Internet Technology

05. Transport Layer

Paul Krzyzanowski
Rutgers University
Spring 2016
• 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
Transport Layer

• Network Layer
  – Logical connection between hosts
• Transport Layer
  – Logical connection between processes
  – Transport layer multiplexing & demultiplexing
• Most common transport-layer protocols in IP: TCP & UDP
  – UDP: unreliable data transfer
  – TCP
    • Reliable data transfer
    • In-order delivery
    • Flow control
    • Congestion control
Today, we’ll discuss

• Transport layer multiplexing/demultiplexing

• Reliable data transfer
Transport Layer
Multiplexing & Demultiplexing
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

Logical view of four transport layer communication streams
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

![Logical view at the network layer](image)
Multiplexing & Demultiplexing

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
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

• Defined in RFC 768
• Eight byte header
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: segments with bad checksums are simply dropped

---

**UDP header**

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port #</td>
<td>16 bits</td>
</tr>
<tr>
<td>Dest Port #</td>
<td>16 bits</td>
</tr>
<tr>
<td>Length</td>
<td>16 bits</td>
</tr>
<tr>
<td>Checksum</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

**IP pseudo header** used for checksum computation

- Set to 0 to compute initial checksum
- Padded with 0 at the end to get to a 16 bit boundary (if needed) [not transmitted; just for checksum]

**IP header fields are used to protect against misrouted segments**
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
    • Do this for all the data you need to checksum
    • 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 (0xFFFF)

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:

  \[
  \begin{array}{cccccccc}
  0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
  + & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1
  \end{array}
  \]

  \[
  = 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0
  \]

\[
\begin{array}{cccccccc}
  1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0
  + & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1
  \end{array}
  \]

\[
= 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0
\]

• Then invert the bits

  \[
  \sim \begin{array}{cccccccc}
  1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0
  \end{array}
  \]

  \[
  = 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1
  \]
One’s Complement Checksum Example

• Validate
  – Sum the numbers, including the checksum

  0 1 1 0 1 0 1 1 0 0 0 0 1 0 1 0
  1 0 1 1 0 0 0 1 1 1 0 0 1 0 0 0
  1 1 0 0 0 0 0 0 1 1 1 1 0 1 0 1

  add the checksum → 0 0 1 0 0 0 1 0 0 1 1 0 1 1

  0 1 1 0 1 0 1 1 0 0 0 0 1 0 1 0
  + 1 0 1 1 0 0 0 1 1 1 0 0 1 0 0 0
  = 1 0 0 0 1 1 1 0 0 1 1 0 1 0 0 1 0
  + 1
  = 0 0 0 1 1 1 0 0 1 1 1 1 0 0 1 1 0 1 0
  + 1 1 0 0 0 0 0 0 1 1 1 1 0 1 0 1 0
  = 1 1 0 1 1 1 0 1 1 1 0 0 0 1 0 0 0 0
  + 0 0 1 0 0 0 1 0 0 1 1 0 1 1 0 1 1 1
  = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

  • A result of all 1’s (= -0) means the transmission was good
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: 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

svr = socket(AF_INET, SOCK_STREAM, 0);

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

Client (135.10.10.1)

<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 (192.11.5.8)

<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>

Address family: Internet (IPv4)
Type: “stream” – connection-oriented (TCP)

N.B.: We refer to a socket table here for convenience but it is just a logical construct. The actual implementation is operating-system specific but this data is generally stored in a list of socket buffer structures. On Linux, for example, the kernel function `tcp_v4_lookup` will search for either a listening or an established socket with specific addresses and ports (see `net/ipv4/tcp_ipv4.c`, around line 507)
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Server (192.11.5.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Addr</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>0.0.0.0</td>
</tr>
</tbody>
</table>

Client (135.10.10.1)
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Server (192.11.5.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Addr</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>0.0.0.0</td>
</tr>
</tbody>
</table>

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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Client (135.10.10.1)

Server (192.11.5.8)

<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

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

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

<table>
<thead>
<tr>
<th>Client (135.10.10.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Local Addr</strong></td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**s**

<table>
<thead>
<tr>
<th>Server (192.11.5.8)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Local Addr</strong></td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>0.0.0.0</td>
</tr>
</tbody>
</table>
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>s</td>
</tr>
</tbody>
</table>

Client (135.10.10.1)

<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: Connect to the server

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

Connect to address 192.11.5.8, port 1234

- Send a connection establishment request to address 192.11.5.8, port 1234 (TCP segment to port 1234 with a connection setup bit set; we’ll look at the exact handshake later)
- On the server, search the table for a LISTEN socket where
  packet’s destination addr == table’s local addr (0.0.0.0 matches any incoming addr)
  packet’s destination port == table’s local port
- Create a new socket for the connection
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>0.0.0.0</td>
<td>7801</td>
<td>192.11.5.8</td>
<td>1234</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>

Connect(s, dest_addr);

Client (135.10.10.1)

Store (svr) and send (snew).

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

Now we can talk!
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

Client (135.10.10.1)

<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)

<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>
<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-to-client communication

- Client 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

### Client (135.10.10.1)

<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)

<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>
<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>
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>

Client (135.10.10.2)

<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>

Client (135.10.10.1)

<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>135.10.10.1</td>
<td>7801</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>

Server (192.11.5.8)

<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>192.11.5.8</td>
<td>7801</td>
<td>LISTEN</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>135.10.10.1</td>
<td>7801</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>135.10.10.1</td>
<td>7801</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>
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>
<tr>
<td>0.0.0.0</td>
<td>7802</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>0.0.0.0</td>
<td>1234</td>
<td></td>
<td></td>
<td>LISTEN</td>
</tr>
<tr>
<td>192.11.5.8</td>
<td>1234</td>
<td>135.10.10.1</td>
<td>7801</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>192.11.5.8</td>
<td>1234</td>
<td>135.10.10.1</td>
<td>7802</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>
Reliable Data Transfer
Reliable Data Transfer (RDT) Goal

Develop a protocol for transmitting data reliably over an unreliable network

sending application

<table>
<thead>
<tr>
<th>rdt_send</th>
</tr>
</thead>
</table>

Reliable data transfer protocol

udt_send

unreliable channel

rdt_rcv

receiving application

deliver

data is received by the host

data is delivered to the app
RDT over a reliable channel

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

Here's the finite state machine (FSM):

**Sender**
- Wait for send
- rdt_send(data)
  - packet = make_pkt(data)
  - udt_send(packet)

**Receiver**
- Wait for receive
- rdt_rcv(packet)
  - extract(packet, data)
  - deliver(data)
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 re**Quest)
    • 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

Sender

- **Start**
- **Wait for send**
- **Wait for ACK/NAK**
- **Transmit a packet**
- **The receiver got a good packet**
- **Retransmit if NAK**

### Code
- `rdt_send(data)`
- `packet = make_pkt(data, checksum)`
- `udt_send(data)`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sendpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`
- `sndpkt = make_pkt(NAK)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && not_corrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver(data)`
- `sndpkt = make_pkt(ACK)`
- `udt_send(sndpkt)`

Receiver

- **Wait for receive**

### Code
- `rdt_rcv (rcvpkt) && corrupt(rcvpkt)`
- `sndpkt = make_pkt(NAK)`
- `udt_send(sndpkt)`
- `rdt_rcv (rcvpkt) && not_corrupt(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver(data)`
- `sndpkt = make_pkt(ACK)`
- `udt_send(sndpkt)`

### Messages
- **We received a bad packet**
  - Send a NAK
- **We received a good packet**
  - Deliver it to the app
  - Send an ACK
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
- seq=0, data 0
- seq=0, data 0
- seq=1, data 1
- seq=1, data 1

retransmission of previous packet

retransmission of previous packet
RDT over a channel with bit errors

Sender

Wait for send (seq=0)

rdt_send(data)
packet = make_pkt(0, data, checksum)
udt_send(data)

Transmit a packet seq=0

Wait for ACK/NAK 0

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sendpkt)

Retransmit if NAK or corrupt

Wait for send (seq=1)

rdt_rcv(rcvpkt) &&
not_corrupt(rcvpkt) && isACK(rcvpkt))
udt_send(sendpkt)

Transmit a packet seq=1

The receiver got a good packet

rdt_rcv(rcvpkt) &&
not_corrupt(rcvpkt) && isACK(rcvpkt))
udt_send(sendpkt)

Retransmit if NAK or corrupt

Wait for ACK/NAK 1

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sendpkt)

The receiver got a good packet

rdt_rcv(rcvpkt) &&
not_corrupt(rcvpkt) && isACK(rcvpkt))
udt_send(sendpkt)

Transmit a packet seq=1

March 29, 2016
CS 352 © 2013-2016 Paul Krzyzanowski
RDT over a channel with bit errors

Receiver

**Corrupt data – send NAK**

\[
\text{rdt\_rcv(rcvpkt) && corrupt(rcvpkt)} \\
\text{sndpkt = makepkt(NAK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

**Duplicate data – send ACK**

\[
\text{rdt\_rcv(rcvpkt) && not\_corrupt(rcvpkt) && has\_seq1(rcvpkt)} \\
\text{sndpkt = makepkt(ACK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

Wait for receive (seq=0)

**Received seq=0**

*Deliver to app; Send ACK*

\[
\text{rdt\_rcv(rcvpkt) && not\_corrupt(rcvpkt) && has\_seq0(rcvpkt)} \\
\text{extract(rcvpkt, data)} \\
\text{deliver(data)} \\
\text{sndpkt = make(ACK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

Wait for receive (seq=1)

**Received seq=1**

*Deliver to app; Send ACK*

\[
\text{rdt\_rcv(rcvpkt) && not\_corrupt(rcvpkt) && has\_seq1(rcvpkt)} \\
\text{extract(rcvpkt, data)} \\
\text{deliver(data)} \\
\text{sndpkt = make(ACK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

**Corrupt data – send NAK**

\[
\text{rdt\_rcv(rcvpkt) && corrupt(rcvpkt)} \\
\text{sndpkt = makepkt(NAK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

**Duplicate data – send ACK**

\[
\text{rdt\_rcv(rcvpkt) && not\_corrupt(rcvpkt) && has\_seq0(rcvpkt)} \\
\text{sndpkt = makepkt(ACK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

**Corrupt data – send NAK**

\[
\text{rdt\_rcv(rcvpkt) && corrupt(rcvpkt)} \\
\text{sndpkt = makepkt(NAK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

**Duplicate data – send ACK**

\[
\text{rdt\_rcv(rcvpkt) && not\_corrupt(rcvpkt) && has\_seq1(rcvpkt)} \\
\text{sndpkt = makepkt(ACK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

**Corrupt data – send NAK**

\[
\text{rdt\_rcv(rcvpkt) && corrupt(rcvpkt)} \\
\text{sndpkt = makepkt(NAK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

**Duplicate data – send ACK**

\[
\text{rdt\_rcv(rcvpkt) && not\_corrupt(rcvpkt) && has\_seq0(rcvpkt)} \\
\text{sndpkt = makepkt(ACK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

**Corrupt data – send NAK**

\[
\text{rdt\_rcv(rcvpkt) && corrupt(rcvpkt)} \\
\text{sndpkt = makepkt(NAK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]

**Duplicate data – send ACK**

\[
\text{rdt\_rcv(rcvpkt) && not\_corrupt(rcvpkt) && has\_seq1(rcvpkt)} \\
\text{sndpkt = makepkt(ACK, checksum)} \\
u\text{dt\_send(sndpkt)}
\]
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
RDT over a channel with bit errors – no NAK

**Sender**

1. **START**
   - Wait for send (seq=0)

2. Transmit a packet seq=0
   - `rdt_send(data)`
   - `packet = make_pkt(0, data, checksum)`
   - `udt_send(data)`

3. **Wait for ACK 0**
   - The receiver got a good packet
   - `rdt_rcv(rcvpkt) && not_corrupt(rcvpkt) && isACK(rcvpkt, 1)`
   - `udt_send(sendpkt)`
   - Retransmit if corrupt or wrong ACK
   - `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 1))`
   - `udt_send(sendpkt)`

4. **Wait for send (seq=1)**
   - Transmit a packet seq=1
   - `rdt_send(data)`
   - `packet = make_pkt(1, data, checksum)`
   - `udt_send(data)`

5. **Wait for ACK 1**
   - The receiver got a good packet
   - `rdt_rcv(rcvpkt) && not_corrupt(rcvpkt) && isACK(rcvpkt, 0)`
   - `udt_send(sendpkt)`

6. **Like the previous FSM but isNAK is replaced with an isACK check for the wrong ACK #.**
RDT over a channel with bit errors – no NAK

**Receiver**

*Wait for receive (seq=0)*

```
rdt_rcv(rcvpkt) && not_corrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt, data)
deliver(data)
sndpkt = make(ACK, 0, checksum)
udt_send(sndpkt)
```

*Received seq=0*
*Deliver to app; Send ACK #0*

*Corrupt or duplicate packet. Send ACK with previous # (1)*

```
routine(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))
sndpkt = makepkt(ACK, 0, checksum)
udt_send(sndpkt)
```

*Receiver*

*Wait for receive (seq=1)*

```
routine(rcvpkt) && not_corrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt, data)
deliver(data)
sndpkt = make(ACK, 1, checksum)
udt_send(sndpkt)
```

*Received seq=1*
*Deliver to app; Send ACK #1*

*Corrupt or duplicate packet. Send ACK with previous # (0)*

```
routine(rcvpkt) && (corrupt(rcvpkt) || has_seq0(rcvpkt))
sndpkt = makepkt(ACK, 0, checksum)
udt_send(sndpkt)
```

*Corrupt or duplicate packet. Send ACK with previous # (1)*

```
routine(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))
sndpkt = makepkt(ACK, 1, checksum)
udt_send(sndpkt)
```
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

**Sender**

- **Wait for send (seq=0)**
  - rdt_send(data)
  - packet = make_pkt(0, data, checksum)
  - udt_send(data)
  - start_timer

  **Transmit a packet seq=0**

  - *Rebroadcast if corrupt, wrong ACK, or timeout*

- **Wait for send (seq=1)**
  - rdt_send(data)
  - packet = make_pkt(1, data, checksum)
  - udt_send(data)
  - start_timer

  **Transmit a packet seq=1**

  - *Rebroadcast if corrupt, wrong ACK, or timeout*

- **Wait for ACK/NAK 1**

  - rdt_rcv(rcvpkt) &&
  - not_corrupt(rcvpkt) && isACK(rcvpkt, 1))
  - stop_timer

- **Wait for ACK 0**

  - rdt_rcv(rcvpkt) &&
  - (corrupt(rcvpkt) || isACK (rcvpkt,1))
  - (do nothing; wait for a timeout)

  - udt_send(sendpkt)
  - start_timer

  **Transmit a packet seq=0**

  - *The receiver got a good packet*

  - rdt_rcv(rcvpkt) &&
  - not_corrupt(rcvpkt) && isACK(rcvpkt, 1))
  - stop_timer

  - The receiver got a good packet

  - rdt_rcv(rcvpkt) &&
  - not_corrupt(rcvpkt) && isACK(rcvpkt, 0))
  - stop_timer

- **Wait for send (seq=0)**

  - *Rebroadcast if corrupt, wrong ACK, or timeout*

  - rdt_rcv(rcvpkt) &&
  - (corrupt(rcvpkt) || isACK (rcvpkt, 0))
  - (do nothing; wait for a timeout)

  - udt_send(sendpkt)
  - start_timer

**Like the previous FSM but with a timer set on transmit and a timeout check when waiting for an ACK**
RDT – Alternating Bit Protocol: no loss

Sender

Send P0

Receive ACK0

Send P1

Receive ACK1

Send P0

Receive ACK0

Receiver

Receive P0

Send ACK0

Receive P1

Send ACK1

Receive P0

Send ACK0

Send ACK0

Receive P0

Send ACK1

Receive P1

Send ACK0

Receive P0

Send ACK1

Receive P1

Send ACK0

Receive P0

Send ACK1

Receive P1

Send ACK0

Receive P0

Send ACK1

Receive P1

Send ACK0

Receive P0

Send ACK1

Receive P1

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0

Send ACK0

Receive P0
RDT – Alternating Bit Protocol: lost Packet

Sender
- Send P0
- Receive ACK0
- Send P1

Receiver
- Receive P0
- Send ACK0
- Receive ACK1
- Send P0
- Receive P1
- Send ACK1
- Receive P0
- Send ACK0

Timeout – no ACK
- Resend P1

RDT – Alternating Bit Protocol: lost ACK

Sender
Send P0
Receive ACK0
Send P1
Receive ACK1
Timeout – no ACK
Resend P1
Receive ACK1
Send P0
Receive P1 (detect duplicate)
Send ACK1
Receiver
Receive P0
Send ACK0
Receive P0
Send ACK0
Receive P1
Send ACK1
Receive P1
Send ACK1
Receive P1
Send ACK1
Receive P0
Send ACK0

RDT – Alternating Bit Protocol: early timeout

Sender

Send P0
Receive ACK0
Send P1
Receive ACK1
Discard it

Receiver

Receive P0
Send ACK0
Receive P0
Send ACK0
Receive P1
Send ACK1
Receive P1
Send ACK1
Receive P1
(detect duplicate)
Receive P0
Send ACK0

Timeout – delayed ACK
Resend P1
Receive ACK1
Send P0
Receive ACK1
Send P0
Receive ACK0
Send P1
Receive ACK1
Send P1
Receive ACK0
Send ACK1
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: \( d_{\text{trans}} = L \div R = 8,000 \div 10^9 = 8 \mu s \)

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

\[
U = \frac{L / R}{RTT + (L / R)} = \frac{0.008}{30.008} = 0.00027 = 0.027\%
\]

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

received ACK

sent, no ACK

not yet sent

window size $N$

base

next_seqnum

These packets can be sent

Packets with sequence # $\geq$ base + $N$ cannot be sent. We can only have $N$ unacknowledged packets
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.

Packets with sequence # ≥ base + $N$ cannot be sent. We can only have $N$ 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

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

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

$N$ = Sliding Window Protocol

The window slides as packets are acknowledged

received ACK

sent, no ACK

not yet sent

base

next_seqnum

Packets with sequence # ≥ base + $N$ cannot be sent. We can only have $N$ unacknowledged packets
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

window size $N$

base

next_seqnum

Packets with sequence # $\geq$ base + $N$ cannot be sent. We can only have $N$ unacknowledged packets

GBN = Sliding Window Protocol
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

Sender

Send data if it's in the window (we can have at most N unacknowledged packets)

\[ \text{rdt\_send(data)} \]
\[ \text{if } (\text{next\_seqnum} < \text{base}+\text{N}) \{ \]
\[ \text{sndpkt}[\text{next\_seqnum}] = \text{make\_pkt(} \text{next\_seqnum, data, checksum)} \]
\[ \text{udt\_send(} \text{sndpkt}[\text{next\_seqnum}]\text{)} \]
\[ \text{if } (\text{base} == \text{next\_seqnum}) \]
\[ \text{start\_timer} \]
\[ \text{next\_seqnum}++ \]
\[ \} \ else \{ \]
\[ \text{refuse\_data(data)} \] // cannot send

Ignore corrupted ACKs

\[ \text{rdt\_rcv(rcvpkt) && corrupt(rcvpkt)} \]

Go Back N:
Timeout means resend all unacknowledged packets

Timeout:

\[ \text{timeout} \]
\[ \text{start\_timer} \]
\[ \text{for } (i = \text{base}; i < \text{next\_seqnum}; i++) \]
\[ \text{udt\_send(sndpkt[i])} \]

Cumulative acknowledgement:
Receipt of a sequence number n ACK means that all packets up to and including n have been received

\[ \text{rdt\_rcv(rcvpkt) && not\_corrupt(rcvpkt)} \]
\[ \text{base} = \text{get\_acknum(rcvpkt)}+1 \]
\[ \text{if } (\text{base} == \text{next\_seqnum}) \]
\[ \text{stop\_timer} \] // we have the latest ACK
\[ \] else
\[ \text{start\_timer} \] // still waiting for ACKs

March 29, 2016
CS 352 © 2013-2016 Paul Krzyzanowski
Extended FSM for a GBN receiver

Receiver

We received a good packet with the expected sequence number
rdt_rcv(rcvpkt) && not_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++

Initialize
expected_seqnum = 1
sndpkt = makepkt(0, ACK, checksum)

If we receive anything else, send the last ACK
default

Wait

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 $P(\text{channel error})$ 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 Windows

Sender’s view of sequence numbers

window size $N$

Send_base

Next_seqnum

Receiver’s view

window size $N$

rcv_base

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

Receiver's view of sequence numbers:
- rcv_base

Legend:
- Green: Sent & got ACK
- Yellow: Sent but no ACK
- Orange: Ready to send
- Red: Cannot send

Out of order, ACKed
- Green

Expected
- Yellow

Acceptable
- Orange

Unusable
- Red

March 29, 2016
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
Out-of-order ACKs

Sender’s view of sequence numbers

- window size $N$
- send_base
- next_seqnum

When an ACK for this packet is received, send_base is advanced to the next packet with no ACK.
Out-of-order ACKs

Sender’s view of sequence numbers

- **send_base**: Represents the last packet sent.
- **next_seqnum**: Represents the next packet to be sent.

- Window size $N$:
  - **Sent & got ACK**: Packet has been received.
  - **Sent but no ACK**: Packet has been sent but not acknowledged.
  - **Ready to send**: Packet is ready to be sent.
  - **Cannot send**: Packet cannot be sent.

The ACK for this packet was received, so send_base was advanced to the next packet with no ACK.
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
Selective Repeat: receiving packets

Receiver’s view

When this packet is received, we can deliver it to the app and deliver all received packets immediately after it.

rcv_base

window size N

Out of order, ACKed
Expected
Acceptable
Unusable
Selective Repeat: receiving packets

Receiver’s view

deliver to app

rcv_base

window size $N$

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 start of the window on the receiver is not always the same as the start of the window on the sender.
The end