TCP Basics

CPSC 433/533, Spring 2021
Anurag Khandelwal

Question from Last Time

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**
  - Has stumped many, but found two possible (& reasonable) responses:

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**
  - Has stumped many, but found two possible (& reasonable) responses:
  - Breaking ***layer agnosticism*** in this case makes things simpler for design of transport layer: don't need *all* transport protocols to put protocol identifier in the first few bytes

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**
  - Has stumped many, but found two possible (& reasonable) responses:
  - Breaking ***layer agnosticism*** in this case makes things simpler for design of transport layer: don't need *all* transport protocols to put protocol identifier in the first few bytes
    - Think encrypted transport protocols

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**
  - Has stumped many, but found two possible (& reasonable) responses:
  - Breaking ***layer agnosticism*** in this case makes things simpler for design of transport layer: don't need *all* transport protocols to put protocol identifier in the first few bytes
    - Think encrypted transport protocols
    - IP is the narrow waist, i.e., everyone understands IP, so why not put it there?

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**
  - Has stumped many, but found two possible (& reasonable) responses:
  - Breaking ***layer agnosticism*** in this case makes things simpler for design of transport layer: don't need *all* transport protocols to put protocol identifier in the first few bytes
    - Think encrypted transport protocols
    - IP is the narrow waist, i.e., everyone understands IP, so why not put it there?
  - Protocol field in IP header permits protocol-based ***in-network action*** (e.g., QoS, filtering, etc.)

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**
  - Has stumped many, but found two possible (& reasonable) responses:
  - Breaking ***layer agnosticism*** in this case makes things simpler for design of transport layer: don't need *all* transport protocols to put protocol identifier in the first few bytes
    - Think encrypted transport protocols
    - IP is the narrow waist, i.e., everyone understands IP, so why not put it there?
  - Protocol field in IP header permits protocol-based ***in-network action*** (e.g., QoS, filtering, etc.)
- **But definitely a case where layering abstraction broken for practicality!**

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**
  - Has stumped many, but found two possible (& reasonable) responses:
  - Breaking ***layer agnosticism*** in this case makes things simpler for design of transport layer: don't need *all* transport protocols to put protocol identifier in the first few bytes
    - Think encrypted transport protocols
    - IP is the narrow waist, i.e., everyone understands IP, so why not put it there?
  - Protocol field in IP header permits protocol-based ***in-network action*** (e.g., QoS, filtering, etc.)
- **But definitely a case where layering abstraction broken for practicality!**
  - Not the only one either, e.g., think 5-tuple (srcIP, srcPort, dstIP, dstPort, protocol)

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**
  - Has stumped many, but found two possible (& reasonable) responses:
  - Breaking ***layer agnosticism*** in this case makes things simpler for design of transport layer: don't need *all* transport protocols to put protocol identifier in the first few bytes
    - Think encrypted transport protocols
    - IP is the narrow waist, i.e., everyone understands IP, so why not put it there?
  - Protocol field in IP header permits protocol-based ***in-network action*** (e.g., QoS, filtering, etc.)
- **But definitely a case where layering abstraction broken for practicality!**
  - Not the only one either, e.g., think 5-tuple (srcIP, srcPort, dstIP, dstPort, protocol)
  - TCP/IP has been historically co-designed, so the lines are often blurry between them

Question from Last Time

- **Why have protocol field in IP header, and not as the first (few) byte(s) in the next header?**
  - Has stumped many, but found two possible (& reasonable) responses:
  - Breaking ***layer agnosticism*** in this case makes things simpler for design of transport layer: don't need *all* transport protocols to put protocol identifier in the first few bytes
    - Think encrypted transport protocols
    - IP is the narrow waist, i.e., everyone understands IP, so why not put it there?
  - Protocol field in IP header permits protocol-based ***in-network action*** (e.g., QoS, filtering, etc.)
- **But definitely a case where layering abstraction broken for practicality!**
  - Not the only one either, e.g., think 5-tuple (srcIP, srcPort, dstIP, dstPort, protocol)
  - TCP/IP has been historically co-designed, so the lines are often blurry between them
  - Not necessarily the best idea, in hindsight!

What did we learn last time?

What did we learn last time?

- **Transport layer has two purposes:**
  - **Multiplexing/demultiplexing:** done using *ports* and *sockets*
  - Optional: **Reliable delivery**

What did we learn last time?

- **Transport layer has two purposes:**
  - Multiplexing/demultiplexing: done using *ports* and *sockets*
  - Optional: Reliable delivery
- **Reliable delivery involves many mechanisms**
  - Requires delicate design to get them to work together
  - TCP is the standard example of a reliable transport

Last Time: Components of Reliable Transport

- **Checksums:** for error detection
- **Timers:** for loss detection
- **Sliding Windows:** for efficiency
- **ACKs**
  - Cumulative
  - Selective
- **Sequence numbers:** tracking duplicates, windows
- **Retransmissions:**
  - Go-Back-N (GBN)
  - Selective Repeat

Last Time: Components of Reliable Transport

- **Checksums:** for error detection
- **Timers:** for loss detection
- **Sliding Windows:** for efficiency
- **ACKs**
  - Cumulative
  - Selective
- **Sequence numbers:** tracking duplicates, windows
- **Retransmissions:**
  - Go-Back-N (GBN)
  - Selective Repeat

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)

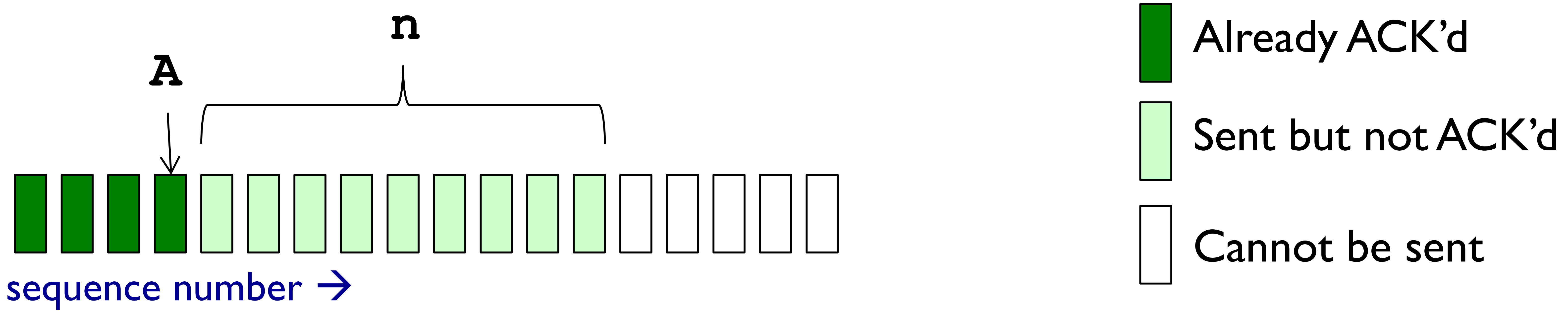
- Sender transmits up to *n* unacknowledged packets
- Receiver only accepts packets in order
  - Discards out-of-order packets (i.e., packets other than B+1)
- Receiver uses **cumulative acknowledgements**
  - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1st outstanding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

Go-Back-N (GBN)

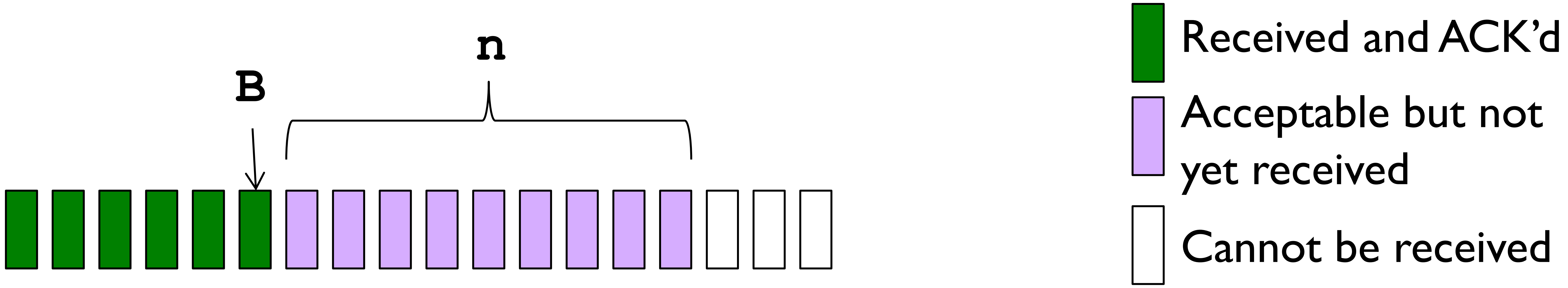
- Sender transmits up to *n* unacknowledged packets
- Receiver only accepts packets in order
  - Discards out-of-order packets (i.e., packets other than B+1)
- Receiver uses **cumulative acknowledgements**
  - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1st outstanding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

Sliding Window with GBN

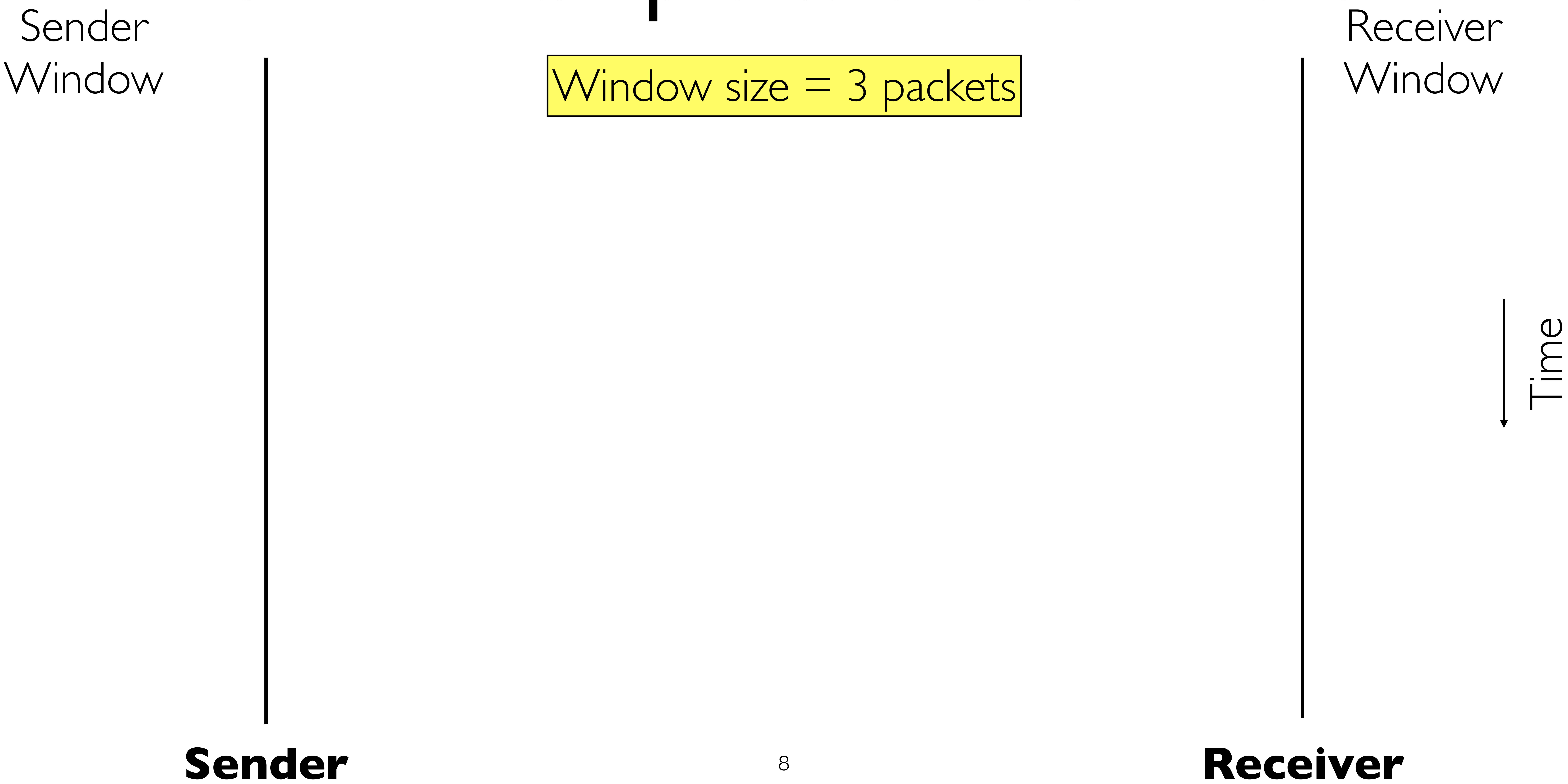
- Let A be the **last ACK'd packet of sender without gap**; then window of sender = {A+1, A+2, …, A+n}



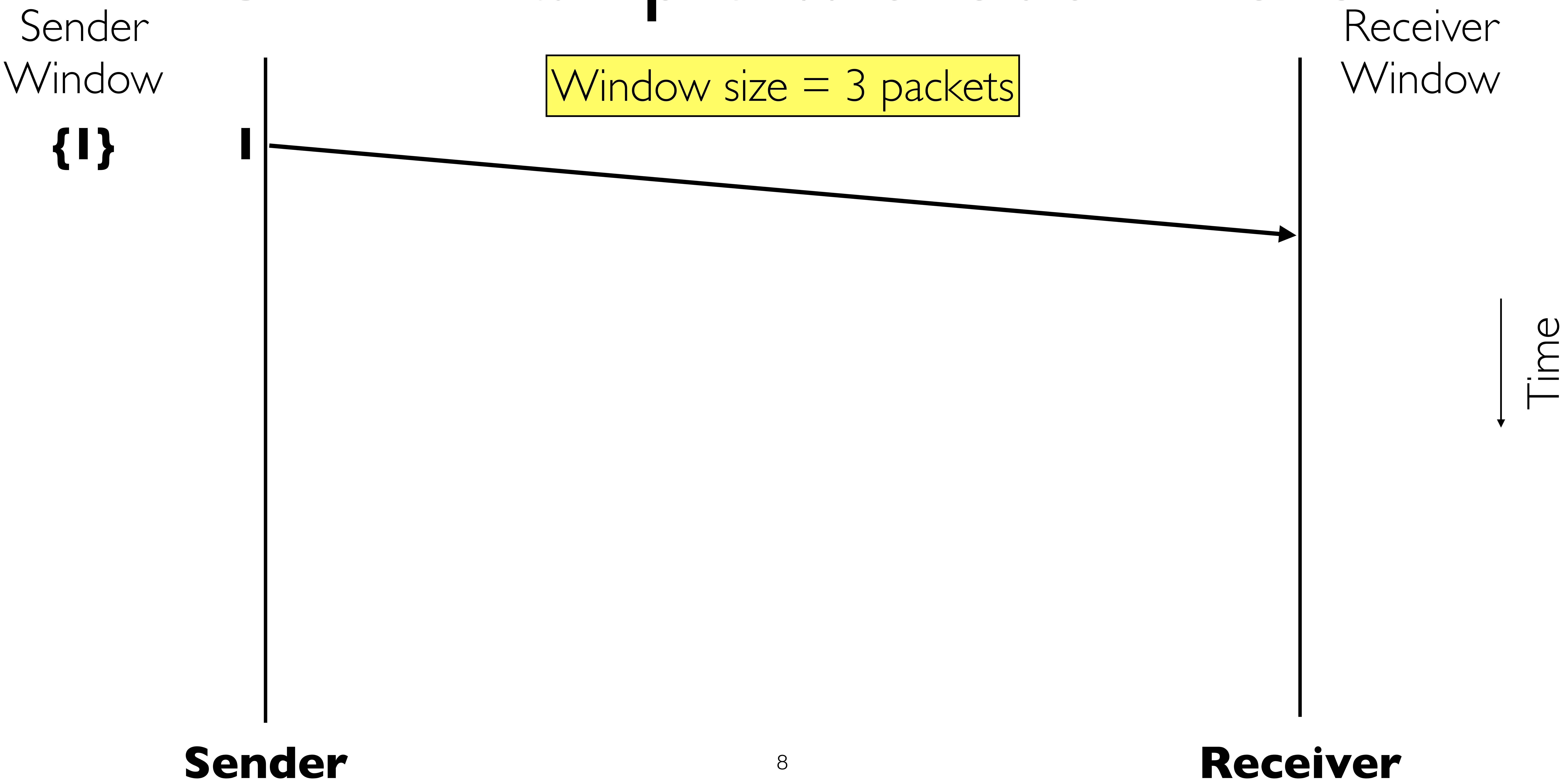
- Let B be the **last received packet without gap** by receiver; then window of receiver = {B+1, …, B+n}



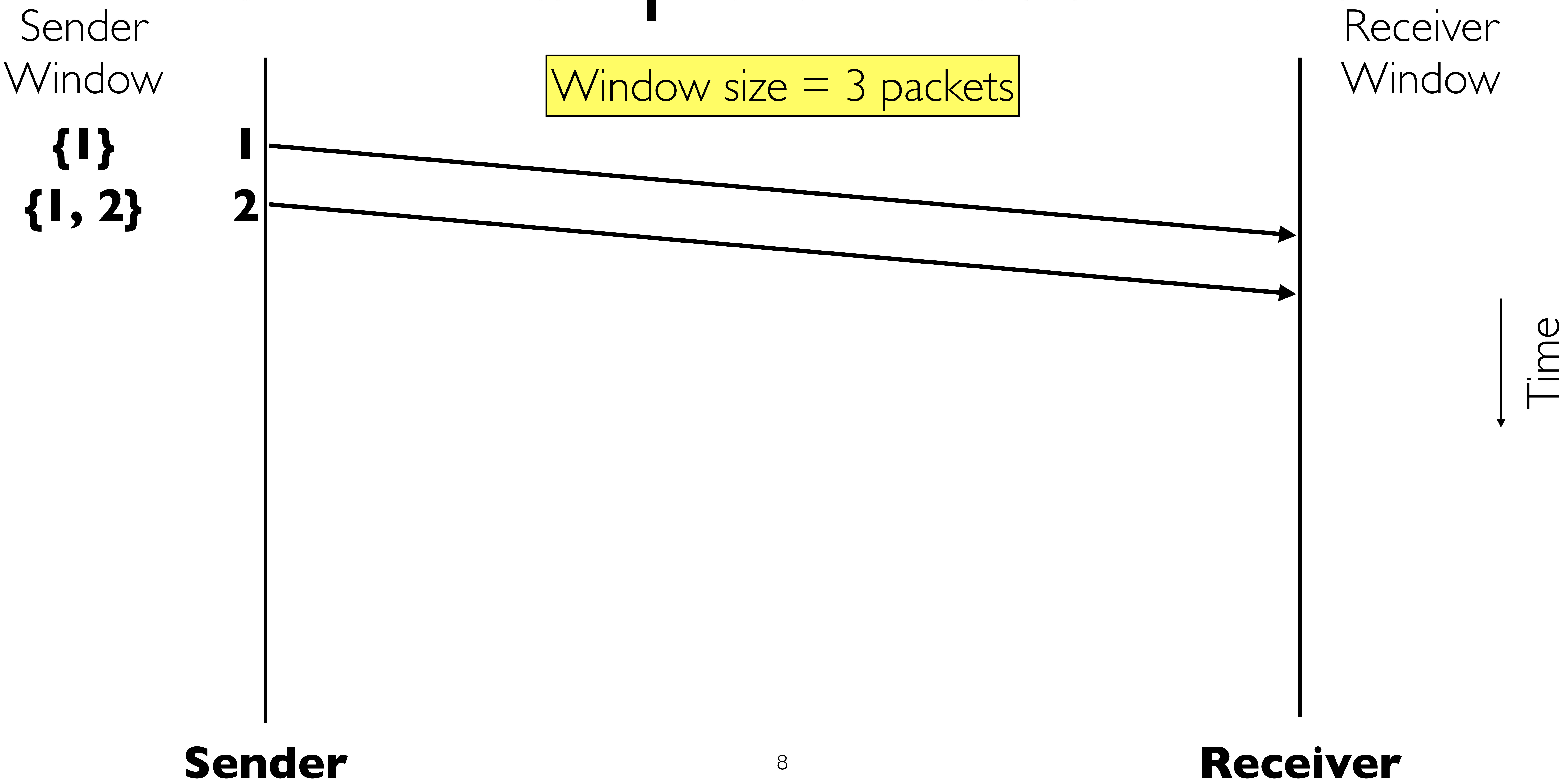
GBN Example without Errors



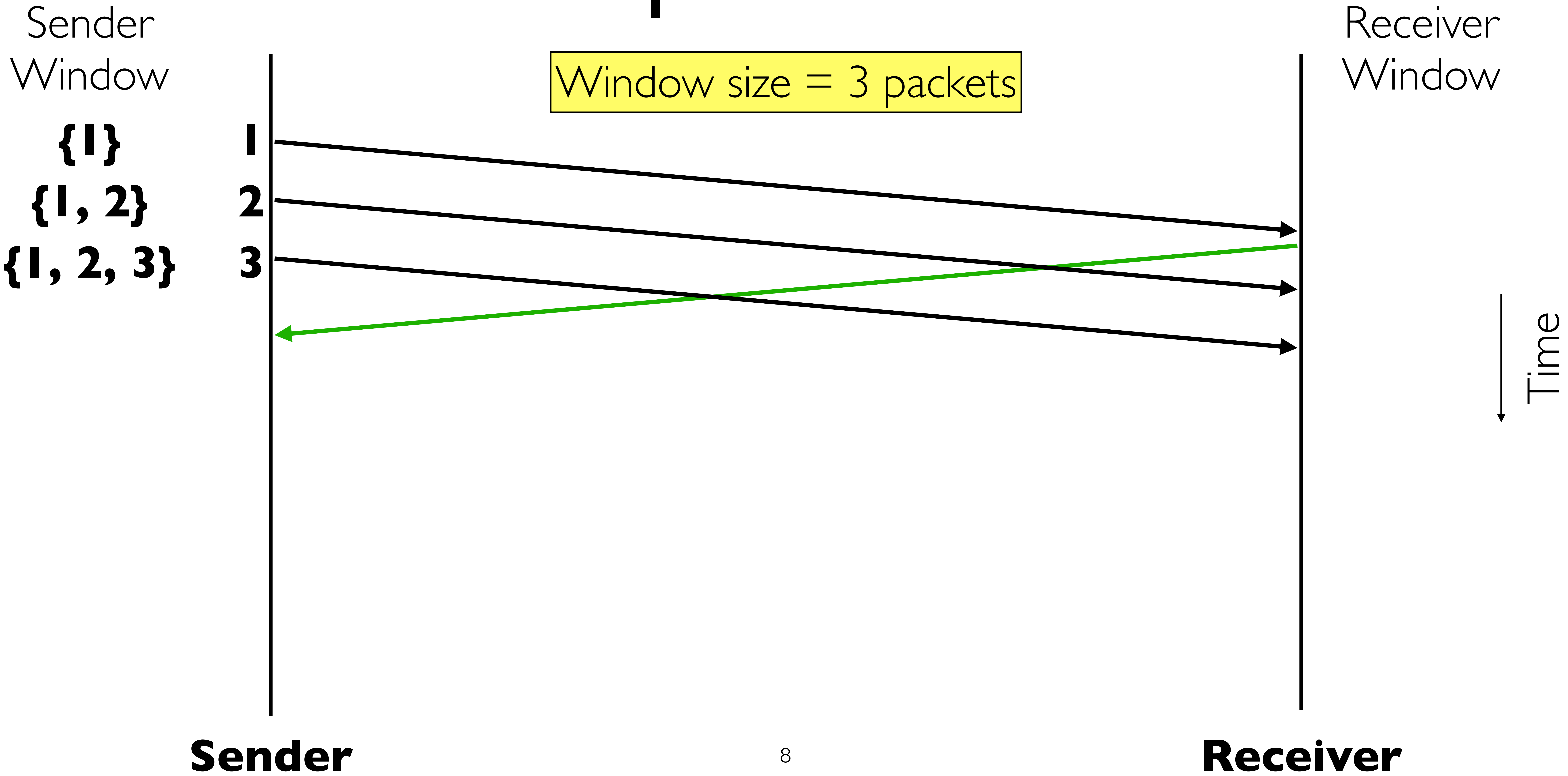
GBN Example without Errors



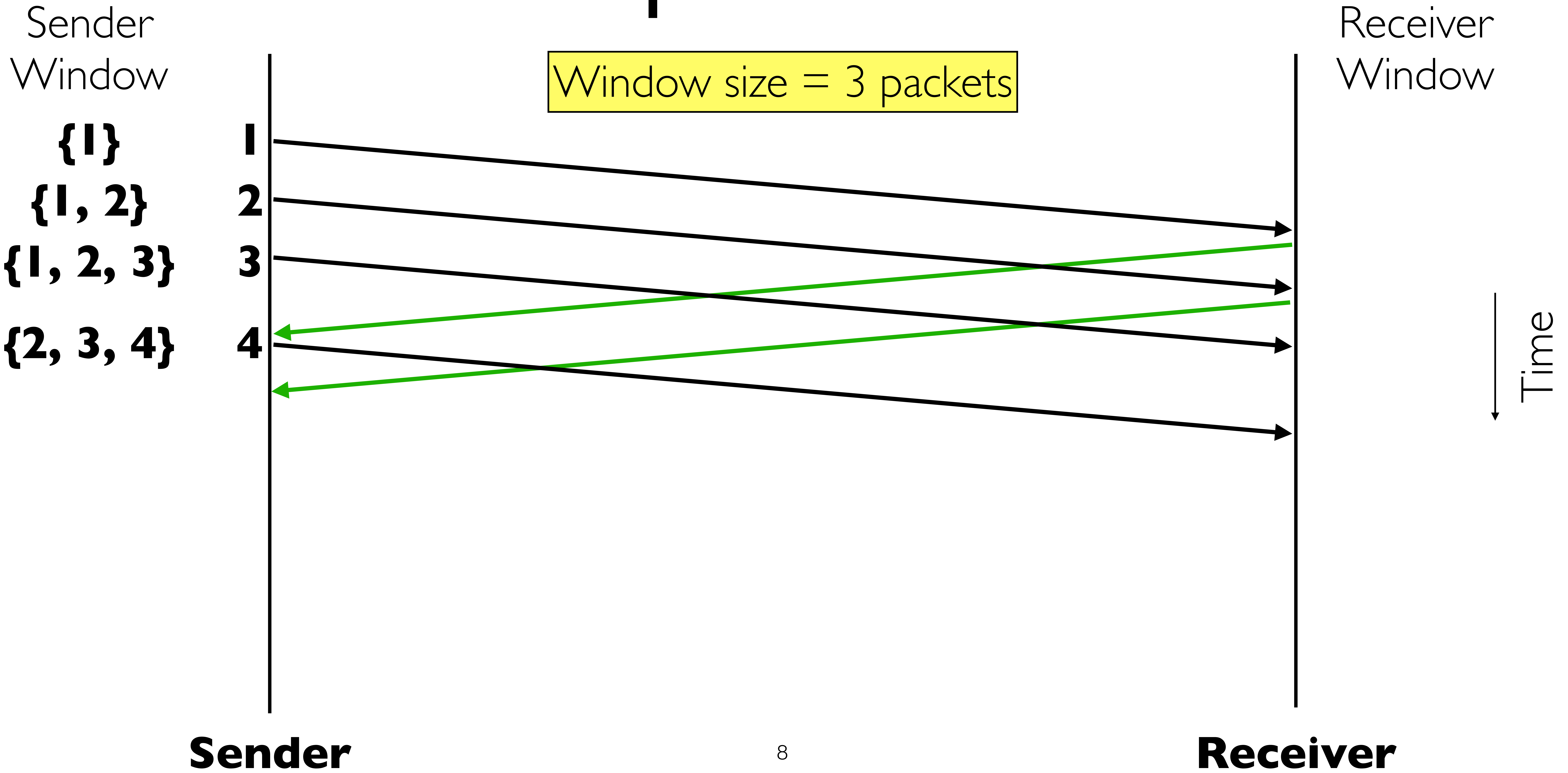
GBN Example without Errors



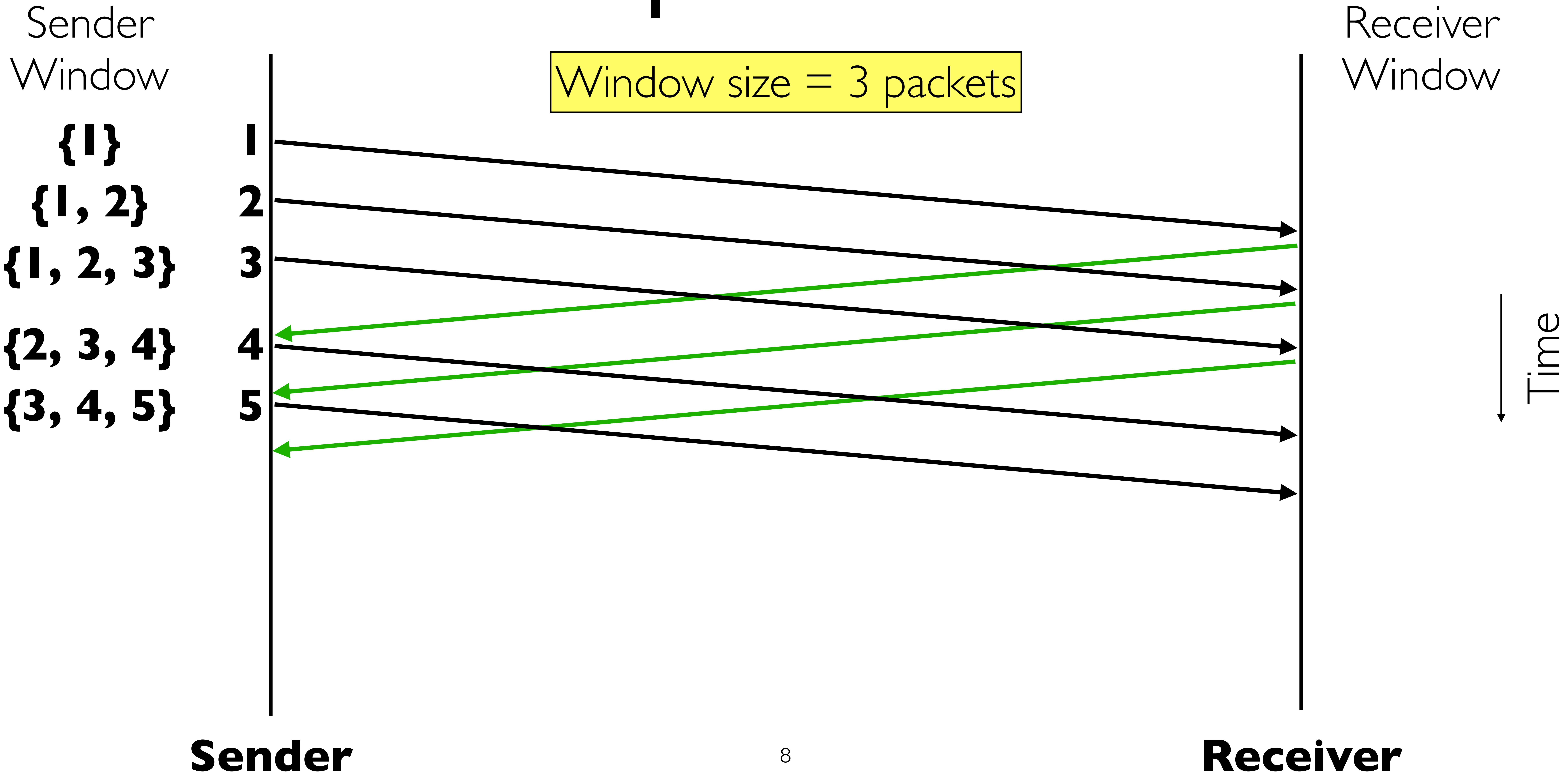
GBN Example without Errors



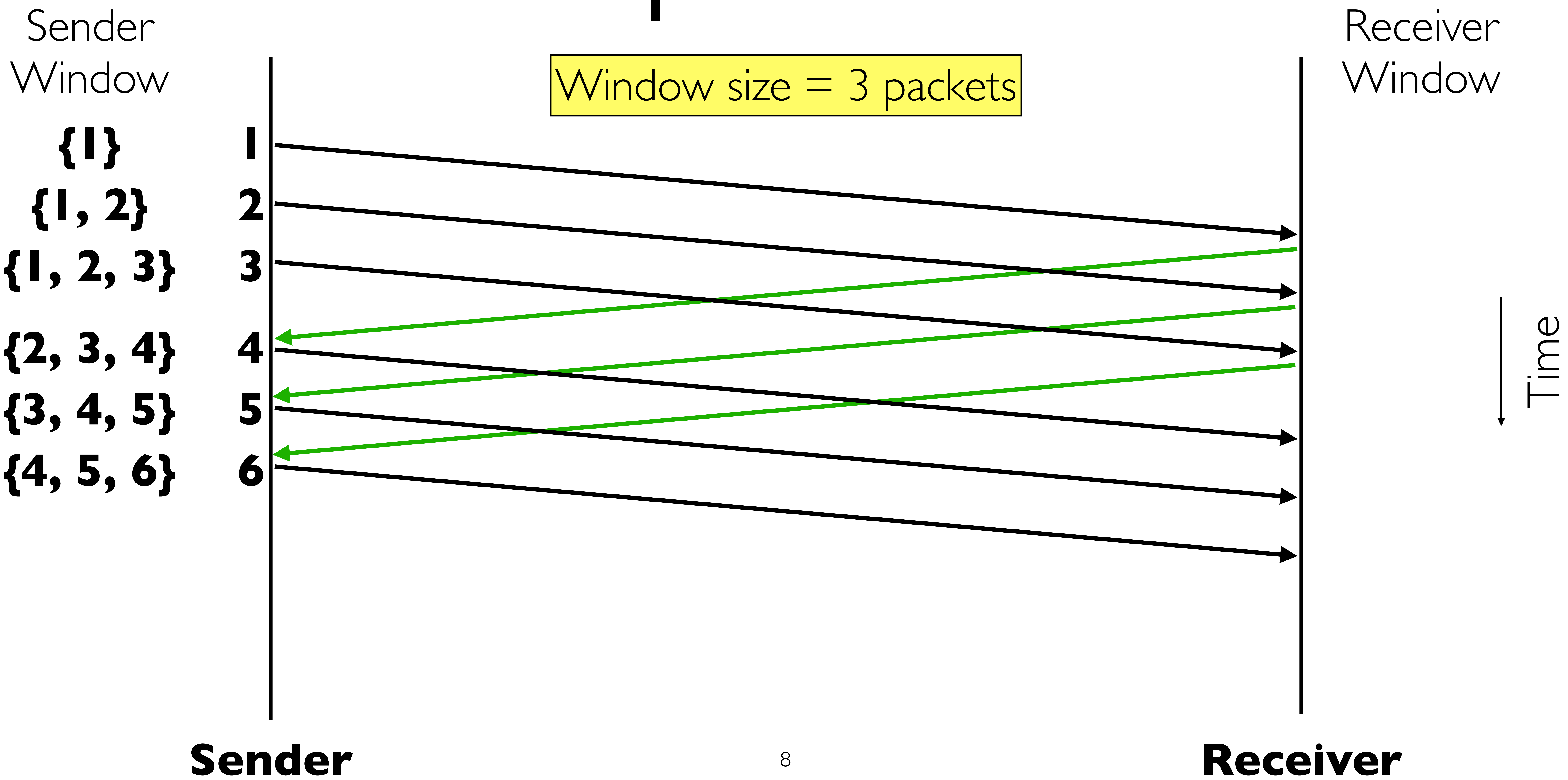
GBN Example without Errors



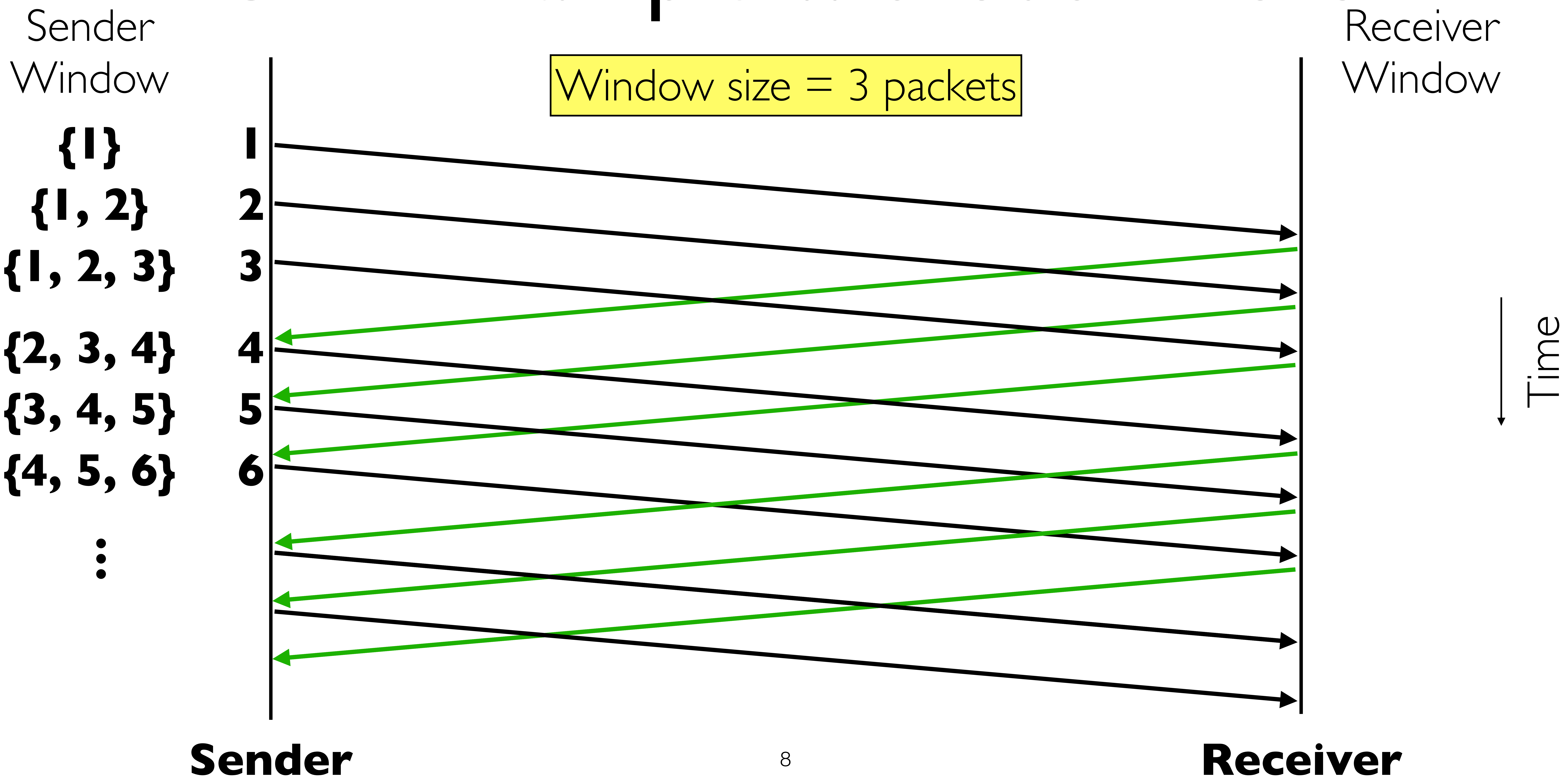
GBN Example without Errors



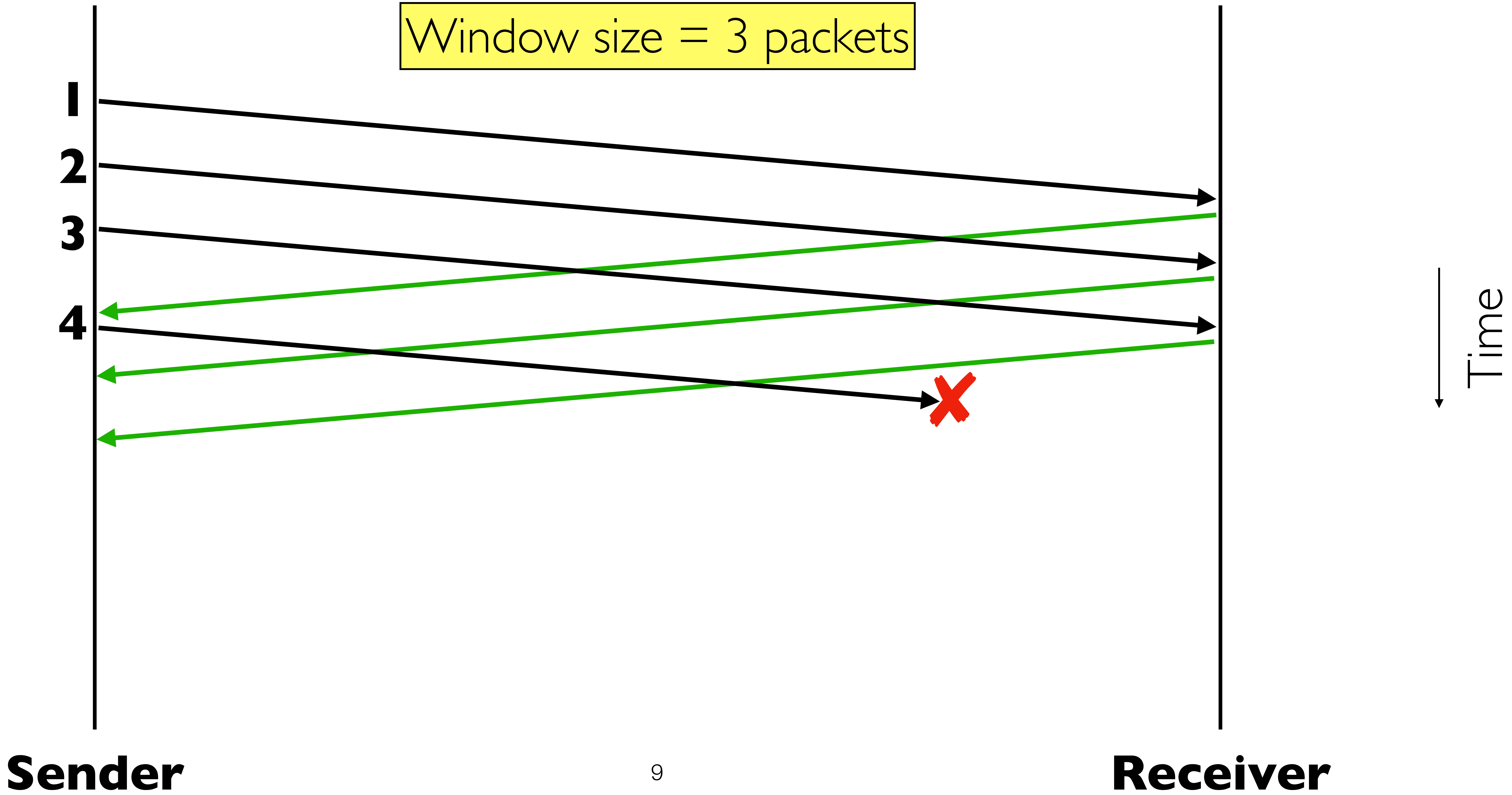
GBN Example without Errors



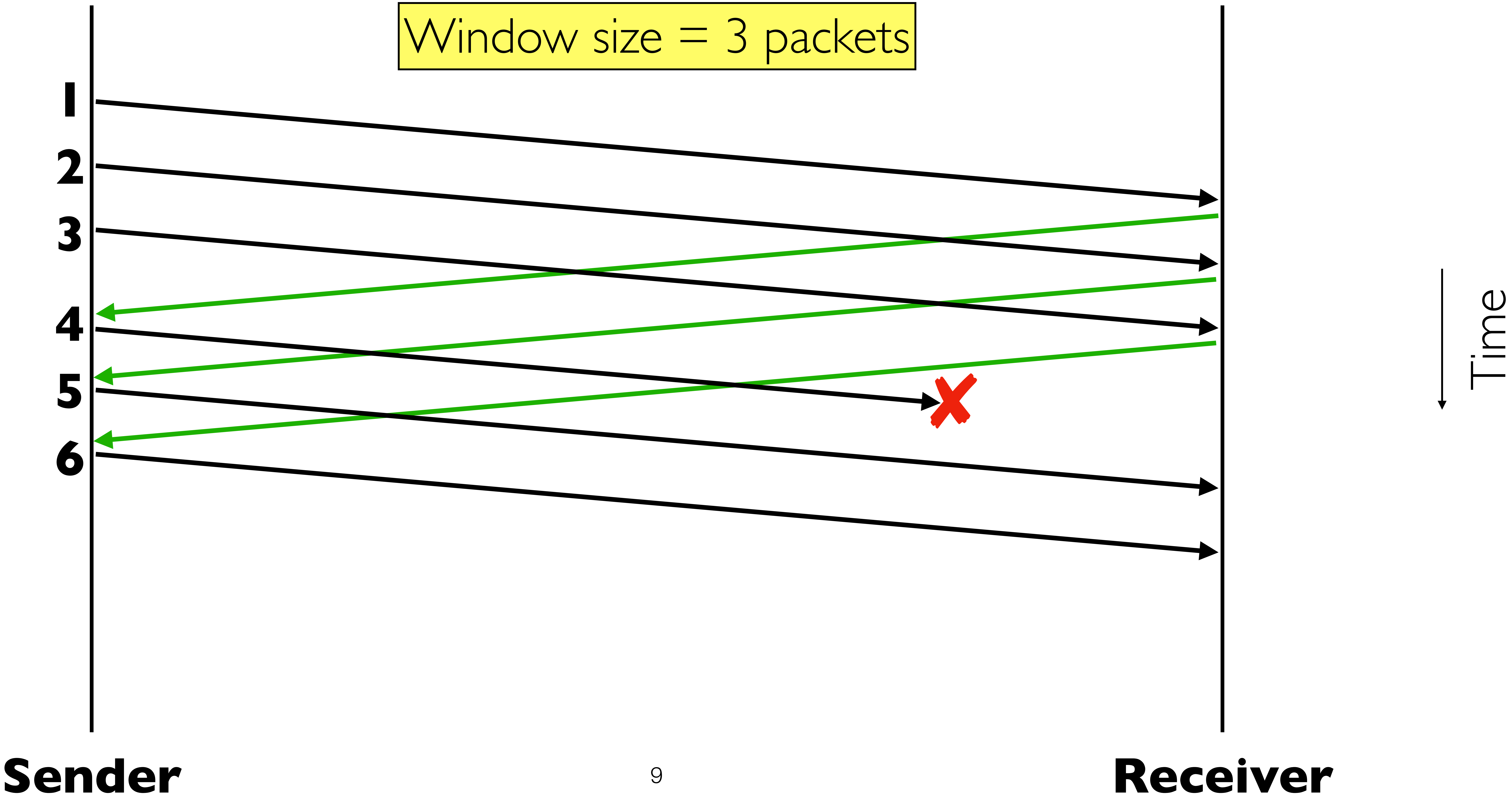
GBN Example without Errors



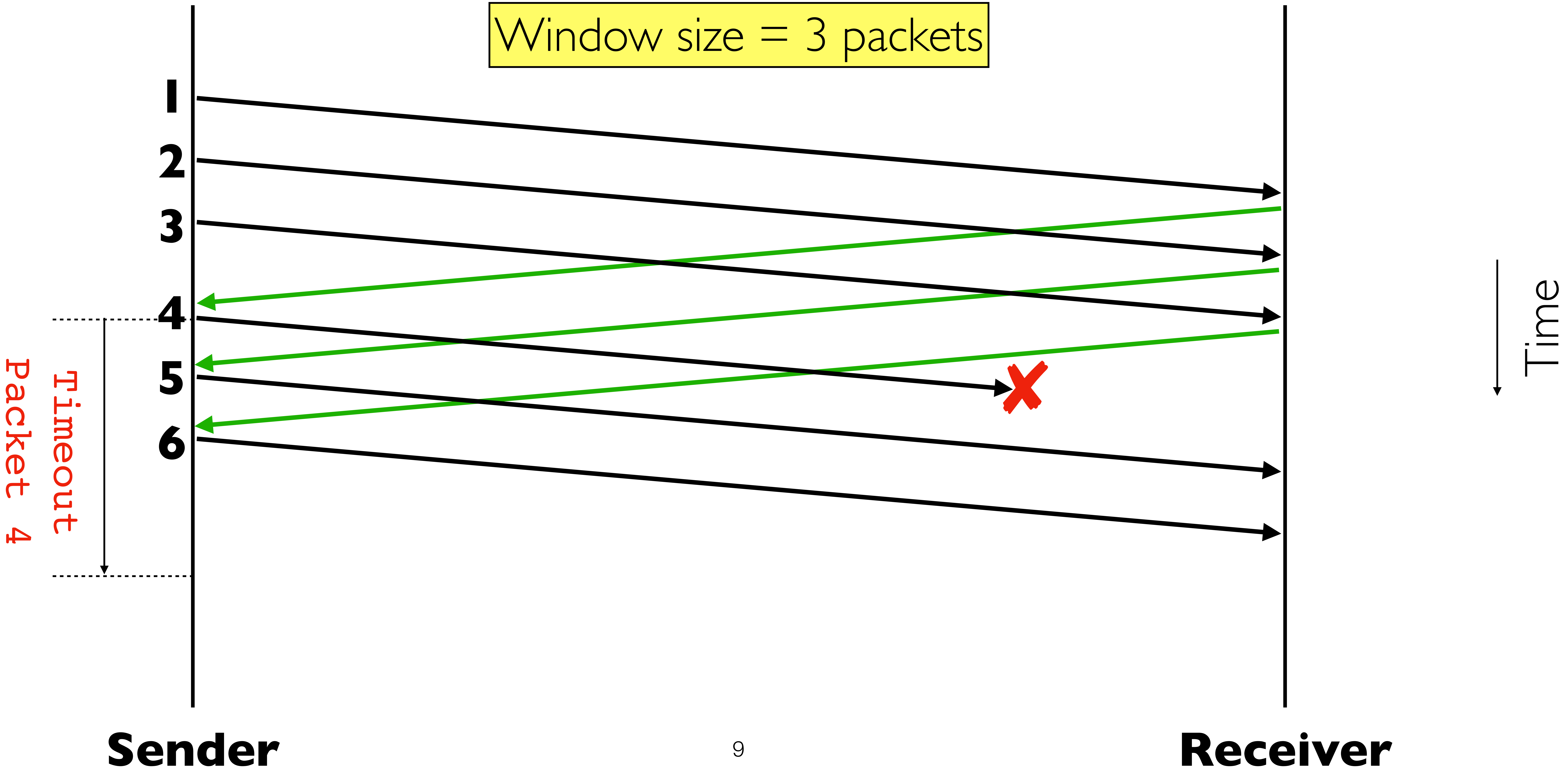
GBN Example with Errors



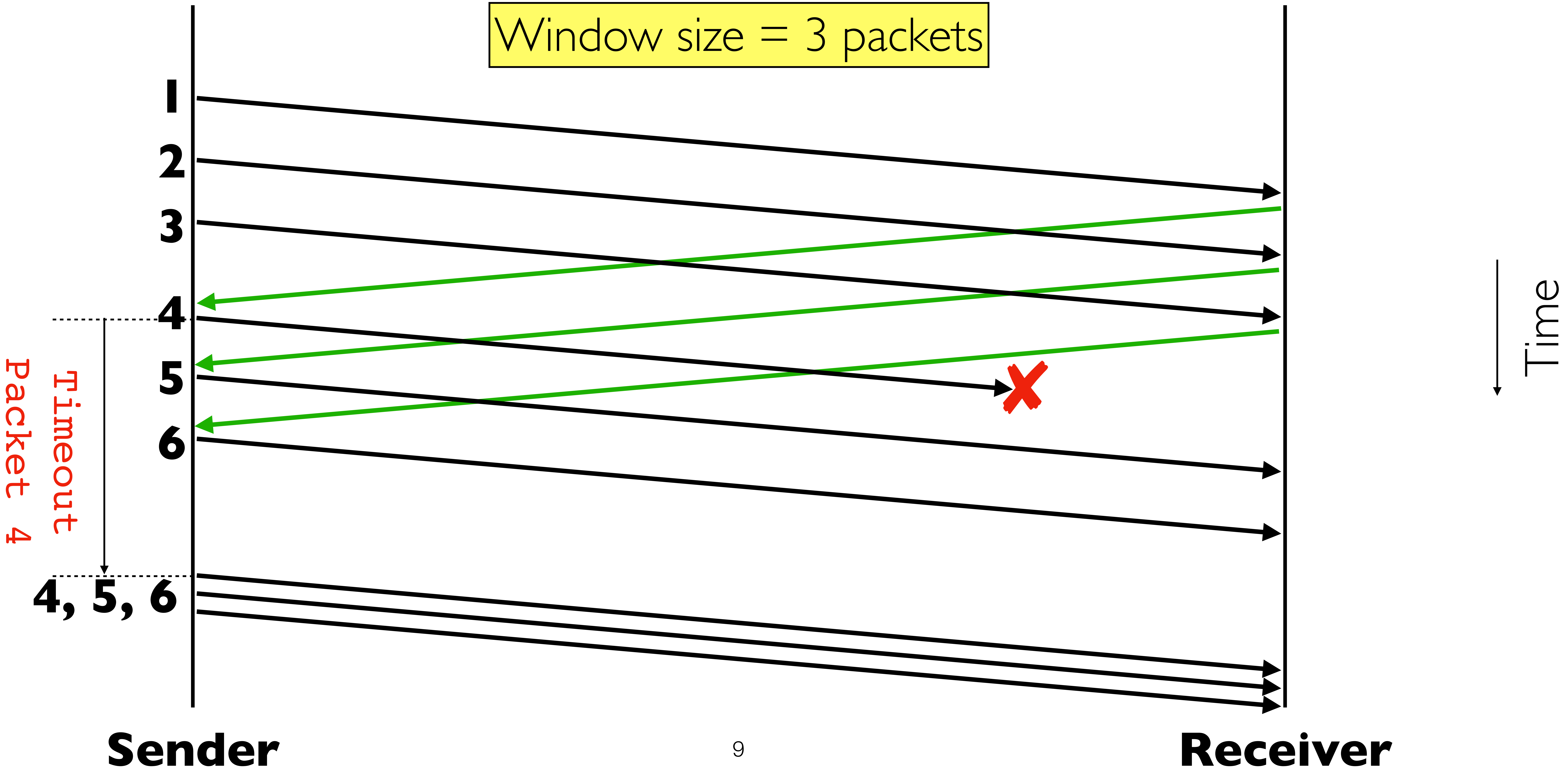
GBN Example with Errors



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

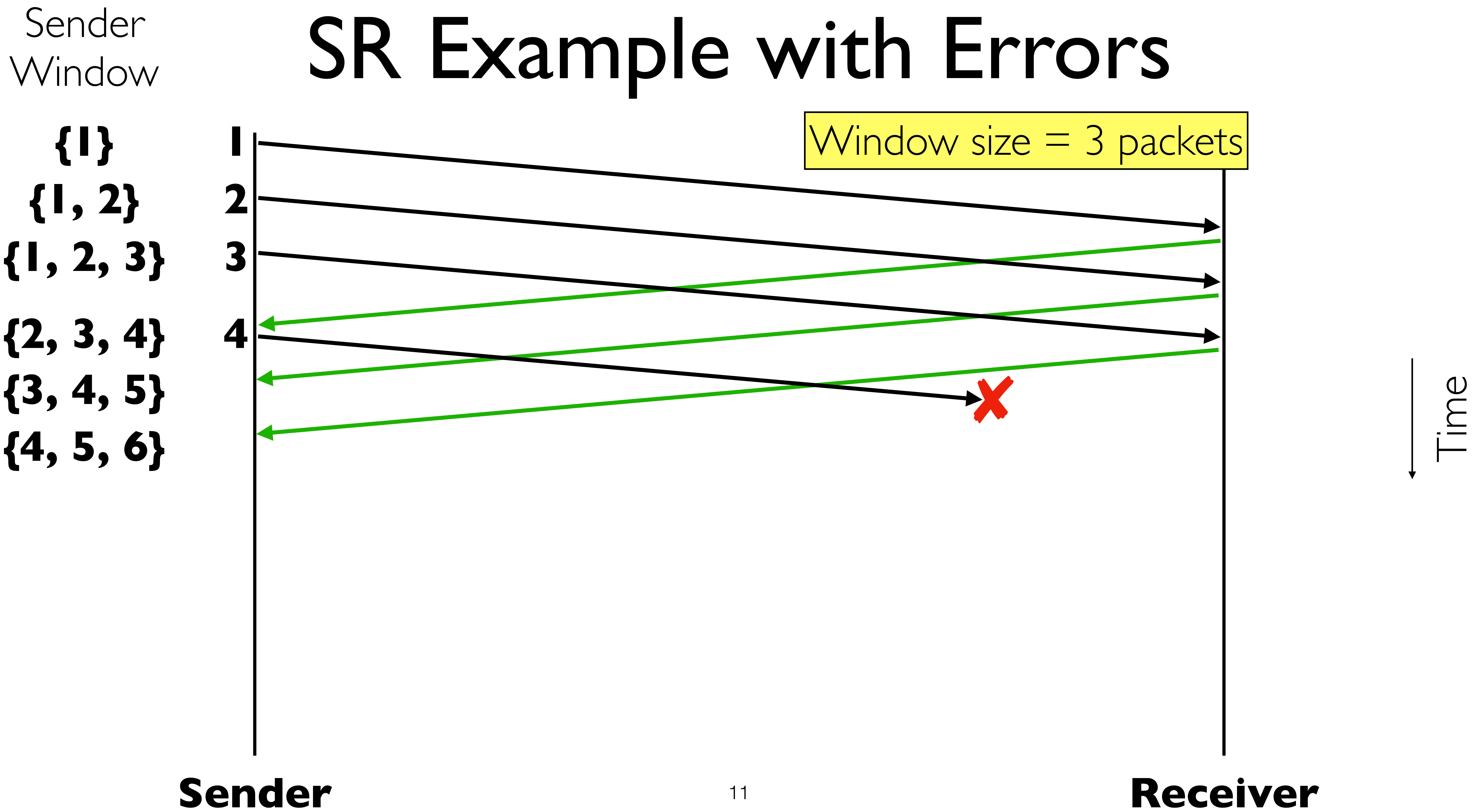
Selective Repeat

- 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

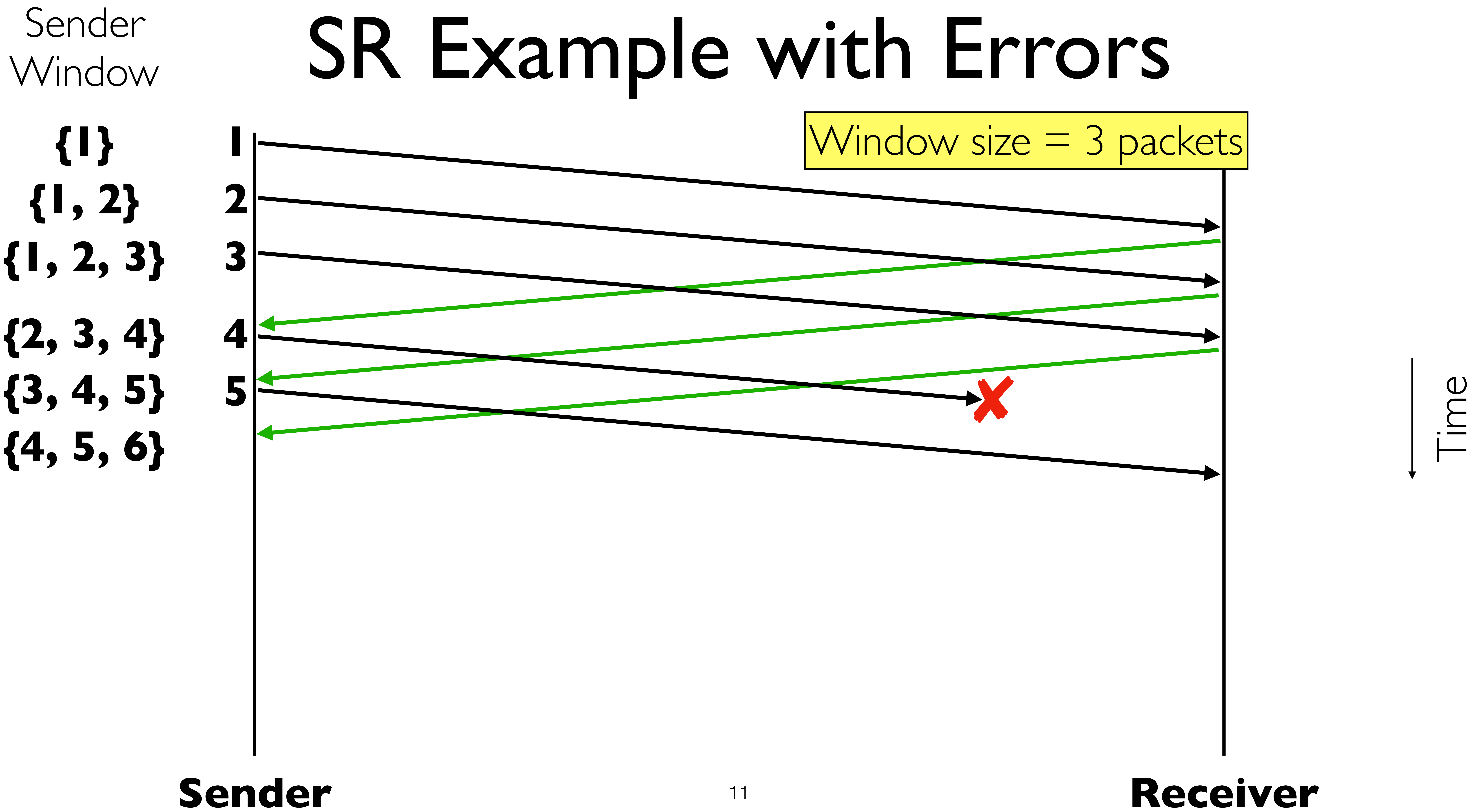
Selective Repeat

- 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!

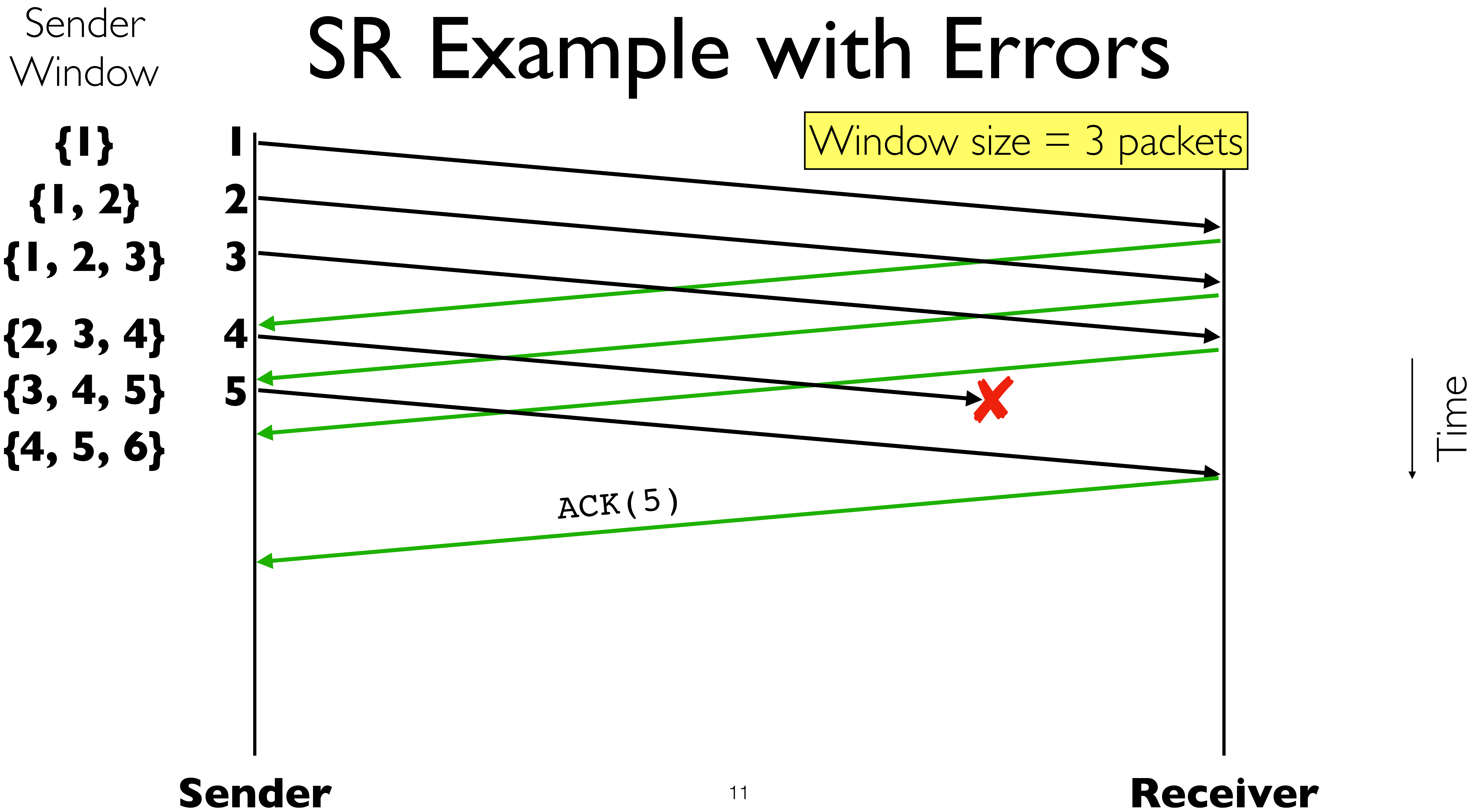
SR Example with Errors



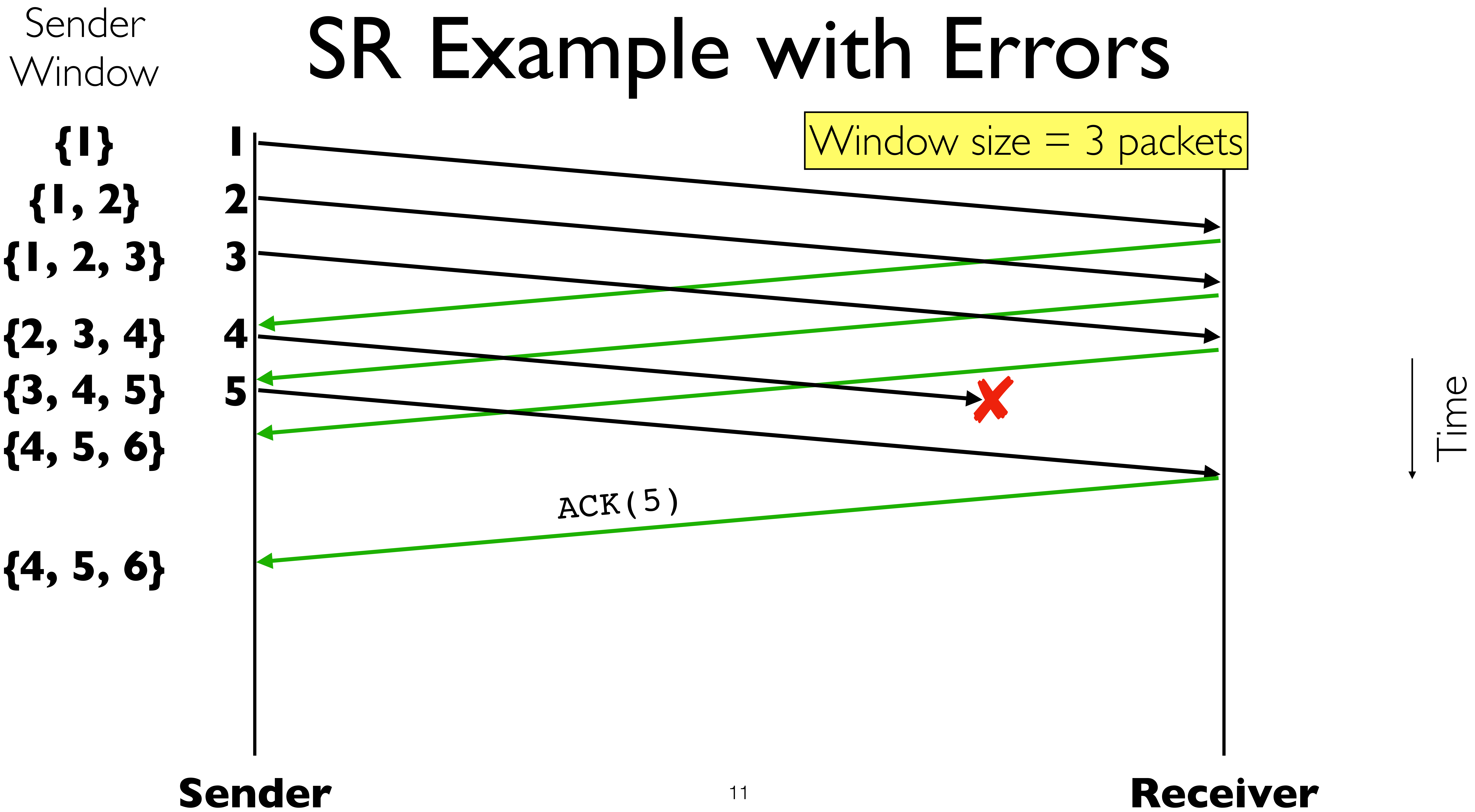
SR Example with Errors



SR Example with Errors

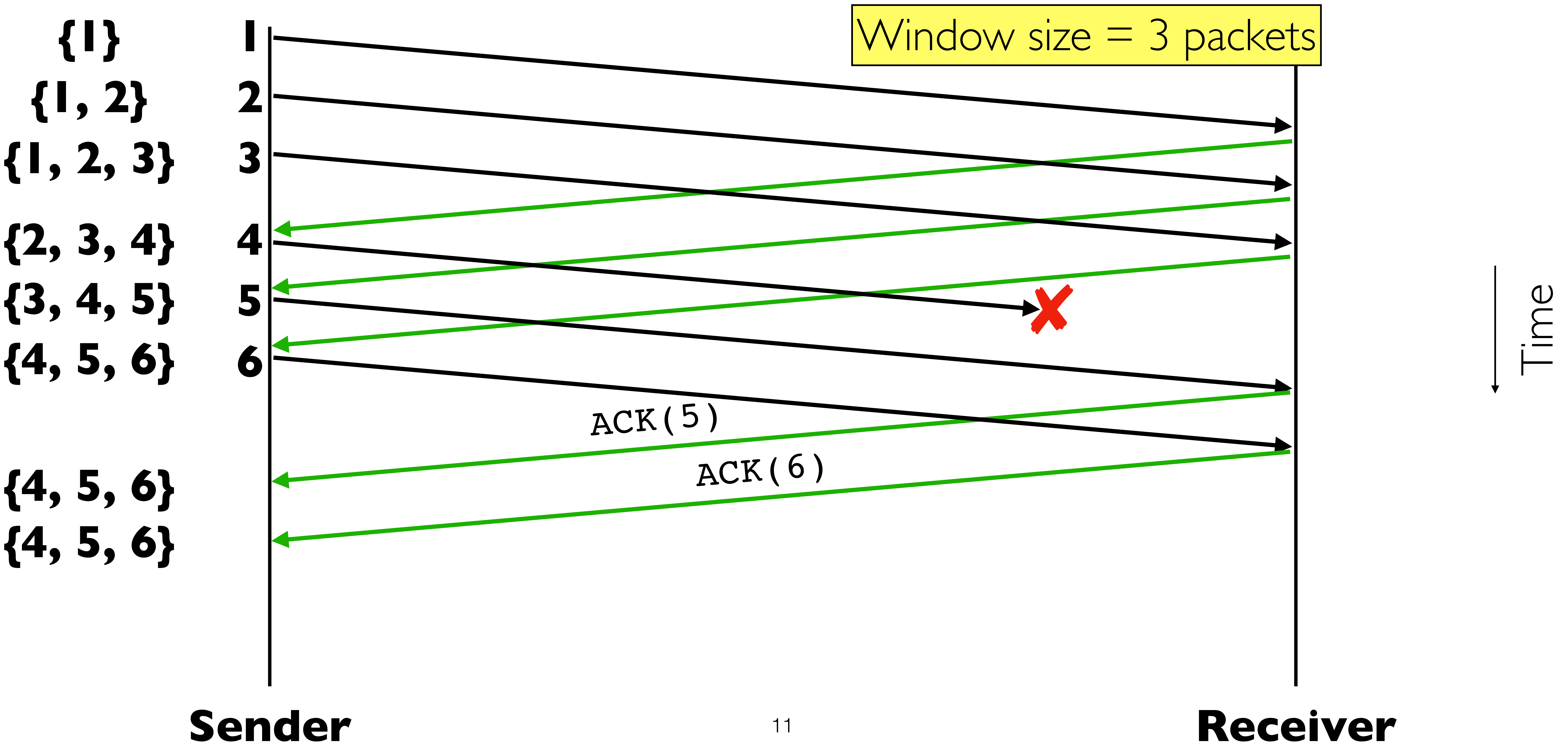


SR Example with Errors

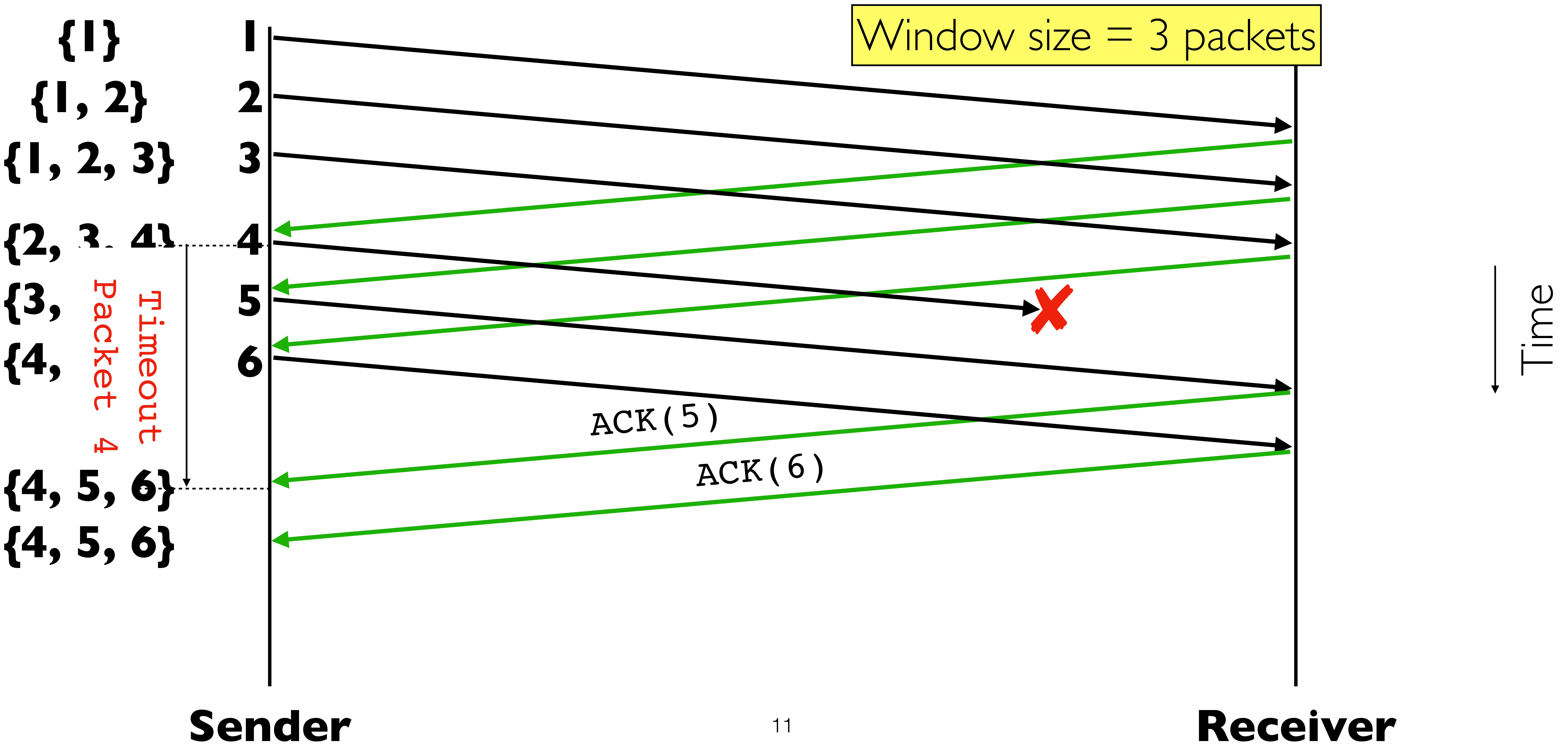


Sender Window

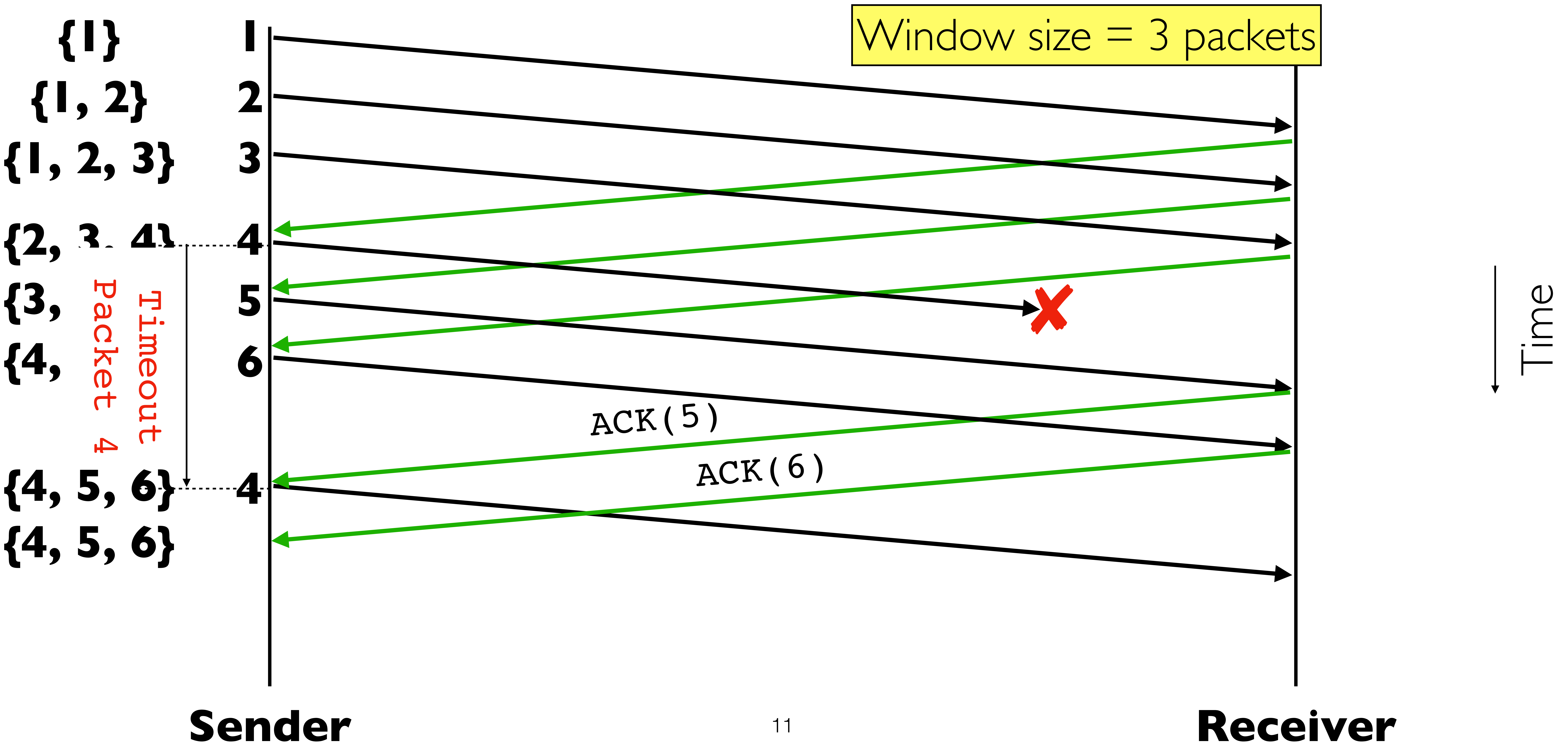
SR Example with Errors



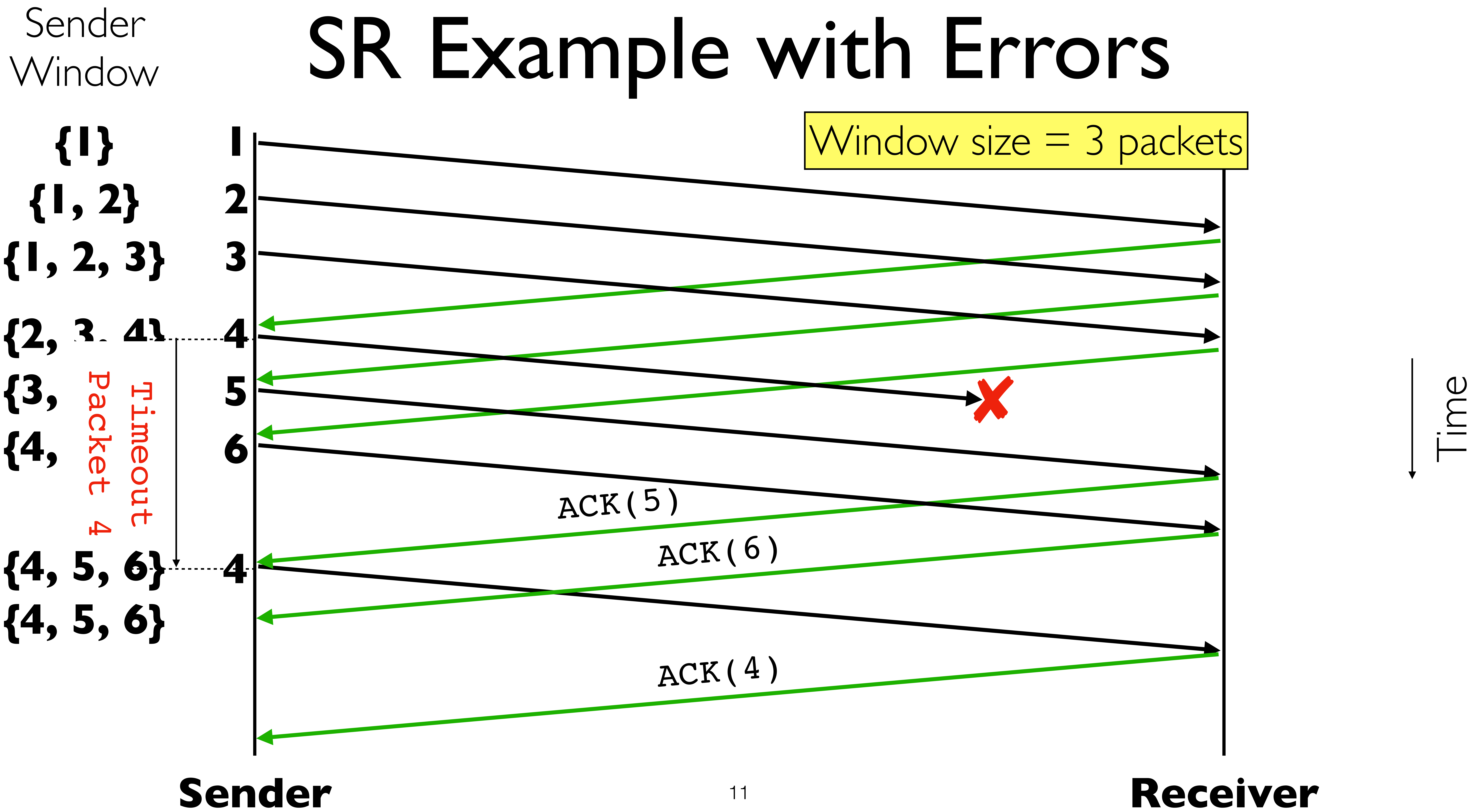
SR Example with Errors



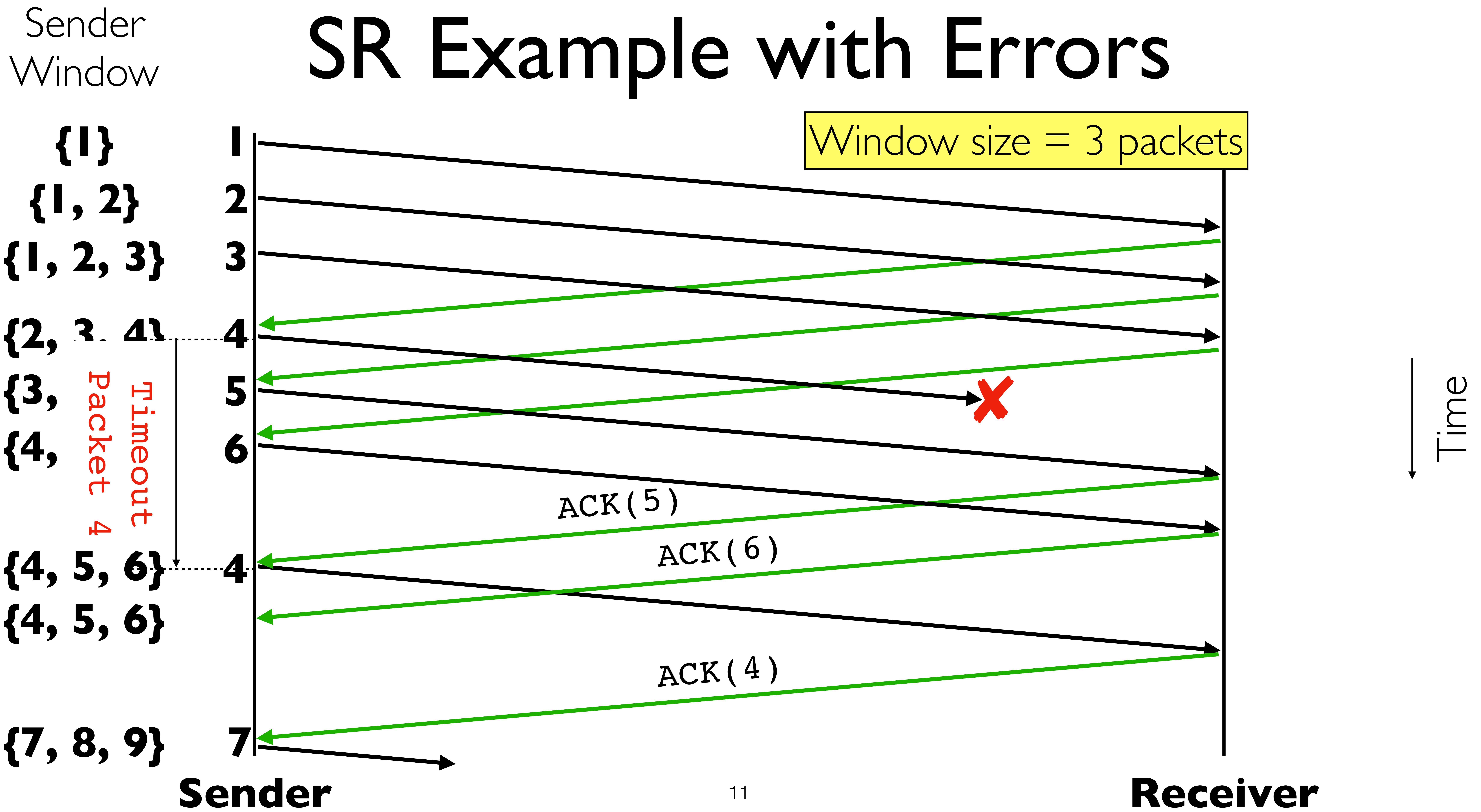
SR Example with Errors



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
- **Sliding Windows:** for efficiency
- **ACKs**
    - Cumulative
    - Selective
- **Sequence numbers:** tracking duplicates, windows

Recap: Components of a solution

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

Recap: Components of a solution

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

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

The TCP Abstraction

- **TCP delivers a *reliable*, *in-order* *byte-stream***
- ***Reliable:*** TCP resends lost packets (recursively)
  - Until it gives up and shuts down connection
- ***In-order:*** TCP only hands consecutive chunks of data to application
- **Byte-stream:** TCP assumes there is an incoming stream of data, and attempts to deliver it to the application

What Will We Cover Today?

What Will We Cover Today?

- **How TCP supports reliability**
  - A header driven approach!

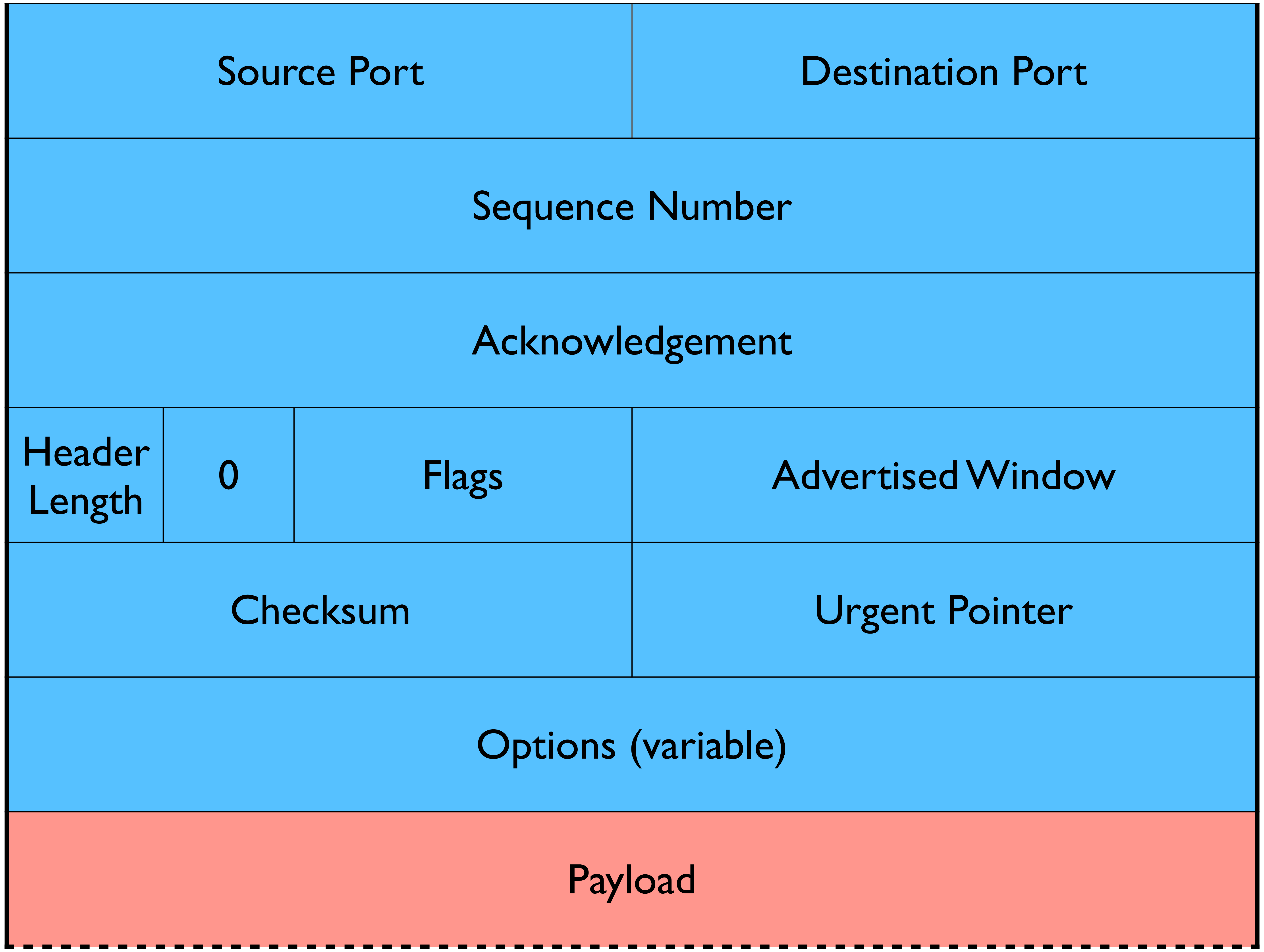
What Will We Cover Today?

- **How TCP supports reliability**
  - A header driven approach!
- **TCP is not a perfect design**
  - Probably wouldn't make exactly the same choices today

What Will We Cover Today?

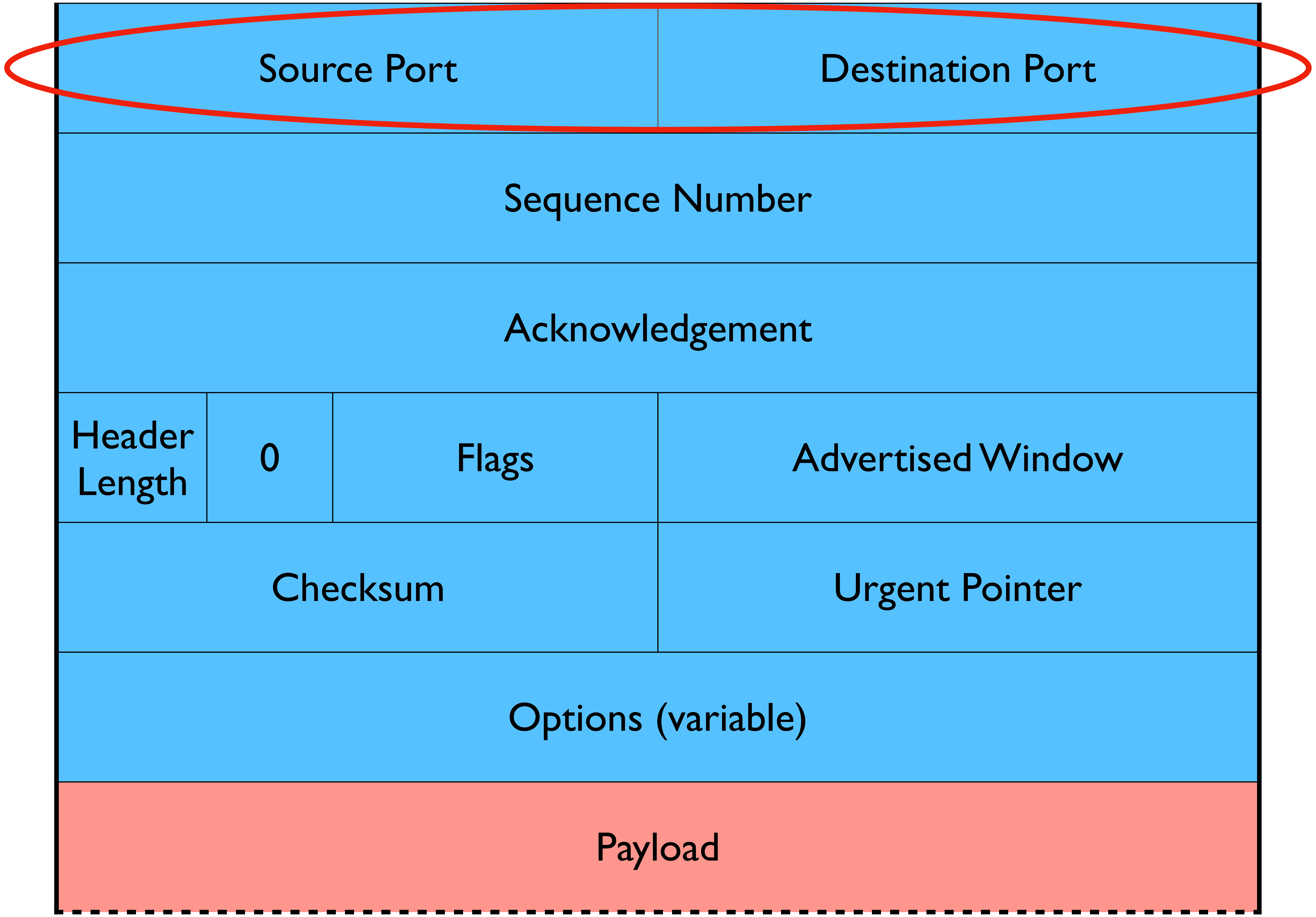
- **How TCP supports reliability**
  - A header driven approach!
- **TCP is not a perfect design**
  - Probably wouldn't make exactly the same choices today
- **But it is good enough**
  - And is a great example of sweating the details…
  - …and just happens to carry most of your traffic

TCP Header


Source Port
Destination Port
Sequence Number
Acknowledgement
Header Length
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Payload

TCP Header

Used to multiplex/demultiplex



What does TCP do for reliability?

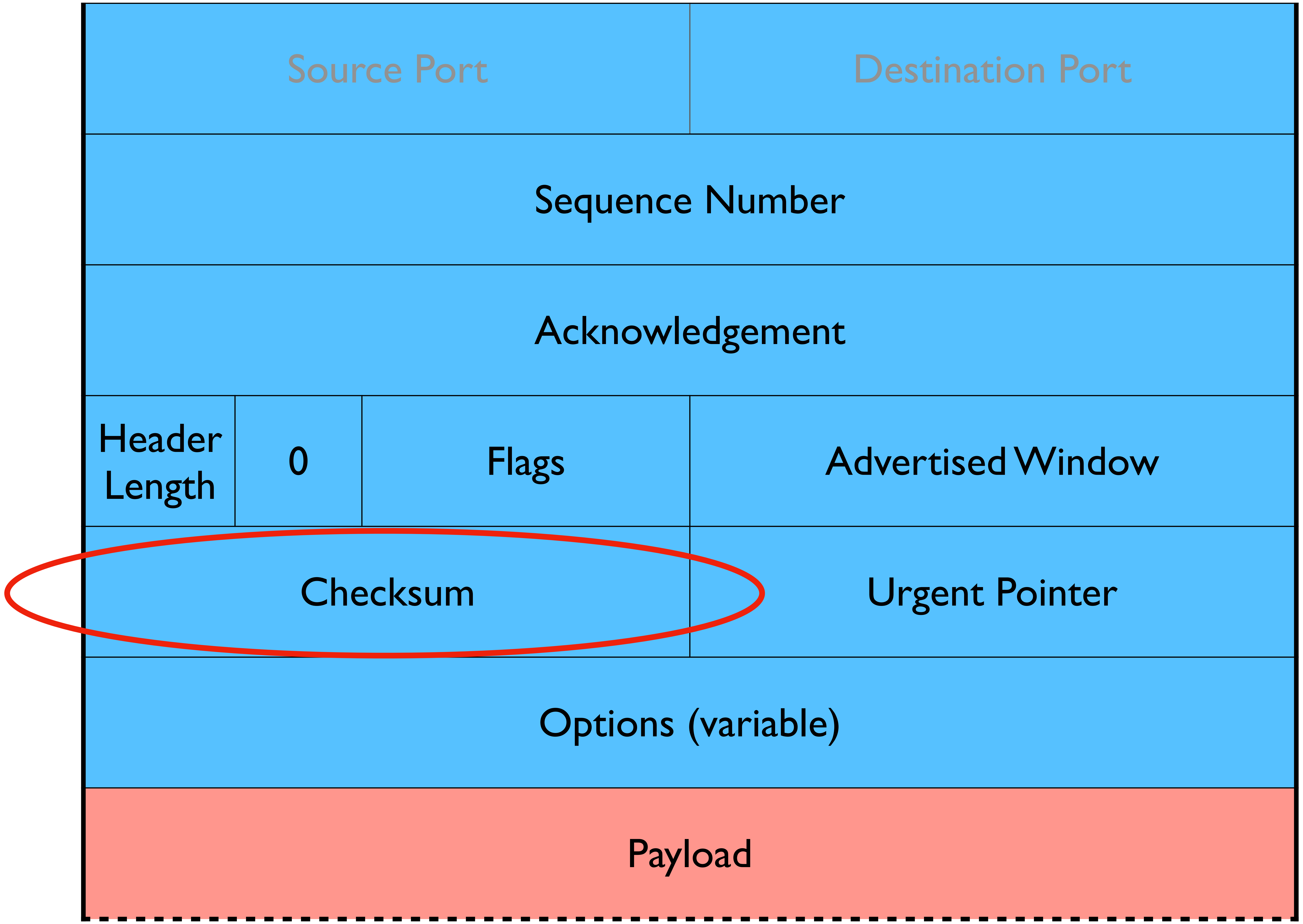
- **Most of our previous tricks + a few differences**

What does TCP do for reliability?

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

TCP Header

Computed over pseudo-header & data



What does TCP do for reliability?

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

What does TCP do for reliability?

- **Most of our previous tricks + a few differences**
  - Checksum
  - Sequence numbers are byte-offsets
    - And also includes the notion of a "segment" and ISNs

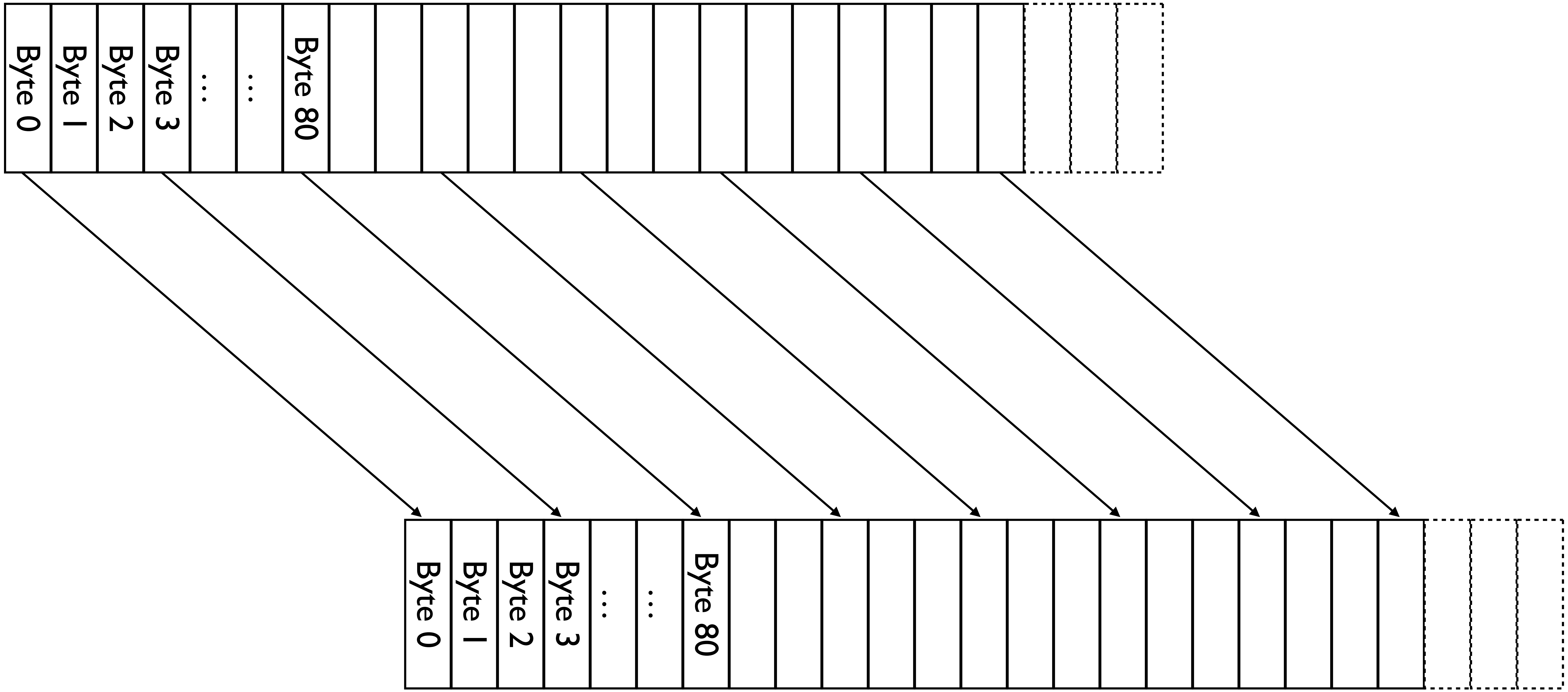
What does TCP do for reliability?

- **Most of our previous tricks + a few differences**
  - Checksum
  - Sequence numbers are byte-offsets
    - And also includes the notion of a "segment" and ISNs
    - Proof that networking is boring: 7 slides on sequence numbers!

TCP: Segments & Sequence Numbers

TCP “Stream-of-Bytes” Service…

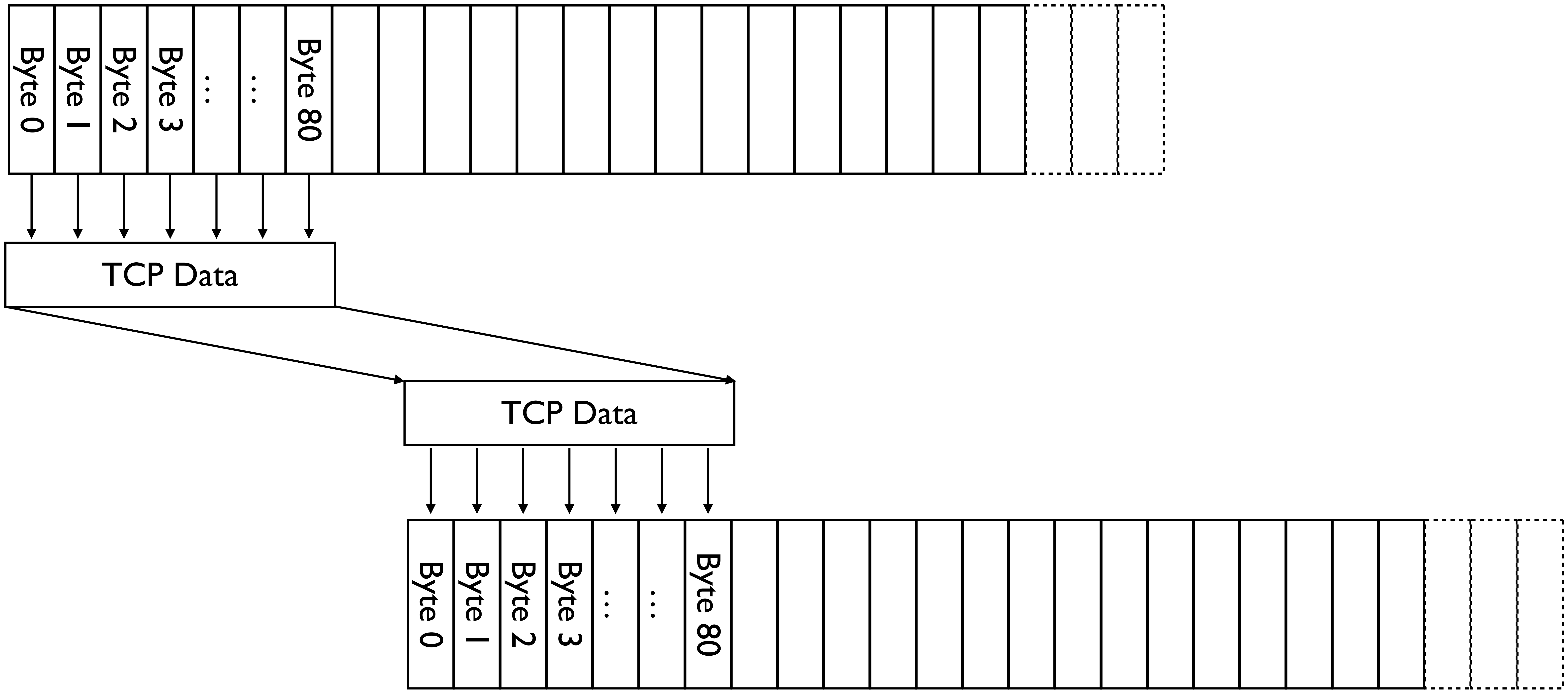
Application @ Host A



Application @ Host B

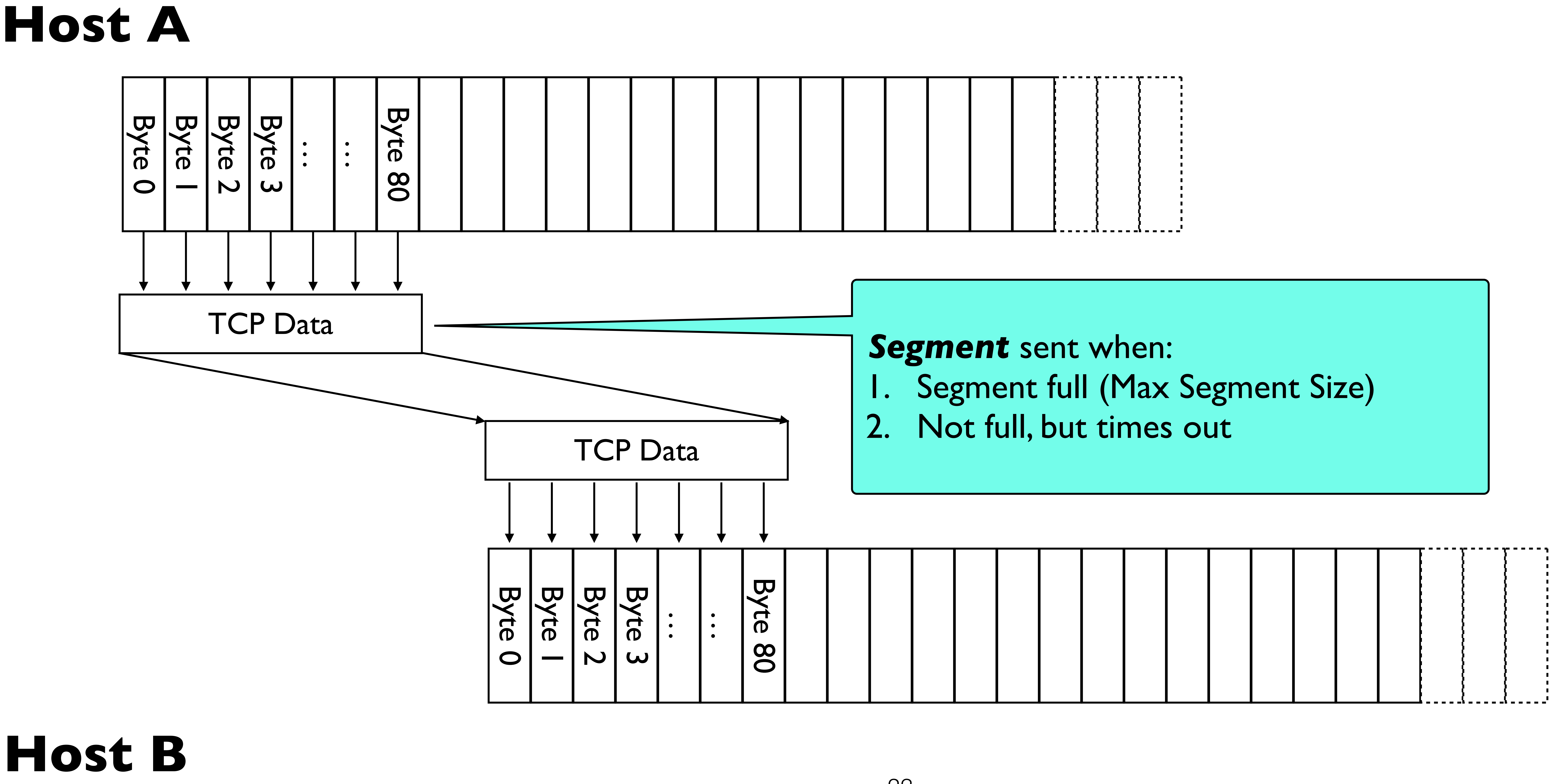
…Provided using TCP "Segments"

Host A



Host B

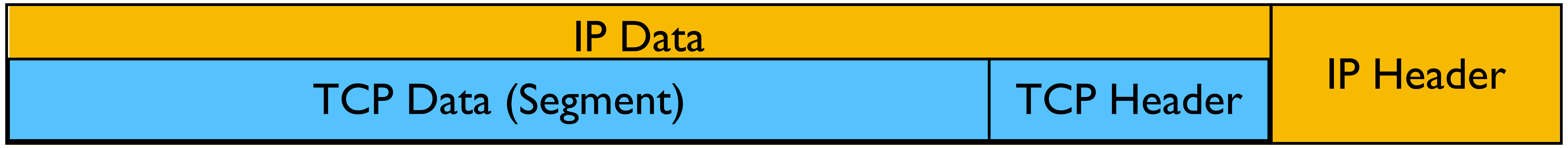
…Provided using TCP "Segments"



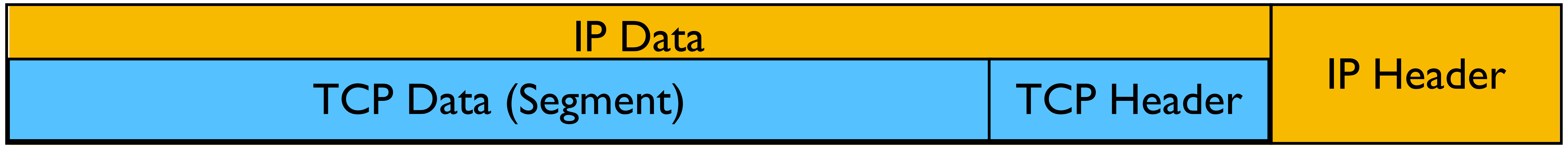
Segment sent when:

1. Segment full (Max Segment Size)
2. Not full, but times out

TCP Segment

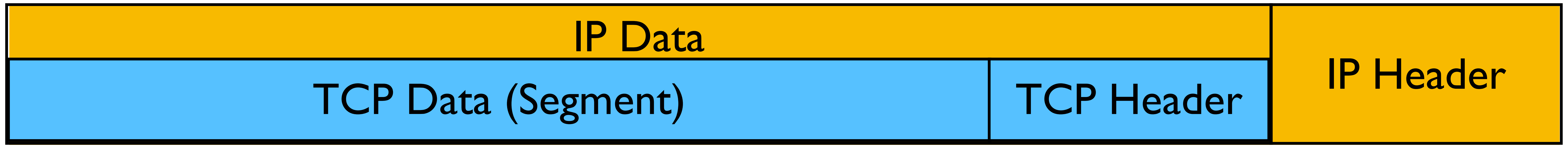


TCP Segment



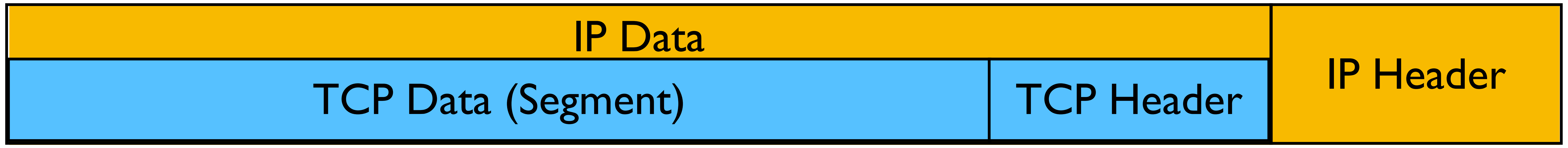
- **IP Packet**
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet

TCP Segment



- **IP Packet**
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- **TCP Packet**
  - IP packet with a TCP header and data inside
  - TCP header >= 20 bytes long

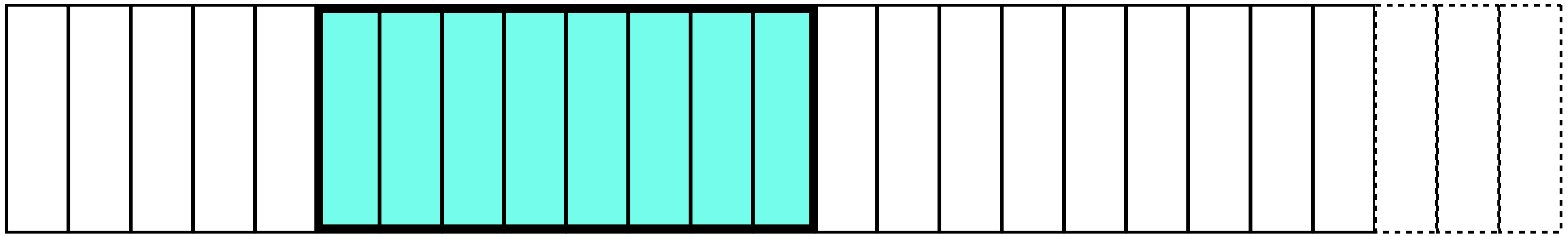
TCP Segment



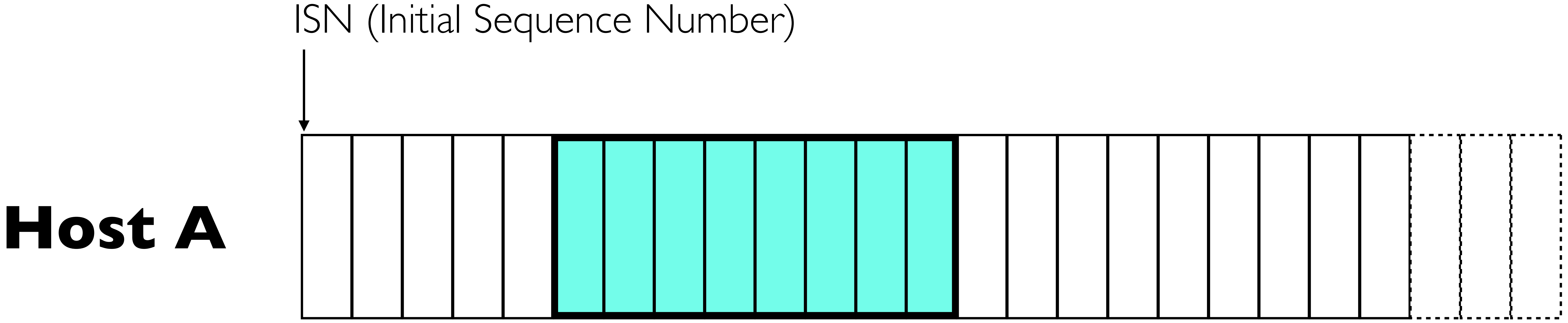
- **IP Packet**
  - No bigger than Maximum Transmission Unit (MTU)
  - E.g., up to 1500 bytes with Ethernet
- **TCP Packet**
  - IP packet with a TCP header and data inside
  - TCP header >= 20 bytes long
- **TCP Segment**
  - No more than **Maximum Segment Size** (MSS) bytes
  - E.g., up to 1460 consecutive bytes from the stream
  - MSS = MTU - (IP Header Size) - (TCP Header Size)

Sequence Numbers

Host A

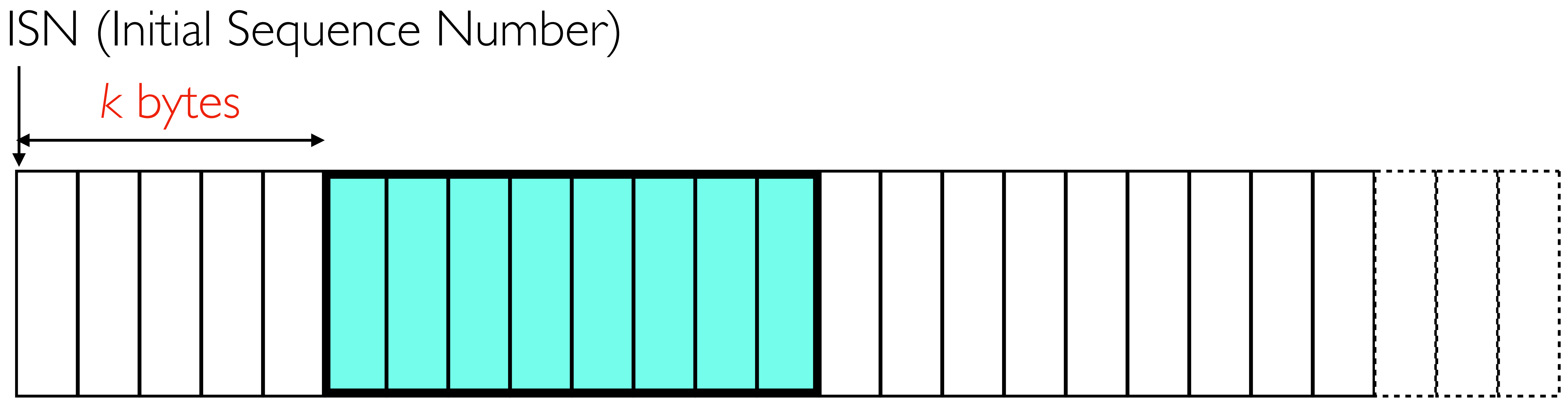


Sequence Numbers

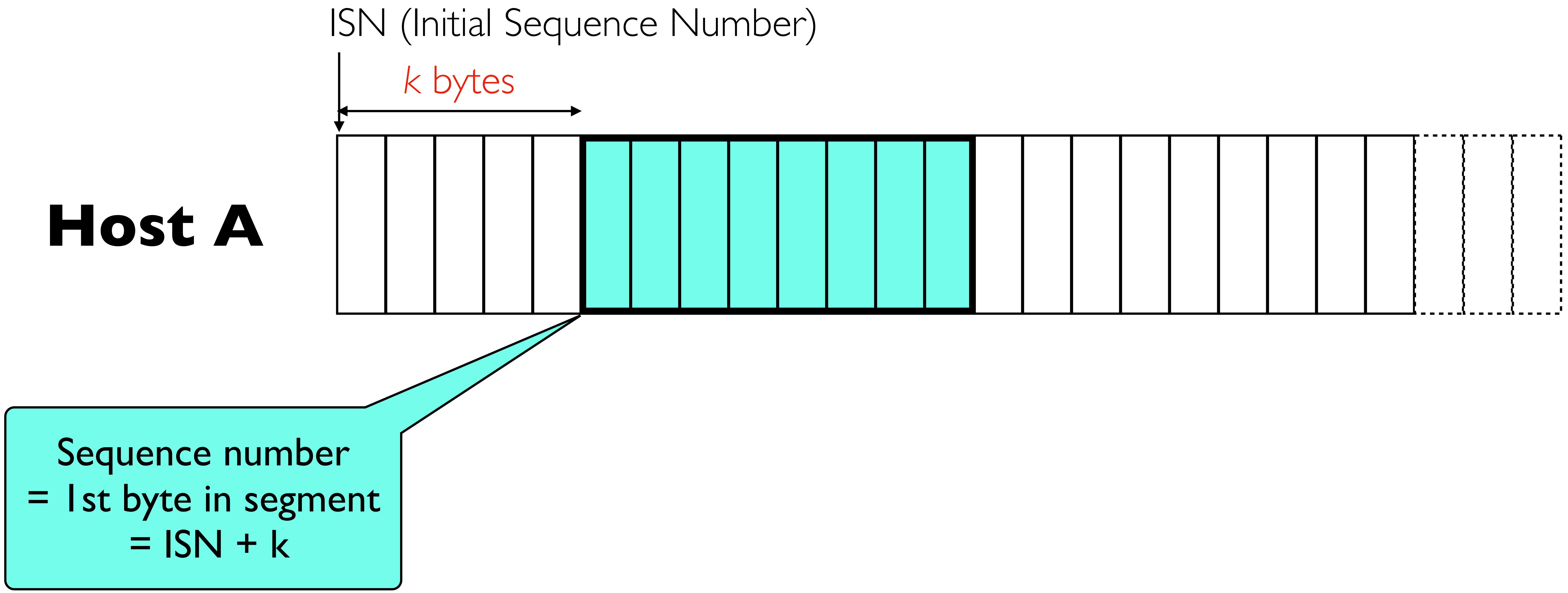


Sequence Numbers

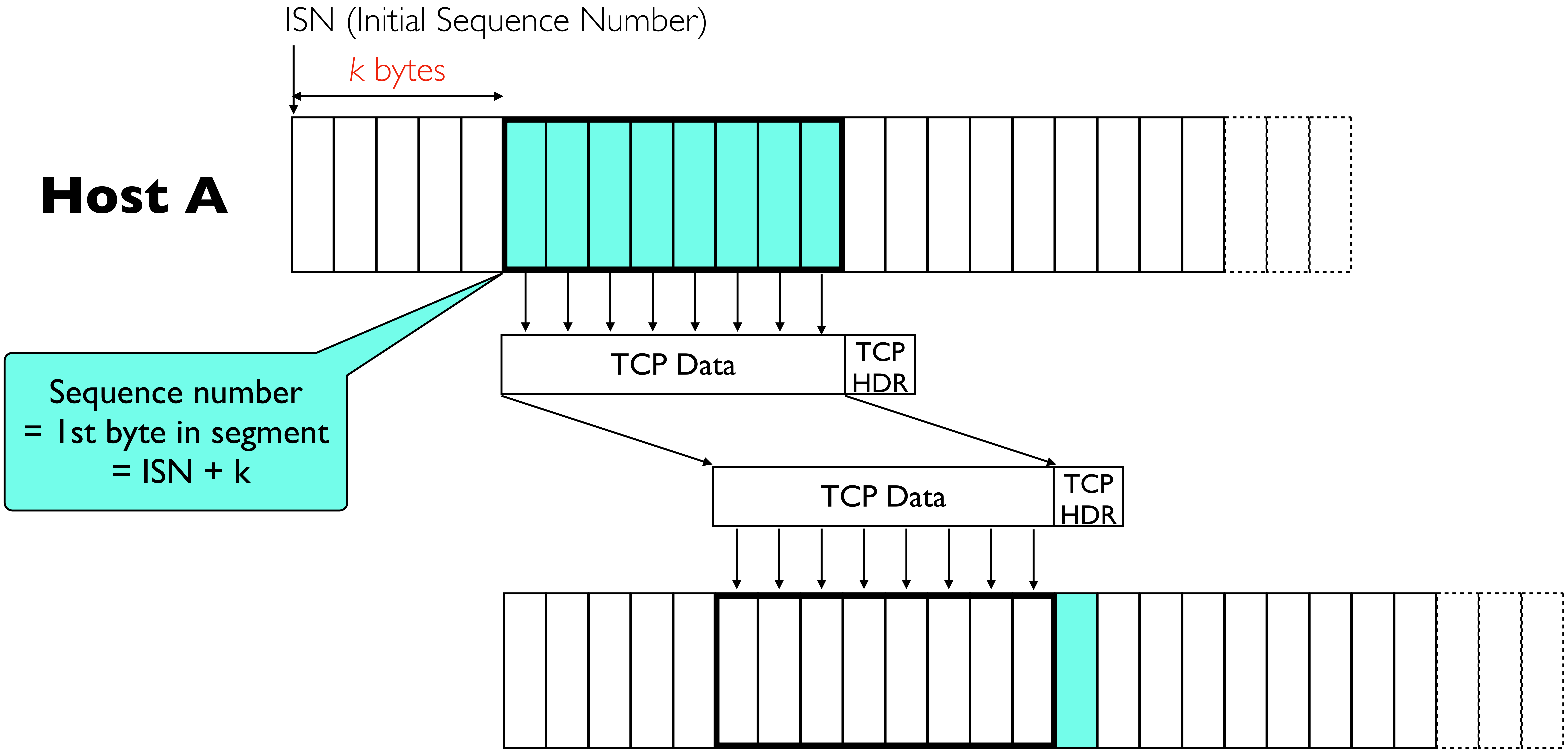
Host A



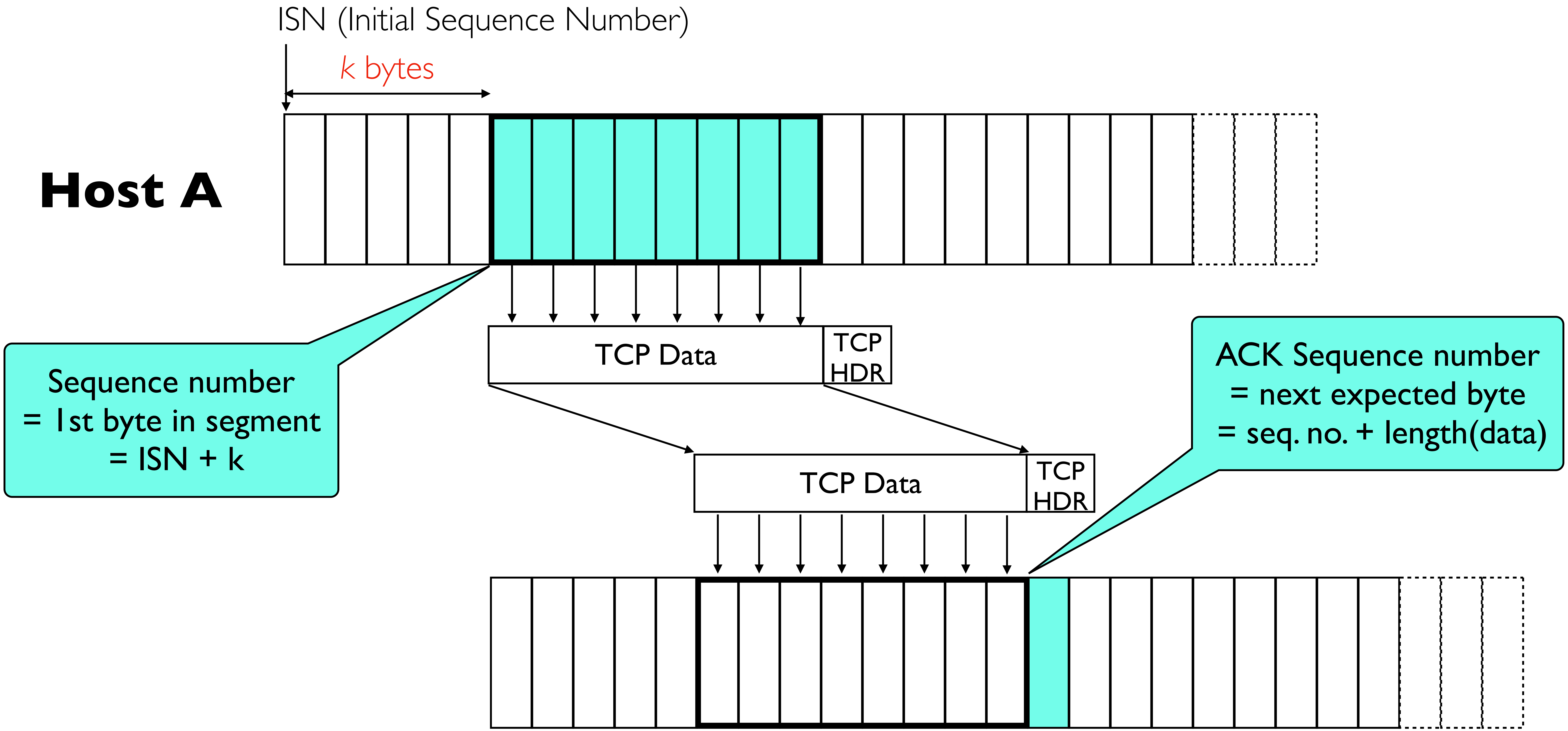
Sequence Numbers



TCP "Stream-of-Bytes" Service…

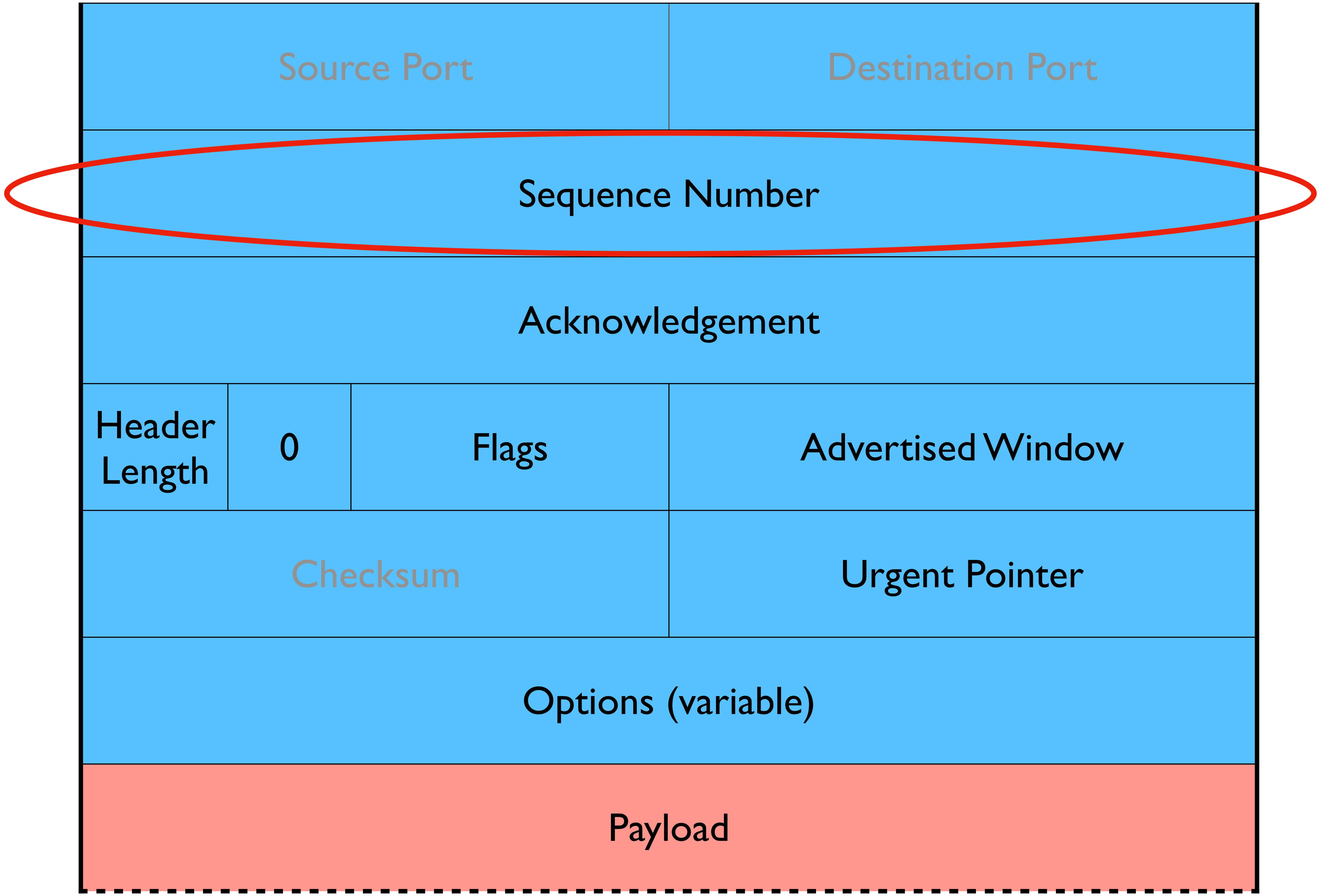


TCP "Stream-of-Bytes" Service…



TCP Header

Starting byte offset of
data carried in this segment



What does TCP do for reliability?

- **Most of our previous tricks + a few differences**
  - Checksum
  - Sequence numbers are byte-offsets
  - Receiver sends cumulative acknowledgements (like GBN)

ACKing and Sequence Numbers

ACKing and Sequence Numbers

- **Sender sends packet**

ACKing and Sequence Numbers

- **Sender sends packet**
    - Data starts with sequence number X

ACKing and Sequence Numbers

- **Sender sends packet**
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, …, X+B-1]

ACKing and Sequence Numbers

- **Sender sends packet**
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, …, X+B-1]
- **Upon receipt of packet, receiver sends an ACK**

ACKing and Sequence Numbers

- **Sender sends packet**
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, …, X+B-1]
- **Upon receipt of packet, receiver sends an ACK**
  - If all data prior to X already received:

ACKing and Sequence Numbers

- **Sender sends packet**
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, …, X+B-1]
- **Upon receipt of packet, receiver sends an ACK**
  - If all data prior to X already received:
    - ACK acknowledges **X+B** (because that is the next expected byte)

ACKing and Sequence Numbers

- **Sender sends packet**
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, …, X+B-1]
- **Upon receipt of packet, receiver sends an ACK**
  - If all data prior to X already received:
    - ACK acknowledges **X+B** (because that is the next expected byte)
  - If the highest in-order byte is Y, such that, (Y+1) < X

ACKing and Sequence Numbers

- **Sender sends packet**
  - Data starts with sequence number X
  - Packet contains B bytes [X, X+1, X+2, …, X+B-1]
- **Upon receipt of packet, receiver sends an ACK**
  - If all data prior to X already received:
    - ACK acknowledges **X+B** (because that is the next expected byte)
  - If the highest in-order byte is Y, such that, (Y+1) < X
    - ACK acknowledges **Y+1,** even if it has been ACKed before

Normal Pattern

Normal Pattern

- Sender: seqno=X, length=B

Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B

Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B

Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B

Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B

Normal Pattern

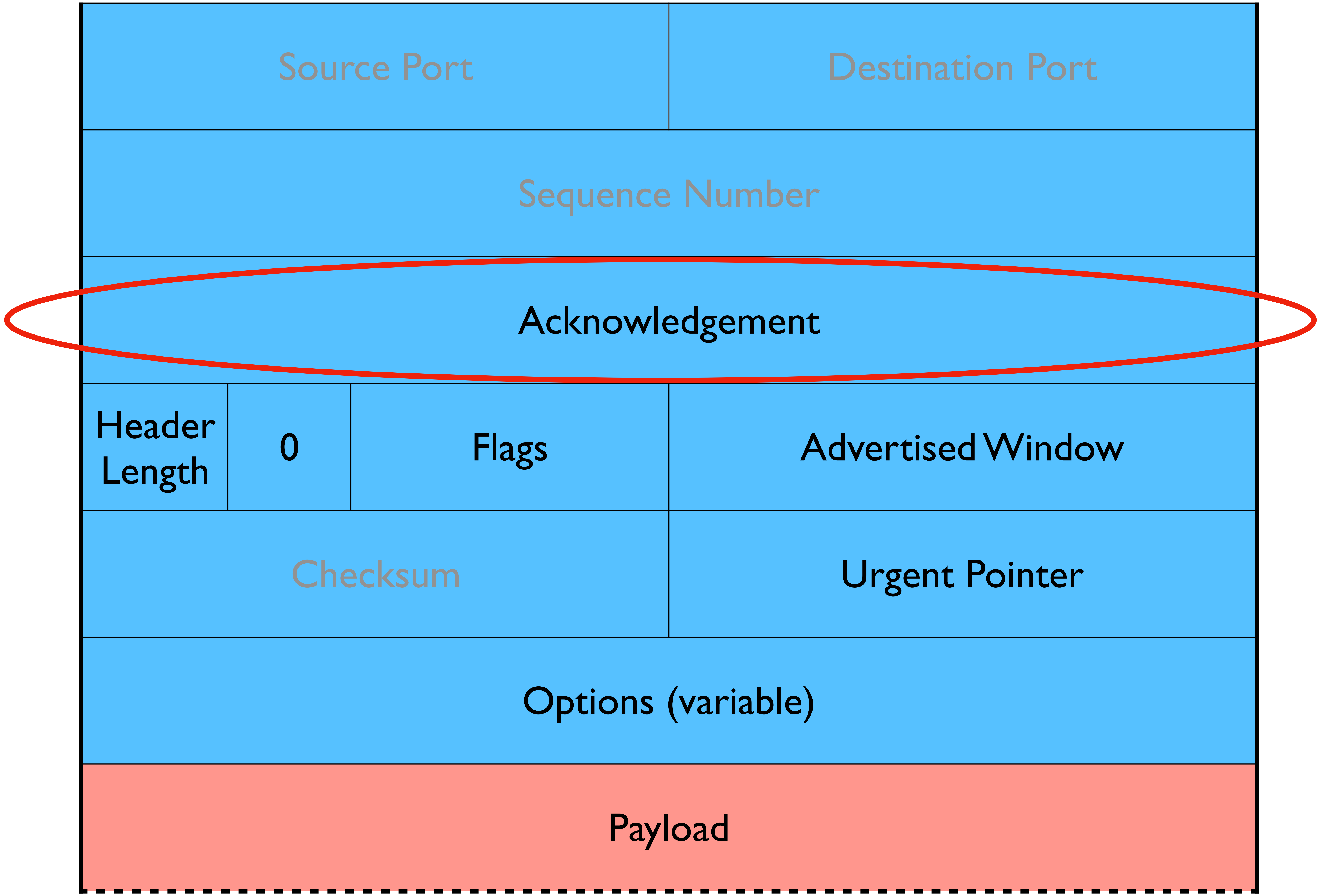
- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B
- ...

Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B
- …
- **seqno** of next packet is same as last **ACK** field

TCP Header

Acknowledgement gives seqno just beyond highest seqno received *in-order*
("What Byte Is Next")



What does TCP do for reliability?

- **Most of our previous tricks + a few differences**
  - Checksum
  - Sequence numbers are byte-offsets
  - Receiver sends cumulative acknowledgements (like GBN)
  - Receivers **can buffer** out of sequence packets (like SR)

Loss with Cumulative ACKs

Loss with Cumulative ACKs

- **Sender sends packets with 100B and seqnos:**
    - 100, 200, 300, 400, 500, 600, 700, 800, 900, …

Loss with Cumulative ACKs

- **Sender sends packets with 100B and seqnos:**
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, …
- **Assume the fifth packet (seqno 500) is lost, but no others**

Loss with Cumulative ACKs

- **Sender sends packets with 100B and seqnos:**
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, …
- **Assume the fifth packet (seqno 500) is lost, but no others**
- **Stream of ACKs will be:**
  - 200, 300, 400, 500, 500, 500, 500, …

What does TCP do for reliability?

- **Most of our previous tricks + a few differences**
  - Checksum
  - Sequence numbers are byte-offsets
  - Receiver sends cumulative acknowledgements (like GBN)
  - Receivers can buffer out of sequence packets (like SR)
  - Introduces **fast retransmit**: optimization that uses duplicate ACKs to trigger early retransmission

Loss with Cumulative ACKs

Loss with Cumulative ACKs

- **Let's look back at our last example**
  - Sender: 100, 200, 300, 400, 500, 600, 700, 800, 900, …
  - Packet with seqno 500 is lost
  - Receiver: 200, 300, 400, 500, 500, 500, 500, …

Loss with Cumulative ACKs

- **Let's look back at our last example**
  - Sender: 100, 200, 300, 400, 500, 600, 700, 800, 900, …
  - Packet with seqno 500 is lost
  - Receiver: 200, 300, 400, 500, 500, 500, 500, …
- **"Duplicate ACKs" are a sign of an <u>isolated</u> loss**
  - The lack of ACK progress means 500 has not been delivered
  - Stream of ACKs means packets are still being delivered

Loss with Cumulative ACKs

- **Let's look back at our last example**
  - Sender: 100, 200, 300, 400, 500, 600, 700, 800, 900, …
  - Packet with seqno 500 is lost
  - Receiver: 200, 300, 400, 500, 500, 500, 500, …
- **"Duplicate ACKs" are a sign of an <u>isolated</u> loss**
  - The lack of ACK progress means 500 has not been delivered
  - Stream of ACKs means packets are still being delivered
- **Optimization:** trigger retransmit on receiving on receiving k dupACKs
  - TCP uses k=3 [Fast Retransmit]

What does TCP do for reliability?

- **Most of our previous tricks + a few differences**
  - Checksum
  - Sequence numbers are byte-offsets
  - Receiver sends cumulative acknowledgements (like GBN)
  - Receivers can buffer out of sequence packets (like SR)
  - Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
  - Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

Retransmission Timeout

Retransmission Timeout

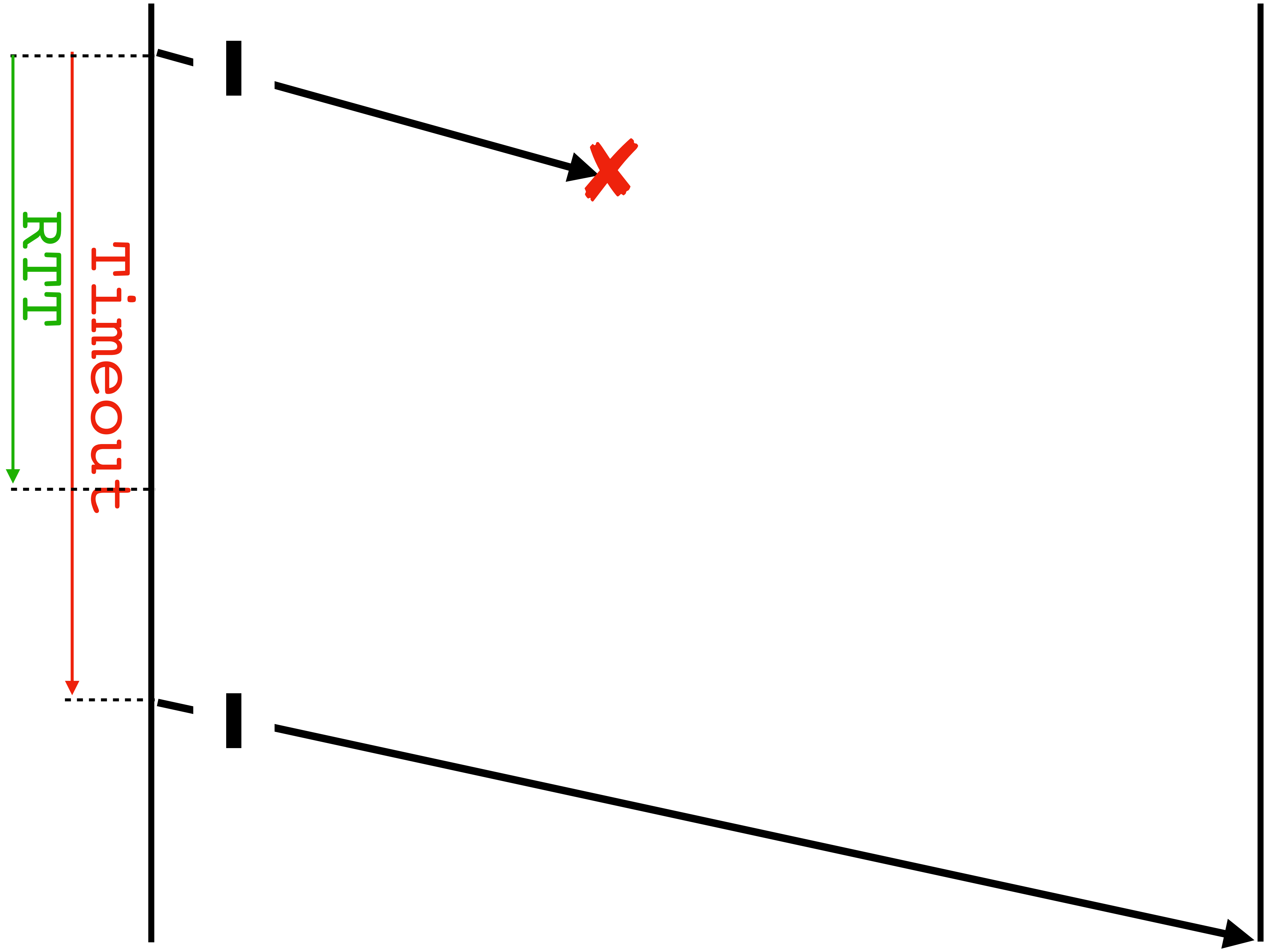
- **If the sender hasn't received an ACK by timeout, retransmit the <u>first</u> segment in the window**

Retransmission Timeout

- **If the sender hasn't received an ACK by timeout, retransmit the <u>first</u> segment in the window**
- **How do we pick a timeout value?**

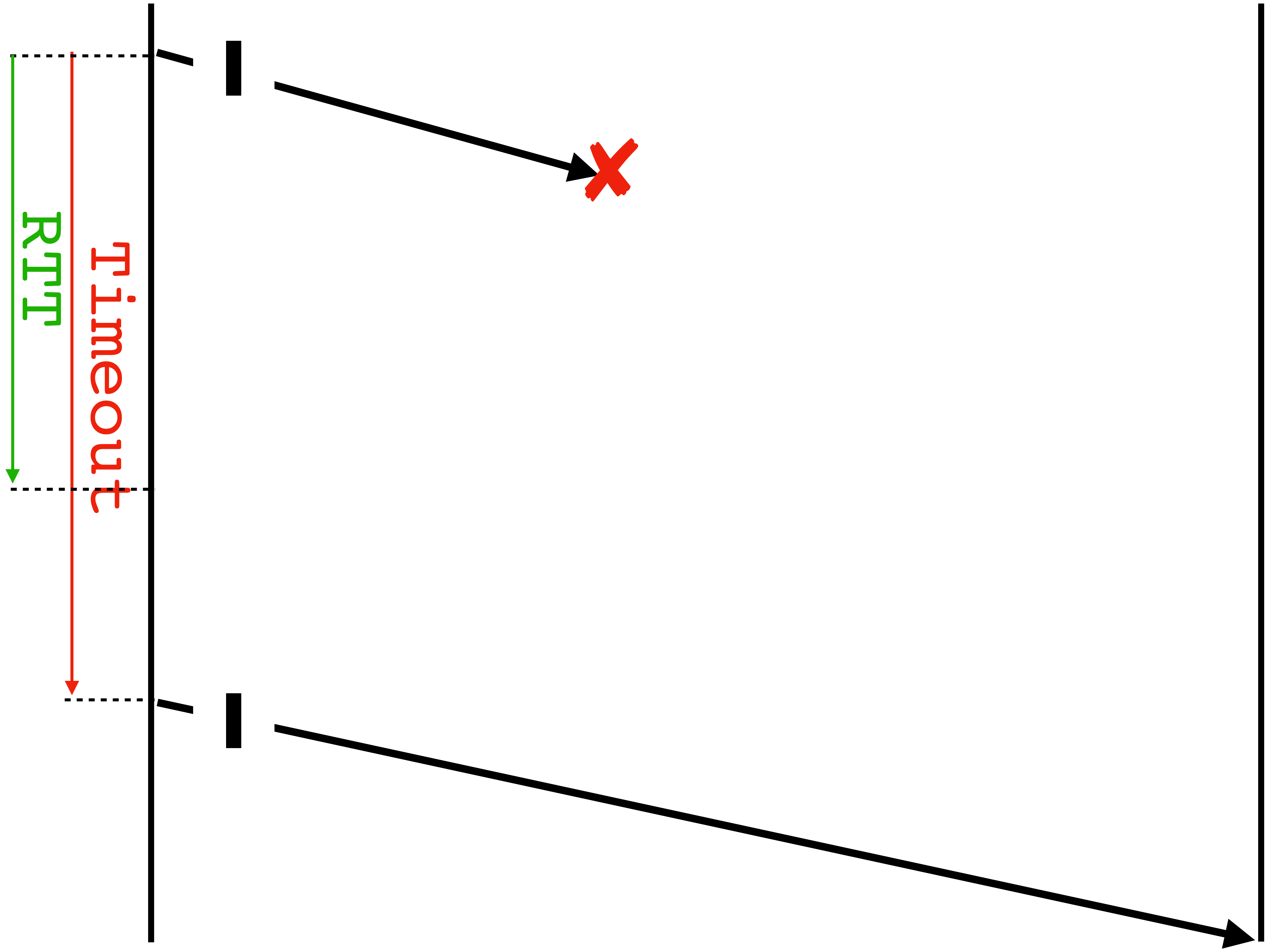
Timing Illustration

Timing Illustration

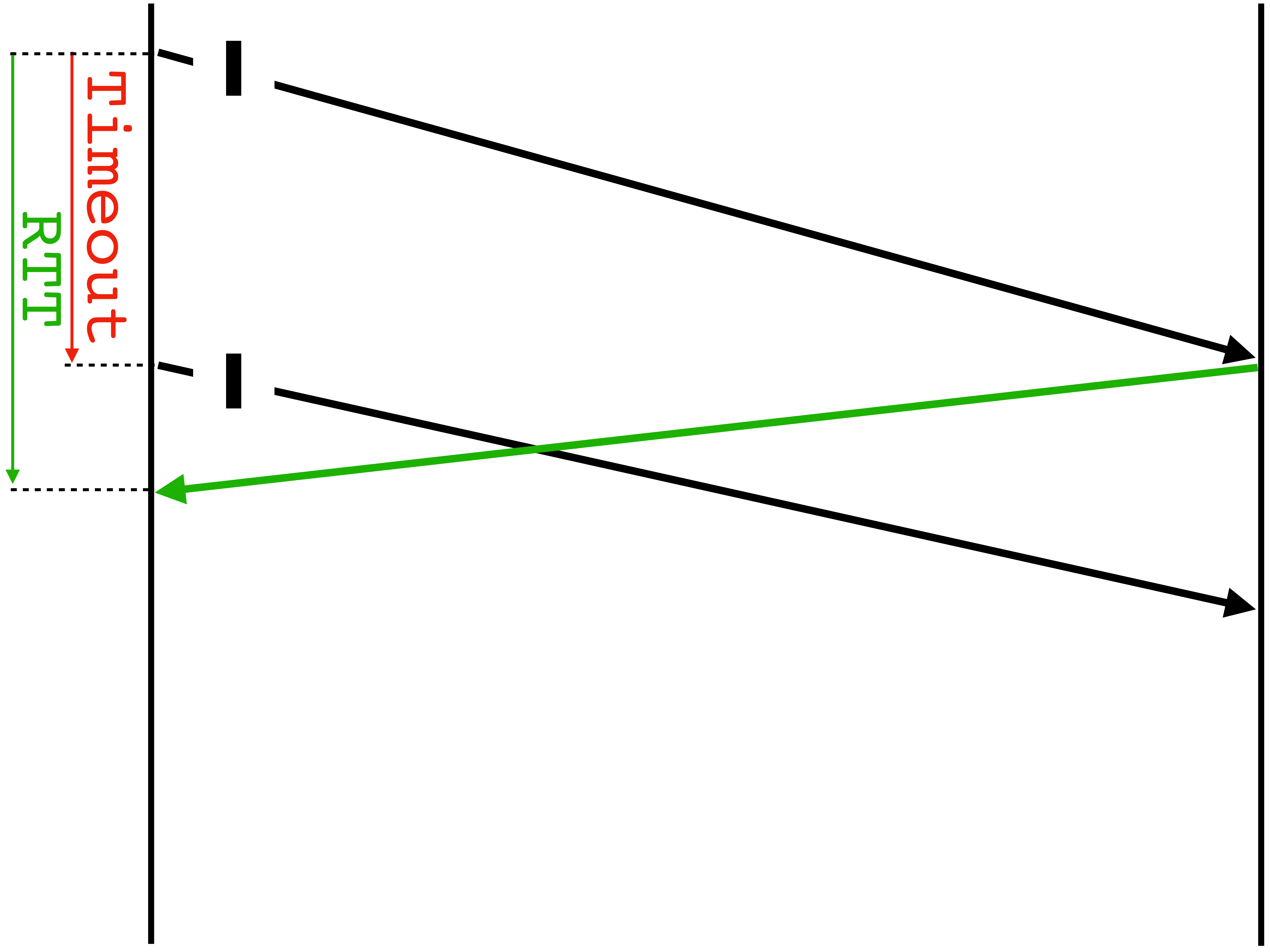


Timeout too long → Inefficient

Timing Illustration



Timeout too long → Inefficient



Timeout too short → Duplicate packets

Retransmission Timeout

- **If the sender hasn't received an ACK by timeout, retransmit the first segment in the window**
- **How do we pick a timeout value?**

Retransmission Timeout

- **If the sender hasn't received an ACK by timeout, retransmit the <u>first</u> segment in the window**
- **How do we pick a timeout value?**
  - *<u>Too long:</u>* connection has low throughput

Retransmission Timeout

- **If the sender hasn't received an ACK by timeout, retransmit the <u>first</u> segment in the window**
- **How do we pick a timeout value?**
  - *<u>Too long:</u>* connection has low throughput
  - *<u>Too short:</u>* retransmit packet that was just delayed

Retransmission Timeout

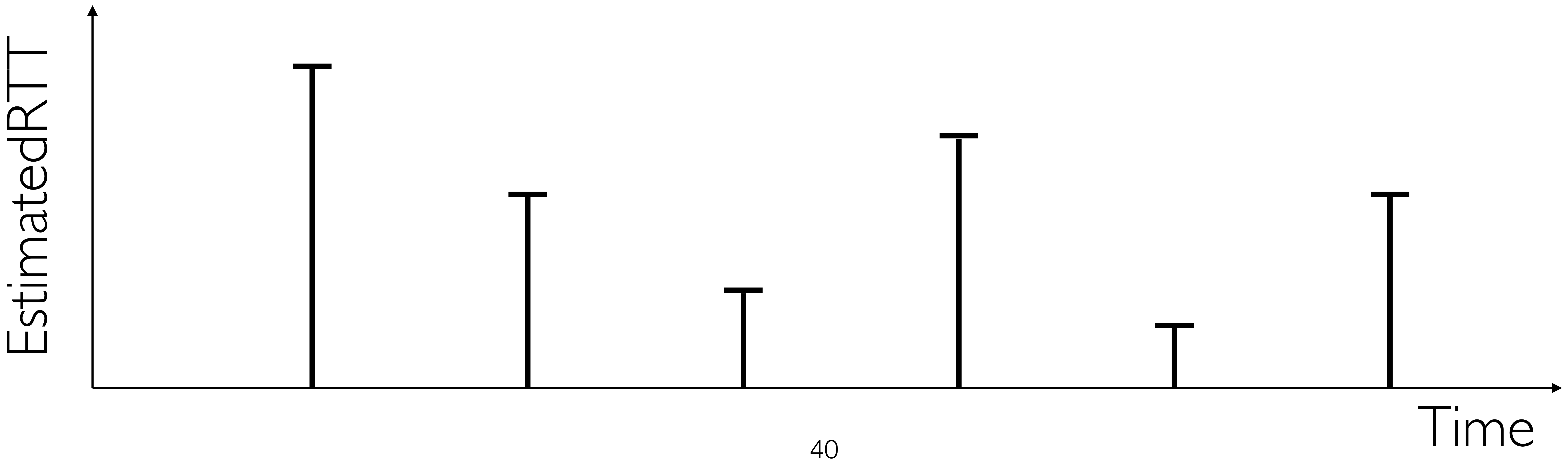
- **If the sender hasn't received an ACK by timeout, retransmit the first segment in the window**
- **How do we pick a timeout value?**
  - *Too long:* connection has low throughput
  - *Too short:* retransmit packet that was just delayed
- **Solution:** make timeout proportional to RTT

Retransmission Timeout

- **If the sender hasn't received an ACK by timeout, retransmit the <u>first</u> segment in the window**
- **How do we pick a timeout value?**
  - <u>*Too long:*</u> connection has low throughput
  - <u>*Too short:*</u> retransmit packet that was just delayed
- **Solution:** make timeout proportional to RTT
- **But how do we measure RTT?**

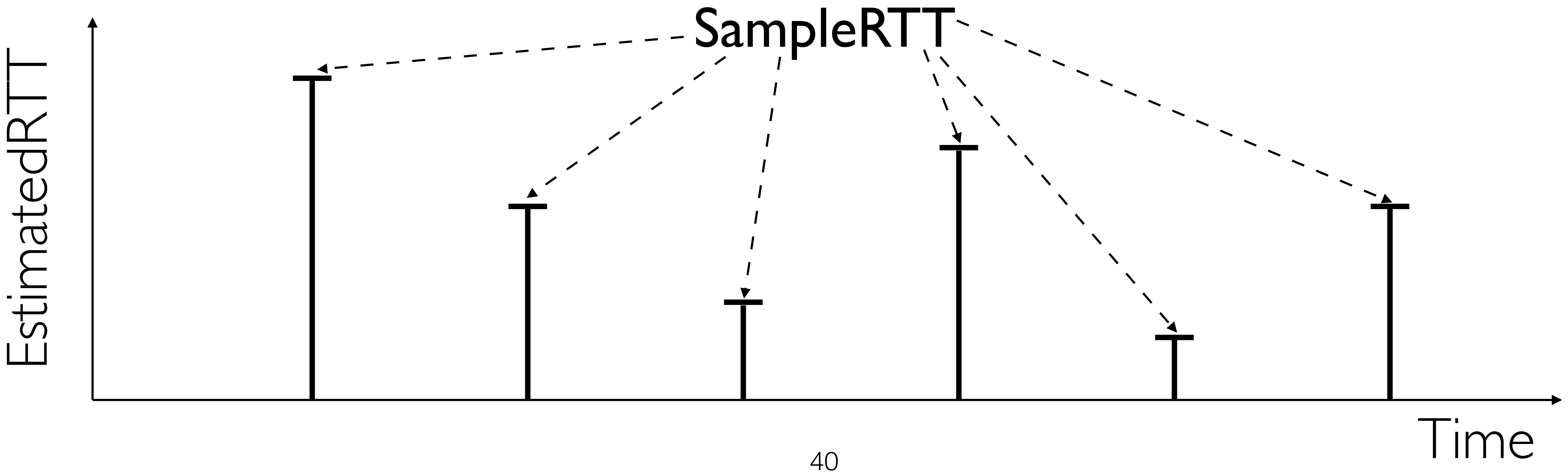
RTT Estimation

- **Use exponential averaging of RTT Samples**



RTT Estimation

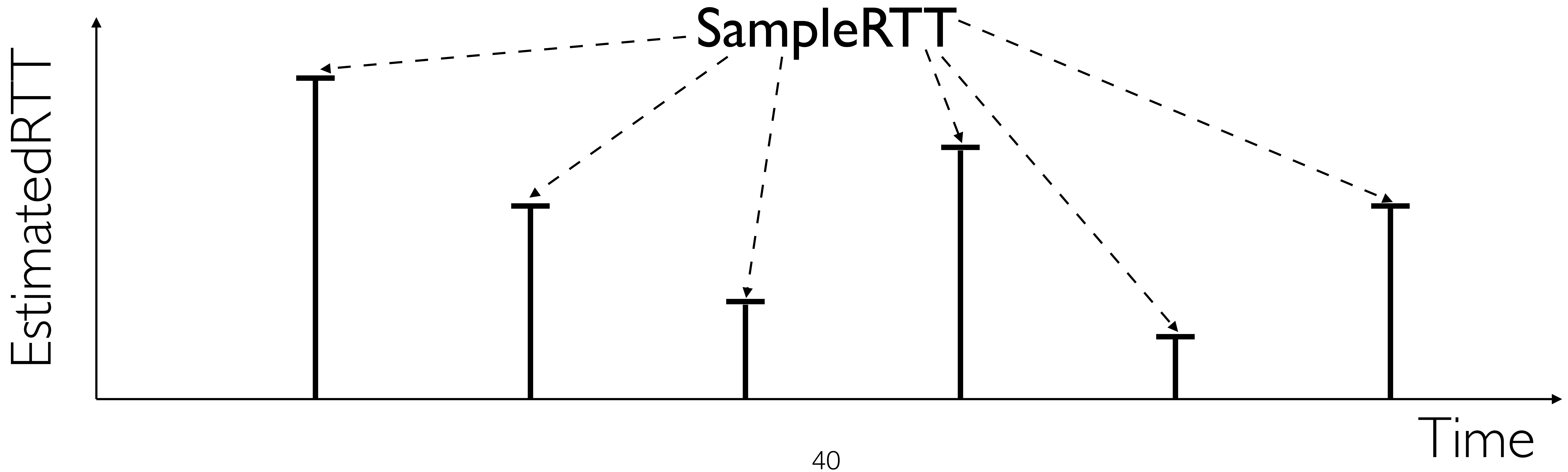
- **Use exponential averaging of RTT Samples**



RTT Estimation

- **Use exponential averaging of RTT Samples**

SampleRTT = AckRcvdTime - SendPacketTime
EstimatedRTT = α x EstimatedRTT + (1- α) x SampleRTT
0 < α <= 1



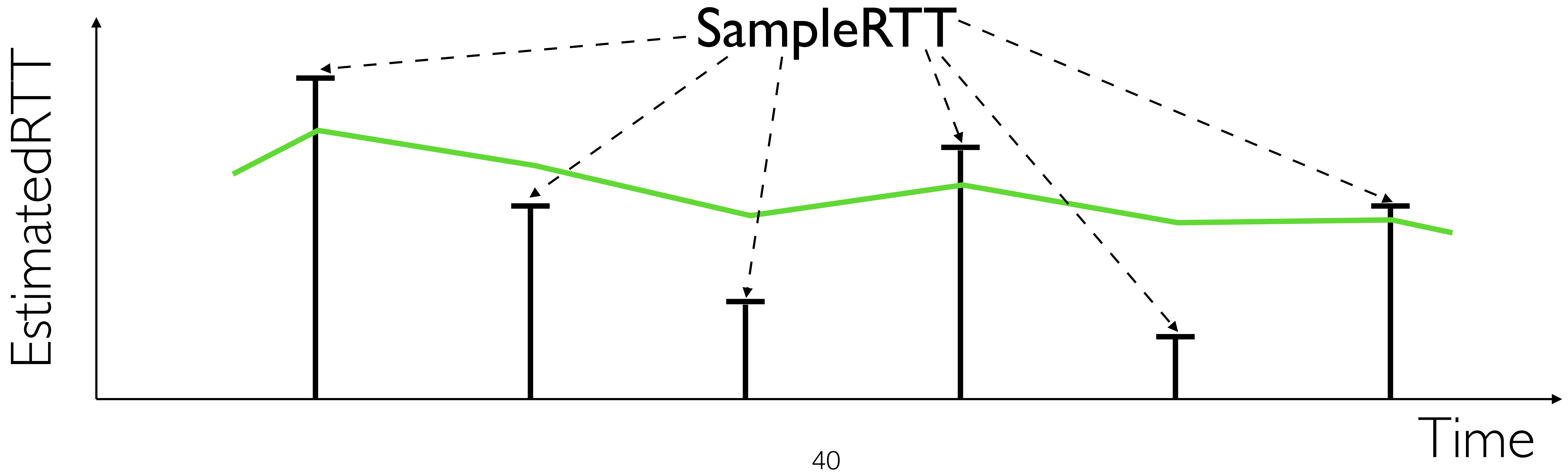
RTT Estimation

- **Use exponential averaging of RTT Samples**

SampleRTT = AckRcvdTime - SendPacketTime

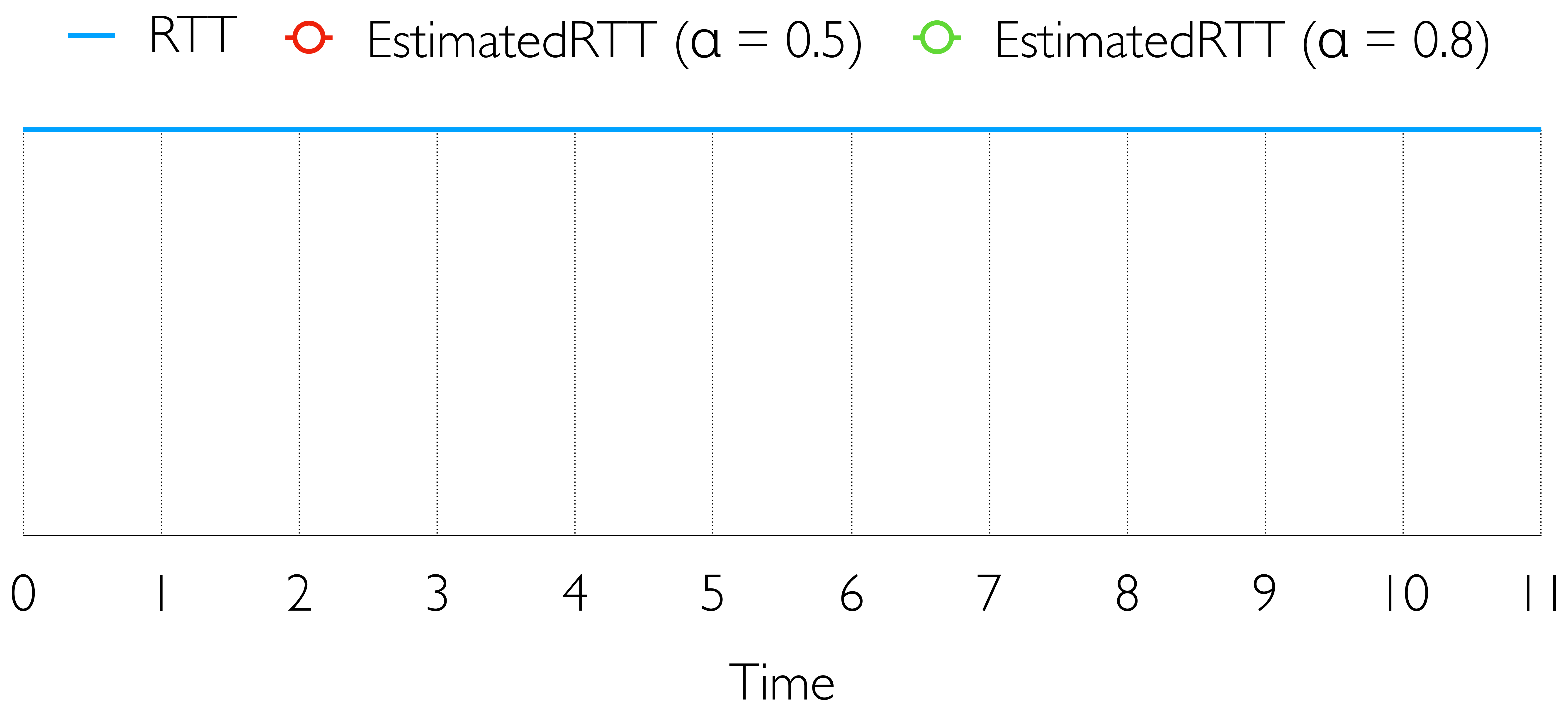
EstimatedRTT = α x EstimatedRTT + (1- α) x SampleRTT

0 < α <= 1



Exponential Averaging Example

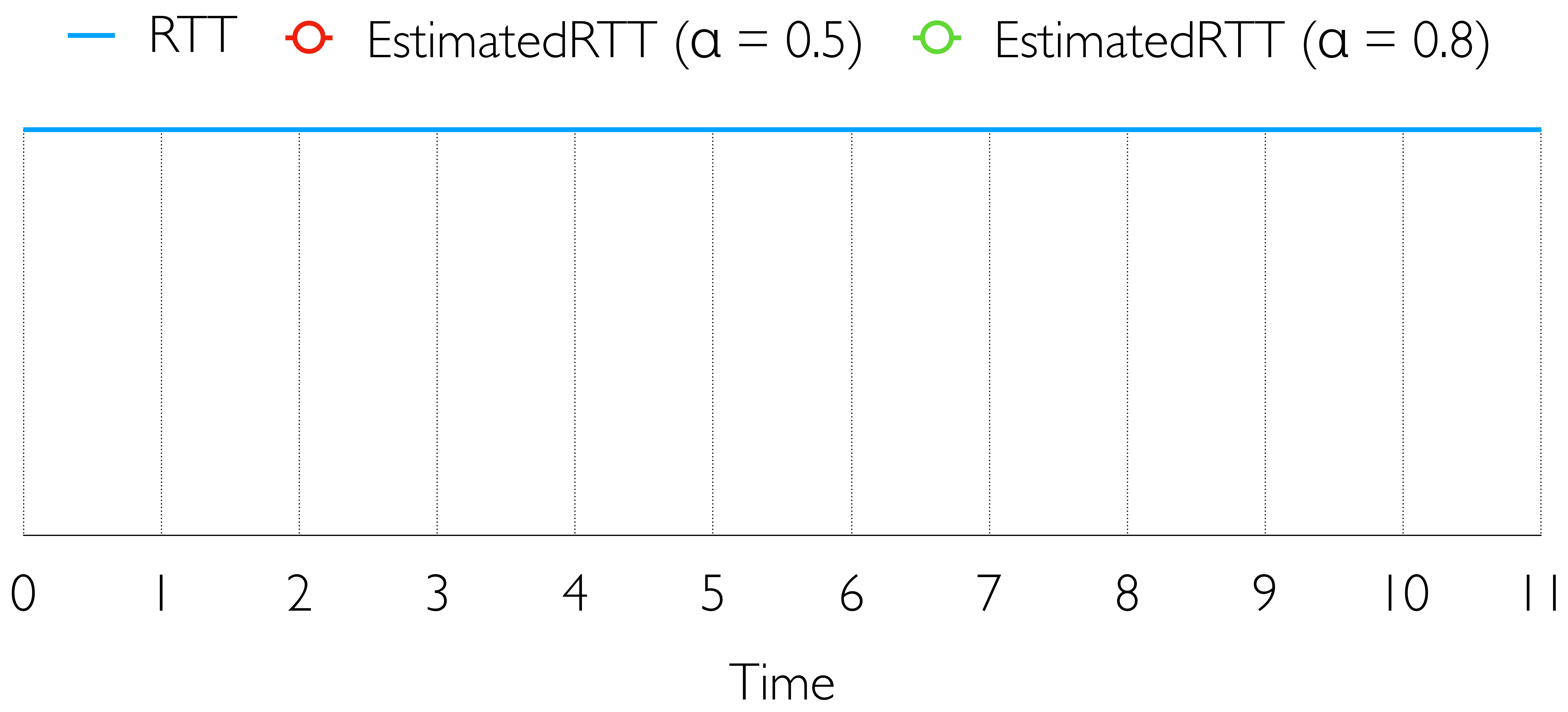
EstimatedRTT = α x EstimatedRTT + (1- α) x SampleRTT



Exponential Averaging Example

EstimatedRTT = α x EstimatedRTT + (1- α) x SampleRTT

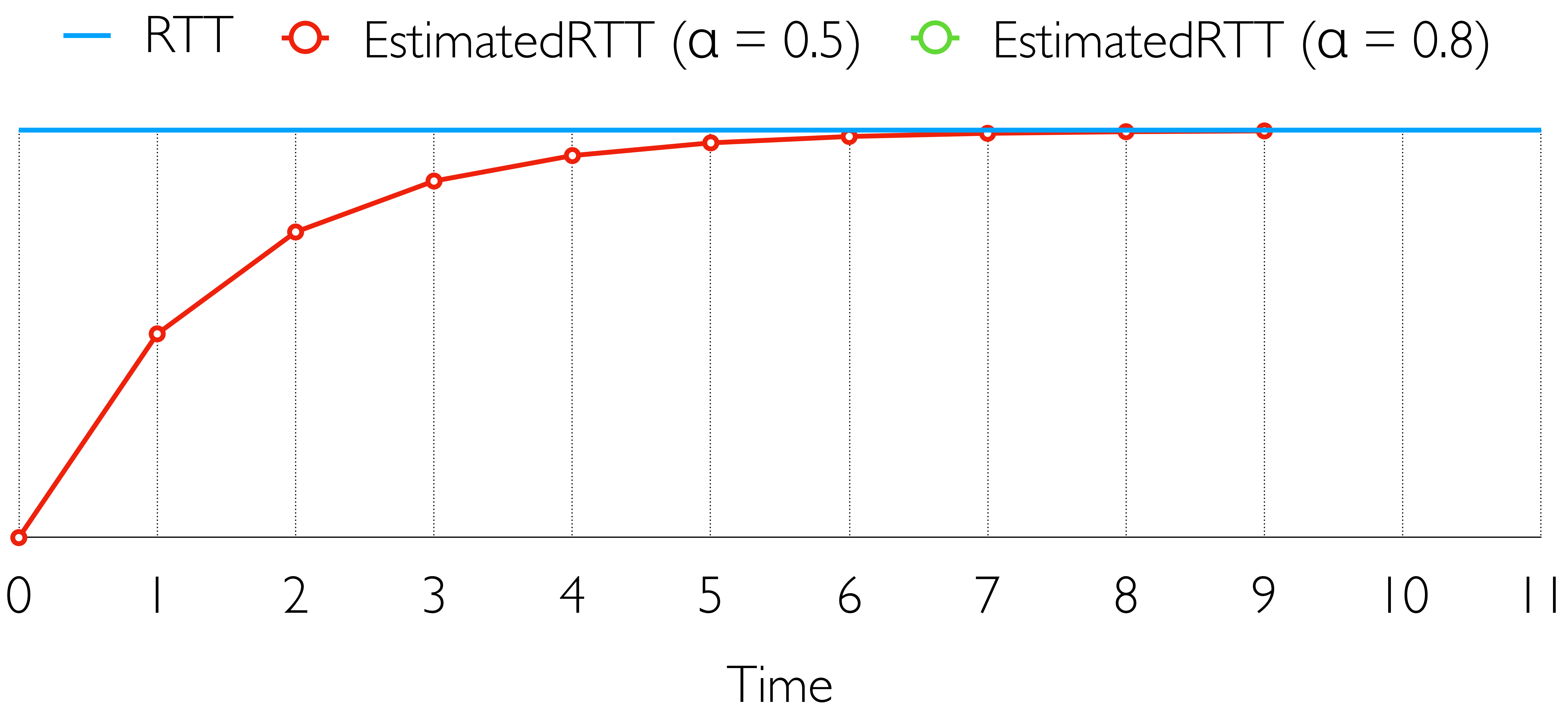
Assume RTT is constant → SampleRTT = RTT



Exponential Averaging Example

EstimatedRTT = α x EstimatedRTT + (1- α) x SampleRTT

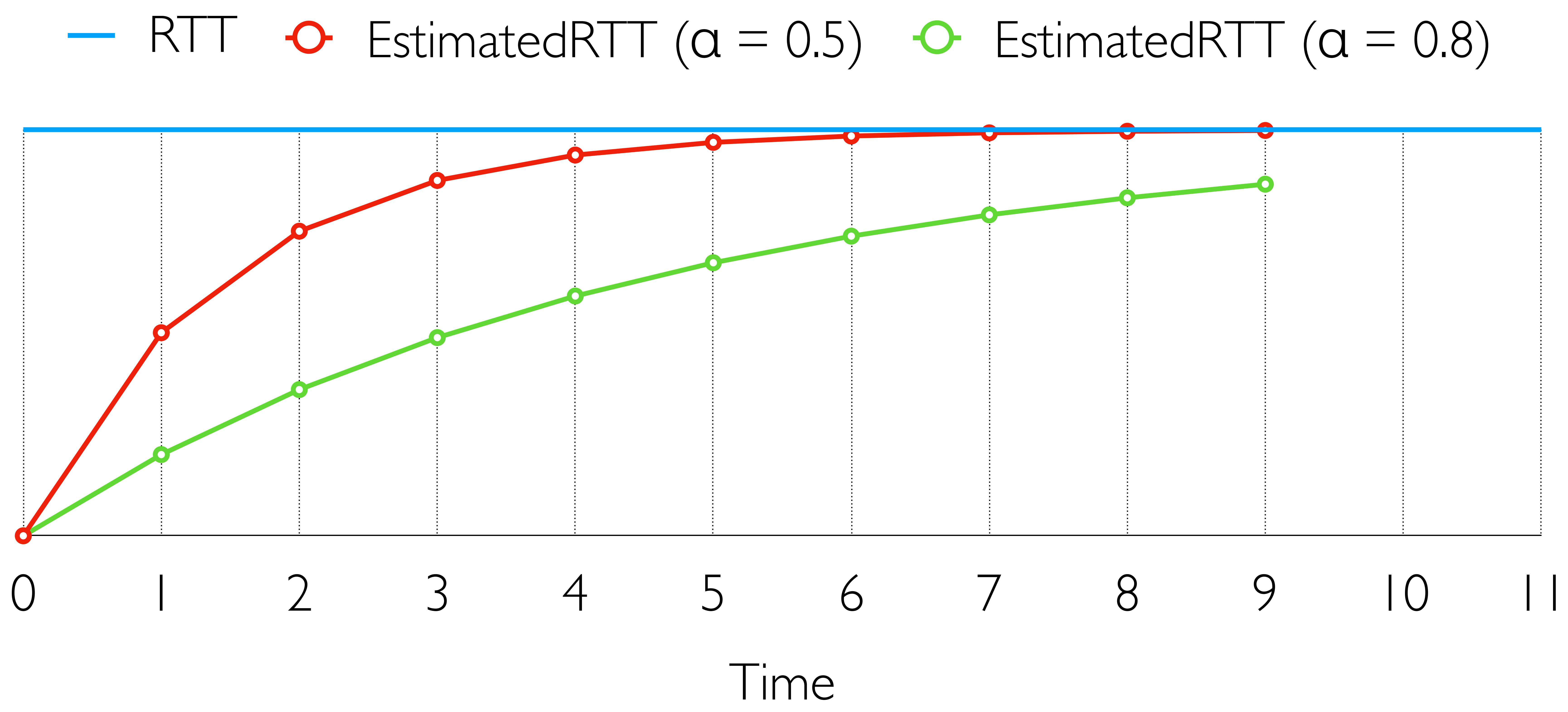
Assume RTT is constant → SampleRTT = RTT



Exponential Averaging Example

EstimatedRTT = α x EstimatedRTT + (1- α) x SampleRTT

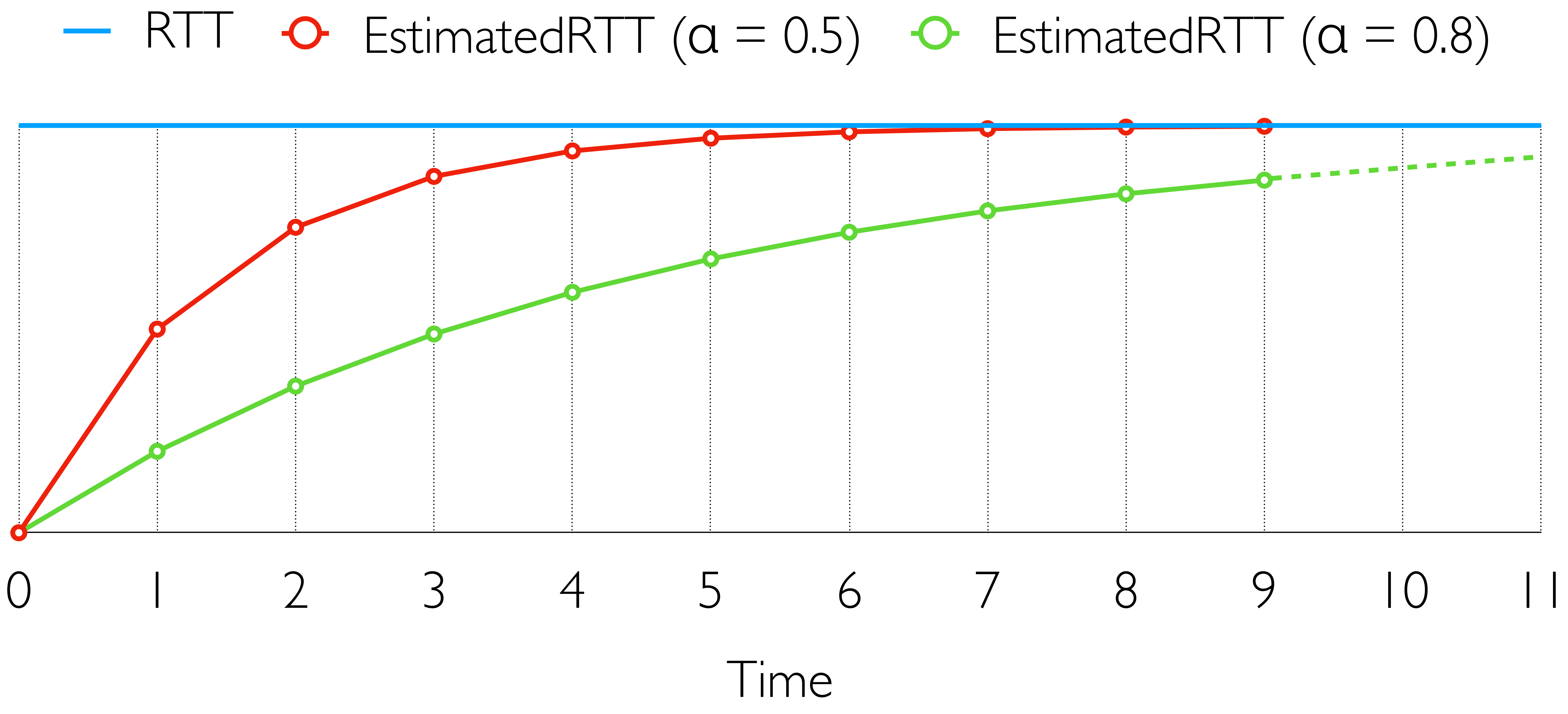
Assume RTT is constant → SampleRTT = RTT



Exponential Averaging Example

EstimatedRTT = α x EstimatedRTT + (1- α) x SampleRTT

Assume RTT is constant → SampleRTT = RTT



Karn/Partridge Algorithm

Karn/Partridge Algorithm

- **Compute EstimatedRTT using α = 0.875**

Karn/Partridge Algorithm

- **Compute EstimatedRTT using α = 0.875**
- **Measure SampleRTT only for original transmissions**
  - Once a segment has been retransmitted, do not use it for any further measurements

Karn/Partridge Algorithm

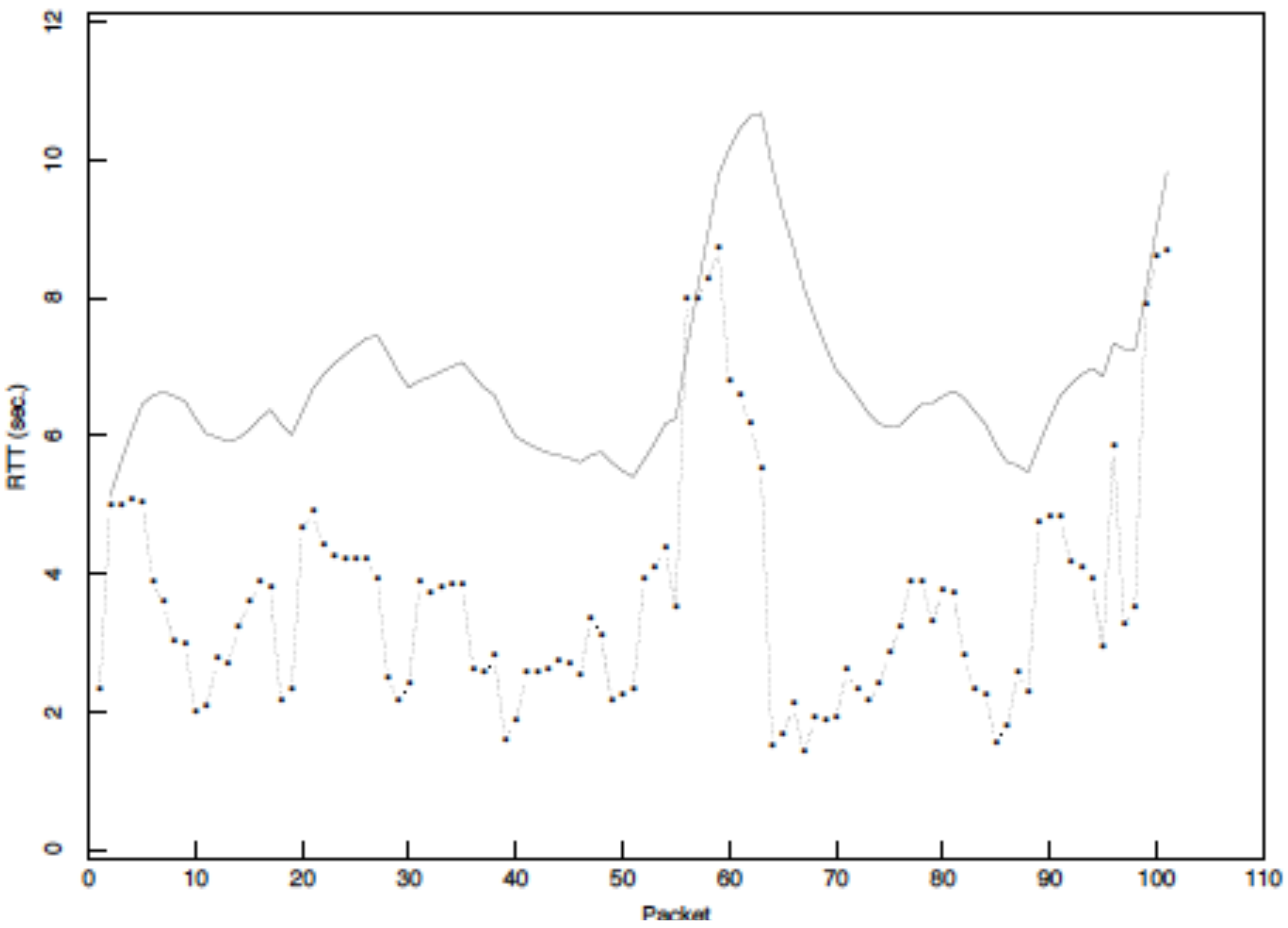
- **Compute EstimatedRTT using α = 0.875**
- **Measure SampleRTT only for original transmissions**
  - Once a segment has been retransmitted, do not use it for any further measurements
- **Timeout value (RTO) = 2 x EstimatedRTT**

Karn/Partridge Algorithm

- **Compute EstimatedRTT using α = 0.875**
- **Measure SampleRTT only for original transmissions**
  - Once a segment has been retransmitted, do not use it for any further measurements
- **Timeout value (RTO) = 2 x EstimatedRTT**
- **Employs *exponential backoff***
  - Every time RTO timer expires, set RTO ← 2 x RTO
  - (Up to a maximum >= 60seconds)
  - Every time new measurement comes in (=successful original transmission)
    - Collapse RTO back to 2 x EstimatedRTT

Karn/Partridge in Action

Figure 5: Performance of an RFC793 retransmit timer



From Jacobson & Karels, SIGCOMM 1988

Jacobson/Karels Algorithm

Jacobson/Karels Algorithm

- **Problem: need to better capture variability in RTT**
  - Directly measure *deviation*

Jacobson/Karels Algorithm

- **Problem: need to better capture variability in RTT**
  - Directly measure *deviation*
- **Deviation = | SampleRTT - EstimatedRTT |**

Jacobson/Karels Algorithm

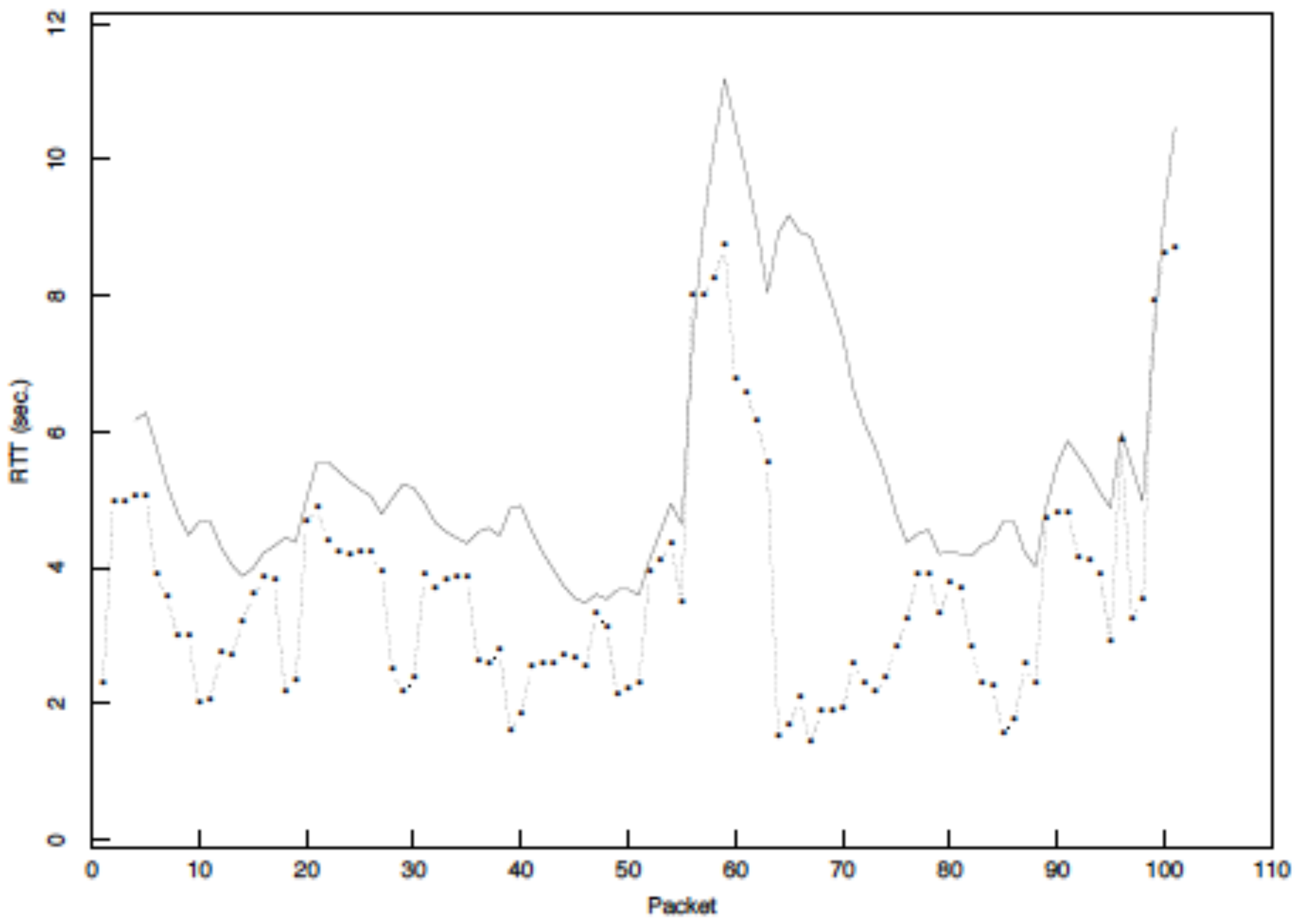
- **Problem: need to better capture variability in RTT**
  - Directly measure *deviation*
- **Deviation = | SampleRTT - EstimatedRTT |**
- **EstimatedDeviation = Exponential Average of Deviation**

Jacobson/Karels Algorithm

- **Problem: need to better capture variability in RTT**
  - Directly measure *deviation*
- **Deviation = | SampleRTT - EstimatedRTT |**
- **EstimatedDeviation = Exponential Average of Deviation**
- **RTO = EstimatedRTT + 4 x EstimatedDeviation**

With Jacobson/Karels

Figure 6: Performance of a Mean+Variance retransmit timer



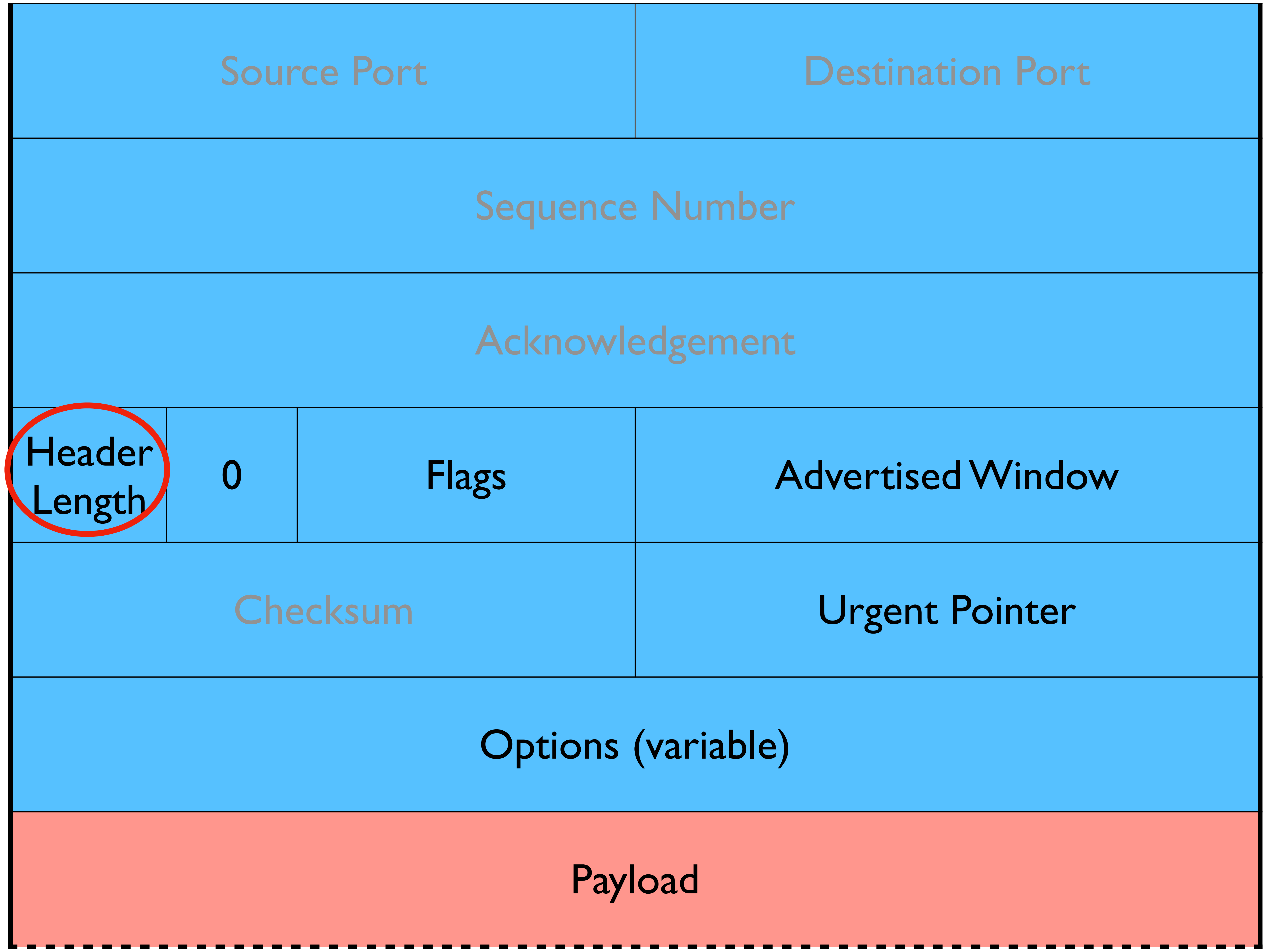
From Jacobson & Karels, SIGCOMM 1988

What does TCP do for reliability?

- **Most of our previous tricks + a few differences**
  - Checksum
  - Sequence numbers are byte-offsets
  - Receiver sends cumulative acknowledgements (like GBN)
  - Receivers can buffer out of sequence packets (like SR)
  - Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
  - Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

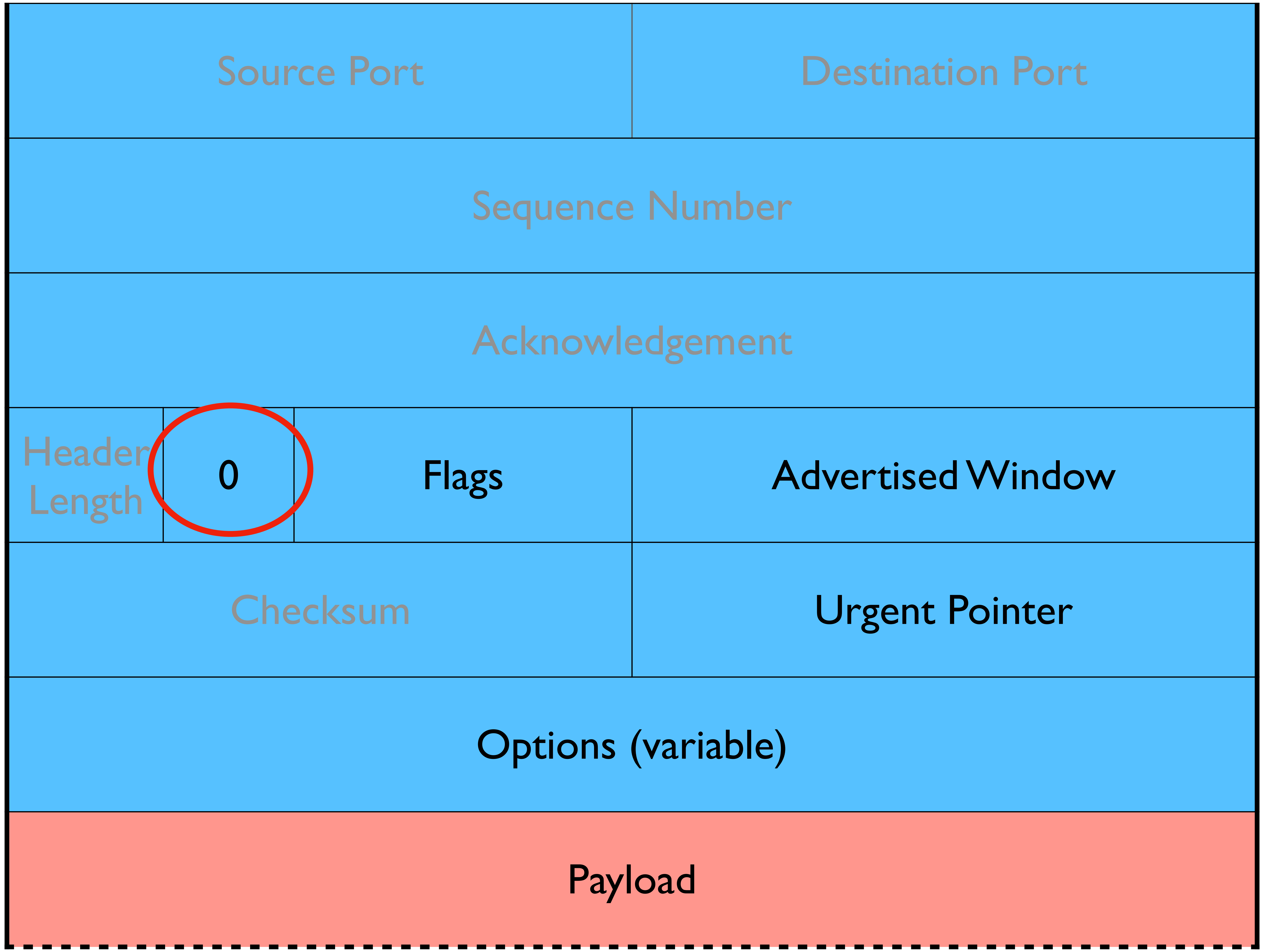
TCP Header: What's Left?

Number of 4-byte words in TCP header; 5 means no options

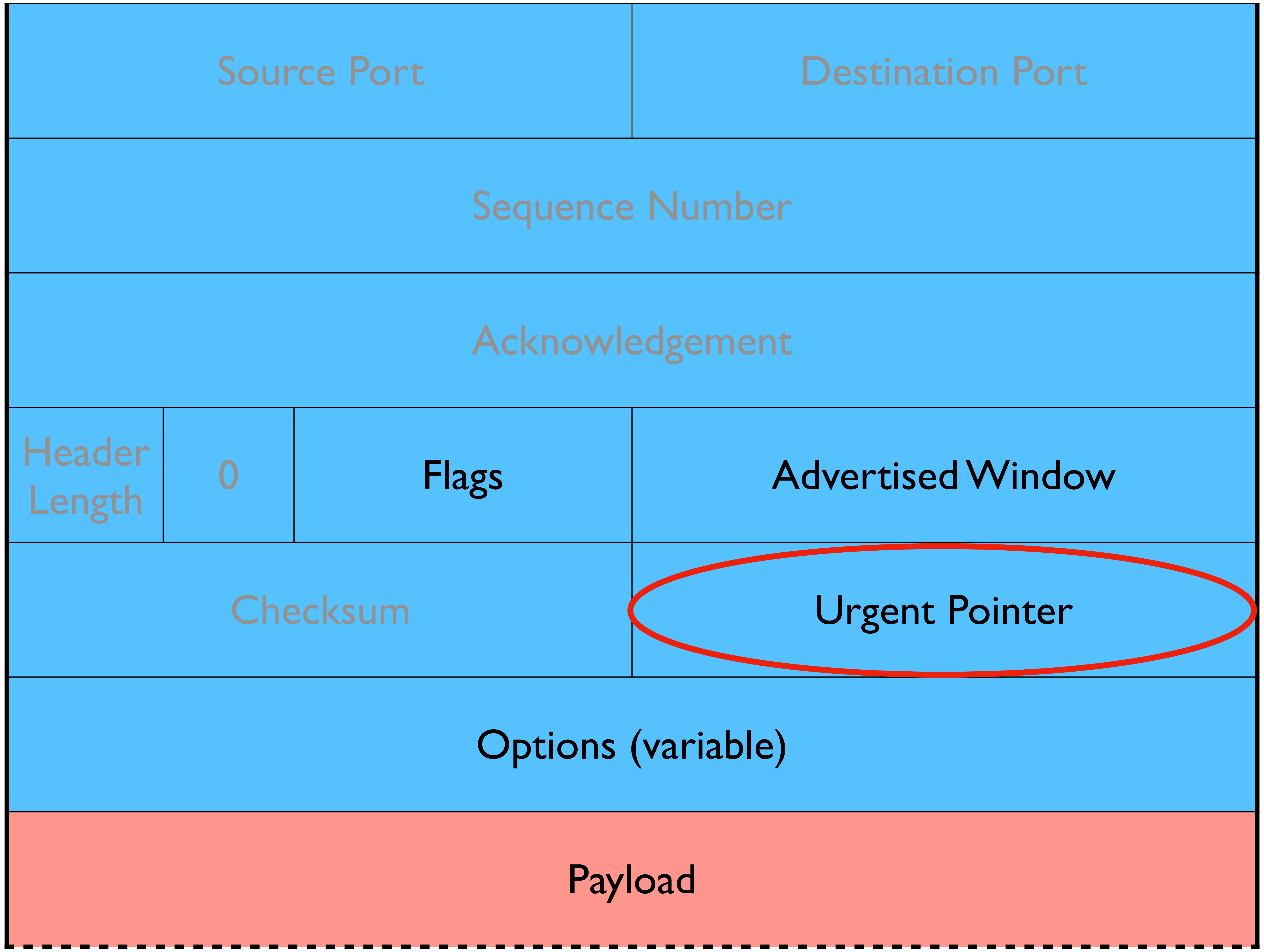


TCP Header: What's Left?

"Must Be Zero"
6 bits reserved

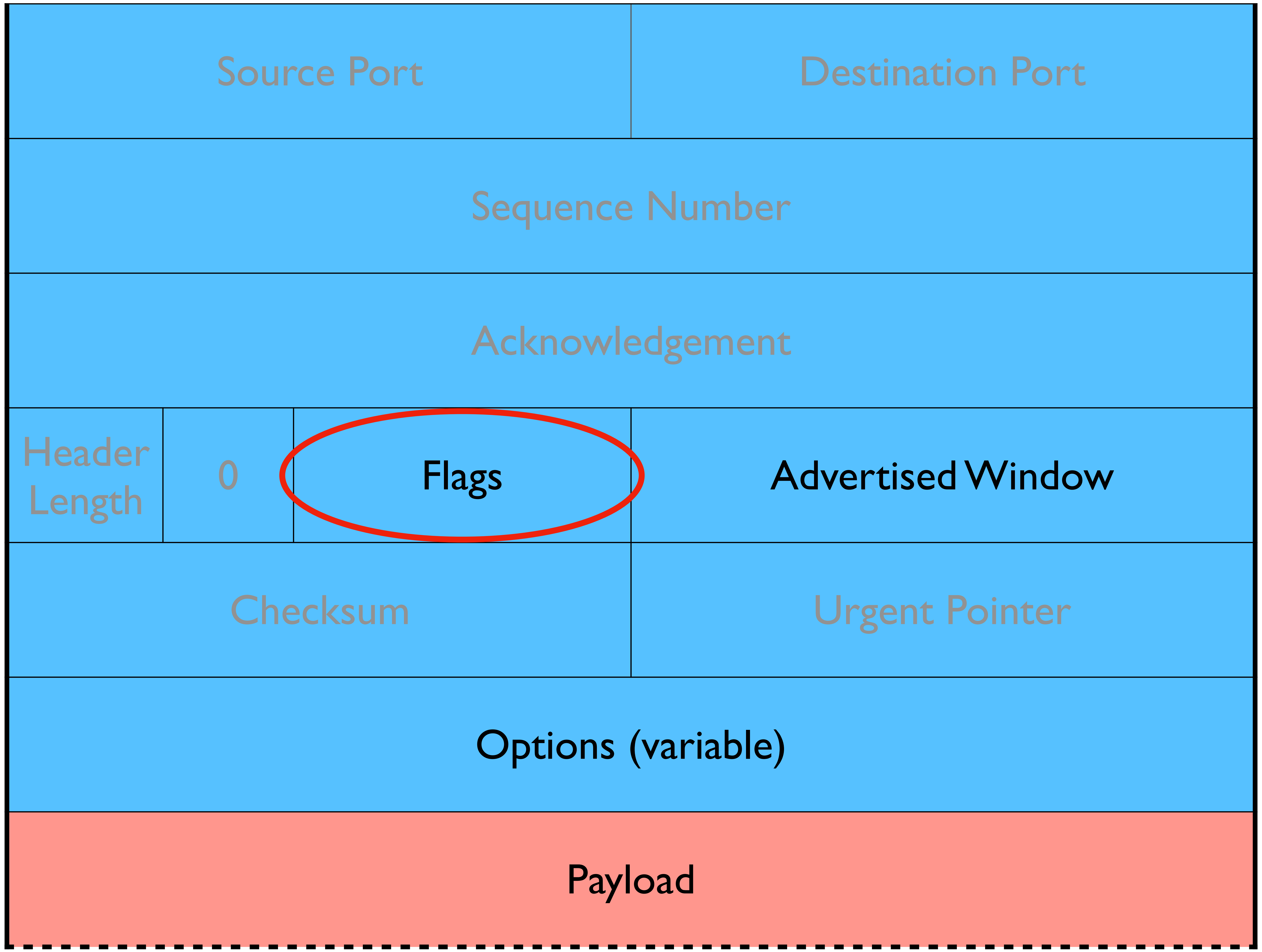


TCP Header: What's Left?



Used with URG flag to indicate urgent data (not discussed further)

TCP Header: What's Left?



TCP Connection Establishment and Initial Sequence Numbers

Initial Sequence Number

Initial Sequence Number

- **Sequence number of the very first byte**

Initial Sequence Number

- **Sequence number of the very first byte**
- **Why not just use ISN=0**

Initial Sequence Number

- **Sequence number of the very first byte**
- **Why not just use ISN=0**
- **Practical Issue:**
  - IP addresses and port numbers uniquely identify a connection
  - Eventually, though, these port numbers do get *used again*
  - … small chance that an old packet is still in flight
  - Also, others might try to spoof your connection
  - *Why does using ISN help?*

Initial Sequence Number

- **Sequence number of the very first byte**
- **Why not just use ISN=0**
- **Practical Issue:**
  - IP addresses and port numbers uniquely identify a connection
  - Eventually, though, these port numbers do get *used again*
  - … small chance that an old packet is still in flight
  - Also, others might try to spoof your connection
  - *Why does using ISN help?*
- TCP therefore <u>requires</u> changing ISN

Initial Sequence Number

- **Sequence number of the very first byte**
- **Why not just use ISN=0**
- **Practical Issue:**
  - IP addresses and port numbers uniquely identify a connection
  - Eventually, though, these port numbers do get *used again*
  - … small chance that an old packet is still in flight
  - Also, others might try to spoof your connection
  - *Why does using ISN help?*
- TCP therefore <u>requires</u> changing ISN
- Hosts exchange ISNs when they establish a connection

Establishing a TCP Connection

Establishing a TCP Connection

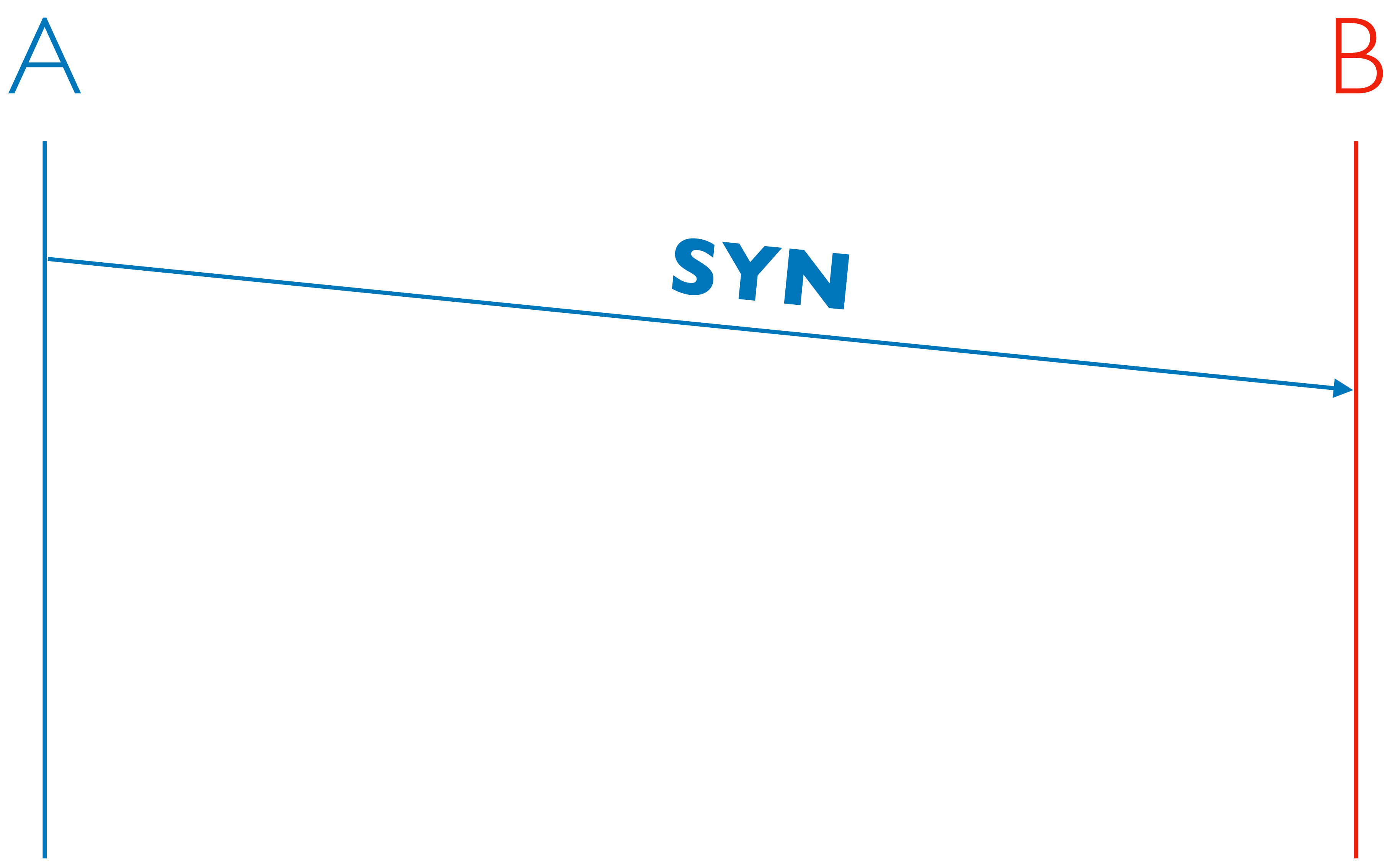
Each host tells its ISN to the other host

Establishing a TCP Connection

Each host tells its ISN to the other host

- **Three-way handshake to establish connection**

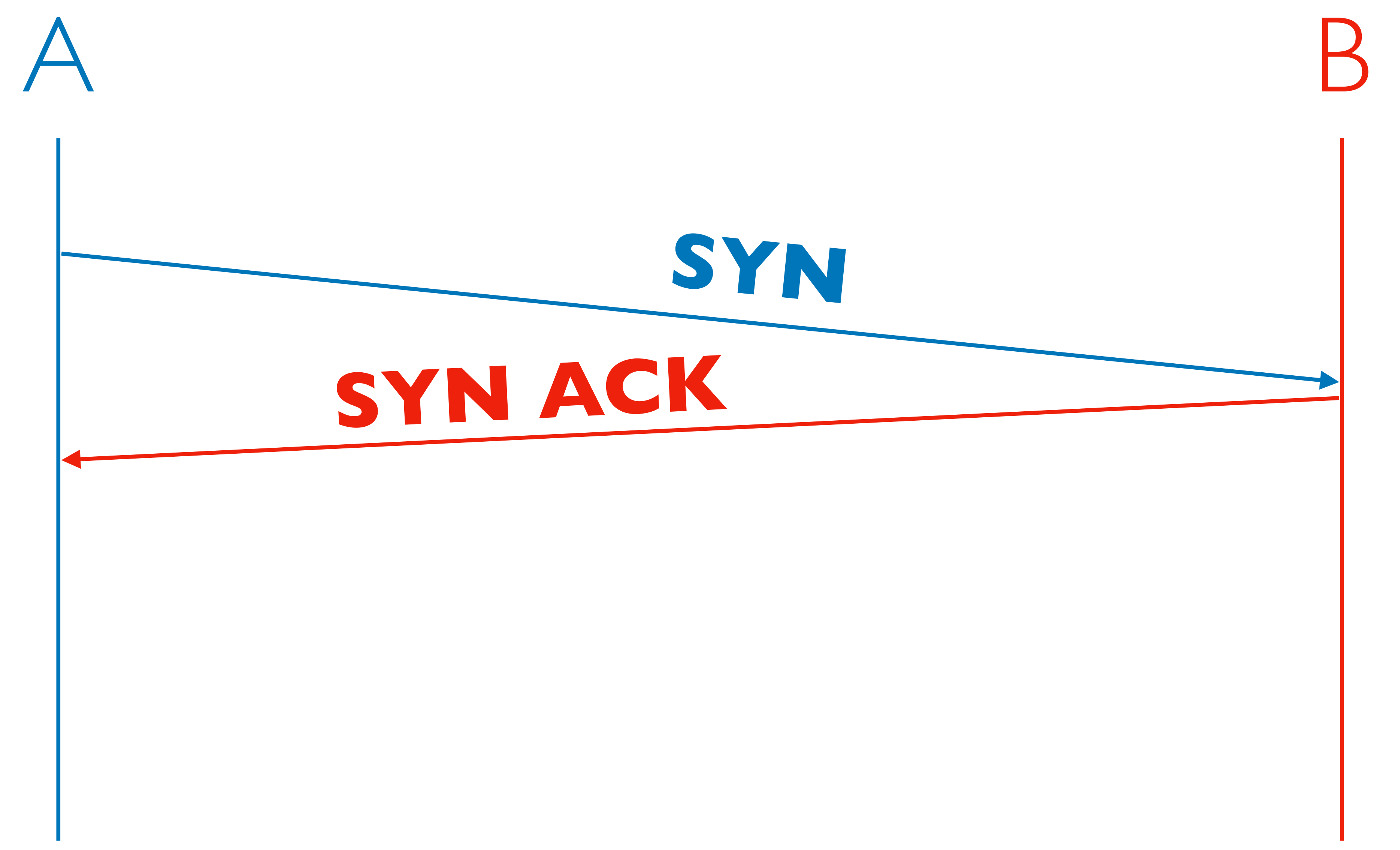
Establishing a TCP Connection



Each host tells its ISN to the other host

- **Three-way handshake to establish connection**
  - Host A sends a **SYN** (open; "synchronize sequence numbers") to Host B

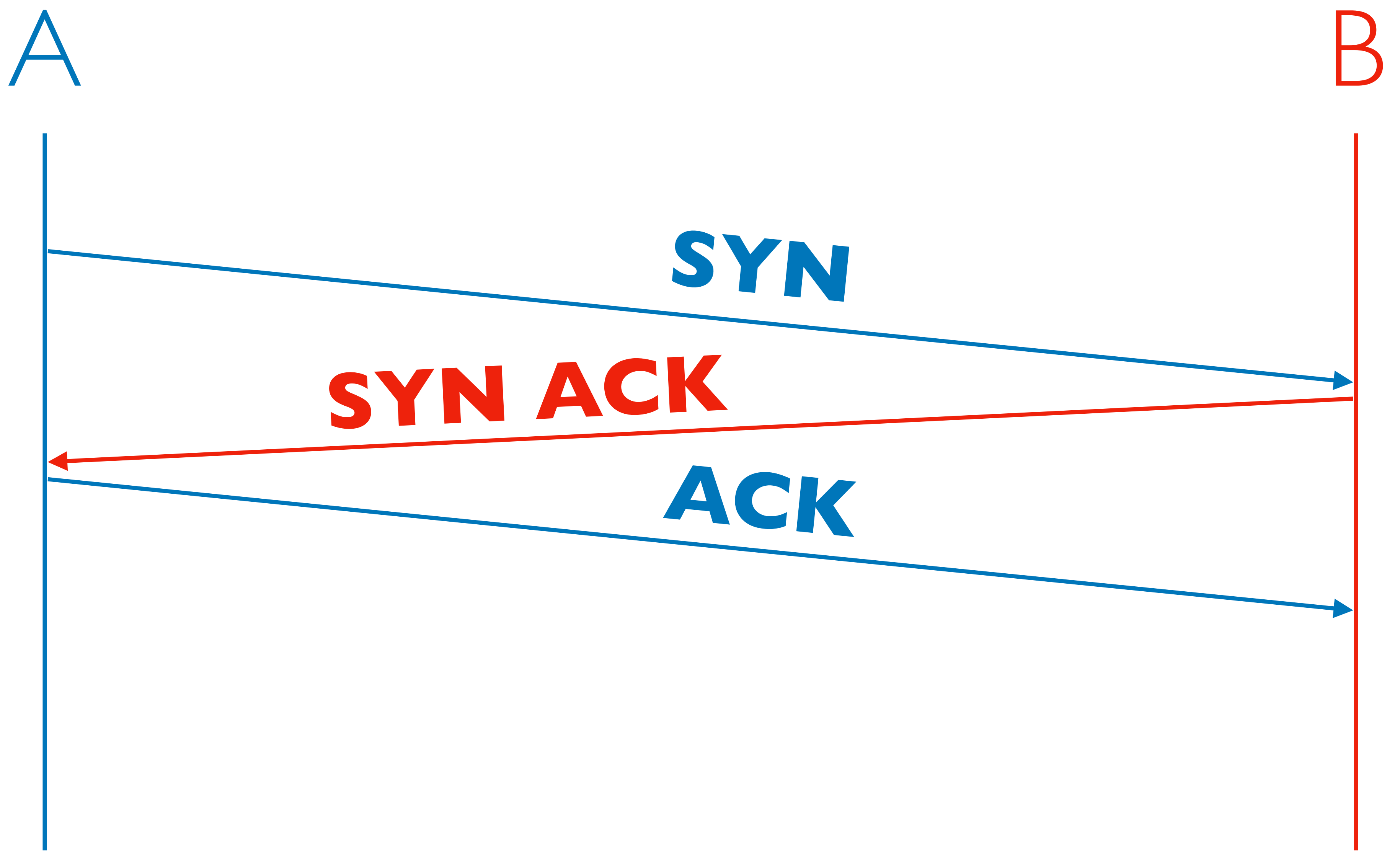
Establishing a TCP Connection



Each host tells its ISN to the other host

- **Three-way handshake to establish connection**
  - Host A sends a **SYN** (open; "synchronize sequence numbers") to Host B
  - Host B returns a SYN Acknowledgement (**SYN ACK**)

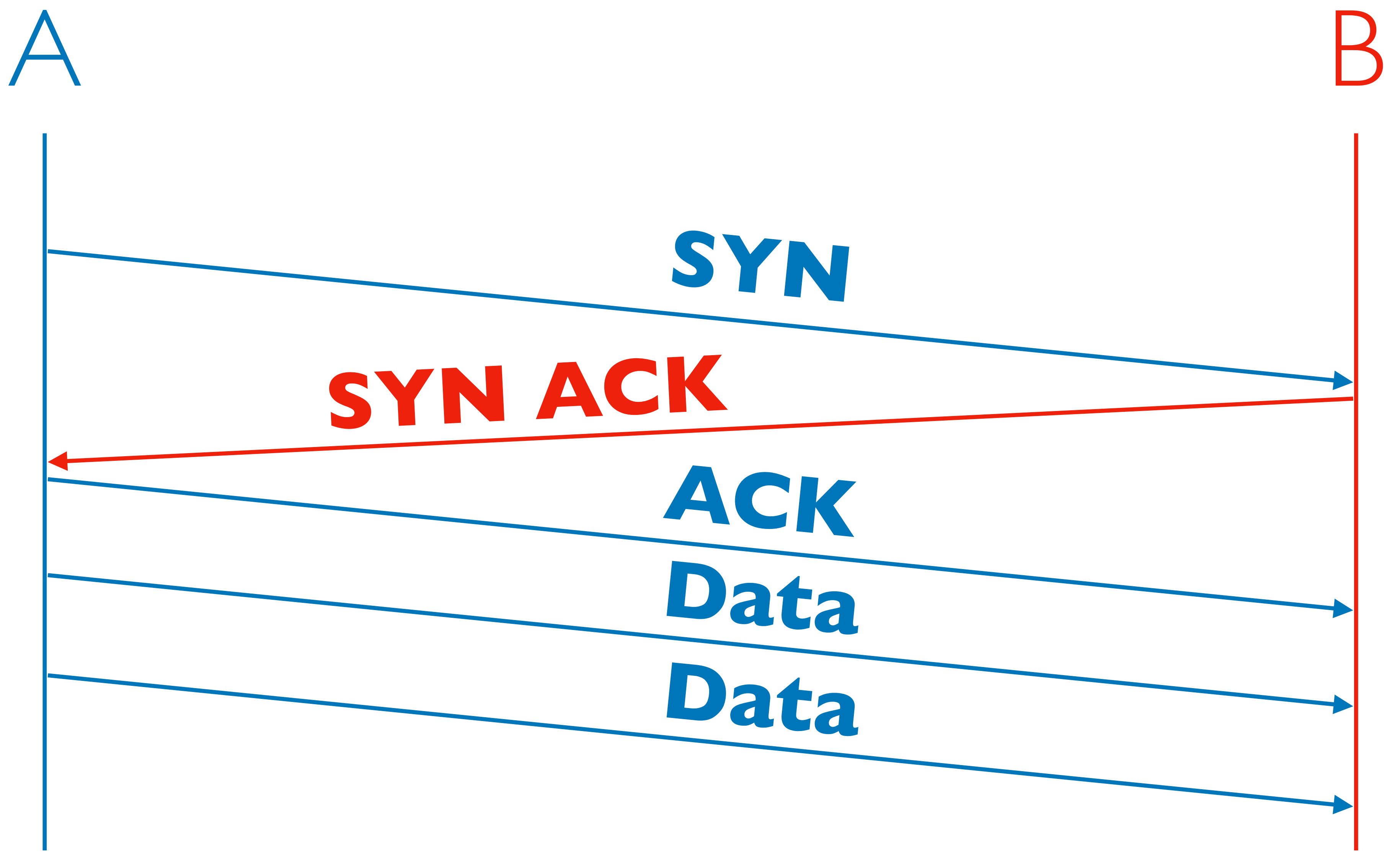
Establishing a TCP Connection



Each host tells its ISN to the other host

- **Three-way handshake to establish connection**
  - Host A sends a **SYN** (open; "synchronize sequence numbers") to Host B
  - Host B returns a SYN Acknowledgement (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

Establishing a TCP Connection

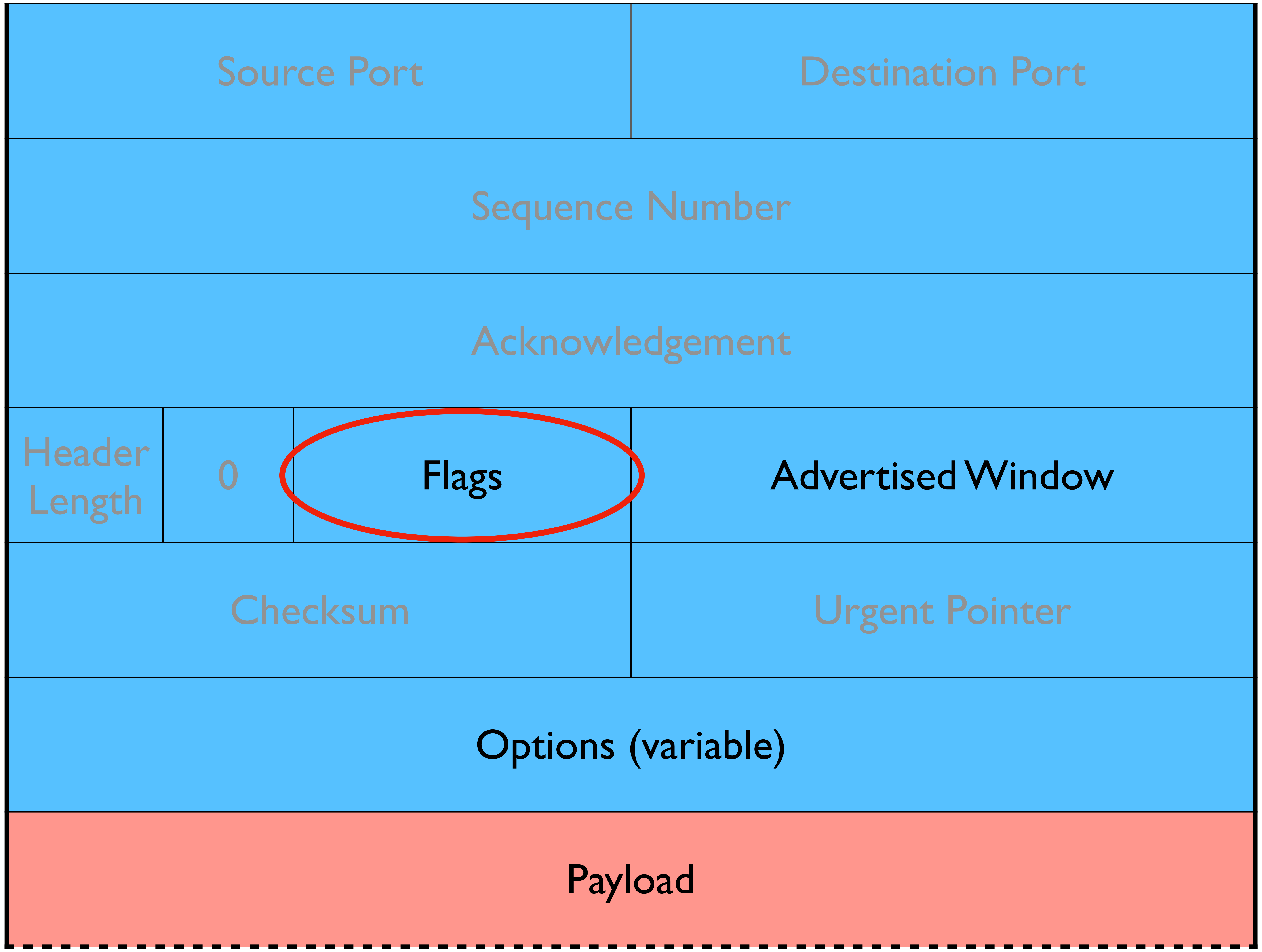


Each host tells its ISN to the other host

- **Three-way handshake to establish connection**
  - Host A sends a **SYN** (open; "synchronize sequence numbers") to Host B
  - Host B returns a SYN Acknowledgement (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK

TCP Header

SYN
ACK
FIN
RST
PSH
URG



Step 1: A's Initial SYN Packet

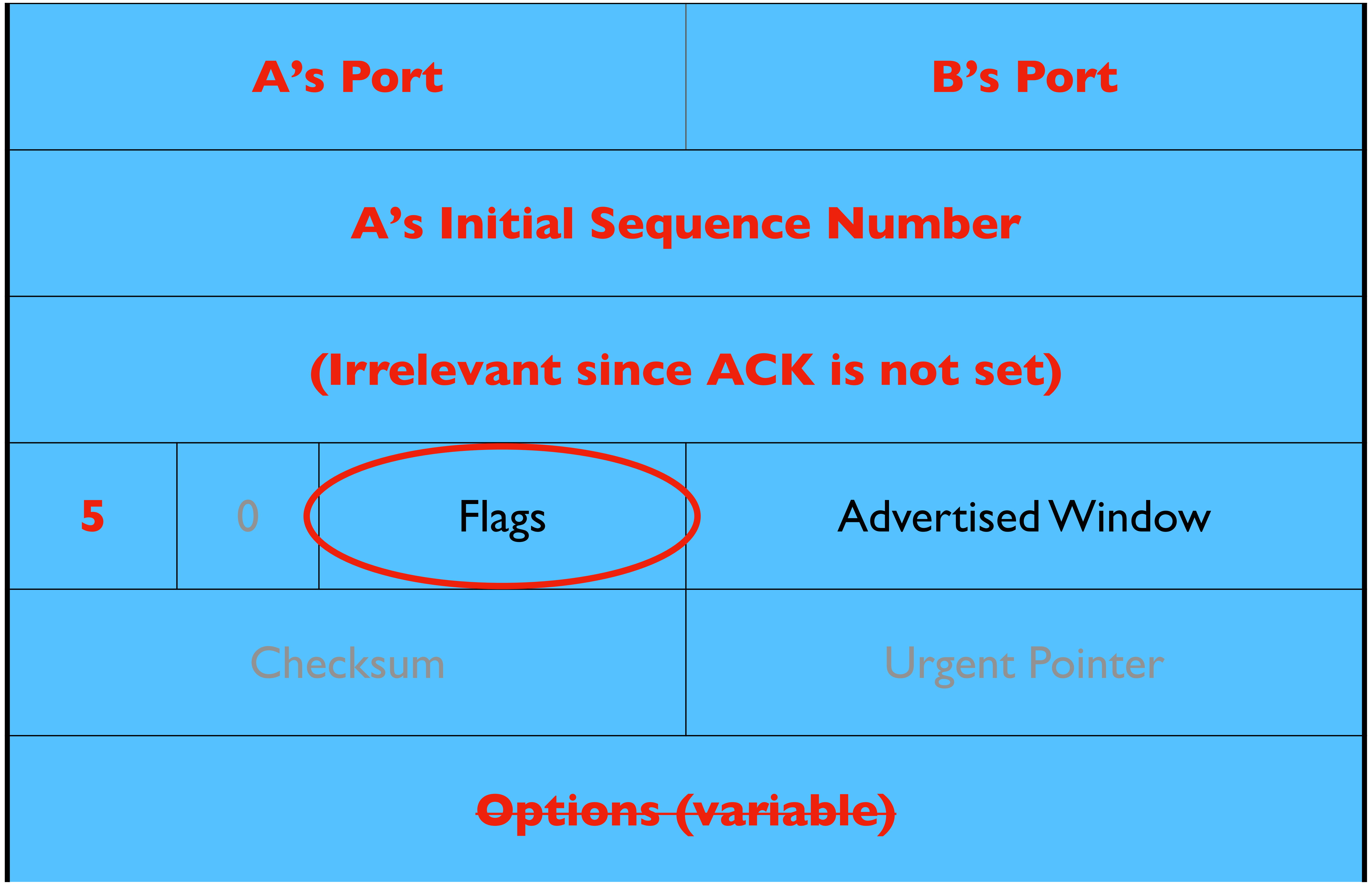
SYN
ACK
FIN
RST
PSH
URG

A's Port			**B's Port**
A's Initial Sequence Number			
(Irrelevant since ACK is not set)			
5	0	Flags	Advertised Window
Checksum			Urgent Pointer
~~**Options (variable)**~~			

Step 1: A's Initial SYN Packet

A tells B it wants to open a connection…

SYN
ACK
FIN
RST
PSH
URG



Step 2: B's SYN ACK Packet

SYN
ACK
FIN
RST
PSH
URG

B's Port			**A's Port**
B's Initial Sequence Number			
ACK = ISN of A plus one			
5	0	Flags	Advertised Window
Checksum			Urgent Pointer
~~**Options (variable)**~~			

Step 2: B's SYN ACK Packet

B tells A it accepts, and is reading to hear the next byte

SYN
ACK
FIN
RST
PSH
URG

B's Port			**A's Port**
B's Initial Sequence Number			
ACK = ISN of A plus one			
5	0	Flags	Advertised Window
Checksum			Urgent Pointer
~~**Options (variable)**~~			

Step 2: B's SYN ACK Packet

B tells A it accepts, and is reading to hear the next byte

SYN
ACK
FIN
RST
PSH
URG

B's Port			A's Port
B's Initial Sequence Number			
ACK = ISN of A plus one			
5	0	Flags	Advertised Window
Checksum			Urgent Pointer
~~Options (variable)~~			

… on receiving this packet, A can start sending data

Step 2: A's ACK of the SYN ACK

SYN
ACK
FIN
RST
PSH
URG

<table>
<tr><td colspan="3">A's Port</td><td>B's Port</td></tr>
<tr><td colspan="4">A's Initial Sequence Number</td></tr>
<tr><td colspan="4">ACK = ISN of B plus one</td></tr>
<tr><td>5</td><td>0</td><td>Flags</td><td>Advertised Window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent Pointer</td></tr>
<tr><td colspan="4">~~Options (variable)~~</td></tr>
</table>

Step 2: A's ACK of the SYN ACK

A tells B its likewise okay to start sending data

SYN
ACK
FIN
RST
PSH
URG

<table>
<tr><td colspan="3">A's Port</td><td>B's Port</td></tr>
<tr><td colspan="4">A's Initial Sequence Number</td></tr>
<tr><td colspan="4">ACK = ISN of B plus one</td></tr>
<tr><td>5</td><td>0</td><td>Flags</td><td>Advertised Window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent Pointer</td></tr>
<tr><td colspan="4">~~Options (variable)~~</td></tr>
</table>

Step 2: A's ACK of the SYN ACK

A tells B its likewise okay to start sending data

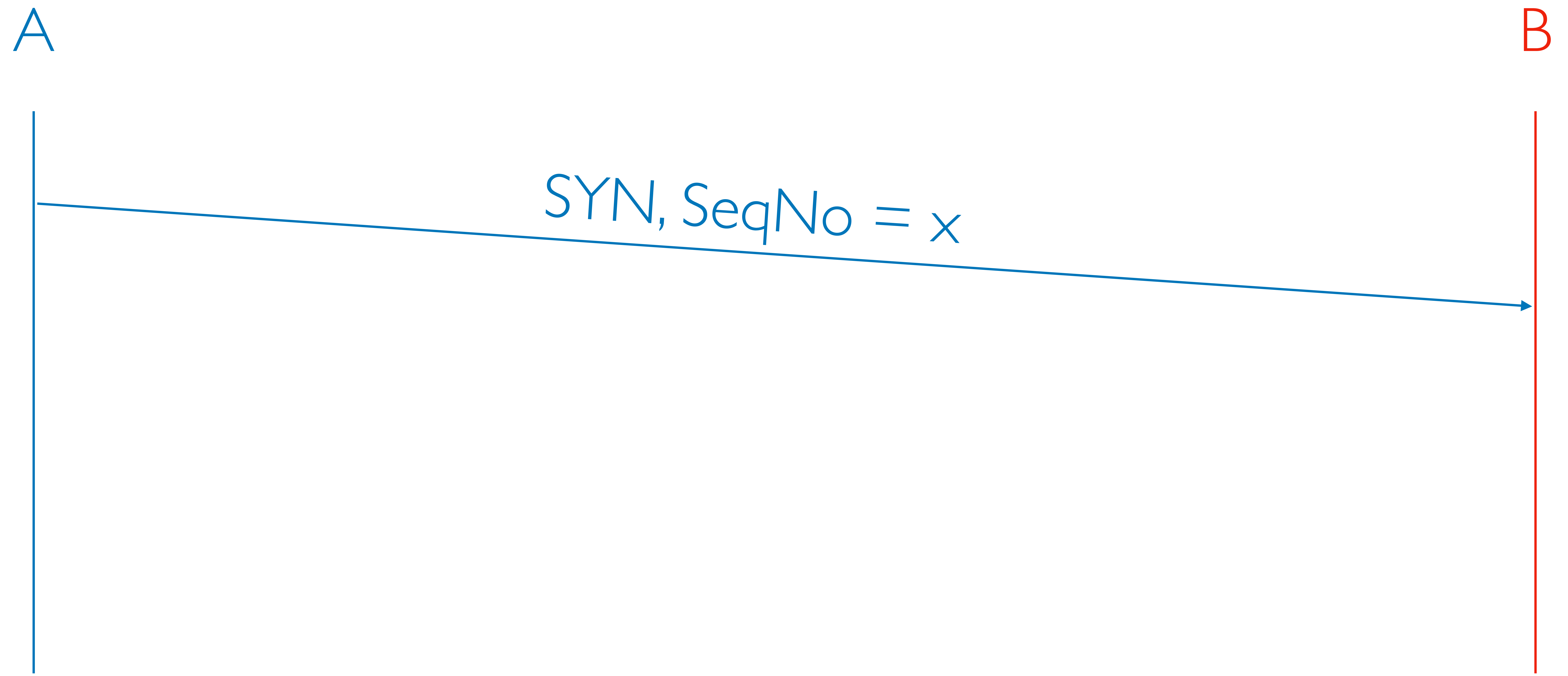
SYN
ACK
FIN
RST
PSH
URG

<table>
<tr><td colspan="3">A's Port</td><td>B's Port</td></tr>
<tr><td colspan="4">A's Initial Sequence Number</td></tr>
<tr><td colspan="4">ACK = ISN of B plus one</td></tr>
<tr><td>5</td><td>0</td><td>Flags</td><td>Advertised Window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent Pointer</td></tr>
<tr><td colspan="4">~~Options (variable)~~</td></tr>
</table>

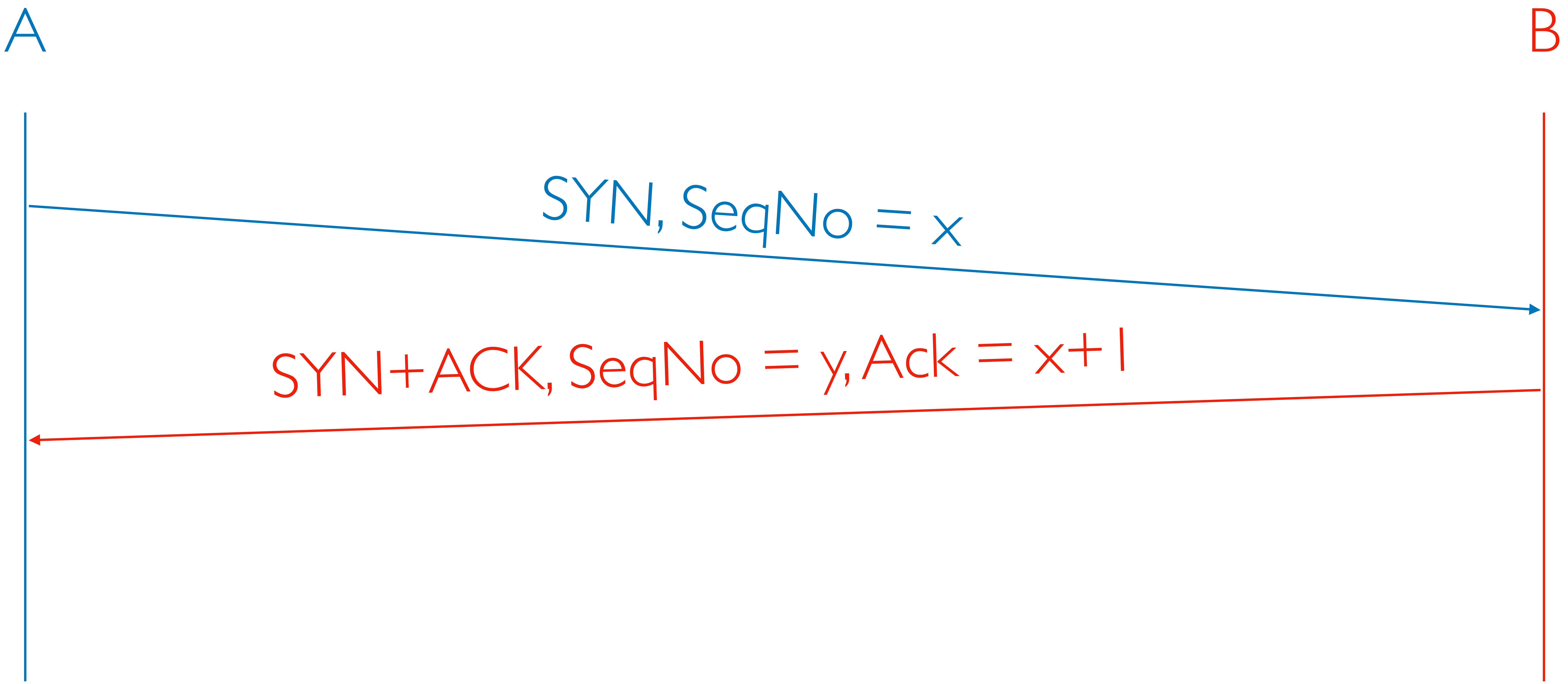
… on receiving this packet, B can start sending data

Timing Diagram: 3-way Handshaking

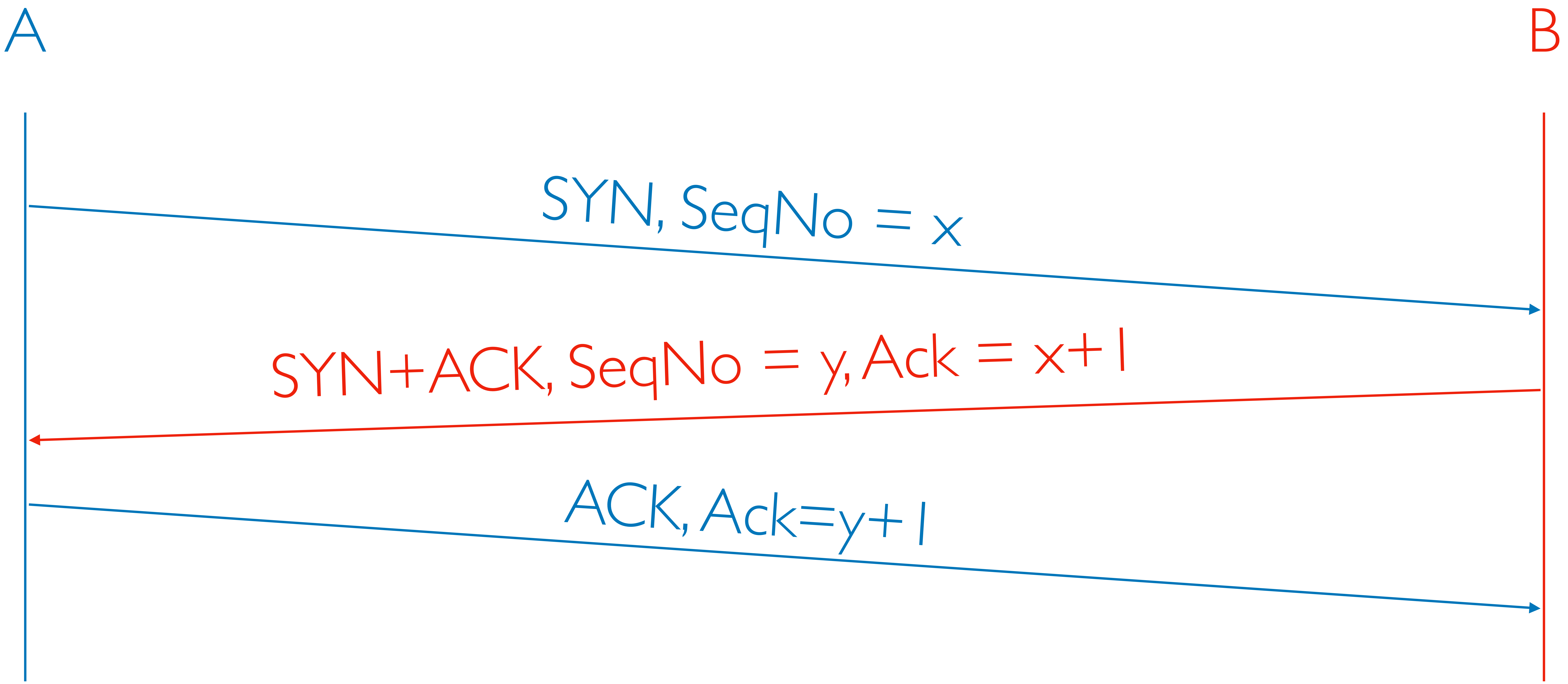
Timing Diagram: 3-way Handshaking



Timing Diagram: 3-way Handshaking



Timing Diagram: 3-way Handshaking



What if the SYN Packet Gets Lost?

What if the SYN Packet Gets Lost?

- **Suppose the SYN packet gets lost**
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., its too busy)

What if the SYN Packet Gets Lost?

- **Suppose the SYN packet gets lost**
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., its too busy)
- **Eventually, no SYN ACK arrives**
  - Sender sets a timer and waits for the SYN ACK
  - … and retransmits the SYN if needed

What if the SYN Packet Gets Lost?

- **Suppose the SYN packet gets lost**
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., its too busy)
- **Eventually, no SYN ACK arrives**
  - Sender sets a timer and waits for the SYN ACK
  - … and retransmits the SYN if needed
- **How should the TCP sender set the timer?**
  - Sender has no idea how far away the receiver is
  - Hard to guess a reasonable length of time to wait
  - ***Should*** (RFCs 1122 & 2988) use default of 3 seconds
    - Some implementations instead use 6 seconds

Are you one of those people?

Are you one of those people?

- **Your friend sends you a link (to a cool new meme)**
    - You click on it (of course, duh)

Are you one of those people?

- **Your friend sends you a link (to a cool new meme)**
  - You click on it (of course, duh)
- **Page is taking too long to load**
  - Ugghhh!!

Are you one of those people?

- **Your friend sends you a link (to a cool new meme)**
  - You click on it (of course, duh)
- **Page is taking too long to load**
  - Ugghhh!!
- **CLICK AGAIN LIKE A THOUSAND TIMES!!!**
  - Or…

Are you one of those people?

- **Your friend sends you a link (to a cool new meme)**
  - You click on it (of course, duh)
- **Page is taking too long to load**
  - Ugghhh!!
- **CLICK AGAIN LIKE A THOUSAND TIMES!!!**
  - Or…
- **REFRESH LIKE A THOUSAND TIMES!!!**

Are you one of those people?

- **Your friend sends you a link (to a cool new meme)**
  - You click on it (of course, duh)
- **Page is taking too long to load**
  - Ugghhh!!
- **CLICK AGAIN LIKE A THOUSAND TIMES!!!**
  - Or…
- **REFRESH LIKE A THOUSAND TIMES!!!**
- Do you ***really*** think this helps?

… well maybe!

… well maybe!

- **You click on a link**
  - Browser creates a socket and does a "connect"
  - The "connect" triggers the OS to transmit a SYN

… well maybe!

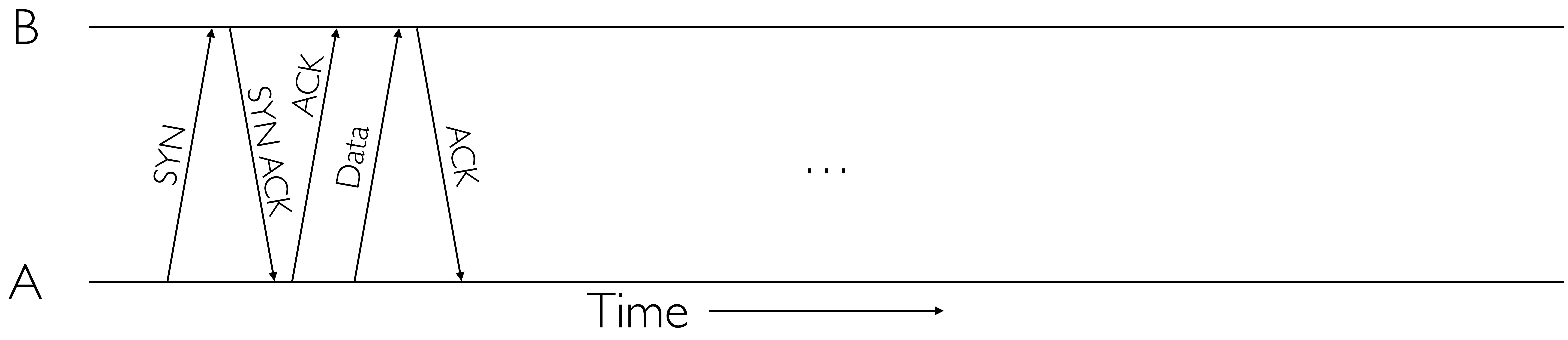
- **You click on a link**
  - Browser creates a socket and does a "connect"
  - The "connect" triggers the OS to transmit a SYN
- **If the SYN is lost…**
  - 3-6 seconds of delay: can be **very long** (especially for a meme)
  - You become impatient
  - … and click the link again, or click "reload" incessantly

… well maybe!

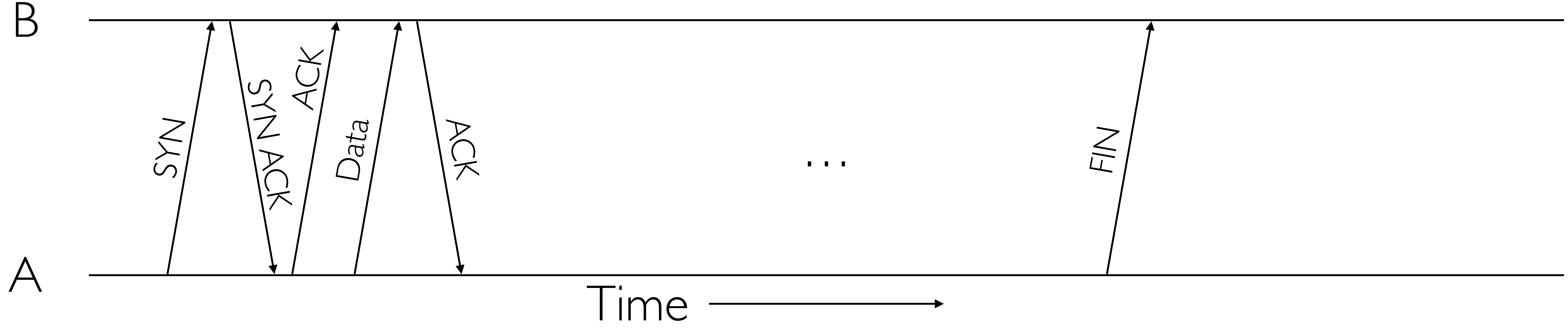
- **You click on a link**
  - Browser creates a socket and does a "connect"
  - The "connect" triggers the OS to transmit a SYN
- **If the SYN is lost…**
  - 3-6 seconds of delay: can be **very long** (especially for a meme)
  - You become impatient
  - … and click the link again, or click "reload" incessantly
- **You inadvertently trigger an "abort" of the "connect"**
  - Browser creates a *new* socket and does another "connect"
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page loads quickly

Tearing Down the Connection

Normal Termination, One Side At a Time

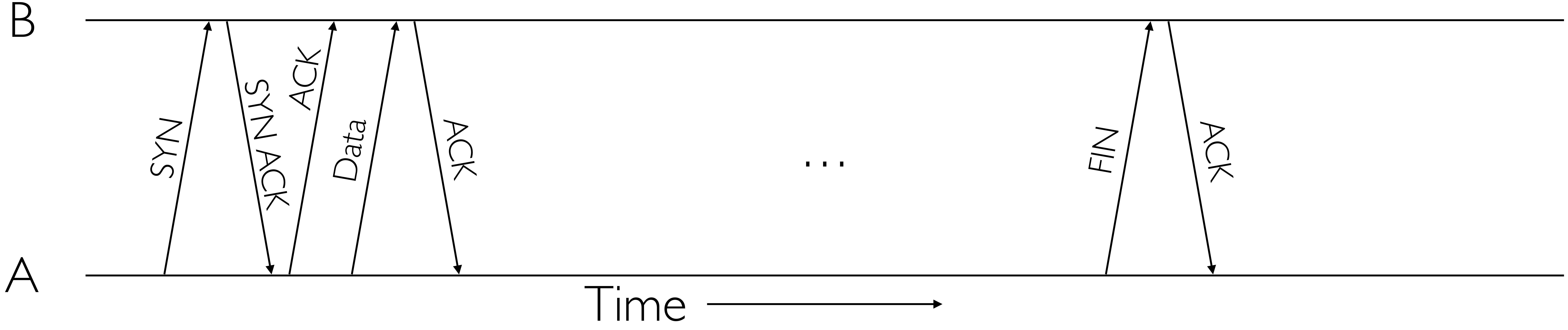


Normal Termination, One Side At a Time



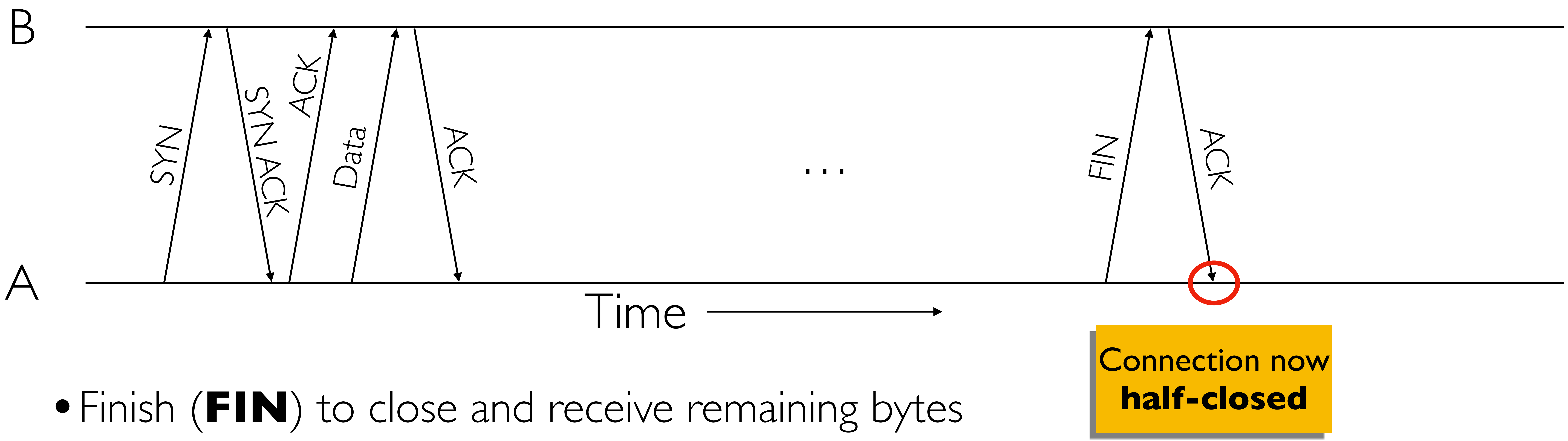
- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space

Normal Termination, One Side At a Time



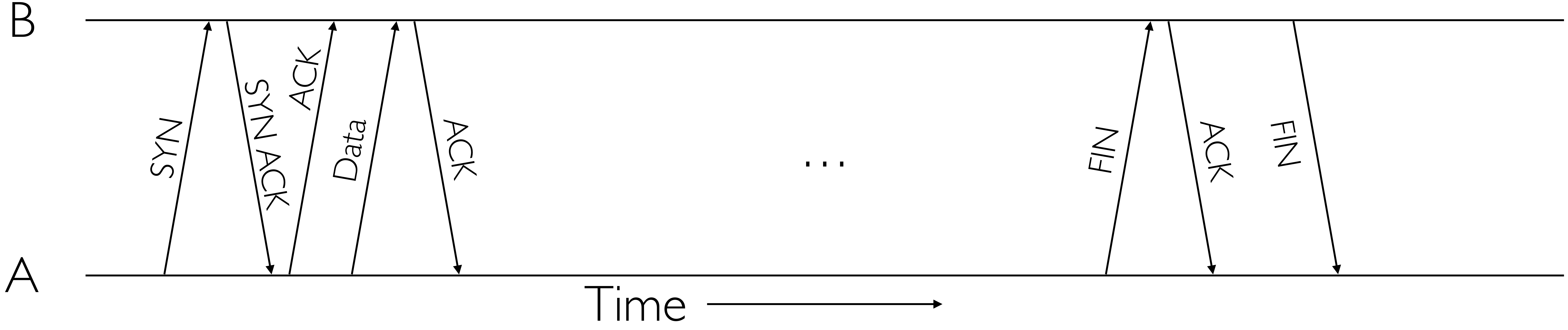
- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm

Normal Termination, One Side At a Time



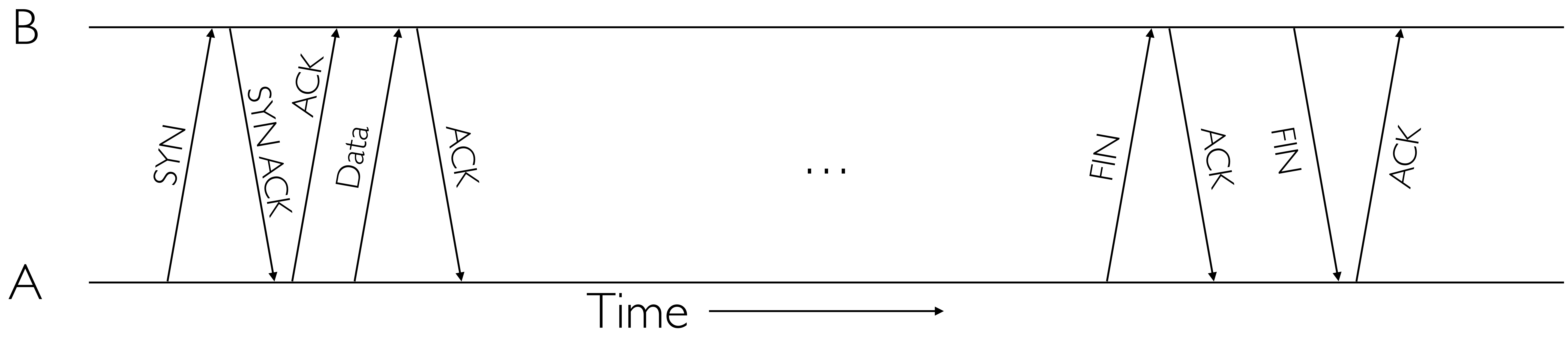
- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's

Normal Termination, One Side At a Time



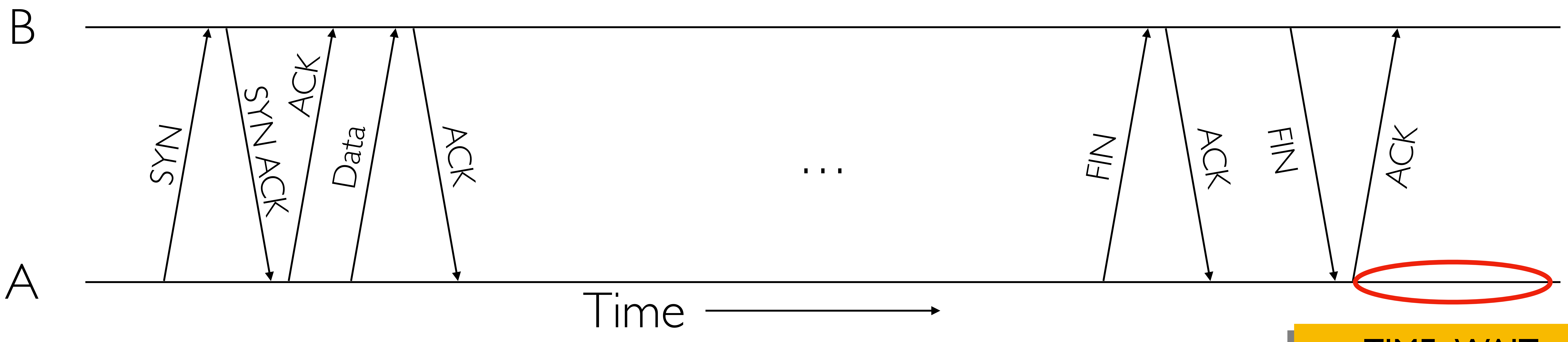
- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's
  - Unti B likewise sends a **FIN**

Normal Termination, One Side At a Time



- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's
  - Unti B likewise sends a **FIN**
  - Which A then ACKs

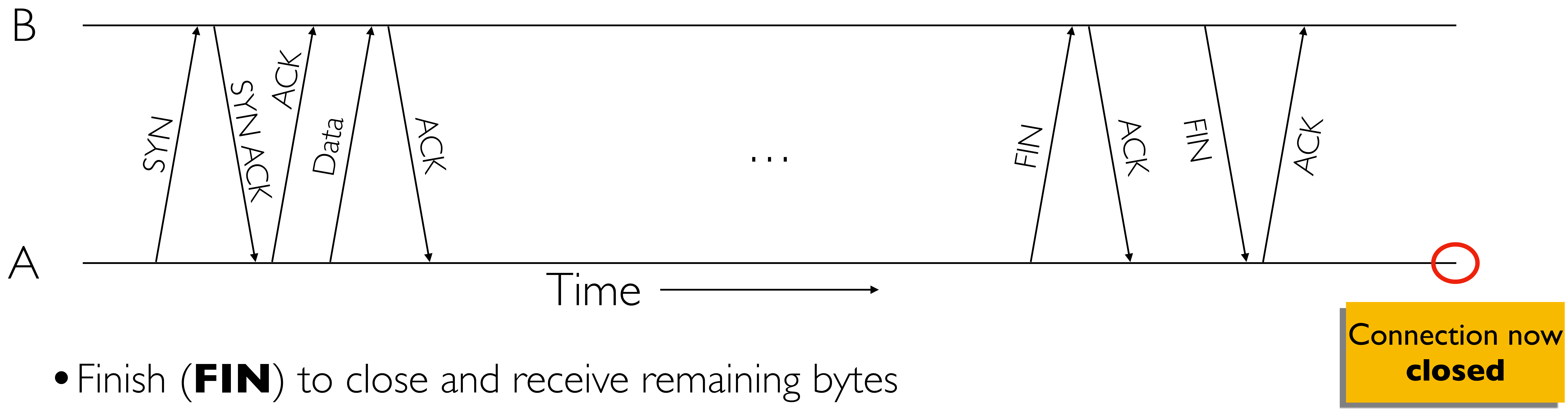
Normal Termination, One Side At a Time



TIME_WAIT:
Avoid reincarnation;
B will retransmit FIN if
ACK is lost

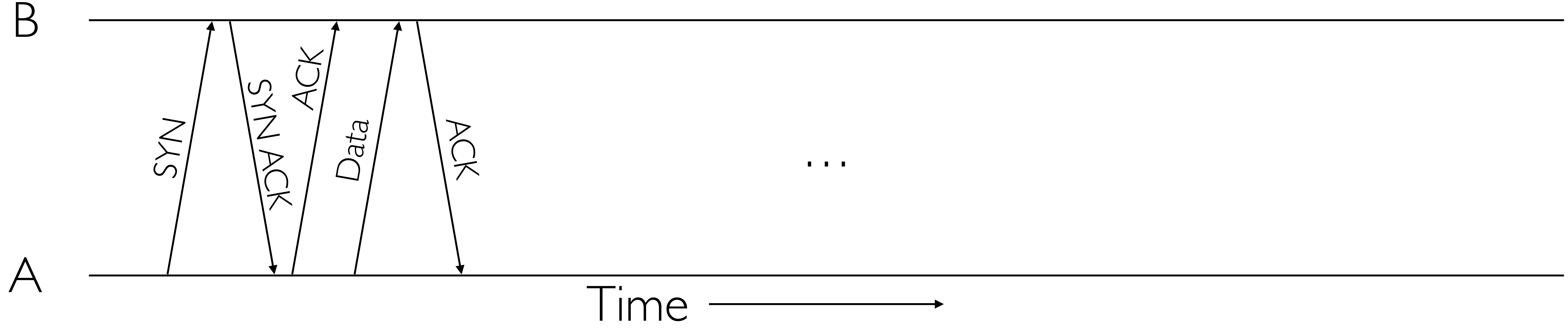
- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's
  - Unti B likewise sends a **FIN**
  - Which A then ACKs

Normal Termination, One Side At a Time

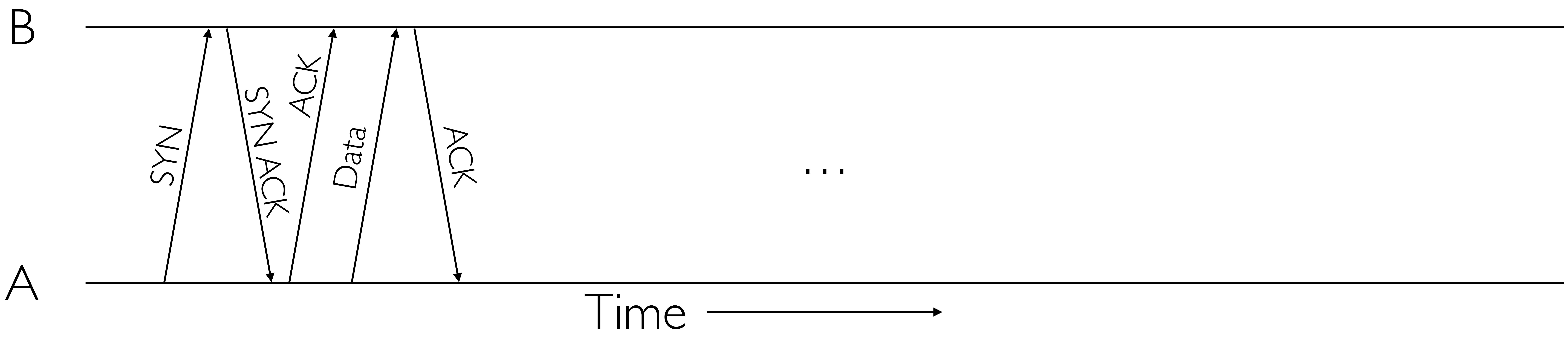


- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host ACKs the byte to confirm
- Closes A's side of the connection but *not* B's
  - Unti B likewise sends a **FIN**
  - Which A then ACKs

Normal Termination, Both Together

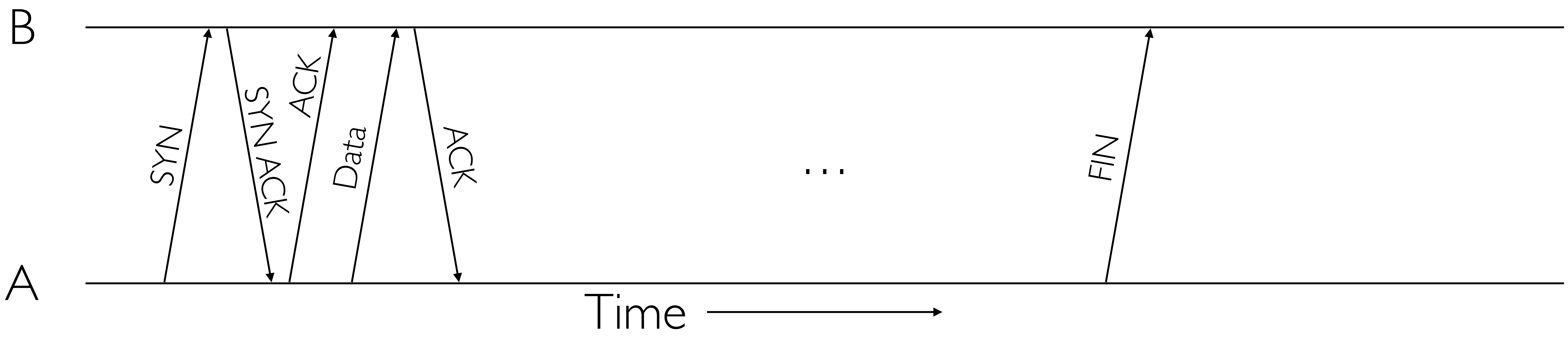


Normal Termination, Both Together



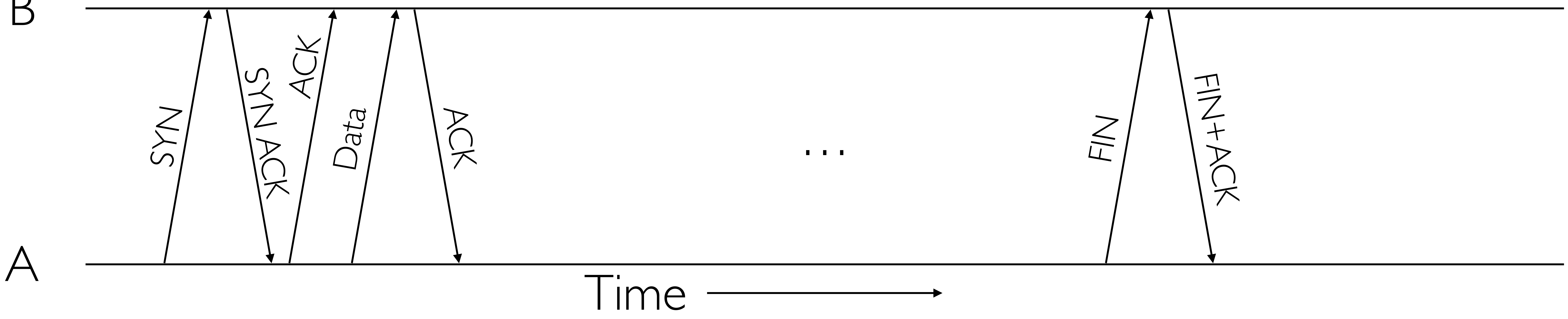
- Same as before, but B sets FIN with their ACK of A's FIN

Normal Termination, Both Together



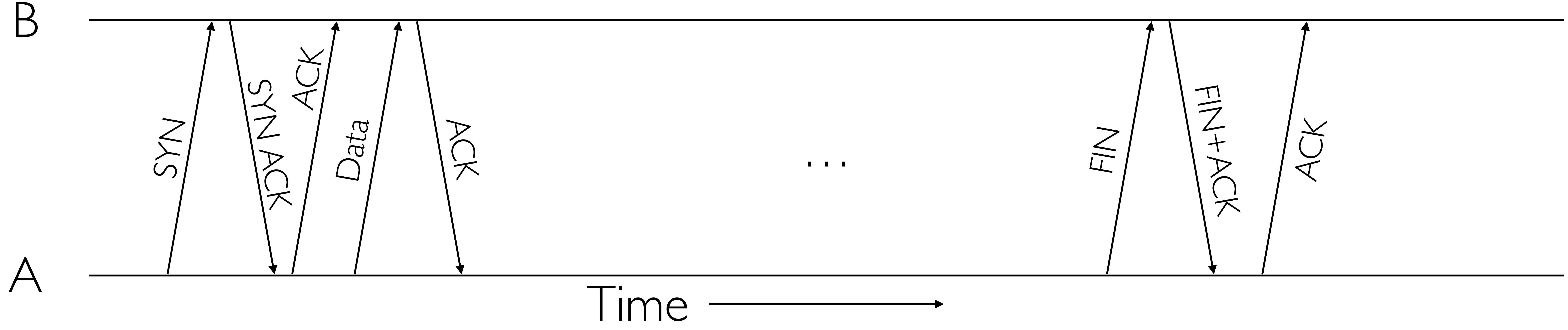
- Same as before, but B sets FIN with their ACK of A's FIN

Normal Termination, Both Together



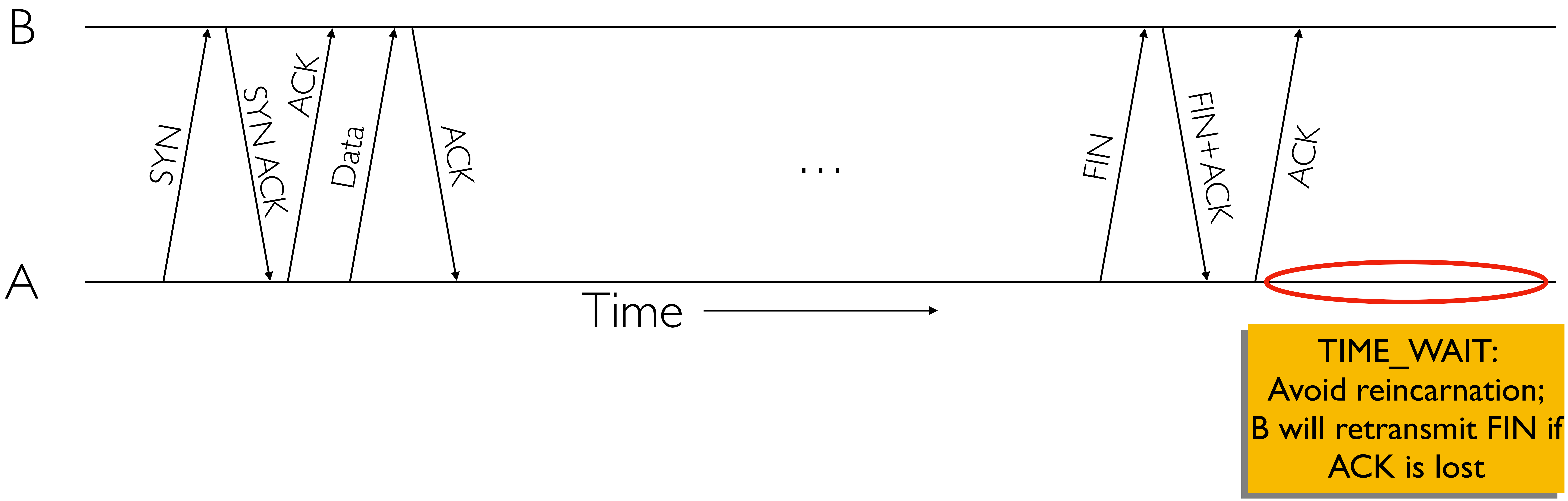
- Same as before, but B sets FIN with their ACK of A's FIN

Normal Termination, Both Together



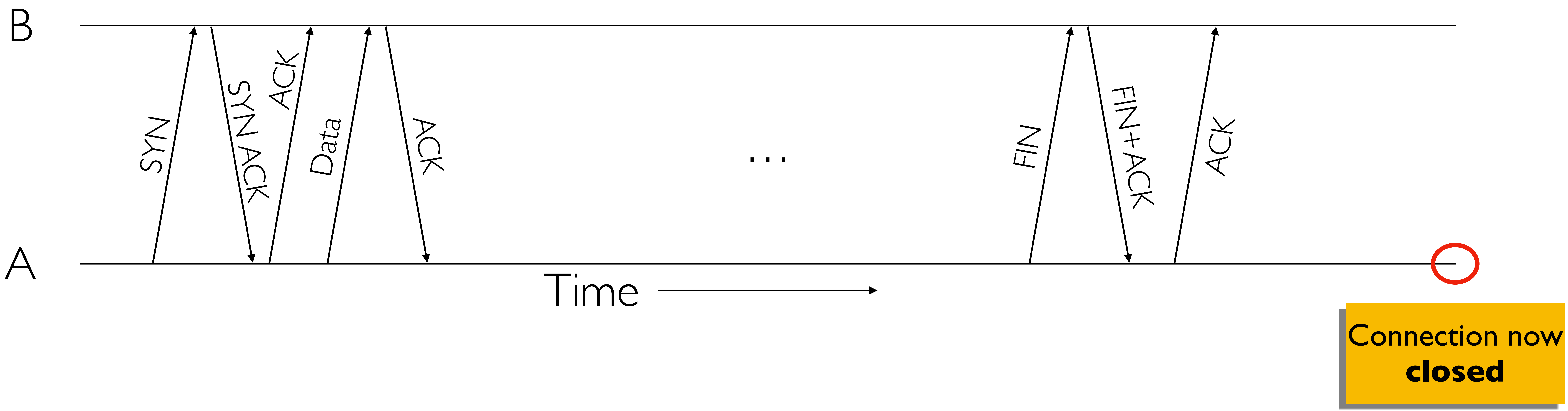
- Same as before, but B sets FIN with their ACK of A's FIN

Normal Termination, Both Together



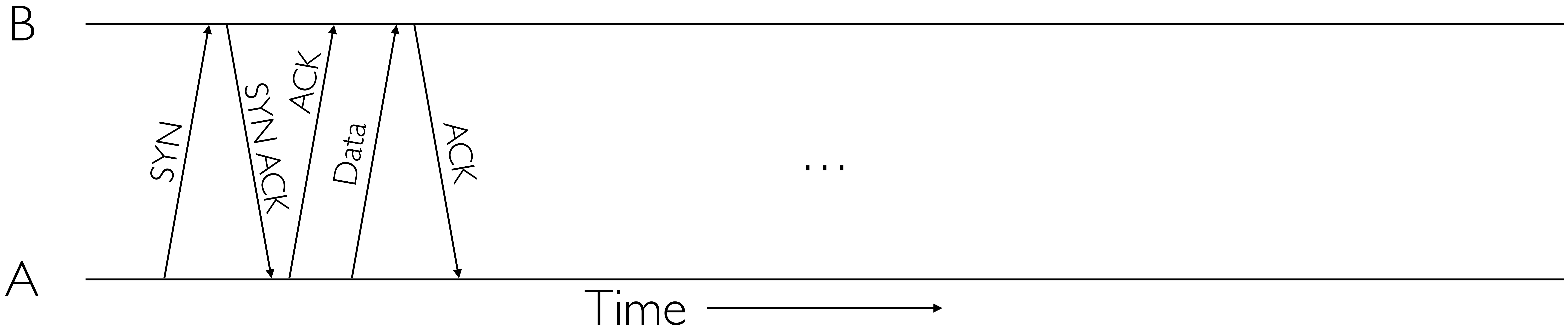
- Same as before, but B sets FIN with their ACK of A's FIN

Normal Termination, Both Together

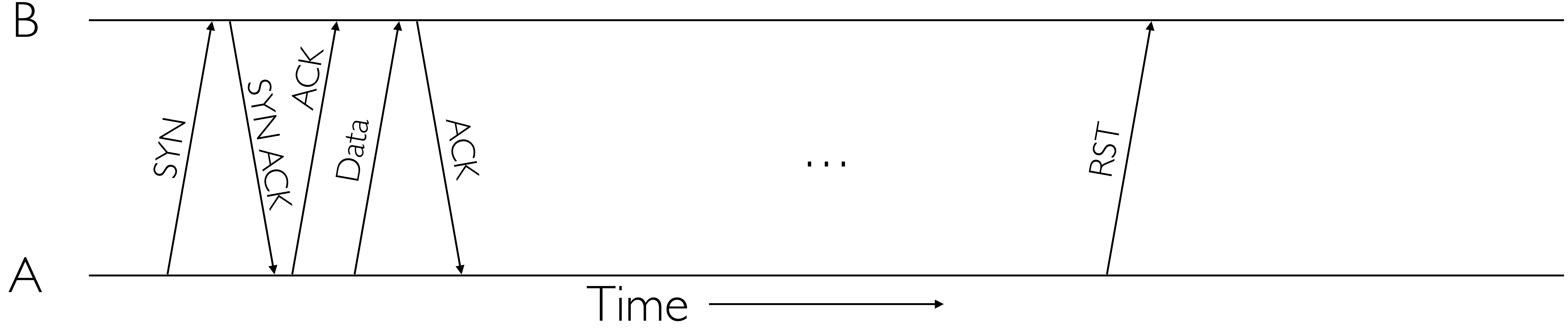


- Same as before, but B sets FIN with their ACK of A's FIN

Abrupt Termination

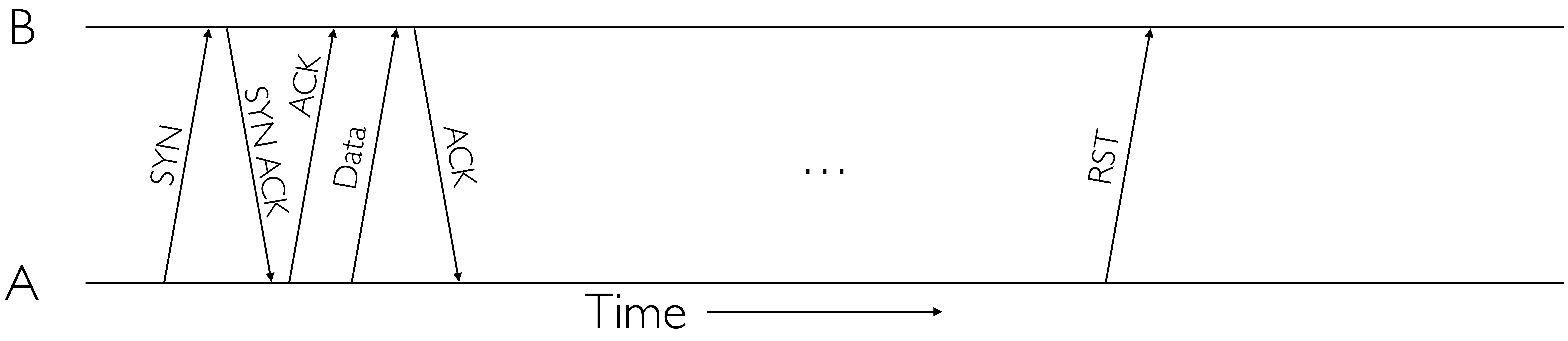


Abrupt Termination



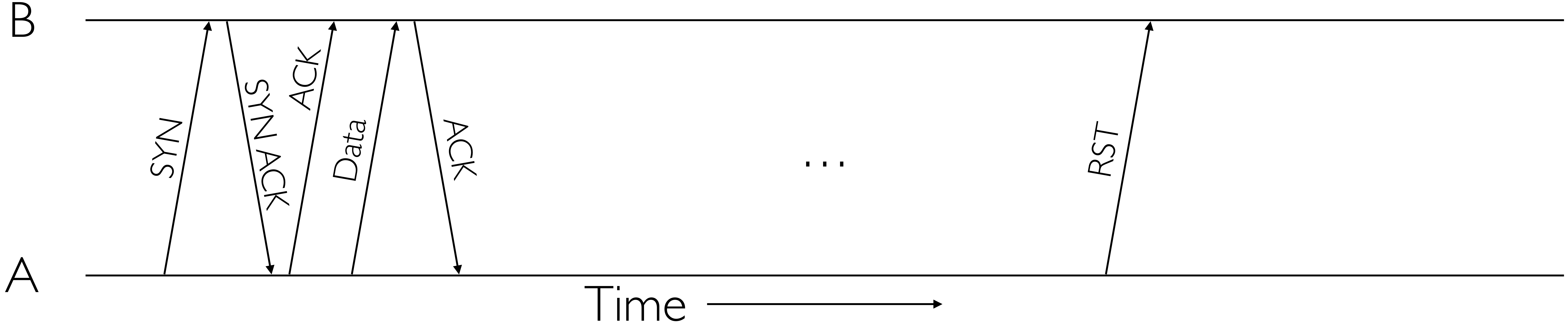
- A sends RESET (**RST**) to B

Abrupt Termination



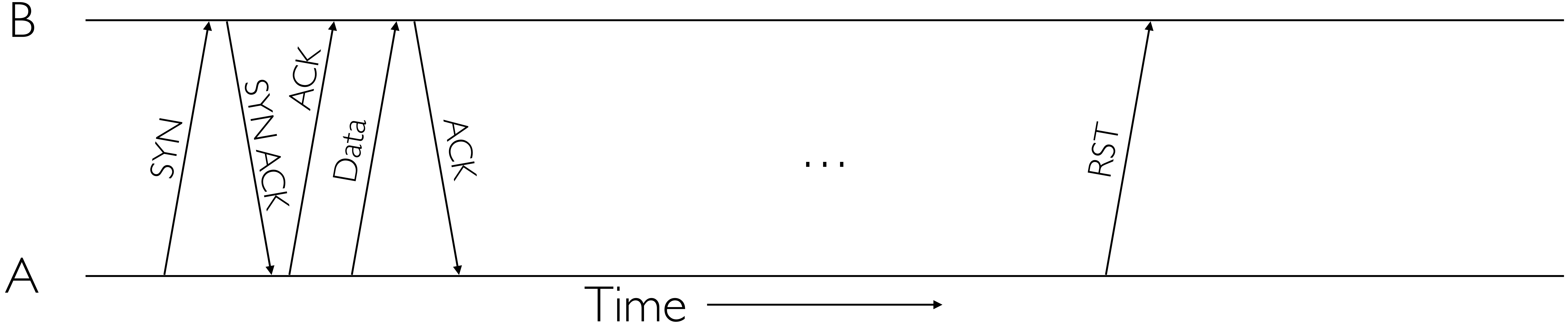
- A sends RESET (**RST**) to B
    - E.g., because application on process A crashed

Abrupt Termination



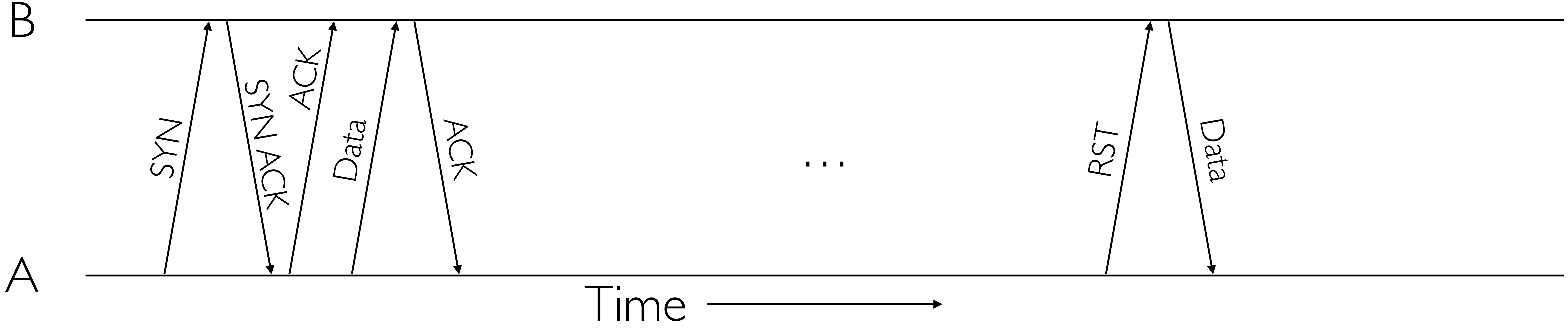
- A sends RESET (**RST**) to B
  - E.g., because application on process A crashed
- **That's it**
  - B does *not* ACK the RST, i.e., RST is not delivered reliably
  - And: any data in flight is lost

Abrupt Termination



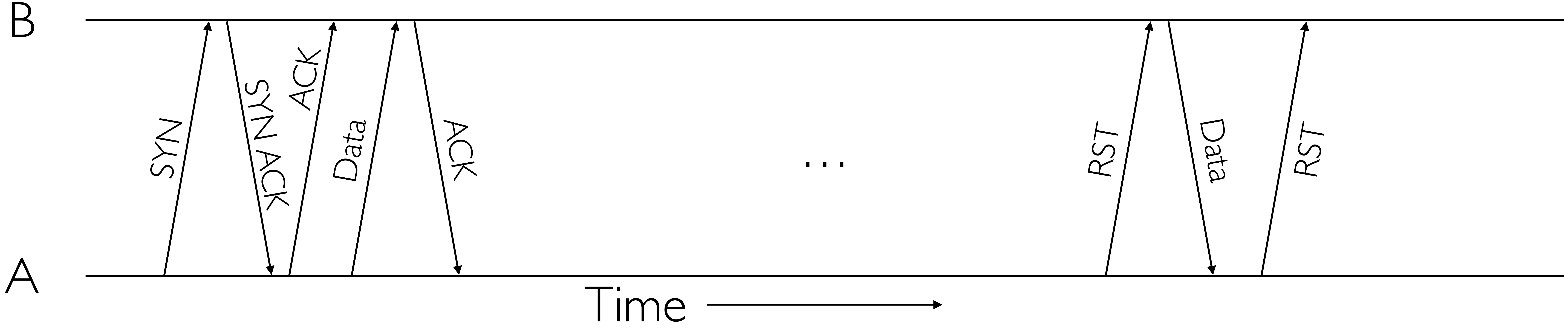
- A sends RESET (**RST**) to B
  - E.g., because application on process A crashed
- **That's it**
  - B does *not* ACK the RST, i.e., RST is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another RST **(So rude!)**

Abrupt Termination



- A sends RESET (**RST**) to B
  - E.g., because application on process A crashed
- **That's it**
  - B does *not* ACK the RST, i.e., RST is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another RST **(So rude!)**

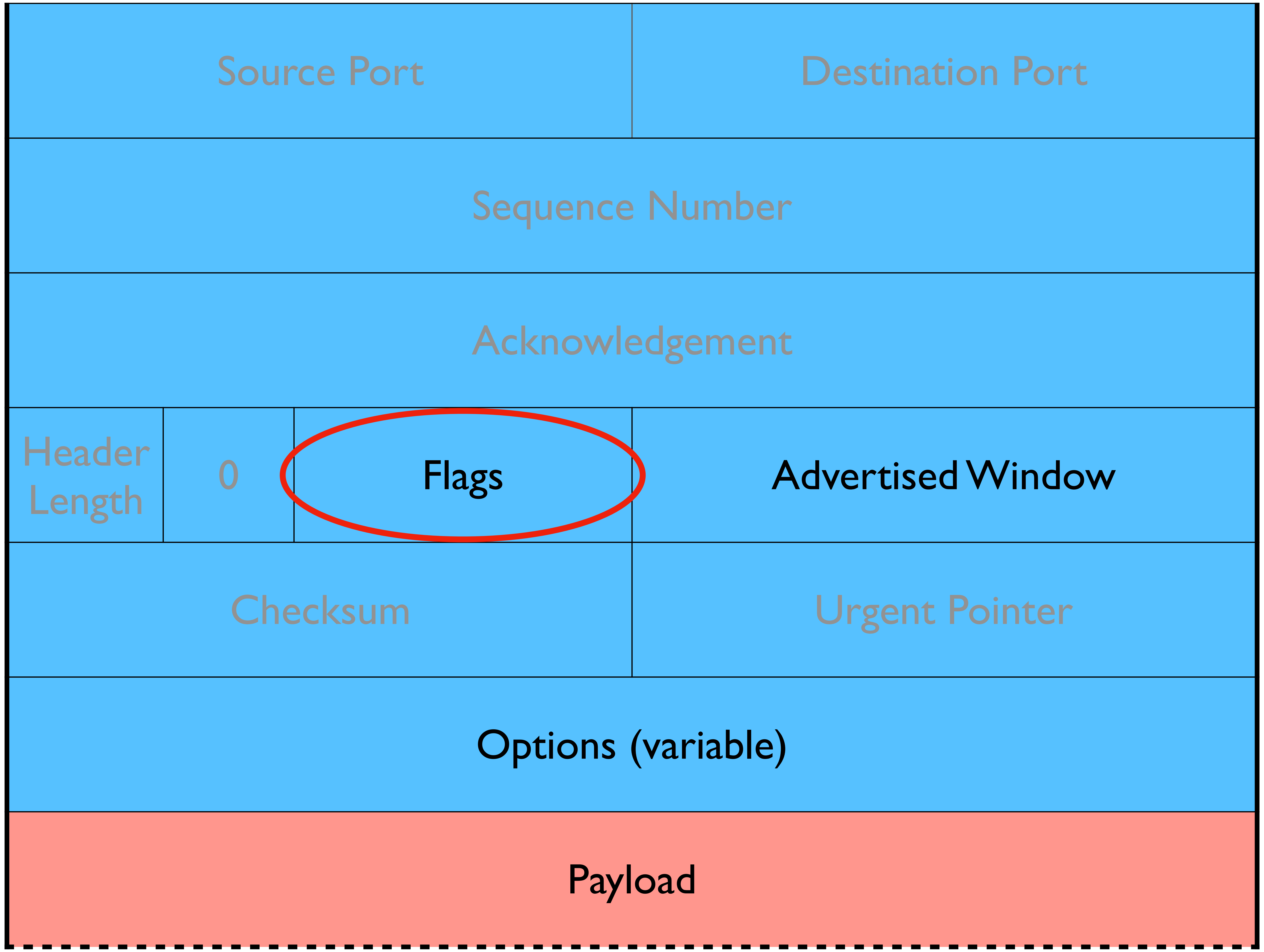
Abrupt Termination



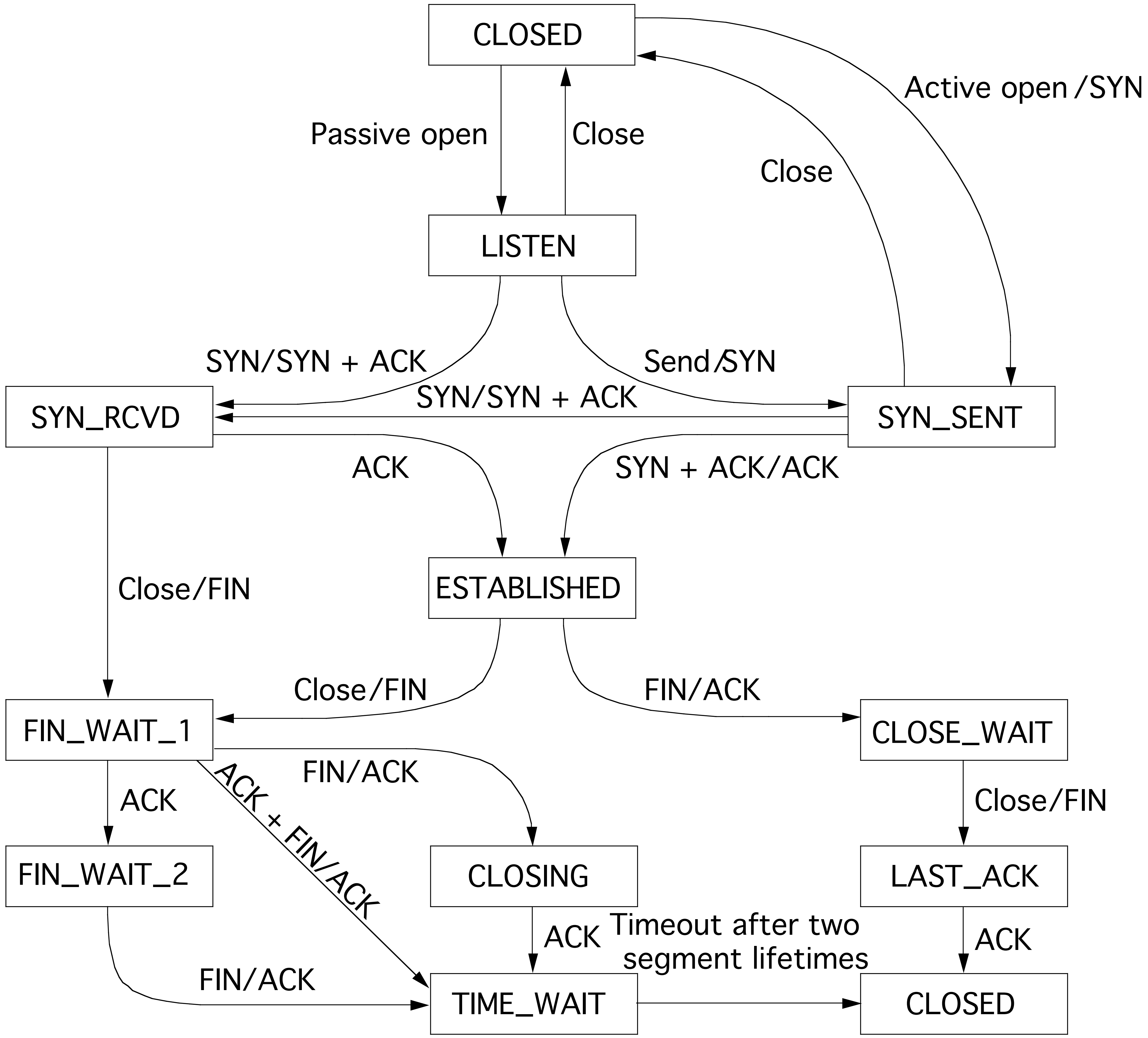
- A sends RESET (**RST**) to B
  - E.g., because application on process A crashed
- **That's it**
  - B does *not* ACK the RST, i.e., RST is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another RST **(So rude!)**

TCP Header

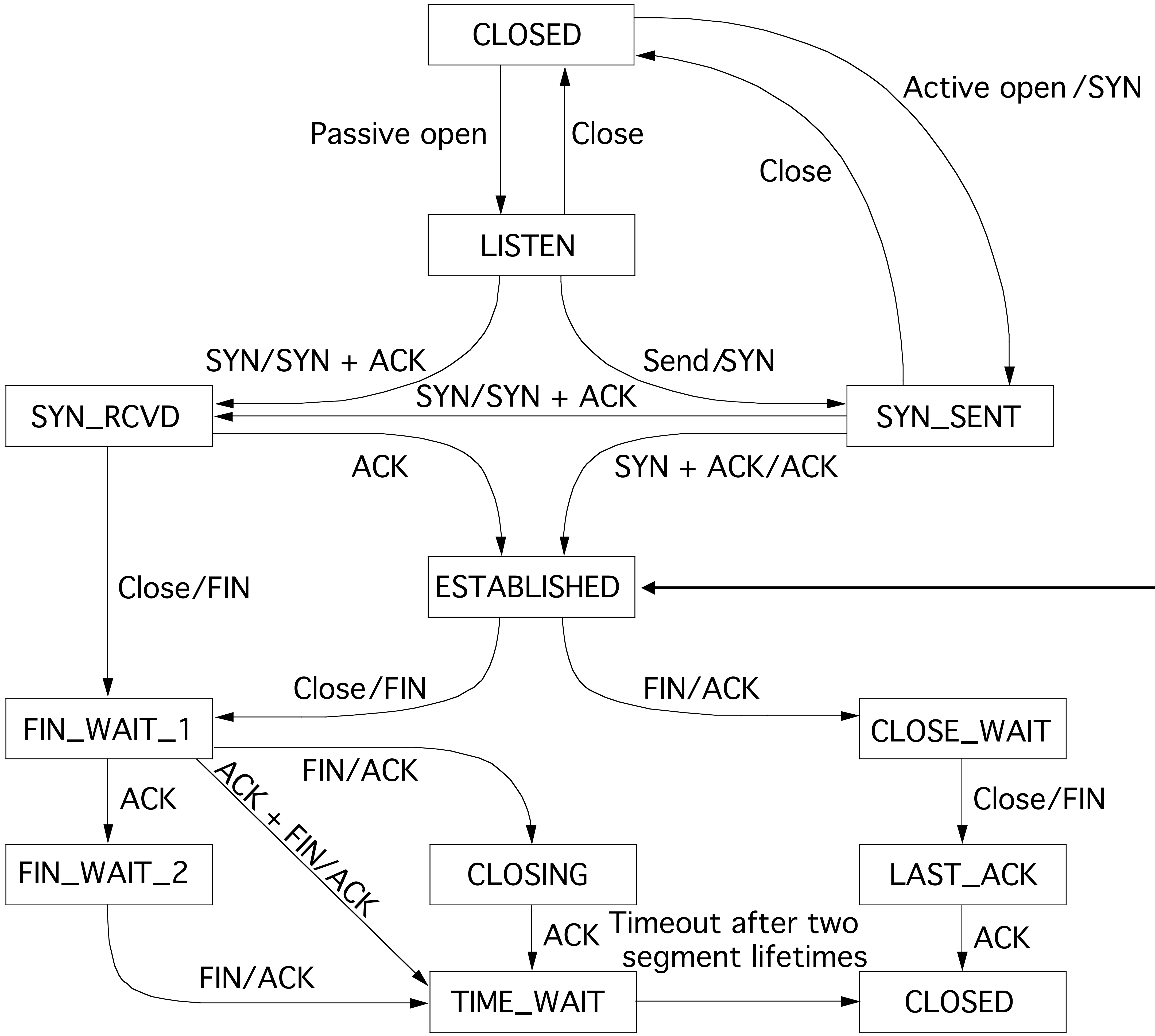
SYN
ACK
FIN
RST
PSH
URG



TCP State Transitions

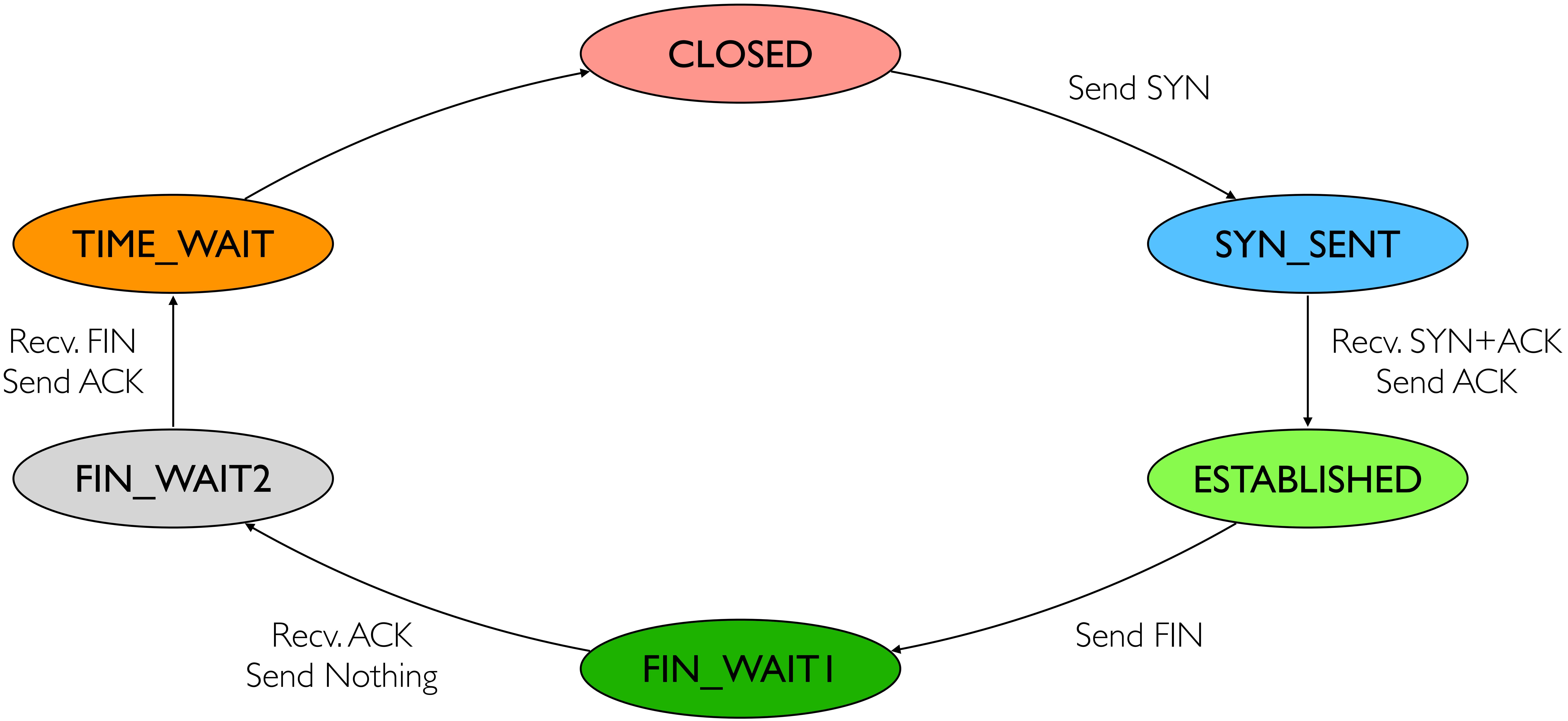


TCP State Transitions



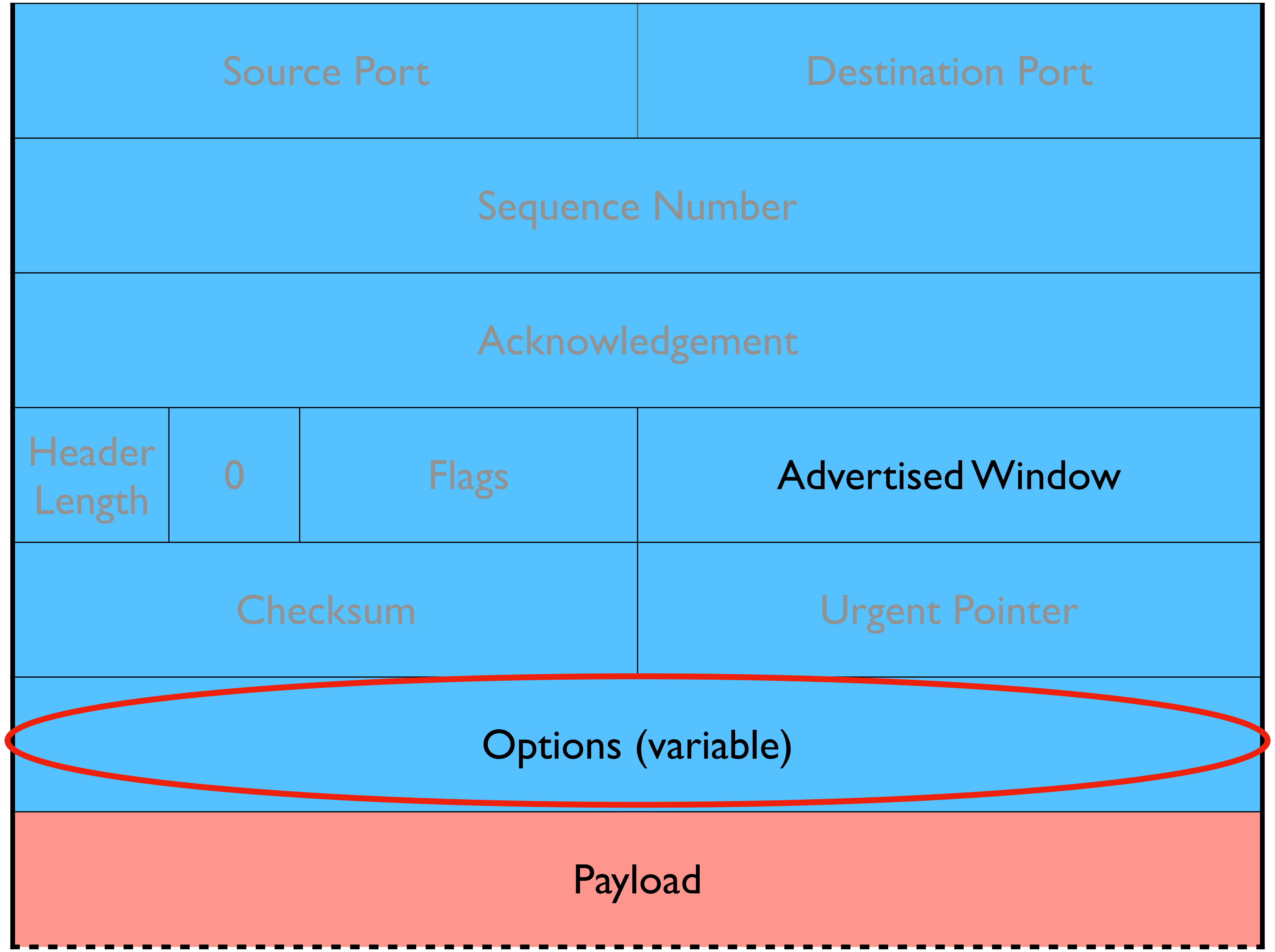
Data, ACK exchanges are in here

A Simpler View on the Client Side

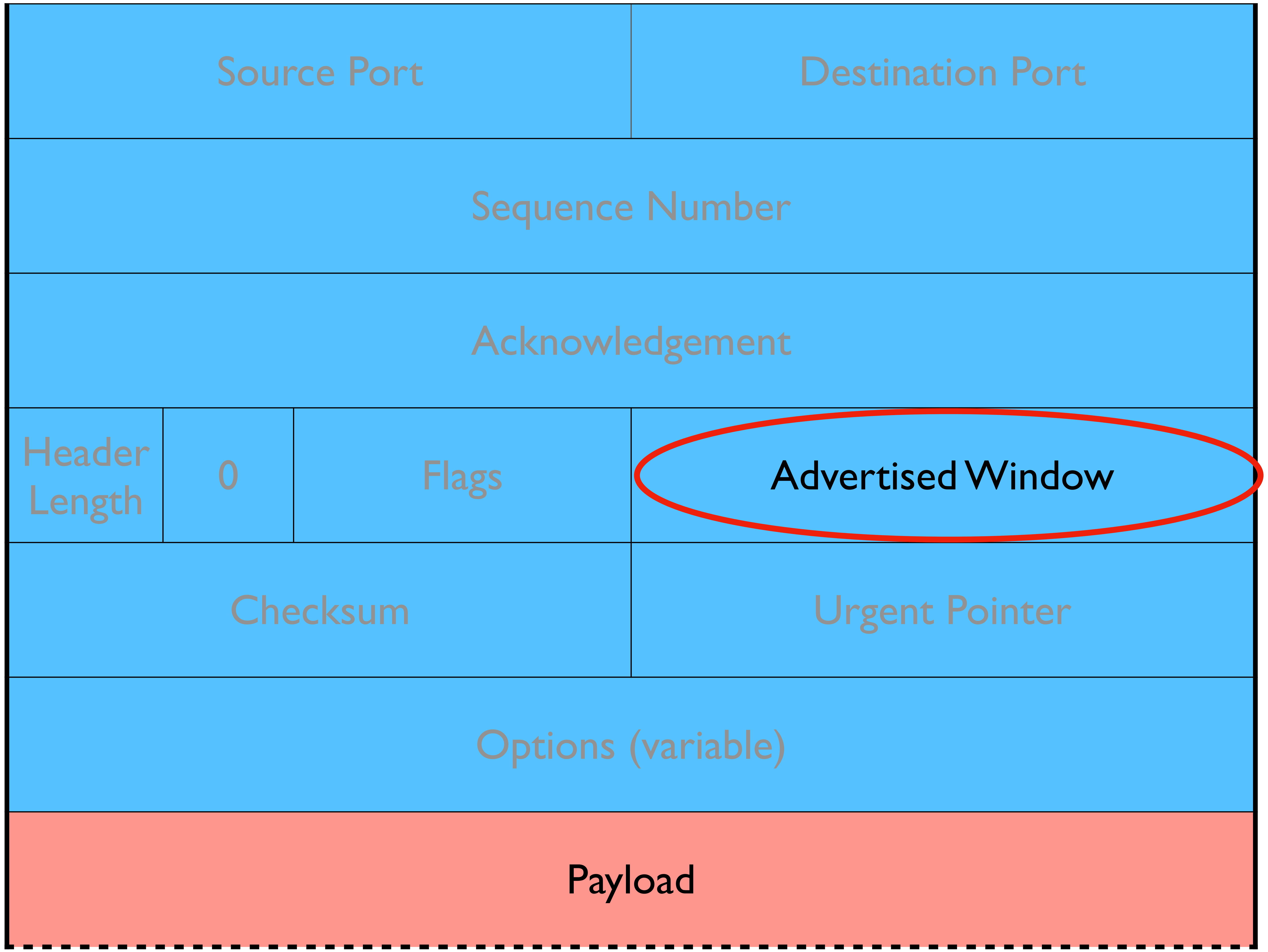


TCP Header

Used to negotiate use of additional features



TCP Header


Source Port
Destination Port
Sequence Number
Acknowledgement
Header Length
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Payload

Last Time: Sliding Window

Last Time: Sliding Window

- **Both sender & receiver maintain a <u>window</u>**

Last Time: Sliding Window

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***expected*** data
    - First "hole" in received data
    - When sender gets ACK, knows that the received's window has moved

Last Time: Sliding Window

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***expected*** data
    - First "hole" in received data
    - When sender gets ACK, knows that the received's window has moved
- **Right edge: left edge + constant**
    - Constant only limited by buffer size in the transport layer

TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***undelivered*** data
- **Right edge: left edge + constant**
  - Constant only limited by buffer size in the transport layer

TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***undelivered*** data
- **Right edge: left edge + constant**
  - Constant only limited by buffer size in the transport layer

TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***undelivered*** data
- **Right edge: left edge + constant**
  - Constant only limited by buffer size in the transport layer

Where in the transport layer?

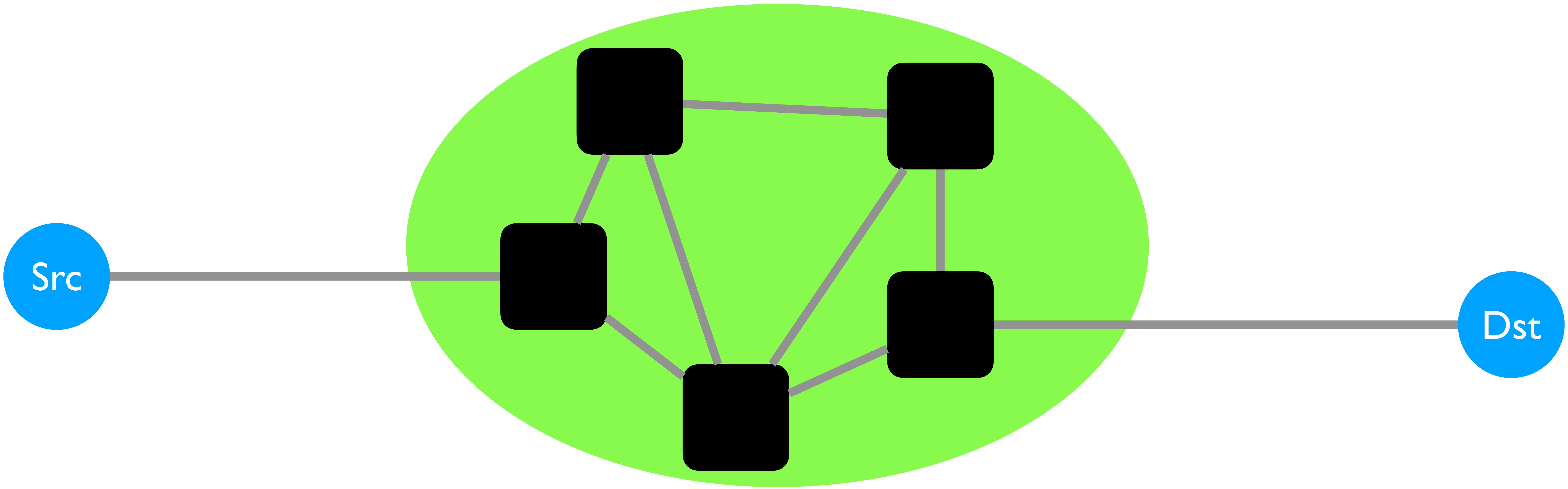
TCP: Sliding Window (so far)

- **Both sender & receiver maintain a <u>window</u>**
- **Left edge of window:**
  - Sender: beginning of ***unacknowledged*** data
  - Receiver: beginning of ***undelivered*** data
- **Right edge: left edge + constant**
  - Constant only limited by buffer size in the transport layer

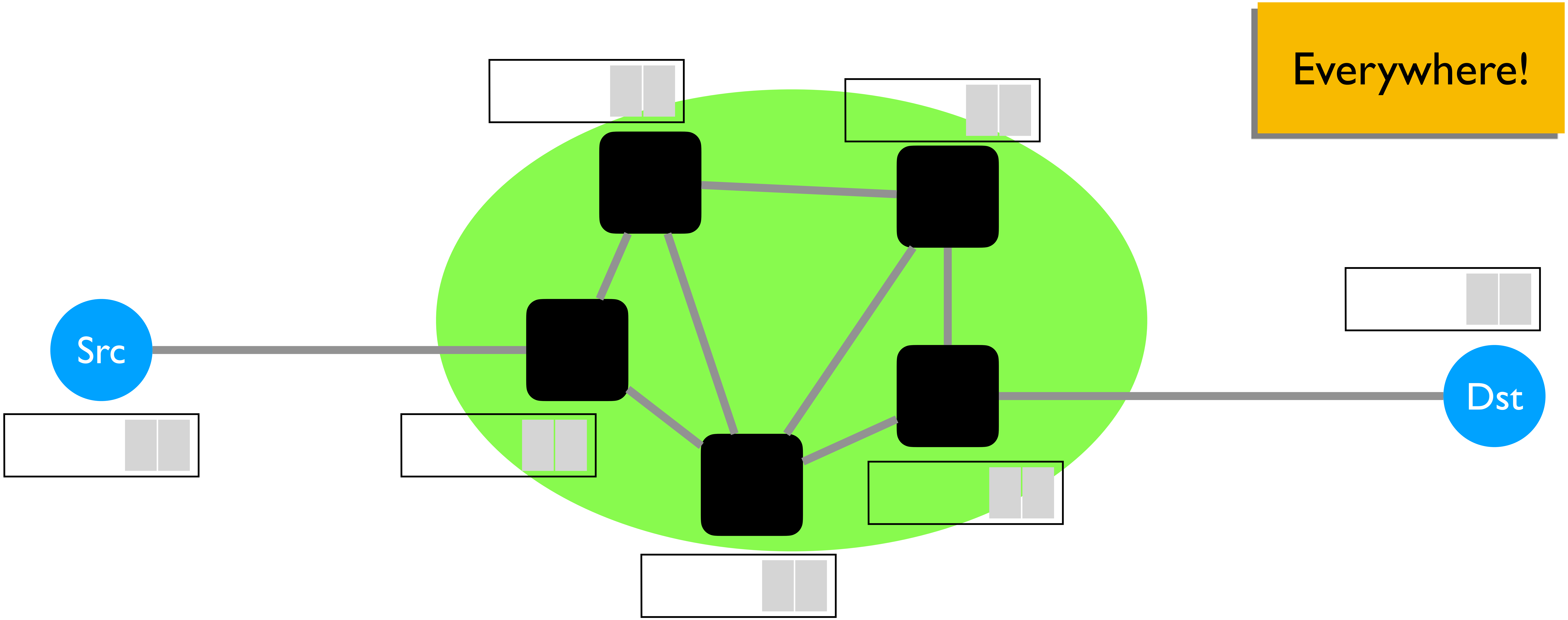
Where in the transport layer?

Only the transport layer?

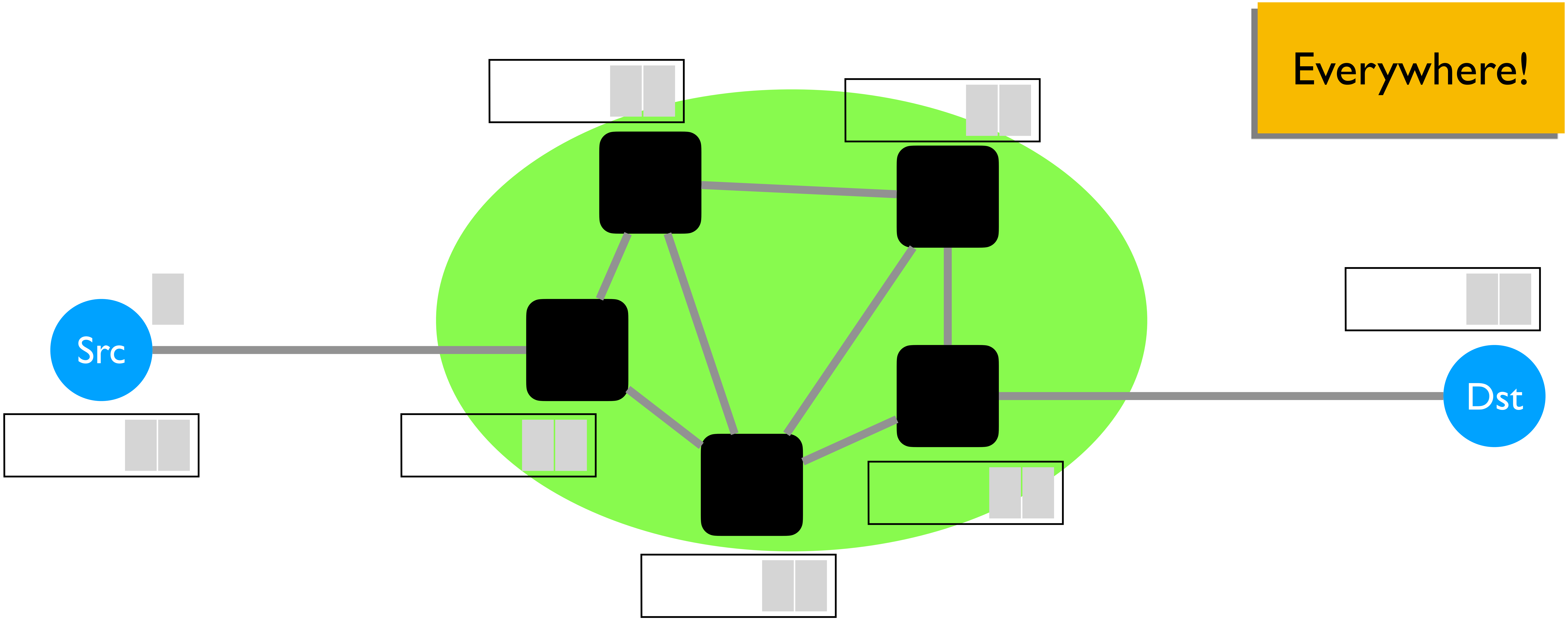
Where is the buffer?



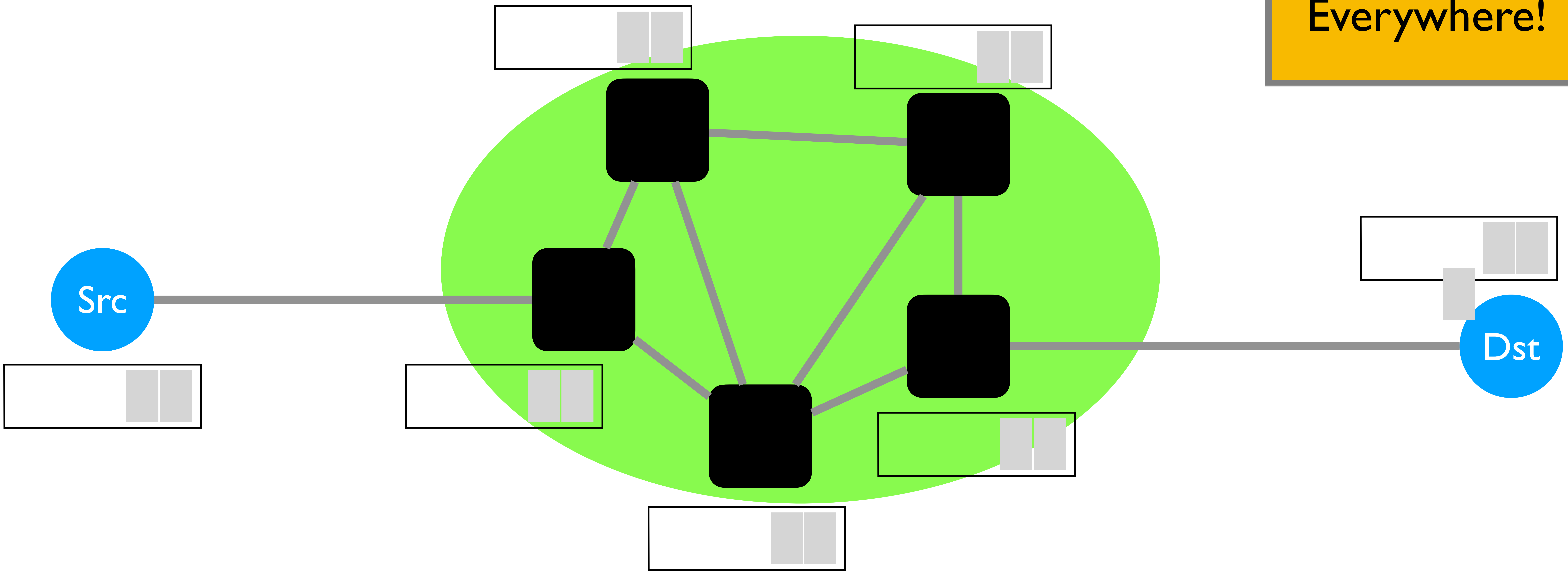
Where is the buffer?



Where is the buffer?

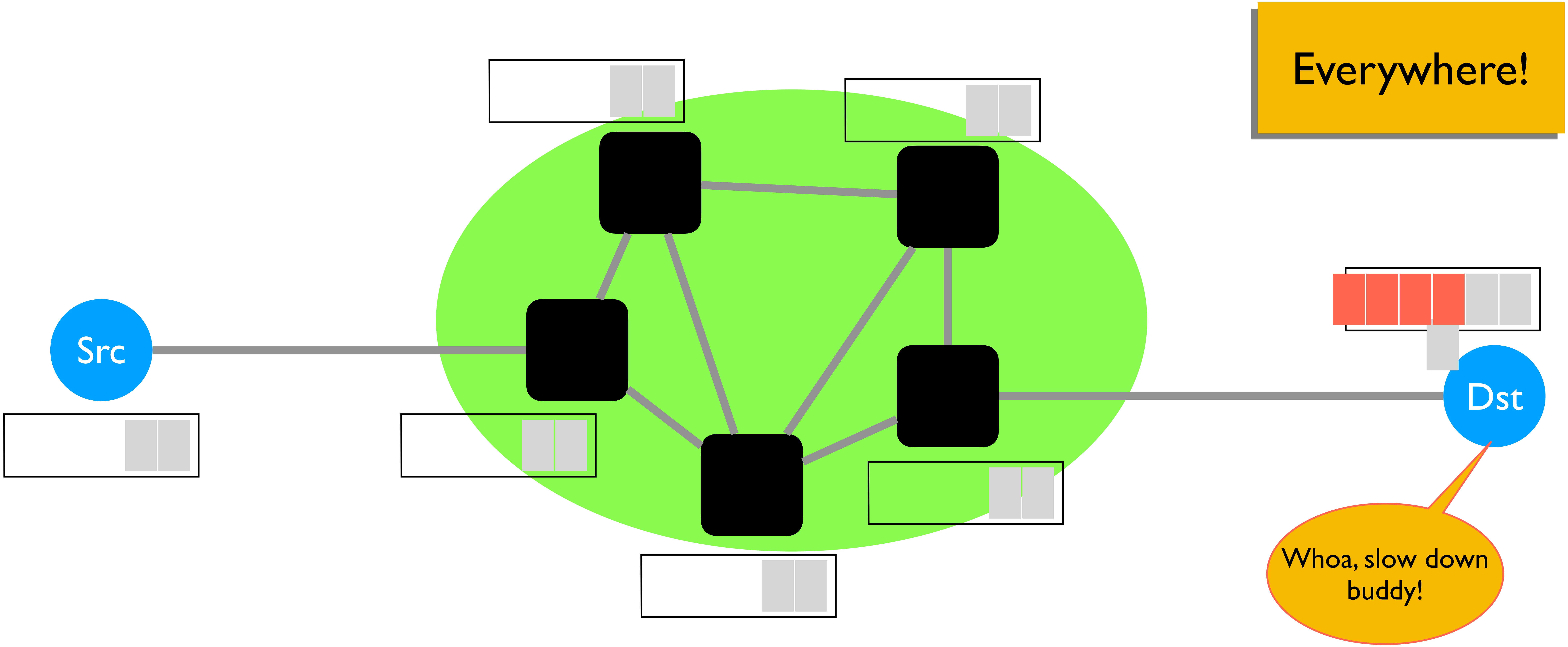


Where is the buffer?

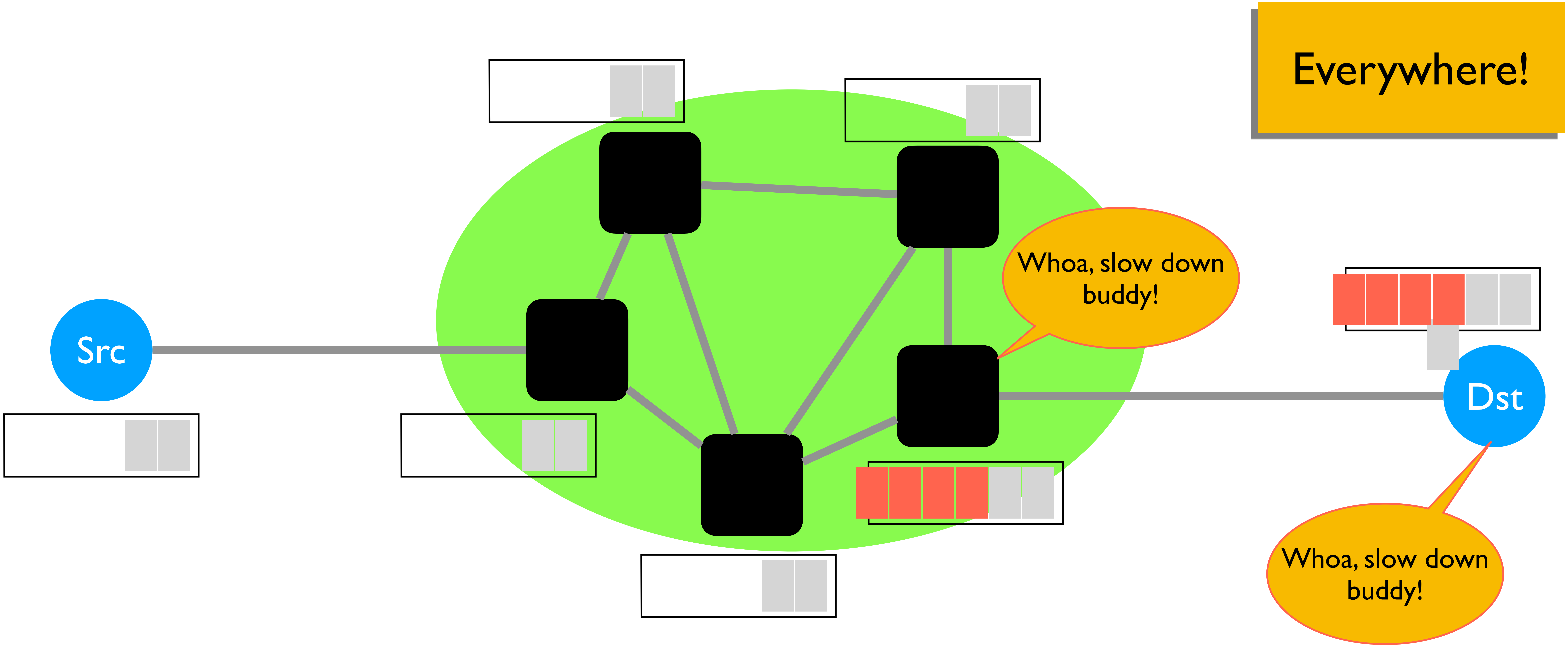


Everywhere!

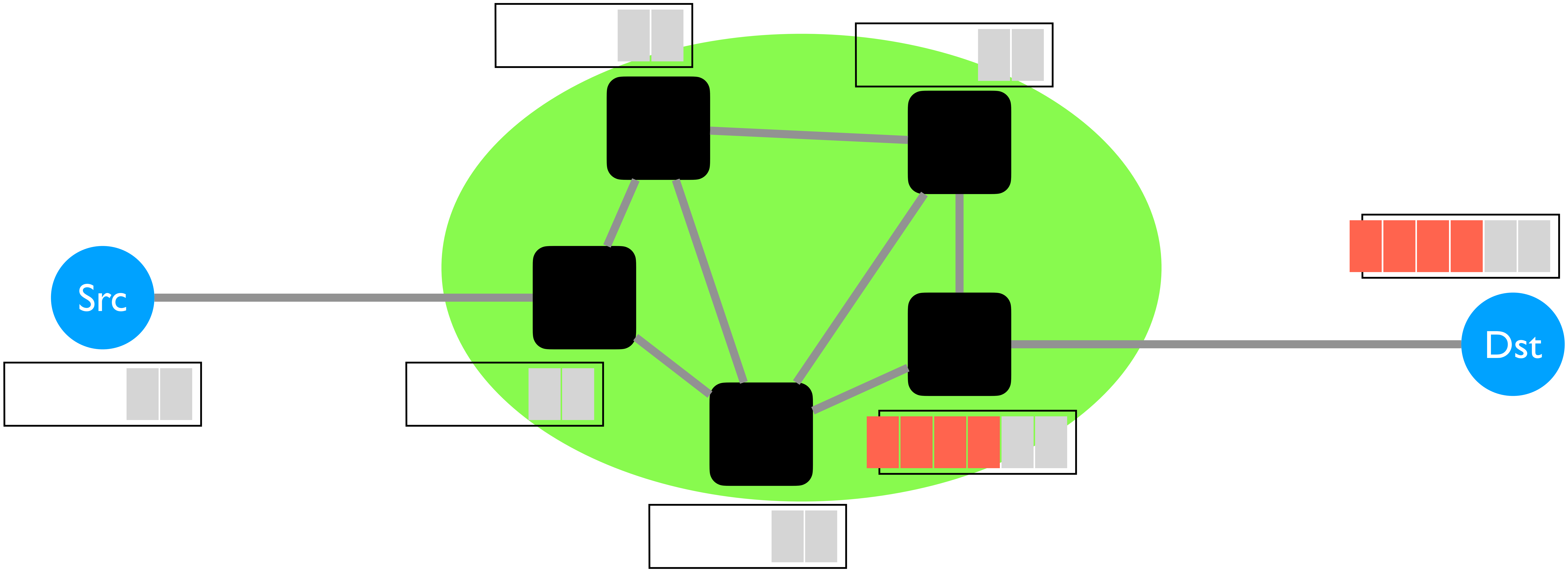
Where is the buffer?



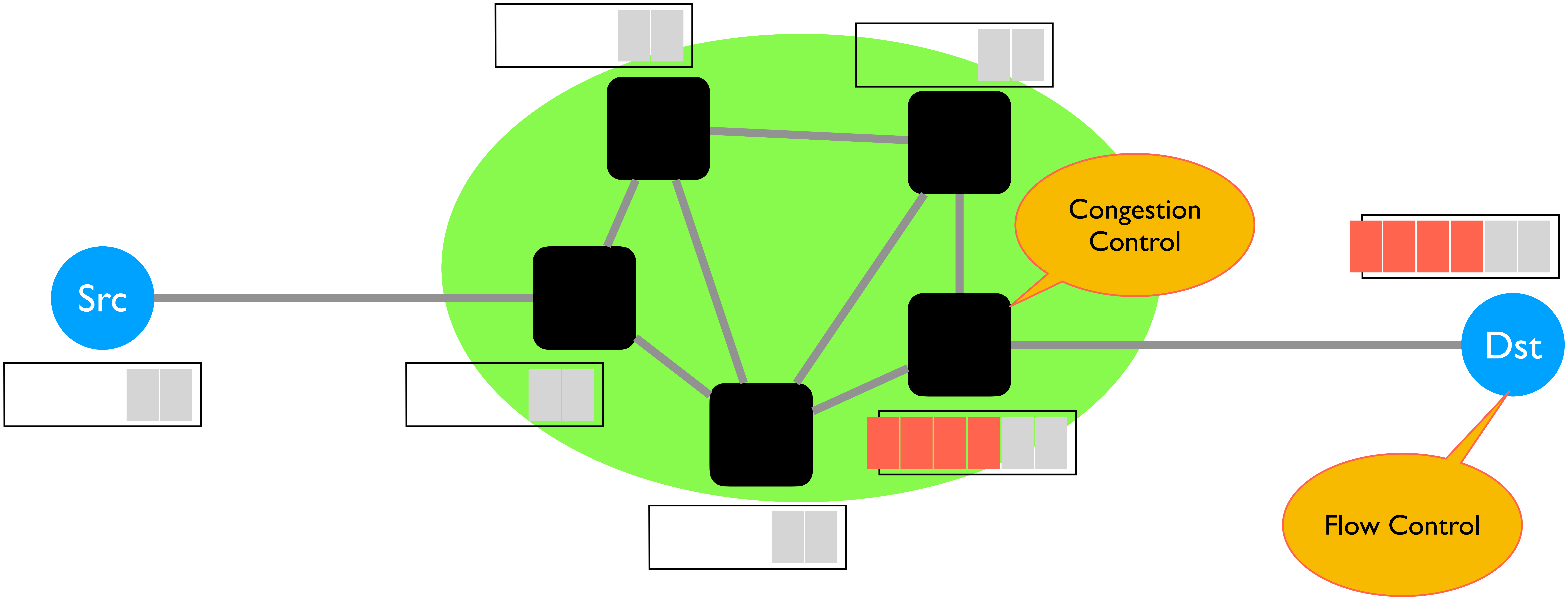
Where is the buffer?



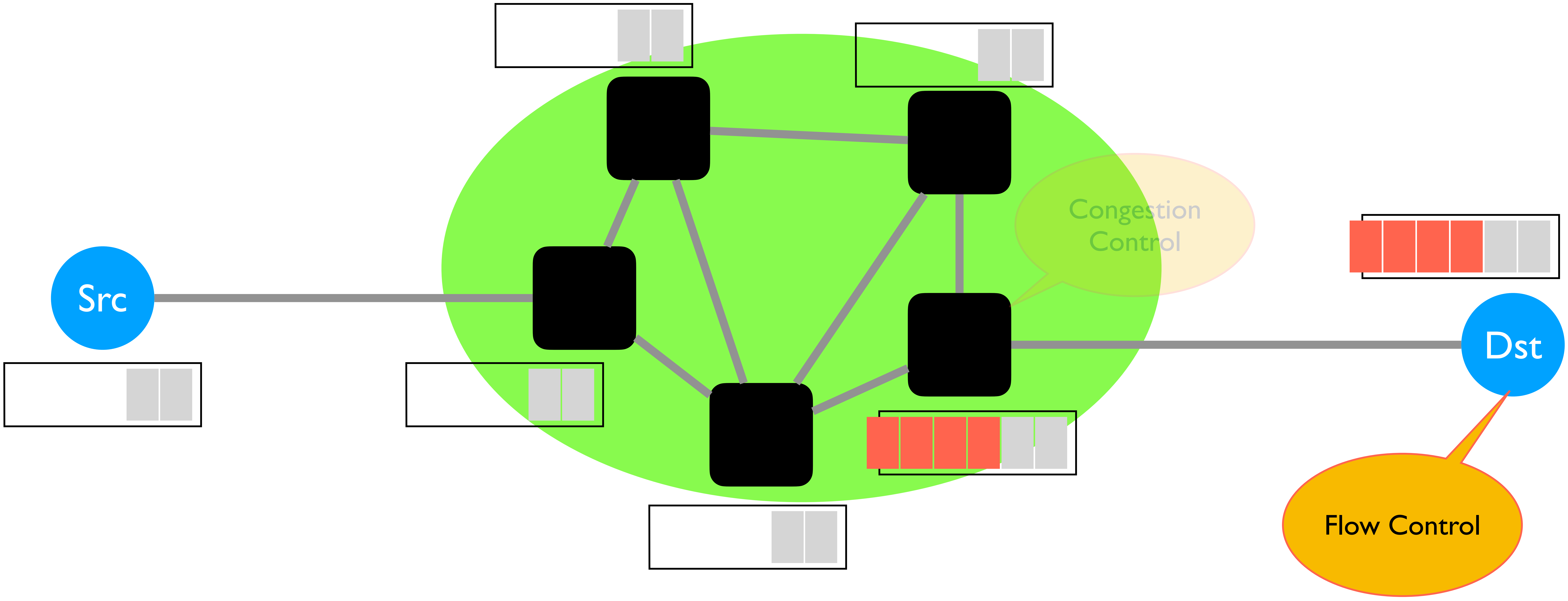
How does TCP deal with buffer limits?



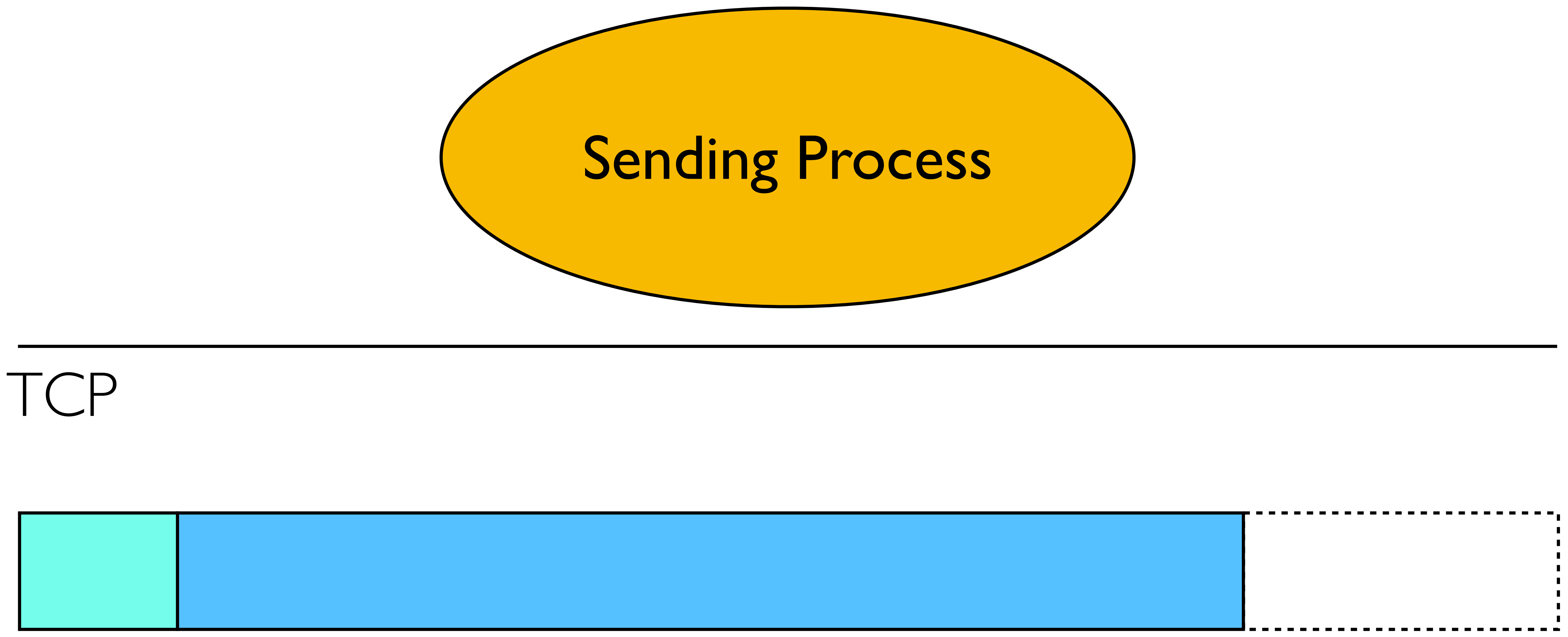
How does TCP deal with buffer limits?



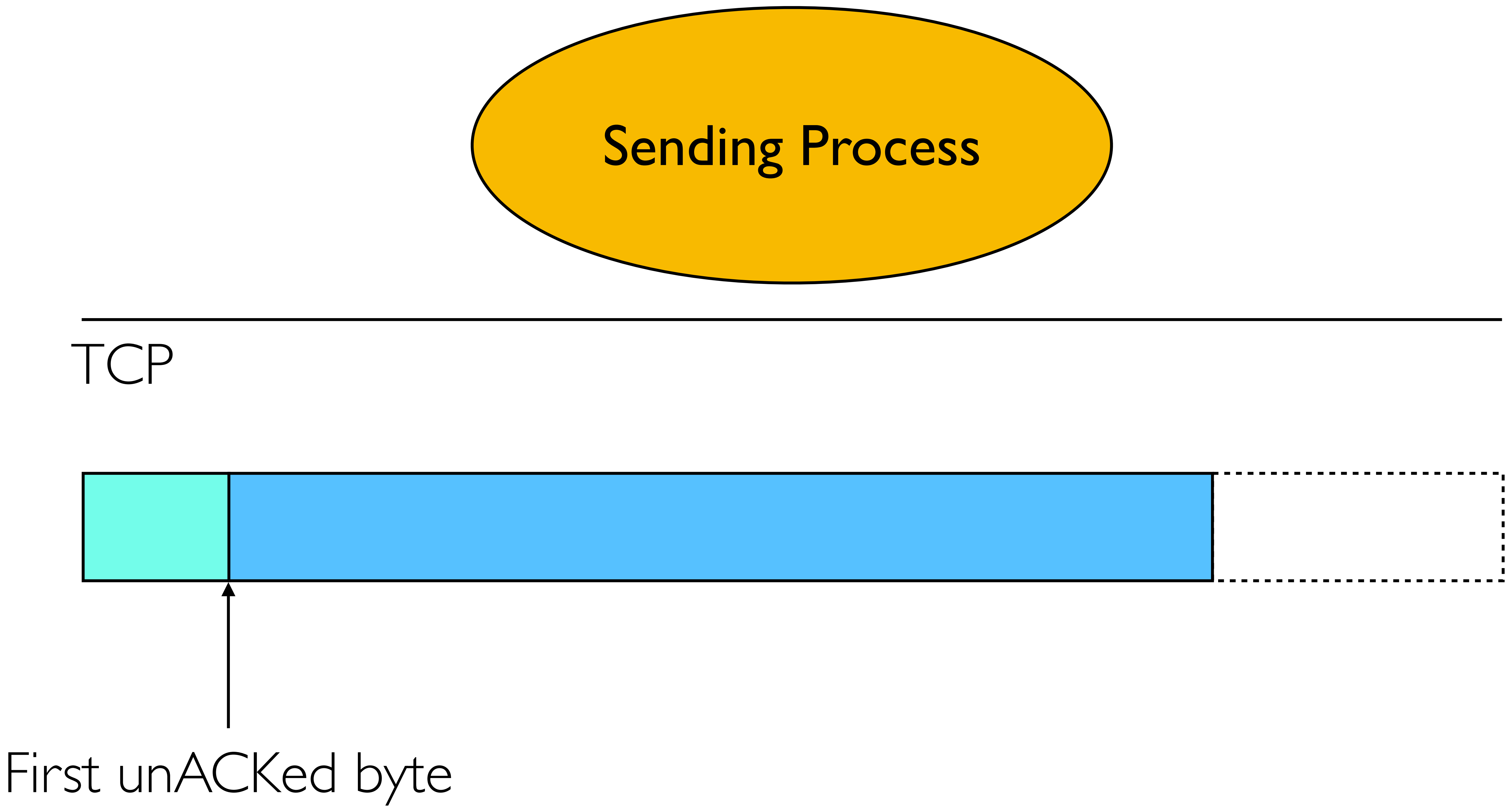
How does TCP deal with buffer limits?



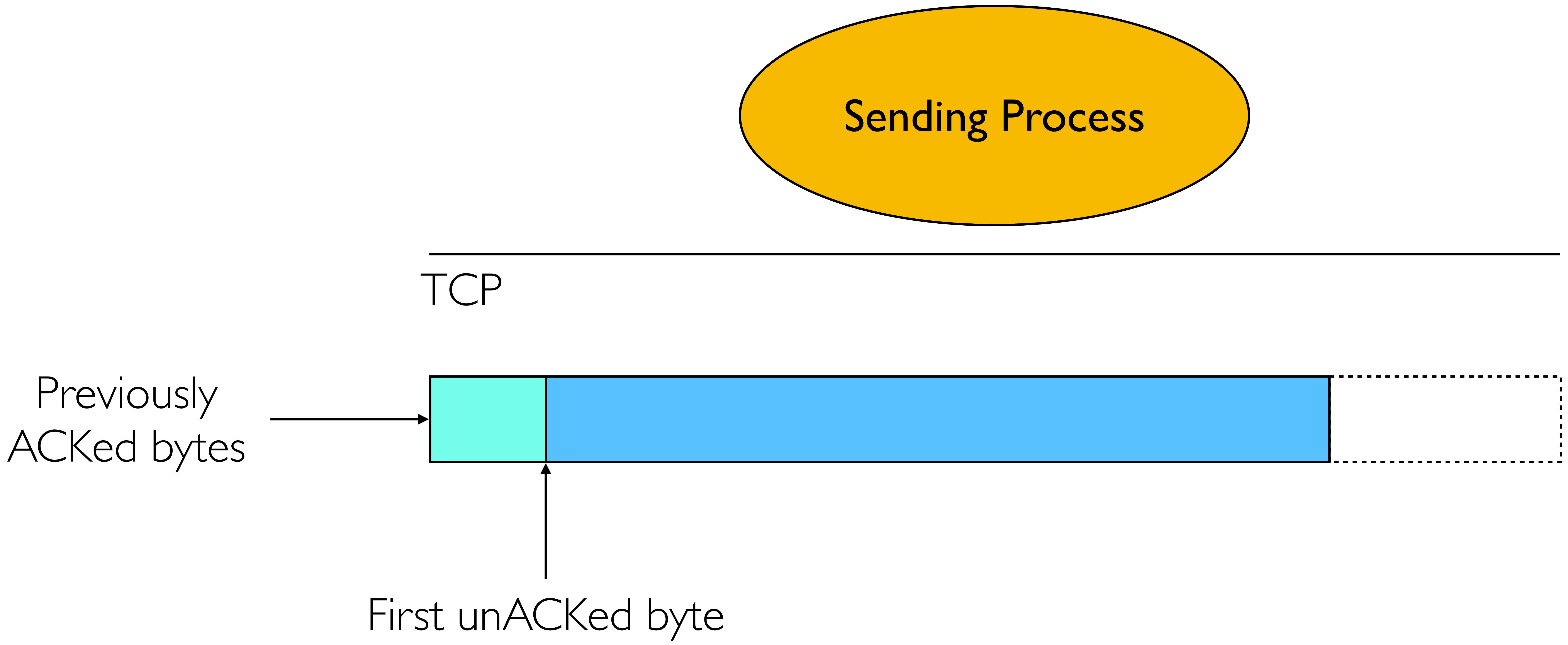
Sliding Window at Sender



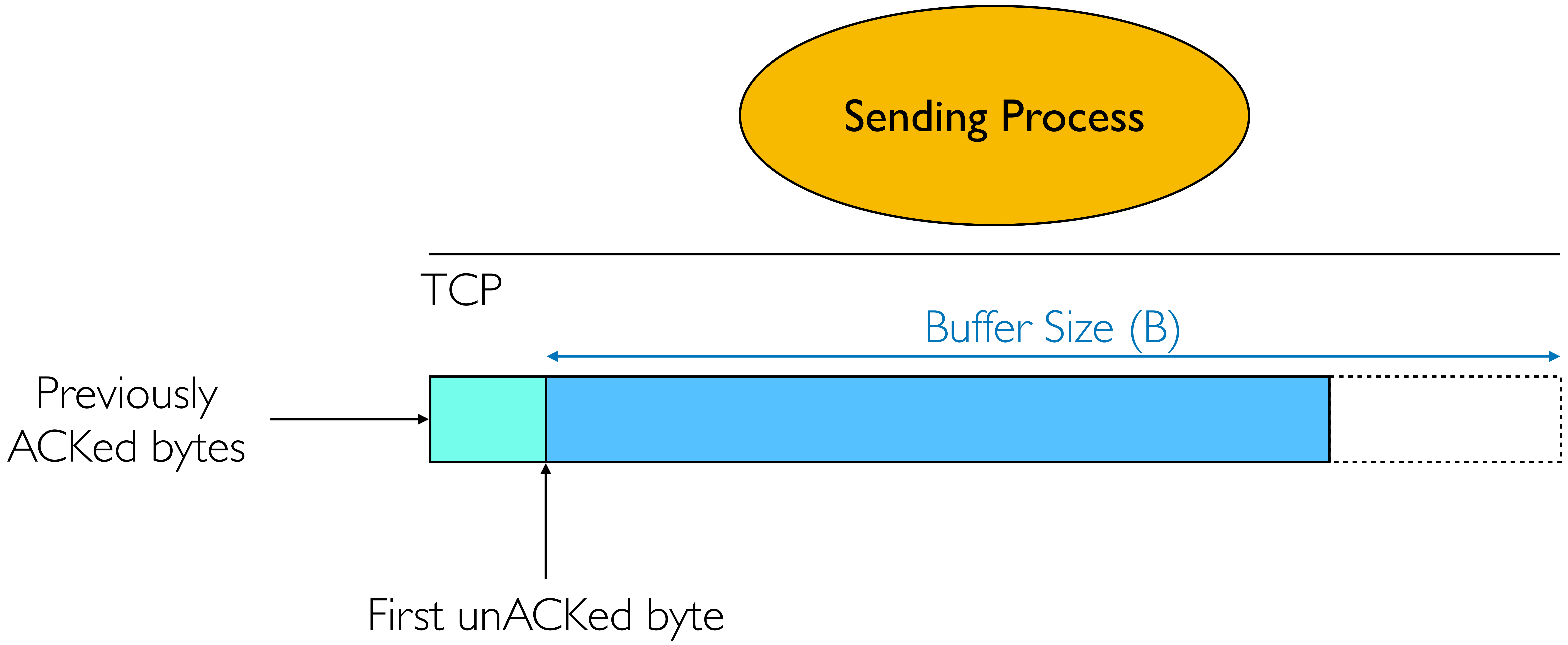
Sliding Window at Sender



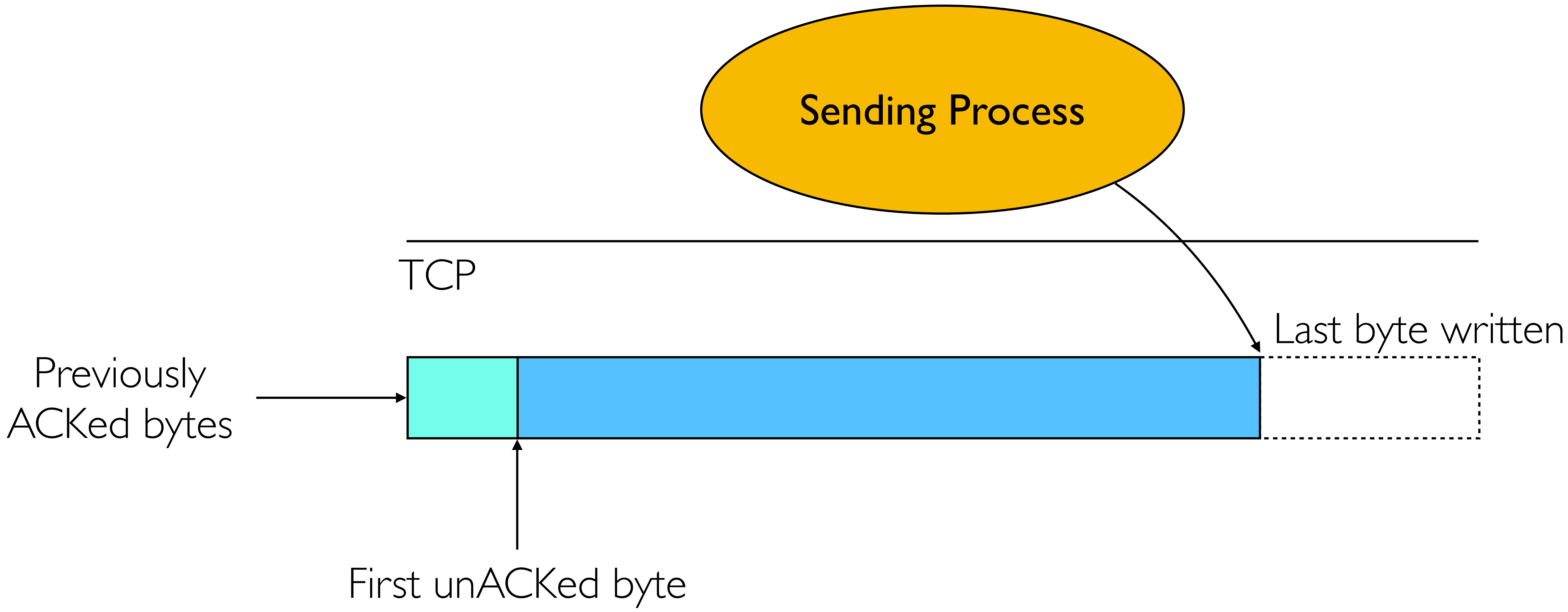
Sliding Window at Sender



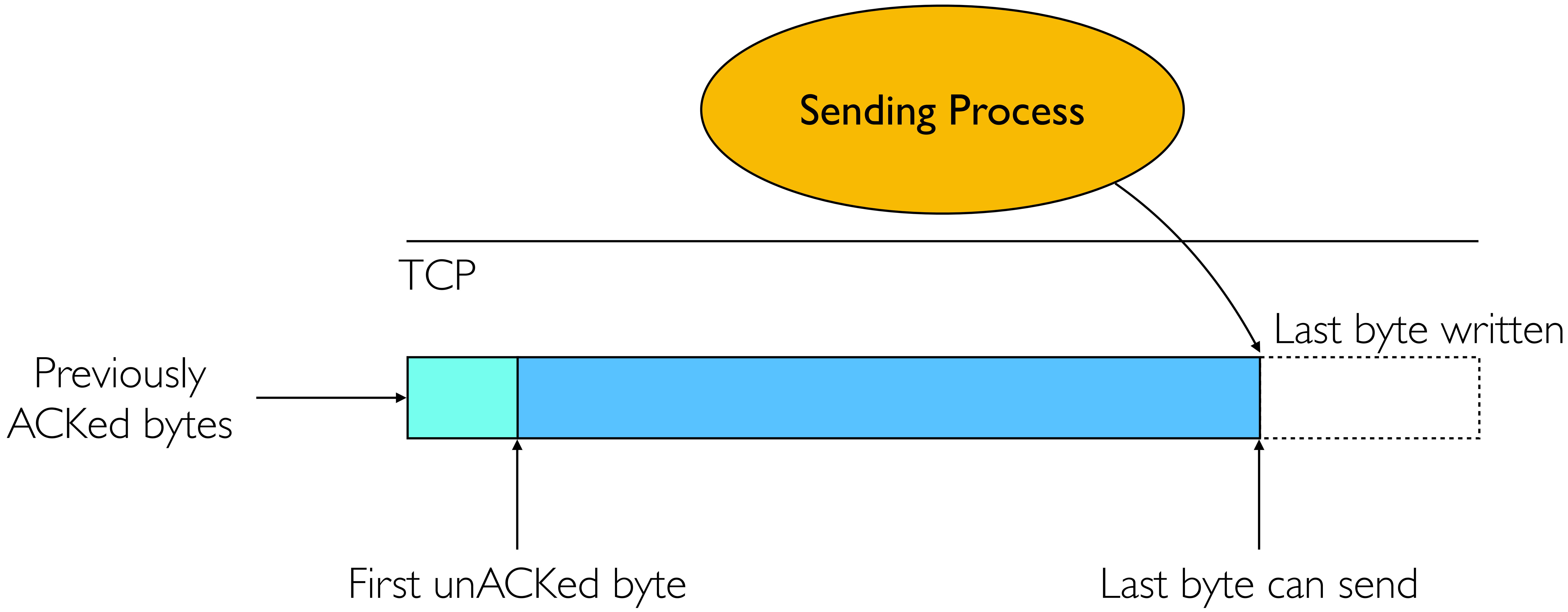
Sliding Window at Sender



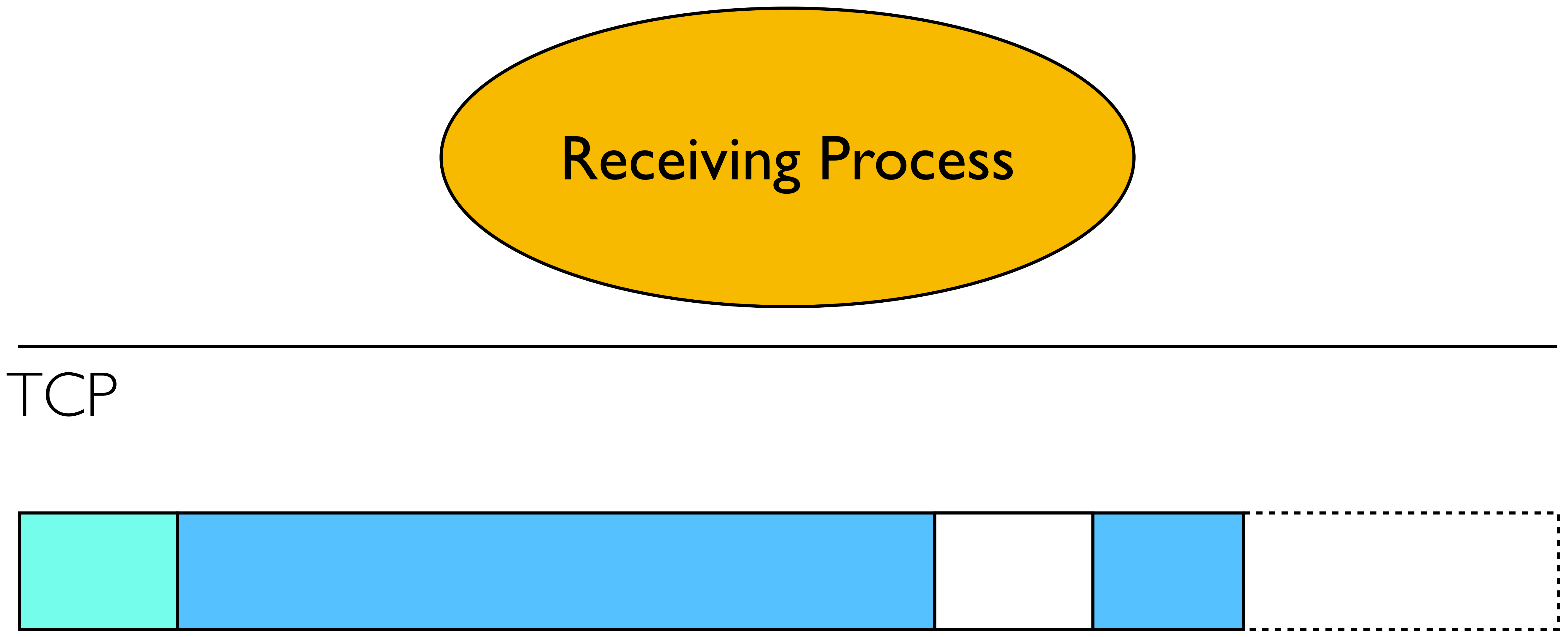
Sliding Window at Sender



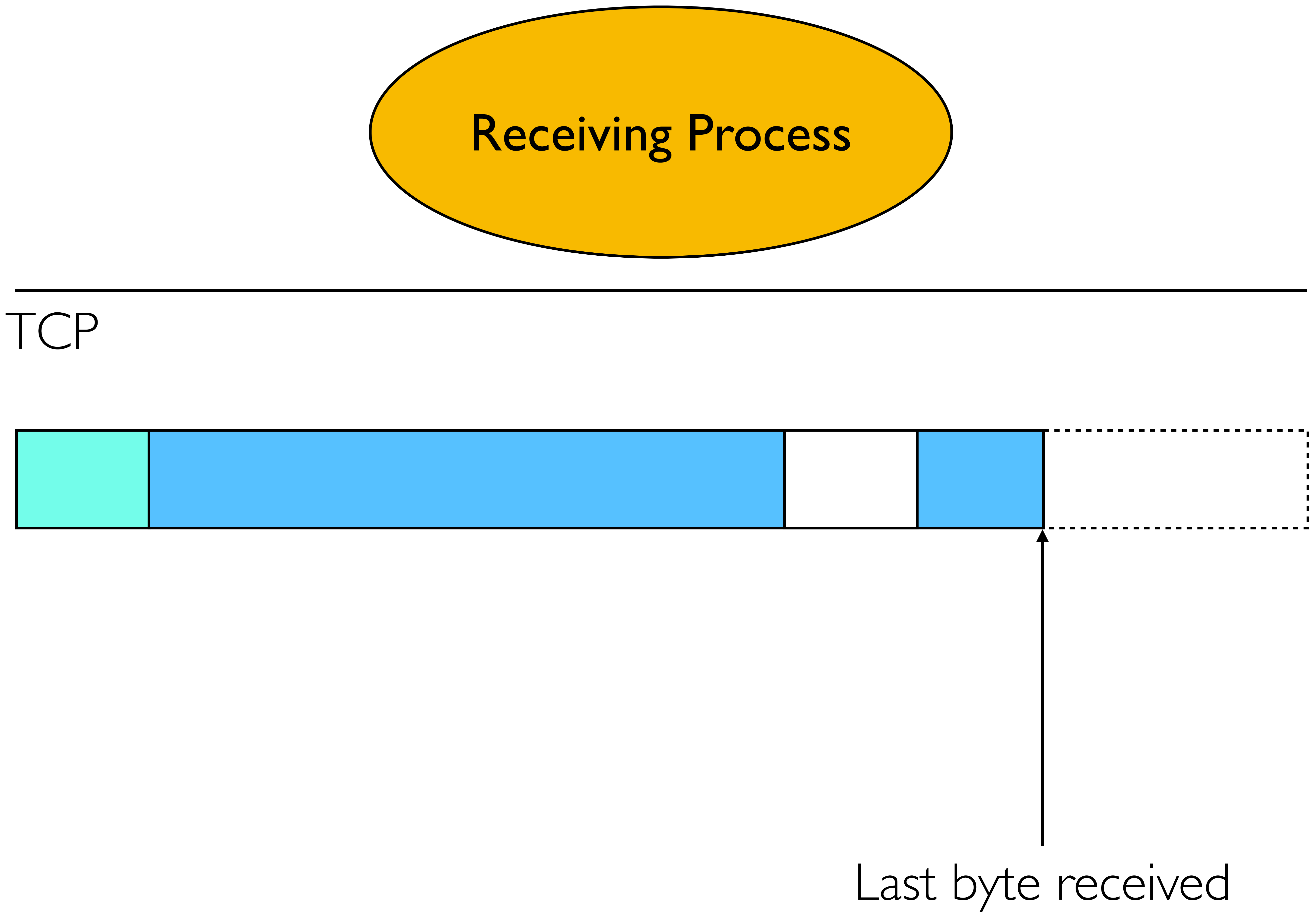
Sliding Window at Sender



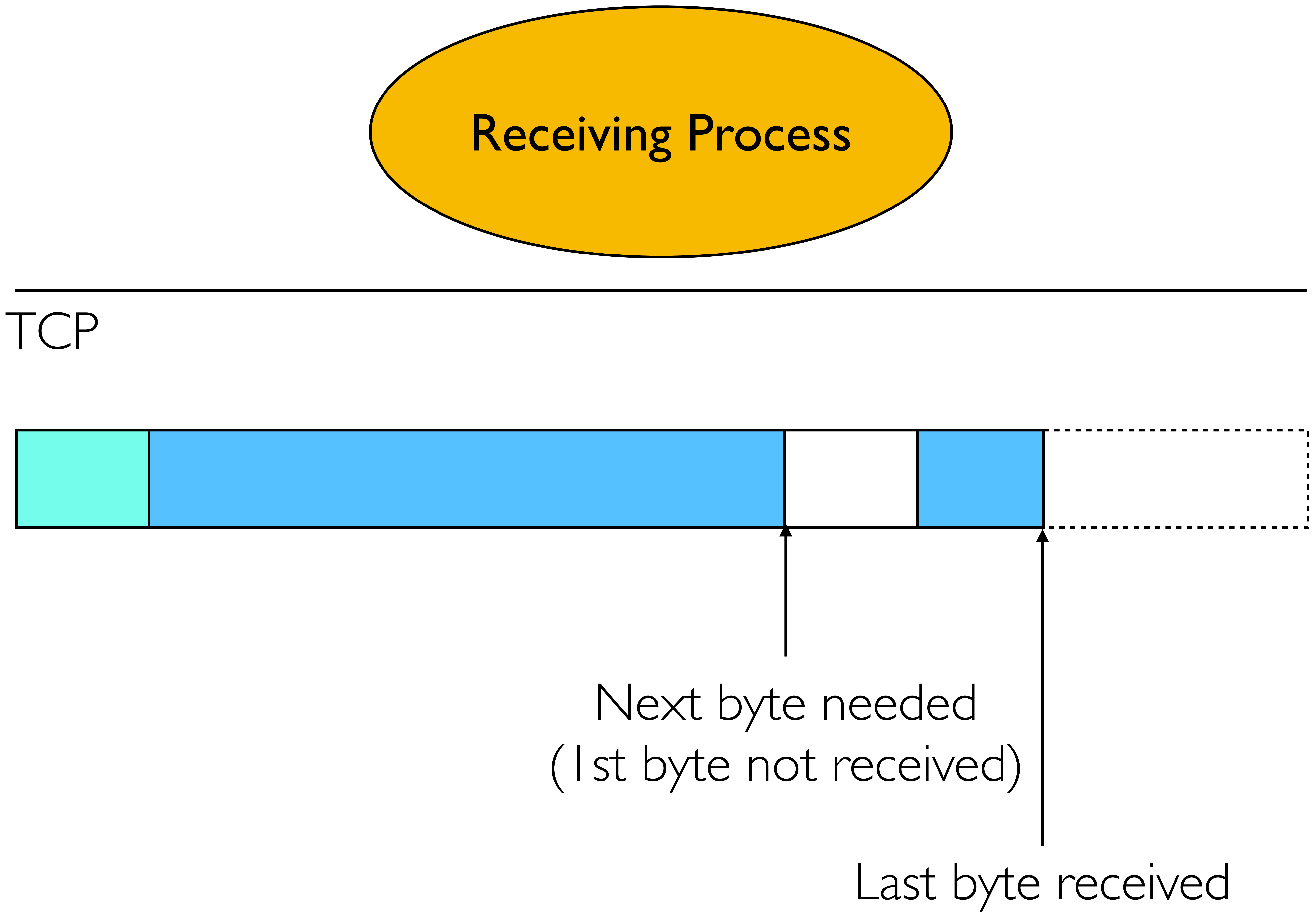
Sliding Window at Receiver



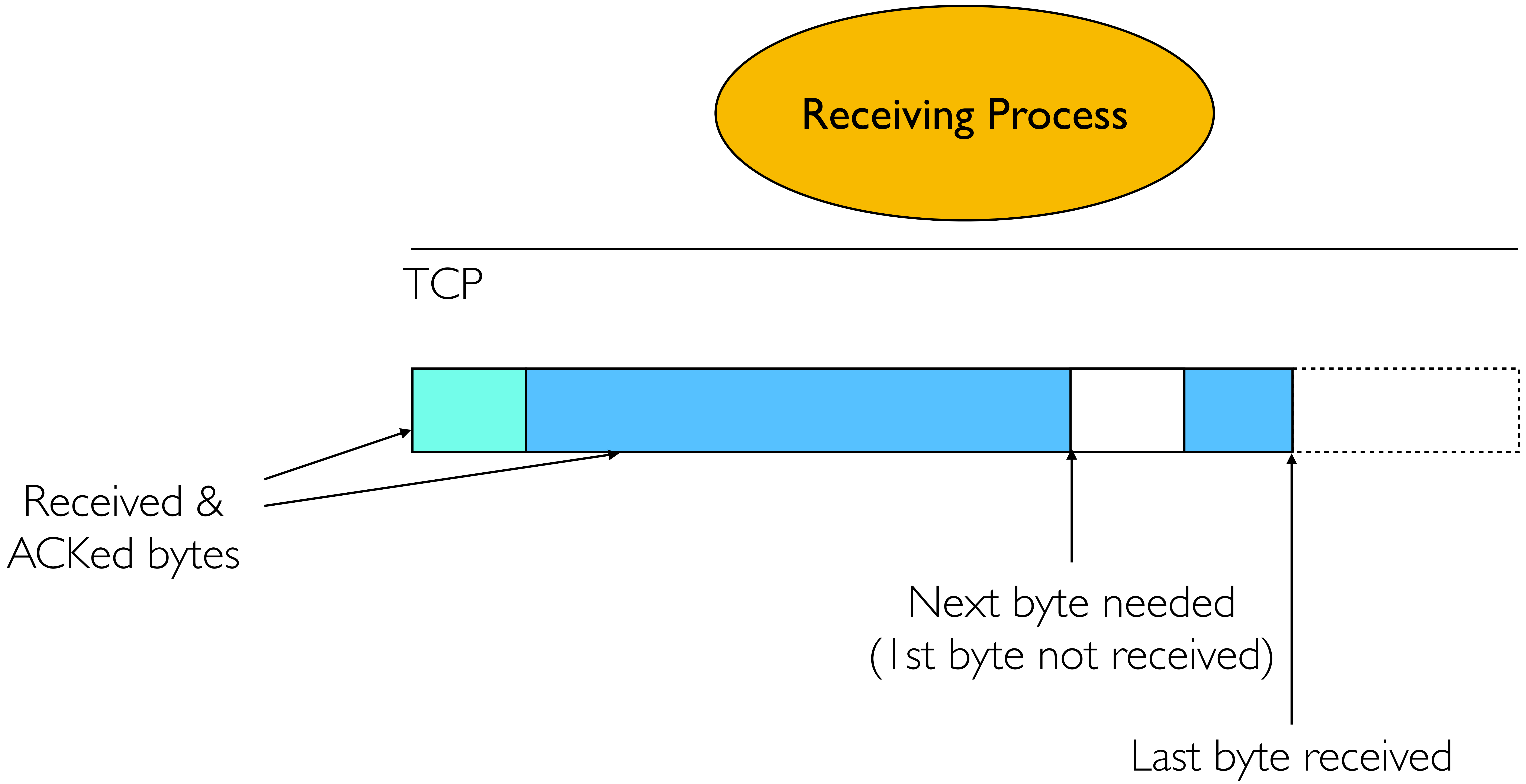
Sliding Window at Receiver



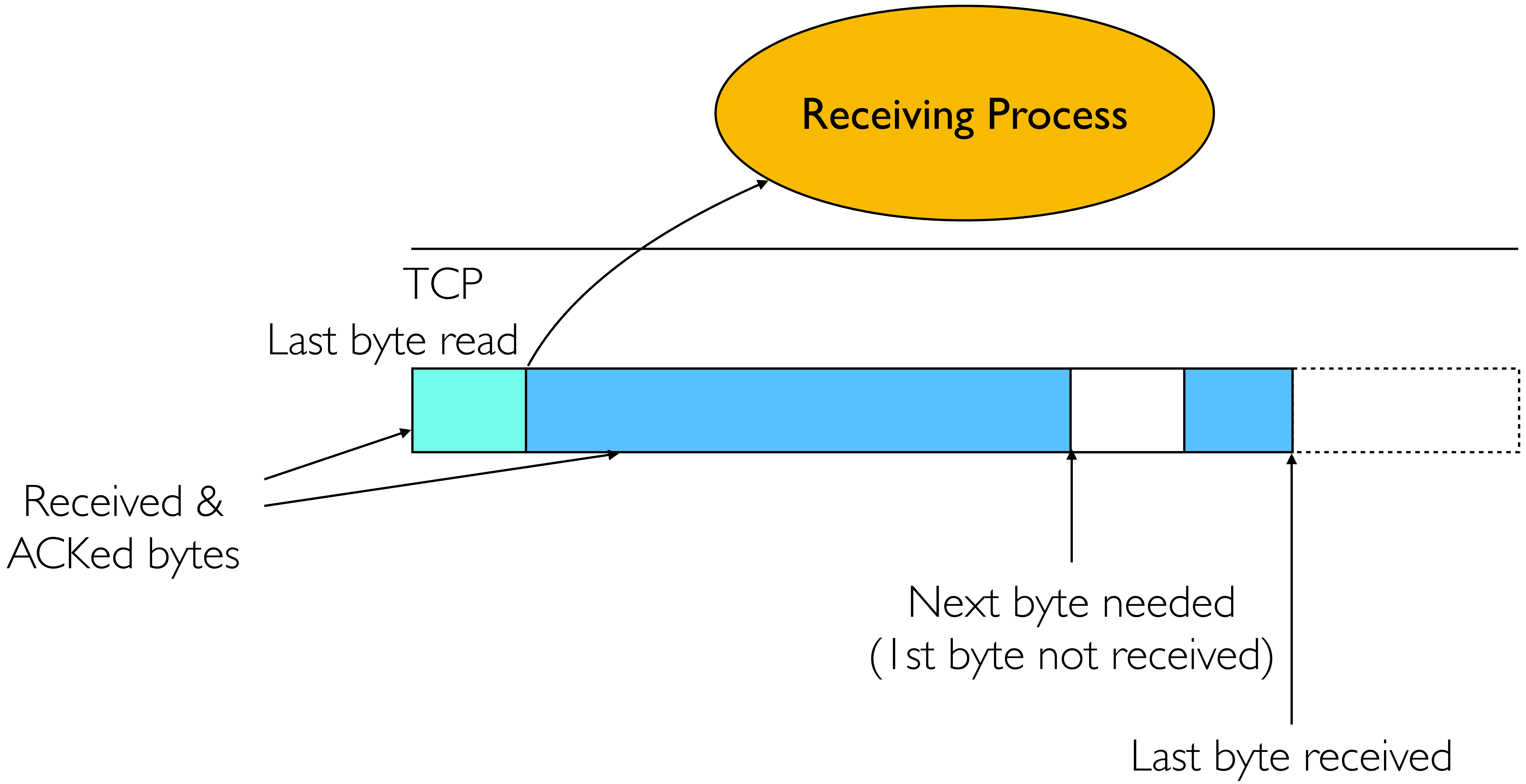
Sliding Window at Receiver



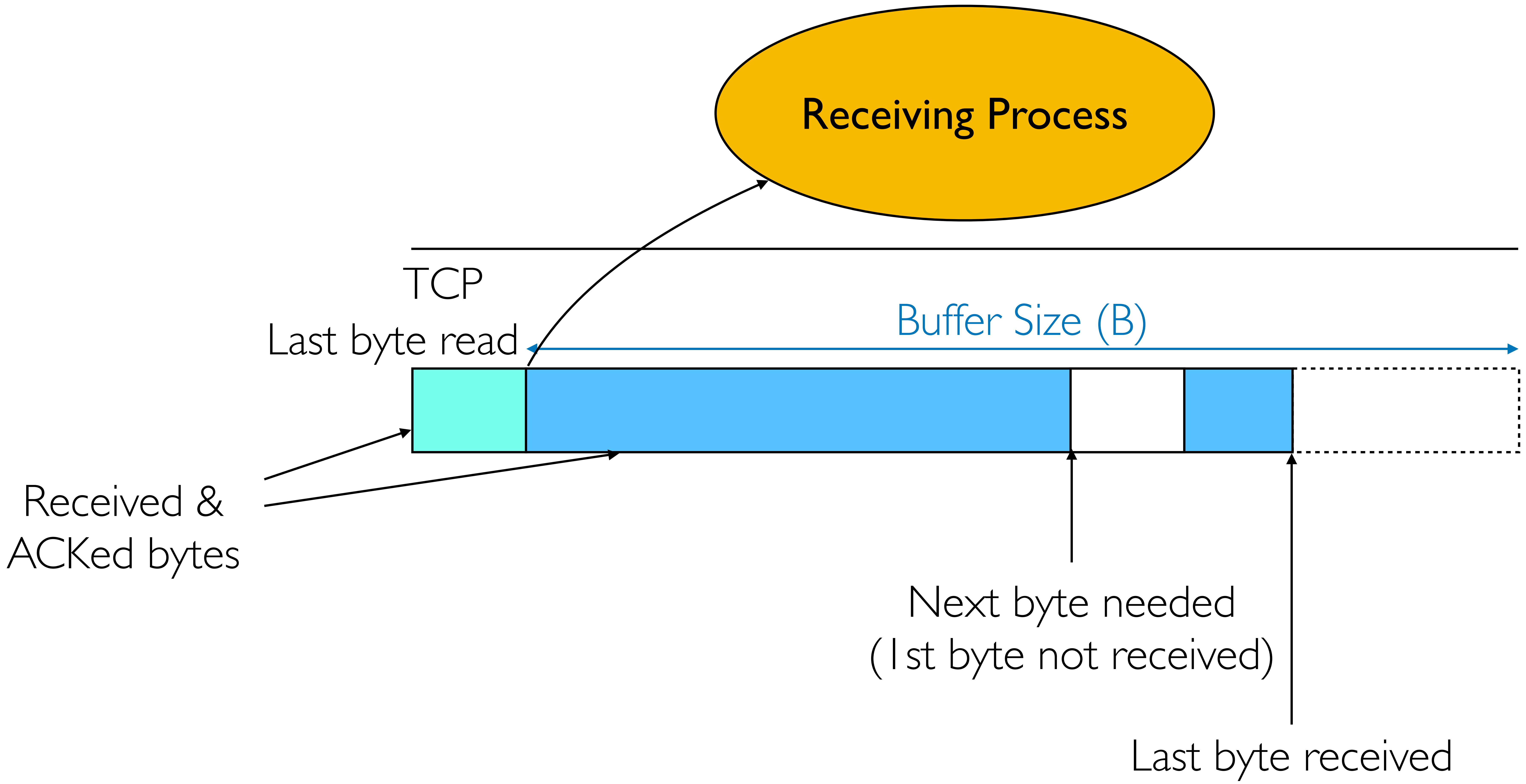
Sliding Window at Receiver



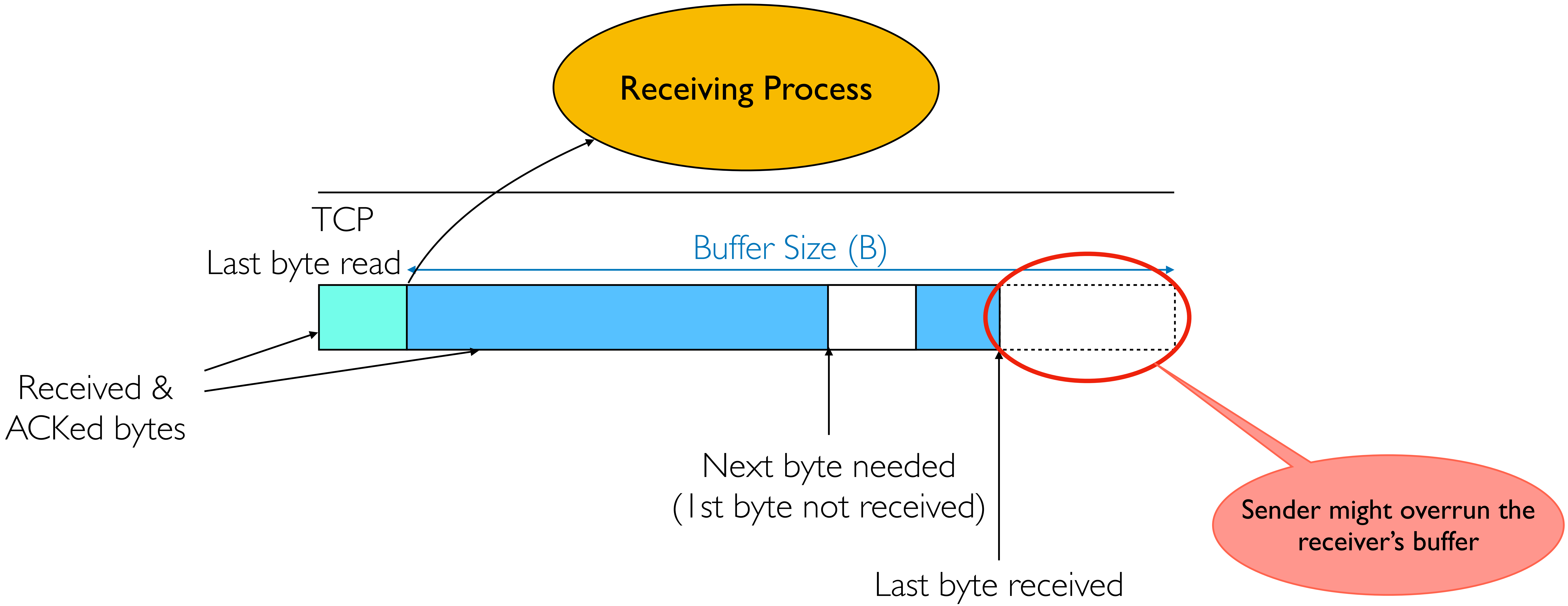
Sliding Window at Receiver



Sliding Window at Receiver



Sliding Window at Receiver

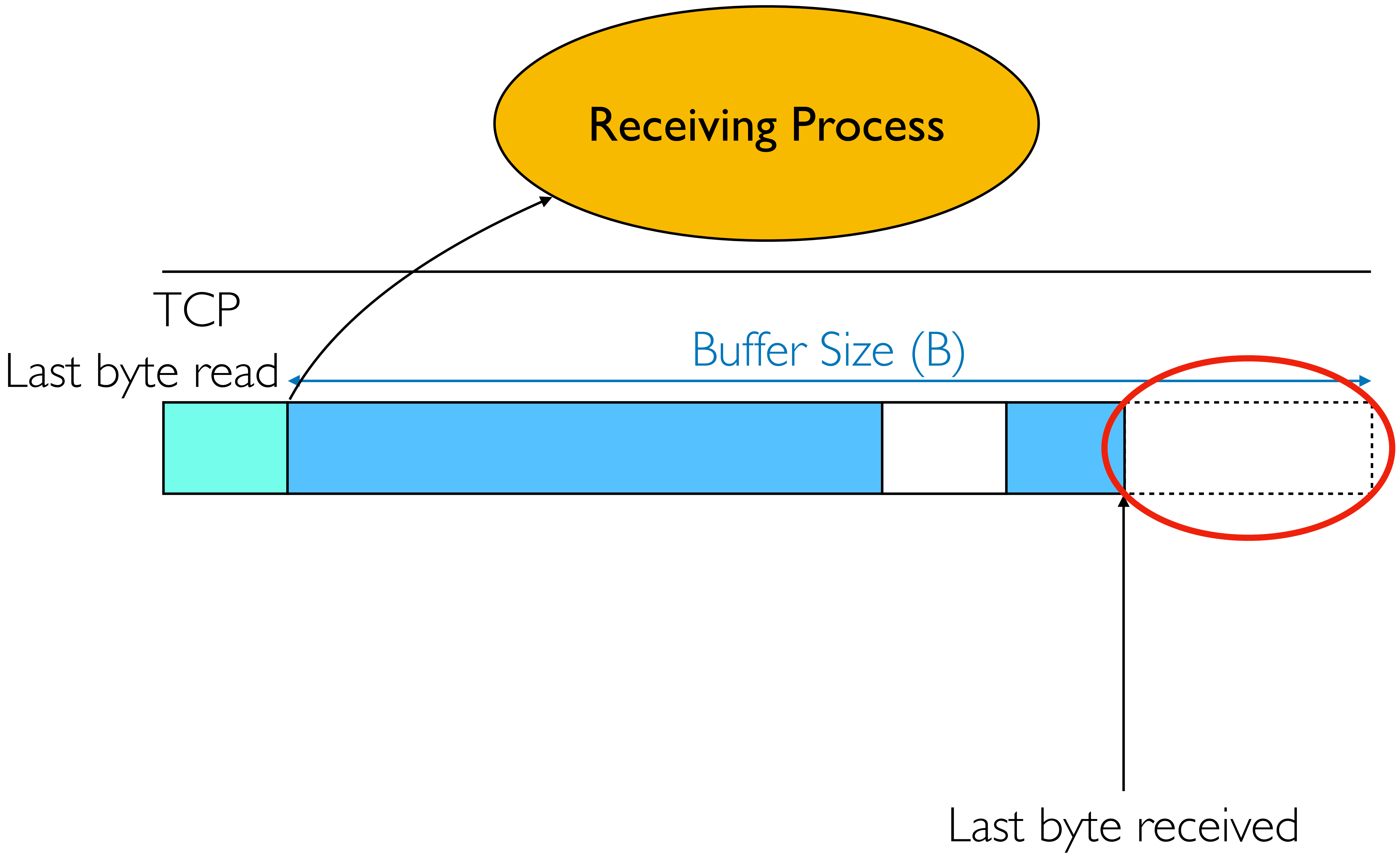


Solution: Advertised Window (Flow Control)

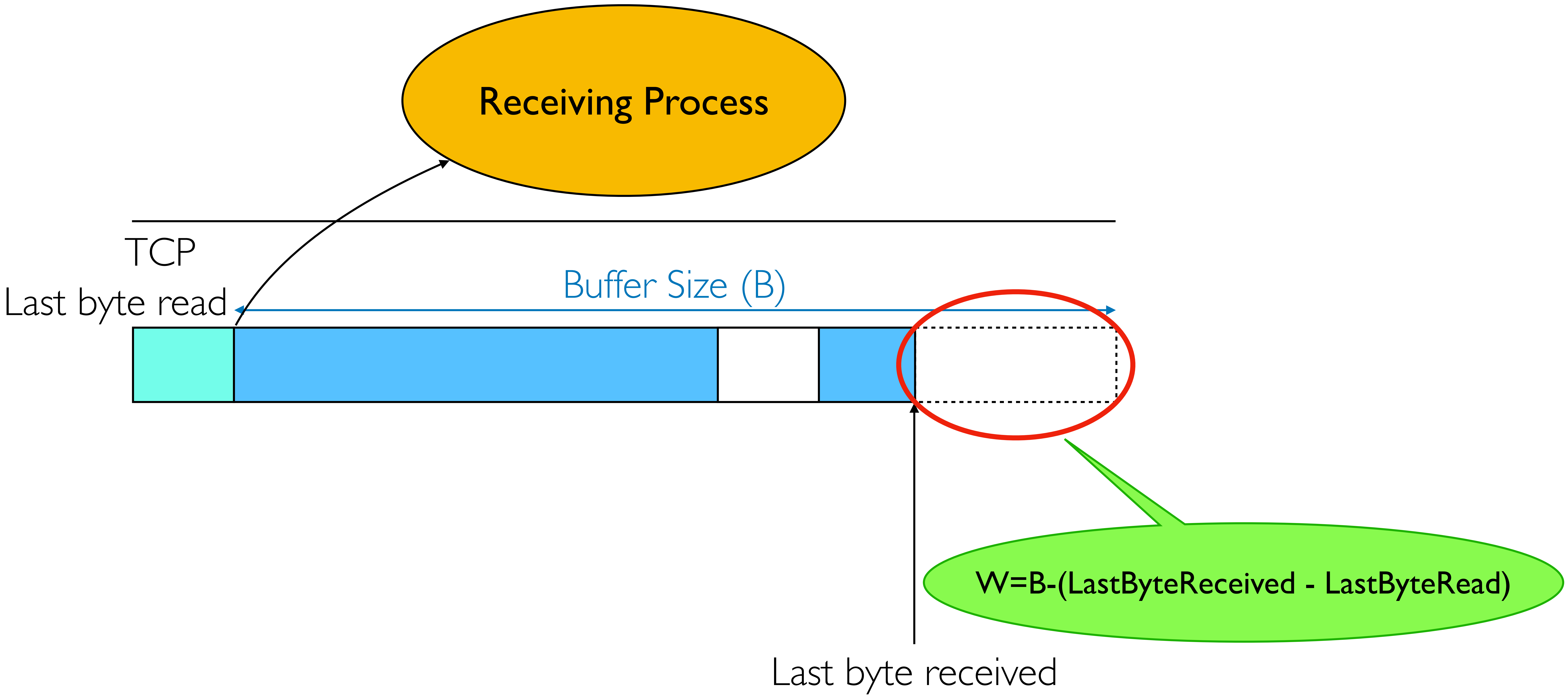
Solution: Advertised Window (Flow Control)

- **Receiver uses an “Advertised Window” (W) to prevent sender from overflowing its window**
  - Receiver indicates value of W in ACKs
  - Sender limits number of bytes it can have in flight <= W

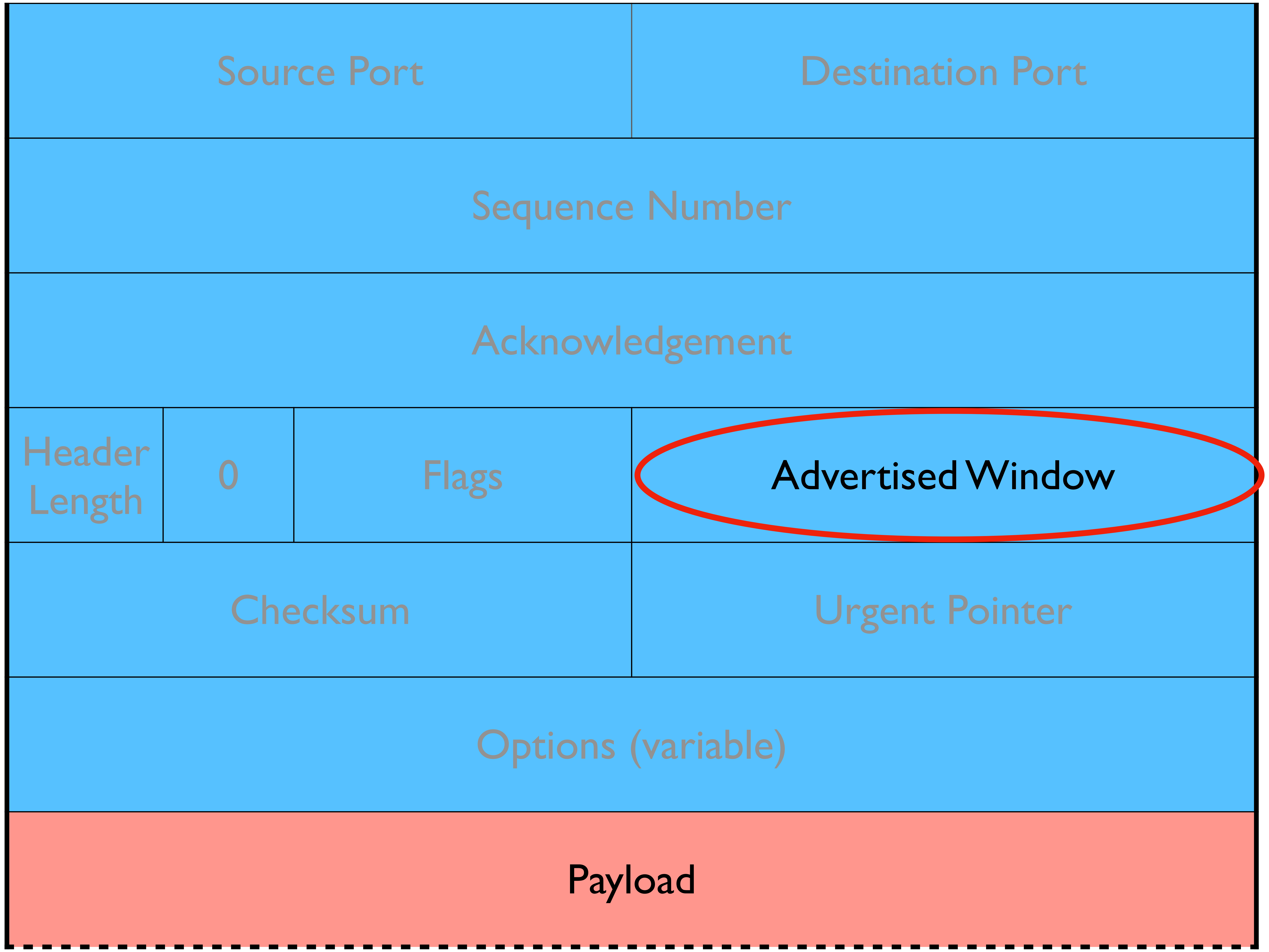
Sliding Window at Receiver



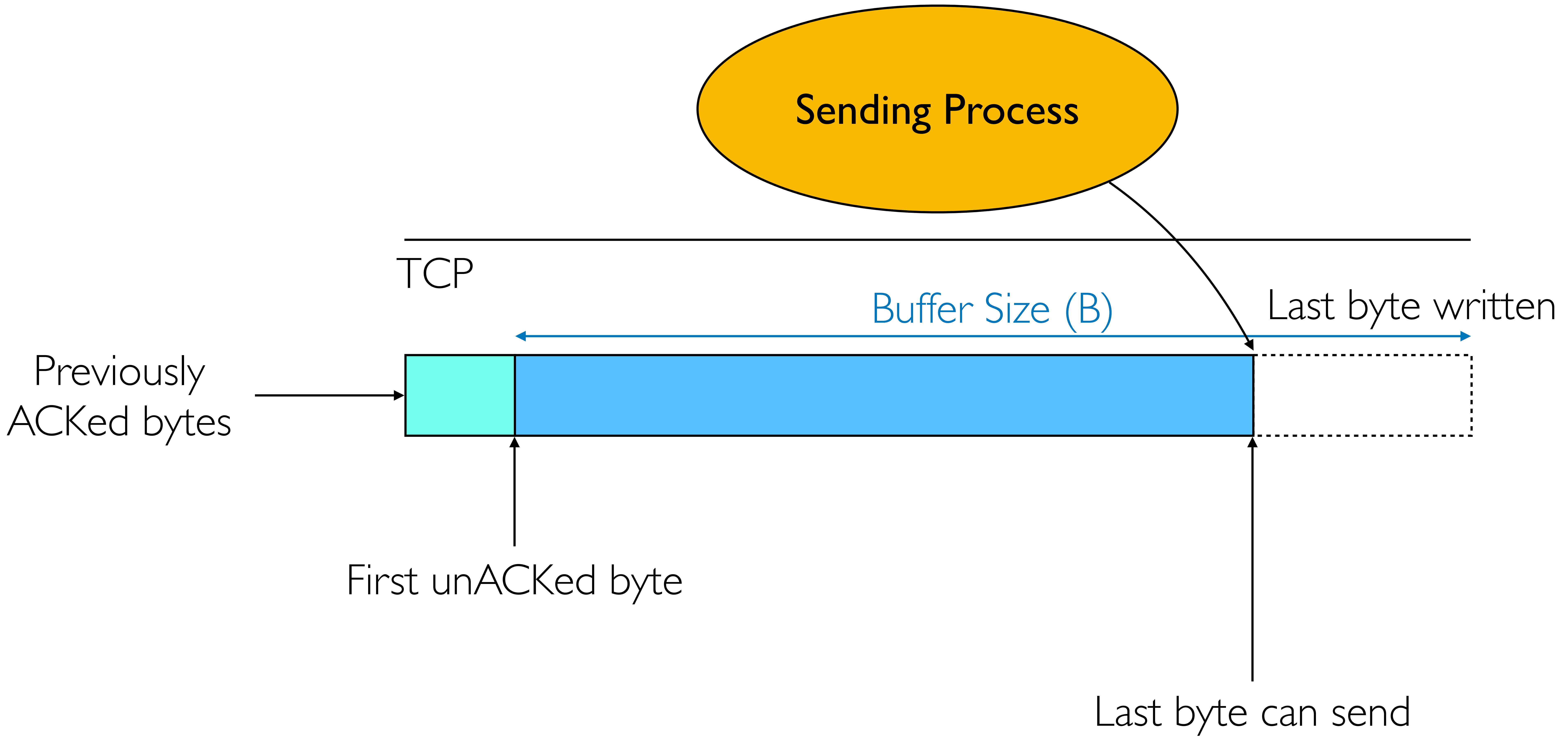
Sliding Window at Receiver



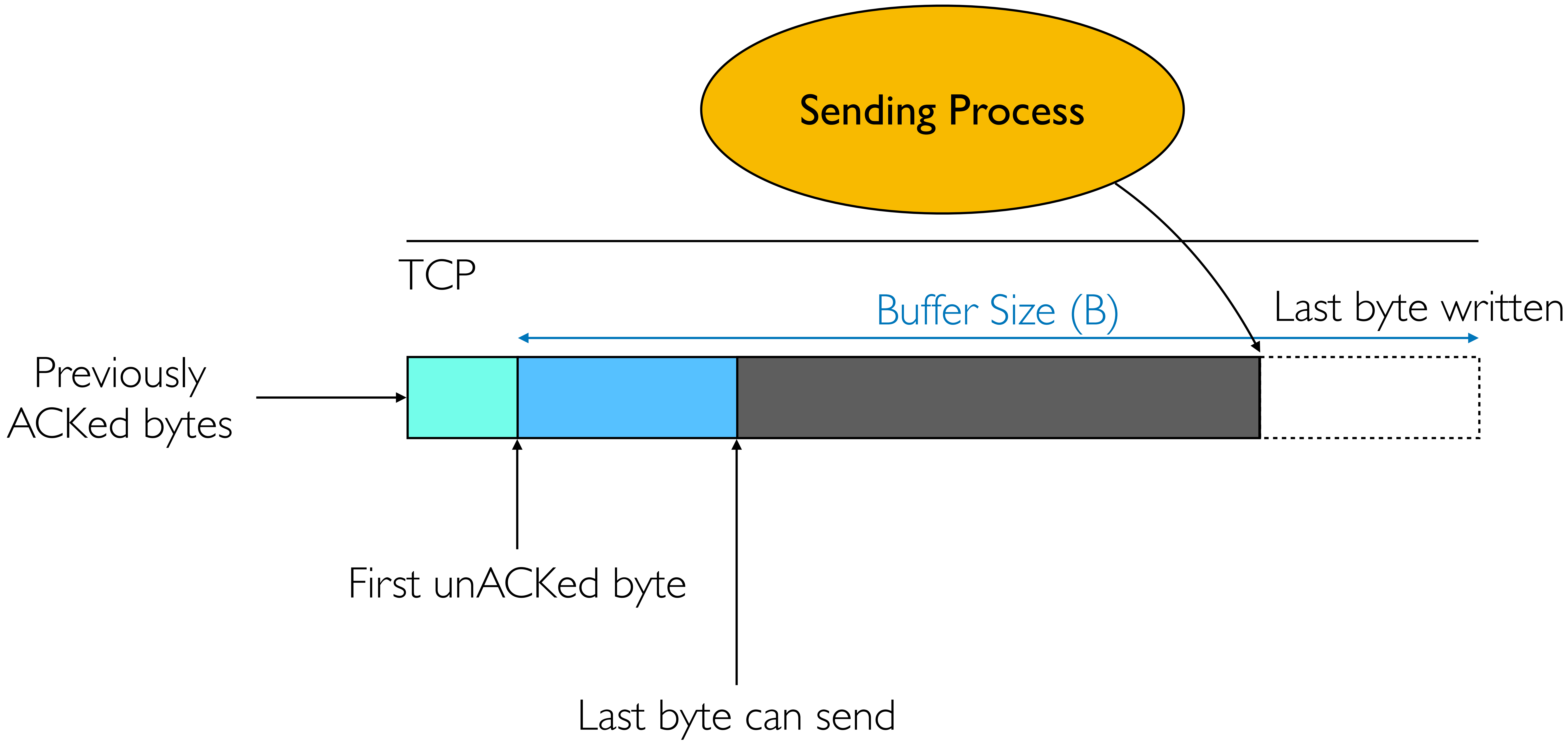
TCP Header


Source Port
Destination Port
Sequence Number
Acknowledgement
Header Length
0
Flags
Advertised Window
Checksum
Urgent Pointer
Options (variable)
Payload

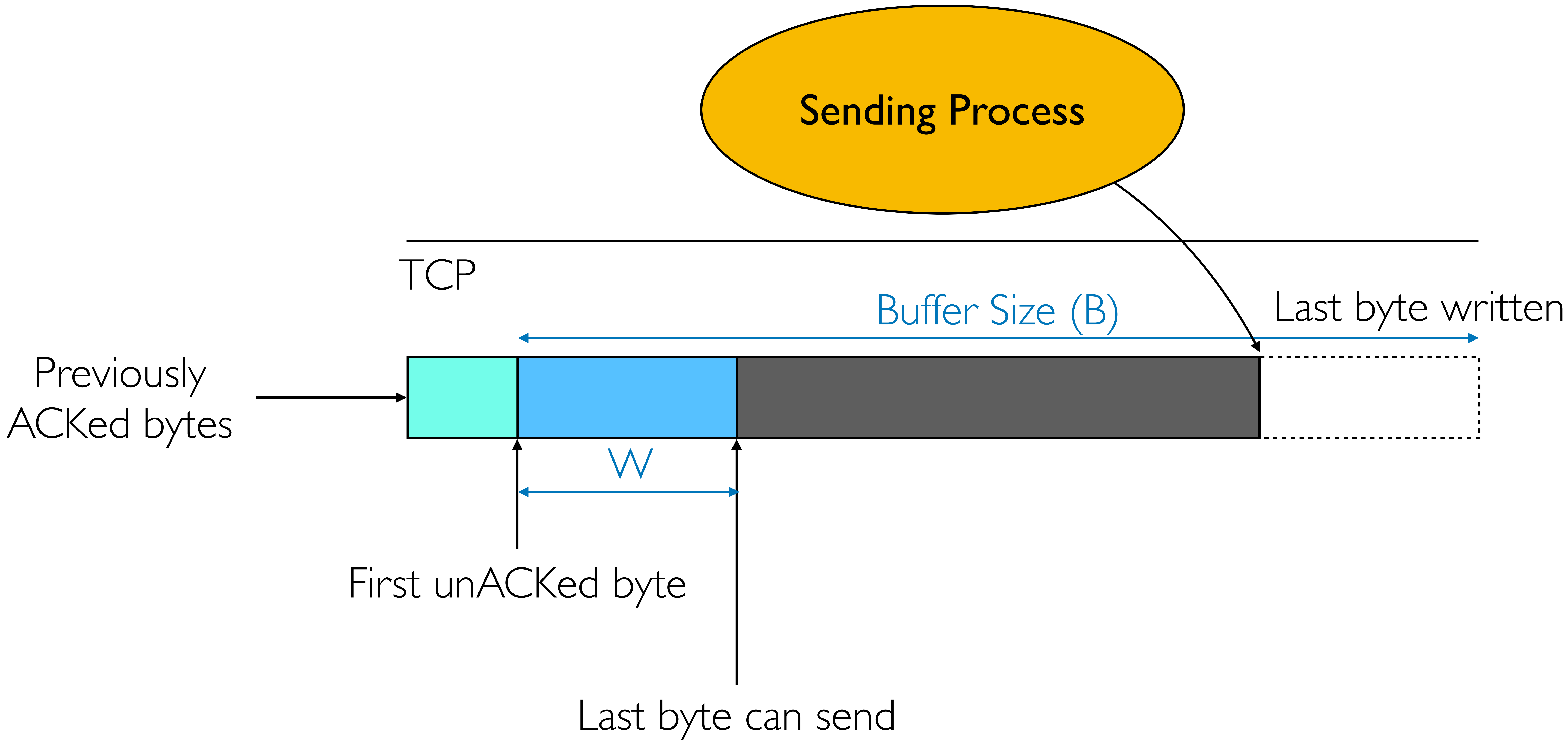
Sliding Window at Sender



Sliding Window at Sender



Sliding Window at Sender



Sending Window with Flow Control

Sending Window with Flow Control

- **Sender:** window advances *when new data ACKed*

Sending Window with Flow Control

- **Sender:** window advances *when new data ACKed*
- **Receiver:** window advances as *receiving process consumes data*

Sending Window with Flow Control

- **Sender:** window advances *when new data ACKed*
- **Receiver:** window advances as *receiving process consumes data*
- Receiver *advertises* to sender where receiver window currently ends ("right hand edge")
  - Sender agrees not to exceed this amount

Advertised Window Limits Rate

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling the sender's rate

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling the sender's rate
- What was missing?

Taking Stock

- The concepts underlying TCP are simple
  - Acknowledgements
  - Timers
  - Sliding Windows
  - Buffer Management
  - Sequence Numbers

Taking Stock

- The concepts underlying TCP are simple
- But tricky in the details
  - How do we set timers
  - What is the seqno for an ACK only packet
  - What happens if the advertised window = 0
  - What if the advertised window is 1/2 an MSS
  - Should receiver acknowledge packets right away
  - What if the application generates data in units of 0.1 MSS
  - What happens if I get a duplicate SYN? Or an RST while I'm in FIN_WAIT?
  - *etc., etc., etc.*

Taking Stock

- The concepts underlying TCP are simple
- But tricky in the details
- Do the details matter?

Sizing Windows for Congestion Control

- What are the problems?
- How might we address them?