Congestion Control In The Internet
Part 2: How it is implemented in TCP

JY Le Boudec
2017
Contents

6. TCP Reno
7. TCP Cubic
8. ECN and RED
9. Other Cool Stuff
6. Congestion Control in the Internet is in TCP

TCP is used to avoid congestion in the Internet

- in addition to what was shown about TCP, a TCP source adjusts its window to the congestion status of the Internet (slow start, congestion avoidance)
- this avoids congestion collapse and ensures some fairness

TCP sources interpret losses as a negative feedback

UDP sources have to implement their own congestion control

- Some UDP sources imitate TCP: “TCP friendly”
- Some UDP sources (e.g. QUIC) implement the same code as TCP congestion control
TCP Reno, New Reno, Vegas, etc

The congestion control module of TCP exists in \( n \) versions; Popular versions are

- TCP Reno with SACK (historic version)
- TCP Cubic (widespread today in Linux servers)
- Data Center TCP (Microsoft and Linux servers)
- TCP BBR (trendy)
TCP Reno Congestion Control Uses ≈AIMD and Slow Start

TCP adjusts the window size based on the approximation rate \( \approx \frac{W}{RTT} \)

\[ W = \min (cwnd, \text{offeredWindow}) \]

offeredWindow = window obtained by TCP’s window field
cwnd = controlled by TCP congestion control

negative feedback = loss, positive feedback = ACK received

increase is \( \approx \) additive (\( \approx +1 \) MSS per RTT),

Multiplicative Decrease (\( u_1 = 0.5 \))

Slow start with increase factor \( w_0 = 2 \) per round trip time (approx.)

When loss is detected by timeout -> slow start
Loss detected by fast retransmit => fast recovery (see next)
TCP Implementations of...

Multiplicative decrease:

\[ \text{ssthresh} = 0.5 \times \text{cwnd} \]

Additive increase:

for every ack received \( \text{cwnd} += \frac{\text{MSS} \times \text{MSS}}{\text{cwnd}} \)

(if we counted in packets, this would be \( \text{cwnd} += \frac{1}{\text{cwnd}} \))

\[
\begin{align*}
\text{cwnd} &= 1 \text{ MSS} \\
2 & \quad 2.5 \quad 2.9 \quad 3.83
\end{align*}
\]

this is slightly less than additive increase

other implementations exist: for example: wait until the cwnd bytes are acked and then increment cwnd by 1 MSS
...multiplicative increase: (Slow Start)
non dupl. ack received during slow start ->

\[ cwnd = cwnd + MSS \text{ (in bytes)} \] (1)

if \( cwnd = \text{ssthresh} \) then go to congestion avoidance

\[ cwnd = 1 \text{ seg} \]

\[ 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \]

(1) is equivalent in packets to
\[ cwnd = cwnd + 1 \text{ (in packets)} \]
target window of slow start is called ssthresh («slow start threshold») there is a slowstart phase initially and after every packet loss detected by timeout
Fast Recovery

Slow start used when we assume that the network condition is new or abruptly changing
i.e. at beginning and after loss detected by timeout
In all other packet loss detection events, slow start is not used, but “fast recovery” is used instead

▶ Problem to be solved: the formula “rate $\approx \frac{W}{RTT}$” is not true when there is a packet loss – sliding window operation may stop sending

With Fast Recovery
▶ target window is halved
▶ But congestion window is allowed to increase beyond the target window until the loss is repaired
Fast Recovery Details

When loss is detected by 3 duplicate acks

\[ \text{ssthresh} = 0.5 \times \text{current-size} \]
\[ \text{ssthresh} = \max (\text{ssthresh}, 2 \times \text{MSS}) \]
\[ \text{cwnd} = \text{ssthresh} + 3 \times \text{MSS} \text{ (exp. increase)} \]
\[ \text{cwnd} = \min (\text{cwnd}, 64K) \]

For each duplicated ACK received

\[ \text{cwnd} = \text{cwnd} + \text{MSS} \text{ (exp. increase)} \]
\[ \text{cwnd} = \min (\text{cwnd}, 64K) \]

When loss is repaired

\[ \text{cwnd} = \text{ssthresh} \]
\[ \text{Goto congestion avoidance} \]
Fast Recovery Example

During congestion avoidance:
\[ \text{cwnd} \leftarrow \text{cwnd} + \frac{\text{MSS}^2}{\text{cwnd}} \]
MSS = 100

\( \text{TcpMaxDupACKs} = 3 \)

\[ \begin{align*}
\text{ssthresh} &= \text{cwnd} = 800 \\
\text{seq} &= 201:301 \\
\text{seq} &= 301:351 \\
\text{seq} &= 351:401 \\
\text{seq} &= 401:501 \\
\text{ssthresh} &= \text{cwnd} = 813 \\
\text{seq} &= 501:601 \\
\text{seq} &= 601:701 \\
\text{seq} &= 701:801 \\
\text{ssthresh} &= 407, \text{cwnd} = 707 \\
\text{seq} &= 201:301 \\
\text{seq} &= 801:901 \\
\text{ssthresh} &= 407, \text{cwnd} = 807 \\
\text{ssthresh} &= 407, \text{cwnd} = 907 \\
\text{seq} &= 901:1001 \\
\text{ssthresh} &= 407, \text{cwnd} = 1007 \\
\text{ssthresh} &= 407, \text{cwnd} = 407
\end{align*} \]
At time 1, the sender is in “congestion avoidance” mode. The congestion window increases with every received non-duplicate ack (as at time 6). The target window (ssthresh) is equal to the congestion window.

The second packet is lost.

At time 12, its loss is detected by fast retransmit, i.e. reception of 3 duplicate acks. The sender goes into “fast recovery” mode. The target window is set to half the value of the congestion window; the congestion window is set to the target window plus 3 packets (one for each duplicate ack received).

At time 13 the source retransmits the lost packet. At time 14 it transmits a fresh packet. This is possible because the window is large enough. The window size, which is the minimum of the congestion window and the advertised window, is equal to 707. Since the last acked byte is 201, it is possible to send up to 907.

At times 15, 16 and 18, the congestion window is increased by 1 MSS, i.e. 100 bytes, by application of the congestion avoidance algorithm. At time 15, this allows to send one fresh packet, which occurs at time 17.

At time 18 the lost packet is acked, the source exits the fast recovery mode and enters congestion avoidance. The congestion window is set to the target window.
How many new segments of size 100 bytes can the source send at time 20?

A. 1
B. 2
C. 3
D. 4
E. $\geq 5$
F. 0
G. I don’t know
Assume a TCP flow uses WiFi with high loss ratio. Assume some packets are lost in spite of WiFi retransmissions. When a packet is lost on the WiFi link...

A. The TCP source knows it is a loss due to channel errors and not congestion, therefore does not reduce the window  
B. The TCP source thinks it is a congestion loss and reduces its window  
C. It depends if the MAC layer uses retransmissions  
D. I don’t know
Solution

Answer C
The congestion window is 407, the advertised window is 1000, and the last ack received is 901.
The source can send bytes 901 to 1308, the segment 901:1001 was already sent, i.e. the source can send 3 new segments of 100 bytes each.

Answer B: the TCP source does not know the cause of a loss.
A finite state machine description

- **Slow Start**
  - exponential increase
  - new connection:
  - congested avoidance
  - fast retransmit: retr. timeout:
  - fast retransmit: retr. timeout:

- **Congestion Avoidance**
  - additive increase
  - cwnd = ssthresh:
  - retr. timeout:
  - fast retr. transmit: retr. transmit:

- **Fast Recovery**
  - exponential increase beyond ssthresh
  - expected ack received:
  - retr. timeout:
  - retr. timeout:
Fairness of TCP Reno

For long lived flows, the rates obtained with TCP are as if they were distributed according to utility fairness, with utility of flow $i$ given by

$$U(x_i) = \frac{\sqrt{2}}{\tau_i} \arctan \frac{x_i \tau_i}{\sqrt{2}}$$

with $x_i = \text{rate} = \frac{W}{\tau_i}$, $\tau_i = \text{RTT}$

For sources that have same RTT, the fairness of TCP is between maxmin fairness and proportional fairness, closer to proportional fairness.

rescaled utility functions;
RTT = 100 ms
maxmin approx. is $U(x) = 1 - x^{-5}$
TCP Reno and RTT

TCP Reno tends to distribute rate so as to maximize utility of source $i$ given by

$$U(x_i) = \frac{\sqrt{2}}{\tau_i} \arctan \frac{x_i \tau_i}{\sqrt{2}}$$

The utility $U$ depends on the roundtrip time $\tau$;

The utility $U$ is a decreasing function of $\tau$

What does this imply?
$S_1$ and $S_2$ send to destination using one TCP connection each, RTTs are 60ms and 140ms. Bottleneck is link «router-destination». Who gets more?

A. $S_1$ gets a higher throughput
B. $S_2$ gets a higher throughput
C. Both get the same
D. I don’t know

![Diagram](image-url)
Solution

For long lived flows, the rates obtained with TCP are as if they were distributed according to utility fairness, with utility of flow $i$ given by $U(x_i) = \frac{\sqrt{2}}{\tau_i} \arctan \frac{x_i \tau_i}{\sqrt{2}}$

$S_1$ has a smaller RTT than $S_2$.

The utility is less when RTT is large, therefore TCP tries less hard to give a high rate to sources with large RTT. $S_2$ gets less.
The RTT Bias of TCP Reno

With TCP Reno, two competing sources with different RTTs are not treated equally

- source with large RTT obtains less

A source that uses *many hops* obtains less rate because of two combined factors, one is good, the other is bad:

1. this source uses more resources. The mechanics of *proportional fairness* leads to this source having less rate – this is desirable in view of the theory of fairness.

2. this source has a *larger RTT*. The mechanics of additive increase leads to this source having less rate – this is an undesired bias in the design of TCP Reno

Cause is: additive increase is one packet per RTT (instead of one packet per constant time interval)
TCP Reno

Loss - Throughput Formula

Consider a *large* TCP connection (many bytes to transmit)

Assume we observe that, in average, a fraction $q$ of packets is lost (or marked with ECN)

The throughput should be close to $\theta = \frac{MSS \times 1.22}{RTT \sqrt{q}}$

Formula assumes: transmission time negligible compared to RTT, losses are rare, time spent in Slow Start and Fast Recovery negligible, losses occur periodically
Guess the ratio between the throughputs $\theta_1$ and $\theta_2$ of $S_1$ and $S_2$

$\theta_1 = \frac{3}{7} \theta_2$

$\theta_1 = \theta_2$

$\theta_1 = \frac{7}{3} \theta_2$

$\theta_1 = \frac{10}{3} \theta_2$

E. None of the above

F. I don’t know
Solution: Guess the ratio between the throughputs $\theta_1$ and $\theta_2$ and of $S_1$ and $S_2$.

If processing time is negligible and router drops packets in the same proportion for all flows, then throughput is proportional to $1/\text{RTT}$, thus

$$\frac{\theta_1}{\tau_1} = \frac{\theta_2}{\tau_2} \quad \text{i.e.} \quad \theta_1 = \frac{7}{3} \theta_2$$
TCP Reno serves as the reference for congestion control in the Internet as it was the first mature implementation of congestion control.

TCP Reno has a number of shortcomings. Can you cite a few?
Solution

RTT bias – not nice for users in New Zealand
Periodic losses must occur, not nice for application (e.g. video streaming).
TCP controls the window, not the rate. Large bursts typically occur when packets are released by host following e.g. a window increase – not nice for queues in the internet, makes non-smooth behaviour.
Self inflicted delay: if network buffers (in routers and switches) are large, TCP first fills buffers before adapting the rate. The RTT is increased unnecessarily. Buffers are constantly full, which reduces their usefulness (bufferbloat) and increases delay for all users.
Interactive, short flows see large latency when buffers are large and full.
Long Fat Networks (LFNs)

In an LFN, additive increase is too slow

(slide from Presentation: "Congestion Control on High-Speed Networks", Injong Rhee, Lisong Xu, Slide 7) the figure assumes congestion avoidance implements a strict additive increase, losses are detected by fast retransmit and ignores the “fast recovery” phase. MSS = 12500B (jumbo packets), RTT = 100 msec
TCP Cubic modifies Congestion Control

Why? increase TCP rate fast on LFNs
How? TCP Cubic keeps the same slow start, congestion avoidance, fast recovery phases as TCP Reno, but:
- Multipliciative Decrease is $\times 0.7$ (instead of $\times 0.5$)
- During congestion avoidance, the increase is not additive but cubic

Say congestion avoidance is entered at time $t_0 = 0$ and let $W_{max} =$ value of cwnd when loss is detected.

Let $W(t) = W_{max} + 0.4(t - K)^3$

with $K$ such that $W(0) = 0.7 W_{max}$

Then the window increases like $W(t)$ until a loss occurs again.

Units are: data = 1MSS; time = 1s
Cubic versus Reno

Cubic increases window in concave way until reaches $W_{max}$ then increases in a convex way.

Cubic’s window function is independent of RTT; is slower than Reno when RTT is small, larger when RTT is large.

Additive Increase ($\approx$ Reno) with RTT = 0.1 s

Cubic Additive Increase ($\approx$ Reno) with RTT = 1 s
The Cubic Window Increase

Cubic makes sure it is at least as fast as additive increase with an additive increase term \( r_{cubic} \) (discussed later):

\[
W_{AIMD}(t) = W(0) + r_{cubic} \frac{t}{RTT}
\]

if \( W(t) < W_{AIMD}(t) \) then Cubic replaces \( W(t) \) by \( W_{AIMD}(t) \)

\( \Rightarrow \) Cubic’s window \( \geq \) AIMD’s window

\( \Rightarrow \) When RTT or bandwidth-delay product is small, Cubic does the same as a modified Reno with additive increase \( r_{cubic} \) MSS per RTT (instead of 1) and multiplicative decrease \( \beta_{cubic} = 0.7 \).

\( r_{cubic} \) is computed such that this modified Reno has the same loss-throughput formula as standard Reno \( \Rightarrow \)

\[
r_{cubic} = 3 \frac{1-\beta_{cubic}}{1+\beta_{cubic}} = 0.529
\]

\( \Rightarrow \) Cubic’s throughput \( \geq \) Reno’ throughput with equality when RTT or bandwidth-delay product is small
Cubic’s Other Bells and Whistles

Cubic’s Loss throughput formula

\[ \theta \approx \max \left( \frac{1.054}{RTT^{0.25}q^{0.75}}, \frac{1.22}{RTT \sqrt{q}} \right) \]

in MSS per second.

Cubic’s formula is same as Reno for small RTTs and small BW-delay products.

Other Cubic details

\[ W_{max} \] computation uses a more complex mechanism called “fast convergence”

see Latest IETF Cubic RFC / Internet Draft

Using loss as a congestion indication has major drawback: losses to application + bufferbloat.


The previous figure illustrates that if the amount of inflight data is just large enough to fill the available bottleneck link capacity, the bottleneck link is fully utilized and the queuing delay is still zero or close to zero. This is the optimal operating point (A), because the bottleneck link is already fully utilized at this point. If the amount of inflight data is increased any further, the bottleneck buffer gets filled with the excess data. The delivery rate, however, does not increase anymore. The data is not delivered any faster since the bottleneck does not serve packets any faster and the throughput stays the same for the sender: the amount of inflight data is larger, but the round-trip time increases by the corresponding amount. Excess data in the buffer is useless for throughput gain and a queuing delay is caused that rises with an increasing amount of inflight data. Loss-based congestion controls shift the point of operation to (B) which implies an unnecessary high end-to-end delay, leading to “bufferbloat” in case the buffer sizes are large.
Explicit Congestion Notification (ECN) aims at avoiding these problems

**What?** signal congestion without dropping packets (= DECbit)

**How?** router marks packet instead of dropping
TCP destination echoes the mark back to source
At the source, TCP interprets a marked packet in the same way as if there would be a loss detected by fast retransmit
Explicit Congestion Notification (ECN)

TCP IP
payload header header

S reduces window by 1/2
1. S sends a packet using TCP
2. Packet is received at congested router buffer; router marks the Congestion Experienced (CE) bit in IP header
3. Receiver sees CE in received packet and set the ECN Echo (ECE) flag in the TCP header of packets sent in the reverse direction
4. 5,6 Packets with ECE is received by source.
7. Source applies multiplicative decrease of the congestion window.

Source sets the Congestion Window Reduced (CWR) flag in TCP header. The receiver continues to set the ECE flag until it receives a packet with CWR set.

Multiplicative decrease is applied only once per window of data (typically, multiple packets are received with ECE set inside one window of data).
Put correct labels

Assume TCP with ECN is used and there is no packet loss.

CA: congestion avoidance
SS: slow start = multiplicative increase
MD: multiplicative decrease

A. 1 = CA, 2 = SS
B. 1 = SS, 2 = MD
C. 1 = CA, 2 = MD
D. I don’t know
Solution

The TCP congestion window when ECN is used and no packet loss occurs

multiplicative decrease
= reduction of target window by \( \frac{1}{2} \)
ECN Flags

2 bits in IP header (direct path) code for 4 possible codepoints
non ECN Capable (non ECT)
ECN capable and no congestion  ECT(0) and ECT(1)
ECN capable and congestion experienced (CE)

3 bits in TCP header (return path)
ECE (ECN echo) bit  = \( y_i = 0 \) or 1
CWR = ack of ECE=1 received (window reduced)
ECE = 1 is set by R until R receives a TCP segment with CWR=1
NS = nonce, used for S to check that ECN is taken seriously.
Nonce Sum (RFC 3540)

Why invented?
S uses ECN ⇒ routers do not drop packets, use ECN instead but ECN could be prevented by: R not implementing ECN, NATs that drop ECN info in header

In such cases, the flow of S is not congestion controlled; this should be detected → revert to non ECN

What does it do?
Nonce Sum bit in TCP header (NS) is used by S to verify that ECN works for this TCP
**Nonce Sum**

*(RFC 3540)*

How does it work?

Source S randomly chooses ECT(0) or ECT(1) in IP header and verifies that the received NS bit is compatible with the ECT(0)/ECT(1) chosen by S

- Non congested router does nothing; congested router sees ECT(0) or ECT(1) and marks packet as CE instead of dropping it
- R echoes back to S the xor of all ECT bits in NS field of TCP header;
- If R does not take ECN seriously, NS does not correspond and S detects it; S detects that ECN does not work; Malicious R cannot compute NS correctly because CE packets do not carry ECT bit
- If router or NAT drops ECN bits then R cannot compute the correct NS bit and S detects that ECN does not work
Sender

Receiver

initial sum = 1

-- 1:4 ECT(0) --> NS = 1 + 0(1:4) = 1(4)

<- ACK 4, NS=1 ---

-- 4:8 ECT(1) --> NS = 1(4) + 1(4:8) = 0(8)

<- ACK 8, NS=0 ---

-- 8:12 ECT(1) --> NS = 0(8) + 1(8:12) = 1(12)

<- ACK 12, NS=1 --

-- 12:16 ECT(1) --> NS = 1(12) + 1(12:16) = 0(16)

<- ACK 16, NS=0 --

Figure 1: The calculation of nonce sums at the receiver.

Sender

Receiver

initial sum = 1

-- 1:4 ECT(0) --> NS = 1 + 0(1:4) = 1(4)

<- ACK 4, NS=1 --

-- 4:8 ECT(1) --> CE --> NS = 1(4) + ?(4:8) = 1(8)

<- ACK 8, ECE NS=1 --

-- 8:12 ECT(1), CWR --> NS = 1(8) + 1(8:12) = 0(12)

<- ACK 12, NS=0 --

-- 12:16 ECT(1) --> NS = 0(12) + 1(12:16) = 1(16)

<- ACK 16, NS=1 --

Figure 2: The calculation of nonce sums at the receiver when a packet (4:8) is marked. The receiver may calculate the wrong nonce sum when the original nonce information is lost after a packet is marked.
RED (Random Early Detection)

Why ? when to mark a packet with ECN
How ? queue estimates its average queue length

\[ \text{avg} \leftarrow a \times \text{measured} + (1 - a) \times \text{avg} \]

incoming packet is marked with probability given by RED curve

- a uniformization procedure is also applied to prevent bursts of marking
Active Queue Management

RED can also be applied even if ECN is not supported
In such a case, a packet is dropped with probability computed by the RED curve
packet may be discarded even if there is some space available!

Expected benefit

» avoid constantly full buffers
» avoid irregular drop patterns

This is called Active Queue Management
» as opposed to passive queue management = drop a packet when queue is full = “Tail Drop”
In a network where all flows use TCP with ECN and all routers support ECN, we expect that ...

A. there is no packet loss
B. there is no packet loss due to congestion
C. there is no packet loss due to congestion in routers
D. none of the above
E. I don’t know
Solution

Answer C

We expect that routers do not drop packets due to congestion if all TCP sources use ECN

However there might be congestion losses in bridges, and there might be non-congestion losses (transmission errors)
9. Other Cool Stuff

Data Center TCP
Per Class Queuing
TCP-friendly apps
TCP-BBR
Data Centers and TCP

What is a data center?
- a room with lots of racks of PCs and switches
- youtube, CFF.ch, switchdrive, etc

What is special about data centers?
- most traffic is TCP
- very small latencies (10-100 μs)
- lots of bandwidth, lots of traffic
- internal traffic (distributed computing) and external (user requests and their responses)
- many short flows with low latency required (user queries, mapReduce communication)
- some jumbo flows with huge volume (backup, synchronizations) may use an entire link
What is your preferred combination for TCP flows *inside* a data center?

A. TCP Reno, no ECN no RED
B. TCP Reno and ECN
C. TCP Cubic, no ECN no RED
D. TCP Cubic and ECN
E. I don’t know
Solution

Answers B or D

Without ECN there will be bufferbloat, which means high latency for short flows.

Cubic has better performance than Reno when bandwidth-delay product is large, which is typically the case when latencies are high, which is only marginally true in data centers. So Cubic or Reno make little difference as long as ECN is used.

Standard operation of ECN (e.g. with Reno or Cubic) still has drawbacks for jumbo flows in data center settings:

- multiplicative decrease by 50% is too abrupt $\Rightarrow$ throughput inefficiency
Data Center TCP

Why ? Improve performance for jumbo flows when ECN is used. Avoid the brutal multiplicative decrease by 50%

How ?

► TCP source estimates proba of congestion $p$

► Multiplicative decrease is $\times \beta_{DCTCP} = \left(1 - \frac{p}{2}\right)$

► ECN echo is modified so that the proportion of CE marked Acks $\approx$ the probability of congestion
In a data center: two large TCP flows compete for a bottleneck link; one uses DCTPC, the other uses Cubic/ECN. Both have same RTT.

A. Both get roughly the same throughput
B. DCTCP gets much more throughput
C. Cubic gets much more throughput
D. I don’t know
Solution

Answer B.

Since latency is very small, Cubic with ECN is same as Reno with ECN and is essentially AIMD with multiplicative decrease $= \times 0.5$.

DCTCP is similar but with multiplicative decrease $= \times (1 - p)$ so the multiplicative decrease is always less. DCTCP decreases less and increases the same, therefore it is more aggressive.

In other words, DCTCP competes unfairly with other TCPs; it cannot be deployed outside data centers (or other controlled environments). Inside data centers, care must be given to separate the DCTCP flows (i.e. the internal flows) from other flows. This can be done with class based queuing (see later).
Class Based Queuing

In general, all flows compete in the Internet using the congestion control method of TCP. In controlled environments (e.g. a data center, a smart grid, a TV distribution network, a cellular network) it is possible to modify the competition by using per-flow or class-based queuing.

E.g. class based queuing

- routers classify packets (using an access list)
- each class is guaranteed a rate
- classes may exceed the guaranteed rate by borrowing from other classes if there is spare capacity

This implemented in routers with dedicated queues for every class and a scheduler (such as Deficit Round Robin) that performs weighted fair queuing.
Example of Class-Based Queuing

Class 1 is guaranteed a rate of 2.5 Mb/s; can exceed this rate by borrowing capacity available from the total 10 Mb/s if class 2 does not need it. Class 2 is guaranteed a rate of 7.5 Mb/s; can exceed this rate by borrowing capacity available from the total 10 Mb/s if class 1 does not need it.
Which rates will PC1 and PC2 achieve?

A. 5 Mb/s each
B. 4 Mb/s each
C. PC1: 5 Mb/s, PC2: 3 Mb/s
D. I don’t know
Answer B

PC1 and PC2 see this network ↑

Since PMU1 and PMU2 stream at 1 Mb/s and class 2 may borrow, the available capacities for class 2 are 9 Mb/s, 8 Mb/s and 8 Mb/s.

TCP allocates rates \( x_1 \) and \( x_2 \) so as to maximize \( U(x_1) + U(x_2) \) where \( U \) is the utility function of TCP; the function \( U \) is the same for PC1 and PC2 because RTTs are the same.

The constraints are \( x_1 \leq 9 \text{ Mb/s}, \ x_1 + x_2 \leq 8 \text{ Mb/s}, \ x_1 + x_2 \leq 8 \text{ Mb/s} \)

Thus TCP solves the problem:

maximize \( U(x_1) + U(x_2) \) subject to \( x_1 + x_2 \leq 8 \text{ Mb/s} \)

By symmetry, \( x_1 = x_2 = 4 \text{ Mb/s} \)

You can also check max-min fair allocation \( (x_1 = x_2 = 4 \text{ Mb/s}) \) and proportionally fair allocation \( (x_1 = x_2 = 4 \text{ Mb/s}) \) .
TCP Friendly UDP Applications

Some UDP applications that can adapt their rate (e.g. VBR video) have to implement congestion control; one method is to use the congestion control module of TCP: e.g. QUIC’s original version, which is over UDP, uses Cubic’s congestion control.

Another method is to rate-control by computing the rate that TCP Reno would obtain. E.g.: TFRC (TCP-Friendly Rate Control) protocol

► application determines the sending rate (hence coding rate for audio and video)
► feedback is received in form of count of lost packets
► sending rate is set to: rate of a TCP flow experiencing the same loss ratio, using the loss throughput formula
TCP-BBR
Bottleneck Bandwidth and RTT

What? Avoid bufferbloat in TCP without ECN

How? TCP source controls rate (not window), estimates the rate and RTT by periodically overshooting/undershooting. Losses are ignored.

http://blog.cerowrt.org/post/bbrs_basic_beauty/
BBR Operation

- Source views network as a single link (the bottleneck link)
- Estimates RTT by taking the min over the last 10 sec
- Estimates bottleneck rate (bandwidth) $b_r = \max$ of delivery rate over last 10 RTTs; delivery rate = amount of acked data per $\Delta t$
- Send data at rate $b_r \times c(t)$

Where $c(t) = 1.25; 0.75,; 1; 1; 1; 1; 1; 1$ i.e. $c(t)$ is 1.25 during one RTT, then 0.75 during one RTT, then 1 during 6 RTTs (“probe bandwidth” followed by “drain excess” followed by steady state)

Data is paced using a spacer at the source

- Max data in flight is limited to $2 \times b_r \times RTT_{est}$ and by the offered window
- There is also a special startup phase with exponential increase of rate
BBR Operation

BBR TCP takes no feedback from network -- no reaction to loss or ECN

Claims: avoids filling buffers because it estimates the bottleneck bandwidth

[Hock et al, 2017] find that it does not work because the estimated bottleneck bandwidth ignores how many flows are competing

bufferbloat may still exist

sustained huge loss rates may exist

fairness issues may exist inside BBR and versus other TCPs
Conclusion

Congestion control is in TCP or in a TCP-friendly UDP application.

Standard TCP uses the window to control the amount of traffic: additive increase or cubic (as long as no loss); multiplicative decrease (following a loss).

Standard TCP uses loss as congestion signal.

Too much buffer is as bad as too little buffer – bufferbloat provokes large latency for interactive flows.

ECN can be used to avoid bufferbloat – it replaces loss by an explicit congestion signal; not widely deployed in the internet, but is in Data Center TCP.

Class based queuing is used to separate flows in enterprise networks.