Livello Trasporto

Il livello trasporto si colloca subito un gradino più in basso nei nostri livelli, in questo livello si **segmentano i dati** e si **impacchettano** con protocolli di **TCP** o **UDP**. La principale differenza tra i due è che il **primo è affidabile**, il **secondo no** perchè è il pacchetto se non viene recapitato non verrà rinviato, infatti con **UDP l'invio è best effort**.

Gli endpoint svolgono una duplice funzione:

- **Parte invio (send side)**: si occupa si dividere i dati in segmenti di ugual dimensione
- **Parte ricezione (rcv side)**: si occupa di assemblare i segmenti per creare il dato completo

Livello trasporto e rete

Il livello trasporto serve per mettere in **comunicazione logica due processi** mentre il livello di rete serve per mettere in comunicazione due host, questo potrebbe essere tradotto come: due case (due **host**) con i membri della famiglia (**processi**), se un membro vuole inviare una lettera deve inviarla per posta (**livello rete**) e quando il capofamiglia prende la lettera la consegna alla persona giusta (**livello trasporto**).

Protocolli livello trasporto

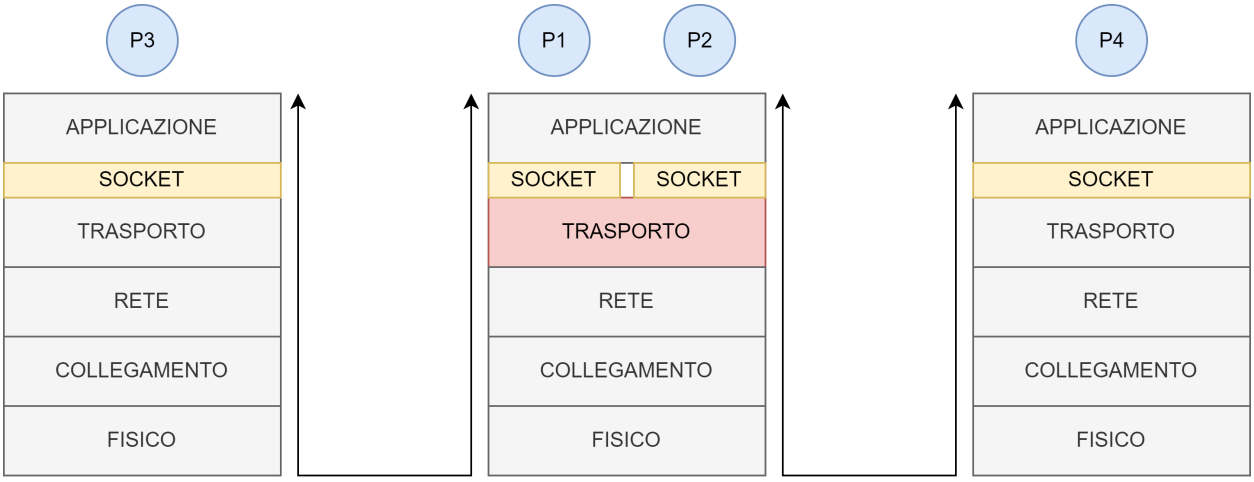
Abbiamo principalmente due protocolli che sono alla base di questo livello.

- **TCP**: affidabile, garantisce un invio ordinato dei pacchetti. Si occupa della gestione di congestione, setup della connessione e controllo del flusso.
- **UDP**: inaffidabile, non garantisce un invio ordinato dei dati.

Nel contesto del livello di trasporto, la comunicazione logica tra processi o applicazioni indica che questi **enti possono inviare e ricevere dati come se fossero collegati direttamente** tra loro, **anche se in realtà** non lo sono, i dati devono prima passare per la rete.

Multiplexing e Demuplexing

Abbiamo una fase di multiplexing quando con più processi apriamo un socket verso un altro processo di un altro host, l'host attuale può avere più socket di comunicazione che inviano e ricevono. Abbiamo multiplexing quando si invia e demuplexing quando si riceve.

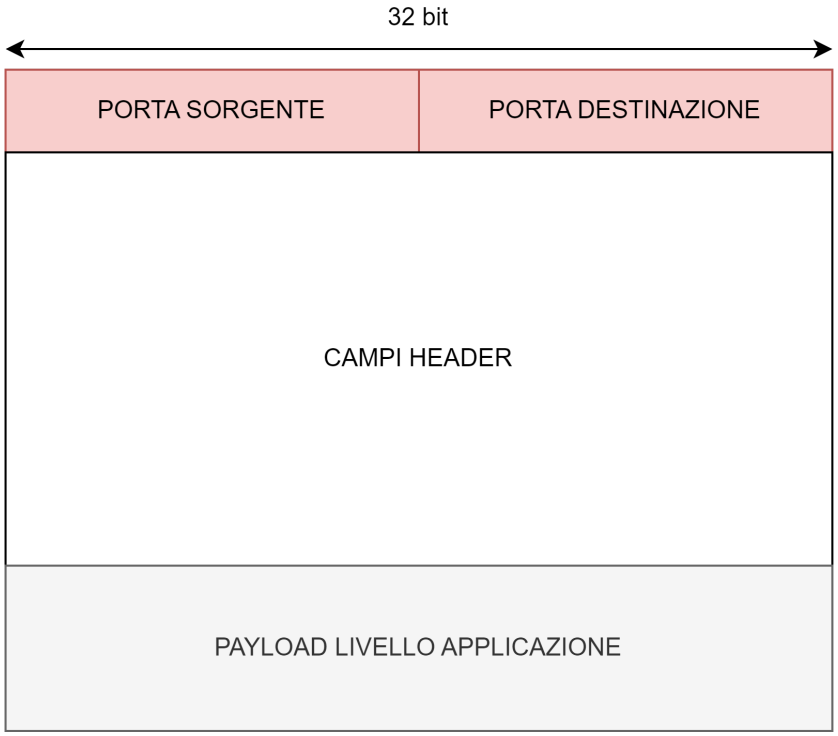


Come vediamo dall'immagine il livello trasporto è in grado i gestire traffico multiplo in uscita (**multiplexing**) e traffico in entrata multiplo (**demuplexing**).

Come funziona il demuplexing

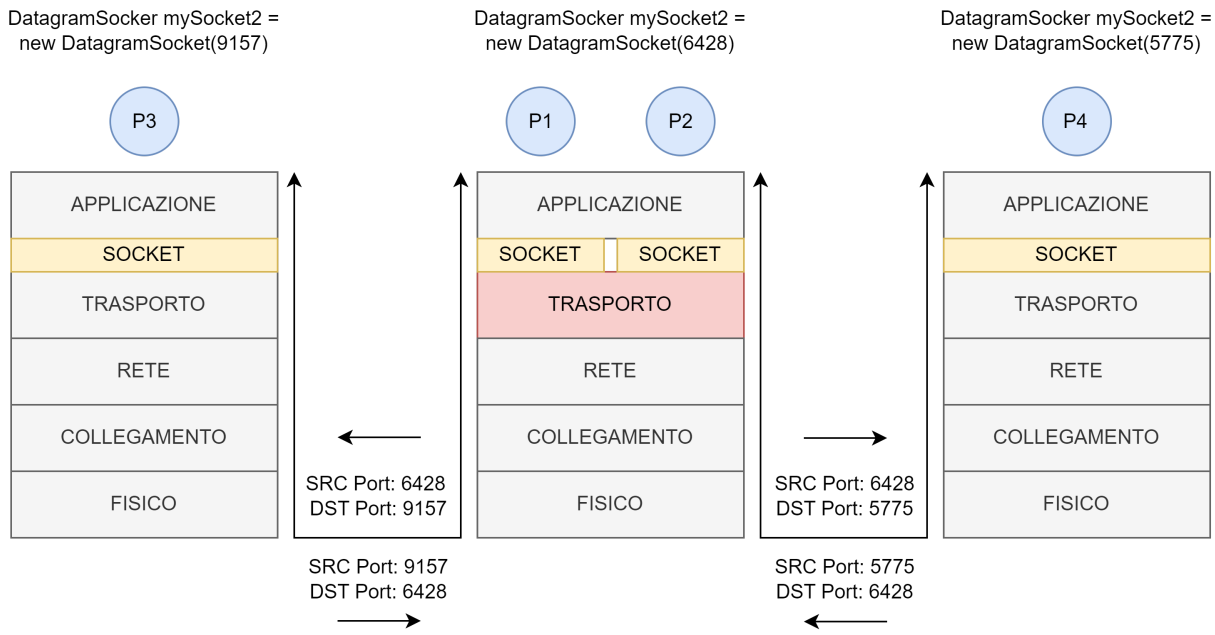
Gli **host ricevono i datagrammi** dell'ip, ogni datagramma contiene **source ip** e **destination ip**, oltre a questo abbiamo anche le **porte** di sorgente e destinazione per capire a quale servizio di livello applicazione stiamo facendo richiesta.

Il segmento TCP/UDP è formato in questo modo:



Non siamo ancora al livello di rete quindi non c'è ancora l'IP. Ma ci sono solo le porte. Gli host usano la **combinazione di IP e porta** per **identificare** un canale di comunicazione chiamato **socket**.

- **UDP**: le richieste sono identificate da **IP di destinazione e porta** di **destinazione**, un server UDP **accetta pacchetti da qualunque risorsa**, in oltre usa lo **stesso socket** per **più sorgenti**, sempre per il discorso che deve essere **veloce** e **non deve fare controlli**.
- **TCP**: le richieste sono identificate da **IP porta di sorgente**, **IP e porta** di **destinazione**, per **ogni sorgente** ha un **socket diverso**.



Il socket TCP è identificato da una tupla di 4 elementi: indirizzo IP sorgente, porta sorgente, indirizzo IP destinazione, porta destinazione, quando arriva la tupla viene indirizzato il segmento al giusto socket (**demux**).

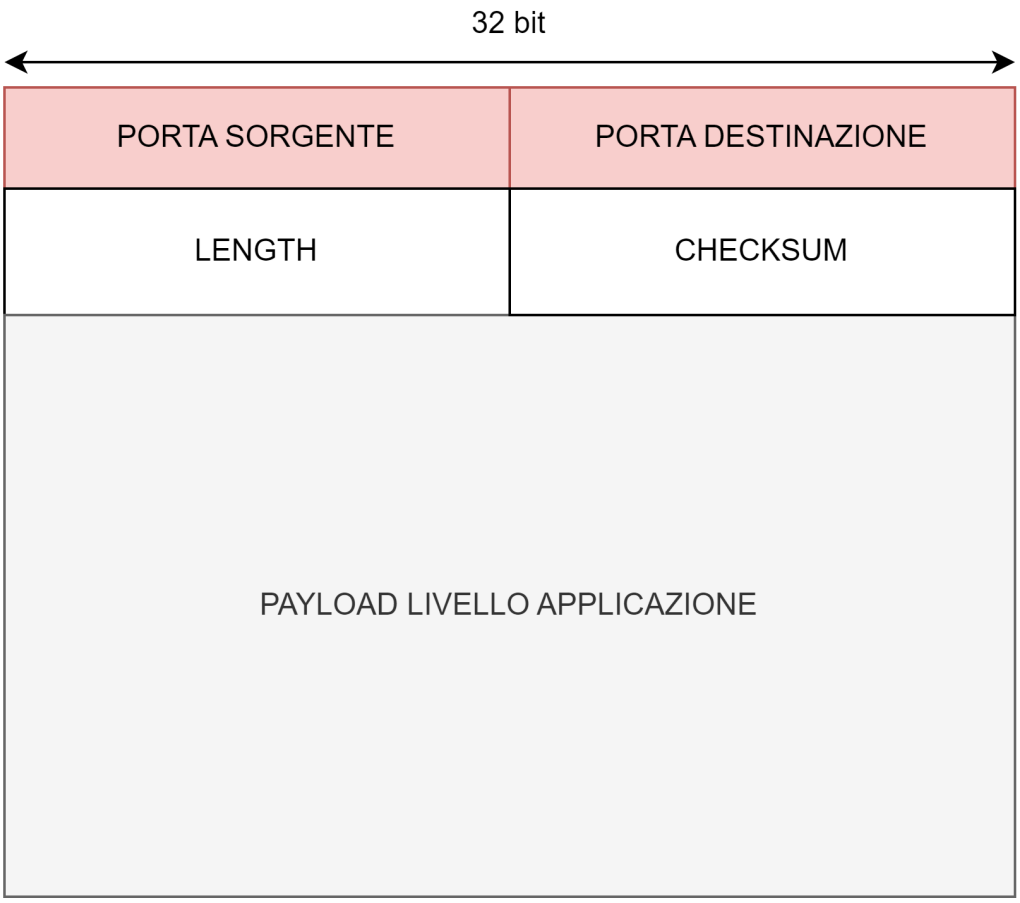
Grazie all'uso dei **thread** possiamo gestire più **socket sulla stessa porta**, questo avviene nei **web server,** che gestiscono molte richieste sulla porta 80. In sostanza quando smistiamo al giusto processo un frammento di dati chiamiamo questa tecnica demuplexing quando raggiunge il giusto thread/processo.

UDP (User Datagram Protocol) RFC 786

Il protocollo UDP è progettato per essere "**best effort**" questo significa che i pacchetti possono essere persi nel mentre e sono consegnati anche non in ordine. Non avviene una handshake a inizio della comunicazione e ogni segmento UDP è indipendente dagli altri.

I principali utilizzi dell'UPD sono per i servizi di streaming o il DNS perchè devono fornire delle risposte veloci senza dover eseguire altro.

Un segmento UDP è formato nel seguente modo:



Perchè dovremmo usare UDP?

- **Semplice**: non si mantiene uno stato di connessione tra mittente e destinatario
- **Non c'è una procedura di connessione:** non c'è una fase di handshake con un ACK
- **Header più piccolo:** è grande 32 bit e basta (4 Bytes)
- **Nessun controllo sulla congestione:** Non c'è limite alla velocità con cui si spedisce un segmento.

UPD Checksum

L'obiettivo principale è quello di rilevare gli errori nei segmenti inviati. Ci sono due elementi principali:

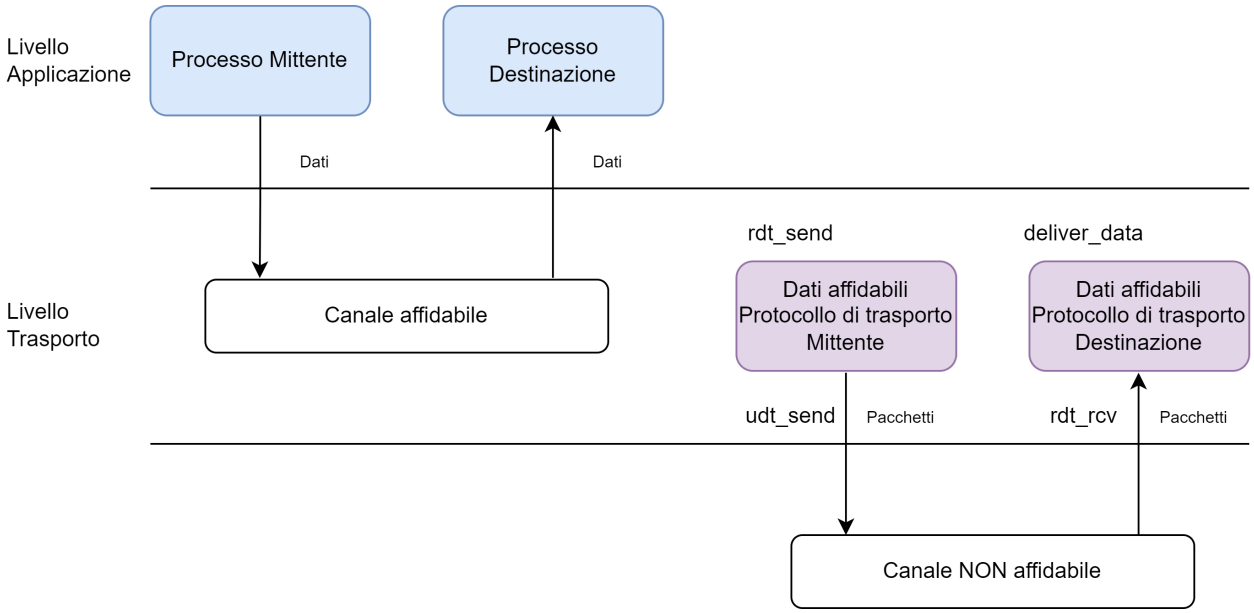
- **Mittente**: inserisce un calcolo nel campo header di 16 bit, il calcolo in questione può essere la somma dell'indirizzo IP e della porta, ad esempio.

- **Destinatario**: il destinatario calcola anche lui il valore del checksum sulla stessa base del mittente (questi meccanismi sono studiati a priori) e se sono uguali significa che l'invio è integro altrimenti no.

Se c'è un riporto nel calcolo, si somma alla somma ottenuta e poi si fa il complemento a 2 del nuovo risultato.

Principi del trasferimento affidabili

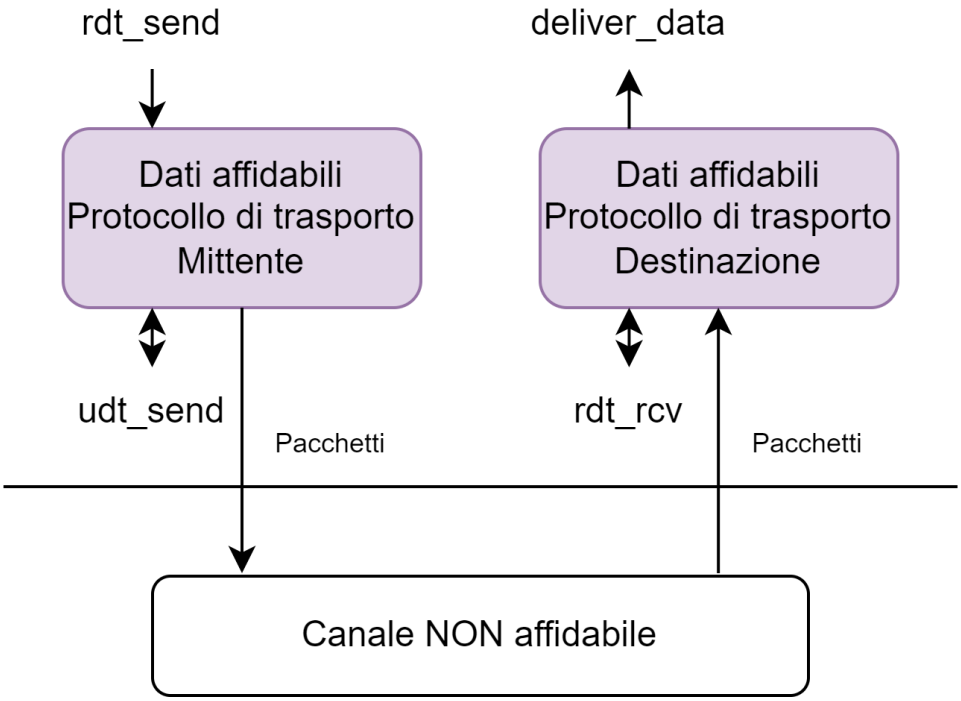
Uno degli argomenti più importanti delle rete è basato sul trasferimento sicuro e affidabile di dati nella rete.



La sigla **RDT** indica il **Reliable Data Transfer**, perciò tutte le volte che comparirà è perchè indica il protocollo che è un'idea di come funziona TCP.

Per iniziare

Da applicazione a trasporto il canale è affidabile, il problema arriva quando da trasporto passiamo al livello di rete, dobbiamo in qualche modo controllare che ci sia una integrità dei dati, il livello di rete e inferiore non è affidabile.



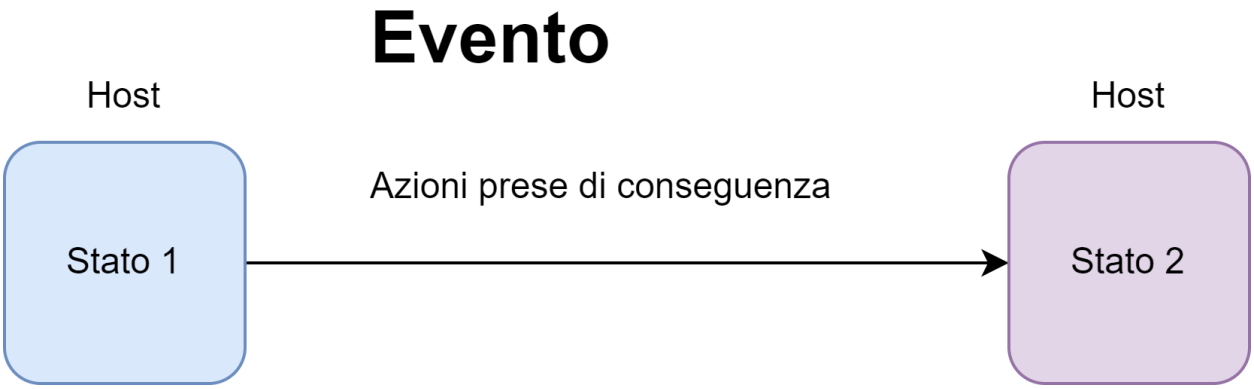
Abbiamo quattro funzioni, spieghiamole al meglio:

- **Lato mittente**:
  - **rdt_send**: accetta dati dal livello applicazione, in sostanza incapsula i dati.
  - **udt_send**: invia i pacchetti attraverso il canale non affidabile
- **Lato destinazione:**
  - **rdt_rcv**: riceve i pacchetti e li estrae dell'incapsulamento, controlla l'ordine se necessario
  - **deliver_data**: consegna i dati all'applicazione locale

Quando arrivano i dati ci sarà un checksum per controllare se i bit non sono stati alterati.

Per definire mittente e destinazione usiamo **FSM** ovvero **Finite States Machines**.

Il mittente e il destinatario hanno diversi stati per cui passano, per una causa passano da uno stato a un altro ed eseguono azioni.

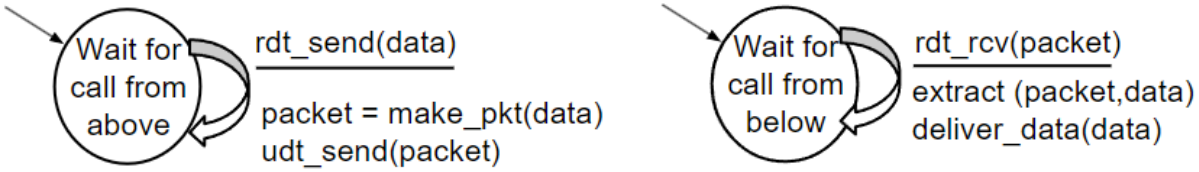


Al termine delle azioni lo stesso host può cambiare il suo stato, infatti nella nostra circostanza il mittente e destinatario rimangono invariati ma le azioni che cambiano a seconda degli eventi.

Protocollo RDT 1.0

Precisazione: il protocollo RDT è un super-semplificazione del protocollo TCP.

In questo ambiente astratta non abbiamo errori nella consegna di bit, non ci sono perdite di pacchetti, ci sono due FSM per il mittente e per chi invia. Gli stati di idle come “Aspetta una chiamata dall’alto” sta ad indicare semplicemente che aspetta dati dal livello applicazione nel caso del mittente, nel caso della destinazione aspetta dati dalla rete.



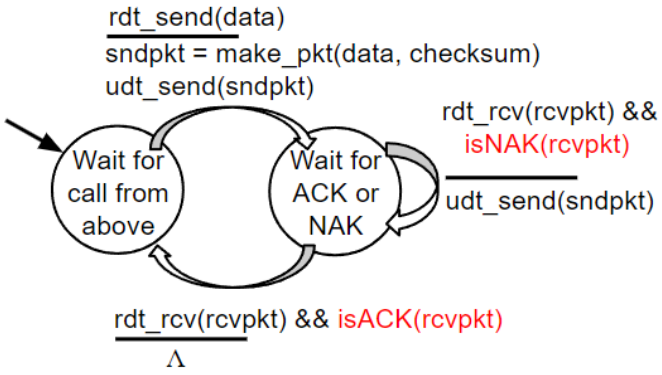
Appena creato il pacchetto si spedisce al canale non affidabile, ovvero internet, quando arriva il pacchetto estrae i dati che erano incapsulati

Protocollo RDT 2.0

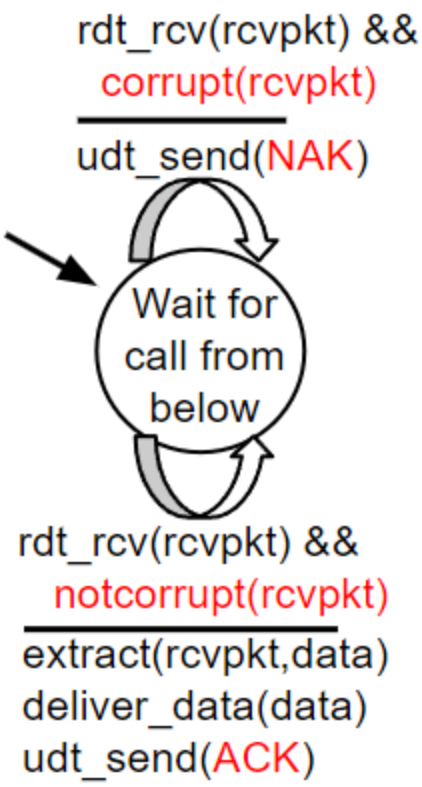
In questo ambiente abbiamo il canale poco affidabile che potrebbe incasinare i bit del pacchetto, proprio per questo bisogna aggiungere un checksum. Ci sono due modi per intervenire all'errore:

- **ACK**: il destinatario invia al mittente un ACK se è andato tutto bene
- **NAKs**: il destinatario invia al mittente un NAK se calcolando il checksum abbiamo una alterazione

Quando troviamo errori ed eseguiamo controlli sugli errori possiamo dire che è il passaggio da rdt 1.0 a rdt 2.0, vediamo lo schema in rdt 2.0 del mittente:



Il mittente aspetterà un ACK dal destinatario, nel caso non ci sia il segnale di ACK, il pacchetto verrà inviato nuovamente, la trasmissione deve essere affidabile. Lo schema rdt 2.0 della destinazione:



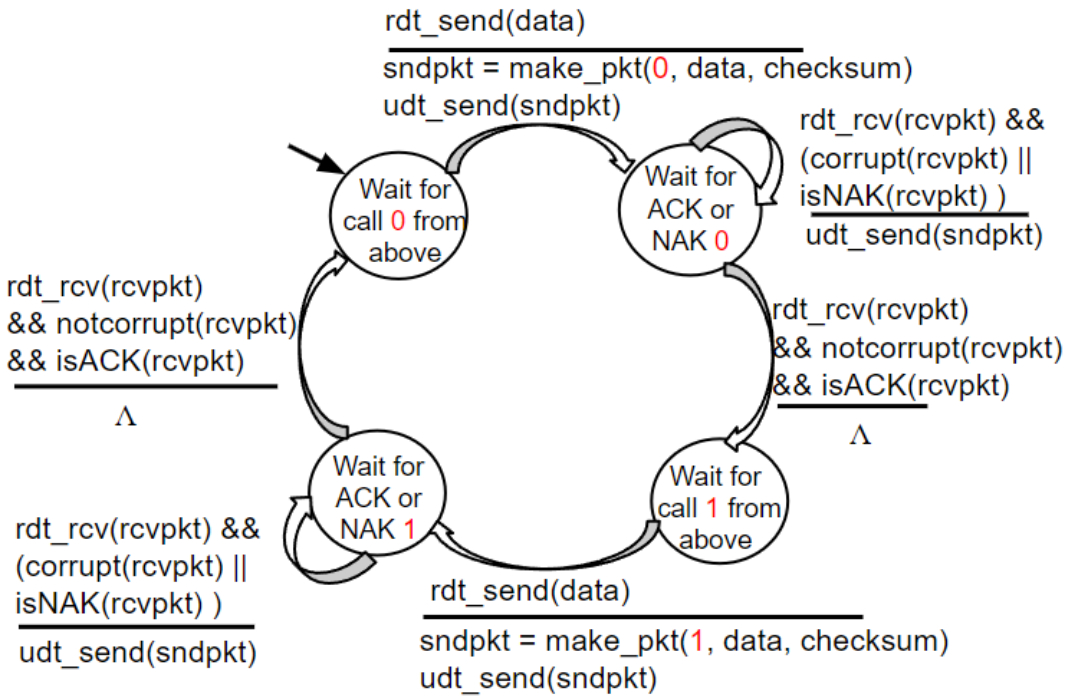
Il destinatario quando arriva il pacchetto deve calcolare il checksum, se viene errato confrontando quello nell'header con quello calcolato interno invia un NAK altrimenti se tutto è andato a buon fine invia un ACK. **Ma cosa accade se AKC o NAK sono corrotti** (criticità del rdt 2.0)?

Protocollo RDT 2.1

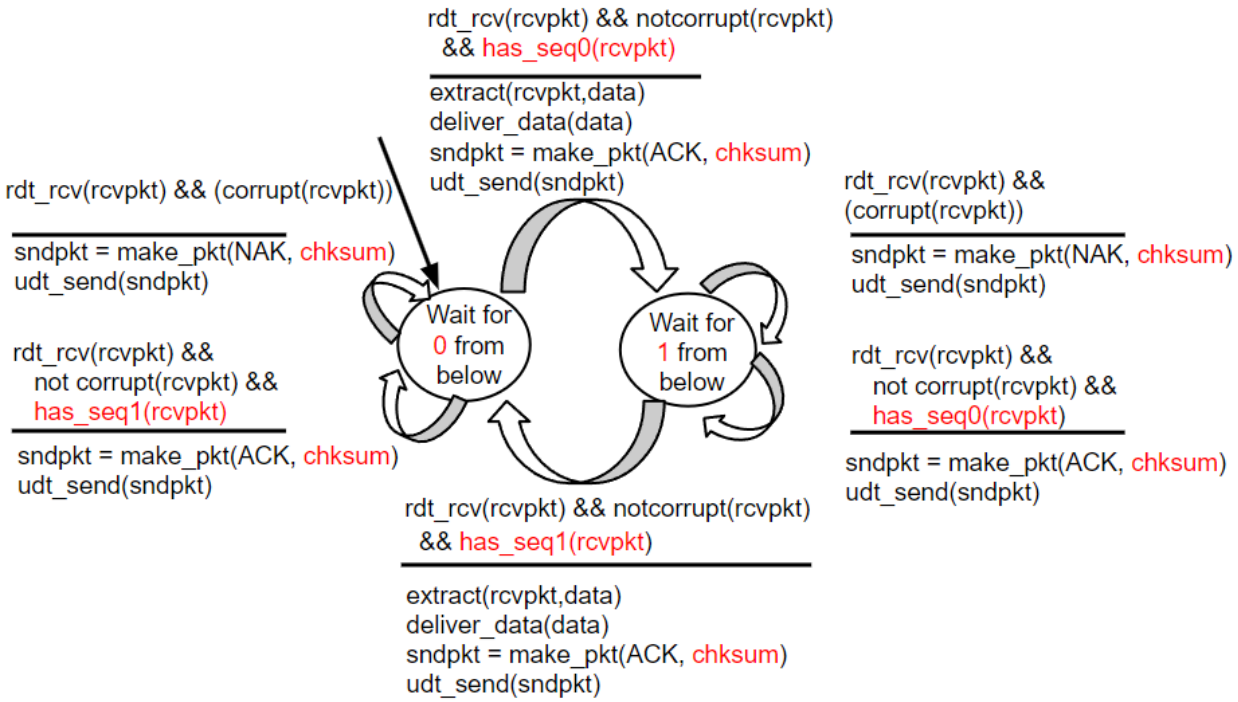
Proprio per questo il destinatario ritrasmette ACK/NAK se viene corrotto, viene aggiunto un numero a ogni pacchetto inviato (0 e 1 per semplicità ma nella realtà ci sono 2^32 possibilità di numeri per la grandezza nell'header), il destinatario scarta i pacchetti con un numero duplicato.

Viene usata la tecnica **stop and wait ovvero che il mittente aspetta una risposta dal destinatario.**

Il mittente avrà una macchina a stati di questo tipo:



La sua funzione è quella di aspettare che venga creato dal livello applicazione il dato, lo invia e se in risposta gli torna un ACK e non è corrotto allora passa allo stato di attesa del prossimo dato. Se è corrotto prova a rinviarlo, stessa cosa accade per i pacchetti successivi. Dal lato del destinatario abbiamo una situazione del genere:



In sostanza, se il destinatario riceve il pacchetto giusto della sequenza allora può continuare a cambiare di stato e quindi aspettarsi 0 o 1. Se riceve un pacchetto che è corrotto invia un NAK invece se non è corrotto ma ha un codice che non è quello che si aspettava manda un ACK per informare che il mittente deve andare avanti con la sequenza.

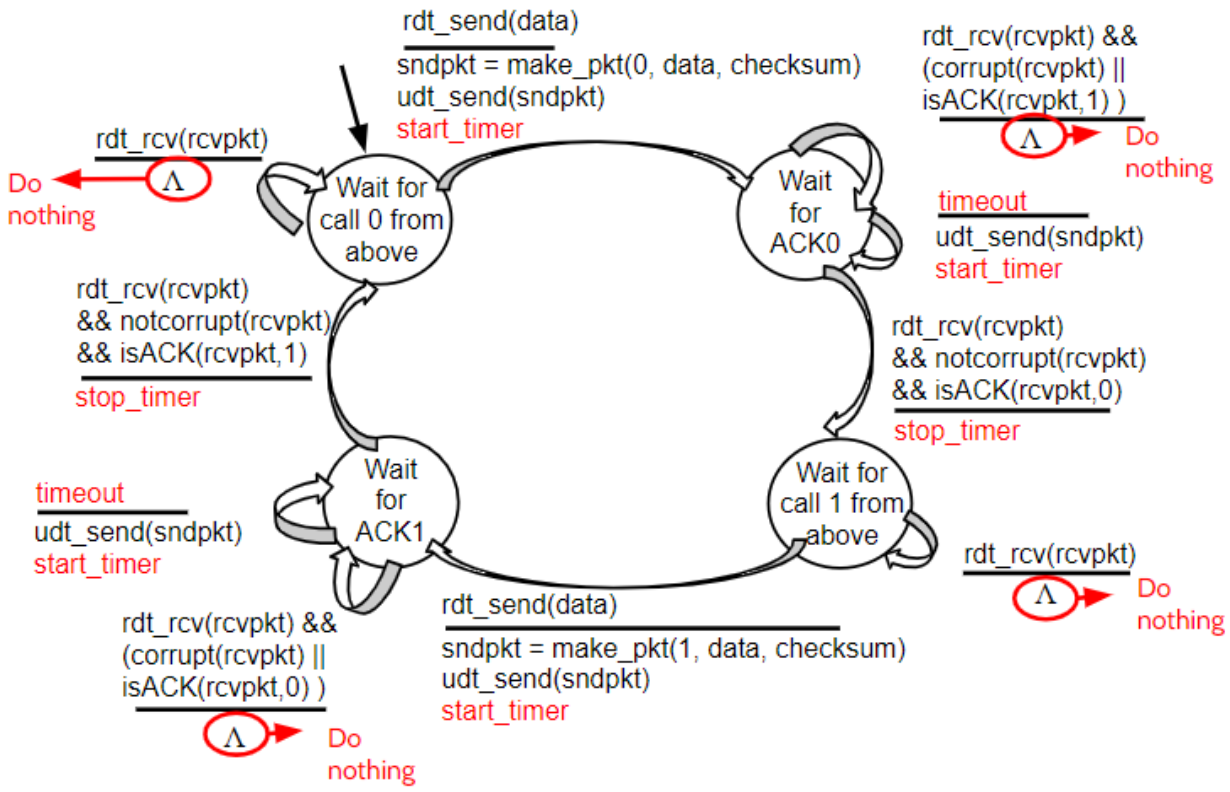
Protocollo RDT 2.2

Abbiamo le stesse funzionalità del RDT 2.1 ma senza usare il NAK, il principio di funzionamento di questa versione è inviare un ACK con l'ultimo pacchetto ricevuto in modo corretto. In sostanza se il pacchetto non è corrotto invia l'ACK con il codice del pacchetto 0 se è l'ultimo che ha ricevuto in modo giusto.

Protocollo RDT 3.0

Fino ad ora abbiamo sempre dato per scontato che il pacchetto arrivasse, magari corrotto ma che arrivasse. Se non arriva il pacchetto abbiamo una situazione di stallo, il mittente invia un pacchetto, il destinatario non riceve il pacchetto e quindi come possiamo sbloccare il mittente che aspetta una risposta di qualche tipo?

Approccio: attente un numero ragionevole di tempo (in ms.) per una ACK. Se il pacchetto non arriva entro un numero di tempo allora si considera perso, il problema è che se il mittente aspetta troppo poco dovrà ritrasmettere il pacchetto, se però il pacchetto era solo molto lento dobbiamo ora gestire un duplicato. Se invece c'è un'attesa troppo lunga non sappiamo come comportarci.



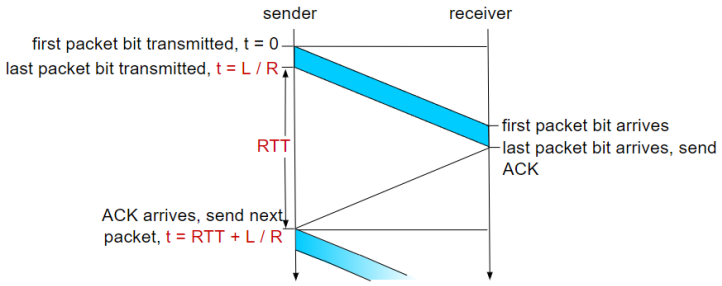
Tutti questi controlli **non sono effettuati da UDP** perchè non ci sono controlli, questi controlli **vengono fatti dal TCP**.

Le performance di RDT 3.0 sono scarse, vediamo un breve calcolo:

- Link: 1Gbps
- Grandezza pacchetto MTU: 8000 bit

Delay = 8000 / 1 * 10^9 = 8ms

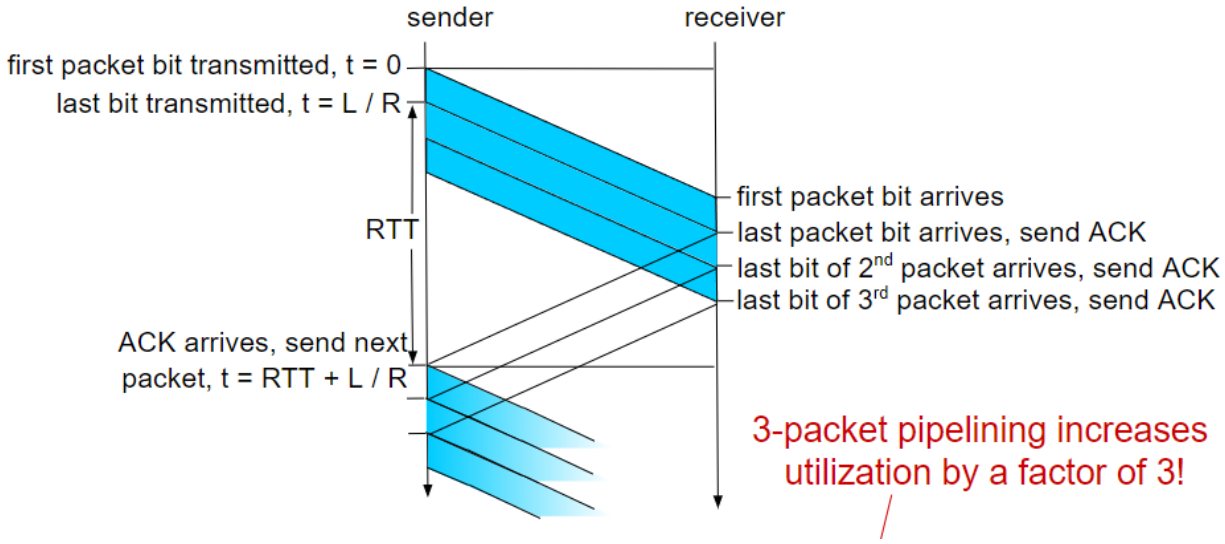
Calcolo per l'invio: Delay / RTT + Delay = 267 Kbps (rivediti il calcolo a casa), ma perchè ci mette così tanto?



La risposta è: perchè ogni volta dovrebbe inviare un ACK di ritorno, quindi il tempo aumenta in modo considerevole.

Introduzione della pipeline

Il mittente non può aspettare ogni volta che vada e torni indietro l'ACK, è troppo lento, dobbiamo quindi introdurre l'invio di più pacchetti.



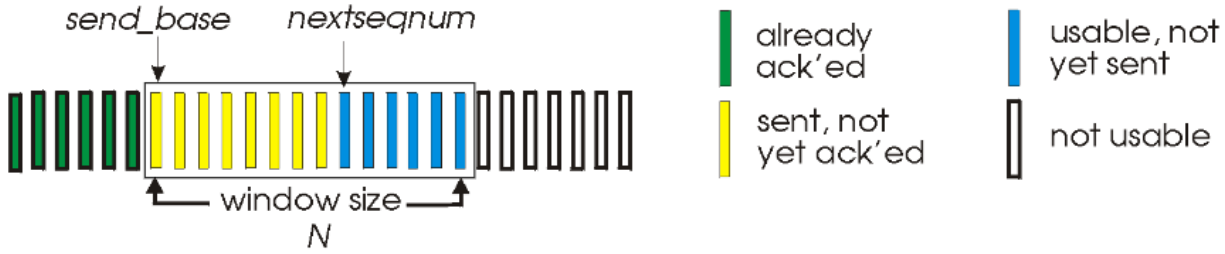
In questo esempio vengono inviati più pacchetti e di ritorno arrivano più ACK, vediamo le due principali strategie di invio in **pipeline di pacchetti**:

- **Go Back N**: il primo approccio è quello di inviare N pacchetti e torna indietro un ACK per dire che gli N pacchetti sono arrivati, se non arriva un pacchetto si invia l'ACK del pacchetto non inviato ad esempio in una sequenza 1, 2, 3, 4, ,5 non arriva il pacchetto 3, il protocollo prevede che il 3 e anche 4 e il 5 vengano ritrasmessi.

- **Selective Repeat:** il destinatario invia una ACK per ogni pacchetto che ha ricevuto, il mittente tiene traccia di ogni pacchetto senza ACK per il nuovo invio, il problema è che bisogna tenere conto di molte più informazioni e quindi è molto più lento rispetto al primo.

Go Back N (GBN)

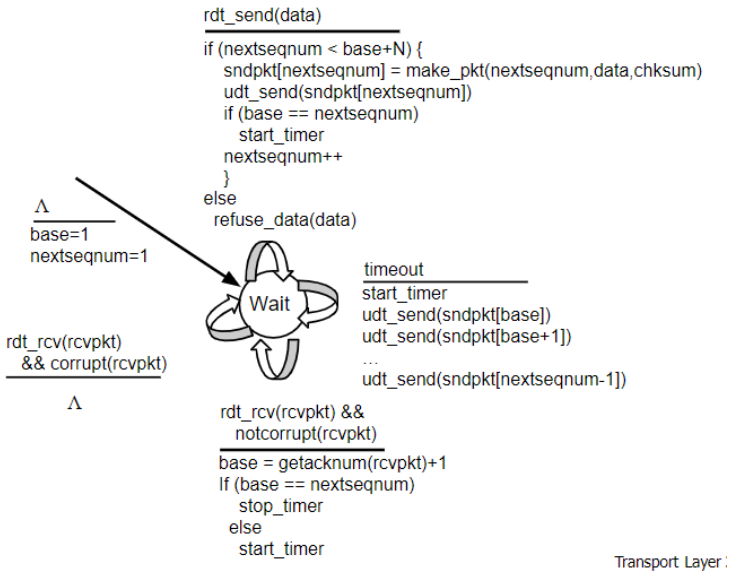
Questo protocollo funziona con una finestra su N pacchetti, più è grande N più sarò veloce, più è piccolo più sarà lento.



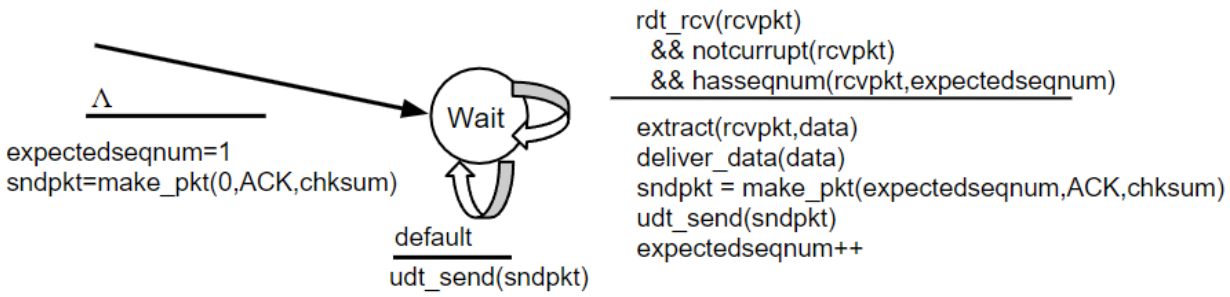
- **Verde**: pacchetto con già un ACK
- **Giallo**: inviato ma senza ACK
- **Blu**: si può usare ma non è ancora stato inviato
- **Bianco**: non utilizzabile

Questa **sliding window** cambia quando riceve degli ACK, ad esempio facciamo caso che tutti i pacchetti gialli siano con l'ACK del pacchetto più a destra (sempre dei gialli) la finestra si sposta di 7 pacchetti in avanti e quindi il primo dei blu sarà a inizio della finestra.

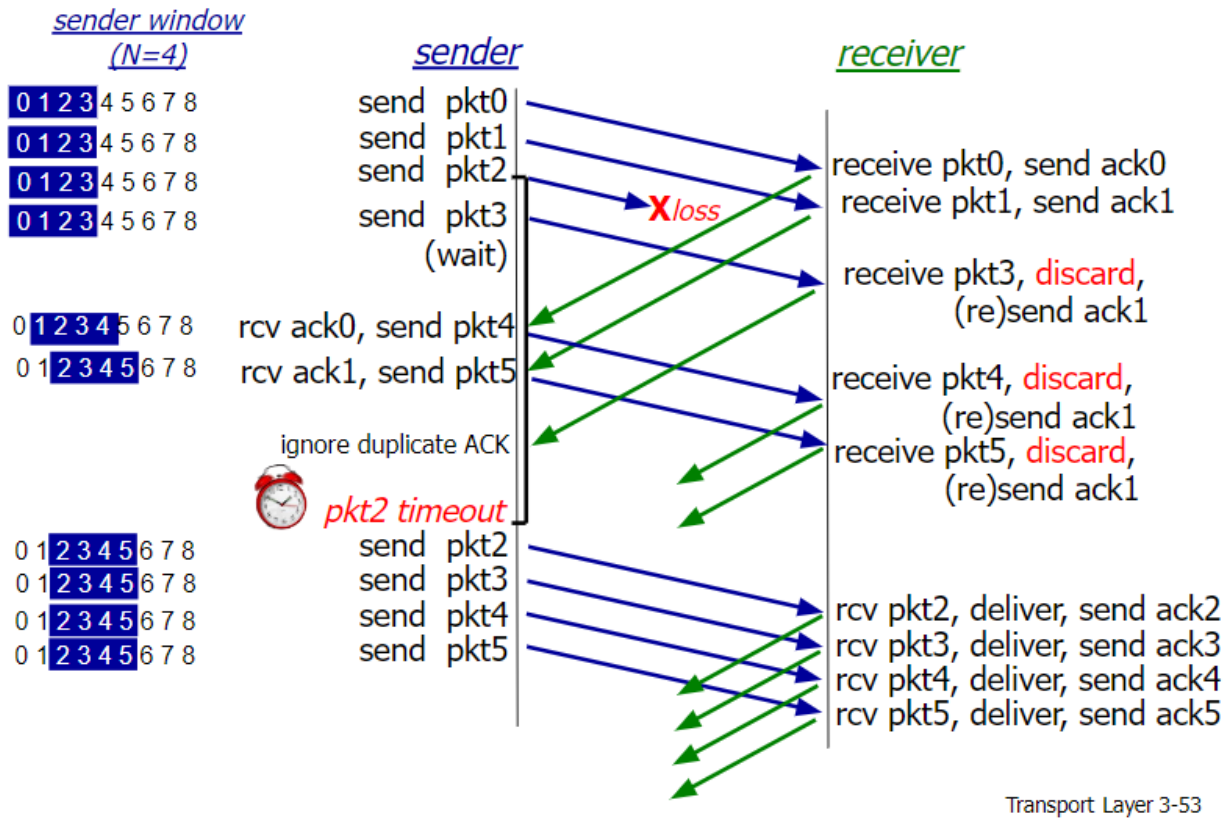
Se invece ci torna l'ACK del quarto pacchetto dei gialli, dovremo ritrasmettere da quel punto in avanti. Ma la finestra si muoverà comunque in avanti. Lo schema FSM del mittente è:



Il destinatario non ha un buffer quindi non può salvarsi quali pacchetti non sono arrivati, semplicemente invia un ACK con il primo pacchetto che non arriva e ci penserà il mittente a rispedire da quel punto. Lo schema FSM del destinatario è:



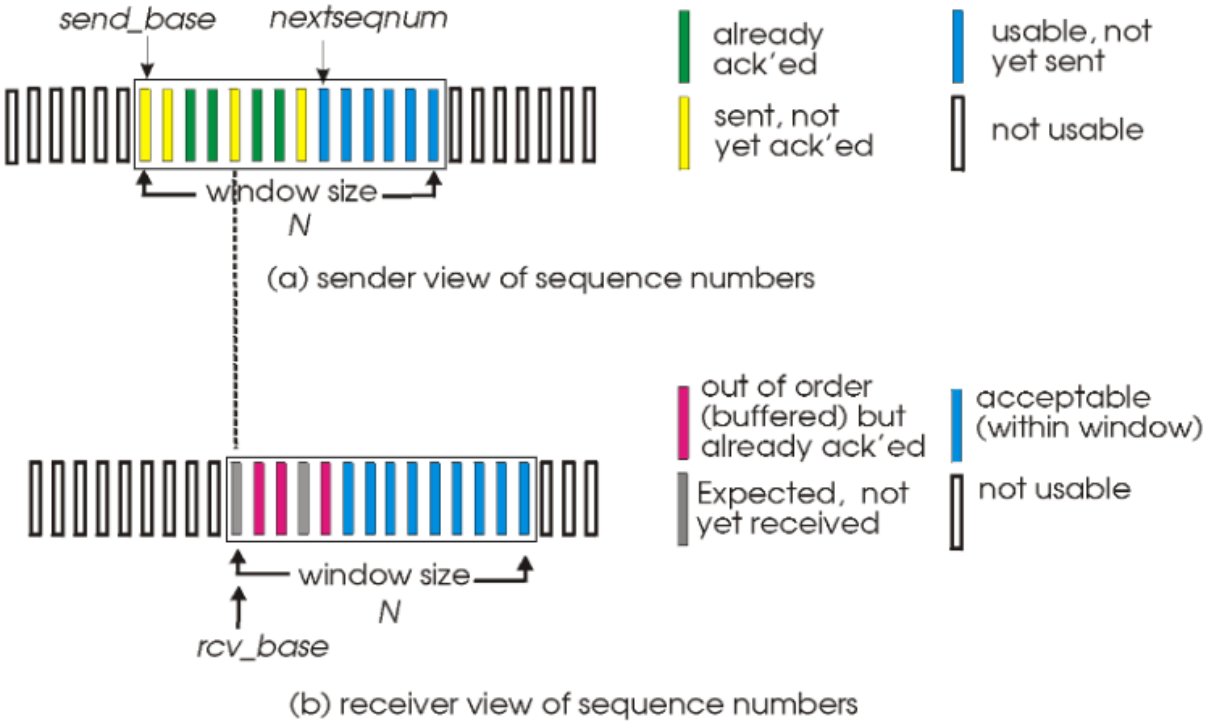
Grazie al **sequence number** il destinatario capisce quando si interrompe la sequenza e informa il mittente fin dove è andato tutto bene



In questo caso il destinatario non può accettare pacchetti che non siano uguali alla variabile **expectedseqnum,** ogni pacchetto che non ha quel numero sarà scartato come da esempio.

Selective Repeat (SR)

Il destinatario sa lo stato di ogni pacchetto in modo individuale, e corregge quelli che non sono arrivati, il mittente spedisce nuovamente solo i pacchetti che non sono stati inviati grazie all'ACK che ha il numero del pacchetto che non va.



Mittente

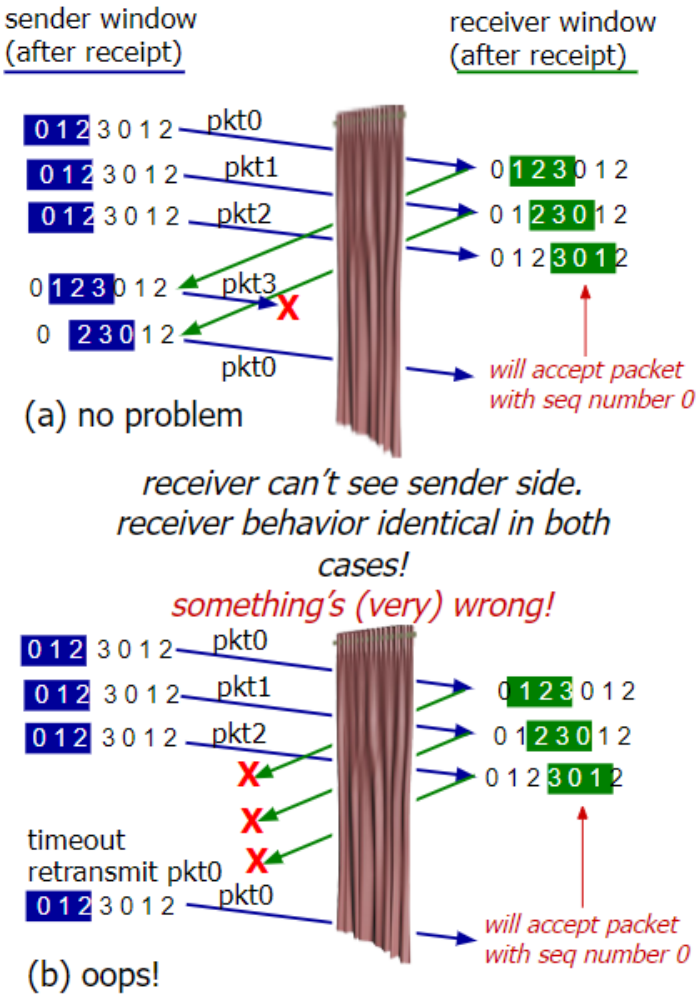
Data dal livello applicativo, se disponibile una posizione nella finestra allora invia un nuovo pacchetto, dopo un certo time-out reinvita il pacchetto n, e resetta il timer. Viene invia un ACK n per ogni pacchetto ricevuto

Destinatario

Se il pacchetto arriva che è nella finestra di invio allora invia un ACK fuori dall'ordine di invio, ha un buffer che tiene traccia dei pacchetti che sono arrivati e serve per aspettare il pacchetto mancante, se il pacchetto è in ordine manda solo un ACK del pacchetto n.

Il problema di SR

Il dilemma che viene facilmente risolto nella realtà è che quando i pacchetti sono inviati e ricevono una ACK è semplice portare avanti la finestra, nel momento in cui non torna l'ACK e il mittente invia nuovamente i primi dati si possono creare ambiguità sui dati:



Il tutto però è risolto semplicemente dal fatto che ci sono 2^32 possibilità di header nella realtà, la possibilità che si ripeta e che ci sia questa confusione è praticamente nulla.

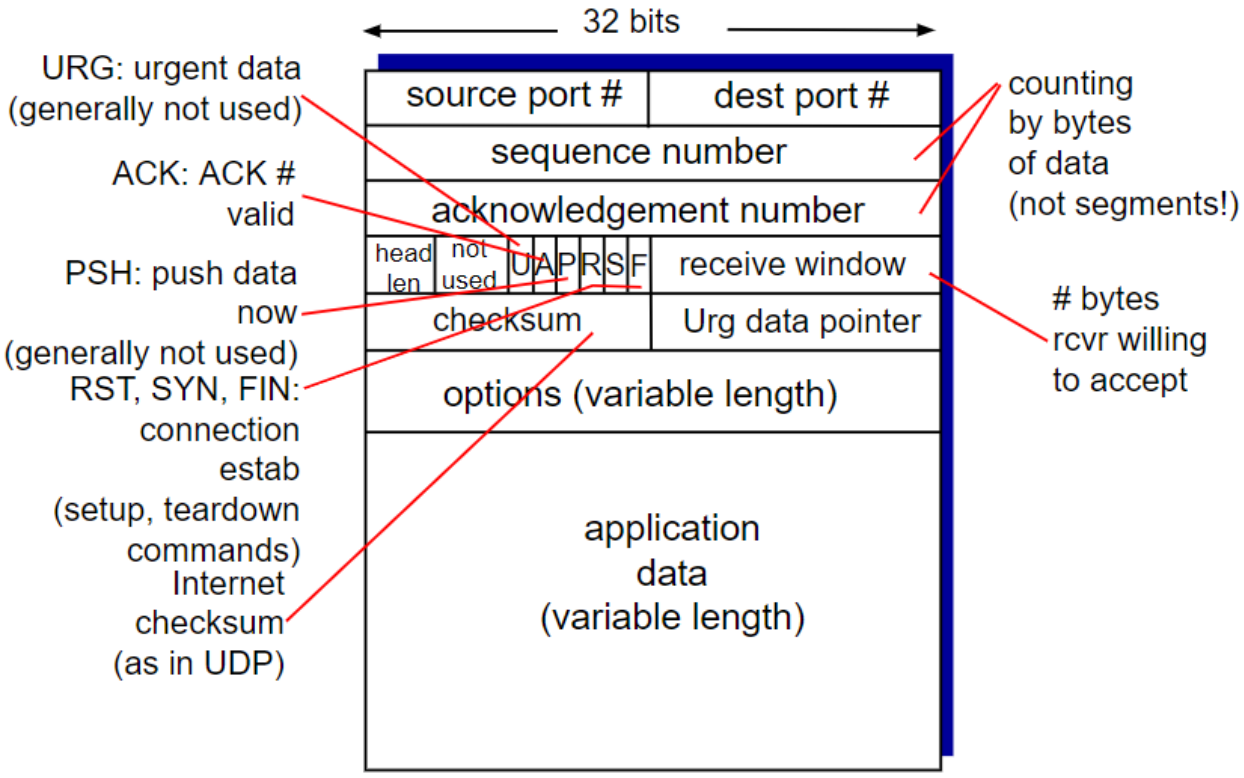
Protocollo TCP

TCP è stato ideato per essere un **protocollo point-to-point** ovvero per un **mittente** e un **destinazione**, è **affidabile** perchè avviene **l'invio in ordine** e ha una **pipeline**. **Non supporta il multicast** (invio e ricezione assieme) ma con appositi accorgimenti è possibile farlo.

Consente di gestire il **full-duplex** ovvero un invio di dati in **modo bidirezionale in contemporanea**, basato su un **MSS** (Maximum Segment Size sinonimo di MTU).

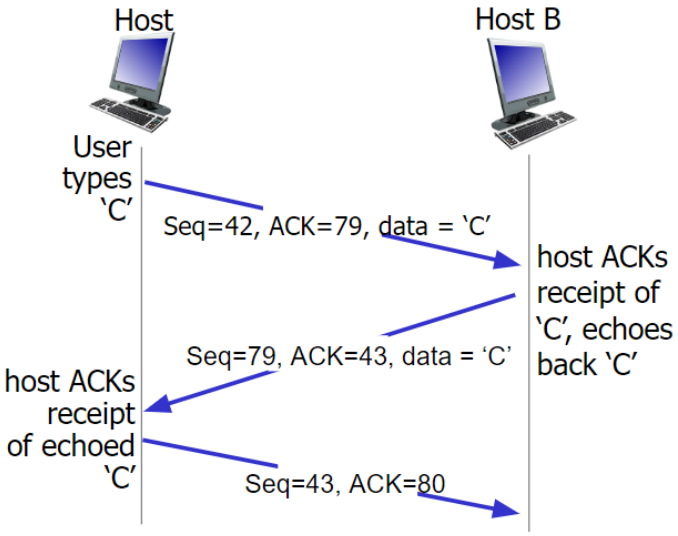
Il protocollo è **orientato** alla **connessione** infatti abbiamo una prima fase **handshake** dove si inizializzano i dati ad esempio quanto è grande la finestra di invio. Ed infine c'è un controllo di flusso per non sovraccaricare la destinazione (quando gli ACK sono in sequenza di **accelera** altrimenti se si ricevono ACK vecchi di **frena**).

Struttura del TCP



Il pacchetto TCP e la risposta ACK sono lo stesso pacchetto, solo che l'ACK è il pacchetto con il bit di A ad 1.

Come avviene lo l'handshake tra due host



simple telnet scenario

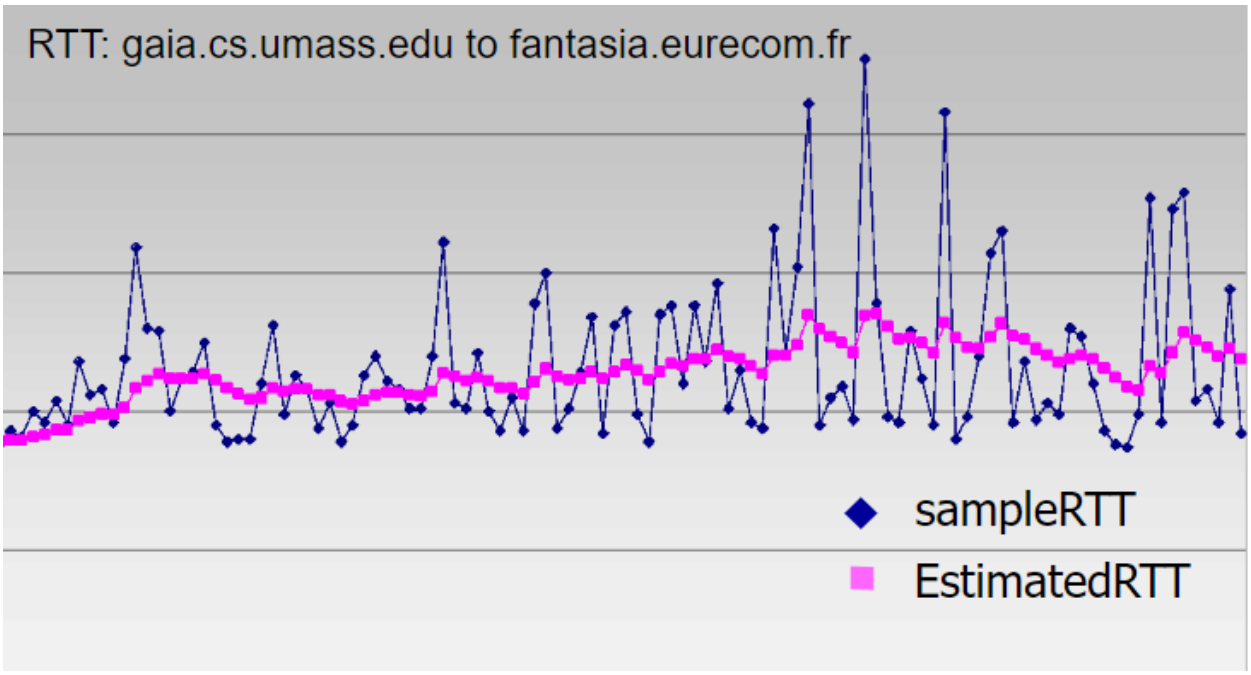
Come possiamo vedere i primi Seq e ACK sono scelti a caso, per evitare che i pacchetti si mischino con altri precedenti. I successivi sono incrementali. L'host risponde con lo stesso payload che ha ricevuto, questa funzione è chiamata **piggybacked Seq di risposta è l'ACK di domanda** mentre il nuovo ACK è il Seq mittente più uno. Abbiamo ancora un ultima comunicazione che ha come Seq l'ACK del destinatario e come ACK il Seq più uno.

Come viene calcolato il timeout

Quanto deve essere lungo questo timeout? Se è **troppo corto rischiamo di fare delle ritrasmissioni** inutili, se sono **troppo lunghi il mittente deve aspettare troppo l'ACK** quindi perdiamo tempo. Dobbiamo quindi stimare il RTT così da far combaciare il RTT con il timeout. Si prende il RTT precedente e poi per le misurazioni successive si prova a calcolarlo.

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$

Il tempo estimato viene pesato in base ad α e solitamente è impostato a 0.125. Serve per dare una importanza in percentuale al RTT attuale:



La parte viola è quella stimata, **va a smussare la curva reale** ovvero la blu. Il timeout calcolato serve per non dare troppa importanza a valori sballati e cerca di creare una curva smussata.

Non basta calcolare RTT, bisogna calcolare anche la **salita o la discesa che fa il grafico**:

$$DevRTT = (1 - \beta) * DevRTT + \beta * |SampleRTT - EstimatedRTT|$$

Questa formula calcola la differenza tra il valore calcolato e vogliamo vedere se è il valore ha un gap elevato o no. Solitamente si imposta β a 0.25. In conclusione il timeout vale:

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

La moltiplicazione per 4 viene chiamato "**margine di sicurezza**" perchè è meglio aspettare un po' di più che dove ritrasmettere un pacchetto, un altro esempio di dati che sono stati scelti empiricamente. Infatti noi possiamo stimare l'RTT ma ci teniamo larghi con il calcoli perchè una ritrasmissione costa molto in fatto di computazione. Lo scenario perfetto sarebbe che RTT e RTO (**Retransmission Timeout**) fossero quasi uguali (che RTO fosse leggermente più lungo di RTT così da renderlo ottimizzato).

Esercizio (sample esame)

Supponete che i 5 valori misurati di SampleRTT siano **106** ms, **120** ms, **140** ms, **90** ms e **115** ms. Calcolate **EstimatedRTT** dopo l'acquisizione di ogni valore di SampleRTT, usando un valore α=**0,125** e assumendo che il valore **EstimatedRTT** appena prima dell'acquisizione del primo di questi cinque campioni fosse **100** ms. Calcolate anche **DevRTT** dopo l'acquisizione di ogni campione, assumendo β=**0,25** e che il valore di DevRTT appena prima dell'acquisizione del primo di questi 5 campioni fosse **5 ms**.

Infine calcolate il valore di TCP TimeoutInterval dopo l'acquisizione di ogni campione.

Calcolo di **EstimatedRTT** per i 5 valori:

1. (1 - 0.125) * 100 + 0.125 * **106** = 100,75ms
2. (1 - 0.125) * 100,75 + 0.125 * **120** = 103,15ms
3. (1 - 0.125) * 103,15 + 0.125 * **140** = 107,75ms
4. (1 - 0.125) * 107,75 + 0.125 * **90** = 105,53ms
5. (1 - 0.125) * 105,53 + 0.125 * **115** = 106,71ms

Calcolo di **DevRTT** per i 5 valori:

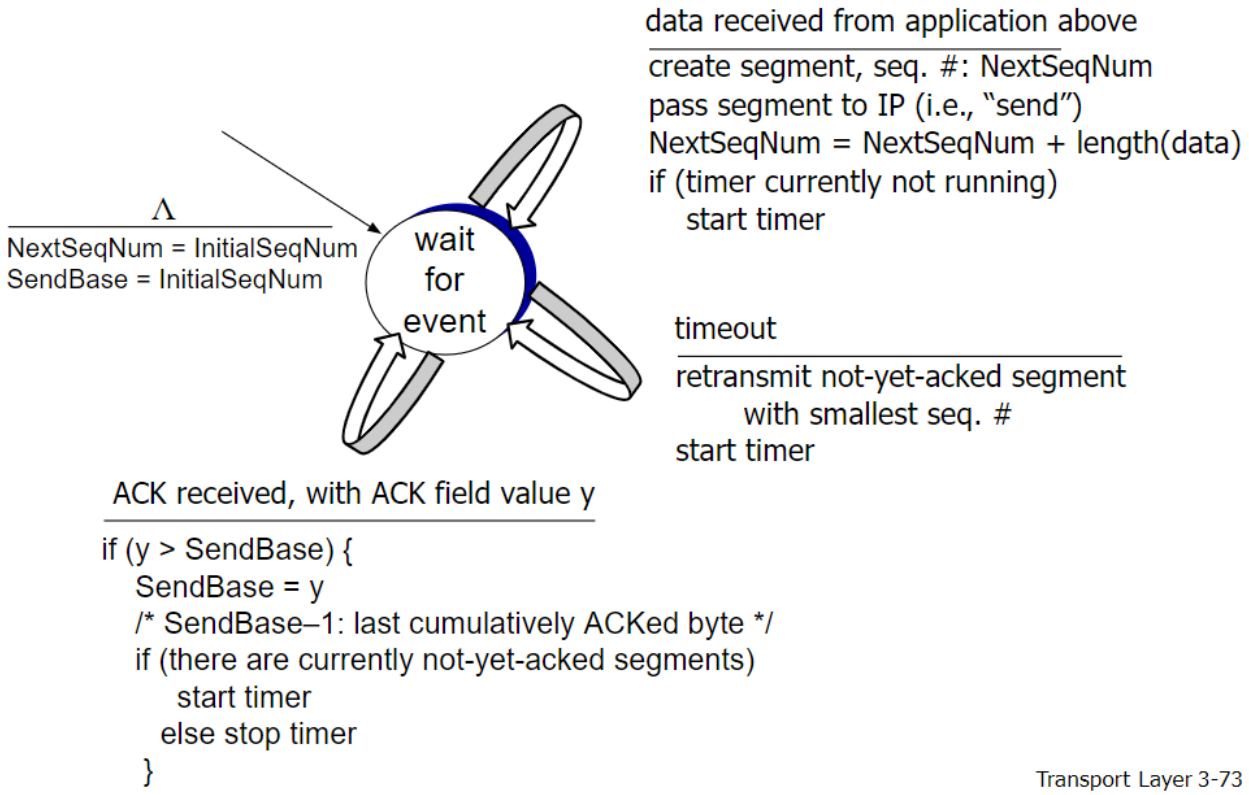
1. (1 - 0,25) * 5 + 0,25 * |**106** - 100,75| = 5,06ms
2. (1 - 0,25) * 5,06 + 0,25 * |**120** - 103,15| = 8ms
3. (1 - 0,25) * 8 + 0,25 * |**140** - 107,75| = 14,06ms
4. (1 - 0,25) * 14,06 + 0,25 * |**90** - 105,53| = 14,42ms
5. (1 - 0,25) * 14,42 + 0,25 * |**115** - 106,71| = 12,88ms

Timeout finale: 158,83ms (106,71 + 4 * 12,88)

TCP e trasferimenti affidabili

Il tutto si basa sulle ACK se si ricevono in fila tutto sta bene, se l'app riceve vecchie ACK allora bisogna intervenire. **L'ACK è legato a quanti byte ha ricevuto**, non cresce di 1 ma del numero di byte del pacchetto ricevuto. Se un pacchetto ha

20 Byte la nuova ACK è ACK_precedente + 20 Byte. Il suo valore viene continuamente ricalcolato così come il timeout per il timer di ritrasmissione.



Transport Layer 3-73

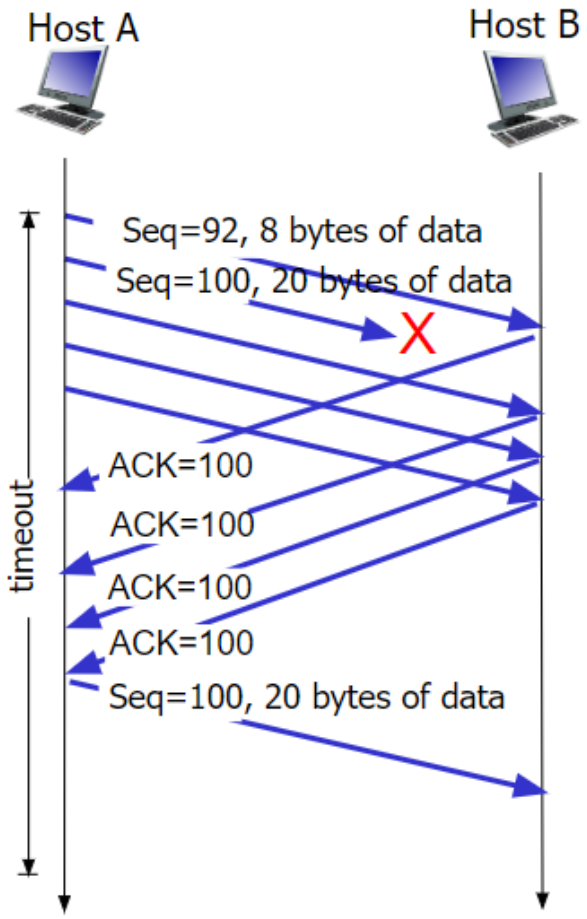
Evento ricezione pacchetto	Azioni TCP destinatario
Arrivo in ordine del segmento: tutti i segmenti fino a quel numero sono già stati segnati con un ACK	**ACK in ritardo**, il mittente può aspettare fino a 500ms per inviare un ACK, se non arrivano altri segmenti lo invia.
Arrivo in ordine del segmento: il segmento è in ordine ma c'è un altro pacchetto con ACK in attesa	Manda immediatamente un **ACK cumulativo**, così conferma entrambi i pacchetti.
Arrivo non in ordine del segmento: il segmento ha un numero più alto di quello aspettato, gap rilevato	Manda immediatamente un ACK duplicato, per far capire che l'ordine si è interrotto.
Arrivo di un segmento che **parzialmente** o **completamente** riempie il **gap**.	Invia un ACK del valore più inferiore del gap che è stato riempito.

La ritrasmissione dei pacchetti non avviene solo per il timeout, possono avvenire anche quando vediamo che ci arrivano gli stessi valori di ACK in modo ripetuto.

Significa che qualcosa non sta andando nel verso giusto.

Ritrasmissione rapida

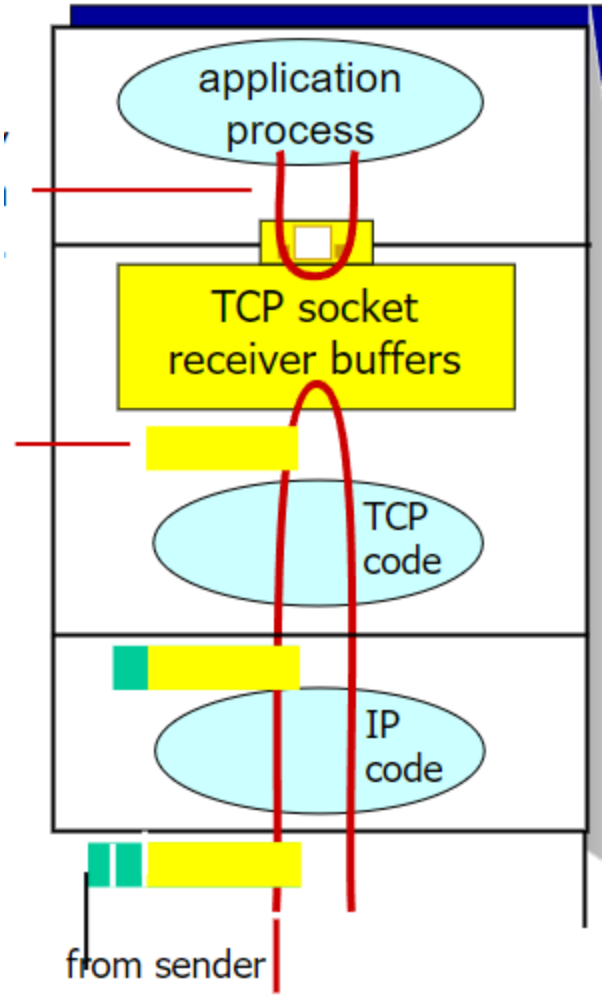
Dopo 3 ACK duplicati oltre a quello di base ci dice che dobbiamo ritrasmettere il pacchetto in modo prematuro, così da diminuire i tempi di attesa.



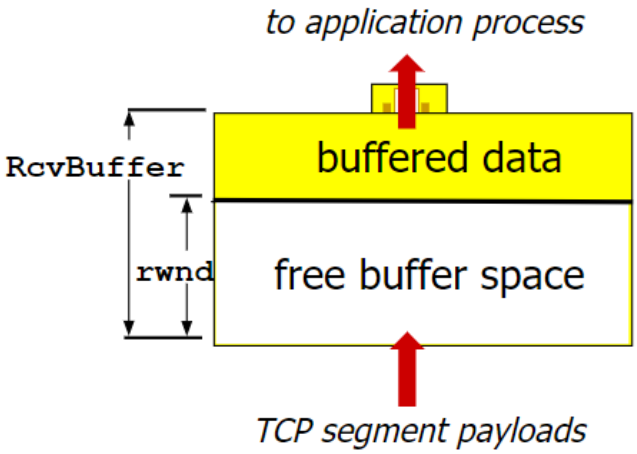
Il TCP funziona con GBN o SR? **La risposta è che è ibrido**, può sfruttare entrambi a seconda delle occasioni, possiamo impostare noi cosa usare.

Controllo del flusso

Il destinatario controlla il mittente su quanti pacchetti può inviare in quel frangente, il mittente non deve sovraccaricare la destinazione perchè si potrebbero perdere delle informazioni inutilmente.



L’applicazione legge a un determinato **ritmo** i **buffer TCP**, se però il **mittente invia** a un **ritmo** troppo **elevato** rispetto alla lettura dei buffer, **c’è una congestione**, per questo bisogna ritmare nel modo corretto l’invio dei segmenti con la loro lettura.



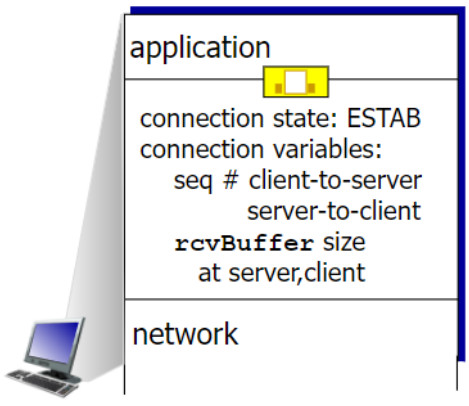
Il buffer solitamente è di 4096 Bytes, nell'header TCP viene incluso il valore di **rwnd** così da specificare quanto è pieno il buffer.

Se la differenza tra i pacchetti inviati con i pacchetti da inviare deve essere minore di rwnd altrimenti si rischia di perdere dei valori.

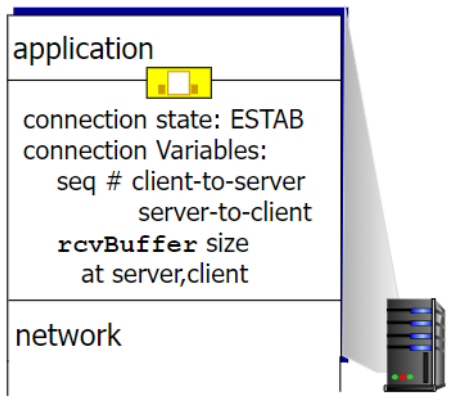
$$LastByteSent - LastByteAcked \leq rwnd$$

Gestione delle connessioni

Prima di gestire lo scambio di informazioni dobbiamo passare per una fase di handshake per la quale si stabiliscono i parametri della connessione.



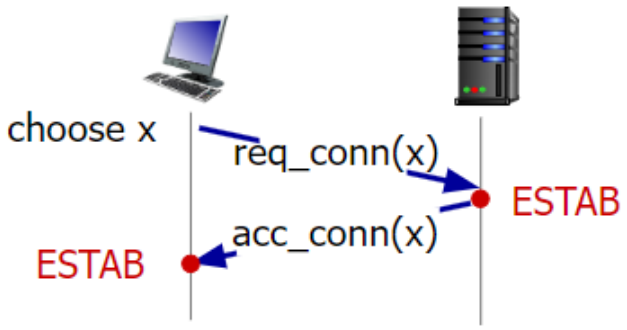

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```



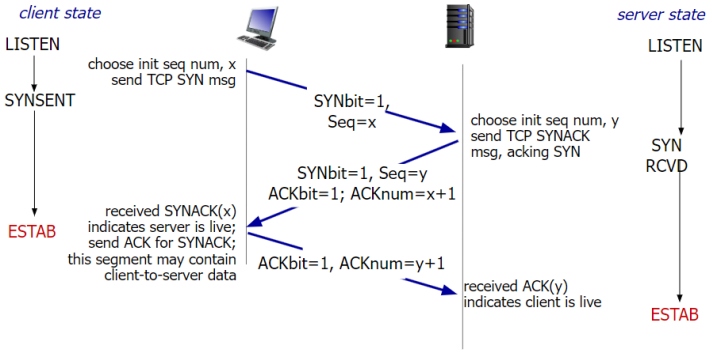

```
Socket connectionSocket =
  welcomeSocket.accept();
```

Handshake a 2

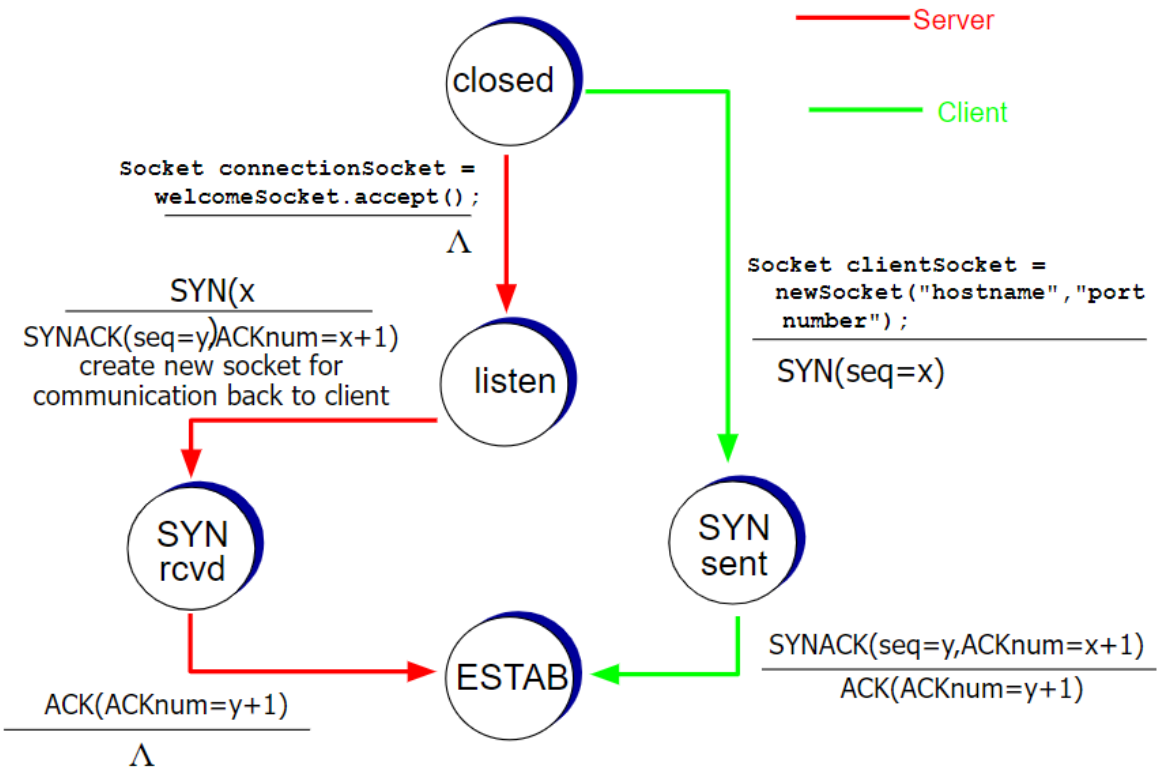
La prima e più rudimentale forma di handshake è quella a 2, dove il mittente chiede se può parlare al destinatario, e il destinatario accetta o meno:



Handshake a 3



In questa versione si inizia la connessione scegliendo un numero di init, si invia, il server notifica la ricezione con un ACKnum che è il numero x + 1, a questo punto il client capisce che il server è ancora vivo e invia un ACKnum che un numero y + 1, viene scelto da lui quando riceve il primo segnale dal client. Così facendo il client informa il server che è vivo.

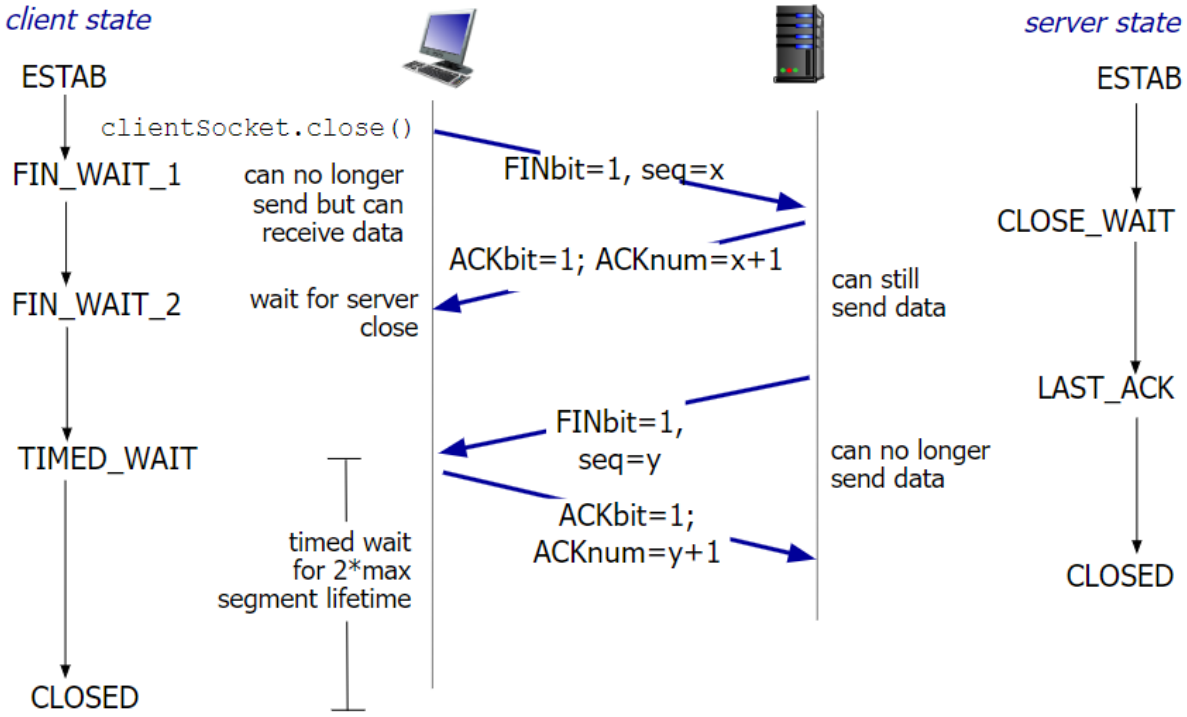


Ci sono un numero limitato di connessioni pendenti ovvero solo 5 host che non mandano l'ultimo ACK ovvero il conclusivo dell'handshake.

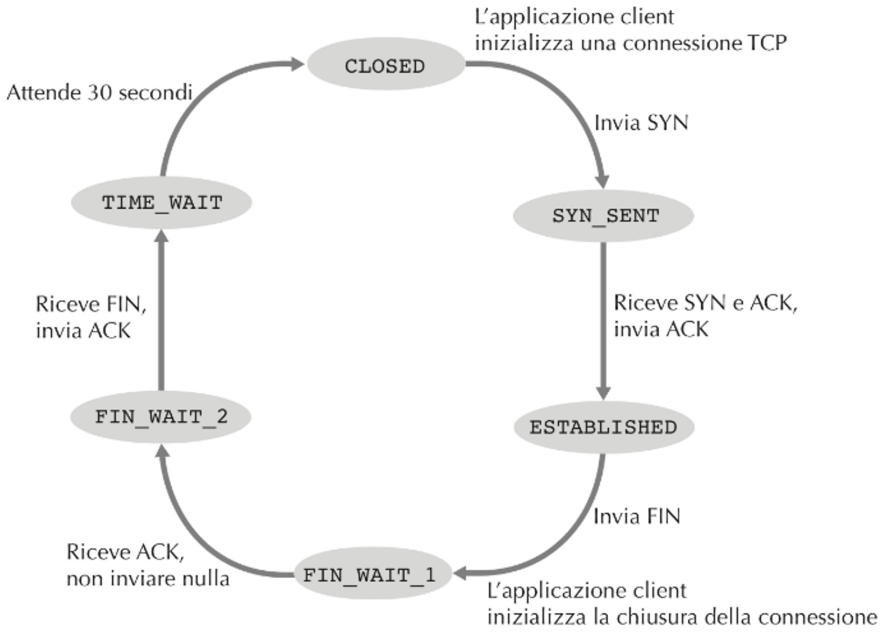
Il primo pacchetto SYN è presente come S dentro il pacchetto di TCP.

Come avviene la chiusura di una applicazione

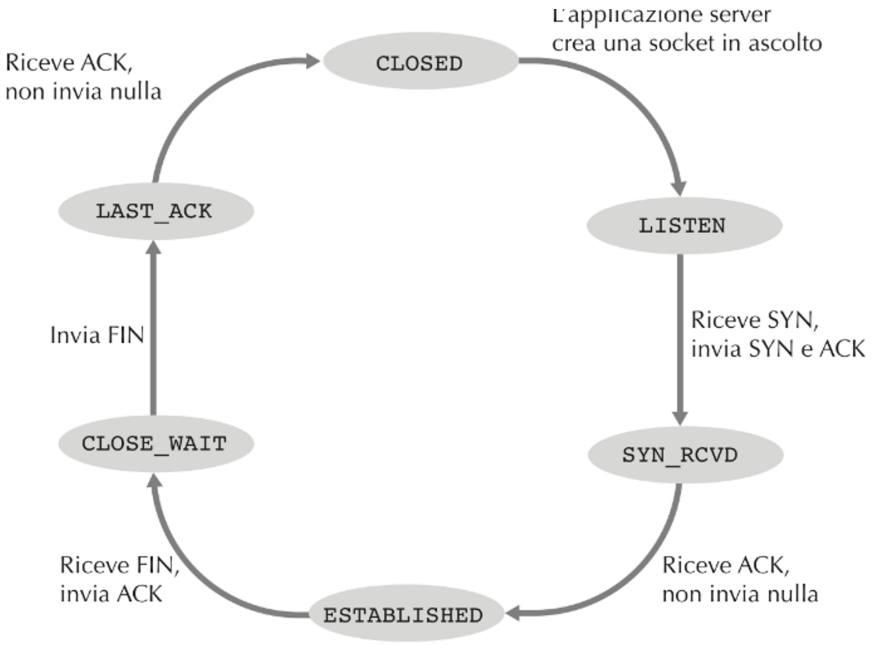
Quando bisogna terminare la comunicazione tra client e server, avviene uno scambio di pacchetti TCP con FIN = 1, il client dopo l'invio di questo pacchetto non **può più inviare dati** ma può ricevere mentre il server può ancora mandare dati dopo, quando è pronto a terminare la comunicazione invia anche lui un pacchetto con la flag di FIN a 1, così facendo chiude la comunicazione e anche lui **non può più inviare dati**.



Ciclo di vita di un client nella connessione



Ciclo di vita di un server nella connessione



Gestione della congestione

Un problema della gestione è che non parliamo di un singolo mittente e un singolo destinatario, ma abbiamo più mittenti per un singolo destinatario e quindi:

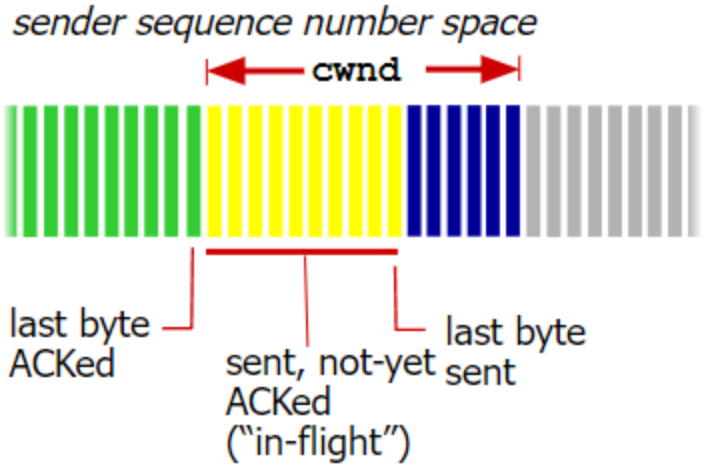
- Se **alziamo** la **velocità** di tutti → **congestione**
- Se **abbassiamo** la **velocità** di tutti → **utilizzo non ottimale della rete**

Una possibile soluzione si basa sull'ACK, l'unico modo per creare una regola:

- Se il mittente riceve un **ACK in sequenza** → **accelera**
- Sei l mittente riceve un **ACK fuori sequenza** → **decelera**

Ma non c'è un controllo reale sulla congestione. Abbiamo quindi tre fasi:

1. **Slow Start** (SS)
2. **Congestion Avoidance** (CA)
3. **Fast Recovery** (FR)



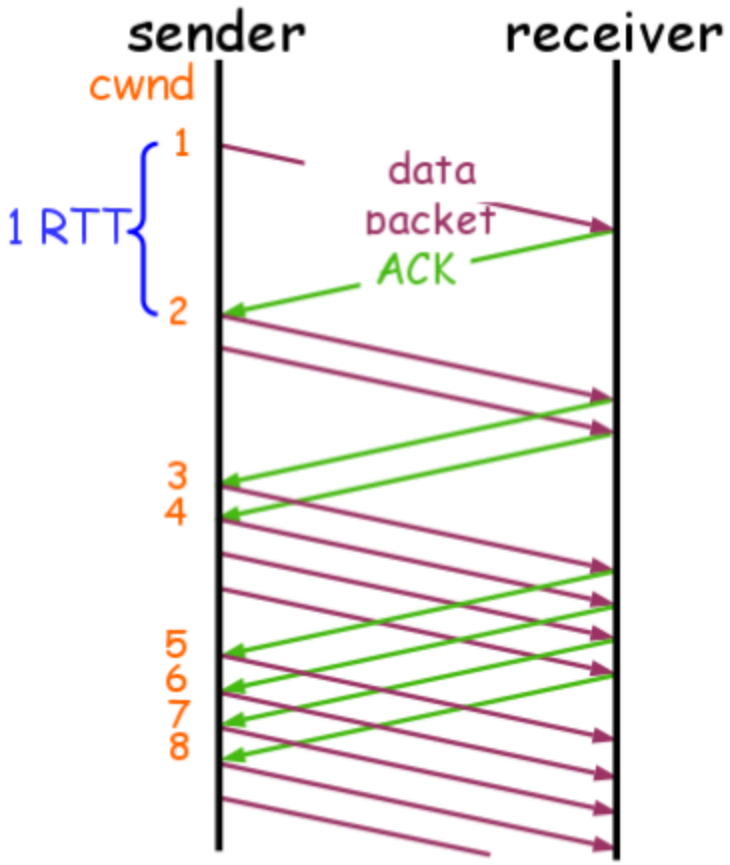
Ci basiamo sempre sulla cwnd è una finestra in cui abbiamo pacchetti di cui aspettiamo ACK (gialli) e pacchetti non inviati (blu). Se tutta la cwnd è "gialla" allora dobbiamo fermarci.

$$Rate = \frac{cwnd}{RTT} = \frac{bytes}{sec}$$

A ogni RTT inviamo dei pacchetti si cambia la grandezza della cwnd per poter "controllare" la congestione.

Slow Start - Prima fase

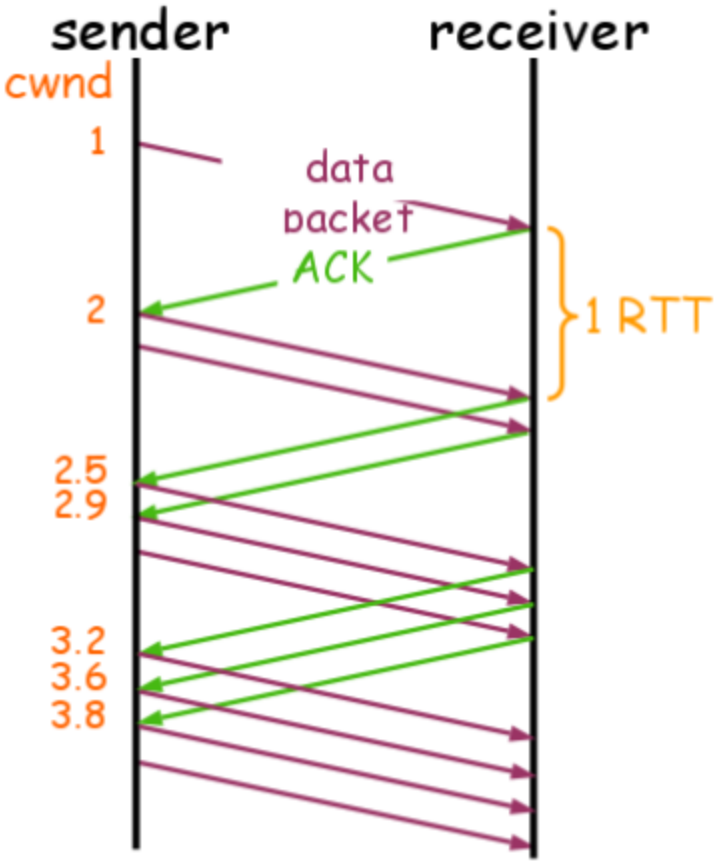
Il primo metodo per gestire la congestione è quello di iniziare lentamente l'invio di pacchetti e incrementare il rate di invio esponenzialmente, si duplica a ogni RTT. All'inizio **cwnd** (grandezza della finestra di invio) sarà 1. Questo fin quando non avviene la prima perdita.



Quando iniziamo ad avere ACK che non sono nuovi, abbiamo un passaggio da SS a CA. Questo perchè molto probabilmente stiamo sovraccaricando troppo il server che non riesce a gestire i dati che abbiamo inviato, può essere anche che abbiamo riempito troppo i buffer di qualche nodo, in ogni caso torna indietro l'ultimo pacchetto ricevuto in ordine.

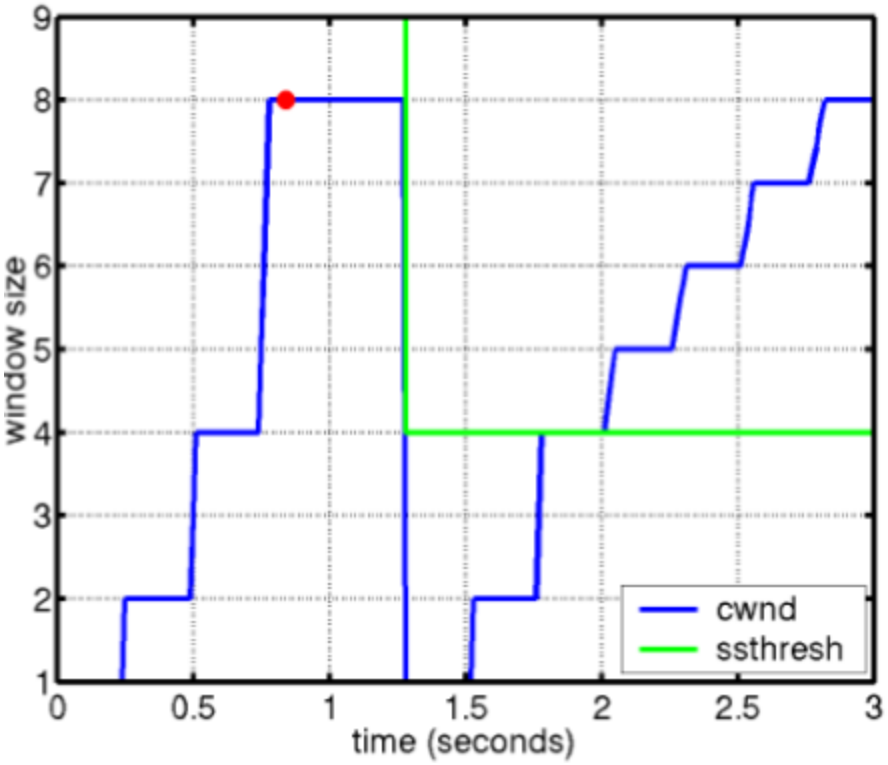
Calcolo: **cwnd si incrementa a ogni RTT di 1. (cwnd += 1)**

Congestion Avoidance - Seconda fase



Dobbiamo assumere che la perdita del pacchetto sia per la congestione, lo capiamo dalle ritrasmissioni dopo 3 duplicati oppure dal timeout.

Calcolo: **cwnd si incrementa a ogni RTT di 1/cwnd. (cwnd += 1/cwnd)**



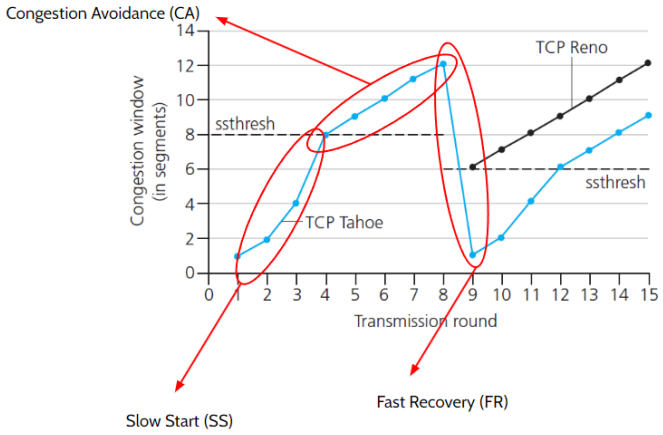
Quando un pacchetto viene perso, si cambia da SS a CA alla metà di dove si è perso il pacchetto, quindi in questo esempio se abbiamo perso un pacchetto quando la finestra era grande 8 allora alla ripartenza ci fermiamo a 4 e dopo proseguiamo in modo più moderato per evitare il problema della congestione.

Abbiamo **due algoritmi** per gestire la ripartenza e quindi la dimensione della cwnd:

- TCP **Reno**: per la ripartenza partiamo dalla metà di dove ci siamo interrotti
- TCP **Tahoe**: per la ripartenza partiamo da 1

Fast Recovery - Terza fase

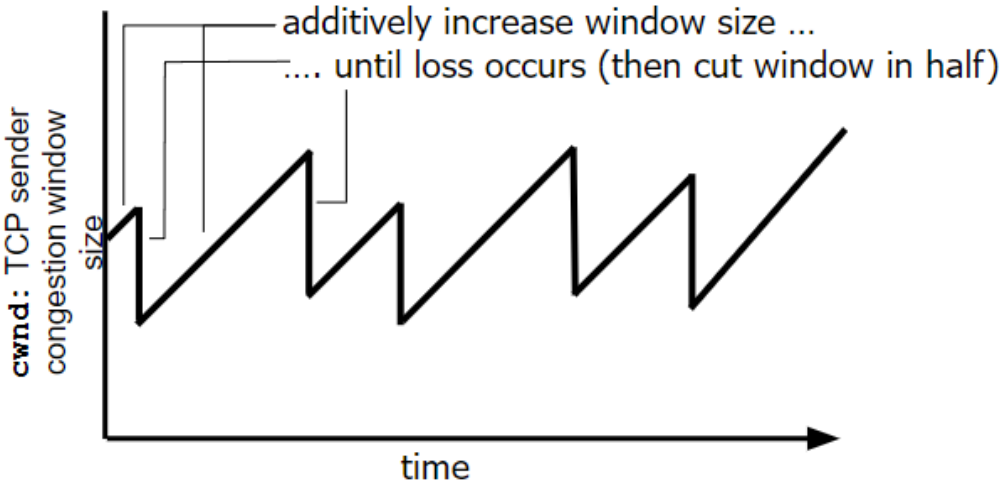
Quando ci troviamo a dover ritrasmettere passiamo sempre per la FR, caliamo drasticamente la dimensione dalle cwnd e riprendiamo a seconda dell'algoritmo scelto.



Quindi iniziamo con la **SS** (slow Start), passata una precedente soglia passiamo alla **CA** (Congestion Avoidance) e se perdiamo ancora un pacchetto (quindi riceviamo 3 ACK) dobbiamo usare la **FR** (Fast Recovery).

Abbiamo quindi:

- **Incremento additivo**: se tutto va bene si continua a incrementare di un pacchetto (CA)
- **Decremento moltiplicativo**: se qualcosa va storto (si perde un pacchetto) si dimezza cwnd dopo la ripartenza

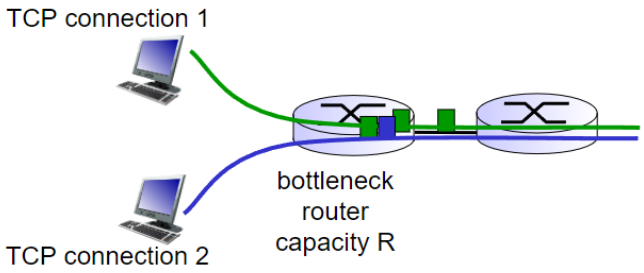


Il grafico tende nelle comunicazioni ad avere questo andamento, dato che ci sono perdite continue bisogna adeguare il flusso degli invii e quindi controllare la congestione.

$$Avg\ Throughput = \frac{3W}{4RTT}$$

TCP Fairness

Vogliamo che tutti gli host abbiano lo stesso rete di invio, come possiamo fare?



Avviene in maniera abbastanza automatico grazie al fatto che se un host invia troppo il suo rate viene dimezzato e al contrario se è troppo poco si incrementa.