Introduction to Computer Networks

CS640

TCP Reliability Support (I)

https://pages.cs.wisc.edu/~mgliu/CS640/S26/index.html

Ming Liu
mgliu@cs.wisc.edu

Outline

- Last
  - TCP Connection Management
- Today
  - TCP Reliability Support (I)
- Announcements
  - Quiz3 in class on 03/26/2026
  - Lab3 due on 03/27/2026

Recap

- Key Questions:
  - How can we destroy a TCP connection?

- Terminology
  - TCP state transition diagram

Recap: UDP Issues

- **#1: Arbitrary communication**
  - **Senders and receivers can talk to each other in any ways**
- #2: No reliability guarantee
  - Packets can be lost/duplicated/reordered during transmission
  - A checksum is not enough
- #3: No resource management
  - Each channel works as an exclusive network resource owner
  - No adaptive support for the physical networks and applications

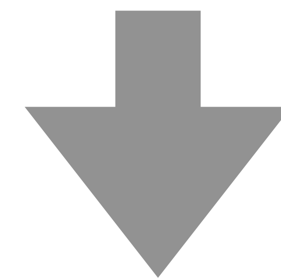
What is the goal of TCP reliability mechanisms?

What is the goal of TCP reliability mechanisms?

Byte stream @sender = Byte stream @receiver

What is the goal of TCP reliability mechanisms?

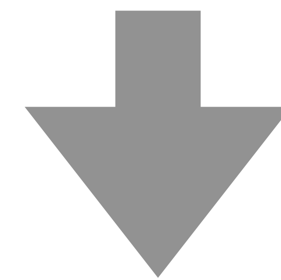
Byte stream @sender = Byte stream @receiver



- #1: TCP segments are delivered with no loss/duplication
- #2: TCP segments are delivered in order
- #3: The sender is not over-running the receiver capability

What is the goal of TCP reliability mechanisms?

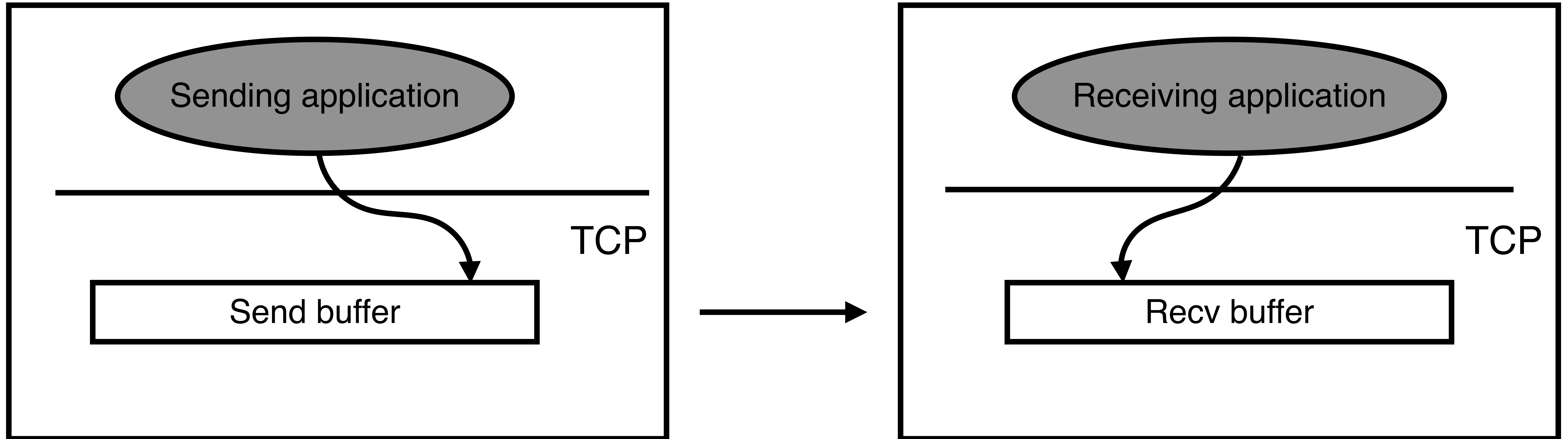
Byte stream @sender = Byte stream @receiver



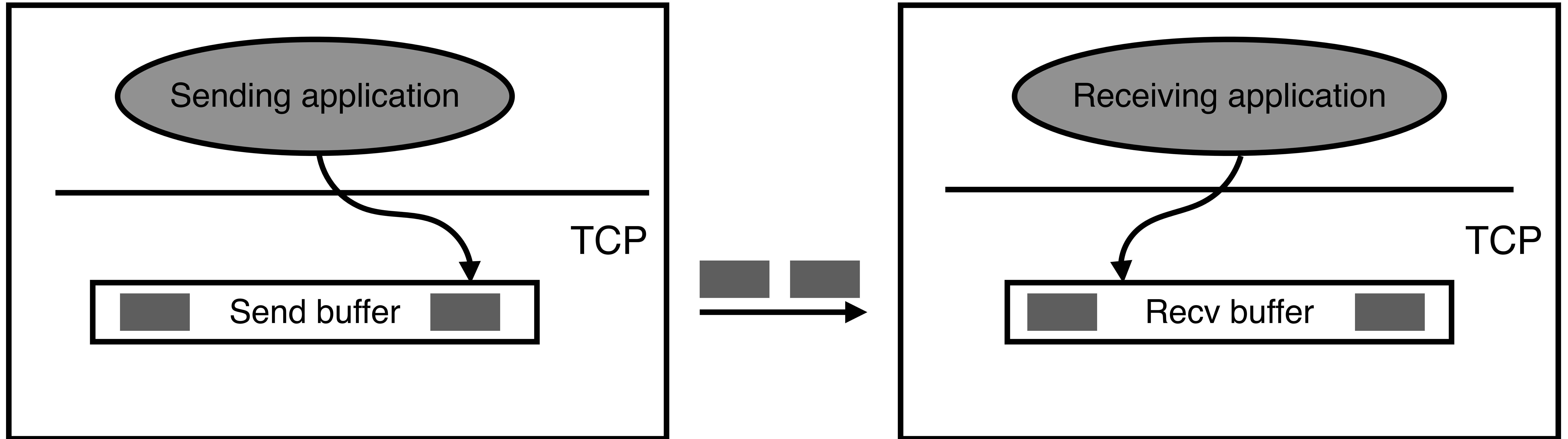
- #1: TCP segments are delivered with no loss/duplication
- #2: TCP segments are delivered in order
- #3: The sender is not over-running the receiver capability

TCP Segment: The smallest data transmission unit under TCP, consisting of a (segment) header and a data payload.

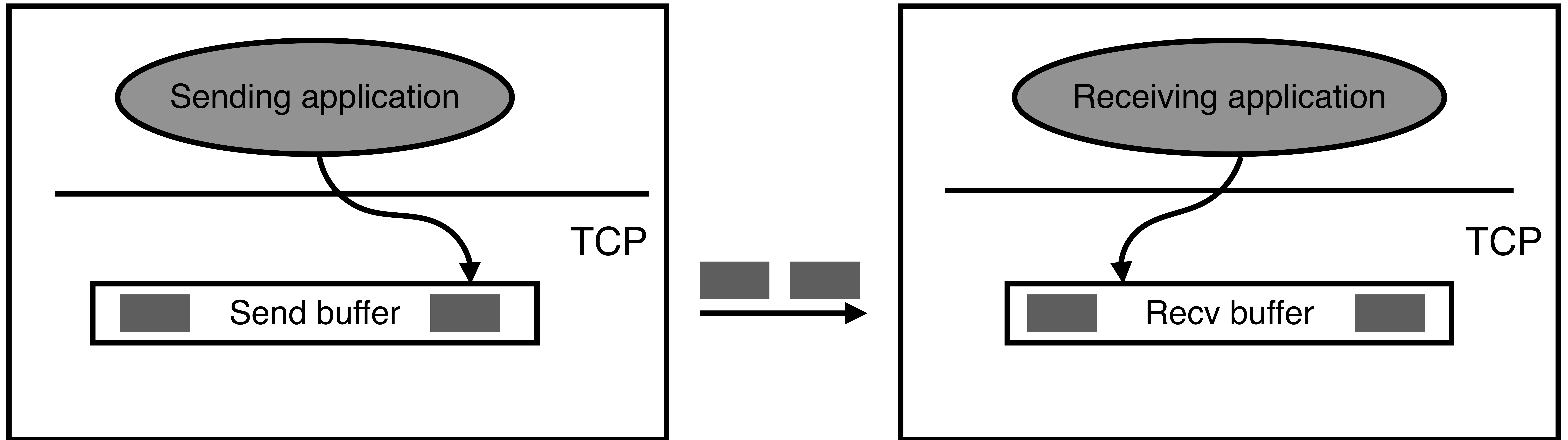
A TCP Send/Recv Example



A TCP Send/Recv Example

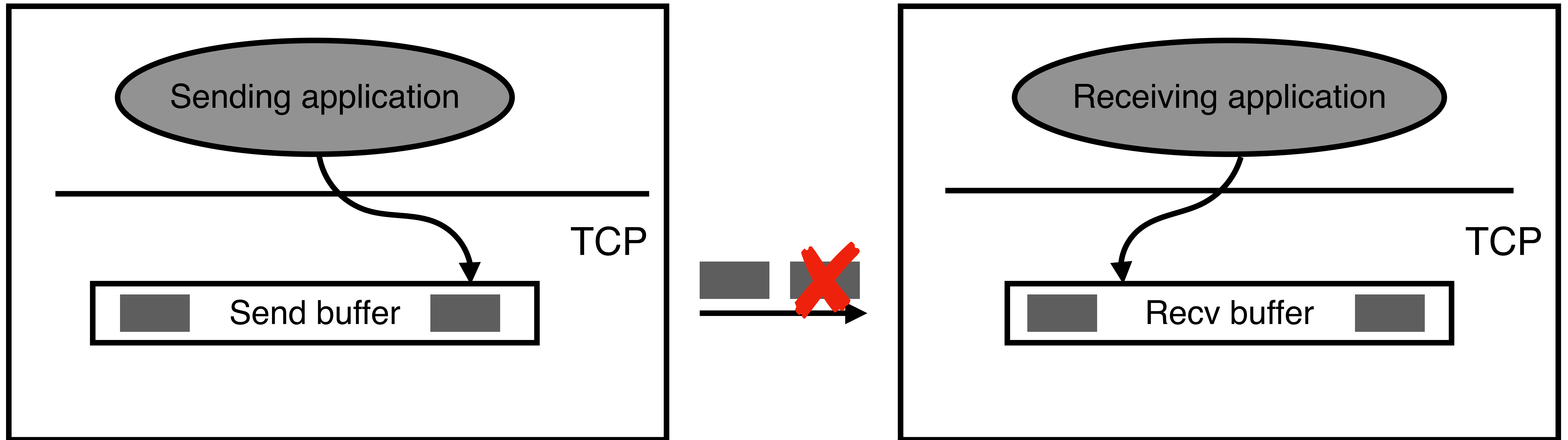


A TCP Send/Recv Example

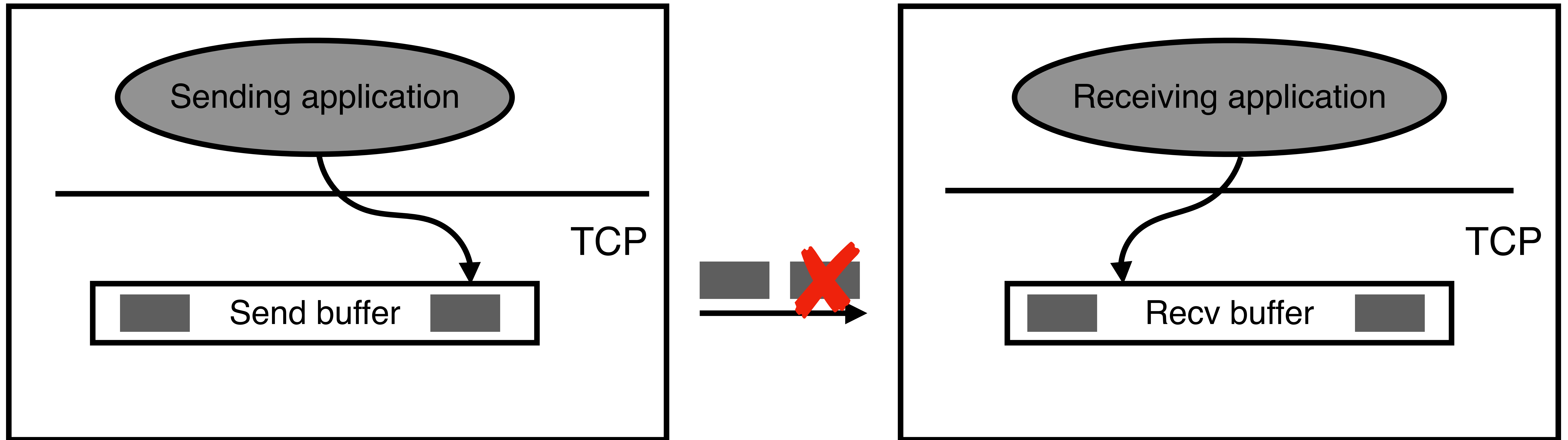


Send/Recv buffer is fixed-sized
(i.e., MaxSendBuffer and MaxRcvBuffer)

Issue #1: Segment Loss

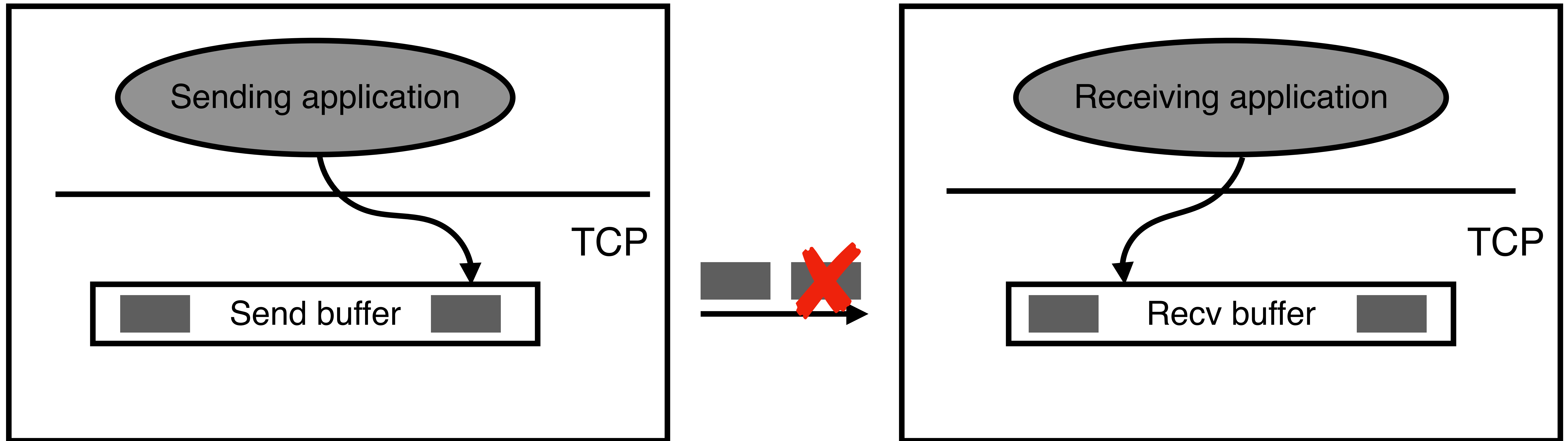


Issue #1: Segment Loss



- How do we know a segment is missing?
- How do we recover a missing segment?

Issue #1: Segment Loss



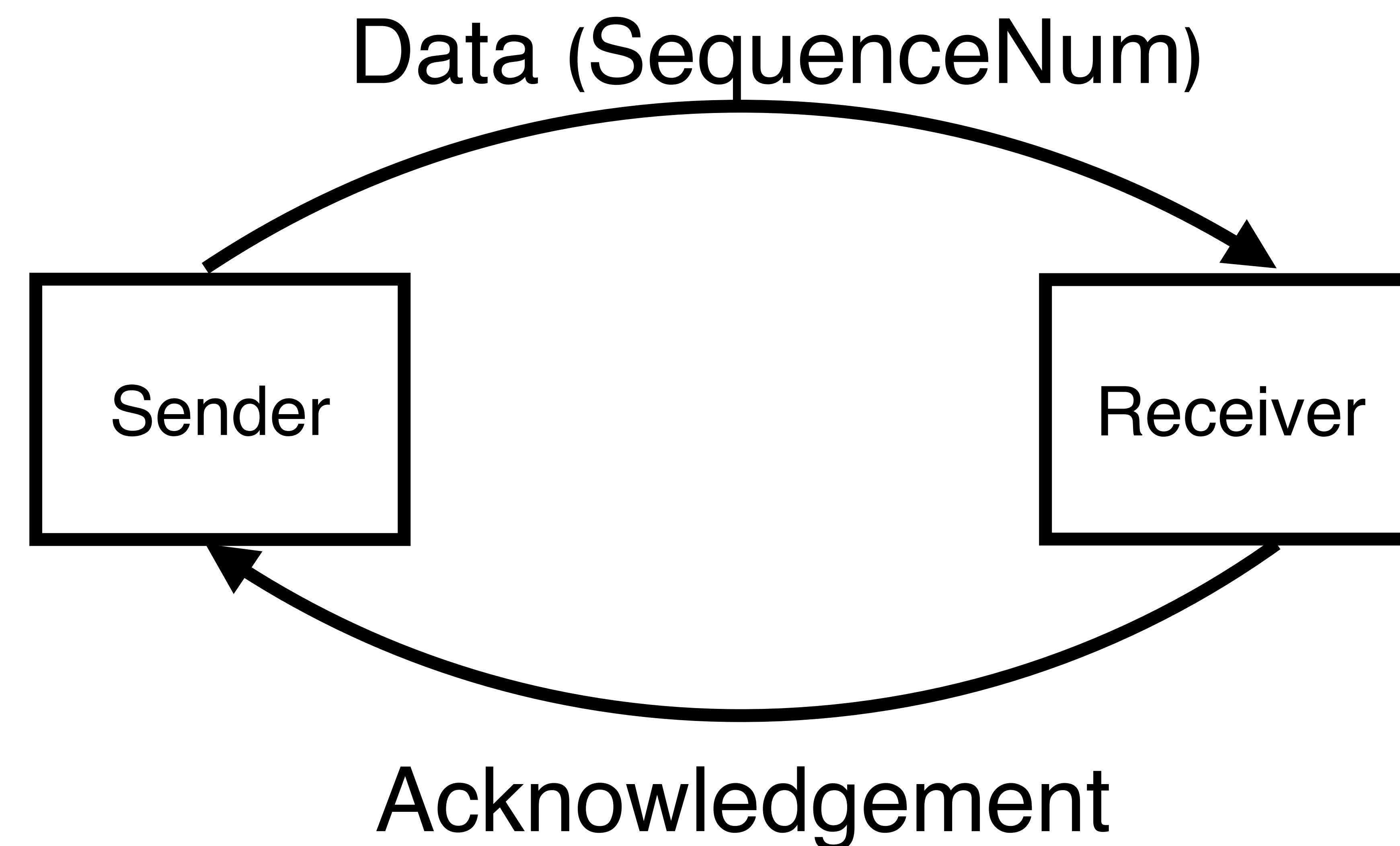
- How do we know a segment is missing? => Detection
- How do we recover a missing segment? => Correction

Detecting a Missing Segment

- Challenge: no in-network observability
  - We will revisit this later.
- Solution: use host-side indirect signals
  - Sender: check if the transmitted packets are confirmed by the receiver
  - Receiver: check if the byte stream misses any segments

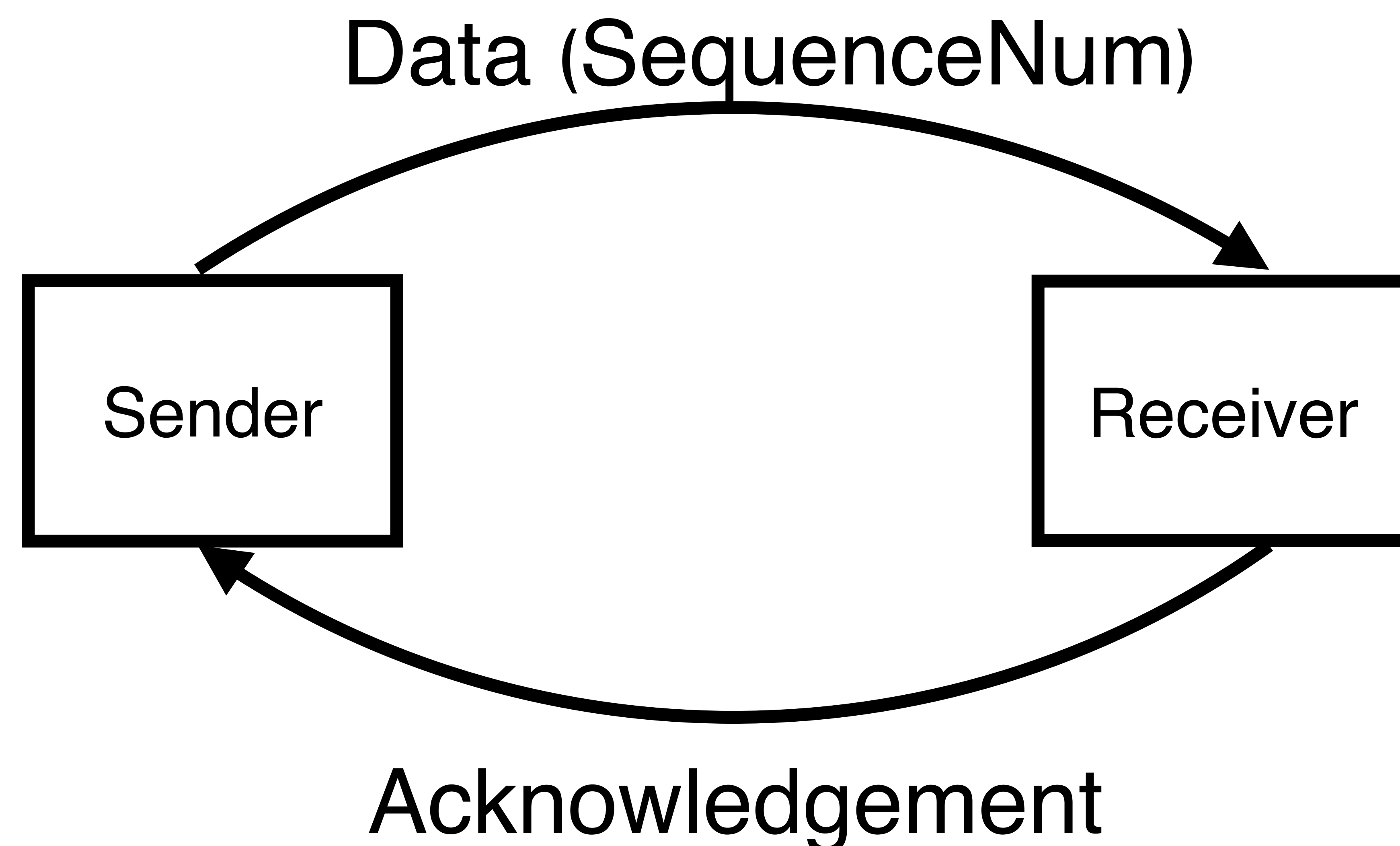
Sender-side Detection — Acknowledgment

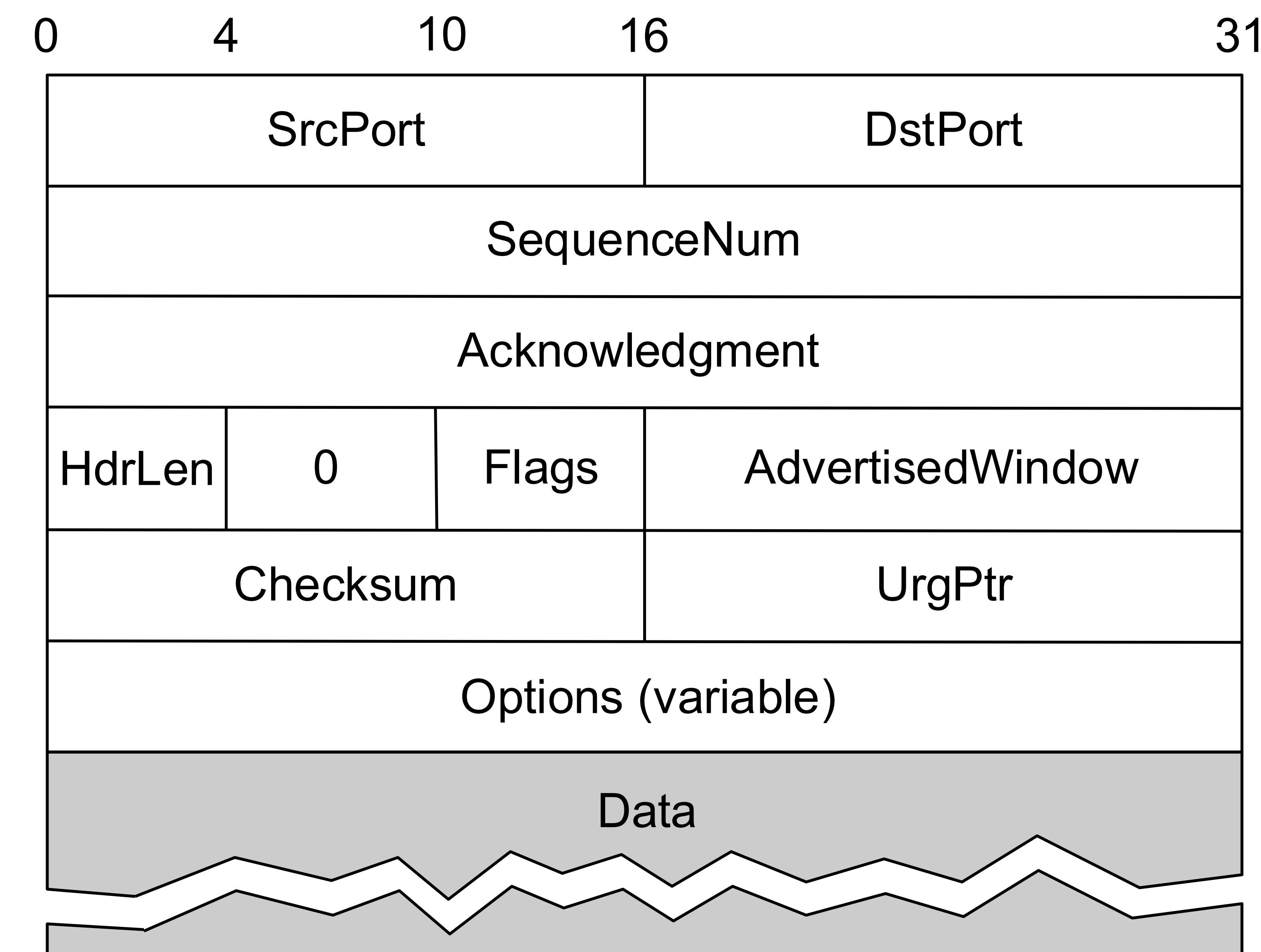
- Acknowledgment
  - Ask the receiver to send back an ACK when a segment is received
  - A missing ACK indicates a missing segment



Sender-side Detection — Acknowledgment

- Acknowledgment
  - Ask the receiver to send back an ACK when a segment is received
  - A missing ACK indicates a missing segment
- TCP realization
  - Making an ACK segment is simply changing a field in the header
  - Data can be piggybacked in ACKs





Sender-side Detection — Acknowledgment

- Acknowledgment
  - Ask the receiver to send back an ACK when a segment is received
  - A missing ACK indicates a missing segment
- TCP realization
  - Making an ACK segment is simply changing a field in the header
  - Data can be piggybacked in ACKs

0 4 10 16 31

Is this good enough?

Acknowledgement

Sender-side Detection — Acknowledgment

- Acknowledgment
  - Ask the receiver to send back an ACK when a segment is received
  - A missing ACK indicates a missing segment
- TCP realization
  - Making an ACK segment is simply changing a field in the header
  - Data can be piggybacked in ACKs

0 4 10 16 31

How can we define a “missing ACK”?

Acknowledgement

Sender-side Detection — Timeout

- Timeout
  - A signal that a segment was sent but has not received its ACK within a specified time frame (threshold)

- A missing segment
  - The corresponding ACK is not received before the timeout is triggered

Sender-side Detection — Timeout

- Timeout
  - A signal that a segment was sent but has not received its ACK within a specified time frame (threshold)
- A missing segment
  - The corresponding ACK is not received before the timeout is triggered

But how to set the timer (threshold)?

The Art of Timeout Threshold

- This is tricky because the network condition is dynamic.

- Approach: use RTTs to estimate the timeout period
  - RTT = the delay between transmission and receipt of packets between the sender and the receiver
  - Measure the RTT of each segment online and dynamically adjust the threshold
  - One RTT is not sufficient since we need to capture the network dynamics

EWMA for RTT Estimation

- EWMA = Exponentially Weighted Moving Average

- EWMA mechanisms
  - #1: Measure SampleRTT for each segment/ACK pair
  - #2: Compute the weighted average of RTT
    - $EstimatedRTT = \alpha \times EstimatedRTT + \beta \times SampleRTT, where\ \alpha + \beta = 1$
    - $0.8 \leq \alpha \leq 0.9$
    - $0.1 \leq \beta \leq 0.2$
  - #3: Set timeout based on the EstimatedRTT
    - $TimeOut = 2 \times EstimatedRTT$

Sender-side Detection Summary

- Acknowledgment
  - Ask the receiver to send back an ACK when a segment is received
  - A missing ACK indicates a missing segment

- Timeout
  - A signal that a segment that was sent but has not received its ACK within a specified time frame (threshold)
  - EWMA = Exponentially Weighted Moving Average

Receiver-side Detection

- Sequence number
  - Ask the sender to assign a unique sequence number for each segment
  - A missing sequence number indicates a segment loss

Receiver-side Detection

- Sequence number
  - Ask the sender to assign a unique sequence number for each segment
  - A missing sequence number indicates a segment loss

Seg 15 | Seg 16 | Seg 17 | ? | Seg 19 | Seg 20 | Seg 21

Is this good enough?

Receiver-side Detection

- Sequence number
  - Ask the sender to assign a unique sequence number for each segment
  - A missing sequence number indicates a segment loss

Seg 15 | Seg 16 | Seg 17 | ? | Seg 19 | Seg 20 | Seg 21

How can we differentiate between a missing segment and a slow-arriving (out-of-order) segment?

Receiver-side Detection

- Sequence number
  - Ask the sender to assign a unique sequence number for each segment
  - A missing sequence number indicates a segment loss

- Approaches
  - #1: view out-of-order segments as missing
  - #2: apply timeout again

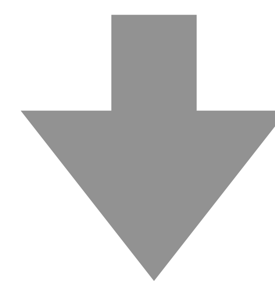
How should we recover the missing segment?

How should we recover the missing segment?

Just send it again!

How should we recover the missing segment?

Just send it again!

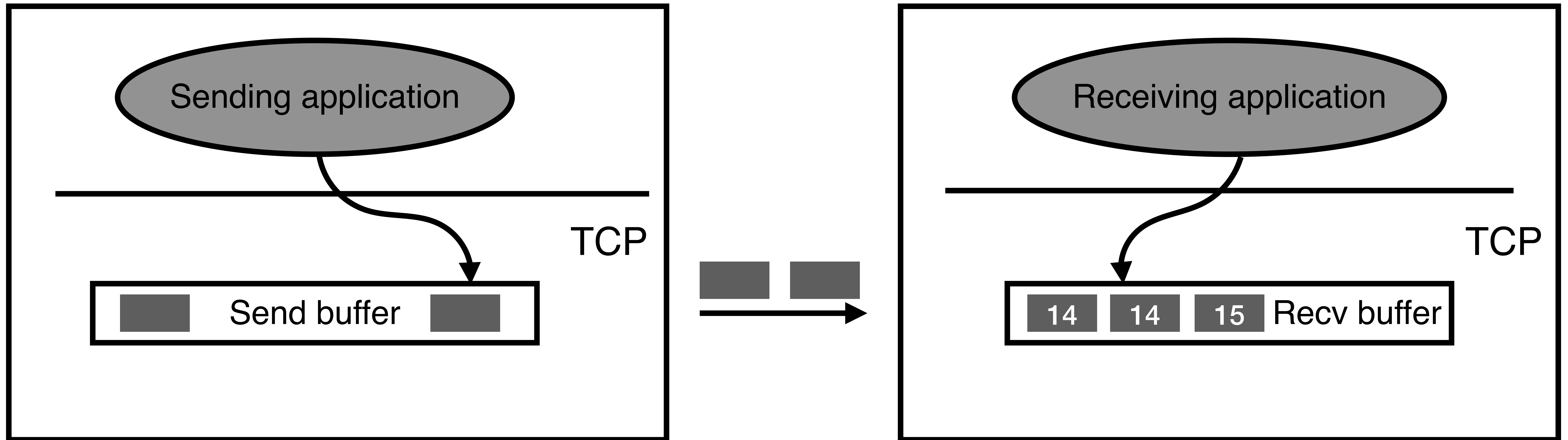


The sender must keep the segment until receiving the ACK.

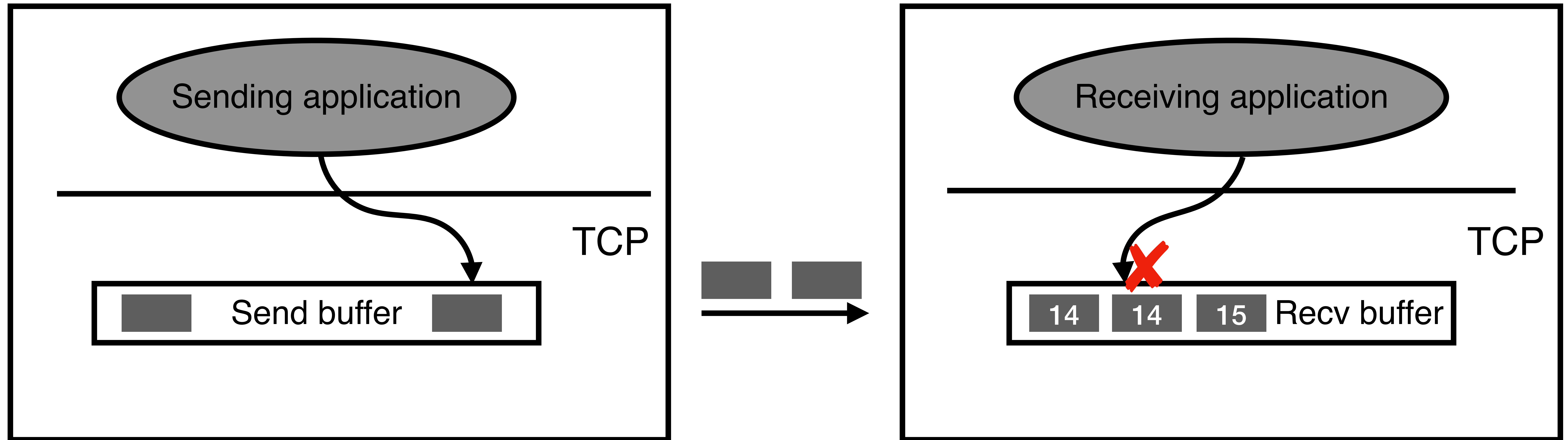
Recover a Missing Segment

- Sender logic
  - Retransmit a segment when its local timer is triggered
  - Retransmit a segment when receiving an explicit ask from the receiver
- Receiver logic
  - Send an explicit ask to fetch the missing segment
  - Co-leasing or piggyback optimizations are possible to save bandwidth

Issue #2: Duplicated Segment



Issue #2: Duplicated Segment



- How do we know a segment is duplicated?
- How do we handle segment duplication?

Receiver-side Detection and Fix

- The segment holds the same sequence number as a prior one
  - Seems simple, but how?

Receiver-side Detection and Fix

- The segment holds the same sequence number as a prior one
  - Seems simple, but how?

The receiver must maintain the sequence number of all received segments.

Receiver-side Detection and Fix

- The segment holds the same sequence number as a prior one
  - Seems simple, but how?

The receiver must maintain the sequence number of all received segments.

- Drop the duplicated segment directly

Receiver-side Detection and Fix

- The segment holds the same sequence number as a prior one
  - Seems simple, but how?

The receiver must maintain the sequence number of all received segments.

- Drop the duplicated segment directly

Duplication is an important signal!

Understanding Duplication

- Why can the receiver receive a duplicated segment?

Understanding Duplication

- Why can the receiver receive a duplicated segment?
  - Because the sender sends the segment again!

Understanding Duplication

- Why can the receiver receive a duplicated segment?
  - Because the sender sends the segment again!

- Why can the sender send the segment again?

Understanding Duplication

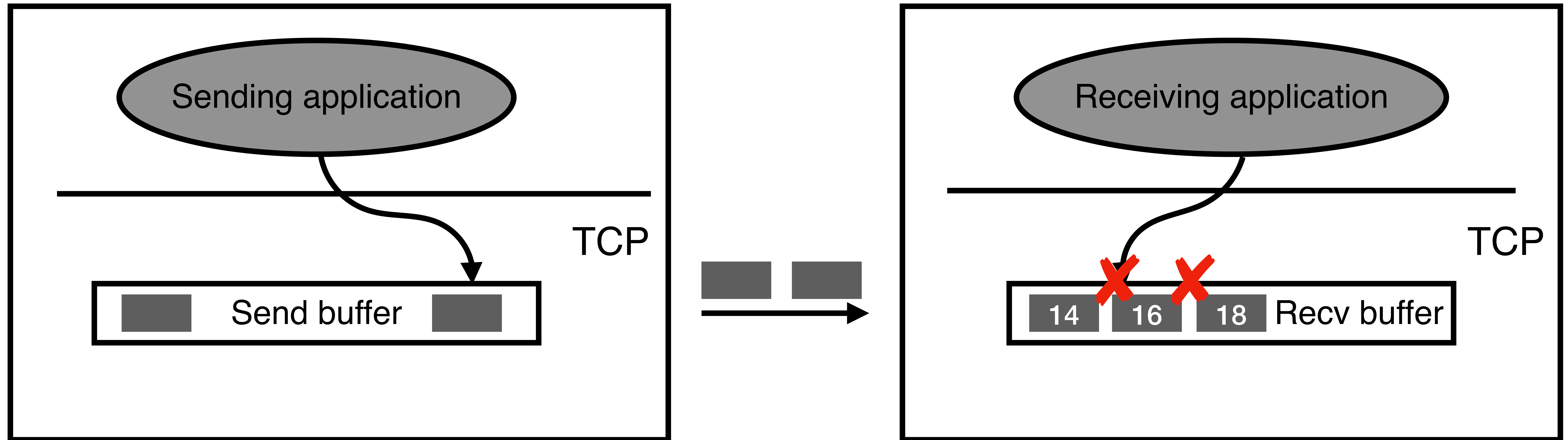
- Why can the receiver receive a duplicated segment?
  - Because the sender sends the segment again!

- Why can the sender send the segment again?
  - Case #1: my local timeout is triggered
  - Case #2: the receiver sends an explicit ask

Understanding Duplication

- Why can the receiver receive a duplicated segment?
  - Because the sender sends the segment again!
- Why can the sender send the segment again?
  - Case #1: my local timeout is triggered
  - Case #2: the receiver sends an explicit ask
- The network is slow
  - The sender should slow down

Issue #3: Out-of-order Segment



- How do we know a segment is out-of-order?
- How do we handle out-of-order segments?

Receiver-side Detection and Fix

- There is a segment hole in the data stream
  - The receiver should know what the next expected segment is
  - A hole happens when the receiving segment number is not as expected

Receiver-side Detection and Fix

- There is a segment hole in the data stream
  - The receiver should know what the next expected segment is
  - A hole happens when the receiving segment number is not as expected

- Solution #1: just drop it and wait for the retransmission
  - Pro: simple logics
  - Con: waste bandwidth and hurt performance

Receiver-side Detection and Fix

- There is a segment hole in the data stream
  - The receiver should know what the next expected segment is
  - A hole happens when the receiving segment number is not as expected
- Solution #1: just drop it and wait for the retransmission
  - Pro: simple logics
  - Con: waste bandwidth and hurt performance
- Solution #2: take it and reconstruct the stream until the hole fills
  - Pro: reduce retransmission and improve performance
  - Con: complex logics

Understanding Out-of-order

- Why can the receiver receive an out-of-order segment?

Understanding Out-of-order

- Why can the receiver receive an out-of-order segment?
  - #1: multiple transmission paths
  - #2: segments are dropped

Understanding Out-of-order

- Why can the receiver receive an out-of-order segment?
  - #1: multiple transmission paths
  - #2: segments are dropped
- Missing v.s. Out-of-order

Understanding Out-of-order

- Why can the receiver receive an out-of-order segment?
  - #1: multiple transmission paths
  - #2: segments are dropped
- Missing v.s. Out-of-order
  - Sometimes they are the same since the indicator is a segment hole
  - But missing segments can also triggered by timeout

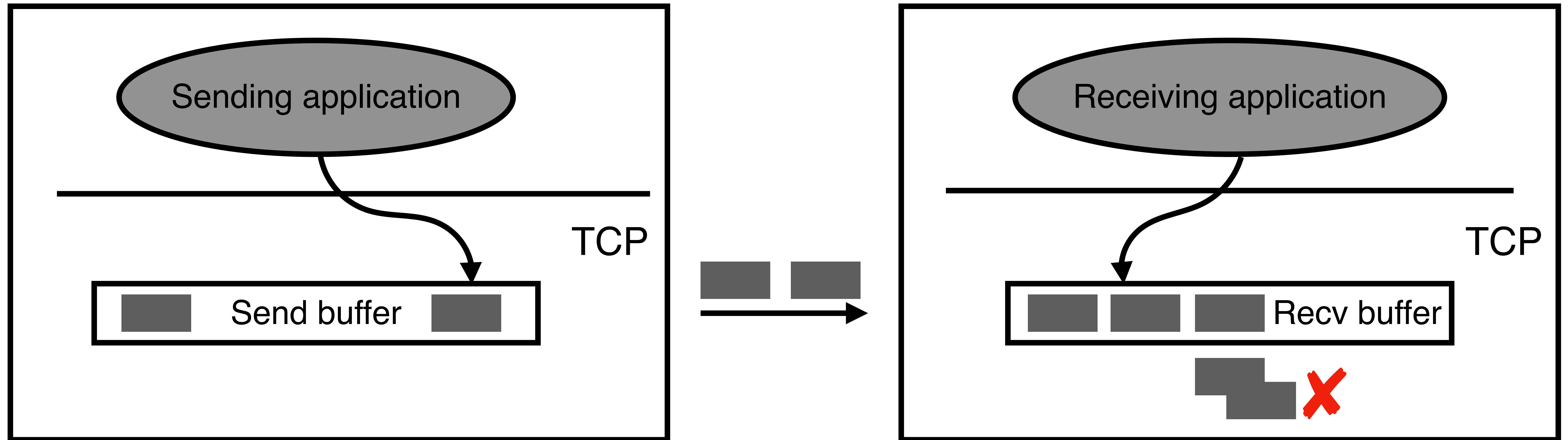
Understanding Out-of-order

- Why can the receiver receive an out-of-order segment?
  - #1: multiple transmission paths
  - #2: segments are dropped

- Missing v.s. Out-of-order
  - Sometimes they are the same since the indicator is a segment hole
  - But missing segments can also triggered by timeout

- The network is unstable.
  - Congestion happens during the transmission.
  - Communication paths become heterogeneous.

Issue #4: Receiver Overwhelming



- How do we know the receiver is overwhelmed?
- How do we handle the receiver overwhelming?

Detection and Fix

- Receiver-side
  - The receiver buffer is full
  - *More advanced, the receiver cannot pull the NIC buffer fast enough*

Detection and Fix

- Receiver-side
  - The receiver buffer is full
  - *More advanced, the receiver cannot pull the NIC buffer fast enough*

- Solution
  - Ask the sender to slow down explicitly

Detection and Fix

- Receiver-side
  - The receiver buffer is full
  - *More advanced, the receiver cannot pull the NIC buffer fast enough*

- Solution
  - Ask the sender to slow down explicitly
  - But, by how much?

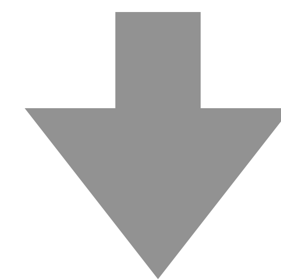
Detection and Fix

- Receiver-side
  - The receiver buffer is full
  - *More advanced, the receiver cannot pull the NIC buffer fast enough*

- Solution
  - Ask the sender to slow down explicitly
  - But, by how much? => Tell the sender my buffer availability

What is the goal of TCP reliability mechanisms?

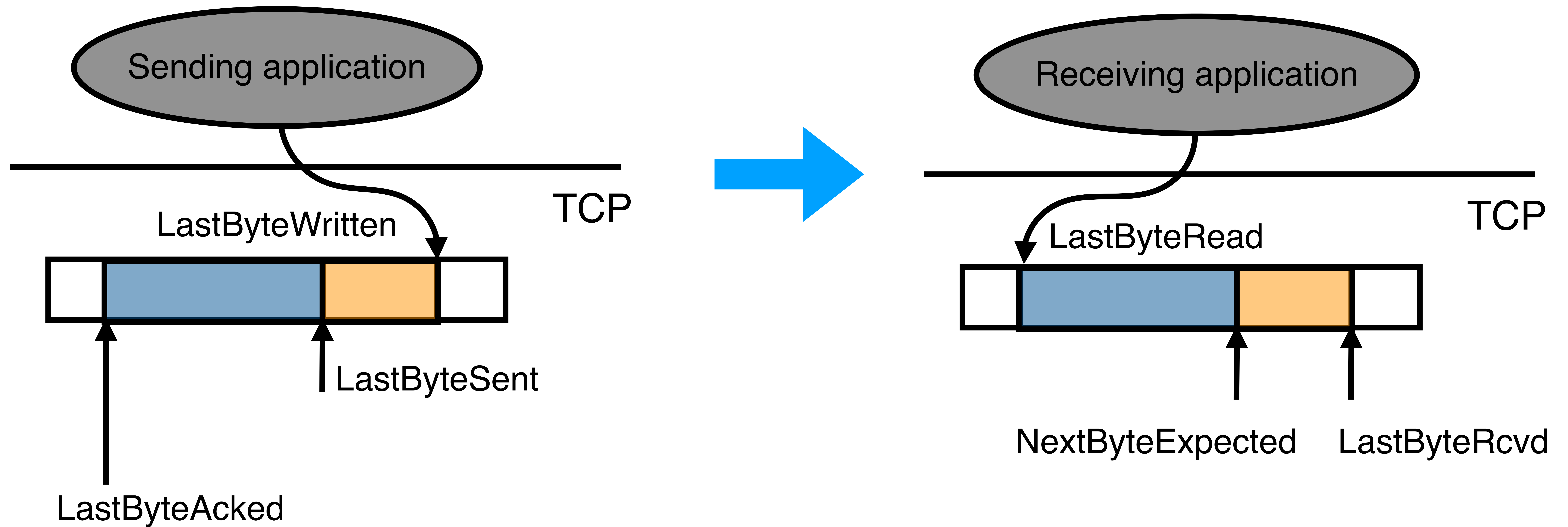
Byte stream @sender = Byte stream @receiver



- #1: TCP segments are delivered with no loss/duplication
- #2: TCP segments are delivered in order
- #3: The sender is not over-running the receiver capability

Combine Everything Together — TCP Sliding Window

- Continuously coordinate sender and receiver during transmission



TCP Sliding Window—Sender

- Four state variables
  - The last byte written by the application (**LastByteWritten**)
  - The last byte being acknowledged (**LastByteAcked**)
  - The last byte sent (**LastByteSent**)
  - The sender buffer size (**MaxSendBuffer**)

TCP Sliding Window—Sender Logics

- Three variables manipulations:
  - Advance **LastByteWritten** when an app writes
  - Advance **LastByteAcked** when a consecutive ACK arrived
  - Advance **LastByteSent** when the segments are sent

- Invariants:
  - **LastByteSent** <=**LastByteWritten**
  - **LastByteAcked** <= **LastByteSent**

- Buffered bytes:
  - **|LastByteWritten - LastByteAcked|** <= **MaxSendBuffer**

TCP Sliding Window—Receiver

- Four state variables
  - The last byte read by the application (**LastByteRead**)
  - The last byte received (**LastByteRcvd**)
  - The next byte supposed to be received (**NextByteExpected**)
  - The receiver buffer size (**MaxRcvBuffer**)

TCP Sliding Window—Receiver Logics

- Three variables manipulations:
  - Advance **LastByteRead** when an app reads
  - Advance **LastByteRcvd** when the segment is received
  - Advance **NextByteExpected** when the next expected segment is received
- Invariants:
  - **LastByteRead** < **NextByteExpected**
  - **NextByteExpected** <= **LastByteRcvd** + 1
- Buffered bytes:
  - **|LastByteRcvd - LastByteRead| <= MaxRcvBuffer**

Summary

- Today
  - TCP reliability support (I)

- Next lecture
  - TCP reliability support (II)