TELE3118 extras
Transport Protocols
week “6” videos
TCP segment structure

Receiver sets *fields in italics & underlined*

- CWR = Congestion Window Reduced
- ECE = Explicit Congestion Notification Echo
- URG = Urgent pointer valid
- ACK = Acknowledgement number valid

**Connection mgmt fields**

- SYN = Synchronise
- RST = Reset
- FIN = Finish
- PSH = Push data

**Flow control fields**

- Window

**Error control fields**

- Acknowledgment Number
- Sequence Number
- Source Port
- Destination Port

**Connection setup packet of the 1st setup packet of connection => blue**

*† only ever invalid for 1st setup packet of connection*
TCP options

- **No Operation (NOP):** Pad & align for 32b words
- **Maximum Segment Size** [RFC 793]
  - Unit of congestion control
- **Window scale factor** [RFC 1323] for flow control
  - Enables full use of long fat links
    - (16b Window field was showing its age)
- **Timestamp** [RFC 1323] for error control
  - Protect Against Wrapped Sequences
  - E.g. simplify sender RTT calculations; avoid retransmission ambiguity;
- **Selective acknowledgements** [RFC 2018, 2883]
  - 2 options:
    - SACK permitted
    - SACK
  - ... and several others.

Standard options used by SYN from:
Windows XP: 8 bytes: MSS, SACK permitted + 2 NOPs
Linux: 20 bytes: MSS, SACK permitted, Time stamp, window scale + 1 NOP
Windows

Transmission Control Protocol, Src Port: dyna-access (3310), Dst Port: http
Source port: dyna-access (3310)
Destination port: http (80)
[Stream index: 1]
Sequence number: 0 (relative sequence number)
Header length: 28 bytes

Flags: 0x02 (SYN)
Window size: 16384
Checksum: 0x4a4e [correct]
Options: (8 bytes)

Maximum segment size: 1460 bytes
NOP
NOP
SACK permitted

0000 00 13 10 28 ab d3 00 13 ce 39 01 4a 08 00 45 00 ...( .... .9.J..E.
0010 00 30 e0 2d 40 00 80 06 cb 98 c0 a8 01 65 d0 4d .0.-@... ......e.M
0020 bc a6 0c ee 00 50 77 92 24 ff 00 00 00 00 70 02 .....Pw. $......p.
0030 40 00 4a 4e 00 00 02 04 05 b4 01 01 04 02 @.JN......
Transmission Control Protocol, Src Port: 60871 (60871), Dst Port: http (80)

Source port: 60871 (60871)
Destination port: http (80)
Sequence number: 0  (relative sequence number)
Header length: 40 bytes

Flags: 0x02 (SYN)
Window size: 5840
Checksum: 0xfcel [correct]
Options: (20 bytes)

Maximum segment size: 1460 bytes
SACK permitted
Timestamps: TSval 7384482, TSecr 0
NOP
Window scale: 6 (multiply by 64)
Unreliable love

Love can be unreliable, even when you use TCP.

From http://blog.ksplice.com/2010/04/dating-is-rough-at-the-transport-layer
Connection state diagram

Sockets calls:
CLOSE()  SEND()
“active OPEN”  = connect()
“passive OPEN”  = listen()

wait for ack to request to finish
wait for other end to finish

Figure mainly from RFC 793. RFC 1122 lists corrections; main one shown here in italics.
TCB = TCP Control Block (holding state info, e.g. seqno, window sizes, RTT estimate...)

Copyright © Apr.15, Tim Moors
DIY: netstat

- Windows & Linux offer “netstat” programs, whose default output lists currently used ports and their states.
  - Options give access to protocol statistics.

```
$ netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address          State
tcp    0      0 dhcp123_45.ee.uns:32832 www.isi.edu:http ESTABLISHED
tcp    0      0 dhcp123_45.ee.uns:32830 sourceforge.net:http ESTABLISHED
tcp    0      0 dhcp123_45.ee.uns:32849 maestro.ee.unsw.:domain TIME_WAIT
tcp    1      0 localhost.localdo:32778 localhost.localdoma:ipp CLOSE_WAIT
tcp    0      1 dhcp123_45.ee.uns:32848 noc.ietf.ORG:http FIN_WAIT1
tcp    0      0 dhcp123_45.ee.uns:32851 noc.ietf.ORG:http ESTABLISHED
tcp    0      0 dhcp123_45.ee.uns:32850 noc.ietf.ORG:http ESTABLISHED
 tcp    0      1 dhcp123_45.ee.uns:32847 noc.ietf.ORG:http FIN_WAIT1
```

- **Ephemeral client ports**
- **Well-known server ports**
- **Connection state**
Sliding window

Any (or all) of the packets/acks “in flight” may be lost due to error
=> sender must identify exactly what is being (re)sent
    receiver must identify exactly what has been received
=> “sliding windows”: window covers packets sent (RHS) but no ACK received yet (LHS)†.

e.g. window of 4 packets:

```
  0 1 2 3 4 5 6 7 8 9 ...
```

Sent? ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ... 
ACKed? ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ...

+ protocol should be able to continue running indefinitely without sequence number field expanding
=> sequence numbers “wrap” back to zero after reaching max value

† Assuming ACKs received in order of packets sent – i.e. no loss.
Go-back-N can be wasteful

**Go-back-N:**

Reducing what must be resent:  
(a)
- NAKs can expedite retransmission, even for go-back-N
- Selective retransmission

**Selective repeat:**
EWMA used in finance

Figure from http://etfdailynews.com/2012/08/03/major-indexes-triple-trend-channel-surfing-update-indexdjx-dji-indexspinx-spy-diarocks/
Extension material
TCP/UDP pseudoheader

Pseudoheaders: extend the transport layer header when calculating checksum, but are not transmitted.

| | Source Address |
| | Destination Address |
| | Zero | Protocol | Length |
| | Source Port | Destination Port |

Transmitter:
send() call specifies:
socket => SA, DA, protocol
data, length
TP assembles pseudoheader,
calculates checksum,
discards pseudoheader,
gives to IP: SA, DA, protocol,
length, segment

Receiver:
IP gives segment to transport layer
+ SA, DA, protocol, length
TP assembles pseudoheader,
calculates checksum
progress iff checksum covering
pseudoheader is OK

Address translators must update transport layer checksums to reflect pseudoheader changes.
Persist timer

TCP only acks data, not feedback. What if lose segment that increases rwnd from 0?
- Blocked source (rwnd=0) won’t send data to destination, so
- Destination won’t send an error control ack repeating the increased rwnd.
- Deadlock.

Solution:
- Source starts a persist timer (initially 1.5sec) when blocked with rwnd=0
- If timer expires before rwnd updated, send segment with 1B of data & double persist timer.
  - Destination might discard data, but will ACK & update rwnd.
Silly Window Syndrome

Problem: Does TCP send a $rwnd$ update every time app reads another byte from buffer? Destination is being too “aggressive” in encouraging the source.

Solution: Don’t send window update until there is a reasonable amount of space available in the receiver’s buffer. “reasonable” = min(maximum segment size, buffer size / 2)
Nagle’s algorithm

... is the source equivalent of the destination-caused Silly Window Syndrome.

Source app *writes* information to TCP in small amounts, e.g. single bytes. What size segments should TCP transmit?
- Small (e.g. as written by app) => many small packets in network.
- Large (e.g. wait to fill segment of maximum size) => delay.

How long should TCP source wait to receive information from app before sending?

Nagle’s algorithm: Only permit one small segment to be unacknowledged at any time. Send info in source’s buffer when either:
- Receive ack
- Volume awaiting transmission >
  - Maximum segment size
  - Half transmission window

To disable Nagle’s algorithm (e.g. when writing info for interactive app) use TCP_NODELAY socket option.

Delayed acks

- With an ack sent for each segment received, half the packets simply carry acks.
- What’s wrong with short packets?
  - Many overheads (e.g. header processing, MAC) are independent of packet length
  - May want to minimise number of packets sent, e.g. to save energy on mobile device.
- **Delayed acks**: Send ack after (whichever comes 1st)
  - Receiving two full-sized segments.
  - 500ms after receiving a segment.
    (Typically choose a smaller limit, e.g. 200ms)

(To simplify examples, we’ll generally assume that acks aren’t delayed.)

For details, see RFC 1122
Retransmission ambiguity

Is an ack that arrives for a segment that was retransmitted:
a. a response to the retransmission? 
or b. a delayed response to the original transmission?

Solution 1: Karn’s algorithm: Ignore RTT measurements made on segments that are retransmitted.

Solution 2: Use timestamps: data-bearing segments carry a timestamp (TCP option) indicating when they were (re)transmitted and acks return this timestamp. Source subtracts timestamp from ack from time of receiving ack to measure RTT.