Transport Layer

- Transport Layer
  - Provides logical communication channels between apps
  - Transport layer managed by end systems
    - Routers are unaware; they provide network layer services
  - Multiple transport protocols available
    - Under IP: TCP, UDP, SCTP, and more

Today, we’ll discuss

- Transport layer multiplexing/demultiplexing
- Reliable data transfer

Transport Layer Multiplexing

- Problem:
  Multiple communication channels over one network link
  - This is a problem whenever a protocol at one layer is used by multiple protocols or communication sessions at a higher layer
Transport Layer Multiplexing

- Problem: Multiple communication channels over one network link
  - This is a problem whenever a protocol at one level is used by multiple protocols or communication session at one level

- Need to identify which segment belongs to which channel

Logical view at the network layer

Multiplexing & Demultiplexing

How is it done?

- Transport layer protocols in IP have port numbers
  - 16 bit integers (0 .. 65535)
  - IP header (network layer) has source address, destination address
  - TCP/UDP headers (transport layer) have source port, destination port

- Each socket is uniquely identified in the operating system

- Before a socket can be used, it is created & named
  - `socket` system call creates a unique socket
  - `bind` system call associates a local address with the socket
    - With an address of INADDR_ANY, the socket is associated with ALL local interfaces
    - With a port of 0, the OS assigns a random unused port number to the socket

Why use UDP?

- Control the timing of data
  - A UDP segment is passed to the network layer immediately for transmission
  - TCP uses congestion control to delay transmission

- Preserve message boundaries
  - With TCP, multiple small messages may be consolidated into one TCP segment

- No connection setup
  - TCP requires a three-way handshake to establish a connection

- No state to keep track of
  - Less memory, easier fault recovery, simple load balancing

- Less network overhead
  - 8-byte header instead of TCP's 20-byte header

UDP multiplexing & demultiplexing

- A UDP socket is identified by its port number

- All UDP segments addressed to a specific port # will be delivered to the socket identified by that port number
  - A socket will request data via `recv()`, `recvfrom()`, or `recvmsg()` system calls
  - OS looks for a UDP socket with a matching destination port: hash table of socket structures; hash key created from UDP destination port

- Limited demultiplexing
  - Segments addressed to the same (host, port) from different processes or different systems will be delivered to the same socket!
  - The receiver can get the source address & port to know how to address reply messages

UDP Structure

- Defined in RFC 768

- Eight byte header

<table>
<thead>
<tr>
<th>Source Port #</th>
<th>Dest Port #</th>
<th>Length</th>
<th>Checksum</th>
<th>Application Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 bytes</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td>2 bytes</td>
<td></td>
</tr>
</tbody>
</table>
UDP Structure in context

Eight byte header within a 20 byte IP header

- Source Port #
- Dest Port #
- Length
- Checksum
- Application Data

UDP Checksum

- IP does not guarantee error-free packet delivery
- The UDP header contains a 16-bit checksum
  - Checks for data corruption
- Checksum is generated by the sender and validated only by the receiver only

IP header fields are used to protect against misrouted segments

Pseudo header for checksum computation

- Source IP Address
- Destination IP Address
- Source Port
- Destination Port
- Length
- Checksum
- Application Data

UDP Checksum Calculation

- Sender
  - Iterate over 16-bit words in the Pseudo header + UDP segment
  - UDP checksum field = 0
  - Create a 1s complement sum
  - Invert the bits

- Receiver
  - Perform the same 1s complement sum on all data including the checksum field
  - The result should be all 1s (0xFFFF)

TCP multiplexing & demultiplexing

- Every TCP socket is identified by:
  (source address, destination address, source port, destination port)

- A TCP socket has a state:
  - LISTEN: the socket is used only for accepting connections
  - ESTABLISHED: the socket is connected
  - Other states that we’ll ignore for now:
    - Connection setup:
      - SYN_SENT: trying to establish a connection
      - SYN_RCVD: received a connection request
    - Connection tear down:
      - FIN_WAIT_1: socket has been closed by the local application; no acknowledgement from remote
      - FIN_WAIT_2: socket has been closed by the local application; remote acknowledged closing
      - CLOSING: socket has been closed by the local & remote apps; remote has not acknowledged close
      - TIME_WAIT: connections closed; waiting to be sure that the remote side received the last ACK

- Let’s look at an example

Server: Create a new socket

\[ \text{svr} = \text{socket}(AF\_INET,\ SOCK\_STREAM, 0); \]

Create a new socket at the server: it has no addresses so far

Server: Bind – assign a local address

\[ \text{bind}(\text{svr}) ; \]

Assign a local address (INADDR_ANY) and port (1234) to the socket

Server (192.11.5.8)

Client (10.0.10.1)

Local Addr | Local Port | Remote Addr | Rem Port | State
-----------|------------|-------------|----------|--------
0.0.0.0     | 1234       |             |          |        

Server (10.1.1.5)

Client (10.0.10.1)

Local Addr | Local Port | Remote Addr | Rem Port | State
-----------|------------|-------------|----------|--------
0.0.0.0     | 1234       |             |          |        

N.B.: We refer to a socket table here for convenience but it is just a logical construct. This actual implementation is operating system-specific but the data is generally shared in a list of socket buffer structures (e.g., for example, the kernel function tcp_v4_lookup will search for either a listening or an established socket with specific addresses and ports (see tcp_v4_lookup, port a, around line 519).
Server: Make it a listening socket

```
listen(svr, 10);
```

Set the state of the socket to `listen`. This socket can only be used to accept connections.

<table>
<thead>
<tr>
<th>Local Addr</th>
<th>Local Port</th>
<th>Remote Addr</th>
<th>Rem Port</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0</td>
<td>1234</td>
<td></td>
<td></td>
<td>LISTEN</td>
</tr>
</tbody>
</table>

Server (192.11.5.8)

Client (135.10.10.1)

Server: Wait for a connection

```
snew = accept(svr);
```

Wait for an incoming connection on this socket.

<table>
<thead>
<tr>
<th>Local Addr</th>
<th>Local Port</th>
<th>Remote Addr</th>
<th>Rem Port</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0</td>
<td>1234</td>
<td></td>
<td></td>
<td>LISTEN</td>
</tr>
</tbody>
</table>

Server (192.11.5.8)

Client (135.10.10.1)

Client: Create a new socket

```
s = socket();
```

Create a new socket at the client: no addresses so far.

<table>
<thead>
<tr>
<th>Local Addr</th>
<th>Local Port</th>
<th>Remote Addr</th>
<th>Rem Port</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7801</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Server (192.11.5.8)

Client (135.10.10.1)

Client: Assign a local address & port #

```
bind(s);
```

Assign any local address (INADDR_ANY) and have the OS pick a port (port=0).

<table>
<thead>
<tr>
<th>Local Addr</th>
<th>Local Port</th>
<th>Remote Addr</th>
<th>Rem Port</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0</td>
<td>7801</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Server (192.11.5.8)

Client (135.10.10.1)

Client: Connect to the server

```
connect(s, dest_addr);
```

Connect to address 192.11.5.8, port 1234

<table>
<thead>
<tr>
<th>Local Addr</th>
<th>Local Port</th>
<th>Remote Addr</th>
<th>Rem Port</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.11.5.8</td>
<td>1234</td>
<td>135.10.10.1</td>
<td>7801</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>

Server (192.11.5.8)

Client (135.10.10.1)

Client: Complete the connection

```
connect(s, dest_addr);
```

Server acknowledges the connection; Client fills in the entry.

<table>
<thead>
<tr>
<th>Local Addr</th>
<th>Local Port</th>
<th>Remote Addr</th>
<th>Rem Port</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.11.5.8</td>
<td>192.11.5.8</td>
<td>1234</td>
<td>7801</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>

Server (192.11.5.8)

Client (135.10.10.1)

Now we can talk!
Communicate

Client-to-server communication
- Server finds socket by searching for a TCP socket with these properties:
  1. Status == ESTABLISHED
  2. IP src addr == remote addr
  3. TCP src port == remote port
  4. IP dest addr == local addr
  5. TCP dest port == local port

<table>
<thead>
<tr>
<th>Local Addr</th>
<th>Local Port</th>
<th>Remote Addr</th>
<th>Rem Port</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0</td>
<td>7801</td>
<td>192.11.5.8</td>
<td>1234</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>

Server (192.11.5.8)

Client (135.10.10.1)

Different source address disambiguates the sessions

Server (192.11.5.8)

Client (135.10.10.1)

Two clients sharing the same port

Server (192.11.5.8)

Client (135.10.10.1)

Two endpoints sharing the same address

Server (192.11.5.8)

Client (135.10.10.1)

Reliable Data Transfer (RDT) Goal
- Develop a protocol for transmitting data reliably over an unreliable network

Reliable Data Transfer
RDT over a reliable channel

- Assume the channel is reliable
- Trivial – nothing to do!

Here’s the finite state machine (FSM):

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt_send(data)</td>
<td>packet = make_pkt(data) udt_send(packet)</td>
</tr>
<tr>
<td>rdt_rcv(packet)</td>
<td>extract(packet, data) deliver(data)</td>
</tr>
</tbody>
</table>

RDT over a channel with bit errors

- All packets are received
- Some might be corrupt

- Approach
  - Acknowledge each packet
    - Positive acknowledgement (ACK): “I got it; looks good!”
    - Negative acknowledgement (NAK): “Please repeat”
  - Sender retransmits a packet if it receives a NAK

  - ARQ (Automatic Repeat reQuest)
    - Set of protocols that use acknowledgements & retransmission

We need to support three capabilities

1. Error detection
   - How do we know if the packet is corrupt?
   - Use a checksum (error detecting code)

2. Receiver feedback
   - The receiver will acknowledge each packet with an ACK or NAK

3. Retransmission
   - If a sender gets a NAK, the packet will be retransmitted

Stop-and-wait

- The sender cannot send any data until it receives an ACK for the previously sent packet
- This type of protocol is a stop-and-wait protocol

What about a corrupted ACK/NAK message?

- The sender does not know whether the last packet was received correctly or not
- We can
  - Have the sender send a “please repeat” in response to a corrupt ACK/NAK
  - But what if that gets corrupted?
  - Add a robust error correcting code
    - Works for a channel that does not lose data
  - Resend the data in response to a corrupted ACK/NAK
  - Duplicate packets may be received
  - Receiver needs to distinguish between new data & a retransmission
  - Use a sequence number. Here, we only need a 1-bit number.
A 1-bit sequence number

Sequence bit flip-flops between consecutive messages

Alternating bit protocol

RDT over a channel with bit errors

Sender

Wait for packet

Received seq=0
Delivered to app; Send ACK

udt_send(sndpkt)

sndpkt = make(ACK, checksum)

deliver(data)

extract(rcvpkt, data)

&& has_seq0(rcvpkt)

rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)

Deliver to app; Send ACK #0

Received seq=0

Send ACK with previous # (0)

udt_send(sndpkt)

sndpkt = make(ACK, 0, checksum)

deliver(data)

extract(rcvpkt, data)

&& has_seq1(rcvpkt)

rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)

Deliver to app; Send ACK #1

Received seq=1

Send ACK with previous # (1)

udt_send(sndpkt)

sndpkt = make(ACK, 1, checksum)

deliver(data)

extract(rcvpkt, data)

&& has_seq0(rcvpkt)

rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)

Deliver to app; Send ACK #0

Received seq=0

Send ACK with previous # (0)

udt_send(sndpkt)

sndpkt = make(ACK, 0, checksum)

deliver(data)

extract(rcvpkt, data)

&& has_seq1(rcvpkt)

rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)

Deliver to app; Send ACK #1

Received seq=1

Send ACK with previous # (1)

udt_send(sndpkt)

sndpkt = make(ACK, 1, checksum)

deliver(data)

extract(rcvpkt, data)

&& has_seq1(rcvpkt)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

Send NAK

We can get rid of NAKs

Report if corrupt or wrong

udt_send(sendpkt)

packet = make_pkt(seq=0, data, checksum)

if corrupt

else

udt_send(sendpkt)

packet = make_pkt(seq=1, data, checksum)

if corrupt

else

RDT over a channel with bit errors – no NAK

Sender

Wait for packet

Received seq=0
Delivered to app; Send ACK

udt_send(sndpkt)

sndpkt = make(ACK, checksum)

deliver(data)

extract(rcvpkt, data)

&& has_seq0(rcvpkt)

rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)

Deliver to app; Send ACK #0

Received seq=0

Send ACK with previous # (0)

udt_send(sndpkt)

sndpkt = make(ACK, 0, checksum)

deliver(data)

extract(rcvpkt, data)

&& has_seq1(rcvpkt)

rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)

Deliver to app; Send ACK #1

Received seq=1

Send ACK with previous # (1)

udt_send(sndpkt)

sndpkt = make(ACK, 1, checksum)

deliver(data)

extract(rcvpkt, data)

&& has_seq1(rcvpkt)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

Send NAK

We can get rid of NAKs

Report if corrupt or wrong

udt_send(sendpkt)

packet = make_pkt(seq=0, data, checksum)

if corrupt

else

udt_send(sendpkt)

packet = make_pkt(seq=1, data, checksum)

if corrupt

else
RDT over a lossy channel

- We considered only bit errors
  - Packets were always delivered
- How do we detect & deal with packet loss?

Dealing with packet loss

- Burden of detection & recovery is on sender
  - If sender’s packet is lost OR receiver’s ACK is lost
    - Sender will not get a reply from the receiver

- Approach
  - Introduce a countdown timer. Set the timer at transmit
  - If time-out and no reply, retransmit
  - How long to wait? Maximum round-trip delay?
    - Long wait until we initiate error recovery
    - Pick a ‘likely loss’ time
    - Retransmit if no response within that time
    - Introduces possibility of duplicate packets
      - But we already know how to deal with them

RDT over lossy channel – with a timer

RDT – Alternating Bit Protocol: no loss

RDT – Alternating Bit Protocol: lost Packet

RDT – Alternating Bit Protocol: lost ACK
RDT – Alternating Bit Protocol: early timeout

Receiver
Send P0
Receive P0
Send ACK0
Receive ACK0
Send P1
Receive P1
[not duplicate]
Send ACK1
Receive P0
Send ACK0
Receive P1
[detected duplicate]
Send ACK1
Receive ACK1
Discard

Network utilization with stop-and-wait

- **A stop-and-wait protocol gives us horrible network utilization**
- Consider
  - Cross-country link ⇒ Round trip propagation delay (RTT) ≈ 30 ms
  - Assume 1 Gbps link (ignore router delays), R = 10^9 bits/second
  - Assume 1,000-byte packets (L = 8,000 bits)
  - Time to transmit the packet: \( t_{trans} = L / R \approx 8 \mu s \)
- With a stop-and-wait protocol
  - one-way delay = \( d_{trans} + d_{prop} \approx 30 \text{ ms} \div 2 + 8 \mu s = 15.008 \text{ ms} \)
  - Assume ACK packets are tiny; one-way delay for ACK packet = 15 ms
  - Next packet can be sent (15.008 + 15) = 30.008 ms after the first one
  - Utilization = fraction of time sender is sending bits into the channel
    \[ U = \frac{L}{RTT + \left( \frac{L}{R} \right)} = 0.0008 \]
    \[ = 0.00027 = 0.027\% \]
    The sender can transmit 1,000 bytes in 30.008 ms: 267 kbps on a 1 Gbps link!

Improve Network Utilization: Pipelining

- Don’t wait for an acknowledgement before sending the next packet
- But then we need to
  1. Increase the range of sequence numbers
     - Each in-transit packet needs a unique number
  2. Hold on to unacknowledged packets at sender
  3. Hold on to out-of-sequence packets at receiver
- Two approaches for pipelined error recovery
  - Go-Back-N
  - Selective Repeat

Go-Back-N (GBN)

- **Sender can send multiple packets without waiting for ACKs**
- **No more than N unacknowledged packets**

\( U = \frac{L}{RTT + \left( \frac{L}{R} \right)} = 0.0008 \)
\[ = 0.00027 = 0.027\% \]

Go-Back-N (GBN)

- **Sender can send multiple packets without waiting for ACKs**
- **No more than N unacknowledged packets**

GBN = Sliding Window Protocol
**Go-Back-N (GBN)**

- Sender can send multiple packets without waiting for ACKs.
- No more than $N$ unacknowledged packets.

The window slides as packets are acknowledged.

```
received ACK  |  sent, no ACK  |  not yet sent
```

- Receiver with sequence # $\leq$ base + $N$.
- Receiver can only have $N$ unacknowledged packets.

**Sequence numbers**

A sequence number will take up a fixed $k$, of bits in the header:

- Range of sequence numbers is $0 \ldots 2^k - 1$.
- Modulo $2^k$ arithmetic: $2^k - 1$ increments to 0.

**Extended FSM for a GBN sender**

```c
Sender

wait

// Send data if it's in the window (we can have at most N unacknowledged packets)
if (next_seqnum < base+N) {
  // there's room in the window
  sndpkt[next_seqnum] = make_pkt(next_seqnum, data, checksum)
  udt_send(sndpkt[next_seqnum])
  if (base == next_seqnum)
    start_timer
  next_seqnum++
} else {
  refuse_data(data) // cannot send
}
```

**Selective Repeat**

- Problem with Go-Back-N:
  - With a large window size and large delays, many packets can be in the pipeline.
  - A single error can cause GBN to retransmit many packets (all that are unacknowledged).
  - If $P$ (channel error) increases, the pipeline can become filled with excess retransmissions.

- **Selective Repeat Protocol**:
  - Retransmit only those packets that were lost or corrupted.
  - Receiver must acknowledge each correctly received packet.
  - Even if it is out of order.
  - Out of order packets must be buffered.
  - Window size = Limit of number of outstanding packets.
  - Some packets in the window may be acknowledged.
  - The window slides when the earliest packet in the window is acknowledged.
Selective Repeat Windows

- Sender's view of sequence numbers
  - send_base
  - next_seqnum

- Receiver's view
  - rcv_base
  - window size N

Selective Repeat: sender operation

- Send requests from application
  - Check next available sequence #
  - If no room in window, reject (or buffer)
  - Else send the packet (with sequence #)

- Timeout
  - Each packet has its own timer
  - Retransmit only the specific packet on timeout

- ACK received
  - If packet is within window
    - Mark packet as received
    - If sequence # == send_base
      - advance the base (start of window) to the next unacknowledged sequence number

Selective Repeat: receiver operation

- Good packet with seq # in [rcv_base, rcv_base+N-1]
  - Packet is within the receiver’s window
  - Send ACK for that sequence #
  - If sequence # == rcv_base
    - Deliver packet to app and deliver all successive packets that have been received
    - Adjust start of window (rcv_base)

- Good packet with seq # in [rcv_base-N, rcv_base-1]
  - Packet is within the before receiver’s window
  - We already saw it — but send ACK anyway

- Anything else
  - Ignore the packet

Out-of-order ACKs

- Sender's view of sequence numbers
  - send_base
  - next_seqnum

- Receiver's view
  - window size N

Selective Repeat: receiving packets

- Receiver's view
  - rcv_base
  - window size N

- Out of order ACKs
  - Expected
  - Acceptable
  - Unusable

- When an ACK for this packet is received, send base is advanced to the next packet with no ACK

- This ACK for this packet was received, so sender must advance the base (start of window) to the next unacknowledged sequence number
Selective Repeat: receiving packets

Receiver's view
\[ \text{rcv_base} \]
deliver to app

This packet was received, so we delivered it and all received packets immediately after it.

The start of the window (base) is moved to the first missing packet. The receiver displays the window as positions 1 to \( N \) and delays the same as the start of the window on the sender.

The end

Out of order, ACKed
Expected
Acceptable
Usable