Transport Layer
3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
  – segment structure
  – reliable data transfer
  – flow control
  – connection management
3.6 Principles of congestion control
3.7 TCP congestion control
Transport Services and Protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP
Internet Transport Layer Protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
3.1 Transport-layer services
3.2 Multiplexing and demultiplexing [Brief Look]
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
3.6 Principles of congestion control
3.7 TCP congestion control
Connection-Oriented Demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number

- receiving host uses all four values to direct segment to appropriate socket.

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple

- Web servers have different sockets for each connecting client.
  - non-persistent HTTP will have different socket for each request.
Connection-Oriented Demux

Client IP: A

SP: 9157
DP: 80
S-IP: A
D-IP: C

Server IP: C

SP: 5775
DP: 80
S-IP: B
D-IP: C

Client IP: B

SP: 9157
DP: 80
S-IP: B
D-IP: C
Connection-Oriented Demux

Threaded Web Server

P1

client
IP: A

SP: 9157
DP: 80
S-IP: A
D-IP: C

P4

server
IP: C

SP: 5775
DP: 80
S-IP: B
D-IP: C

P2

P3

Client
IP: B

SP: 9157
DP: 80
S-IP: B
D-IP: C
3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
   - segment structure
   - reliable data transfer
   - flow control
   - connection management
3.6 Principles of congestion control
3.7 TCP congestion control
UDP: User Datagram Protocol [RFC 768]

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others.

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired.
UDP Details

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

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 segment format
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…
Internet Checksum Example

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

```
1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

<table>
<thead>
<tr>
<th>wraparound</th>
<th>sum</th>
<th>checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1</td>
<td>1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1</td>
<td>1 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1</td>
</tr>
</tbody>
</table>
Kurose's Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

We will use Tanenbaum's Data Link Layer Treatment to study this in place of K&R's Transport Layer Discussion.
Principles of Reliable Data Transfer

- important in application, transport, and data link layers
- top-10 list of important networking topics!

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Principles of Reliable Data Transfer

- important in application, transport, and data link layers
- top-10 list of important networking topics!

characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rtd)
Principles of Reliable Data Transfer

- important in application, transport, and data link layers
- top-10 list of important networking topics!

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

(a) provided service
(b) service implementation
Reliable Data Transfer: Getting Started

**send side**
- `rdt_send()` : called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
- `udt_send()` : called by rdt, to transfer packet over unreliable channel to receiver

**receive side**
- `deliver_data()` : called by rdt to deliver data to upper
- `rdt_rcv()` : called when packet arrives on rcv-side of channel
- `rdt_rcv()` : called when packet arrives on rcv-side of channel

**Diagram:**
- `rdt_send()` : called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
- `udt_send()` : called by rdt, to transfer packet over unreliable channel to receiver
- `deliver_data()` : called by rdt to deliver data to upper
- `rdt_rcv()` : called when packet arrives on rcv-side of channel
### TCP Segment Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>head len</td>
<td>Header length</td>
</tr>
<tr>
<td>URG</td>
<td>URG flag</td>
</tr>
<tr>
<td>ACK</td>
<td>ACK flag</td>
</tr>
<tr>
<td>P</td>
<td>Push flag</td>
</tr>
<tr>
<td>R</td>
<td>Reset flag</td>
</tr>
<tr>
<td>S</td>
<td>Syn flag</td>
</tr>
<tr>
<td>F</td>
<td>Fin flag</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>Receive window</td>
<td>Receive window</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Options</td>
<td>Options</td>
</tr>
<tr>
<td>application data</td>
<td>Application data</td>
</tr>
<tr>
<td>(variable length)</td>
<td>(variable length)</td>
</tr>
</tbody>
</table>

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **Internet checksum** (as in UDP) {needed now for error detection}
- **counting by bytes of data (not segments!)**
- **# bytes rcvr willing to accept**

**URG**: urgent data (generally not used)

**ACK**: ACK # valid

**Internet checksum** (as in UDP) {needed now for error detection}
Explanation of Reliable Data Transport will now be explained using the Data Link Layer.
We’ll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

state: when in this “state” next state uniquely determined by next event

event causing state transition
actions taken on state transition
Rdt1.0: Reliable Transfer over a Reliable Channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel

sender

receiver
Rdt2.0: Channel with Bit Errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors

- the question: how to recover from errors:
  - 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.

- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK, NAK) rcvr->sender
New assumption: underlying channel can also lose packets (data or ACKs)
  - checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Approach: sender waits “reasonable” amount of time for ACK
  - retransmits if no ACK received in this time
  - if pkt (or ACK) just delayed (not lost):
    - retransmission will be duplicate, but use of seq. #’s already handles this
    - receiver must specify seq # of pkt being ACKed
  - requires countdown timer
rdt3.0 Sender

Wait for call 0 from above

Wait for ACK0

Wait for call 1 from above

Wait for ACK1

rdt_send(data)

sndpkt = make_pkt(0, data, checksum)

udt_send(sndpkt)

start_timer

rdt_rcv(rcvpkt)

\[ \Lambda \]

\[ \Lambda \]

rdt_rcv(rcvpkt)

\&\& notcorrupt(rcvpkt)

\&\& isACK(rcvpkt,1)

stop_timer

timeout

udt_send(sndpkt)

start_timer

rdt_rcv(rcvpkt)

\&\& notcorrupt(rcvpkt)

\&\& isACK(rcvpkt,0)

stop_timer

rdt_rcv(rcvpkt)

\[ \Lambda \]

\[ \Lambda \]

rdt_rcv(rcvpkt)

\&\& ( corrupt(rcvpkt) \|\| isACK(rcvpkt,0) )

udt_send(sndpkt)

start_timer

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)

udt_send(sndpkt)

start_timer

timeout

udt_send(sndpkt)

start_timer
Pipelining and Sliding Windows

- Lecture returns back to this point after Data Link Layer.
- Diagrams from textbook!!
Pipelining:: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets.

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

• Two generic forms of pipelined protocols: Go-Back-N and Selective Repeat
Pipelining increases Utilization

first packet bit transmitted, $t = 0$

last bit transmitted, $t = L / R$

first packet bit arrives

last packet bit arrives, send ACK

last bit of 2nd packet arrives, send ACK

last bit of 3rd packet arrives, send ACK

ACK arrives, send next packet, $t = RTT + L / R$

sender

receiver

$U_{sender} = \frac{3 \cdot L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.0008$

Increase utilization by a factor of 3!
Go-back-N: overview

- **sender**: up to N unACKed pkts in pipeline
- **receiver**: only sends cumulative ACKs
  - doesn’t ACK pkt if there’s a gap
- **sender**: has timer for oldest unACKed pkt
  - if timer expires: retransmit all unACKed packets

Selective Repeat: overview

- **sender**: up to N unACKed packets in pipeline
- **receiver**: ACKs individual pkts
- **sender**: maintains timer for each unACKed pkt
  - if timer expires: retransmit only unACKed packet.
**Go-Back-N**

**Sender:**

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

![Diagram showing Go-Back-N](image)

- **ACK(n):** ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- **timer for each in-flight pkt**
- **timeout(n):** retransmit pkt n and all higher seq # pkts in window.
GBN: Sender Extended FSM

```c
void rdt_send(data)
{
    if (nextseqnum < base+N) {
        sndpkt[nextseqnum] = make_pkt(nextseqnum, data, checksum)
        udt_send(sndpkt[nextseqnum])
        if (base == nextseqnum)
            start_timer
        nextseqnum++
    }
    else
        refuse_data(data)
}
```

Diagram:
- `Wait`
- `timeout`
- `start_timer`
- `udt_send(sndpkt[base])`
- `udt_send(sndpkt[base+1])`
- `...`
- `udt_send(sndpkt[nextseqnum-1])`

- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `base = getacknum(rcvpkt)+1`
- `If (base == nextseqnum)`
  - `stop_timer`
- `else`
  - `start_timer`

`base=1`
`nextseqnum=1`
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!
  - Re-ACK pkt with highest in-order seq #
Selective Repeat

- receiver **individually** acknowledges all correctly received packets.
  - buffers packets, as needed, for eventual in-order delivery to upper layer.
- sender only resends packets for which ACK not received.
  - sender timer for each unACKed packet
- **sender window**
  - N consecutive sequence #'s
  - again limits sequence #'s of sent, unACKed packets
Selective Repeat
Sender, Receiver Windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
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
Selective Repeat in Action

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

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

ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9

ACK1 rcvd, pkt5 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

pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9

pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9

ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2,pkt3,pkt4,pkt5 delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9

sliding window
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 is the required relationship between seq # size and window size?
Kurose's Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control
TCP Flow Control

- receive side of TCP connection has a receive buffer:
  - speed-matching service: matching send rate to receiving application’s drain rate.
  - flow control: sender won’t overflow receiver’s buffer by transmitting too much, too fast

- app process may be slow at reading from buffer.
TCP Flow Control: how it works

unused buffer space:

- receiver: advertises unused buffer space by including \texttt{rwnd} value in segment header
  - guarantees receiver’s buffer doesn’t overflow.

- sender: limits \# of unACKed bytes to \texttt{rwnd}

\texttt{rwnd} known as the receiver’s advertised window.
3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
3.6 Principles of congestion control
3.7 TCP congestion control