CSE 3231
Computer Networks

Chapter 6
The Transport Layer
part 4 - Flow Control

William Allen, PhD

Spring 2022

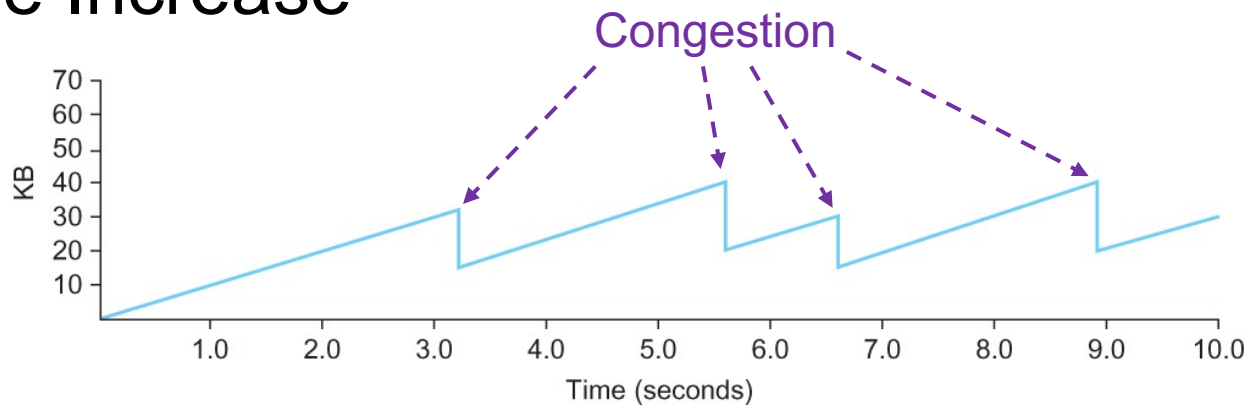
Review TCP Congestion Control

The TCP source sets the *CongestionWindow* based on the level of congestion that it *notices* in the network

- This involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down
- Taken together, the mechanism is called *Additive Increase / Multiplicative Decrease* (abbreviated AIMD)

TCP Congestion Control

Additive Increase



Behavior of TCP congestion control.

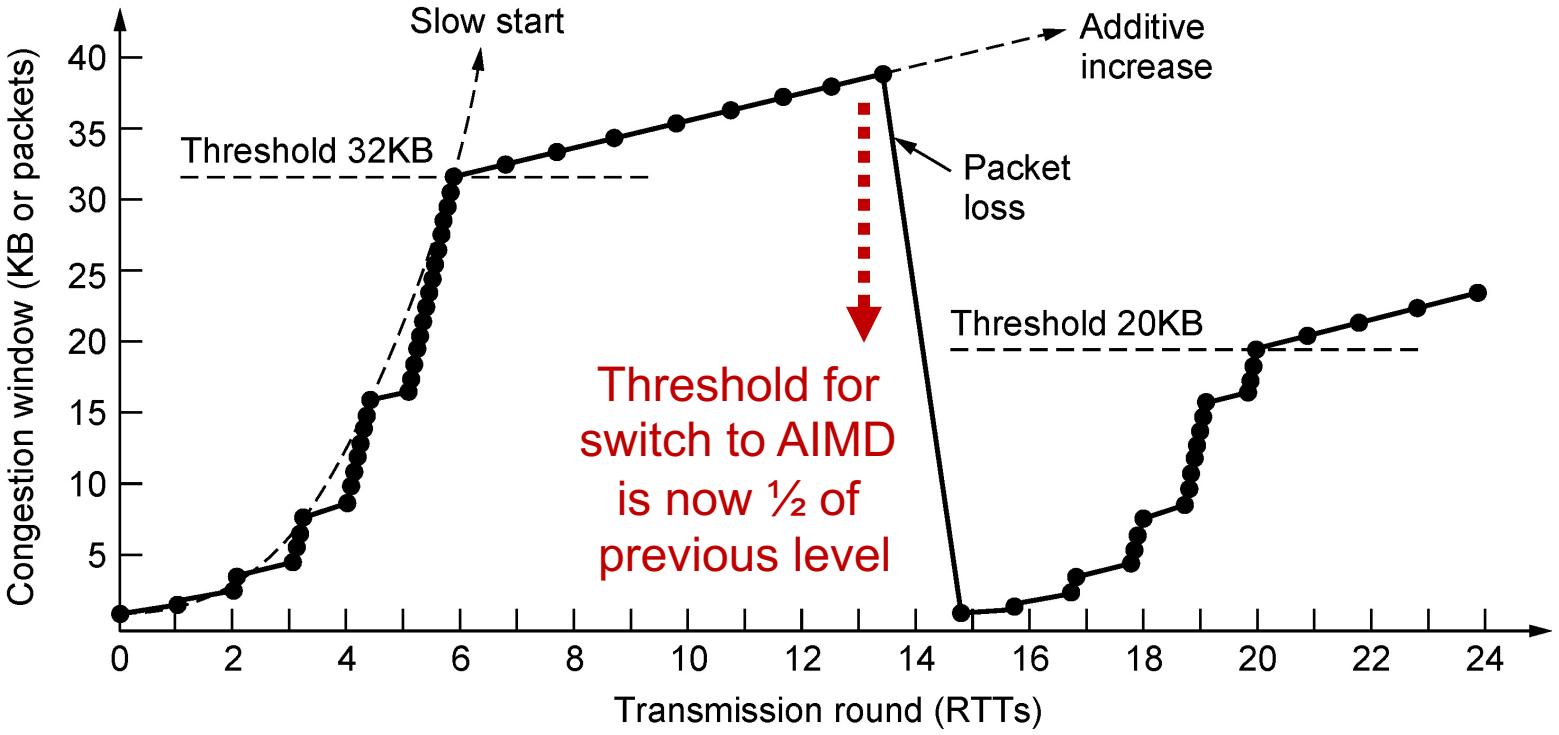
- blue line = value of *CongestionWindow* over time
- Congestion occurs at certain points and AIMD reacts by immediately halving the *CongestionWindow* size then slowly increasing it again

TCP Congestion - Slow Start

- The source starts by setting *CongestionWindow* to the size of one packet (MSS)
- When this packet's ACK arrives, TCP adds 1 to *CongestionWindow* and then sends two packets
- Upon receiving the corresponding two ACKs, TCP increments *CongestionWindow* by 2—*one for each ACK*—and sends four packets
- The end result is that TCP effectively doubles the number of packets in transit every RTT
- This continues until a threshold is reached

TCP Congestion - Slow Start

1. *Slow start* begins by doubling the Congestion Window size
2. When the threshold is reached, it switches to *additive increase*
3. When a packet is lost, window goes to 1 & *slow start* repeats



Slow start followed by additive increase

TCP Congestion - Timeouts

- Because the path from sender to receiver is not known, the sender's packet timeout period must be long enough to allow for longer paths
- If a packet is lost, a number of packets can be sent before the sender's timeout expires
- But, the receiver can't ACK those packets because the missing packet caused a gap in the sequence numbers
- The sender can reach its *CongestionWindow* limit while waiting for the timeout to expire

TCP Congestion - Timeouts

- One solution is for the receiver to send an ACK when each of the later packets arrives, but with the ACK number of the last packet before the lost packet - i.e., *duplicate acknowledgments*
  - This tells the sender that at least one packet hasn't arrived, but since ACK's were sent, this indicates that some of the later packets did arrive
  - The duplicate ACK number also tells the sender which packet wasn't received
- The sender can then *resend* that missing packet *without waiting* for the timeout to expire

TCP Congestion - Fast Retransmission

This feature is called *fast retransmission* and can trigger transmission of dropped packets sooner than the regular timeout mechanism

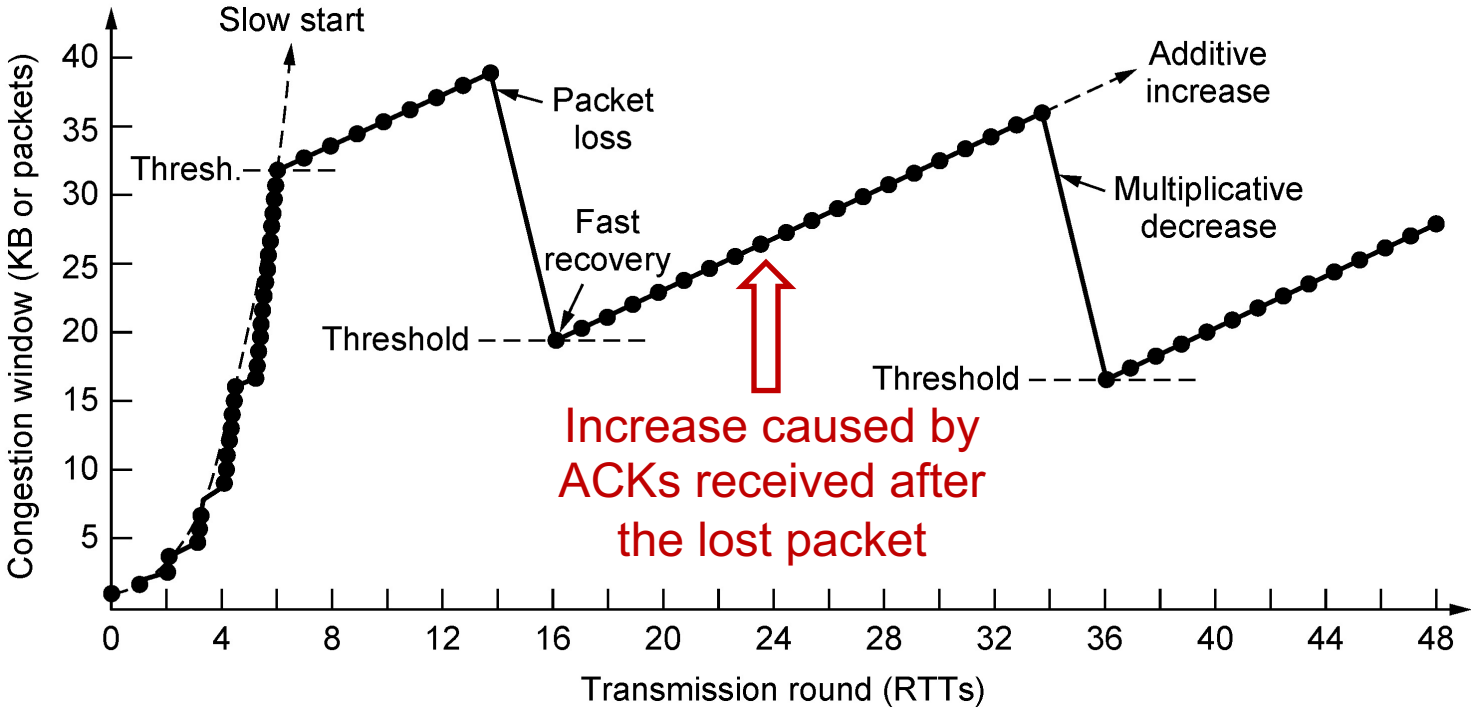
- to make sure a packet was actually lost and not just delayed, fast retransmission is not triggered until *three duplicate ACK's* have arrived
- the sender now knows that a packet was lost and halves the *slow start threshold* and goes into slow start to ramp up more quickly

TCP Congestion - Fast Recovery

- A lost packet decreases the *CongestionWindow* size to *one* and starts slow start
- When the fast retransmission mechanism signals congestion, rather than drop the congestion window further and enter slow start, it is possible to use the ACKs that are still in transit to trigger the sending of new packets
- This approach is called *fast recovery* and effectively removes the slow start phase that occurs when fast retransmit detects a lost packet

TCP Congestion - Fast Recovery

1. *Slow start* ramps up until the threshold is reached and then it switches to *additive increase* until a packet is lost
2. The lost packet triggers a halving of the slow start threshold
3. *Fast recovery* avoids dropping the *CongestionWindow* to one



Fast recovery and the sawtooth pattern of traffic increase

Official Versions of TCP Related to Those Features

- The version that used three duplicate ACK's to trigger *fast retransmit* and resets to slow start is called TCP Tahoe
- The version which waited for three duplicate ACKs but skipped slow start to enter *fast recovery* is called TCP Reno
- There is also an approach that looks at packet delays instead of dropped packets and it is called TCP Vegas

TCP Can Be Inefficient

One issue with TCP flow control is that some interactions can lead to unusual situations

- For example, an application may produce data slowly and the *sender* will transmit a large number of very small segments - e.g., from SSH or telnet
  - sending 1 character requires 41 bytes (IP & TCP headers)
  - the receiver ACK's the character (returning 40 bytes)
  - SSH and telnet will echo the character (41 more bytes)
  - the echo is ACK'd (40 more bytes)
- This can use up to 4 packets to send one character and is referred to as the "*Silly Window*" syndrome

TCP's "Silly Window"

One solution is to delay acknowledgements to see if they can be aggregated

- a 500ms delay might allow reception of several packets before sending an ACK, reducing overhead

Another solution is called Nagle's algorithm

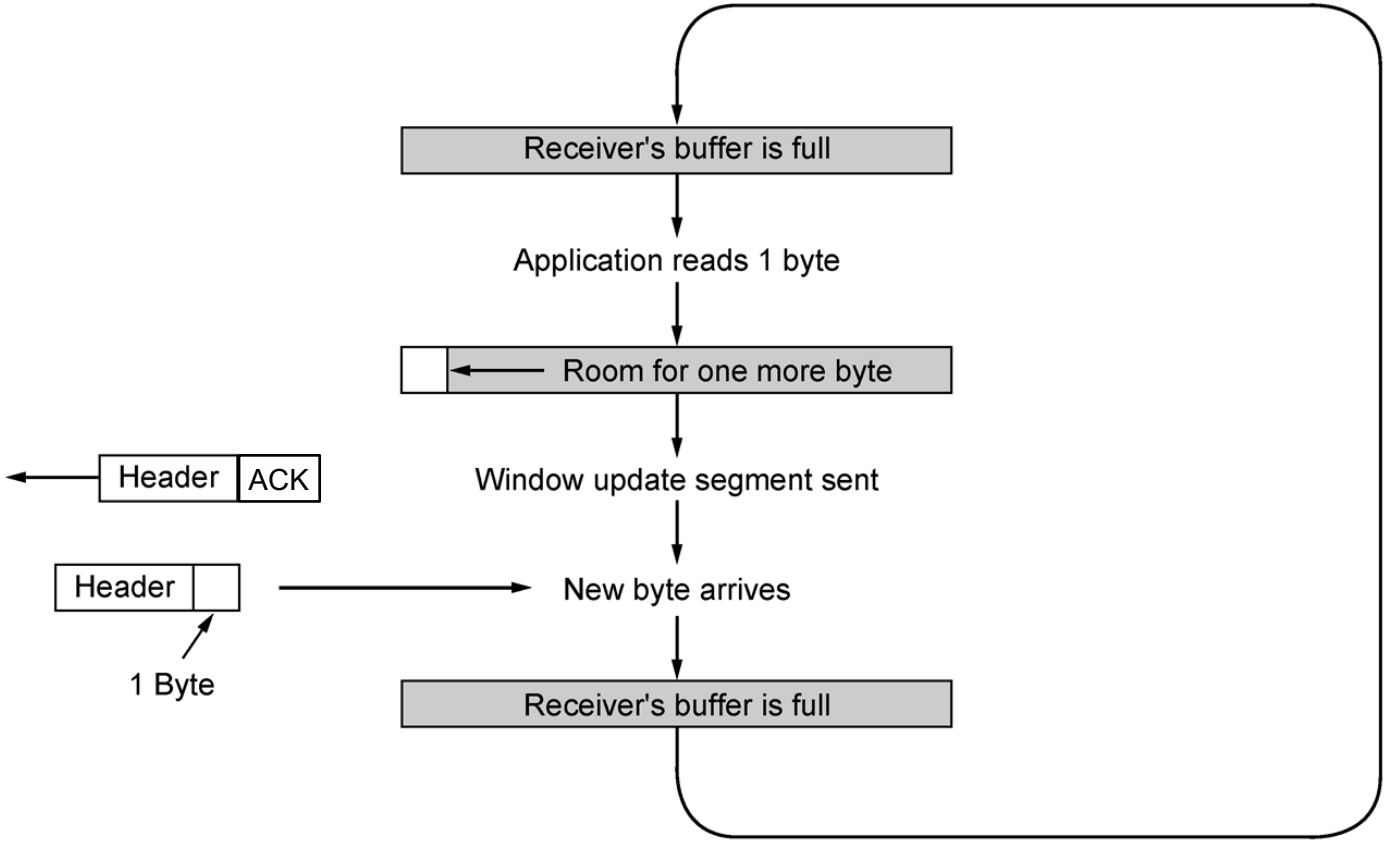
- Send the first byte to keep the connection alive
- Then buffer the rest of the data until a full segment is gathered before sending additional data
- Fewer packets are in transit, reducing overhead
- However, it may need to be disabled for certain applications like games or interactive sessions

TCP's "Silly Window"

Another version of the *Silly Window* happens at the receiver where the receiver's application is slow in processing and the buffer becomes full

- Each time the application reads one byte, the receiver sends an ACK with *AdvertisedWindow* = 1
- The sender can only send one byte and the buffer will be full again, this can continue for some time

TCP's "Silly Window"



Silly window syndrome.

TCP's "Silly Window"

Another version of the *Silly Window* happens at the receiver where the receiver's application is slow in processing and the buffer becomes full

- Each time the application reads one byte, the receiver sends an ACK with *AdvertisedWindow* = 1
- The sender can only send one byte and the buffer will be full again, this can continue for some time

Clark's algorithm tries to reduce this by making the receiver wait until it has a larger amount of space in its buffer before it sends an ACK

- This lets the sender send a larger block of bytes too

TCP's "Silly Window"

The length of the delay before sending an ACK is critical because too long a delay will cause the sender to timeout and resend the packet

- Nagle's and Clark's approaches work well together because Nagle's approach waits until it gathers more data before sending, giving time for the receiving application to clear a larger space in the receive buffer

The Timeout Period Depends on the Transmission Media

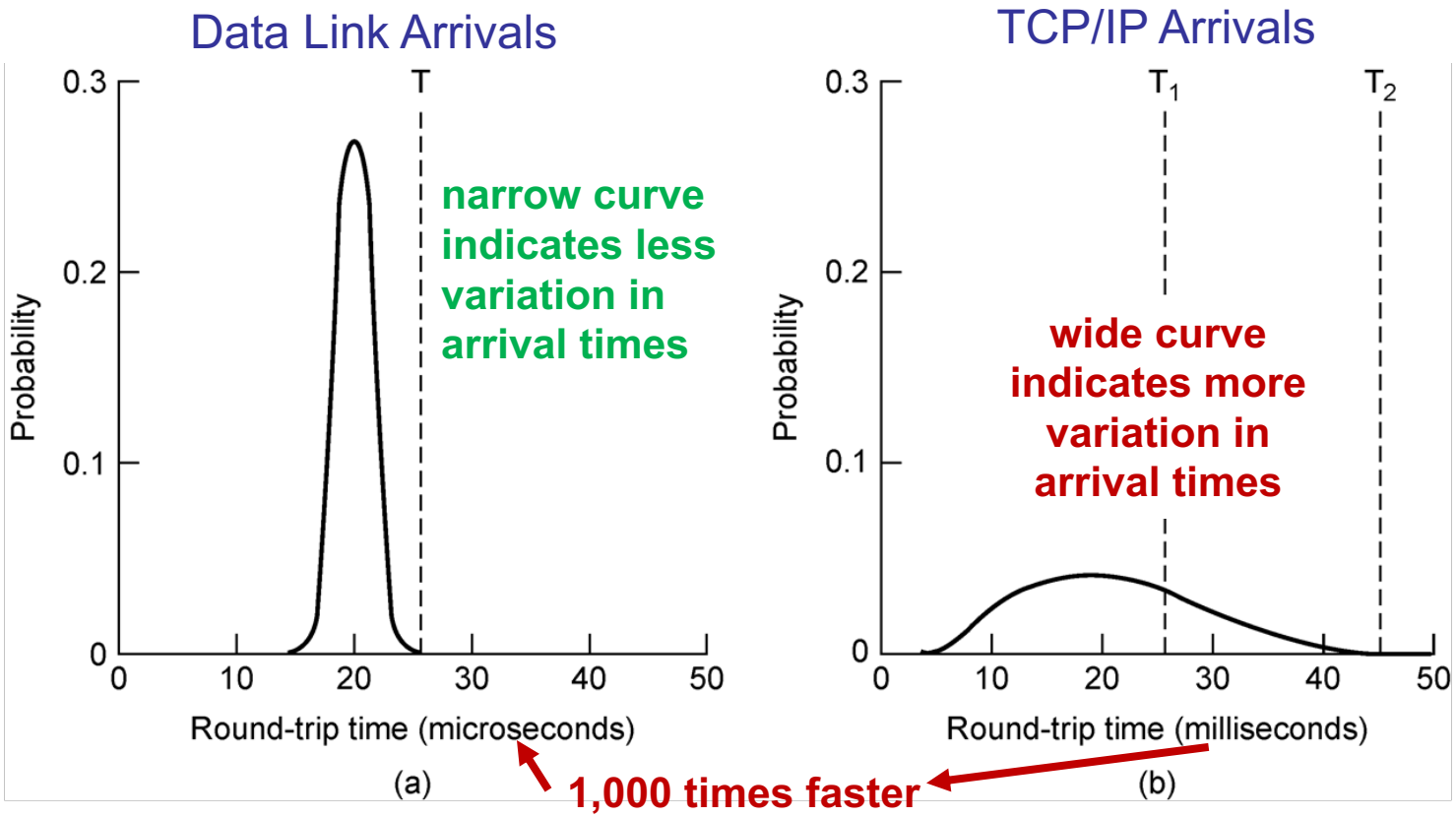
The timeout periods for lost frames at the Data Link layer are very different from the timeout period for packets sent across the Internet

- The Data Link layer operates across short links and normally has only one node transmitting
- Round Trip Time (RTT) in a LAN is short, normally in microseconds, versus much longer times across the Internet, normally measured in milliseconds
- The *variance* (how much it changes over time) in RTT over a LAN is also much smaller than for traffic across the Internet

The Timeout Period Depends on the Transmission Media

- Just as with wired LANs, 802.11 operates over a short distance and also has an RTT that is measured in microseconds
- In both cases, this leads to a more predictable RTT and that allows calculation of a more accurate timeout period
- This is *not so easy* with TCP networks since paths can be all different lengths and can also change from connection to connection and from packet to packet

Delivery Times for Frames and Packets



(a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

Calculating the TCP Time Out Period

The *Retransmission TimeOut* (RTO) for a TCP connection is based on the *Round Trip Time* (RTT) for that connection

- Fortunately, once established, path lengths don't change too often, mainly because links are more reliable than they used to be
- At connection time, the sender and receiver determine the RTT for the connection and then the sender uses that to calculate the RTO
- However, delays can happen over time, so the sender calculates an *average RTT* to deal with them

Determining the RTT

- The goal is to calculate an average RTT, but to avoid altering it too much based on short-term changes that are due to recent delays
- The sender and receiver both need RTT so they put timestamps in the options field to track when packets were sent and received
- A *Smoothed RTT* (SRTT) is calculated based on both the SRTT *averaged* over time and the *most recent* RTT

$$SRTT = \alpha * SRTT + (1-\alpha)RTT \text{ where } \alpha = 0.9$$

Determining the RTT

- Over time, the *Smoothed RTT* calculation was revised to include the *variance* in RTT instead of just using the average RTT time
  - Variance measures how much the RTT changes over time and proved to work well over a wide range of TCP devices

$$VarRTT = \beta * VarRTT + (1-\beta) * |SRTT - RTT| \text{ where } \beta = 0.75$$

- From this, the *Retransmission TimeOut* (RTO) can be calculated as follows:

$$RTO = SRTT + 4 * VarRTT$$

(multiplying by 4 is based on experiments)

Additional Timers in TCP

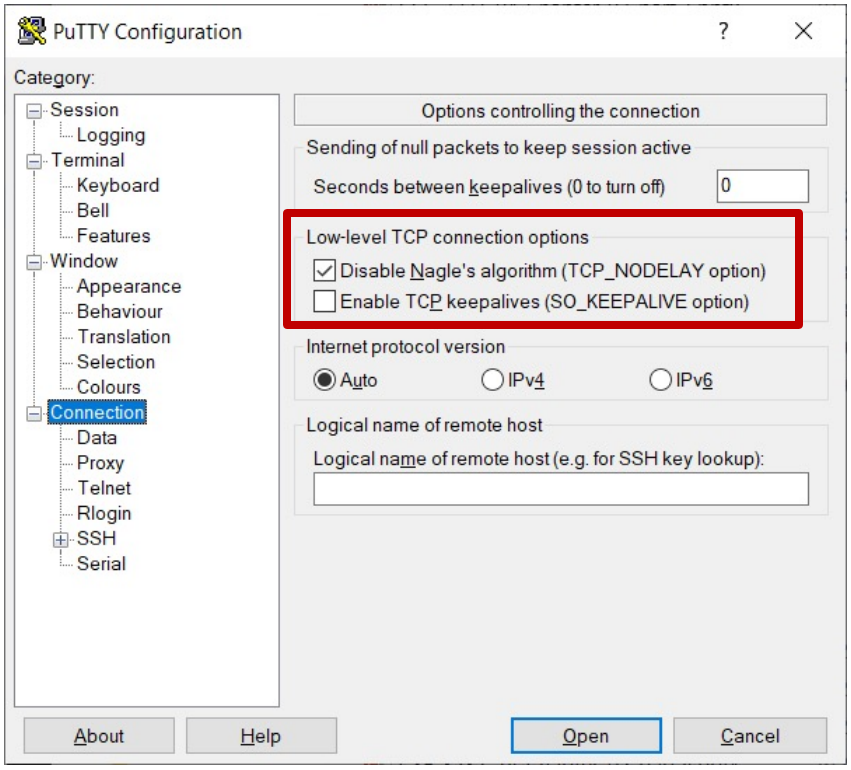
- If a receiver gets a packet that fills the buffer, it will send an ACK with a *AdvertisedWindow* of 0 bytes to prevent more data from arriving
  - When the application clears space in the buffer, the receiver can update the *AdvertisedWindow* to let the sender know it can transmit more data
  - But, if that update is lost, the sender and receiver will deadlock, each waiting on the other
  - The sender has a *persistence timer* that tells it to periodically check with the receiver to see if the buffer now has any available space
    - The receiver replies with the latest *AdvertisedWindow*

Additional Timers in TCP

- If a connection has been idle for a long time, the endpoints don't know if the other end is offline and they should close the connection
- A *keepalive timer* is reset after each packet arrives and when it expires, the node sends a packet to see if the other end is still online
- If *no ACK arrives*, the node that sent the packet "knows" to *close the connection*
  - Sometimes a client needs this to stay connected to a server that has been configured to quickly drop idle connections

Keepalive and Nagle's Algorithm

PuTTY (and similar SSH/Telnet clients) allow users to select these TCP options



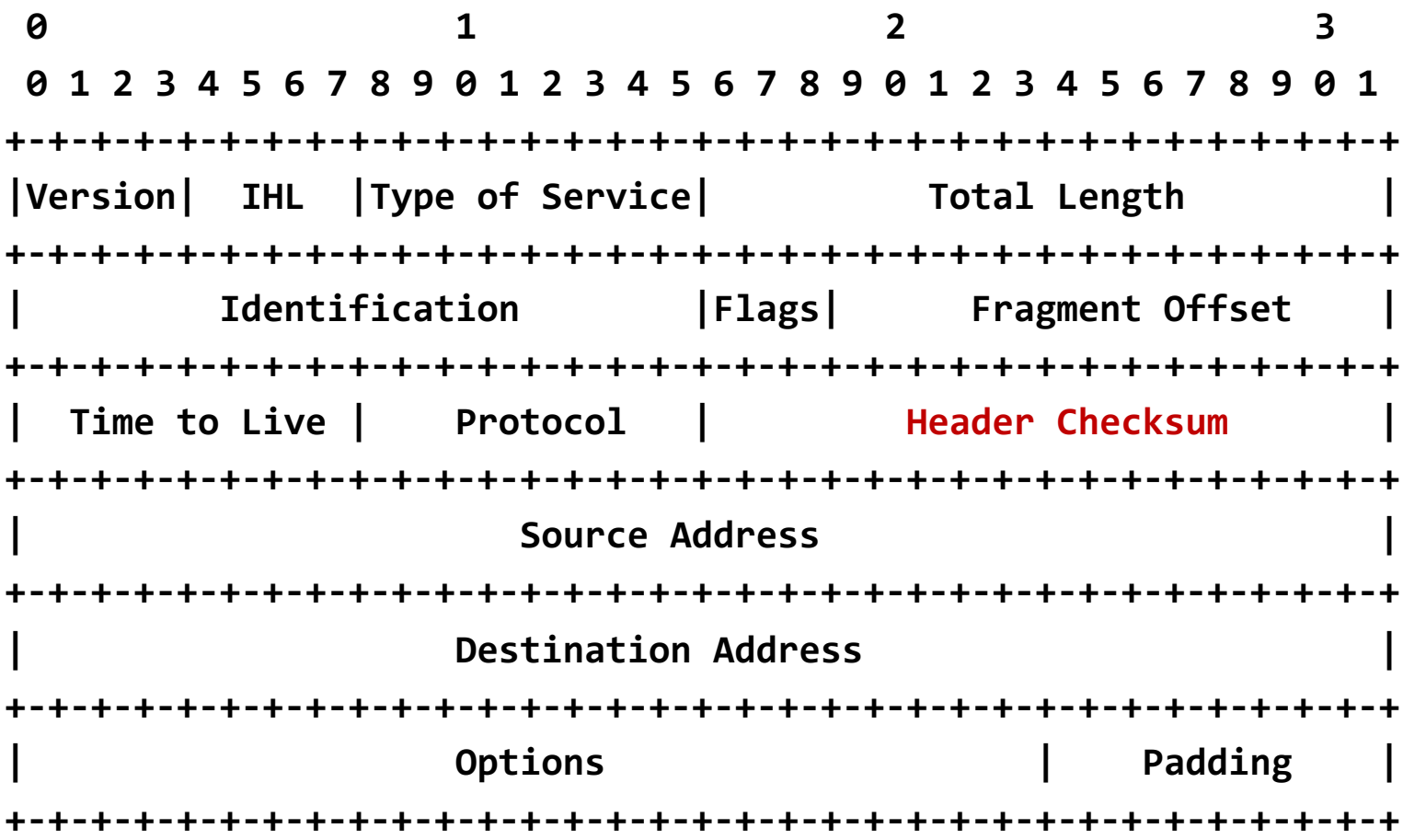
IP Packet Checksum?

- The Ethernet protocol does error checking over the entire frame and the UDP/TCP protocols error check their header and their data segment
- *What about the IP protocol?*
- To check on this, I went to the official documents in the Internet RFC (Request For Comment) archive: https://tools.ietf.org/html/
  - It contains pages for the entire IP range from the original protocol proposals to the latest version (IPv6)
  - IEN # 2 (1977) proposes TCP as a separate protocol from TCP, RFC 791 presents the official IPv4 version

RFC 791 - Internet Protocol

- This document, created in 1981, finalized the original documentation on IPv4
  - It explains the reasons for IP, how it interfaces with other network protocols, and how it handles traffic
  - It describes addressing, fragmentation and routing
    - other RFCs cover the details of routing protocols
  - It also describes the IP header format in detail with definitions of terms and possible options
  - Because Internet applications didn't handle graphics well at that time, diagrams are all created using ASCII characters instead of line drawings or images

RFC 791 - Internet Header Format



RFC 791 - Internet Header Format

The document describes the IP Checksum as:

Header Checksum: 16 bits

A checksum on the header only. Since some header fields change (e.g., time to live), this is *recomputed* and verified at each point that the internet header is processed.

The checksum algorithm is:

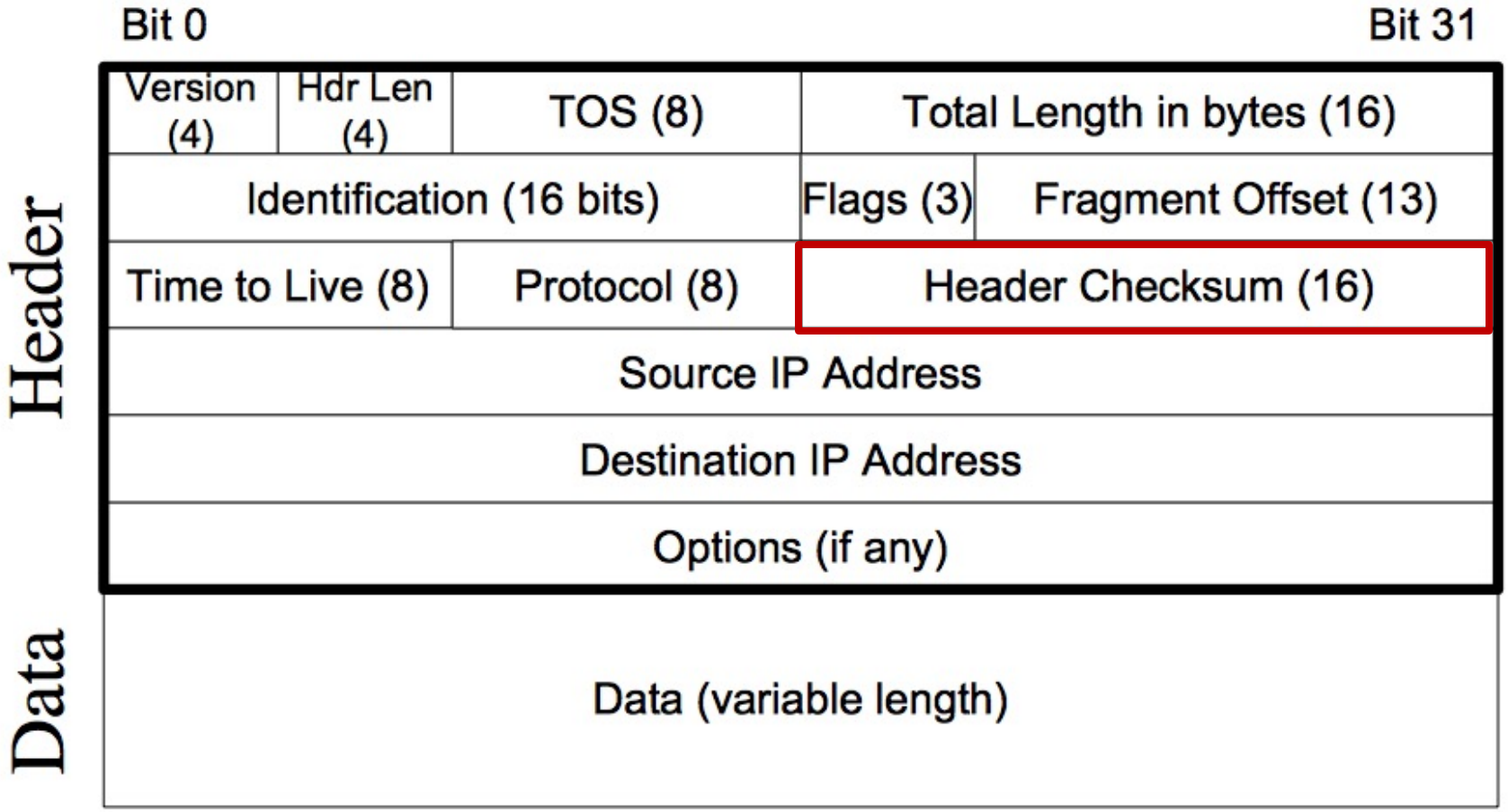
The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

This is a simple to compute checksum and experimental evidence indicates it is adequate, but it is provisional and may be replaced by a CRC procedure, depending on further experience.

Internet Checksum Algorithm

- The Internet Checksum is based on summing the numeric values of the data in a range
  - Consider the data as a sequence of 16-bit integers
  - Sum them using *one's complement* arithmetic
  - The one's complement of the result is the checksum
- The Network Layer (IPv4 packet) uses this method to detect errors in the packet header
  - Examples shown in the following slides
  - During the calculation, it is assumed that the Header Checksum field is all 0's

The IP Header Checksum



Internet Checksum Algorithm

- In *one's complement arithmetic*, a negative integer $-x$ represents the complement of x;
  - Each bit of x is inverted, changing all 1's into 0's and all 0's into 1's

    x = 10011110

    $-x$ = 01100001 (one's complement of x)

- Note: when adding numbers in one's complement arithmetic, if there is a carry out from the most significant bit, it must be added to the result (see the next slide for an example)

Internet Checksum Algorithm

- Consider, for example, the addition of −5 and −3 in ones complement arithmetic on 4-bit integers
  +5 is 0101, so −5 is 1010; +3 is 0011, so −3 is 1100
  - If we add 1010 and 1100, ignoring the carry, we get 0110, but this is the one's complement of 9 (1001)
- Remember, in one's complement arithmetic, when a carry occurs from the most significant bit, we must increment the result:
  - thus 0110 becomes 0111, as we would expect, since the one's complement representation of −8 is 0111 (by inverting 1000), which is equal to (-5 + -3)

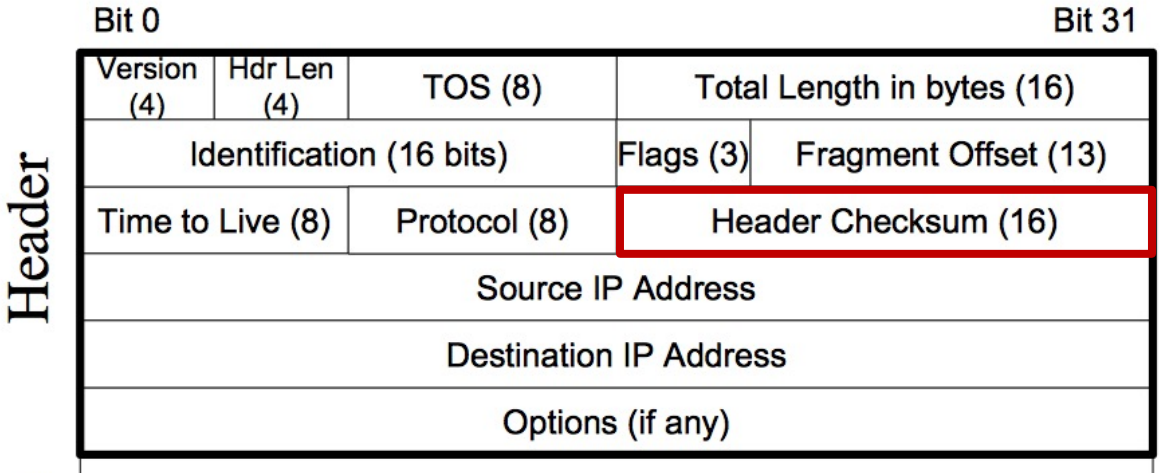
16-bit checksum

- Break the sequence into 16-bit words
- Add the 16-bit values. Each time a carry-out (17th bit) is produced, add that bit into the LSb (the one's digit).
- Once the sum is completed, invert all bits (to get the one's complement of the result) to obtain the checksum

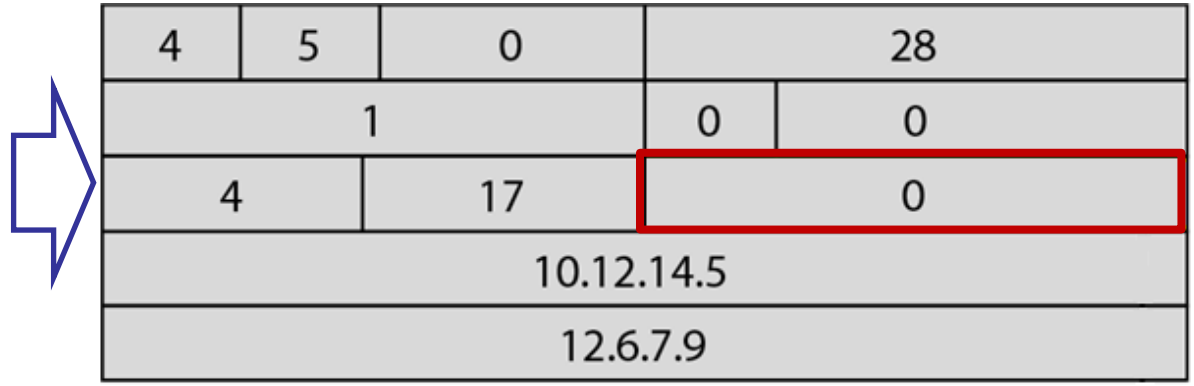
```
      1000 0110 0101 1110
      1010 1100 0110 0000
      0111 0001 0010 1010
      1000 0001 1011 0101

First, we add the 16-bit values 2 at a time:

         1000 0110 0101 1110   First 16-bit value
      +  1010 1100 0110 0000   Second 16-bit value
        ---------------------
        1 0011 0010 1011 1110  Produced a carry-out, which gets added
      +  \----------------> 1  back into LBb
        ---------------------
         0011 0010 1011 1111
      +  0111 0001 0010 1010   Third 16-bit value
        ---------------------
        0 1010 0011 1110 1001  No carry to swing around (**)
      +  1000 0001 1011 0101   Fourth 16-bit value
        ---------------------
        1 0010 0101 1001 1110  Produced a carry-out, which gets added
      +  \----------------> 1  back into LBb
        ---------------------
         0010 0101 1001 1111   Our "one's complement sum"
```

```
0010 0101 1001 1111   Our "one's complement sum"

1101 1010 0110 0000   The "one's complement"
```

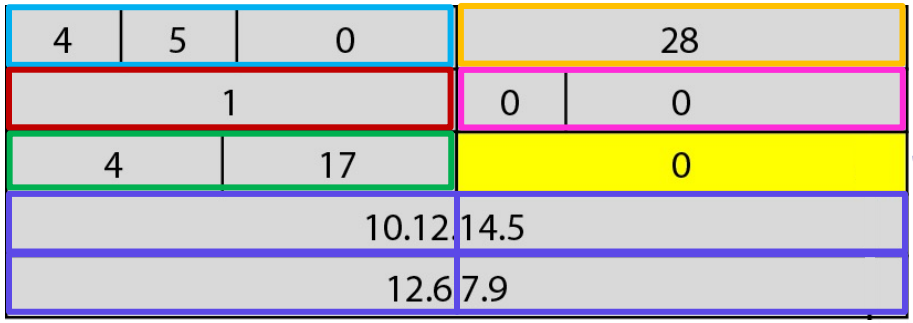
The IP Header: an example



(shown as decimal numbers)



IP Header Checksum Example



(shown as decimal numbers)

Convert the fields into 16-bit values. Then sum all of the 16-bit values, one after another.

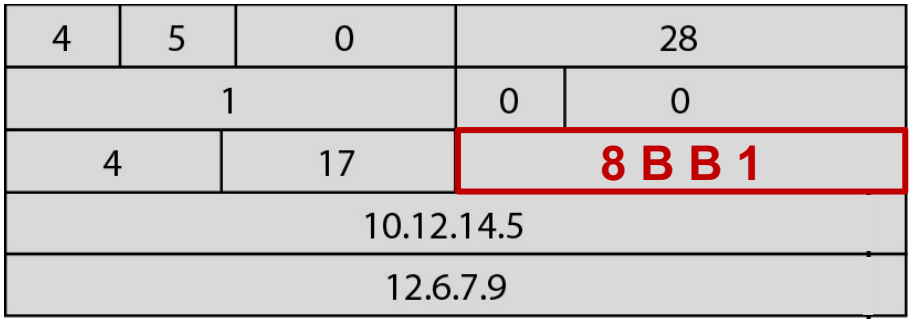
4, 5, and 0 →	4	5	0	0
28 →	0	0	1	C
1 →	0	0	0	1
0 and 0 →	0	0	0	0
4 and 17 →	0	4	1	1
0 →	0	0	0	0
10.12 →	0	A	0	C
14.5 →	0	E	0	5
12.6 →	0	C	0	6
7.9 →	0	7	0	9
Sum →	7	4	4	E

Checksum

(shown as hexadecimal numbers)

7 4 4 E = 0111 0100 0100 1110

IP Header Checksum Example



Convert the fields into 16-bit values. Then sum all of the 16-bit values, one after another. Next, calculate the **1's complement** of the total to get the checksum.

4, 5, and 0 →	4	5	0	0
28 →	0	0	1	C
1 →	0	0	0	1
0 and 0 →	0	0	0	0
4 and 17 →	0	4	1	1
0 →	0	0	0	0
10.12 →	0	A	0	C
14.5 →	0	E	0	5
12.6 →	0	C	0	6
7.9 →	0	7	0	9
Sum →	7	4	4	E

7 4 4 E = 0111 0100 0100 1110

1's complement → 8 B B 1 = 1000 1011 1011 0001

0x8BB1 equals 35761_{10}

Internet Checksum Algorithm

- After inserting the calculated result into the header, in this case:

  8BB1 = 1000 1011 1011 0001

- the new calculation should result in *all zeros* because the checksum value in the header is the 1's complement of the previous result:

  744E = 0111 0100 0100 1110