Università di Roma Tor Vergata
Corso di Laurea triennale in Informatica

Sistemi operativi e reti

A.A. 2016-17

Pietro Frasca

Parte II: Reti di calcolatori

Lezione 12 (36)

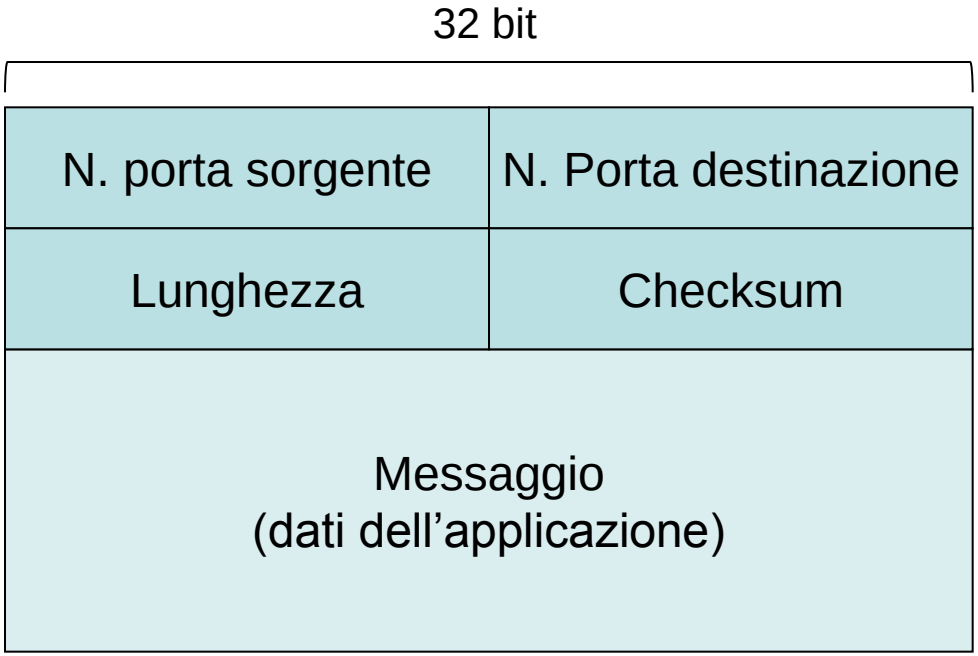
Martedì 18-04-2017

UDP

- L'UDP è un protocollo di trasporto che svolge solo le funzioni di multiplexing/demultiplexing e una semplice verifica degli errori.

Struttura del segmento UDP

- L'intestazione dell'UDP è di solo 8 byte, ed è formata di quattro campi, ciascuno di due byte:
  - Numero di **porta sorgente**
  - Numero di **porta destinazione**
  - Il campo **lunghezza** specifica la lunghezza del segmento UDP (intestazione più dati) in byte.
  - La **checksum** è usata per controllare se si sono verificati errori nel segmento nella trasmissione.
  - Il campo **dati** contiene i dati dell'applicazione (messaggio).



Struttura del segmento UDP.

Checksum di UDP

- La checksum di UDP è usata per determinare se i bit nel segmento UDP hanno subito errori nella trasmissione.
- L'UDP nel lato mittente calcola il **complemento a 1** della **somma di tutte le parole a 16 bit** del segmento e di alcuni campi dell'intestazione IP, e l'eventuale riporto finale, del bit più significativo, viene sommato al bit meno significativo (primo bit). Il risultato è inserito nel **campo checksum** del segmento.
- Per esempio, supponiamo di avere le seguenti tre parole di 16 bit:

  0110011001100110
  0101010101010101
  1000111100001111

- La somma delle prime due di queste parole di 16 bit è

```
  0110011001100110
  0101010101010101
------------------
  1011101110111011
```

Aggiungendo la terza parola la somma dà

```
  1011101110111011
  1000111100001111
------------------
  0100101011001010
                 1
------------------
  0100101011001011
```

Complemento a 1 →

checksum

1011010100110100

- Il complemento a 1 si ottiene invertendo tutti gli **0** in **1** e viceversa gli **1** in **0**. Quindi il complemento a 1 della somma 0100101011001011 è 1011010100110100, che diventa la checksum.
- Al destinatario arrivano tutte le parole a 16 bit, inclusa la checksum. Se nel pacchetto arriva senza errori, la somma calcolata al ricevente deve essere **1111111111111111**. Se uno o più bit vale zero, significa che il pacchetto contiene degli errori.

Trasporto orientato alla connessione: TCP

- Per fornire un trasporto affidabile dei dati, il TCP utilizza molti algoritmi relativi alla ritrasmissione di segmenti, riscontri cumulativi, rilevazione degli errori, timer e campi di intestazione per i numeri di sequenza e numeri di riscontro.

La connessione TCP

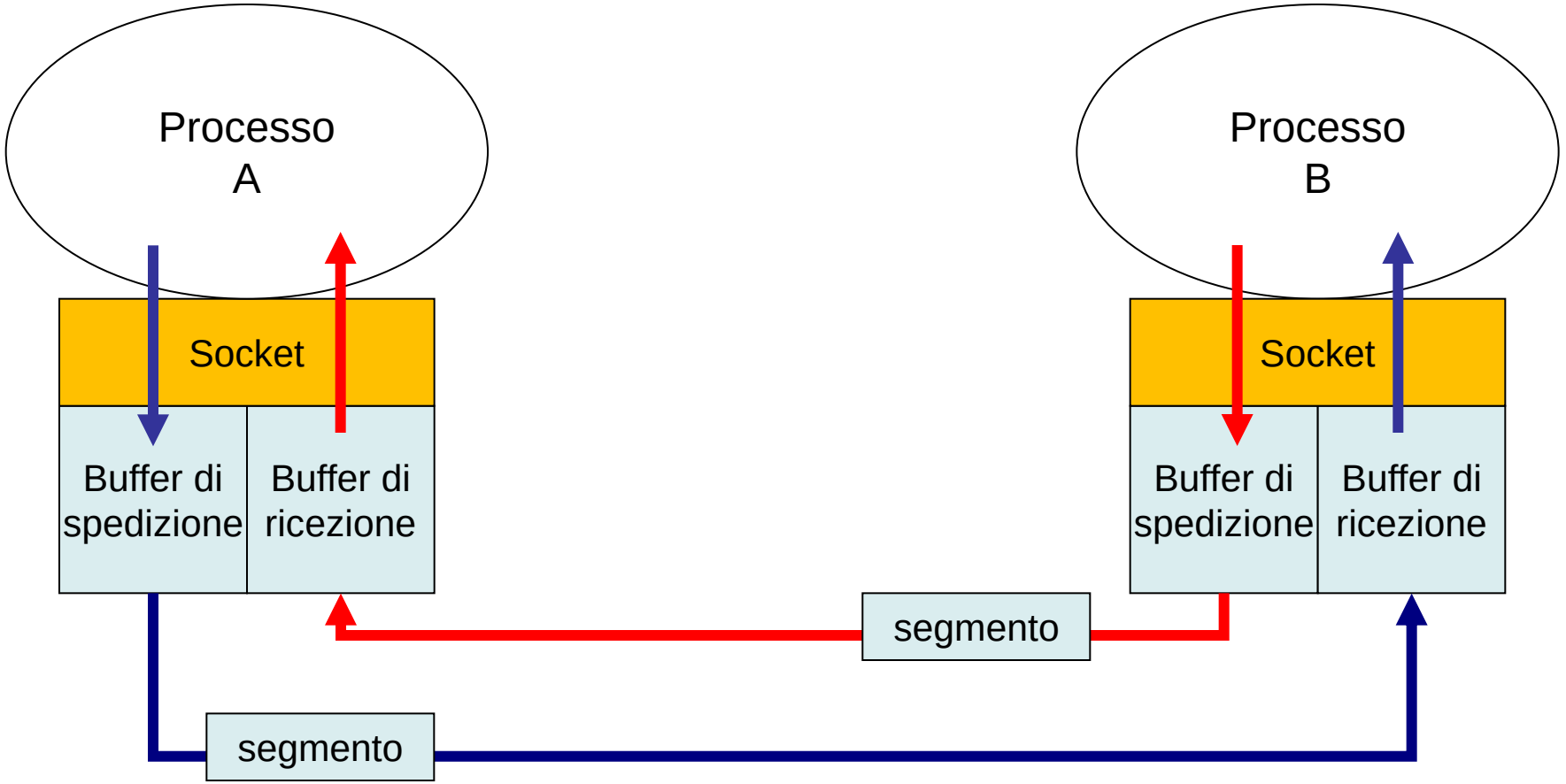
- Col TCP prima che due processi possano trasmettere dati, devono inizialmente eseguire una procedura di handshake. In questa fase, entrambi i lati TCP inizializzeranno diverse "**variabili di stato**" associate alla connessione stessa.
- Una connessione TCP consente un trasferimento dei dati **full duplex** cioè permette di inviare i dati contemporaneamente nelle due direzioni.
- Una connessione TCP è di tipo **unicast punto-punto**, cioè tra un singolo mittente e un singolo destinatario.

- In Java la connessione TCP si realizza creando nel lato client un oggetto della classe Socket:

  Socket clientSocket = new Socket("hostname", numeroPorta);

- Poiché tre segmenti speciali sono scambiati tra i due host, spesso questa procedura è chiamata ***handshake a tre vie***.
- In questa fase,
  - **il client invia uno speciale segmento TCP**;
  - **il server risponde con un secondo segmento speciale TCP**
  - **alla fine il client risponde ancora con un terzo segmento speciale.**
- I primi due segmenti non portano dati dell'applicazione (***carico utile, payload***), mentre il terzo segmento può trasportare dati dell'applicazione.
- Una volta stabilita la connessione TCP, i due processi client e server si possono scambiare dati.

- Consideriamo la trasmissione di dati dal client al server.
- Durante la fase di handshake, il TCP, sia nel lato client che nel lato server, crea un **buffer di spedizione** (send buffer) e un **buffer di ricezione** (receive buffer).
- Il processo client passa uno stream di dati attraverso il socket. Una volta passati al socket, i dati sono gestiti dal TCP del client. Il TCP pone questi dati nel buffer di spedizione. Di tanto in tanto il TCP invierà blocchi di dati prelevandoli dal buffer di spedizione.
- La dimensione massima di dati che può trasportare un segmento è determinato dalla variabile **MSS** (***Maximum Segment Size* - dimensione massima del segmento**)



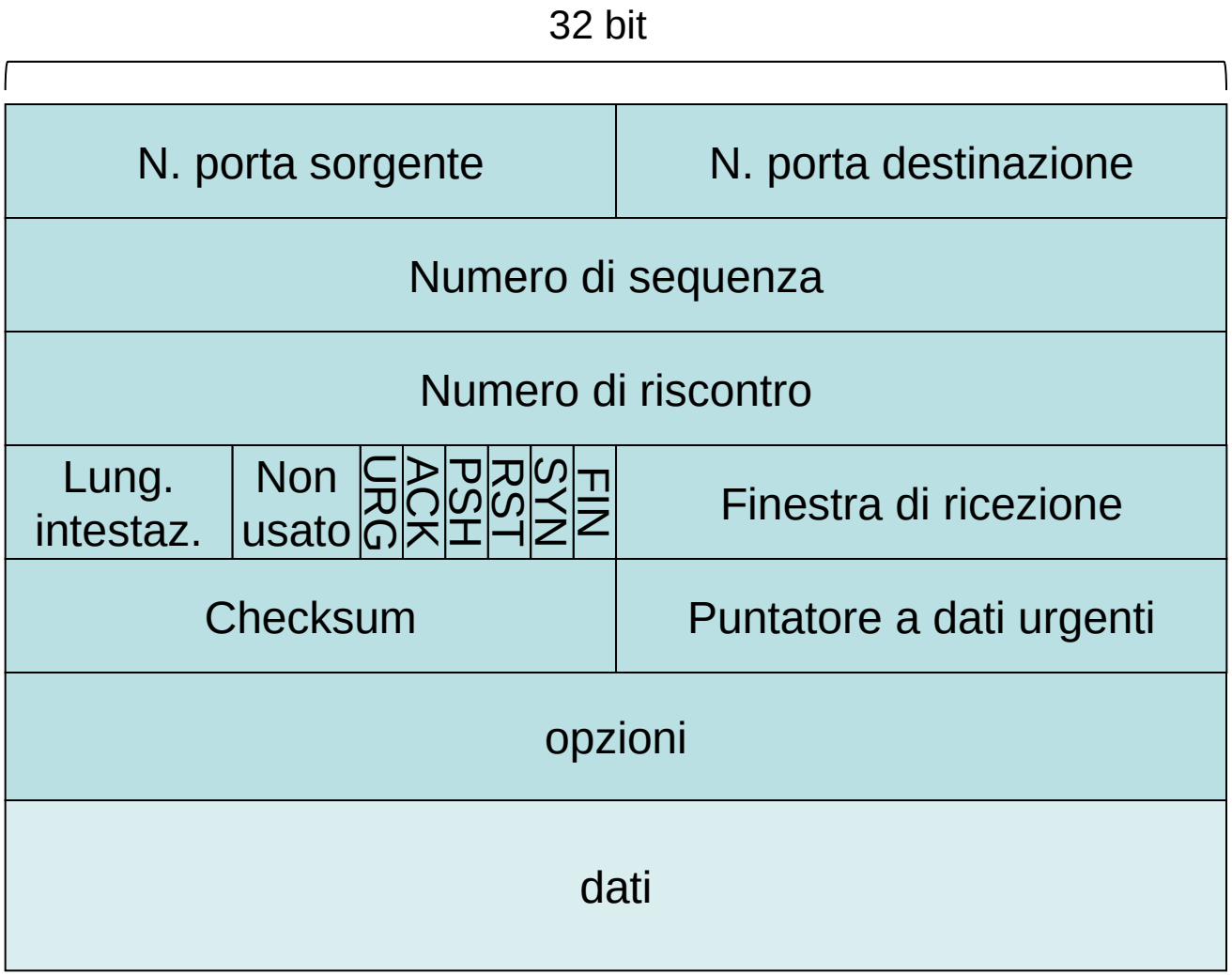
Buffer di invio e ricezione del TCP.

- Il valore di MSS dipende dall'implementazione del TCP e spesso può essere configurato; valori tipici sono, 512, 536 e 1460 byte. Queste dimensioni di segmenti sono scelte soprattutto per evitare la **frammentazione IP**, che esamineremo in seguito.
- Più precisamente, l'MSS è la **massima quantità dei dati dell'applicazione** presente nel segmento, non la massima dimensione del segmento TCP.
- Quando il TCP invia un file di grandi dimensioni, come per esempio un file multimediale, lo **suddivide in parti** di dimensione **MSS** byte (ad eccezione dell'ultima parte, che generalmente ha dimensione inferiore a MSS).
- Tuttavia, le applicazioni interattive generalmente trasmettono parti di dati più piccole di MSS byte. Per esempio, Telnet (o ssh), può avere il campo dati nel segmento di un solo byte. Poiché **l'intestazione TCP tipica è di 20 byte**, la dimensione dei segmenti inviati mediante Telnet può essere di soli **21 byte**.

- Il TCP aggiunge ai dati dell'applicazione un'**intestazione TCP**, formando in tal modo i segmenti TCP.
- I segmenti sono passati allo strato di rete, dove sono incapsulati separatamente nei datagram IP dello strato di rete.
- i segmenti TCP, quando arrivano a destinazione, sono posti nel **buffer di ricezione** del lato ricevente, come mostrato nella figura precedente. L'applicazione legge il flusso di dati da questo buffer.
- Ciascun lato della connessione ha i suoi propri buffer di spedizione e di ricezione.

Struttura del segmento TCP

- La figura seguente mostra l'intestazione del segmento TCP.



- Come per UDP, i **numeri di porta sorgente** e di **destinazione**, sono usati per la multiplazione e demultiplazione.
- **numero di sequenza** e **numero di riscontro** sono usati rispettivamente da mittente e destinatario per implementare un servizio di trasferimento affidabile dei dati.
- **lunghezza intestazione** (4 bit) specifica la lunghezza dell'intestazione del TCP in parole di 32 bit. L'intestazione TCP può avere lunghezza variabile in dipendenza alla lunghezza del campo opzioni. Generalmente il campo opzioni non è usato, pertanto la lunghezza standard dell'intestazione TCP è di 20 byte.
- **finestra di ricezione** (16 bit) è usato per il **controllo del flusso di dati**. Indica il numero di byte che il destinatario è in grado di ricevere.
- Il campo **checksum** è simile a quello dell'UDP.
- **opzioni** è un campo facoltativo e di lunghezza variabile. Le opzioni più importanti sono negoziabili durante l'instaurazione della connessione, e sono:
  - dimensione massima dei segmenti da spedire (**MSS: Maximum Segment Size**), serve soprattutto per adattare la dimensione del segmento in modo che ogni segmento venga incluso in un datagram IP che non debba essere frammentato;

- Scelta dell'algoritmo di controllo del flusso come ad esempio *selective repeat* invece che *go-back-n.*

- **flag** (6 bit).
  - I bit **RST**, **SYN** e **FIN** sono usati per instaurare e chiudere la connessione.
  - Il bit **ACK** se settato a 1 indica che il valore presente nel campo **numero di riscontro** è valido.
  - I bit PSH e URG sono raramente usati. Il bit **PSH** quando è settato, indica che i dati in arrivo non devono essere bufferizzati ma dovrebbero essere passati immediatamente allo strato superiore. Il bit **URG** se settato a 1 indica che in questo segmento ci sono dati che il mittente ha contrassegnato come "urgenti".

Numeri di sequenza e numeri di riscontro

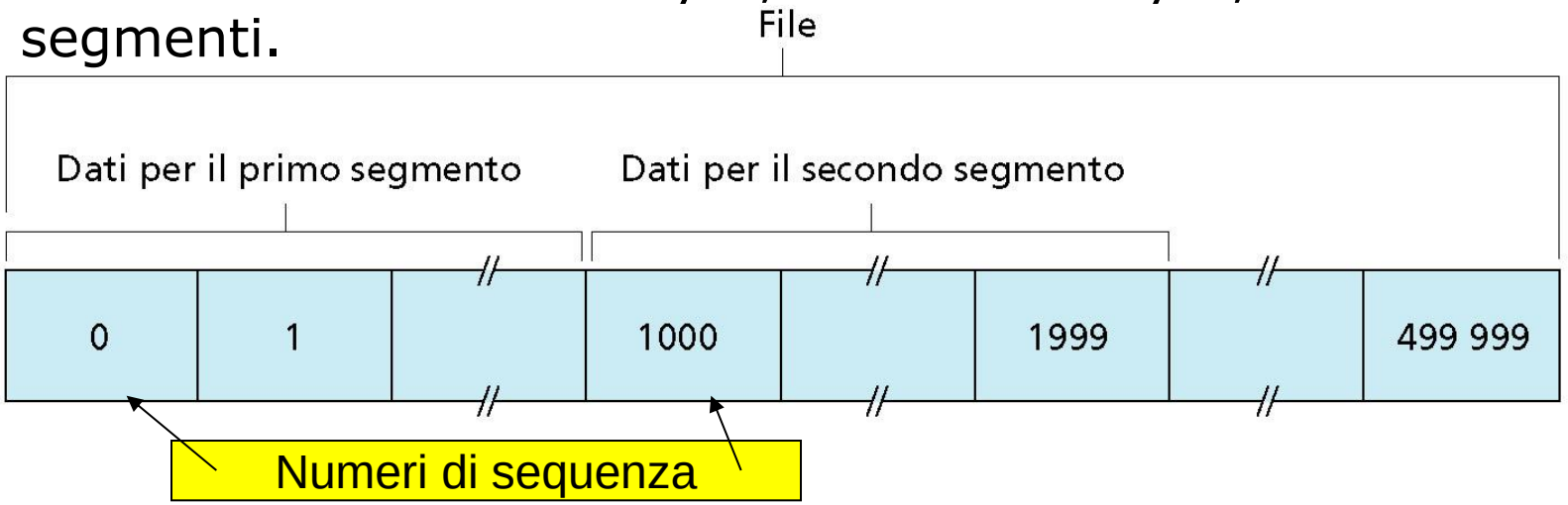
- Questi campi sono usati per il servizio di trasferimento affidabile dei dati del TCP.
- Il TCP considera i dati dell'applicazione come un flusso di byte ordinato.

Numeri di sequenza

- Il numero di sequenza di un segmento è il numero d'ordine del primo byte nel segmento all'interno del flusso.

Esempio

File dimFile = 500.000 byte; MSS=1000 byte; -> 500 segmenti.



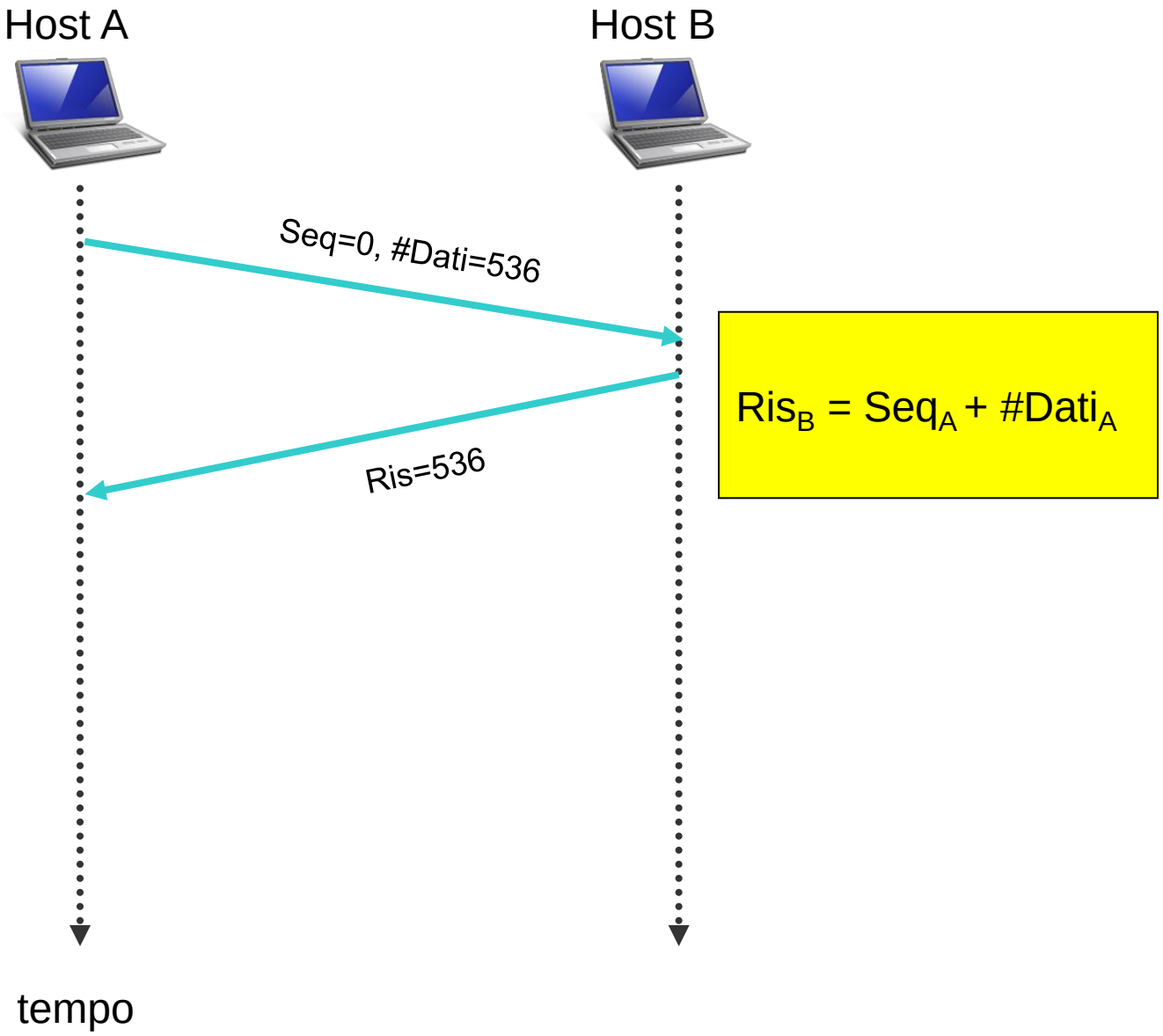
- Nella figura precedente il numero di sequenza iniziale è posto a zero. In realtà, entrambi i lati di una connessione TCP **generano casualmente un numero di sequenza iniziale,** che si scambiano durante la fase di handshake in modo che ciascun lato conosca il numero di sequenza iniziale dell'altro.

Numeri di riscontro

- Dato che il TCP è full-duplex, due host **A** e **B** che comunicano possono sia ricevere che trasmettere i dati nello stesso momento.
- Ogni segmento che l'host A invia all'host B, ha un numero di sequenza relativo ai dati che A invia a B. Il numero di riscontro (RIS) che l'host B inserisce nel suo segmento è pari a $\boldsymbol{RIS_B = SEQ_A + num\_dati_A}$

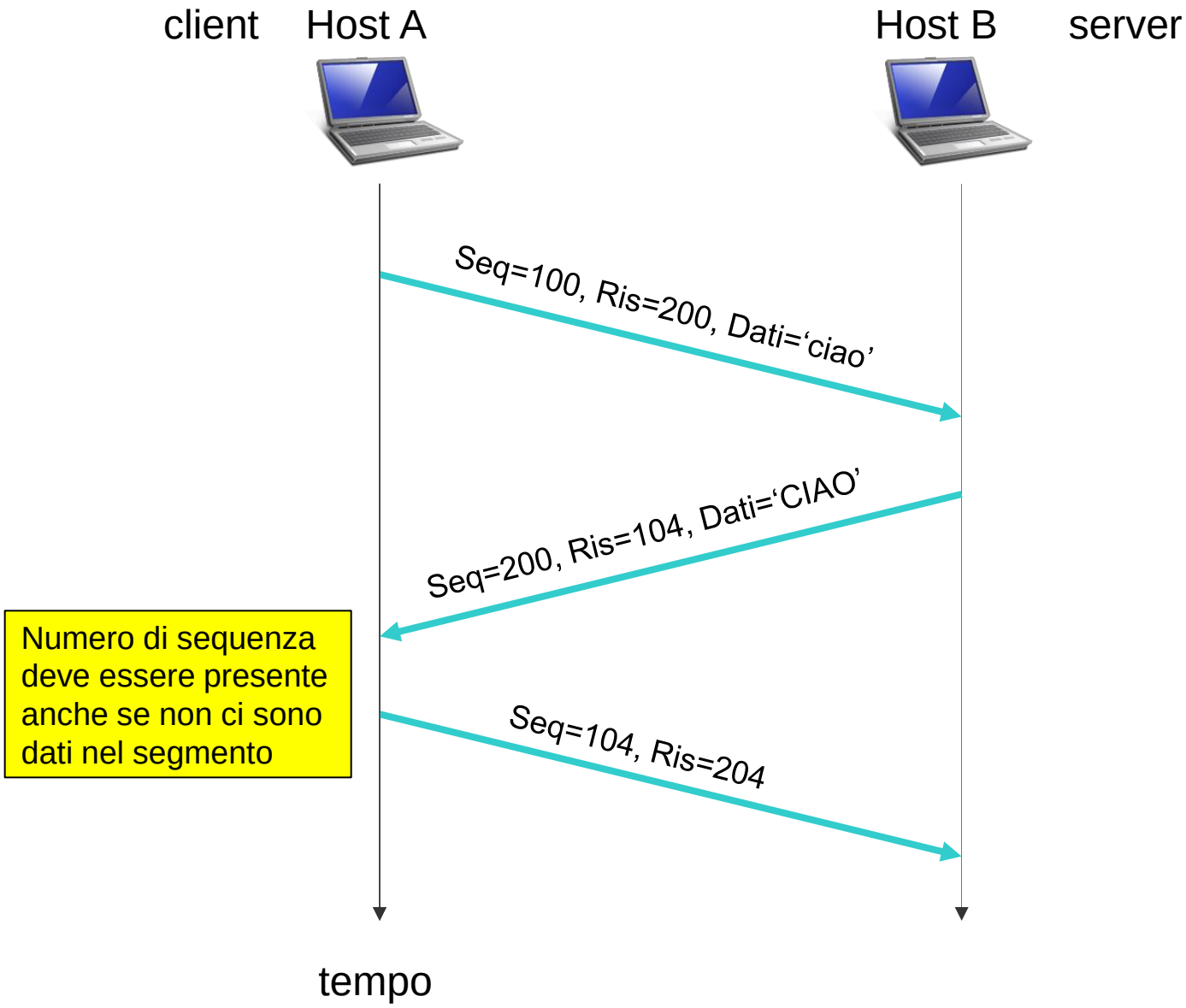
Esempio riscontri

- Supponiamo che l'host B riceva da A un segmento contenente nel campo dati 536 byte, numerati da 0 a 535, e supponiamo che B risponda ad A inviando un segmento. L'host B riscontra i byte ricevuti da A inserendo il valore 536 nel campo *numero di riscontro* del segmento che invia ad A.


Host A
Host B
Seq=0, #Dati=536
Ris=536
$Ris_B = Seq_A + \#Dati_A$
tempo

Esempio di numeri di sequenza e di riscontro

- Per chiarire i numeri di sequenza e di riscontro facciamo riferimento alle applicazioni ClientTCP e ServerTCP scritte in java. Ricordiamo che il client permetteva all'utente di scrivere una frase e di inviarla al server. Il server rinviava al client la stessa frase ma scritta in lettere maiuscole.
- Ora, supponiamo che l'utente digiti la parola "**ciao**"ed esaminiamo i segmenti TCP che client e server si scambiano.
- Supponiamo che i numero di **sequenza iniziale** sia **100** per il client e **200** per il server.
- Quindi, dopo l'instaurazione della connessione, il client attende dati a partire dal byte 200 e il server dati a partire dal byte 100.
- Nell'esempio supponiamo che ciascun carattere abbia la misura di un byte e non consideriamo il carattere *«return».*
- Come mostra la figura seguente, sono spediti tre segmenti.

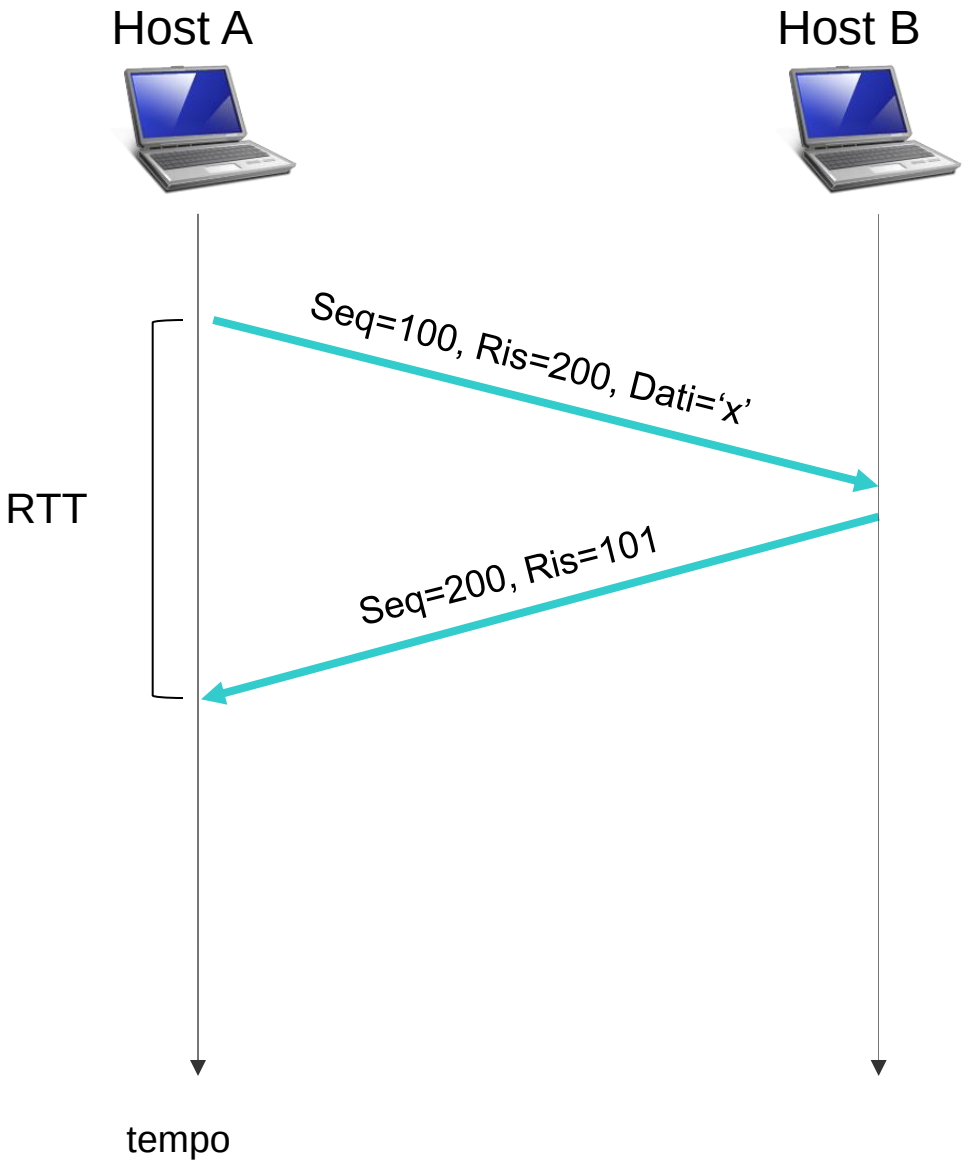

client
Host A
Host B
server
Seq=100, Ris=200, Dati='ciao'
Seq=200, Ris=104, Dati='CIAO'
Numero di sequenza deve essere presente anche se non ci sono dati nel segmento
Seq=104, Ris=204
tempo

- **Il primo segmento,** spedito dal client al server, contiene nel campo dati il codice ASCII (a un byte) della frase ‘ciao’, il numero 100 nel campo **numero di sequenza** e, dato che il client non ha ancora ricevuto dati dal server, il suo avrà 200 nel campo **numero di riscontro**.
- **Il secondo segmento**, inviato dal server al client, svolge due compiti. Primo, fornisce un riscontro per i dati ricevuti dal client. Inserendo 104 nel campo numero di riscontro, il server informa il client di aver ricevuto i dati fino al byte 103 e che ora aspetterà il byte 104. Il secondo compito è il rinvio della frase 'CIAO‘. Questo secondo segmento ha numero di sequenza 200, il numero iniziale di sequenza per il flusso di dati dal server al client di questa connessione TCP, poiché questo è il primo blocco di byte di dati che il server spedisce. Notate che il riscontro per i dati dal client al server è trasportato da un segmento di dati dal server al client; questo riscontro è detto a **piggyback** (a cavalluccio, sovrapposto) sul segmento di dati dal server al client.

- **Il terzo segmento** è inviato dal client al server. Il suo unico scopo è il riscontro dei dati ricevuti dal server. Questo terzo segmento ha il **campo dati vuoto** (cioè, il riscontro non è stato sovrapposto ad alcun dato dal client al server). Il segmento ha nel campo numero di riscontro il valore 204, perché il client ha ricevuto il flusso di byte fino al numero di sequenza 203 e sta ora aspettando i byte dal 204 in poi. **Questo terzo segmento ha anch'esso un numero di sequenza anche se non contiene dati, perché il TCP prevede che questo campo del segmento deve essere necessariamente riempito.**

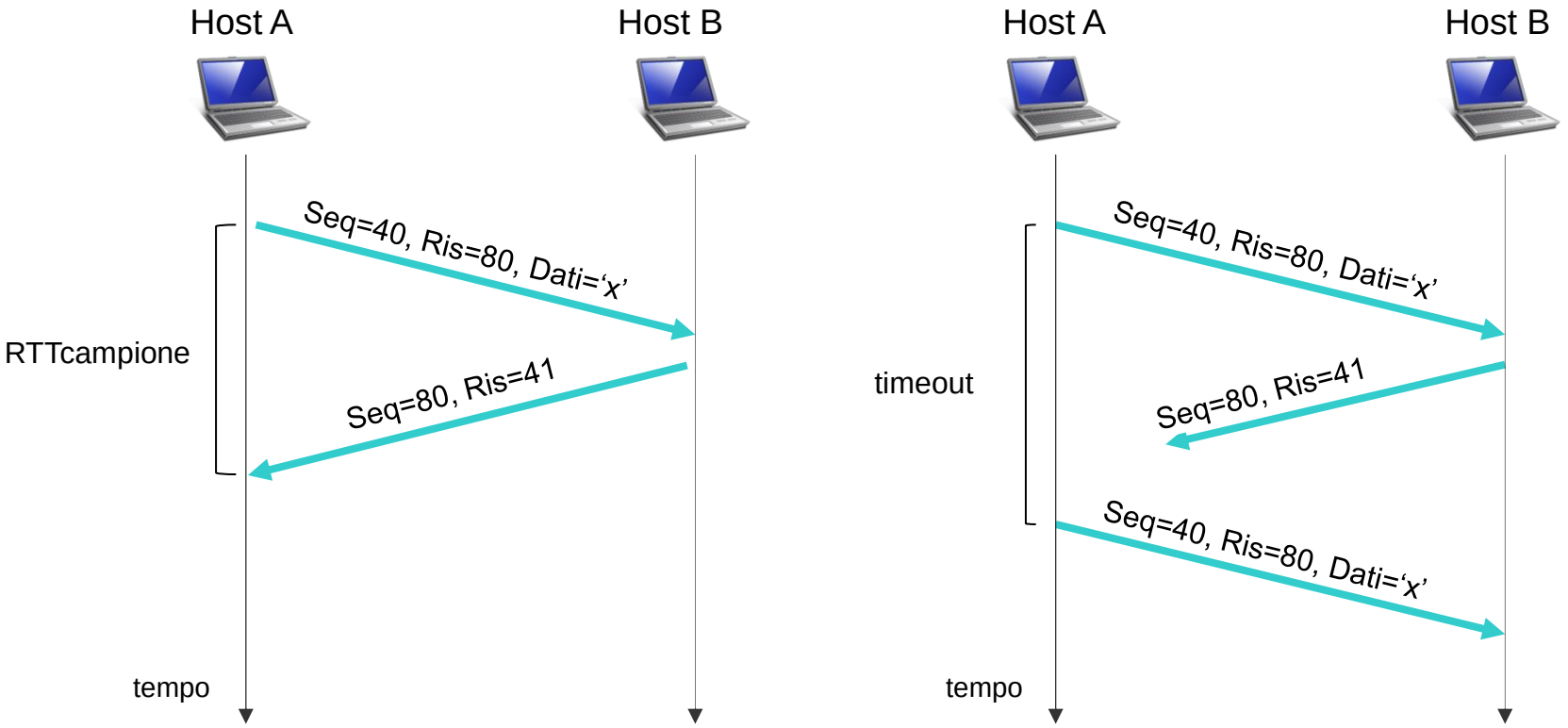
Stima del tempo di andata e ritorno e timeout

- Quando si verifica una perdita di segmenti, il TCP usa algoritmi basati su timeout e ritrasmissione per recuperare i segmenti persi.
- Quando si utilizzano tale algoritmi, bisogna risolvere vari problemi.
- Il primo problema è la scelta della durata degli intervalli di **timeout**. E' evidente che, per evitare ritrasmissioni inutili, il timeout dovrebbe essere maggiore del tempo di andata e ritorno **RTT** (**round trip time**) cioè, del **tempo che passa da quando un segmento viene trasmesso a quando viene riscontrato**.
- Un secondo problema è stabilire quanto deve essere maggiore il timeout rispetto a RTT. E' necessario quindi effettuare una stima di RTT.

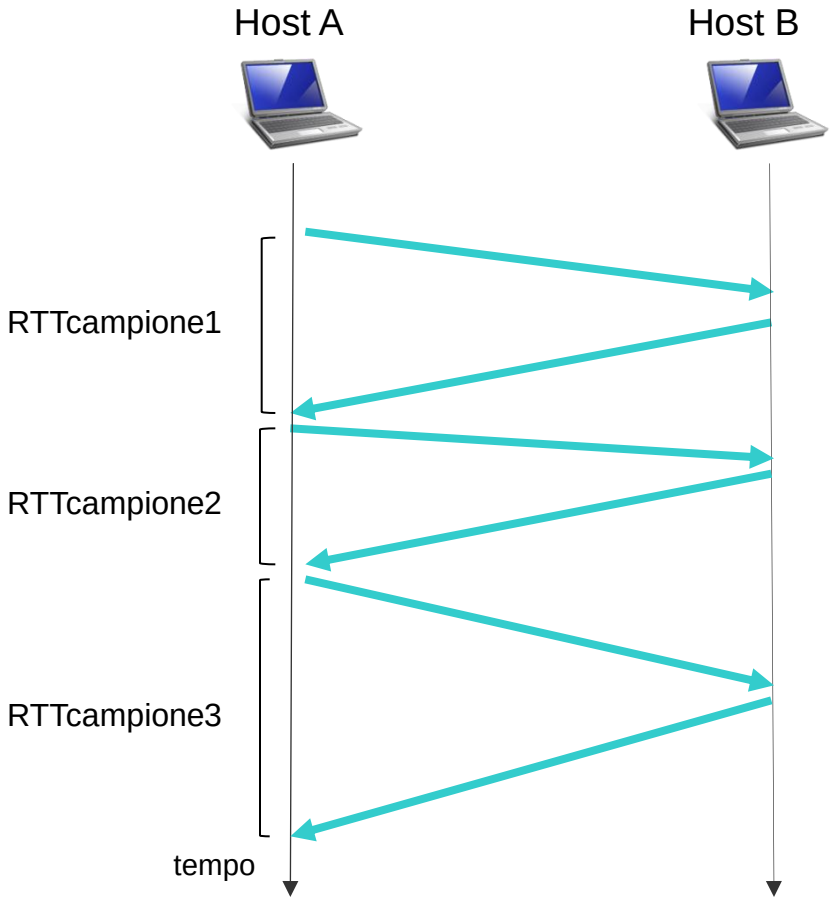

Host A
Host B
Seq=100, Ris=200, Dati='x'
RTT
Seq=200, Ris=101
tempo

Stima del tempo di andata e ritorno (RTT)

- Il TCP misura l'**RTT** (che indicheremo con **RTTcampione**) solo per i segmenti che sono stati trasmessi e riscontrati, non per quelli che vengono ritrasmessi.



- Ovviamente, il valore di **RTTcampione** varierà da segmento a segmento a causa del traffico di rete e al carico variabile sugli host. E' necessario quindi ricorrere ad una stima per ottenere un valore medio per l'RTT.



- Il TCP stima il valore del prossimo RTT calcolando la **media esponenziale mobile pesata** (**EWMA, *Exponential Weighted Moving Average***) dei valori dei RTTcampione misurati negli istanti precedenti:

$$\mathbf{RTTstimato}_{n+1} = \alpha \cdot \mathbf{RTTcampione}_n + (1-\alpha) \cdot \mathbf{RTTstimato}_n$$

- dove $\mathbf{RTTcampione}_n$ è la misura dell'n-esimo RTT, $\mathbf{RTTstimato}_{n+1}$ il valore previsto per il prossimo RTT e α [0..1] è il peso che deve essere assegnato all'ultimo RTT misurato, cioè a $\mathbf{RTTcampione}_n$.
- Con tale stima il peso di un RTTcampione diminuisce esponenzialmente al passare del tempo. Espandendo la relazione si può notare come i valori dei singoli RTTstimato abbiano un peso tanto minore quanto più sono vecchi:

- $$\begin{aligned}\mathbf{RTTstimato}_{n+1} = {} & \alpha \cdot \mathbf{RTTcampione}_n + \\ & (1-\alpha) \cdot \alpha \cdot \mathbf{RTTcampione}_{n-1} + \\ & (1-\alpha)^2 \cdot \alpha \cdot \mathbf{RTTcampione}_{n-2} + \ldots \\ & (1-\alpha)^k \cdot \alpha \cdot \mathbf{RTTcampione}_{n-k} + \ldots \\ & (1-\alpha)^{n+1} \mathbf{RTTstimato}_0\end{aligned}$$

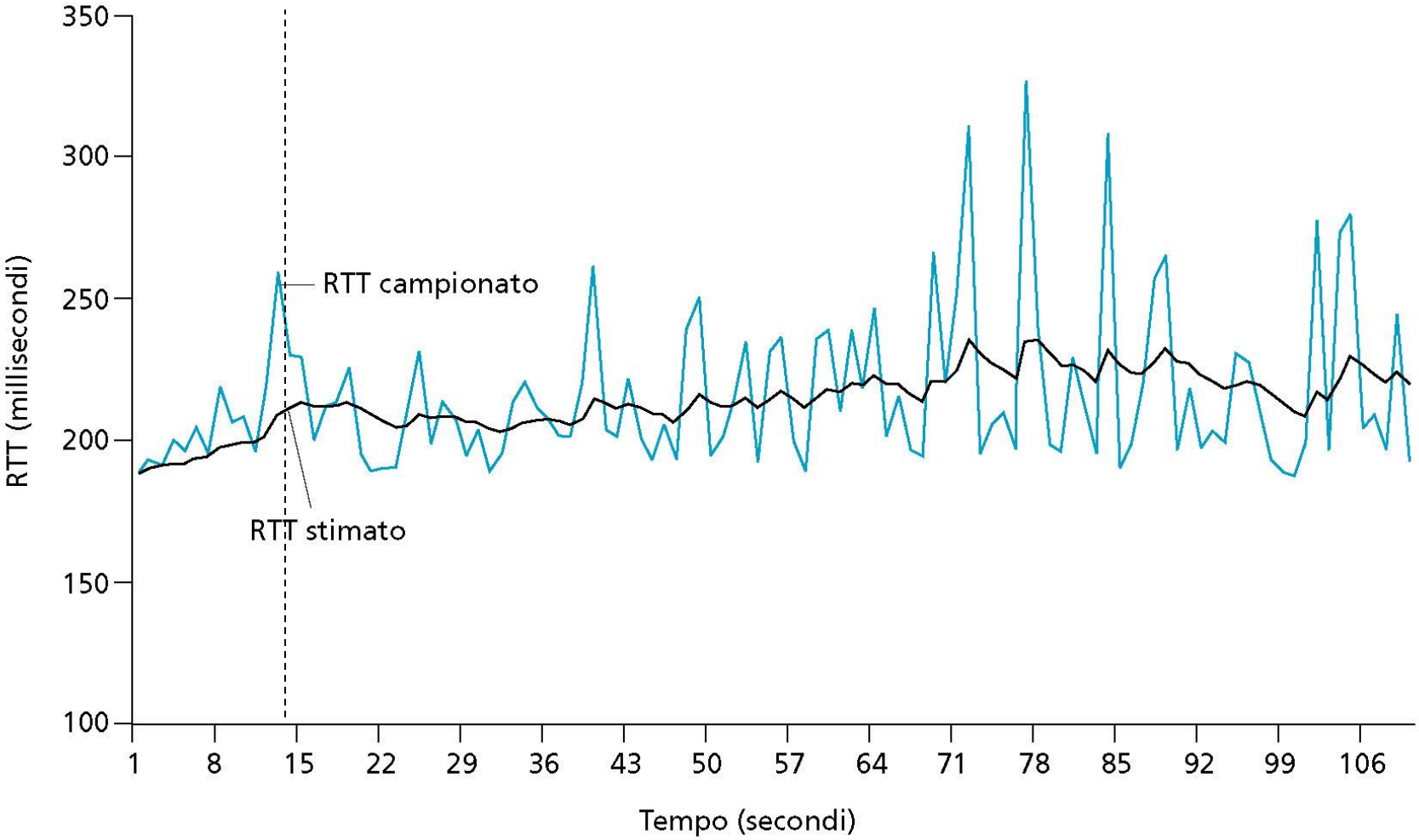
- In conclusione, il TCP, ogni volta che invia un segmento e ne ottiene il riscontro, misura un nuovo valore di RTTcampione e aggiorna RTTstimato in base alla seguente relazione "informatica":

  RTTstimato = α·RTTcampione + (1–α)·RTTstimato

  Il valore raccomandato di α **è 0,125 (1/8)**, quindi possiamo scrivere:

  RTTstimato = 0,125·RTTcampione + 0,875·RTTstimato
  (RTTstimato = 1/8·RTTcampione + 7/8·RTTstimato)

- La figura seguente mostra i valori di RTTcampione e di RTTstimato per un valore di $\alpha = 0{,}125$ (1/8) per un esempio di una connessione TCP fra due host. Come si vede, le oscillazioni nei RTTcampione sono fortemente attenuate nel calcolo di RTTstimato.



RTT campionati e RTT stimati

- Oltre ad avere una stima di RTT, è anche necessaro avere una **misura della variabilità di RTT**. L'RFC 2988 definisce la variazione del tempo di round-trip, **DevRTT**, come una stima di quanto RTTcampione si discosta da RTTstimato:

$$\mathbf{DevRTT = \beta \cdot |RTTcampione - RTTstimato| + (1-\beta) \cdot DevRTT}$$

- È da notare che DevRTT è una media pesata della differenza tra **RTTcampione e RTTstimato**.
- Se i valori di RTTcampione hanno piccole fluttuazioni, allora DevRTT sarà piccola; viceversa, se ci sono ampie fluttuazioni, DevRTT sarà grande. Il valore raccomandato di **β è 0,25**, per cui la relazione diventa:

$$\mathbf{DevRTT = 0{,}25 \cdot |RTTcampione - RTTstimato| + 0{,}75 \cdot DevRTT}$$

Calcolo dell'intervallo di timeout per le ritrasmissioni

- Ora che abbiamo calcolato **RTTstimato** e **DevRTT**, vediamo come usarli per determinare **l'intervallo di timeout** di TCP.
- Chiaramente, l'intervallo di timeout dovrebbe essere maggiore di **RTTstimato**, altrimenti sarebbero effettuate ritrasmissioni non necessarie. Ma il timeout non dovrebbe essere molto maggiore di **RTTstimato** altrimenti quando un segmento si perde, il TCP aspetterebbe troppo tempo, introducendo quindi notevoli ritardi nel trasferire dati per l'applicazione. E' quindi necessario **impostare il timeout pari a RTTstimato più un margine di sicureza**.
- Il margine dovrebbe essere grande quando ci sono fluttuazioni ampie nei valori di RTTcampione, viceversa dovrebbe essere piccolo quando ci sono piccole fluttuazioni. Il valore di **DevRTT** tiene conto di queste fluttuazioni.
- Tenendo conto di tutte queste considerazioni si arriva a determinare l'intervallo di timeout per le ritrasmissioni con la seguente relazione:

$$\mathbf{IntervalloTimeout = RTTstimato + 4 \cdot DevRTT}$$