microsec$$

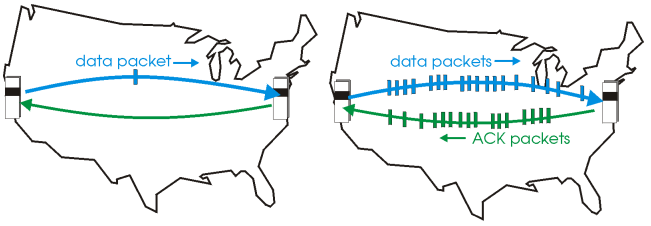
$$\text{Utilizzazione} = U = \frac{\text{frazione del tempo mitt. impegn. spedire}}{} = \frac{8\ microsec}{30.016\ msec} = 0.00015$$

  - 1KB pkt ogni 30 msec -> 33kB/sec throughput su un link 1 Gbps
  - Il protocollo limita l'uso delle risorse fisiche (teoricamente potrei andare a 128MB/sec!!!).

Protocolli Pipelined

Pipelining: il mittente accetta l'esistenza di molti pacchetti "in viaggio", non ancora riscontrati

  - L'intervallo di numeri di sequenza si deve incrementare
  - Bufferizzazione al mittente e/o al destinatario



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

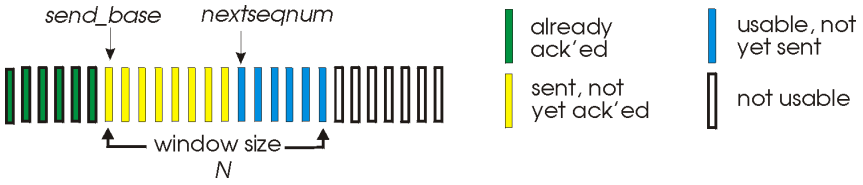
- Due generiche forme di protocolli pipelined:

go-Back-N, selective repeat

Go-Back-N

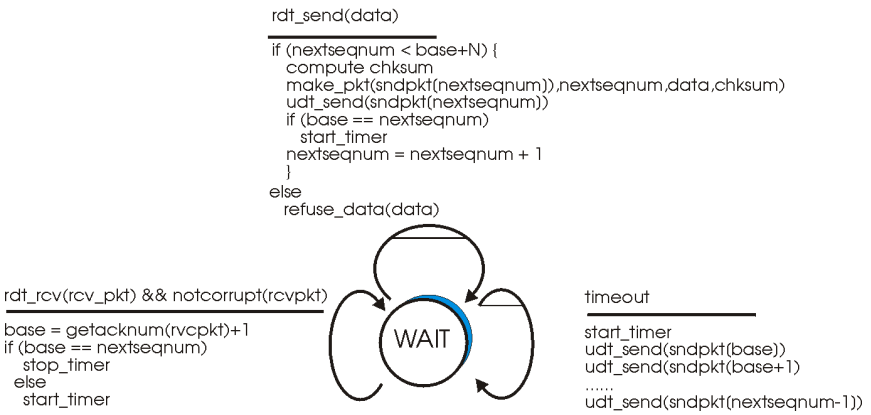
Mittente:

- k-bit seq # nell'header del pkt
- "finestra" consentita di (al massimo) N pacchetti consecutivi non riscontrati

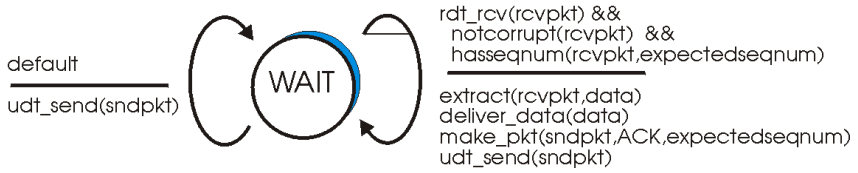


- ACK(n): riscontra tutti i pkts fino a seq # n - "ACK cumulativo"
  - può nascondere ACK duplicati (vedi destinatario)
- timer per ogni pkt "in volo"
- *timeout(n):* ritrasmetti il pkt n e tutti i pacchetti con seq # maggiori nella finestra

GBN: FSM estesa del mittente



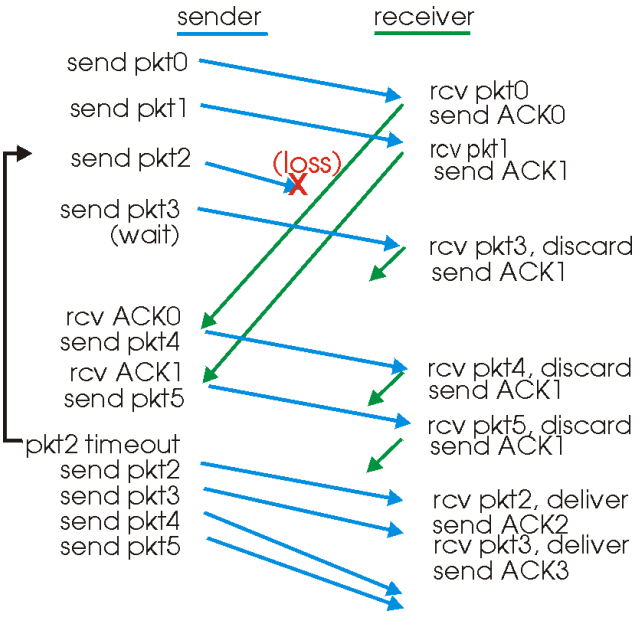
GBN: FSM estesa del destinatario



Destinatario semplice:

- ACK-only: invia sempre un ACK per il pkt correttamente ricevuto col più alto seq # *in-ordine*
  - può generare ACK duplicati
  - deve solo ricordare **`expectedseqnum`**
- Pkt fuori ordine :
  - scarta (non bufferizza) -> non c'è buffer al destinatario!
  - dai un ACK per il pkt con il più alto seq # in ordine

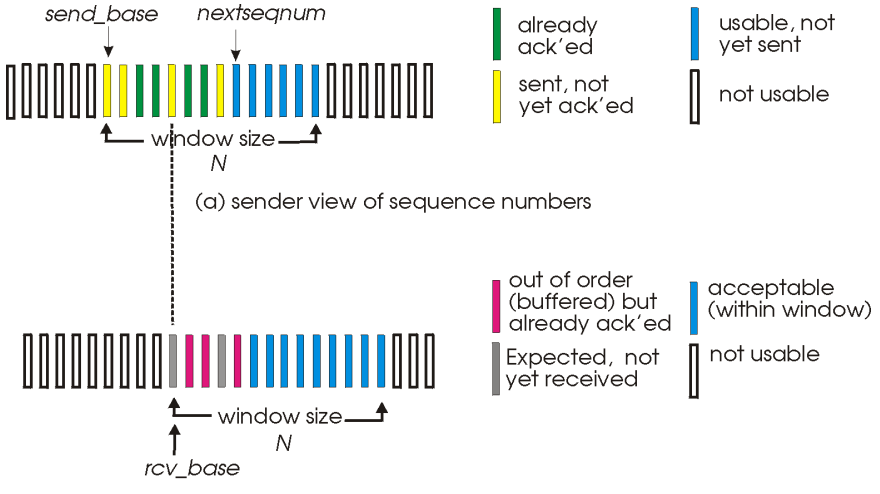
GBN in azione



Selective Repeat

- destinatario riscontra *individualmente tutti* i pkt correttamente ricevuti
  - Bufferizza i pkt, come necessario, per poterli alla fine consegnare in ordine al livello superiore
- mittente re-invia solo i pkt per i quali non riceve ACK
  - timer del mittente per ciascun pkt non riscontrato
- Finestra del mittente
  - N seq # consecutivi
  - Limita di nuovo i seq # dei pkt inviati e non riscontrati

Selective repeat: finestre mittente/destin.



Selective repeat

sender

Dati dall'alto:

- Se c'è un seq # disponibile nella finestra, invia pkt

timeout(n):

- reinvia pkt n, riavvia timer

ACK(n) in [sendbase,sendbase+N]:

- segna pkt n come ricevuto
- if n è il più piccolo unACKed pkt, avanza window base al prossimo unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- invia ACK(n)
- Fuori ordine: buffer
- In ordine: consegna, avanza la window al prossimo pkt non ancora ricevuto

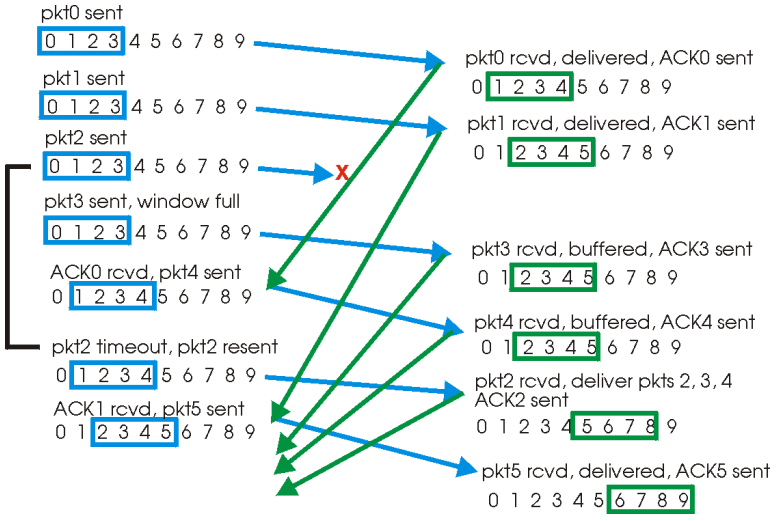
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

altrimenti:

- ignora

Selective repeat in azione



Selective repeat: un dilemma

Esempio:

- seq #: 0, 1, 2, 3
- window size=3

- Il destinatario non distingue le due situazioni!
- in (a) sbaglia e considera come nuovi pkt i duplicati

Q: che relazione tra la dimensione dei seq # e quella della window?