# LIVELLO TRASPORTO
# Protocollo TCP

# TCP

## *Transmission Control Protocol*

## RFC: 793, 1122, 1323, 2018, 2581, 2988

# Protocollo TCP

- **Come discusso, il livello trasporto estende le funzionalità del protocollo IP permettendo la comunicazione tra due processi applicativi**

- **Rispetto a UDP, il protocollo TCP (*Transmission Control Protocol*) aggiunge:**
  - **Garanzie di comunicazioni affidabili**
  - **Offre un paradigma di comunicazione**
    - **Orientato alla connessione (connection-oriented)**
    - **Orientato allo scambio di un flusso di dati [di byte]**

# Problema del trasferimento affidabile

- Il problema del **trasferimento affidabile di dati su infrastruttura inaffidabile** riguarda un contesto più generale del livello TCP
  - E' uno dei problemi principali del networking!
  - Esiste anche a livello 2 e a livello 7 (applicazioni)

- **Il livello di inaffidabilità del canale di comunicazione determina la complessità del protocollo che deve gestire tale inaffidabilità**

# **Affidabilità** su canali inaffidabili [1]

- Vogliamo creare un **canale di comunicazione affidabile «virtuale»**
  - **Il «vero» canale è inaffidabile, ma tutta la complessità per «costruire» l'affidabilità non è visibile agli utilizzatori**
- Capacità di **rilevare** un problema di trasferimento
  - Quali problemi possono verificarsi? Come accorgersene?
- Capacità di **rimediare**
  - **Tecniche ritrasmissione dei dati** fino a che il trasferimento non avviene correttamente

# Affidabilità su canali inaffidabili [2]

- **Rilevare**
  - L'uso di **checksum** permette di garantire integrità rispetto a **errori di trasmissione**
  - Necessario però introdurre ulteriori meccanismi per rilevare altre possibili problemi, che dipendono dal paradigma di comunicazione
    - **Perdite**
    - **Duplicati**
    - **Consegne non in ordine**

# Affidabilità su canali inaffidabili [3]

- TCP fornisce tutte le garanzie di affidabilità, ma altri protocolli a livello applicativo potrebbero richiederne solo una parte
  - Ad esempio, alcune applicazioni potrebbero non essere sensibili rispetto a dati fuori ordine
  - *Approfondiremo come TCP offre queste garanzie in queste slide*
- In questi casi il protocollo applicativo può utilizzare UDP e implementare un protocollo di trasporto che implementi soltanto alcune garanzie

# Cenni di affidabilità di TCP [1]

- ***Rilevare (e rimediare) alla Perdita di pacchetti*:**
  **acknowledgment + time-out + (ritrasmissione)**
  - Ogni trasmissione andata a buon fine viene notificata (**acknowledged**) dall'host ricevente
  - Se l'host mittente non riceve un acknowledgement entro un intervallo di tempo predefinito (***time-out***), il mittente ritrasmette i dati
  - Acknowledgment e ritrasmissioni dovute ad eventuali perdite sono gestite in modo trasparente rispetto al processo applicativo

# Trasporto orientato alla connessione [1]

- Avevamo già parlato di paradigmi di comunicazione **a commutazione di circuito** (ovvero, basati su instaurazione di connessione) in contesti H2N (e.g., PPP)

- Un protocollo orientato alla connessione ha 3 fasi
  - **Apertura** della connessione
  - **Utilizzo** della connessione (Scambio dati)
  - **Chiusura** della connessione

# Trasporto orientato alla connessione [2]

- A livello trasporto il problema si affronta in un contesto «logico»
  - Ci basiamo su IP, che non ha concetto di connessione, quindi **non c'è comunque allocazione di risorse sul canale fisico**
- Instaurare la connessione l'esecuzione di vari controlli, ad esempio **verificare la disponibilità dell'altro partecipante** (nella comunicazione) e stabilire uno **stato della connessione fra i due partecipanti**
  - Mantenere uno stato significa mantenere delle risorse, che vengono liberate solo dopo la chiusura

# Trasporto orientato allo scambio di un flusso (stream) di dati (byte) [1]

- TCP vuole offrire un'interfaccia per comunicare secondo un paradigma «più evoluto» rispetto al semplice scambio di pacchetti

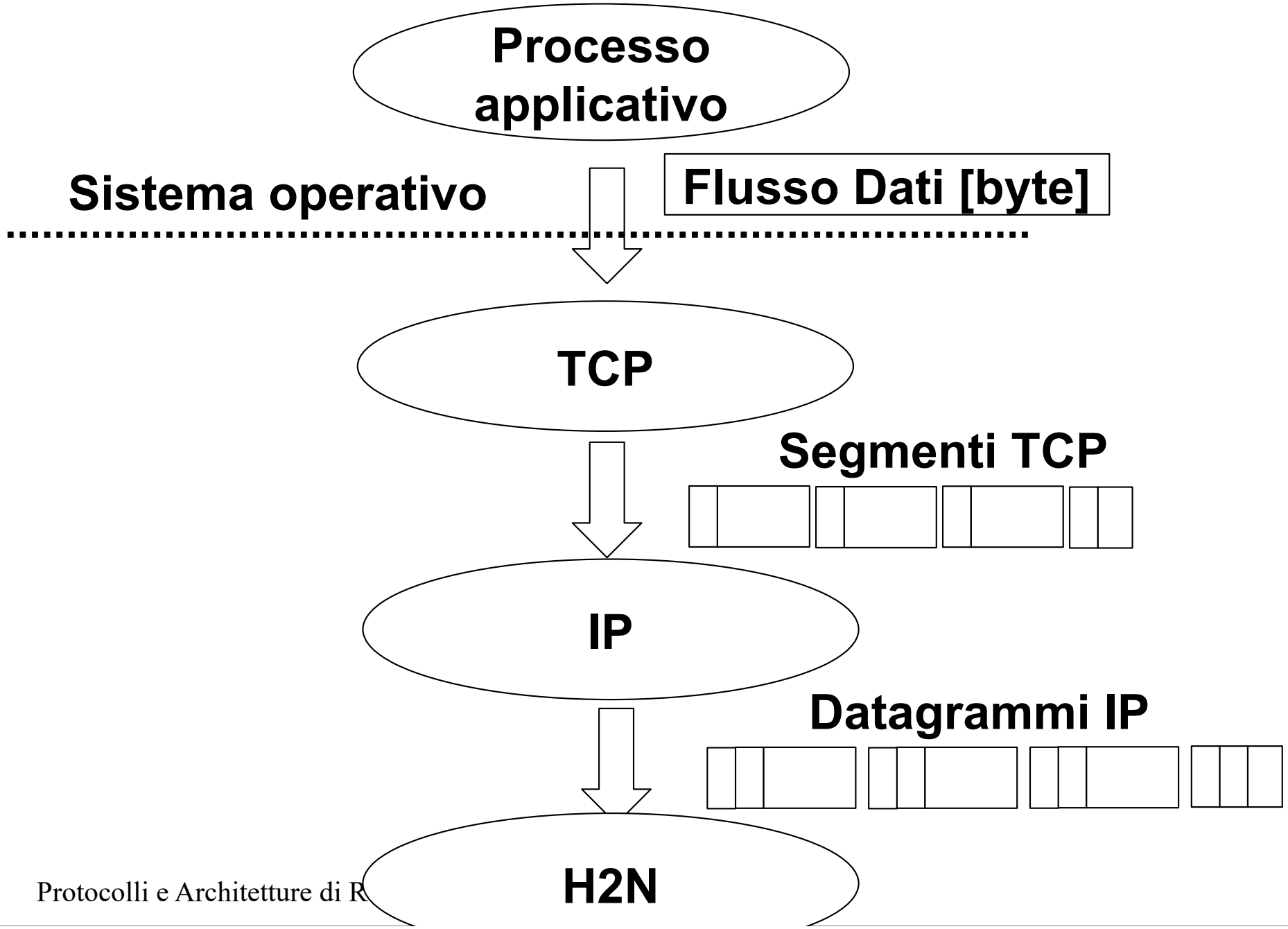
- Flusso di dati o byte (*byte stream*): il protocollo accetta come input una quantità di dati arbitraria da inviare
  - Non ci sono limiti del pacchetto IP (vincolo per UDP)
  - Non ci sono limiti per via del protocollo H2N

# Trasporto orientato allo scambio di un flusso (stream) di dati (byte) [2]

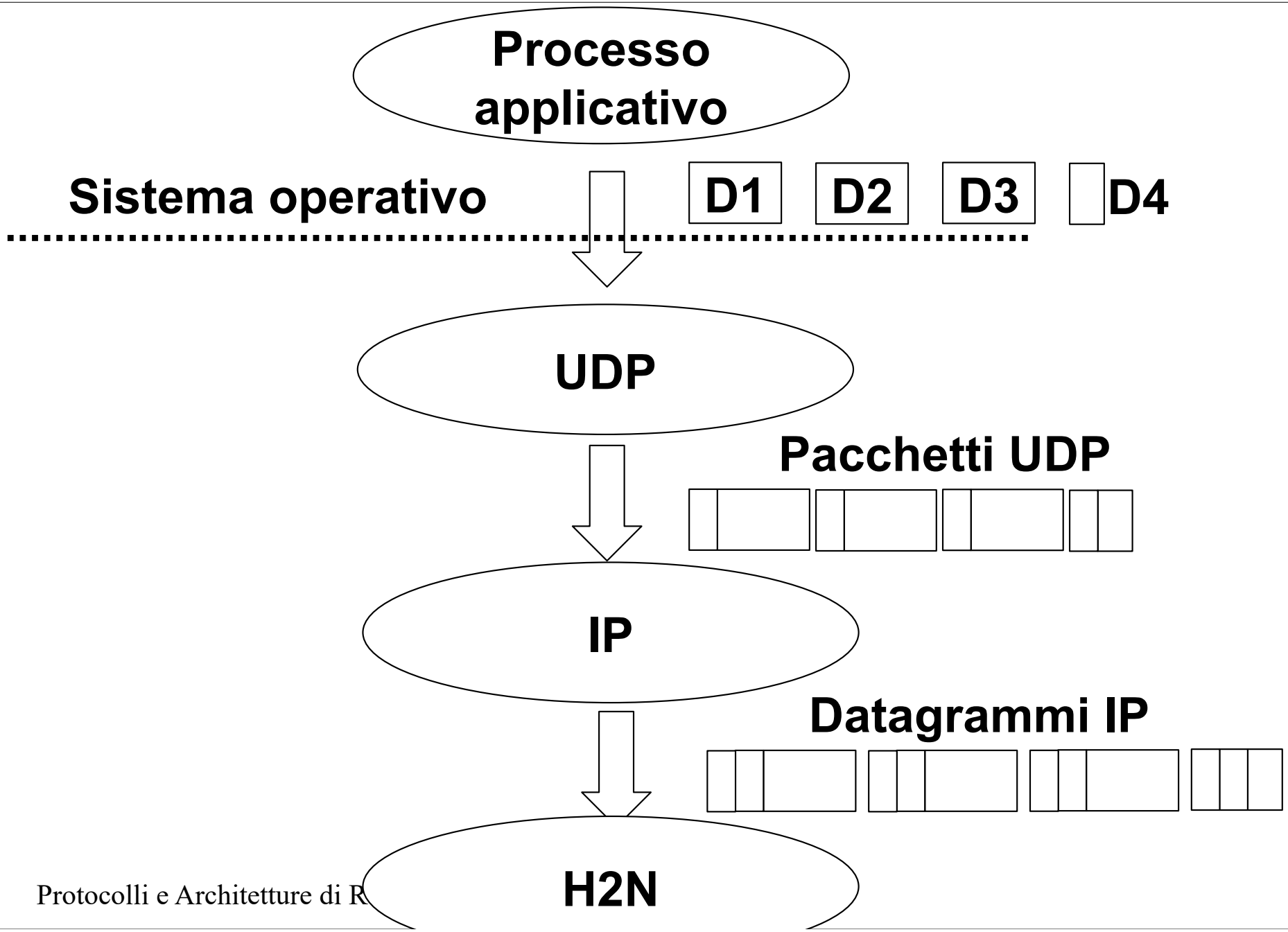
- TCP offre una comunicazione orientato al flusso dati, ma si basa su IP che è basato su pacchetti

- TCP implementa la logica di **segmentazione** del flusso di dati in pacchetti
  - Il flusso dati viene separato in tanti **segmenti**
    - Discuteremo a breve come deciderne la dimensione
  - Viene chiamato **segmento** il blocco di dati che TCP invia a IP per costruire il pacchetto

# Trasporto orientato allo scambio di un flusso (stream) di dati (byte) [3]



# Distinguere da UDP che offre lo stesso paradigma orientato ai pacchetti



# Affidabilità di TCP

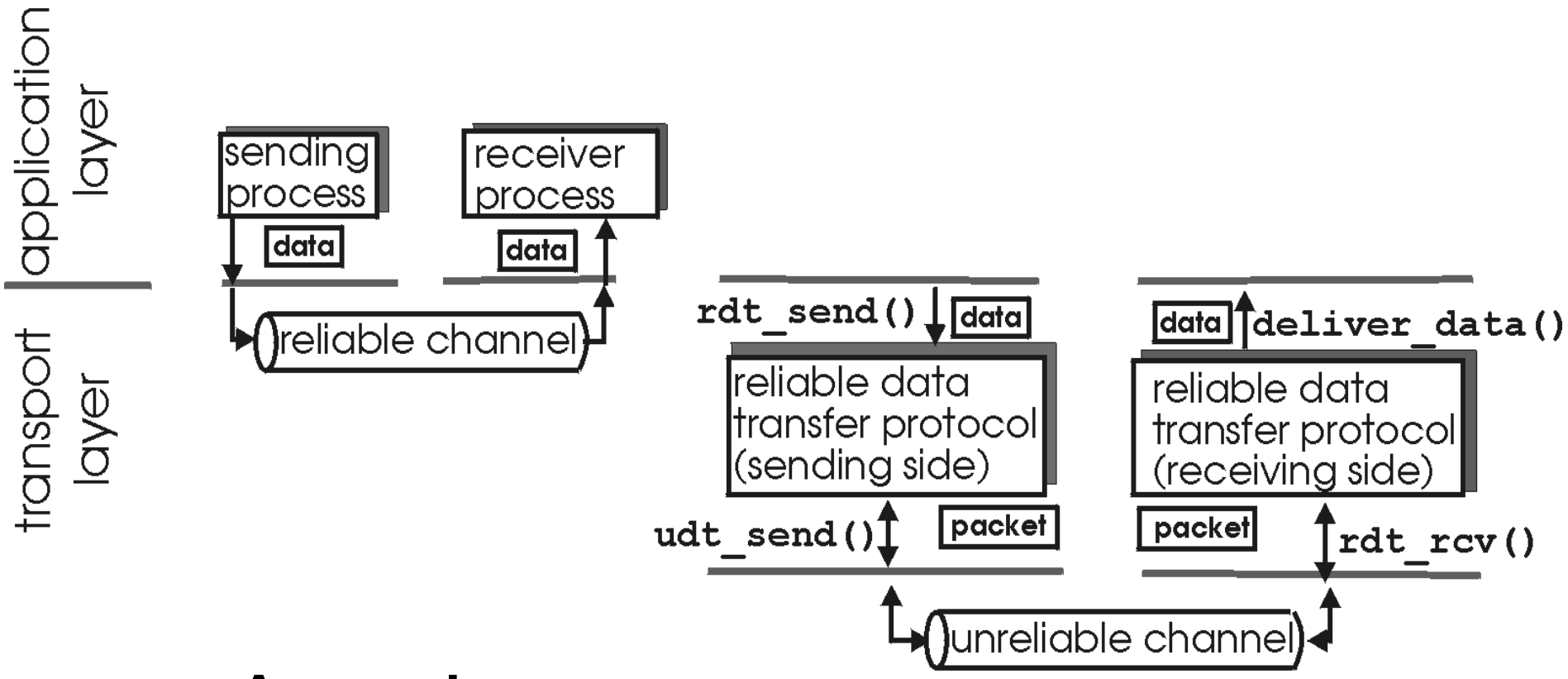
- Le caratteristiche di affidabilità dipendono dal **paradigma di comunicazione**
- TCP: un protocollo affidabile orientato alla connessione e orientato a flusso (di byte) deve **prevenire l'alterazione dell'intero flusso**
- L'apertura e la chiusura della connessione devono essere anch'essi affidabili
  - In TCP vedremo l'eccezione della chiusura con RST
- Tutti i segmenti devono arrivare a destinazione:
  - Senza perdite, individuando e scardando i duplicati, riordinando eventuali pacchetti consegnati fuori ordine

# Cenni di Affidabilità di TCP [2]

- Rileviamo errori di trasmissione con checksum
- Ritrasmettiamo pacchetti persi grazie all'uso di acknowledgment e timeout

- Identifichiamo univocamente ciascun segmento (**numeri di sequenza – sequence number)** per consentire al destinatario di:
  - **identificare e scartare segmenti duplicati**
  - **riordinare i segmenti prima di inoltrarli al processo destinatario in caso di consegna fuori ordine**

# Applicazioni e trasporto



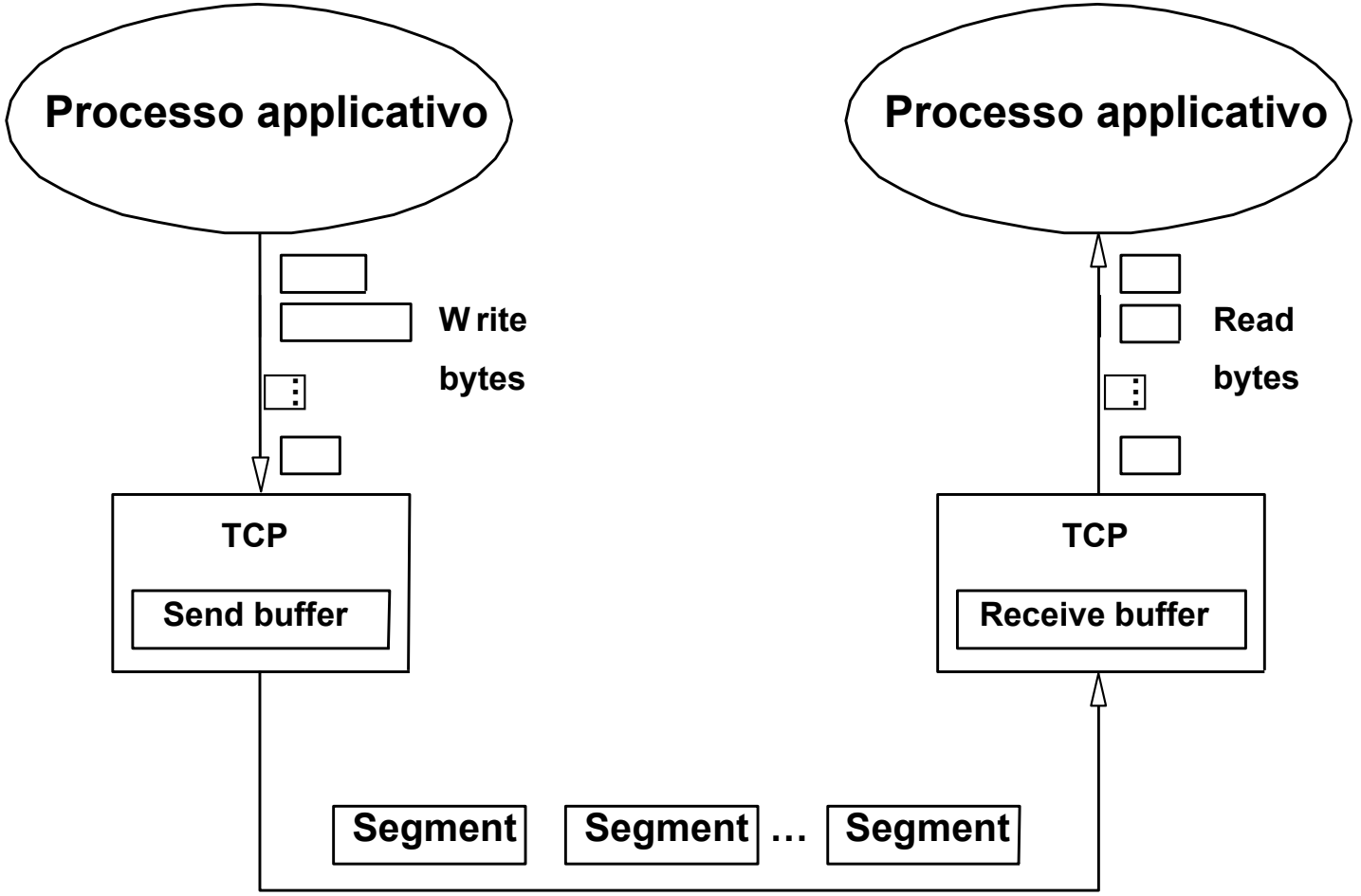

**Assunzione delle applicazioni**

**Realtà implementativa**

# Protocollo TCP

- **Altri dettagli del protocollo TCP**
  - ***trasferimento con buffer***: le logiche di gestione vengono gestite tramite l'utilizzo di buffer di memoria gestiti dal sistema operativo
  - ***connessione full duplex*** (bi-direzionale): una volta instaurata una connessione, è possibile il trasferimento contemporaneo in entrambe le direzioni della connessione
  - ***controllo di flusso e congestione***: il protocollo regola la «velocità» dei dati scambiati (throughput) rispetto alla capacità delle entità comunicanti (flusso) e della rete (congestione)

# Trasmissione dati nel TCP




Trasmissione segmenti

# Buffering: *asincronia*

- **Il TCP fa parte del Sistema Operativo, non del livello applicativo che gestisce l'invio e la ricezione dei dati**
- Deve tener conto di tutti gli eventi che si verificano in modo **asincrono**:
  - Il livello TCP del mittente non sa quando il processo applicativo deciderà di spedire dati
  - Il livello TCP del destinatario non sa quando il processo applicativo accetterà (o chiederà) di prendere i dati arrivati
- Nota: il SO decide quando inviare dati in base a diversi criteri
  - Ad esempio: dimensioni dei segmenti, «urgenza» dei dati, controllo della velocità di invio

# Cosa il TCP non garantisce

- Comunicazioni in *tempo reale*
- Garanzia di disponibilità di banda tra mittente e destinatario
- Multicast (un mittente, molti destinatari) affidabile

# Problemi da affrontare a livello TCP

- **Eterogeneità degli host e dei processi in comunicazione**
  → c'è bisogno di un meccanismo per attivare e concludere una comunicazione in modo esplicito per entrambi
- **Eterogeneità dei tempi di trasmissione**
  → c'è bisogno di un meccanismo di *timeout adattativo*
- **Possibilità di ritardi molto lunghi nella rete**
  → c'è bisogno di gestire il possibile arrivo di pacchetti molto vecchi
- **Possibilità di avere un host destinatario con capacità computazionale e di memoria diversa dall'host mittente**
  → c'è bisogno di gestire capacità dei nodi eterogenee
- **Possibilità di avere connessioni di rete molto diverse**
  → c'è bisogno di gestire possibili congestioni dovute alla rete

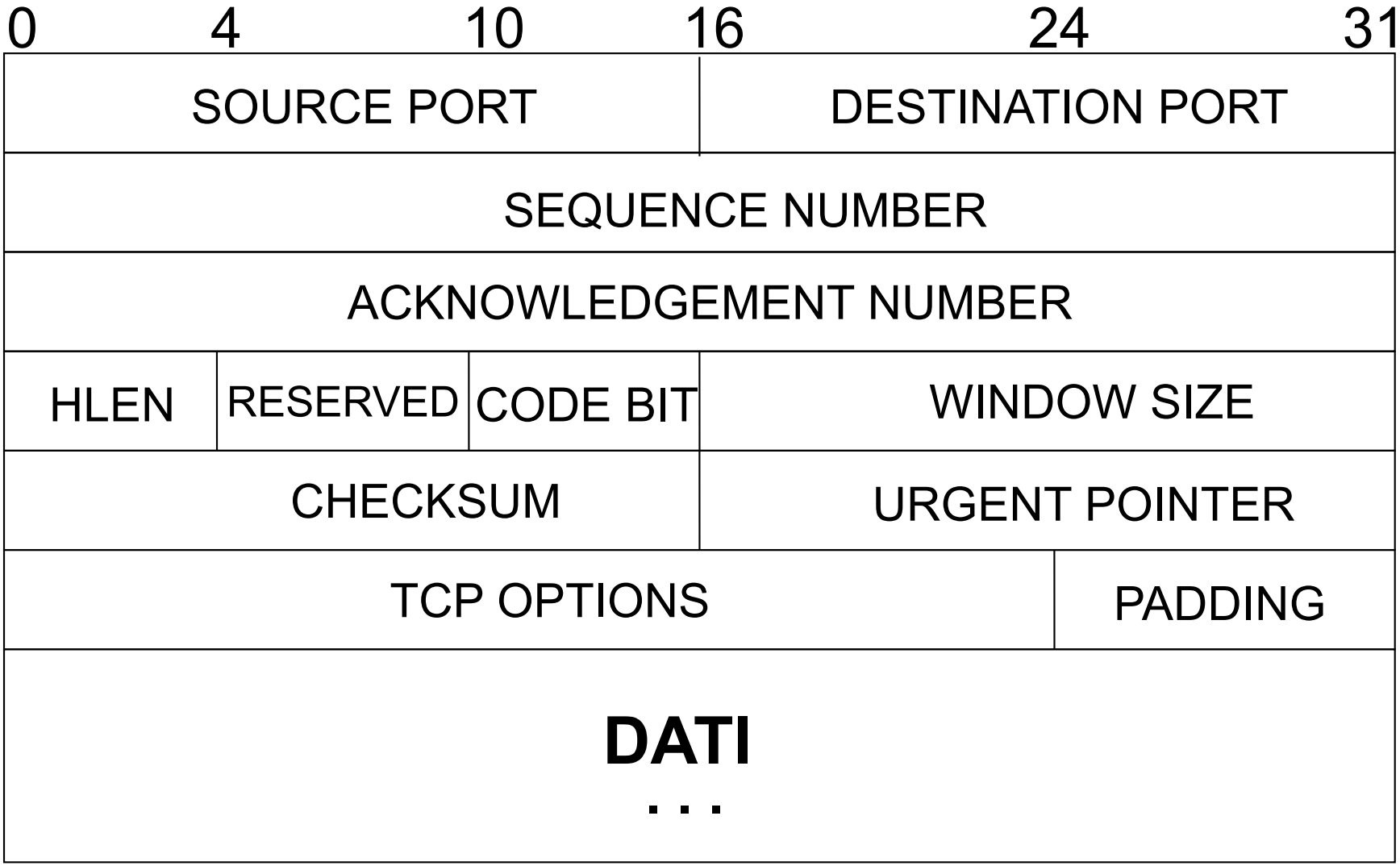
# Segmento TCP

# Segmento TCP

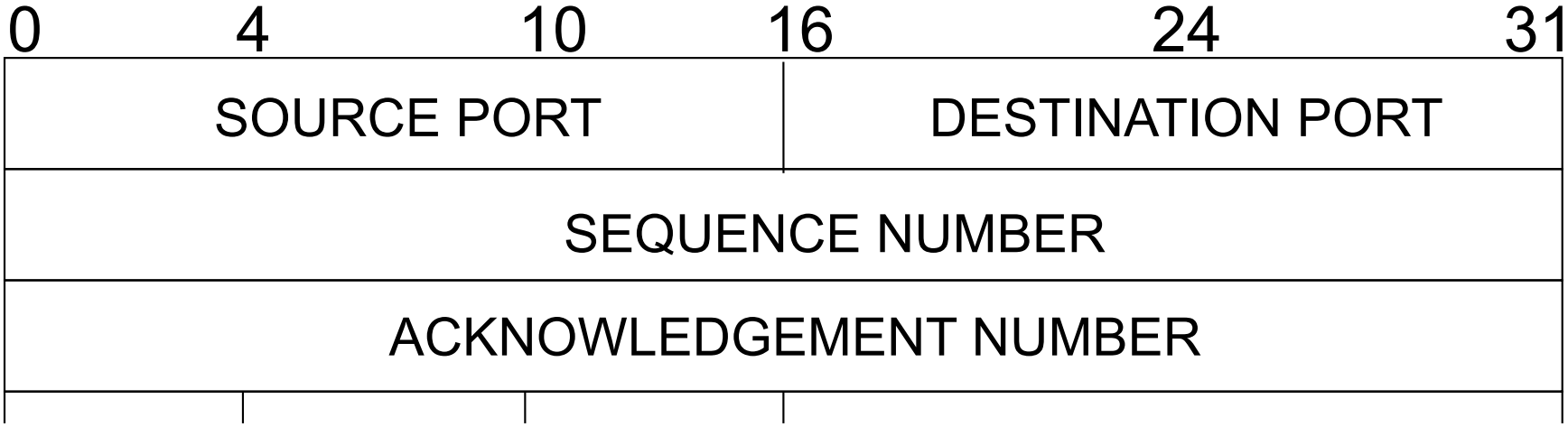
- L'insieme di dati che il livello TCP chiede di trasferire al livello IP è detto ***segmento TCP***

- Ogni segmento TCP contiene:
  - ***Payload*: dati del byte stream**
  - ***Header*: metadati e informazioni di controllo**

# Formato del segmento TCP

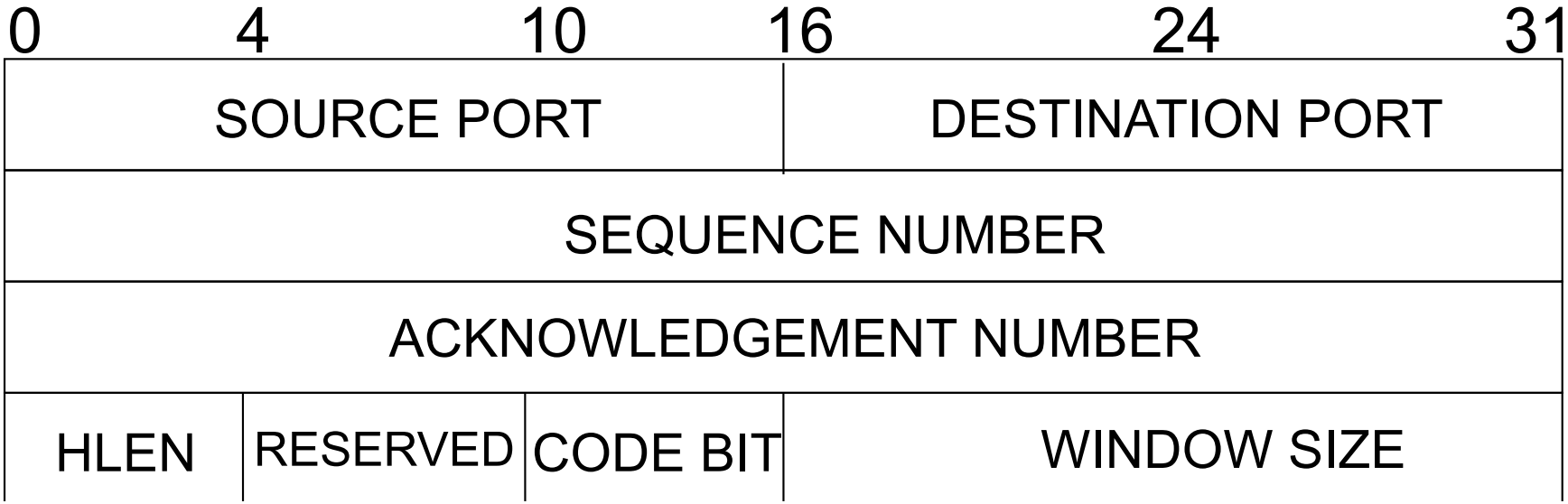


# Formato del segmento TCP




- **source port** (16 bit): numero di porta del mittente
- **destination port** (16 bit): numero di porta del destinatario
- **sequence number** (32 bit): numero di sequenza relativo al flusso di byte che si sta trasmettendo
- **acknowledgement number** (32 bit): ACK relativo ad un numero di sequenza del flusso di byte che si sta ricevendo (poiché il flusso è bi-direzionale, vi è la possibilità di ***piggybacking*** → si vedrà in seguito)

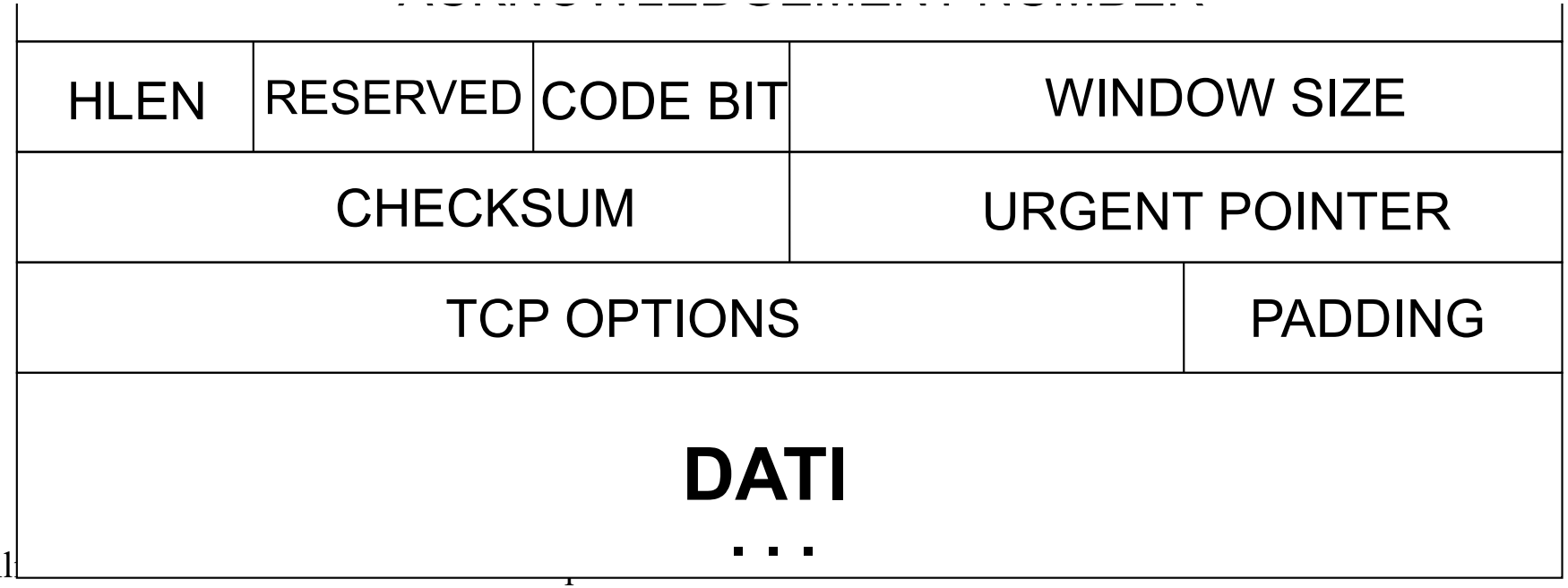
# Formato del segmento TCP




- **hlen** (4 bit): lunghezza dell'header TCP (in multipli di 32 bit) se non vi sono opzioni $\rightarrow$ hlen = 20 byte
- **reserved** (4 bit): per usi futuri
- **code bit** (6 bit): scopo e contenuto del segmento
  - **URG** (urgent): dati segnati come urgenti dal livello applicativo
  - **ACK** (acknowledgement): valore del campo acknowledgement è valido
  - **PSH** (push): il destinatario deve passare i dati all'applicazione immediatamente
  - **SYN** (synchronize), **FIN**, **RST** (reset): usati per instaurazione, chiusura ed interruzione della connessione

# Formato del segmento TCP

- **window size** (16 bit): dimensione della finestra in ricezione (indica il numero di byte che si è disposti ad accettare in ricezione)
- **checksum** (16 bit): controllo integrità dei dati trasportati nel segmento TCP (del tutto analogo al caso del protocollo UDP)
- **urgent pointer** (16 bit): puntatore al termine dei dati urgenti (utilizzato raramente, nel caso di trasmissione di caratteri speciali)
- **TCP options**: campo opzionale di lunghezza variabile (serve a negoziare la dimensione del ***segmento massimo scambiato*** → **MSS**)
- **zero padding**: per header con lunghezza multipla di 32 bit (se opzioni)



# Checksum TCP

**Utilizzato per rilevazione errori nei dati trasportati:**

calcolato usando un maggior numero di informazioni di quelle presenti nell'header TCP (vi sono anche informazioni IP)

$\rightarrow$ definizione di uno ***pseudo-header*** TCP


- ***zero padding***: dimensione dello pseudo-header, multiplo di 32 bit
- ***protocollo***: campo protocollo del datagram IP
- pseudo-header anteposto al segmento TCP
- checksum calcolato su pseudo-header e intero segmento TCP
- pseudo-header *non* è trasmesso dal mittente

# Dati urgenti
## (trasmissione “out of band”)

- Servono a trasportare segnali speciali come **^C**, **^Z**, … in modo che possano essere recapitati immediatamente al processo applicativo
- All’arrivo all’host destinatario, scavalcano lo stream e vengono recapitati immediatamente al processo applicativo
- Il puntatore punta alla fine del blocco dei dati urgenti
- I dati urgenti iniziano all’inizio del segmento

# Negoziazione del MSS

- Le TCP OPTIONS consentono di negoziare il **Maximum Segment Size** (**MSS**) per
  - garantire che il segmento entri nei rispettivi buffer
  - evitare il più possibile la frammentazione al livello h2n
  - sfruttare al meglio la banda

**MSS troppo piccolo** → overhead eccessivo dovuto agli header

**MSS troppo grande** → elevati rischi di frammentazione nell'attraversamento dei livelli dello stack sottostanti IP-h2n (ACK inviato a livello del segmento originale. Quindi, anche se un solo frammento viene perso, tutto il segmento deve essere riinviato)

**➔ Default MSS = 536 byte**

# Numeri di sequenza e acknowledgment

L'obiettivo è di rendere praticamente nulla la probabilità che sia presente un segmento identificato con lo stesso numero appartenente a una connessione precedente con identici numeri di porta

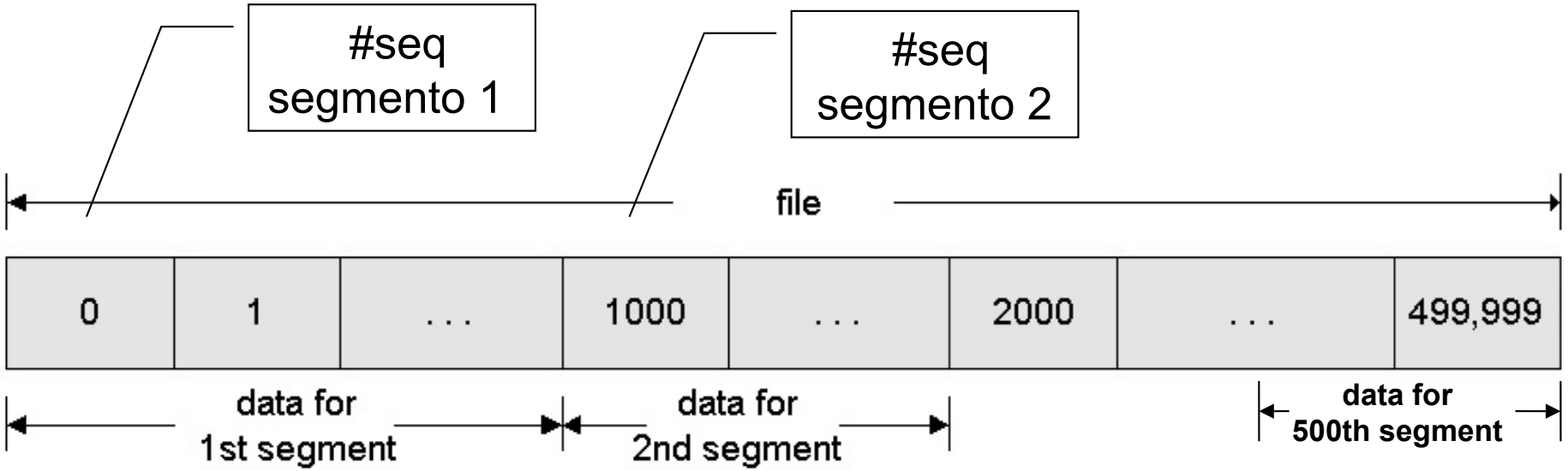
**Numero di sequenza per segmento TCP dato da:**

- Il primo *Initial Sequence Number* (ISN) di un flusso è pseudo-casuale (con numerazione iniziale basata su timer locale con tick=4 $\mu$s)
- I successivi *sequence number* sono offset del primo byte del flusso dati inviati dal mittente
- L'intervallo dei numeri di sequenza (32 bit) garantisce che lo stesso numero non sia riutilizzato prima di qualche ora
- Grazie al TTL dei pacchetti IP, segmenti con lo stesso numero non coesisteranno sulla rete

# Esempio

Si supponga di dover trasferire un file di 500000 byte, con MSS=1000 byte.

- ISN = X (numero psudo-casuale)
- Numeri di sequenza: X, X+1000, X+2000, …



# Numeri sequenza e acknowledgment (2)

## Il numero di acknowledgment per un segmento TCP:

- TCP è full duplex: l'host A può ricevere dati dall'host B mentre sta inviando dati a B sulla stessa connessione
- Segmento da B a A:
  - **numero di sequenza**: numero sequenziale del byte del flusso dati
  - **numero di acknowledgment**: numero di sequenza del successivo byte che A si aspetta di ricevere da B (tutti i byte precedenti sono stati ricevuti (***acknowledgement incrementale***)

# Numeri sequenza e acknowledgment (3)

**Esempi**

A ha ricevuto da B i segmenti da 0 a 999 byte e da 1000 a 1999 byte, per cui il numero di acknowledgment nel segmento da A a B → 2000

A ha ricevuto da B i segmenti da 0 a 999 byte e da 2000 a 2999 byte, per cui il numero di acknowledgment nel segmento da A a B → 1000

# Gestione ack

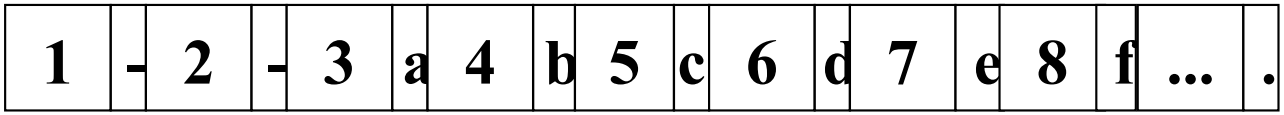
- Nella direzione da host1 a host2 viaggiano i segmenti dati inviati da host1 a host2 e i segmenti di ack inviati da host1 a host2 (in risposta ai segmenti dati inviati da host2 a host1)
- Nella direzione da host2 a host1 viaggiano i segmenti dati inviati da host2 a host1 e i segmenti di ack inviati da host2 a host1 (in risposta ai segmenti dati inviati da host1 a host2);
- Il campo ***code bit*** serve a distinguere fra i due tipi di segmenti (dati e ack) che viaggiano nella stessa direzione

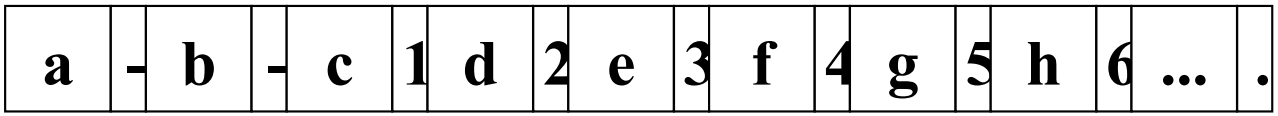
# Gestione ack migliorata: *piggybacking*

- Normalmente il TCP non invia un ack istantaneamente, ma ritarda l'invio sperando di avere dati da spedire insieme all'ack
- ***Piggybacking*** → "portare a cavallo sulle spalle"

- Per esempio, nella direzione da host2 a host1 si fanno viaggiare nello stesso segmento:
  - sia i dati che l'host2 deve inviare a host1
  - sia gli ack che host2 deve inviare a host1 in risposta ai segmenti dati inviati da host1 a host2

# Piggybacking




**Si sfrutta il flusso inverso opposto per portare gli ack dei pacchetti ricevuti**

# Instaurare e chiudere una connessione TCP

# Instaurazione di una connessione

- Nel TCP il mittente ed il destinatario, prima di iniziare il trasferimento dei segmenti contenenti i dati, instaurano la connessione
- **Modello client/server**
  - ***client***: inizia la connessione
  - ***server***: deve essere già attivo, in attesa, viene contattato dal client
- **Inizializzazione delle variabili del TCP**
  - **numeri di sequenza dei segmenti**
  - **informazioni necessarie per la gestione del buffer di trasmissione e ricezione**

# Instaurazione di una connessione (2)

- Quando un client richiede una connessione, invia un segmento TCP speciale, detto **"SYN" segment** (**SYN** sta per **syn**chronize) al server
- Il **client** deve conoscere a chi spedire la richiesta, per cui nell'header del segmento deve specificare:
  - La porta del server? **SI**
  - L'indirizzo IP? **NO**
- Per accettare la connessione, il **server** deve essere già in attesa di ricevere connessioni

# Instaurazione di una connessione (3)

Il **segmento SYN del client** include:

- **Initial Sequence Number** (**ISN**) del client:
  - un numero di 32 bit
  - Il numero è scelto in modo casuale tra 0 e 2  -1
  - Se il numero iniziale è 2032 e ci sono da spedire 5000 byte, tutti i byte saranno numerati da 2032 a 7031
  - La numerazione dei byte in una direzione è indipendente dalla numerazione dei byte nell'altra direzione
  - Nell'intestazione di ogni segmento è riportato solo il numero di sequenza del primo byte dei dati contenuto nel segmento. **Gli altri byte del segmento sono numerati di conseguenza**

# Instaurazione di una connessione (3)

Il **segmento SYN del client** include anche:

- **Maximum Receive Window** (**MRW**) del client: il massimo numero di byte che il client è in grado di ricevere nel suo buffer – *necessario per la regolazione del flusso dello stream di byte*
- **Maximum Segment Size** (**MSS**): la massima dimensione del segmento (informazione non sempre inviata)
- **NON HA *payload* (dati del messaggio), ma solo il TCP header!**

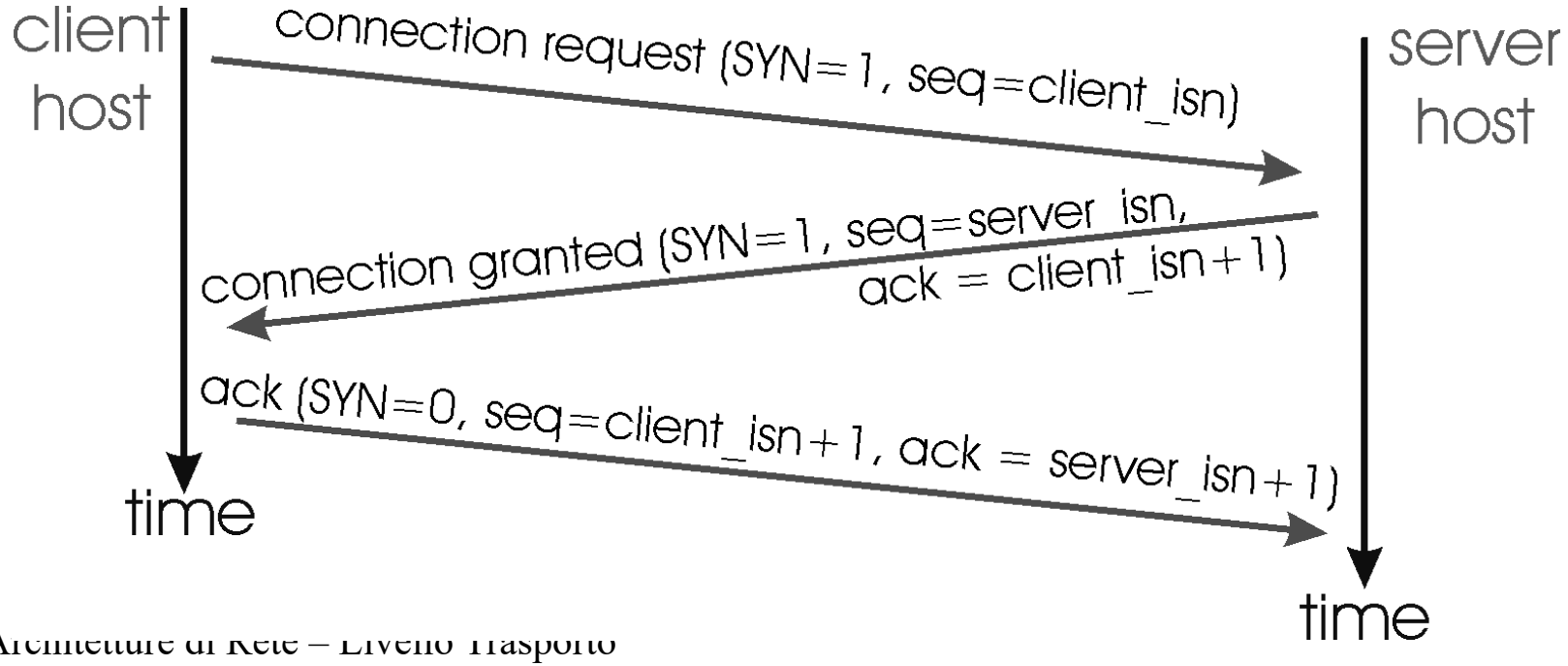
# Instaurazione di una connessione (4)

Il **segmento SYN del server** include:

- **Initial Sequence Number** (**ISN**) del server: un numero pseudo-casuale
- **ACK** del server: **client_ISN+1**
- **Maximum Receive Window** (**MRW**) del server
- **Maximum Segment Size** (**MSS**): la massima dimensione del segmento (informazione non sempre inviata)
- **NON HA *payload* (dati del messaggio), ma solo il TCP header**

# Instaurazione di una connessione
## (*Three-way handshaking*)

- Il client invia al server un segmento di controllo con SYN=1, e specifica nello stesso segmento il proprio numero iniziale di sequenza (*client_isn*)
- Il server riceve il segmento del client con SYN=1
- Se accetta, il server invia un segmento di controllo con SYN=1, ACK=*client_isn*+1, ed il proprio numero iniziale di sequenza (*server_isn*)
- Il client segnala la definitiva apertura della connessione inviando un segmento di controllo con SYN=0, ACK=*server_isn*+1, e numero di sequenza *client_isn* +1



# Three-way handshacking

1. **Client**: “Hello server, voglio parlare con te, e comincerò con il byte che indicherò con il numero **X**”

2. **Server**: “OK, sono disponibile a parlare. Il mio primo byte sarà indicato con il numero **Y** e so che il tuo prossimo byte sarà indicato con **X+1**”

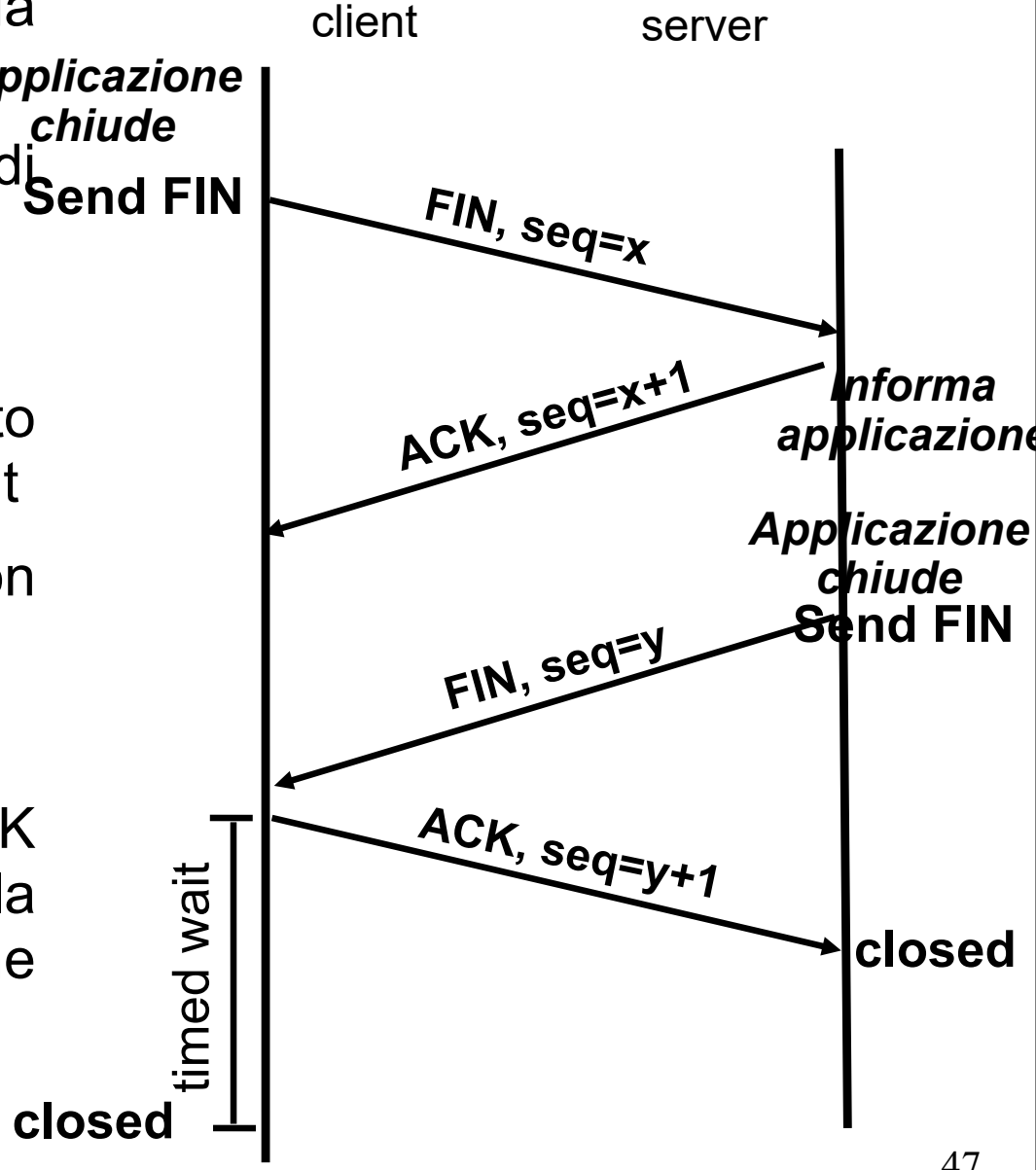
3. **Client**: “D’accordo, so che il prossimo byte che mi invierai sarà indicato con il numero **Y+1**”

# Chiusura (*polite*) della connessione (esempio lato client)

Essendo full-duplex,va chiusa da entrambe leparti

- Il client invia un segmento di controllo con bit FIN=1 al server
- Il server riceve FIN, invia ACK
- Il server chiude la connessione lato client-server ed invia FIN=1 al client
- Il client riceve il segmento con FIN=1 ed invia ACK
- Il server riceve ACK
- Il client attende il timeout dell'ACK inviato; allo scadere anche la connessione lato server-client viene chiusa

Perché tante "fasi" nella chiusura?



# Chiusura (*polite*) della connessione (2)

- Dopo che la connessione TCP è stata chiusa (ovvero è stato inviato l'ultimo ACK dal client), ci potrebbe essere ancora qualcosa da fare. Per esempio:
  - **Se l'ultimo ACK si perde?**

    → L'ultimo segmento FIN verrà inviato nuovamente, e dovrà essere notificato con un ACK
  - **Se segmenti persi o duplicati dovessero raggungere la destinazione dopo un lungo ritardo?**

- Quindi, il client TCP attende per un tempo **TIME_WAIT** (es., 30 secondi) prima di chiudere definitivamente la connessione per poter gestire queste situazioni anomale

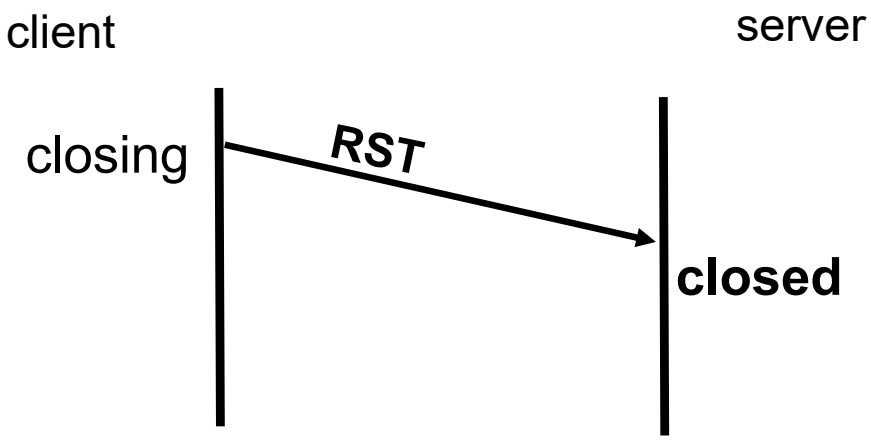
# Maximum Segment Lifetime

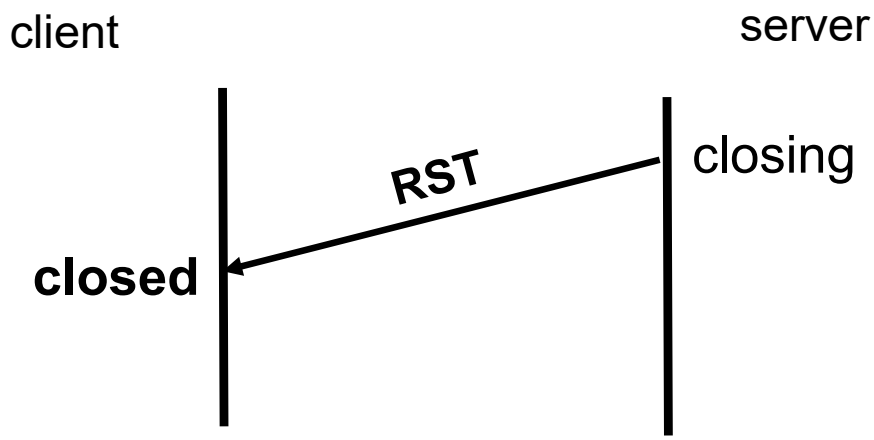
- Il ***Maximum Segment Lifetime*** (MSL=2 min in RFC 793) indica il massimo tempo per il quale un segmento TCP può sopravvivere nella rete prima di essere scartato. Attendere 2MSL nello stato TIME WAIT garantisce che tutti i segmenti relativi alla connessione siano spariti dalla rete
- Nello stato TIME_WAIT si impedisce che nel client possa aprirsi una connessione con lo stesso indirizzo di quella appena chiusa (porte+IP)
- Il vincolo più rigido in molte implementazioni è che non sia riusato il numero di porta locale
- Per il server questo non avviene (la porta essendo pubblicata, deve rimanere attiva)

# Chiusura (*reset*) della connessione

In condizioni normali, la connessione viene chiusa in modo “polite” tramite lo scambio di segmenti di controllo FIN e ACK, visto in precedenza

- Talvolta, si verificano condizioni che portano ad interrompere la connessione in modo “brusco”, o dal lato server (tipicamente, per errori o sovraccarico) o dal lato client (tipicamente, in seguito a azione dell’utente)
- TCP fornisce un meccanismo per la chiusura rapida: **reset**
- L’host che decide il reset pone il campo del segmento RST=1
- L’altro nodo chiude immediatamente la connessione
- Vengono rilasciate tutte le risorse utilizzate dalla connessione



# Affidabilità del protocollo TCP

# Meccanismi per l'affidabilità

1. **Acknowledgment**
2. **Time-out**
3. **Ritrasmissione**

   **Meccanismi gestiti dal livello TCP in modo del tutto trasparente rispetto al processo applicativo**

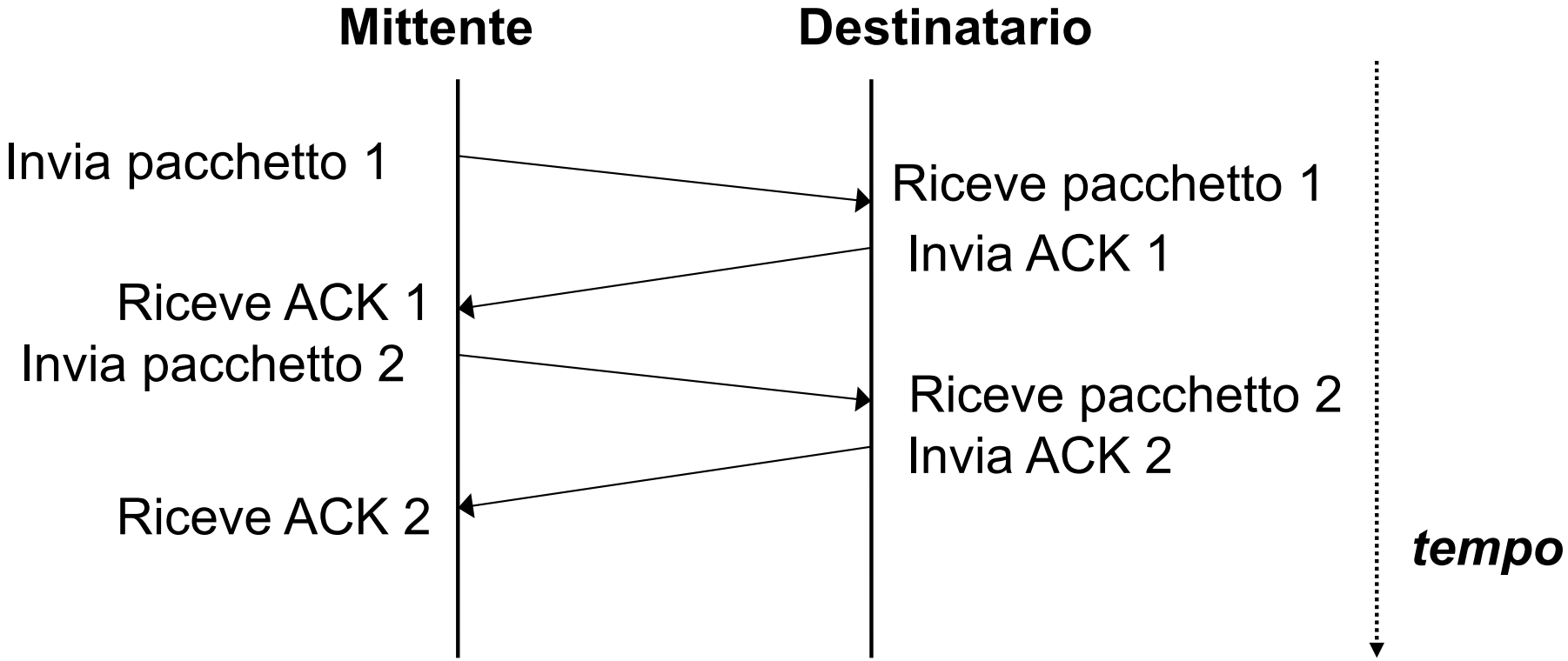
Princìpi:

- Ogni trasmissione andata a buon fine viene notificata (**acknowledged**) dall'host ricevente
- Se l'host mittente non riceve un acknowledgement entro il ***time-out***, il mittente ritrasmette i dati

# Affidabilità del protocollo TCP

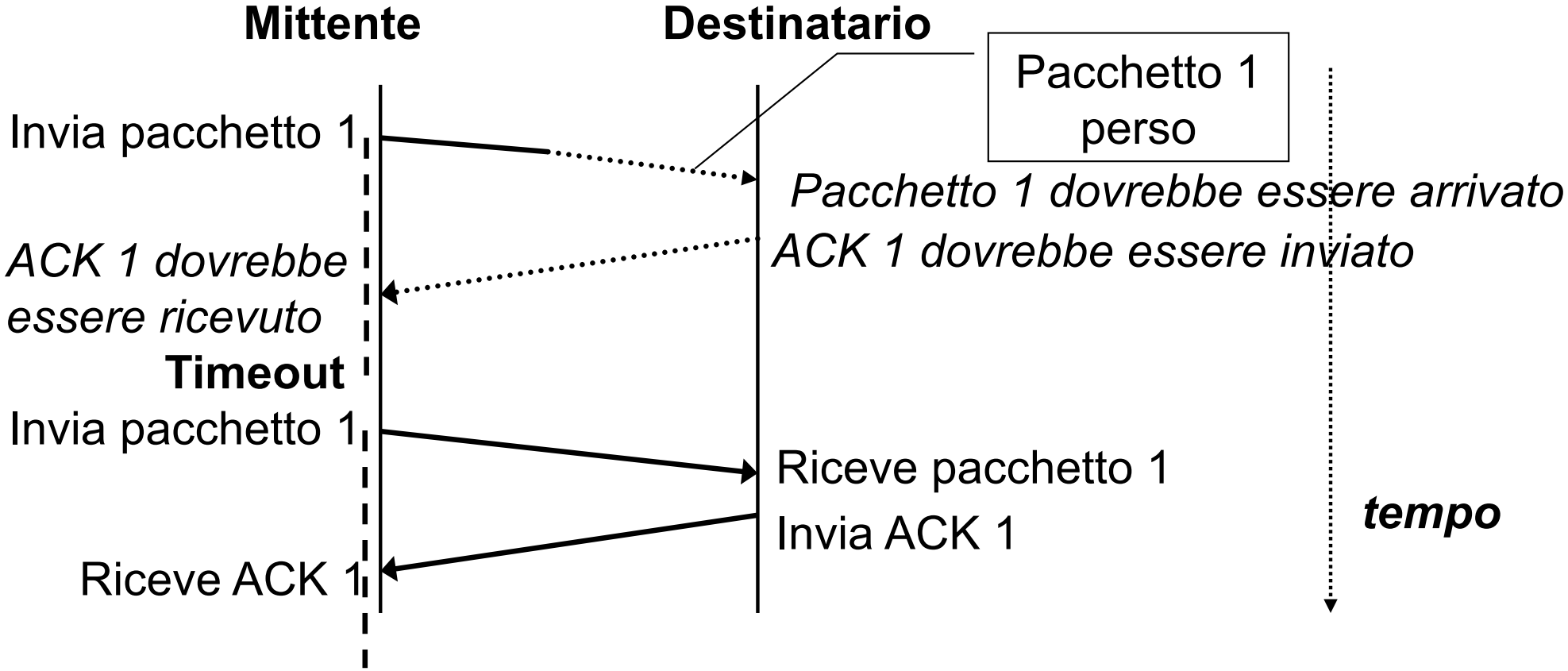
**Affidabilità**: uso della tecnica di acknowledgement positivo con ritrasmissione → Il destinatario, quando riceve i dati, invia un acknowledgement (**ACK**) al mittente, che attende di ricevere un ACK prima di inviare il segmento successivo.

Esempio (Stop&Wait)



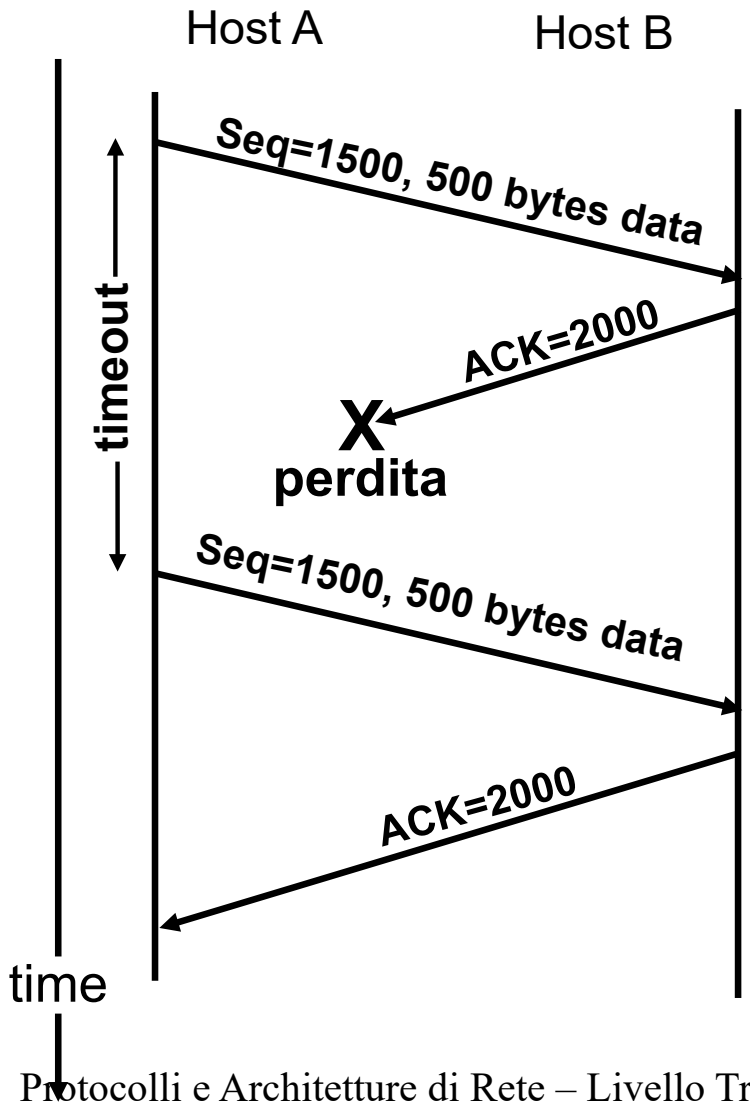
# Scenari di ritrasmissione

**Affidabilità**: uso della tecnica di **acknowledgement positivo con timeout e ritrasmissione** $\rightarrow$ **se il mittente non ha ricevuto ACK di un segmento dopo un certo periodo (*timeout*), ritrasmette il segmento**

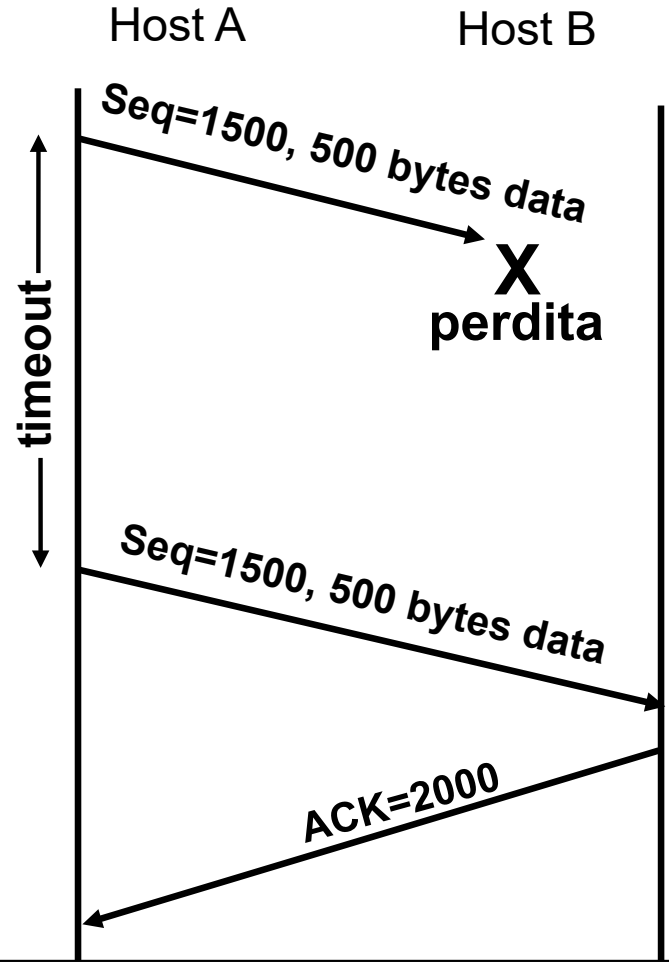


# Scenari di ritrasmissione (protocollo *stop-and-wait*)

## Ritrasmissione causata dalla perdita dell'ACK




## Ritrasmissione causata dalla perdita del segmento



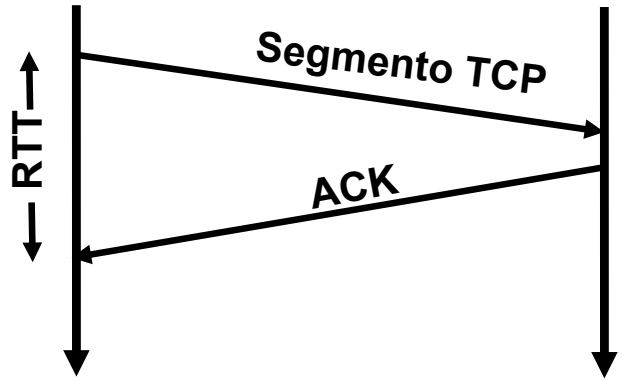

**In entrambi i casi, l'ack=2000 non arriva. Nel primo caso, c'è una duplicazione.**

# Come stimare il “time-out”?

# Round Trip Time e Timeout

**Come stabilire il valore del timeout in trasmissioni TCP?**

- *Timeout troppo breve*: si effettuano ritrasmissioni non necessarie
- *Timeout troppo lungo*: reazione lenta alla perdita di segmenti

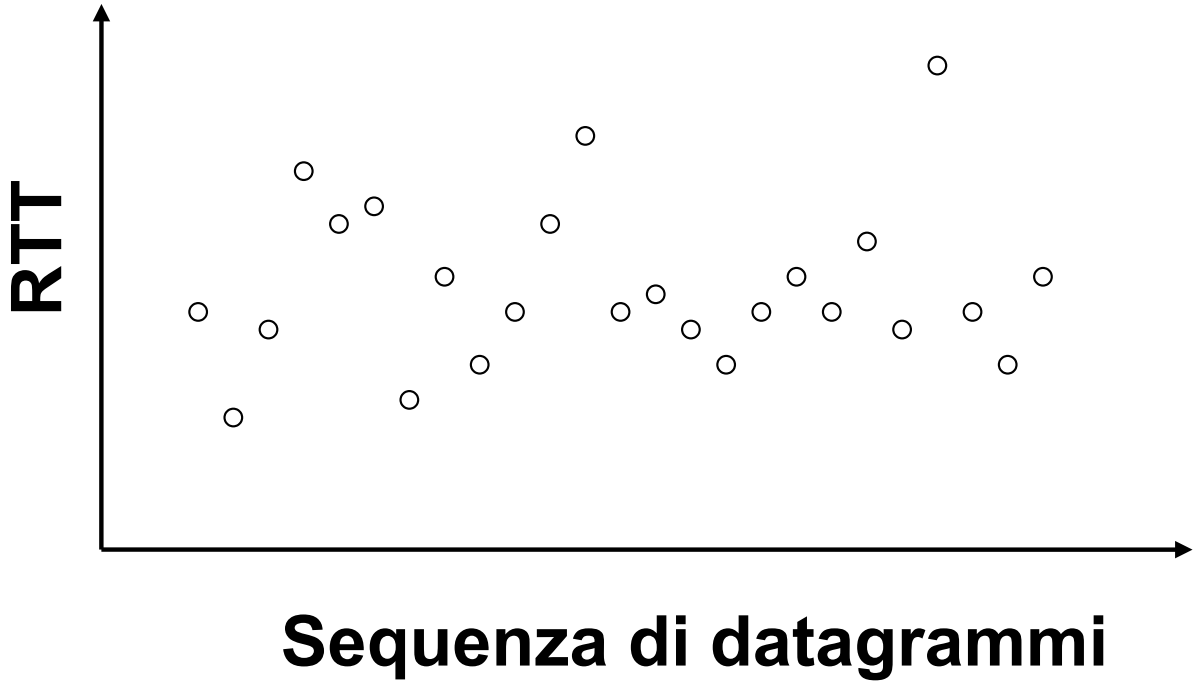



UNICA CERTEZZA: Il timeout deve essere maggiore del **Round Trip Time** (**RTT**). Ma di quanto?

# RTT varia continuamente

**Motivazioni delle differenze del RTT:**

- **fluttuazioni delle condizioni di traffico della rete**
- **possibili cambiamenti di router nel percorso tra mittente e destinatario**



# Scelta del Timeout (*old*)

Inizialmente si sceglieva:

$$\textbf{Timeout} = \beta * \textbf{RTT}_{\textbf{medio}}$$

dove la raccomandazione era: $\beta = 2$

**Adesso, si utilizzano stime molto più sofisticate**

# Scelta del Timeout

**SampleRTT**: misura del tempo trascorso dalla trasmissione del segmento alla ricezione del suo ACK

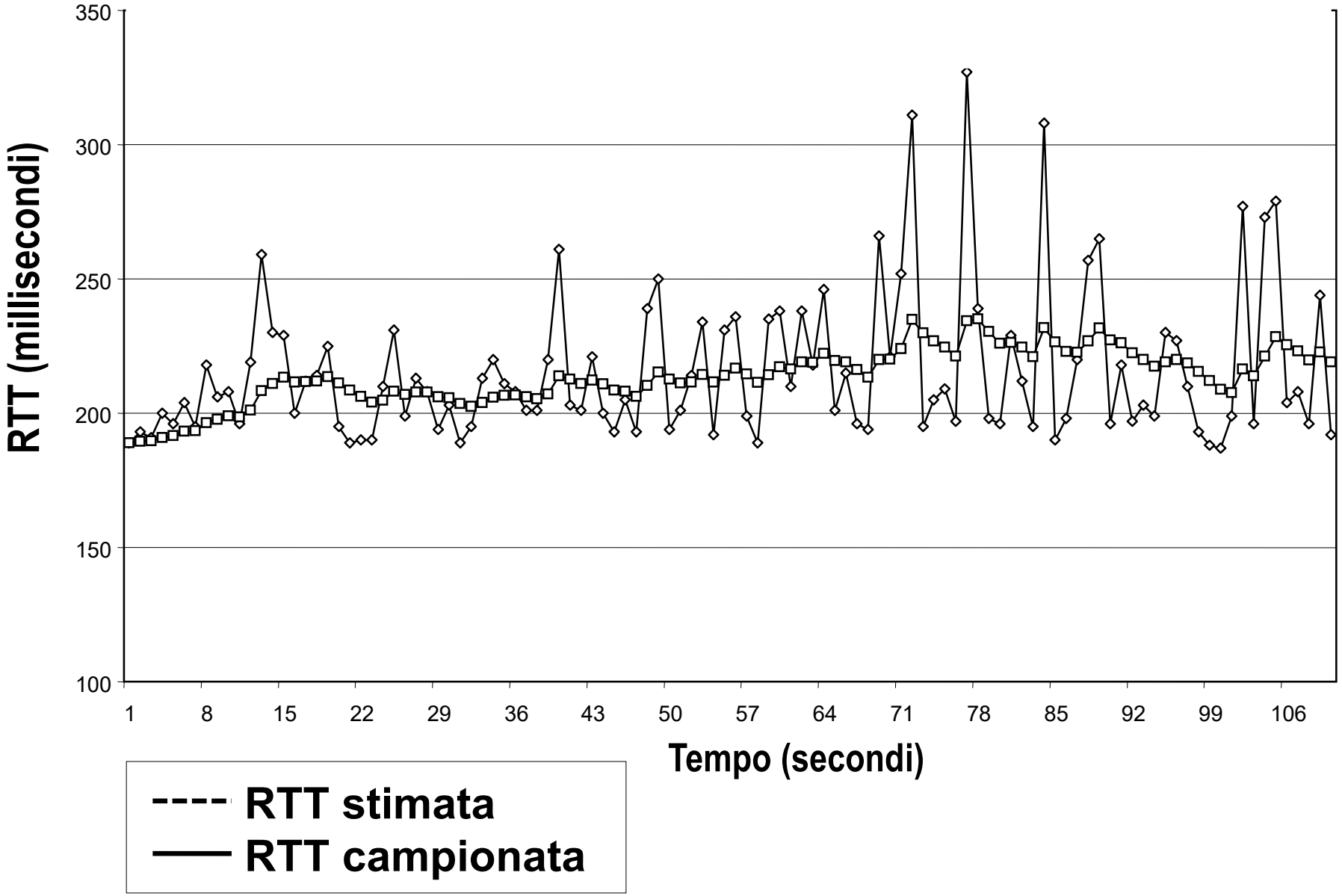
- Ignora ritrasmissioni, segmenti con ack cumulativi
- SampleRTT varia dinamicamente ➔ **si usa una media pesata**

- **EstimatedRTT**: media pesata per stimare RTT al tempo ***t***

  **EstimatedRTT(t) = (1-x)*EstimatedRTT(t-1) + x*SampleRTT(t)**

  - **Exponential Weighted Moving Average** (**EWMA**)
  - L'influenza dei campioni passati diminuisce in modo esponenziale
  - Il valore di ***x*** è compreso tra 0 e 1. Inizialmente, si sceglie tipicamente **x=1/(n+1)** dove ***n*** è il numero di campioni di RTT usati per il calcolo

# Esempio della stima RTT



# Scelta del Timeout (2)

**Scelta del timeout**:

**Valore del RTT stimato più un margine di errore**

$$\text{Timeout}(t) = \text{EstimatedRTT}(t) + 4*\text{Deviation}(t)$$

dove:

$$\text{Deviation}(t) = (1-x)*\text{Deviation}(t-1) + x*\text{abs}[\text{SampleRTT}(t)-\text{EstimatedRTT}(t)]$$

# Prestazioni del protocollo TCP

# Prestazioni

- **L'esempio precedente mostrava un protocollo stop-and-wait: attendiamo l'ack prima di inviare il segmento successivo**
- **Il meccanismo di trasporto *stop-and-wait* è estremamente affidabile e semplice da implementare**
- **Tuttavia, utilizza le risorse di rete e degli host in modo totalmente inefficiente**

***MOTIVAZIONI →***

# **Prestazioni** (*alcune definizioni*)

- **Round Trip Time** (**RTT**): tempo impiegato dal pacchetto per andare dal mittente al destinatario e ritorno
- **Tempo di propagazione**: **RTT/2**
- **Utilizzazione**: percentuale di utilizzo di una risorsa in un intervallo di tempo

  **Rate = transmission rate** (***throughput***) → capacità trasmissiva della rete, espressa in: **bit/sec** (**bps**), **Kbit/sec** (**Kbps**), **Mbit/sec** (**Mbps**), **Gbit/sec** (**Gbps**)

  **L(pkt) = lunghezza pacchetto** (espressa in ***byte*** o ***bit***, e loro multipli)

# **Prestazioni** (*alcune definizioni*)

- **Tempo di trasmissione pacchetto**

  ➔ **L(pkt) / Rate**

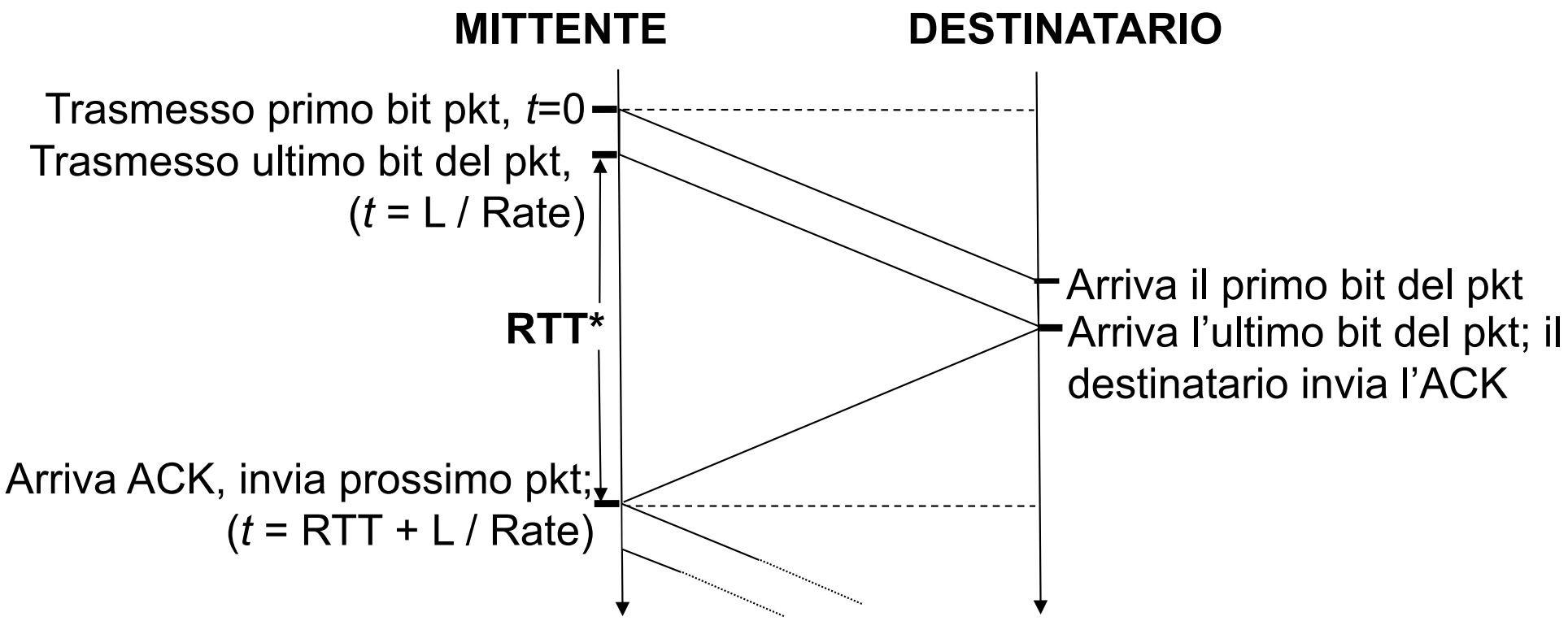
  [dimensionalmente: (**bit**) / (**bit/sec**) = **sec** ]

- **Tempo di trasferimento pacchetto** (*vista mittente*)

➔ [propagazione + trasmissione] + [$\text{propagazione}_{\textbf{ack}}$]

= [$\text{RTT}_{\textbf{pkt}}$/2 + L(pkt) / Rate] + [$\text{RTT}_{\textbf{ack}}$/2]

# Prestazioni protocollo *stop-and-wait*




**RTT*= Round Trip Time ideale** (se il destinatario non è rallentato da altri processi)

# Es. prestazioni protocollo *stop-and-wait*

## Esempio

- Canale fisico di capacità 20 Mbps (= 20^6 bit/sec)
- Ritardo di propagazione (RTT/2) = 15 msec
- Pacchetto da 1KB (=8 Kb)

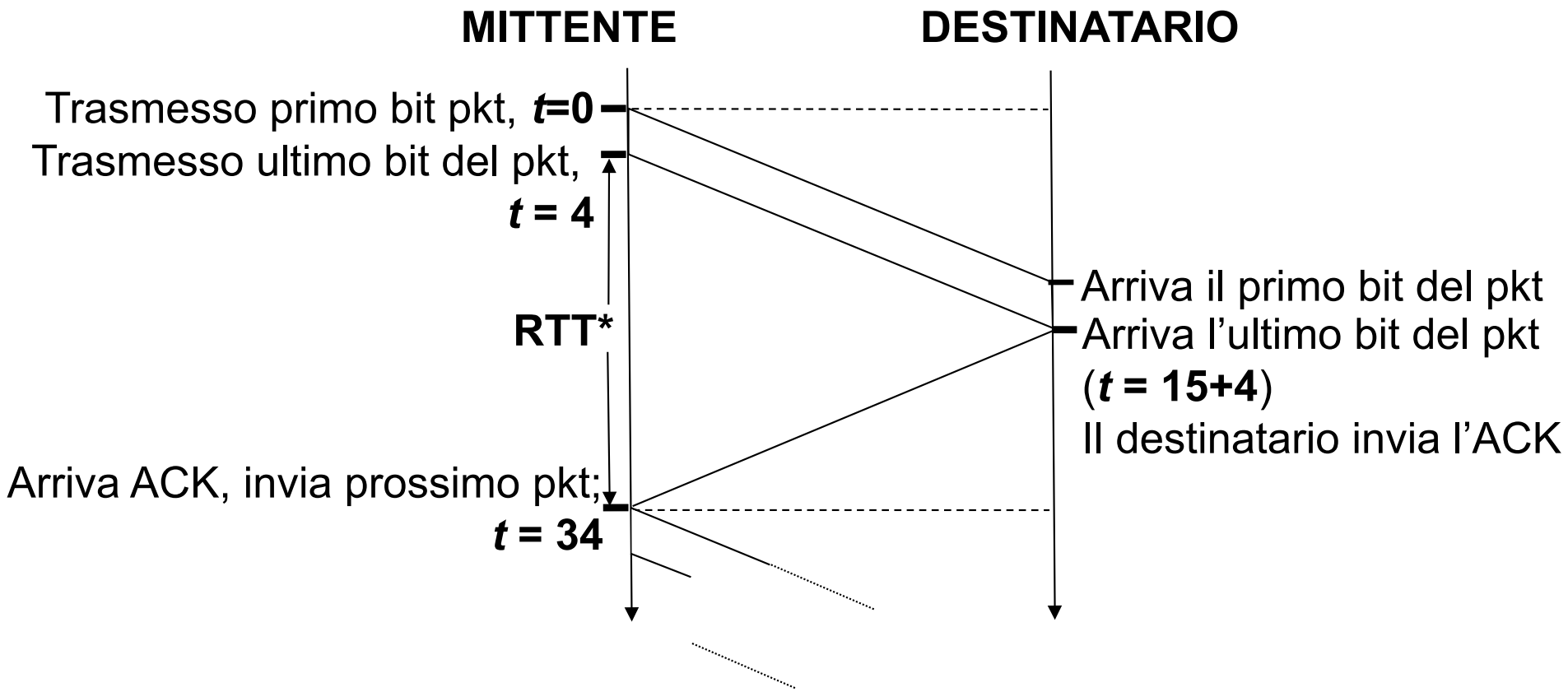
Rate = *transmission rate*
L(pkt) = lunghezza pacchetto

**Calcolare il tempo di trasmissione pacchetto:**

$$T_{trasm/pkt} = \frac{L(pkt)}{Rate} = \frac{8Kb}{20\text{^}6 \text{ bit/sec}} = \frac{8 \cdot 10\text{^}3 \text{ bit}}{20\text{^}6 \text{ bit/sec}} =$$

$$= \frac{4 \text{ bit}}{10\text{^}3 \text{ bit/sec}} = 0.004 \text{ sec} = 4 \text{ msec}$$

# Es. prestazioni protocollo *stop-and-wait* (2)
## Uso limitato delle risorse di rete

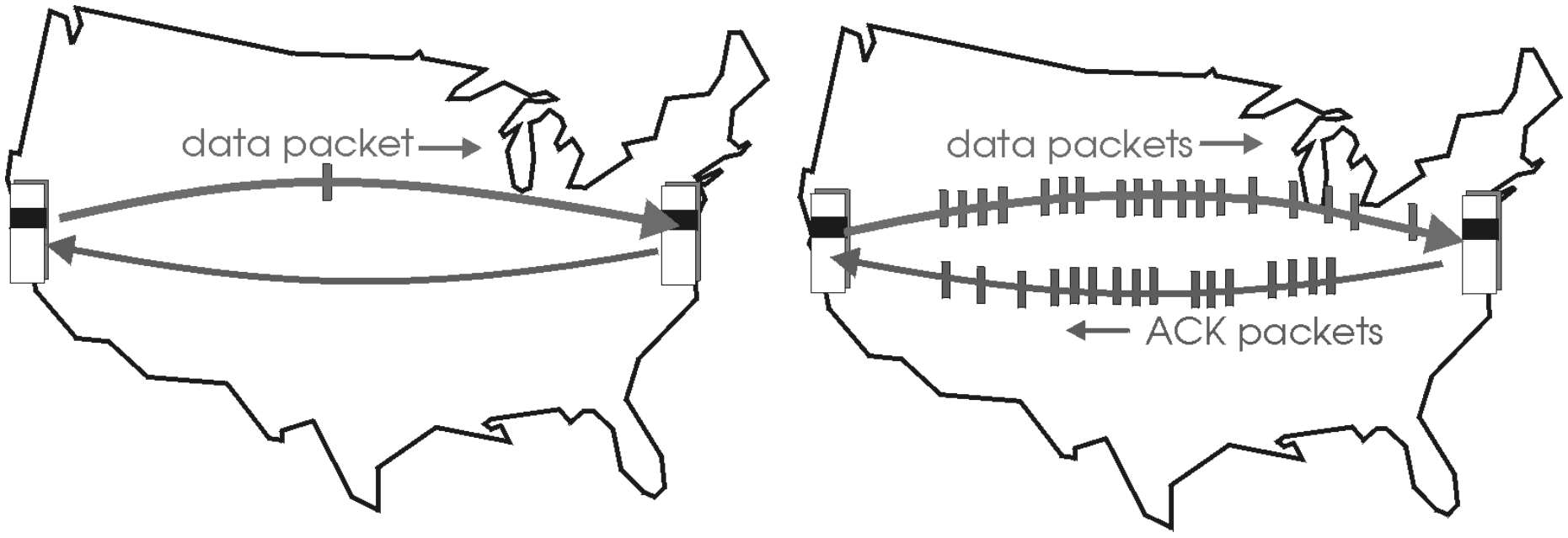



$$\text{Utilizzazione}_{(\textit{mittente})} = \frac{\text{Tempo trasm.}}{\text{Tempo totale}} = \frac{T}{RTT + T} = \frac{L/R}{RTT + L/R} = \frac{4}{30 + 4} = \mathbf{0.12}$$

**In pratica, si ha un canale di comunicazione utilizzato a circa il 10% della sua capacità**

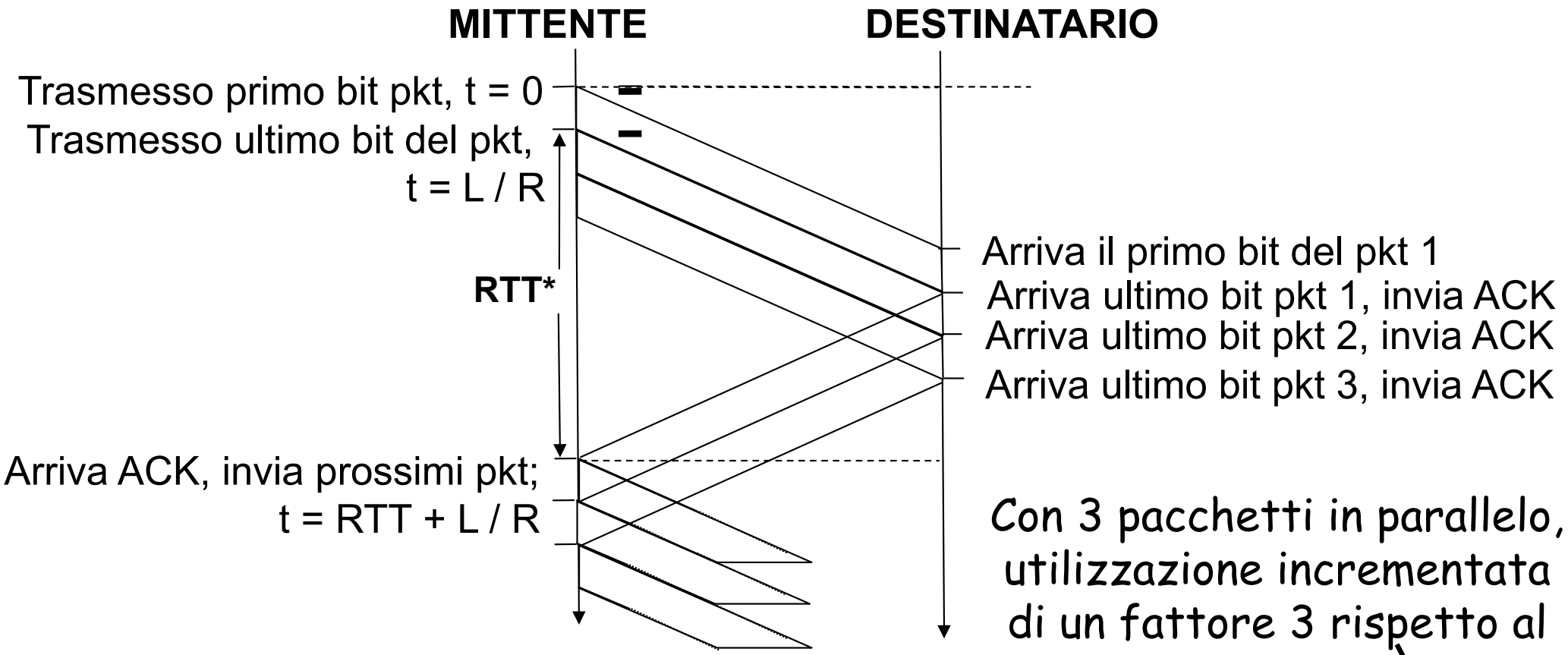
# Come migliorare le prestazioni?

**Utilizzando una tecnica di *pipelining* → il mittente invia un numero multiplo di segmenti prima di ricevere un ACK**




**Dal protocollo stop-and-wait ad un protocollo pipelined**

# Incremento delle prestazioni (es. per 3)




$$\text{Utilizzazione (mittente)} = \frac{\text{Tempo trasm.}}{\text{Tempo totale}} = \frac{T}{RTT + T} = \frac{3*L/R}{RTT + L/R} = \frac{12}{30 + 4} = 0.35$$

# Requisiti per l'implementazione di un protocollo di pipelining

1. Necessità di un **buffer lato mittente**
   - Per mantenere i dati inviati e di cui non ha ancora ricevuto l'ack
2. Necessità di un **buffer lato destinatario**
   - Per mantenere le sequenze di dati dove non tutti i dati sono arrivati o sono arrivati correttamente

3. Necessità di una **"finestra a scorrimento"** (***sliding window***) che denota il numero massimo di dati che il mittente può inviare senza aver ricevuto un ACK dal destinatario