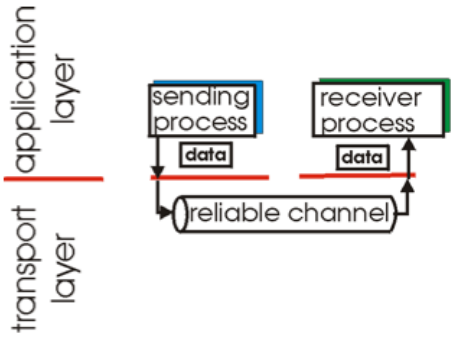
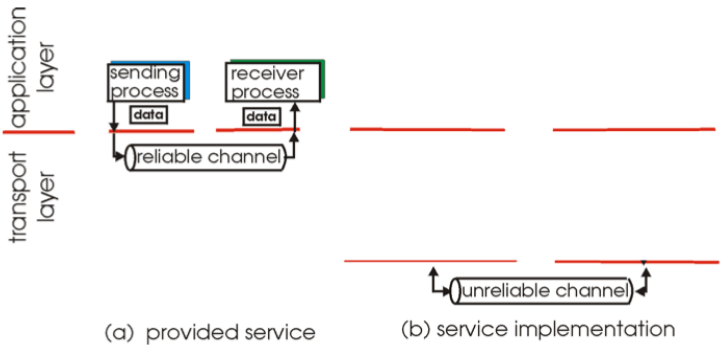
Principi di trasferimento dati affidabile

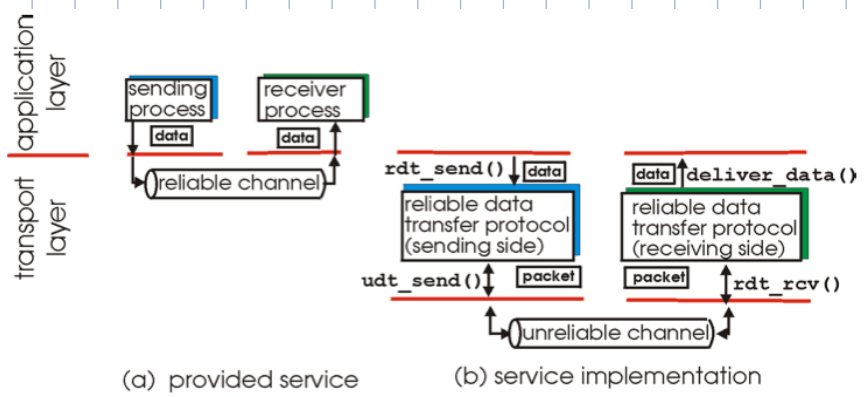
Trattiamo un trasferimento dati, quindi c'è un'entità che genera dati e invia e una che riceve.



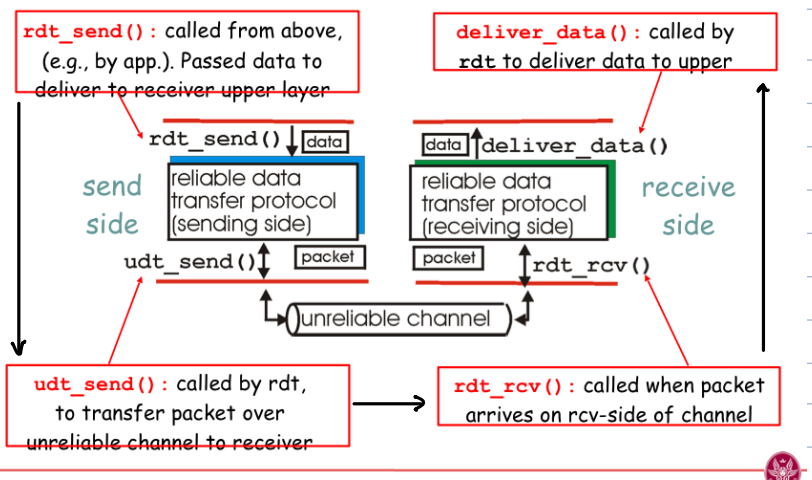
Le entità di uno strato dialogano direttamente tra di loro in senso logico ma poi operativamente per effettuare il trasferimento l'entità che invia passa i dati all'entità locale dello strato inferiore che a sua volta comunica con l'entità di strato remota, fino ad arrivare ai mezzi di comunicazione.
Se il trasferimento offerto dallo strato inferiore è di qualità adeguata, cioè è sufficientemente affidabile nel trasferimento dati, allora non c'è nient'altro da aggiungere.
Se questo non accade:



Il livello sottostante è inaffidabile, ossia il livello dì integrità non è sufficiente per gli scopi dei processi. Bisogna colmare quindi questo divario, si mette in mezzo una funzione che si chiama "funzione di trasferimento affidabile" (Reliable data transfer, RDT), è come uno strato che si mette in mezzo tra il livello superiore dove ci sono il processo sender e reciver e lo strato dove c'è il trasferimento non adeguato. Lo scopo di questo strato aggiuntivo è elevare l'affidabilità del canale, introducendo uno specifico protocollo.

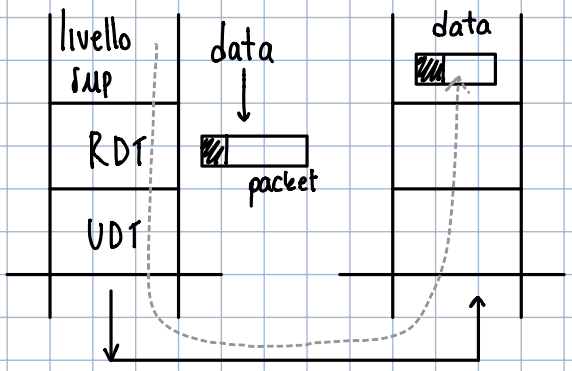


Avremo un'entità che chiameremo RDT sending side e una RDT receiving side.



rdt_send() è un comando che viene inviato dal livello superiore diretto al rdt sending side di questo strato a sua volta questa entità da comandi al livello inferiore, udt_send(). Il livello inferiore quando porta i dati a destinazione ha una primitiva che si chiama rdt_rcv() che "avvisa" che sta arrivando il pacchetto.
Il protocollo rdt avvisrà che ci sono dei dati da consegnare attraverso il comando deliver_data.

È vedere in dettaglio quello che abbiamo visto in passato:



Non è detto che il reciver mandi direttamente i dati, prima si accerta che non ci siano errori (sennò non avrebbe senso di stare la).

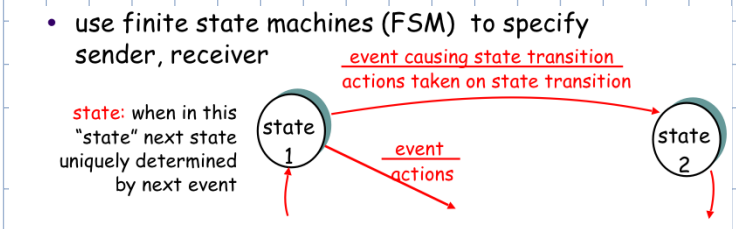
Progettiamo il protocollo rdt attraverso una macchina a stati, è la descrizione di un processo nel tempo dove si identificano una serie di variabili che rappresentano uno stato.
Ad esempio :

$n \in [0, 1, \ldots, 16]$

$b \in \{TRUE, FALSE\}$

$(n, b) \leftarrow$ 34 possibili valori (= stati)

Tra uno stato e l'altro ci possono essere transizioni, ossia passaggio da stato a stato che viene denotato con un arco e sopra l'arco si mette un'etichetta che descrive l'evento che produce la transizione sotto l'azione che si esegue nel fare questa transizione.

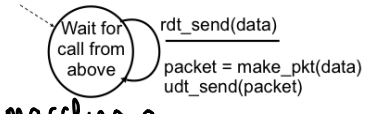


È un modo di descrivere formalmente come funzionano sender e reciver.

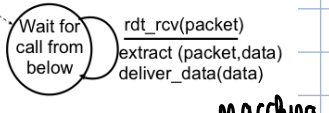
Macchine a stati del protocollo RDT :
- Lato sender
- Lato receiver

Progetto per gradi di approssimazione successivi

Supponiamo che il canale unreliable sia perfetto, allora il protocollo rdt.0 non fa niente.
Il lato sender attende che dall'alto ci sono dati da mandare (wait for call from above). Quando arriva questa chiamata, rdt_send(data) gli dà come azione packet= make_pk(data), udt_send(packet) ossia manda diretto perché il canale è affidabile.
Il lato receiver aspetta una chiamata dal basso (wait for call from below). Quando gli arrivano i pacchetti li estrai e notifica al livello superiore che ci sono dati per lui.
In questo caso non c'è bisogno di nessun protocollo se il canale è affidabile.



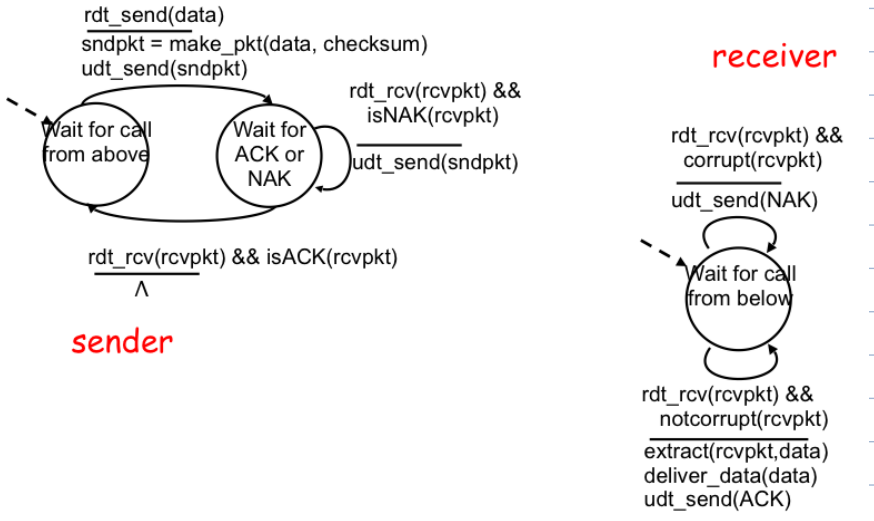
macchina a stati del sender



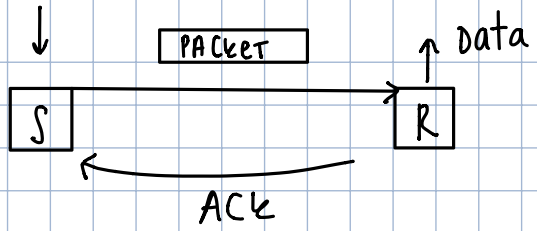
receiver macchina a stati del receiver

Nel caso in cui non sia affidabile, passiamo a rdt2.0, ossia potrebbe modificare alcuni bit. L'idea è che siccome so che il canale inaffidabile potrebbe mandare pacchetti con errori allora inserisco nel pacchetto un comando di rilevazione degli errori. Quando ricevo un blocco di dati dal basso dal momento in cui controllo e non ci sono errori allora li estraggo e li mando allo strato superiore, se trovo invece degli errori scarto quei dati e richiedo una trasmissione. Questo metodo si basa sul fatto che gli errori sono considerati casuali, il tutto funziona bene se la probabilità che ci siano errori nella PDU è bassa.
Quando un sender manda dati, il reciver controlla che non ci siano errori e ne, caso mando un riscontro positivo (Acks). Se quello che ha ricevuto contiene errori manda un riscontro negativo (Naks) e invita l'altra entità a ritrasmettere.

Vediamo come funziona:

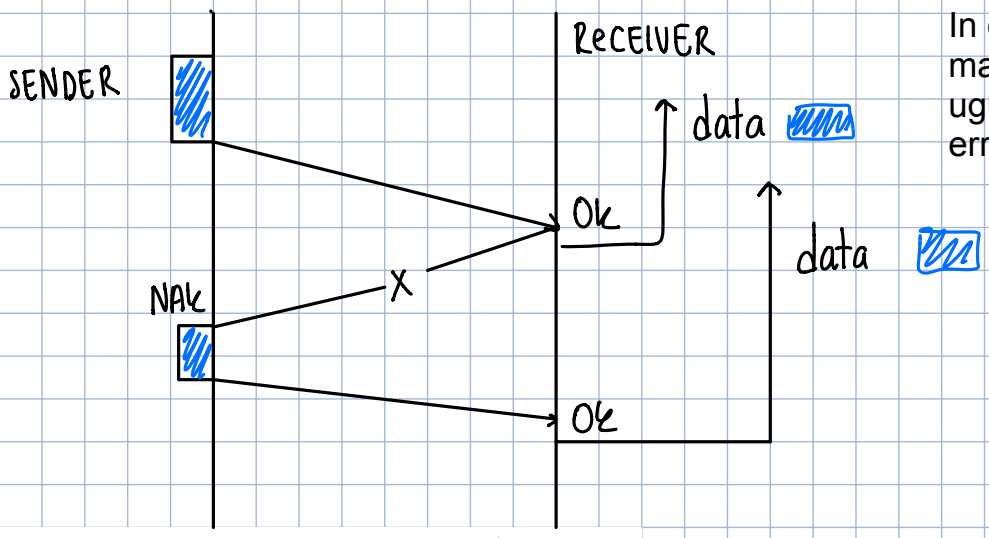


Lato sender aspetta che dall'alto gli arrivi una chiamata. Gli arriva l'rdt_data(), evento che lo smuove. Costruisce un blocco di bit che si chiama send packet, è fatto da dati che contengono un checksum (un campo di verifica di errori, ad esempio un CRC polinomiale). Chiama il livello sottostante mandandogli questo pacchetto . Dopo che ha mandato aspetta che il ricevitore dia un riscontro, quindi il sender aspetta una chiamata dal basso che gli dà un ACK o un NAK. Nel caso sia NAK gli viene chiesto di ritrasmettere. Si esce dalla ritrasmissione solo quando si riceve un ACK. Quando sta nella wait area, non si fa mandare pacchetti dall'alto.
Il lato ricevitore aspetta una chiamata dal basso e attraverso il checksum vede se ci sono errori, nel caso ci siano il livello inferiore viene incaricato di mandare un NAK.
Nel caso in cui non ci siano errori, i dati vengono estratti e consegna i dati al livello superiore e consegna alla sua entità alla pari un ACK.



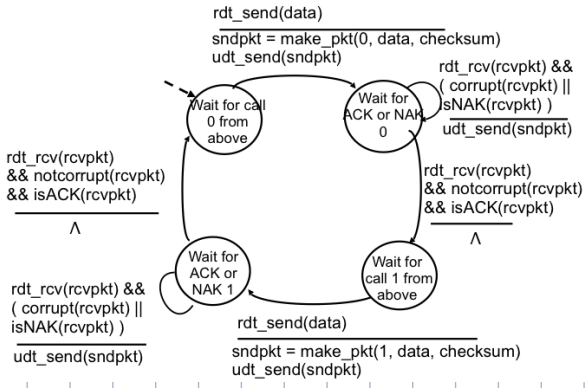
Una mancanza che c'è in questo protocollo è che non c'è un limite in questa ritrasmissione quindi andrebbe inserita una variabile di stato che conta il numero di tentativi mettendo una soglia massima.

La critica a questa soluzione è che anche ACK e NAK sono dei pacchetti, seppur brevi, e quindi possono essere colpiti da errore.



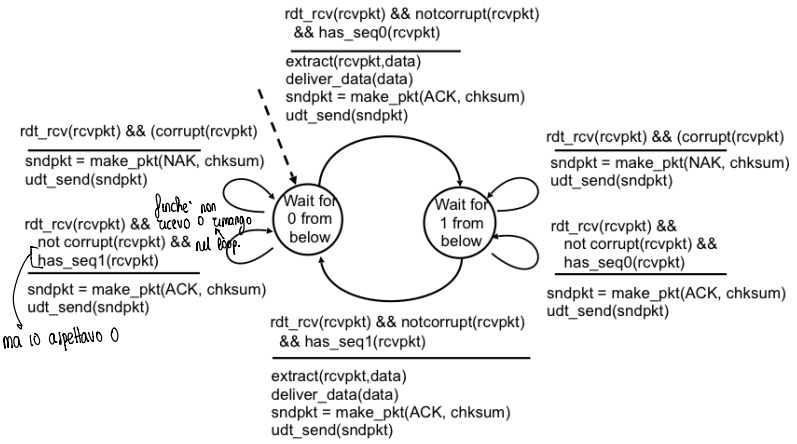
In questo modo si mandano due pacchetti uguali, il che è un errore

rdt2.1: sender, handles garbled ACK/NAKs

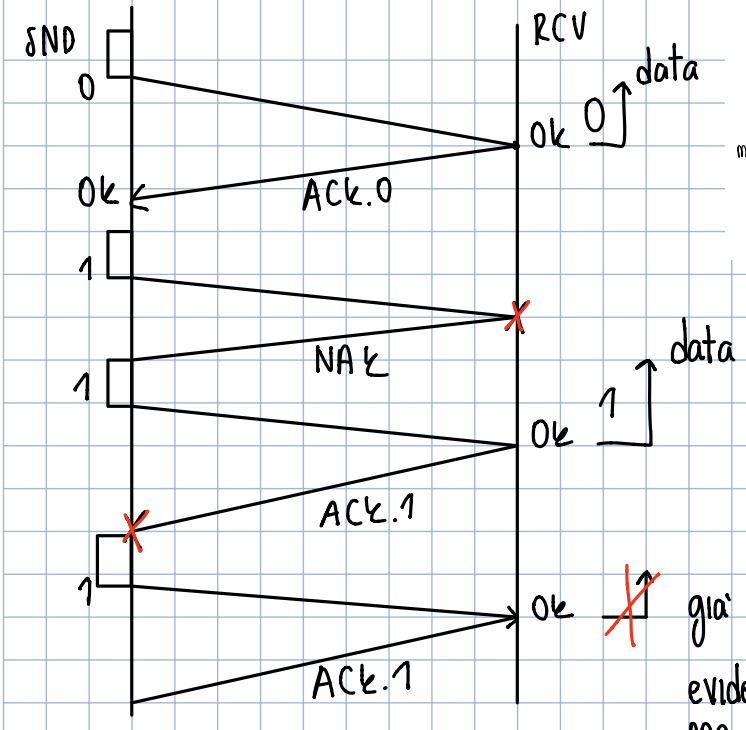


Inserisco due stati: 0 e 1, indicano il lo stato a cui si riferiscono.
Quando viene mandato il pacchetto 0 si attende il riscontro del pacchetto 0.
Dopo 1 posso riusare 0 senza ambiguità.

rdt2.1: receiver, handles garbled ACK/NAKs



Esempio di evoluzione:

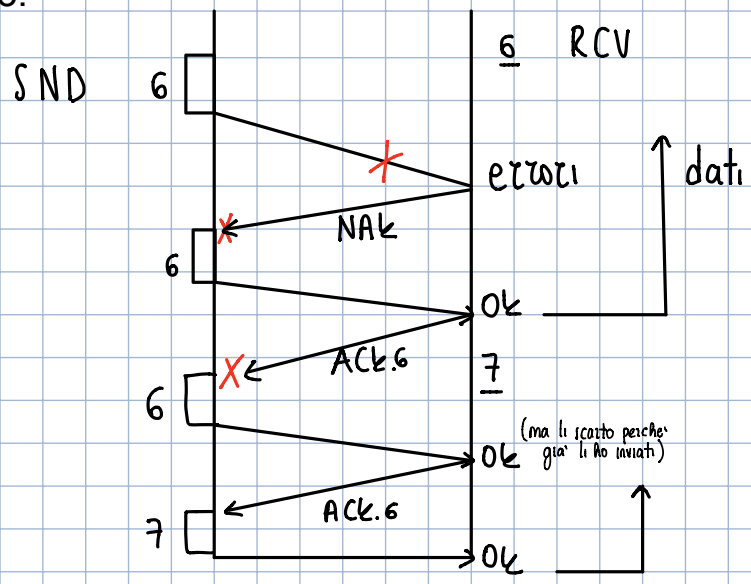


già li ho mandati

evidentemente il sender non ha capito ⇒ rimanda ACK.1

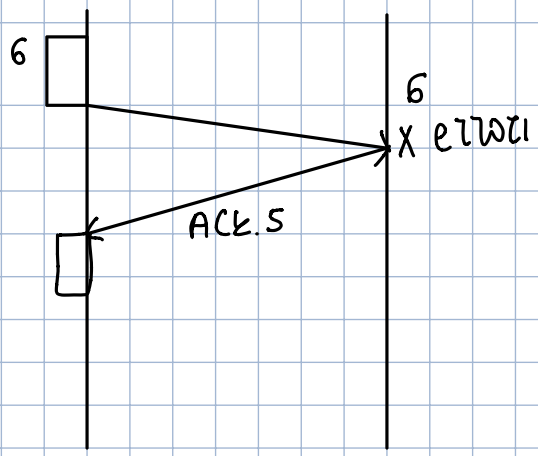
È sbagliato dire che se due pacchetti con etichette diverse sono uguali allora sono duplicati e non li mando. Il contenuto non deve interessare il trasferimento.

Esempio:

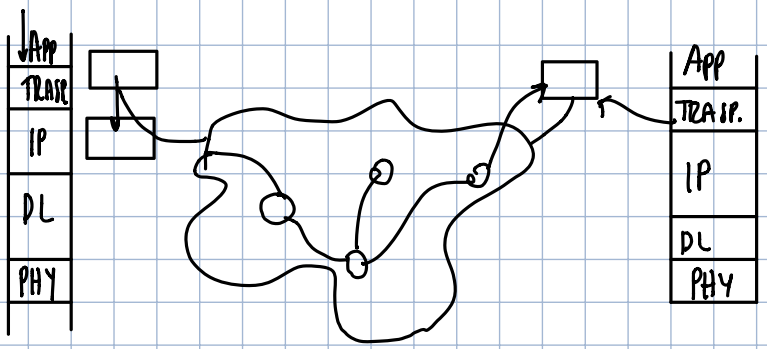


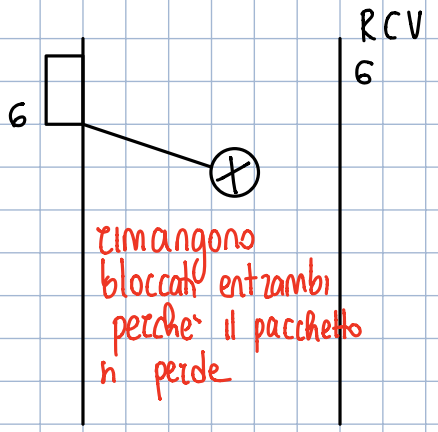
Comunque se si mandano 1000 pacchetti, sempre 1000 pacchetti si ricevono anche se con errori.

Quindi è indispensabile: l'etichetta del pacchetto e gli ACKS.
Invece i NAKS non sono indispensabili, posso sostituirli con riscontri positivi sul pacchetto precedente.



Consideriamo un canale inaffidabile con errore e perdite, quindi è possibile che il pacchetto non arrivi.



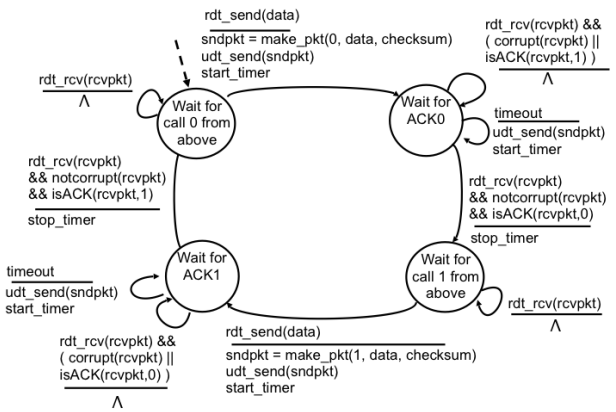


Come protocollo finale rdt3.0 è con l'aggiunta dei timer.

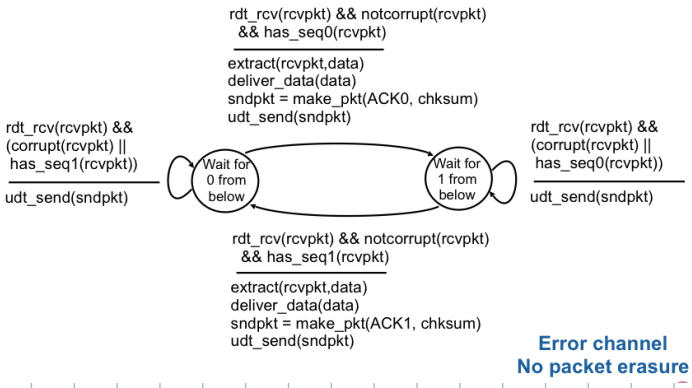
4 elementi fondamentali di un rdt:
1) checksum
2) numeri di sequenza
3) ACK
4) Timer

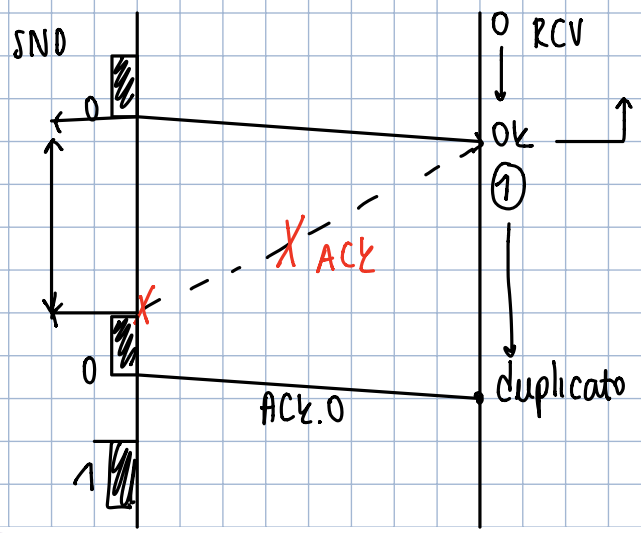
Questi protocolli vengono chiamati anche ARQ (automatic repeat request).

rdt3.0: sender FSM



rdt3.0: receiver FSM

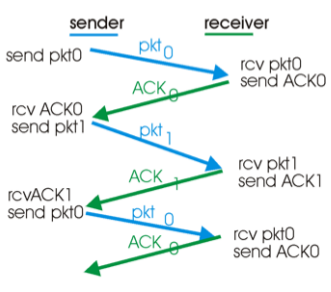




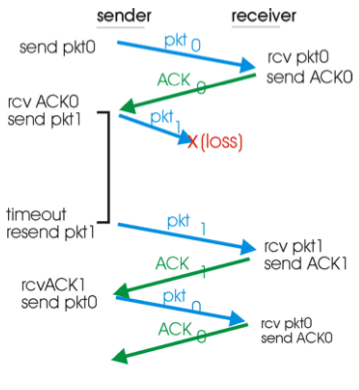
I PROTOCOLLI RDT AVVENGONO QUANDO C'È CONNESSIONE

rdt3.0 in action

STOP AND WAIT



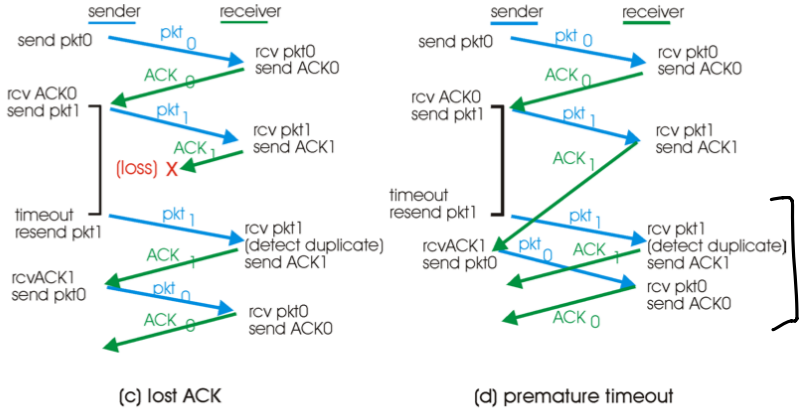
(a) operation with no loss



(b) lost packet

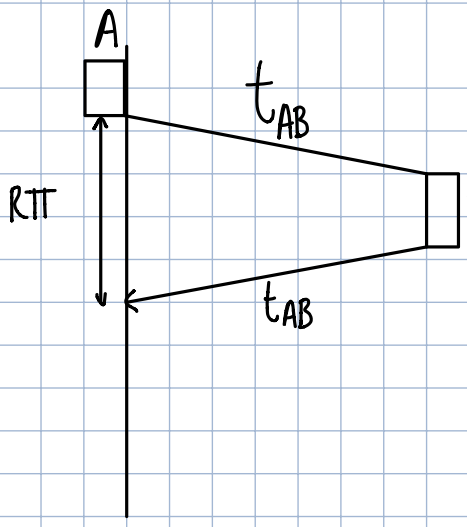
Questo protocollo può però mandare un solo pacchetto alla volta, non può mandarne più in contemporanea.

rdt3.0 in action



Il timer non deve scadere precocemente.

Quanto deve valere il timer?



Round trip time (RTT) → tempo di andata e ritorno → se questo è costante

⇒ il timer ≥ RTT

(se è variabile la situazione si complica)

tempo di elaborazione

$timer \geq t_{AB} + t_e + t_{ACK} + t_{BA}$

t_{AB}: tempo per mandare il pacchetto

t_{ACK}: tempo per il riscontro

t_{BA}: tempo trasmissione riscontro

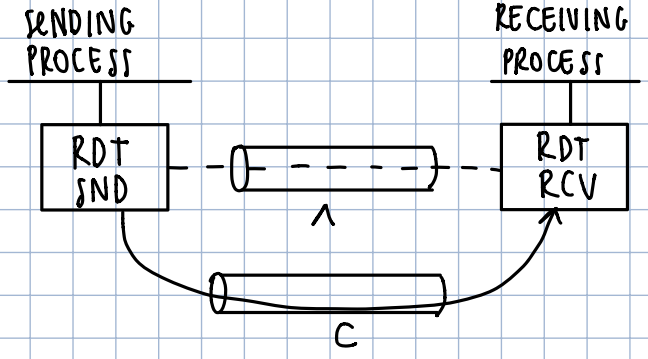
Quando RTT è variabile c'è il problema di quale timer mettere, che vedremo più avanti.

RDT perfomance

1) Problemi di portata: la capacità non è sfruttata al massimo poichè "attendo."

C = unreliable capacity (bit/s)

Λ = net (average) capacity offered by RDT to upper layer entities.



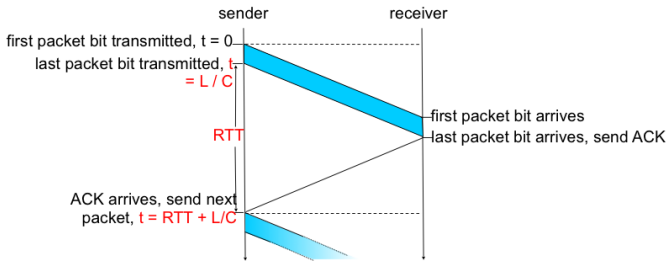
$\Lambda < C$ (l'efficienza è < 1)

C = 10 Mbit/s

Λ = 8 Mbit/s

è il prezzo che pago sulla capacità per avere una trasmissione affidabile

stop-and-wait operation = RDT. 3.0



$$U = \frac{L/C}{RTT + L/C} = \frac{1}{1 + C \cdot RTT/L}$$] BANDWITH DELAY PRODUCT

è la frazione di tempo in cui utilizzo il canale affidabile sottostante ⟶ appena finisce il tempo viene mandato un pacchetto.

Esempio:

1 Gbps link, 15 ms prop. delay, 8000 bit packet: $d_{trans} = \frac{L}{C} = \frac{8000 \text{ bits}}{10^9 \text{ bps}} = 8 \mu s$

$$BDP = \frac{C \cdot RTT}{L} = \frac{10^9 \cdot 30 \cdot 10^{-3}}{8000} = 3.75 \cdot 10^9$$

$$U_{sender} = \frac{L/C}{RTT + L/C} \approx 0.00027$$

L'efficienza dello stop and wait dipende dal prodotto banda ritardo, se è basso allora è efficiente.
Come si rimedia quando è grande? Con i protocolli Pipelining