TCP Flow Control – an illustration of distributed system thinking

David E. Culler

CS162 – Operating Systems and Systems Programming

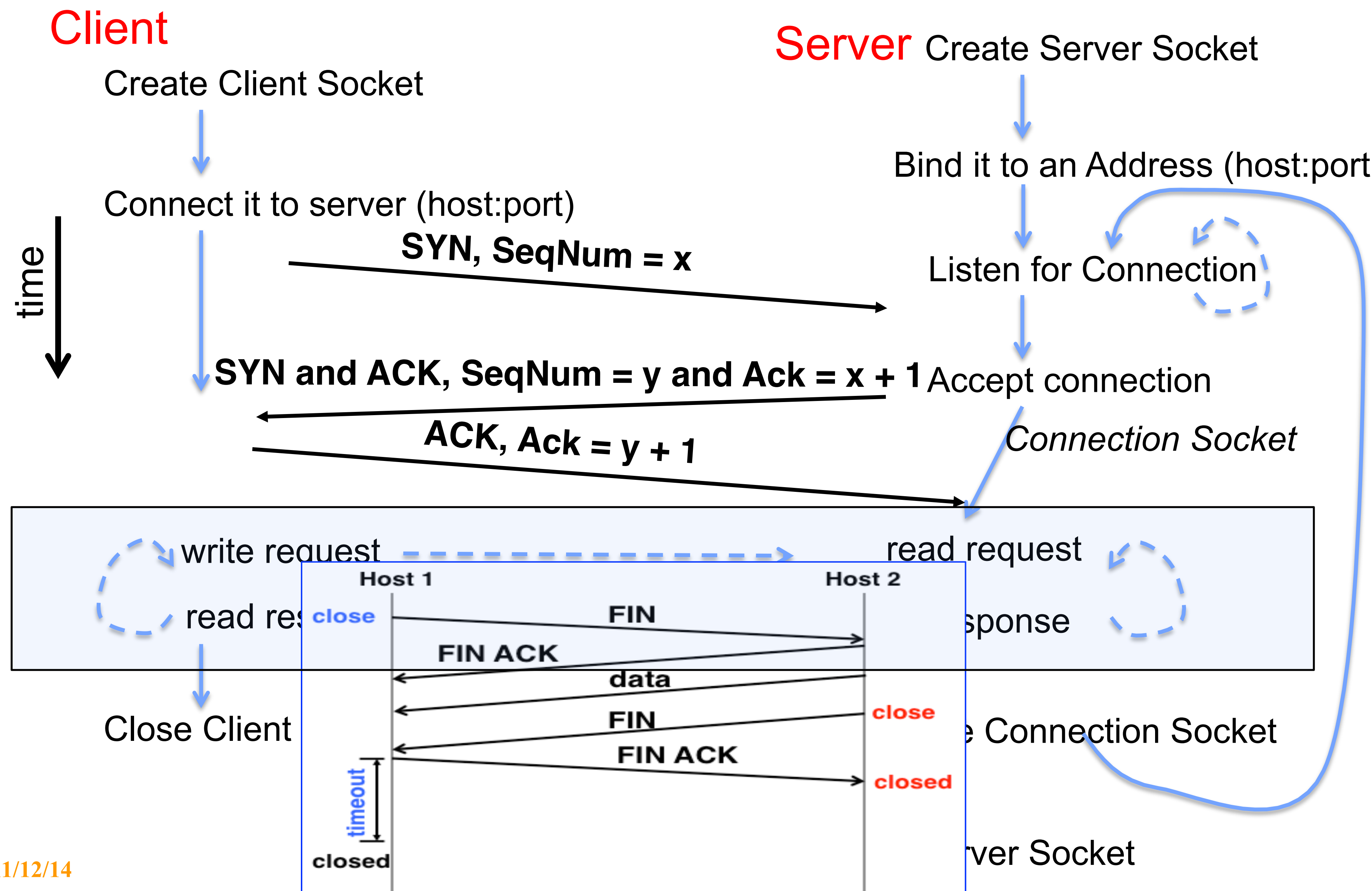
http://cs162.eecs.berkeley.edu/

Lecture 33

Nov 17, 2014

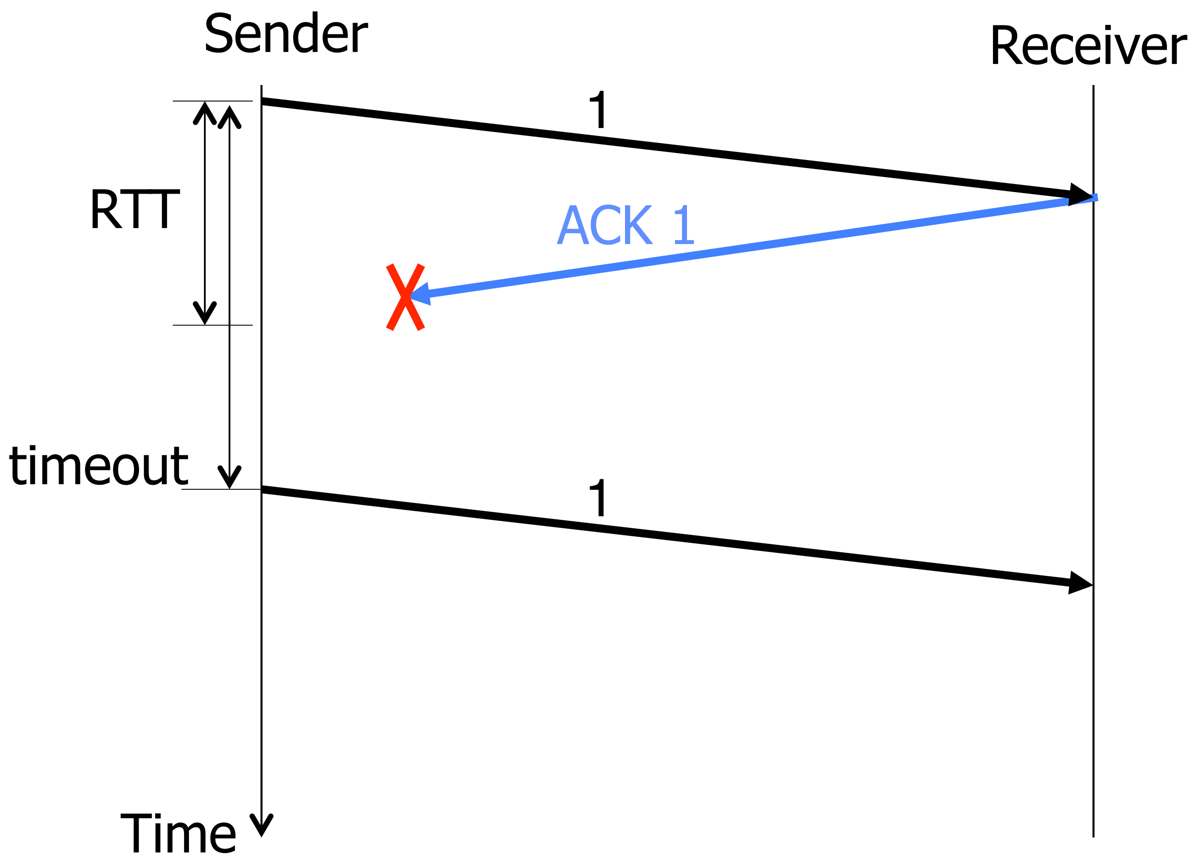
Read: TCP '88

Recall: Connecting API to Protocol



Recall: Stop & Wait with Errors

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



Where we are

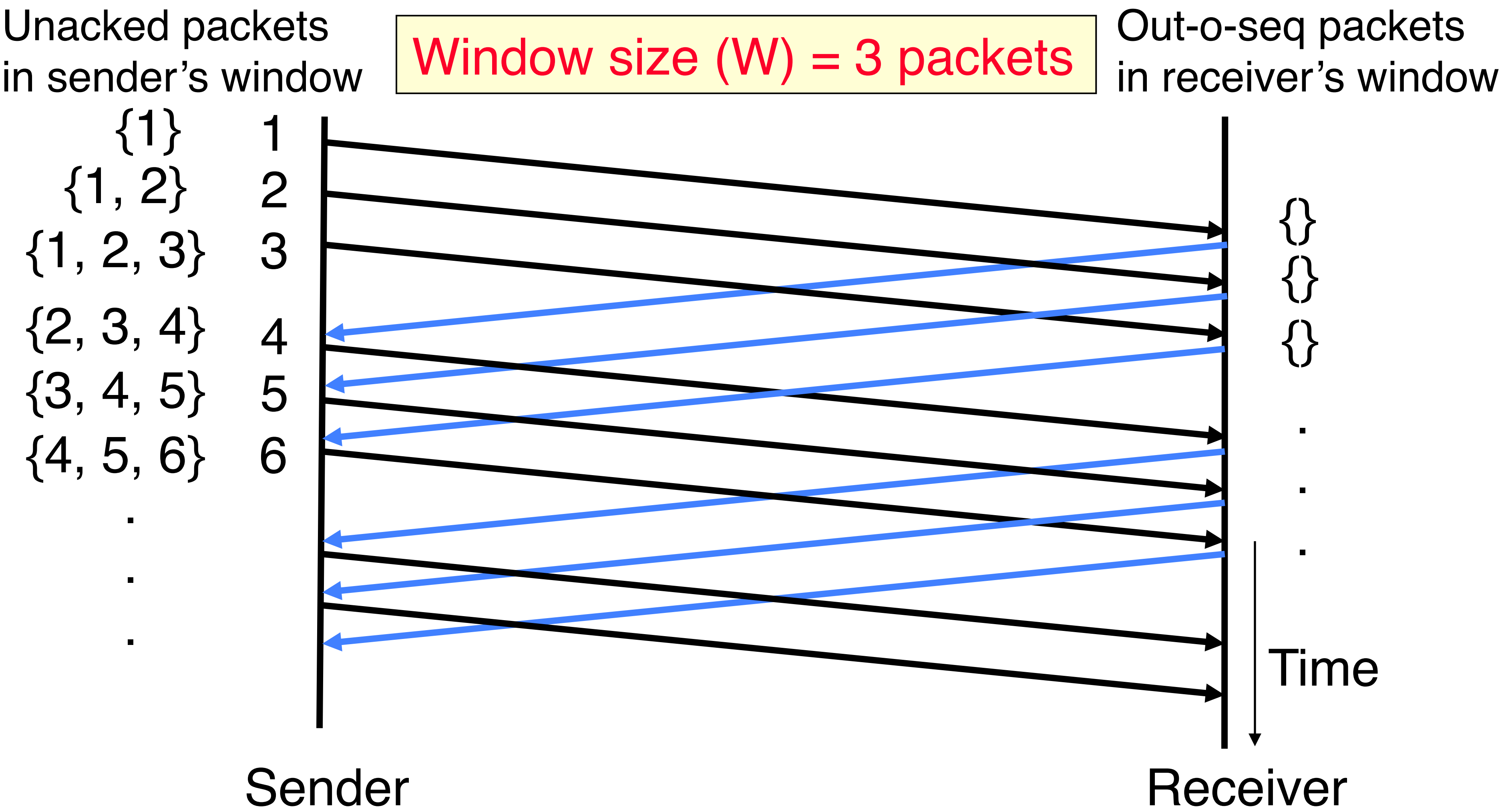
- **TCP: Reliable Byte Stream**
  - Open connection (3-way handshaking)
  - Close connection: no perfect solution; no way for two parties to agree absolutely in the presence of arbitrary message losses (Byzantine General's Problem)
- **Reliable transmission**
  - Stop&Wait not efficient for links with large capacity, i.e., bandwidth-delay product
  - *Sliding window more efficient but more complex*
- **Flow Control**
  - OS on sender and receiver manage buffers
  - Sending rate adjusted according to acks and losses
  - Receiver drops to slow sender on over-run

Recap: 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**



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 = 10^9bps*80*10^{-3}s/(8000b) = 10^4 packets

Bandwidth-Delay Product

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

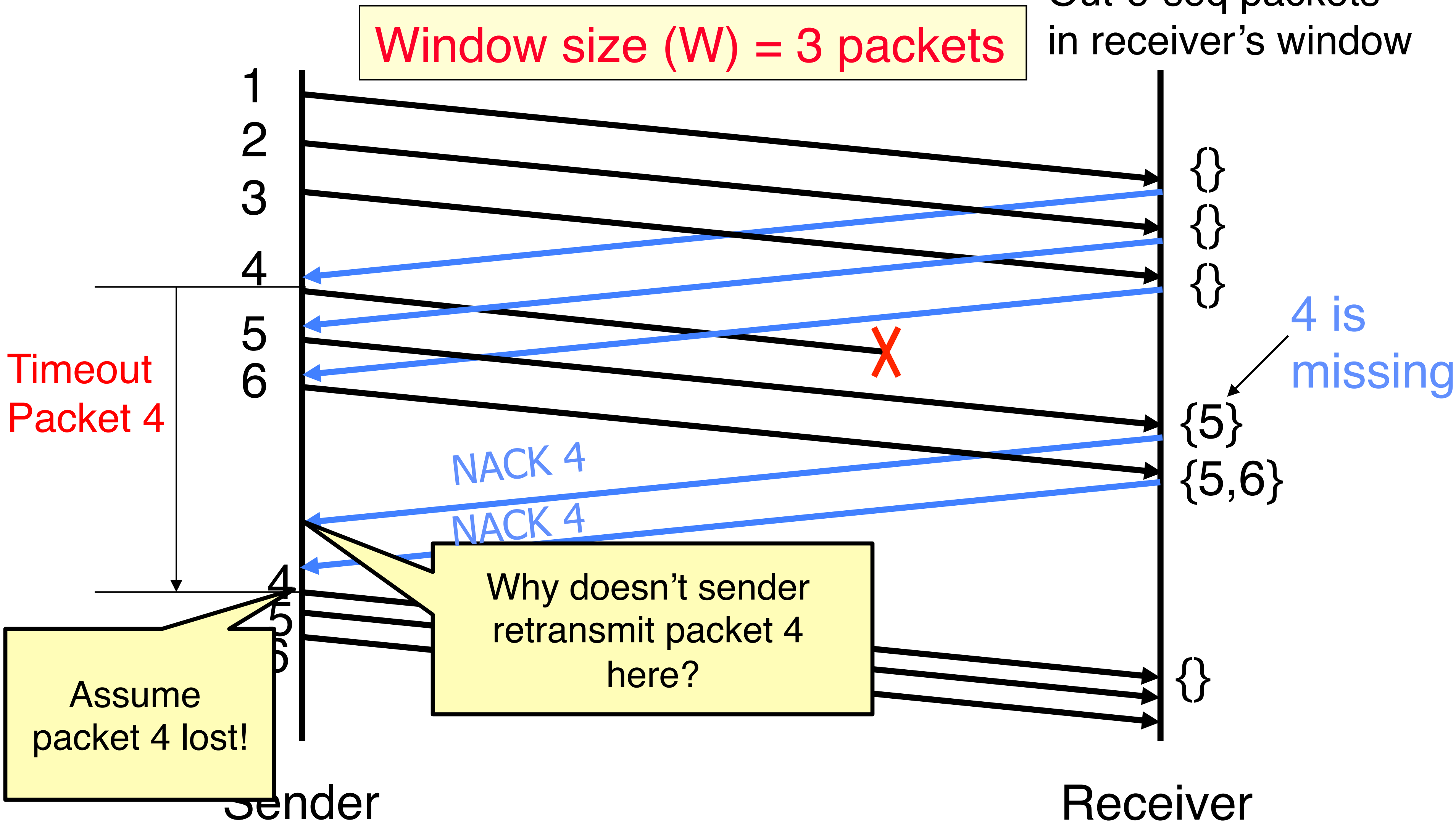
Remember Little's Law !

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

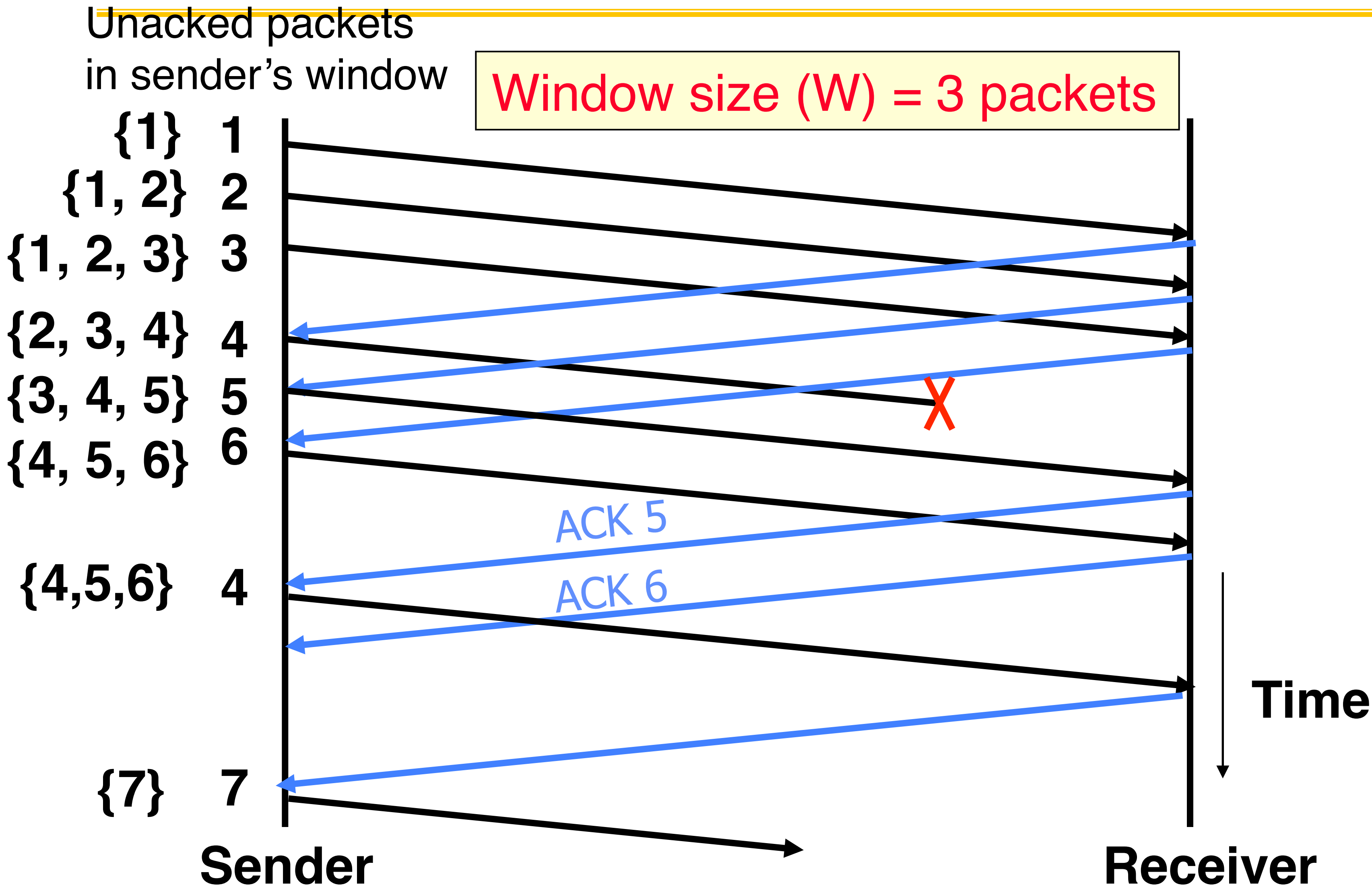
GBN Example with Errors



Selective Repeat (SR)

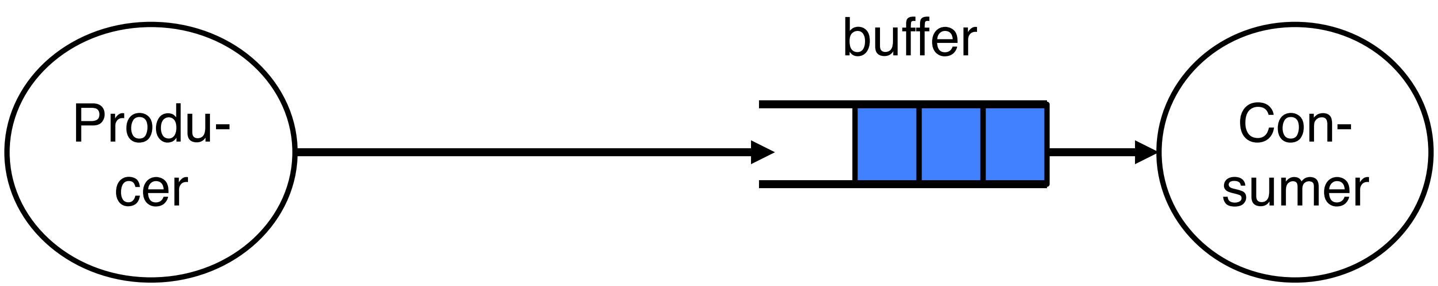
- **Sender: transmit up to *n* unacknowledged packets**
- **Assume packet *k* is lost**
- **Receiver: indicate packet *k* is missing (use ACKs)**
- **Sender: retransmit packet *k***

SR Example with Errors

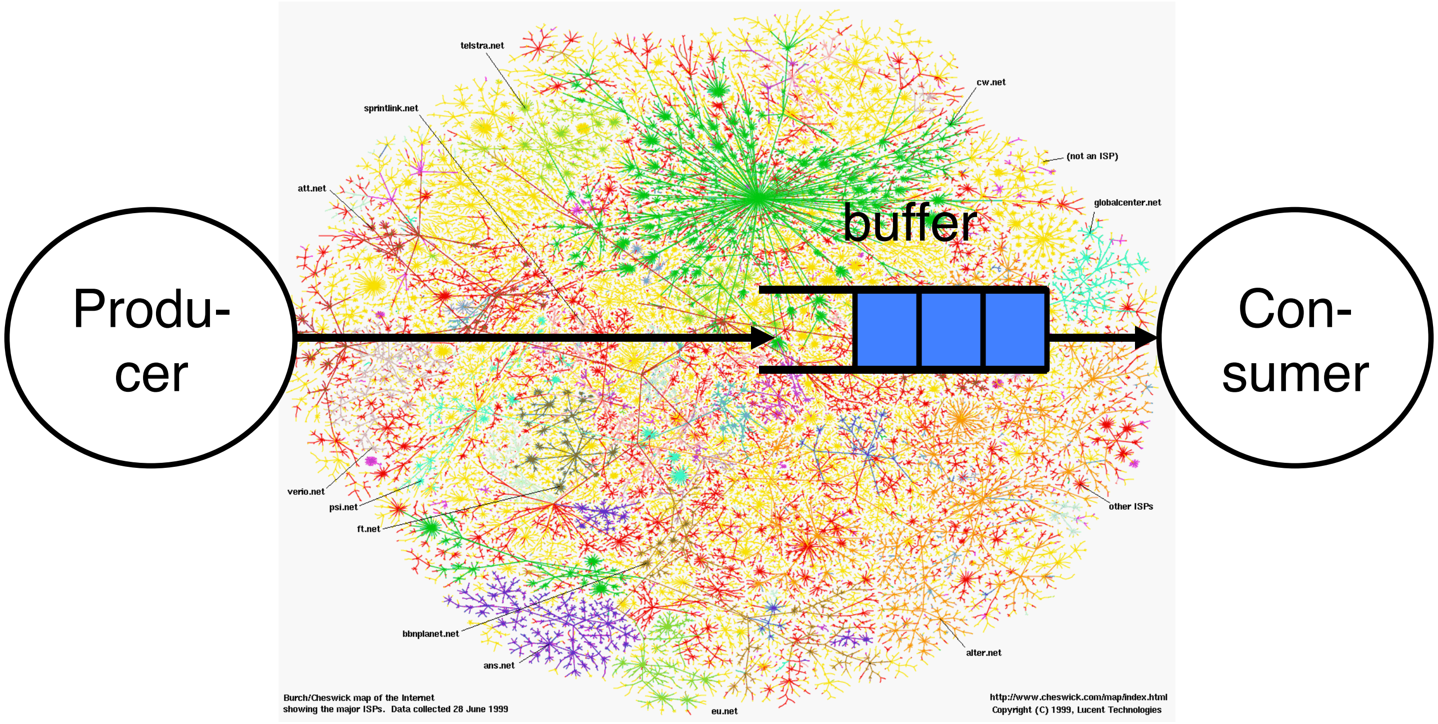


Flow Control

- **Recall: Flow control ensures a fast sender does not overwhelm a slow receiver**
- **Example: Producer-consumer with bounded buffer**
  - **A buffer between producer and consumer**
  - **Producer puts items into buffer as long as buffer not full**
  - **Consumer consumes items from buffer**
- **Recall: solutions on one machine using locks, etc.**



The Distributed Case



- **Think Globally – Act Locally**

When the Internet was young …

Congestion Avoidance and Control

V. Jacobson

(Originally Published in: Proc. SIGCOMM '88, Vol 18 No. 4, August 1988)

In October of '86, the Internet had the first of what became a series of 'congestion collapses'. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and three IMP hops) dropped from 32 Kbps to 40 bps. Mike Karels[1] and I were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad. We wondered, in particular, if the 4.3BSD (Berkeley UNIX) TCP was mis-behaving or if it could be tuned to work better under abysmal network conditions. The answer to both of these questions was "yes".

Since that time, we have put seven new algorithms into the 4BSD TCP:

(*i*) round-trip-time variance estimation

(*ii*) exponential retransmit timer backoff

(*iii*) slow-start

(*iv*) more aggressive receiver ack policy

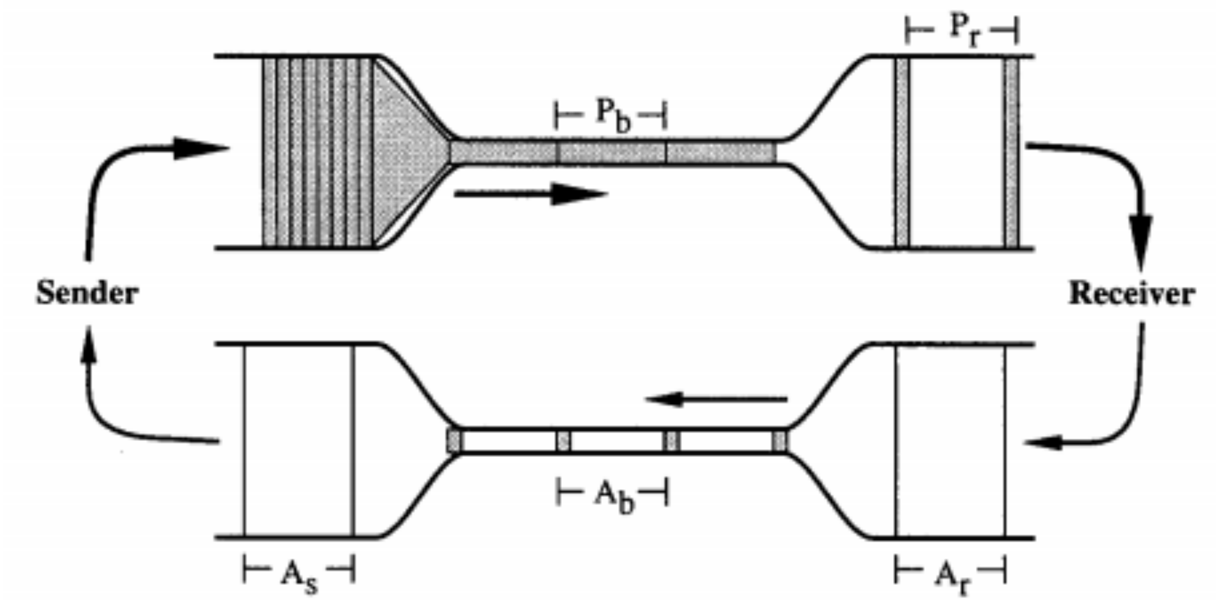
(*v*) dynamic window sizing on congestion

(*vi*) Karn's clamped retransmit backoff

(*vii*) fast retransmit

Our measurements and the reports of beta testers suggest that the final product is fairly good at dealing with congested conditions on the Internet.

Van Jacobson's Concept

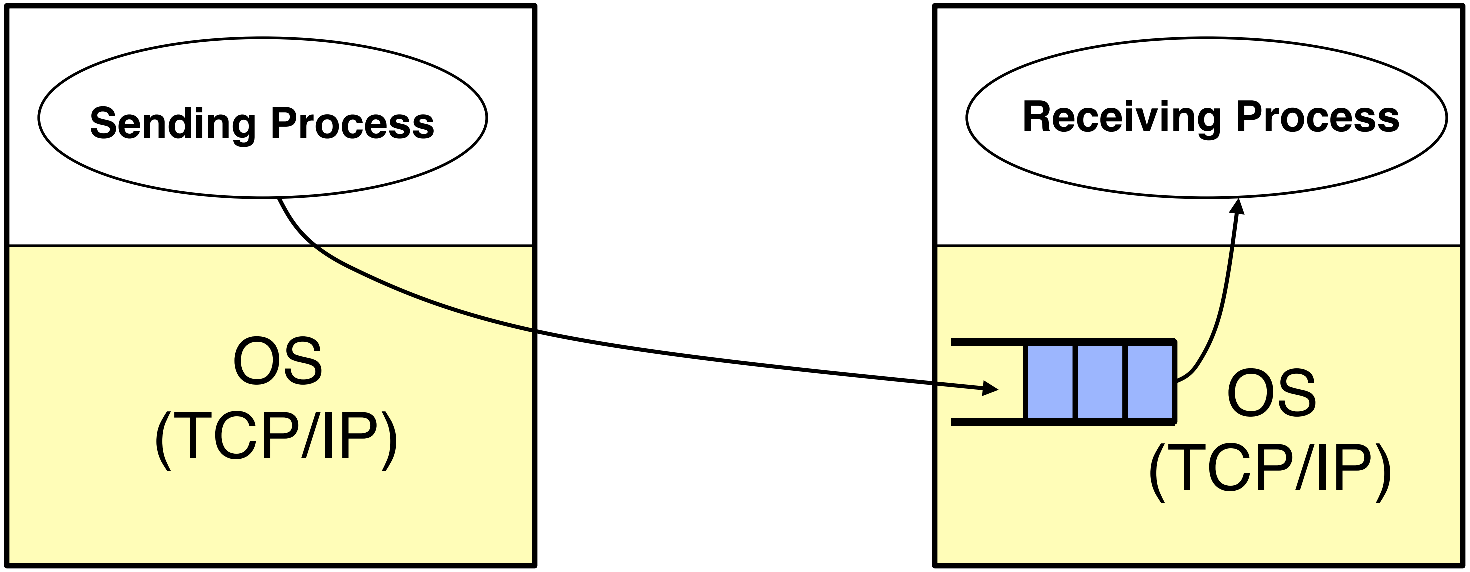


- Packets get "space out" going through bottleneck
- Sender learns this spacing (rate) from ack timing
- Loss is due primarily to congestion, including receiver over-run
- Start slow and continually increase rate, but …
- Slow-down in response to loss

TCP Flow Control

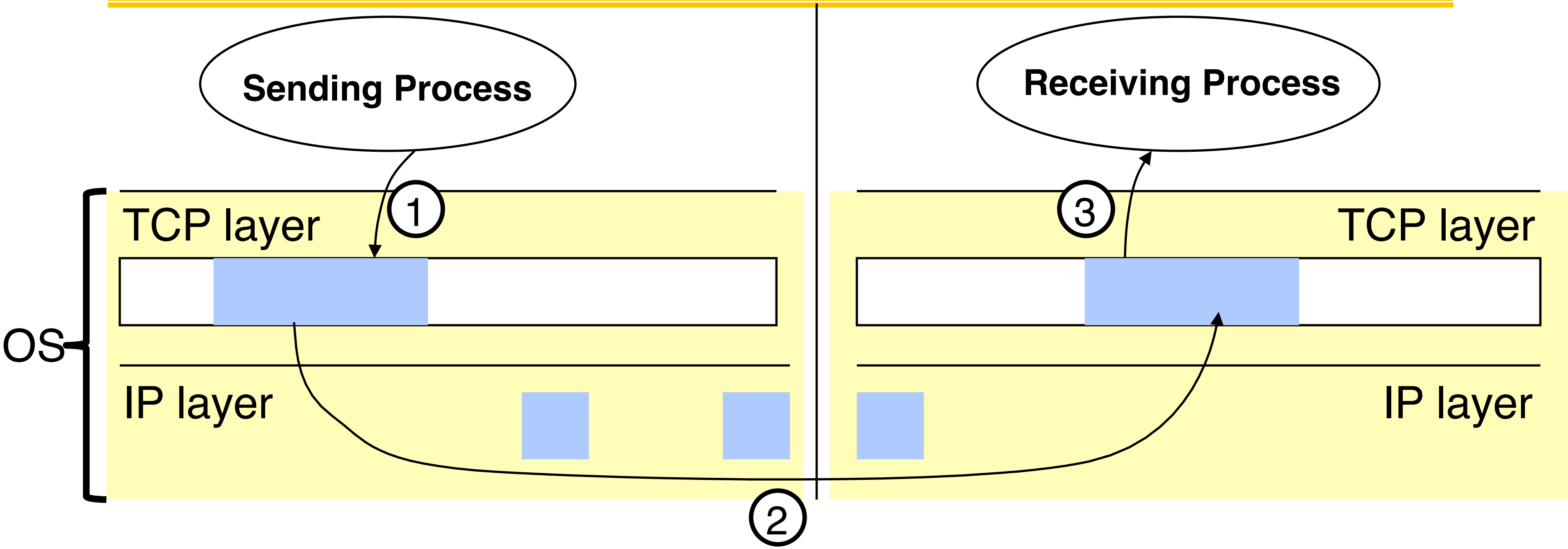
- TCP: sliding window protocol at byte (not packet) level
  - Go-back-N: TCP Tahoe, Reno, New Reno
  - Selective Repeat (SR): TCP Sack
- Receiver tells sender how many more bytes it can receive without overflowing its buffer
  - the AdvertisedWindow
- The ACK contains sequence number N of next byte the receiver expects,
  - receiver has received all bytes in sequence up to and including N-1

TCP Flow Control



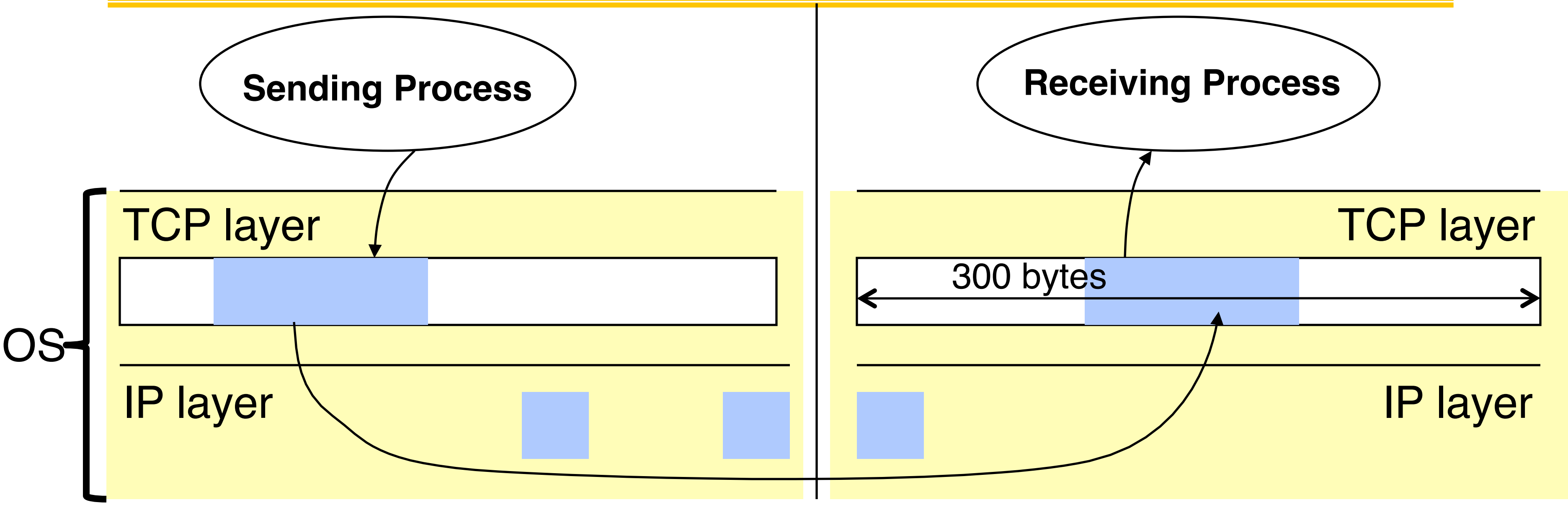
- **TCP/IP implemented by OS (Kernel)**
  - **Cannot do context switching on sending/receiving every packet**
    - » **At 1Gbps, it takes 12 usec to send an 1500 bytes, and 0.8usec to send an 100 byte packet**
- **Need buffers to match …**
  - **sending app with sending TCP**
  - **receiving TCP with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

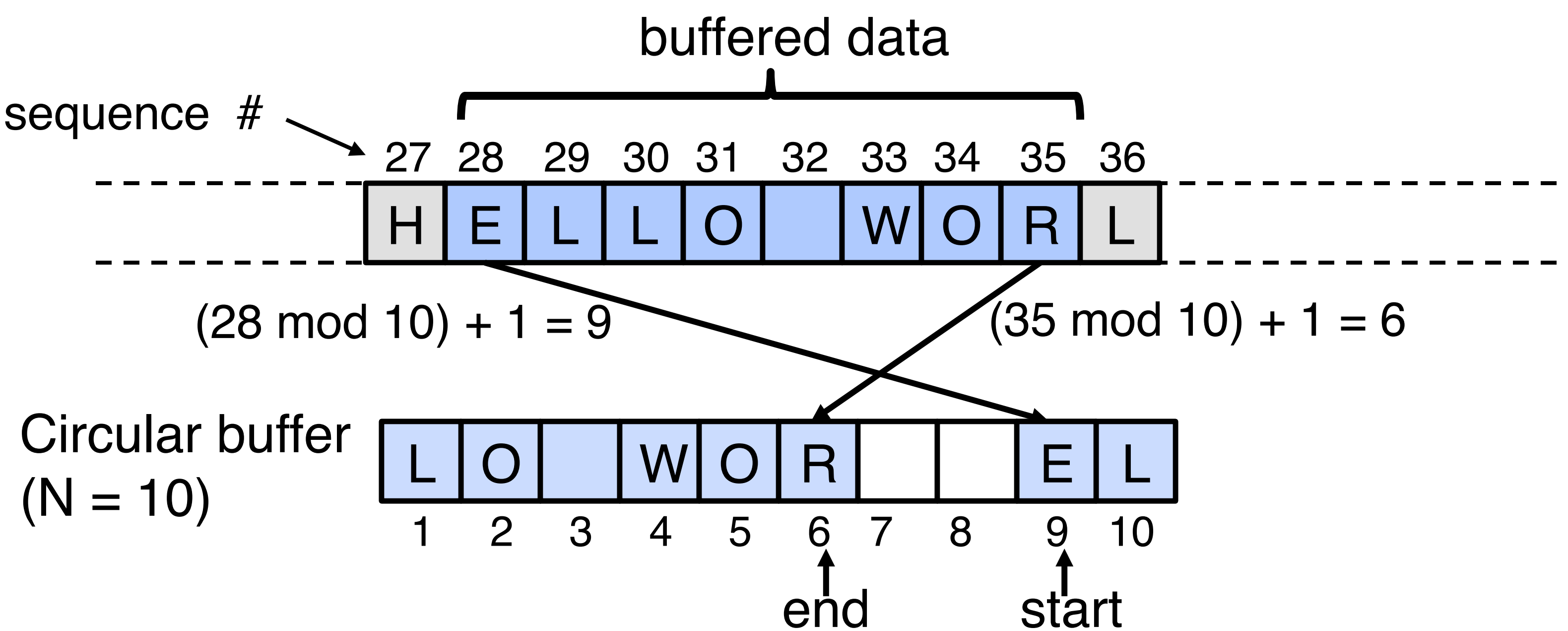
TCP Flow Control



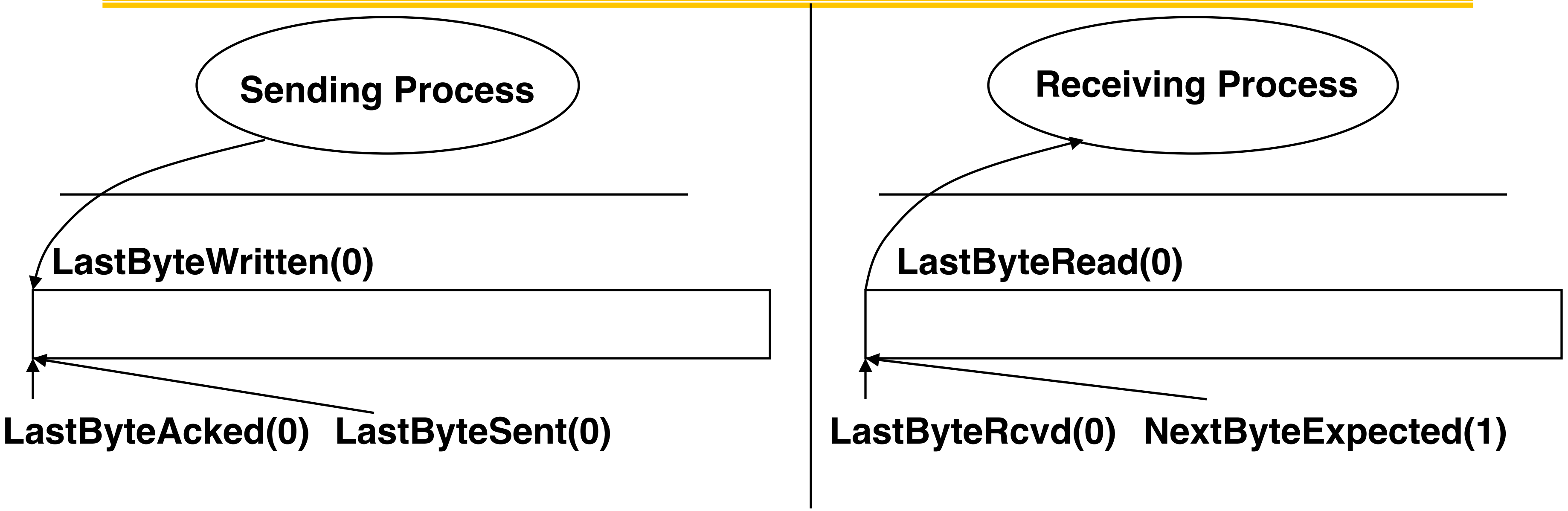
- **Example assumptions:**
  - **Maximum IP packet size = 100 bytes**
  - **Size of the receiving buffer (MaxRcvBuf) = 300 bytes**
- **Recall, ack indicates the next expected byte in-sequence, not the last received byte**
- **Use circular buffers**

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**

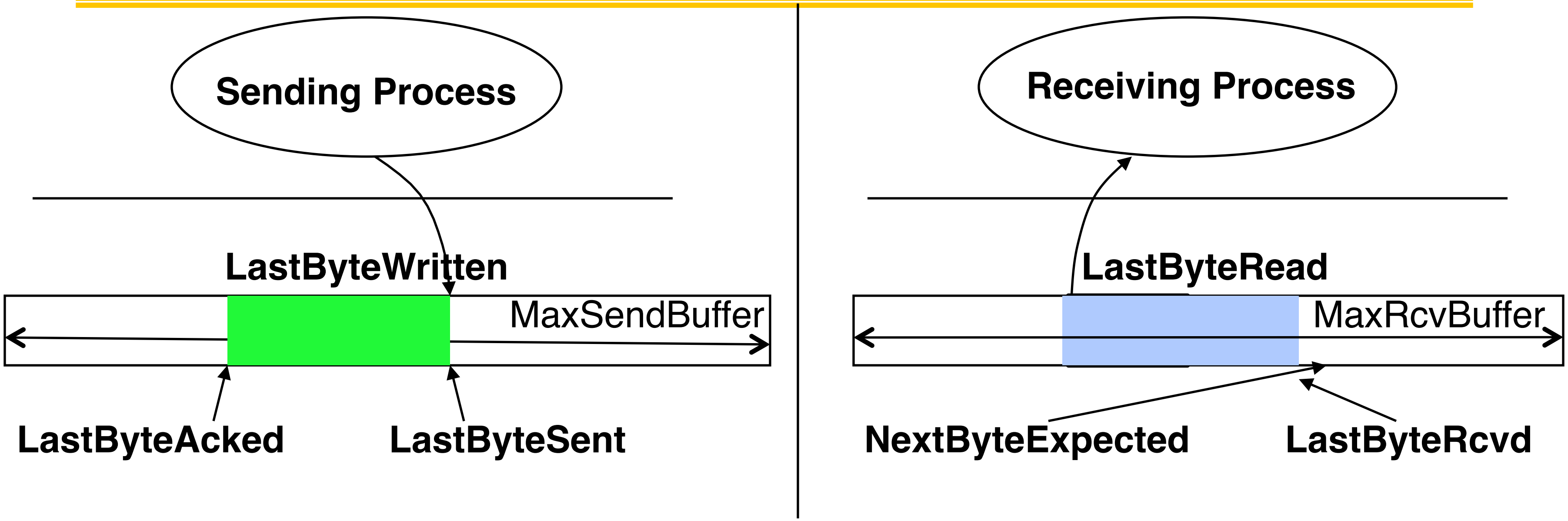


TCP Flow Control



- 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



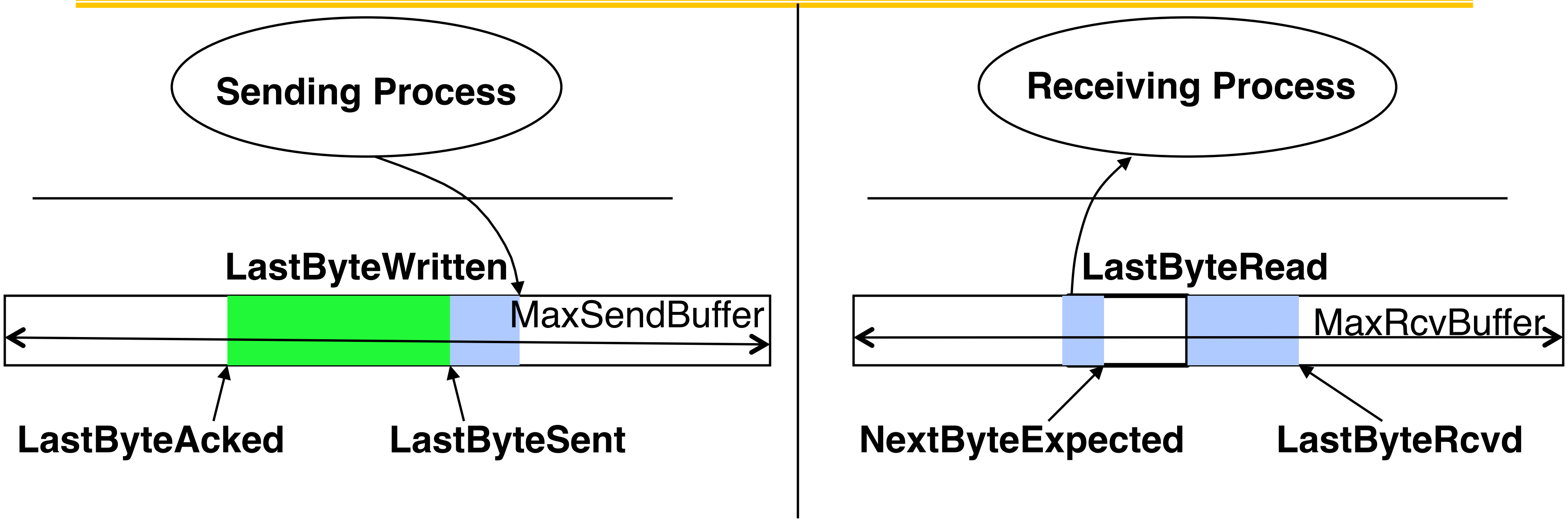
- **AdvertisedWindow: number of bytes TCP receiver can receive**

  AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- **SenderWindow: number of bytes TCP sender can send**

  SenderWindow = AdvertisedWindow – (LastByteSent – LastByteAcked)

TCP Flow Control



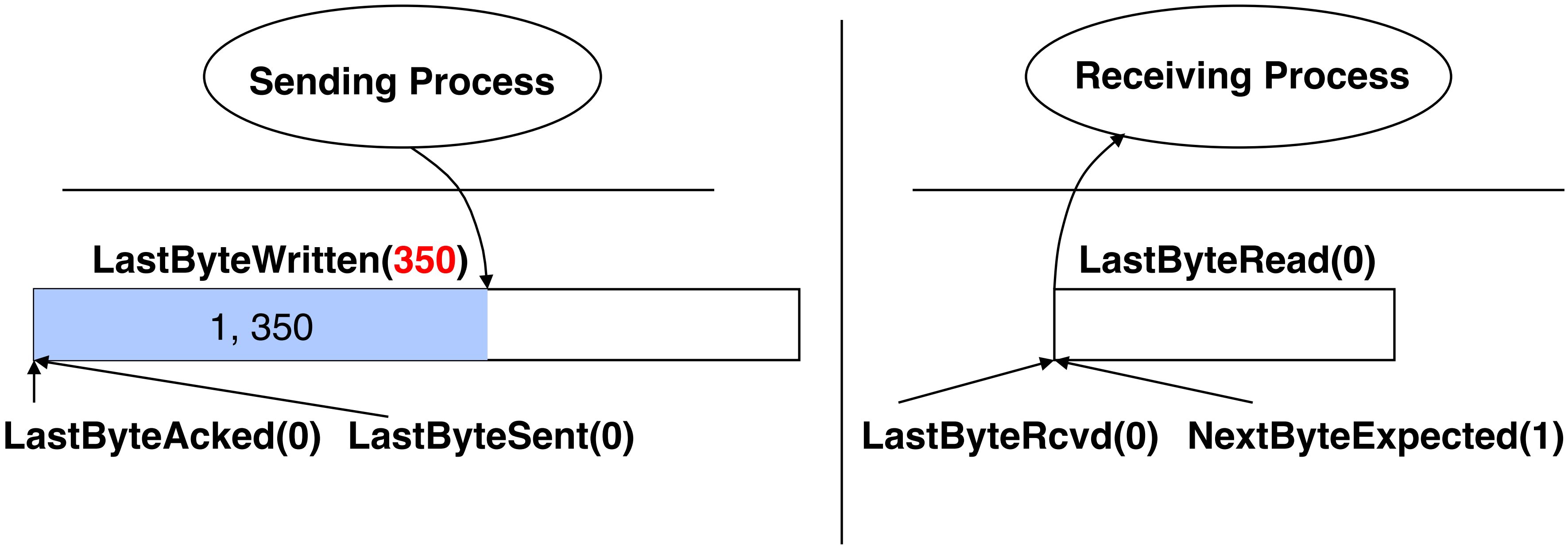
- **Still true if receiver missed data….**

AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

- **WriteWindow: number of bytes sending process can write**

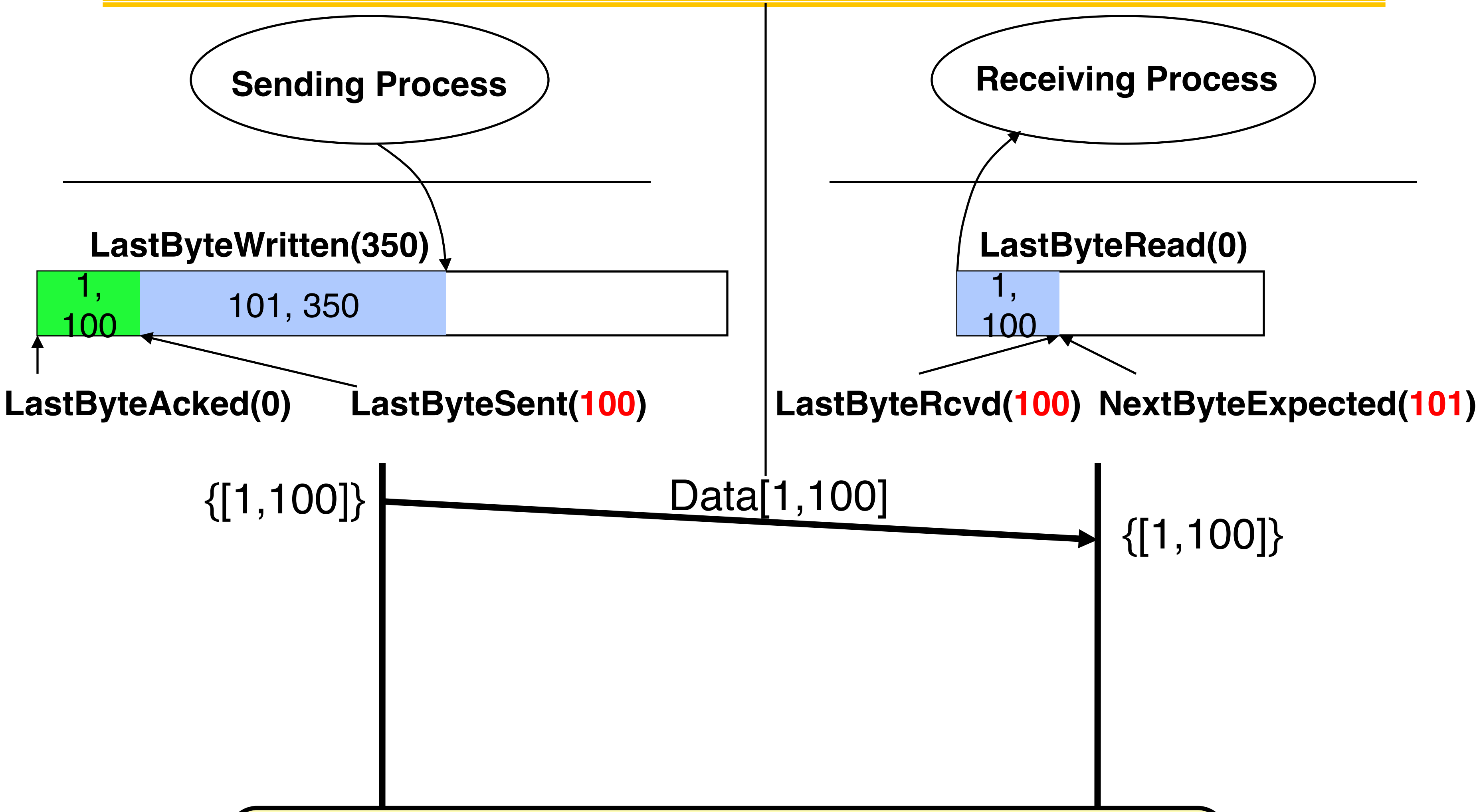
WriteWindow = MaxSendBuffer – (LastByteWritten – LastByteAcked)

TCP Flow Control



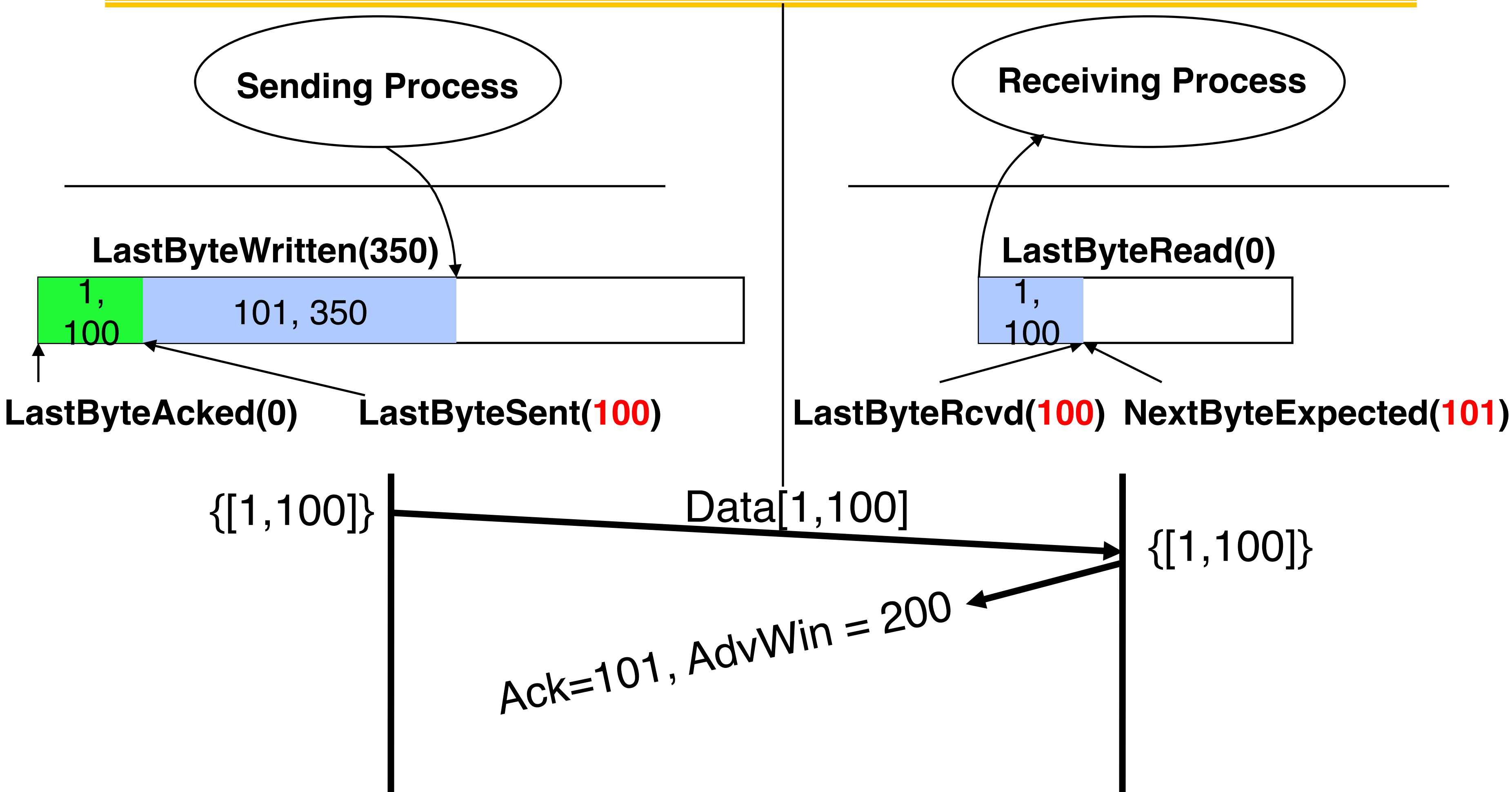
- **Sending app sends 350 bytes**
- **Recall:**
  - **We assume IP only accepts packets no larger than 100 bytes**
  - **MaxRcvBuf = 300 bytes, so initial Advertised Window = 300 byets**

TCP Flow Control



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

TCP Flow Control

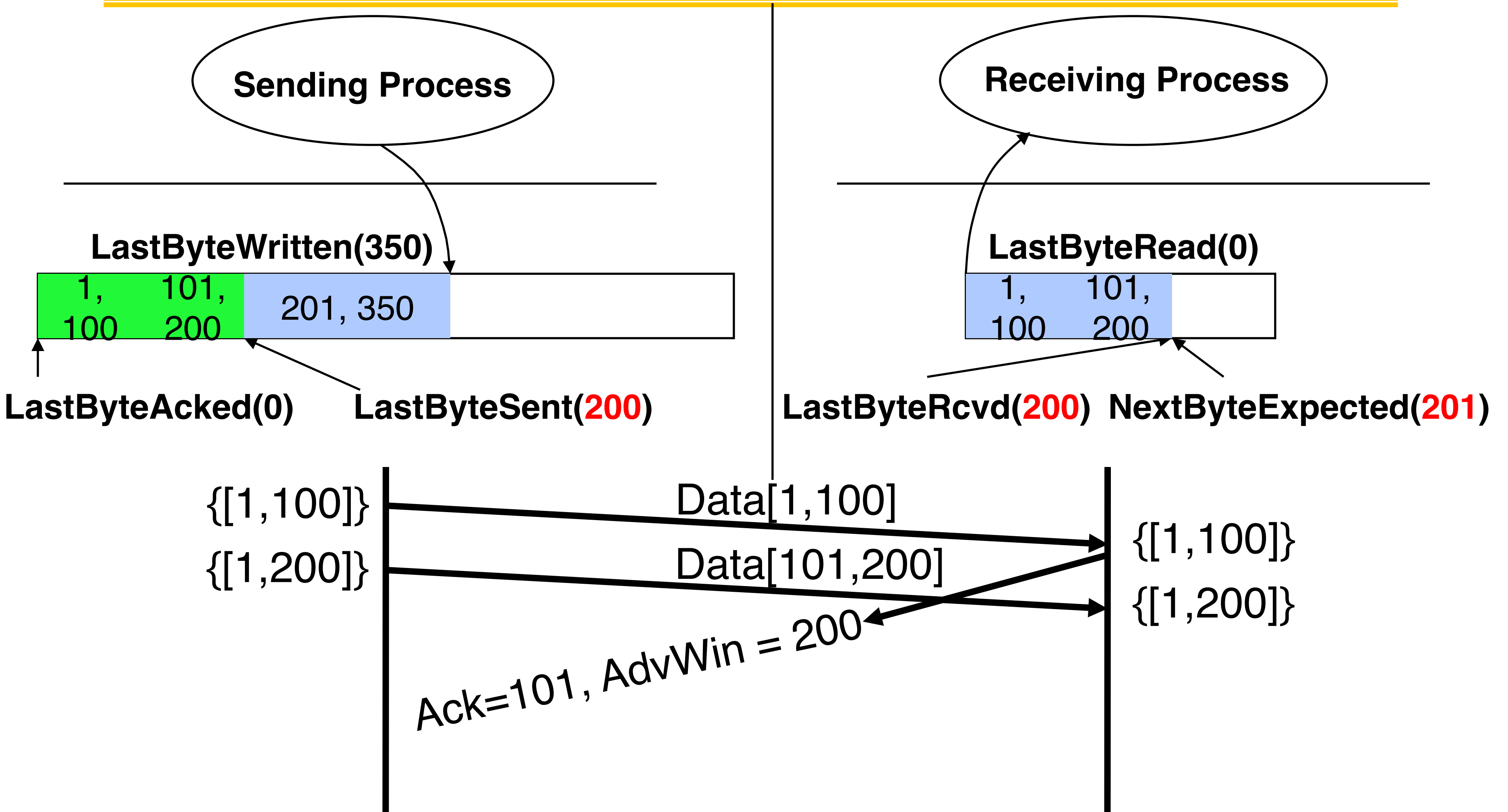


Receiver sends ack for 1st packet

AdvWin = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

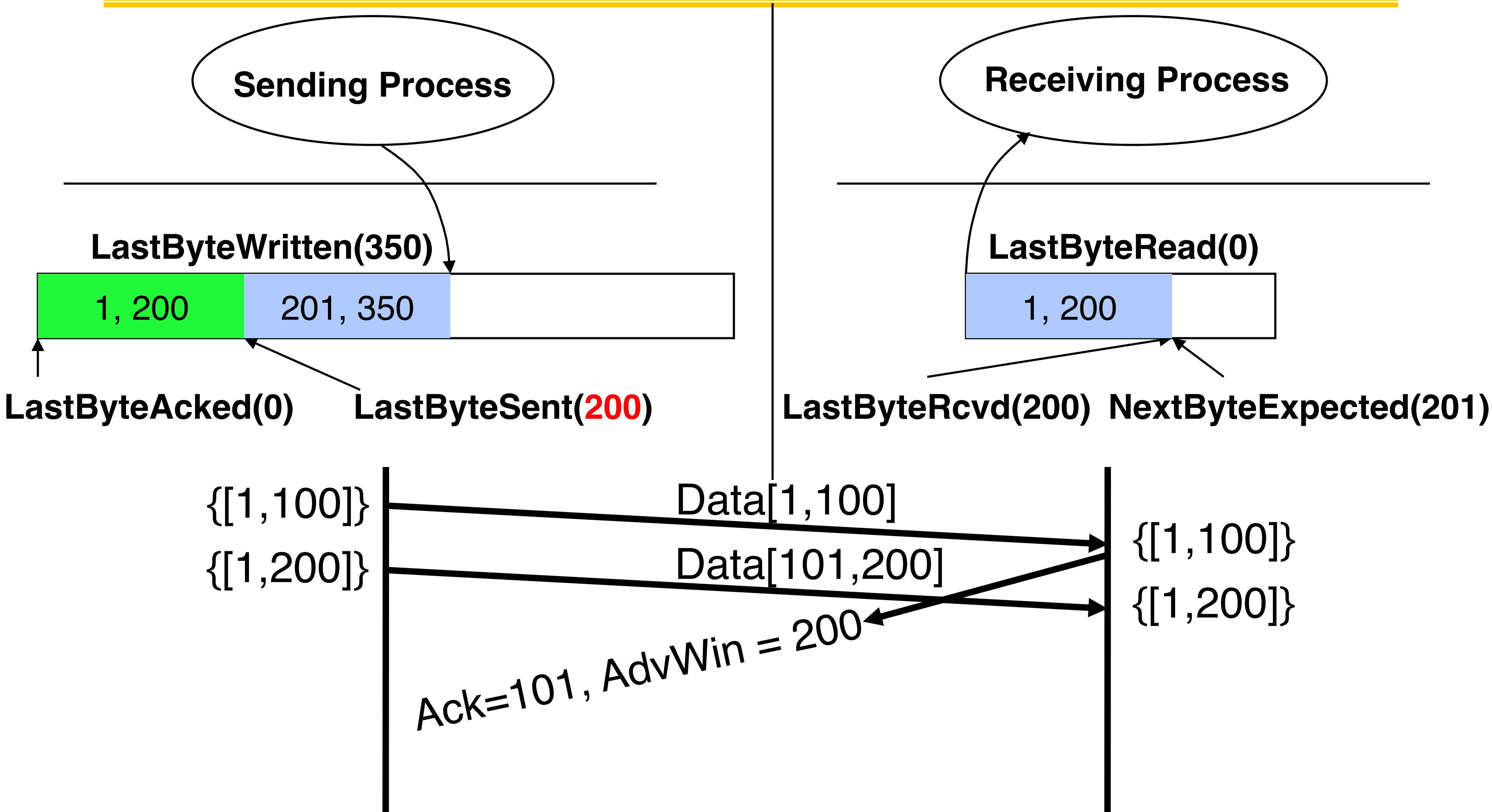
= 300 – (100 – 0) = 200

TCP Flow Control



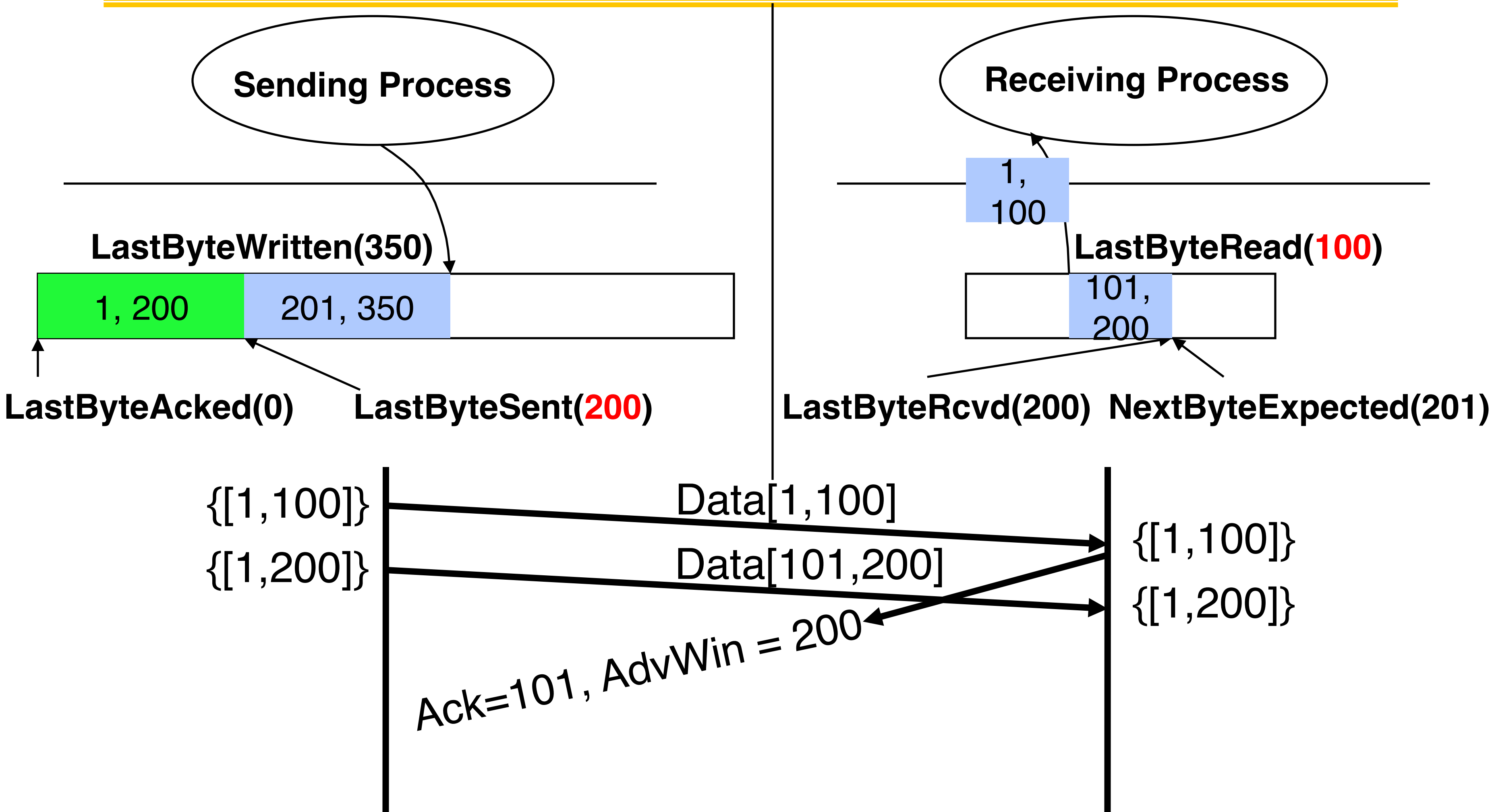
Sender sends 2nd packet (i.e., next 100 bytes) and receiver gets the packet

TCP Flow Control



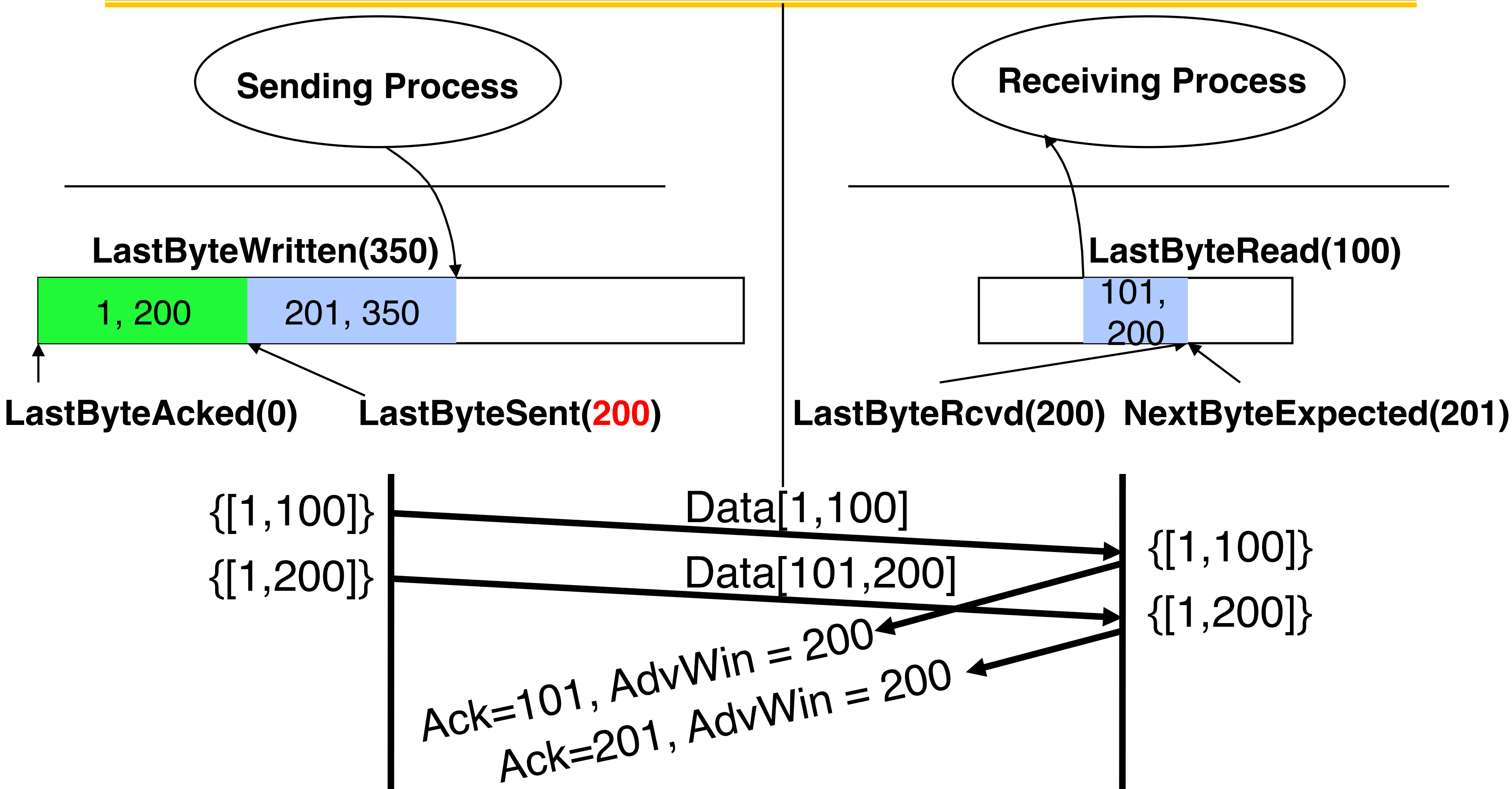
Sender sends 2nd packet (i.e., next 100 bytes) and receiver gets the packet

TCP Flow Control



Receiving TCP delivers first 100 bytes to receiving process

TCP Flow Control

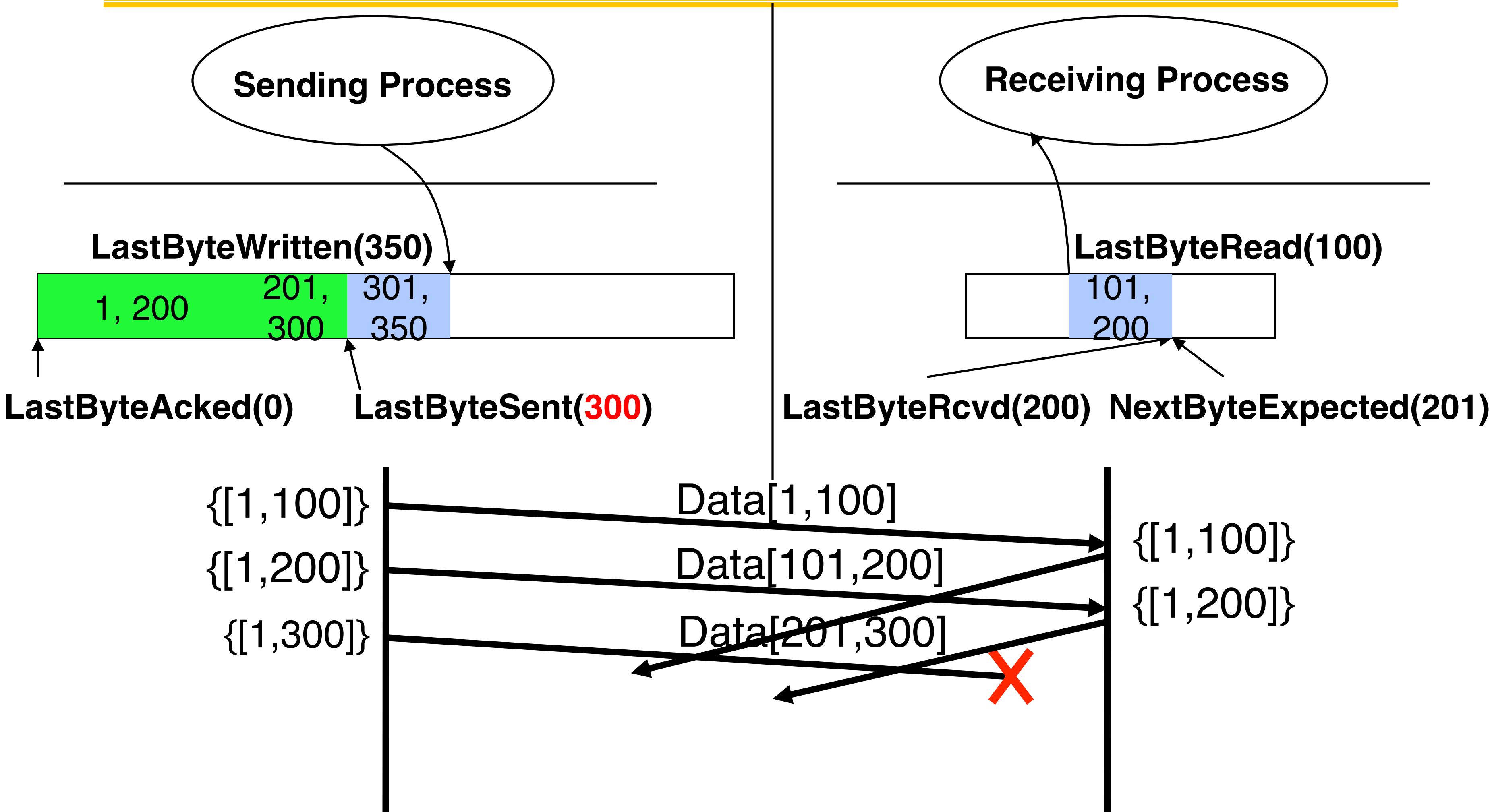


Receiver sends ack for 2nd packet

AdvWin = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

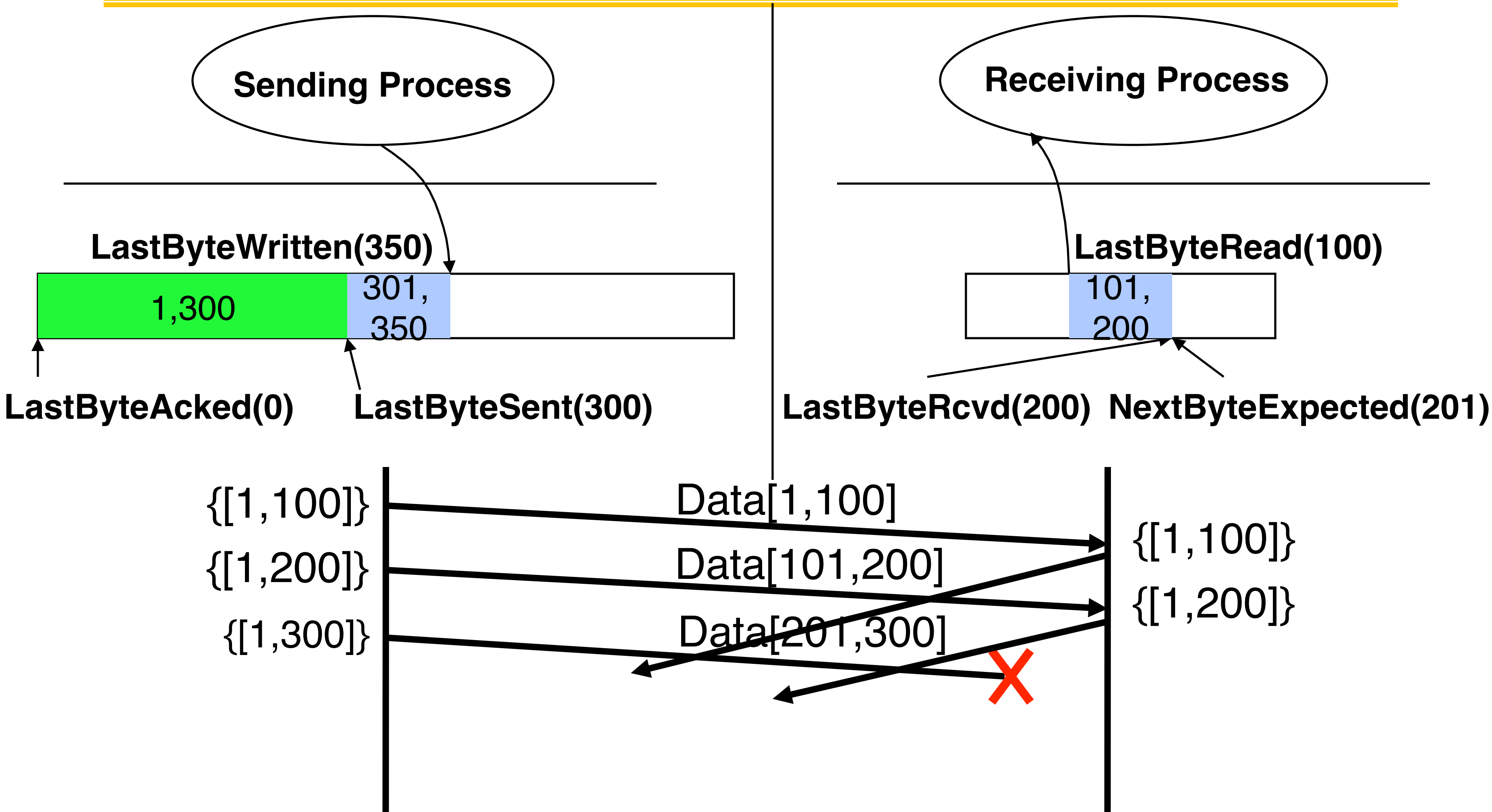
= 300 – (200 – 100) = 200

TCP Flow Control



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

TCP Flow Control

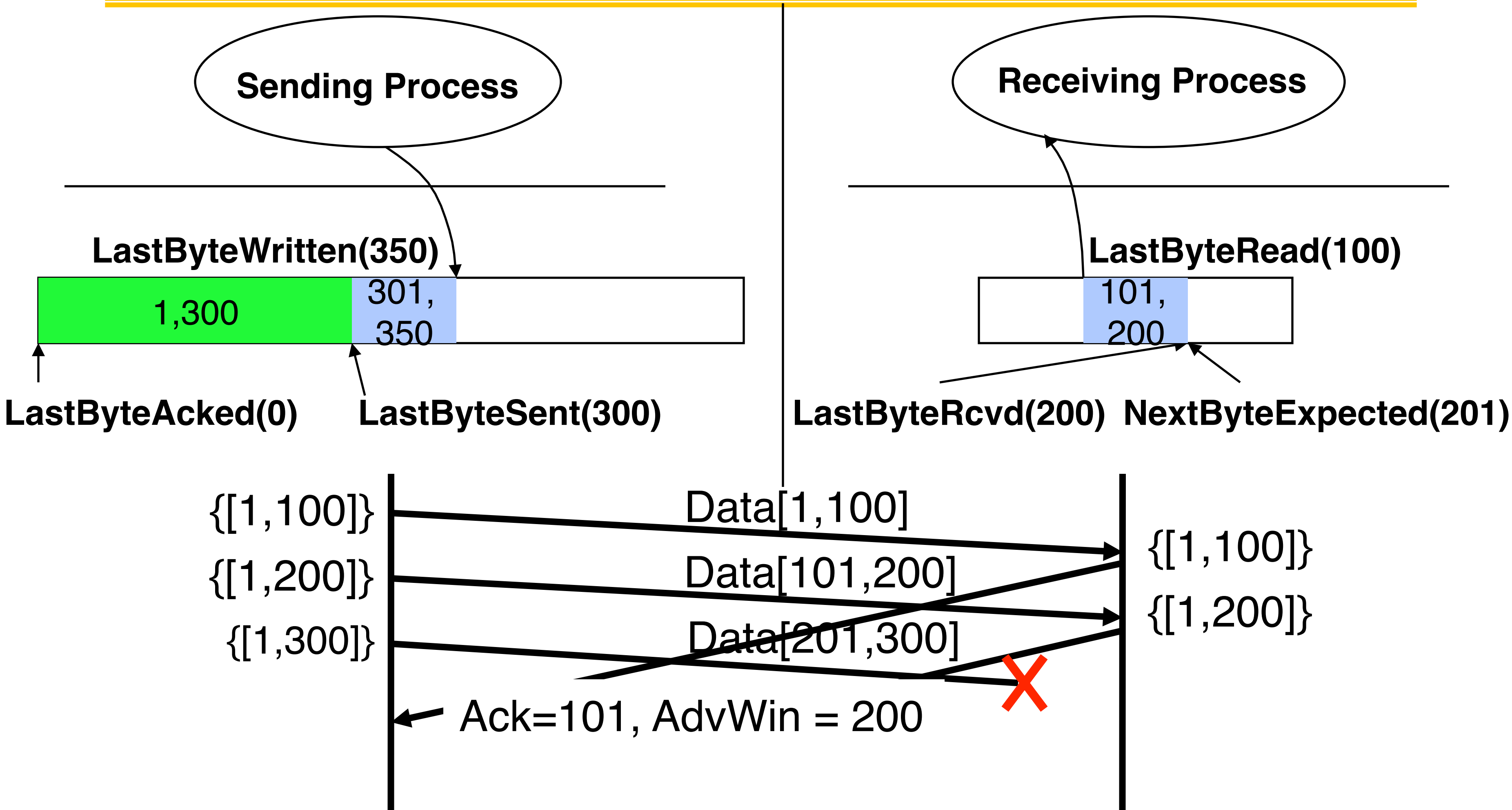


Sender stops sending as window full

SndWin = AdvWin – (LastByteSent – LastByteAcked)

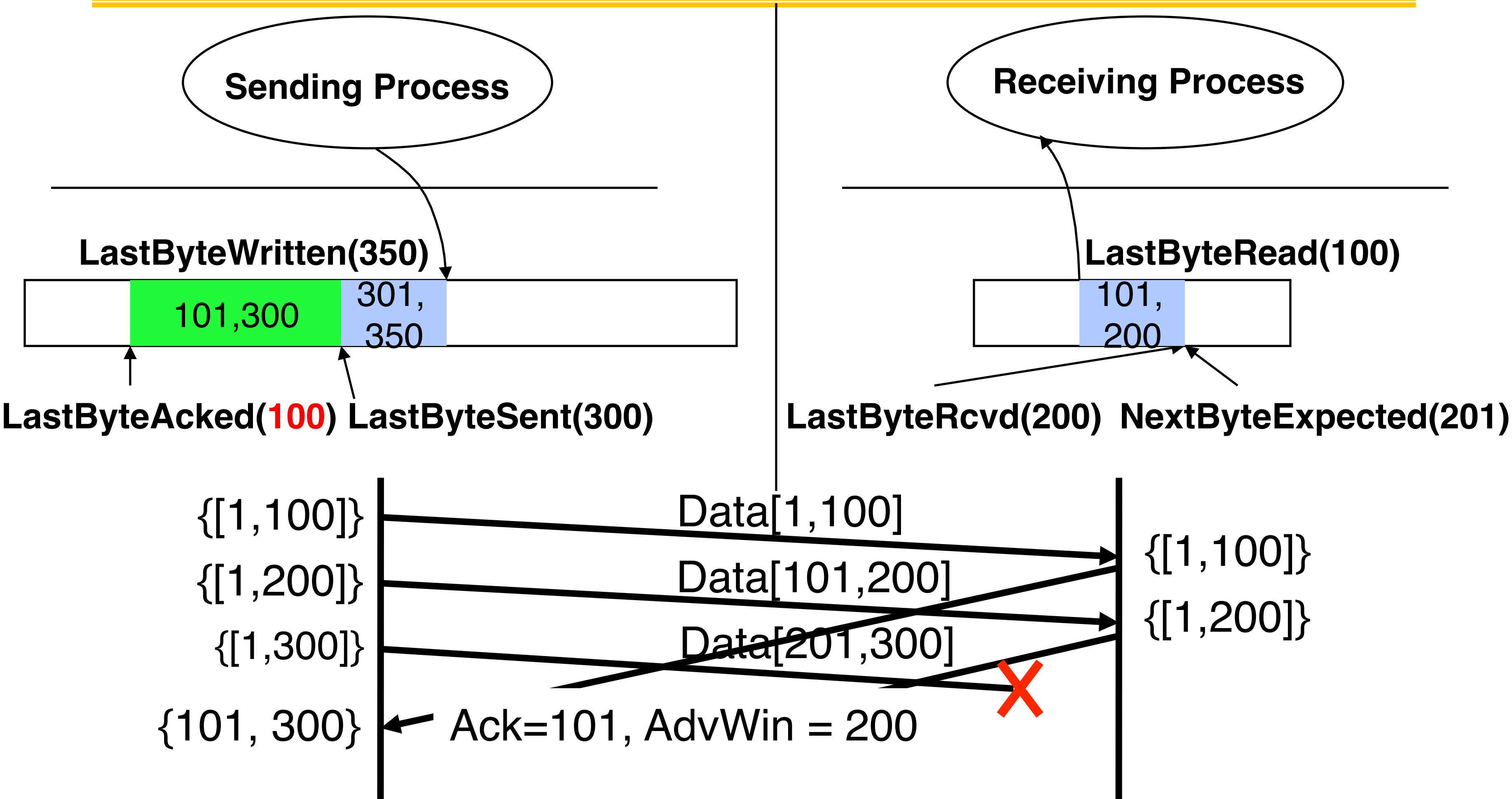
= 300 – (300 – 0) = 0

TCP Flow Control



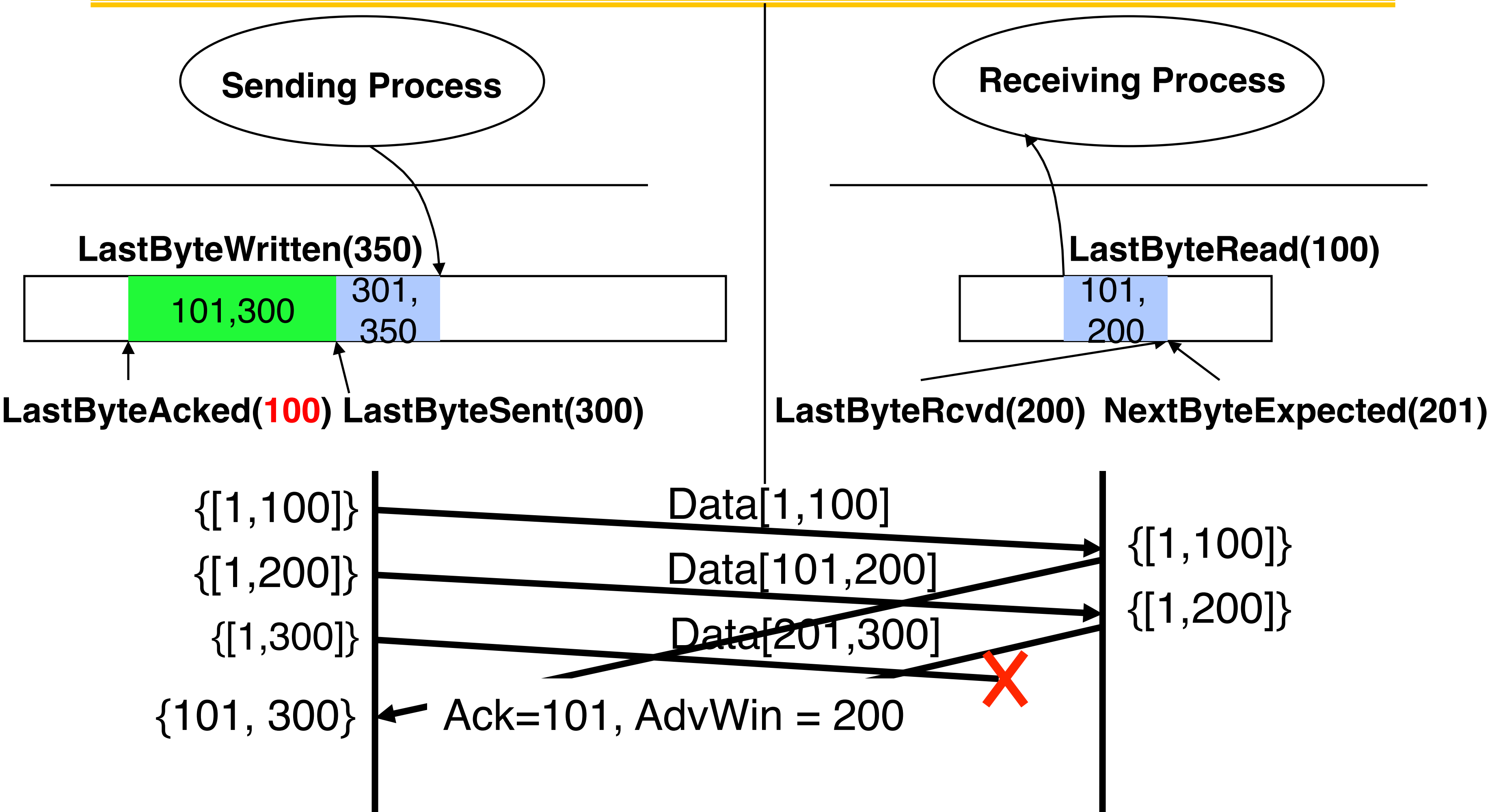
- Sender gets ack for 1st packet
- AdWin = 200

TCP Flow Control



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

TCP Flow Control

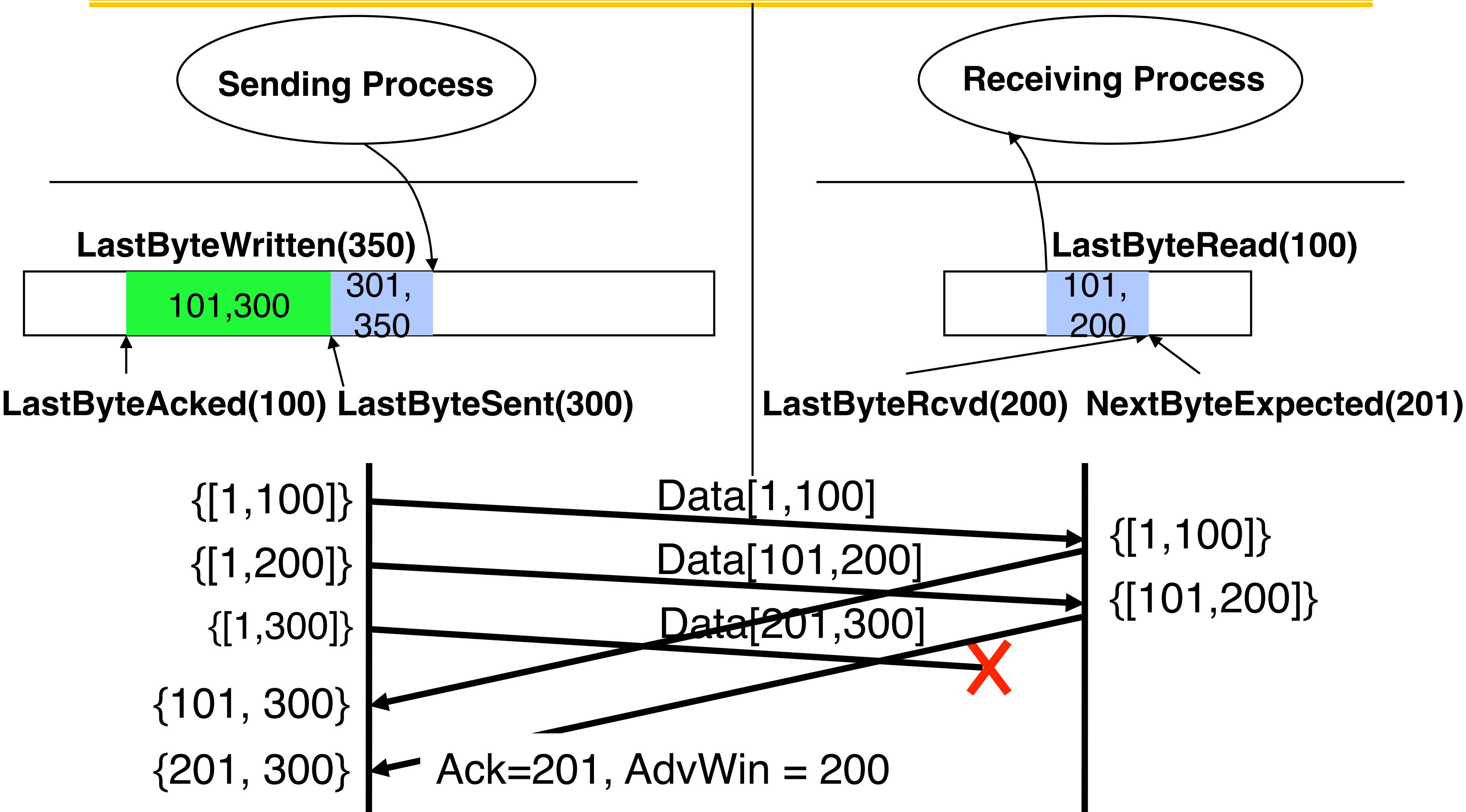


Sender still cannot send as window full

SndWin = AdvWin – (LastByteSent – LastByteAcked)

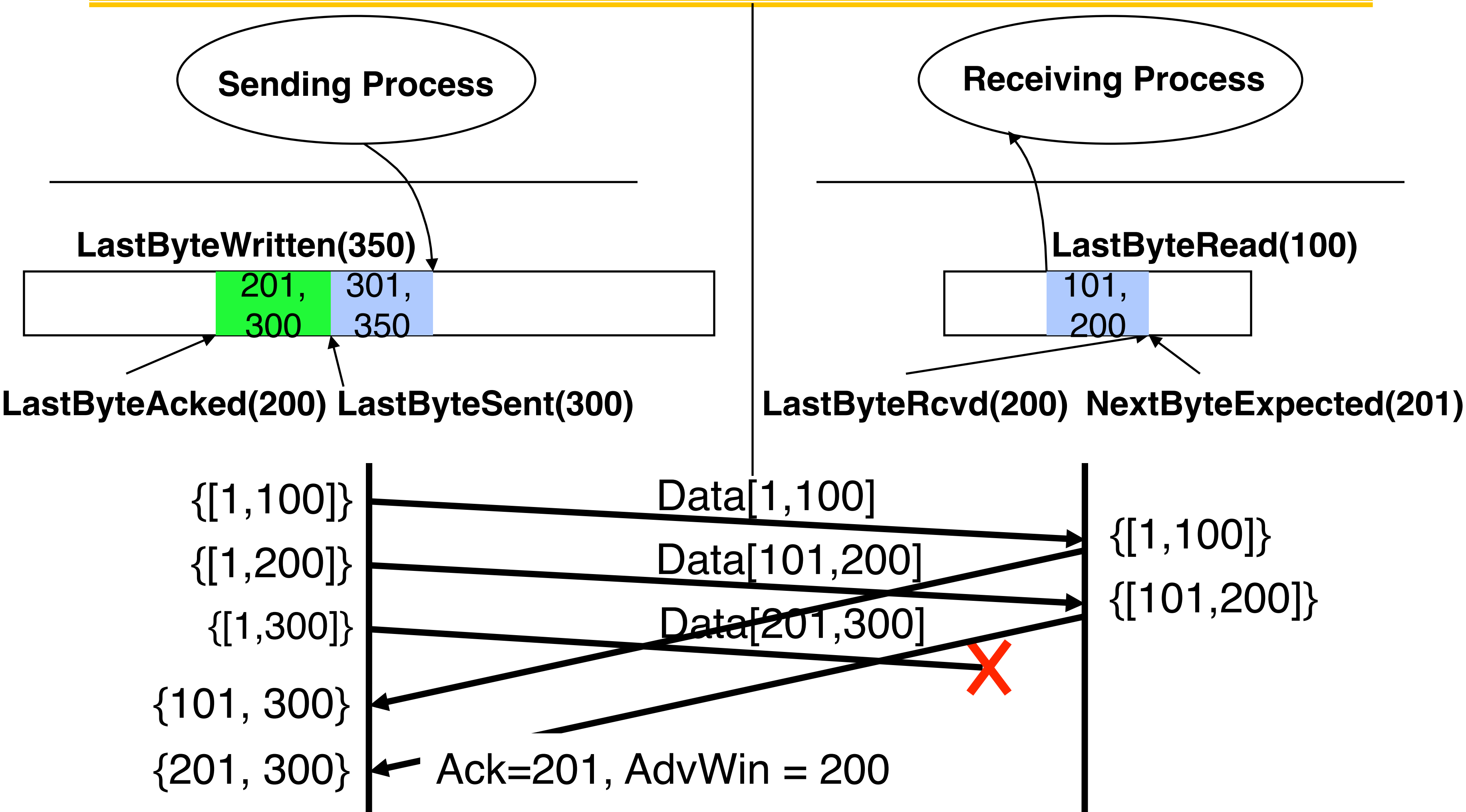
= 200 – (300 – 100) = 0

TCP Flow Control



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

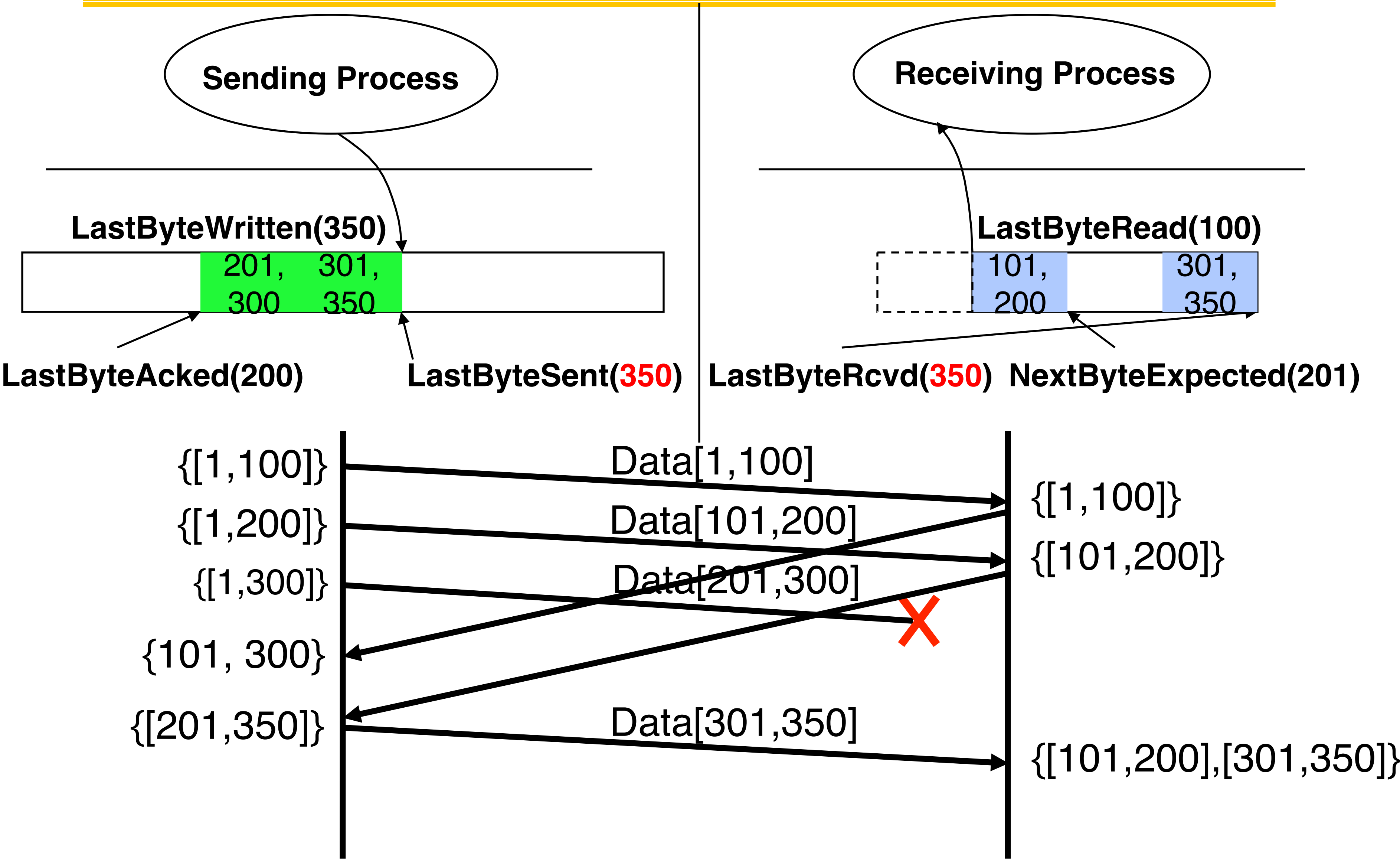
TCP Flow Control



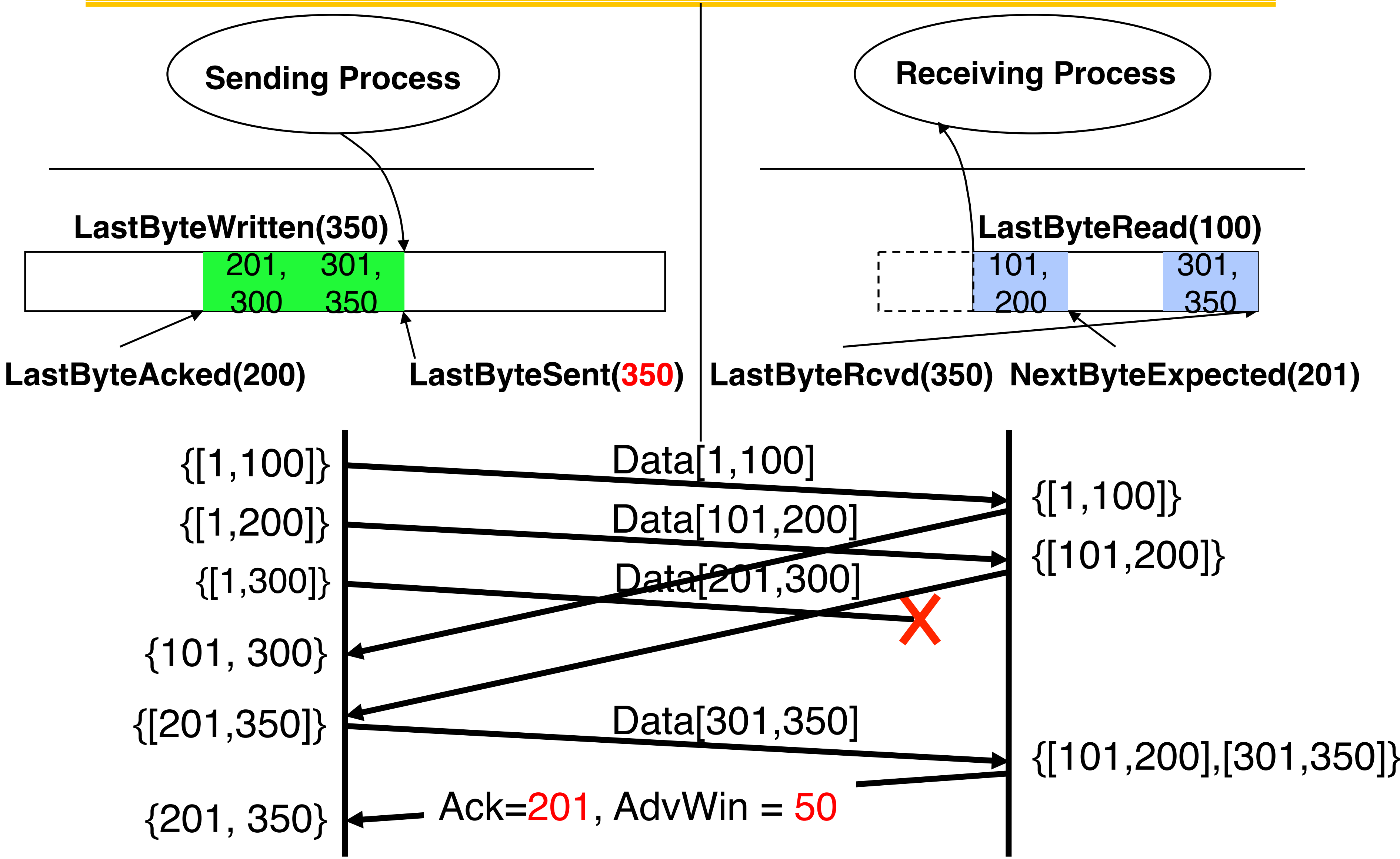
Sender can now send new data!

SndWin = AdvWin – (LastByteSent – LastByteAcked) = 100

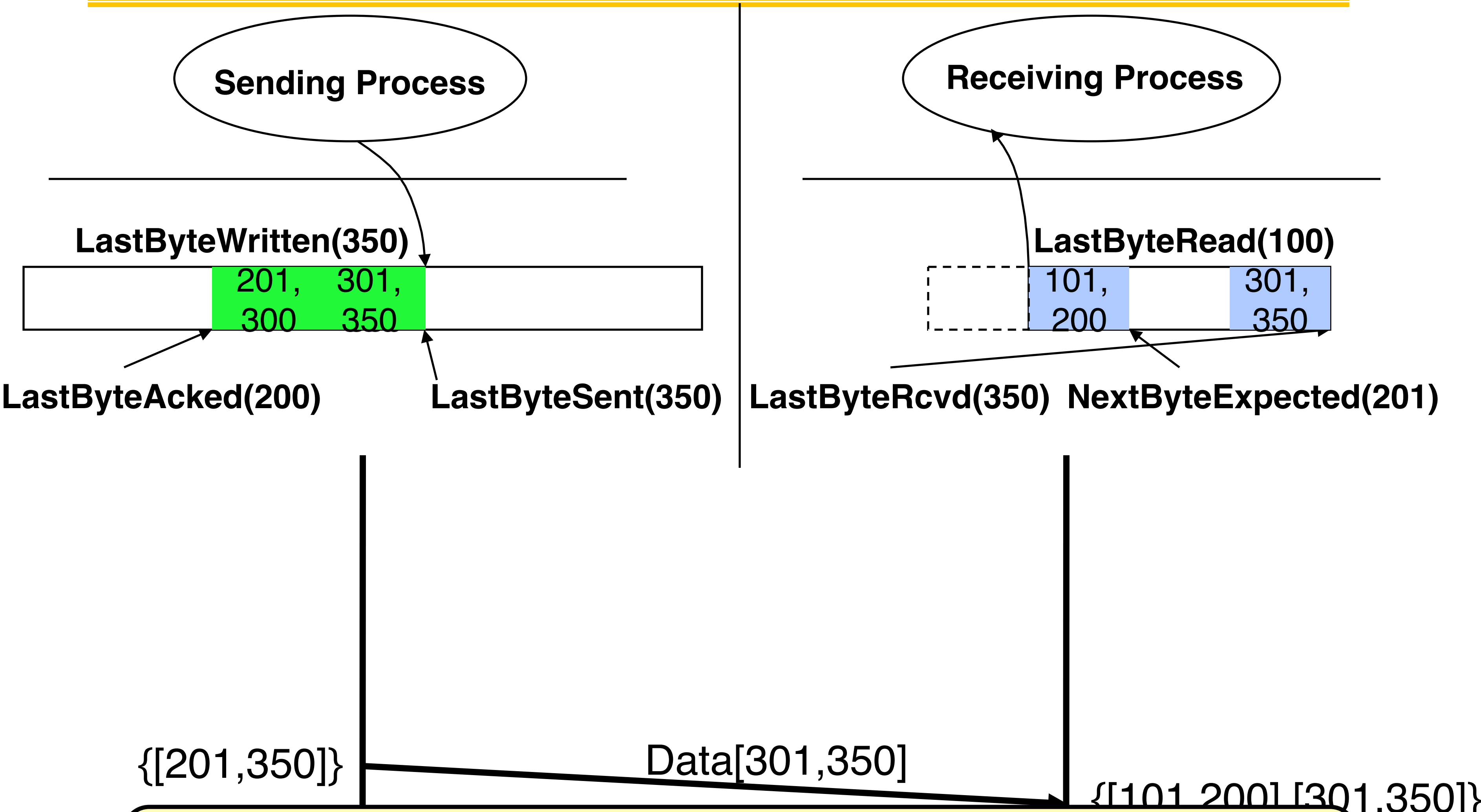
TCP Flow Control



TCP Flow Control

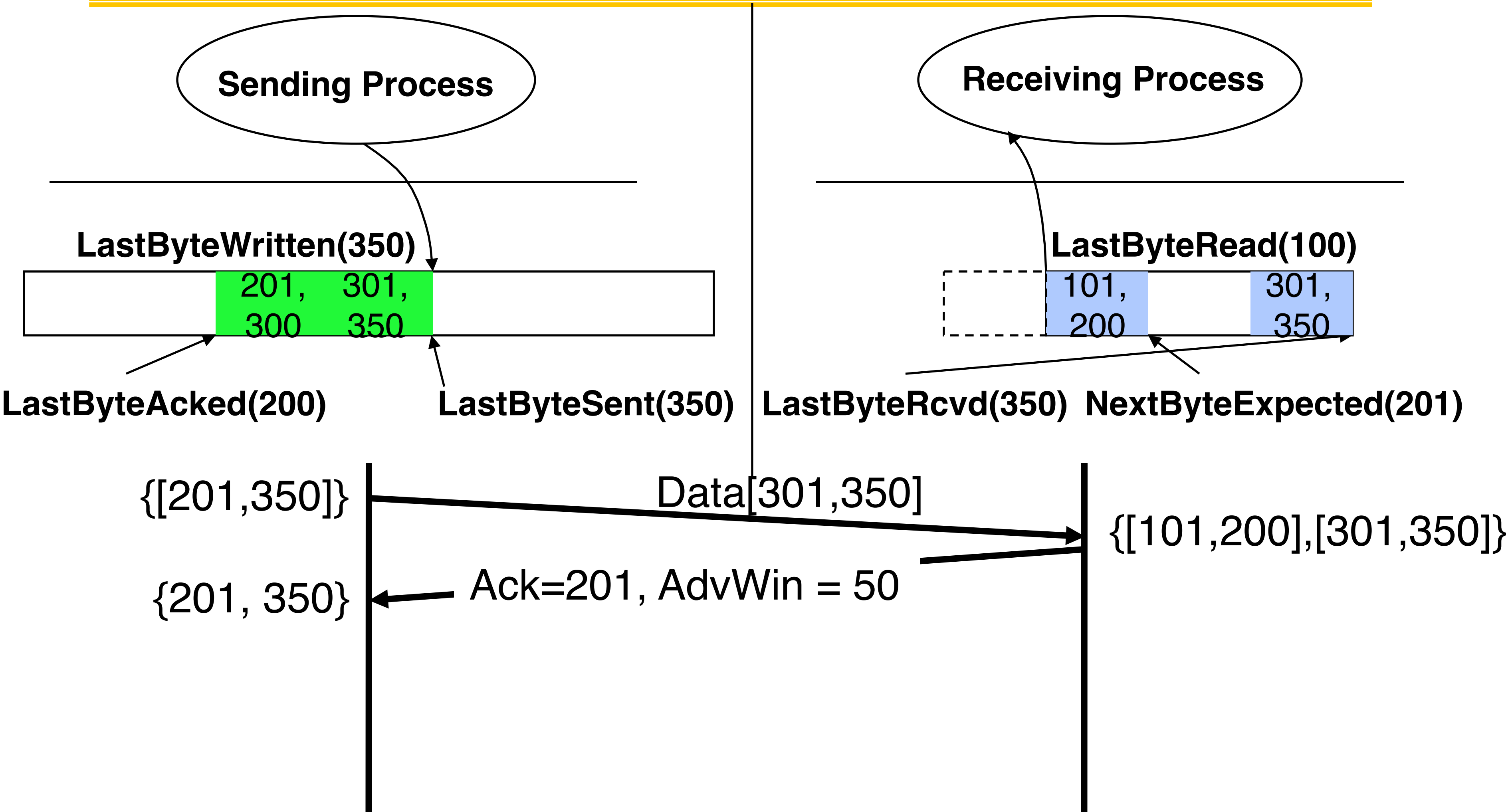


TCP Flow Control



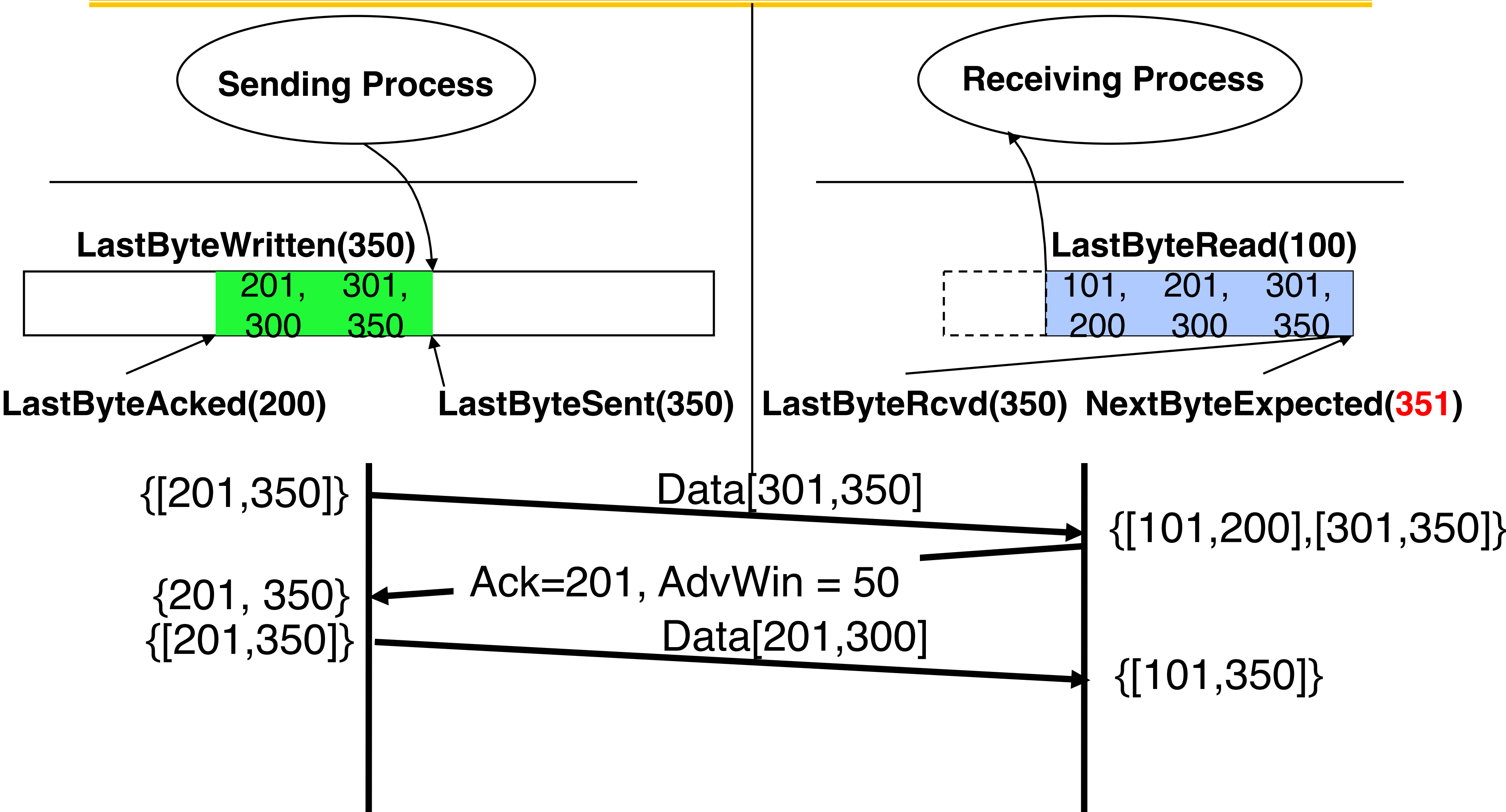
- Ack still specifies 201 (first byte out of sequence)
- AdvWin = 50, so can sender re-send 3rd packet?

TCP Flow Control



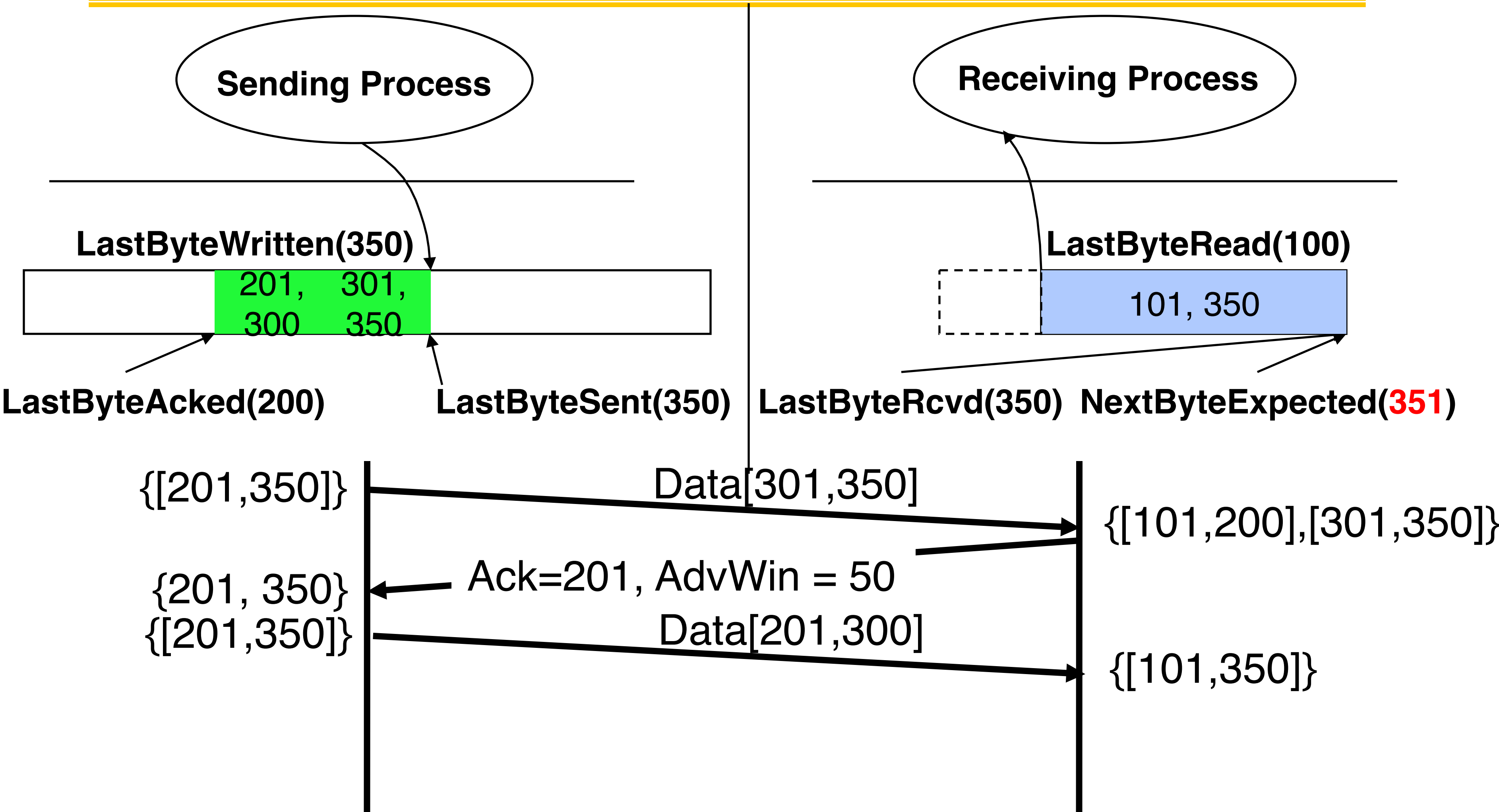
- Ack still specifies 201 (first byte out of sequence)
- AdvWin = 50, so can sender re-send 3rd packet?

TCP Flow Control



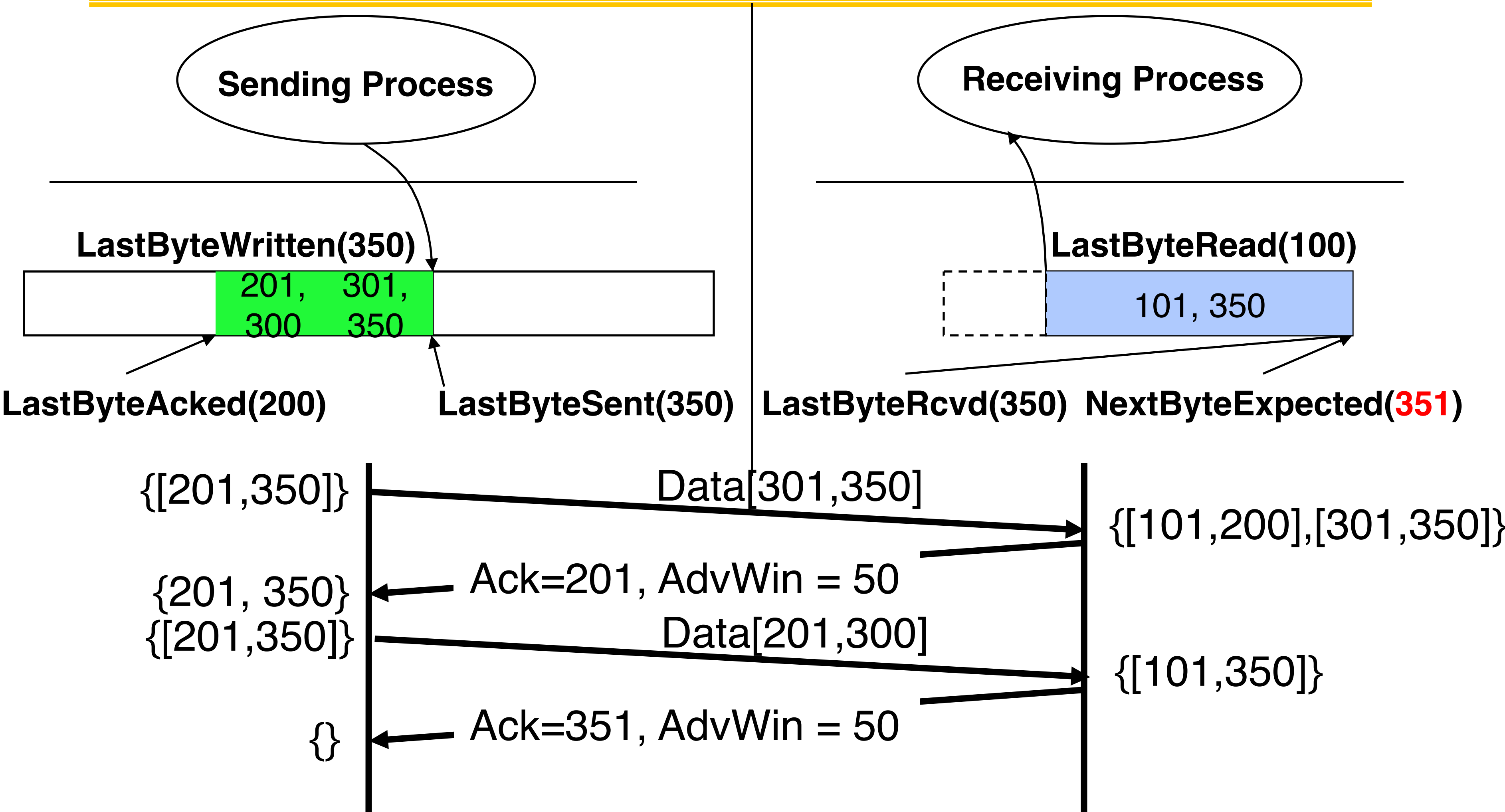
Yes! Sender can re-send 3rd packet since it's in existing window – won't cause receiver window to grow

TCP Flow Control



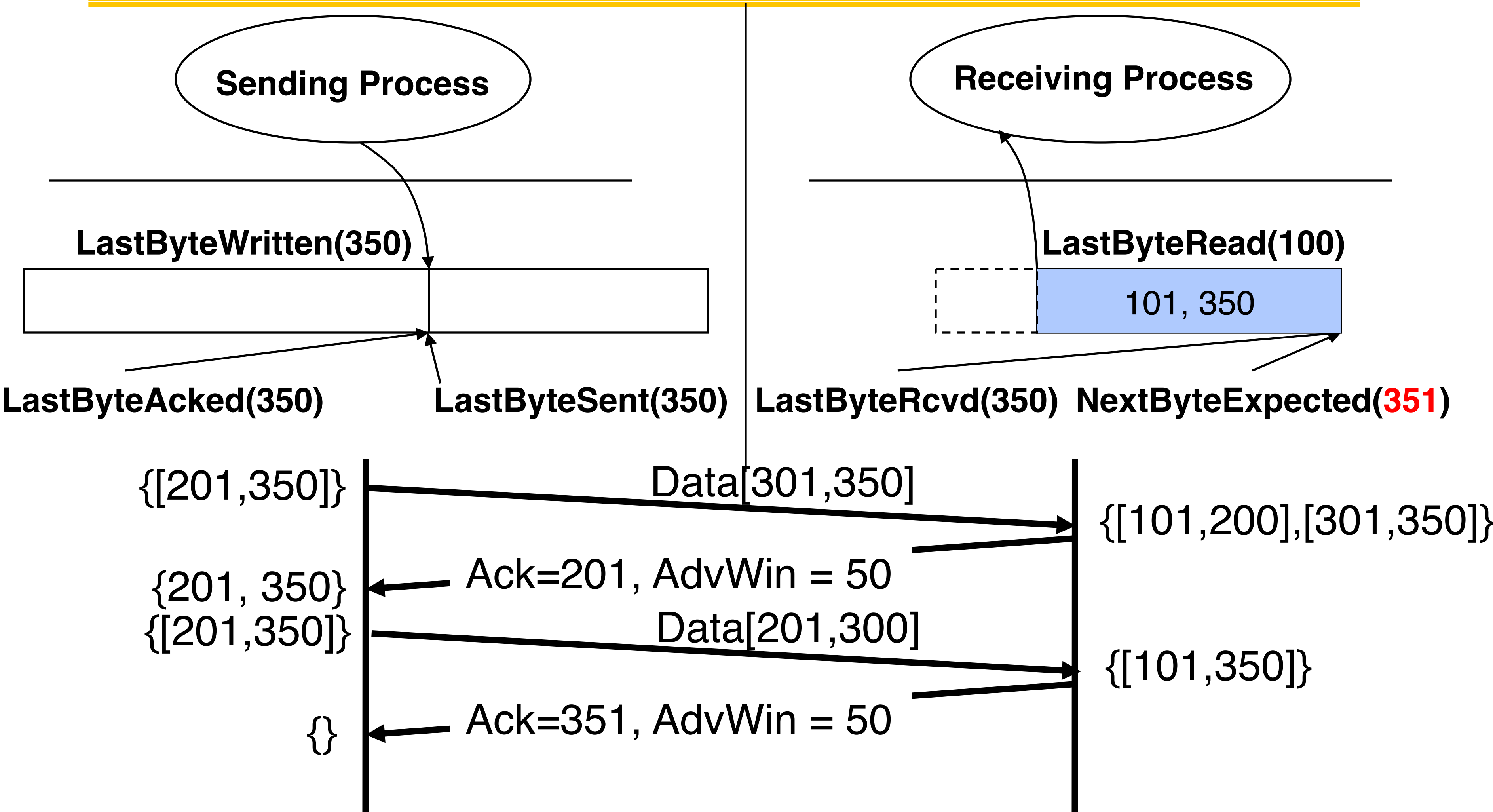
Yes! Sender can re-send 3rd packet since it's in existing window – won't cause receiver window to grow

TCP Flow Control



- Sender gets 3rd packet and sends Ack for 351
- AdvWin = 50

TCP Flow Control



Sender DONE with sending all bytes!

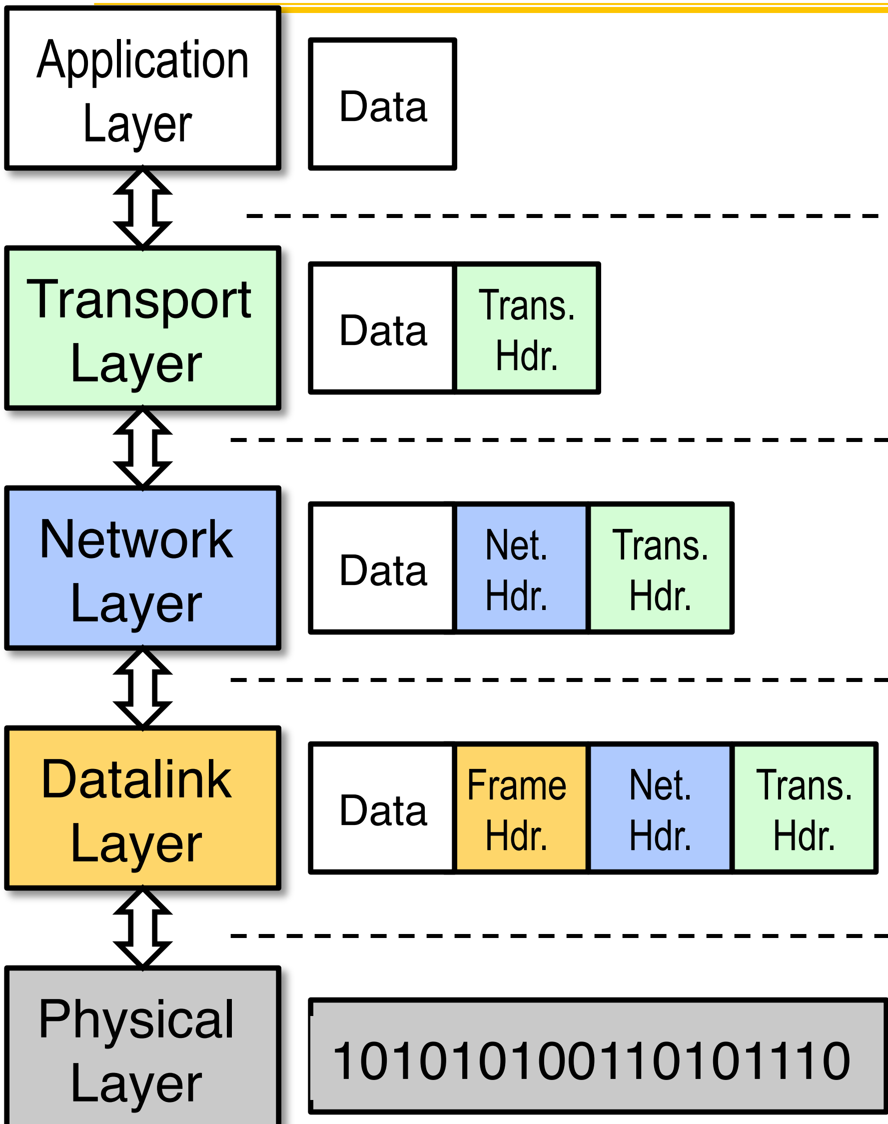
Discussion

- **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 CS168**
- **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

Internet Layering – engineering for intelligence and change



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