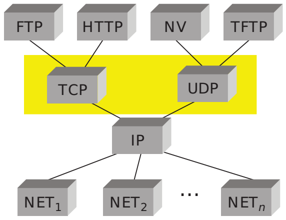
Overview

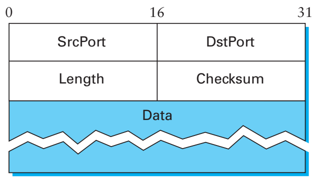
- **User datagram protocol (UDP)**
- **Packet checksums**
- **Reliability: sliding window**
- **TCP connection setup**
- **TCP windows, retransmissions, and acknowledgments**

Transport Protocol Review



- **Transport protocols sit on top of the network layer (IP)**
- **Can provide:**
  - Application-level multiplexing ("ports")
  - Error detection, reliability, etc.

UDP – *user datagram protocol*



0	16	31
SrcPort	DstPort	
Length	Checksum	
Data		

- **Unreliable and unordered datagram service**
- **Adds multiplexing, checksum on whole packet**
- **No flow control, reliability, or order guarantees**
- **Endpoints identified by ports**
- **Checksum aids in error detection**

Error detection

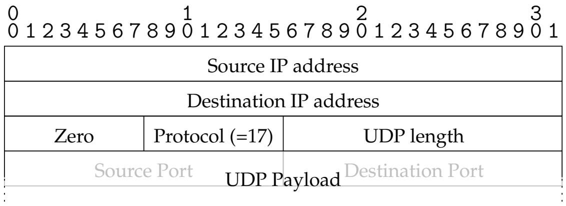
- **Transmission errors definitely happen**
  - Cosmic rays, radio interference, etc.
  - If error probability is 2^{-30}, that's 1 error per 128 MB!
- **Some link-layer protocols provide error detection**
  - But UDP/IP must work over many link layers
  - Not all links on a path may have error detection
  - Moreover, recall end-to-end argument!
    Need end-to-end check
- **UDP detects errors with a *checksum***
  - Compute small checksum value, like a hash of the packet
  - If packet corrupted in transit, checksum likely to be wrong
  - Similar checksum on IP header, but doesn't cover payload

Checksum algorithms

- **Good checksum algorithms**
  - Should detect errors that are likely to happen
    (E.g., should detect any single bit error)
  - Should be efficient to compute
- **IP, UDP, and TCP use 1s complement sum:**
  - Set checksum field to 0
  - Sum all 16-bit words in pkt
  - Add any carry bits back in
    (So 0x8000 + 0x8000 = 0x0001)
  - Flip bits (`sum = ~sum;`) to get checksum (`0xf0f0` → `0x0f0f`),
    Unless sum is `0xffff`, then checksum just `0xffff`
  - To check: Sum whole packet (including sum), should get `0xffff`

```
              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
```

UDP pseudo-header



```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

Source IP address		
Destination IP address		
Zero	Protocol (=17)	UDP length
Source Port	UDP Payload	Destination Port

- **Checksum actually includes "pseudo-header"**
  - Not transmitted, just pre-pended to compute checksum
  - Ensures UDP checksum includes IP addresses
- **Trick question: Is UDP a layer on top of IP?**

How good is UDP/IP checksum?

+ **Very fast to compute in software**
  - Same implementation works on big & little endian CPUs

– **16 bits is not very long (misses $1/2^{16}$ errors)**

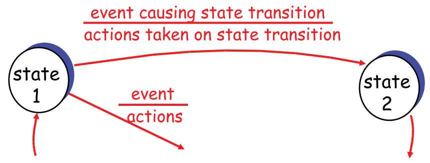
+ **Checksum does catch any 1-bit error**

– **But not any two-bit error**
  - E.g., increment one word ending 0, decrement one ending 1

– **Checksum also optional on UDP**
  - All 0s means no checksum calculated
  - If checksum word gets wiped to 0 as part of error, bad news

- **Good thing most link layers have stronger checksums**
- **Next problem: If you discard bad packets, how to ensure reliable delivery? (E.g., stop & wait lab 1)**

Finite State Machines

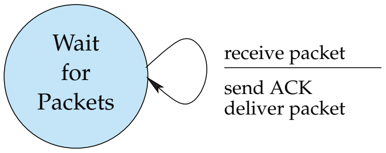


event causing state transition / actions taken on state transition

- **Represent protocols using state machines**
  - Sender and receiver each have a state machine
  - Start in some initial state
  - Events cause each side to select a state transition
- **Transition specifies action taken**
  - Specified as events/actions
  - E.g., software calls `send`/put packet on network
  - E.g., packet arrives/send acknowledgment

Stop and wait FSMs

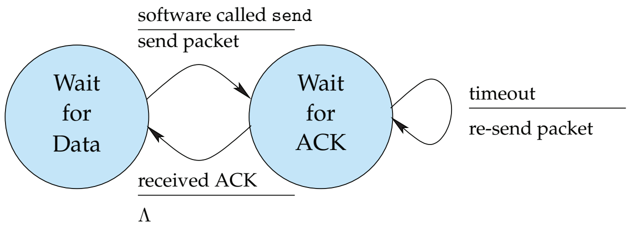
- **Receiver FSM:**



Wait for Packets

receive packet / send ACK, deliver packet

- **Sender FSM:**



Wait for Data

Wait for ACK

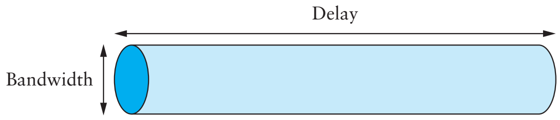
software called `send` / send packet

timeout / re-send packet

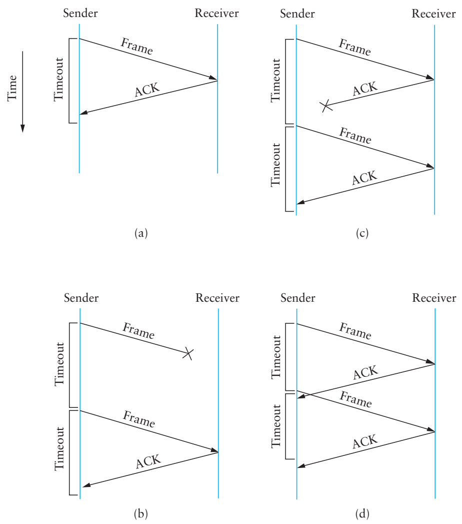
received ACK / Λ

Problems with Stop and Wait

- **Might duplicate packet... how?**
- **Can't keep pipe full**

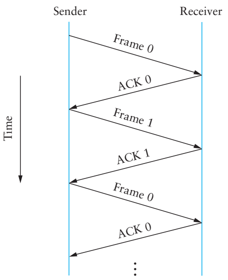


- For full utilization want # bytes in flight ≥ bandwidth×delay (But don't want to overload the network, either)

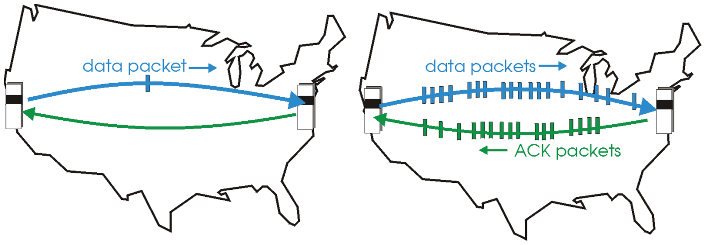


Duplicates

- **Solve problem with 1-bit counter**
  - Place in both Frame and ACK
  - Receiver knows if duplicate of last frame
  - Sender won't interpret duplicate old ACK as for new packet
- **This still requires some simplifying assumptions**
  - Network itself might duplicates packets
  - Packet might be heavily delayed and reordered
  - Assume these don't happen for now
  - But usually prefer weaker assumption: *Maximum Segment Lifetime* (MSL)



Effect of RTT on performance



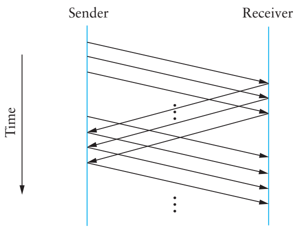
(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- **Stop & wait goodput depends on Round-Trip Time (RTT)**
  - Capped by packet size/RTT regardless of underlying link b/w
- **Need *pipelineing* for goodput to approach link throughput**

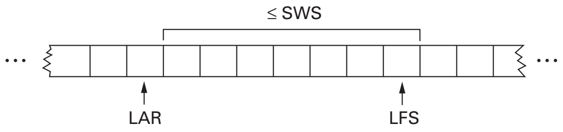
Sliding window protocol

- **Addresses problem of keeping the pipe full**
  - Generalize previous protocol with $>$ 1-bit counter
  - Allow multiple outstanding (unACKed) frames
  - Upper bound on unACKed frames, called window



SW sender

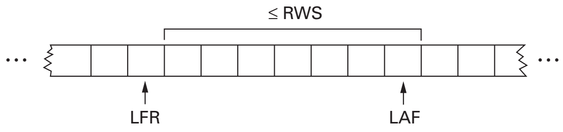
- **Assign sequence number to each frame (SeqNum)**
- **Maintain three state variables:**
  - Send Window Size (SWS)
  - Last Acknowledgment Received (LAR)
  - Last Frame Sent (LFS)



- **Maintain invariant: $\text{LFS} - \text{LAR} \leq \text{SWS}$**
- **Advance LAR when ACK arrives**
- **Buffer up to SWS frames**

SW receiver

- **Maintain three state variables**
  - Receive Window Size (RWS)
  - Largest Acceptable Frame (LAF)
  - Last Frame Received (LFR)



- **Maintain invariant: $\text{LAF} - \text{LFR} \leq \text{RWS}$**

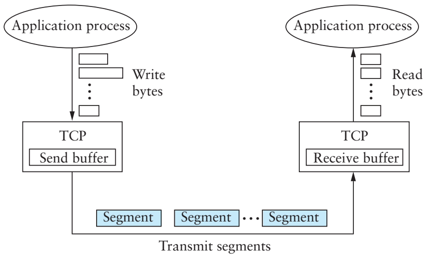
SW receiver, continued

- **When frame # SeqNum arrives:**
  - if LFR $<$ SeqNum $\leq$ LFA accept
  - if SeqNum $\leq$ LFR or SeqNum $>$ LFA discarded
- **Send *cumulative* ACKs**
  - I.e., ACK n means received all packets w. SeqNo $\leq n$
  - E.g., if received packets $1, 2, 3, 5$, must ACK 3
- **Or can alternatively use TCP-style ACKs, which specify first pkt. *not* received**
  - E.g., if received packets $1, 2, 3, 5$, must ACK 4, not 3
  - Note Labs 1 and 2 use TCP-style cumulative ACKs

Sequence number space

- **How big should RWS be?**
  - At least 1. No bigger than SWS
    (Don't accept a packet the sender shouldn't have sent)
- **How many distinct sequence numbers needed?**
- **If RWS=1, need at least SWS+1**
  - This protocoal is often called "Go-Back-N"
- **If RWS=SWS, need at least 2SWS**
  - Otherwise, bad news if ACKs are lost
  - Sender may retransmit a window that was already received
  - Receiver will think retransmissions are from next window
- **Generally need at least RWS+SWS**
  - RWS packets in unknown state (ACK may/may not be lost)
  - SWS packets in flight must not overflow sequence space

High-level view of TCP



- **Full duplex, connection-oriented byte stream**
- **Flow control**
  - If one end stops reading, writes at other eventuall block/fail
- **Congestion control**
  - Keeps sender from overrunning network [more next lecture]

2-minute stretch

TCP segment

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

<table>
<tr><td colspan="10">source port</td><td colspan="16">destination port</td></tr>
<tr><td colspan="26">sequence number</td></tr>
<tr><td colspan="26">acknowledgment number</td></tr>
<tr><td>data
offset</td><td>reserved</td><td>U
R
G</td><td>A
C
K</td><td>P
S
H</td><td>R
S
T</td><td>S
Y
N</td><td>F
I
N</td><td colspan="18">Window</td></tr>
<tr><td colspan="8">checksum</td><td colspan="18">urgent pointer</td></tr>
<tr><td colspan="18">options</td><td colspan="8">padding</td></tr>
<tr><td colspan="26">data</td></tr>
</table>

TCP fields

- **Ports**
- **Seq no. – segment position in byte stream**
  - Unlike Lab 1, sequence #s corresponds to *bytes*, not packets
- **Ack no. – seq no. sender expects to receive next**
- **Data offset – # of 4-byte header & option words**
- **Window – willing to receive**
  - Lets receiver limit SWS (possibly to 0) for flow control (more in a few slides)
- **Checksum**
- **Urgent pointer**

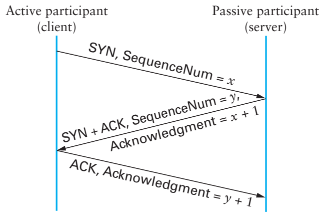
TCP Flags

- **URG – urgent data present**
- **ACK – ack no. valid (all but first segment)**
- **PSH – push data up to application immediately**
- **RST – reset connection**
- **SYN – "synchronize" establishes connection**
- **FIN – close connection**

A TCP Connection (no data)

```
orchard.48150 > essex.discard:
        S 1871560457:1871560457(0) win 16384
essex.discard > orchard.48150:
        S 3249357518:3249357518(0) ack 1871560458 win 17376
orchard.48150 > essex.discard: . ack 1 win 17376
orchard.48150 > essex.discard: F 1:1(0) ack 1 win 17376
essex.discard > orchard.48150: . ack 2 win 17376
essex.discard > orchard.48150: F 1:1(0) ack 2 win 17376
orchard.48150 > essex.discard: . ack 2 win 17375
```

Connection establishment



- **Need SYN packet in each direction**
  - Typically second SYN also acknowledges first
  - Supports "simultaneous open," seldom used in practice
- **If no program listening: server sends RST**
- **If server backlog exceeded: ignore SYN**
- **If no SYN-ACK received: retry, timeout**

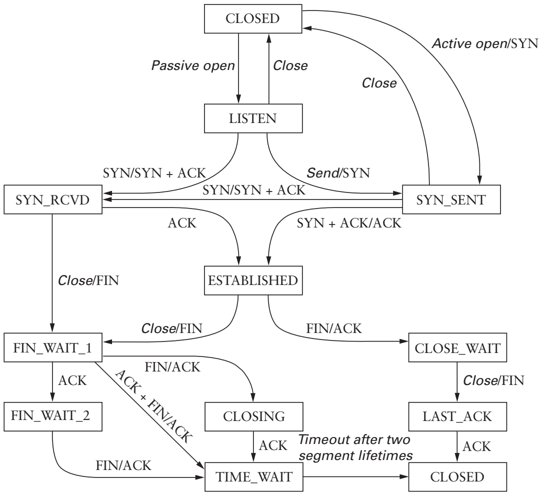
Connection termination

- **FIN bit says no more data to send**
  - Caused by close or shutdown on sending end
  - Both sides must send FIN to close a connection
- **Typical close:**
  - $A \rightarrow B$: FIN, seq S_A, ack S_B
  - $B \rightarrow A$: ack $S_A + 1$
  - $B \rightarrow A$: FIN, seq S_B, ack $S_A + 1$
  - $A \rightarrow B$: ack $S_B + 1$
- **Can also have simultaneous close**
- **After last message, can A and B forget about closed socket?**

TIME_WAIT

- **Problems with closed socket**
  - What if final ack is lost in the network?
  - What if the same port pair is immediately reused for a new connection? (Old packets might still be floating around.)
- **Solution: "active" closer goes into TIME_WAIT**
  - Active close is sending FIN before receiving one
  - After receiving ACK and FIN, keep socket around for 2MSL (twice the "maximum segment lifetime")
- **Can pose problems with servers**
  - OS has too many sockets in TIME_WAIT, slows things down
    Hack: Can send RST and delete socket, set SO_LINGER socket option to time 0 (useful for benchmark programs)
  - OS won't let you re-start server because port still in use
    SO_REUSEADDR option lets you re-bind used port number

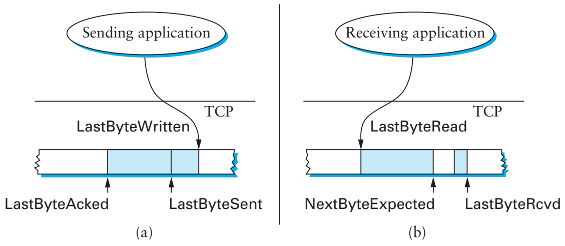
State summary [RFC 793]



Sending data

- **Bulk data sent in MSS-sized segments**
  - Chosen to avoid fragmentation (e.g., 1460 on ethernet LAN)
  - Write of 8K might use 6 segments—PSH set on last one
  - PSH avoids unnecessary context switches on receiver
- **Sender's OS can delay sends to get full segments**
  - Nagle algorithm: Only one unacknowledged short segment
  - TCP_NODELAY option avoids this behavior
- **Segments may arrive out of order**
  - Sequence number used to reassemble in order
- **Window achieves flow control**
  - Receiver sets max window w. SO_RCVBUF
  - If window 0 and sender's buffer full, write will block or return EAGAIN

Sliding window revisited



- **Used to guarantee reliable & in-order delivery**
- **New: Used for *flow control***
  - Instead of fixed window size, receiver sends AdvertisedWindow in window field of TCP header
- **Next lecture: used for *congestion control***
  - $\text{SWS} = \min(\text{AdvertisedWindow}, \text{CongestionWindow})$

A TCP connection (3 byte echo)

```
orchard.38497 > essex.echo:
        S 1968414760:1968414760(0) win 16384
essex.echo > orchard.38497:
        S 3349542637:3349542637(0) ack 1968414761 win 17376
orchard.38497 > essex.echo: . ack 1 win 17376
orchard.38497 > essex.echo: P 1:4(3) ack 1 win 17376
essex.echo > orchard.38497: . ack 4 win 17376
essex.echo > orchard.38497: P 1:4(3) ack 4 win 17376
orchard.38497 > essex.echo: . ack 4 win 17376
orchard.38497 > essex.echo: F 4:4(0) ack 4 win 17376
essex.echo > orchard.38497: . ack 5 win 17376
essex.echo > orchard.38497: F 4:4(0) ack 5 win 17376
orchard.38497 > essex.echo: . ack 5 win 17375
```

Delayed ACKs

- **Goal: Piggy-back ACKs on data**
  - Echo server just echoes, why send separate ack first?
  - Can delay ACKs for 200 msec in case application sends data
  - If more data received, immediately ACK second segment
  - Note: Never delay duplicate ACKs (if segment out of order)
- **Warning: Can interact *very* badly with Nagle**
  - "My login has 200 msec delays"
  - Set TCP_NODELAY

Other details

- **Persist timer**
  - Sender can block because of 0-sized receive window
  - Receiver may open window, but ACK message lost
  - Sender keeps probing (sending one byte beyond window)
- **Keep-alives [RFC 1122]**
  - Detect dead connection even when no data to send
  - E.g., remote login server, and client rebooted
  - Solution: Send "illegal" segments with no data and already acknowledged sequence number (`SND.NXT-1`)
  - Or can include one byte of garbage data
  - Remote side will RST (if rebooted), or timeout (if crashed)

Path MTU discovery

- **Problem: How does TCP know what MSS to use?**
  - On local network, obvious, but for more distant machines?
- **Solution: Exploit ICMP—another protocol on IP**
  - ICMP for control messages, not intended for buik data
  - IP supports **DF** (don't fragment) bit in IP header
  - Set DF to get ICMP can't fragment when segment too big
- **Can do binary search on packet sizes**
  - But better: Base algorithm on most common MTUs
  - Common algorithm may underestimate slightly (better than overestimating and loosing packet)
  - See RFC1191 for details
- **Is TCP a layer on top of IP?**

Retransmission

- **TCP dynamically estimates round trip time**
- **If segment goes unacknowledged, must retransmit**
- **Use exponential backoff (in case loss from congestion)**
- **After ∼10 minutes, give up and reset connection**
- **Problem: Don't necessarily want to halt everything for one lost packet**
  - Next lecture will explain fast retransmit optimization

32-bit seqno wrap around

Bandwidth	Time Until Wrap Around
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
Ethernet (100 Mbps)	6 minutes
STS-3 (155 Mbps)	4 minutes
STS-12 (622 Mbps)	55 seconds
STS-24 (1.2 Gbps)	28 seconds

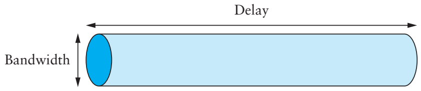
Keeping the pipe full w. 100 msec delay

Bandwidth	Delay × Bandwidth Product
T1 (1.5 Mbps)	18KB
Ethernet (10 Mbps)	122KB
T3 (45 Mbps)	549KB
Ethernet (100 Mbps)	1.2MB
STS-3 (155 Mbps)	1.8MB
STS-12 (622 Mbps)	7.4MB
STS-24 (1.2 Gbps)	14.8MB

Summary

- **User datagram protocol (UDP)**
- **Packet checksums**
- **Reliability: sliding window**
- **TCP connection setup**
- **TCP sliding windows, retransmissions, and acknowledgments**
- **Using windows for flow control**

How to fill high bw×delay pipe? [RFC 1323]



- **Extensions implemented as header options**
- **Window scale option for 16-bit window field**
  - Multiplies window by fixed power of 2 in each direction
  - Otherwise, could only fill pipe with 64 KB
- **Extend sequence space with 32-bit timestamp**
  - Protection Against Wrapped Sequence #s (PAWS)
- **Also include most recently received timestamp**
  - Allows much more accurate RTT estimation

Limitations of Flow Control

- ***Link* may be the bottleneck**
- **Sending too fast will cause heavy packet loss**
- **Many retransmissions, lost acks, poor performance**
- **Flow control provides *correctness***
- **Need more for performance: congestion control (Next lecture…)**