Università degli Studi di di Napoli Federico II

FACOLTÀ DI SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
Corso di Laurea in Ingegneria Informatica

Il controllo di congestione BBR

Candidati
Riccardo Carbone

Anno Accademico 2016-2017

Indice

Introduzione

Nonostante gli enormi passi avanti che sono stati compiuti nel campo delle tecnologie usate nelle reti di calcolatori, sono ancora molteplici i casi in cui Internet non trasmette le informazioni alle velocità sperate.

Eppure molto s'investe nella ricerca e sviluppo, di un'infrastruttura efficiente per la trasmissione dei segnali, che garantisca elevati livelli di delivery rate end-to-end (in ogni normale condizione).

Come dunque, si può immaginare, la causa principale non è nè l'uso di tecnologie inadeguate, nè la mancanza di conoscenza nel loro uso. Il reale freno alle potenziali velocità raggiungibili, nasce da una scelta effettuata negli anni '80, quando fu definito il primo schema per il controllo di congestione del protocollo TCP.

Tale scelta riguardava l'identificare il fenomeno di congestione (in senso stretto) con il fenomeno di perdita dei pacchetti.

Tuttavia, nonostante i due concetti siano molto differenti, al tempo tale scelta aveva senso, solo in virtù delle limitazioni tecnologiche.

Oggi anche la migliore versione loss-based di TCP: CUBIC, fa fatica a garantire un delivery rate elevato. Basti pensare che per poter sostenere 10Gbps con un RTT di 100ms, è necessario un tasso di perdita dello 0.000003%, e con lo stesso RTT ed un tasso di perdita dell' 1% sono sostenibili solo 3Mbps [RSc17, p. 12].

Lo scopo di quest'elaborato è dunque quello di presentare un'alternativa per tali approcci, che non sia più basata sull'evento di packet loss, ma esplicitamente sull'evento di congestione. L'alternativa che è stata approfondita è il nuovo controllo di congestione BBR.

Prima di iniziare, è lecito far presente che lo sviluppo di tale soluzione è ancora in corso, per cui alcuni aspetti di questa prima versione potrebbero subire delle modifiche nel corso del tempo.

I problemi degli approcci loss-based

In questo primo capitolo, illustreremo come mai un approccio guidato da eventi di packet loss, non riesce a garantire un buon delivery rate.

Il primo passo è comprendere le reali potenzialità del delivery rate.

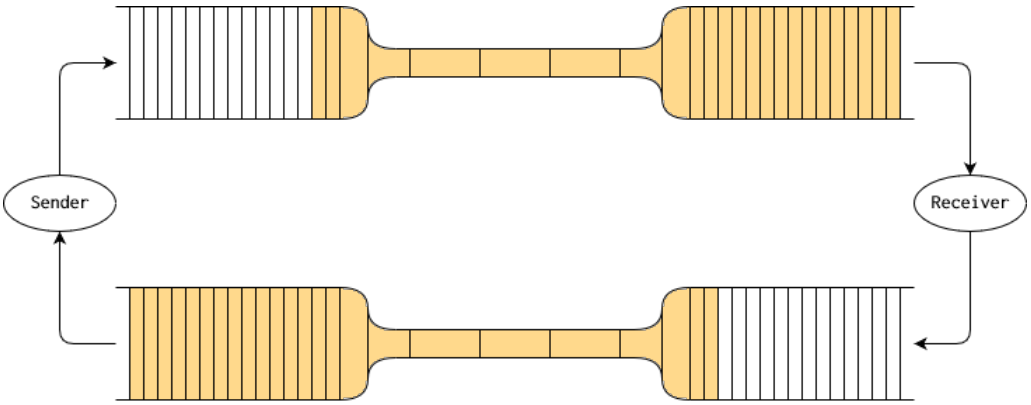
Il limite del delivery rate

Nel momento in cui due end-point iniziano a comunicare attraverso la rete Internet, i loro pacchetti attraverseranno una serie di collegamenti (link) intermedi, ognuno dei quali sarà caratterizzato da una propria larghezza di banda.

In ogni istante, esiste esattamente un solo bottleneck link: il collegamento con la velocità di trasmissione più bassa. E' dunque necessario adeguarsi alle velocità di quest'ultimo, che pertanto determinerà il limite superiore del delivery rate end-to-end.

La seguente figura fornisce un'immagine semplificata ma coincisa, sul ruolo che riveste il bottleneck link nella comunicazione:

Figura 2.1: Visione semplificata del bottleneck link



E' fisicamente impossibile per il delivery rate operare al di sopra del bottleneck rate.

Il sending rate e la congestione di rete

Dalla definizione del bottleneck rate (*BtlBw*), discende subito la seguente affermazione:

Un sending rate > bottleneck rate porta ad una crescita del bottleneck buffer, con conseguente aumento dei ritardi

Per poter caratterizzare il ritardo end-to-end introduciamo un altro parametro caratteristico del path di rete, ovvero il Round-trip-propagation-delay (*RTprop*).

L'*RTprop* costituisce il minimo ritardo di propagazione, o se si vuole il limite inferiore dell'RTT per una data comunicazione.

Ora, il mittente dovrà attendere un RTT prima d'iniziare una seconda fase d'invio. Infatti ricordiamo che l'RTT per un dato pacchetto, è il tempo che intercorre tra la trasmissione e la ricezione del relativo ack. In una tale fase:

- Se il sending rate fosse minore o prossimo al bottleneck rate, il mittente introdurrebbe nella rete un amount inflight[1], tale da non contribuire alla crescita del bottleneck buffer. Per cui in tali condizioni l'RTT = *RTprop*:

Figura 2.2: Delivery rate and round trip time vs inflight part 1

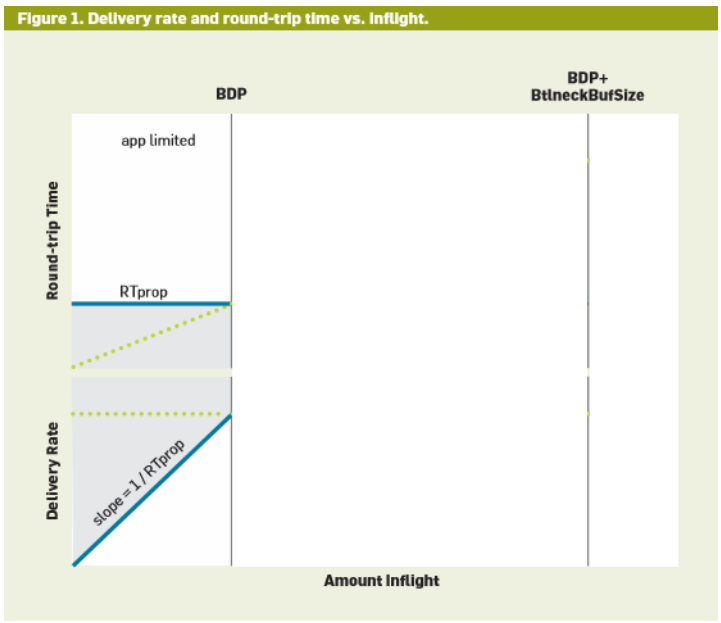


Figura tratta da [Car+17b, p. 60]

- Se il sending rate fosse superiore al bottleneck rate, il mittente introdurrebbe nella rete un amount inflight superiore alla capacità di quest'ultima. L'RTT aumenta, per via della crescita del bottleneck buffer, e per bilanciare l'amount inflight così da fissare il delivery rate sul bottleneck rate:

[1] Quantità di dati inviata dal sender, ma non ancora riscontrata

Figura 2.3: Delivery rate and round trip time vs inflight

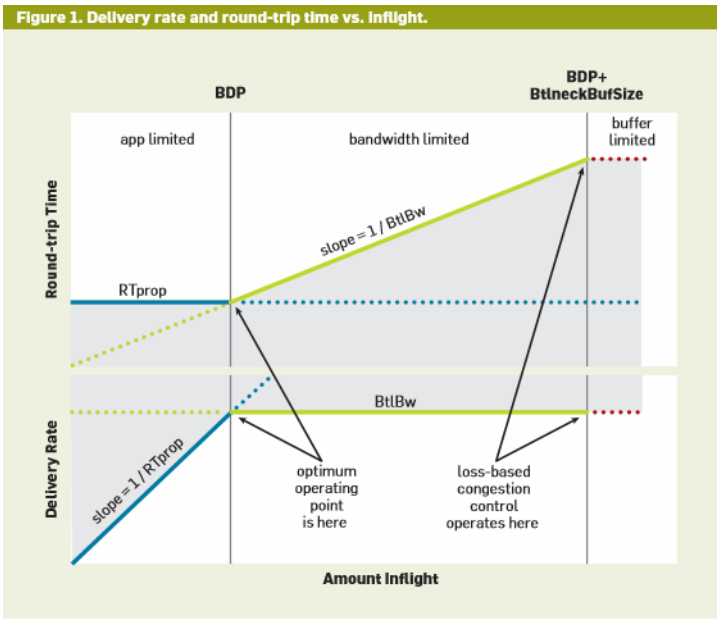


Figura tratta da [Car+17b, p. 60]

Questi due casi sono molto importanti, in quanto nel primo il mittente non contribuisce all'aumento dei ritardi, mentre nel secondo si. E' possibile quindi definire la condizione limite, che separa le due situazioni, introducendo il *BDP* (bandwidth delay product [Car+17b, p. 59]):

$$BDP = BtlBw * RTprop$$

Esso definisce la capacità di rete, ovvero il massimo amount inflight che il mittente può introdurre nella rete in un *RTprop*, al massimo delivery rate, senza contribuire nè alla generazione di code, nè all'espansione dei ritardi.

Osservazione: interpretazione fisica dei parametri caratteristici

Per fornire una immagine più coincisa di tali parametri, possiamo immaginare una pipe virtuale tra il mittente ed il destinatario. Il *BtlBw* definisce il diametro della pipe, l'*RTprop* la sua lunghezza, mentre il *BDP* la sua capacità (per definizione).

Ora abbiamo tutti gli elementi per comprendere in che condizione il mittente contribuisce ad una congestione di rete, basta trasmettere in un *RTprop* una quantità di dati superiore al *BDP* corrente (pipe overfill).

Gli approcci loss based

Tali approcci limitano il sending rate del mittente, ponendo un limite superiore per l'inflight amount: la congestion window (`cwnd`).

Sulla base delle proprie informazioni locali, l'algoritmo modula la congestion window nel corso del tempo.

La modulazione provvede semplicemente ad incrementare la `cwnd` (ogni versione usa tecniche differenti) , finchè non si verifica o un evento di timeout, o un evento di loss.

Nel caso di un timeout, si porta la `cwnd` al valore minimo, in quanto si assume perso tutto ciò che è stato tramesso (anche il BBR manterrà questo comportamento). Nel caso di un packet loss, la `cwnd` subisce un decremento moltiplicativo di un fattore β.

Quest'approccio non produce un controllo accurato del sending rate. Non è infatti possibile rilevare il momento in cui quest'ultimo eccede il bottleneck rate.

Il risultato conseguente sono condizioni operative altalenanti, in cui per la maggior parte si espandono i ritardi, e caricano i buffer intermedi.

Per essere più specifici, a seconda delle dimensioni dei buffer intermedi (in particolare del bottleneck buffer), potremmo riscontrare i seguenti problemi:

- *Shallow buffers* : in virtù della loro piccola dimensione, saranno molto frequenti i fenomeni di perdita. Una strategia AIMD porta a decrementi multipli continui della congestion window, quindi un delivery rate di piccola entità;
- *Deep buffers* : l'algoritmo reagirà nel momento in cui il bottleneck buffer sarà completamente saturo. L'RTT avrà raggiunto in questa condizione il suo picco massimo: valore che cresce all'aumentare della BtlneckBufSize. Quindi nel corso del tempo, è favorita la progressione dei ritardi, e delle perdite.

In aggiunta le perdite portano a dedicare parte della banda alle ritrasmissioni.

A questo punto è comprensibile la necessità di ricercare una strategia alternativa per limitare il sending rate, che non sia proibitiva in termini di loss rate, e che porti il delivery rate in prossimità del bottleneck rate.

Il controllo di congestione BBR

Il comportamento di BBR è dettato da un modello del path di rete, su cui il flusso di pacchetti viaggia. Tale modello traccia l'andamento nel corso del tempo del BtlBw e dell'RTprop.

Guidato dal proprio modello il BBR setta i propri parametri di controllo :

1. `pacing_rate`: è il parametro principale dell'algoritmo, e regola il sending rate del mittente nel corso del tempo;
2. `send_quantum`: definisce la massima unità di aggregazione per i pacchetti da trasmettere. Utile per diminuire l'overhead della CPU del destinatario.
3. `cwnd`: non ricopre un ruolo principale, ma continua ad essere il limite superiore per la quantità di dati inflight possibile del sender.

L'apposita configurazione di tali parametri, permetterà al BBR di operare nel punto operativo migliore (target operating point). Ovvero quello in cui sono soddisfatte le condizioni di :

- *Rate balance* : il sending rate del mittente ed il bottleneck rate sono perfettamente bilanciati;
- *Full pipe* : la quantità di dati inflight è uguale al BDP corrente. La pipe opera al 100%, nulla più, nulla meno;

Nota: focus sull'installazione del BBR

Tutto ciò che successivamente diremo dell'algoritmo BBR, riguarda solo il lato mittente TCP. Non è necessario alcun cambiamento nel destinatario TCP per utilizzare quest'algoritmo di controllo di congestione.

E' importante però mettere l'accento sul fatto che ciò valga per il TCP che già fa uso di un sistema di ARQ (automatic repeat request).

Fondamentalmente il BBR potrebbe essere utilizzato anche da un altro protocollo di livello trasporto, l'importante è che disponga di un sistema di ARQ, in quanto gli ack sono alla base del corretto funzionamento dell'algoritmo.

Il network path model

In questa sezione approfondiamo come vengono misurati i parametri caratteristici del network path model.

Bottleneck bandwidth

Nella regione di bandwidth limited illustrata nel grafico 2.3, il delivery rate è praticamente pari al bottleneck rate, essendo questo il suo limite superiore. Quindi la stima del BtlBw può essere raggiunta attraverso il campionamento del delivery rate.

Il tempo con la quale sono generati i campioni utili alla stima è temporizzato dagli ACK ricevuti dal mittente. Approfondiamo adesso, il processo di generazione dei campioni. (per una trattazione specifica consultare [YCh17b])

3.1.1.1 L'ack rate

Un ottimo indice possibile per il delivery rate è l'ack rate, ovvero il tasso di dati riscontrati istantaneamente dal destinatario.

Per il suo calcolo è necessario mantenere delle informazioni di stato, per ogni pacchetto, e per la connessione:

1. `C.delivered`: quantità di dati riscontrati, fino all'istante `C.delivered_time`;
2. `C.delivered_time`: istante in cui è pervenuto l'ack più recente;
3. `P.delivered`: valore di `C.delivered` all'atto della trasmissione di P;
4. `P.delivered_time`: valore di `C.delivered_time` all'atto della trasmissione di P;

A partire da queste è possibile definire l'espressione dell'ack rate:

ack rate = `(C.delivered - P.delivered) / (C.delivered_time - P.delivered_time)`

E' fondamentale comprendere che l'ack rate è vicino al delivery rate se la quantità di dati riscontrata nel Δt individuato, corrisponde a ciò che effettivamente il mittente trasferisce a destinazione.

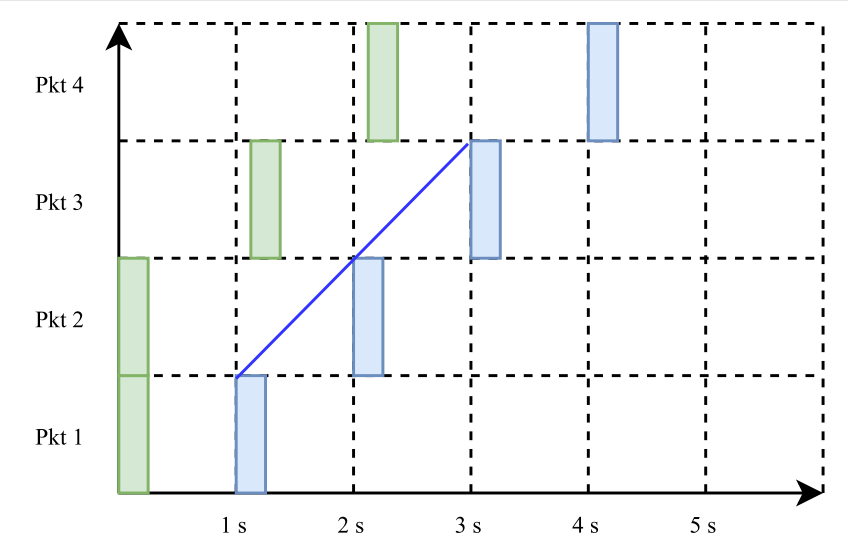
Fenomeni come la compressione, o soppressione degli ack potrebbero alterare la loro normale temporizzazione, portando a sovrastimare o sottostimare (rispettivamente) il delivery rate [Abr15]

La compressione è il principale nemico, in quanto una sovrastima per ragioni chiare avanti, è più pericolosa di una sottostima.

Significato matematico dell'ack rate

Mostriamo una possibile sequenza di trasmissione, da parte di un generico sender:

Figura 3.1: Ack rate determinato in corrispondenza del riscontro per Pkt 3



All'atto della ricezione del riscontro per il pacchetto 3, viene determinato l'ack rate come :

$$ack\ rate = (Pkt3.size + Pkt2.size) / (3s - 1s)$$

Dunque l'ack rate rappresenta la pendenza del segmento evidenziato. Ovvero la pendenza con la quale cresce il tasso di dati riscontrati.

In aggiunta si fa riflettere che il risultato (considerando pacchetti di dimensione fissa) è 1 pacchetto/secondo. Ed intuitivamente è ciò che ci saremo aspettati come delivery rate. (Ecco perchè l'ack rate è un buon indice in condizioni normali)

3.1.1.2 Il send rate

Per valutare la veridicità dell'ack rate, viene introdotto il send rate.

Ancora una volta per la sua determinazione abbiamo bisogno di mantenere delle informazioni di stato (ulteriori alle precedenti), per ogni pacchetto:

1. `P.first_sent_time`: istante in cui è iniziata la trasmissione dell'ultimo pacchetto, riscontrato dall'ack ricevuto al tempo `P.delivered_time` (a monte dell'invio di P)
2. `P.sent_time`: istante in cui parte la trasmissione del pacchetto

A partire da queste definiamo l'espressione del send rate:

send rate = `(C.delivered - P.delivered) / (P.sent_time - P.first_sent_time)`

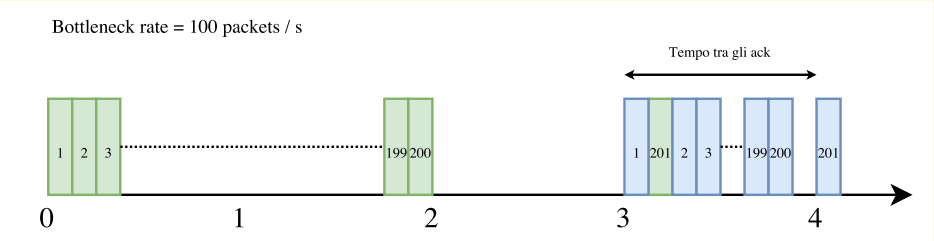
La differenza con l'ack rate sta nell'intervallo di tempo considerato. Il send time rappresenta il tempo che trascorso tra la trasmissione dell'ultimo pacchetto riscontrato, e il pacchetto attualmente riscontrato.

Questo permette che il Δt considerato includa il tempo di trasmissione dei dati riscontrati. Il send time è in generale >= dell'effettivo tempo di trasmissione. Così è possibile evitare il problema delle sovrastime dell'ack rate.

Un caso di compressione degli ack

Mostriamo una possibile sequenza di trasmissione, in cui si verifica una compressione degli ack ricevuti:

Figura 3.2: Fenomeno di compressione degli ack



Come si nota dalla figura gli ack per i pacchetti 2-200 vengono ritardati e compressi, ed arrivano in burst tra la trasmissione del pacchetto 201 ed il relativo ack.

Ciò porta l'ack rate ad essere 200 packets/s, ma si comprende bene che tra 3s e 4s il sender non ha trasferito 200 pacchetti a destinazione. Quindi in tale situazione l'ack rate non è un indice veritiero del delivery rate.

Il reale tempo che i 200 pacchetti impiegano per arrivare al destinatario è di 2s.

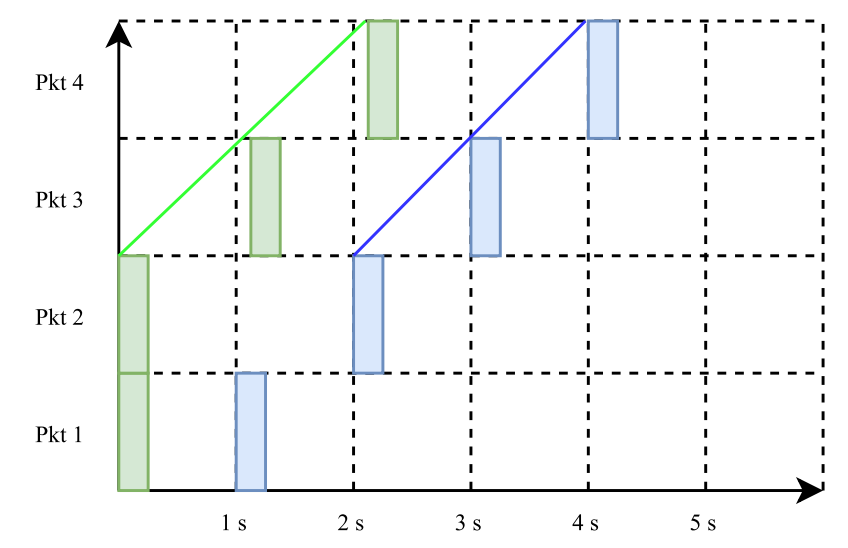
In questo caso il send rate è di 60 packets/s, perchè considera il tempo trascorso dalla trasmissione del pacchetto 1, a quella del pacchetto 201. Il secondo in più è dovuto all'inclusione del secondo tra 2 e 3 secondi, in cui il mittente è inattivo.

Confrontando quindi il send rate e l'ack rate si comprende che il destinatario non può aver ricevuto 200 pacchetti tra 3s e 4s.

Significato matematico del send rate

Consideriamo la stessa sequenza di trasmissione precedente, da parte di un generico sender:

Figura 3.3: Send rate determinato in corrispondenza del riscontro per Pkt 4



All'atto della ricezione del riscontro per il pacchetto 4, viene determinato il send rate come :

send rate = (Pkt3.size + Pkt4.size) / (2s - 0s)

Dunque il send rate rappresenta la pendenza del segmento evidenziato in verde. Ovvero la pendenza con la quale cresce il tasso di dati trasmessi.

3.1.1.3 Stima del BtlBw

Ad ogni ack sarà generato un campione relativo al delivery rate come segue:

Codice 3.1: Delivery rate evaluation

```
time_elapsed = max(send_time, ack_time)
delivery_rate = data_acked / time_elapsed
```

Non ci resta altro che scoprire come da tali campioni venga ottenuta una stima del bottleneck rate.

Per poter far questo BBR usa un BtlBwMaxFilter, un windowed filter, di ampiezza W_B :

$$\widehat{BtlBw_T} = max(deliveryRate_t) \; \forall t \in [T - W_B, T]$$

L'ampiezza W_B è misurata in multipli di RTprop, e deve essere:

- Abbastanza estesa da contenere un ProbeBW cycle gain (sezione 3.3.3). Così che essa contenga dei sample raccolti nella regione di bandwidth limited, che è effettivamente l'unica regione in cui la stima potrà essere corretta;
- Abbastanza estesa da combattere l'eventuale presenza di rumore, introdotta da eventuali code lungo il percorsa;
- Abbastanza corta da poter permettere una pronta a reazione ad una diminuzione del bottleneck rate;

Un ottimo valore è costituito da 10 RTprops. [Car+17b, p. 61]

Round Trip propagation delay

Per la stima dell'RTprop siamo aiutati dal fatto che il protocollo TCP, già effettua un campionamento dell'RTT, per la gestione dei timeout di ritrasmissione.

Anche per determinare RTprop BBR usa un windowed filter, RTpropMinFilter, di ampiezza W_R :

$$\widehat{RTprop}_T \ = \ min(RTT_t) \ \forall t \ \in \ [T - W_R, T]$$

L'ampiezza W_R è misurata in secondi, e deve essere :

- Estesa almeno quanto un ProbeRTTInterval (sezione 3.3.4). Così che essa contenga dei sample raccolti nella regione di application limited, che è effettivamente l'unica regione in cui la stima potrà essere corretta;
- Abbastanza estesa da evitare una consistente diminuzione del delivery rate raramente (sezione 3.3.4);
- Abbastanza estesa da poter sfruttare gli istanti in cui l'applicazione limita il delivery rate del mittente, per poter aggiornare la stima dell'RTprop senza diminuire fortemente il delivery rate;
- Abbastanza corta da poter permettere una pronta reazione ad un variazione dei ritardi;

I parametri di controllo

Compreso in primo luogo come il BBR mantenga aggiornato il modello del network path. Vediamo ora di approfondire i parametri di controllo, usati da quest'ultimo per portarsi nel target operating point.

Pacing rate

Il pacing rate del BBR costituisce un parametro importantissimo (e primario), con cui vengono incontrate nel tempo le condizioni di *Rate balance*, e di *Full pipe*.

Effettuare il pacing dei pacchetti significa equi-distanziare le loro trasmissioni. Quindi a monte dell'invio di un dato pacchetto il successivo sarà trasmesso al tempo:

$$nextSendTime = now + packet.size \,/\, pacing\ rate$$

Ciò porta ad un adattamento del tasso di trasmissione del mittente al pacing rate, nonostante la capacità del proprio link d'ingresso possa essere superiore. La spaziatura definisce dunque le pause adatte, per sincronizzare il sending rate come voluto.

Il set di tale parametro (così come gli altri) avviene ad ogni ack, e può essere riassunto con il seguente pseudocodice:

Codice 3.2: BBRSetPacingRate

```
BBRSetPacingRate():
  BBRSetPacingRateWithGain(BBR.pacing_gain)

BBRSetPacingRateWithGain(pacing_gain):
  rate = pacing_gain * BBR.BtlBw
  if (BBR.filled_pipe || rate > BBR.pacing_rate)
    BBR.pacing_rate = rate
```

Da cui come ci si aspettava il pacing rate è settato proporzionalmente al BtlBw, attraverso il pacing gain. La condizione dell'if non ne permette il decremento, in condizioni di pipe underfilled.

Un valore di pacing gain pari 1 può permettere di operare nel target operating point, se non sono state formate code in precedenza. Un valore di pacing gain > 1 è utile per ottenere sample utili alla stima del BtlBw, in quanto viene saturato il delivery rate. Ed infine un valore di pacing gain < 1, può essere utilizzato per svuotare le code generate, ed uscire da una situazione di pipe overfill.

Send quantum

Rappresenta la massima unità di aggregazione per i pacchetti inviati dal mittente. Permette quindi d'inviare gruppi di pacchetti in un unico burst, diminuendo l'overhead della CPU del destinatario.

Settato ad ogni ack, il suo valore dipende fortemente dal pacing rate. Per valori alti di quest'ultimo, è possibile usare alti valori di send quantum, mentre per valori bassi è meglio diminuire questo ultimo, in quanto creerebbe un aumento dei ritardi, e code lungo il percorso.

Congestion window

La cwnd impone un limite alla quantità di dati inflight del mittente. Nonostante il parametro principale del BBR per il controllo del sending rate sia il pacing rate, la cwnd è molto utilizzata nelle situazioni in cui si vuole subito portare il limite inflight in un particolare punto. Verrebbe speso un maggior tempo se si usufruisse del pacing rate.

Per la sua gestione vengono definiti un upper ed un lower bound:

- *target cwnd*: limite superiore della cwnd, ottenuto in funzione del $\widehat{BDP}$ (BDP stimato) come:

  target cwnd = *cwnd gain* * $\widehat{BDP}$

- *BBRMinPipeCwnd*: limite inferiore della cwnd, tipicamente tale da consetire l'invio di 2/4 pacchetti al mittente.

Tali limiti sono da considerare strettamente rigidi in condizioni normali (non occorrono nè eventi di loss, nè di timeout).

Discutiamo adesso della modulazione del parametro in tali condizioni.

Se la pipe è piena, è molto probabile si sia raggiunto il target. Dunque si provvede ad incrementare la cwnd solo se essa non supererà il target. In caso contrario resterà fissata su quest'ultimo.

Invece se siamo al di sotto del target, avviene un semplice incremento pari alla quantità di dati consegnati a destinazione.

In ogni caso viene sempre garantito il rispetto del lower bound.

Tuttavia, come è ben noto, gli eventi di packet loss e di timeout, portano ad una modulazione diversa della cwnd.

Ad un evento di timeout, il BBR porta la cwnd cautamente ad una SMSS, in quanto viene assunto perso, tutto ciò che è stato inviato. Da qui poi si ripartirà come descritto in precedenza.

In occorrenza invece di un evento di packet loss, la cwnd viene subito (al primo round) adeguata al delivery rate attuale, come segue:

Codice 3.3: BBR enter in loss recovery

```
2 BBR.prior_cwnd = BBRSaveCwnd()
  cwnd = packets_in_flight + max(packets_delivered, 1)
4 BBR.packet_conservation = true
```

Adeguandosi poi, nelle fasi successive, sempre al delivery rate attuale:

Codice 3.4: BBRModulateCwndForRecovery

```
1
BBRModulateCwndForRecovery():
3   if (packets_lost > 0)
        cwnd = max(cwnd - packets_lost, 1)
5   if (BBR.packet_conservation)
        cwnd = max(cwnd, packets_in_flight + packets_delivered)
```

Dopo un round trip time in Loss Recovery, si ripristina la cwnd al suo precedente valore (salvato con BBRSaveCwnd):

Codice 3.5: BBR exit from loss recovery

```
1 BBR.packet_conservation = false
  BBRRestoreCwnd()
3
  BBRRestoreCwnd():
5   cwnd = max(cwnd, BBR.prior_cwnd)
```

A questo punto riportiamo il codice completo di gestione della congestion window:

Codice 3.6: BBRSetCwnd

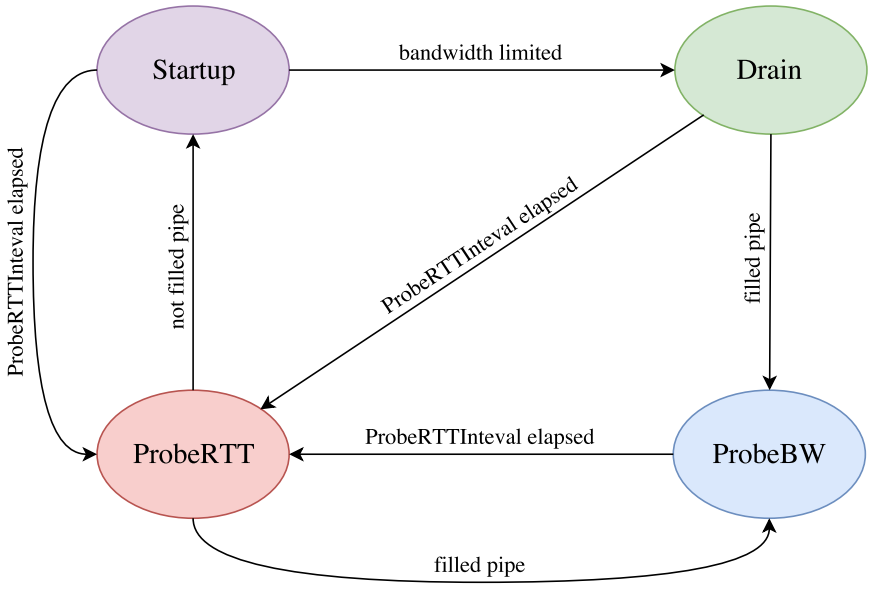
```
  BBRSetCwnd():
2   BBRUpdateTargetCwnd()
    BBRModulateCwndForRecovery()
4   if (not BBR.packet_conservation) {
        if (BBR.filled_pipe)
6           cwnd = min(cwnd + packets_delivered, BBR.target_cwnd)
      else if (cwnd < BBR.target_cwnd || BBR.delivered <
     InitialCwnd)
8       cwnd = cwnd + packets_delivered
      cwnd = max(cwnd, BBRMinPipeCwnd)
10    }
    BBRModulateCwndForProbeRTT()
```

Visione complessiva dell'algoritmo

In questa sezione fonderemo, tutti gli aspetti visti in precedenza. Comprendendo affondo come lavora un flusso BBR nel corso della sua vita.

Il miglior punto di partenza è visualizzare gli stati in cui il flusso può trovarsi, con uno state chart diagram:

Figura 3.4: BBR state diagram



Ricordiamo che gli obiettivi della state machine sono:

- Raggiungere il punto di massimo per il delivery rate, ed il punto di minimo per l'RTT;
- Mantenere continuamente aggiornato il network path model, stimando sequenzialmente, nel corso del tempo, il bottleneck rate, ed il round trip propagation delay;
- Garantire una equa suddivisione della capacità di rete, tra flussi BBR concorrenti;

Passiamo poi all'analisi degli stati, e delle transizioni.

Startup

Durante la fase di Startup, così com'era per il classico Slow Start, è d'interesse scovare subito il limite massimo del delivery rate.

Ora considerando gli standard attuali, è ragionevole avere a che fare con 9 ordini di grandezza (1 Gbps), 10 ordini o 11 ordini di grandezza (10/100 Gbps).

Pertanto è necessario che progressivamente ad ogni round, il sending rate perlomeno raddoppi. Ciò viene ottenuto configurando il pacing gain, ed il cwnd gain ad un valore pari al BBRHighGain (2/ln(2)). Ovvero la quantità di guadagno minima, che garantisce un raddoppio del tasso del mittente:

Codice 3.7: BBREnterStartup

```
BBREnterStartup():
  BBR.state = Startup
```

```
  BBR.pacing_gain = BBRHighGain
  BBR.cwnd_gain = BBRHighGain
```

La fase di Startup termina nel momento in cui si riempie completamente la pipe. Ciò che tipicamente accade, e che non si riesce ad arrivare precisamente nel punto operativo, ma un pò aldilà, generando così delle code, di una quantità pari a (cwnd gain - 1) * BDP.

Per individuare poi, in che momento si è saturato il delivery rate, viene usato il network path model. Ciò che si fa è attendere, che per 3 round di fila, il bottleneck rate stimato non cresca più del 25%. Quindi viene ricercato fondamentalmente un plateau per quest'ultimo.

Motiviamo poi perchè siano necessari 3 round. Il primo round serve al destinatario, per effettuare l'auto-tuning della finestra di ricezione, sulla base dell'attuale capacità di rete. Il secondo round è usato dal mittente per riempire la finestra di ricezione. Il terzo, è quello in cui arriveranno i riscontri, per i pacchetti inviati, e cioè i sample che effettivamente sono indicativi del delivery rate.

Drain

Dopo una fase di Startup entriamo in Drain. Lo scopo di tale fase, è svuotare le code formate in precedenza. Ciò viene ottenuto in un round, invertendo il valore del pacing gain (così da lavorare con un sending rate < bottleneck rate):

Codice 3.8: BBREnterDrain

```
BBREnterDrain():
  BBR.state = Drain
  BBR.pacing_gain = 1/BBRHighGain
  BBR.cwnd_gain = BBRHighGain
```

La fase di drain termina nel momento in cui la quantità di dati inflight arriva al valore attuale stimato del BDP. Quindi le code saranno tutte svuotate.

A questo punto si passa in ProbeBW.

ProbeBW

Il BBR arriva nello stato ProbeBW, senza la presenza di code sul path di comunicazione.

E' in questo stato, in cui il BBR spende la maggior parte del suo tempo.

In ProbeBW, il BBR stima il bottleneck rate, operando in condizioni di massimo delivery rate, e ritardo di propagazione minimo.

Ciò viene ottenuto attraverso una tecnica, chiamata gain cycling. Un gain cycle è tipicamente costituito da 8 fasi, in ognuna di queste fasi, il pacing gain viene impostato ad un valore utile a raggiungere le condizioni sopra espresse.

I valori di pacing gain usati per ogni fase sono i seguenti: [5/4,3/4,1,1,1,1,1,1].

Partiamo dalla fase 0. In questa fase il pacing gain, porta il mittente, ad operare con un pacing rate > BtlBw. Questo serve ad ottenere campioni utili alla stima di quest'ultimo, in quanto è solo nella regione di bandwidth limited (figura 2.3) ove il delivery rate $\simeq$ bottleneck rate.

La fase 0 dura almeno un RTprop, e termina nel momento in cui la quantità di dati inflight è pari a 5/4*BDP, o si verificano degli eventi di loss (non è possibile raggiungere la condizione inflight >= 5/4*BDP).

Dopo la fase 0, l'obiettivo della fase 1, è drenare le code che sono state formate, impostando un valore di pacing rate di 3/4. Dunque in un tempo pari a RTprop, o prima (se le code si svuotano rapidamente), ci riportiamo nuovamente nel target point.

Dopodichè dalla fase 2 in poi, il flusso opera nelle condizioni ideali, con un pacing gain pari a 1.

Proponiamo di seguito uno scenario reale di funzionamento del cycle gaining, in cui il bottleneck rate è di 10Mbps:

Figura 3.5: BBR probe bw gain cycling

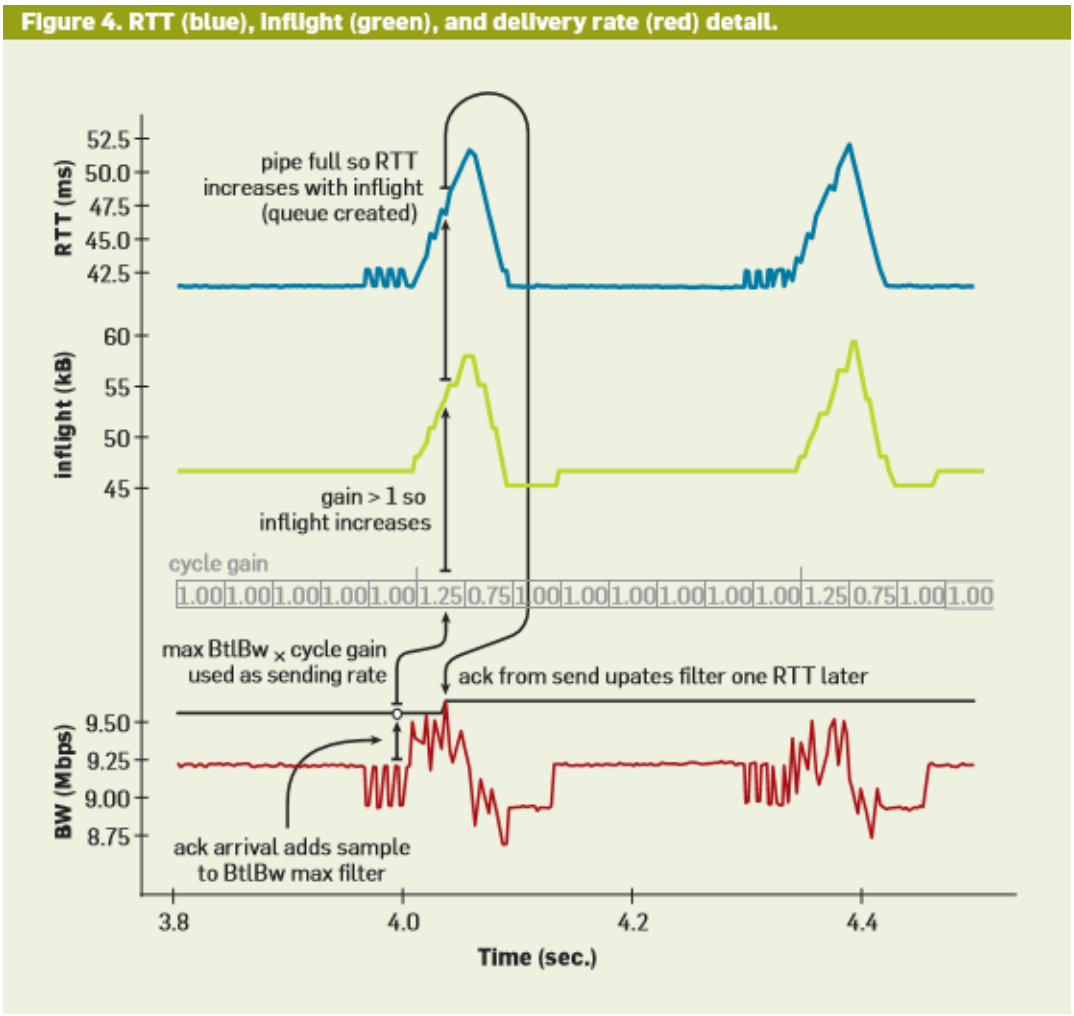


Figura tratta da [Car+17b, p. 62]

Aggiungiamo solo che in questo stato, la cwnd non ha un ruolo centrale, ed il suo target viene impostato a 2*BDP. Questo per non disturbare la modulazione del pacing rate, e per avere comunque un limite di sicurezza per l'inflight amout.

ProbeRTT

Il BBR usa tale stato, per poter ottenere sample dell'RTT utili a stimare correttamente l'RTprop.

Ogni ProbeRTTInterval (di una durata tipica di 10 s) il BBR transita in ProbeRTT, se la stima non è stata aggiornata. In tale stato ci si serve della cwnd per dare un colpo netto all quantità di dati inflight, impostandola al suo limite minimo (BBRMinCwndPipe). In questo modo, il flusso opera nella regione di application limited, ove non essendoci code, si otterrano sample dell'RTT $\simeq$ RTprop.

La cwnd è mantenuta a BBRMinCwndPipe, per un tempo pari al ProbeRTTDuration, che è circa 200 ms. Questa scelta permette di spendere solo il 2% del tempo in ProbeRTT, dove il delivery rate è limitato fortemente, a discapito del 98% speso, giustamente in ProbeBW.

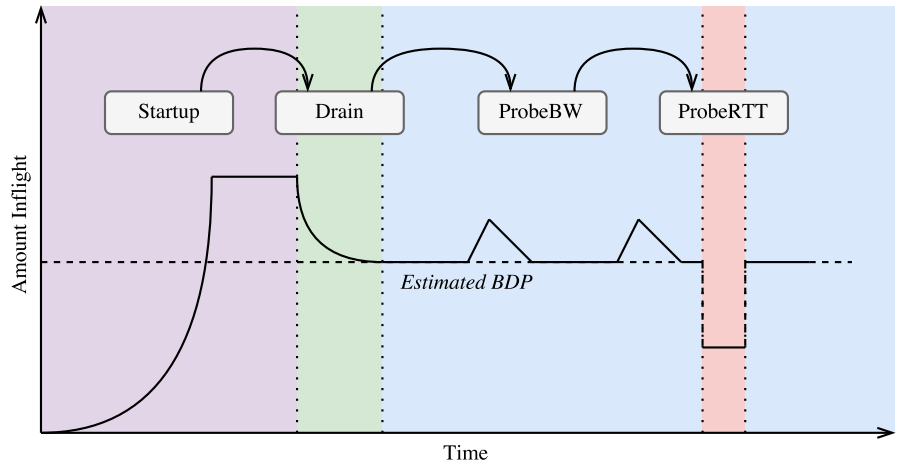
Allo scadere di un ProbeRTTDuration, viene ripristinata la cwnd, e si transita o in ProbeBW, o in Startup, a seconda dello stato attuale della rete.

Evoluzione di un flusso BBR

In conclusione possiamo dire che l'evoluzione di un flusso BBR, passa per un:

1. *Warm up*: si passa in stati non stabili come Startup e Drain, per scovare velocemente il limite della banda di trasmissione;
2. *Stati stabili*: quali ProbeBW e ProbeRTT, in cui il BBR aggiorna il proprio network path, ed imposta i propri parametri per raggiungere il target operating point;

Figura 3.6: BBR state evolution



Casi reali d'impiego del BBR

Cubic vs BBR

Sia Cubic che BBR, si servono di uno stato iniziale, quali lo Slow Start e lo Startup (rispettivamente), per scovare rapidamente il limite superiore del delivery rate.

Tuttavia, hanno un intento differente, essendo due approcci al controllo di congestione diversi.

Il Cubic cerca il limite superiore, per fissare la soglia di passaggio dallo Slow Start, al Congestion Avoidance. In cui semplicemente, continuerà ad aumentare il sending rate, ma in modo più lento. Questo finchè non raggiungerà il punto in cui scadrà un timeout, o il bottleneck buffer sarà saturo. Condizione in cui si verificheranno delle perdite.

Il BBR invece, raggiunto il limite del delivery rate, con la consapevolezza di esser andato un pò oltre (il target point), e di aver generato delle code, passa in Drain per scaricarle. Quindi arriva in ProbeBW, con le condizioni giuste e necessarie ad operare regolarmente al massimo tasso, ed ai minimi ritardi.

Possiamo infatti sottolineare la loro evidente differenza nella gestione dell'RTT, con il seguente scenario:

Figura 4.1: First second of a 10Mbps, 40ms BBR flow

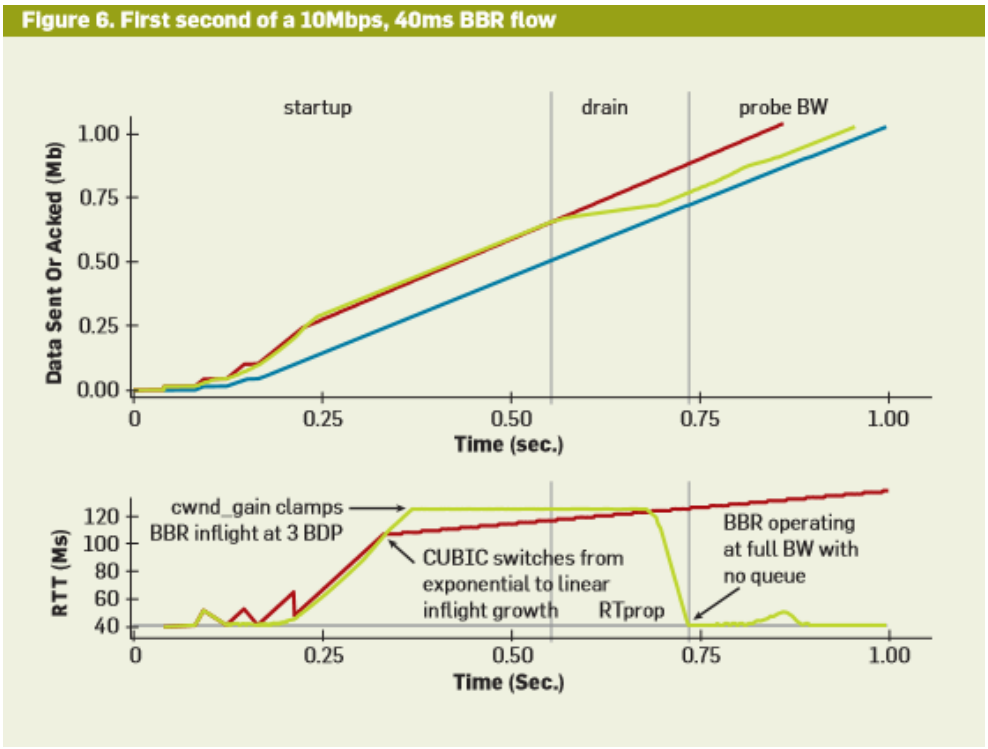


Figura tratta da [Car+17b, p. 63]

L'ambito operativo di Cubic, è sicuramente quello peggiore. In quanto avremo dei livelli alternati di RTT, tra il 70% ed il 100% (con 0% intendiamo la condizione in cui RTT $\simeq$ RTprop), ed un delivery rate si al massimo, ma dimezzato ogni volta in corrispondenza di una perdita (o abbattuto ad un timeout):

Figura 4.2: First eight seconds of a 10Mbps, 40ms cubic and BBR flow

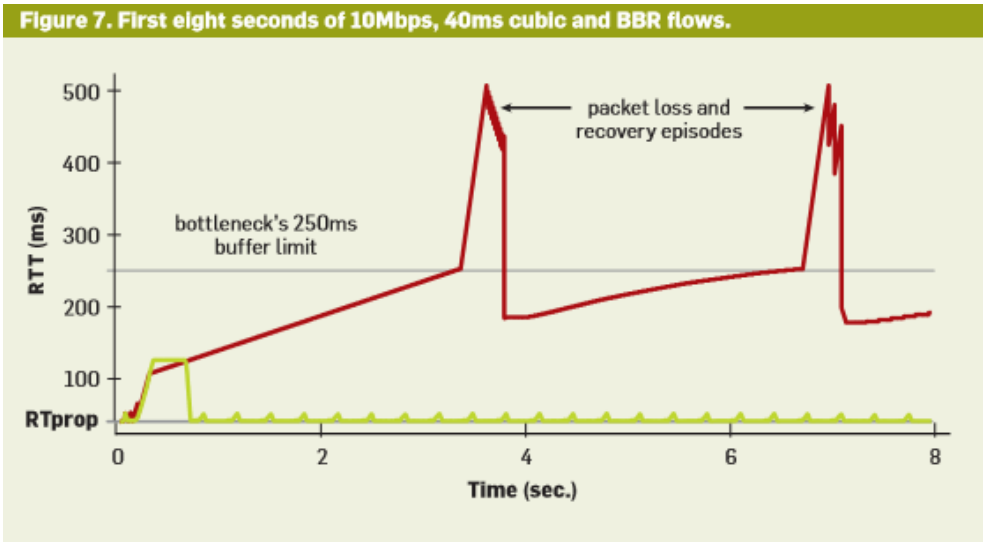


Figura tratta da [Car+17b, p. 64]

Bisogna però prendere atto di una cosa, ovvero che Cubic a differenza del BBR non dispone di un network path model. Quindi non è possibile contenere l'RTT se non lo si traccia nel tempo, e se non è quest'ultimo a determinare il valore dei parametri di controllo della soluzione.

Fairness tra flussi BBR

Tra flussi BBR concorrenti, il livello di fairness raggiunto, testimoniato anche da appositi esperimenti, è molto soddisfacente.

Ecco infatti un caso d'esempio di 5 flussi concorrenti, che condividono un bottleneck link da 100 Mbps, ed un RTT da 10 ms:

Figura 4.3: Throughput of five BBR flows sharing a bottleneck

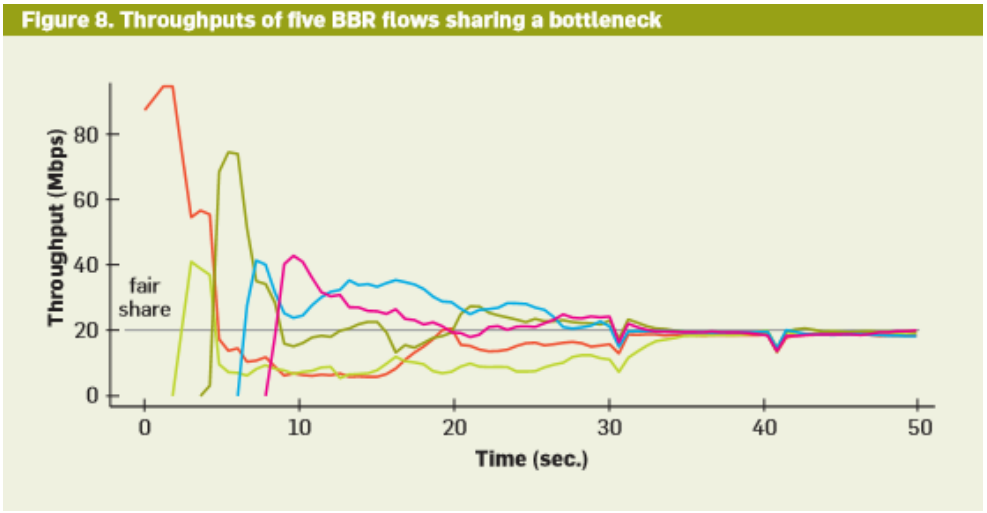


Figura tratta da [Car+17b, p. 64]

Nella figura viene presentato uno scenario abbastanza generale, in cui i flussi BBR non nascono tutti nello stesso istante.

Ciò che ci aspettiamo accada è semplice. Nello stato di ProbeBW i flussi al di sopra del fair share, scopriranno un calo del bottleneck rate. Quindi si accorgeranno della perdita di banda, riconfigurando poi il loro pacing rate.

Andando poi avanti, quando (di nuovo) un flusso al di sopra del fair share - che dunque sta generando un maggior traffico - entrerà in ProbeRTT, svuoterà considerevolmente il bottleneck buffer. Ciò porterà ad un aggiornamento dell'RTprop degli altri flussi concorrenti.

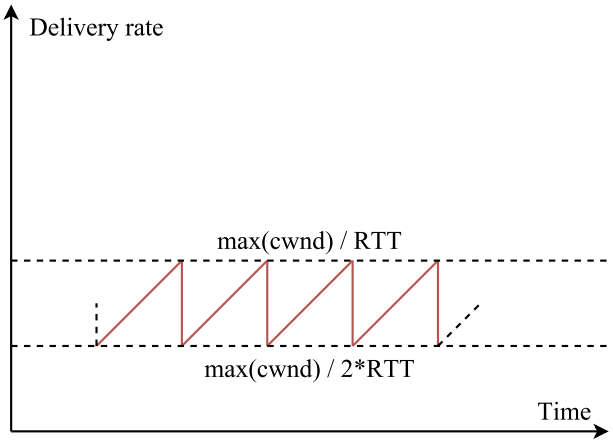
Alla lunga, il tutto porta a sincronizzare l'entrata nello stato di ProbeRTT, di tutti i flussi. Così tutti portranno stimare il reale valore dell'RTprop, così come del bottleneck rate.

Fairness tra flussi loss-based e BBR

In questo caso, è abbastanza (purtroppo) prevedibile, che allo stato attuale delle cose, sia difficile raggiungere una suddivisione equa della banda.

Fondamentalmente, un flusso loss-based presenta un andamento del throughput, pressochè a dente di sega:

Figura 4.4: TCP loss-based evoluzione del throughput



Allora ci aspetteremo che un flusso BBR, abbia maggiore banda, nelle fasi di low-throughput del flusso loss, mentre questi diminuisca quando quest'ultimo arriva al suo massimo (le sue reali possibilità). Tuttavia, un flusso loss, ad esempio come Reno, raggiunge il massimo in corrispondenza di un evento di perdita, per cui non manterrà a lungo tale livello.

Per la consultazione di appositi esperimenti si veda [Car+17a].

Rete Google B4

B4 è una rete proprietaria ad estensione geografica, dalle alte performance, di Google.

Nel 2015, quando BBR non era più solo una teoria, ma una idea concreta, si diede inizio ad una transizione verso quest'ultimo. Così nel 2016, Cubic è stato completamente sostituito dal BBR, che non essendo un approccio loss-based, ha portato un notevole incremento delle prestazioni.

Il primo artefatto d'analisi, mette a rapporto il throughput raggiunto con BBR rispetto quello raggiunto con Cubic:

Figura 4.5: BBR vs CUBIC relative throughput improvement

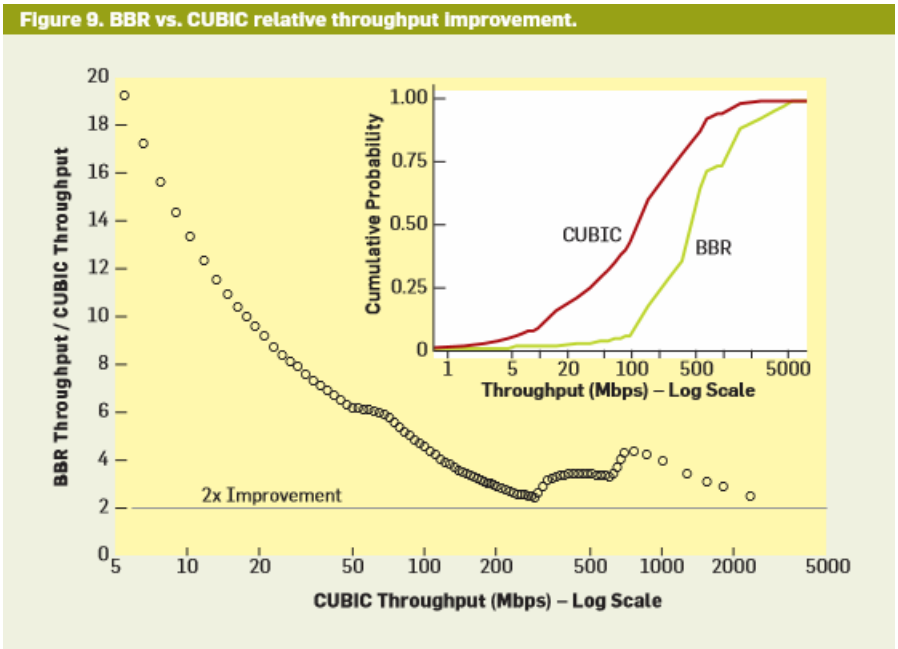


Figura tratta da [Car+17b, p. 64]

Tali dati, raccolti da varie sonde, disposte in vari punti della rete B4, mettono in evidenza un incremento prestazionale che va dal 2x al 25x. In realtà, l'incremento del BBR è mascherato dal limite imposto ai buffer di ricezione TCP (8 MB).

Alzando tale limite è stato ottenuto, su un path tra Europa e USA, un throughput di 2 Gbps, contro i 15 Mbps di Cubic. Parliamo di un incremento: 133x.

Nel grafico vengono anche messe a confronto le distribuzioni di probabilità cumulative. Da esso possiamo notare (qualitativamente) che:

Tabella 4.1: Throughput CDF

	BBR	Cubic
P(thput < 1)	$\simeq 0$	$\simeq 0$
P(thput < 20)	$\simeq 0$	0.19
P(thput < 100)	0.06	0.5
P(thput < 500)	0.62	0.87
P(thput < 5000)	1	1

Il secondo artefatto mette a confronto il throughput utile sostenibile, in funzione del loss-rate:

Figura 4.6: BBR vs CUBIC goodput under loss

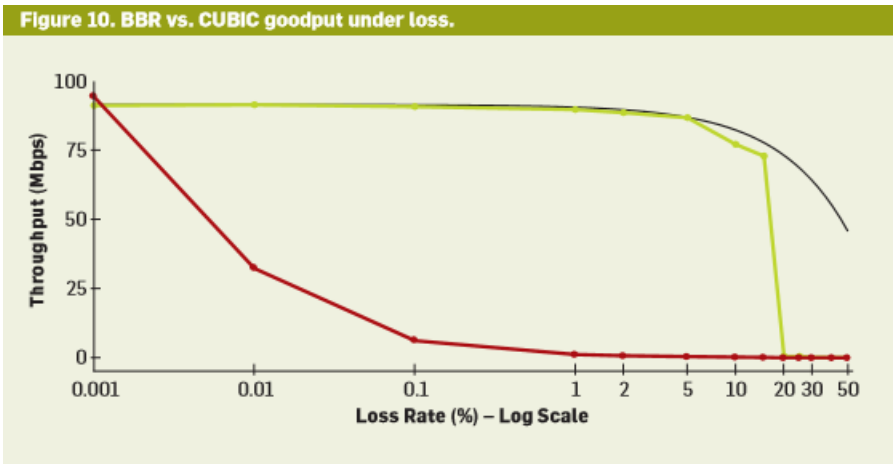


Figura tratta da [Car+17b, p. 65]

Ricordando che un loss-rate ragionevole è circa dell'1%, e notando che il BBR perde performance tra 5% ed 20%, questo è un ottimo miglioramento.

Note

Tutto lo pseudo-codice presente nella relazione, è tratto dagli internet draft:

- [YCh17a]
- [YCh17b]

Bibliografia

[Abr15] Mikael Abrahamsson. *TCP ACK Suppression*. IETF AQM mailing list. Nov. 2015. URL: https://www.ietf.org/mail-archive/web/aqm/current/msg01480.html.

[Car+17a] Neal Cardwell et al. *BBR Congestion Control: An Update*. Slides. Mar. 2017. URL: https://www.ietf.org/proceedings/98/slides/slides-98-iccrg-an-update-on-bbr-congestion-control-00.pdf.

[Car+17b] Neal Cardwell et al. «BBR: Congestion-based Congestion Control». In: *Commun. ACM* 60.2 (gen. 2017), pp. 58–66. ISSN: 0001-0782. DOI: 10.1145/3009824. URL: http://doi.acm.org/10.1145/3009824.

[RSc17] I.Rhee NCSU L.Xu UNL S.Ha Colorado A.Zimmermann L.Eggert NetApp R.Scheffenegger. *CUBIC for Fast Long-Distance Networks*. Internet-Draft draft-ietf-tcpm-cubic-05.txt. IETF Secretariat, lug. 2017.

[YCh17a] Inc Y.Cheng N.Cardwell S.Hassas Yeganeh V.Jacobson Google. *BBR Congestion Control*. Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00.txt. IETF Secretariat, lug. 2017.

[YCh17b] Inc Y.Cheng N.Cardwell S.Hassas Yeganeh V.Jacobson Google. *Delivery Rate Estimation*. Internet-Draft draft-cheng-iccrg-delivery-rate-estimation-00.txt. IETF Secretariat, lug. 2017.