# IP header

![IP header diagram](image)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vers</td>
<td>Version</td>
</tr>
<tr>
<td>hdr len</td>
<td>Header length</td>
</tr>
<tr>
<td>TOS</td>
<td>Type of Service</td>
</tr>
<tr>
<td>Total Length</td>
<td></td>
</tr>
<tr>
<td>Identification</td>
<td>Identification</td>
</tr>
<tr>
<td>Fragment offset</td>
<td>Fragment offset</td>
</tr>
<tr>
<td>TTL</td>
<td>Time to Live</td>
</tr>
<tr>
<td>Protocol</td>
<td>Protocol</td>
</tr>
<tr>
<td>hdr checksum</td>
<td>hdr checksum</td>
</tr>
<tr>
<td>Source IP address</td>
<td>Source IP address</td>
</tr>
<tr>
<td>Destination IP address</td>
<td>Destination IP address</td>
</tr>
<tr>
<td>Options</td>
<td>Options</td>
</tr>
<tr>
<td>Padding</td>
<td>Padding</td>
</tr>
</tbody>
</table>
### TCP header

<table>
<thead>
<tr>
<th>Data offset</th>
<th>Reserved</th>
<th>Options</th>
<th>Padding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U R C S</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A P S S</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Y I S F</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>G K H T</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>T N N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **source port**
- **destination port**
- **sequence number**
- **acknowledgment number**
- **window**
- **checksum**
- **urgent pointer**
- **options**
- **padding**
- **data**
TCP fields

- Ports
- Seq no. – segment position in byte stream
- Ack no. – seq no. sender expects to receive next
- Data offset – # of 4-byte header & option words
- Window – willing to receive (flow control)
- Checksum
- Urgent pointer
TCP Flags

• URG – urgent data present
• ACK – ack no. valid (all but first segment)
• PSH – push data up to application immediately
• RST – reset connection
• SYN – “synchronize” establishes connection
• FIN – close connection
A TCP Connection (no data)

orchard.48150 > essex.discard:
   S 1871560457:1871560457(0) win 16384
essex.discard > orchard.48150:
   S 3249357518:3249357518(0) ack 1871560458 win 17376
orchard.48150 > essex.discard: . ack 1 win 17376
orchard.48150 > essex.discard: F 1:1(0) ack 1 win 17376
essex.discard > orchard.48150: . ack 2 win 17376
essex.discard > orchard.48150: F 1:1(0) ack 2 win 17376
orchard.48150 > essex.discard: . ack 2 win 17375
Connection establishment

- **Three-way handshake:**
  - $C \rightarrow S$: SYN, seq $S_C$
  - $S \rightarrow C$: SYN, seq $S_S$, ack $S_C + 1$
  - $C \rightarrow S$: ack $S_S + 1$

- If no program listening: server sends RST

- If server backlog exceeded: ignore SYN

- If no SYN-ACK received: retry, timeout

- **Questions:**
  - What is a SYN-bomb attack, why is it bad?
  - How do firewalls block incoming connections?
Connection termination

- **FIN bit says no more data to send**
  - Caused by close or shutdown on sending end
  - Both sides must send FIN to close a connection

- **Typical close:**
  - $A \rightarrow B$: FIN, seq $S_A$, ack $S_B$
  - $B \rightarrow A$: ack $S_A + 1$
  - $B \rightarrow A$: FIN, seq $S_B$, ack $S_A + 1$
  - $A \rightarrow B$: ack $S_B + 1$

- Can also have simultaneous close

- After last message, can $A$ and $B$ forget about closed socket?
TIME_WAIT

• Problems with closed socket
  - What if final ack is lost in the network?
  - What if the same port pair is immediately reused for a new connection? (Old packets might still be floating around.)

• Solution: “active” closer goes into TIME_WAIT
  - Active close is sending FIN before receiving one
  - After receiving ACK and FIN, keep socket around for 2MSL (twice the “maximum segment lifetime”)
Sending data

• **Data sent in MSS-sized segments**
  - Chosen to avoid fragmentation (e.g., 1460 on ethernet LAN)
  - Write of 8K might use 6 segments—PSH set on last one
  - PSH avoids unnecessary context switches on receiver

• **Sender’s OS can delay sends to get full segments**
  - Nagle algorithm: Only one unacknowledged short segment
  - TCP_NODELAY option avoids this behavior

• **Segments may arrive out of order**
  - Sequence number used to reassemble in order

• **Window achieves flow control**
  - If window 0 and sender’s buffer full, write will block or return EAGAIN
A TCP connection (3 byte echo)

orchard.38497 > essex.echo:
    S 1968414760:1968414760(0) win 16384
essex.echo > orchard.38497:
    S 3349542637:3349542637(0) ack 1968414761 win 17376
orchard.38497 > essex.echo: . ack 1 win 17376
orchard.38497 > essex.echo: P 1:4(3) ack 1 win 17376
essex.echo > orchard.38497: . ack 4 win 17376
essex.echo > orchard.38497: P 1:4(3) ack 4 win 17376
orchard.38497 > essex.echo: . ack 4 win 17376
orchard.38497 > essex.echo: F 4:4(0) ack 4 win 17376
essex.echo > orchard.38497: . ack 5 win 17376
essex.echo > orchard.38497: F 4:4(0) ack 5 win 17376
orchard.38497 > essex.echo: . ack 5 win 17375
Delayed ACKs

• **Goal: Piggy-back ACKs on data**
  - Echo server just echoes, why send separate ack first?
  - Delay ACKs for 200 msec in case application sends data
  - If more data received, immediately ACK second segment
  - Note: Never delay duplicate ACKs (if segment out of order)

• **Warning: Can interact badly with Nagle**
  - “My login has 200 msec delays”
  - Set TCP_NODELAY
Retransmission

• TCP dynamically estimates round trip time
• If segment goes unacknowledged, must retransmit
• Use exponential backoff (in case loss from congestion)
• After \( \sim 10 \) minutes, give up and reset connection
• Problem: Don’t necessarily want to halt everything for one lost packet
Congestion avoidance

• Transmit at just the right rate to avoid congestion
  - Slowly increase transmission rate to find maximum
  - One lost packet means too fast, cut rate
  - Use additive increase, multiplicative decrease

• Sender-maintained congestion window limits rate
  - Maximum amount of outstanding data:
    \[ \min(\text{congestion-window}, \text{flow-control-window}) \]

• Cut rate in half after 3 duplicate ACKs
  - Fewer duplicates may just have resulted from reordering
  - Fast retransmit: resend only lost packet

• If timeout, cut cong. window back to 1 segment
  - Slow start – exponentially increase to ss thresh
Other details

• **Persist timer**
  - Sender can block because of 0-sized receive window
  - Receiver may opens window, but ACK message lost
  - Sender keeps probing (sending one byte beyond window)

• **Path MTU discovery (optional)**
  - Dynamically discover appropriate MSS
  - Set don’t fragment bit in IP, and binary search on known sizes