Transmission Control Protocol(TCP)

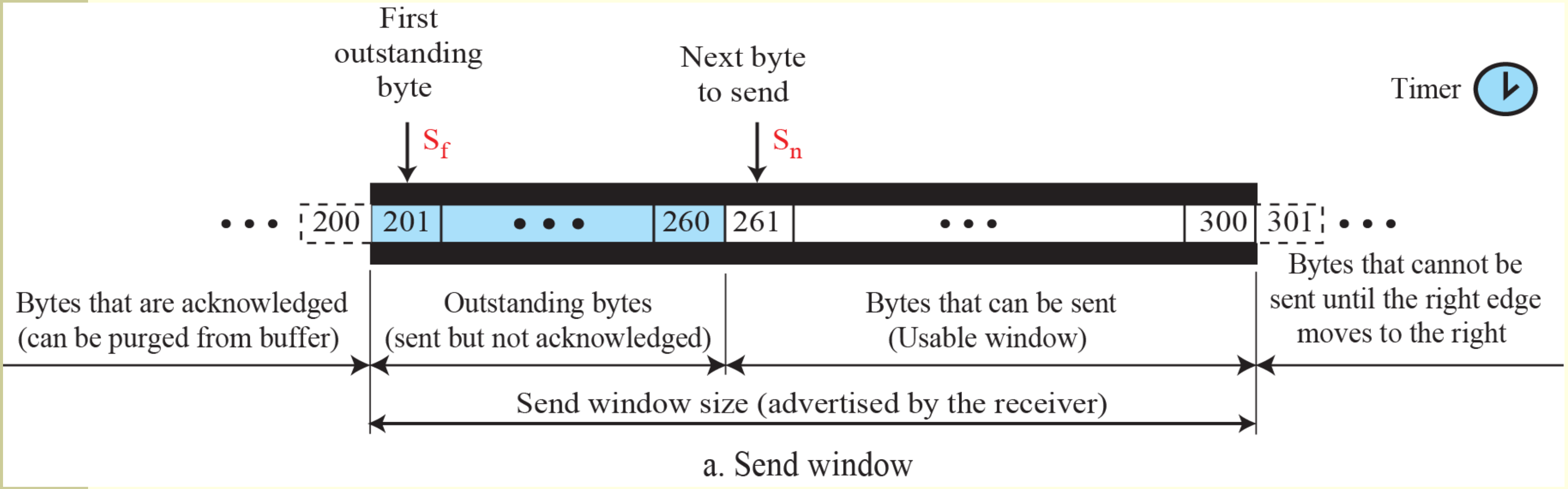
Kumkum Saxena

3.4.6 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 unidirectional; the bidirectional communication can be inferred using two unidirectional communications with piggybacking.

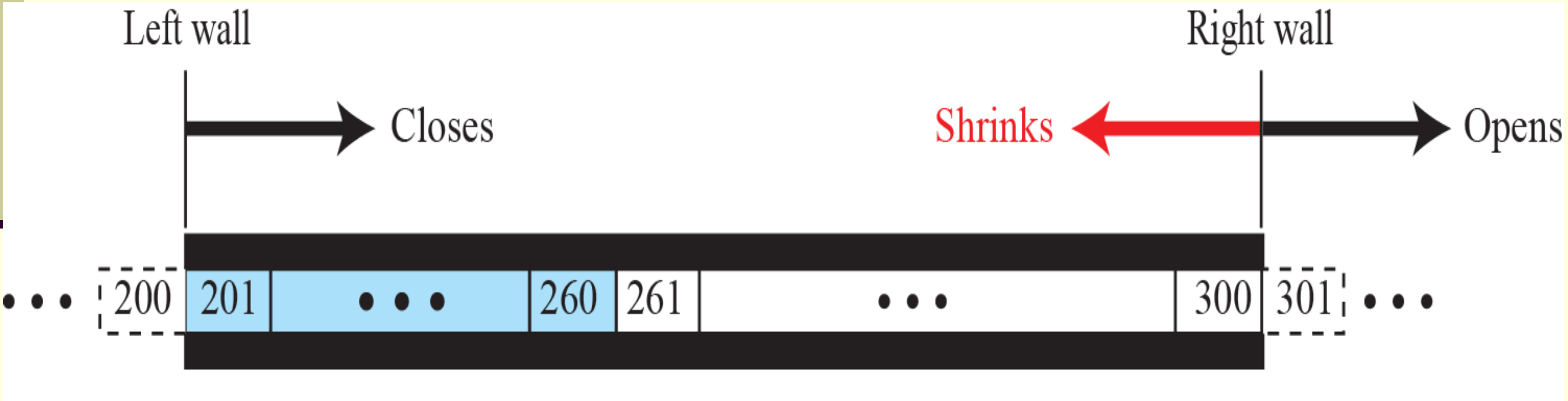
- ***Send Window***
- ***Receive Window***

Figure 3.54: Send window in TCP



First outstanding byte
S_f
Next byte to send
S_n
Timer
• • • 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

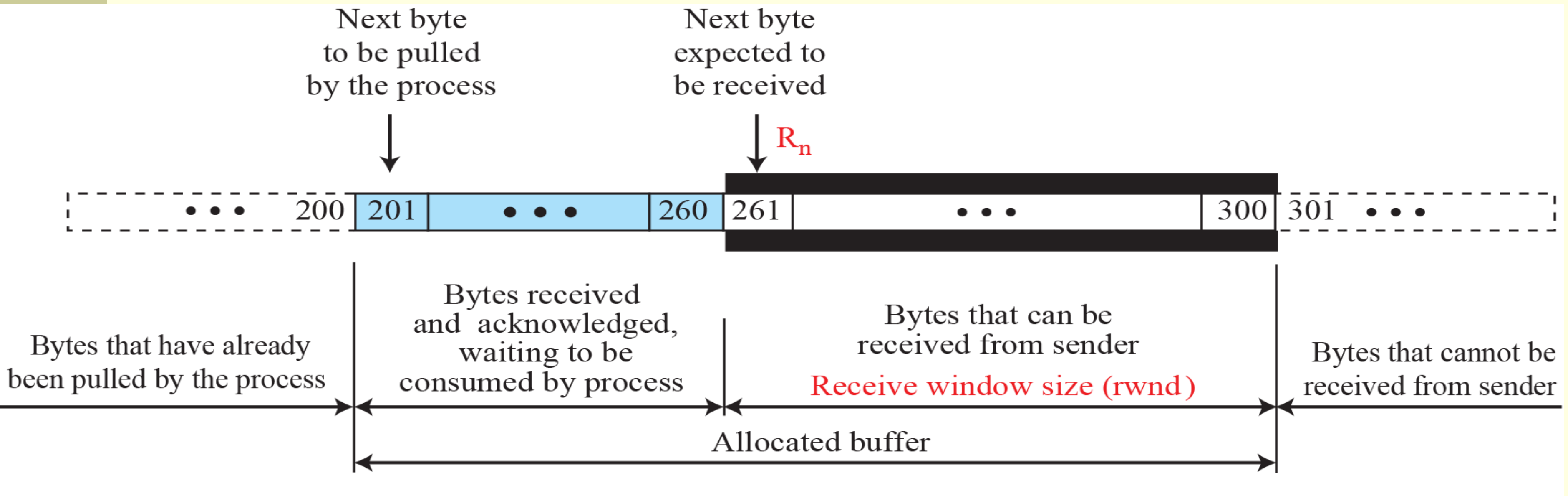


b. Opening, closing, and shrinking send window

Difference between send window in SR and TCP

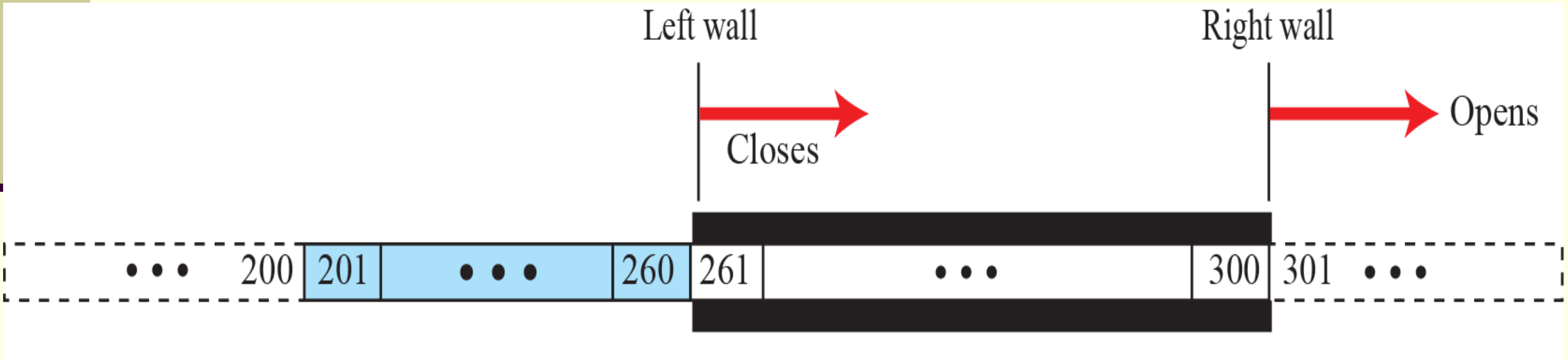
- Window size
  - SR---No. Of packets
  - TCP--- No. Of Bytes.
- TCP(SR) store data and send them later. But TCP is capable of sending segments of data as soon as it receives them from the process.
- Timers
  - SR-several timers for each packet sent
  - TCP- only one timer.

Figure 3.55: 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

Difference between Receive window in SR and TCP

- TCP allows the receiving process to pull data at its own pace...
    - rwnd=buff_size-no. Of bytes waiting to be pulled.
- Acknowledgements
    - SR-selective
    - TCP- cumulative

3.4.7 Flow Control

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. In this section we discuss flow control, ignoring error control. We assume that the logical channel between the sending and receiving TCP is error-free.

3.4.7 (continued)

- ❑ Opening and Closing Windows
  - ❖ A Scenario
- ❑ Shrinking of Windows
  - ❖ Window Shutdown
- ❑ Silly Window Syndrome
  - ❖ Syndrome Created by the Sender
  - ❖ Syndrome Created by the Receiver

Figure 3.56: Data flow and flow control feedbacks in TCP

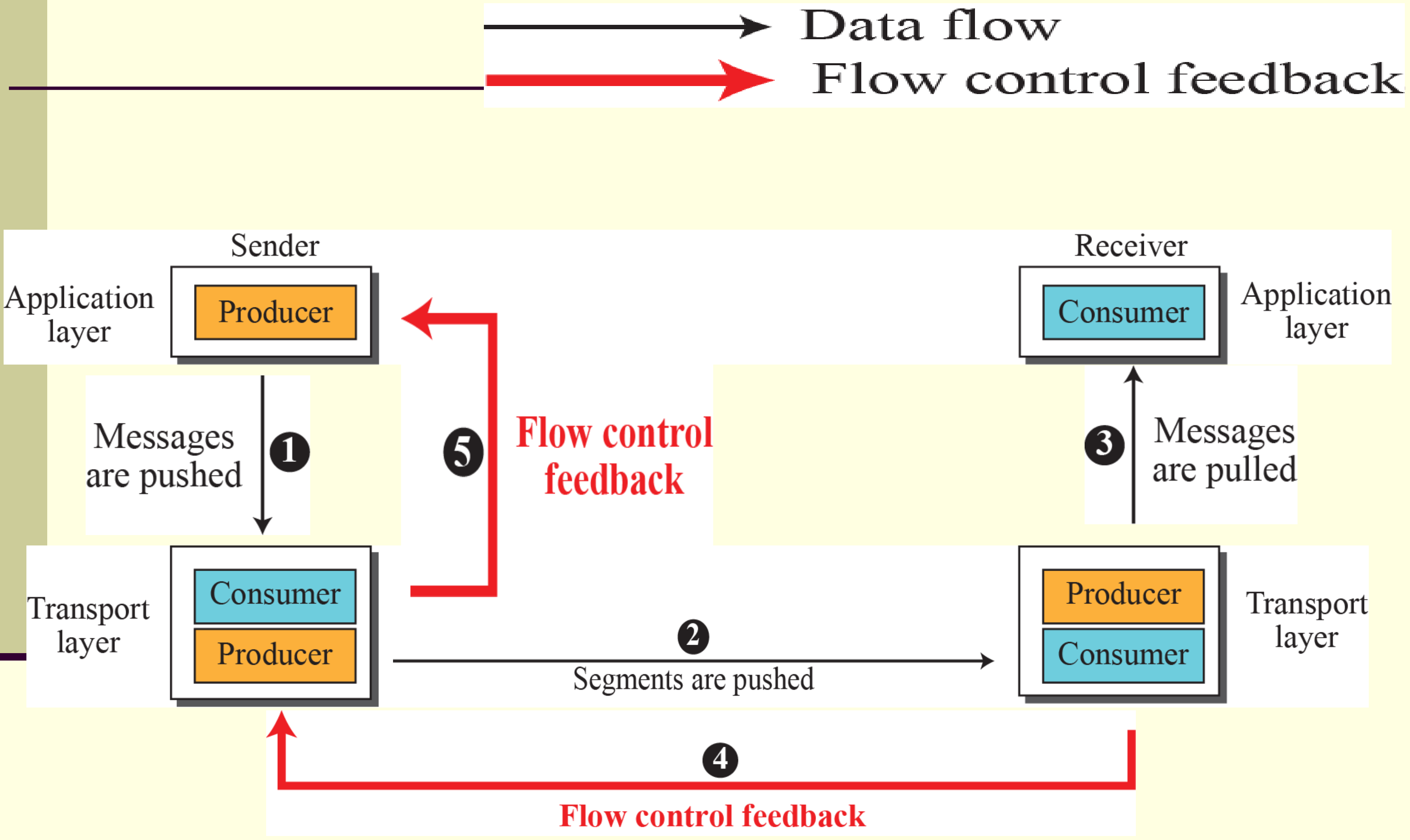
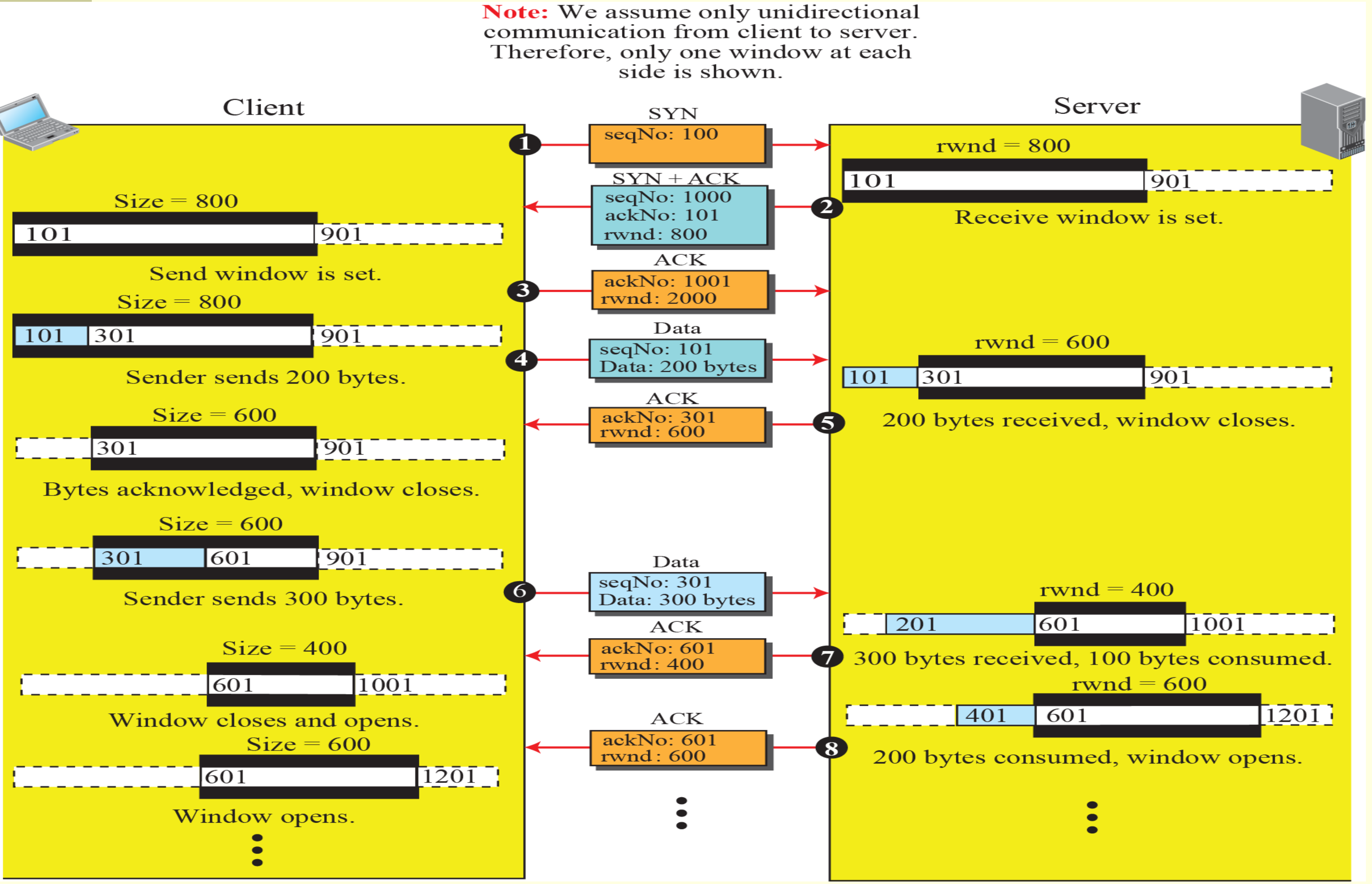


Figure 3.57: An example of flow control



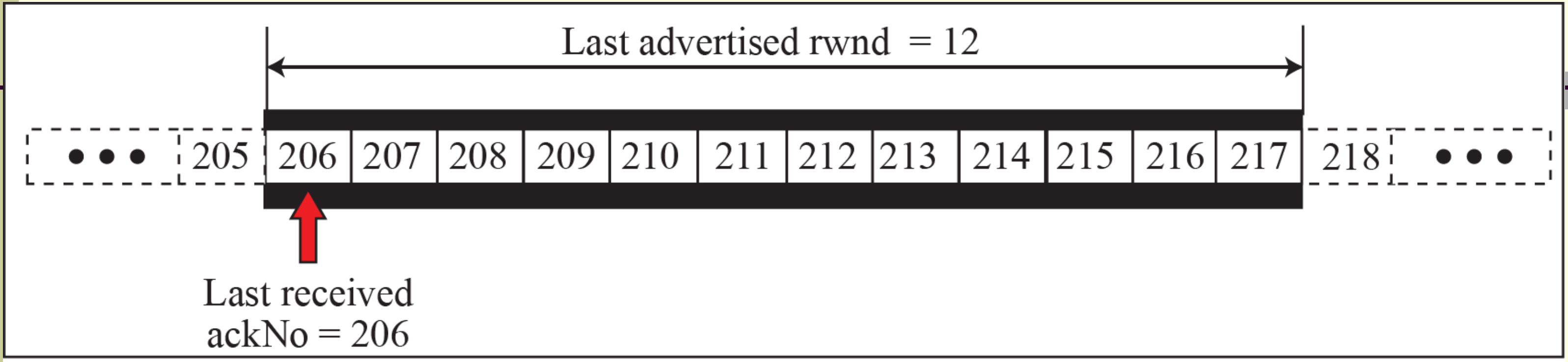
Example 3.18

Figure 3.58 shows the reason for this mandate.
Part a of the figure shows the values of the last acknowledgment and rwnd. Part b shows the situation in which the sender has sent bytes 206 to 214. Bytes 206 to 209 are acknowledged and purged. The new advertisement, however, defines the new value of rwnd as 4, in which 210 + 4 < 206 + 12. When the send window shrinks, it creates a problem: byte 214, which has already been sent, is outside the window. The relation discussed before forces the receiver to maintain the right-hand wall of the window to be as shown in part a, because the receiver does not know which of the bytes 210 to 217 has already been sent. described above.

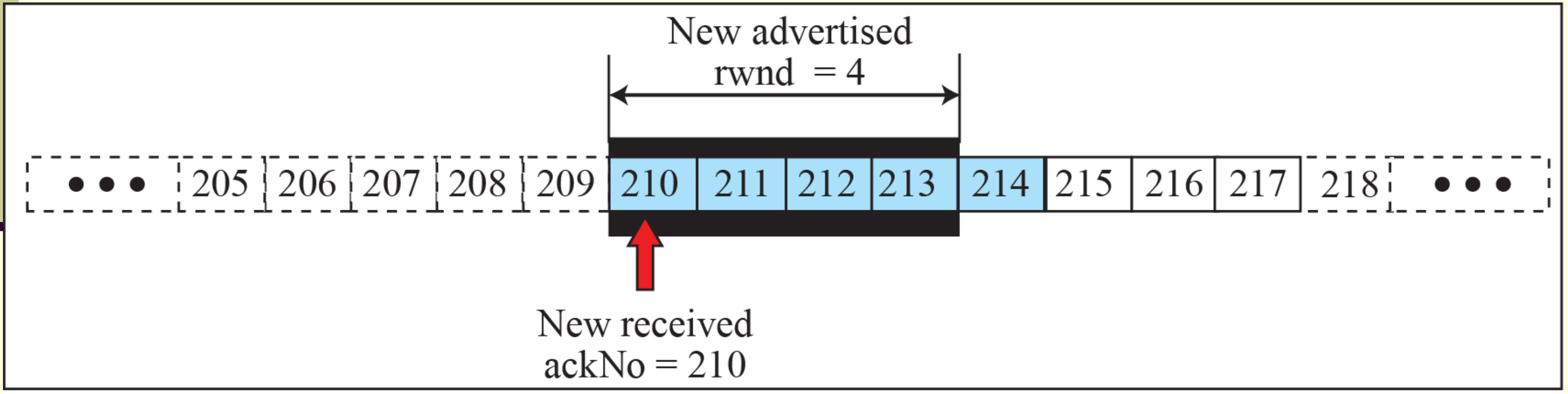
Figure 3.58: Example 3.18

Prevent the shrinking of the send window:

new ackNo + new rwnd >= last ackNo + last rwnd



a. The window after the last advertisement



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

The ‘silly window’ syndrome

This situation arises when data-bytes are sent one-at-a-time within TCP/IP packets
Such packets have at least 20 bytes of IP Header plus 20 bytes of TCP Header, yet only 1 byte of data, thus a 40-to-1 ratio of “protocol overhead” to “useful information”
On Ethernet, with14 more bytes of header, this yields an efficiency-rate of under 2%

Silly Window Syndrome (1)

- **Sending data in very small segments**

1.Syndrome created by the Sender

-Sending application program creates data slowly (e.g. 1 byte at a time)

-Wait and collect data to send in a larger block

-How long should the sending TCP wait?

-Solution: Nagle’s algorithm

-Nagle’s algorithm takes into account (1) the speed of the application program that creates the data, and (2) the speed of the network that transports the data

The Nagle's algorithm

- It's a means for improving the efficiency of TCP/IP networks, by reducing the number of packets that need to be transmitted
- It combines several small-sized outgoing packets and sends them out all at once
- (It can interact poorly with TCP's 'delayed acknowledgement' algorithm, so a way to disable it is quite commonly provided)

pseudo-code

```
// A high-level description of John Nagle's algorithm for congestion control

        if there is any available data to send
            {
            if the window-size is >= MSS and available data >= MSS
                        send a complete MSS packet now
            else        {
                        if there is unacknowledged data still in transit
                             enqueue data in the send buffer until an ACK arrives
                    else send available data immediately
                        }
            }

// NOTE: Here MSS denotes the TCP socket's 'Maximum Segment Size'
```

Silly Window Syndrome (2)

1.Syndrome created by the Receiver

-Receiving application program consumes data slowly (e.g. 1 byte at a time)

-The receiving TCP announces a window size of 1 byte. The sending TCP sends only 1 byte…

-Solution 1: Clark's solution

-Sending an ACK but announcing a window size of zero until there is enough space to accommodate a segment of max. size or until half of the buffer is empty

Silly Window Syndrome (3)

-Solution 2: Delayed Acknowledgement
-The receiver waits until there is decent amount of space in its incoming buffer before acknowledging the arrived segments
-The delayed acknowledgement prevents the sending TCP from sliding its window. It also reduces traffic.
-Disadvantage: it may force the sender to retransmit the unacknowledged segments
-To balance: should not be delayed by more than 500ms

3.4.8 Error Control

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 in TCP is achieved through the use of three tools: checksum, acknowledgment, and time-out.

3.4.8 (continued)

- ***Checksum***
- ***Acknowledgment***
  - ***Cumulative Acknowledgment (ACK)***
  - ***Selective Acknowledgment (SACK)***
- ***Generating Acknowledgments***
- ***Retransmission***
  - ***Retransmission after RTO***
  - ***Retransmission after Three Duplicate ACK***
- ***Out-of-Order Segments***

3.4.8 (continued)

- **FSMs for Data Transfer in TCP**
  - Sender-Side FSM
  - Receiver-Side FSM
- **Some Scenarios**
  - Normal Operation
  - Lost Segment
  - Fast Retransmission
  - Delayed Segment
  - Duplicate Segment
  - Automatically Corrected Lost ACK
  - Correction by Resending a Segment
  - Deadlock Created by Lost Acknowledgment

Note

ACK segments do not consume sequence numbers and are not acknowledged.

Acknowledgement Type

- In the past, TCP used only one type of acknowledgement: Cumulative Acknowledgement (ACK), also namely accumulative positive acknowledgement
- More and more implementations are adding another type of acknowledgement: Selective Acknowledgement (SACK), SACK is implemented as an option at the end of the TCP header.

Rules for Generating ACK (1)

1. When one end sends a data segment to the other end, it must include an ACK. That gives the next sequence number it expects to receive. (Piggyback)
2. The receiver needs to delay sending (until another segment arrives or 500ms) an ACK segment if there is only one outstanding in-order segment. It prevents ACK segments from creating extra traffic.
3. There should not be more than 2 in-order unacknowledged segments at any time. It prevent the unnecessary retransmission

Rules for Generating ACK (2)

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. (for fast retransmission)
5. When a missing segment arrives, the receiver sends an ACK segment to announce the next sequence number expected.
6. If a duplicate segment arrives, the receiver immediately sends an ACK.

Retransmission

- Retransmission after RTO(Retransmission Time Out)
- Retransmission after three duplicate ACK Segments.(fast retransmission)

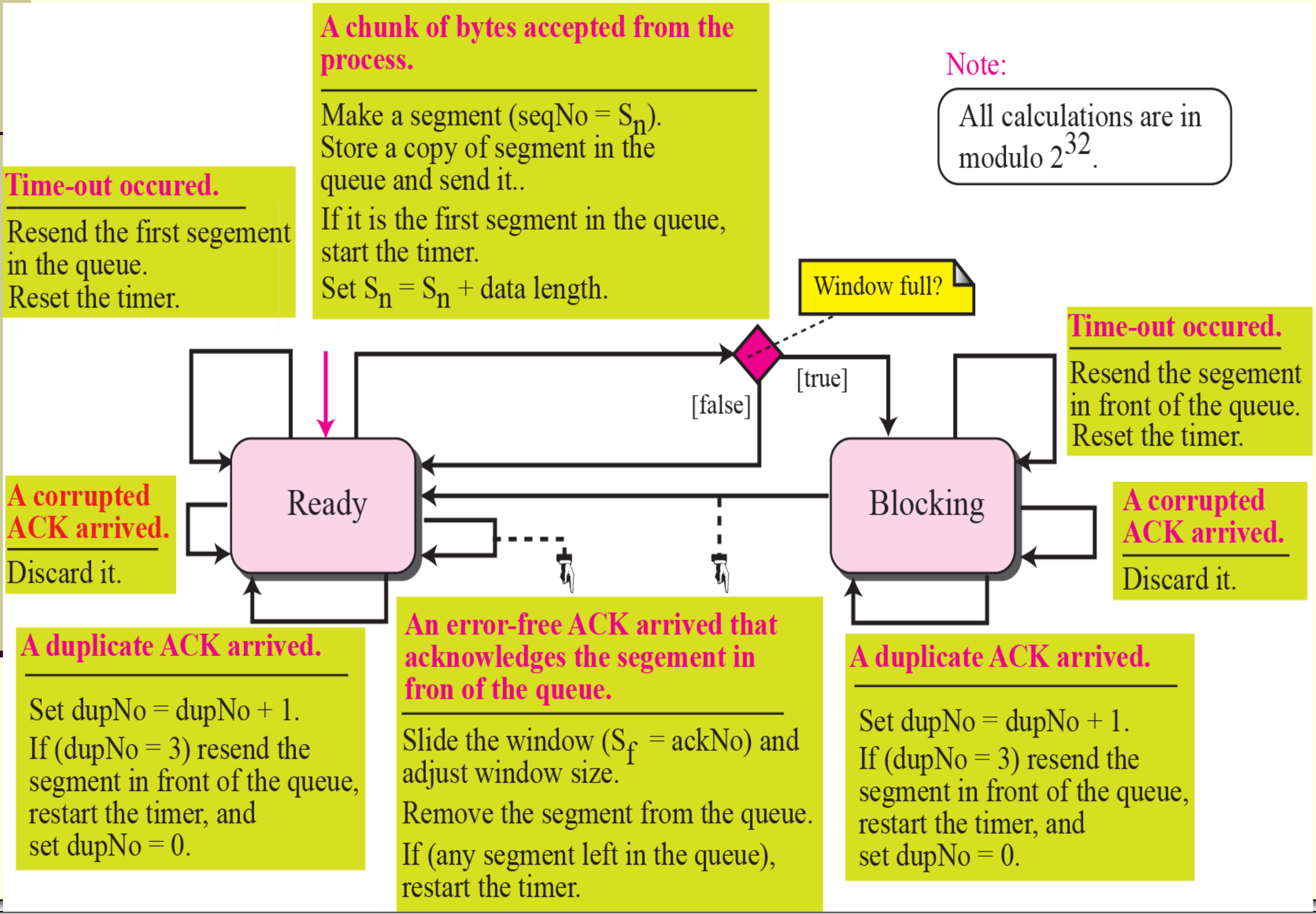
Note

Data may arrive out of order and be temporarily stored by the receiving TCP, but TCP guarantees that no out-of-order data are delivered to the process.

Note

TCP can be best modeled as a Selective Repeat protocol.

Figure 3.59 *Simplified FSM for sender site*



A chunk of bytes accepted from the process.
Make a segment (seqNo = S_n).
Store a copy of 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 occured.
Resend the first segement in the queue.
Reset the timer.
Window full?
[true]
[false]
Time-out occured.
Resend the segement in front of the queue.
Reset the timer.
Ready
Blocking
A corrupted ACK arrived.
Discard it.
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 segement in fron of the queue.
Slide the window (S_f = ackNo) and adjust window size.
Remove the segment from the queue.
If (any segment 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.

Figure 3.60 *Simplified FSM for the receiver site*

Note:

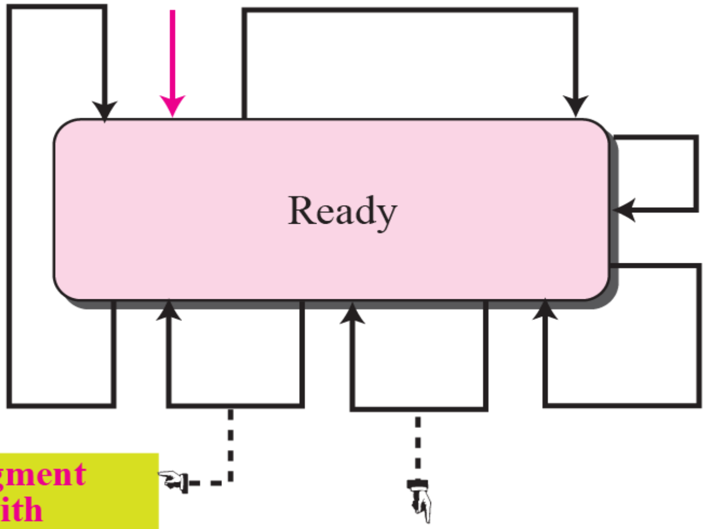
All calculations are in modulo 2^{32}.

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 cummulative ACK. Else, 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 ouside 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.

Difference between FSM and SR (FSM)

- Sender Side
  - Fast transmission
  - Window size adjustment based on rwnd.
- Receiver Side
  - Delayed ACK
  - Sending Duplicate ACK

Figure 3.61: Normal operation

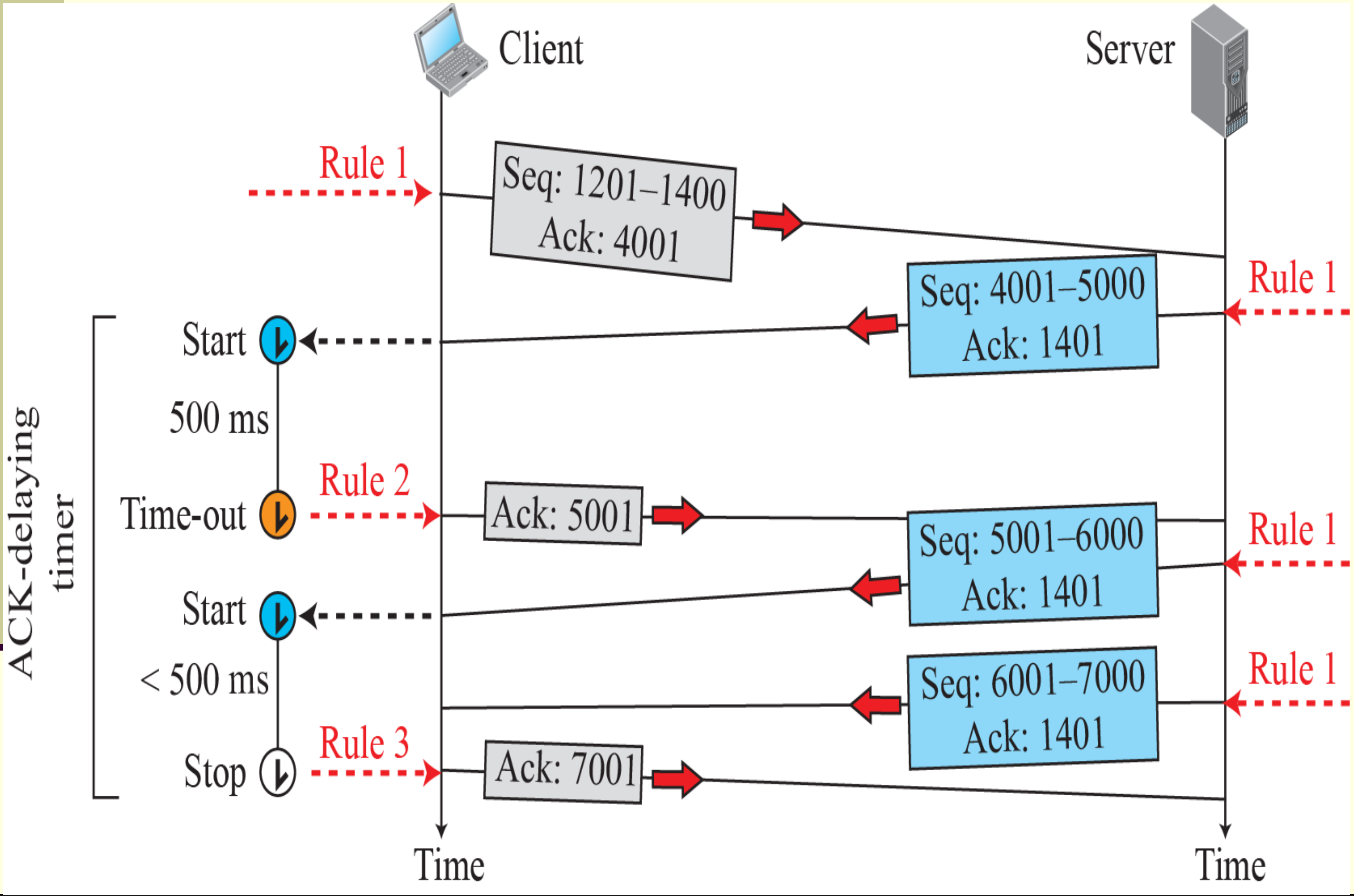
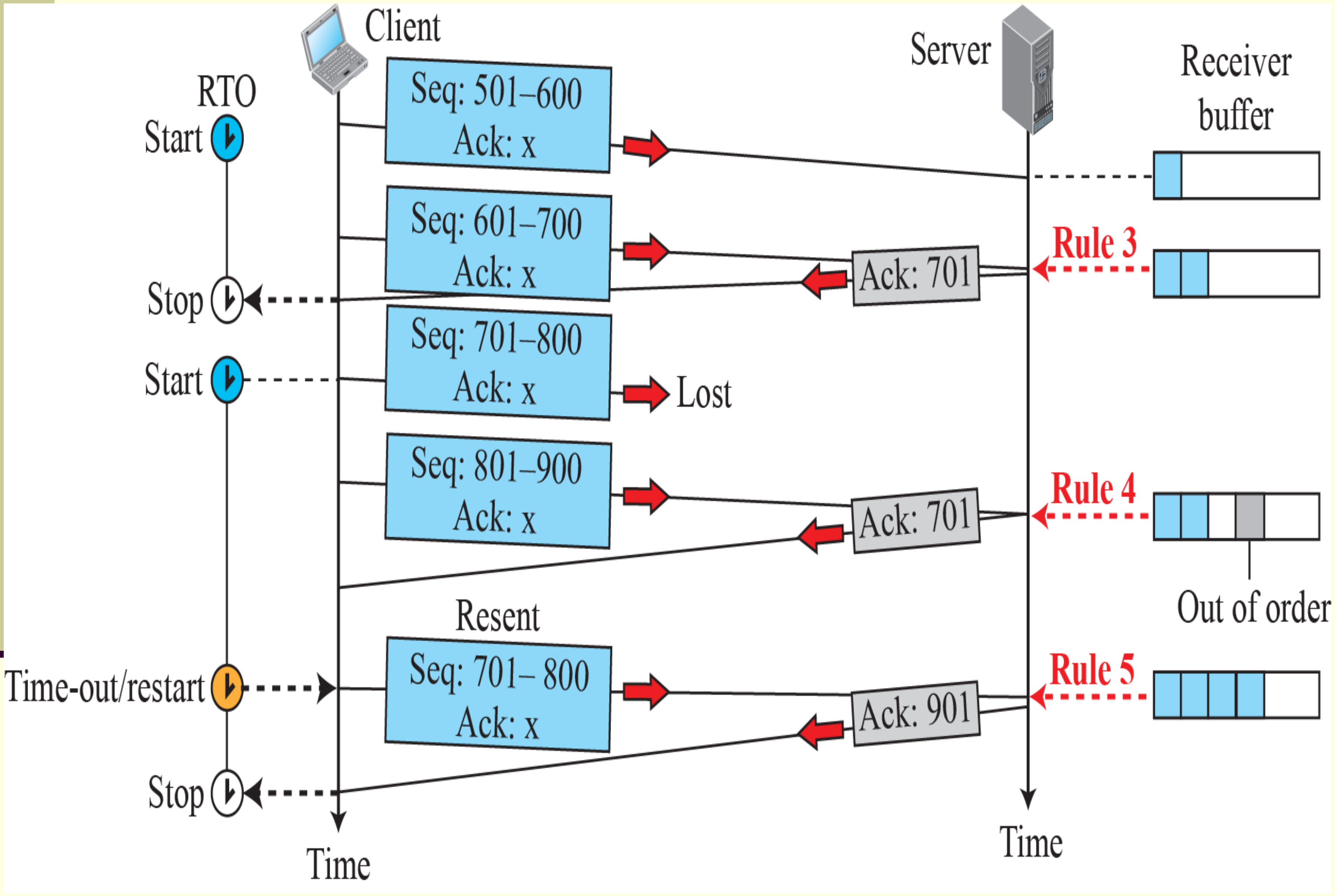


Figure 3.62: Lost segment



Note

The receiver TCP delivers only ordered data to the process.

Figure 3.63: Fast retransmission

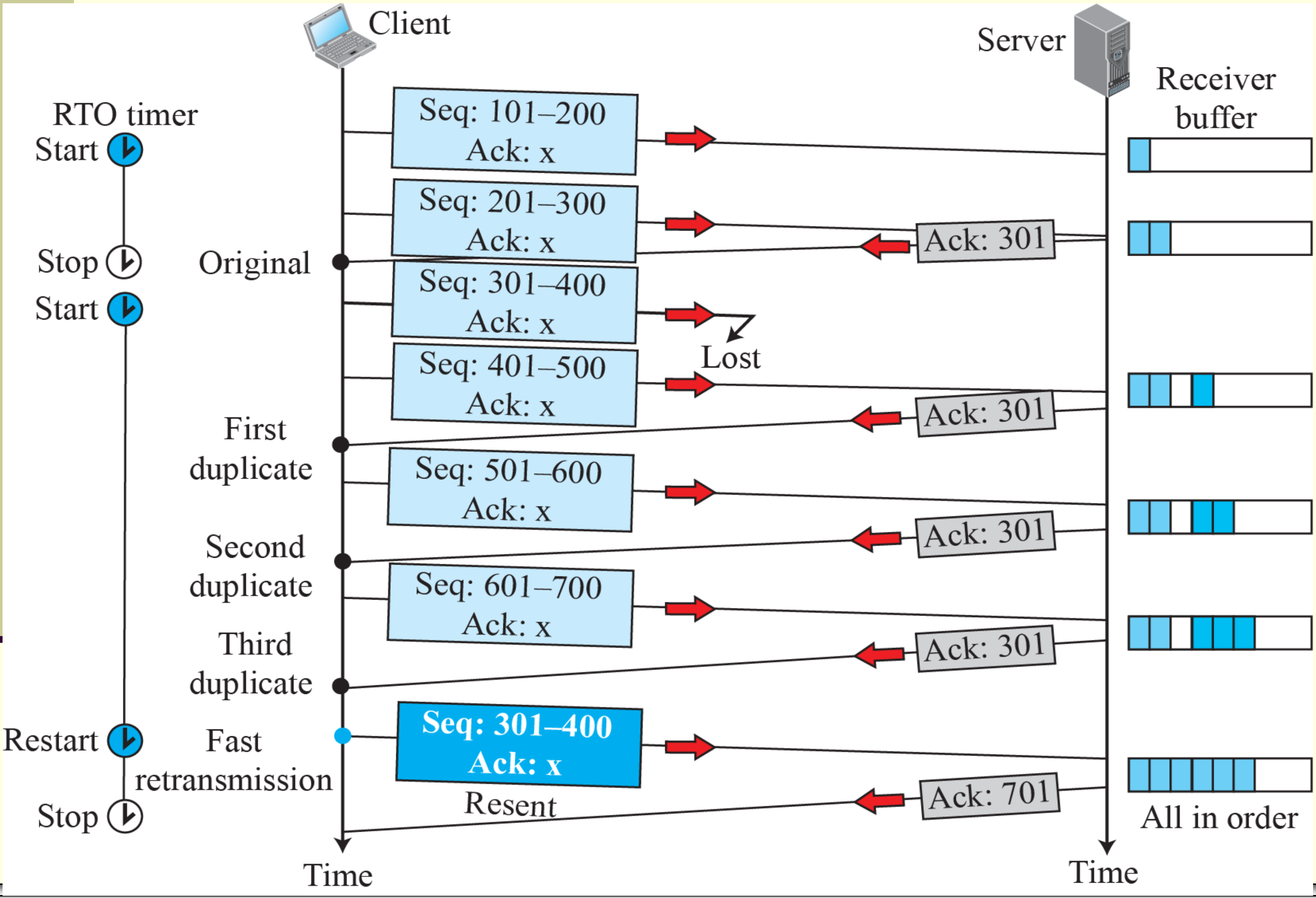


Figure 3.64: Lost acknowledgment

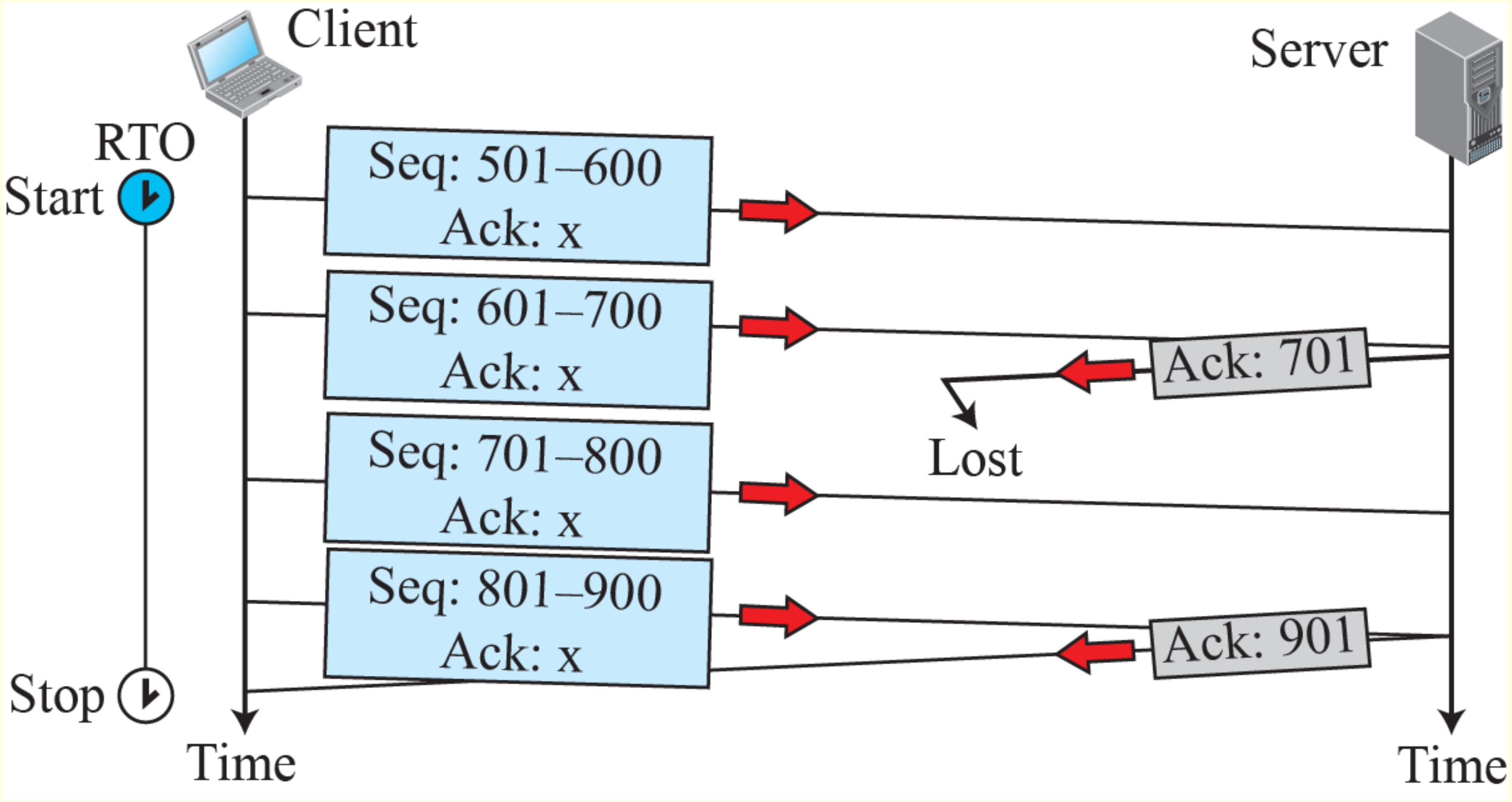
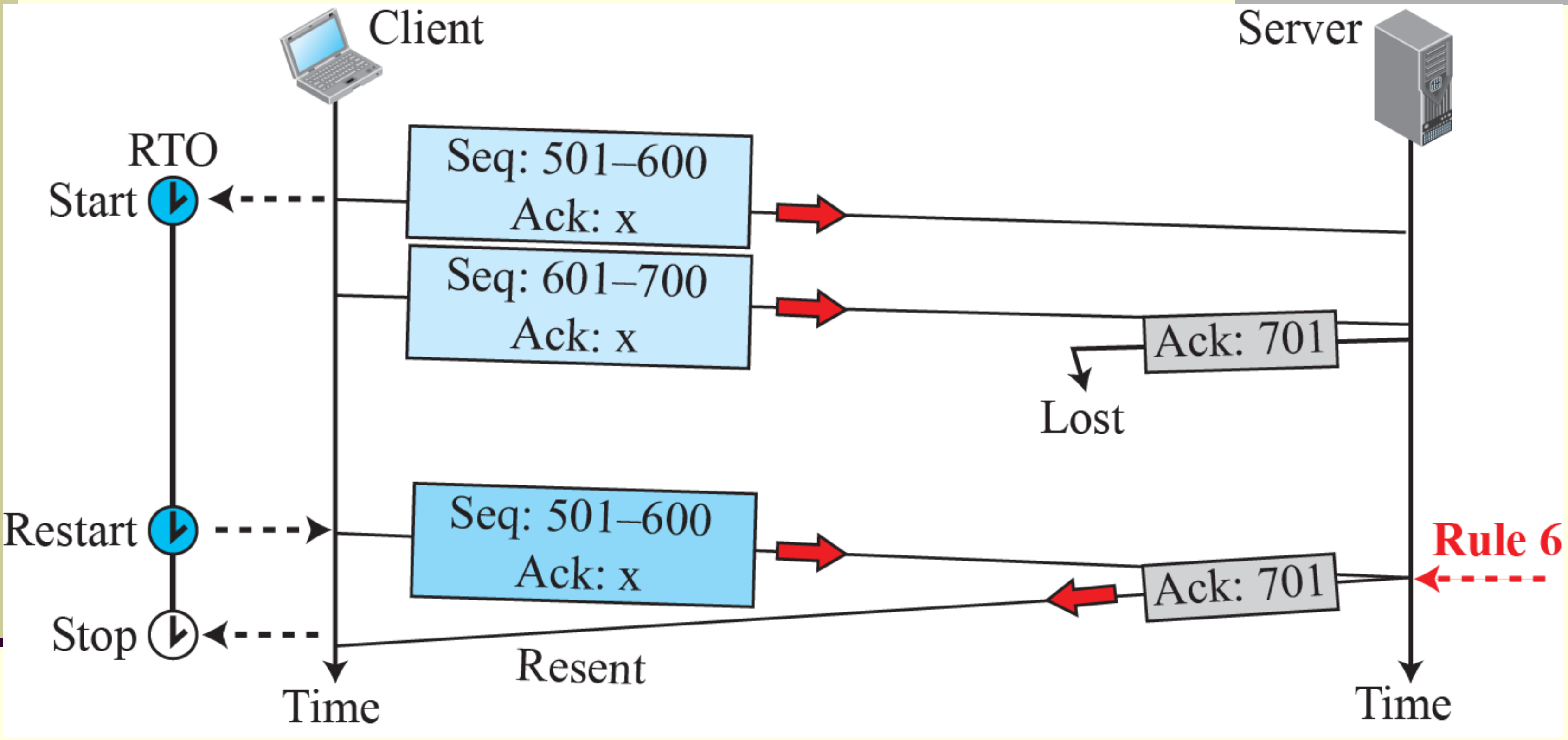


Figure 3.65: Lost acknowledgment corrected by resending a segment



Note

Lost acknowledgments may create deadlock if they are not properly handled.

3.4.9 TCP Congestion Control

TCP uses different policies to handle the congestion in the network. We describe these policies in this section.

- ***Congestion Window***
- ***Congestion Detection***
- ***Congestion Policies***
  - ***Slow Start: Exponential Increase***
  - ***Congestion Avoidance: Additive Increase***

3.4.9 (continued)

- ❑ *Policy Transition*
  - ❖ *Taho TCP*
  - ❖ *Reno TCP*
  - ❖ *NewReno TCP*
- ❑ *Additive Increase, Multiplicative Decrease*
- ❑ *TCP Throughput*

Congestion Window

CongestionWindow (cwnd) is a variable held by the TCP source for each connection.

MaxWindow :: min (**CongestionWindow** , AdvertisedWindow)

EffectiveWindow = MaxWindow – (LastByteSent -LastByteAcked)

cwnd is set based on the perceived level of congestion. The Host receives *implicit* (packet drop) or *explicit* (packet mark) indications of internal congestion.

Congestion Detection

- Time Out
- Three Duplicate ACKs

Congestion Policies

- Slow start
- Congestion Avoidance
- Fast Recovery

Slow Start

- Linear additive increase takes <u>too long</u> to ramp up a new TCP connection from cold start.
- Beginning with TCP Taho, the slow start mechanism was added to provide an initial exponential increase in the size of cwnd.

- *Remember mechanism by:* ***slow start <u>prevents</u> a slow start. Moreover, slow start is slower than sending a full advertised window's worth of packets all at once.***

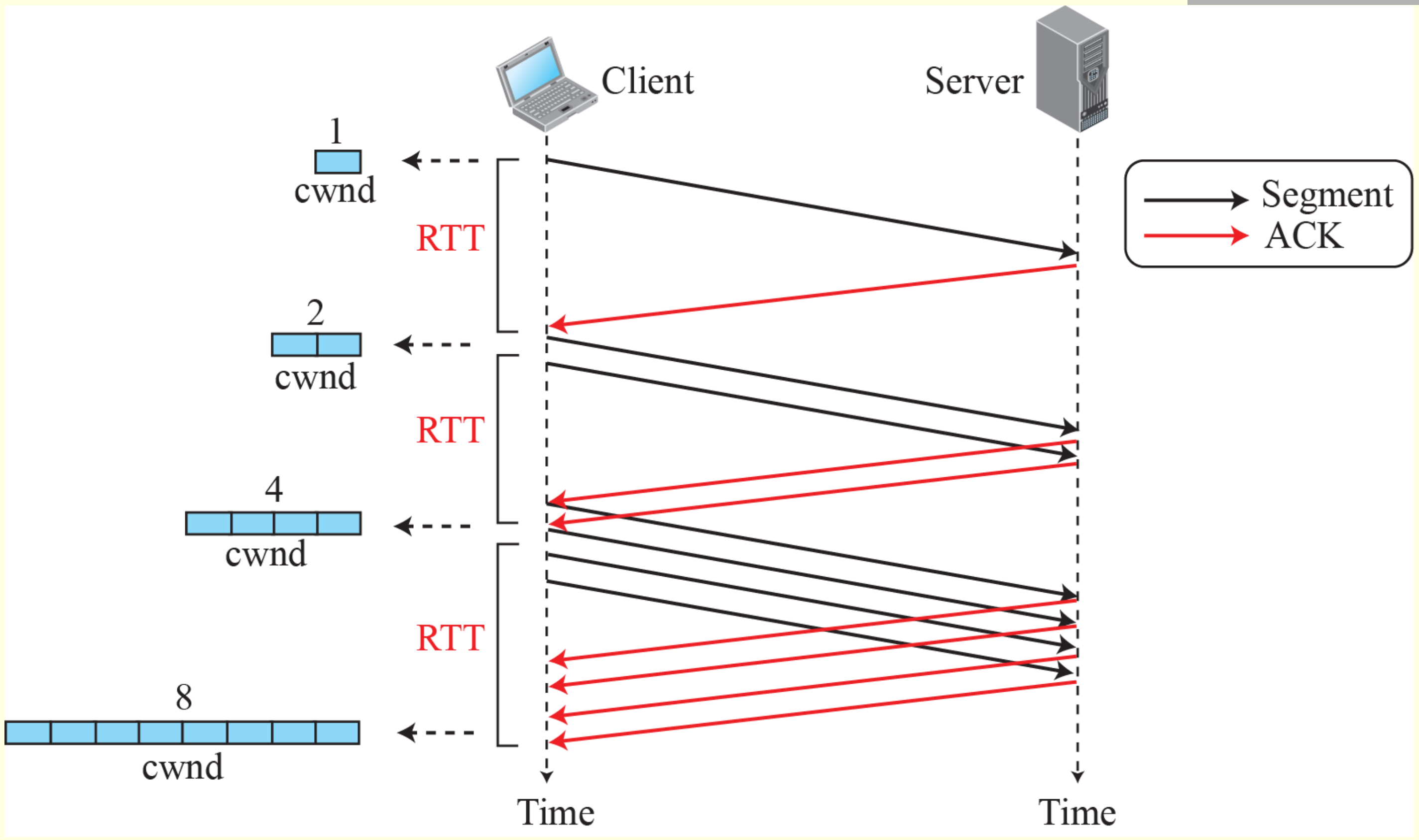
Slow Start:Exponential Increase

- The source starts with cwnd = 1.
- Every time an ACK arrives, cwnd is incremented.
- cwnd is effectively doubled per RTT “epoch”.
- Two slow start situations:
  - At the very beginning of a connection **{cold start**}.
  - When the connection goes dead waiting for a timeout to occur (i.e., the advertised window goes to zero!)

- If an ACK arrives ,
  - Cwnd=cwnd+1

- Start cwnd=1
- After 1 RTT cwnd= cwnd+1=2
- After 2 RTT cwnd= cwnd+2=4
- After 3 RTT cwnd= cwnd+3=8

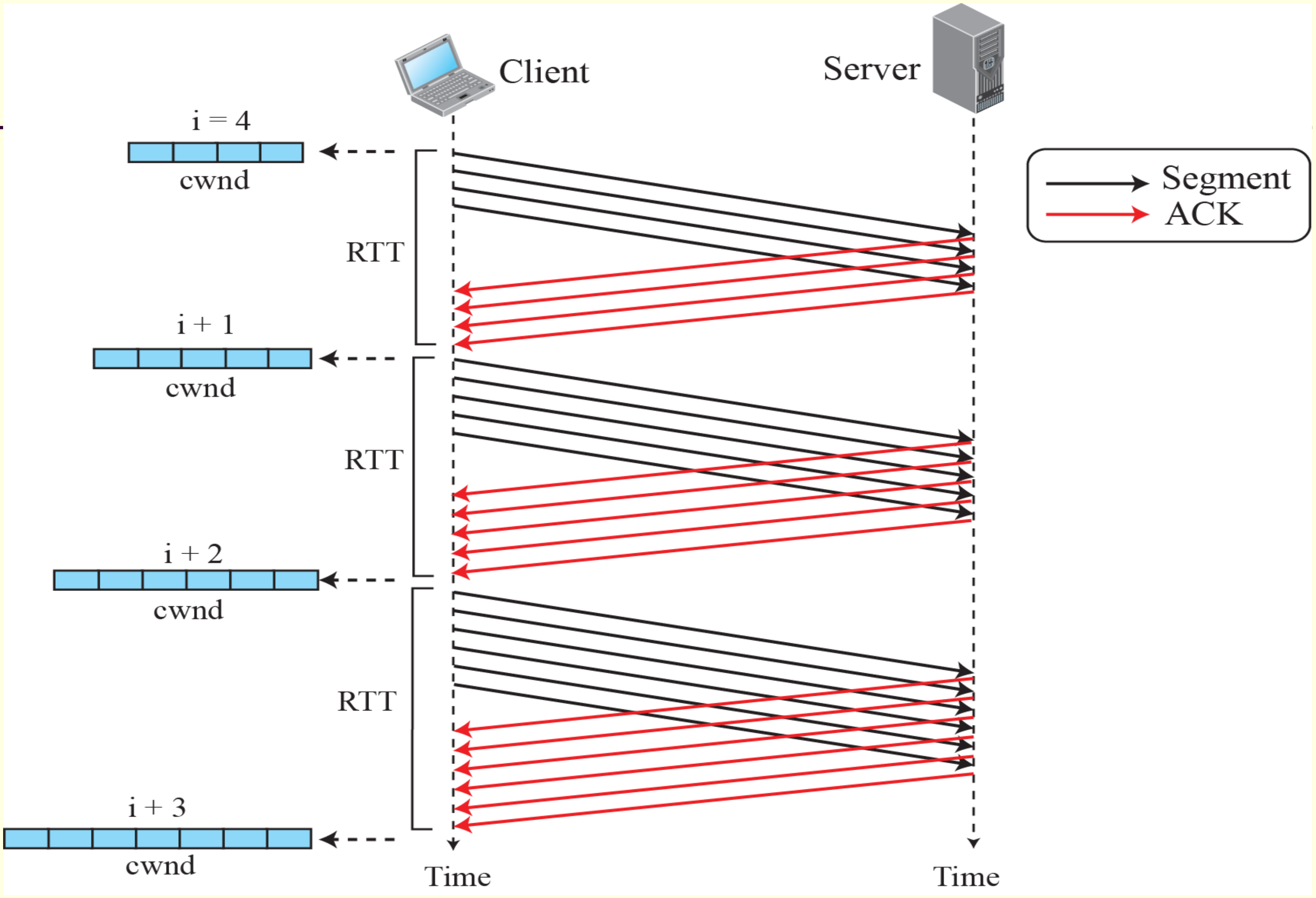
Figure 3.66: Slow start, exponential increase



Note

In the slow start algorithm, the size of the congestion window increases exponentially until it reaches a threshold.

Figure 3.67: Congestion avoidance, additive increase



Additive Increase

- If an ACK arrives ,
  - Cwnd=cwnd+(1/cwnd)
- Start cwnd=i
- After 1 RTT cwnd= i+1
- After 2 RTT cwnd= i+2
- After 3 RTT cwnd= i+3

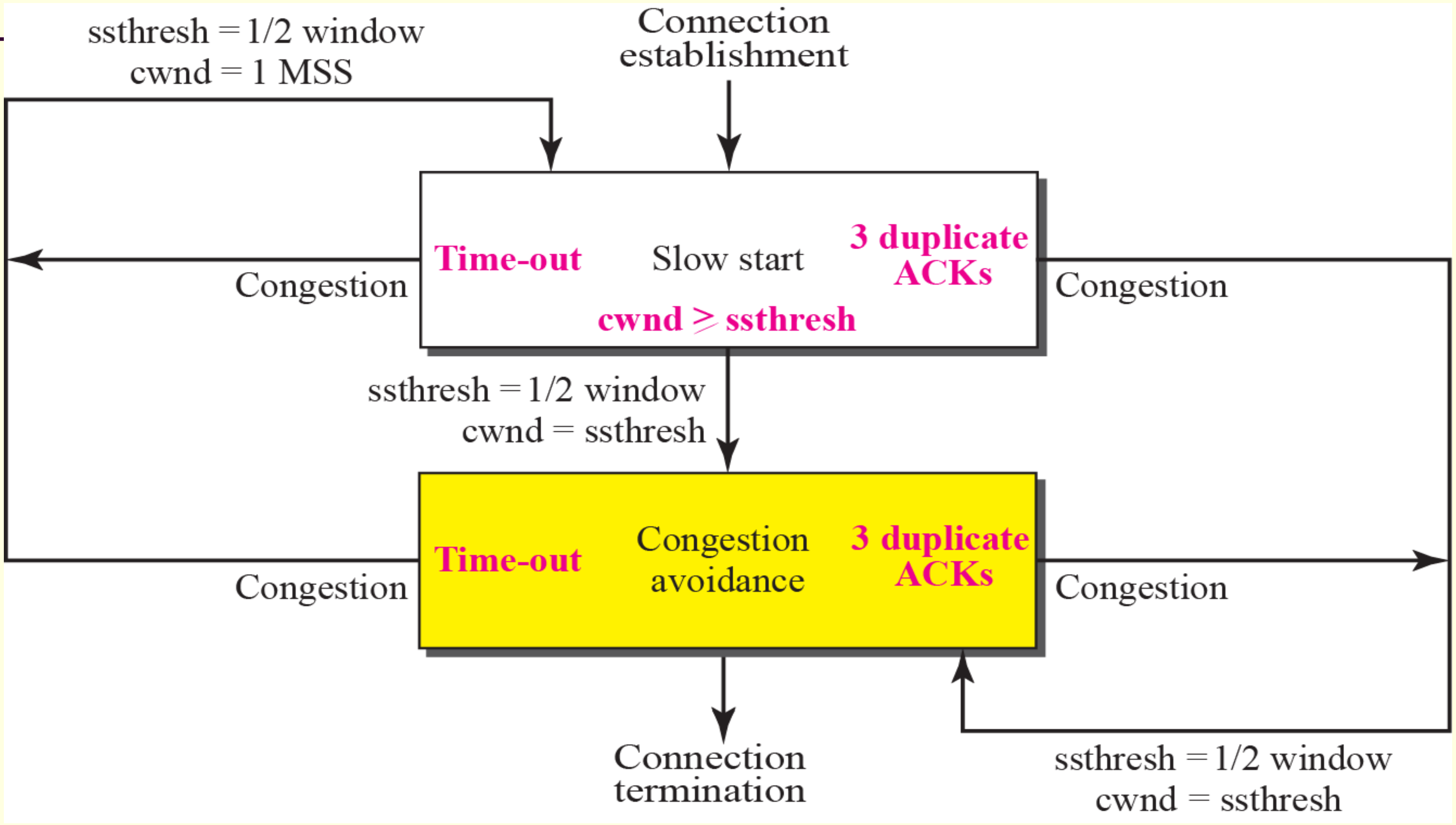
Note

In the congestion avoidance algorithm the size of the congestion window increases additively until congestion is detected.

Fast Recovery

- Optional,Older versions did not use it.
- It starts when 3 duplicate ACKs arrive
- --->light congestion.
- It is like additive increase ,but it increases cwnd when a duplicate ACK arrives
- If a duplicate ACK arrives
  - Cwnd= cwnd +(1/cwnd)

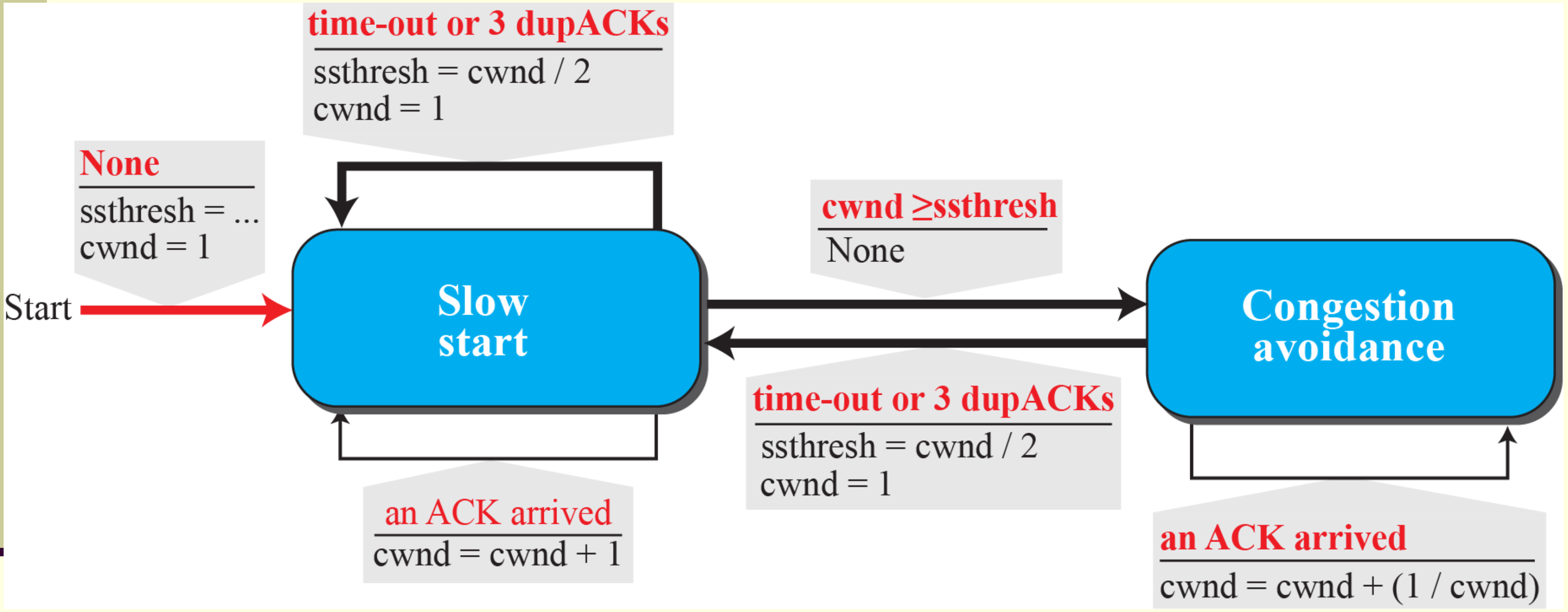
TCP Congestion policy summary



Three versions

- Taho TCP
- Reno TCP
- New Reno TCP

Figure 3.68: FSM for Taho TCP



Example 3.19

Figure 3.69 shows an example of congestion control in a Taho TCP. TCP starts data transfer and sets the ssthresh variable to an ambitious value of 16 MSS. TCP begins at the slow-start (SS) state with the cwnd = 1. The congestion window grows exponentially, but a time-out occurs after the third RTT (before reaching the threshold). TCP assumes that there is congestion in the network. It immediately sets the new ssthresh = 4 MSS (half of the current cwnd, which is 8) and begins a new slow start (SA) state with cwnd = 1 MSS. The congestion grows exponentially until it reaches the newly set threshold. TCP now moves to the congestion avoidance (CA) state and the congestion window grows additively until it reaches cwnd = 12 MSS.

Example 3.19 (continued)

At this moment, three duplicate ACKs arrive, another indication of the congestion in the network. TCP again halves the value of ssthresh to 6 MSS and begins a new slow-start (SS) state. The exponential growth of the cwnd continues. After RTT 15, the size of cwnd is 4 MSS. After sending four segments and receiving only two ACKs, the size of the window reaches the ssthresh (6) and the TCP moves to the congestion avoidance state. The data transfer now continues in the congestion avoidance (CA) state until the connection is terminated after RTT 20.

Figure 3.69: Example of Taho TCP

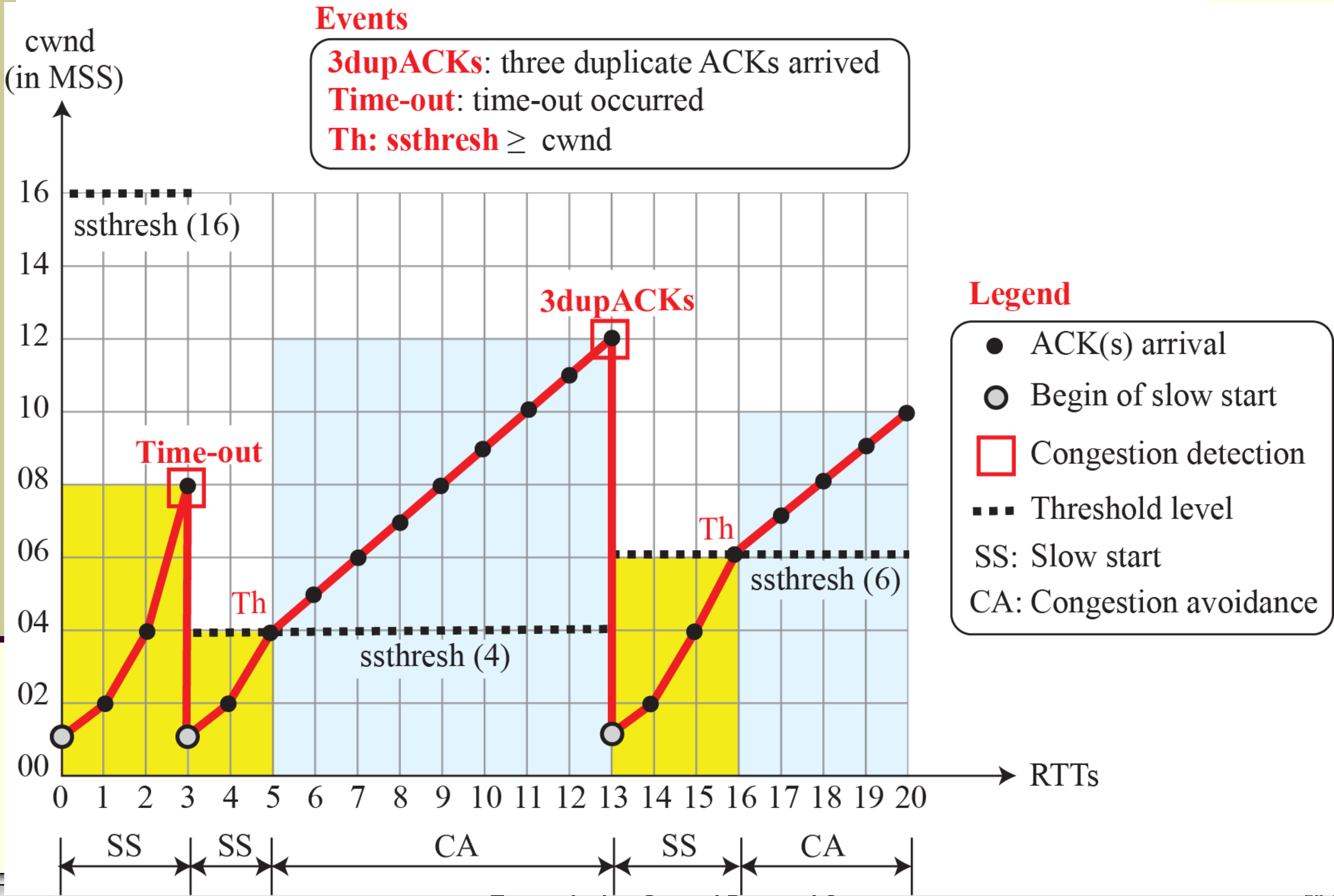
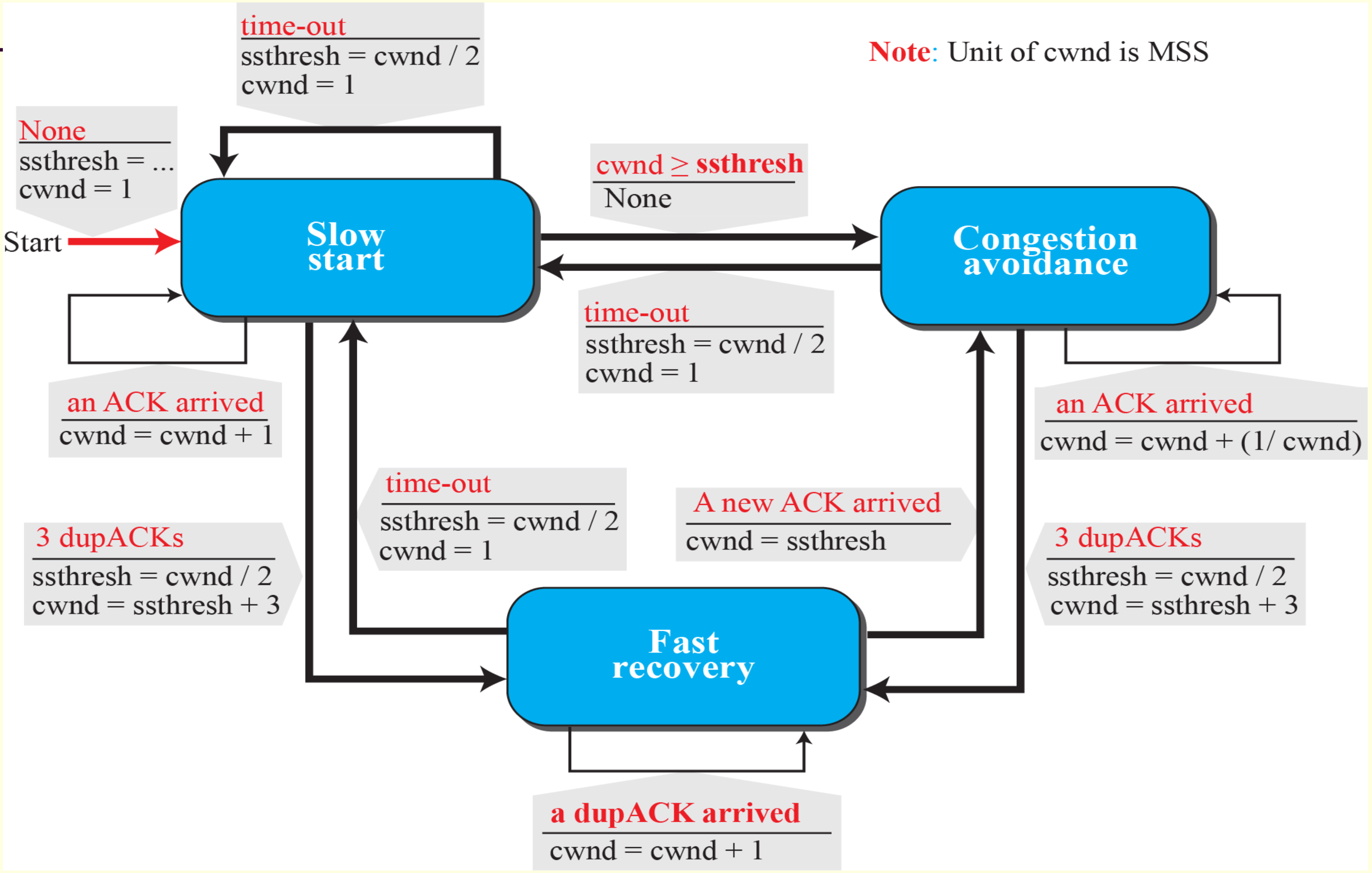


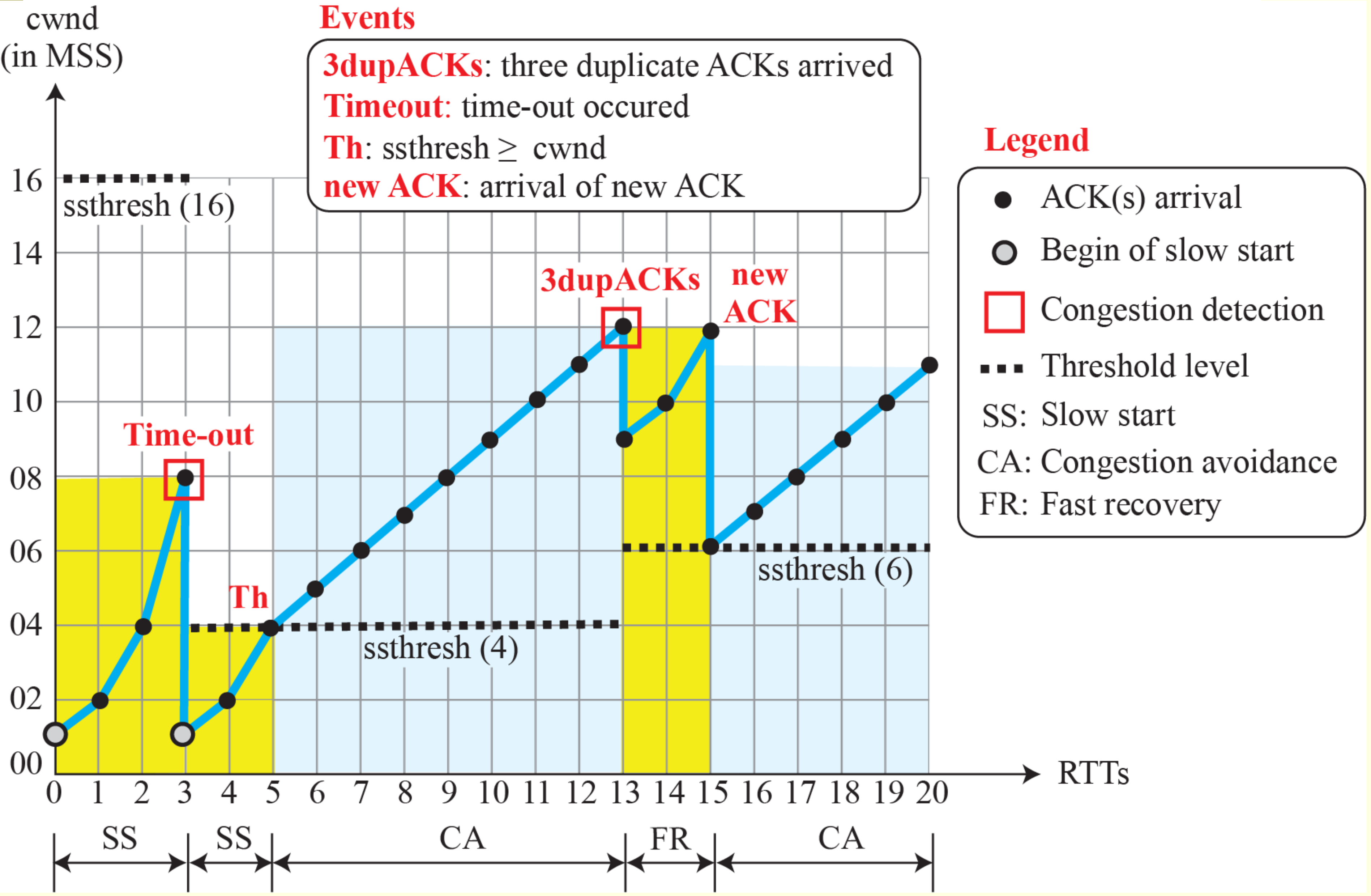
Figure 3.70: FSM for Reno TCP



Example 3.20

Figure 3.71 shows the same situation as Figure 3.69, but in Reno TCP. The changes in the congestion window are the same until RTT 13 when three duplicate ACKs arrive. At this moment, Reno TCP drops the ssthresh to 6 MSS, but it sets the cwnd to a much higher value (ssthresh + 3 = 9 MSS) instead of 1 MSS. It now moves to the fast recovery state. We assume that two more duplicate ACKs arrive until RTT 15, where cwndgrows exponentially. In this moment, a new ACK (not duplicate) arrives that announces the receipt of the lost segment. It now moves to the congestion avoidance state, but first deflates the congestion window to 6 MSS as though ignoring the whole fast-recovery state and moving back to the previous track.

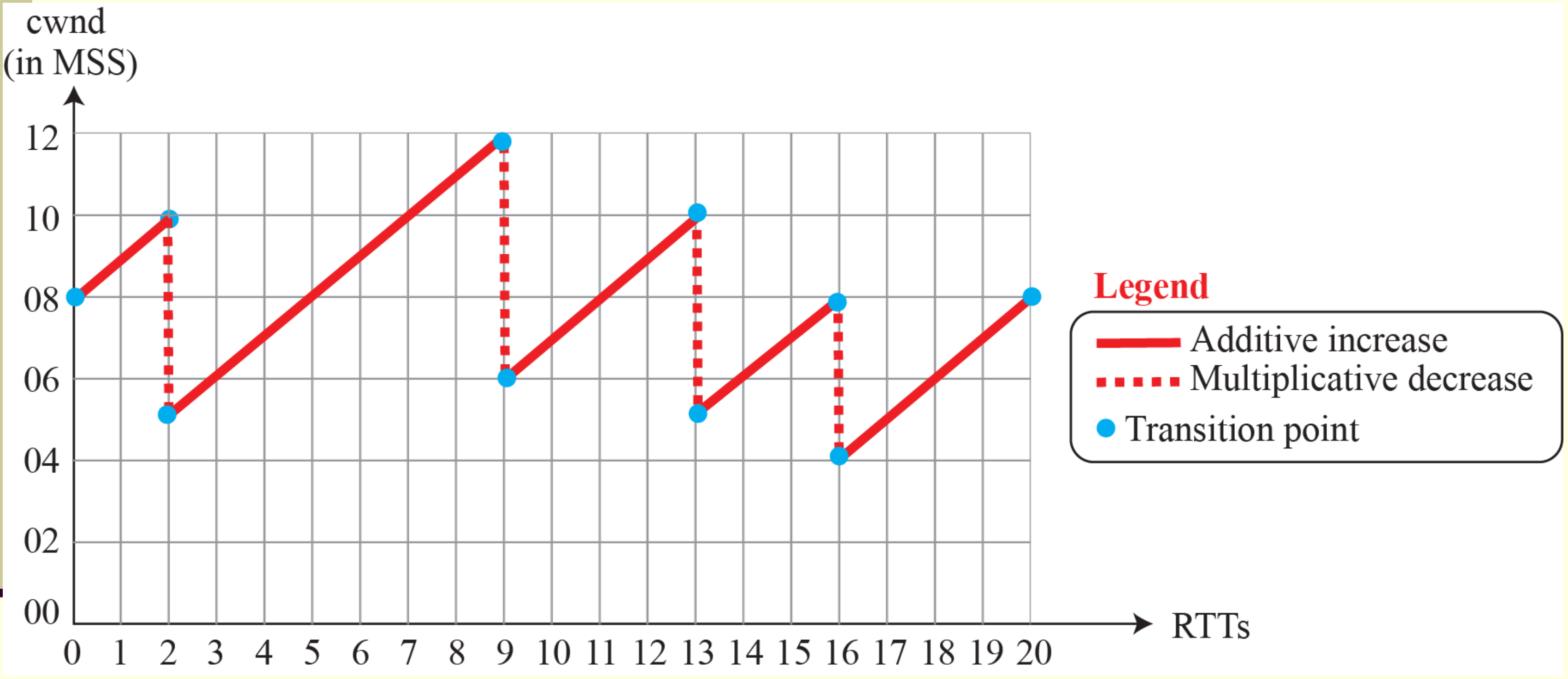
Figure 3.71: Example of a Reno TCP



New Reno TCP

- Extra optimization on Reno
- It checks if more than one segment is lost in current window when 3 duplicate ACKs arrive
- It retransmits the lost segment until a new ACK arrives.
- New ACK defines
  - End of window ....congestion ..only one segment lost
  - If position in between retransmitted segment and end of window-- retransmit that segment.

Figure 3.72: Additive increase, multiplicative decrease (AIMD)

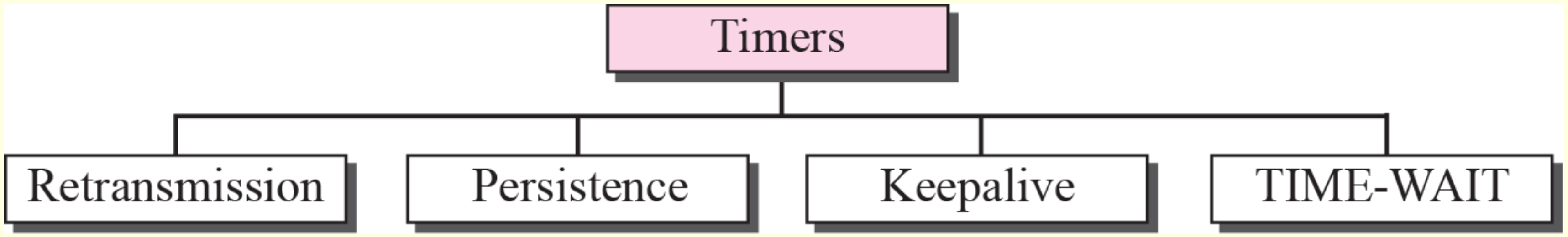


3.4.10 TCP Timers

To perform their operations smoothly, most TCP implementations use at least four timers.

- ***Retransmission Timer***
  - ***Round-Trip Time (RTT)***
  - ***Karn's Algorithm***
  - ***Exponential Backoff***
- ***Persistence Timer***
- ***Keepalive Timer***
- ***TIME-WAIT Timer***

TCP timers



Retransmission Timer(Rules)

- When TCP send the segment in the front of the sending queue, it starts the timer.
- When the timer expires, TCP resends the first segment in front of the queue and restarts the timer.
- When a segment or segments are cumulatively acknowledged ,the segment or segments are purged form the queue.
- If the queue is empty, TCP stops the timer, otherwise TCP restarts the timer.

Measured RTT

- How long it takes to send a segment and receive an acknowledgement for it.—RTT(M)
- Even though acknowledgment may include other segments too.

Note

In TCP, there can be only one RTT measurement in progress at any time.

Since the segments and their ACKs do not have a 1-1 relationship

Calculation of RTO (1)

- RTT(m) is likely to change for each RTT.So we can't rely on single RTT.RTT(s) is a weighted avergage of RTT(m) and Previous RTT(s)
- Smoothed RTT: RTT_S
  - Original → No value
  - After 1st measurement → $RTT_S = RTT_M$
  - 2nd … → $RTT_S = (1-\alpha)*RTT_S + \alpha*RTT_M$
  - α is set to 1/8(implementation dependent)
- RTT Deviation : RTT_D
  - Original → No value
  - After 1st measurement → $RTT_D = 0.5*RTT_M$
  - 2nd … → $RTT_D = (1-\beta)*RTT_D + \beta*|RTT_S - RTT_M|$
  - β is set to 1/4(implementation dependent)

Calculation of RTO (2)

- Retransmission Timeout (RTO)
  - Original= Initial value
  - After any measurement
  - → $RTO = RTT_S + 4RTT_D$

Example 3.22

Let us give a hypothetical example. Figure 3.73 shows part of a connection. The figure shows the connection establishment and part of the data transfer phases.

1. ***When the SYN segment is sent, there is no value for RTTM, RTTS, or RTTD. The value of RTO is set to 6.00 seconds. The following shows the value of these variables at this moment:***

$$\boxed{\text{RTO} = 6}$$

Example 3.22 (continued)

2. When the SYN+ACK segment arrives, RTTM is measured and is equal to 1.5 seconds. The following shows the values of these variables:

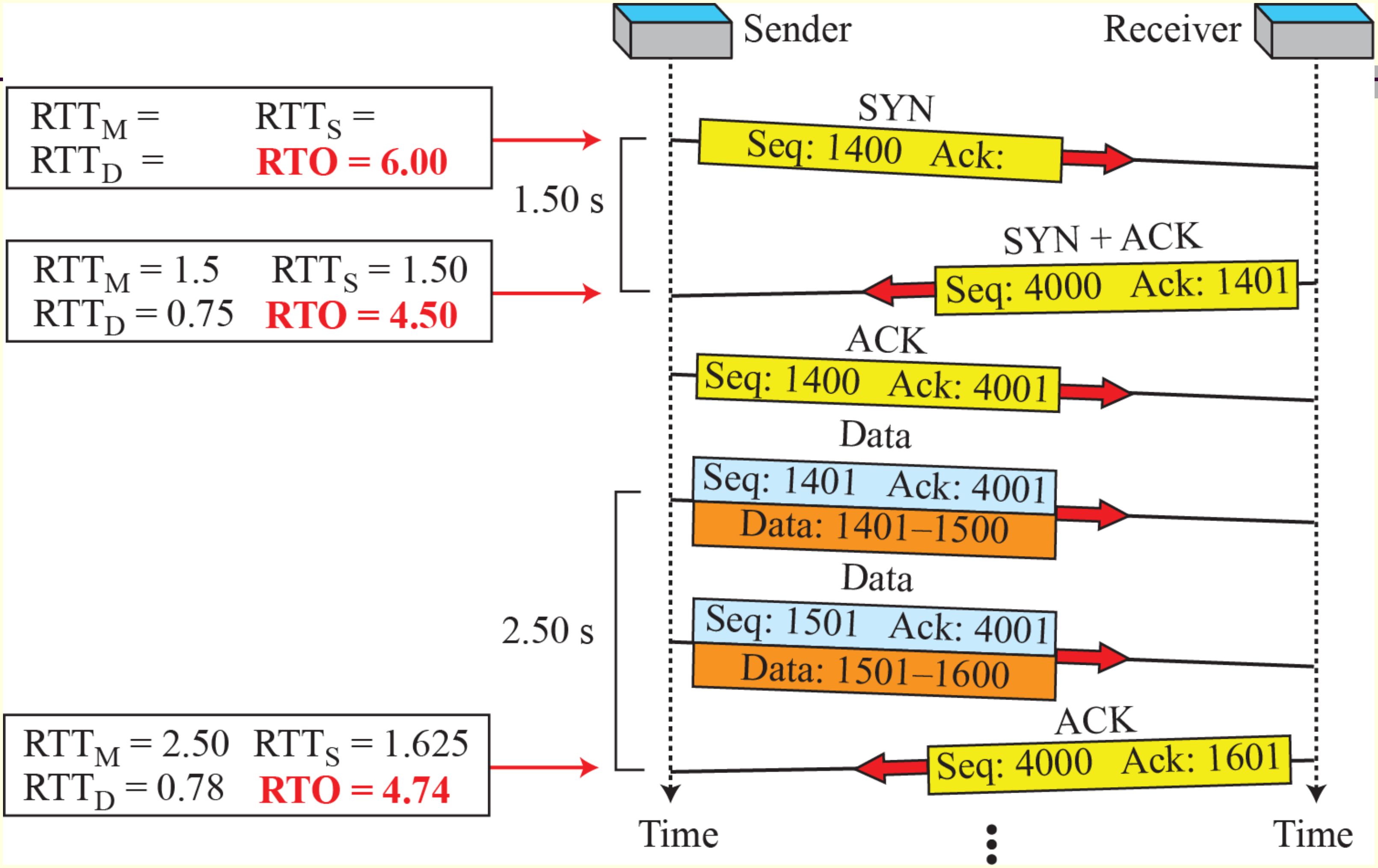
$$\boxed{\begin{aligned}
\mathrm{RTT_M} &= 1.5 \\
\mathrm{RTT_S} &= 1.5 \\
\mathrm{RTT_D} &= (1.5)/2 = 0.75 \\
\mathrm{RTO} &= 1.5 + 4 \times 0.75 = 4.5
\end{aligned}}$$

Example 3.22 (continued)

3. When the first data segment is sent, a new RTT measurement starts. Note that the sender does not start an RTT measurement when it sends the ACK segment, because it does not consume a sequence number and there is no time-out. No RTT measurement starts for the second data segment because a measurement is already in progress.

$$\begin{aligned}
RTT_M &= 2.5\\
RTT_S &= (7/8) \times (1.5) + (1/8) \times (2.5) = 1.625\\
RTT_D &= (3/4) \times (0.75) + (1/4) \times |1.625 - 2.5| = 0.78\\
RTO &= 1.625 + 4 \times (0.78) = 4.74
\end{aligned}$$

Figure 3.73: Example 3.22



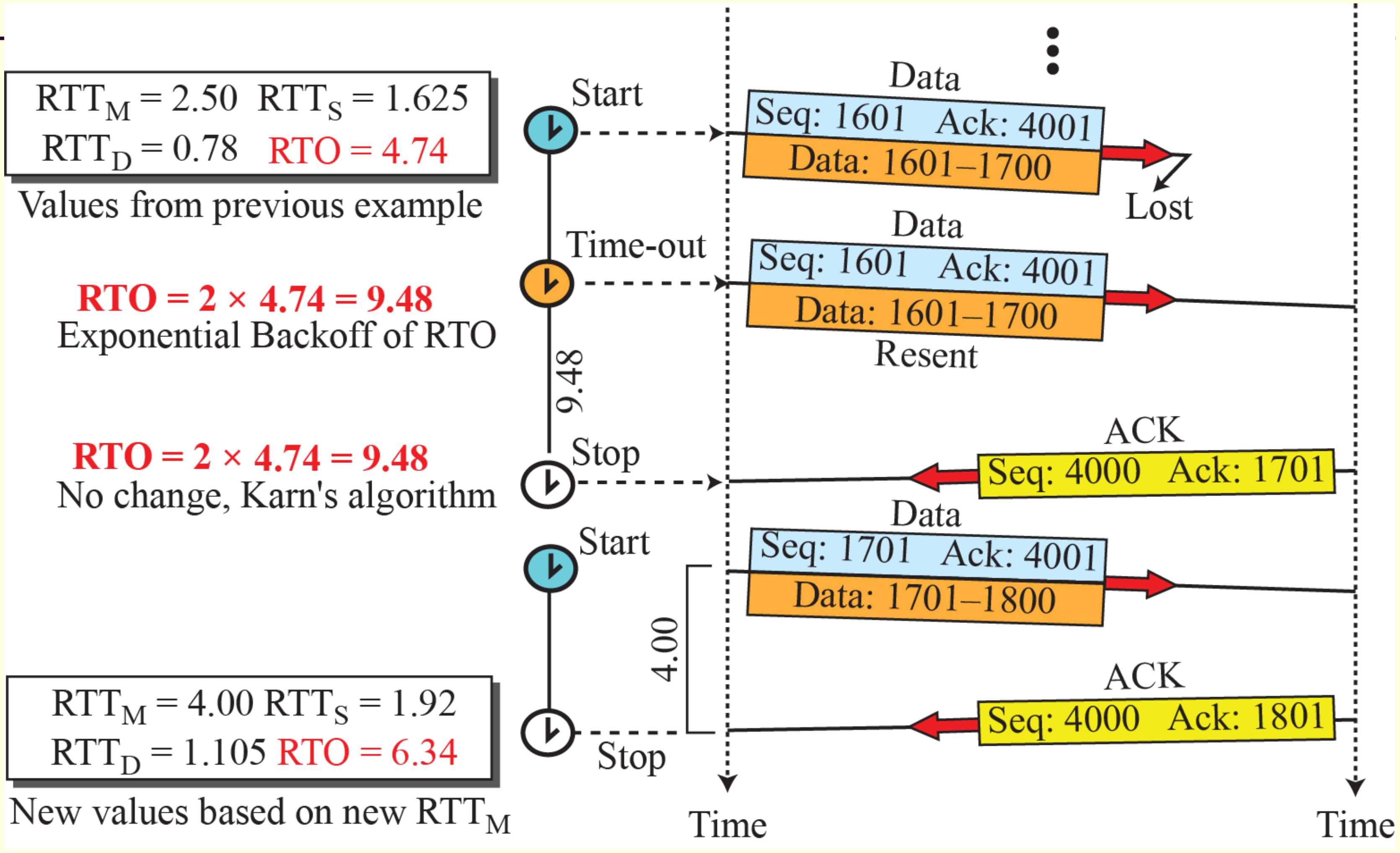
Note

TCP does not consider the RTT of a retransmitted segment in its calculation of a new RTO.

Example 3.23

Figure 3.74 is a continuation of the previous example. There is retransmission and Karn's algorithm is applied. The first segment in the figure is sent, but lost. The RTO timer expires after 4.74 seconds. The segment is retransmitted and the timer is set to 9.48, twice the previous value of RTO. This time an ACK is received before the time-out. We wait until we send a new segment and receive the ACK for it before recalculating the RTO (Karn's algorithm).

Figure 3.74: Example 3.23



Persistence Timer

- To deal with a zero-window-size deadlock situation, TCP uses a persistence timer.
- If the receiving TCP announces a window size of zero ,the sending TCP stops transmitting segments until the receiving TCP sends an ACK segment announcing a non zero window size.
- This ACK segment can be lost.
- ACK segments are not acknowledged nor retransmitted in TCP.
- If this acknowledgement is lost ,the receiving TCP thinks that it has done its job and waits for sending TCP to send more segments.
- Whereas sending TCP is still waiting for non zero window announcement.
- This creates a deadlock.

Persistence Timer

- To deal with a zero-window-size deadlock situation, TCP uses a persistence timer.
- When the sending TCP receives an acknowledgment with a window size of zero, it starts a persistence timer.
- When the persistence timer goes off, the sending TCP sends a special segment called a probe. This segment contains only 1 byte of new data.
- It has a sequence number, but its sequence number is never acknowledged; it is even ignored in calculating the sequence number for the rest of the data.
- The probe causes the receiving TCP to resend the acknowledgment which was lost.

Persistence Timer

- Value of persistence timer is set to value of RTT.
- However if the response is not received form the receiver ,another probe segment is sent and value of persistence timer is doubled and reset.
- The process continues till persistence timer value becomes 60 sec(threshold).
- After that sender sends one probe every 60 seconds until the window is reopened.

KeepAlive Timer

- A keepalive timer is used to prevent a long idle connection between two TCPs.
- If a client opens a TCP connection to a server transfers some data and becomes silent the client will crash.
- In this case, the connection remains open forever. So a keepalive timer is used.
- Each time the server hears from a client, it resets this timer.
- The time-out is usually 2 hours. If the server does not hear from the client after 2 hours, it sends a probe segment. If there is no response after 10 probes, each of which is 75 s apart, it assumes that the client is down and terminates the connection.

TIME-WAIT Timer or Quiet Timer

- This timer is used during tcp connection termination.
- The timer starts after sending the last ACK for 2nd FIN and closing the connection.
- *After a TCP connection is closed, it is possible for datagrams that are still making their way through the network to attempt to access the closed port.*
- *The quiet timer is intended to prevent the just-closed port from reopening again quickly and receiving these last datagrams.*
- The **quiet timer** is usually set to twice the maximum segment lifetime (the same value as the Time-To-Live field in an IP header), ensuring that all segments still heading for the port have been discarded.