2. Livello di Trasporto

2.1 Servizi del livello di trasporto

Il livello di trasporto ha lo scopo di fornire una comunicazione logica tra i processi applicativi in esecuzione su host differenti. Tutti i protocolli di trasporto funzionano in modalità end-to-end, ovvero:

- **Lato mittente**: il protocollo prende in carico i messaggi generati dall'applicazione, li divide in segmenti di dimensioni adeguate e li inoltra al livello di rete.
- **Lato ricevente**: il protocollo raccoglie i segmenti ricevuti dal livello di rete, li riassembla correttamente in messaggi e li consegna al livello applicativo.

Nella suite di protocolli TCP/IP, il livello di trasporto mette a disposizione due protocolli principali per le applicazioni Internet:

- **TCP (Transmission Control Protocol)**, orientato alla connessione e affidabile.
- **UDP (User Datagram Protocol)**, privo di connessione e non affidabile.

Il livello di trasporto rappresenta quindi una separazione netta tra i livelli superiori (applicazione) e inferiori (rete, data link e fisico).

End-to-end

La modalità di comunicazione end-to-end implica direttamente soltanto il mittente e il destinatario finale della comunicazione, senza coinvolgere i nodi intermedi (come router o switch) nella gestione dei dati a livello di trasporto. È importante precisare che i messaggi applicativi non necessariamente hanno una lunghezza fissa: al contrario, il protocollo di trasporto può gestire flussi continui di dati, spesso interpretati dalle applicazioni come una sequenza ininterrotta senza una struttura predefinita.

2.2 Multiplexing e demultiplexing

Il **multiplexing** e il **demultiplexing** sono due funzionalità cruciali svolte dal livello di trasporto nelle reti di calcolatori, necessarie per gestire correttamente i dati

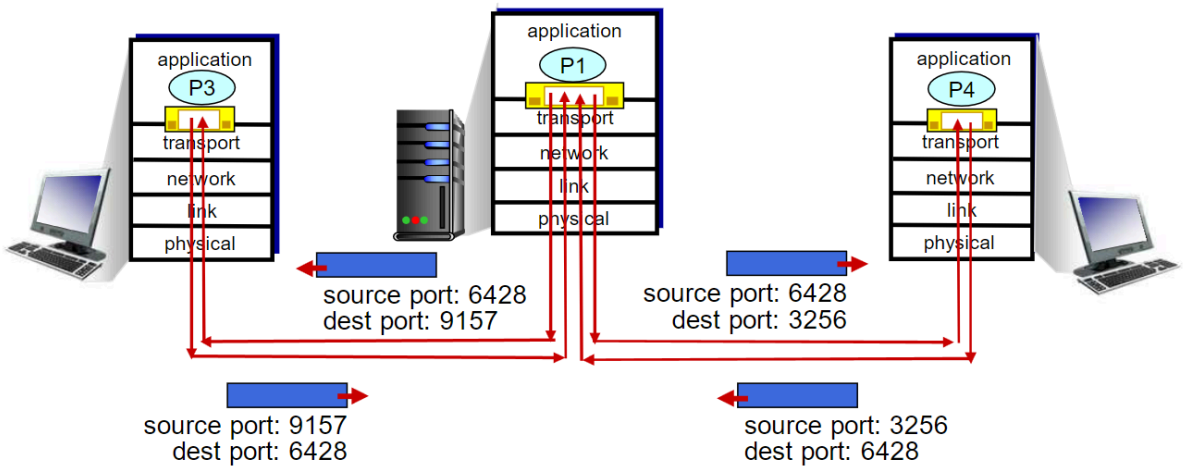
destinati a più applicazioni o processi contemporaneamente attivi sullo stesso host.

> *Il **multiplexing** è il processo che avviene lato mittente. Consiste nel raccogliere dati provenienti da molteplici processi applicativi attivi, incapsularli in segmenti (nel caso del TCP) o datagrammi (nel caso dell'UDP), e quindi passarli al livello di rete.*

> *Il **demultiplexing** è invece il processo complementare, che avviene lato ricevente. Consiste nell'identificare a quale processo applicativo debbano essere consegnati i segmenti (o datagrammi) ricevuti dal livello di rete.*

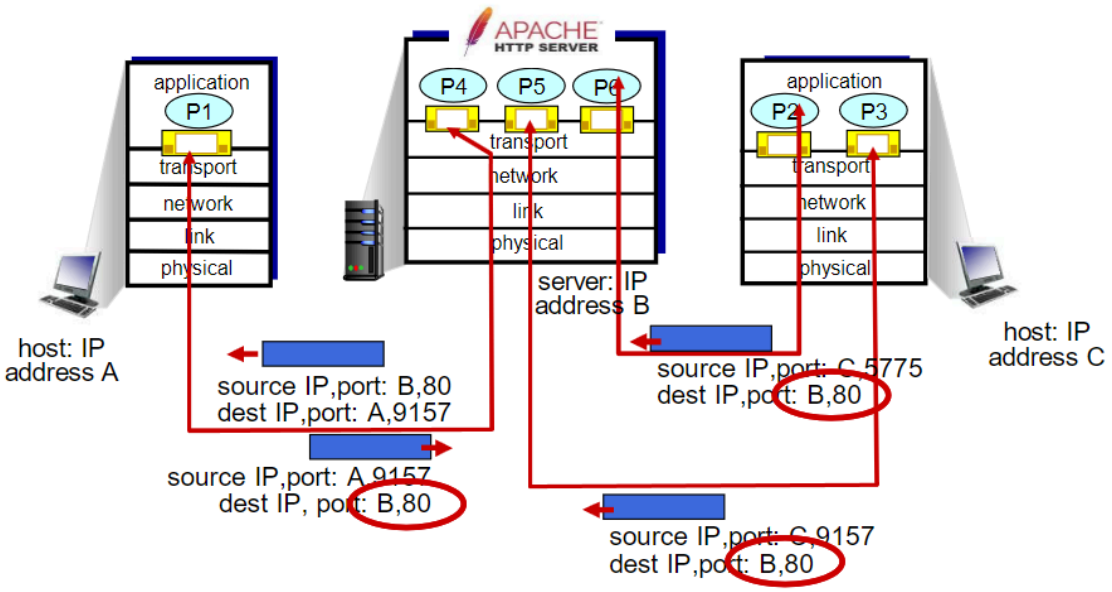
2.2.1 Demultiplexing senza connessione

Il multiplexing senza connessione, utilizzato da UDP, ha come campi di intestazione 16 bit di porta di origine, 16 bit di porta di destinazione e altri campi. A destinazione, per fornire una risposta alla richiesta del client, il server inverte le due porte e manda un altro messaggio.



2.2.2. Demultiplexing con connessione

Il multiplexing con connessione, utilizzato da TCP, invece presenta nell'intestazione un indirizzo IP sorgente, una numero di porta sorgente, un indirizzo IP destinazione e un numero di porta di destinazione. A differenza del multiplexing senza connessione, qui due messaggi aventi indirizzo IP sorgente o numero di porta sorgente differenti verranno inviati a due socket differenti: il server può quindi gestire più socket contemporaneamente.



2.3 UDP: User Datagram Protocol

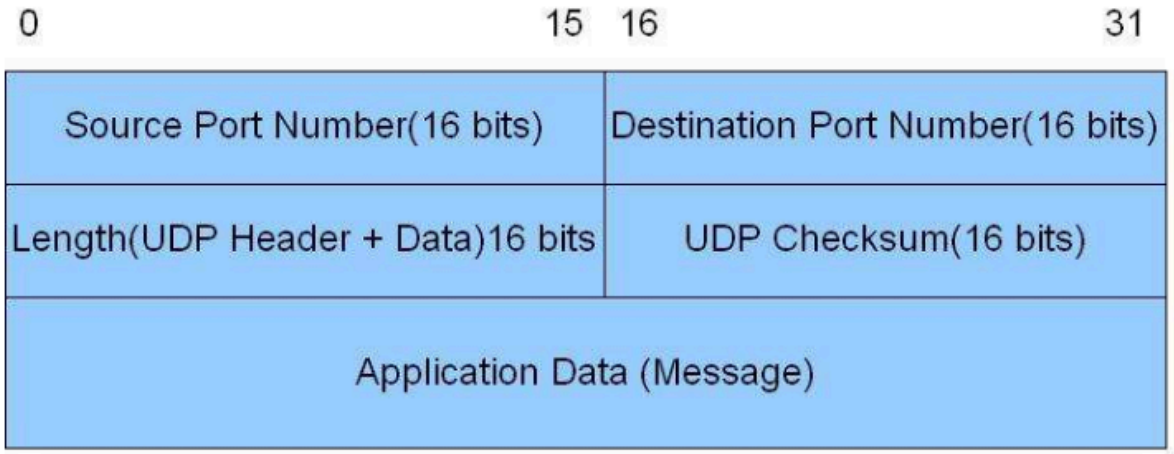
Il protocollo UDP è un protocollo minimo che viene attuato, oltre alle funzioni di multiplexing/demultiplexing e una forma di controllo degli errori molto semplice non aggiunge nulla al livello sotto: IP.

2.3.1 Intestazione

UDP invia i segnali utilizzando una intestazione molto semplice:

- 16 bit dedicati alla porta sorgente.
- 16 bit dedicati alla porta destinazione.
- 16 bit di lunghezza del messaggio.
- 16 bit di checksum.

- Il payload alla fine di lunghezza definita precedentemente.



2.3.2 Vantaggi

Questo tipo di comunicazione non è affidabile, infatti tra mittente e ricevitore non esiste handshaking e un pacchetto potrebbe essere perso. Inoltre ogni segmento UDP è indipendente.

Questo ha degli evidenti svantaggi, ma in certi casi i vantaggi sono di più:

1. **Bassa latenza**: in applicazioni che implementano servizi di videoconferenza o streaming real-time questo è molto più rilevante rispetto alla perdita di qualche pacchetto.
2. **Overhead inferiore**: il server ha meno lavoro da fare quando il pacchetto arriva, non esistono infatti controlli per la congestione o controlli per l'integrità dei pacchetti.
3. **Semplicità**: un protocollo semplice è più utile da gestire dal processore del client e del server.

2.3.3 Controllo degli errori

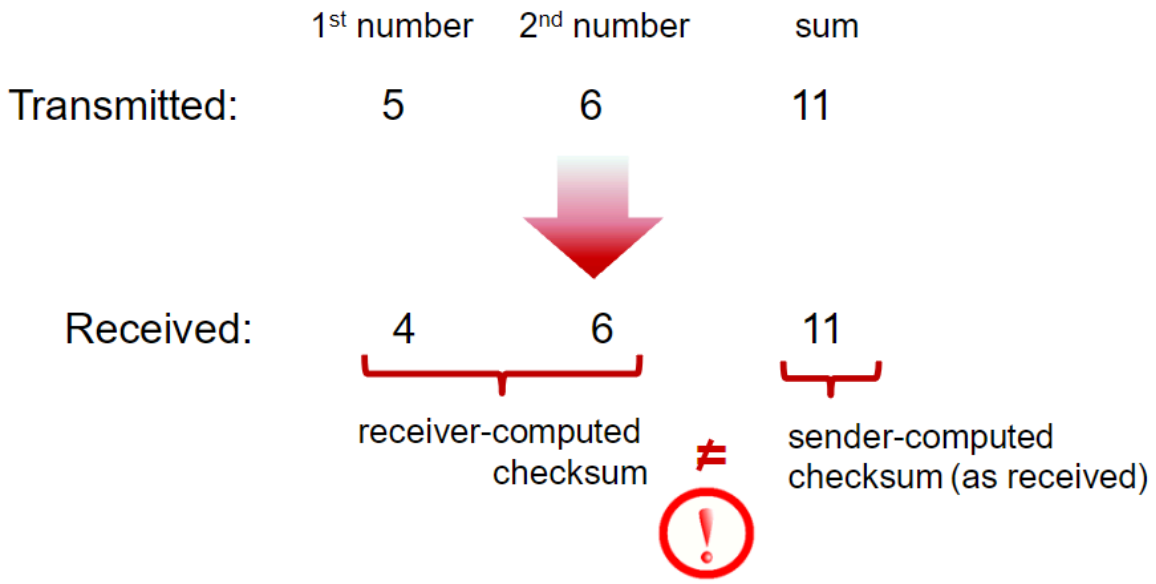
Mittente

Lato mittente, UDP tratta il segmento come sequenze di interi a 16 bit. Quindi per prima cosa prende i numeri e fa la somma, scrivendo nel campo checksum poi il complemento a uno della somma. Può non essere impostato, se viene scritto 0 come

Ricevitore

Lato ricevitore, viene computato il checksum in questo modo: viene fatta la somma dei campi precedenti, poi dopo viene sommata con il checksum e se questa risulta uguale a 1 allora il checksum è giusto.

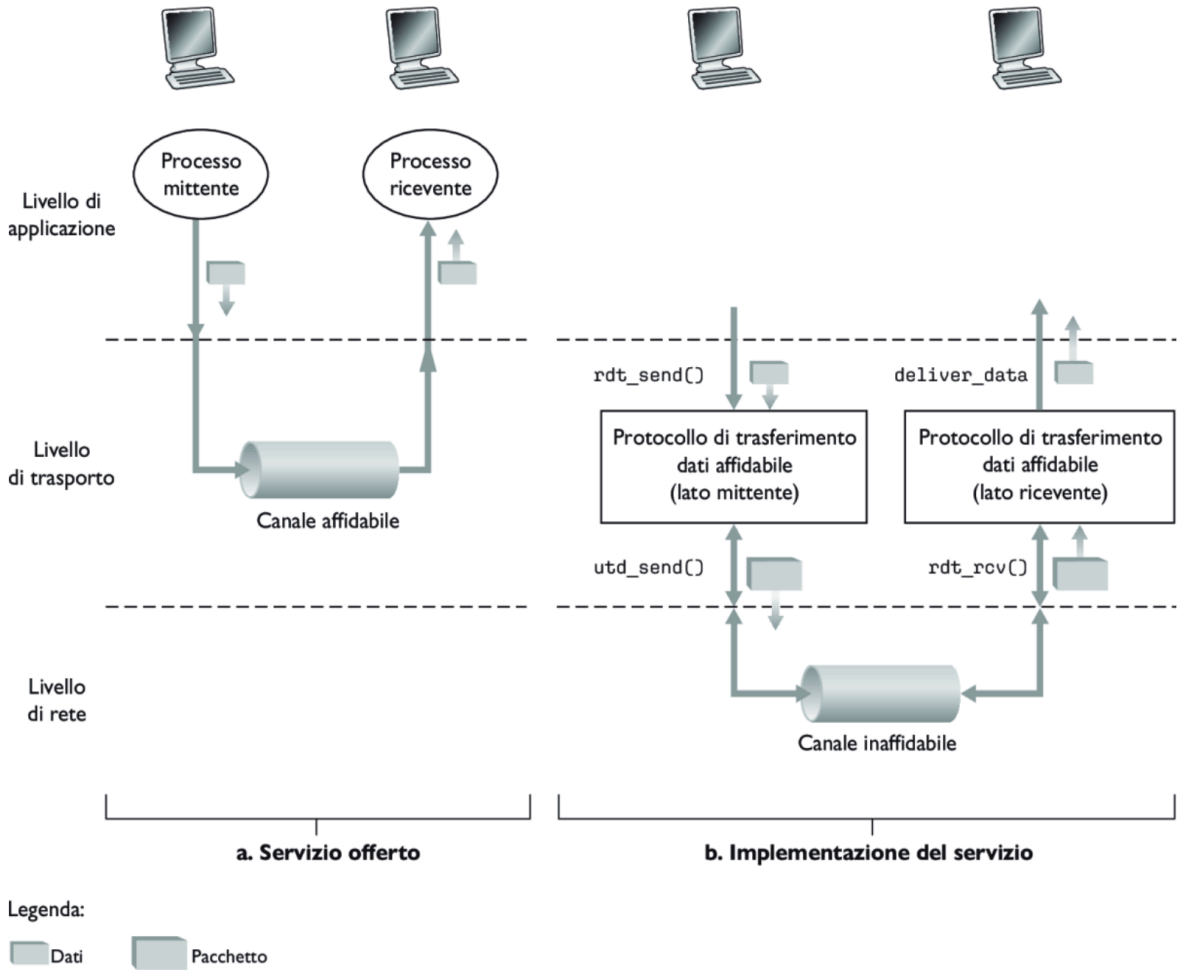
checksum allora non viene calcolato a destinazione.



2.4 Principi del trasferimento dati affidabile

2.4.1 Astrazione

Il compito di un protocollo di trasferimento dati affidabile è l'implementazione di un modo di trasmettere dati in modo affidabile su un livello sottostante inaffidabile: TCP ha come livello sottostante IP, che è inaffidabile end-to-end.

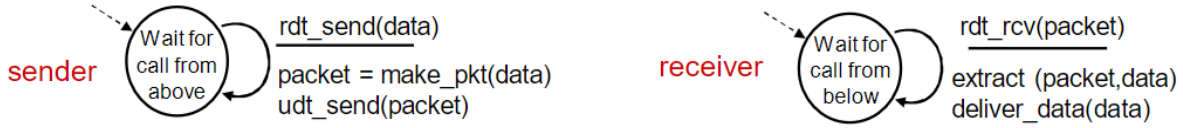


2.4.2 Implementazione

Si può pensare a questa implementazione come a un automa a stati finiti dell'astrazione.

Rdt 1.0: trasferimento dati affidabile su un canale perfettamente affidabile

Il caso più semplice è quello dove il canale sottostante è completamente affidabile: i pacchetti arrivano corretti e nell'ordine di invio. In questo caso, l'implementazione è molto semplice: dobbiamo aspettare di ricevere dal basso un pacchetto lato destinatario e aspettare di ricevere un pacchetto lato mittente. Questo si traduce in un solo nodo del grafo dell'automa con un cappio a sè stesso. Gli automi sono separati in questo caso, ne esiste uno lato mittente e uno lato destinatario.

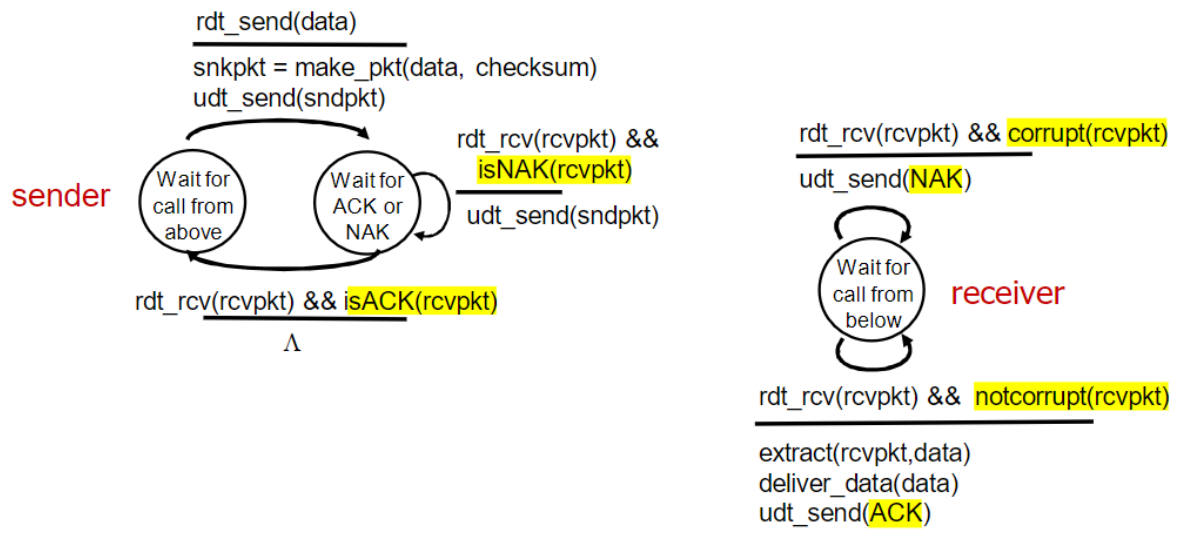


Rdt 2.0: trasferimento dati affidabile su un canale con errori di bit

In questo modello i bit non andranno persi ma possono corrompersi (quindi invertire 0 e 1), si può assumere però che i pacchetti arrivino nell'ordine di invio. In questo scenario è necessario sviluppare un modello in cui gli errori vengono individuati e corretti, una tipologia particolare di protocolli chiamati **ARQ: Automatic Repeat reQuest**.

Questi protocolli funzionano tramite uno schema di notifiche positive, chiamate **ACK**, e notifiche negative, chiamate **NAK** per confermare i pacchetti inviati. Oltre questo schema di feedback del destinatario vengono rilevati gli errori tramite checksum e un meccanismo di ritrasmissione nel caso di errore.

Questo tipo di protocolli è noto come **protocolli stop-and-wait**, poiché finché non viene inviato un pacchetto di ACK o NAK il mittente rimane in attesa.



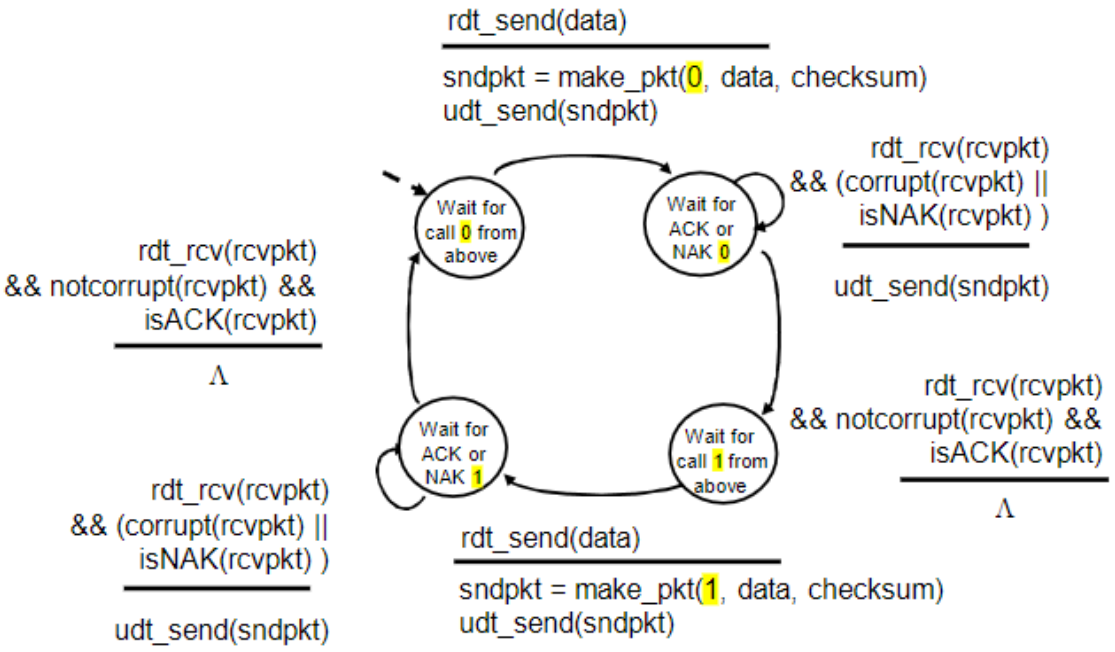
Questo livello **aggiunge ACK/NAK con trasmissione stop&wait** per gestire la conferma dei pacchetti.

Rdt 2.1: correzione degli errori

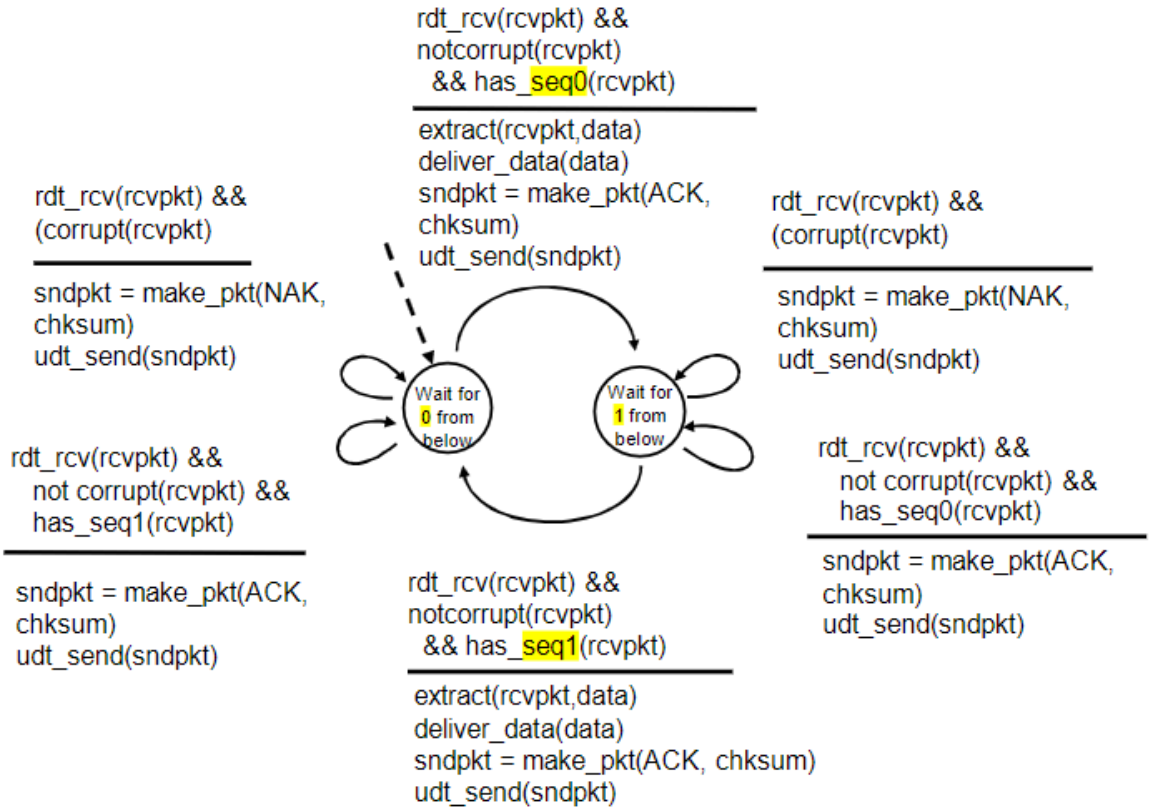
Un problema di questa implementazione di rdt 2.0 è la corruzione dei pacchetti ACK e NAK. In questo caso si agisce, alla ricezione di un pacchetto ACK/NAK corrotto inviandolo ancora.

Ma questo introduce un altro problema: possono esistere **pacchetti duplicati**. Allora, come si fa nella maggior parte dei protocolli di trasferimento dati, si aggiunge un campo al pacchetto dati numerandolo con un **numero di sequenza**.

Automa a stati finiti mittente



Automa a stati finiti destinatario



Il protocollo rdt 2.1 utilizza quindi ACK positivi e negativi dal destinatario verso il mittente. Il destinatario manda un NAK positivo quando riceve un pacchetto fuori sequenza e un ACK riceve un pacchetto alterato.

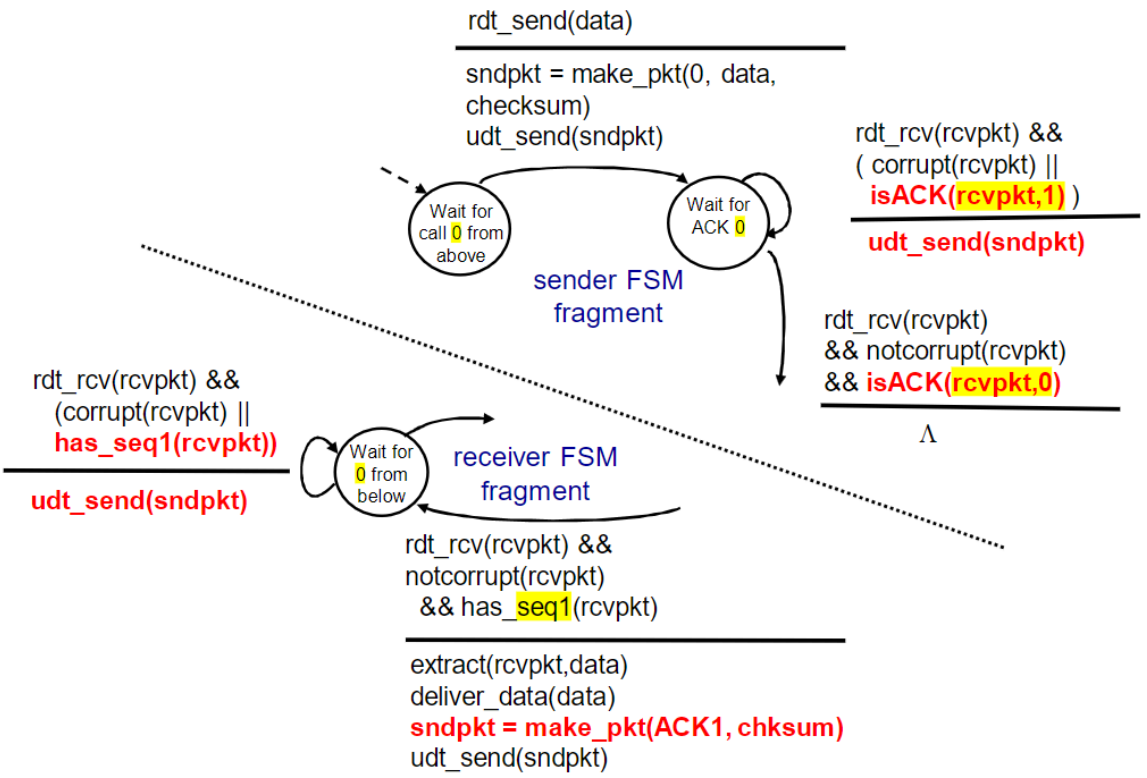
Questo livello aggiunge **un bit di sequenza** e **la logica** necessaria **per gestire**:

- **Ritrasmettere pacchetti su ACK/NAK corrotti**
- **Scartare duplicati al ricevitore**
- **Mantenere due stati alternati al mittente**

Rdt 2.2: Ottimizzazione

Possiamo ottenere lo stesso risultato spendendo piuttosto un ACK per il pacchetto ricevuto più recentemente, così che il mittente se dovesse ricevere due ACK per lo stesso pacchetto (duplicati quindi) sa che il destinatario non ha ricevuto correttamente il pacchetto successivo all'ultimo confermato.

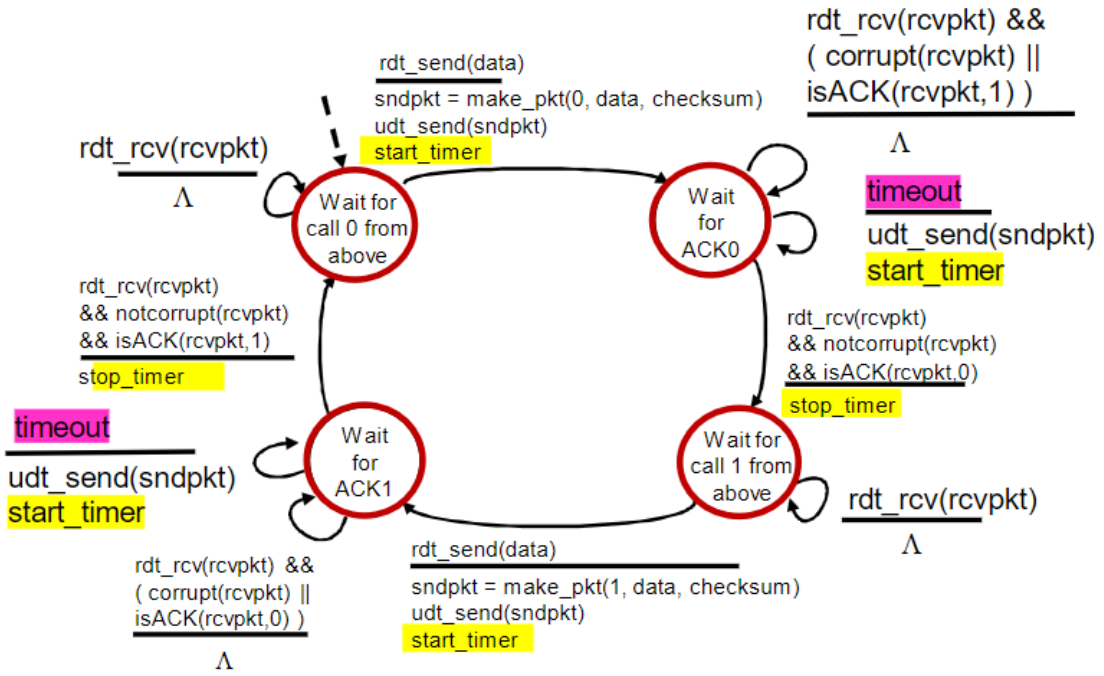
Il destinatario quindi, deve confermare l'arrivo includendo un argomento all'interno del messaggio ACK, che esso sia 1 nel caso di fallimento o 0 nel caso di successo.



Questo livello **rimuove il meccanismo di NAK** e **aggiunge solo ACK** per gestire sia **riscontri positivi che negativi**.

Rdt 3.0: trasferimento dati affidabile su un canale con perdite ed errori sui bit

Supponiamo ora che il canale di trasmissione oltre a danneggiare i bit, possa anche smarrire i pacchetti. In questo caso, si pone un tempo di attesa per ricevere un certo pacchetto, se questo timeout non è rispettato il pacchetto viene contato come non arrivato a va rispedito. Anche se questo, dovesse introdurre pacchetti di dati duplicati, il protocollo rdt 2.2 ha già risolto questo problema.



Questo livello aggiunge:

- la **capacità di rilevare** pacchetti **persi** (oltre che corrotti)
- un meccanismo di **timeout + ritrasmissione**
- e sfrutta il **bit di sequenza** per sopprimere i duplicati generati dalle ritrasmissioni

2.4.3 Throughput nei protocolli di trasferimento dati affidabile

Il **throughput** rappresenta la quantità effettiva di dati utili trasmessi con successo da mittente a destinatario in un dato intervallo di tempo. Nei protocolli di trasferimento dati affidabile del livello di trasporto, esso dipende da molteplici fattori, tra cui la dimensione della finestra, il round-trip time (RTT), e il tempo di trasmissione dei segmenti.

Throughput vs Actual Throughput

Throughput teorico

Rappresenta la **massima quantità** di dati trasmessi per unità di tempo in

Throughput effettivo

È la quantità **effettiva** di dati ricevuti correttamente e senza duplicati, considerando ritrasmissioni e ritardi.

condizioni ideali (senza ritrasmissioni,
perdite, o attese inutili).

$$\text{Throughput teorico} = \frac{L}{RTT + \text{Transmission Time}}$$

$$\text{Throughput effettivo} = \frac{\text{Dati effettivamente ricevuti}}{\text{Tempo totale}}$$

Dove L è la **dimensione del pacchetto**, RTT è il **round trip time** e il tempo di trasmissione è L/R con R capacità del canale in bits.

Throughput nello Stop-and-Wait

Nel protocollo *Stop-and-Wait*, il throughput è fortemente limitato dal fatto che il mittente può inviare **un solo pacchetto per volta**, aspettando l'ACK prima di procedere con il successivo. Questo rende il protocollo semplice ma **inefficiente**, soprattutto su collegamenti ad alta latenza. Il throughput massimo può essere approssimato con:

$$\text{Throughput} \approx \frac{L}{RTT + \text{Trasmission time}} [\text{bit/s}]$$

dove L è la dimensione del pacchetto in bit.

Poiché il mittente è inattivo durante l'attesa dell'ACK, il canale resta spesso inutilizzato: ciò lo rende inadatto a reti ad alta capacità o con alta latenza (come i collegamenti satellitari).

Esempio

Consideriamo:

- $L = 8000 \text{ bit}$
- $R = 1\text{Mb/s}$
- $RTT = 100\text{ms}$

Tempo di trasmissione

$$\frac{8000}{10^6} = 0.008\,\text{s} = 8\,\text{ms}$$

Throughput

$$\text{Throughput} = \frac{8000}{0.1 + 0.008} = \frac{8000}{0.108} \approx 74.07\,\text{kbps}$$

Quindi, sebbene il canale supporti 1 Mbps, il throughput effettivo è solo circa **7.4%** della capacità massima, a causa dell'attesa dell'ACK.

Throughput con trasferimento in pipeline

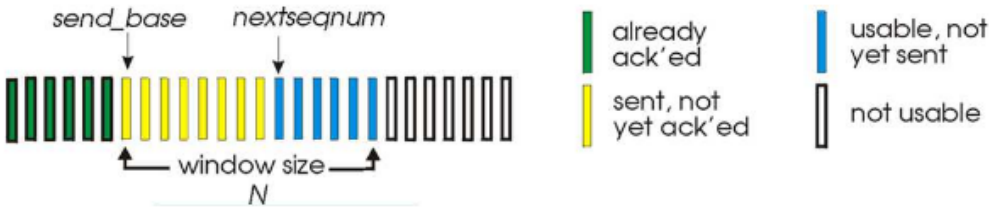
I protocolli **Go-Back-N (GBN)** e **Selective Repeat (SR)** introducono la pipeline per massimizzare l'utilizzo del canale, permettendo l'invio di più pacchetti consecutivi senza attendere conferme individuali. Il throughput può essere notevolmente aumentato, approssimandosi a:

$$Throughput \approx \frac{N \cdot L}{RTT + \text{Trasmission time}} [\text{bit/s}]$$

dove N è la dimensione della finestra.

Go-Back-N (GBN)

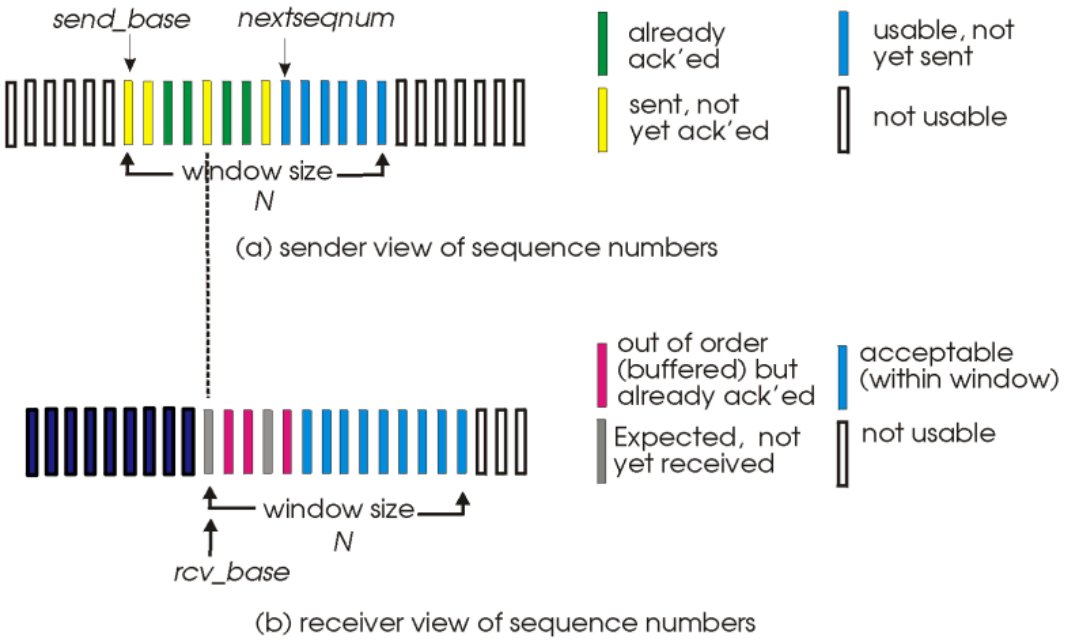
In questo protocollo, il mittente può trasmettere più pacchetti consecutivi senza attendere l'ACK per ognuno (fino a un massimo definito dalla **dimensione della finestra di invio**, NNN). Tuttavia, il ricevente **accetta solo pacchetti in ordine** e scarta quelli ricevuti fuori sequenza. Se un pacchetto viene perso o danneggiato durante la trasmissione, il mittente — dopo il timeout o la ricezione di un NAK — deve **ritrasmettere tutti i pacchetti a partire da quello errato**, anche se alcuni tra quelli successivi erano stati ricevuti correttamente (ma non accettati).



Questo meccanismo di ritrasmissione massiva introduce un **degrado del throughput**, specialmente su collegamenti con alta latenza o con frequenti errori, poiché ogni errore causa la ripetizione di pacchetti già trasmessi, aumentando il carico e riducendo l'efficienza complessiva.

Selective Repeat (SR)

In SR, sia il mittente che il ricevente gestiscono una **finestra di dimensione** N e i pacchetti vengono identificati tramite un numero di sequenza. Il mittente può trasmettere più pacchetti senza attendere gli ACK, ma, a differenza di GBN, **ritrasmette soltanto i pacchetti che non sono stati confermati** (cioè per cui non ha ricevuto l'ACK entro un certo timeout).



Il ricevente è in grado di **accettare pacchetti fuori ordine** e memorizzarli temporaneamente in un buffer, fornendo un ACK individuale per ciascuno. Solo i pacchetti mancanti verranno ritrasmessi, evitando sprechi di banda. Questo comportamento rende il protocollo **più efficiente dal punto di vista del throughput**, soprattutto in presenza di perdite o errori isolati, in quanto limita il numero di ritrasmissioni al minimo indispensabile.

Esempio

Consideriamo:

- $L = 8000 \text{ bit}$
- $R = 1\text{Mb/s}$

- $RTT = 100\text{ms}$

Da queste condizioni, possiamo stimare il **throughput**:

$$\text{Throughput} = \frac{N \cdot L}{RTT + L/R} = \frac{10 \cdot 8000}{0.1 + 0.008} \approx 740.7\,\text{kbps}$$

Quindi sfruttiamo circa **74%** della capacità del canale, grazie alla pipeline.

Banda-ritardo (Bandwidth-Delay Product)

Un concetto centrale nell'analisi del throughput è il **bandwidth-delay product**, che rappresenta la quantità di dati "in volo" necessaria per saturare il collegamento. Si calcola come:

$$\text{Banda} \cdot RTT\ [bit]$$

Per ottenere un throughput massimo, il mittente dovrebbe poter avere almeno tanti bit "non riconosciuti" in transito quanto il bandwidth-delay product. Questo implica che la **dimensione della finestra di trasmissione** deve essere almeno uguale a questo valore, altrimenti il collegamento risulterà sottoutilizzato.

Considerazioni sul bandwidth-delay product

Il **bandwidth-delay product (BDP)** rappresenta la quantità di dati che possono essere "in volo" nel canale per saturarlo:

$$\text{BDP} = R \cdot RTT = 10^6 \cdot 0.1 = 100000\,\text{bit} = 12.5\,\text{kB}$$

Per sfruttare **pienamente** il canale, la finestra del mittente dovrebbe permettere di avere almeno **12.5 kB di dati in transito simultaneo**. In altre parole:

$$\text{Dimensione minima della finestra} = \frac{\text{BDP}}{L} = \frac{100000}{8000} = 12.5 \Rightarrow 13\,\text{segmenti}$$

Implicazioni pratiche

In ambienti **a bassa latenza** (LAN), anche Stop-and-Wait può raggiungere throughput accettabili.

In ambienti **ad alta latenza** (WAN o reti wireless), l'uso della pipeline è fondamentale per mantenere alte prestazioni.

L'adozione di **meccanismi dinamici di controllo del flusso e della congestione**, come quelli integrati in TCP, permette di adattare dinamicamente la finestra per massimizzare il throughput evitando il collasso della rete.

2.5 TCP

2.5.1 Descrizione del protocollo

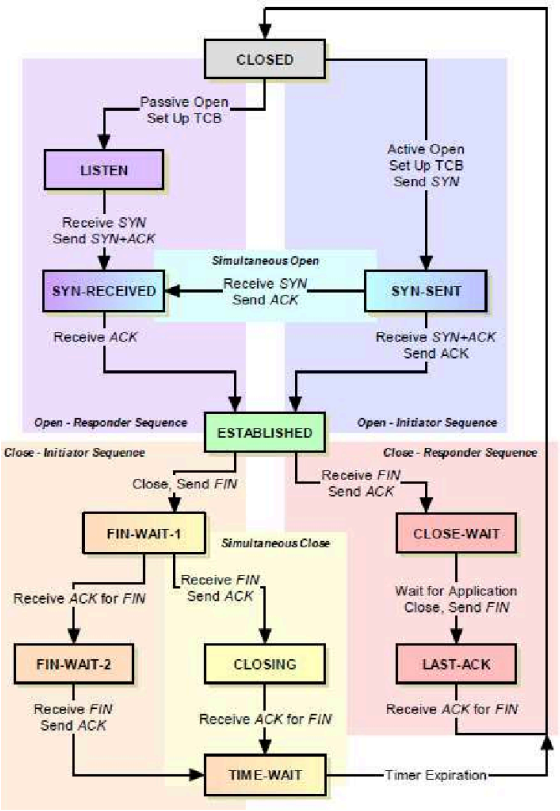
Introduzione

Il **Transmission Control Protocol (TCP)** è un protocollo di trasporto **orientato alla connessione**, **affidabile**, **full-duplex** e **basato su byte-stream**, ampiamente utilizzato nelle applicazioni Internet come HTTP, FTP, SMTP e Telnet. A differenza di UDP, che offre un servizio non affidabile e connection-less, TCP fornisce una comunicazione robusta, garantendo che i dati arrivino correttamente, nell'ordine previsto e senza duplicazioni.

TCP realizza una **connessione logica end-to-end** tra processi in esecuzione su host distinti. Opera secondo un modello client-server, nel quale entrambi i lati mantengono uno stato della connessione, fondamentale per la gestione della sequenza, dell'affidabilità e del controllo di flusso.

Comunicazione

La comunicazione inizia con una fase di **three-way handshake**, che sincronizza i numeri di sequenza e conferma la disponibilità dei due host. Per garantire l'affidabilità, TCP utilizza **numeri di sequenza** per ordinare i byte, **ACK espliciti** per confermare la ricezione, **timeout e ritrasmissioni** per gestire perdite, e un **checksum** per il controllo d'errore.



Inoltre, implementa meccanismi di **controllo del flusso** per evitare che il mittente sovraccarichi il ricevente, e **controllo della congestione** per adattarsi dinamicamente allo stato della rete, prevenendo il collasso in caso di sovraccarico. Infine, la comunicazione è **bidirezionale** (full-duplex) e trasparente all'applicazione,

che percepisce un semplice canale affidabile end-to-end.

MSS

Il **Maximum Segment Size (MSS)** è il massimo numero di byte di **dati** (payload TCP) che un host è disposto a ricevere in un singolo segmento TCP. È strettamente legato alla **MTU (Maximum Transmission Unit)**, ovvero la dimensione massima di un pacchetto che può essere trasmesso su un collegamento di rete senza essere frammentato a livello IP.

Nel caso di una rete Ethernet standard, la **MTU è di 1500 byte**. Poiché un pacchetto TCP/IP include:

- 20 byte di intestazione IP,
- 20 byte di intestazione TCP,

Un valore tipico di MSS TCP è dunque $\mathrm{MSS} = 1500 - 20 - 20 = 1460$

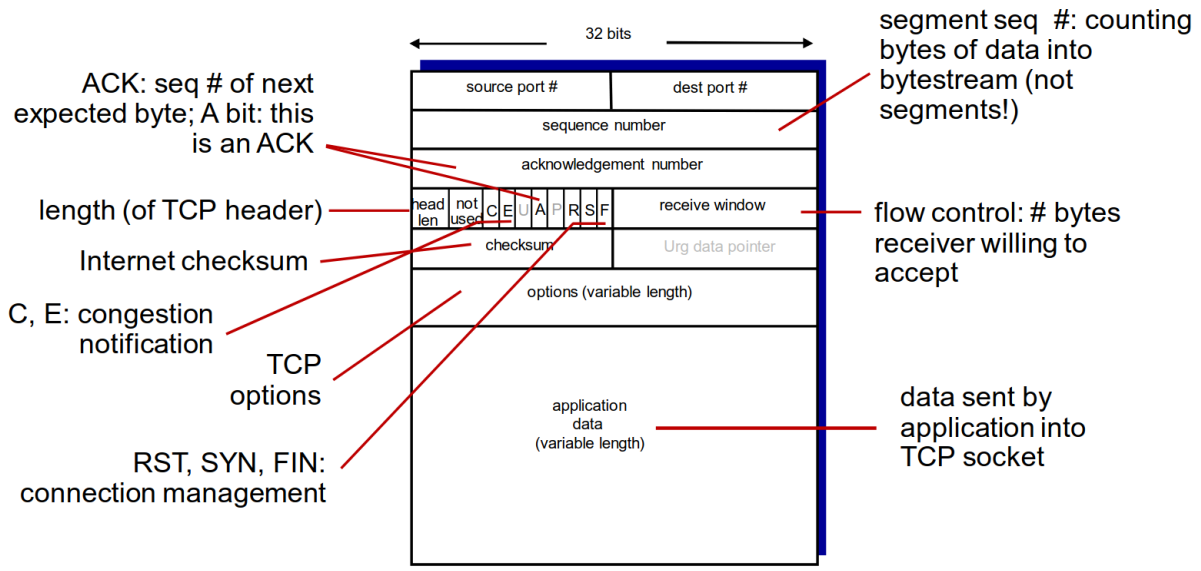
Questo valore viene **negoziato** durante il three-way handshake per evitare la **frammentazione IP**, che comporterebbe un peggioramento delle prestazioni e un aumento del rischio di perdita. Il minimo comunque richiesto è 576 byte.

In sintesi

MTU riguarda l'intero pacchetto IP (dati + intestazioni),	**MSS** riguarda solo il payload TCP e viene scelto in funzione della MTU per garantire un uso efficiente del canale senza frammentazioni.

2.5.2 Struttura del segmento TCP

Un **segmento TCP** è l'unità base di trasferimento dati utilizzata dal protocollo. Ogni segmento è costituito da un'**intestazione (header)** seguita da un **payload**, ovvero i dati applicativi. L'intestazione contiene informazioni fondamentali per garantire l'affidabilità, l'ordinamento e il controllo della comunicazione.



Campi

I campi principali del segmento TCP sono:

- **Source Port (16 bit)**: numero di porta del processo mittente.
- **Destination Port (16 bit)**: numero di porta del processo destinatario.
- **Sequence Number (32 bit)**: indica il numero del primo byte di dati nel segmento. Serve per l'ordinamento del flusso.
- **Acknowledgment Number (32 bit)**: valido se il flag ACK è attivo; rappresenta il numero di sequenza atteso per il prossimo byte.
- **Header Length (4 bit)**: lunghezza dell'intestazione TCP in parole da 32 bit.
- **Reserved (3 bit)**: riservati per usi futuri.
- **Flags (9 bit)**: bit di controllo:
  - **URG** (dati urgenti),
  - **ACK** (campo acknowledgment valido),
  - **PSH** (push: invia subito i dati),
  - **RST** (reset connessione),
  - **SYN** (inizio connessione),
  - **FIN** (fine connessione).
- **Window Size (16 bit)**: dimensione della finestra di ricezione del destinatario (controllo di flusso).

- **Checksum (16 bit)**: usato per il controllo degli errori su header e dati.
- **Urgent Pointer (16 bit)**: valido se URG è attivo, indica la fine dei dati urgenti.
- **Options (variabile)**: estensioni opzionali (es. MSS, timestamp). L'uso delle opzioni fa variare la lunghezza dell'intestazione.

2.5.3 Calcolo dell'RTT in TCP

Il **Round-Trip Time (RTT)** è il tempo che intercorre tra l'invio di un segmento e la ricezione del relativo **ACK**. È una misura fondamentale nel protocollo TCP, utilizzata per impostare i **timeout di ritrasmissione (RTO, Re-transmission TimeOut)**: se un ACK non arriva entro questo tempo, il segmento viene ritrasmesso.

Poiché l'RTT varia nel tempo (a causa della congestione o della variazione del percorso), TCP non usa una stima statica, ma un meccanismo dinamico e adattivo.

Misurazione dell'RTT campionato (SampleRTT)

TCP misura continuamente l'**RTT campionato (SampleRTT)** come il tempo trascorso tra l'invio di un segmento e la ricezione del suo ACK **senza ritrasmissioni**. Non tutti i segmenti sono usati: per evitare ambiguità, solo gli ACK univoci (non duplicati) vengono presi in considerazione.

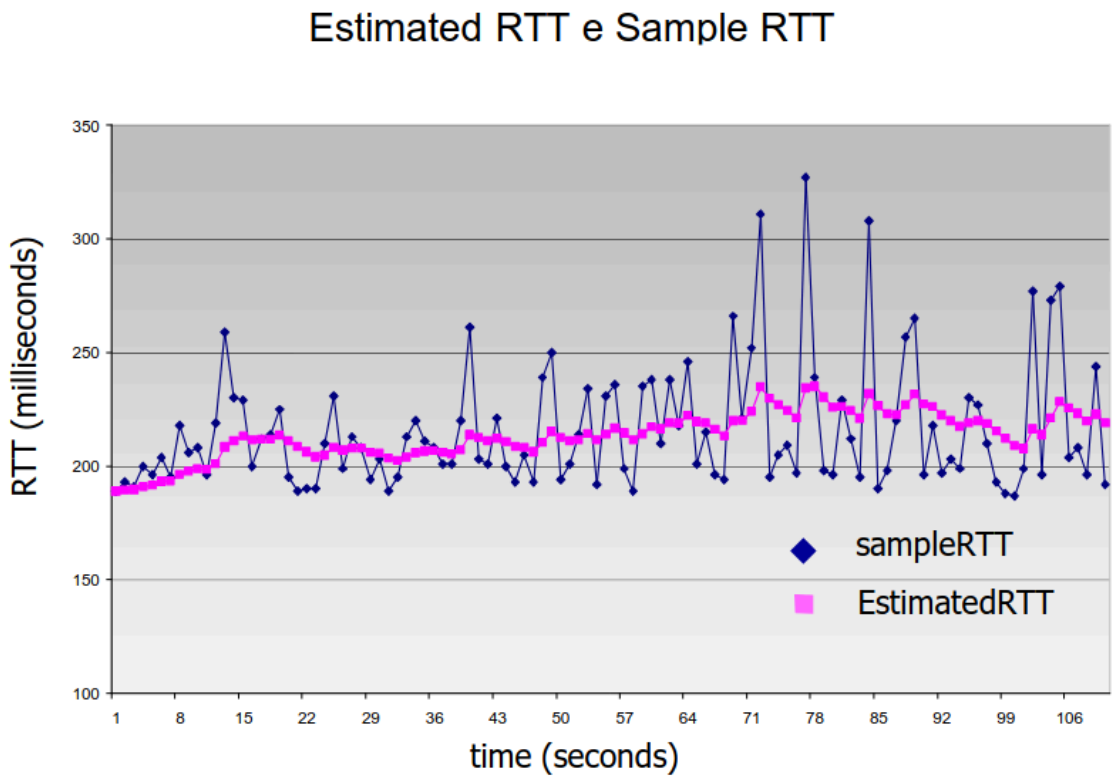
Stima dell'RTT (EstimatedRTT)

TCP aggiorna una stima smussata dell'RTT usando una media mobile esponenziale:

$$\text{EstimatedRTT}_n = (1 - \alpha) \cdot \text{EstimatedRTT}_{n-1} + \alpha \cdot \text{SampleRTT}$$

Dove:

- α è un valore di smoothing (tipicamente $\alpha = 0.125$)
- EstimatedRTT cambia lentamente nel tempo, filtrando le fluttuazioni brevi.
- $\text{EstimatedRTT}_0 = \text{SampleRTT}$



Calcolo del valore di α

In effetti, il calcolo del valore α è possibile e dimostrabile.

$$
\begin{aligned}
E_{RTT_{n-2}} &= (1-\alpha)\cdot E_{RTT_{n-3}} + \alpha \cdot S_{RTT_{n-2}} \\
E_{RTT_{n-1}} &= (1-\alpha)\cdot E_{RTT_{n-2}} + \alpha \cdot S_{RTT_{n-1}} \\
E_{RTT_{n}} &= (1-\alpha)\cdot E_{RTT_{n-1}} + \alpha \cdot S_{RTT_{n}}
\end{aligned}
$$

$$
\begin{aligned}
E_{RTT_{n}} &= (1-\alpha)^2\cdot E_{RTT_{n-2}} + \alpha(1-\alpha) \cdot S_{RTT_{n-1}} + \alpha \cdot S_{RTT_{n}} \\
E_{RTT_{n}} &= (1-\alpha)^3\cdot E_{RTT_{n-3}} + \alpha(1-\alpha)^2 \cdot S_{RTT_{n-2}} \\
&\quad + \alpha(1-\alpha) \cdot S_{RTT_{n-1}} + \alpha \cdot S_{RTT_{n}}
\end{aligned}
$$

$$
\begin{aligned}
E_{RTT_{n}} &= 0.6699\cdot E_{RTT_{n-3}} + 0.0957 \cdot S_{RTT_{n-2}} \\
&\quad + 0.1094 \cdot S_{RTT_{n-1}} + 0.125 \cdot S_{RTT_{n}}
\end{aligned}
$$

Timeout di ritrasmissione (RTO)

Oltre ad avere una stima di RTT è anche importante possede la misura della sua variabilità. Viene introdotto allora la variazione di RTT, **DevRTT** calcolato come una

stimpa di quanto SampleRTT generalmente si discosta da EstimatedRTT

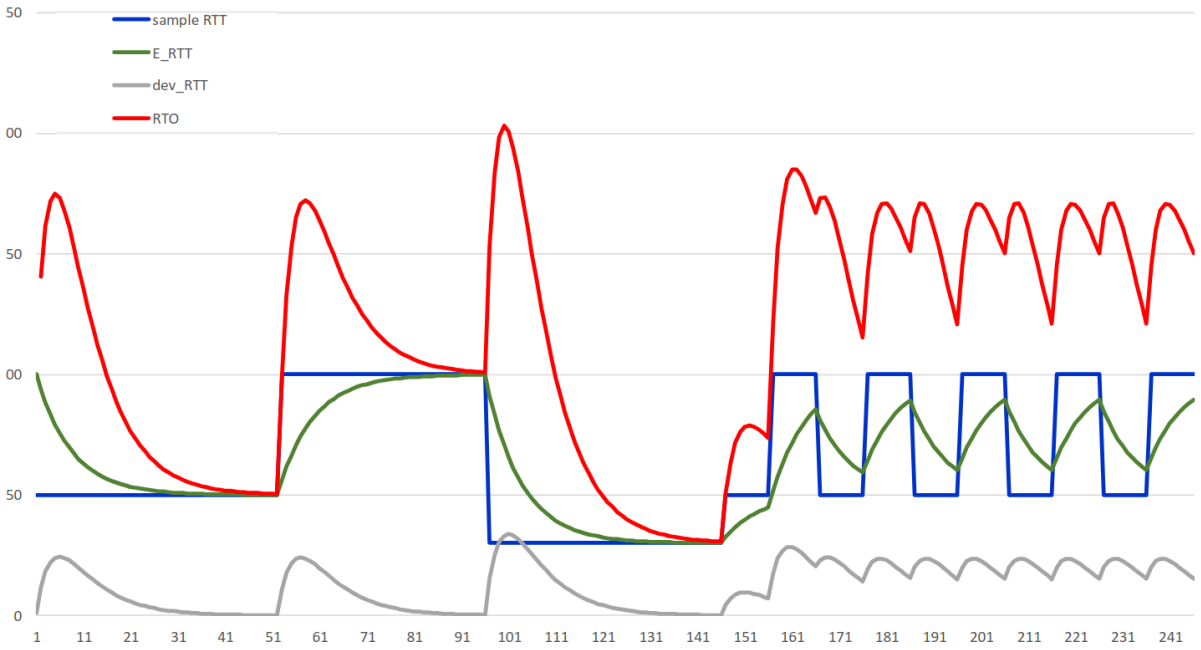
$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot |\text{SampleRTT} - \text{EstimatedRTT}|$$

Con $\beta = 0.25$ come valore standard.

E da questo possiamo calcolare **il tempo di ritrasmissione (RTO)**:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

Il fattore 4 fornisce un margine di sicurezza per evitare ritrasmissioni premature in presenza di variazioni del ritardo. Nel caso di timeout TimeoutInterval viene raddoppiato, per essere poi riaggiornato al prossimo pacchetto inviato.



2.5.4 Controllo di flusso (Flow Control)

Il **controllo di flusso** in TCP è un meccanismo che previene il sovraccarico del destinatario, evitando che il mittente invii dati più velocemente di quanto il ricevente sia in grado di elaborarli o immagazzinarli. In altre parole, il flow control **garantisce che il buffer del ricevente non venga saturato**.

Sliding Window TCP

TCP implementa un controllo di flusso tramite un meccanismo detto **finestra scorrevole (sliding window)**. Ogni host TCP mantiene un buffer di ricezione: quando riceve dati, li salva temporaneamente in questo buffer prima di passarli

all'applicazione. Se l'applicazione consuma i dati lentamente, il buffer può riempirsi.

Per evitare la perdita di dati, il ricevente comunica al mittente, all'interno di ogni segmento TCP, la quantità di spazio disponibile nel suo buffer tramite il campo "windowsize". Questo valore, chiamato anche **rwnd (receiver window)**, indica quanti byte il mittente può ancora trasmettere oltre il byte già confermato con ACK. Se `rwnd = 0`, significa che il ricevente ha il buffer pieno, e il mittente deve sospendere temporaneamente la trasmissione.

Blocco e ripresa

Quando `rwnd = 0`, il mittente entra in attesa. Periodicamente, può inviare un **segmento "sonda"** (zero window probe) per chiedere al ricevente se si è liberato spazio nel buffer. Appena il ricevente può accettare nuovi dati, invierà un ACK aggiornato con `rwnd > 0`, consentendo al mittente di riprendere la trasmissione.

Differenze rispetto al controllo di congestione

È importante **non confondere** il controllo di flusso con il **controllo della congestione**:

- **Flow control**: gestisce la capacità del *destinatario*.
- **Congestion control**: gestisce la capacità della rete.

Nagle's Algorithm

Per migliorare l'efficienza dell'invio, TCP implementa anche il **Nagle's Algorithm**, che previene l'invio di segmenti piccoli (ad esempio, pochi byte) quando c'è già un segmento non ancora riconosciuto (ACK). Questo riduce il numero di pacchetti e l'overhead trasmissivo, contrastando la **silly window syndrome** (il comportamento da parte di mittente o destinatario di accettare pacchetti molto piccoli, inducendo così una trasmissione inefficiente).

Regola base:

> Se ci sono dati da inviare e il mittente ha segmenti non ancora confermati, allora accumula i dati e aspetta l'ACK prima di inviare nuovi segmenti.

Pseudocodice – Nagle's Algorithm

```
Se (nuovi dati da inviare):
   Se (segmenti precedenti confermati) OR (buffer >= MSS):
      invia segmento
   Altrimenti:
      accumula dati nel buffer e attendi ACK
```

Dove MSS è il Maximum Segment Size, e rappresenta il limite massimo del payload per evitare la frammentazione IP.

Considerazioni:

- **Pro**: migliora l'efficienza della rete evitando troppi pacchetti piccoli.
- **Contro**: può introdurre **latenza** indesiderata in applicazioni interattive (es. SSH, giochi, VoIP), motivo per cui può essere **disabilitato** (es. `TCP_NODELAY` nelle API socket).

2.5.5 Handshake

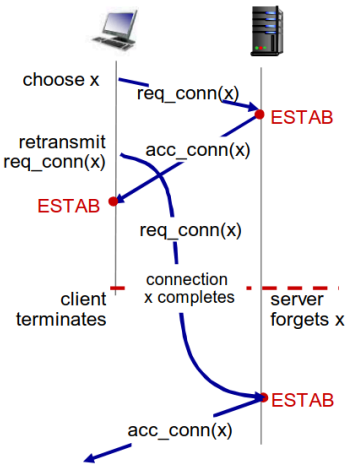
TCP è un protocollo **orientato alla connessione**, il che significa che prima dello scambio dati, i due host devono **stabilire una connessione**, negoziando i parametri iniziali come i **numeri di sequenza iniziali**. Al termine della comunicazione, la connessione deve essere **chiusa in modo controllato**. Questi due processi avvengono tramite meccanismi chiamati **handshake.**

Two-Way Handshake

Storicamente, un metodo semplificato chiamato **two-way handshake** prevedeva solo **due scambi** per aprire una connessione:

1. Il **client** invia una richiesta di connessione (`SYN`).
2. Il **server** risponde con un `SYN+ACK` .

A questo punto, il client considera la connessione stabilita e inizia l'invio di dati. Tuttavia, questo schema presenta un grave problema: **l'incapacità di verificare la raggiungibilità reciproca in modo affidabile**, causando possibili situazioni di **half-open connection**, dove una delle due parti pensa che la connessione sia attiva, mentre l'altra no.

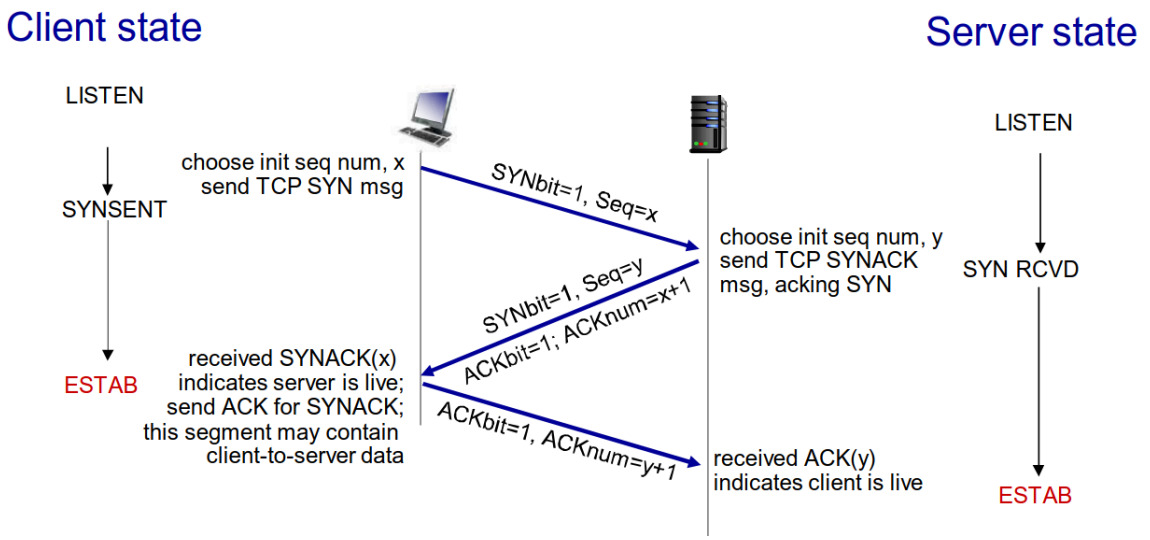


Three-Way Handshake

Per evitare questi problemi, TCP adotta il **three-way handshake**, che garantisce che **entrambi gli host siano sincronizzati e operativi**, e che **i numeri di sequenza** siano concordati in modo sicuro.

Fasi del three-way handshake:

1. **SYN (Synchronize)**: Il client invia un segmento con il flag `SYN` e il proprio numero di sequenza iniziale `Seq = x`.
2. **SYN-ACK (Synchronize-Acknowledgement)**: Il server risponde con un segmento che ha `SYN` e `ACK` attivi. Conferma il `Seq = x + 1` e propone il proprio numero di sequenza `Seq = y`.
3. **ACK (Acknowledgement)**: Il client conferma il numero di sequenza del server inviando `ACK` con `Seq = x + 1`, `Ack = y + 1`.



Solo dopo il terzo scambio **la connessione è considerata attiva** da entrambi.

Benefici:

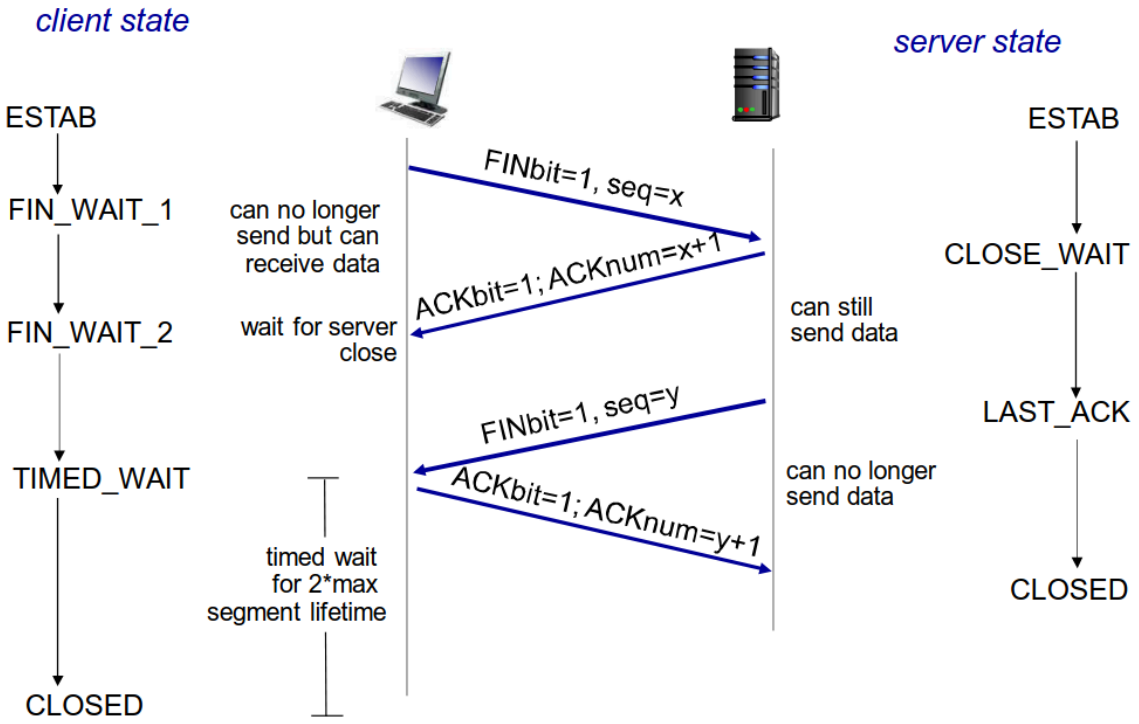
- Evita connessioni unilaterali "fantasma".
- Garantisce la sincronizzazione bidirezionale.
- Permette la negoziazione dell'**Initial Sequence Number (ISN)** da entrambe le parti.

Chiusura della connessione: Four-Way Handshake

La terminazione della connessione TCP è **asimmetrica**: ogni lato deve chiudere in modo indipendente il proprio flusso di dati. Questo richiede **quattro segmenti** distinti, poiché la chiusura avviene **unidirezionalmente per volta**.

Fasi del four-way handshake:

1. **FIN**: Il client (o server) invia un segmento con il flag `FIN` per indicare che ha terminato l'invio dei dati.
2. **ACK**: L'altra parte risponde con un ACK, confermando la ricezione del FIN.
3. **FIN**: Quando anche l'altro lato ha terminato l'invio, invia un proprio segmento `FIN`.
4. **ACK**: Il primo host risponde con un ultimo ACK.



A questo punto la connessione è chiusa. TCP impone uno stato chiamato **TIME_WAIT**, durante il quale il lato che ha chiuso per ultimo resta attivo per un tempo prefissato (tipicamente 2×RTT) per garantire che gli ultimi ACK non vadano persi.

2.5.6 Problema della congestione

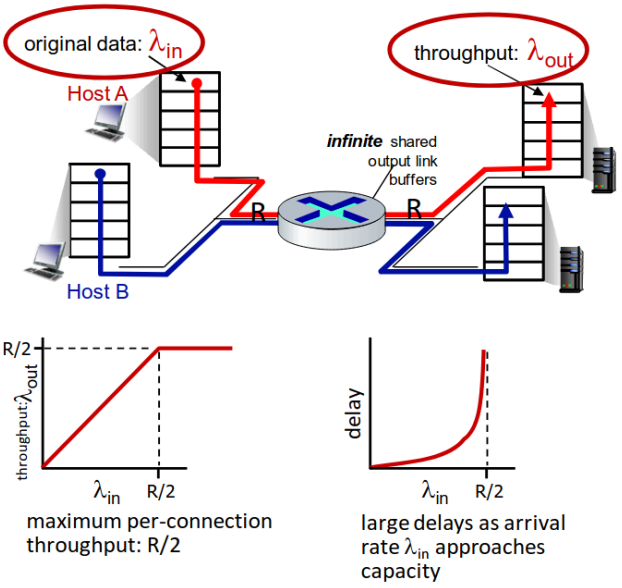
Concetto di congestione

La **congestione** in una rete si verifica quando il carico di traffico supera la capacità dei collegamenti o dei router lungo il percorso, causando **ritardi**, **perdite di pacchetti** e **riduzione dell'efficienza del throughput**. In TCP, il problema è che i mittenti non possono osservare direttamente la rete: si accorgono della congestione solo **indirettamente**, ad esempio quando non ricevono un ACK o quando il tempo di risposta aumenta sensibilmente.

Per meglio comprendere le cause e gli effetti della congestione, si analizzano tre scenari fondamentali.

Scenario 1: Router con buffer infiniti

Ipotizzando due host che inviano dati verso un router condiviso, il quale ha una **capacità d'uscita** R e **buffer infiniti** si ha un modello ideale dove non si perdono pacchetti perché il router può sempre accumularli. Tuttavia, man mano che il tasso d'ingresso dei dati (λ_{in}) si avvicina a R, i **pacchetti si accumulano nei buffer** causando **ritardi sempre maggiori**. Questo scenario mostra che anche **senza perdite**, una rete può essere inefficiente: i pacchetti impiegano troppo tempo ad attraversarla.



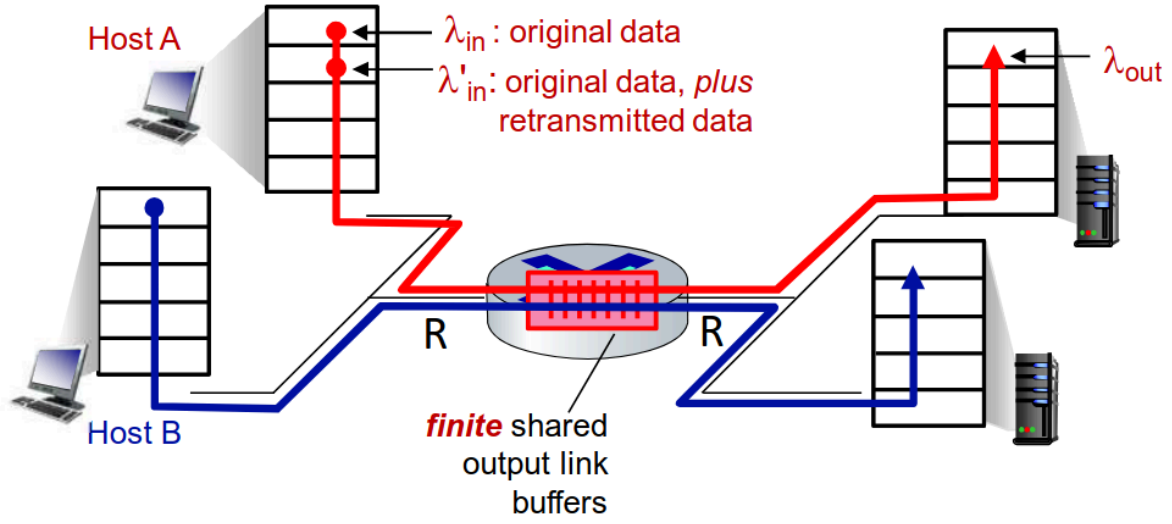
maximum per-connection throughput: R/2

large delays as arrival rate λ_{in} approaches capacity

Scenario 2: Router con buffer finiti e ritrasmissioni

Qui il buffer del router è **limitato**, come accade nella realtà. Quando i pacchetti arrivano troppo velocemente, alcuni vengono **scartati**. I mittenti TCP, non ricevendo ACK per questi pacchetti, **attivano la ritrasmissione** dopo un timeout. Ma ecco il punto critico: se il mittente **non sa esattamente cosa è stato perso**, può finire per **inviare duplicati non necessari**.

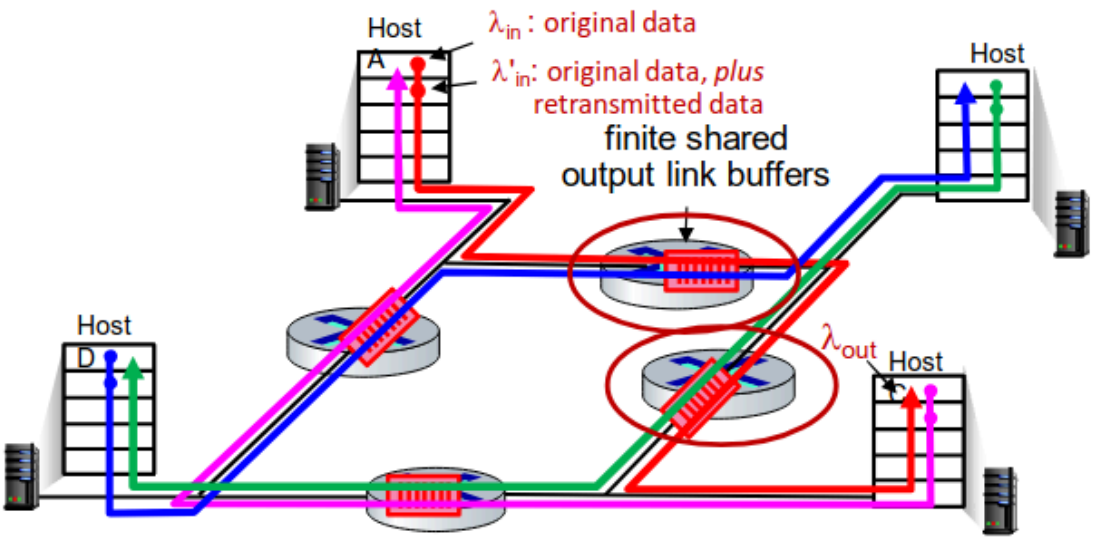
La conseguenza è che il traffico trasmesso (λ'_{in}) contiene **sia dati originali sia dati ritrasmessi**, spesso inutilmente. Questo **sovraccarica ulteriormente la rete**, peggiora la congestione e **spreca la capacità di banda**, perché lo stesso pacchetto può essere trasmesso più volte. L'effetto finale è che, anche trasmettendo a una velocità nominale accettabile (es. $R/2$), il throughput effettivo può essere **molto più basso**, proprio a causa di queste ritrasmissioni superflue.



Scenario 3: Molti mittenti e topologia a più salti

Questo scenario è ancora più realistico: diversi host (es. 4) trasmettono dati su percorsi **multi-hop** (attraversare più dispositivi intermedi nella comunicazione). I pacchetti attraversano più router, condividendo i collegamenti. Quando il traffico totale in ingresso cresce, alcuni pacchetti vengono **scartati a metà strada**. Questo significa che **le risorse già usate a monte** (come la capacità del primo link o i buffer intermedi) sono **sprecate** per pacchetti che **non arriveranno mai a destinazione**.

In particolare, succede che alcuni flussi riescono a "dominare" l'accesso al link d'uscita, mentre altri (ad esempio quelli identificati dal colore blu nelle slide) vedono tutti i loro pacchetti scartati: il loro **throughput si azzera**. Questo scenario mostra che la congestione non solo abbassa il throughput totale, ma può anche causare **inequità tra flussi**, e la rete spreca risorse già consumate da pacchetti poi persi.



2.5.7 Gestione della congestione

La gestione della congestione può essere affrontata con due approcci principali:

Closed-loop vs Open-loop

- **Closed-loop control**: TCP rientra in questa categoria. È un approccio **reattivo**: il mittente **adatta dinamicamente il tasso di invio** in base ai feedback impliciti della rete (ritardi, perdite, ACK duplicati).
- **Open-loop control**: è un approccio **preventivo**, adottato in sistemi come ATM, dove si tenta di evitare la congestione regolando a priori il traffico ammesso nella rete.

End-to-end vs Network-assisted

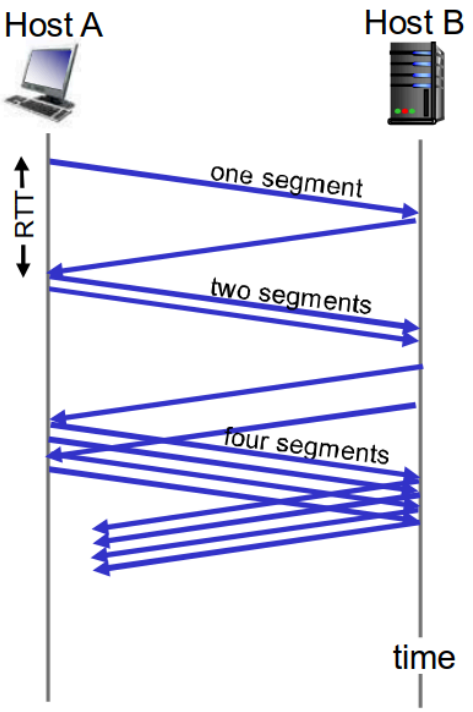
- **End-to-end congestion control** (come TCP): non c'è feedback esplicito dai router. Il mittente rileva congestione solo indirettamente, osservando **timeout** o **ACK duplicati**. È una soluzione completamente **distribuita**, robusta e scalabile.
- **Network-assisted congestion control**: la rete (es. i router) comunica **esplicitamente** agli host quando c'è congestione, ad esempio impostando **bit nei pacchetti IP** (es. ECN – Explicit Congestion Notification).

2.5.8 Controllo della congestione in TCP

Il controllo della congestione in TCP ha lo scopo di evitare che la rete venga sovraccaricata, adattando dinamicamente la velocità di invio dei dati in base alle condizioni percepite nella rete. TCP non riceve segnali espliciti di congestione dai router (salvo in implementazioni avanzate come ECN), quindi **deduce la congestione tramite l'osservazione dei pacchetti persi o dei ritardi negli ACK**.

1. Slow Start

Quando una connessione TCP inizia, il mittente non conosce ancora la capacità della rete. Per questo motivo, TCP parte con un valore molto basso per la **finestra di congestione** (`cwnd`), inizialmente pari a **1 MSS** (Maximum Segment Size).

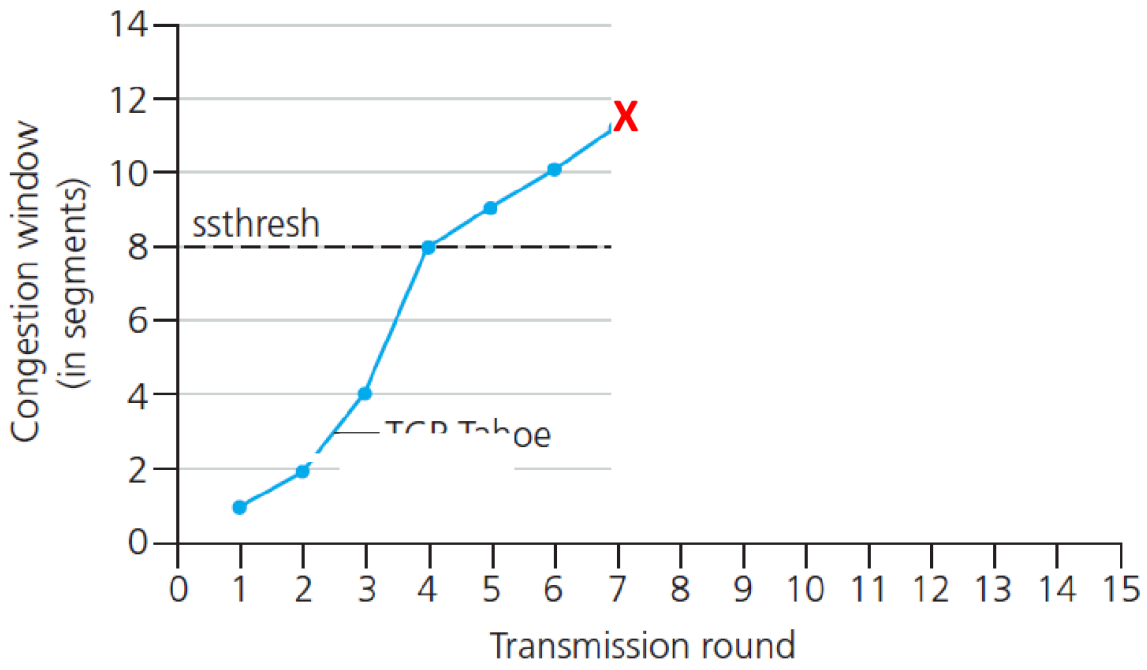


A ogni ACK ricevuto, `cwnd` aumenta, e poiché ogni segmento genera più ACK, la crescita è **esponenziale**: si raddoppia a ogni RTT (Round Trip Time).

Questa fase è chiamata **slow start**: il nome può ingannare, perché in realtà è un aumento molto rapido, ma parte da una base modesta per non rischiare un collasso immediato della rete. La fase termina quando `cwnd` raggiunge una soglia chiamata **ssthresh** (slow start threshold).

2. Congestion Avoidance

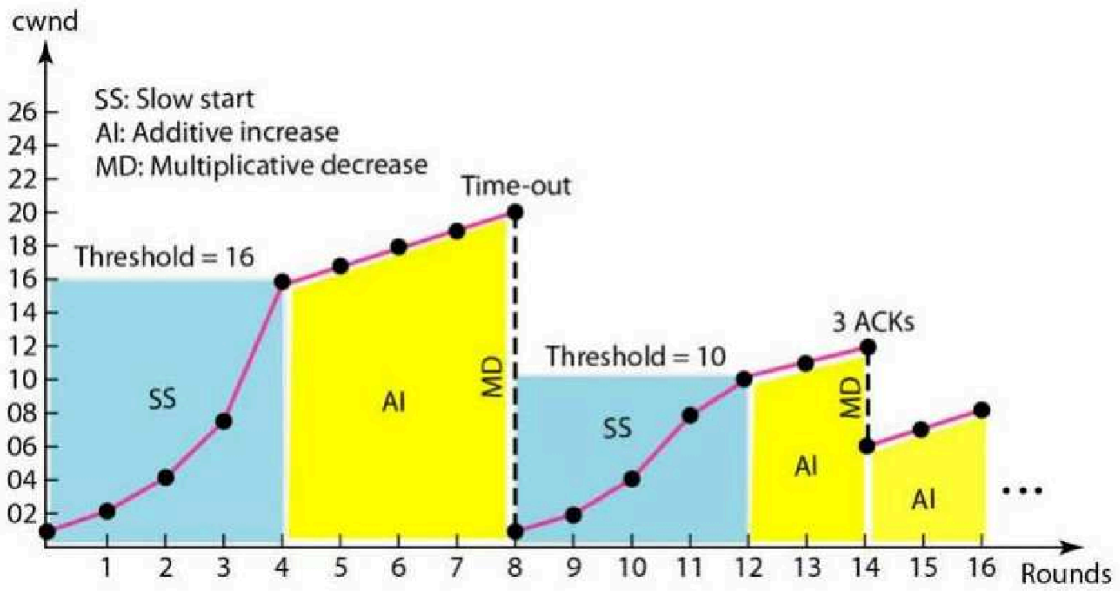
Una volta superata `ssthresh`, TCP entra in modalità **congestion avoidance**, in cui `cwnd` cresce in modo **lineare**, tipicamente di circa 1 MSS per ogni RTT. Questo serve a sondare più cautamente il margine di banda disponibile senza causare congestione.



L'alternanza tra slow start e congestion avoidance permette a TCP di adattarsi progressivamente alla capacità del percorso di rete, senza saturarlo improvvisamente.

3. Reazione alla perdita di pacchetti

La perdita di pacchetti è il principale indicatore di congestione. Quando TCP rileva una perdita tramite un **timeout** (nessun ACK ricevuto), la reazione è drastica: `ssthresh` viene impostata a metà del valore di `cwnd`, e `cwnd` torna a 1 MSS, riprendendo da slow start. Questo comportamento è tipico della variante **TCP Tahoe**.



In alternativa, se TCP riceve **tre ACK duplicati** consecutivi (che indicano che un pacchetto intermedio è andato perso, ma i successivi sono stati ricevuti), entra in **fast retransmit.**

4. Fast Retransmit

Fast retransmit ritrasmette subito il pacchetto perso **senza aspettare il timeout**, dopo, se previsto, entra in **fast recovery**, altrimenti riparte da **slow start**.

`ssthresh` è dimezzato, ma `cwnd` viene impostato a `ssthresh + 3 MSS`. Questo permette di continuare la trasmissione in modalità congestion avoidance, evitando una ripartenza troppo conservativa. Questo comportamento è tipico di **TCP Reno**.

5. Fast Recovery

La **Fast Recovery** è una fase che si attiva immediatamente dopo la **Fast Retransmit**, ed è progettata per evitare di ritornare alla fase iniziale di **Slow Start**, riducendo l'interruzione nella trasmissione dei dati dopo una perdita moderata.

Quando TCP riceve **3 ACK duplicati**, capisce che un segmento è stato perso ma che gli altri stanno ancora arrivando correttamente (i pacchetti successivi al perduto sono stati ricevuti). In risposta:

1. TCP **esegue subito la ritrasmissione** del segmento mancante (Fast Retransmit).
2. TCP **non azzera** la finestra di congestione (`cwnd`), ma:
    - imposta **ssthresh = cwnd / 2**;

- imposta **cwnd = ssthresh + 3·MSS**, per tenere conto dei tre ACK duplicati che segnalano segmenti ricevuti.

3. Per ogni ulteriore ACK duplicato, TCP **aumenta cwnd** di 1 MSS → continua a inviare nuovi segmenti (come se gli ACK fossero conferme implicite).
4. Quando riceve finalmente un **ACK cumulativo** per il pacchetto originariamente perso, TCP:
   - imposta `cwnd = ssthresh`,
   - passa alla fase **Congestion Avoidance**.

Questa strategia permette di **continuare la trasmissione** anche durante una perdita singola, evitando un'interruzione totale come accadrebbe con il timeout. È un comportamento introdotto da **TCP Reno**, ed è più efficiente di quello di **Tahoe**, che tornava sempre a slow start.

Differenze tra TCP Reno e TCP Tahoe

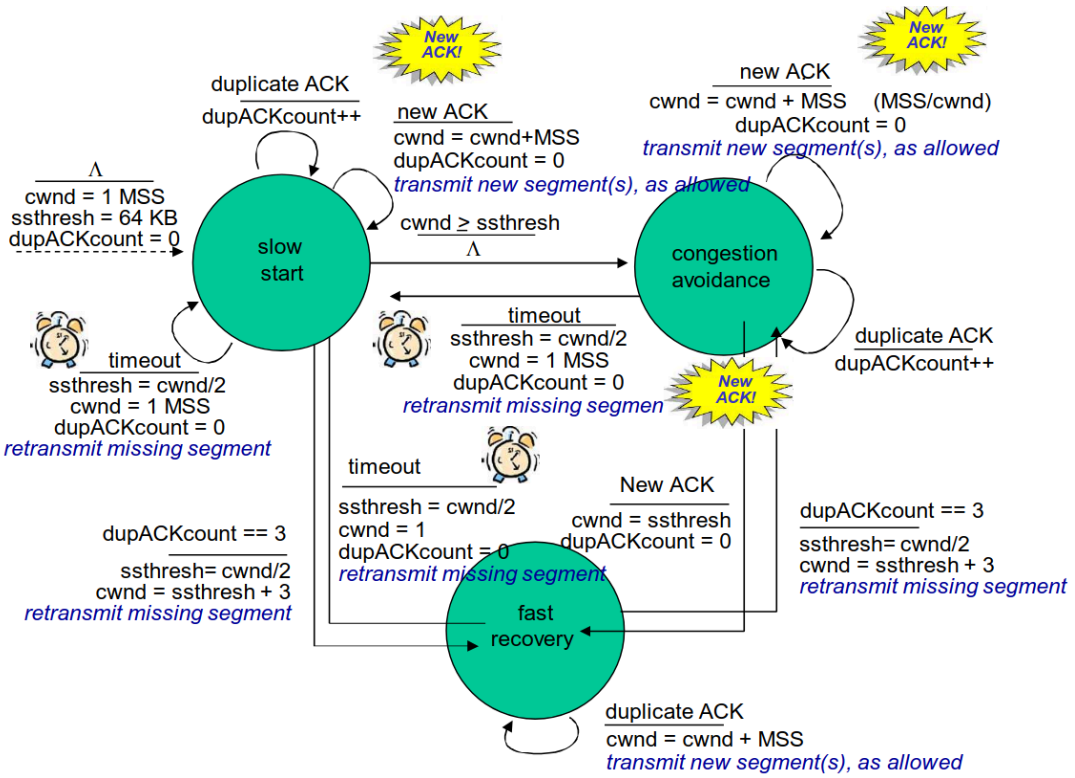
	TCP Tahoe	**TCP Reno**
Fast Retransmit	Sì	Sì
Fast Recovery	❌ Non presente	✅ Presente
Reazione ai 3 ACK duplicati	Ritrasmette il pacchetto perso e riparte da slow start (`cwnd = 1`)	Ritrasmette il pacchetto, dimezza `cwnd`, poi entra in congestion avoidance
Reazione al timeout	`cwnd = 1`, `ssthresh = cwnd / 2`	`cwnd = 1`, `ssthresh = cwnd / 2`
Efficienza	Più conservativo, minor throughput	Più aggressivo, maggior throughput in reti stabili
Convergenza	Lenta dopo perdita	Più rapida (evita ripartenza da 1)
Fase post-perdita	Sempre Slow Start	Fast Recovery, poi Congestion Avoidance

Schema del comportamento di TCP

Nel tempo, il comportamento di `cwnd` assume una forma a **dente di sega (sawtooth)**:

- **Sale lentamente (lineare)** durante congestion avoidance,
- **Crolla bruscamente (moltiplicativamente)** quando si rileva una perdita,
- **Riprende l'ascesa**, sondando la rete con cautela.

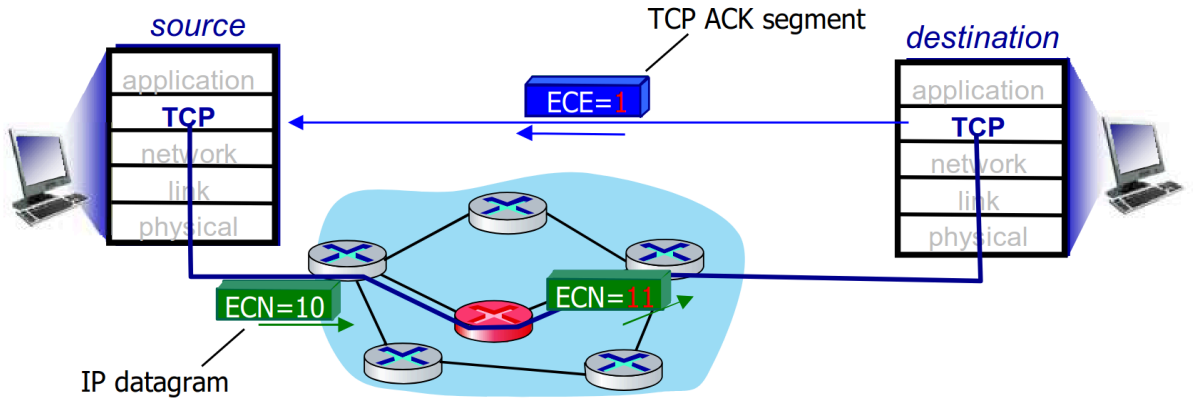
Questa dinamica implementa l'algoritmo **AIMD (Additive Increase / Multiplicative Decrease)**, che si è dimostrato **stabile, equo e scalabile**, permettendo a più connessioni TCP di coesistere e condividere in modo efficiente la banda.



ECN – Explicit Congestion Notification

La **ECN (Explicit Congestion Notification)** è una tecnica avanzata di controllo della congestione che consente ai **router** di segnalare la presenza di congestione **senza dover scartare i pacchetti**, come accade invece nei meccanismi tradizionali. È una soluzione di tipo **network-assisted**, perché richiede il supporto esplicito dei router e degli host TCP coinvolti.

Nelle versioni classiche di TCP, la congestione viene rilevata indirettamente solo quando un pacchetto viene perso – perdita che TCP interpreta come segnale di sovraccarico della rete. ECN invece consente di rilevare la congestione **prima che si verifichi la perdita**, migliorando la reattività e l'efficienza della comunicazione.



Il meccanismo funziona tramite **due bit nel campo TOS (Type of Service)** dell'intestazione IP (IPv4 o IPv6), che i router marcano quando rilevano congestione (ad esempio sulla base dell'occupazione dei buffer). Quando un router nota che un pacchetto sta attraversando una coda congestionata, invece di scartarlo, **imposta i bit ECN** nel pacchetto IP.

Se il mittente e il destinatario TCP supportano ECN (funzionalità negoziata durante il three-way handshake), il destinatario riceve il pacchetto marcato e **avverte il mittente** includendo una notifica nel successivo ACK. Il mittente TCP, ricevendo questa segnalazione, **riduce la finestra di congestione** esattamente come farebbe in presenza di una perdita, **senza però la necessità di ritrasmettere**.

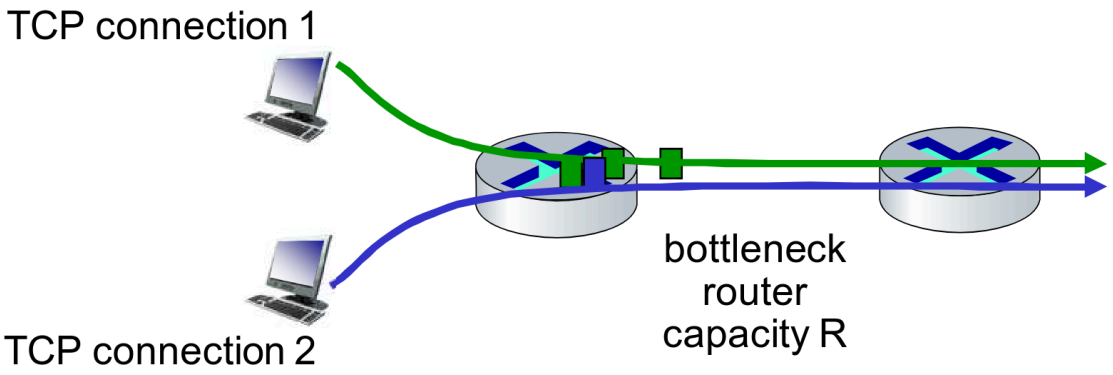
In questo modo ECN permette di:

- **ridurre le perdite di pacchetti**,
- mantenere più alta l'efficienza della rete,
- migliorare le prestazioni in scenari a bassa latenza e buffer limitati.

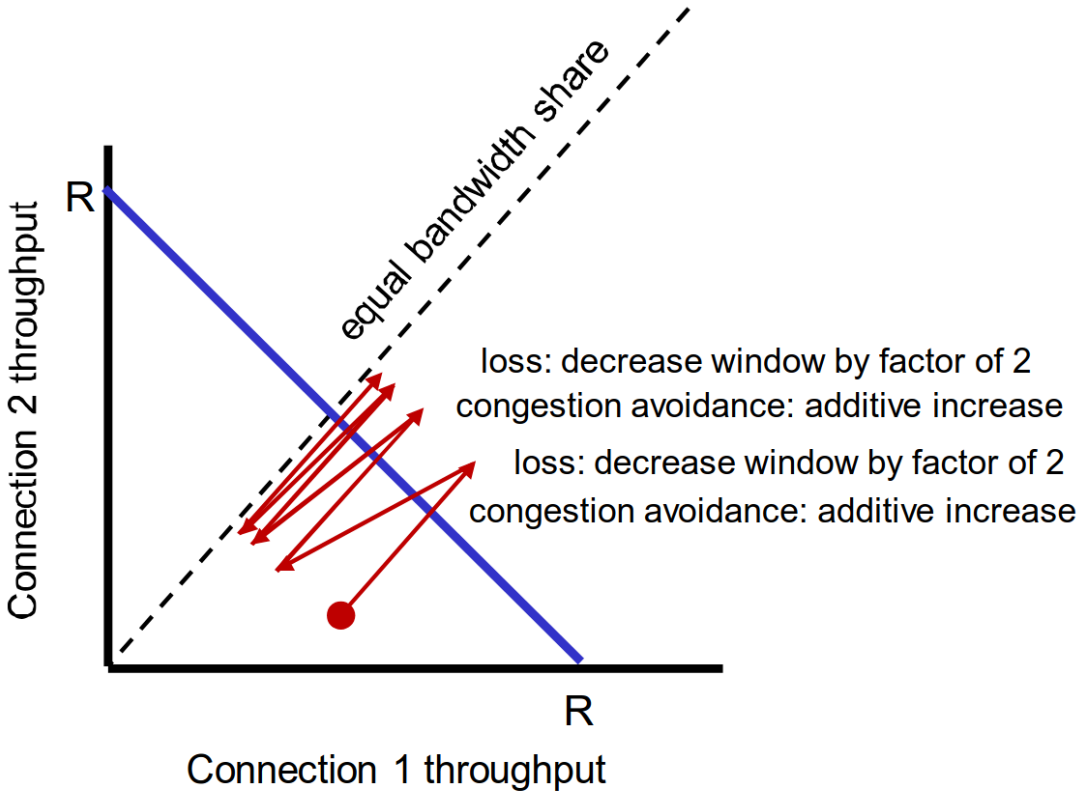
Tuttavia, l'utilizzo di ECN richiede che **router, mittente e destinatario** siano tutti compatibili e configurati per supportarlo, cosa che ne ha limitato la diffusione su Internet fino a tempi relativamente recenti.

2.5.9 Fairness

Quando più connessioni TCP condividono lo stesso collegamento, l'obiettivo è che ognuna riceva una porzione equa della banda disponibile. Se ad esempio due connessioni condividono un link di capacità `R`, ci si aspetta che ciascuna possa ottenere circa `R/2` in condizioni ideali. Questo concetto viene chiamato **fairness**, ovvero equità tra flussi.

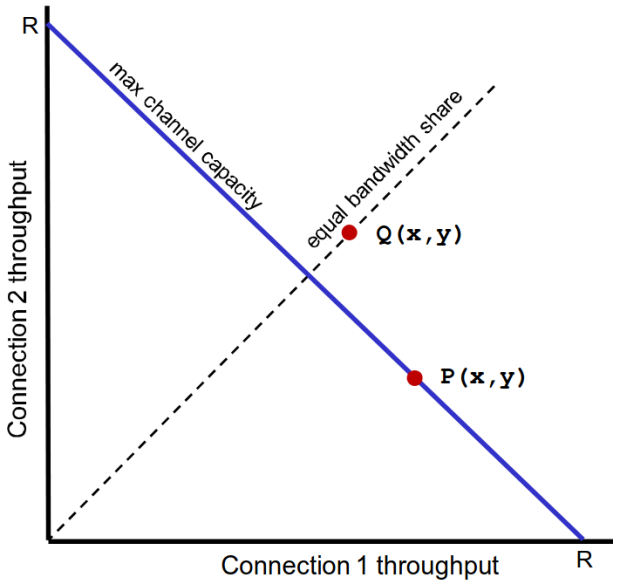


Il meccanismo che permette questo comportamento è l'**AIMD (Additive Increase / Multiplicative Decrease)**. Durante la fase di congestion avoidance, ogni connessione aumenta la propria finestra `cwnd` in modo lento e costante (aumento additivo), ma in caso di perdita la riduce in modo proporzionale (riduzione moltiplicativa). Questo tipo di risposta fa sì che i flussi TCP si avvicinino nel tempo a una divisione più equa della banda.

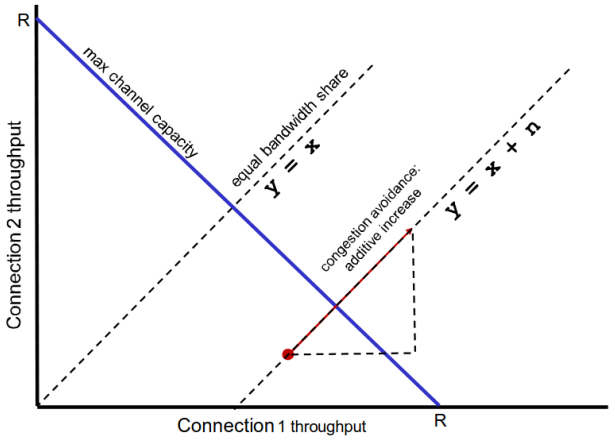


La retta $x + y = R$ (retta blu) rappresenta il limite massimo del canale, mentre $x = y$ (retta tratteggiata) rappresenta la perfetta equità. Il comportamento di **AIMD tende a far convergere i due flussi verso il punto centrale sulla retta**, dove $x = y = R/2$.

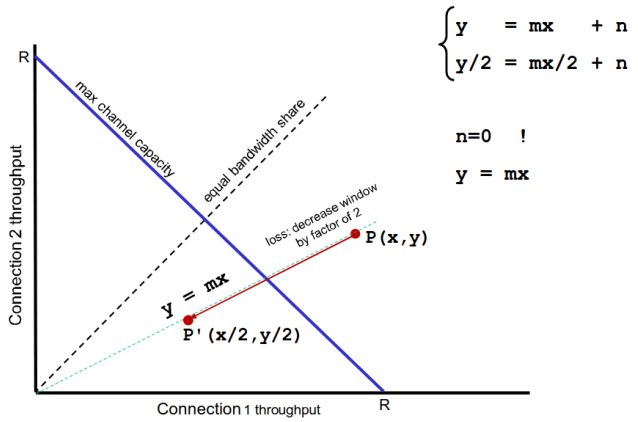
Questo avviene perché le crescite additive e le riduzioni moltiplicative "spingono" le connessioni verso l'equilibrio.



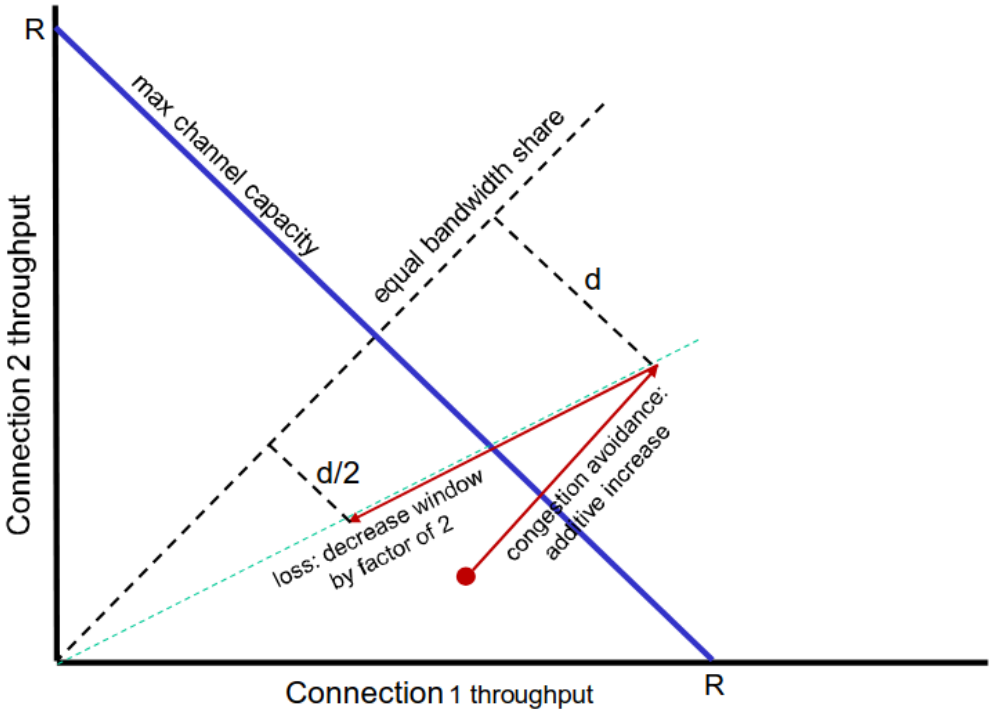
Gli assi x e y indicano i throughput di due connessioni TCP che condividono lo stesso link. La retta $x + y = R$ rappresenta la capacità totale del canale. Quando entrambe le connessioni aumentano `cwnd`, il punto (x, y) si muove in diagonale verso l'alto (additive increase). Quando avviene una perdita, entrambe riducono `cwnd` moltiplicativamente, e il punto si sposta verso l'origine (multiplicative decrease).



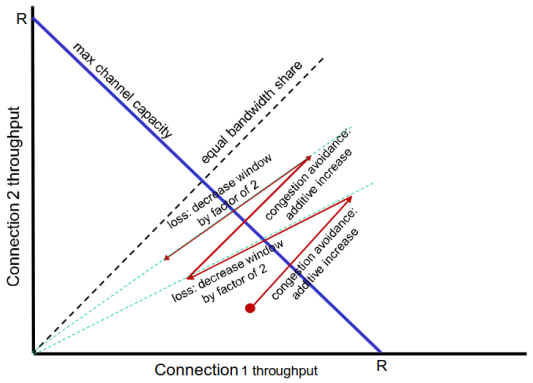
Ripetendo questo ciclo, il punto converge verso `(R/2, R/2)`, cioè una situazione di **massima equità ed efficienza**, senza bisogno di coordinamento diretto tra i flussi.

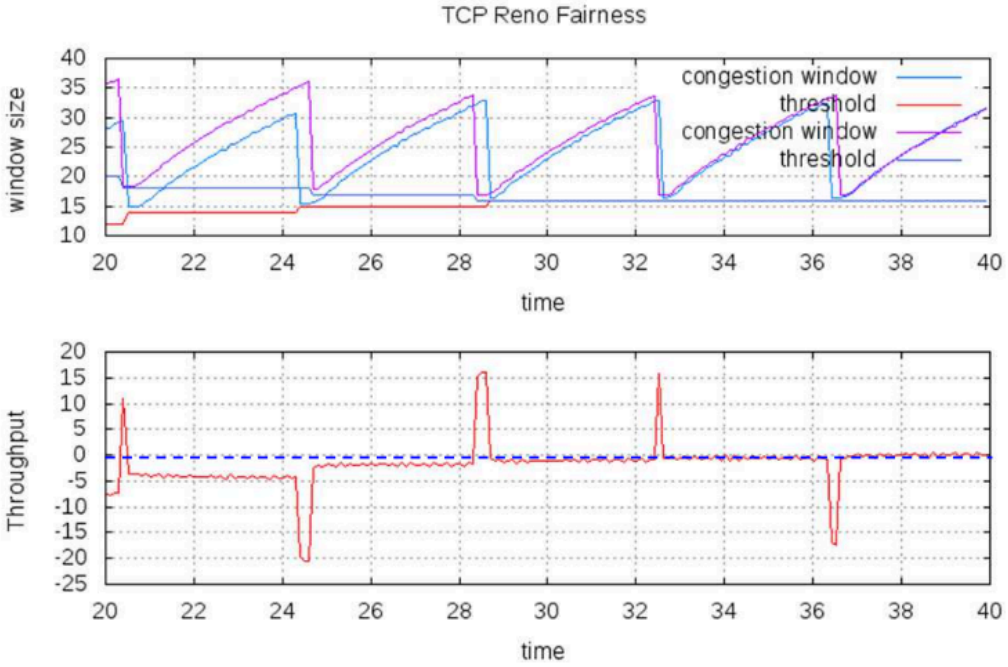


Tuttavia, la fairness non è sempre garantita. Se due connessioni TCP hanno **RTT diversi**, quella con RTT minore tenderà ad aumentare `cwnd` più rapidamente, ottenendo una fetta maggiore della banda.



Inoltre, se un'applicazione crea flussi TCP multipli (es. download paralleli), può ottenere più banda rispetto a un singolo flusso TCP


R
max channel capacity
equal bandwidth share
Connection 2 throughput
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Connection 1 throughput
R


TCP Reno Fairness
window size
congestion window
threshold
congestion window
threshold
time
Throughput
time