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 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

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

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 Structure in context

Eight byte header within a 20 byte IP header

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td></td>
</tr>
<tr>
<td>Dest Port</td>
<td></td>
</tr>
<tr>
<td>Length</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td></td>
</tr>
<tr>
<td>Application Data</td>
<td></td>
</tr>
</tbody>
</table>

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 segments with bad checksums are simply dropped

UDP Checksum Calculation

Sender
- Iterate over 16-bit words in the Pseudo header + UDP segment
- UDP checksum field = 0
- Create a one’s complement checksum
  - Add two 16-bit values. If overflow, add 1 to the result
  - Invert the bits of the result to get the checksum value

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

The same checksum calculation is used for the IP header, UDP header, & TCP header

One’s Complement Checksum Example

- How to compute a One’s complement
  - Sum the numbers
    - Add any overflow carry to the result
  - Create checksum for:
    - Add the checksum

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, SYN/ACK: trying to establish a connection
    - SYN/ACK/ACK: received an acknowledge request
    - Connection teardown:
      - FIN, FIN/ACK: socket has been closed by the local application
      - LASTACK: socket has been closed by the local application
      - Connect closed because the local & remote app. failed to connect
      - 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

```c
svr = socket(AF_INET, SOCK_STREAM, 0);
```

Create a new socket at the server: it has 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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Server: Bind – assign a local address

```c
bind(svr);
```

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

<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: Wait for a connection

```c
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>

### Client: Create a new socket

```c
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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Client: Assign a local address & port #

```c
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>1234</td>
<td></td>
<td></td>
<td></td>
<td>LISTEN</td>
</tr>
</tbody>
</table>
On the server, search the table for a LISTEN socket where Status == ESTABLISHED
IP src addr  == remote addr
TCP src port = remote port
TCP dest port == local port
Server finds socket by searching for a TCP socket with these properties:
- IP dest addr == local addr
Client finds socket by searching for a TCP socket with these properties:
TCP src port = remote port
IP
Create a new socket for the connection
TCP dest port == local port
IP dest addr == local addr
Status == ESTABLISHED
Send a connection establishment request to address 192.11.5.8, port 1234

Different source address disambiguates the sessions:

Two endpoints sharing the same address:
The OS will not allow two sockets to share the same port on one client

Paul Krzyzanowski
Reliable Data Transfer (RDT) Goal

Develop a protocol for transmitting data reliably over an unreliable network.

Reliable Data Transfer

Goal

Dev elop a protocol for transmitting data reliably over an unreliable network.

RDT over a reliable channel

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

Here’s the finite state machine (FSM):

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

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**

- seq=0: data 0
- seq=1: data 1
- seq=0: data 2
- seq=1: data 3

### 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, it knows that the previous packet has not been received correctly.
- Modify protocol: add sequence numbers to ACKs.

---

**Diagram of RDT over a channel with bit errors**

- Diagram shows the sequence of events for receiving and transmitting packets, including handling of corrupted and duplicate packets using sequence numbers.

---

**Diagram of alternating bit protocol**

- The sequence bit flips between 0 and 1 for consecutive messages, illustrating how sequence numbers can be used to distinguish between new data and retransmissions.
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

sender

[Diagram of RDT over a channel with bit errors – no NAK]

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 – Alternating Bit Protocol: lost Packet

<table>
<thead>
<tr>
<th>Time</th>
<th>Receiver</th>
<th>Sender</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>send</td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>receive</td>
<td>P0</td>
</tr>
<tr>
<td>ACK0</td>
<td>send</td>
<td>ACK0</td>
</tr>
<tr>
<td>P1</td>
<td>send</td>
<td></td>
</tr>
<tr>
<td>timeout</td>
<td>-</td>
<td>no ACK</td>
</tr>
<tr>
<td>P1</td>
<td>resend</td>
<td>P1</td>
</tr>
<tr>
<td>ACK1</td>
<td>receive</td>
<td>ACK1</td>
</tr>
<tr>
<td>P0</td>
<td>send</td>
<td>P0</td>
</tr>
<tr>
<td>P1</td>
<td>receive</td>
<td>P1</td>
</tr>
<tr>
<td>ACK1</td>
<td>send</td>
<td>ACK1</td>
</tr>
<tr>
<td>P0</td>
<td>send</td>
<td>ACK0</td>
</tr>
<tr>
<td>P1</td>
<td>receive</td>
<td>P1</td>
</tr>
<tr>
<td>ACK1</td>
<td>send</td>
<td>ACK1</td>
</tr>
</tbody>
</table>

### RDT – Alternating Bit Protocol: lost ACK

<table>
<thead>
<tr>
<th>Time</th>
<th>Receiver</th>
<th>Sender</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>send</td>
<td></td>
</tr>
<tr>
<td>P0</td>
<td>receive</td>
<td>P0</td>
</tr>
<tr>
<td>ACK0</td>
<td>send</td>
<td>ACK0</td>
</tr>
<tr>
<td>P1</td>
<td>send</td>
<td></td>
</tr>
<tr>
<td>timeout</td>
<td>-</td>
<td>no ACK</td>
</tr>
<tr>
<td>P1</td>
<td>resend</td>
<td>P1</td>
</tr>
<tr>
<td>ACK1</td>
<td>receive</td>
<td>ACK1</td>
</tr>
<tr>
<td>P0</td>
<td>send</td>
<td>P0</td>
</tr>
<tr>
<td>P1</td>
<td>receive</td>
<td>P1</td>
</tr>
<tr>
<td>ACK1</td>
<td>receive</td>
<td>P1</td>
</tr>
<tr>
<td>P0</td>
<td>send</td>
<td>P0</td>
</tr>
<tr>
<td>P1</td>
<td>receive</td>
<td>P1</td>
</tr>
<tr>
<td>ACK1</td>
<td>send</td>
<td>ACK1</td>
</tr>
<tr>
<td>P0</td>
<td>send</td>
<td>ACK0</td>
</tr>
<tr>
<td>P1</td>
<td>receive</td>
<td>P1</td>
</tr>
<tr>
<td>ACK1</td>
<td>send</td>
<td>ACK1</td>
</tr>
</tbody>
</table>

### 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), \( R = 10^9 \text{ bits/second} \)
- Assume 1,000-byte packets (\( L = 8,000 \text{ bits} \))
- Time to transmit the packet: \( t_{trans} = \frac{L}{R} = \frac{8,000}{10^9} = 8 \mu s \)

With a stop-and-wait protocol:
- one-way delay = \( d_{trans} = d_{prop} = (30 \text{ ms} ÷ 2) + 8 \mu s = 15.008 \text{ ms} \)
- Assume ACK packets are tiny; one-way delay for ACK packet = 15 ms
- ACK is received at 15.008 + 15 = 30.008 ms after the first one
- Utilization = fraction of time sender is sending bits into the channel

\[
U = \frac{L}{RTT} + \left( \frac{L}{RTT} \right)^2 - 0.0008 \left( \frac{L}{RTT} \right) + 0.00027 = 0.027\%
\]

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
  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

- window size \( N \)

- received ACK
- sent, no ACK
- not yet sent

New packets can be sent

Packets with sequence \( \geq base + N \) cannot be sent.
Go-Back-N (GBN)

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

Extended FSM for a GBN sender

- $k$ bits in a window
- Out of order packets
- Retransmitted
- All sequence numbers known
- All packets are delivered
- No out-of-order delivery
- All packets acknowledged
- No data loss
- No timeouts

### Cumulative acknowledgment

Receipt of a sequence number $n$ ACK means that all packets up to and including $n$ have been received
**Extended FSM for a GBN receiver**

<table>
<thead>
<tr>
<th>Receiver</th>
</tr>
</thead>
</table>
| We received a good packet if the expected sequence number matches, and this packet is in sequence order. The receiver checks:  
| packet - send_base   |
| expected_seqnum      |
| !corrupt(rcvpkt)      |
| has_seqnum(rcvpkt, expected_seqnum) |
| extract(rcvpkt, data) |
| deliver(data)         |
| // give it to the app |
| sndpkt = makepkt(expected_seqnum, ACK, checksum) |
| udt_send(sndpkt)      |
| // send the ACK to the sender |
| expected_seqnum       |
| += 1                  |

The receiver discards out-of-order packets. If packet n is lost and n+1 arrives, the receiver does not buffer packet n+1. The sender will retransmit all unacknowledged packets (go back N).

The receiver has to only keep track of the next sequence number.

---

**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 N = limit of number of outstanding packets
    - But some packets in the window may be acknowledged
  - The window slides when the earliest packet in the window is acknowledged

---

**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 Windows**

- **Sender's view of sequence numbers**
  - window size N
  - send_base
  - next_seqnum

- **Receiver's view**
  - window size N
  - rcv_base
  - next_seqnum

---

**Out-of-order ACKs**

- **Sender's view of sequence numbers**
  - window size N
  - send_base
  - next_seqnum

- **Receiver's view**
  - window size N

---

**Out-of-order ACKs**

- **Sender's view of sequence numbers**
  - window size N
  - send_base
  - next_seqnum

- **Receiver's view**
  - window size N
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