Transport Layer


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

• Logical communication between processes

• Internet Transport-Layer Processes
  • TCP
    • Connection Setup
    • Flow Control
    • Congestion Control
  • UDP
    • Best Effort IP

• Services not supported
  • Delay Guarantee
  • Bandwidth Guarantees
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
Multiplexing & Demultiplexing

• Multiplex
  • Performed at sender
  • Handle data from multiple sockets and add transport header

• Demultiplex
  • Performed at receiver
  • Use header information to deliver received segments to correct socket
Demultiplexing

• Host receives IP datagrams

• Each datagram
  • Source IP address

• Destination IP address

• Carries one Transport-Layer Segment
  Segment has
  • Source port number
  • Destination port number

• **IP Address + Port number** used to direct segment to appropriate socket
Demultiplexing

• Connectionless
  • Information from datagrams used for Demux
    • Destination IP
    • Destination port number

• Connection-Oriented
  • Information from datagrams used for Demux
    • Source IP
    • Destination IP
    • Source port number
    • Destination port number
Connectionless Demux

- Created socket has host-local port number
- IP datagrams with same destination port number, but different source IP addresses and/or source port numbers will be directed to same socket at destination
Connection-Oriented Demux

- **TCP socket** identified by 4-tuple
  - Source IP address
  - Source port number
  - Destination IP address
  - Destination port number

  Receiver uses all four values to de-mux and 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

three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to different sockets
Connection-Oriented Demux

[Diagram of network protocols and IP addressing]
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
**UDP**

- **UDP**: User Datagram Protocol
  - Best effort
    - Loss
    - Out of order delivery
  - Connectionless
    - No handshaking between UDP sender and receiver
    - Each UDP segment handled independently of others

- Reliable Transfer over UDP is possible
  - Add reliability at application layer
  - Application-specific error recovery
UDP

• UDP Usage
  • Why?
    • No connection establishment
    • Simple: No connection state at sender and receiver
    • Small header size
    • No congestion control: Can be used as fast as desired

• Where?
  • Streaming multimedia
    Loss tolerant & rate sensitive traffic
  • DNS
  • SNMP
UDP

• Checksum
  • Sender
    • Treat segment contents, including header fields, as sequence of 16-bit integers
    • Checksum: Addition (one’s compliment 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 errors detected!
UDP

Example: add two 16-bit integers

\[
\begin{array}{c}
\text{111001100110} \\
\text{110101010101} \\
\hline
\text{11011101110111} \\
\end{array}
\]

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
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
Reliable Data Transfer

• Reliable transfer of information important for many applications

• Reliable transfer in application, transport, and link layers in top ten important networking topics.

• Characteristics of unreliable channel will determine the complexity of data transfer protocol.
Reliable Data Transfer

`rdt_send()` : called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

`deliver_data()` : called by `rdt` to deliver data to upper

`udt_send()` : called by `rdt`, to transfer packet over unreliable channel to receiver

`rdt_rcv()` : called when packet arrives on rcv-side of channel
Reliable Data Transfer

To understand the needs of reliable data transfer, we will examine:

- Incrementally develop sender & receiver sides of what we need for reliable data transfer
- Consider only unidirectional data transfer but control info will flow on both directions
- Use finite state machines (FSM) to specify sender & receiver
RDT1.0: Reliable Data Transfer over a Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets

- Separate FSMs for sender and receiver:
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel

Sender

- Wait for call from above
  - `rdt_send(data)`
  - `packet=make_pkt(data)`
  - `udt_send(packet)`

Receiver

- Wait for call from below
  - `rdt_rcv(packet)`
  - `extract(packet,data)`
  - `deliver_data(data)`
RDT2.0: Reliable Data Transfer over a Channel with Bit Errors

- Underlying channel unreliable
  - Channel may flip bits in packet
  - Checksum to detect bit errors

- **The question**: How to recover from errors?
  - **Tip**: How do humans recover from error during conversation?
    - Error Detection
    - Feedback
RDT2.0: Reliable Data Transfer over Channel with Bit Errors

- Reliable Data Transfer Mechanisms
  - Detect
    - **Checksum**: Bit errors
  - Feedback
    - **Acknowledgements (ACKs)**: receiver explicitly tells sender that packet received OK
    - **Negative acknowledgements (NAKs)**: receiver explicitly tells sender that packet had errors
  - Correction
    - Sender retransmits packet on receipt of NAK
RDT2.0: FSM Specification

**Sender**

- `rdt_send(data)`
- `sndpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`

**States:**
- Wait for call from above
- Wait for ACK or NAK

- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(sndpkt)`

**Actions:**
- `Lambda`  

**Receiver**

- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`
- `udt_send(NAK)`

**States:**
- Wait for call from below
- Wait for call from above

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

RDT2.0: Operation with No Errors

```
rdt_send(data)

snkpkt = make_pkt(data, checksum)
udt_send(snkpkt)

Wait for call from above

rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(snkpkt)

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isACK(rcvpkt)
```
RDT2.0: Error Scenario
RDT2.0: Fatal Flaw!

- What happens if ACK/NAK corrupted?
  - Sender does not know what happened at receiver!
  - Can not just retransmit: possible duplicate

- Stop and wait
  - Sender sends one packet, then waits for receiver response

- Handling duplicates
  - Sender retransmits current packet if ACK/NAK corrupted
  - Sender adds sequence number to each packet
  - Receiver discards (does not deliver up) duplicate packet
RDT2.1: Sender (Garbled ACK/NAKs)
RDT2.1: Discussions

• **Sender**
  • Sequence number added to packet
  • Two sequence numbers (0,1) will suffice. **Why?**
  • Must check if received ACK/NAK corrupted
  • Twice as many states
    • State must **remember** whether **expected** packet should have sequence number of 0 or 1

• **Receiver**
  • Must check if received packet is duplicate
    • State indicates whether 0 or 1 is expected packet sequence number
    • Note: receiver cannot know if its last ACK/NAK received OK at sender
**RDT2.2: A NAK-Free Protocol**

- Same functionality as RDT2.1 using **ACKs only**

- Instead of NAK, receiver sends ACK for last packet received OK
  - Receiver must **explicitly** include sequence number of packet being ACKed

- Duplicate ACK at sender results in same action as NAK: **Retransmit Current Packet**
RDT2.2: Sender & Receiver

sender FSM fragment

\[
\text{rdt\_send(data)} \\
\text{sndpkt = make\_pkt(0, data, checksum)} \\
\text{udt\_send(sndpkt)} \\
\]

```
wait for call 0 from above
```

```
rdt\_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has\_seq1(rcvpkt))
udt\_send(sndpkt)
```

receiver FSM fragment

```
wait for 0 from below
```

```
rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has\_seq1(rcvpkt)
extract(rcvpkt, data)
deliver\_data(data)
sndpkt = make\_pkt(ACK1, chksum)
udt\_send(sndpkt)
```

RDT3.0: Channels with Errors & Loss

- **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 handles this
  - Receiver must specify sequence number of packets being ACKed

- Requires countdown timer
RDT3.0 Sender
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 pkt0
send ACK0
rcv pkt0
send ACK0

(b) Packet loss

Sender
send pkt0
rcv ACK0
send pkt1
timeout resend pkt1
rcv pkt1
send ACK1
rcv ACK1
send pkt0

Receiver
Pkt0
ACK0
Pkt1
ACK1
Pkt0
ACK0
rcv pkt0
send ACK0
rcv pkt0
send ACK0
RDT3.0 in action

(c) ACK loss

Sender

rcv ACK0
send pkt1

Receiver

rcv pkt1
send ACK1

(timeout) X

rcv packet (detect duplicate)
send ACK1

rcv pkt0
send ACK0

(d) Premature timeout/ delayed ACK

Sender

rcv packet
send pkt1

Receiver

rcv packet
send packet

timeout

rcv packet
send packet

rcv packet
send packet

rcv packet
send packet
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

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

- If RTT = 30 msec, 1KB packet every 30 msec: 33kB/sec throughput over 1 Gbps link
- Network protocol limits use of physical resources!
RDT 3.0 Stop and Wait Operation

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

\[ U_{\text{sender}} = \frac{3L}{R} \cdot \frac{R}{RTT + \frac{L}{R}} = \frac{0.0024}{30.008} = 0.00081 \]

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

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

- **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

\[
\begin{align*}
\text{rdt}_\text{rcv}(rcvpkt) \\
&\quad \& \text{notcorrupt}(rcvpkt) \\
&\quad \& \text{hasseqnum}(rcvpkt, \text{expectedseqnum}) \\
\end{align*}
\]

extract(rcvpkt, data)
deliver_data(data)
sndpkt=make_pkt(expectedseqnum, ACK, checksum)
udt_send(sndpkt)
expectedseqnum++

\[
\begin{align*}
\text{udt}_\text{send}(\text{sndpkt}) \\
\end{align*}
\]

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)

receive pkt0, send ack0
receive pkt1, send ack1

rcv ack0, send pkt4
rcv ack1, send pkt5

receive pkt3, discard, (re)send ack1
receive pkt4, discard, (re)send ack1
receive pkt5, discard, (re)send ack1

ignore duplicate ACK

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
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 \([\text{rcvbase}, \text{rcvbase}+N-1]\)

- 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 \([\text{rcvbase}-N, \text{rcvbase}-1]\)

- ACK\((n)\)

Otherwise

- Ignore
Selective Repeat In Action

**sender window (N=4)**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**sender**

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

**receiver**

- receive pkt0, send ack0
- receive pkt1, send ack1
- receive pkt3, buffer, send ack3
- receive pkt4, buffer, send ack4
- receive pkt5, buffer, send ack5
- receive 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)?
Summary

• Multiplexing and Demultiplexing

• Connectionless Transport : UDP

• Reliable Data Transfer
Next: TCP

• 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
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