Transport Layer

Dr. Xiaoming Fu
Prof. Dr. Dieter Hogrefe

Telematics group
University of Göttingen, Germany
Our goals:

- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

- Learn about the interface between OS and TCP/IP networks
  - socket

Learn about transport layer protocols in the Internet:

- UDP: connectionless transport
- TCP: connection/stream-oriented transport
- TCP congestion control
- SCTP: message-oriented transport: multi-homing, multi-stream support within a connection
- DCCP: datagram congestion control protocol.

Outline

- Transport-layer services
- Socket
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control
- SCTP and DCCP
Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP

WS 2004/05, fu@informatik.cs.uni-goettingen.de
Transport vs. network layer

- **network layer**: logical communication between hosts
- **transport layer**: logical communication between processes
  - relies on, enhances, network layer services

**Household analogy:**

12 kids sending letters to 12 kids
- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service
Overview of Sockets: the TCP/IP user interface

- **Socket**: Create a new communication end point
- **Bind**: Attach a local address to a socket
- **Listen**: Announce willingness to accept connections
  - Give queue size
- **Accept**: Block the caller until a connection attempt arrives
  - Accept next connection request
- **Connect**: Actively attempt to establish a connection
- **Send**: Send some data over the connection
- **Recv**: Receive some data from the connection
- **Close**: Release the connection
Step 1: Create a socket

- Suppose we make the call:
  \[
  s = \text{socket} ( \ldots \text{ assorted parameters here } \ldots);
  \]
  and the result is file descriptor \( s = 3 \)

- We have the following situation:
Step 2: Bind the socket to a port

- Suppose we do:
  \[
  \text{result} = \text{bind}( \ldots 80, \ldots );
  \]
  and the result is not -1 (that is, it worked!)
- We now have the following situation:
Step 3: Listen for connections

- Suppose we do:
  \[
  \text{result} = \text{listen}( 2 );
  \]
  and the result is not -1 (that is, it worked!)
- We now have the following situation:
Step 4: Incoming connection request

- Suppose we do:
  
  ```
  name.sin_port = 80;
  name.sin_addr.s.addr = inet_addr("123.90.47.122");
  result = connect( s, name, sizeof( name ) );
  ```

  and the result is not -1 (that is, it worked!)

- We now have:

![Diagram showing network connections between client and server programs.]
Step 5: Accept a connection request

• Suppose we do:
  
  ```c
  newSocket = accept( s, addr, &addr_length );
  ```
  
  and the result is a new socket with file descriptor 4.

• We now have the following situation:
But, what we really want is…

- Why?
  - Server can handle multiple, simultaneous connection requests
  - A separate handler process can be created for each request
Programming with sockets

• **Styles of communication:**
  – stream: reliable, two-way byte streams
  – datagram: unreliable, two-way record-oriented
  – etc.

• **Communication domains**
  – UNIX
    • endpoints (sockets) named with file-system pathnames
    • supports stream and datagram
  – Internet
    • endpoints named with IP addresses
    • supports stream and datagram
  – others

• **Protocols**
  – e.g., TCP/IP, UDP/IP
Using Datagram Sockets (1)

- **Receiver steps**
  1) create socket
     - *socket* system call
       ```c
       int socket(int domain, int type, int protocol);
       fd = socket(PF_UNIX, SOCK_DGRAM, 0);
       ```
  2) set up receiver’s name
     - put name in *sockaddr_un* structure
       ```c
       struct sockaddr_un {
       short sun_family; /* PF_UNIX */
       char sun_path[108]; /* path name */
       } name;
       ```
       ```c
       name.sun_family = PF_UNIX;
       memcpy(name.sun_path, path, strlen(path));
       ```
Using Datagram Sockets (2)

- Receiver steps (continued)
  3) bind receiver’s name to socket
     - bind takes a generic `struct sockaddr` argument and is given the combined length of the first part of the structure and that portion of the second part that is used

```c
name_len = sizeof(name.sun_family) + strlen(name.sun_path);
bind(fd, (struct sockaddr *)&name, name_len);
```
Using Datagram Sockets (3)

• Receiver steps (continued)

4) receive (and send) data

– use `recvfrom` system call to obtain caller’s address

```c
int recvfrom(int s, char *buf, int len, int flags, struct sockaddr *from, int *fromlen);

struct sockaddr_un sender_name;
int sender_len = sizeof(sender_name);

recvfrom(fd, buf, sizeof(buf), 0,
(struct sockaddr *)&sender_name, &sender_len);
```
Using Datagram Sockets (4)

- **Sender steps**
  1) create socket
  2) set up sender’s name (optional)
  3) bind sender’s name to socket (optional)
  4) set up receiver’s name

```c
struct sockaddr_un recvr_name;
int recvr_len;

recvr_name.sun_family = PF_UNIX;
memcpy(recvr_name.sun_path, path, strlen(path));
recvr_len = sizeof(recvr_name.sun_family) +
            strlen(recvr_name.sun_path);
```

WS 2004/05, fu@informatik.cs.uni-goettingen.de
Using Datagram Sockets (5)

- Sender steps (continued)
  5) send (and receive) data
  - use `sendto` system call to send datagram

```c
int sendto(int s, const char *msg, int len, int flags, const struct sockaddr *to, int tolen);
```

```c
sendto(fd, buf, sizeof(buf), 0,
       (const struct sockaddr *)&recvr_name, recvr_len);
```
Using Stream Sockets (1)

• Server steps
  1) create socket
     – *socket* system call

```c
int socket(int domain, int type, int protocol);
fd = socket(PF_INET, SOCK_STREAM, 0);
```
Using Stream Sockets (2)

- **Server steps (continued)**

  2) set up server’s address
     - put “wildcard” internet address in `sockaddr_in` structure
       - server may have multiple interfaces; we want to be able to receive on all of them
       - must convert from *host byte order to network byte order*

```c
struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8]; /* padding */
} my_addr;

my_addr.sin_family = PF_INET;
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
my_addr.sin_port = htonl(port);
```
Using Stream Sockets (3)

- Server steps (continued)

3) bind server’s name to socket
   ```c
   bind(fd, (struct sockaddr *)&my_addr,
        sizeof(my_addr));
   ```

4) set up socket to be in listening mode
   - `backlog` is the max length of the queue of waiting connections
   ```c
   int listen(int fd, int backlog);
   ```
Using Stream Sockets (4)

- **Server steps (continued)**
  5) wait for a connection
  - when connection is received, a new socket is created for communication on it
  - as an option, the address of the connector (client) is passed back
    ```c
    int accept(int fd, struct sockaddr *addr, int *addrlen);
    ```
  6) receive and send data
  - simple `read` and `write` system calls can be used
Using Stream Sockets (5)

• Client steps
  1) create socket
    – as an option, one can bind the socket to a particular port number
    – port numbers less than 1024 are reserved for privileged users
Using Stream Sockets (6)

• Client steps (continued)

2) find internet address of the server
   – look it up in domain name service (DNS)
   – each host may have a list of interfaces; we choose the first one

```c
struct hostent *hostinfo;
hostinfo = gethostbyname("botrytis.cs.brown.edu");
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family = PF_INET;
memcpy(&server_addr.sin_addr, hostinfo->h_addr_list[0], hostinfo->h_length);
server_addr.sin_port = htons(port);
```
Using Stream Sockets (7)

- **Client steps (continued)**
  3) **connect to server**
      ```c
      connect(fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
      ```
  4) **send and receive data**
      - simple *write* and *read* system calls can be used
Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
Multiplexing/demultiplexing

Demultiplexing at rcv host:
delivering received segments
to correct socket

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

= socket  = process

application
transport
network
link
physical

P3

P1

P2

P4

application
transport
network
link
physical

host 1

host 2

host 3

WS 2004/05, fu@informatik.cs.uni-goettingen.de
How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket
Connectionless demultiplexing

- Create sockets with port numbers:
  
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(99111);
  DatagramSocket mySocket2 = new DatagramSocket(99222);
  ```

- UDP socket identified by two-tuple:
  
  `(dest IP address, dest port number)`

- When host receives UDP segment:
  
  - checks destination port number in segment
  - directs UDP segment to socket with that port number

- IP datagrams with different source IP addresses and/or source port numbers directed to same socket
Connectionless demux (cont)

DatagramSocket serverSocket = new DatagramSocket(6428);

SP provides “return address”
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to 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 (cont)
UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- 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

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP:
  add reliability at application layer
  - application-specific error recovery!

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>

Length, in bytes of UDP segment, including header

Application data (message)
UDP checksum

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

**Sender:**
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement 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 error detected.

*But maybe errors nonetheless? More later…..*
TCP: Overview

- **point-to-point:**
  - one sender, one receiver

- **reliable, in-order **byte** steam:**
  - no “message boundaries”

- **pipelined:**
  - TCP congestion and flow control set window size

- **send & receive buffers**

  ![TCP diagram](image)

- **full duplex data:**
  - bi-directional data flow in same 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

RFCs: 793, 1122, 1323, 2018, 2581
### TCP Segment Structure

- **32 bits**
- **Source Port #**
- **Destination Port #**
- **Sequence Number**
- **Acknowledgement Number**
- **Receive Window**
- **Checksum**
- **Urgent Data Pointer**
- **Options (variable length)**
- **Application Data (variable length)**

**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)

**Counting by bytes of data (not segments!)**

**# bytes rcvr willing to accept**
TCP 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
  - A: TCP spec doesn’t say, - up to implementor

Simple telnet scenario:
- Host A sends sequence 4 and ACK 7
- Host B responds with sequence 9 and data 'C'
- Host A receives 'C' and ACKs receipt of 'C'
- Host B echoes 'C' back

User types 'C'
- Host B ACKs receipt of 'C'
- Host A receives ACK 79 and data 'C'

TCP Round Trip Time and 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?
- \textit{SampleRTT}: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- \textit{SampleRTT} will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current \textit{SampleRTT}
TCP Round Trip Time and 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: \( \alpha = 0.125 \)
Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr
TCP Round Trip Time and Timeout

Setting the timeout

- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

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

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

Then set timeout interval:

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4\times\text{DevRTT}
\]
TCP reliable data transfer

- TCP creates rdt service on top of IP’s unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer

- Retransmissions are triggered by:
  - timeout events
  - duplicate acks

- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control
TCP sender events:

**data rcvd from app:**
- Create segment with seq #
- seq # 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 rcvd:**
- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments
TCP sender (simplified)

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
    switch(event)
        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum = NextSeqNum + length(data)
        event: timer timeout
            retransmit not-yet-acknowledged segment with
            smallest sequence number
            start timer
        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
            }
} /* end of loop forever */

Comment:
• SendBase-1: last cumulatively ack’ed byte
Example:
• SendBase-1 = 71; y = 73, so the rcvr wants 73+; y > SendBase, so that new data is acked
TCP: retransmission scenarios

Host A
Seq=100, 20 bytes data
ACK=100

Host B
Seq=92, 8 bytes data
ACK=120
Seq=92 timeout
ACK=120

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

Host B
Seq=92, 8 bytes data
ACK=100

SendBase = 100

SendBase = 100
SendBase = 120

lost ACK scenario

premature timeout

WS 2004/05, fu@informatik.cs.uni-goettingen.de
TCP retransmission scenarios (more)

Cumulative ACK scenario

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

Host B
Seq=100, 20 bytes data
ACK=120

SendBase = 120

Timeout

Loss

WS 2004/05, fu@informatik.cs.uni-goettingen.de
## 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>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 seq #. 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 seq. #. Gap detected</td>
<td>Immediately send duplicate ACK, indicating seq. # 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>
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.

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires
Fast retransmit algorithm:

event: ACK received, with ACK field value of y
  if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
      start timer
  }
else {
  increment count of dup ACKs received for y
  if (count of dup ACKs received for y = 3) {
    resend segment with sequence number y
  }
}
TCP Flow Control

- receive side of TCP connection has a receive buffer:
  - flow control: sender won’t overflow receiver’s buffer by transmitting too much, too fast
  - speed-matching service: matching the send rate to the receiving app’s drain rate

- app process may be slow at reading from buffer
TCP Flow control: how it works

- Rcvr advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow
  - guarantees receive buffer doesn’t overflow

(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer
  - $\text{RcvWindow} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$
Recall: TCP sender, receiver establish “connection” before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. RcvWindow)

- **client**: connection initiator
  ```java
  Socket clientSocket = new Socket("hostname","port number");
  ```

- **server**: contacted by client
  ```java
  Socket connectionSocket = welcomeSocket.accept();
  ```

Three way handshake:

**Step 1**: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

**Step 2**: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

**Step 3**: client receives SYNACK, replies with ACK segment, which may contain data
TCP Connection Management (cont.)

**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.
**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 (cont)

TCP client lifecycle

TCP server lifecycle

wait 50 seconds
receive FIN send ACK
receive ACK send nothing
client application initiates a TCP connection
send SYN
receive SYN & ACK send ACK
client application initiates close connection
send FIN
receive ACK send nothing
receive FIN send ACK
receive ACK send nothing
server application creates a listen socket
receive SYN send SYN & ACK
receive ACK send nothing
Principles of Congestion Control

Congestion:

- informally: “too many sources sending too much data too fast for network to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!
Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

- large delays when congested
- maximum achievable throughput
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet
Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- “perfect” retransmission only when loss: $\lambda_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes $\lambda_{in}$ larger (than perfect case) for same $\lambda_{out}$

“costs” of congestion:
- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
 Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_{\text{in}}$ and $\lambda'_{\text{in}}$ increase?
Another “cost” of congestion:

- when packet dropped, any “upstream transmission capacity used for that packet was wasted!
Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at
Case study: ATM ABR congestion control

ABR: available bit rate:
- “elastic service”
- if sender’s path “underloaded”:
  - sender should use available bandwidth
- if sender’s path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:
- sent by sender, interspersed with data cells
- bits in RM cell set by switches (“network-assisted”)
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact
Case study: ATM ABR congestion control

- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender’s send rate thus minimum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell
TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:
  \[ \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \]
- Roughly,
  \[ \text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec} \]
- \text{CongWin} is dynamic, function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout or 3 duplicate acks
- TCP sender reduces rate (\text{CongWin}) after loss event

three mechanisms:
- AIMD
- slow start
- conservative after timeout events
TCP AIMD

**multiplicative decrease:**
cut \( \text{CongWin} \) in half after loss event

**additive increase:**
increase \( \text{CongWin} \) by 1 MSS every RTT in the absence of loss events: *probing*

Long-lived TCP connection
TCP Slow Start

• When connection begins, $CongWin = 1$ MSS
  – Example: MSS = 500 bytes & RTT = 200 msec
  – initial rate = 20 kbps

• available bandwidth may be >> MSS/RTT
  – desirable to quickly ramp up to respectable rate

• When connection begins, increase rate exponentially fast until first loss event
TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double CongWin every RTT
  - done by incrementing CongWin for every ACK received

- **Summary:** initial rate is slow but ramps up exponentially fast
Refinement

- After 3 dup ACKs:
  - $\text{CongWin}$ is cut in half
  - window then grows linearly
- But after timeout event:
  - $\text{CongWin}$ instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

Philosophy:
- 3 dup ACKs indicates network capable of delivering some segments
- timeout before 3 dup ACKs is “more alarming”
Q: When should the exponential increase switch to linear?
A: When $\text{CongWin}$ gets to 1/2 of its value before timeout.

**Implementation:**
- Variable Threshold
- At loss event, Threshold is set to 1/2 of $\text{CongWin}$ just before loss event
Summary: TCP Congestion Control

- When $\text{CongWin}$ is below $\text{Threshold}$, sender in slow-start phase, window grows exponentially.
- When $\text{CongWin}$ is above $\text{Threshold}$, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, $\text{Threshold}$ set to $\text{CongWin}/2$ and $\text{CongWin}$ set to $\text{Threshold}$.
- When timeout occurs, $\text{Threshold}$ set to $\text{CongWin}/2$ and $\text{CongWin}$ is set to 1 MSS.
TCP Fairness

**Fairness goal:** if $K$ TCP sessions share same bottleneck link of bandwidth $R$, each should have average rate of $R/K$
Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally
Fairness and UDP

• Multimedia apps often do not use TCP
  – do not want rate throttled by congestion control
• Instead use UDP:
  – pump audio/video at constant rate, tolerate packet loss
• Research area: TCP friendly

Fairness and parallel TCP connections

• nothing prevents app from opening parallel connections between 2 hosts.
• Web browsers do this
• Example: link of rate R supporting 9 connections;
  – new app asks for 1 TCP, gets rate R/10
  – new app asks for 11 TCPs, gets R/2!
Stream Control Transmission Protocol

Features common to TCP:
- Reliable Data delivery
- Ordered Data delivery
- Congestion Control
- Flow Control

New features different from TCP:
- Multi-homing
- Multiple Streams in a Association
- Message oriented (TCP: Byte-oriented)
- Extension for unreliable UDP-style data transfer
- More: security mechanisms

SCTP – another transport protocol (RFC 2960)
DCCP

- DCCP = UDP + congestion control modules + “connection” for DOS prevention
- “2 half connections” (A→B, B→A): separating different directions, works better with firewall/NAT
- “Plug & Play” Congestion Control:
  - CC mechanism and parameters (both ways) chosen during connection setup
  - Currently two mechanisms:
    - TFRC (“TCP Friendly Rate Control”, uses TCP throughput equation, per RFC3448)
    - TCP-like (TCP with tweaked parameters)
    - Can add more later
Transport Layer: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP
  - Intro to SCTP and DCCP

Next:
- Leave transmission technology
- Move into multimedia application
- With QoS and other enhanced services