TCP Flow Control

Computer Networks(CS31204)

Prof. Sudip Misra

Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

Email: smisra@sit.iitkgp.ernet.in

Website: http://cse.iitkgp.ac.in/~smisra/

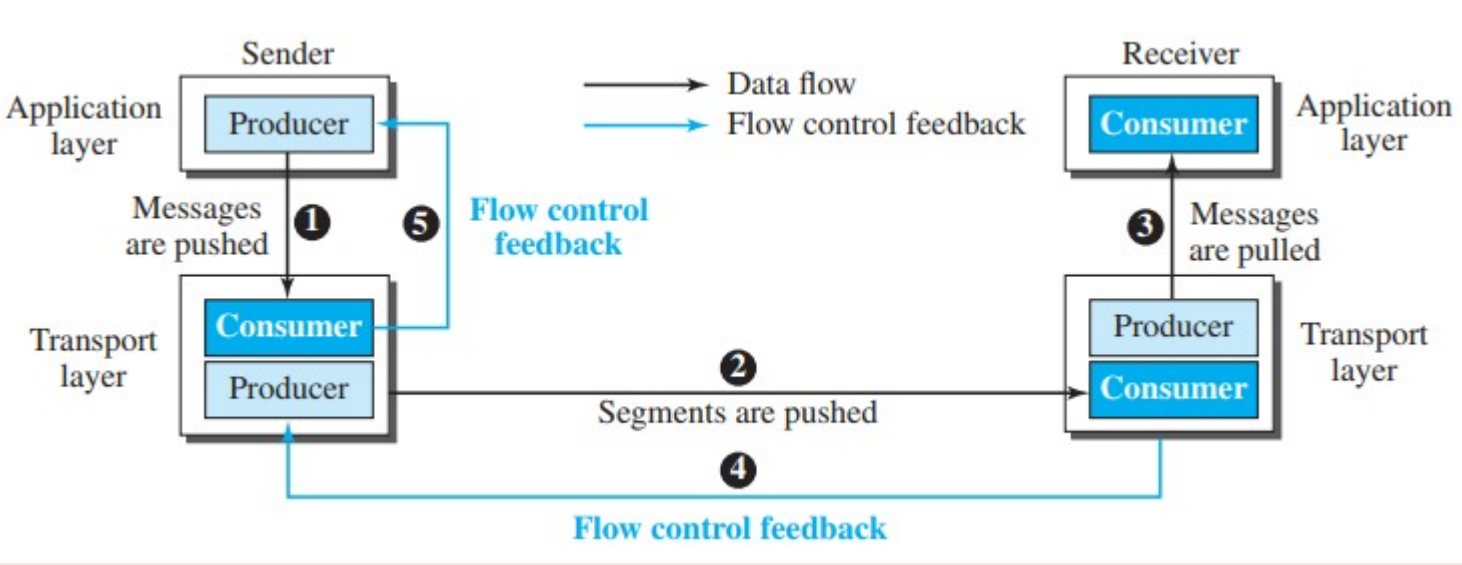
Research Lab: cse.iitkgp.ac.in/~smisra/swan/

Introduction

- TCP uses a sliding window protocol without selective or negative acknowledgments.
- Selective acknowledgments would let the protocol say it's missing a range of bytes. TCP can only say that it has received "up to byte N".
- The protocol has no way to specify a negative acknowledgment. It can only say what has been received
- Concepts same as discussed earlier, with some differences

Introduction

- As discussed before, *flow control* balances the rate a producer creates data with the rate a consumer can use the data.
- TCP separates flow control from error control.
- TCP to the receiving TCP, and from the receiving TCP up to the receiving process (paths 1, 2, and 3).
- Flow control feedbacks, however, are traveling from the receiving TCP to the sending TCP and from the sending TCP up to the sending process (paths 4 and 5).

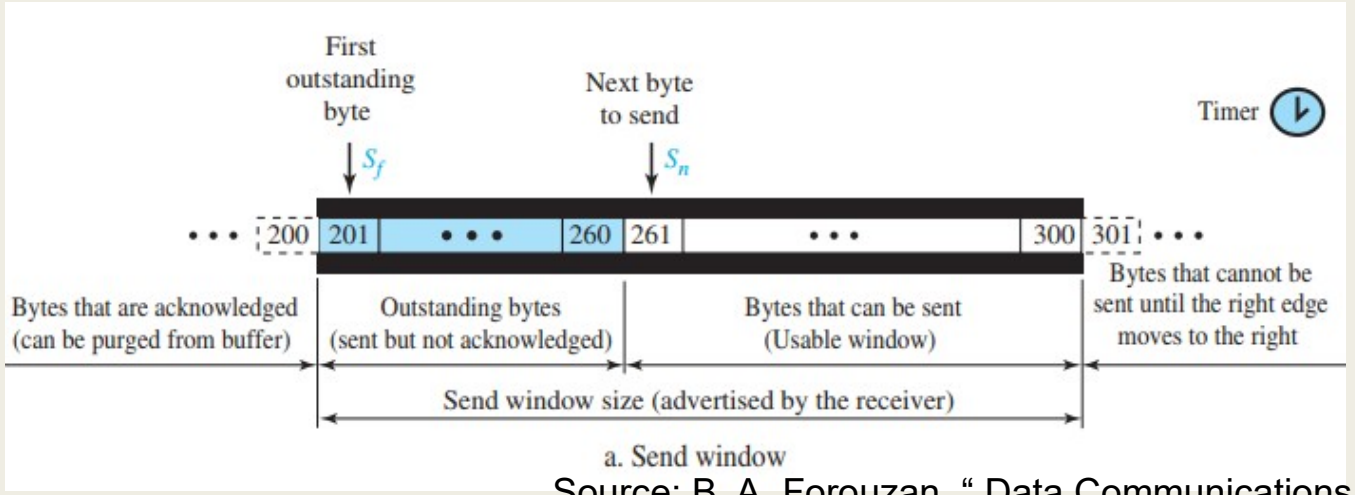


Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Sender Window

The send window in TCP is similar to the one used with the Selective-Repeat protocol, but with some differences:

- One difference is the nature of entities related to the window. The window size in SR is the number of packets, but the window size in TCP is the number of bytes. Although actual transmission in TCP occurs segment by segment, the variables that control the window are expressed in bytes.
- The second difference is that, in some implementations, TCP can store data received from the process and send them later, but we assume that the sending TCP is capable of sending segments of data as soon as it receives them from its process.
- Another difference is the number of timers. The theoretical Selective-Repeat protocol may use several timers for each packet sent, but as mentioned before, the TCP protocol uses only one timer.



a. Send window

Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Receiver Window

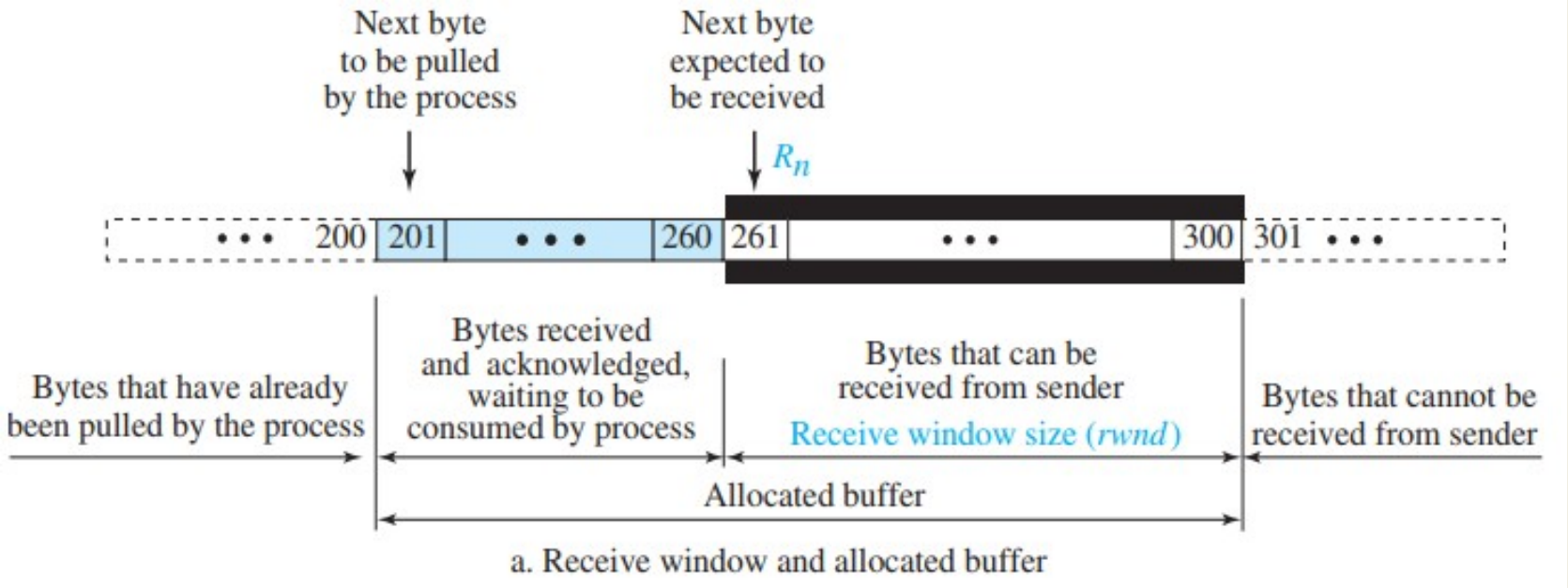
There are two differences between the receive window in TCP and the one we used for SR.

- The first difference is that TCP allows the receiving process to pull data at its own pace. This means that part of the allocated buffer at the receiver may be occupied by bytes that have been received and acknowledged, but are waiting to be pulled by the receiving process. The receive window size is then always smaller than or equal to the buffer size. The receive window size determines the number of bytes that the receive window can accept from the sender before being overwhelmed (flow control). In other words, the receive window size, normally called *rwnd,* can be determined as:

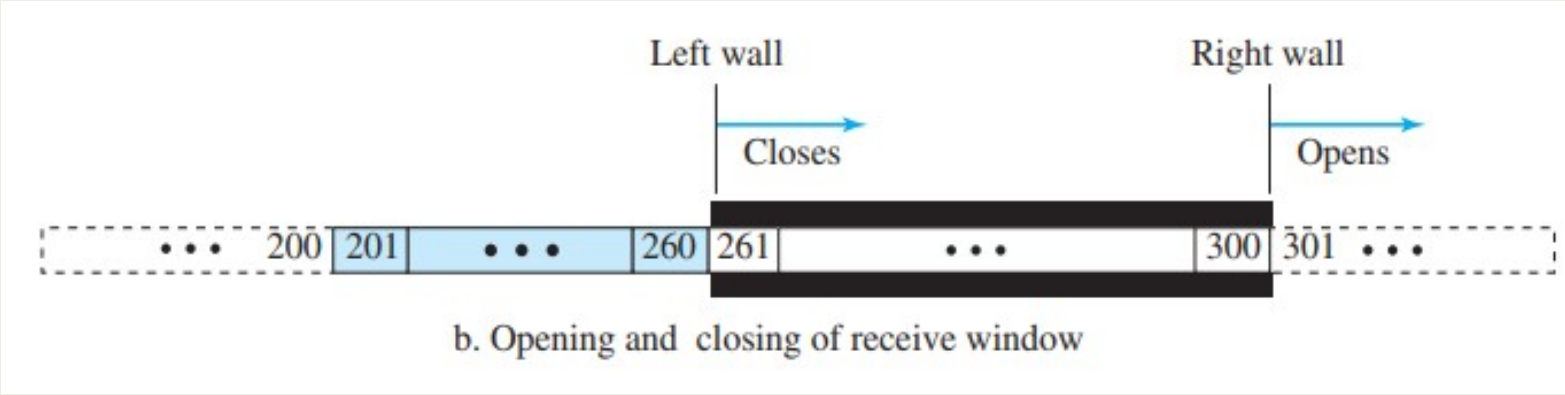
  ***rwnd* = buffer size - number of waiting bytes to be pulled**

- The second difference is the way acknowledgments are used in the TCP protocol. The acknowledgement in SR is selective, defining the uncorrupted packets that have been received. The major acknowledgment mechanism in TCP is
  a cumulative acknowledgment announcing the next expected byte to receive (in this way TCP looks like GBN. The new version of TCP, however, uses both cumulative and selective acknowledgments.

Receiver Window



a. Receive window and allocated buffer



b. Opening and closing of receive window

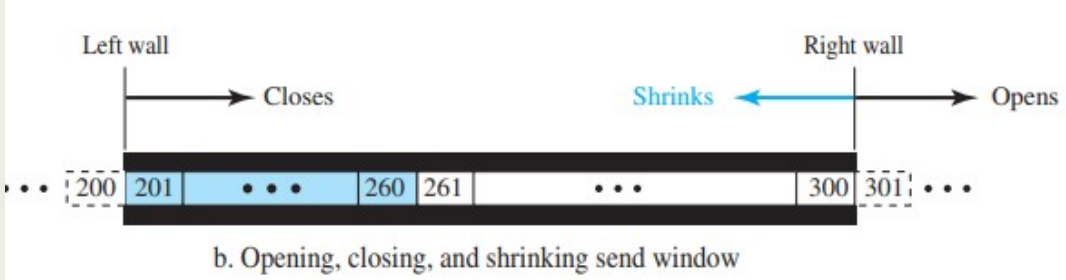
Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Opening and Closing Window

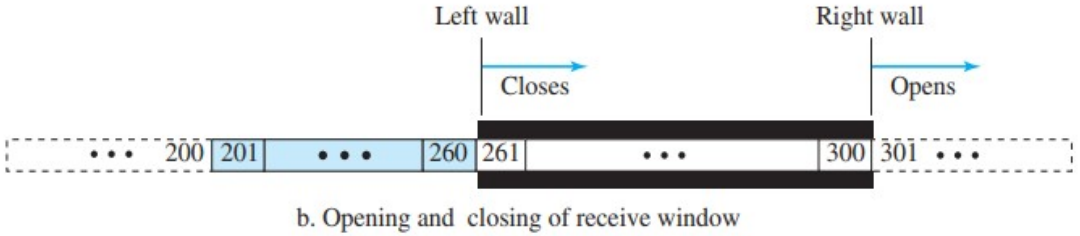
TCP forces the sender and the receiver to adjust their window sizes, although the size of the buffer for both parties is fixed when the connection is established.

The receiver window closes (moves its left wall to the right) when more bytes arrive from the sender; it opens (moves its right wall to the right) when more bytes are pulled by the process.

The opening, closing, and shrinking of the send window is controlled by the receiver. The send window closes (moves its left wall to the right) when a new acknowledgment allows it to do so. The send window opens (its right wall moves to the right) when the receive window size (rwnd) advertised by the receiver allows it to do so (new ackNo + new rwnd > last ackNo + last rwnd). The send window shrinks in the event this situation does not occur.



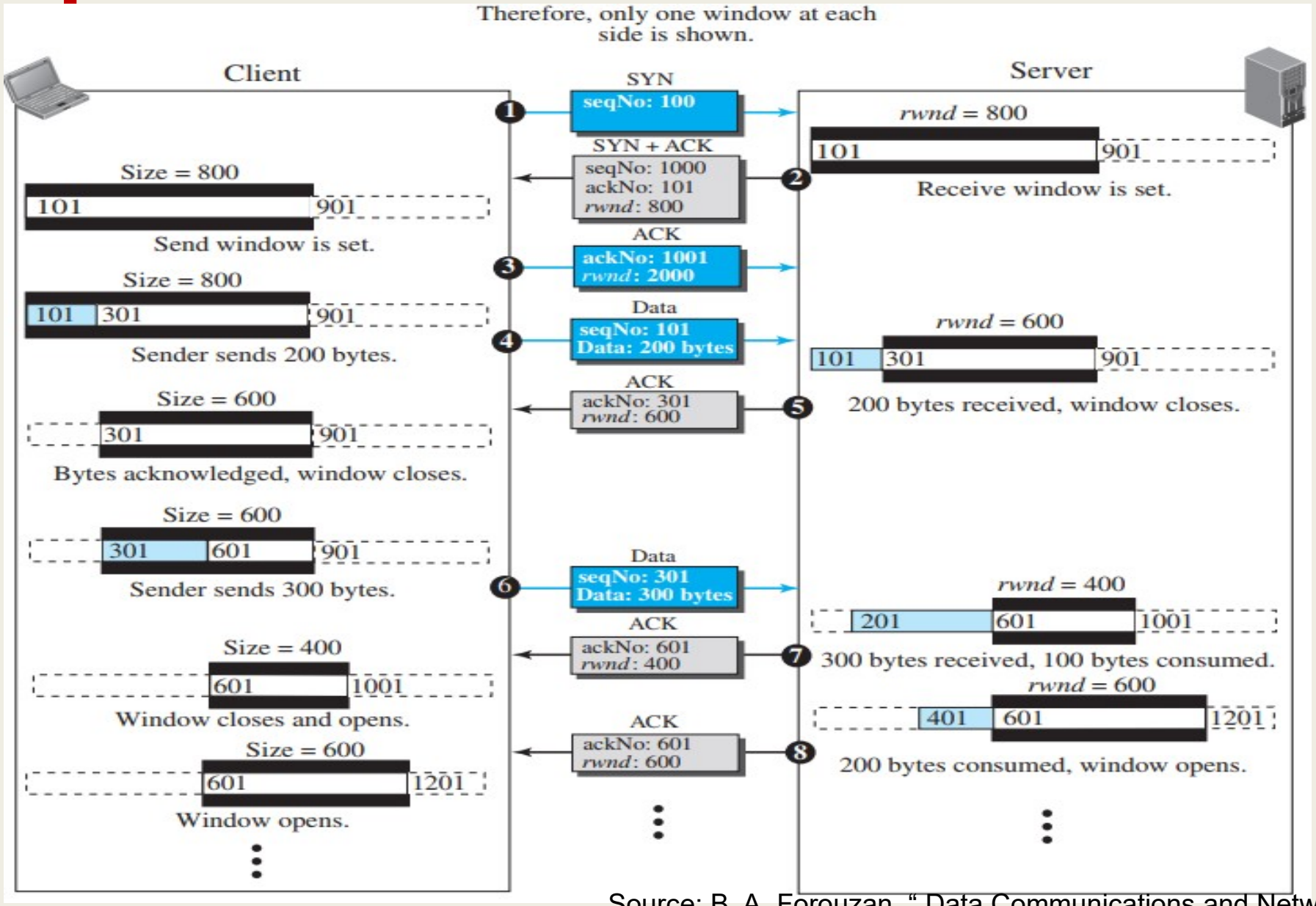
b. Opening, closing, and shrinking send window



b. Opening and closing of receive window

Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Example



Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Shrinking of Windows

- The receive window cannot shrink.
- The send window can shrink if the receiver defines a value for *rwnd* that results in shrinking the window.
- Some implementations do not allow shrinking of the send window.
- The limitation does not allow the right wall of the send window to move to the left. In other words, the receiver needs to keep the following relationship between the last and new acknowledgment and the last and new *rwnd* values to prevent shrinking of the send window.
- The left side of the inequality represents the new position of the right wall with respect to the sequence number space; the right side shows the old position of the right wall.
- The relationship shows that the right wall should not move to the left. The inequality is a mandate for the receiver to check its advertisement.
- The inequality is valid only if $S_f < S_n$; we need to remember that all calculations are in modulo 2^{32}.

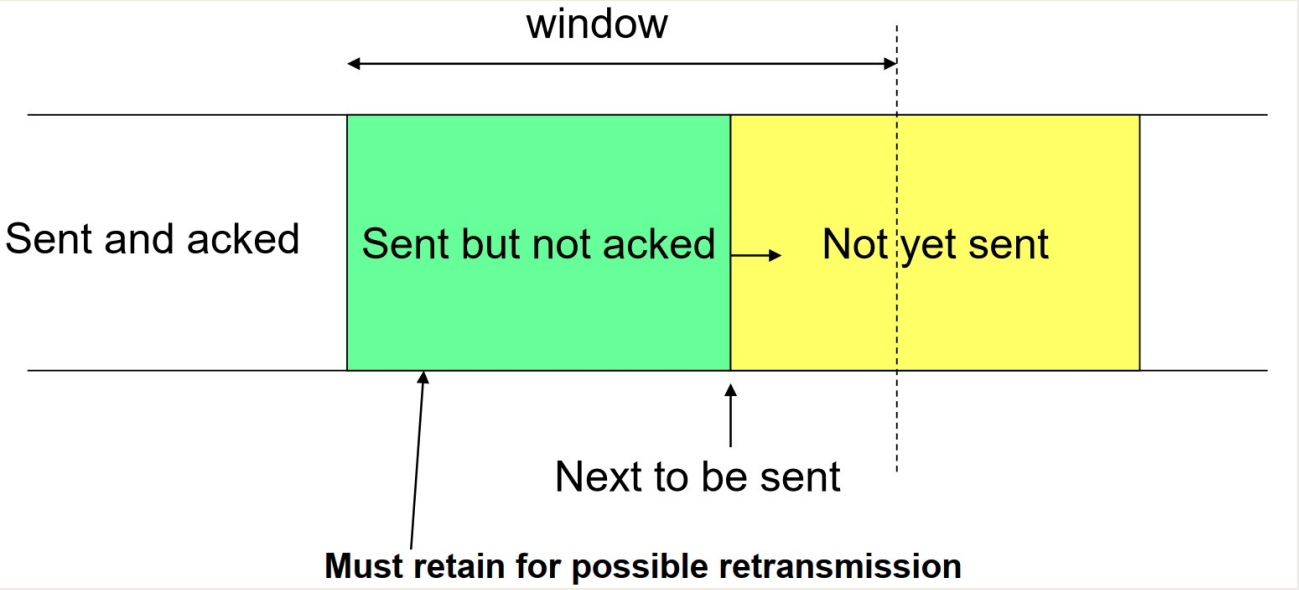
Problem

- Acknowledgment may be sent immediately by receiver, but receiver can delete acknowledged data from its buffer only after the data has been delivered to the application

- Application may read the data at different speeds and at different times

- So, depending on when and how fast the application reads the data, the receiver's window size may change

- So even if the receiver current window size is W bytes and the receiver receives and acknowledges p bytes, the window size may not be reset to W, may still be only W – p until the application picks up the data



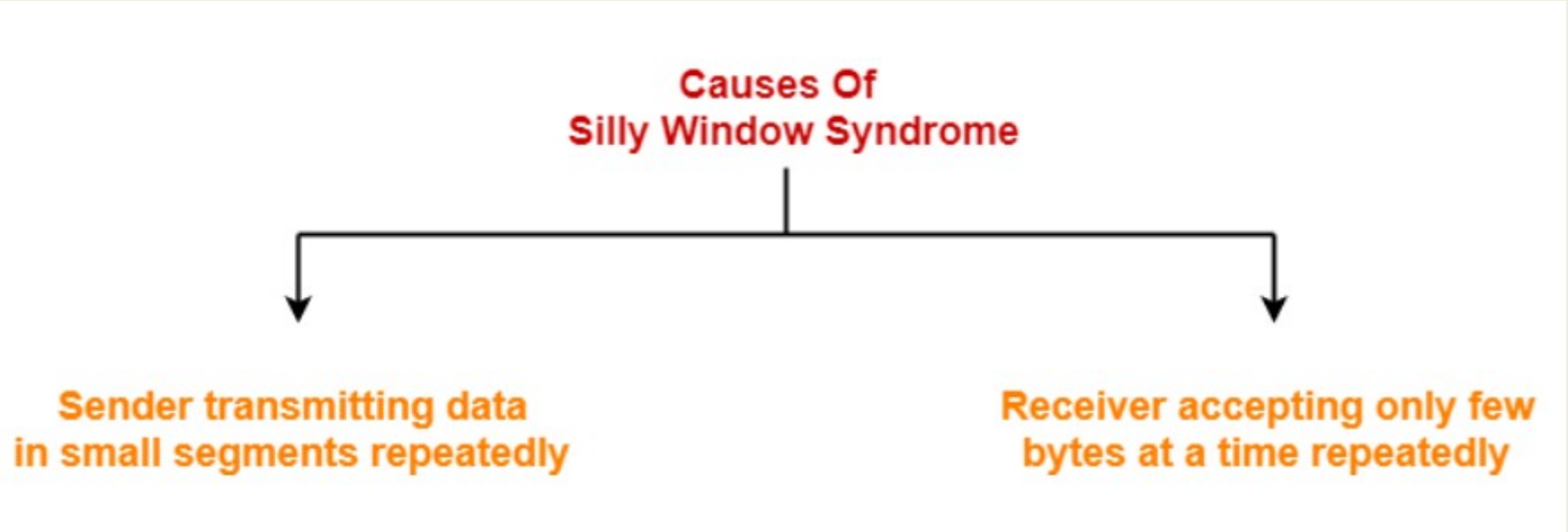
Solution

- Receiver tells sender what is the current window size in every segment it transmits to the sender (in Window field of header)
- This can be data sent from receiver to sender in the other direction, or ack's for the data received from the sender
- Sender uses this advertised window size instead of a fixed value
- Window size (also called advertised window) is continuously changing, = current free buffer space at receiver
- Can go to zero; sender not allowed to send anything!
- Naïve implementations can cause silly window syndrome

Silly Window Syndrome

The problem is called so because-

- It causes the sender window size to shrink to a silly value.
- The window size shrinks to such an extent where the data being transmitted is smaller than **TCP Header**.



Source:[Online], Available: https://www.gatevidyalay.com/silly-window-syndrome-nagles-algorithm/

Nagle's Algorithm

- Nagle's algorithm is an algorithm used in implementations of TCP/IP that controls traffic congestion on a network.
- Nagle's algorithm limits transmission of small datagrams and controls the size of the TCP sending window.
- The algorithm increases the efficiency of routers by reducing the latency of the routing process.

Nagle's algorithm suggests-

- Sender should send only the first byte on receiving one byte data from the application.
- Sender should buffer all the rest bytes until the outstanding byte gets acknowledged.
- In other words, sender should wait for 1 RTT.
- After receiving the acknowledgement, sender should send the buffered data in one TCP segment.
- Then, sender should buffer the data again until the previously sent data gets acknowledged.

When Acks are Sent?

Suppose A is sending data to B

- TCP acks are cumulative, acknowledges the longest contiguous sequence from the start that is received
- If ISN of A = 1000, A has sent 4 segments with sequence numbers 1001 (why not 1000?), 1600, 2800, and 3100, and B has received the 1st , 2nd and 4th segments only, then Longest contiguous sequence from start received is byte numbers1001 to 2799
- TCP will ack with ack no. = 2800 (next byte expected)
- Segment 1 and 2 are received in-order, Segment 3 is a missing segment, Segment 4 is a segment received out-of order
- Note that receiver does not know there is one missing segment, it just knows there is at least one

When Acks are Sent?

B sends an acknowledgement if/when

- If B has data to send to A, always piggyback the ack for the data received from A (ACK flag set, byte no. of next byte to expect put in Acknowledgement Number field
- If B has no data to send, receives a segment from A in-order (sequence number = next sequence number expected)
- If all previous in-order segments are acknowledged, delay sending the acknowledgement until one more segment arrives or a time elapses (typically 500 milliseconds), then send
- If B gets an out-of-order segment having a higher than expected sequence number, send an ack with next sequence number expected
- If the receiver gets a missing segment which extends the longest contiguous byte stream from the start that it has received, send an ack with the next sequence number to expect
- If a duplicate segment arrives, discard the segment but send an ack with the next sequence number expected

TCP Round Trip Estimator

- Original
  Round trip times estimated as a moving average:
- New RTT = α (old RTT) + (1 – α) (new sample)
- Set timeout to β ×RTT
- Typically, α = 0.8-0.9, β = 2 originally
- However, a fixed β does not adapt well to high variance in RTT
- Ideally, timeout > real RTT = estimated RTT + X
- If there is a high variance in the RTT, need a higher X to ensure timeout is always greater than RTT
- RTT variance is high at high loads

TCP Round Trip Estimator

Modified:

- Set timeout = Estimated RTT + δ × deviation
- δ = 4 typically
- How to compute the deviation in RTT?
- Deviation = (1–ρ)×deviation + ρ × (sample RTT – estimated RTT)
- Typically, ρ = 0.25

Acknowledgement Ambiguity

- If a segment is retransmitted, and an ack for it is received, is it an ack for the retransmitted frame or the original frame? What RTT sample value to take?

Solution: Karn's algorithm

- Do not accept samples for segments that are retransmitted
- Use a timer backoff scheme to increase timeout to account for scenarios when round trip delay increases suddenly

Karn's Algorithm

- Suppose that a segment is not acknowledged during the retransmission time-out period and is therefore retransmitted.
- When the sending TCP receives an acknowledgment for this segment, it does not know if the acknowledgment is for the original segment or for the retransmitted one.
- The value of the new RTT is based on the departure of the segment.
- However, if the original segment was lost and the acknowledgment is for the retransmitted one, the value of the current RTT must be calculated from the time the segment was retransmitted. This ambiguity was solved by Karn.
- **Karn's algorithm** do not consider the round-trip time of a retransmitted segment in the calculation of RTTs.
- Do not update the value of RTTs until you send a segment and receive an acknowledgment without the need for retransmission.

Karn's Algorithm

- Compute timeout using the estimated round trip time as before
- Use only samples for segments that are not retransmitted
- If timer expires and retransmission occurs
- For every retransmission, set timeout = γ × timeout, subject to an upper limit
- Typically, γ = 2

Ensuring In-Order Delivery

- TCP uses sequence number field in segment headers to reconstruct the data stream at the receiver side
- What happens to segments received out-of-order?
- TCP specification does not restrict, up to implementations

Out of Band Data

- Sometimes there is need to send urgent data that needs to be delivered to the application out of turn
- Set URG flag bit to indicate presence of out-of-band data in segment
- Set URGENT pointer to position in segment where urgent data ends

Resetting the TCP Connection

- Can be used to abort a TCP connection in case of any abnormal condition
- Initiated by one side, sends a TCP segment with the RST flag set in header
- The other side responds with a TCP segment with the RST flag set in header
- Immediately releases all resources and closes the connection in both directions

Cumulative Acknowledgment

- TCP was originally designed to acknowledge receipt of segments cumulatively.
- The receiver advertises the next byte it expects to receive, ignoring all segments received and stored out of order. This is sometimes referred to as *positive cumulative acknowledgment,* or ACK.
- The word *positive* indicates that no feedback is provided for discarded, lost, or duplicate segments.
- The 32-bit ACK field in the TCP header is used for cumulative acknowledgments, and its value is valid only when the ACK flag bit is set to 1.

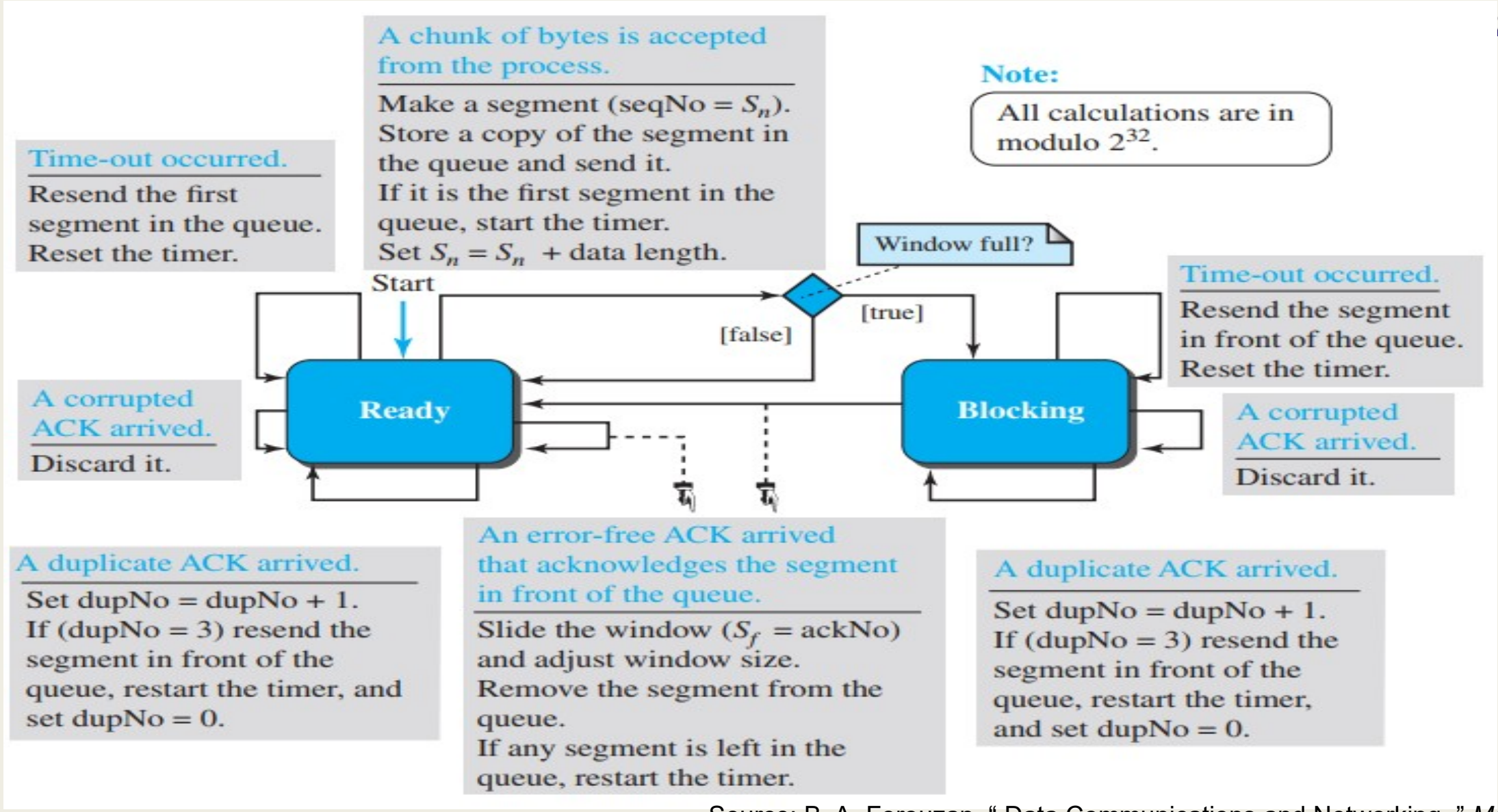
Selective Acknowledgment

- More and more implementations are adding another type of acknowledgment called selective acknowledgment, or SACK.
- A SACK does not replace an ACK, but reports additional information to the sender.
- A SACK reports a block of bytes that is out of order, and also a block of bytes that is duplicated, i.e., received more than once.
- However, since there is no provision in the TCP header for adding this type of information, SACK is implemented as an option at the end of the TCP header.

Contd.

- When a segment arrives with an out-of-order sequence number that is higher than expected, the receiver immediately sends an ACK segment announcing the sequence number of the next expected segment. This leads to the *fast retransmission* of missing segments.
- When a missing segment arrives, the receiver sends an ACK segment to announce the next sequence number expected. This informs the receiver that segments reported missing have been received.
- If a duplicate segment arrives, the receiver discards the segment, but immediately sends an acknowledgment indicating the next in-order segment expected. This solves some problems when an ACK segment itself is lost.

Sender Side FSM



Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Receiver Side FSM

Note:

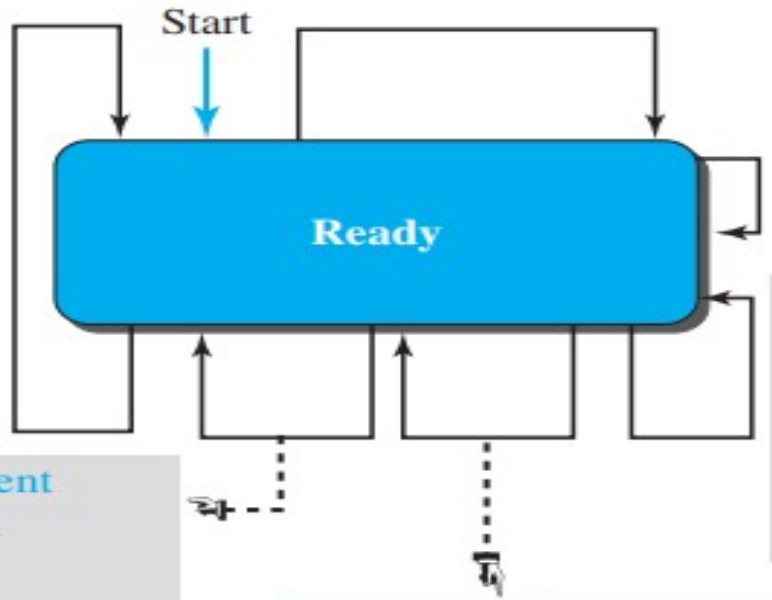
All calculations are in modulo 2^{32}.

An expected error-free segment arrived.

Buffer the message.
$R_n = R_n$ + data length.
If the ACK-delaying timer is running, stop the timer and send a cumulative ACK. Otherwise, start the ACK-delaying timer.

A request for delivery of *k* bytes of data from process came.

Deliver the data.
Slide the window and adjust window size.



ACK-delaying timer expired.

Send the delayed ACK.

An error-free, but out-of-order segment arrived.

Store the segment if not duplicate.
Send an ACK with ackNo equal to the sequence number of expected segment (duplicate ACK).

An error-free duplicate segment or an error-free segment with sequence number outside window arrived.

Discard the segment.
Send an ACK with ackNo equal to the sequence number of expected segment (duplicate ACK).

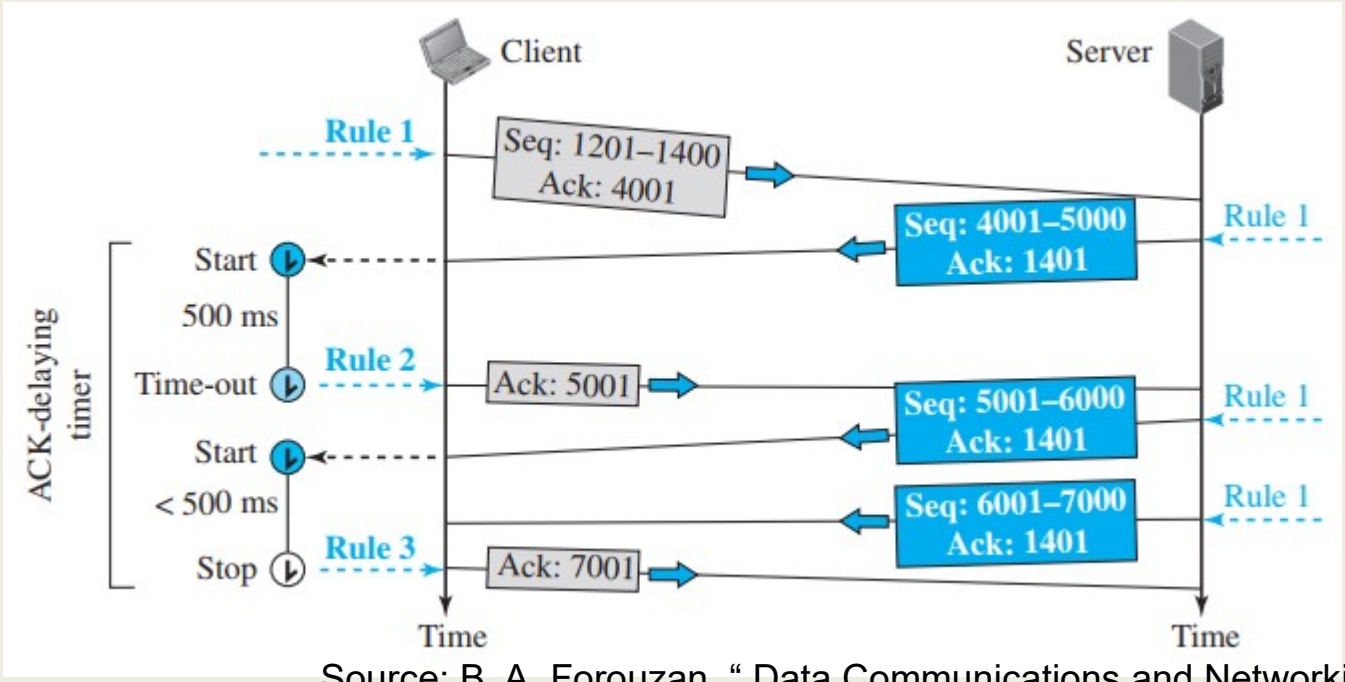
A corrupted segment arrived.

Discard the segment.

Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

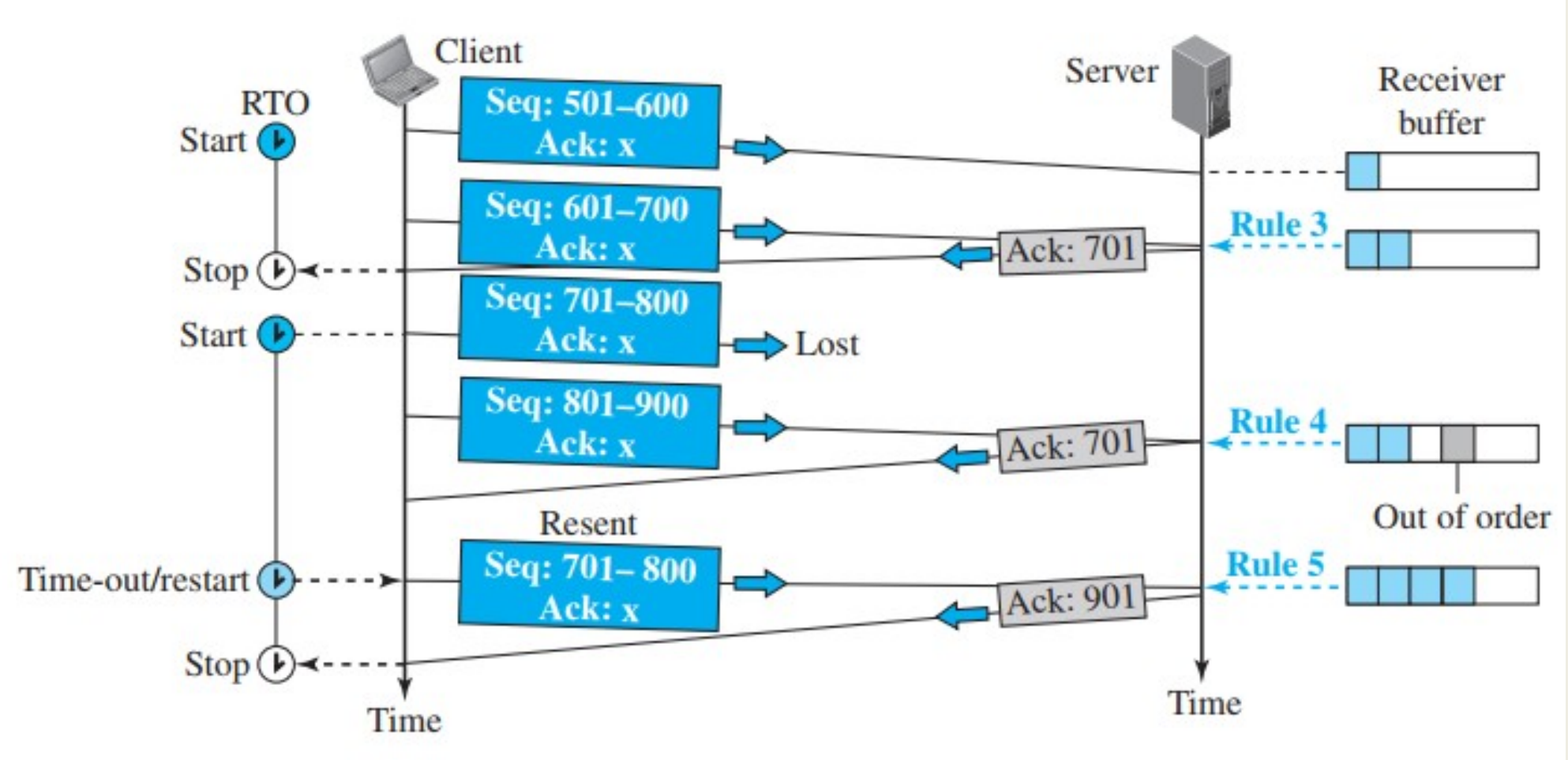
Normal Operation of TCP

- When the client receives the first segment from the server, it does not have any more data to send; it needs to send only an ACK segment.
- The acknowledgment needs to be delayed for 500 ms to see if any more segments arrive. When the ACK-delaying timer matures, it triggers an acknowledgment. This is because the client has no knowledge of whether other segments are coming; it cannot delay the acknowledgment forever.
- When the next segment arrives, another ACK delaying timer is set.
- However, before it matures, the third segment arrives. The arrival of the third segment triggers another acknowledgment.



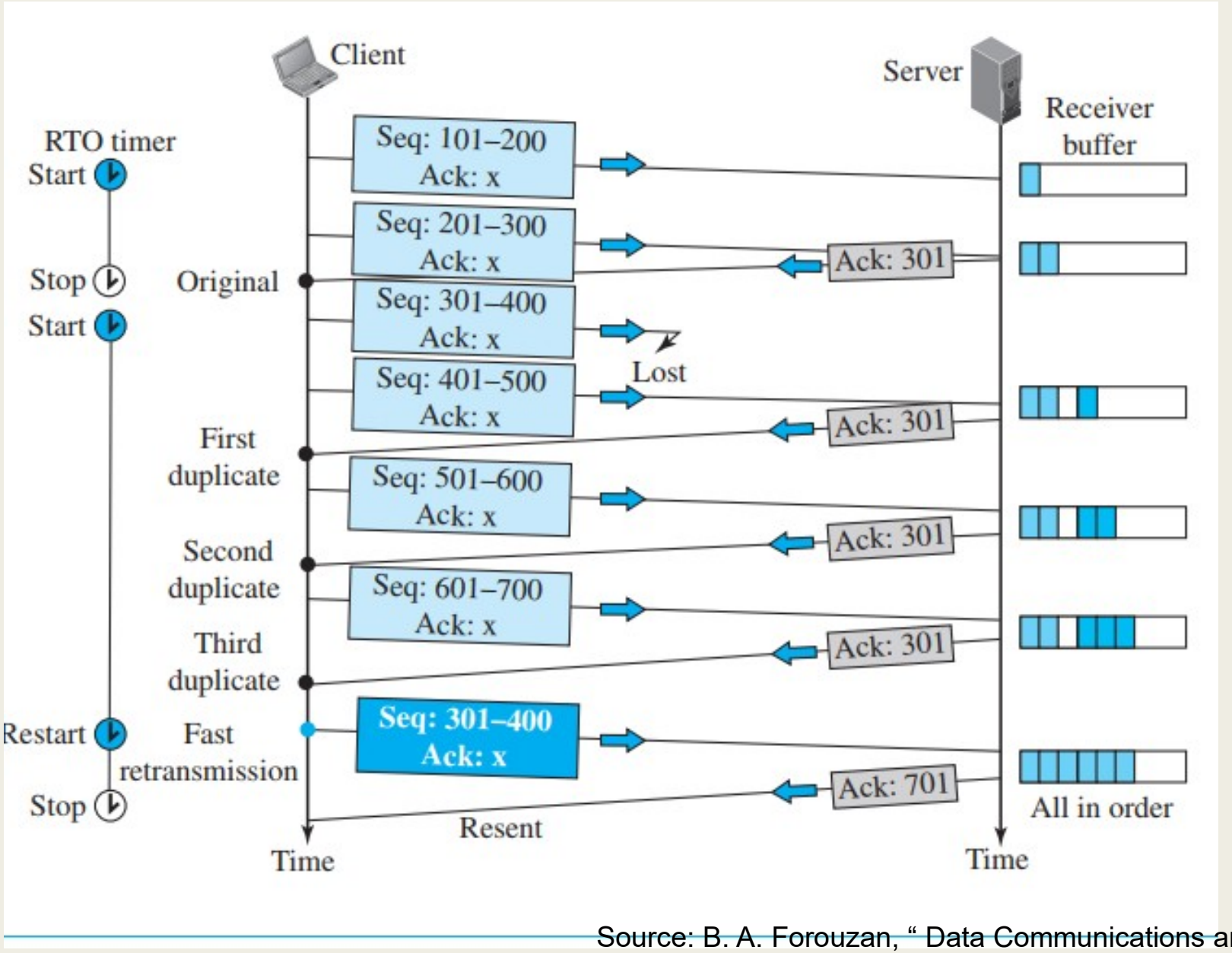
Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Lost Segment



Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Fast Retransmission



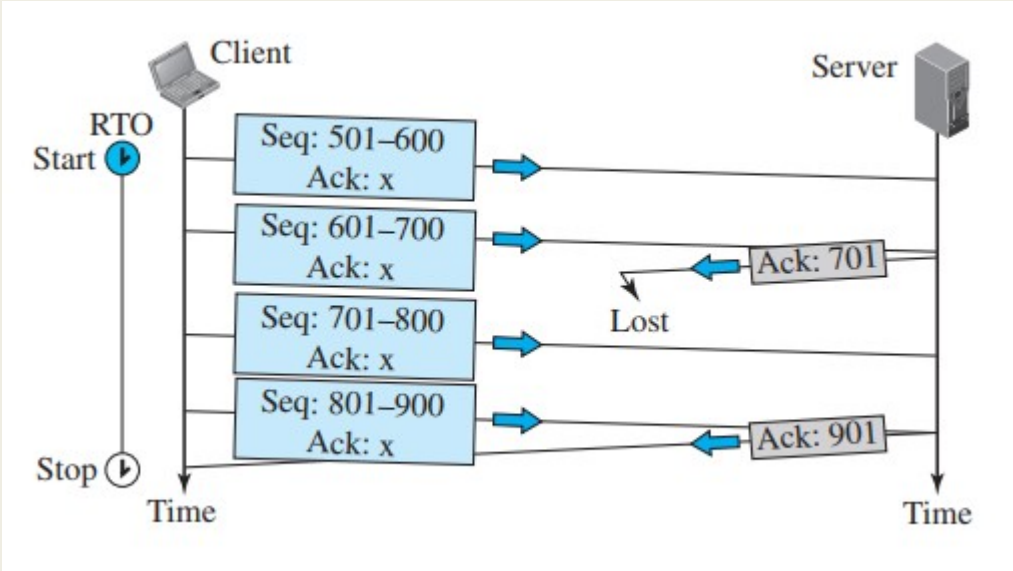
Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Delayed Segment

- TCP uses the services of IP, which is a connectionless protocol.
- Each IP datagram encapsulating a TCP segment may reach the final destination through a different route with a different delay. Hence TCP segments may be delayed.
- Delayed segments sometimes may time out and be resent.
- If the delayed segment arrives after it has been resent, it is considered a duplicate segment and discarded.

Automatically Corrected Lost Segment

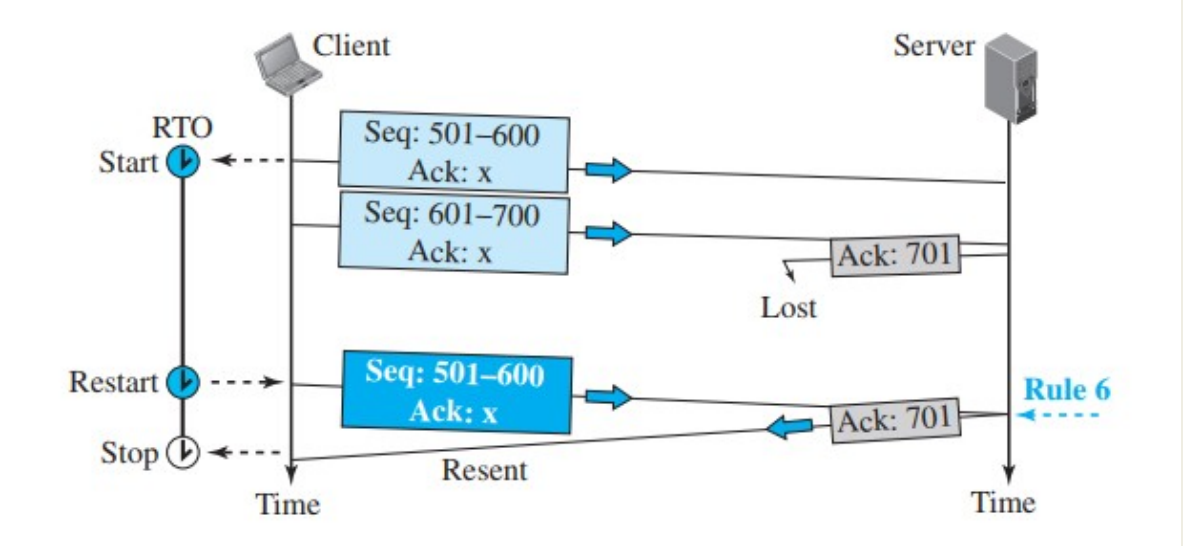
- In the TCP acknowledgment mechanism, a lost acknowledgment may not even be noticed by the source TCP.
- TCP uses cumulative acknowledgment.
- We can say that the next acknowledgment automatically corrects the loss of the previous acknowledgment.



Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Lost Acknowledgment Corrected by Resending a Segment

- If the next acknowledgment is delayed for a long time or there is no next acknowledgment (the lost acknowledgment is the last one sent), the correction is triggered by the RTO timer.
- A duplicate segment is the result. When the receiver receives a duplicate segment, it discards it and resends the last ACK immediately to inform the sender that the segment or segments have been received.



Source: B. A. Forouzan, " Data Communications and Networking ," *McGraw-Hill Forouzan Networking Series*,5E.

Thank You!!!