

Cerchiamo di implementare un sistema affidabile di comunicazione. (verso TCP) SCHEMA PAG. 183-185

RDT (1.0)

(Reliable data transfer)

Implementare come un affidabile protocollo di comunicazione.

Come un affidabile servizio gestito (implementazione)

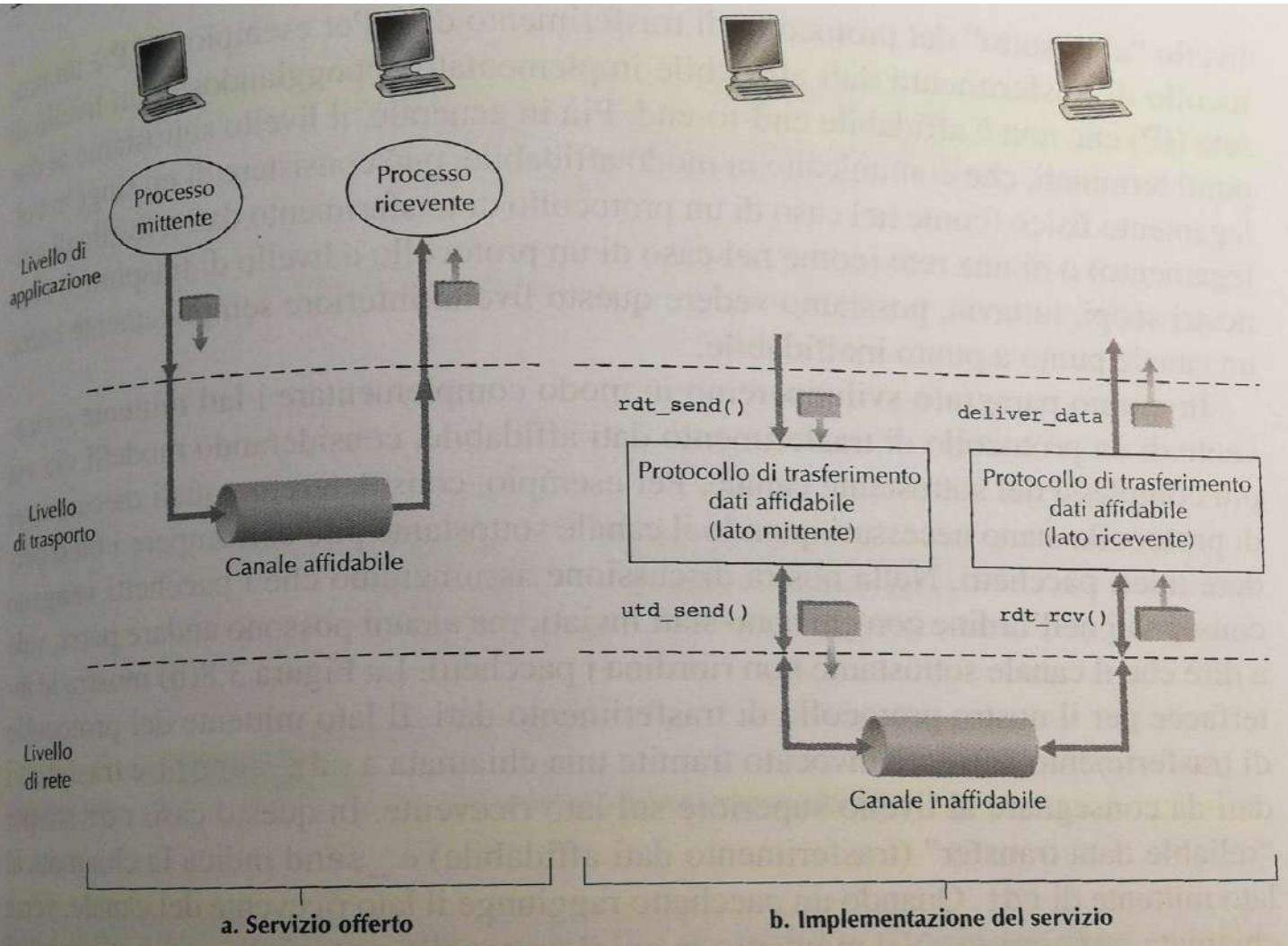
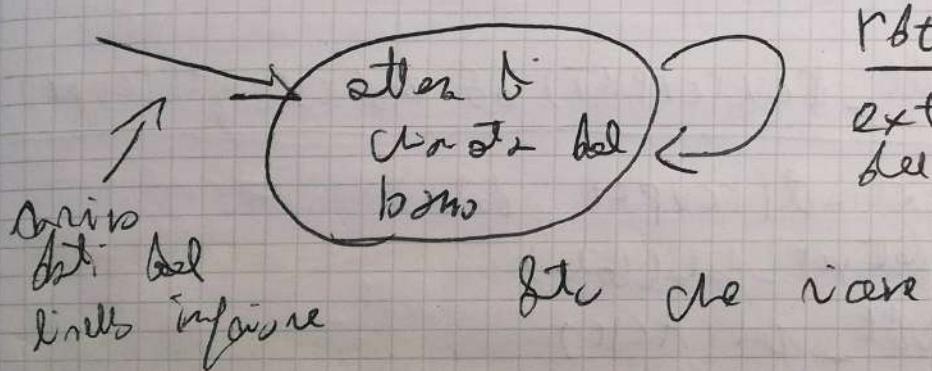
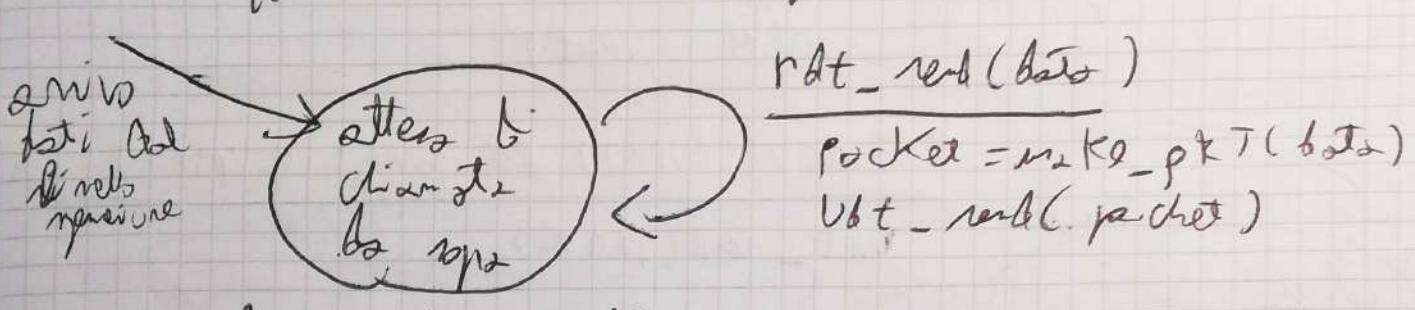


Figura 3.8 Trasferimento dati affidabile: modello di servizio e implementazione.



Il grande problema è mai nei pacchetti

= li abbiamo già visti

Quindi il receiver manda

ACK se il pacchetto è OK

NAK // // / e' male

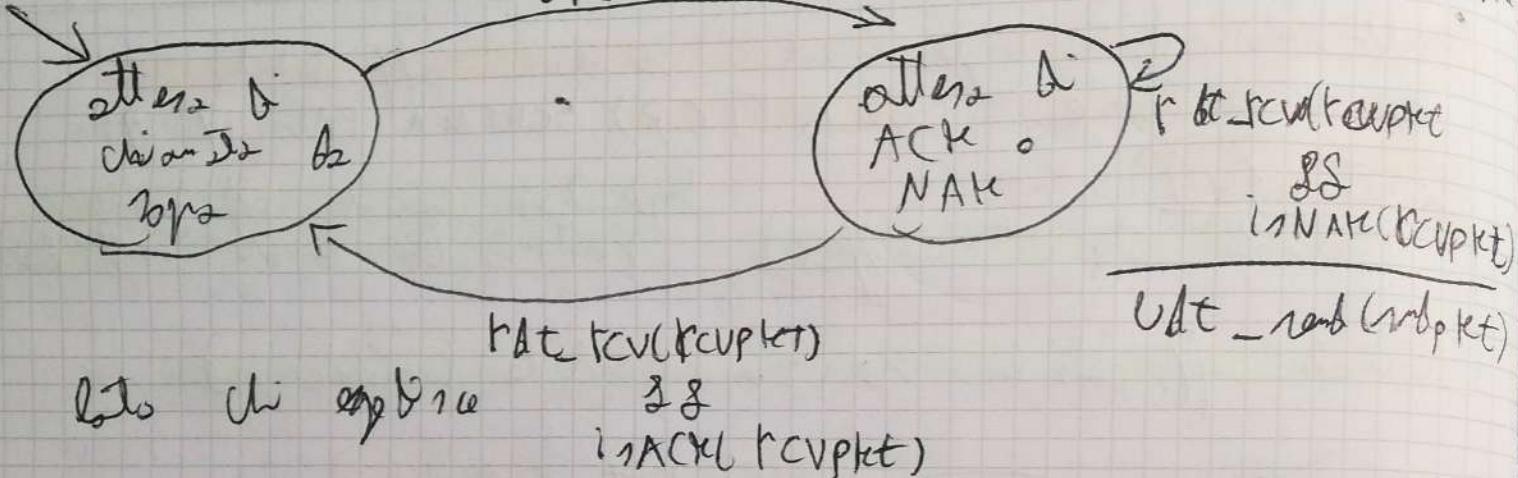
BDT 2.0 = Canale con errori ma senza perdite

I pacchetti arrivano regolare, ma possono essere errati
ma non mancano

Serve funzione per escludere o controllare
checksum

rat-read(data)

rat-read(subpkt) = rat-read(data, checksum) Udt-read(subpkt)



rat-tcv(rcvpkt) & convert(rcvpkt)

Udt-read(NAK)

rat-tcv(rcvpkt) & not convert(rcvpkt)

extract(rcvpkt, data)

deliver-data(data)

Udt-read(ACK)

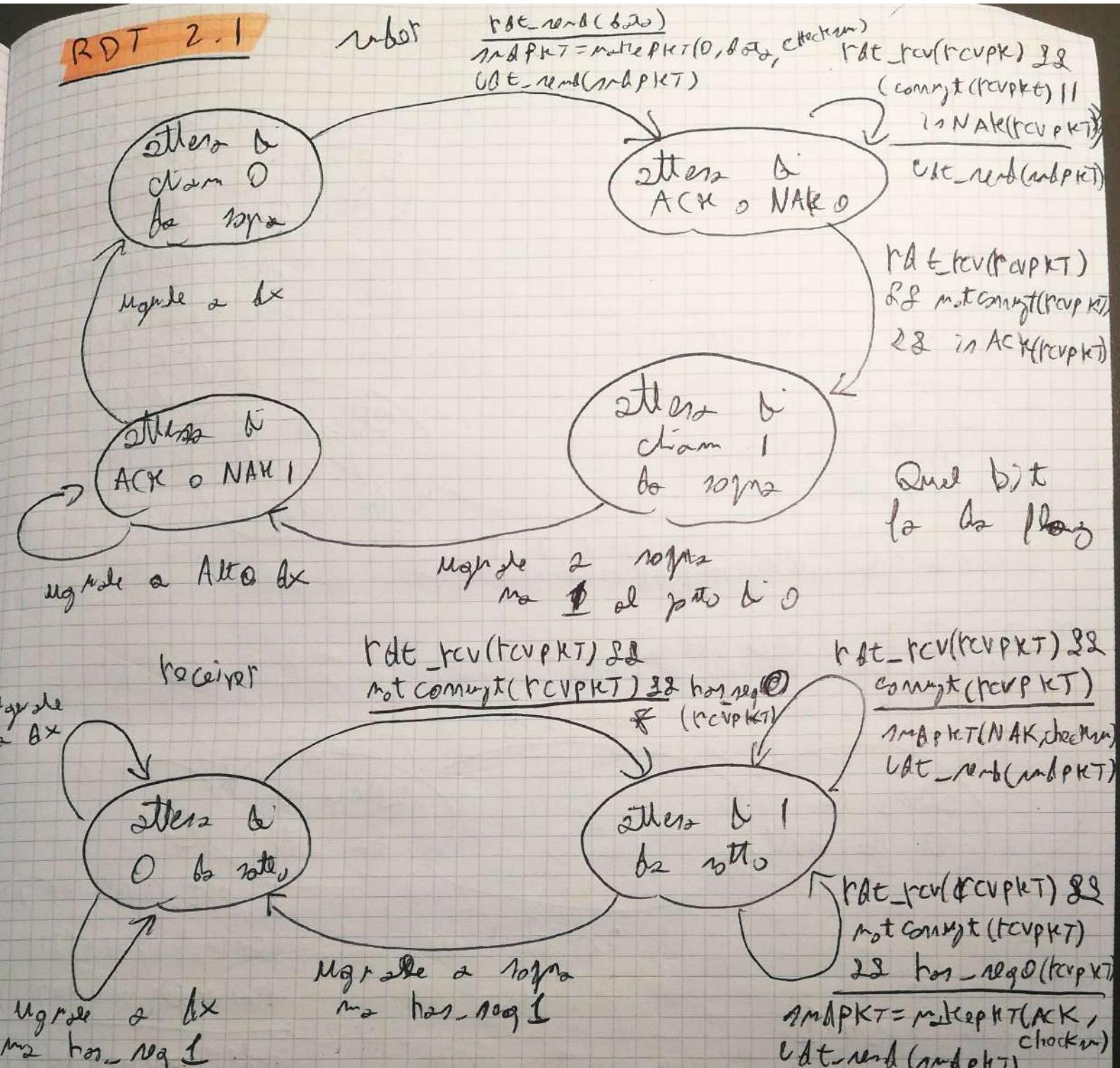
latenza di ricezione

Cosa succede se ACK e NAK sono inviati con
errore?

Il writer si blocca

Miglioriamo il protocollo ottimizzando ACK e NAK

RDT 2.1



Antico rischio alla duplicazione dei pacchetti: ACK e NAK
 I NAK possono essere eritati rispettando l'ACK
 Relativo all'ultimo pacchetto ricevuto. Sono comunque

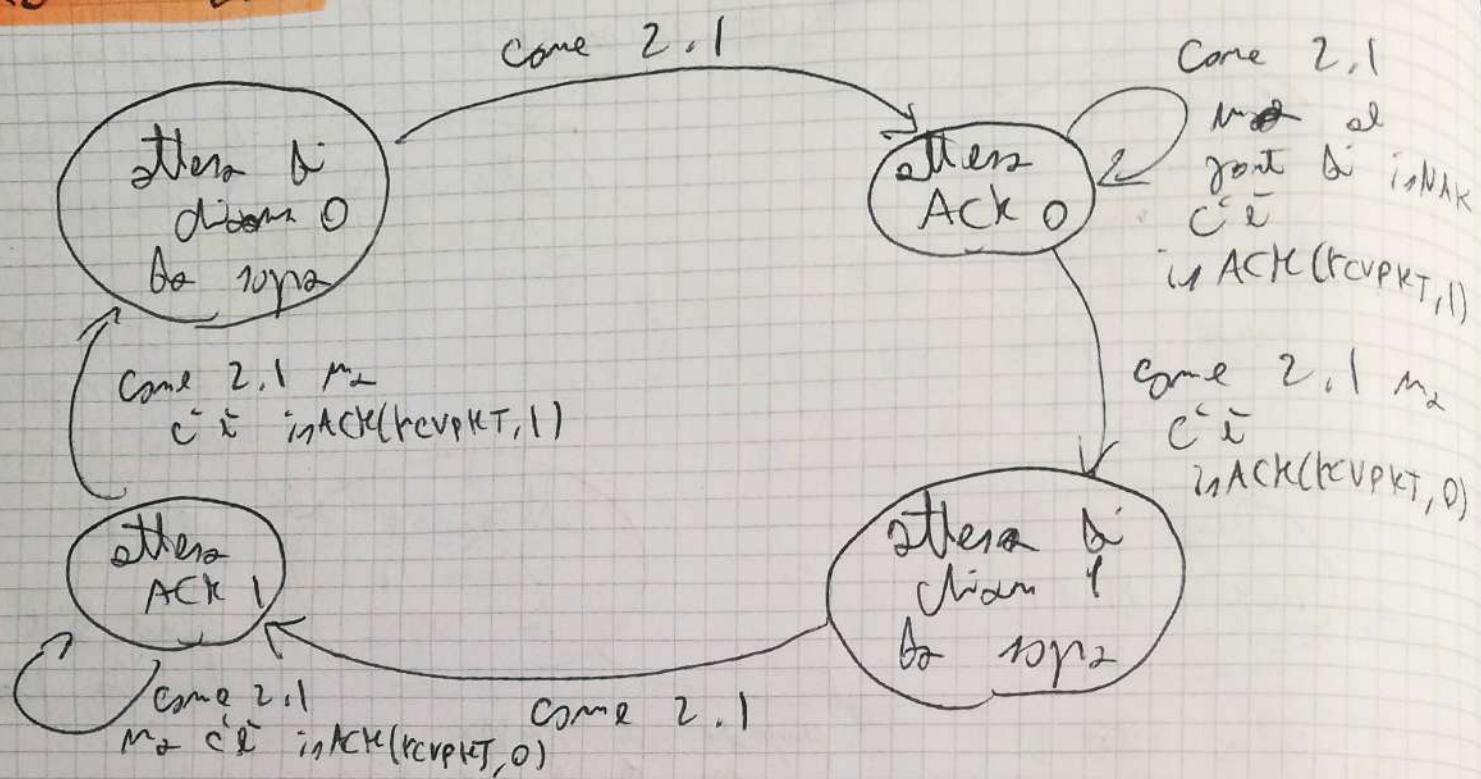
extraJ(rcvpkt, b₂₂)
 deliverB22(b₂₂)
 mdpkt=make_PKT(ACK, checksum)
 vdt_send(mdpkt)

O e 1 serve per
 distinguere tra pacchetti
 ricevuti e nuovi

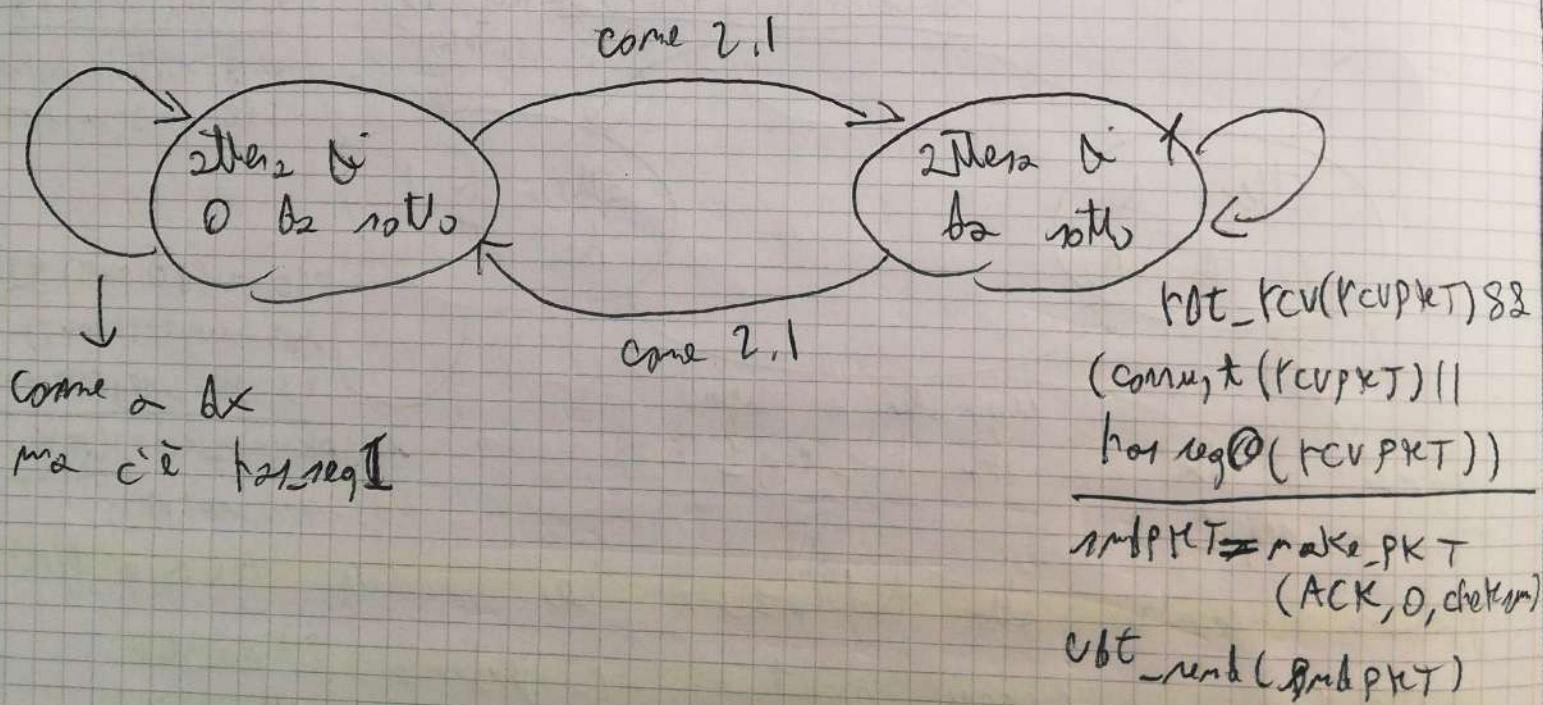
RDT 2.2

remember

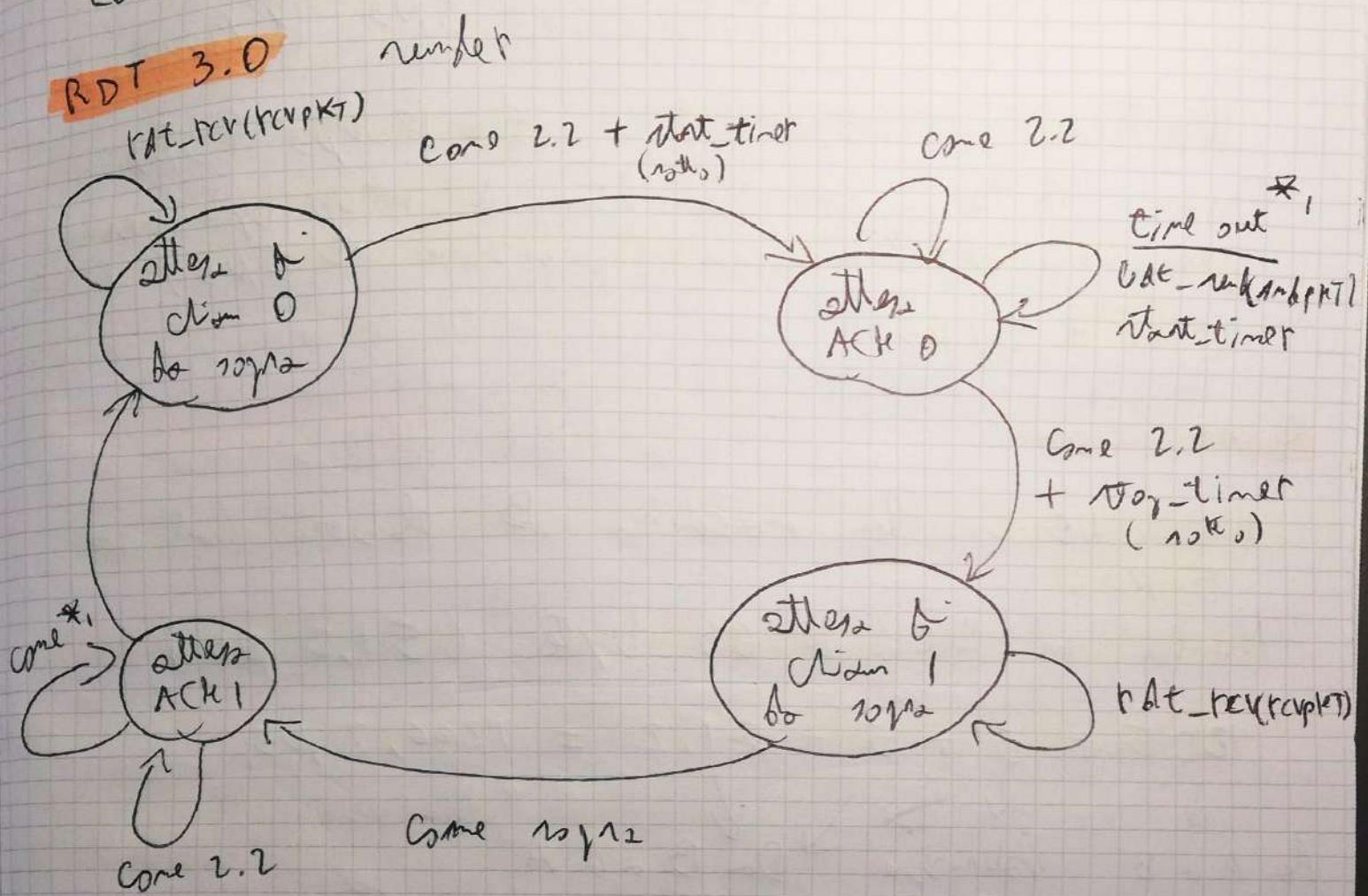
The function is NATE now or does it



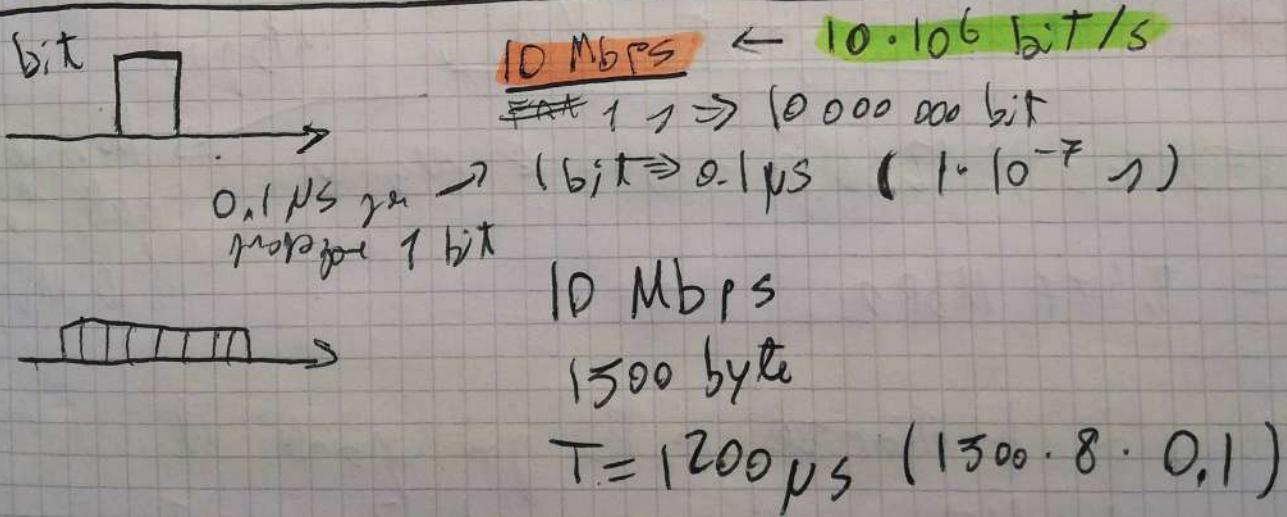
RDT 2.2 receiver



A DDM considera la probabilidad de que
paquetes da getine el timet
LOSSY CHANNEL WITH BIT ERRORS



If se devan eran numeros: por copiar la inputada
a QMALLIET ~~que~~ no comprendo



(A)

(B)

$$RTT = 10 \text{ ns}$$

1 Km

$$v = 200,000 \text{ km/s} \quad \checkmark$$

$$T_{\text{propagazione}} = 5 \text{ ns} \quad (1 / 200,000)$$

Il frame A è 1500 byte $t = 1200 \mu\text{s} + 5 \text{ ns}$

Ora il tempo per lo frame A 1500 byte per
percorrere un Km (con propagazione di 5 ns) è
(1200 + 5) ns

Connessione

1205 ns ... lo scambio B elabora dati in
5 ns.

Numero in ACK di 64 byte (51,2 ns)

$$1205 + 5 + 5 + 51,2 = 1266,2 \text{ ns}$$

↓ ↓ ↓ ↓
Da A a B elaborazione Da B a A

Dunque se avessi voluto fare un timer basedoc in
queste caratteristiche sarebbe dovrà essere più
di 1266,2 ns

Calcolare il throughput effettivo (MB/s)

$$\rightarrow 1500 \cdot 8 / 1266,2 \cdot 10^6 \leftarrow \text{result} \rightarrow \text{Numero di quadri inviati} \\ = 9,577 \text{ Mbps}$$

Metterebbero a distanza molto più grande il
throughput effettivo diminuire drasticamente

Questo protocollo è in cui trasmetti, invia e aspetti
l'ACK e fatto Stop and Wait

Il mittente non invia messaggi dati finché non è
certo che il destinatario abbia ricevuto correttamente
il pacchetto corrente

RTT = tempo ritardo di propagazione + arabo +
ritorno

R: tempo tra invio del codice

L: dimensione pacchetto

Tempo richiesto per trasmettere ~~un bit~~ il
pacchetto sul collegamento

$$d_t = \frac{L}{R} \quad (\text{a questo istante è messo l'ultimo bit})$$

1° bit giunge al destinatario in $t = RTT/2 + 0$

Ultimo bit giunge al destinatario in $t = RTT/2 + L/R$

L'ACK dell'ultimo bit giunge al mittente in

$$t = RTT/2 + L/R + RTT/2$$

Utilizzo del mittente:

$$\text{Utilizzo} = \frac{L/R}{RTT + L/R} = \frac{\text{Tempo in cui invia}}{\text{tempo totale}}$$

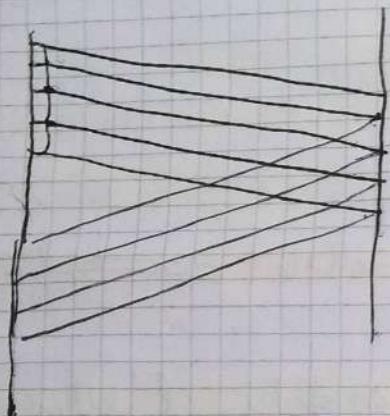
Notiamo che c'è una convenzione fatta che
nella rete che il mittente è stato già inviato nulla
tempo Totale.

Allora facciamo inviare al mittente più pacchetti
insieme belli per farla aspettare l'ACK

Giungiamo a un meccanismo di tipo Pipeline.

Non possiamo utilizzare il metodo di ACK 0 e i come in RDT 3.0 poiché ci sono già pacchetti e di conseguenza già ACK

Sender Receiver Ci sono l'ACK 0, 1 ...



Definiamo la WMA di un certo numero di frame.

Mettendoendo un numero di frame tip 10, ovvero il throughput effettivo (ma oltre il teorico)

Conseguenze del pipeline

- Intervalli numeri di seq. non finiti di 2
- Lato di invio e ricezione passa dovrà maximizzarne in un buffer già N in pacchetti
- Gestione errori / pacchetti persi  Go back N  Riconsegna relativa



N-finestra

- pacchetti con ACK
- // inviati senza ACK
- // disponibili da inviare
- // non disponibili

Visione solo n-th byte.

Anche lato ricevente non ha numero N
(bere sovrapposizioni, e qualche buco finestrato)

Protocollo Go back N

Per ogni pacchetto invio un timer, se non timer esce (non è arrivato o danneggiato) torna indietro
battendo i pacchetti ricevuti fino anche se corretti
Go back N = in caso di errore torna indietro di N pacchetti

Quindi se il mittente vede scadrere il timer del pacchetto 50, torna a 50 e ripete 50...51...52...

Il ricevente è a conoscenza di questa politica e invia l'ACK di conferma cumulativo, cioè che è arrivato l'ACK del 90 vuole dire che tutti i pacchetti fino al 90 sono confermati

Autori pag 209.

In modo approssimativo G-BN ^{risponde a 3 errori} risponde a 3 errori

- Invocazione dell'atto: Controlla finezza e in caso mille pacchetti.
- Ricezione ACK \rightarrow basta un ACK di conferma cumulativo
- timeout

Azioni GBN receiver

- Arriva pacchetto con numero $\geq N \rightarrow$ Monta ACK per quel pacchetto e manda pacchetto al link successivo
- Negli altri casi riceve il pacchetto e invia ACK per l'ultimo pacchetto corretto

Svantaggio: si eliminano molte volte pacchetti ricevuti correttamente perché tornano indietro di N pacchetti

Rigettazione selettiva

Per ogni pacchetto che ricevo mando un ACK relativo solo a quel pacchetto.

Il mittente se sta inviando il PRT 100 e il timer del PRT 90 scade, finisce di inviare il 100 e invia (nuovamente) il 90.

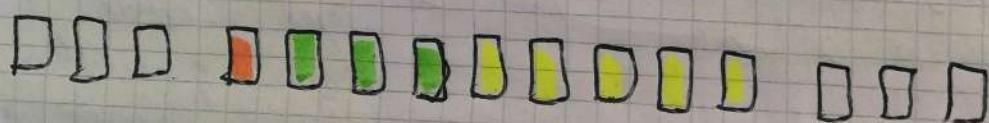
Approssimativamente 6s dopo back N che ritorna brevemente indietro a N anni e dopo tutto.

Qui si cerca di imbarcare le parole e rimuovere solo quella.

Non è un protocollo migliore di 6s back N, la rigettazione selettiva è utile se ci sono diversi pacchi e i vari, se invece su quel che permane si riuscisse a leggere sequenze di bit contigui 6s back N è più utile.



Sequenza vista dal reader



Sequenza vista dal receiver

- atteso
- arrivato, sincronizzato (prima ordine)
- accettabile
- non disponibile

finisce la ricezione e invia ~~getto~~ non comprende
perché magari il sender invia un pacchetto con
SWR ma l'ACK è verde.

Ciò accade perché il sender lo invia, invia di nuovo
il receiver ricalcamente lo invia.

Il receiver regola poi in altri modi che il
pacchetto è arrivato e scende questa ricezione.

Il receiver ne vede un pacchetto arrivare che
ovviamente non ha elaborato ma non ha
l'ACK, né l'errore (ogni m^a non si riferisce)

TCP

Si preoccupa di fare la flow control, multiplexing
e demultiplexing.

Fa inizializzazione, ripartizione e terminazione
della comunicazione

Fa trasferimento affidabile (con manipolazione e
ingaggiamenti)

Controlla qualità e condizioni, e flusso dati.

NON basta alla sicurezza, il TCP NON
è sicuro per nulla dall'applicativo

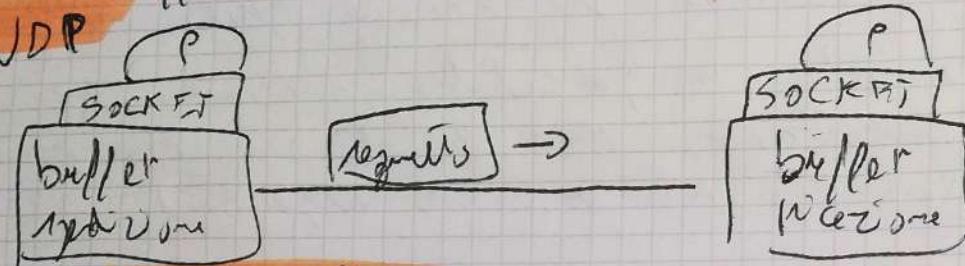
NON c'è garanzia sulla comunicazione

Caratteristiche

- Orientato allo connessione
- bidirezionale
- E' multiplex - symmetric (entrambe le parti)
- Applicabile
- Usa i messaggi ACK

- Stream - Orientato: il flusso trasmesso di dati è visto come unico, non a blocchi
- Dati NON strutturati
- Gestisce il flusso dati

TCP bufferizza dati prima di inviarli, non come UDP



Segmento:
Intestazione + dati
e opzioni

STRUTTURA

Il pacchetto ha un certo formato 20 byte + opzioni + dati

- 16 bit N. porta TCP
- 16 bit N. porta Dest
- 32 N. seg • 32 N. ACK
- 3 bit lung. intestazione • 3 bit Non usato
- 6 flag • 16 bit Finestra ricezione
- 16 che ck sum (per correggere errori) • 16 puntatore a dati segnati

Dopo questi 20 byte negliamo le opzioni e i dati

FLAG:	ACK: indica se l'ACK-field è valido URG: bit dati segnati PSH: bit di push RST: usato per resettare una connessione SYN: per creare una connessione (gradi 2 pacchetti) FIN: per chiudere una connessione
-------	--

Nel corso di ogni sessione si possono trasferire varie informazioni. Es: lunghezza relativa

Informazione: ~~il numero~~ indica il max
lunghezza del pacchetto (dimensione)

Dimensione

Header IP = 20 byte

Header TCP = 20 byte

(min) Dati TCP = 536 byte

TCP richiede che la trascinare pacchetti
di almeno 536 byte

MSS: dimensione pacchetto

Connessione

tramite un solo pacchetto

Problemi se cercano le tre mosse contemporaneamente
di aprire la connessione

Se per esempio arriva un pacchetto "reciso" e
il receiver invia l'ACK e il mittente chiude
la connessione.

Fibonacci: se c'è un errore blocca tutto (non c'è back-off)

Crescione: 1° pacchetto fa #1, il 2° #1 deve avere
errore parziale del 1°, quindi 3° deve
essere parziale del 2°.

Three-way Handshake per aprire la connessione
La creazione che offre entrambi i negozi, non fa
Chiusura della Connessione

comunque dev'essere
alla connessione

Più delicate della creazione

Non ci sono un protocollo norma per la chiusura
della connessione (ACK degli ACK degli ACK ...)

N.B.: Se è inviato il pacchetto con $\text{Seq} = 42$, $\text{ACK} = 42$
la risposta sarà $\text{Seq} = 43$, $\text{ACK} = 43$
ACK: ha il val. del primo byte stesso Pendativo

Sintesi (se ne parla una)

Normal flag FIN, segnala un ACK se non arriva più lo scrittore.

TCP prevede elenco di byte, A chiede verso B e B chiede verso A.
A volte può accadere che in coda c'è chiedere e l'altra no.

Telnet

Così che biglietti da client inviati al server e reinviati al client

Usa TCP, ha un rapporto 50:1 in interazione e file: rei

Il telnet produce pacchetti piccoli con spazio di banda

Il receiver può regolare il mittente di non inviare momentaneamente dati per non neccare il buffer con le variabili limitate di ricezione.

Quando si apre una connessione si può allocare memoria per fare il buffer.

Se questa base (richiesta di MacCabe rischia di allargare l'apertura della connessione) nessuno gli MacCabe
Dato che TCP usa piggybacking per non sprecare pacchetti respi e t.

Algoritmo di Nagle: fa bufferizzare se le

si stazza troppo tempo per non far saltare la rete

oppure riduce l'overhead bufferizzando dati.

N.B.

L'ACK identifica il byte successivo ottenuto, non

RTT basso: tanti RTT → \leftarrow bisogna fare RTT come spazio

RTT alto: bufferizzazione

* Grazie alla finestra di ricezione con celle
me misura nello stesso abbinato la
stanzione del buffer del ricevente
(esempio TCP full-duplex, ci sono due buffer)
sono fornite nel pacchetto informazioni necessarie
per far entrare da il mittente intesi il ricevente
In questo modo si dice che TCP ha un controllo
del flusso, e della congestione.

Finestra di ricezione strisciante & protocollo congestione

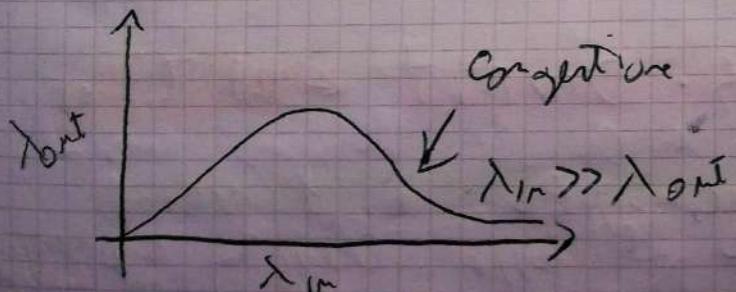
In cosa si risolve variazioni nel buffer (si libera
ma allo) si avvia il mittente che fa ripetere
consegnare i tanti pacchetti giusti che restano la
rete.

Soluzione di Clark: il ricevente non aggiorna
controllivamente la variabile, ma app che si
sono liberate già celle.

Con questo si invia sulla rete tantissimi
piccoli pacchetti

Congestione di rete | La rete ha quindi il traffico offerto
dalla rete e vicino o meno la capacità
di rete. Si creano code \rightarrow ritardi
buffer giornaliero di gettare le mitate dei pacchetti,
ma ritardano il ritratta.

Il ritardo causa timeout e negozio molto sojole
(e cos'è di sojole)



I pacchetti in rete
vengono finiti molto i
pacchetti per i
invati a livello
applicativo.

La congestione NON è soluzone della banda, ma
può negare completamente i buffer

L'obiettivo da seguire è raggiungere
completamente il canale e tenere gli utenti
possibile i buffer

La congestione è un problema della rete legato a
i buffer, che però c'è molto al alto livello.

Lo scatto del timer è fondamentale per abbattere
la congestione.

Mettere un ~~scatto~~ timer alto diminuirebbe la
congestione ma creerebbe altri problemi.

Un buon valore per il timer permette di
migliorare la congestione e funziona

TCP fa una stima di Round Trip per
finire in rete, basandosi sulla stima precedente.

$$\bullet \text{Estimated_RTT}_n = (1-\alpha) \cdot \text{Estimated_RTT}_{n-1} + \alpha \cdot \text{SingleRTT}_n$$

EWMA : Exponential weighted moving average

$$\alpha = 0,125$$

A volte la differenza tra gli stima e i reali è
troppo alta

Variabilità di Long RTT

$$\bullet \text{DevRTT}_n = (1-\beta) \cdot \text{DevRTT}_{n-1} + \beta \cdot |\text{SingleRTT}_n - \text{EstimatedRTT}_n|$$
$$\beta = 0,25$$

$$\bullet \text{BTO} = \text{EstimatedRTT} + \gamma \cdot \text{DevRTT} \leftarrow \text{Vel Timer}$$

Di solito il γ valore scelto è 1 secolo
Se ci tiene conto, timer raddoppia al doppio. Cresce la esperienza
del timer iniziale

Il timer è eseguito

TCP usa un timer per volta, non uno per pacchetto,
la ricezione di un ACK segnala l'arrivo del timer
per il segmento successivo (che non è ancora stato
confermato)

Per bloccare il rallentare la comunicazione si

Fast Retransmit (mette una pausa al timer eseguito)

Se arriva un segmento from above mondo un
ACK duplicato relativo all'ultimo pacchetto
arrivato in ordine.

Dopo 3 pacchetti fuisciti si rivede il
10° segmento non confermato. Dicendo RTT.

Se ho $RTO = 5NS$ e il 30° ~~pacchetto~~ ACK duplicato
arriva a 3NS, il timer è resettato a
quest'istante. (è stata forzata un timeout prima)
di ricevere gli dati deve problemare i pacchetti.

~~Other Condition: A B C D, A arrives so~~

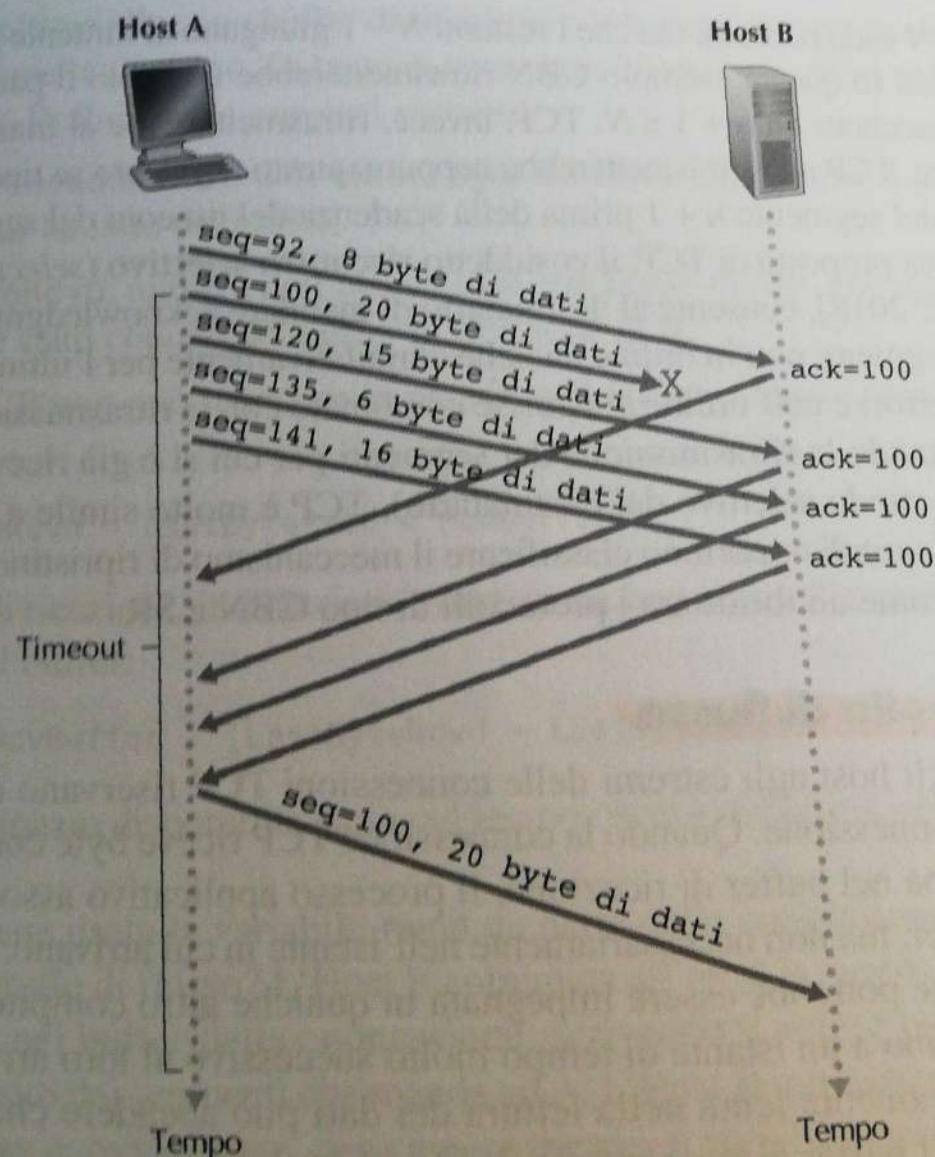


Figura 3.37 Ritrasmissione veloce: il segmento perduto viene ritrasmesso prima che il suo timer scada.

Nel grafico Nettivo 2

RTO
SAMPLE RTT
E-RTT
dev-RTT

RTO è la media sample RTT (in modo obiettivo)

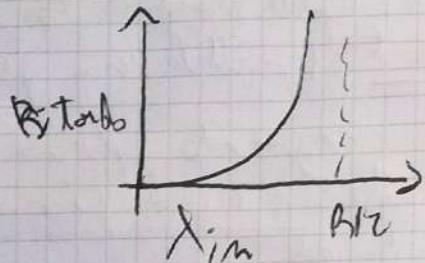
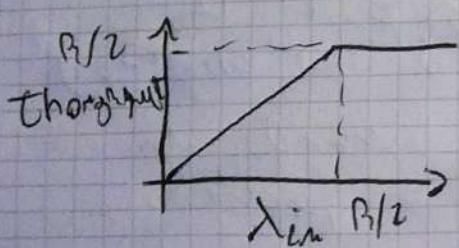
Sample e E-RTT più blu, alternativi

dev-RTT di solito ha valori bassi

Come è costituito il buffer

1) Due arrivati, router di mezzo con buffer illimitato

A e B comunicano, i pacchetti passano per un
solo router con capacità di mezzo infinito (buffer)



L'host può trarre il massimo $R/2$

(in realtà può usare oltre ma il throughput può usare oltre)

Poiché $R/2$? "Divisione" idealmente la connessione in 2; arrivati e ricevuti

Potrebbe sembrare utile avere λ_{in} vicino a $R/2$ visto che throughput sarà più vicino ai presentersi due pacchetti da inviare contemporaneamente

In questo scenario avremo lungi ritardi

2) Due arrivati, router di mezzo con buffer limitato

I pacchetti che vengono in un buffer sono accatastati.

λ_{in} : tassi d'arrivo verso socket (caus effetti)

λ_{in}' : tassi del livello di trasporto

Caso A) mithente per un messaggio binario inviato

che il buffer è libero

~~λ<λ_max~~

$$\lambda_{in}' = \lambda_{in}$$

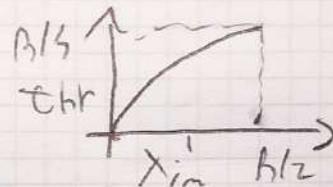
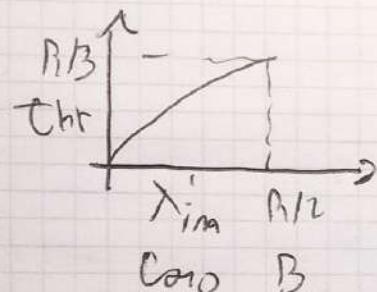
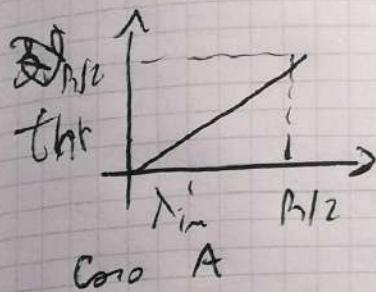
Questo è il caso ideale

Caso B) Mittente i dati sulle richieste
che il pochetto è pronto.
Le richieste vengono in più di nello

Caso C) Mittente potrebbe ritrasmettere un pochetto
anche se non è pronto per mettere giù

Per i lavori si vorrà che venga nel caso
ritrasmetta un pochetto non giù.

Ovvero Caso congetturali: trasmissione prenotata
e non richiesta



Caso C (più realistico)

3) Più intendi frontezi di mezzo con buffer
~~lavori~~ finiti

Arrivano una drastica diminuzione del throughput
fondata su una grande quantità di
lavori inviati, nella rete (nei buffer)

L'eliminazione dei pochetti nei buffer o ~~eliminazione~~
trasmissioni non necessarie fanno riacquistare
il throughput.

Infatti se in pochetto attraverso Z buffer e
nel tempo è ricavato, i primi Z hanno
per loro inutile

