Presentation 6

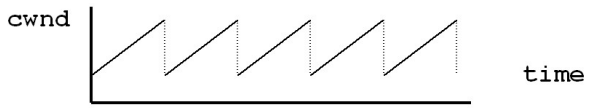
TCP over Wireless

The TCP protocol was designed many years before the introduction of WiFi and cellular networking, and its methods of congestion and flow control are based on assumptions relevant to wired transmission. In this chapter we discuss the implications of TCP for wireless and mobile networks.

Slides 2 – 17 Classic TCP-Reno

TCP-Reno is the basis for the **standard TCP implementation** in use today. In ideal operation, the number of segments in-flight (transmitted and waiting for ACK) is equal to cwnd (the congestion window, the maximum number of bytes that may be sent without stopping to wait for an ACK, measured in units of the MSS). The receiver generally returns a cumulative ACK for each 2 to 4 segments, and so with cwnd > 0, there is near-continuous transmission most of the time.

Given the finite-sized buffers in intermediate routers and the collision avoidance scheme, by which cwnd increases by 1 until a buffer loss divides cwnd in half, the expected behavior of cwnd over time is as shown in the graph:



The **RTT (Round Trip Time)** is the sum of four factors: the data transmit time, the time the segment spends in send buffers, the ACK transmit time and the time the ACK spends in ACK buffers.

The **buffer time** is approximately equal to the typical service time per segment × the buffer level.

Because these factors vary, the RTT is a random variable that can rise and fall sharply.

In TCP a **timeout** occurs when the buffer level is high, so the buffer time is large and the total RTT is larger than the RTO (retransmit timeout). For the TCP sender, the packet is considered lost. A packet arrives at the receiver **out-of-order** (OOO) if the buffer level is high enough for packet k that its RTT is larger than the RTT for packet k+1 (so that packet k is considered lost, but packet k + 1 is ACKed before its RTO runs out). The receiver will send a cumulative ACK if the OOO packet was not lost (for example if packet 16 arrives before packet 15 because over different data paths).

A **buffer error condition** occurs whenever a buffer level causes RTT > RTO for time interval T_{error}. The system sees an **isolated error** if T_{error} is shorter than the time between packets. In this case, the buffer empties before next packet arrives and there is no need to lower the transmission rate. **Congestion** occurs when T_{error} is longer than the time between packets, so that the problem causes multiple buffer errors and it is necessary to lower the transmission rate to reduce buffer errors.

The average transmission rate is $R = \left\langle \frac{\text{packets sent}}{\text{time to ACK}} \right\rangle = \left\langle \frac{\text{cwnd}}{\text{RTT}} \right\rangle$ and so R decreases if RTT increases or cwnd decreases. In an **isolated buffer error**, T_{error} is shorter than the time between packets so that RTT increases causing R to decrease during the time interval $T_{error.}$ In this case, the buffer corrects itself and there is no need to change cwnd. In **congestion**, T_{error} is longer

than the time between packets, so that even though RTT increases and causes R to decrease, the buffer is still too full. To address this, senders must decrease cwnd to decrease R even more.

Congestion control depends on an accurate estimate of RTT. The standard method for determining RTT in TCP is shown on slide 34. It turns out that this method tracks RTT too slowly. By taking an average that includes all past data, this measurement does not predict local congestion well.

Slide 6 shows the typical behavior of TCP-Reno when there is no congestion. The cwnd grows by 1 after each ACK, and on the next transmit cycle 1 additional segment can be transmitted. The value of in-flight in this example is stable at 3 (less than cwnd) because the receiver sends an ACK after 3 segments.

Slide 7 shows a numerical simulation of the variables SEQ, ACK, and cwnd over time. In the slow start phase ACK and cwnd overlap.

- At time = 0, cwnd = 1 so that SEQ = 1.
- At time = 0.5 the receiver returns ACK = 2 (SEQ = 2 is expected), which is 1 new ACK, so cwnd is raised to 2. Two segments are transmitted (SEQ = 2 and SEQ =3), and so SEQ = 3.
- At time = 1.0 the receiver returns ACK = 4 (SEQ = 4 is expected), which is 2 new ACKs, so cwnd is raised to 4. Four segments are transmitted (SEQ = 4, ... , 7), and so SEQ = 7.
- At time = 1.3 the receiver returns ACK = 7 (SEQ = 8 is expected), which is 4 new ACKs, so cwnd is raised to 8. Eight segments are transmitted (SEQ = 8, ... , 15), and so SEQ = 15.
- At time = 1.6 the receiver returns ACK = 12 and then ACK = 16, which are 8 new ACKs, so cwnd is raised to 16. Sixteen segments are transmitted (SEQ = 16, ... , 31), and so SEQ = 31.
- At time = 1.9 the receiver returns ACK = 20, ACK = 24, ACK = 28 and ACK = 32, which are 16 new ACKs, so cwnd is raised to 32. Thirty-two segments are transmitted (SEQ = 32, ... , 63), and so SEQ = 63. At cwnd = 32 the sender enters the congestion avoidance phase. ACKs arrive 4 at a time, and for each ACK, cwnd rises by 1.
- The sender transmits SEQ = 63 + 33 = 96, receives 4 ACKs at a time, and stops.

The latency (time from initial SEQ to final ACK) for the transmission of 96 segments is 2.77 seconds. The utilization (segments transmitted as a proportion of possible segments transmitted with cwnd $\rightarrow \infty$) is 34.2% and because there were no errors, this is also the goodput.

Slide 8 reviews the model of **steady state collision avoidance**. Assuming that the cwnd is fully utilized, n new ACKs reduces in-flight by n and raises cwnd by 1. The sender will then transmit n+1 new segments. The data transmission rate is just (cwnd × segment size) / RTT, which will decrease if RTT (round trip time) grows due to congestion or if cwnd is reduced by the sender. Decreasing the transmission rate is intended to give the buffers time to empty and reduce the congestion. The fill level of the buffer is indicated o the right side of the slide.

Slide 9 shows transmission under congestion (solid lines) compared with the congestion-free case of slide 4 (dotted lines). Congestion delays an ACK from time 2.2 to time 2.5. The total latency increases to 3.70 (0.3 is the actual excess delay and 0.63 is the delay in raising cwnd from a more restricted value). The utilization drops from 34.2% to 25.6%.

Slide 10 shows a typical steady state condition. The cwnd grows as usual (exponentially or linearly) until a segment is lost (SEQ = 9, 29, 49, 69, 89). After 3 dupACKs, the cwnd is cut in half and transmission continues. The drops in the red line show the retransmissions, which return the SEQ variable to the last ACKed segment. In this case, the total latency is 5.32, which is 92%

higher than the error-free case. The utilization drops to 20.8% and since some of those transmissions required retransmission, the goodput is only 17.8%.

Slides 11 and 12 show a loss of one segment in detail. With cwnd = 4, the first of 4 segments (SEQ = 7000) is lost. Because 3 additional segments follow, the sender receives 3 dupACKs, cuts cwnd in half, and sends 1 segment during RTT. The ACK raises cwnd to 3 and the sender transmits 3 more segments. Slide 9 shows the values of cwnd, SEQ, and ACK over time for this case.

Slide 13 shows the case of multiple successive segment losses. Because the sender cannot transmit additional segments, there are no dupACKs to indicate lost segments, and the sender cannot act until the timeout, which returns cwnd to 1 to begin retransmission. In this case, the long latency is required to reduce the transmission rate and permit the buffer to recover from congestion.

Slide 14 shows that a similar excess latency can be caused by the loss of a single packet early in the transmission when cwnd is small. SEQ = 1 is sent and ACK = 2 arrives at time 0.25. The cwnd is now 2 so that SEQ = 2 and SEQ = 3 are transmitted. At time 0.50 ACK = 3 arrives (acknowledging SEQ = 2) but no ACK arrives for SEQ = 3. Because the sender cannot transmit additional segments, there are no dupACKs for the lost segment and at time 4.0 the timeout returns cwnd to 1 and SEQ = 3 is retransmitted. If there are no additional errors, the transmission of SEQ = 96 and ACK = 97 will complete at time 6.87. Compared to the error-free case (slide 7) the excess latency is 48% higher and the utilization is only 14.1%. This shows that one early packet loss can be as costly as several packet losses in steady state. Slide 15 shows the very similar case of one lost packet retransmitted after 3 dupACKs, but followed by a loss of the retransmission.

Slides 16 and 17 summarize the operation of congestion control in TCP-Reno. The probability of a single bit error (which causes a single TCP segment loss) on a wired network is very small (around 10^{-5}) and so lost packets are interpreted as a sign of congestion. TCP-Reno has a moderate response to a packet loss, reacting after 3 dupACKs and lowering the cwnd by half. The probability of multiple successive packet errors on a wired network is extremely small, and so the more serious response to a timeout is considered necessary to allow the buffers to overcome the congestion condition.

Slides 18 – 21 The Trouble with Wireless

Despite the assumption is TCP, the probability of a single bit error in a wireless network, such as WiFi and cellular, is much higher than the bit error rate on a wired network. In addition, the probability of a long sequence of correlated errors is much higher, because errors are caused by external interference that may last over multiple segment transmission times. When TCP treats multiple successive segment losses as a sign of heavy congestion, it is usually wrong. This means that the overall utilization of wireless channels is kept unnecessarily low, when congestion is not a problem.

Transmission errors in wireless networks are caused by variations in the transmission medium and external interference. The conditions that cause refraction, reflection, and absorption can change with temperature and wind conditions, as well as with mobility of the wireless station. Sources or electromagnetic interference (other wireless users, electrical equipment, microwave ovens, and so on) may be active for seconds or minutes at a time, causing long sequences of correlated errors that do not occur on wired networks.

Slide 19 shows the problem of mobility drops — delays caused by mobility management. As the user moves from one wireless domain to another and additional node buffers are added to the system, the delay time for sending and receiving ACK messages may suddenly become longer than average, causing dupACKs and timeouts. This delay variation will be interpreted by TCP as a sign of congestion, leading to a drop in cwnd and loss of throughput.

Slide 20 shows the problem of ACK compression. Wireless links are usually asymmetric, with the upstream channel (user to base) transmitting much slower than the downstream channel (base to user). As a result the mobile device will operate with a larger buffer, leading to a longer buffer delay to accommodate the lower average transmission rate. In the example, the base sends three segments and the mobile device buffers several ACKs until a convenient time to transmit in the wireless system. The base station receives a larger than normal number of ACKs very quickly so it raises the cwnd very quickly. The RTT for the first ACK was long, but the remaining ACKs arrived soon after, so the average RTT is seen to be short. The base can then transmit a large number of segments very quickly, which may flood the mobile device and cause overflow in its receive buffer.

Slide 21 shows a problem in TCP caused by the use of the reliable MACA protocol in WiFi. MACA has overhead associated with the RTS, CTS and ACK_{MAC} packets. The time to handle this overhead may be as long at the TCP ACK time, and so RTT will be twice the RTT on a wired network. Because the long RTT is interpreted as congestion, the TCP agent will lower the average transmission rate unnecessarily.

Slides 22 – 23 Typology of Approaches to Wireless TCP

Several approaches have been proposed to modify TCP for use over wireless networks. The basic approaches can be divided into two groups:

- Split mode approaches were proposed in the 1990s, using standard TCP between the wireless base station and server, and specialized TCP between the base and mobile.
- End-to-end approaches have been proposed more recently. These are refinements to TCP-Reno for improved handling of non-congestion packet loss. They often require infrastructure-awareness — the use of IP and infrastructure layer information at the TCP layer (as was seen when ECN provides IP router information to TCP).

Within these groups, proposals can be divided into reactive and proactive congestion control:

- Reactive congestion control responds to a perceived congestion loss in the way that TCP-Reno cuts cwnd after 3 dupACKs.
- Proactive congestion control attempts to predict the actual available bandwidth from ACK arrival rates.

Slide 23 is an example of the case that many wireless TCP proposal address.

- With cwnd = 16 the sender transmits 16 segments, SEQ = 16 to SEQ = 31.
- 10 segments (SEQ = 16 to SEQ = 25) arrive correctly and the receiver returns ACK = 26. The sender reduces in-flight to 16 – 10 = 6 and raises cwnd to 16 + 10 = 26.
- 3 segments (SEQ = 26, 27, 28) are lost.
- 3 segments (SEQ = 29, 30, 31) arrive correctly. Because the receiver is still waiting for SEQ = 26, it returns ACK = 26 for each of these segments.

- The 3 dupACKs of 26 cause the sender to cut cwnd from 26 to 13 and retransmit SEQ = 26.
- The retransmission of SEQ = 26 arrives correctly, but now the receiver is still waiting for SEQ = 27 and sends ACK = 27. The sender adds 1 to cwnd and subtracts 1 from in-flight. But because cwnd > in-flight, and there has been no timeout for segments in-flight, the sender can assume that the ACK = 28 to ACK = 32 will arrive later. It transmits 9 more segments (SEQ = 32 to 40).
- The receiver responds to the 9 new segments by sending 9 dupACKs with ACK = 27.
- After 3 of these dupACKs, the sender cuts cwnd from 14 to 7 and retransmits SEQ = 27. The retransmission arrives correctly but the receiver is still waiting for SEQ = 28 and sends ACK = 28.
- But because cwnd > in-flight, and there has been no timeout for segments in-flight, the sender can assume that the ACK = 29 to ACK = 40 will arrive later. It transmits 1 more segments (SEQ = 41).
- The receiver responds to the new segment by returning a dupACK for ACK = 28, but it cannot send more dupACKs because no more segments arrive.
- Finally the sender has a timeout on SEQ = 28. It retransmits SEQ = 28.
- The receiver has buffered the out-of-order segments (SEQ = 29 to 41) and sends a cumulative ACK = 42. The sender can now continue transmitting.

Slide 24 New Reno

New Reno is a slight modification of TCP-Reno standardized by the Internet Engineering Task Force (IETF) as RFC 6582. The idea is that after 3 dupACKs the sender continues **fast recovery** until all in-flight segments are ACKed. This means that it retransmits all segments following the dupACK until it receives a cumulative ACK.

Slide 24 shows the same case as on slide 23 — the sender transmits 16 segments with but 3 segments (SEQ = 26 to 28) are lost. The receiver returns 3 dupACKs for ACK = 26, putting the sender into fast recovery mode. After receiving ACK = 27, instead of assuming that the remaining ACKs (28 to 31) will arrive later, the sender continues to retransmit SEQ = 27 and 28. When SEQ = 28 arrives, the receiver returns a cumulative ACK = 31, which ACKs all the segments in-flight. The receiver can now continue transmitting, with no loss of time caused by a timeout.

Slide 25 TCP-NCL (Non-Congestive Loss)

TCP-NCL was proposed in 2010 to improve wireless transmission based on a heuristic assumption: a single timeout could be caused by a bit error, but a timeout of the retransmitted segment is probably caused by congestion. The method is expressed in the slide as a decision tree. There are two separate timeout clocks, the Retransmission Decision (RD) whose timeout interval is the standard RTO, and the Congestion Decision (CD) with a longer timeout interval. A timeout of the RD causes a retransmission of the segment and starts the CD timer, but does not affect the value of cwnd (the transmission rate continues as it was). An ACK following the retransmission resets both the RD and the CD. If there is no ACK when the CD timeout occurs, then the congestion control mechanism returns cwnd to 1.

Slide 26 TCP Vegas

TCP Vegas was first proposed in 1995. Instead of changing the cwnd by a large amount after dupACKs or a timeout, it adjusts cwnd every few ACKs based on the changing RTT.

As in Reno, the slow start phase begins with cwnd = 1 and exponential growth of cwnd after each ACK. The sender defines an **expected throughput** given by the cwnd divided by the minimum value of RTT measured in the system (under no congestion). The sender also defines lower and upper **threshold values** α and β. Every few transmissions, the sender transmits a special segment, measures the RTT for that segment, and calculates the **actual throughput** as the size of the ACK received divided by the RTT for the special segment. The sender then calculates the variable **diff**, which is the difference between the actual and expected throughput. If diff < α then the sender raises cwnd by 1 on each ACK. If diff > β then the sender lowers cwnd by 1 on each ACK. If diff is between α and β then cwnd remains unchanged.

Fast recovery is also modified. On 3 dupACKs the sender retransmits the missing segment (as in Reno) but does not change the cwnd. This prevents packet errors from being treated like congestion. The sender also defines a Vega-timeout RTT. If the actual RTT is larger than this value then the sender begins retransmission after 1 dupACK.

Slides 27 – 28 TCP — Modified Fast Retransmission (MFR)

TCP — MFR is an attempt to count dupACKs as a measure of traffic that arrives but is not ACKed.

In Reno, as seen in slide 15, if cwnd segments are transmitted and segment n is lost then k dupACKs are returned, for some value of k. The sender retransmits after 3 dupACKs and waits for the ACK of segment n. But if the retransmitted segment is also lost, then there are no more dupACKs, and the sender must wait for a timeout on segment n.

In TCP-MFR, the sender receives dupACKs, assumes that only one segment was lost, and retransmits the missing segment. The sender also counts the k dupACKs that arrive and determines that k segments must have arrived (to trigger the return of k dupACKs). If segment n was lost and k segments arrived then k + 1 segments are accounted for, and cwnd – (k + 1) are unaccounted for.

If more than 3 dupACKs arrive then the sender increases cwnd by (k – 3) and sends more segments. If a cumulative ACK arrives then the retransmitted and additional segments all arrived. But if there are more dupACKs for segment n then some new segments arrived, but the retransmission of segment n was also lost. Now the sender retransmits segment n again with no timeout.

Slide 28 shows an example. The sender transmits 16 segments (SEQ = 16 to 31). All 16 segments arrive correctly except SEQ = 21. Segments SEQ = 16 to 20 arrive correctly and the receiver sends ACK = 21. The in-flight now drops from 16 to 11 and the cwnd rises from 16 to 21. Ten more segments (SEQ = 22 to 31) arrive and the receiver returns dupACKs for ACK = 21. After 3 dupACKs the sender drops cwnd to 13 and retransmits SEQ = 21 but it is also lost.

At this point, the sender has received 8 dupACKs for ACK = 21, so it knows that of the 16 segments transmitted, 5 were ACKed, at least 1 was lost and 8 must have arrived (but it does not know which segments). So, 2 segments are unaccounted for. So now, k – 3 = 8 – 3 = 5 which is bigger than the number of unaccounted segments, and the sender can add 5 to cwnd (now 18) and send 7 more segments (SEQ = 32 to 38). The receiver is still waiting for segment SEQ = 21 and sends dupACKs for ACK = 21. After 3 more dupACKs, the sender does not know if these are

new or delayed dupACKs. But then 6 more dupACKs for ACK = 21 arrive (9 dupACKs = 2 delayed dupACKs from the first segment loss and 7 new dupACKs from the second segment loss). The sender sees that k – 3 = 9 – 3 = 6 is larger than the 2 unaccounted segments and retransmits SEQ = 21 again. No timeout takes place.

Slide 29 Modified Congestion Control (MCC)

Modified Congestion Control (MCC) is an attempt to set cwnd according to measured capacity instead of just raining or lowering the value. The method defines a Bandwidth Estimate (BWE) based on measurements of RTT and the formula

$$\text{BWE} = \text{weighted average of} \frac{\text{bits ACKed}}{\text{RTT for bits_to_ACK}}$$

On ACK number k, the sender calculates the BWE for sample k and the weighted average

$$\text{BWE_s[k]} = \frac{\text{packets ACKed} \times \text{bytes per packet}}{\text{now} - \text{last_ACK_time}}$$

$$\text{BWE[k]} = (1-\beta) \times \frac{\text{BWE_sample[k] + BWE_sample[k-1]}}{2} + \beta \times \text{BWE[k-1]}$$

The value of cwnd is used as in TCP-Reno, but it is set after congestion as:

On 3 dupACKs, ssthresh is set as

$$\begin{aligned}
\text{ssthresh} &= \text{BWE} \frac{\text{RTT}_{\text{min}}}{\text{segment size}} \\
&= \text{average measured} \left(\frac{\text{bytes ACKed}}{\text{measured RTT}} \right) \times \left(\frac{\text{RTT}_{\text{min}}}{\text{bytes per segment}} \right) \\
&= \text{average measured} \left(\frac{\text{RTT}_{\text{min}}}{\text{measured RTT}} \times \frac{\text{bytes per segment} \times \text{segments ACKed in RTT}}{\text{bytes per segment}} \right) \\
&= \text{average measured} \left(\frac{\text{RTT}_{\text{min}}}{\text{measured RTT}} \times \text{segments ACKed in RTT} \right)
\end{aligned}$$

and cwnd is lowered to ssthresh (if cwnd > ssthresh).

On a timeout,

ssthresh = max{(BWE*RTTmin) / segment size, 2}

cwnd = 1

Slide 30 Proposed Solutions for WiFi Overhead

Slide 31 Freeze-TCP

Freeze-TCP is an attempt to handle mobility drops caused by handover delays and temporary signal fading by obstacles. The basic idea is that the mobile station monitors the radio signal strength from its base station and predicts impending disconnections caused by handover (from cell to cell) or obstacles. The Freeze-TCP receiver proactively sets the window field in the TCP header to 0, so that the TCP sender enters persist mode (it cannot transmit until the persist

timer runs out — see slide 7 in chapter 5). When the mobile station reconnects and establishes a strong radio connection, the receiver sends multiple ACKs of the last received packet. This prevents the exponential back off of cwnd in the sender (decrease of cwnd caused by dupACKs).

This method requires **infrastructure awareness** in the TCP layer, which must be exposed to networking details handled by lower layers, such as roaming and handover at layers 2 and 3, and radio signal quality at layers 1 and 2.

Slides 32 – 33 TCP-Jersey

TCP-Jersey defines Available Bandwidth Estimation (ABE), which is similar to the BWE used in MCC (slide 29), but measured in segments sent over a fixed time (instead of segments per second). Like, MCC the method uses ABE to determine expected capacity, instead of raising and lowering cwnd. Like Freeze-TCP (slide 31), the method requires explicit congestion notification (ECN) support in routers, making it infrastructure aware. The method defines three cases:

- A **congestion warning (CW)** occurs when an IP router marks a segment on a congested link and this information is passed to the TCP receiver, which returns an ECN warning to the TCP sender. If CW occurs with no packet loss then cwnd is set to ABE.
- On **3 dupACKs**, the sender retransmits the lost segment as usual. If the segment carrying the dupACK includes an ECN congestion warning, the cwnd is set to ABE. Otherwise, the sender assumes that the loss was caused by a data error and leaves cwnd unchanged.
- On a **timeout**, the sender retransmits the lost segment as usual. If the segment carrying the dupACK includes an ECN congestion warning, the cwnd is set to 1. Otherwise, the sender assumes that the loss was caused by a data error and leaves cwnd unchanged.

Slide 29 shows results of measurements on wireless networks. Jersey-TCP is seen to perform better (goodput at some bit error rate) than Reno, Tahoe (a predecessor of Reno), and Westwood, under conditions of congestion and no congestion. The problem with **TCP-Jersey** is that it requires modification to the TCP sender and receiver, as well as infrastructure awareness.

Slides 34 – 47 TCP Westwood

TCP Westwood combines the dupACK counting used in MFR (slide 27) with the BWE used in MCC (slide 29). Its performance is almost as good as TCP-Jersey but it has several advantages. First it only modifies TCP sender and so is not dependent on negotiation with TCP receiver. It is also not dependent on ECN support in routers or any other infrastructure issue.

In estimating the available capacity, Westwood counts both ACKs and dupACKs as successful traffic. On a packet loss it sets cwnd equal to the measured available bandwidth.

Westwood uses the Reno slow start and congestion avoidance modes to raise **cwnd**. As in MCC, on dupACKs and timeouts Westwood sets cwnd according to an estimate of available capacity. As in MFR, Westwood counts dupACKs in calculating the bandwidth samples. The basic idea is to count a dupACK as a segment that arrived (some segment must have arrived or there would not have been a dupACK) and then avoid double counting the ACK that confirms the actual arrival.

Slide 36 shows an example this dupACK counting.

- At time t_0 segment SEQ = 1 is sent.
- At time t_1 ACK = 2 (acknowledging SEQ = 1) arrives and the sender estimates the bandwidth as $1 / (t_1 – t_0)$. Segments SEQ = 2, 3 are sent.
- At time t_2 ACK = 4 arrives and the sender estimates the bandwidth as $2 / (t_2 – t_1)$. Segments SEQ = 4, 5, 6, 7 are sent. SEQ = 5 is lost.
- At time t_3 ACK = 5 arrives and the sender estimates the bandwidth as $1 / (t_3 – t_2)$. Segment SEQ = 8 is sent.
- At time t_4 ACK = 5 arrives and the sender estimates the bandwidth as $1 / (t_4 – t_3)$. The dupACK counter = 1.
- At time t_5 ACK = 5 arrives and the sender estimates the bandwidth as $1 / (t_5 – t_4)$. The dupACK counter = 2.
- At time t_6 ACK = 5 arrives and the sender estimates the bandwidth as $1 / (t_6 – t_5)$. The dupACK counter = 3. Segment SEQ = 5 is retransmitted (after 3 dupACKs) and new segment SEQ = 9 is transmitted.
- At time t_7 cumulative ACK = 10 arrives. The ACK jumps from 5 to 10, which provides 5 new ACKs. The sender already counted 3 dupACKs (following SEQ = 6, 7, 8) as segments that arrived, so the bandwidth estimate is based on 5 new ACKs – 3 dupACKs = 2 new segment arrivals. The sender estimates the bandwidth as $2 / (t_7 – t_6)$.

Slide 37 formalizes this counting method as follows:

`newACK = ACK – prevACK ;`	The number of packets ACKed by the new ACK
`// if (newACK = 1) do nothing`	No error condition — report 1 arrival

`if (newACK = 0)`
`  count++ ;`
`  newACK = 1 ;`
	A new ACK arrives but the ACK value is "stuck" on the old value, so that this ACK is a dupACK.
	Increment the dupACK counter and report this dupACK as 1 new arrival.

`if (newACK > 1)`
`  if (count >= newACK)`
`  count = count – newACK + 1 ;`
`  newACK = 1 ;`
	The ACK value advances, but not all packets that arrived at the receiver have been ACKed.
	Remove the ACKed packets from the dupACK count and count this dupACK as 1 new arrival.

`  else if (count < newACK)`
`  newACK = newACK - count ;`
`  count = 0;`
	All packets have been ACKed in the correct order.
	Report ACKed – counted (new ACKs) as the number of arrivals.
	Zero dupACK counter

`prevACK = ACK ;` `return(newACK);`	

Slide 38 shows an example of the counting method when there are no lost packets. The sender transmits SEQ = 1, SEQ = 2,3, SEQ = 4,5,6,7, and SEQ = 8,9,10,11,12,13,14,15 and the receiver returns ACKs 2, 4, 8, 12, and 16.

- On ACK = 2, newACK = 2 – 1 = 1, so the counter reports 1 new arrival.
- On ACK = 4, newACK = 4 – 2 = 2, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 2 new arrivals.
- On ACK = 8, newACK = 8 – 4 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 4 new arrivals.
- On ACK = 12, newACK = 12 – 8 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 4 new arrivals.
- On ACK = 16, newACK = 12 – 8 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 4 new arrivals.

Slide 39 shows an example of the counting method when one packet arrives in the wrong order. The sender transmits SEQ = 1, SEQ = 2,3, SEQ = 4,5,6,7, and SEQ = 8,9,10,11,12,13,14,15, but SEQ = 5 arrives before SEQ = 4. The receiver returns ACKs 2, 4, 4, 6, 10, 14, and 16.

- On ACK = 2, newACK = 2 – 1 = 1, so the counter reports 1 new arrival.
- On ACK = 4, newACK = 4 – 2 = 2, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 2 new arrivals.
- On the second ACK = 4, newACK = 4 – 4 = 0, so the counter increments the dupACK count and reports 1 new arrival.
- On ACK = 6, newACK = 6 – 4 = 2, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 1, the counter reports 1 new arrival and zeros the dupACK counter.
- On ACK = 10, newACK = 10 – 6 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 4 new arrivals.
- On ACK = 14, newACK = 10 – 6 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 4 new arrivals.
- On ACK = 16, newACK = 16 – 10 = 2, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 2 new arrivals.

Slide 40 shows another example of the counting method when one packet arrives in the wrong order. The sender transmits SEQ = 1, SEQ = 2,3, SEQ = 4,5,6,7, and SEQ = 8,9,10,11,12,13,14,15, but SEQ = 5, 6, 7 arrive before SEQ = 4. The receiver returns ACKs 2, 4, 4, 4, 4, 8, 12, and 16.

- On ACK = 2, newACK = 2 – 1 = 1, so the counter reports 1 new arrival.
- On ACK = 4, newACK = 4 – 2 = 2, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 2 new arrivals.
- On the second ACK = 4, newACK = 4 – 4 = 0, so the counter increments the dupACK count to 1 and reports 1 new arrival.
- On the third ACK = 4, newACK = 4 – 4 = 0, so the counter increments the dupACK count to 2 and reports 1 new arrival.
- On the fourth ACK = 4, newACK = 4 – 4 = 0, so the counter increments the dupACK count to 3 and reports 1 new arrival.

- On ACK = 8, newACK = 8 – 4 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 3, the counter reports 1 new arrival and zeros the dupACK counter.
- On ACK = 12, newACK = 12 – 8 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 4 new arrivals.
- On ACK = 16, newACK = 16 – 12 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 0, the counter reports 4 new arrivals.

Slides 41 and 42 show variations of the case in slide 40.

Slide 43 shows another example of the counting method when two packets arrive in the wrong order. The sender transmits SEQ = 1, SEQ = 2,3, SEQ = 4,5,6,7, and SEQ = 8,9,10,11,12,13,14,15, but SEQ = 9, 10, 11 and SEQ = 13, 14, 15 arrive before SEQ = 8 and SEQ = 12. The receiver returns ACKs 8, 8, 8, 8, 8, 8, 12, and 16.

- On ACK = 8 (not shown), newACK = 8 – 4 = 4, so the counter reports 4 new arrivals.
- On ACK = 8 (2^{nd} time), newACK = 8 – 8 = 0, so the counter increments the dupACK count to 1 and reports 1 new arrival.
- On ACK = 8 (3^{rd} time), newACK = 8 – 8 = 0, so the counter increments the dupACK count to 2 and reports 1 new arrival.
- On ACK = 8 (4^{th} time), newACK = 8 – 8 = 0, so the counter increments the dupACK count to 3 and reports 1 new arrival.
- On ACK = 8 (5^{th} time), newACK = 8 – 8 = 0, so the counter increments the dupACK count to 4 and reports 1 new arrival.
- On ACK = 8 (6^{th} time), newACK = 8 – 8 = 0, so the counter increments the dupACK count to 5 and reports 1 new arrival.
- On ACK = 8 (7^{th} time), newACK = 8 – 8 = 0, so the counter increments the dupACK count to 6 and reports 1 new arrival.
- On ACK = 12, newACK = 12 – 8 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 6, which is larger than newACK, the counter updates the count to 6 – 4 + 1 = 3 and reports 1 new arrival.
- On ACK = 16, newACK = 16 – 12 = 4, which is larger than 1, so the counter compares this value with the dupACK count. Since the count is 3, the counter reports 4 – 3 = 1 new arrival.

Slides 44 – 47 Westwind Congestion Control

Slide 44 shows the modifications of Reno made by Westwood. Reno slow start and congestion avoidance operate without change. As in MCC, on dupACKs and timeouts Westwood sets cwnd according to an estimate of available capacity, instead of lowering it arbitrarily. On a timeout, Westwood resets cwnd to 1, but set ssthresh to the bandwidth estimate.

Slide 45 compares performance of Westwood to Reno and SACK (slide 24 in chapter 5) and shows improved throughput at all error rates.

Slide 46 shows the values of cwnd and ssthresh over time for Westwood (left) and Reno (right). As expected, Westwood maintains higher cwnd (the main factor in throughput in the absence of heavy congestion). If congestion is low but there are frequent packet losses caused by bit errors,

then the behavior of cwnd in Reno will be similar to the steady state shown in slide 10 and repeated here.

In Westwood cwnd is set according to the measured throughput and not affected by bit errors. In particular, is useful to compare Reno and Westwood in **congestion avoidance**, in which case **cwnd = ssthresh**. Reno and Westwood raise cwnd linearly, adding 1 for each ACK. If there are periodic packet losses, then each time Reno cuts cwnd in half, leading to the steady state behavior shown in slide 46. But in Westwood, the bandwidth estimate (BWE) is based on the number of bytes ACKed (including dupACKs) and so ssthresh will continue to grow until it reaches a reasonable approximation to actual capacity. On a packet loss, the cwnd will not be cut (unless there is actual congestion that decreases ssthresh) and cwnd will keep growing linearly until it reaches the value of ssthresh that represents actual capacity.