TCP: Transmission Control Protocol

- IP is an *unreliable* datagram protocol
  - congestion or transmission errors cause lost packets
  - multiple routes may lead to out-of-order delivery
- If senders send too fast, routers or receivers cannot keep up (making congestion worse)
- When many senders compete, capacity must be fairly shared
- TCP's job is to fix those three problems
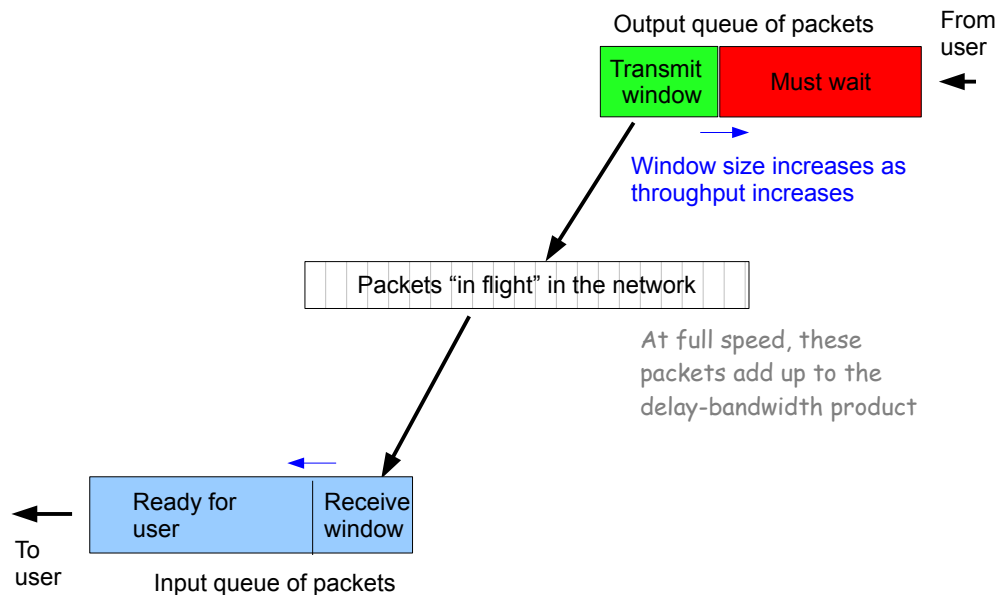 - flow control
  - retransmission after errors

1

Two approaches to Flow Control

- Rate Control - sender determines the maximum safe sending rate and never exceeds it
- Sliding Window - sender sends up to a "window" full of data but then pauses for an acknowledgement
  - Window size is adjusted dynamically to match network capacity
    - Window size is also known as "credit"
  - Missing acknowledgement causes retransmission
- TCP is a sliding window protocol

2

Sliding windows in action

Output queue of packets

From user

Transmit window

Must wait

Window size increases as throughput increases

Packets "in flight" in the network

At full speed, these packets add up to the delay-bandwidth product

Ready for user

Receive window

To user

Input queue of packets

3

Cranking up to speed

- TCP starts slowly
  - Initial window size is small
  - Send packets until window is empty
  - Increase window size as data flow accelerates
  - Decrease window size if data flow slows down
  - Retransmit when acknowledgments don't arrive
- Note that when A is talking to B and B is talking to A, the paths may be asymmetric, so TCP windows work independently in the two directions
- Note that TCP transmits a stream of *bytes* as far as user programs are concerned, broken up into *segments* by TCP itself

4

Why is this better than rate control?

- The sliding window approach works over an enormous range of speeds
  - It was designed in the days of 9600 baud modems, and it works (with some tuning) in the days of 10 Gb/s links
  - Rate control works best in fixed-speed networks
- It works reasonably well as router load increases towards 100%
  - Sharing between thousands or millions of competing TCP sessions is reasonably fair
  - Rate control has real trouble sharing fairly at that scale
- Retransmission fits naturally into TCP
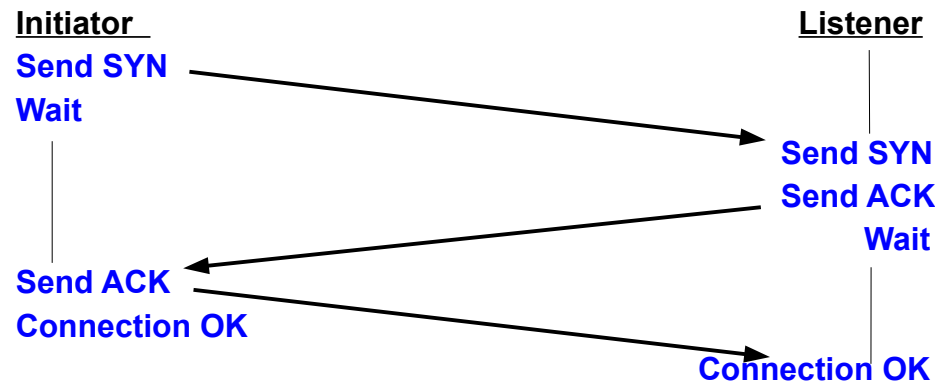 - Rate control protocols have to "break step" to deal with retransmissions
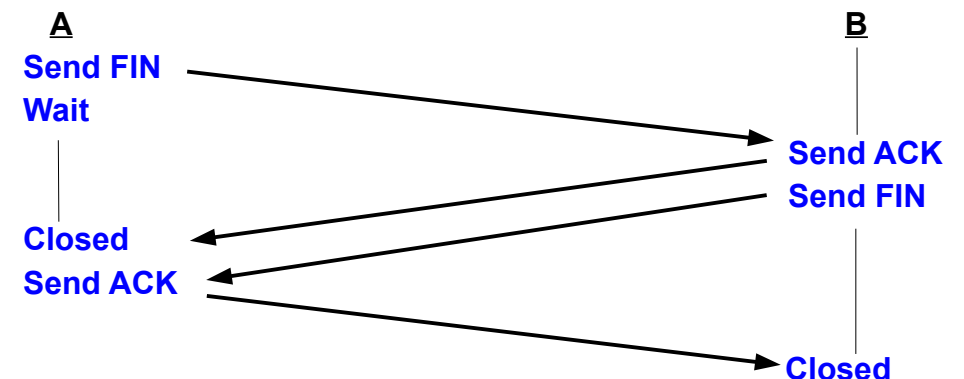
TCP connection phases

- A TCP connection has three main phases:
  - Establishment
  - Data transfer
  - Disconnection
- One end (the "listener") has to be willing to accept incoming TCP connections, and the other end (the "initiator") has to choose to start
- The listener is listening to a specific *port number* which serves as a meeting point
  - IP address + port number = *Layer 4* address

Connection establishment

Initiator   **Listener**

Send SYN
Wait
                                    Send SYN
                                    Send ACK
                                         Wait
Send ACK
Connection OK
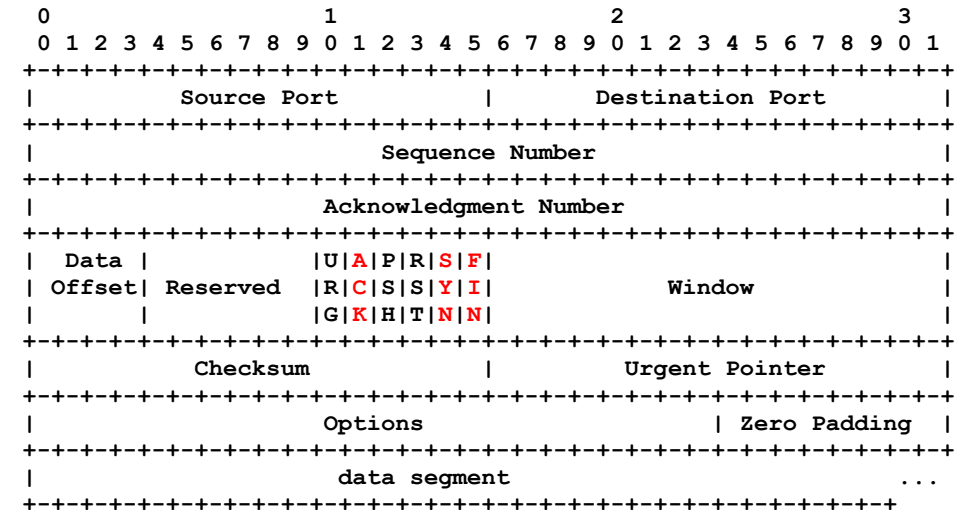   **Connection OK**

- Note that whoever goes first, there is one SYN and one ACK in each direction
  - This will work even if both initiate simultaneously
  - Listener's SYN and ACK are usually sent in a single packet

Disconnection

A   **B**

Send FIN
Wait
                                        Send ACK
                                        Send FIN
Closed
Send ACK
                                             Closed

- Doesn't matter which end closes first
  - If one end dies, a timeout will eventually close the other end
  - B may continue to send bytes from its buffer before sending its FIN

TCBs

- During connection establishment, each end creates a TCB (transmission control block) data structure
  - A TCB links the user program at each end to the TCP process
- Typical TCB contents:
  - local and remote port numbers for this connection
  - current send and receive window sizes
  - pointers into the send and receive buffers
  - status of send and receive sequence numbers
- To understand this we need to look at the TCP header format
  - The TCP header follows the IP header in a packet, when Protocol Number (IPv4) or Next Header (IPv6) is **6**

TCP header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |        Urgent Pointer         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    | Zero Padding  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    data segment                           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **Protocol Number or Next Header is 6**

TCP header fields (1)

- Port numbers - used to find TCBs at each end
- Sequence number
  - the sequence number of the first data byte in this TCP segment
  - goes up by 1 for each data byte sent on the connection
  - initialised in SYN packet (random value)
- Acknowledgement number
  - only valid in ACK packet
  - next sequence number the sender of the segment is expecting
  - in other words, sending Ack Number 12345 means "I have correctly received up to byte 12344"
  - a duplicate ACK means "I've **still** only received up to byte 12344"

TCP header fields (2)

- Data offset
  - Size of TCP header in 32 bit words
- URG - urgent bit (not too important)
- ACK – the ACK number os valid for this packet
- PSH - push bit (kick received data to the user)
- RST - reset bit (emergency disconnect)
- SYN - SYN packet ("synchronise sequence numbers")
- FIN - FIN packet ("finished," starts normal disconnect)

TCP header fields (3)

- Window
  - The number of data bytes beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept
- Checksum (next slide)
- Urgent pointer (not too important)
- Options
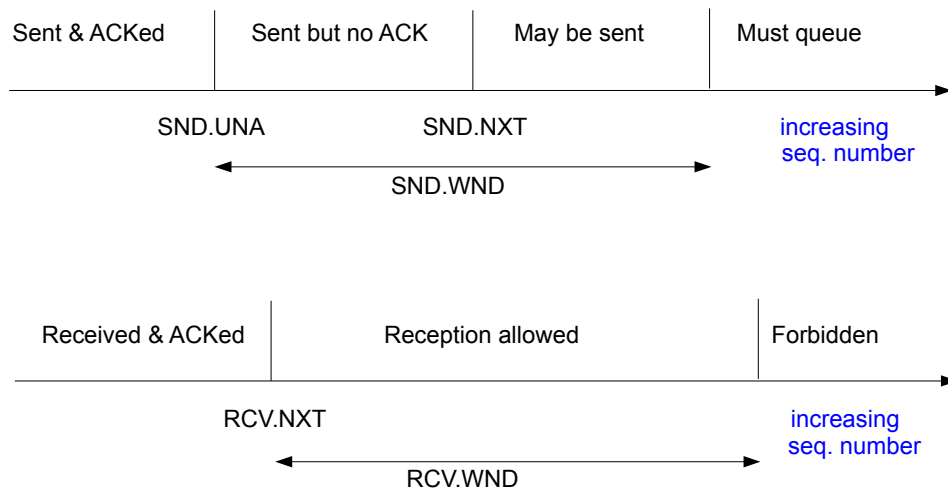 - For example, specify maximum receive segment size

TCP checksum

- *This is the primary protection against transmission errors in the Internet*
  - 16 bit one's-complement of the one's-complement sum of all 16 bit words in the TCP header and data
  - If a segment contains an odd number of bytes to be checksummed, the last byte is padded on the right with zeros to form a 16 bit word for checksum purposes. (The padding is not transmitted as part of the segment)
  - While computing the checksum, the checksum field itself is replaced with zeros
  - The checksum also covers a "pseudo header" conceptually prefixed to the TCP header. This pseudo header contains the Source & Destination IP Addresses, the Protocol or Next Header Number, and TCP segment length

Sequence number state at sender and receiver

Sent & ACKed	Sent but no ACK	May be sent	Must queue

SND.UNA   SND.NXT   increasing seq. number

SND.WND

Received & ACKed	Reception allowed	Forbidden

RCV.NXT   increasing seq. number

RCV.WND

TCP data transfer phase

- After SYN/ACK, the two ends know initial sequence numbers and initial window sizes
- Both ends may start sending, as long as they stay within the allowed sending window
  - sending a segment moves SND.NXT along
  - receiving an ACK for a given sequence number moves SND.UNA along
  - if SND.NXT = SND.UNA+SND.WND, wait
- Both ends receive
  - when a segment arrives, increase RCV.NXT and send ACK
  - if RCV.NXT reaches end of window (i.e. RCV.WND=0), only ACKs will be treated. Incoming data is discarded and not ACKed

Adjusting the Window

- The size of the send window (SND.WND) decides how much data can be sent without waiting for an ACK
  - SND.WND must be decreased when things are going slowly, and can be increased when things are going well
  - SND.WND tracks RCV.WND via ACK messages
  - The algorithm for adjusting RCV.WND is the most critical feature of a TCP implementation and has been modified many times
- See comments below on congestion control

Algorithm for Send Window to track Receive Window

- Variables
  ```
  SND.NXT  - next sequence number to be sent
  SND.WND  - current send window size
  LatestAckSeq - acknowledgement number in latest ACK
  CurrentSeq - sequence number of segment carrying ACK
  AckWindow - receiver's window size in ACK
  PreviousSeq, PreviousAck - from previous window update
  ```
- Algorithm
  ```
  if LatestAckSeq <= SND.NXT then      # waiting for ACKs
  if (PreviousSeq < CurrentSeq or      # don't use stale
      (PreviousSeq == CurrentSeq       #   window size
       and PreviousAck <= LatestAckSeq))
  then {SND.WND := AckWindow;      # update window
       PreviousSeq := CurrentSeq;
       PreviousAck := LatestAckSeq;}
  ```

Retransmission

- If an ACK does not arrive within a certain timeout, all segments since the previous ACK will be retransmitted
  - no difference whether packet was discarded due to congestion or lost due to transmission fault or checksum error
  - can be optimised with "Selective ACK" to avoid retransmitting correctly received segments
- The retransmission timeout is dynamically calculated
  - Typically by measuring a running average Round Trip Time (RTT) between sending a segment and receiving its ACK
  - Then set the timeout to, say, 2xRTT

Clarification about delay-bandwidth product

- The one-way delay in a TCP session is roughly half the RTT
- Therefore, the delay-bandwidth product is roughly
$$\frac{bandwidth \times RTT}{2}$$
- The TCP window size in a stable state is roughly
$$bandwidth \times RTT$$
  which is double the delay-bandwidth product, because the window has to allow for ACKs to come back
  - 'roughly' because the outbound delay (for data) and the return (for ACKs) will never be exactly equal
  - some web references get this wrong

- *bandwidth* = link transmission rate (b/s)

Congestion control

- TCP as described above is "greedy" - it will pump as much data as the path will take
  - With millions of connections, this leads to "congestive collapse" where saturated routers must discard most packets
- Modern TCPs use various techniques to avoid this, all of which amount to being "good neighbours"
  - Slow Start: start small and expand window gently
  - Congestion Avoidance: when duplicate ACKs indicate that later segments were lost, limit number of (re)transmissions
  - Fast Recovery: after 3 duplicate ACKs, retransmit once and wait. If still no ACK, revert to Slow Start
- Modern routers keep an eye out for greedy "cheats" and selectively discard their packets

A day in the life of a TCP session

- User A: Listen (portA)
- User B: Open (AddressA, portA)
- SYN/ACK exchange
- Data transfer phase
  - User A: Send (DataA)
  - User B: Receive (DataA)
  - User B: Send (DataB)
  - User A: Receive (DataB)
  - (repeat as required by application)
- User B: Close
- FIN/ACK exchange
- User A: Listen (portA)

Retransmission, windowing, and congestion control as needed

References

- Shay 11.4
- Any of the TCP/IP books listed for IPv4
- RFCs:
  - RFC 793, the original definition
  - Many advisory RFCs and other publications on implementation techniques to tune performance. (Implementing TCP is not for amateurs!)
  - RFC 2460 (IPv6) modifies TCP's checksum formula
  - RFC 3168 adds *Explicit Congestion Notification* to TCP and IP
  - RFC 4614 is a roadmap for TCP specifications