The Transport Layer

CPSC 433/533, Spring 2021

Anurag Khandelwal

Administrivia

- **HW1 grades posted, HW2 out!**
  - Any issues with HW1 grading (e.g., regrade request), email Ramla, cc-me
- Hopefully Project#1 is going well — **due next Tue!**
  - **No class next Tue though (break day)…**
- **Periodic check-in:**
  - Are you keeping up with the course fine? If not, reach out! Email, come to OH…

Taking Stock: Network Layer

Taking Stock: Network Layer

- ***Best-effort global delivery of packets***

Taking Stock: Network Layer

- ***Best-effort global delivery of packets***
- **Control Plane:** Routing

Taking Stock: Network Layer

- ***Best-effort global delivery of packets***
- **Control Plane:** Routing
- **Data Plane:** Forwarding

Taking Stock: Network Layer

- ***Best-effort global delivery of packets***
- **Control Plane:** Routing
- **Data Plane:** Forwarding
- **Key enabler of scalability:** Addressing

Addressing: Hierarchical for Scalability

Addressing: Hierarchical for Scalability

- **Hierarchical address structure**

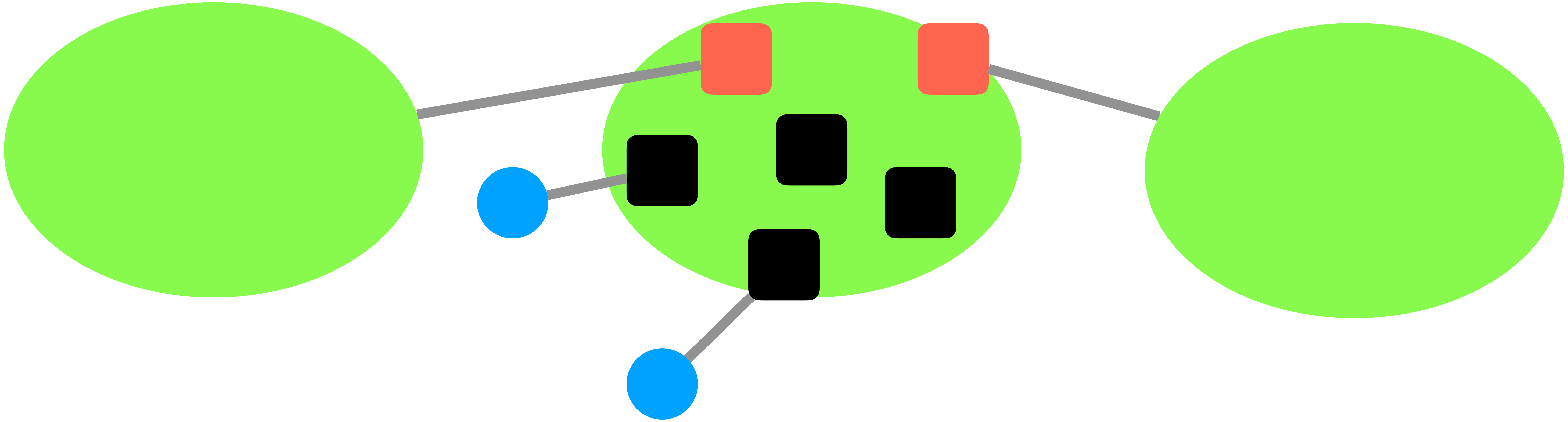
Addressing: Hierarchical for Scalability

- **Hierarchical address structure**
- **Hierarchical address allocation**

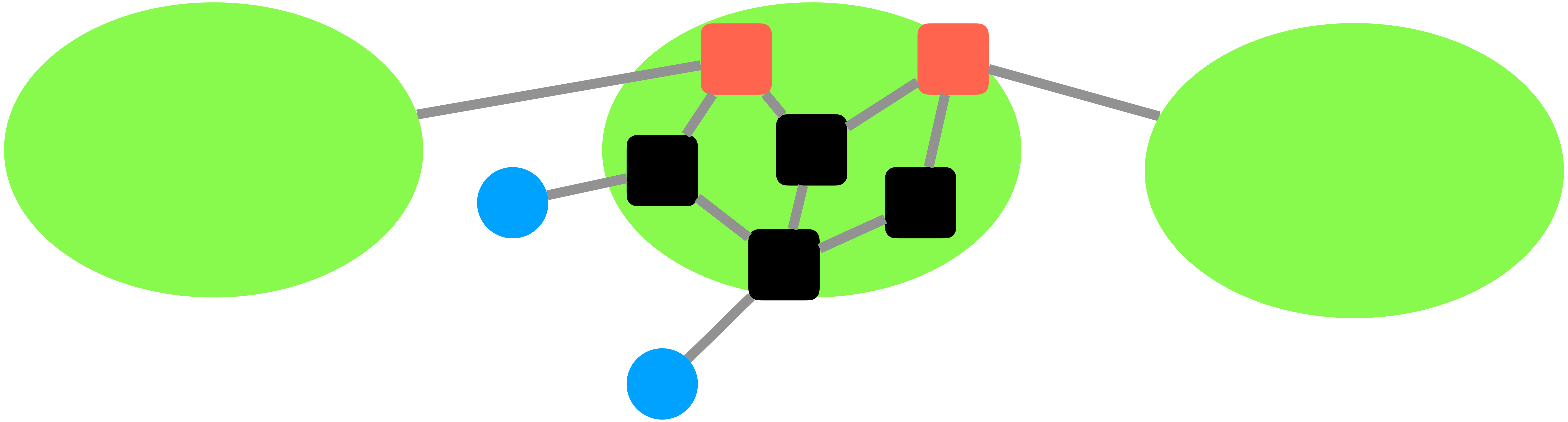
Addressing: Hierarchical for Scalability

- **Hierarchical address structure**
- **Hierarchical address allocation**
- **Hierarchical addresses and routing scalability**

Control Plane: **Routing**

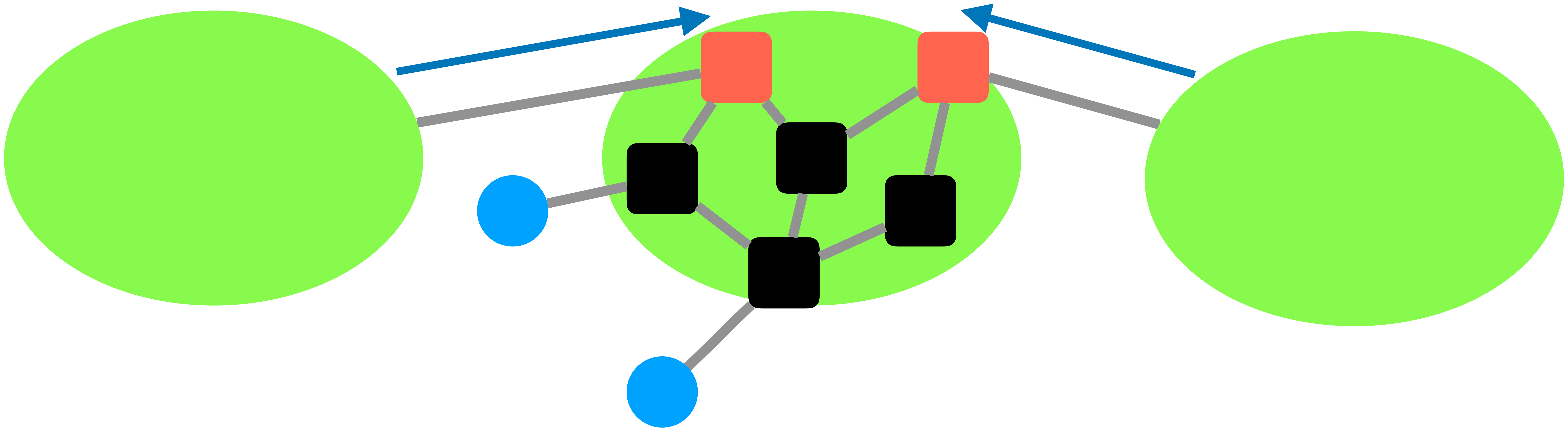


Control Plane: **Routing**



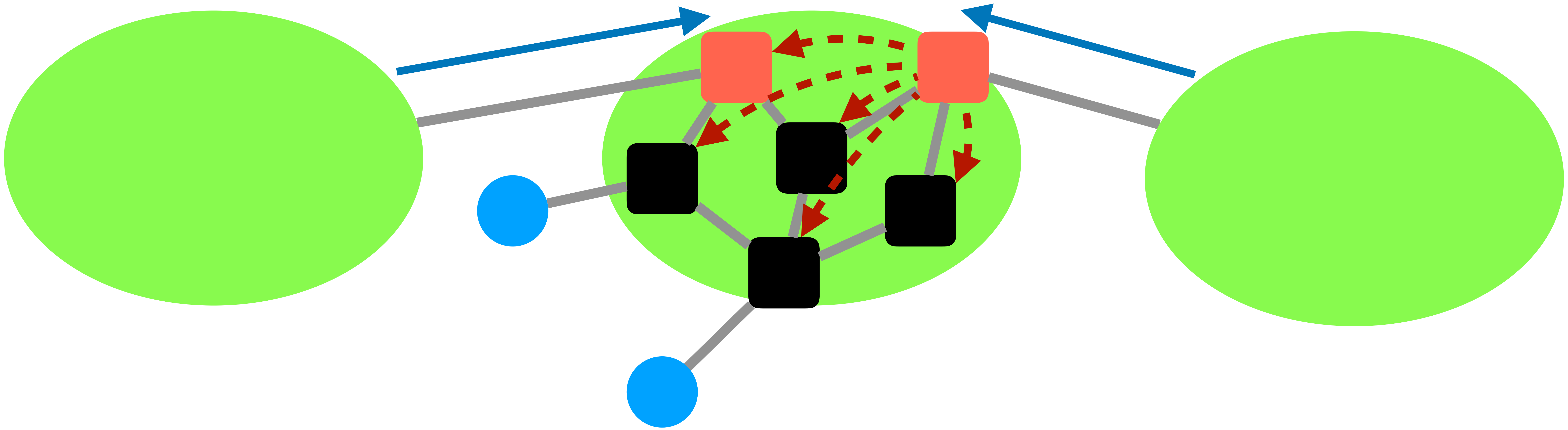
1. Provide internal reachability (**IGP**) 

Control Plane: **Routing**



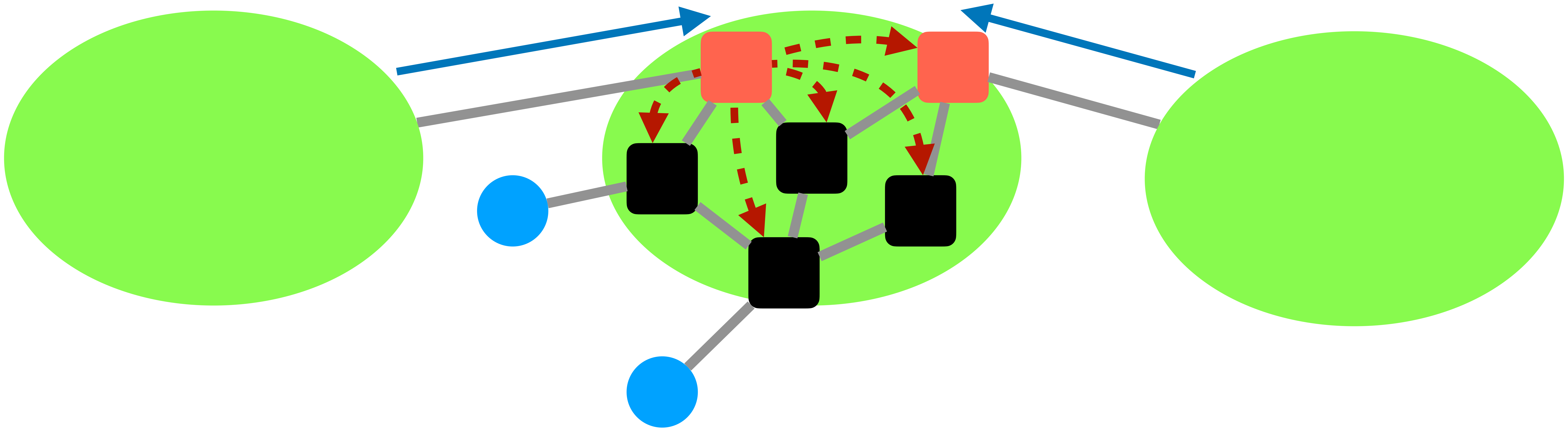
1. Provide internal reachability (**IGP**)
2. Learn routes to external destinations (**eBGP**)

Control Plane: **Routing**



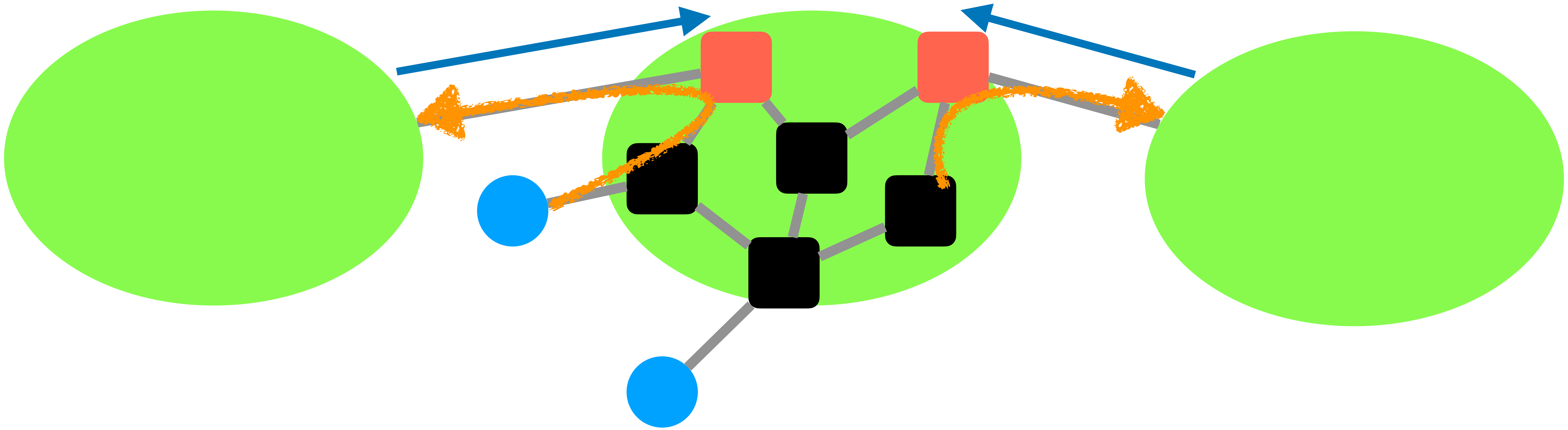
1. Provide internal reachability (**IGP**)
2. Learn routes to external destinations (**eBGP**)
3. Distribute externally learned routes internally (**iBGP**)

Control Plane: **Routing**



1. Provide internal reachability (**IGP**)
2. Learn routes to external destinations (**eBGP**)
3. Distribute externally learned routes internally (**iBGP**)

Control Plane: **Routing**

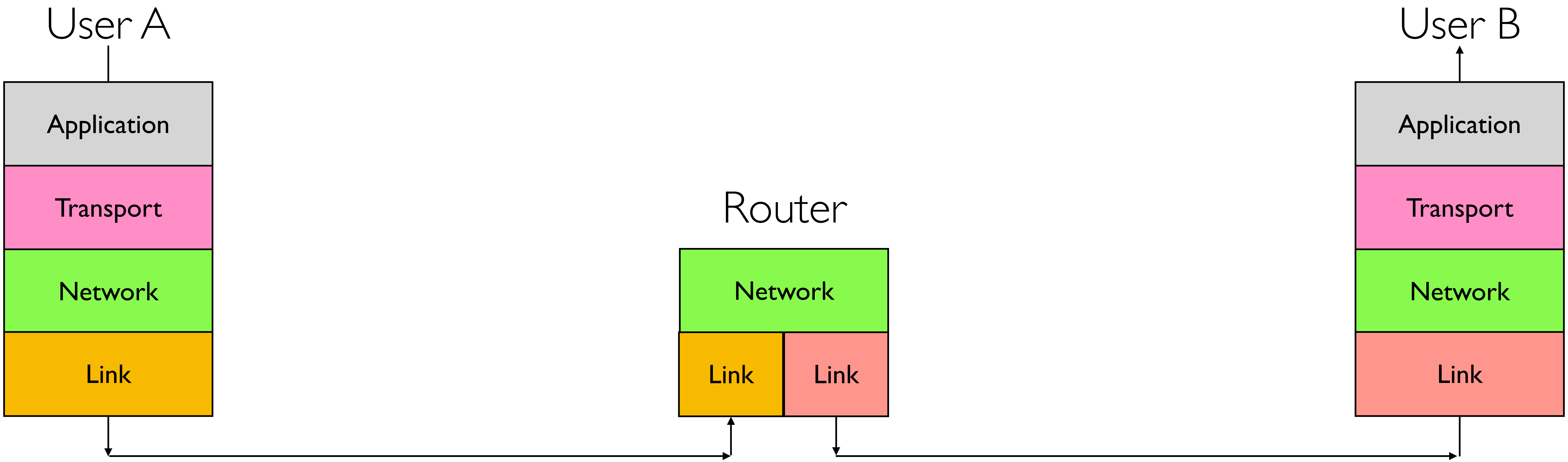


1. Provide internal reachability (**IGP**)
2. Learn routes to external destinations (**eBGP**)
3. Distribute externally learned routes internally (**iBGP**)
4. Travel shortest path to egress (**IGP**)

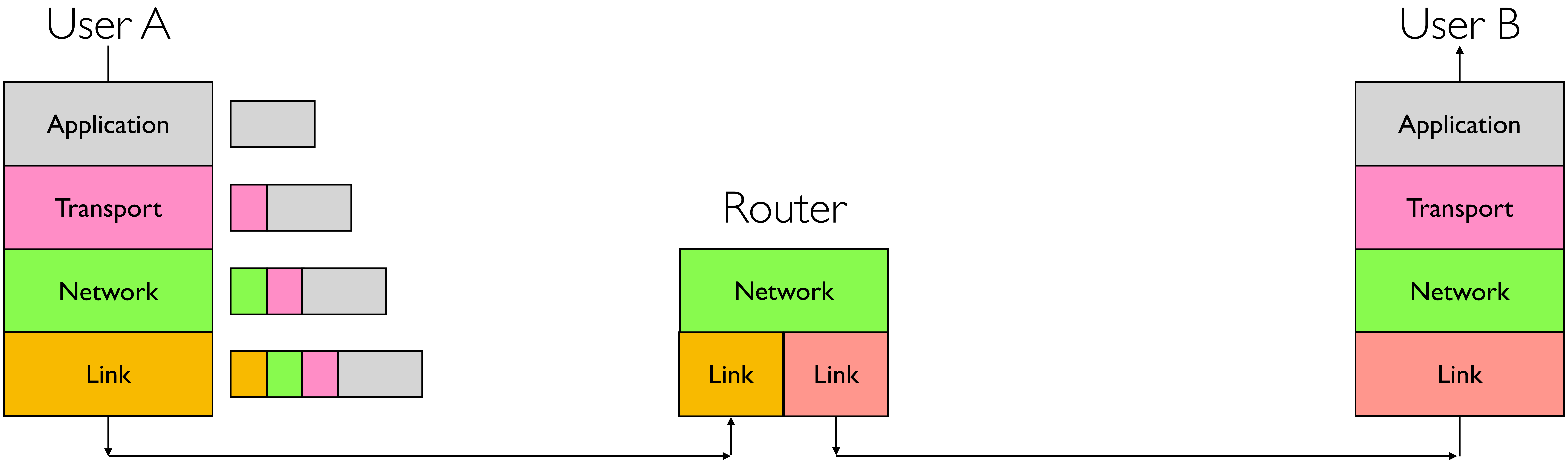
Data Plane: **Forwarding**

Data Plane: **Forwarding**

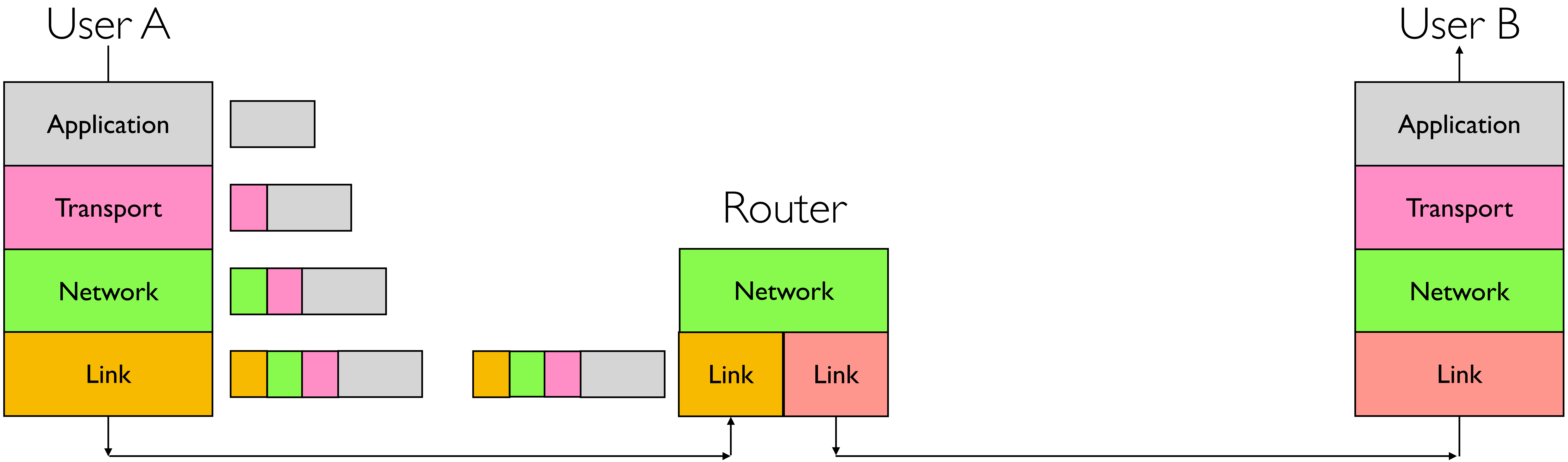
Data Plane: **Forwarding**



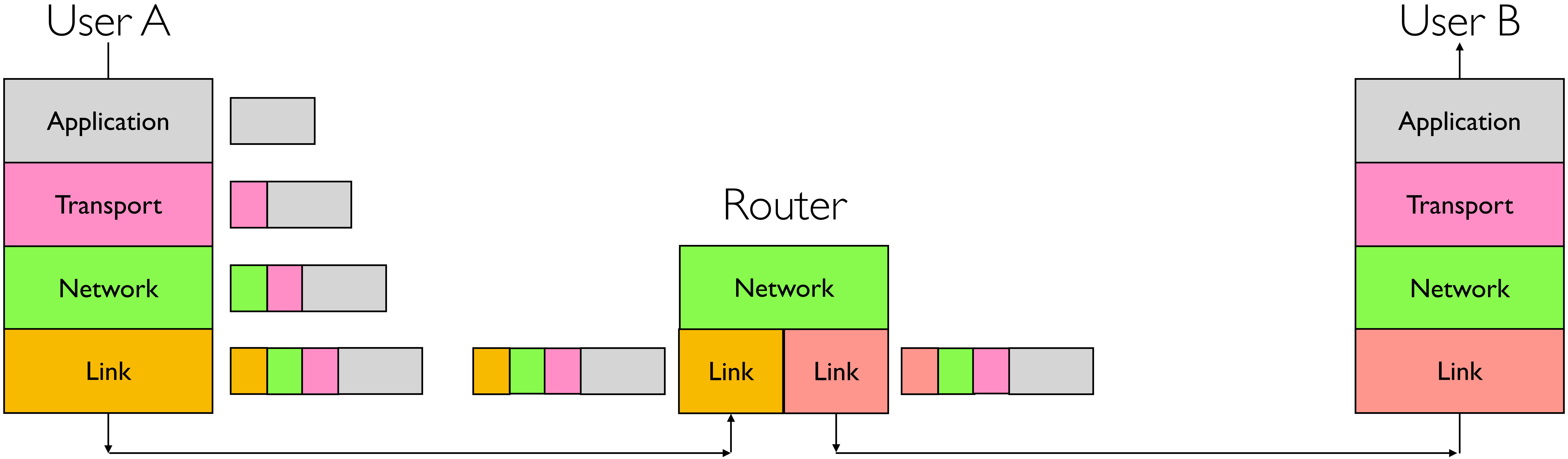
Data Plane: **Forwarding**



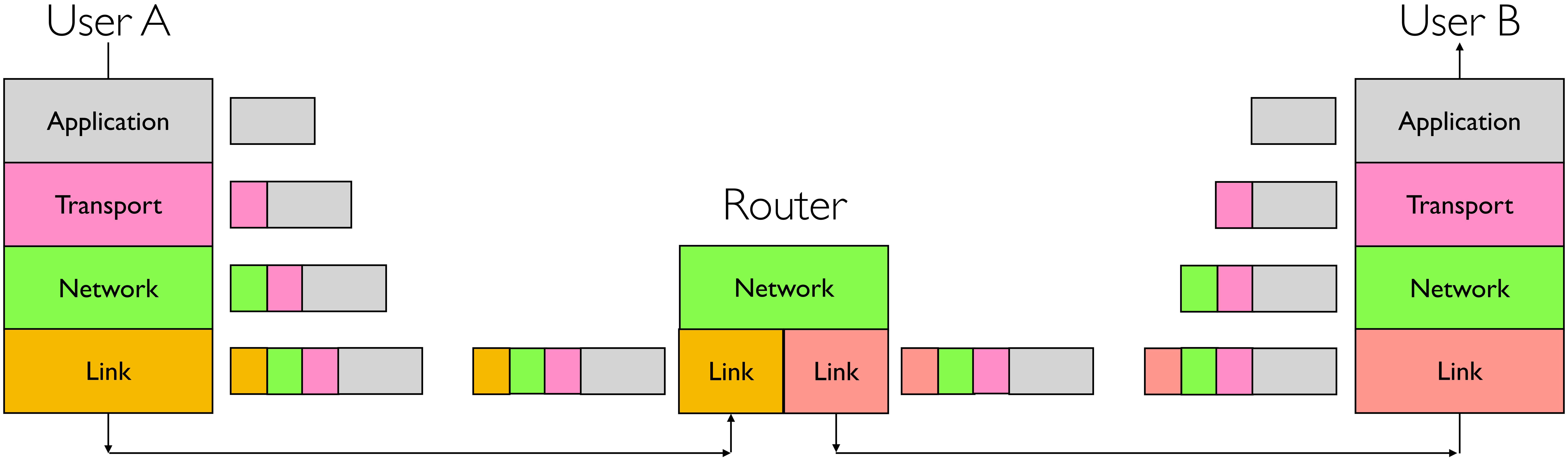
Data Plane: **Forwarding**



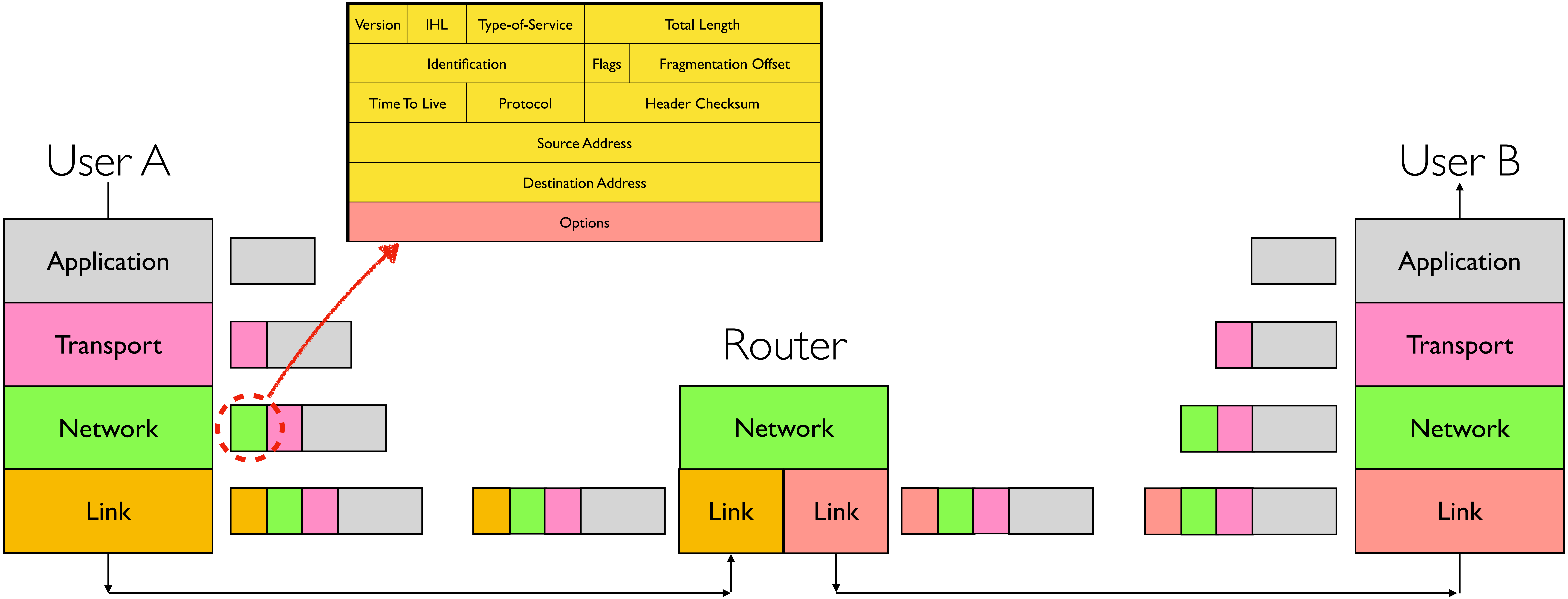
Data Plane: **Forwarding**



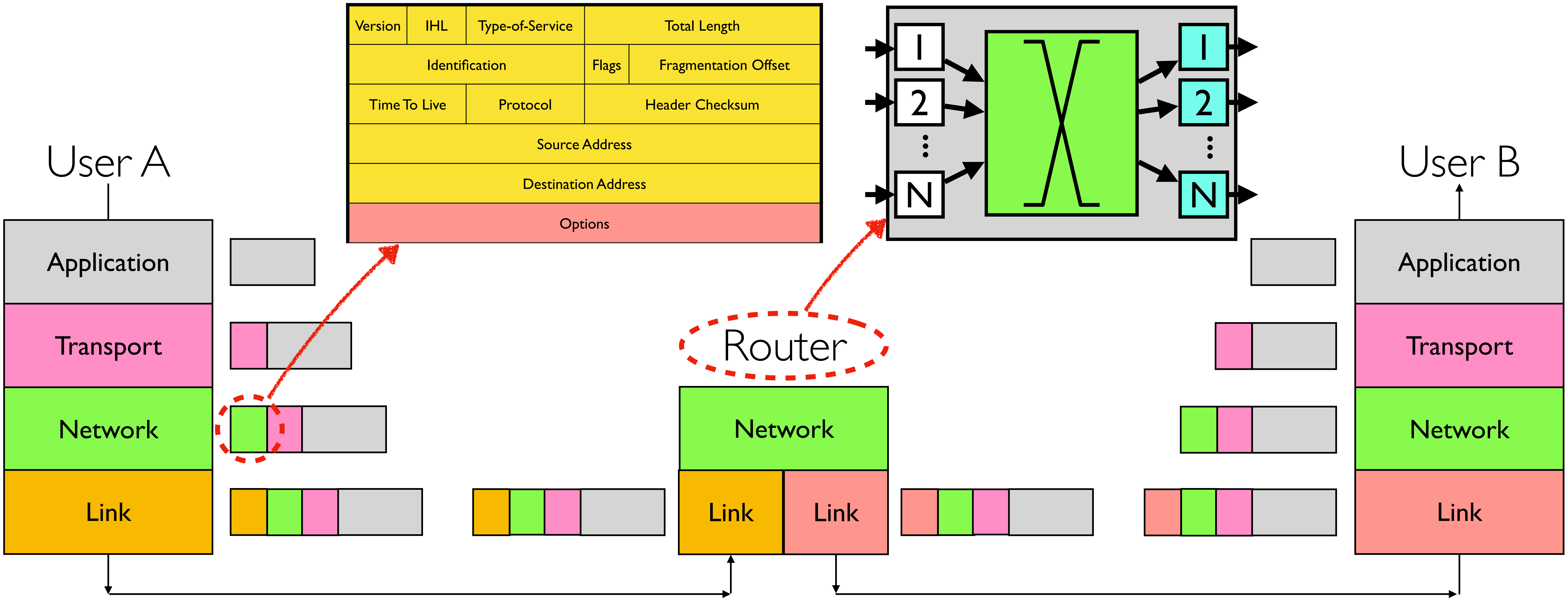
Data Plane: **Forwarding**



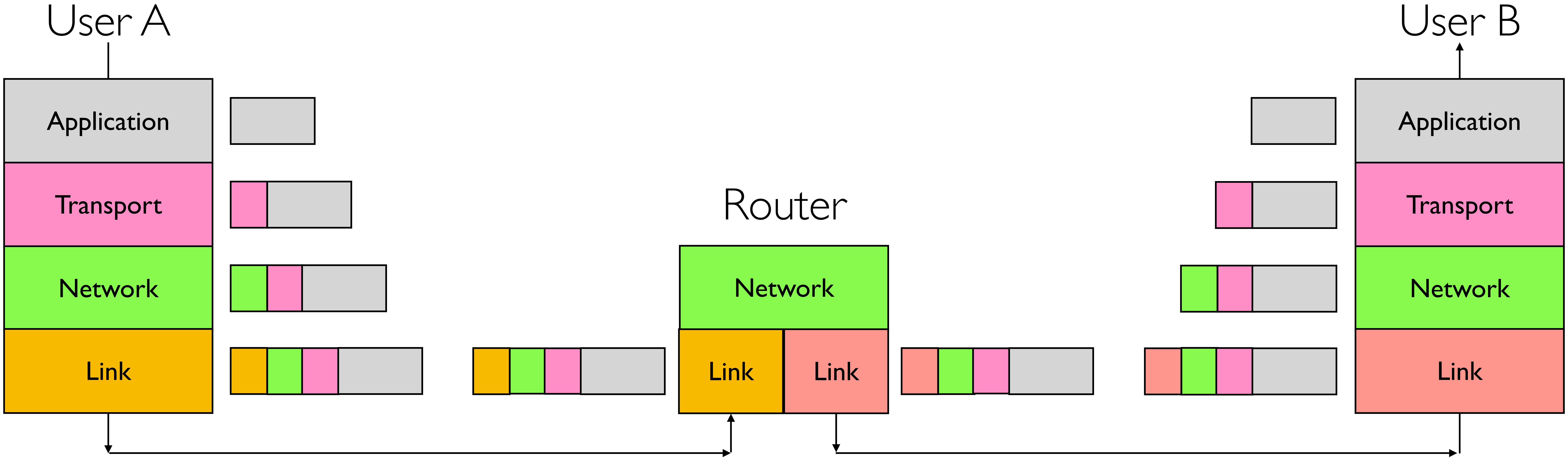
Data Plane: **Forwarding**



Data Plane: **Forwarding**



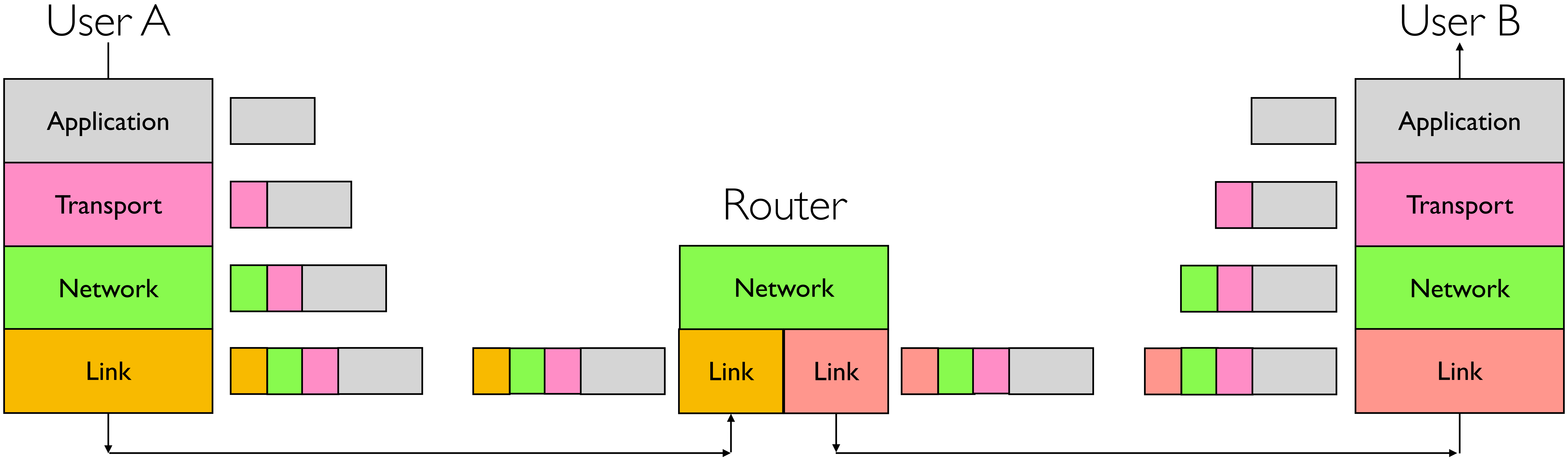
Up Next…



User A
User B
Application
Transport
Network
Link
Router
Network
Link
Link
Application
Transport
Network
Link

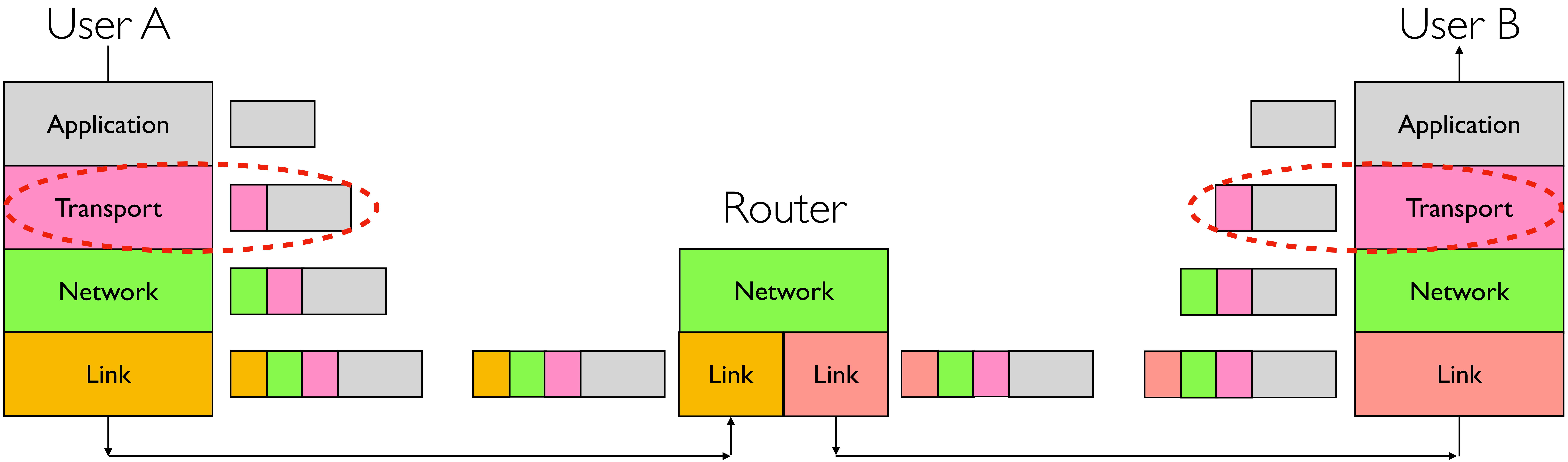
Up Next…

- **So far: Network Layer,** *Best-effort global delivery of packets*



Up Next…

- **So far: Network Layer,** *Best-effort global delivery of packets*
- **Next: Transport Layer,** *Reliable (or unreliable) delivery of data*
  - *Layer at* ***end-hosts,*** *between the application and network layers*



Why a Transport Layer?

Why a Transport Layer?

- **Transport layer and application both on host**

Why a Transport Layer?

- **Transport layer and application both on host**
- **Why not combine the two?**

Why a Transport Layer?

- **Transport layer and application both on host**
- **Why not combine the two?**
- **And what should that code do anyway?**

Why a Transport Layer?

Why a Transport Layer?

1. **De-multiplex packets between many applications**

Why a Transport Layer?

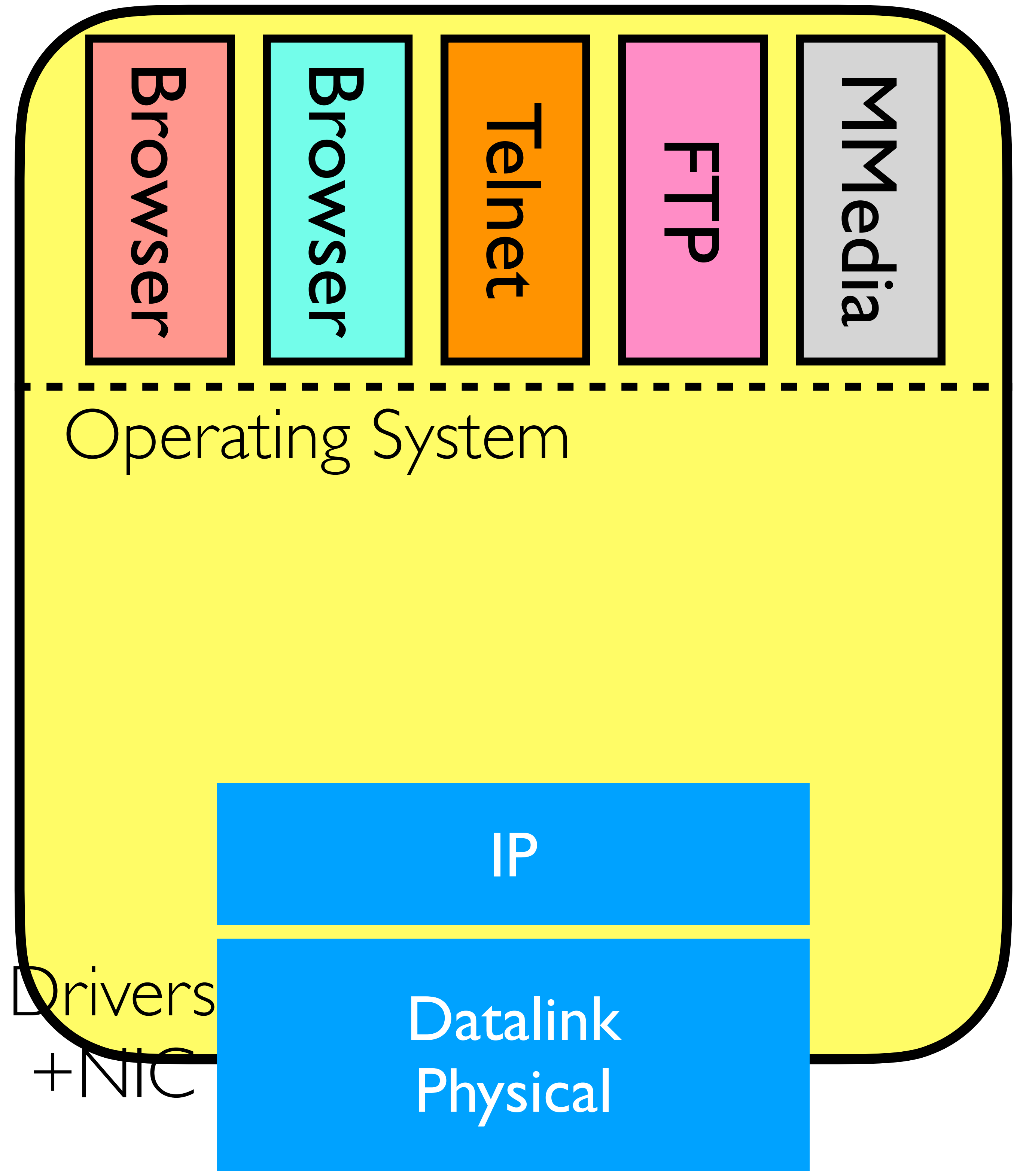
1. **De-multiplex packets between many applications**
2. **Additional (optional) services on top of IP**

Why a Transport Layer: Demultiplexing

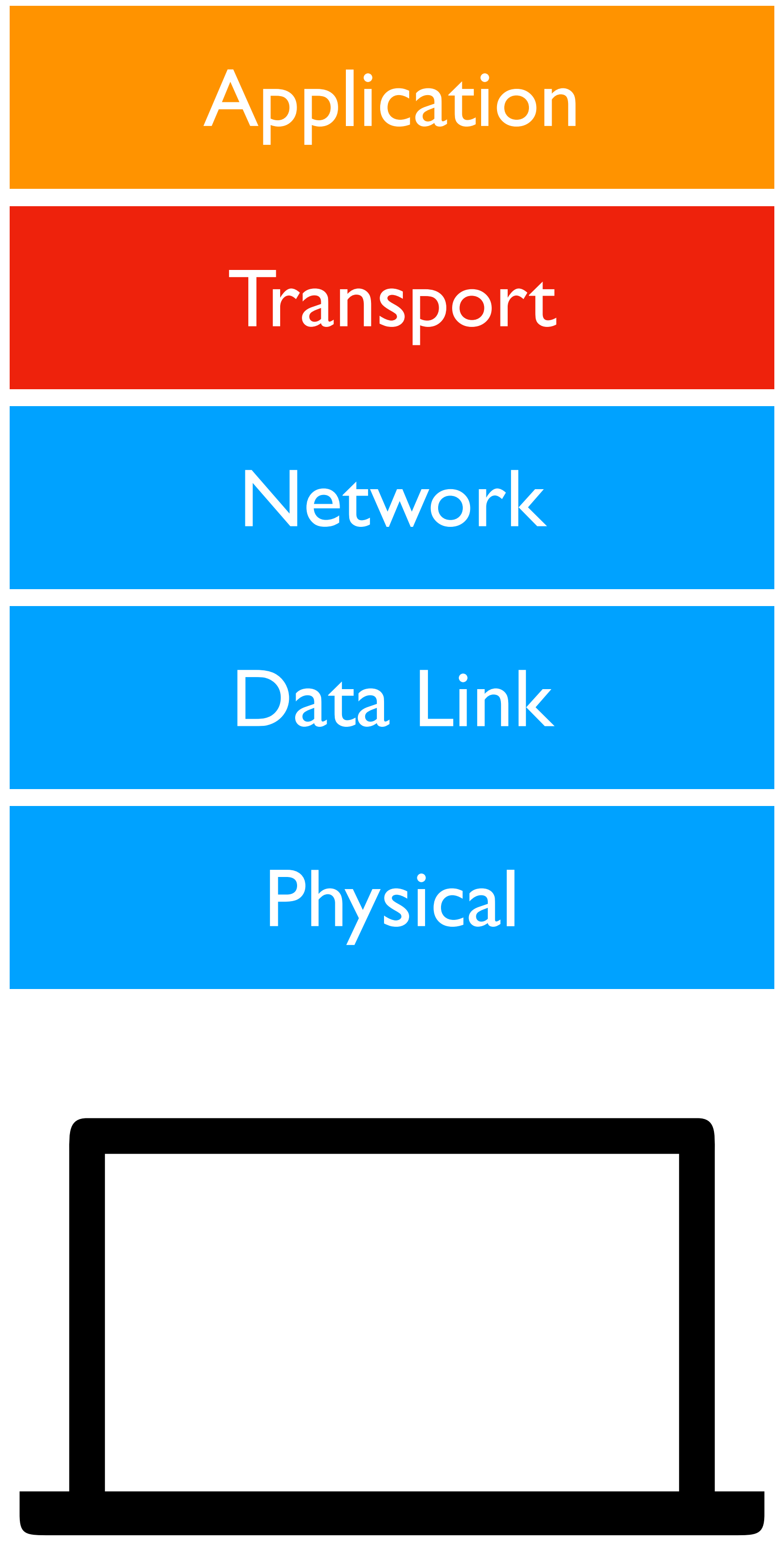
Why a Transport Layer: Demultiplexing

- **IP packets are addressed to a host, but end-to-end communication is between application processes at hosts**
  - Need a way to decide which packets go to which applications (*multiplexing/ demultiplexing*)

Why a Transport Layer: Demultiplexing

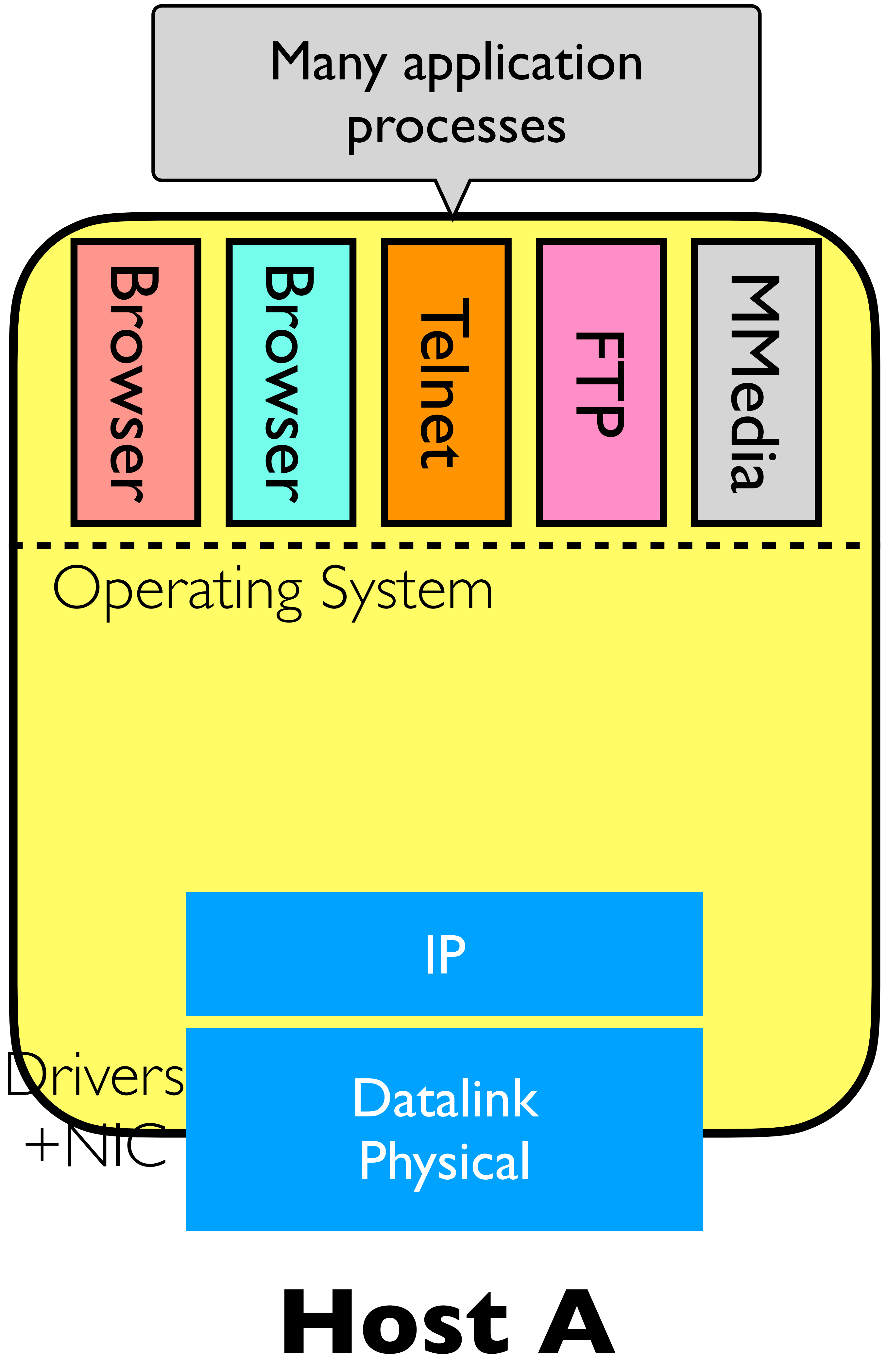


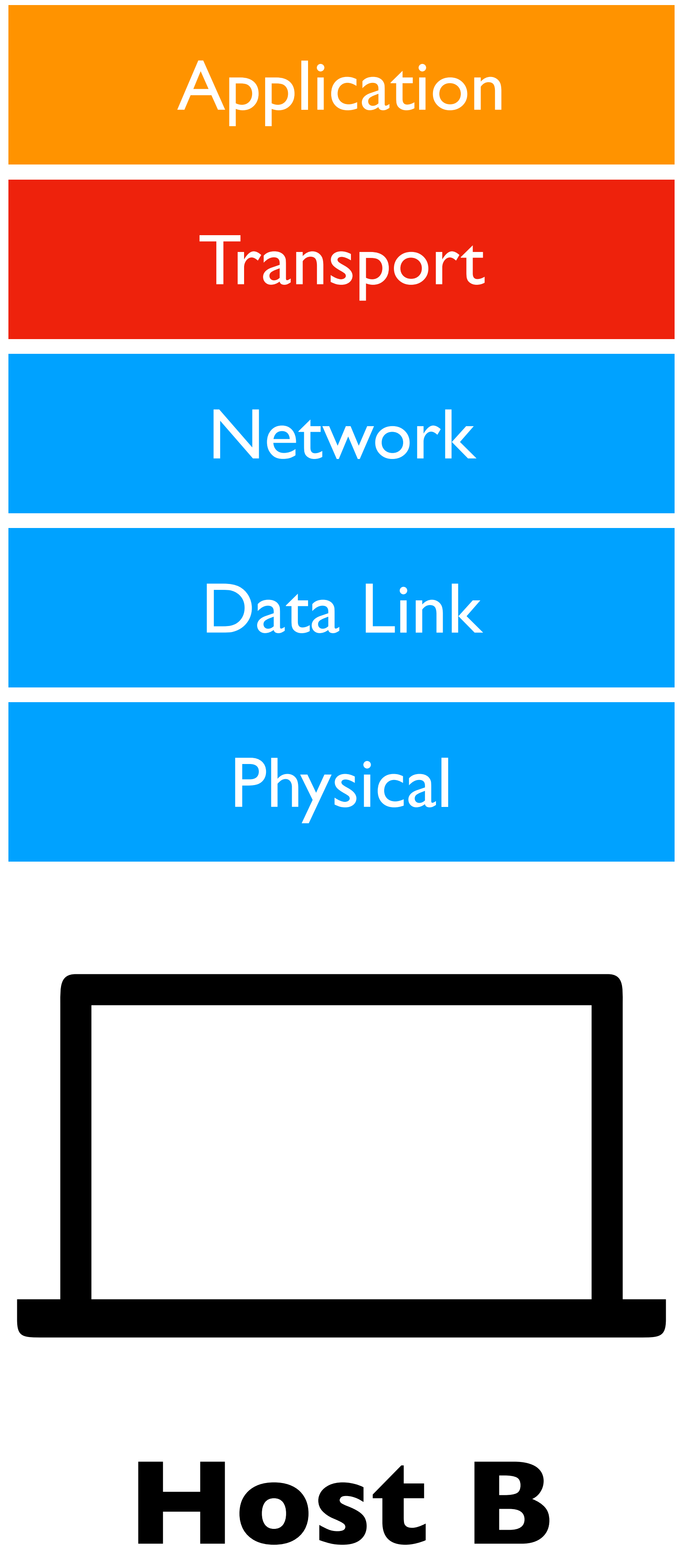
Host A



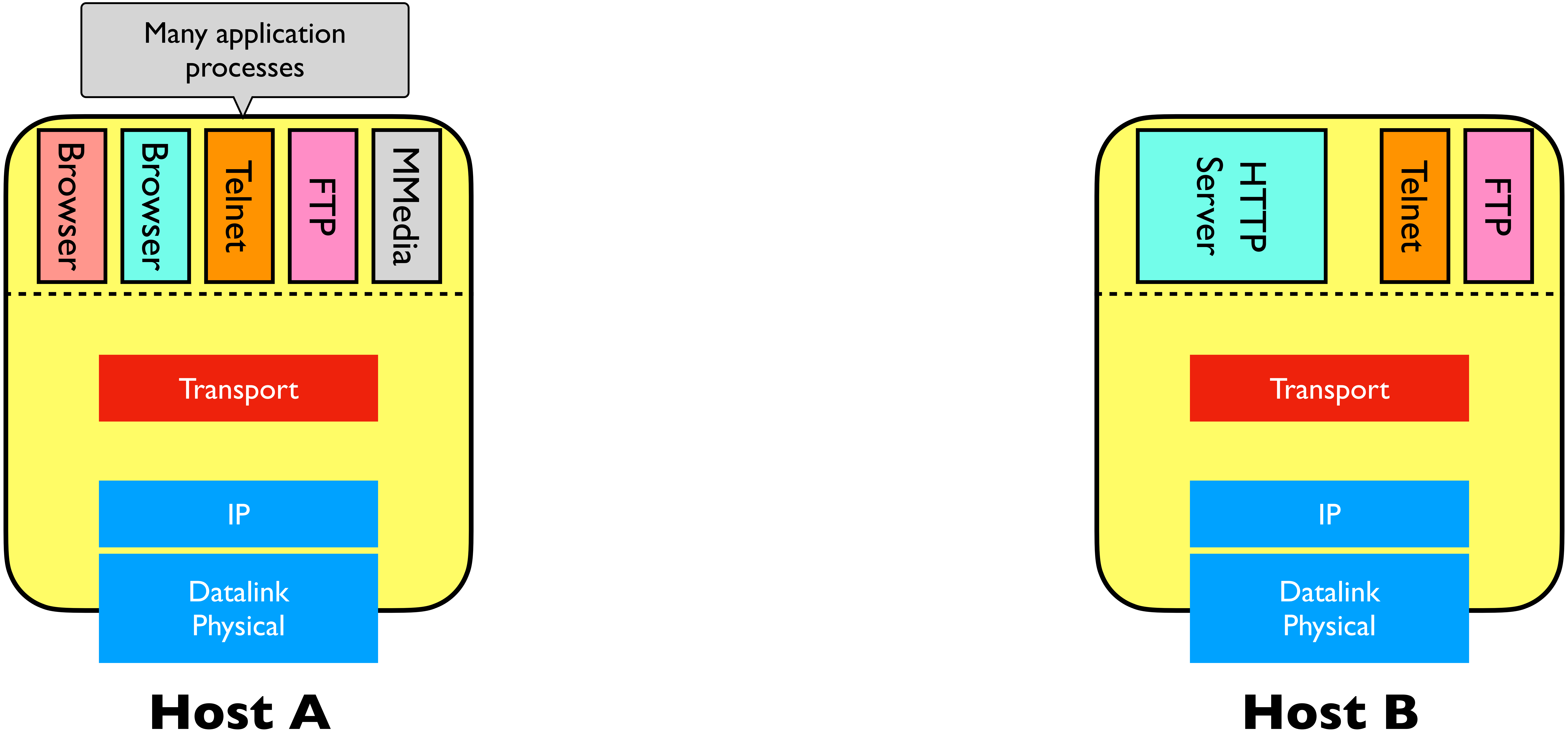
Host B

Why a Transport Layer: Demultiplexing

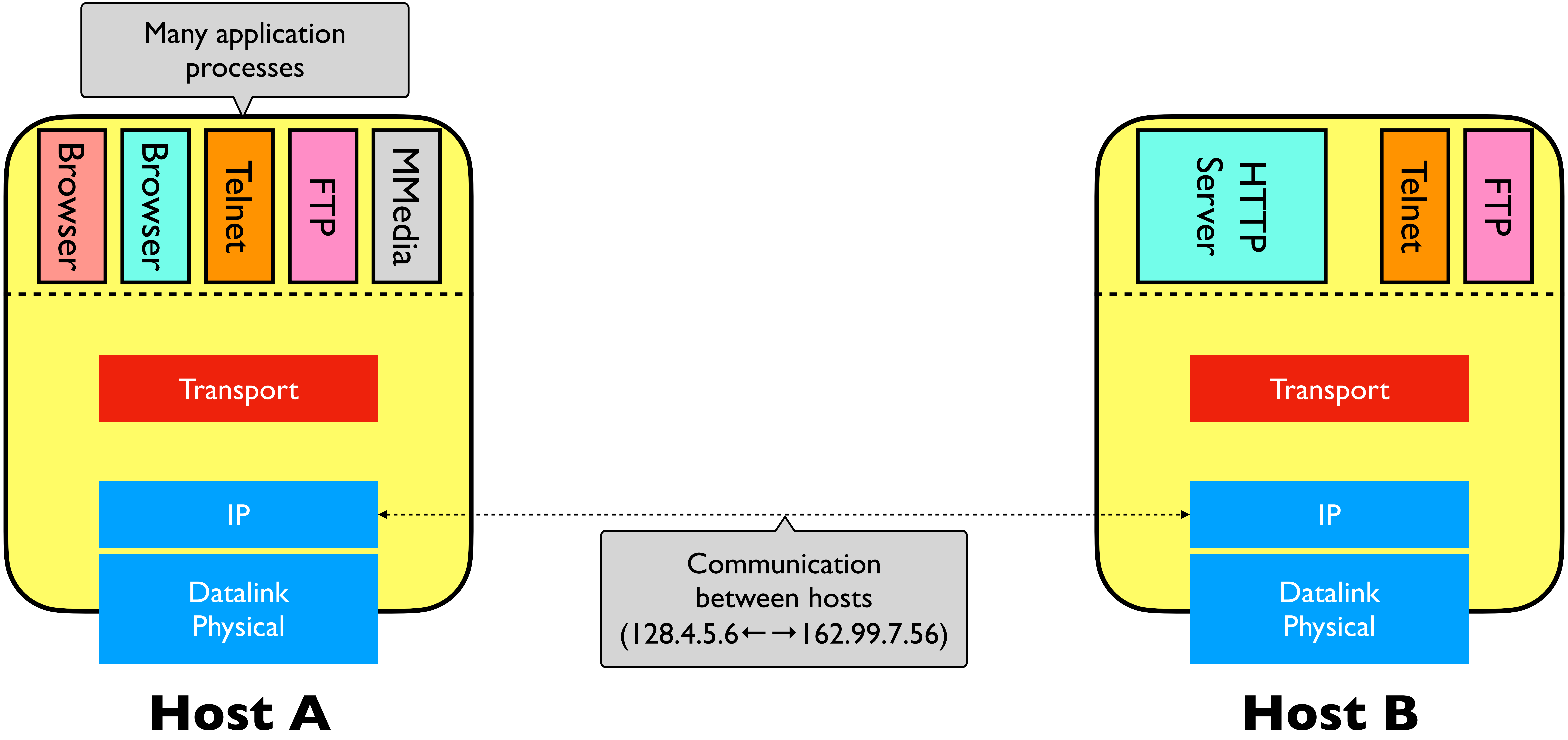




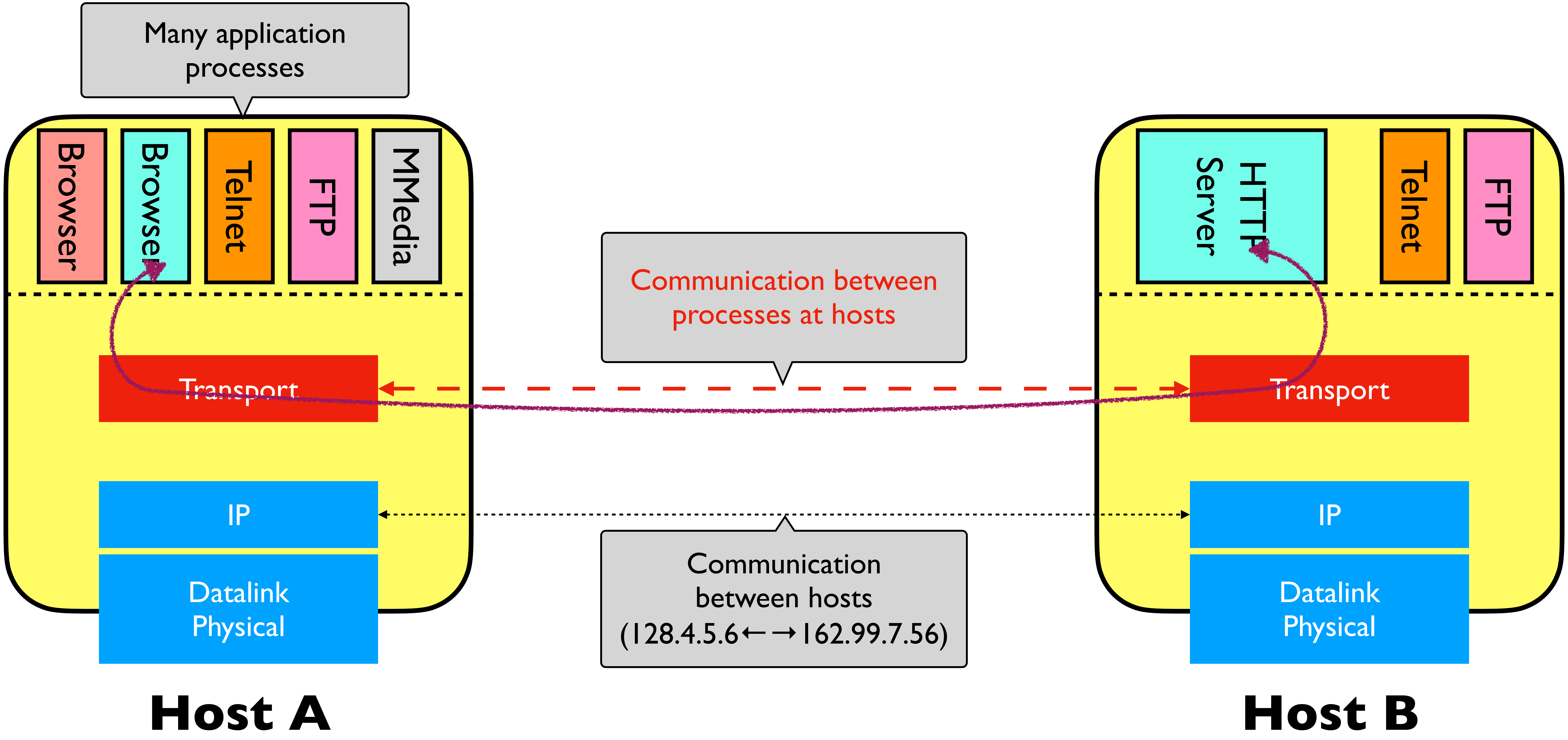
Why a Transport Layer: Demultiplexing



Why a Transport Layer: Demultiplexing



Why a Transport Layer: Demultiplexing



Why a Transport Layer: Improved Service Model

Why a Transport Layer: Improved Service Model

- **IP provides a weak service model (*best-effort*)**
  - Packets can be corrupted, delayed, dropped, reordered, duplicated
  - No guidance on how much traffic to send and when
  - Dealing with this is tedious for application developers

Role of the Transport Layer

Role of the Transport Layer

- **Communication between application processes**
  - Multiplex and demultiplex from/to application processes
  - Implemented using ***ports*** (not the same as router ports!)

Role of the Transport Layer

- **Communication between application processes**

- **Provide common end-to-end services for application layer [optional]**
  - Reliable, in-order data delivery
  - Well-placed data delivery
    - Too fast may overwhelm the network
    - Too slow is not efficient

Role of the Transport Layer

- **Communication between application processes**
- **Provide common end-to-end services for application layer [optional]**
- **TCP and UDP are the common transport protocols**
  - Also SCTP, MTCP, SST, RDP, DCCP, …

Role of the Transport Layer

- **Communication between application processes**
- **Provide common end-to-end services for application layer [optional]**
- **TCP and UDP are the common transport protocols**
- **UDP is a minimalist, no-frills transport protocol**
  - Only provides multiplex/demultiplex capabilities

Role of the Transport Layer

- **Communication between application processes**
- **Provide common end-to-end services for application layer [optional]**
- **TCP and UDP are the common transport protocols**
- **UDP is a minimalist, no-frills transport protocol**
- **TCP is the whole-hog protocol**
  - Offers applications a reliable, in-order, byte stream abstraction
  - With congestion control
  - But no performance guarantees (delay, bw, etc.)

Why a transport layer?

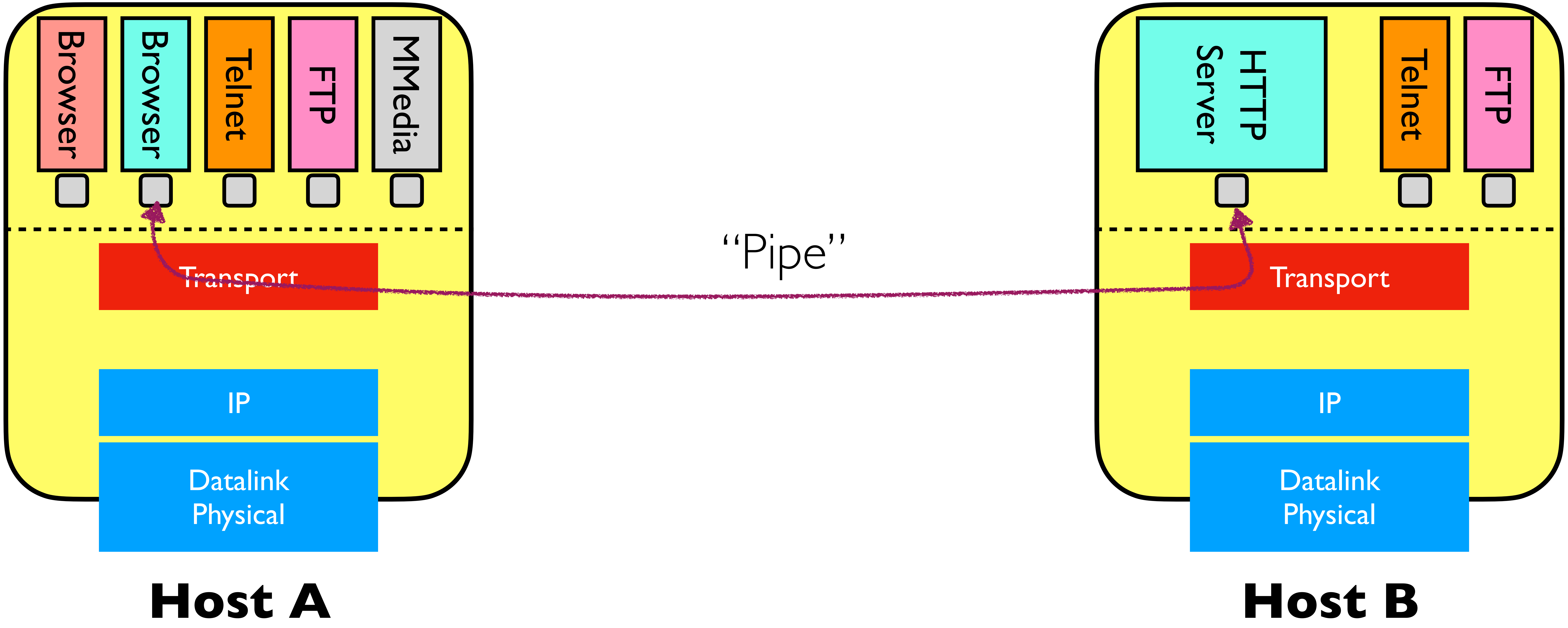
- **IP packets are addressed to a host, but end-to-end communication is between application processes at hosts**
  - Need a way to decide which packets go to which applications (*multiplexing/demultiplexing*)
- **IP provides a weak service model (*best-effort*)**
  - Packets can be corrupted, delayed, dropped, reordered, duplicated

Why a transport layer?

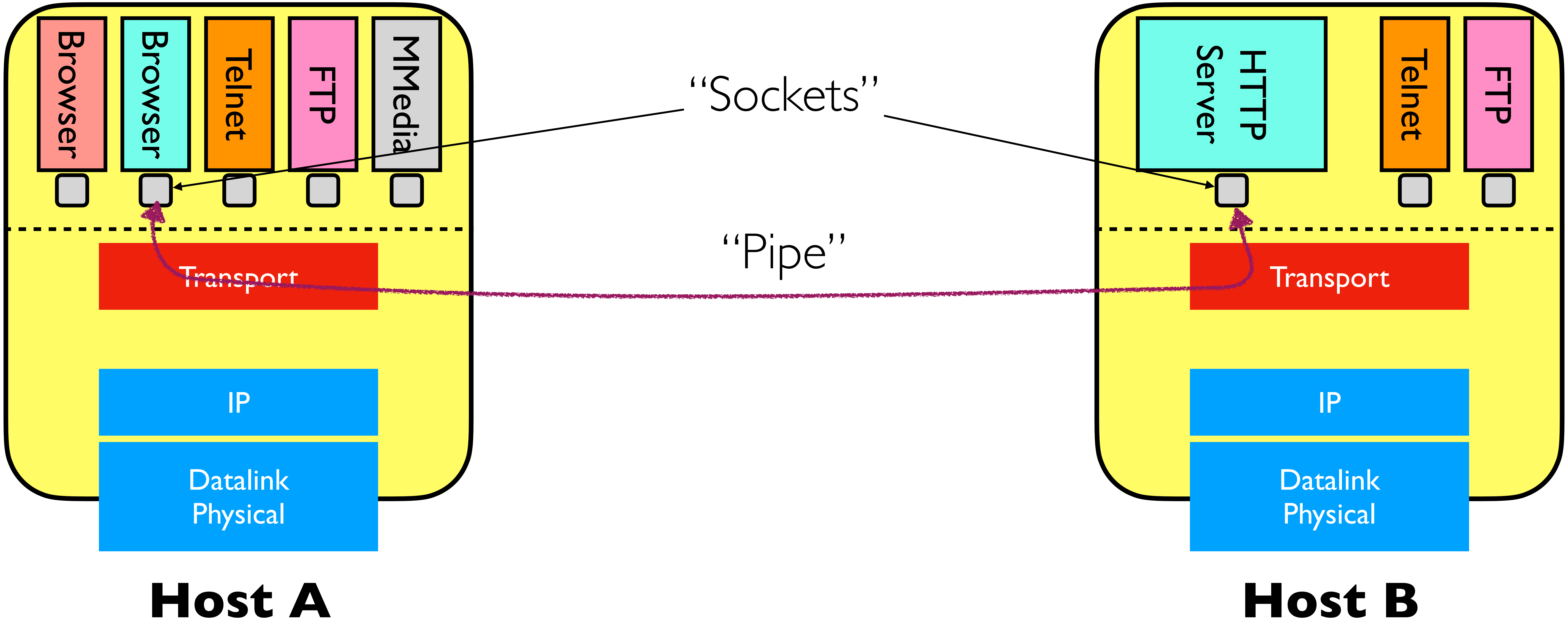
- **IP packets are addressed to a host, but end-to-end communication is between application processes at hosts**
  - Need a way to decide which packets go to which applications (*multiplexing/demultiplexing*)
- **IP provides a weak service model (*best-effort*)**
  - Packets can be corrupted, delayed, dropped, reordered, duplicated

Abstraction: Pipes & Sockets

Abstraction: Pipes & Sockets



Abstraction: Pipes & Sockets



Two “Pipe” Abstractions

Two “Pipe” Abstractions

- **Unreliable packet delivery:** User Datagram Protocol (UDP)
  - Messages are limited to a single packet (“datagrams”)
  - Pipe is “leaky”: applications need to deal with leaks (lost/corrupted datagrams)

Two "Pipe" Abstractions

- **Unreliable packet delivery:** User Datagram Protocol (UDP)
  - Messages are limited to a single packet ("datagrams")
  - Pipe is "leaky": applications need to deal with leaks (lost/corrupted datagrams)
- **Reliable byte stream:** Transmission Control Protocol (TCP)
  - Bytes inserted into pipe by sender
  - They emerge, in order, at the receiver (to the application)

UDP (Datagram-oriented)

UDP (Datagram-oriented)

- Sources send packets, destinations receive packets

UDP (Datagram-oriented)

- Sources send packets, destinations receive packets
- If packets delayed/reordered/lost:
    - *"It's not my problem!"*
    - Let application deal with it

UDP (Datagram-oriented)

- Sources send packets, destinations receive packets
- If packets delayed/reordered/lost:
  - *"It's not my problem!"*
  - Let application deal with it
- Discarding corrupted packets: optional!

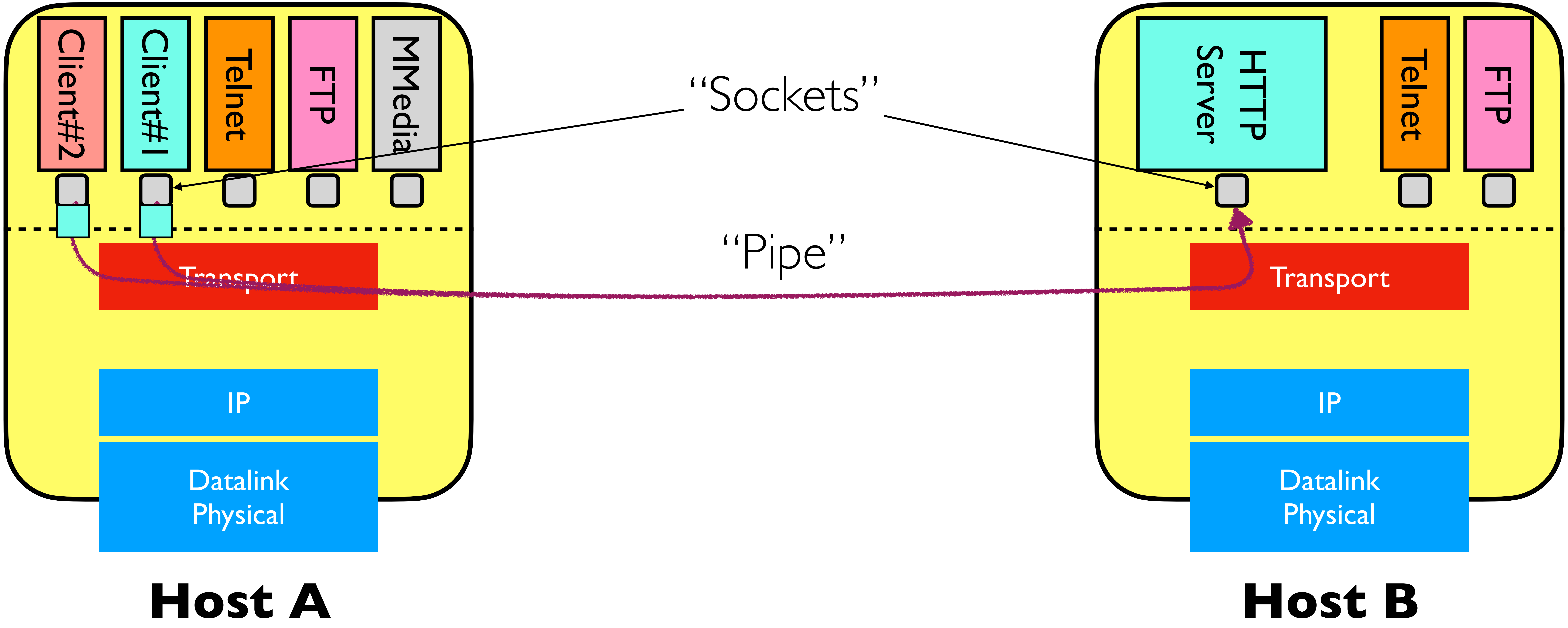
UDP (Datagram-oriented)

- Sources send packets, destinations receive packets
- If packets delayed/reordered/lost:
  - *"It's not my problem!"*
  - Let application deal with it
- Discarding corrupted packets: optional!
- Nothing else!

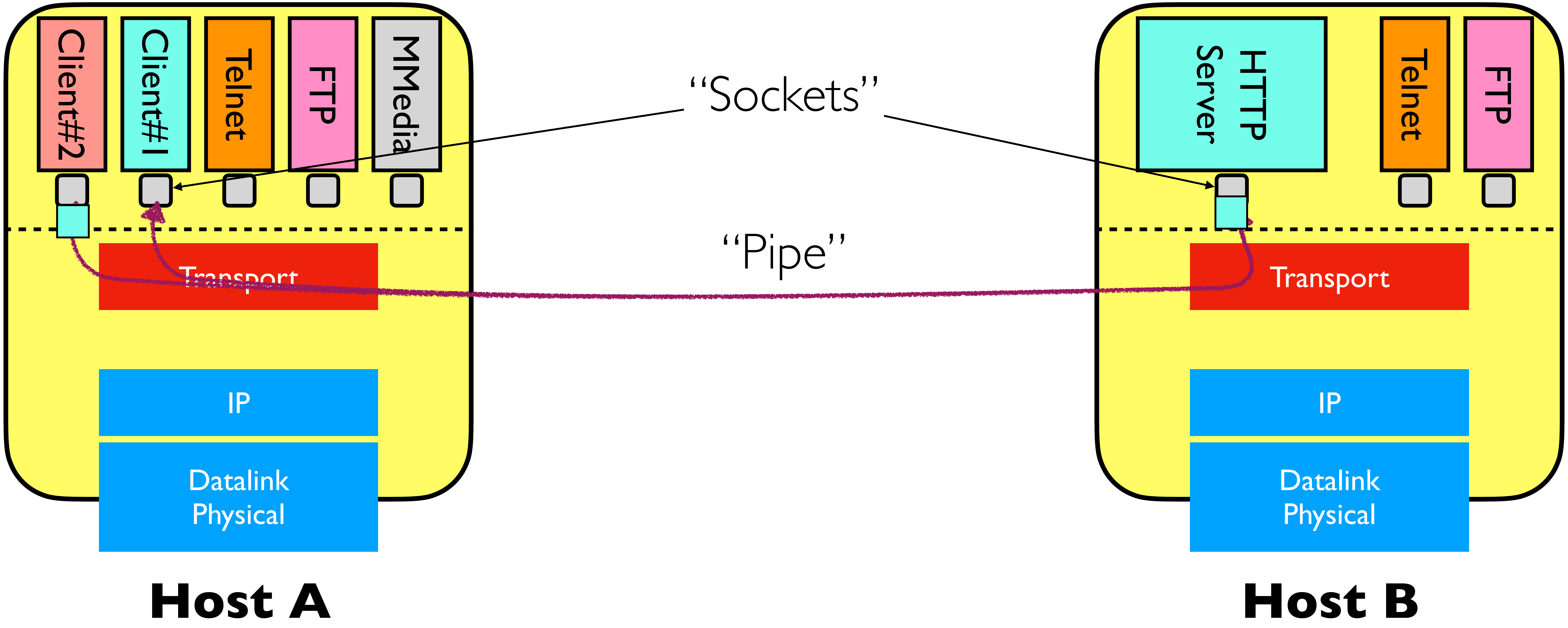
UDP (Datagram-oriented)

- Sources send packets, destinations receive packets
- If packets delayed/reordered/lost:
  - *"It's not my problem!"*
  - Let application deal with it
- Discarding corrupted packets: optional!
- Nothing else!
- Minimal extension to IP

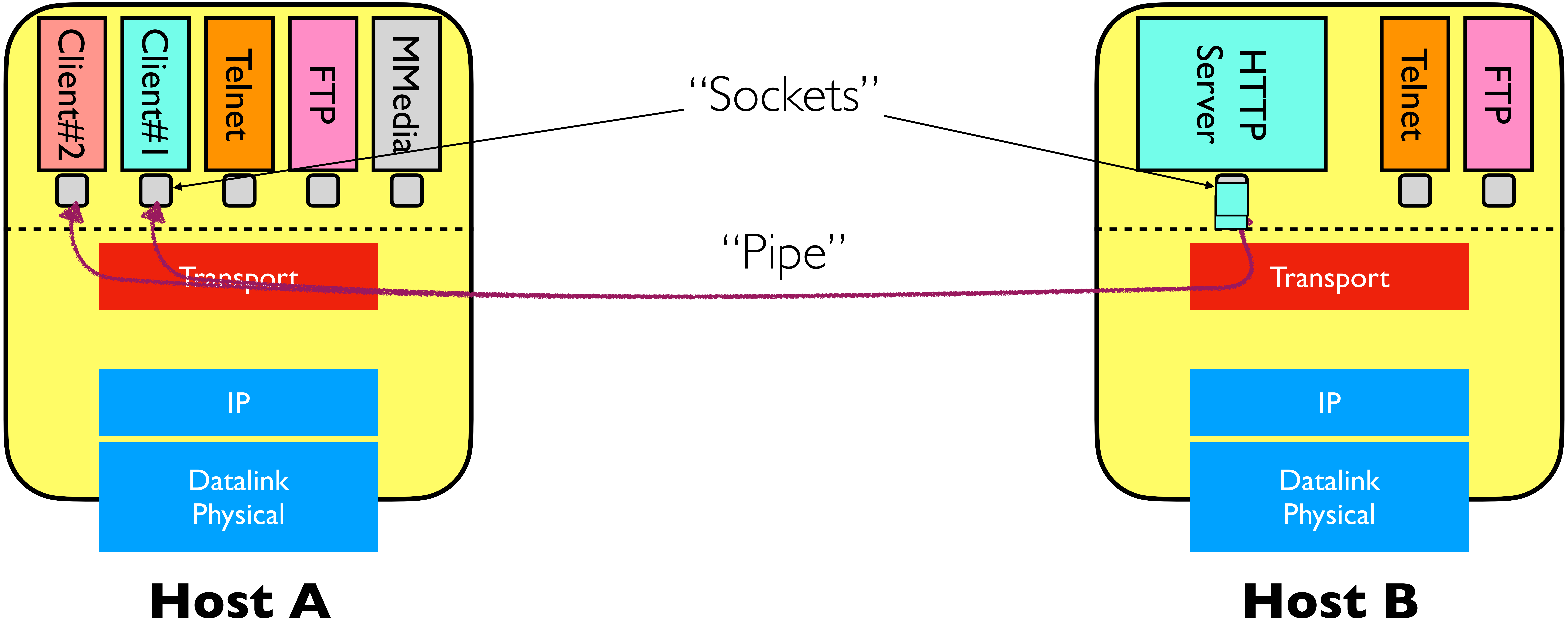
Abstraction: Pipes & Sockets



Abstraction: Pipes & Sockets



Abstraction: Pipes & Sockets



TCP (Stream-oriented)

TCP (Stream-oriented)

- Sources send **byte-segments**

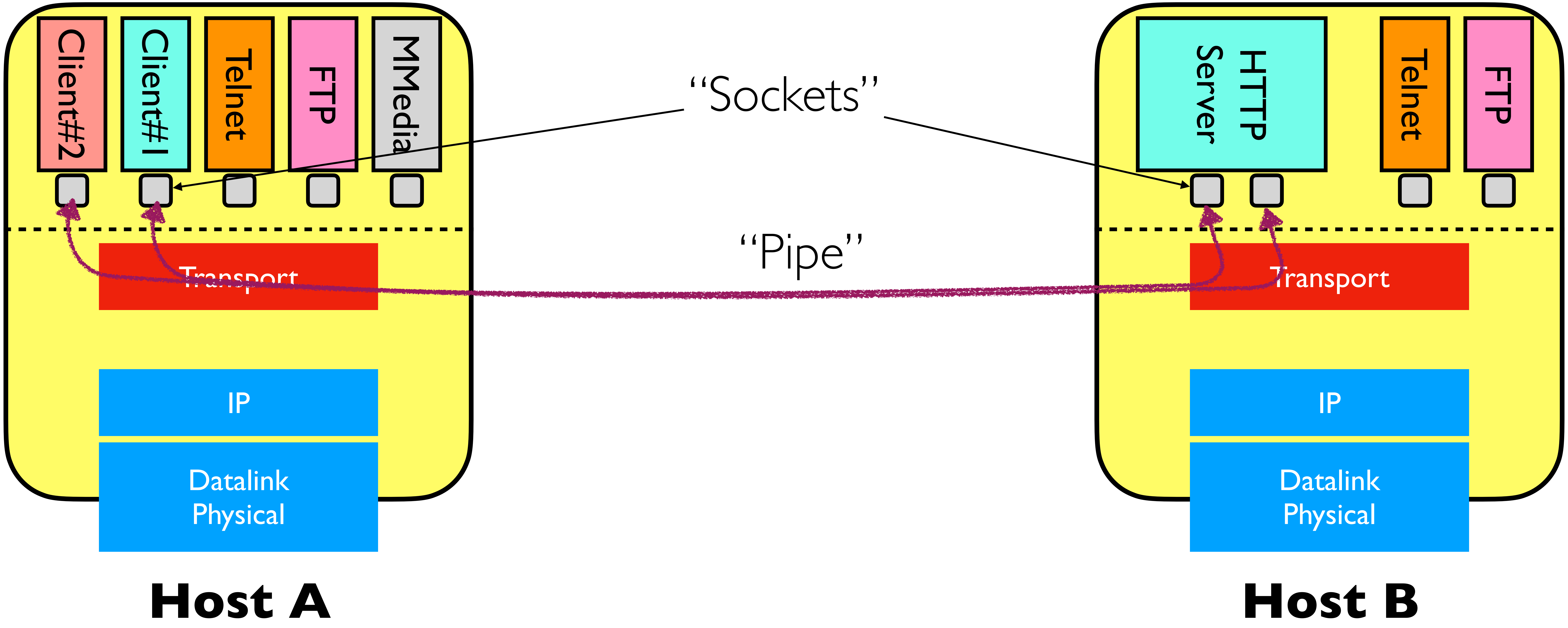
TCP (Stream-oriented)

- Sources send **byte-segments**
- Sources/destinations handle:
  - Lost and/or corrupted packets [source + destination]
  - Flow control (to prevent overwhelming receiver) [source]
  - Congestion control (to prevent overwhelming network) [source]

TCP (Stream-oriented)

- Sources send **byte-segments**
- Sources/destinations handle:
    - Lost and/or corrupted packets [source + destination]
    - Flow control (to prevent overwhelming receiver) [source]
    - Congestion control (to prevent overwhelming network) [source]
- Every source-destination pair has a dedicated stream oriented “connection” (or “session”)

Abstraction: Pipes & Sockets



Sockets

Sockets

- **Socket:** software abstraction by which an application process exchanges network messages with the (transport layer of the) operating system
  - `socketID = socket(…, SOCK_TYPE)`
  - `socketID.sendto(message, …)`
  - `socketID.recvfrom(…)`
  - Will cover in detail later in the course

Sockets

- **Socket:** software abstraction by which an application process exchanges network messages with the (transport layer of the) operating system
    - `socketID = socket(…, SOCK_TYPE)`
    - `socketID.sendto(message, …)`
    - `socketID.recvfrom(…)`
    - Will cover in detail later in the course
- Two important types of sockets
    - UDP socket: `SOCK_TYPE` is `SOCK_DGRAM`
    - TCP socket: `SOCK_TYPE` is `SOCK_STREAM`

Ports

Ports

- **Problem:** deciding which application (socket) gets which packets

Ports

- **Problem:** deciding which application (socket) gets which packets
- **Solution:** <u>*port*</u> as a transport layer identifier (16 bits)
    - Packet carries source/destination port numbers in transport header

Ports

- **Problem:** deciding which application (socket) gets which packets
- **Solution:** *port* as a transport layer identifier (16 bits)
  - Packet carries source/destination port numbers in transport header
- OS stores mapping between sockets and ports
  - *Port: in packets*
  - *Socket: in OS*

Ports

- **Problem:** deciding which application (socket) gets which packets
- **Solution:** <u>*port*</u> as a transport layer identifier (16 bits)
  - Packet carries source/destination port numbers in transport header
- OS stores mapping between sockets and ports
  - <u>*Port: in packets*</u>
  - <u>*Socket: in OS*</u>
- For UDP ports (**`SOCK_DGRAM`**)
  - OS stores (local port, local IP address) ←→ socket

Ports

- **Problem:** deciding which application (socket) gets which packets
- **Solution:** <u>*port*</u> as a transport layer identifier (16 bits)
  - Packet carries source/destination port numbers in transport header
- OS stores mapping between sockets and ports
  - <u>*Port: in packets*</u>
  - <u>*Socket: in OS*</u>
- For UDP ports (`SOCK_DGRAM`)
  - OS stores (local port, local IP address) ←→ socket
- For TCP ports (`SOCK_STREAM`)
  - OS stores (local port, local IP, remote port, remote IP) ←→ socket

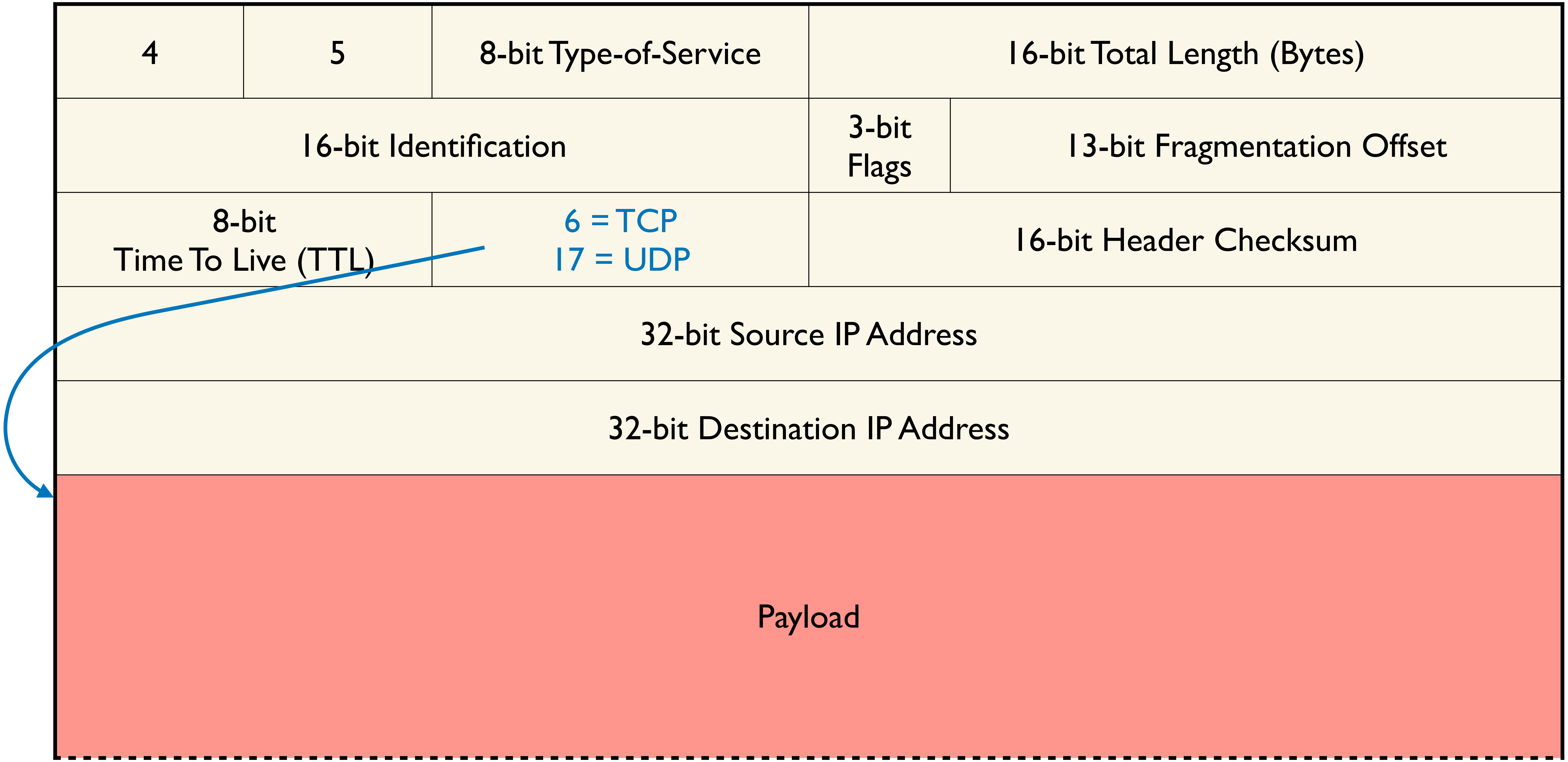
Ports

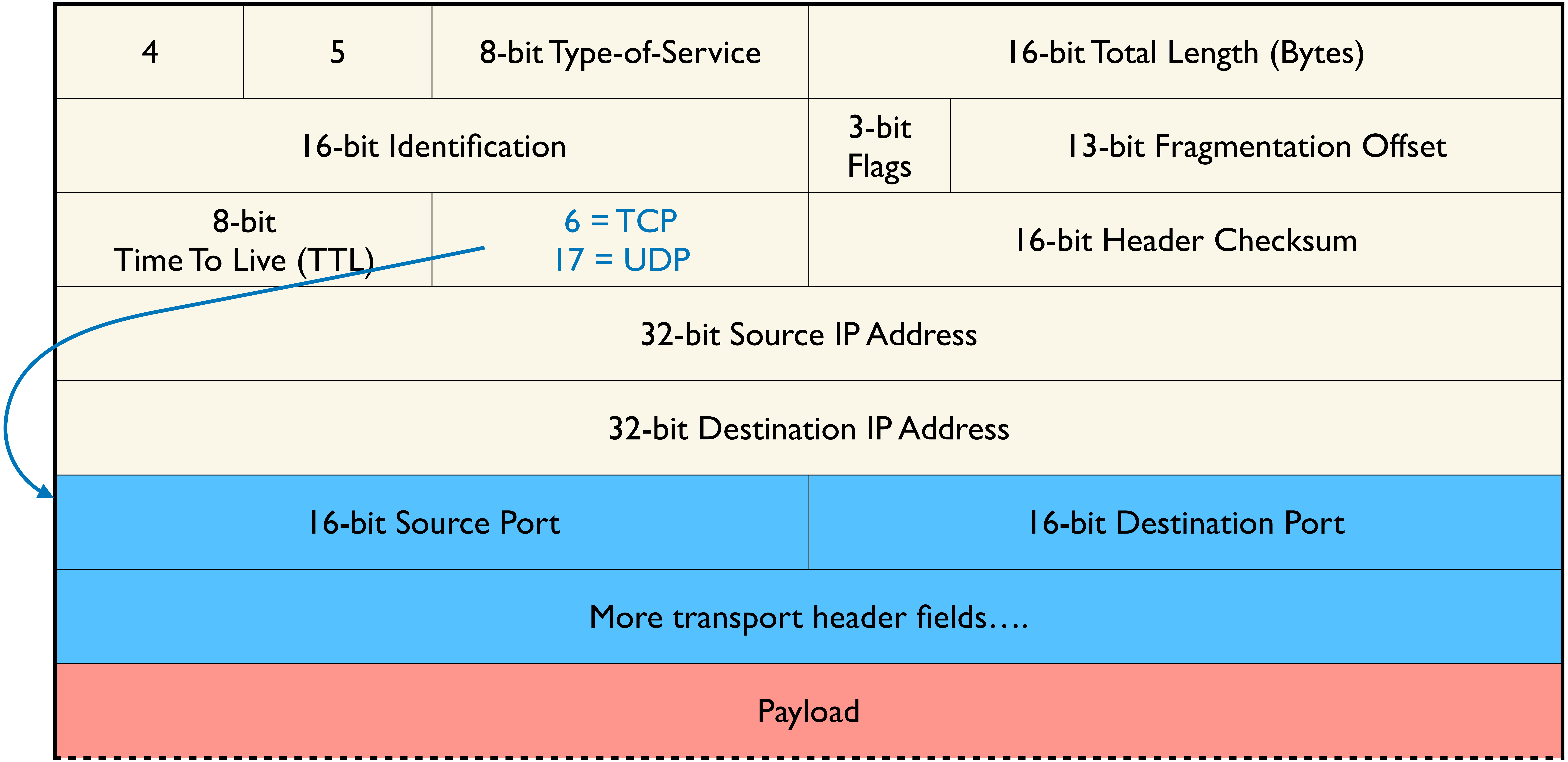
- **Problem:** deciding which application (socket) gets which packets
- **Solution:** *port* as a transport layer identifier (16 bits)
  - Packet carries source/destination port numbers in transport header
- OS stores mapping between sockets and ports
  - *Port: in packets*
  - *Socket: in OS*
- For UDP ports (`SOCK_DGRAM`)
  - OS stores (local port, local IP address) ←→ socket
- For TCP ports (`SOCK_STREAM`)
  - OS stores (local port, local IP, remote port, remote IP) ←→ socket
- **Q: Why the difference?**

<table>
<tr><td>4-bit Version</td><td>4-bit Header Length</td><td>8-bit Type-of-Service</td><td colspan="2">16-bit Total Length (Bytes)</td></tr>
<tr><td colspan="3">16-bit Identification</td><td>3-bit Flags</td><td>13-bit Fragmentation Offset</td></tr>
<tr><td colspan="2">8-bit Time To Live (TTL)</td><td>8-bit Protocol</td><td colspan="2">16-bit Header Checksum</td></tr>
<tr><td colspan="5">32-bit Source IP Address</td></tr>
<tr><td colspan="5">32-bit Destination IP Address</td></tr>
<tr><td colspan="5">Options (if any)</td></tr>
<tr><td colspan="5">Payload</td></tr>
</table>

<table>
<tr><td>4</td><td>5</td><td>8-bit Type-of-Service</td><td colspan="2">16-bit Total Length (Bytes)</td></tr>
<tr><td colspan="3">16-bit Identification</td><td>3-bit Flags</td><td>13-bit Fragmentation Offset</td></tr>
<tr><td colspan="2">8-bit Time To Live (TTL)</td><td>8-bit Protocol</td><td colspan="2">16-bit Header Checksum</td></tr>
<tr><td colspan="5">32-bit Source IP Address</td></tr>
<tr><td colspan="5">32-bit Destination IP Address</td></tr>
<tr><td colspan="5">Payload</td></tr>
</table>

<table>
<tr><td>4</td><td>5</td><td>8-bit Type-of-Service</td><td colspan="2">16-bit Total Length (Bytes)</td></tr>
<tr><td colspan="3">16-bit Identification</td><td>3-bit
Flags</td><td>13-bit Fragmentation Offset</td></tr>
<tr><td colspan="2">8-bit
Time To Live (TTL)</td><td>6 = TCP
17 = UDP</td><td colspan="2">16-bit Header Checksum</td></tr>
<tr><td colspan="5">32-bit Source IP Address</td></tr>
<tr><td colspan="5">32-bit Destination IP Address</td></tr>
<tr><td colspan="5">Payload</td></tr>
</table>


4
5
8-bit Type-of-Service
16-bit Total Length (Bytes)
16-bit Identification
3-bit Flags
13-bit Fragmentation Offset
8-bit Time To Live (TTL)
6 = TCP
17 = UDP
16-bit Header Checksum
32-bit Source IP Address
32-bit Destination IP Address
Payload


4
5
8-bit Type-of-Service
16-bit Total Length (Bytes)
16-bit Identification
3-bit Flags
13-bit Fragmentation Offset
8-bit Time To Live (TTL)
6 = TCP
17 = UDP
16-bit Header Checksum
32-bit Source IP Address
32-bit Destination IP Address
16-bit Source Port
16-bit Destination Port
More transport header fields….
Payload

More on Ports

More on Ports

- **Separate 16-bit port address space for UDP and TCP**

More on Ports

- **Separate 16-bit port address space for UDP and TCP**
- **“Well known” ports (0-1023):** everyone agrees which services run on these ports
  - e.g., ssh:22, http:80
  - Helps client know server’s port
  - Services can listen on well-known ports

More on Ports

- **Separate 16-bit port address space for UDP and TCP**
- **"Well known" ports (0-1023):** everyone agrees which services run on these ports
  - e.g., ssh:22, http:80
  - Helps client know server's port
  - Services can listen on well-known ports
- **Ephemeral ports (most of 1024-65535):** given to clients

Summary: Multiplexing & Demultiplexing

Summary: Multiplexing & Demultiplexing

- **Host receives IP packets**
  - Each IP header has source and destination IP address
  - Each Transport Layer header has source and destination port number

Summary: Multiplexing & Demultiplexing

- **Host receives IP packets**
  - Each IP header has source and destination IP address
  - Each Transport Layer header has source and destination port number
- **Host uses IP addresses and port numbers to direct the message to appropriate socket**
  - UDP maps local destination port and address to socket
  - TCP maps address pair and port pair to socket

Why a transport layer?

- **IP packets are addressed to a host, but end-to-end communication is between application processes at hosts**
  - Need a way to decide which packets go to which applications (*multiplexing/demultiplexing*)
- **IP provides a weak service model (*best-effort*)**
  - Packets can be corrupted, delayed, dropped, reordered, duplicated

Rest of Lecture

Rest of Lecture

- **Reliable Transport**

Rest of Lecture

- **Reliable Transport**
- **Next Lecture: Details of TCP**

Reliable Transport

- **In a perfect world, reliable transport is easy**

@Sender
- Send packets

@Receiver
- Wait for packets

Reliable Transport

- **In a perfect world, reliable transport is easy**
- **All the bad things "best-effort" can permit:**
  - A packet is corrupted (bit errors)
  - A packet is lost (*why?*)
  - A packet is delayed (*why?*)
  - Packets are reordered (*why?*)
  - A packet is duplicated (*why?*)

Reliable Transport

- **Mechanisms for coping with bad things:**

Reliable Transport

- **Mechanisms for coping with bad things:**
  - **Checksums:** to detect corruption

Reliable Transport

- **Mechanisms for coping with bad things:**
  - **Checksums:** to detect corruption
  - **ACKs:** receiver tells sender that it received packet

Reliable Transport

- **Mechanisms for coping with bad things:**
  - **Checksums:** to detect corruption
  - **ACKs:** receiver tells sender that it received packet
  - **NACKs:** receiver tells sender that it did not receive packet

Reliable Transport

- **Mechanisms for coping with bad things:**
  - **Checksums:** to detect corruption
  - **ACKs:** receiver tells sender that it received packet
  - **NACKs:** receiver tells sender that it did not receive packet
  - **Sequence numbers:** a way to identify and order packets

Reliable Transport

- **Mechanisms for coping with bad things:**
  - **Checksums:** to detect corruption
  - **ACKs:** receiver tells sender that it received packet
  - **NACKs:** receiver tells sender that it did not receive packet
  - **Sequence numbers:** a way to identify and order packets
  - **Retransmissions:** sender resends packets

Reliable Transport

- **Mechanisms for coping with bad things:**
  - **Checksums:** to detect corruption
  - **ACKs:** receiver tells sender that it received packet
  - **NACKs:** receiver tells sender that it did not receive packet
  - **Sequence numbers:** a way to identify and order packets
  - **Retransmissions:** sender resends packets
  - **Timeouts:** a way of deciding when to resend a packet

Reliable Transport

- **Mechanisms for coping with bad things:**
  - **Checksums:** to detect corruption
  - **ACKs:** receiver tells sender that it received packet
  - **NACKs:** receiver tells sender that it did not receive packet
  - **Sequence numbers:** a way to identify and order packets
  - **Retransmissions:** sender resends packets
  - **Timeouts:** a way of deciding when to resend a packet
  - ***Forward error correction:*** *a way to mask errors without retransmission*

Reliable Transport

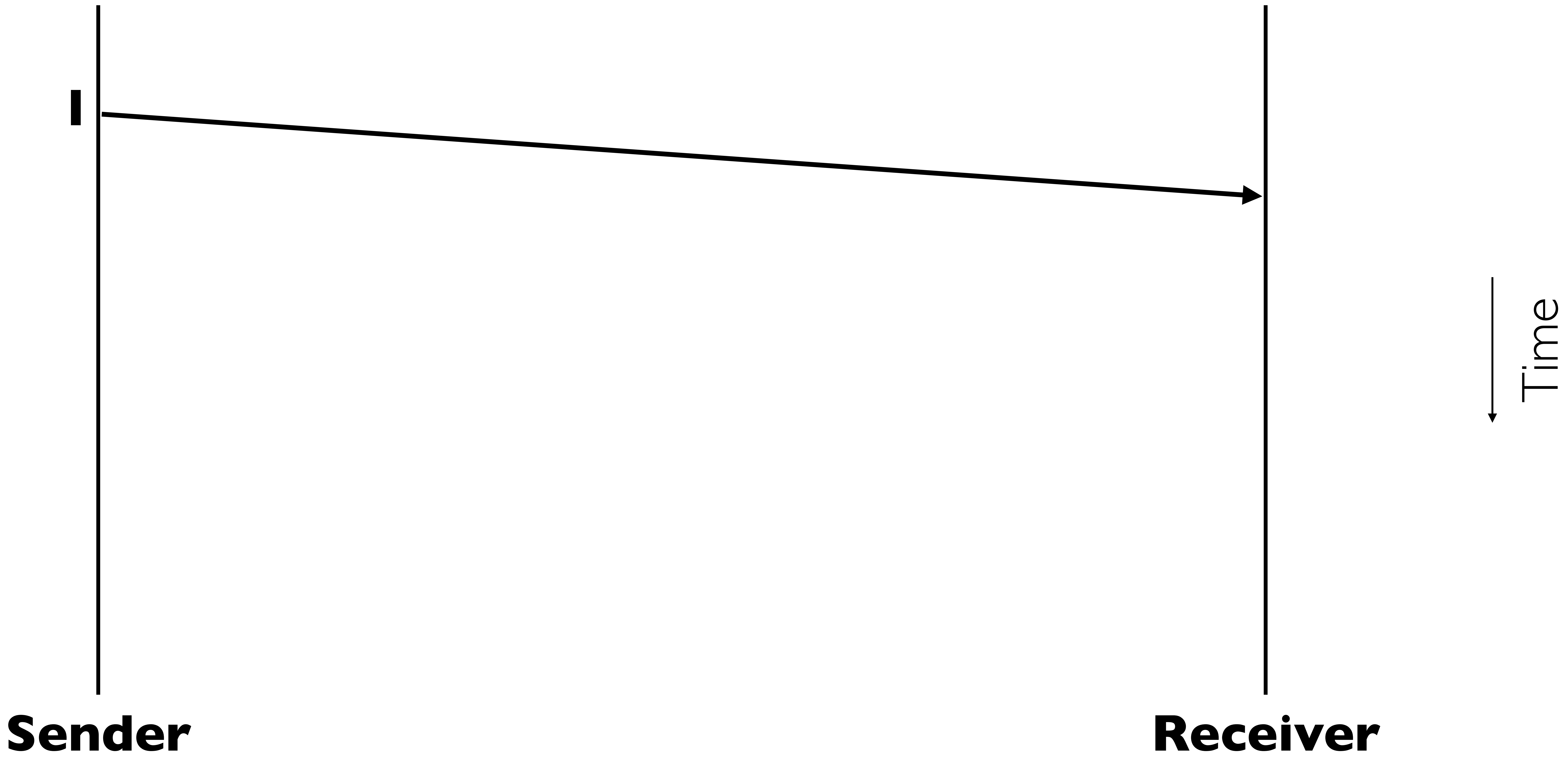
- **Mechanisms for coping with bad things:**
  - **Checksums:** to detect corruption
  - **ACKs:** receiver tells sender that it received packet
  - **NACKs:** receiver tells sender that it did not receive packet
  - **Sequence numbers:** a way to identify and order packets
  - **Retransmissions:** sender resends packets
  - **Timeouts:** a way of deciding when to resend a packet
  - ***Forward error correction:*** *a way to mask errors without retransmission*
  - ***Network encoding:*** *an efficient way to repair errors*

Reliable Transport

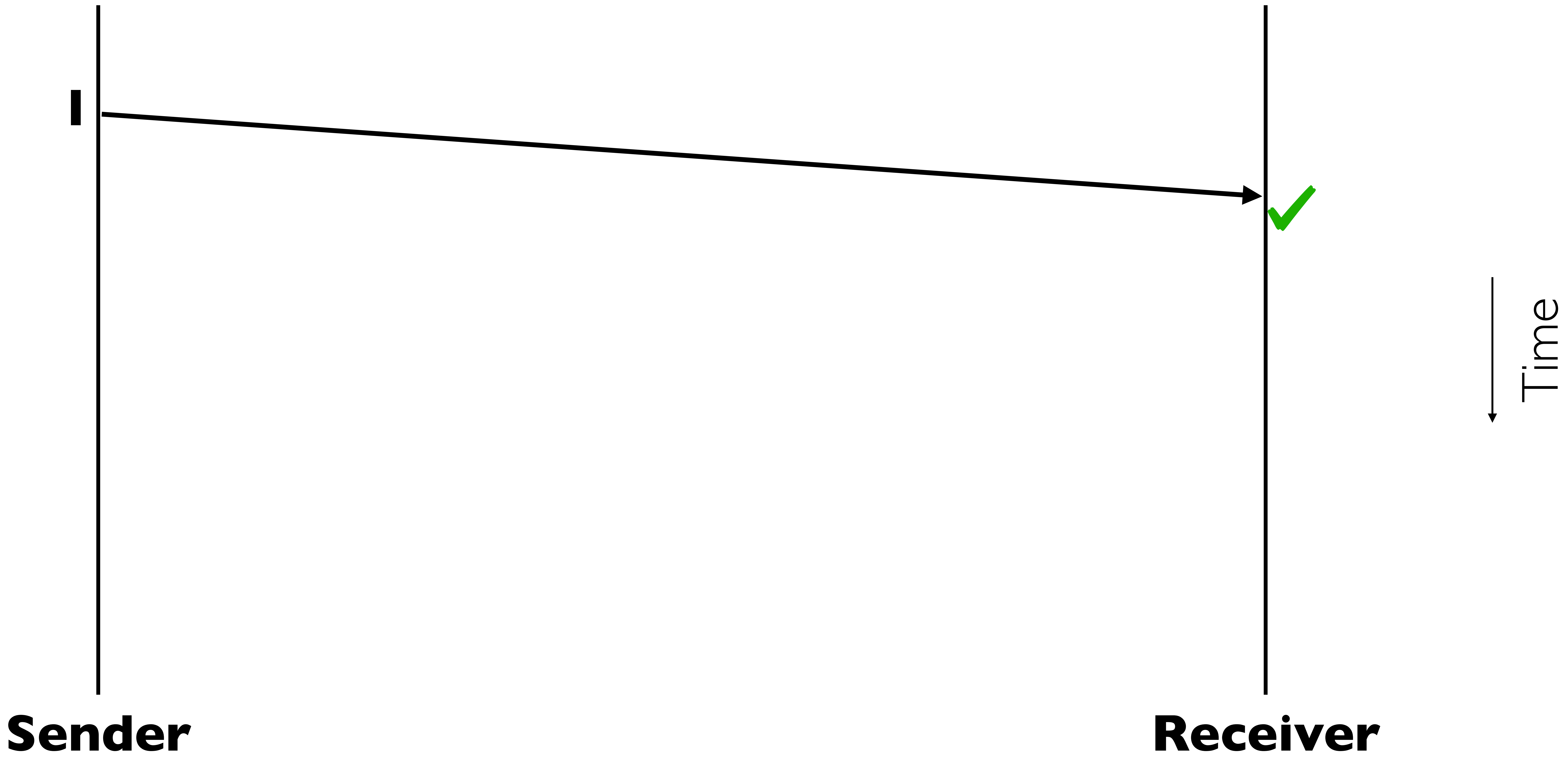
- **Mechanisms for coping with bad things:**
  - **Checksums:** to detect corruption
  - **ACKs:** receiver tells sender that it received packet
  - **NACKs:** receiver tells sender that it did not receive packet
  - **Sequence numbers:** a way to identify and order packets
  - **Retransmissions:** sender resends packets
  - **Timeouts:** a way of deciding when to resend a packet
  - ***Forward error correction:*** *a way to mask errors without retransmission*
  - ***Network encoding:*** *an efficient way to repair errors*
  - …

Dealing with Packet Corruption

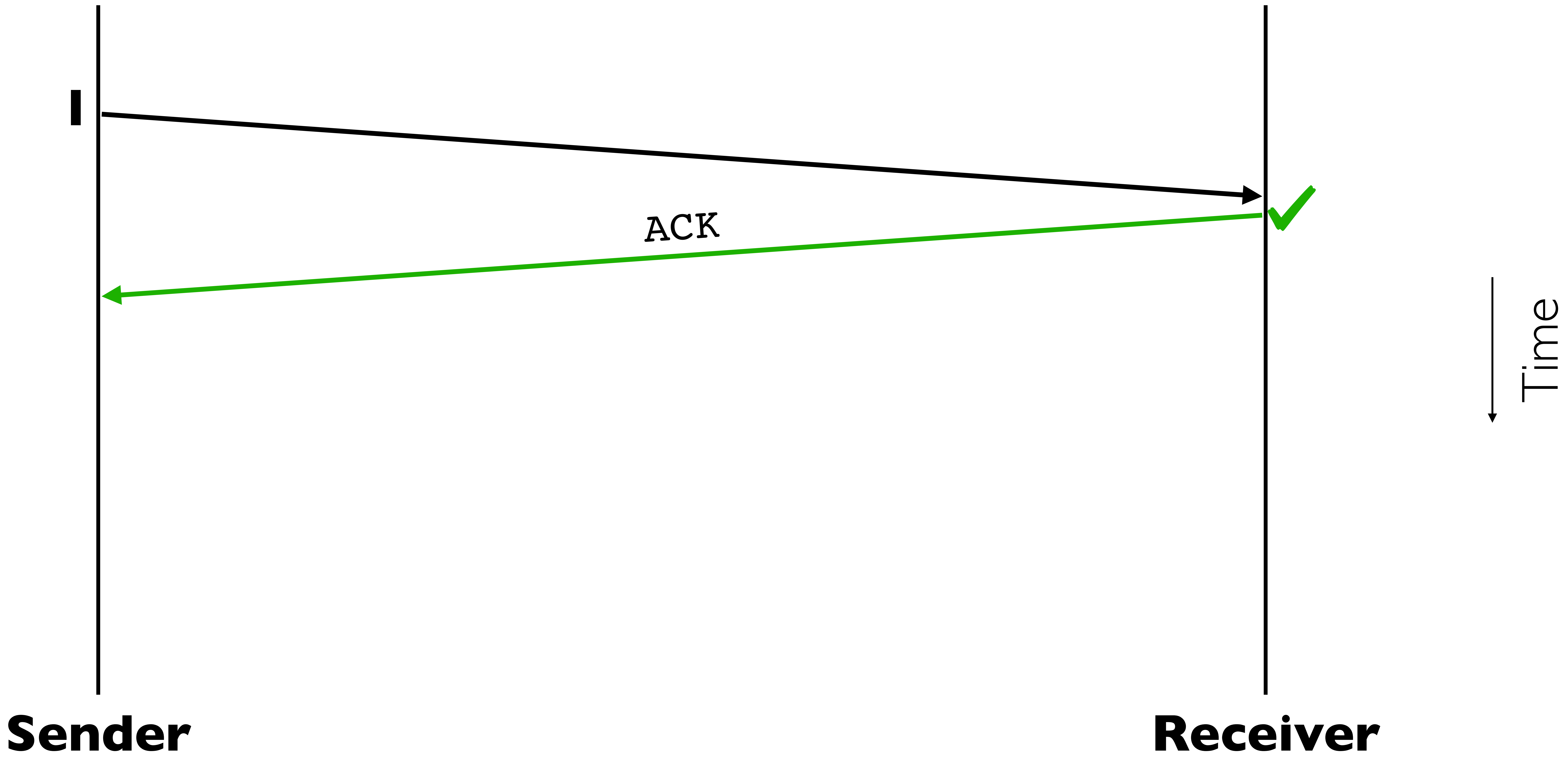
Dealing with Packet Corruption



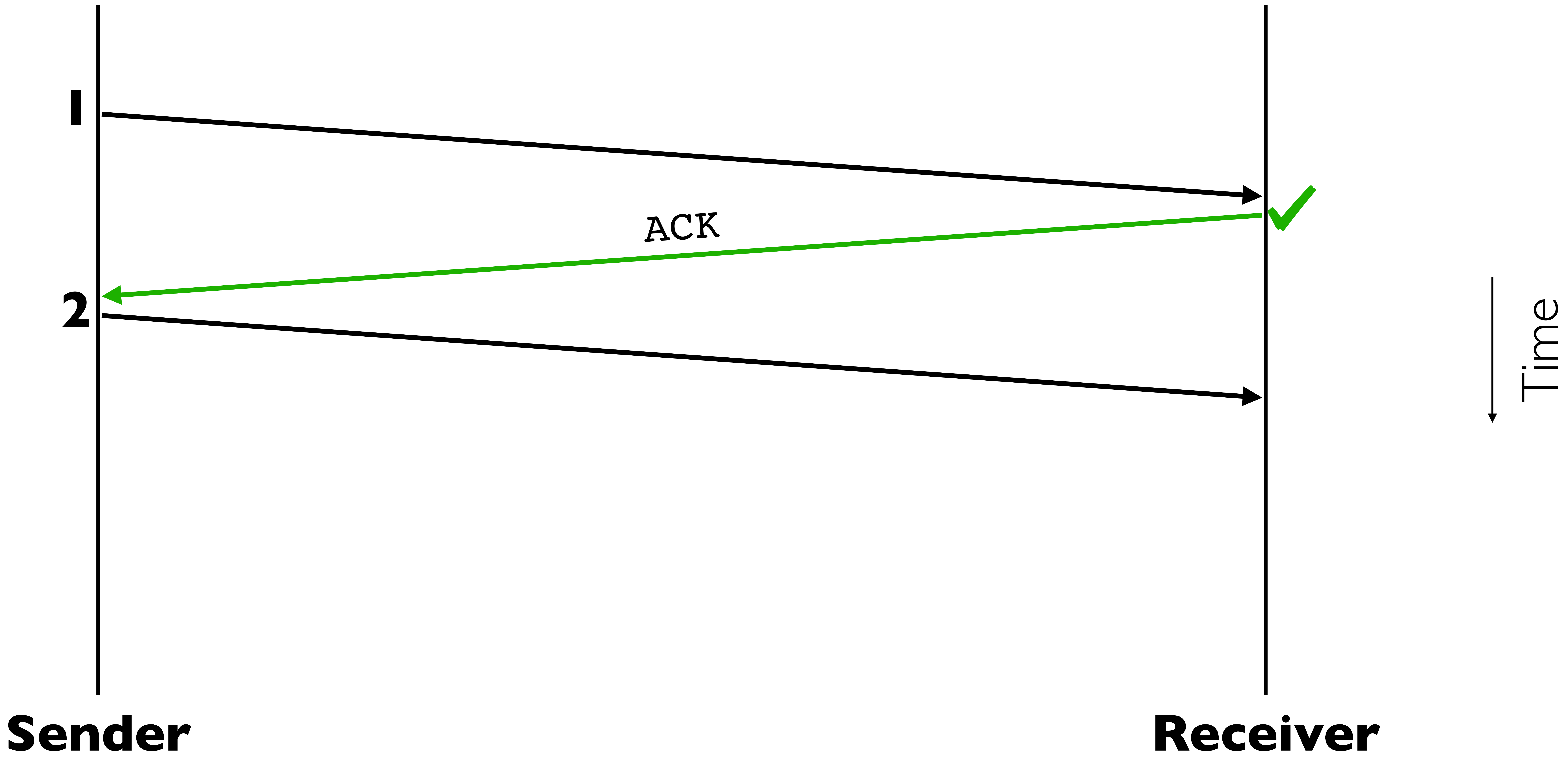
Dealing with Packet Corruption



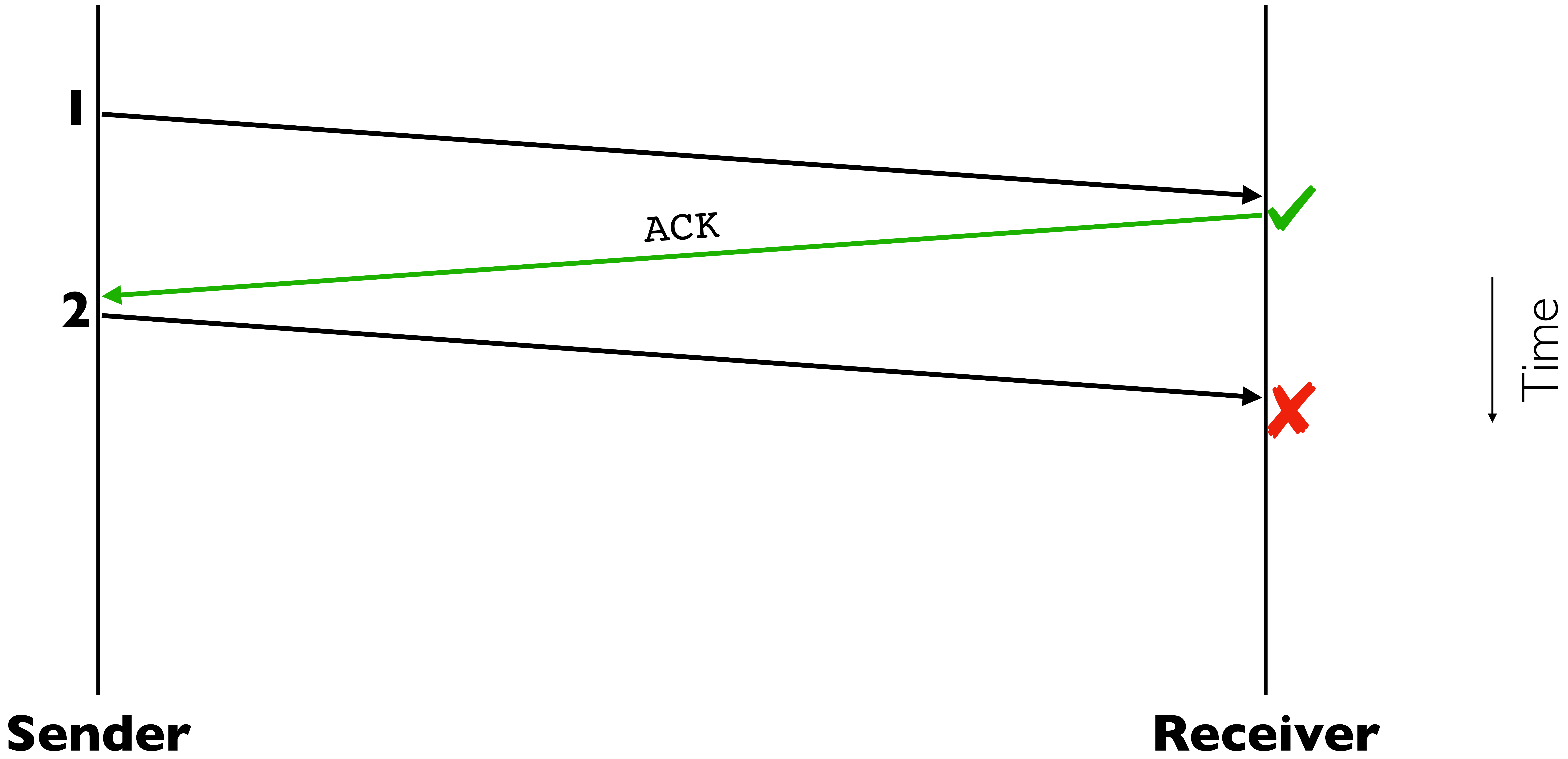
Dealing with Packet Corruption



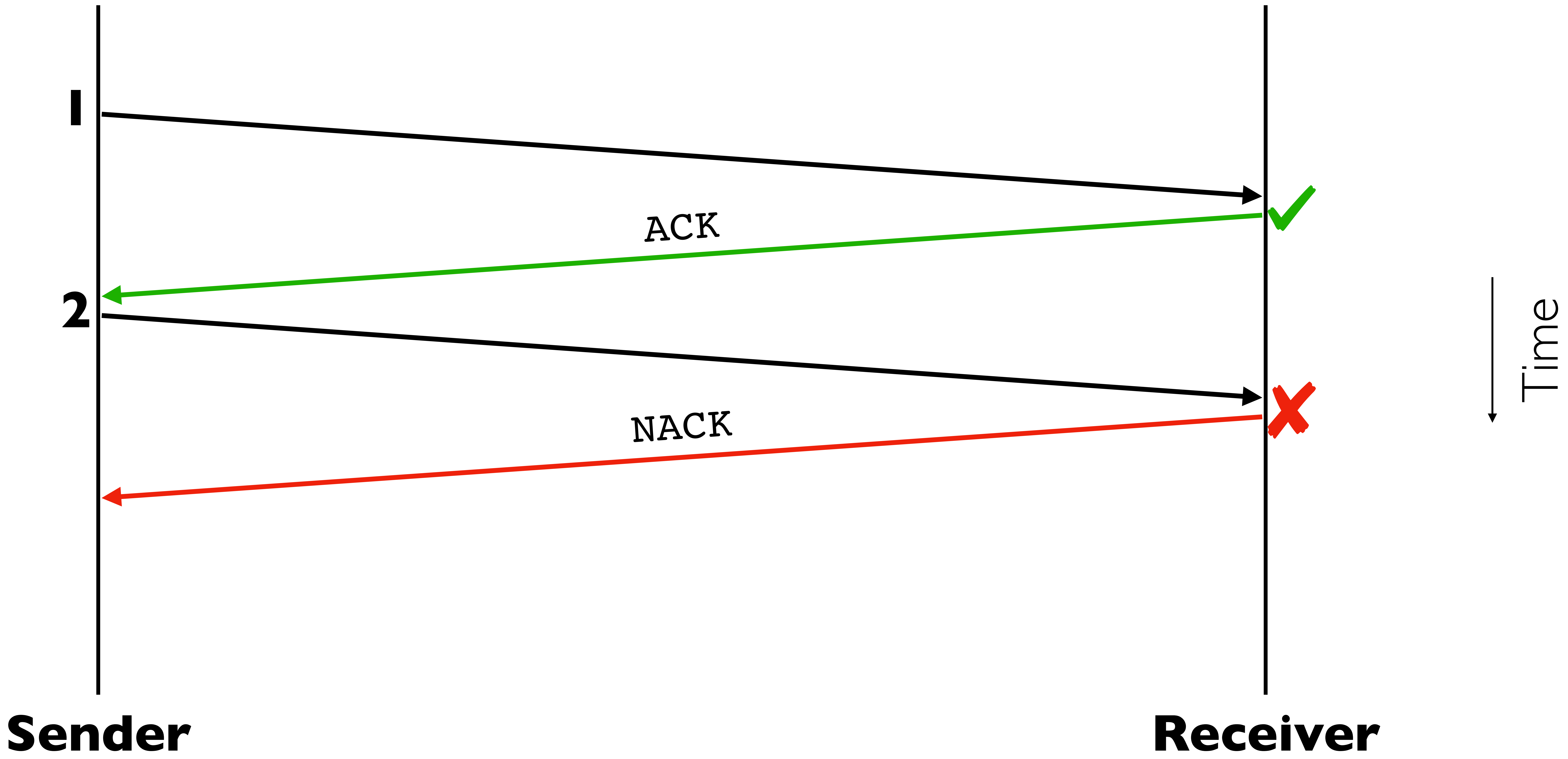
Dealing with Packet Corruption



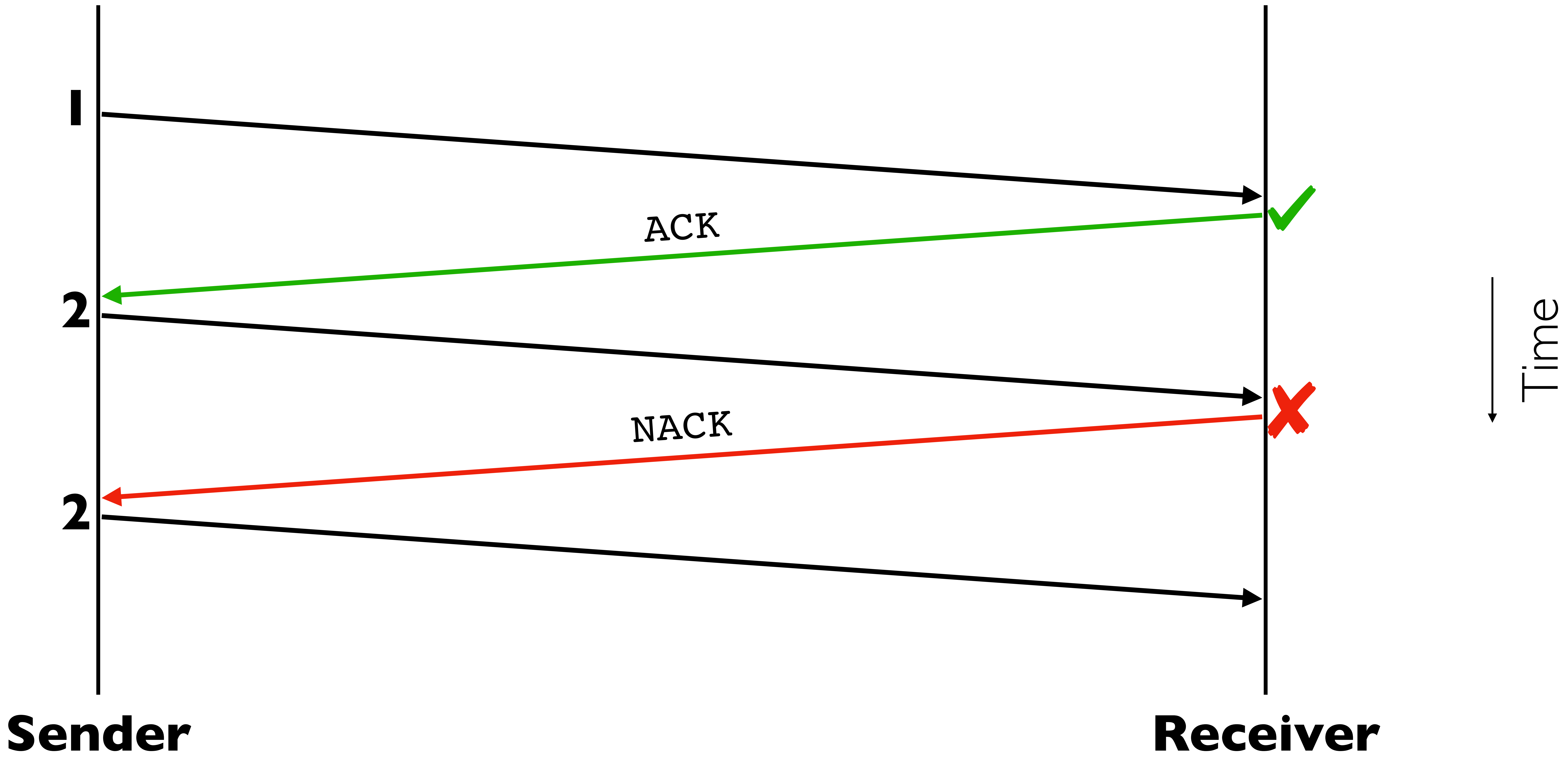
Dealing with Packet Corruption



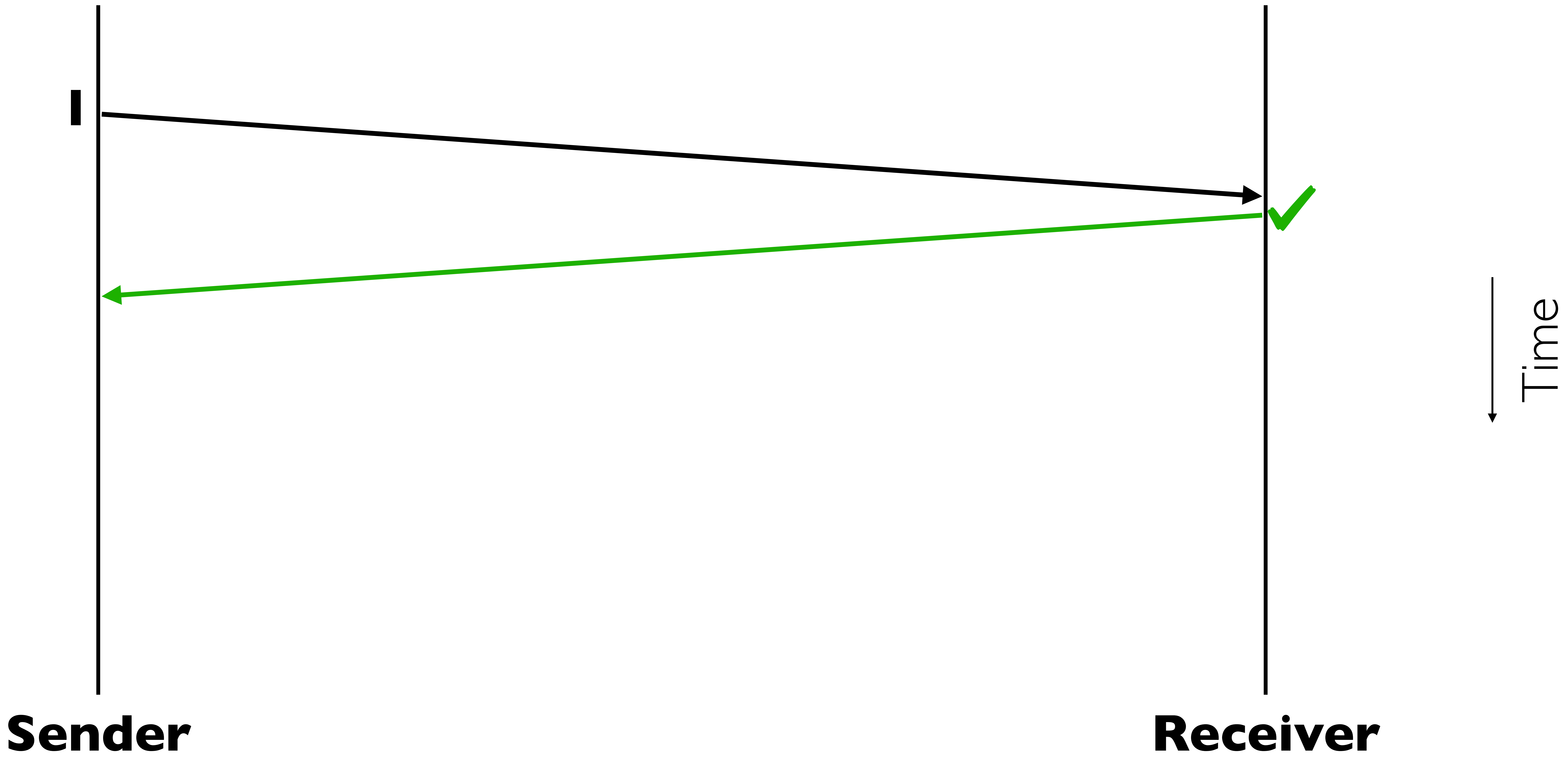
Dealing with Packet Corruption



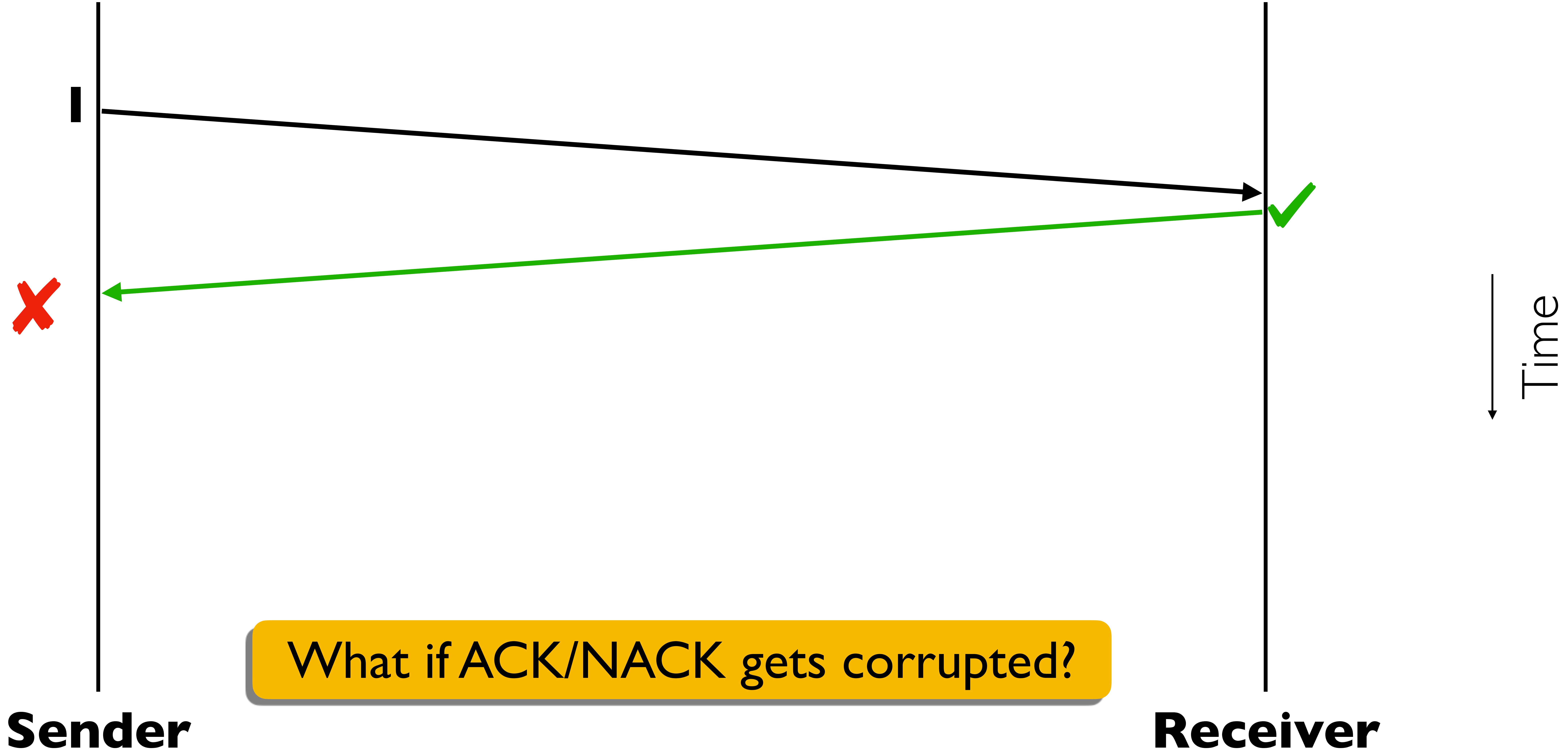
Dealing with Packet Corruption



Dealing with Packet Corruption

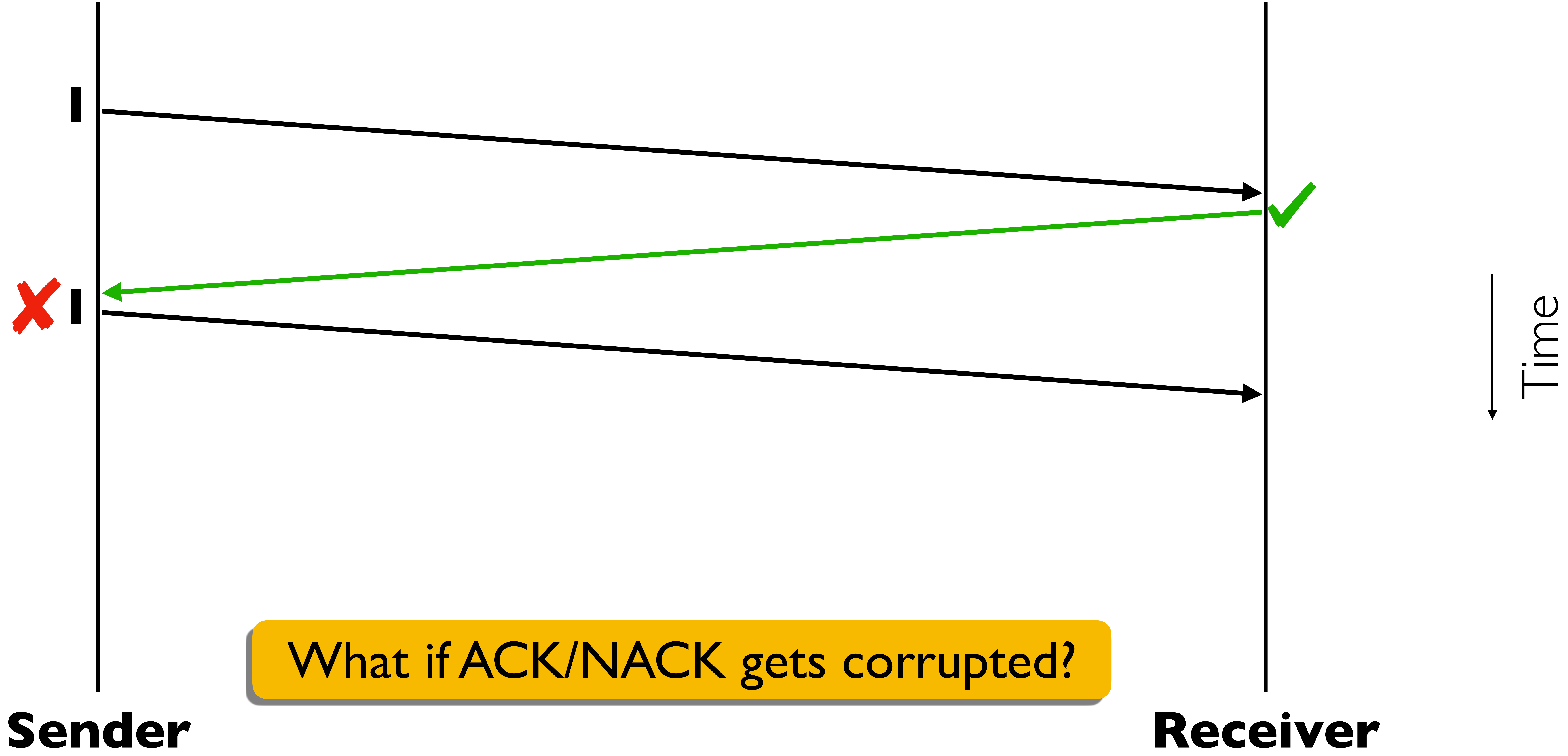


Dealing with Packet Corruption

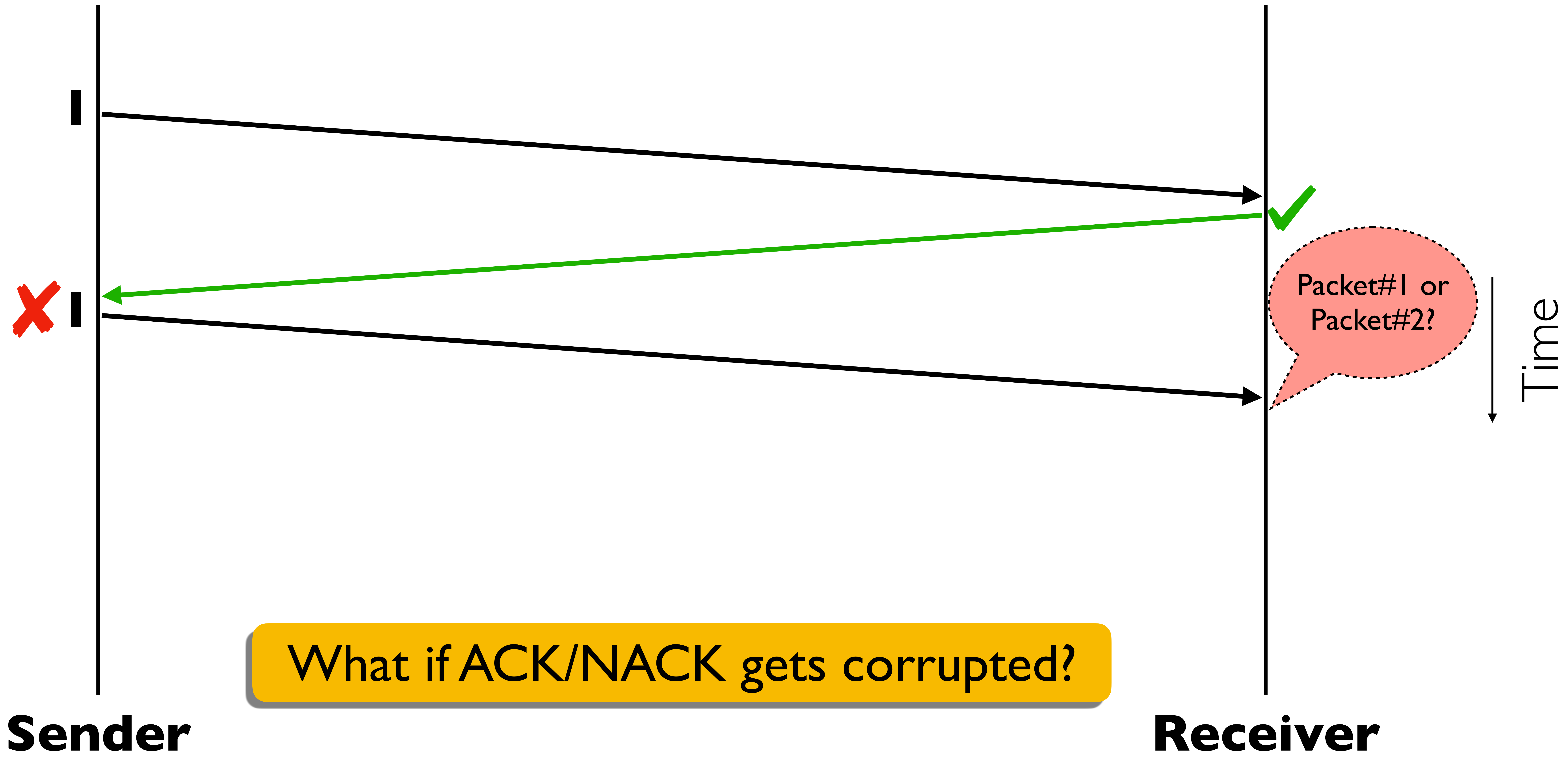


What if ACK/NACK gets corrupted?

Dealing with Packet Corruption

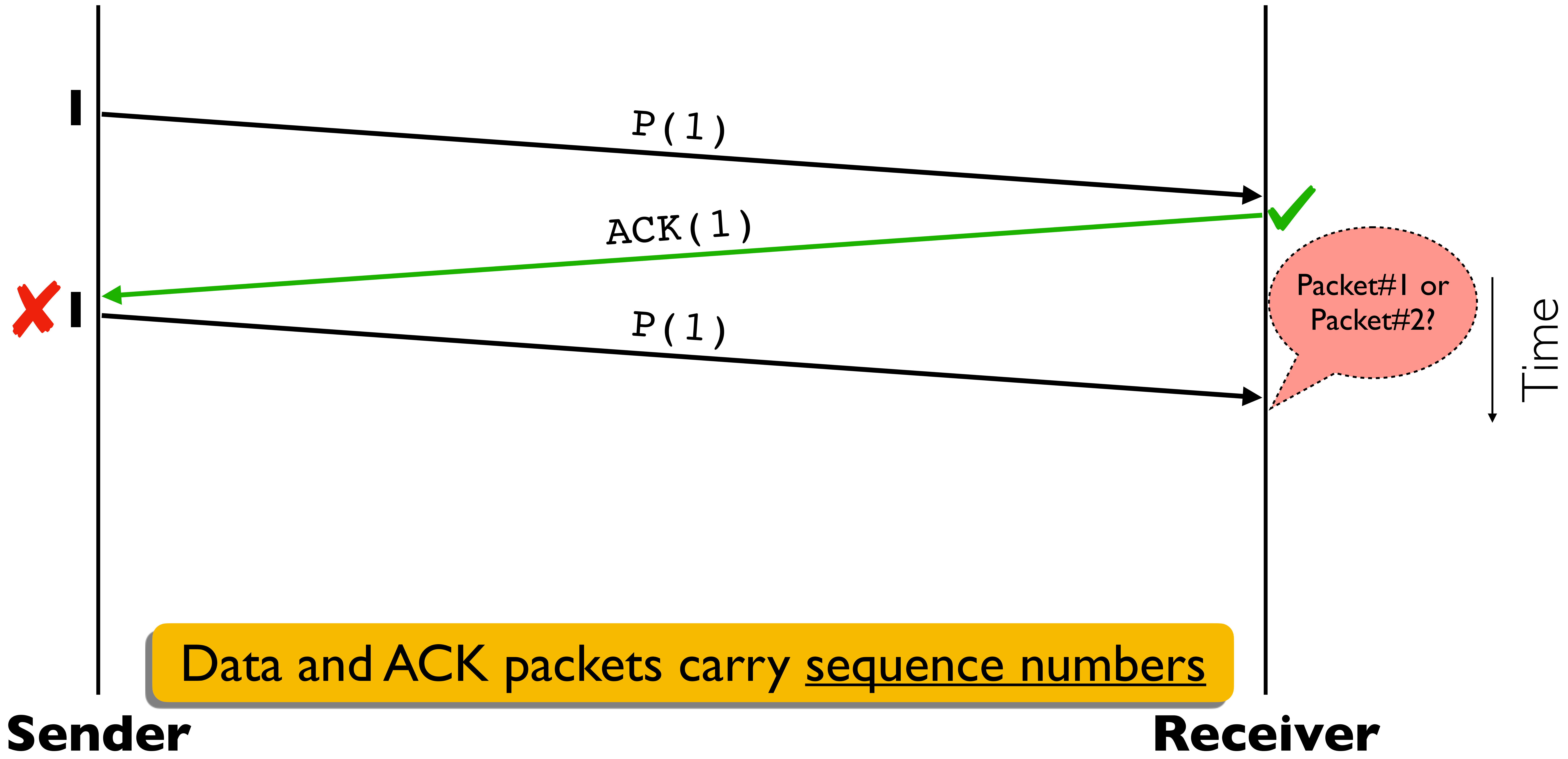


Dealing with Packet Corruption

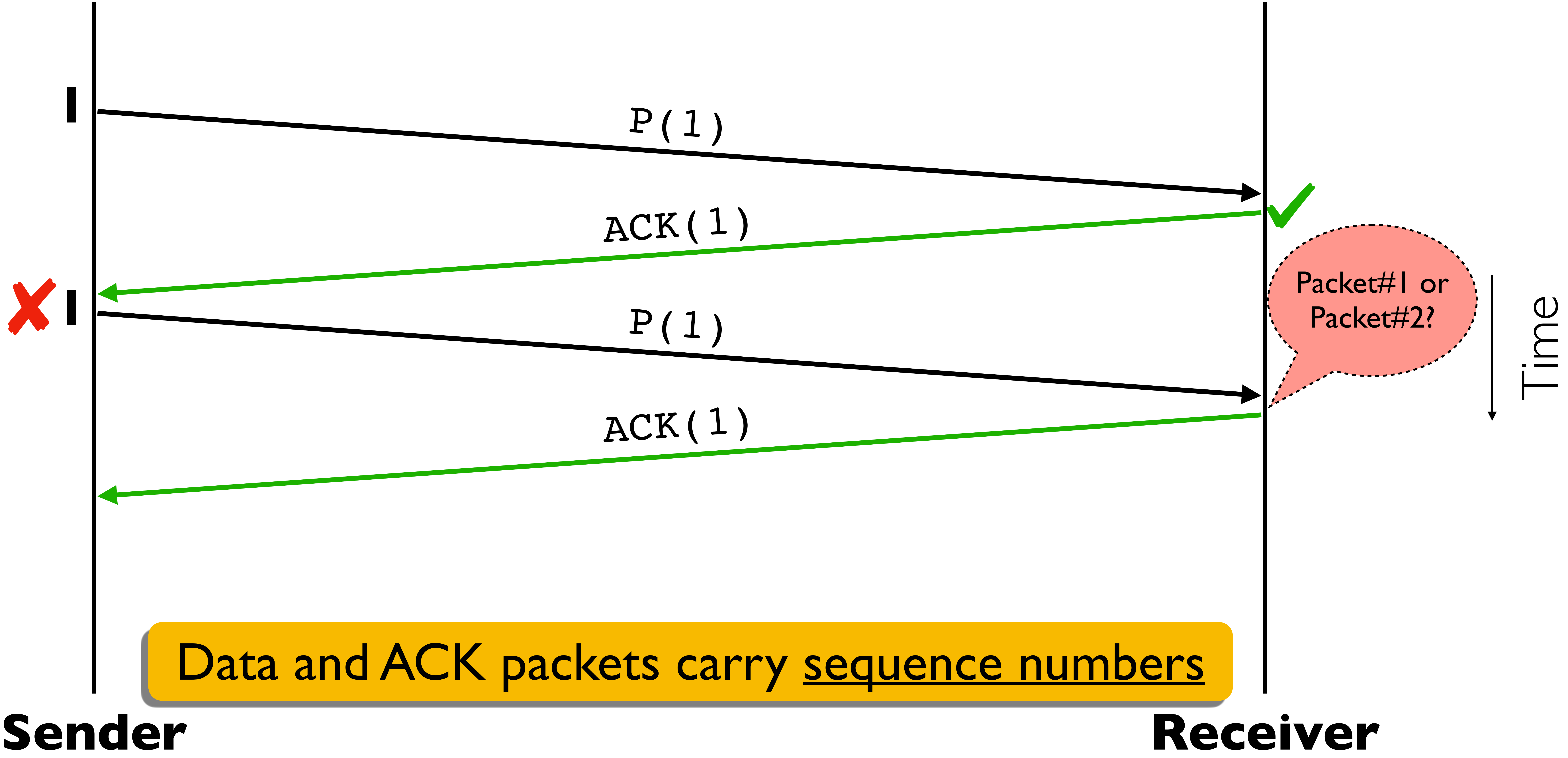


What if ACK/NACK gets corrupted?

Dealing with Packet Corruption

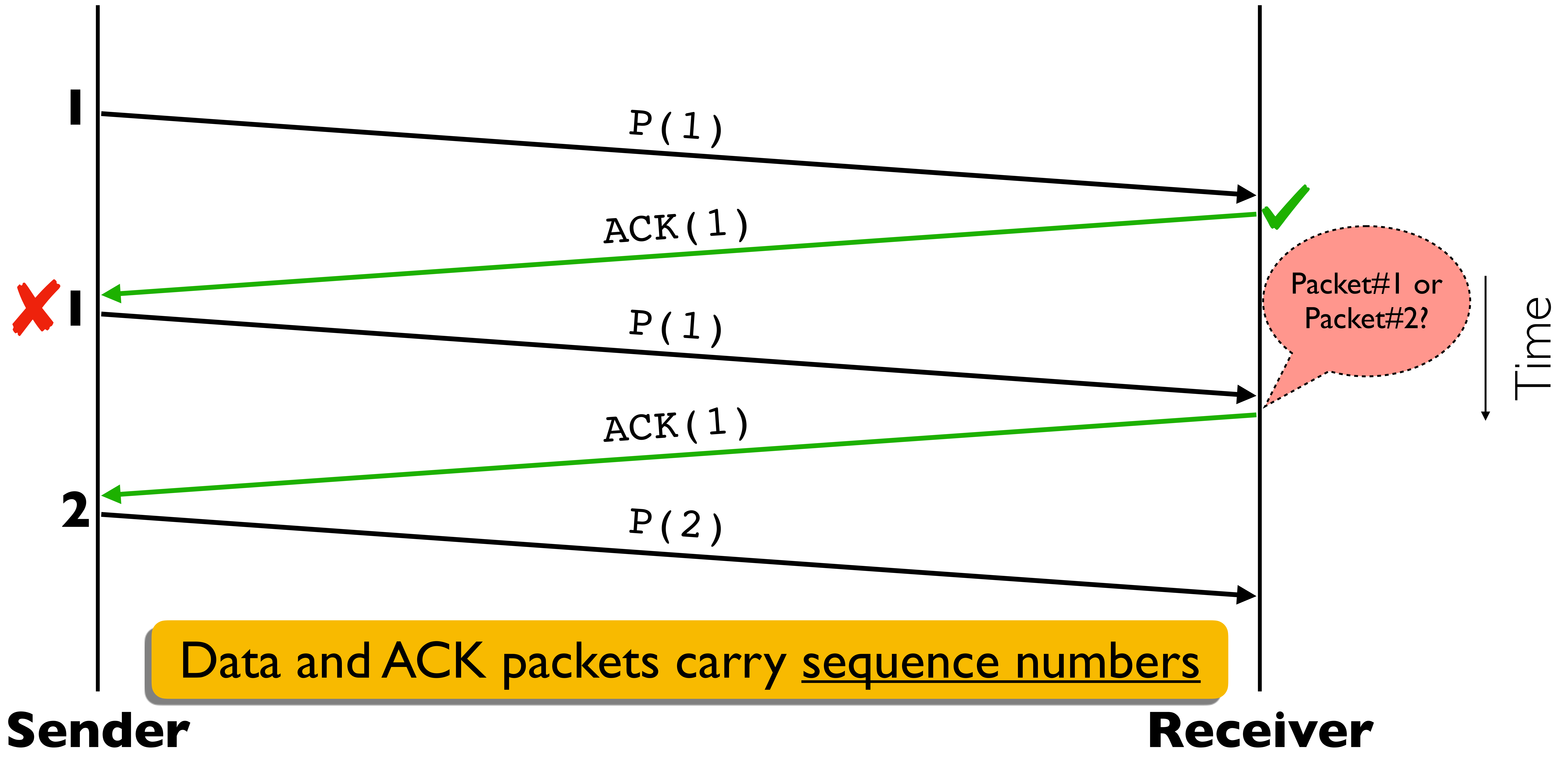


Dealing with Packet Corruption

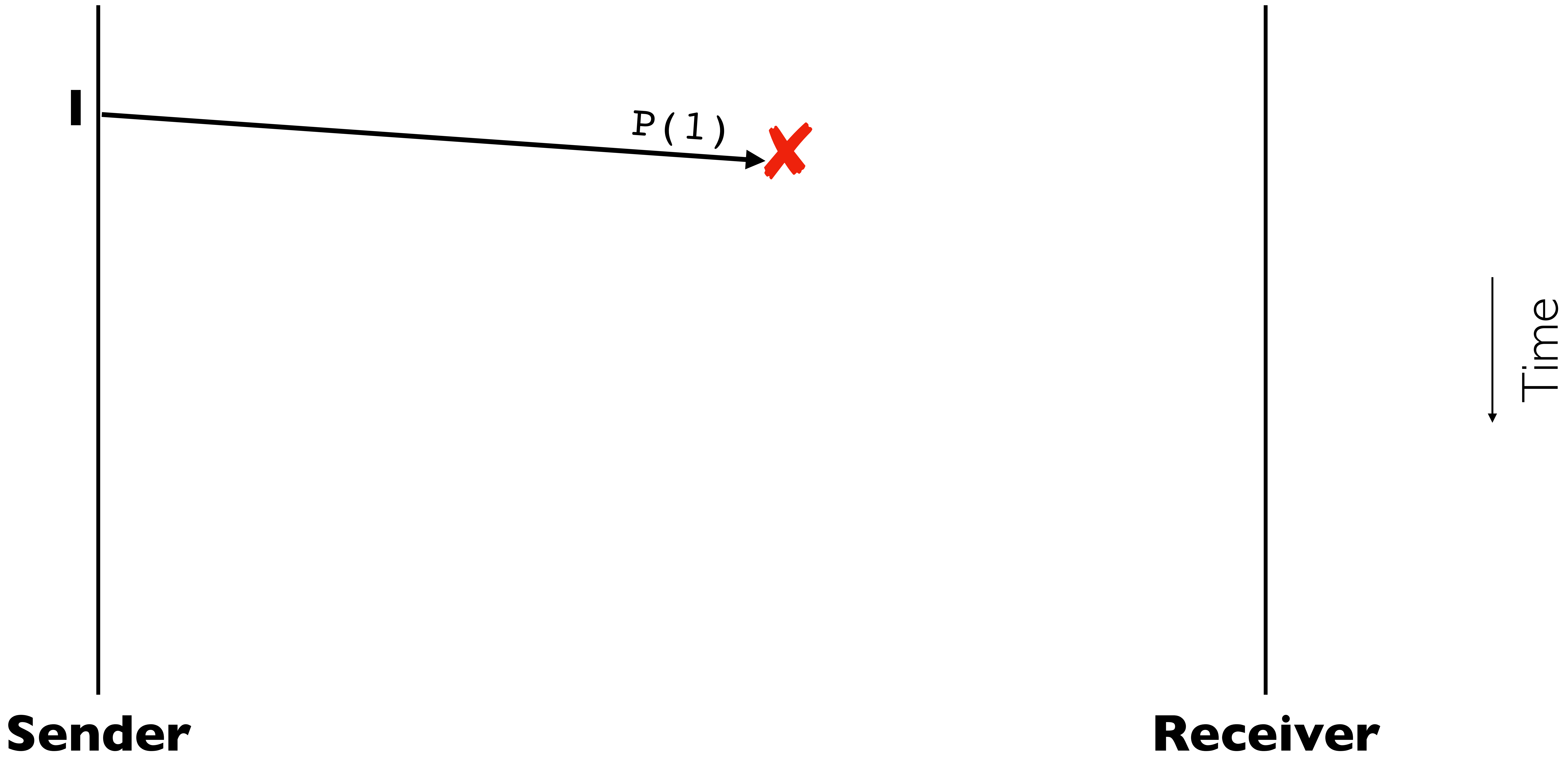


Data and ACK packets carry sequence numbers

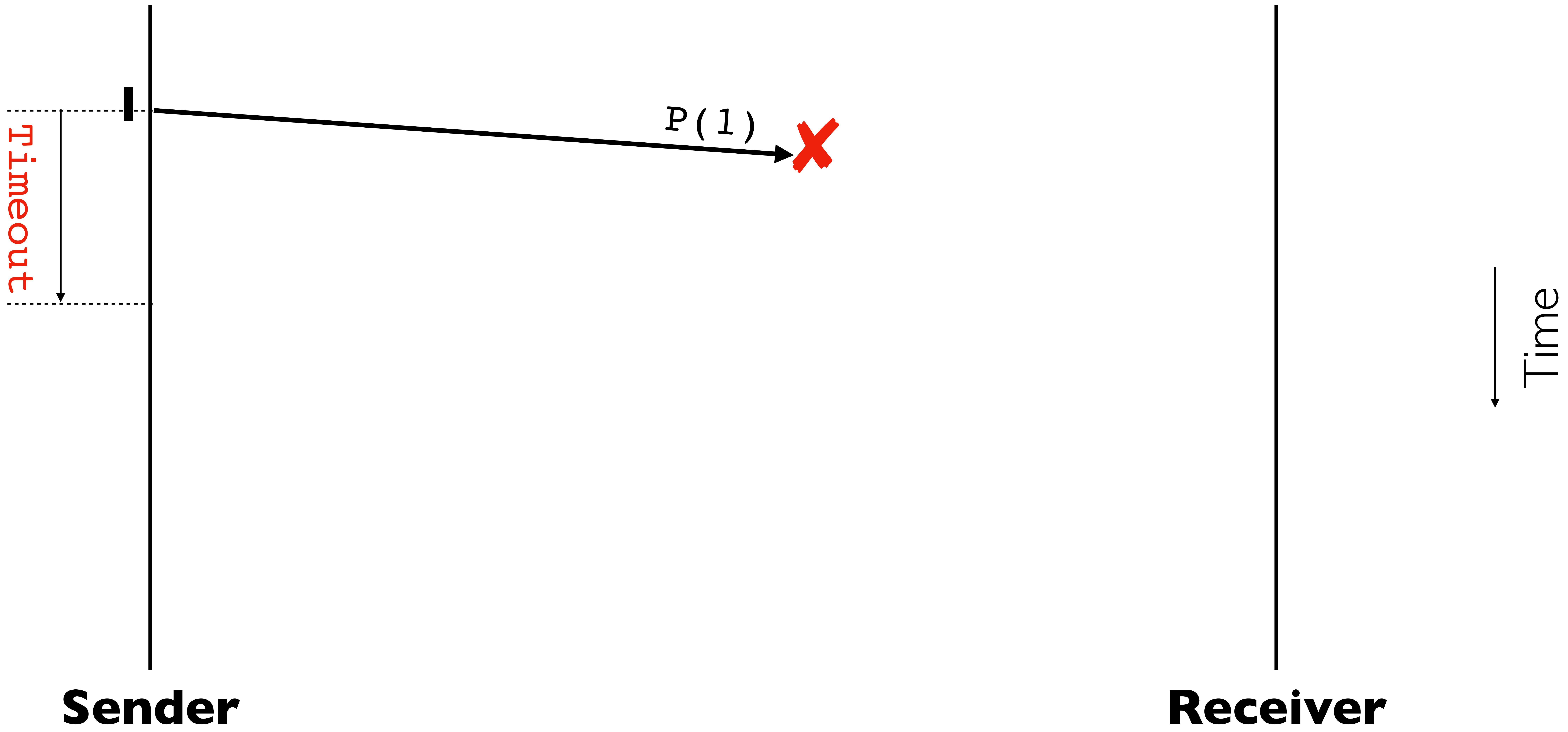
Dealing with Packet Corruption



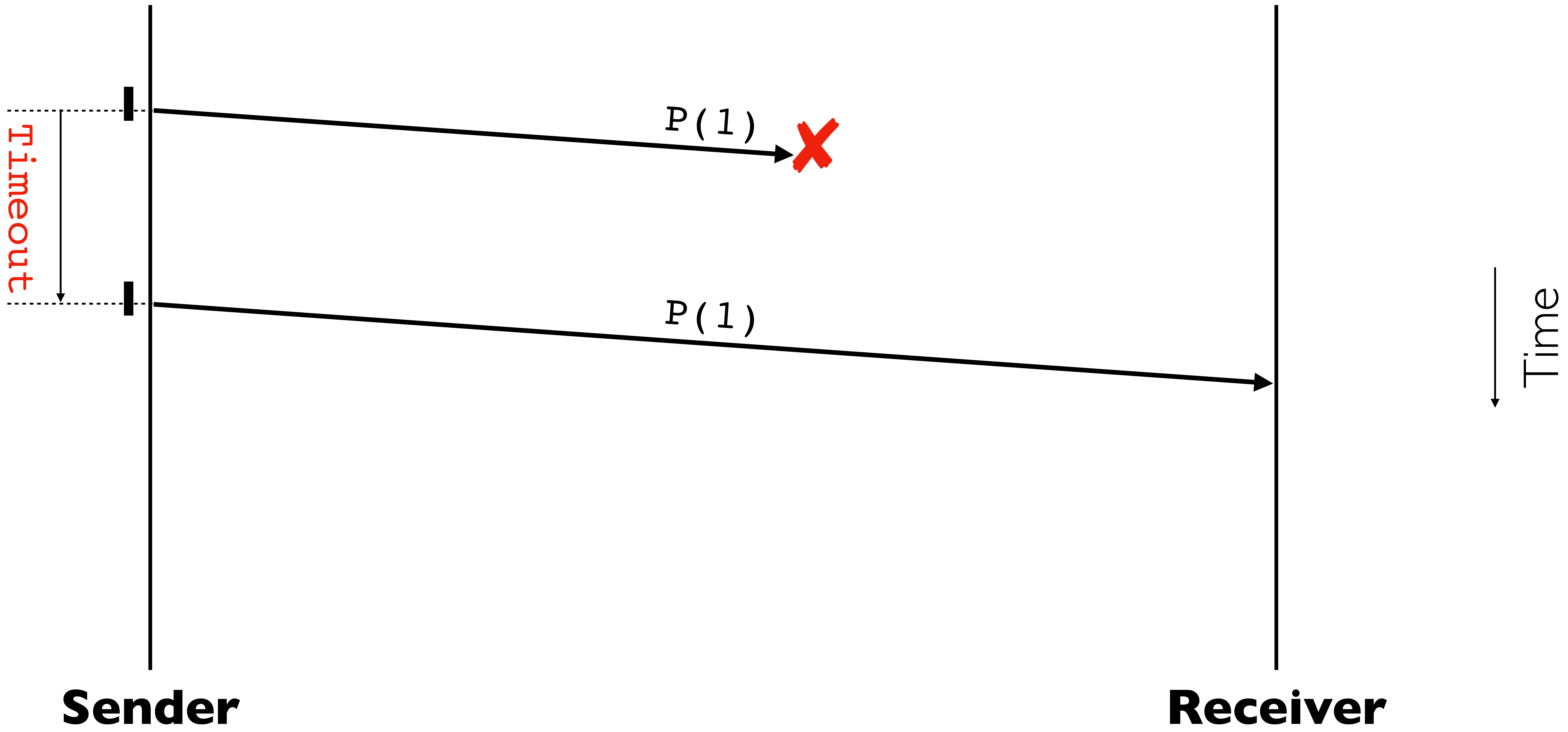
Dealing with Packet Loss



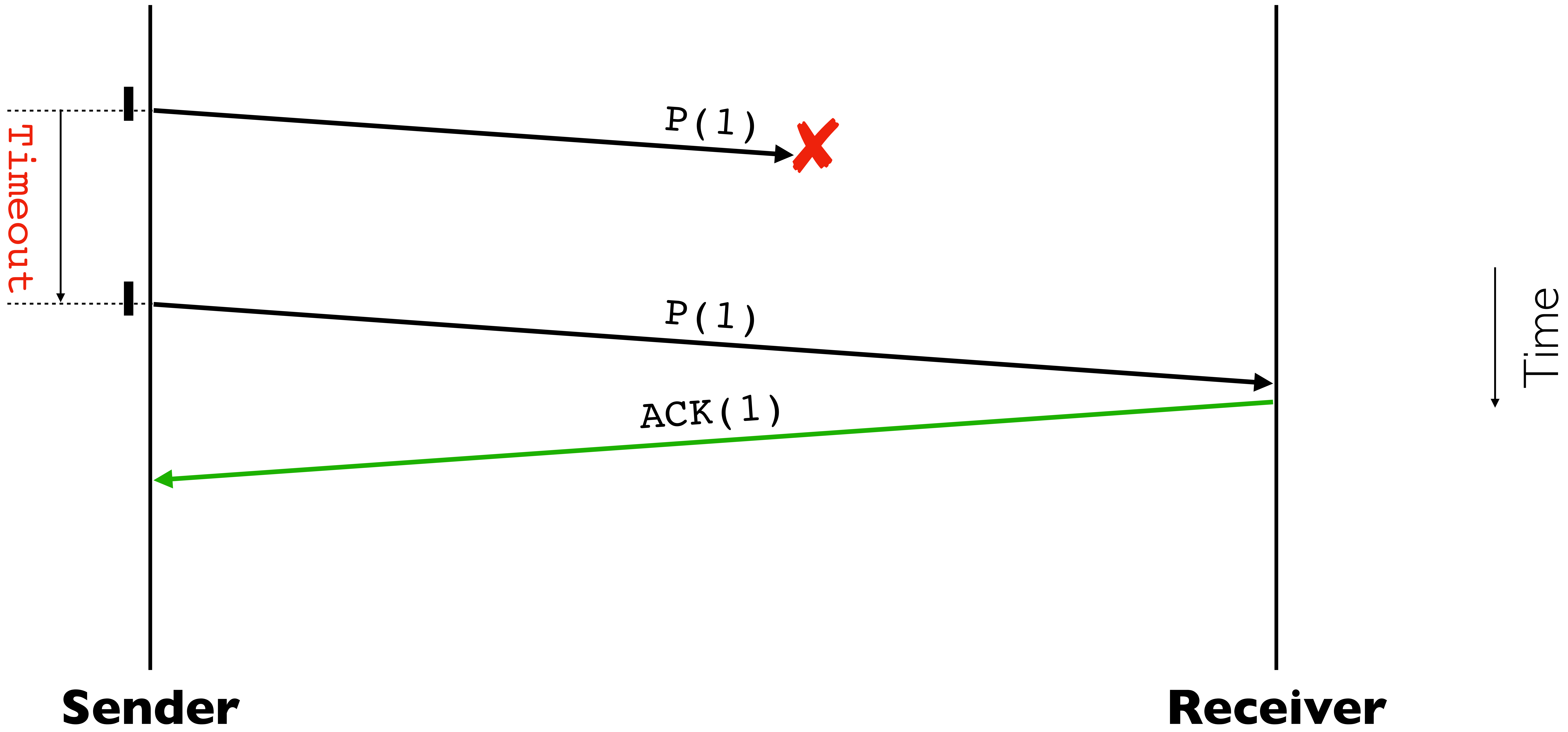
Dealing with Packet Loss



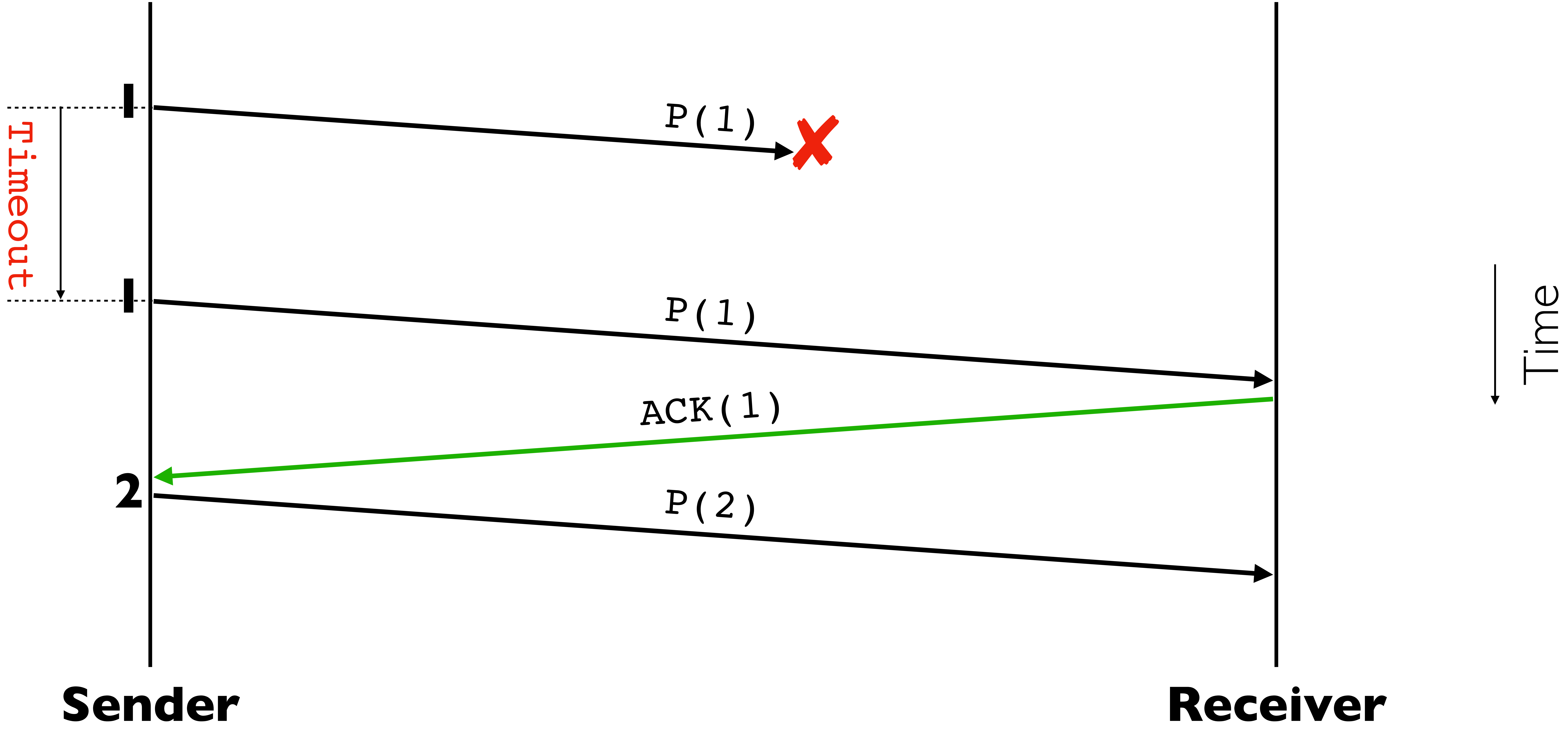
Dealing with Packet Loss



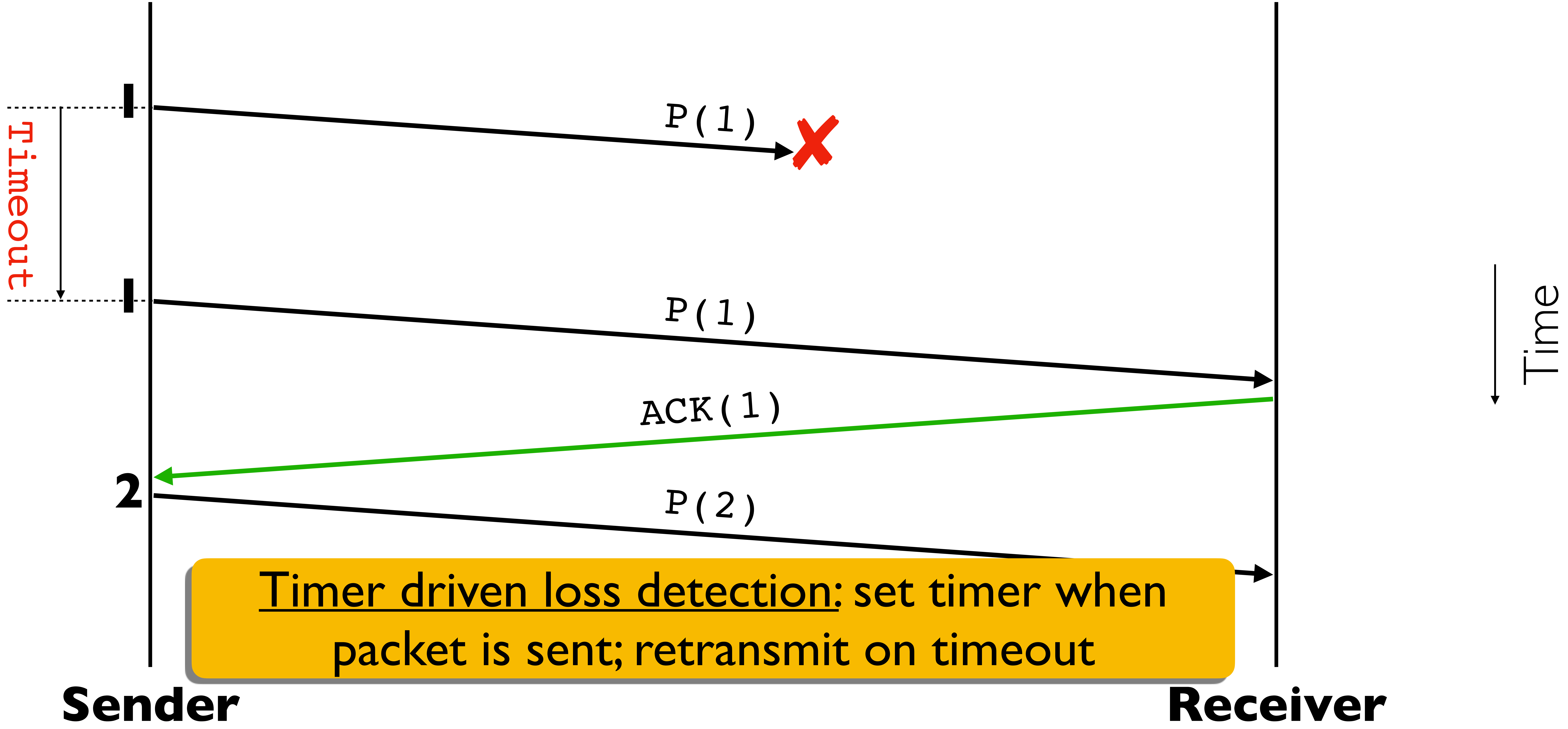
Dealing with Packet Loss



Dealing with Packet Loss

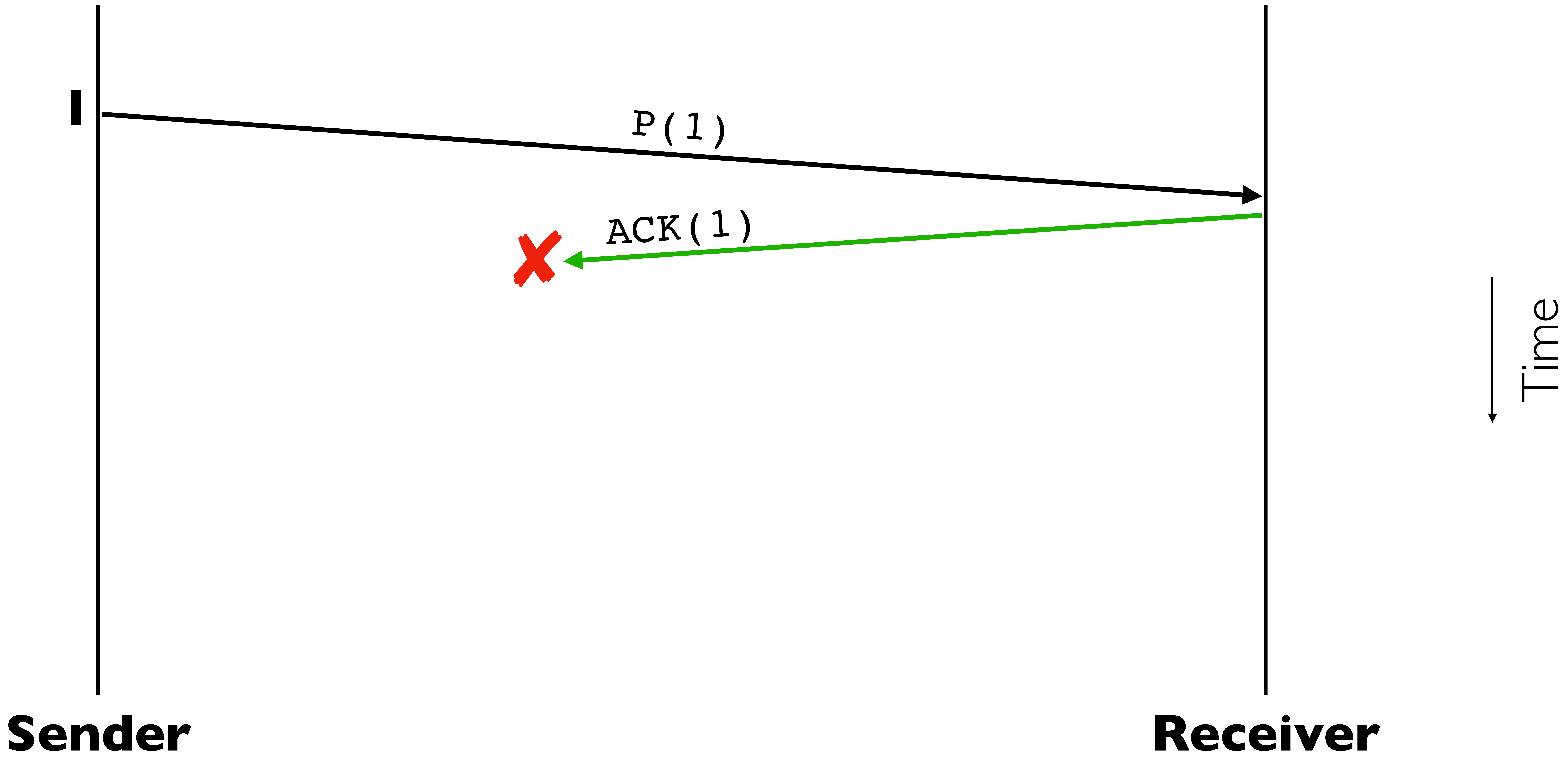


Dealing with Packet Loss

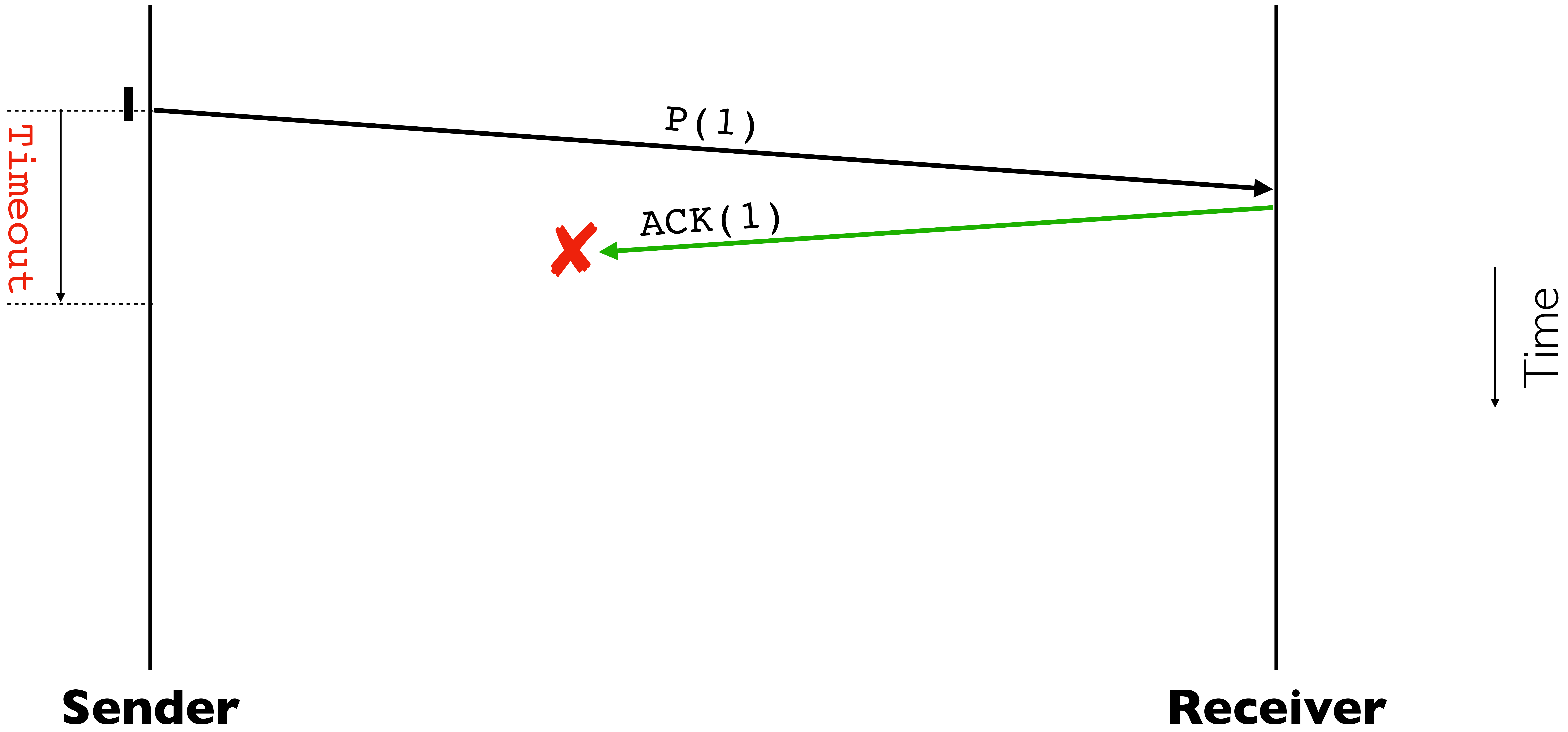


<u>Timer driven loss detection</u>: set timer when packet is sent; retransmit on timeout

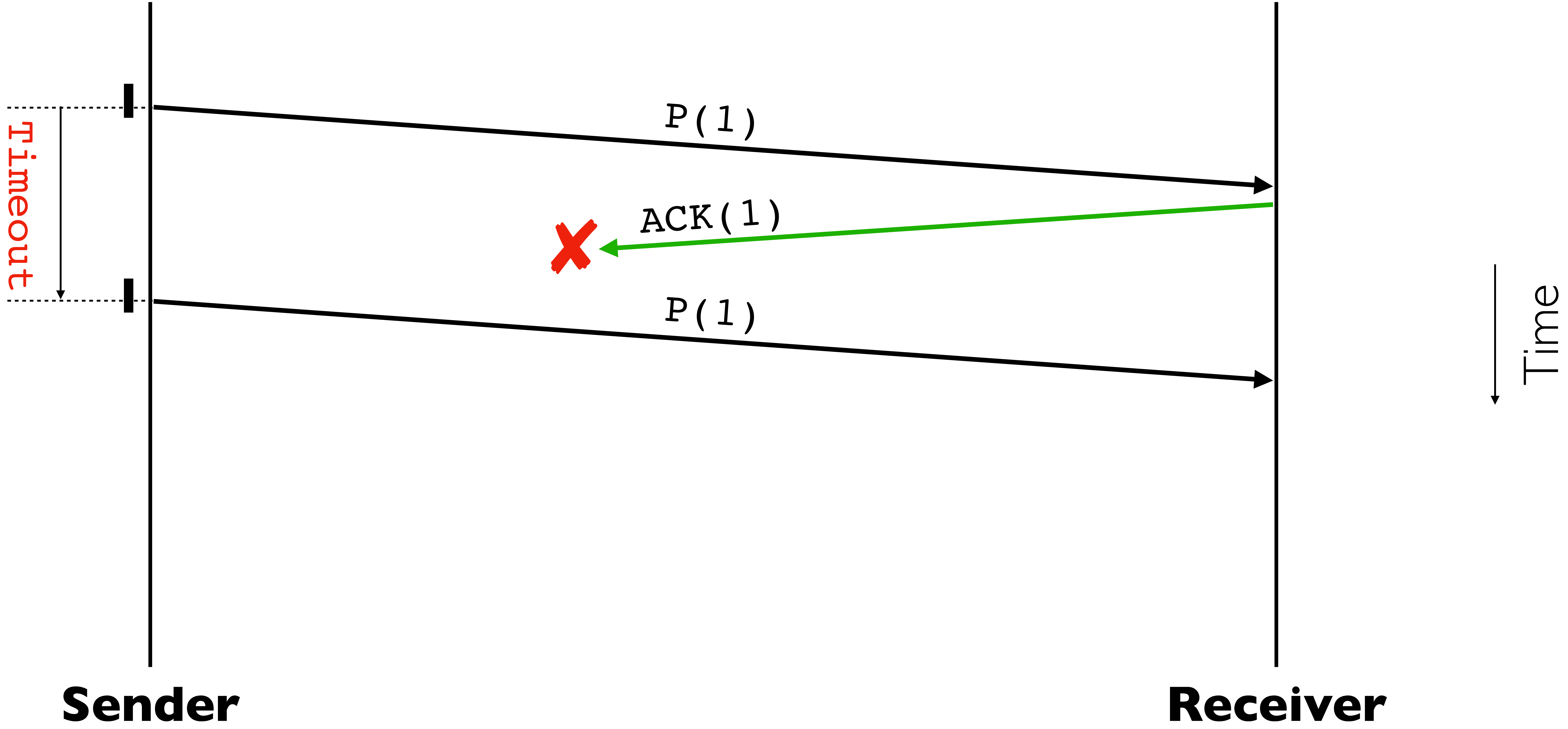
Dealing with Packet Loss (of ACK)



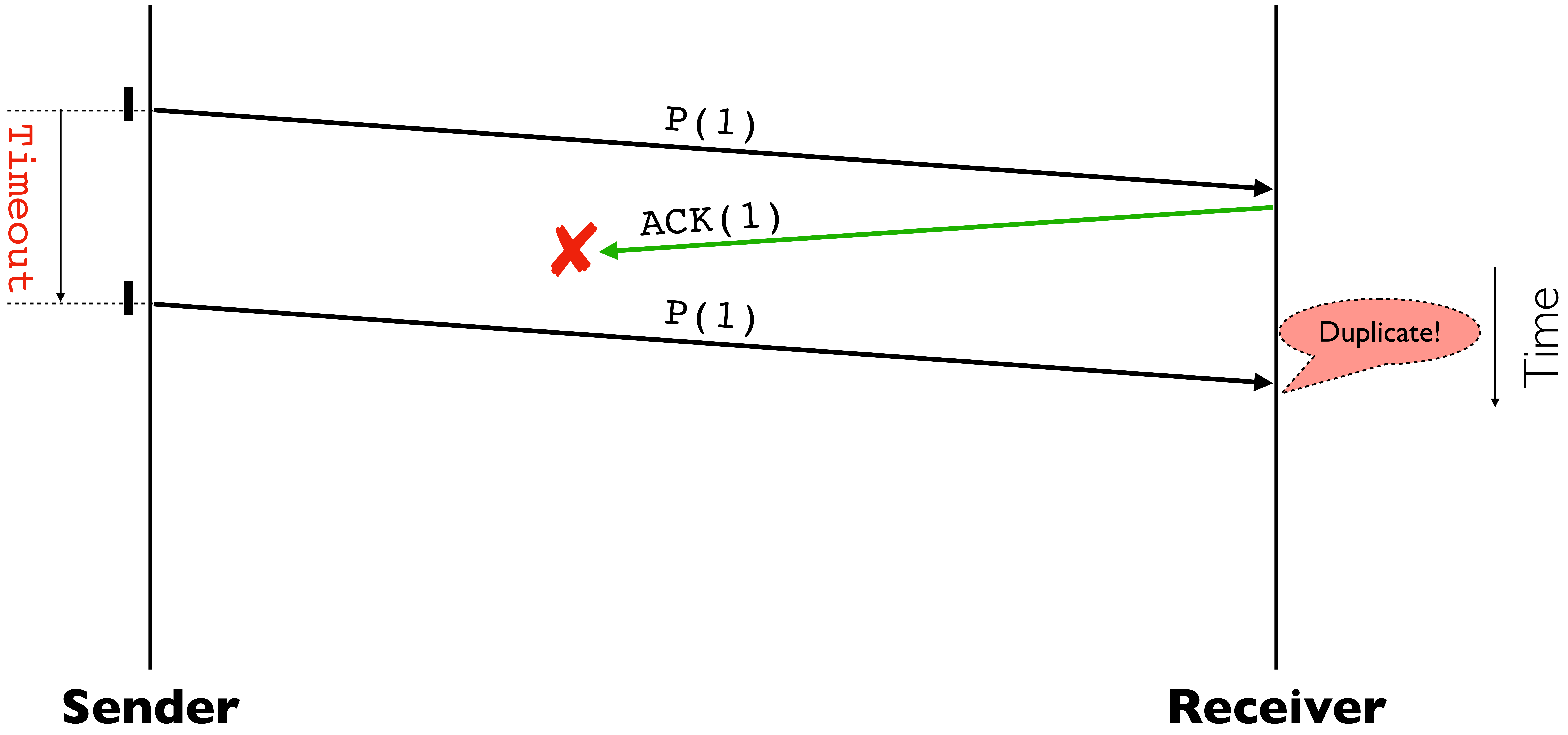
Dealing with Packet Loss (of ACK)



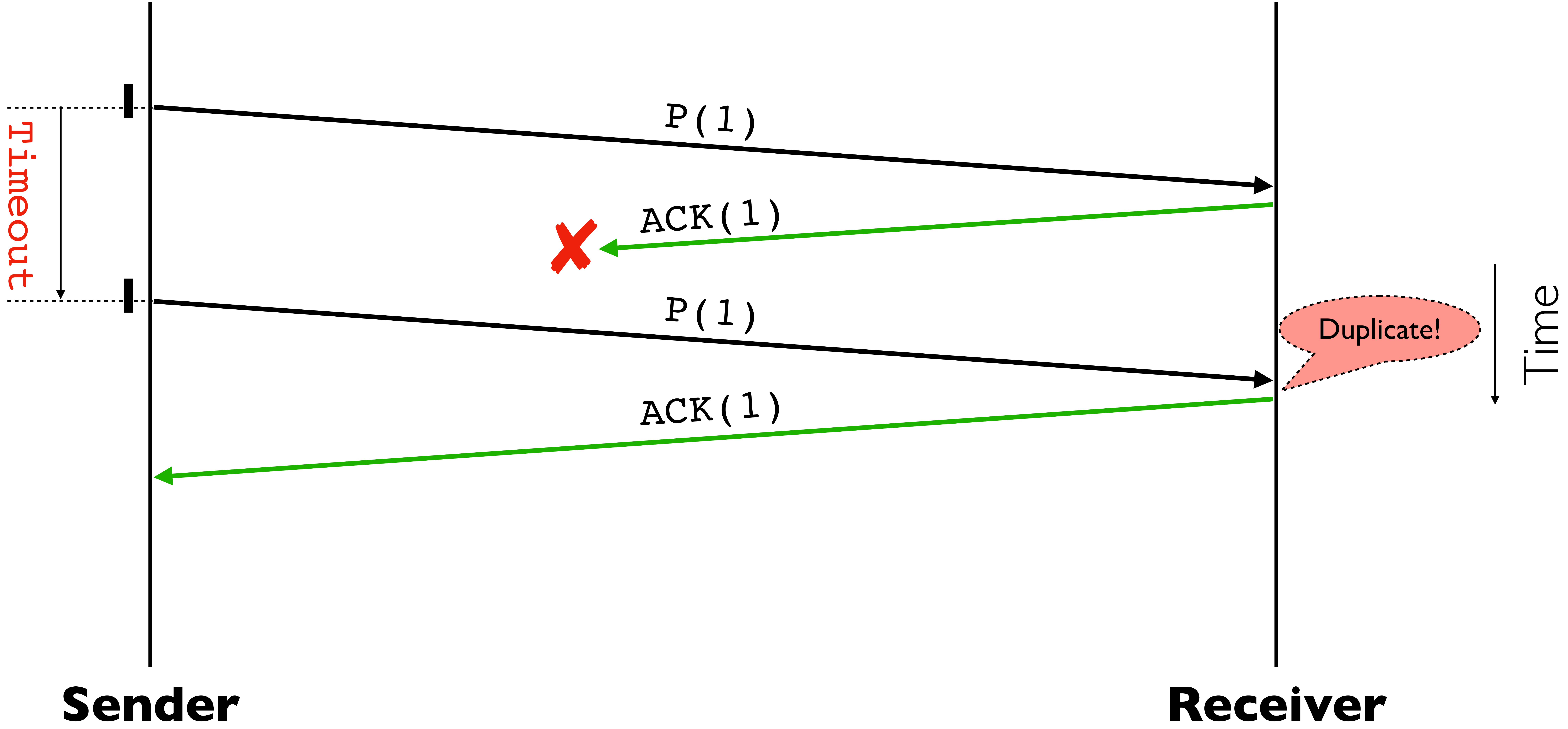
Dealing with Packet Loss (of ACK)



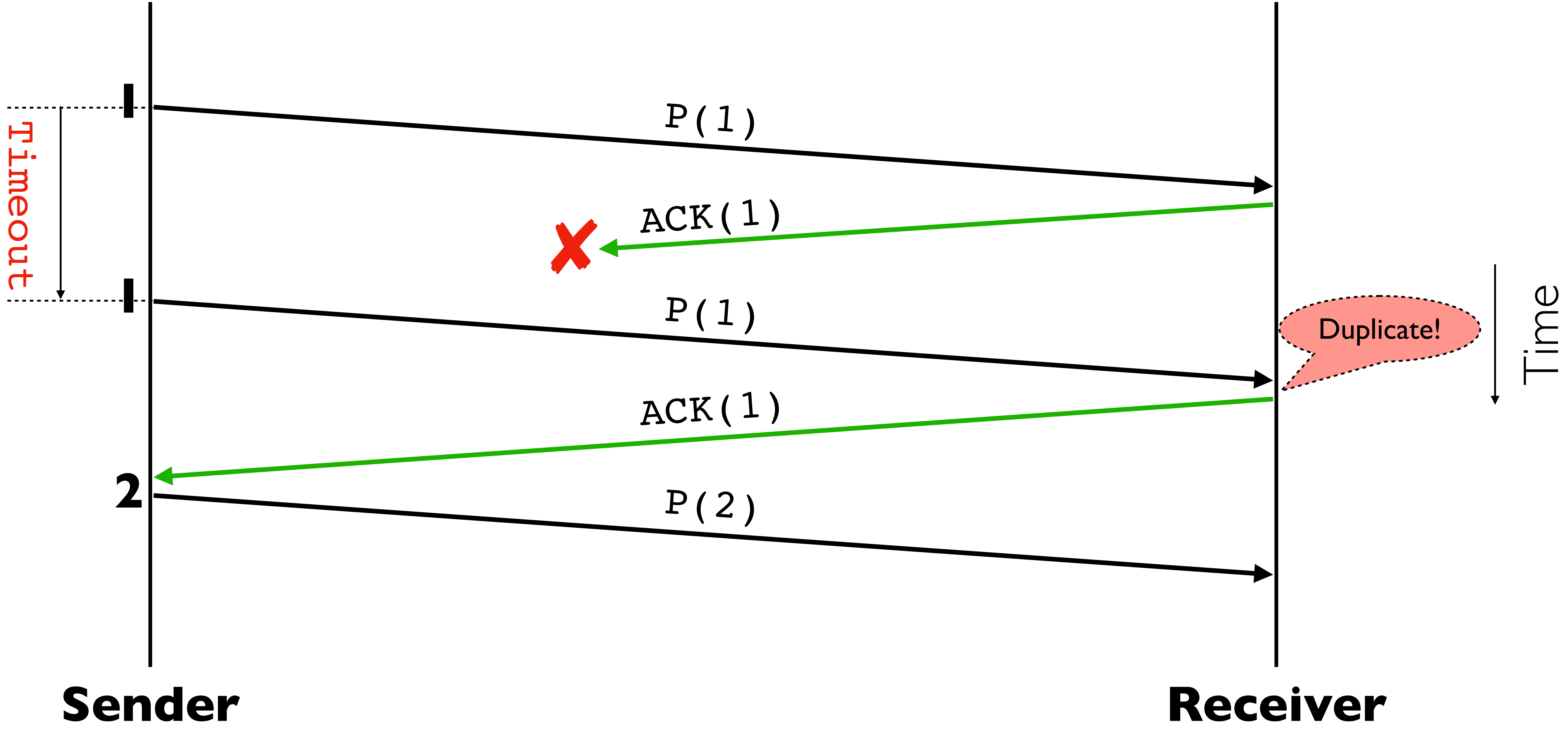
Dealing with Packet Loss (of ACK)



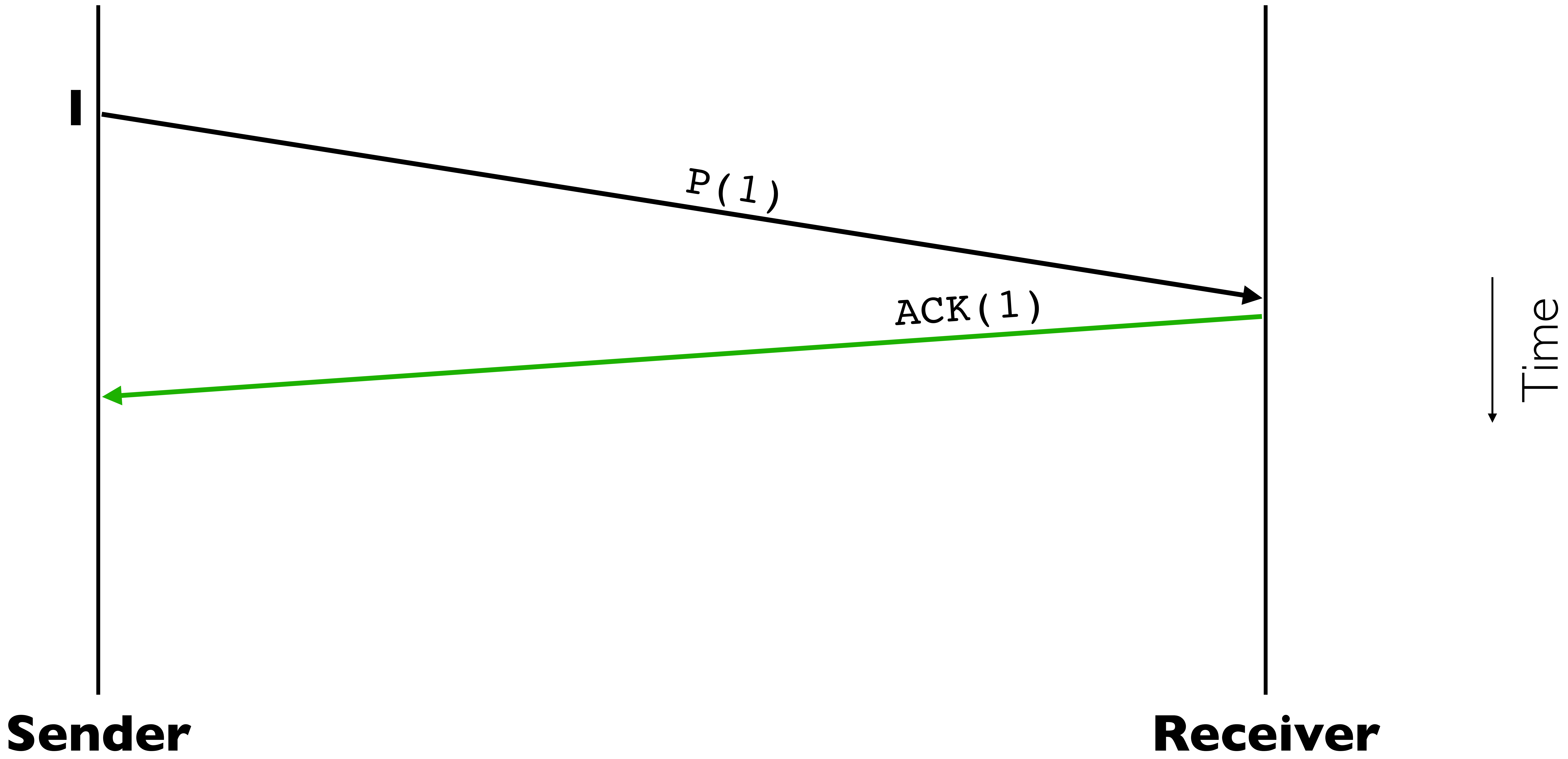
Dealing with Packet Loss (of ACK)



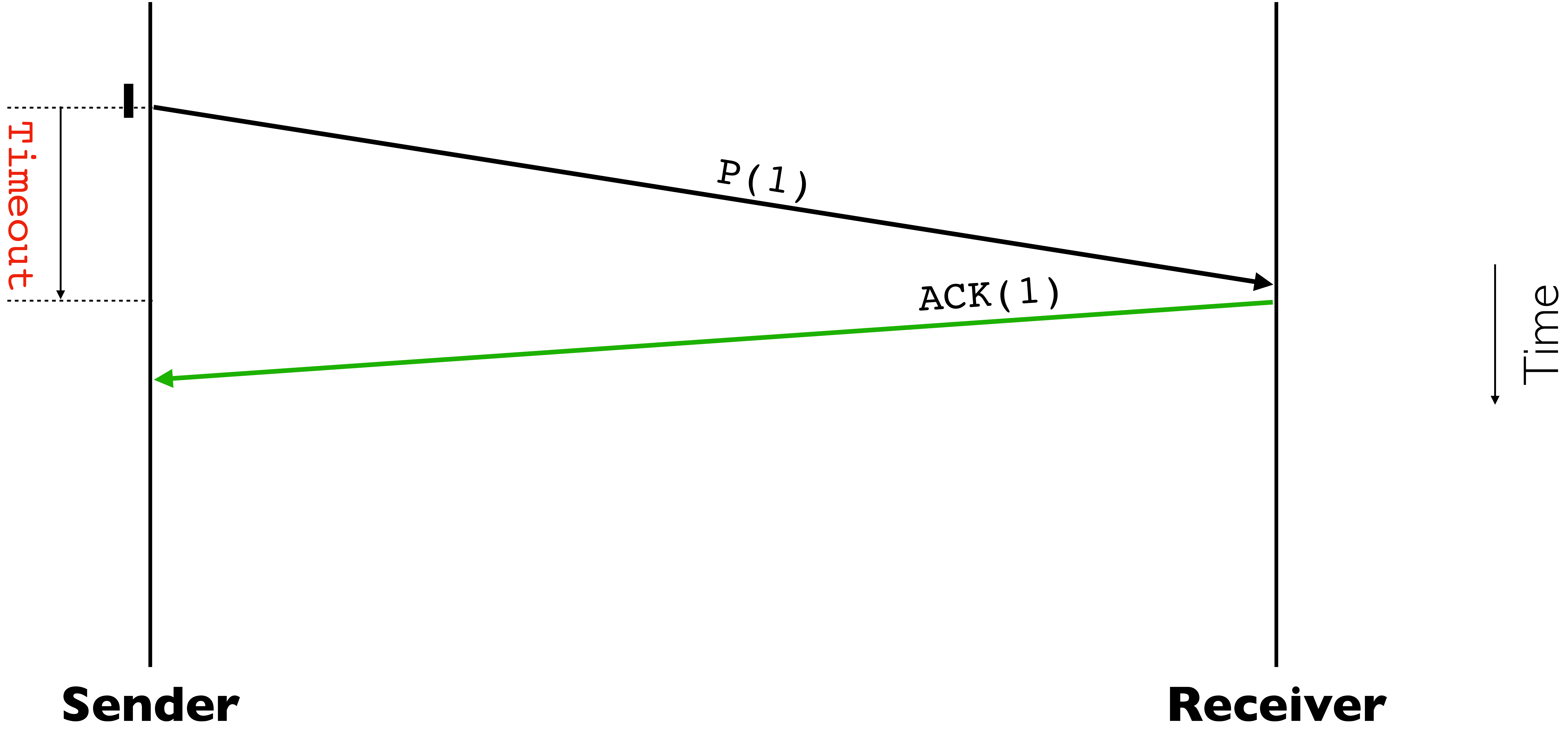
Dealing with Packet Loss (of ACK)



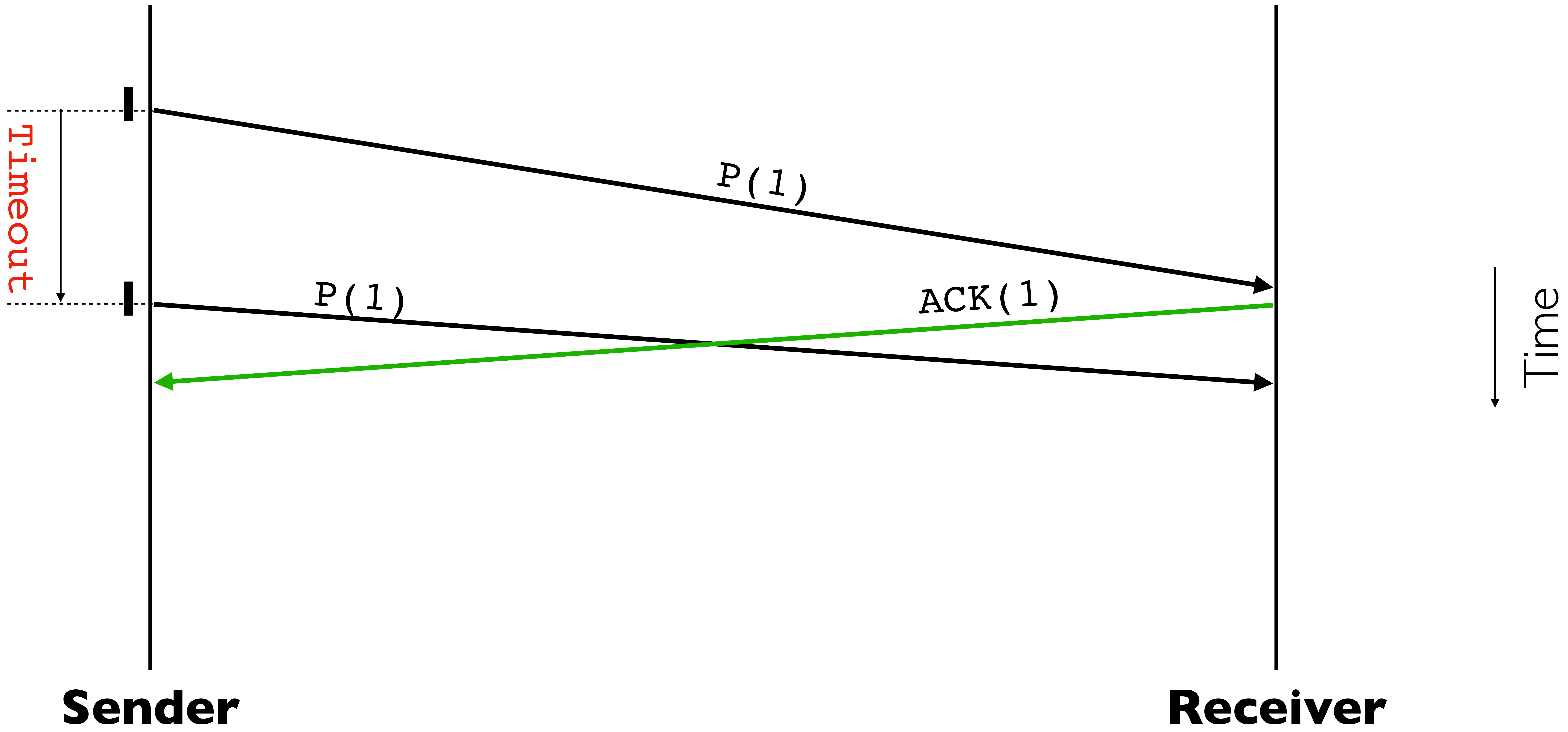
Dealing with Packet Delays



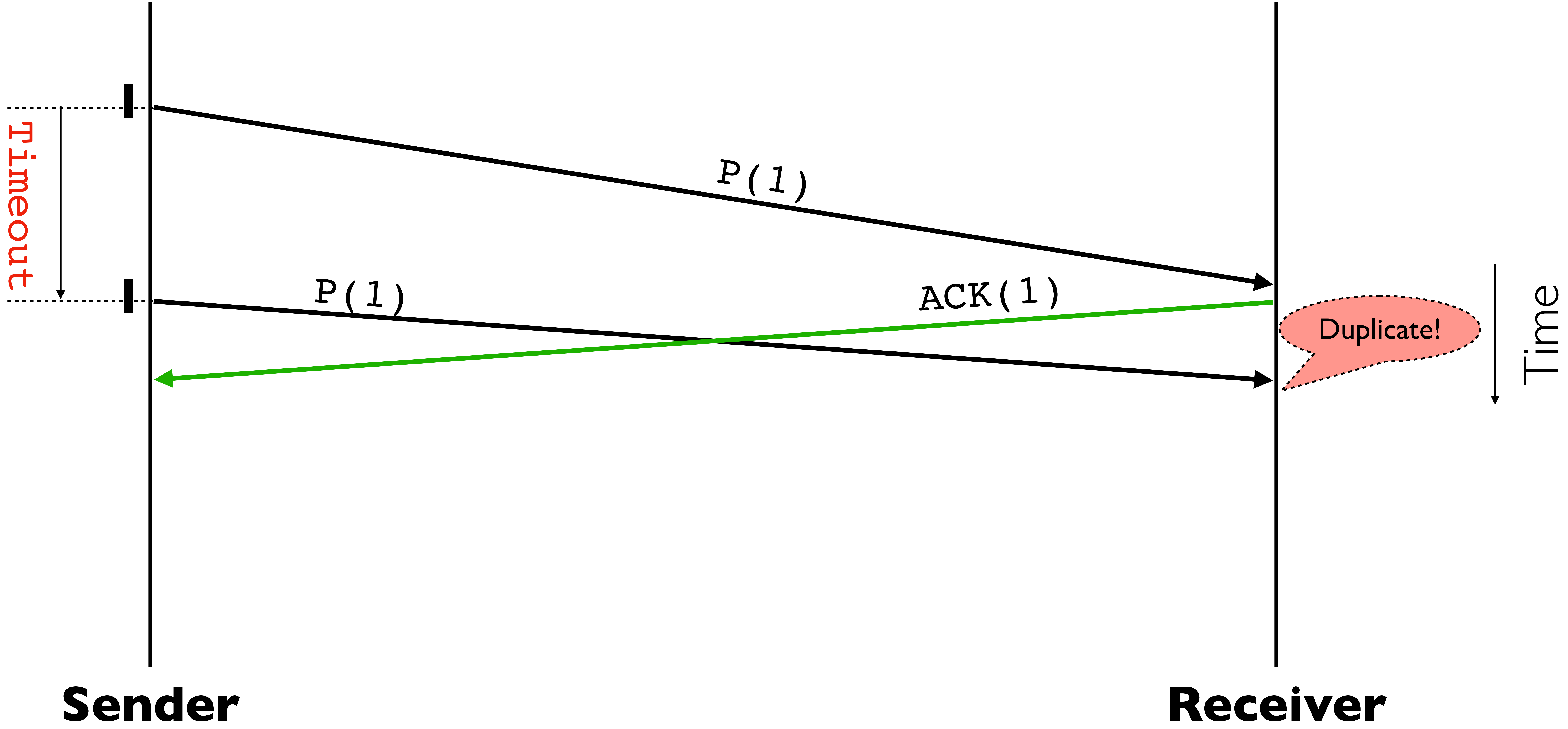
Dealing with Packet Delays



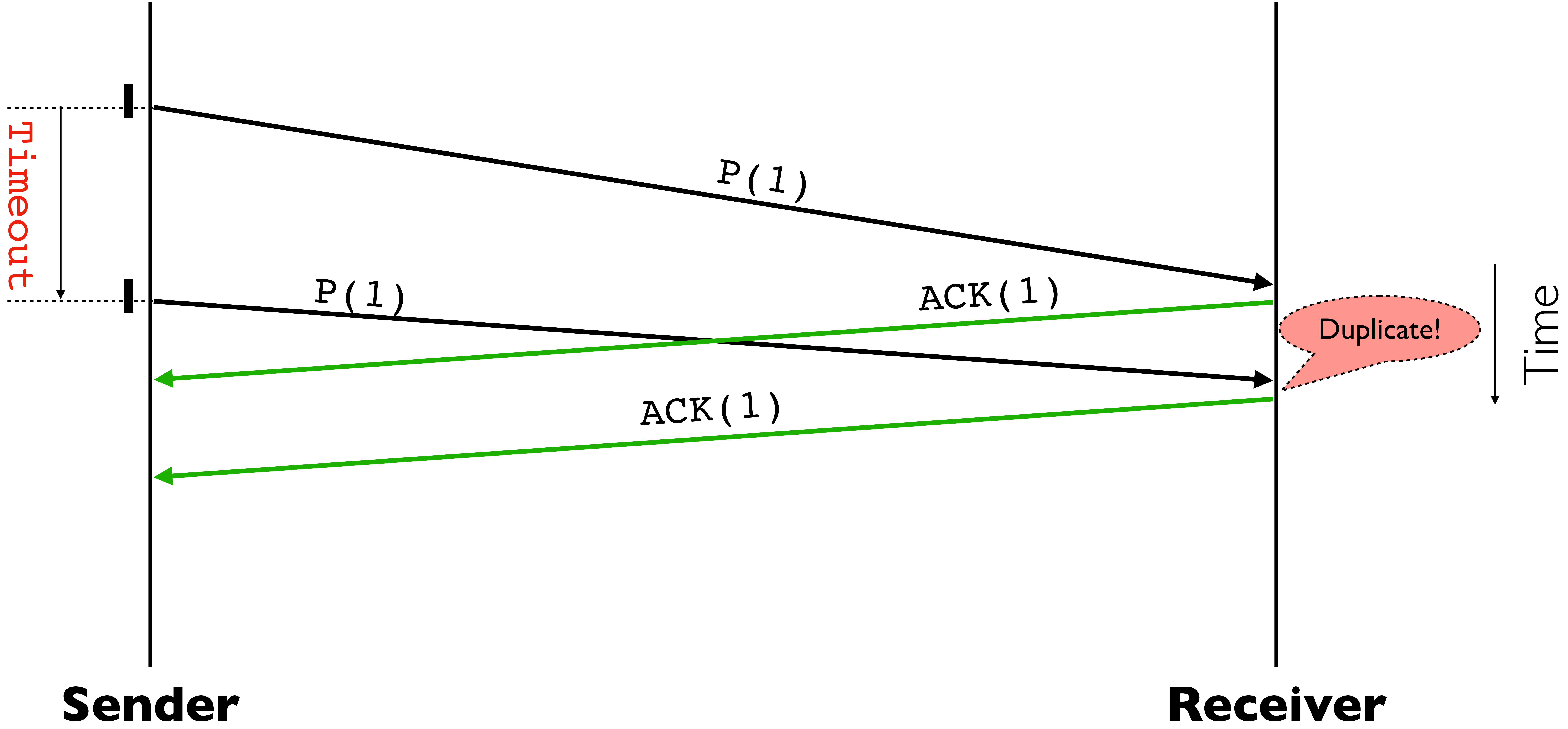
Dealing with Packet Delays



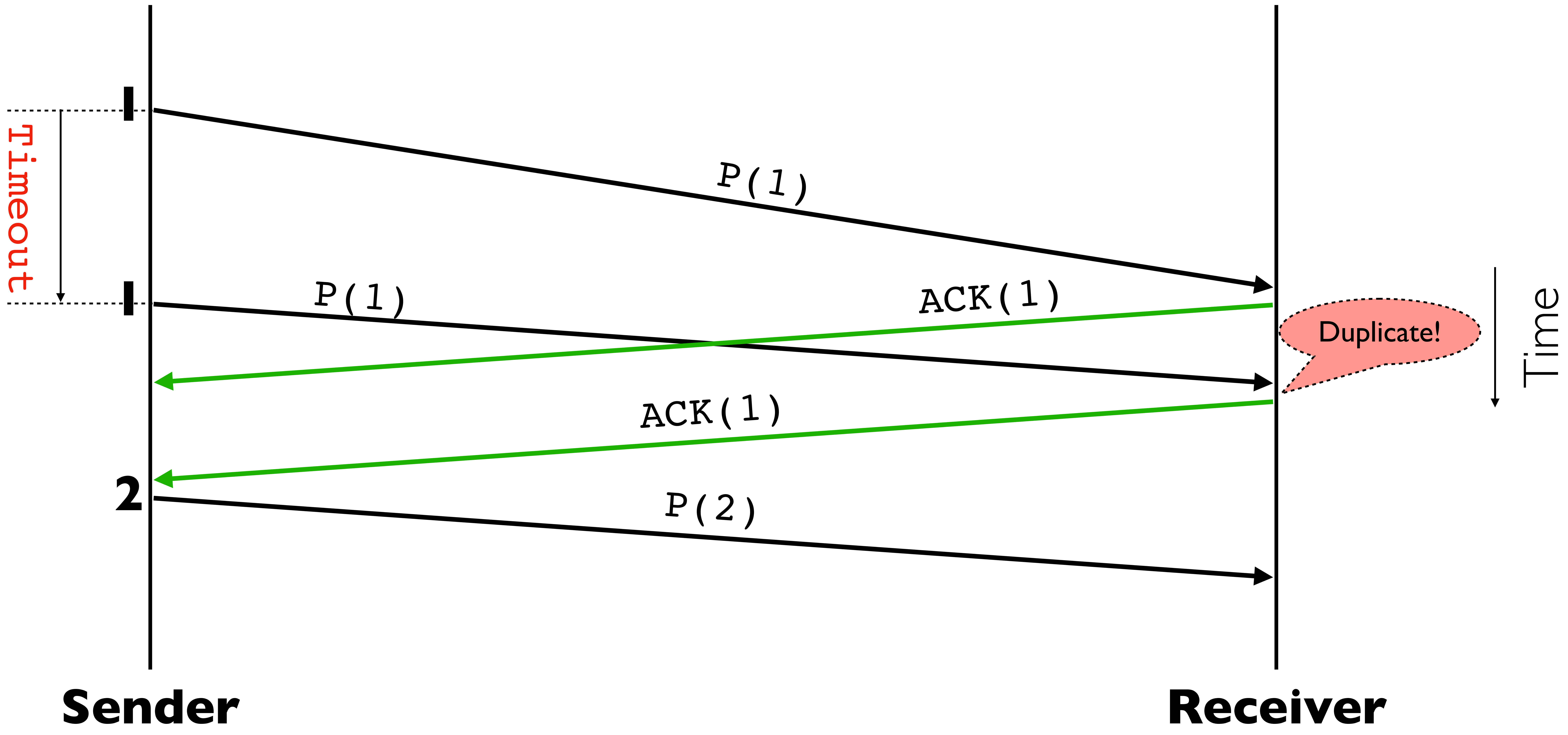
Dealing with Packet Delays



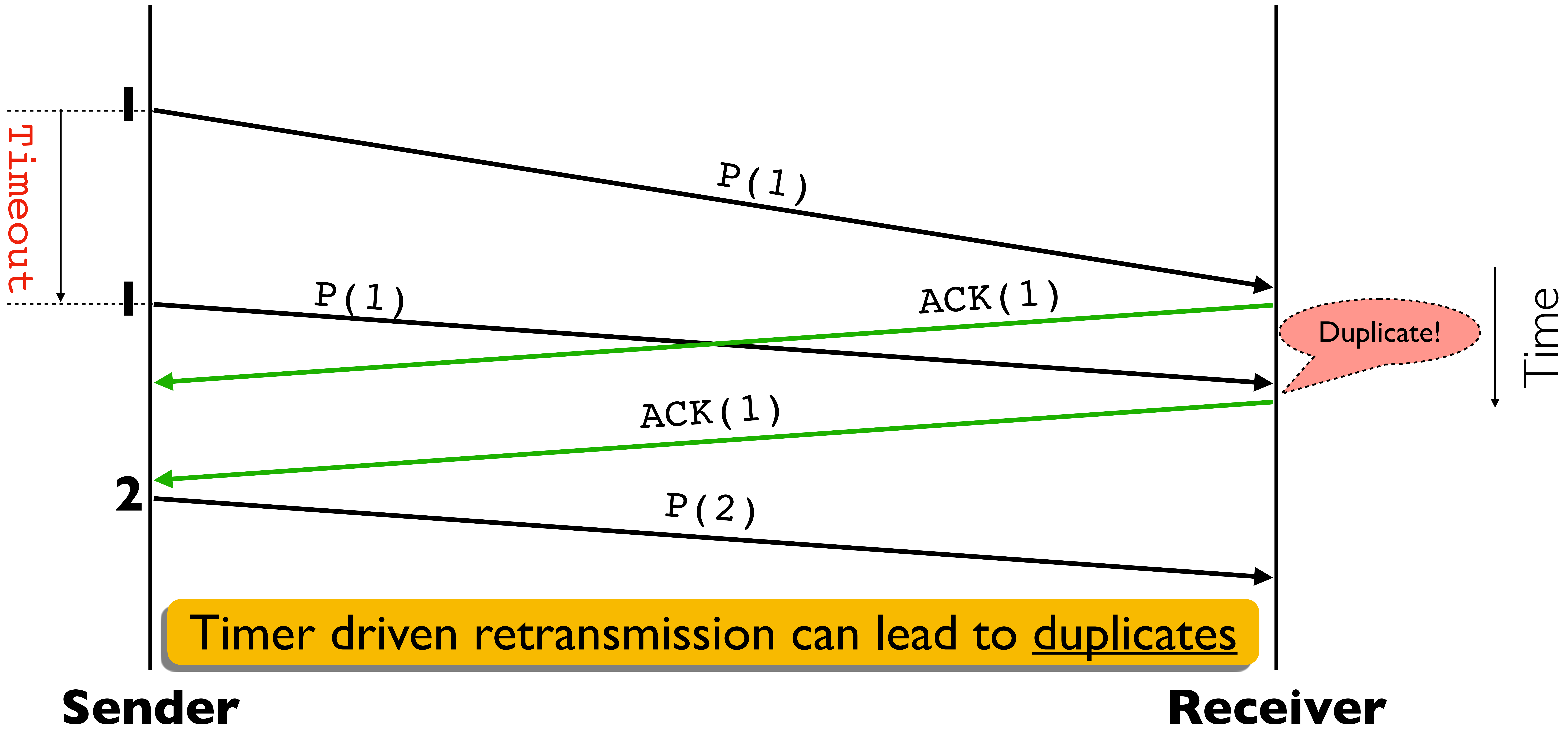
Dealing with Packet Delays



Dealing with Packet Delays



Dealing with Packet Delays



Components of a solution (so far)

- **Checksums:** to detect corruption

Components of a solution (so far)

- **Checksums:** to detect corruption
- **ACKs:** receiver tells sender that it received packet

Components of a solution (so far)

- **Checksums:** to detect corruption
- **ACKs:** receiver tells sender that it received packet
- **NACKs:** receiver tells sender that it did not receive packet

Components of a solution (so far)

- **Checksums:** to detect corruption
- **ACKs:** receiver tells sender that it received packet
- **NACKs:** receiver tells sender that it did not receive packet
- **Sequence numbers:** a way to identify and order packets

Components of a solution (so far)

- **Checksums:** to detect corruption
- **ACKs:** receiver tells sender that it received packet
- **NACKs:** receiver tells sender that it did not receive packet
- **Sequence numbers:** a way to identify and order packets
- **Retransmissions:** sender resends packets

Components of a solution (so far)

- **Checksums:** to detect corruption
- **ACKs:** receiver tells sender that it received packet
- **NACKs:** receiver tells sender that it did not receive packet
- **Sequence numbers:** a way to identify and order packets
- **Retransmissions:** sender resends packets
- **Timeouts:** a way of deciding when to resend a packet

Components of a solution (so far)

- **Checksums:** to detect corruption
- **ACKs:** receiver tells sender that it received packet
- **NACKs:** receiver tells sender that it did not receive packet
- **Sequence numbers:** a way to identify and order packets
- **Retransmissions:** sender resends packets
- **Timeouts:** a way of deciding when to resend a packet

- **But we have not put together into a coherent design…**

Designing Reliable Transport

A solution: "Stop and Wait"

@Sender

- Send packet(i); (re)set timer; wait for ack
- If (ACK)
  - i++; repeat
- If (NACK or TIMEOUT)
  - repeat

@Receiver

- Wait for packet(i)
- If (packet is OK)
  - Send ACK(i)
- Else
  - Send NACK(i)
- repeat

A solution: “Stop and Wait”

@Sender
- Send packet(i); (re)set timer; wait for ack
- If (ACK)
  - i++; repeat
- If (NACK or TIMEOUT)
  - repeat

@Receiver
- Wait for packet(i)
- If (packet is OK)
  - Send ACK(i)
- Else
  - Send NACK(i)
- repeat

- **We have a correct reliable protocol!**

A solution: "Stop and Wait"

@Sender
- Send packet(i); (re)set timer; wait for ack
- If (ACK)
  - i++; repeat
- If (NACK or TIMEOUT)
  - repeat

@Receiver
- Wait for packet(i)
- If (packet is OK)
  - Send ACK(i)
- Else
  - Send NACK(i)
- repeat

- **We have a correct reliable protocol!**
- **Probably the world's most inefficient one…** ***why?***

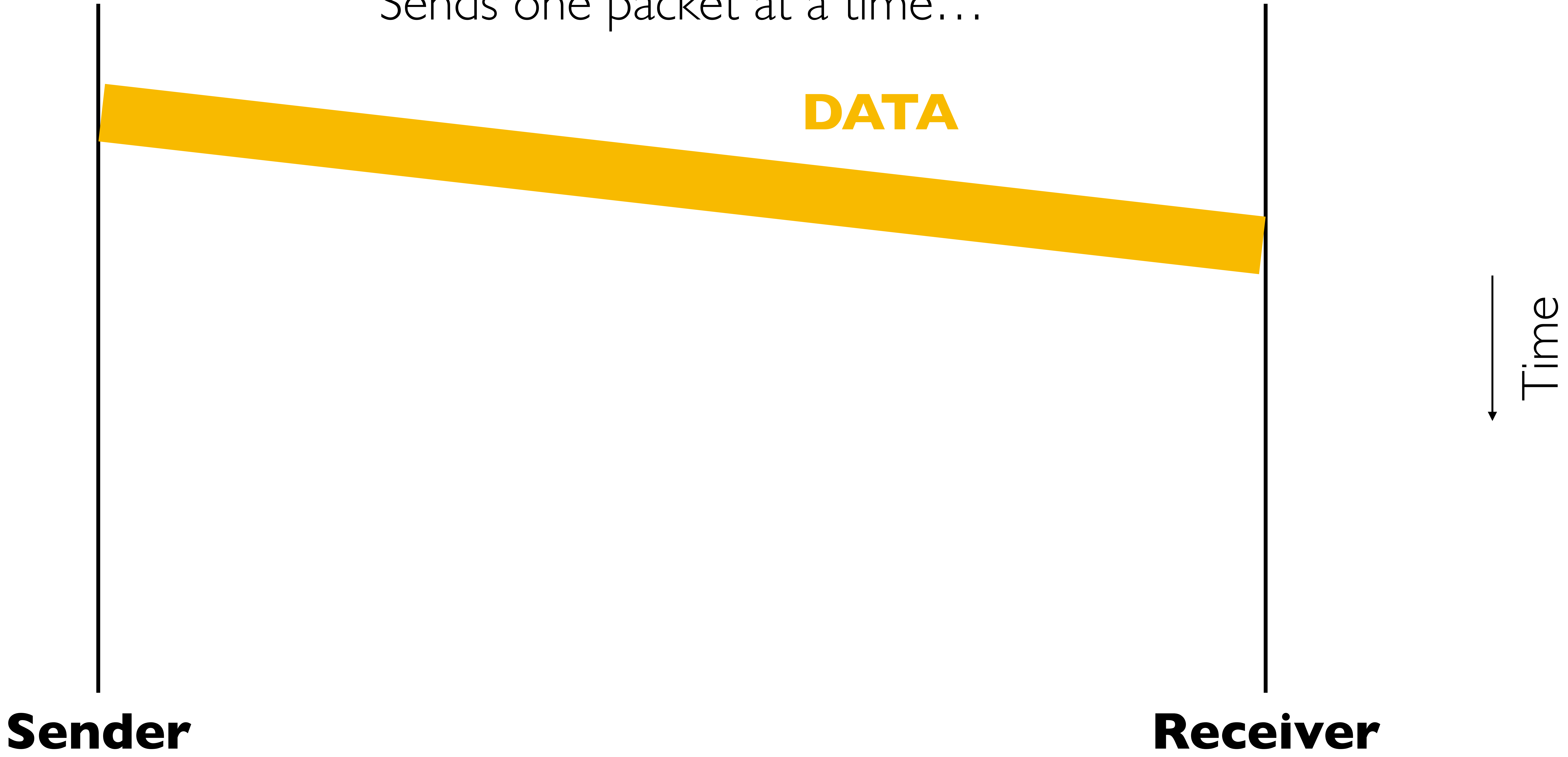
Stop & Wait is Inefficient

Stop & Wait is Inefficient

Sends one packet at a time…

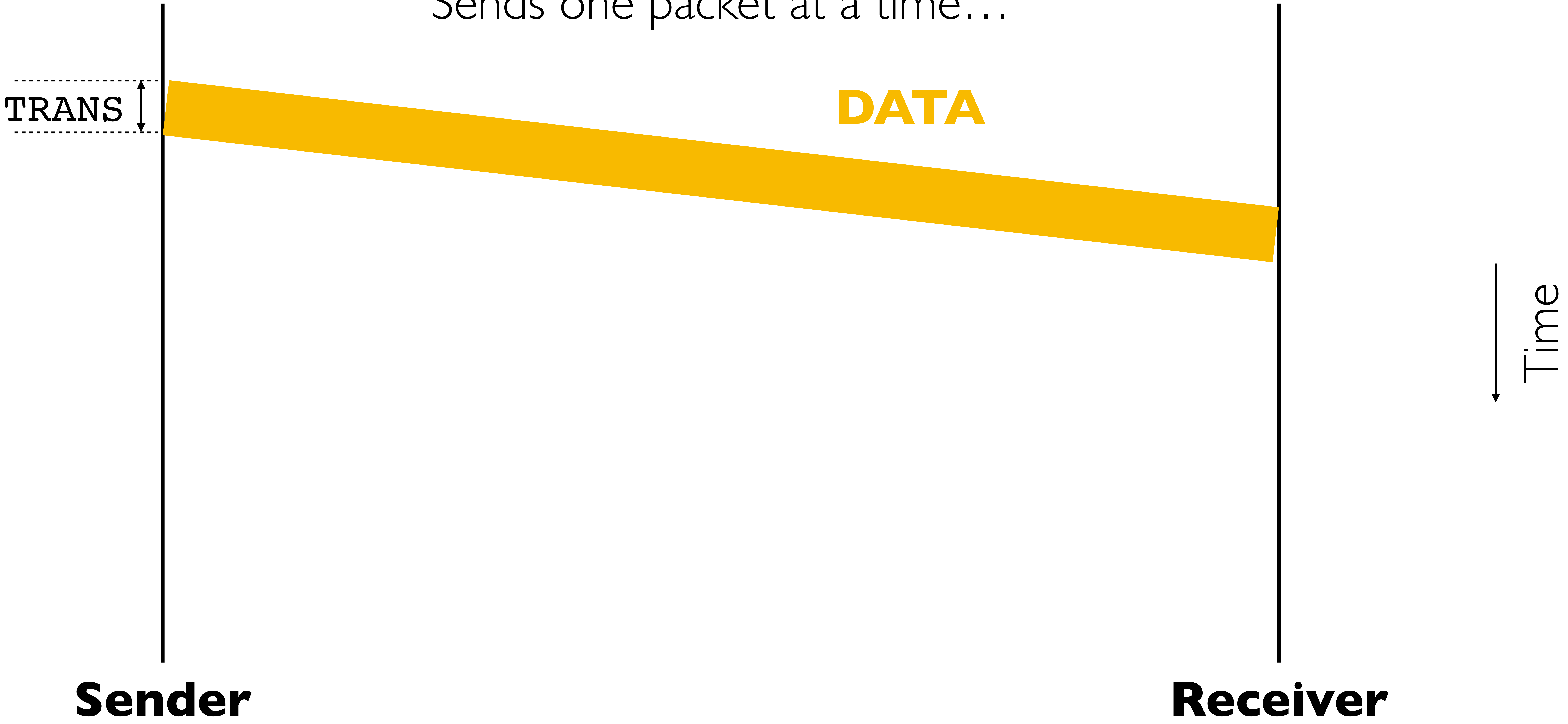
Stop & Wait is Inefficient

Sends one packet at a time…



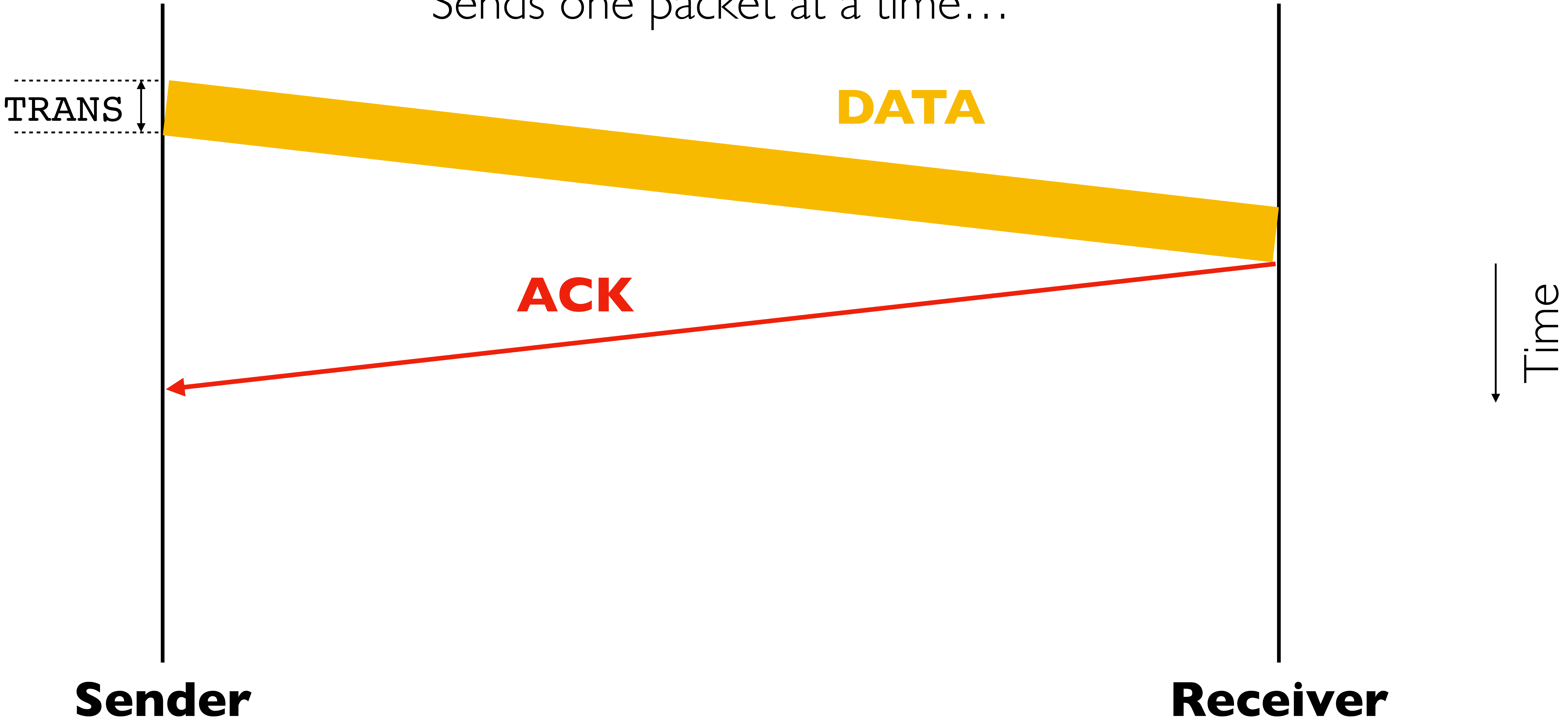
Stop & Wait is Inefficient

Sends one packet at a time…



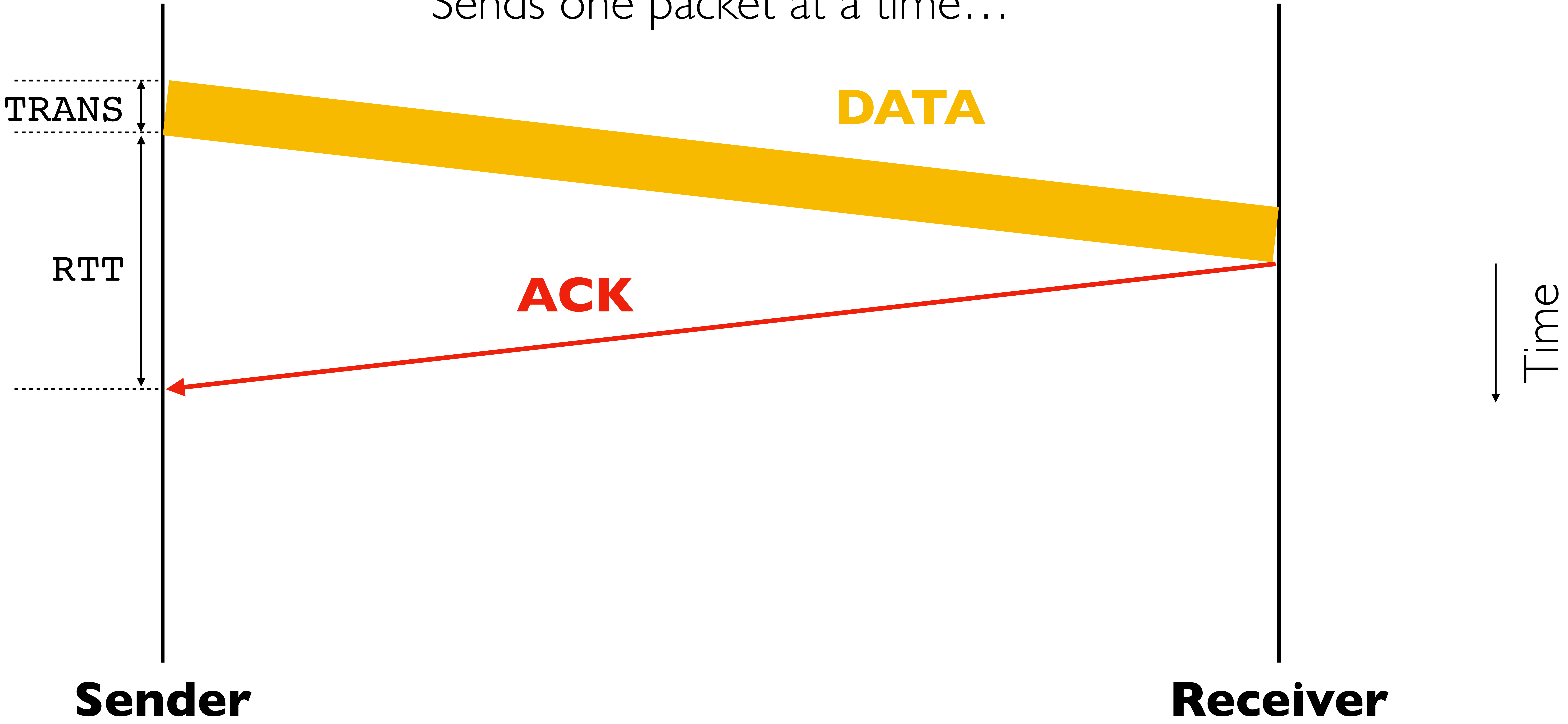
Stop & Wait is Inefficient

Sends one packet at a time…



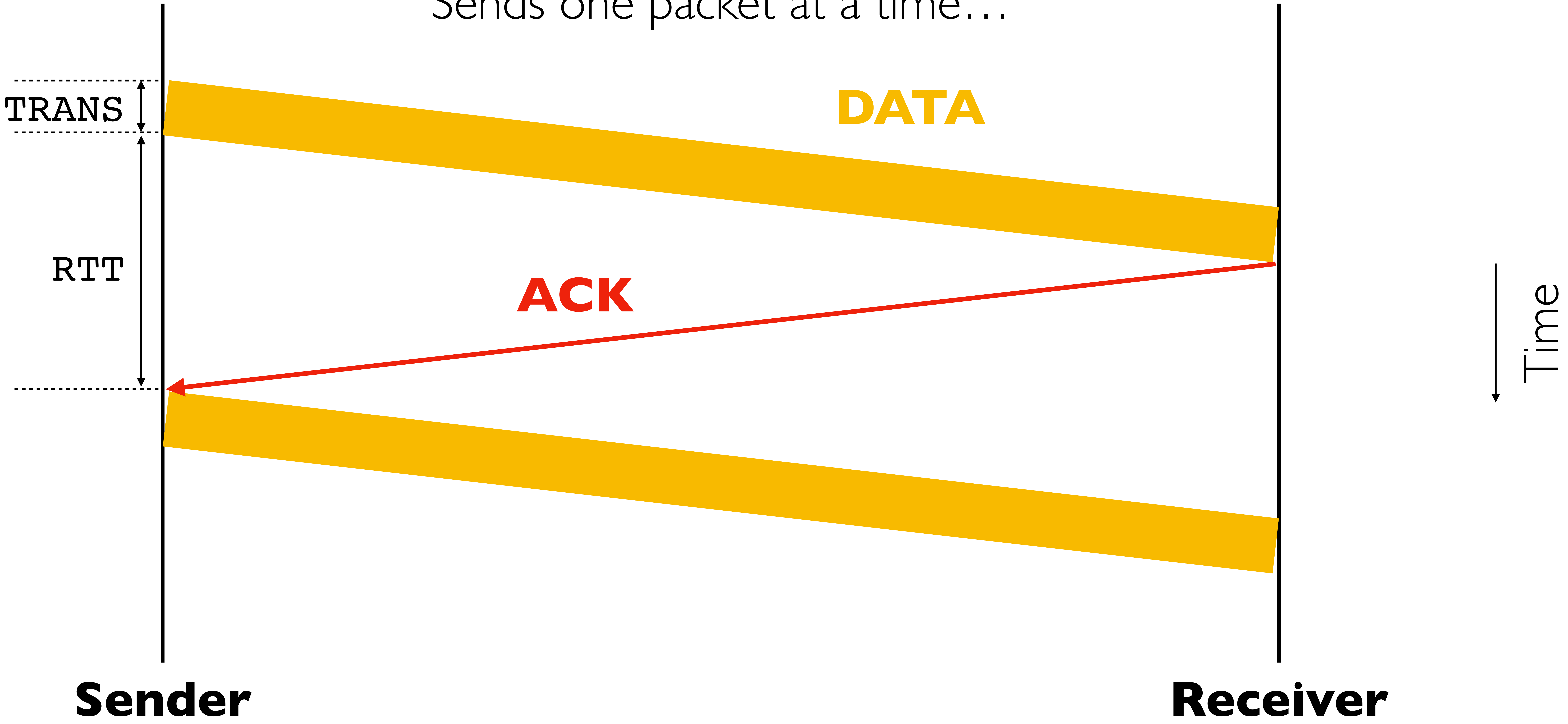
Stop & Wait is Inefficient

Sends one packet at a time…



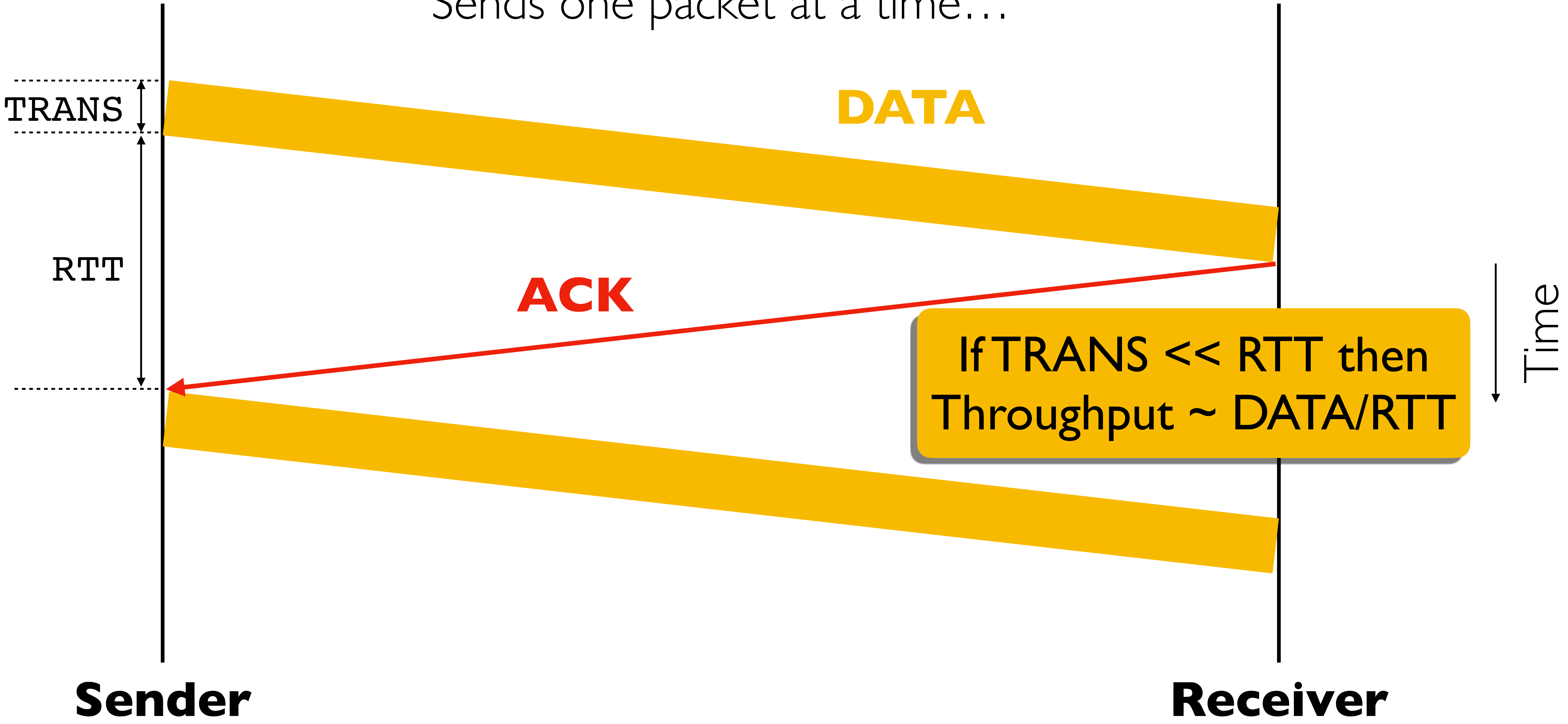
Stop & Wait is Inefficient

Sends one packet at a time…



Stop & Wait is Inefficient

Sends one packet at a time…



Orders of Magnitude

- **Transmission time for 10Gbps link**

Orders of Magnitude

- **Transmission time for 10Gbps link**
  - ~microsecond for 1500 byte packet

Orders of Magnitude

- **Transmission time for 10Gbps link**
  - ~microsecond for 1500 byte packet
- **RTT:**

Orders of Magnitude

- **Transmission time for 10Gbps link**
  - ~microsecond for 1500 byte packet
- **RTT:**
  - 1000 km ~ O(10) milliseconds!

Orders of Magnitude

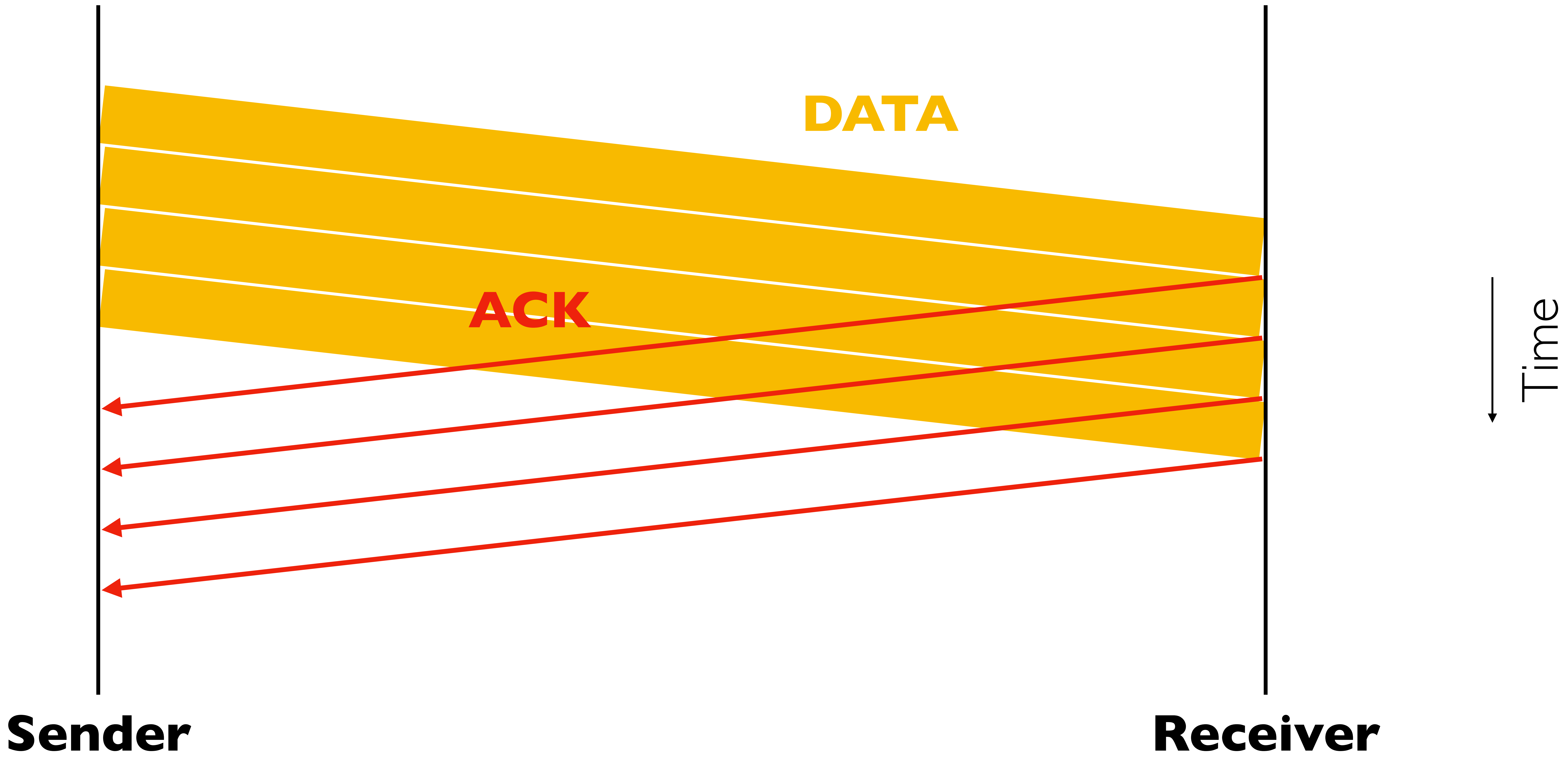
- **Transmission time for 10Gbps link**
  - ~microsecond for 1500 byte packet
- **RTT:**
  - 1000 km ~ O(10) milliseconds!
- **Throughput:**

Orders of Magnitude

- **Transmission time for 10Gbps link**
  - ~microsecond for 1500 byte packet
- **RTT:**
  - 1000 km ~ O(10) milliseconds!
- **Throughput:**
  - 1500 byte / 10 ms = 1.5Mbps!

How to make it more efficient?

How to make it more efficient?



Three Design Decisions

- **Which packets can sender send?**

Three Design Decisions

- **Which packets can sender send?**
  - Sliding window

Three Design Decisions

- **Which packets can sender send?**
  - Sliding window
- **How does received ACK packets?**

Three Design Decisions

- **Which packets can sender send?**
  - Sliding window
- **How does received ACK packets?**
  - Cumulative

Three Design Decisions

- **Which packets can sender send?**
  - Sliding window

- **How does received ACK packets?**
  - Cumulative
  - Selective

Three Design Decisions

- **Which packets can sender send?**
  - Sliding window
- **How does received ACK packets?**
  - Cumulative
  - Selective
- **Which packets does sender resend?**

Three Design Decisions

- **Which packets can sender send?**
  - Sliding window

- **How does received ACK packets?**
  - Cumulative
  - Selective

- **Which packets does sender resend?**
  - GBN

Three Design Decisions

- **Which packets can sender send?**
  - Sliding window
- **How does received ACK packets?**
  - Cumulative
  - Selective
- **Which packets does sender resend?**
  - GBN
  - Selective repeat

Sliding Window

- **Window = set of adjacent sequence numbers**

Sliding Window

- **Window = set of adjacent sequence numbers**
  - The size of the set is the *window size*; assume window size is *n*

Sliding Window

- **Window = set of adjacent sequence numbers**
  - The size of the set is the *window size*; assume window size is *n*

- **General Idea: send up to *n* packets at a time**

Sliding Window

- **Window = set of adjacent sequence numbers**
  - The size of the set is the *window size*; assume window size is *n*
- **General Idea: send up to *n* packets at a time**
  - Sender can send packets in its window

Sliding Window

- **Window = set of adjacent sequence numbers**
  - The size of the set is the *window size*; assume window size is *n*
- **General Idea: send up to *n* packets at a time**
  - Sender can send packets in its window
  - Receiver can accept packets in its window

Sliding Window

- **Window = set of adjacent sequence numbers**
  - The size of the set is the *window size*; assume window size is *n*
- **General Idea: send up to *n* packets at a time**
  - Sender can send packets in its window
  - Receiver can accept packets in its window
  - Window of acceptable packets *slides* on successful reception/acknowledgement

Sliding Window

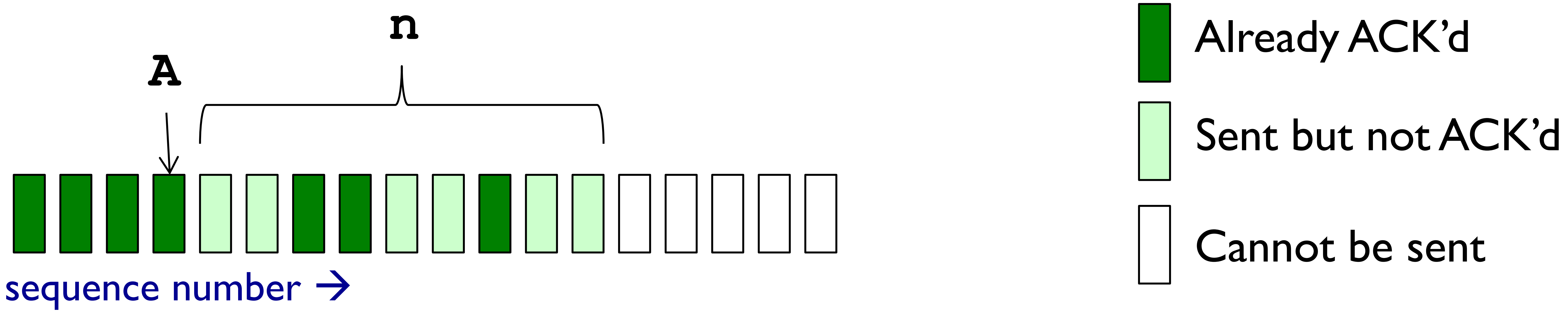
- **Window = set of adjacent sequence numbers**
  - The size of the set is the *window size*; assume window size is *n*

- **General Idea: send up to *n* packets at a time**
  - Sender can send packets in its window
  - Receiver can accept packets in its window
  - Window of acceptable packets *slides* on successful reception/acknowledgement

- **Sliding window often called "packets in flight"**

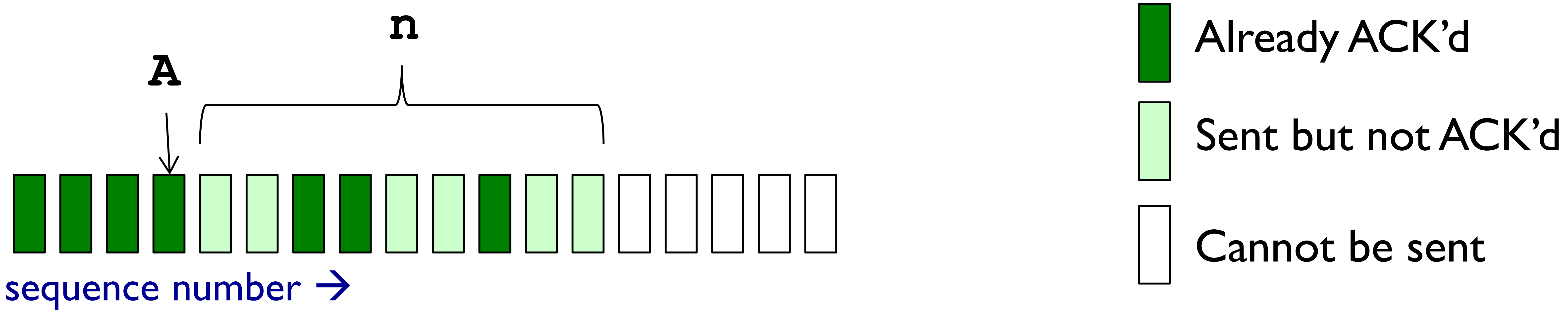
Sliding Window

- Let A be the **last ACK'd packet of sender without gap**; then window of sender = {A+1, A+2, …, A+n}



Sliding Window

- Let A be the **last ACK'd packet of sender without gap**; then window of sender = {A+1, A+2, …, A+n}



- Let B be the **last received packet without gap** by receiver; then window of receiver = {B+1, …, B+n}

Throughput of Sliding Window

Throughput of Sliding Window

- If window size is n, then throughput is roughly:

MIN[**n *** DATA/RTT, Link bandwidth]

Throughput of Sliding Window

- If window size is n, then throughput is roughly:

MIN[**n *** DATA/RTT, Link bandwidth]

- Compare to Stop and Wait:

MIN[DATA/RTT, Link bandwidth]

Throughput of Sliding Window

- If window size is n, then throughput is roughly:

MIN[**n *** DATA/RTT, Link bandwidth]

- Compare to Stop and Wait:

MIN[DATA/RTT, Link bandwidth]

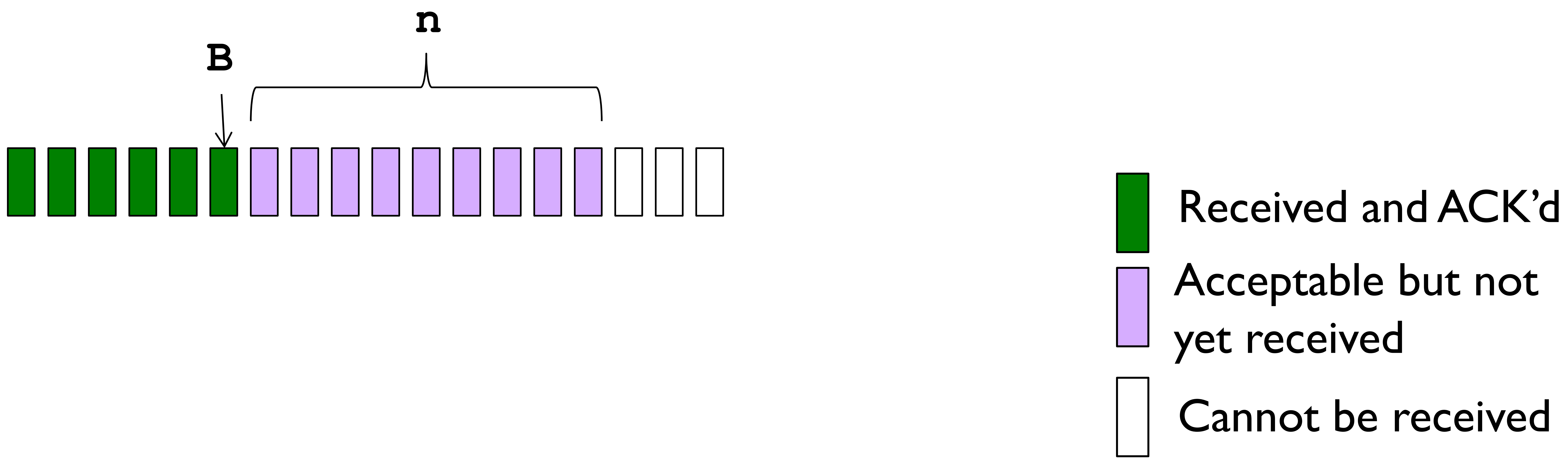
- Two questions:
    - What happens when n gets too large?
    - How do we choose n?

Acknowledgements with Sliding Window

- **Two common options**
  - **Cumulative ACKs:** ACK carries *next in-order sequence number* that the receiver expects

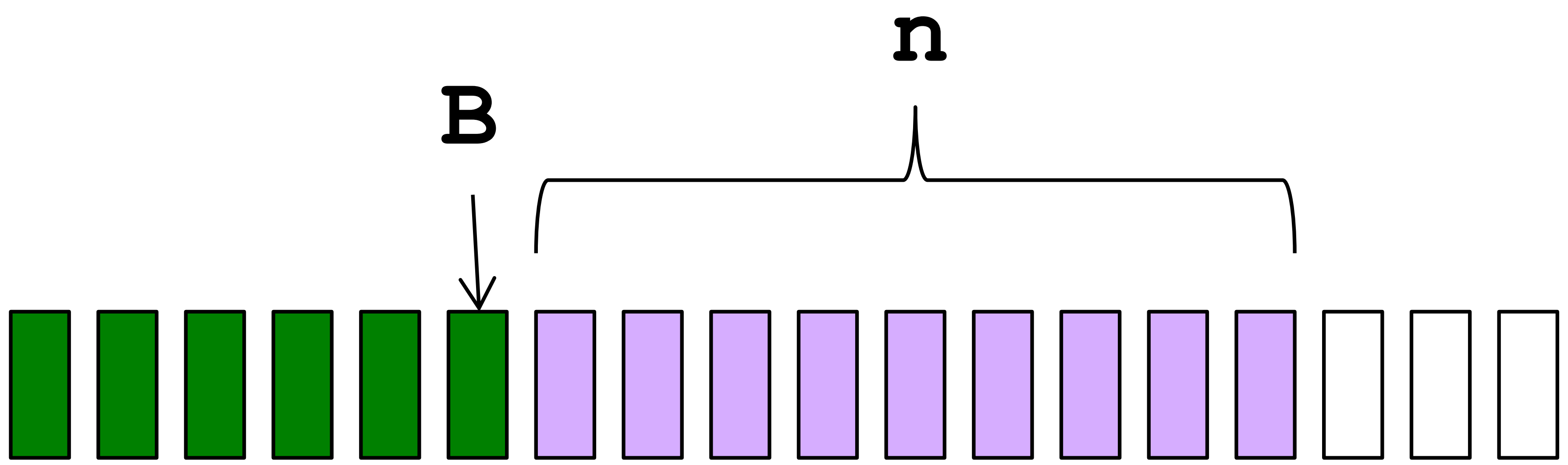
Cumulative Acknowledgements (1)

- At receiver

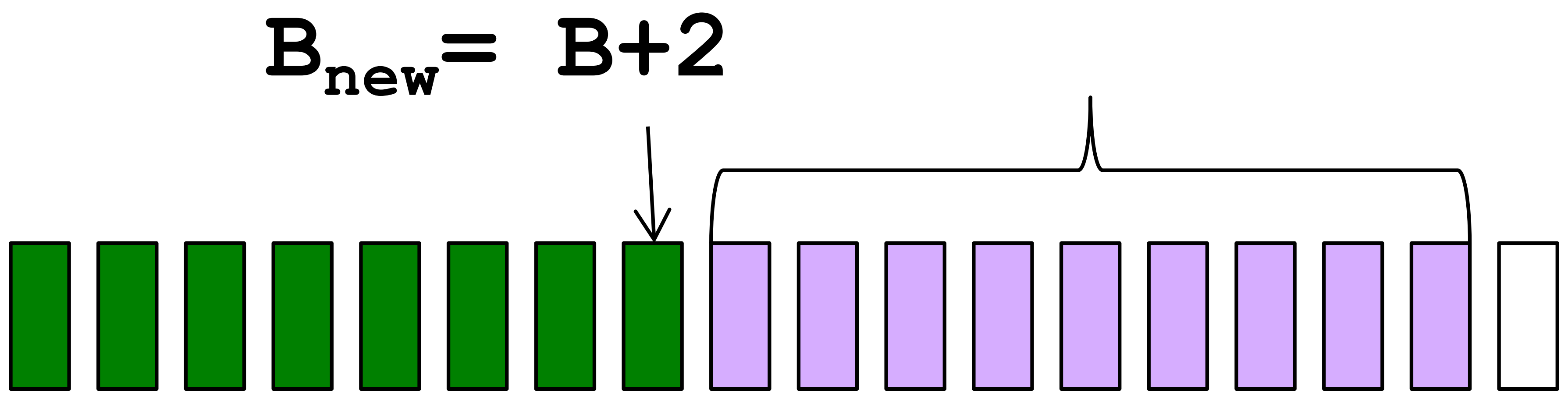


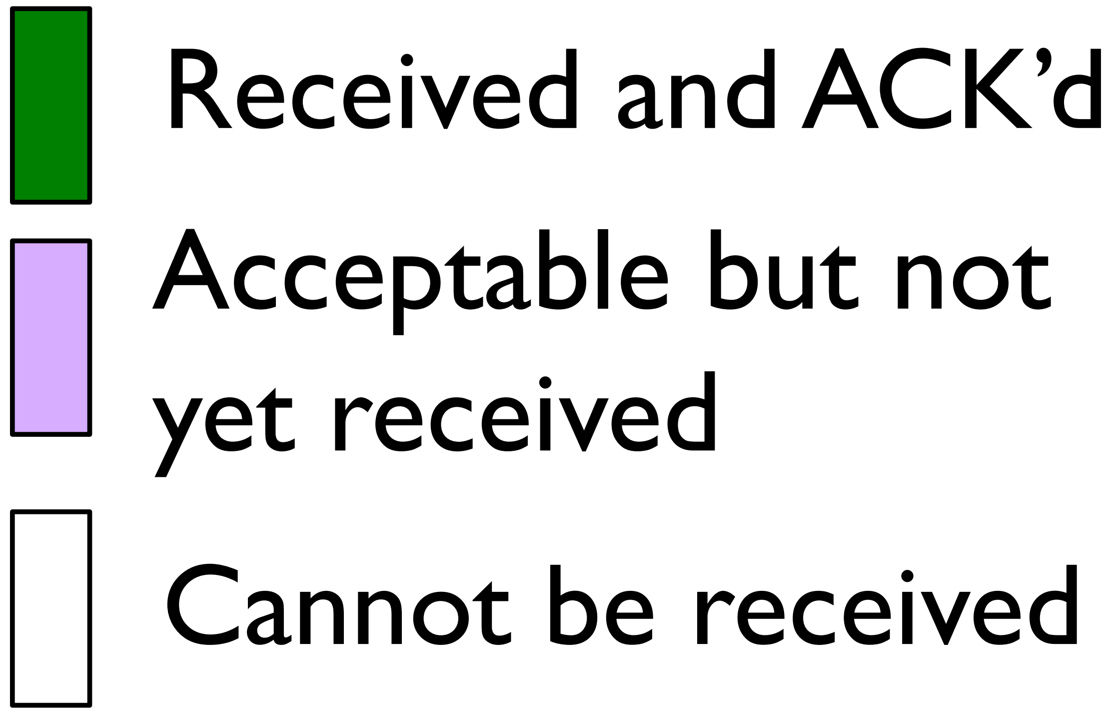
Cumulative Acknowledgements (1)

- At receiver



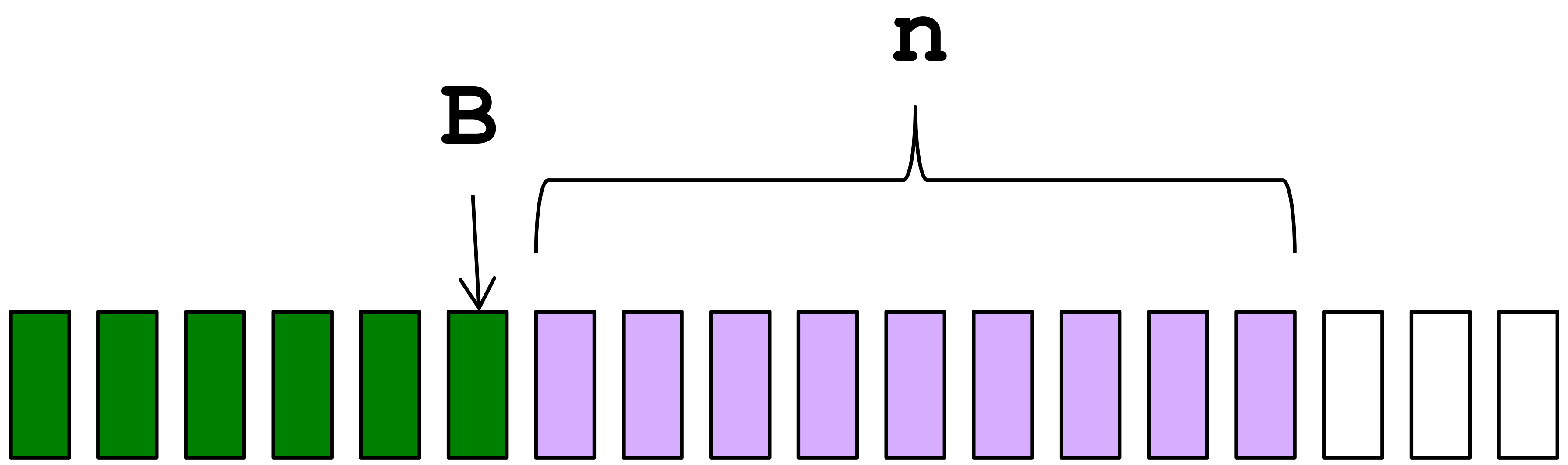
- After receiving B+1, B+2



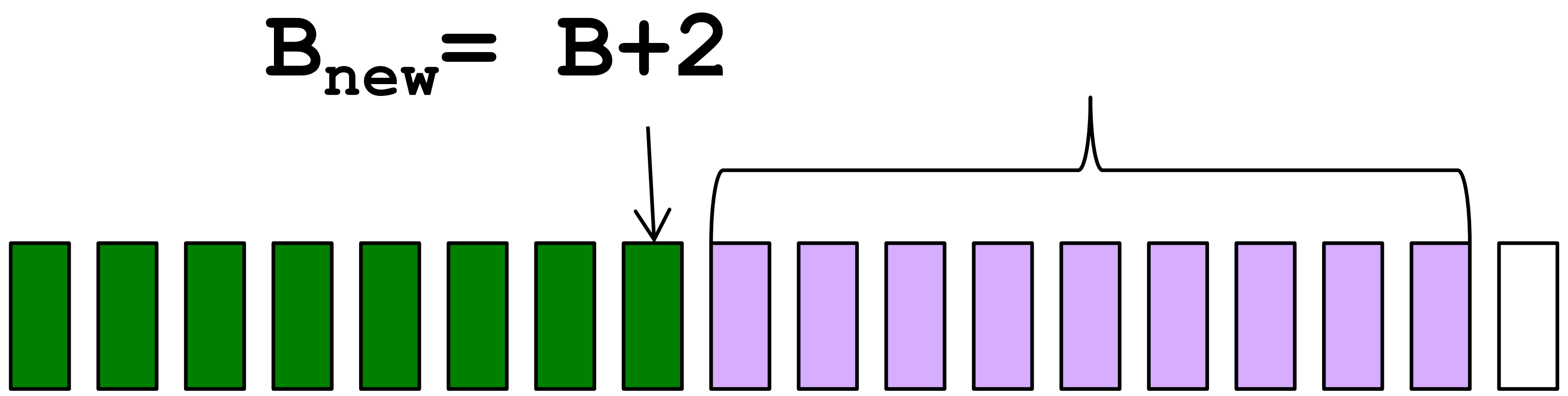


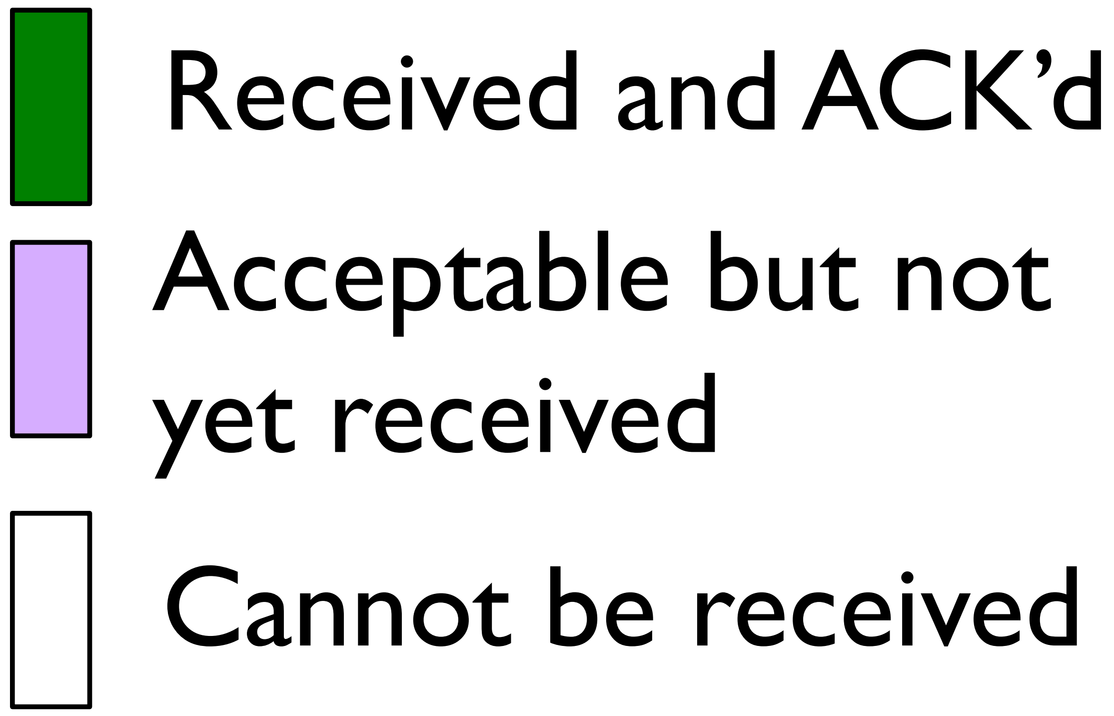
Cumulative Acknowledgements (1)

- At receiver



- After receiving B+1, B+2

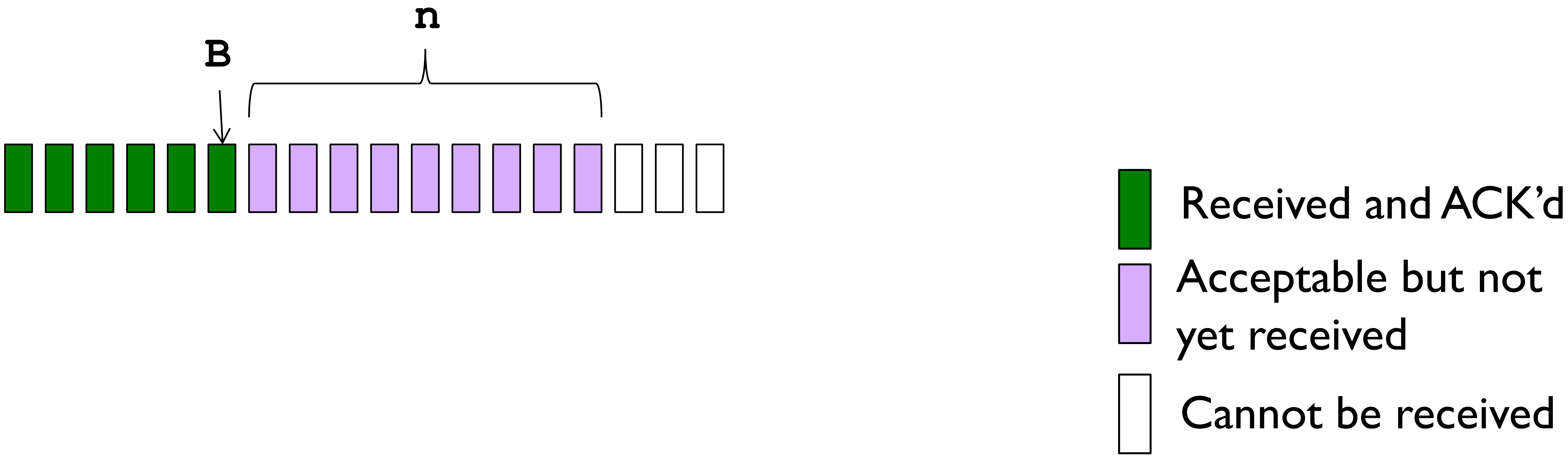




- Receiver sends **ACK(B+3) = ACK(B_{new}+1)**

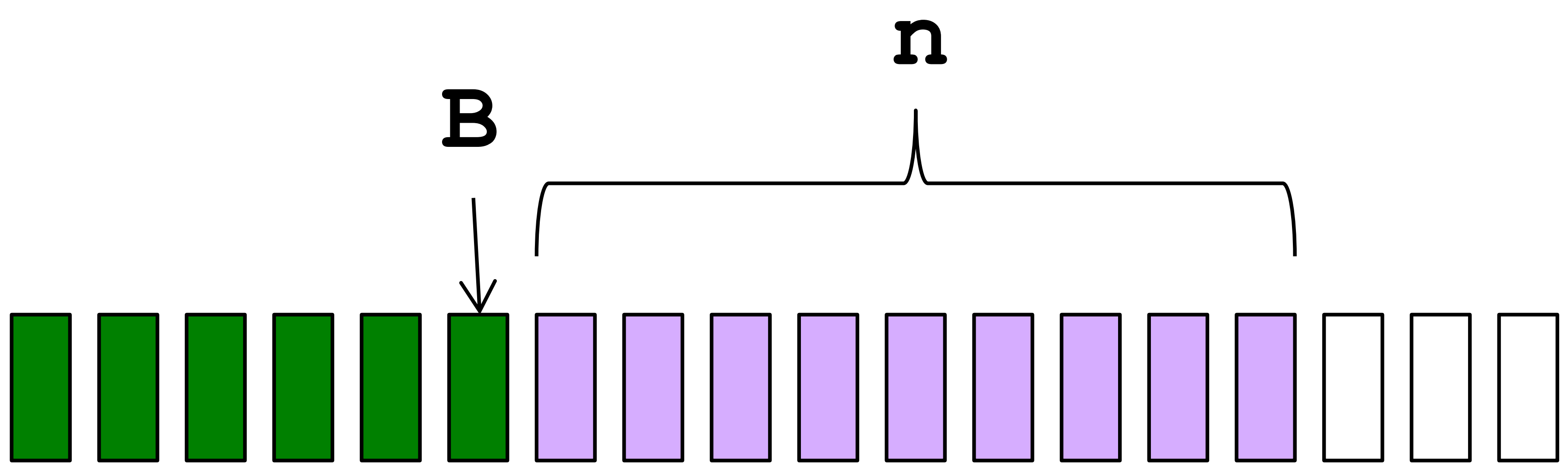
Cumulative Acknowledgements (2)

- At receiver

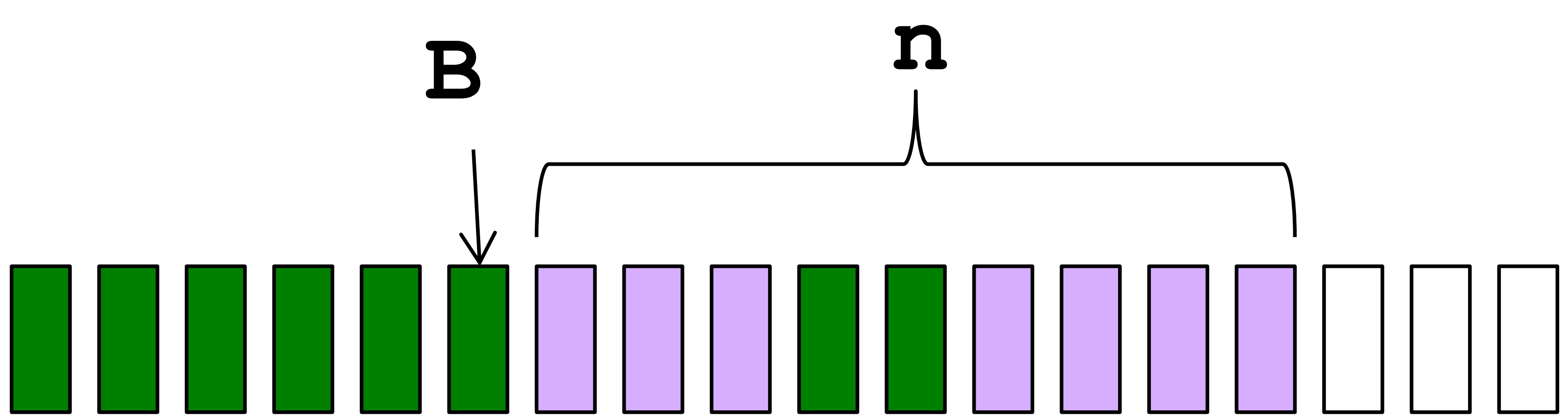


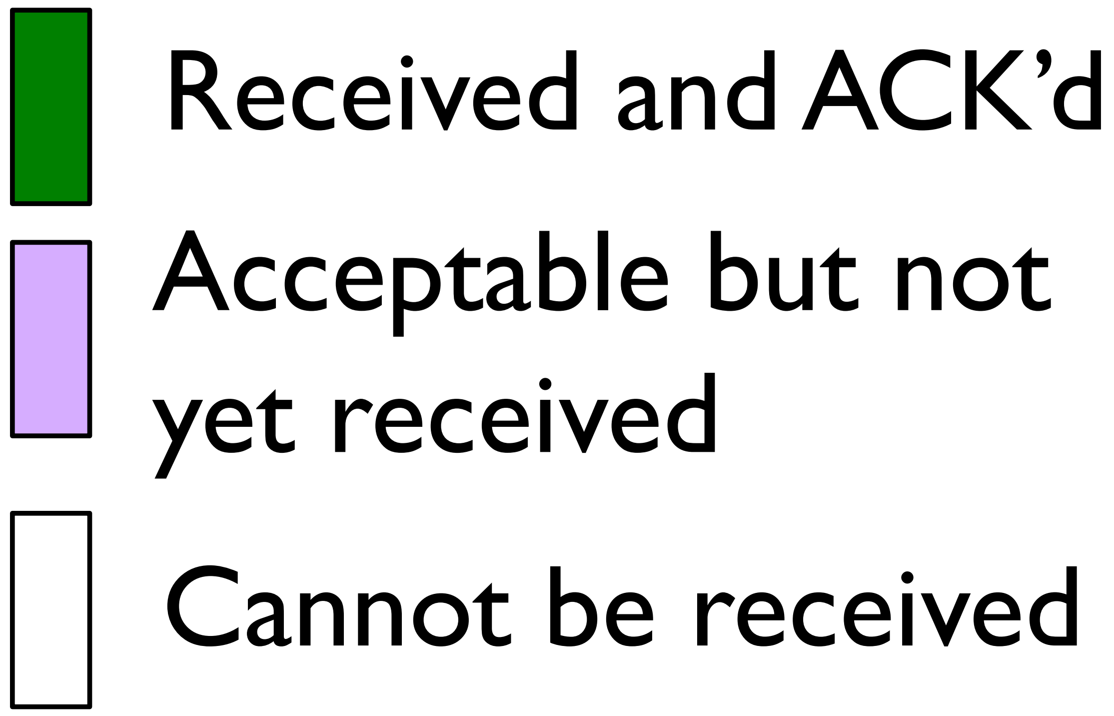
Cumulative Acknowledgements (2)

- At receiver



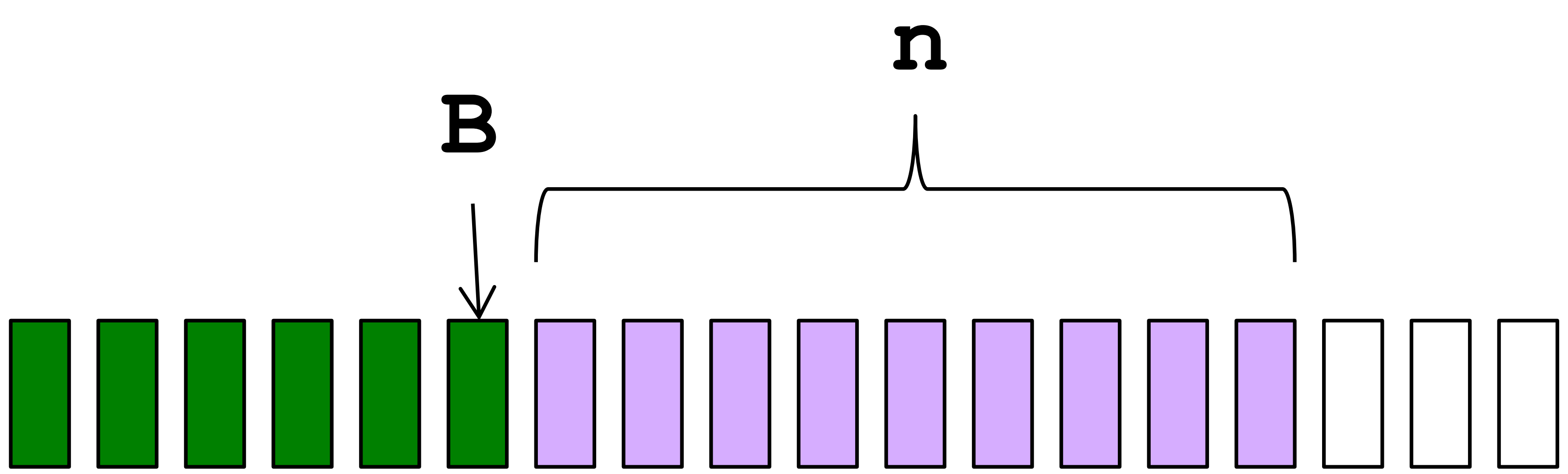
- After receiving B+4, B+5



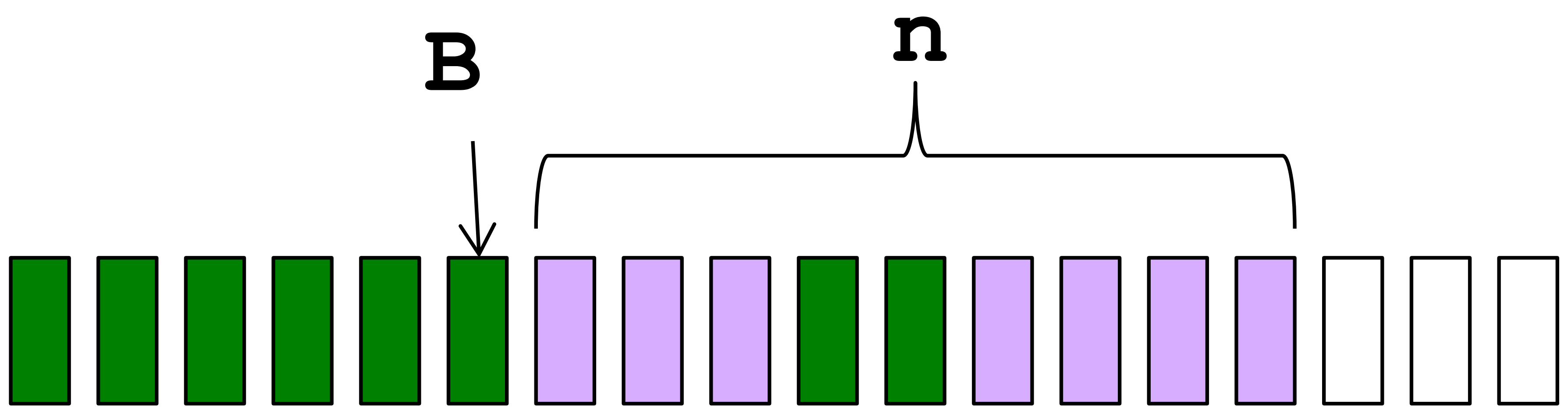


Cumulative Acknowledgements (2)

- At receiver



- After receiving B+4, B+5



- Receiver sends **ACK(B+1)**

Received and ACK'd

Acceptable but not yet received

Cannot be received

Acknowledgements with Sliding Window

- **Two common options**
  - **Cumulative ACKs:** ACK carries *next in-order sequence number* that the receiver expects

Acknowledgements with Sliding Window

- **Two common options**
  - **Cumulative ACKs:** ACK carries *next in-order sequence number* that the receiver expects
  - **Selective ACKs:** ACK *individually acknowledges* correctly received packets

Acknowledgements with Sliding Window

- **Two common options**
  - **Cumulative ACKs:** ACK carries *next in-order sequence number* that the receiver expects
  - **Selective ACKs:** ACK *individually acknowledges* correctly received packets
- Selective ACKs offer **more precise information** but require **more complicated book-keeping**

Sliding Window Protocols

- **Resending packets:** two canonical approaches
  - Go-Back-N (GBN)
  - Selective Repeat
- Many variants that differ in implementation details

Go-Back-N (GBN)

Go-Back-N (GBN)

- Sender transmits up to *n* unacknowledged packets

Go-Back-N (GBN)

- Sender transmits up to *n* unacknowledged packets
- Receiver only accepts packets in order
  - Discards out-of-order packets (i.e., packets other than B+1)

Go-Back-N (GBN)

- Sender transmits up to *n* unacknowledged packets
- Receiver only accepts packets in order
  - Discards out-of-order packets (i.e., packets other than B+1)
- Receiver uses **cumulative acknowledgements**
  - i.e., sequence# in ACK = next expected in-order sequence#

Go-Back-N (GBN)

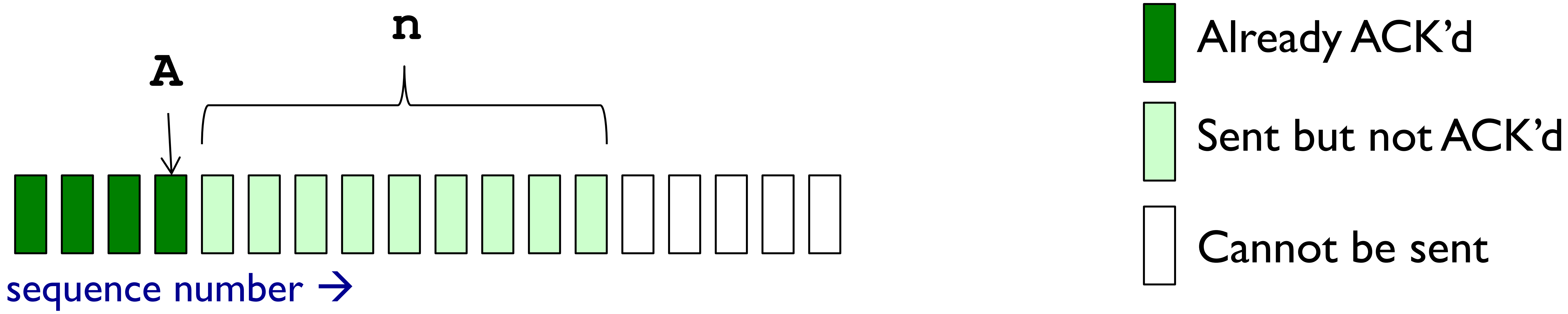
- Sender transmits up to *n* unacknowledged packets
- Receiver only accepts packets in order
  - Discards out-of-order packets (i.e., packets other than B+1)
- Receiver uses **cumulative acknowledgements**
  - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1st outstanding ACK(A+1)

Go-Back-N (GBN)

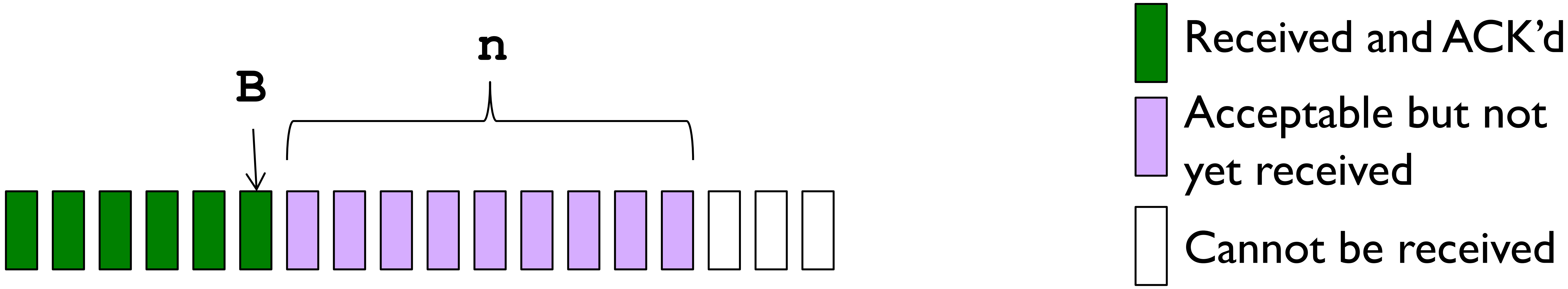
- Sender transmits up to *n* unacknowledged packets
- Receiver only accepts packets in order
  - Discards out-of-order packets (i.e., packets other than B+1)
- Receiver uses **cumulative acknowledgements**
  - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1st outstanding ACK(A+1)
- If timeout, retransmit A+1, …, A+n

Sliding Window with GBN

- Let A be the **last ACK'd packet of sender without gap**; then window of sender = {A+1, A+2, …, A+n}



- Let B be the **last received packet without gap** by receiver; then window of receiver = {B+1, …, B+n}



GBN Example without Errors

Sender Window

Receiver Window

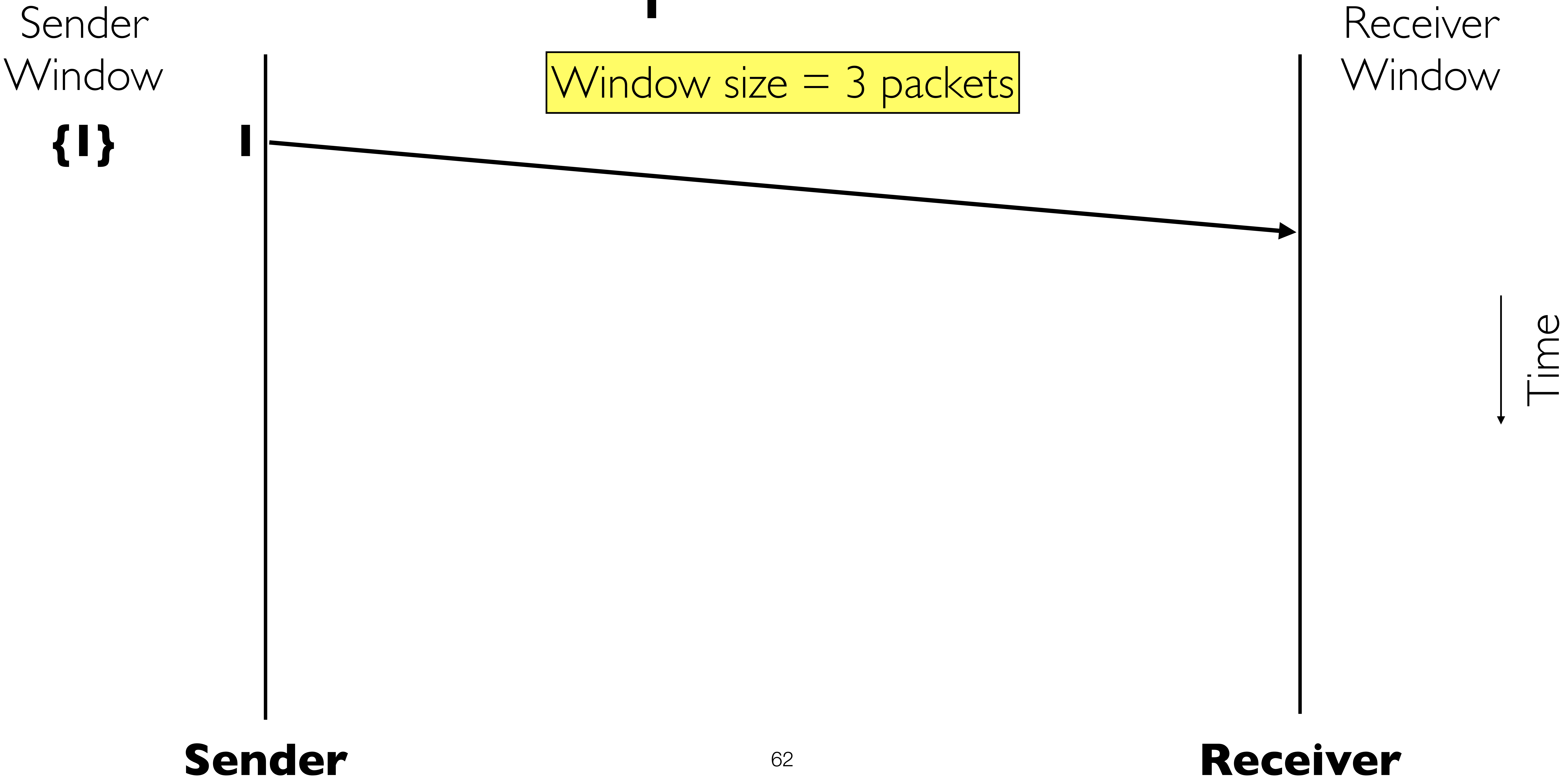
Window size = 3 packets

Time

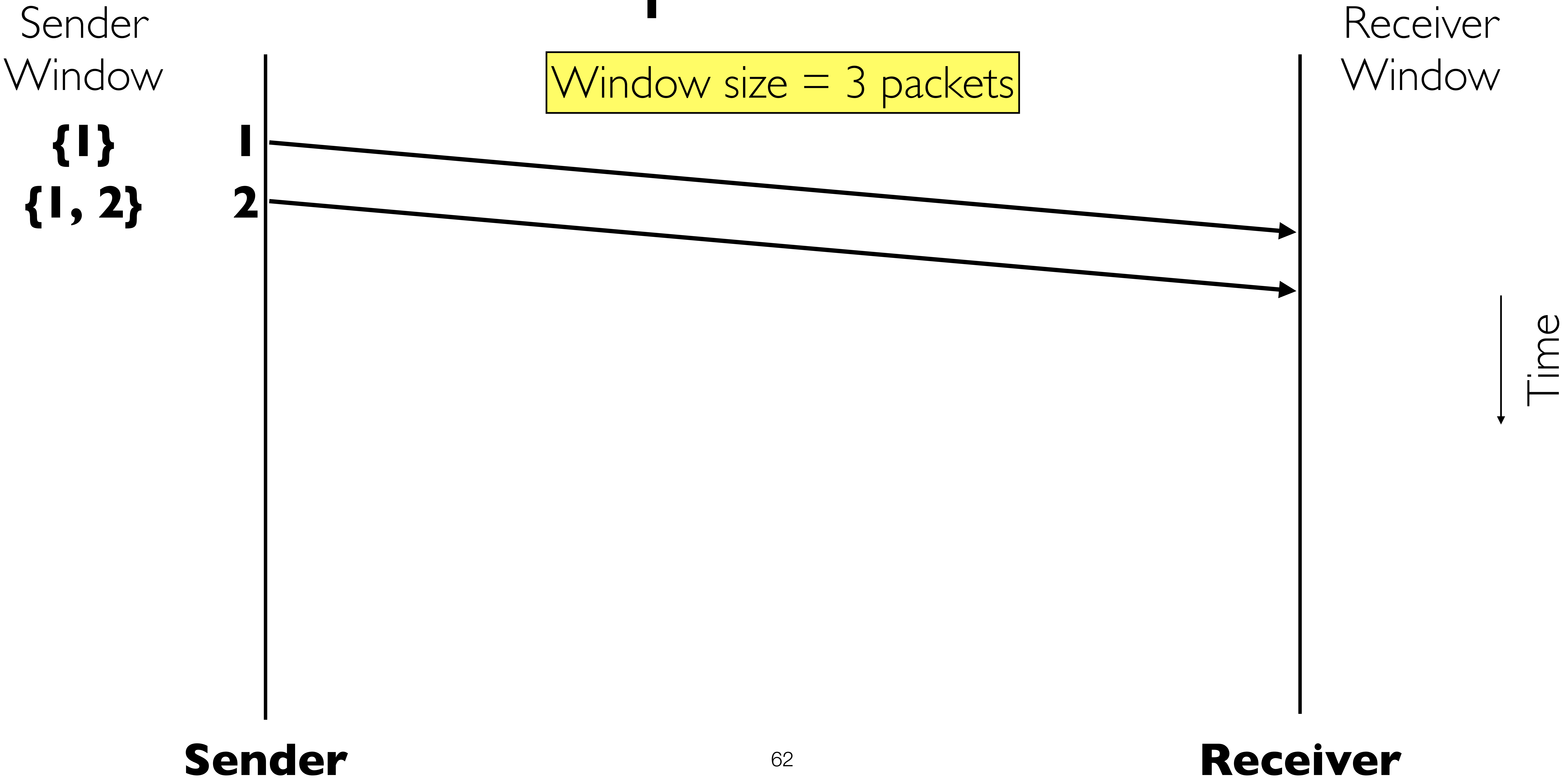
Sender

Receiver

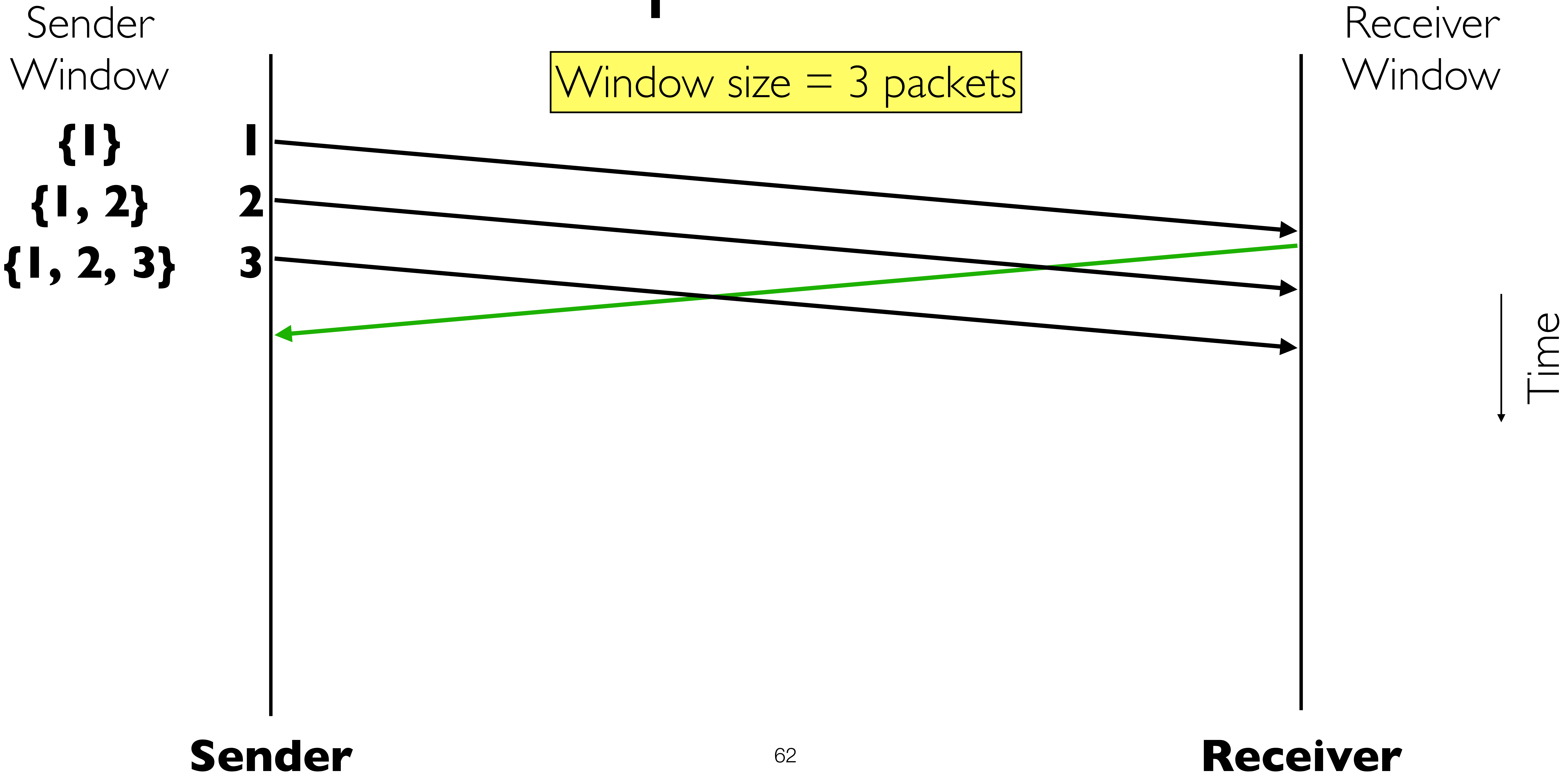
GBN Example without Errors



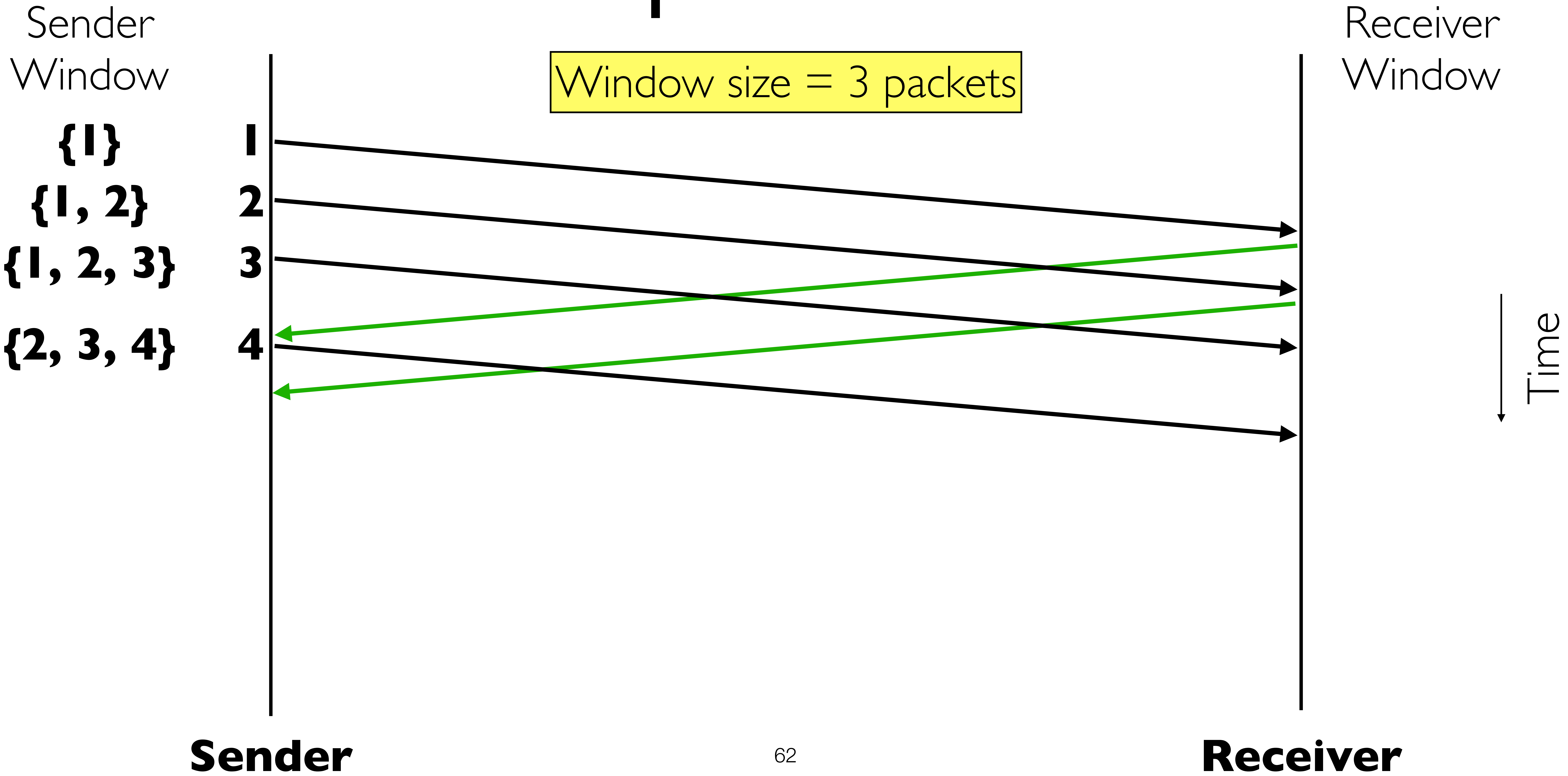
GBN Example without Errors



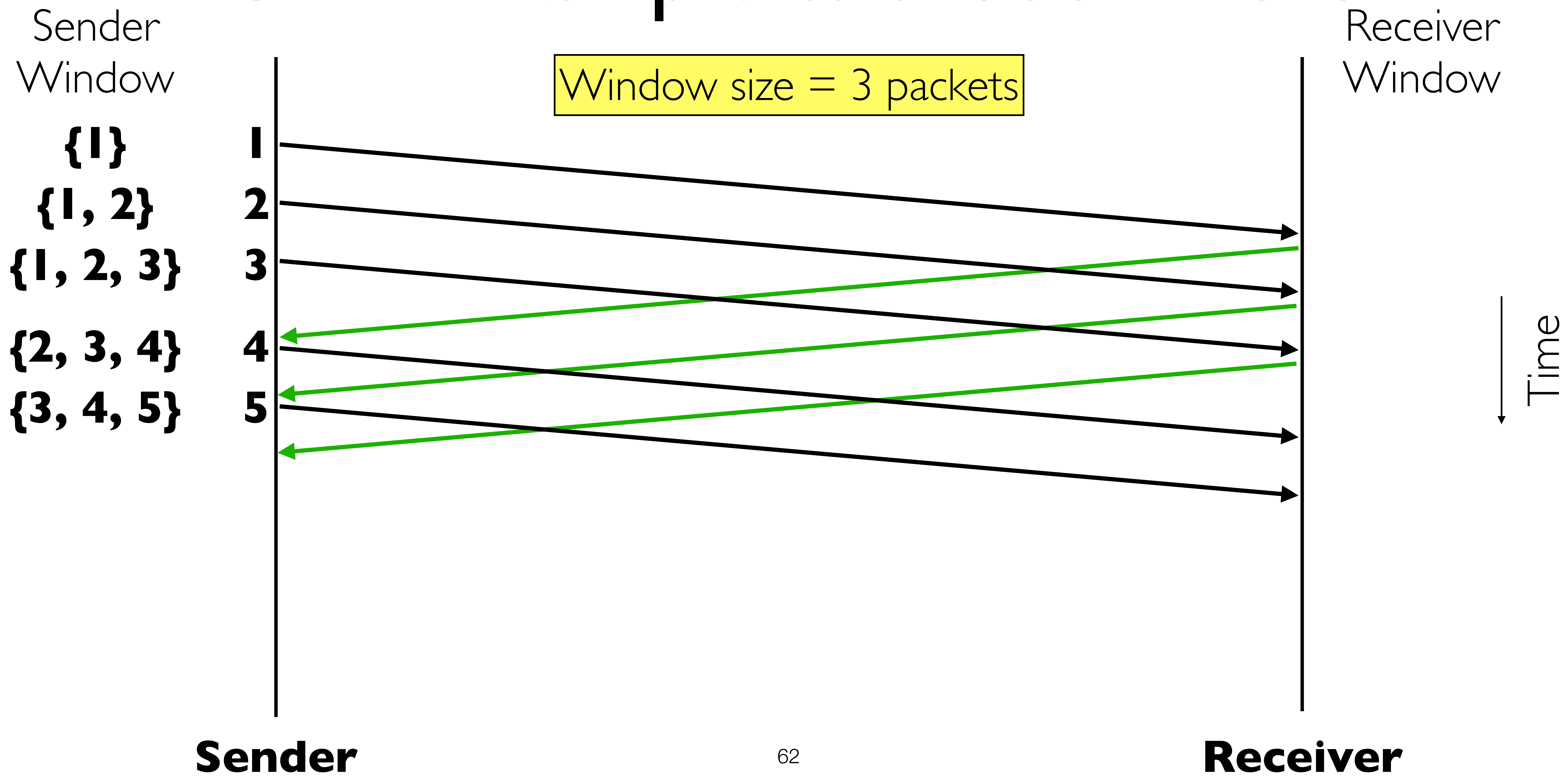
GBN Example without Errors



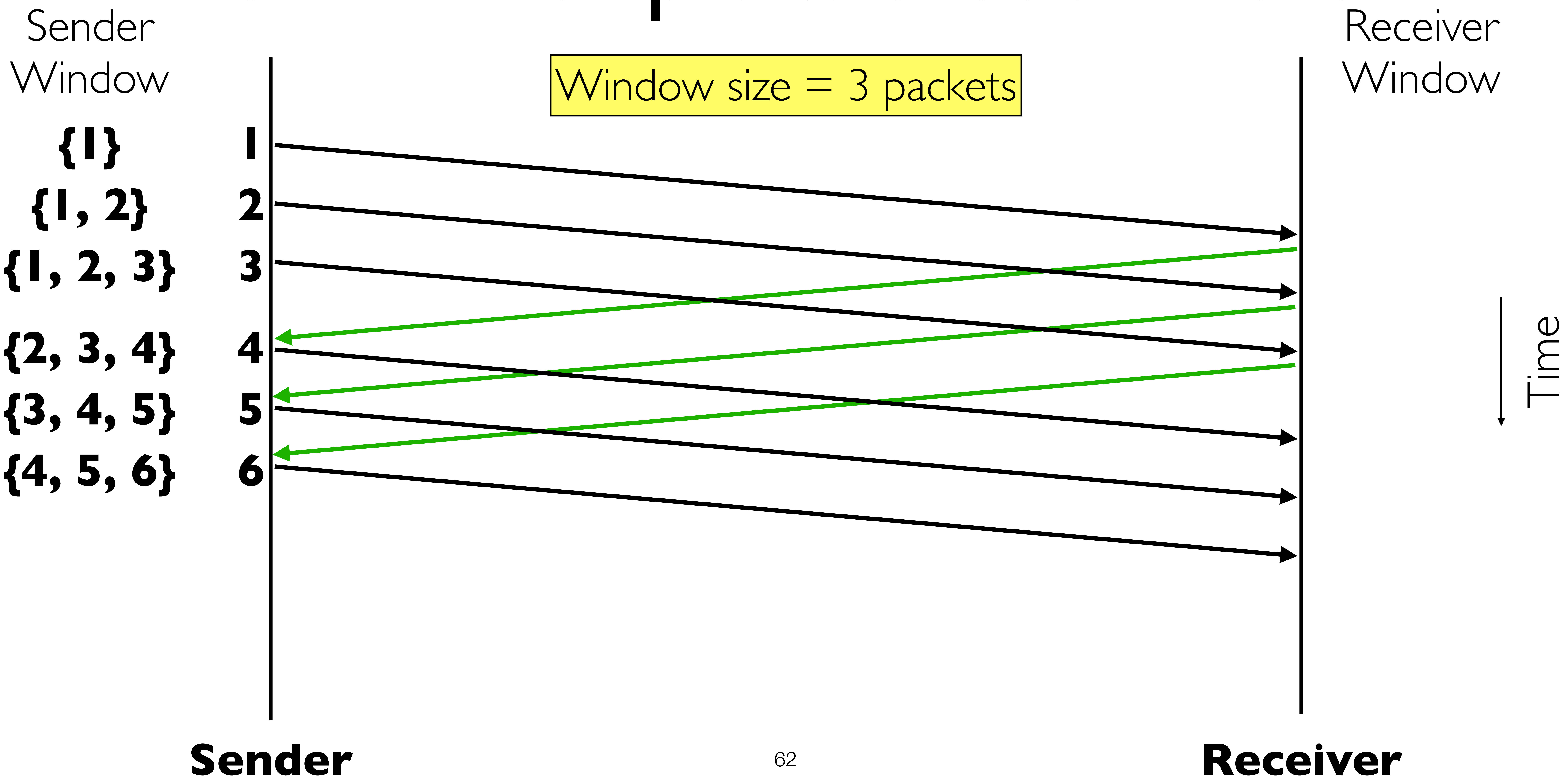
GBN Example without Errors



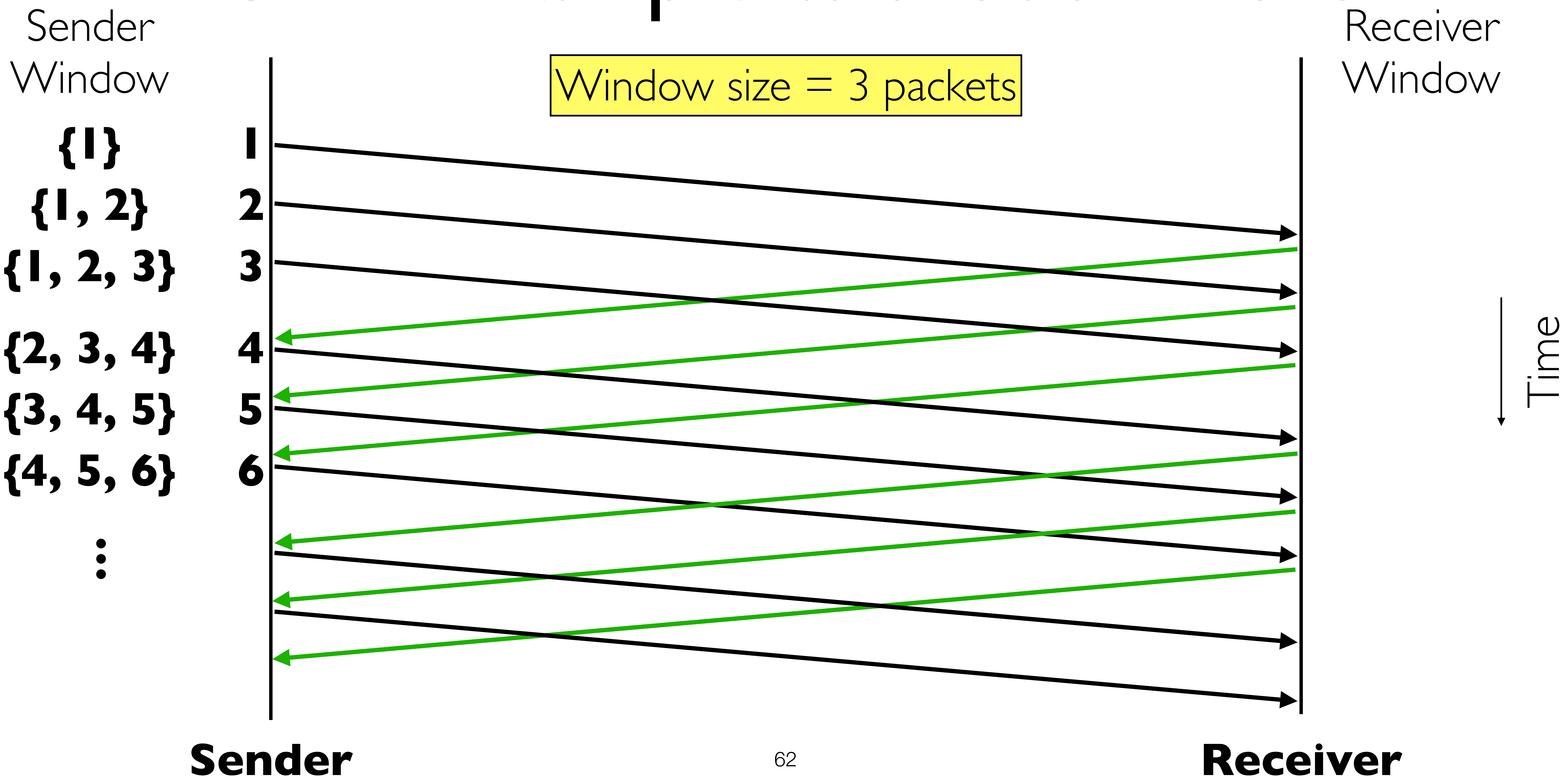
GBN Example without Errors



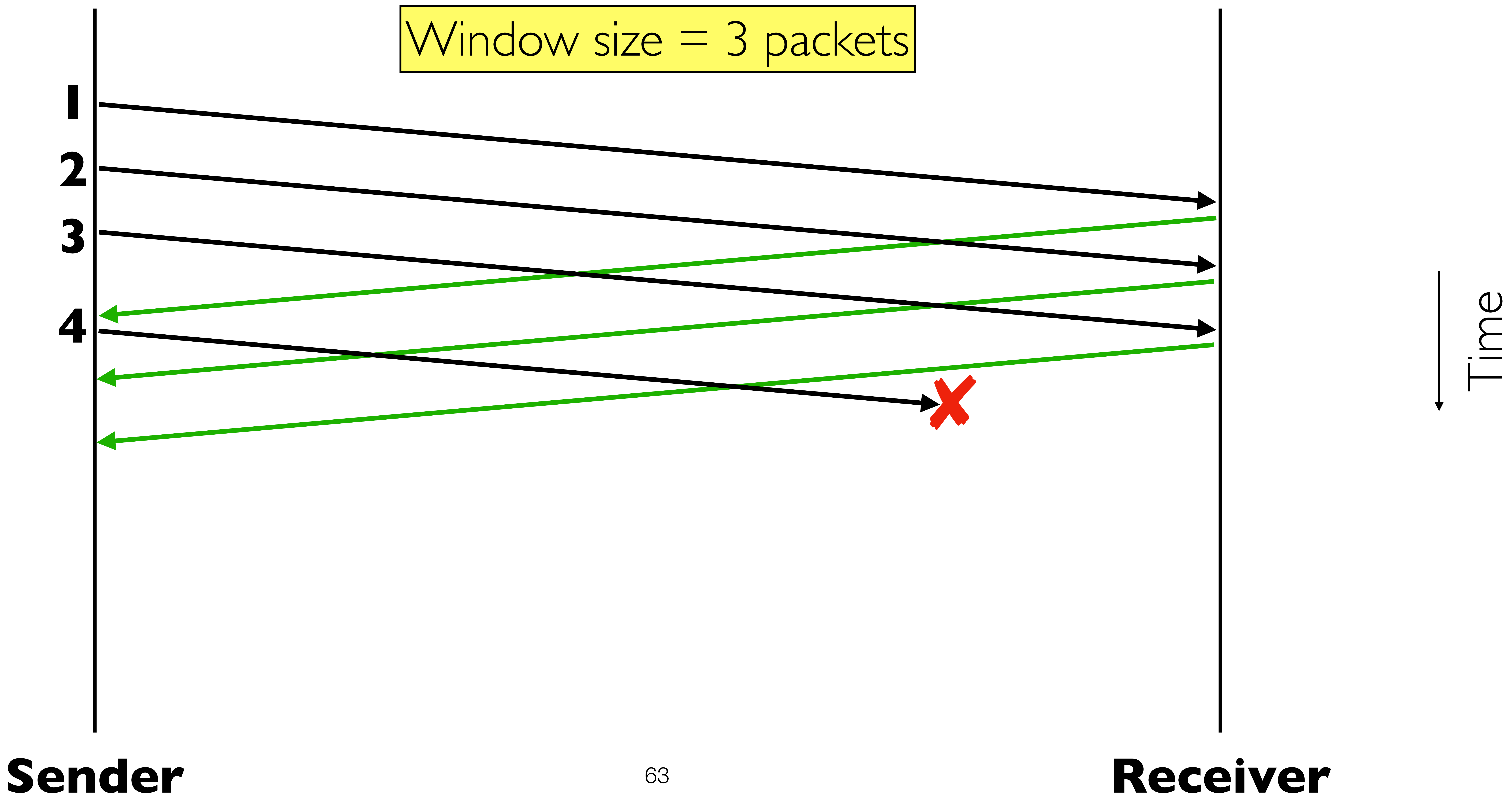
GBN Example without Errors



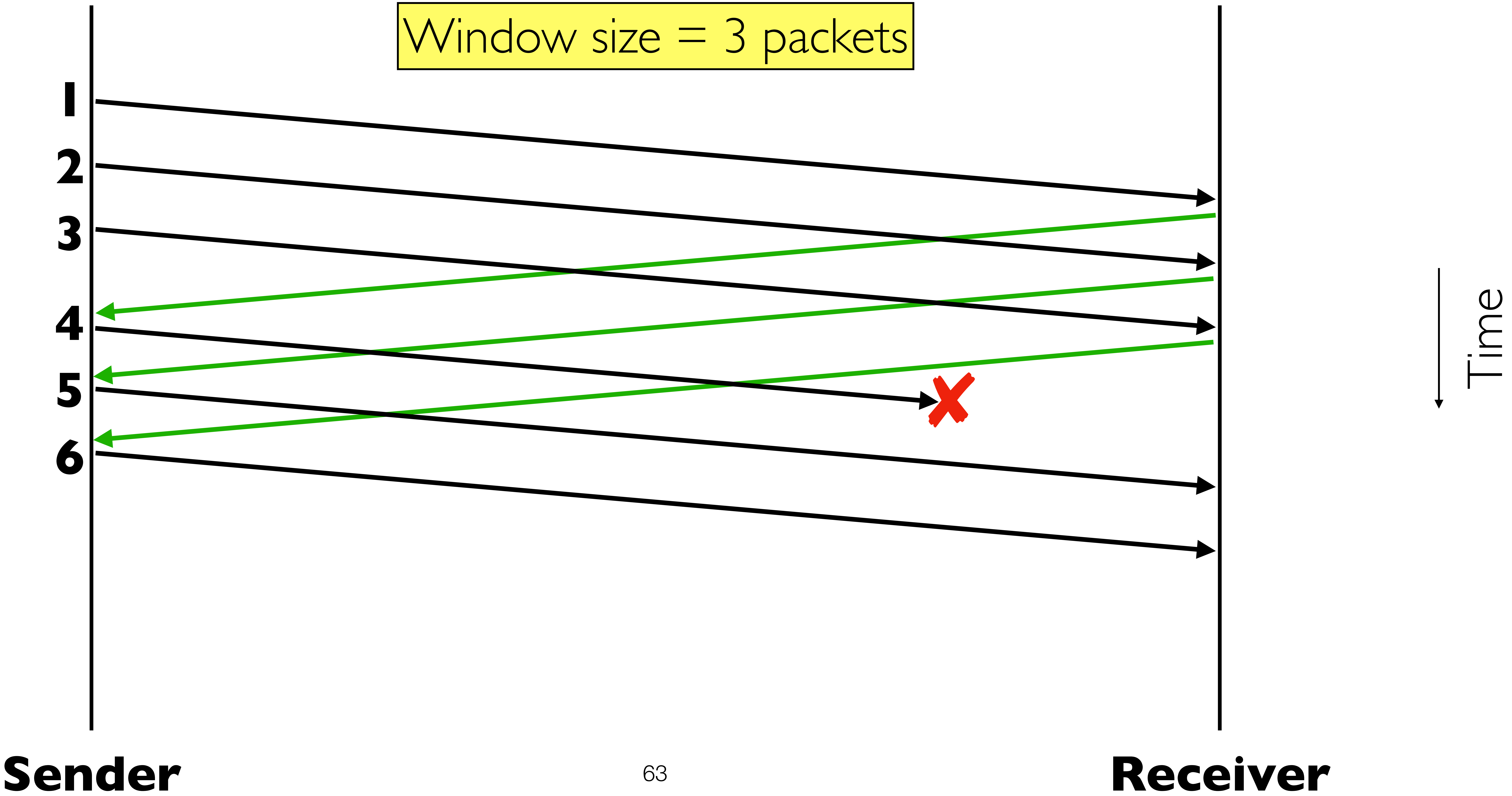
GBN Example without Errors



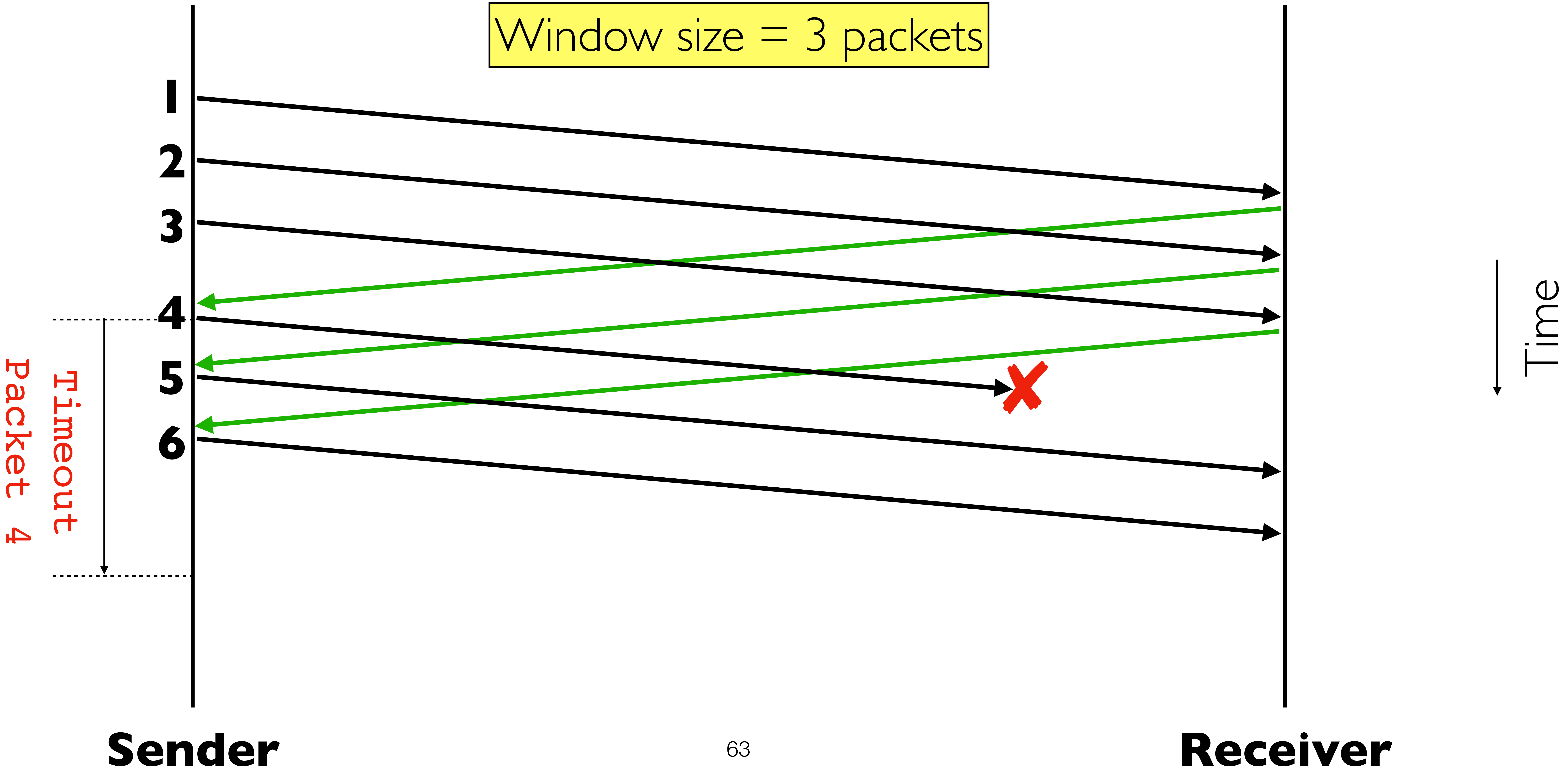
GBN Example with Errors



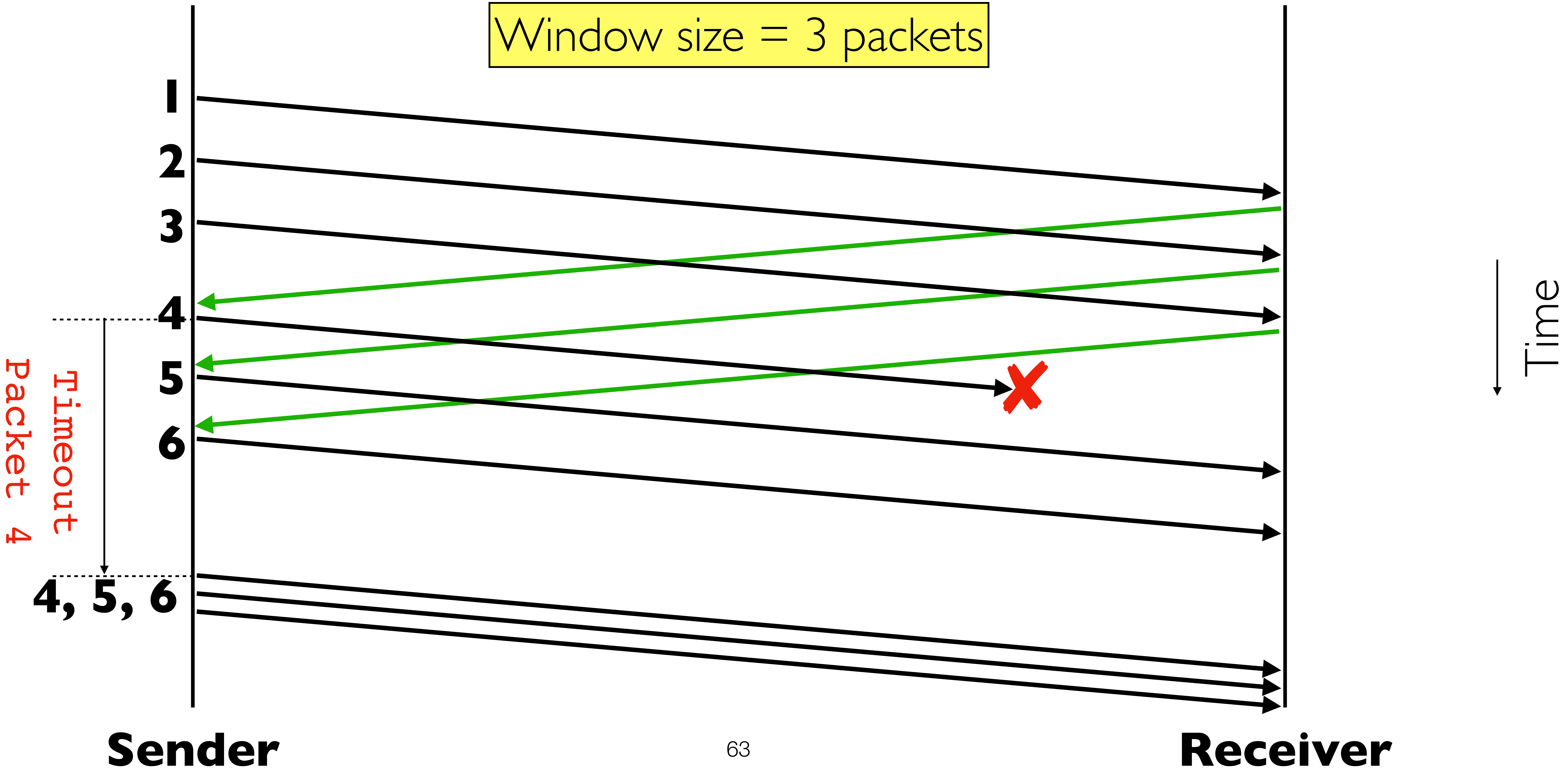
GBN Example with Errors



GBN Example with Errors



GBN Example with Errors



Selective Repeat

Selective Repeat

- Sender transmits up to *n* unacknowledged packets

Selective Repeat

- Sender transmits up to *n* unacknowledged packets
- Assume packet *k* is lost, *k+1* is not

Selective Repeat

- Sender transmits up to *n* unacknowledged packets
- Assume packet *k* is lost, *k+1* is not
- Receiver indicates packet *k+1* correctly received

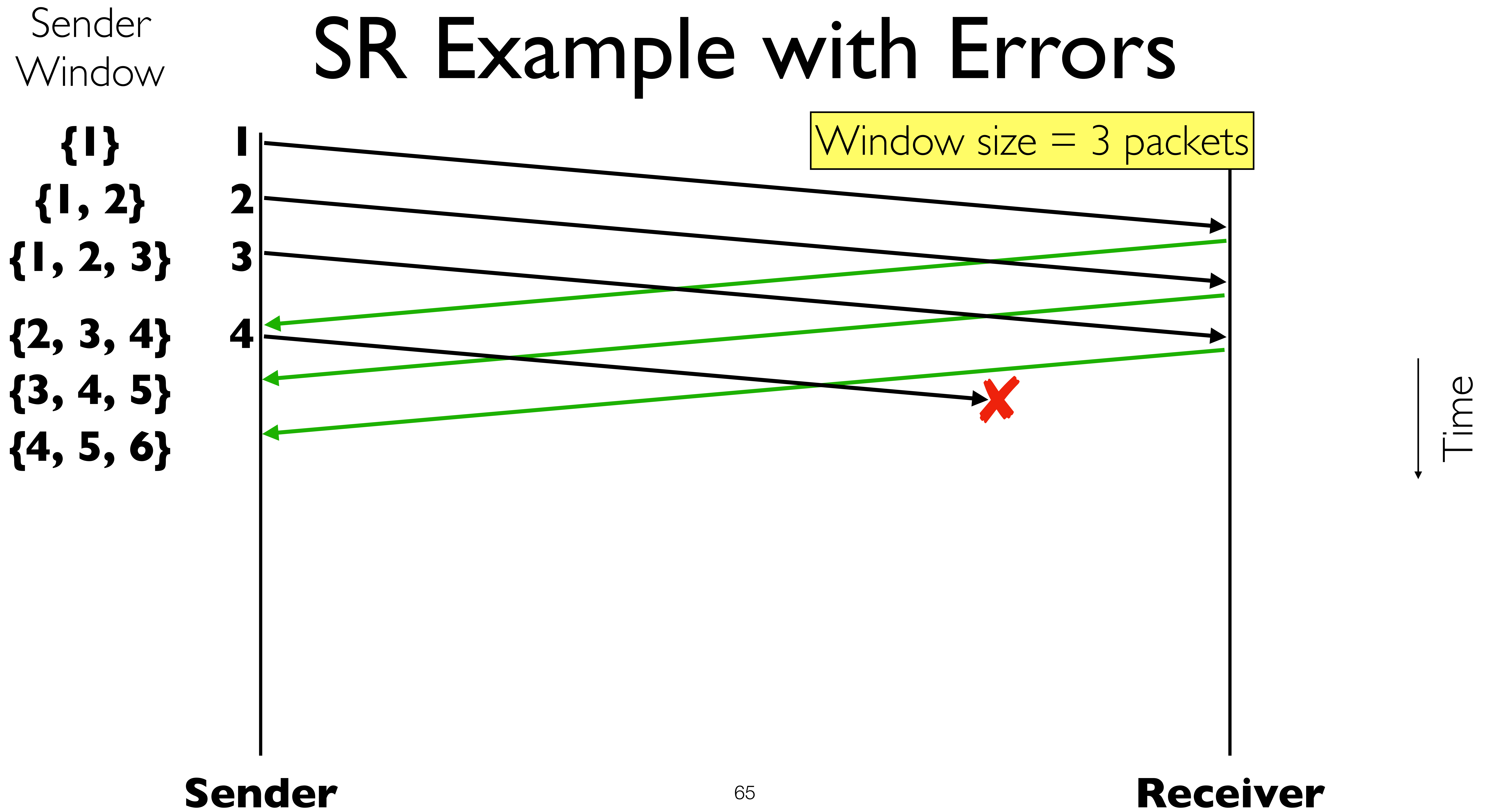
Selective Repeat

- Sender transmits up to *n* unacknowledged packets
- Assume packet *k* is lost, *k+1* is not
- Receiver indicates packet *k+1* correctly received
- Sender retransmits only packet *k* on timeout

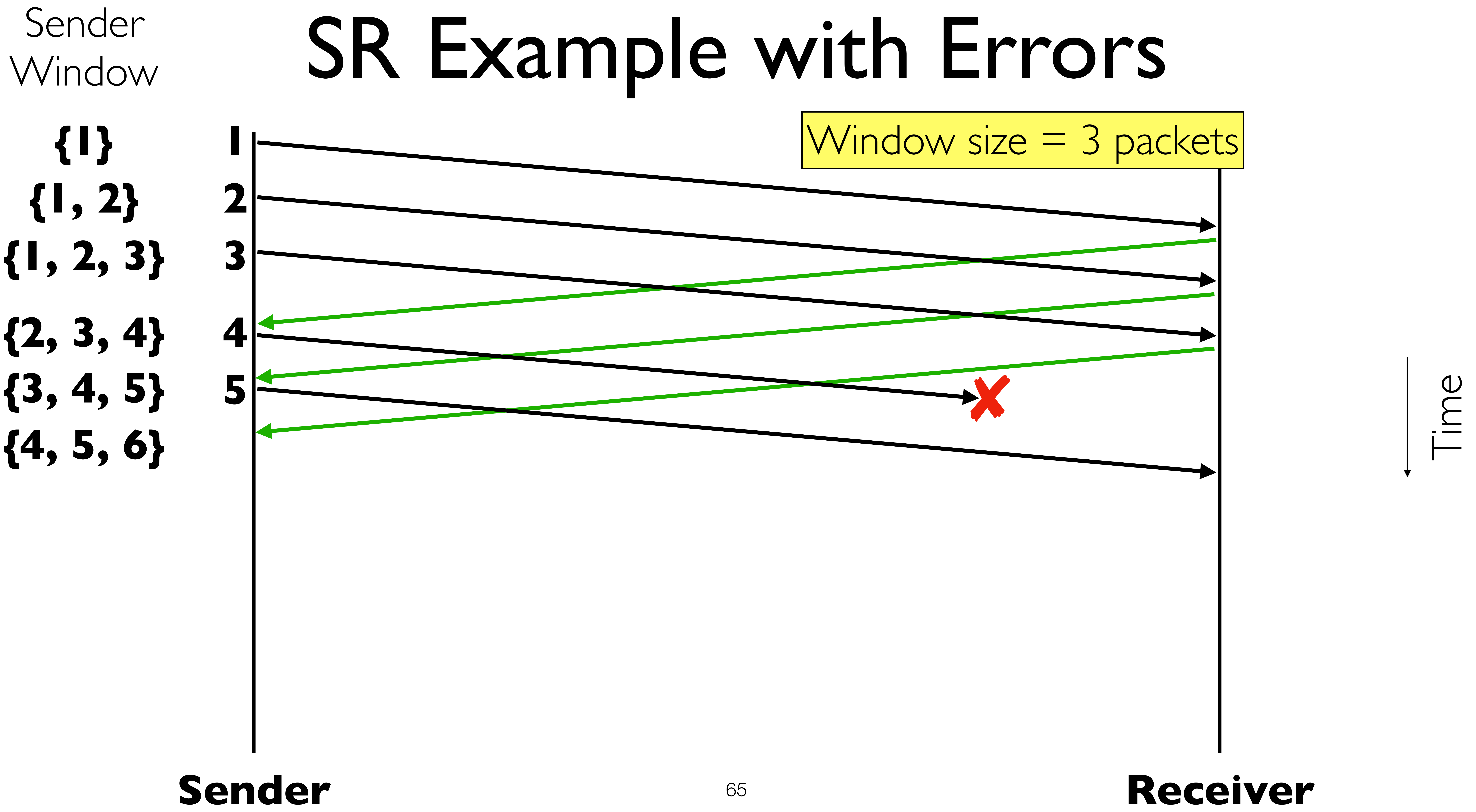
Selective Repeat

- Sender transmits up to *n* unacknowledged packets
- Assume packet *k* is lost, *k+1* is not
- Receiver indicates packet *k+1* correctly received
- Sender retransmits only packet *k* on timeout
- Efficient in retransmissions but complex bookkeeping
    - Need a timer per packet!

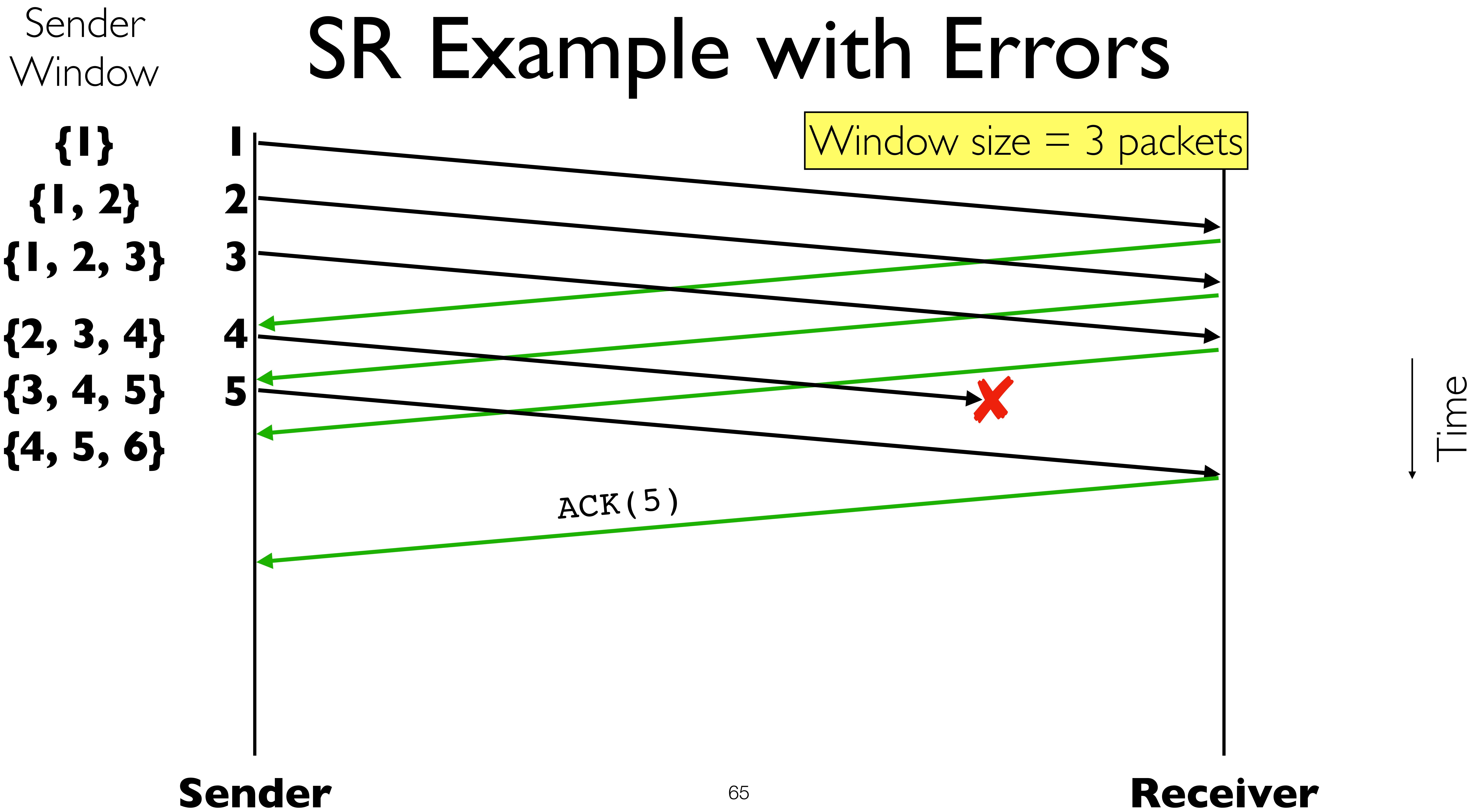
SR Example with Errors



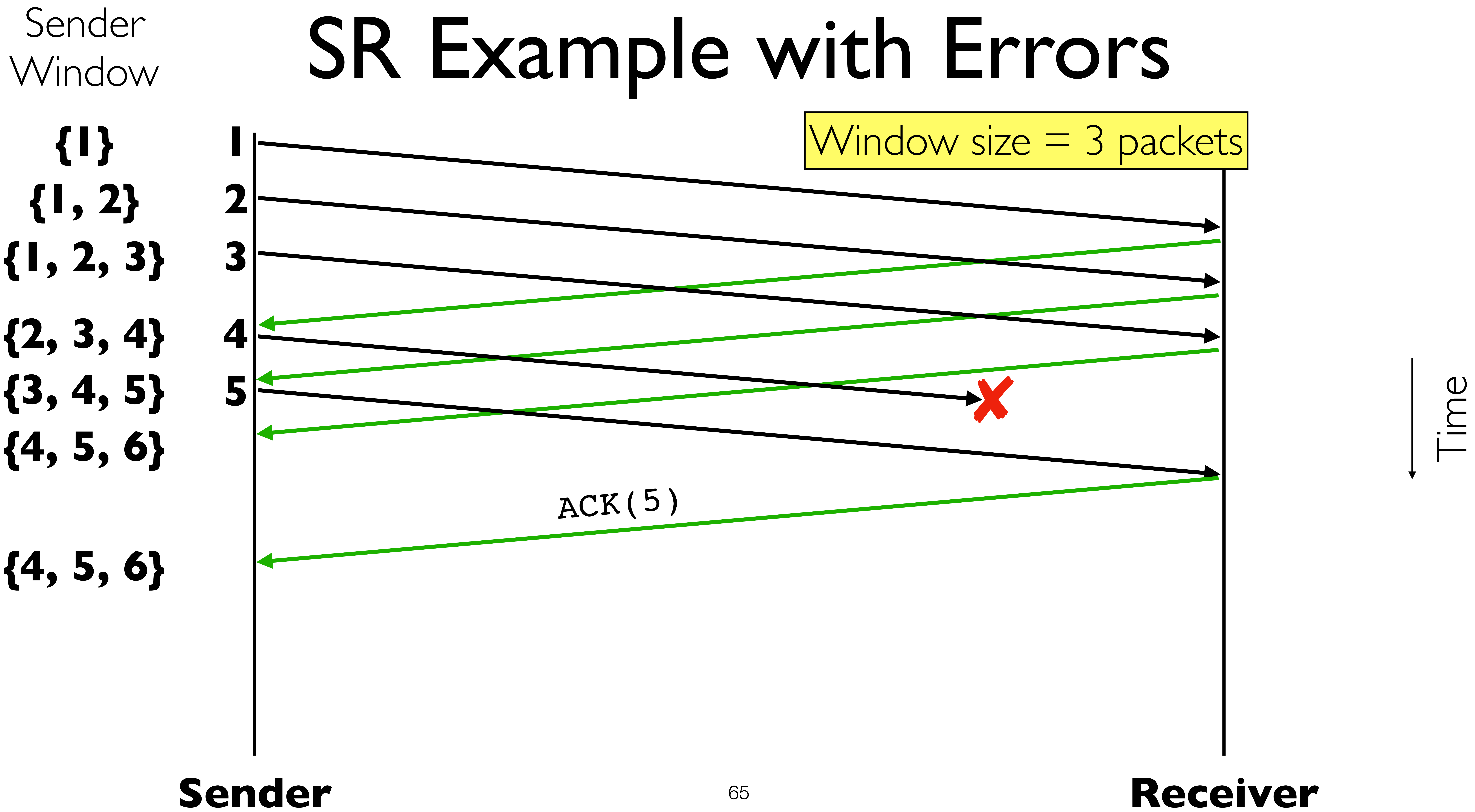
SR Example with Errors



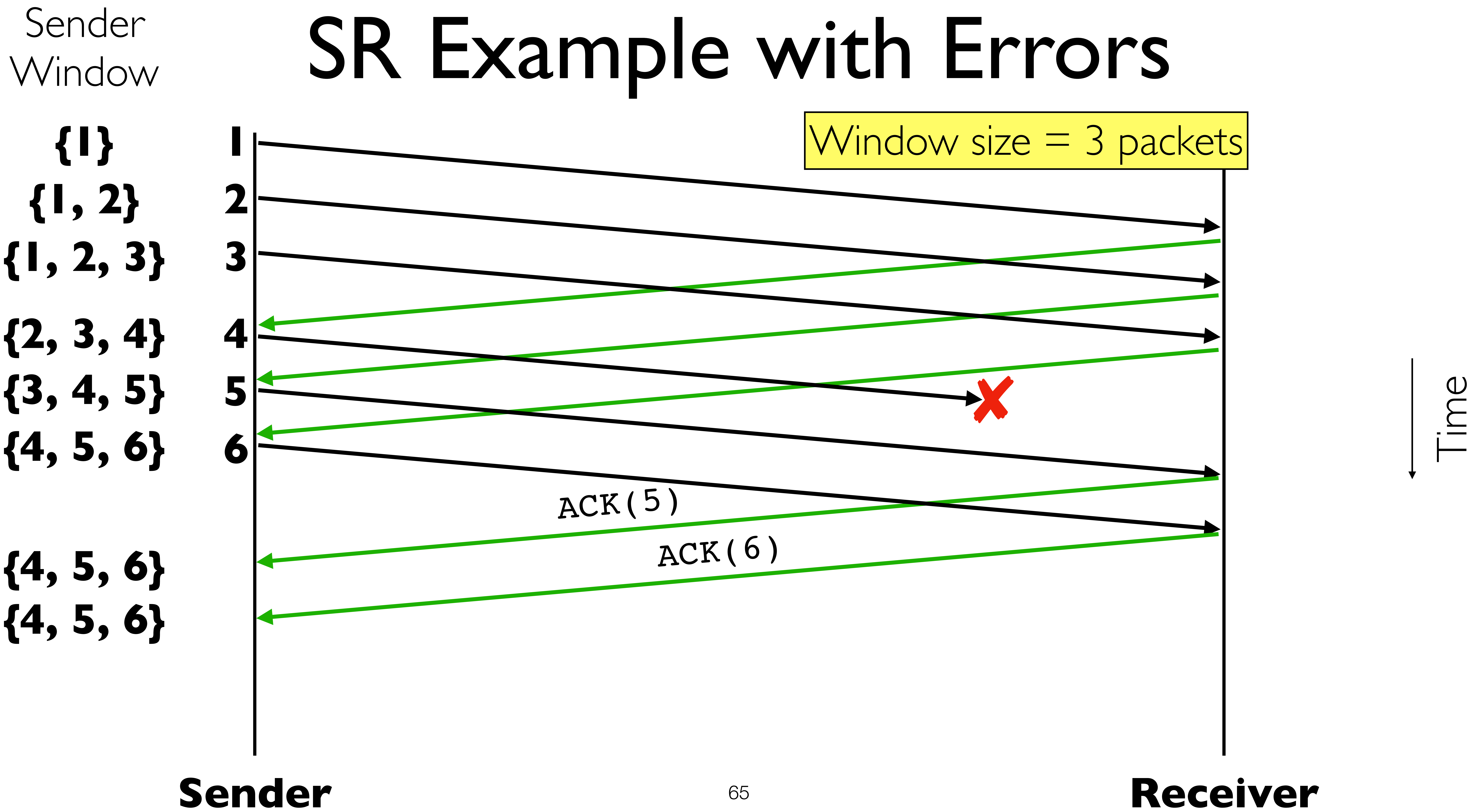
SR Example with Errors



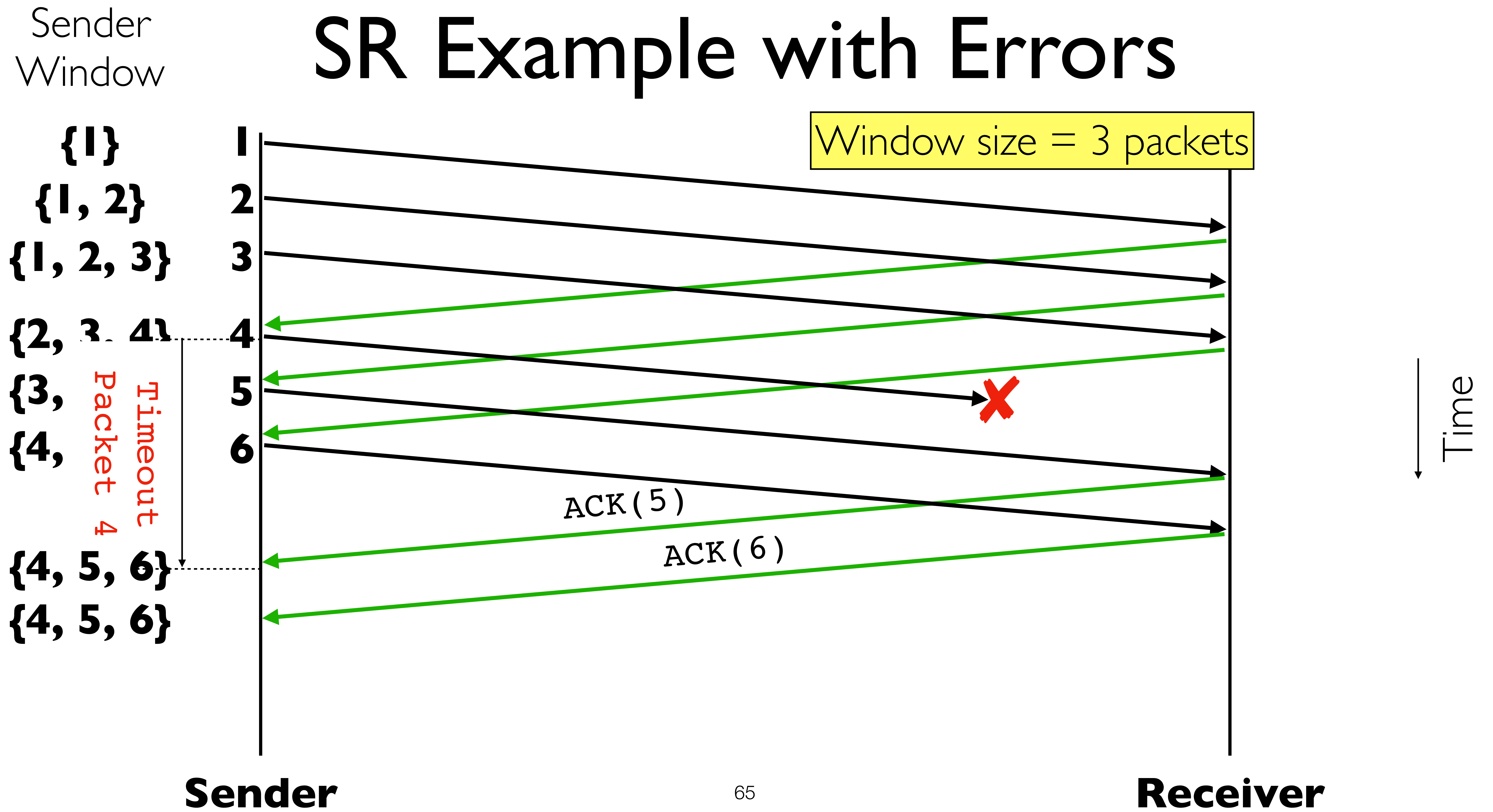
SR Example with Errors



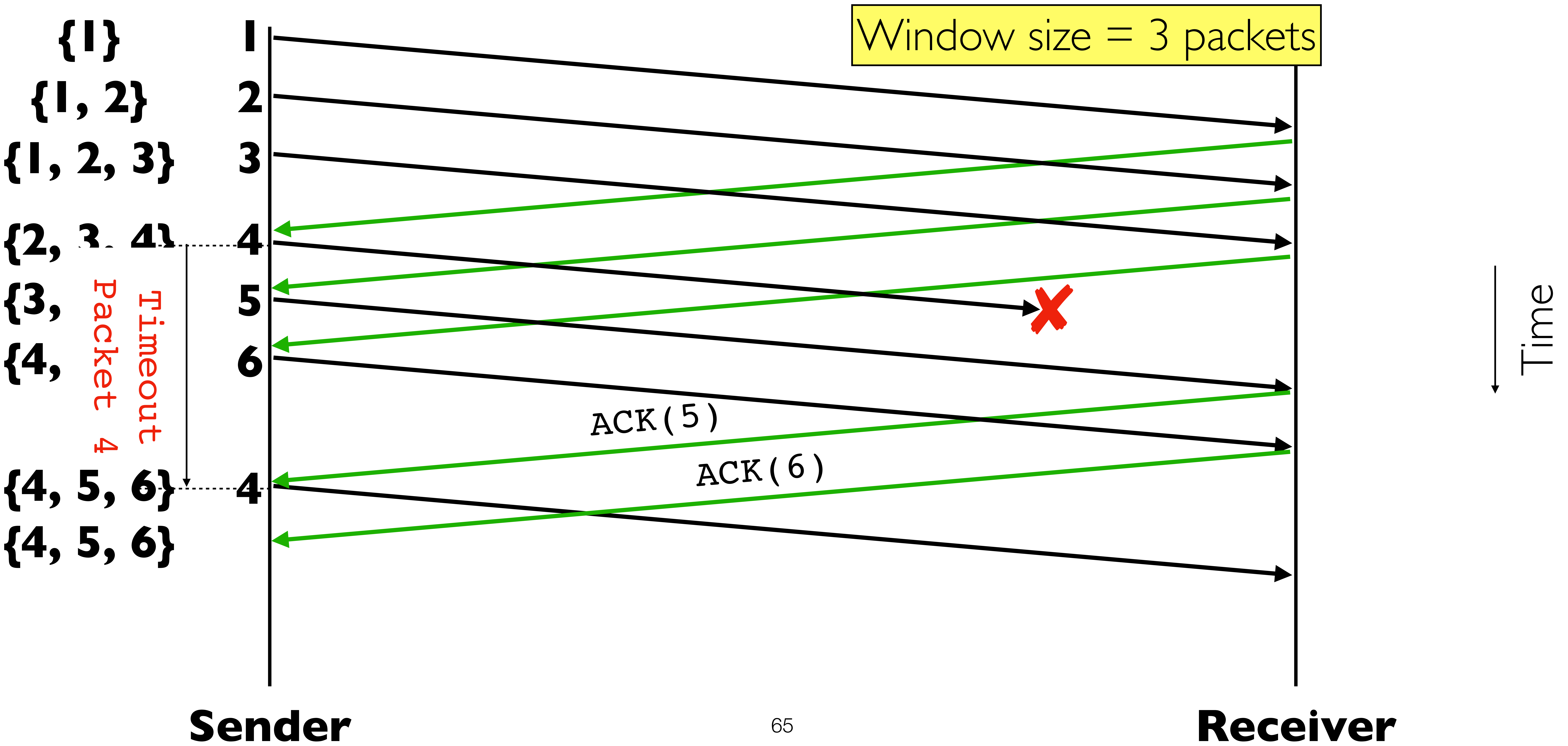
SR Example with Errors



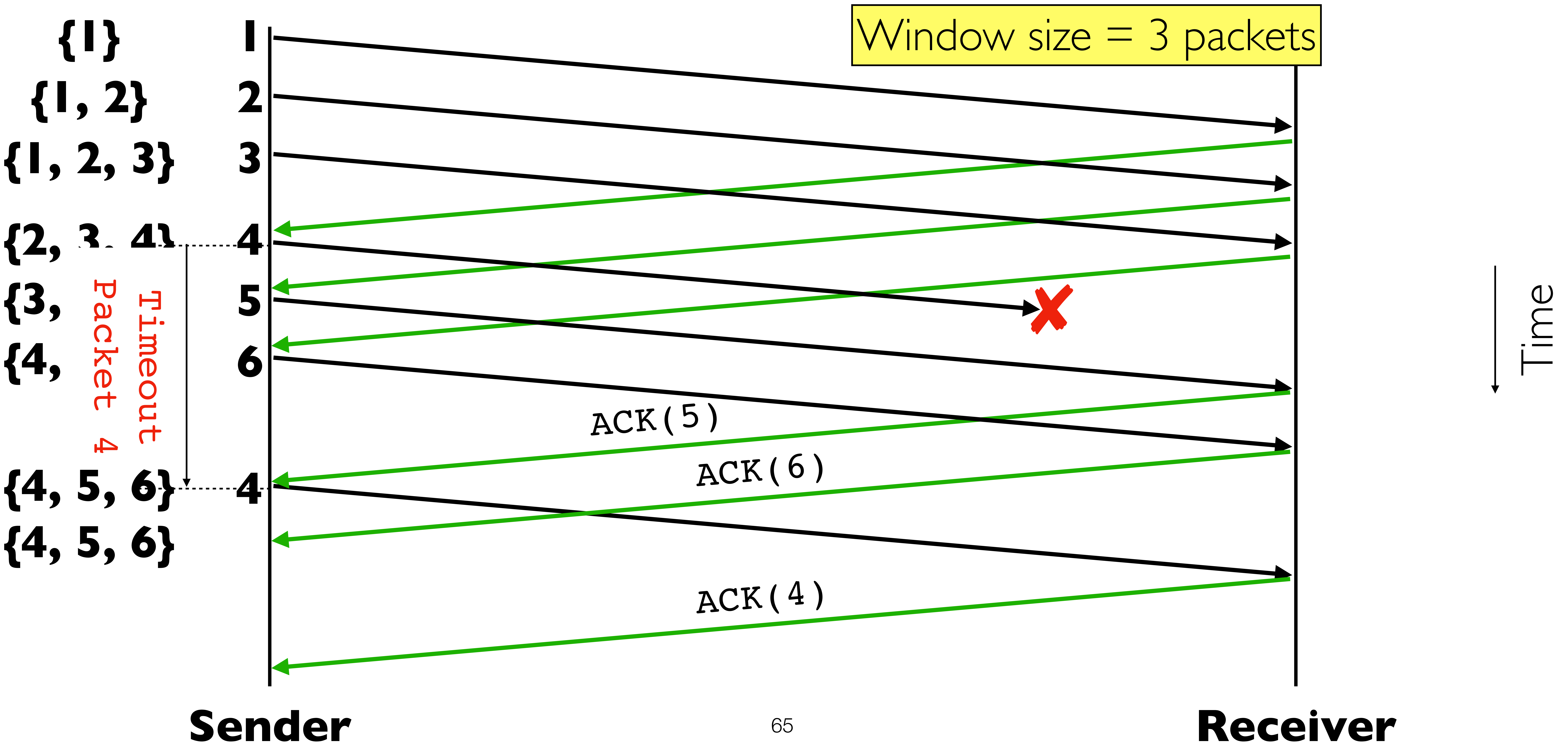
SR Example with Errors



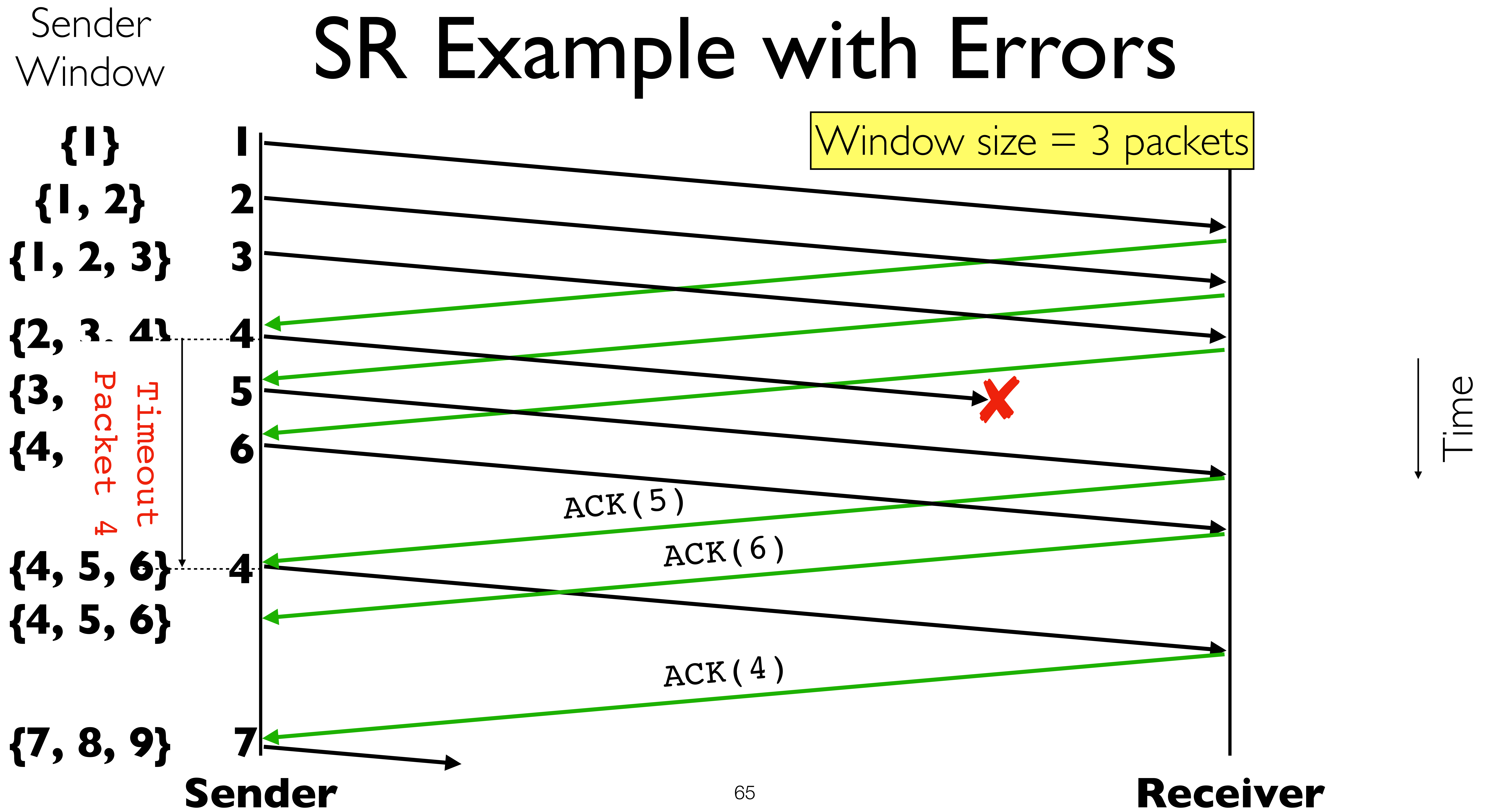
SR Example with Errors



SR Example with Errors



SR Example with Errors



GBN vs Selective Repeat

- When would GBN be better?
- When would SR be better?

Observations

Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough

Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough
- Sender has to buffer all unacknowledged packet, because they may require retransmission

Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough

- Sender has to buffer all unacknowledged packet, because they may require retransmission

- Receiver may be able to accept out-of-order packets, but only up to its buffer limits

Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough
- Sender has to buffer all unacknowledged packet, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits
- Implementation complexity depends on protocol details (GBN vs. SR)

Recap: Components of a solution

- **Checksums:** for error detection
- **Timers:** for loss detection
- **ACKs**
  - Cumulative
  - Selective
- **Sequence numbers:** tracking duplicates, windows
- **Sliding Windows:** for efficiency

Recap: Components of a solution

- **Checksums:** for error detection
- **Timers:** for loss detection
- **ACKs**
  - Cumulative
  - Selective
- **Sequence numbers:** tracking duplicates, windows
- **Sliding Windows:** for efficiency

- **Reliability protocols use the above to decide when and what to retransmit or acknowledge.**

What does TCP do?

- **Most of our previous tricks + a few differences**

What does TCP do?

- **Most of our previous tricks + a few differences**
  - Sequence numbers are *byte offsets*

What does TCP do?

- **Most of our previous tricks + a few differences**
  - Sequence numbers are *byte offsets*
  - Sender and receiver maintain a sliding window

What does TCP do?

- **Most of our previous tricks + a few differences**
  - Sequence numbers are *byte offsets*
  - Sender and receiver maintain a sliding window
  - Receiver sends cumulative acknowledgements (like GBN)

What does TCP do?

- **Most of our previous tricks + a few differences**
  - Sequence numbers are *byte offsets*
  - Sender and receiver maintain a sliding window
  - Receiver sends cumulative acknowledgements (like GBN)
  - Sender maintains a single retransmission timer

What does TCP do?

- **Most of our previous tricks + a few differences**
  - Sequence numbers are *byte offsets*
  - Sender and receiver maintain a sliding window
  - Receiver sends cumulative acknowledgements (like GBN)
  - Sender maintains a single retransmission timer
  - Receivers do not drop out-of-sequence packets (like SR)

What does TCP do?

- **Most of our previous tricks + a few differences**
  - Sequence numbers are *byte offsets*
  - Sender and receiver maintain a sliding window
  - Receiver sends cumulative acknowledgements (like GBN)
  - Sender maintains a single retransmission timer
  - Receivers do not drop out-of-sequence packets (like SR)
  - Introduces *fast retransmit:* optimization that uses duplicate ACKs to trigger early retransmissions (next class)

What does TCP do?

- **Most of our previous tricks + a few differences**
  - Sequence numbers are *byte offsets*
  - Sender and receiver maintain a sliding window
  - Receiver sends cumulative acknowledgements (like GBN)
  - Sender maintains a single retransmission timer
  - Receivers do not drop out-of-sequence packets (like SR)
  - Introduces *fast retransmit:* optimization that uses duplicate ACKs to trigger early retransmissions (next class)
  - Introduces timeout estimation algorithms (next time)

Next Time

- **TCP**
  - Reliability
  - Congestion control