# CS162
# Operating Systems and Systems Programming
# Lecture 17
# TCP, Flow Control, Reliability

April 3, 2013

Anthony D. Joseph

http://inst.eecs.berkeley.edu/~cs162

# Quiz 16.2: Layering

- Q1: True _  False _  Layering improves application performance
- Q2: True _  False _  Routers forward a packet based on its destination address
- Q3: True _  False _  “Best Effort” packet delivery ensures that packets are delivered in order
- Q4: True _  False _  Port numbers belong to network layer
- Q5: True _  False _  The hosts on Berkeley’s campus share the same IP address prefix

# Quiz 16.2: Layering

- Q1: True _  False **X**  Layering improves application performance
- Q2: True **X**  False _  Routers forward a packet based on its destination address
- Q3: True _  False **x**  "Best Effort" packet delivery ensures that packets are delivered in order
- Q4: True _  False **x**  Port numbers belong to network layer
- Q5: True **X**  False _  The hosts on Berkeley's campus share the same IP address prefix

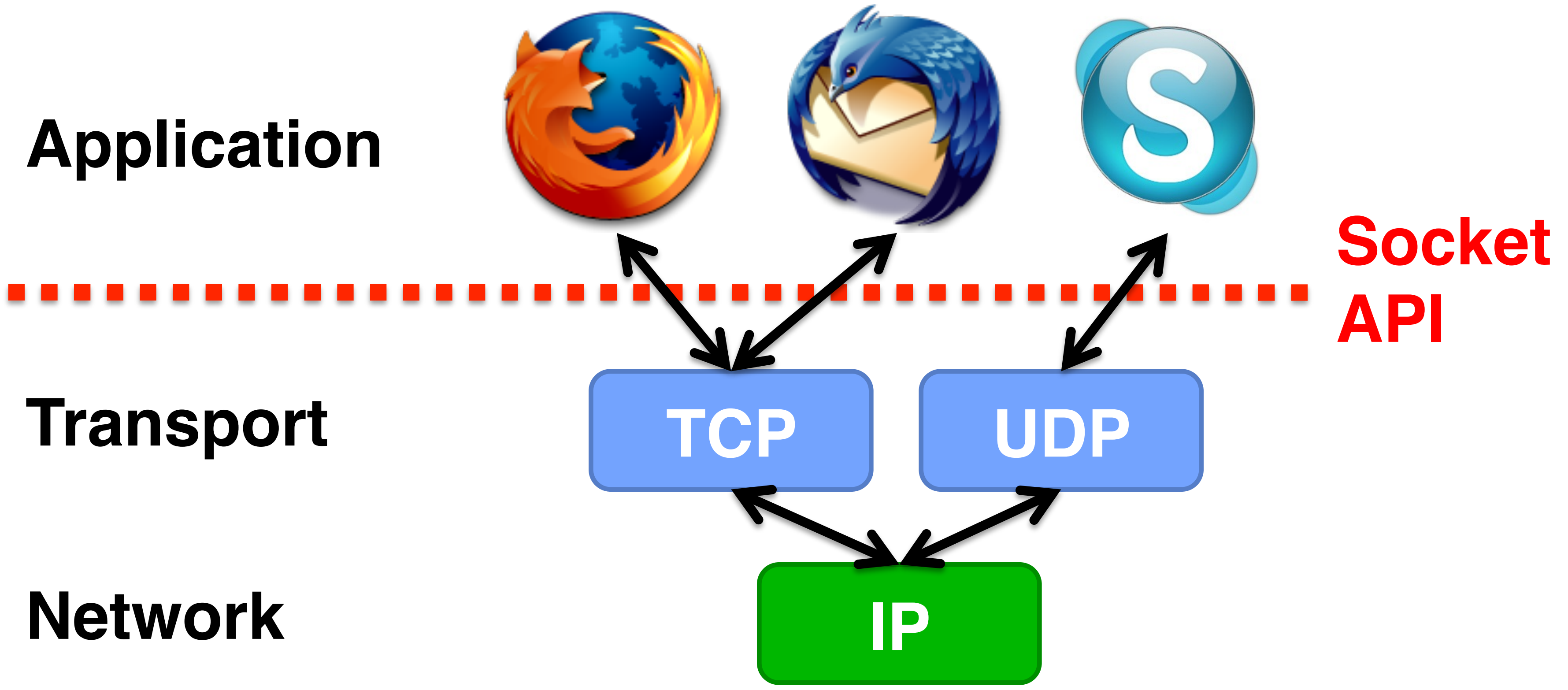
# Goals for Today

- Socket API

- TCP
  - Open connection (3-way handshake)
  - Reliable transfer
  - Tear-down connection
  - Flow control

# Socket API

- Socket API: Network programming interface



# BSD Socket API

- Created at UC Berkeley (1980s)

- Most popular network API

- Ported to various OSes, various languages
  - Windows Winsock, BSD, OS X, Linux, Solaris, …
  - Socket modules in Java, Python, Perl, …

- Similar to Unix file I/O API
  - In the form of *file descriptor* (sort of handle).
  - Can share same `read()`/`write()`/`close()` system calls

# TCP: Transport Control Protocol

- Reliable, in-order, and at most once delivery
- Stream oriented: messages can be of arbitrary length
- Provides multiplexing/demultiplexing to IP
- Provides congestion and flow control
- Application examples: file transfer, chat

# TCP Service

1) Open  connection: 3-way handshaking

2) Reliable byte stream transfer from (IPa, TCP_Port1) to (IPb, TCP_Port2)
   - Indication if connection fails: Reset

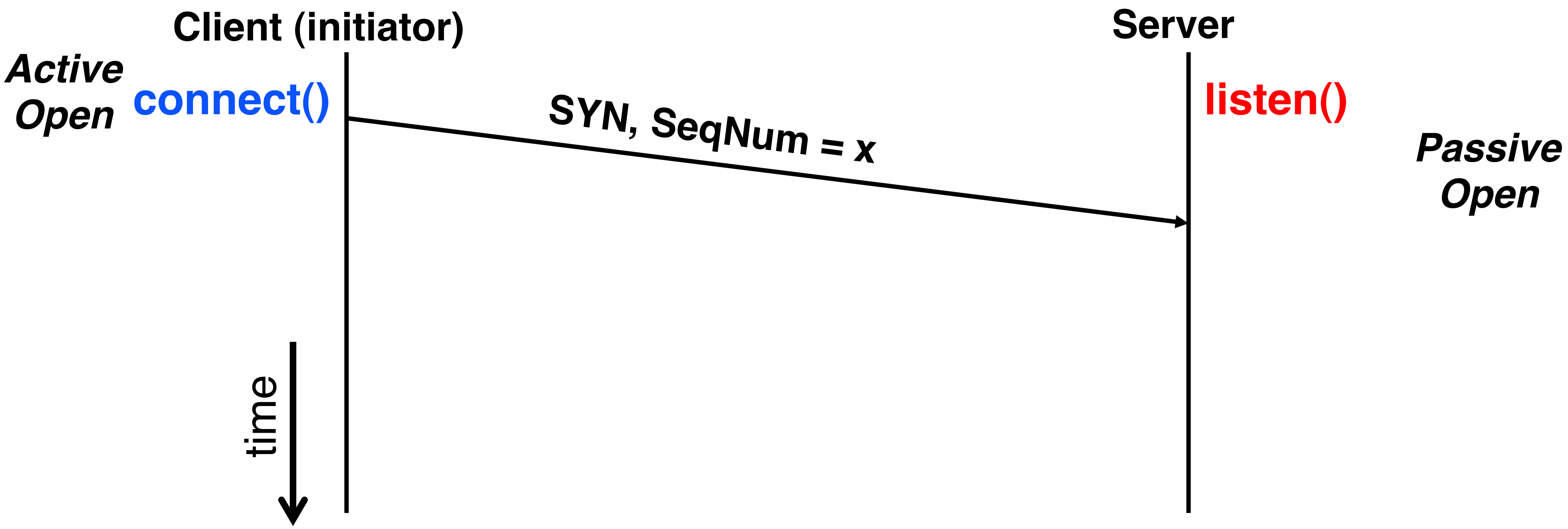
3) Close (tear-down) connection

# Open Connection: 3-Way Handshaking

- Goal: agree on a set of parameters, i.e., the start sequence number for each side
  - Starting sequence number: sequence of first byte in stream
  - Starting sequence numbers are random

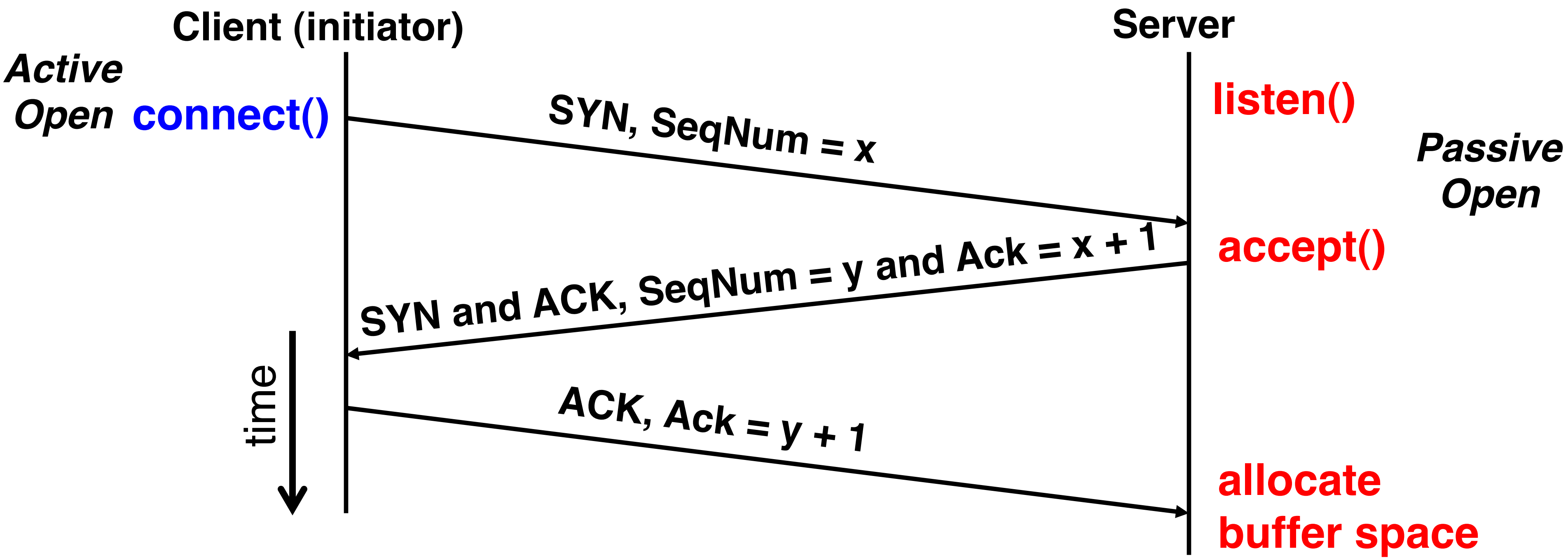
# Open Connection: 3-Way Handshaking

- Server waits for new connection calling listen()
- Sender call connect() passing socket which contains server's IP address and port number
  - OS sends a special packet (SYN) containing a proposal for first sequence number, x



# Open Connection: 3-Way Handshaking

- If it has enough resources, server calls accept() to accept connection, and sends back a SYN ACK packet containing
  - Client's sequence number incremented by one, (x + 1)
    - Why is this needed?
  - A sequence number proposal, y, for first byte server will send



# 3-Way Handshaking (cont'd)

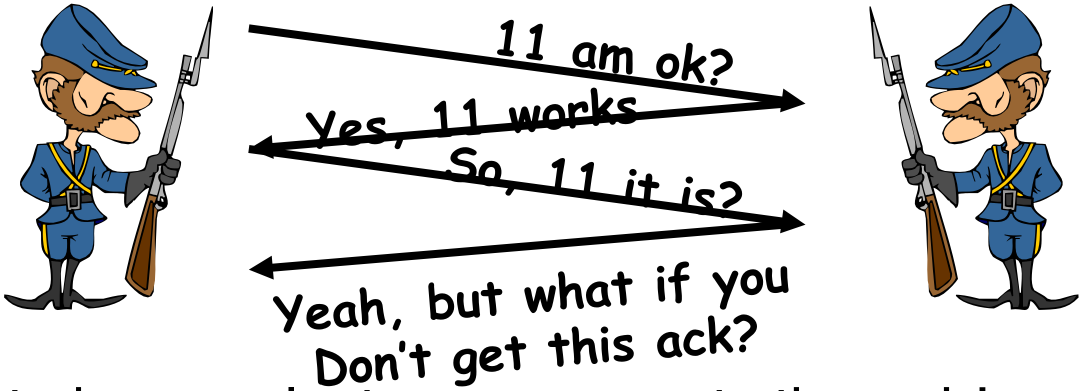
- Three-way handshake adds 1 RTT delay

- Why?
  - – Congestion control: SYN (40 byte) acts as cheap probe
  - – Protects against delayed packets from other connection (would confuse receiver)

# General’s Paradox

- General’s paradox:
  - Constraints of problem:
    - » Two generals, on separate mountains
    - » Can only communicate via messengers
    - » Messengers can be captured
  - Problem: need to coordinate attack
    - » If they attack at different times, they all die
    - » If they attack at same time, they win
  - Named after Custer, who died at Little Big Horn because he arrived a couple of days too early

- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, “no”, even if all messages get through



11 am ok?

Yes, 11 works

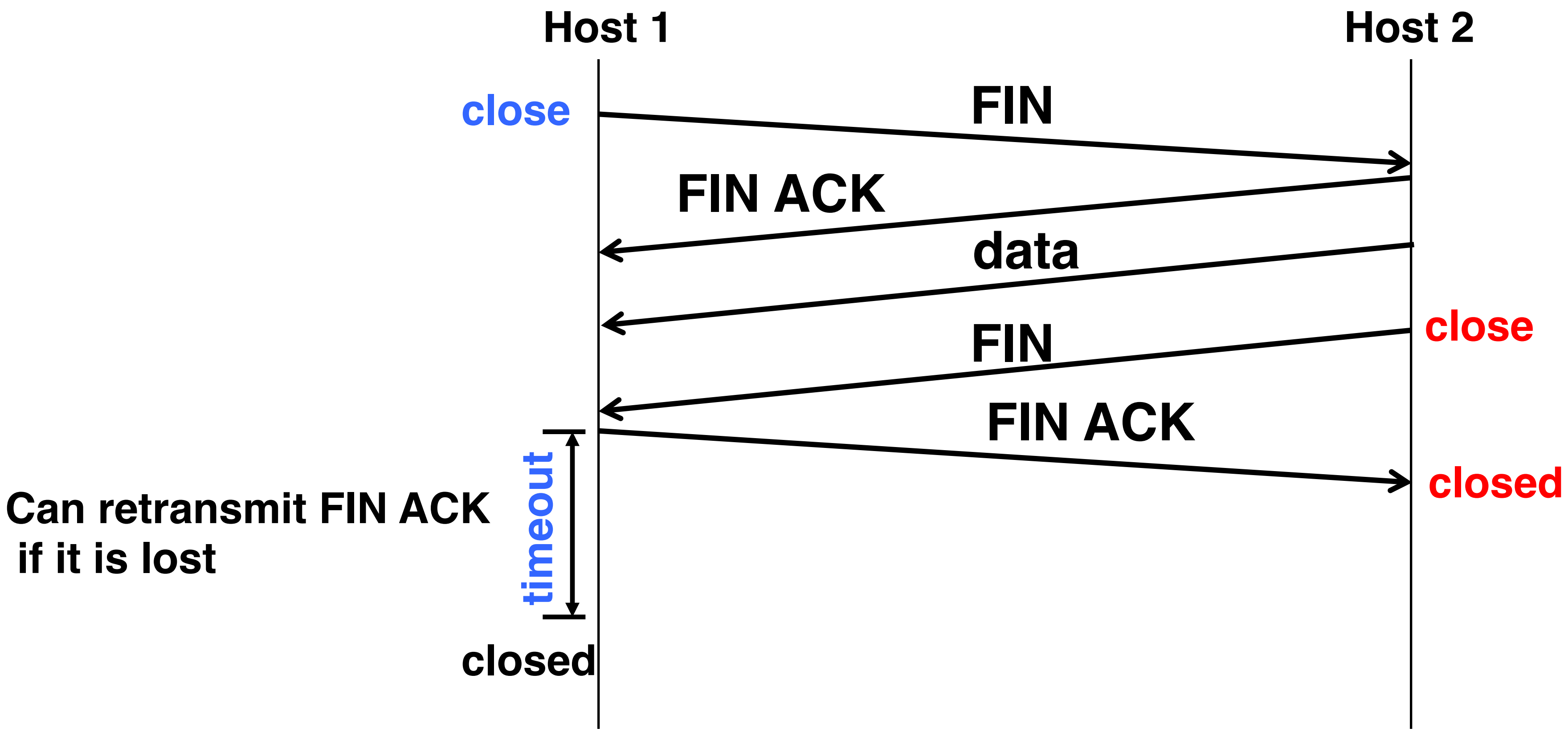
So, 11 it is?

Yeah, but what if you Don't get this ack?

  - No way to be sure last message gets through!

# Close Connection

- Goal: both sides agree to close the connection
- 4-way connection tear down



# Reliable Transfer

- Retransmit missing packets
  - Numbering of packets and ACKs

- Do this efficiently
  - Keep transmitting whenever possible
  - Detect missing packets and retransmit quickly

- Two schemes
  - Stop & Wait
  - Sliding Window (Go-back-n and Selective Repeat)

# Detecting Packet Loss?

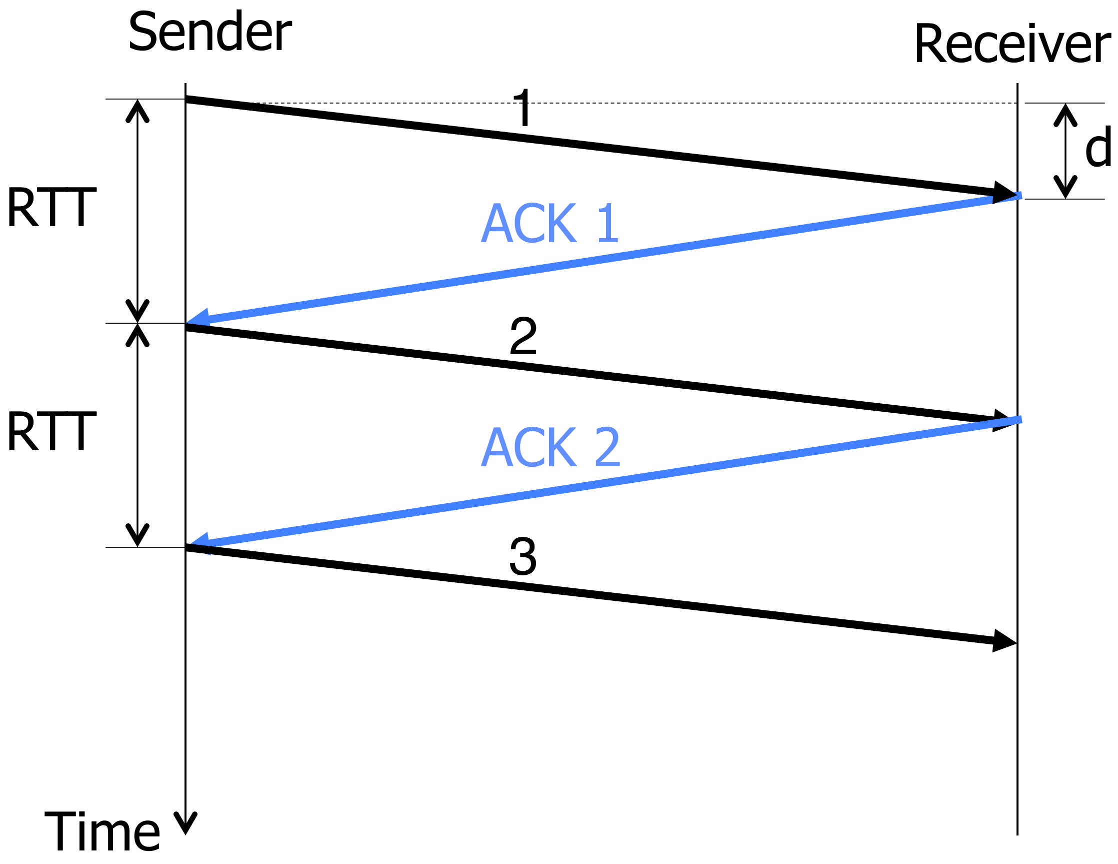
- Timeouts
  - Sender timeouts on not receiving ACK

- Missing ACKs
  - Receiver ACKs each packet
  - Sender detects a missing packet when seeing a gap in the sequence of ACKs
  - Need to be careful! Packets and ACKs might be reordered

- NACK: Negative ACK
  - Receiver sends a NACK specifying a packet it is missing

# Stop & Wait w/o Errors

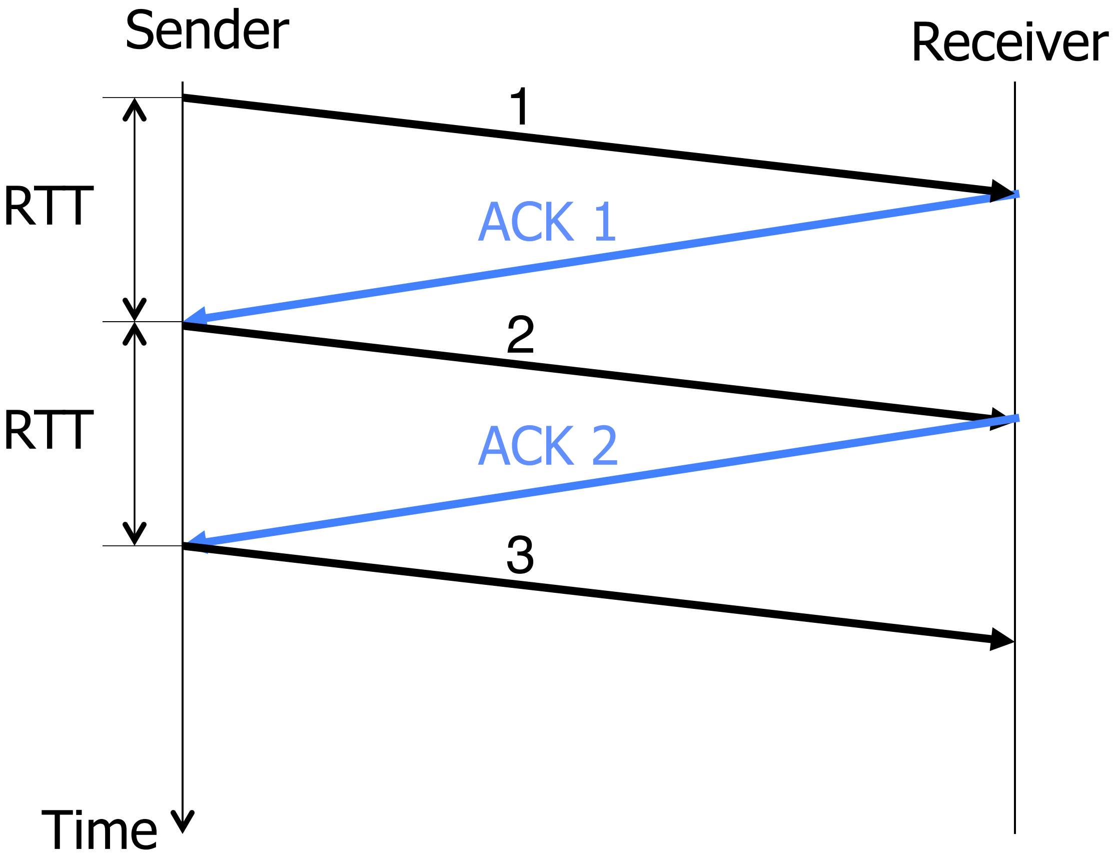
- Send; wait for ack; repeat
- RTT: Round Trip Time (RTT): time it takes a packet to travel from sender to receiver and back
  - One-way latency (d): one way delay from sender and receiver



RTT = 2*d
(if latency is
symmetric)

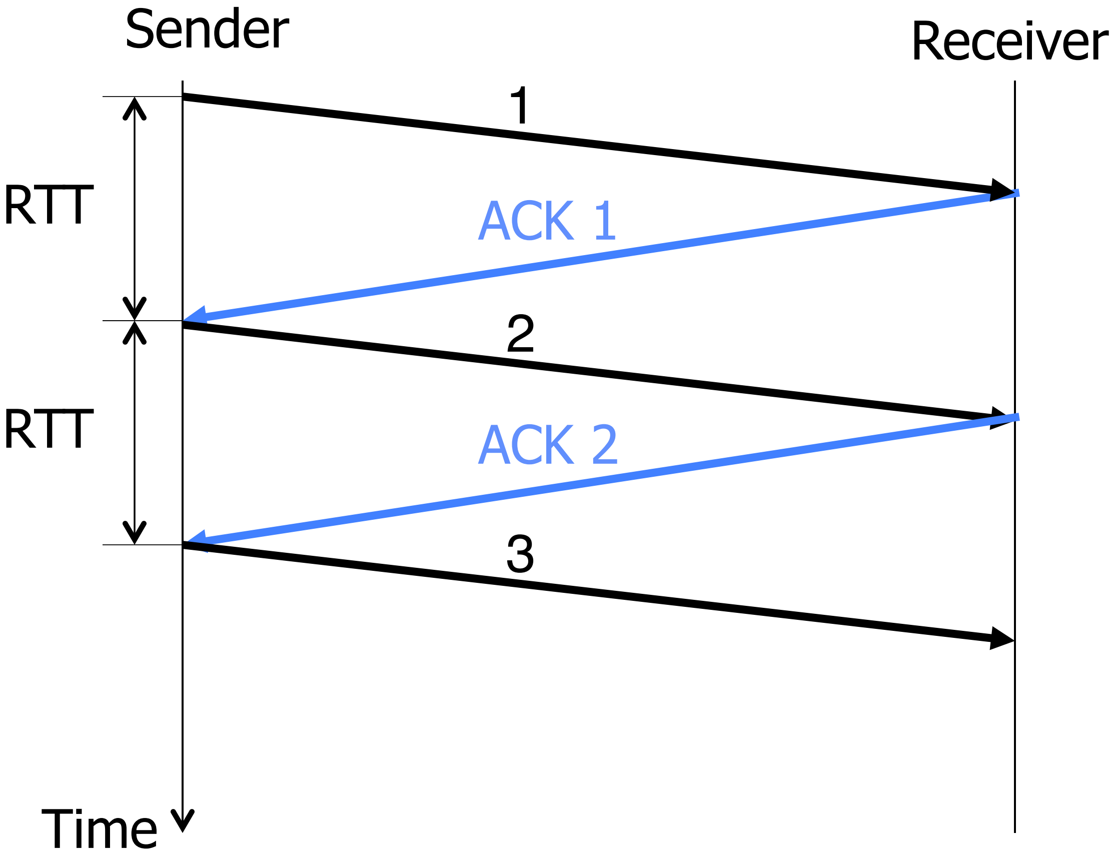
# Stop & Wait w/o Errors

- How many packets can you send?
- 1 packet / RTT
- Throughput: number of bits delivered to receiver per sec



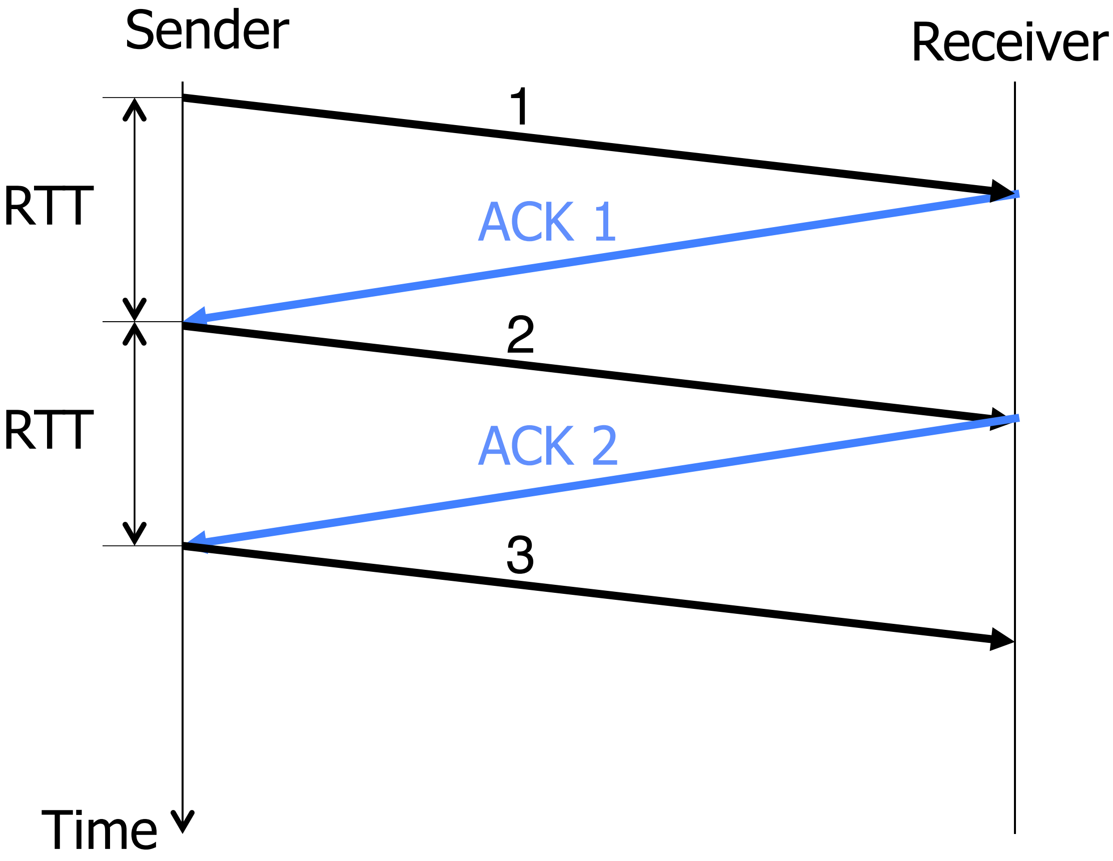
# Stop & Wait w/o Errors

- Say, RTT = 100ms
- 1 packet = 1500 bytes
- Throughput = 1500*8bits/0.1s = 120 Kbps



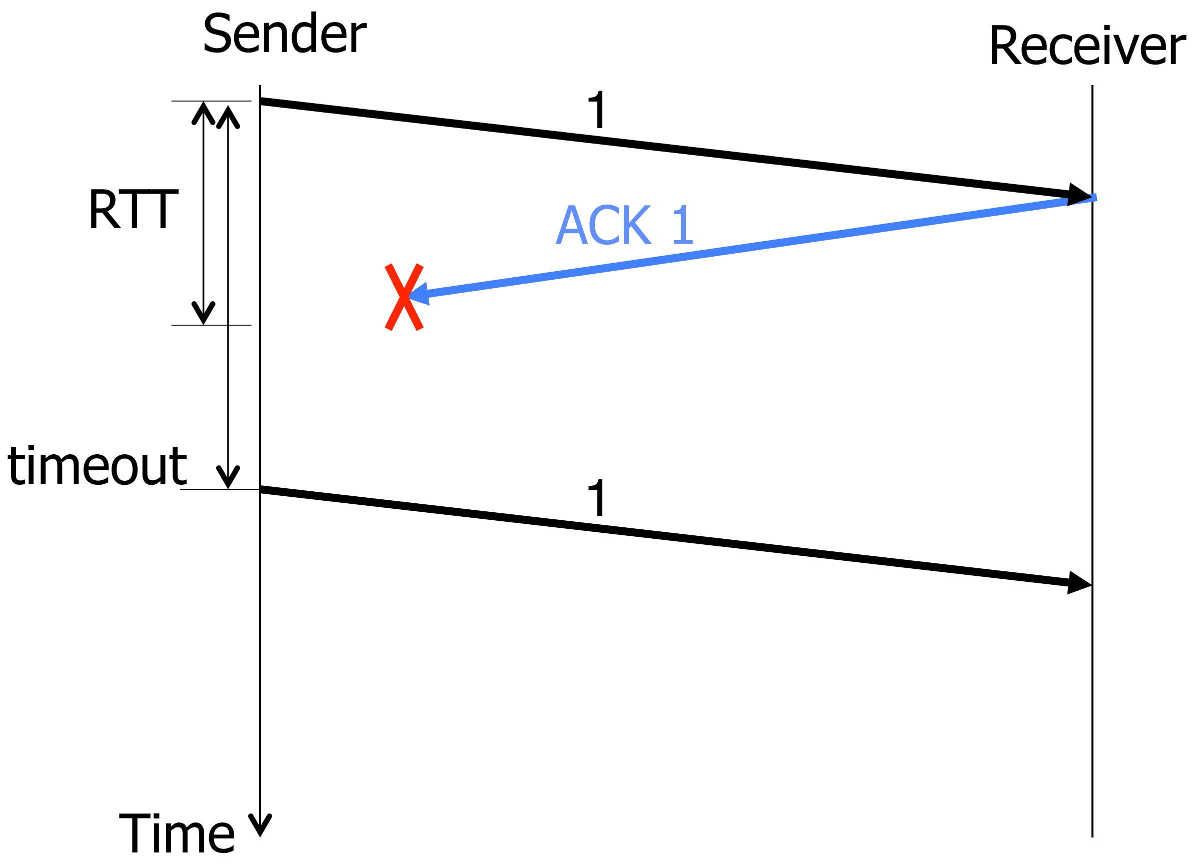
# Stop & Wait w/o Errors

- Can be highly inefficient for high capacity links
- Throughput doesn’t depend on the network capacity → even if capacity is 1Gbps, we can only send 120 Kbps!



# Stop & Wait with Errors

- If a loss wait for a retransmission timeout and retransmit
- Ho do you pick the timeout?

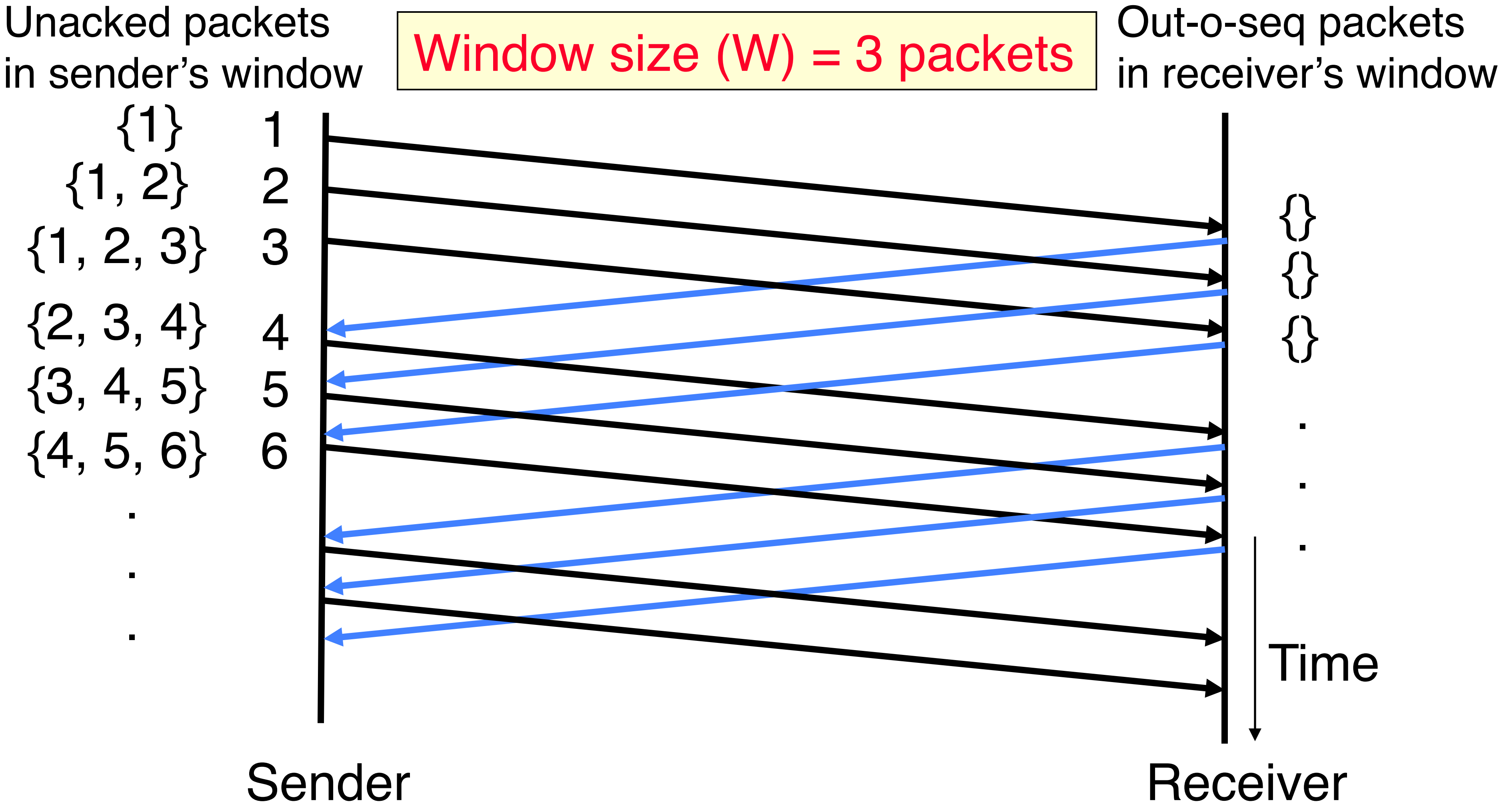


# Sliding Window

- *window* = set of adjacent sequence numbers
- The size of the set is the *window size*
- Assume window size is n
- Let A be the last ACK'd packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}
- Sender can send packets in its window
- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}
- Receiver can accept out of sequence, if in window

# Sliding Window w/o Errors

- Throughput = W*packet_size/RTT


Unacked packets in sender's window
Window size (W) = 3 packets
Out-o-seq packets in receiver's window
{1} 1
{1, 2} 2
{1, 2, 3} 3
{2, 3, 4} 4
{3, 4, 5} 5
{4, 5, 6} 6
.
.
.
{}
{}
{}
.
.
.
Time
Sender
Receiver

# Example: Sliding Window w/o Errors

- Assume
  - – Link capacity, C = 1Gbps
  - – Latency between end-hosts, RTT = 80ms
  - – packet_length = 1000 bytes
- What is the window size W to match link's capacity, C?

- Solution

  We want Throughput = C

  Throughput = W*packet_size/RTT

  C = W*packet_size/RTT

  **W = C*RTT/packet_size** = $10^9$bps*80*$10^{-3}$s/(8000b) = $10^4$ packets

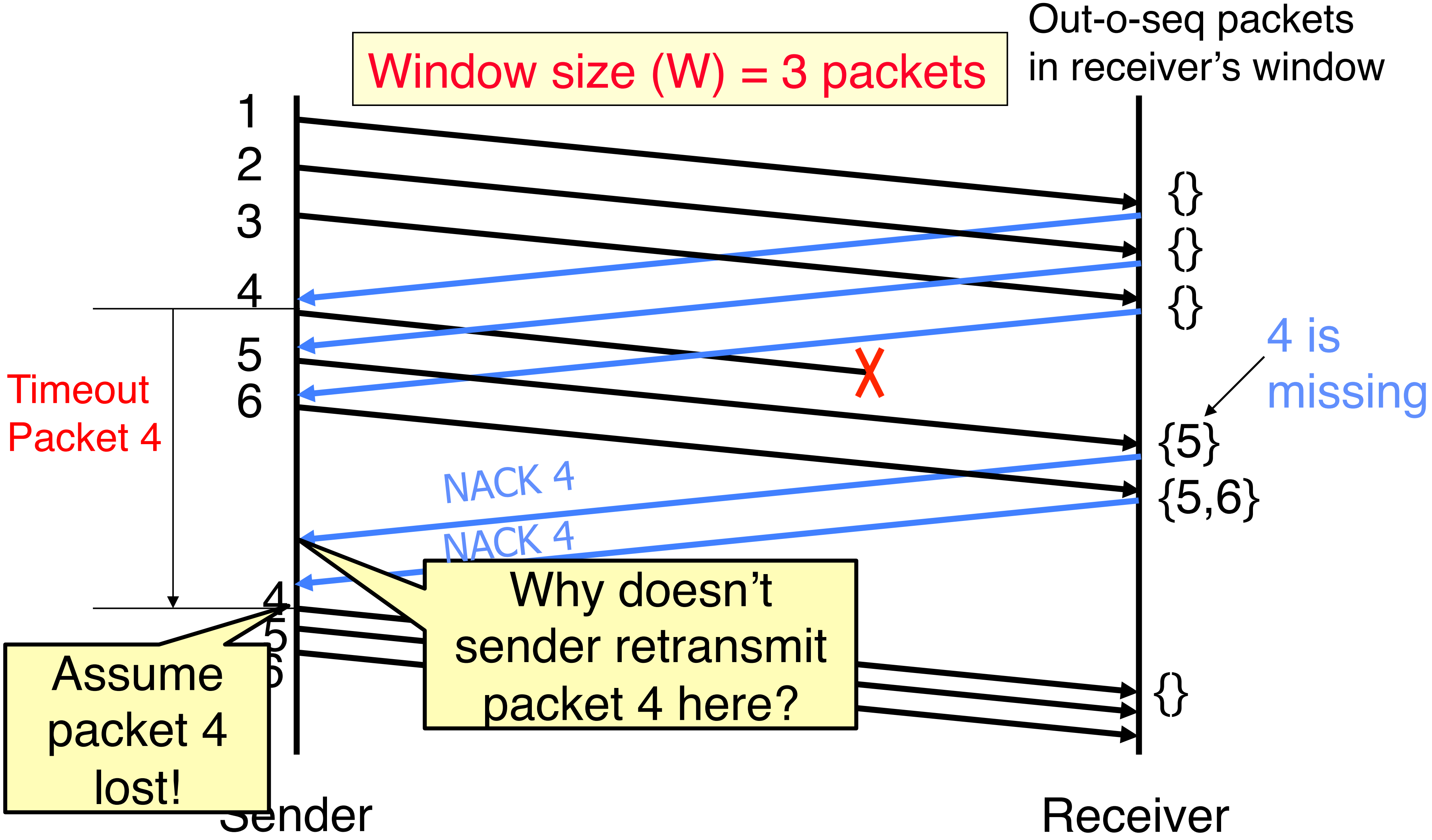
Window size ~ Bandwidth (Capacity), delay (RTT/2)

# Sliding Window with Errors

- Two approaches
  - – Go-Back-n (GBN)
  - – Selective Repeat (SR)
- In the absence of errors they behave identically

- Go-Back-n (GBN)
  - – Transmit up to *n* unacknowledged packets
  - – If timeout for ACK(*k*), retransmit *k*, *k+1*, …
  - – Typically uses NACKs instead of ACKs
    - » Recall, NACK specifies first in-sequence packet missed by receiver

# GBN Example with Errors



Why doesn't sender retransmit packet 4 here?

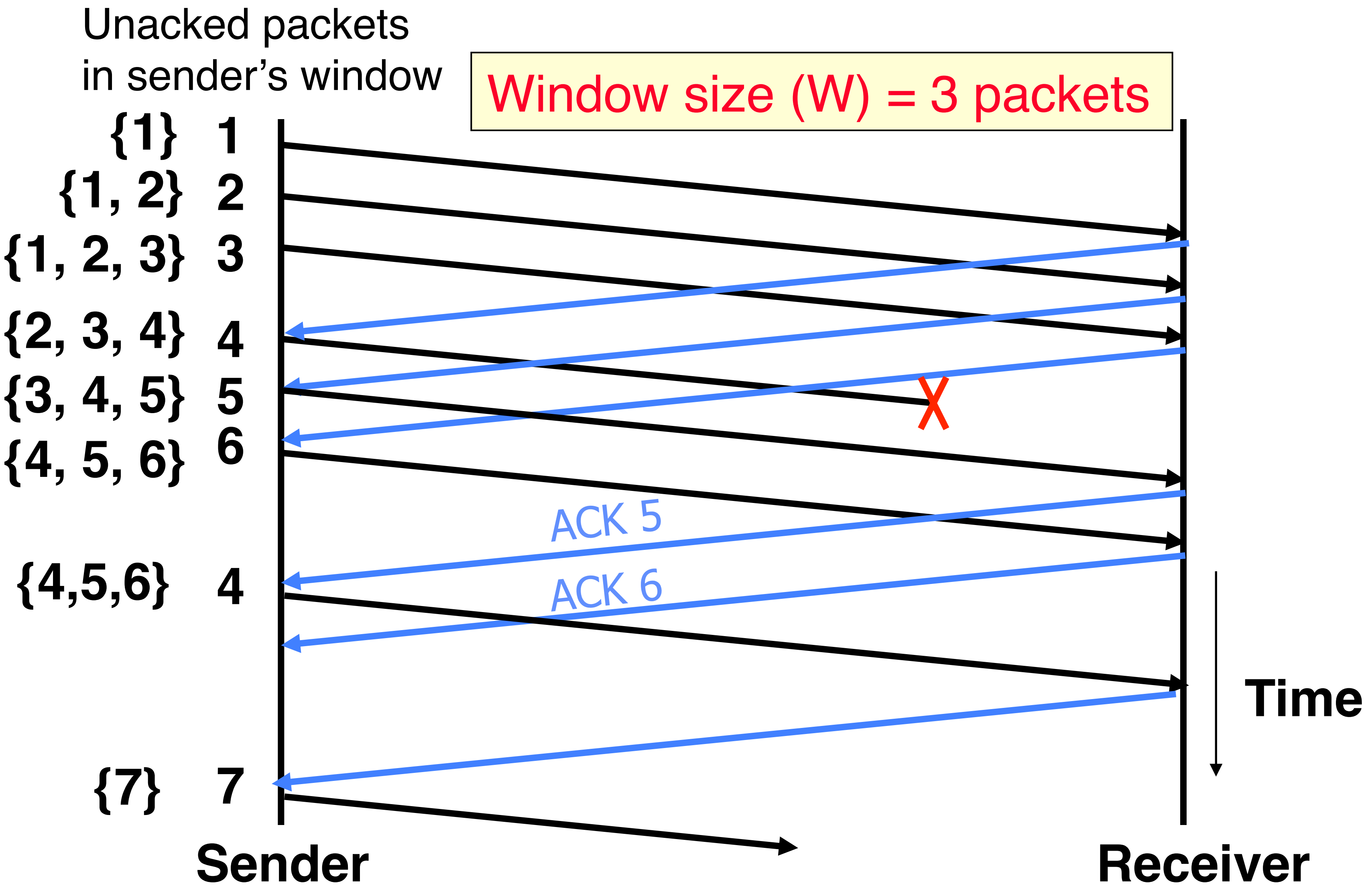
Assume packet 4 lost!

# Selective Repeat (SR)

- Sender: transmit up to *n* unacknowledged packets
- Assume packet *k* is lost
- Receiver: indicate packet *k* is missing (use ACKs)
- Sender: retransmit packet *k*

# SR Example with Errors

Window size (W) = 3 packets



# Summary

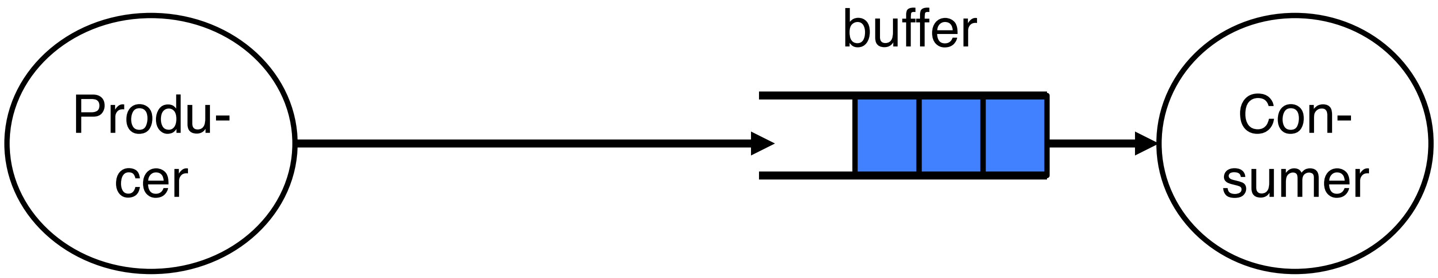
- TCP: Reliable Byte Stream
  - Open connection (3-way handshaking)
  - Close connection: no perfect solution; no way for two parties to agree in the presence of arbitrary message losses (General's Paradox)

- Reliable transmission
  - S&W not efficient for links with large capacity (bandwidth) delay product
  - Sliding window more efficient but more complex

# 5min Break

# Flow Control

- Recall: Flow control ensures a fast sender does not overwhelm a slow receiver
- Example: Producer-consumer with bounded buffer (Lecture 5)
  - – A buffer between producer and consumer
  - – Producer puts items into buffer as long as buffer **not full**
  - – Consumer consumes items from buffer



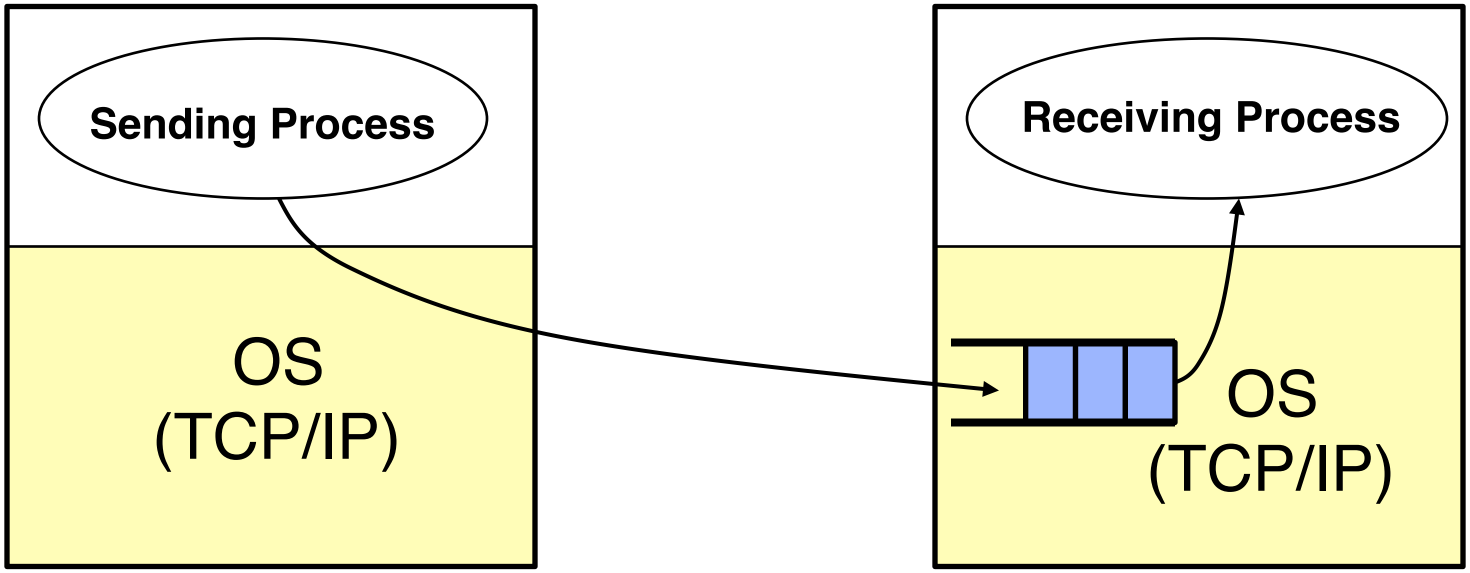
# TCP Flow Control

- TCP: sliding window protocol at byte (not packet) level
  - Go-back-N: TCP Tahoe, Reno, New Reno
  - Selective Repeat (SR): TCP Sack

- Receiver tells sender how many more bytes it can receive without overflowing its buffer (i.e., AdvertisedWindow)

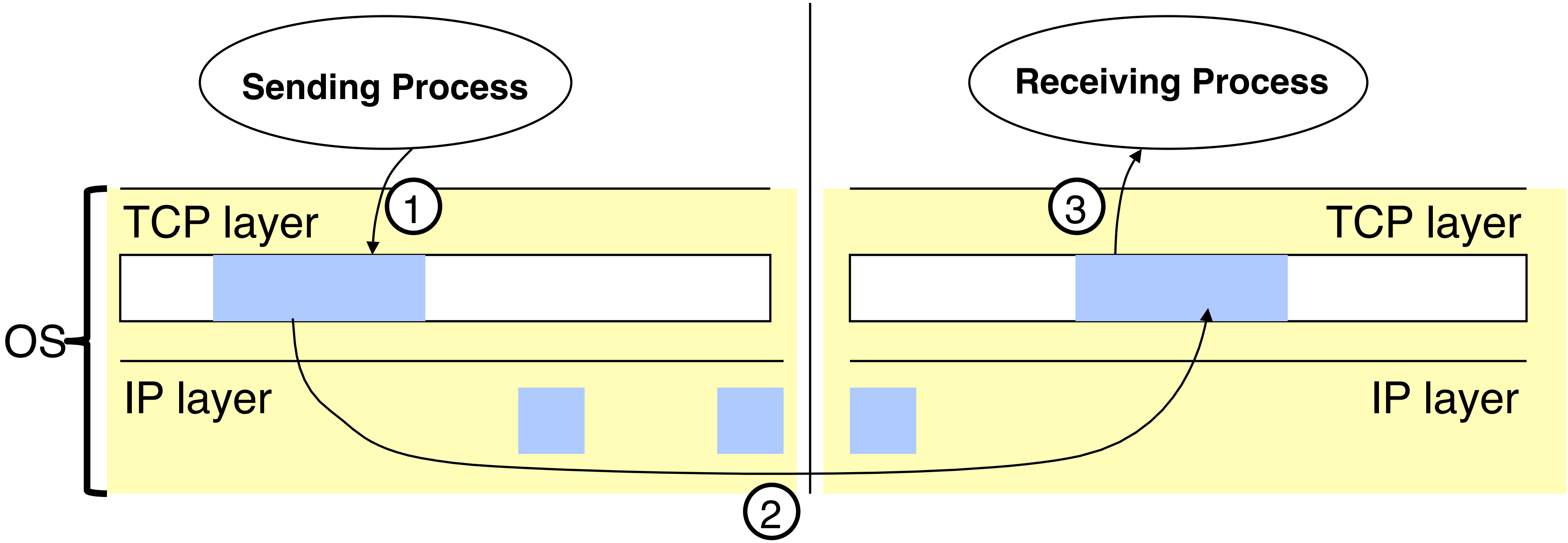
- The ACK contains sequence number N of next byte the receiver expects, i.e., receiver has received all bytes in sequence up to and including N-1

# TCP Flow Control



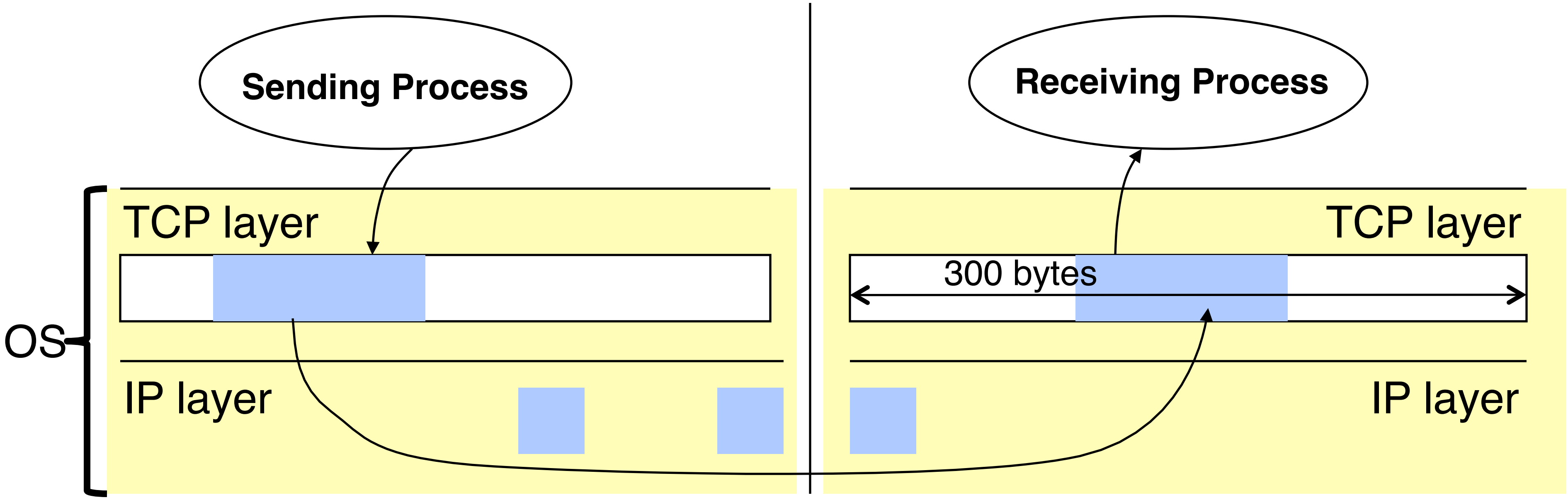
- TCP/IP implemented by OS (Kernel)
  - – Cannot do context switching on sending/receiving every packet
    - » At 1Gbps, it takes 12 usec to send an 1500 bytes, and 0.8usec to send an 100 byte packet
- Need buffers to match …
  - – sending app with sending TCP
  - – receiving TCP with receiving app

# TCP Flow Control



- Three pairs of producer-consumer’s
  - ① sending process → sending TCP
  - ② Sending TCP → receiving TCP
  - ③ receiving TCP → receiving process

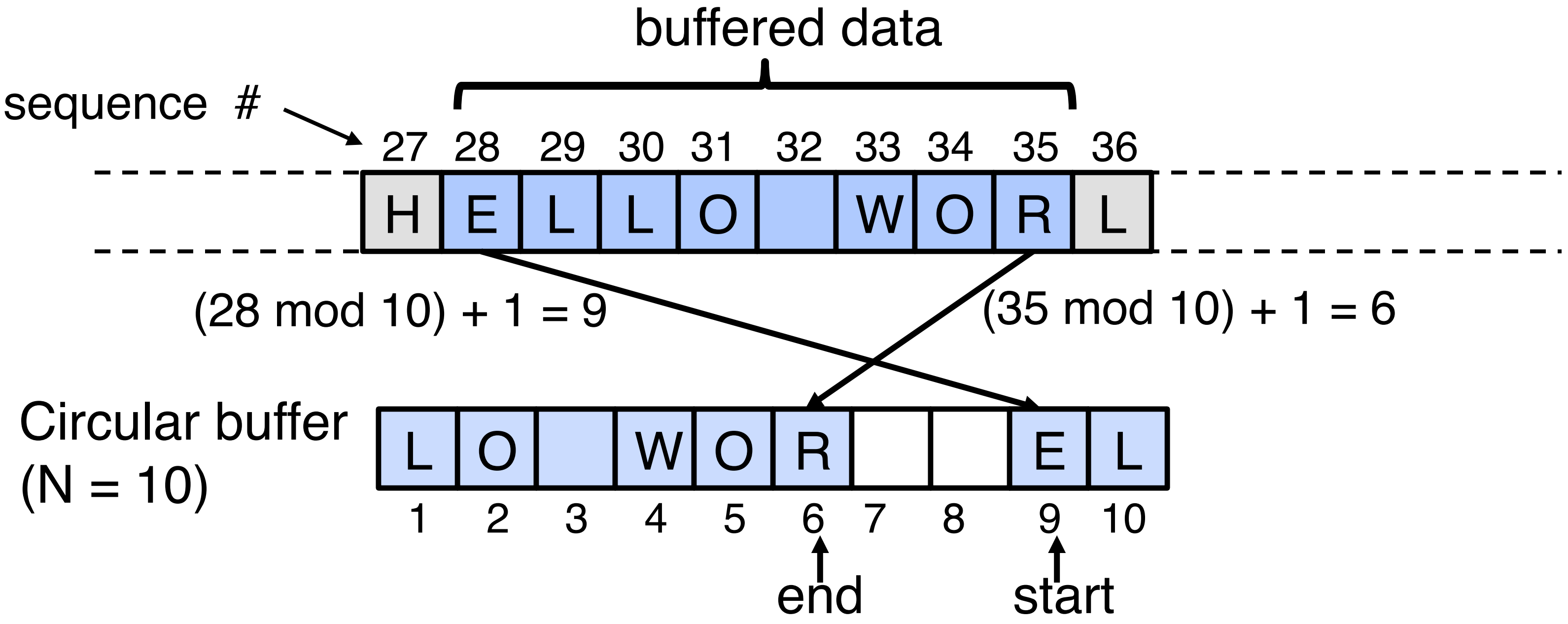
# TCP Flow Control



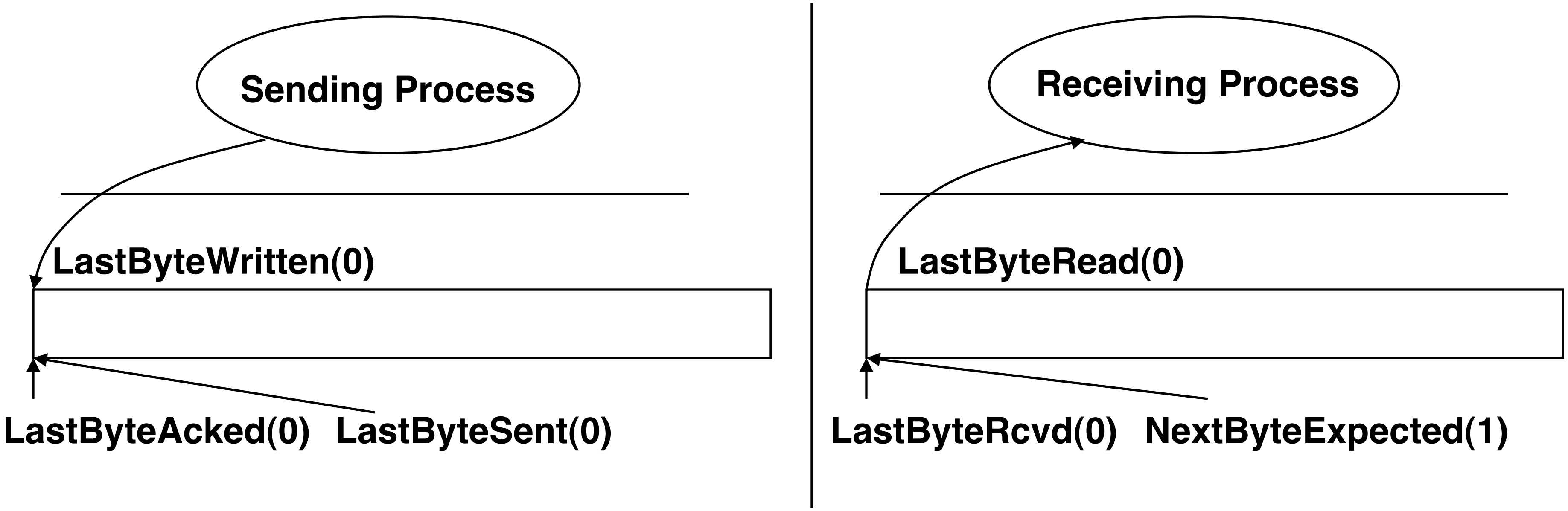
- Example assumptions:
  - Maximum IP packet size = 100 bytes
  - Size of the receiving buffer (MaxRcvBuf) = 300 bytes
- Recall, ack indicates the next expected byte in-sequence, not the last received byte
- Use circular buffers

# Circular Buffer

- Assume
  - A buffer of size N
  - A stream of bytes, where bytes have increasing sequence numbers
    - » Think of stream as an unbounded array of bytes and of sequence number as indexes in this array
- Buffer stores at most N consecutive bytes from the stream
- Byte k stored at position (k mod N) + 1 in the buffer

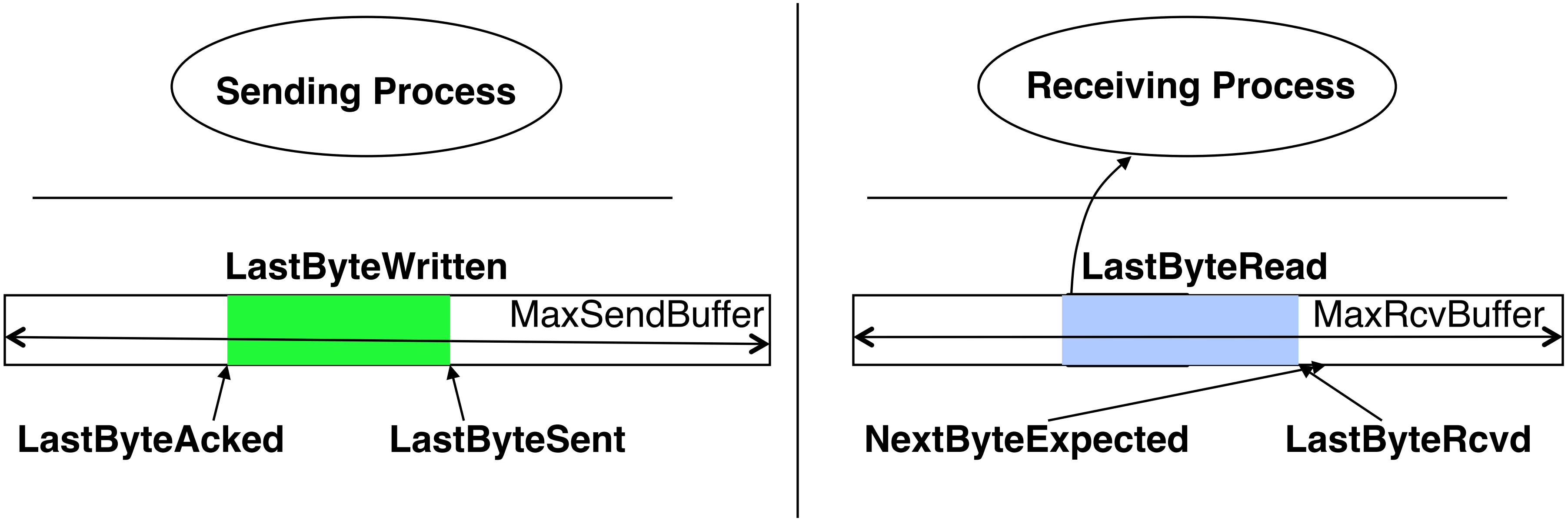


# TCP Flow Control



- LastByteWritten: last byte written by sending process
- LastByteSent: last byte sent by sender to receiver
- LastByteAcked: last ack received by sender from receiver
- LastByteRcvd: last byte received by receiver from sender
- NextByteExpected: last in-sequence byte expected by receiver
- LastByteRead: last byte read by the receiving process

# TCP Flow Control



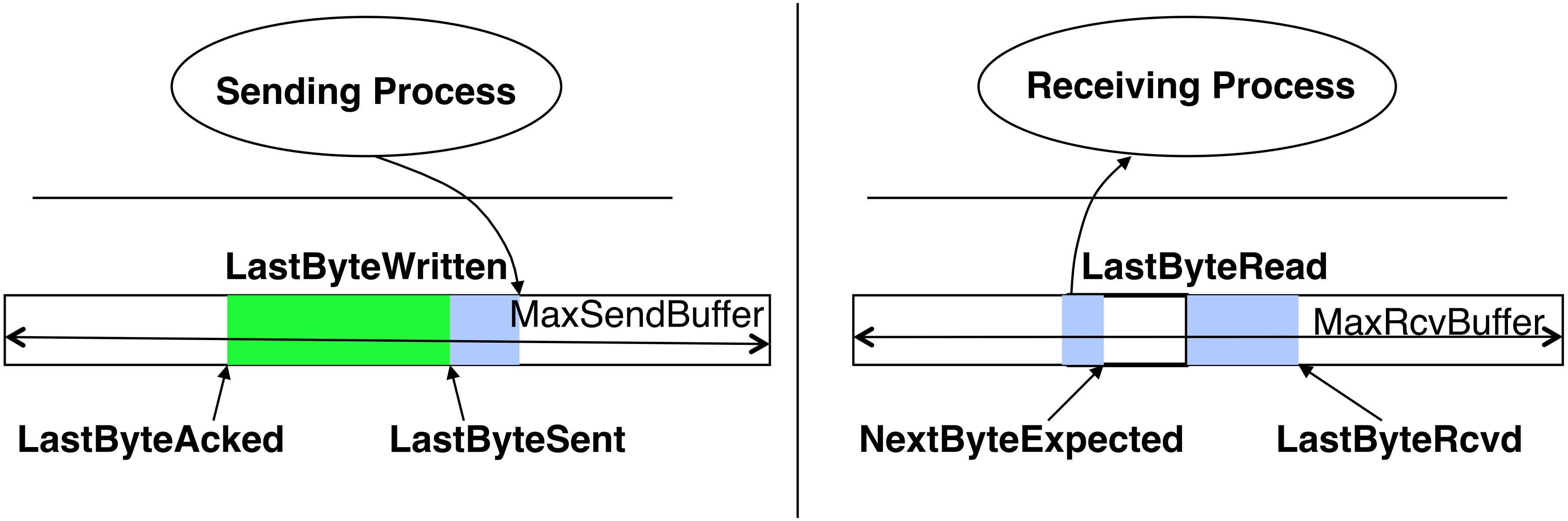
- AdvertisedWindow: number of bytes TCP receiver can receive

**AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)**

- SenderWindow: number of bytes TCP sender can send

**SenderWindow = AdvertisedWindow – (LastByteSent – LastByteAcked)**

# TCP Flow Control



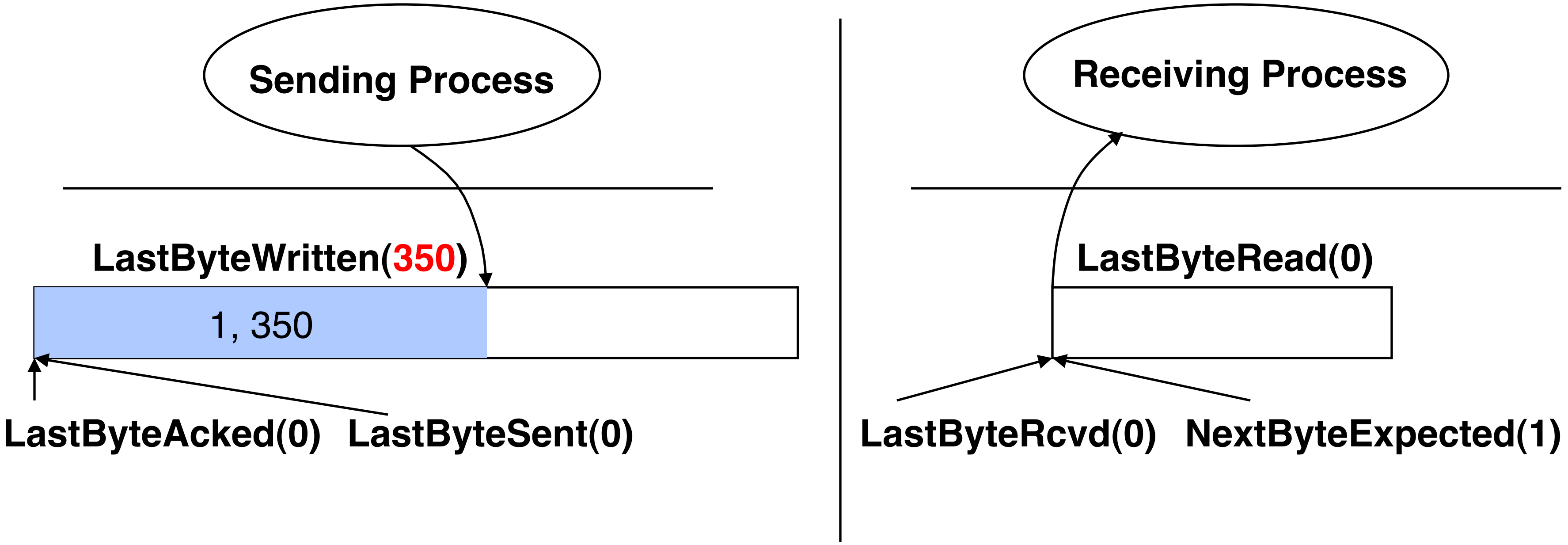
- Still true if receiver missed data….

**AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)**

- WriteWindow: number of bytes sending process can write

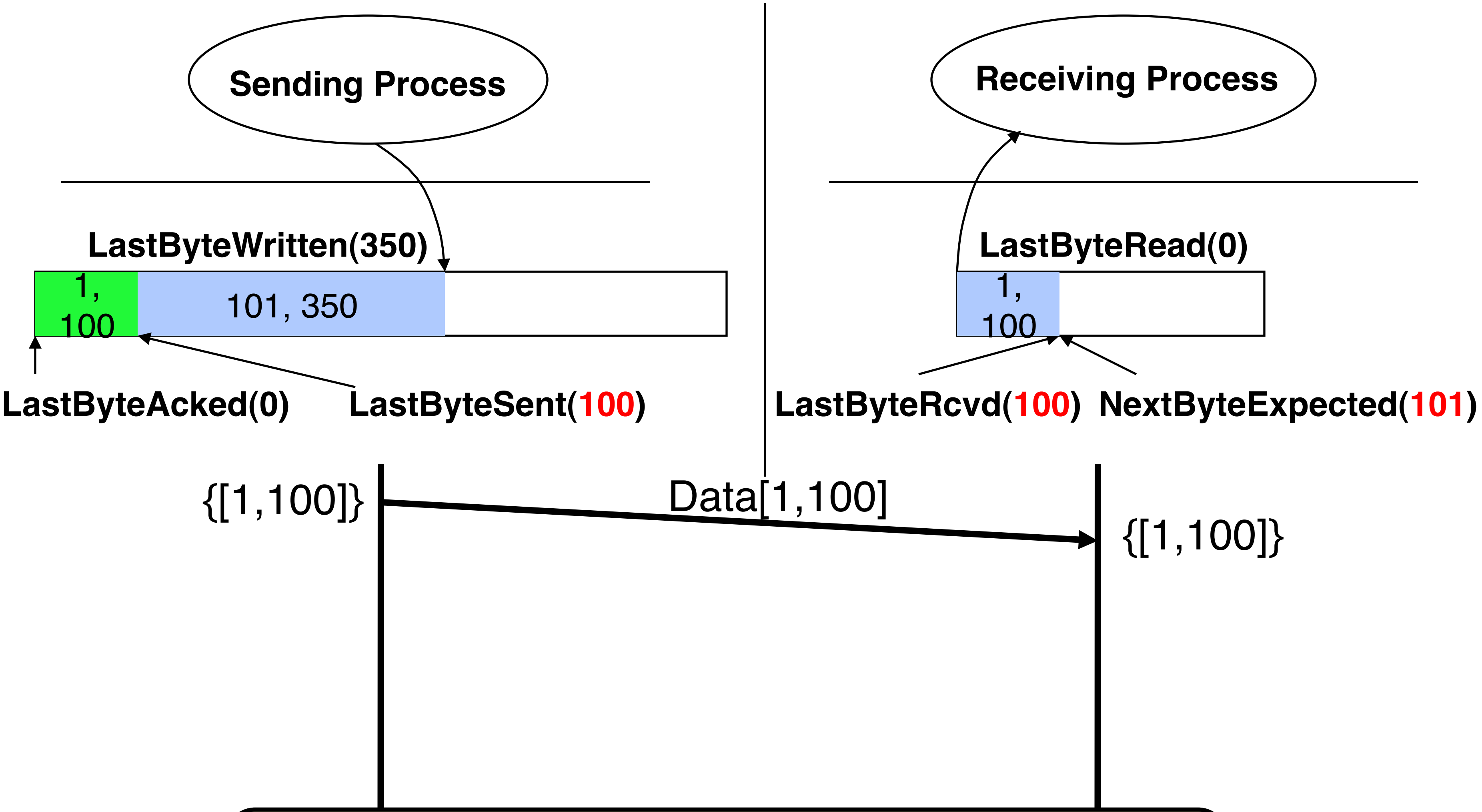
**WriteWindow = MaxSendBuffer – (LastByteWritten – LastByteAcked)**

# TCP Flow Control



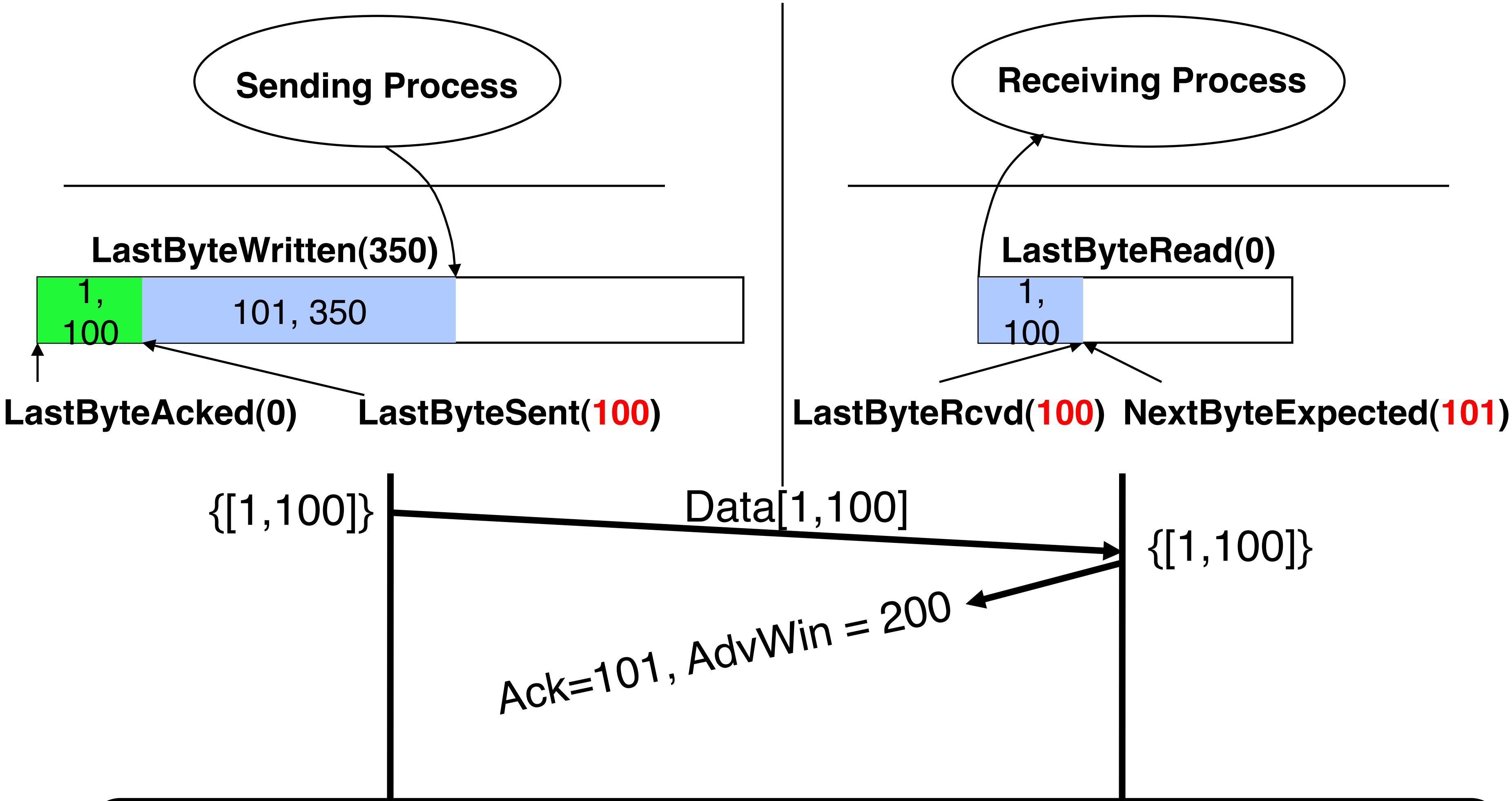
- Sending app sends 350 bytes
- Recall:
  - – We assume IP only accepts packets no larger than 100 bytes
  - – MaxRcvBuf = 300 bytes, so initial Advertised Window = 300 byets

# TCP Flow Control



Sender sends first packet (i.e., first 100 bytes) and receiver gets the packet

# TCP Flow Control

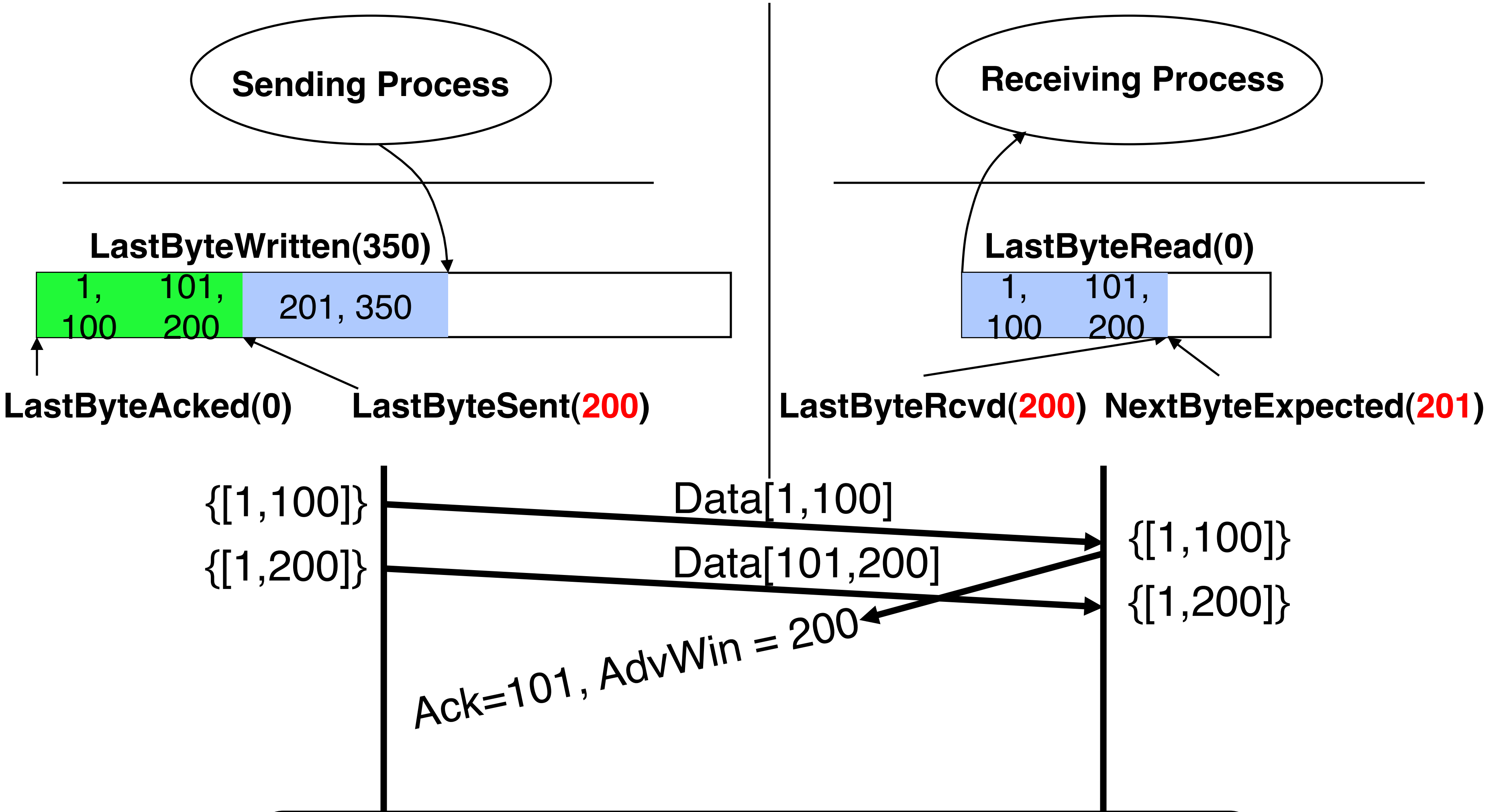


Receiver sends ack for 1st packet

**AdvWin = MaxRcvBuffer – (LastByteRcvd – LastByteRead)**

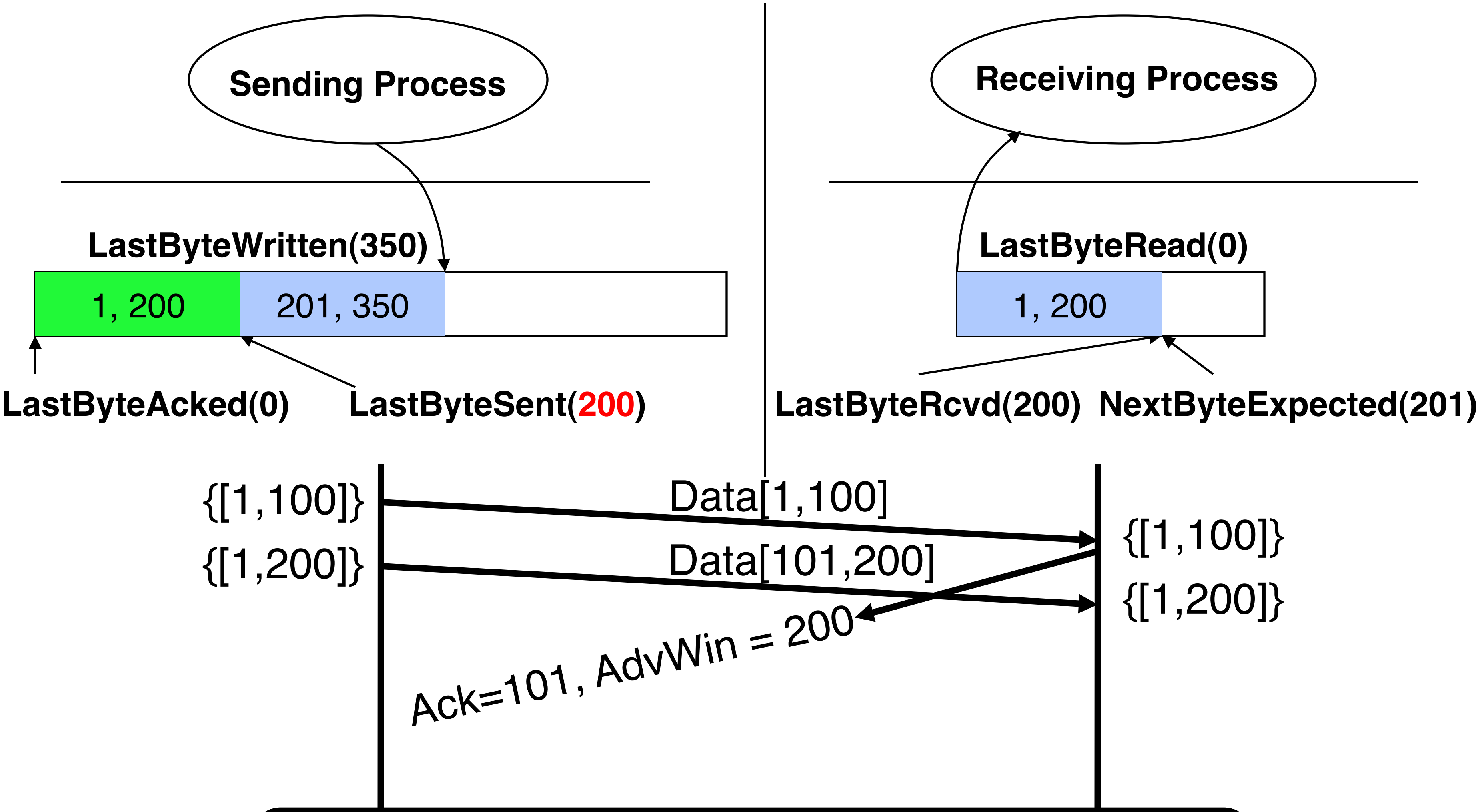
**= 300 – (100 – 0) = 200**

# TCP Flow Control



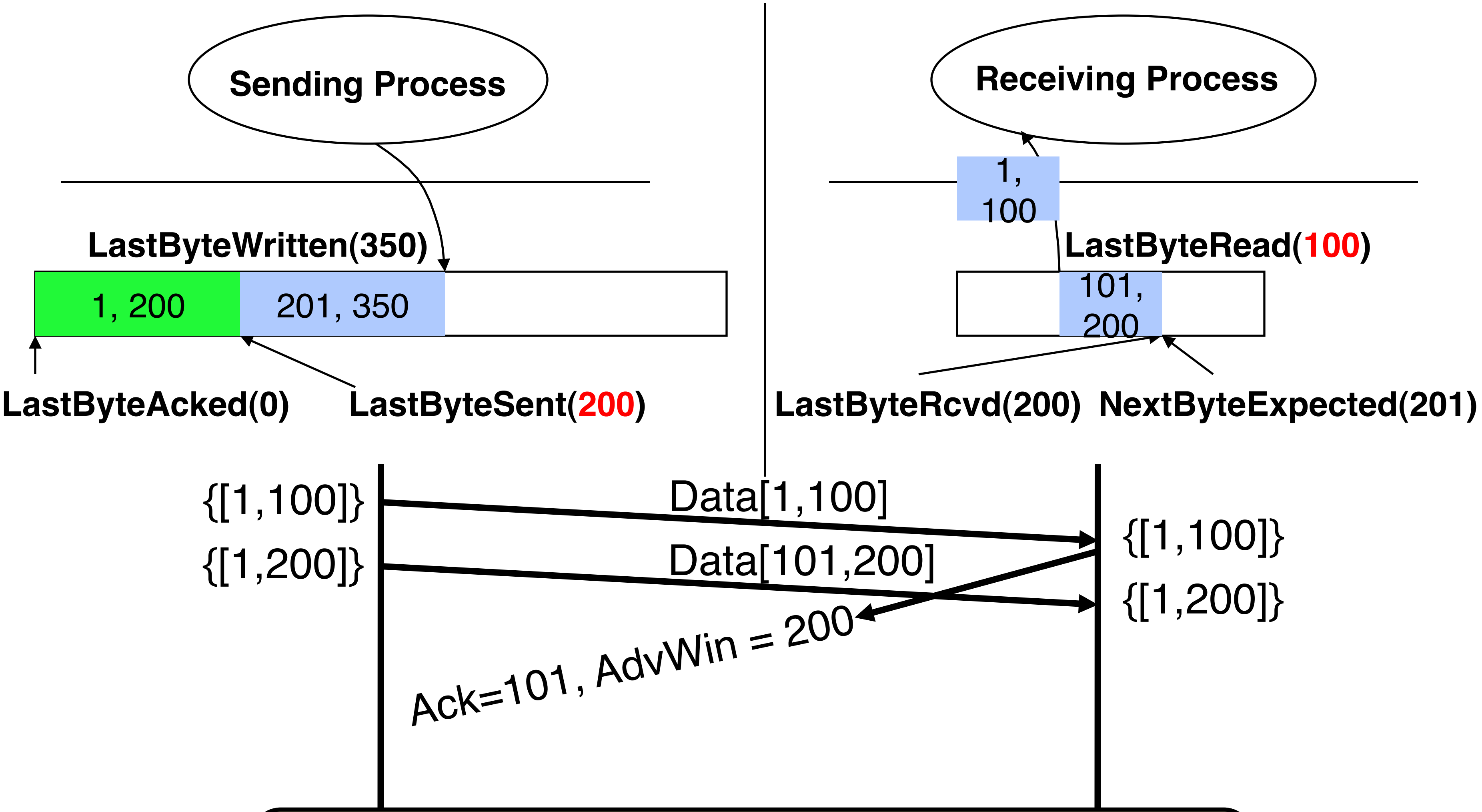
Sender sends 2nd packet (i.e., next 100 bytes) and receiver gets the packet

# TCP Flow Control



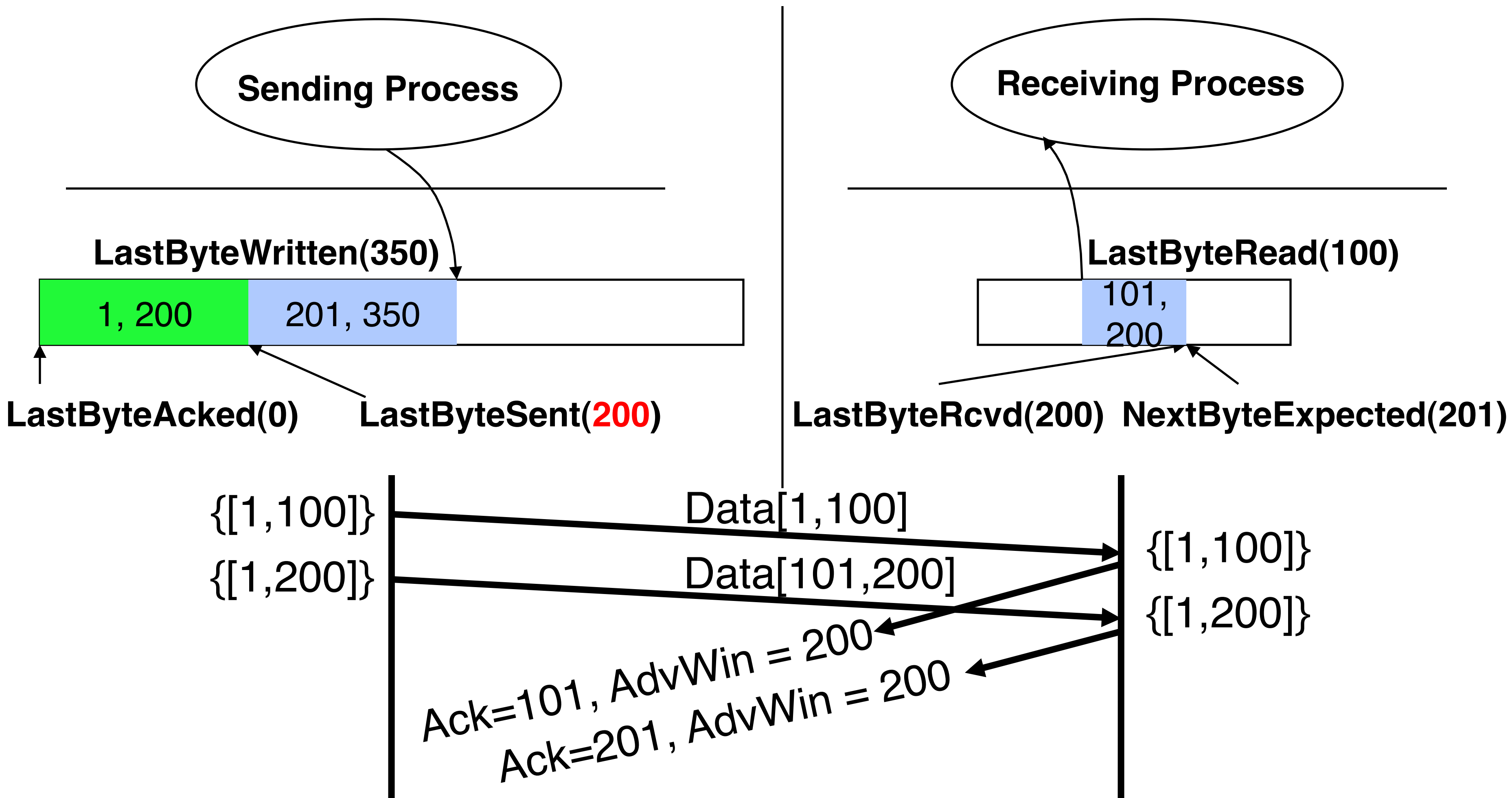
Sender sends 2nd packet (i.e., next 100 bytes) and receiver gets the packet

# TCP Flow Control



Receiving TCP delivers first 100 bytes to recienving process

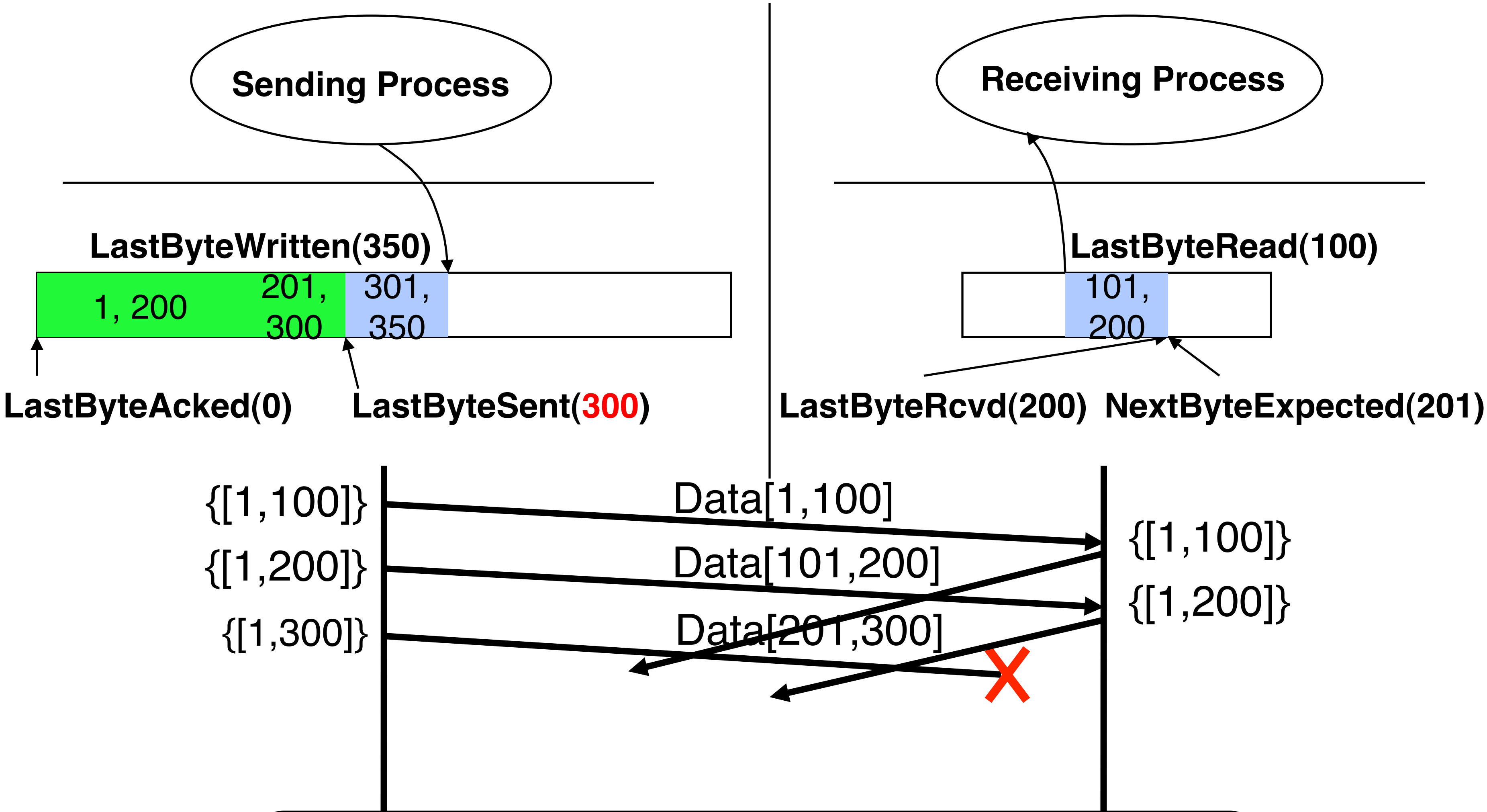
# TCP Flow Control



Receiver sends ack for 2nd packet

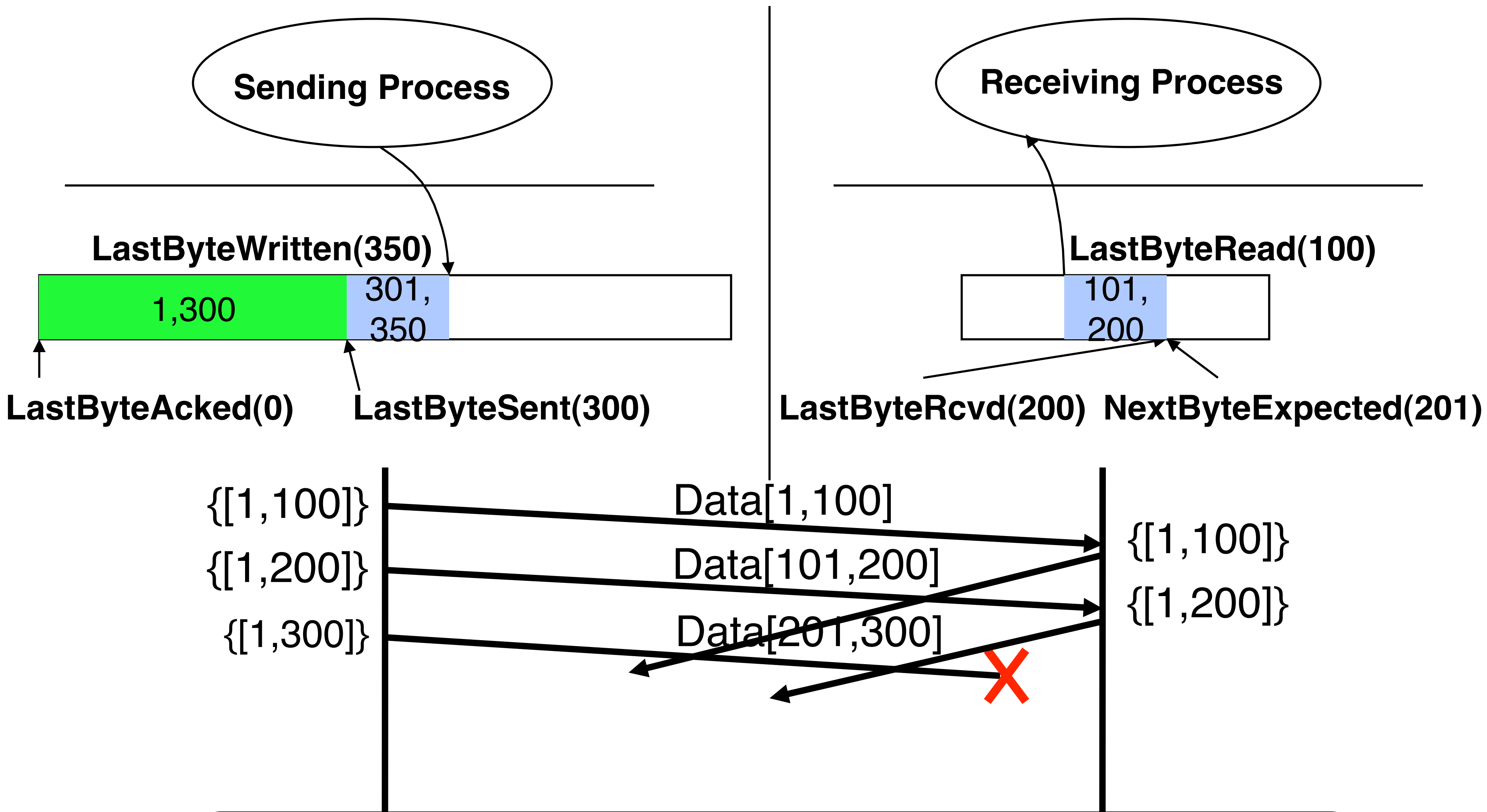
**AdvWin = MaxRcvBuffer – (LastByteRcvd – LastByteRead)**
**= 300 – (200 – 100) = 200**

# TCP Flow Control



Sender sends 3rd packet (i.e., next 100 bytes) and the packet is lost

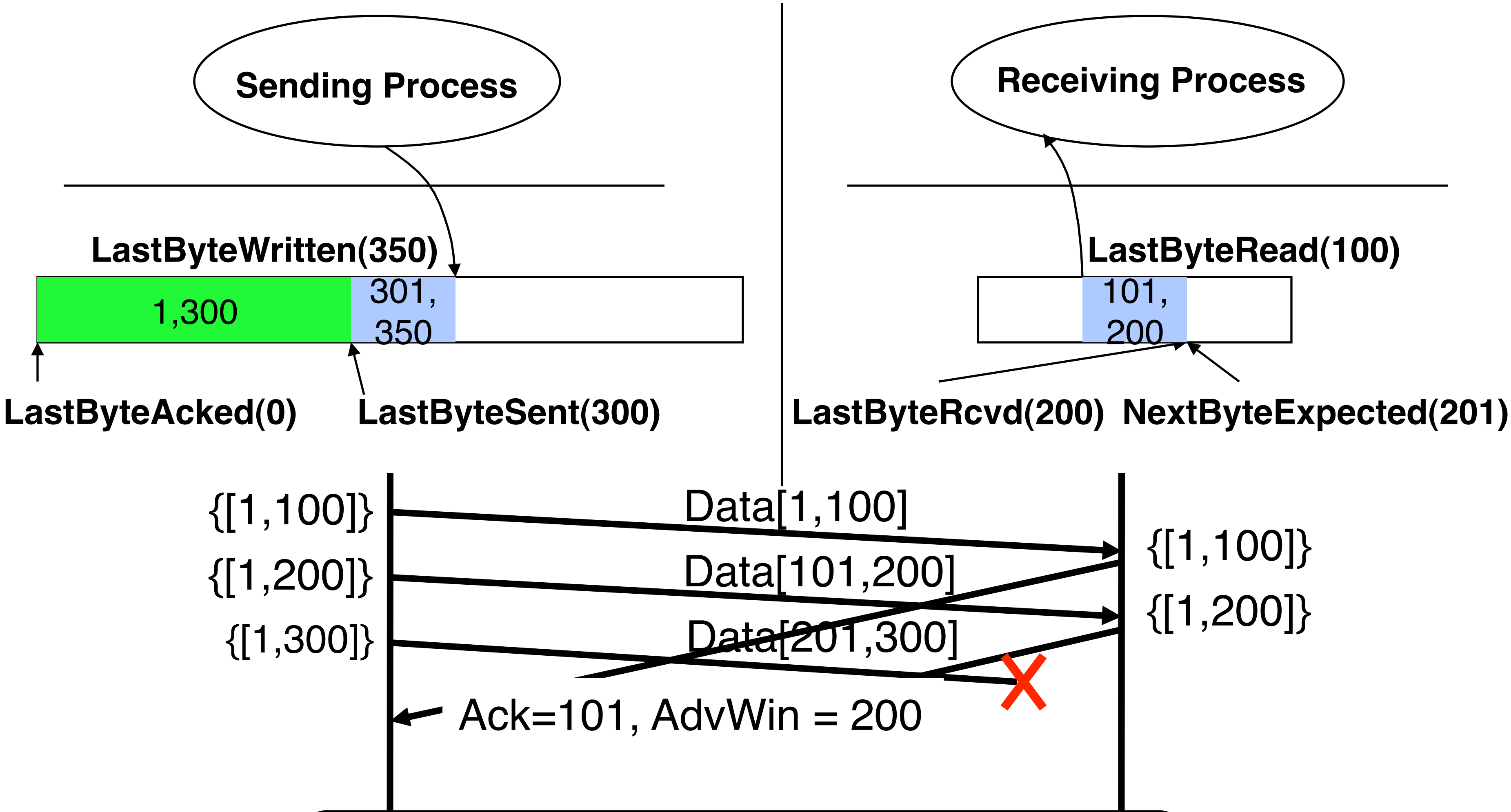
# TCP Flow Control



Sender stops sending as window full

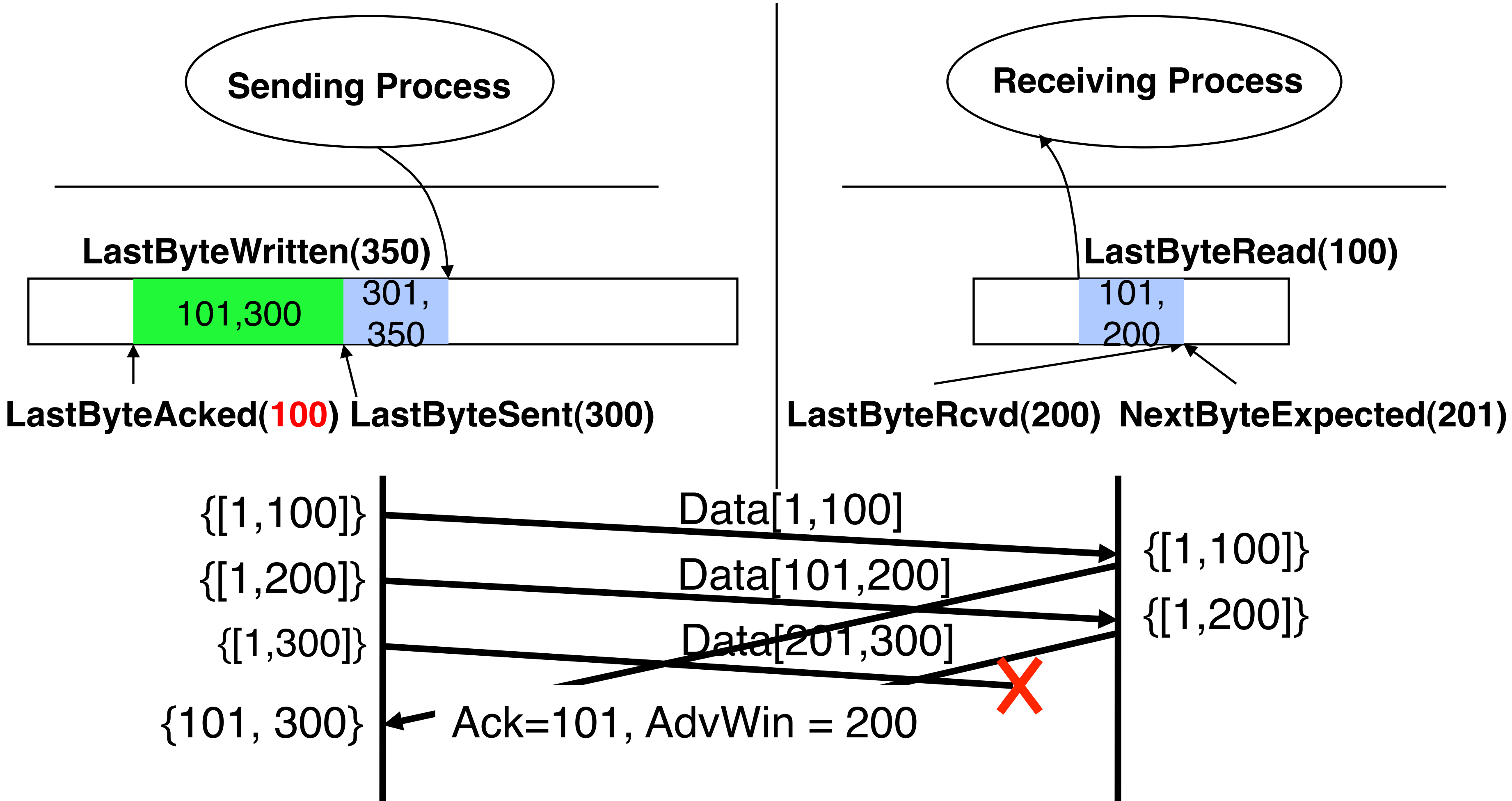
**SndWin = AdvWin – (LastByteSent – LastByteAcked)**
**= 300 – (300 – 0) = 0**

# TCP Flow Control



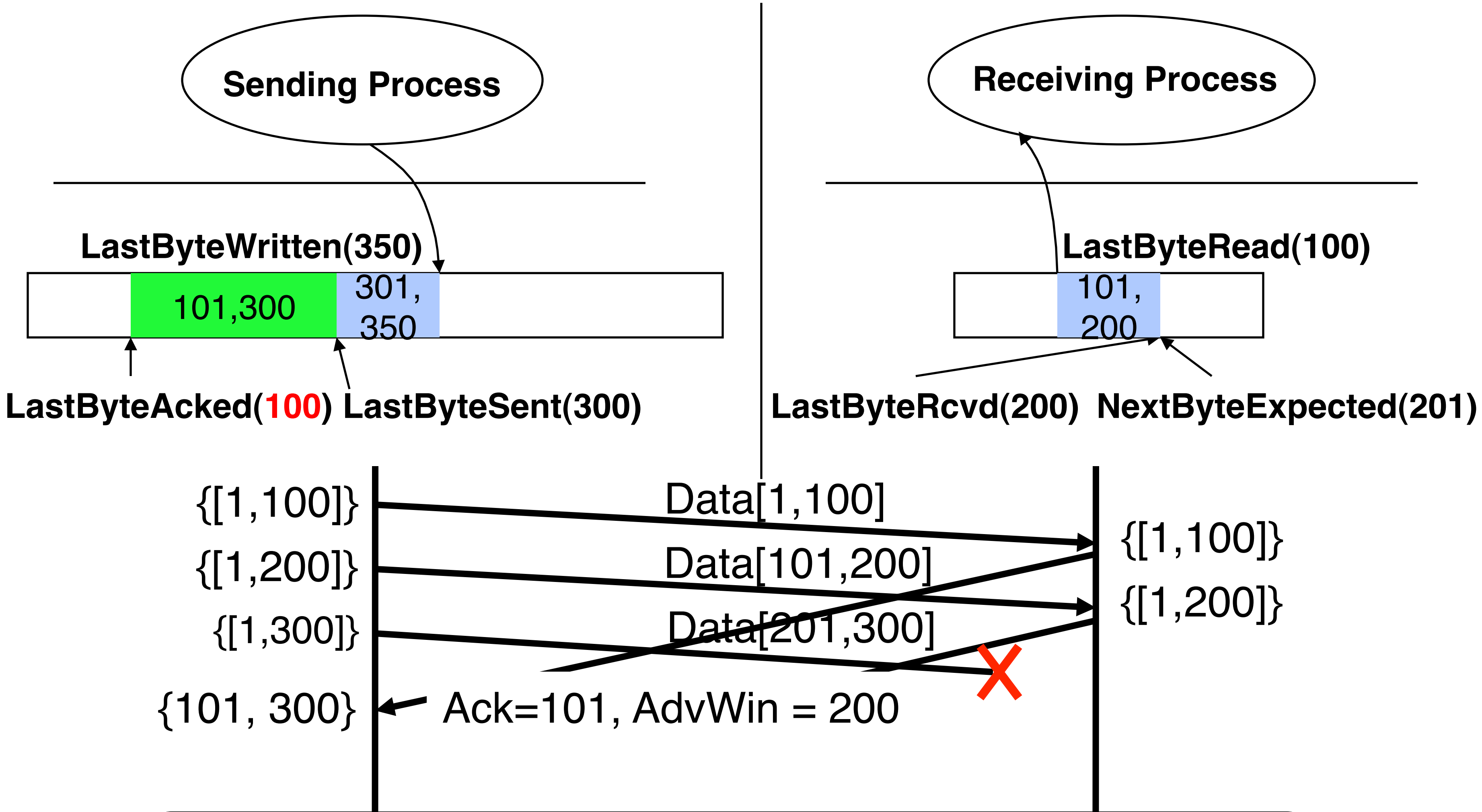
- Sender gets ack for 1$^{st}$ packet
- AdWin = 200

# TCP Flow Control



- Ack for 1st packet (ack indicates next byte expected by receiver)
- Receiver no longer needs first 100 bytes

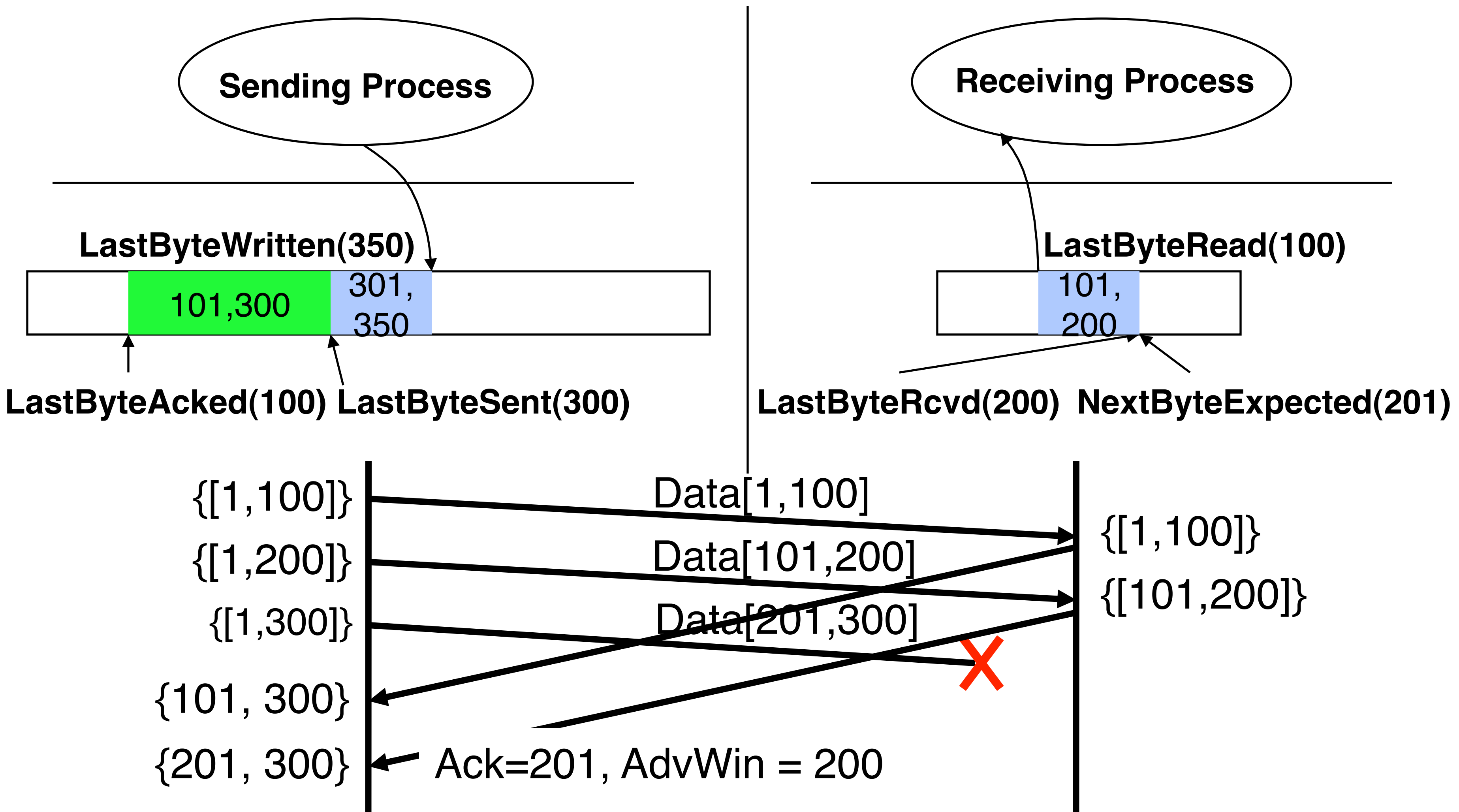
# TCP Flow Control



Sender still cannot send as window full

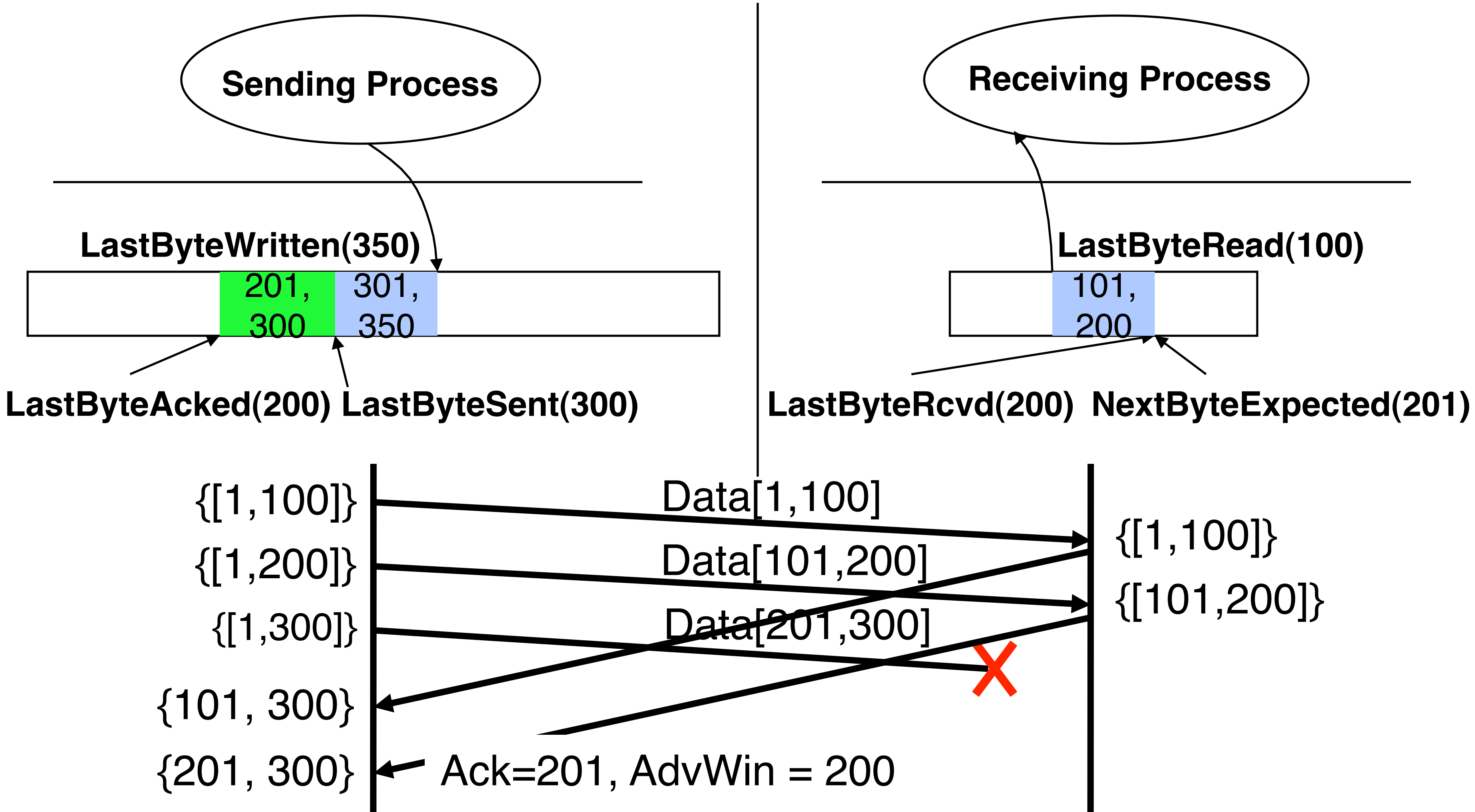
**SndWin = AdvWin – (LastByteSent – LastByteAcked)**
**= 200 – (300 – 100) = 0**

# TCP Flow Control



- Receiver gets ack for 2nd packet
- AdvWin = 200 bytes

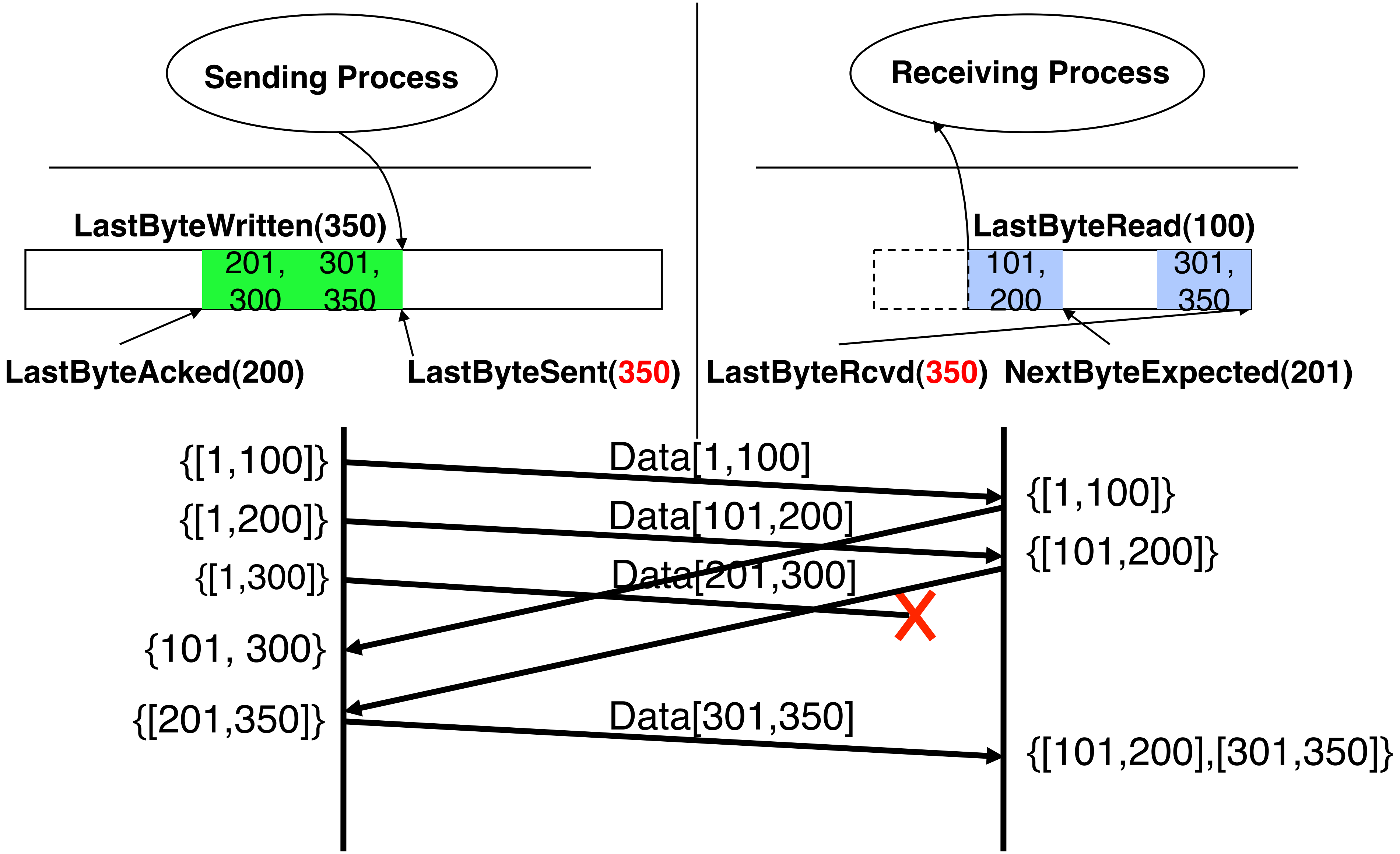
# TCP Flow Control



Sender can now send new data!

**SndWin = AdvWin – (LasByteSent – LastByteAcked) = 100**

# TCP Flow Control



Sending Process

LastByteWritten(350)

201, 300

301, 350

LastByteAcked(200)

LastByteSent(350)

Receiving Process

LastByteRead(100)

101, 200

301, 350

LastByteRcvd(350)

NextByteExpected(201)

{[1,100]}

{[1,200]}

{[1,300]}

{101, 300}

{[201,350]}

Data[1,100]

Data[101,200]

Data[201,300]

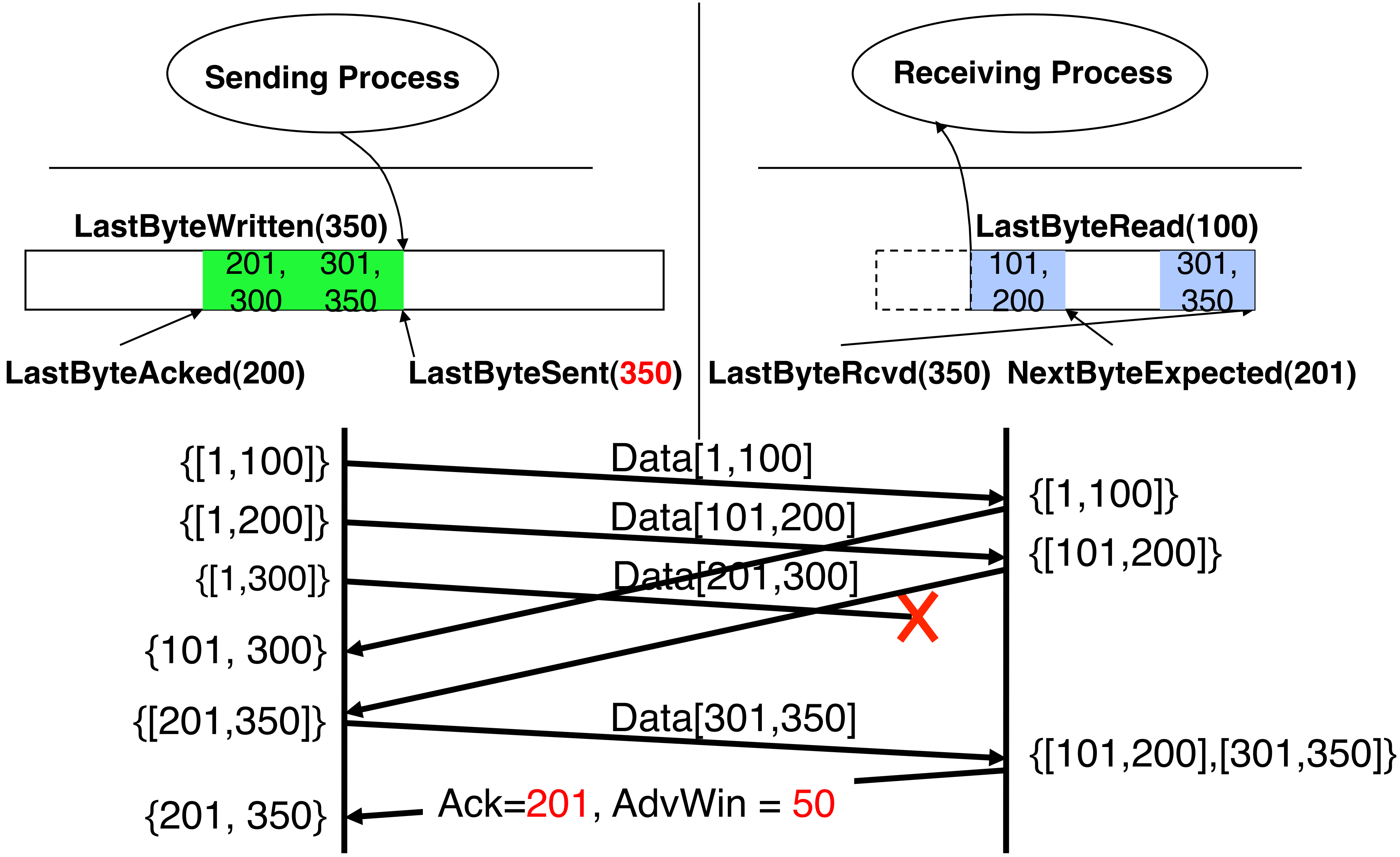
Data[301,350]

{[1,100]}

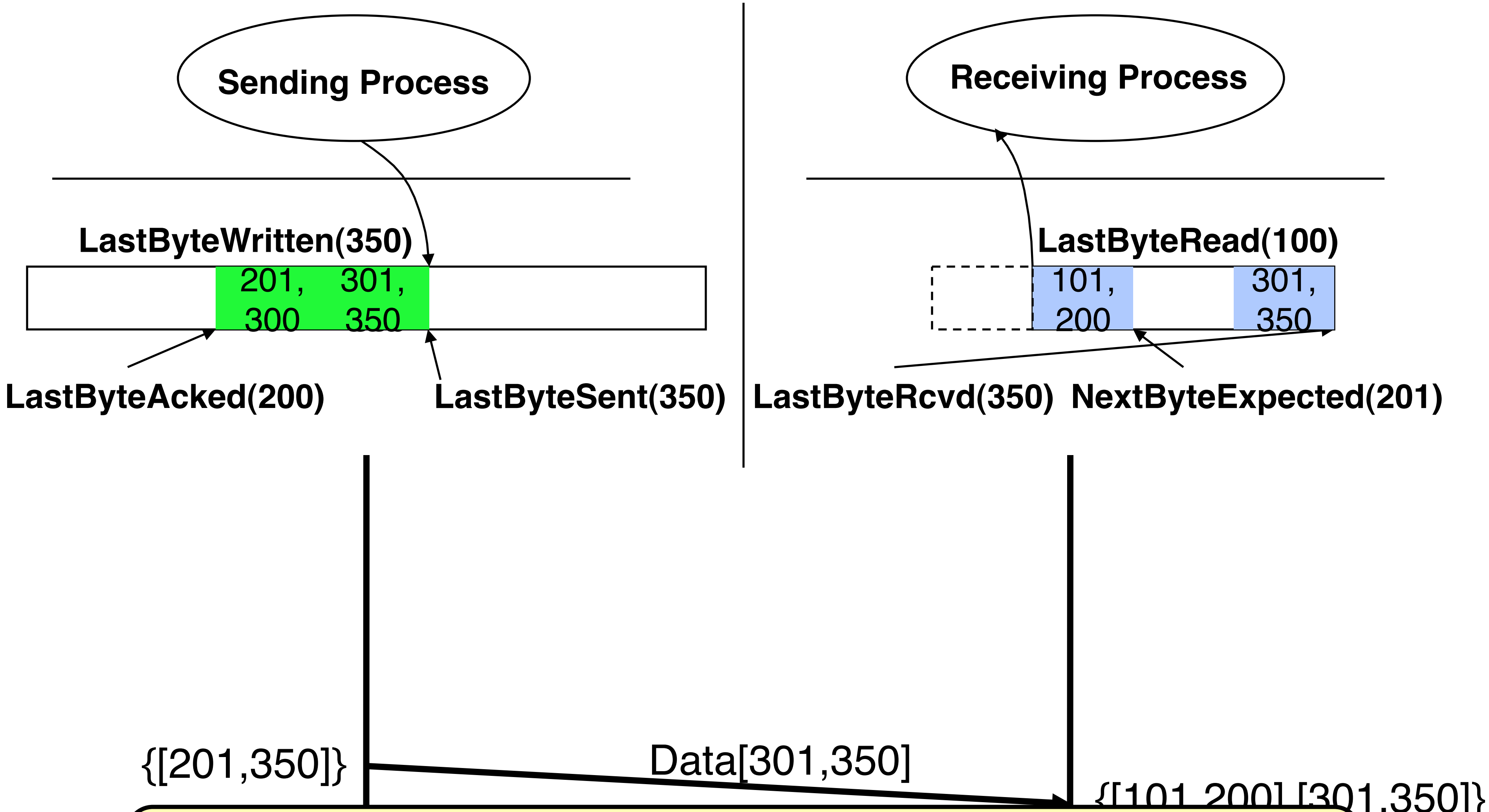
{[101,200]}

{[101,200],[301,350]}

# TCP Flow Control

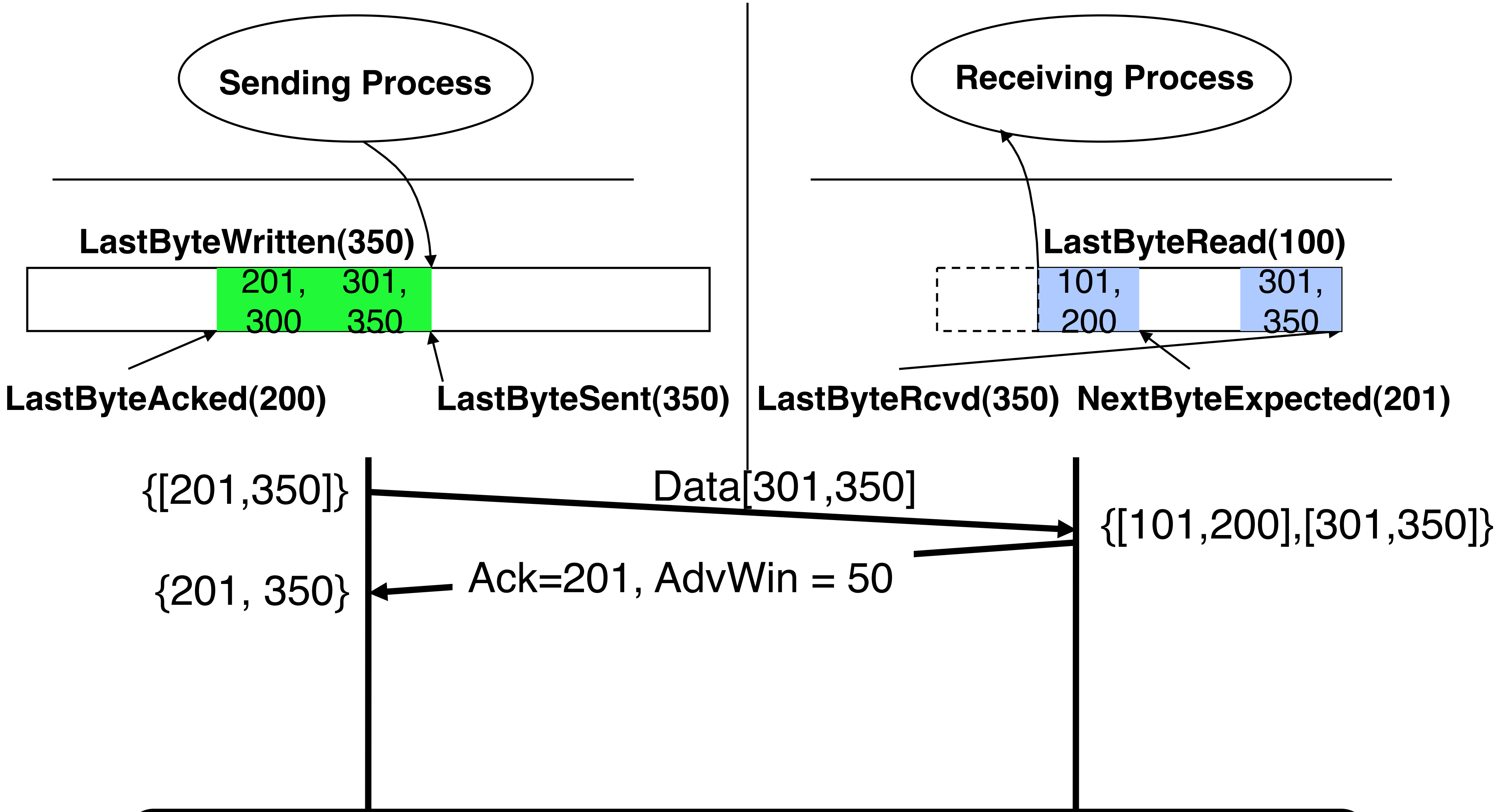


# TCP Flow Control



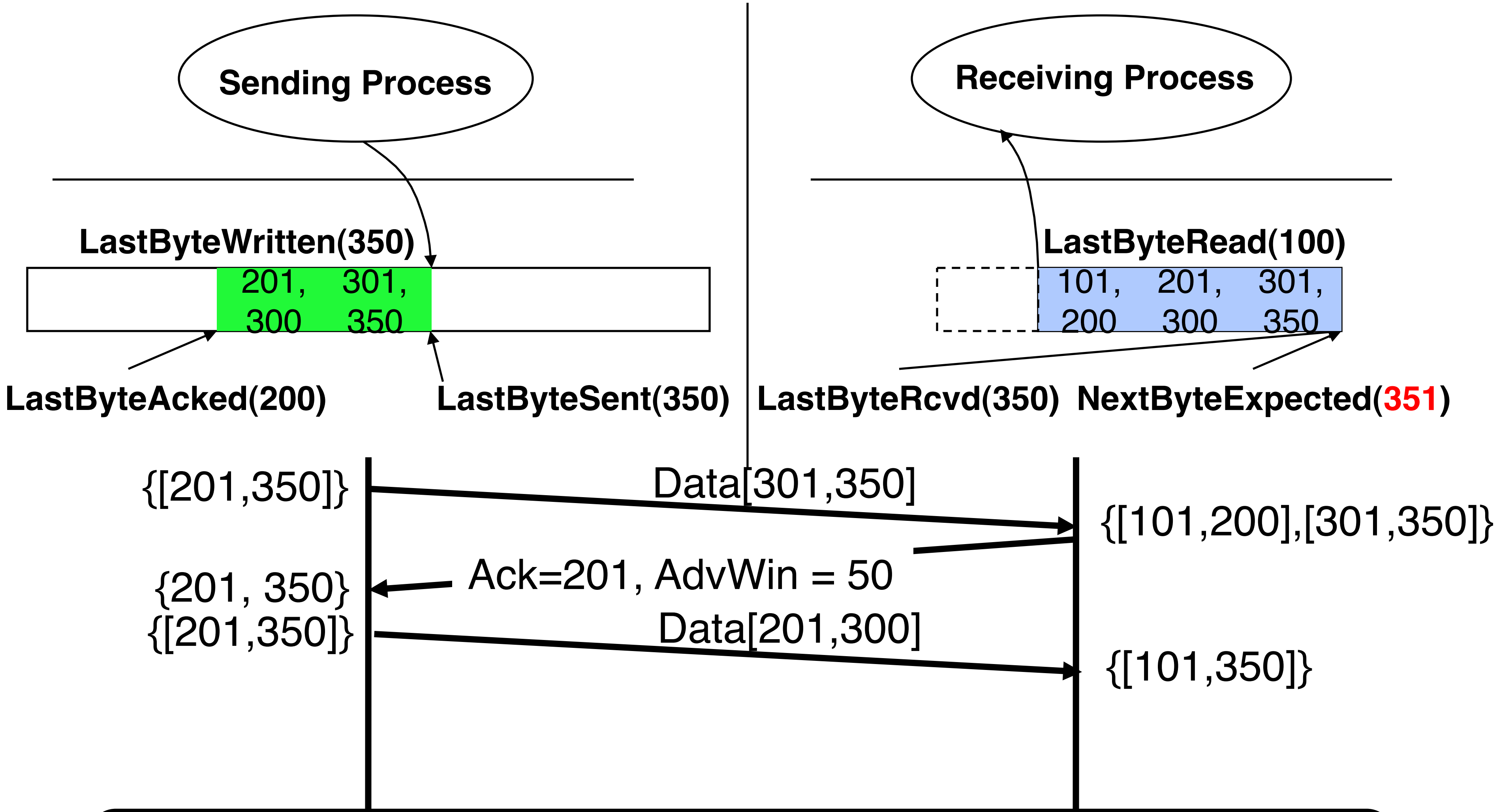
- Ack still specifies 201 (first byte out of sequence)
- AdvWin = 50, so can sender re-send 3rd packet?

# TCP Flow Control



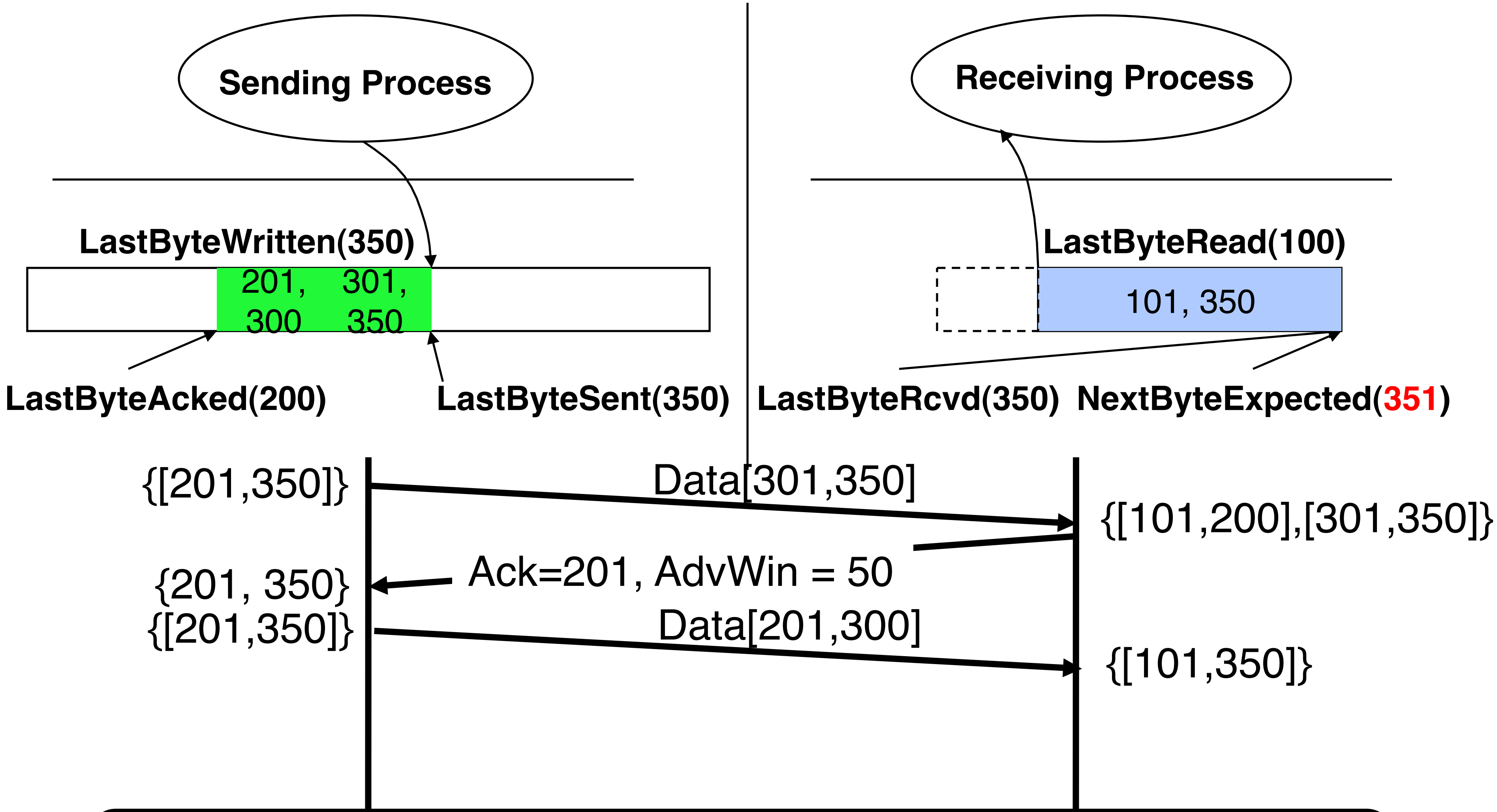
- Ack still specifies 201 (first byte out of sequence)
- AdvWin = 50, so can sender re-send 3$^{rd}$ packet?

# TCP Flow Control



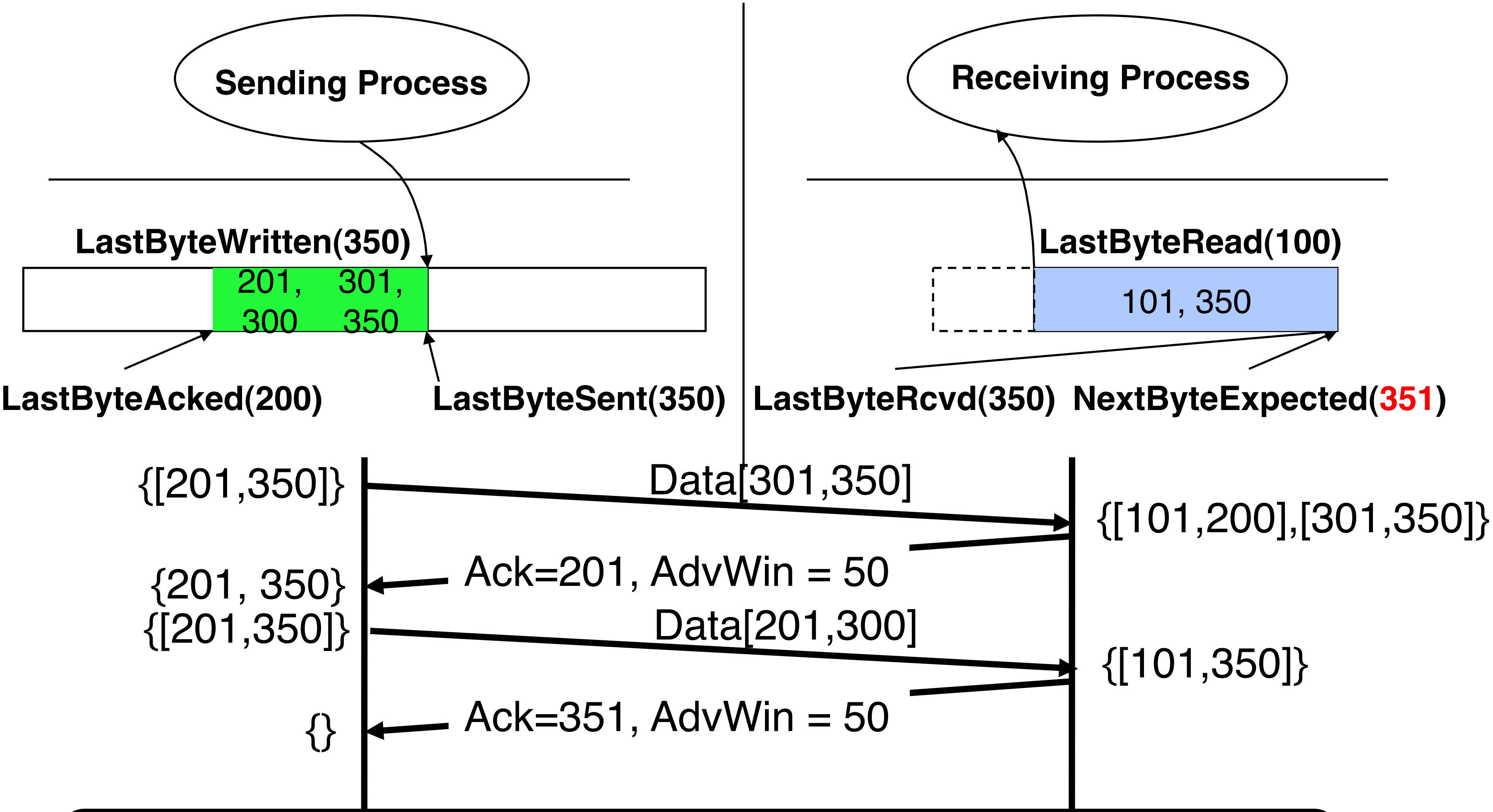
Yes! Sender can re-send 2$^{nd}$ packet since it’s in existing window – won’t cause receiver window to grow

# TCP Flow Control



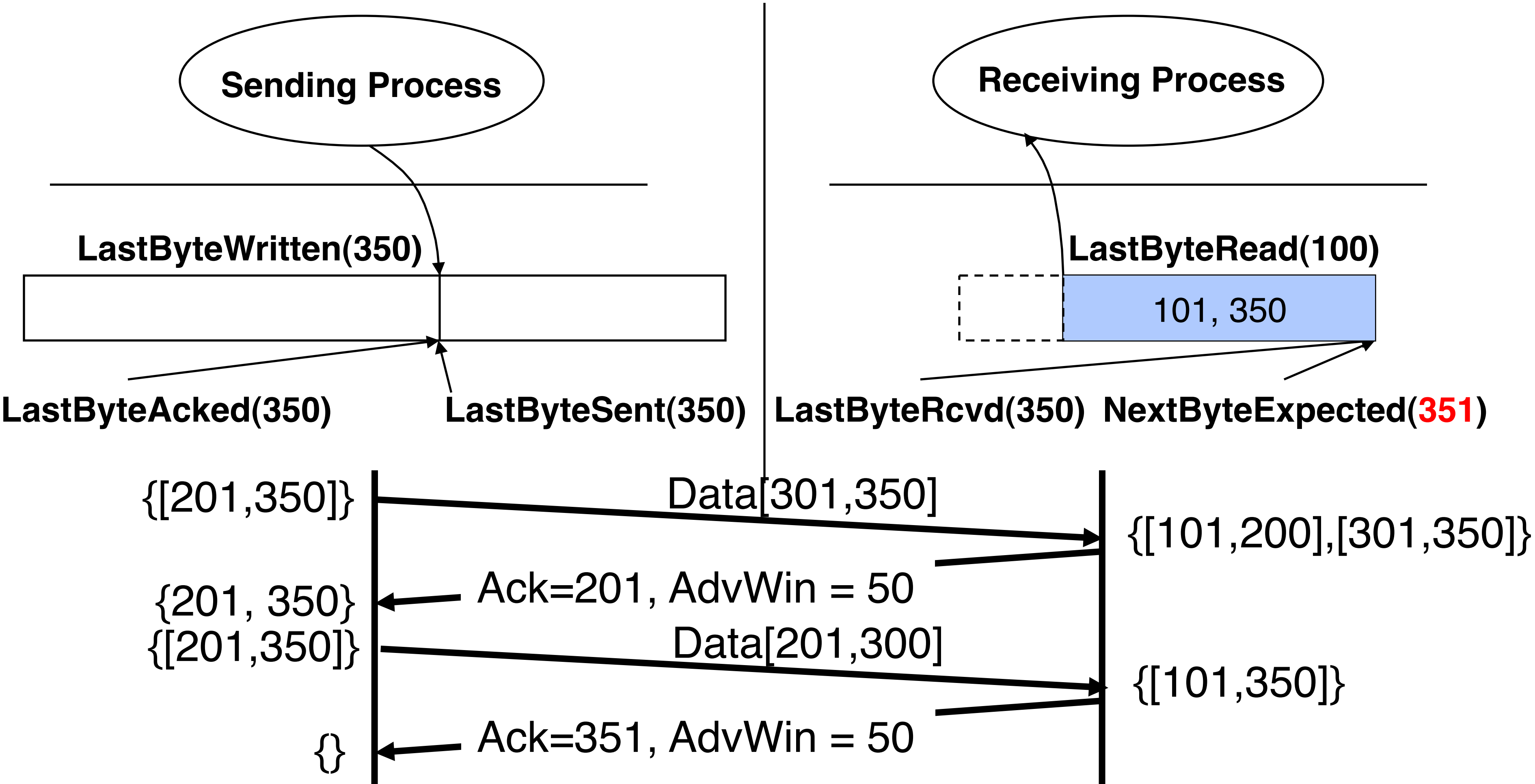
Yes! Sender can re-send 2nd packet since it's in existing window – won't cause receiver window to grow

# TCP Flow Control



- Sender gets 3$^{rd}$ packet and sends Ack for 351
- AdvWin = 50

# TCP Flow Control



Sender DONE with sending all bytes!

# Discussion

- Why not have a huge buffer at the receiver (memory is cheap!)?

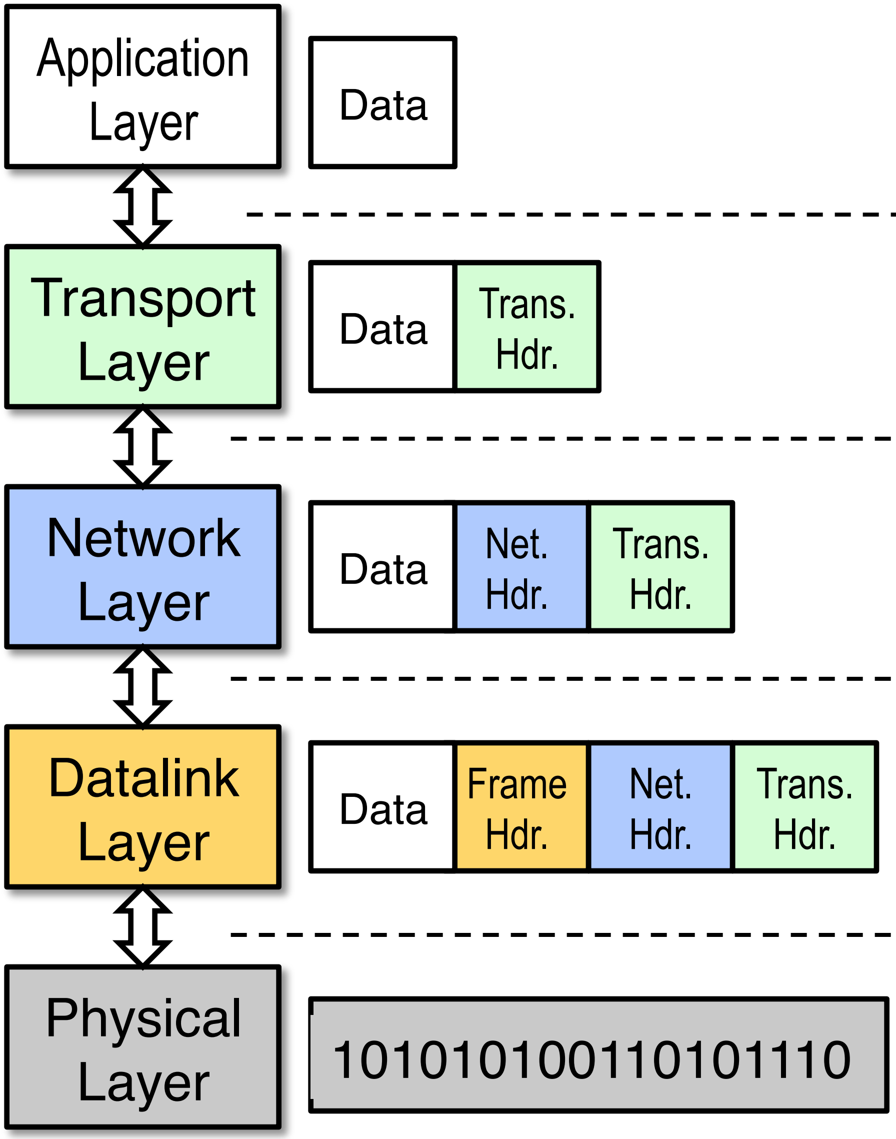
- Sending window (SndWnd) also depends on network congestion
  - **Congestion control**: ensure that  a fast sender doesn't overwhelm a router in the network (discussed in detail in EE122)

- In practice there is another set of buffers in the protocol stack, at the **link layer** (i.e., Network Interface Card)

# Summary: Reliability & Flow Control

- Flow control: three pairs of producer consumers
  - Sending process → sending TCP
  - Sending TCP → receiving TCP
  - Receiving TCP → receiving process
- AdvertisedWindow: tells sender how much new data the receiver can buffer
- SenderWindow: specifies how many more bytes the sending application can send to the sending OS
  - Depends on AdvertisedWindow and on data sent since sender received AdvertisedWindow

# Summary: Networking (Internet Layering)



Application Layer | Data

Transport Layer | Data | Trans. Hdr.

Network Layer | Data | Net. Hdr. | Trans. Hdr.

Datalink Layer | Data | Frame Hdr. | Net. Hdr. | Trans. Hdr.

Physical Layer | 101010100110101110

Any distributed protocol
(e.g., HTTP, Skype, p2p,
KV protocol in your project)

Send *segments* to another *process* running on same or different node

Send *packets* to another node possibly *located* in a different network

Send *frames* to other node directly connected to same physical network

Send *bits* to other node directly connected to same physical network