Congestion and Flow Control in TCP

Flow Control and Congestion Control

Flow control

Sender avoids overflow of receiver buffer

Congestion control

All senders avoid overflow of intermediate network buffers

Buffer fill rate

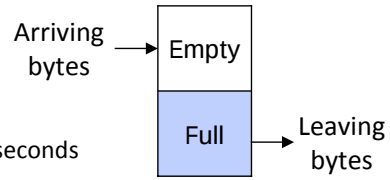
Bytes / second arriving from network

Buffer empty rate

Bytes / second leaving to network or application layer

Buffer file time

$$T_{overflow} = \frac{\text{buffer size}}{\text{buffer fill rate} - \text{buffer empty rate}}$$



Example

$$T_{overflow} = \frac{64\text{ KB}}{8\text{ KB/sec} - 4\text{ KB/sec}} = \frac{64\text{ KB}}{4\text{ KB/sec}} = 16\text{ seconds}$$

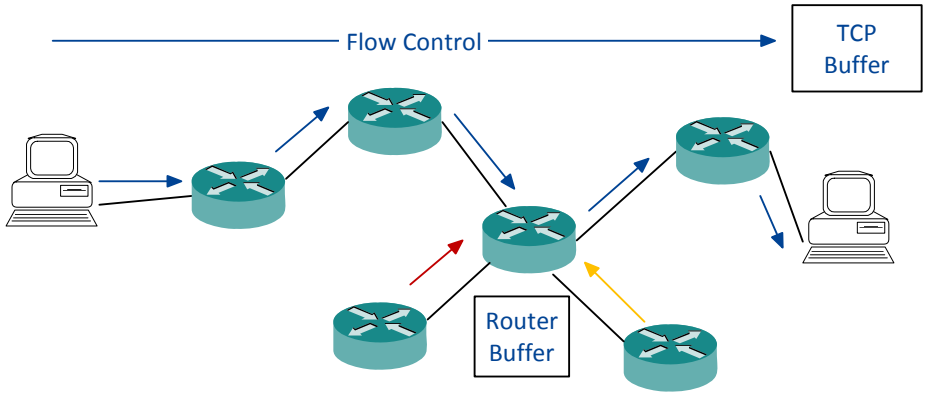
Congestion Control

Flow control

Avoid overflow in TCP receiver buffer

Congestion control

Avoid overflow in router buffers



Queuing Theory

Assumptions

Segments arrive independently (Poisson statistics)

Random length (bytes)

Average arrival rate in steady state — Packets/sec, Bytes/sec, or bps

Segments leave independently (Poisson statistics)

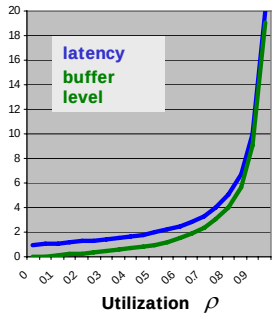
Average emptying rate in steady state — Packets/sec, Bytes/sec, or bps

Results

$$\rho = \text{Utilization} = \frac{\text{arrival rate}}{\text{empty rate}}$$

$$\text{Latency} = \frac{1}{\text{empty rate} - \text{arrival rate}} = \frac{1}{\text{empty rate}}\left(\frac{1}{1-\rho}\right)$$

$$\text{Buffer Level} = \text{Latency} \times \text{arrival rate} = \frac{\rho}{1-\rho}$$



Example

Arrival rate = 90 packets / sec Empty rate = 100 packets

Utilization = (90 packets / sec) / (100 packets / sec) = 0.9 = 90%

Buffer level = 0.9 / (1 – 0.9) = 9 packets in buffer

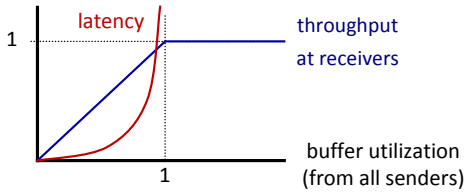
Latency = 1 / (100 packets / sec) × 9 packets in buffer = (9 / 100) sec = 0.09 sec

Buffer Throughput

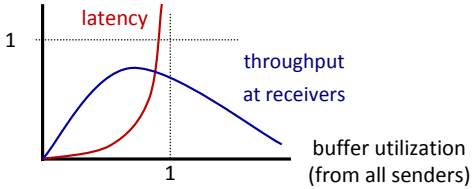
(Over)-simplified throughput model

$$\text{throughtput} = \frac{\text{receive rate}}{\text{maximum receive rate}}$$

$$\text{buffer utilization} = \frac{\text{arrival rate}}{\text{empty rate}}$$



$$\text{goodput} = \frac{\text{receive rate (error-free in-order)}}{\text{maximum receive rate}}$$



Realistic throughput behavior

- High arrival rate at buffer
- Longer latency + overflow
- Sender timeouts
- Re-transmit ⇒ more segments ⇒ higher arrival rate at buffer

TCP Flow Control

Source window

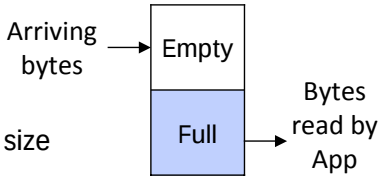
- Initial source window = maximum number of "unACKed" bytes
- Determined by congestion + flow control

Destination window

- Number of bytes receiver can accept
- Determined by available space in receiver buffer
- Buffer level = Previous level + arriving bytes – bytes read by App
- Application reads too slowly ⇒ decrease destination window

Sliding window

- Window field in TCP header
- Number of bytes receiver will accept
- Receiver discards bytes above window size



Flow Control Example

Source Window	In Flight		Buffer Level	Dest Window
64 KB	—		0	8 KB
64 KB	2 KB	2 KB →	0	8 KB
64 KB	4 KB	2 KB →	2 KB	6 KB
64 KB	4 KB		4 KB	4 KB
4 KB	0	← ACK 4 KB (2+2 = 4) window = 4 KB		
2 KB	2 KB	2 KB →		
			6 KB	2 KB
			App reads 4 KB	
			2 KB	6 KB
6 KB	0 KB	← ACK 6 KB (2+4 = 6) window = 6 KB		
0	6 KB	6 KB →		
			8 KB	0
0	0	← ACK 12 KB (6+6 = 12) window = 0 KB		
		error ← ACK 12 KB (6+6 = 12) window = 4 KB	App reads 4 KB	
Persist Timeout		1 B →		
			4 KB	4 KB
4 KB	0	← ACK 12 KB + 1B window = 4 KB		

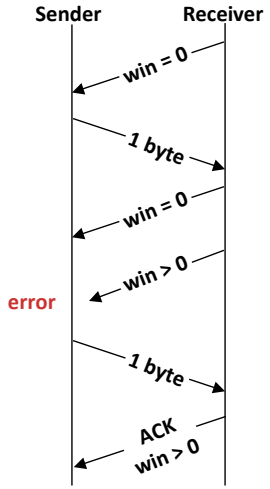
Receive Window Bugs — 1

Bug — deadlock

- Receiver advertises window = 0
- Window update with window > 0 is lost → deadlock

Fix — persist timeout

- Sender attempts small segment
- ACK contains new window size



Receive Window Bugs — 2

Silly Window Problem

Application reads received data slowly

Receiver advertises small window

Data bytes ~ header bytes

More segments / file transfer ⇒ larger total traffic (data + headers)

Nagle Algorithm — bug fix for Silly Window

Sender accumulates application data — sends large segments

Works badly with Telnet (requires small segments)

Receiver side bug fix

Receiver keeps 0 window size until it can advertise large window

TCP Congestion Control

End-to-end congestion control

Based on host estimates

No feedback from intermediate network nodes

Slow-start

Begin session with low transmission rate

Increase rate until timeouts begin

Fast retransmit

Do not wait for timeout

Re-transmit after duplicate ACKs (dupACKs)

Congestion avoidance

Limit transmission rate after duplicate ACKs

Growth rate of transmission rate slows

Slow-Start

Congestion window (cwnd)

Source window

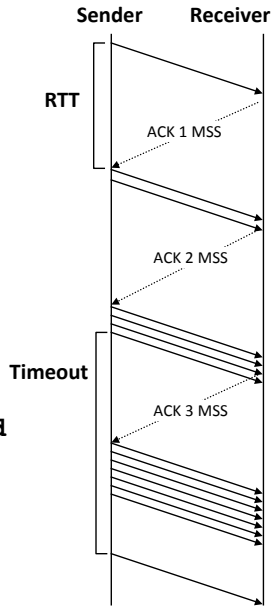
Maximum number of "unACKed" bytes

Initial **cwnd** = 1 MSS (maximum segment size)

Data rate = 1 MSS / RTT

RTT = round trip time = time from send to ACK

Maximum **cwnd** = destination window

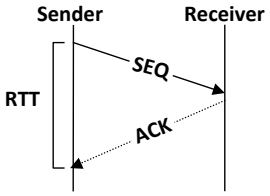


Exponential growth

```
On (ACK)
    cwnd ← cwnd + size of data ACKed
    if (cwnd > maximum cwnd)
        cwnd ← max cwnd
On (ACK timeout)
    cwnd ← initial cwnd = 1 MSS
```


Computing TCP's Retransmission Timer — RFC 2988

Initialize



RTO ← 3 seconds

G ← clock granularity (typically 500 ms)

R ← first **RTT** measurement (round trip time)

SRTT ← **R**

RTTVAR ← **R/2**

RTO ← **max(1 sec, SRTT + max(G, 4 * RTTVAR))**

Update after measurements R'

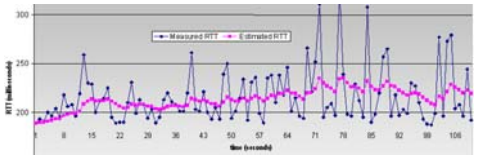
RTTVAR ← **(1 - β) * RTTVAR + β * |SRTT - R'|**

SRTT ← **(1 - α) * SRTT + α * R'**

RTO ← **max(1 sec, SRTT + max (G, 4 * RTTVAR))**

α = 1/8

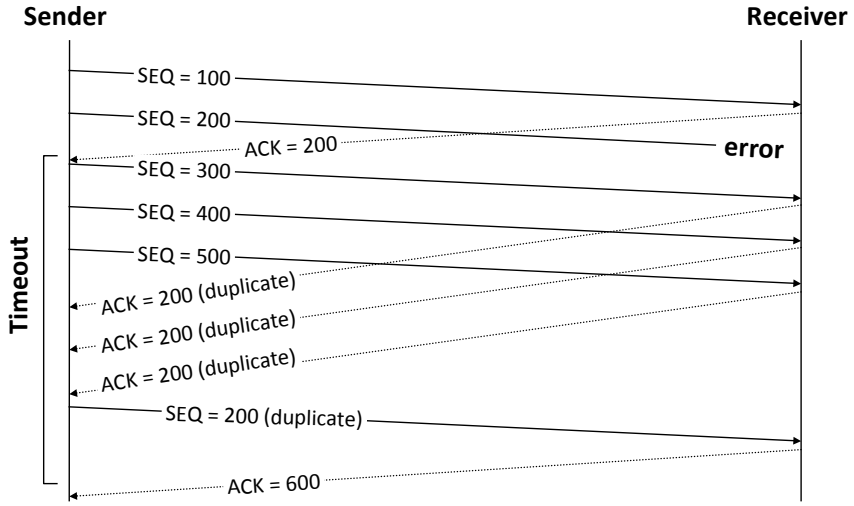
β = 1/4



Fast Retransmit

Better performance with RTO >> RTT

3 duplicate ACKs (dupACKs) for segment ⇒ re-send segment



Congestion Avoidance

Reno protocol

Slow start phase

```
On (ACK && cwnd < ssthresh)
    cwnd ← cwnd + size of data ACKed
On (ACK timeout)
    ssthresh ← cwnd
    cwnd ← initial cwnd = 1 MSS
    RTO ← 2 * RTO
```

Congestion avoidance phase

```
On (ACK && cwnd > ssthresh)
    cwnd ← cwnd + 1 MSS
```

Fast retransmit with fast recovery

```
On (3 dupACKs)
    ssthresh ← cwnd / 2
    cwnd ← ssthresh + 3
    retransmit, wait 1 RTT → continue
```

If dupACKs > 3
cwnd++ on each dupACK

TCP Sender with Reno — 1

```
// initialize
    SEQ = ISN + 1
    SendBase = ISN + 1
    InFlight = 0
    cwnd = 1 MSS
    Set ssthreshold large (local policy)
    RTO = timeout interval

on (new data from application)
    Prepare data segment:sequence number = SEQ
    if InFlight < min{cwnd,SendWindow,RecvWindow)
        Pass segment to IP
        SEQ = SEQ + length(data)
        InFlight = InFlight + length(data)
        if !(timer running) timer = RTO
```


TCP Sender with Reno — 2

```
if (receive ACK = y)
    stop timer
    if (y > SendBase)
        dupACK = 0
        newACKs = y − SendBase            // bytes ACKed
        SendBase = y
        InFlight = InFlight − newACKs
        if (cwnd < ssthresh) cwnd = cwnd + newACKs
        else cwnd = cwnd + 1 MSS
        if (InFlight > 0) timer = RTO
```

TCP Sender with Reno — 3

```
    // if (y > SendBase)
    else
        dupACK++
        if (dupACK = 3)
            SEQ = SendBase = min{unACKed SEQ} and retransmit
            timer = RTO
            ssthresh = cwnd / 2
            cwnd = ssthresh + 3
            wait 1 RTT    // wait for ACK of resent packet
        if (dupACK > 3)
            cwnd = cwnd + 1 MSS and resend again
if (timeout)
        SEQ = SendBase = min{unACKed SEQ} and retransmit
        ssthresh = cwnd
        cwnd = initial cwnd = 1 MSS
        RTO = 2 * RTO
        timer = RTO
```


TCP Receiver with Reno — 1

```
// initialize
    Set RecvWindow = receiver buffer size
    expected = Sender ISN + 1
    ack_buffer = 0
    ack_max (local policy: delayed ACK trigger)
    ack_delay = 250 msec (local policy: < 500 msec)
    Start ACK delay timer = ack_delay

if (ACK delay timer = 0 && ack_buffer > 0)
    Send ACK = expected with updated RecvWindow
    ACK delay timer = ack_delay
    ack_buffer = 0
```


TCP Receiver with Reno — 2

```
if (receive SEQ = x)
    if (x = expected && error-free)
        expected = expected + length(data)
        if (NACK = 1)
            Send ACK = expected with updated RecvWindow
            ACK delay timer = ack_delay
            ack_buffer = 0
            NACK = 0
        else if (ack_buffer < ack_max)
            nextACK = expected
            ack_buffer++
        else if (ack_buffer = ack_max)
            Send ACK = expected with updated RecvWindow
            ACK delay timer = ack_delay
            ack_buffer = 0
    else
        Send ACK = expected with updated RecvWindow
        ACK delay timer = ack_delay
        NACK = 1
```


Reno Example — 1

Counting in bytes: SEQ = SEQm × (1 MSS) ACK = ACKm × (1 MSS)

ssthresh = 32 Receiver sends ACK for every 4 packets (or ACK delay)

Sender				Receiver	
cwnd	in-flight	SEQm Sent	Packets Sent	ACKm Sent	New ACKs
1	1	1	1 (1)		
				2	1
1+1 = 2	0				
2	2	2	2 – 3 (2)		
				4	2
2+2 = 4	0				
4	4	4	4 – 7 (4)		
				8	4
4+4 = 8	0				
8	8	8	8 – 15 (8)		
				12	4
8+4 = 12	4				

Reno Example — 2

cwnd	in-flight	SEQm Sent	Packets Sent	ACKm Sent	New ACKs
12	4				
				16	4
12+4 = 16	0				
16	16	16	16 – 31 (16)		
				20	4
16+4 = 20	12				
20	20	32	32 – 39 (8)		
				24	4
20+4 = 24	16				
24	24	40	40 – 47 (8)		
				28	4
24+4 = 28	20				
				32	4
28+4 = 32	16				
32	32	48	48 – 63 (16)		

Reno Example — 3

cwnd	in-flight	SEQm Sent	Packets Sent	ACKm Sent	New ACKs
32	32	48	48 – 63 (16)		
				36	4
32+1 = 33	28				
33	32	64	64 – 67 (4)		
				40	4
33+1 = 34	28				
				44	4
34+1 = 35	24				
				48	4
35+1=36	20				
				52	4
36+1 = 37	16				
				56	4
37+1 = 38	12				

Reno Example — 4

cwnd	in-flight	SEQm Sent	Packets Sent	ACKm Sent	New ACKs
38	20	68	68 – 75 (8)		
				60	4
38+1 = 39	16				
				60	0
39	16				
				60	0
39	16				
		3rd dupACK		60	0
ssthresh ← 19 19 + 3 = 22	16				
22	16	60	60	Retransmit 1 packet	
				76 (cumulative)	16
23	0				
23	23	76	76 – 98 (23)		

Reno: First Alternative Scenario

cwnd	in-flight	SEQm Sent	Packets Sent	ACKm Sent	New ACKs
1	1	1	1 (1)		
				2	1
1+1 = 2	0				
2	2	2	2 – 3 (2)		
				3	1
2+1 = 3	1				
3	3	4	4 – 5 (2)		
				3	0
3+0 = 3	3				
				3	0
cwnd = in-flight ⇒ no more packets ⇒ no 3 dupACKs ⇒ Timeout					
1	0				
1	1	3	3 (1)		
				4	1
1+1 = 2	0				

Reno: Second Alternative Scenario – 1

cwnd	in-flight	SEQm Sent	Packets Sent	ACKm Sent	New ACKs
1	1	1	1 (1)		
				2	1
1+1 = 2	0				
2	2	2	2 – 3 (2)		
				4	2
2+2 = 4	0				
4	4	4	4 – 7 (4)		
				7	3
4+3 = 7	1				
7	7	8	8 – 13 (6)	7 packets can cause 7 dupACKs	
				7	0
				7	0
			3rd dupACK	7	0
retransmit	7	7	7 (1)		
ssthresh ← 3 3 + 3 = 6	8				

Reno: Second Alternative Scenario – 2

cwnd	in-flight	SEQm Sent	Packets Sent	ACKm Sent	New ACKs
			4th dupACK	7	0
7	8				
			5th dupACK	7	0
8	8				
			6th dupACK	7	0
9	8				
9	9	7	7 (1)		
				14 (cumulative)	7
9+1 = 10	0				
10	14	10	14 – 23 (10)		

Selective Acknowledgment Option

Selective ACK (SACK)

- Permits ACK for segments with gaps
- Option negotiated between hosts
- Defined in RFC 2018

Example

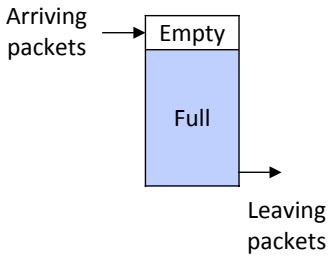
- Last ACK = 5000
- Send 8 segments × 500 data bytes / segment
- Case 1
  - First 4 segments received and last 4 dropped
  - Receiver returns normal ACK = 5000 + 4 * 500 = 7000
  - No SACK option field
- Case 2
  - First segment lost and 7 segments received
  - For each segment receiver returns segment with
    - ACK = 5000
    - SACK option field with start + end ACK

Data	ACK	Option Field	
		Start	End
5000	—	—	—
5500	5000	5500	5999
6000	5000	5500	6499
6500	5000	5500	6999
7000	5000	5500	7499
7500	5000	5500	7999
8000	5000	5500	8499
8500	5000	5500	8999

Active Queue Management (AQM)

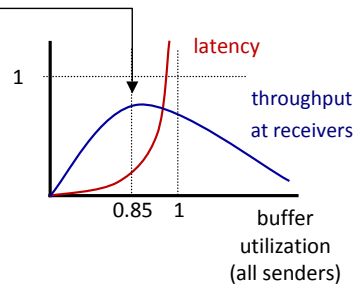
Standard Queue

- At receiver
  - Full buffer → drop excess packets
- At sender
  - No ACK → timeout → signal congestion



Random Early Detection (RED)

- Router
  - Detects congestion early
  - Drops random packets
- Sender
  - Sees dupACKs or timeout
  - Assumes congestion
  - Lowers **`cwnd`**



RED Algorithm

Algorithm

```
for each packet arrival
    calculate avg = average queue size
    if min_th ≤ avg < max_th
        calculate probability p_a
        with probability p_a:
            mark arriving packet for drop
    else if max_th ≤ avg
        mark arriving packet for drop
```

Parameters

max_p = maximum mark probability (0.1 to 0.5)

min_{th} ~ 5

max_{th} ~ 30

$p_b \leftarrow max_p (avg - min_{th}) / (max_{th} - min_{th})$

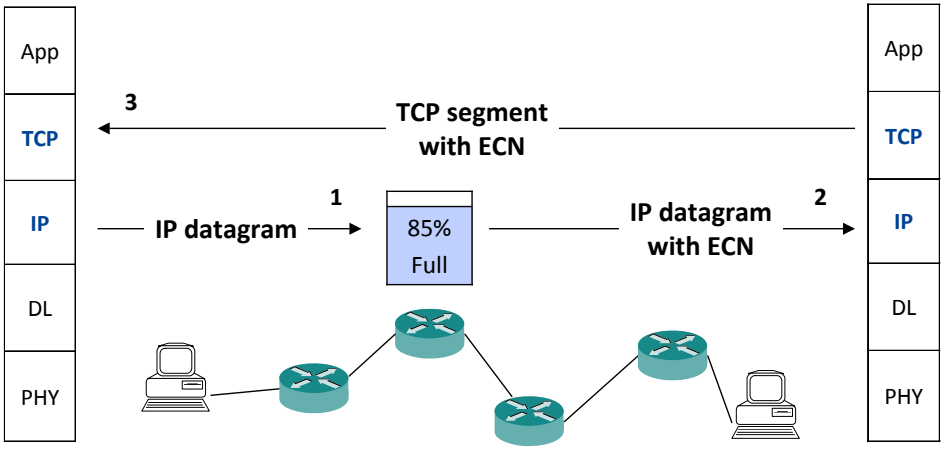
$p_a \leftarrow p_b / (1 - count \times p_b)$

count = number of consecutive dropped packets

AQM with ECN

Explicit Congestion Notification (RFC 3168)

1. IP router predicts congestion — RED with mark (no drop)
2. IP router indicates congestion to receiver in IP header
3. Receiver indicates congestion to sender in TCP ACK header



Explicit Congestion Notification (ECN)

IP datagram

4 bits	4 bits	6 bits	2 bits	16 bits
Version	Hlen	**DSCP**	**ECN**	Total Length (header + data in bytes)
Identification			Flags	Fragment Offset (13 bits)
Time to Live		Protocol		Header Checksum
Source IP Address				
Destination IP Address				
Options				
Data				

Differentiated Services Code Point (DSCP)

QoS requirements

Explicit Congestion Notification (ECN)

00	Not ECN capable	For retransmissions
01	ECT(0) – ECN Capable Transport (0)	To allow protocol error checking
10	ECT(1) – ECN Capable Transport (1)	
11	CE (Congestion Experienced)	

Explicit Congestion Notification (ECN)

TCP header flags

32 bits			
source port			destination port
sequence number (SEQ)			
acknowledgement number (ACK)			
HLEN	not used	**flags**	window size
checksum			urgent pointer
Options			

NS	ECN-nonce concealment protection
CWR	Congestion Window Reduced (CWR) flag
ECE	ECN-Echo

URG	Urgent pointer
ACK	**Acknowledgment**
PSH	Push buffer
RST	**Reset**
SYN	**Synchronize**
FIN	**No more data**

ECN Negotiation

TCP client

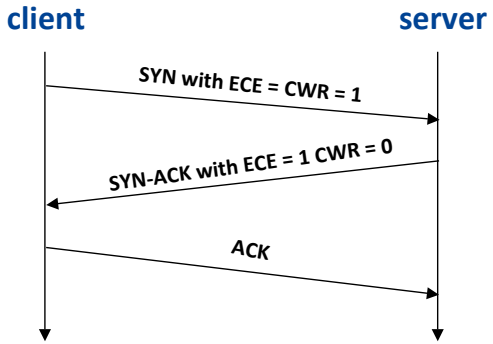
SYN ECE = CWR = 1 in SYN

TCP server

ECE = 1 in SYN-ACK

IP

ECT(0) , ECT(1) in SYN and SYN-ACK

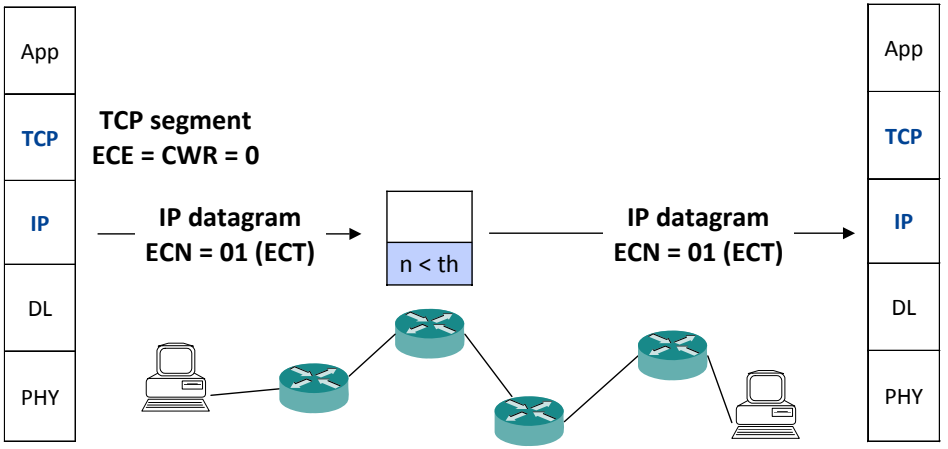


ECN Operation — 1

No congestion

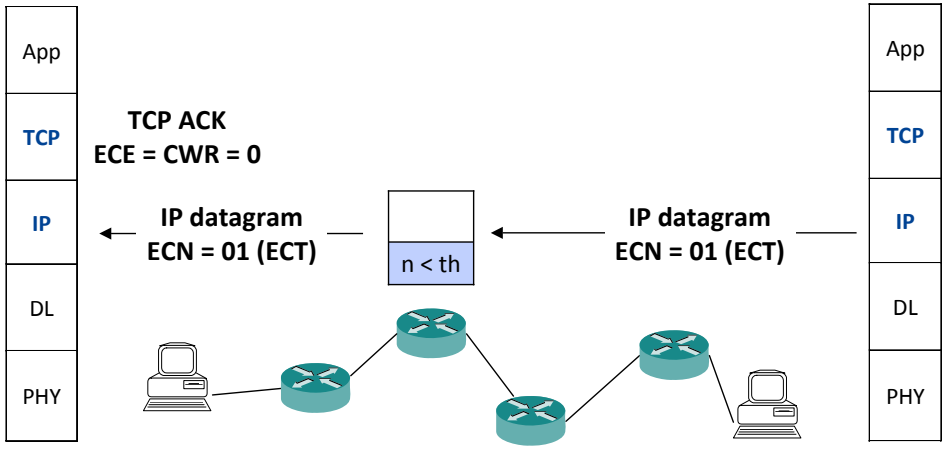
Measure long term average buffer level **n**

Compare with threshold level **th**



ECN Operation — 2

No congestion



ECN Operation — 3

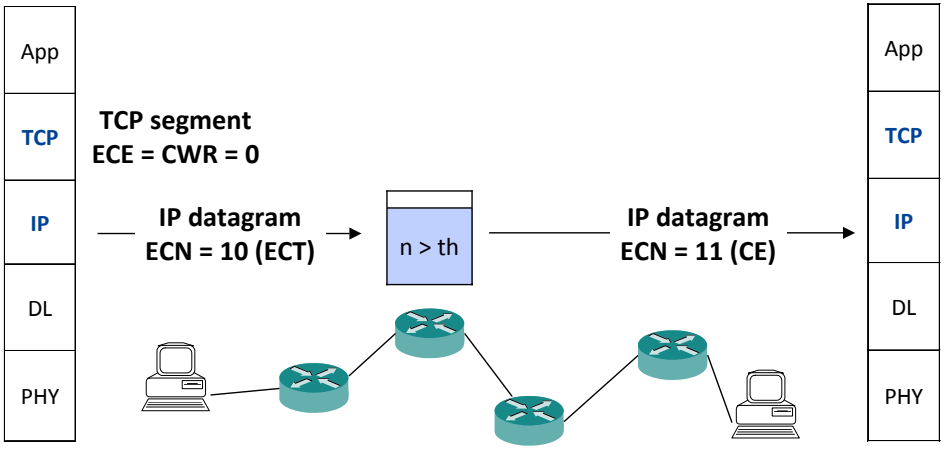
Incipient congestion

Router

Sees ECN = ECT in incoming IP header

Sets ECN = CE in outgoing IP header

Notifies receiver of incoming congestion



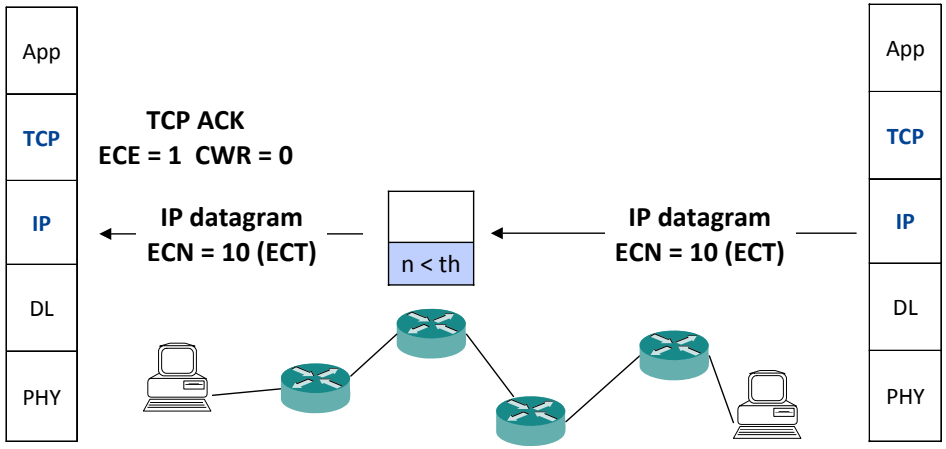
ECN Operation — 4

Incipient congestion

Receiver

Sets ECE = 1 in TCP header

Notifies sender of congestion



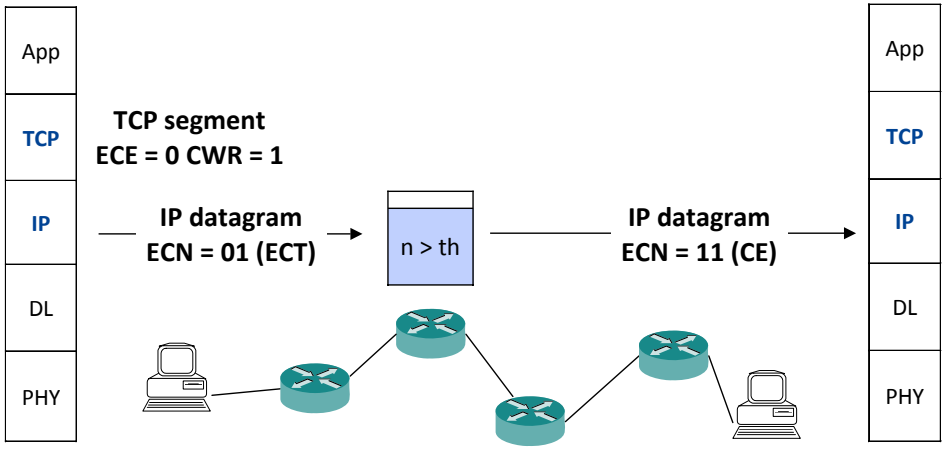
ECN Operation — 5

Incipient congestion

Sender

Lowers congestion window (once per RTT)

Sets CWR = 1 in TCP header (ACK of ECE to receiver)



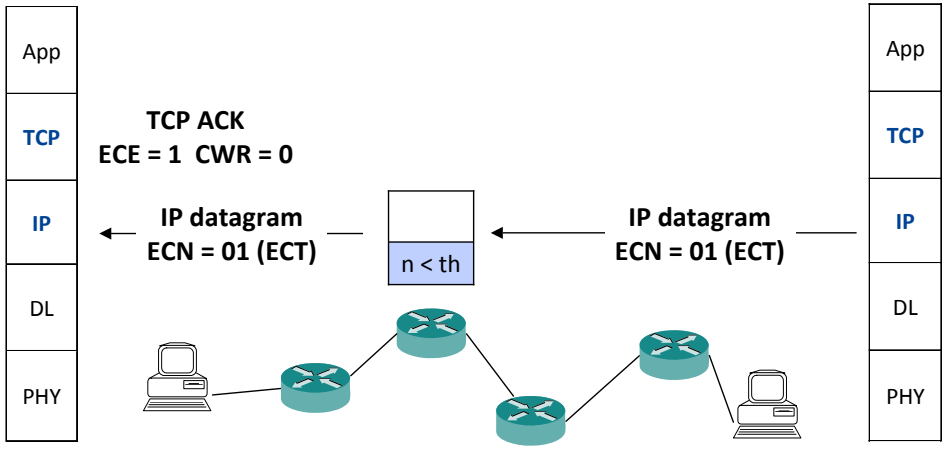
ECN Operation — 6

Incipient congestion

Receiver

Sees CWR = 1 in sender TCP header

CE in IP header new incoming ⇒ ECE = 1 in ACK TCP header



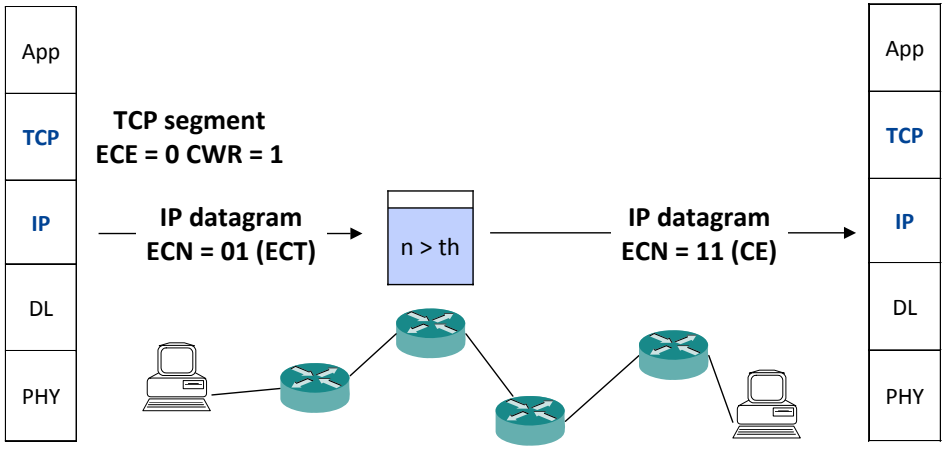
ECN Operation — 7

Continued congestion

Sender

Lowers congestion window once per RTT

Sets CWR = 1 in TCP header (ACK of ECE)



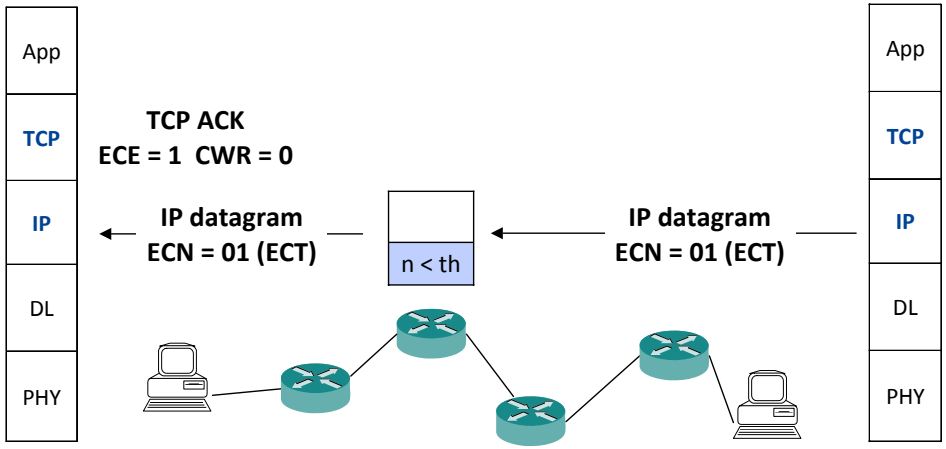
ECN Operation — 8

Continued congestion

Receiver

Sees CWR = 1 in sender TCP header

CE in IP header new incoming ⇒ ECE = 1 in ACK TCP header

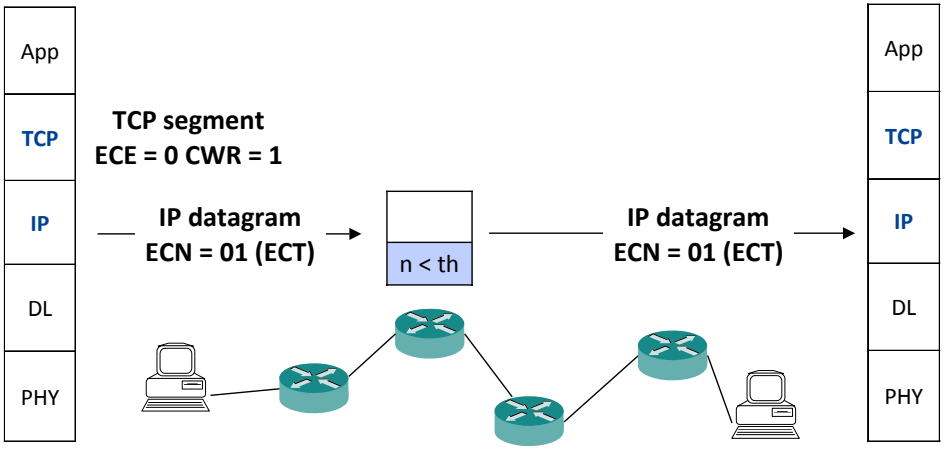


ECN Operation — 9

End of congestion

Sender sets CWR = 1 in TCP header (ACK of ECE)

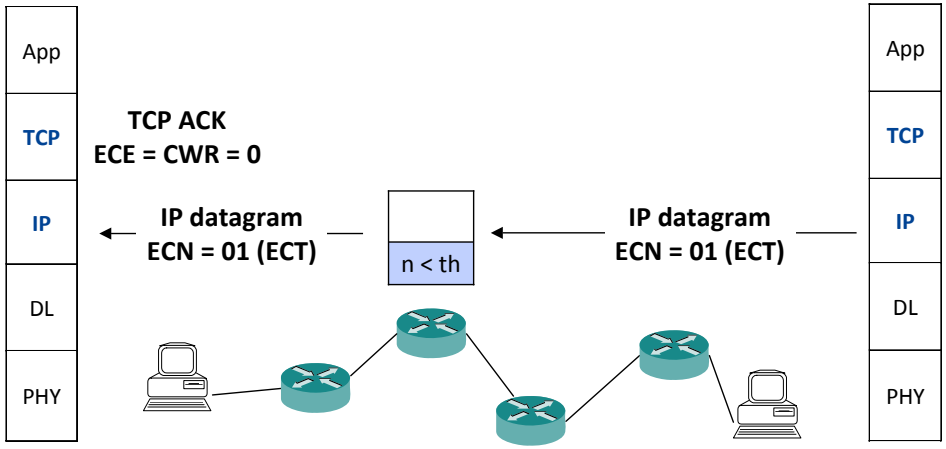
Router sends ECN = 01 in IP header (signals no congestion)



ECN Operation — 10

End of congestion

Receiver sends ECE = 0 in TCP header (signals no congestion)

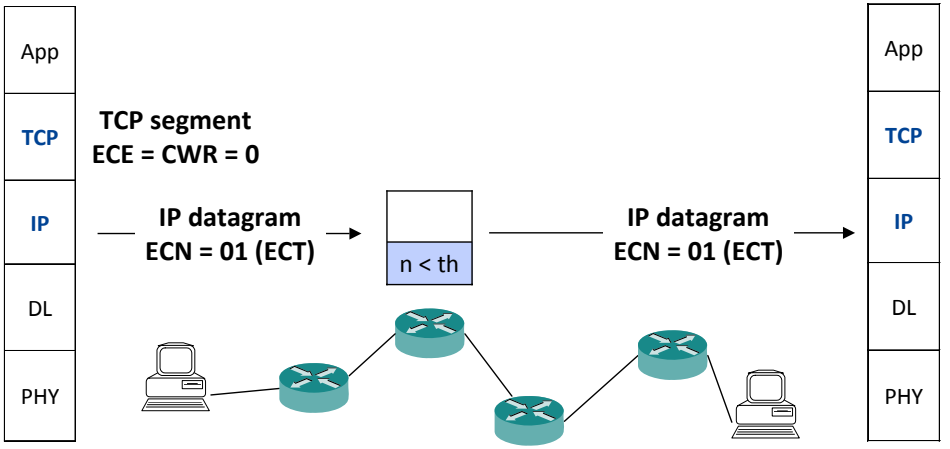


ECN Operation — 11

End of congestion

Sender clears CWR and begins raising congestion window

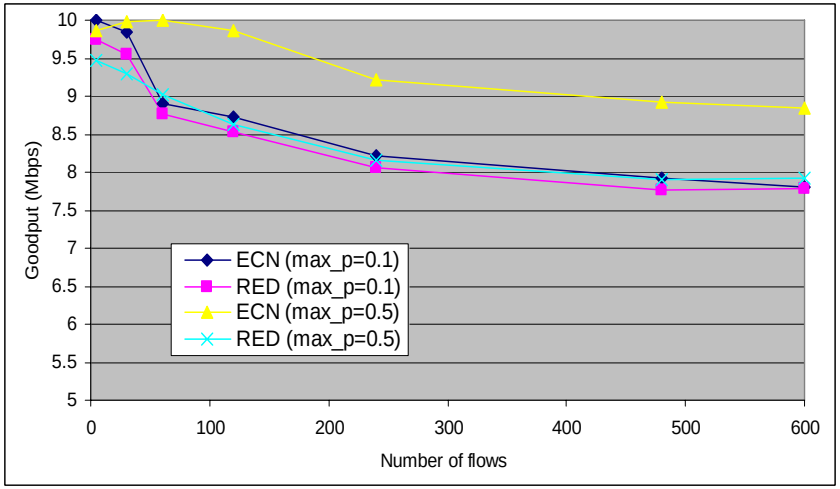
Router sends ECN = 01 in IP header



RED and ECN Goodput

Parameters

min_{th} = 5 max_{th} = 30

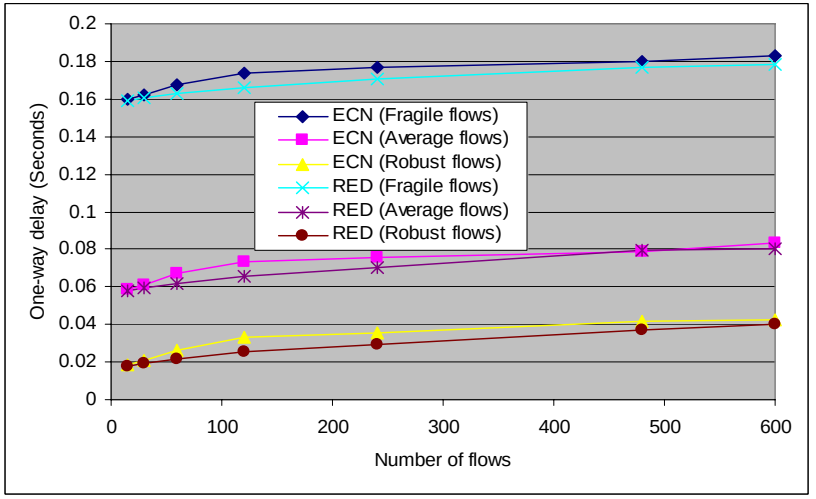


Ref: Kinicki and Zheng, A Performance Study of Explicit Congestion Notification (ECN) with Heterogeneous TCP Flows

RED and ECN Delay

Parameters

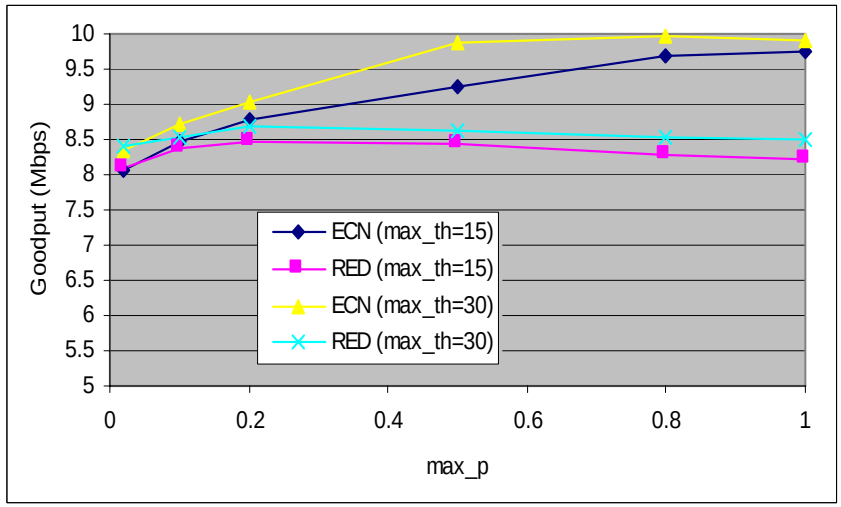
min_{th} = 5 max_{th} = 30 max_p =0.5



Goodput with 120 flows

Parameters

min_{th} = 5 max_{th} = 30



ECN Nonce (RFC 3540)

Problem

Unscrupulous or poorly implemented receiver

Clears ECN-Echo — no congestion signals to sender

Gives receiver advantage over connections that behave properly

Sender

IP header with **`ECN = 01 = ECT(0)`** or **`ECN = 10 = ECT(1)`**

Except retransmissions (**`Not ECN Capable`**) and **`CE`** packets

Keeps per-packet map of **`SEQ`** to nonce (`0` or `1`)

Router

Forwards packet or overwrites ECT with **`ECN = 11 = CE`**

Receiver

Keeps cumulative **`ACK`** number (standard TCP)

Keeps cumulative **`sum % 2`** of received nonces for ACKed packets

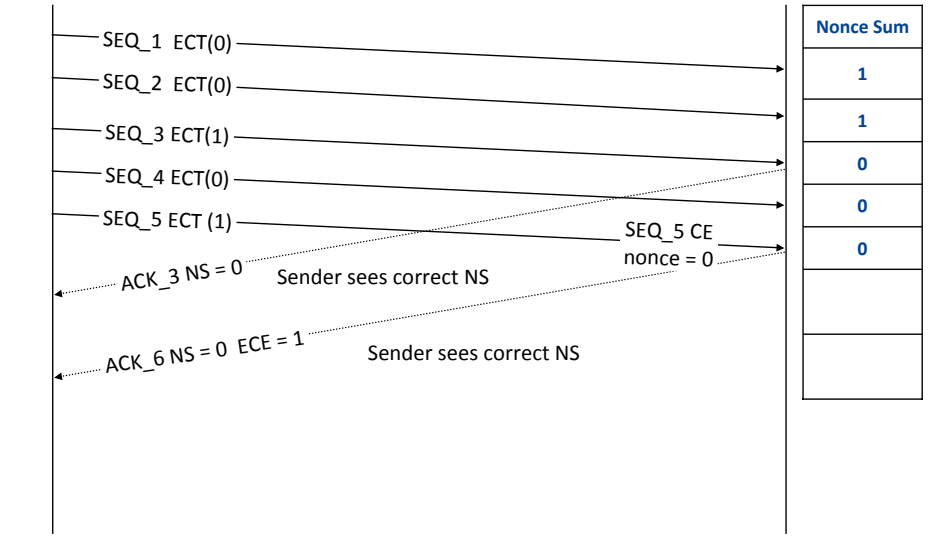
`NS` flag in TCP header = sum of nonces for ACKed packets

`CE` packets — use nonce = 0

Nonce Example

Honest Receiver

NS initialized to 1
Sent in SYN-ACK and ACK of handshake



Nonce Example

Cheating Receiver

Receiver ignores CE
Does not set ECE
Guesses nonce sum after CE

