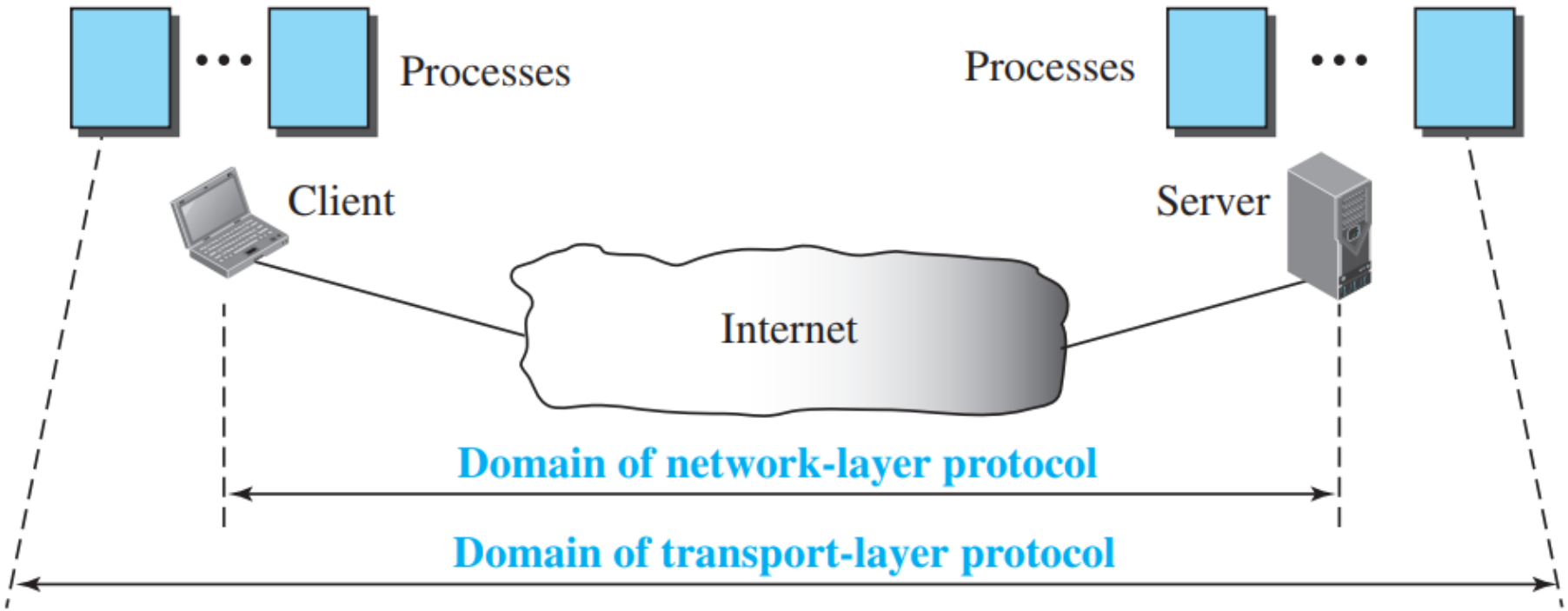
UNIT IV

TRANSPORT LAYER

Services

- Duty of a TL protocol is to provide process-to-process communication.
- A process is an application-layer entity (running program) that uses the services of the transport layer
- difference between host-to-host communication and process-to-process communication
- The network layer is responsible for communication at the computer level (host-to-host communication). A NL protocol can deliver the message only to the destination computer (incomplete delivery)
- A TL Protocol is responsible for delivery of the message to the appropriate process.

Figure 23.2 *Network layer versus transport layer*



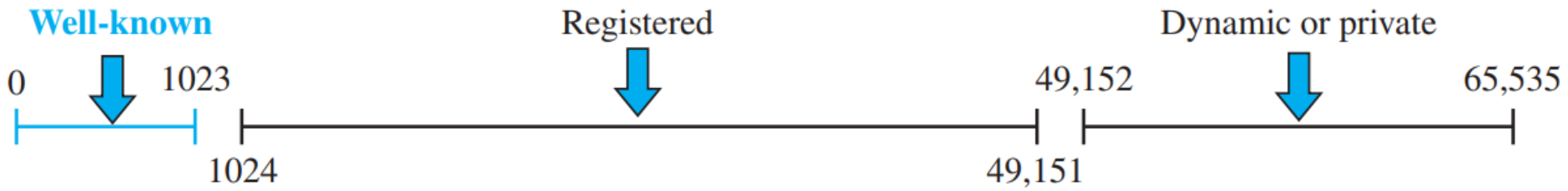
Addressing: Port Numbers

- to achieve process-to-process communication, the most common is through the client-server paradigm
- A process on the local host, called a client, needs services from a process usually on the remote host, called a server
- operating systems today support both multiuser and multiprogramming environments
- For communication, we must define the local host, local process, remote host, and remote process. The local host and the remote host are defined using IP addresses
- To define the processes, we need second identifiers, called port numbers. In the TCP/IP protocol suite, the port numbers are integers between 0 and 65,535 (16 bits)

- The client program defines itself with a port number, called the ephemeral port number. The word ephemeral means “short-lived”
- An ephemeral port number is recommended to be greater than 1023
- The server process must also define itself with a port number. This port number, cannot be chosen randomly as client may not be able to know this
- TCP/IP has decided to use universal port numbers for servers; these are called well-known port numbers.

ICANN Ranges

Figure 23.5 *ICANN ranges*



- ***Well-known ports.*** The ports ranging from 0 to 1023 are assigned and controlled by ICANN. These are the well-known ports.
- ***Registered ports.*** The ports ranging from 1024 to 49,151 are not assigned or controlled by ICANN. They can only be registered with ICANN to prevent duplication.
- ***Dynamic ports.*** The ports ranging from 49,152 to 65,535 are neither controlled nor registered. They can be used as temporary or private port numbers.

Figure 23.3 *Port numbers*

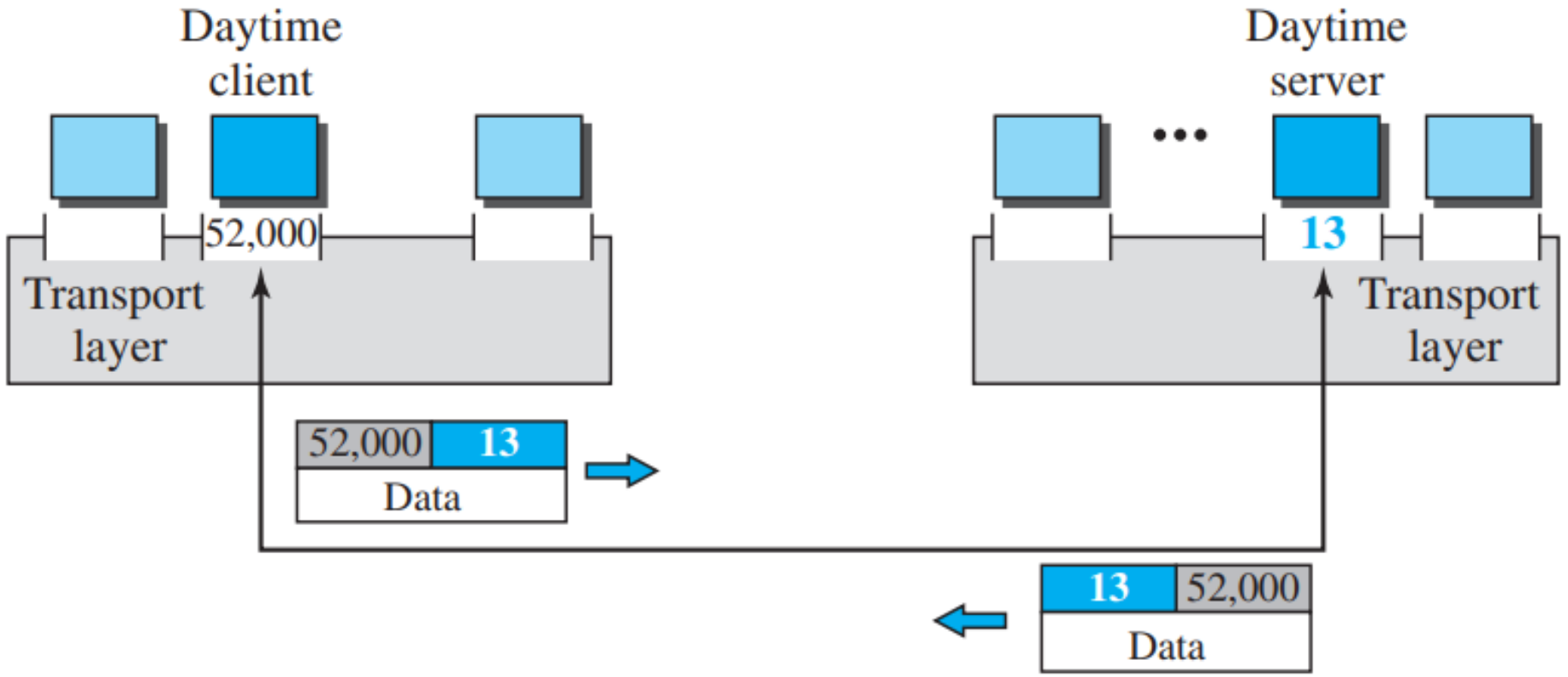
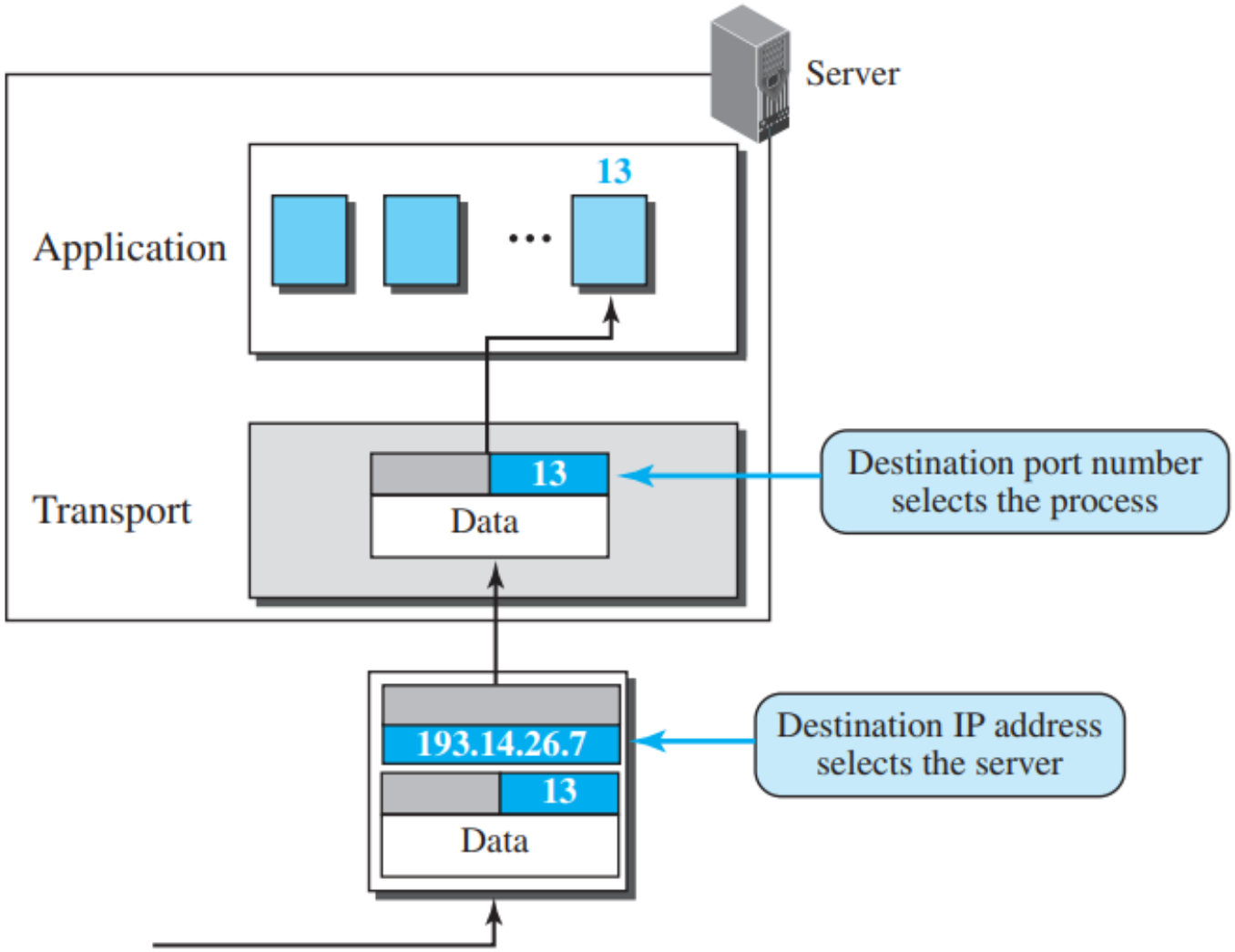


Figure 23.4 *IP addresses versus port numbers*



```
$grep     tftp/etc/services
tftp 69/tcp
tftp 69/udp
```

```
$grep     snmp/etc/services
snmp161/tcp#Simple Net Mgmt Proto
snmp161/udp#Simple Net Mgmt Proto
snmptrap162/udp#Traps for SNMP
```

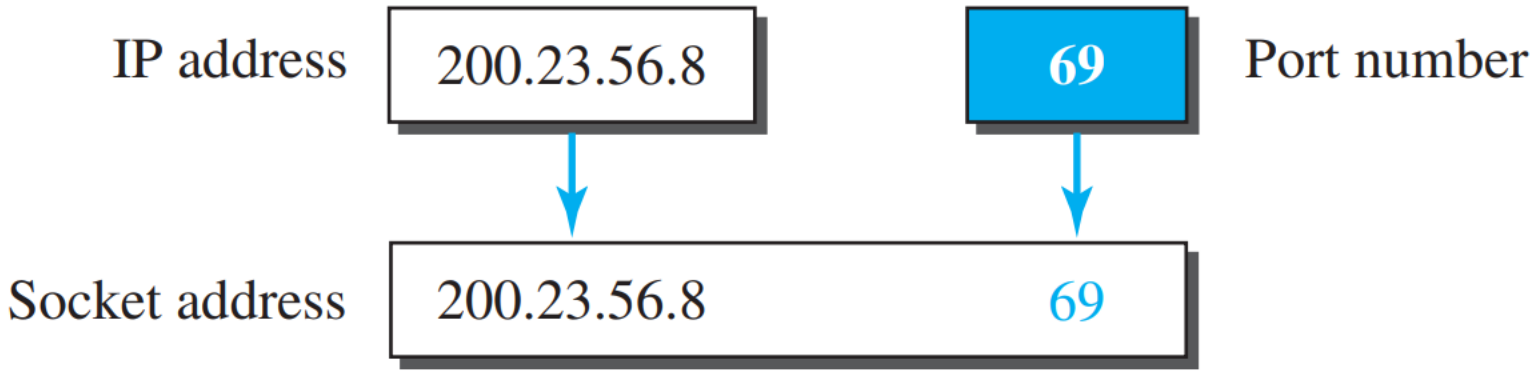
Socket Addresses

A transport-layer protocol in the TCP suite needs both the IP address and the port number, at each end, to make a connection. The combination of an IP address and a port number is called a ***socket address.*** The client socket address defines the client process uniquely just as the server socket address defines the server process uniquely (see Figure 23.6).

Socket Addresses

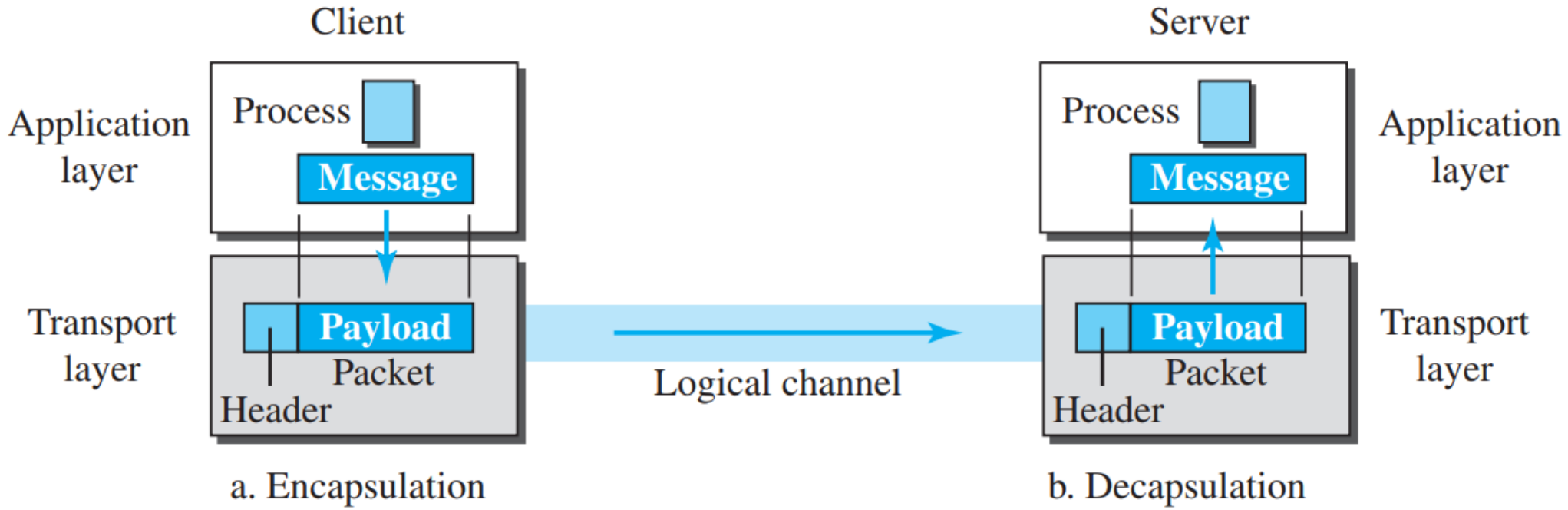
- The combination of an IP address and a port number is called a socket address. To use the services of the transport layer in the Internet, we need a pair of socket addresses: the client socket address and the server socket address. These four pieces of information are part of the network-layer packet header and the transport-layer packet header. The first header contains the IP addresses; the second header contains the port numbers.

Figure 23.6 *Socket address*



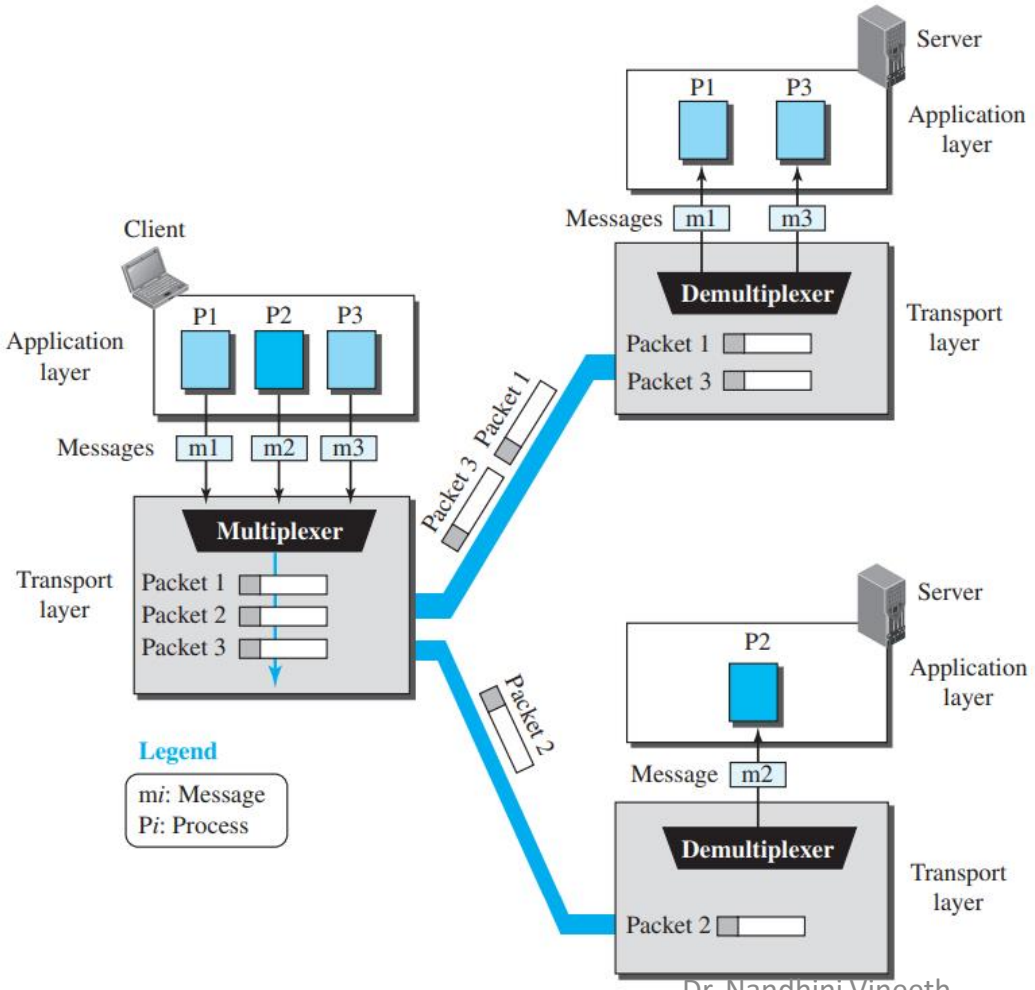
Encapsulation and Decapsulation

Figure 23.7 *Encapsulation and decapsulation*



Multiplexing and Demultiplexing

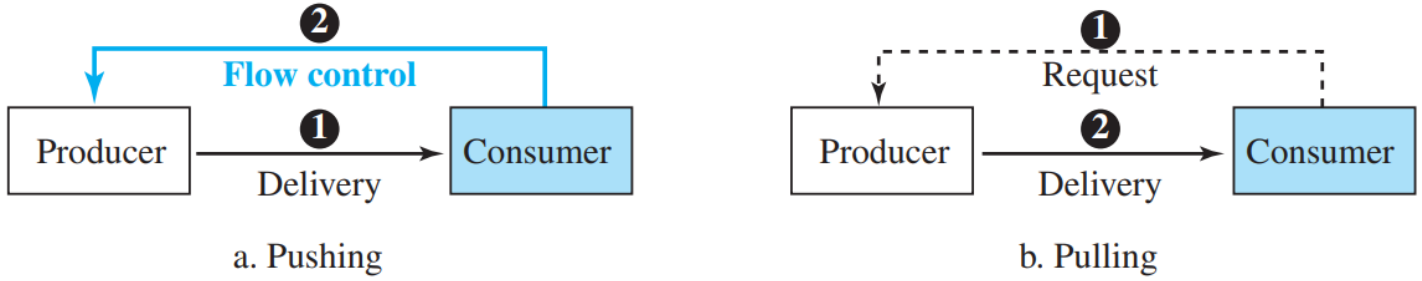
Figure 23.8 *Multiplexing and demultiplexing*



Flow Control

- If produced faster – packets discarded (Flow control)
- If consumer faster- waiting time is more and hence efficiency reduces
- Pushing or Pulling:
  - If the sender delivers items whenever they are produced–without a prior request from the consumer–the delivery is referred to as pushing.
  - If the producer delivers the items after the consumer has requested them, the delivery is referred to as pulling.

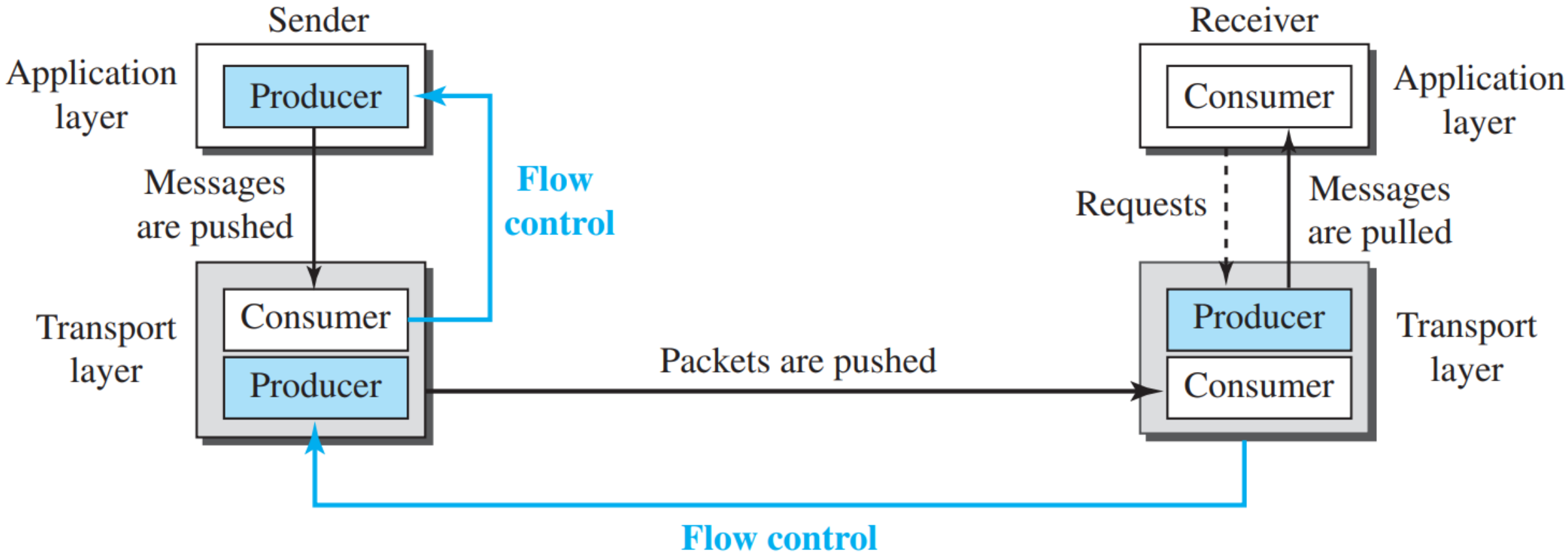
Figure 23.9 *Pushing or pulling*



Flow Control at Transport Layer

- In communication at the transport layer, we are dealing with four entities: sender process, sender transport layer, receiver transport layer, and receiver process
- The sending process at the AL -only a producer.
- The sending TL has a double role-
  - both a consumer and a producer
  - Consumer of AL messages
  - Producer for receiving TL.
- The receiving transport layer also has a double role:
  - consumer for sending TL
  - Producer for receiving AL
  - The last delivery, however, is normally a pulling delivery

Figure 23.10 *Flow control at the transport layer*



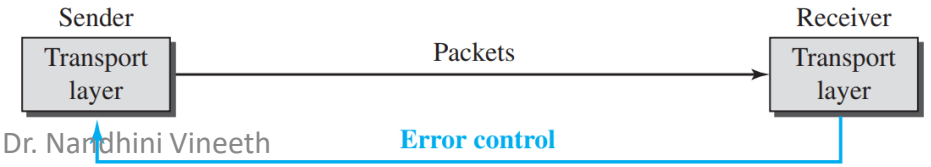
Buffers

- use two buffers: one at the sending transport layer and the other at the receiving transport layer.
- A buffer is a set of memory locations that can hold packets at the sender and receiver.
- Flow control communication
- When the buffer of the sending transport layer is full, it informs the application layer to stop passing chunks of messages; when there are some vacancies, it informs the application layer that it can pass message chunks again.
- When the buffer of the receiving transport layer is full, it informs the sending transport layer to stop sending packets. When there are some vacancies, it informs the sending transport layer that it can send packets again.

Error Control

- since the underlying network layer (IP) is unreliable, we need to make the transport layer reliable if the application requires reliability.
- Reliability can be achieved to add error control services to the transport layer.
- Error control at the transport layer is responsible for
  1. Detecting and discarding corrupted packets.
  2. Keeping track of lost and discarded packets and resending them.
  3. Recognizing duplicate packets and discarding them.
  4. Buffering out-of-order packets until the missing packets arrive.
- Error control, unlike flow control, involves only the sending and receiving transport layers

Figure 23.11 *Error control at the transport layer*



Sequence Numbers

- duplicate or packets arriving out of order - can be known if the packets are numbered.
- We can add a field to the transport-layer packet to hold the sequence number of the packet.
- If the header of the packet allows m bits for the sequence number, the sequence numbers range from 0 to 2^m − 1

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

In other words, the sequence numbers are modulo 2^m.

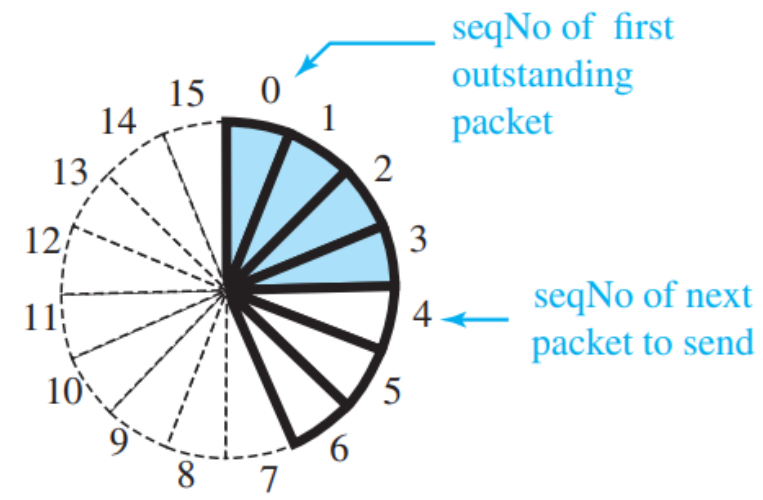
For error control, the sequence numbers are modulo 2^m, where m is the size of the sequence number field in bits.

Combination of Flow and Error Control

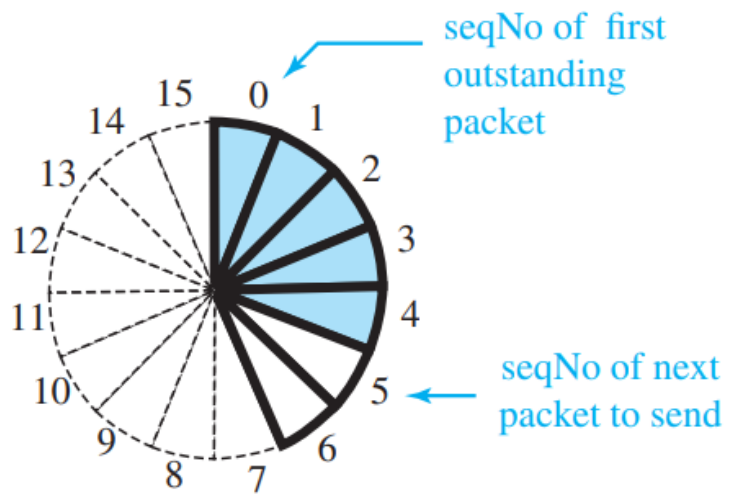
- Acknowledgment:
  - Sender – timer set – if expires resends
  - Out-of-order packets can be either discarded (to be treated as lost packets by the sender), or stored until the missing one arrives.
- Combination of Flow and Error Control
  - flow control requires the use of two buffers
  - error control requires the use of sequence and acknowledgment numbers by both sides
  - Combn: we use two numbered buffers, one at the sender, one at the receiver.
  - the number of the next free location, x, in the buffer as the sequence number of the packet
  - When an acknowledgment related to a sent packet arrives, the packet is purged and the memory location becomes free.
  - At the receiver, when a packet with sequence number y arrives, it is stored at the memory location y until the application layer is ready to receive it. An acknowledgment can be sent to announce the arrival of packet y.

- Sl

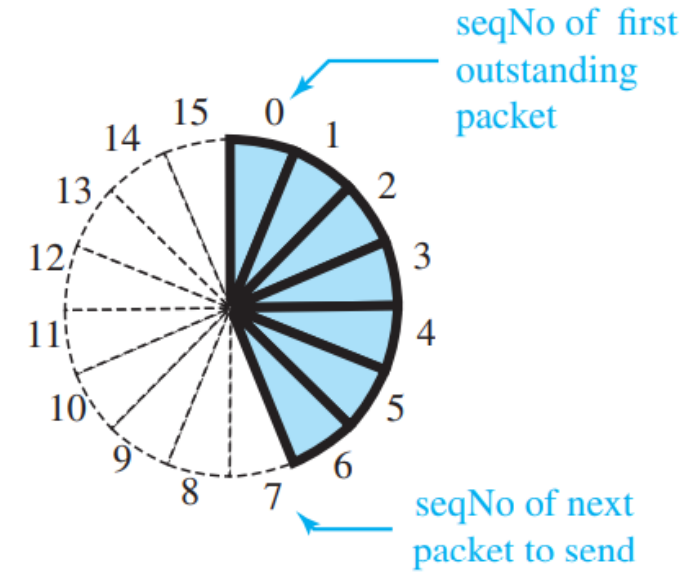
Figure 23.12 *Sliding window in circular format*



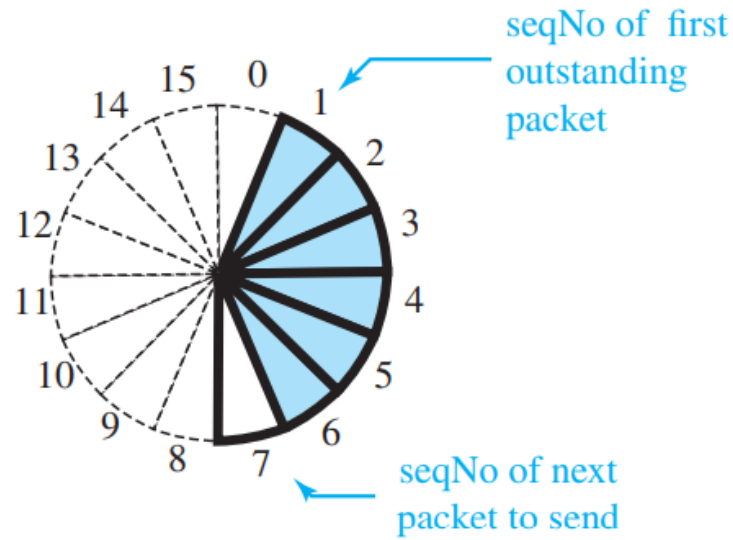
a. Four packets have been sent.



b. Five packets have been sent.

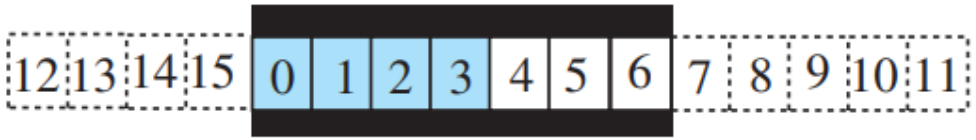


c. Seven packets have been sent; window is full.



d. Packet 0 has been acknowledged; window slides.

Figure 23.13 *Sliding window in linear format*



a. Four packets have been sent.

b. Five packets have been sent.

c. Seven packets have been sent; window is full.

d. Packet 0 has been acknowledged; window slides.

Congestion Control

- mechanisms and techniques that control the congestion and keep the load below the capacity.
- Congestion in a network or internetwork occurs because routers and switches have queues—buffers that hold the packets before and after processing.
- A router, for example, has an input queue and an output queue for each interface. If a router cannot process the packets at the same rate at which they arrive, the queues become overloaded and congestion occurs.
- Congestion at the transport layer is actually the result of congestion at the network layer, which manifests itself at the transport layer.

Connectionless and Connection-Oriented Protocols

- Connectionless service at the transport layer means independency between packets; connection-oriented means dependency.
- Connectionless Service:

Figure 23.14 *Connectionless service*

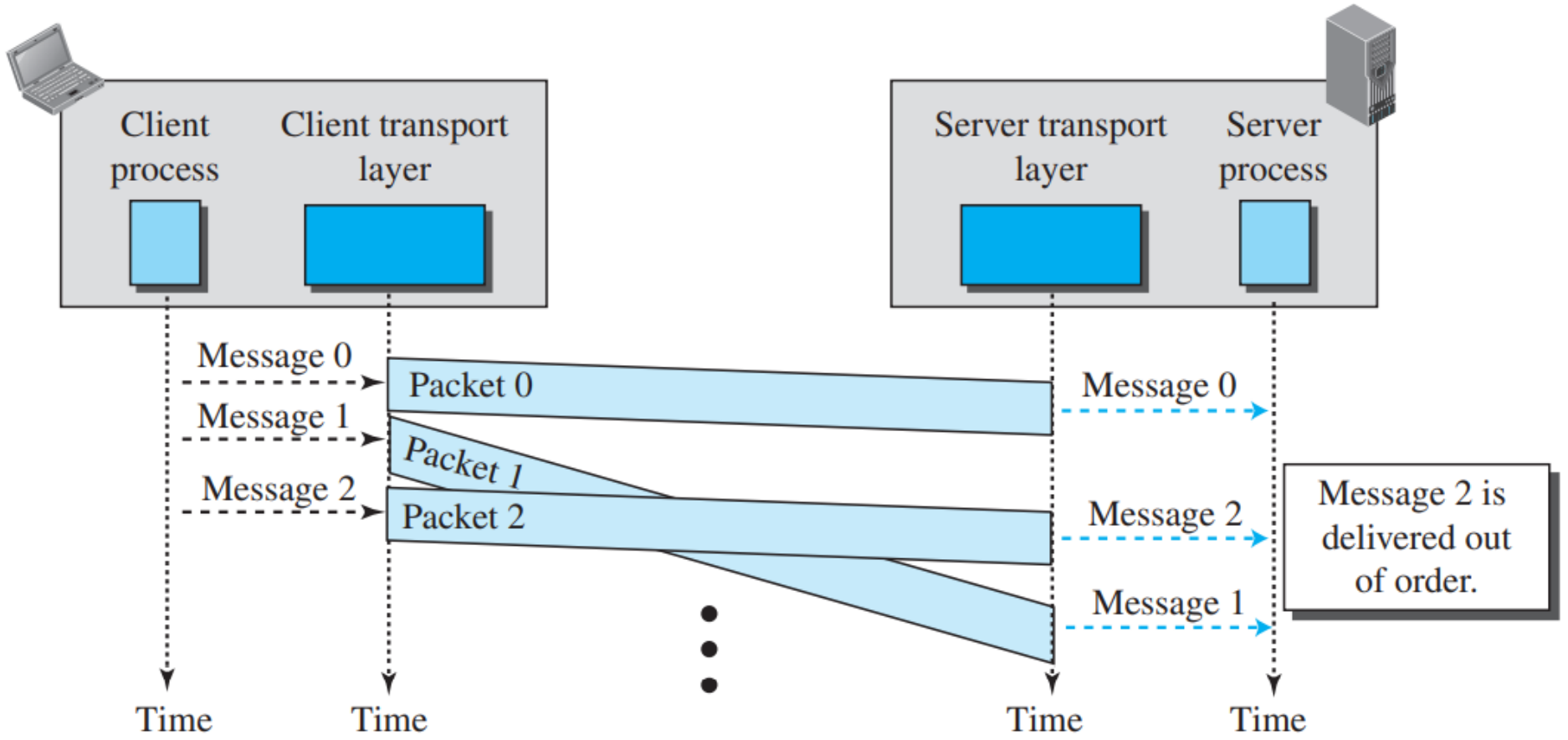


Figure 23.15 *Connection-oriented service*

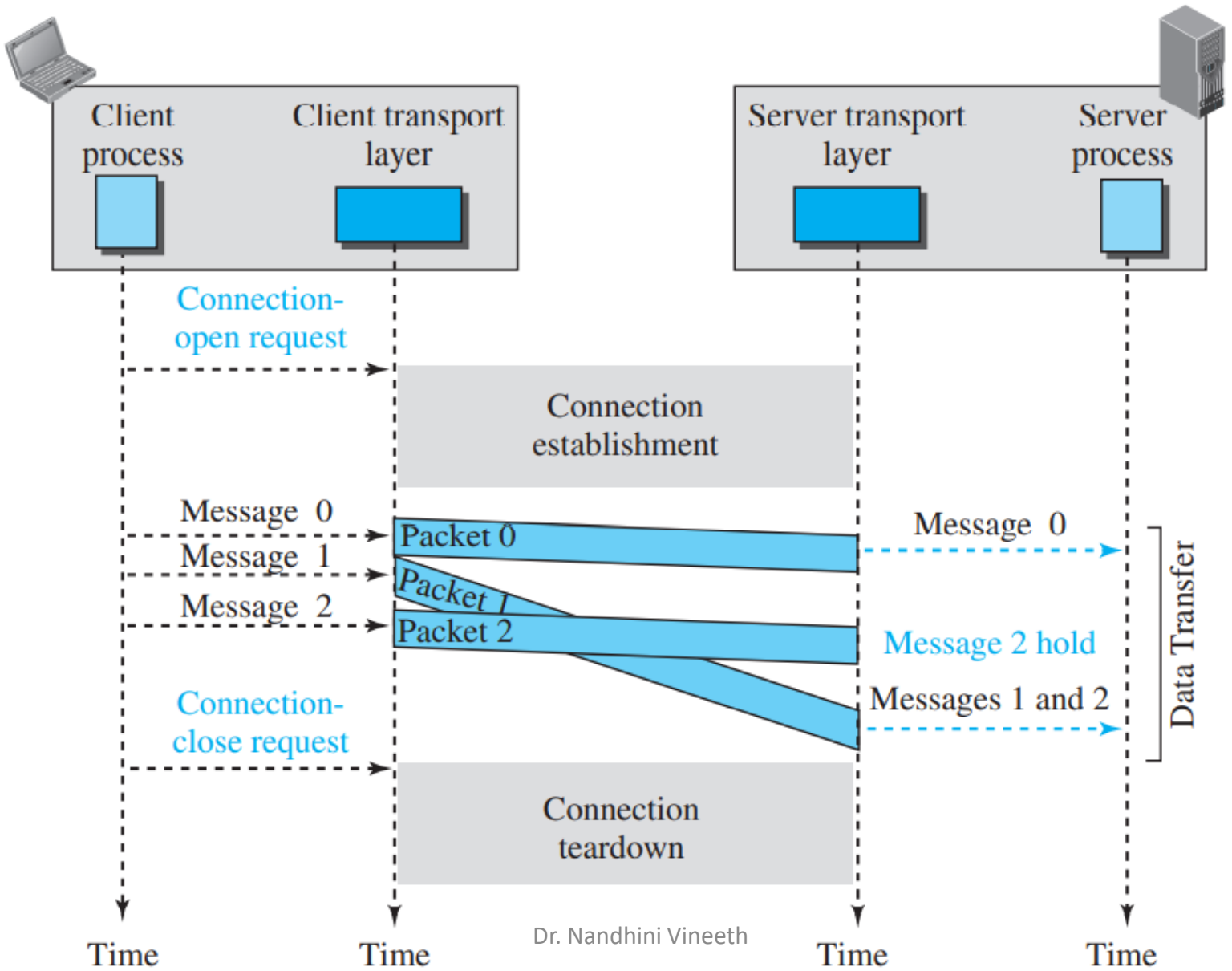
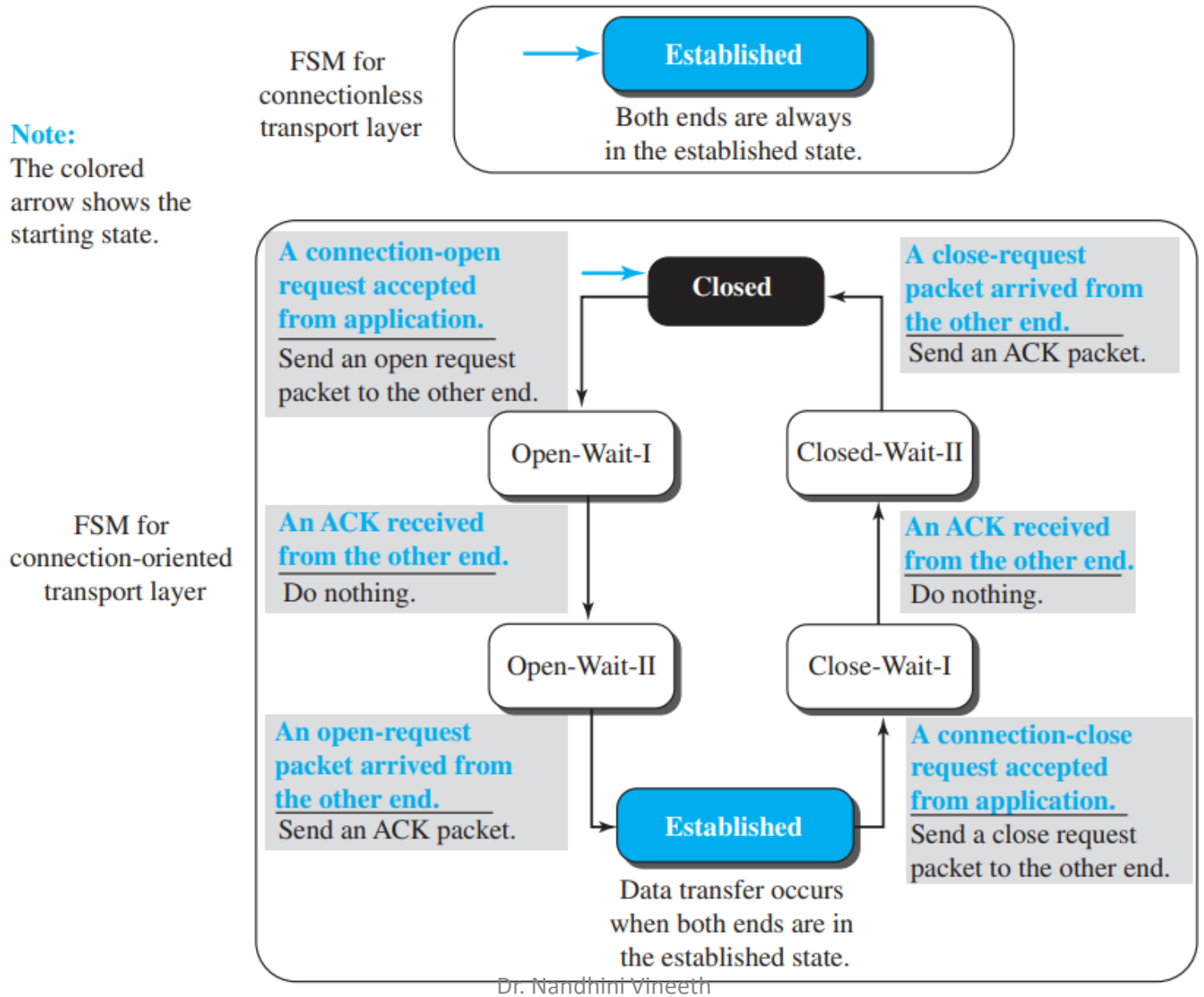


Figure 23.16 *Connectionless and connection-oriented service represented as FSMs*



23.2 TRANSPORT-LAYER PROTOCOLS - Simple Protocol

Figure 23.17 *Simple protocol*

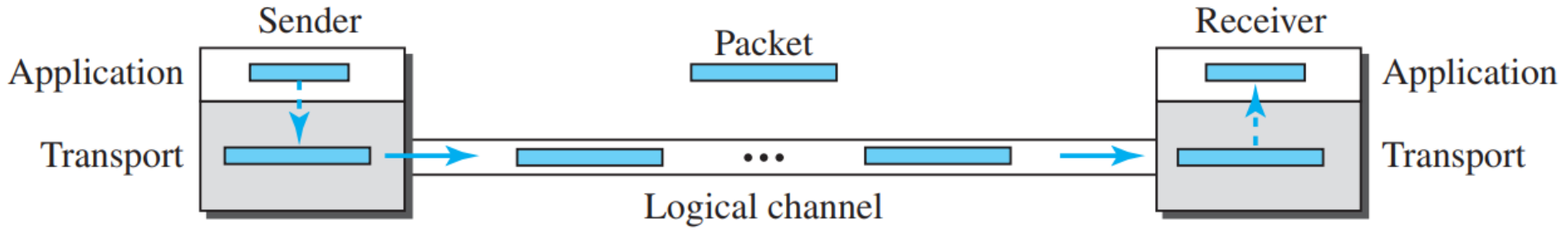


Figure 23.18 *FSMs for the simple protocol*

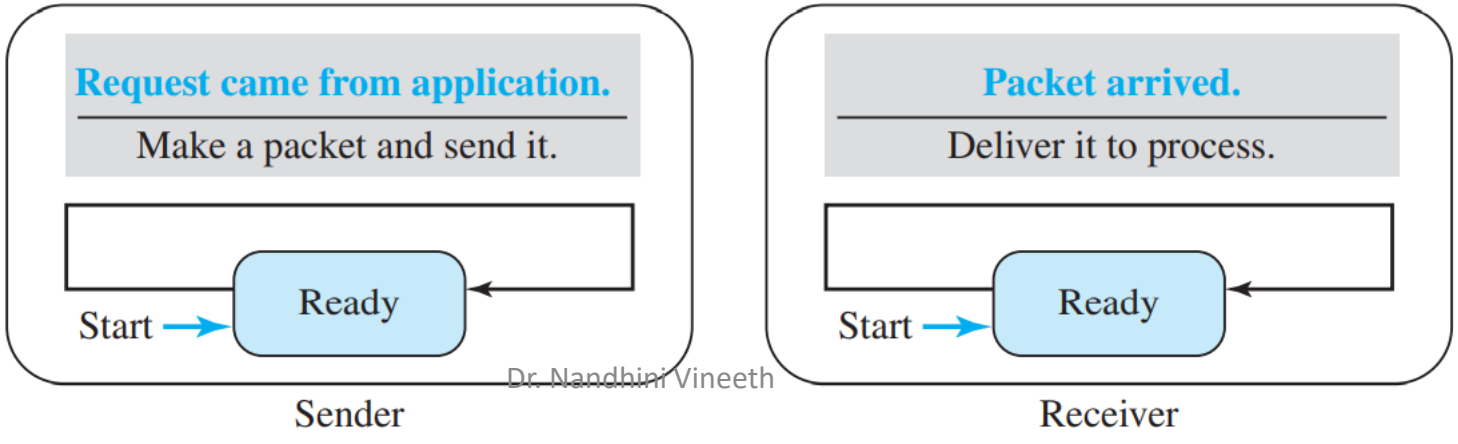
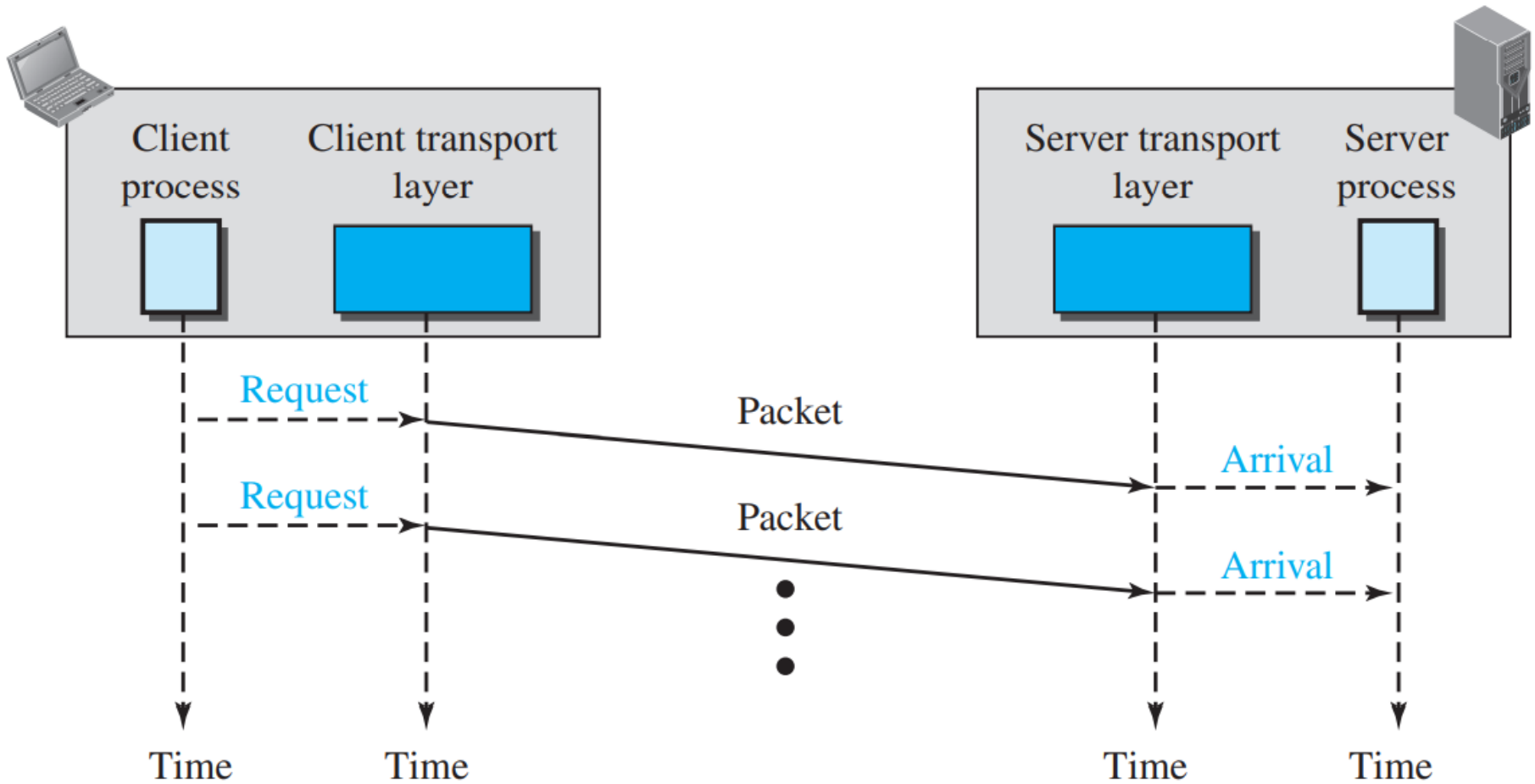
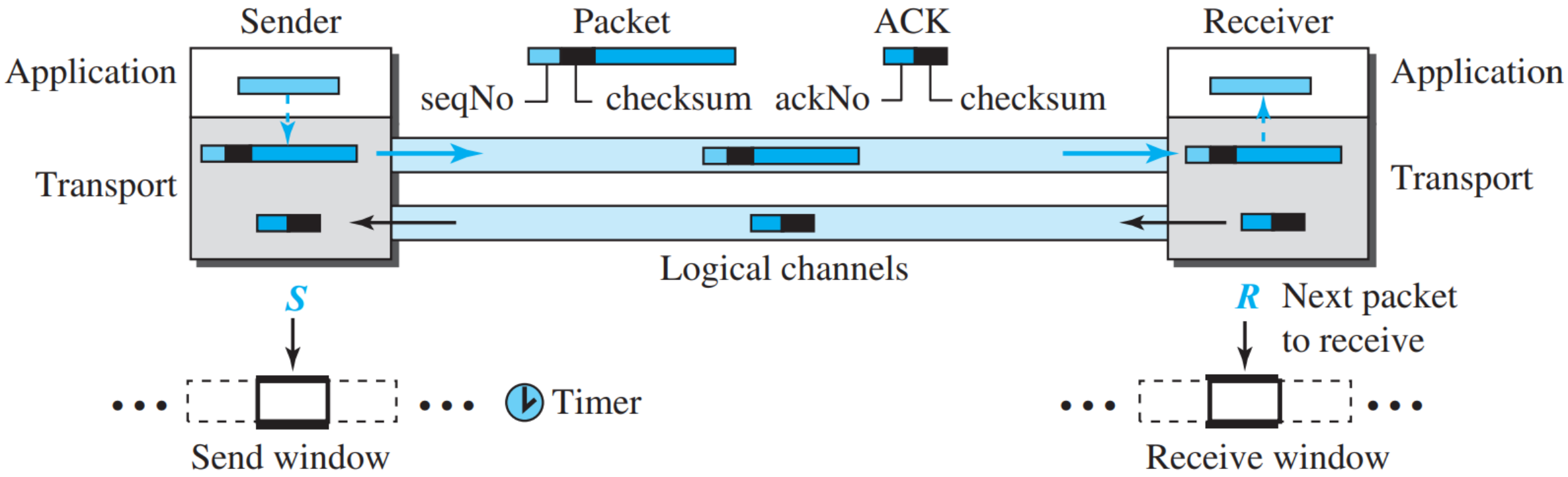


Figure 23.19 *Flow diagram for Example 23.3*



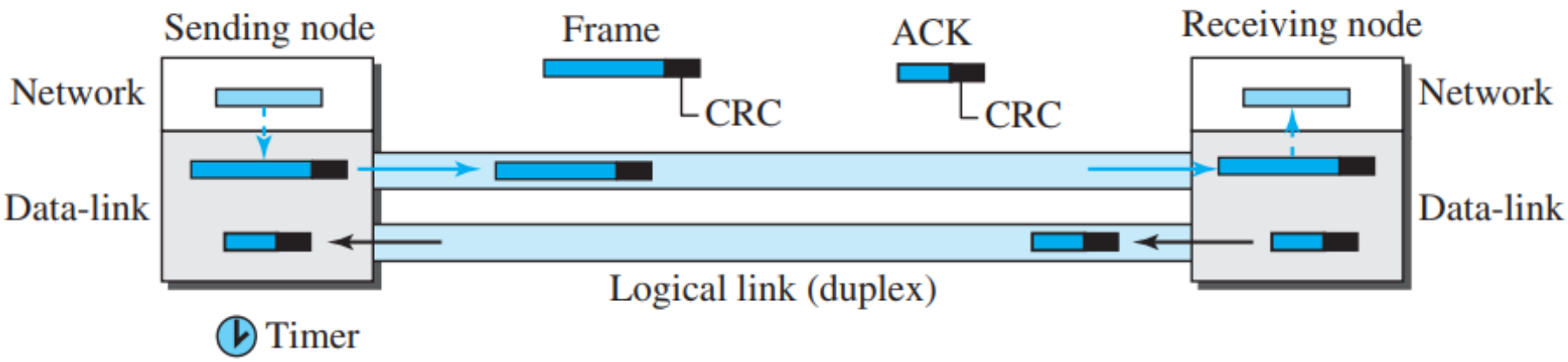
Stop-and-Wait Protocol

Figure 23.20 *Stop-and-Wait protocol*



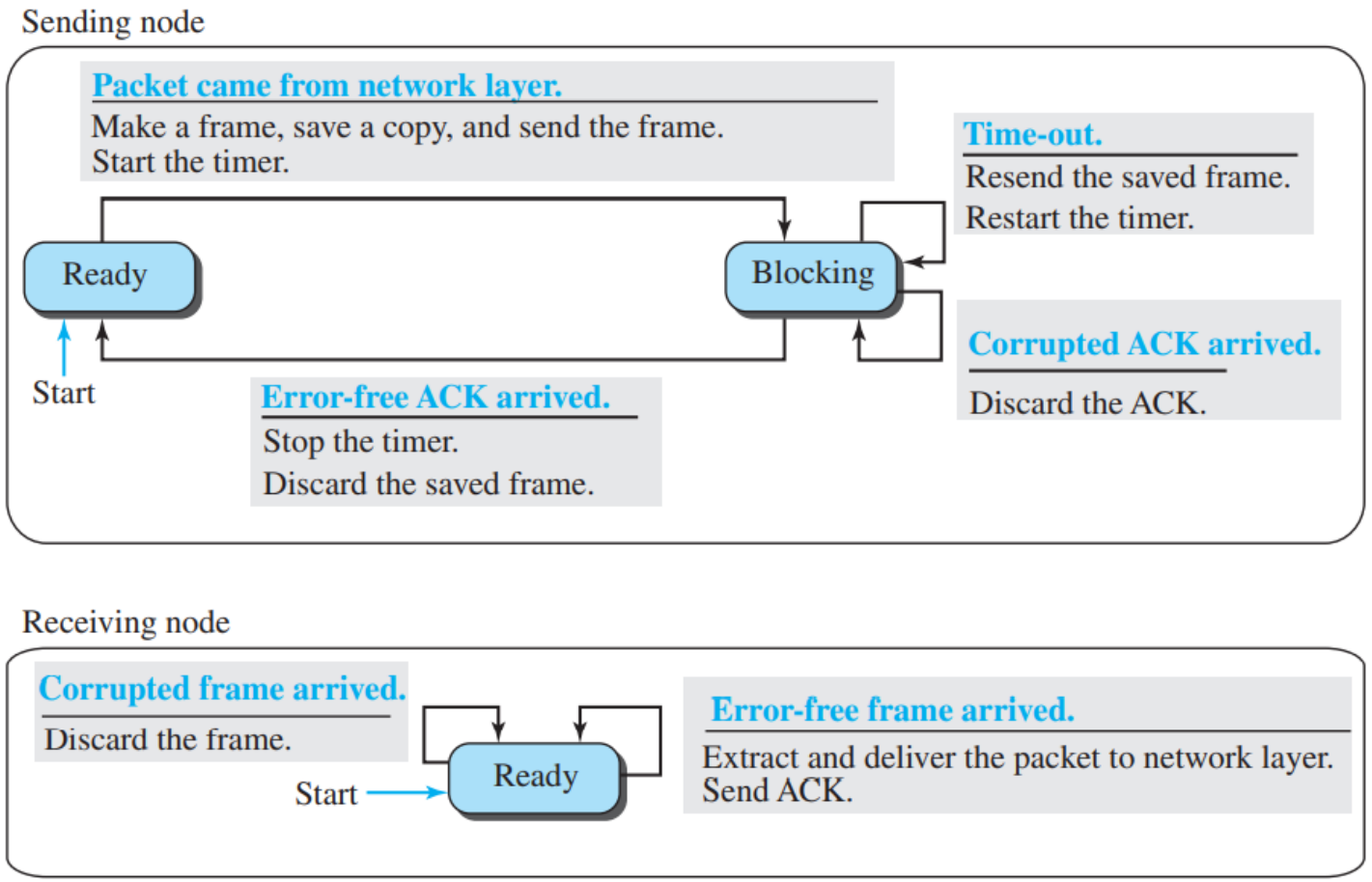
DLL – ONLY FOR COMPARISON

Figure 11.10 *Stop-and-Wait protocol*



DLL – ONLY FOR COMPARISON

Figure 11.11 *FSM for the Stop-and-Wait protocol*



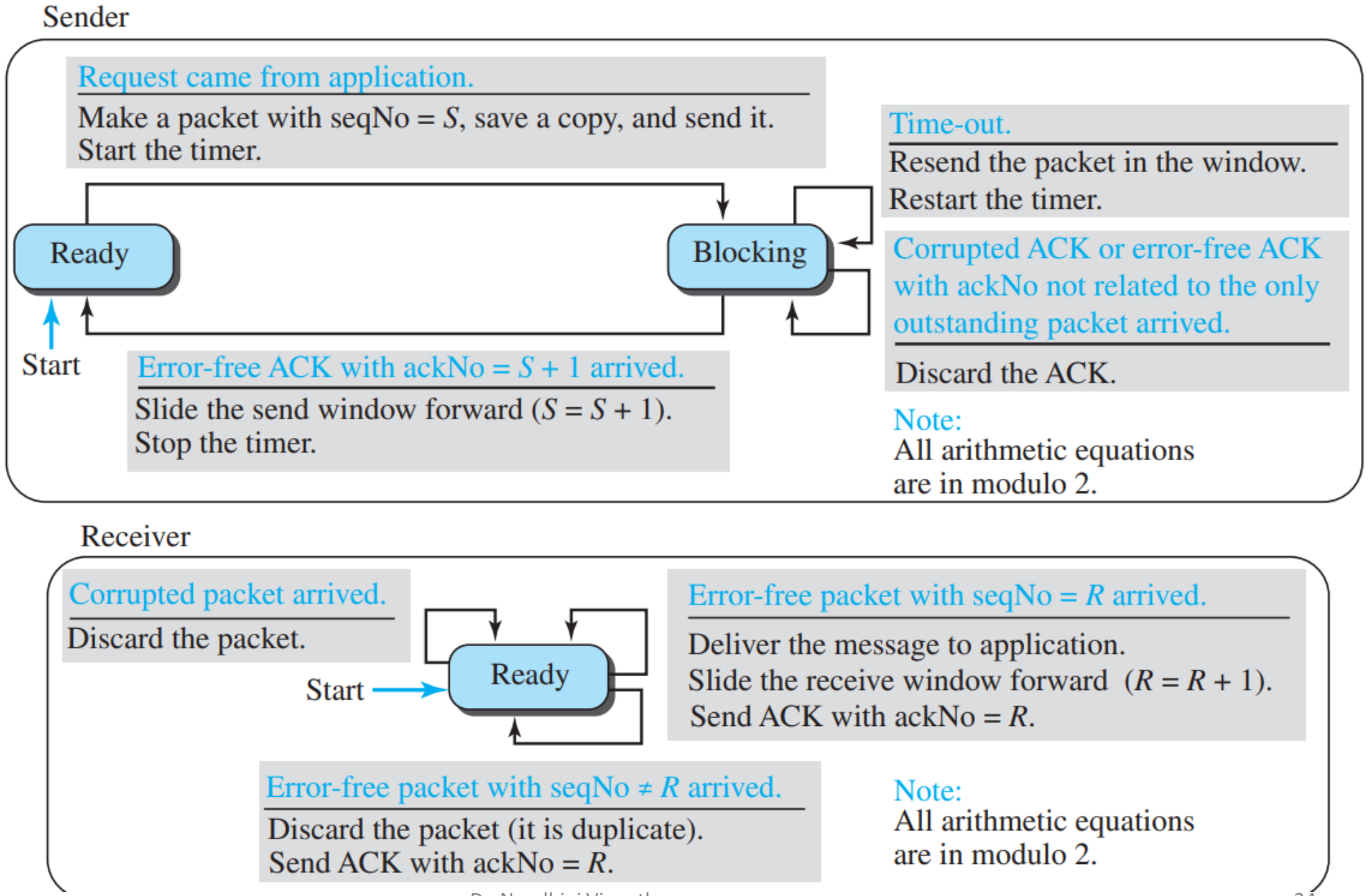
Sequence Numbers

- 1. The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment. The acknowledgment arrives at the sender site, causing the sender to send the next packet numbered x + 1.
- 2. The packet is corrupted or never arrives at the receiver site; the sender resends the packet (numbered x) after the time-out. The receiver returns an acknowledgment.
- 3. The packet arrives safe and sound at the receiver site; the receiver sends an acknowledgment, but the acknowledgment is corrupted or lost. The sender resends the packet (numbered x) after the time-out. Note that the packet here is a duplicate. The receiver can recognize this fact because it expects packet x + 1 but packet x was received.

Acknowledgment Numbers

In the Stop-and-Wait protocol, the acknowledgment number always announces, in modulo-2 arithmetic, the sequence number of the next packet expected.

FSM


Sender
Request came from application.
Make a packet with seqNo = S, save a copy, and send it.
Start the timer.
Time-out.
Resend the packet in the window.
Restart the timer.
Ready
Blocking
Start
Corrupted ACK or error-free ACK with ackNo not related to the only outstanding packet arrived.
Discard the ACK.
Error-free ACK with ackNo = S + 1 arrived.
Slide the send window forward (S = S + 1).
Stop the timer.
Note:
All arithmetic equations are in modulo 2.
Receiver
Corrupted packet arrived.
Discard the packet.
Error-free packet with seqNo = R arrived.
Deliver the message to application.
Slide the receive window forward (R = R + 1).
Send ACK with ackNo = R.
Start
Ready
Error-free packet with seqNo ≠ R arrived.
Discard the packet (it is duplicate).
Send ACK with ackNo = R.
Note:
All arithmetic equations are in modulo 2.

DIFF STATES

Sender

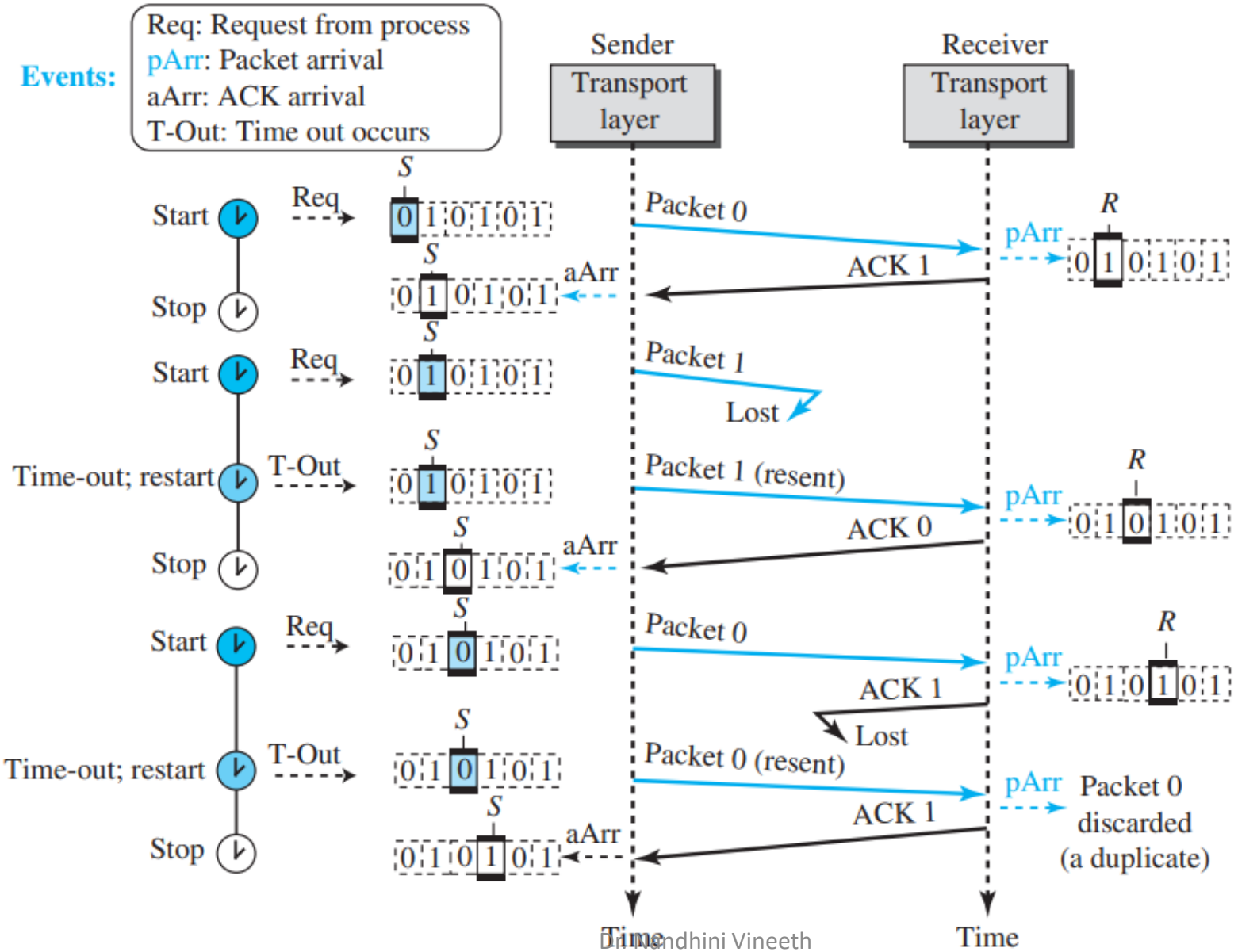
The sender is initially in the ready state, but it can move between the ready and blocking state. The variable S is initialized to 0.

Ready state. When the sender is in this state, it is only waiting for one event to occur. If a request comes from the application layer, the sender creates a packet with the sequence number set to S. A copy of the packet is stored, and the packet is sent. The sender then starts the only timer. The sender then moves to the blocking state.

Blocking state. When the sender is in this state, three events can occur: a. If an error-free ACK arrives with the ackNo related to the next packet to be sent, which means ackNo = (S + 1) modulo 2, then the timer is stopped. The window slides, S = (S + 1) modulo 2. Finally, the sender moves to the ready state. b. If a corrupted ACK or an error-free ACK with the ackNo ≠ (S + 1) modulo 2 arrives, the ACK is discarded. c. If a time-out occurs, the sender resends the only outstanding packet and restarts the timer.

Receiver The receiver is always in the ready state. Three events may occur: a. If an error-free packet with seqNo = R arrives, the message in the packet is delivered to the application layer. The window then slides, R = (R + 1) modulo 2. Finally an ACK with ackNo = R is sent. b. If an error-free packet with seqNo ≠ R arrives, the packet is discarded, but an ACK with ackNo = R is sent. c. If a corrupted packet arrives, the packet is discarded.

Figure 23.22 *Flow diagram for Example 23.4*

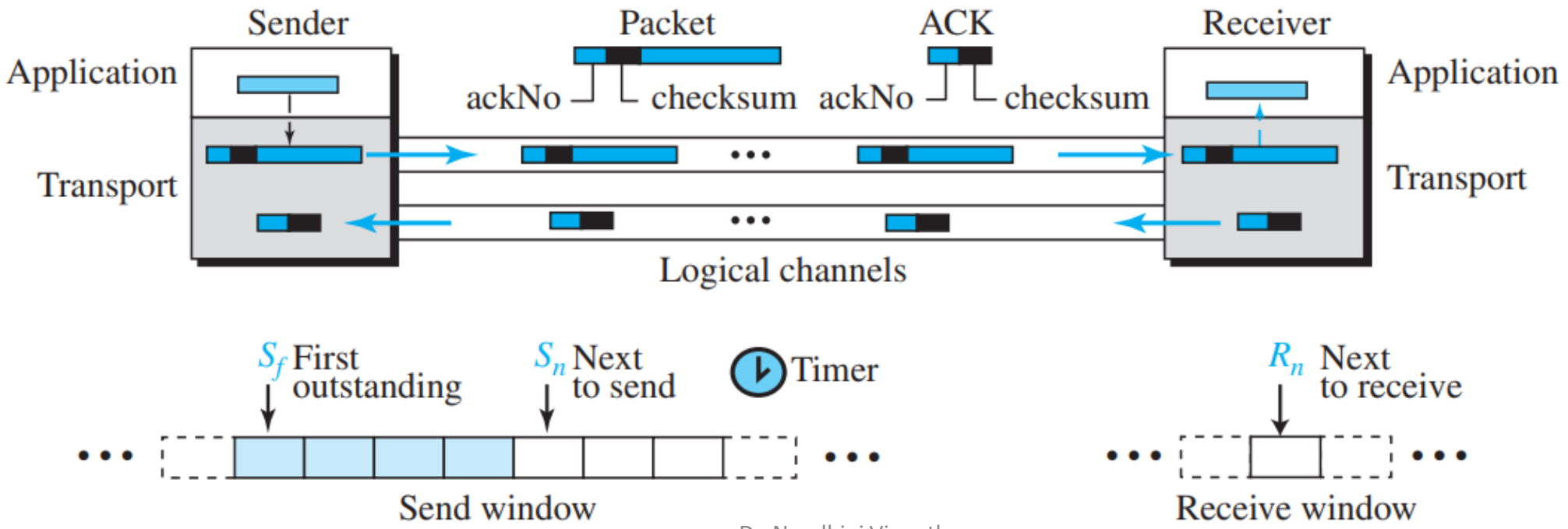


Go-Back-N Protocol (GBN)

- In the Go-Back-N protocol,
  - we can send several packets before receiving acknowledgments, but the receiver can only buffer one packet. We keep a copy of the sent packets until the acknowledgments arrive.
  - the acknowledgment number is cumulative and defines the sequence number of the next packet expected to arrive
  - The send window at any time divides the possible sequence numbers into **four regions**.
    - The first region, left of the window, defines the sequence numbers belonging to packets that are already acknowledged. The sender does not worry about these packets and keeps no copies of them.
    - The second region, colored, defines the range of sequence numbers belonging to the packets that have been sent, but have an unknown status. The sender needs to wait to find out if these packets have been received or were lost. We call these outstanding packets.
    - The third range, white in the figure, defines the range of sequence numbers for packets that can be sent; however, the corresponding data have not yet been received from the application layer.
    - fourth region, right of the window, defines sequence numbers that cannot be used until the window slides

Go-Back-N Protocol (GBN)

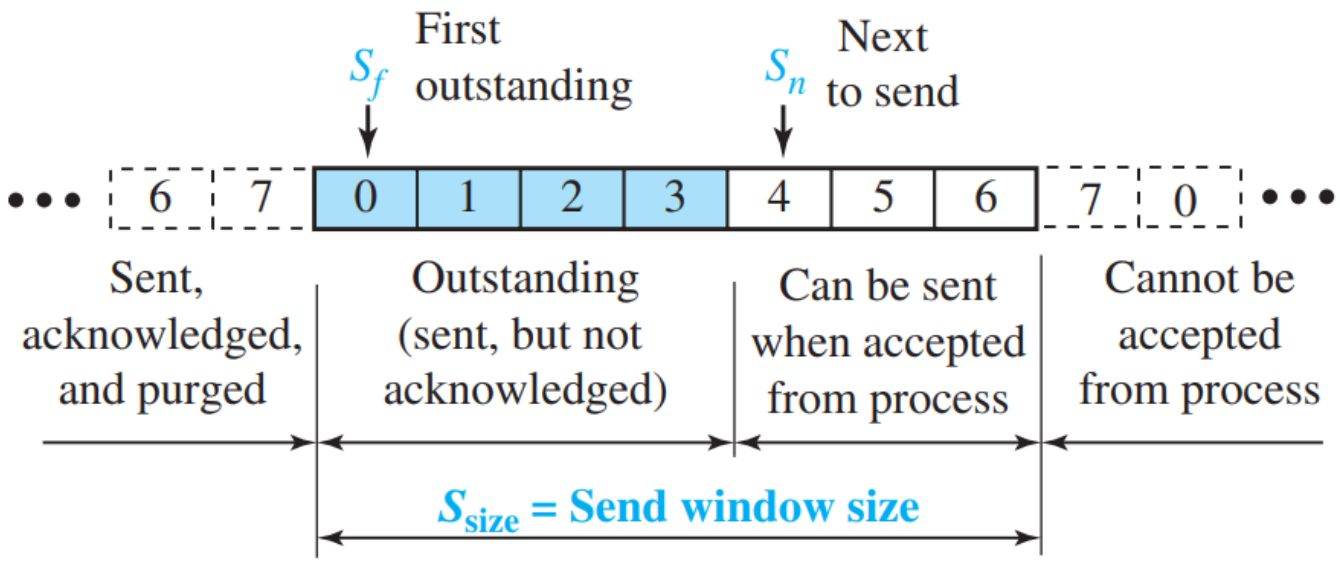
Figure 23.23 *Go-Back-N protocol*



Go-Back-N Protocol (GBN)

The send window is an abstract concept defining an imaginary box of maximum size = 2m − 1 with three variables: Sf, Sn, and Ssize.

Figure 23.24 *Send window for Go-Back-N*



Go-Back-N Protocol (GBN)

The send window can slide one or more slots when an error-free ACK with ackNo greater than or equal to Sf and less than Sn (in modular arithmetic) arrives.

Figure 23.25 *Sliding the send window*

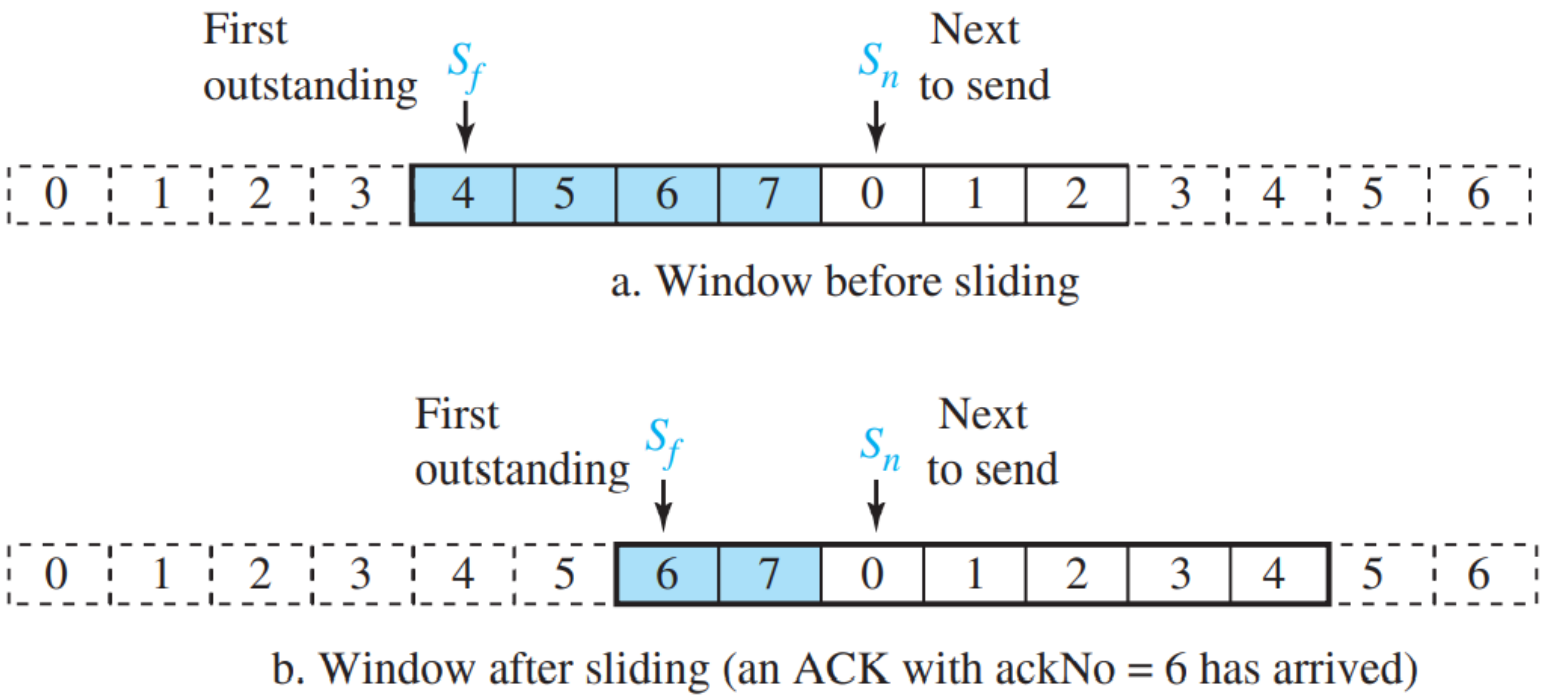


Figure 23.26 *Receive window for Go-Back-N*

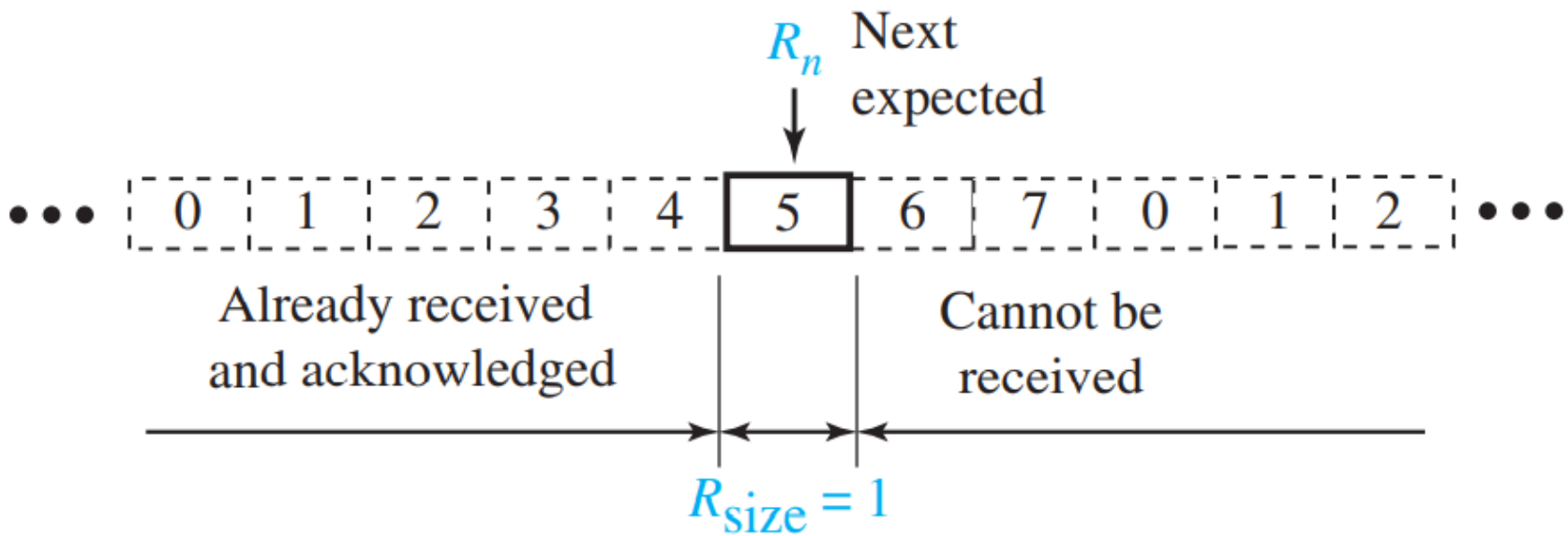


Figure 23.27 *FSMs for the Go-Back-N protocol*

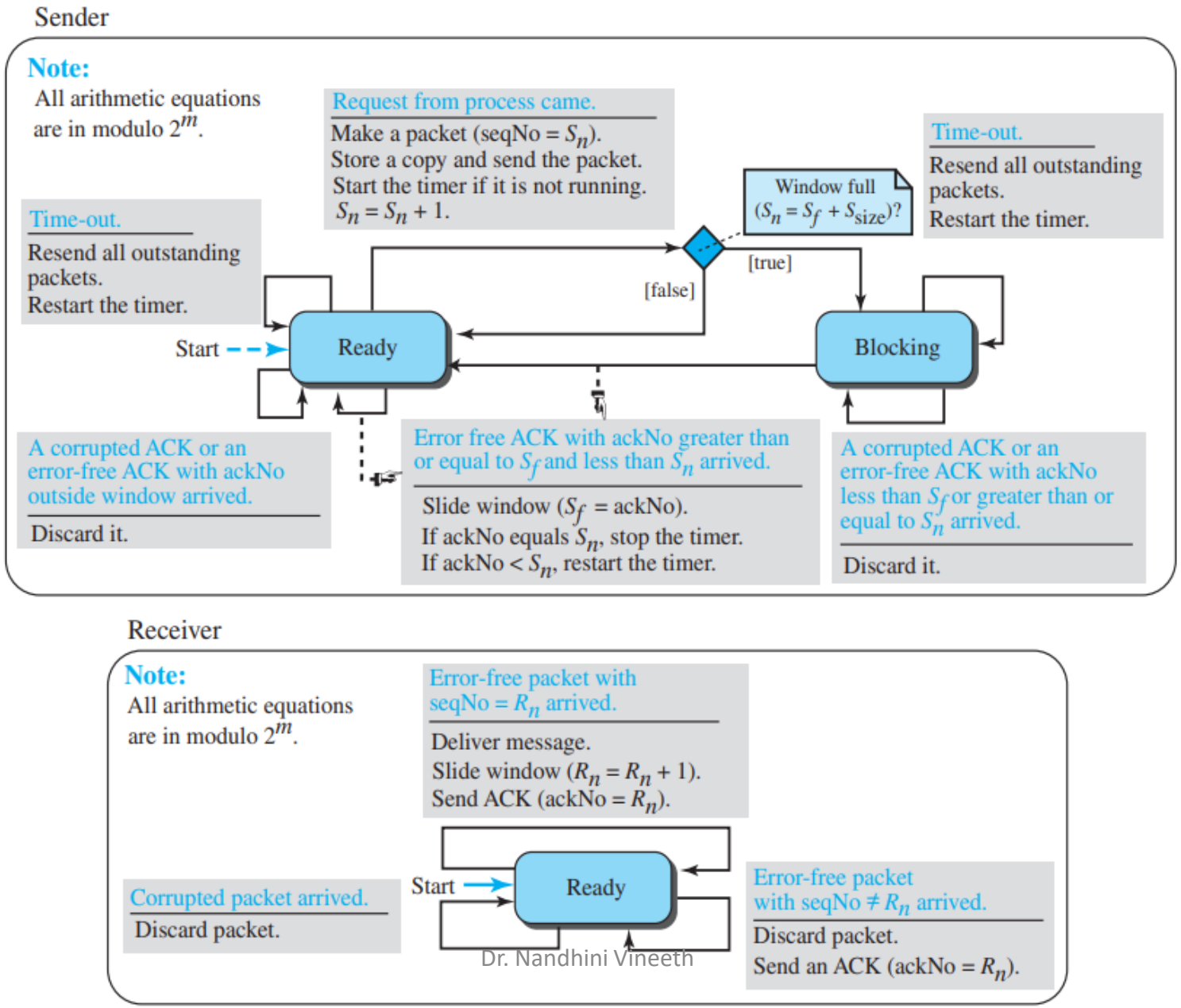
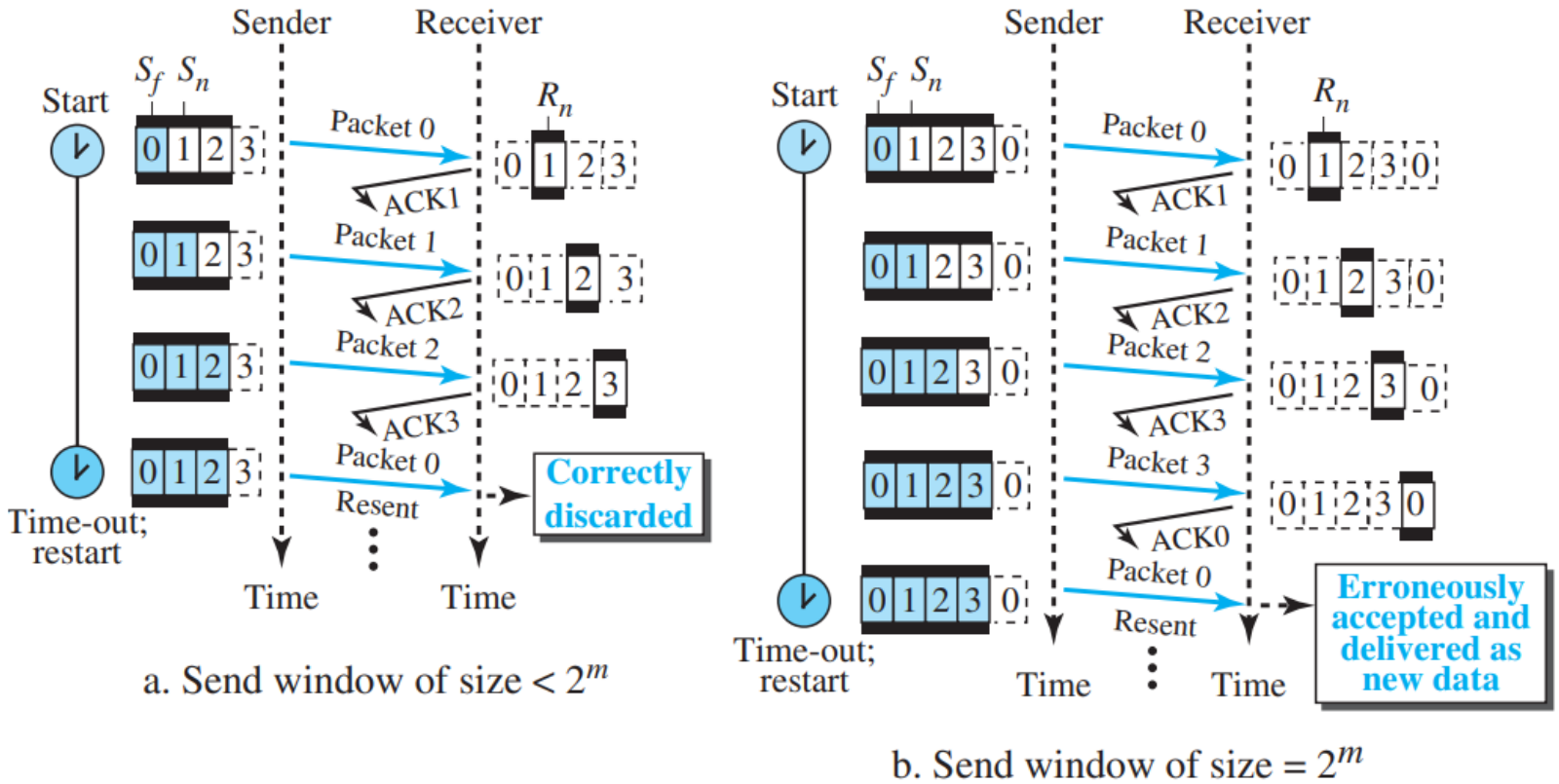


Figure 23.28 *Send window size for Go-Back-N*



In the Go-Back-*N* protocol, the size of the send window must be less than 2^m; the size of the receive window is always 1.

Figure 23.29 *Flow diagram for Example 23.7*

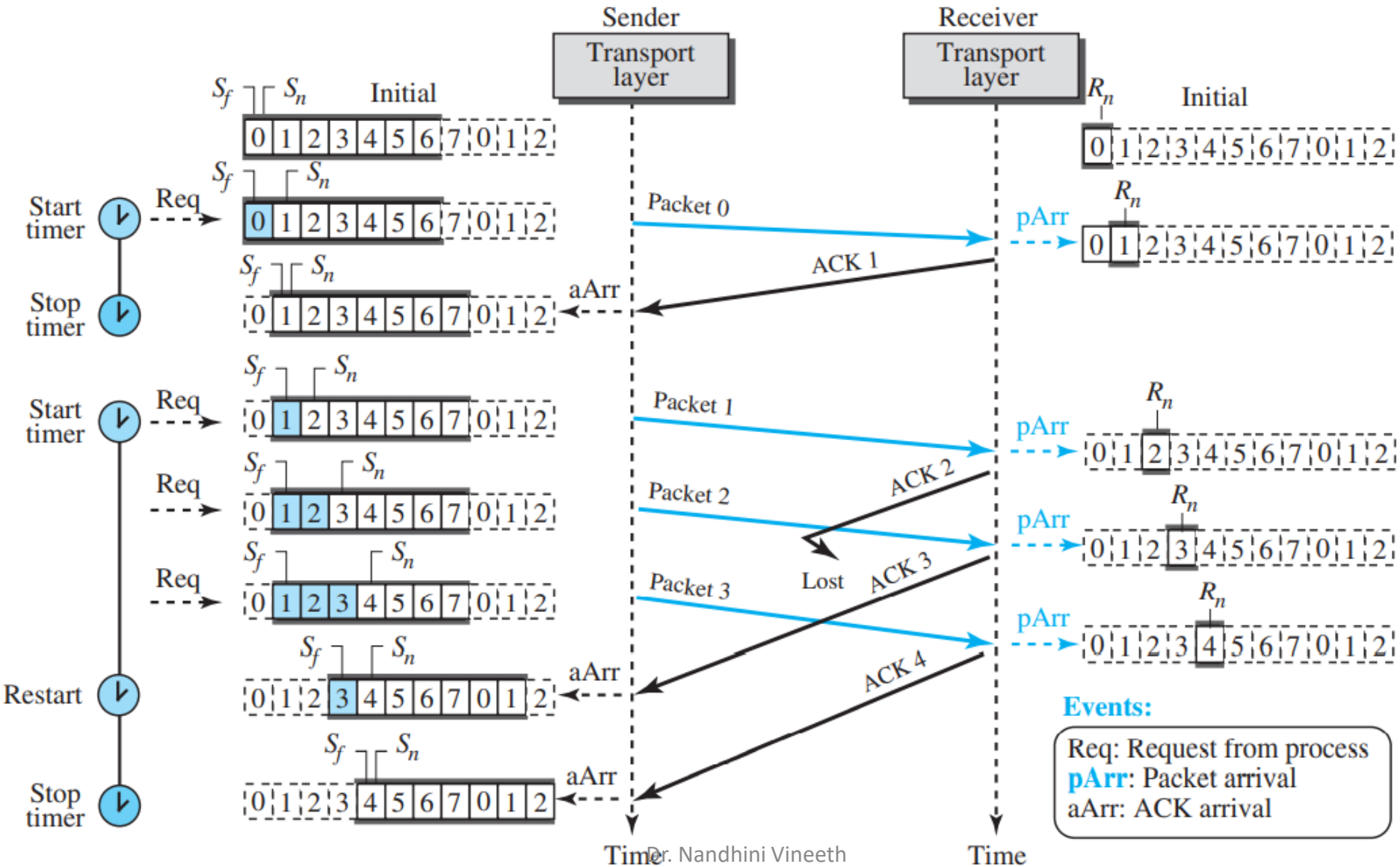
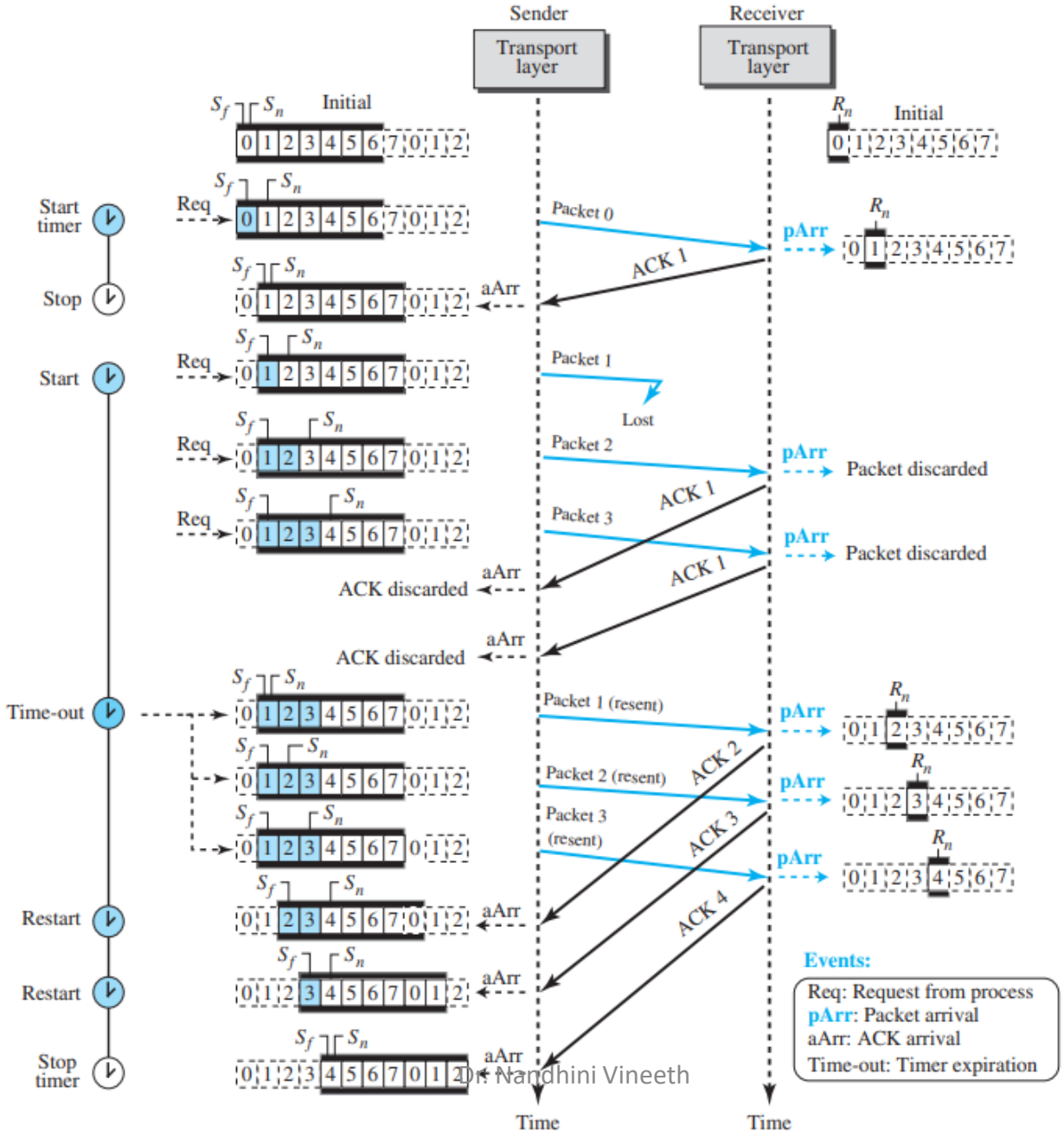


Figure 23.30 *Flow diagram for Example 23.8*



Selective-Repeat Protocol

- resends only selective packets, those that are actually lost.
- WINDOWS
  - sending window
    - **SR-The send window maximum size can be 2^{m-1}**
    - **GN-The send window maximum size can be 2^m -1**
  - **Receive window**
    - **SR – Same as sending window (Out of Order delivery not supported)**
    - **GN – 1 (Out of Order delivery supported)**

Timers:

Theoretically – one timer sep for each of the outstanding packets

Most protocols – only one timer for all outstanding packets

Acknowledgments:

SR – Sep ack for each

GN - Cumulative ack

Selective-Repeat Protocol

- Example 23.9
- Assume a sender sends 6 packets: packets 0, 1, 2, 3, 4, and 5. The sender receives an ACK with ackNo = 3.
- What is the interpretation if the system is using GBN or SR?
- Solution If the system is using GBN, it means that packets 0, 1, and 2 have been received uncorrupted and the receiver is expecting packet 3. If the system is using SR, it means that packet 3 has been received uncorrupted; the ACK does not say anything about other packets.

Selective-Repeat Protocol

Figure 23.31 *Outline of Selective-Repeat*

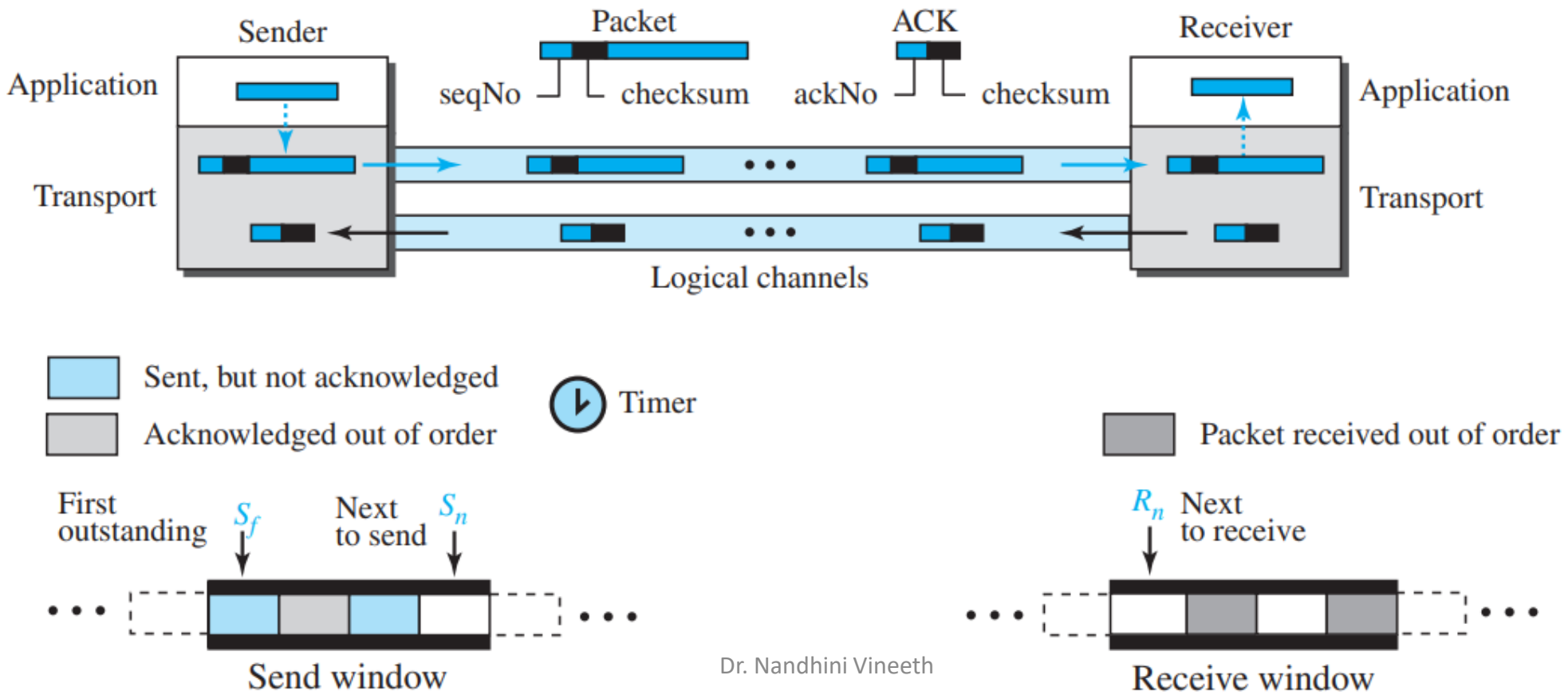


Figure 23.32 *Send window for Selective-Repeat protocol*

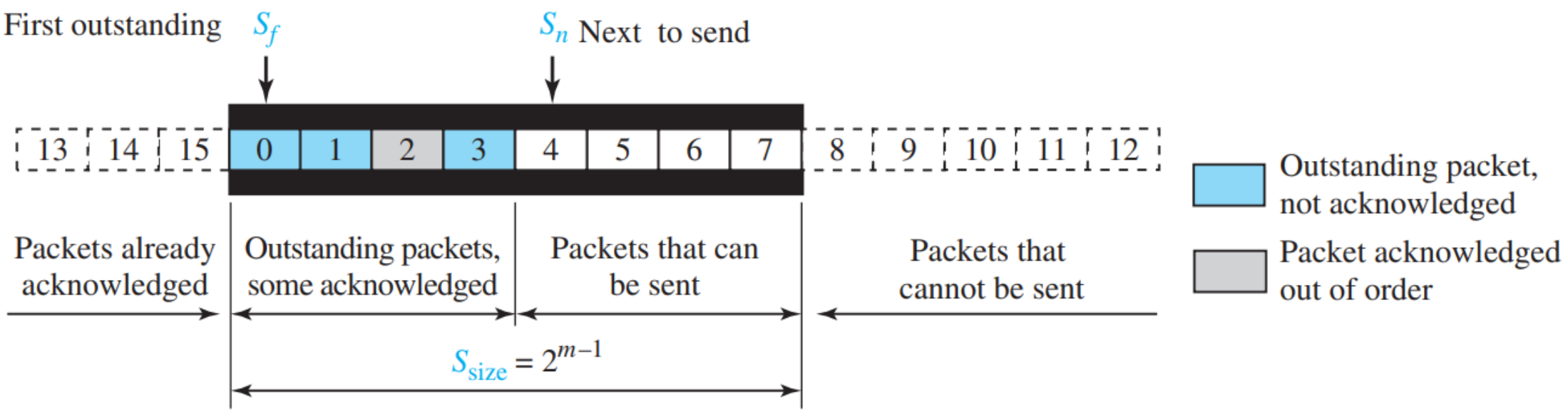
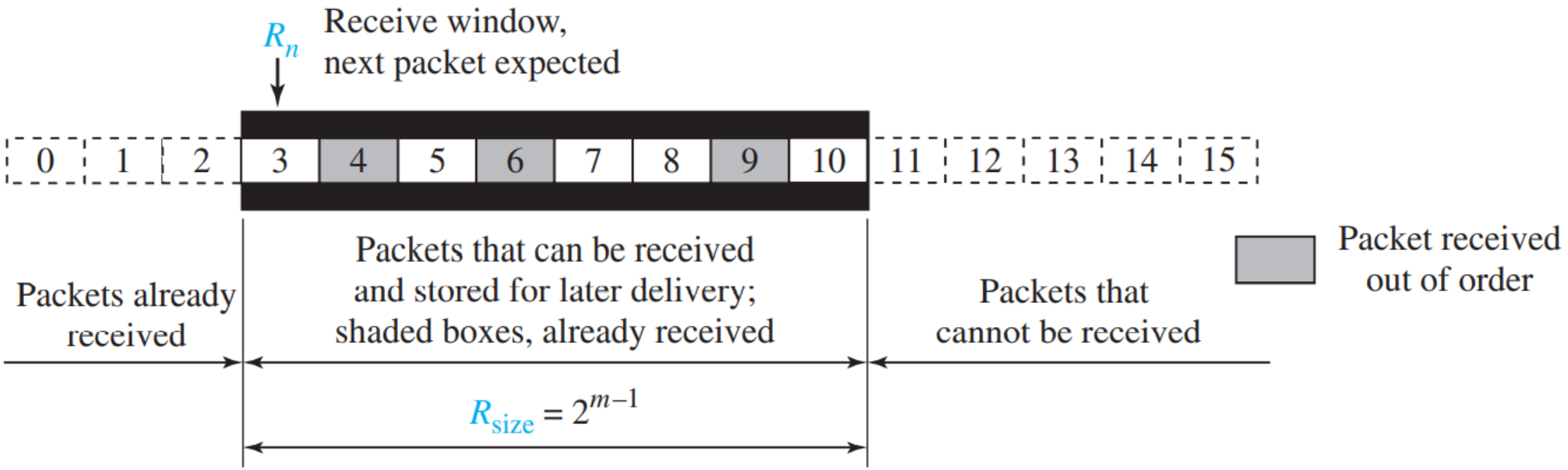
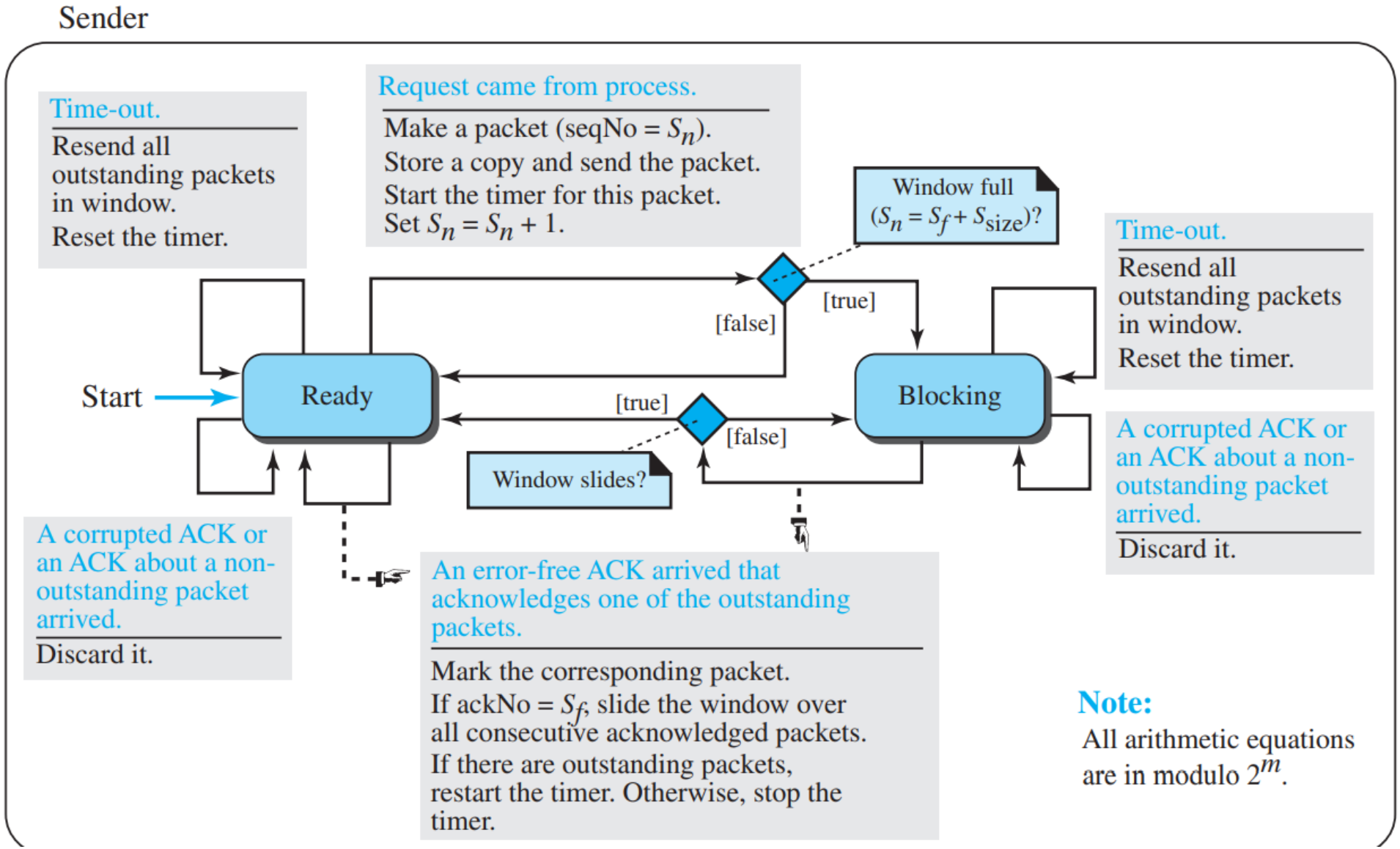
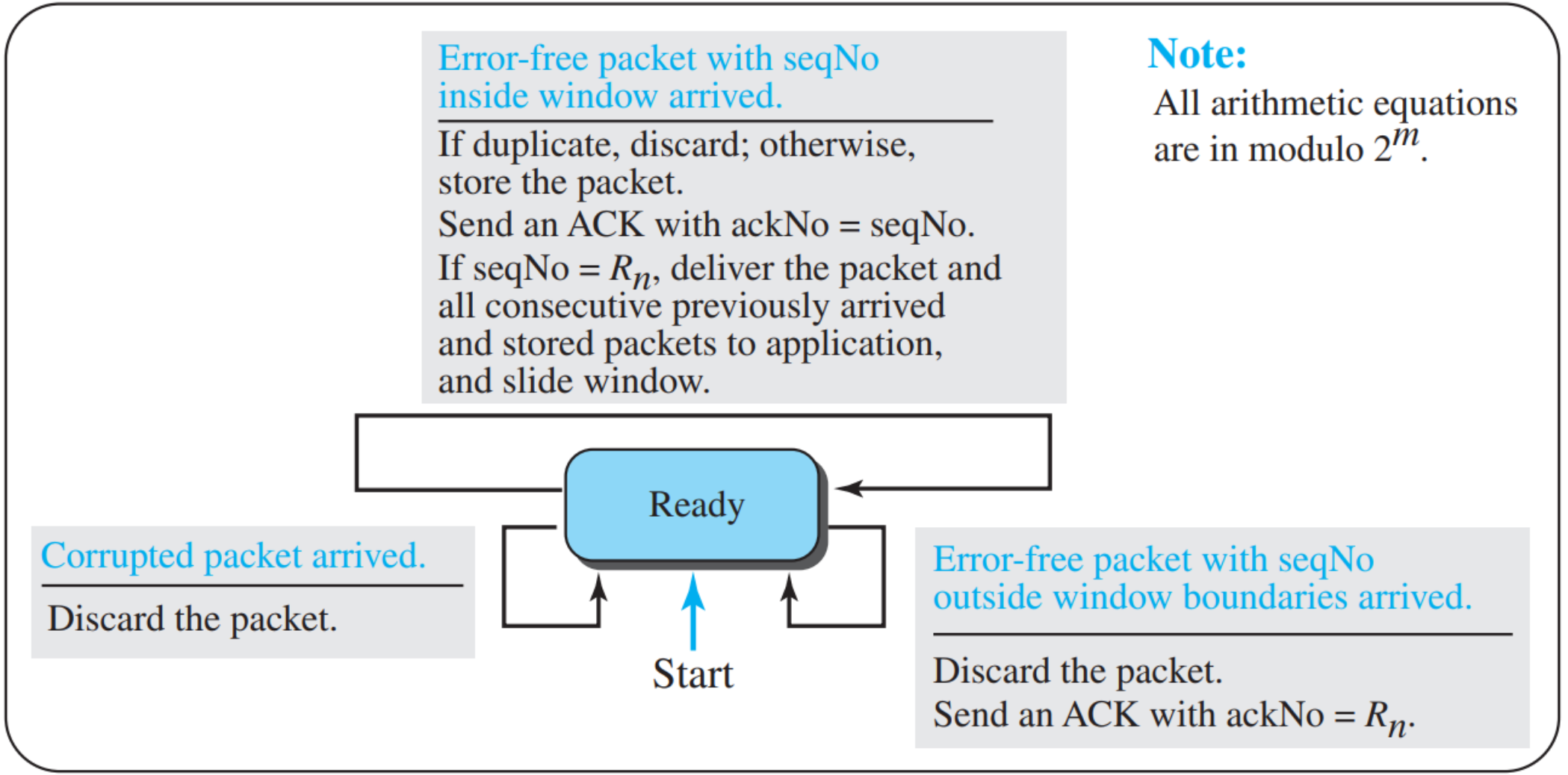


Figure 23.33 *Receive window for Selective-Repeat protocol*




Sender
Time-out.
Resend all outstanding packets in window.
Reset the timer.
Request came from process.
Make a packet (seqNo = S_n).
Store a copy and send the packet.
Start the timer for this packet.
Set $S_n = S_n + 1$.
Window full ($S_n = S_f + S_{size}$)?
[false]
[true]
Time-out.
Resend all outstanding packets in window.
Reset the timer.
Start
Ready
Blocking
[true]
[false]
Window slides?
A corrupted ACK or an ACK about a non-outstanding packet arrived.
Discard it.
A corrupted ACK or an ACK about a non-outstanding packet arrived.
Discard it.
An error-free ACK arrived that acknowledges one of the outstanding packets.
Mark the corresponding packet.
If ackNo = S_f, slide the window over all consecutive acknowledged packets.
If there are outstanding packets, restart the timer. Otherwise, stop the timer.
Note:
All arithmetic equations are in modulo 2^m.

Receiver


Error-free packet with seqNo inside window arrived.
If duplicate, discard; otherwise, store the packet.
Send an ACK with ackNo = seqNo.
If seqNo = R_n, deliver the packet and all consecutive previously arrived and stored packets to application, and slide window.
Note:
All arithmetic equations are in modulo 2^m.
Ready
Start
Corrupted packet arrived.
Discard the packet.
Error-free packet with seqNo outside window boundaries arrived.
Discard the packet.
Send an ACK with ackNo = R_n.

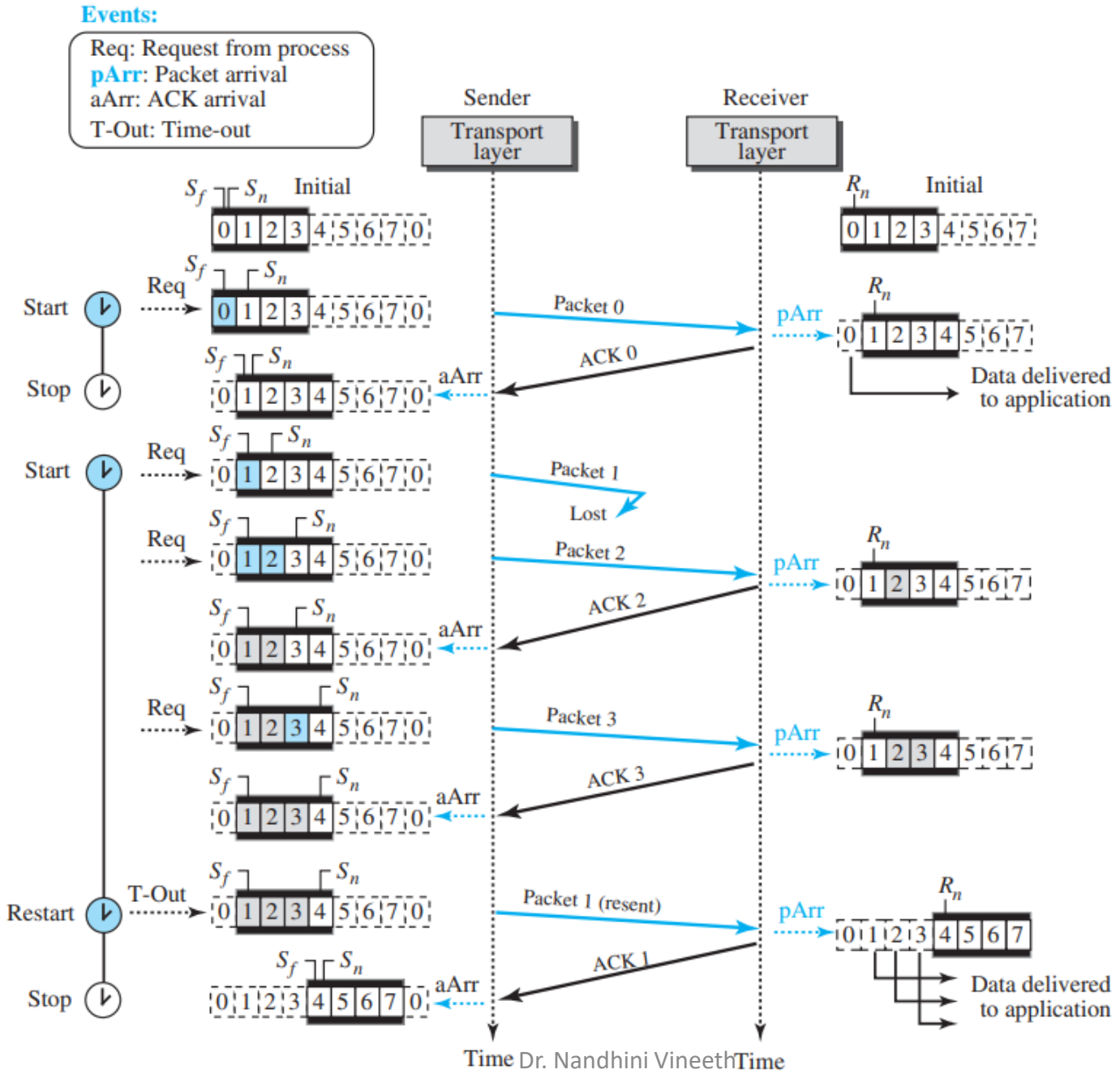
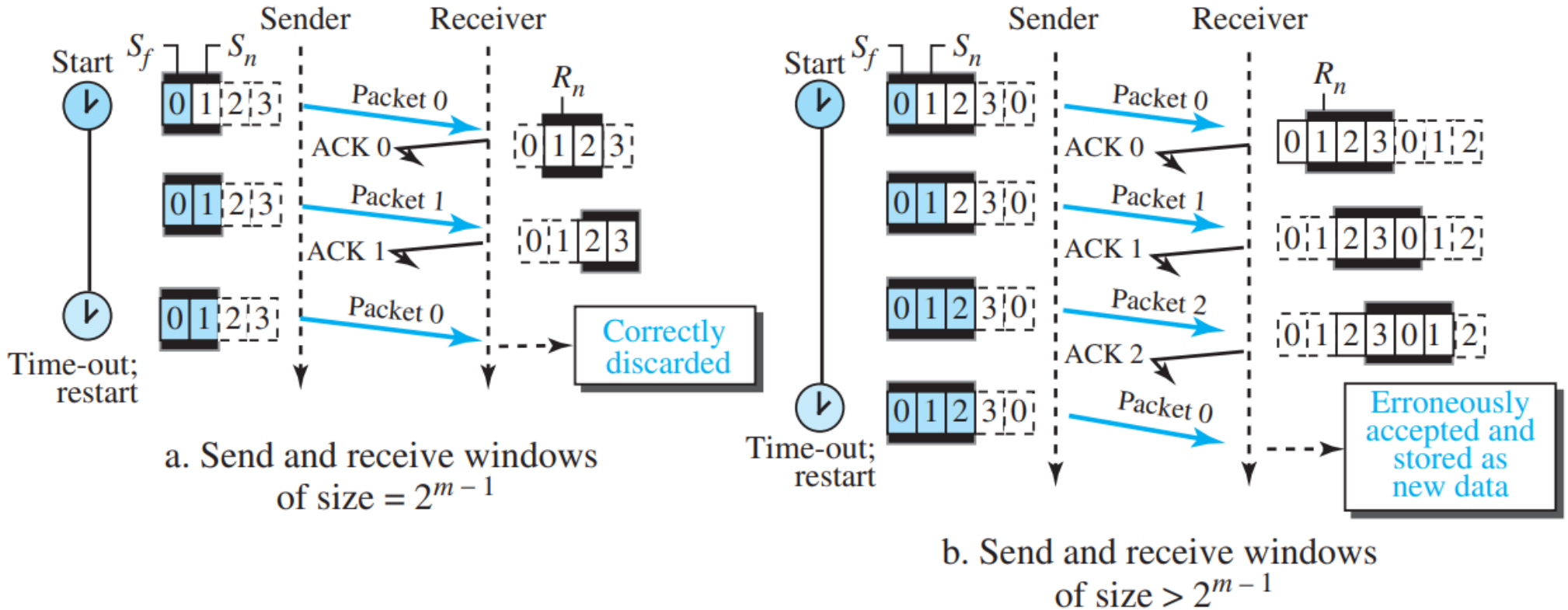
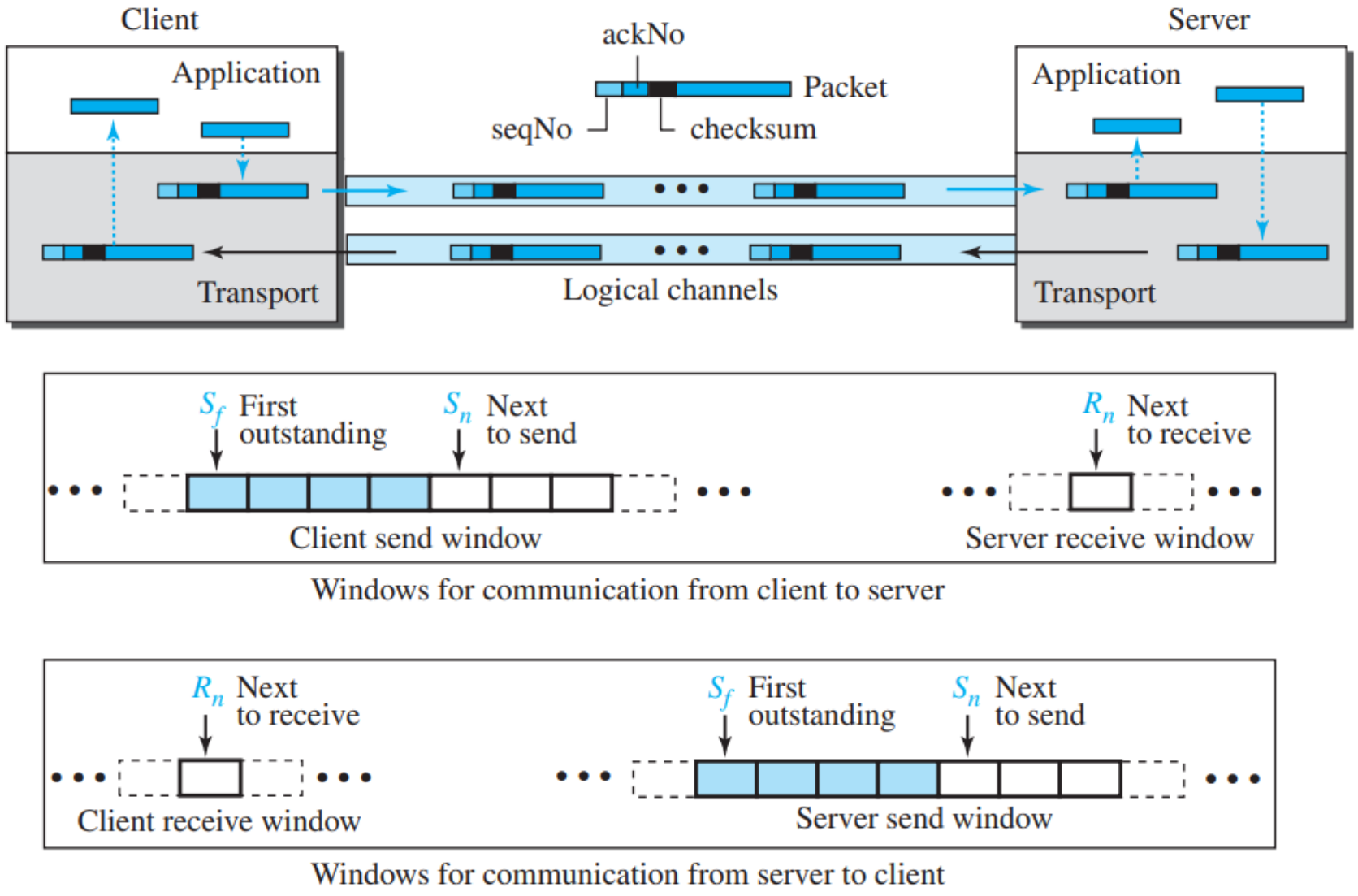

Events:
Req: Request from process
pArr: Packet arrival
aArr: ACK arrival
T-Out: Time-out
Sender
Transport layer
Receiver
Transport layer
Initial
Start
Stop
Req
Packet 0
pArr
ACK 0
aArr
Data delivered to application
Packet 1
Lost
Packet 2
ACK 2
Packet 3
ACK 3
T-Out
Restart
Packet 1 (resent)
ACK 1
Time
Time

Figure 23.36 *Selective-Repeat, window size*

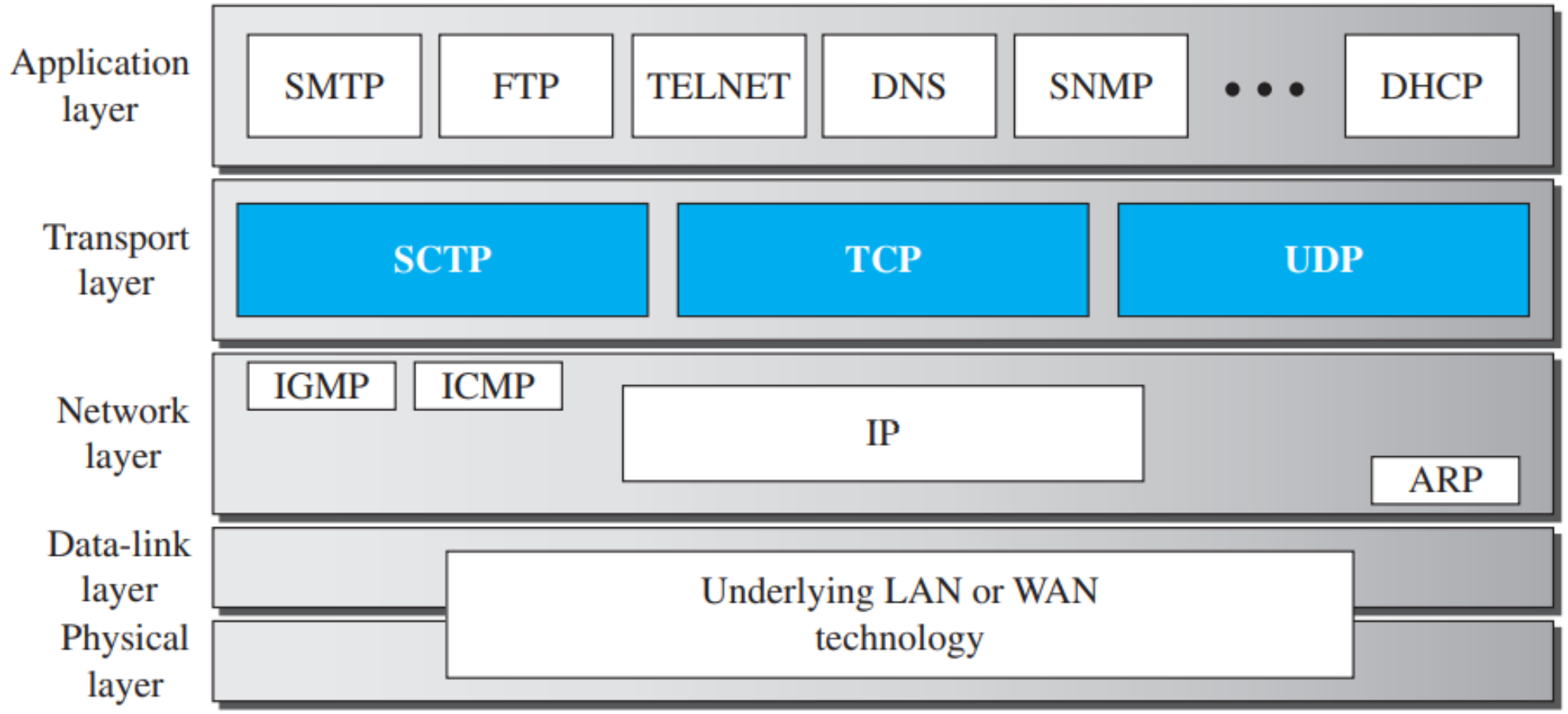


Bidirectional Protocols: Piggybacking

Figure 23.37 *Design of piggybacking in Go-Back-N*



Position of transport-layer protocols in the TCP/IP protocol suite



Services

- Each protocol provides a different type of service and should be used appropriately.
- **UDP**
- UDP is an unreliable connectionless transport-layer protocol used for its simplicity and
- efficiency in applications where error control can be provided by the application-layer
- process.
- **TCP**
- TCP is a reliable connection-oriented protocol that can be used in any application
- where reliability is important.
- SCTP
- SCTP is a new transport-layer protocol that combines the features of UDP and TCP.

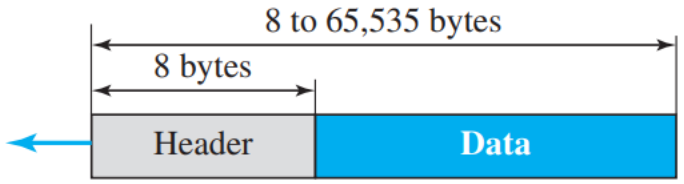
Port Numbers

Port	*Protocol*	*UDP*	*TCP*	*SCTP*	*Description*
7	Echo	√	√	√	Echoes back a received datagram
9	Discard	√	√	√	Discards any datagram that is received
11	Users	√	√	√	Active users
13	Daytime	√	√	√	Returns the date and the time
17	Quote	√	√	√	Returns a quote of the day
19	Chargen	√	√	√	Returns a string of characters
20	FTP-data		√	√	File Transfer Protocol
21	FTP-21		√	√	File Transfer Protocol
23	TELNET		√	√	Terminal Network
25	SMTP		√	√	Simple Mail Transfer Protocol
53	DNS	√	√	√	Domain Name Service
67	DHCP	√	√	√	Dynamic Host Configuration Protocol
69	TFTP	√	√	√	Trivial File Transfer Protocol
80	HTTP		√	√	HyperText Transfer Protocol
111	RPC	√	√	√	Remote Procedure Call
123	NTP	√	√	√	Network Time Protocol
161	SNMP-server	√			Simple Network Management Protocol
162	SNMP-client	√			Simple Network Management Protocol

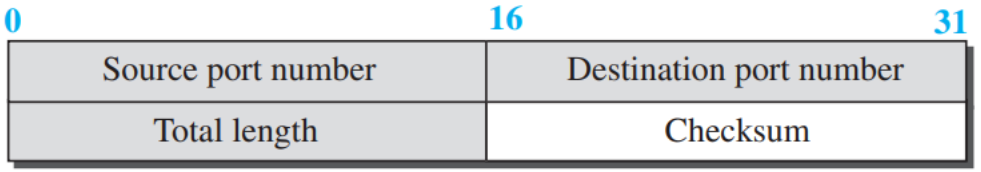
USER DATAGRAM PROTOCOL

- UDP is a very simple protocol using a minimum of overhead.
- Suits a small message where reliability is not a concern
- much less interaction between the sender and receiver than using TCP.

Figure 24.2 *User datagram packet format*



a. UDP user datagram



b. Header format

Example 24.1

The following is the content of a UDP header in hexadecimal format.

CB84000D001C001C

a. What is the source port number?
b. What is the destination port number?
c. What is the total length of the user datagram?
d. What is the length of the data?
e. Is the packet directed from a client to a server or vice versa?
f. What is the client process?

Solution

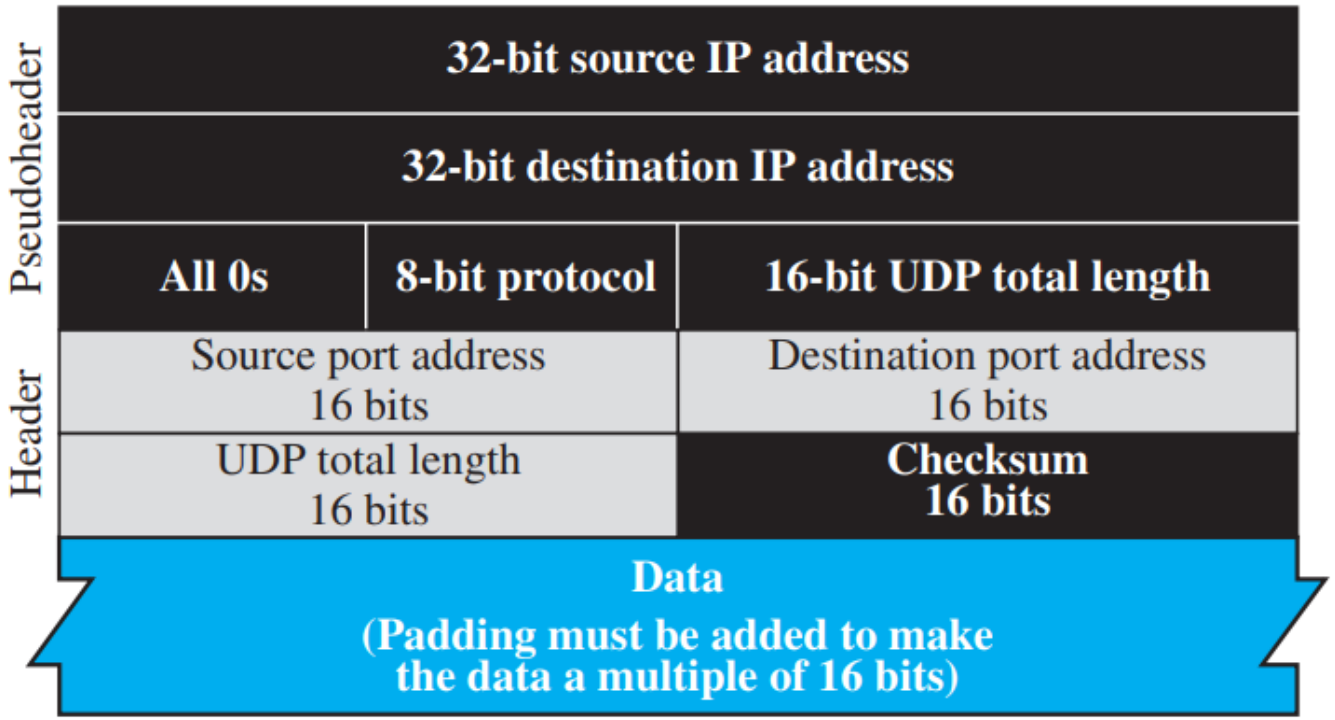
a. The source port number is the first four hexadecimal digits $(CB84)_{16}$, which means that the source port number is 52100.
b. The destination port number is the second four hexadecimal digits $(000D)_{16}$, which means that the destination port number is 13.
c. The third four hexadecimal digits $(001C)_{16}$ define the length of the whole UDP packet as 28 bytes.
d. The length of the data is the length of the whole packet minus the length of the header, or $28 - 8 = 20$ bytes.
e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.
f. The client process is the Daytime (see Table 24.1).

UDP Services

- Process-to-Process Communication
  - process-to-process communication using socket addresses
- Connectionless Services
- Flow Control, Error Control, Congestion Control
  - No FC and EC
- Checksum
  - UDP checksum calculation includes three sections: a pseudoheader, the UDP header, and the data coming from the application layer. The pseudoheader is the part of the header of the IP packet in which the user datagram is to be encapsulated with some fields filled with 0s

PSEUDOHEADER FOR CHECKSUM CALCULATION

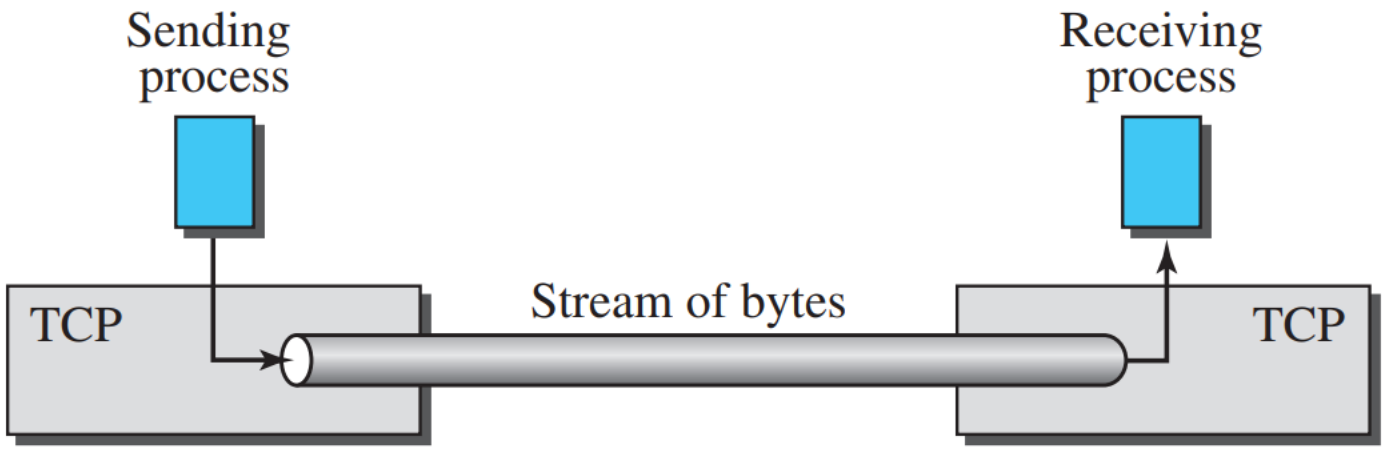
Figure 24.3 *Pseudoheader for checksum calculation*



- **Optional Inclusion of Checksum**
- The sender of a UDP packet can choose not to calculate the checksum. In this case, the checksum field is filled with all 0s before being sent.
- the checksum is changed to all 1s before the packet is sent. In other words, the sender complements the sum two times.

TRANSMISSION CONTROL PROTOCOL

Figure 24.4 *Stream delivery*



TCP Services

- Process-to-Process Communication
- Stream Delivery Service
- Full-Duplex Communication
- Multiplexing and Demultiplexing
- Connection-Oriented Service
- Reliable Service

Sending and Receiving Buffers

Figure 24.5 *Sending and receiving buffers*

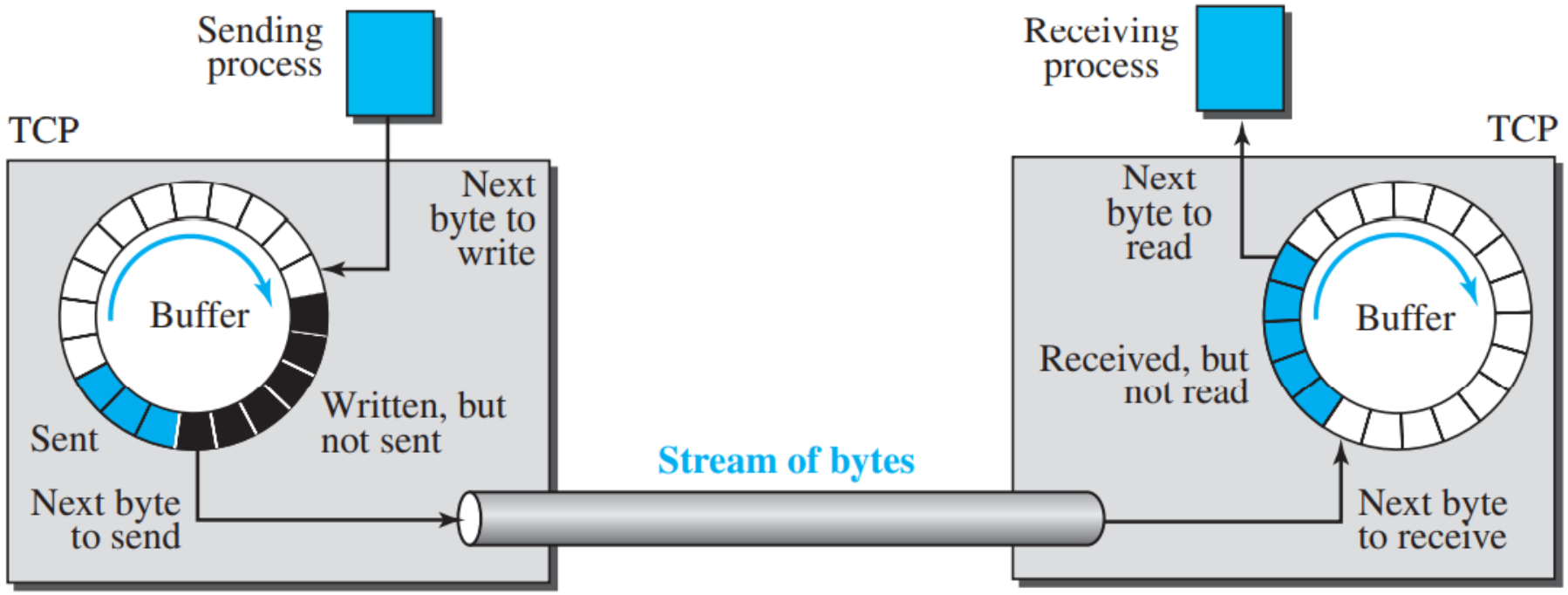


Figure 24.6 *TCP segments*

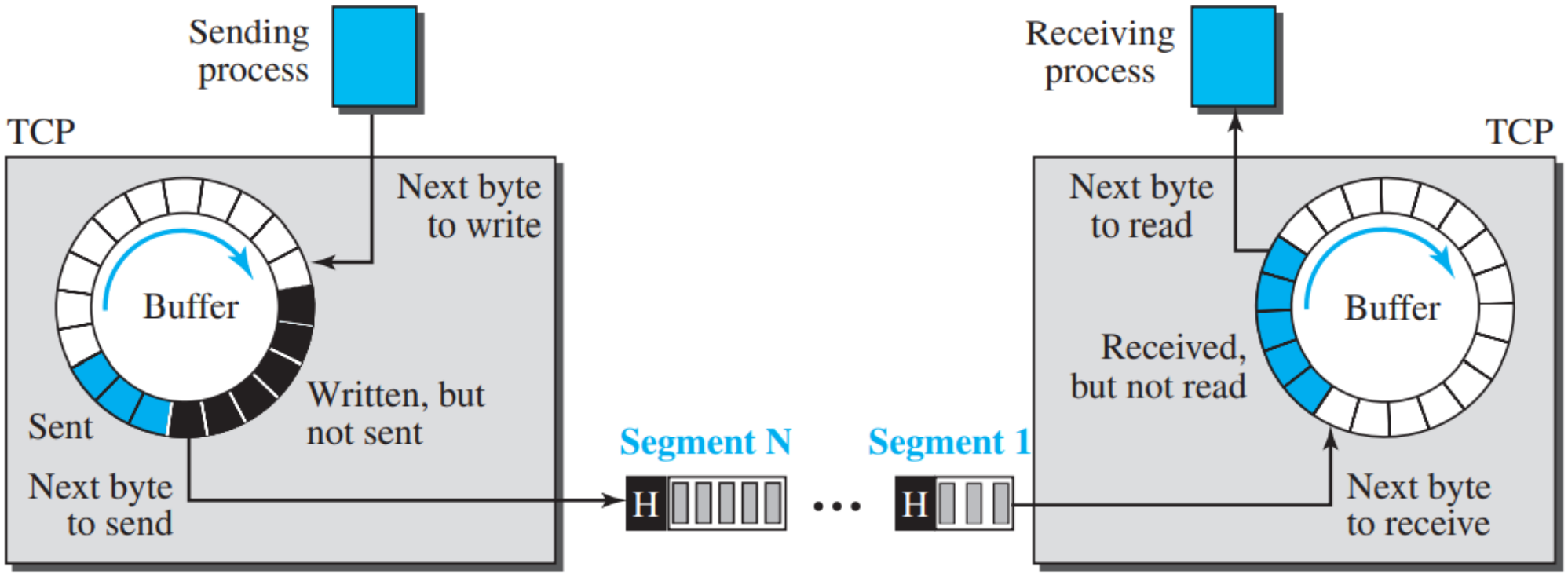


Figure 24.7 *TCP segment format*

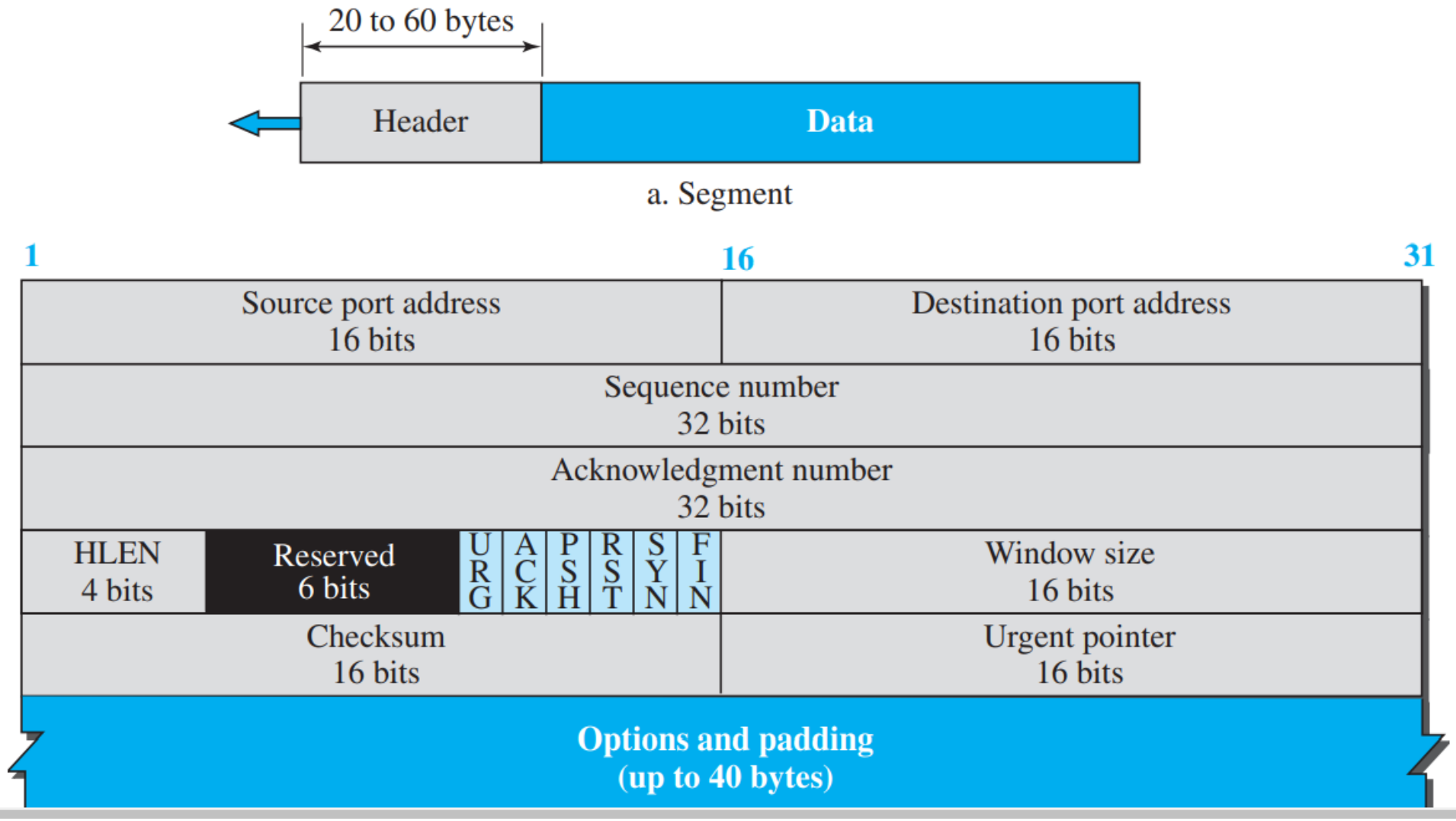
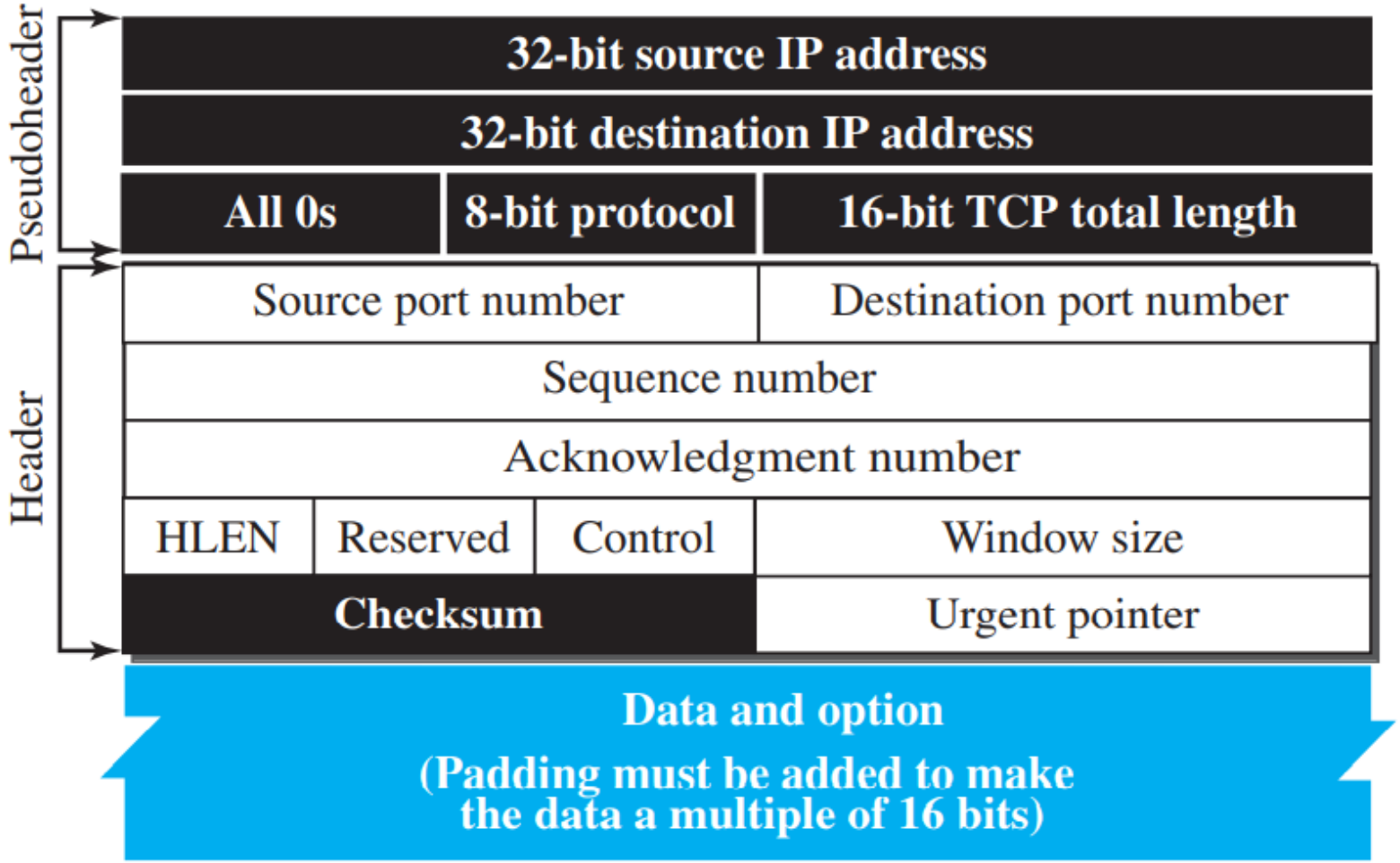


Figure 24.9 *Pseudoheader added to the TCP datagram*



TCI

Figure 24.7 *TCP segment format*

a. Segment

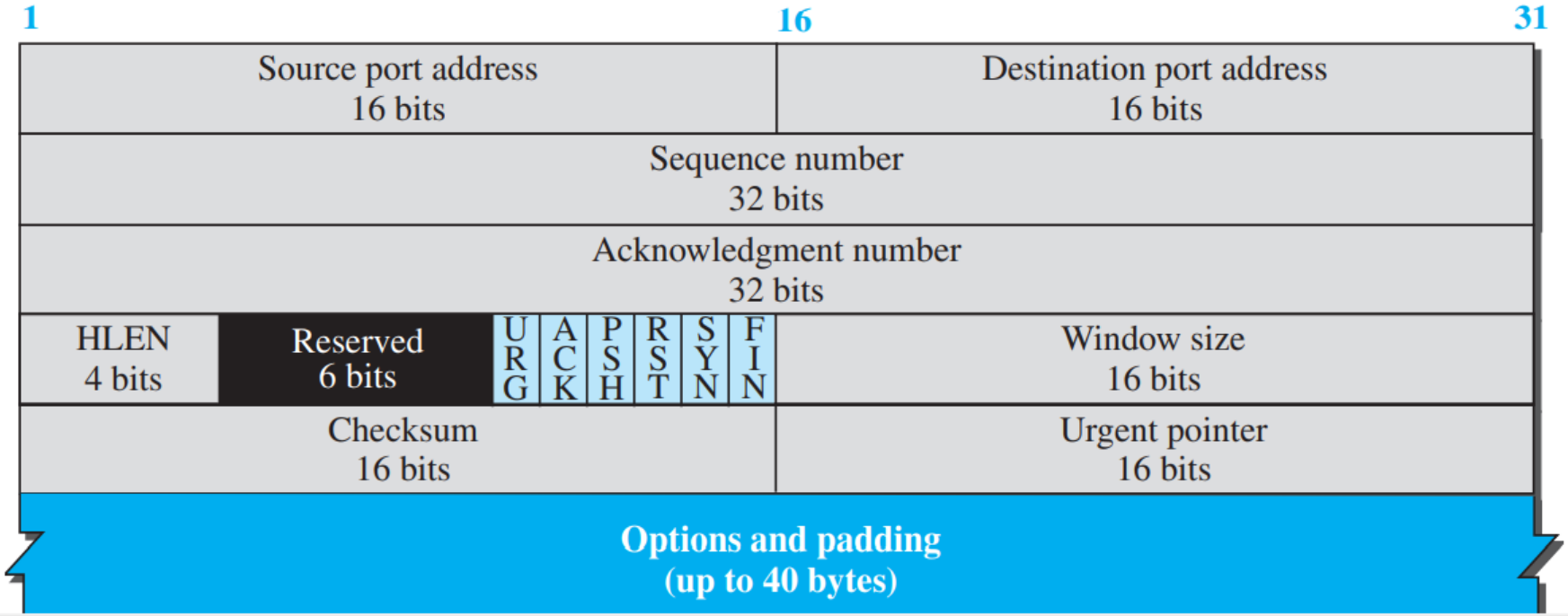
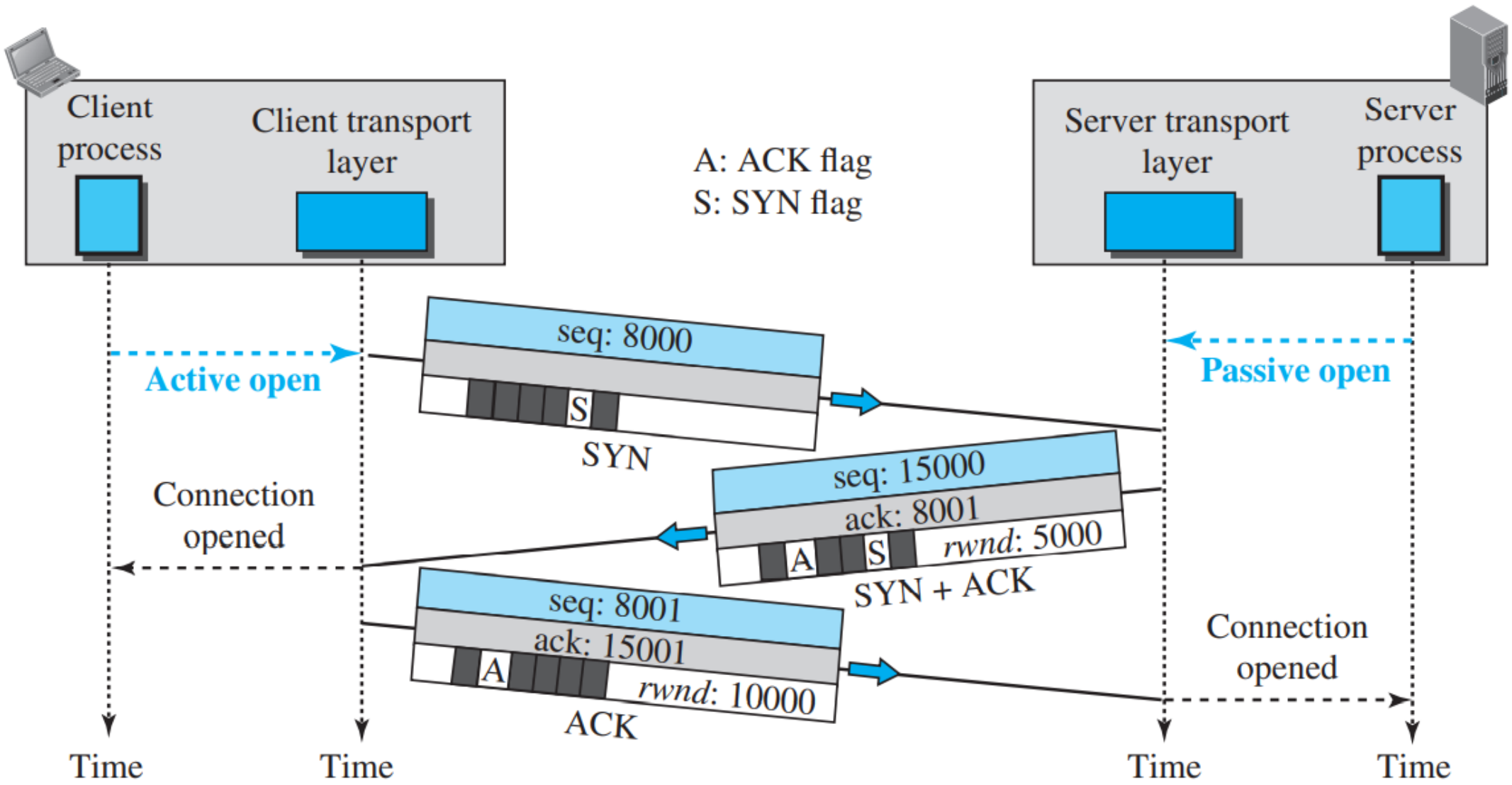


Figure 24.10 *Connection establishment using three-way handshaking*



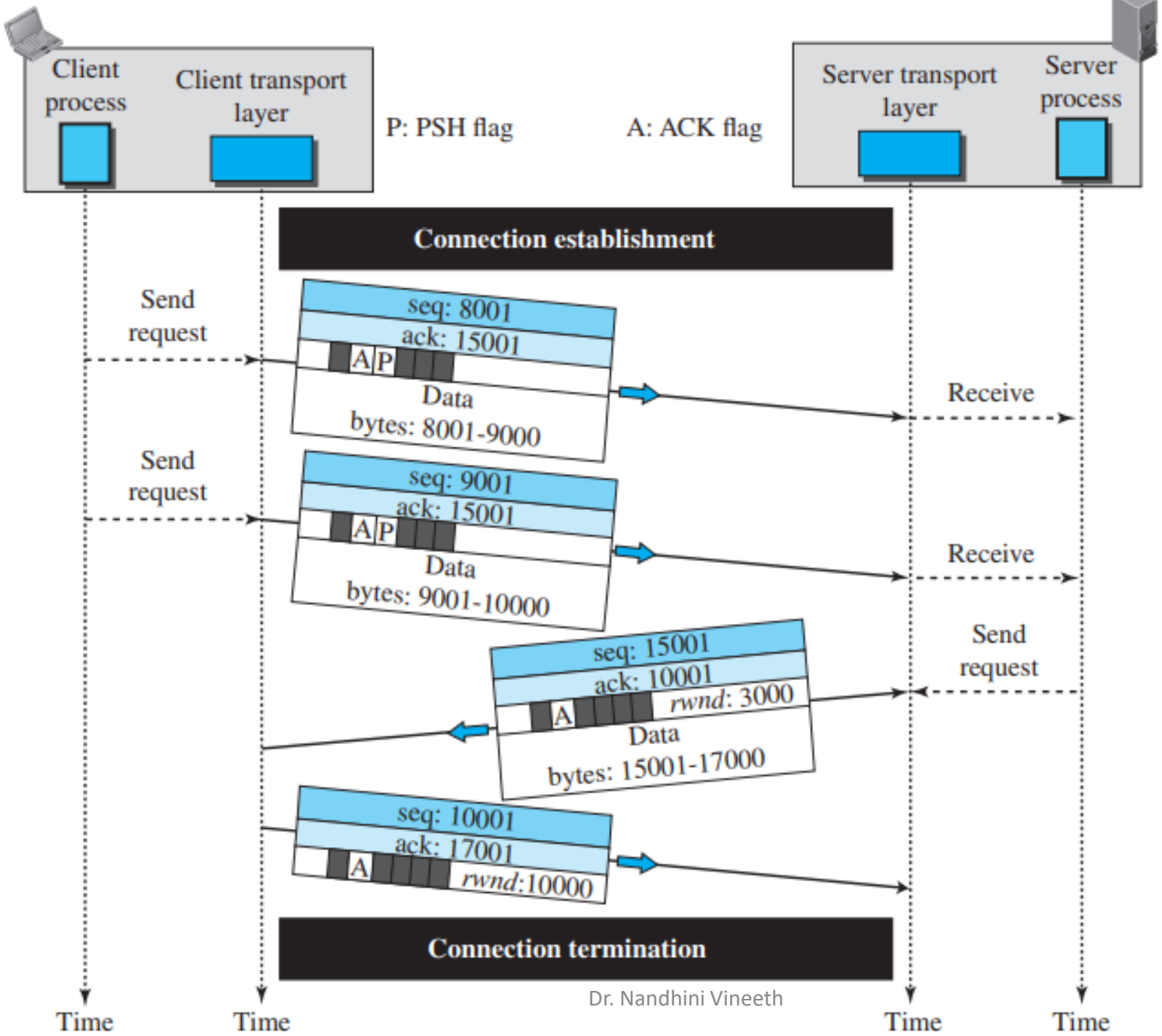

Client process
Client transport layer
P: PSH flag
A: ACK flag
Server transport layer
Server process
Connection establishment
Send request
seq: 8001
ack: 15001
A P
Data
bytes: 8001-9000
Receive
Send request
seq: 9001
ack: 15001
A P
Data
bytes: 9001-10000
Receive
Send request
seq: 15001
ack: 10001
rwnd: 3000
A
Data
bytes: 15001-17000
seq: 10001
ack: 17001
A
rwnd:10000
Connection termination
Time
Time
Time
Time

Figure 24.12 *Connection termination using three-way handshaking*

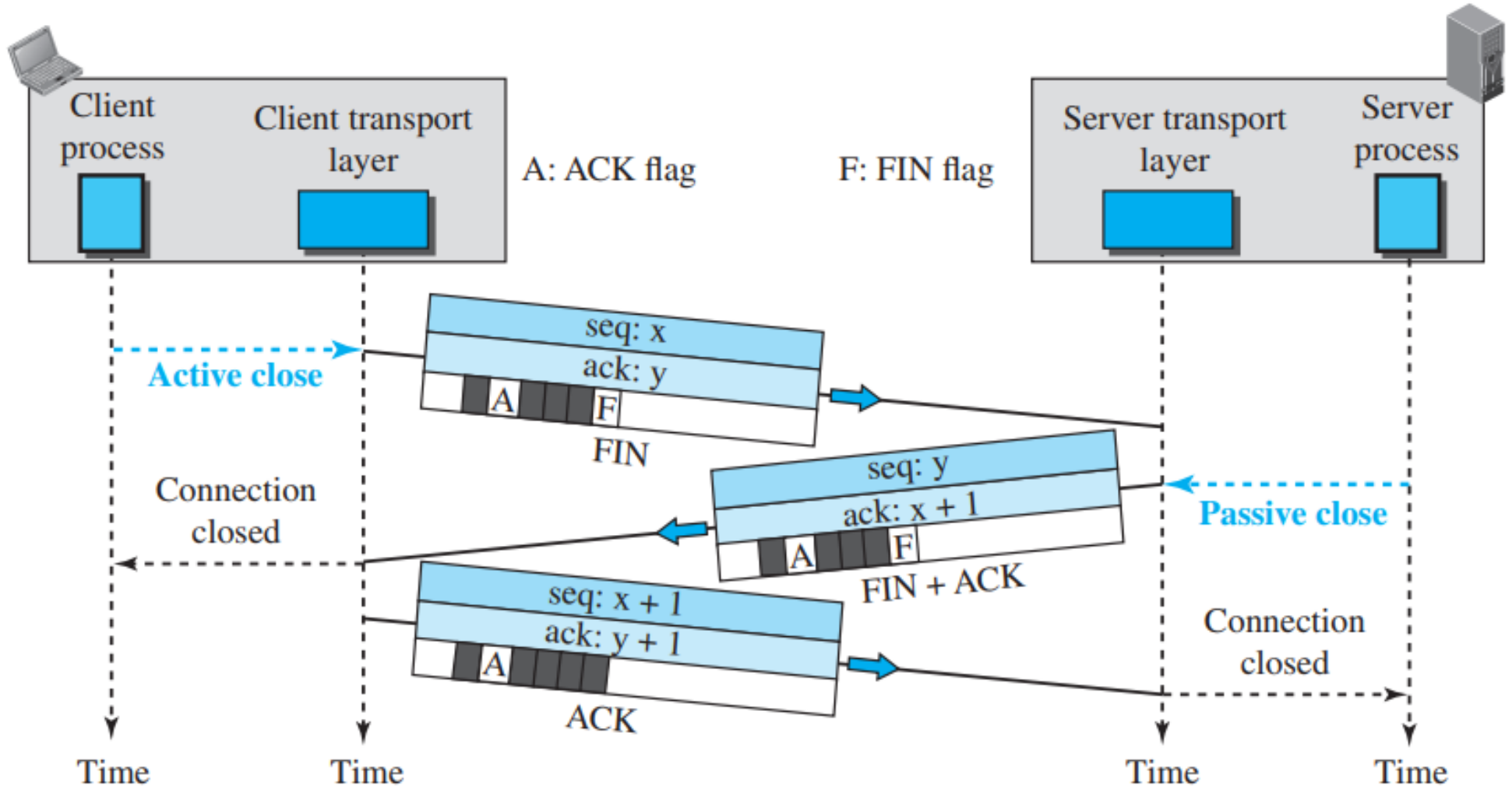


Figure 24.13 *Half-close*

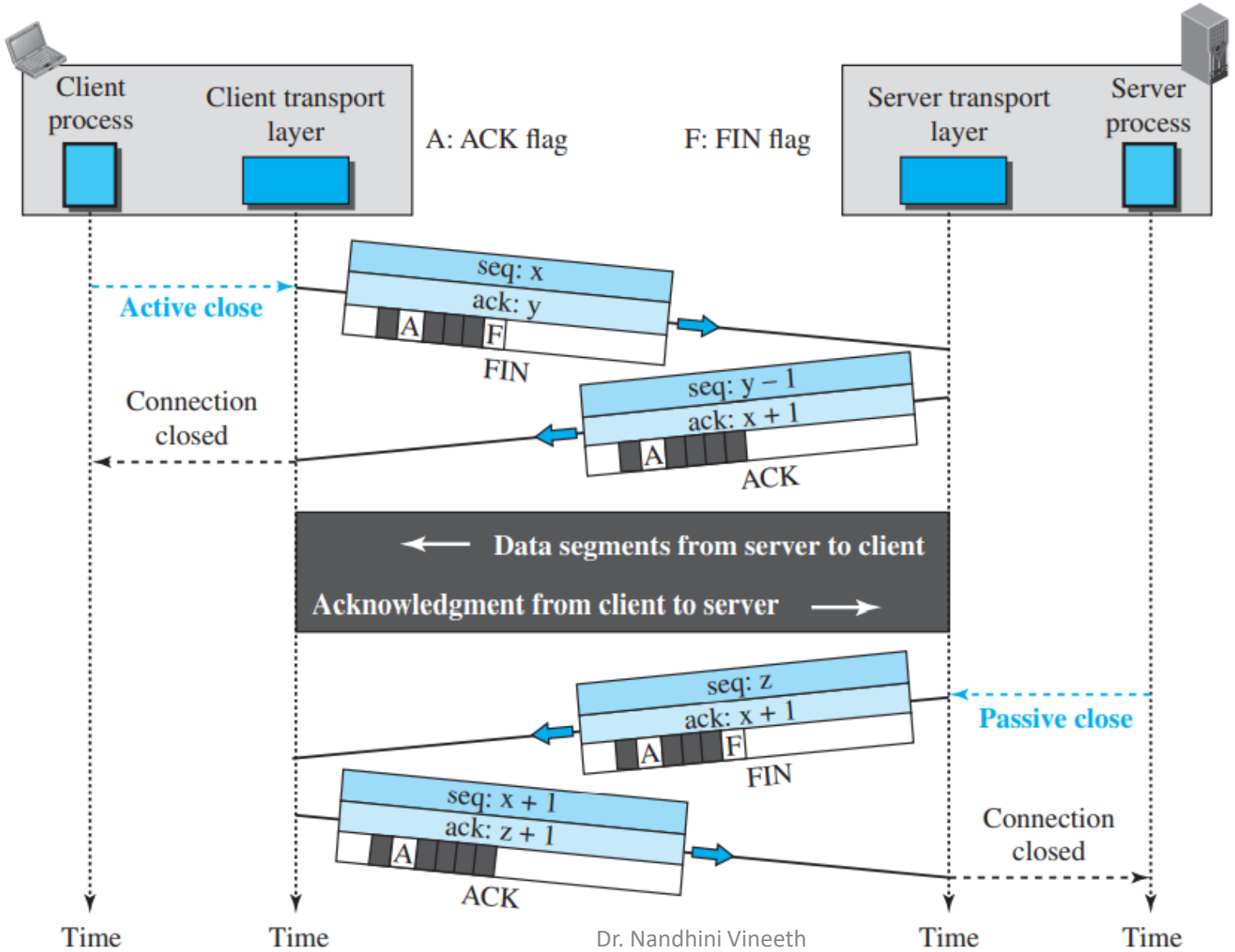


Figure 24.14 *State transition diagram*

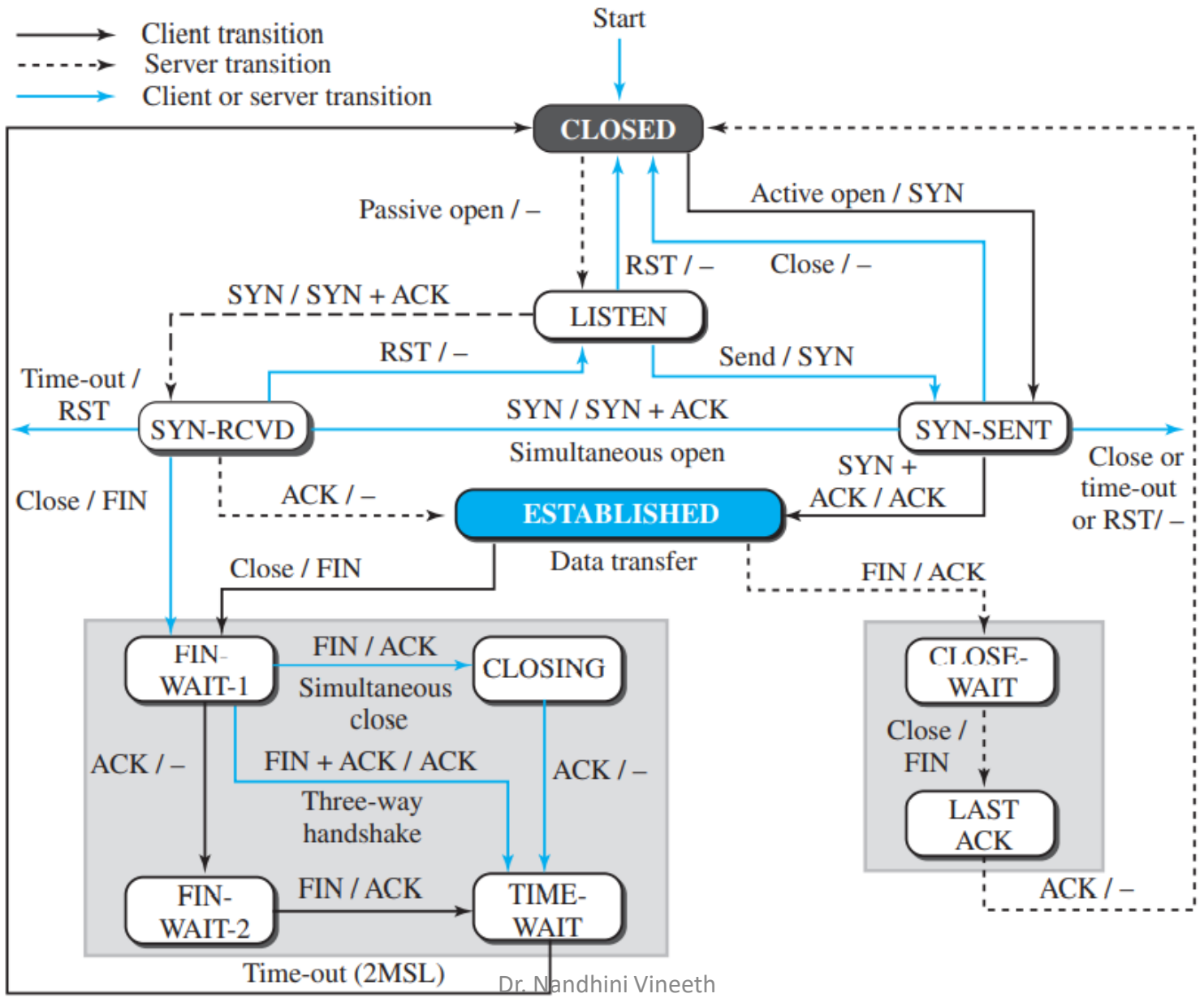


Table 24.2 *States for TCP*

State	*Description*
CLOSED	No connection exists
LISTEN	Passive open received; waiting for SYN
SYN-SENT	SYN sent; waiting for ACK
SYN-RCVD	SYN + ACK sent; waiting for ACK
ESTABLISHED	Connection established; data transfer in progress
FIN-WAIT-1	First FIN sent; waiting for ACK
FIN-WAIT-2	ACK to first FIN received; waiting for second FIN
CLOSE-WAIT	First FIN received, ACK sent; waiting for application to close
TIME-WAIT	Second FIN received, ACK sent; waiting for 2MSL time-out
LAST-ACK	Second FIN sent; waiting for ACK
CLOSING	Both sides decided to close simultaneously

Figure 24.15 *Transition diagram with half-close connection termination*

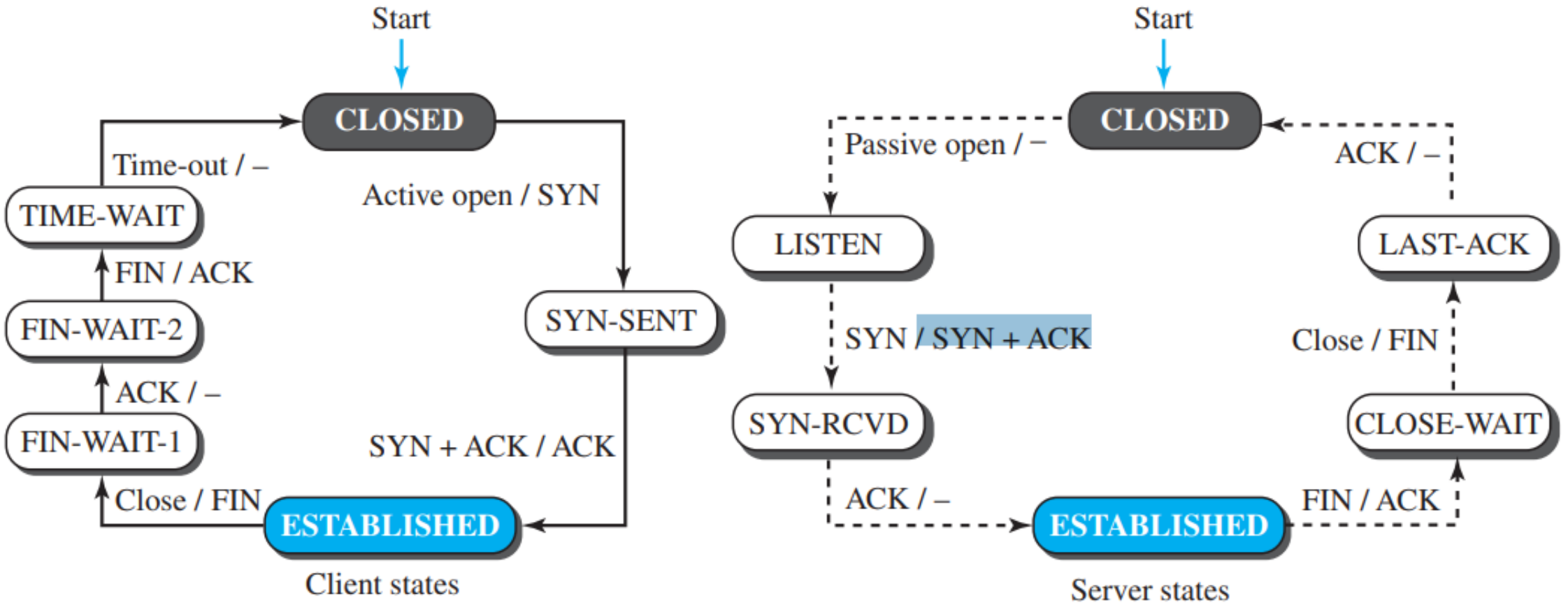
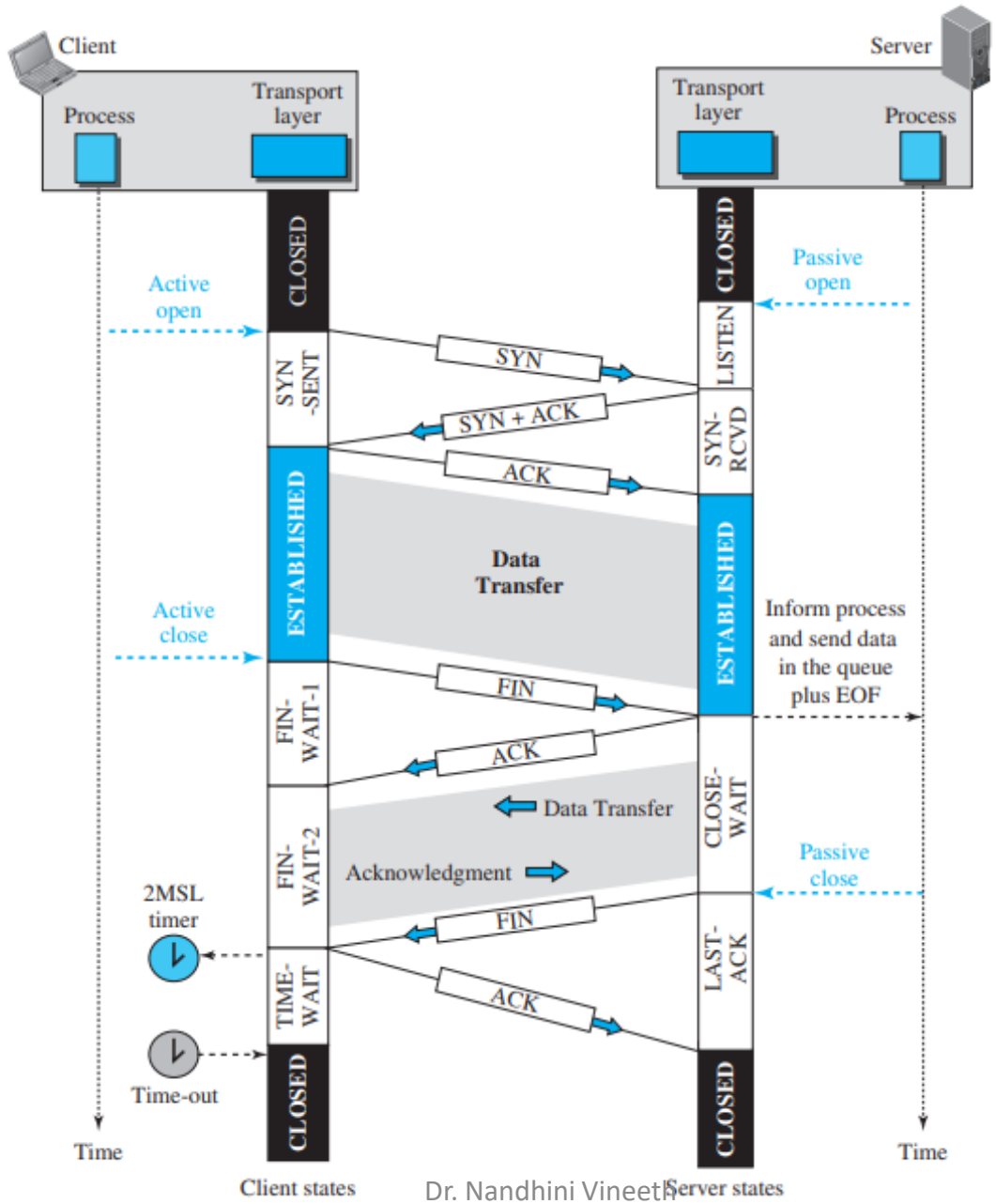
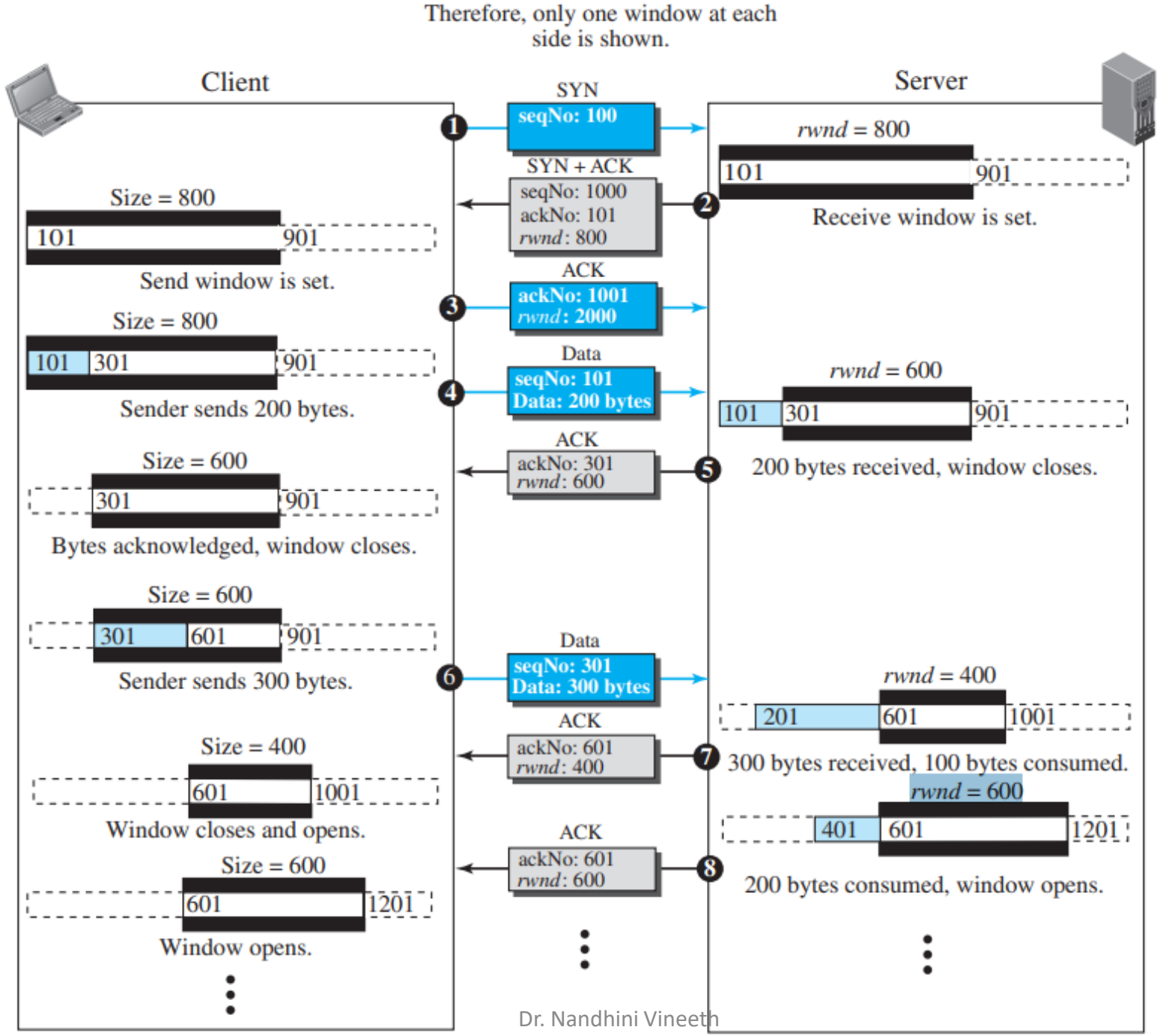


Figure 24.16 *Time-line diagram for a common scenario*




Therefore, only one window at each side is shown.
Client
Server
SYN
seqNo: 100
rwnd = 800
101
901
Receive window is set.
SYN + ACK
seqNo: 1000
ackNo: 101
rwnd: 800
Size = 800
101
901
Send window is set.
ACK
ackNo: 1001
rwnd: 2000
Size = 800
101
301
901
Sender sends 200 bytes.
Data
seqNo: 101
Data: 200 bytes
rwnd = 600
101
301
901
200 bytes received, window closes.
ACK
ackNo: 301
rwnd: 600
Size = 600
301
901
Bytes acknowledged, window closes.
Size = 600
301
601
901
Sender sends 300 bytes.
Data
seqNo: 301
Data: 300 bytes
rwnd = 400
201
601
1001
300 bytes received, 100 bytes consumed.
ACK
ackNo: 601
rwnd: 400
Size = 400
601
1001
Window closes and opens.
rwnd = 600
401
601
1201
200 bytes consumed, window opens.
ACK
ackNo: 601
rwnd: 600
Size = 600
601
1201
Window opens.

Windows in TCP

- Sending window – Diff b/w TCP and SR

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.

2. 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.

3. 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.**

Windows in TCP

- Receive Window
  - 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 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).

Figure 24.17 *Send window in TCP*

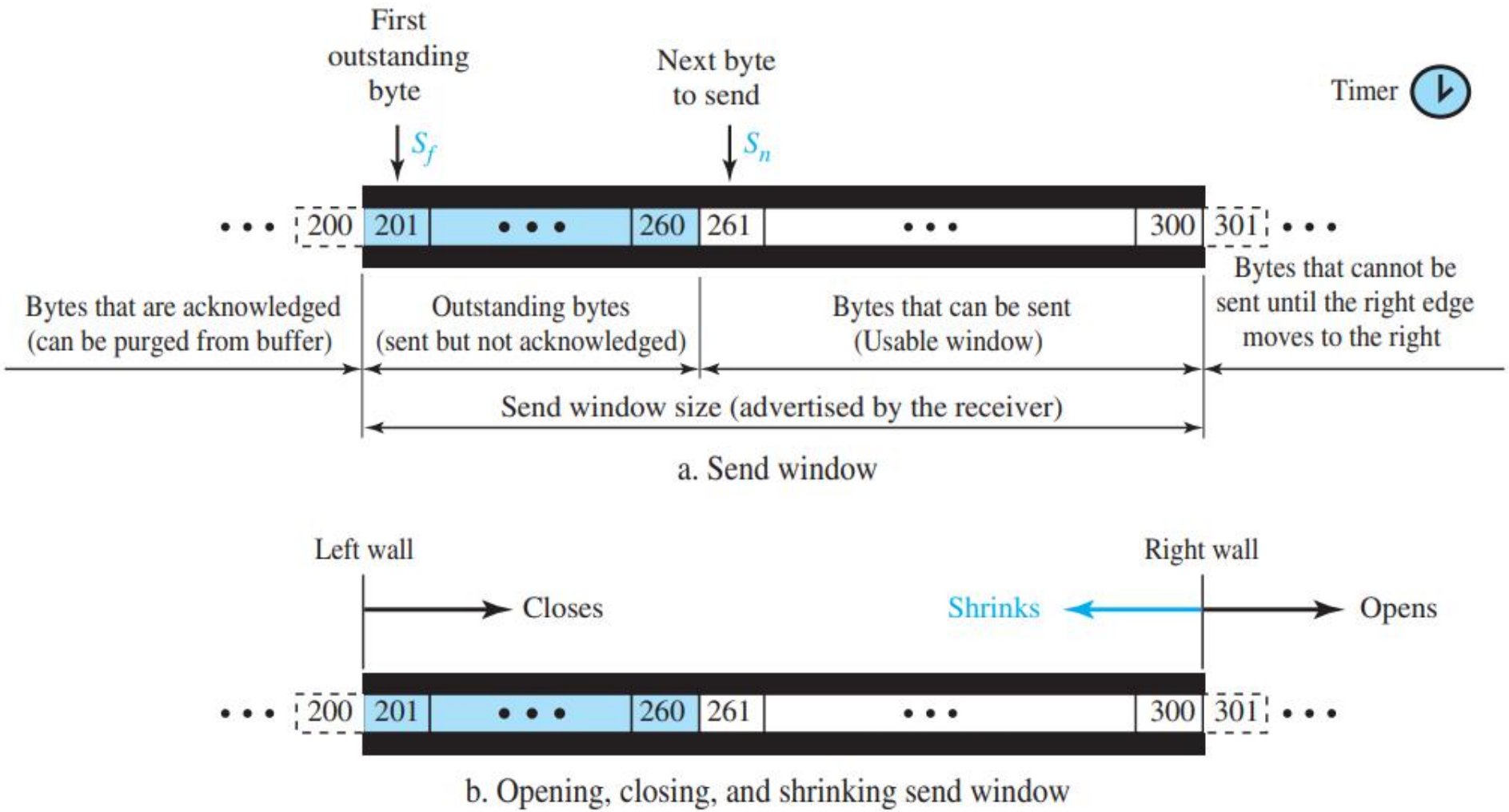
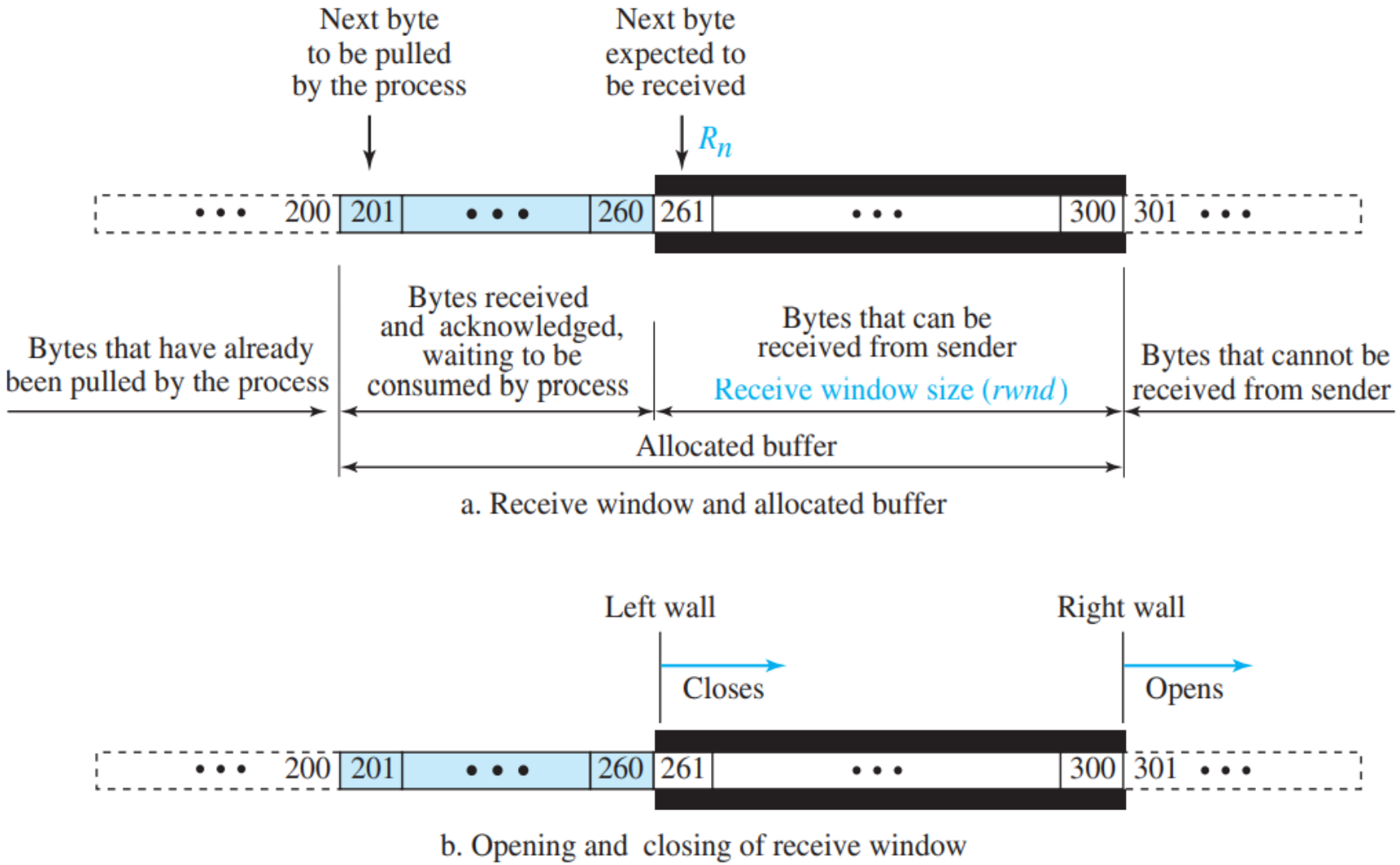


Figure 24.18 *Receive window in TCP*

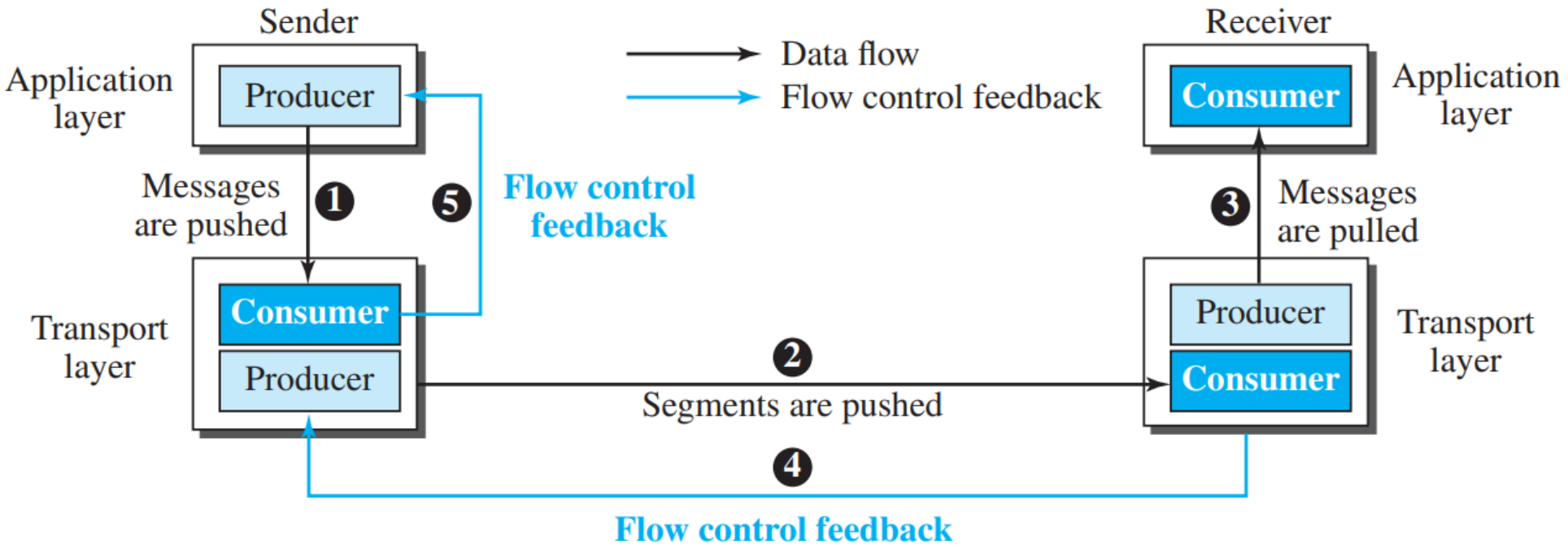


a. Receive window and allocated buffer

b. Opening and closing of receive window

Flow Control

Figure 24.19 *Data flow and flow control feedbacks in TCP*



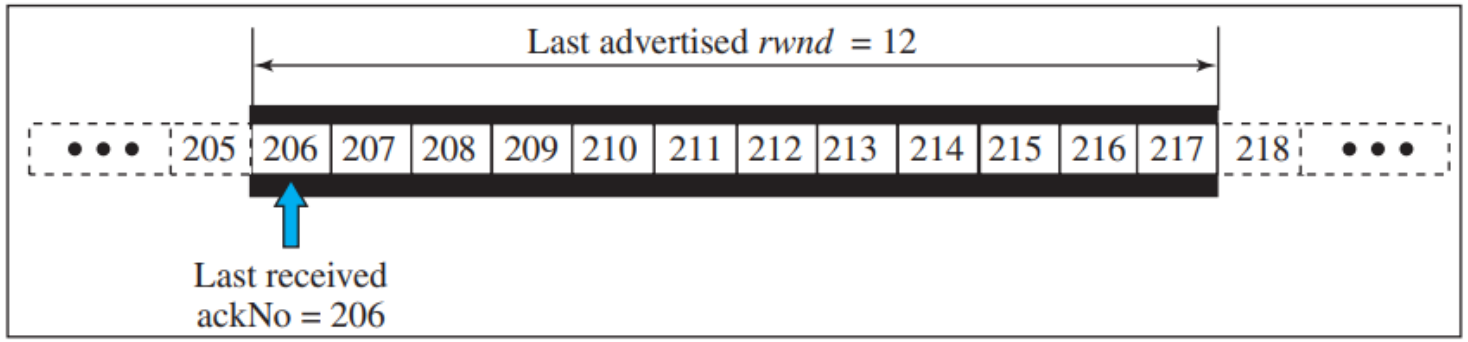
Opening and Closing Windows

- 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.
- We assume that it does not shrink (the right wall does not move to the left).
- 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

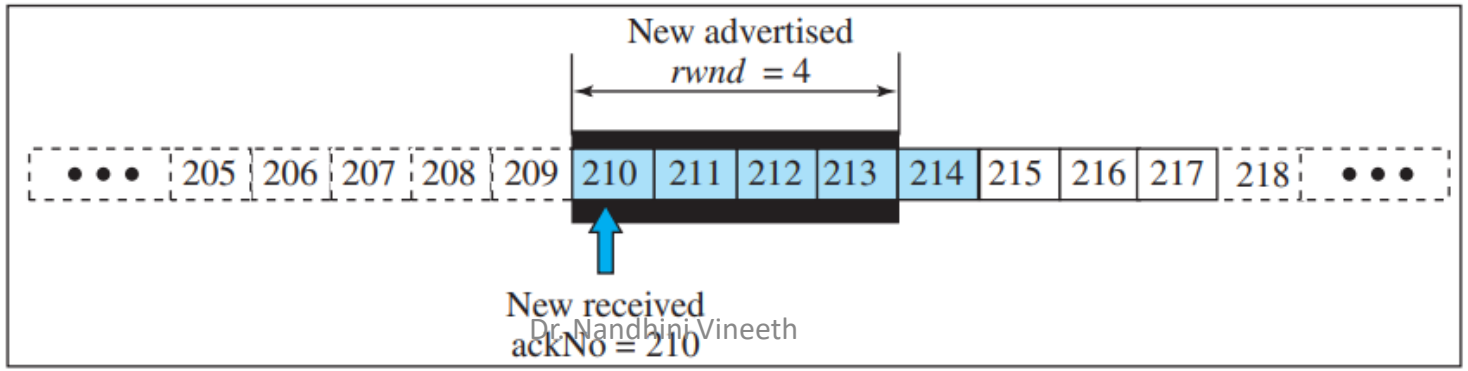
Shrinking of Windows

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. **new ackNo + new rwnd ≥ last ackNo + last rwnd**

Figure 24.21 *Example 24.8*



a. The window after the last advertisement



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

Window Shutdown

- the receiver can temporarily shut down the window by sending a **rwnd of 0**.
- the sender does not actually shrink the size of the window, but **stops sending data** until a new advertisement has arrived.
- **Probing** – 1 byte segment sent to check if the paused connection can be activated

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**.
- Ex. If 1 byte of data is transmitted with 40 bytes of header, inefficient tx results
- The inefficiency is even worse after accounting for the data-link layer and physical-layer overhead. This problem is called the **silly window syndrome.**
- **Syndrome Created by the Sender**
- **Nagle's algorithm** is simple:
- 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 have 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.

Silly Window Syndrome

- **Syndrome Created by the Receiver**
- If receiver application is slow, receiver TL buffer will be almost always full.
- It **advertises a window size of zero**, which means the sender should stop sending data.
- If one byte is pulled, it may send rwnd of 1, inefficient tx results
- Two solutions have been proposed
- **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**. The receiver **waits until there is a decent amount of space in its incoming buffer** before acknowledging the arrived segments. The delayed acknowledgment prevents the sending TCP from sliding its window. After the sending TCP has sent the data in the window, it stops. This kills the syndrome
- Delayed acknowledgment also has another **advantage: it reduces traffic**. **Disadvantage** in that the delayed acknowledgment may result in the sender **unnecessarily retransmitting** the unacknowledged segments.

Error Control

- Error control in TCP is achieved through the use of three simple tools**: checksum, acknowledgment, and time-out.**
- Acknowledgment Type
  - Cumulative Acknowledgment (ACK) - TCP Header ack
  - Selective Acknowledgment (SACK) -
    - in Options used to indicate duplicate packetes.
    - not for regular ack
- Generating Acknowledgments
  - **Rule 1:** Piggybacking – reduces traffic
  - **Rule 2:** the receiver needs to delay sending an ACK segment if there is only one outstanding in-order segment waiting for other ACKs or 500ms whichever is minimum
  - **Rule 3:** there should not be more than two in-order unacknowledged segments at any time
  - **Rule 4:** out-of-order sequence number announcing the sequence number of the next expected segment. Fast retransmission of missing segments
  - **Rule 5:** 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.
  - **Rule 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**.**

Error Control

- **Retransmission**
  - When ACK Timer runs out - Retransmission after **retransmission time-out** (RTO)
    - value of **RTO is dynamic in TCP** and is updated based on the round-trip time **(RTT**) of segments. RTT is the time needed for a segment to reach a destination and for an acknowledgment to be received
  - When 3 duplicate ACKs received for the first byte - Retransmission after Three Duplicate ACK Segments
    - If RTO is large, this is applied. This feature is called **fast retransmission.**
- **Out-of-Order Segments**
  - Stored and marked and not sent to the process.

Figure 24.22 *Simplified FSM for the TCP sender side*

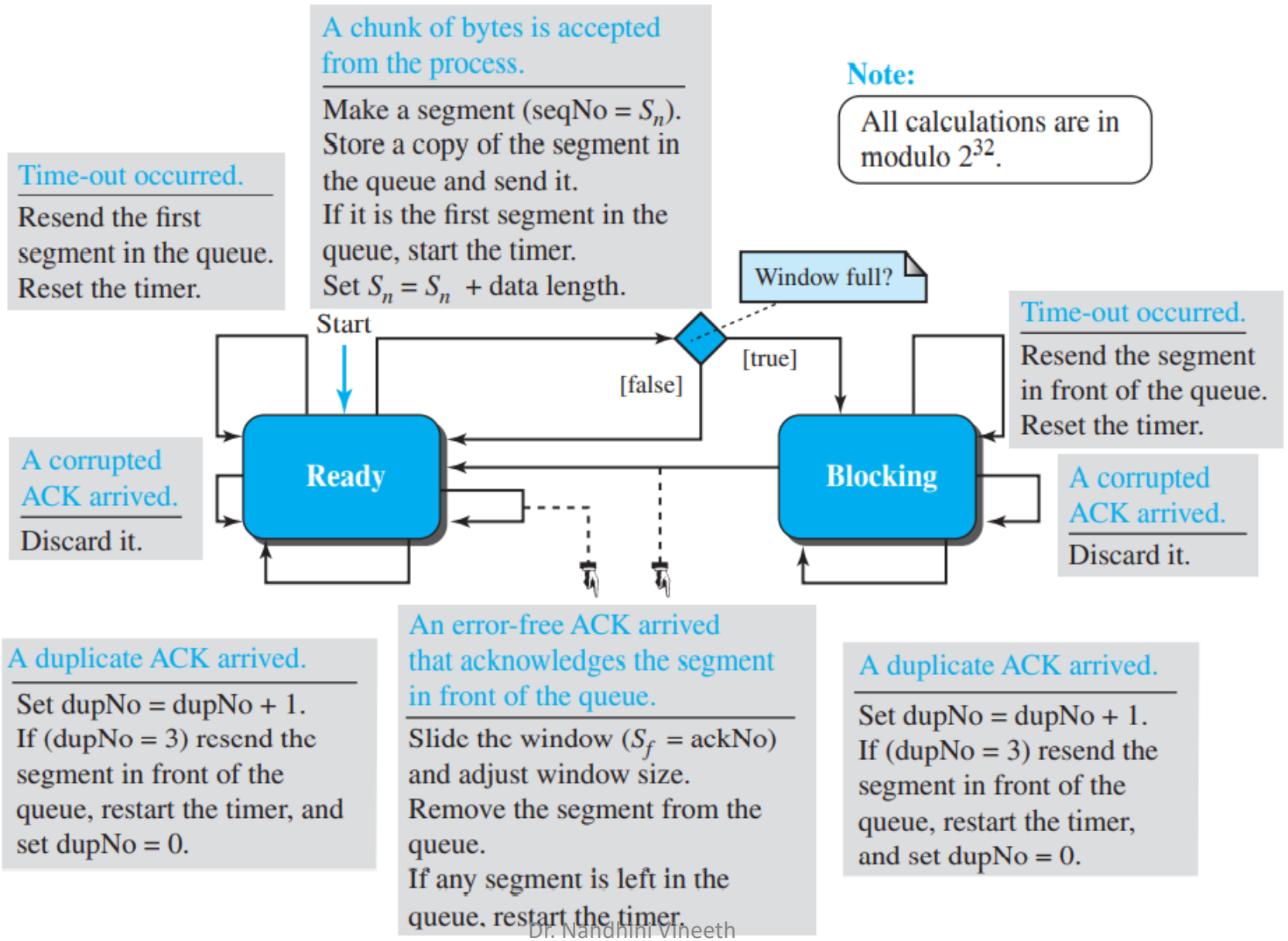


Figure 24.23 *Simplified FSM for the TCP receiver side*

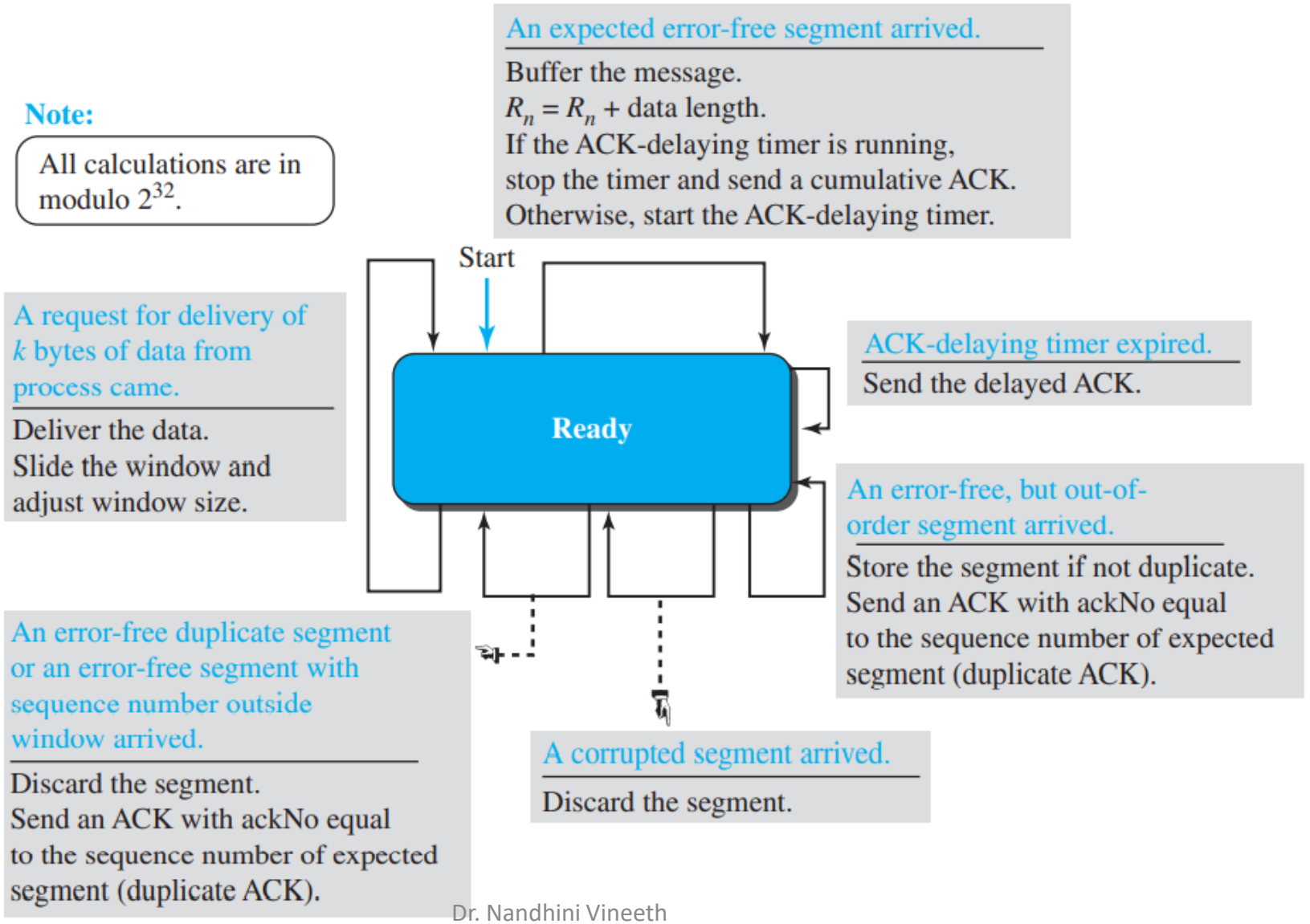


Figure 24.24 *Normal operation*

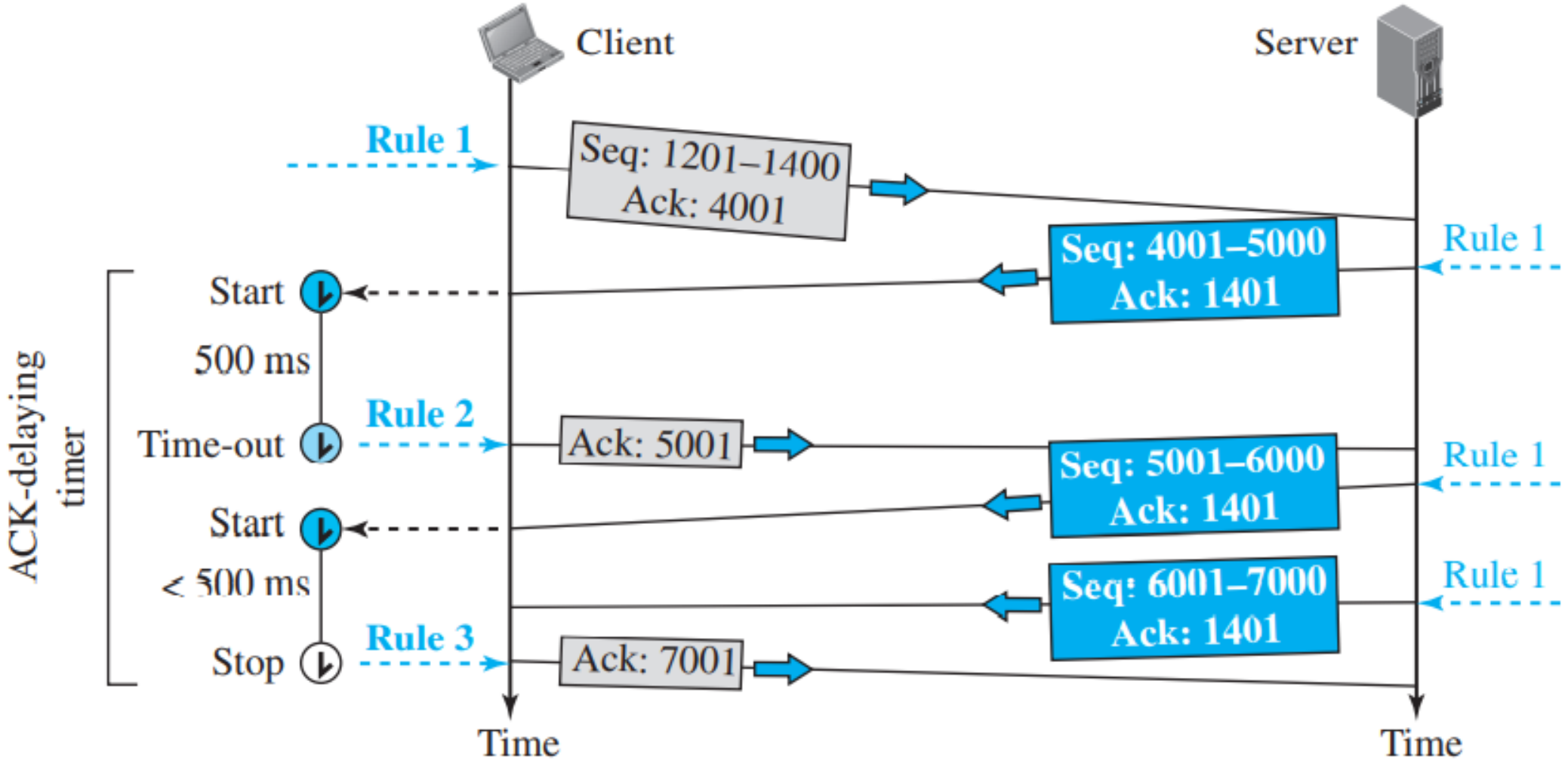


Figure 24.25 *Lost segment*

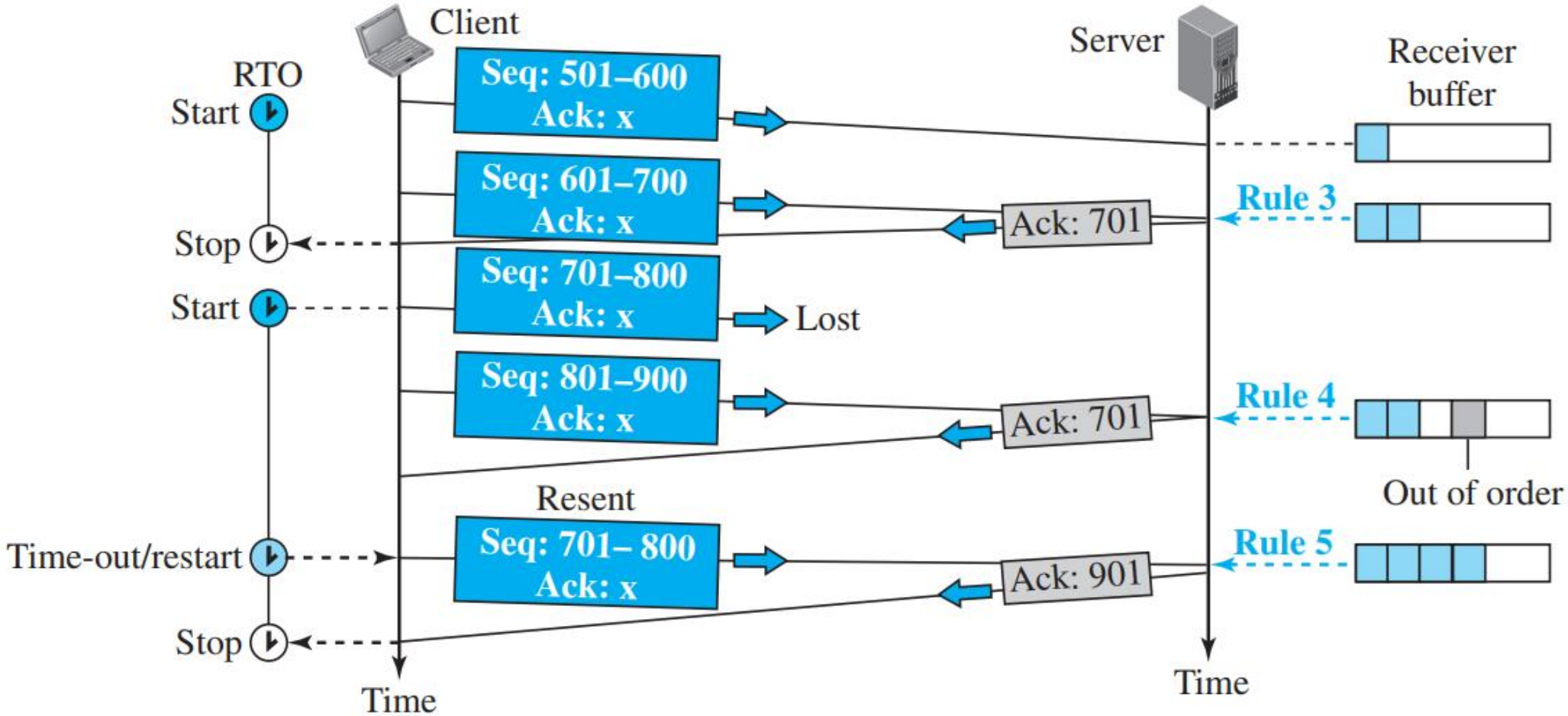
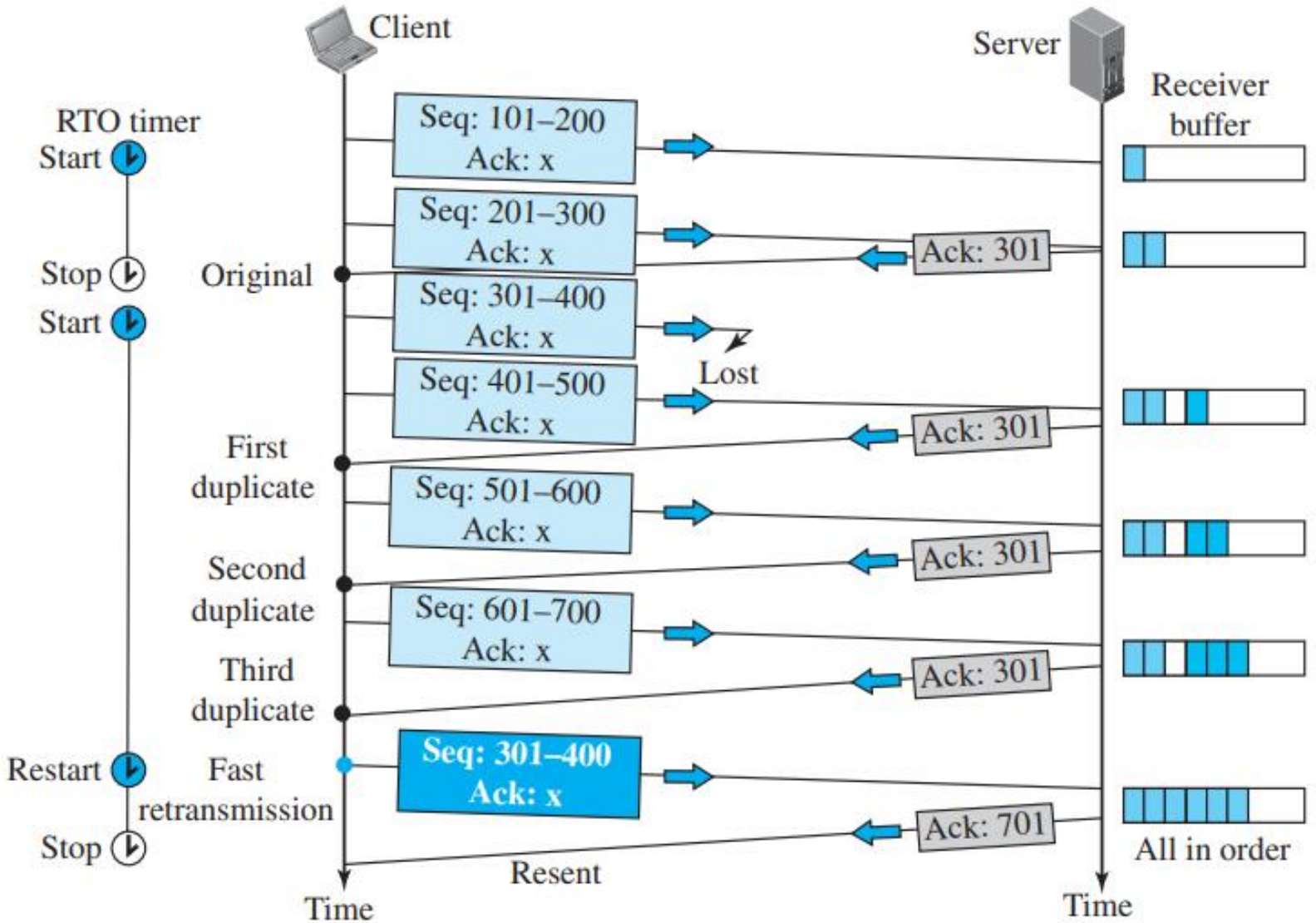
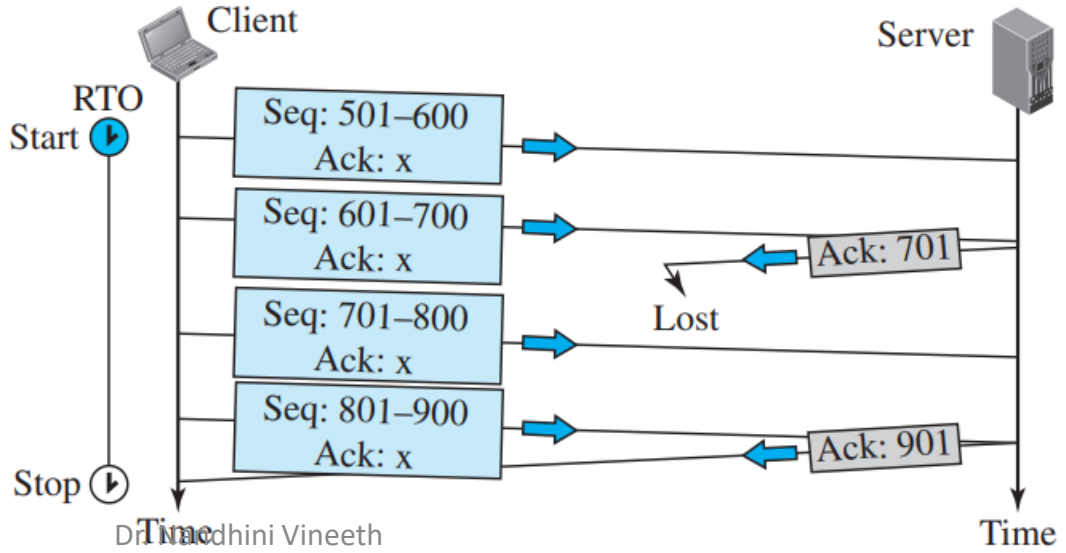


Figure 24.26 *Fast retransmission*



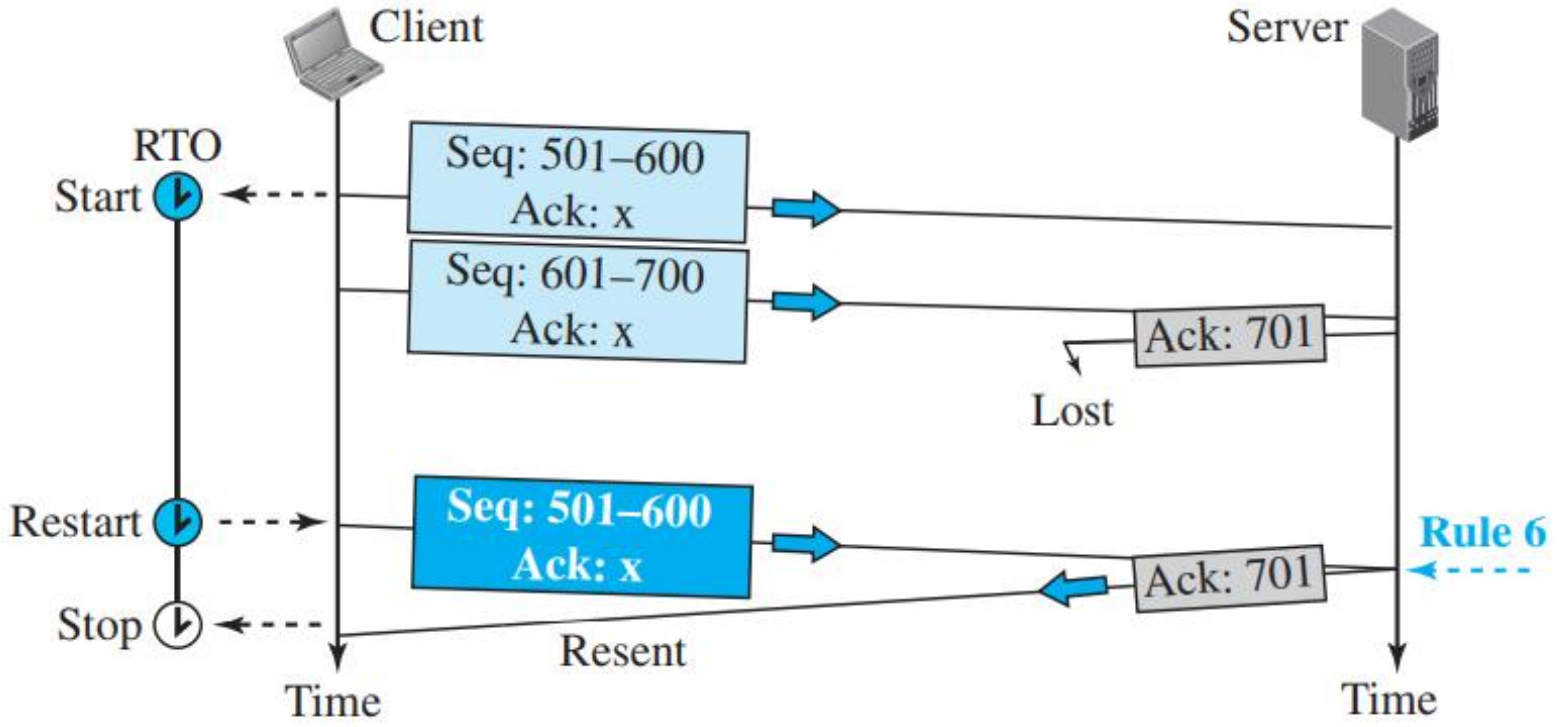
- Delayed Segment – leads to duplicate segments
- Duplicate segments
- Automatically Corrected Lost ACK

Figure 24.27 *Lost acknowledgment*



Lost Acknowledgment Corrected by Resending a Segment

Figure 24.28 *Lost acknowledgment corrected by resending a segment*



Deadlock Created by Lost Acknowledgment

- 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.

TCP Congestion Control

- A **router may receive data from more than one sender**. No matter how large the buffers of a router may be, it may be overwhelmed with data, which results in **dropping some segments** sent by a specific TCP sender.
- To control the number of segments to transmit, TCP uses another variable called a congestion window, **cwnd**, whose size is controlled by the congestion situation in the network. **The cwnd variable and the rwnd variable together define the size of the send window in TCP**. The first is related to the congestion in the middle (network); the second is related to the congestion at the end. **The actual size of the window is the minimum of these two.**
- TCP needs to define policies that **accelerate** the data transmission when there is no congestion and **decelerate** the transmission when congestion is detected.
- Actual window size = minimum (rwnd, cwnd)

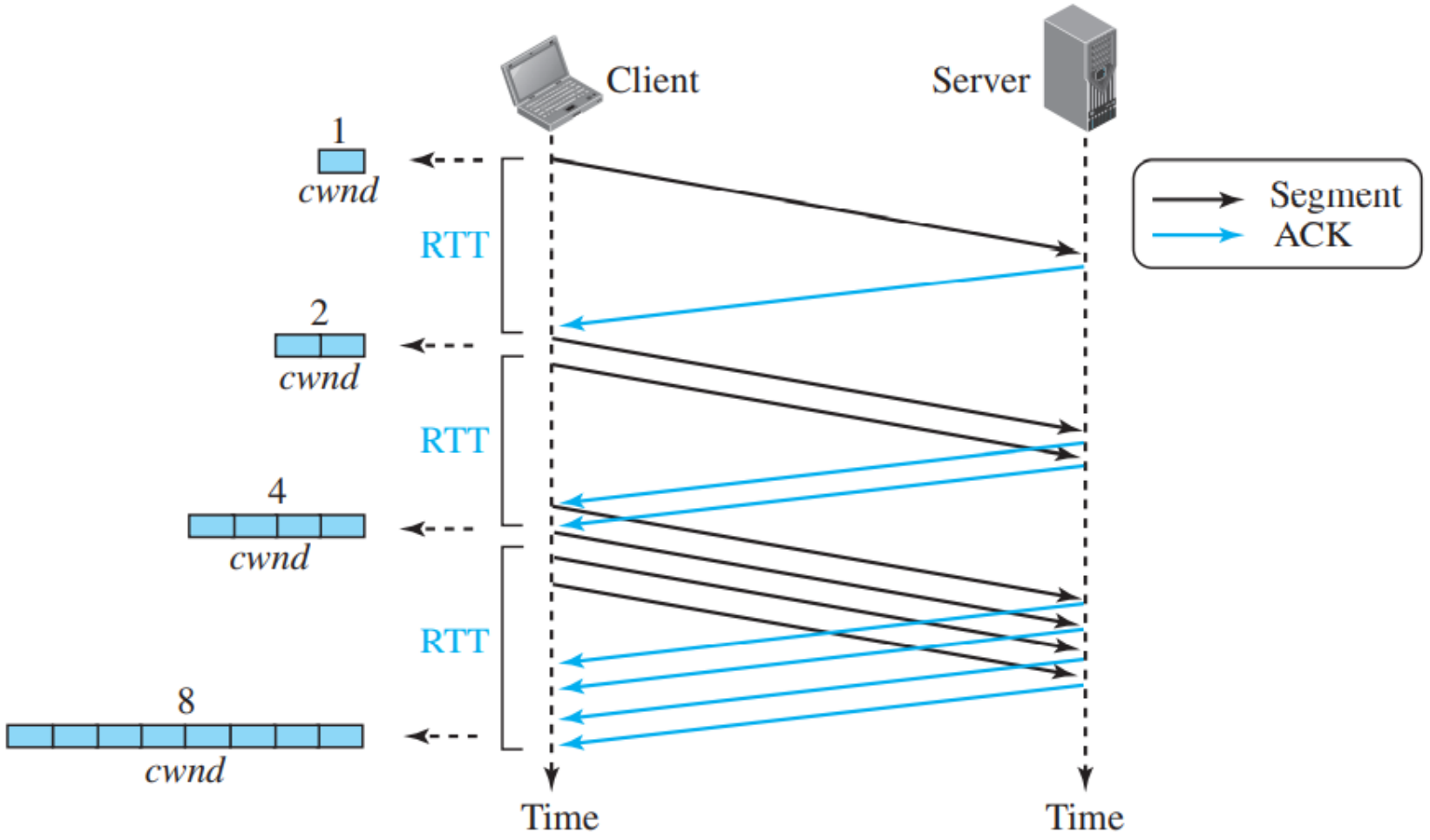
Congestion Detection

- The TCP sender uses the occurrence of two events as signs of congestion in the network: **time-out and receiving three duplicate ACKs.**
- First indicates severe congestion compared to second
- **Taho TCP** -treated both events (time-out and three duplicate ACKs) similarly, but the later version of TCP, called Reno TCP, treats these two signs differently.
- TCP sender uses only one feedback from the other end to detect congestion: **ACKs.**

Congestion control

- Congestion Policies
- TCP's general policy for handling congestion is based on three algorithms: **slow start, congestion avoidance, and fast recovery.**
- Slow Start: Exponential Increase
- The slow-start algorithm is based on the idea that the size of the congestion window (cwnd) starts with one maximum segment size (MSS), but it increases one MSS each time an acknowledgment arrives.

Figure 24.29 *Slow start, exponential increase*

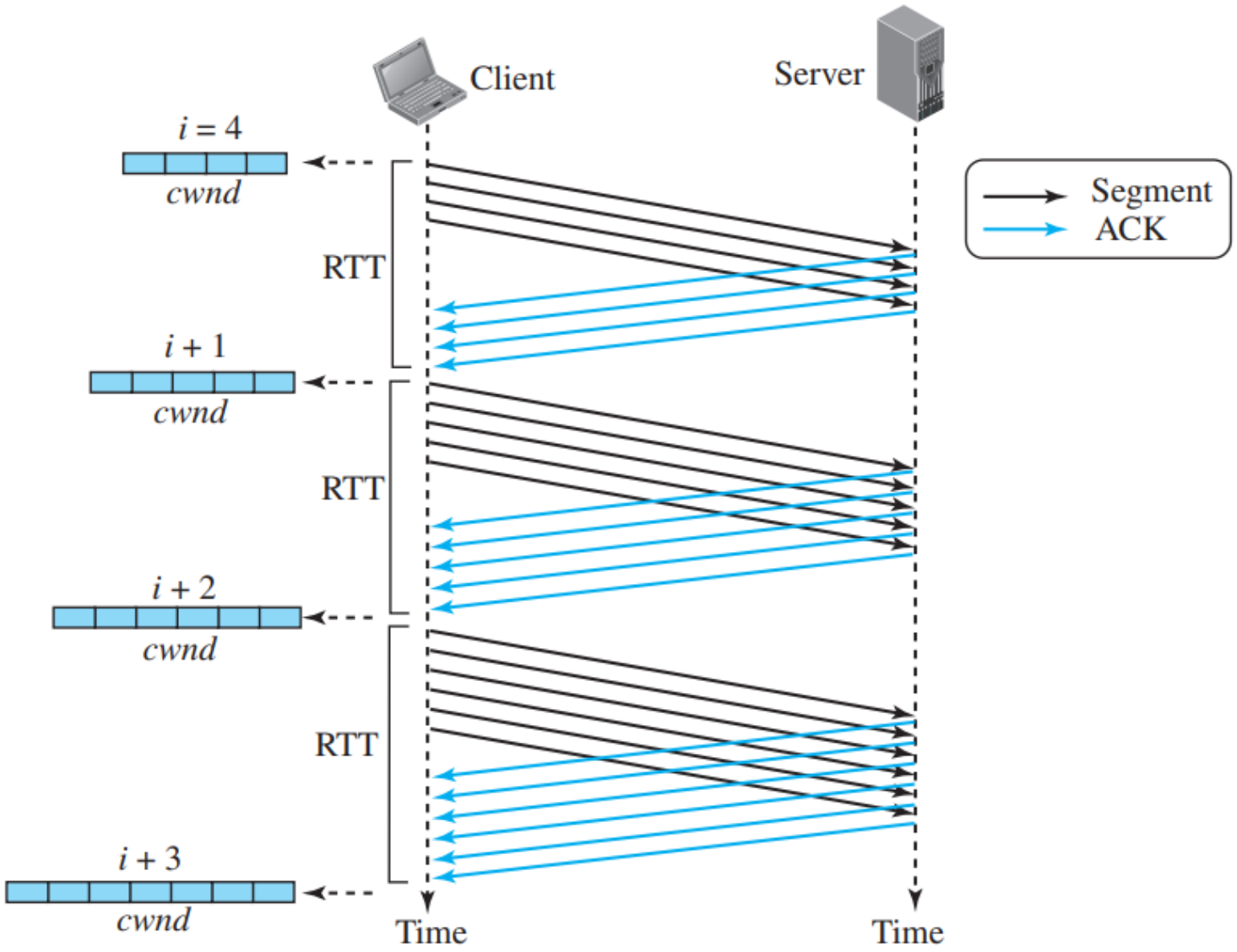


Congestion control

- Start → cwnd = 1 → 20
- After 1 RTT → cwnd = cwnd + 1 = 1 + 1 = 2→ 2^1
- After 2 RTT → cwnd = cwnd + 2 = 2 + 2 = 4→ 2^2
- After 3 RTT → cwnd = cwnd + 4 = 4 + 4 = 8→ 2^3

- **Congestion Avoidance: Additive Increase**
- To avoid congestion before it happens, we must slow down this exponential growth. TCP defines another algorithm called **congestion avoidance**, which **increases the cwnd additively instead of exponentially**. When the size of the congestion window reaches the slow-start threshold in the case where cwnd = i, the slow-start phase stops and the **additive phase begins**. In this algorithm, each time **the whole "window" of segments is acknowledged**, the size of the congestion window is **increased by one**.

Figure 24.30 *Congestion avoidance, additive increase*



Congestion control

- Start → cwnd = i
- After 1 RTT → cwnd = i + 1
- After 2 RTT → cwnd = i + 2
- After 3 RTT → cwnd = i + 3
- Fast Recovery
  - The fast-recovery algorithm is optional in TCP.
  - It starts when three duplicate ACKs arrive, which is interpreted as light congestion in the network. Like **congestion avoidance,** this algorithm is also an **additive increase**, but it increases the size of the congestion window when a duplicate ACK arrives (after the three duplicate ACKs that trigger the use of this algorithm). We can say If a duplicate ACK arrives, **cwnd = cwnd + (1 / cwnd)**

Policy Transition

- we need to refer to three versions of TCP: Taho TCP, Reno TCP, and New Reno TCP.
- Taho TCP
- Taho TCP, used only two different algorithms in their congestion policy: slow start and congestion avoidance

Figure 24.31 *FSM for Taho TCP*

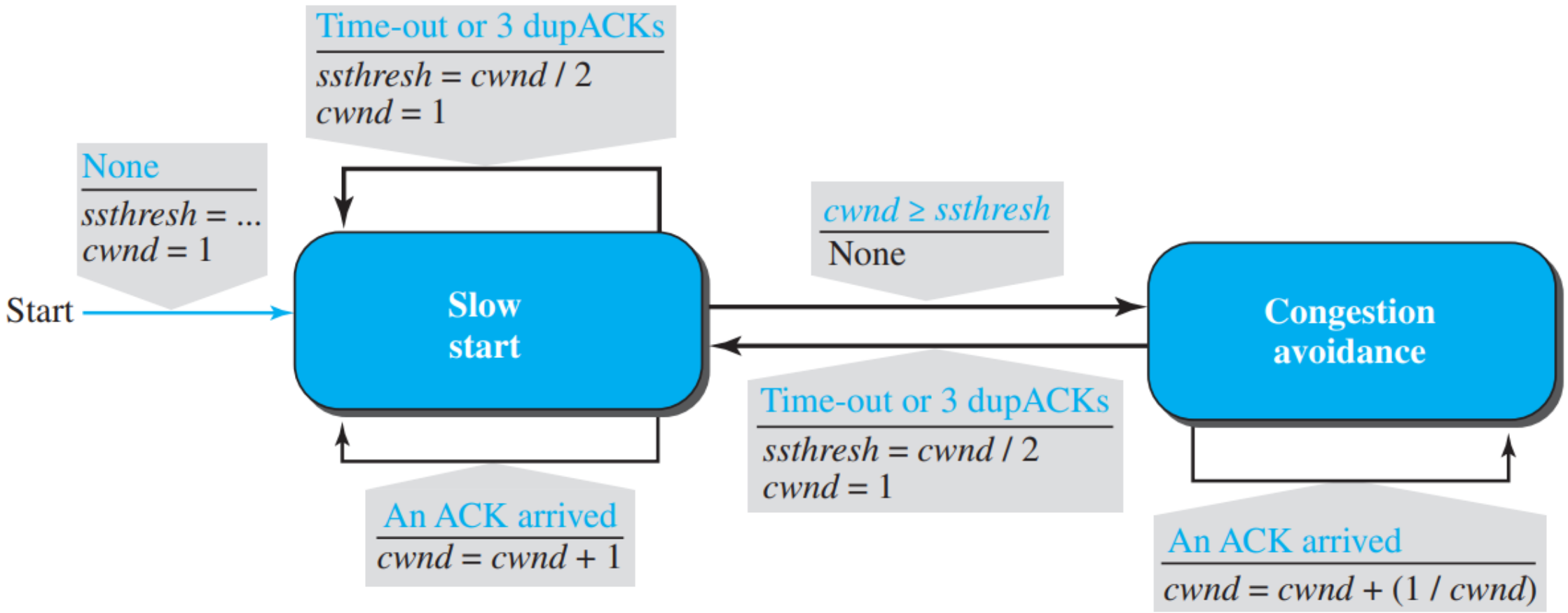


Figure 24.32 *Example 24.9*

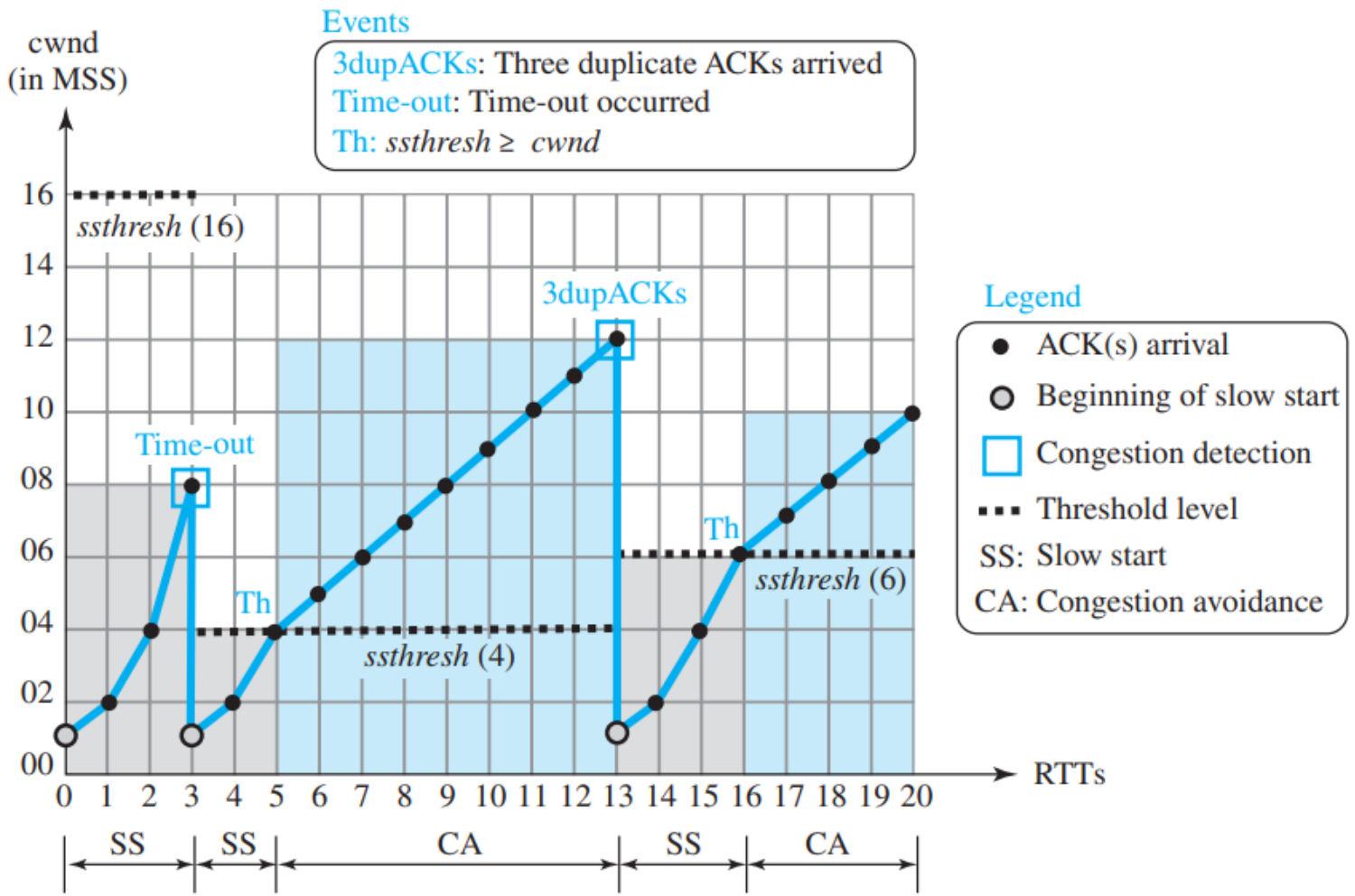


Figure 24.33 *FSM for Reno TCP*

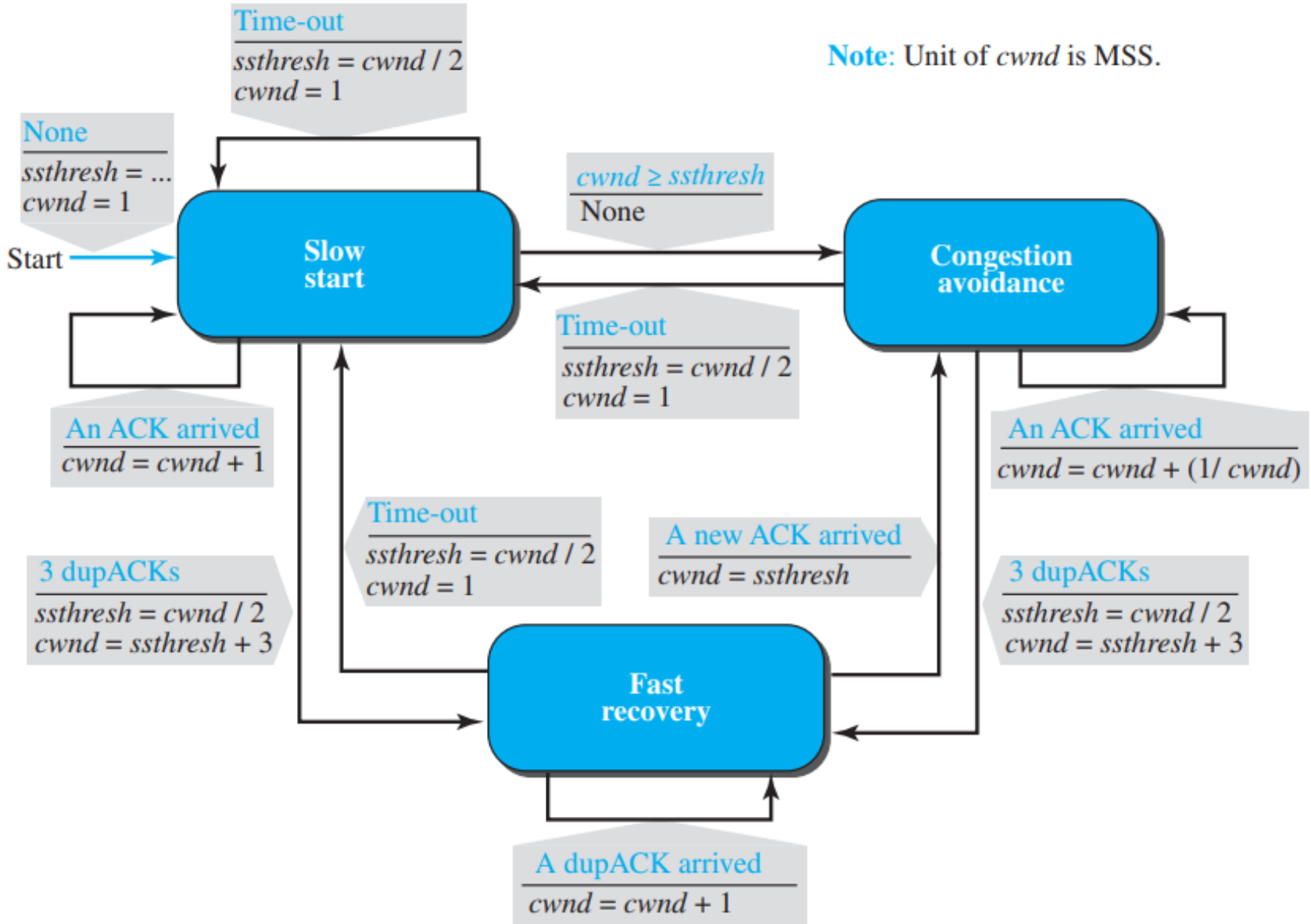


Figure 24.34 *Example 24.10*

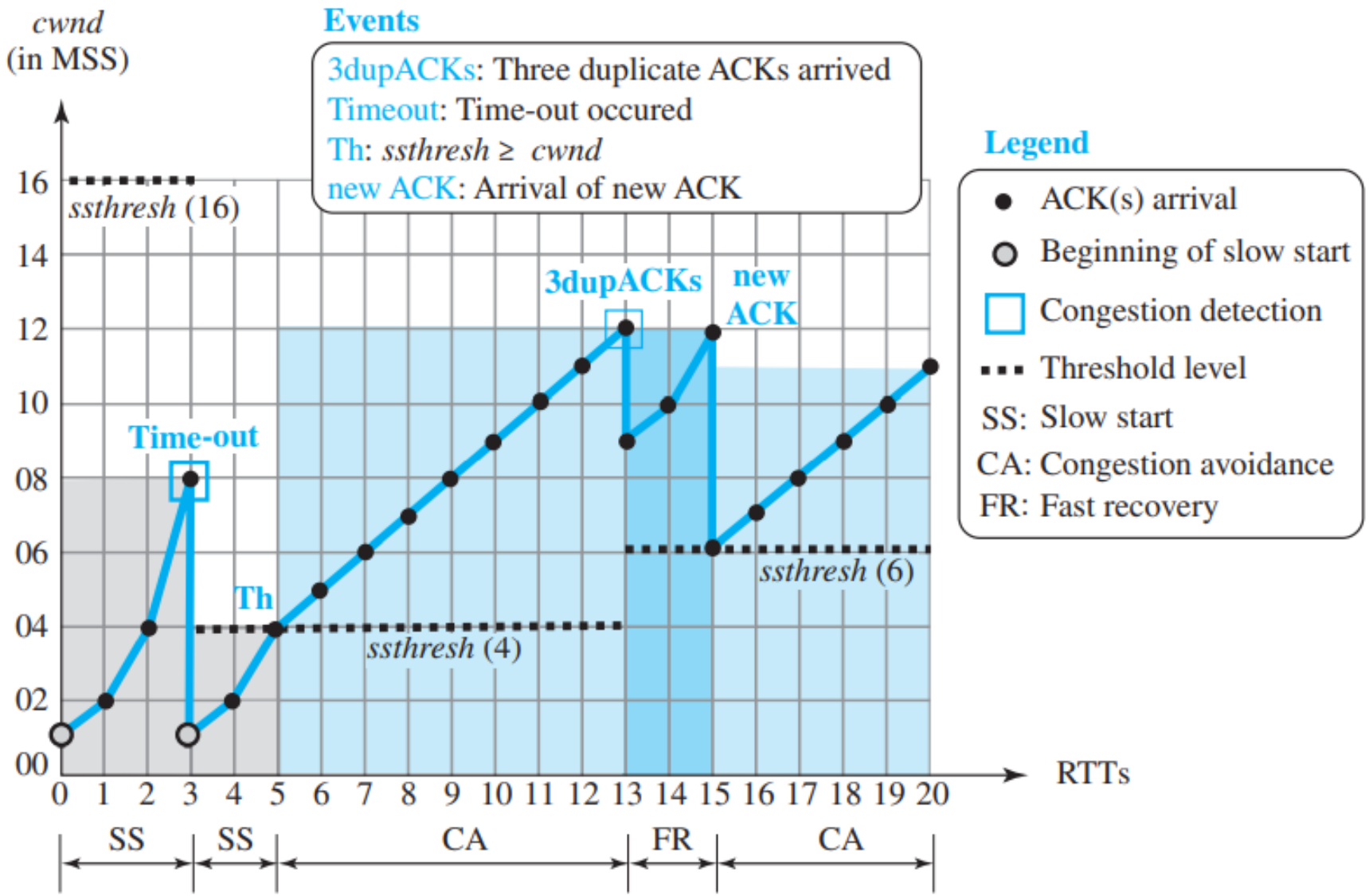
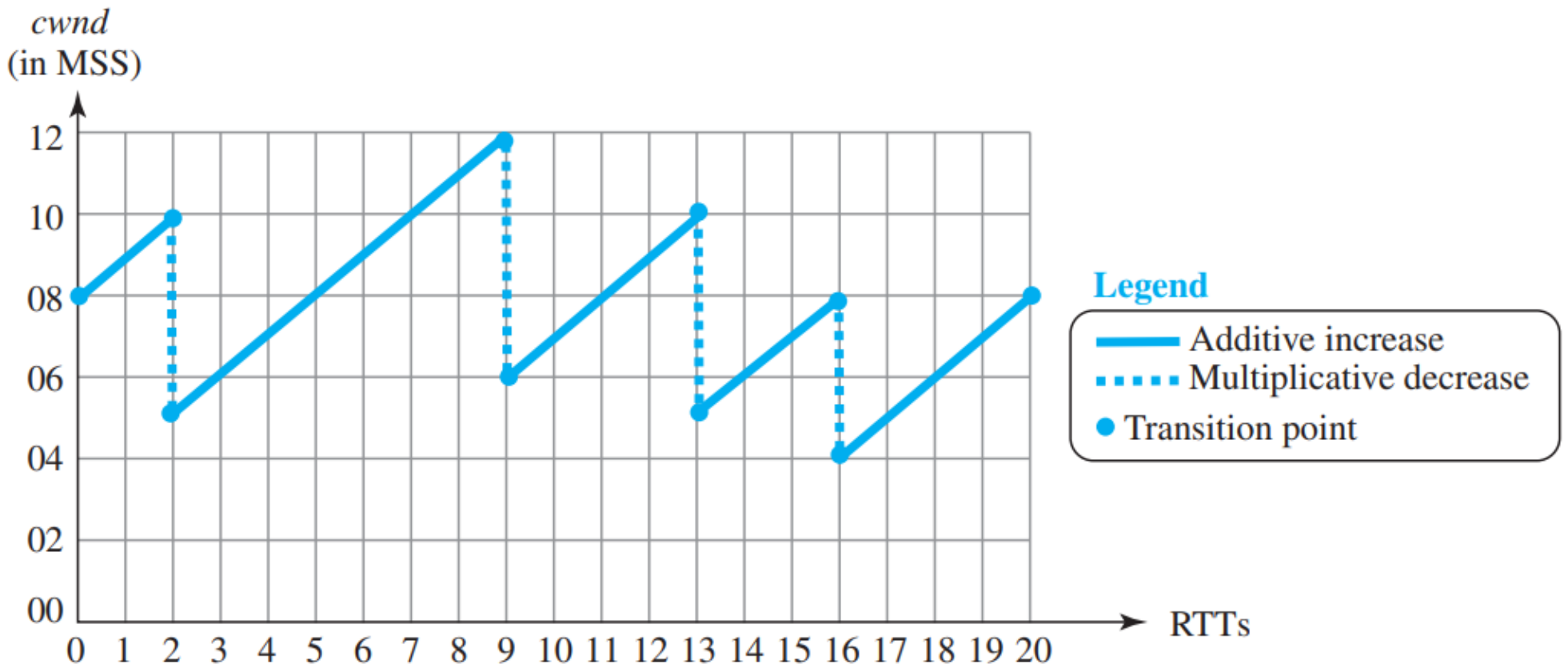


Figure 24.35 *Additive increase, multiplicative decrease (AIMD)*



TCP Throughput

- throughput = cwnd / RTT for figure with flat lines like in the fig 24.35
- But practically, **throughput = (0.75) Wmax / RTT**

TCP Timers

- TCP implementations use at least four timers: **retransmission, persistence, keepalive, and TIME-WAIT**
- Retransmission Timer
  - We can define the following rules for the retransmission timer:
  - 1. When TCP sends the segment in front of the sending queue, it starts the timer.
  - 2. When the timer expires, TCP resends the first segment in front of the queue, and restarts the timer.
  - 3. When a segment or segments are cumulatively acknowledged, the segment or segments are purged from the queue.
  - 4. If the queue is empty, TCP stops the timer; otherwise, TCP restarts the timer.

Round-Trip Time (RTT)

- Measured RTT
  - find how long it takes to send a segment and receive an acknowledgment for it. This is the measured RTT.
- Smoothed RTT.
  - The measured RTT, RTTM, is likely to change for each round trip.
  - a weighted average of RTT_M and the previous RTT – alpha ->1/8

Initially	$\rightarrow$	**No value**
After first measurement	$\rightarrow$	$\mathbf{RTT_S = RTT_M}$
After each measurement	$\rightarrow$	$\mathbf{RTT_S = (1 - \alpha)\, RTT_S + \alpha \times RTT_M}$

Round-Trip Time (RTT)

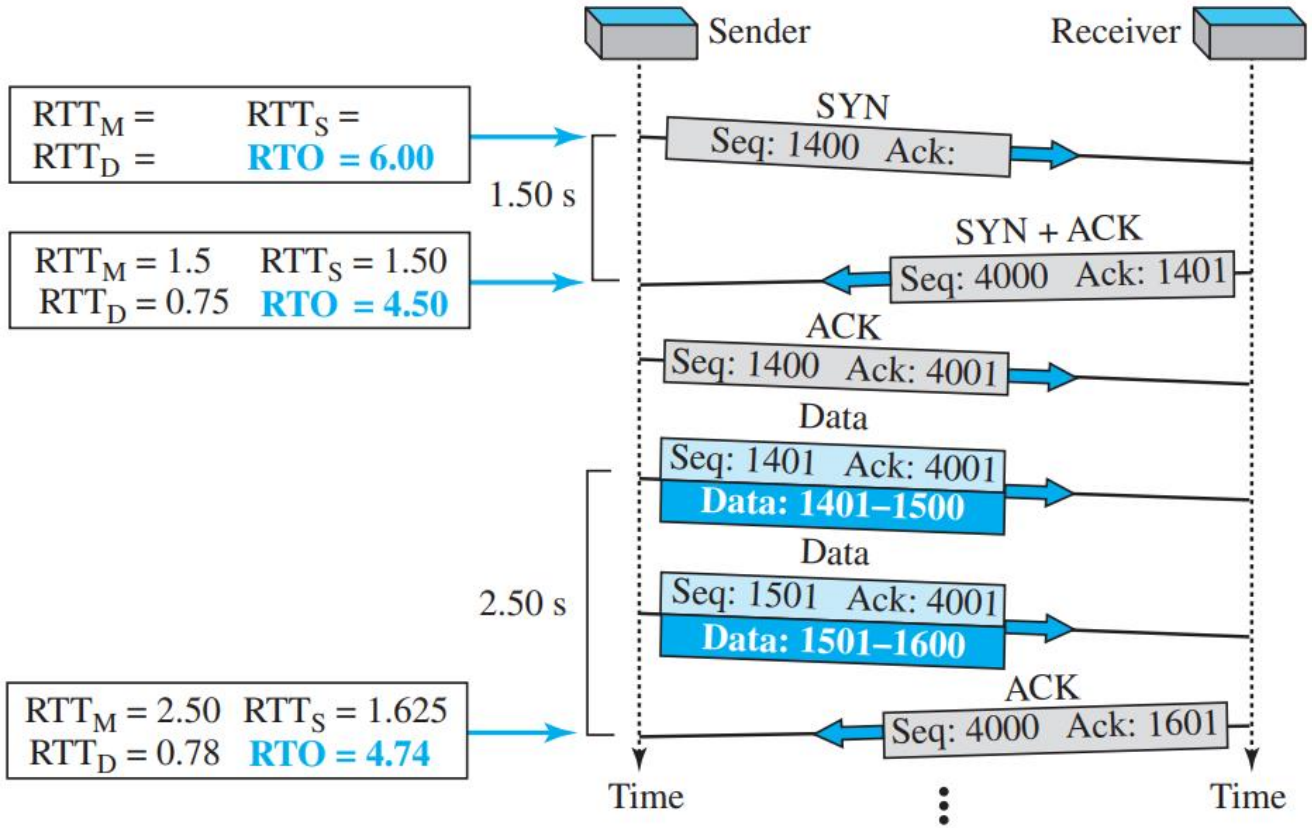
- ***RTT Deviation.*** Most implementations do not just use RTT_S; they also calculate the RTT deviation, called RTT_D, based on the RTT_S and RTT_M, using the following formulas. (The value of β is also implementation-dependent, but is usually set to 1/4.)

Initially	→	**No value**
After first measurement	→	$\mathbf{RTT_D = RTT_M / 2}$
After each measurement	→	$\mathbf{RTT_D = (1 - \beta)\, RTT_D + \beta \times \lvert RTT_S - RTT_M \rvert}$

Retransmission Time-out (RTO) The value of RTO is based on the smoothed round-trip time and its deviation. Most implementations use the following formula to calculate the RTO:

Original	→	**Initial value**
After any measurement	→	$\mathbf{RTO = RTT_S + 4 \times RTT_D}$

Figure 24.36 *Example 24.12*



RTTM = 2.5

RTTS = (7/8) × (1.5) + (1/8) × (2.5) = 1.625

RTTD = (3/4) × (0.75) + (1/4) × |1.625 − 2.5| = 0.78

RTO = 1.625 + 4 × (0.78) = 4.74

TCP Timers

- **Persistence Timer**
  - To correct this deadlock, TCP uses a persistence timer for each connection.
  - 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.** The value of the persistence timer is set to the value of the retransmission time. However, if a response is not received from the receiver**, another probe segment is sent** and the value of the **persistence timer is doubled** and reset. After that the sender sends one probe segment every **60 seconds** until the window is reopened.
- **Keepalive Timer** A keepalive timer is used in some implementations to prevent a **long idle connection between two TCPs**. Suppose that a client opens a TCP connection to a server, **transfers some data, and becomes silent**. Perhaps the client has crashed. In this case, the connection remains open forever. To remedy this situation, most implementations **equip a server with a keepalive timer.** 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 seconds apart, it assumes that the client is down **and terminates the connection.**

- **TIME-WAIT** Timer The TIME-WAIT (2MSL) timer is used during **connection termination**. The **maximum segment lifetime (MSL)** is the amount of time any segment can exist in a network before being discarded. The implementation needs to choose a value for MSL. Common values are **30 seconds, 1 minute, or even 2 minutes**. The 2MSL timer is used **when TCP performs an active close** and sends the final ACK. The connection must stay open for 2 MSL amount of time to allow TCP to resend the final ACK **in case the ACK is lost.** This requires that the RTO timer at the other end times out and new FIN and ACK segments are resent.