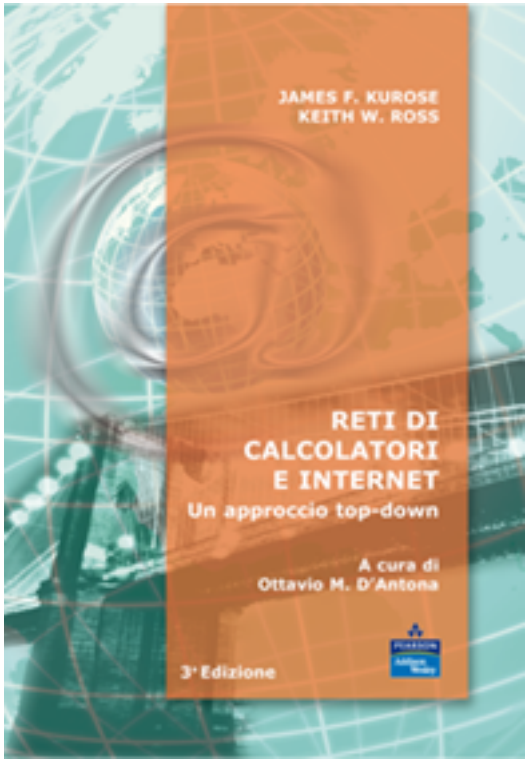
Reti di calcolatori: Livello Trasporto

(Capitolo 3 Kurose-Ross)

Marco Roccetti

20 Marzo 2023

(Capitolo 3 Kurose-Ross)



Reti di calcolatori e Internet: Un approccio top-down

3ª edizione
Jim Kurose, Keith Ross
Pearson Education Italia
©2005

Il livello Trasporto

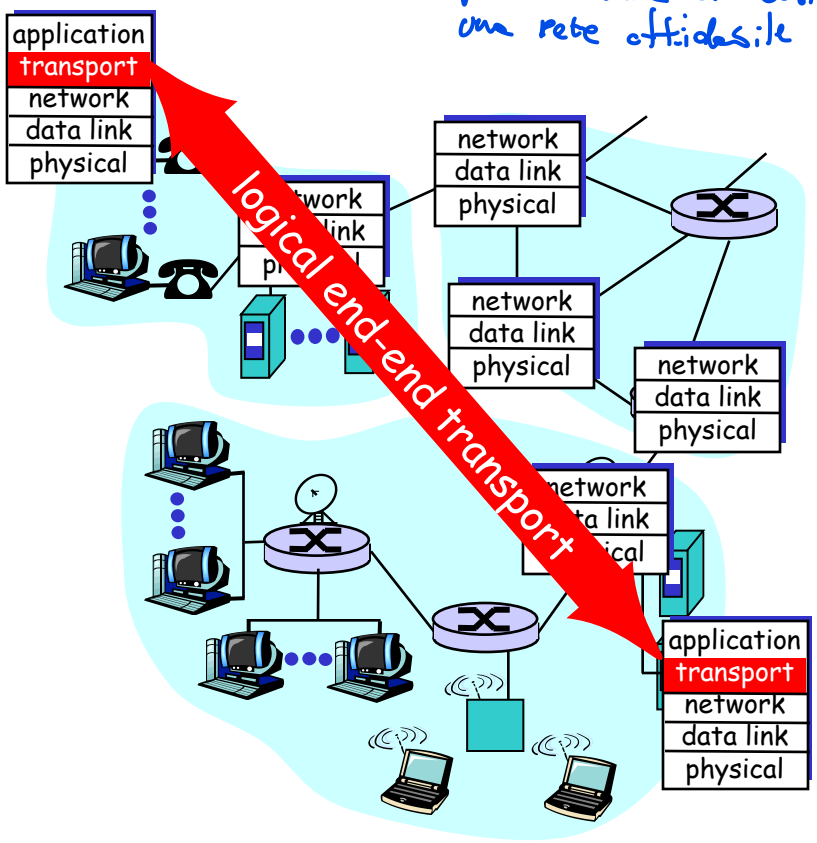
Livello 4 (trasporto): cosa vedremo

- Servizi di livello trasporto
- multiplexing/demultiplexing
- Protocollo non orientato alla connessione: UDP
- Principi di trasferimento end-to-end affidabile
- Protocollo orientato alla connessione: TCP
  - Trasferimento affidabile end-to-end
  - Controllo di flusso e controllo della congestione
  - Gestione della connessione
- Principi di controllo della congestione
- TCP

Servizi e protocolli di livello trasporto

TCP → Aff:dob,le
Es:ste un'unica strada per arrivare a costruire una rete affidabile

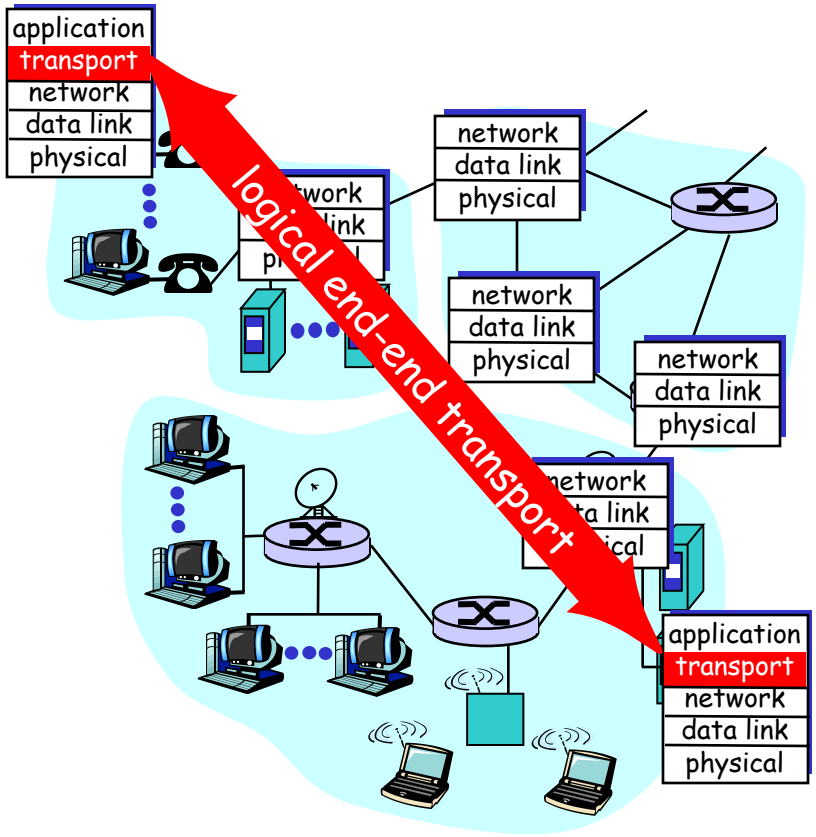
- fornire *comunicazione logica* tra processi in esecuzione su host remoti
- I protocolli di livello trasporto sono eseguiti solo dagli host agli estremi del cammino
- Servizi di livello trasporto vs. servizi di livello rete:
- *Livello rete:* trasferimento dati da host a host
- *Livello trasporto:* trasferimento dati tra processi agli estremi (terminali, processi)
  - basato su livello rete



Protocolli di livello trasporto

Servizi di livello trasporto di Internet:

- Consegna ordinata e affidabile (TCP)
  - Controllo di congestione
  - Controllo di flusso
  - Setup della connessione
- Consegna inaffidabile e disordinata ("best-effort"): UDP
- Servizi non realizzati:
  - real-time
  - Qualità del servizio garantita
  - Multicast affidabile



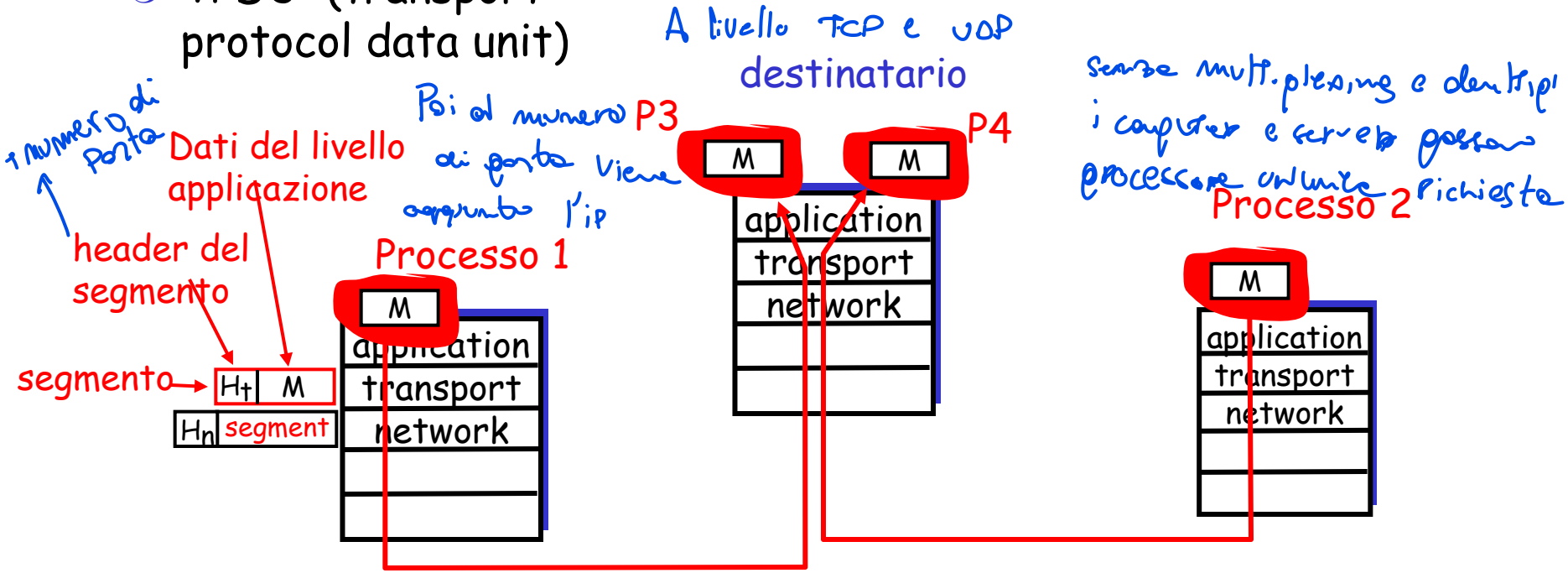
Multiplexing/demultiplexing

Rimane garantito che da un medesimo host io posso far girare più istanze dello stesso host

N.B.: *segmento* - unità di dati scambiata tra entità di livello trasporto

- TPDU: (transport protocol data unit)

Demultiplexing: la fase di consegna dei dati ricevuti all'applicazione superiore

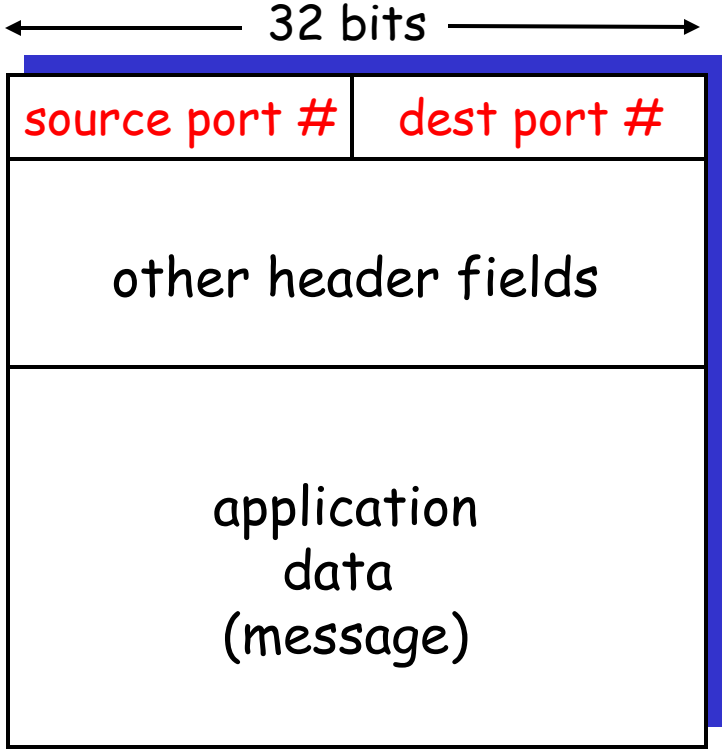


Multiplexing/demultiplexing

Multiplexing:
I dati ricevuti dalle applicazioni vengono incapsulati in segmenti con le informazioni che serviranno al demultiplexing

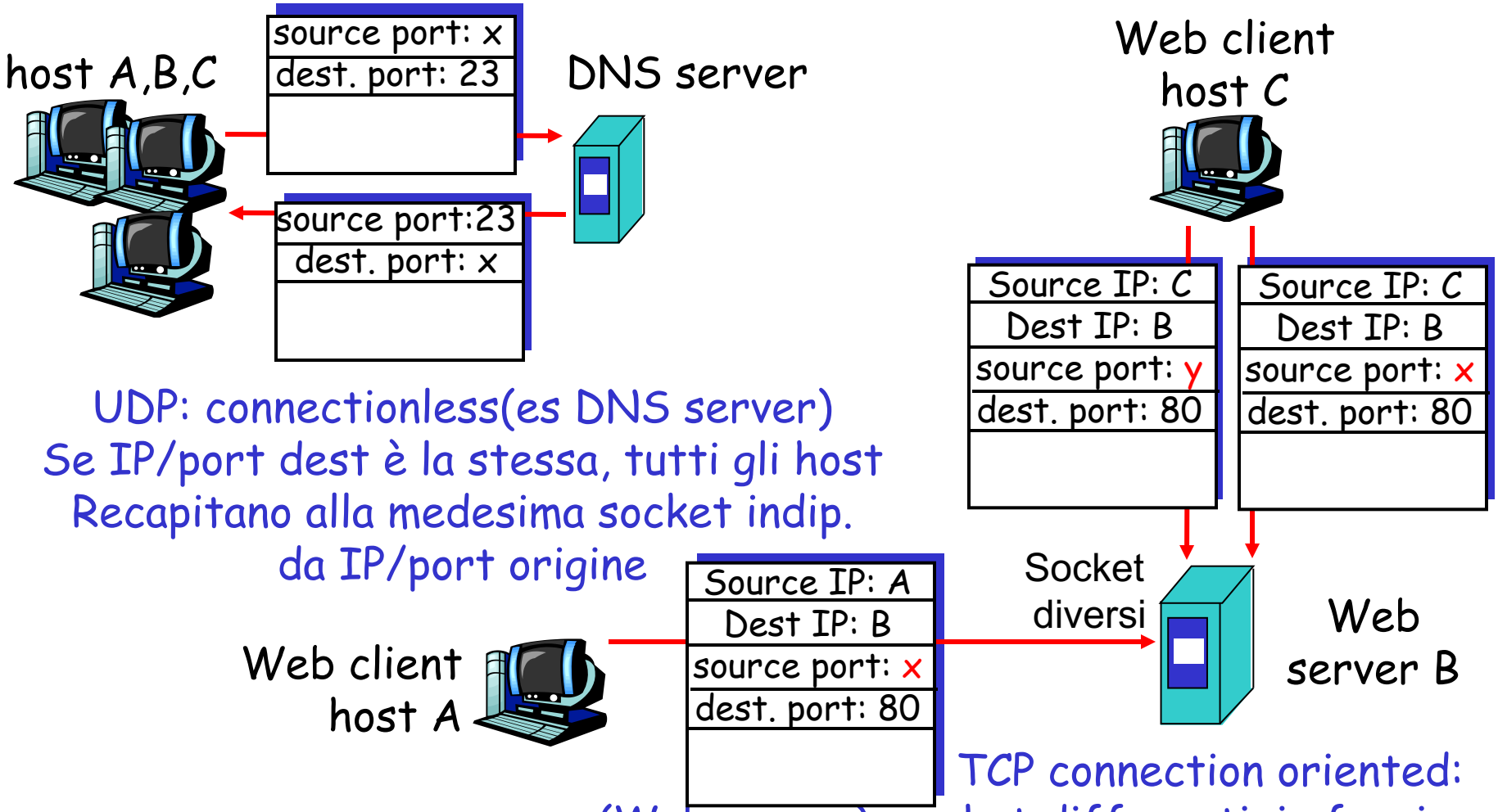
multiplexing/demultiplexing:

- Si basa sui Numeri di Porta e indirizzi IP del mittente e destinatario finale
  - Inserite in ogni segmento
  - N.B. well-known port numbers sono usati per applicazioni note



TCP/UDP segment format

Multiplexing/demultiplexing: differenze UDP/TCP socket



UDP: connectionless(es DNS server)
Se IP/port dest è la stessa, tutti gli host
Recapitano alla medesima socket indip.
da IP/port origine

TCP connection oriented:
(Web server), socket differenti, in funzione di
4 parametri: Ip/port di origine e di dest

UDP: User Datagram Protocol [RFC 768]

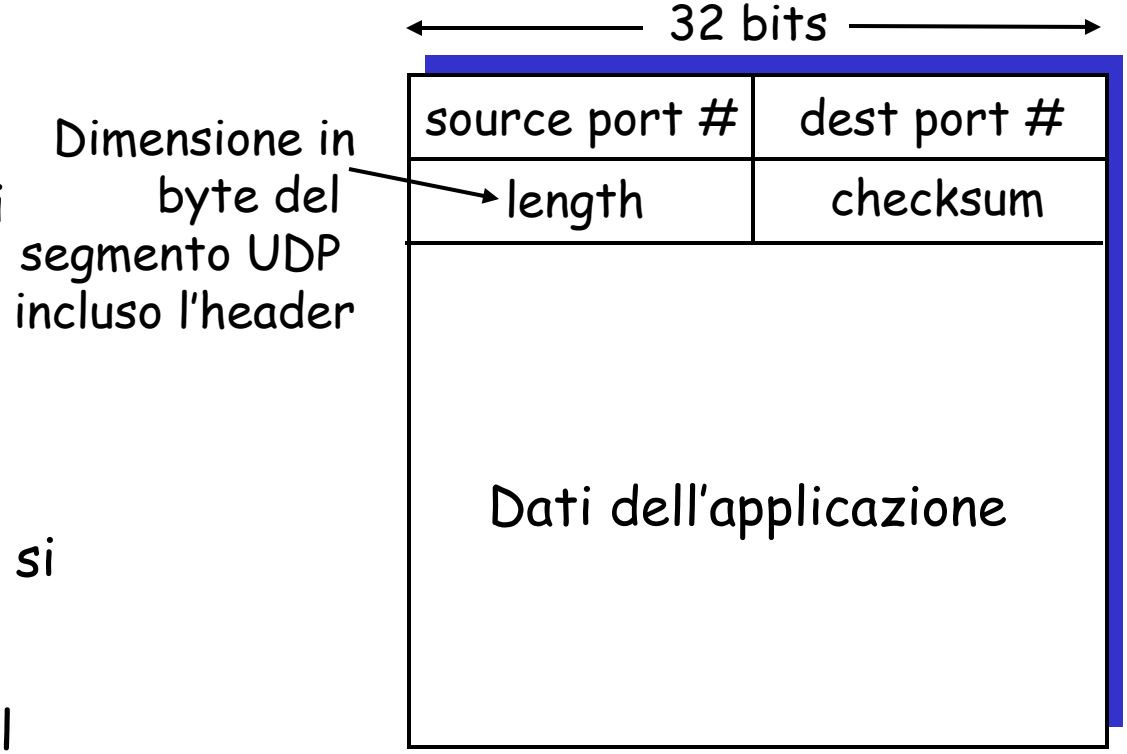
- Semplice e pratico
- Servizio "best effort", i segmenti UPD possono
  - perdersi
  - arrivare disordinati
- *Servizio connectionless:*
  - Non c' instaurazione preliminare della connessione tra sender e receiver
  - Ogni segmento UDP è gestito in modo indipendente (non c'è stato della connessione)

Perchè è utile UDP?

- Non c'è instaurazione preliminare di connessione (trasferimento immediato e più veloce dei dati)
- Semplicità: non serve mantenere lo stato della connessione
- header del segmento ridotto (8 B)
- Non c'è controllo della congestione: UDP può essere spinto alla massima velocità (non sempre...)

UDP e suo utilizzo

- Si usa spesso per lo streaming multimediale
  - Tollera perdita di pacchetti
  - Sensibile al ritmo di invio dei pacchetti
- UDP usato anche in
  - DNS
  - SNMP
- R-UDP: UDP affidabile, si ottiene garantendo controlli a livello di ricezione al di sopra del livello trasporto
  - Soluzioni di gestione specifiche del livello applicativo



Formato del segmento UDP

UDP checksum

obiettivo: rilevare errori sui bit del segmento

Sender:

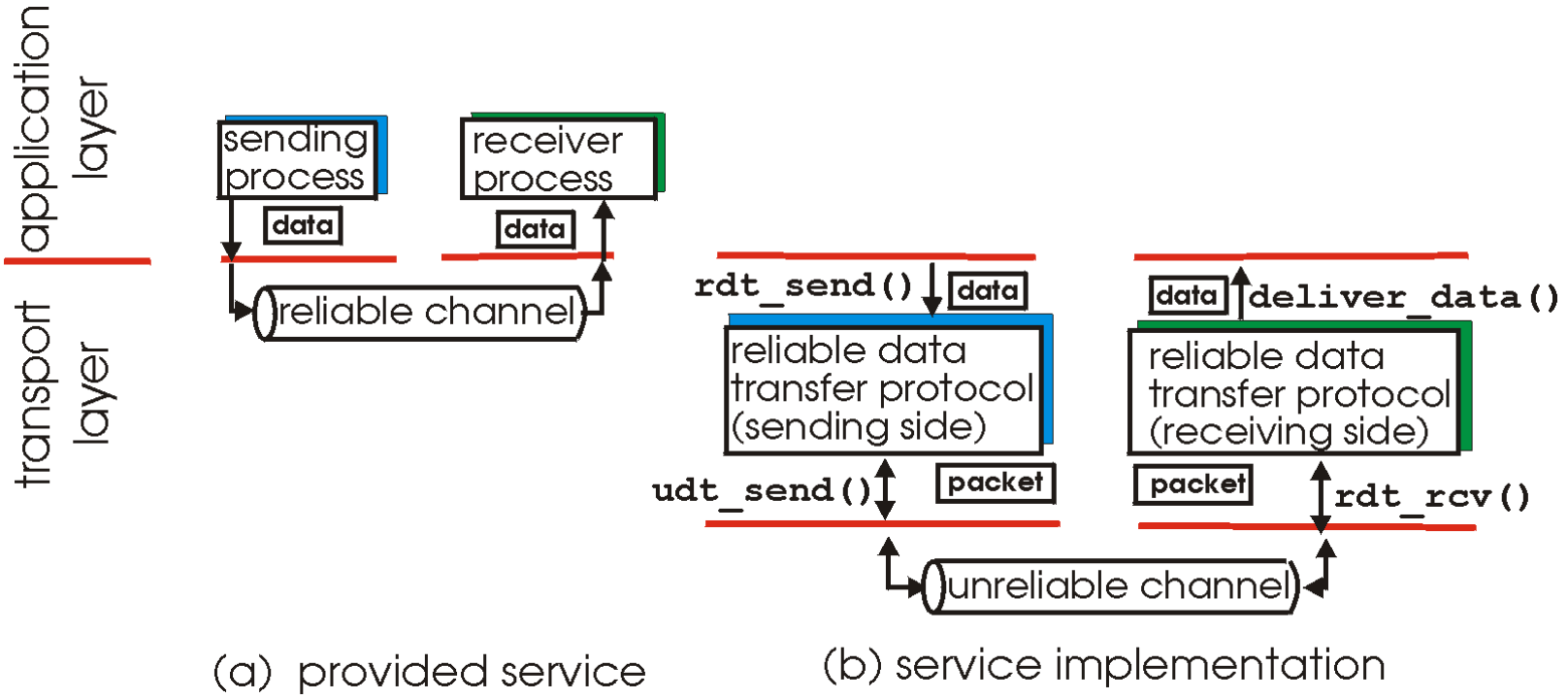
- Considera i dati come interi a 16 bit
- checksum: somma gli interi (con eventuale riporto addizionato al primo bit) e fa il complemento a 1
- Inserisce il valore della somma nel campo checksum del segmento

Receiver:

- Calcola la somma di controllo dei dati ricevuti
- Verifica la presenza di errori
  - Non ci sono errori: passa i dati all'applicazione
  - Ci sono errori: elimina i dati

Principi di trasferimento dati affidabile

- Vogliamo affidabilità al livello applicativo attraverso servizi del livello rete inaffidabili: come si fa?

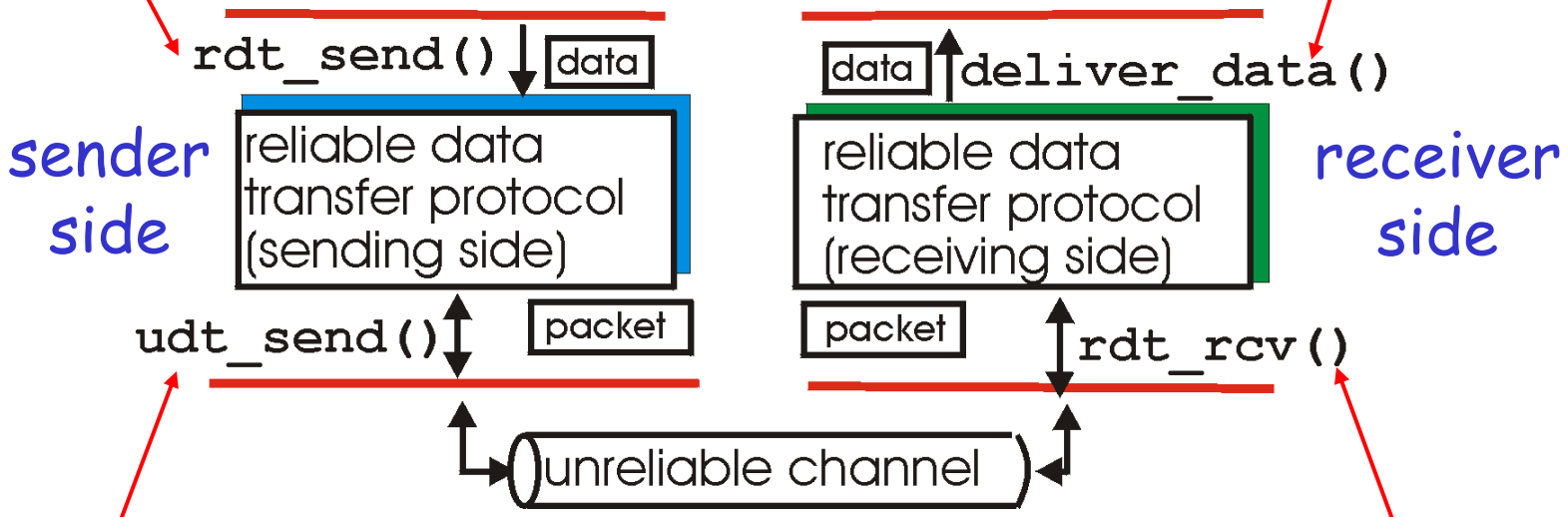


- Le caratteristiche del canale di comunicazione (e dei servizi di rete forniti) determinano il grado di complessità del livello trasporto per determinare un livello affidabile.

Livello trasporto affidabile: come si fa?

rdt_send(): chiamata da sopra (applicazione), passa i dati da inoltrare all'applicazione ricevente

deliver_data(): chiamata dal livello trasporto per consegnare i dati ricevuti all'applicazione sopra



udt_send(): chiamata del servizio inaffidabile di spedizione dei dati sulla rete

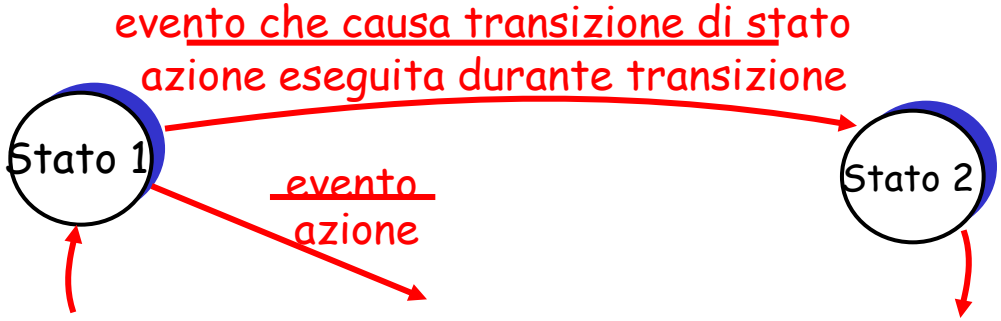
rdt_rcv(): chiamata dal livello rete quando ci sono dati ricevuti per il livello trasporto

Livello trasporto affidabile: come si fa?

Assunzioni:

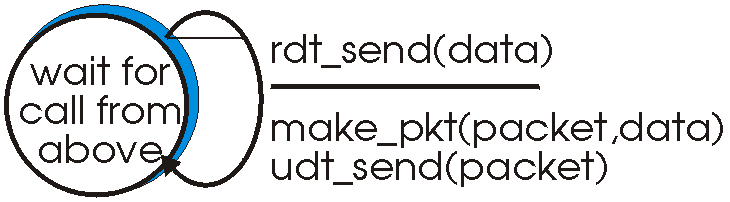
- Partiamo da esempi semplici e poi via via arriviamo al caso reale
- Consideriamo solo trasferimento in una direzione
  - Anche se le info viaggiano nei due sensi
- Usiamo la notazione delle macchine a stati finiti per descrivere i protocolli

stato: dato uno stato abbiamo tante transizioni verso altri stati quanti sono i possibili eventi



Rdt1.0: trasferimento affidabile su rete affidabile

- È un caso quasi banale: vogliamo creare un livello trasporto affidabile (RDT) avendo già un livello rete affidabile
  - La rete non crea bit errati e non si perdono pacchetti!!!
- Vediamo come funzionano il sender e il receiver:
  - Sender spedisce dati sulla rete
  - Receiver riceve dati dalla rete



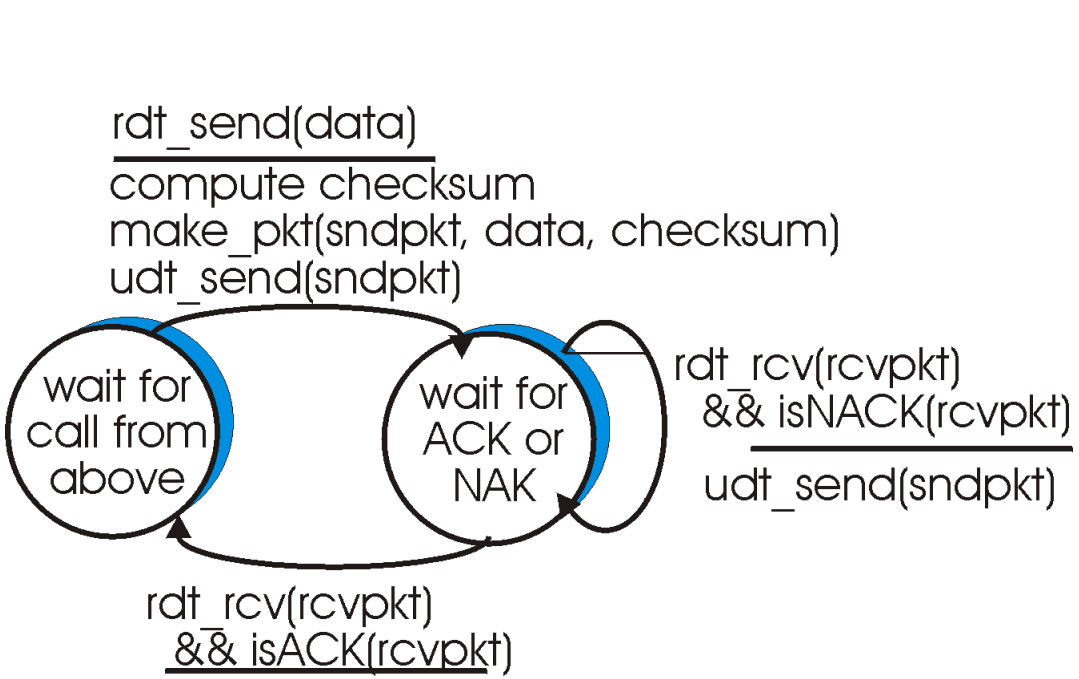
(a) rdt1.0: sending side

(b) rdt1.0: receiving side

Rdt2.0: rete con bit errati, senza perdita

- Assunzione: la rete altera i bit ma non perde pacchetti
  - N.B.: UDP checksum rileva i bit errati
- *Come si recuperano le situazioni con bit errati?*
  - *acknowledgements (ACKs):* il receiver manda un pacchetto ACK al sender per dire che i dati sono OK
  - *negative acknowledgements (NAKs o NACKs):* il receiver manda un pacchetto NAK al sender per dire che i dati sono errati
  - sender ritrasmette i dati se riceve NAK
- Nuovi meccanismi in `rdt2.0` (aggiunti a `rdt1.0`):
  - Rilevazione di errori sui bit (es. checksum)
  - Messaggi di controllo da parte del receiver (ACK,NAK)

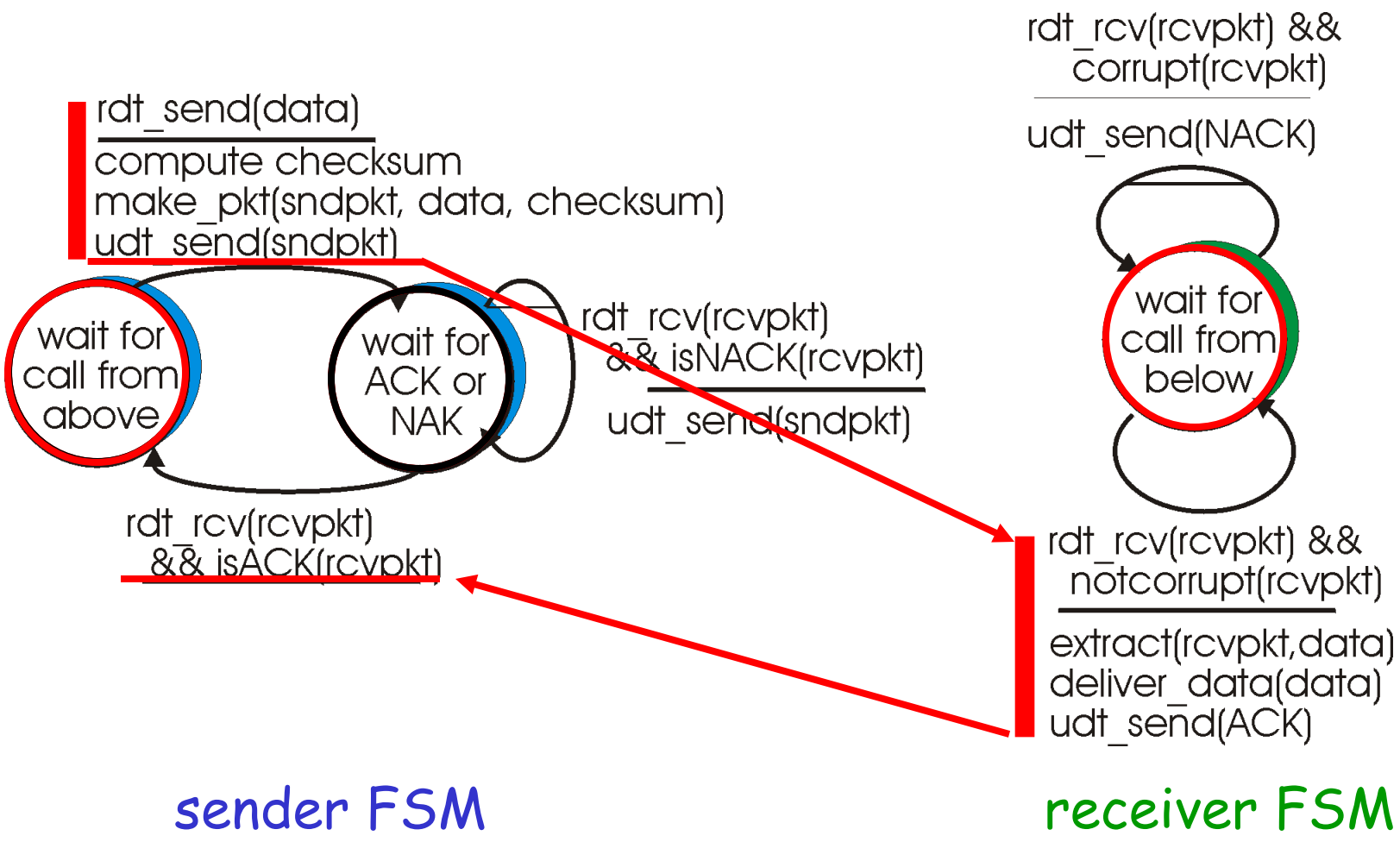
rdt2.0: definizione protocollo



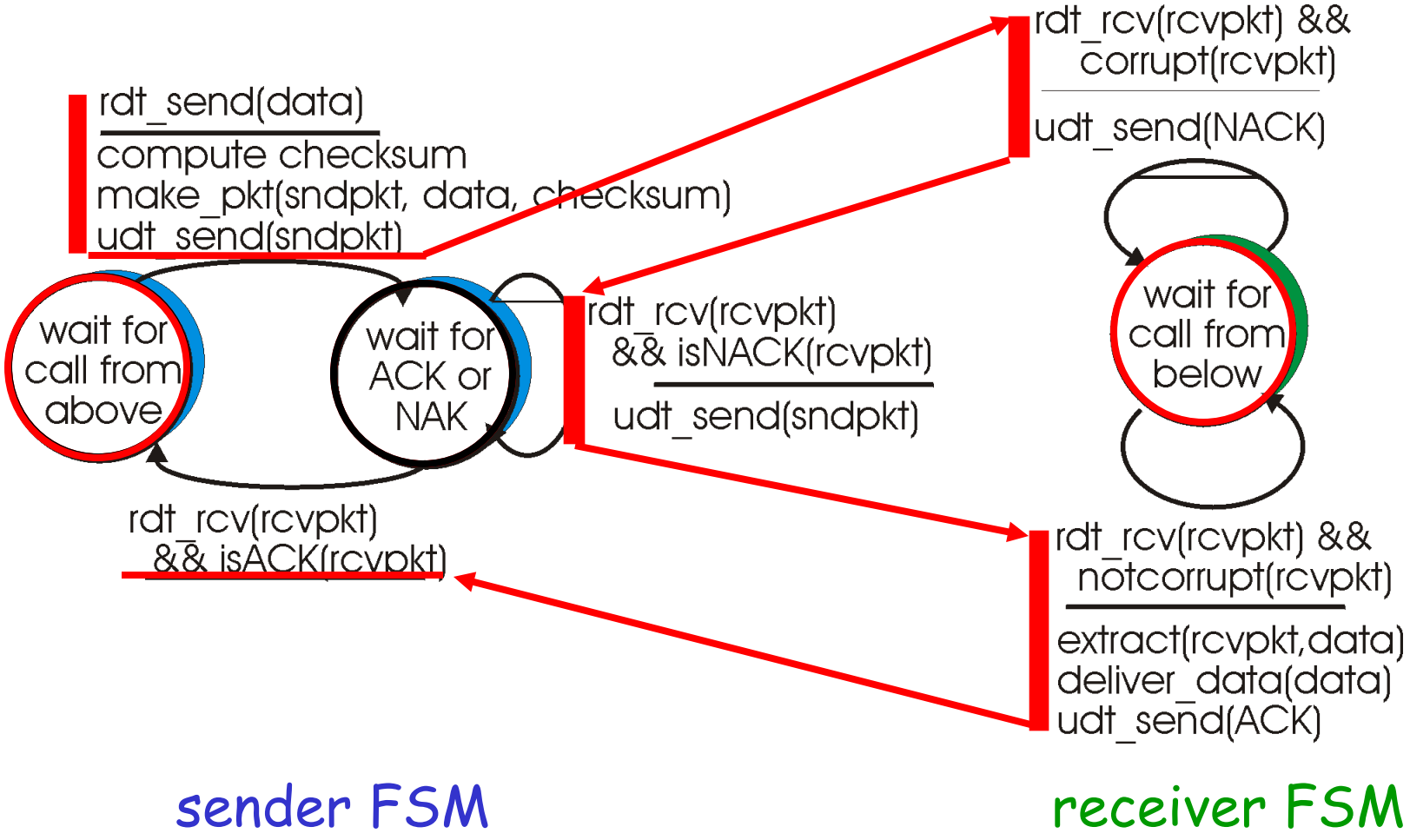
Macchina a stati finiti del sender STOP and WAIT

Macchina a stati finiti del receiver

rdt2.0: caso1: spedizione senza errore



rdt2.0: caso 2: spedizione con errore e ritrasmissione



rdt2.0 ha un problema grave!

Cosa succede se ci sono bit errati su ACK e NAK?

- Il sender non capisce cosa è accaduto sul receiver
- In questo caso potrebbe decidere di ritrasmettere cmq pkt -> si originano duplicati del segmento

Come si fa?

- Inviare ACK di ACK?
  - No. Stesso problema
- Il sender ritrasmette!
  - Ma possono nascere duplicati del segmento!

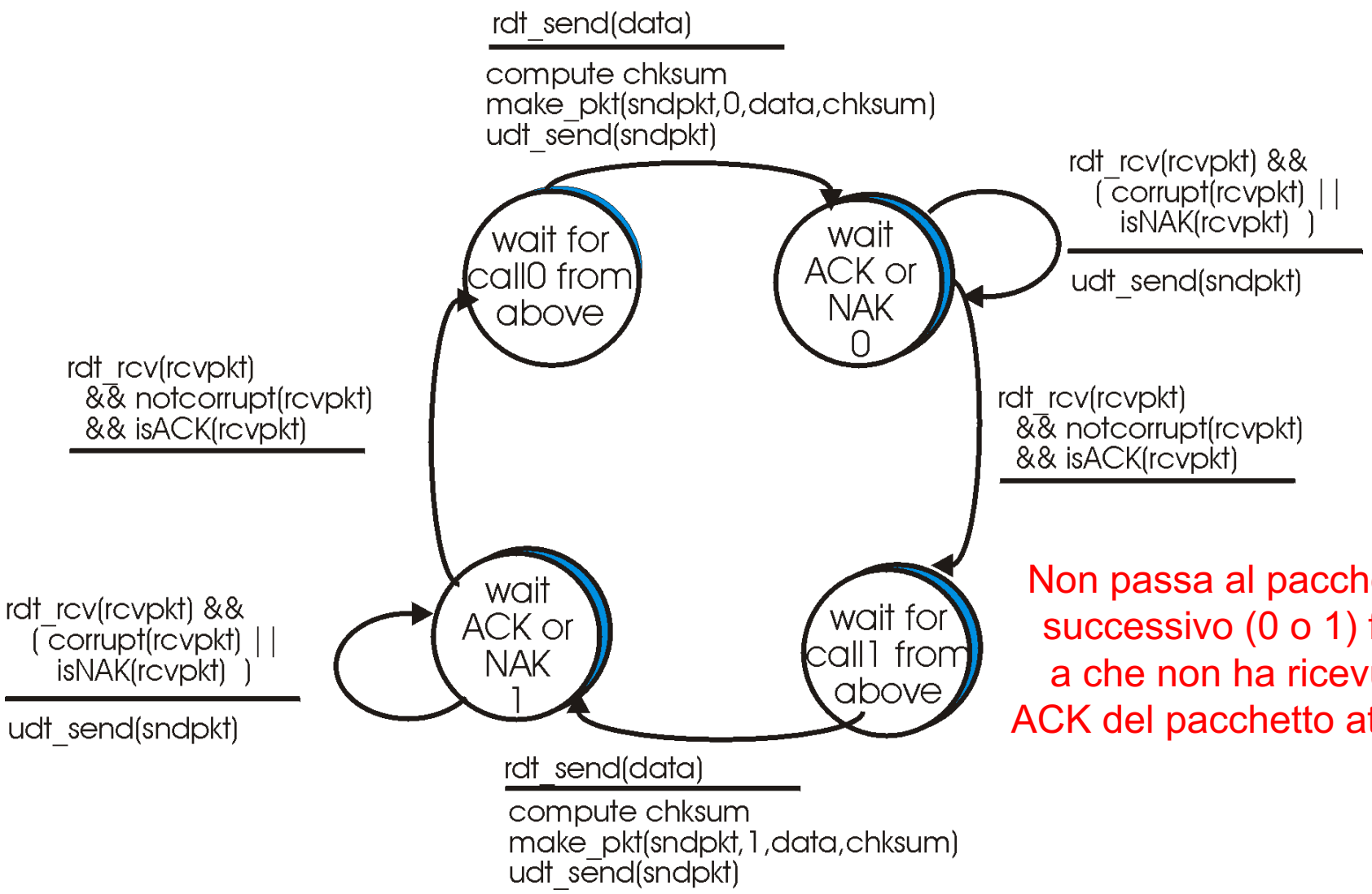
Gestione dei duplicati

- Il sender aggiunge *numero di sequenza (SN)* ad ogni segmento
- Il sender ritrasmette l'ultimo pacchetto con stesso SN finchè non riceve ACK
- Il ricevente elimina i segmenti duplicati (con stesso SN)

Protocollo Stop and wait

Questa tecnica si basa sull'invio iterato di uno stesso segmento fino alla corretta ricezione (senza passare al successivo fino ad allora)

rdt2.1: lato sender, ora gestisce ACK e NAK errati

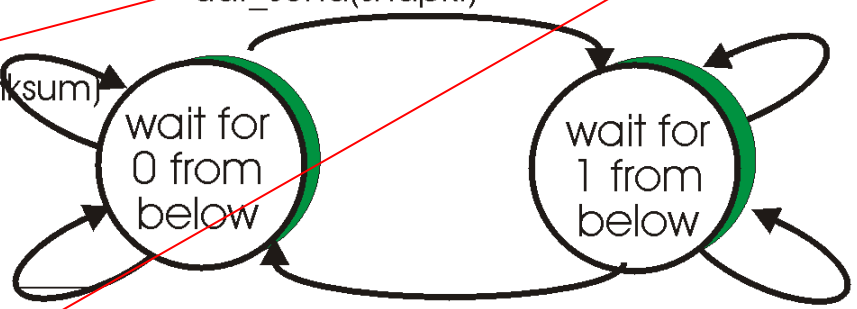


Non passa al pacchetto successivo (0 o 1) fino a che non ha ricevuto ACK del pacchetto attuale

rdt2.1: lato receiver, gestisce segmenti duplicati

Se il sender numera i pacchetti il receiver sa distinguere eventuali duplicati inviati a causa di un errore sull'ACK

Invia sia ACKs che NACKs



rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
make_pkt(sendpkt,ACK,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

compute chksum
make_pkt(sndpkt,NAK,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

compute chksum
make_pkt(sndpkt,ACK,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

compute chksum
make_pkt(sndpkt,NAK,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

compute chksum
make_pkt(sndpkt,ACK,chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
compute chksum
make_pkt(sendpkt,ACK,chksum)
udt_send(sndpkt)

Gestisce pacchetto 1 duplicato

Gestisce pacchetto 0 duplicato

rdt2.1: punto della situazione

Sender:

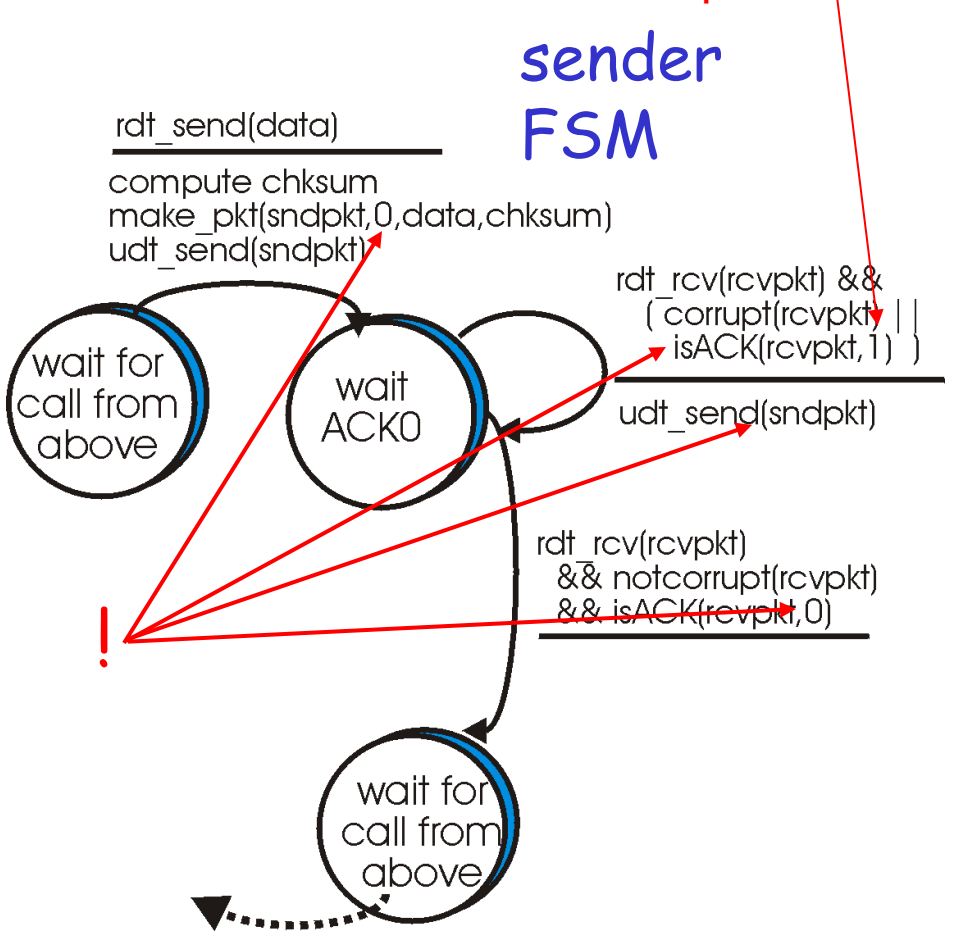
- Aggiunge numero di sequenza ai pacchetti
- Bastano numeri 0 e 1 perchè è stop & wait
- Verifica se ACK e NAK sono corretti
- Ha ora il doppio degli stati
  - Lo stato serve a ricordare se l'ultimo segmento era 0 o 1

Receiver:

- Deve verificare se riceve segmenti duplicati
  - Lo stato dice quale pacchetto si aspetta (0 o 1)
- N.B. il receiver non sa se il ACK o NAK sia stato ricevuto dal sender...

rdt2.2: eliminiamo ora il NAK!

- Invece di NAK, receiver spedisce sempre ACK per l'ultimo pacchetto ricevuto corretto!
  - receiver inserisce il numero del pacchetto corretto nell'ACK
- Se il sender riceve ACK duplicati li interpreta come NAK! ...*E ritrasmette il pacchetto successivo a quello a cui si riferisce l'ACK duplicato*



rdt3.0: rete con errori sui bit *e perdita dei pacchetti*

Nuova assunzione: la rete può anche perdere i pacchetti (sia dati che ACK)

- checksum, seq. #, ACKs, e ritrasmissione non bastano più...

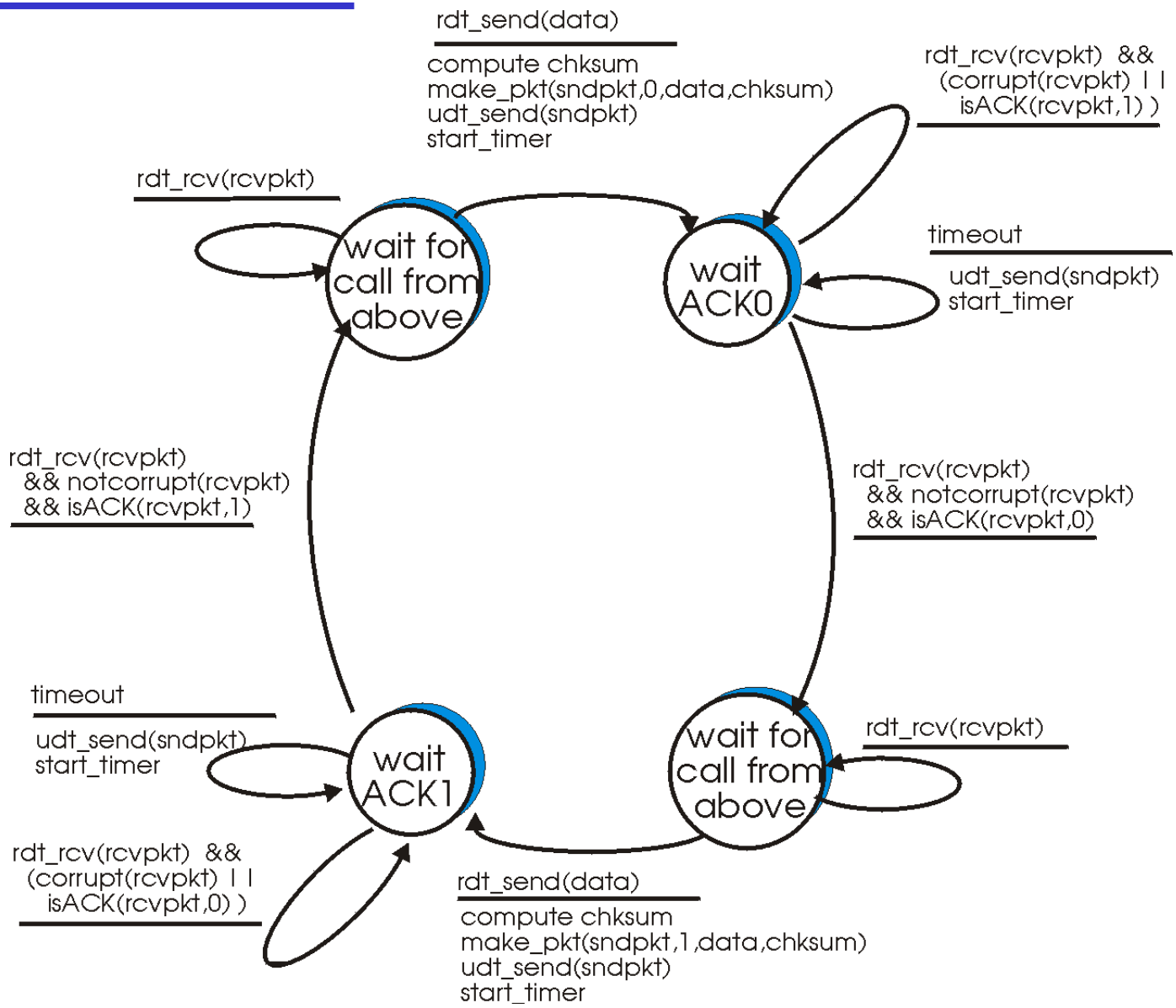
Q: come si gestisce la perdita di pacchetti?

- Il sender rischia di bloccarsi all'infinito nell'attesa di un ACK.

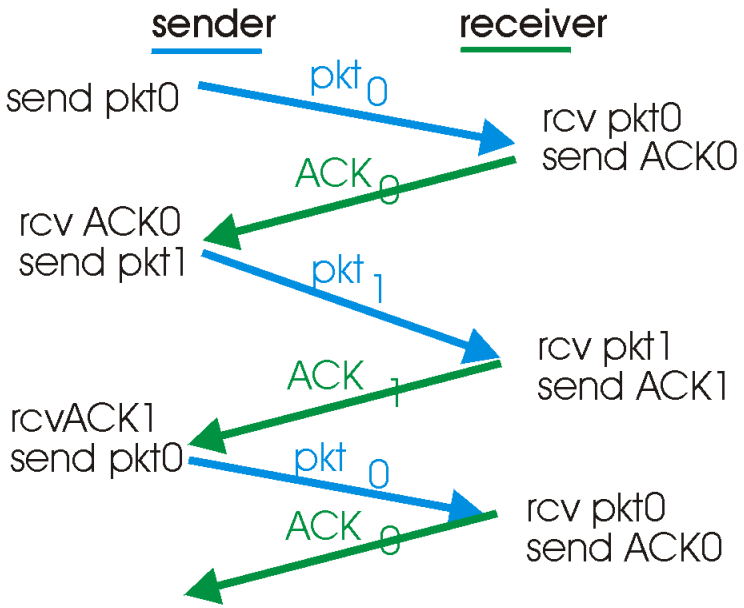
Soluzione: sender attende l'ACK per un tempo ragionevole (timeout)

- ...e ritrasmette se il Timeout scade senza avere ricevuto ACK
- Ma se il pacchetto dati (o l'ack) arrivassero dopo il timeout?
  - Allora si è trasmesso un duplicato, ma il numero di sequenza risolve già il problema sul receiver
  - receiver deve specificare il numero di pacchetto del quale invia l'ACK
- Ma ora serve un timer!!!

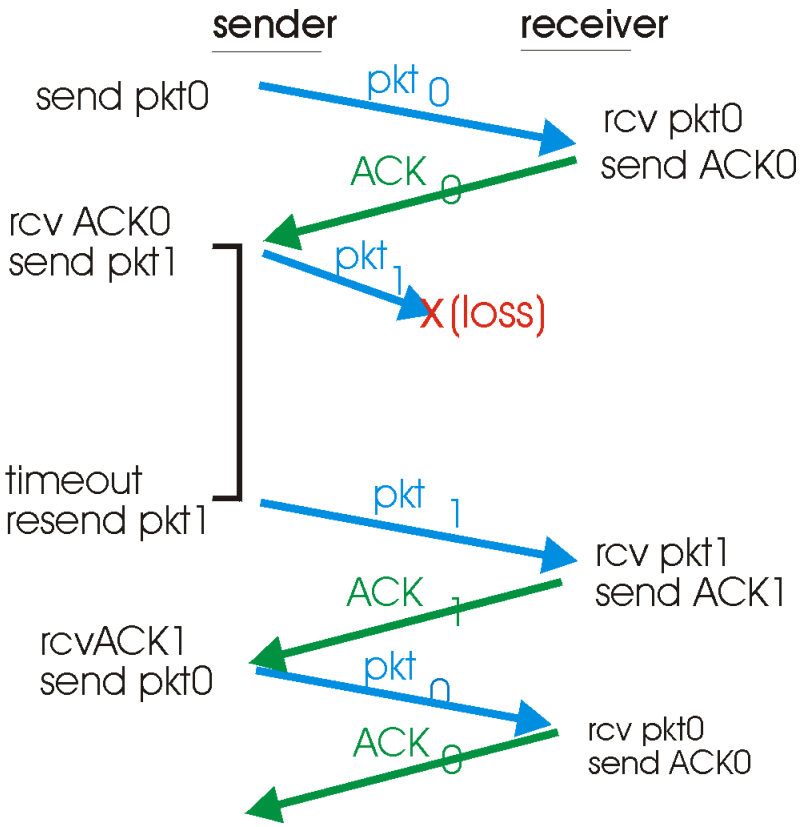
rdt3.0 sender



rdt3.0: esempio 1

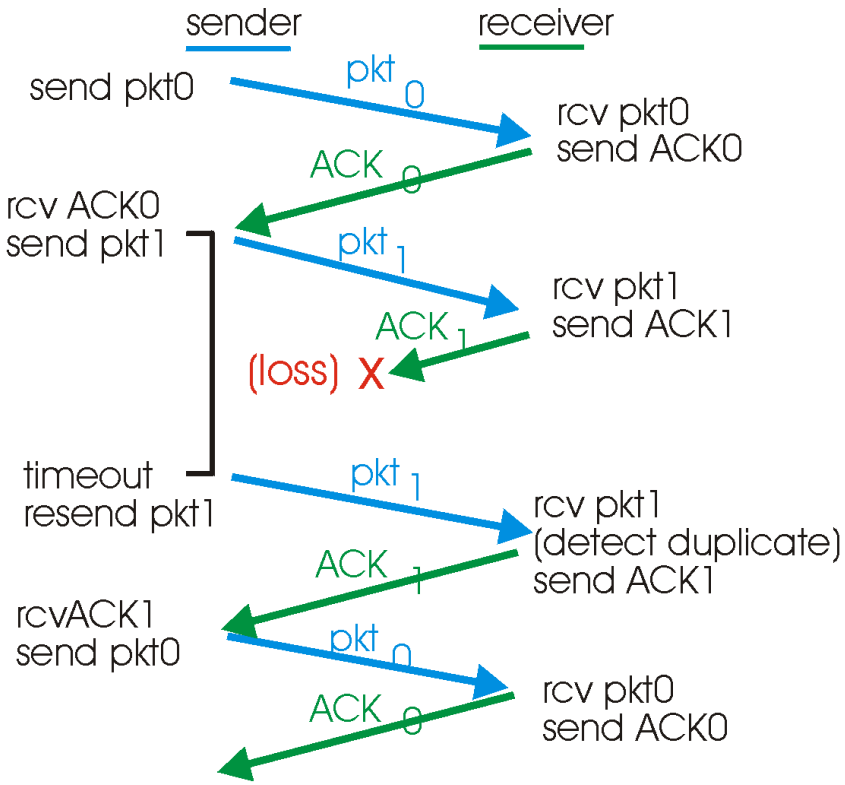


(a) operation with no loss

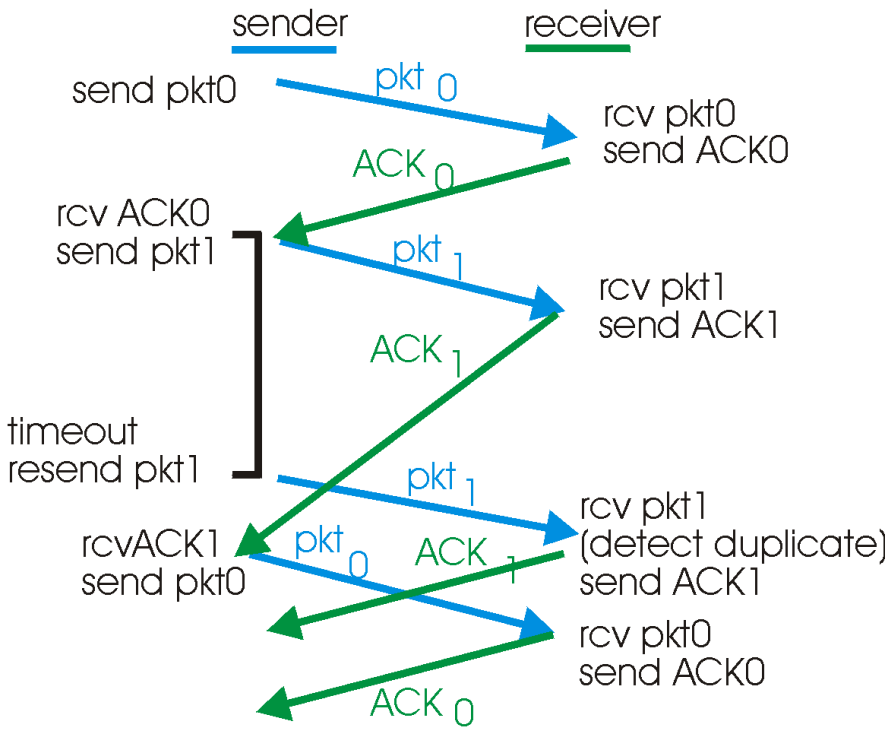


(b) lost packet

rdt3.0: esempio 2



(c) lost ACK



(d) premature timeout

Prestazioni del protocollo nella versione: rdt3.0

- rdt3.0 è un livello di trasporto affidabile, e in pratica racconta come funziona TCP, ma le prestazioni sono inaccettabili!
- esempio: data rete a R=1 Gbps, RTT=15 ms ritardo da sender a receiver, e voglio spedire pacchetti da L=1KB
- Infilo nel canale un pkt (T) e poi aspetto che mi torni l'ACK (2 RTT)

$$T_{transmit} = \frac{8kb/pkt}{10**9\ b/sec} = L/R = 8\ microsec$$

$$\text{Utilizzo} = U = \frac{\text{Frazione di tempo in cui il canale è occupato}}{} = \frac{8\ microsec}{30.008\ msec} = 0.00027$$

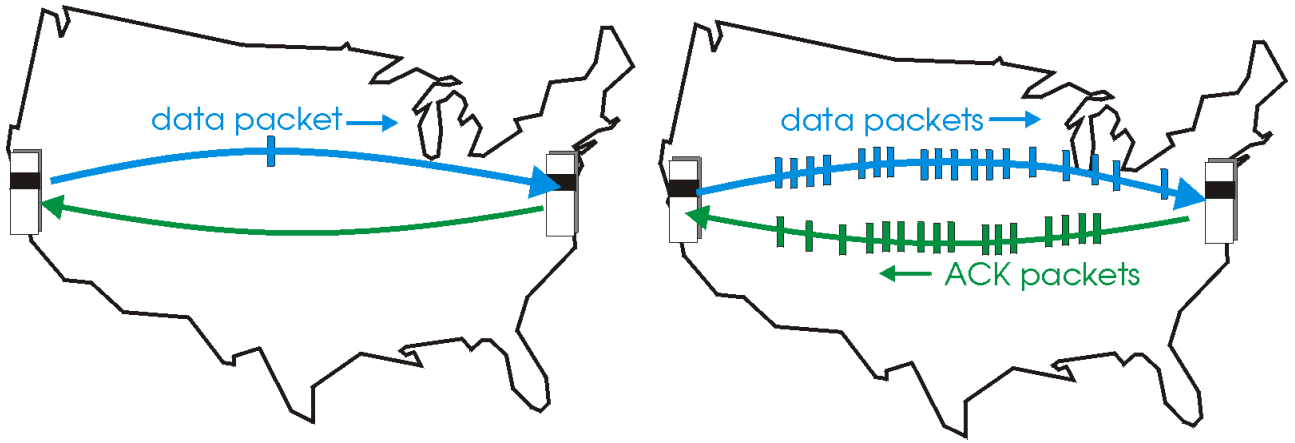
$$\text{Utilizzo} = U = \frac{\text{Frazione di tempo in cui il canale è occupato}}{} = \frac{L/R}{RTT + L/R} = 0.00027$$

  - 1KB circa ogni 30 msec -> 33kB/sec su una rete a 1 Gbps!!!
  - Il protocollo di livello trasporto limita troppo le prestazioni del sistema. Occorre trovare una soluzione più valida, e pensare ad eliminare il principio Stop & Wait in favore dei protocolli a Pipeline.

Protocolli a Pipeline

Pipelining: sender trasmette più di un segmento prima di ricevere il riscontro del primo di essi...

- Quindi non basta più numerare 0 e 1 i pacchetti
- Serve anche buffering su sender e receiver



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

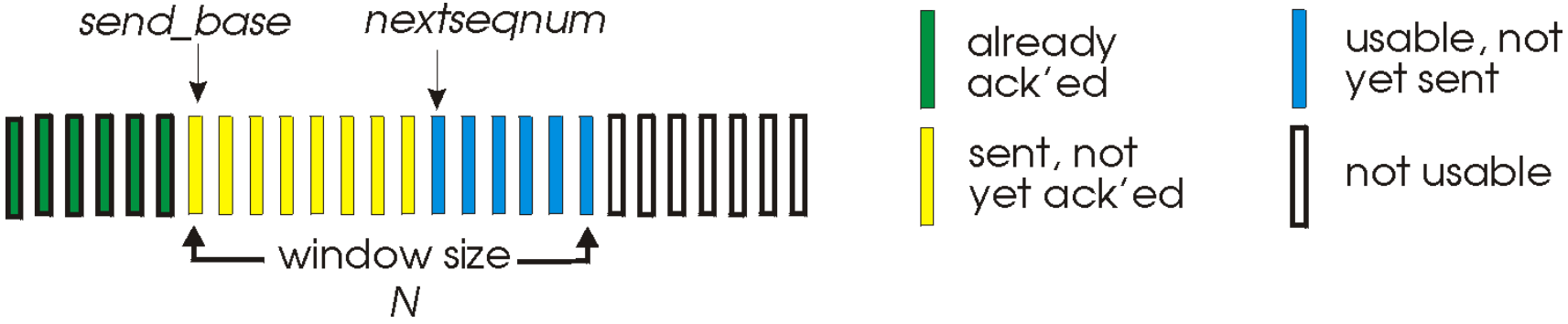
(b) a pipelined protocol in operation

- Esistono due classi di protocolli a pipeline: *go-Back-N, selective repeat*

Go-Back-N (o Sliding Window)

Sender:

- Nell'header di ogni pacchetto c'è un numero di sequenza su k bit
- "finestra" di max N=2**k (per controllo di flusso), consecutivi pacchetti possono essere spediti

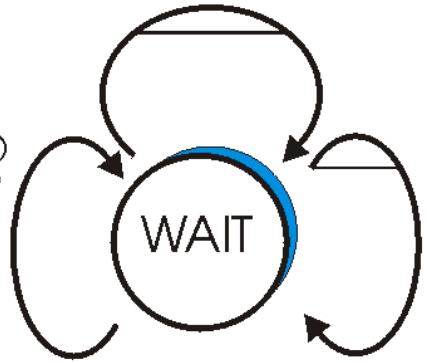


- Se ricevo ACK(n): vale per tutti i pacchetti fino a n - "ACK cumulativo"
- C'è un timer per ogni pacchetto in sospeso (timeout)
  - *Se scatta timeout(n):* ritrasmetto tutti i pacchetti da n in poi appartenenti alla finestra scorrevole (sliding window)

GoBackN: lato sender

rdt_send(data)

```
if (nextseqnum < base+N) {
   compute chksum
   make_pkt(sndpkt(nextseqnum)),nextseqnum,data,chksum)
   udt_send(sndpkt(nextseqnum))
   if (base == nextseqnum)
     start_timer
   nextseqnum = nextseqnum + 1
   }
else
  refuse_data(data)
```



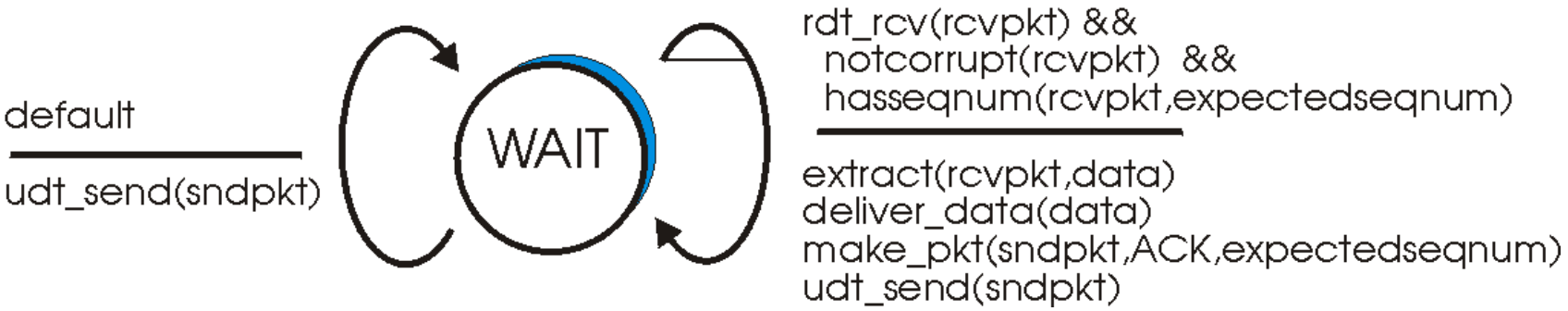
rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)

```
base = getacknum(rvcpkt)+1
if (base == nextseqnum)
   stop_timer
  else
   start_timer
```

timeout

```
start_timer
udt_send(sndpkt(base))
udt_send(sndpkt(base+1)
……
udt_send(sndpkt(nextseqnum-1))
```

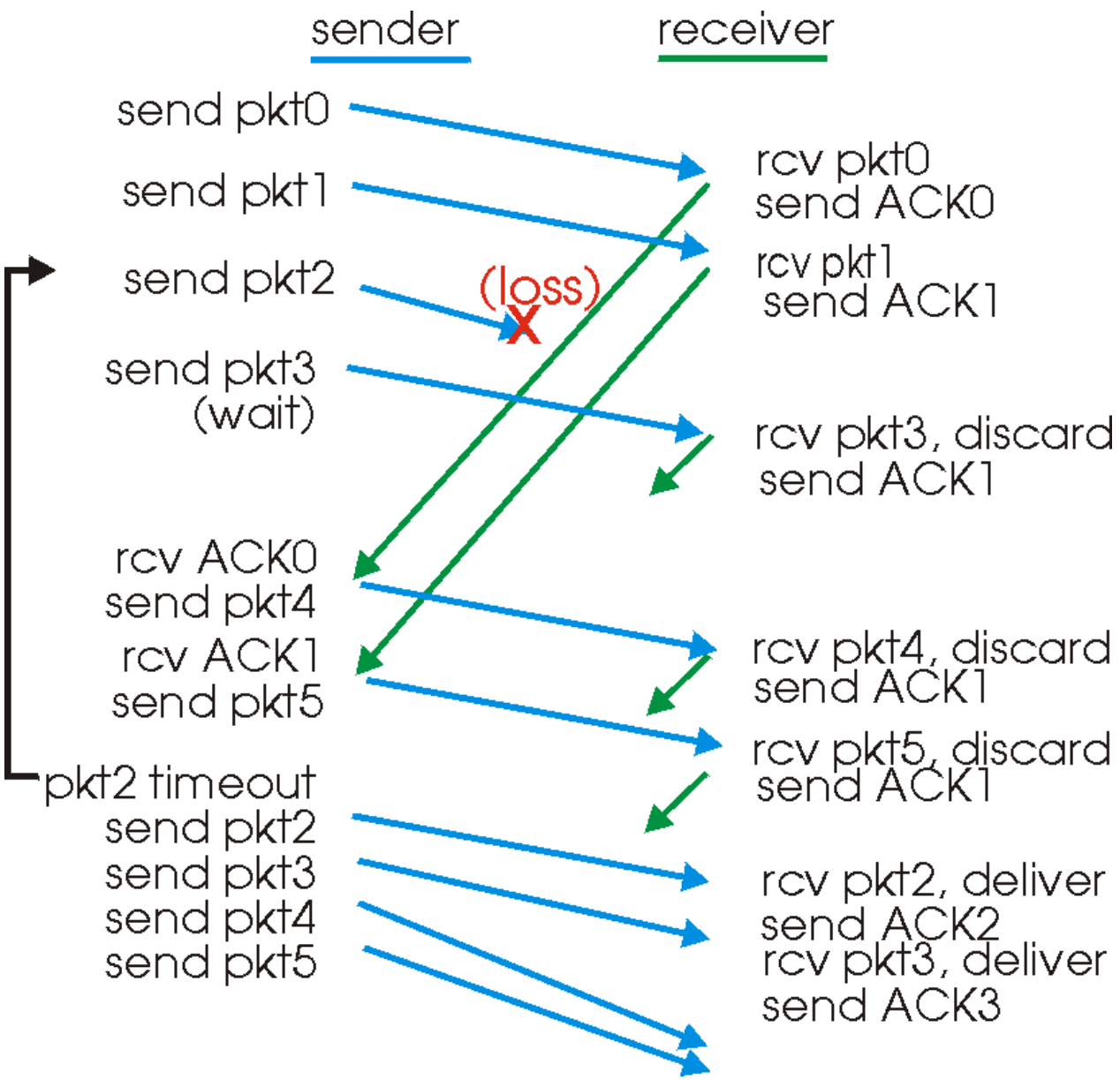
GoBackN: lato receiver



Il receiver è semplice:

- ACK: spedisce sempre ACK relativo al pacchetto con numero di sequenza maggiore, prima del quale tutti i pacchetti sono stati ricevuti; pacchetti ricevuti fuori ordine sono scartati!
  - Può spedire in modo testardo ack duplicati per ribadire che sta ricevendo pacchetti successivi a quello mancante
  - Deve ricordare solo il numero di sequenza del prossimo pacchetto che gli manca, in ordine = **`expectedseqnum`**
- I pacchetti ricevuti correttamente, ma fuori ordine
  - Sono eliminati(non bufferizzato) -> perchè no?
  - Invia sempre e solo Ack del pacchetto più in alto nella sequenza completa dei pacchetti ricevuti

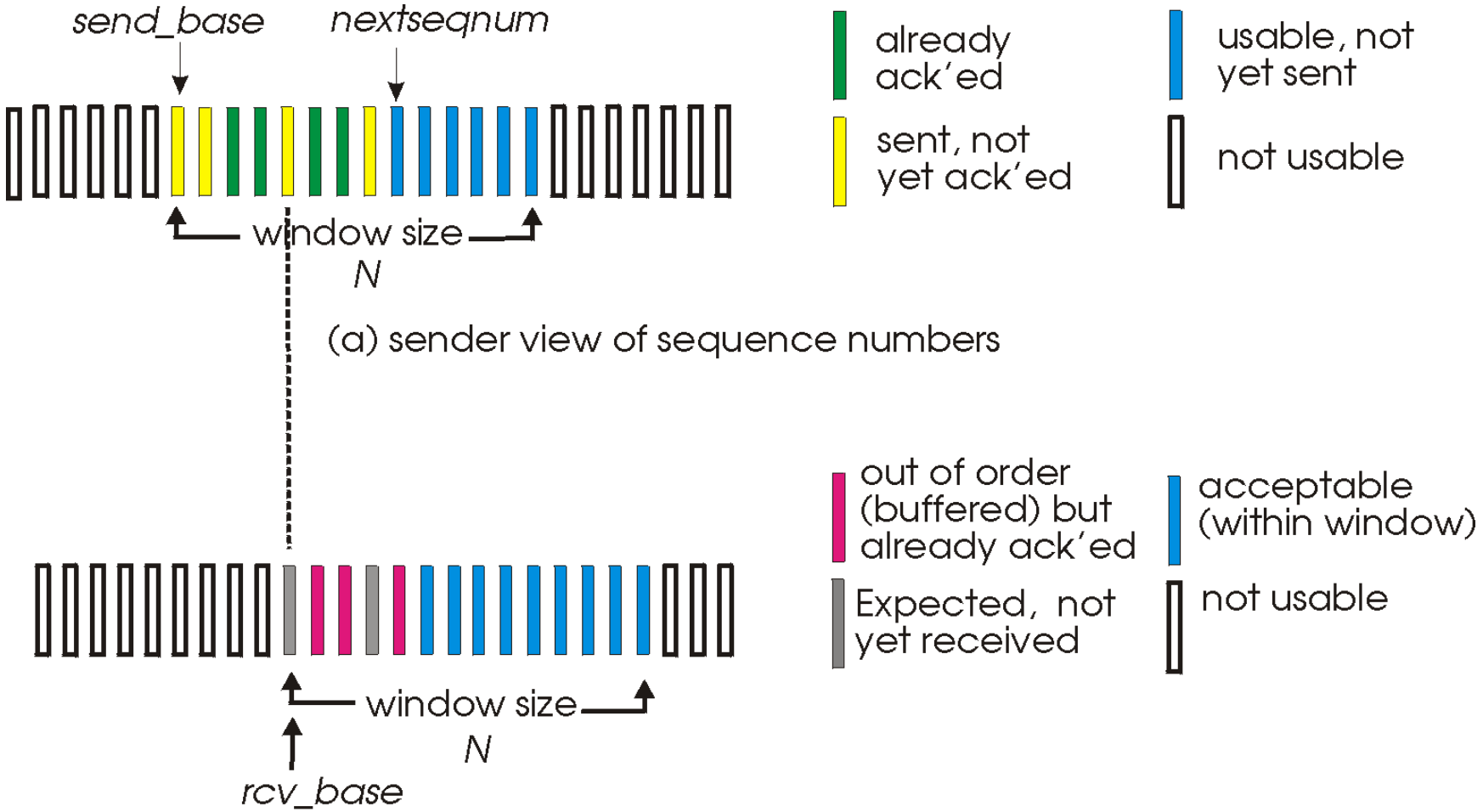
GoBackN in azione N=4



Selective Repeat

- receiver *spedisce ACK individuali di ogni pacchetto ricevuto (no ack cumulativi)*
  - Bufferizza i pacchetti ricevuti fuori sequenza per poi mandare segmenti completi all'applicazione
- sender rispedisce solo i pacchetti per i quali non ha ricevuto ACK (e non tutta la finestra)
  - C'è un timer per ogni pacchetto in sospeso
- La finestra del sender esiste ancora
  - N numeri di sequenza consecutivi
  - Limita il numero massimo di pacchetti in sospeso (controllo di flusso)
- Esiste anche una finestra sul receiver

Selective repeat: finestre di sender e receiver



Selective repeat

sender

Dati dall'applicazione :

- Se la finestra non è tutta usata spedisci il pacchetto

timeout(n):

- Rispedisci pacchetto n e riavvia il timer(n)

ACK(n) ricevuto appartenente alla finestra [sendbase,sendbase+N]:

- Marca pacchetto n ricevuto
- se n era il primo pacchetto senza ACK, avanza la finestra fino al prossimo pacchetto senza ACK

receiver

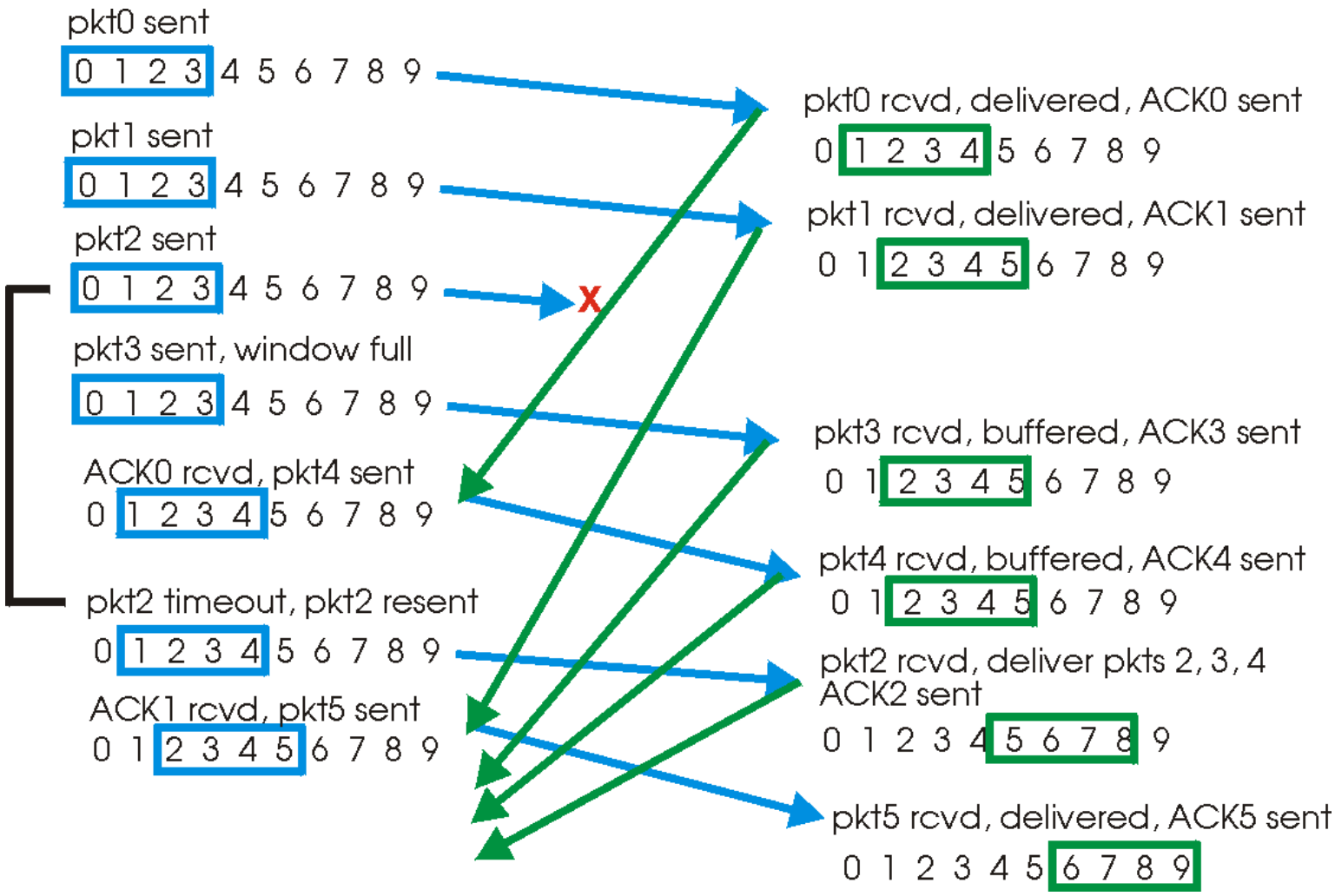
Se riceve pacchetto n appartenente alla finestra del receiver [rcvbase, rcvbase+N-1]

- spedisce ACK(n)
- Se è fuori ordine: inserisci pacchetto in buffer
- Se la finestra contiene una serie di pacchetti in ordine, passali all'applicazione e avanza la finestra sul primo pacchetto mancante

Se pacchetto n duplicato

- ACK(n)

Selective repeat in action



Selective repeat: problema con la dimensione della finestra

esempio

- seq num: 0, 1, 2, 3
- Dimensione finestra=3
- receiver non vede differenza nei due scenari! In (a) quindi passa un duplicato all'applicazione come se fosse il dato atteso (errore!)
- La dimensione della finestra deve essere inferiore o uguale alla meta' dello spazio dei numeri di sequenza
- E se disordina il canale e ripropone casualmente vecchi pacchetti? TCP ipotizza TTL = 3 min