Transport: UDP and TCP

EECS 489 Computer Networks
http://www.eecs.umich.edu/~zmao/eecs489

Z. Morley Mao
Thursday Sept 30, 2004

Acknowledgement: Some slides taken from Kurose&Ross and Katz&Stoica
Two general approaches
- Bottom-up
- Top-down
Bottom-up

- Classic approach – start from the physical layer all the way up to the application layer
- Advantages – allow “natural” explanation of protocols, algorithms; lower layers provide assumptions for higher layers
- Disadvantages – harder to justify the goals, i.e., what are the algorithms/protocols good for
Top-down

- New approach (E.g., Kurose & Ross) – start from the application layer all the way down to the physical layer
- Advantages – goals are very clear → start from application needs
- Disadvantages – harder to understand some assumptions made about lower layers (e.g., packet losses in the Internet are because congestion)
Mixed

- Our approach
- Advantages: hope to combine the advantages of both “Bottom-up” and “Top-down” approaches
- Disadvantage: more confusing
Outline

- Motivation
  - Transport layer
  - TCP
  - UDP
Motivation

- IP provides a weak, but efficient service model (*best-effort*)
  - Packets can be delayed, dropped, reordered, duplicated
  - Packets have limited size (why?)
- IP packets are addressed to a host
  - How to decide which application gets which packets?
- How should hosts send into the network?
  - Too fast is bad; too slow is not efficient
Outline

- Motivation
  - Transport layer
    - TCP
    - UDP
Transport Layer

- Can provide more reliability, in order delivery, at most once delivery
- Supports messages of arbitrary length
- Provide a way to decide which packets go to which applications (*multiplexing/demultiplexing*)
- Govern when hosts should send data → can implement congestion and flow control
Congestion & Flow Control

- Flow Control – avoid overflowing the receiver
- Congestion Control – avoid congesting the network

- What is network congestion?
Transport Layer (cont’d)

UDP: Not reliable
TCP: Ordered, reliable, well-paced
Ports

- Need to decide which application gets which packets
- Solution: map each socket to a *port*
- Client must know server’s port
- Separate 16-bit port address space for UDP and TCP
  - \((\text{src	extunderscore IP}, \text{src	extunderscore port}, \text{dst	extunderscore IP}, \text{dst	extunderscore port})\) uniquely identifies TCP connection
- *Well known ports* (0-1023): everyone agrees which services run on these ports
  - e.g., ssh:22, http:80
  - on UNIX, must be root to gain access to these ports (why?)
- *Ephemeral ports* (most 1024-65535): given to clients
  - e.g. chat client gets one of these
Headers

- IP header → used for IP routing, fragmentation, error detection… (we study that when we explore IP)
- UDP header → used for multiplexing/demultiplexing, error detection
- TCP header → used for multiplexing/demultiplexing, flow and congestion control
Outline

- Motivation
- Transport Layer
  - UDP
- TCP
UDP

- User Datagram Protocol
- Minimalist transport protocol
- Same best-effort service model as IP
- Messages up to 64KB
- Provides multiplexing/demultiplexing to IP
- Does not provide flow and congestion control
- Application examples: video/audio streaming
**UDP Service & Header**

- **Service:**
  - Send datagram from (IPA, Port 1) to (IPb, Port 2)
  - Service is unreliable, but error detection possible

- **Header:**

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP length</td>
<td>UDP checksum</td>
</tr>
<tr>
<td>Payload (variable)</td>
<td></td>
</tr>
</tbody>
</table>

- UDP length is UDP packet length (including UDP header and payload, but not IP header)
- Optional UDP checksum is over UDP packet
  - Why have UDP checksum in addition to IP checksum?
  - Why not have just the UDP checksum?
  - Why is the UDP checksum optional?
Outline

- Motivation
- Transport Layer
  - UDP
  - TCP
TCP

- Transmission Control Protocol
- Reliable, in-order, and at most once delivery
- Messages can be of arbitrary length
- Provides multiplexing/demultiplexing to IP
- Provides congestion control and avoidance
- Application examples: file transfer, chat
TCP Service

1) Open connection

2) Reliable byte stream transfer from (IPa, TCP Port1) to (IPb, TCP Port2)
   • Indication if connection fails: Reset

3) Close connection
Timing Diagram

3-way handshake

SYN k

SYN n; ACK k+1

DATA k+1; ACK n+1

ACK k+n+1

data exchange

FIN

FIN ACK

½ close

FIN

FIN ACK

½ close

Open connect.

Transfer

Close connect.
Open Connection: 3-Way Handshaking

- Goal: agree on a set of parameters: the start sequence number for each side
  - Starting sequence numbers are random.

Client (initiator)

<table>
<thead>
<tr>
<th>Active Open</th>
<th>connect()</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN, SeqNum = x</td>
<td></td>
</tr>
<tr>
<td>SYN and ACK, SeqNum = y and Ack = x + 1</td>
<td></td>
</tr>
<tr>
<td>ACK, Ack = y + 1</td>
<td></td>
</tr>
</tbody>
</table>

Server

<table>
<thead>
<tr>
<th>Passive Open</th>
<th>listen()</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept()</td>
<td></td>
</tr>
<tr>
<td>allocate buffer space</td>
<td></td>
</tr>
</tbody>
</table>
3-Way Handshaking (cont’d)

- Three-way handshake adds 1 RTT delay
- Why?
  - Congestion control: SYN (40 byte) acts as cheap probe
  - Protects against delayed packets from other connection (would confuse receiver)
Close Connection (Two-Army Problem)

- **Goal**: both sides agree to close the connection
- **Two-army problem**:
  - “Two blue armies need to simultaneously attack the white army to win; otherwise they will be defeated. The blue army can communicate only across the area controlled by the white army which can intercept the messengers.”

- What is the solution?
Close Connection

- 4-ways tear down connection

- Avoid reincarnation
- Can retransmit FIN ACK if it is lost
Reliable Transfer

- Retransmit missing packets
  - Numbering of packets and ACKs

- Do this efficiently
  - Keep transmitting whenever possible
  - Detect missing ACKs and retransmit quickly

- Two schemes
  - Stop & Wait
  - Sliding Window (Go-back-n and Selective Repeat)
Stop & Wait

- Send; wait for ack
- If timeout, retransmit; else repeat

Inefficient if \( TRANS << RTT \)
**Sliding Window**

- *window* = set of adjacent sequence numbers
- The size of the set is the *window size*

- Assume window size is $n$

- Let $A$ be the last ack’d packet of sender without gap; then window of sender = \{A+1, A+2, \ldots, A+n\}

- Sender can send packets in its window

- Let $B$ be the last received packet without gap by receiver, then window of receiver = \{B+1, \ldots, B+n\}

- Receiver can accept out of sequence, if in window
Go-Back-n (GBN)

- Transmit up to $n$ unacknowledged packets
- If timeout for $\text{ACK}(k)$, retransmit $k$, $k+1$, …
GBN Example

n = 9 packets in one RTT instead of 1

→ Fully efficient
GBN Example with Errors

Window size = 3 packets

Timeout Packet 5

Sender

1
2
3
4
5
6
7

Receiver

Mao F04

30
Selective Repeat (SR)

- Sender: transmit up to \( n \) unacknowledged packets
- Receiver: indicate packet \( k \) is missing
- Sender: retransmit packet \( k \)
SR Example with Errors

Window size = 3 packets

nack = 5
Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough. Throughput is $\sim (n/\text{RTT})$
  - Stop & Wait is like $n = 1$.

- Sender has to buffer all unacknowledged packets, because they may require retransmission

- Receiver may be able to accept out-of-order packets, but only up to its buffer limits
Setting Timers

- The sender needs to set retransmission timers in order to know when to retransmit a packet that may have been lost.

- **How long to set the timer for?**
  - **Too short**: may retransmit before data or ACK has arrived, creating duplicates.
  - **Too long**: if a packet is lost, will take a long time to recover (inefficient).
Timing Illustration

Timeout too long → inefficiency

Timeout too short → duplicate packets
Adaptive Timers

- The amount of time the sender should wait is about the round-trip time (RTT) between the sender and receiver
- For link-layer networks (LANs), this value is essentially known
- For multi-hop WANS, rarely known
- Must work in both environments, so protocol should adapt to the path behavior
- Measure successive ack delays $T(n)$
  Set timeout = average + 4 deviations
Timer Algorithm

- Use exponential averaging:

\[
\begin{align*}
A(n) &= b \times A(n-1) + (1 - b) \times T(n) \\
D(n) &= b \times D(n-1) + (1 - b) \times (T(n) - A(n)) \\
\text{Timeout}(n) &= A(n) + 4 \times D(n)
\end{align*}
\]

Notes:
1. Measure \(T(n)\) only for original transmissions
2. Double Timeout after timeout …
   Justification: timeout indicates likely congestion;
   Further retransmissions would make things worse
3. Reset Timeout = \(A + 4D\) for new packet and when receive ACK
TCP Header

- Sequence number, acknowledgement, and advertised window – used by sliding-window based flow control
- Flags:
  - SYN, FIN – establishing/terminating a TCP connection
  - ACK – set when Acknowledgement field is valid
  - URG – urgent data; Urgent Pointer says where non-urgent data starts
  - PUSH – don’t wait to fill segment
  - RESET – abort connection

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>10</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Destination port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence number</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acknowledgement</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
<td>Advertised window</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- UDP: Multiplex, detect errors
- TCP: Reliable Byte Stream
  - 3-way handshaking
  - Reliable transmissions: ACKs…
  - S&W not efficient $\rightarrow$ Go-Back-n
  - What to ACK? (cumulative, …)
  - Timer Value: based on measured RTT