Wrapping up the IP header
&
Reliability Concepts

Spring 2024

Sylvia Ratnasamy

CS168.io

Designing IP: two remaining topics

- IPv4 → IPv6
- Security implications of the IP header

IPv6

IPv6

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

IPv6

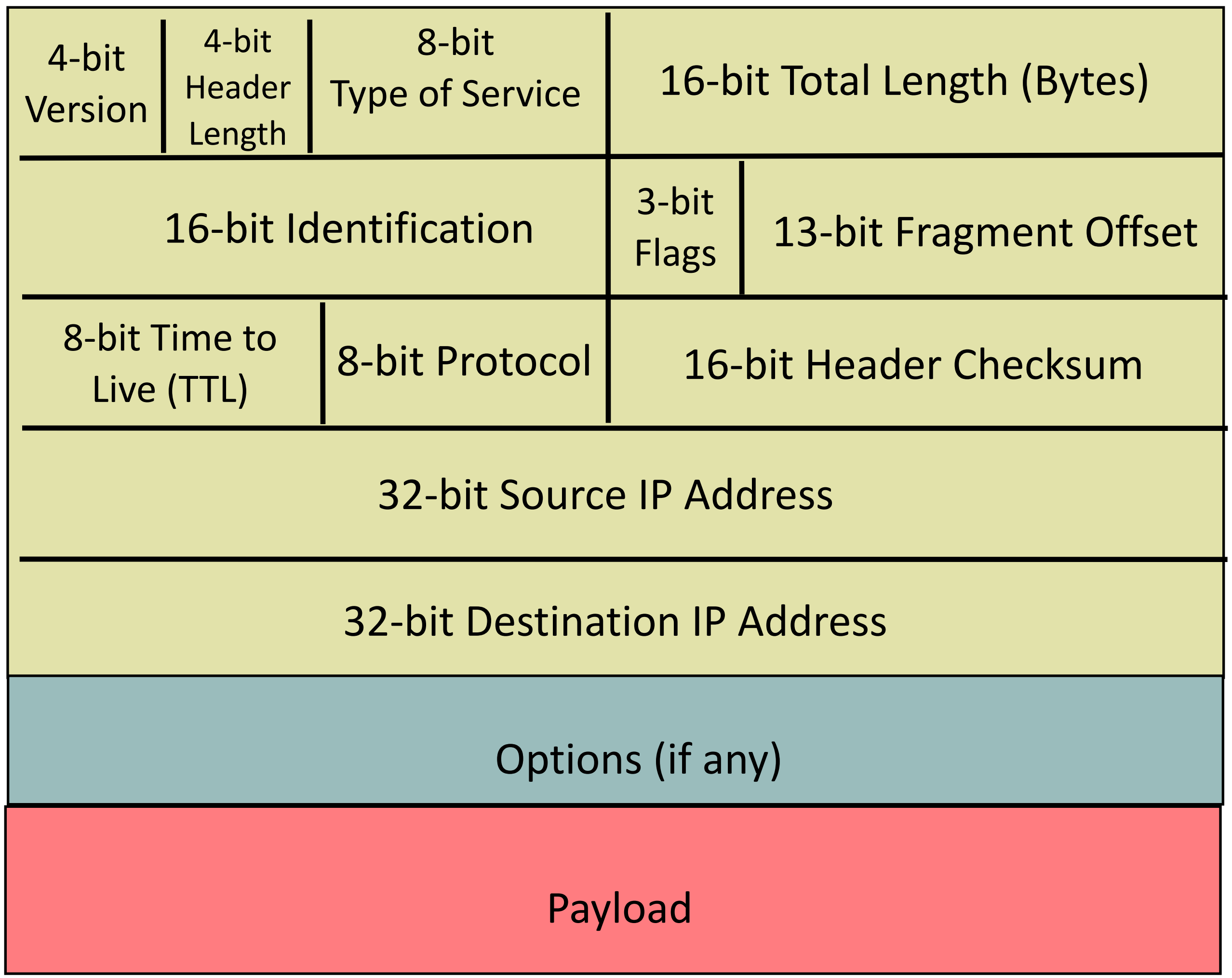
- Motivated by address exhaustion
  - Addresses ***four*** times as big

- 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?



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

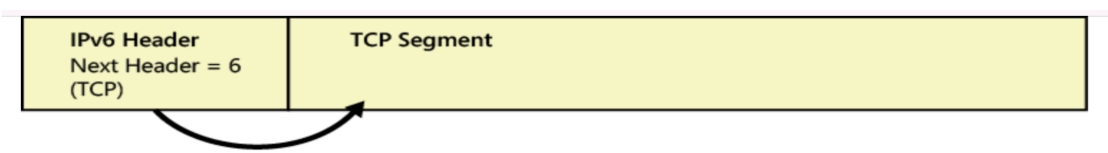
- Recall idea: options specify advanced techniques that the router should implement for this packet
  - Example: “follow this route” (source routing)
- Problem: leads to variable IP header lengths

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

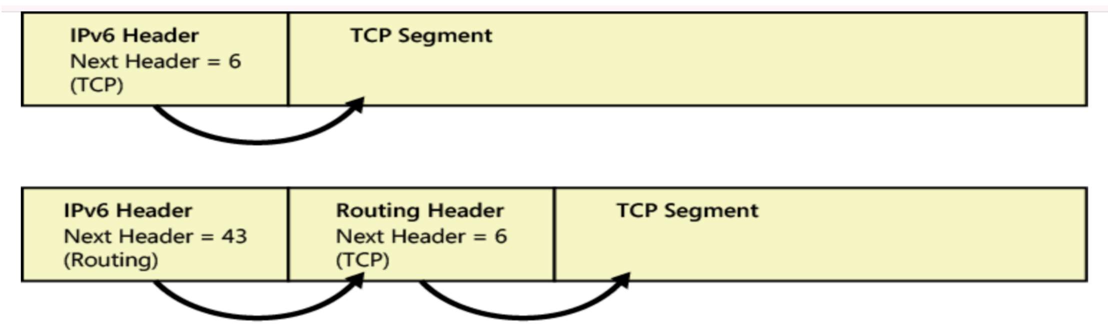
Options

- 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



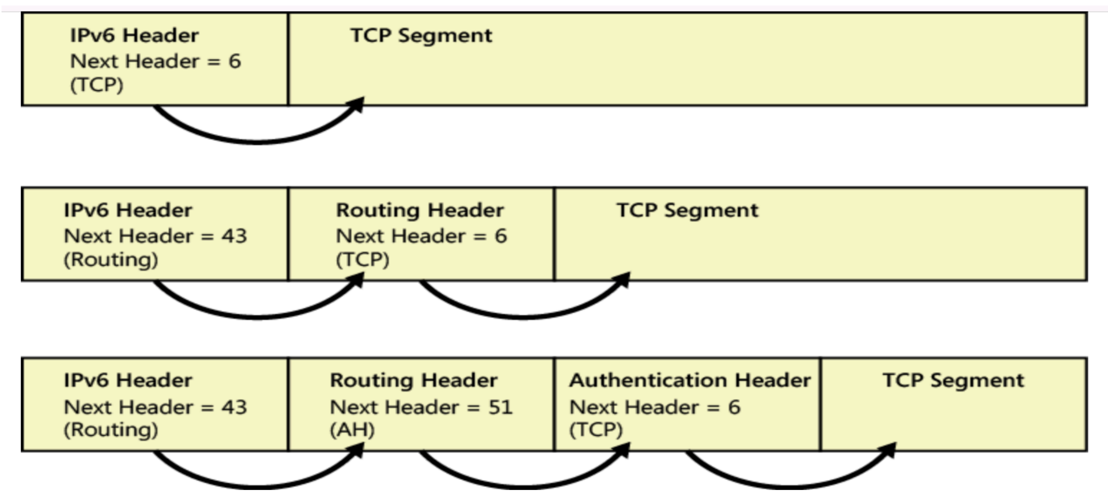
Options

- 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

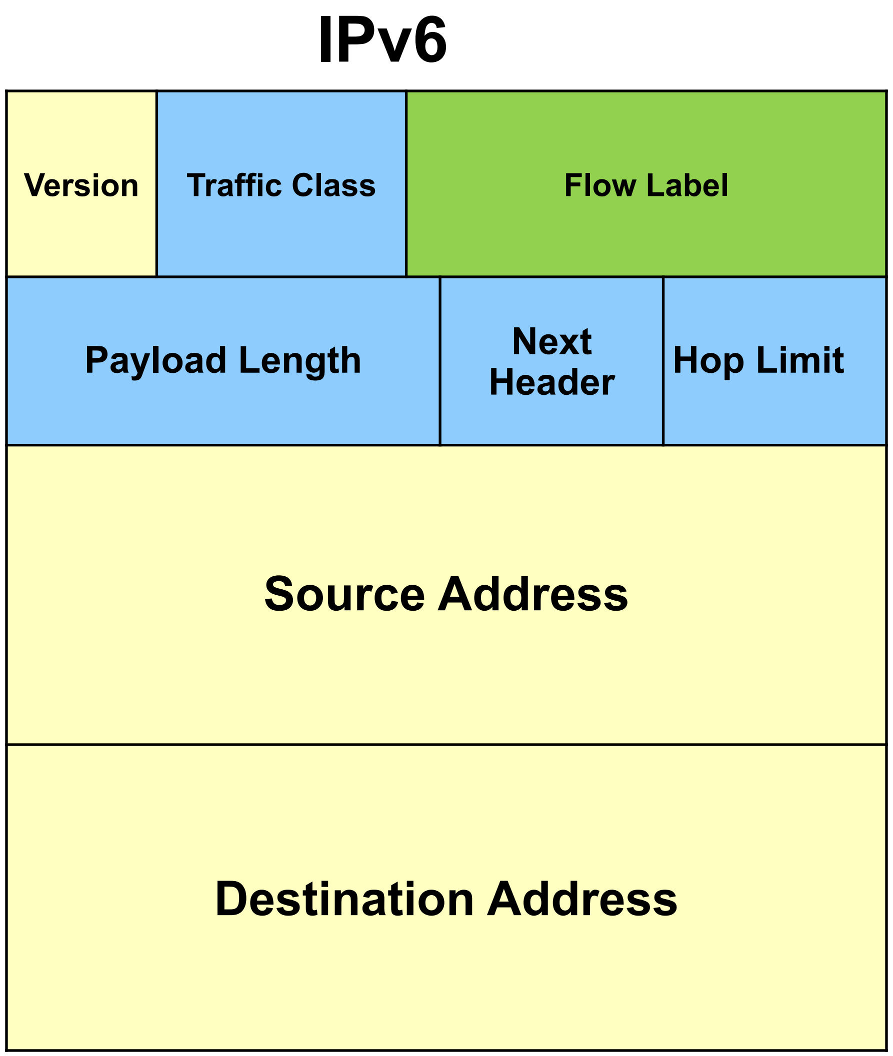
- Expanded addresses
- Eliminated checksum
- Eliminated fragmentation
- New options mechanism → "next header"
- Eliminated header length

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

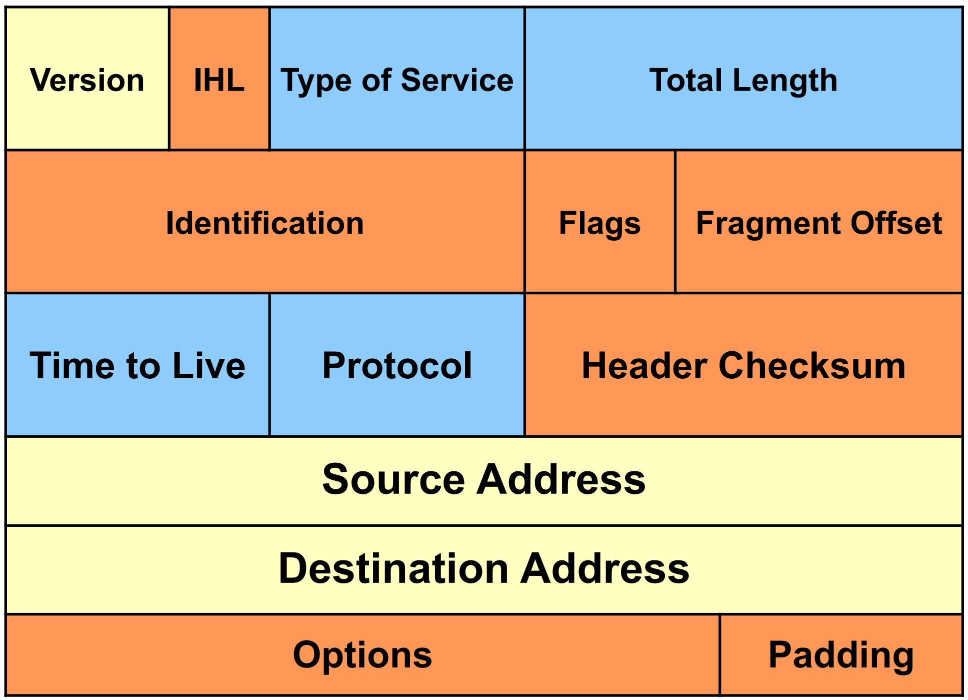
IPv4 and IPv6 Header Comparison

IPv4 and IPv6 Header Comparison

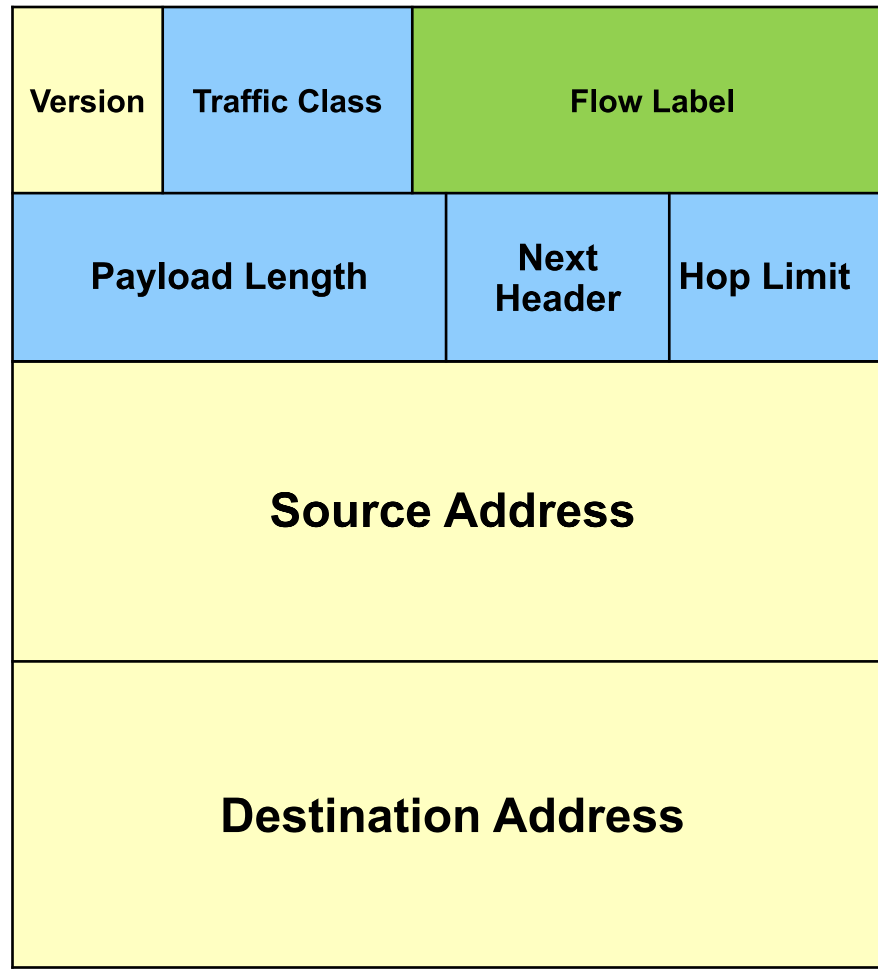


IPv4 and IPv6 Header Comparison

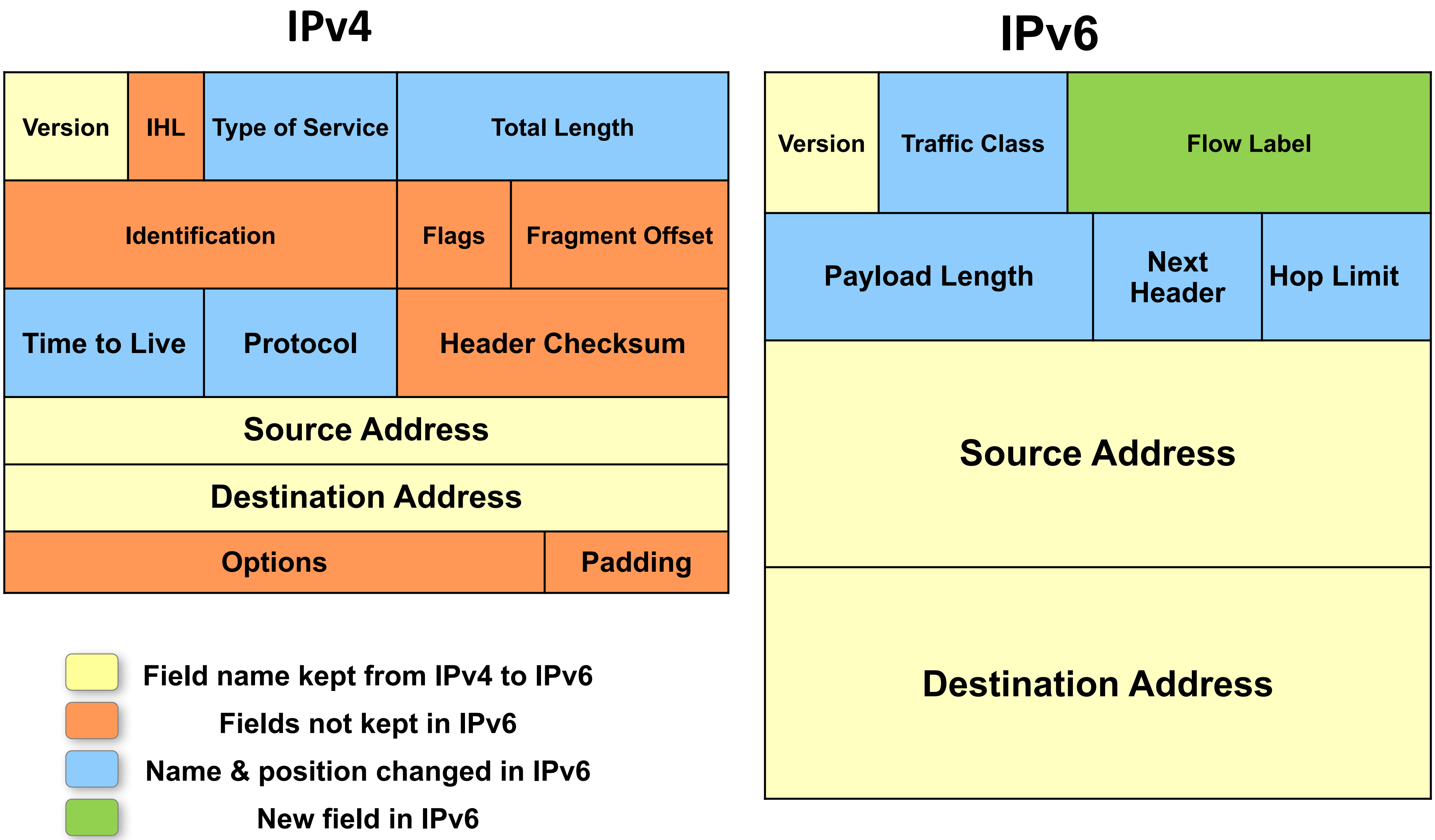
IPv4



IPv6



IPv4 and IPv6 Header Comparison



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

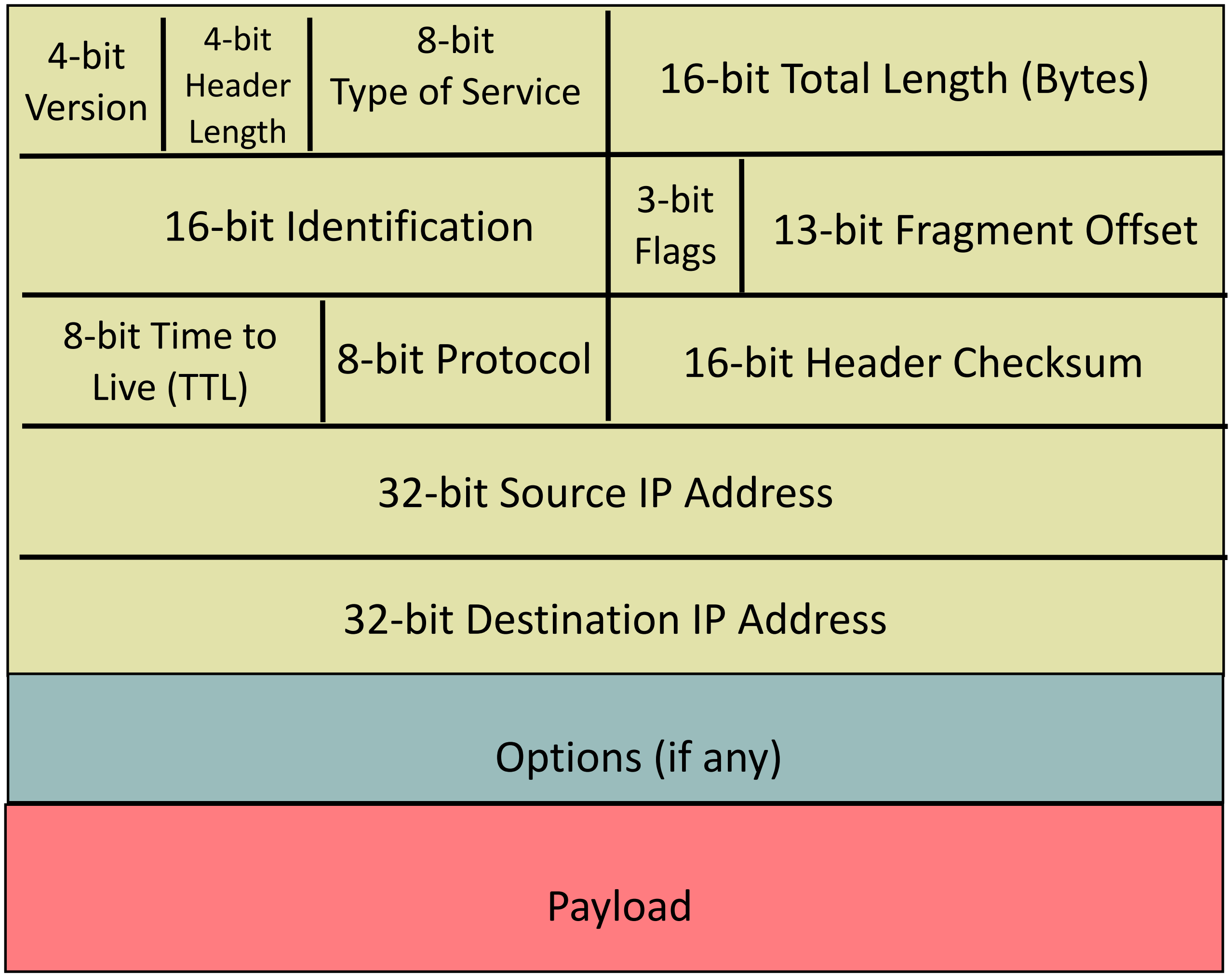
- Don’t deal with problems: leave to ends
  - Eliminated fragmentation
  - Eliminated checksum
  - Why retain TTL?
- Simplify:
  - Got rid of options
  - Got rid of IP header length
- While still allowing extensibility
  - general next-header approach
  - general flow label for packet

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



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

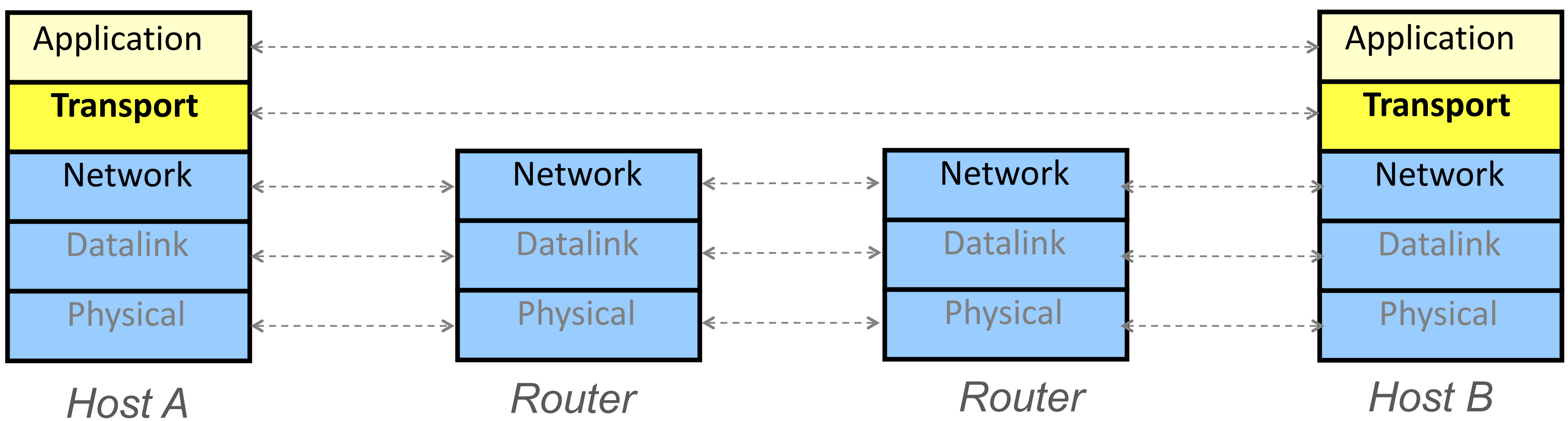
Reliable Delivery Is Necessary

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

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

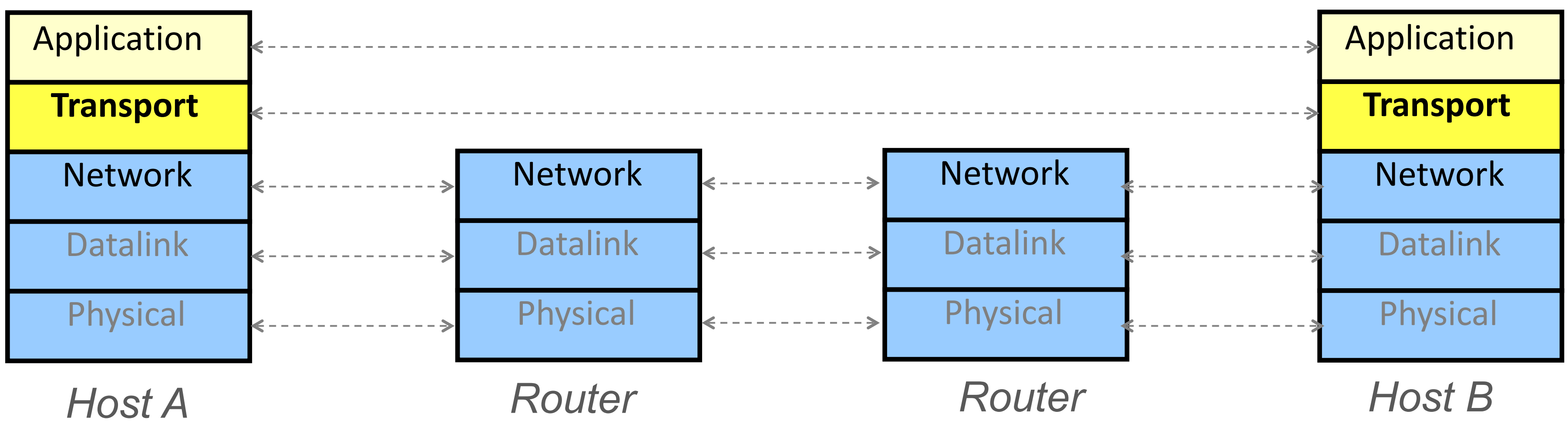
- Bridging the gap between
  - **the abstractions application designers want**
  - **the abstractions networks can easily support**

Semantics of correct delivery



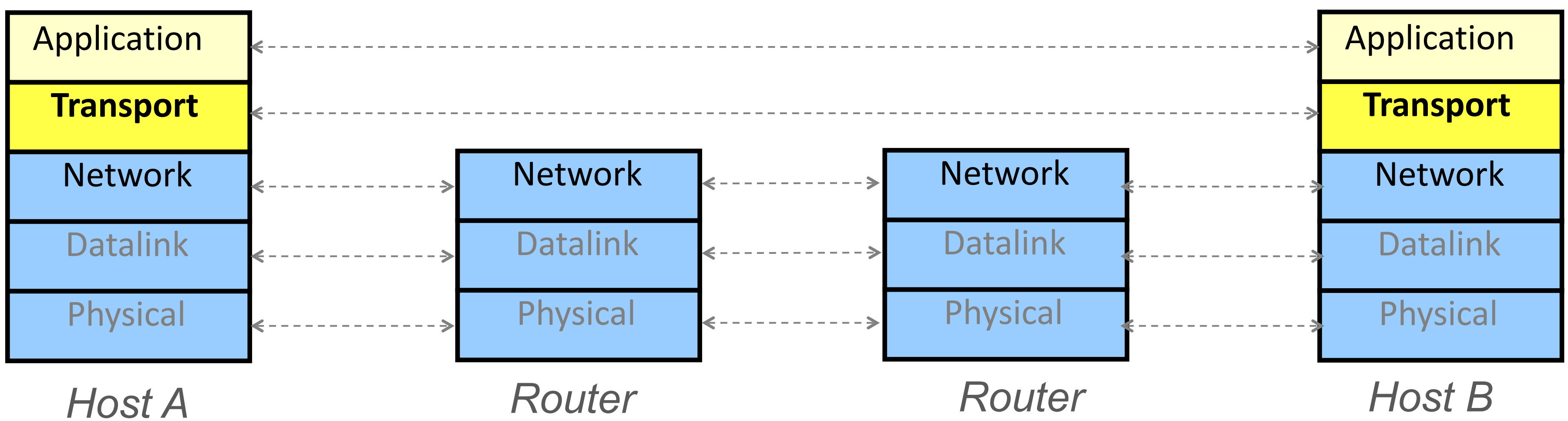
Semantics of correct delivery

- At network layer: *best-effort* delivery



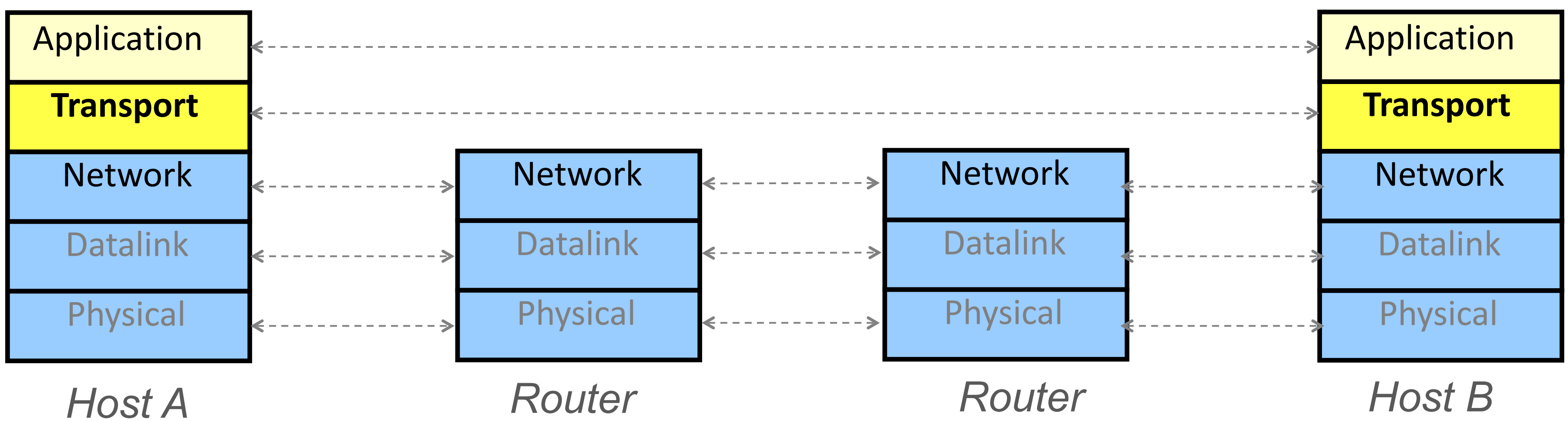
Semantics of correct delivery

- 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

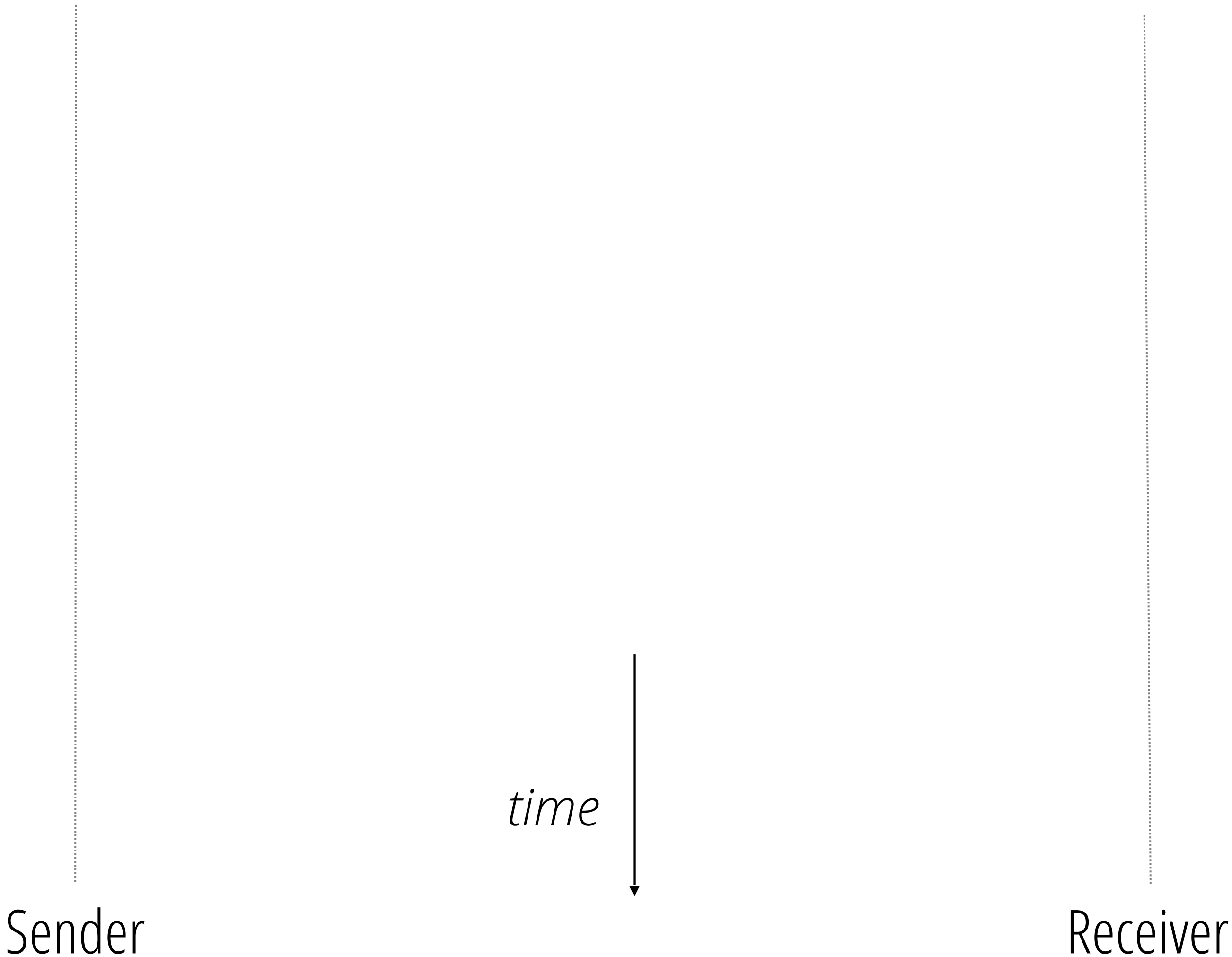
A best-effort network

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

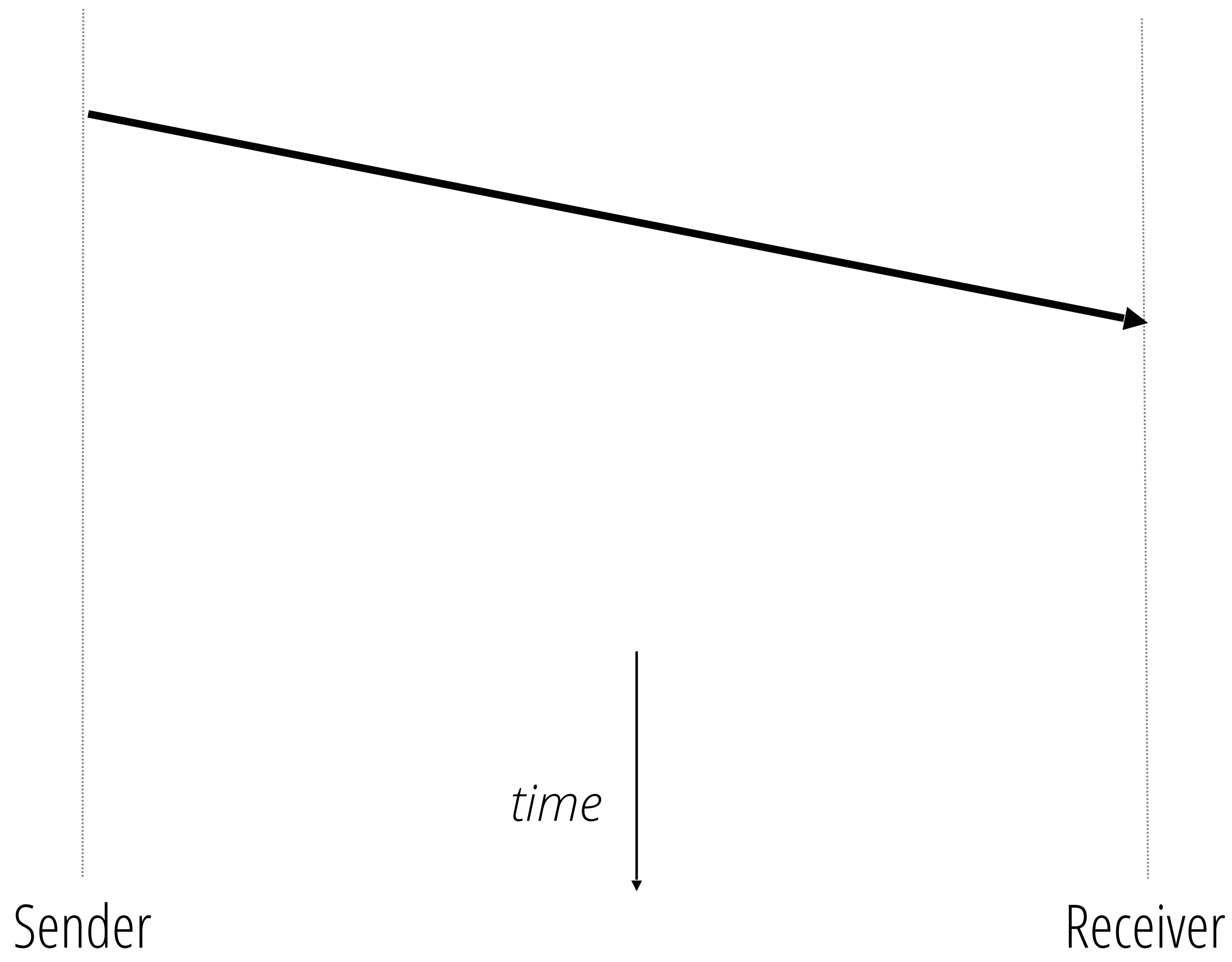
A best-effort network

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

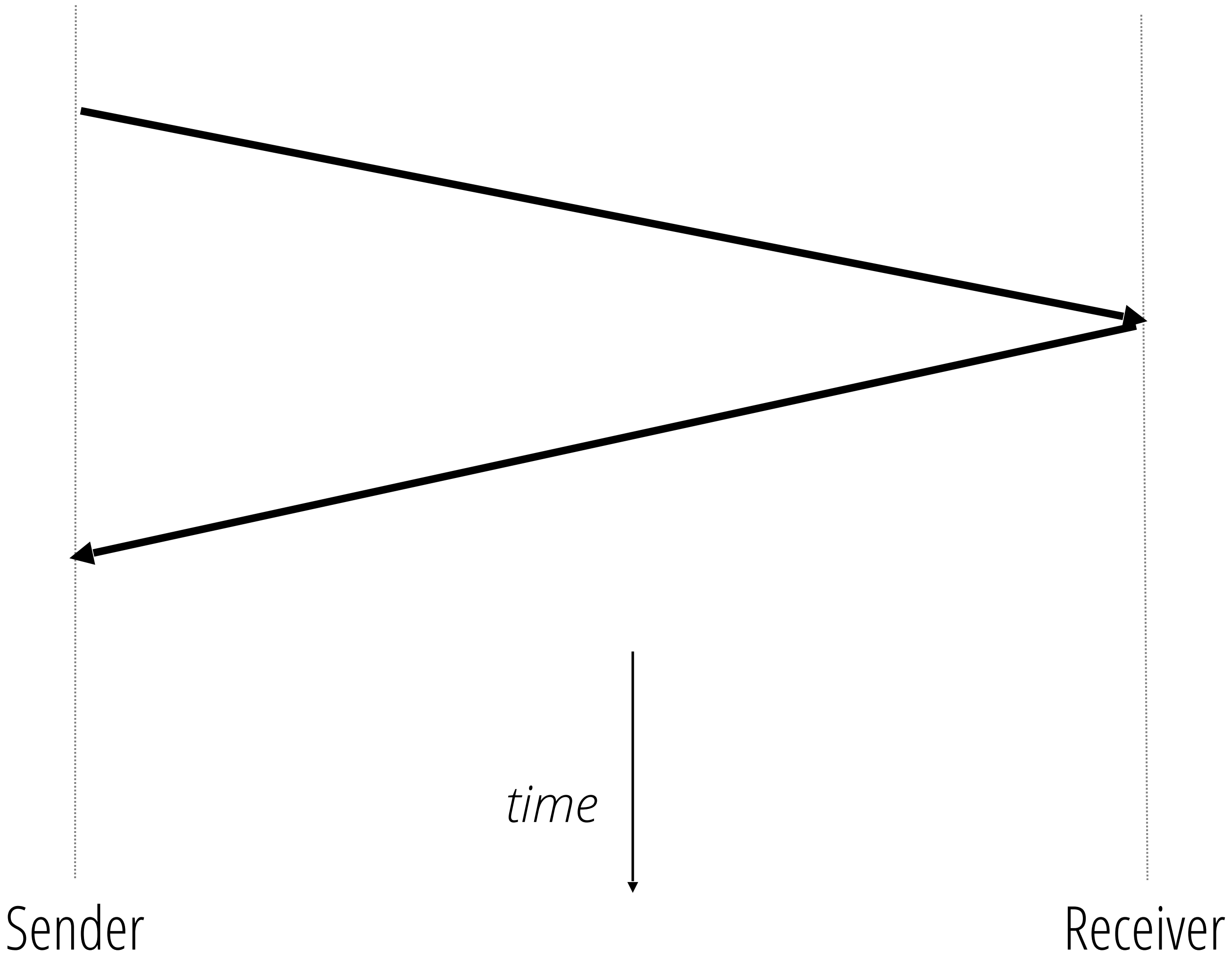
Quick reminder



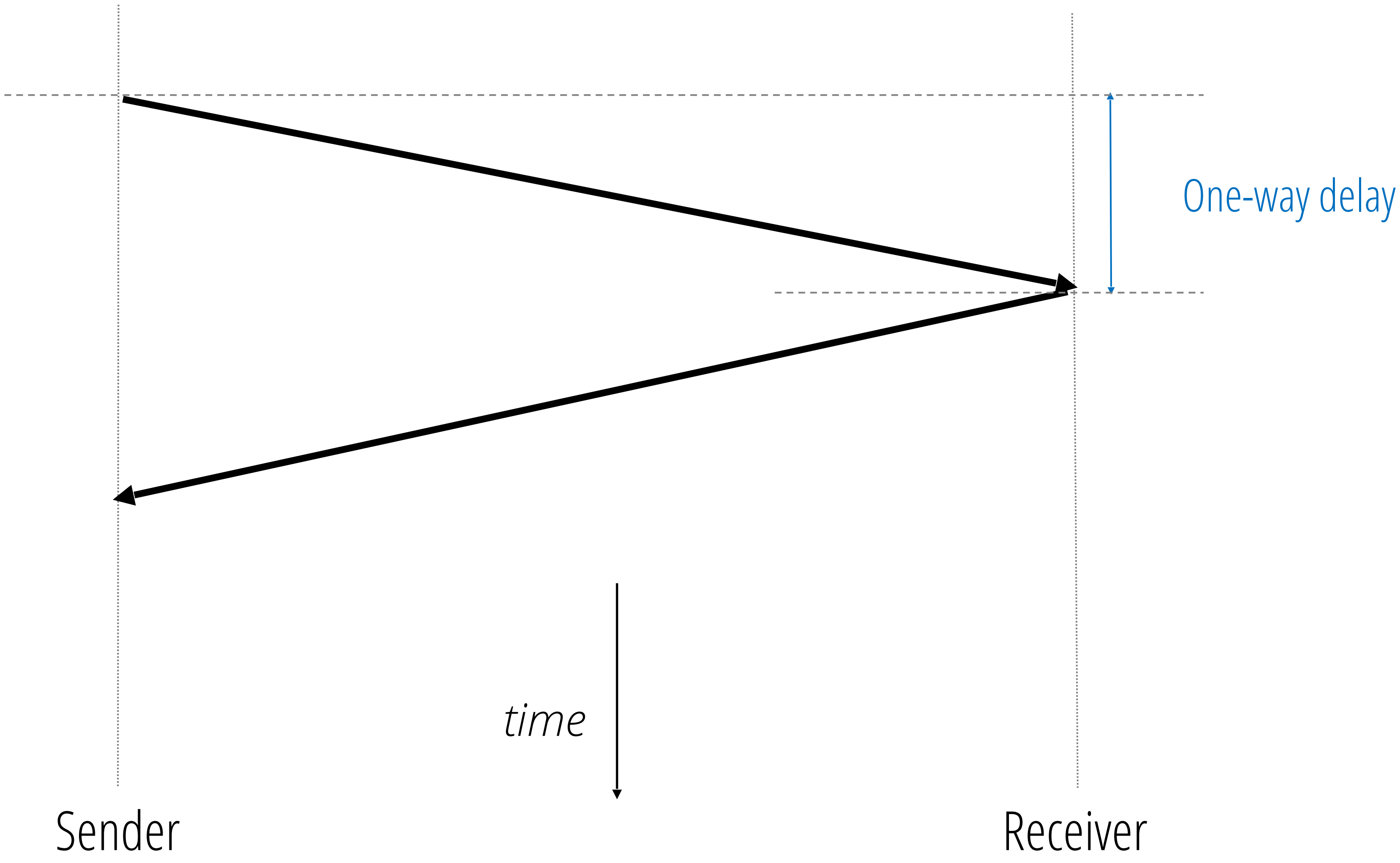
Quick reminder



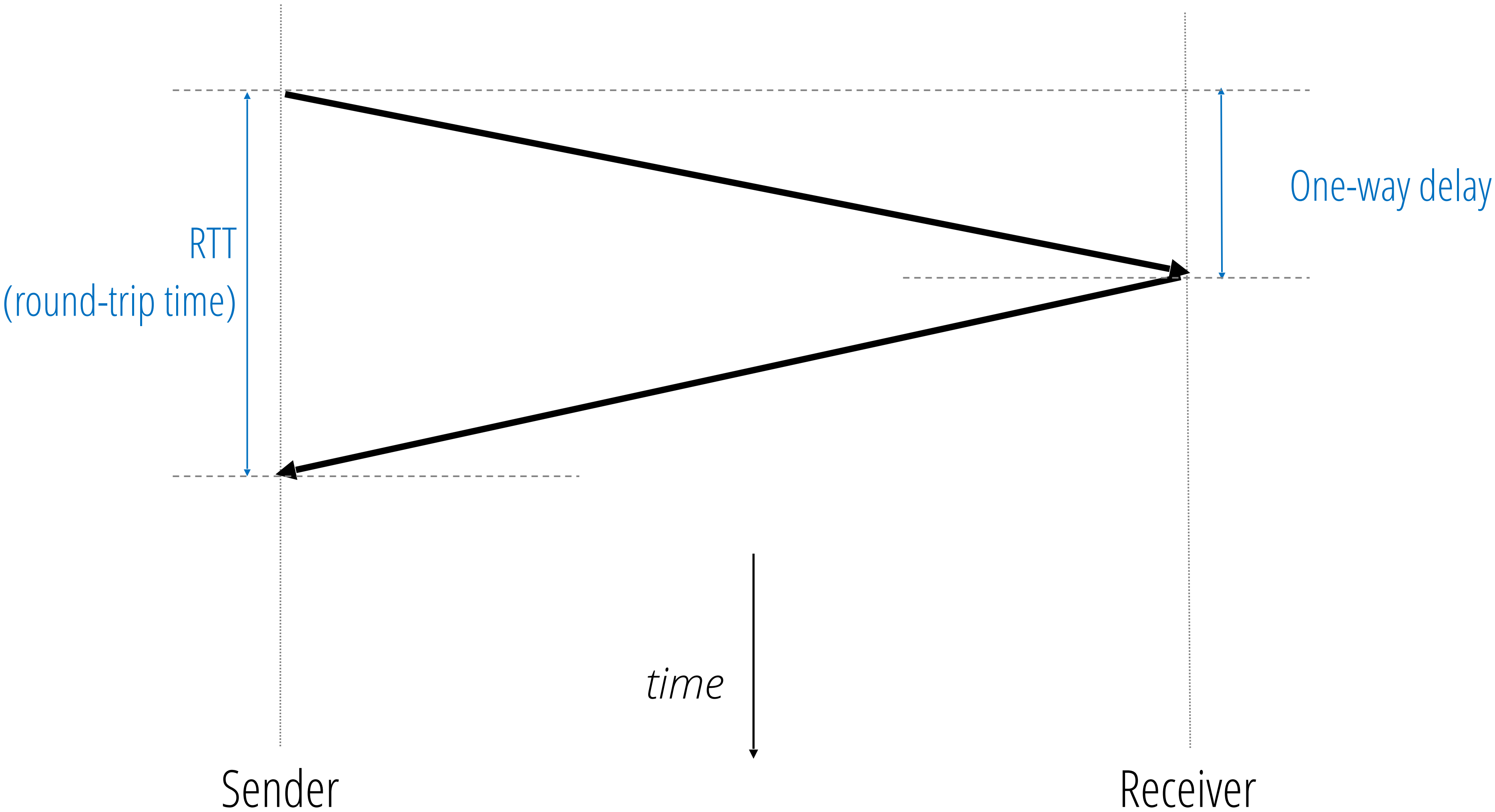
Quick reminder



Quick reminder



Quick reminder



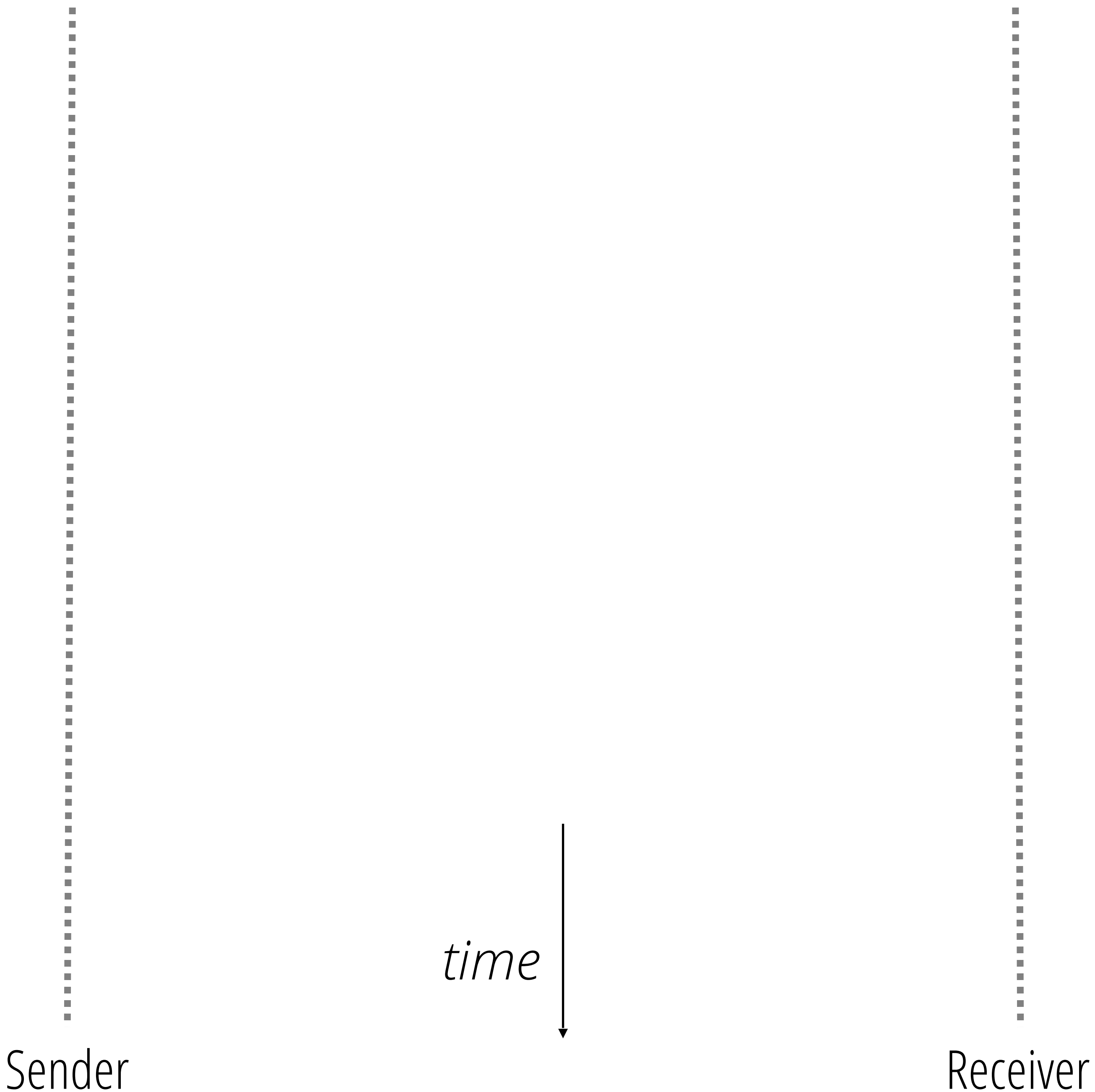
Designing a reliability protocol

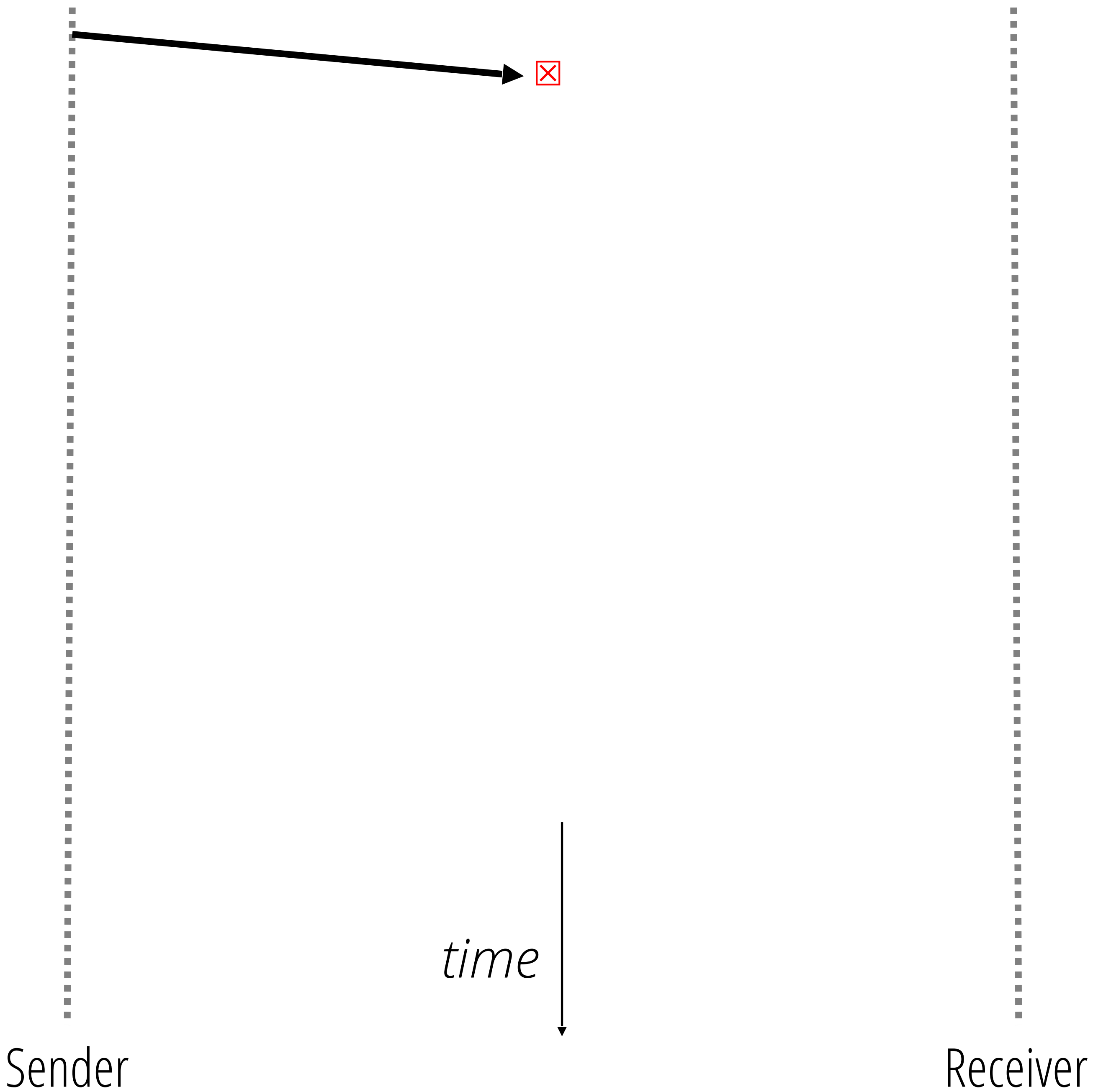
Designing a reliability protocol

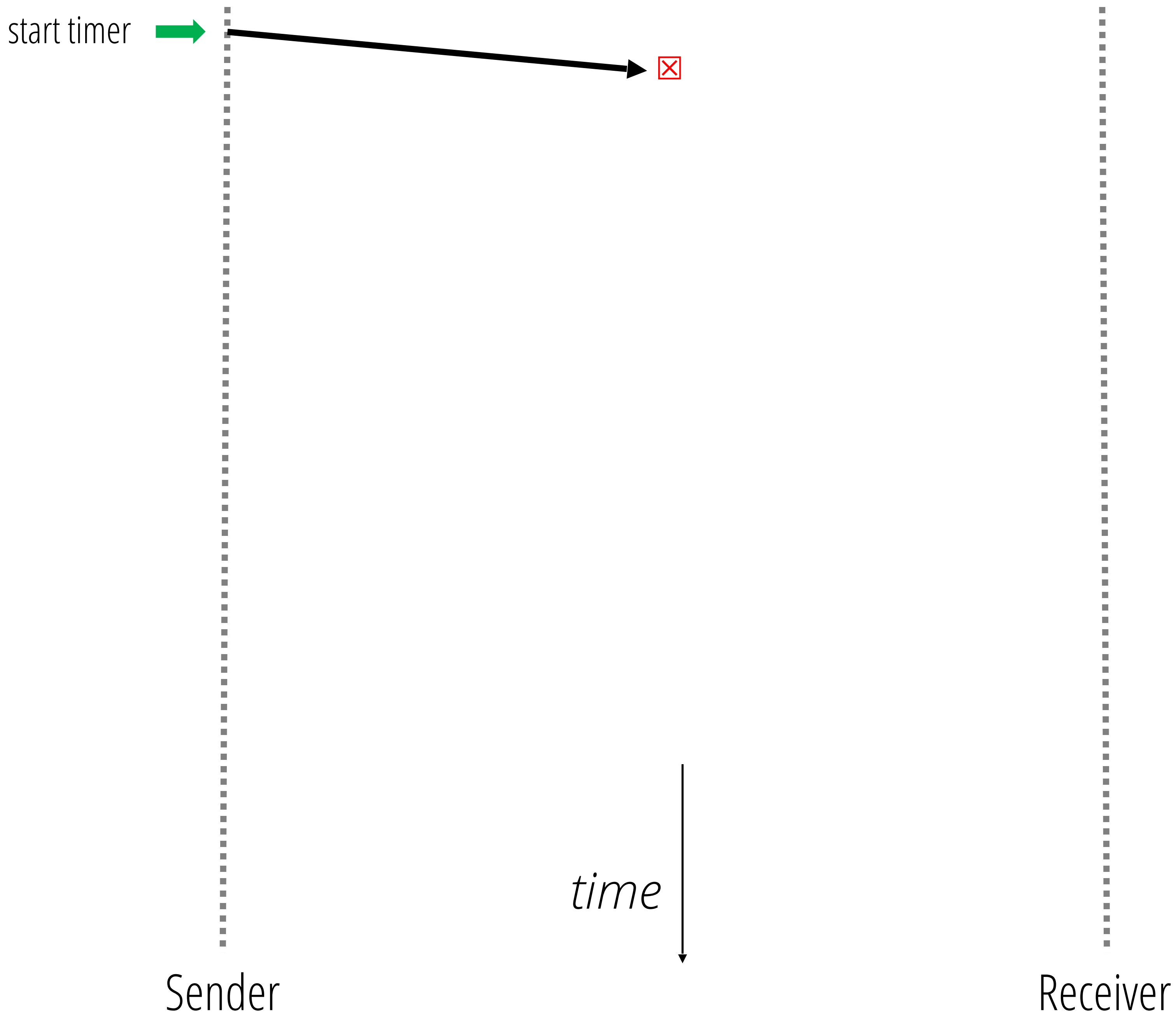
- Let’s start with the single packet case

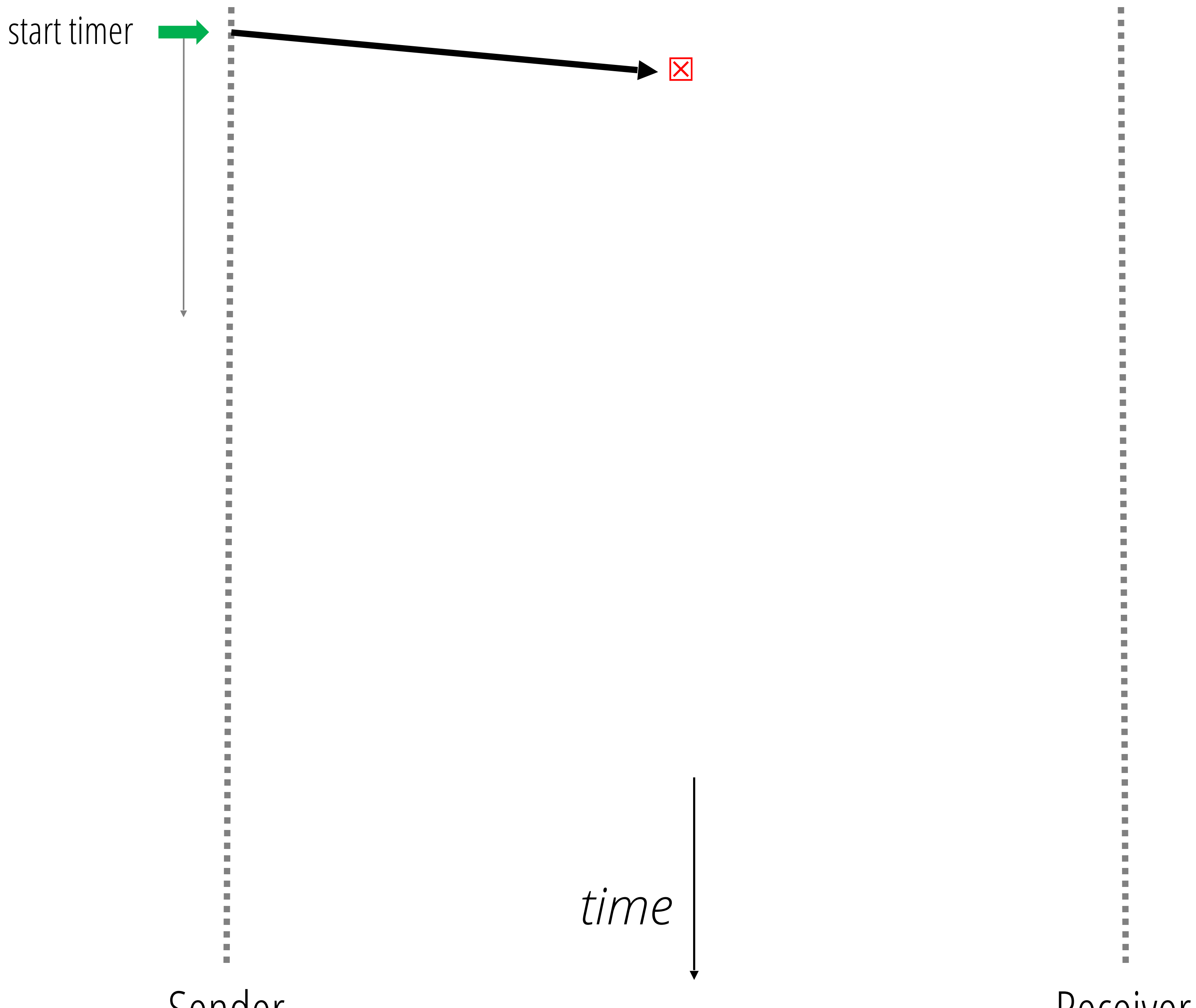
Designing a reliability protocol

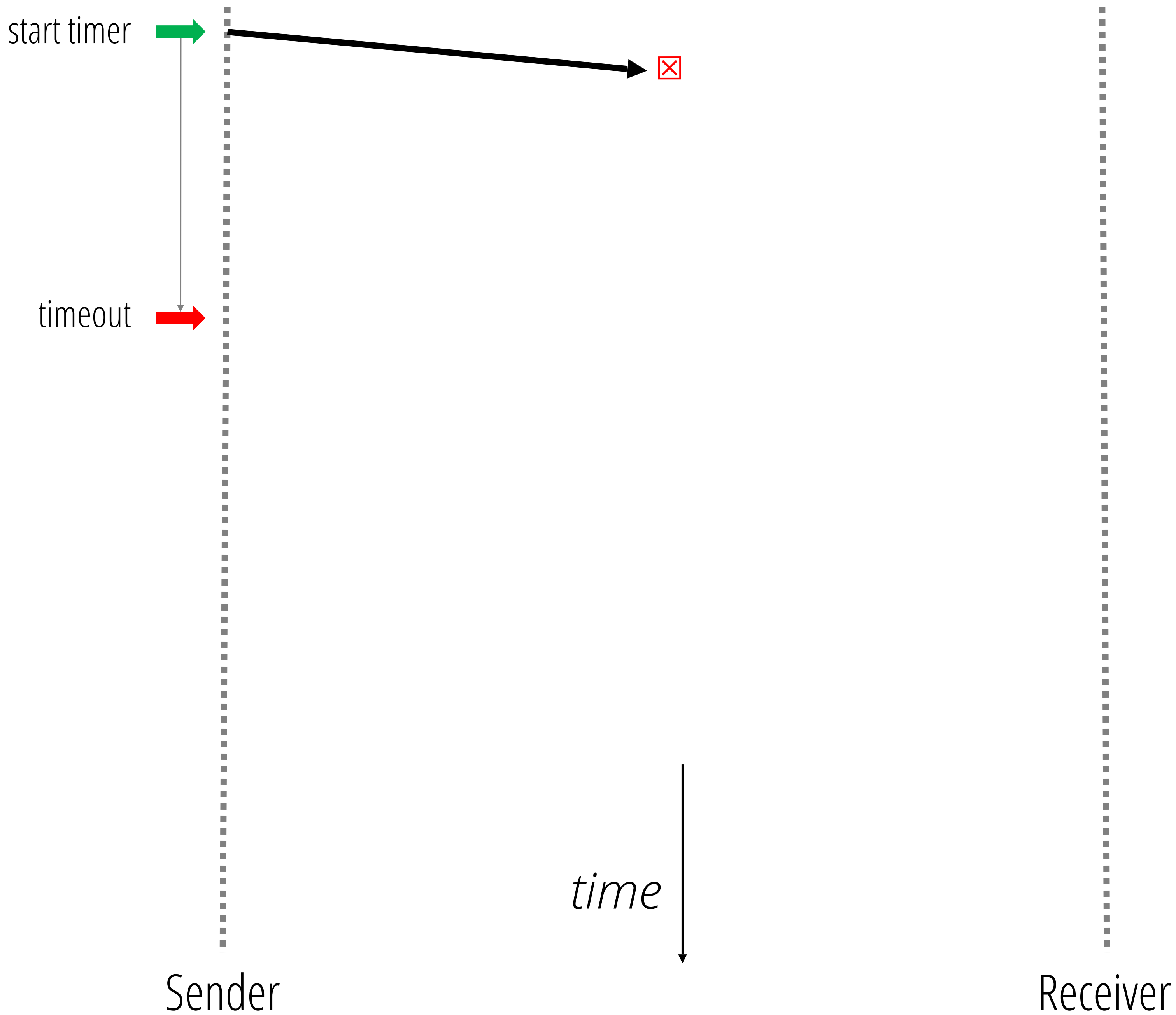
- Let’s start with the single packet case
- Remember
  - Packets can be dropped
  - Packets can be corrupted
  - Packets can be reordered
  - Packets can be delayed
  - Packets can be duplicated
  - ….

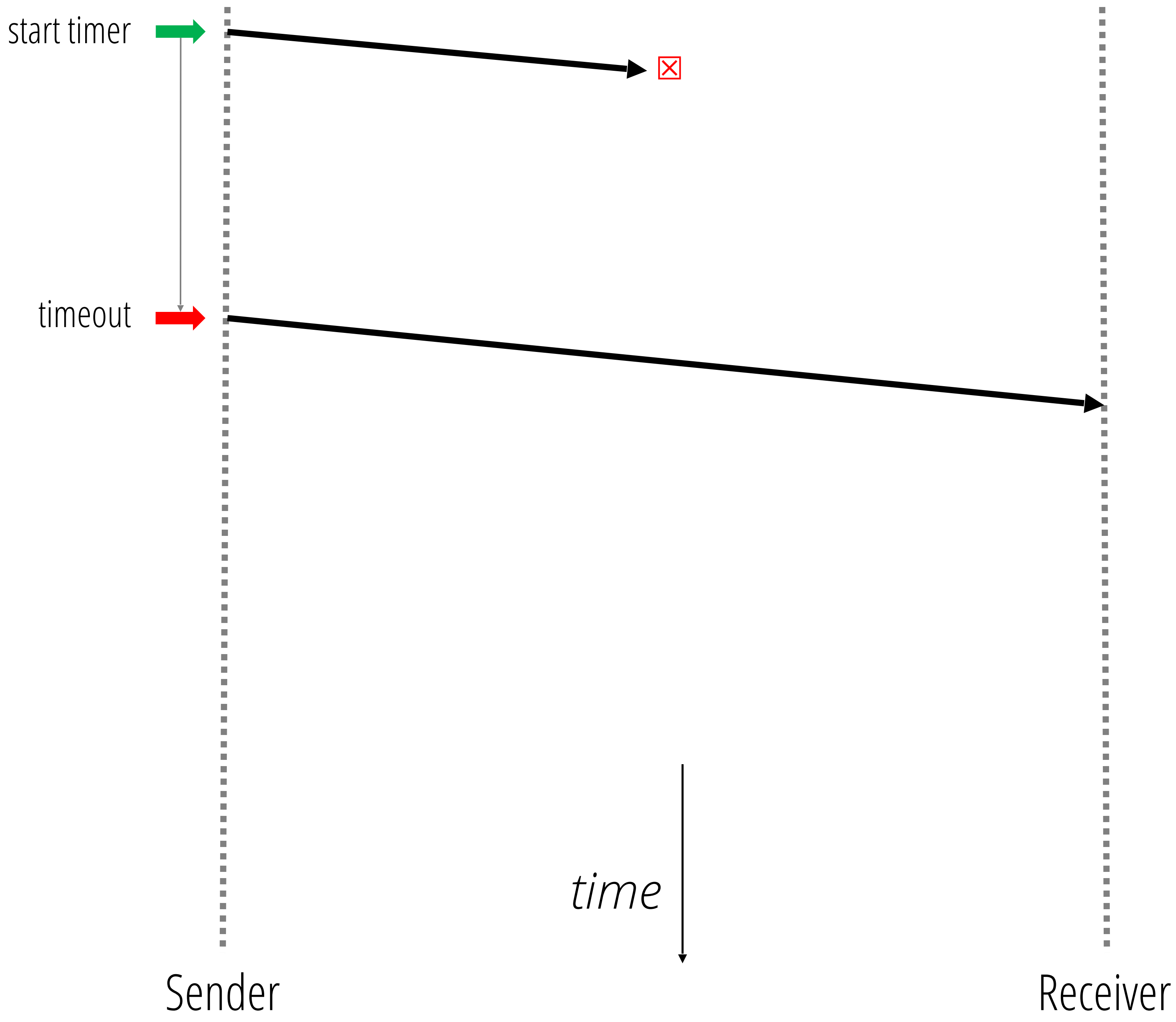

time
Sender
Receiver

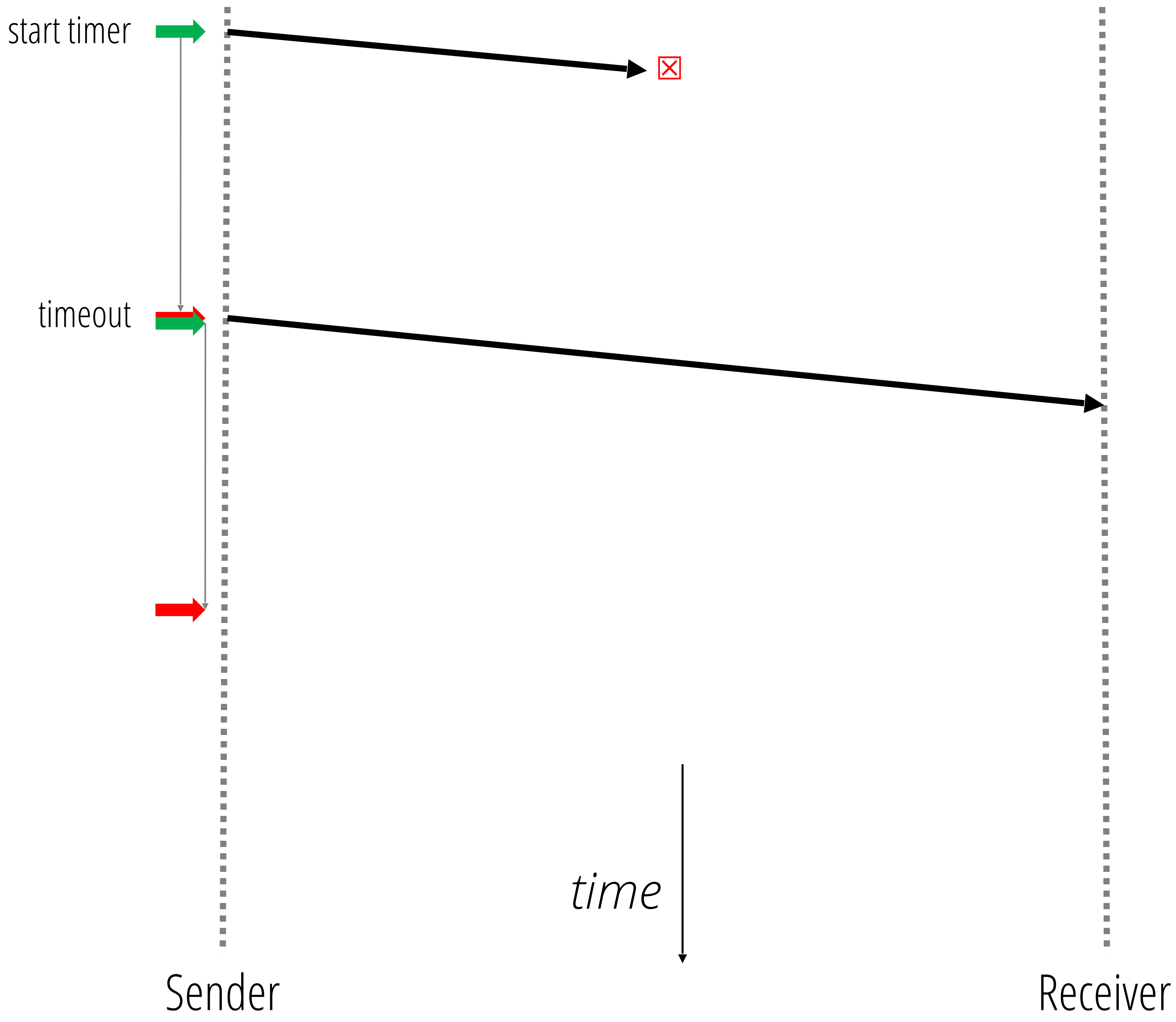

time
Sender
Receiver


start timer
time
Sender
Receiver

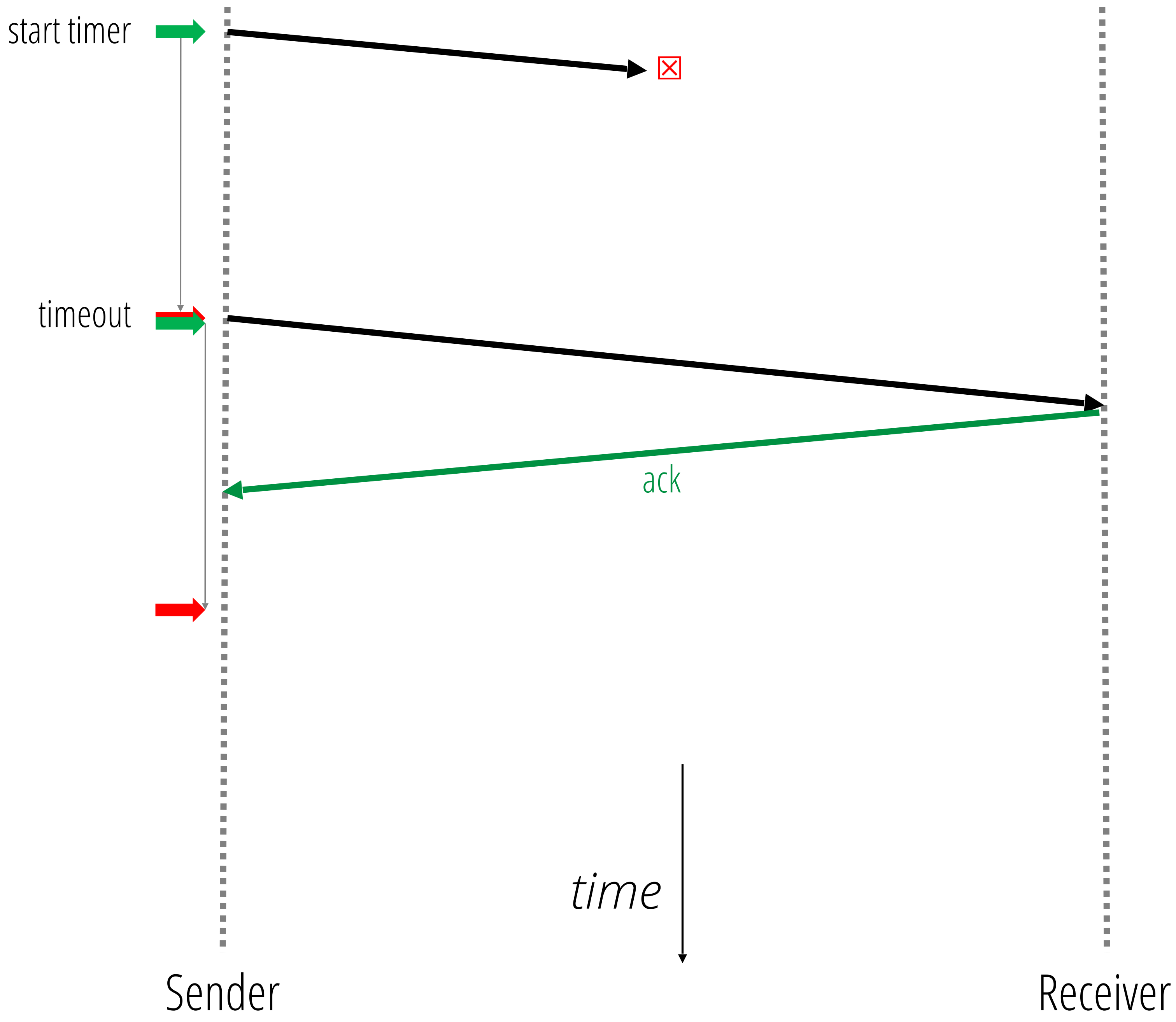

start timer
time
Sender
Receiver

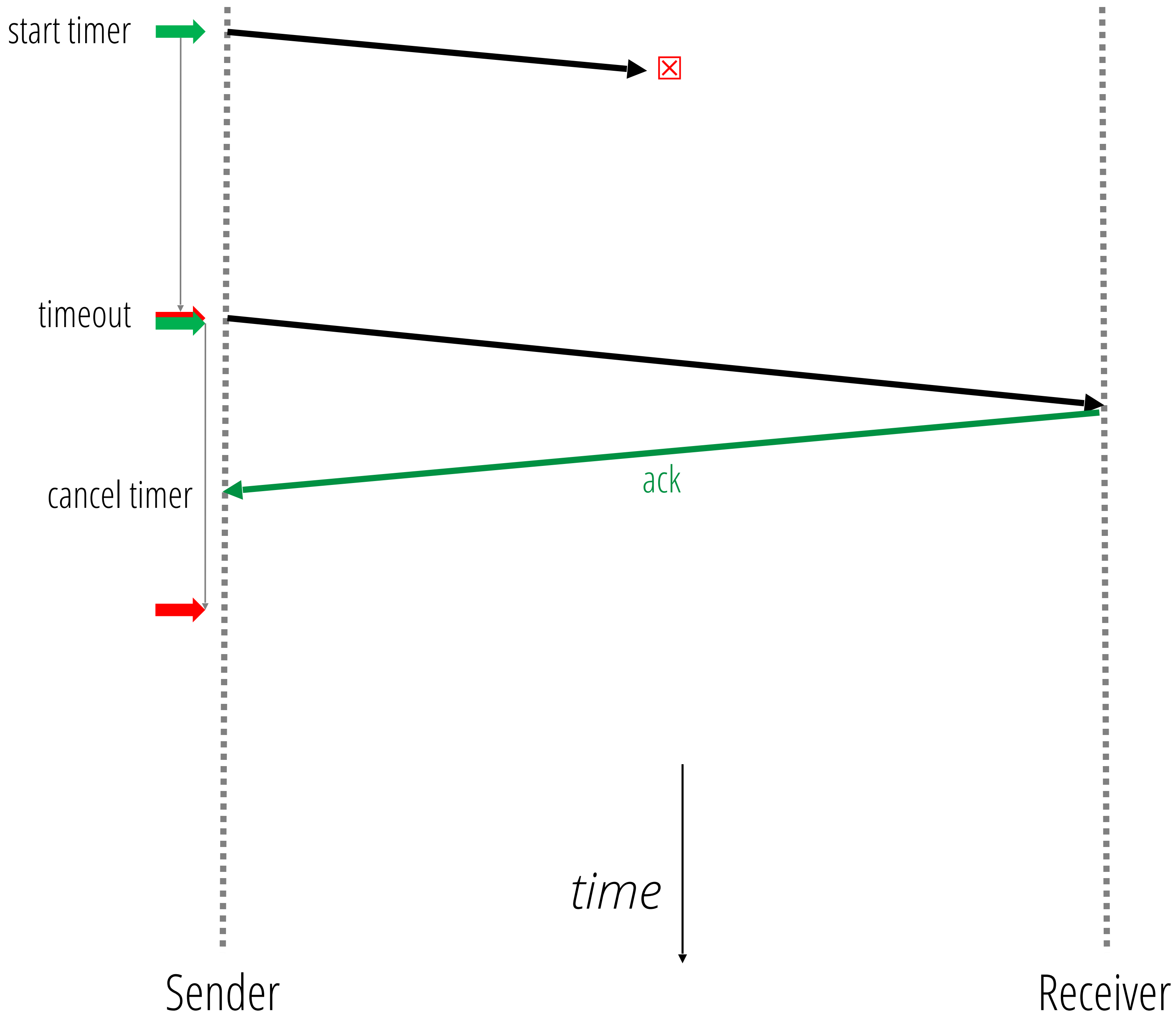

start timer
timeout
time
Sender
Receiver

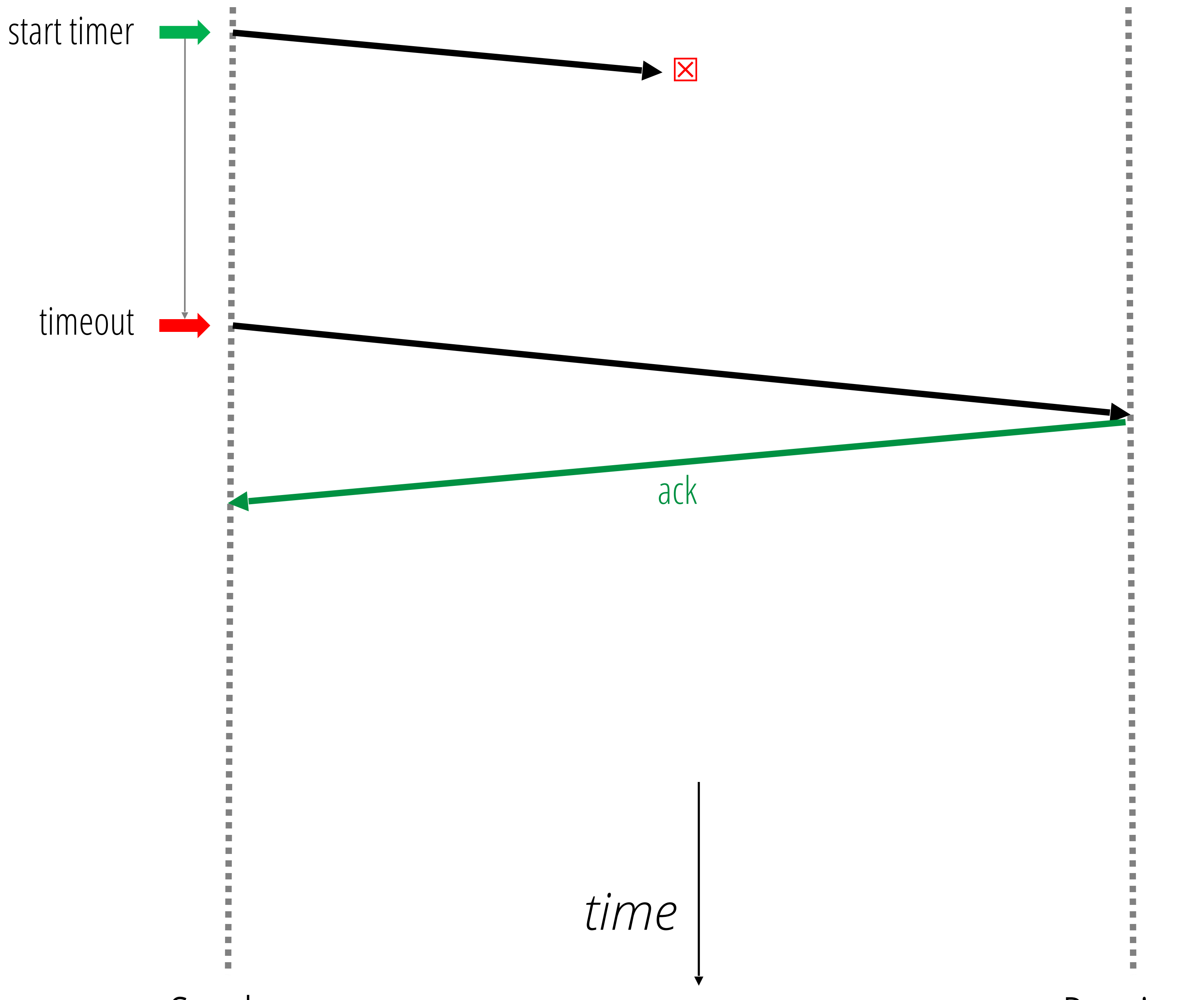

start timer
timeout
time
Sender
Receiver

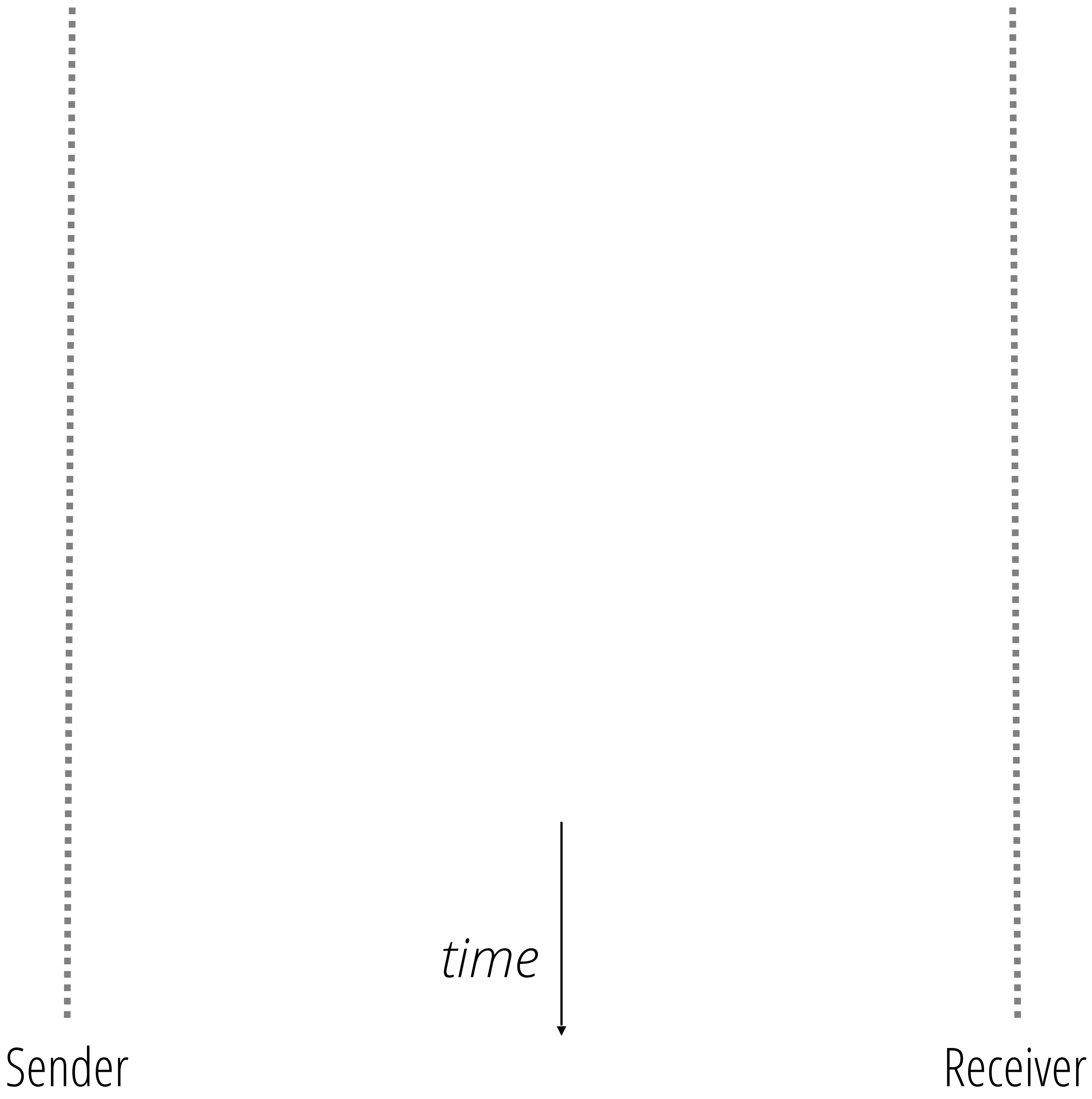


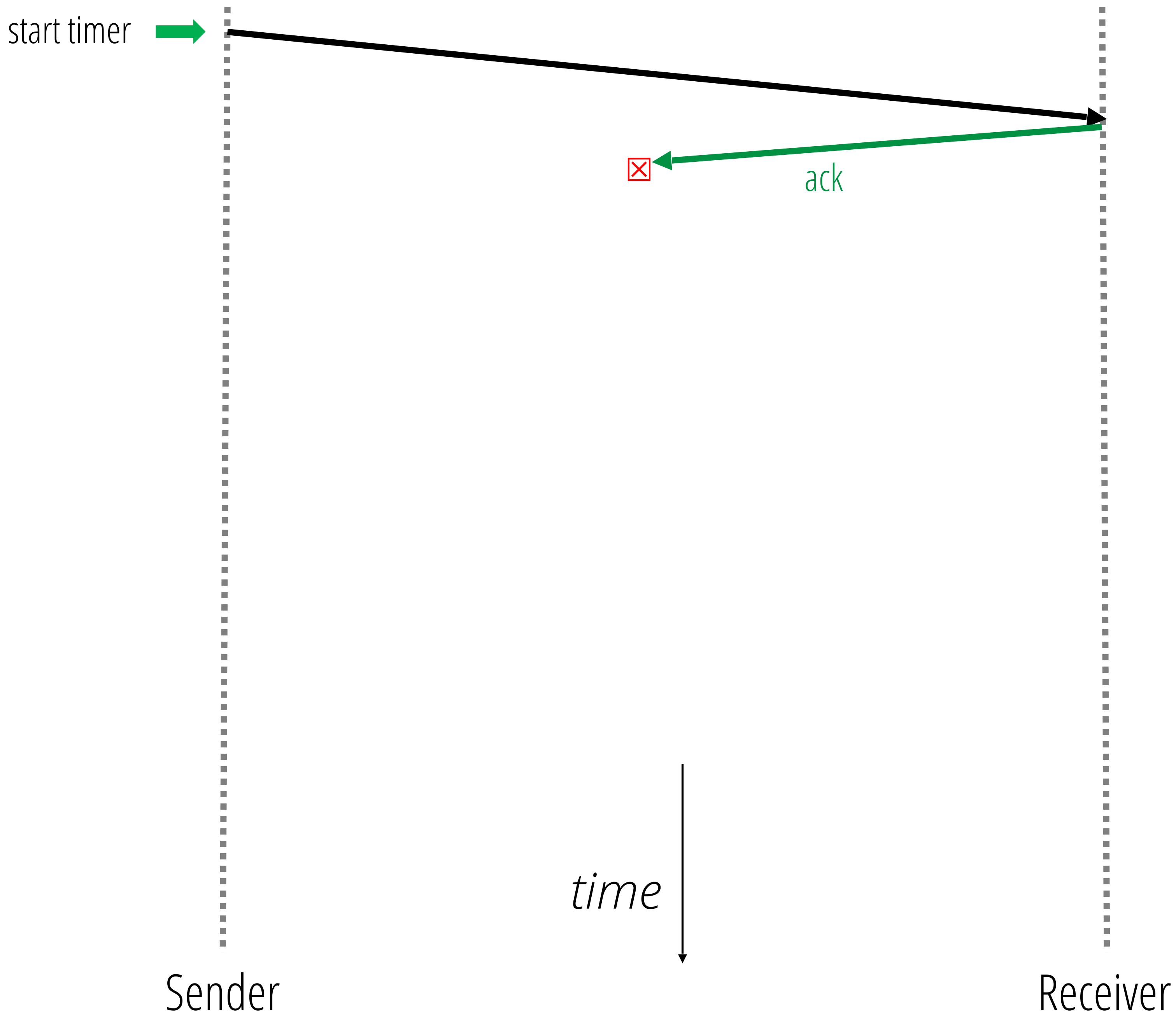
start timer
timeout
time
Sender
Receiver


start timer
timeout
ack
time
Sender
Receiver

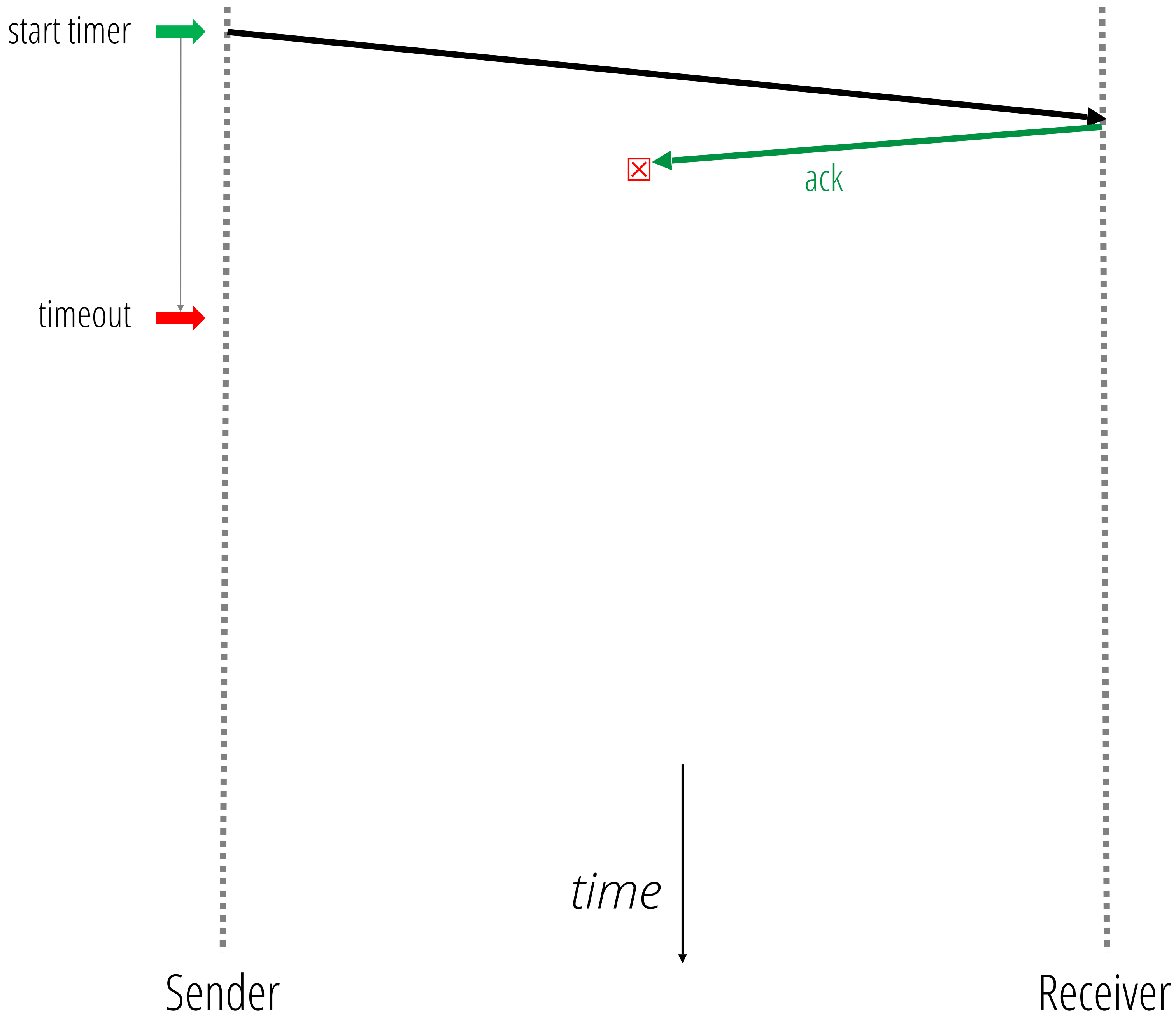

start timer
timeout
cancel timer
ack
time
Sender
Receiver

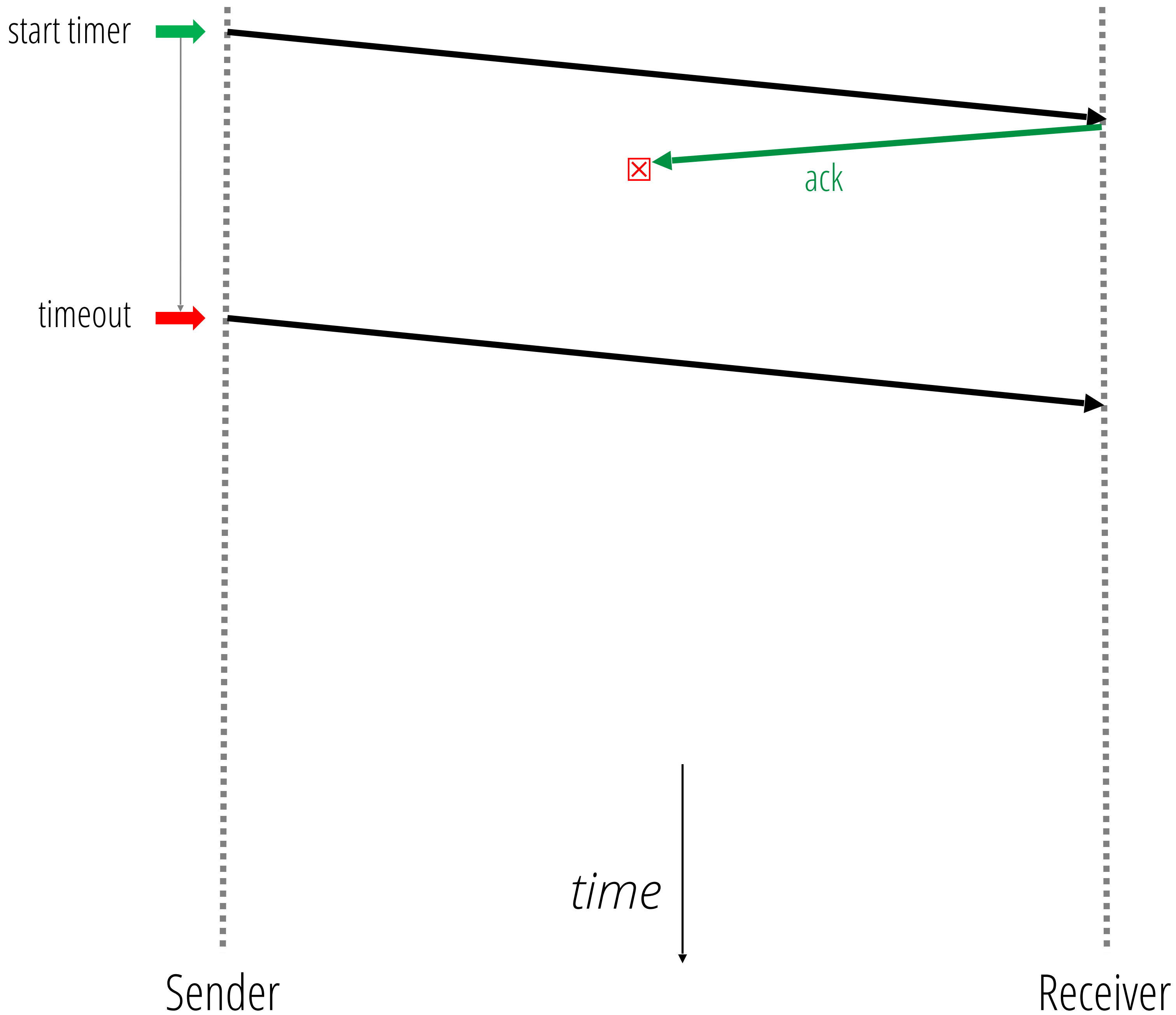

start timer
timeout
ack
time
Sender
Receiver

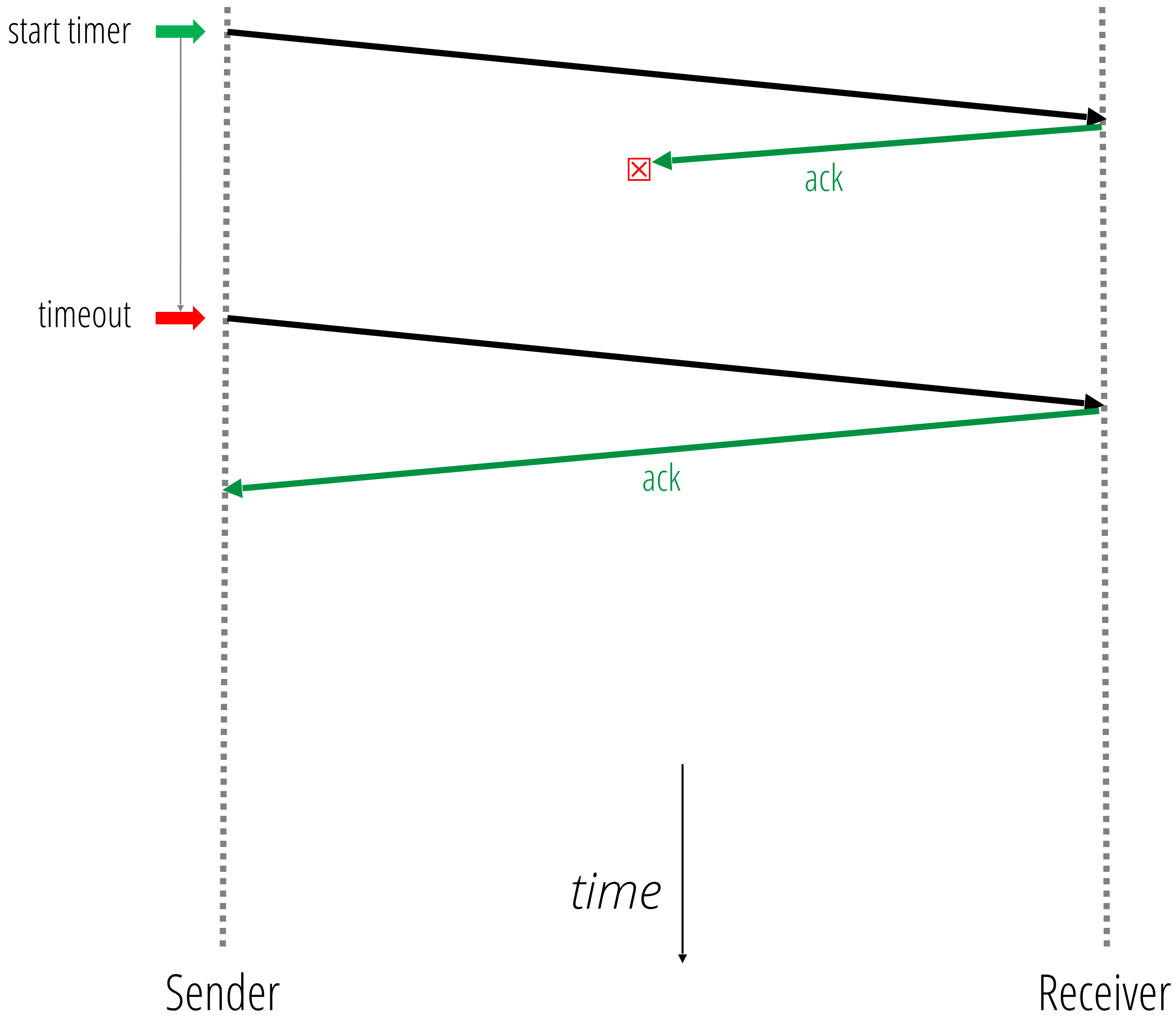

time
Sender
Receiver

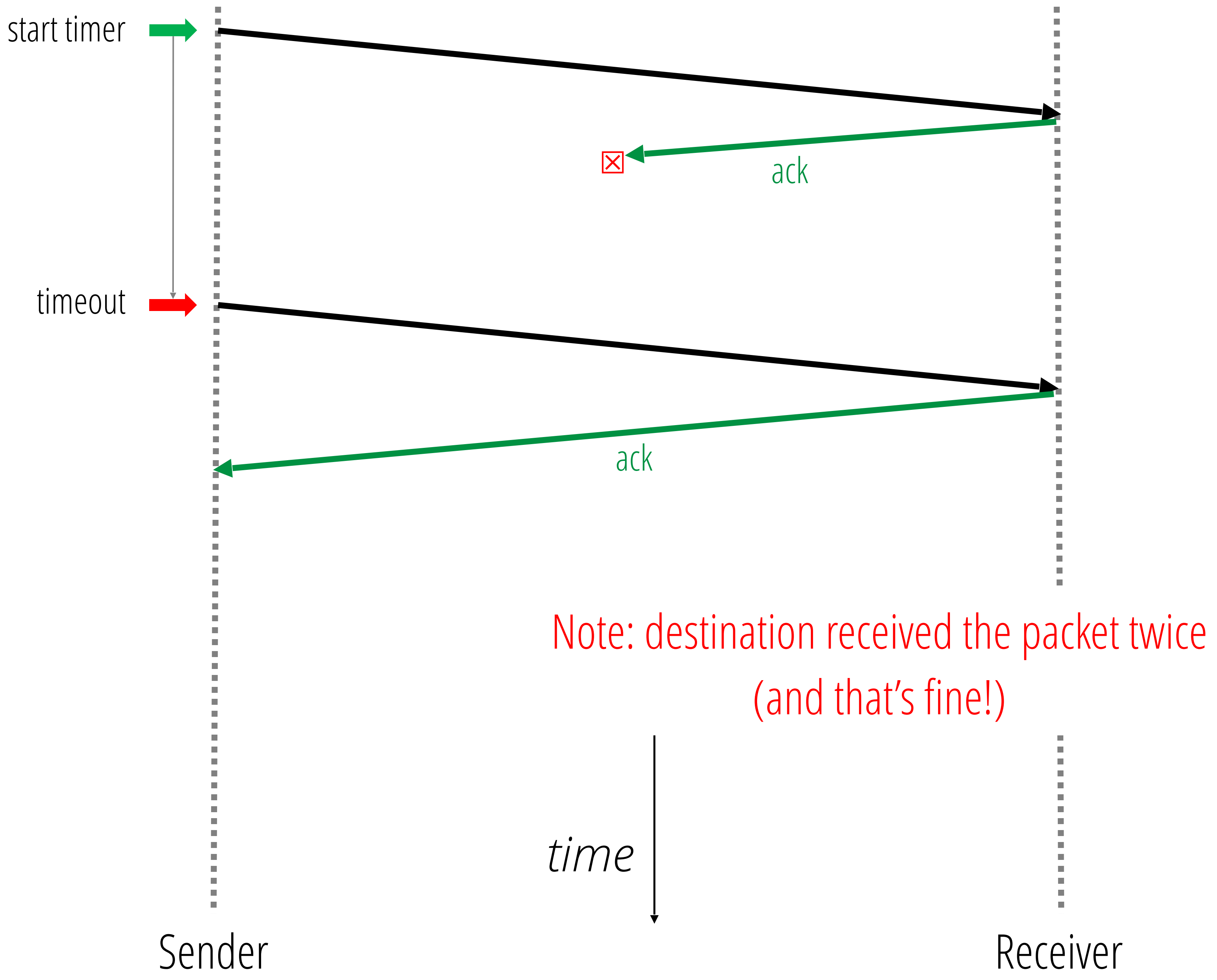


start timer
ack
time
Sender
Receiver


start timer
ack
timeout
time
Sender
Receiver


start timer
ack
timeout
time
Sender
Receiver


start timer
ack
timeout
ack
time
Sender
Receiver


start timer
ack
timeout
ack
Note: destination received the packet twice
(and that's fine!)
time
Sender
Receiver

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?

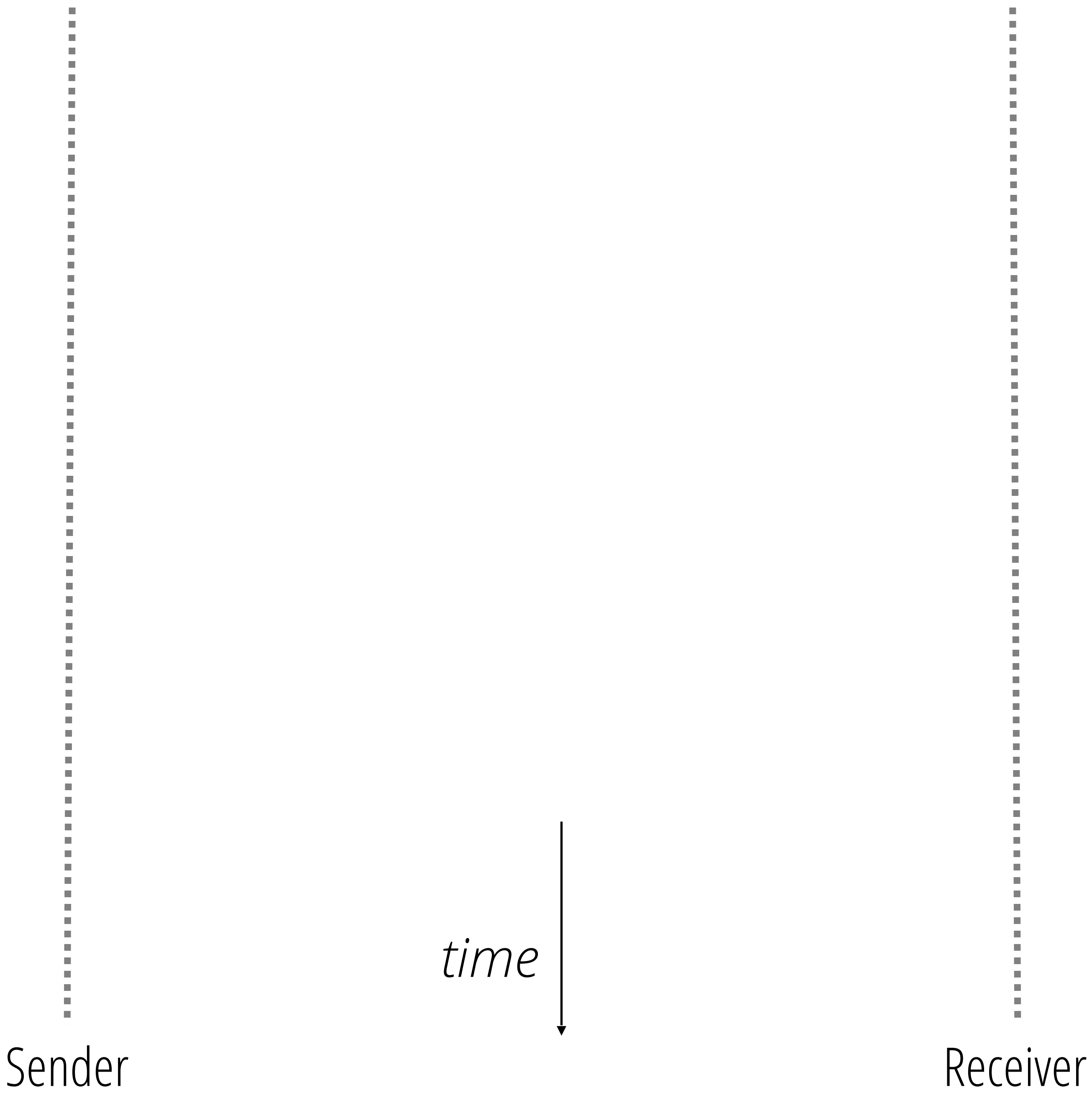
- Too long: will delay delivery
- Too short: unnecessary retransmissions

- Ideally, proportional to the RTT (next lecture)

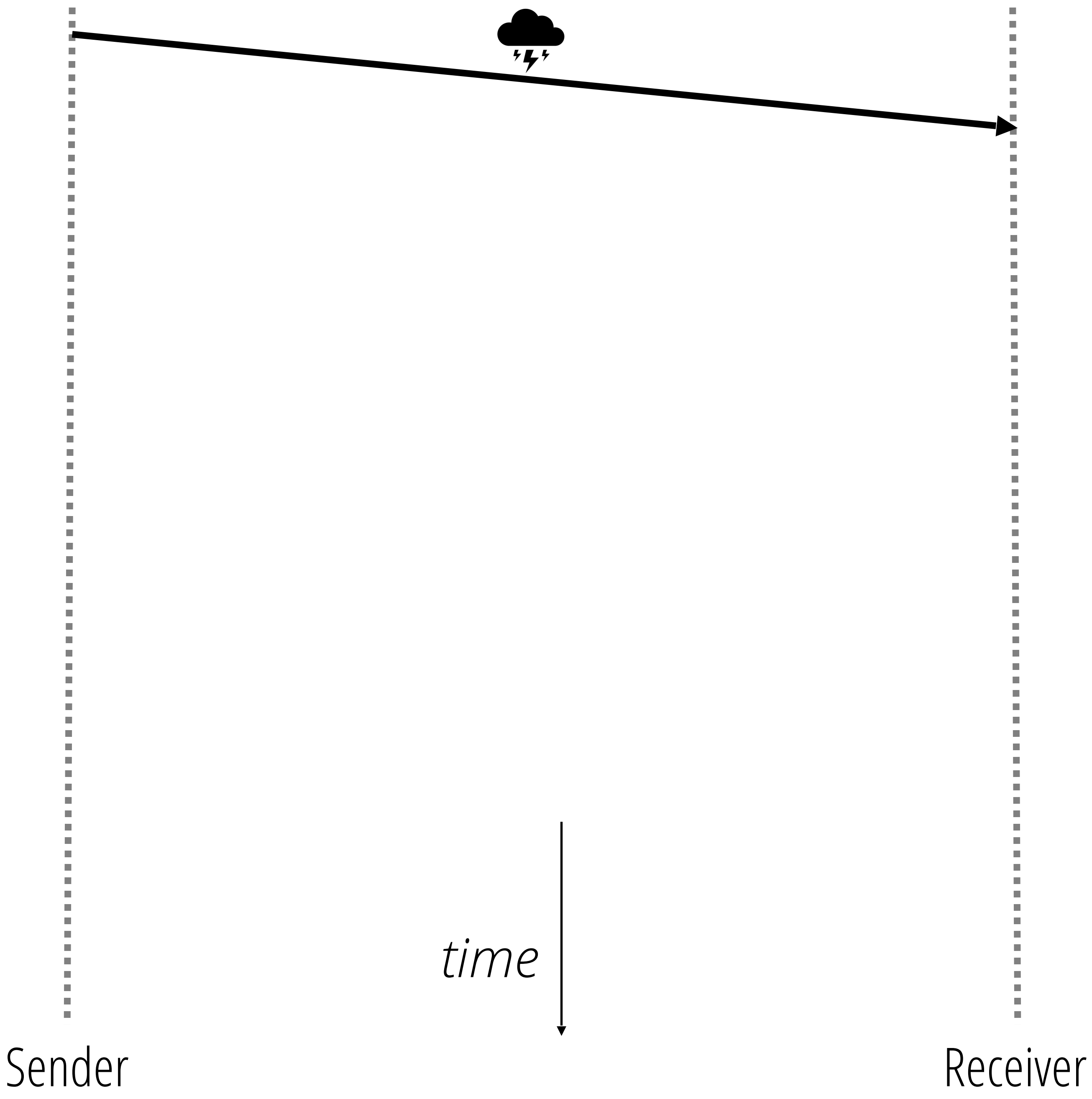
- Non-trivial to get right in practice
  - RTTs vary across paths (10μs to 100s ms)
  - RTT of a fixed path varies over time (load, congestion)

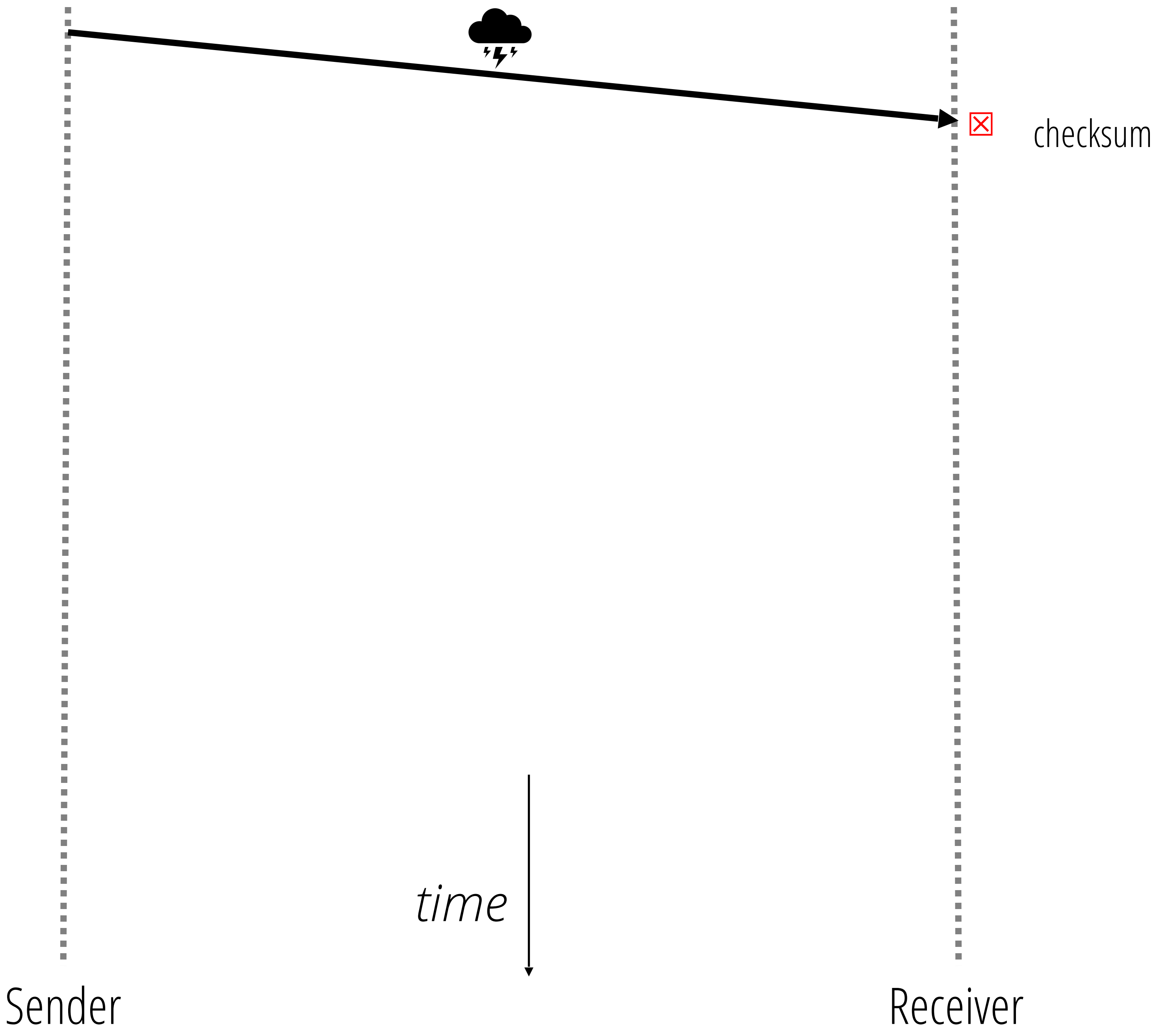
- Hence, often used as last resort

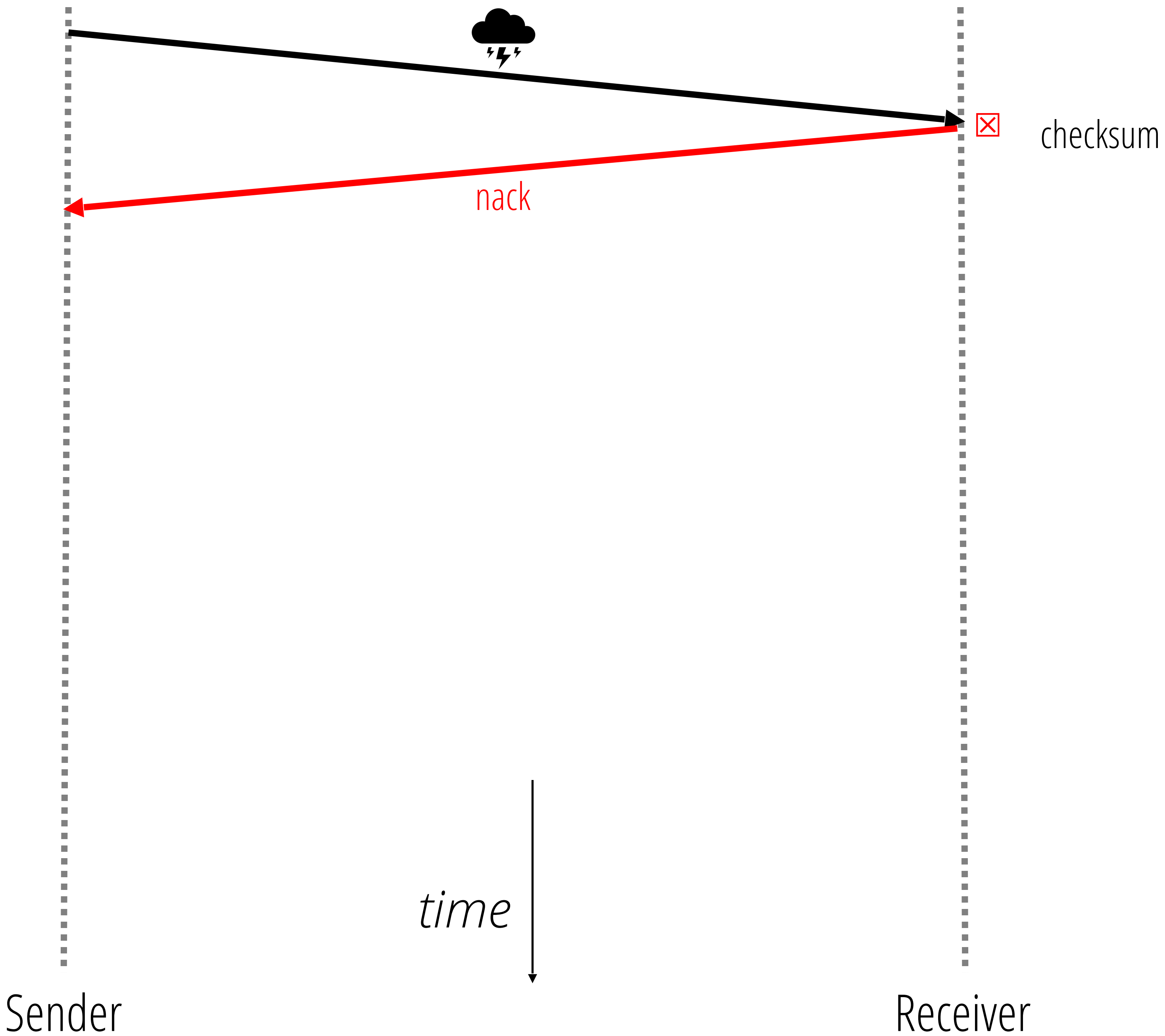
- We said
  - Packets can be lost (data or ACKs) 👍
  - Packets can be corrupted
  - Packets can be delayed
  - Packets can be duplicated
  - Packets can be reordered

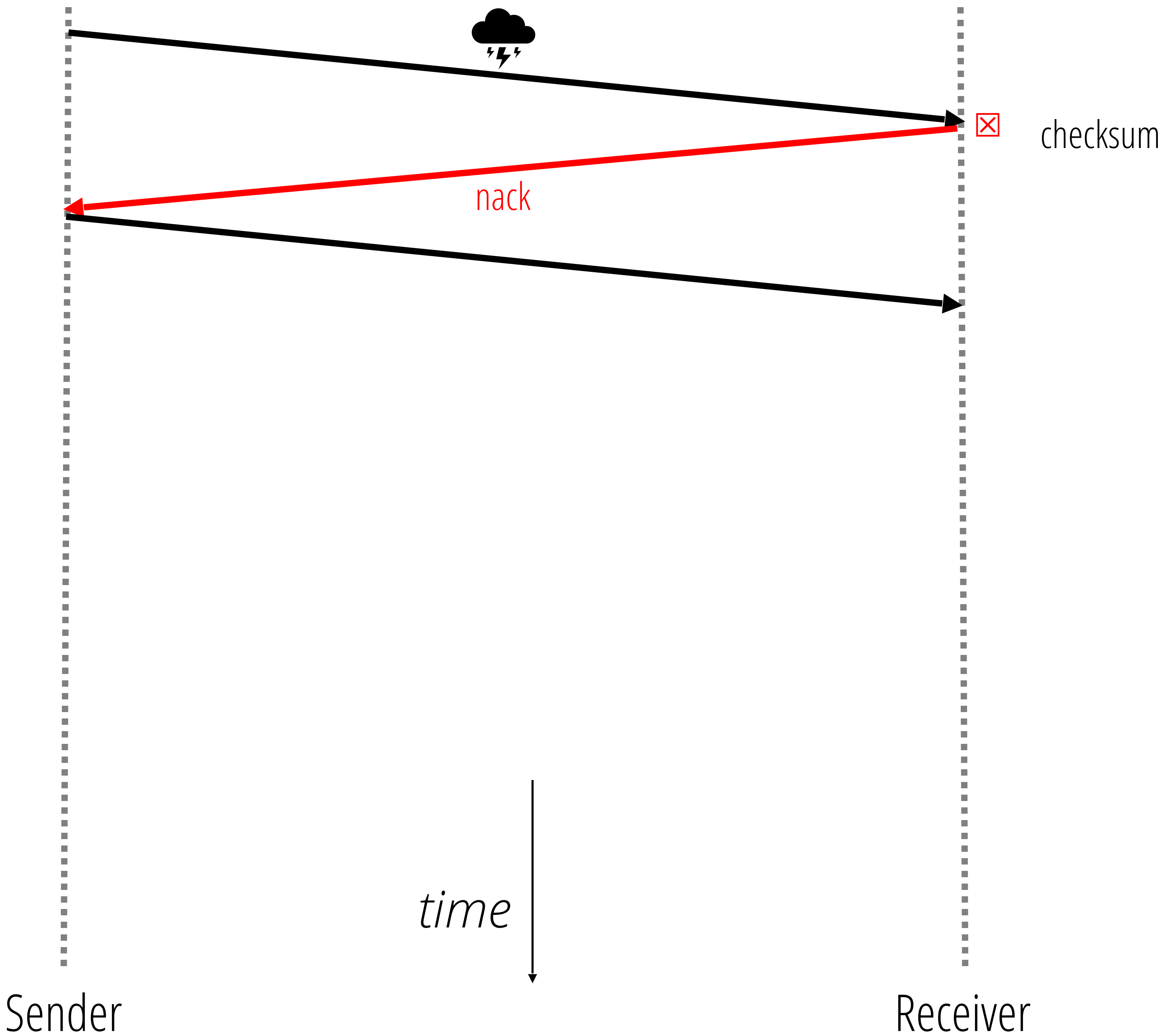


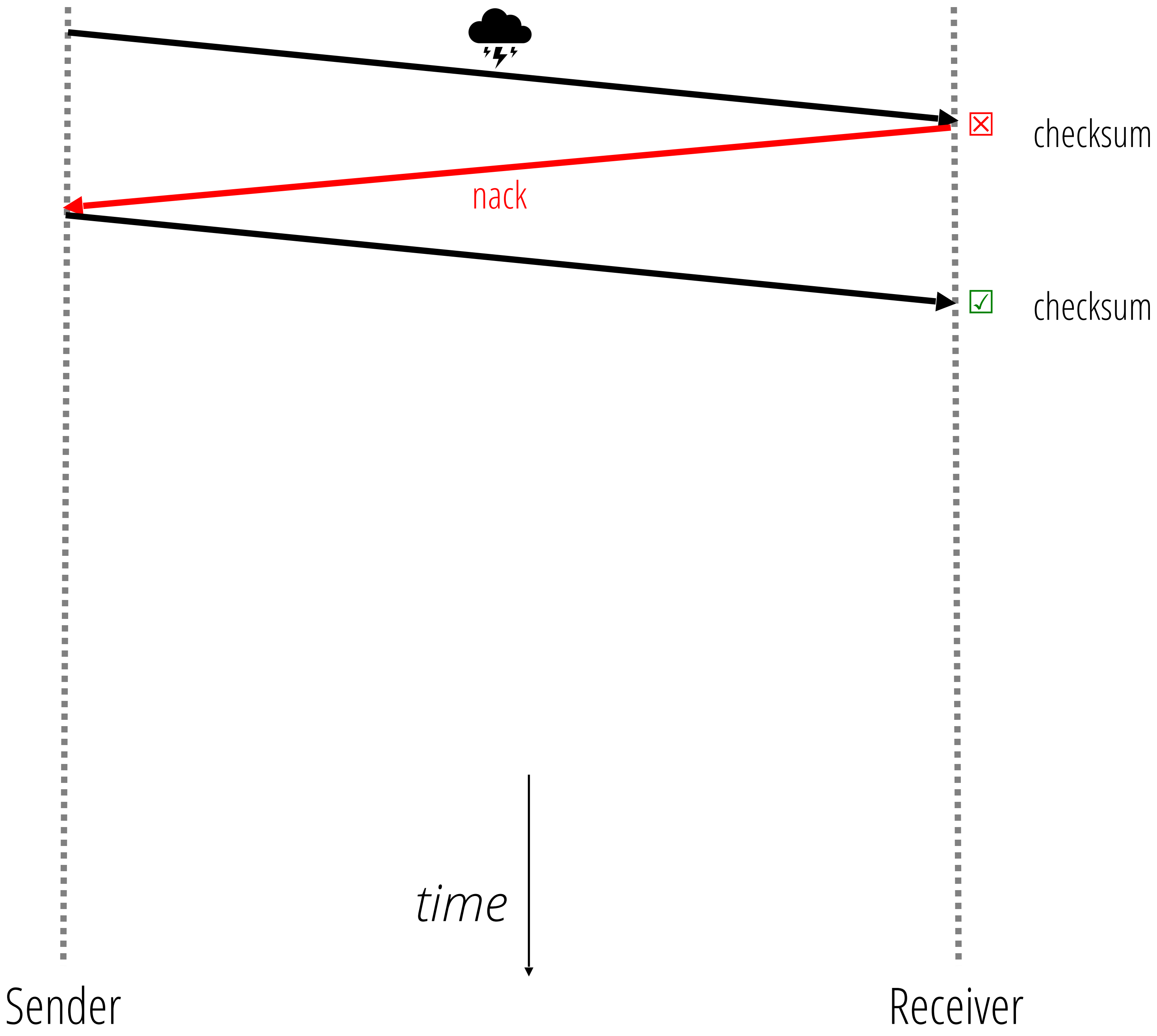
time
Sender
Receiver

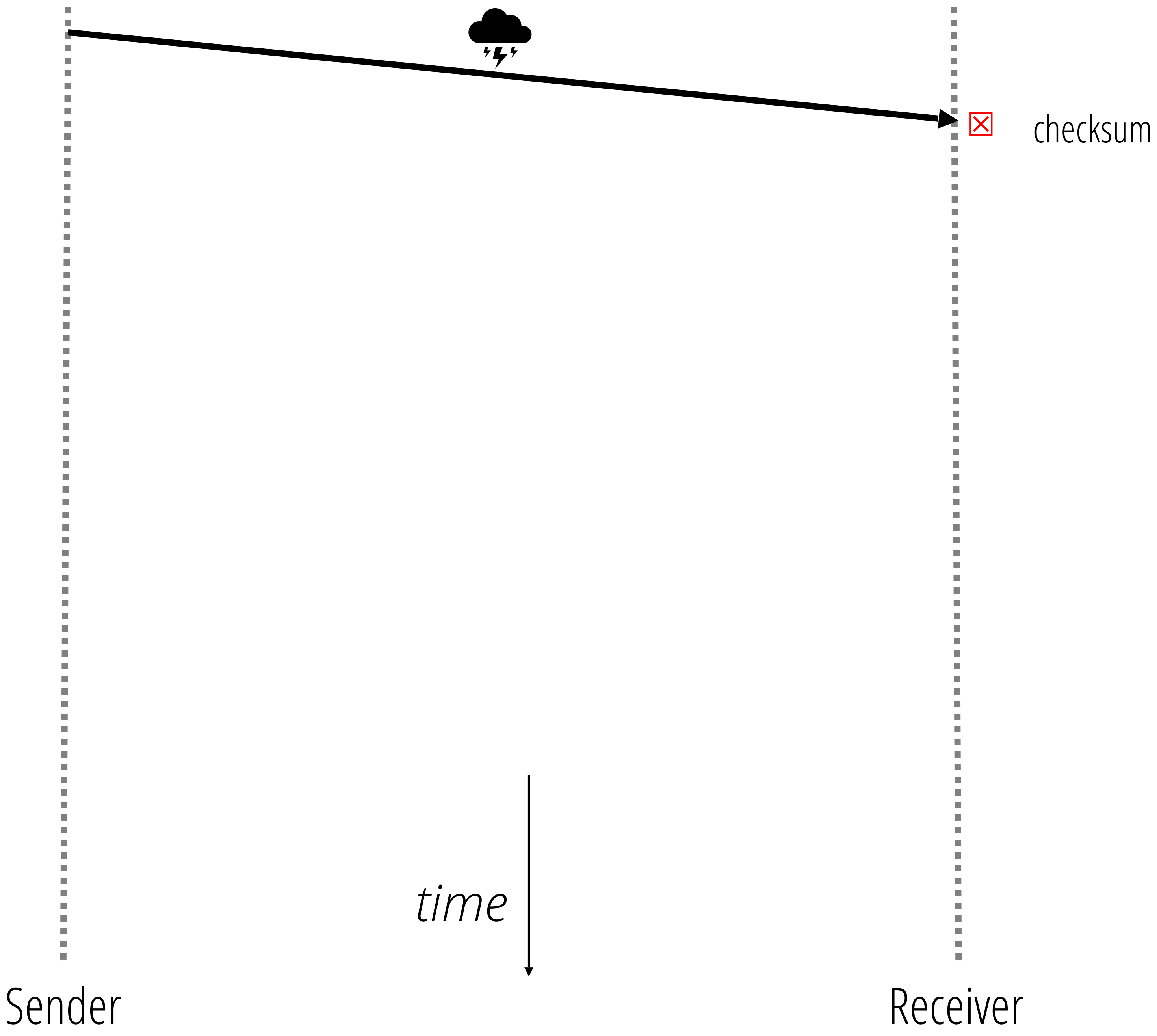

time
Sender
Receiver

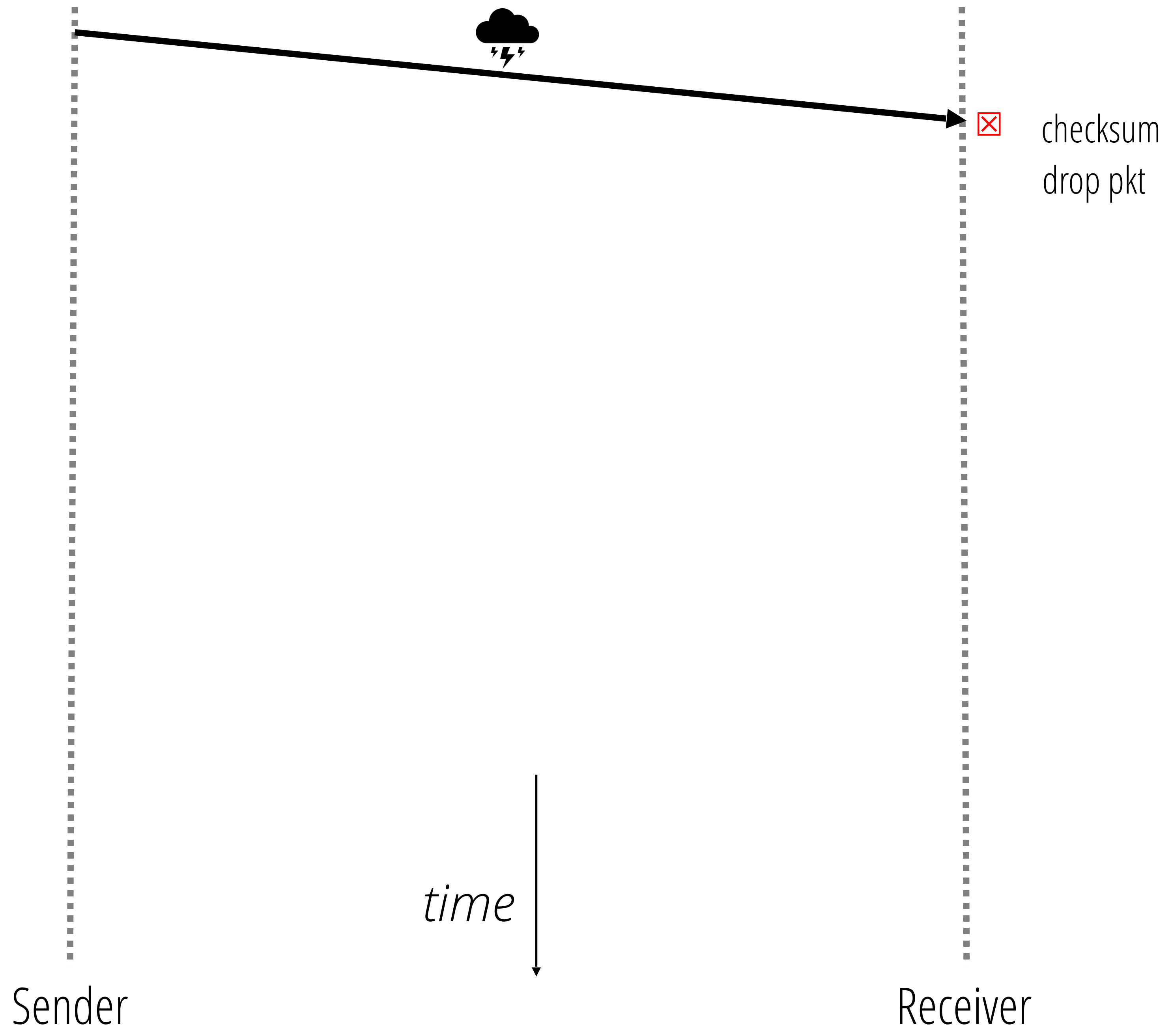

checksum
time
Sender
Receiver

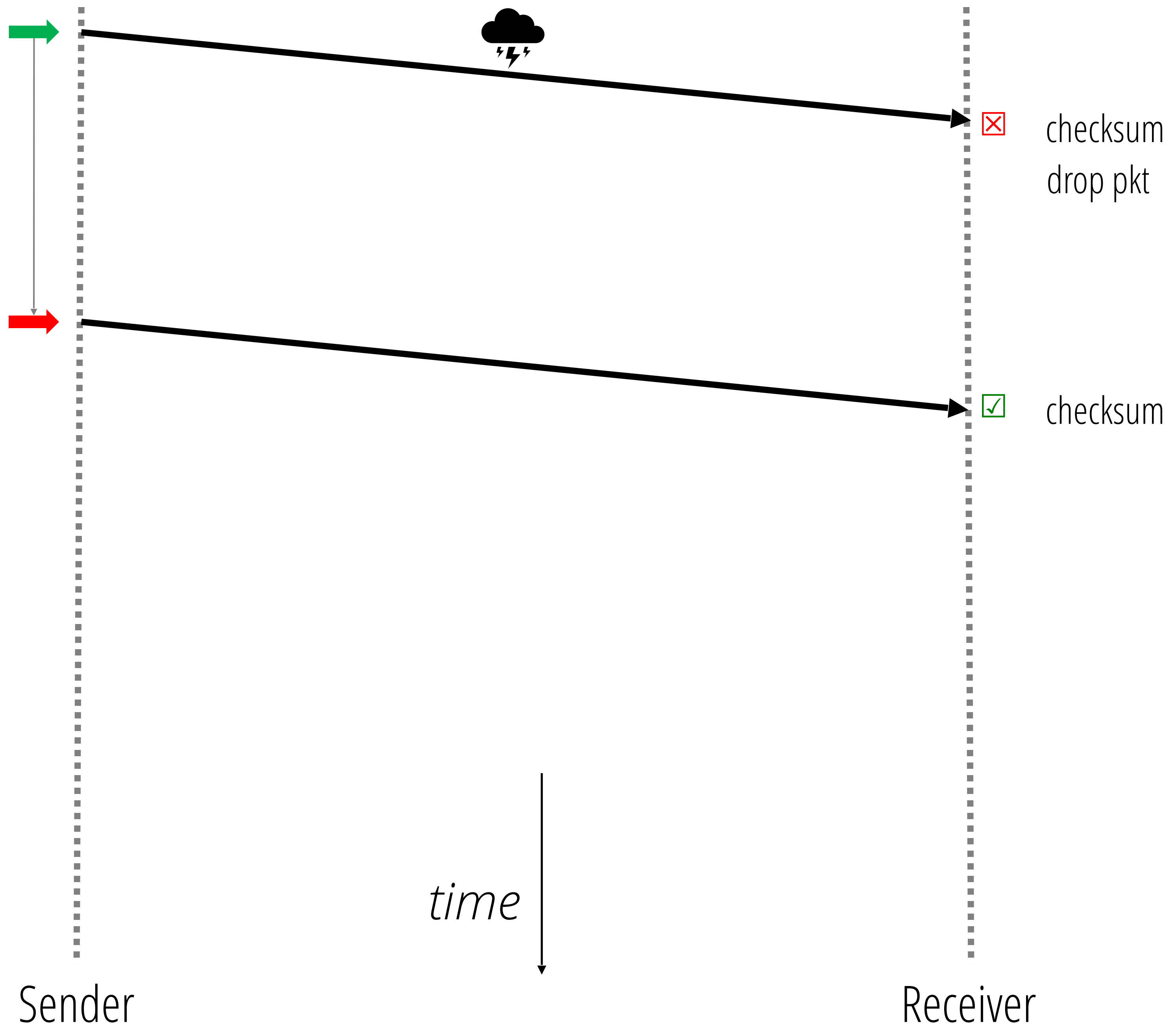

checksum
nack
time
Sender
Receiver


checksum
nack
time
Sender
Receiver

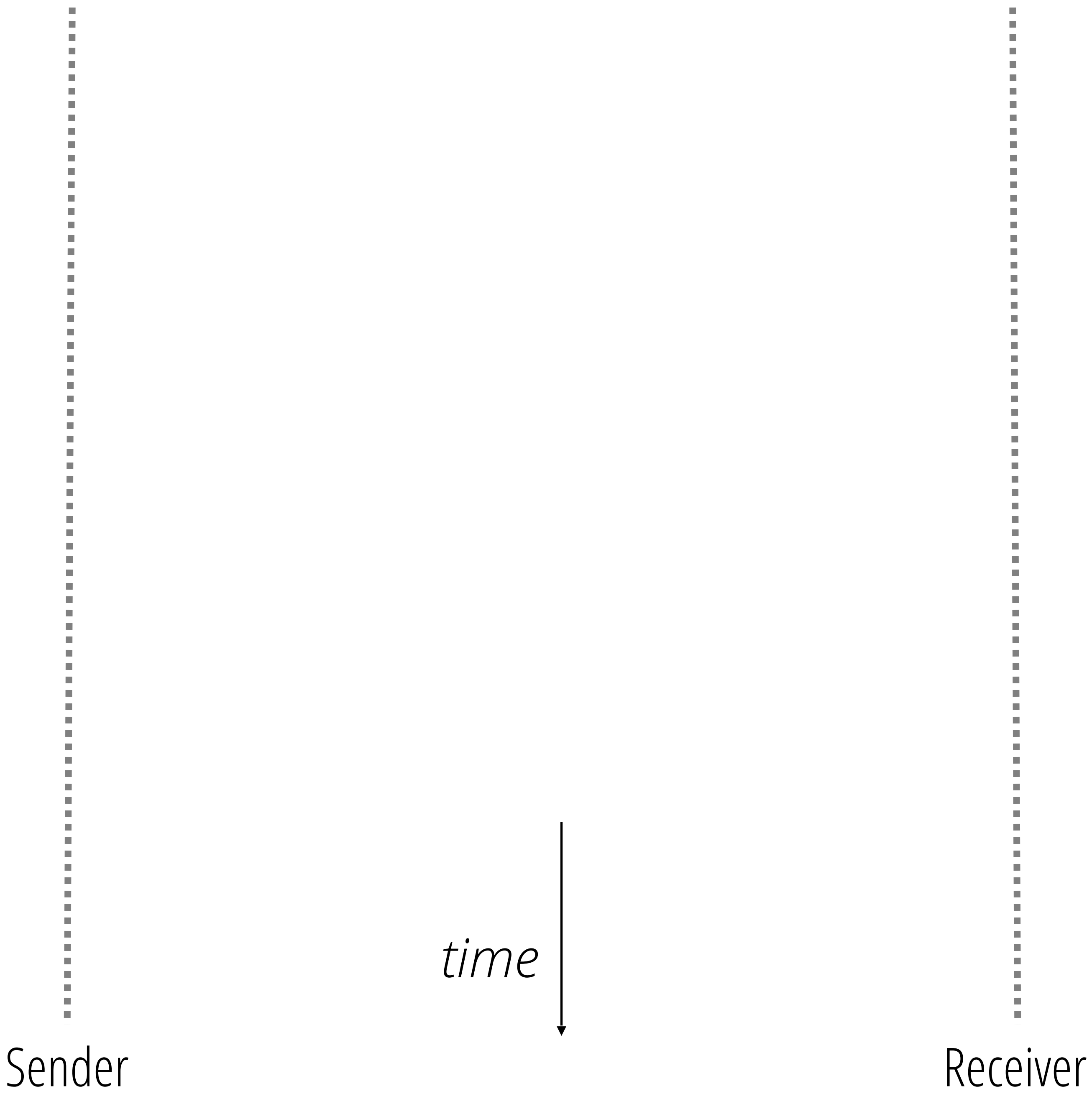

checksum
nack
checksum
time
Sender
Receiver

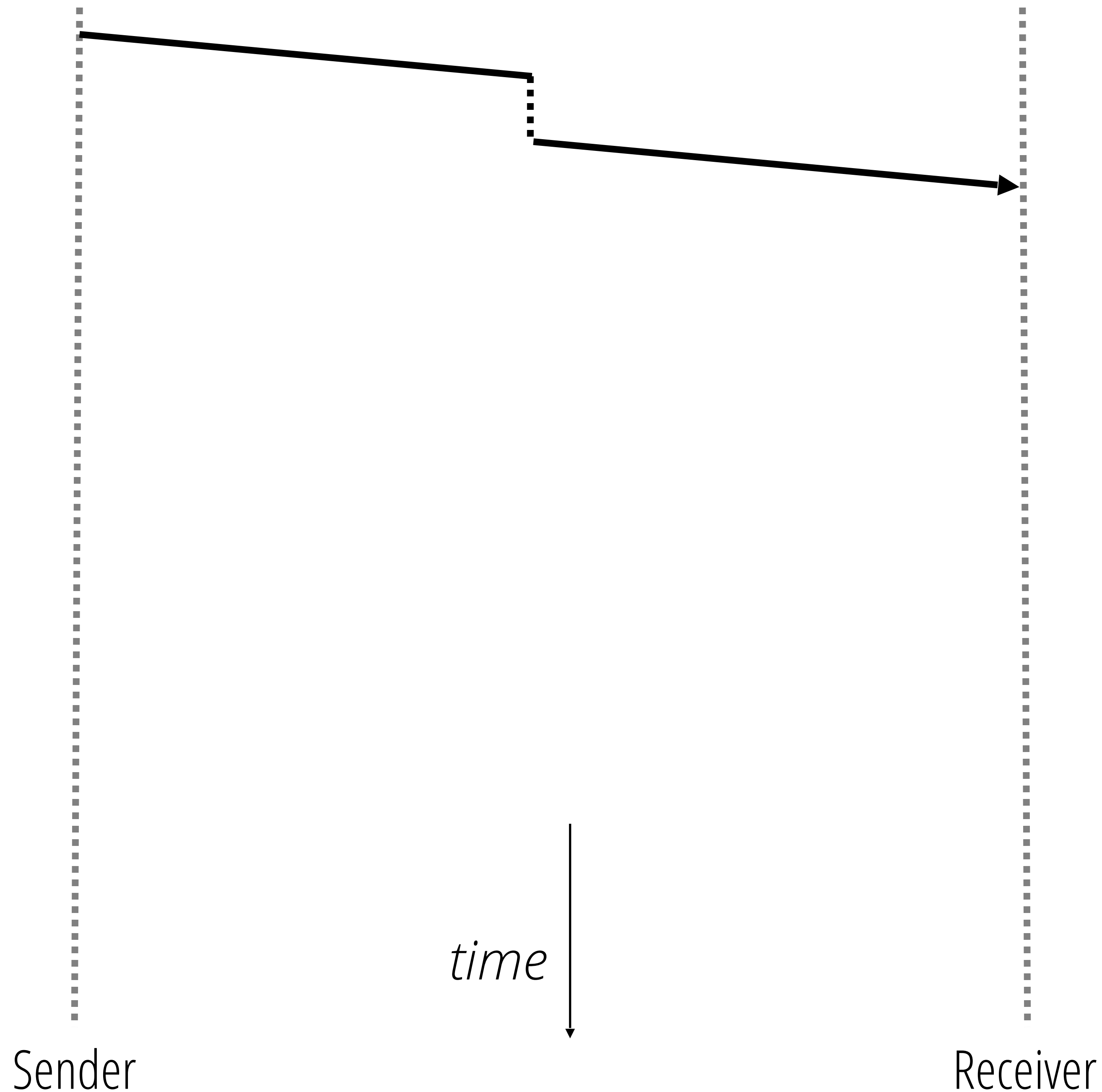

checksum
time
Sender
Receiver

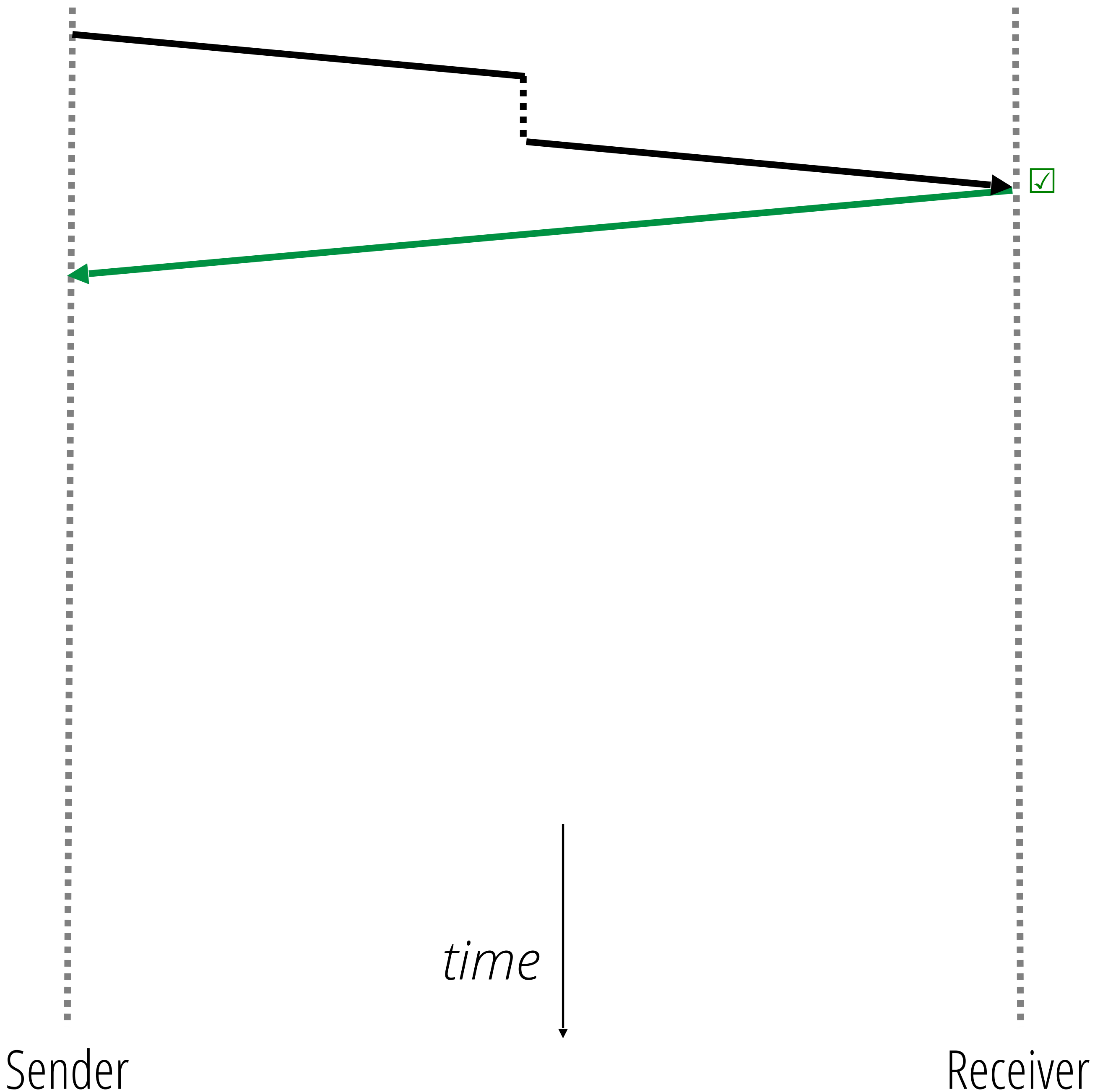

checksum
drop pkt
time
Sender
Receiver

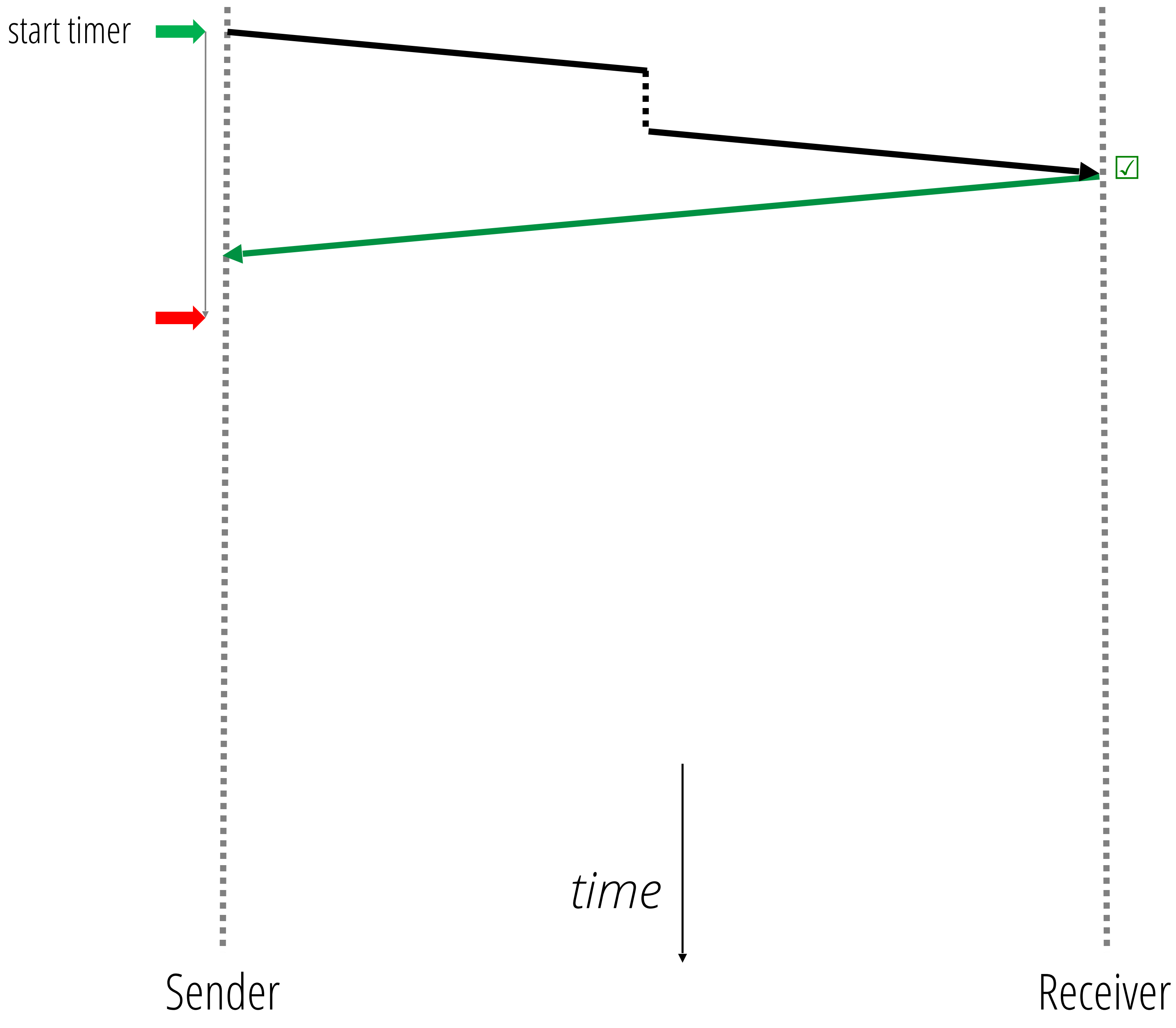

checksum
drop pkt
checksum
time
Sender
Receiver

- 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

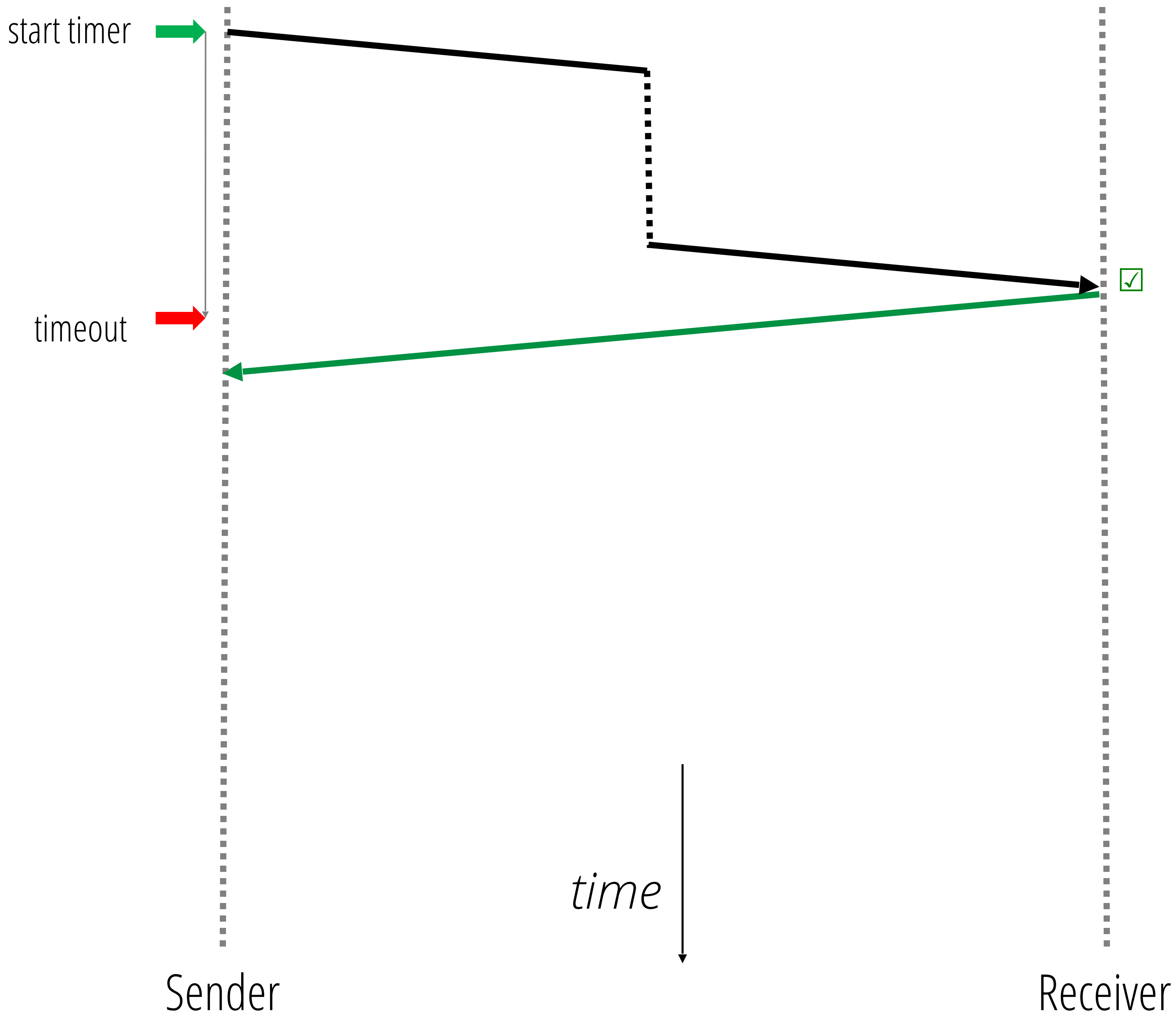

time
Sender
Receiver

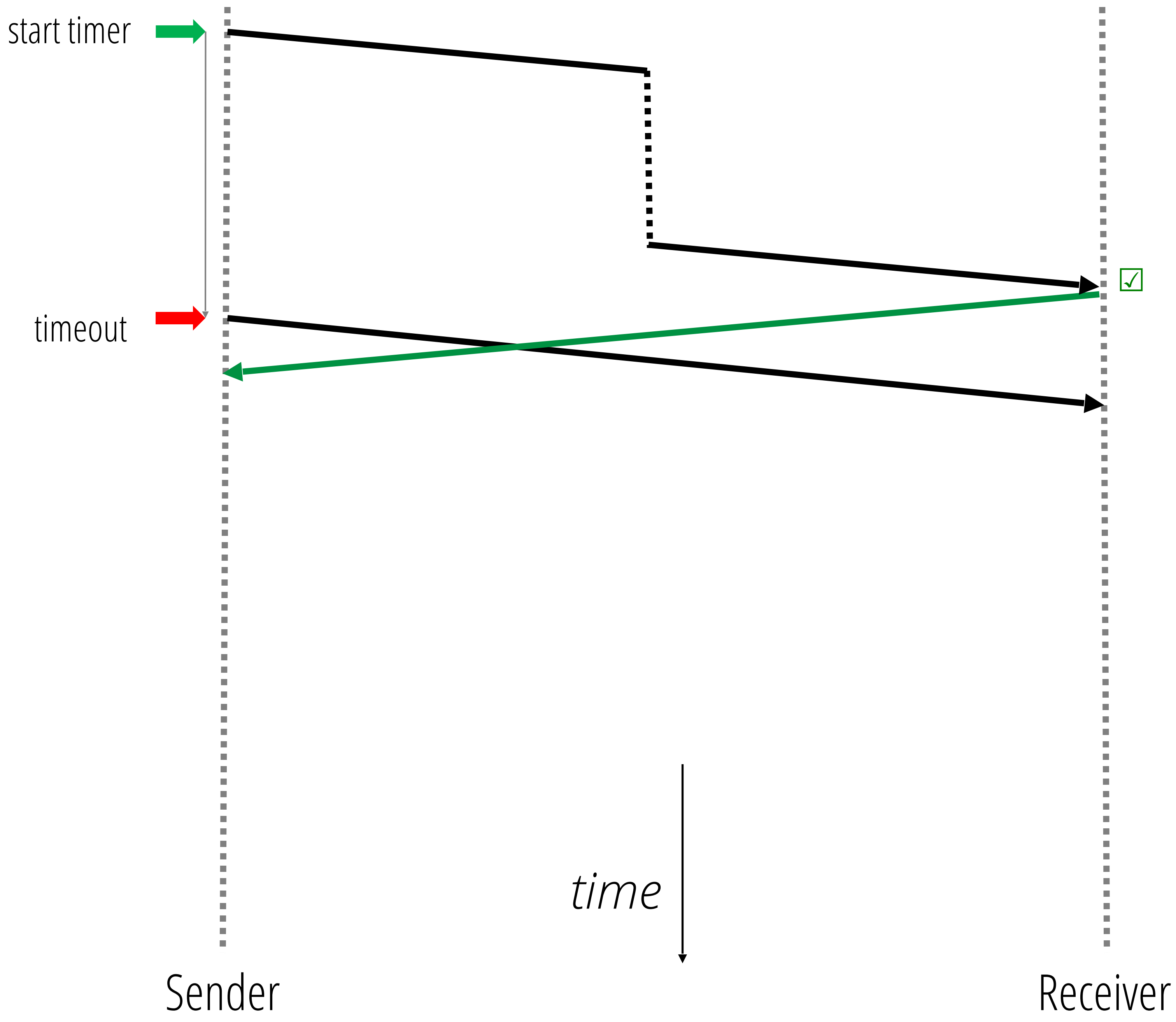

time
Sender
Receiver

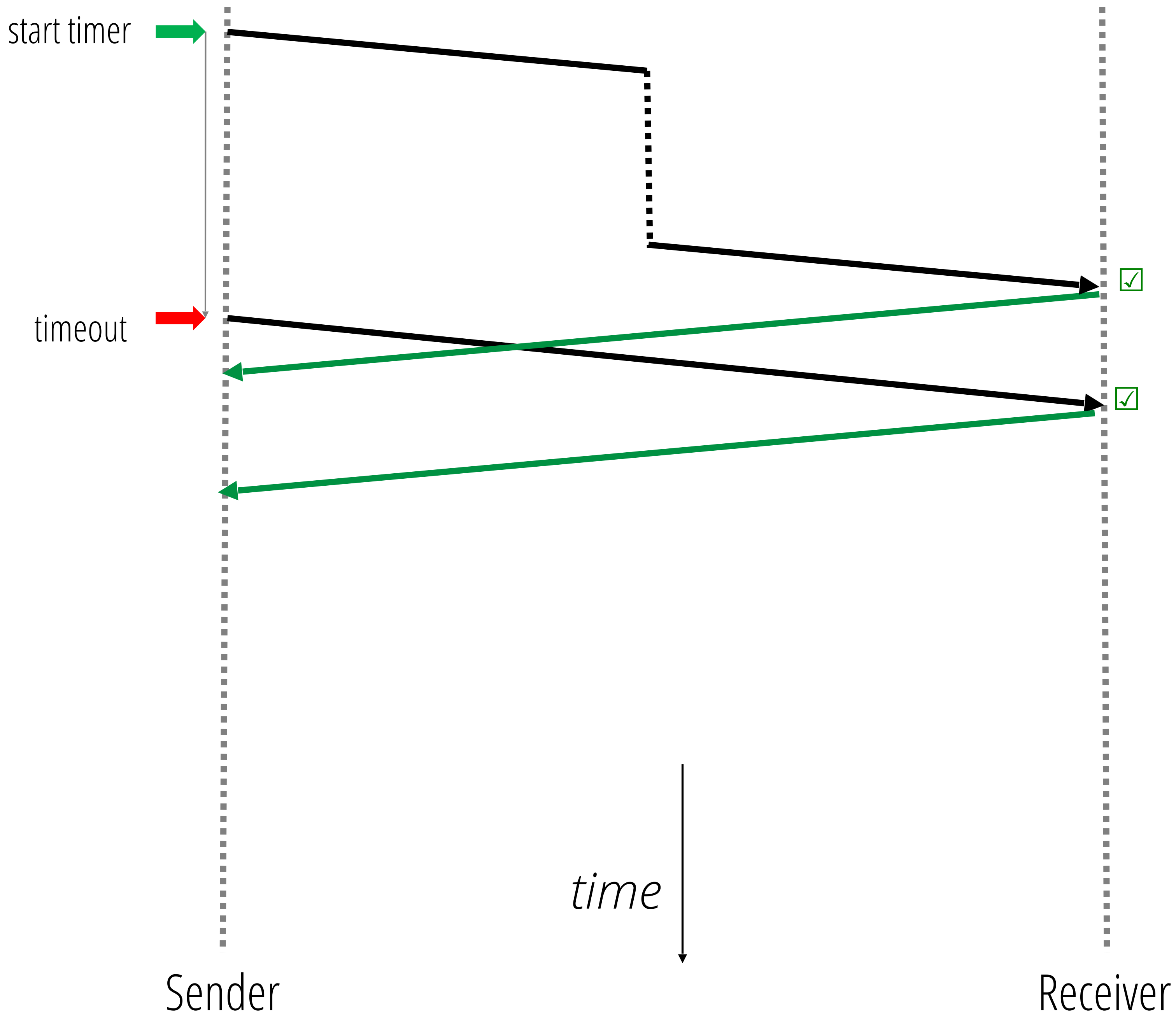

☑
time
Sender
Receiver



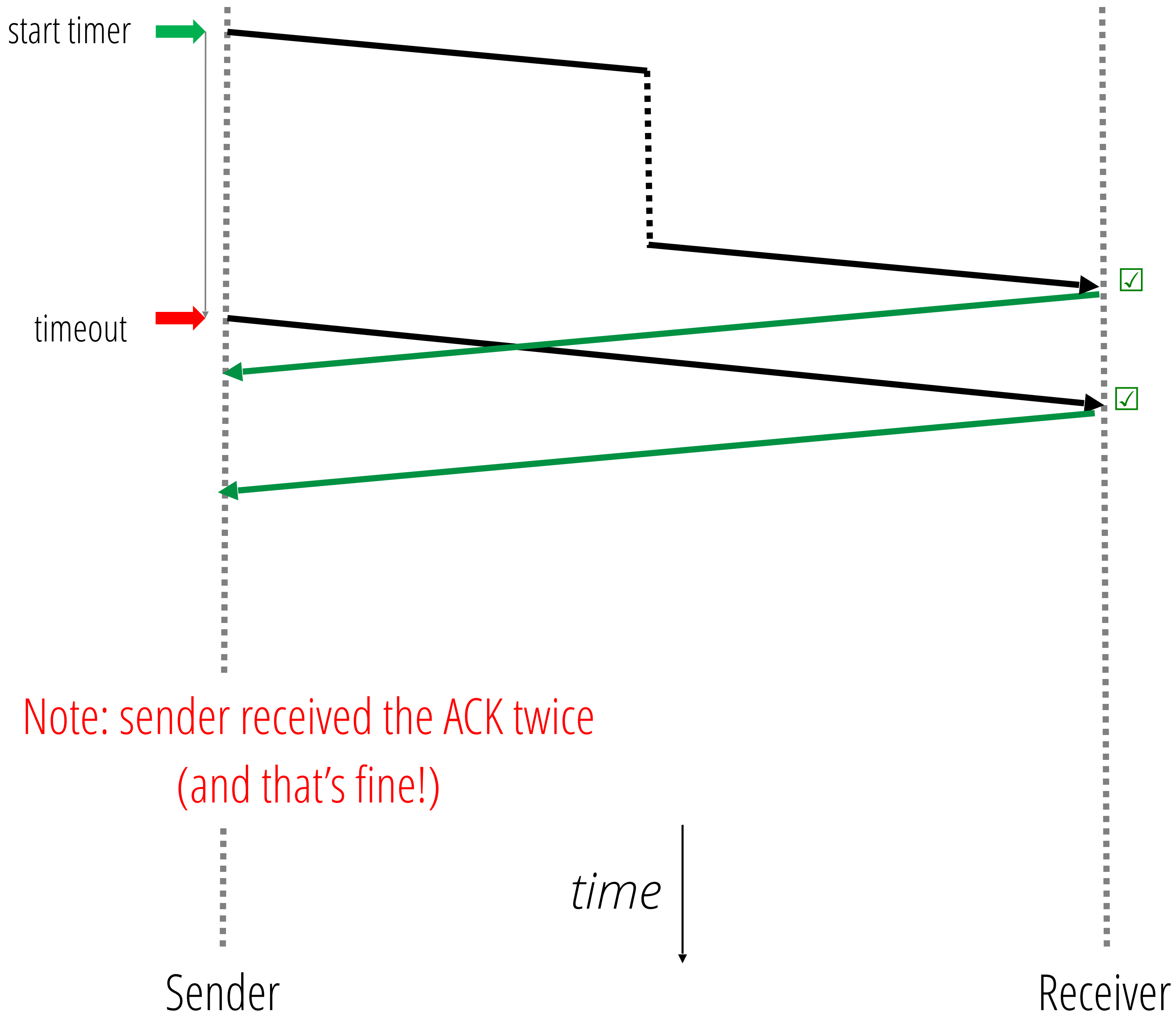
start timer
☑
time
Sender
Receiver


start timer
timeout
☑
time
Sender
Receiver


start timer
timeout
☑
time
Sender
Receiver

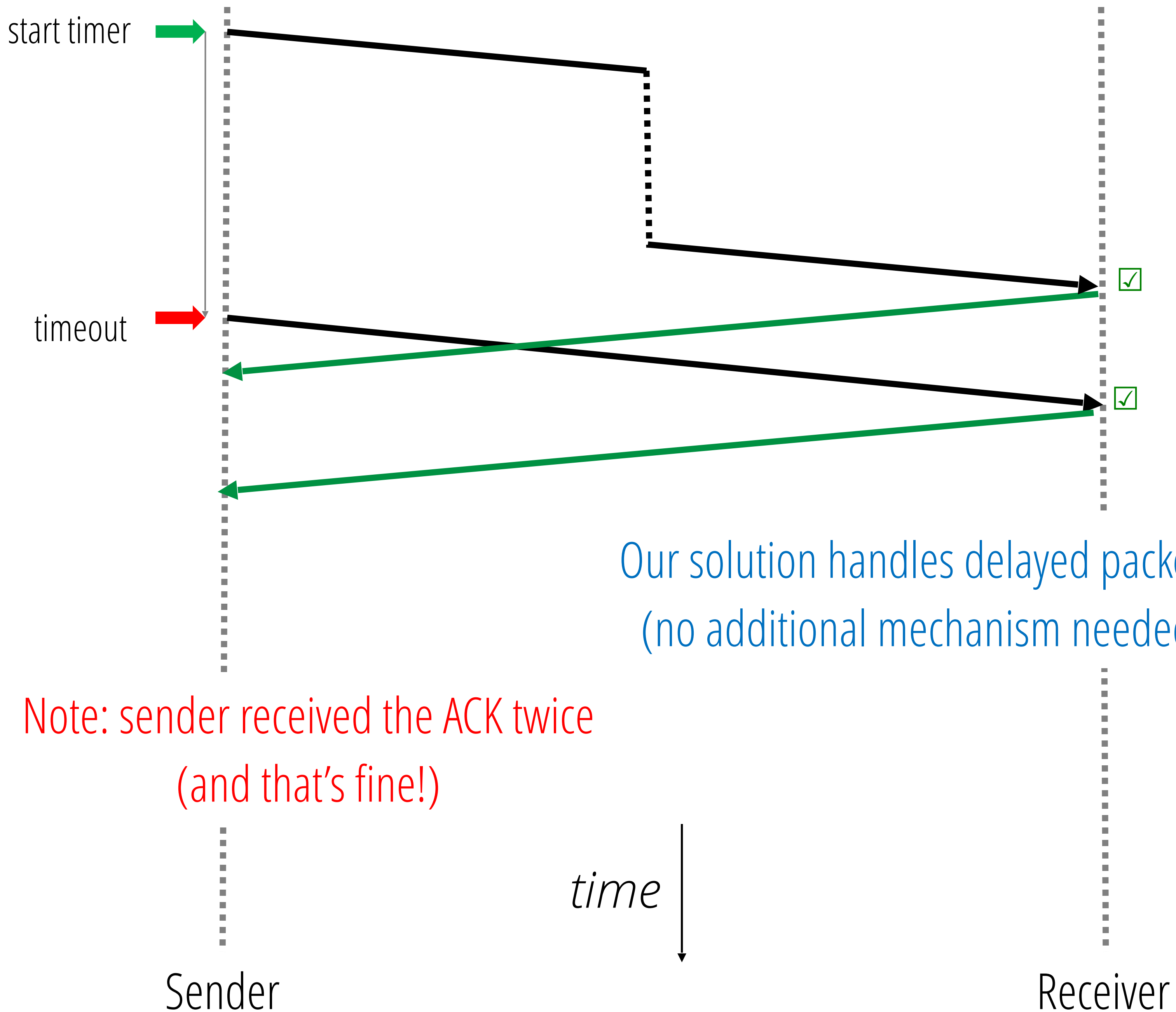


start timer
timeout
☑
☑
time
Sender
Receiver



Note: sender received the ACK twice

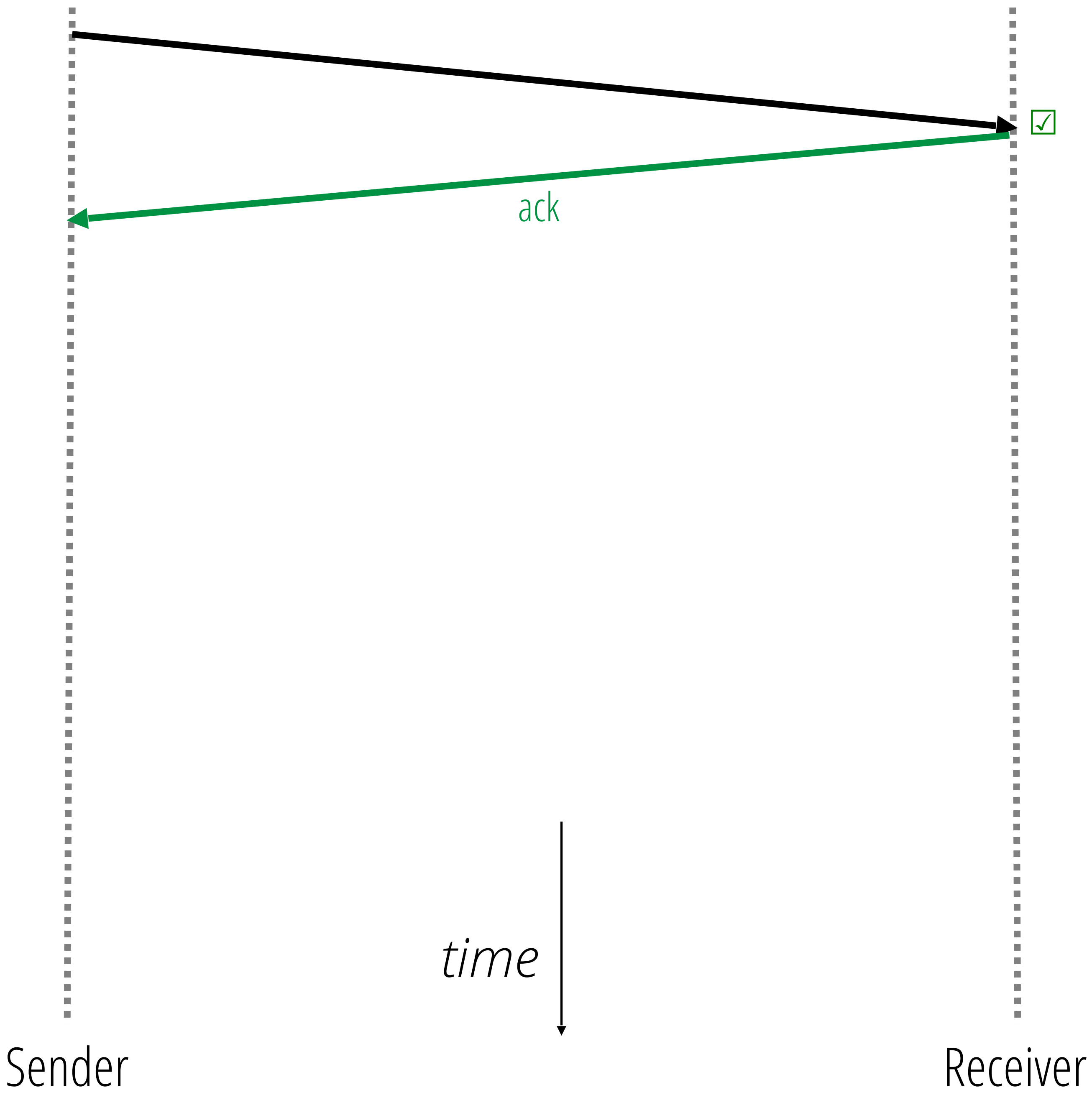
(and that's fine!)

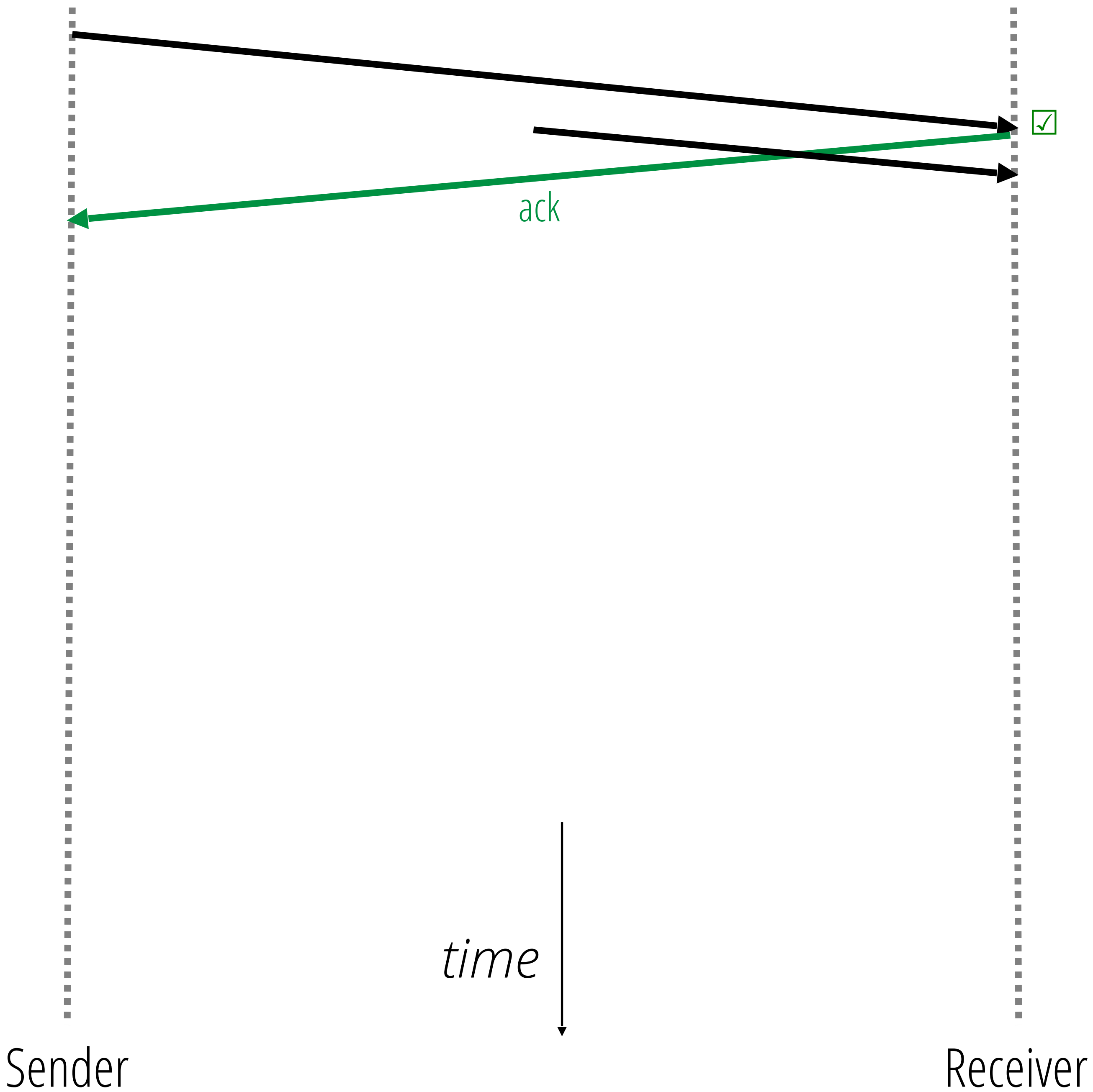


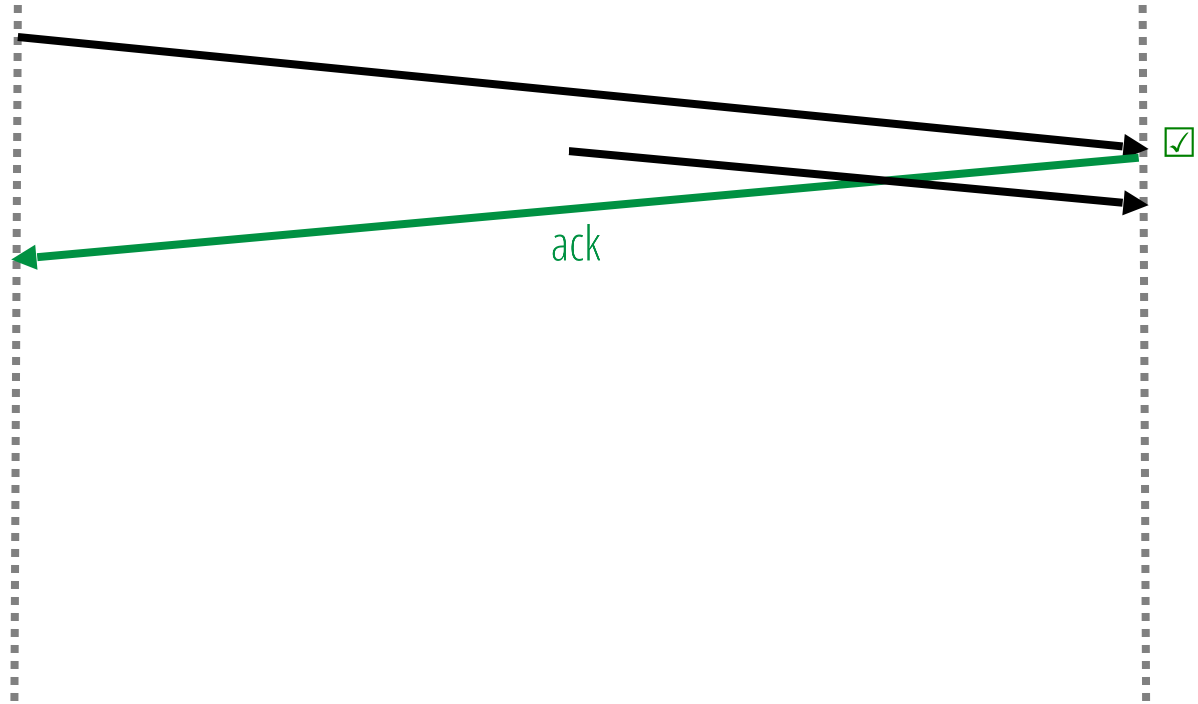
Our solution handles delayed packets!
(no additional mechanism needed)

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

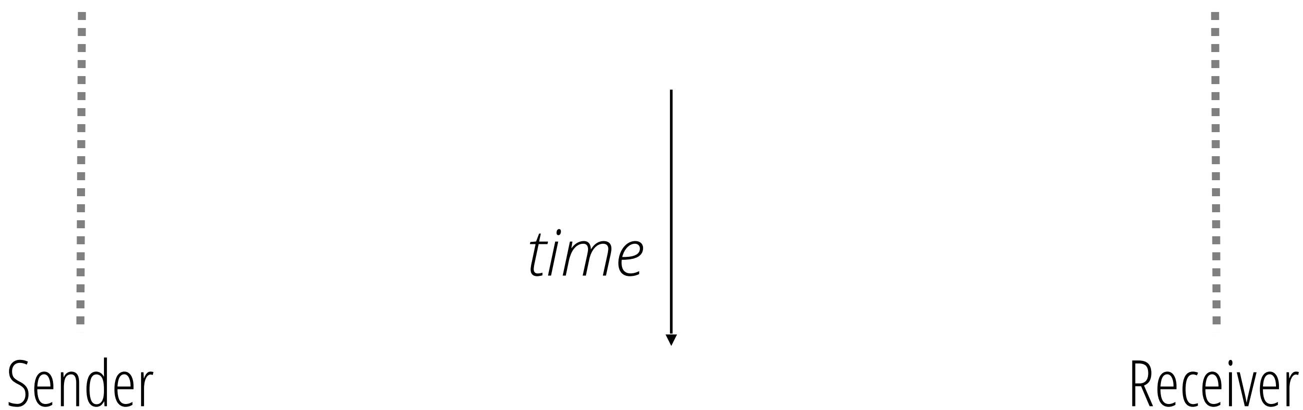

☑
ack
time
Sender
Receiver

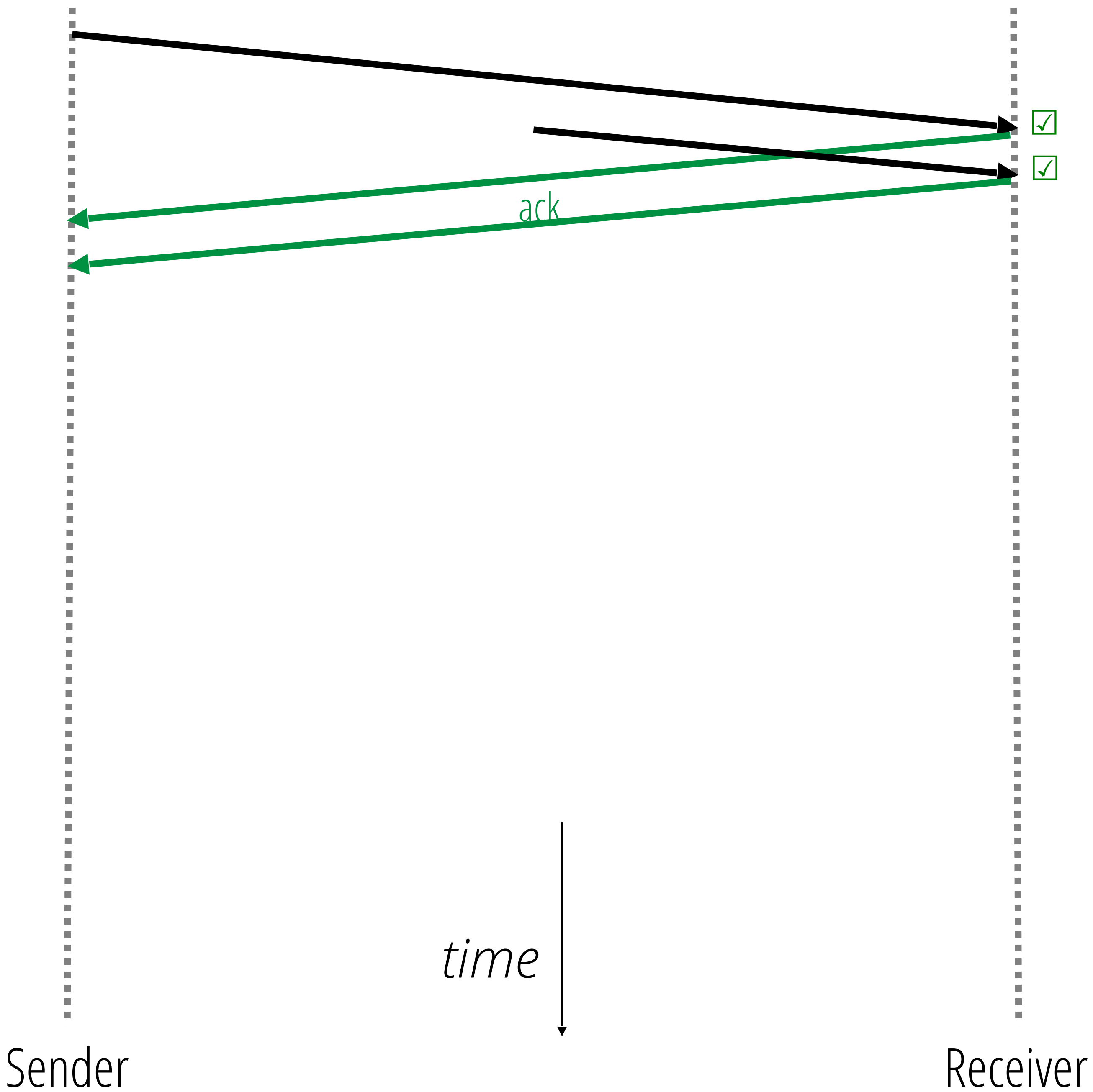

☑
ack
time
Sender
Receiver

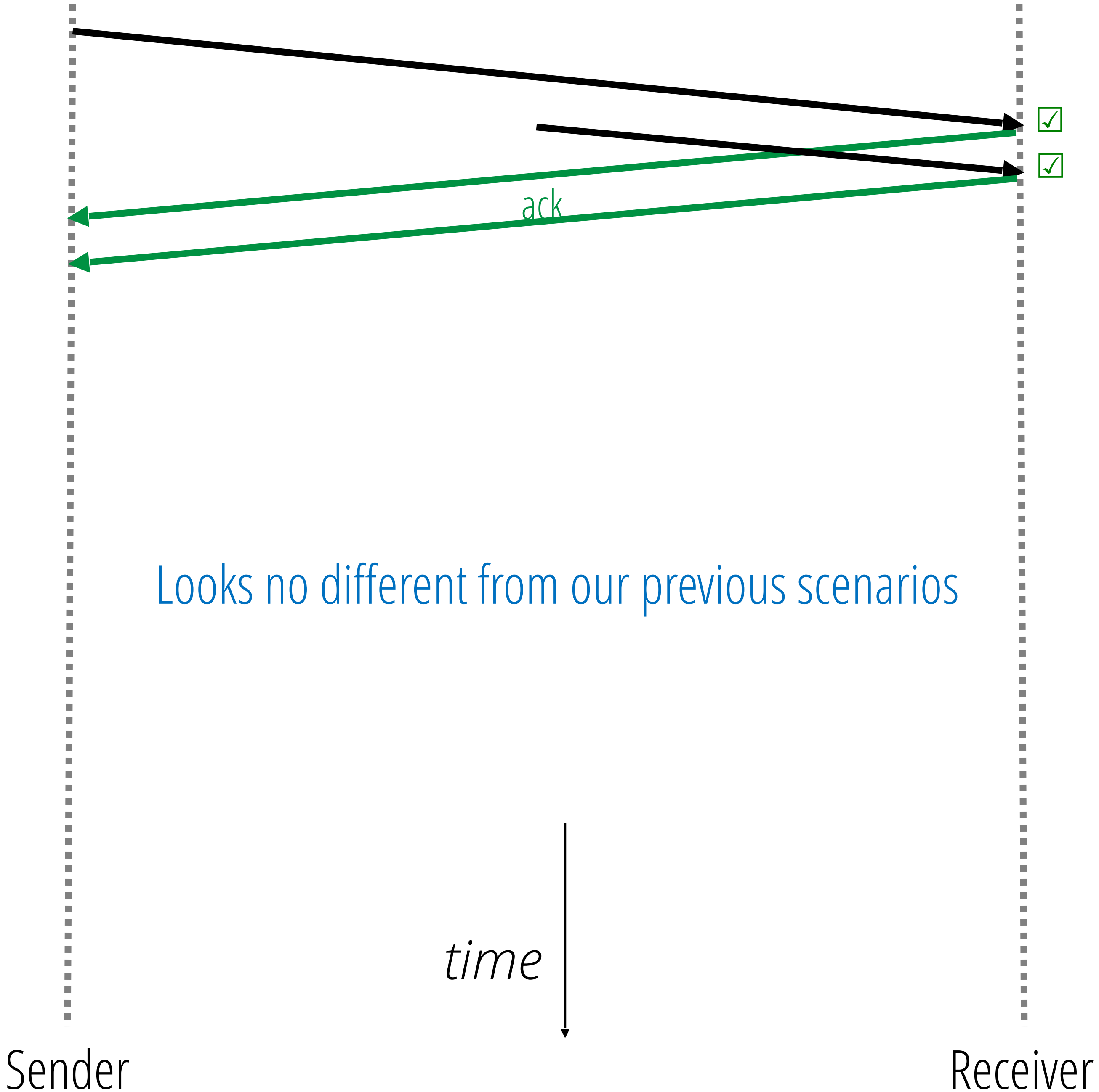


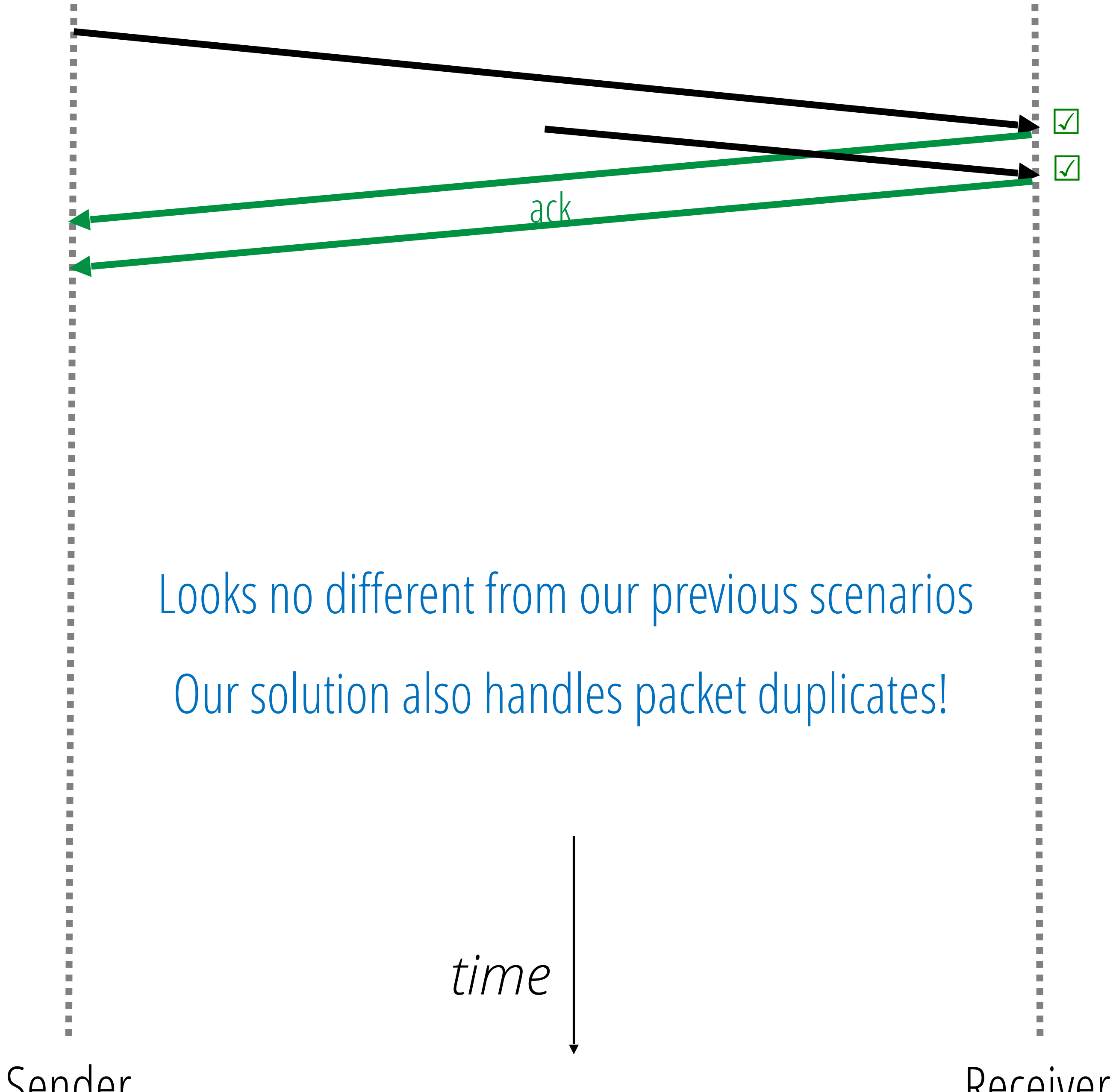
Why would the network even duplicate a packet?

Usually, because of link-level reliability gone wrong (very rare)




☑
☑
ack
time
Sender
Receiver


ack
Looks no different from our previous scenarios
time
Sender
Receiver



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

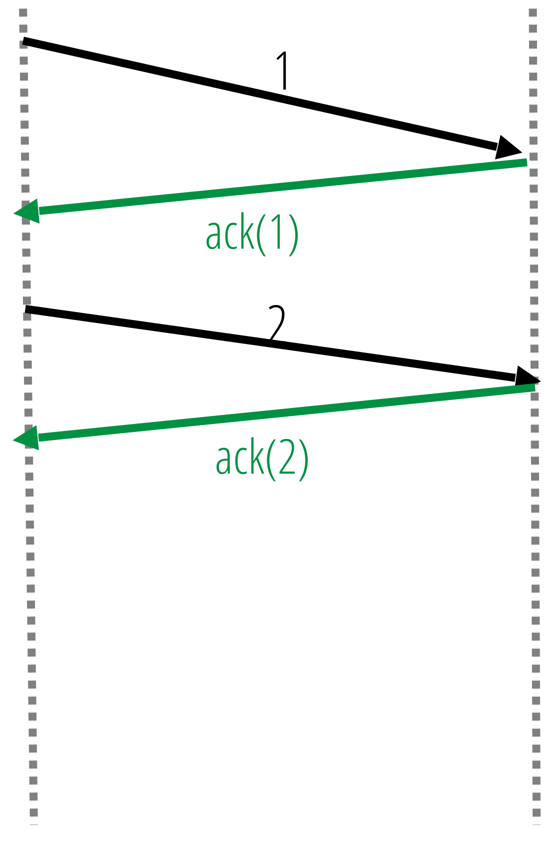
- Semantics of a solution: “at least once”
  - Receiver can receive the same packet more than once
  - Sender can see the same ack/nack more than once

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

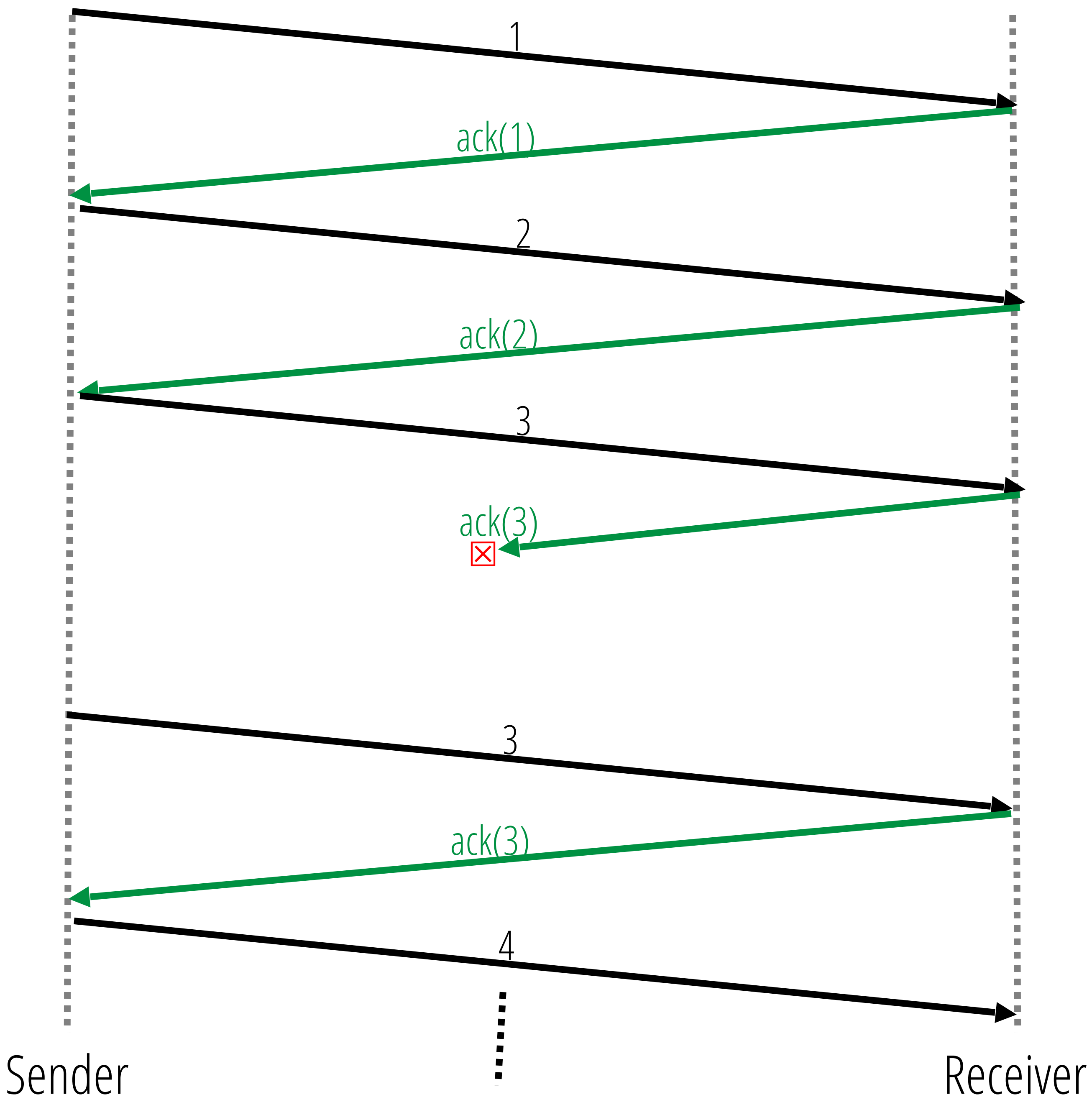
- Use our single-packet solution repeatedly
  - Wait for packet *i* to be acknowledged before sending *i+1*
- We have a correct reliable delivery protocol!

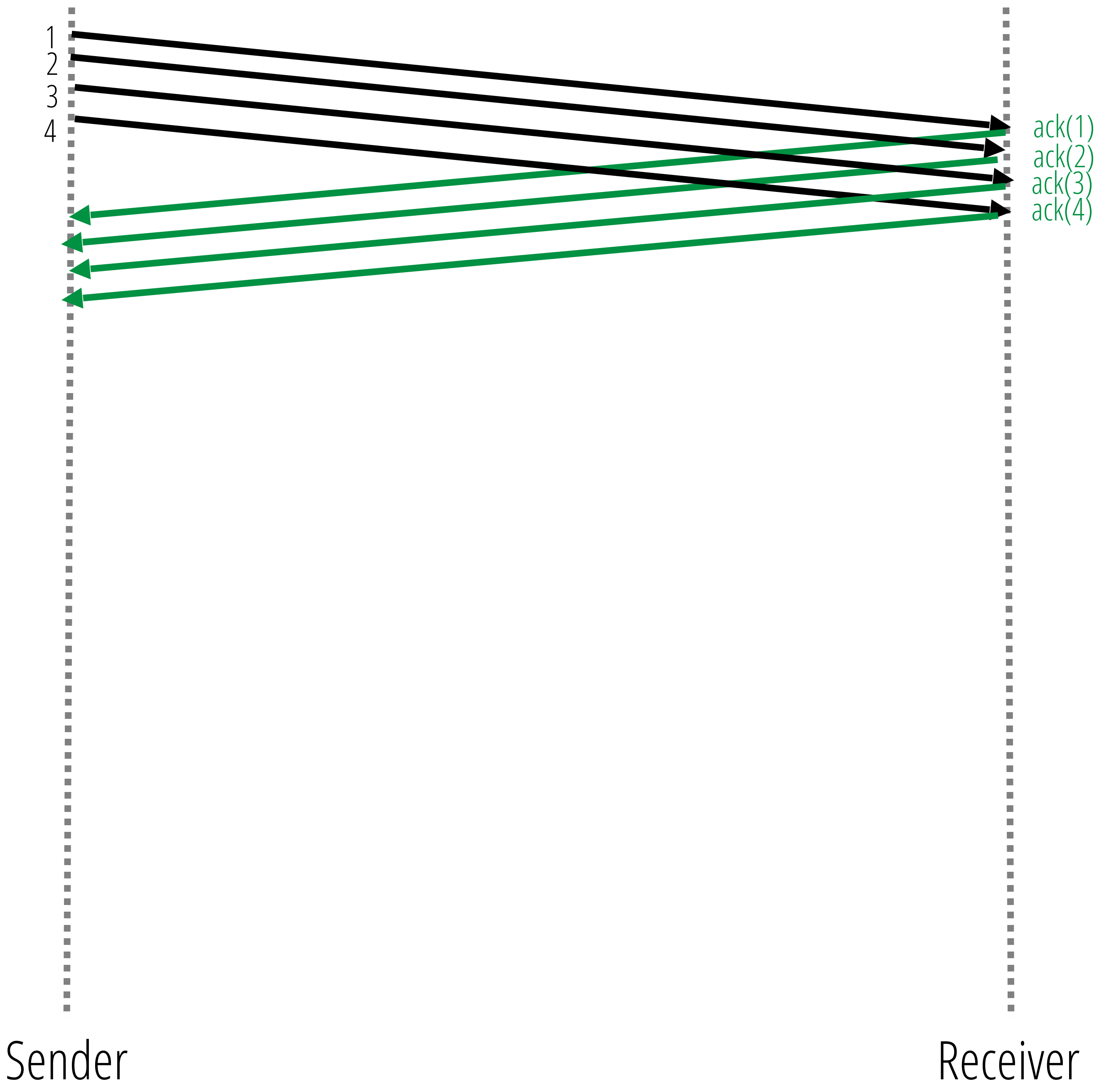
The “Stop and Wait” protocol

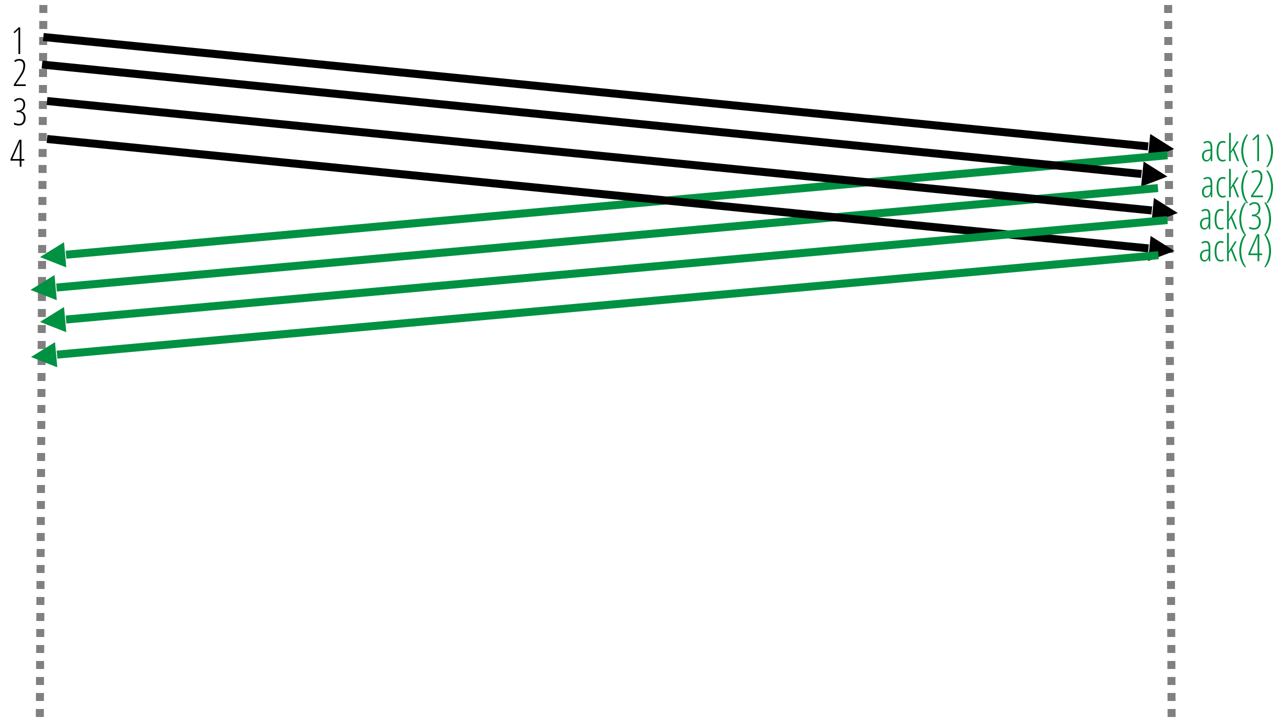
- Use our single-packet solution repeatedly
  - Wait for packet *`i`* to be acknowledged before sending *`i+1`*

- We have a correct reliable delivery protocol!

- Probably the world’s most inefficient one
  - Max throughput ~ one packet per RTT


1
ack(1)
2
ack(2)
3
ack(3)
3
ack(3)
4
Sender
Receiver


1
2
3
4
ack(1)
ack(2)
ack(3)
ack(4)
Sender
Receiver



Idea: have multiple packets "in flight"

(send additional packets while waiting for ACKs to come in)

Window-based Algorithms

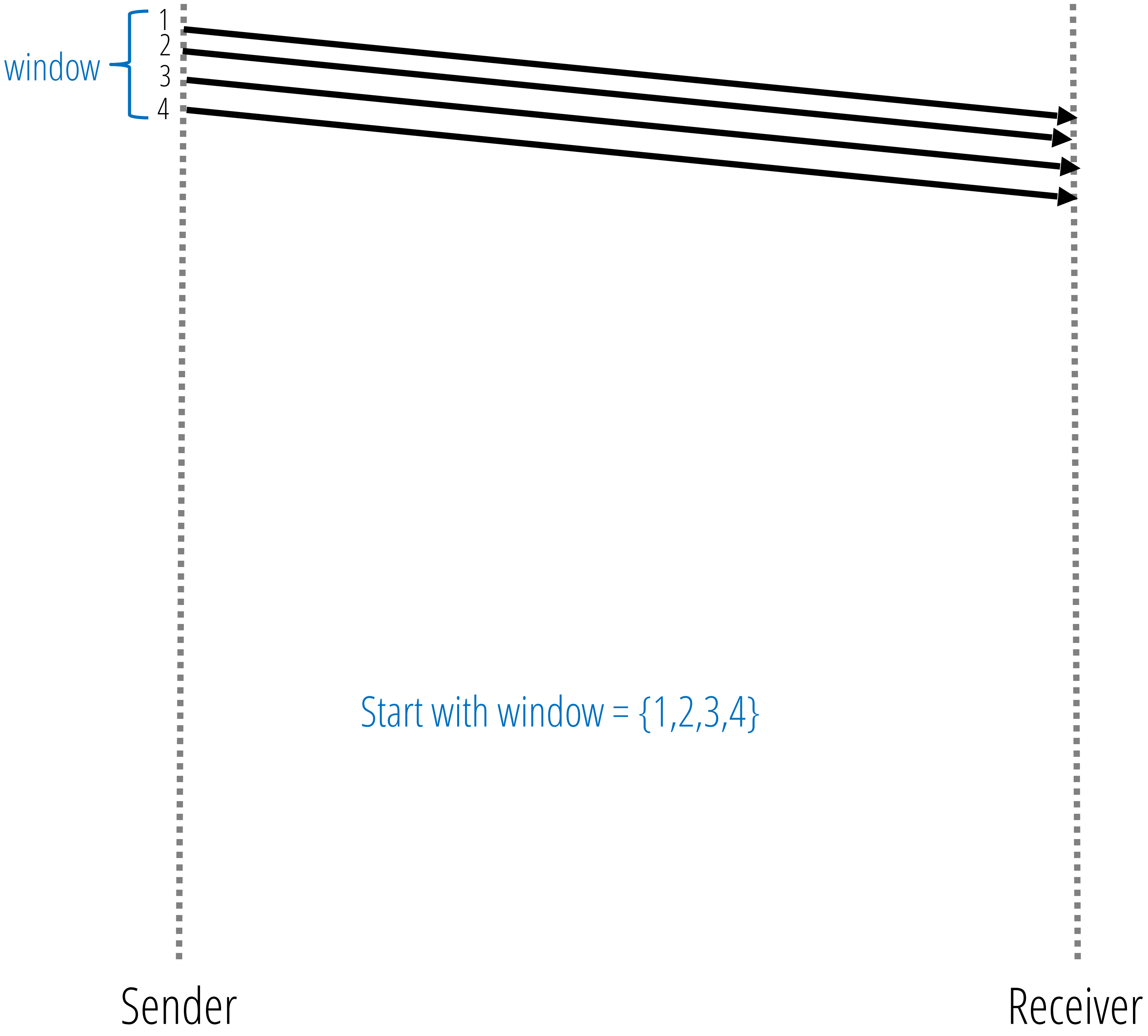
Window-based Algorithms

- Basic idea: allow **W** packets "in flight" at any time
  - W is the size of the window

Window-based Algorithms

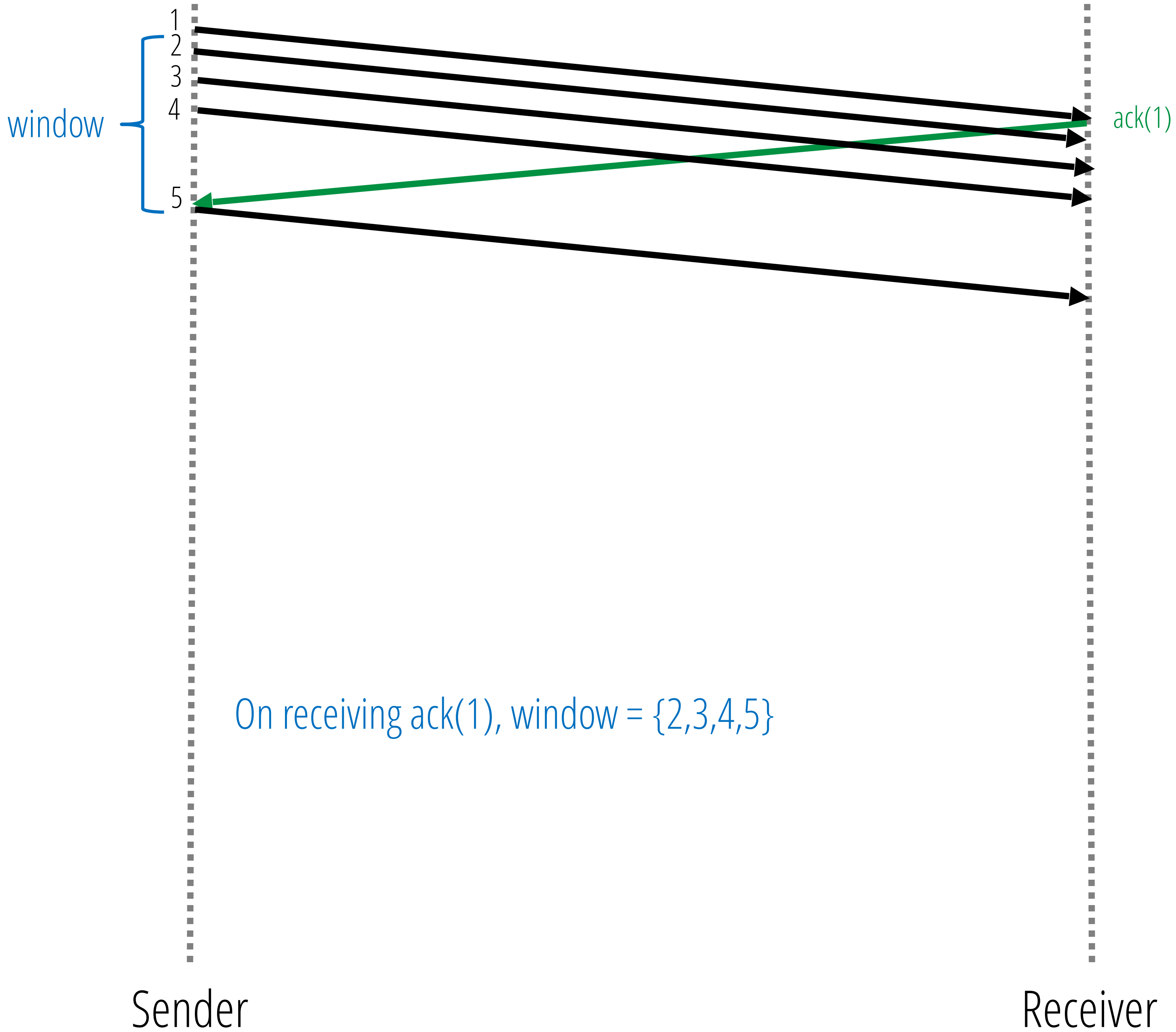
- Basic idea: allow **W** packets "in flight" at any time
  - W is the size of the window
- Hence, a simple algorithm (at sender)
  - Send W packets
  - When one gets ACK'ed, send the next packet in line

Example with W=4



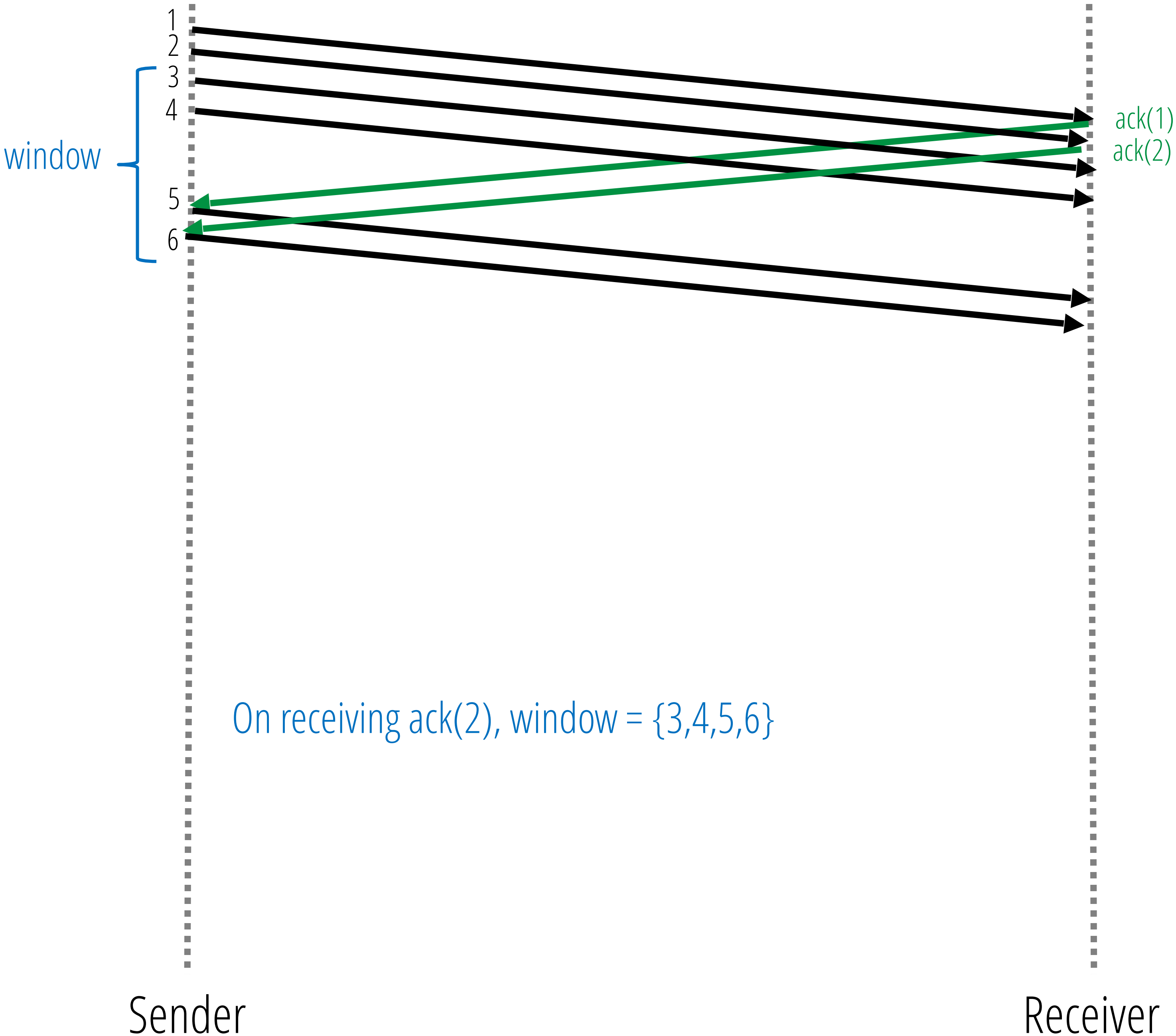
Start with window = {1,2,3,4}

Example with W=4



On receiving ack(1), window = {2,3,4,5}

Example with W=4



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?

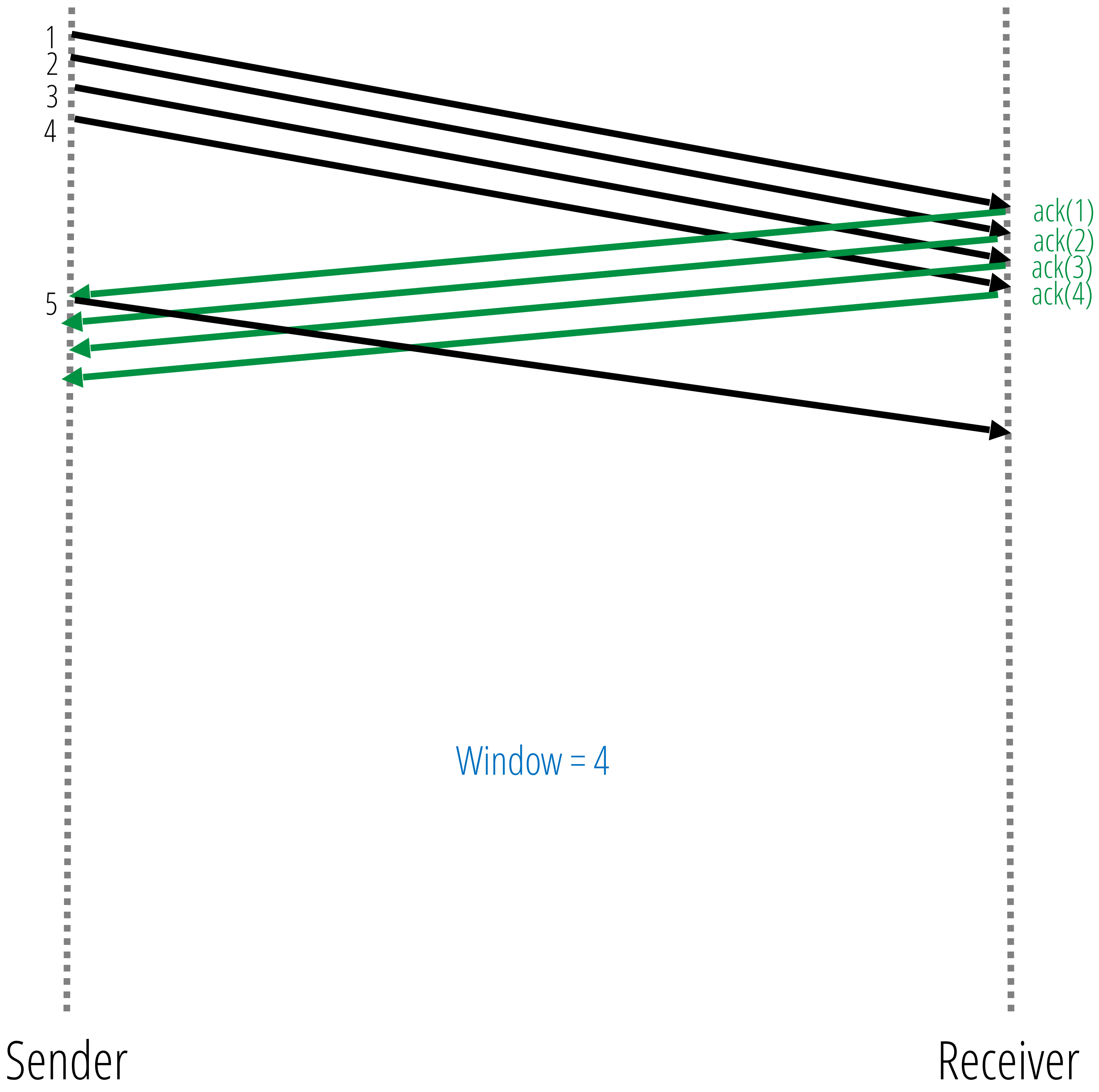
- Pick window size **W** to balance three goals
  - Take advantage of network capacity (“fill the pipe”)
  - But don’t overload links (congestion control)
  - And don’t overload the receiver (flow control)

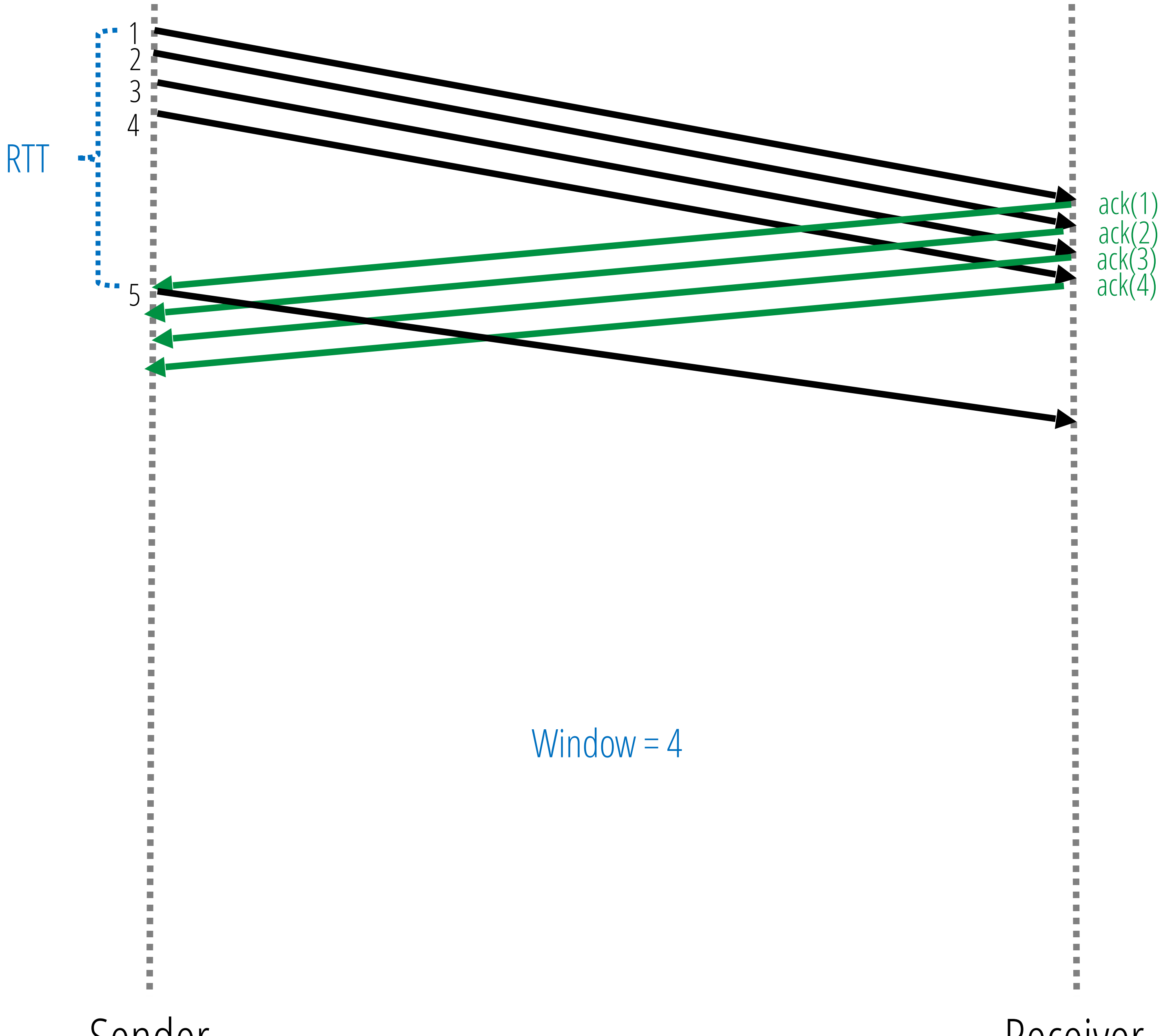
- If we ignore all but the first goal then we want to keep the sender always sending (ideal case)

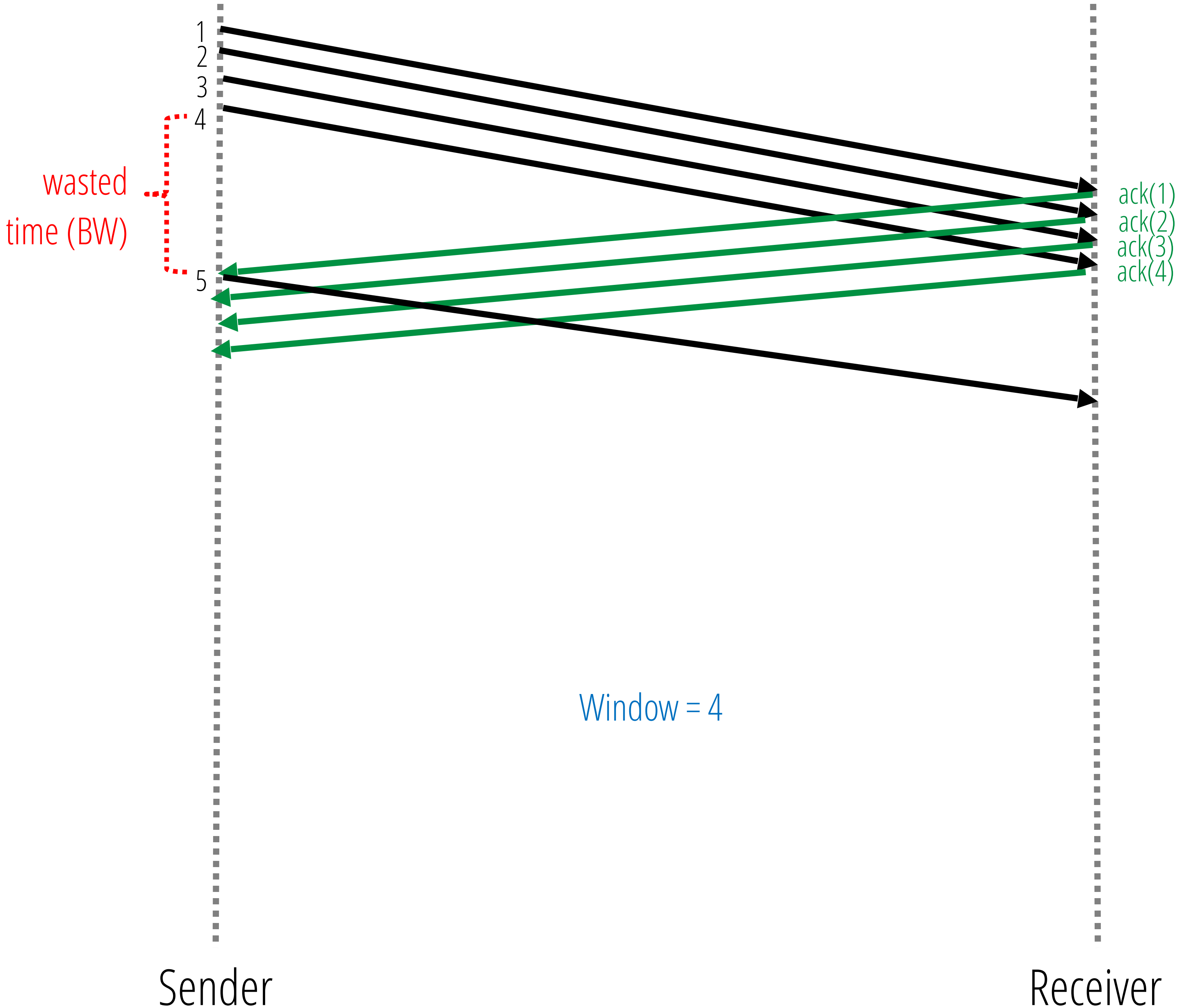
How big should the window be?

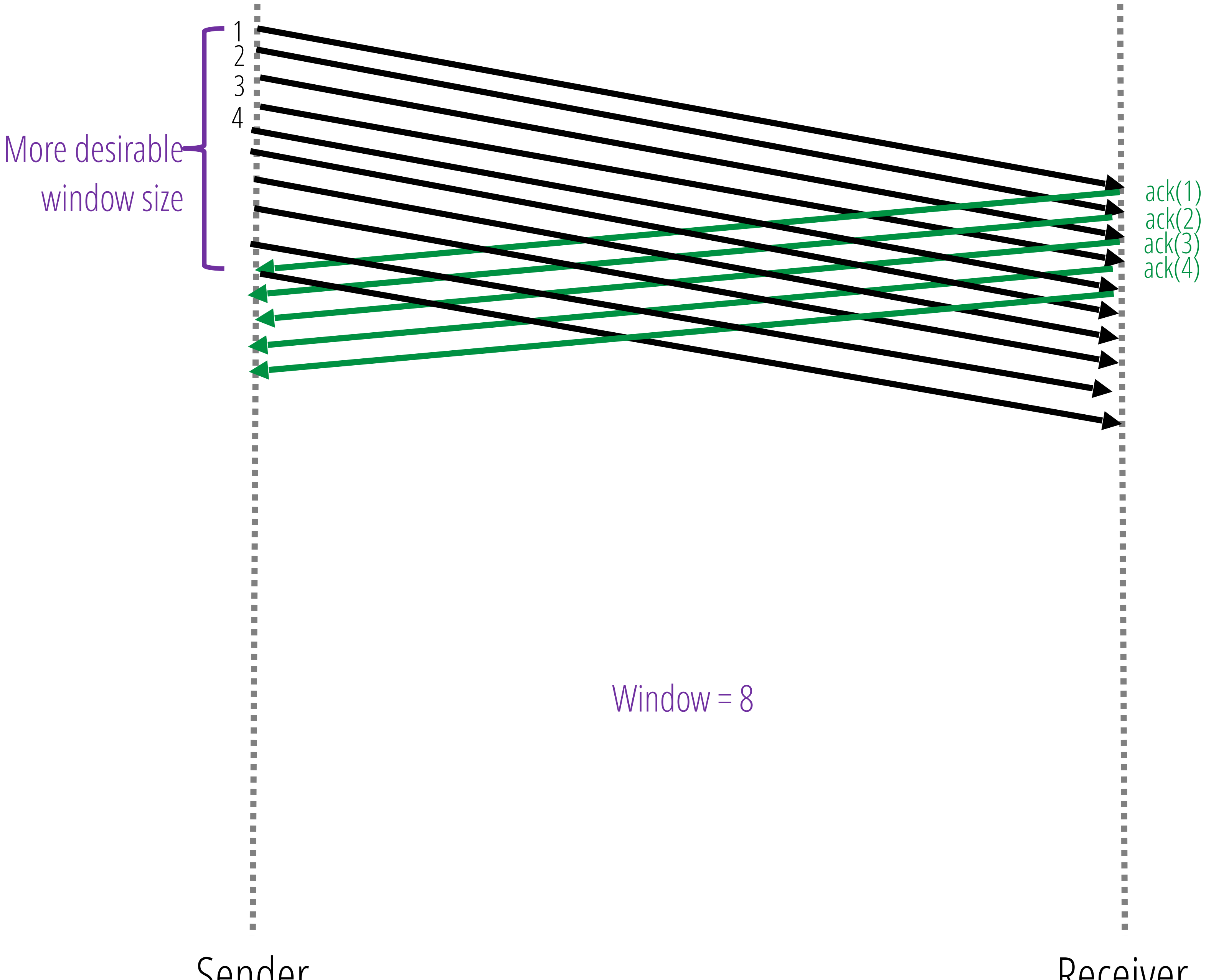
- Pick window size **W** to balance three goals
  - Take advantage of network capacity (“fill the pipe”)
  - But don’t overload links (congestion control)
  - And don’t overload the receiver (flow control)

- 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


1
2
3
4
ack(1)
ack(2)
ack(3)
ack(4)
5
Window = 4
Sender
Receiver


1
2
3
4
RTT
5
ack(1)
ack(2)
ack(3)
ack(4)
Window = 4
Sender
Receiver


1
2
3
4
wasted
time (BW)
5
ack(1)
ack(2)
ack(3)
ack(4)
Window = 4
Sender
Receiver


More desirable window size
1
2
3
4
ack(1)
ack(2)
ack(3)
ack(4)
Window = 8
Sender
Receiver

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?

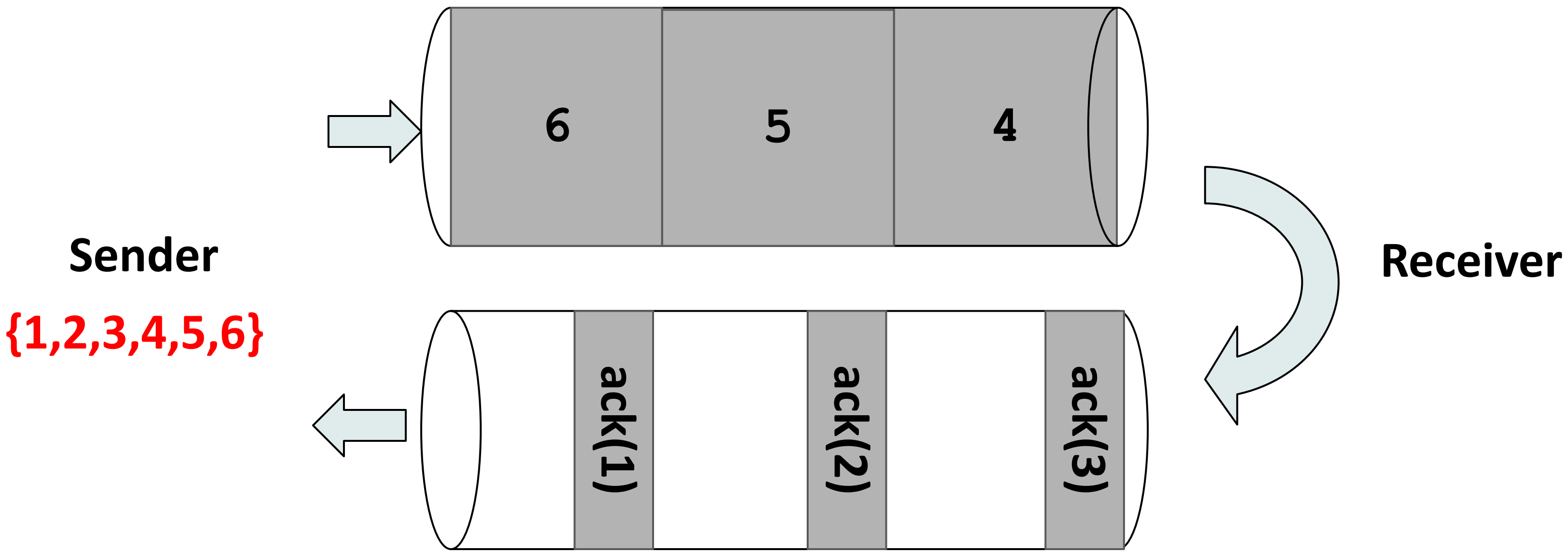
- Let B be the minimum (“bottleneck”) link bandwidth along the path
  - Obviously shouldn’t send faster than that
  - Don’t want to send slower than that (for first goal)

- 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 x Packet-Size ~ RTT x B
  - E.g., for a path with RTT=1 second and bottleneck B = 8 Mbits/second, if packet size = 100 Bytes, we want a window size W = 10,000 packets

Setting W to be one RTT of packets

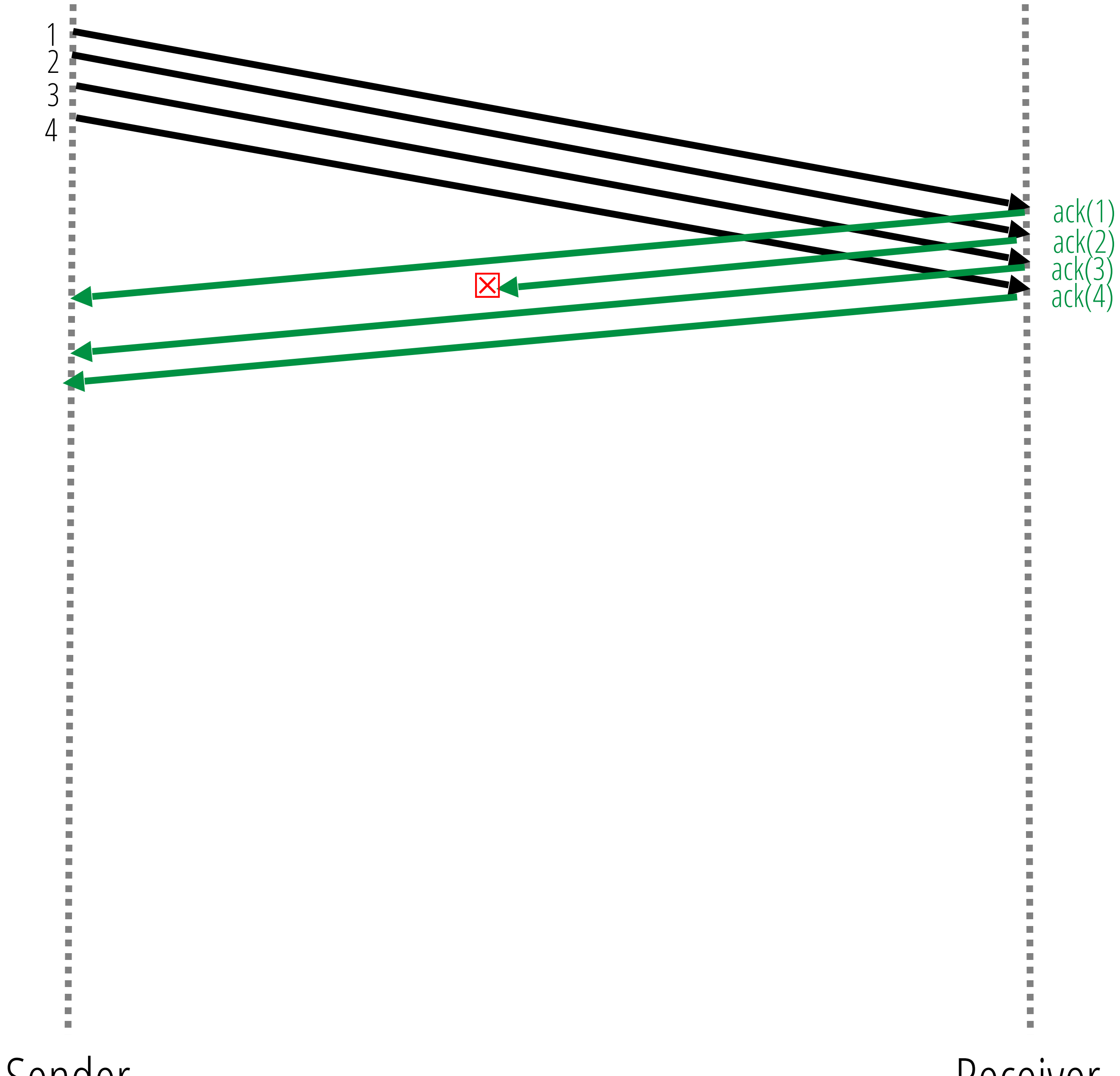


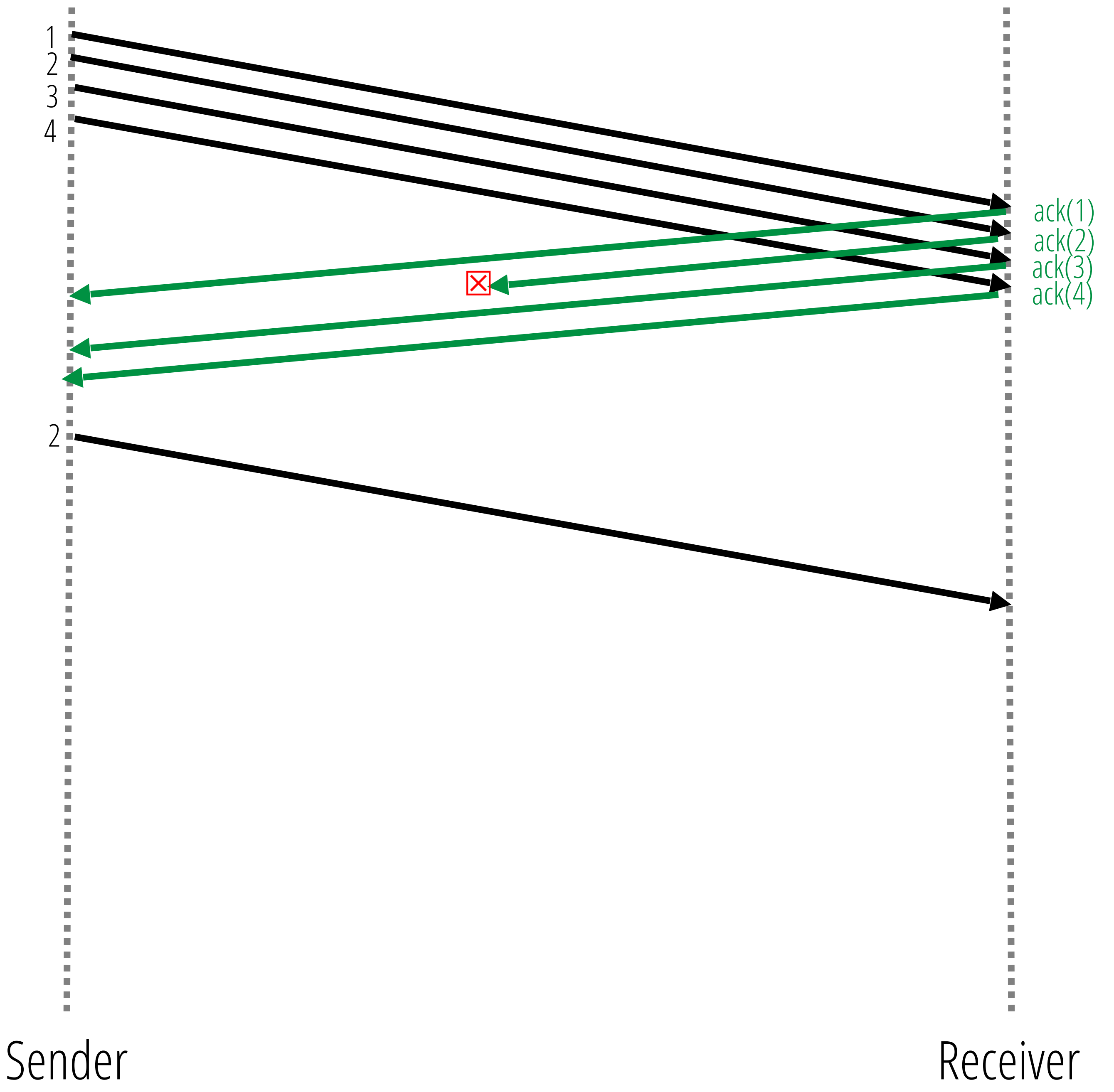
New Design Considerations

- 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?

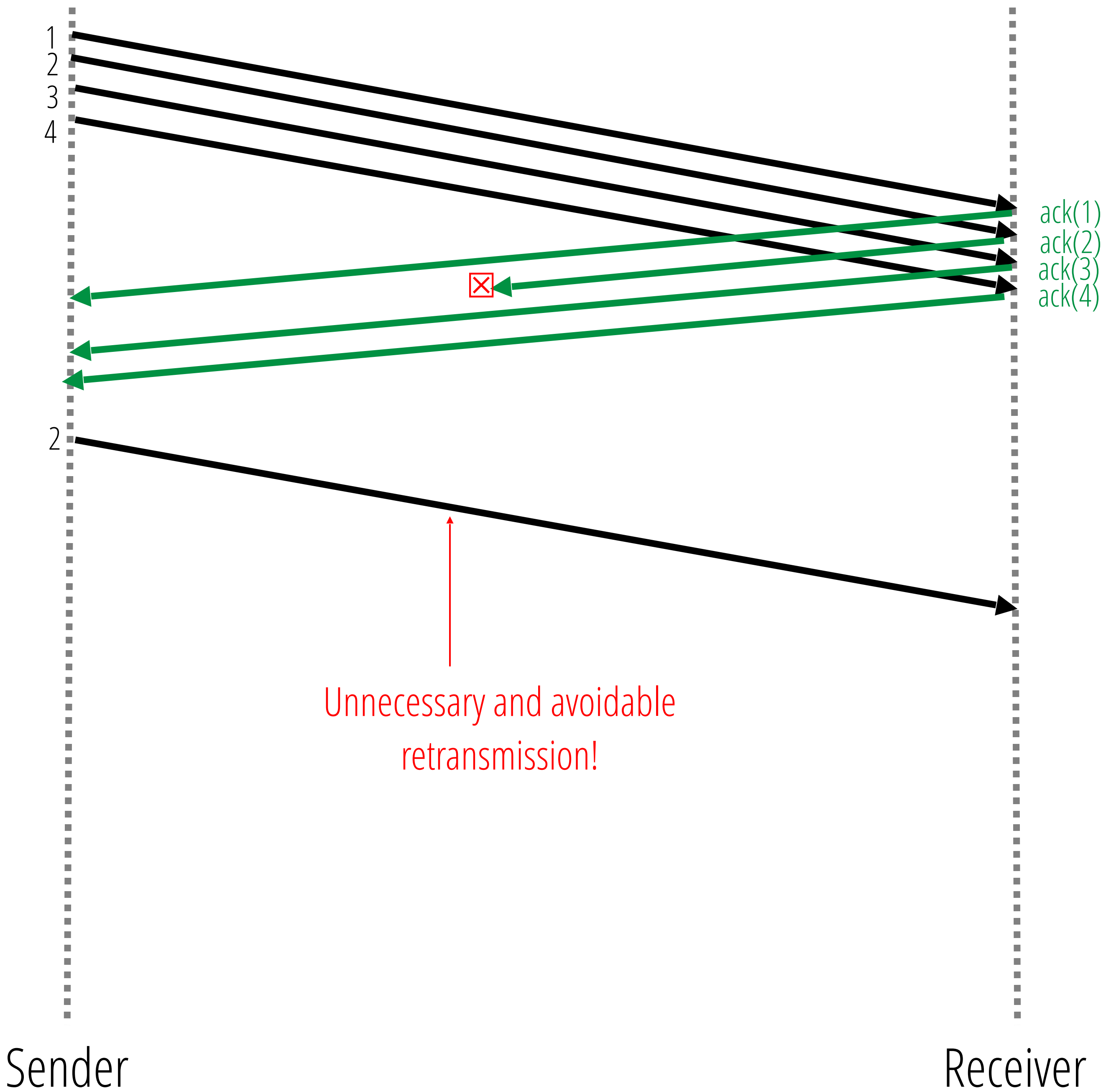
ACKs: design options

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


1
2
3
4
ack(1)
ack(2)
ack(3)
ack(4)
Sender
Receiver



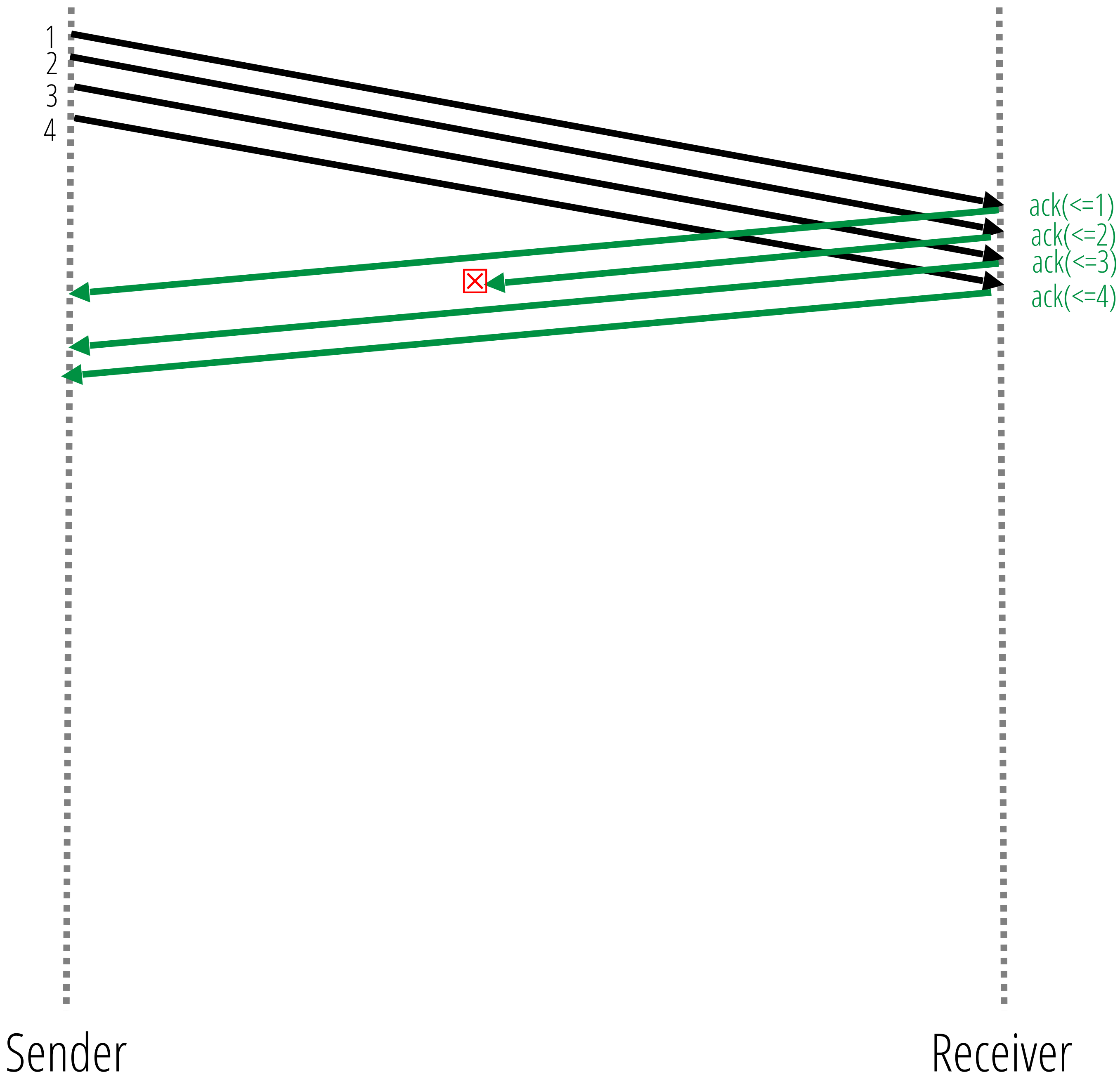
1
2
3
4
ack(1)
ack(2)
ack(3)
ack(4)
2
Sender
Receiver

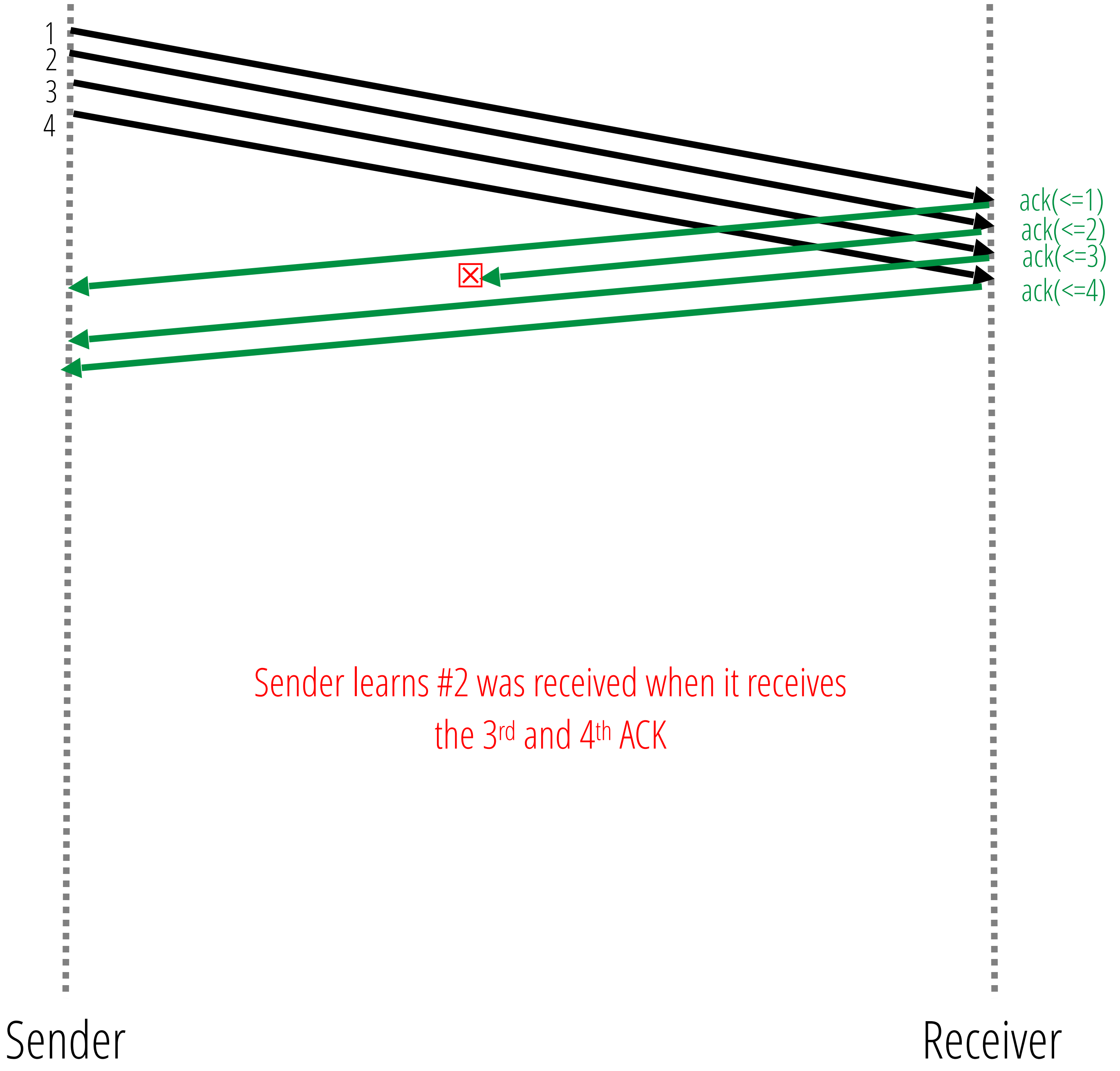


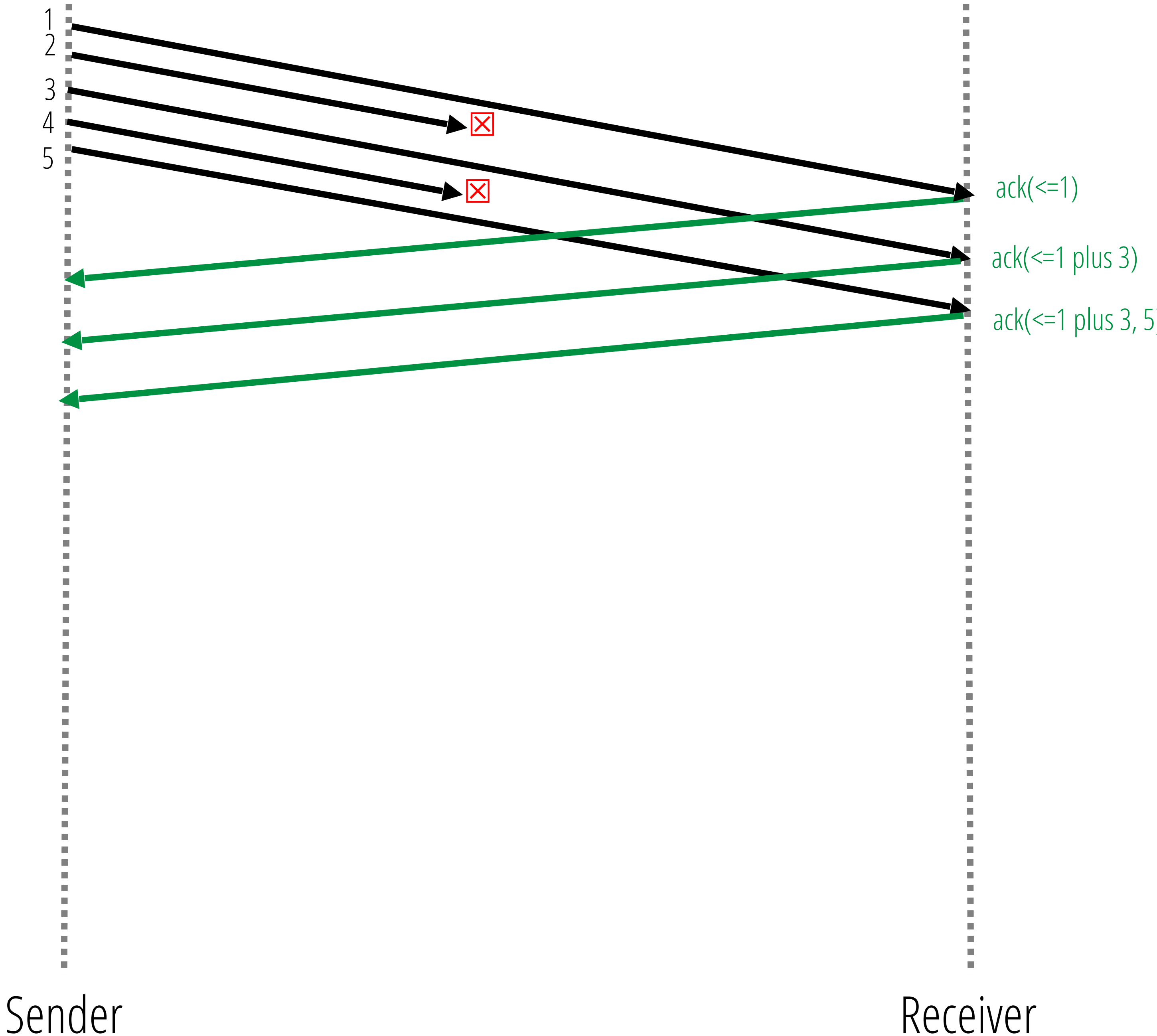
1
2
3
4
ack(1)
ack(2)
ack(3)
ack(4)
2
Unnecessary and avoidable
retransmission!
Sender
Receiver

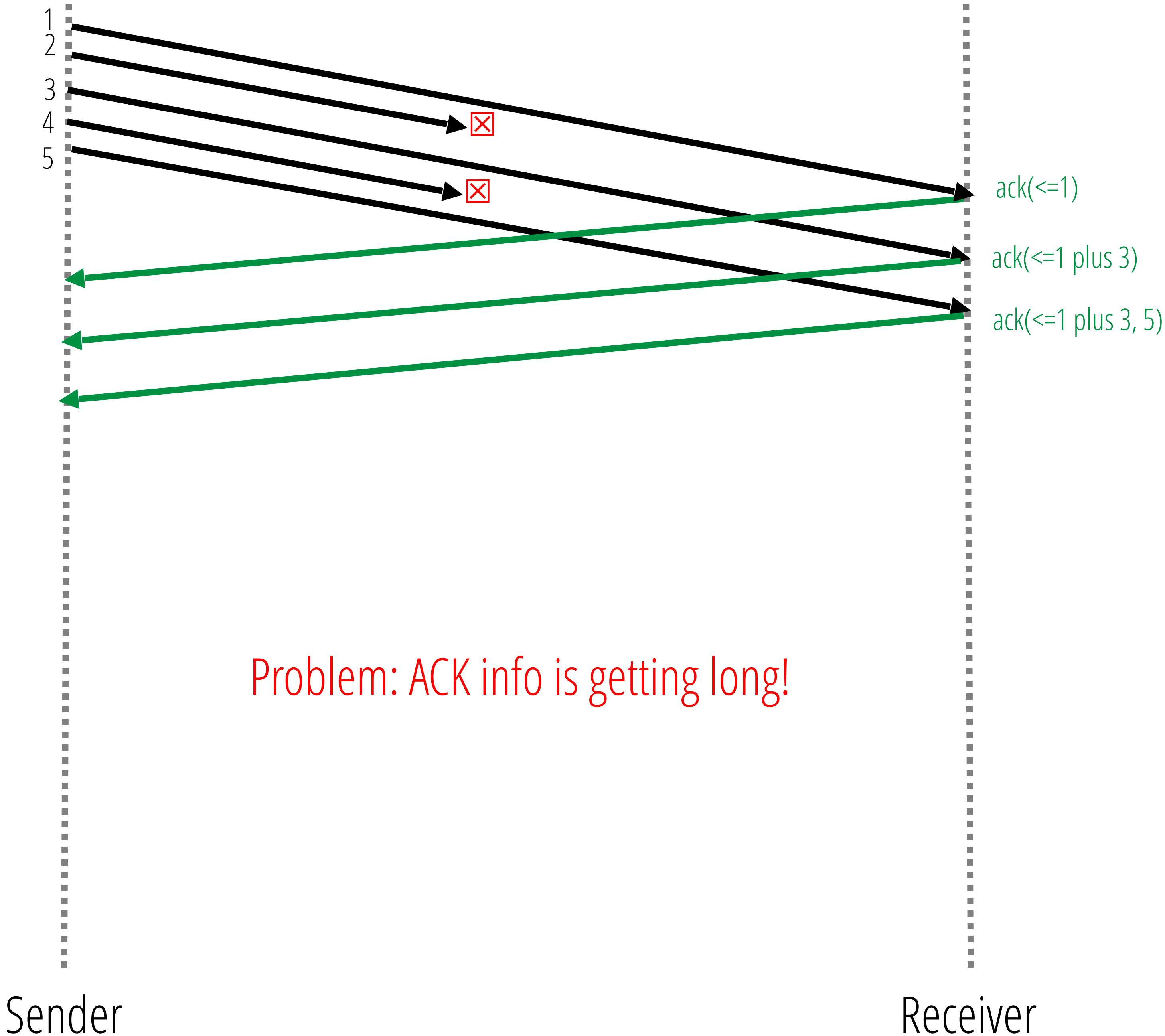
ACKs: design options

- **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*”)


1
2
3
4
ack(<=1)
ack(<=2)
ack(<=3)
ack(<=4)
Sender
Receiver


1
2
3
4
ack(<=1)
ack(<=2)
ack(<=3)
ack(<=4)
Sender learns #2 was received when it receives the 3rd and 4th ACK
Sender
Receiver


1
2
3
4
5
ack(<=1)
ack(<=1 plus 3)
ack(<=1 plus 3, 5)
Sender
Receiver



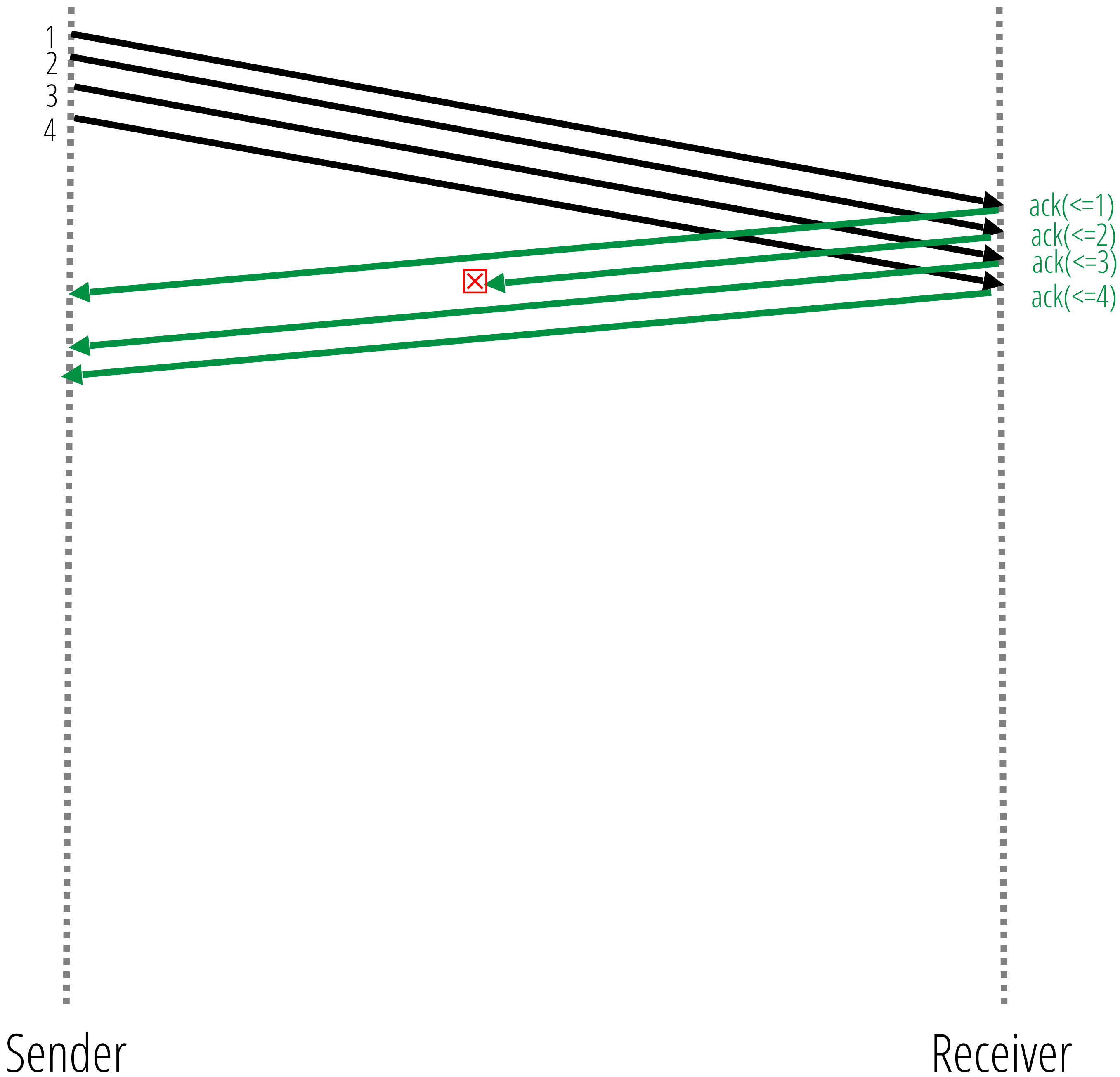
Problem: ACK info is getting long!

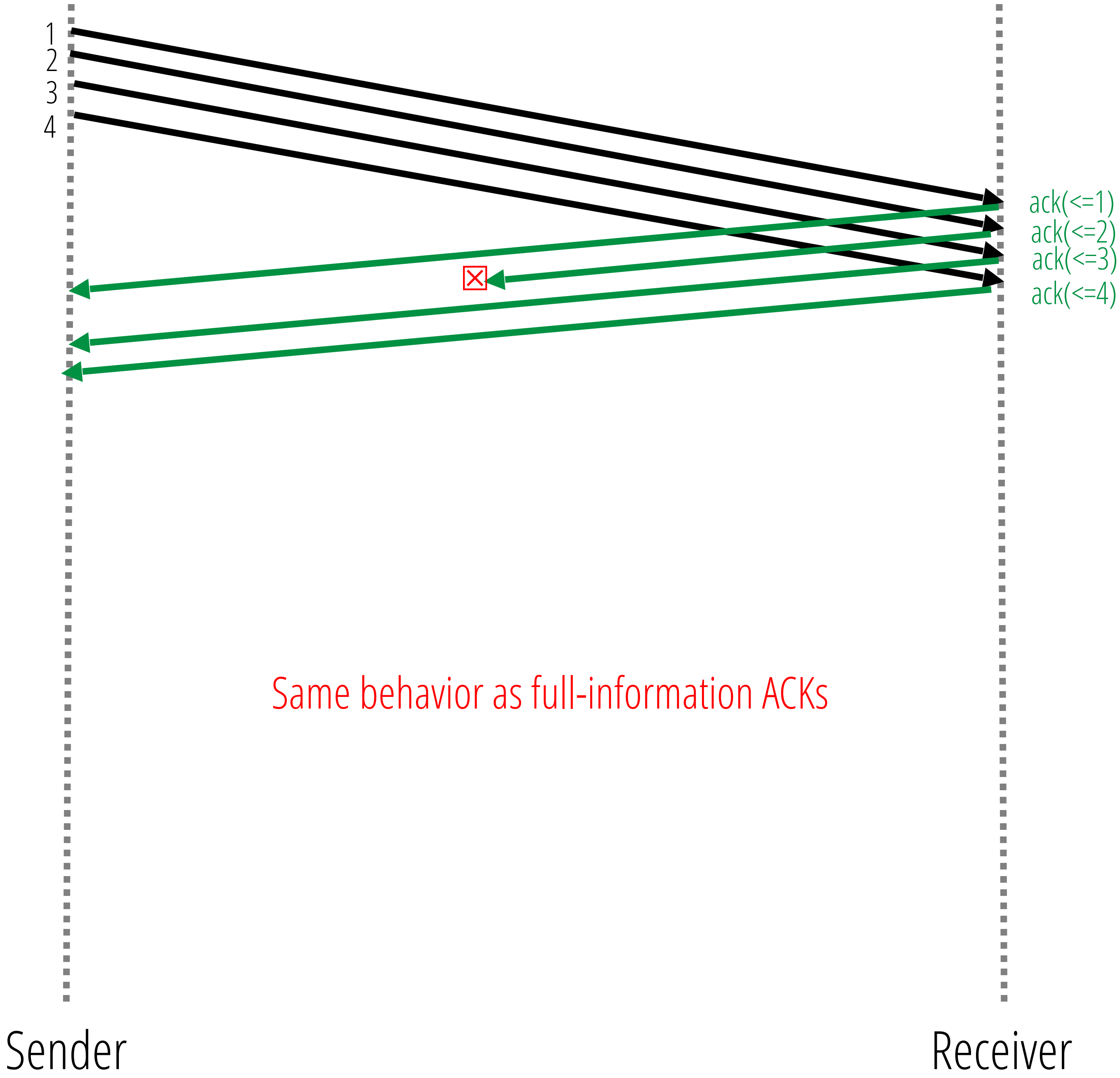
ACKs: design options

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

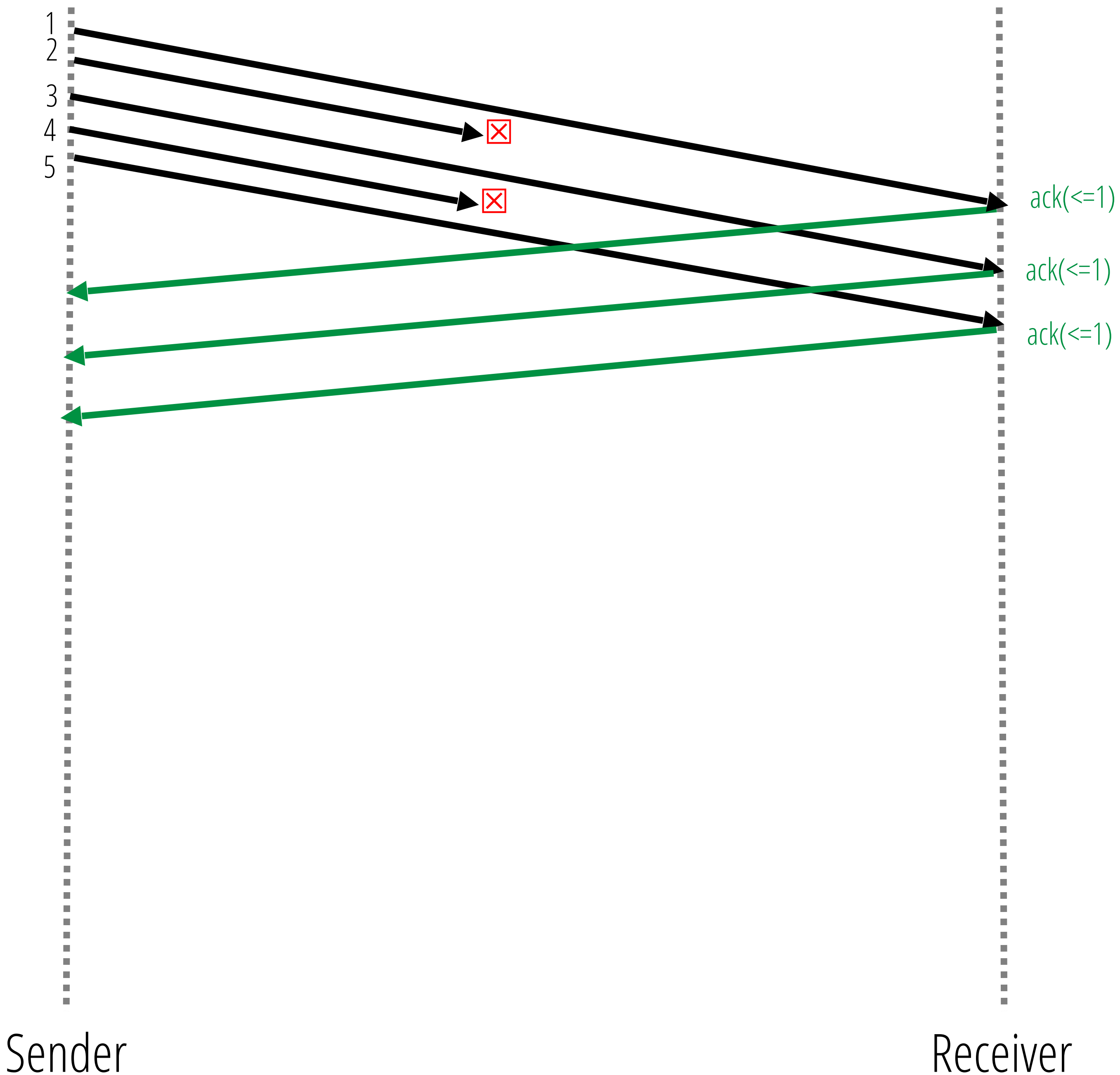
- **Full Information ACKs**
  - Give highest cumulative ACK plus any additional packets received ("*everything up to #12 and #14, #15*")

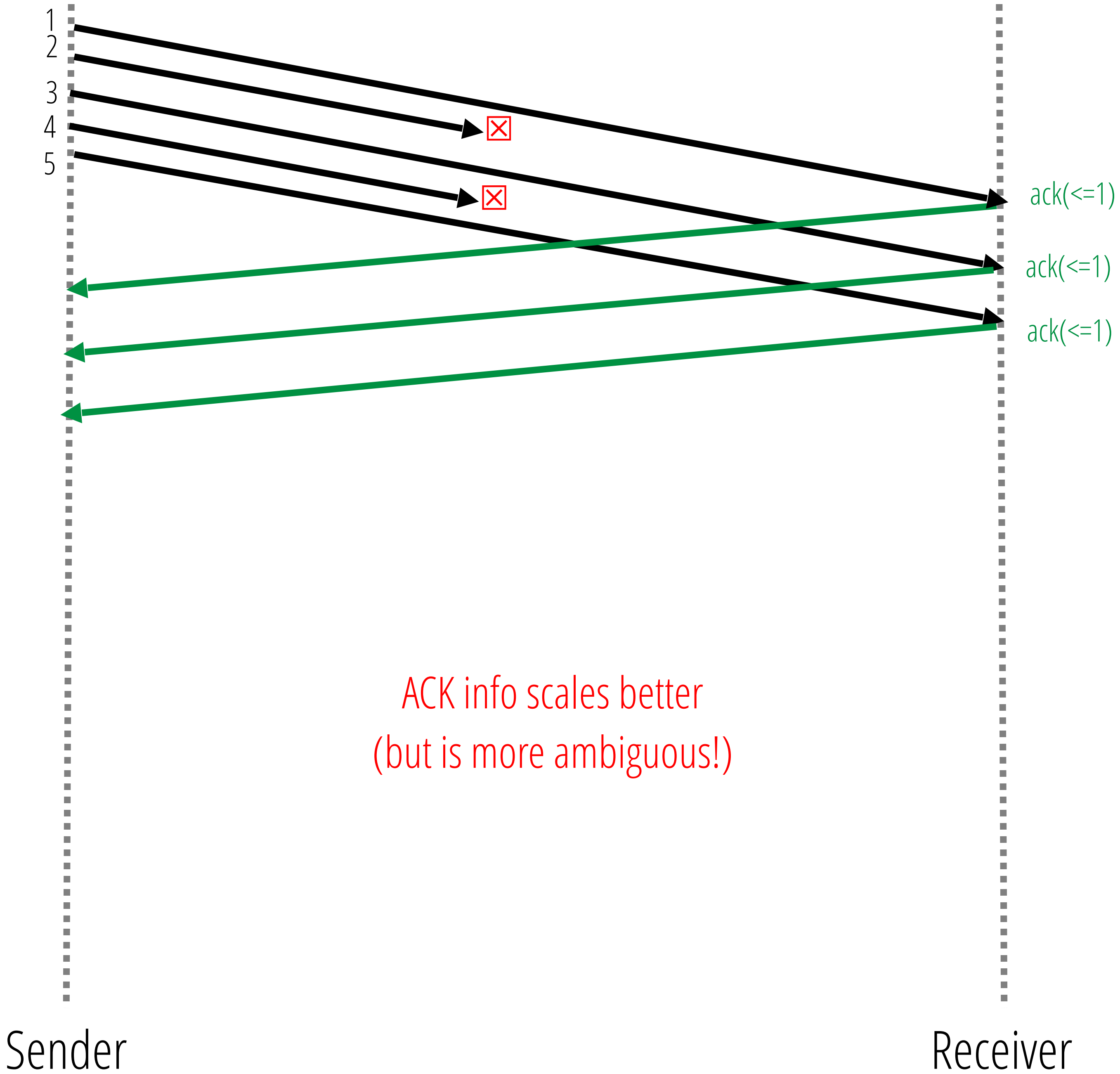
- **Cumulative ACKs**
  - ACK the highest sequence number for which all previous packets have been received


1
2
3
4
ack(<=1)
ack(<=2)
ack(<=3)
ack(<=4)
Sender
Receiver



Same behavior as full-information ACKs


1
2
3
4
5
ack(<=1)
ack(<=1)
ack(<=1)
Sender
Receiver



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)

Duplicate ACKs
(dupACKs)

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)

Duplicate ACKs (dupACKs)

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”

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”

1 2 3 4 5 6 7 8

- 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 → send 9
- ACK 6 arrives → 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”

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

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”

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

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

#duplicate ACKs = 1

- (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 9

#duplicate ACKs = 1

- (for packet 4) ACK 2 → send 9

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 → send 9
- (for packet 6) ACK 2 → 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

#duplicate ACKs = 3

- (for packet 4) ACK 2 → send 9
- (for packet 6) ACK 2 → send 10
- (for packet 7) 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 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...

#duplicate ACKs = 4+

- (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!

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?

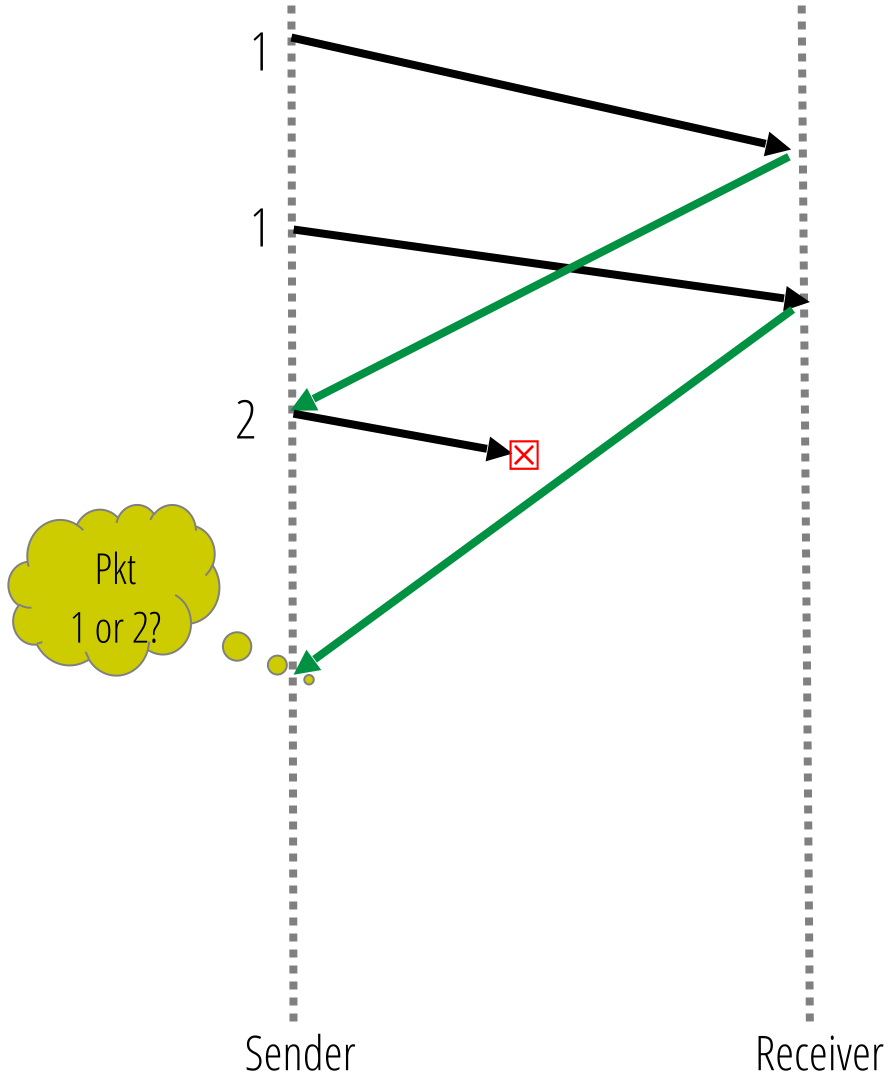
- Sender **encodes** the data to be resilient to loss
  - Basic idea: add some redundancy to data / packet stream
  - E.g., take k packets, encode as n (>k) packets
  - Original packets can be recovered if any k' of n packets are received (n > k' > k)
  - Efficiency depends on k'/k

- 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