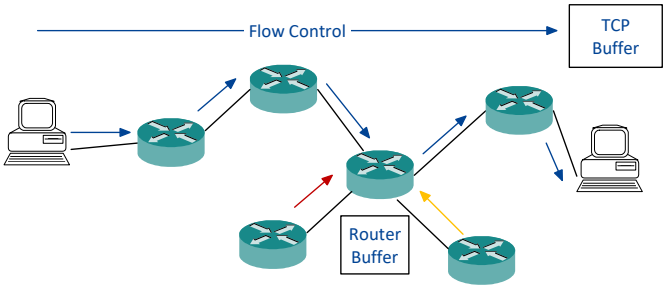
Presentation 5

Congestion and Flow Control in TCP

Congestion and flow control are basic and essential functions for reliable transport, and for most use cases are implemented in the TCP protocol. These functions add overhead to network operation and in many circumstances degrades the transmission throughput unnecessarily.

Slides 2 – 5 Flow Control and Congestion Control

The hosts in a TCP session maintain buffers filled and emptied by the application layer. Internet routers maintain buffers of IP datagrams waiting to be forwarded.



Recalling the basic definitions,

- **Flow control** permits the sender to avoid overflow of the receiver's buffer.
- **Congestion control** permits all senders to avoid overflow of intermediate network buffers, especially router buffers.
- **Buffer fill rate** is the number of bytes per second arriving at a buffer from the network.
- **Buffer empty rate** is the number of bytes per second leaving the buffer to the network or to the application layer at the destination host.
- **Buffer fill time** is the time to overflow an empty buffer, given by

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

From queuing theory we learn that buffer level and latency (waiting time for an arriving packet in a buffer) depends on **buffer utilization** defined as

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

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

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

Because long buffer delays can cause timeouts and retransmissions in TCP, the data throughput begins to decline at about 85% utilization.

Slides 6 – 9 TCP Flow Control

Flow control in TCP is managed by a "sliding window" method. The sender maintains a **source window**, the maximum number of TCP segments it is permitted to transmit before stopping to wait for ACKs. The initial source window is set by a system policy and is the maximum number of bytes permitted "in flight" (send and not yet ACKed). The source window is adjusted by mechanisms of congestion and flow control.

The receiver maintains a **destination window**, which is the number of bytes that the receiver can accept, and is determined by the available space in the receiver buffer. At any given time, the receiver buffer level is adjusted as the previous level plus the number of arriving bytes minus the number of bytes read by the application layer. If the application reads too slowly, then TCP must decrease its destination window.

To coordinate the flow rate the sender and receiver use a **sliding window**. The receiver uses the **window field** in the TCP header to report the number of bytes it will accept. The sender must adjust its source window — the receiver will discard any received bytes beyond its window size.

Slide 7 shows an example of TCP flow control.

- Before transmission (no bytes in flight), the sender sets a policy-based source window of 64 KB, and the receiver (with empty buffer) sets its destination window to 8 KB (its actual buffer size).
- The sender transmits two segments of 2 KB each. The in-flight counter rises to 4 KB and the buffer level at the receiver also rises to 4 KB. The receiver resets its destination window to 4 KB, and reports this number to the sender in the window field of the ACK.
- After the first ACK, the sender in-flight counter returns to 0 (all sent bytes were ACKed), and adjusts its source window to 4 KB (as reported by the receiver).
- The sender transmits another 2 KB, adjusting its in-flight counter and source window, and the receiver adjusts its buffer level and destination window.
- The application then reads 4 KB from the buffer. The receiver readjusts its buffer level and destination window, and reports the new destination window in the ACK.
- The sender now transmits 6 KB, adjusting its in-flight counter and setting its source window to 0. The receiver buffer is now full and the receiver reports a receive window of 0. The in-flight counter is 0 but the sender cannot transmit because the source window is 0 as part of the flow control mechanism.
- When the application reads 4 KB of data, the receiver repeats the previous ACK to report a receive window of 4 KB. But this ACK is lost and the sender does not know that it can transmit. The sender waits a certain interval (**persist timeout**) and then sends one byte to check the receiver. The receiver sends an ACK of the 1 byte and again reports a destination window of 4 KB (minus 1 byte). The sender is now free to continue transmitting.

Slide 8 again shows the deadlock problem solved by the persist timeout. Slide 9 shows another bug, the **silly window problem**, in which a low receiver application leads to a large number of short segments and ACKs (more header than data). In the Nagle algorithm, the sender buffers data, waiting until it can send a large segment. This works badly with text-oriented transmission like Telnet. An alternative is to keep the window size at 0 until the destination window is a reasonable size.

Slide 10 TCP Congestion Control

End-to-end congestion control in TCP is far more complex than flow control. The transport layer (layer 4 in the Internet model) is meant to ignore the details of the Network layer (layer 3) and receives no feedback from the intermediate network nodes. Therefore congestion control in TCP is based on heuristic estimates based on data the host receives.

In **slow-start**, the sender begins the session with a low transmission rate and increases the rate exponentially until timeouts begin. In order to avoid waiting for a timeout (with no transmissions if the in-flight is at maximum), a **fast retransmit** mechanism triggers retransmission after 3 duplicate ACKs (dupACKs), which usually means that a packet was lost. **Congestion avoidance** moderates the growth of the transmission rate after duplicate ACKs.

Slides 11 – 12 Slow-Start

In the slow start method used by current TCP implementations, the source window is called a **congestion window** and associated with the internal sender variable **`cwnd`**. The congestion window is initialized to the maximum segment size (MSS), which is the largest number of bytes that can be sent in one TCP segment (packet of header and data). The sender can thus send one MSS and then must wait for an ACK before it can send more data.

The **round trip time (RTT)** is the average time between the transmission of a segment by the sender and the arrival of the ACK at the sender. Since the sender starts by sending 1 MSS and waiting RTT seconds for the ACK, the average data rate is 1 MSS / RTT.

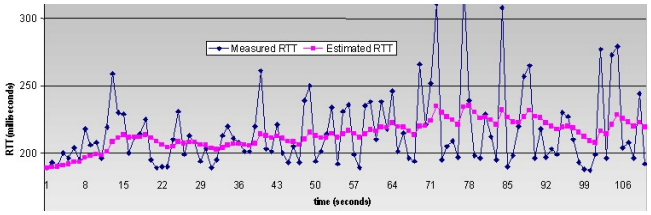
The send can raise the cwnd but the maximum cwnd is the destination window reported by the receiver. In slow start cwnd grows exponentially. On each ACK that arrives, the sender adds the size of ACKed data to the previous cwnd until cwnd reaches its maximum value. If there is congestion then at some value of cwnd the data rate will be too high, packets will be lost, and there will be a timeout while waiting for the required ACK. On a timeout, the cwnd is reset to 1 MSS.

This means that under conditions of congestion, the data rate will begin well below the capacity rate, grow exponentially until it is too high and causes a timeout, and then drops back below the capacity rate. On average, the **utilization** will be about half of the practical capacity under congestion.

In order to set the timeout interval (**RTO**) to a reasonable value, the sender must monitor the RTT of each ACK. The method of determining the RTO is shown on slide 12. The initial RTO is 3 seconds. After the first measured RTT, the RTO is set as

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

where SRTT is the measured RTT, G is a fixed granularity of 0.5 seconds, and RTTVAR is an estimate of the variance of RTT given by the measured RTT. After each measurement of subsequent values of RTT, the RTO is reset by the weighted average of old and new values shown in slide 12. The pink line in the graph shows the averaging outcome.



Slide 13 Fast Retransmit

In order to improve underutilization in the TCP slow start method, **fast retransmit** sets the RTO to be much longer than the average RTT and begins lost packet retransmission when 3 duplicate ACKs (**dupACKs**) are received with the same ACK number.

In the example, 5 segments are transmitted, each with 100 bytes of data so that the SEQ number rises from 100 to 500. After the first segment, the receiver sends ACK = 200, which acknowledges all data up to 199 and states an expectation to receive SEQ = 200. But the segment with SEQ = 200 is corrupt. When the segment with SEQ = 300 arrives, the receiver is still expecting SEQ = 200 and responds to the out-of-order segment by repeating the ACK of 200. Similarly, when the segments with SEQ = 400 and SEQ = 500 arrive, the receiver is still expecting SEQ = 200 and again responds by repeating the ACK of 200.

After receiving 3 dupACKs (4 ACKs) for 200, the sender retransmits the missing segment with SEQ = 200. Here the behavior of the receiver is implementation dependent. The receiver is permitted to discard all segments that arrive out of order (in this case, the receiver is expecting SEQ = 200 so a segment with SEQ = 300, 400 or 500 can be deleted). But most receivers (with a large enough buffer) will save the out-of-order segments. In the example, the receiver buffered the last 3 segments, so when the retransmitted segment with SEQ = 200 arrives, it sends a cumulative ACK with ACK = 600, which means that all previous data was received. In this case, the sender can transmit SEQ = 600 and does not need to retransmit the segments that arrived out of order.

Slide 14 Congestion Avoidance

Another way to avoid underutilization in the TCP slow start method is by moderating the exponential growth. The sender defines a new parameter called **slow start threshold** (**`ssthresh`**). As long as the **`cwnd < ssthresh`**, the data rate grows exponentially, by adding the number of ACKed bytes to the previous cwnd value. When **`cwnd > ssthresh`**, the data rate grows linearly, by adding 1 to the previous cwnd value.

Another modification called **Fast Retransmission** is that when there are 3 dupACKs, ssthresh is reset to half the previous cwnd and cwnd set to the new ssthresh + 3 (instead of being reset to 1). Only on a timeout is the cwnd reset to 1, and the RTO is doubled to avoid future timeouts.

On the 3rd dupACK, the sender retransmits the last unACKed packet and then sets

```
ssthresh ← cwnd / 2
cwnd ← ssthresh + 3
```

The sender waits for 1 RTT to see if the missing packet is ACKed.

If there are more dupACKs (instead of ACK for the missing packet), then the sender performs cwnd++ for each dupACK and resends the missing packet.

Slides 15 – 23 TCP Sender with Reno

TCP Reno is the basis for most implementations of TCP in use today. It includes, slow start, fast retransmit, and congestion avoidance.

Adding congestion control to the simplified TCO sender, the simplified Reno sender becomes

```
// initialize variables and parameters
       SEQ = ISN + 1
       SendBase = ISN + 1
       InFlight = 0
       cwnd = 1 MSS
       Set ssthreshold large (local policy)
       RTO = timeout interval
on (new data from application)                  // Handle data to be transmitted
       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
if (receive ACK = y)                            // Handle an arriving ACK with value y
       stop timer
       if (y > SendBase)                        // check that y is a new ACK
              dupACK = 0
              newACKs = y – SendBase            // number of bytes ACKed this time
              SendBase = y                      // number of bytes ACKed so far
              InFlight = InFlight – newACKs     // number of bytes waiting for ACK
              if (cwnd < ssthresh)              // below threshold
                     cwnd = cwnd + newACKs      // exponential growth
              else                              // above threshold
                      cwnd = cwnd + 1 MSS       // linear growth
              if (InFlight > 0) timer = RTO     // set RTO if still waiting for ACKs

       else                                     // ACK is not a new ACK
              dupACK++                          // count dupACKs
              if (dupACK = 3)                   // fast retransmit trigger
                     SEQ = SendBase = min{unACKed SEQ} // just 1 packet
                     resend                     // resend last unACKed segment
                     timer = RTO                // reset timer
                     ssthresh = cwnd / 2        // update threshold
                     cwnd = ssthresh + 3        // reduce cwnd
                     wait 1 RTT                 // wait for ACK of resent packet
              if (dupACK > 3)
                     cwnd = cwnd + 1 MSS and resend again     // to permit retransmissions
if (timeout)                                    // Handle a timeout
       SEQ = SendBase = min{unACKed SEQ}
       resend                                   // resend last unACKed segment
       ssthresh = cwnd                          // update threshold
       cwnd = initial cwnd = 1 MSS              // reduce cwnd
       RTO = 2 * RTO                            // update RTO
       timer = RTO                              // reset timer
```

Adding congestion control to the simplified TCO receiver, the simplified Reno receiver becomes

```
// initialize variables and parameters
```

```
        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)      // Handle delayed ACKs to be sent
        Send ACK = expected with updated RecvWindow
        ACK delay timer = ack_delay             // reset ACK delay timer
        ack_buffer = 0                          // reset delayed ACK number
if (receive SEQ = x)                            // Handle received segment with SEQ value x
        if (x = expected && error-free)         // correct order and not corrupted
                expected = expected + length(data)       // prepare next ACK value to send
        if (NACK = 1)                           // If this segment repairs a previous error
                Send ACK = expected with updated RecvWindow     // send immediate ACK
                ACK delay timer = ack_delay     // reset ACK delay timer
                ack_buffer = 0                  // reset delayed ACK number
                NACK = 0                        // reset NACK flag
        else if (ack_buffer < ack_max)          // If delayed ACK buffer not full
                nextACK = expected              // prepare next ACK value to send
                ack_buffer++                    // update delayed ACK number
        else if (ack_buffer = ack_max)          // If delayed ACK buffer is full
                Send ACK = expected with updated RecvWindow     // send immediate ACK
                ACK delay timer = ack_delay     // reset ACK delay timer
                ack_buffer = 0                  // reset delayed ACK number
        else                                    // Handle received segment with error or bad SEQ value x
                Send ACK = expected with updated RecvWindow     // send immediate ACK of expected
                ACK delay timer = ack_delay
                NACK = 1                        // set error flag
```

Slides 20 to 23 show an example of Reno TCP congestion control. The SEQ number is 1 MSS times the segment number SEQm, and the ACK number is 1 MSS times the number of the ACK segment ACKm. The receiver sends a delayed ACK for every 4 segments received, unless the delayed ACK timer times out or there is an error. The ssthresh is 32.

In the first line, the cwnd is 1 and the sender transmits SEQm = 1, so the in-flight counter is 1. Then the receiver sends ACKm = 2 (SEQm = 1 was received and now expecting SEQm = 2). This is an ACK of 1 segment. The sender adds 1 (new ACKs) to cwnd an resets in-flight to 0.

Next, the sender transmits 2 segments (SEQm = 2 and SEQm = 3) and in-flight is 2. The receiver sends 2 new ACKs by writing ACKm = 4 (received SEQm = 2 and SEQm = 3, and now expecting SEQm = 4). The sender adds the 2 new ACKs to cwnd and resets the in-flight counter.

Now the sender and receiver enter a "**self-clocking**" condition. The cwnd number rises exponentially with each received ACK. But the sender receives another ACK for every 4 segments it transmits. So, when the sender transmits 8 segments at one time (segments 8 to 15), it first receives an ACK for 4 segments (8 to 11) and then another ACK for 4 segments (12 to 15).

At the bottom of slide 21, cwnd reaches the threshold value of ssthresh = 32. So, on slide 22, for each new ACK the sender raises the cwnd by 1.

On slide 22 the sender does not transmit new segments so the in-flight drops by 4 on each ACK. At the top of slide 23, with cwnd = 38 the sender can transmit 26 new segments but transmits 8 new segments (SEQm = 68, 69, ... , 75) and raises in-flight from 12 to 20. The receiver sends an ACK of 4 more segments, with ACKm = 60 (previously sent at the top of slide 3). The sender adds 1 to cwnd and removes 4 from in-flight.

But now the receiver again sends ACKm = 60 another 3 times (3 dupACKs). These carry 0 new ACKs. This means that segment SEQm = 60 was lost or corrupted, but the sender does not know if additional segments were lost. The sender updates ssthresh to cwnd / 2 = 39/2 = 19, and sets cwnd to ssthresh + 3 = 19 + 3 = 22. The sender retransmits segment SEQm = 60 and waits 1 RTT.

The receiver now sends an ACK with ACKm = 76 (all segments up to SEQm = 75 correctly received). This is a cumulative ACK, so the sender can continue transmitting new segments from SEQm = 76 (it does not need to retransmit segments 68 to 75). Because cwnd = 22 is larger than ssthresh = 19, the 16 new ACKs in the cumulative ACK add 1 to cwnd which rises to 23. Since the in-flight counter is 0, the sender can transmit 23 segments.

Slide 24 Reno: First Alternative Scenario

In this case, the sender again starts from cwnd = 1, first sending 1 packet and then sending 2 packets (2 and 3). But this time the ACK following the second group is only for 1 packet. The sender updates cwnd to 3 but since InFlight = 1 it can only send 2 new packets (SEQ = 4, 5). Following each of these packets the sender receives a dupACK for 3. Since there are only 2 dupACKs (and no more to come, since cwnd = InFlight and so the sender must wait), there is a timeout, returning cwnd to 1 and ssthresh to 3 (the old cwnd).

Slides 25 – 26 Reno: Second Alternative Scenario

In this case, the sender again starts from cwnd = 1, first sending 1 packet, then sending 2 packets (2 and 3), and then sending 4 packets (4 – 7). This time the ACK following the third group is only for 3 packets (4 – 6). The sender updates cwnd to 7 but since InFlight = 1 it can only send 6 new packets (SEQ = 8 – 13). Following the first 5 of these packets the sender receives a dupACK for 7. After the 3rd dupACK the sender retransmits the missing packet 7 and reduced ssthresh to 3 and cwnd to 3+3 = 6. Since InFlight > cwnd, the sender cannot transmit, but there are more dupACKs arriving from the last 7 packets sent. Each dupACK > 3 causes cwnd++ so when cwnd > InFlight the sender can transmit packet 7 again. This time the sender receives a cumulative ACK for all the sent packets.

Slide 27 Selective Acknowledgment Option

The **selective acknowledgment (SACK)** option in TCP permits a partial ACK for a series of segments with gaps caused by lost segments. This is an option negotiated between hosts when both are capable of supporting the service. SACK acknowledgements use standard values for the SEQ and ACK fields, but include option fields showing data bytes that have arrived correctly.

For example, suppose that the last ACK received by the sender was 5000, and then the sender transmits 8 segments of 500 data bytes/segment (a total of 4000 additional bytes). If all the segments arrive with no errors, then the receiver will send ACK = 5000 + 8 × 500 = 9000.

In case 1, the first 4 segments are received correctly and the last 4 are dropped (lost). The receiver returns a normal ACK = 5000 + 4 × 500 = 7000. After a timeout, the sender will resend the last 4 segments and there is no need for a SACK option field.

In case 2, the first segment is lost and 7 segments are correctly received. For each received segment, the receiver returns a segment with ACK = 5000 and a SACK option field. The option field contains the start and end ACK for one received segment. The first segment (with

SEQ = 5000) is lost. The second segment with SEQ = 5500 contains 500 bytes. The receiver sends a segment with ACK = 5000 (to signify that SEQ = 5000 is still expected), and an option field with start = 5500 and end = 5999 (500 bytes received). The third segment with SEQ = 6000 also contains 500 bytes. The receiver sends a segment with ACK = 5000 (to signify that SEQ = 5000 is still expected), and an option field with start = 5500 and end = 6499 (a total of 1000 bytes received).

Slides 25 – 26 Active Queue Management (AQM)

In standard buffer queue management, a full buffer causes the receiver to drop (delete) excess packets. At the sender, there is no ACK for the dropped packet, which causes dupACKs or a timeout. The dupACKs or timeout cause retransmission of the dropped packet, but also cause the transmission rate to be lowered through cwnd. Thus dropped packets are a signal of congestion.

In **Random Early Detection (RED)**, the router detects congestion early. When the buffer level reaches about 85% of capacity, the router starts to drops random packets. The sender sees dupACKs or a timeout, assumes congestion, and lowers the cwnd before the congested buffer is completely full.

The RED algorithm is shown in slide 26. The parameters are:

- max_p is the maximum probability (0.1 to 0.5) of marking a packet as dropped
- min_{th} is a minimum threshold for the buffer queue, approximately 5
- max_{th} is a maximum threshold for the buffer queue, approximately 30
- p_b is a scaled marking probability found as $p_b = max_p \times (\text{average buffer level} - min_{th}) / (max_{th} - min_{th})$
- p_a is another scaled marking probability found as $p_b / (1 - \text{count} \times p_b)$, where count is the number of consecutive that were packets dropped

For each packet arrival,

- The buffer calculates avg = average queue size
- If $min_{th} \leq avg < max_{th}$ then the buffer calculates the probability p_a and marks the arriving packet for drop with probability p_a.
- If $max_{th} \leq avg$ then the buffer marks the arriving packet for drop with probability 1.

Notice that if the average buffer level is between the upper and lower thresholds, then $0 < (\text{average buffer level} - min_{th}) / (max_{th} - min_{th}) < 1$ so $0 < p_b < p_{max}$.

If p_b is stable, $1 - \text{count} \times p_b$ decreases with each consecutive dropped packed, and so p_a will increase as the number of consecutive dropped packets goes up.

Slides 27 – 44 AQM with ECN

Explicit Congestion Notification (ECN) is a method for using the RED algorithm in buffers to alert TCP to lower the transmission rate. In ECN, the IP router predicts congestion using the RED algorithm, but it marks the packets without actually dropping packets. The router indicates congestion in the direction from the sender to the receiver in the IP header. At the receiver the

IP software agent alerts TCP about the congestion, and TCP indicates congestion to the sender in the header of the TCP ACK.

Slide 28 shows the ECN fields in the IP datagram header. Two ECN bits are used to indicate 4 conditions:

00	Not ECN capable	Set for retransmissions to indicate that this packet cannot be randomly dropped
01	ECT(0) — ECN Capable Transport (0)	The sender alternates 01 and 10 to permit error control on the ECN mechanism
10	ECT(1) — ECN Capable Transport (1)	
11	CE (Congestion Experienced)	Set to indicate congestion in the direction of the IP datagram

Slide 29 shows the NS, CWR, and ECE flags in the TCP header. TCP uses the ECE flag to indicate congestion in the opposite direction, and uses CWR to allow the sender to acknowledge receipt of the ECE flag. The NS bit is used for error checking.

Slide 30 shows the setup of ECN in the TCP initial handshake. To use ECN, the SYN packet has ECE and CWR set (the IP packet sets ECN = 01), the server responds with ECE set in the SYN-ACK packet (the IP packet sets ECN = 10).

Slides 31 to 41 show the indication of congestion:

- Slide 31 shows no congestion. In TCP, ECE = CWR = 0, and in IP, ECN = 01 (ECT).
- Slide 32 shows no congestion in the reverse direction.
- Slide 33 shows the onset of congestion. In TCP, ECE = CWR = 0. In the IP header from the host to the router, ECN = 10 (the alternate ECT). But the router is starting to experience congestion and changes ECN to 11 (CE) in the IP header.
- Slide 34 shows the reverse direction. The TCP header now sets ECE = 1 CWR = 0 to report congestion to the sender TCP agent. The IP headers are set to ECT because there is no congestion in the reverse direction.
- Slide 35 shows that the sender lowers cwnd and sets CWR = 1 to acknowledge the congestion indication from the receiver side. ECE is 0 because there is no congestion to report in the reverse direction. In the IP headers, the first is ECT (no congestion from sender to router) and the second is CE (continued congestion from router to receiver).
- Slide 36 repeats the conditions of slide 34 (because the congestion continues).
- Slide 37 repeats the conditions of slide 35 (because the congestion continues).
- Slide 38 repeats the conditions of slide 36 (because the congestion continues).
- Slide 39 shows that the congestion has been cleared. The CWR is still 1 to acknowledge the previous congestion indication from the receiver side. This time the router sets the ECN bits to ECT (no congestion).
- Slide 40 shows the reverse direction with no congestion reporting. The TCP header has ECE = CWR = 0 and the IP headers have ECN set to ECT.
- Slide 41 shows that the congestion has been cleared, with ECE, CWR and ECN cleared. The sender can now raise the cwnd.

Slides 42 to 44 show performance results of RED and ECN. Goodput is the throughput of good packets (actual received data over TCP). Slide 42 shows that goodput drops with the number of

flows (TCP sessions running through the same router buffer) for RED and ECN with $p_{max} = 0.1$, but improved goodput for ECN with $p_{max} = 0.5$. Slide 43 shows delay, which is approximately the same for RED and ECN, but dependent on flow types (robust flows, such as large data transfers, are relatively insensitive to congestion, while fragile flows, such as video streaming, may stall during congestion). Slide 44 shows the dependence on RED parameters, with ECN outperforming pure RED.

Slides 45 – 47 ECN Nonce

The NS bit in the TCP header is used to detect errors in the ECN mechanism. Suppose that an unscrupulous or poorly implemented receiver clears the ECN-Echo signal (CWR) so that no congestion signals are ever sent to the sender. This can gives that receiver an advantage over connections that behave properly because it never warns the sender to lower the data transmission rate through cwnd.

To detect cheating, the NS bit (nonce) is set according to the history of the ECT signals. The sender and receiver keep a cumulative sum modulo 2 of all received nonces for all ACKed packets. The NS bit is this running sum (or 0 if the CE bit was set).

Slide 46 shows an honest receiver.

- NS is initialized to 1 in the SYN-ACK handshake.
- SEQ_1 arrives with ECT(0) so the nonce is (1 + 0)%2 = 1.
- SEQ_2 arrives with ECT(0) so the nonce is (1 + 0)%2 = 1.
- SEQ_3 arrives with ECT(1) so the nonce is (1 + 1)%2 = 0.
- The ACK of SEQ_3 is sent with NS = 0 and the sender sees the correct nonce for SEQ_3.
- SEQ_4 arrives with ECT(0) so the nonce is (0 + 0)%2 = 0.
- SEQ_5 is sent with ECT(1) but arrives with CE so the nonce is 0.
- The ACK of SEQ_6 is sent with NS = 0 and with ECE = 1, so the sender knows that there was a CE that set the nonce to 0 (otherwise it would have been 1).

Slide 47 shows a cheating receiver.

- NS is initialized to 1 in the SYN-ACK handshake.
- SEQ_1 arrives with ECT(1) so the nonce is (1 + 1)%2 = 0.
- SEQ_2 arrives with ECT(0) so the nonce is (0 + 0)%2 = 0.
- SEQ_3 is sent with ECT(1) but arrives with CE so the nonce is 0.
- The receiver does not set ECE = 1 (to cheat) and so it does not automatically set NS = 0. The receiver must guess what value of ECT to use for the nonce for SEQ_3. The ACK of SEQ_3 is sent with NS = 0 but the sender sees that the correct nonce should have been 1.
- The receiver will continue to guess the running sums and will be wrong most of the time.