Chapter 3: Transport Layer

- Transport layer services
- Multiplexing/demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - Reliable transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control
Transport Services and Protocols

- **Network layer service**: data transfer between end systems
- **Transport layer**: data transfer between processes running on different hosts
- **Transport layer relies on, enhances, network layer services**
Internet Transport-layer Protocols

- **Reliable, in-order unicast delivery (TCP)**
  - connection setup
  - flow control
  - congestion control

- **Unreliable ("best-effort"), unordered unicast or multicast delivery:** UDP

- **Services NOT available:**
  - for real-time applications - delay bound requirement
  - bandwidth guarantees
  - reliable multicast
Demultiplexing

- TPDU: transport protocol data unit

Demultiplexing: delivering received segments to correct app layer processes
Multiplexing

- Gather data from multiple app processes
- Envelop data with header (later used for demultiplexing)
UDP: User Datagram Protocol [RFC 768]

- “Best effort” Internet transport protocol
- UDP segments may be:
  - Errored
  - Lost
  - Delayed
  - duplicated or
  - delivered out of order
- Error detection, handling and recovery by the upper layer applications

- **Connectionless:**
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- no congestion control: UDP can blast away as fast as desired
**UDP: more**

- Header - 8 bytes
- Often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses (why?):
  - DNS (53)
  - SNMP (161, 162)

**UDP segment format**

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

Length, in bytes of UDP segment, including header

Application data (message)
**UDP checksum**

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

**Sender:**
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

**Receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless? More later …
**UDP Datagram Dispatching**

![Diagram of UDP Datagram Dispatching]

**Figure 12.5** Example of demultiplexing one layer above IP. UDP uses the UDP destination port number to select an appropriate destination port for incoming datagrams.
# UDP Ports Assigned

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Keyword</th>
<th>UNIX Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>ECHO</td>
<td>echo</td>
<td>Echo</td>
</tr>
<tr>
<td>9</td>
<td>DISCARD</td>
<td>discard</td>
<td>Discard</td>
</tr>
<tr>
<td>11</td>
<td>USERS</td>
<td>systat</td>
<td>Active Users</td>
</tr>
<tr>
<td>13</td>
<td>DAYTIME</td>
<td>daytime</td>
<td>Daytime</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>netstat</td>
<td>Who is up or NETSTAT</td>
</tr>
<tr>
<td>17</td>
<td>QUOTE</td>
<td>qotd</td>
<td>Quote of the Day</td>
</tr>
<tr>
<td>19</td>
<td>CHARGEN</td>
<td>chargen</td>
<td>Character Generator</td>
</tr>
<tr>
<td>37</td>
<td>TIME</td>
<td>time</td>
<td>Time</td>
</tr>
<tr>
<td>42</td>
<td>NAMESERVER</td>
<td>name</td>
<td>Host Name Server</td>
</tr>
<tr>
<td>43</td>
<td>NICNAME</td>
<td>whois</td>
<td>Who Is</td>
</tr>
<tr>
<td>53</td>
<td>DOMAIN</td>
<td>nameserver</td>
<td>Domain Name Server</td>
</tr>
<tr>
<td>67</td>
<td>BOOTPS</td>
<td>bootps</td>
<td>Bootstrap Protocol Server</td>
</tr>
<tr>
<td>68</td>
<td>BOOTPC</td>
<td>bootpc</td>
<td>Bootstrap Protocol Client</td>
</tr>
<tr>
<td>69</td>
<td>TFTP</td>
<td>tftp</td>
<td>Trivial File Transfer</td>
</tr>
<tr>
<td>111</td>
<td>SUNRPC</td>
<td>sunrpc</td>
<td>Sun Microsystems RPC</td>
</tr>
<tr>
<td>123</td>
<td>NTP</td>
<td>ntp</td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td>161</td>
<td>-</td>
<td>snmp</td>
<td>SNMP net monitor</td>
</tr>
<tr>
<td>162</td>
<td>-</td>
<td>snmp-trap</td>
<td>SNMP traps</td>
</tr>
<tr>
<td>512</td>
<td>-</td>
<td>biff</td>
<td>UNIX comsat</td>
</tr>
<tr>
<td>513</td>
<td>-</td>
<td>who</td>
<td>UNIX rwho daemon</td>
</tr>
<tr>
<td>514</td>
<td>-</td>
<td>syslog</td>
<td>system log</td>
</tr>
<tr>
<td>525</td>
<td>-</td>
<td>timed</td>
<td>Time daemon</td>
</tr>
</tbody>
</table>

Figure 12.6 An illustrative sample of currently assigned UDP ports showing the standard keyword and the UNIX equivalent; the list is not exhaustive. To the extent possible, other transport protocols that offer identical services use the same port numbers as UDP.
Reliable Data Transfer
Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Error Control

For service interfaces that provide “reliable” delivery

Problem 1:

“how to let sender know receiver correctly receives the data that sender sends?”

=> Need some feedback from the receiver

Solution 1.1: Use Positive or Negative Acknowledgement (ACK)

Solution 1.2: Use timer in the presence of error or hardware malfunction.
Error Control (cont’d)

- Sender starts a timer when transmits a frame out.
  - **Timeout interval** must be properly set.
  - **At least a round trip time** from sender to receiver
  - Sum of transmission time from sender to receiver, processing time delay at receiver, and ack transmission time from receiver to sender.
Problem 2:

“How to distinguish received packets, i.e. packet retransmission and duplicate packet?”

Solution 2.1: Packets are numbered

Sequence number assignment
Summary

Positive ACK + timer
- P-ACK(n) by receiver
- P-ACK(n) lost
- Timer (n) goes off at sender
- Sender retransmits packet(n)

Negative ACK + timer at receiver
- Timer (n) goes off
- N-ACK(n) by receiver
- Sender retransmits packet(n)
Discussions

- Congestion at receiver
  - Ack(n) is on the way
  - Timer(n) goes off;
  - **Sender** retransmits packet(n)
  - **Receiver** receives duplicate packet(n)
  - **Receiver** discards retransmitted packet(n)
Reliable data transfer: getting started

- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use **Finite State Machines (FSM)** to specify sender, receiver
Rdt2.0: channel with bit errors

- Error detection
  - UDP checksum

- Recover from errors:
  - Receiver feedback: control msgs (ACK, NAK) 
    rcvr→sender
    - acknowledgements (ACKs): receiver explicitly 
      tells sender that pkt received OK
    - negative acknowledgements (NAKs): receiver 
      explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
rdt2.0: FSM specification

sender FSM

wait for call from above

wait for ACK or NAK

rdt_send(data)
compute checksum
make_pkt(sndpkt, data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& isACK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& isNACK(rcvpkt)
udt_send(NACK)

receiver FSM

wait for call from below

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)

3: Transport Layer  3a-20
Acknowledgement Packet

- Can be transmitted either via separate packets (e.g. use “type” field in the frame header to distinguish them) or

- Piggybacking
  - Attach acknowledgement information to the outgoing data packets.
  - i.e. include an “ack” field in the packet header

- Problem
  - May result in variable delays for ack transmission
Acknowledgement Packet (cont’d)

Advantages
- Use less resources (e.g., bandwidth)
- Less interrupts to local processing unit

“How long should the receiver wait for a packet onto which to piggyback the ACK?”

Solution:
- Wait for a fixed amount of time $T$
- If a new frame to transmit, piggyback the ack onto it
- Otherwise, send a separate ack packet
- Note $T$ should be determined based on the traffic characteristics, e.g., RTT.
Robustness

We say a protocol is robust if it works under all circumstances, such as errored packets, lost packets, and premature timeouts or their combinations.)
Flow Control
Basics on Flow Control: Sliding Window Protocols

- Stop-and-wait (one bit) sliding window
- Go-back-n
- Selective repeat

Note: these methods differ in efficiency, complexity and buffer requirements.
Sliding window
Sliding Window Protocols - basic idea

- Each outbound packet contains a sequence number, ranging from 0 to some maximum number (usually 0~2^n-1 using n-bit field)

- **Sender** maintains a list of consecutive sequence numbers, corresponding to packets it is *permitted* to send that is called *sending window*.

- **Receiver** also maintains a list of consecutive sequence numbers, corresponding to packets it is *permitted* to accept that is called *receiving window*. 
Sliding Window Scheme

Sender

0 1 2 3 ... i ... j ... N-1 0 1

Outstanding (unacked)

Maximum window size

time
Stop-and-Go

- Maximum window size is one.
- Sender sends a packet and waits for its ACK before sending the next one.

Figure 13.1 A protocol using positive acknowledgement with retransmission in which the sender awaits an acknowledgement for each packet sent. Vertical distance down the figure represents increasing time and diagonal lines across the middle represent network packet transmission.
Timer-based Retransmission

(a) operation with no loss

(b) lost packet

3: Transport Layer  3a-30
Timer-based Retransmission (cont’d)

(c) lost ACK
(d) premature timeout
Stop and Wait: discussion

Consider to use stop-and-wait protocol for a network with long propagation delay.

Let

- $B$: bandwidth of a satellite channel
- $\tau$: round-trip propagation delay
- $L$: frame size

Assume $B=50\text{kbps}$, $\tau=500\text{msec}$, $L=1000\text{bits}$
Stop and Wait: discussion (cont’d)

- Long propagation delay
- Short transmission time

Utilization (efficiency) = \( \frac{20}{500+20} \approx 4\% \)
( \( U = \frac{L/B}{(\tau+L/B)} \) )

- Network protocol limits use of physical resources!
Pipelining

- To achieve better efficiency
  - Allows the sender to transmit up to \( w \) packets before blocked
  - Multiple outstanding packets
- Issues: determine \( w \)?
  - e.g., consider the previous example
  - \( 500/20=25 \) \( \rightarrow \) \( w=25 \)
- \( w \) is the maximum number of outstanding unacked packets
  - Wrong!
- Issue
  - What happens if a packet (data or ack) in the middle of the long stream is damaged or lost?
- Two approaches: go-back-n & Selective repeat
**rdt3.0: stop-and-wait operation**

![Diagram showing stop-and-wait operation]

- First packet bit transmitted, $t = 0$
- Last packet bit transmitted, $t = L / R$
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = RTT + L / R$

**Formula for $U_{sender}$**

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} = 0.00027$$
Pipelining: increased utilization

\[ U_{\text{sender}} = \frac{3 \times \frac{L}{R}}{\text{RTT} + \frac{L}{R}} = \frac{0.024}{30.008} = 0.0008 \]

Increase utilization by a factor of 3!
$U_{\text{pipe}} = \frac{w \cdot (L/B)}{\tau + (L/B)}$

$U_{\text{pipe}} = w \cdot U_{\text{stop-and-wait}}$
Pipelined protocols

**Pipelining**: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

- Two generic forms of pipelined protocols: go-Back-N, selective repeat

![Diagram of a stop-and-wait protocol in operation](a)

![Diagram of a pipelined protocol in operation](b)
Sliding window
Sender's Sliding window

send_base  nextseqnum

window size $N$

front  rear

already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable
Go-Back-N Sliding Window Protocol

- When receiver receives an error packet, it discards all subsequent packets, i.e. drop all out-of-sequence packets.

- Drawback
  - Waste bandwidth in high error rate channel

- Advantage
  - Simpler operational complexity for receivers
Go-Back-N Sliding Window Protocol

Sender:

- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed

ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”.

- may deceive duplicate ACKs (see receiver).
- timer for each in-flight pkt.
- timeout(n): retransmit pkt n and all higher seq # pkts in window.
GBN in action

sender
- send pkt0
- send pkt1
- send pkt2
- send pkt3 (wait)

receiver
- rcv pkt0
- send ACK0
- rcv pkt1
- send ACK1

Window advances
- pkt2 timeout
- send pkt2
- send pkt3
- send pkt4
- send pkt5

(rcv ACK0
- send pkt4
- rcv ACK1
- send pkt5

(rcv pkt3, discard
- send ACK1

(rcv pkt4, discard
- send ACK1

(rcv pkt5, discard
- send ACK1

(rcv pkt2, deliver
- send ACK2
- rcv pkt3, deliver
- send ACK3

3: Transport Layer 3a-43
GBN: sender extended FSM

```
rdt_send(data)

if (nextseqnum < base+N) {
    compute chksum
    make_pkt(sndpkt(nextseqnum)),nextseqnum,data,chksum
    udt_send(sndpkt(nextseqnum))
    if (base == nextseqnum)
        start_timer
        nextseqnum = nextseqnum + 1
    else
        refuse_data(data)
}
```

```
rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)

base = getacknum(rcvpkt)+1
if (base == nextseqnum)
    stop_timer
else
    start_timer

timeout

start_timer
udt_send(sndpkt(base))
udt_send(sndpkt(base+1))
......
udt_send(sndpkt(nextseqnum-1))
```
GBN: receiver extended FSM

receiver simple:
- **ACK-only**: always send ACK for correctly-received pkt with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember `expectedseqnum`
- **out-of-order pkt**:
  - **discard** (don't buffer) -> no receiver buffering!
  - ACK pkt with highest in-order seq #

- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && 
  hasseqnum(rcvpkt,expectedseqnum)
- extract(rcvpkt,data)
- make_pkt(sndpkt,ACK,expectedseqnum)
- udt_send(sndpkt)
Sliding Window Protocol using “Selective Repeat”

- Receiver is able to accept and buffer all correctly received, out-of-sequence packets.
- Receiver individually acknowledges all correctly received pkts.
- Eventual in-order delivery to upper layer

Algorithm at the receiver
- For an out-of-sequence packet, check if falls within the receiving window.
- Check if it is not a duplicate
- If both are ok, store the packet in the buffer
Sliding Window Protocol using “Selective Repeat” (cont’d)

Algorithm at the sender

- Sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt

- Sender window
  - N consecutive seq #’s
  - again limits seq #s of sent, unACKed pkts
“Selective Repeat” in action

Outstanding

pkt0 sent

0 1 2 3 4 5 6 7 8 9

pkt1 sent

0 1 2 3 4 5 6 7 8 9

pkt2 sent

0 1 2 3 4 5 6 7 8 9

pkt3 sent, window full

0 1 2 3 4 5 6 7 8 9

ACK0 rcvd, pkt4 sent

0 1 2 3 4 5 6 7 8 9

pkt2 timeout, pkt2 resent

0 1 2 3 4 5 6 7 8 9

ACK1 rcvd, pkt5 sent

0 1 2 3 4 5 6 7 8 9

Expected

pkt0 rcvd, delivered, ACK0 sent

0 1 2 3 4 5 6 7 8 9

pkt1 rcvd, delivered, ACK1 sent

0 1 2 3 4 5 6 7 8 9

pkt3 rcvd, buffered, ACK3 sent

0 1 2 3 4 5 6 7 8 9

pkt4 rcvd, buffered, ACK4 sent

0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, deliver pkts 2, 3, 4

ACK2 sent

0 1 2 3 4 5 6 7 8 9

pkt5 rcvd, delivered, ACK5 sent

0 1 2 3 4 5 6 7 8 9

3: Transport Layer  3a-48
Selective repeat: dilemma

Example:
- seq #’s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?
Selective repeat

**sender**

data from above:
- if next available seq # in window, send pkt

timeout(n):
- resend pkt n, restart timer

ACK(n) in [sendbase, sendbase+N]:
- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

**receiver**

pkt n in [rcvbase, rcvbase+N-1]:
- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in [rcvbase-N, rcvbase-1]:
- ACK(n)
otherwise:
- ignore