Lecture 9: UDP/TCP
Transport Layer

- **Function:**
  - Demultiplexing of data streams

- **Optional functions:**
  - Creating long lived connections
  - Reliable, in-order packet delivery
  - Error detection
  - Flow and congestion control

- **Key challenges:**
  - Detecting and responding to congestion
  - Balancing fairness against high utilization
UDP
TCP
The Case for Multiplexing

- Datagram network
  - No circuits
  - No connections
The Case for Multiplexing

- Datagram network
  - No circuits
  - No connections
- Clients run many applications at the same time
  - Who to deliver packets to?
The Case for Multiplexing

- Datagram network
  - No circuits
  - No connections
- Clients run many applications at the same time
  - Who to deliver packets to?
- IP header “protocol” field
  - 8 bits = 256 concurrent streams
The Case for Multiplexing

- Datagram network
  - No circuits
  - No connections
- Clients run many applications at the same time
  - Who to deliver packets to?
- IP header “protocol” field
  - 8 bits = 256 concurrent streams
- Insert Transport Layer to handle demultiplexing
Demultiplexing Traffic

Host 1

Host 2

Host 3

Application
Demultiplexing Traffic

Unique port for each application
Demultiplexing Traffic

Applications share the same network.
Server applications communicate with multiple clients.
Endpoints identified by \(<src\_ip, src\_port, dest\_ip, dest\_port>\)
Layering, Revisited

- Lowest level end-to-end protocol (in theory)
  - Transport header only read by source and destination
  - Routers view transport header as payload
Layering, Revisited

- Lowest level end-to-end protocol (in theory)
  - Transport header only read by source and destination
  - Routers view transport header as payload
- Lowest level end-to-end protocol (in theory)
  - Transport header only read by source and destination
  - Routers view transport header as payload
**User Datagram Protocol (UDP)**

- Simple, connectionless datagram
  - C sockets: SOCK_DGRAM
- Port numbers enable demultiplexing
  - 16 bits = 65535 possible ports
  - Port 0 is invalid
- Checksum for error detection
  - Detects (some) corrupt packets
  - Does not detect dropped, duplicated, or reordered packets
Uses for UDP

- Invented after TCP
  - Why?
Uses for UDP

- Invented after TCP
  - Why?
- Not all applications can tolerate TCP
Uses for UDP

- Invented after TCP
  - Why?
- Not all applications can tolerate TCP
- Custom protocols can be built on top of UDP
  - Reliability? Strict ordering?
  - Flow control? Congestion control?
- Examples
  - RTMP, real-time media streaming (e.g. voice, video)
  - Facebook datacenter protocol
9 Outline

- UDP
- TCP
Transmission Control Protocol

- Reliable, in-order, bi-directional byte streams
- Port numbers for demultiplexing
- Virtual circuits (connections)
- Flow control
- Congestion control, approximate fairness

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>Destination Port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgement Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLen</td>
<td>Flags</td>
<td>Advertised Window</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
<td>Urgent Pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Options</td>
<td></td>
</tr>
</tbody>
</table>
Transmission Control Protocol

- Reliable, in-order, bi-directional byte streams
  - Port numbers for demultiplexing
  - Virtual circuits (connections)
  - Flow control
  - Congestion control, approximate fairness

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Source Port</strong></td>
<td><strong>Destination Port</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sequence Number</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Acknowledgement Number</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HLen</strong></td>
<td><strong>Flags</strong></td>
<td><strong>Advertised Window</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Checksum</strong></td>
<td></td>
<td><strong>Urgent Pointer</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Options</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why do we need connection setup?
Why do we need connection setup?

- To establish state on both hosts
- Most important state: sequence numbers
  - Count the number of bytes that have been sent
  - Initial value chosen at random
  - Why?
Connection Setup

- Why do we need connection setup?
  - To establish state on both hosts
  - Most important state: sequence numbers
    - Count the number of bytes that have been sent
    - Initial value chosen at random
    - Why?

- Important TCP flags (1 bit each)
  - SYN – synchronization, used for connection setup
  - ACK – acknowledge received data
  - FIN – finish, used to tear down connection
Three Way Handshake

Each side:
- Notifies the other of starting sequence number
- ACKs the other side’s starting sequence number
Three Way Handshake

- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side’s starting sequence number
Three Way Handshake

- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side’s starting sequence number
Three Way Handshake

Each side:
- Notifies the other of starting sequence number
- ACKs the other side’s starting sequence number
Three Way Handshake

- Each side:
  - Notifies the other of starting sequence number
  - ACKs the other side’s starting sequence number

- Why Sequence # +1?

Client

Server

SYN <SeqC, 0>

SYN/ACK <SeqS, SeqC+1>

ACK <SeqC+1, SeqS+1>
Connection Setup Issues

- Connection confusion
  - How to disambiguate connections from the same host?
  - Random sequence numbers
Connection Setup Issues

- Connection confusion
  - How to disambiguate connections from the same host?
  - Random sequence numbers

- Source spoofing
  - Kevin Mitnick
  - Need good random number generators!
Connection Setup Issues

- Connection confusion
  - How to disambiguate connections from the same host?
  - Random sequence numbers

- Source spoofing
  - Kevin Mitnick
  - Need good random number generators!

- Connection state management
  - Each SYN allocates state on the server
  - SYN flood = denial of service attack
  - Solution: SYN cookies
Either side can initiate tear down
Either side can initiate tear down

Client

FIN <SeqA, *>  

ACK <*, SeqA+1>

Server
Connection Tear Down

- Either side can initiate tear down
- Other side may continue sending data
  - Half open connection
  - `shutdown()`
Connection Tear Down

- Either side can initiate tear down
- Other side may continue sending data
  - Half open connection
  - `shutdown()`
- Acknowledge the last FIN
  - Sequence number + 1

```
Client
```

```
Server
```
TCP uses a byte stream abstraction
- Each byte in each stream is numbered
- 32-bit value, wraps around
- Initial, random values selected during setup
 Sequence Number Space

- TCP uses a byte stream abstraction
  - Each byte in each stream is numbered
  - 32-bit value, wraps around
  - Initial, random values selected during setup
- Byte stream broken down into segments (packets)
  - Size limited by the Maximum Segment Size (MSS)
  - Set to limit fragmentation
TCP uses a byte stream abstraction
- Each byte in each stream is numbered
- 32-bit value, wraps around
- Initial, random values selected during setup

Byte stream broken down into segments (packets)
- Size limited by the Maximum Segment Size (MSS)
- Set to limit fragmentation

Each segment has a sequence number
Bidirectional Communication

- Each side of the connection can send and receive
  - Different sequence numbers for each direction
Bidirectional Communication

- Each side of the connection can send and receive
  - Different sequence numbers for each direction

<table>
<thead>
<tr>
<th>Seq.</th>
<th>Ack.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seq.</th>
<th>Ack.</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>1</td>
</tr>
</tbody>
</table>
Bidirectional Communication

- Each side of the connection can send and receive
  - Different sequence numbers for each direction

**Diagram:**

- Client
  - Seq.: 1
  - Ack.: 23

- Server
  - Seq.: 23
  - Ack.: 1

- Data (1460 bytes)
Bidirectional Communication

- Each side of the connection can send and receive
  - Different sequence numbers for each direction

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>1461</td>
<td>753</td>
<td>23</td>
<td>1461</td>
</tr>
</tbody>
</table>

Data (1460 bytes) and ACK (730 bytes) in the same packet
Each side of the connection can send and receive

- Different sequence numbers for each direction
Flow Control

- Problem: how many packets should a sender transmit?
  - Too many packets may overwhelm the receiver
  - Size of the receivers buffers may change over time
Problem: how many packets should a sender transmit?
- Too many packets may overwhelm the receiver
- Size of the receivers buffers may change over time

Solution: sliding window
- Receiver tells the sender how big their buffer is
- Called the *advertised* window
- For window size $n$, sender may transmit $n$ bytes without receiving an ACK
- After each ACK, the window slides forward
Flow Control

- **Problem:** how many packets should a sender transmit?
  - Too many packets may overwhelm the receiver
  - Size of the receiver's buffers may change over time

- **Solution:** sliding window
  - Receiver tells the sender how big their buffer is
  - Called the *advertised window*
  - For window size \( n \), sender may transmit \( n \) bytes without receiving an ACK
  - After each ACK, the window slides forward

- Window may go to zero!
## Flow Control: Sender Side

### Packet Sent

<table>
<thead>
<tr>
<th>Src. Port</th>
<th>Dest. Port</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence Number</strong></td>
<td></td>
</tr>
<tr>
<td>Acknowledgement Number</td>
<td></td>
</tr>
<tr>
<td>HL</td>
<td>Flags</td>
</tr>
<tr>
<td>Window</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent Pointer</td>
</tr>
</tbody>
</table>

### Packet Received

<table>
<thead>
<tr>
<th>Src. Port</th>
<th>Dest. Port</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence Number</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Acknowledgement Number</strong></td>
<td></td>
</tr>
<tr>
<td>HL</td>
<td>Flags</td>
</tr>
<tr>
<td>Window</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent Pointer</td>
</tr>
</tbody>
</table>
Flow Control: Sender Side

Packet Sent

<table>
<thead>
<tr>
<th>Src. Port</th>
<th>Dest. Port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence Number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgement Number</td>
<td></td>
</tr>
<tr>
<td>HL</td>
<td>Flags</td>
</tr>
<tr>
<td>Window</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent Pointer</td>
</tr>
</tbody>
</table>

Packet Received

<table>
<thead>
<tr>
<th>Src. Port</th>
<th>Dest. Port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence Number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgement Number</td>
<td></td>
</tr>
<tr>
<td>HL</td>
<td>Flags</td>
</tr>
<tr>
<td>Window</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent Pointer</td>
</tr>
</tbody>
</table>

App Write
Flow Control: Sender Side

Packet Sent

- Src. Port
- Dest. Port
- Sequence Number
- Acknowledgement Number
- HL
- Flags
- Window
- Checksum
- Urgent Pointer

Packet Received

- Src. Port
- Dest. Port
- Sequence Number
- Acknowledgement Number
- HL
- Flags
- Window
- Checksum
- Urgent Pointer

ACKed

App Write
Flow Control: Sender Side

Packet Sent
- Src. Port
- Dest. Port
- Sequence Number
- Acknowledgement Number
- HL
- Flags
- Window
- Checksum
- Urgent Pointer

Packet Received
- Src. Port
- Dest. Port
- Sequence Number
- Acknowledgement Number
- HL
- Flags
- Window
- Checksum
- Urgent Pointer

ACKed
Sent
App Write
### Flow Control: Sender Side

<table>
<thead>
<tr>
<th>Packet Sent</th>
<th>Packet Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>Src. Port</td>
<td>Dest. Port</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>Sequence Number</td>
</tr>
<tr>
<td>Acknowledgement Number</td>
<td>Acknowledgement Number</td>
</tr>
<tr>
<td>HL</td>
<td>HL</td>
</tr>
<tr>
<td>Flags</td>
<td>Flags</td>
</tr>
<tr>
<td>Window</td>
<td>Window</td>
</tr>
<tr>
<td>Checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>Urgent Pointer</td>
</tr>
</tbody>
</table>

- **Must be buffered until ACKed**

- **ACKed**
- **Sent**
- **App Write**
## Flow Control: Sender Side

### Packet Sent

<table>
<thead>
<tr>
<th>Src. Port</th>
<th>Dest. Port</th>
<th>Sequence Number</th>
<th>Acknowledgement Number</th>
<th>HL</th>
<th>Flags</th>
<th>Window</th>
<th>Checksum</th>
<th>Urgent Pointer</th>
</tr>
</thead>
</table>

### Packet Received

<table>
<thead>
<tr>
<th>Src. Port</th>
<th>Dest. Port</th>
<th>Sequence Number</th>
<th>Acknowledgement Number</th>
<th>HL</th>
<th>Flags</th>
<th>Window</th>
<th>Checksum</th>
<th>Urgent Pointer</th>
</tr>
</thead>
</table>

- **ACKed**
- **Sent**
- **To Be Sent**
- **Outside Window**

---

**App Write**
Sliding Window Example
Sliding Window Example
Sliding Window Example
Sliding Window Example

Time

Time
Sliding Window Example
Sliding Window Example
Sliding Window Example
Sliding Window Example
TCP is ACK Clocked

- Short RTT $\rightarrow$ quick ACK $\rightarrow$ window slides quickly
- Long RTT $\rightarrow$ slow ACK $\rightarrow$ window slides slowly
What Should the Receiver ACK?

1. ACK every packet
What Should the Receiver ACK?

1. ACK every packet
2. Use cumulative ACK, where an ACK for sequence $n$ implies ACKS for all $k < n$
3. Use negative ACKs (NACKs), indicating which packet did not arrive
What Should the Receiver ACK?

1. ACK every packet
2. Use *cumulative ACK*, where an ACK for sequence $n$ implies ACKS for all $k < n$
3. Use *negative ACKs* (NACKs), indicating which packet did not arrive
4. Use *selective ACKs* (SACKs), indicating those that did arrive, even if not in order
   - SACK is an actual TCP extension
What Should the Receiver ACK?

1. **ACK every packet**

2. Use **cumulative ACK**, where an ACK for sequence $n$ implies ACKS for all $k < n$

3. Use **negative ACKs** (NACKs), indicating which packet did not arrive

4. Use **selective ACKs** (SACKs), indicating those that did arrive, even if not in order
   - SACK is an actual TCP extension
What Should the Receiver ACK?

1. ACK every packet

2. Use cumulative ACK, where an ACK for sequence $n$ implies ACKS for all $k < n$

3. Use negative ACKs (NACKs), indicating which packet did not arrive

4. Use selective ACKs (SACKs), indicating those that did arrive, even if not in order
   - SACK is an actual TCP extension
Sequence Numbers, Revisited

- 32 bits, unsigned
  - Why so big?
Sequence Numbers, Revisited

- 32 bits, unsigned
  - Why so big?
- For the sliding window you need...
  - $|\text{Sequence # Space}| > 2 \times |\text{Sending Window Size}|$
  - $2^{32} > 2 \times 2^{16}$
Sequence Numbers, Revisited

- **32 bits, unsigned**
  - Why so big?

- **For the sliding window you need…**
  - $|\text{Sequence } \# \text{ Space}| > 2 \times |\text{Sending Window Size}|$
  - $2^{32} > 2 \times 2^{16}$

- **Guard against stray packets**
  - IP packets have a maximum segment lifetime (MSL) of 120 seconds
    - i.e. a packet can linger in the network for 3 minutes
  - Sequence number would wrap around at 286Mbps
Sequence Numbers, Revisited

- 32 bits, unsigned
  - Why so big?
- For the sliding window you need…
  - \( |\text{Sequence \# Space}| > 2 \times |\text{Sending Window Size}| \)
  - \( 2^{32} > 2 \times 2^{16} \)
- Guard against stray packets
  - IP packets have a maximum segment lifetime (MSL) of 120 seconds
    - i.e. a packet can linger in the network for 3 minutes
  - Sequence number would wrap around at 286Mbps
    - What about GigE? PAWS algorithm + TCP options
Problem: what if the window size is very small?
Silly Window Syndrome

- Problem: what if the window size is very small?
  - Multiple, small packets, headers dominate data
Problem: what if the window size is very small?
- Multiple, small packets, headers dominate data

Equivalent problem: sender transmits packets one byte at a time

1. for (int x = 0; x < strlen(data); ++x)
2. write(socket, data + x, 1);
Nagle’s Algorithm

1. If the window $\geq$ MSS and available data $\geq$ MSS: 
   Send the data
2. Elif there is unACKed data: 
   Enqueue data in a buffer (send after a timeout)
3. Else: send the data
Nagle’s Algorithm

1. If the window $\geq$ MSS and available data $\geq$ MSS:
   Send the data
2. Elif there is unACKed data:
   Enqueue data in a buffer (send after a timeout)
3. Else: send the data

Send a non-full packet if nothing else is happening
Nagle’s Algorithm

1. If the window $\geq$ MSS and available data $\geq$ MSS:
   Send the data

2. Elif there is unACKed data:
   Enqueue data in a buffer (send after a timeout)

3. Else: send the data

Send a full packet

Send a non-full packet if nothing else is happening
Nagle’s Algorithm

1. If the window $\geq$ MSS and available data $\geq$ MSS: Send the data
2. Elif there is unACKed data: Enqueue data in a buffer (send after a timeout)
3. Else: send the data

Problem: Nagle’s Algorithm delays transmissions
- What if you need to send a packet immediately?
  1. int flag = 1;
  2. setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (char *) &flag, sizeof(int));
Error Detection

- Checksum detects (some) packet corruption
  - Computed over IP header, TCP header, and data
Error Detection

- Checksum detects (some) packet corruption
  - Computed over IP header, TCP header, and data
- Sequence numbers catch sequence problems
  - Duplicates are ignored
  - Out-of-order packets are reordered or dropped
  - Missing sequence numbers indicate lost packets
Error Detection

- Checksum detects (some) packet corruption
  - Computed over IP header, TCP header, and data

- Sequence numbers catch sequence problems
  - Duplicates are ignored
  - Out-of-order packets are reordered or dropped
  - Missing sequence numbers indicate lost packets

- Lost segments detected by sender
  - Use `timeout` to detect missing ACKs
  - Need to estimate RTT to calibrate the timeout
  - Sender must keep copies of all data until ACK
Retransmission Time Outs (RTO)

- Problem: time-out is linked to round trip time
Retransmission Time Outs (RTO)

- Problem: time-out is linked to round trip time

[Diagram showing initial send with a cross indicating the time-out]

Initial Send
Retransmission Time Outs (RTO)

- Problem: time-out is linked to round trip time
Retransmission Time Outs (RTO)

- Problem: time-out is linked to round trip time
Retransmission Time Outs (RTO)

- Problem: time-out is linked to round trip time
Retransmission Time Outs (RTO)

- Problem: time-out is linked to round trip time

- Diagram:
  - Initial Send
  - Retry
  - ACK
  - Timeout is too short
Problem: time-out is linked to round trip time

Timeout is too short

What about if timeout is too long?
Round Trip Time Estimation
**Round Trip Time Estimation**

- **Original TCP round-trip estimator**
  - RTT estimated as a moving average
  - \( \text{new}_\text{rtt} = \alpha (\text{old}_\text{rtt}) + (1 - \alpha)(\text{new}_\text{sample}) \)
  - Recommended \( \alpha \): 0.8-0.9 (0.875 for most TCPs)
  - \( \text{RTO} = 2 * \text{new}_\text{rtt} \) (i.e. TCP is conservative)
RTT Sample Ambiguity
RTT Sample Ambiguity

Initial Send
Retry
ACK

Initial Send
ACK
Retry
RTT Sample Ambiguity

Sample

RTO

Initial Send

Retry

ACK

Sample?

RTO

Initial Send

ACK

Retry
Karn’s algorithm: ignore samples for retransmitted segments