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

• Need to identify which segment belongs to which channel

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

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

UDP Structure

32 bits
8 bits
16 bits
8 bits
Source Port
Dest Port
Length
Checksum
Application Data
UDP Structure in context

Eight byte header within a 20 byte IP header

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

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 teardown:
      - 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 the closing
      - CLOSING: socket has been closed by the local & remote apps; remote has not acknowledged close
      - TIME_WAIT: connection closed; waiting to be sure that the remote side received the last ACK
- Let’s look at an example

Server: Create a new socket

svr = 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

bind (svr);

Assign a local address (INADDR_ANY) and port (1234) to the socket
Server: Make it a listening socket

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

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

Server: Wait for a connection

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

Wait for an incoming connection on this socket.

Client: Create a new socket

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

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

Client: Assign a local address & port #

```c
bind(s);
```

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

Client: Connect to the server

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

Connect to address 192.11.5.8, port 1234.

Client: Complete the connection

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

Server acknowledges the connection; Client fills in the entry.
Communicate

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

<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>0.0.0.0</td>
<td>7801</td>
<td>LISTEN</td>
</tr>
</tbody>
</table>

   Server (192.11.5.8)

   Client (135.10.10.1)

Server-to-client communication
1. Client finds socket by searching for a TCP socket with these properties:
   - Port == ESTABLISHED
   - IP src addr == remote addr
   - TCP src port == remote port
   - IP dest addr == local addr
   - 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>

<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>0.0.0.0</td>
<td>7801</td>
<td>LISTEN</td>
</tr>
</tbody>
</table>

   Server (192.11.5.8)

   Client (135.10.10.1)

Two clients sharing the same port

Different source address disambiguates the sessions

<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>

<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>0.0.0.0</td>
<td>7801</td>
<td>LISTEN</td>
</tr>
</tbody>
</table>

   Server (192.11.5.8)

   Client (135.10.10.1)

   Client (135.10.10.2)

Two endpoints sharing the same address

The OS will not allow two sockets to share the same port on one client

<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>

<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>0.0.0.0</td>
<td>7801</td>
<td>LISTEN</td>
</tr>
</tbody>
</table>

   Server (192.11.5.8)

   Client (135.10.10.1)

   Client (135.10.10.2)

Reliable Data Transfer

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

RDTSend
    RDTRcv
data is received by the host
data is delivered to the app

sending implementation
receiving implementation
unreliable channel
Reliable data transfer protocol
Reliable data transfer protocol

Reliable Data Transfer
**RDT over a reliable channel**

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

Here’s the finite state machine (FSM):

![Finite State Machine Diagram]

**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**

- **Error detection**
  - How do we know if the packet is corrupt?
    - Use a checksum
- **Receiver feedback**
  - The receiver will acknowledge each packet with an ACK or NAK
- **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 more 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 bits flip-flop between consecutive messages

Alternating bit protocol

RDT over a channel with bit errors

• If a corrupted packet is received
  – Send a NAK

• If a duplicate packet is received
  – Send an ACK since we already processed the packet

• We can get rid of NAKs
  – Send an ACK for the last correctly received packet
  – If a sender receives duplicate ACKs, if knows that the previous packet has not been received correctly
  – Modify protocol: add sequence numbers to ACKs

RDT over a channel with bit errors – no NAK

Like the previous FSM but if NAK is replaced with an uACK check for the wrong ACK #.
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 a 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
  - Receive P0
  - Sending ACK0
  - Send P1
  - Resend P1
  - Receive ACK1
  - Receive P0
  - Send ACK0
  - Send P1
  - Receive ACK1
  - Receive P0
  - Send ACK0
  - Receive P1 (detect duplicate)
  - Send ACK1
  - Receive ACK1
  - Discard it

- Sender
  - Send P0
  - Receive P0
  - Send P1

**Network utilization with stop-and-wait**

- **A stop-and-wait protocol gives us horrible network utilization**
- **Consider**
  - Cross-country 1 Gbps link, no routers
  - Round trip propagation delay (RTT) ≈ 30 ms
  - Assume 1,000-byte packets (8,000 bits)
  - Time to transmit the packet = $L / R = 8,000 / 10^9 = 8 \mu s$
- **With a stop-and-wait protocol**
  - one-way delay = $d_{trans} + d_{prop} = (30-2) ms + 8 \mu s = 15.008 ms$
  - 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

\[
U = \frac{L}{RTT + \left(\frac{L}{R}\right)} = \frac{8,000}{30.008} \approx 0.00027 = 0.027\%
\]

The sender can transmit 1,000 bytes in 30.008 ms: 267 kbps on a 1 Gbps link!

**Pipelining**

- **Don’t wait for an acknowledgement before sending the next packet**
- **We need to**
  - Increase the range of sequence numbers
  - Hold on to unacknowledged packets at sender
  - 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**

**Go-Back-N (GBN)**

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

**Go-Back-N (GBN)**

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

$GBN = \text{Sliding Window Protocol}$
**Go-Back-N (GBN)**

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

\[ \text{GBN} = \text{Sliding Window Protocol} \]

**Sequence numbers**

- A sequence number will take up a fixed #, $k$, of bits in the header
  - Range of sequence numbers is $0 .. 2^k - 1$
  - Modulo $2^k$ arithmetic: $2^k - 1$ increments to 0

**Extended FSM for a GBN sender**

**Selective Repeat**

- 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
  - 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 $N = \text{limit of number of outstanding, unacknowledged packets}$
  - But 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`: window size `N`
- `next_seqnum`

**Receiver’s view**

- `rcv_base`: window size `N`

---

**Selective Repeat: sender operation**

- **Send requests from app**
  - Check next available sequence #
  - If no room in window, reject (or buffer)
  - Else send the packet
- **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 # == base, advance the base (start of window) to the next unacknowledged sequence number

---

Out-of-order ACKs

**Sender’s view of sequence numbers**

- `send_base`: window size `N`
- `next_seqnum`

---

**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]**
  - Before window – we already saw this – send acknowledgement
- **Anything else**
  - Ignore the packet

---

**Selective Repeat: receiving packets**

**Receiver’s view**

- `rcv_base`: window size `N`

---

Out-of-order ACKs

**Sender’s view of sequence numbers**

- `send_base`: window size `N`
- `next_seqnum`

---
Selective Repeat: receiving packets

Receiver's view

deliver to app

rcv_base

window size $N$

Out of order

ACKed

Expected

Acceptable

Unusable

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 expected base is not always the same as the start of the window on the sender.

The end