Wrapping up the IP header & Reliability Concepts

Spring 2024
Sylvia Ratnasamy
CS168.io
Designing IP: two remaining topics

- IPv4  $\Rightarrow$  IPv6
- Security implications of the IP header
IPv6
IPv6

- Motivated by address exhaustion
  - Addresses *four* times as big
IPv6

- Motivated by address exhaustion
  - Addresses *four* times as big

- Took the opportunity to do some “spring cleaning”
  - Got rid of all fields that were not absolutely necessary
IPv6

- Motivated by address exhaustion
  - Addresses *four* times as big

- Took the opportunity to do some “spring cleaning”
  - Got rid of all fields that were not absolutely necessary

- Result is an elegant, if unambitious, protocol
What “clean up” would you do?

<table>
<thead>
<tr>
<th>4-bit Version</th>
<th>4-bit Header Length</th>
<th>8-bit Type of Service</th>
<th>16-bit Total Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Identification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-bit Flags</td>
<td>13-bit Fragment Offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8-bit Protocol</td>
<td>16-bit Header Checksum</td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary of Changes
Summary of Changes

- Expanded addresses
Summary of Changes

- Expanded addresses
- Eliminated checksum
Summary of Changes

- Expanded addresses
- Eliminated checksum
- Eliminated fragmentation
Summary of Changes

- Expanded addresses
- Eliminated checksum
- Eliminated fragmentation
- New options mechanism → “next header”
Options

- Recall idea: options specify advanced techniques that the router should implement for this packet
  - Example: “follow this route” (source routing)
Options

- Recall idea: options specify advanced techniques that the router should implement for this packet
  - Example: “follow this route” (source routing)
- Problem: leads to variable IP header lengths
Options

- Recall idea: options specify advanced techniques that the router should implement for this packet
  - Example: “follow this route” (source routing)
- Problem: leads to variable IP header lengths
- IPv6 approach: encode options in a separate header
Options

- Recall idea: options specify advanced techniques that the router should implement for this packet
  - Example: “follow this route” (source routing)
- Problem: leads to variable IP header lengths
- IPv6 approach: encode options in a separate header
Options

- Recall idea: options specify advanced techniques that the router should implement for this packet
  - Example: “follow this route” (source routing)
- Problem: leads to variable IP header lengths
- IPv6 approach: encode options in a separate header
Options

• Recall idea: options specify advanced techniques that the router should implement for this packet
  • Example: “follow this route” (source routing)
• Problem: leads to variable IP header lengths
• IPv6 approach: encode options in a separate header
Summary of Changes

- Expanded addresses
- Eliminated checksum
- Eliminated fragmentation
- New options mechanism → “next header”
Summary of Changes

- Expanded addresses
- Eliminated checksum
- Eliminated fragmentation
- New options mechanism $\rightarrow$ “next header”
- Eliminated header length
Summary of Changes

- Expanded addresses
- Eliminated checksum
- Eliminated fragmentation
- New options mechanism → “next header”
- Eliminated header length
- Added Flow Label
  - *Explicit* mechanism to denote related streams of packets
IPv4 and IPv6 Header Comparison
IPv4 and IPv6 Header Comparison

IPv6

<table>
<thead>
<tr>
<th>Version</th>
<th>Traffic Class</th>
<th>Flow Label</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Payload Length</th>
<th>Next Header</th>
<th>Hop Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source Address

Destination Address
IPv4 and IPv6 Header Comparison

**IPv4**

- **Version**
- **IHL**
- **Type of Service**
- **Total Length**
- **Identification**
- **Flags**
- **Fragment Offset**
- **Time to Live**
- **Protocol**
- **Header Checksum**
- **Source Address**
- **Destination Address**
- **Options**
- **Padding**

**IPv6**

- **Version**
- **Traffic Class**
- **Flow Label**
- **Payload Length**
- **Next Header**
- **Hop Limit**
- **Source Address**
- **Destination Address**
IPv4 and IPv6 Header Comparison

### IPv4

- **Version**
- **IHL**
- **Type of Service**
- **Total Length**
- **Identification**
- **Flags**
- **Fragment Offset**
- **Time to Live**
- **Protocol**
- **Header Checksum**
- **Source Address**
- **Destination Address**
- **Options**
- **Padding**

### IPv6

- **Version**
- **Traffic Class**
- **Flow Label**
- **Payload Length**
- **Next Header**
- **Hop Limit**
- **Source Address**
- **Destination Address**

**Legend:**
- Yellow: Field name kept from IPv4 to IPv6
- Orange: Fields not kept in IPv6
- Blue: Name & position changed in IPv6
- Green: New field in IPv6
Philosophy of Changes
Philosophy of Changes

- Don’t deal with problems: leave to ends
Philosophy of Changes

- Don’t deal with problems: leave to ends
  - Eliminated fragmentation
  - Eliminated checksum
Philosophy of Changes

- Don’t deal with problems: leave to ends
  - Eliminated fragmentation
  - Eliminated checksum
  - Why retain TTL?
Philosophy of Changes

- Don’t deal with problems: leave to ends
  - Eliminated fragmentation
  - Eliminated checksum
  - Why retain TTL?

- Simplify:
Philosophy of Changes

- Don’t deal with problems: leave to ends
  - Eliminated fragmentation
  - Eliminated checksum
  - Why retain TTL?

- Simplify:
  - Got rid of options
  - Got rid of IP header length
Philosophy of Changes

- Don’t deal with problems: leave to ends
  - Eliminated fragmentation
  - Eliminated checksum
  - Why retain TTL?

- Simplify:
  - Got rid of options
  - Got rid of IP header length

- While still allowing extensibility
Philosophy of Changes

- Don’t deal with problems: leave to ends
  - Eliminated fragmentation
  - Eliminated checksum
  - Why retain TTL?
- Simplify:
  - Got rid of options
  - Got rid of IP header length
- While still allowing extensibility
  - general next-header approach
Philosophy of Changes

- Don’t deal with problems: leave to ends
  - Eliminated fragmentation
  - Eliminated checksum
  - Why retain TTL?

- Simplify:
  - Got rid of options
  - Got rid of IP header length

- While still allowing extensibility
  - general next-header approach
  - general flow label for packet
Quick Security Analysis of IP Header
Focus on Sender Attacks

- Vulnerabilities a sender can exploit

- Note: not a comprehensive view of potential attacks!
  - For example, we’ll ignore attackers other than the sender
## IP Packet Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit Version</td>
<td>4-bit</td>
<td>Version</td>
</tr>
<tr>
<td>4-bit Header Length</td>
<td>4-bit</td>
<td>Header Length</td>
</tr>
<tr>
<td>8-bit Type of Service</td>
<td>8-bit</td>
<td>Type of Service</td>
</tr>
<tr>
<td>16-bit Total Length (Bytes)</td>
<td>16-bit</td>
<td>Total Length (bytes)</td>
</tr>
<tr>
<td>16-bit Identification</td>
<td>16-bit</td>
<td>Identification</td>
</tr>
<tr>
<td>3-bit Flags</td>
<td>3-bit</td>
<td>Flags</td>
</tr>
<tr>
<td>13-bit Fragment Offset</td>
<td>13-bit</td>
<td>Fragment Offset</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8-bit</td>
<td>Time to Live (TTL)</td>
</tr>
<tr>
<td>8-bit Protocol</td>
<td>8-bit</td>
<td>Protocol</td>
</tr>
<tr>
<td>16-bit Header Checksum</td>
<td>16-bit</td>
<td>Header Checksum</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td>32-bit</td>
<td>Source IP Address</td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td>32-bit</td>
<td>Destination IP Address</td>
</tr>
<tr>
<td>Options (if any)</td>
<td></td>
<td>Options</td>
</tr>
<tr>
<td>Payload</td>
<td></td>
<td>Payload</td>
</tr>
</tbody>
</table>
IP Address Integrity
IP Address Integrity

● Source address should be the sending host
  ● But who’s checking?
  ● You could send packets with any source you want
Implications of IP Address Integrity
Implications of IP Address Integrity

- Why would someone use a bogus source address?
Implications of IP Address Integrity

● Why would someone use a bogus source address?

● Attack the destination
  ● Send excessive packets, overload network path to destination
  ● But: victim can identify/filter you by the source address
  ● Hence, evade detection by putting different source addresses in the packets you send (“spoofing”)

Implications of IP Address Integrity

- Why would someone use a bogus source address?

- Attack the destination
  - Send excessive packets, overload network path to destination
  - But: victim can identify/filter you by the source address
  - Hence, evade detection by putting different source addresses in the packets you send ("spoofing")

- Or: as a way to bother the spoofed host
  - Spoofed host is wrongly blamed
  - Spoofed host may receive return traffic from the receiver(s)
Security Implications of TOS?
Security Implications of TOS?

- Attacker sets TOS priority for their traffic?
  - Network *prefers* attack traffic
Security Implications of TOS?

- Attacker sets TOS priority for their traffic?
  - Network *prefers* attack traffic

- What if the network *charges* for TOS traffic ... 
  - ... and attacker spoofs the victim’s source address?
Security Implications of TOS?

● Attacker sets TOS priority for their traffic?
  ● Network *prefers* attack traffic

● What if the network *charges* for TOS traffic ...  
  ● ... and attacker spoofs the victim’s source address?

● Today, mostly set/used by operators, not end-hosts
Security Implications of Fragmentation?
Security Implications of Fragmentation?

- Send packets larger than MTU → make routers do extra work
  - Can lead to resource exhaustion
More Security Implications

- IP options
More Security Implications

- IP options
  - Processing IP options often processed in router’s control plane (i.e., slow path) → attacker can try to overload routers
More Security Implications

- IP options
  - Processing IP options often processed in router’s control plane (i.e., slow path) → attacker can try to overload routers

- Routers often ignore options / drop packets with options
Security Implications of TTL? (8 bits)
Security Implications of TTL? (8 bits)

- Allows discovery of topology (a la traceroute)
Security Implications of TTL? (8 bits)

- Allows discovery of topology (a la traceroute)
- Some routers do not respond with a TTL exceeded error message
Other Security Implications?

- No apparent problems with `protocol` field (8 bits)
  - It’s just a de-muxing handle
  - If set incorrectly, next layer will find packet ill-formed

- Bad IP `checksum` field (16 bits) will cause packet to be discarded by the network
  - Not an effective attack...
Recap: IP header design
Recap: IP header design

- More nuanced than it first seems!
Recap: IP header design

- More nuanced than it first seems!

- Must juggle multiple goals
  - Efficient implementation
  - Security
  - Future needs
Questions?
Next topic: Reliable Transport

- **Today**: focus on concepts and mechanisms
- **Next lecture**: the design of TCP
Reliable Delivery Is Necessary

- Many app semantics involve reliable delivery
  - E.g., file transfer
Reliable Delivery Is Necessary

- Many app semantics involve reliable delivery
  - E.g., file transfer

- Challenge: building a reliable service on top of unreliable packet delivery
Reliable Delivery Is Necessary

● Many app semantics involve reliable delivery
  ● E.g., file transfer

● Challenge: building a reliable service on top of unreliable packet delivery

● Bridging the gap between
  ● the abstractions application designers want
  ● the abstractions networks can easily support
Semantics of correct delivery
Semantics of correct delivery

- At network layer: *best-effort* delivery
Semantics of correct delivery

- At network layer: *best-effort* delivery
- At transport layer: *at-least-once* delivery
Semantics of correct delivery

- At network layer: *best-effort* delivery
- At transport layer: *at-least-once* delivery
- At the app layer: *exactly-once* delivery
Goals For Reliable Transfer
(at the Transport Layer)
Goals For Reliable Transfer
(at the Transport Layer)

● **Correctness**
  - The destination receives every packet, uncorrupted, at least once
Goals For Reliable Transfer
(at the Transport Layer)

● **Correctness**
  ● The destination receives every packet, uncorrupted, at least once

● **Timeliness**
  ● Minimize time until data is transferred
Goals For Reliable Transfer (at the Transport Layer)

- **Correctness**
  - The destination receives every packet, uncorrupted, at least once

- **Timeliness**
  - Minimize time until data is transferred

- **Efficiency**
  - Would like to minimize use of bandwidth
  - i.e., avoid sending packets unnecessarily
Note!

- A reliability protocol (at the transport layer) can “give up”, but must announce this to application
  - E.g., if the network is partitioned

- But it can never falsely claim to have delivered a packet
A best-effort network
A best-effort network

- Packets can be lost (i.e., dropped)
A best-effort network

- Packets can be lost (i.e., dropped)
- Packets can be corrupted
A best-effort network

● Packets can be lost (i.e., dropped)
● Packets can be corrupted
● Packets can be reordered
A best-effort network

- Packets can be lost (i.e., dropped)
- Packets can be corrupted
- Packets can be reordered
- Packets can be delayed
A best-effort network

- Packets can be lost (i.e., dropped)
- Packets can be corrupted
- Packets can be reordered
- Packets can be delayed
- Packets can be duplicated
Quick reminder
Quick reminder
Quick reminder
Quick reminder

Sender

\[\text{time}\]

Receiver

One-way delay
Quick reminder

- RTT (round-trip time)
- One-way delay

Sender

Receiver
Designing a reliability protocol
Designing a reliability protocol

- Let’s start with the single packet case
Designing a reliability protocol

- Let’s start with the single packet case

- Remember
  - Packets can be dropped
  - Packets can be corrupted
  - Packets can be reordered
  - Packets can be delayed
  - Packets can be duplicated
  - ....
start timer

Sender

time

Receiver
start timer

Sender

time

Receiver
start timer

timeout

ack

time

Sender

Receiver
start timer

timeout

time

Sender

Receiver

ack

ack

☒
start timer

Note: destination received the packet twice (and that's fine!)
How to set timers?

- Too long: will delay delivery
- Too short: unnecessary retransmissions
How to set timers?

- Too long: will delay delivery
- Too short: unnecessary retransmissions
- Ideally, proportional to the RTT (next lecture)
How to set timers?

- Too long: will delay delivery
- Too short: unnecessary retransmissions

- Ideally, proportional to the RTT (next lecture)

- Non-trivial to get right in practice
  - RTTs vary across paths (10µs to 100s ms)
  - RTT of a fixed path varies over time (load, congestion)
How to set timers?

- Too long: will delay delivery
- Too short: unnecessary retransmissions

- Ideally, proportional to the RTT (next lecture)

- Non-trivial to get right in practice
  - RTTs vary across paths (10µs to 100s ms)
  - RTT of a fixed path varies over time (load, congestion)

- Hence, often used as last resort
We said

- Packets can be lost (data or ACKs)
- Packets can be corrupted
- Packets can be delayed
- Packets can be duplicated
- Packets can be reordered
time

Sender

$\rightarrow$

$\rightarrow$

checksum
drop pkt

Receiver

time
We said

- Packets can be lost (data or ACKs)
- Packets can be corrupted
- Packets can be delayed
- Packets can be duplicated
- Packets can be reordered
start timer

timeout

Sender

time

Receiver
Note: sender received the ACK twice (and that’s fine!)
Our solution handles delayed packets! (no additional mechanism needed)

Note: sender received the ACK twice (and that's fine!)
We said

- Packets can be lost (data or ACKs)
- Packets can be corrupted
- Packets can be delayed
- Packets can be duplicated
- Packets can be reordered
time

Sender

Receiver

ack
Why would the network even duplicate a packet? Usually, because of link-level reliability gone wrong (very rare)
Looks no different from our previous scenarios
Looks no different from our previous scenarios

Our solution also handles packet duplicates!
We said

- Packets can be lost (data or ACKs)
- Packets can be corrupted
- Packets can be delayed
- Packets can be duplicated
- Packets can be reordered
Have solved the single packet case!
Have solved the single packet case!

● Sender:
  ● Send packet
  ● Set timer
  ● If no ACK arrives before the timer goes off, resend packet
    ● And reset timer
Have solved the single packet case!

- **Sender:**
  - Send packet
  - Set timer
  - If no ACK arrives before the timer goes off, resend packet
    - And reset timer

- **Receiver**
  - When receiver gets packet, sends ACK
What have we learnt?
What have we learnt?

- Building blocks for a solution
  - **Checksums**: to detect corruption
  - **Feedback** from receiver: positive/negative (ack/nack)
  - **Retransmissions**: sender resends packets
  - **Timeouts**: when to resend a packet
What have we learnt?

● Building blocks for a solution
  ● **Checksums**: to detect corruption
  ● **Feedback** from receiver: positive/negative (ack/nack)
  ● **Retransmissions**: sender resends packets
  ● **Timeouts**: when to resend a packet

● **Semantics** of a solution: “at least once”
  ● Receiver can receive the same packet more than once
  ● Sender can see the same ack/nack more than once
Questions?
Next: reliably send multiple packets
Next: reliably send multiple packets

- Will need +1 design component: sequence numbers!
Data packets carry sequence numbers; and ACKs indicate what sequence numbers have been received.
Next: reliably send multiple packets

- Will need +1 design component: sequence numbers!
- We now have all the necessary building blocks!
The “Stop and Wait” protocol

- Use our single-packet solution repeatedly
The “Stop and Wait” protocol

- Use our single-packet solution repeatedly
  - Wait for packet $i$ to be acknowledged before sending $i+1$
The “Stop and Wait” protocol

- Use our single-packet solution repeatedly
  - Wait for packet $i$ to be acknowledged before sending $i+1$
- We have a correct reliable delivery protocol!
The “Stop and Wait” protocol

- Use our single-packet solution repeatedly
  - Wait for packet $i$ to be acknowledged before sending $i+1$

- We have a correct reliable delivery protocol!

- Probably the world’s most inefficient one
  - Max throughput $\sim$ one packet per RTT
Idea: have multiple packets “in flight”
(send additional packets while waiting for ACKs to come in)
Window-based Algorithms
Window-based Algorithms

- Basic idea: allow $W$ packets “in flight” at any time
  - $W$ is the size of the window
Window-based Algorithms

- Basic idea: allow \( W \) packets “in flight” at any time
  - \( W \) is the size of the window

- Hence, a simple algorithm (at sender)
  - Send \( W \) packets
  - When one gets ACK’ed, send the next packet in line
Start with window = \( \{1,2,3,4\} \)
Example with $W=4$

On receiving $\text{ack}(1)$, $\text{window} = \{2, 3, 4, 5\}$
Example with $W=4$

On receiving $\text{ack}(2)$, window = \{3,4,5,6\}
Reliably sending many packets

- Will need +1 design component: sequence numbers!
- We now have all the *necessary* building blocks
Reliably sending many packets

- Will need +1 design component: sequence numbers!
- We now have all the necessary building blocks
- Plus one more, for efficiency (performance)
  - Window
New Design Considerations
New Design Considerations

- Window size
  - How many in-flight packets do we want?
New Design Considerations

- **Window size**
  - How many in-flight packets do we want?

- **Nature of feedback**
  - Can we do better than ACKing one packet at a time?

- **Detection of loss**
  - Can we do better than waiting for timeouts?
New Design Considerations

- Window size
  - How many in-flight packets do we want?

- Nature of feedback
  - Can we do better than ACKing one packet at a time?

- Detection of loss
  - Can we do better than waiting for timeouts?

- Response to loss
  - Which packet should sender resend?
How big should the window be?
How big should the window be?

- Pick window size $W$ to balance three goals
How big should the window be?

- Pick window size $W$ to balance three goals
  - Take advantage of network capacity ("fill the pipe")
How big should the window be?

- Pick window size $W$ to balance three goals
  - Take advantage of network capacity (“fill the pipe”)
  - But don’t overload links (congestion control)
How big should the window be?

- Pick window size $W$ to balance three goals
  - Take advantage of network capacity (“fill the pipe”)
  - But don’t overload links (congestion control)
  - And don’t overload the receiver (flow control)
How big should the window be?

- Pick window size $W$ to balance three goals
  - Take advantage of network capacity (“fill the pipe”)
  - But don’t overload links (congestion control)
  - And don’t overload the receiver (flow control)

- If we ignore all but the first goal then we want to keep the sender always sending (ideal case)
How big should the window be?

- Pick window size $W$ to balance three goals
  - Take advantage of network capacity ("fill the pipe")
  - But don’t overload links (congestion control)
  - And don’t overload the receiver (flow control)

- If we ignore all but the first goal then we want to keep the sender always sending (ideal case)
  - $W$ should allow sender to transmit for entire RTT
    - From sending first packet until receive first ACK
Window = 4

wasted time (BW)
More desirable window size

Window = 8
What Does This Mean?
What Does This Mean?

- Let B be the minimum ("bottleneck") link bandwidth along the path
  - Obviously shouldn’t send faster than that
  - Don’t want to send slower than that (for first goal)
What Does This Mean?

- Let $B$ be the minimum ("bottleneck") link bandwidth along the path
  - Obviously shouldn’t send faster than that
  - Don’t want to send slower than that (for first goal)

- Want the sender to send at rate $B$ for the duration of RTT
  - I.e., ACK for the first packet arrives at the sender, just as the last of $W$ packets leaves
What Does This Mean?

- Let $B$ be the minimum ("bottleneck") link bandwidth along the path
  - Obviously shouldn’t send faster than that
  - Don’t want to send slower than that (for first goal)

- Want the sender to send at rate $B$ for the duration of RTT
  - I.e., ACK for the first packet arrives at the sender, just as the last of $W$ packets leaves

- Hence, condition: $W \times \text{Packet-Size} \sim RTT \times B$
  - E.g., for a path with RTT=1 second and bottleneck $B = 8 \text{ Mbits/second}$, if packet size $= 100 \text{ Bytes}$, we want a window size $W = 10,000 \text{ packets}$
Setting W to be one RTT of packets

Sender
{1,2,3,4,5,6}

Receiver
New Design Considerations

- Window size
  - How many in-flight packets do we want?

- Nature of feedback
  - Can we do better than ACKing one packet at a time?

- Detection of loss
  - Can we do better than waiting for timeouts?

- Response to loss
  - Which packet should sender resend?
ACKs: design options

- Individual packet ACKs (our design so far)
  - On receiving packet $i$, send $\text{ack}(i)$
Unnecessary and avoidable retransmission!
ACKs: design options

- **Individual packet ACKs (our design so far)**
  - On receiving packet $i$, send ack($i$)

- **Full Information ACKs**
  - Give highest cumulative ACK plus any additional packets received ("everything up to #12 and #14, #15")
Sender learns #2 was received when it receives the 3rd and 4th ACK
ack(<=1)
ack(<=1 plus 3)
ack(<=1 plus 3, 5)
Problem: ACK info is getting long!
ACKs: design options

- Individual packet ACKs (our design so far)
  - On receiving packet $i$, send $\text{ack}(i)$

- Full Information ACKs
  - Give highest cumulative ACK plus any additional packets received ("everything up to #12 and #14, #15")

- Cumulative ACKs
  - ACK the highest sequence number for which all previous packets have been received
Same behavior as full-information ACKs
ACK info scales better
(but is more ambiguous!)
Recap: ACK tradeoffs
Recap: ACK tradeoffs

- Individual
  - Pro: compact; simple
  - Con: loss of ACK packet *always* requires a retransmission
Recap: ACK tradeoffs

● **Individual**
  ● Pro: compact; simple
  ● Con: loss of ACK packet *always* requires a retransmission

● **Full Information**
  ● Pro: complete info on data packets; more resilient to ACK loss
  ● Con: Could require sizable overhead in bad cases
Recap: ACK tradeoffs

- **Individual**
  - Pro: compact; simple
  - Con: loss of ACK packet *always* requires a retransmission

- **Full Information**
  - Pro: complete info on data packets; more resilient to ACK loss
  - Con: Could require sizable overhead in bad cases

- **Cumulative**
  - Pro: compact; more resilient to ACK loss (vs. individual ACKs)
  - Con: Incomplete info on which data packets arrived
New Design Considerations

- **Window size**
  - How many in-flight packets do we want?

- **Nature of feedback**
  - Can we do better than ACKing one packet at a time?

- **Detection of loss**
  - Can we do better than waiting for timeouts?

- **Response to loss**
  - Which packet should the sender resend?
Detecting Loss
Detecting Loss

- If packet times out, assume it is lost...

- How else can you detect loss?
Detecting Loss

- If packet times out, assume it is lost...
- How else can you detect loss?
- When ACKs for k “subsequent packets” arrive
  - E.g., only packet 5 is lost, will receive ACKs for 6, 7, ...
  - E.g., if k=3, retransmit 5 after we receive ACKs for 6, 7, 8
  - Details look a little different for each ACK option (next slides)
Loss with individual ACKs

- Assume packet 5 is lost, but no others
Loss with individual ACKs

- Assume packet 5 is lost, but no others

- Stream of ACKs will be:
  - 1
  - 2
  - 3
  - 4
  - 6
  - 7
  - 8
  - ....
Loss with individual ACKs

● Assume packet 5 is lost, but no others

● Stream of ACKs will be:
  ● 1
  ● 2
  ● 3
  ● 4
  ● 6
  ● 7
  ● 8
  ● ....
Loss with individual ACKs

- Assume packet 5 is lost, but no others

- Stream of ACKs will be:
  - 1
  - 2
  - 3
  - 4
  - 6
  - 7
  - 8
  - ....

Declare packet 5 lost! (Because received k=3 subsequent ACKs)
Loss with full information

- Same story, except that the “hole” is explicit in each ACK
Loss with full information

- Same story, except that the “hole” is explicit in each ACK

- Stream of ACKs will be:
  - Up to 1
  - Up to 2
  - Up to 3
  - Up to 4
  - Up to 4, plus 6
  - Up to 4, plus 6,7
  - Up to 4, plus 6,7,8
Loss with full information

- Same story, except that the “hole” is explicit in each ACK

- Stream of ACKs will be:
  - Up to 1
  - Up to 2
  - Up to 3
  - Up to 4
  - Up to 4, plus 6
  - Up to 4, plus 6,7
  - Up to 4, plus 6,7,8

Declare packet 5 lost! (Received k=3 subsequent ACKs)
Loss with cumulative ACKs

- Assume packet 5 is lost, but no others
Loss with cumulative ACKs

- Assume packet 5 is lost, but no others

- Stream of ACKs will be:
  - Up to 1
  - Up to 2
  - Up to 3
  - Up to 4
  - Up to 4 (sent when packet 6 arrives)
  - Up to 4 (sent when packet 7 arrives)
  - Up to 4 (sent when packet 8 arrives)
Loss with cumulative ACKs

- Assume packet 5 is lost, but no others

- Stream of ACKs will be:
  - Up to 1
  - Up to 2
  - Up to 3
  - Up to 4
  - Up to 4 (sent when packet 6 arrives)
  - Up to 4 (sent when packet 7 arrives)
  - Up to 4 (sent when packet 8 arrives)
Loss with cumulative ACKs

- Assume packet 5 is lost, but no others

- Stream of ACKs will be:
  - Up to 1
  - Up to 2
  - Up to 3
  - Up to 4
  - Up to 4 (sent when packet 6 arrives)
  - Up to 4 (sent when packet 7 arrives)
  - Up to 4 (sent when packet 8 arrives)

- Packet 5 lost! (Received k=3 dupACKs)
New Design Considerations

● Window size
  ● How many in-flight packets do we want?

● Nature of feedback
  ● Can we do better than ACKing one packet at a time?

● Detection of loss
  ● Can we do better than waiting for timeouts?

● Response to loss
  ● Which packet should sender resend?
Response to loss

- On timeout, always retransmit corresponding packet
Response to loss

- On timeout, always retransmit corresponding packet
- What about when our ACK-based rule fires?
Response to loss

- On timeout, always retransmit corresponding packet

- What about when our ACK-based rule fires?
  - Retransmit unACKed packet, but which one?
Response to loss

- On timeout, always retransmit corresponding packet

- What about when our ACK-based rule fires?
  - Retransmit unACKed packet, but which one?
  - Decision is clear with individual and full-info ACKs
Response to loss

- On timeout, always retransmit corresponding packet

- What about when our ACK-based rule fires?
  - Retransmit unACKed packet, but which one?
  - Decision is clear with individual and full-info ACKs
  - Decision is clear with cumulative ACKs and a single packet loss
Response to loss

- On timeout, always retransmit corresponding packet

- What about when our ACK-based rule fires?
  - Retransmit unACKed packet, but which one?
  - Decision is clear with individual and full-info ACKs
  - Decision is clear with cumulative ACKs and a single packet loss
  - But can be ambiguous with cumulative ACKs and multiple losses
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1  2  3  4  5  6  7  8
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

- ACK 4 arrives
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

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

- ACK 4 arrives ➔ send 9
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1 2 3 4 5 6 7 8 9

- ACK 4 arrives => send 9
- ACK 6 arrives
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1  2  3  4  5  6  7  8  9  10

- ACK 4 arrives → send 9
- ACK 6 arrives → send 10
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1 2 3 4 5 6 7 8 9 10

- ACK 4 arrives $\rightarrow$ send 9
- ACK 6 arrives $\rightarrow$ send 10
- ACK 7 arrives (3rd ACK for subsequent packet)
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

![Packet Diagram]

- ACK 4 arrives $\rightarrow$ send 9
- ACK 6 arrives $\rightarrow$ send 10
- ACK 7 arrives $\rightarrow$ resend 3, send 11
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1  2  3  4  5  6  7  8  9  10  11

- ACK 4 arrives → send 9
- ACK 6 arrives → send 10
- ACK 7 arrives → **resend 3**, send 11
- ACK 8 arrives
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”
Response with individual ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1 2 3 4 5 6 7 8 9 10 11 12

- ACK 4 arrives → send 9
- ACK 6 arrives → send 10
- ACK 7 arrives → resend 3, send 11
- ACK 8 arrives → resend 5, send 12
- ACK 9 arrives → send 13, and so on...
Response with full-info ACKs

- Similar behavior as with Individual ACKs
Response with cumulative ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1  2  3  4  5  6  7  8
Response with cumulative ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1  2  3  4  5  6  7  8

- (for packet 4) ACK 2
Response with cumulative ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1 2 3 4 5 6 7 8

(for packet 4) ACK 2

#duplicate ACKs = 1
Response with cumulative ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1 2 3 4 5 6 7 8 9

(for packet 4) ACK 2 → send 9

#duplicate ACKs = 1
Response with cumulative ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1, 2 have been ACKed
  - 3-8 are “in flight”

1  2  3  4  5  6  7  8  9

#duplicate ACKs = 2

- (for packet 4) ACK 2 → send 9
- (for packet 6) ACK 2
Response with cumulative ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

\[1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\]

#duplicate ACKs = 2

- (for packet 4) ACK 2 \(\Rightarrow\) send 9
- (for packet 6) ACK 2 \(\Rightarrow\) send 10
Response with cumulative ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1 2 3 4 5 6 7 8 9 10

- (for packet 4) ACK 2 \rightarrow \text{send 9}
- (for packet 6) ACK 2 \rightarrow \text{send 10}
- (for packet 7) ACK 2

#duplicate ACKs = 3
Response with cumulative ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1  2  3  4  5  6  7  8  9  10  11

#duplicate ACKs = 3

- (for packet 4) ACK 2 → send 9
- (for packet 6) ACK 2 → send 10
- (for packet 7) ACK 2 → resend 3, send 11
Response with cumulative ACKs

- Consider a sender with a window size = 6 & k=3
  - Packets 1,2 have been ACKed
  - 3-8 are “in flight”

1 2 3 4 5 6 7 8 9 10 11 12...

- (for packet 4) ACK 2 → send 9
- (for packet 6) ACK 2 → send 10
- (for packet 7) ACK 2 → resend 3, send 11
- (for packet 8,9,10) ACK 2 → unclear what packet to resend!

#duplicate ACKs = 4+
Response with cumulative ACKs
Response with cumulative ACKs

- Cumulative ACKs don’t tell the sender exactly which packets were received
Response with cumulative ACKs

- Cumulative ACKs don’t tell the sender exactly which packets were received

- Can tell *how many* packets to send
  - Because #dupACKs tells us how many pkts were *received*
Response with cumulative ACKs

- Cumulative ACKs don’t tell the sender exactly which packets were received

- Can tell *how many* packets to send
  - Because #dupACKs tells us how many pkts were *received*

- But not *which ones* to (re)send
  - Ambiguity leads to ad-hoc heuristics
Response with cumulative ACKs

- Cumulative ACKs don’t tell the sender exactly which packets were received

- Can tell how many packets to send
  - Because #dupACKs tells us how many pkts were received

- But not which ones to (re)send
  - Ambiguity leads to ad-hoc heuristics
Response with cumulative ACKs

- Cumulative ACKs don’t tell the sender exactly which packets were received

- Can tell how many packets to send
  - Because #dupACKs tells us how many pkts were received

- But not which ones to (re)send
  - Ambiguity leads to ad-hoc heuristics

- Unfortunately, TCP uses cumulative ACKs...
Taking Stock...
Taking Stock...

- We’ve identified our design building blocks
  - Checksums
  - ACK/NACKs
  - Timeouts
  - Retransmissions
  - Sequence numbers
  - Windows
Taking Stock...

- We’ve identified our design building blocks
  - Checksums
  - ACK/NACKs
  - Timeouts
  - Retransmissions
  - Sequence numbers
  - Windows

- And discussed tradeoffs in how to apply them
  - Individual vs. Full vs. Cumulative ACKs
  - Timeout vs. ACK-driven loss detection
From design options to design
From design options to design

- Can put together a variety of reliability protocols from our building blocks!
  - We saw one already: Stop-and-Wait
  - Another possibility: “Go-Back-N” (in section)
  - TCP implements yet another (next lecture)
From design options to design

- Can put together a variety of reliability protocols from our building blocks!
  - We saw one already: Stop-and-Wait
  - Another possibility: “Go-Back-N” (in section)
  - TCP implements yet another (next lecture)

- More important that you know how to design and evaluate a reliability protocol, than that you memorize the details of any one implementation!
Preview: what does TCP do?

- Uses most of our building blocks w/ a few diffs.
  - Checksums
  - ACKs (no explicit NACKs)
  - “Sliding” Windows
  - Sequence numbers → measured in byte offsets
  - Cumulative ACKs (and counting dupACKs)
  - Option for a form of full-information ACKs (SACK)
  - Timers (w/ timer estimation algorithm)
Final thought: other approaches?
Final thought: other approaches?

- Sender **encodes** the data to be resilient to loss
  - Basic idea: add some redundancy to data / packet stream
  - E.g., take k packets, encode as n (>k) packets
  - Original packets can be recovered if any k’ of n packets are received (n > k’ > k)
  - Efficiency depends on k’/k
Final thought: other approaches?

- **Sender** *encodes* the data to be resilient to loss
  - Basic idea: add some redundancy to data / packet stream
  - E.g., take $k$ packets, encode as $n$ (>k) packets
  - Original packets can be recovered if any $k'$ of $n$ packets are received ($n > k' > k$)
  - Efficiency depends on $k'/k$

- Vast literature on coding schemes
  - E.g., fountain codes, raptor codes, ...
Final thought: other approaches?

- Sender **encodes** the data to be resilient to loss
  - Basic idea: add some redundancy to data / packet stream
  - E.g., take k packets, encode as n (>k) packets
  - Original packets can be recovered if any k’ of n packets are received (n > k’ > k)
  - Efficiency depends on k’/k

- Vast literature on coding schemes
  - E.g., fountain codes, raptor codes, ...

- Historically not used very much but that could change...
Questions?
Backup#2: We need sequence numbers with stop-and-wait