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

socket

A host-local, application-created/owned, OS-controlled interface (a “door”) into which 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>
<th>transport</th>
<th>network</th>
<th>link</th>
<th>physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>host 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>application</th>
<th>P2</th>
<th>transport</th>
<th>network</th>
<th>link</th>
<th>physical</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>host 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P4</th>
<th>application</th>
<th>transport</th>
<th>network</th>
<th>link</th>
<th>physical</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>host 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>other header fields</td>
<td></td>
</tr>
<tr>
<td>application data (message)</td>
<td></td>
</tr>
</tbody>
</table>
Multiplexing/Demultiplexing: Examples

Port use: Simple telnet app

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: Usage

- 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>32 bits</td>
<td></td>
</tr>
</tbody>
</table>

Length, in bytes of UDP segment, including header

Application data (message)
UDP: Checksum

- Ones-complement of 16 bit words
- Covers data plus a 12 byte pseudo header
  - IP addresses, protocol identifier, length
  - 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

RFCs: 793, 1122, 1323, 2018, 2581
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:

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

![Graph showing simulation results with throughput in packets per second over time for TCP 1, TCP 2, and UDP.]
TCP Segment Structure

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>head len</td>
<td>not used</td>
</tr>
<tr>
<td>checksum</td>
<td>ptr urgent data</td>
</tr>
<tr>
<td>Options (variable length)</td>
<td></td>
</tr>
<tr>
<td>application data</td>
<td>(variable length)</td>
</tr>
</tbody>
</table>

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)

# bytes rcvr willing to accept
counting by bytes of data (not segments!)
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
- Seq=42, ACK=79, data = ‘C’
- User types ‘C’
- host ACKs receipt of echoed ‘C’

Host B
- Seq=79, ACK=43, data = ‘C’
- host ACKs receipt of ‘C’, echoes back ‘C’
- Seq=43, ACK=80

simple telnet scenario
TCP: Reliable Data Transfer

- Simplified sender
  - Assumption
    - One way data transfer
    - No flow, no congestion control
  - Packet loss detection:
    - Retransmission timeout
    - Fast retransmit
      - Three duplicate ACKs
- Retransmission mechanisms
  - ARQ: Go-Back-N, selected retransmissions
TCP: Retransmission Scenarios

- **Lost ACK scenario**: Host A sends Seq=92, 8 bytes data. Host B ACKs with ACK=100. Due to network loss, Host B does not receive the data and retransmits Seq=92, 8 bytes data. Host A receives the retransmitted data and ACKs with ACK=100 again. This process is repeated until the data is successfully received.

- **Premature timeout scenario**: Host A sends Seq=100, 20 bytes data. Host B ACKs with ACK=100 and then timeout occurs due to network congestion. Host A retransmits Seq=100, 20 bytes data. Host B receives the retransmitted data and sends ACK=120. This process is repeated until the data is successfully received.

**Cumulative ACKs**: In both scenarios, Host A uses cumulative ACKs to indicate that multiple segments have been successfully received together. This helps to avoid duplicate retransmissions and improves network efficiency.
## 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 = $\alpha \cdot \text{(old RTT)} + (1 - \alpha) \cdot \text{(new sample)}$
  - $\alpha = 0.875$ for most TCPs

- Retransmit timer set to $\beta \cdot \text{RTT}$, 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: $\text{RTT} + 4 \cdot \text{rttvar}$
  - $\text{rttvar} = \chi \cdot \text{dev} + (1 - \chi) \cdot \text{rttvar}$
    - $\text{dev} = \text{linear deviation}$ (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 ($\text{RTO} \geq 2\text{ticks}$)
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

**flow control**

sender won’t overrun receiver’s buffers by transmitting too much, too fast

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

receiver buffering
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 ⇒ wasted bandwidth
- Window size > delay*bw ⇒
  - Queuing at intermediate routers ⇒ 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. \texttt{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
  - Wait 30 seconds
  - Send SYN
  - Receive SYN & ACK
  - Send ACK
- TIME_WAIT
  - Receive FIN
  - Send ACK
- FIN_WAIT_2
  - Receive ACK
  - Send nothing
- FIN_WAIT_1
  - Receive ACK
  - Send FIN
- SYN_SENT
  - Send SYN
  - Receive SYN & ACK
- ESTABLISHED
  - Client application initiates close connection
TCP Connection Management (6)

TCP server lifecycle

CLOSED

server application creates a listen socket

LISTEN

receive SYN
send SYN & ACK

SYN_RCVD

receive ACK
send nothing

ESTABLISHED

receive FIN
send ACK

CLOSE_WAIT

send FIN

LAST_ACK

receive ACK
send nothing
TCP state machine

Diagram showing the TCP state transition diagram with states including:
- CLOSED
- LISTEN
- SYN_RCVD
- SYN_SENT
- ESTABLISHED
- FIN_WAIT_1
- CLOSING
- FIN_WAIT_2
- TIME_WAIT
- CLOSE_WAIT
- LAST_ACK
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: cwnd = 1
for (each segment ACKed)
   cwnd++
until (loss event OR cwnd> 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 \sim 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

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

- 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 ⇒ 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