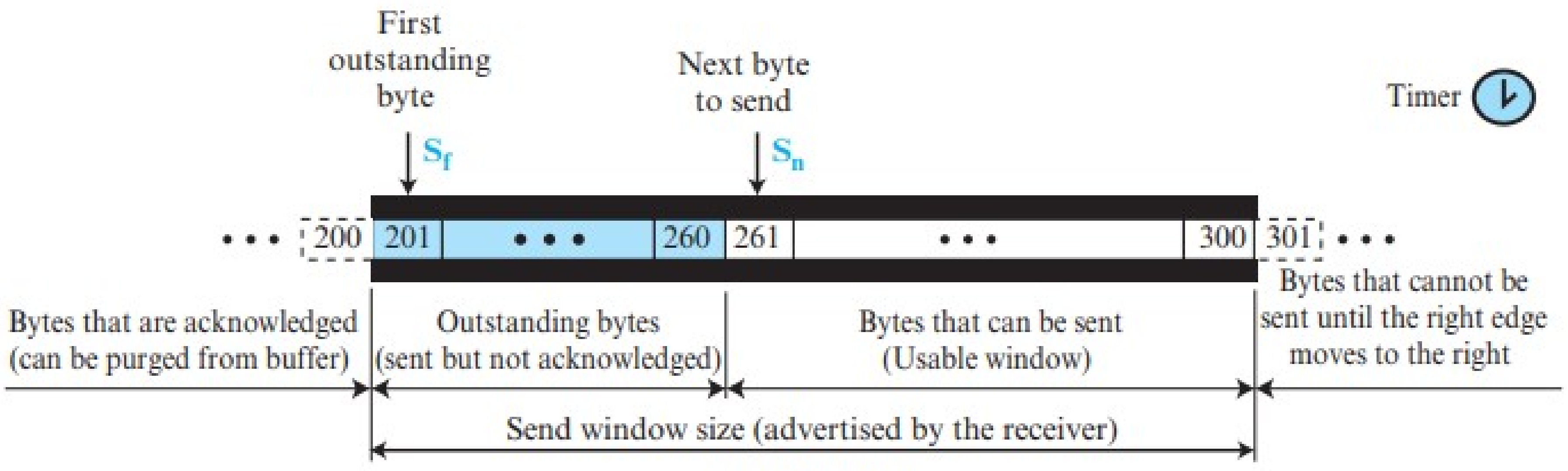
Windows in TCP

- TCP uses two windows (send window and receive window) for each direction of data transfer, which means four windows for a bidirectional communication.
- To make the discussion simple, we make an unrealistic assumption that communication is only unidirectional (say from client to server); the bidirectional communication can be inferred using two unidirectional communications with piggybacking.

Windows in TCP

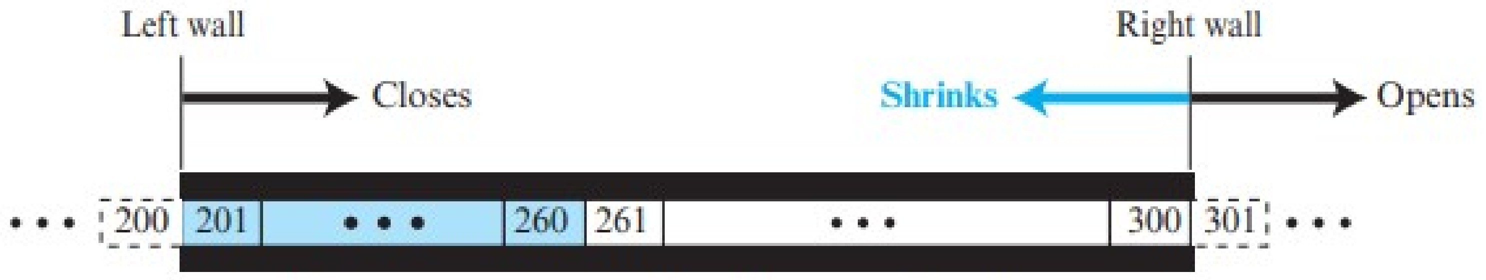
- **Send Window**
  - The window size is 100 bytes but can be changed depending receiver's flow control capacity and network's congestion control capacity.
  - The send window in TCP is same as send window in selective-repeat protocol with few differences -
    - The window in SR stores packets but in case TCP window stores bytes.
    - As we have discussed that theoretical Selective-Repeat protocol may use a separate timer for each packet sent but the TCP only uses one timer.

Send Window in TCP



First outstanding byte
Next byte to send
Timer
S_f
S_n
200
201
260
261
300
301
Bytes that are acknowledged (can be purged from buffer)
Outstanding bytes (sent but not acknowledged)
Bytes that can be sent (Usable window)
Bytes that cannot be sent until the right edge moves to the right
Send window size (advertised by the receiver)

a. Send window

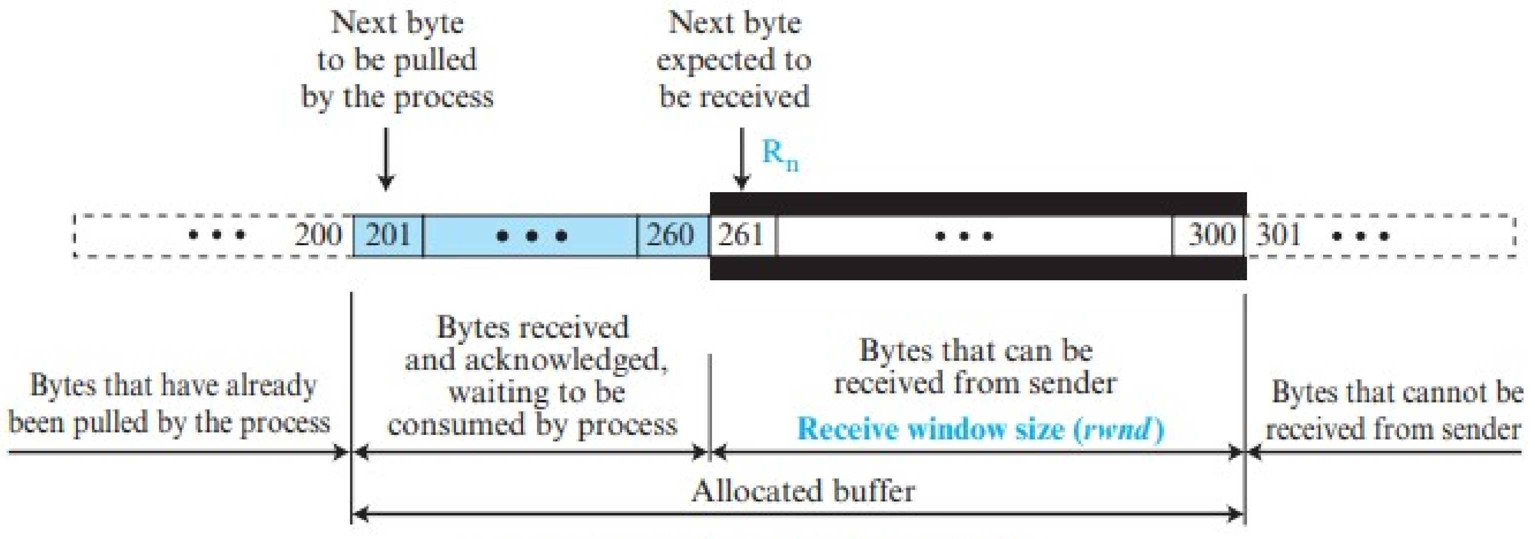


b. Opening, closing, and shrinking send window

Windows in TCP

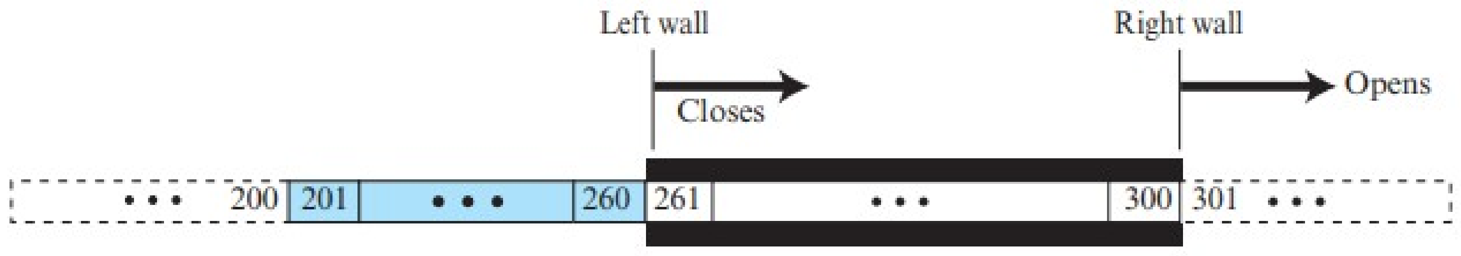
- **Receive Window**
  - In TCP, Receive window size is 100 bytes same as the send window. Receive window also stores bytes instead of packets.
  - Few differences from receive window in selective-repeat -
    - The first difference is that TCP allows the receiving process to pull data at its own pace. 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 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).
    - The second difference is the way acknowledgments are used in the TCP protocol. Remember that an 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).

Receive Window in TCP



Next byte to be pulled by the process
Next byte expected to be received
R_n
• • • 200
201
• • •
260
261
• • •
300
301 • • •
Bytes that have already been pulled by the process
Bytes received and acknowledged, waiting to be consumed by process
Bytes that can be received from sender
Receive window size (rwnd)
Bytes that cannot be received from sender
Allocated buffer

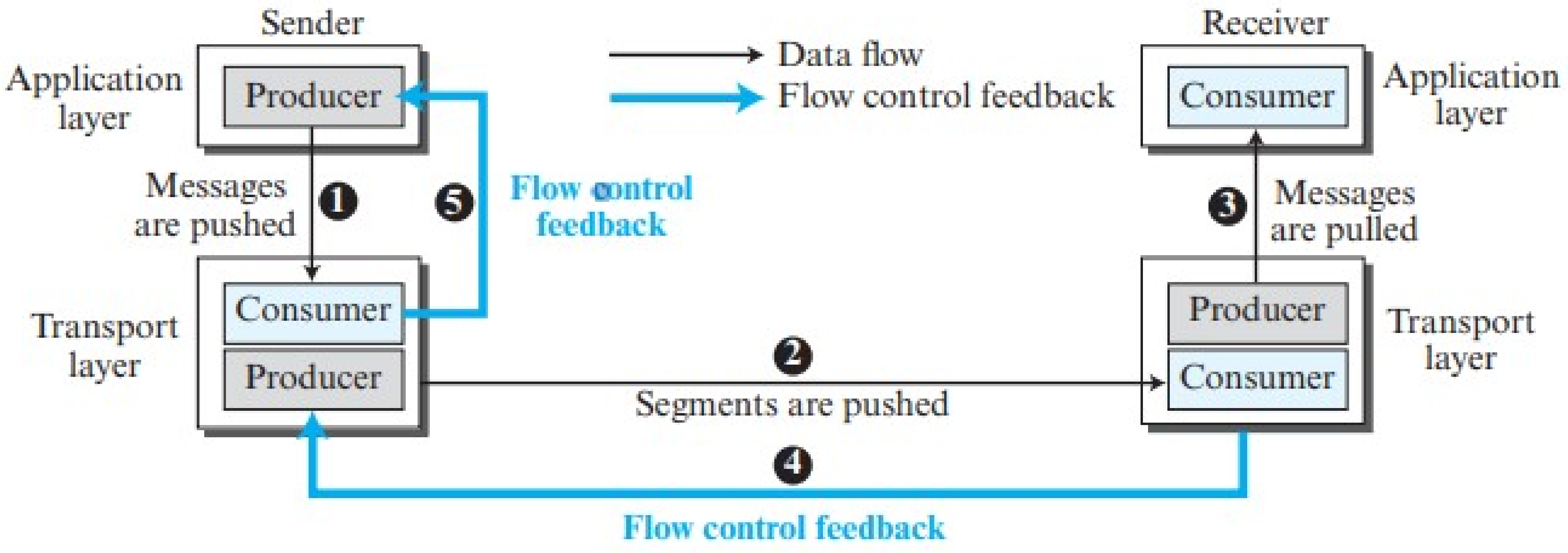
a. Receive window and allocated buffer



b. Opening and closing of receive window

rwnd = buffer size - number of bytes waiting to be pulled by receiver process

Flow Control in TCP

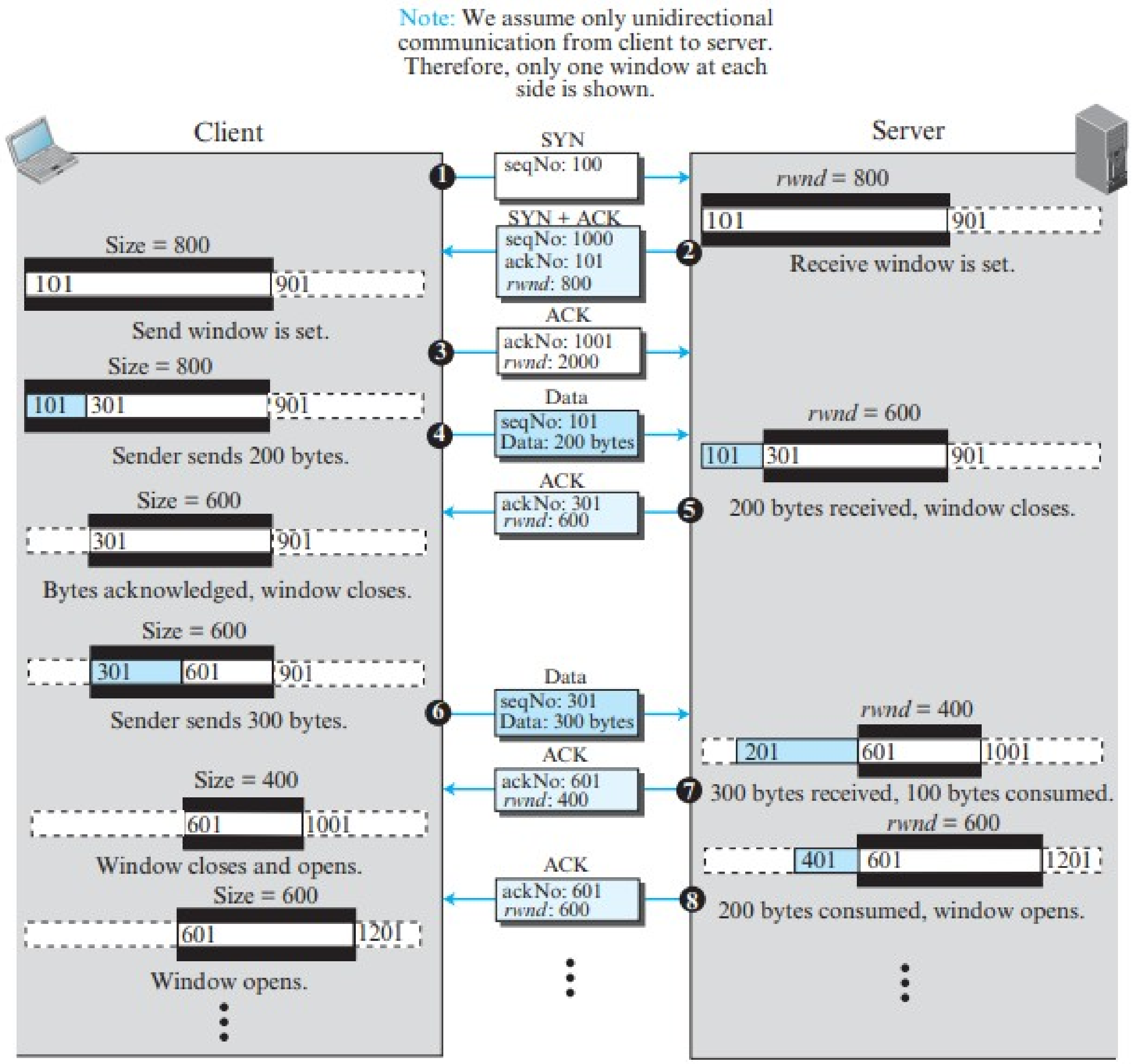


Flow control feedback from the sending TCP to the sending process (path 5) is achieved through simple rejection of data by the sending TCP when its window is full. This means that our discussion of flow control concentrates on the feedback sent from the receiving TCP to the sending TCP (path 4).

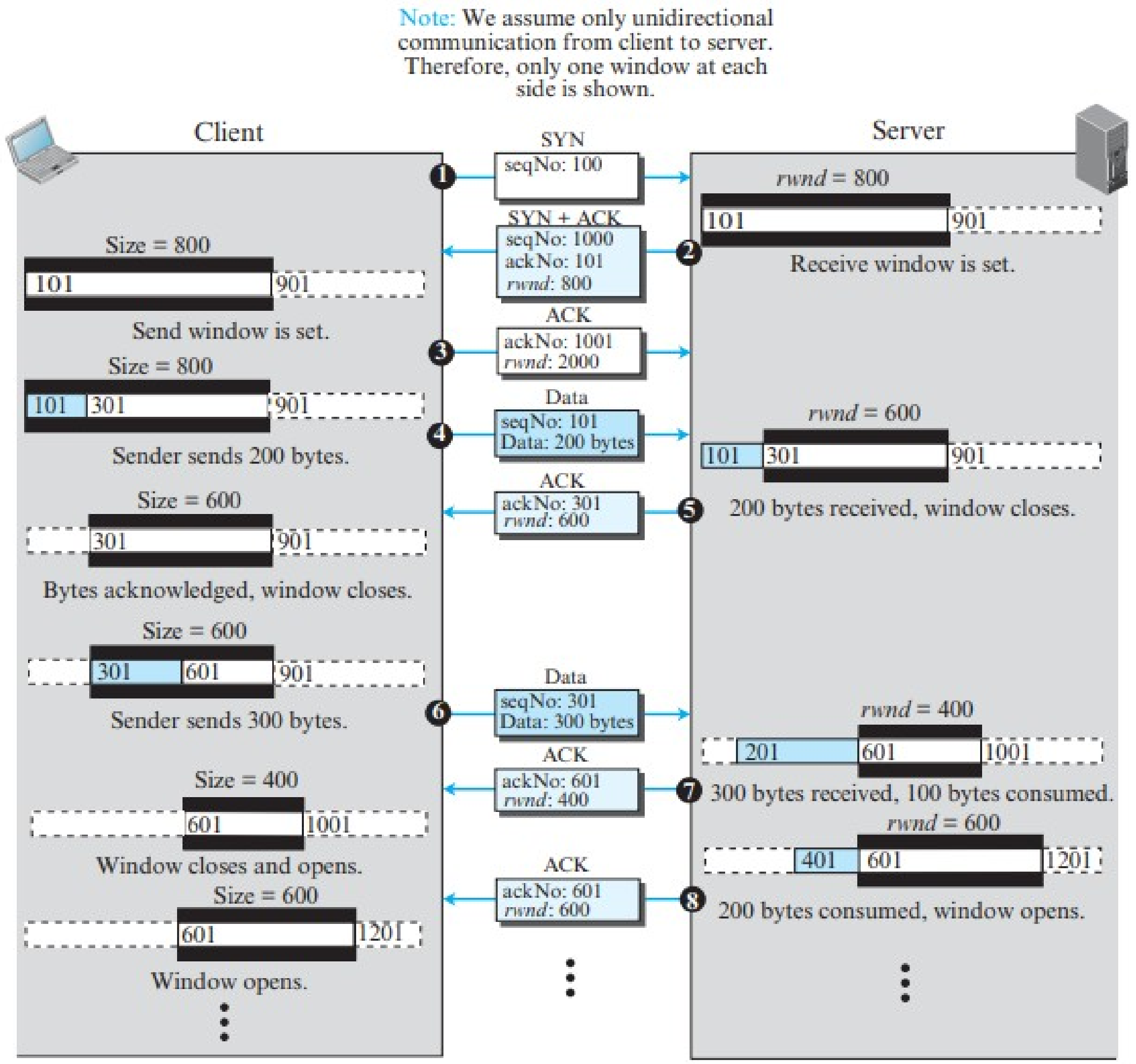
Opening and Closing of Windows

- To achieve flow control, 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 receive 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**).

Simple Scenario

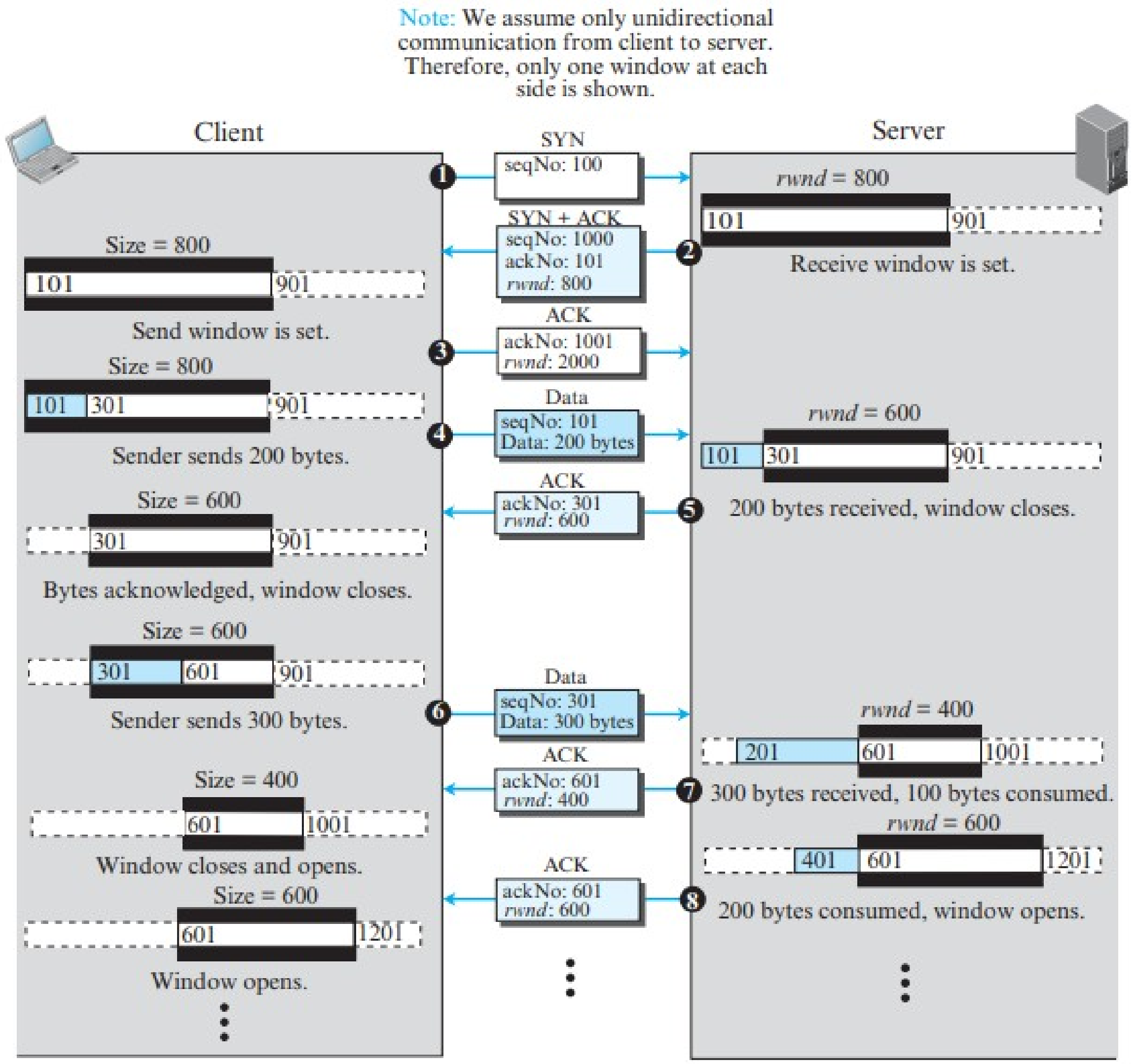


Simple Scenario



1. The first segment is from the client to the server (a SYN segment) to request connection. The client announces its initial seqNo = 100. When this segment arrives at the server, it allocates a buffer size of 800 (an assumption) and sets its window to cover the whole buffer (rwnd = 800). Note that the number of the next byte to arrive is 101.
2. The second segment is from the server to the client. This is an ACK + SYN segment. The segment uses ackNo = 101 to show that it expects to receive bytes starting from 101. It also announces that the client can set a buffer size of 800 bytes.
3. The third segment is the ACK segment from the client to the server. Note that the client has defined a rwnd of size 2000, but we do not use this value in our figure because the communication is only in one direction.

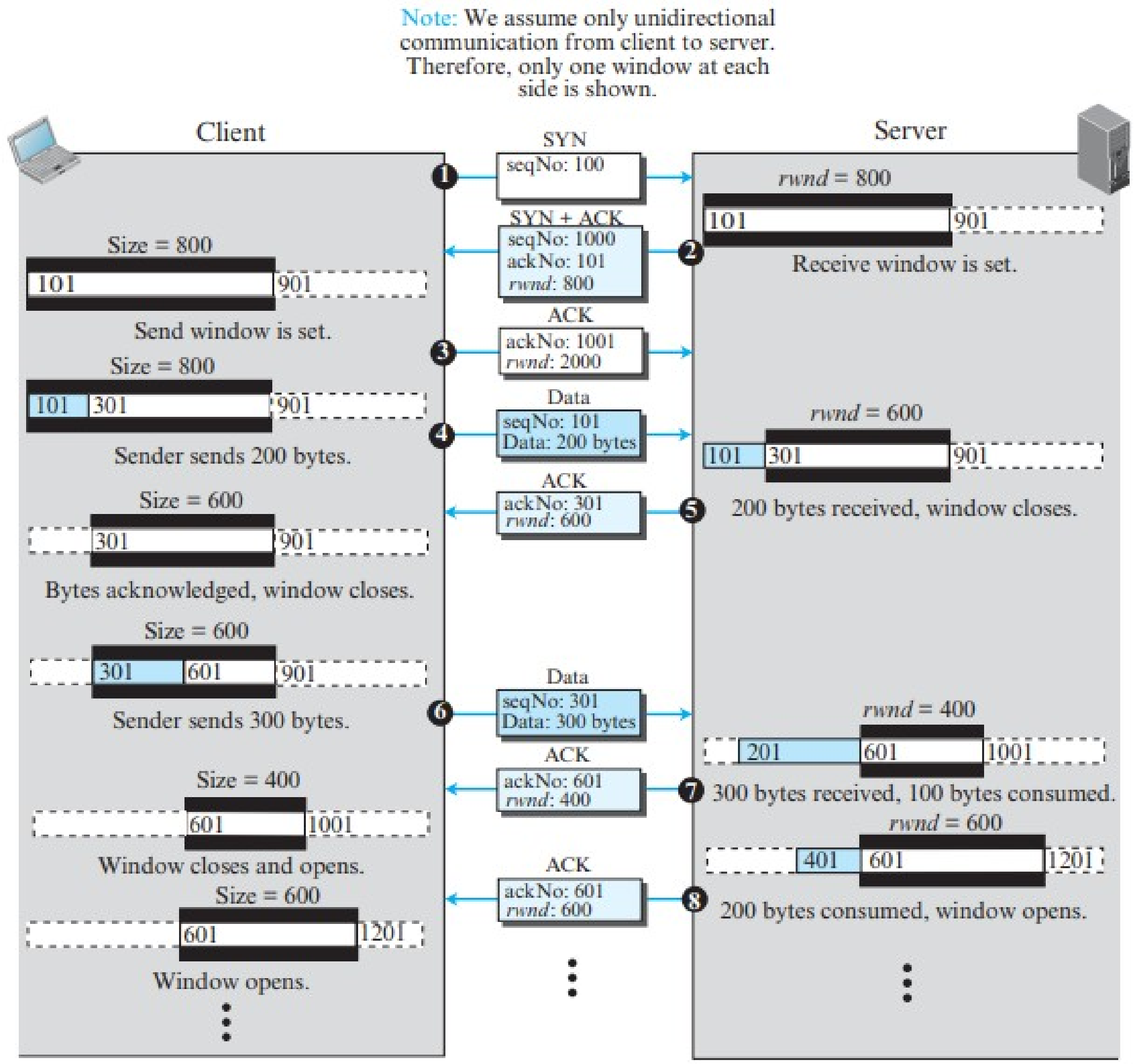
Simple Scenario



4. After the client has set its window with the size (800) dictated by the server, the process pushes 200 bytes of data. The TCP client numbers these bytes 101 to 300. It then creates a segment and sends it to the server. The segment shows the starting byte number as 101 and the segment carries 200 bytes. The window of the client is then adjusted to show that 200 bytes of data are sent but waiting for acknowledgment. When this segment is received at the server, the bytes are stored, and the receive window closes to show that the next byte expected is byte 301; the stored bytes occupy 200 bytes of buffer.

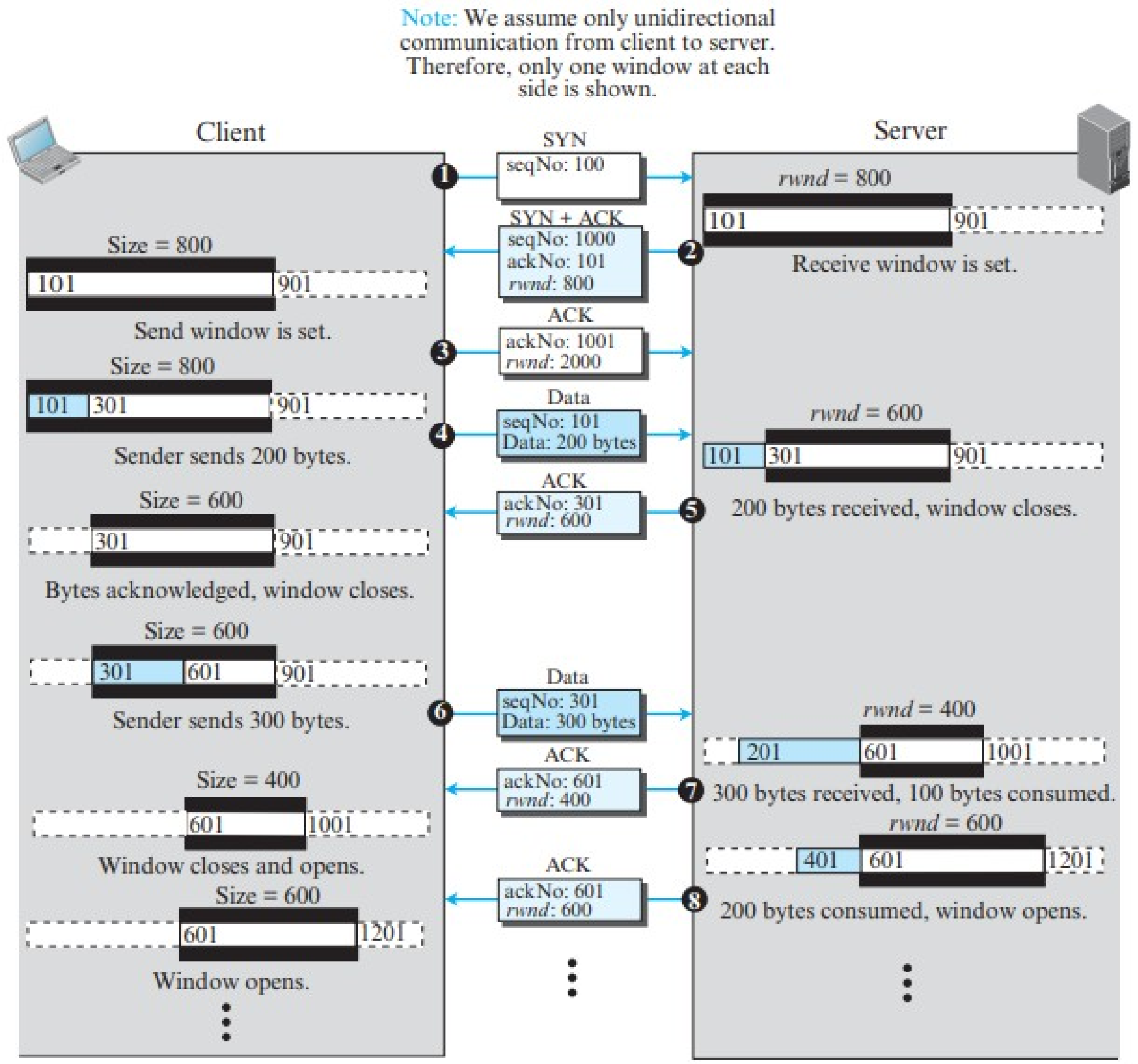
5. The fifth segment is the feedback from the server to the client. The server acknowledges bytes up to and including 300 (expecting to receive byte 301). The segment also carries the size of the receive window after decrease (600). The client, after receiving this segment, purges the acknowledged bytes from its window and closes its window to show that the next byte to send is byte 301. The window size, however, decreases to 600 bytes. Although the allocated buffer can store 800 bytes, the window cannot open (moving its right wall to the right) because the receiver does not let it.

Simple Scenario



6. Segment 6 is sent by the client after its process pushes 300 more bytes. The segment defines seqNo as 301 and contains 300 bytes. When this segment arrives at the server, the server stores them, but it has to reduce its window size. After its process has pulled 100 bytes of data, the window closes from the left for the amount of 300 bytes, but opens from the right for the amount of 100 bytes. The result is that the size is only reduced 200 bytes. The receiver window size is now 400 bytes.
7. In segment 7, the server acknowledges the receipt of data, and announces that its window size is 400. When this segment arrives at the client, the client has no choice but to reduce its window again and set the window size to the value of rwnd = 400 advertised by the server. The send window closes from the left by 300 bytes, and opens from the right by 100 bytes.

Simple Scenario



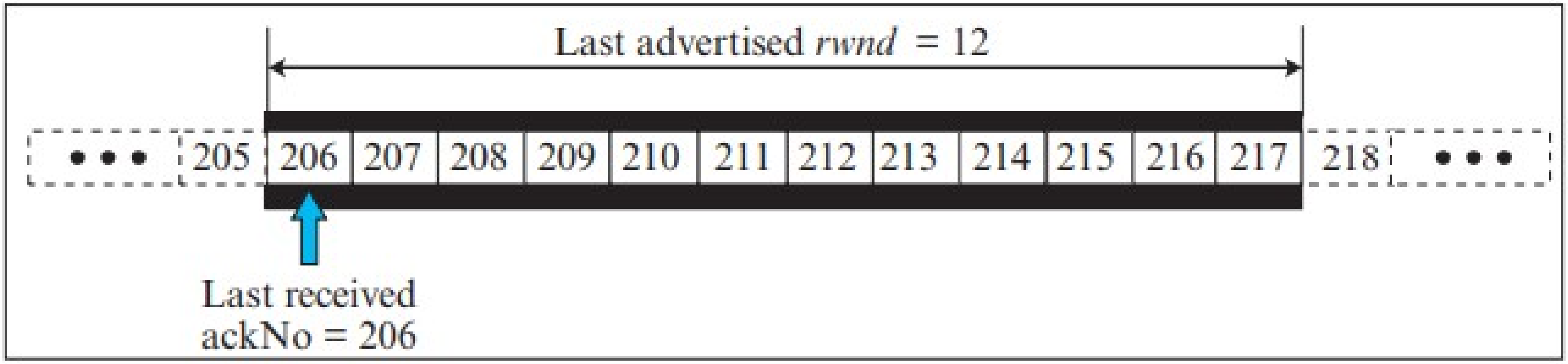
8. Segment 8 is also from the server after its process has pulled another 200 bytes. Its window size increases. The new rwnd value is now 600. The segment informs the client that the server still expects byte 601, but the server window size has expanded to 600. We need to mention that the sending of this segment depends on the policy imposed by the implementation. Some implementations may not allow advertisement of the rwnd at this time; the server then needs to receive some data before doing so. After this segment arrives at the client, the client opens its window by 200 bytes without closing it. The result is that its window size increases to 600 bytes.

Shrinking of Windows

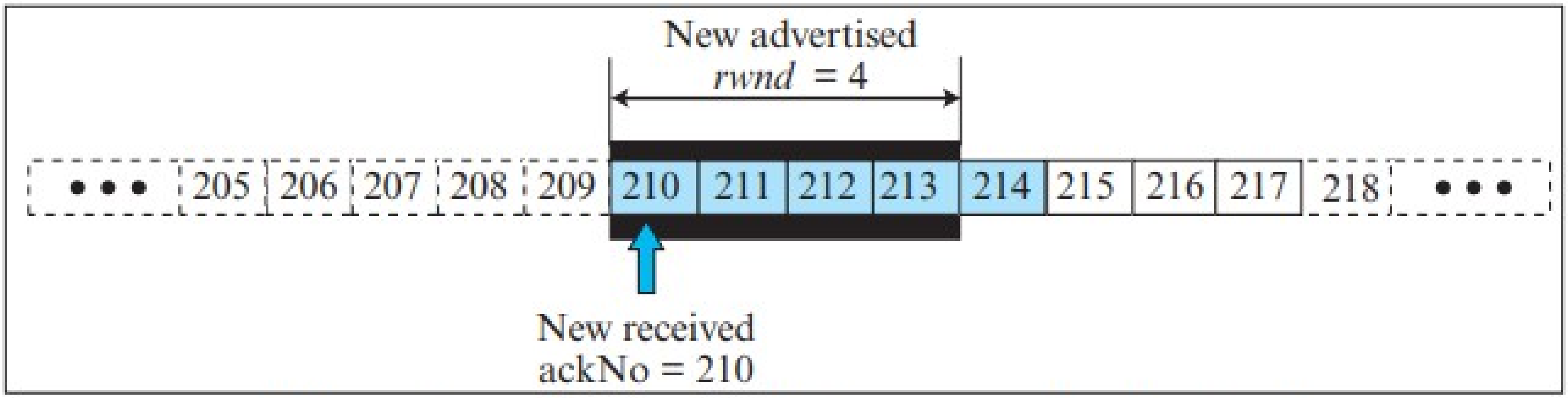
- As we said before, the receive window cannot shrink. The send window, on the other hand, can shrink if the receiver defines a value for rwnd that results in shrinking the window.
- However, some implementations do not allow shrinking of the send window. The limitation does not allow the right wall 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.

$$\text{new ackNo} + \text{new } \mathit{rwnd} \geq \text{last ackNo} + \text{last } \mathit{rwnd}$$

Shrinking of Windows



a. The window after the last advertisement



b. The window after the new advertisement; window has shrunk

Window Shutdown

- We said that shrinking the send window by moving its right wall to the left is strongly discouraged. However, there is one exception: the receiver can temporarily shut down the window by sending a rwnd of 0.
- This can happen if for some reason the receiver does not want to receive any data from the sender for a while. In this case, the sender does not actually shrink the size of the window, but stops sending data until a new advertisement has arrived.

Silly Window Syndrome

- A serious problem can arise in the sliding window operation when either the sending application program creates data slowly or the receiving application program consumes data slowly, or both.
- Any of these situations results in the sending of data in very small segments, which reduces the efficiency of the operation. This problem is called the **Silly Window Syndrome.**

Silly Window Syndrome: Created by the Sender

- The sending TCP may create a silly window syndrome if it is serving an application program that creates data slowly, for example, 1 byte at a time.
- The application program writes 1 byte at a time into the buffer of the sending TCP. If the sending TCP does not have any specific instructions, it may create segments containing 1 byte of data.
- The solution is to prevent the sending TCP from sending the data byte by byte. The sending TCP must be forced to wait and collect data to send in a larger block.
- But how long should sending TCP wait?

Silly Window Syndrome: Created by the Sender

- **Nagle's Algorithm -**
  1. The sending TCP sends the first piece of data it receives from the sending application program even if it is only 1 byte.
  2. After sending the first segment, the sending TCP accumulates data in the output buffer and waits until either the receiving TCP sends an acknowledgment or until enough data has accumulated to fill a maximum-size segment. At this time, the sending TCP can send the segment.
  3. Step 2 is repeated for the rest of the transmission. Segment 3 is sent immediately if an acknowledgment is received for segment 2, or if enough data have accumulated to fill a maximum-size segment.
- The elegance of Nagle's algorithm is in its simplicity and in the fact that it takes into account the speed of the application program that creates the data and the speed of the network that transports the data. If the application program is faster than the network, the segments are larger (maximum-size segments). If the application program is slower than the network, the segments are smaller (less than the maximum segment size).

Silly Window Syndrome: Created by the Receiver

- The receiving TCP may create a silly window syndrome if it is serving an application program that consumes data slowly, for example, 1 byte at a time.
- Suppose that the sending application program creates data in blocks of 1 kilobyte, but the receiving application program consumes data 1 byte at a time. Also suppose that the input buffer of the receiving TCP is 4 kilobytes.
- The sender sends the first 4 kilobytes of data. The receiver stores it in its buffer. Now its buffer is full. It advertises a window size of zero, which means the sender should stop sending data.
- The receiving application reads the first byte of data from the input buffer of the receiving TCP. Now there is 1 byte of space in the incoming buffer. The receiving TCP announces a window size of 1 byte, which means that the sending TCP, which is eagerly waiting to send data, takes this advertisement as good news and sends a segment carrying only 1 byte of data.
- The procedure will continue. One byte of data is consumed and a segment carrying 1 byte of data is sent. Again we have an efficiency problem and the silly window syndrome.

Silly Window Syndrome: Created by the Receiver

- Two solutions have been proposed to prevent the silly window syndrome created by an application program that consumes data more slowly than they arrive.
- The first solution **(Clark's solution)** is to send an acknowledgment as soon as the data arrive, but to announce a window size of zero until either there is enough space to accommodate a segment of maximum size or until at least half of the receive buffer is empty.
- The second solution is to delay sending the acknowledgment. This means that when a segment arrives, it is not acknowledged immediately. The receiver waits until there is a decent amount of space in its incoming buffer before acknowledging the arrived segments.

Error Control in TCP

- TCP is a reliable transport-layer protocol. This means that an application program that delivers a stream of data to TCP relies on TCP to deliver the entire stream to the application program on the other end in order, without error, and without any part lost or duplicated.
- Error control includes mechanisms for detecting and resending corrupted segments, resending lost segments, storing out-oforder segments until missing segments arrive, and detecting and discarding duplicated segments.
- Error control in TCP is achieved through the use of three simple tools: checksum, acknowledgment, and time-out.

ACKNOWLEDGMENT

- TCP uses acknowledgments to confirm the receipt of data segments. Control segments like SYN that carry no data, but consume a sequence number, are also acknowledged.
- ACK segments are never acknowledged.
- TCP uses two types of acknowledgments -
  - Cummulative Acknowledgment
  - Selective Acknowledgment

Cummulative Acknowledgment (ACK)

- 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 (SACK)

- Newer implementations of TCP are adding another type of acknowdgment 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.

GENERATING ACKNOWLEDGMENTS

- When does a receiver generate acknowledgments?
  1. When end A sends a data segment to end B, it must include (piggyback) an acknowledgment that gives the next sequence number it expects to receive. This rule decreases the number of segments needed and therefore reduces traffic.
  2. When the receiver has no data to send and it receives an in-order segment (with expected sequence number) and the previous segment has already been acknowledged, the receiver delays sending an ACK segment until another segment arrives or until a period of time (normally 500 ms) has passed. In other words, the receiver needs to delay sending an ACK segment if there is only one outstanding in-order segment. This rule reduces ACK segments.

Generating Acknowledgments

- When does a receiver generate acknowledgments?

3. When a segment arrives with a sequence number that is expected by the receiver, and the previous in-order segment has not been acknowledged, the receiver immediately sends an ACK segment. In other words, there should not be more than two in-order unacknowledged segments at any time. This prevents the unnecessary retransmission of segments that may create congestion in the network.
4. 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.
5. 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.
6. 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.

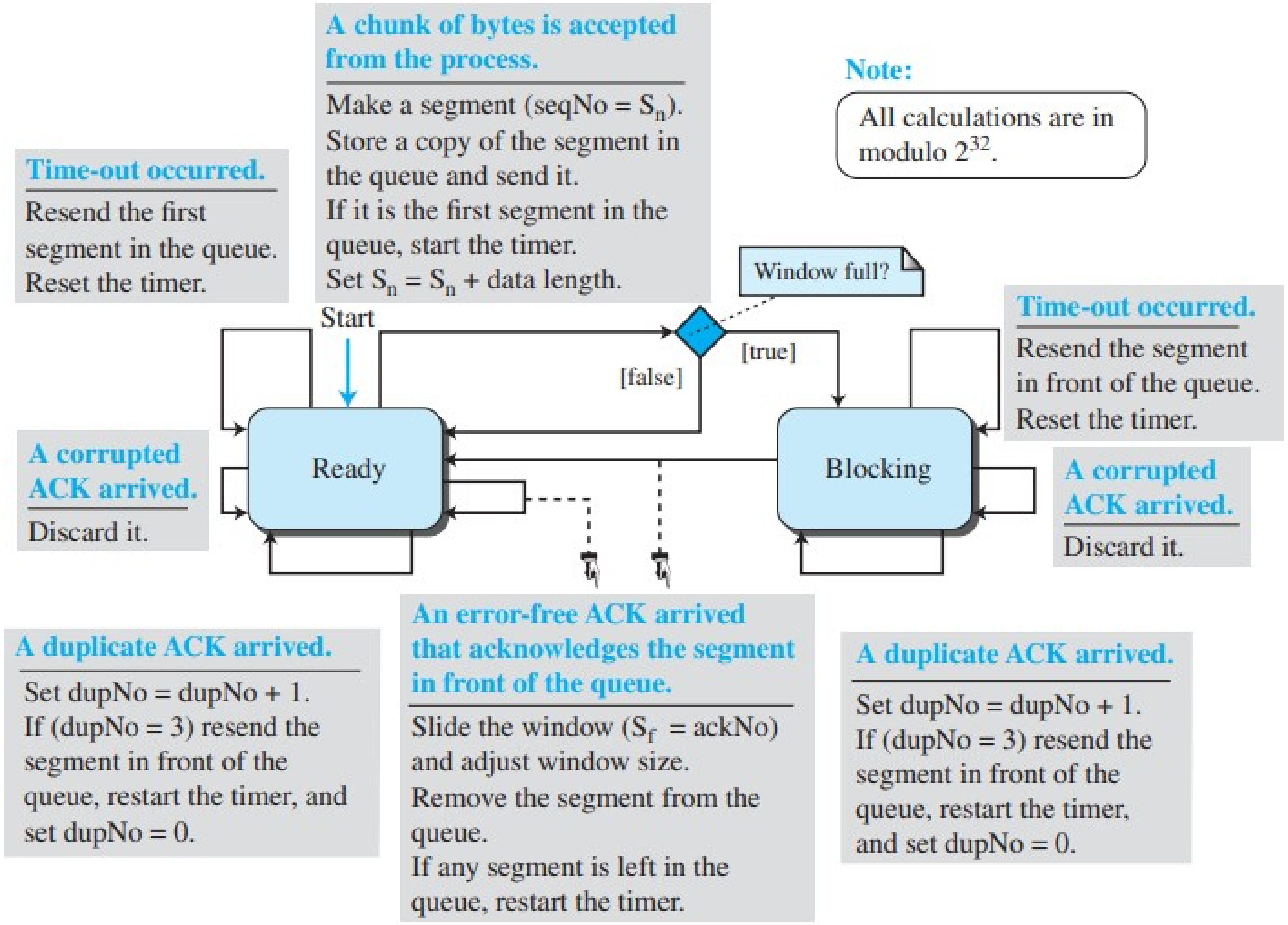
Retransmission of Segments

- **Retransmission after RTO**
  - The sending TCP maintains one **retranmission time-out (RTO)** for each connection. When there is time out, TCP resends the segment with the smallest sequence number and restarts the timer.
- **Retransmission after Three Duplicate ACK Segments**
  - To expedite service throughout the Internet by allowing senders to retransmit without waiting for a time out, most implementations today follow the three duplicate ACKs rule and retransmit the missing segment immediately. This feature is called fast retransmission.
  - In this version, if three duplicate acknowledgments (i.e., an original ACK plus three exactly identical copies) arrive for a segment, the next segment is retransmitted without waiting for the time-out.

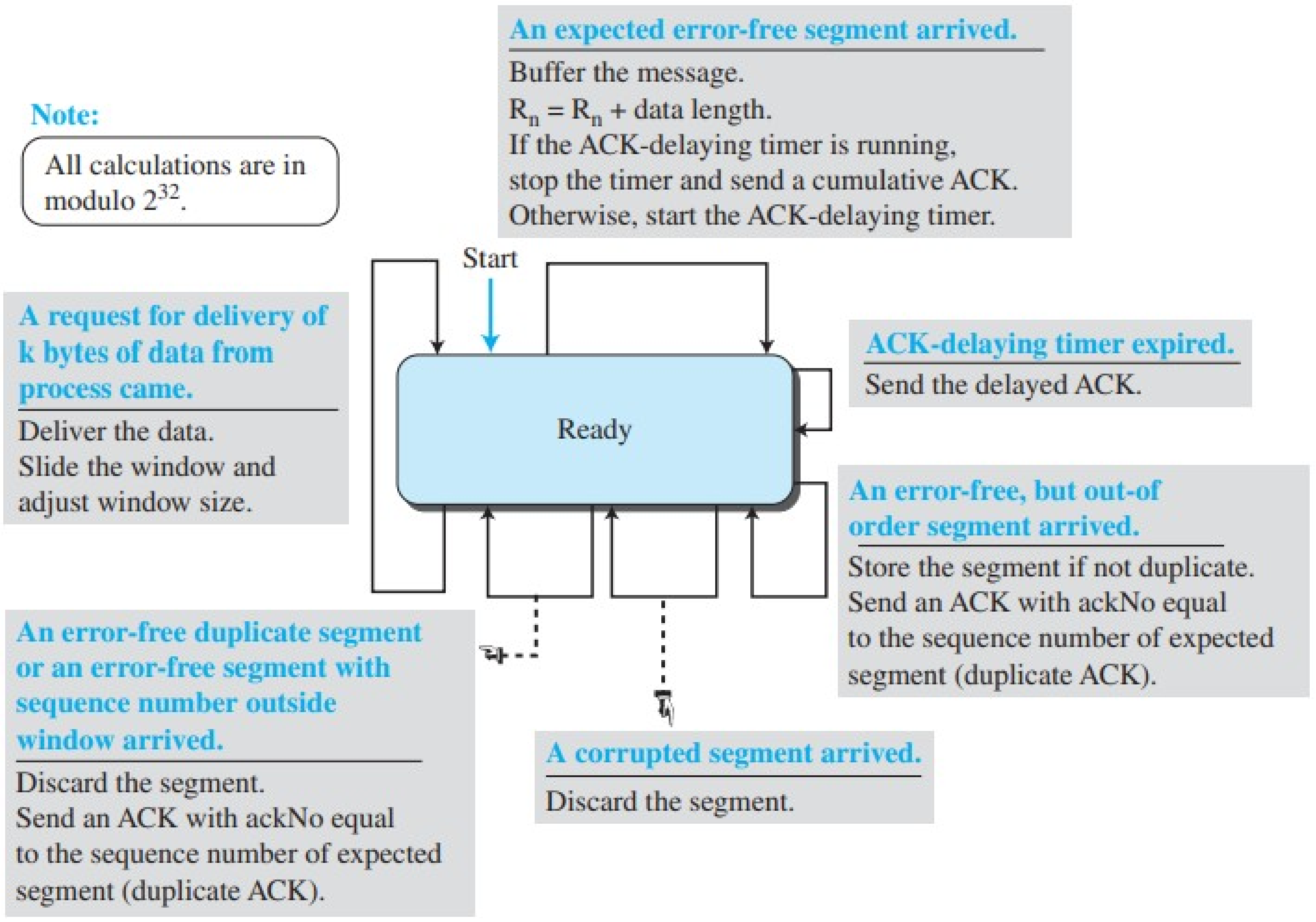
FSMs for Data Transfer in TCP

- Data transfer in TCP is close to the Selective-Repeat protocol with a slight similarity to GBN.
- Since TCP accepts out-of-order segments, TCP can be thought of as behaving more like the SR protocol, but since the original acknowledgments are cumulative, it looks like GBN.
- However, if the TCP implementation uses SACKs, then TCP is closest to SR.

Sender-Side FSM


A chunk of bytes is accepted from the process.
Make a segment (seqNo = S_n). Store a copy of the segment in the queue and send it. If it is the first segment in the queue, start the timer. Set $S_n = S_n$ + data length.
Note:
All calculations are in modulo 2^{32}.
Time-out occurred.
Resend the first segment in the queue. Reset the timer.
Window full?
Start
[false]
[true]
Time-out occurred.
Resend the segment in front of the queue. Reset the timer.
A corrupted ACK arrived.
Discard it.
Ready
Blocking
A corrupted ACK arrived.
Discard it.
A duplicate ACK arrived.
Set dupNo = dupNo + 1. If (dupNo = 3) resend the segment in front of the queue, restart the timer, and set dupNo = 0.
An error-free ACK arrived that acknowledges the segment in front of the queue.
Slide the window (S_f = ackNo) and adjust window size. Remove the segment from the queue. If any segment is left in the queue, restart the timer.
A duplicate ACK arrived.
Set dupNo = dupNo + 1. If (dupNo = 3) resend the segment in front of the queue, restart the timer, and set dupNo = 0.

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).

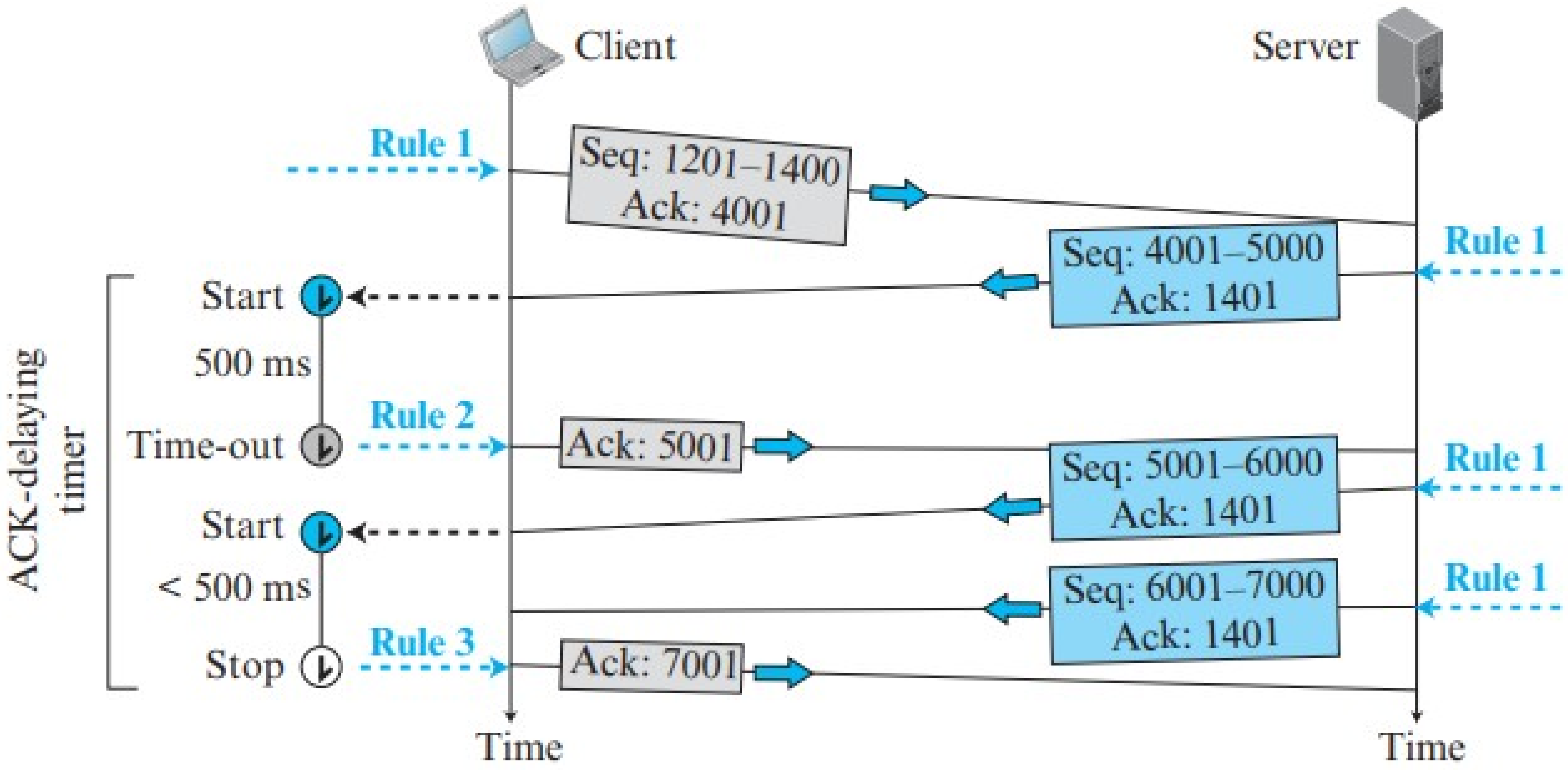
An error-free duplicate segment or an error-free segment with sequence number outside window arrived.

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

A corrupted segment arrived.

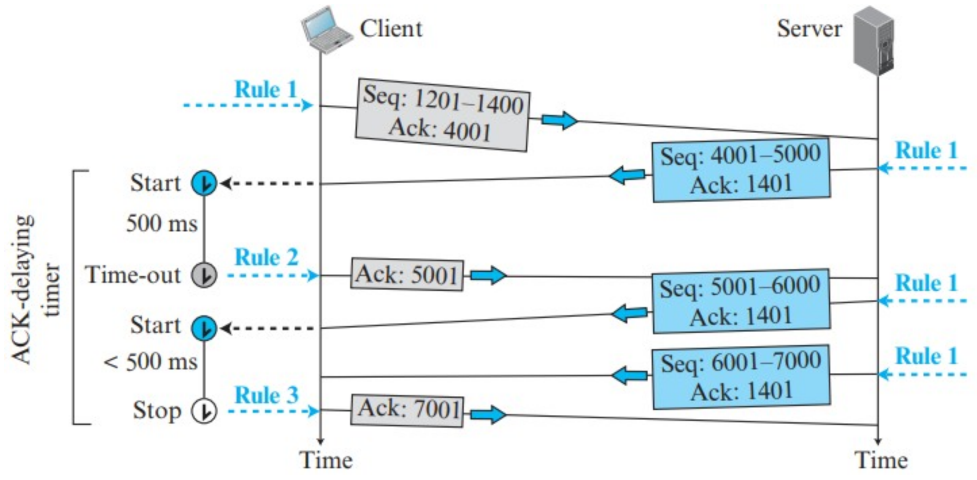
Discard the segment.

Normal Operation Scenario

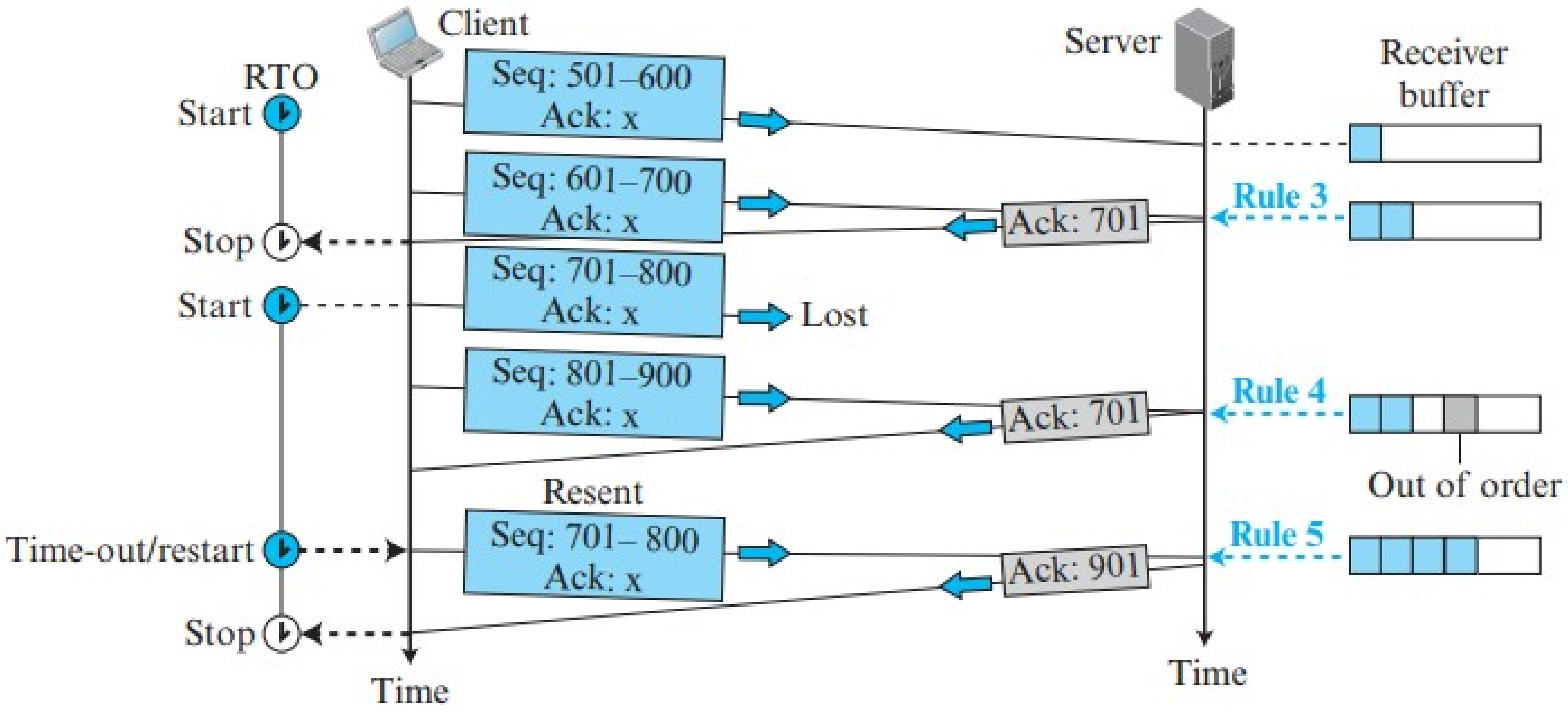


Normal Operation Scenario

- The client TCP sends one segment; the server TCP sends three. The figure shows which rule applies to each acknowledgment. At the server site, only rule 1 applies. There are data to be sent, so the segment displays the next byte expected.
- 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. However, according to rule 2, 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 if 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 based on rule 3. We have not shown the RTO timer because no segment is lost or delayed. We just assume that the RTO timer performs its duty.

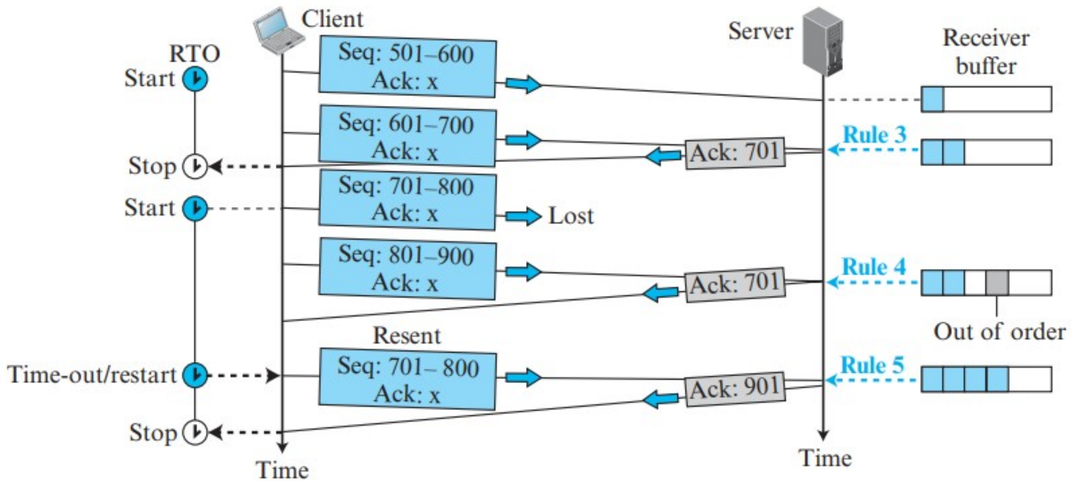


Lost Segment Scenario

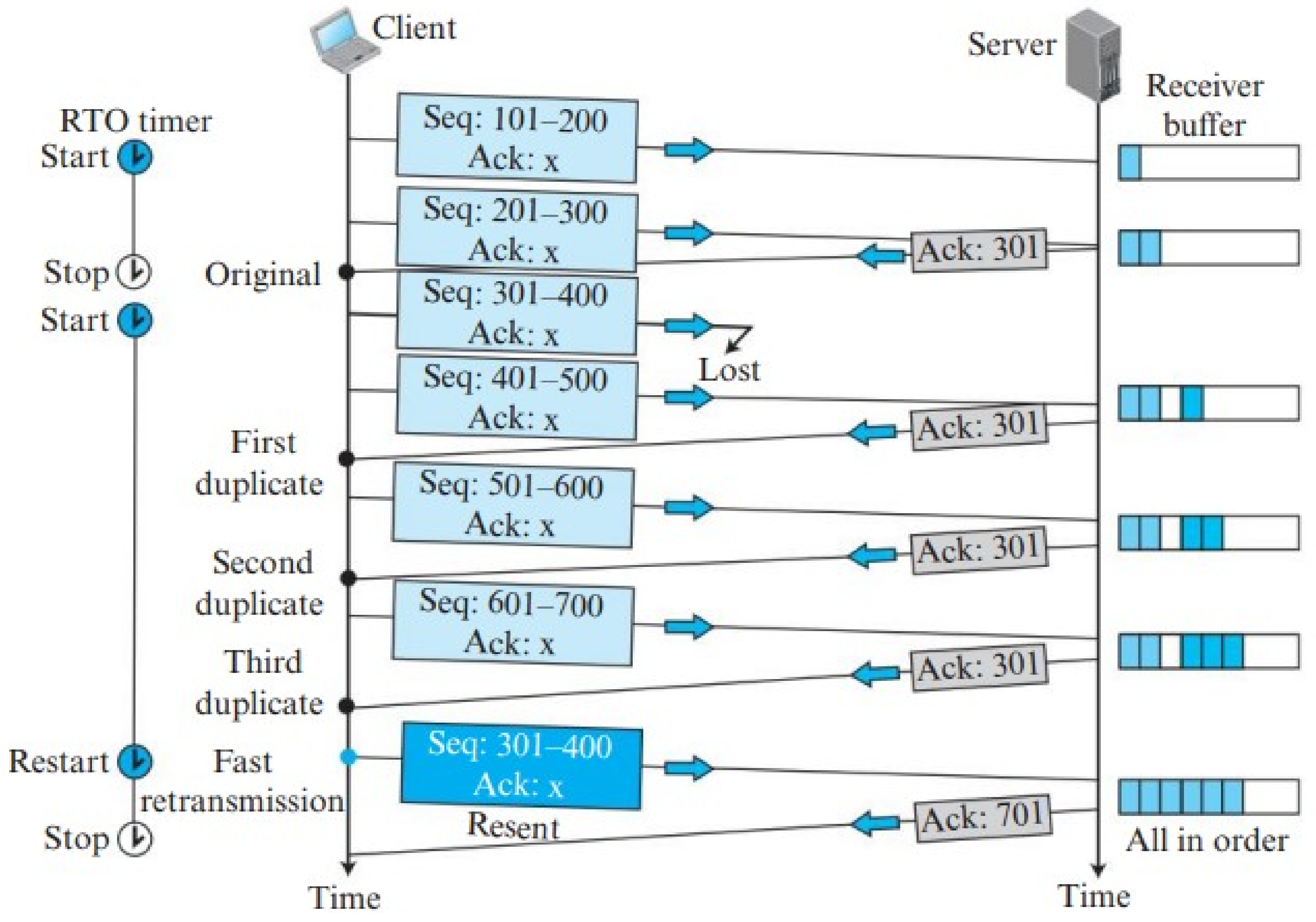


Lost Segment Scenario

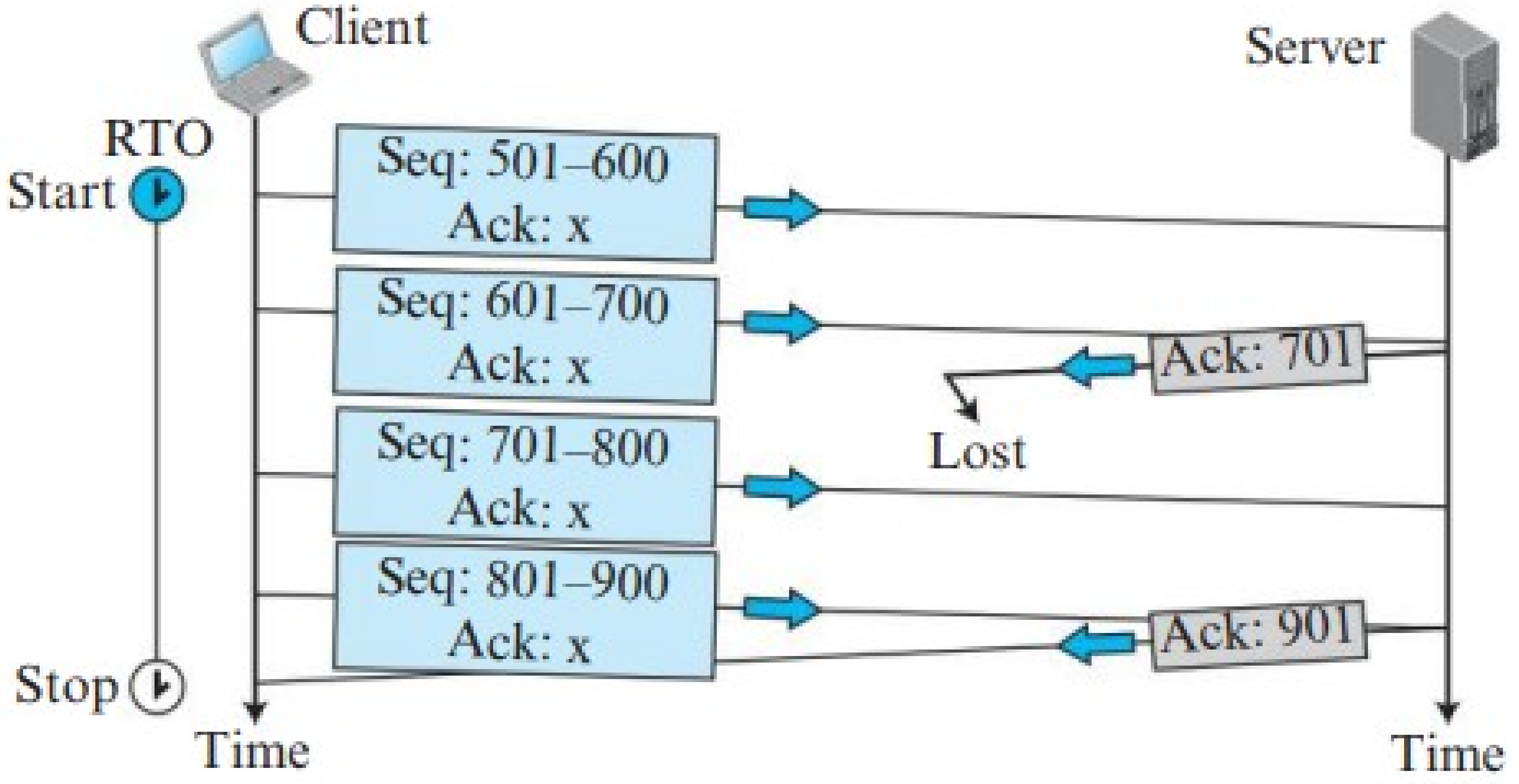
- In this scenario, we show what happens when a segment is lost or corrupted. A lost or corrupted segment is treated the same way by the receiver. A lost segment is discarded somewhere in the network; a corrupted segment is discarded by the receiver itself. Both are considered lost.
- We are assuming that data transfer is unidirectional: one site is sending, the other receiving. In our scenario, the sender sends segments 1 and 2, which are acknowledged immediately by an ACK (rule 3).
- Segment 3, however, is lost. The receiver receives segment 4, which is out of order. The receiver stores the data in the segment in its buffer but leaves a gap to indicate that there is no continuity in the data. The receiver immediately sends an acknowledgment to the sender displaying the next byte it expects (rule 4). Note that the receiver stores bytes 801 to 900, but never delivers these bytes to the application until the gap is filled.



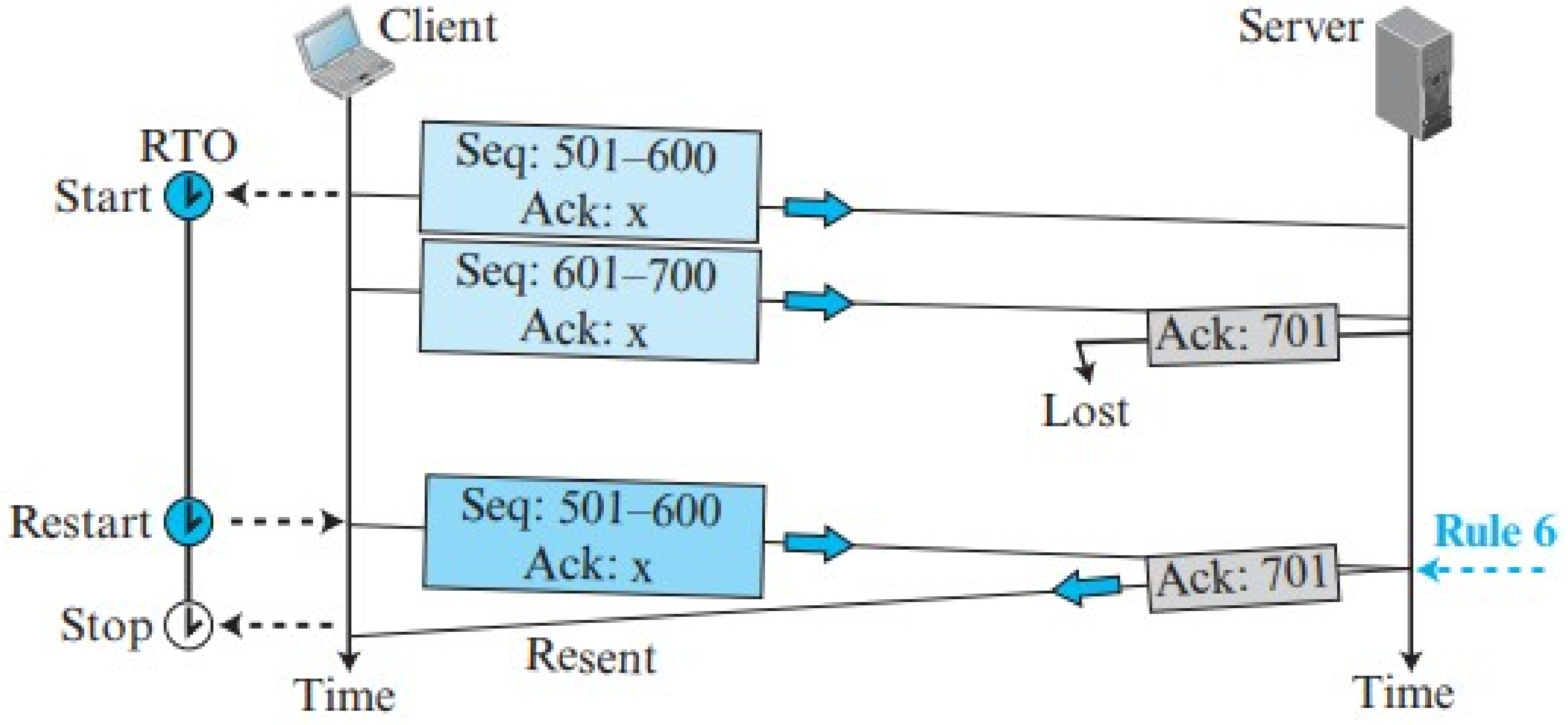
Fast Retransmission Scenario



Automatically Corrected Lost ACK



Lost ACK Corrected by Resending a Segment



Deadlock Created by Lost Acknowledgment

- There is one situation in which loss of an acknowledgment may result in system deadlock. This is the case in which a receiver sends an acknowledgment with rwnd set to 0 and requests that the sender shut down its window temporarily.
- After a while, the receiver wants to remove the restriction; however, if it has no data to send, it sends an ACK segment and removes the restriction with a nonzero value for rwnd.
- A problem arises if this acknowledgment is lost. The sender is waiting for an acknowledgment that announces the nonzero rwnd. The receiver thinks that the sender has received this and is waiting for data.
- This situation is called a deadlock; each end is waiting for a response from the other end and nothing is happening. A retransmission timer is not set.