

TRANSPORT LAYER

Multiplexing / Demultiplexing :

Risolviamo il seguente PROBLEMA:

al livello 3 (Network layer) si scombinano datagrammi tra hosts. Come scombinarli tra PROCESSI, anche su uno stesso host?

- Lo chiede sono 4 campi nell'header di un datagramma:

- 2 di livello 3 : IP sorgente e IP destinazione

- 2 di livello 4 : PORT NUMBER sorgente e PORT NUMBER destinazione

Questi campi si usano per identificare la SOCKET del processo giusto.

UDP: connectionless:

- Una socket è identificata una 2-tuple (IP address, port number)

- Datagrammi provenienti da host diversi o da processi diversi sullo stesso host, cioè con port # diverso, se hanno lo stesso port number destinazione vengono indirizzati sulla STESSA SOCKET (non c'è una connessione dedicata).

TCP: connection-oriented:

- Una socket è identificata da una 4-tuple:

- (source IP address, source PORT#, dest IP address, dest PORT#)

- Server possono gestire più socket per uno stesso processo, essendo una socket dedicata tra source e destination (connessione)

DEMULTIPLEXING: Indirizzare un segmento di livello 4 alle giuste socket una volta ricevuto.

MULTIPLEXING: Inserire gli header contenenti le informazioni del livello di trasporto e moltiplicare un datagramma di livello 4 nel livello di rete (3).

* VIOLAZIONE DELL'INDIPENDENZA TRA LIVELLI:

È vero che esiste, perché per fare mux/demux bisogna leggere informazioni di livello 3: gli indirizzi IP. Tuttavia, le cose non sono così perché originariamente i livelli 3 e 4 erano un unico livello!

UDP: User Datagram Protocol

E' un protocollo di trasporto che non aggiunge quasi nulla ai servizi offerti dal protocollo di rete IP; dunque, proprio come IP, è

BEST-EFFORT: fa del proprio meglio nel trasportare i pacchetti, ma NON garantisce né AFFIDABILITÀ, né ORDINE nell'arrivo dei pacchetti.

- Quelli servizi aggiunge ad IP?

- multiplexing / demultiplexing
- lieve controllo d'errore (checksum)

- UDP è **CONNECTIONLESS**: non richiede un handshaking preliminare e ogni segmento UDP è indipendente l'uno dall'altro.

VANTAGGI:

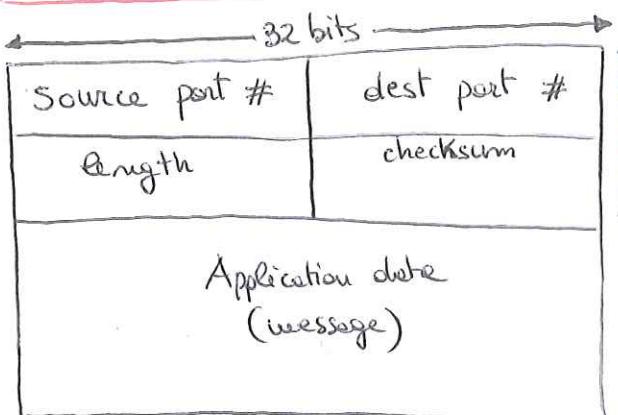
- 1) connectionless → riduce ritardo dovuto all'istaurazione di connessione, almeno 1 RTT.
- 2) no stato, semplice → non ha uno STATO delle connessioni da gestire, è molto semplice
- 3) header piccolo → riduce l'overhead: 8 bytes VS 20 bytes di TCP
- 4) no congestion control → non avendo controllo di congestione, permette l'invio di dati alla velocità e alla dimensione desiderate!

* Anche se non è affidabile, l'affidabilità può essere "implementata" a livello applicativo; lo fa ad esempio il **QUIC** (Quick UDP Internet Connection) protocollo, usato dal browser di Google Chrome.

Quando è utile UDP?

- 1) Per app multimediali e real-time: alla velocità e poco importa alla perdita di dati.
- 2) da uscire DNS: il poco overhead lo rende veloce.

• SEGMENTO UDP:



header

- header di 8 bytes: 4 campi, ognuno de 2 byte
- length: lunghezza del datagramme (header + dati)
- checksum: campo di controllo usato per l'error detection

• CHECKSUM:

E' un campo di 16 bit usato per rilevare gli errori (flipped bits).

Lo si costruisce nel seguente modo:

- 1) Si fa la somma a 16 bit dei dati da inviare; se c'è ripetizione, lo si somma, facendo wrapped sum
- 2) Si fa il complemento a 1 di quanto ottenuto. Quello che ne viene fuori è il campo CHECKSUM.

- Il receiver, calcola a sua volta il checksum e lo confronta con quello ricevuto:

- se sono diversi → So che c'è un errore, ma non sono in grado di correggerlo → **PACCHETTO SCARTATO**
- se sono uguali → Non sono sicuro che non ci siano errori! Se 2 bit nelle stesse colonne sono stati invertiti?

* Tuttavia, se le prob. di errore è piccole, quella di doppio errore è minima.
Ha comunque protezione dai singoli errori!

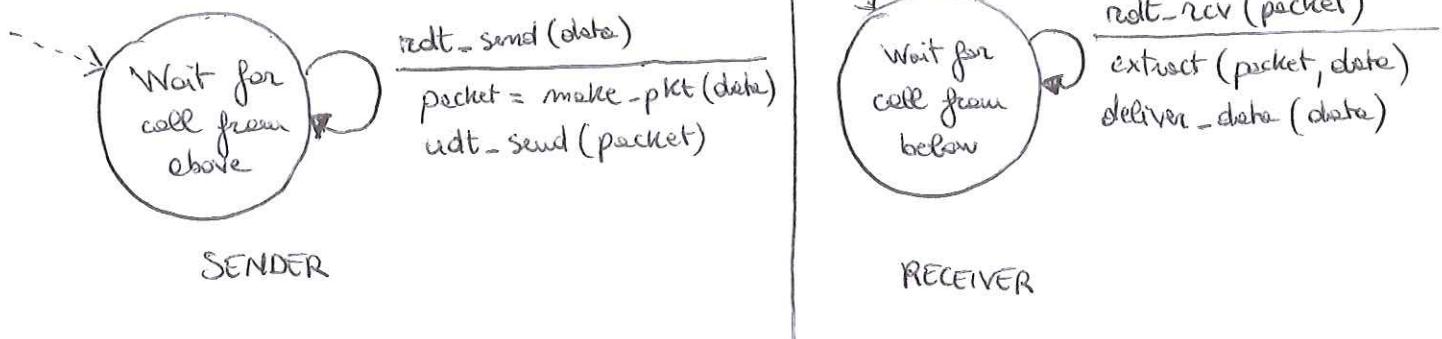
• PERCHÉ FARE ERROR DETECTION A LIVELLO 4 SE GIÀ VIENE FATTA A LIVELLO 2?

1. Alcuni link tra source e dest a livello 2 potrebbero non garantire error detection. anche se lo garantissero, sarebbe comunque bene farla a livello 4 in virtù del principio di indipendenza tra livelli.
2. E' possibile che alcuni bit errors vengano introdotti quando un segmento è conservato nella memoria di un router, in coda all'attesa.

PRINCIPLES OF RELIABLE DATA TRANSFER

rdt 1.0 → Canale affidabile :

Se il canale è affidabile (no errori, no perdite), NON bisogna fare NIENTE, se non pacchettizzare e depacchettizzare!



rdt 2.0 → Bit error channel :

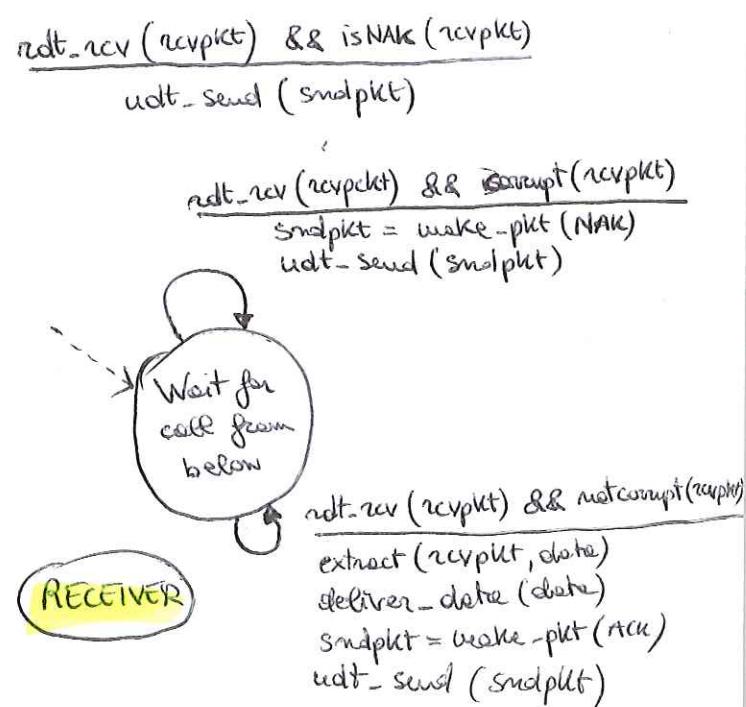
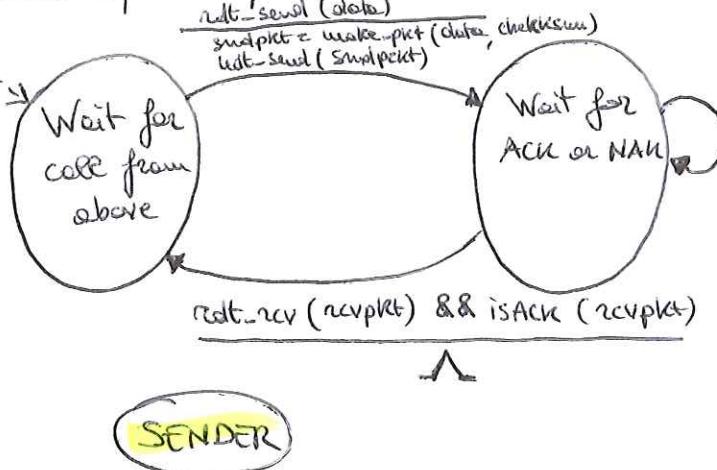
Bisogna capire quando c'è un errore e RECUPERARE dall'errore, ritrasmettendo il pacchetto. Come recuperare?

- ACKs: il ricevitore dice al mittente che il pacchetto ricevuto è ok
- NACKs: il pacchetto ricevuto aveva errori

Dunque, COSA BISOGNA AGGIUNGERE?

- 1) ERROR DETECTION → CHECKSUM
- 2) CONTROL MSG / RECEIVER FEEDBACK → ACK e NACK
- 3) RETRANSMISSION

Questi tipi di protocolli sono noti come ARQ (Automatic Repeat reQuest)



* Nota : Non posso inviare pacchetti fin quando non ricevo l'Ack!
Protocollo STOP & WAIT → L'effidabilità ha un COSTO IN PRESTAZIONI!

Problema: E se c'è l'Ack o il NAK vengono corrotti? Cose fare?

• rdt 2.1 : Sequence number → STOP & WAIT

SOLUZIONE : RITRASMETTO ; ma se ritrasmetto pacchetti già arrivati correttamente, come gestisco PACCHETTI DPLICATI ?

→ Aggiungo un SEQUENCE NUMBER : numero i pacchetti, così il receiver saprà se è un vecchio pacchetto o uno nuovo.

Essendo che inviamo 1 pacchetto per volta, basta 1 bit per numerare i pacchetti in questo protocollo.

Questo protocollo prende le nome di **STOP & WAIT**:

- 1) Il sender numeri i pacchetti con un Sequence Number (0 o 1);
- 2) Il sender, quando riceve ACK o NAK difettosi, RITRASMETTE il pacchetto;
- 3) Il receiver, se riceve pacchetti duplicati, li SCARTA, e rinvia l'Ack.

• rdt 2.2 : NAK fine :

Non si usano NAK, ma si inviano solo Ack:

- il receiver invia sempre l'Ack dell'ultimo pacchetto ricevuto correttamente;
- il sender, se riceve un ACK con sequence number diverso da quello atteso, allora RITRASMETTE! → Anche gli ACK vanno numerati!

Problema: Il protocollo funziona, ma su un canale che perde pacchetti NON funziona più!

• rdt 3.0 : Packet loss :

Se un pacchetto viene perso, seppiamo gestire la situazione → RETRANSMISSION

Ma il problema è: COME CAPIRE QUANDO C'E' PERDITA?

SOLUZIONE: Il sender aspetta un tempo "ragionevole" per l'Ack ; per ogni pacchetto viene fatto partire un **TIMER**

1. Invio pacchetto e start del timer;
2. Se il timer scade (interrupt), allora RITRASMETTO
3. Se ricevo ACK, stop del timer
- 3.1. Se ricevo ACK con seq # diverso da quello atteso, ASPETTO che scada il TIMER.

* Puoi introdurre pacchetti duplicati, ma rdt 2.2 sapeva già gestirli (scarto + ACK).

Limitazioni: molti 3.0 è ancora un protocollo STOP & WAIT ed è POCO EFFICIENTE su link ad alte velocità.

Era pensato per le prime reti, molto lente, su cui era molto efficiente.

• PROTOCOLLI PIPELINED:

Anziché inviare un pacchetto e aspettare l'ACK, si può permettere al sender di continuare ad inviare altri pacchetti.

PIPELINING:

Al sender è consentito di avere pacchetti multipli IN-FLIGHT (in volo), cioè di cui si deve ancora ricevere l'ACK

Ciò ha 2 importanti conseguenze:

- 1) Aumentare il range dei SEQUENCE NUMBER;
- 2) Effettuare BUFFERING al sender e al receiver

• Ci sono due forme principali di protocolli pipelined che sono:
Go-Back-N e Selective Repeat.

• Go-Back-N (GBN):

- Il sender può avere massimo N pacchetti in volo, con $N = \text{Window Size}$;
- Il receiver manda ACK CUMULATIVI: ACK(m) vuol dire "ho ricevuto TUTTO fino ed m compreso"; se riceve pacchetti fuori ordine può scartarli o bufferizzarli, ma risponde con l'ACK dell'ultimo pacchetto in ordine ricevuto correttamente.
- 1 Timer per il pacchetto in posizione send-bese, cioè per il più vecchio pacchetto inviato ma ancora unchecked. Se il timer scade, RITRASMETTE TUTTI i pacchetti inviati

* Nota: La "potenza" dell'ACK cumulativo è che spesso è in grado di evitare le ritrasmissioni quando si perdono gli ACK;
Se ACK(3) viene perso e ancora non scade il timeout, se ricevere ACK(4) non bisogna far nulla!

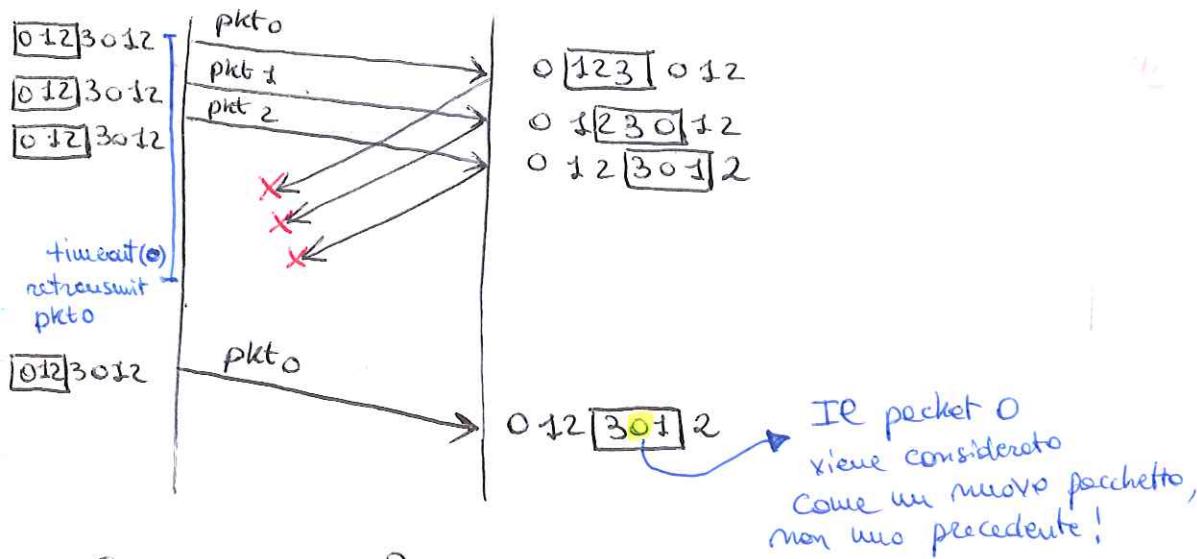
ACK(4) vuol dire che ha ricevuto tutto fino a 4!

• SELECTIVE REPEAT (SR):

- Massimo N pacchetti in volo;
- Il receiver invia ACK INDIVIDUALI, per pacchetto (segmento);
- Il sender ha 1 TIMER PER PACCHETTO e ritrasmette solo il pacchetto di cui scade il timer! → SELECTIVE
- * Ci sono 2 finestre! Una per il sender (W_{Tx}) e una per il receiver (W_{Rx}) e non devono per forza corrispondere!
- * Il receiver è OBBLIGATO a BUFFERIZZARE per gestire out-of-order packets, poiché deve ricostruire l'ordine giusto prima di consegnare i dati a livello superiore e fare SLIDING di W_{Rx} .
- * ACK NON CUMULATIVI: Se si perde l' $Ack(m)$, il sender non sapeva mai che il pacchetto m è arrivato, anche se arriva $Ack(m+1)$. Scaduto $timeout(m)$ e RITRASMETTERA', generando un pacchetto duplicato. Queste sono differenze con GoBackN.
 - Se il receiver dovesse ricevere un pacchetto m , ma con $m < receive_base$, cioè appartenente ad una VECCHIA W_{Rx} , invia comunque $Ack(m)$, per permettere al sender di fare SLIDING WINDOW di W_{Tx} in avanti.

• SR DILEMMA:

Una dimensione delle finestre troppo grande potrebbe portare a dei problemi, sorta di "frontieramenti":



• Come risolvere?

Invece:

$$N \leq \frac{\max\{seq\# \} + 1}{2}$$

Se i pacchetti hanno 4 diverse numerazioni: $(0, 1, 2, 3) \Rightarrow N \leq \frac{4}{2} = 2$

TCP

TCP = Transmission Control Protocol

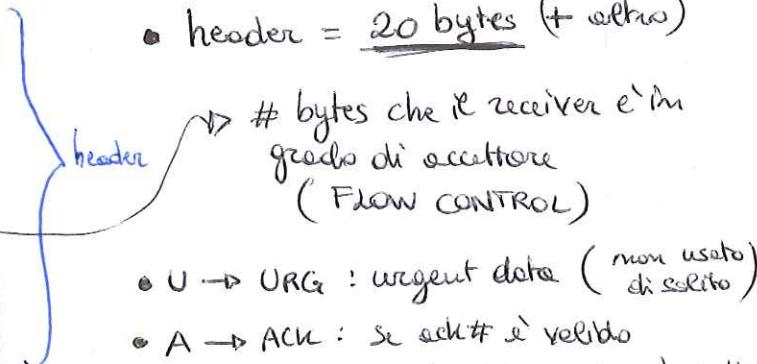
TCP è un protocollo:

- PUNTO - PUNTO: c'è 1 sender e 1 receiver; non supporta broadcast
- RELIABLE, IN ORDER BYTE-STREAM: non ci sono "boundary" di "messaggio"; non c'è il concetto di messaggio, ma si trasmettono Byte.
- PIPELINED: il controllo di flusso e il controllo di congestione di TCP **DIMENSIONANO OPPORTUNAMENTE LA FINESTRA (N)** per **2 scopi**:
 - 1) evitare la congestione del buffer del receiver
 - 2) evitare la congestione della rete
- SEND and RECEIVE BUFFERS: bufferizza i byte e li rende disponibili; è il livello applicativo ad encoderli e leggerli.
- FULL DUPLEX DATA: flusso di dati bidirezionale sulla stessa connessione; inoltre c'è il concetto di MSS (Maximum Segment Size)
→ occorre fare **SEGMENTAZIONE**
- CONNECTION - ORIENTED: c'è bisogno di un handshaking preliminare per instaurare una connessione (non fisica) prima di scambiare dati. Ciò comporta il concetto di **STATO** della connessione
- FLOW CONTROLLED: il sender non potrà mandare in overflow il buffer del receiver.

TCP SEGMENT STRUCTURE:

Source port #	dest port #
sequence number	
acknowledgement number	
header len	not used
U A P R S F	Receive Window
checksum	Urg date pointer
Options (Variable Length)	
Application data (Variable length)	
32 bits	

- header = 20 bytes (+ altri)



- U → URG: urgent data (non usato di solito)
- A → ACK: se seq# è valido
- P → PSH: push data now, incita la lettura e ev. applicativo
- R → RST: reset, condizione di errore
- S → SYN: richiesta di connessione
- F → FIN: chiusura di connessione

• SEQUENCE NUMBER :

Contiene il numero di sequenze associato al 1° Byte nel segmento dati; il numero di sequenze iniziale è scelto random.

• ACK :

TCP usa **ACK CUMULATIVI**; tuttavia, l'ACK NUMBER indica il sequence number del prossimo Byte atteso:

ACK(m):

"ho ricevuto tutto fino ad ($m-1$); mi aspetto (m)!"

• OUT-OF-ORDER :

Lo standard non specifica come il receiver deve gestire segmenti che arrivano fuori ordine, ma tipicamente li **BUFFERIZZA**!

• TCP RELIABLE DATA TRANSFER :

TCP offre un servizio di trasporto affidabile sul servizio inaffidabile di IP. Use ACK cumulativi ed è un protocollo pipelined. Dovrebbe usare 1 timer, ma dipende, spesso è 1 per segmento.

Le ritrasmissioni sono generate da:

1) TIMEOUT

2) ACK DUPLICATI, secondo il meccanismo del FAST RETRANSMIT.

• TCP SENDER :

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever)

switch (event)

event: data received from application above

create TCP segment with seq# NextSeqNum

if (timer currently not running)

start timer

pass segment to IP

NextSeqNum = NextSeqNum + Length (data)

break

event: timer timeout

retransmit not-yet-acked segment with smallest seq#

start timer

break

event: Ack received, with ACK field value of y

if ($y > \text{SendBase}$) {

SendBase = y ; if (there are currently any not yet acked segment)

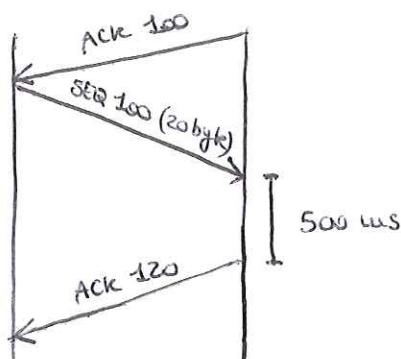
start timer

break

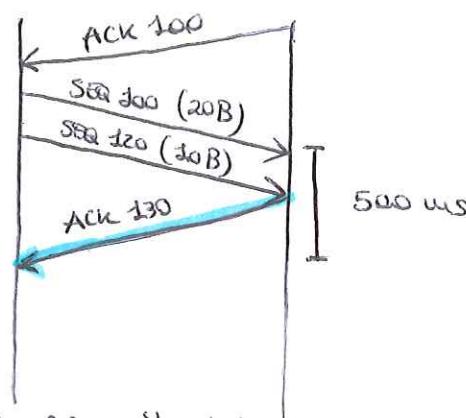
MECCANISMO DELL'ACK RITARDATO:

Immemoritutto, il valore del timer NON è costante, va in base al Time Out Interval e dipende dal controllo di flusso e congestione.

- Tipicamente, il receiver TCP sfrutta gli ACK cumulativi e invia 1 ACK ogni 2 SEGMENTI. Tuttavia, se il sender non invia un altro segmento entro un tot di tempo (ALTRO TIMER) il receiver invierà comunque l'ACK, per evitare le ritrasmissioni. Questo è detto meccanismo dell'ACK RITARDATO.
- * Tipicamente, da standard, il timer di attesa del receiver è di 500 ms



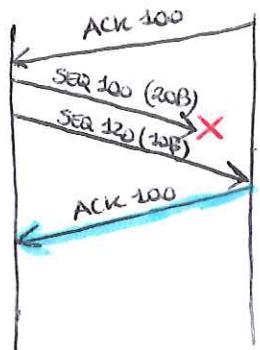
(a) attesa del timer



(b) ACK ritardato

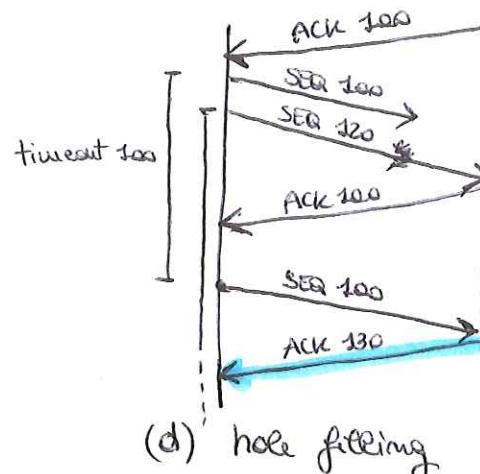
- Si cerca di minimizzare il numero di ACK da inviare; inviare meno traffico è sempre positivo.

E PER SEGMENTI FUORI ORDINE?



(c) ACK duplicato

↓
Invia l'ACK di ciò che si aspetta,
ma lo **IN VIA SUBITO**! L'ACK
non è ritardato!



(d) hole filling

↓
Se il buco creato dall'out-of-order viene riempito, anche pericolmente,
invia sempre l'ACK di ciò che si
aspetta e lo **HANDA SUBITO**!

• FAST RETRANSMIT:

Siccome la ritrasmissione deve attendere lo scadere del timer per essere effettuata, timeout lunghi possono introdurre grossi ritardi.

- Si può capire che qualcosa sia andato storto dalla ricezione di ACK DUPLICATI. Se un segmento si è perso, dovrà fuggire a molti ACK duplicati.

TCP adotta il seguente meccanismo per la ritrasmissione in aggiunta ai timeout event!

FAST RETRANSMIT:

Se il sender riceve **3 ACK DUPLICATI** ($1 + 3$ duplicati) per lo stesso segmento, assume che il segmento è stato perso e lo **RITRASMETTE SUBITO**, senza attendere che scade il timer!

event: ACK received, with ACK field value of y

if ($y > \text{SendBase}$) {

 SendBase = y

 if (there are any not yet acked segment)

 start timer

}

else {

 increment count of dup ACKs received for y

 if (count ~~received~~ of dup ACKs received for $y == 3$)

 resend segment with seq# y

}

break

• COME SETTARE IL TIMER IN TCP?

Sicuramente, deve essere più grande dell'RTT, ma l'RTT è variabile, anche all'interno delle stesse connessioni.

Inoltre, se il timer è troppo piccolo, si ha TIMEOUT PREMATURO e ritrasmissioni non necessarie. Se troppo grande, scorse reattività alle perdite di segmenti.

Ci si sposta al seguente problema: **COME STIMARE IL RTT?**

- Sample RTT: campionamento dell'RTT ogni volta che invia un pacchetto e ne riceve l'ACK, calcolando il tempo trascorso. Vengono ignorati pacchetti ritrasmessi

Usando questi campionamenti, si fa una media pesata, dando più valore ai campionamenti recenti; si usa un **FILTRO PASSA BASSO del 1° ORDINE!**

$$\text{Estimated RTT} = (1-\alpha) * \text{Estimated RTT} + \alpha * \text{Sample RTT}$$

- Tipicamente: $\alpha = \frac{1}{8} = 0,125 \rightarrow \text{Shift right di 3}$, poco costoso per un processore!
- Media pesata che si muove esponenzialmente
- L'influenza dei campionamenti passati decresce esponenzialmente veloce.
- Essendo che l'RTT cambia di molto anche velocemente, c'è bisogno di stimare anche di quanto il Sample RTT VARIA dell'Estimated RTT, così da aggiungere un "margin" al timeout.

Per farlo, si calcola la **DEVIAZIONE STANDARD** (σ) con la seguente formula approssimata, anch'essa un filtro pesato basso:

$$\text{Dev RTT} = (1-\beta) * \text{Dev RTT} + \beta * |\text{Sample RTT} - \text{Estimated RTT}|$$

- Tipicamente: $\beta = \frac{1}{4} = 0,25 \rightarrow \text{Shift right di 2}$

Ora, siamo in grado di definire un valore per il timeout:

TIMEOUT INTERVAL

$$\text{TimeOut Interval} = \text{Estimated RTT} + 4 * \text{Dev RTT}$$

- INIZIALIZZAZIONE: $\text{RTT} = \emptyset \text{ sec}$
 $\text{Timeout Interval} = 3 \text{ sec} \rightarrow \text{Dev RTT} = \frac{3}{4} = 0,75 \text{ sec.}$

- * La cosa importante da notare è che rende tale aspetto del protocollo interessante è che il **TIMER E' DINAMICO**! Cambia in continuazione ed è calcolato segmento per segmento!

Dipende dall'RTT, quindi dal traffico / delay della rete \rightarrow È legato alla congestione e al controllo di congestione!

Se la rete è congestionata, ormai ritardati, ma allunga dinamicamente il timer per evitare ritrasmissioni "precoci" che contribuirebbero solo ad aumentare la congestione e quindi il packet loss, portando potenzialmente al collasso!

TCP Flow Control

PROBLEMA: I buffer sono di dimensione finita. L'applicazione potrebbe leggere dati dal buffer delle socket TCP ad una frequenza minore delle frequenze di invio di segmenti del sender; dunque il buffer del receiver potrebbe endere in overflow e verrebbero scartati immediatamente dei segmenti.

FLOW CONTROL

Il receiver controlla il sender, così che il sender non manderebbe in overflow il buffer del receiver trasmettendo dati troppo e/o troppo velocemente.

- NOTA: In UDP il problema non si pone e i pacchetti vengono SCARTATI, poiché non devo garantire affidabilità.

COME SI FA?

→ Il receiver avverte il sender di quanto spazio libero ha nel buffer delle socket TCP, includendo il valore rwmwl (receive window) nel campo opposto "Receive Window" dell' HEADER dei segmenti TCP.

- * NOTA: Come già detto, il controllo di flusso è strettamente legato al concetto di AFFIDABILITÀ; non posso permettermi di buttare pacchetti causando retransmissioni inutili. Conviene controllare il flusso di dati ad una "frequenza" gestibile sia da sender che receiver.

TCP CONNECTION MANAGEMENT

TCP è un protocollo CONNECTION-ORIENTED, quindi deve prevedere un meccanismo di instaurazione e chiusura di connessione.

- Dunque, prima di trasmettere dati, client e server fanno un HANDSHAKE in cui inizializzano e comunicano variabili: taglie dei buffer, RcvWindow per il controllo di flusso e i Seq #s, scelti RANDOM, per evitare orecchi!

2-WAY HANDSHAKE:

È un meccanismo che prevede l'invio di 1 messaggio di richiesta di connessione + 1 messaggio di accettazione da parte del server.

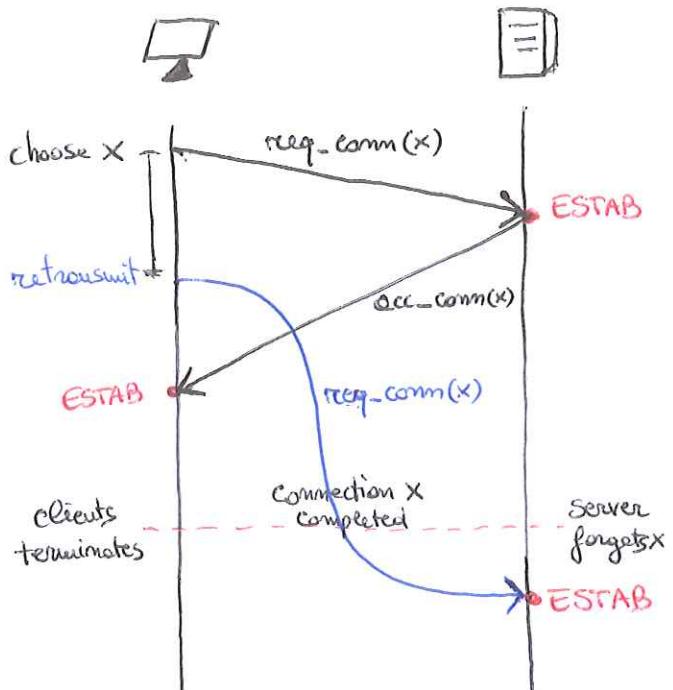
Ma funziona sempre bene?

NO



- ritardi variabili
- messaggi ritrasmessi, dovuti a perdite
- out-of-order
- "non si puo' vedere" l'altro lato

Vediamo 2 esempi di casi in cui il 2-way handshake FALLISCE:



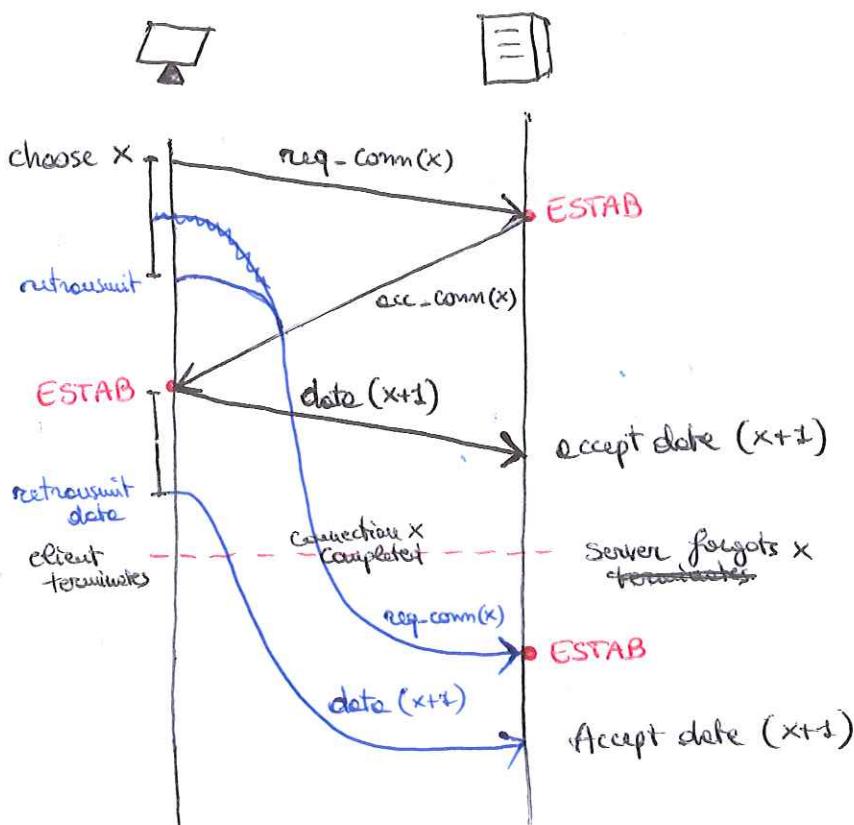
(1) HALF OPEN CONNECTION!
(ma client)

- Le connessioni da parte del server rimane aperte, ma non c'è più la contro parte client!

→ Spreco di risorse da parte del Server

3 Vulnerabilità ed attacchi di tipo
DENIAL OF SERVICE (DOS)

Vediamo il seguente scenario ancora più grave!



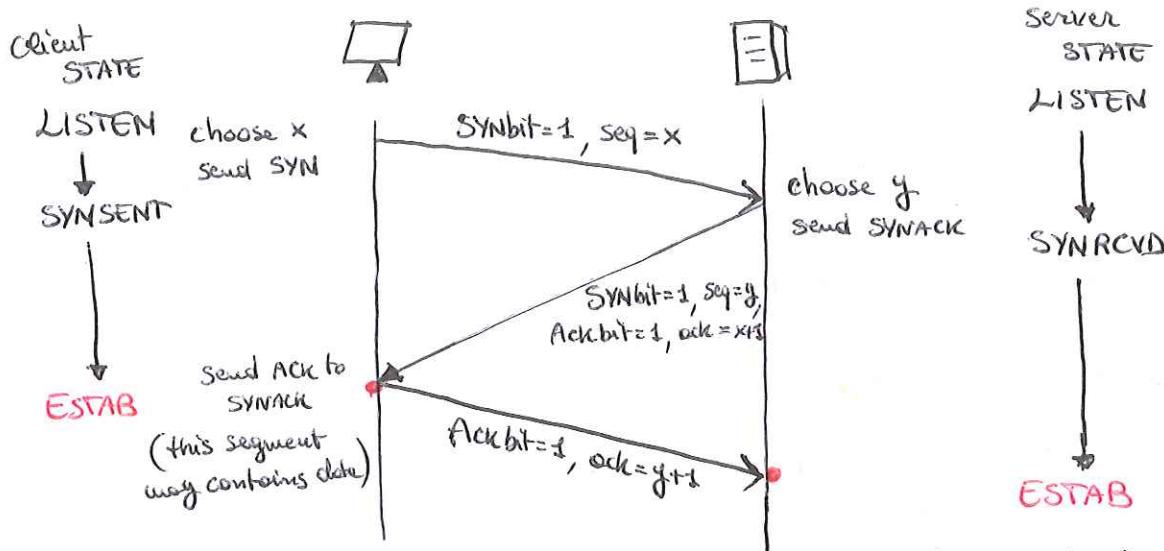
(2) HALF OPEN CONNECTION
WITH DATA PROCESSED
TWICE!

- Oltre a consumare risorse per tenere aperte le connessioni, il server spreca ulteriori risorse e potenze di calcolo per processare i dati 2 volte, quando il client potrebbe già non essere più in rete!

* TCP utilizza un 3-WAY HANDSHAKE!

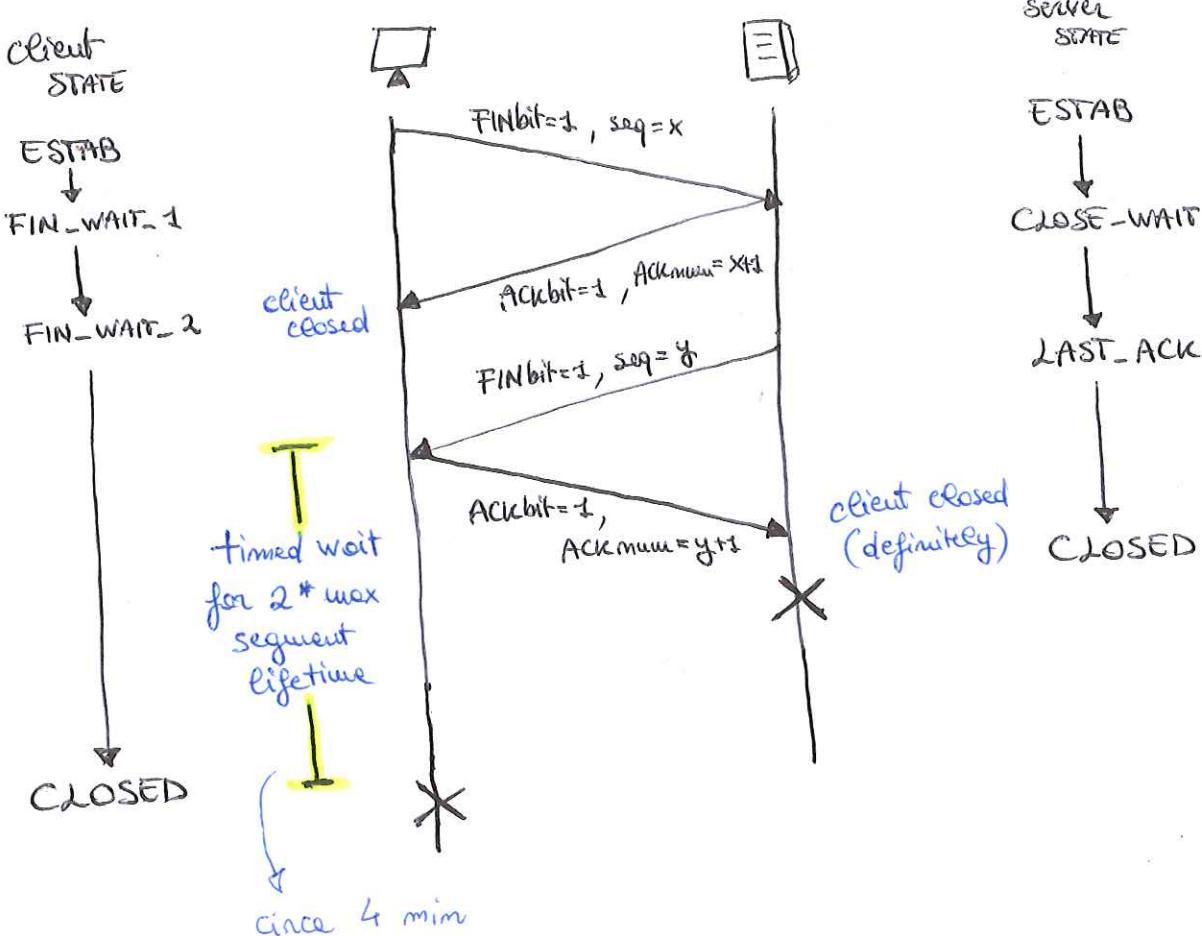
• 3-WAY HANDSHAKE :

- (1) Il client invia un **SYN** segment con **SYNbit=1** e **seq.# = x**, scelto casualmente;
- (2) Il server risponde con un **SYNACK** segment: **SYNbit=1**, **seq.# = y**, **ACKbit=1**, **ACK# = x+1**. Connessione stabilita per il client.
- (3) Il client invia un **ACK** segment al server: **ACKbit=1**, **ACK# = y+1**. Connessione stabilita al server.



* Il 3° segmento di ACK, Segnala al server che il client è VIVO !

• CHIUSURA DI CONNESSIONE :



- Se il client che il server inviano FIN segment per chiudere ENTRAMBI la connessione, e devono ricevere l'ACK.

Perché il client aspetta un timer?

Chi richiede la chiusura di connessione per primo, dopo aver ricevuto il FIN della controparte deve restare in attesa per un tempo pari a $2 * \text{max segment lifetime}$ (circa 4 minuti) perché SE L'ACK DEL FIN SI PERDESSE DOVREBBE RITRASMETTERLO! (il server rimanderebbe il FIN)

* Nel caso in cui fosse il server a chiudere per primo (tipico scenario in HTTP), dovrebbe attendere potenzialmente per 4 minuti!
 → Enorme spreco di risorse! ⇒ Spesso si riduce o si taglia!

• PRINCIPI DI CONTROLLO DI CONGESTIONE:

CONGESTIONE (informalmente):

"Troppe sorgenti inviano troppi dati troppo velocemente per essere gestiti dalla rete. C'è troppo traffico nella rete"

Come si manifesta? → pacchetti persi: buffer overflow ai router
 cumpi ritardi: code nei buffer dei router

Ci sono 2 possibili approcci:

1) END-END CONGESTION CONTROL:

- La rete NON dà un feedback esplicito della congestione agli end systems;
- La congestione è PERCEPITA dagli end systems osservando ritardi e perdite di segmenti (ATTENZIONE alle perdite frequenti dovute a rumori in reti wireless)
- approccio adottato da TCP

2) NETWORK ASSISTED CONGESTION CONTROL:

- i router forniscono un FEEDBACK esplicito agli end systems; possono farlo con 1 bit indicante la congestione (SNA, Debit, TCP/IP ECN, ATM) oppure indicando il rate esplicito che l'end system deve adottare.

• ATM: Controllo di congestione ABR (Available Bit Rate):

"ELASTIC SERVICE":

- se il percorso del sender è "scorico" → dovrebbe usare TUTTA le larghezze di banda
- se il percorso è congestionato → dovrebbe essere "strozzato" al MINIMO RATE GARANTITO!

* TCP/IP non ha il concetto di "minimo garantito", ma è molto più fair, tentando di accontentare tutti; qui invece si può rifiutare qualcuno.

- La rete dell'ATM usa dei pacchetti particolari, detti RM (Resource Management) CELLS insieme alle altre celle:

- questi pacchetti vengono settati dagli SWITCH (\leftrightarrow router) con informazioni sulle congestioni → **NETWORK ASSISTED**
- le RM CELLS vengono inviate al sender, per controllare la rete ed applicare controllo di congestione.

- Si basa sui seguenti meccanismi e bit:

1) EFCI (Explicit Forward Congestion Indication):

un bit presente in tutte le celle e settato a 1 se switch congestionato.

Se al receiver arrivano celle con EFCI=1 \Rightarrow il bit CI delle RM cells si impostava ad 1 prima di rinviarle indietro.

2) CI/NI:

Sono 2 bit delle RM cells:

- CI (Congestion Indicator)
- NI (No increase) → dice al sender di non aumentare il rate

3) ER field:

Campo di 2 byte presente nelle RM cells che indica il rate esplicito a cui bisogna trasmettere;

- gli switch, se congestionati, potranno diminuire l'ER nelle celle
- garantisce al sender la massima velocità supportabile dalla rete! (sue path).

TCP CONGESTION CONTROL

- E' di tipo END-END CONTROL (no network assistance)
- Il sender limita le trasmissioni, facendo in modo che valga sempre la seguente diseguaglianza:

$$\text{LostByteSent} - \text{LostByteAcked} \leq \min \{ \text{CongWin}, \text{RecvWindow} \}$$

RecvWindow è chiaramente legato al Controllo di Flusso; supponiamo che sia infinito.

- CongWin cosa è e come si controlla?
E' la finestra di trasmissione, e, essendo che mi aspetto di trasmettere tutti i pacchetti nelle finestre e poi iniziare a ricevere ACK, posso fare le seguenti approssimazioni, che lega CongWin al Rate:

$$\text{Rate} \approx \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- Ma il punto qual è?

CongWin è DINAMICA; è funzione della CONGESTIONE PERCEPITA della rete.

- IL SENDER COME PERCEPISCE LA CONGESTIONE?

Si basa sui LOSS EVENT che si verificano quando:

- scade un timeout
- arrivano 3 ACK duplicati

→ Il sender TCP riduce il rate dopo un loss event, ma riduce il RATE INDIRETTAMENTE, riducendo le CongWin.

Il meccanismo di congestion control in TCP, si può suddividere in 3 fasi:

- 1) Slow Start
- 2) AIMD (Additive Increase, Multiplicative Decrease)
- 3) conservative after timeout events

• TCP Slow Start:

- Quando inizia la connessione : CongWim = 1 MSS
- con $1 \text{ MSS} = 1500 \text{ Byte}$ tipicamente.
* All'inizio si comporta come un protocollo Stop & Wait
- Ogni volta che ricevo un ACK, aumento le CongWim di 1 MSS
 \Rightarrow quasi ogni RTT, le finestre RADDOPPIA \rightarrow cresce esponenzialmente!

Perché partire lento?

Quando un nuovo utente si aggiunge alla rete, trasmettendo molto lentamente non perturba la congestione in modo significativo e non impatta sui servizi degli altri utenti \rightarrow Slow Start

- Il rate incrementa esponenzialmente fino al 1° loss event.

• TCP AIMD:

AIMD = Additive Increase, Multiplicative Decrease

MULTIPLICATIVE DECREASE :

Quando avviene un loss event \rightarrow DIMINUO le CongWim

ADDITIONAL INCREASE :

Dopo un certo punto, le finestre non crescono più esponenzialmente, bensì CongWim aumenta di 1 MSS OGNI RTT in presenza di perdite (ad ogni ACK aumenta di $1 \text{ MSS} * \frac{1 \text{ MSS}}{\text{CongWim}}$)

\hookrightarrow PROBING della rete: prova ad aumentare, non sapendo quante sono le bande totali e quelle disponibile

• Raffinamento: INFERRING LOSS

Ho 2 comportamenti differenti a seconda del tipo di loss event:

(1) Dopo 3 ACK duplicati \rightarrow - TCP DIMINUO CongWim
- poi incrementa linearmente

(2) Dopo timeout \rightarrow - CongWim diventa pari a 1 MSS
- Ricomincia delle fasi di Slow Start
- Le finestre crescono esponenzialmente

{ Comincia DA CAPO!

• PERCHE' FA QUESTO?

- Ricevere 3 ACK duplicati indica che 1 segmento è stato perso, ma anche 3 segmenti successivi sono arrivati!
Non è indice di una congestione altissima.
- Se scatta il timeout, vuol dire che il segmento è stato perso, ma sono stati persi anche i successivi! → **CONGESTIONE GRAVE**
→ TCP si comporta in modo CONSERVATIVO e impone le CongWin = 1 MSS

* **ATTENZIONE:** Se sono all'inizio, e cause delle dimensione piccole delle finestre, in caso di perdite potrei non essere in grado di ricevere 3 ACK DUPLICATI e perderei molto tempo, essendo che il timer all'inizio è ancora grande.

• Raffinamento: THRESHOLD

Quando l'incremento dovrebbe cambiare da esponenziale a lineare?
Perché aspettare che venga perso qualcosa?

→ SI PASSA DA EXP A LINEARE:

quando CongWin diventa $\frac{1}{2}$ del valore che CongWin avrebbe prima dell'ultimo decremento dovuto a loss event

- Si utilizza una variabile detta **ssthresh** per tenere il valore di tale THRESHOLD; quando CongWin raggiunge ssthresh, cresce linearmente

- Quando avviene un loss event:
ssthresh viene settato ad $\frac{1}{2}$ CongWin prima dell'evento.

* Nel caso di 3 ACK DUPLICATI, CongWin viene dimezzato, quindi è legata alla ssthresh e riparte linearmente!

- ssthresh è inizializzato dal protocollo

Raffinamento: FAST RECOVERY

In realtà, in TCP esistono 3 fasi:

- SLOW START
- CONGESTION AVOIDANCE
- FAST RECOVERY

Si entra in fase di fast recovery da una qualsiasi delle altre 2 fasi quando si ha una perdita rilevata tramite 3 ACK DUPLICATI:

- in questo caso, come detto: $ssthresh = \frac{CongWim}{2}$
- però, CongWim non viene dimezzato subito, bensì si pone:

$$CongWim = ssthresh + 3$$

- all'arrivo di ogni ACK duplicato, CongWim continua ad aumentare; solo quando arriva un ACK NUOVO le CongWim diventa uguale alla threshold.

PERCHÉ FARE QUESTO?

Essendo arrivati almeno 3 ACK duplicati, vuol dire che ci sono 3 segmenti in volo e ancora UNACKED; dimezzare netamente le finestre vorrebbe dire saturarle e bloccare la trasmissione di nuovi pacchetti/segmenti per molto tempo, fino alla ricezione di diverse ACK.

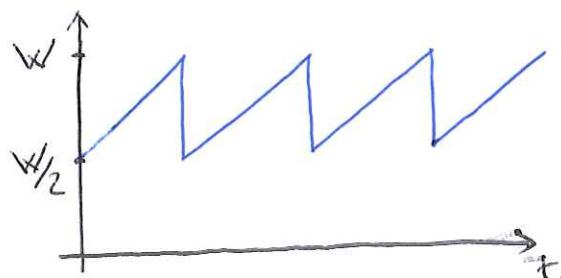
Aumentare temporaneamente le finestre di 3 MSS aiuta ad evitare questo blocco di trasmissione e contribuisce ad aumentare notevolmente le PRESTAZIONI!

TCP THROUGHPUT:

Ignorando lo Slow Start e assumendo di avere sempre dati da trasmettere, vediamo come evolve il throughput in TCP, in funzione della larghezza delle finestre (dipende anche dall'RTT, che assumiamo costante).

Se W è la dimensione delle finestre prima dell'event Coss:

⇒ La dimensione media delle finestre sarà $\frac{3}{4}W$ e il throughput medio è $\frac{3}{4}W$ per RTT.



throughput medio =

$$\frac{\frac{3}{4} W}{RTT} \text{ bytes/sec}$$

Per segmenti di 1500 Byte, un RTT di 100 ms, se si volesse un throughput di 10 Gbps $\Rightarrow W = 83.333$ SEGMENTI \rightarrow è un numero ENORME!

* Per andare da W_2 a W_1 con +1 ogni ack ricevuto, ci vogliono tantissimi ack per giungere al throughput desiderato

\hookrightarrow E' una forte LIMITAZIONE di TCP! (Protocollo pensato non per elevate velocità)

- Le versioni di TCP BIC e CUBIC (standard in Linux), tentano di stabilire un valore delle finestre W_{max} che sia il più ampio possibile e a cui si ha pochissime perdite; stabilito tale valore, tentano di far crescere le finestre ESPOENZIALMENTE verso tale valore W_{max} , per poi procedere con la progressione lineare.

Ad esempio, in BIC, ad ogni ack ricevuto:

$$W_t = \frac{W_{max} - W}{2}$$

\rightarrow Le finestre diventano la META' della DISTANZA dal valore di W_{max}

- THROUGHPUT IN TERMINI DI LOSS RATE:

$$\text{avg. TCP throughput} = \frac{1,22 \cdot MSS}{RTT \cdot \sqrt{L}}$$

Per avere un throughput di 10 Gbps $\rightarrow L \approx 2 \cdot 10^{-10}$ \rightarrow Bisognerebbe inviare 2 segmenti ogni 10 MILIARDI!
IMPOSSIBILE!

- Ripetiamo: TCP non è stato progettato per essere eseguito ad alte velocità!

• TCP FAIRNESS :

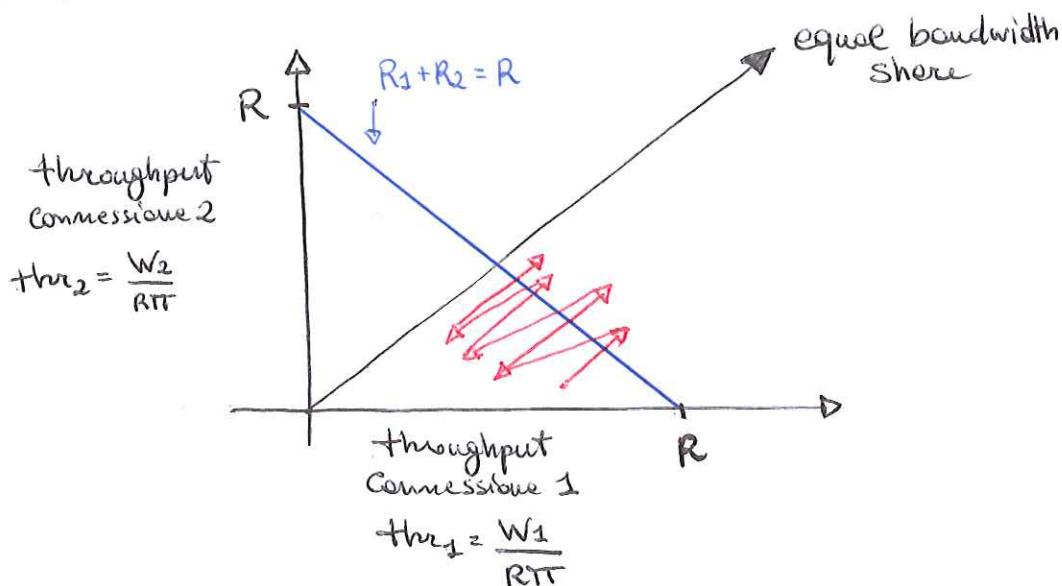
OBIETTIVO DELLA FAIRNESS :

Se K sessioni TCP condividono lo stesso link con collo di bottiglia di banda R , ogni sessione dovrebbe avere un rate medio di $\frac{R}{K}$.

- TCP tende ad essere FAIR, se le sessioni hanno RTT simile, per come è intrinsecamente progettato TCP stesso, concedendo le stesse bande ad ogni sessione!

Vediamo perché: assumiamo di avere 2 sessioni TCP, con stesso RTT, che condividono una banda R , e vediamo come evolve il loro throughput nel tempo.

- CONGESTION AVOIDANCE \rightarrow crescita lineare
- PERDITA con 3 ACK duplicati \rightarrow dimezza le finestre (quindi il throughput).



- Quello che sia il punto iniziale, si tende ad andare verso la bisettrice, che è proprio dove c'è la condivisione delle stesse quantità di banda!

Senza che una sessione abbia conoscenze delle altre, tenderanno, a pari di RTT, ad avere a disposizione tutte le stesse bande e la stessa dimensione delle finestre \rightarrow FAIRNESS

* NOTA: è fatto tutto in maniera AUTOMATICA e DECENTRALIZZATA!

* ATTENZIONE: TCP è fair, ma non è detto che lo siano le applicazioni che lo utilizzano (Browser Web in primis)!

I Browser Web, con HTTP 1.1 PERSISTENTE, non riaprono ogni volta una connessione, non ripartono da Slow Start e già così ne guadagnano in efficienza.

- Ma fanno di peggio: aprendo **PIÙ CONNESSIONI IN PARALLELO** verso il server, diventano poco fair, occupando molte bande!
TCP è fair rispetto alle sessioni, non rispetta all'host!
 - Questo potrebbe potenzialmente scatenare una "guerra" tra i browser su chi apre più connessioni TCP in parallelo; tuttavia, ciò non succede e tipicamente un browser non ne apre più di 4, anche perché gestire connessioni in parallelo è costoso in termini di risorse anche per i browser.
- * NOTA: **UDP** non è fair (effetto!); danneggia di molto le connessioni TCP, "costringendole" ad abbassare il rate di trasmissione.
↳ Aree di ricerca: tentare di rendere UDP più TCP Friendly.