Introduction to Computer Networks

CS640

TCP Reliability Support (II)

https://pages.cs.wisc.edu/~mgliu/CS640/S25/index.html

Ming Liu
mgliu@cs.wisc.edu

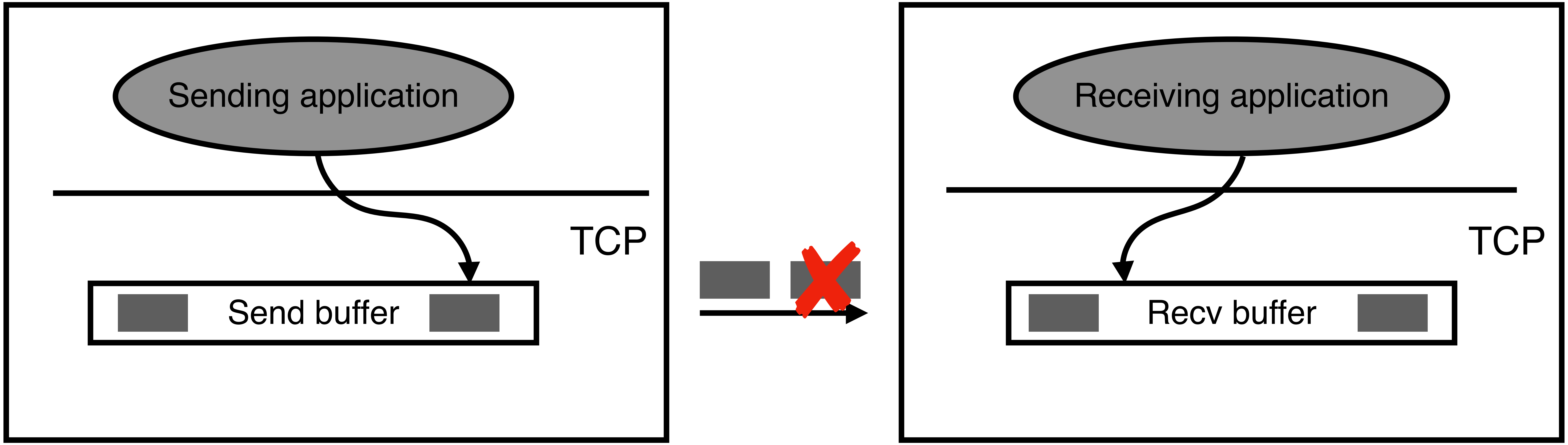
Outline

- Last
  - TCP Reliability Support (I)

- Today
  - TCP Reliability Support (II)

- Announcements
  - Lab 4 due date 05/01/2025 12:01PM

Issue #1: Segment Loss



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

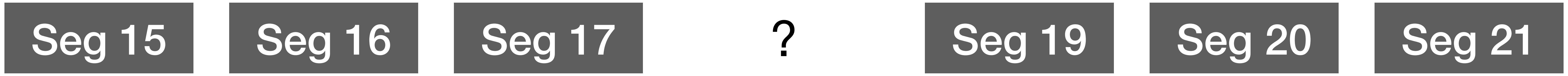
Sender-side Detection

- 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

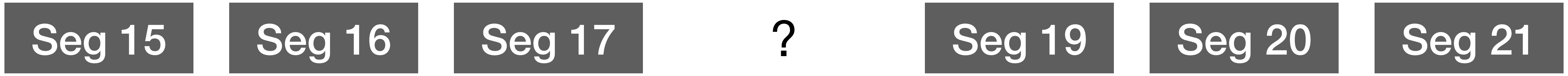
- Sequence number
  - Ask the sender to assign a unique sequence number for each segment
  - A missing sequence number indicates a segment loss



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

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

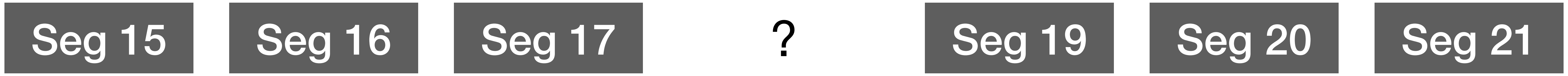


Seg 15 Seg 16 Seg 17 ? Seg 19 Seg 20 Seg 21

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

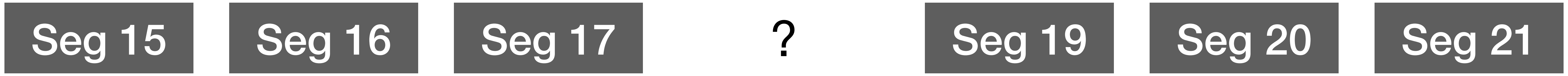


Seg 15 Seg 16 Seg 17 ? Seg 19 Seg 20 Seg 21

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



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

- 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!

↓

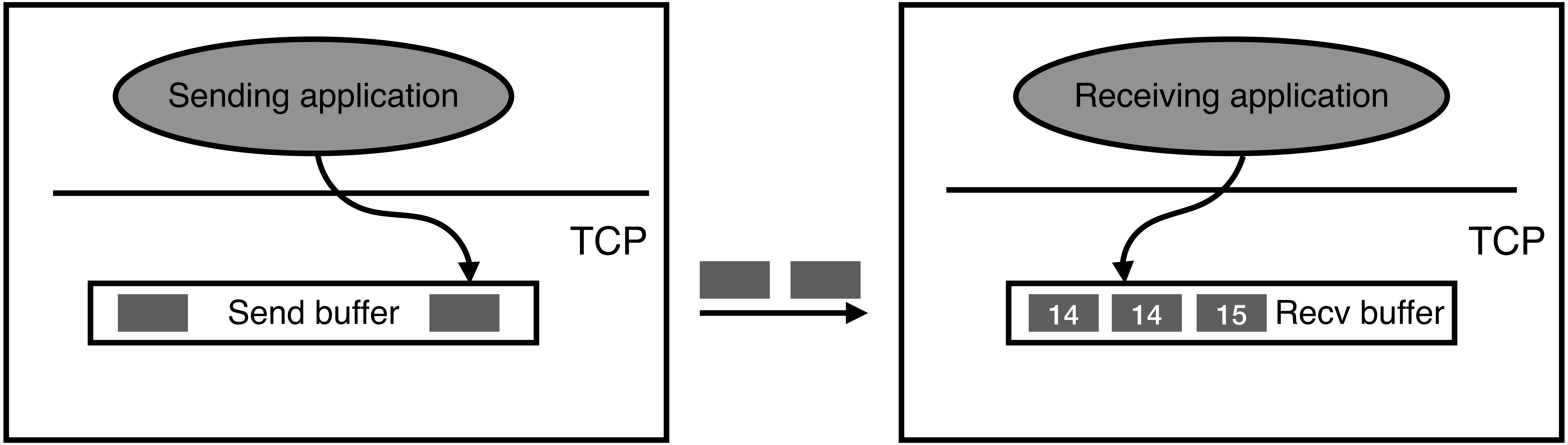
The sender must keep the segment until receiving the ACK.

Recover a Missing Segment

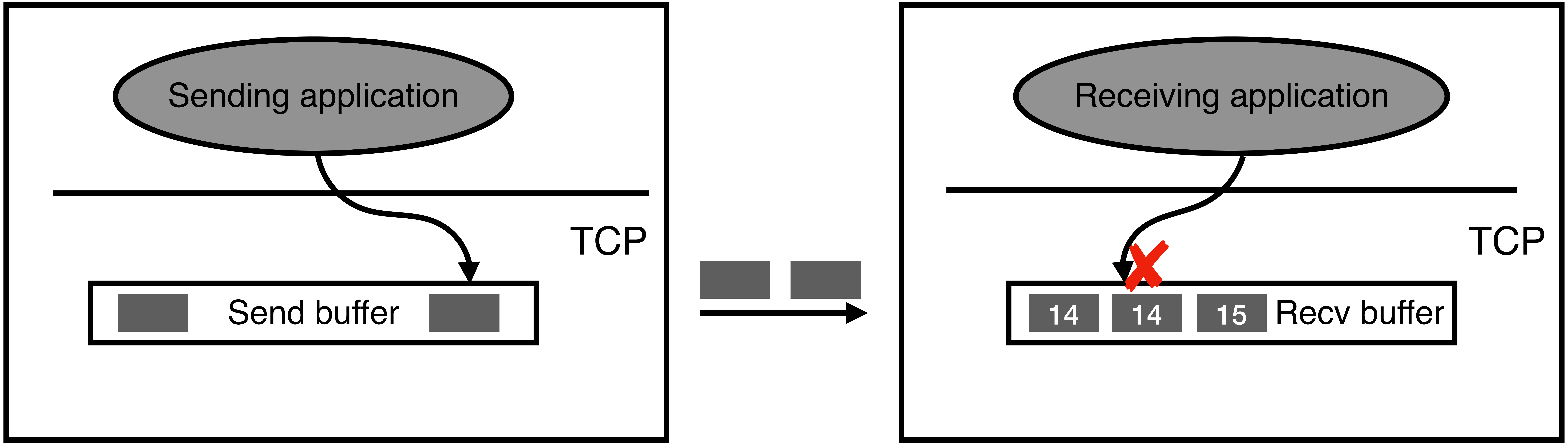
- Sender logic
  - Retransmit a segment when its local timer is triggered
  - Retransmit a segment when receiving an explicit ask from the receiver

- 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

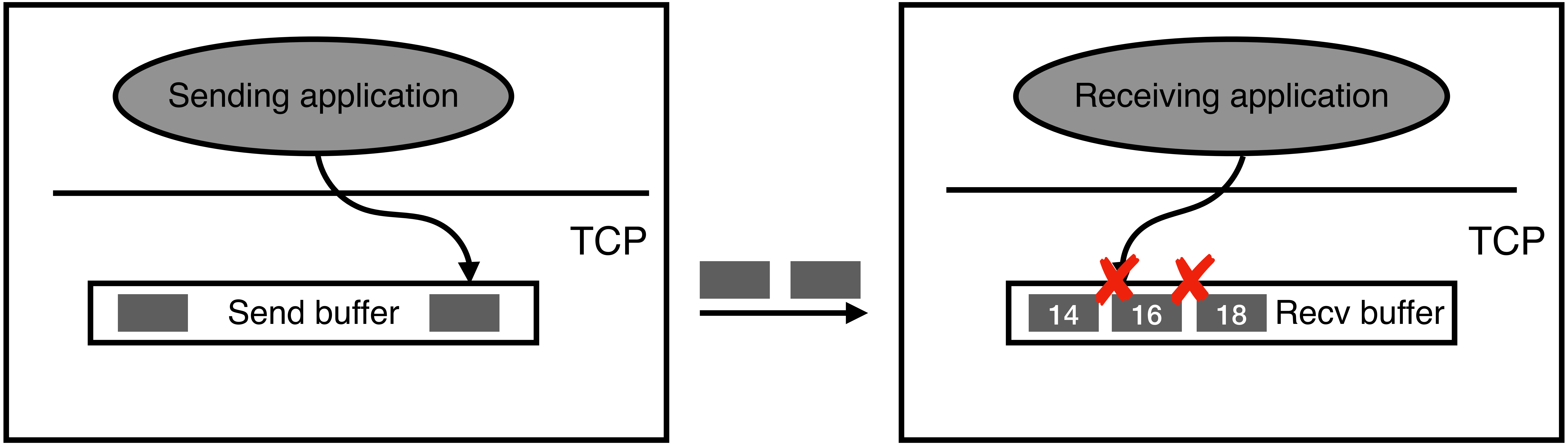
- Why can the receiver receive a duplicated segment?
  - Because the sender sends the segment again!

- 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 hold 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

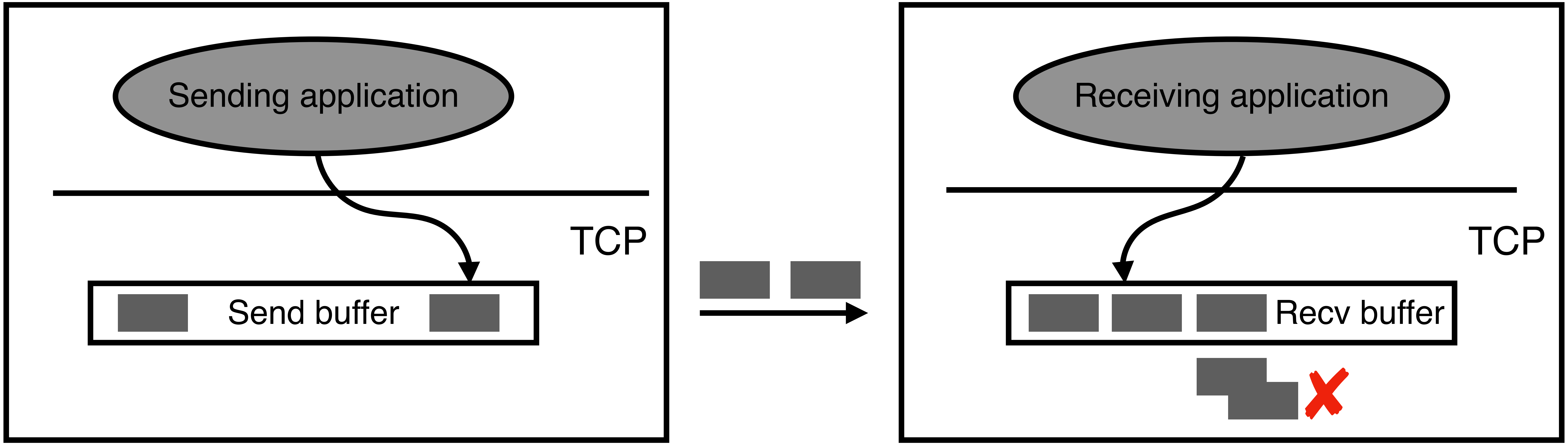
Understanding Out-of-order

- 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?

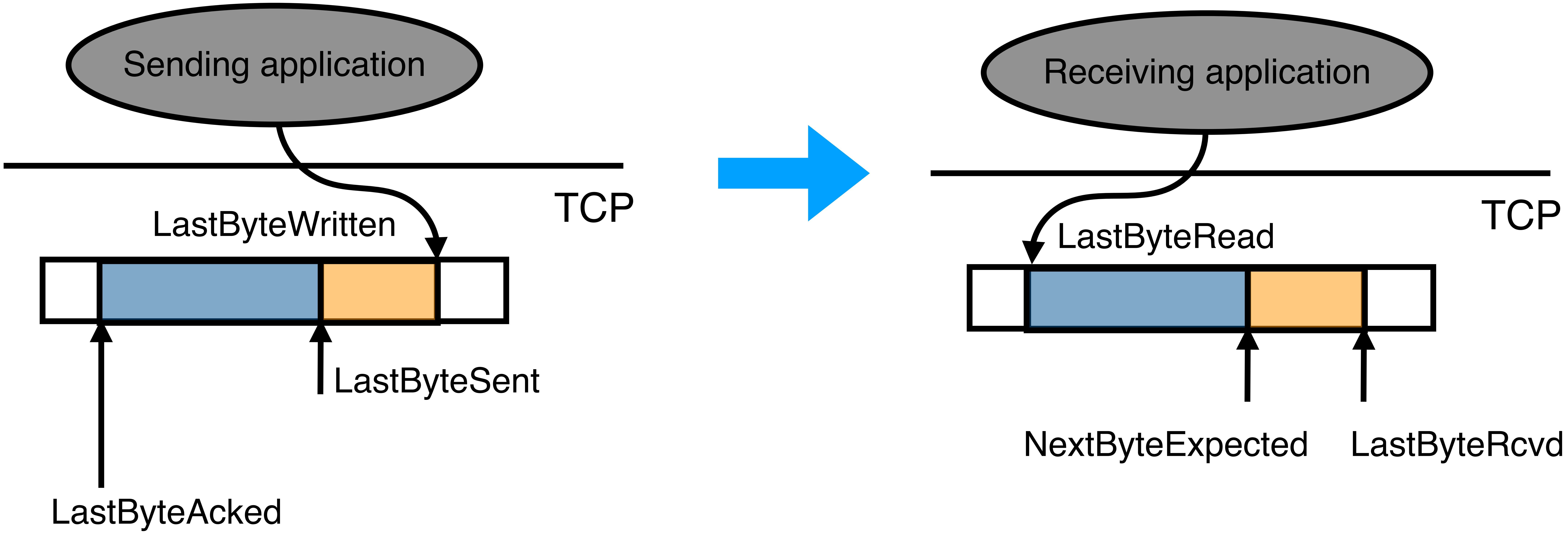
Byte stream @sender = Byte stream @receiver

↓

- #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

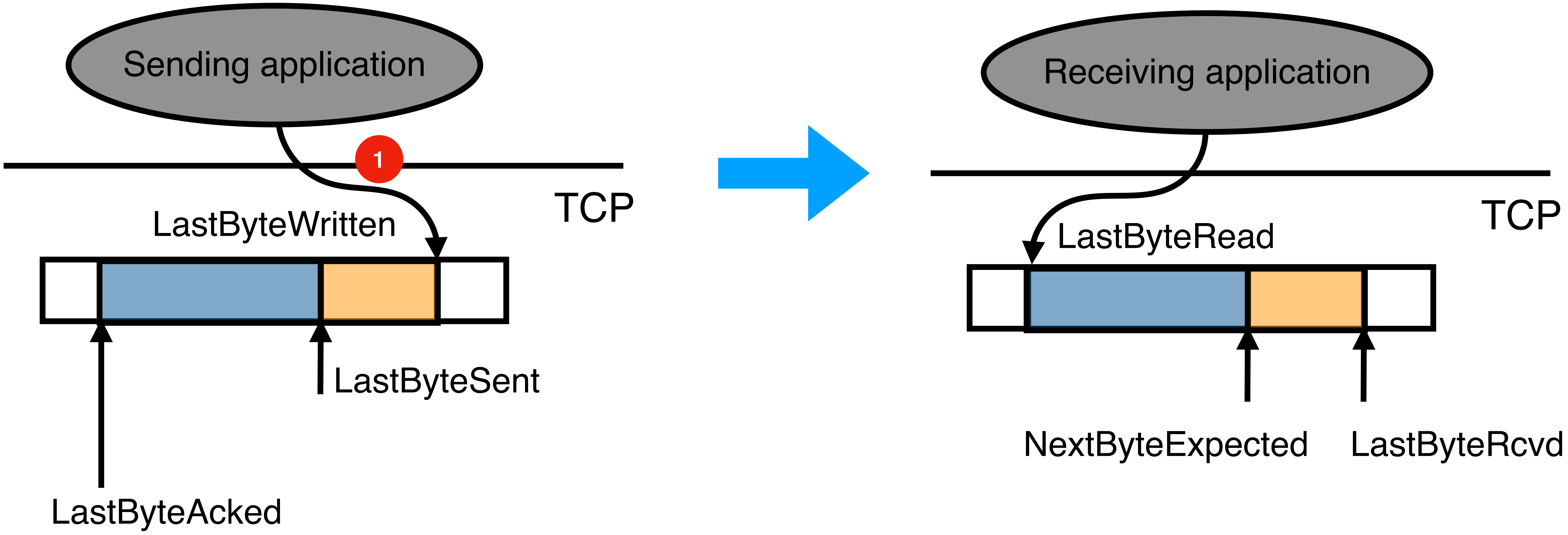
- Three variables manipulations:
  - Advance **LastByteRead** when an app reads
  - Advance **LastByteRcvd** when the segment is received
  - Advance **NextByteExpected** when the next expected segment is received

- Invariants:
  - **LastByteRead** < **NextByteExpected**
  - **NextByteExpected** <= **LastByteRcvd** + 1

- Buffered bytes:
  - **|LastByteRcvd - LastByteRead| <= MaxRcvBuffer**

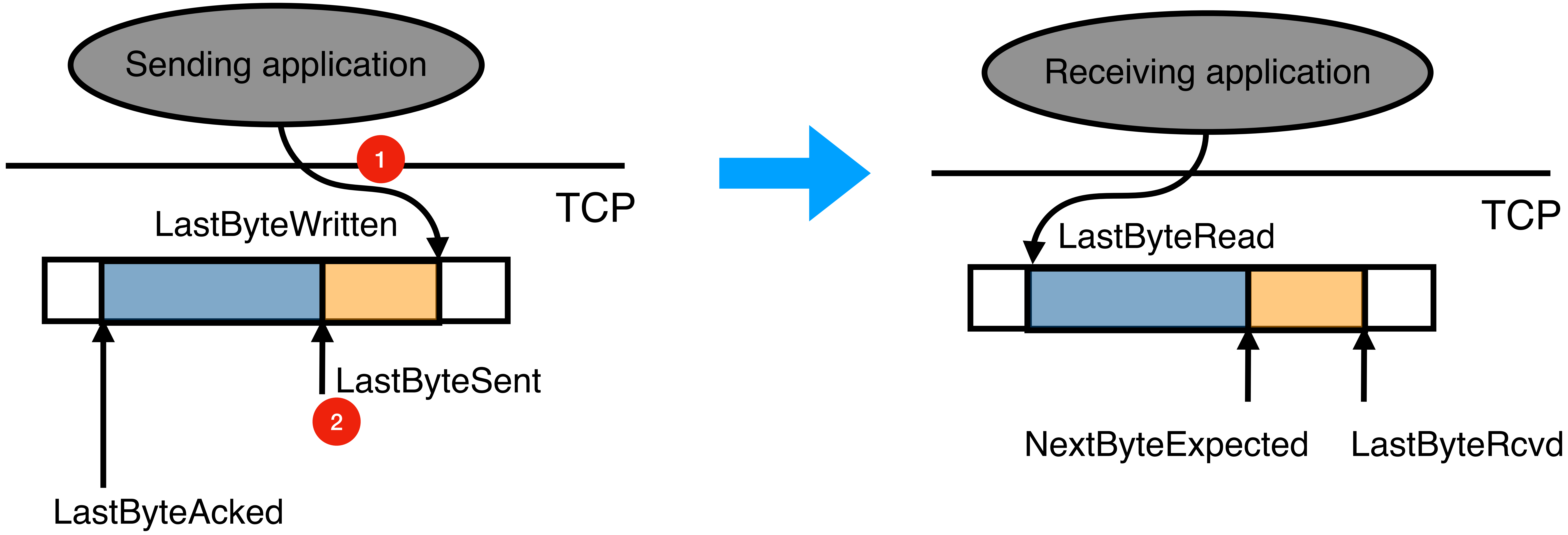
How it works

- Step #1: The sending application writes data to the send buffer
  - **LastByteWritten += sizeof (written data)**



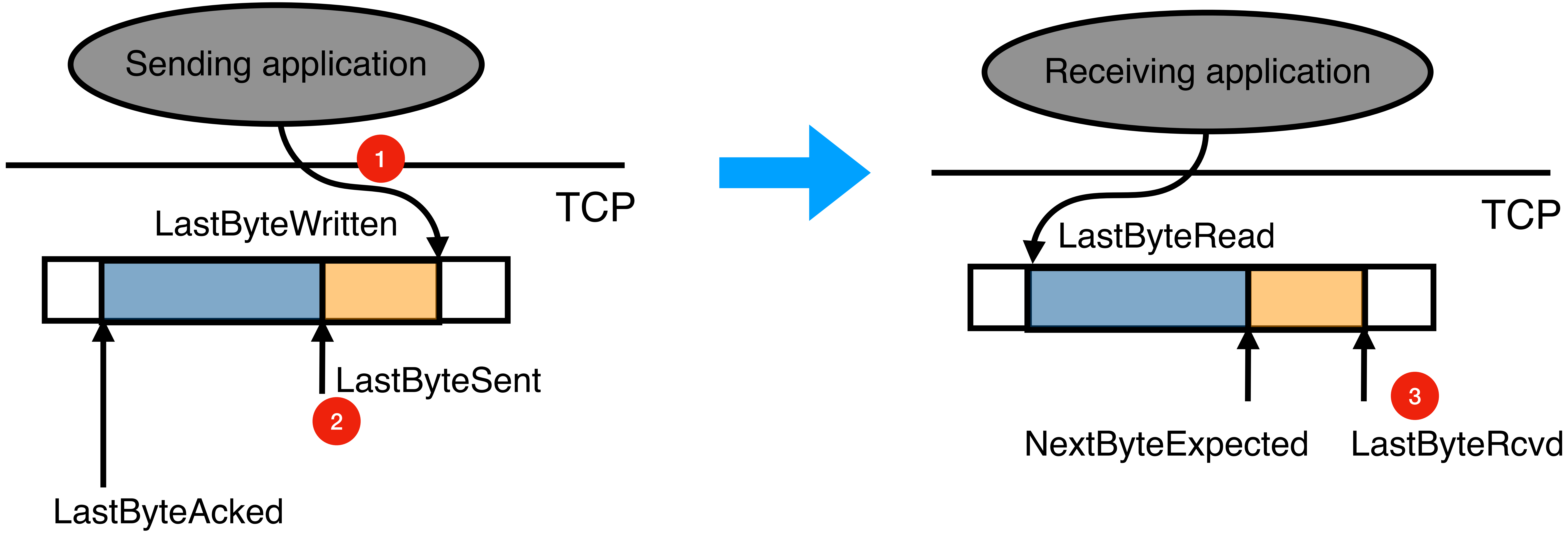
How it works

- Step #2: The buffered data is sent out by OS/NIC
  - **LastByteSent += sizeof (sent data)**



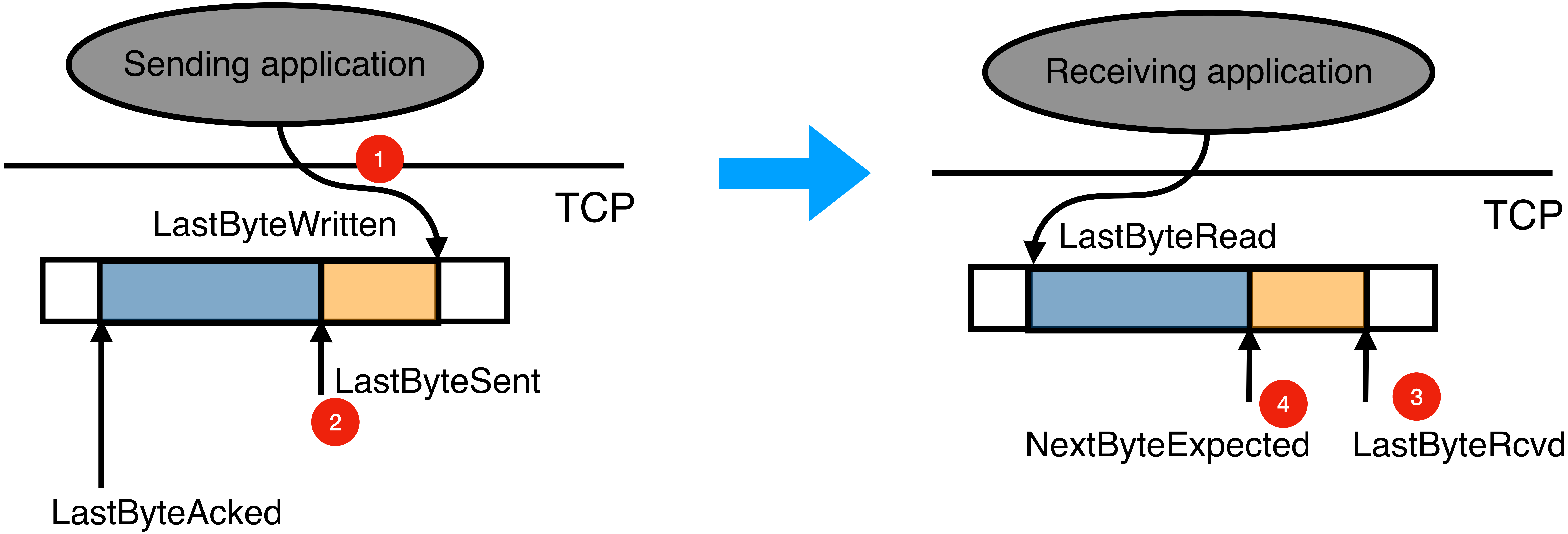
How it works

- Step #3: The data is by the received host and put into the buffer
  - **LastByteRcvd += sizeof (received data)**
  - Advance **NextByteExpected** depending on if there is a hole



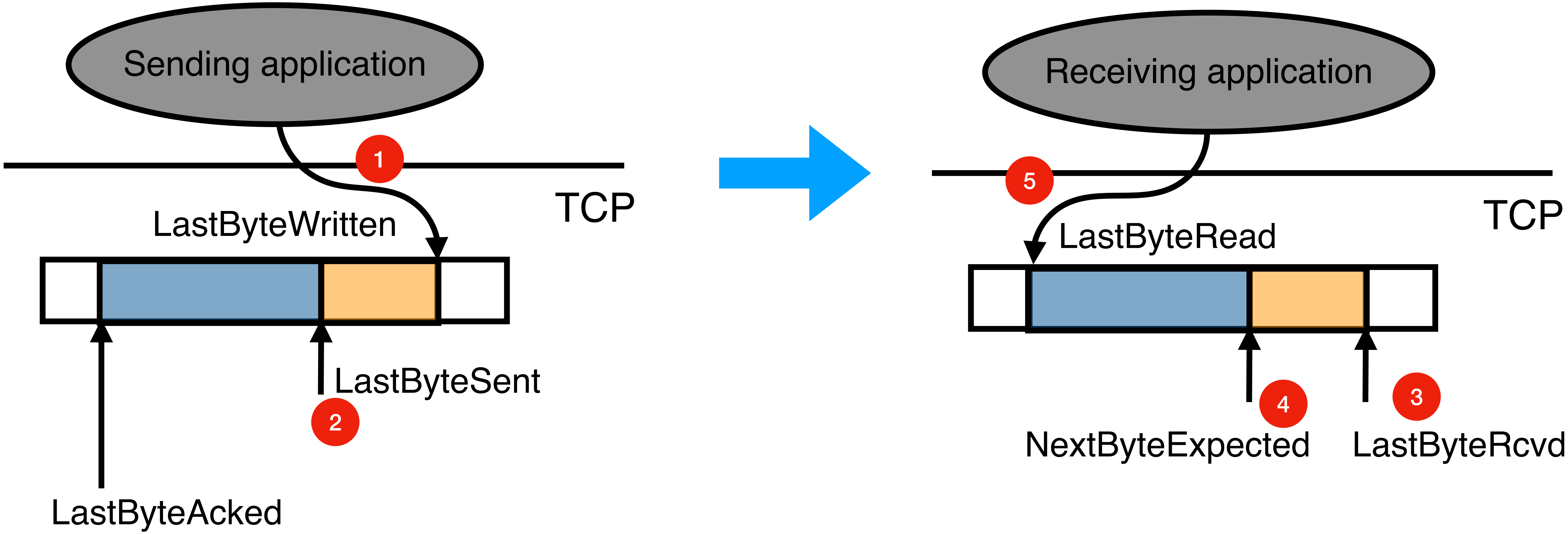
How it works

- Step #4: Received data is sequenced in the buffer
  - Advance **NextByteExpected** when necessary



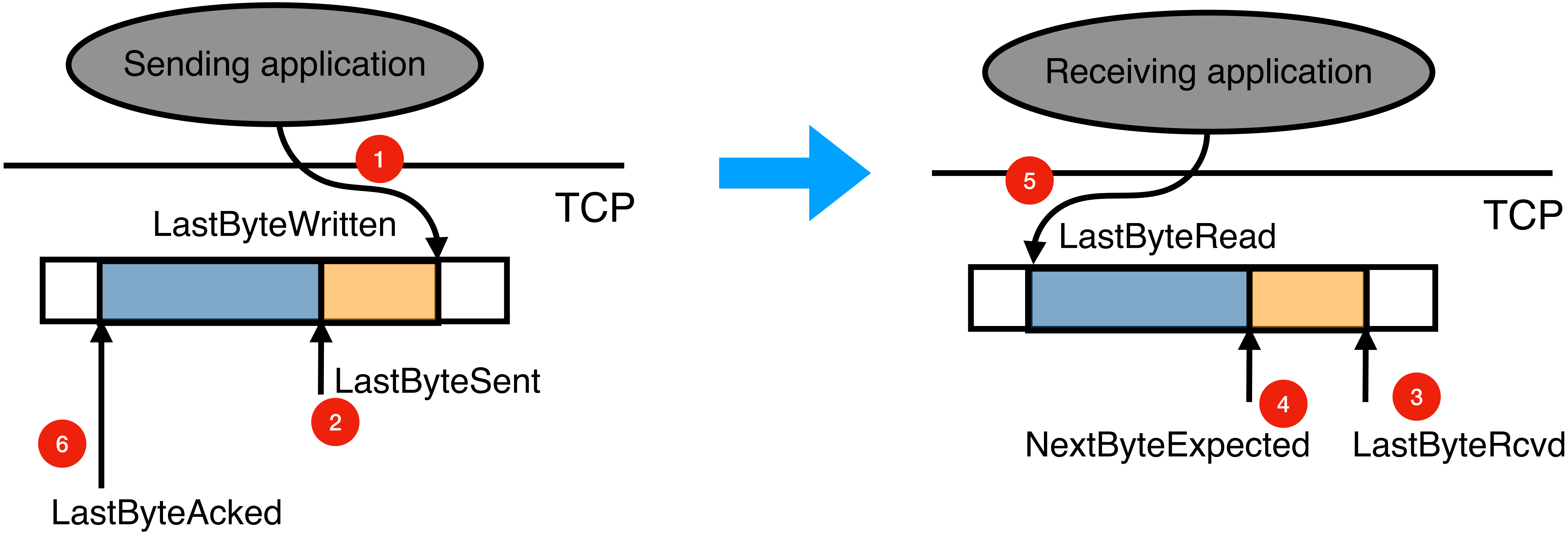
How it works

- Step #5: The receiving application reads data from the buffer
  - **LastByteRead += sizeof (read data)**



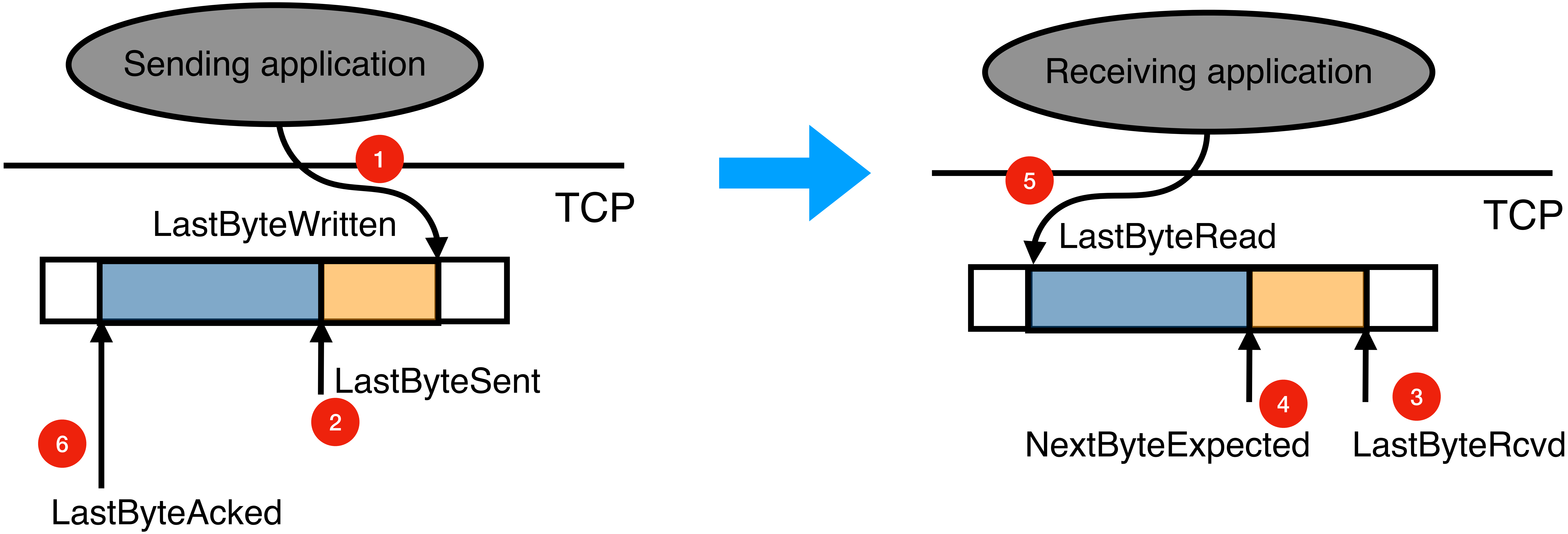
How it works

- Step #6: The receiving application sends ACKs to the sender
  - Advance **LastByteAcked** when necessary



Why Sender Invariants

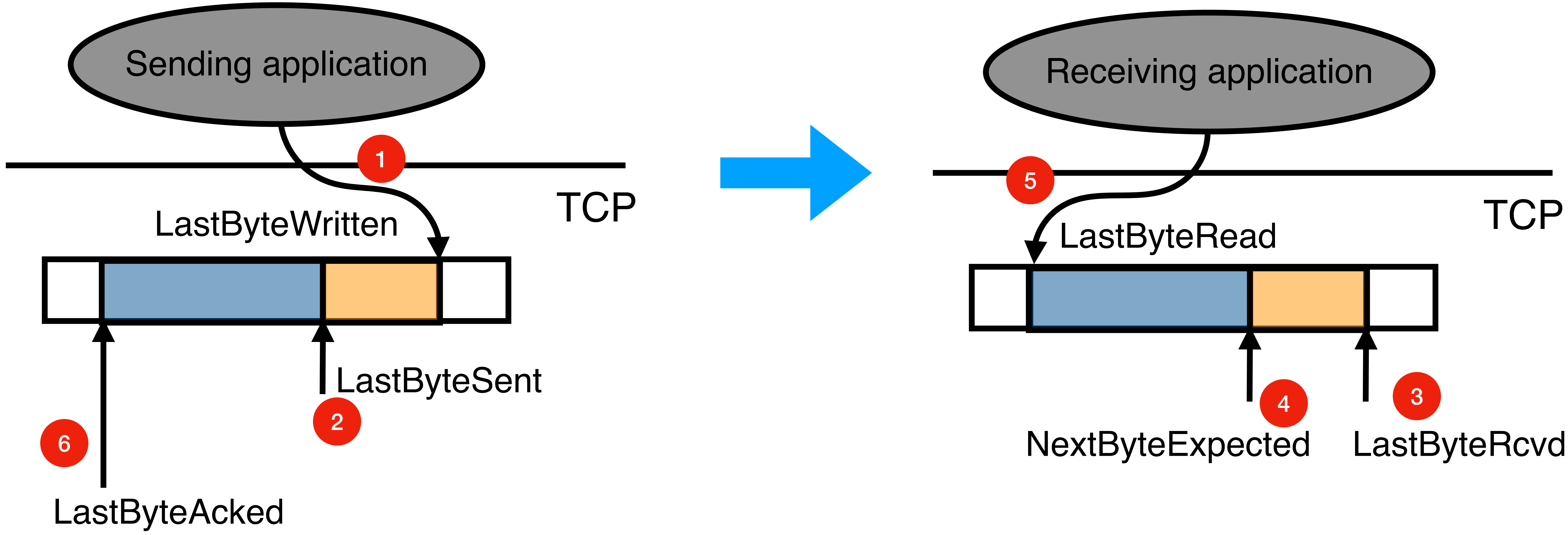
- **LastByteSent** <=**LastByteWritten**
- **LastByteAcked** <= **LastByteSent**



Why Receiver Invariants

- LastByteSent <=LastByteWritten
- LastByteAcked <= LastByteSent

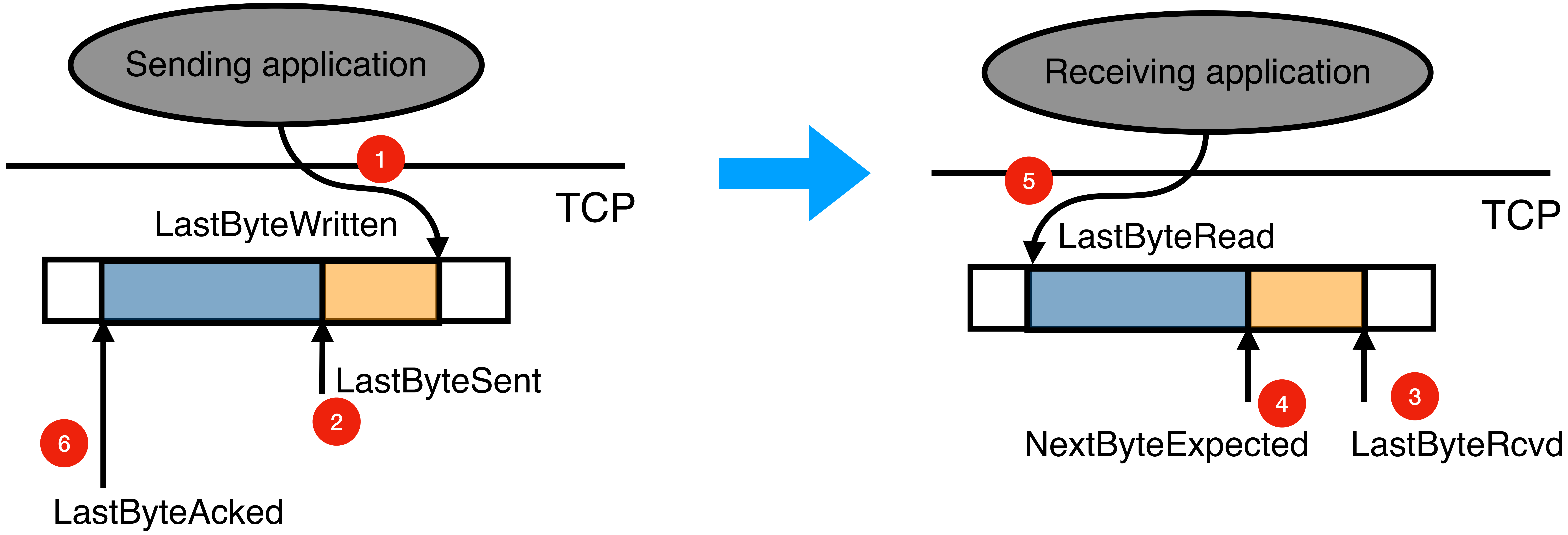
- LastByteRead < NextByteExpected
- NextByteExpected <= LastByteRcvd + 1



Understanding the Sender Buffer

- **LastByteSent** <=**LastByteWritten**
- **LastByteAcked** <= **LastByteSent**

- **LastByteRead** < **NextByteExpected**
- **NextByteExpected** <= **LastByteRcvd** + 1

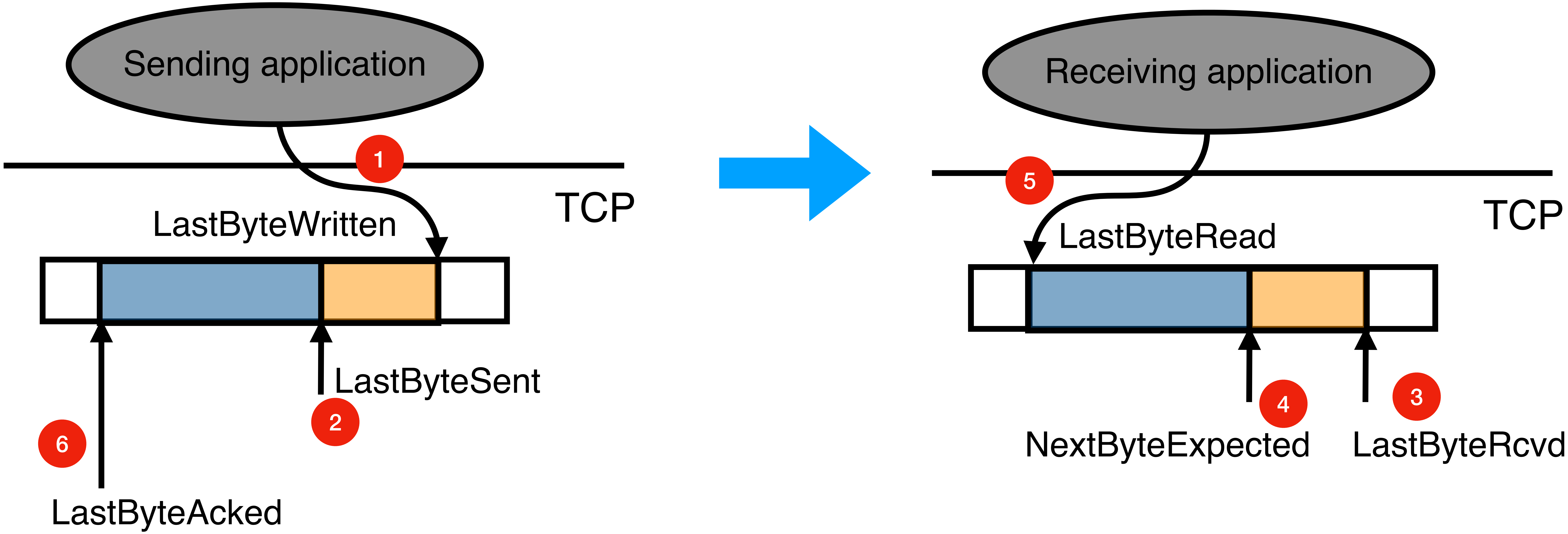


- **|LastByteWritten - LastByteAcked|** <= **MaxSendBuffer**

Understanding the Receiver Buffer

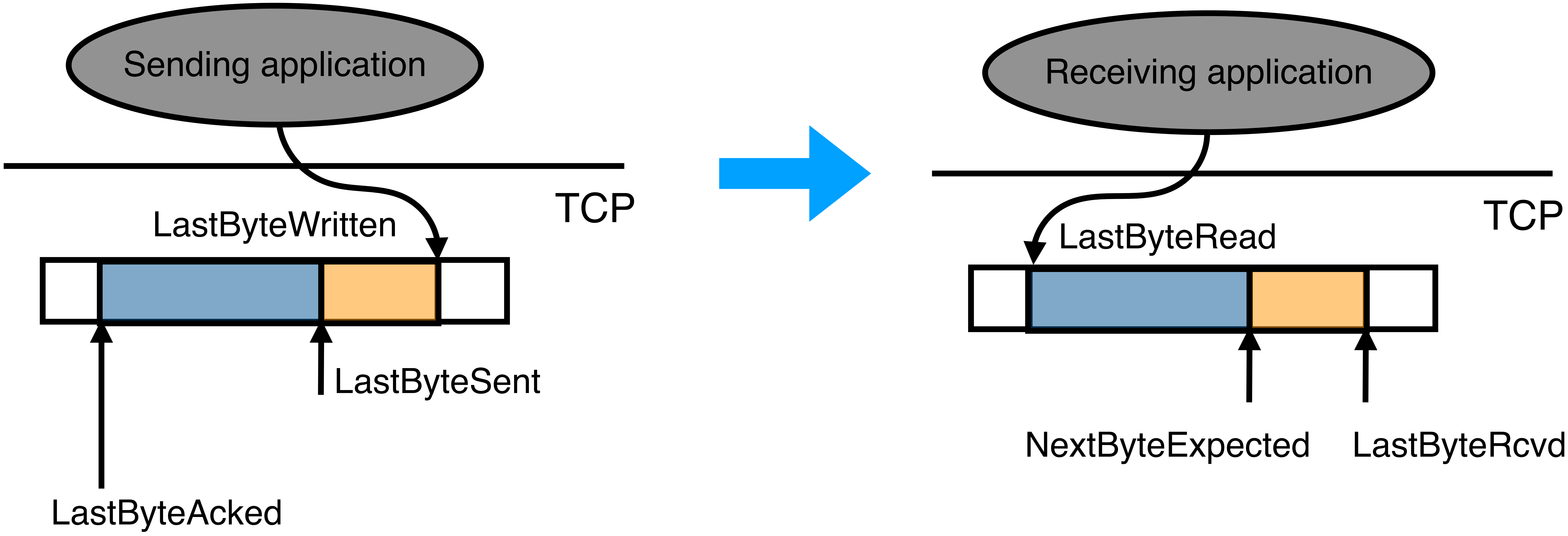
- LastByteSent <=LastByteWritten
- LastByteAcked <= LastByteSent

- LastByteRead < NextByteExpected
- NextByteExpected <= LastByteRcvd + 1



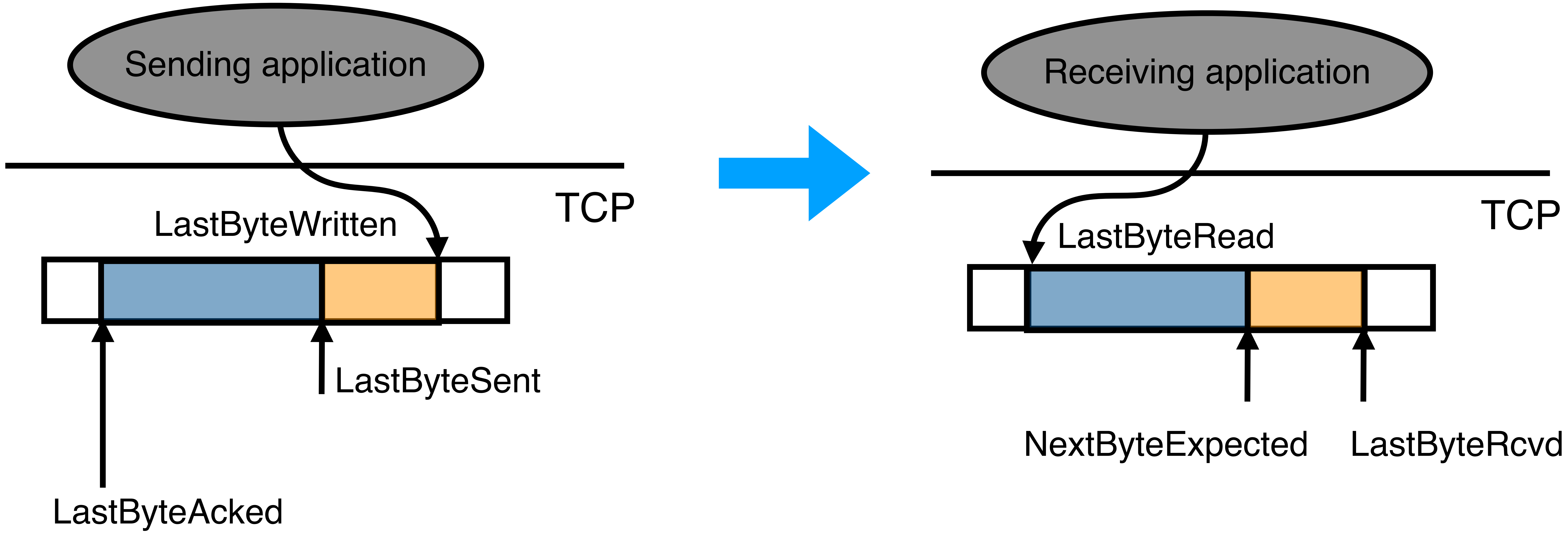
- |LastByteRcvd - LastByteRead| <= MaxRcvBuffer

Tackling Issue #1 (Missing Segment)

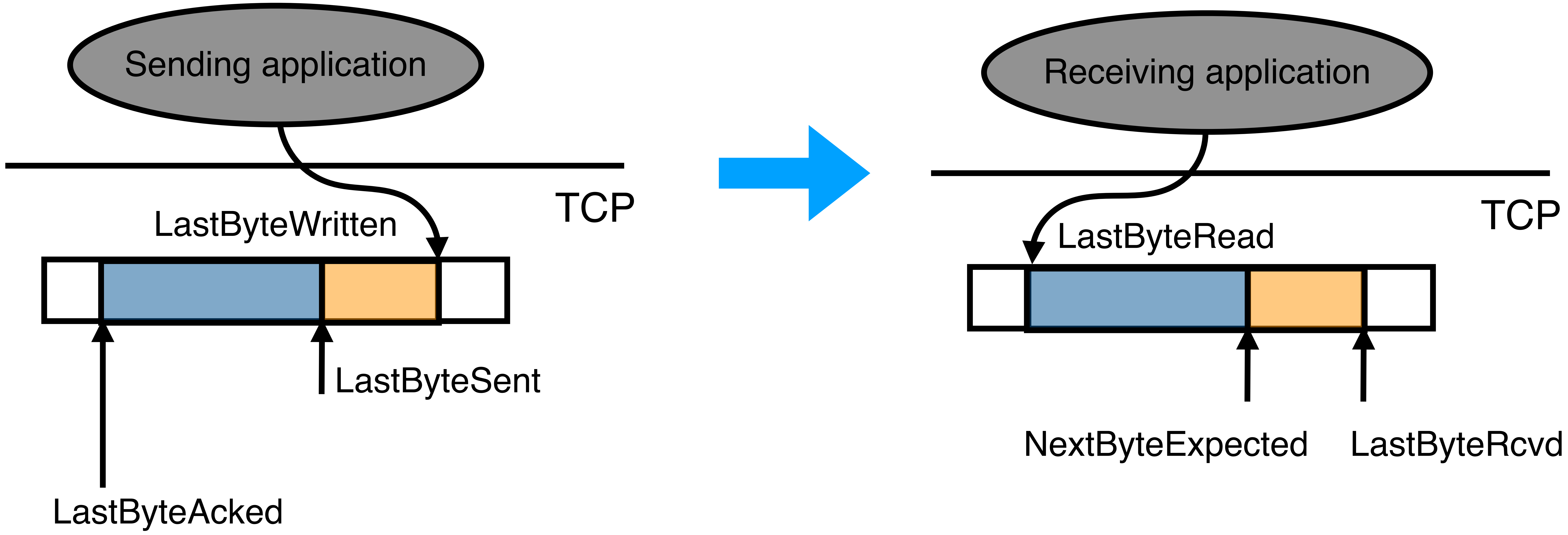


Tackling Issue #1 (Missing Segment)

- Receiver-side detection: [**NextByteExpected**, **LastByteRcvd**]
- Sender-side detection: [**LastByteAcked**, **LastByteSent**]
- Fix: buffered bytes are only freed before **LastByteAcked**

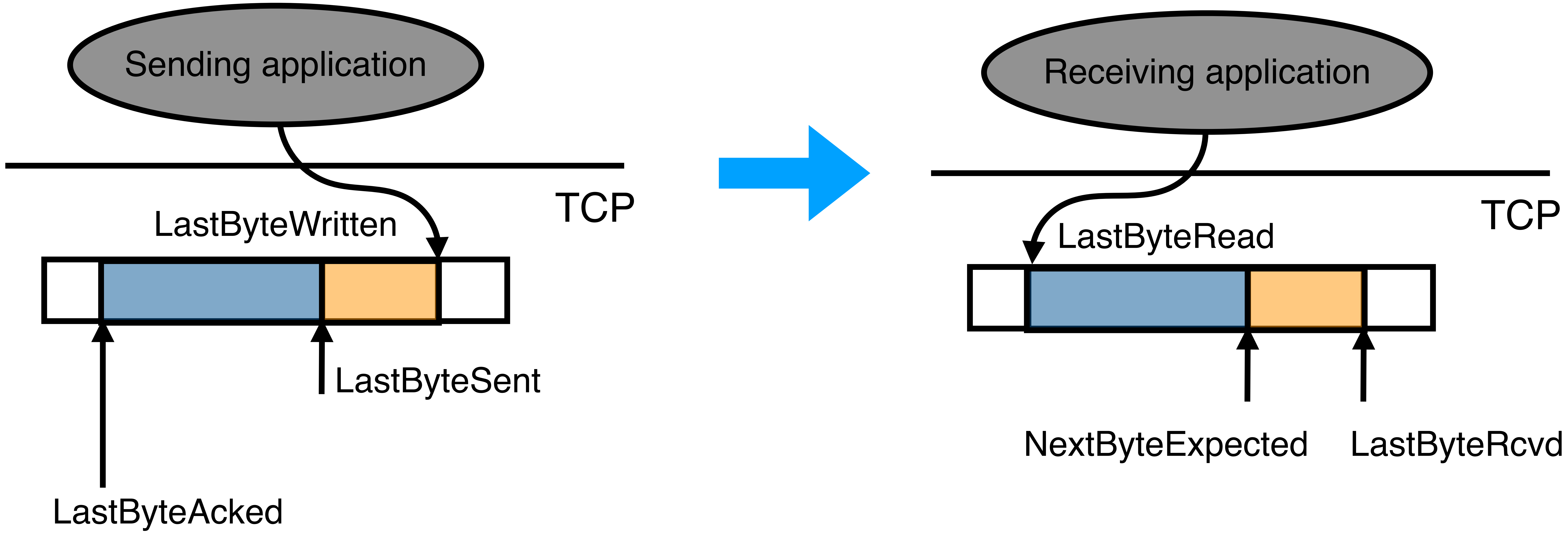


Tackling Issue #2 (Duplicated Segment)

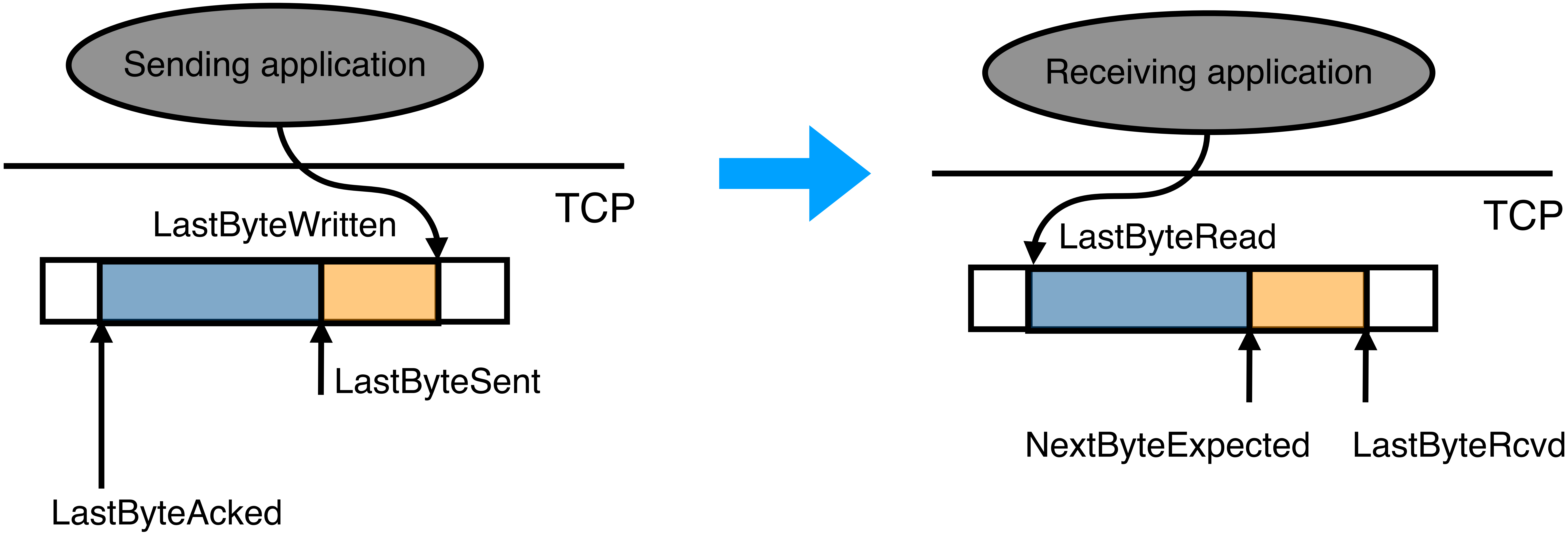


Tackling Issue #2 (Duplicated Segment)

- Detection: [**LastByteRead**, **NextByteExpected**]
- Fix: drop

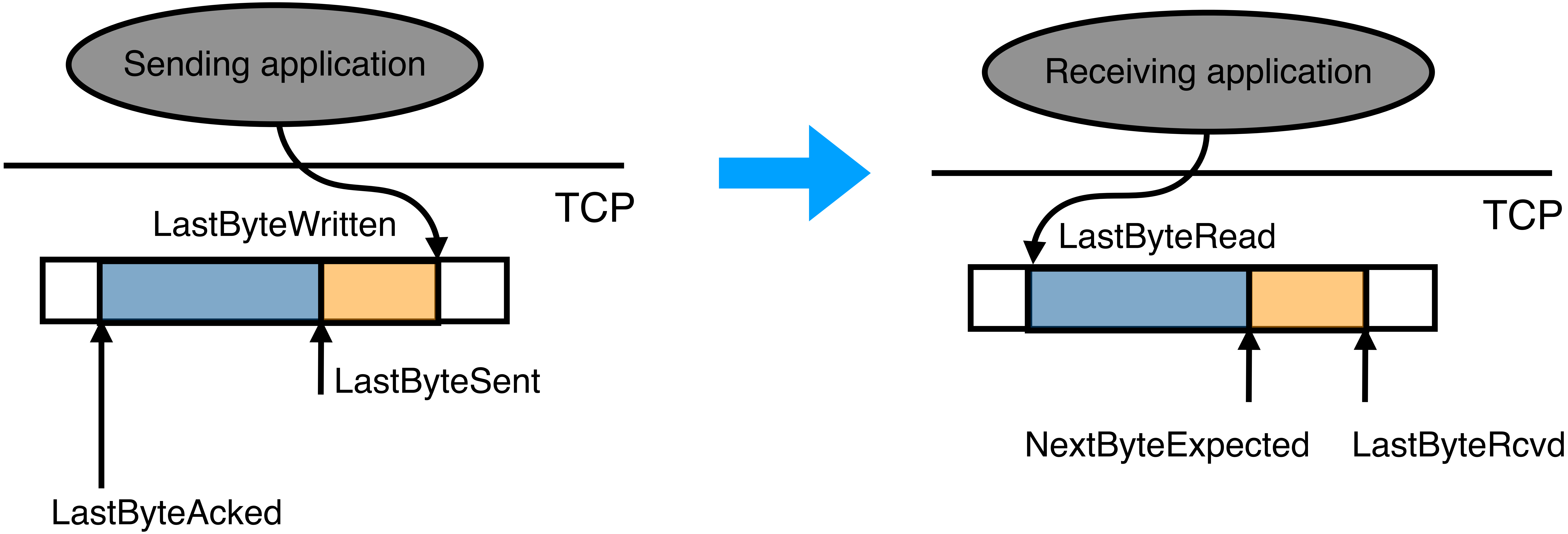


Tackling Issue #3 (Out-of-order Segment)

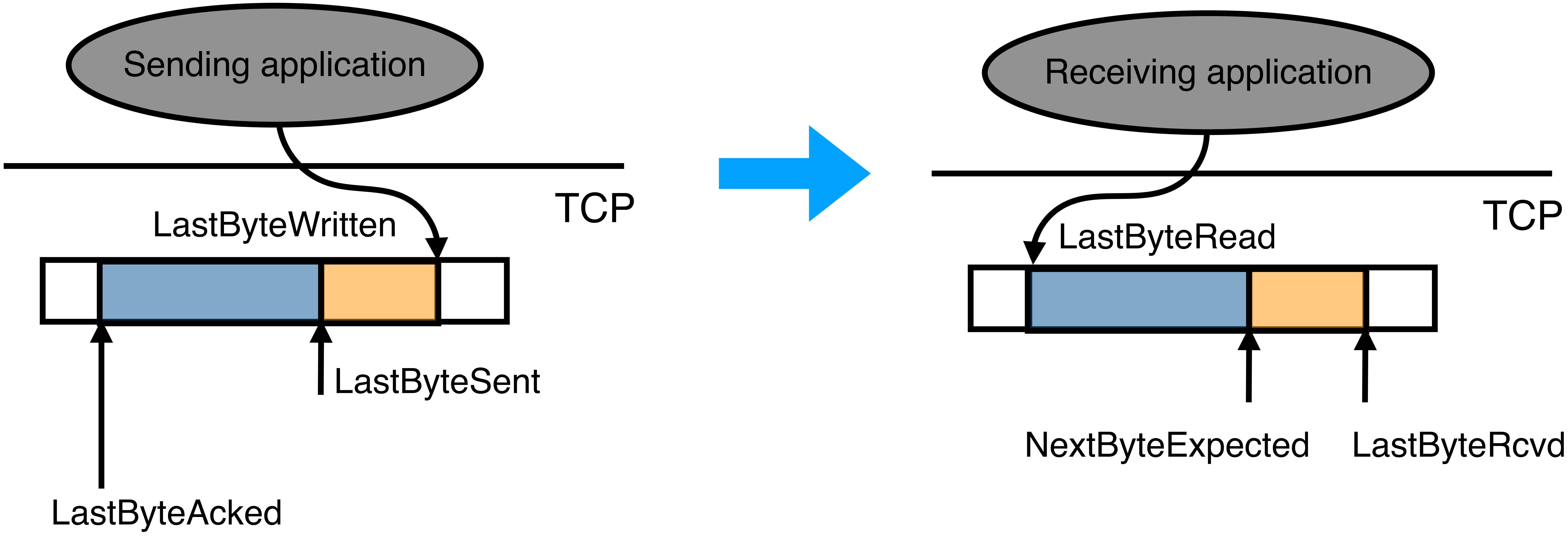


Tackling Issue #3 (Out-of-order Segment)

- Detection: [**NextByteExpected**, **LastByteRcvd**]
- Fix: take if **|LastByteRcvd - LastByteRead| <= MaxRcvBuffer**

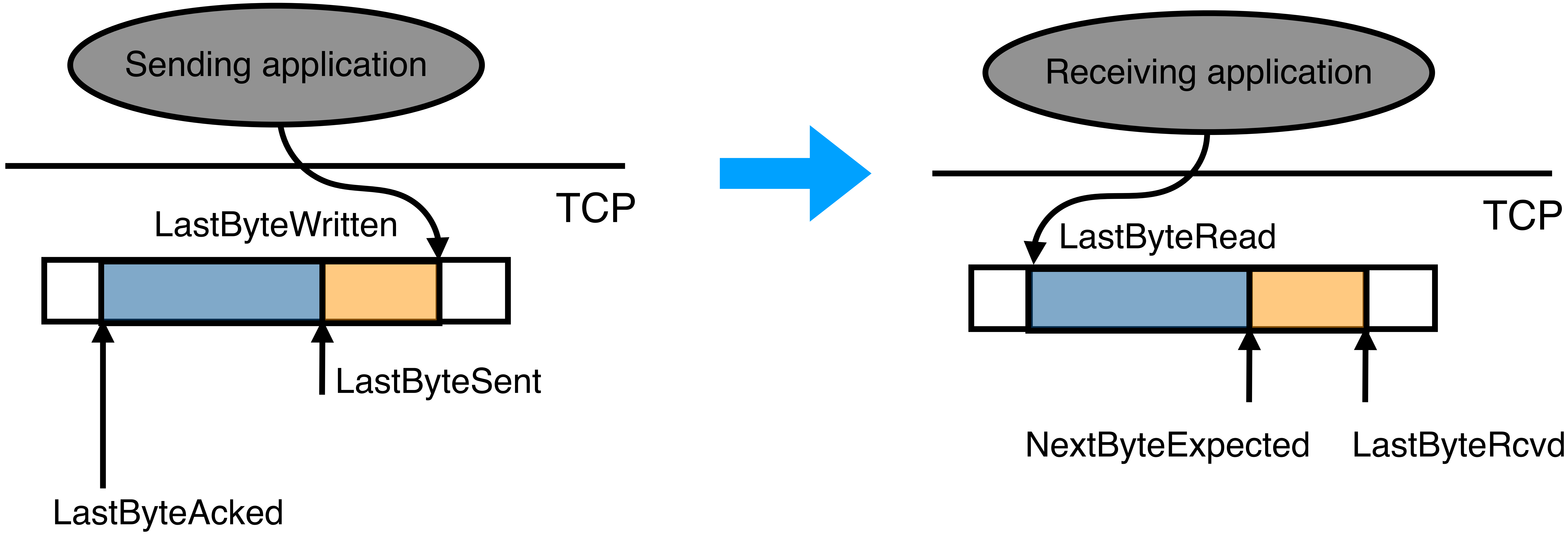


Tackling Issue #4 (Receiver Overwhelming)



Tackling Issue #4 (Receiver Overwhelming)

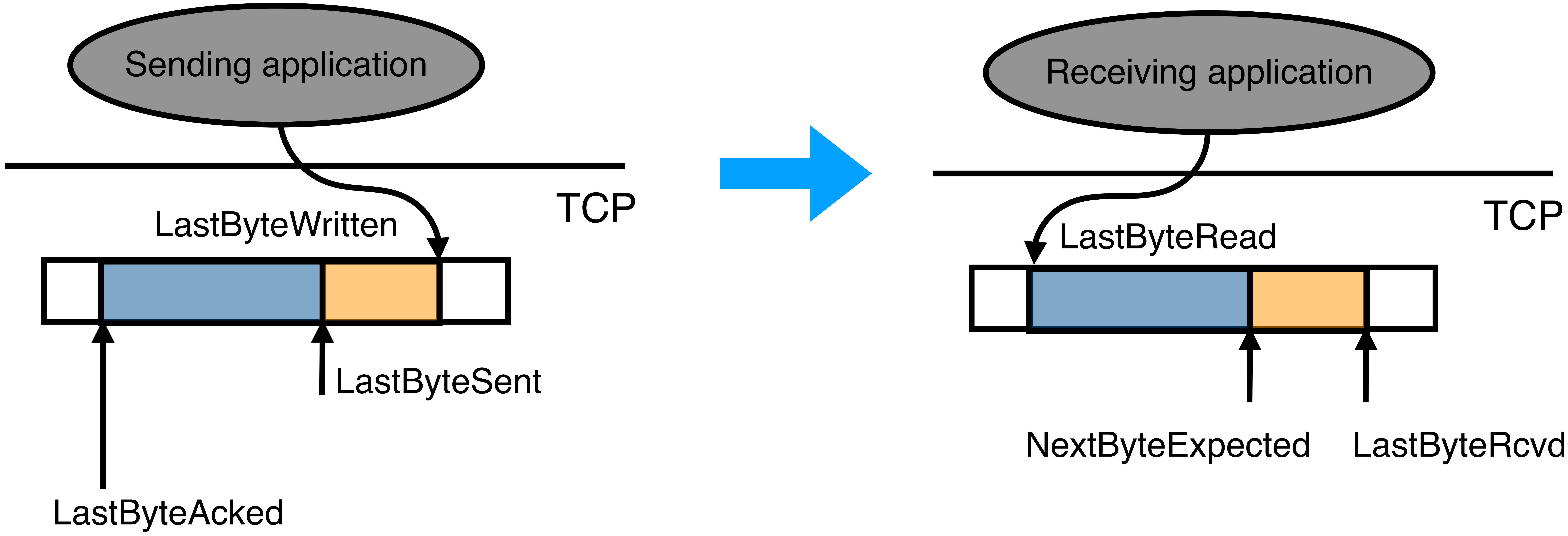
- Detection: **lLastByteRcvd - LastByteReadl <= MaxRcvBuffer**
- Fix: tell the sender the available space (**AdvertisedWindow**)



Tackling Issue #4 (Receiver Overwhelming)

- Detection: **lLastByteRcvd - LastByteReadl <= MaxRcvBuffer**
- Fix: tell the sender the available space (**AdvertisedWindow**)

AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - LastByteRead)

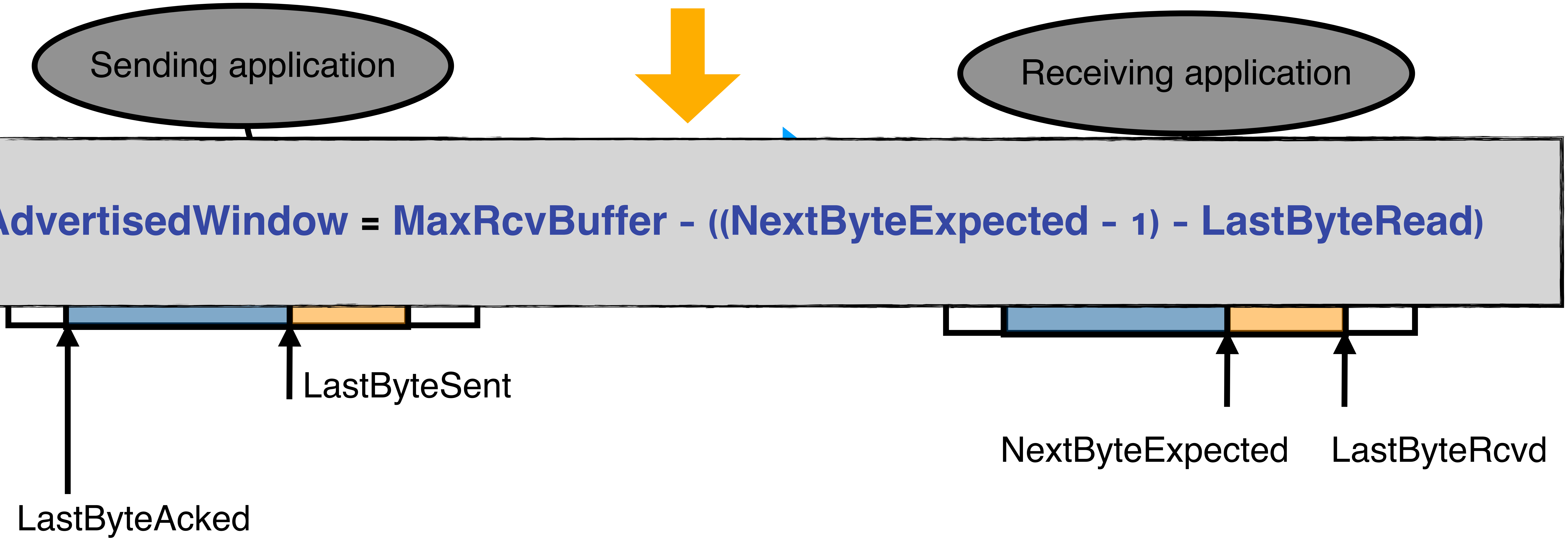


Tackling Issue #4 (Receiver Overwhelming)

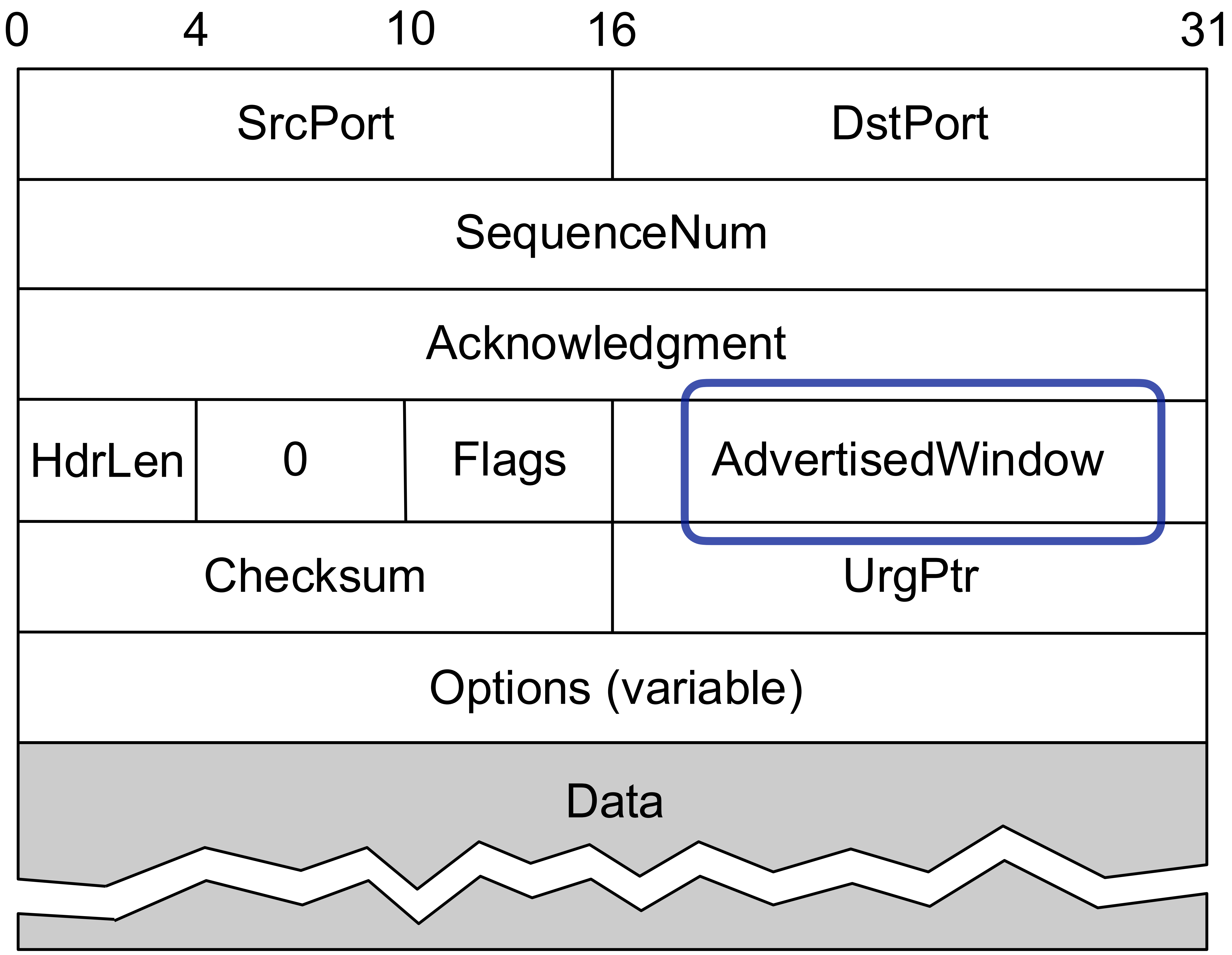
- Detection: **|LastByteRcvd - LastByteRead| <= MaxRcvBuffer**
- Fix: tell the sender the available space (**AdvertisedWindow**)

AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - LastByteRead)

AdvertisedWindow = MaxRcvBuffer - ((NextByteExpected - 1) - LastByteRead)



Revisiting TCP Header



0 4 10 16 31

SrcPort DstPort

SequenceNum

Acknowledgment

HdrLen 0 Flags AdvertisedWindow

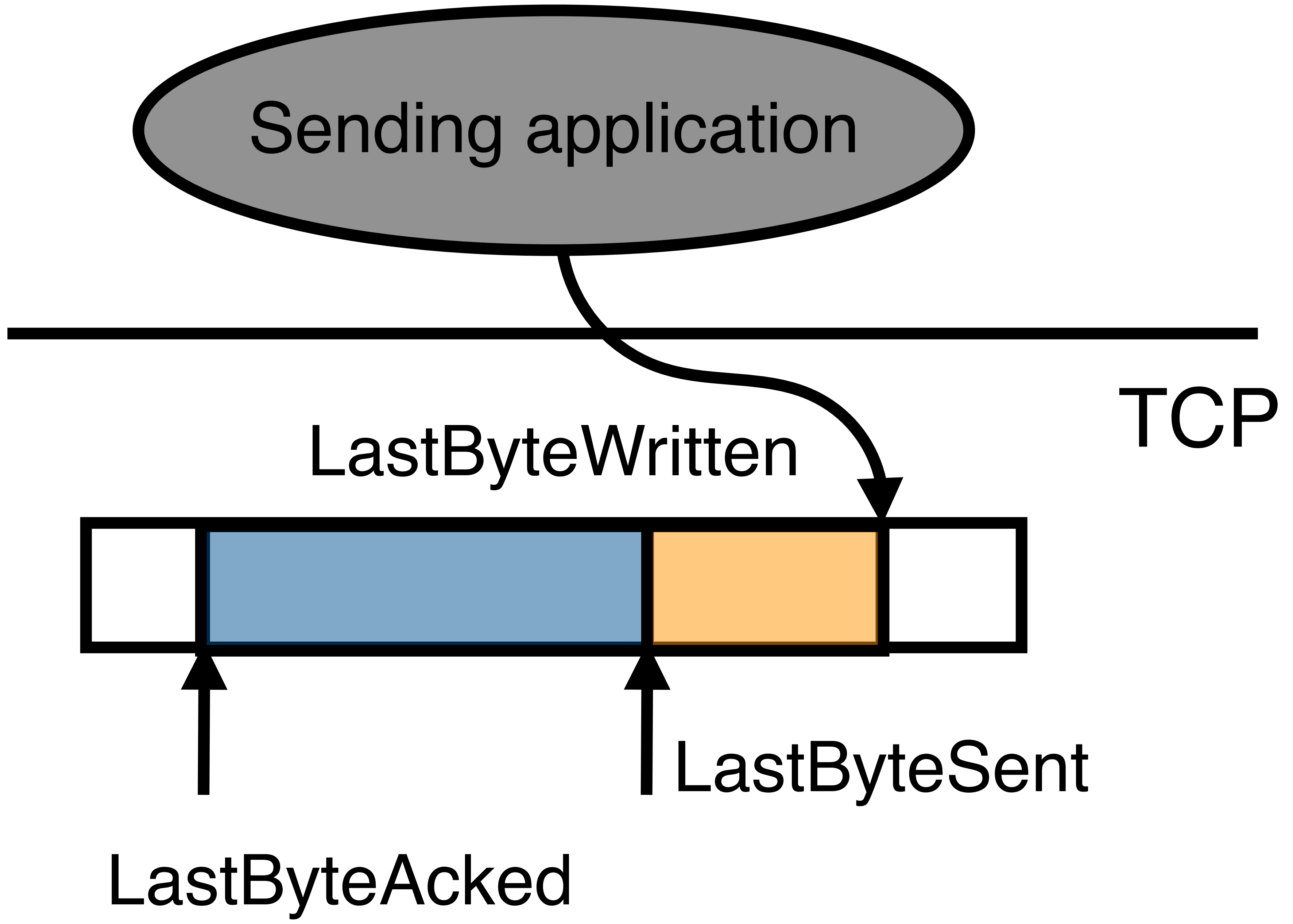
Checksum UrgPtr

Options (variable)

Data

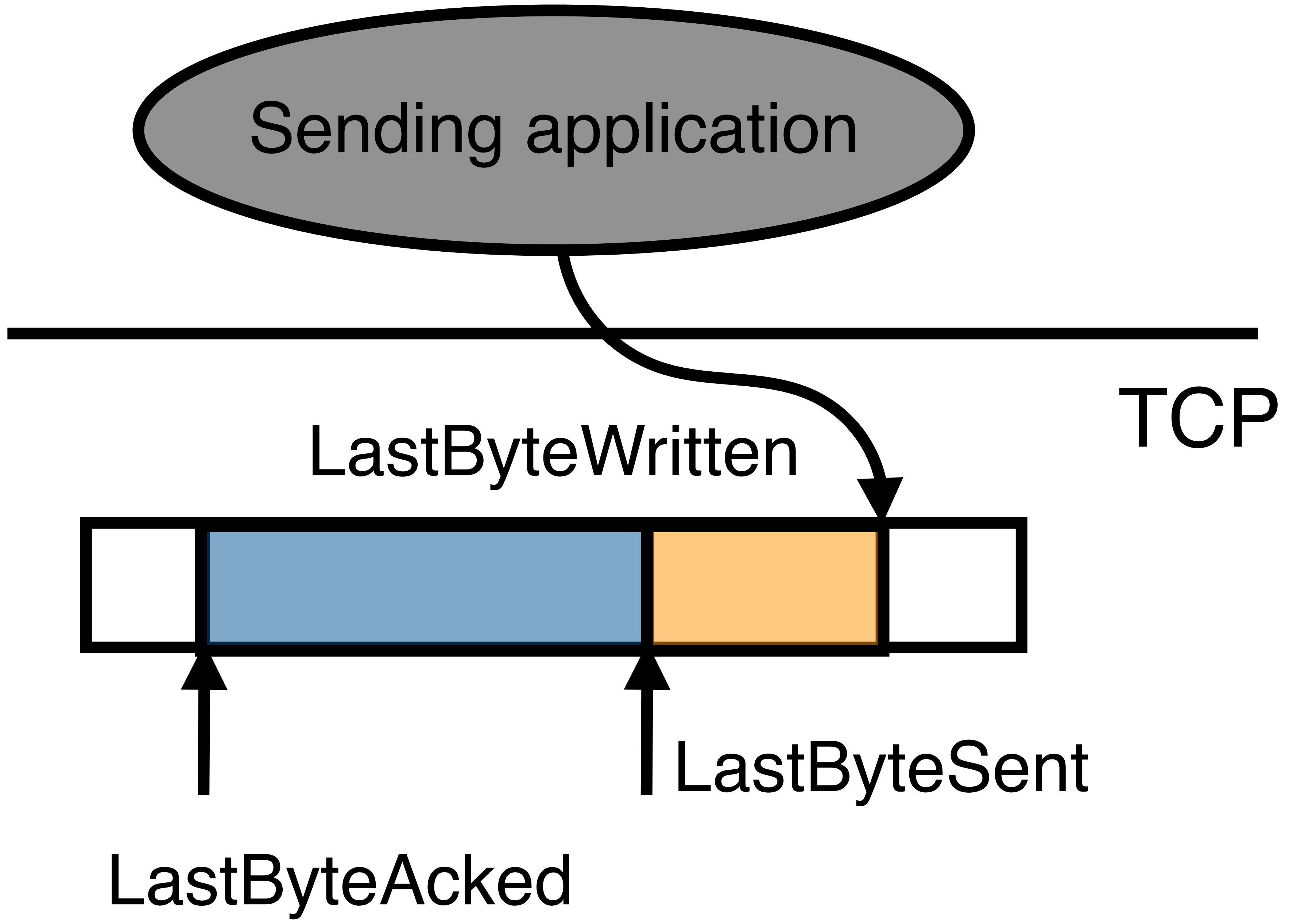
TCP Flow Control

- The sender controls the transmission rate
  - LastByteSent - LastByteAcked <= AdvertisedWindow
  - EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)



TCP Flow Control Affects Application Performance

- The application speed is throttled
  - LastByteWritten - LastByteAcked <= MaxSendBuffer
  - Block sender if (LastByteWritten - LastByteAcked) + y > MaxSendBuffer



Flow Control More

- The receiver
  - Always send ACKs in response to arriving data segments

- The sender
  - Persistent sending at least one byte when AdvertisedWindow = 0

How does TCP solve the second issue?

- #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

Summary

- Today
  - TCP reliability support (II)

- Next lecture
  - TCP congestion control