Transport Layer (TCP/UDP)
Where we are in the Course

• Moving down to the Transport Layer!
The Transport Layer

• The transport layer provides *end-to-end* connectivity
• To the transport layer, its payload is just bytes
Encapsulation

TCP Segment

IP Packet

Frame

UDP Datagram

802.11 | IP | TCP | HTTP | HTTP payload

802.11 | IP | UDP | DNS header and payload
Transport Layer Services

• Provide different kinds of data delivery across the network to applications

<table>
<thead>
<tr>
<th></th>
<th>Unreliable</th>
<th>Reliable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packets</td>
<td>Datagrams (UDP)</td>
<td>Streams (TCP)</td>
</tr>
<tr>
<td>Bytestream</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Could there be protocols in the two empty boxes?
## Comparison of Internet Transports: Function

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streams</td>
<td>Datagrams</td>
<td></td>
</tr>
<tr>
<td>Connections</td>
<td>Connectionless</td>
<td></td>
</tr>
<tr>
<td>Bytes are delivered to receiving app <strong>reliably</strong> (once, and in order)</td>
<td>Packets may be <strong>lost, reordered, duplicated</strong> (but not corrupted)</td>
<td></td>
</tr>
<tr>
<td><strong>Arbitrary length content</strong></td>
<td><strong>Fixed maximum datagram size</strong></td>
<td></td>
</tr>
</tbody>
</table>
## Comparison of Internet Transports: Performance

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection latency</td>
<td>No delay</td>
<td>No delay</td>
</tr>
<tr>
<td>Segment delivery latency (“nagling”)</td>
<td>Datagram is sent now</td>
<td>No flow control (can lead to many lost datagrams)</td>
</tr>
<tr>
<td><strong>Flow control</strong></td>
<td>matches sender’s rate to receiver’s capability</td>
<td></td>
</tr>
<tr>
<td>Congestion control</td>
<td>matches sender’s rate to network’s capability</td>
<td>No congestion control (can lead to many lost datagrams)</td>
</tr>
</tbody>
</table>
Socket API

• Simple **OS abstraction** to use the network
  • The “network” API (really Transport service) used to write all Internet apps
  • Part of all major OSes and languages; originally Berkeley (Unix) ~1983

• Supports both Internet transport services (TCP and UDP)

• The OS provides sockets; the Internet provides the port abstraction
Socket API

- Sockets are associated with (“bound to”) Internet ports
Socket API

Same API used for Streams and Datagrams

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET</td>
<td>Create a new communication endpoint</td>
</tr>
<tr>
<td>BIND</td>
<td>Associate a local address (port) with a socket</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Announce willingness to accept connections</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Passively establish an incoming connection</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>SEND(ToF)</td>
<td>Send some data over the socket</td>
</tr>
<tr>
<td>RECEIVE(From)</td>
<td>Receive some data over the socket</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Release the socket</td>
</tr>
</tbody>
</table>

Note: A language layer can obscure this interface
Ports

• Application process is identified by the tuple 
  \(<\text{IP address, protocol, and port}>\)
  • Ports are 16-bit integers representing local “mailboxes” that a process leases

• Servers often bind to “well-known ports”
  • numbered below 1024
  • require administrative privileges (“privileged ports”)

• Clients often assigned “ephemeral” ports
  • Chosen by OS, used temporarily
## Some Well-Known Ports

<table>
<thead>
<tr>
<th>Port</th>
<th>Protocol</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>20, 21</td>
<td>FTP</td>
<td>File transfer</td>
</tr>
<tr>
<td>22</td>
<td>SSH</td>
<td>Remote login, replacement for Telnet</td>
</tr>
<tr>
<td>25</td>
<td>SMTP</td>
<td>Email</td>
</tr>
<tr>
<td>80</td>
<td>HTTP</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>110</td>
<td>POP-3</td>
<td>Remote email access</td>
</tr>
<tr>
<td>143</td>
<td>IMAP</td>
<td>Remote email access</td>
</tr>
<tr>
<td>443</td>
<td>HTTPS</td>
<td>Secure Web (HTTP over SSL/TLS)</td>
</tr>
<tr>
<td>543</td>
<td>RTSP</td>
<td>Media player control</td>
</tr>
<tr>
<td>631</td>
<td>IPP</td>
<td>Printer sharing</td>
</tr>
</tbody>
</table>
Topics

• Service models
  • Socket API and ports
  • Datagrams, Streams
• User Datagram Protocol (UDP)
• Connections (TCP)
• Sliding Window (TCP)
• Flow control (TCP)
• Retransmission timers (TCP)
• Congestion control (TCP)
UDP
User Datagram Protocol (UDP)

• Used by apps that don’t want TCP semantics or for which TCP performance characteristics are unacceptable
  • Voice-over-IP
  • DNS, RPC
  • DHCP

(If application wants reliability and messages then it has work to do!)
Datagram Sockets

Client (host 1)  Time  Server (host 2)

request

reply
Datagram Sockets

The protocol implied by this diagram is horribly broken!
UDP Buffering

Application

Ports

Transport (UDP)

Message queues

Port Mux/Demux

Network (IP)

packet
UDP Header

- Uses ports to identify sending and receiving application processes
- Datagram length limited to 64K
- Checksum (16 bits) for reliability

![UDP header diagram](image-url)
UDP Header

- Optional checksum covers UDP segment and IP pseudoheader
  - Checks key IP fields (addresses)
  - Value of zero means “no checksum”
Internet Checksum

• Idea:
  • sender sums up data in N-bit words
    • results in a 16-bit value that is a function of the data
  • receiver performs same summation
  • if value receiver computes doesn’t match value sent by sender, the packet has been corrupted
  • Widely used in, e.g., TCP/IP/UDP
Internet Checksum

• Sum is defined in 1s complement arithmetic (must add back carries)
  • And it’s the negative sum

• “The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words ...” – RFC 791
Internet Checksum (2)

Sending:
1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum
Internet Checksum (3)

Sending:
1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover back to get 16 bits
4. Negate (complement) to get sum

\[
\begin{align*}
0001 \\
f204 \\
f4f5 \\
f6f7 \\
+ (0000) \\
\hline \\
2ddf1 \\
\downarrow \\
ddf1 \\
\downarrow + 2 \\
\hline \\
ddf3 \\
\downarrow \\
220c
\end{align*}
\]
Internet Checksum (4)

Receiving:

1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0

```
0001
f204
f4f5
f6f7
+ 220c
-----
```
Internet Checksum (5)

Receiving:
1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0
(Pre-TCP)
Reliability and Retransmissions
Context on Reliability

- Where in the stack should we place reliability?
Context on Reliability (2)

• Everywhere! It is a key issue
  • Different layers contribute differently

Recover actions (correctness)

Mask errors (performance optimization)
ARQ (Automatic Repeat reQuest)

- ARQ often used when errors are common or must be corrected
  - E.g., WiFi (common) and TCP (must correct)

- Rules at sender and receiver:
  - Receiver automatically acknowledges correct frames with an ACK
    - *positive acknowledgements*
  - Sender automatically resends after a timeout
  - Keep re-sending until an ACK is received
• Normal operation (no loss)
ARQ

• Loss and retransmission
So What’s Tricky About ARQ?

• Two non-trivial issues:
  • How long to set the timeout?
  • How to avoid accepting duplicate frames as new frames

• Want performance in the common case and correctness always
Timeouts

• Timeout should be:
  • Not too big (link goes idle)
  • Not too small (spurious resend)

• Fairly easy on a LAN
  • Clear worst case, little variation

• Fairly difficult over the Internet
  • Much variation, no obvious bound
  • We’ll revisit this with TCP (later)
Duplicates

• What happens if an ACK is lost?
Duplicates

• What happens if the timeout is early?

![Diagram showing the sender and receiver with frames and ACKs]

Sender

Timeout

Frame

ACK

Receiver

Frame

ACK
Duplicates

• What happens if the timeout is early?

Which frame is this ACK ACK’ing?
Sequence Numbers

- For correctness, frames and ACKs must both carry sequence numbers.

- At an extreme, to distinguish the current frame from the next one, a single bit (two numbers) is sufficient.
  - Called Stop-and-Wait protocol.

- In general, the number of packets that can be in flight is limited to half the range of the sequence numbers.
Stop-and-Wait

• In the normal case:
Stop-and-Wait (2)

• In the normal case:
Stop-and-Wait (3)

• With ACK loss:
Stop-and-Wait (4)

• With ACK loss:

Sender

Receiver

Timeout

Frame 0

ACK 0

It’s a Resend!

Frame 0

ACK 0

X

ACK 0
Stop-and-Wait (5)

• With early timeout:
Stop-and-Wait (6)

• With early timeout:

Sender

Frame 0

ACK 0

Timeout

OK …

Receiver

Frame 0

ACK 0

It’s a Resend

CSE 461 University of Washington
Limitation of Stop-and-Wait

• It allows only a single frame to be outstanding from the sender:
  • Good for LAN, not efficient for high latency communication

• Ex: R=1 Mbps, D = 50 ms
  • How many frames/sec? If R=10 Mbps?
Sliding Window

• Generalization of stop-and-wait
  • Allows $W$ frames to be outstanding
  • Can send $W$ frames per *round trip time* ($=2D$)

• Various options for numbering frames/ACKs and handling loss
  • Will look at along with...
TCP
TCP Header

From https://nmap.org/book/tcpip-ref.html
TCP Protocol

• TCP Consists of 3 primary phases:
  • Connection Establishment (Setup)
  • Sliding Windows/Flow Control
  • Connection Release (Teardown)
Connection Establishment

• Both sender and receiver must be ready before we start the transfer of data
  • Need to agree on a set of parameters
  • e.g., the Maximum Segment Size (MSS)

• This is *signaling*
  • It sets up state at the endpoints
  • Like “dialing” for a telephone call
Three-Way Handshake

- Used in TCP; opens connection for data in both directions
- Each side probes the other with a fresh Initial Sequence Number (ISN)
  - Sends on a SYNchronize segment
  - Echo on an ACKnowledge segment
- Chosen to be robust even against delayed duplicates
Three-Way Handshake (2)

- Three steps:
  - Client sends SYN(x)
  - Server replies with SYN(y)ACK(x+1)
  - Client replies with ACK(y+1)
  - SYNs are retransmitted if lost

- Sequence and ack numbers carried on further segments
Three-Way Handshake (3)

• Suppose delayed, duplicate copies of the SYN and ACK arrive at the server!
  • Improbable, but anyhow ...

Active party (client)

Passive party (server)

SYN (SEQ=x)

(SEQ=x+1, ACK=z+1)
Three-Way Handshake (4)

• Suppose delayed, duplicate copies of the SYN and ACK arrive at the server!
  • Improbable, but anyhow ...

• Connection will be cleanly rejected on both sides 😊
Both parties run instances of this state machine
TCP Release

• Follow the active party
TCP Release

• Follow the passive party
TCP Release

• Again, with states …

![TCP State Diagram]

- Active party
  - ESTABLISHED
  - FIN_WAIT_1
  - FIN_WAIT_2
  - TIME_WAIT (timeout)
  - CLOSED

- Passive party
  - ESTABLISHED
  - CLOSE_WAIT
  - LAST_ACK
  - CLOSED

1. FIN (SEQ=x)
2. (SEQ=y, ACK=x+1)
3. (SEQ=x+1, ACK=y+1)
TIME_WAIT State

• Wait a long time after sending all segments and before completing the close
  • Two times the maximum segment lifetime of 60 seconds

• Why?
  • ACK might have been lost, in which case FIN will be resent for an orderly close
  • Could otherwise interfere with a subsequent connection
Flow Control
Recall

• ARQ with one message at a time is Stop-and-Wait (normal case below)
Limitation of Stop-and-Wait

- It allows only a single message to be outstanding from the sender:
  - Fine for LAN (only one frame fit)
  - Not efficient for network paths with BD >> 1 packet
Limitation of Stop-and-Wait (2)

• Example: R=1 Mbps, D = 50 ms
  • RTT (Round Trip Time) = 2D = 100 ms
  • How many packets/sec?

• What if R=10 Mbps?
Sliding Window

• Generalization of stop-and-wait
  • Allows $W$ packets to be outstanding
  • Can send $W$ packets per RTT ($=2D$)

• Pipelining improves performance
  • Need $W=2BD$ to fill network path
Sliding Window (2)

- What W will use the network capacity?
- Ex: R=1 Mbps, D = 50 ms

- Ex: What if R=10 Mbps?
Sliding Window (3)

• Ex: R=1 Mbps, D = 50 ms
  • 2BD = 10^6 b/sec x 100. 10^{-3} sec = 100 kbit
  • W = 2BD = 10 packets of 1250 bytes

• Ex: What if R=10 Mbps?
  • 2BD = 1000 kbit
  • W = 2BD = 100 packets of 1250 bytes
Sliding Window Protocol

• Many variations, depending on how buffers, acknowledgements, and retransmissions are handled

• Go-Back-N
  • Simplest version, can be inefficient

• Selective Repeat
  • More complex, better performance
Sliding Window – Sender

- Sender buffers up to $W$ segments until they are acknowledged
  - $\text{LFS}=\text{LAST FRAME SENT}$, $\text{LAR}=\text{LAST ACK REC'D}$
  - Sends while $\text{LFS} - \text{LAR} \leq W$

![Diagram]

- Sliding Window
- W=5
- Available
- ..
- Acked
- Unacked
- Unavailable
- ..
- LAR
- LFS
- seq. number
Sliding Window – Sender (2)

• Transport accepts another segment of data from the Application ...
  • Transport sends it (as LFS–LAR → 5)
Sliding Window – Sender (3)

- Next higher ACK arrives from peer...
  - Window advances, buffer is freed
  - LFS–LAR $\rightarrow$ 4 (can send one more)
Sliding Window – Go-Back-N

- Receiver keeps only a single packet buffer for the next segment
  - State variable, LAS = LAST ACK SENT
- On receive:
  - If seq. number is LAS+1, accept and pass it to app, update LAS, send ACK
  - Otherwise discard (as out of order)
Sliding Window – Selective Repeat

• Receiver passes data to app in order, and buffers out-of-order segments to reduce retransmissions

• ACK conveys highest in-order segment, plus hints about out-of-order segments

• TCP uses a selective repeat design; we’ll see the details later
Sliding Window – Selective Repeat (2)

-Buffers W segments, keeps state variable \( \text{LAS} = \text{LAST ACK SENT} \)

-On receive:
  - Buffer segments \([\text{LAS}+1, \text{LAS}+W]\)
  - Send app in-order segments from \(\text{LAS}+1\), and update \(\text{LAS}\)
  - Send ACK for \(\text{LAS}\) regardless
Sliding Window – Retransmissions

• Go-Back-N uses a single timer to detect losses
  • On timeout, resends buffered packets starting at LAR+1

• Selective Repeat uses a timer per unacked segment to detect losses
  • On timeout for segment, resend it
  • Hope to resend fewer segments
Sequence Numbers

• Need more than 0/1 for Stop-and-Wait …
  • But how many?
• For Selective Repeat, need W numbers for packets, plus W for acks of earlier packets
  • 2W seq. numbers
  • Fewer for Go-Back-N (W+1)
• Typically implement seq. number with an N-bit counter that wraps around at $2^N - 1$
  • E.g., N=8: ..., 253, 254, 255, 0, 1, 2, 3, ...
Sequence Time Plot

Transmissions (at Sender)

Delay (=RTT/2)

Acks (at Receiver)
Sequence Time Plot (2)

Go-Back-N scenario
Sequence Time Plot (3)

- Retransmissions
- Loss
- Timeout

Seq. Number vs Time
Problem

• Sliding window has pipelining to keep network busy
  • What if the receiver is overloaded?

![Diagram showing streaming video from Big Iron to Wee Mobile]
Sliding Window – Receiver

• Consider receiver with W buffers
  • LAS = LAST ACK SENT, app pulls in-order data from buffer with recv() call
Sliding Window – Receiver (2)

- Suppose the next two segments arrive but app does not call `recv()`
Sliding Window – Receiver (3)

• Suppose the next two segments arrive but app does not call recv()
  • LAS rises, but we can’t slide window!

![Diagram showing sliding window with LAS and seq. number]
Sliding Window – Receiver (4)

• Further segments arrive (in order) we fill buffer
  • Must drop segments until app recvs!

Nothing Acceptable!

W=5

Finished Acked

Too high

.. LAS seq. number ..
Sliding Window – Receiver (5)

- App `recv()` takes two segments
  - Window slides (phew)
Flow Control

• Avoid loss at receiver by telling sender the available buffer space
  • $WIN=#$Acceptable, not $W$ (from LAS)
Flow Control (2)

• Sender uses lower of the sliding window and flow control window \( \text{WIN} \) as the effective window size.
Flow Control (3)

- TCP-style example
  - SEQ/ACK sliding window
  - Flow control with WIN
  - SEQ + length < ACK+WIN
  - 4KB buffer at receiver
  - Circular buffer of bytes
How to set the timeout for sending a retransmission

Adapting to the network path
Retransmissions

• With sliding window, detecting loss with timeout
  • Set timer when a segment is sent
  • Cancel timer when ack is received
  • If timer fires, retransmit data as lost
Timeout Problem

• Timeout should be “just right”
  • Too long wastes network capacity
  • Too short leads to spurious resends
  • But what is “just right”?  
• Easy to set on a LAN (Link)
  • Short, fixed, predictable RTT
• Hard on the Internet (Transport)
  • Wide range, variable RTT
Example of RTTs

Round Trip Time (ms)

| BCN → SEA → BCN |

Seconds
Example of RTTs (2)

Variation due to queuing at routers, changes in network paths, etc.

Propagation (+transmission) delay $\approx 2D$
Example of RTTs (3)

Timer too high!

Need to adapt to the network conditions

Timer too low!
Adaptive Timeout

• Smoothed estimates of the RTT (1) and variance in RTT (2)
  • Update estimates with a moving average
    1. \( \text{SRTT}_{N+1} = 0.9 \times \text{SRTT}_N + 0.1 \times \text{RTT}_{N+1} \)
    2. \( \text{Svar}_{N+1} = 0.9 \times \text{Svar}_N + 0.1 \times |\text{RTT}_{N+1} - \text{SRTT}_{N+1}| \)
• Set timeout to a multiple of estimates
  • To estimate the upper RTT in practice
  • TCP Timeout\(_N\) = \( \text{SRTT}_N + 4 \times \text{Svar}_N \)
Example of Adaptive Timeout
Example of Adaptive Timeout (2)

Early timeout

Timeout (SRTT + 4*Svar)
Adaptive Timeout (2)

• Simple to compute, does a good job of tracking actual RTT
  • Little “headroom” to lower
  • Yet very few early timeouts

• Turns out to be important for good performance and robustness