These slides:

- Introduce transport layer functions
- Presents the TCP protocol
- Presents the TCP congestion control algorithms
- Illustrates via tcpdump traces TCP Connections
  - involving flow control
  - that demonstrate TCP connection establishment, slow-start sending rate growth, and connection tear down
  - That demonstrate TCP loss event recovery by triple duplicate ACK (fast retransmission and fast recovery).
Services Offered to an Application

• Connection orientation
• Point-to-Point
• Reliability
• Full duplex
• Stream interface
• Reliable startup
• Graceful connection shutdown
Transmission Control Protocol (TCP)

- TCP is an end-to-end protocol because it provides a connection directly from one application to another running on a remote computer.
- The connections are virtual connections because they are achieved in software.
IP Datagram Format: UDP example

This is what gets sent ‘on the wire’: A frame which contains an IP Packet
TCP Datagram Format

MAC Header  IP Header  TCP Header  TCP Application Data

TCP Segment

First bit

Last bit

SOURCE PORT  DESTINATION PORT
SEQUENCE NUMBER
ACKNOWLEDGEMENT NUMBER
HLEN  NOT USED  CODE BITS  WINDOW
CHECKSUM  URGENT POINTER
OPTIONS (if any)
BEGINNING OF DATA

First bit

TCP Segment
Packet Loss and Retransmission

Example of retransmission. Items on the left correspond to events in a computer sending data, items on the right correspond to events in a computer receiving data, and time goes down the figure. The sender retransmits lost data.
Transmission Control Protocol (TCP)

- Error detection/recovery
- Adaptive retransmission:
  - If the retransmission timeout is too low, you might retransmit unnecessarily.
  - If the timeout is too high, user perceived performance can be very bad.
Flow Control

- A TCP receiver allocates a receive buffer.
- As data arrives (and fills the buffer), the receiver sends ACKs that also specify the remaining buffer size.
- The amount of buffer space available at any time is the window and the notification that specifies the size is the window advertisement.
Flow Control

Sender Events
- send data octets 1-1000
- send data octets 1001-2000
- send data octets 2001-2500
- receive ack for 1000
- receive ack for 2000
- receive ack for 2500

Receiver Events
- advertise window=2500
- ack up to 1000, window=1500
- ack up to 2000, window=500
- ack up to 2500, window=0
- application reads 2000 octets
- ack up to 2500, window=2000

Sender Events
- send data octets 2501-3500
- send data octets 3501-4500
- receive ack for 3500
- receive ack for 4500

Receiver Events
- ack up to 3500, window=1000
- ack up to 4500, window=0
- application reads 1000 octets
- ack up to 4500, window=1000

Receiver Events
- ...
Network Congestion Control

- What if Host A sends many back-to-back UDP Echo Messages to Host B?
- TCP’s base congestion control algorithm based on a dynamic window (slow start)
  - \( cwnd = 1; \)  //init the congestion window to 1 segment
  - \( \text{wnd} = \min(cwnd, \text{advertised window}) \)  //The actual window used…
  - On a timeout, \( cwnd = 1 \)
  - When a new ACK arrives
    - \( cwnd += 1; \)
- Leads to TCP’s ‘saw-tooth behavior’
## TCP Sockets Programming

<table>
<thead>
<tr>
<th>TCP Client</th>
<th>TCP Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket()</td>
<td>socket()</td>
</tr>
<tr>
<td>connect()</td>
<td>bind()</td>
</tr>
<tr>
<td></td>
<td>listen()</td>
</tr>
<tr>
<td>write()</td>
<td>accept()</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>read()</td>
<td>write()</td>
</tr>
<tr>
<td>close()</td>
<td>read()</td>
</tr>
<tr>
<td></td>
<td>close()</td>
</tr>
</tbody>
</table>
Server Side Calls: Listen and Accept

- **Listen**
  - Converts an active socket into a passive socket meaning the kernel should accept incoming connection requests.
  - Sets the maximum number of connections the kernel should queue for this socket.
    
    \[ \text{int listen (int sockfd, int backlog)} \]
  - There are two queues:
    - incomplete cx queue
    - completed cx queue
  - the backlog (roughly) indicates the sum of the two queues
- **Accept** returns a Cx from the completed queue
  
  \[ \text{int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addren)} \]
  - Returns a new socket descriptor.
    - Server will have at least a listenfd and a connectfd socket descriptors
TCP startup/tear down

- **TCP connection setup:** 3 way handshake
  - guarantees that both sides are ready to transfer data (handles simultaneous opens)
  - Allows both sides to agree on initial sequence numbers (ISNs).

- **TCP connection termination:**
  - modified 3 way handshake
  - The TCP close requires 4 flows rather than 3
  - A FIN leads to an EOF to a receiving application
  - Supports a half close: one side terminates its output but still receives data from the other side.
  - Sockets supports this- but most applications don’t use it.
TCP State Machine

Figure 13.15 The TCP finite state machine. Each endpoint begins in the closed state. Labels on transitions show the input that caused the transition followed by the output if any.
TCP Acknowledgement Strategy

- Acks indicate the next byte the receiver expects (we show the next packet that is expected)
- Acks are cumulative
  - Easy to generate
  - But they are ambiguous.
  - Lost Acks don’t necessarily force a retransmission.

- Ack strategy: an Ack is typically generated for every 2 segments that arrive
Congestion Avoidance Algorithm

- Combined slow start and congestion avoidance algorithm (Tahoe)
- Create a second state variable, ssthresh, to switch between the two algorithms.
- Assume the \( \text{wnd} = \min(\text{cwnd}, \text{advertised window}) \)
- On a timeout
  \[ \text{ssthresh} = \frac{\text{wnd}}{2} \text{ (min value of 2 segments)} \]
  \[ \text{cwnd} = 1 \]
- When a new ACK arrives
  if (cwnd \( \leq \) ssthresh)
    /* open the window exponentially */
    cwnd += 1;
  else
    /* otherwise do Congestion Avoidance */
    increment linearly */
    cwnd += 1/cwnd;
Fast Recovery, Fast Retransmit Algorithms

- **Fast Retransmit**:  
  - If three or more duplicate ACKs arrive at a sender, this is a strong indicator that a packet was dropped.  
  - The sender retransmits without waiting for the retransmit timer.

- **Fast Recovery**: After a fast retransmit, the sender goes to congestion avoidance rather than slow-start.

- **Enhanced algorithm**:  
  - When the third duplicate ACK is received  
    - set ssthresh = cwnd/2  
    - retransmit the segment  
    - set cwnd = ssthresh + 3 packets  
  - For each additional duplicate ACK (after the third duplicate ACK), increment cwnd by 1 and transmit a new packet (if allowed by the new cwnd value).  
  - When the next ACK arrives that acknowledges new data, set cwnd to ssthresh.
The TCP Congestion Control terminology

- **Four key terms**
  1. **Slow start** – relies on the cwnd variable. Is meant to get the Cx to find the knee.
  2. **Congestion avoidance** – once cwnd exceeds ssthresh, the Cx needs to be careful because it is now operating within the knee.
  3. **Fast retransmission** – a way to recover from loss faster by letting the sender assume a loss event has occurred when three duplicate ACKs arrive.
  4. **Fast recovery** – minor tweak to the algorithm to inflate the cwnd (add 1 for each duplicate ack that arrives) to try to avoid a timeout. After the loss event is corrected, the cwnd is set to cwnd /2 based on the value of cwnd prior to the loss event.

Items 1-3 collectively are referred to as TCP/Tahoe.

Items 1-4 collectively are referred to as TCP/Reno.
Two Modes of Congestion Control

1. Probing for the available bandwidth
   - slow start \((cwnd < ssthresh)\)

2. Avoid overloading the network
   - congestion avoidance \((cwnd \geq ssthresh)\)

   • Hint to understanding this:
     • Slow start - Once TCP detects loss while in SS, it remembers where the knee begins by recording ssthresh = cwnd / 2.
     • Congestion avoidance – While operating with cwnd \(\geq ssthresh\) it operates in CA mode.
Key to understanding TCP CC

- Behaviors while in slow start or Congestion Avoidance:
  - While in slow start: Each new ACK increments cwnd by 1 (exponential growth in sending rate).
  - While in CA, each new ACK increments cwnd by 1/cwnd (linear growth)
- Determining the 'switching point' between the two modes:
  - Ssthresh determines the mode (slow start if cwnd < ssthresh)
  - Ssthresh is updated each time a loss event occurs
    - A loss event might consist of multiple packets dropped but it represents one congestion event to TCP
    - Once the loss event is over (i.e., recovery has completed) ssthresh is set to the value cwnd/2 with the value of cwnd saved at the beginning of the loss event.
  - The algorithm tries to adapt to current conditions – meaning the optimal W at any given point is unknown and further is always a moving target.
Slow Start

• Cwnd is initialized to 1 MSS (maximum segment size is a similar topic to the MTU or max transmission unit - it’s the largest amount of data TCP should place into a IP datagram. The MSS is usually set to avoid IP fragmentation.

• Modern TCP implementation may set initial cwnd to a much larger value

• When receiving an ACK, cwnd+= 1 MSS

• Again, in our discussions we simplify this by saying the cwnd is in units of packets rather than in bytes.
  • So, when a new ACK (as opposed to a duplicate ACK, cwnd+=1
If $cwnd \geq ssthresh$ then each time an ACK is received, increment $cwnd$ linearly:

- $cwnd += MSS \times (MSS / cwnd)$ (cwnd measured in bytes)
- Again, in our discussions, $cwnd += 1/cwnd$

Logic behind this:

- In an RTT time period, $cwnd$ packets are sent and acknowledged. So at the end of the RTT period: $cwnd$ will have increased $cwnd \times 1/cwnd = 1$
- This means during CA, the $cwnd$ is increased by 1 segment each RTT
- This translates to linear growth in the sending rate (remember, during slow start, the send rate grows exponentially).
- So, by definition, during CA, senders are very timid with how they behave because they know they are near the knee of the load vs throughput curve!
TCP Congestion Control

TCP is designed to operate around the knee
- Slow start allows a new Cx to find the knee (meaning estimate the W that is a good operating point)
- Congestion avoidance allows TCP to oscillate around the knee
Example of Slow Start/Congestion Avoidance

Assume $ssthresh = 8$ MSS
Congestion detection

• What would happen if a sender keeps increasing $cwnd$?
  – Packet loss

• TCP uses packet loss as a congestion signal

• Loss detection
  1. Receipt of a duplicate ACK (cumulative ACK)
  2. Timeout of a retransmission timer
Reaction to Congestion: Timeouts

- Timeout: Indicates severe congestion
  - cwnd is reset to one MSS:
    \[ \text{cwnd} = 1 \text{ MSS} \]
  - ssthresh is set to half of the current size of the congestion window:
    \[ \text{ssthresh} = \text{cwnd} / 2 \]
  - Since cwnd < ssthresh, it goes to slow start
Fast Retransmission

- Duplicate ACKs: Assumes the network is not as congested as if a timeout occurs.
- Fast retransmit
  - Three duplicate ACKs indicate a packet loss
  - Retransmit without timeout

Typically very effective at avoiding the traditional saw-tooth behavior, and allows TCP to oscillate around the knee.
Reaction to congestion: Fast Recovery

• Avoiding slow start (changed from TCP88)
  – ssthresh = cwnd/2
  – cwnd = cwnd + 3MSS
  – Increase cwnd by one MSS for each additional duplicate ACK

• When ACK arrives that acknowledges “new data,” set:
  cwnd = ssthresh
  enter congestion avoidance
**Congestion Avoidance Algorithm**

- Combined slow start and congestion avoidance algorithm (Tahoe)
- Create a second state variable, ssthresh, to switch between the two algorithms.
- Assume the $\text{wnd} = \min(\text{cwnd, advertised window})$
- On a timeout
  - $\text{ssthresh} = \text{wnd} / 2$ (min value of 2 segments)
  - $\text{cwnd} = 1$
- When a new ACK arrives
  - if ($\text{cwnd} \leq \text{ssthresh}$)
    - /*open the window exponentially*/
      - $\text{cwnd} += 1$
    - else
      - /*otherwise do Congestion Avoidance*/
      - $\text{cwnd} += 1/\text{cwnd}$
Fast Recovery, Fast Retransmit Algorithms

- **Fast Retransmit:**
  - If three or more duplicate ACKs arrive at a sender, this is a strong indicator that a packet was dropped.
  - The sender retransmits without waiting for the retransmit timer.

- **Fast Recovery:** After a fast retransmit, the sender goes to congestion avoidance rather than slow-start.

- **Enhanced algorithm:**
  - When the third duplicate ACK is received:
    - set ssthresh = cwnd/2
    - retransmit the segment
  - set cwnd = ssthresh + 3 packets
  - For each additional duplicate ACK (after the third duplicate ACK), increment cwnd by 1 and transmit a new packet (if allowed by the new cwnd value).
  - When the next ACK arrives that acknowledges new data, set cwnd to ssthresh.
Tcpdump illustration of flow control

192.168.1.100 212.208.230.29
Host A-----------------------------Host B

The following shows the three-way handshake

1. 982120155.354974 192.168.1.100.1279 > 212.208.230.29.45912: S 143851464:143851464(0) win 16384 <mss 1460> (DF)
2. 982120155.567089 212.208.230.29.45912 > 192.168.1.100.1279: S 2709967313:2709967313(0) ack 143851465 win 32120 <mss 1460> (DF)
3. 982120155.567271 192.168.1.100.1279 > 212.208.230.29.45912: . ack 1 win 17520 (DF)

The application is a web browser (at Host A) opens a web cx with Host B that initiates a download of a file that is at least 17,520 bytes large

4. 982120156.248161 212.208.230.29.45912 > 192.168.1.100.1279: P 1:1461(1460) ack 1 win 32120 (DF) [tos 0x10]
5. 982120156.258408 212.208.230.29.45912 > 192.168.1.100.1279: P 1461:2921(1460) ack 1 win 32120 (DF) [tos 0x10]
6. 982120156.266857 192.168.1.100.1279 > 212.208.230.29.45912: . ack 2921 win 14600 (DF)
TCPDump Trace

7. 982120156.248161 212.208.230.29.45912 > 192.168.1.100.1279: P 1:1461(1460) ack 1 win 32120 (DF) [tos 0x10]
8. 982120156.258408 212.208.230.29.45912 > 192.168.1.100.1279: P 1461:2921(1460) ack 1 win 32120 (DF) [tos 0x10]
10. 982120156.487286 212.208.230.29.45912 > 192.168.1.100.1279: P 2921:4381(1460) ack 1 win 32120 (DF) [tos 0x10]
11. 982120156.496933 212.208.230.29.45912 > 192.168.1.100.1279: P 4381:5841(1460) ack 1 win 32120 (DF) [tos 0x10]
12. 982120156.507092 212.208.230.29.45912 > 192.168.1.100.1279: P 5841:7301(1460) ack 1 win 32120 (DF) [tos 0x10]
13. 982120156.666886 192.168.1.100.1279 > 212.208.230.45912: . ack 7301 win 10220 (DF)
14. 982120156.936903 212.208.230.29.45912 > 192.168.1.100.1279: P 7301:8761(1460) ack 1 win 32120 (DF) [tos 0x10]
15. 982120156.951005 212.208.230.29.45912 > 192.168.1.100.1279: P 8761:10221(1460) ack 1 win 32120 (DF) [tos 0x10]
16. 982120156.963396 212.208.230.29.45912 > 192.168.1.100.1279: P 10221:11681(1460) ack 1 win 32120 (DF) [tos 0x10]
17. 982120156.982527 212.208.230.29.45912 > 192.168.1.100.1279: P 11681:13141(1460) ack 1 win 32120 (DF) [tos 0x10]
18. 982120157.066889 192.168.1.100.1279 > 212.208.230.45912: . ack 13141 win 4380 (DF)
19. 982120157.299244 212.208.230.29.45912 > 192.168.1.100.1279: P 13141:14601(1460) ack 1 win 32120 (DF) [tos 0x10]
20. 982120157.314335 212.208.230.29.45912 > 192.168.1.100.1279: P 14601:16061(1460) ack 1 win 32120 (DF) [tos 0x10]
21. 982120157.321023 212.208.230.29.45912 > 192.168.1.100.1279: P 16061:17521(1460) ack 1 win 32120 (DF) [tos 0x10]
22. 982120157.466886 192.168.1.100.1279 > 212.208.230.45912: . ack 17521 win 0 (DF)
23. 982120158.988267 212.208.230.29.45912 > 192.168.1.100.1279: . ack 1 win 32120 (DF) [tos 0x10]
24. 982120158.998477 192.168.1.100.1279 > 212.208.230.45912: . ack 17521 win 0 (DF)
25. 982120161.846314 212.208.230.29.45912 > 192.168.1.100.1279: . ack 1 win 32120 (DF) [tos 0x10]
26. 982120161.846590 192.168.1.100.1279 > 212.208.230.45912: . ack 17521 win 0 (DF)
27. 982120162.258374 192.168.1.100.1279 > 212.208.230.45912: . ack 17521 win 10240 (DF)
28. 982120162.266653 192.168.1.100.1279 > 212.208.230.45912: . ack 17521 win 17520 (DF)
Tcpdump illustration of tcp slow start

See the tcpdump examples8.pdf - shows 10000 bytes sent and received using the TCPSend application.

Host A ------ wireless net ------ Host B
Jjmnuc2          jjmnuc1
TCPTx jjmnuc1 5000 10000      TCPRx 5000

Example1 – no loss
Example2 – no loss
Trace 3 - the first 2 segments are lost
Trace 4 – two segments in the middle of the transfer are lost

Remember the params to TCPTx
TCPTx: servername service dataSize chunkSize pipeSize
TCPRx service chunkSize pipeSize

dataSize: Number of bytes sent by the TCPTx. If 0, sends forever
chunkSize : specifies the max amount of data sent each socket send(). Therefore the TCPTx loop iterations dataSize/chunkSize times (possibly one more time to send the remainder of dataSize/chunkSize)

pipeSize: Sets the size of the endpoints socket Tx and Rx buffers. Setting the socket Rx Buffer size has the most impact as it will limit the TCPTx throughput over large RTT paths. Setting the pipeSize to a value of an MSS (1 segment, such as 1448) causes TCP to behave in a stop-and-wait manner.
TCP Congestion Control

The figures show the dynamics within a network as the total load continues to increase:

- Top curve shows the throughput follows load up to about 90% of capacity.
- This is the knee – packet loss begins to occur.
- If load continues to increase, the throughput can not exceed the capacity.
- If congestion control does not kick in to reduce the load the network falls into ‘congestion collapse’ where all resources are used primarily for retransmission and goodput drops potentially to 0.

Design Goals

- Congestion avoidance: making the system operate around the knee to obtain low latency and high throughput.
- Congestion control: making the system operate left to the cliff to avoid congestion collapse.
TCP SACK – Selective Acknowledgement

Both TCP endpoints indicate if they will accept TCP-SACK in the three-way handshake when they negotiate TCP options

- A TCP receiver that supports TCP-SACK and that knows the TCP sender (its TCP peer) supports SACK, will include in ACK messages blocks of segments that have successfully arrived and blocks of segments that did not arrive.

TERMs: TCP-SACK and Gap: TCP-SACK is TCP’s selective repeat ACK strategy. A Gap observed by a TCP Receiver is a stream of octets with a group of octets missing

- Example let’s say a TCP Receiver observes the following:
  - Segments 1 – 100 arrive
  - 101-110 do not arrive (Gap 1)
  - 111-113 arrive (Block 1)
  - 114-115 do not arrive (Gap 2)
  - 116-200 arrive and completes the TCP Cx
- Possible recovery: sometime after 116 arrives, the SACK ACK causes the sender to retransmit all segments from Gap 1 and Gap 2.

- This would be a very normal scenario - multiple packet loss within the same RTT time period is difficult to recover using just fast retransmission
- In this example there are two gaps.
- When segment 116 arrives, the receiver would issue a TCP ACK that indicated the segments in Gap 1 and Gap 2 did not arrive and that the segments in Block 1 arrived. This single SACK could cause the Sender to retransmit all missing packets which, once they successfully arrive and are ACK’ed would complete the error recover.

- By default, Linux uses a variant of TCP/Reno called TCP/Cubic (optimized for high speed throughput over long distance, high speed connections, also referred to as big fat pipes).
- TCP-SACK is turned on by default
- The sysctl command allows an admin to change TCP’s CC algorithm, tweak many parameters, disable tcp-SACK, set the TCP MSS size