TCP: Overview

- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size

- **connection-oriented:**
  - handshaking (exchange of control msgs) initiates sender, receiver state before data exchange

- **flow controlled:**
  - sender will not overwhelm receiver

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

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

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

Send and Receive Window

- Send and receive buffer sizes can be set on individual sockets by using the `SO_SNDBUF` and `SO_RCVBUF` socket options with the `setsockopt(2)` call.

SO_RCVBUF

- Sets or gets the maximum socket receive buffer in bytes.
- The kernel doubles this value (to allow space for bookkeeping overhead) when it is set using `setsockopt(2)`, and this doubled value is returned by `getsockopt(2)`.
- The default value is set by the `/proc/sys/net/core/rmem_default` file, and the maximum allowed value is set by the `/proc/sys/net/core/rmem_max` file.
- The minimum (doubled) value for this option is 256.
SO_SNDBUF

- Sets or gets the maximum socket send buffer in bytes.
- The kernel doubles this value (to allow space for bookkeeping overhead) when it is set using `setsockopt(2)`, and this doubled value is returned by `getsockopt(2)`.
- The default value is set by the `/proc/sys/net/core/wmem_default` file and the maximum allowed value is set by the `/proc/sys/net/core/wmem_max` file.
- The minimum (doubled) value for this option is 2048.

Example:

```c
int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen);
```

```c
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

Example:

```c
int n = 1024 * 3;
setsockopt(socket, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));
```

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)

TCP seq. numbers, ACKs

**sequence numbers:**
- byte stream “number” of first byte in segment’s data

**acknowledgements:**
- 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
**TCP seq. numbers, ACKs**

USER

- Types 'C'
- Host ACKs receipt of 'C’, echoes back 'C'
- Host A

- Seq=42, ACK=79, data = 'C'
- Seq=79, ACK=43, data = 'C'
- Seq=43, ACK=80

SIMPLE TELNET SCENARIO

**TCP round trip time, 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?
- SampleRTT: measured time from segment transmission until ACK receipt
- ignore retransmissions
- SampleRTT will vary, want estimated RTT “smoother”
- average several recent measurements, not just current SampleRTT

**Q:** how to set TCP timeout value?
- timeout interval: EstimatedRTT plus “safety margin”
- large variation in EstimatedRTT -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:
  \[ \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}| \]
  (typically, \( \beta = 0.25 \))
- \[ \text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT} \]

- Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/
**Chapter 3 outline**

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
   • segment structure
   • reliable data transfer
   • flow control
   • connection management
3.6 principles of congestion control
3.7 TCP congestion control

**TCP reliable data transfer**

- TCP creates rdt service on top of IP’s unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

Let’s 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 ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

**TCP sender (simplified)**

- data received from application above
- create segment, seq #: NextSeqNum
- pass segment to IP (i.e., “send”)
- NextSeqNum = NextSeqNum + length(data)
- if (timer currently not running)
  - start timer
- else stop timer

- ACK received, with ACK field value y
  - if (y > SendBase) {
    - SendBase = y
    - /* SendBase−1: last cumulatively ACKed byte */
    - if (there are currently not-yet-acked segments)
    - start timer
    - else stop timer
  - }
### TCP: retransmission scenarios

- **lost ACK scenario**
  - Host A
    - Seq=92, 8 bytes of data
    - ACK=100
    - Timeout
  - Host B
    - Seq=92, 8 bytes of data
    - ACK=100

- **premature timeout**
  - Host A
    - Seq=100, 20 bytes of data
    - ACK=120
  - Host B
    - Seq=120, 15 bytes of data
    - Timeout

- **cumulative ACK**
  - Host A
    - SendBase=100
  - Host B
    - SendBase=120

### TCP ACK generation

<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>

### TCP 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.

  **TCP fast retransmit**
  - if sender receives 3 ACKs for same data (“triple duplicate ACKs”), resend unacknowledged segment with smallest seq #
  - likely that unacknowledged segment lost, so don’t wait for timeout
### TCP fast retransmit

Fast retransmit after sender receipt of triple duplicate ACK.

- **Host A**
  - Seq=92, 8 bytes of data
  - ACK=100
  - ACK=100
  - ACK=100
- **Host B**
  - Seq=100, 20 bytes of data
  - ACK=100

### Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
3.6 principles of congestion control
3.7 TCP congestion control

### TCP flow control

- Receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast.
- Flow control

### TCP flow control

- Receiver “advertises” free buffer space by including `rwnd` value in TCP header of receiver-to-sender segments
  - `RcvBuffer` size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust `RcvBuffer`
- Sender limits amount of unacked (“in-flight”) data to receiver’s `rwnd` value
- Guarantees receive buffer will not overflow
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
3.6 principles of congestion control
3.7 TCP congestion control

Connection Management

before exchanging data, sender/receiver “handshake”:
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

Agreeing to establish a connection

2-way handshake:

Q: will 2-way handshake always work in network?
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side

Agreeing to establish a connection

2-way handshake failure scenarios:
TCP 3-way handshake

client state
- LISTEN
- SYNSENT
- ESTAB

server state
- LISTEN
- SYN RCVD

TCP 3-way handshake: FSM

- client state
  - LISTEN
  - SYN SENT
  - ESTAB

- server state
  - LISTEN
  - SYN RCVD

TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection

- client state
  - ESTAB
  - FIN_WAIT_1
  - FIN_WAIT_2
  - TIMED_WAIT
  - CLOSED

- server state
  - ESTAB
  - CLOSE_WAIT
  - LAST_ACK
  - CLOSED
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
   - segment structure
   - reliable data transfer
   - flow control
   - connection management
3.6 principles of congestion control
3.7 TCP congestion control

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
- output link capacity: R
- no retransmission

Causes/costs of congestion: scenario 2

- one router, finite buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: \( \lambda_{in} = \lambda_{out} \)
  - transport-layer input includes retransmissions: \( \lambda'_{in} \geq \lambda_{in} \)
**Causes/costs of congestion: scenario 2**

**Idealization: perfect knowledge**
- sender sends only when router buffers available

- **$\lambda_{in}$**: original data
- **$\lambda'_{in}$**: original data, plus retransmitted data
- **$\lambda_{out}$**

**Causes/costs of congestion: scenario 2**

**Idealization: known loss**
- packets can be lost, dropped at router due to full buffers
- sender only resends if packet known to be lost

-**$\lambda_{in}$**: original data
- **$\lambda'_{in}$**: original data, plus retransmitted data
- **$\lambda_{out}$**

**Causes/costs of congestion: scenario 2**

**Idealization: known loss**
- packets can be lost, dropped at router due to full buffers
- sender only resends if packet known to be lost

- **$\lambda_{in}$**: original data
- **$\lambda'_{in}$**: original data, plus retransmitted data
- **$\lambda_{out}$**

**Causes/costs of congestion: scenario 2**

**Idealization: known loss**
- packets can be lost, dropped at router due to full buffers
- sender only resends if packet known to be lost

- **$\lambda_{in}$**: original data
- **$\lambda'_{in}$**: original data, plus retransmitted data
- **$\lambda_{out}$**

**Causes/costs of congestion: scenario 2**

**Realistic: duplicates**
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending two copies, both of which are delivered

- **$\lambda_{in}$**: original data
- **$\lambda'_{in}$**: original data, plus retransmitted data
- **$\lambda_{out}$**

**Causes/costs of congestion: scenario 2**

**Realistic: duplicates**
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending two copies, both of which are delivered

- **$\lambda_{in}$**: original data
- **$\lambda'_{in}$**: original data, plus retransmitted data
- **$\lambda_{out}$**

**Causes/costs of congestion: scenario 2**
Causes/costs of congestion: scenario 2

Realistic: duplicates
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending two copies, both of which are delivered

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

Causes/costs of congestion: scenario 3

Q: what happens as \( \lambda_{in} \) and \( \lambda_{in}' \) increase?
A: as red \( \lambda_{in} \) increases, all arriving blue pkts at upper queue are dropped, blue throughput \( \to 0 \)

Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
3.6 principles of congestion control
3.7 TCP congestion control
**TCP congestion control:** additive increase multiplicative decrease

- **approach:** sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **additive increase:** increase $cwnd$ by 1 MSS every RTT until loss detected
  - **multiplicative decrease:** cut $cwnd$ in half after loss

**AIMD saw tooth behavior:** probing for bandwidth

**TCP congestion Control: details**

- **sender limits transmission:**
  - $cwnd$ is dynamic, function of perceived network congestion

**TCP Slow Start**

- **when connection begins,** increase rate exponentially until first loss event:
  - initially $cwnd = 1$ MSS
  - double $cwnd$ every RTT
  - done by incrementing $cwnd$ for every ACK received

**summary:** initial rate is slow but ramps up exponentially fast

**TCP: detecting, reacting to loss**

- **loss indicated by timeout:**
  - $cwnd$ set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- **loss indicated by 3 duplicate ACKs:** TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - $cwnd$ is cut in half window then grows linearly
- **TCP Tahoe always sets $cwnd$ to 1** (timeout or 3 duplicate acks)
Q: when should the exponential increase switch to linear?
A: when cwnd gets to 1/2 of its value before timeout.

Implementation:
- variable ssthresh
- on loss event, ssthresh is set to 1/2 of cwnd just before loss event

TCP throughput
- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send
  - W: window size (measured in bytes) where loss occurs
    - avg. window size (# in-flight bytes) is 3/4 W
    - avg. throughput is 3/4W per RTT

TCP Futures: TCP over “long, fat pipes”
- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:
  \[ \text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \cdot \sqrt{L}} \]
  - to achieve 10 Gbps throughput, need a loss rate of \( L = 2 \cdot 10^{-10} \) – a very small loss rate!
- new versions of TCP for high-speed
**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 (more)**

*Fairness and UDP*
- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

*Fairness, parallel TCP connections*
- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

---

**Summary**

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP
Disclaimer

- Parts of the lecture slides contain original work of James Kurose and Keith Ross. The slides are intended for the sole purpose of instruction of computer networks at the University of Rochester. All copyrighted materials belong to their original owner(s).