PROTOCOLLI DEL LIVELLO TRASPORTO

Mattia Dutto ~ January 2021

UDP

~ INTRODUZIONE ~

UDP sta per **User Datagram Protocol** che è definito dal RFC 768.

Con questo protocollo ritroviamo una forte iterazione tra il livello di trasporto e quello di rete.

~ COSA FA? ~

- Impacchetta il dato
- Aggiunge l'header
- Passa il nuovo pacchetto al livello sottostante

UDP

~ MOTIVAZIONI PER PREFERIRLO A TCP ~

1) Controllo più fine a livello applicazione sui dati inviati e ricevuti. UDP incapsula il dato e lo invia direttamente, creando una linea diretta con il destinatario. Si dice che UDP è senza fronzoli.

2) Non vi è richiesta di connessione, posso inviare i dati dopo aver creato il segmento.

3) Nessuno stato della connessione.

4) Intestazione più piccola.

UDP

~ SEGMENTI ~

L’header del protocollo UDP è formata da segmenti.

Ogni segmento ha dimensione di 2Byte.

Ogni header ha 4 segmenti.

Nell’header ritroviamo tutte le possibilità del protocollo.

INTESTAZIONE UDP

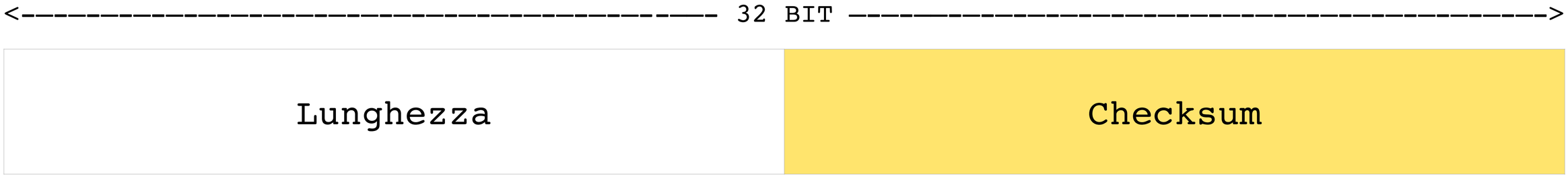
<-------------------------------- 32 BIT -------------------------------->

Numero di porta di origine	Numero di porta di destinazione
Lunghezza	Checksum
Dati	

PORTE: 16bit ci permettono di effettuare il multiplexing / demultiplexing del messaggio.

LUNGHEZZA: 16bit si intende la lunghezza in Byte del campo dati più l'intestazione.

INTESTAZIONE UDP



CHECKSUM: Controllo degli errori. UDP effettua il complemento a 1 delle 3 parole (porta mittente, porta destinatario e lunghezza). Per non aver corruzione devo ottenere 16 uni.

UDP

~ ESEMPIO CHECKSUM ~

Abbiamo i seguenti tre segmenti:
1) 0110011001100000
2) 0101010101010101
3) 1000111100001100
Quello che andremo a fare è: 1) + 2) e poi al risultato sommeremo 3).

Si somma al risultato l'eventuale riporto prima di procedere allo step successivo.

UDP

~ ESEMPIO CHECKSUM ~

Segmento 1) + 2)

```
    0110011001100000 +
    0101010101010101 =
    ____________________
    1011101110110101
```

Somma risultato precedente + 3)

```
    1011101110110101 +
    1000111100001100 =
    ____________________
    0100101011000001 e 1 di riporto
```

UDP

~ ESEMPIO CHECKSUM ~

Somma risultato precedente + 3)

```
    1011101110110101 +
    1000111100001100 =
    ____________________
    0100101011000001 e 1 di riporto
```

Sommo il riporto e trovo il valore del checksum

```
    0100101011000001 +
                   1 =
    ____________________
    0100101011000010 -> Valore del checksum
```

TCP

~ INTRODUZIONE ~

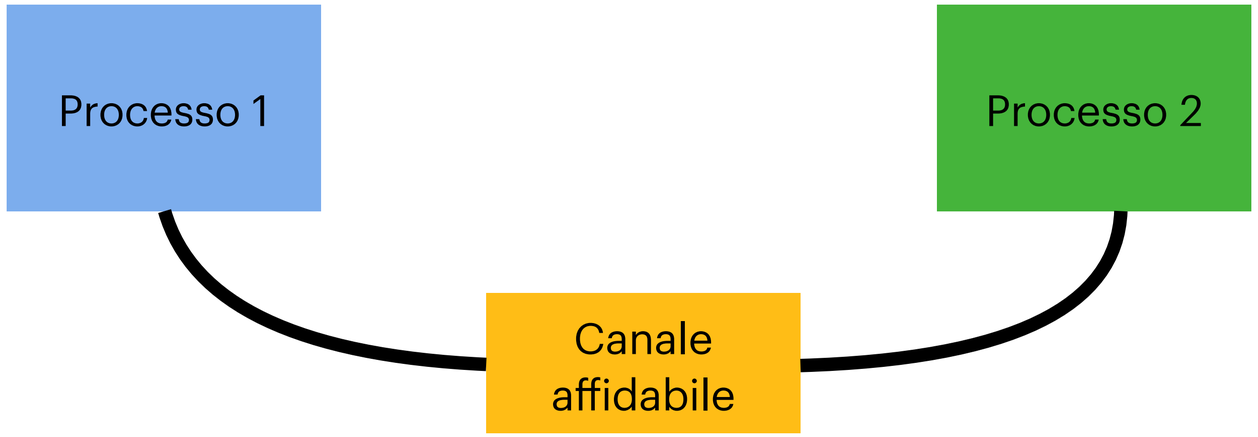
TCP sta per **Transmission Control Protocol** che è definito dal RFC 793.

~ CARATTERISTICHE ~

- Orientato alla connessione
- Affidabilità
- Controllo della congestione

TCP

~ AFFIDABILITA' ~



Affidabilità pura: creo il canale di trasporto
mi disinteresso dei livelli sottostanti.

TCP

~ TIPI DI ERRORE ~

Sono errori dovuti alla rete e sono:

1) **corruzione dei bit**:
   - arrivano al contrario
   - non riesco a leggere il valore

2) **perdita di bit**

3) **disordine dei bit**: l'ordine dei bit è diverso da quello originario.

TCP

~ CORRUZIONE DEI BIT ~

I bit arrivano tutti avrò:

- o un bit corrotto
- o non riuscirò a leggerlo

Dobbiamo comunicare con il mittente e dirgli:

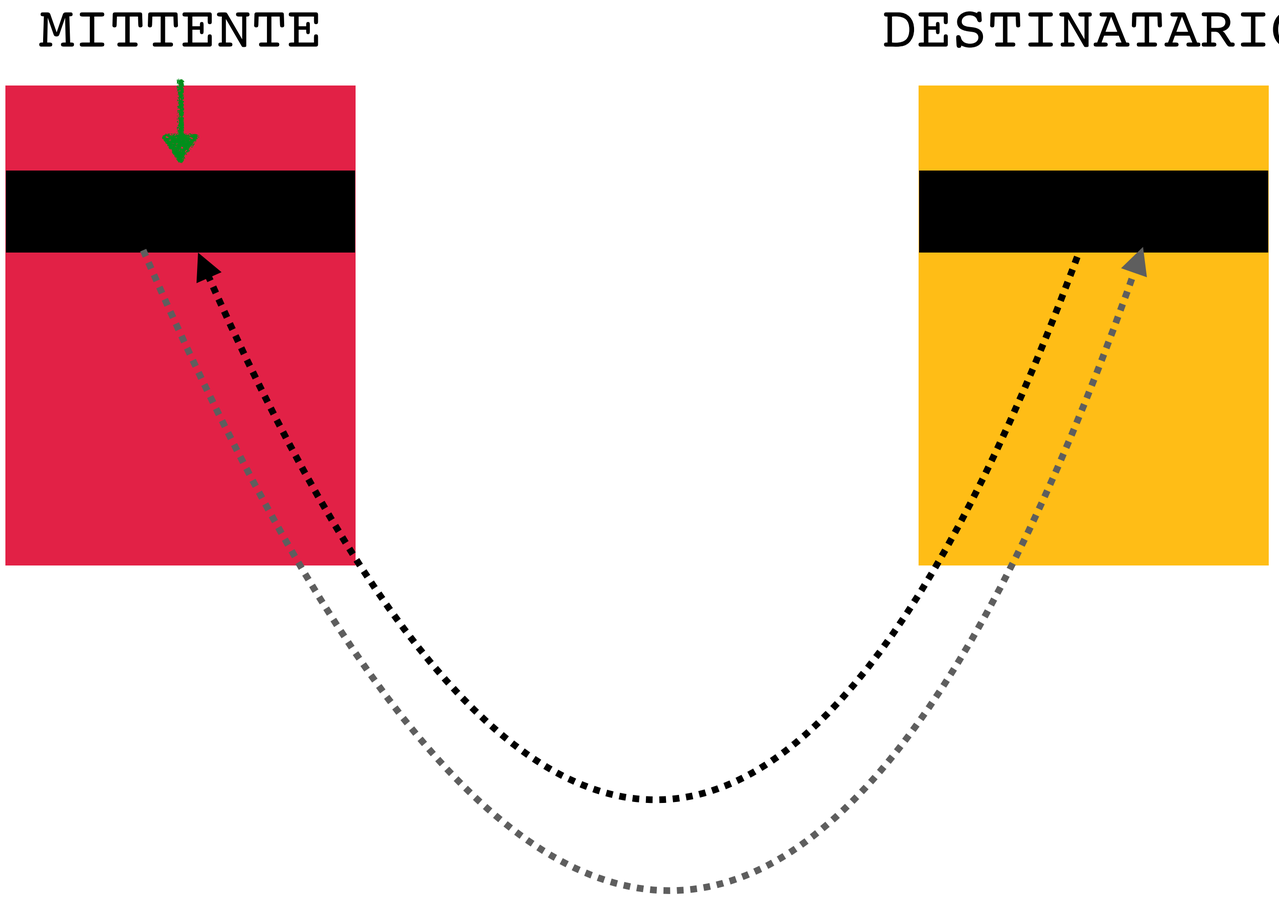
A) Sequenza **corretta** -> mandami un nuovo pacchetto

B) Sequenza **non corretta** -> mandami di nuovo il pacchetto

Ci si basa sui protocolli di ritrasmissione (ARQ)

TCP

~ PROTOCOLLI ARQ ~



TCP

~ PROTOCOLLI ARQ ~

I protocolli ARQ devono offrire/avere:

A) Rilevamento degli errori

B) Deve poter leggere il feedback del destinatario (ricevente del messaggio)

~ FEEDBACK DEL DESTINATARIO ~

1) Pacchetto **ACK**

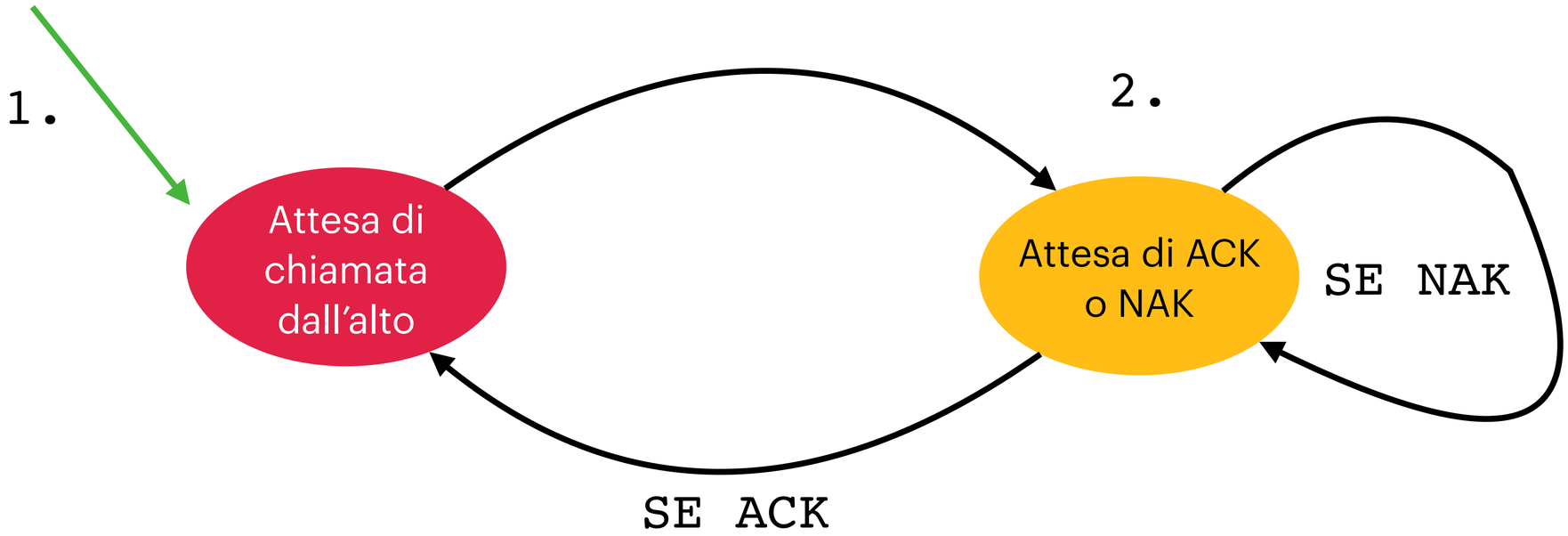
2) Pacchetto **NAK**

Ogni volta che il mittente inoltra un messaggio sul canale trasmissivo il destinatario risponde con ACK o NAK.

C) Ritrasmissione del pacchetto: possibilità di rimandare lo stesso pacchetto.

TCP

~ MITTENTE ~

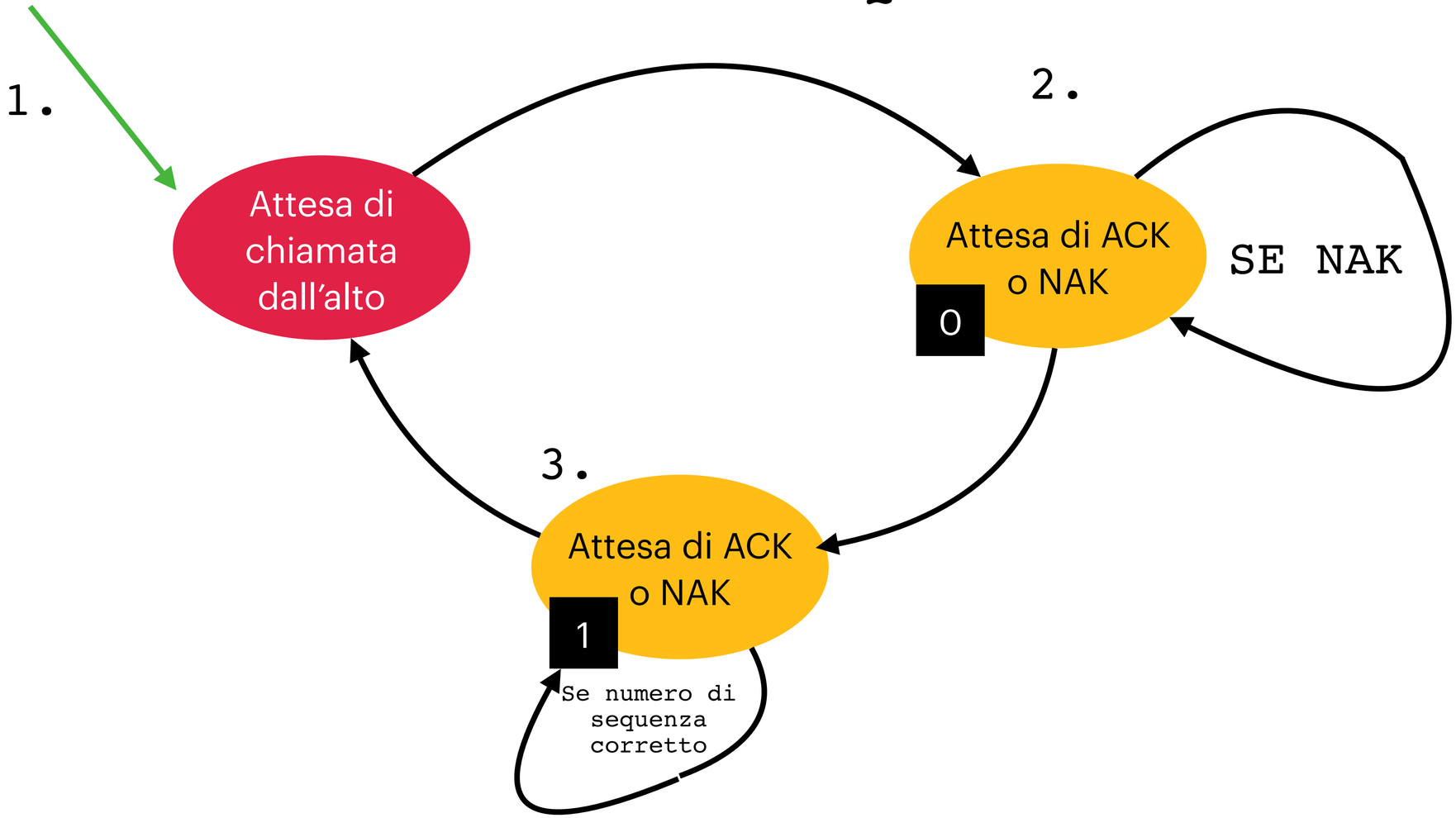


TCP

~ DESTINATARIO ~

TCP

~ NUMERO DI SEQUENZA ~



TCP

~ PERDITA DEI BIT + CORRUZIONE DEI BIT ~

Due possibili casi:

- Perdiamo dei bit all’interno del pacchetto
- Perdiamo l’intero pacchetto.

Per risolverli utilizziamo due contatori:

- Un contatore per tener traccia delle trasmissioni
- Un timer per tener traccia del tempo d’attesa per ottenere una risposta

TIMER = Tempo d’andata + Tempo di ritorno + X

TCP

~ PERDITA DEI BIT + CORRUZIONE DEI BIT ~

X Dev’essere calcolato in modo corretto, se no rischio un sovraccarico nella rete o uno stallo nella comunicazione.

X dovrà tener conto dei ritardi della rete e per farlo 2 strategie:

1) Ci basiamo sui ritardi dei pacchetti già mandati
2) Mi baso su quelle che sono definite come “Costanti di ritardo” introdotte dai mezzi fisici.

TCP

~ PERDITA DEI BIT ~

Se perdo uno o più bit:

- Si esegue un controllo / confronto con il valore di lunghezza contenuto all'interno del segmento aggiuntivo
- Il destinatario manda un NAK se non coincidono i valori
- Dopo il NAK del destinatario il mittente ritrasmette il pacchetto.

Il disordine dei bit viene risolto come la corruzione dei bit

TCP

~ UN PO’ DI CHIAREZZA ~

TCP ha più di un RFC, si è evoluto con l’evoluzione della rete.
Due proprietà chiave:

- AFFIDABILITA’
- ORIENTATO ALLA CONNESSIONE

A livello di RFC TCP non garantisce il throughput utilizza un’altro protocollo.
TCP garantisce la sicurezza grazie all’SSL.
Il timing non viene gestito da TCP ma da un protocollo che gestisce la trasmissione ma non la ritrasmissione.
Il controllo della congestione viene fornito dai livelli sottostanti.

TCP

~ CONCETTI ~

Comunicazione **FULL-DUPLEX**: la comunicazione avviene in entrambe le direzioni, anche contemporaneamente, sullo stesso binario o canale di comunicazione.

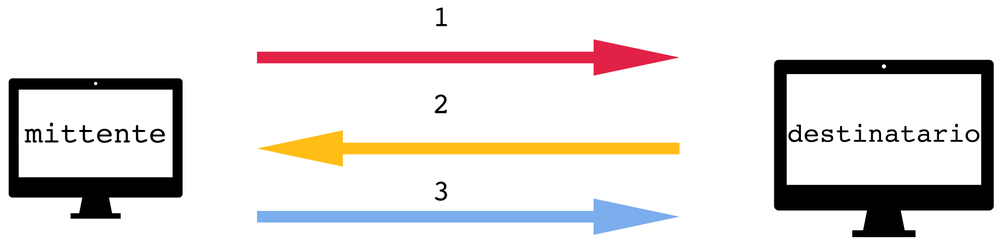
Comunicazione **PUNTO A PUNTO**: da singolo mittente a singolo destinatario.

TCP

~ HANDSHAKE a 3 VIE ~

Sono 3 i pacchetti che vengono scambiati tra il processo mittente e il processo destinatario servono per stabilire la connessione. I processi si scambiano 3 segmenti speciali:

1) Da **mittente** a destinatario
2) Da destinatario a **mittente**
3) Da **mittente** a destinatario



Tra i parametri di connessione vi è anche la dimensione massima di un segmento (MSS).

TCP

~ INVIO DEI DATI ~

Per inviare i dati si passa attraverso il buffer, posso inviare anche se non ho il segmento completo. Anche i segmenti speciali passano dal buffer TCP.

Questo sistema in linea di principio scambia dei segmenti senza perdite.

INTESTAZIONE TCP

<-------------------------------- 32 BIT -------------------------------->

Numero di porta di origine							Numero di porta di destinazione
Numero di sequenza							
Numero di acknowledgement							
Lunghezza intestaizone	U R G	A C K	P H S	R S T	S Y N	F I N	Finestra di ricezione
Checksum							Puntatore ai dati urgenti
Opzioni							
Dati							

TCP

<------------------------------------ 32 BIT ------------------------------------>

Numero di sequenza
Numero di acknowledgement

Questi due campi vengono utilizzati per poter ottenere il trasferimento affidabile.

Lunghezza intestaizone	URG	ACK	PHS	RST	SYN	FIN	Finestra di ricezione

LUNGHEZZA INTESTAZIONE: variabile per via del campo opzioni.

FINESTRA DI RICEZIONE: utilizzata per il controllo del flusso

TCP
~ FLAGS ~

URG & **PHS**: collegati al puntatore ai dati urgenti

ACK: L'unico flag che ha senso, serve per validare il pacchetto di ritorno del destinatario.

RST, **SYN** & **FIN**: utilizzati durante l'instaurazione e la fine della connessione.

Checksum	Puntatore ai dati urgenti
Opzioni	

CHECKSUM: utilizzato per il rilevamento degli errori.

PUNTATORE AI DATI URGENTI: creato per dare priorità a un segmento invece che ad un altro. Causava problemi quindi non utilizzato.

OPZIONI: utilizzato ad esempio per rettare l'MSS. Ha lunghezza variabile.

TCP

~ RTO ~

RTO = Retransmission Time Out Ci permette di definire dopo quanto andiamo a rispedire il nostro pacchetto.

Dobbiamo calcolare il RTT che è il tempo intercorso tra quando spedisco a quando leggo il feedback del pacchetto di ritorno dal destinatario.

Questo RTO viene calcolato a campioni non per ogni segmento spedito. Quando lo valuto calcolo il SAMPLE_RTT.
Da qui dobbiamo effettuare una media terminata.

TCP

~ RTO ~

ESTIMATED_RTT = (1-α)*ESTIAMATED_RTT + α*SAMPLE_RTT

α = Non viene definita dal RFC, ma viene preso come valore 0.125

DEV_RTT = Deviazione standard da quanto varia dalla realtà.

= (1-β)*DEV_RTT + β*|SAMPLE_RTT - ESTIMATED_RTT|

β = 0.25 valore consigliato.

RTO = ESTIMATED_RTT + 4 * DEV_RTT

RTO è ricorso viene calcolato e aggiornato ogni volta che andremo ad estrarre un SAMPLE_RTT.

TCP

~ CONTROLLO DEL FLUSSO ~

Si effettua in quanto possiamo avere diverse velocità di trasmissione TCP e di lettura dei segmenti dal buffer di TCP da parte del livello applicazione.

Servizio relativo per il livello applicativo.

Io potrei ottenere due diverse velocità:
- di trasmissione
- di lettura dal buffer tcp.

Questo potrebbe causare overflow.

TCP

~ CONTROLLO DELLA CONGESTIONE ~

Servizio per il livello **IP**, vogliamo evitare l'intasamento della rete.

TCP utilizza la finestra di ricezione per evitare tutto ciò, essa è definita a livello protocollare come **rwnd**.

rwnd: è un segmento dinamico dell'intestazione TCP, ci dice quanto spazio libere vi è nel buffer di ricezione in byte.

TCP

~ CONTROLLO DELLA CONGESTIONE - Lato destinatario ~

RcvBuffer: grandezza del buffer del ricevente

Per valutarla ho due variabili da considerare

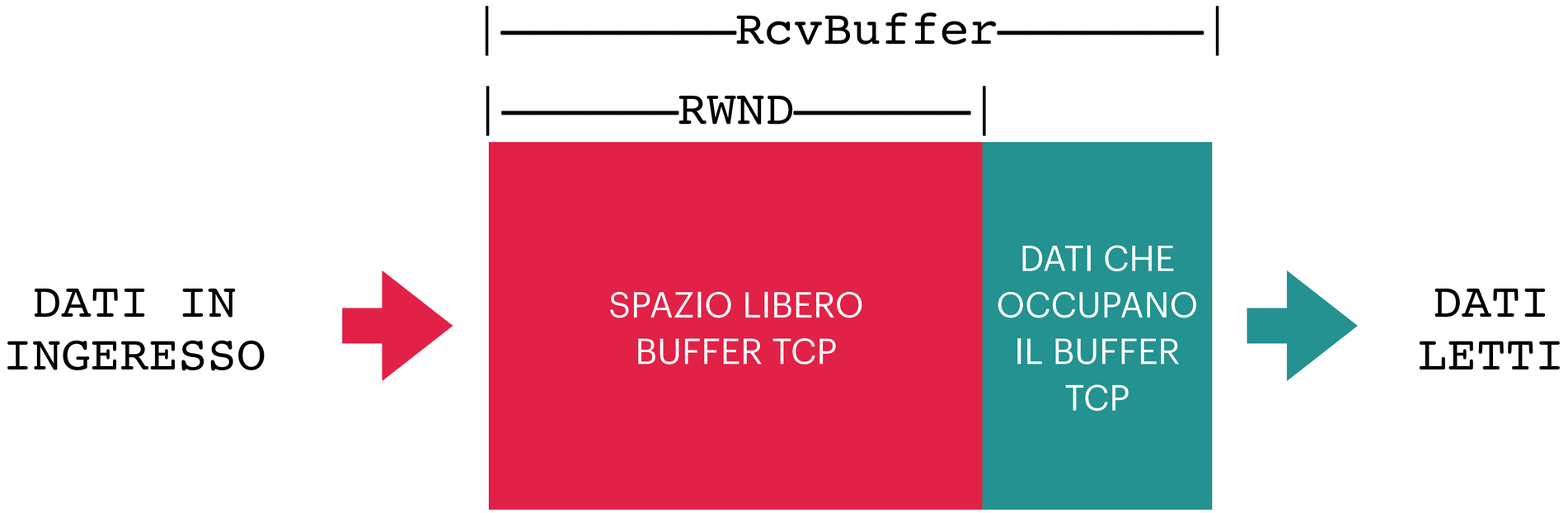
- Last Byte Read: ultimo byte letto
- Last byte RCVD: ultimo byte ricevuto dal ricevente da parte del mittente

Durante la comunicazione questo dovrà esser sempre vero:

$$LastByteRCVD - LastByteRead \leq RcvBuffer$$

TCP

~ CONTROLLO DELLA CONGESTIONE - Lato destinatario ~



TCP

~ CONTROLLO DELLA CONGESTIONE - Lato mittente ~

Bisogna tenere in considerazione 2 variabili

- Last Byte Sent: ultima sequenza inviata
- Last Byte Acked: ultima sequenza di cui ho ricevuto ACK

La regola dell'invio è:

$$LastByteSent - LastByteAcked \leq rwnd$$

Entrambe le regole devono essere sempre verificate.

TCP

~ CONTROLLO DELLA CONGESTIONE ~

Insieme all'affidabilità sono i maggiori problemi di TCP.

Caso ideale: - mezzi fisici con buffer illimitati
- grafo completamente connesso

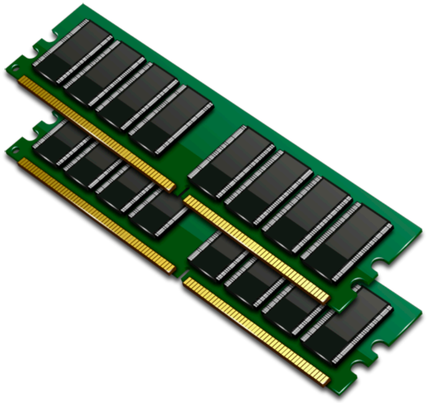
Ma abbiamo i seguenti problemi:

1) buffer limitati
2) router deve aver letto tutto il pacchetto prima di elaborarlo e inoltrarlo.

Saturazione vi è stata una richiesta di incapsulamento ma non è possibile salvare tali dati nel router.

TCP

~ CONTROLLO DELLA CONGESTIONE ~



Come evitare il problema:
All’inizio era possibile ampliare la dimensione dei buffer, in seguito hanno pensato di distribuire il traffico su più nodi.
Ma questo causa problemi.

La miglior soluzione è: controllare le sequenze e le priorità di esse.

TCP

~ CONTROLLO DELLA CONGESTIONE ~

TCP non può garantire il throughtput.

Possiamo dividere il controllo della congestione in due gruppi:

- controllo della congestione **end-to-end:** TCP deve farsi carico di gestirla lui.

- controllo della congestione **assistito dalla rete**: vuol dire che è la rete stessa a fornire un feedback. Questo è possibile solo nelle reti proprietarie. La rete TCP/IP non è in grado di dare feedback.

TCP

~ CONTROLLO DELLA CONGESTIONE ~

Solitamente ci si pone i 3 quesiti della congestione:

1) **Come fa il mittente tcp a variare la velocità di trasmissione?**

Questo viene fatto introducendo una nuova variabile, finestra di congestione: **cwnd**. Mi impone un vincolo alla velocità di trasmissione. Anche in questo caso vi è una disuguaglianza da rispettare:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min \{\text{cwnd}, \text{rwnd}\}$$

Nelle reti più semplici è il cwnd, nei casi più complessi ho bisogno di entrambi i valori.

TCP

~ CONTROLLO DELLA CONGESTIONE ~

2) **come fa il mittente TCP a percepire la congestione sulla rete?**

Il segnale per TCP è “l’**evento di perdita**”.
Ho due possibile casi:
- Velocità di trasmissione troppo alta
- Ho i buffer in overflow

Ogni evento fa partire una serie di procedimenti per risolvere il problema.

TCP

~ CONTROLLO DELLA CONGESTIONE ~

3) **Quale tipo di algoritmo dovrà utilizzare il protocollo TCP per gestire i problemi di congestione, nel caso end-to-end?**

Algoritmo autogestito: è automatico e a gestione di TCP chiamato anche auto-temporizzato.
Gestisce in modo automatico lo scalare della velocità.

Argomento collegato al problema del flusso e della perdita.

TCP

~ CONTROLLO DELLA CONGESTIONE ~

Mi restano aperti i seguenti problemi:

- Non deve essere la mia comunicazione ad abbassare la velocità
- Come faccio a capire quanta banda ho disponibile
- Come faccio a gestire i mittenti TCP

TCP

~ ALGORITMO DI CONTROLLO DELLA CONGESTIONE ~

Viene definito dall'RFC 5681.

Gli obbiettivi sono:

- Gestire la velocità di trasmissione dei segmenti in modo dinamico
- Deve accorgersi di problemi di congestione
- Gestito in maniera autonoma

L'algoritmo viene diviso in 2 blocchi principali:

1) Slow start
2) Congestion Avoidance
3) Fast recovery

TCP

~ SLOW START ~

La CWND viene impostata a un MSS:

Esempio:

- MSS 500byte
- RTT 20ms
- MSS/RTT = Velocità di trasmissione 25kbyte/s

Cercherò di aumentare la velocità di trasmissione man mano che il destinatario riceverà il mio pacchetto.

Ogni volta che ricevo un ACK per un segmento inviato la mia CWND viene incrementata di un MSS.

TCP

~ SLOW START ~

Aumento o fino a quando non ho errori o fino a quando non saturo la banda.

Quando succede un problema / evento di perdita, al segmento successivo viene riportata la CWND a **1 MSS**.

Viene introdotta un nuova variabile statica di soglia: **SSTHRES**, come valore è la cwnd di quando ho avuto l'errore divisa 2.

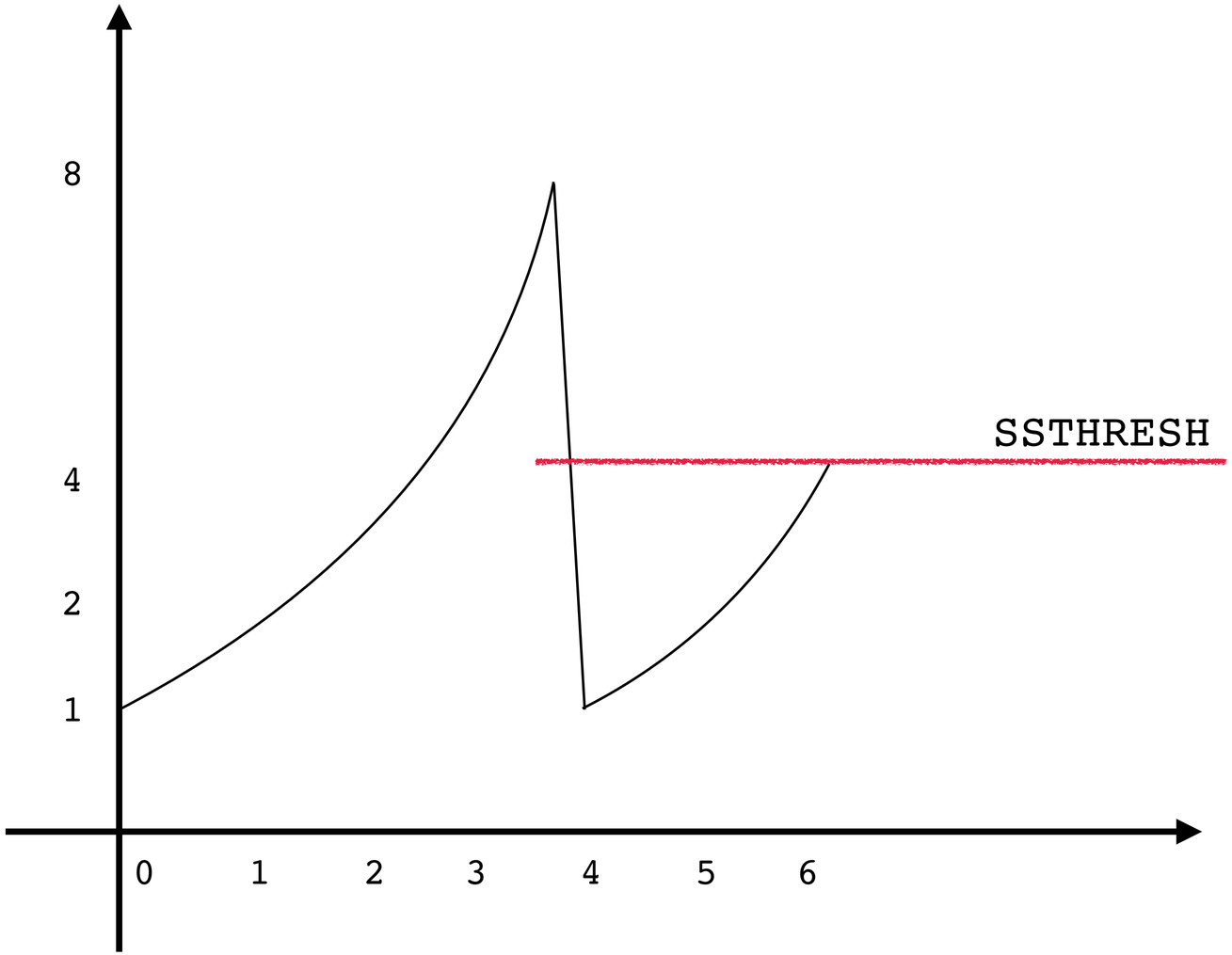
Dimezzo la velocità di trasmissione.

Se mi ritrovo in un evento di perdita passo allo stadio 2.

Abbiamo uno stadio Slow Start e uno Congestion Avoidance.

TCP

~ SLOW START ~



TCP

~ CONGESTION AVOIDANCE ~

I due algoritmi si differiscono per la funzione:
1) Funzione di arrivare subito all'obbiettivo
2) Funzione conservativa

Caso evento di perdita prima della soglia si riporta la CWND a 1 MSS.

SSTHRESH: CWND / 2 stesso valore di slow start.
Si passa da una situazione esponenziale a una lineare. Ogni volta verrà aggiunto 1 all'MSS.

TCP

~ CONGESTION AVOIDANCE ~

La soglia è sempre la stessa se riesco da arrivare la soglia ritorno in slow start.

Se subisco un evento di perdita rimango alla fase 2 se invece arrivo alla soglia, incrementiamo la velocità di 1.

TCP

~ FAST RECOVERY ~

A questo punto si pone un problema perché per ogni pacchetto inviato deve avere un feedback.

Può essere che uno dei pacchetti spediti non arrivi al destinatario.

Io allora avviso il mittente inviando sempre lo stesso n° di ACK.

TCP

~ FAST RECOVERY ~

Se io ho un problema di perdita.

Mi continua ad inviare l'ultimo valore di ACK che ho.
Si dice che si usa la tecnica dell'ACK duplicato.

ACK Duplicato: se io visualizzo un triplo ACK duplicato, 3 volte lo stesso ACK, si può considerare un evento di perdita.

TCP

~ FAST RECOVERY ~

Allora effettuo la ritrasmissione rapida spedisce i pacchetti e dopo il destinatario gli ACK dei pacchetti successivi.

Mi restituisce il numero di ACK del pacchetto di cui sono in attesa.

La parte di fast recovery serve se visualizziamo un evento di perdita e visualizzazione un triplo ACK duplicato sia in slow start sia in congestion avoidance passiamo in fast recovery.

TCP

~ FAST RECOVERY ~

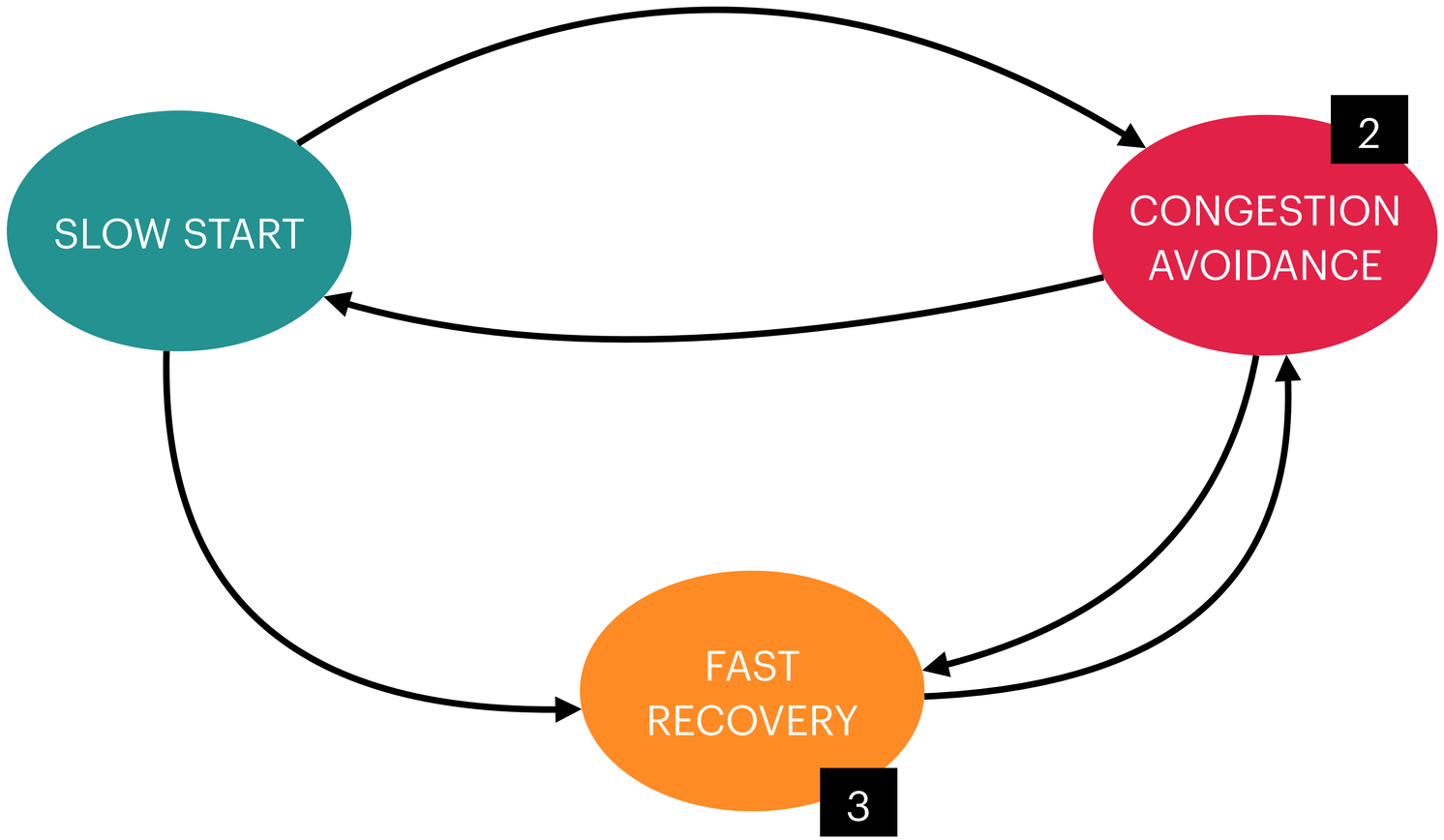
Una volta entrati in fast recovery si imposta la cwnd = 1 MSS.

Aumento la cwnd di 1 per ogni ACK duplicato visualizzato e a quel punto rientro in un altro stadio.

Se siamo in congestion avoidance e si verifica il triplo ACK duplicato vado in fast recovery e dopo ritorno in congestion avoidance.

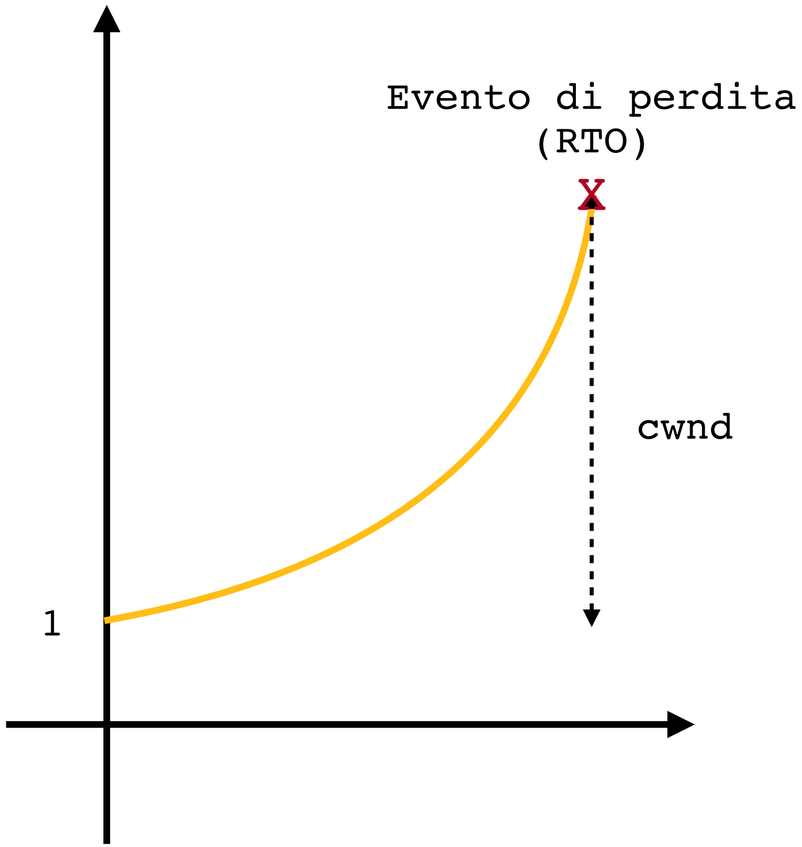
TCP

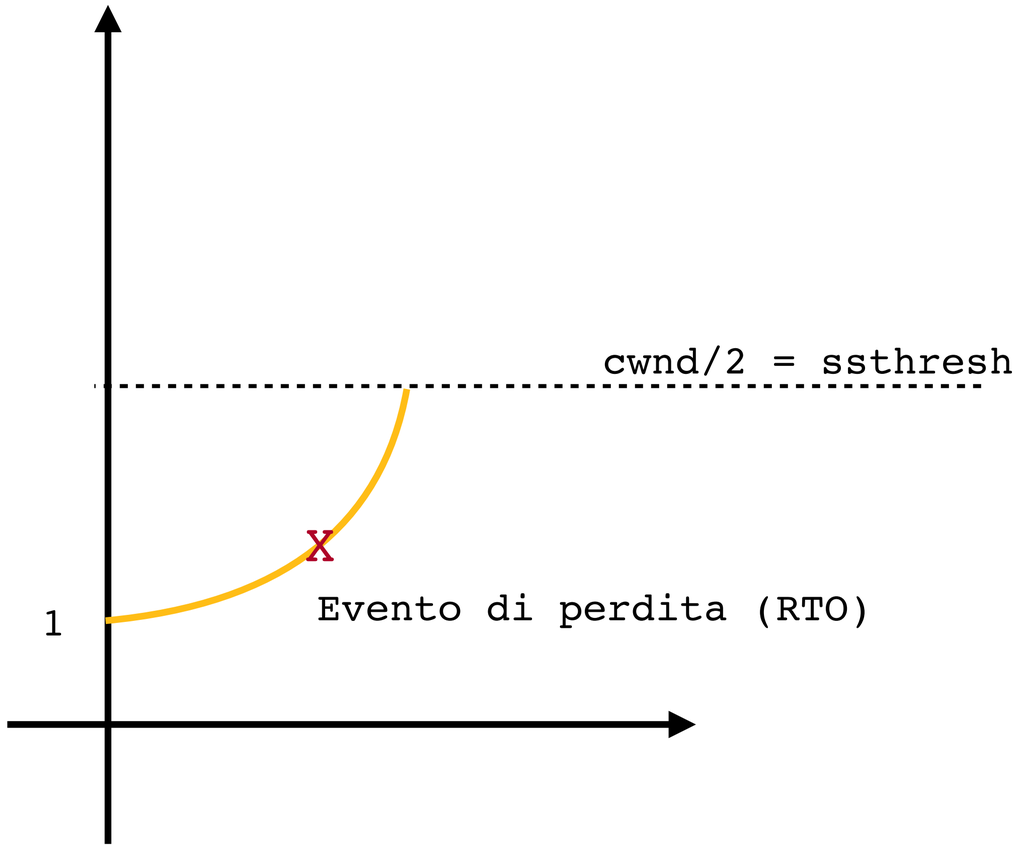
~ ALGORITMO DI GESTIONE DELLA CONGESTIONE ~



TCP

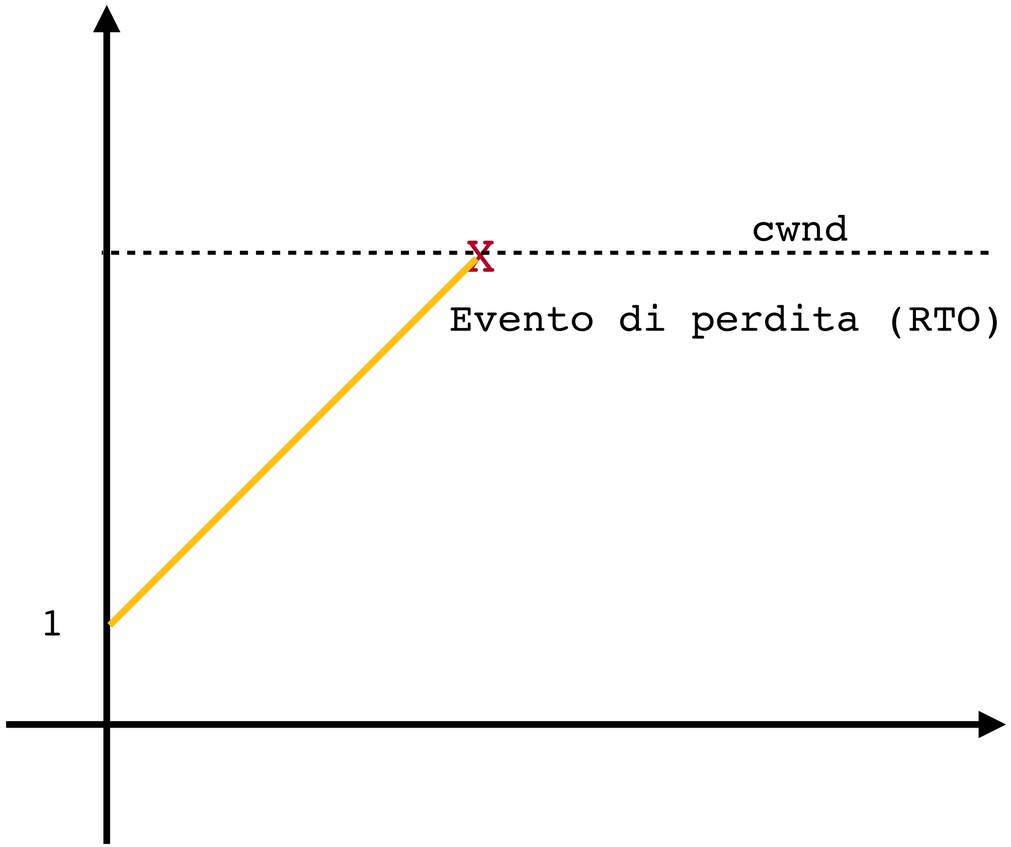
~ SLOW START ~

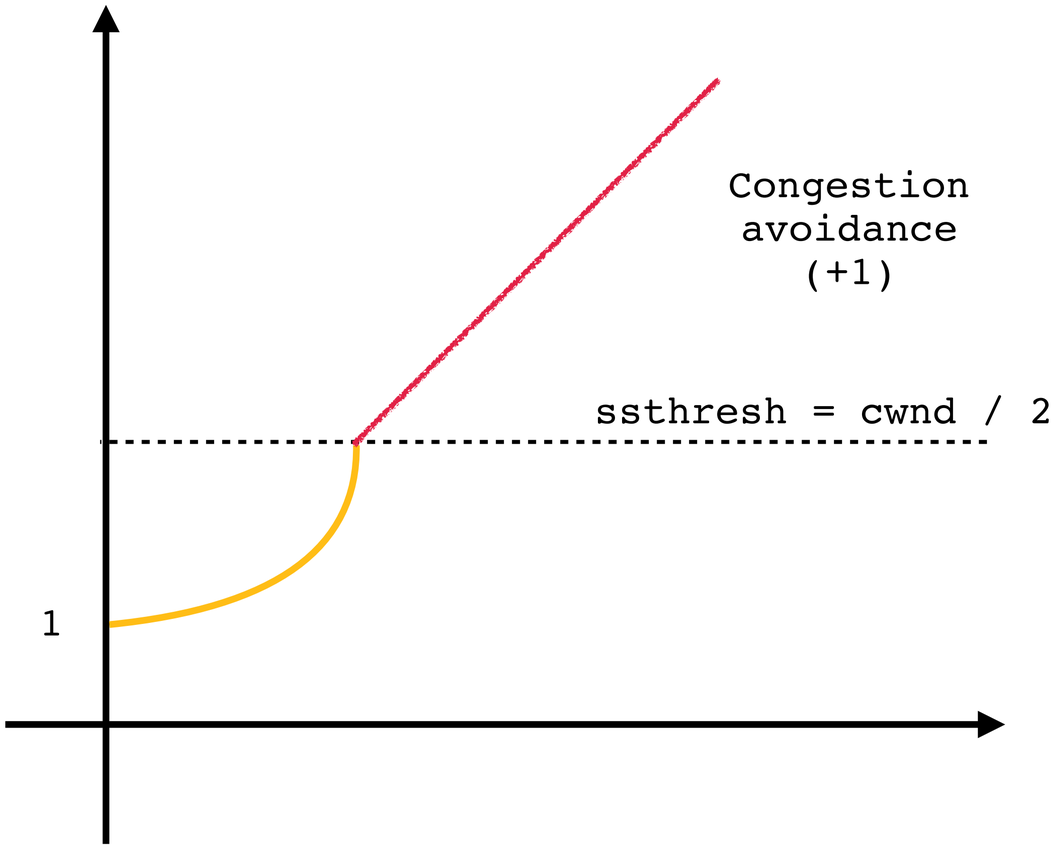




TCP

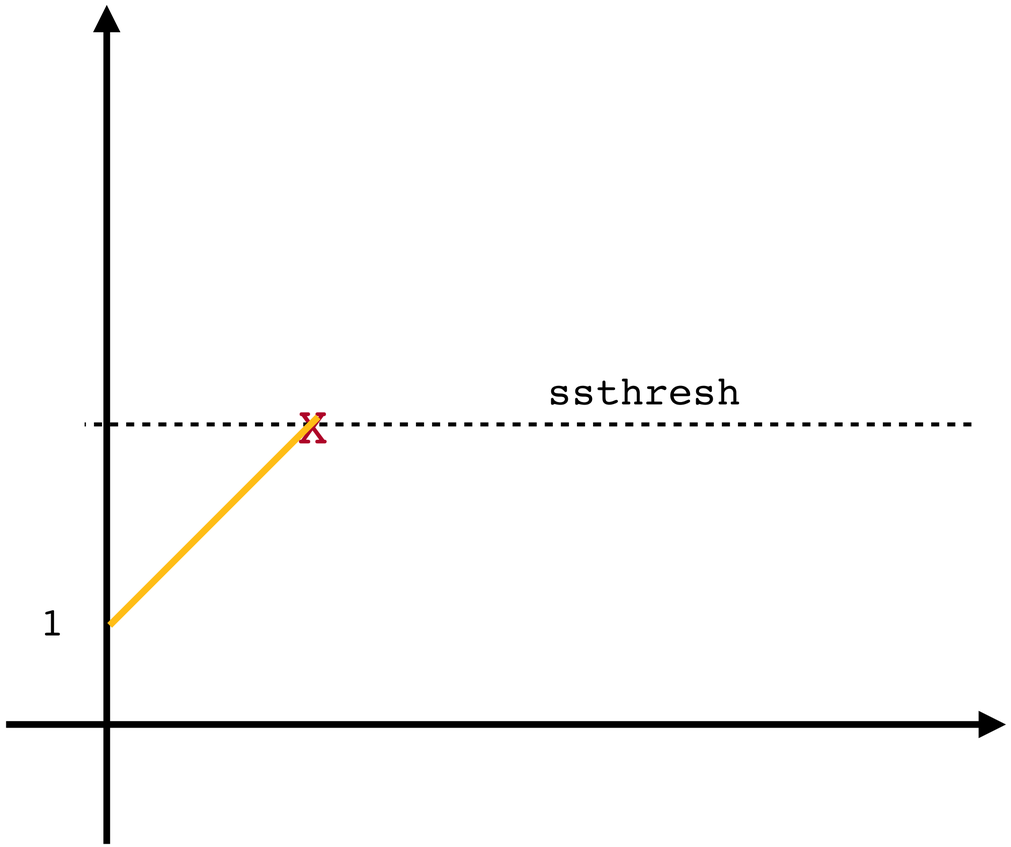
~ DA SLOW START A CONGESTION AVOIDANCE ~

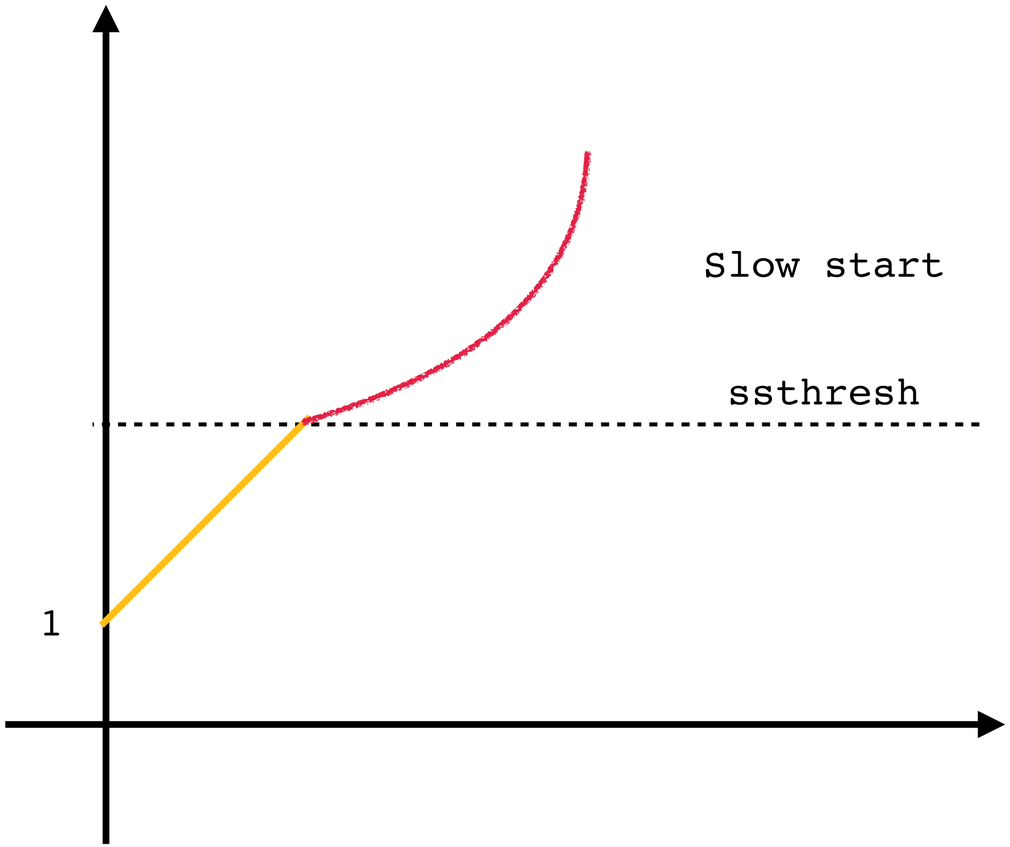




TCP

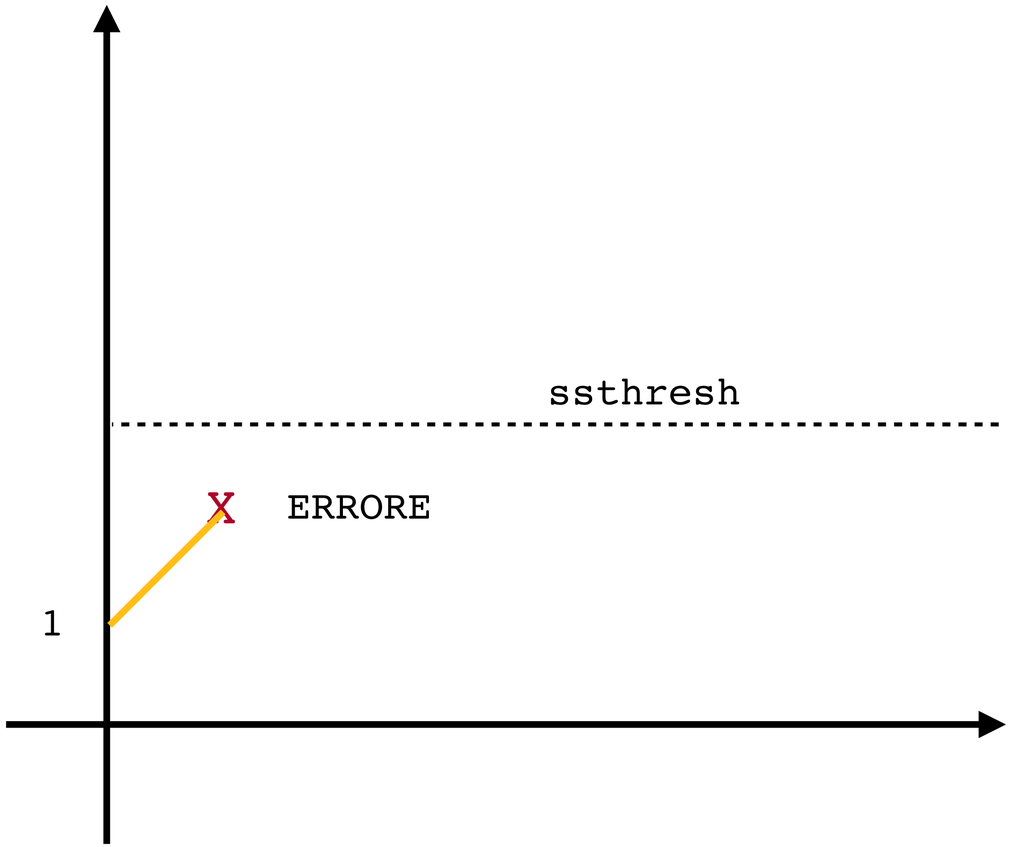
~ DA CONGESTION AVOIDANCE A SLOW START ~

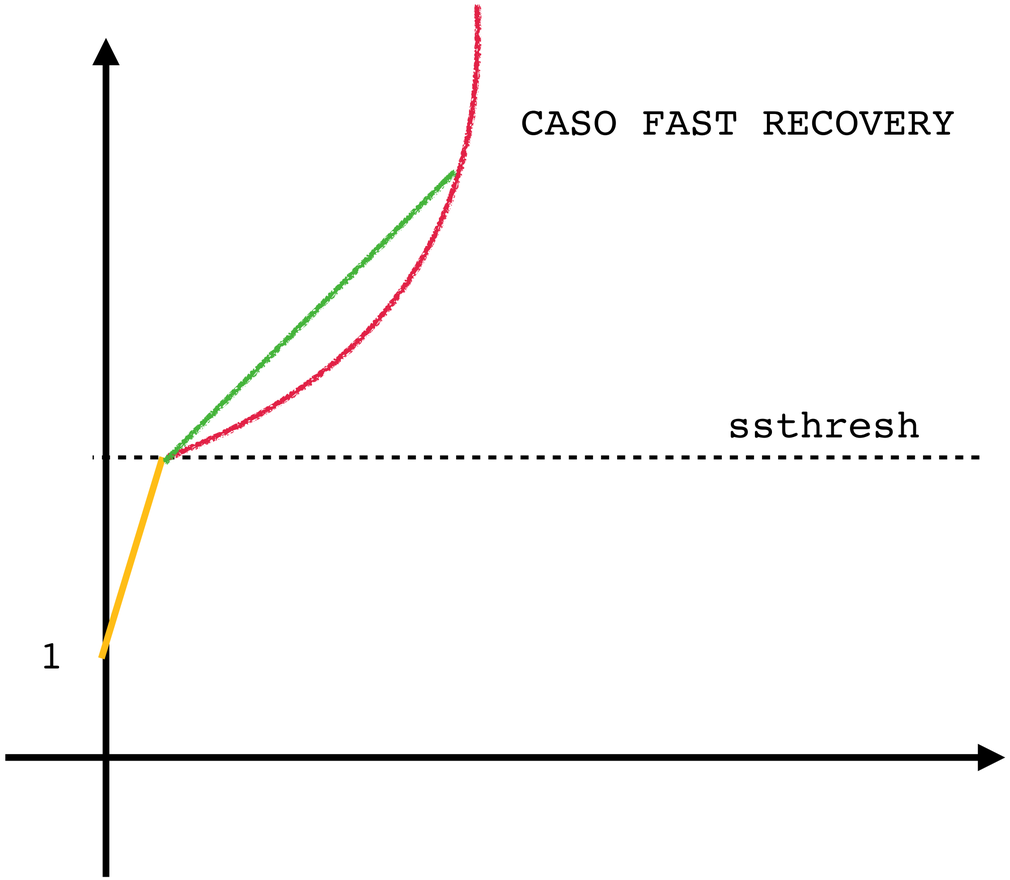




TCP

~ SLOW START { 2° ERRORE } ~





TCP

~ CONGESTION AVOIDANCE {2° ERRORE} ~