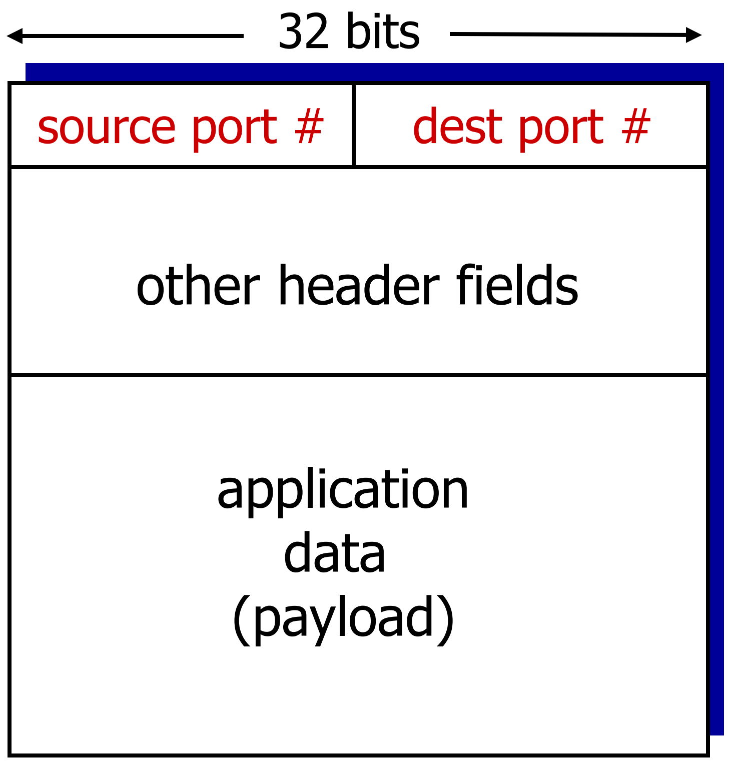
Problem 1

- A UDP socket is fully identified by the destination IP address and the destination port.

- A TCP socket, instead, is fully identified by the source IP address, the source port, the destination address, and the destination port. This happens as TCP establishes a bi-directional full-duplex session between the sender and the receiver.

How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Big Picture

- Multiplexing: many app flows share one lower-layer path.
- Demultiplexing: the receiver uses label fields (protocol, addresses, ports, stream IDs in higher protocols) to route each packet to the correct handler.

- Goal: deliver each arriving transport segment (UDP datagram or TCP segment) to the correct socket in the correct process.
- Tooling: the network stack “demultiplexes” using addresses and port numbers found in headers.

UDP Connectionless demultiplexing

- created socket has host-local port #:

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
```

- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

TCP Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Problem 2

Server host B runs an HTTP server on TCP port 80.

Two clients make HTTP requests:

- Host A uses an ephemeral source port 26145.
- Host C runs two client processes at the same time, using source ports 7532 and 26145 (two separate TCP connections).

What does the server send back on each connection?

- A reply segment always swaps the transport endpoints of the request

Problem 2

Suppose the IP addresses of the hosts A, B, and C are a, b, c, respectively. (Note that a, b, c are distinct.)

To host A: Source port =80, source IP address = B, dest port = 26145, dest IP address = B

To host C, left process: Source port =80, source IP address = B, dest port = 7532, dest IP address = C

To host C, right process: Source port =80, source IP address = B, dest port = 26145, dest IP address = C

Internet checksum: example

example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Problem 3

- 1's complement: Negation is bitwise NOT. Has two representations for zero (+0 and −0).
- 2's complement: Negation is invert bits then add 1. Single zero, asymmetric range.
  - More symmetric, Ubiquitous for signed integers in CPUs, programming languages, filesystems.
- Why 1's complement persists in Internet checksums
  - Historical constraints: early routers/hosts were slow and simple. 1's-complement addition with end-around carry is easy to implement in hardware.

Problem 3

```
                00100011
      +         01001110
-----------------------------
                01110001

                01110001
      +         01010100
-----------------------------
                11000101
```

One's complement =   00111010

To detect errors, the receiver adds the four words (the three original words and the checksum). If the sum contains a zero, the receiver knows there has been an error.

Problem 3

All one-bit errors will be detected, but two-bit errors can be undetected (e.g., if the last digit of the first word is converted to a 0 and the last digit of the second word is converted to a 1).

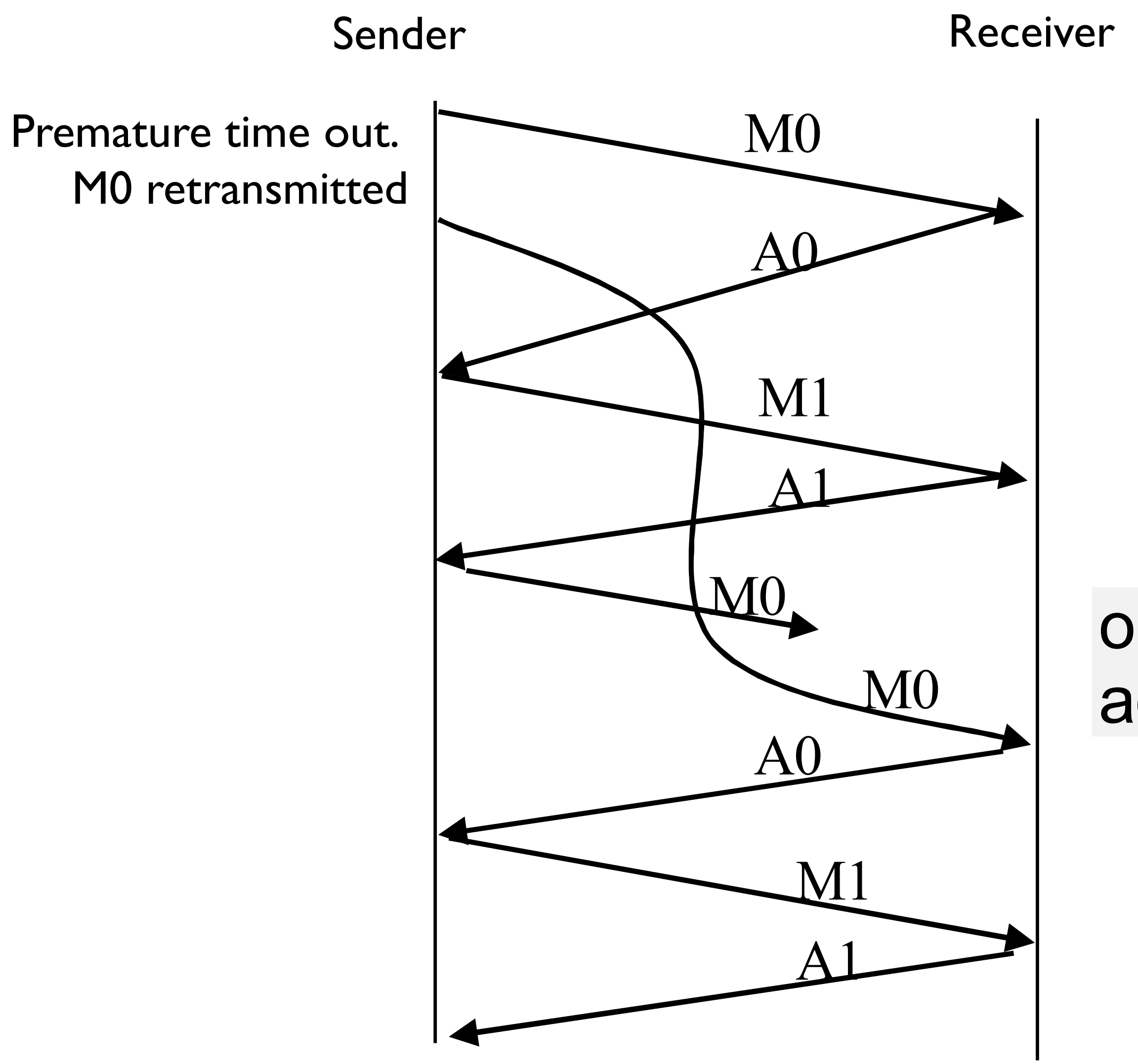
Problem 4

- Goal
  - We want the receiver to get each message exactly once and in order, even if the network is messy (can delay, lose, or reorder packets).
- The simple trick (Alternating-Bit Protocol, ABP)
  - Number messages 0, 1, 0, 1, 0, 1, …
  - The receiver remembers which number it expects next (starts at 0).
- The sender sends one packet at a time and waits:
  - If it gets the matching ACK, it flips its number and sends the next packet.
  - If the timer expires (ACK lost or delayed), it retransmits the same packet with the same number.

Problem 4

- The Problem: What If the Network Reorders Packets?
  - Networks aren't perfect—packets can take different paths and arrive out of order (reordering).
  - RDT 3.0 assumes they arrive in order (FIFO: first in, first out).
  - Without reordering, the original M0 arrives fast enough to beat the timer, avoiding resend. With reordering, the delay makes the timer expire early (from sender's view), forcing an unnecessary retransmit. Now two identical M0s are in flight.
- Real-World Fix?
  - Use more bits (e.g., 2+ for sequence numbers) so packets have unique IDs that spot duplicates, even if reordered. That's what fancier protocols like TCP do.

Problem 4



old version of M0 accepted!