Lecture 7.
TCP mechanisms for:

- data transfer control / flow control
- error control
- congestion control

Graphical examples (applet java) of several algorithms at:
http://www.ce.chalmers.se/~fcela/tcp-tour.html
Data transfer control over TCP
a double-face issue:

- **Bulk data transfer**
  - HTTP, FTP, ...
  - goal: attempt to send data as fast as possible
  - problems: sender may transmit faster than receiver
    - Flow control

- **Interactive**
  - TELNET, RLOGIN, ...
  - goal: attempt to send data as soon as possible
  - Problem: efficiency - interactivity trade-off
    - The tinygrams issue!
      - (1 byte payload / segment - 20 TCP + 20 IP header)
TCP pipelining

- More than 1 segment "flying" in the network
- Transfer efficiency increases with $W$

$$thr = \min\left( C, \frac{W \cdot MSS}{RTT + MSS / C} \right)$$

- So, why an upper limit on $W$?
Why flow control?

- **Limited receiver buffer**
  - If MSS = 2KB = 2048 bytes
  - And receiver buffer = 8 KB = 8192 bytes
  - Then W must be lower or equal than 4 x MSS

- **A possible implementation:**
  - During connection setup, exchange W value.
  - *DOES NOT WORK. WHY?*
Window-based flow control

- Receiver buffer capacity varies with time!

  ⇒ Upon application process read()
  [asynchronous, not depending on OS, not predictable]

- MSS = 2KB = 2048 bytes
- Receiver Buffer capacity = 10 KB = 10240 bytes
- TCP data stored in buffer: 3 segments
- Receiver window = Spare room: 10-6 = 4KB = 4096 bytes

  ⇒ Then, at this time, W must be lower or equal than 2 × MSS
- **Window size field**: used to advertise receiver's remaining storage capabilities

- 16 bit field, on every packet
- Measure unit: bytes, from 0 (included) to 65535
- **Sender rule**: \( \text{LastByteSent} - \text{LastByteAcked} \leq \text{RcvWindow} \).
- \( W=2048 \) means:
  - I can accept other 2048 bytes since ack, i.e. bytes \([\text{ack}, \text{ack}+W-1]\)
  - also means: sender may have 2048 bytes outstanding (in multiple segments)
What is flow control needed for?

- Window flow control guarantees receiver buffer to be able to accept outstanding segments.
- When receiver buffer full, just send back win=0
- In essence, flow control guarantees that transmission bit rate never exceed receiver rate
  - in average!
  - Note that instantaneous transmission rate is arbitrary…
  - as well as receiver rate is discretized (application reads)
Dynamic window based reduces to pure sliding window when receiver app is very fast in reading data…
Dynamic window - example

TCP CONN SETUP

Application does a 2K write

2K, seq=2048

Exchanged param: MSS=2K, sender ISN=2047, WIN=4K (carried by receiver SYN-ACK)

Application does a 3K write

2K, seq=4096

Ack=4096, win=2048

Sender blocked

Ack=6144, win=0

Sender unblocks may send last 1K

Ack=6144, win=2048

1K, seq=6144

Application does a 2K read

Ack=6144, win=0

2K, seq=6144

FULL

Rec. Buffer

0 4K

EMPTY

0 4K

2K

0 4K

FULL

2K
Performance: bounded by receiver buffer size

- Up to 1992, common operating systems had transmitter & receiver buffer defaulted at 4096
  - e.g. SunOS 4.1.3
- way suboptimal over Ethernet LANs
  - raising buffer to 16384 = 40% throughput increase (Papadopoulos & Parulkar, 1993)
  - e.g. Solaris 2.2 default
- most socket APIs allow apps to set (increase) socket buffer sizes
  - But theoretical maximum remains $W=65535$ bytes…
Maximum achievable throughput
(assuming infinite speed line...)

\[ W = 65535 \text{ bytes} \]
Window Scale Option

- Appears in SYN segment
  - operates only if both peers understand option
- allows client & server to agree on a different W scale
  - specified in terms of bit shift (from 1 to 14)
  - maximum window: $65535 \times 2^b$
  - $b=14$ means max $W = 1,073,725,440$ bytes!!
Blocked sender deadlock problem

Since ACK does not carry data, no ack from sender expected....
Solution: Persist timer

- When win=0 (blocked sender), sender starts a “persist” timer
  - Initially 500ms (but depends on implementation)
- When persist timer elapses AND no segment received during this time, sender transmits “probe”
  - Probe = 1 byte segment; makes receiver reannounce next byte expected and window size
  - this feature necessary to break deadlock
  - if receiver was still full, rejects byte
  - otherwise acks byte and sends back actual win
- Persist time management (exponential backoff):
  - Doubles every time no response is received
  - Maximum = 60s
Interactive applications

ideal rlogin operation: 4 transmitted segments per 1 byte!!!!!

Header overhead:

20 TCP header
+ 20 IP header
+ 1 data

Interactive apps: create some tricky situations....
The silly window syndrome

SCENARIO

Bulk data source  TCP connection  Full recv buffer

Interactive user (one byte at the time)
The silly window syndrome

Network loaded with tinygrams (40 bytes header + 1 payload!!)

Forever!

Ack=X, win=1

Ack=X+1, win=0

Ack=X+1, win=1

Ack=X+2, win=0

1 byte read

Buffer FULL

1 byte

Buffer FULL

1 byte

Buffer FULL

Fill up buffer until win=0
Silly window solution

- Problem discovered by David Clark (MIT), 1982
- Easily solved, by preventing receiver to send a window update for 1 byte
- Rule: send window update when:
  - Receiver buffer can handle a whole MSS
  - Or
  - Half received buffer has emptied (if smaller than MSS)
- Sender also may apply rule
  - By waiting for sending data when win low
Nagle’s algorithm
(RFC 896, 1984)

NAGLE RULE: a TCP connection can have only ONE SMALL outstanding segment

self-clocking algorithm:

on LANs, plenty of tynigrams

on slow WANs, data aggregation

G.Bianchi, G.Neglia, V.Mancuso
Comments about Nagle’s algo

→ Over ethernet:
  ⇒ about 16 ms round trip time
  ⇒ Nagle algo starts operating when user digits more than 60 characters per second (!!!)

→ Disabling Nagle’s algorithm
  ⇒ a feature offered by some TCP APIs
    → set TCP_NODELAY
  ⇒ example: mouse movement over X-windows terminal
**PUSH flag**

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 bit Sequence number</td>
<td></td>
</tr>
<tr>
<td>32 bit acknowledgement number</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header length</th>
<th>6 bit</th>
<th>Reserved</th>
<th>U R G</th>
<th>A C K</th>
<th>P R S</th>
<th>T S Y</th>
<th>F I N</th>
<th>Window size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Used to notify**

- TCP sender to send data
  - but for this an header flag NOT needed! Sufficient a “push” type indication in the TCP sender API
- TCP receiver to pass received data to the application
**Urgent data**

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **32 bit Sequence number**
- **32 bit acknowledgement number**

<table>
<thead>
<tr>
<th>Header length</th>
<th>6 bit Reserved</th>
<th>U</th>
<th>R</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>H</th>
<th>T</th>
<th>S</th>
<th>Y</th>
<th>N</th>
<th>F</th>
<th>I</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window size</td>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- URG on: notifies rx that “urgent” data placed in segment.
- When URG on, *urgent pointer* contains position of *last byte* of urgent data
  - or the one after the last, as some bugged implementations do??
  - and the first? No way to specify it!
- receiver is expected to pass all data up to urgent ptr to app
  - interpretation of urgent data is left to the app
- typical usage: ctrlC (interrupt) in rlogin & telnet; abort in FTP
- urgent data is a *second* exception to blocked sender

G.Bianchi, G.Neglia, V.Mancuso
TCP
Error control
TCP: a reliable transport

- **TCP is a reliable protocol**
  - all data sent are guaranteed to be received
  - *very important feature, as IP is unreliable network layer*

- **employs positive acknowledgement**
  - cumulative ack
  - selective ack may be activated when both peers implement it (use option)

- **does not employ negative ack**
  - error discovery via timeout (retransmission timer)
  - But “implicit NACK” is available
Error discovery
via retransmission timer expiration

Fundamental problem:
setting the retransmission timer right!
although lost ack may be discovered via subsequent acks
Retransmission timer setting

$\text{RTO} = \text{retransmission TimeOut}$

$\rightarrow \text{TOO SHORT: unnecessary retransmission occurs, loading the Internet with unnecessary packets}$

$\rightarrow \text{TOO LONG: throughput impairment when packets lost}$

G.Bianchi, G.Neglia, V.Mancuso
Retransmission timer setting

→ Cannot be fixed by protocol! Two reasons:

¬ different network scenarios have very different performance
  → LANs (short RTTs)
  → WANs (long RTTs)

¬ same network has time-varying performance (very fast time scale)
  → when congestion occurs (RTT grows) and disappears (RTT drops)
Adaptive RTT setting

Proposed in RFC 793

based on dynamic RTT estimation

- sender samples time between sending SEQ and receiving ACK ($M$)
- estimates RTT ($R$) by low pass filtering $M$
  (autoregressive, 1 pole)
  \[ R = \alpha R + (1-\alpha) M \]
  \[ \alpha = 0.9 \]
- sets $RTO = R \beta$
  \[ \beta = 2 \text{ (recommended)} \]
Problem: constant value $\beta=2$

SCENARIO 1: lightly loaded long-distance communication

Propagation = 100
Queueing = mean 5, most in range 0-10

RTO = $2 \times \text{measured RTT} \sim 2 \times 105 = 210$ TOO LARGE!

SCENARIO 2: mildly loaded short-distance communication

Propagation = 1
Queueing = mean 10, most in range 0-50

RTO = $2 \times \text{measured RTT} \sim 2 \times 11 = 22$ WAY TOO SMALL!
Problem: constant value $\beta=2$

SCENARIO 3: slow speed links

28.8 Kbps

Natural variation of packet sizes causes a large variation in RTT!

(from RFC 1122: utilization on 9.6 kbps link can improve from 10% up to 90% With the adoption of Jacobson algorithm)
Jacobson RTO (1988)

idea: make it depend on measured variance!

\[
\text{Err} = M - A \\
A := A + g \ \text{Err} \\
D := D + h (|\text{Err}| - D) \\
\text{RTO} = A + 4D
\]

\[g = \text{gain } (1/8)\]
- conceptually equivalent to $1-\alpha$, but set to slightly different value

\[D = \text{mean deviation}\]
- conceptually similar to standard deviation, but cheaper (does not require a square root computation)
- $h = 1/4$

\[\text{Jacobson’s implementation: based on integer arithmetic (very efficient)}\]
Guessing right?
Karn’s problem

Scenario 1

Scenario 2

G.Bianchi, G.Neglia, V.Mancuso
Solution to Karn’s problem

➔ Very simple: DO NOT update RTT when a segment has been retransmitted because of RTO expiration!

➔ Instead, use Exponential backoff

  ➔ double RTO for every subsequent expiration of same segment

  ➔ When at 64 secs, stay

  ➔ persist up to 9 minutes, then reset
Need for implicit NACKs

- TCP does not support negative ACKs
- This can be a serious drawback
  - Especially in the case of single packet loss
- Necessary RTO expiration to start retransmit lost packet
  - As well as following ones!!

- ISSUE: is there a way to have NACKs in an implicit manner???
The Fast Retransmit Algorithm

- Idea: use duplicate ACKs!
  - Receiver responds with an ACK every time it receives an out-of-order segment
  - ACK value = last correctly received segment

- FAST RETRANSMIT algorithm:
  - if 3 duplicate acks are received for the same segment, assume that the next segment has been lost. Retransmit it right away.
  - Helps if single packet lost. Not very effective with multiple losses

- And then? A congestion control issue...