

Il Livello Trasporto

Obiettivi:

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

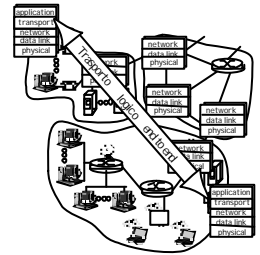
Panoramica:

- 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

LivelloTrasporto 3a-1

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:
- *Livellonetwork*: trasferimento dati tra end systems
- *Livellotrasporto*: trasferimento dati tra processi
 - Si appoggia su, e migliora, i servizi di livello network

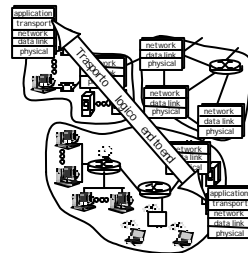


LivelloTrasporto 3a-2

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



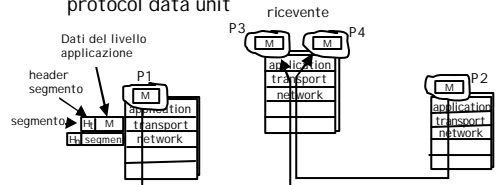
LivelloTrasporto 3a-3

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



LivelloTrasporto 3a-4

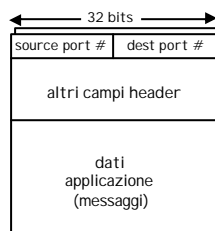
Multiplexing/demultiplexing

Multiplexing:

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

demultiplexing:

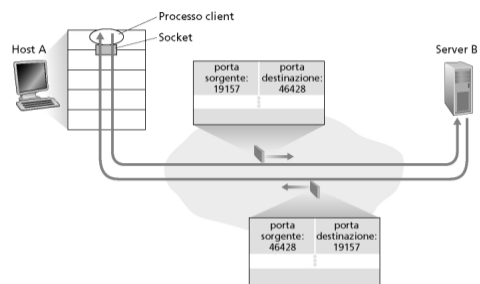
- Basati sui numeri di porta e gli indirizzi IP del mittente e del destinatario
 - porte di source, dest in ogni segmento
 - Porte well-known per applicazioni specifiche



Formato del segmento TCP/UDP

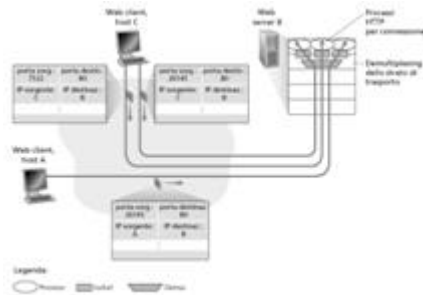
LivelloTrasporto 3a-5

Multiplexing/demultiplexing: esempi



LivelloTrasporto 3a-6

Multiplexing/demultiplexing: esempi



Livello Trasporto 3a-7

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

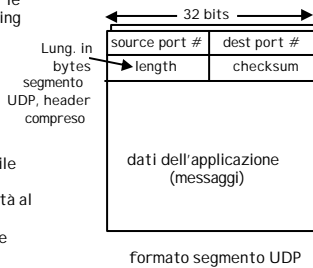
Perchè esiste UDP?

- Non occorre 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

LivelloTrasporto 3a-8

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!



LivelloTrasporto 3a-9

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

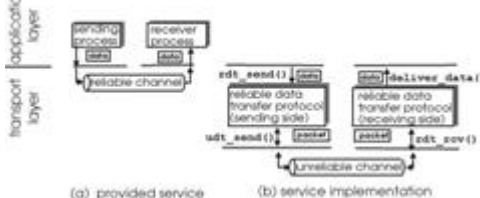
Destinatario:

- 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 ...)

LivelloTrasporto3a-10

Principi di trasferimento affidabile

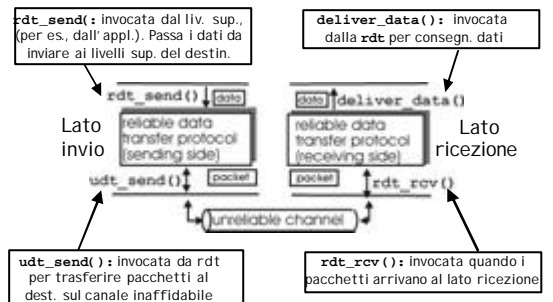
- È importante per i livelli applicazione, trasporto, data link
- È nella top-10 delle problematiche di rete importanti!



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

LivelloTrasporto3a-11

Reliable data transfer (1)



LivelloTrasporto3a-12

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



LivelloTrasporto3a-13

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



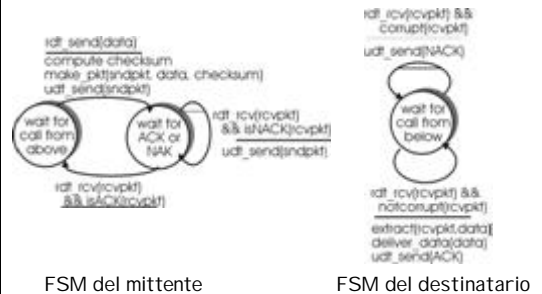
LivelloTrasporto3a-14

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

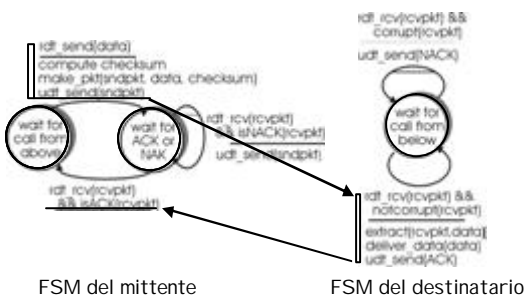
LivelloTrasporto3a-15

rdt2.0: specifica delle FSM



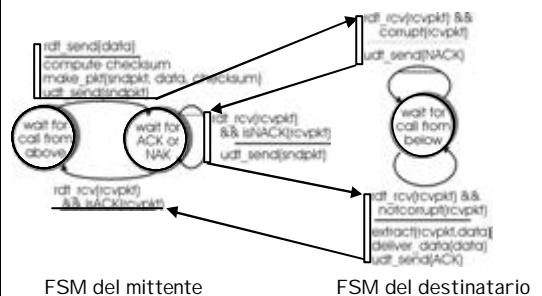
LivelloTrasporto3a-16

rdt2.0: in azione (nessun errore)



LivelloTrasporto3a-17

rdt2.0: in azione (scenario errori)



LivelloTrasporto3a-18

rdt2.0

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 succedesse si perde un ACK/NAK del mittente?
- ritrasmettere, ma si possono ritrasmettere pacchetti ricevuti correttamente

Trattamento duplicati :

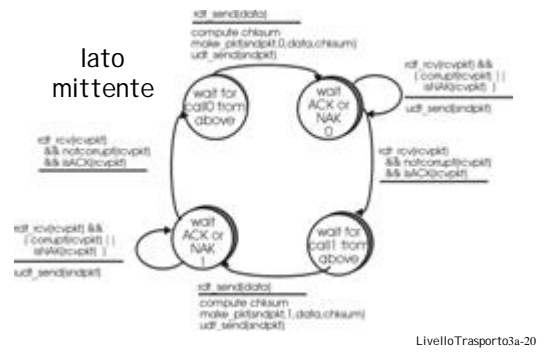
- il mitt. aggiunge *sequence number* a ciascun pkt
- il mitt. ritrasmette il pkt corrente se ACK/NAK corrotti
- dest. scarta (non consegna ai livelli superiori) i pkt duplicati

stop and wait

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

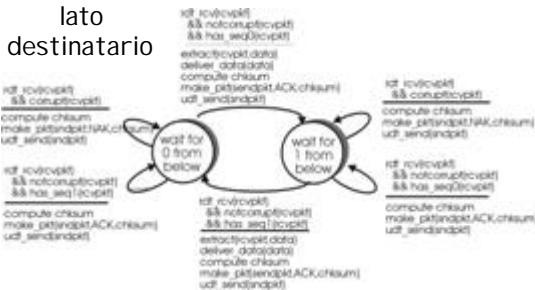
LivelloTrasporto3a-19

rdt2.1: gestione ACK/NAK corrotti



rdt2.1: gestione ACK/NAK corrotti

lato destinatario



rdt2.1: discussione

Mittente:

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

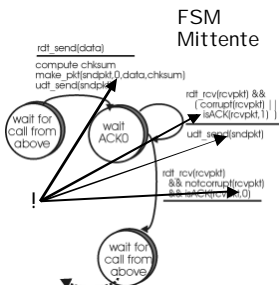
Destinatario:

- 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

LivelloTrasporto3a-22

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

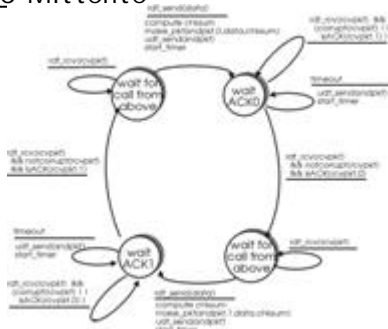
D: come gestire gli smarrimenti?

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

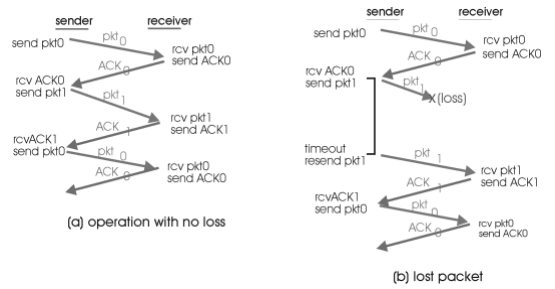
LivelloTrasporto3a-24

rdt3.0 Mittente



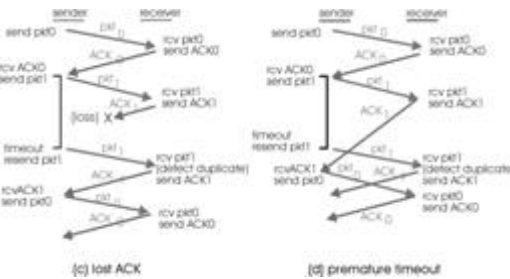
LivelloTrasporto3a-25

rdt3.0 in azione



LivelloTrasporto3a-26

rdt3.0 in azione



LivelloTrasporto3a-27

Performance del rdt3.0

- rdt3.0 funziona, ma le prestazioni sono molto basse!
- esempio: link da 1 Gbps, ritardo prop. 15 ms e-e, pacchetto 8KB :

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

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

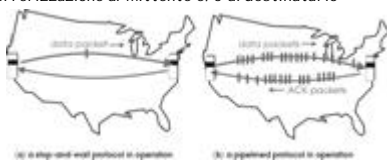
- 1KB pkt ogni 30 msec -> 33kB/sec throughput su un link 1 Gbps
- Il protocollo limita l'uso delle risorse fisiche (teoricamente potrei andare a 128MB/sec!!!).



LivelloTrasporto3a-28

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



- Due generiche forme di protocolli pipelined: *go-Back-N*, *selective repeat*

LivelloTrasporto3a-29

Go-Back-N

Mittente:

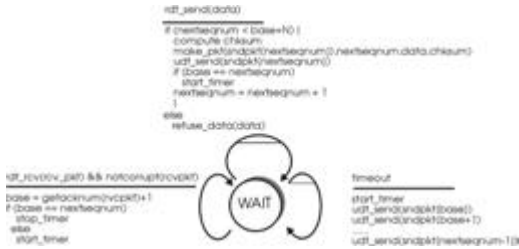
- k-bit seq # nell'header del pkt
- "finestra" consentita di (al massimo) N pacchetti consecutivi non riscontrati



- ACK(n): riscontra tutti i pkts fino a seq # n - "ACK cumulativo"
 - può nascondere ACK duplicati (vedi destinatario)
- Un solo timer
- timeout(n): ritrasmette il pkt n e tutti i pacchetti con seq # maggiori nella finestra

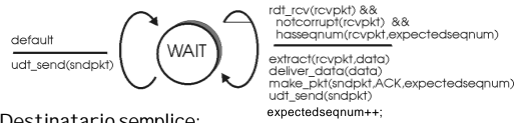
LivelloTrasporto3a-30

GBN: FSM estesa del mittente



LivelloTrasporto3a-31

GBN: FSM estesa del destinatario

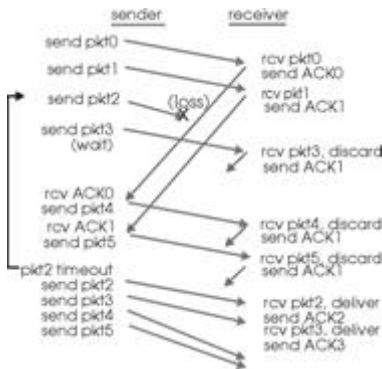


Destinatario semplice:

- ACK-only: invia sempre un ACK per il pkt correttamente ricevuto col pi u 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

LivelloTrasporto3a-32

GBN in azione



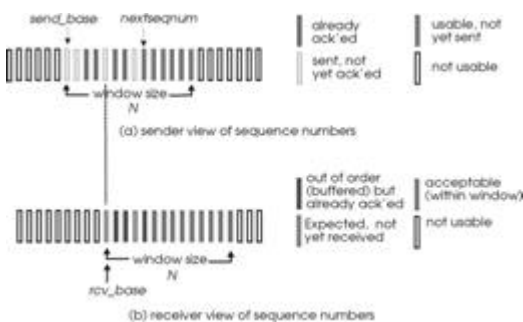
LivelloTrasporto3a-33

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

LivelloTrasporto3a-34

Selective repeat: finestre mittente/destin.



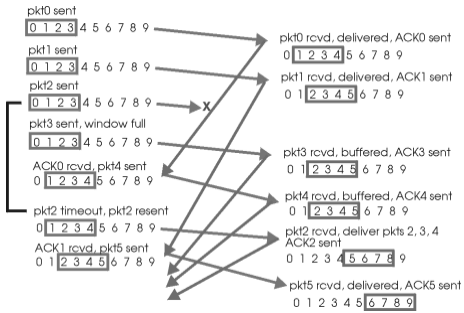
LivelloTrasporto3a-35

Selective repeat

- | Sender | Receiver |
|---|---|
| Dati dall'alto: <ul style="list-style-type: none"> □ Se c'è un seq # disponibile nella finestra, invia pkt timeout(n): <ul style="list-style-type: none"> □ reinvia pkt n, riavvia timer ACK(n) in [sendbase, sendbase+N]: <ul style="list-style-type: none"> □ segna pkt n come ricevuto □ if n è il più piccolo unACKed pkt, avanza window base al prossimo unACKed seq # | pkt n in [rcvbase, rcvbase+N-1] <ul style="list-style-type: none"> □ 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] <ul style="list-style-type: none"> □ ACK(n) altrimenti: <ul style="list-style-type: none"> □ ignora |

LivelloTrasporto3a-36

Selective repeat in azione



LivelloTrasporto3a-37

Selective repeat: un dilemma

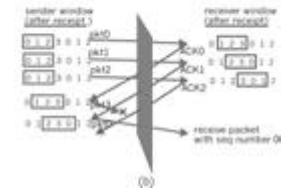
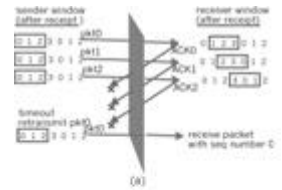
Esempio:

- 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?

R: $Win_size \leq MaxSeq_range \# / 2$



LivelloTrasporto3a-38