1 Lezione del 14-11-25

Riprendiamo la discussione del controllo di flusso TCP.

Avevamo introdotto come il ricevitore TCP poteva in qualche modo *"pubblicizzare"* lo spazio libero di buffer attraverso il campo `<receive window>` dell segmento TCP. Ciò che fa il trasmettitore è quidni limitare il numero di segmenti privi di ACK (i cosiddetti segmenti *in-flight*) allo spazio pubblicizzato in `<receive window>`.

Interroghiamoci su come sono disposti i buffer di ricevitore e trasmettitore a tempo di trasmissione di una certa mole di dati.

- Vediamo innanzitutto il **ricevitore**. Ci aspetteremo di mantenere nel suo buffer i seguenti puntatori:
  - `last_byte_read`: punta all'ultimo byte letto dal livello applicazione;
  - `last_byte_acked`: punta all'ultimo byte ricevuto correttamente, su cui si è fatto ACK, non ancora letto dal livello applicazione;
  - `last_byte_recv`: punta all'ultimo byte ricevuto su cui non si è ancora fatto ACK. Notiamo che questo potrebbe trovarsi oltre `last_byte_acked` in quanto potrebbero esserci buchi nel buffer (segmenti in ordine non ancora ricevuti, per cui `last_byte_recv` punta a un segmento fuori ordine (o la fine di un loro blocco)).

  Assunto che la dimensione completa del buffer è `buf_size`, ci chiediamo come quantificare lo spazio libero del buffer. Certamente i segmenti di *"buco"* presenti fra `last_byte_acked` e `last_byte_recv` non andranno considerati, in quanto se sono arrivati segmenti fuori ordine successivi, i segmenti precedenti sono stati trasmessi e quindi verranno ritrasmessi dal server a breve.

  Calcoliamo lo spazio libero come `buf_size` meno lo spazio occupato, cioè:

```
receive_window = buf_size - (last_byte_recv - last_byte_read +
    buf_size) % buf_size
```

  che sarà esattamente quello che inseriremo nei segmenti di ACK del ricevitore, al campo `<receive window>`.

- Vediamo quindi il **trasmettitore**. Nel suo buffer vorremo i puntatori:
  - `last_byte_acked`: punta all'ultimo byte inviato correttamente su cui si è ricevuto un ACK;
  - `last_byte_sent`: punta all'ultimo byte inviato, su cui non si è ancora ricevuto ACK.

  Quello che il trasmettitore dovrà fare sarà quindi mantenere il numero di byte privi di ACK al di sotto della `<receive window>` ricevuta, ergo assicurare che:

```
(last_byte_sent - last_byte_acked + buf_size) % buf_size <=
    receive_window
```

1.1 Controllo di congestione TCP

Vediamo quindi un altro meccanismo fornito da TCP: il **controllo di congestione**.

Una **congestione** si verifica quando le sorgenti di una rete inviano dati a una frequenza troppo alta per essere gestita dalla rete. Per le applicazioni si manifesta sotto forma di:

- Lunghi *ritardi* (dati dall'accodamento nei router);
- *Perdita* di pacchetti (dati dall'overflow nei buffer di router, cioè sempre nelle code).

1.1.1 Differenze col controllo di flusso

Potrebbe essere facile confondere *controllo di flusso* e *controllo di congestione* TCP. Ricordiamone le differenze:

- Il **controllo di flusso** si occupa di prevenire sovraccarichi al *ricevitore*, assunto che la rete stessa funzioni a pieno regime;
- Il **controllo di congestione** non si preoccupa invece del ricevitore (che assume come a capacità infinita), ma ha come obiettivo la salvaguardia della rete stessa. Questo significha che ha come obiettivo la prevenzione di sovraccarichi dell'infrastruttura di rete stessa (cioè dei router).

La soluzione per il controllo di flusso era stata ottenere esplicitamente lo spazio di buffer disponibile dal ricevitore e modulare il numero di segmenti *in-flight* su di esso. Per il controllo di congestione dovrà invece essere una limitazione complessiva della frequenza di trasmissione, coordinata con i router e fra più sorgenti all'interno della stessa rete.

1.1.2 Controllo di congestione network-assisted

Il controllo di congestione *network-assisted* (o **assistito dalla rete**) prevede che i router siano capaci di segnalare alle sorgenti che inviano dati il suo stato di congestione.

Ciò che vorremo segnalare sarà quindi il *livello* di congestione in corso, se non direttamente specificare alle sorgenti la frequenza di trasmissione desiderata.

Riguardo a *come* i router possono effettuare questo tipo di comunicazione a ritroso verso le sorgenti, possiamo fare alcune considerazioni: Avevamo detto, discutendo il segmento TCP in 20.3.1, che sono presenti due bit (C ed E) di notifica di congestione esplicita. Il router potrebbe aspettare, per un dato datagramma inviato, il successivo ACK per impostare tale bit. Era già stato appurato, però, che questa soluzione è effettivamente inutilizzata.

1.1.3 Controllo di congestione end-to-end

Vediamo quindi l'approccio alternativo al controllo di congestione, detto **end-to-end**, che viene più usato oggi.

In questo caso non prevediamo che i router ci diano alcun tipo di feedback, ma *rileviamo* la congestione agli host sulla base dei ritardi e la perdita di segmenti osservati.

Esistono diversi modi per rilevare la perdita di segmenti:

- Semplicemente, lo scatto del timeout di segmento segnala che quel segmento è stato realisticamente perso, e va reinviato;
- Come abbiamo introdotto in 21.1.3, una funzionalità del TCP è che un certo numero di *ACK duplicati* (avevamo detto 3) segnalano con molta probabilità che un segmento è stato perso.

Notiamo che la perdita dei segmenti non è direttamente legata alla congestione di rete: ad esempio i segmenti si possono perdere per malfunzionamenti a livelli inferiori. In ogni caso, i nostri obiettivo sono:

- Ottenere la *massima* frequenza di trasmissione;
- Contemporaneamente, ridurre al *minimo* le congestioni.

I due obiettivi sono chiaramente in contrasto.

1.1.4 Finestra di congestione

Vediamo quindi come le sorgenti agiscono sul traffico in uscita quando si rileva una congestione.

Quello che vorremo stabilire è una certa `congestion_window` (finestra di congestione), che ci darà una disequazione simile a quella vista per il controllo di flusso:

```
(last_byte_sent - last_byte_acked + buf_size) % buf_size <=
    congestion_window
```

`congestion_window` e `receive_window` potrebbero sembrarci simili. Effettivamente, il trasmettitore è limitato, in quanto a segmenti *in-flight*, da:

$$\texttt{window} = \min(\texttt{receive\_window}, \texttt{congestion\_window})$$

La differenza starà nel fatto che la `congestion_window` detterà anche la *frequenza* di trasmissione del trasmettitore. In particolare, vorremo che:

$$f = \frac{\texttt{congestion\_window}}{RTT}$$

dovee l'RTT è il *Round Trip Time* stimato come in 21.1.1 (al tempo ci serviva per determinare il timeout), e `congestion_window` varia nel tempo sulla base della congestione di rete percepita dal trasmettitore.

Ciò che vogliamo fare è quindi, nella pratica, assicurarci di inviare al massimo `congestion_window` byte per ogni RTT.

1.1.5 Stima della finestra di congestione

Per la stima della finestra di congestione adottiamo un'approccio detto **AIMD** (*Additive Increase, Multiplicative Decrease*).

I trasmettitori in questo caso possono aumentare la frequenza di trasmessione finché non si verifica perdita di segmenti (che segnala congestione). A questo punto sono obbligati a ridurre la frequenza.

La particolarità è come si effettuano tali aumenti e riduzioni:

- L'*aumento* è **additivo**: si aggiunge una certa quantità di banda (1 MSS) ad ogni iterazione, portando ad un andamento lineare;
- La *riduzione* è invece **moltiplicativo**: si divide per 2 in evento di triplo ACK duplicato, portando a riduzioni molto più ripide degli aumenti.

Nel dettaglio, l'attivita di controllo di congestione del trasmettitore dovrà svolgersi in *fasi*:

1. **Slow start**;
2. **Congestion avoidance**;
3. **Reazione ad eventi di timeout**.

La fase di *congestion avoidance* è quella vista finora. Analizziamo nel dettaglio le altre 2.

1.1.6 Slow start

Vediamo quindi la prima fase, quella di **slow start**.

All'inizio della connessione si stabilisce `congestion_window` uguale a 1 MSS, e quindi la frequenza di trasmissione:

$$f = \frac{MSS}{RTT}$$

La frequenza viene quindi aumentata esponenzialmente (raddoppiando i segmenti inviati, cioè incrementando `congestion_window` di 1 per ogni ACK ricevuto). Tale regime di operazione viene mantenuto fino a che, alternativamente:

- Si ha il primo evento di timeout o perdita segmento;
- La `congestion_window` raggiunge una certa *soglia prefissata*.

  La soglia stessa viene modificata quando si verificano eventi di timeout. Vedremo poi come questo è necessario in quanto potremmo tornare nella fase di slow start in seguito.

Dalla fase di slow start si passa quindi alla fase di *congestion avoidance*, discussa nella scorsa sezione, dove si impiega l'approccio AIMD.

1.1.7 Reazione ad eventi di timeout

L'ultima fase, quella di **reazione ad eventi di timeout**, ci mostra come eventi di perdita pacchetti esplicita (triplo ACK duplicato) e eventi di timeout vengono trattati diversamente:

- Il *triplo ACK duplicato* viene gestito normalmente in fase di *congestion avoidance*, segnala perdita di segmenti e quindi richiede una divisione per due della `congestion_window` .

  Nel caso il triplo ACK duplicato arrivi in fase di slow start, invece, si prende come un evento di perdita e si passa alla fase di congestion avoidance;

- Gli eventi di *timeout* vengono invece gestiti, quando in fase di *congestion avoidance*, passando nuovamente alla fase di *slow start*.

  Questo spiega come mai, nella scorsa sezione, avevamo detto che la soglia della `congestion_window` viene aggiornata in sede di eventi di timeout: ogni volta che rientriamo nella fase di slow start vogliamo uscirne prima, con una finestra di dimensione massima più piccola.

  Nella fase di slow start, l'evento di timeout è sempre visto come evento di perdita e quindi comporta transizione a fase di *congestion avoidance*.

1.1.8 Fast recovery

Esiste una quarta fase, non ancora vista, che esiste in qualche modo in maniera ancillare alla fase di *congestion avoidance*. Questa è la fase di **fast recovery**, dove si giunge in caso di triplo ACK duplicato.

Ciò che accade in fast recovery è che si aumenta `congestion_window` di 1 MSS per ogni ACK duplicato ricevuto sul segmento che a causato l'entrata in fase di fast recovery.

Questo meccanismo è opzionale per TCP, ed è stato introdotto in **TCP Reno**. Le versioni precedenti (dette **TCP Tahoe**) preferivano entrare in fase di slow start a partire sia da un evento di triplo ACK duplicato che di timeout.

1.1.9 TCP Cubic

Un'ulteriore evoluzione di *TCP Reno* è data da **TCP Cubic**.

Senza perdersi nei dettagli, abbiamo che TCP Cubic varia solamente la fase di *congestion avoidance*: in particolae prevediamo che la fase di aumento non sia lineare, ma *cubica*, quindi più veloce per aumentare il throughput.

TCP Cubic è usato di default ad esempio nei sistemi Linux.

1.1.10 Fairness TCP

Uno degli obiettivi di TCP è la **fairness**: nel caso di n host che condividono un link di bitrate R, vorremo che ogni host avesse un throughput pari a:

$$R_i = \frac{R}{n}$$

Immaginiamo allora l'esempio di due sessioni TCP in competizione, che sfruttano l'approccio AIMD:

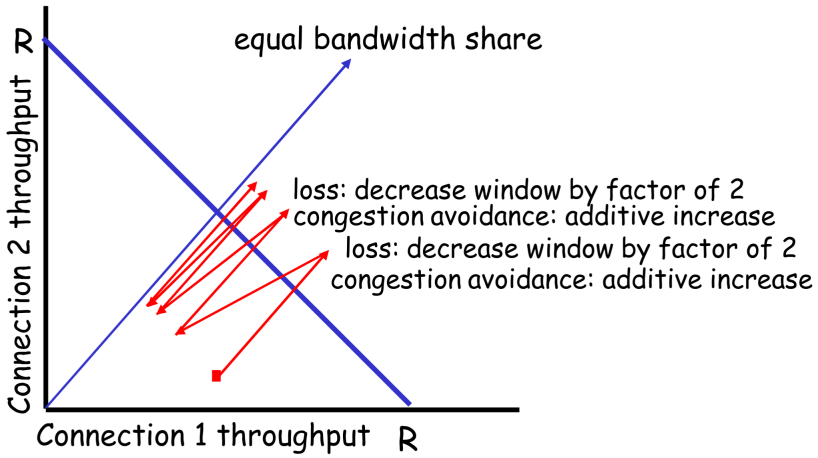
- L'aumento additivo dà pendenza 1, in quanto è lineare;
- La riduzione moltiplicativa riduce invece il throughput in maniera proporzionale.

Questa configurazione favorisce la fairness, in quanto:

1. Inizialmente, entrambe le sessioni cercano di ottenere nuova banda in maniera lineare, cioè aumentare le loro frequenze di trasmissione R_1 e R_2;
2. Prima o poi la frequenza di trasmissione cumulativa $R_1 + R_2$ supera R, per cui si verificano eventi di perdita. In questo caso, per AIMD, si ha una riduzione proporzionale della banda: questo significa che la sessione che aveva più banda perde *più* banda dell'altra sessione;
3. Questo processo si ripete in maniera iterativa ogni volta che le due sessioni superano in frequenza cumulativa il bitrate massimo R.

Il risultato è che si ha uno spostamento delle frequenze di trasmissione, sul lungo termine, verso la divisione equa della banda (in quanto chi invia di più ha effetti più significativi in fase di eventi di perdita).

Possiamo spiegare cosa succede usando il seguente grafico:



dove la linea piena indica la barriera di limite massimo ($R_1 + R_2 \leq R$) del bitrate, e la linea tratteggiata indica dove le sessioni si trovano in competizione equa. Come vediamo, il percorso seguito dalle frequenze di trasmissione si avvicina, iterativamente, verso quest'ultima linea.

Chiaramente questo discorso è valido solo in un mondo *ideale*: non ci aspettiamo che nella realtà tutti gli host rispettino il controllo di congestione TCP.

- I trasmettitori UDP, infatti, non hanno tale meccanismo e usano il rate che preferiscono in ogni momento. Per questo motivo si preferisce usare UDP per applicazioni ad altra prestazione come videogiochi o streaming video.
- Un altro modo per *"barare"* su TCP può essere quello di stabilire *più* connessioni TCP parallele: se vogliamo che ognuna di queste sia fair, aprendo m connessioni TCP si ha m volta la banda che ci meritiamo. Questo è ciò che fanno i browser: aprono diverse connessioni TCP con lo stesso server per ottenere contemporaneamente più dati della stessa pagina. Visto che tale approccio può essere effettivamente considerato poco *fair*, molti amministratori server limitano il numero di connessioni parallele cohe un singolo client può aprire.

  Abbiamo visto come di recente si preferisce usare il protocollo **QUIC** (*Quick UDP Internet Connection*) su HTTP/3.