Last time: Reliable Data Transfer

- **Checksum**: so we can determine if the data is damaged
- **ARQ (automatic repeat request) protocols**
  - Use acknowledgements to request retransmission
- **Acknowledgement (receiver feedback)**
  - **Retransmit** if NAK or corrupt ACK
- **Sequence numbers**
  - Allow us identify duplicate segments
  - No need for NAK if we use sequence numbers for ACKs
- **Timeouts**
  - Detect segment loss
  - Time expiration = assume that a segment was lost

TCP: Transmission Control Protocol

- **Transport-layer protocol ... like UDP**
- **But**:  
  - Connection-oriented
  - Bidirectional communication channel
  - Reliable data transfer
  - Flow control
- **Network stacks on both end systems keep state**
  - “Connection” managed only in end systems
  - Routers are not aware of TCP

TCP: Connection Setup

- **Connection setup**
  - Three way handshake
  - Negotiate parameters
  - Initialize state variables

(See more details later!)
TCP Data Exchange

- TCP provides **full duplex service**
  - If a TCP connection has been established between processes A and B, A can send messages to B and B can send messages to A over the same connection.
- Outgoing data is placed in TCP’s **send buffer**
  - TCP takes data from here, creates segments, and sends them out.
  - Data grabbed must be ≤ maximum allowable segment size.

TCP Segment Size

- Data Link
- Network
- Transport
- Application Data

Protocol encapsulation: logical view

- MSS = Maximum Segment Size
- MTU = Maximum Transmission Unit

MTU = Maximum Transmission Unit
- 1500 bytes for Ethernet (v2) (→ MSS = 1460 bytes)
- 9000 bytes for Jumbo frames in gigabit Ethernet

Maximum Segment Size (MSS) is dependent on MTU (=MTU-40)

Path MTU Discovery

- What do we use for MTU?
  - No greater than the link layer’s MTU (typically 1500 or 9000 bytes)

- Path MTU = Smallest MTU of any of the hops along the path to the destination
  - No easy (foolproof) way of determining this.

- Path MTU Discovery (RFC 1191, 1981)
  - Send ICMP (Internet Control Message Protocol) packets (TCP in later versions)
  - Use MTU of “hop” and set DF “don’t fragment” bit on the IP packet
  - If the MTU of any hop is smaller, the router will
    - Discard the packet
    - Return an “ICMP Destination Unreachable” message with a code indicating “fragmentation needed”
  - Place the MTU of the next hop in a 16-bit field in the ICMP header
  - The source will reduce its MTU and try again until it gets to the destination
  - Repeat the discovery process periodically: default = 10 minutes on Windows & Linux

- Routers must handle an MTU of at least 576 bytes (512 bytes + headers)
- Minimum MTU for IPv6 = 1280 bytes

Try `tracepath` on Linux or `mturoute` on Windows

UDP Structure

- Defined in RFC 768
- Eight byte header

TCP Structure

- Defined in RFC 1122 (and others)
- 20-byte header

TCP Structure: port numbers

- Source & Destination port numbers
  - Used for multiplexing & demultiplexing
TCP Structure: checksum

- 16-bit checksum checks for data corruption in transmission

TCP Checksum

- As with UDP, the TCP header contains a 16-bit checksum
  - Checks for data corruption
- Checksum is generated by the sender and validated only by the receiver
- Checksum is 1s complement sum of:
  - IP pseudo header, TCP header, and data

TCP Structure: sequence numbers

- 32 bit sequence # and acknowledgement #
  - use for creating a reliable data transfer service

TCP Structure: receive window

- number of bytes the receiver is willing to accept
  - used for flow control

TCP Structure: header length

- 4-bit header length: length of TCP header in 32-bit words
  - This is almost always 5 (20 bytes)

TCP Structure: options

- Variable size options field
  - empty in most segments
  - maximum segment size negotiation, window scaling factor, timestamps, alternate checksum, selective acknowledgements
TCP Structure: flags

- **ACK**: acknowledgement number contains valid data
- **RST, SYN, FIN**: used for connection setup/teardown
- **PSH**: pass data to upper layer immediately
- **URG**: application data contains a region of “urgent” data
  - 16-bit urgent data pointer points to last byte of this data
- **NS, CWR, ECE**: used for congestion notification

TCP Structure Diagram:

- Header length: 20 bytes
- SYN, RST, PSH, ACK, URG, FIN, NS, CWR, ECE
- Options (if header length > 5)

TCP acknowledgement numbers

- Number of the next byte the host is expecting from the other side
  - Starting from the initial sequence number at the start of the connection

TCP sequence numbers

- TCP views application data as an ordered stream of bytes
- Sequence numbers count bytes, not segments

TCP acknowledgements

- If a host has TCP data to transmit on a connection
  - Acknowledgement placed in that TCP header (piggyback)
  - No need to send a separate acknowledgement message
- If there is no data to transmit
  - Acknowledgement sent with no data

Out of order data

- A segment that arrives out of order is not acknowledged
  - Instead, a duplicate ACK is sent asking for the missing sequence
- TCP protocol does not define what happens to the received segment
  - Two options:
    1. Discard it
    2. Hold on to out of order segments and wait for missing data
      - More complex
      - But much more efficient for the network
  - This is the preferred approach
TCP ACK generation

<table>
<thead>
<tr>
<th>Event</th>
<th>Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment.</td>
<td>Delayed ACK. Wait up to 500 ms for the</td>
</tr>
<tr>
<td>All data up to this sequence # has</td>
<td>arrival of another in-order segment.</td>
</tr>
<tr>
<td>been acknowledged.</td>
<td>Otherwise send ACK.</td>
</tr>
<tr>
<td>Arrival of in-order segment.</td>
<td>Send a single cumulative ACK. This</td>
</tr>
<tr>
<td>One other in-order segment</td>
<td>acknowledges both segments.</td>
</tr>
<tr>
<td>waiting for ACK transmission.</td>
<td></td>
</tr>
<tr>
<td>Arrival of out-of-order segment</td>
<td>Send duplicate ACK with sequence number</td>
</tr>
<tr>
<td>with higher sequence #.</td>
<td>of next expected byte.</td>
</tr>
<tr>
<td>Arrival of out-of-order segment</td>
<td>Send ACK with sequence number of next</td>
</tr>
<tr>
<td>that fills in a gap</td>
<td>unfilled byte (might be duplicate).</td>
</tr>
</tbody>
</table>

TCP Timeouts

Round-trip time estimation

- Round trip time:
  - elapsed time from sending a segment to getting an ACK
- RTT helps us determine a suitable timeout value
- TCP measures RTT for each non-retransmitted segment
- RTTs fluctuate
  - SRTT = "Smoothed Round Trip Time" = weighted average
    \[ SRTT = (1 - \alpha) \cdot \text{SRTT} + \alpha \cdot \text{RTT} \]
  - \alpha = 0.125
  - Exponential weighted moving average (EWMA)
  - Greater weight on recent measurements

Round-trip time estimation

- Compute the average variation in round-trip time from the estimate (smoothed average)
- Another exponential weighted moving average
  \[ \text{RTTVAR} = (1 - \beta) \cdot \text{RTTVAR} + \beta \cdot (\text{SRTT} - \text{RTT}) \]
  - \beta = 0.25
- RTTVAR = estimate of how much RTT typically deviates from SRTT

Setting the TCP timeout interval

- Timeout ≥ SRTT
  - Otherwise we’ll time out too early and retransmit too often
  - But don’t want a value that’s too high
  - Because we will introduce excessive delays for retransmission
- Use SRTT + x
  - x should be large when there is a lot of variation in RTT
  - x should be small when there is little variation in RTT
  - This is what RTTVAR gives us!
- TCP sets retransmission timeout to:
  \[ \text{Timeout interval} = \text{SRTT} + 4 \cdot \text{RTTVAR} \]
  - Initial value of 1 second
- When timeout occurs, the timeout interval is doubled until the next round trip

Reliable Data Transfer
TCP reliable data transfer

- TCP uses a single timer
  - Even if there are multiple transmitted unacknowledged segments
  - Less overhead than a timer per segment
- Timer is associated with oldest unacknowledged segment
- Receiver sends cumulative acknowledgements

If receive data from application
- Create TCP segment
- Set sequence #
- Send to IP layer
  - next sequence # = sequence # + data size

If timeout
- Retransmit non-acknowledged segment with smallest sequence #
  - Start timer
- If (y > SendBase)
  - SendBase = y
- If any non-acknowledged segments remaining, start timer

If receive ACK value y
- Receiver tells us it correctly received all bytes up to y-1

TCP Fast Retransmit

- TCP uses pipelining
  - Will usually send many segments before receiving ACKs for them
- If a receiver detects a missing sequence #
  - It means out-of-order delivery or a lost segment
  - TCP does not send NAKs
  - Instead, acknowledge every segment with the last in-order seq #
  - Each segment received after a missing one will generate replies with duplicate ACKs

Timeouts

- Timeout interval is normally set to
  \[ \text{Timeout interval} = \text{SRTT} + 4 \cdot \text{RTTVAR} \]
- But if a timeout occurs
  - Retransmit unacknowledged segment with smallest seq #
  - Set timer to \[ \text{Timeout interval} = 2 \cdot \text{previous timeout interval} \]
  - If timer expires again, do the same thing:
    - Retransmit & double the timeout
    - This gives us exponentially longer time intervals
    - This is a form of congestion control
- Any other event that requires a timer reset
  - Set normal time interval (SRTT + 4 \cdot RTTVAR)
TCP Fast Retransmit

- Waiting for timeouts causes a delay in retransmission
  - Increases end-to-end latency

- But a sender can detect segment loss via duplicate ACKs
  - Duplicate ACK:
    - Sender receives an ACK for a segment that was already ACKed
    - That means that a segment was received but not the sequentially next one

- If a sender receives three duplicate ACKs
  - Sender assumes the segment was lost
    (it could have been received out of order but we're guessing that's unlikely since three segments after it have been received)
  - Performs a fast retransmit
    - Sends missing segment before the retransmission timer expires

GBN or SR?

- TCP looks like a Go-Back-N protocol
  - Sender only keeps track of smallest sequence # that was transmitted but not acknowledged

- But not completely…
  - GBN will retransmit all segments in the window on timeout
  - TCP will retransmit at most one segment (lowest #)
  - TCP will retransmit no segments if it gets ACKs for higher-numbered segments before a timeout
  - Most TCP receivers will buffer out-of-order segments

- We can call it a modified Go-Back-N

SACK: Selective Acknowledgements

- Enhancement to TCP to make it be a Selective Repeat protocol

- RFC 2018: TCP Selective Acknowledgement Options

- When receiving an out-of-order segment:
  - Send duplicate ACK segment
  - But append TCP option field containing range of data received
    - List of (start byte, end byte) values
  - Negotiated between hosts at the start of a connection
  - SACK may be used if both hosts support it

Flow Control

- Incoming data goes to receive buffer
- What if it comes in faster than the process reads it?
- We don’t want overflow!

- Flow control: match transmission rate with rate at which the app is reading data

Flow control

- Receiver sends window size to sender in reply segments
- If the receiver has no messages for the sender, the sender won’t know that the buffer is being emptied!

- Probing
  - If the sender sees the receive window = 0, it will periodically send messages with 1 byte of data
  - Receiver will not accept them if the window size is really 0
  - Eventually one of them will come with an ACK reporting a non-zero window
Connection Management

Connection setup: Three-way handshake

- **Client**
  - Create SYN segment
  - Random initial seq # (client_seq)
  - No data

- **Server**
  - Allocate TCP buffers & variables
  - Create SYN-ACK segment
  - SYN = 1
  - ACK = client_seq + 1
  - Data optional

- **Client**
  - Create SYN segment
  - SYN = 1
  - ACK = client_seq + 1
  - No data

- **Server**
  - Allocate TCP buffers & variables
  - Create ACK segment
  - SYN = 0
  - ACK = server_seq + 1
  - Data optional

Server knows the client has the sequence #
Connection is established!

SYN Flooding

- An OS will allocate only a finite # of TCP buffers
- **SYN Flooding attack**
  - Send lots of SYN segments but never complete the handshake
  - The OS will not be able to accept connections until those time out
- Preventing SYN flooding attacks
  - Do not allocate buffers & state when a SYN segment is received
  - Create initial sequence # = hash(src_addr, dest_addr, src_port, dest_port, SECRET)
  - When an ACK comes back, validate the ACK #
    Compute the hash as before & add 1

MSS Announcement

- Remember the Maximum Segment Size (MSS)?
- For direct-attached networks
  - MSS = MTU of network interface – protocol headers
  - Ethernet MTU of 1500 bytes yields MSS of 1460 (1500-20-20)
- For destinations beyond the LAN (routing needed)
  - Use TCP Options field to set Maximum Segment Size
  - MSS may be obtained from PATH MTU discovery
  - Other side receives this and records it as MSS for sent messages. It can respond with the MSS it wants to use for incoming messages in the SYN-ACK message
  - All IP routers must support MSS ≥ 536 bytes

Special cases

- What if the host receives a TCP segment where the port numbers or source address do not match any connection?
  - Host sends back a "reset" segment (RST = 1)
    "I don't have a socket for this"
- For UDP messages to non-receiving ports
  - Send back an ICMP message to the sending host

Connection teardown

- Either side can end a connection
- Buffers & state variables need to be freed
- Both sides agree to send no more messages
- To close:
  - Send a TCP segment with the FIN bit set (FIN = Finish)
  - You are saying "I will not send any more data on this connection"
  - Other side acknowledges this
  - Other side then agrees to close the connection
  - Sends a TCP segment with the FIN bit set
  - You acknowledge receipt of this
  - Time wait (TIME_WAIT state) to ensure that your ACK had time to get to the other side and that any stray segments for the connection have been received
  - Wait time = twice maximum segment lifetime (timeout interval × 2)
### Connection teardown

- **FIN_WAIT_1 state**
  - Receive ACK to close request
  - Set the TIME_WAIT timer

- **FIN_WAIT_2 state**
  - Receive ACK to close request

- **TIME_WAIT state**
  - Wait until we’re sure the remote side received the final ACK

- **CLOSED state**
  - FIN=1

### Congestion control

- **Goal**
  - Limit rate at which a sender sends traffic based on congestion
  (Flow control goal: limit traffic based on remote side’s ability to process)

- **Must use end-to-end mechanisms**
  - The network gives us no information
  - We need to infer that the network is congested
  - Generally, more packet loss = more congestion

### Regulating Rate: Congestion Window

- **Window size** = # bytes we can send without waiting for ACKs

- **Receive Window (rwnd)** = flow control request from receiver
  - # bytes that a receiver is willing to receive (reported in header)

- **Congestion Window (cwnd)** = flow control by sender
  - Window size to limit the rate at which TCP sender will transmit

- **TCP will use maximum window** = \( \min(rwnd, cwnd) \)
  - These are per-connection values!

- **How does a window regulate transmission rate?**
  - If we ignore loss and delays, we transmit cwnd bytes before waiting
  - The time we wait is the round-trip time (RTT)
  - \( \text{Rate} = \frac{cwnd}{RTT} \) bytes/second

### Basic mechanisms

- **Timeout or three duplicate ACKs**
  - Assume segment loss → decrease cwnd = decrease sending rate

- **Sender gets ACKs**
  - Assume no congestion → increase cwnd = increase sending rate

- **ACKs pace the transmission of segments**
  - ACKs trigger increase in cwnd size
  - If ACKs arrive slowly (slow network) → cwnd increases slowly
  - TCP is self-clocking

- **Bandwidth probing**
  - Increase rate in response to arriving ACKs
  - ... until loss occurs; then back off

### Basic Principle: Additive Increase (AI)

If we feel we have extra network capacity

- Increase window by 1 segment each RTT
- If we successfully send cwnd bytes, increase window by 1 MSS
- That means increase fractionally for each ACK

  \[ cwnd = cwnd + \left( \frac{\text{MSS}}{cwnd(MSS)} \right) \]

- This is Additive (linear) Increase
Basic Principle: Multiplicative Decrease (MD)

If we feel we have congestion (timeout due to lost segment)

- Decrease cwnd by halving it
  \[ cwnd = \frac{cwnd}{2} \]
- This is **Multiplicative decrease**

Additive Increase / Multiplicative Decrease (AIMD)

- AIMD is a necessary condition for TCP congestion control to be stable

TCP Congestion Control

Three Parts:

1. **Slow Start**
2. **Congestion Avoidance**
3. **Fast Recovery**

Speeding things up at the start

AIMD gives us linear ramps

- Transmission follows a sawtooth pattern
- But it can take a long time to ramp up the transmission speed

TCP Slow Start

- Prevent the slow ramp at startup
- Start with an initial exponential increase in cwnd size

**Example:**
- Increase cwnd by 1 MSS for each acknowledged segment
  - Start with 1 MSS (get 1 ACK)
    - Then cwnd = 2 MSS (get 2 ACKs)
    - Then cwnd = 4 MSS (get 4 ACKs)
    - Then cwnd = 8 MSS ...
  - Transmission rate grows exponentially
    - Doubles every RTT

TCP Slow Start

- "Slow Start" actually grows quickly!
- When do we stop going faster?

  - **On timeout**
    - Sender sets cwnd=1 and restarts Slow Start process
    - Set **slow start threshold**, \( \text{ssthresh} = \frac{cwnd}{2} \)
  - **When cwnd ≥ ssthresh**
    - switch to **Congestion Avoidance** mode (slow the ramp)
    - This is not set at cold start; we will time out
  - **When three duplicate ACKs received**
    - Perform **Fast Retransmit** of segment
    - Enter **Fast Recovery State**
Congestion Avoidance

- cwnd is ½ of the size when we saw congestion
  - We think that’s safe
  - ... it worked before but a full cwnd gave a timeout – so we’re close

- Increase rate additively: 1 MSS each RTT
  - # segments in window = cwnd/MSS
    - E.g., if MSS = 1460 bytes & cwnd= 23360 bytes, cwnd/MSS = 16
  - Each ACK means we increase cwnd by MSS/(cwnd/MSS)
    - E.g., after 16 ACKs, cwnd increased by 1 MSS

- Now we have a linear growth in transmission speed

Slow Start + Congestion Avoidance

- Start with Slow Start
  - On timeout, save sthresh, restart Slow Start
  - If sthresh is reached, switch to Congestion Avoidance

Fast Recovery

- Fast Retransmit was used when duplicate ACKs received
- Duplicate ACKs means data is flowing to the receiver
  - ACKs are generated only when a segment is received
  - Might indicate that we don’t have congestion and the loss was a rare event.
  - Don’t reduce flow abruptly by going into Slow Start
    - Adjust cwnd
    - Enter Collision Avoidance state
    - Sender resumes transmission with linear growth of window

Why the name?

- Why do we call it Fast Recovery?
  - Prior to its use, TCP would set cwnd = 1 and enter Slow Start for both timeouts as well as triple duplicate ACKs
  - Slow Start ramps quickly but slower than an additive increase starting close to where you expect to be
  - We try to distinguish casual packet loss from packet loss due to congestion
TCP congestion control state summary

Timeouts should be rare: we expect most segment losses to be detected by triple ACKs.
TCP is effectively an Additive Increase / Multiplicative Decrease (AIMD) form of congestion control.