Transport services and protocols

- provide logical communication between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP

Transport vs. network layer

- network layer: logical communication between hosts
- transport layer: logical communication between processes
  - relies on, enhances, network layer services

Household analogy:
- 12 kids sending letters to 12 kids
- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
Layering

![Layering Diagram](Image)

How demultiplexing works

- host receives IP datagrams
  + each datagram has source IP address, destination IP address
  + each datagram carries 1 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

Connectionless demultiplexing

- recall: create sockets with host-local port numbers:
  DataOutputStream mySocket1 = new 
  DataOutputStream soket(12534);
  DataOutputStream mySocket2 = new 
  DataOutputStream soket(12535);
- recall: when creating datagram to send into UDP socket, must specify 
  (dest IP address, dest port number)
- when host receives UDP segment:
  + checks destination port number in segment
  + directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

UDP client server

```
UDPClient()
UDPserver()
```

```
DatagramSocket mySocket1 = new 
DatagramSocket (12534);
DatagramSocket mySocket2 = new 
DatagramSocket (12535);
```

```
UDPreceiver() {
try:
r sd = mySocket.socket(
mysoc.AF_INET,
mysoc.SOCK_DGRAM)
except mysoc.error as err:
exit()
# Define the port on which you want to receive  from  the server
Rport = 50007
myip = mySocket.gethostbyname
(mySocket.gethostname)
server_binding = (
myip,
Rport)
rsd.bind(
server_binding)
data, addr = 
rsd.recvfrom
(1024)
print(data.decode
("utf-8"))
# Close the receiver socket
rsd.close() 
exit()
```
UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

UDP segment format

UDP checksum

**Goal:** detect "errors" (e.g., flipped bits) in transmitted segment

**Sender:**
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

**Receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless? More later....

Internet Checksum Example

- Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

\[ \begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
+ & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
\hline
\end{array} \]

wraparound \[ \begin{array}{cccccccccccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
\end{array} \]

\[ \begin{array}{cccccccccccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\end{array} \]
Internet Checksum Example

- Complement of the sum is stored in the checksum field
- At the receiver, all the byte fields are added along with the checksum
- Sum + checksum must be all 1s
- No error else discard packet
- UDP checksum is optional in IPv4
- UDP checksum is mandatory in IPv6

Reliable Data Transfer

- Problem: Reliability
  - Want an abstraction of a reliable link even though packets can be corrupted or get lost
  - Where can packets be corrupted or lost?
    - In the network
    - At the receiver
- Solution: keep track of the packets
  - Not as simple as one would expect

Reliable transmission & Flow Control

- What to do when there is a packet loss?
  - On the link (in the network)
  - At the receiver (buffer overflow)
- Need to recoup losses
- What happens if the packet is lost in the network?
  - A random event, retransmit
  - What happens if the sender tries to transmit faster than the receiver can accept?
    - Data will be lost unless flow control is implemented

Reliability support

- Sender needs to know if a packet was lost
- How?
  - Acknowledgement
  - Positive ack and negative ack
- Sender needs to retransmit
- How?
  - Timeouts
  - Acks can also get lost
Some Flow Control Algorithms

- Flow control for the ideal network
- Stop and Wait for noiseless channels
- Stop and Wait for noisy channels
- Sliding window protocols
- Sliding window with error control
  - Go Back N
  - Selective Repeat

Flow control in the ideal network

Assumptions:
Error free transmission link,
Infinite buffer at the receiver

No acknowledgement of frames necessary
Since the data link is error-free and the receiver can buffer as many frames as it likes, no frame will ever be lost

Flow control in the ideal network (cont'd)

Stop-and-wait Normal Operation

Packet Length = L; Bandwidth = R; RTT = 2 * Prop Delay

- First packet bit transmitted, $t = 0$
- Last packet bit transmitted, $t = \frac{L}{R}$
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = RTT + \frac{L}{R}$
- Timeout

sender
receiver
Infinite bucket
stop-and-wait Packet Lost

Timeout

ACK

first packet bit arrives, then packet bit arrives, send ACK

stop-and-wait Ack Lost

Timeout

ACK

Packet retransmitted

stop-and-wait Delayed Ack

Timeout too short

Duplicate Transmission

stop-and-wait Detecting duplicates

Timeout too short

Duplicate Transmission
Performance of stop and wait

- Example: 1 Gbps link, 1.5 ms e-e prop. delay, 1KB packet:

\[
T_{\text{transmit}} = \frac{L}{R} \quad \text{(packet length in bits)} \quad \frac{10^9 \text{ b/sec}}{8 \text{kb/pt}} = 8 \text{ microsec}
\]

\[
U_{\text{sender}} = \frac{L/R}{\text{RTT} + L/R} = \frac{0.008}{3.008} = 0.0027
\]

- \( U_{\text{sender}} \): utilization - fraction of time sender busy sending
- 1KB pkt every 3 msec \( \rightarrow \) 330kB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources!

Bandwidth delay product

- Continuously send data until first Ack
- How much? \( BW \times RTT \)
- Known as Bandwidth delay product
- Number of packets \( N = BW \times RTT / \text{Packet size} \)

Pipelined protocols

Pipelining: Sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

\[
U_{\text{sender}} = \frac{3L/R}{\text{RTT} + L/R} = \frac{0.024}{3.008} = 0.008
\]

Increase utilization by a factor of 3!
Sliding Window Protocols

Sender Definitions

Sequence Number: Each frame is assigned a sequence number that is incremented as each frame is transmitted.

Sender Window: Keeps track of sequence numbers for frames that have been sent but not yet acknowledged.

Sender Window size: The maximum number of frames the sender may transmit without receiving any acknowledgements.

Last Frame sent: The highest sequence number frames sent.

Last Ack received: Sequence number of last ACK.

Sliding Window with Maximum Sender Window Size SWS

With a maximum window size of $SWS$, the sender can transmit up to $SWS$ frames before "being blocked.

This allows the sender to transmit several frames before waiting for an acknowledgment.

Sender-Side Window with $W_S=2$

Receiver-Side Window with $W_R=2$
**Sliding Window Protocols**

**Receiver Definitions**

- **Receiver Window**: Keeps track of sequence numbers for frames the receiver is allowed to accept.
- **Receiver Window size**: The maximum number of frames the receiver may receive before returning an acknowledgement to the sender.
- **Largest Frame received**: The highest in-sequence numbered frame received.
- **Largest Acceptable Frame**: Highest sequence number acceptable.

---

**Sliding Window with Maximum Receiver Window Size RWS**

With a maximum window size of \( RWS \), the receiver rejects packets if \( \text{SeqNum} \leq \text{LFR} \) or \( \text{SeqNum} > \text{LAF} \).

Why?

- \( \text{LAF} - \text{LFR} \leq RWS \)
- Receiver acknowledges sequential number so all earlier frames have been received.

---

**What about Errors?**

What if a data or acknowledgement frame is lost when using a sliding window protocol?

- **Two Solutions**: Go Back N, Selective Repeat.

---

**Sliding Window with Go Back N**

- When the receiver notices a missing or erroneous frame, it simply discards all frames with greater sequence numbers and sends no ACK.
- The sender will eventually time out and retransmit all the frames in its sending window.
**Go Back N**

Go Back N can recover from erroneous or missing frames.

**But...**

It is wasteful. If there are errors, the sender will spend time retransmitting frames the receiver has already seen.

---

**Sliding Window with Selective Repeat**

The sender retransmits only the frame with errors:
- The receiver stores all the correct frames that arrive following the bad one. (Note that the receiver requires a frame buffer for each sequence number in its receiver window.)
- When the receiver notices a skipped sequence number, it keeps acknowledging the last good sequence number.
- When the sender times out waiting for an acknowledgement, it just retransmits the one unacknowledged frame, not all its successors.

---

**Selective Repeat**

Selective Repeat is shown in the diagram.
TCP

- TCP provides the end-to-end reliable connection that IP alone cannot support.

- The protocol:
  - Connection management
  - Retransmission
  - Flow control
  - Congestion control
  - Frame format

Connection Management

TCP Connection Establishment

- Three-way Handshake

TCP Connection Tear-down

- Two double handshakes:
**Retransmission**

- When a packet remains unacknowledged for a period of time, TCP assumes it is lost and retransmits it.
- TCP tries to calculate the round trip time (RTT) for a packet and its acknowledgement.
- From the RTT, TCP can guess how long it should wait before timing out.

**Round Trip Time (RTT)**

\[
\text{RTT} = \text{Time for packet to arrive at destination} + \text{Time for ACK to return from destination}
\]

**RTT Calculation**

\[
\text{RTT} = 2.2 \text{ sec} - 0.9 \text{ sec} = 1.3 \text{ sec}
\]
Smoothing the RTT measurement

- First, we must smooth the round trip time due to variations in delay within the network:

  \[ SRTT = \alpha \cdot SRTT + (1 - \alpha) \cdot RTT_{arriving\,ACK} \]

- The smoothed round trip time (SRTT) weights previously received RTTs by the \( \alpha \) parameter
- \( \alpha \) is typically equal to 0.875

Retransmission Timeout Interval (RTO)

- The timeout value is then calculated by multiplying the smoothed RTT by some factor (greater than 1) called \( \beta \)

  \[ \text{Timeout} = \beta \cdot SRTT \]

- This coefficient of \( \beta \) is included to allow for some variation in the round trip times.

Example

Initial SRTT = 1.50
\( \alpha = 0.875 \), \( \beta = 4.0 \)

<table>
<thead>
<tr>
<th>RTT Meas.</th>
<th>SRTT</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5 s</td>
<td>1.50</td>
<td>( \beta \times 1.50 = 6.00 )</td>
</tr>
<tr>
<td>1.0 s</td>
<td>1.50 + 1.0 ( \times (1 - \alpha) ) = 1.44</td>
<td>( \beta \times 1.44 = 5.76 )</td>
</tr>
<tr>
<td>2.2 s</td>
<td>1.44 + 2.2 ( \times (1 - \alpha) ) = 1.54</td>
<td>( \beta \times 1.54 = 6.16 )</td>
</tr>
<tr>
<td>1.0 s</td>
<td>1.54 + 1.0 ( \times (1 - \alpha) ) = 1.47</td>
<td>( \beta \times 1.47 = 5.88 )</td>
</tr>
<tr>
<td>0.8 s</td>
<td>1.47 + 0.8 ( \times (1 - \alpha) ) = 1.39</td>
<td>( \beta \times 1.39 = 5.56 )</td>
</tr>
<tr>
<td>3.1 s</td>
<td>1.39</td>
<td>( \beta \times 1.39 = 5.56 )</td>
</tr>
<tr>
<td>2.0 s</td>
<td>1.39</td>
<td>( \beta \times 1.39 = 5.56 )</td>
</tr>
</tbody>
</table>

Problem with RTT Calculation
Karn’s Algorithm

- Retransmission ambiguity
  - Measure RTT from original data segment
  - Measure RTT from most recent segment
- Either way there is a problem in RTT estimate
- One solution
  - Never update RTT measurements based on acknowledgements from retransmitted packets
- Problem: Sudden change in RTT can cause system never to update RTT
  - Primary path failure leads to a slower secondary path

Karn’s algorithm

- Use back-off as part of RTT computation
- Whenever packet loss, RTO is increased by a factor
- Use this increased RTO as RTO estimate for the next segment (not from SRTT)
- Only after an acknowledgment received for a successful transmission is the timer set to new RTT obtained from SRTT

Flow Control

TCP Flow Control

- TCP uses a modified version of the sliding window
- In acknowledgements, TCP uses the “Window size” field to tell the sender how many bytes it may transmit
- TCP uses bytes, not packets, as sequence numbers
TCP Flow Control (cont'd)

Important information in TCP/IP packet headers

- **Number of bytes in packet (N)**
- **Sequence number of first data byte (SEQ)**
- **Window size at the receiver (WIN)**
- **Sequence number of next expected byte (ACK)**
- **Transport control flags**

TCP data transfer

- **Packet 1**: 15:15:13.152339 eth0 > remus.4706 > scully.echo: S
  1264296504:1264296504(0) win 32120 <mss 1460,sackOK,timestamp 7125312 0,nop,wscale 0> (DF)

TCP Flow Control (cont'd)

- **Sender**
  - Data
  - ACK: 32120
  - SACK: 0
  - WScale: 0
  - Window: 32120

- **Receiver**
  - Buffer: 4K
  - Window: 32120

Example TCP session

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Source IP:port</th>
<th>Dest IP:port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet 1</td>
<td>15:15:13.152339 eth0 &gt; remus.4706 &gt; scully.echo: S</td>
<td>1264296504:1264296504(0) win 32120 &lt;mss 1460,sackOK,timestamp 7125312 0,nop,wscale 0&gt; (DF)</td>
</tr>
<tr>
<td>Packet 2</td>
<td>15:15:13.153845 eth0 &lt; scully.echo &gt; remus.4706: S</td>
<td>87676030:87676030(0) ack 1264296505 win 8760 &lt;mss 1460&gt;</td>
</tr>
<tr>
<td>Packet 3</td>
<td>15:15:13.153912 eth0 &gt; remus.4706 &gt; scully.echo: .</td>
<td>1264296505:1264296505(0) ack 875676031 win 32120</td>
</tr>
<tr>
<td>Packet 4</td>
<td>15:15:18.591716 eth0 &gt; remus.4706 &gt; scully.echo: P</td>
<td>1264296505:1264296508(3) ack 875676031 win 32120</td>
</tr>
<tr>
<td>Packet 5</td>
<td>15:15:18.593255 eth0 &lt; scully.echo &gt; remus.4706: P</td>
<td>875676031:875676034(3) ack 1264296508 win 8760</td>
</tr>
</tbody>
</table>

TCP Flow Control (cont'd)

- **Sender**
  - Application writes 4K
  - First 2K is sent
  - Second 2K is sent with SACK

- **Receiver**
  - Application reads 2K
  - Application reads 2K

TCP Flow Control (cont'd)

- **Sender**
  - Application writes 4K
  - First 2K is sent
  - Second 2K is sent with SACK

- **Receiver**
  - Application reads 2K
  - Application reads 2K
TCP Flow Control (cont’d)

Piggybacking: Allows more efficient bidirectional communication

Congestion Control

TCP Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>16 bit port identifier for each packet</td>
</tr>
<tr>
<td>Destination Port</td>
<td>16 bit port identifier for each packet</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>The packet's unique sequence ID</td>
</tr>
<tr>
<td>Acknowledgement number</td>
<td>The sequence number of the next packet expected by the receiver</td>
</tr>
<tr>
<td>Window Size</td>
<td></td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>0 or more 32-bit words</td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

TCP Header Fields

- Source & Destination Ports
- Sequence number
- Acknowledgement number
**TCP Header Fields (cont’d)**
- **Window size**
  - Specifies how many bytes may be sent after the first acknowledged byte
- **Checksum**
  - checksums the TCP header and IP address fields
- **Urgent Pointer**
  - Points to urgent data in the TCP data field

**TCP Header Fields (cont’d)**
- **Header bits**
  - URG = Urgent pointer field in use
  - ACK = Indicates whether frame contains acknowledgment
  - PSH = Data has been "pushed". It should be delivered to higher layers right away.
  - RST = Indicates that the connection should be reset
  - SYN = Used to establish connections
  - FIN = Used to release a connection

**Principles of Congestion Control**

**Congestion:**
- Informally: “too many sources sending too much data too fast for network to handle”
- Different from flow control!
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)
- A top-10 problem!

**TCP Congestion Control**
- Recall: Network layer is responsible for congestion control
- However, TCP/IP blurs the distinction
- In TCP/IP:
  - The network layer (IP) simply handles routing and packet forwarding
  - Congestion control is done end-to-end by TCP
TCP Congestion Control

- Goal: fully (fairly) utilize the resource (bandwidth)
  - Don't over use - congestion
  - Don't under use - waste
- Goal: achieve self-clocking state
  - Even if don't know bandwidth of bottleneck
  - Bottleneck may change over time

TCP Congestion Window

- TCP introduces a second window, called the “congestion window”
- This window maintains TCP’s best estimate of amount of outstanding data to allow in the network to achieve self-clocking
- Sending size = min(congestion control window, flow control window)

TCP Congestion Control

- Two phases to keep bottleneck busy (fully utilize the resource):
  - Increase the usage (window size) to keep probing the network
  - Decrease the usage when congestion is detected

TCP Slow Start

- When connection begins, CongWin = 1 MSS
  - Example: MSS = 500 bytes
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
  - Increase exponentially until first loss

MSS - “maximum segment size”, the maximum size a TCP packet can be (including header)
TCP Slow Start (more)

- Incrementing CongWin for every ACK received
- Double CongWin every RTT

Initial rate is slow but ramps up exponentially fast.

TCP Slow Start (cont'd)

- Congestion detection
  - Packet losses
  - Sender side Timeout
- Timeout
  - The congestion window is reduced to 1 MSS
  - Everything starts over

TCP Linear Increase

- Don’t push the network too fast
- Slow start (exponential)
  - Threshold → linear increase
TCP Linear Increase Algorithm

**Algorithm:**
- Start the threshold at 64K
- Slow start
- Once the threshold is passed
  - For each ack received, $cwnd = cwnd + (mss \times mss) / cwnd$
    - 1 MSS for each congestion window of data transmitted
  - Timeout
    - reset the congestion window size to 1 MSS
    - Set threshold to max$(2 \times mss, 1/2 \times \text{MIN} \text{(sliding window, congestion window)})$

TCP Linear Increase Threshold Phase

Example: Maximum segment size = 1K
Assume thresh=32K

TCP Congestion Control

- Can we do better at detecting congestion than using timeout?
- Receiver sends duplicate ACKs for out-of-order packets
  - Possible loss?

TCP Fast Retransmit

- Idea: When sender sees 3 duplicate ACKs, it assumes something went wrong
- The packet is immediately retransmitted instead of waiting for it to timeout
**TCP Fast Retransmit Example**

- MSS = 1K
- Fast Retransmit occurs (2nd packet is now retransmitted w/o waiting for it to timeout)
- ACK of new data
  - Duplicate ACK #1
  - Duplicate ACK #2
  - Duplicate ACK #3

**TCP Recap**

- **Timeout Computation**
  - Timeout is a function of 2 values
    - The weighted average of sampled RTTs
    - The sampled variance of each RTT

- **Congestion control:**
  - Goal: Keep the self-clocking pipe full in spite of changing network conditions
  - 3 key Variables:
    - Sliding window (Receiver flow control)
    - Congestion window (Sender flow control)
    - Threshold (Sender’s slow start vs. linear mode line)

**TCP Recap (cont)**

- **Slow start**
  - Add 1 segment for each ACK to the congestion window
    - Double’s the congestion window’s volume each RTT

- **Linear mode (Congestion Avoidance)**
  - Add 1 segment’s worth of data to each congestion window
  - Adds 1 segment per RTT

**Algorithm Summary: TCP Congestion Control**

- When CongWin is below Threshold, sender in slow-start phase, window grows exponentially.
- When CongWin is above Threshold, sender in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, Threshold set to max(FlightSize/2, 2*mss) and CongWin set to Threshold+3*mss. (Fast retransmit, Fast recovery)
- When timeout occurs, Threshold set to max(FlightSize/2, 2*mss) and CongWin is set to 1 MSS.

**FlightSize**: The amount of data that has been sent but not yet acknowledged.
### TCP sender congestion control

<table>
<thead>
<tr>
<th>Event</th>
<th>State</th>
<th>TCP Sender Action</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK receipt for previously unacked data</td>
<td>Slow Start (SS)</td>
<td>CongWin + CongWin + MSS, If (CongWin &gt; Threshold) set state to &quot;Congestion Avoidance&quot;</td>
<td>Resulting in a doubling of CongWin every RTT</td>
</tr>
<tr>
<td>ACK receipt for previously unacked data</td>
<td>Congestion Avoidance (CA)</td>
<td>CongWin + CongWin*MSS * (MSS/CongWin)</td>
<td>Additive increase, resulting in increase of CongWin by 1 MSS every RTT</td>
</tr>
<tr>
<td>Loss event detected by triple duplicate ACK</td>
<td>SS or CA</td>
<td>Threshold = max(FlightSize/2,2<em>MSS), CongWin = Threshold - 3</em>MSS, Set state to &quot;Congestion Avoidance&quot;</td>
<td>Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.</td>
</tr>
<tr>
<td>Timeout</td>
<td>SS or CA</td>
<td>Threshold = max(FlightSize/2,2*MSS), CongWin = 1 MSS, Set state to &quot;Slow Start&quot;</td>
<td>Enter slow start</td>
</tr>
<tr>
<td>Duplicate ACK</td>
<td>SS or CA</td>
<td>Increment duplicate ACK count for segment being acked</td>
<td>CongWin and Threshold not changed</td>
</tr>
</tbody>
</table>
