Transport Layer


Reference: Computer Networks: A Systems Approach. Larry Peterson, Bruce Davie, Morgan Kaufmann
Transport Layer

• Multiplexing and Demultiplexing
• Connectionless Transport: UDP

✓ Reliable Data Transfer
• Connection-oriented Transport: TCP
  • Segment Structure
  • Reliable Data Transfer
  • Flow Control
  • Connection Management

• Congestion Control
• TCP Congestion Control
• **New assumption**: underlying channel can also lose packets (data, ACKs)
  • Checksum, sequence number, 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 packet (or ACK) just delayed (not lost):
  • Retransmission will be duplicate, but sequence numbers already handle this
  • Receiver must specify sequence number of packets being ACKed

• Requires countdown timer
RDT3.0 Sender

```plaintext
rdt_send(data)

sndpkt-make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)

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

timeout
udt_send(sndpkt)
start_timer

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

Lambda

rdt_send(data)

sndpkt-make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer
```
RDT3.0 In Action

(a) No loss

Sender
send pkt0
rcv ACK0
send pkt1
rcv ACK1
send pkt0

Receiver
Pkt0
ACK0
Pkt1
ACK1
Pkt0
ACK0
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt0
send ACK0

(b) Packet loss

Sender
send pkt0
rcv ACK0
send pkt1
rcv pkt1
send ACK1
rcv pkt0
send ACK0

Receiver
Pkt0
ACK0
Pkt1
ACK1
Pkt0
ACK0
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt0
send ACK0

RDT3.0 in action

(c) ACK loss

(d) Premature timeout/ delayed ACK
RDT3.0 Performance

• RDT3.0 is correct but low performance

• Example: 1 Gbps link, 15ms propagation delay, 8000 bit packet:

\[
d_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits / packet}}{10^9 \text{ bits / sec}} = 8 \text{ microseconds}
\]
RDT3.0 Performance

- $U_{\text{sender}}$: **Utilization** – fraction of time sender busy sending

$$U_{\text{sender}} = \frac{L}{RTT + \frac{L}{R}} = \frac{.008}{30.008} = 0.00027$$

- If RTT = 30 msec, 1KB packet every 30 msec: 33kB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources!
RDT3.0 Stop and Wait Operation

\[ U_{\text{sender}} = \frac{L}{RTT + \frac{L}{R}} = \frac{0.008}{30.008} = 0.00027 \]
Pipelined Protocols

- **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, selective repeat**
Pipelining: Increased Utilization

3-packet pipelining increases utilization by a factor of 3!

\[
U_{sender} = \frac{3L}{RTT + \frac{L}{R}} = \frac{0.0024}{30.008} = 0.00081
\]
Pipelined Protocols: Overview

Go-back-N
- Sender can have up to N un-acked packets in pipeline
- Receiver only sends cumulative ack
  - Does not ack packet if there’s a gap
- Sender has timer for oldest un-acked packet
  - When timer expires, retransmit all un-acked packets

Selective Repeat
- Sender can have up to N un-acked packets in pipeline
- Receiver sends individual ack for each packet
- Sender maintains timer for each un-acked packet
  - When timer expires, retransmit only that un-acked packet

Go-Back N: Sender

- k-bit sequence number in packet header
- **Window** of up to N, consecutive un-acked packets allowed

```
• ACK (n): ACKs all packets up to, including sequence number n – Cumulative ACK
  • May receive duplicate ACKs (see receiver)
• Timer for oldest in-flight packet
• Timeout(n): retransmit packet n and all higher sequence number packets in window
```
GBN: Sender Extended FSM

```
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)

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

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

```

GBN: Receiver Extended FSM

- \( \text{rdt_rcv(rcvpkt)} \)
- \( \&\& \text{notcorrupt(rcvpkt)} \)
- \( \&\& \text{hasseqnum(rcvpkt, expectedseqnum)} \)

- \( \text{extract(rcvpkt, data)} \)
- \( \text{deliver_data(data)} \)
- \( \text{sndpkt=make_pkt(expectedseqnum, ACK, checksum)} \)
- \( \text{udt_send(sndpkt)} \)
- \( \text{expectedseqnum++} \)

Diagram:
- Transition from Wait state to default state:
  - \( \text{udt_send(sndpkt)} \)
- Transition from Wait state to \( \text{\&\& expectedseqnum=1} \):
  - \( \text{sndpkt=make_pkt(0, ACK, checksum)} \)

GBN: Receiver Extended FSM

- ACK-only: always send ACK for correctly-received packet with highest **in-order** sequence number
  - May generate duplicate ACKs
  - Need only remember `expectedseqnum`

- Out-of-order packet:
  - Discard (don’t buffer): **no receiver buffering!**
  - Re-ACK packet with highest in-order sequence number
GBN In Action

sender window (N=4)

sender

receiver

send pkt0
send pkt1
send pkt2
send pkt3 (wait)
rcv ack0, send pkt4
rcv ack1, send pkt5
ignore duplicate ACK

loss

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1
receive pkt4, discard,
(re)send ack1
receive pkt5, discard,
(re)send ack1

pkt 2 timeout

send pkt2
send pkt3
send pkt4
send pkt5

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

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 un-ACKed packet

- Sender window
  - \( N \) consecutive sequence numbers
  - Limits sequence numbers of sent, unACKed packets
Selective Repeat: Sender, Receiver Windows

a. Sender view of sequence numbers

b. Receiver view of sequence numbers

Key:
- Already ACK’d
- Sent, not yet ACK’d
- Usable, not yet sent
- Not usable

Window size $N$
Selective Repeat: Sender

Data from above

- If next available sequence number in window, send packet

Timeout(n)

- Resend packet n, restart timer

ACK(n) in [sendbase, sendbase+N]:

- Mark packet n as received
- If n smallest un-ACKed packet, advance window base to next un-ACKed sequence number
Selective Repeat: Receiver

Packet n in \([rcv\text{base}, \; rcv\text{base}+N-1]\)

- Send ACK(n)
- Out-of-order: buffer
- In-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

Packet n in \([rcv\text{base}-N, \; rcv\text{base}-1]\)

- ACK(n)

Otherwise

- Ignore
Selective Repeat In Action

sender window (N=4)

sender

0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver

0 1 2 3 4 5 6 7 8
receive pkt0, send ack0
receive pkt1, send ack1
rcv ack0, send pkt4
rcv ack1, send pkt5
record ack3 arrived

xloss

0 1 2 3 4 5 6 7 8
receive pkt3, buffer, send ack3
receive pkt4, buffer, send ack4
receive pkt5, buffer, send ack5
pkt 2 timeout
send pkt2
record ack4 arrived
record ack5 arrived

rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?
Selective Repeat: Dilemma

• Example:
  • Sequence numbers: 0, 1, 2, 3
  • Window size=3
  • Receiver sees no difference in two scenarios!
  • Duplicate data accepted as new in (b)

• Receiver can not see sender side.
• Receiver behavior identical in both cases! Something’s (very) wrong!
Q: what relationship between sequence number size and window size to avoid problem in (b)?
Transport Layer

• Multiplexing and Demultiplexing
• Connectionless Transport: UDP
• Reliable Data Transfer

✓ Connection-oriented Transport: TCP
  ✓ Segment Structure
  • Reliable Data Transfer
  • Flow Control
  • Connection Management

• Congestion Control
• TCP Congestion Control

**TCP: Overview (RFCs 793, 1122, 1323, 2018, 2581)**

- **Point-to-point**: one sender, one receiver
- **Reliable, in-order byte stream**: no message boundaries
- **Pipelined**: TCP congestion and flow control set window size
- **Full duplex data**:
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- **Connection-oriented**: Handshaking (exchange of control messages) initiates sender, receiver state before data exchange
- **Flow controlled**: Sender will not overwhelm receiver
### TCP Segment Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URG</td>
<td>Urgent data (generally not used)</td>
</tr>
<tr>
<td>ACK</td>
<td>ACK # valid</td>
</tr>
<tr>
<td>PSH</td>
<td>Push data now (generally not used)</td>
</tr>
<tr>
<td>RST, SYN, FIN</td>
<td>Connection estab (setup, teardown commands)</td>
</tr>
<tr>
<td>options (var)</td>
<td>Options variable length</td>
</tr>
<tr>
<td>application data (var)</td>
<td>Application data (variable length)</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>source port #</td>
<td>Source 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>Head length</td>
</tr>
<tr>
<td>UAPR</td>
<td>Urg available</td>
</tr>
<tr>
<td>SF</td>
<td>Syn flag</td>
</tr>
<tr>
<td>receive window</td>
<td>Window size</td>
</tr>
</tbody>
</table>

**Counting**
- By bytes of data (not segments!)
- # bytes rcvr willing to accept
TCP Sequence Numbers and ACKs

**Sequence numbers:**
- Byte stream **number** of first byte in segment’s data

**Acknowledgements:**
- Sequence number of next byte expected from other side
- Cumulative ACK

**Q:** How receiver handles out-of-order segments
- **A:** TCP spec doesn’t say, - up to implementor

TCP Sequence Numbers and ACKs

TCP RTT & Timeout

Q: How to set TCP timeout value?
• Longer than RTT
  • But RTT varies
• Too short: premature timeout, unnecessary retransmissions
• Too long: slow reaction to segment loss

Q: How to estimate RTT?
• SampleRTT: Measured time from segment transmission until ACK receipt
  • Ignore retransmissions
• SampleRTT will vary, want estimated RTT smoother
  • Average several recent measurements, not just current SampleRTT
TCP RTT & Timeout

\[
\text{EstimatedRTT} = (1-\alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}
\]

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: \( \alpha = 0.125 \)
TCP RTT & Timeout

• **Timeout interval**: EstimatedRTT plus safety margin

  large variation in EstimatedRTT → larger safety margin.

• Estimate SampleRTT deviation from EstimatedRTT:

  \[
  \text{DevRTT} = (1-\beta)\times\text{DevRTT} + \\
  \beta\times|\text{SampleRTT} - \text{EstimatedRTT}|
  \]

  (typically, \(\beta = 0.25\))

  \[
  \text{TimeoutInterval} = \text{EstimatedRTT} + 4\times\text{DevRTT}
  \]

Transport Layer

- Multiplexing and Demultiplexing
- Connectionless Transport: UDP
- Reliable Data Transfer
  ✓ Connection-oriented Transport: TCP
    - Segment Structure
      ✓ Reliable Data Transfer
      - Flow Control
      - Connection Management
- Congestion Control
- TCP Congestion Control
TCP Reliable Data Transfer

• TCP creates RDT service on top of IP’s unreliable service
  • Pipelined segments
  • Cumulative acks
  • Single retransmission timer

• Retransmissions triggered by:
  • Timeout events
  • Duplicate acks
TCP Reliable Data Transfer

• Let’s initially consider simplified TCP sender
  • Ignore duplicate acks
  • Ignore flow control, congestion control
TCP Sender

- **Data received from application**
  - Create segment with sequence number
  - Sequence number is byte-stream number of first data byte in segment
  - Start timer if not already running
    - Think of timer as for oldest unACKed segment
    - Expiration interval: `TimeOutInterval`

- **Timeout**
  - Retransmit segment that caused timeout
  - Restart timer

- **ACK received**
  - If ACK acknowledges previously un-ACKed segments
    - Update what is known to be ACKed
    - Start timer if there are still un-acked segments
Simplified TCP Sender

- \( \text{NextSeqNum} = \text{InitialSeqNum} \)
- \( \text{SendBase} = \text{InitialSeqNum} \)

wait for event

- data received from application above
- create segment, seq. #: NextSeqNum
- pass segment to IP (i.e., “send”)
- NextSeqNum = NextSeqNum + length(data)
- if (timer currently not running)
  - start timer

timeout
- retransmit not-yet-acked segment with smallest seq. #
- start timer

- ACK received, with ACK field value \( y \)
  - if \( y > \text{SendBase} \) {
    - \( \text{SendBase} = y \)
    - /* SendBase–1: last cumulatively ACKed byte */
    - if (there are currently not-yet-acked segments)
      - start timer
      - else stop timer
  }
TCP Retransmission Scenarios

- **Lost ACK Scenario**: Host A sends a packet with sequence number 92, 8 bytes of data, and the acknowledgment (ACK) is 100. The acknowledgment is lost, leading to a timeout at Host B.

- **Premature Timeout**: Host A sends a packet with sequence number 92, 8 bytes of data, and the acknowledgment (ACK) is 100. The acknowledgment is lost, causing a timeout at Host B. Host A then sends a new packet with sequence number 100, 20 bytes of data, and the acknowledgment (ACK) is 120. Another acknowledgment is lost, causing another timeout at Host B.

- **Cumulative ACK**: Host A sends a packet with sequence number 92, 8 bytes of data, and the acknowledgment (ACK) is 100. The acknowledgment is lost, causing a timeout at Host B. Host A then sends a new packet with sequence number 100, 20 bytes of data, and the acknowledgment (ACK) is 120. Another acknowledgment is lost, causing another timeout at Host B. Host A then sends a new packet with sequence number 120, 15 bytes of data, and the acknowledgment (ACK) is 120.
### TCP ACK Generation (RFC 1122, RFC 2581)

<table>
<thead>
<tr>
<th>Event at receiver</th>
<th>TCP receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected sequence number. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>arrival of in-order segment with expected sequence number. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>arrival of out-of-order segment higher-than-expect Sequence #. Gap detected</td>
<td>Immediately send duplicate ACK, indicating Sequence # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
TCP Fast Retransmit

• Time-out period often relatively long
  • Long delay before resending lost packet

• Detect lost segments via duplicate ACKs.
  • Sender often sends many segments back-to-back
  • If segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

• If sender receives 3 ACKs for same data (Triple Duplicate ACKs), resend un-ACKed segment with smallest sequence number
  • Likely that un-ACKed segment lost, so don’t wait for timeout
TCP Fast Retransmit

[Diagram showing TCP packet transmission with fast retransmit after triple duplicate ACK]

Host A
Seq=92, 8 bytes of data
Seq=100, 20 bytes of data
ACK=100
ACK=100
ACK=100
ACK=100
Seq=100, 20 bytes of data

Host B

timeout

fast retransmit after sender receipt of triple duplicate ACK
Transport Layer

- Multiplexing and Demultiplexing
- Connectionless Transport: UDP
- Reliable Data Transfer
  - Connection-oriented Transport: TCP
    - Segment Structure
    - Reliable Data Transfer
      - Flow Control
        - Connection Management
  - Congestion Control
  - TCP Congestion Control
TCP Flow Control

Flow control

- Receiver controls sender
- Sender will not overflow receiver’s buffer by transmitting too much, too fast
TCP Flow Control

- Receiver advertises free buffer space by including $rwnd$ value in TCP header of receiver-to-sender segments
  - $RcvBuffer$ size set via socket options (typical default is 4096 bytes)
  - Many operating systems auto-adjust $RcvBuffer$

- Sender limits amount of un-ACKed (in-flight) data to receiver’s $rwnd$ value

- Guarantees receive buffer will not overflow
Transport Layer

- Multiplexing and Demultiplexing
- Connectionless Transport: UDP
- Reliable Data Transfer

✓ Connection-oriented Transport: TCP
  - Segment Structure
  - Reliable Data Transfer
  - Flow Control
  ✓ Connection Management

- Congestion Control
- TCP Congestion Control
Connection Management

Before exchanging data, sender and receiver **handshake**

- Agree to establish connection
  (each knowing the other willing to establish connection)
- Agree on connection parameters

```java
Socket clientSocket = newSocket("hostname","port number");

Socket connectionSocket = welcomeSocket.accept();
```
Agree to Establish a Connection

Q: will two-way handshake always work in network?

• Variable delays

• Retransmitted messages due to message loss

• Message reordering

• Can not see other side
Agreeing To Establish a Connection

• Two-way handshake failure scenarios
TCP Three-Way Handshake

**client state**
- LISTEN
- SYNSENT
- ESTAB

**server state**
- LISTEN
- SYN RCVD
- ESTAB

**Client Initialization**
- Choose initial sequence number, $x$
- Send TCP SYN message

**Server Response**
- Choose initial sequence number, $y$
- Send TCP SYN-ACK message, acknowledging SYN

**Client Acknowledgment**
- ACK bit set, ACK number set to $x + 1$

**Server Acknowledgment**
- ACK bit set, ACK number set to $y + 1$

**Client Status**
- Indicates server is alive; sends ACK for SYN-ACK;
  - This segment may contain client-to-server data

**Server Status**
- Indicates client is alive
TCP Three-Way Handshake: FSM

Socket connectionSocket = welcomeSocket.accept();
SYN(x)
SYNACK(seq=y,ACKnum=x+1)
create new socket for communication back to client

Socket clientSocket = newSocket("hostname","port number");
SYN(seq=x)

SYN rcvd
ACK(ACKnum=y+1)
Λ

SYN sent
SYNACK(seq=y,ACKnum=x+1)
ACK(ACKnum=y+1)
TCP: Closing A Connection

• Client and server each close their side of connection
  • Send TCP segment with FIN bit = 1

• Respond to received FIN with ACK
  • On receiving FIN, ACK can be combined with own FIN

• Simultaneous FIN exchanges can be handled
TCP: Closing A Connection

[Diagram of TCP connection states: ESTAB, FIN_WAIT_1, FIN_WAIT_2, TIMED_WAIT, CLOSED, clientSocket.close(), client state, server state, transition rules for FIN bit and ACK number updates.

Acknowledgements

• The following materials have been used in preparation of this slide set:

   7th Edition
   James Kurose, Keith Ross
   Pearson
   2016

   5th Edition
   Larry Peterson, Bruce Davie
   Morgan Kaufmann
   2011