1 Lezione del 12-11-25

1.1 RDT su TCP

Abbiamo visto i frame del protocollo TCP e le sue funzionalità di gestione della connessione. Adesso concentriamoci su come TCP implementa il **RDT** (*Reliable Data Transfer*).

In 10.3 avevamo introdotto l'RDT su un mezzo qualsiasi (che al tempo associavamo ad un link fisico inaffidabile). Vediamo adesso che tale mezzo inaffidabile può essere rappresentato anche da un link di livello network, come quello offerto dal protocollo IP. Quello che fa TCP è quindi implementare un livello di RDT al di sopra di questo link inaffidabile.

1.1.1 Stima del RTT

Un problema che dobbiamo risolvere prima di poter implementare l'RDT su TCP è quello della stima del **RTT** (*Round Trip Time*), cioè il tempo necessario a inviare un segmento e riceverne l'ACK.

Questo è infatti utile, ad esempio, per calcolare il valore del timeout per i segmenti TCP trasmessi:

- Se il timeout è troppo corto, si hanno ritrasmissioni inutili;
- Se il timeout è troppo lungo, si ha una reazione troppo lenta alla perdita di segmenti.

L'approccio che possiamo usare è quello di fare una stima *a posteriori* dei RTT precedenti, campionati in sede di trasmissioni precedenti. Una nota riguardo a questa stima è che preferiamo non valutare il tempo impiegato da segmenti che richiedono ritrasmissione, in quanto questi sono chiaramente degli *outlier* (sposterebbero la stima inutilmente verso l'alto).

Possiamo quindi realizzare la nostra stima come *media esponenziale mobile*:

```
avg_rtt = sample_rtt * h + avg_rtt * (1 - h)
```

dove `avg_rtt` è l'RTT medio che vogliamo considerare, `sample_rtt` è l'ultimo RTT che abbiamo valutato, e `h` è una qualche costante, con:

$$\texttt{h} \in (0, 1)$$

Al variare di `h` si può dare più o meno peso alle misurazioni passate dell'RTT. Una buona regola è di dare meno peso all'ultima misurazione (prendere $\texttt{h} < 0.5$, solitamente ~ 0.125).

Possiamo svolgere il calcolo dell'RTT stimato (chiamiamolo ERTT) sulla base degli RTT_i misurati, cioè espandere completamente la media esponenziale mobile:
$\text{ERTT}_1 = \text{RTT}_0$
$\text{ERTT}_2 = h \cdot \text{RTT}_1 + (1-h) \cdot \text{RTT}_1$
$\text{ERTT}_3 = h \cdot \text{RTT}_2 + (1-h)h \cdot \text{RTT}_1 + (1-h)^2 \cdot \text{RTT}_1$
...
$\text{ERTT}_{n+1} = h \cdot \text{RTT}_n + h(1-h) \cdot \text{RTT}_{n-1} + h(1-h)^2 \cdot \text{RTT}_{n-2} + ... + (1-h)^n \cdot \text{RTT}_0$

che si riscrive semplicemente come:
$\text{ERTT}_{n+1} = h \cdot \text{RTT}_n + (1-h) \cdot \text{ERTT}_n$

Quest'ultima formula è esattamente quello che lo pseudocodice riportato sopra codifica.

Risulta utile tenere conto anche della deviazione dell'RTT, stimata secondo lo stesso metodo:

```
avg_devrtt = abs(avg_rtt - sample_rtt) * k + avg_devrtt * (1 - k)
```

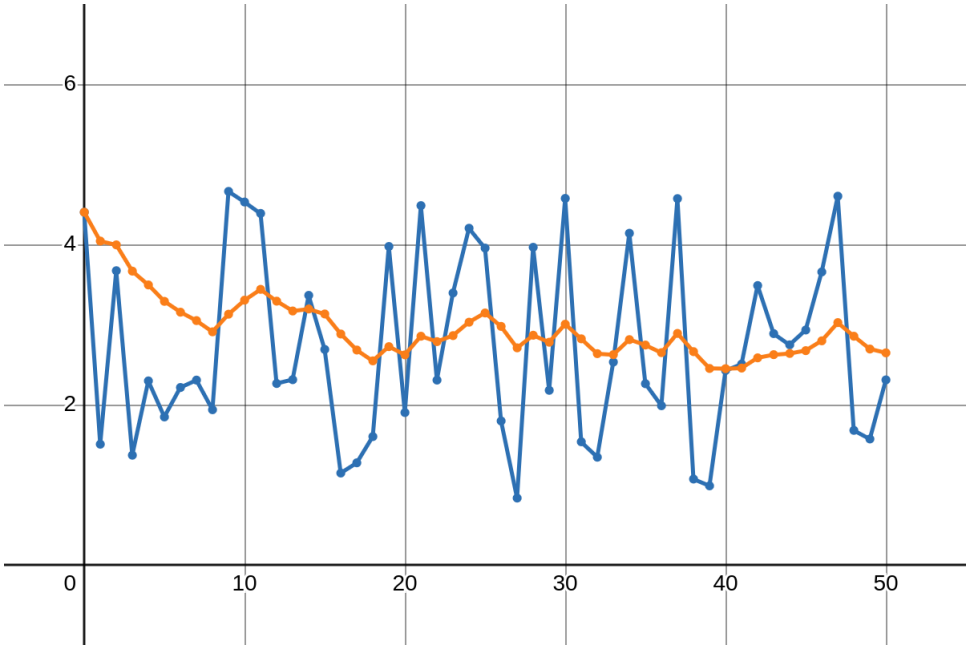
dove `k` è solitamente ~ 0.25.

A questo punto avremo modo di calcolare il timeout ideale (secondo l'euristica presa), come:

```
timeout = avg_rtt + j * avg_devrtt
```

dove j esprime la grandezza di un qualche *"margine di sicurezza"*, con j solitamente ~ 4.

Per concludere questa sezione, vediamo l'esempio di una stima su un dataset casuale, effettuata usando la media esponenziale mobile con $h = 0.125$:



dove in blu si è riportato la funzione originale, campionata ad intervalli $\Delta t = 1$ (per quanto ci riguarda, l'RTT misurato), e in giallo la media esponenziale moile.

1.1.2 Trasmettitore TCP semplificato

Iniziamo quindi a vedere come opera un **trasmettitore TCP**. Avremo che questo andrà sviluppato per *eventi*:

- Evento: dati ricevuti da applicazione.

  1. Crea un segmento con numero di sequenza pari al numero, nel bytestream, del primo byte nel segmento;
  2. Avvia il timer se non è già stato avviato. Consideriamo questo timer come un timer per il più vecchio segmento privo di ACK.

     In questo TCP è più simile al *go-back-N* (vedere 10.3) che al *selective repeat*, con la differenza che ritrasmettiamo il primo segmento e non tutti i successivi per cui non si è avuto ACK. In ogni caso, c'è un solo timer in ogni momento.

     Notimo inoltre che il tempo di timeout è ottenuto dai calcoli visti nella scorsa sezione.

- Evento: timeout.
  1. Ritrasmetti il segmento che ha causato timeout (quindi il più vecchio segmento privo di ACK);
  2. Riavvia il timer.
- Evento: ACK ricevuto.

  In questo caso procediamo solo se l'ACK è per segmenti ancora privi di ACK.
  1. Aggiorna lo stato interno impostando il segmento su cui si è fatto ACK come fornito di ACK;
  2. Se ci sono ancora segmenti privi di ACK (i successivi su cui avevamo detto non si fa ACK come in *go-back-N*), riavvia il timer.

1.1.3 Ricevitore TCP semplificato

Discutiamo quindi il **ricevitore TCP**. Anche qui adottiamo un simile approccio ad *eventi*:

- Evento: arrivo di un segmento in ordine con numero di sequenza aspettato. Tutti i dati precedenti hanno già ricevuto ACK.
  1. Effettuiamo un ACK ritardato: iniziamo con aspettare 500ms;
  2. Quando i 500ms scadono, se non è stato ottenuto nessun segmento, si fa ACK.
- Evento: arrivo di un segmento in ordine con numero di sequenza aspettato. Almeno un'altro segmento ha un ACK in attesa (dall'ACK ritardato).
  - Si fa immediatamente un ACK cumulativo per gli ultimi due segmenti in ordine.
- Evento: arrivo di un segmento fuori ordine. Si rileva un buco.
  - Si fa immediatamente un **ACK duplicato**, indicando il numero di sequenza del prossimo byte aspettato.
- Evento: arrivo di un segmento che riempie parzialmente o completamente un buco.
  - Si invia immediatamente un ACK, ammesso che il segmento risieda nell'estremo inferiore del buco.

Abbiamo quindi introdotto una nuova funzionalità non ancora discussa: quella dell'*ACK duplicato*. Questo è un meccanismo che il ricevitore TCP può sfruttare per richiedere al trasmettitore una **ritrasmissione rapida**.

Sostanzialmente, abbiamo che una ritrasmissione normale avviene quando scatta il timeout su un certo segmento (che abbiamo detto essere il primo privo di ACK lato trasmettitore). In TCP si prevede anche che il ricevitore segnali esplicitamente al trasmettitore di aver perso un segmento, appunto attraverso gli ACK duplicati. Quando il trasmettitore riceve un ACK duplicato, reinvia immediatamente i dati richiesti senza aspettare il timer.

In verità, la maggior parte delle implementazioni di TCP non si aspettano propriamente ACK duplicati, ma aspettano una serie di ACK consecutivi con lo stesso numero di sequenza (nell'ordine di ~ 3) prima di effettuare una ritrasmissione.

1.2 Controllo di flusso TCP

Iniziamo quindi a vedere le basi del **controllo di flusso** in TCP. Questo non è da confondersi col *controllo di congestione*, che verrà visto in seguito.

L'obiettivo del controllo di flusso è infatti realizzare una coordinazione fra trasmettitore e ricevitore, che permetta al primo di modulare la sua velocità di trasmissione in modo che il secondo non incorra in problemi di overflow del buffer.

L'idea di fondo del controllo di flusso TCP è quello di permettere al ricevitore di *"pubblicizzare"* lo spazio di buffer libero. Ricordiamo che nell'header di segmento TCP (visto in 20.3.1) avevamo previsto un campo `<receive window>`. Questo può essere appunto usato per indicare la dimensione del buffer disponibile al ricevitore.