Internet Transport Protocols

UDP / TCP

Prof. Anja Feldmann, Ph.D.
anja@net.t-labs.tu-berlin.de

TCP/IP Illustrated, Volume 1, W. Richard Stevens
http://www.kohala.com/start
Transport Layer: Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management
- Principles of congestion control
- TCP congestion control
Internet Transport-Layer Protocols

- **Network layer**: Logical communication between hosts
- **Transport layer**: Logical communication between processes
  - Relies on, enhances network layer services
- More than one transport protocol available to apps
  - Internet:
    - TCP
    - UDP
Sockets: interface to applications

Socket API
- Introduced in BSD4.1 UNIX, 1981
- Explicitly created, used, released by apps
- Client/server paradigm
- Two types of transport service via socket API:
  - Unreliable datagram
  - Reliable, byte stream-oriented

A host-local, application-created/owned, OS-controlled interface (a “door”) into which an application process can both send and receive messages to/from another (remote or local) application process
Sockets and OS

**Socket:** a door between application process and end-end-transport protocol (UDP or TCP)
Multiplexing/Demultiplexing

Demultiplexing at rcv host:
Delivering received segments to correct application (socket)

Multiplexing at send host:
Gathering data from multiple appl. (sockets), enveloping data with header (later used for demultiplexing)

= socket  = process

<table>
<thead>
<tr>
<th>application</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td></td>
</tr>
<tr>
<td>network</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td>physical</td>
<td></td>
</tr>
</tbody>
</table>

host 1

<table>
<thead>
<tr>
<th>P1</th>
<th>application</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>transport</td>
<td></td>
</tr>
<tr>
<td></td>
<td>network</td>
<td></td>
</tr>
<tr>
<td></td>
<td>link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>physical</td>
<td></td>
</tr>
</tbody>
</table>

host 2

<table>
<thead>
<tr>
<th>P4</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>transport</td>
</tr>
<tr>
<td></td>
<td>network</td>
</tr>
<tr>
<td></td>
<td>link</td>
</tr>
<tr>
<td></td>
<td>physical</td>
</tr>
</tbody>
</table>

host 3
Multiplexing/Demultiplexing

Multiplexing/demultiplexing:
- Based on sender, receiver port numbers, IP addresses
  - Source, dest port #s in each segment
  - Well-known port numbers for specific applications (see /etc/services)

TCP/UDP segment format

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>other header fields</td>
<td></td>
</tr>
<tr>
<td>application data (message)</td>
<td></td>
</tr>
<tr>
<td>32 bits</td>
<td></td>
</tr>
</tbody>
</table>
### Multiplexing/Demultiplexing: Examples

**Port use: simple telnet app**

- **Source IP:** C  
  **Dest IP:** B  
  **source port:** x  
  **dest. port:** 80

**WWW client**

- **Source IP:** A  
  **Dest IP:** B  
  **source port:** x  
  **dest. port:** 80

**WWW server B**

- **Source IP:** C  
  **Dest IP:** B  
  **source port:** y  
  **dest. port:** 80

**WWW client**

- **Source IP:** A  
  **Dest IP:** B  
  **source port:** 23  
  **dest. port:** 80

**WWW server B**

- **Source IP:** C  
  **Dest IP:** B  
  **source port:** x  
  **dest. port:** 80

**Port use: WWW server**
UDP: User Datagram Protocol [RFC 768]

- “Bare bones” Internet transport protocol
- “Best effort” service, UDP segments may be:
  - Lost
  - Delivered out of order to application
- **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: More

- Each user request transferred in a single datagram
- UDP has a receive buffer but no sender buffer
- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses (why?):
  - DNS, SNMP, NFS
- Reliable transfer over UDP: add reliability at application layer

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>

Application data (message)
UDP Checksum

- Ones-complement of 16 bit words
- Covers data plus a 12 byte pseudo header
  - IP addresses, 0, protocol identifier (8 bits), length (16 bits)
  - Ensures that packet has reached the correct host
- Pad byte in case of an odd packet length
- Optional: checksum=0 indicates no checksum
  - Should always be enabled
- Receiver has to verify checksum
TCP: Overview

- 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 one connection

- MSS: maximum segment size

- Connection-oriented:
  - Handshaking (exchange of control msgs) init’s sender, receiver state before data exchange

- Flow controlled:
  - Sender will not overwhelm receiver

- Congestion controlled:
  - Sender will not overwhelm the network
Simulating Transport Protocols

- Network simulator
- Examples:
  - Network Simulator (NS), SSFNet, ...

- Animation of NS traces via NAM (Network Animator)
- Try it!
Simulating Transport Protocols

- Example: 2 TCP connections + 1 UDP flow
- Topology:

  TCP 1
  \[ 2 \text{Mb}  \]
  \[ 25 \text{ms}  \]

  TCP 2
  \[ 380 \text{Kb}  \]
  \[ 10 \text{ms}  \]

  UDP

  TCP 2
  \[ 2 \text{Mb}  \]
  \[ 25 \text{ms}  \]

  TCP 1
  \[ 380 \text{Kb}  \]
  \[ 10 \text{ms}  \]

  UDP

- TCP1 starts at time 0 seconds, TCP2 at time 3 seconds
- UDP starts at time 15 seconds
Simulation Results
TCP Segment Structure

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Internet checksum**: (as in UDP)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dest port #</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequence number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>acknowledgement number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>head len</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rcvr window size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>checksum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ptr urgent data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>(variable length)</td>
<td></td>
</tr>
<tr>
<td>application data</td>
<td>(variable length)</td>
<td></td>
</tr>
</tbody>
</table>

- Counting by bytes of data (not segments!)
- # bytes rcvr willing to accept
TCP Reliability: Seq. #'s and ACKs

Seq. #'s:
- **Byte stream**
  "Number" of first byte in segment’s data

ACKs:
- Seq # of next byte expected from other side
- Cumulative ACK

Q: How receiver handles out-of-order segments?
- TCP spec doesn’t say, – up to implementer

Host A
- User types ‘C’
- Seq=42, ACK=79, data = ‘C’
- Seq=79, ACK=43, data = ‘C’
- Seq=43, ACK=80
time

Host B
- host ACKs receipt of ‘C’, echoes back ‘C’
- host ACKs receipt of echoed ‘C’

simple telnet scenario
TCP: Reliable Data Transfer

- **Packet loss detection:**
  - Retransmission timeout
  - Fast retransmit
    - Three duplicate ACKs

- **Simplified sender Assumption**
  - One way data transfer
  - No flow, no congestion control

- **Retransmission mechanisms**
  - ARQ: Go-Back-N, selected retransmissions
TCP: Retransmission Scenarios

Lost ACK scenario:
- Seq=92, 8 bytes data
- ACK=100
- Timeouts:
  - Host A: Seq=92 timeout
  - Host B: Seq=100, 20 bytes data timeout

Premature timeout, cumulative ACKs:
- Seq=92, 8 bytes data
- ACK=100
- Cumulative ACKs:
  - Host A: Seq=100 timeout
  - Host B: Seq=100, 20 bytes data timeout
# 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 seq #. All data up to expected seq # already ACKed</td>
<td><strong>Delayed ACK.</strong> Wait up to 500ms for next segment. If no next segment, send ACK (reduces ACK traffic)</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single <strong>cumulative ACK</strong>, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq. #. Gap detected</td>
<td>Immediately send <strong>duplicate ACK</strong>, indicating seq. # of next expected byte (trigger fast retransmit)</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 Retransmission Timeout

- TCP uses one timer for one pkt only
- Retransmission Timeout (RTO) calculated dynamically
  - Based on Round Trip Time estimation (RTT)
  - Wait at least one RTT before retransmitting
  - Importance of accurate RTT estimators:
    - Low RTT $\rightarrow$ unneeded retransmissions
    - High RTT $\rightarrow$ poor throughput
  - RTT estimator must adapt to change in RTT
    - But not too fast, or too slow!
  - Spurious timeouts
    - “Conservation of packets” principle – more than a window worth of packets in flight
Retransmission Timeout Estimator

- Round trip times exponentially averaged:
  - New RTT = $\alpha$ (old RTT) + (1 - $\alpha$) (new sample)
  - $\alpha$ = 0.875 for most TCPs
- Retransmit timer set to $\beta$ RTT, where $\beta$ = 2
  - Every time timer expires, RTO exponentially backed-off
- Key observation: At high loads round trip variance is high
- Solution (currently in use):
  - Base RTO on RTT and standard deviation of RTT:
    RTT + 4 * rttvar
  - rttvar = $\chi$ * dev + (1 - $\chi$)rttvar
    - dev = linear deviation (also referred to as mean deviation)
    - $\chi$ = 0.25 for most TCPs
      - Inappropriately named – actually smoothed linear deviation
  - RTO is discretized into ticks of 500ms (RTO $\geq$ 2ticks)
Retransmission Ambiguity

- Karn’s RTT Estimator
  - If a segment has been retransmitted:
  - Don’t count RTT sample on ACKs for this segment
  - Keep backed off time-out for next packet
  - Reuse RTT estimate only after one successful transmission
TCP Flow Control: Sliding Window Protocol

Receiver: Explicitly informs sender of (dynamically changing) amount of free buffer space
- **rcvr window size** field in TCP segment

Sender: Amount of transmitted, unACKed data less than most recently-receiver **rcvr window size**
TCP Flow Control

- TCP is a sliding window protocol
  - For window size $n$, can send up to $n$ bytes without receiving an acknowledgement
  - When the data is acknowledged, the window slides forward

- Original TCP always sent entire window
  - Congestion control now limits this via congestion window determined by the sender! (network limited)
  - If not, data rate is receiver limited

- Silly window syndrome
  - Too many small packets in flight
  - Limit the # of smaller pkts than MSS to one per RTT
Window Flow Control:

Sender Side

- Sent and acknowledged
- Sent but not acknowledged
- Not yet sent
- Next to be sent

Receiver Side

- Acked but not delivered to user
- Not yet acknowledged

sender window

Receive buffer

rcvr window
Ideal Window Size

- Ideal size = delay * bandwidth
  - Delay-bandwidth product (RTT * bottleneck bitrate)
- Window size < delay*bw $\Rightarrow$ wasted bandwidth
- Window size > delay*bw $\Rightarrow$
  - Queuing at intermediate routers $\Rightarrow$ increased RTT
  - Eventually packet loss
TCP Connection Management

**Recall:** TCP sender, receiver establish “connection” before exchanging data segments

- Initialize TCP variables:
  - Seq. #s
  - Buffers, flow control info (e.g. `RcvWindow`)
  - MSS and other options

- **Client:** connection initiator, **server:** contacted by client

- Three-way handshake
  - Simultaneous open

- TCP Half-Close (four-way handshake)

- Connection aborts via RSTs
TCP Connection Management (2)

Three way handshake:

**Step 1:** Client end system sends TCP SYN control segment to server
- Specifies initial seq #
- Specifies initial window #

**Step 2:** Server end system receives SYN, replies with SYNACK control segment
- ACKs received SYN
- Allocates buffers
- Specifies server → receiver initial seq. #
- Specifies initial window #

**Step 3:** Client system receives SYNACK
TCP Connection Management (3)

Closing a connection:

Client closes socket:

```java
clientSocket.close();
```

**Step 1:** **Client** end system sends TCP FIN control segment to **server**

**Step 2:** **Server** receives FIN, replies with ACK. Closes connection, sends FIN.
TCP Connection Management (4)

**Step 3:** Client receives FIN, replies with ACK.

- Enters “timed wait” – will respond with ACK to received FINs

**Step 4:** Server, receives ACK. Connection closed.

**Note:** With small modification, can handle simultaneous FINs.
TCP Connection Management (5)

TCP client lifecycle

- **CLOSED**
  - client application initiates a TCP connection

- **SYN_SENT**
  - receive SYN & ACK
  - send ACK

- **ESTABLISHED**
  - client application initiates close connection

- **FIN_WAIT_1**
  - send FIN

- **FIN_WAIT_2**
  - receive FIN
  - send ACK

- **TIME_WAIT**
  - wait 30 seconds

- **FIN_WAIT_1**
  - receive ACK
  - send nothing

- **FIN_WAIT_2**
  - receive FIN
  - send ACK
TCP Connection Management (cont)

TCP server lifecycle

- **CLOSED**: server application creates a listen socket
  - receive ACK
  - send nothing
- **LISTEN**: receive SYN
  - send SYN & ACK
- **SYN_RCVD**: receive ACK
  - send nothing
- **ESTABLISHED**: receive FIN
  - send ACK
- **CLOSE_WAIT**: send FIN
  - receive FIN
- **LAST_ACK**: receive ACK
  - send nothing
Excursion: Congestion Control Principles
TCP Acknowledgement Clocking

- TCP is “self-clocking”
- New data sent when old data is acked
- Ensures an “equilibrium”
- But how to get started?
  - Slow Start
  - Congestion Avoidance
- Other TCP features
  - Fast Retransmission
  - Fast Recovery
TCP Congestion Control:

- "Probing" for usable bandwidth:
  - Ideally: Transmit as fast as possible (cwnd as large as possible) without loss
  - Increase cwnd until loss (congestion)
  - Loss: Decrease cwnd, then begin probing (increasing) again

- Two "phases"
  - Slow start
  - Congestion avoidance

- Important variables:
  - cwnd
  - threshold (ssthresh): Defines threshold between two slow start phase, congestion control phase
TCP Slowstart

- Exponential increase (per RTT) in window size (not so slow!)
- Loss event: Timeout (Tahoe TCP) and/or or three duplicate ACKs (Reno TCP)

**Slowstart algorithm**

initialize: $\text{cwnd} = 1$
for (each segment ACKed)
    $\text{cwnd}++$
until (loss event OR $\text{cwnd} > \text{threshold}$)
Congestion Avoidance

- Loss implies congestion – why?
  - Not necessarily true on all link types
- If loss occurs when cwnd = W
  - Network can handle 0.5W ~ W segments
  - Set cwnd to 0.5W (multiplicative decrease)
- Upon receiving new ACK
  - Increase cwnd by 1/cwnd
  - Results in additive increase
TCP Congestion Avoidance

/* slowstart is over */
/* cwnd> threshold */
until (loss event) {
    every cwnd segments ACKed:
        cwnd++
}
threshold = cwnd/2
cwnd = 1
perform slowstart

1: TCP Reno skips slowstart (fast recovery) after three duplicate ACKs
Return to Slow Start

- If packet is lost we lose self clocking
  - Need to implement slow-start and congestion avoidance together

- When timeout occurs
  - Set threshold to 0.5 cwnd
  - Set cwnd to one segment

- When three duplicate acks occur:
  - Set threshold to 0.5 cwnd
  - Retransmit missing segment == Fast Retransmit
  - cwnd = threshold + number of dupacks
  - Upon receiving acks cwnd = threshold (cut in half!)
  - Use congestion avoidance == Fast Recovery
TCP Congestion Control

- End-end control (no network assistance)
- TCP throughput limited by rcvr window (flow control)
- Transmission rate limited by congestion window size, cwnd, over segments:

  - \( w \) segments, each with MSS bytes sent in one RTT
Fast Recovery Example

- Fast Recovery Example

- cwnd = 6; in congestion avoidance
Sequence Number Plot (Simulation)
Seq. Number Plot (Simulation) zoom
TCP Flavors / Variants

- TCP Tahoe
  - Slow Start
  - Congestion Avoidance
  - Timeout, 3 duplicate acks $\rightarrow$ cwnd = 1 $\Rightarrow$ slow start

- TCP Reno
  - Slow-start
  - Congestion avoidance
  - Fast retransmit, Fast recovery
  - Timeout $\rightarrow$ cwnd = 1 $\Rightarrow$ slow start
  - Three duplicate acks $\rightarrow$ Fast Recovery, Congestion Avoidance
Extensions

- Fast recovery, multiple losses per RTT $\Rightarrow$ timeout

- TCP New-Reno
  - Stay in fast recovery until all packet losses in window are recovered
  - Can recover 1 packet loss per RTT without causing a timeout

- Selective Acknowledgements (SACK) [rfc2018]
  - Provides information about out-of-order packets received by receiver
  - Can recover multiple packet losses per RTT
Additional TCP Features

- Urgent Data
  - Nice for interactive applications
  - In-Band via urgent pointer

- Nagle algorithm
  - Avoidance of small segments
  - Needed for interactive applications
  - Methodology: only one outstanding packet can be small
Summary

- Reviewed principles of transport layer:
  - Reliable data transfer
  - Flow control
  - Congestion control
  - (Multiplexing)

- Instantiation in the Internet
  - UDP
  - TCP