EFSM/SDL modeling of the original TCP standard (RFC793) and the Congestion Control Mechanism of TCP Reno

Raid Y. Zaghal and Javed I. Khan
Networking and Media Communications Research Laboratories
Department of Computer Science, Kent State University
233 MSB, Kent, OH 44242
javed|rzaghal@cs.kent.edu

Abstract—*in this document we provide a complete EFSM/SDL model for the original TCP standard that was proposed in RFC 793 and the congestion control mechanism of TCP Reno. We have developed this model as a supplement material for the (InTraN) paradigm [Zag05] which we have developed recently for extensible networking. We felt that this model can be beneficial for other researchers who might be interested in the formal description of the TCP standard using the EFSM/SDL notation.*

1. Introduction

The original Transmission Control Protocol (TCP) standard was described in RFC 793 [Pos81] which provided a formal description of a highly reliable host-to-host protocol between hosts in a packet-switched network. Although many enhancement has been made on the original standard and different generations of TCP has been proposed (e.g., Taho, Reno, Vegas), most of the fundamental features have remained unchanged. The SDL model presented here extends the original TCP standard by adding the congestion control mechanisms proposed by Jacobson. These include the slow start-congestion avoidance mechanism [Jac88], and the fast retransmit-fast recovery mechanism [Jac90]. Other TCP variants can also be modeled by simply extending the original model in the same fashion.

This report is organized as follows: in the next section we provide some background information on the EFSM/SDL notation, in section 3 we formally present the TCP EFSM/SDL model, in section 4 we show how other TCP variants (e.g., TCP Vegas) can modeled based on the SDL model presented in this document, and in section 5 we conclude the document.

2. Background: EFSM/SDL

SDL (Specification and Description Language) [Ell97, SDLfrm] is an ITU-standardized language for the formal description of communication protocols. It is also suited for any application based on the finite state machine concept, such as circuit design. The programming model used by SDL is based on extended finite state machines (EFSM) [Ell97, Byu01]. SDL augments the finite state machine model by providing variables and timers and by supporting object-oriented programming. Informally, the EFSM is composed of states and transitions among them. For a transition to occur, the system must receive an event from the environment which triggers corresponding actions. After performing the actions, the EFSM produces output signals to the environment. An SDL system is composed of several protocol entities; each entity is designed as a single EFSM. Formally, An EFSM is a 6-tuple (S, s_0, E, f, O, V), where S is a set of states, s_0 is an initial state, E is a set of events, f is a state transition function, O is a set of output signals, and V is a set of variables. The function f returns a next state, a set of output signals, and an action list for each combination of a current state and an input event. An EFSM also uses predicates to control the behavior of the protocol. These predicates usually allow similar states to be grouped therefore reducing the total number of states.

3. TCP EFSM/SDL Model

3.1. Remarks and Simplifying Assumptions:

1- Always remembers the current state in the variable (CurrState) and the previous state in the variable (PrevState)

2- The TCP endpoint has unlimited buffer space (e.g., buffer space to queue SENDs and RECEIVEs is always available)

3- In any state, whenever a segment is sent, the segment is added to the Retransmission Queue (Rexmt Queue) and the retransmission timer (REXMT) is started.

4- The (REXMT TIMEOUT) event has been modeled in all states except (FIN-WAIT-2, TIME-WAIT, CLOSED), since in these states the endpoint have already received an ACK of its FIN segment (i.e., will not transmit any segments afterwards).

5- The (TIMEWAIT TIMEOUT) event has been modeled in (TIME-WAIT) state only. In all other states, this timer is irrelevant.

3.2. The following were not modeled from RFC 793:

1- Security/Compartment and Precedence processing.

2- The STATUS user call.

3- The PUSH mechanism (i.e., PSH control bit)

4- The URGENT mechanism (i.e., URG control bit)

3.3. The TCP EFSM=(S, s_0, E, f, O, V):

1. States (S) = {CLOSED, LISTEN, SYN-SENT, SYN-RCVD, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSING, CLOSE-WAIT, LAST-ACK, TIME-WAIT}.

2. Initial State (s_0) = {CLOSED}

3. Events (E)

User Calls (subscriber events) = {Active OPEN, Passive OPEN, SEND, RECEIVE, CLOSE, ABORT}.

Arriving Segments (service events) = {SEGMENT ARRIVE (SYN, ACK, RST, FIN)}.

Timeouts (internal events) = {REXMT TIMEOUT, TIME-WAIT TIMEOUT, USER-TIME TIMEOUT}.

4. Transition Function (f) = {described in the SDL figures below}

5. Output Signals (O) = {Return (message), Return Error (error message), Signal User (message), and Segment (SEG)}.

6. Variables (V)

A. Segment Variables

SEG.SEQ: segment sequence number
SEG.ACK: segment acknowledgment number
SEG.LEN: segment length
SEG.WND: segment window (Receiver Advertised Window)
SEG.CTL: control bits (ACK, RST, SYN, FIN)

B. Send Sequence Variables

SND.UNA: send unacknowledged
SND.NXT: send next

SND.WND: send window
ISS: initial send sequence number

C. Receive Sequence Variables

RCV.NXT: receive next
RCV.WND: receive window
IRS: initial receive sequence number

D. Timers

REXMT: Retransmission Timer
TIMEWAIT: Time-wait Timer
USERTIME: User Timer

E. Counters

dACK: duplicate ACK counter
ExpBoff: exponential backoff counter

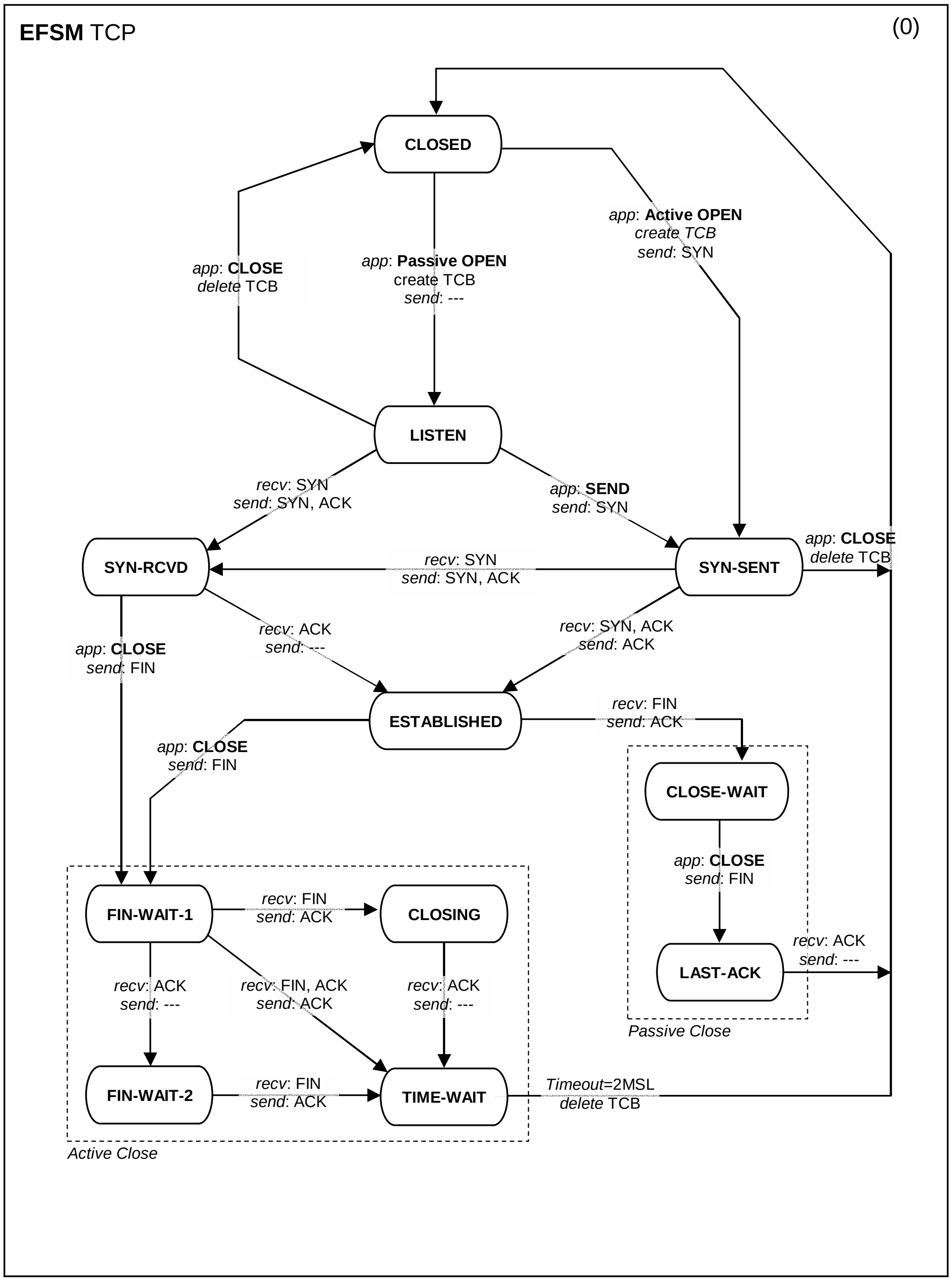
F. Other

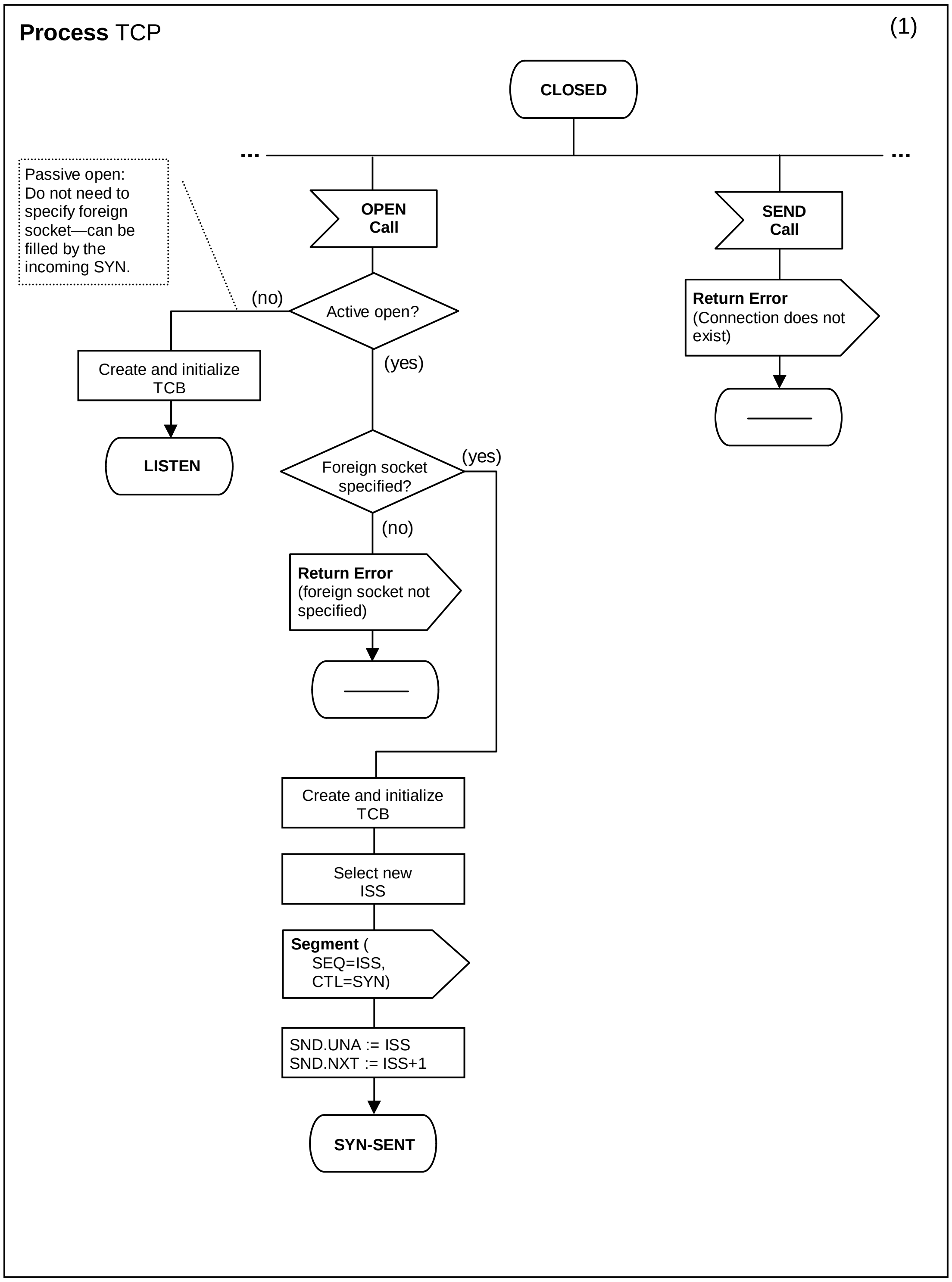
CurrState: Current State
PrevState: Previous State
RTO: Retransmission Timer Out value
RTT: Round Trip Time—used to calculate RTO
SRTT: Smoothed RTT—used to calculate RTO
CWND: Congestion window
MSS: Maximum Segment Size
SSthresh: Slow Start Threshold
MSL: Maximum Segment Lifetime

G. Buffers

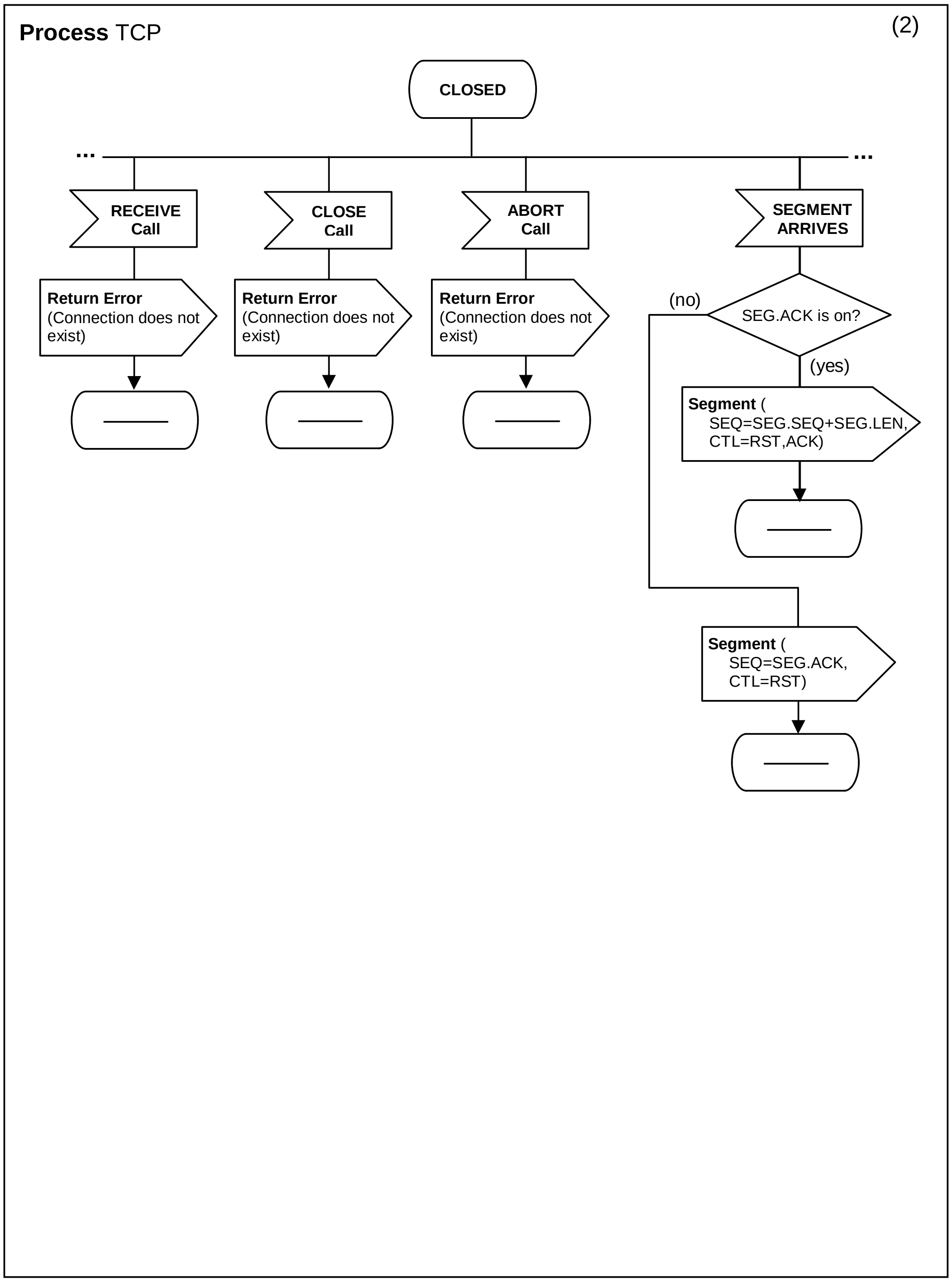
Send Buffer: Send Buffer [*referred to in the table as* SBuff]
RCV Buffer: Receive Buffer [*referred to in the table as* RBuff]
OO RCV Buffer: Out of Order Receive Buffer [*referred to in the table as* ORBuff]
Rexmt Queue: Holds sent but unacknowledged segments [*referred to in the table as* RexQ]
User Calls Queue: Holds outstanding user calls (e.g., SEND, RECEIVE, CLOSE) [*referred to in the table as* UCallsQ]

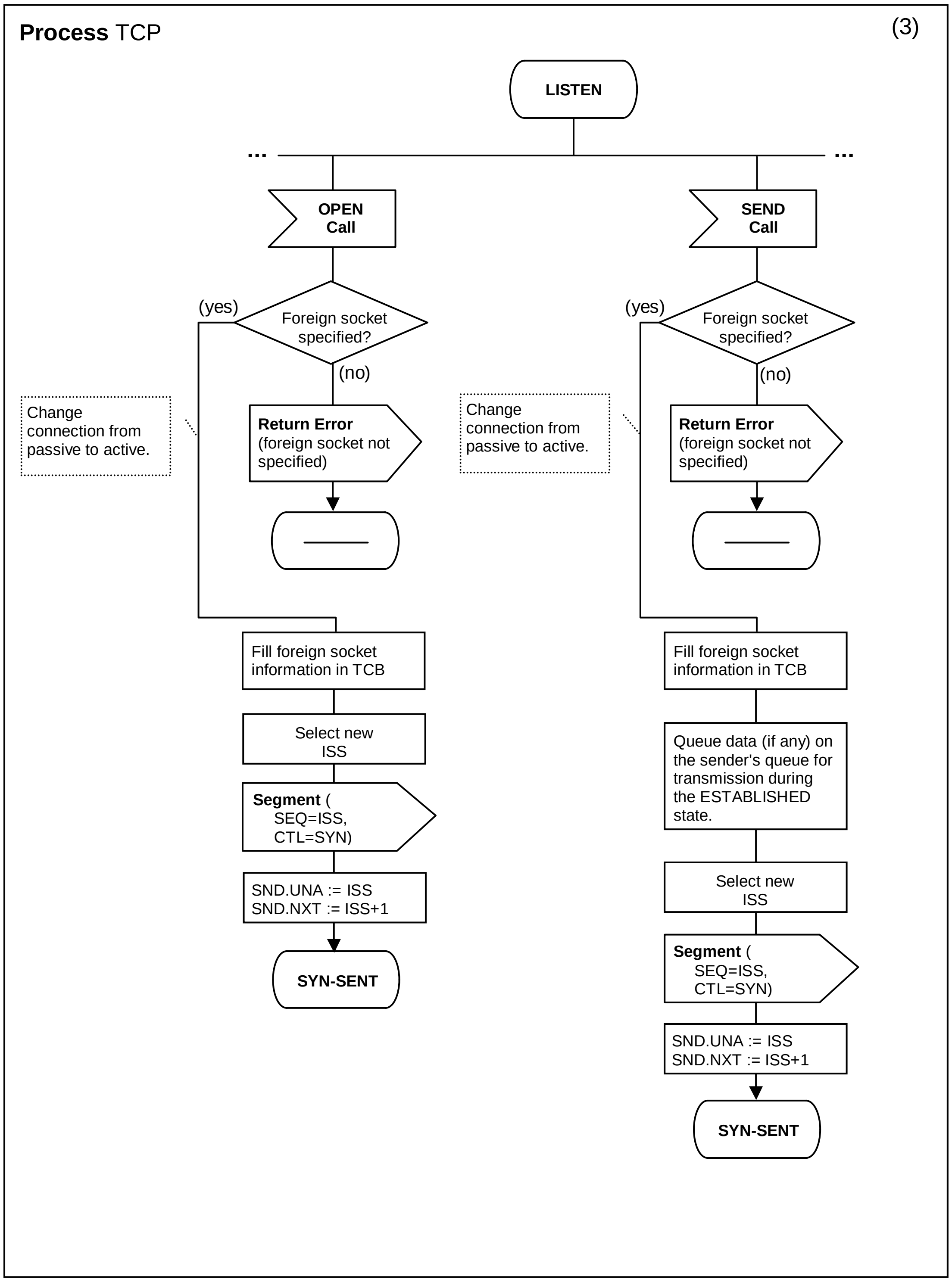
In the following pages (5 to 44) we present the EFSM/SDL model. Each figure has a number in the upper-right corner. Figure 0 on the next page depicts the EFSM diagram and the subsequent figures (1 to 39) provide the SDL description of all transitions.


EFSM TCP
(0)
CLOSED
app: **Passive OPEN** create TCB *send*: ---
app: **Active OPEN** *create* TCB *send*: SYN
app: **CLOSE** *delete* TCB
LISTEN
recv: SYN *send*: SYN, ACK
app: **SEND** *send*: SYN
SYN-RCVD
recv: SYN *send*: SYN, ACK
SYN-SENT
app: **CLOSE** *delete* TCB
app: **CLOSE** *send*: FIN
recv: ACK *send*: ---
recv: SYN, ACK *send*: ACK
ESTABLISHED
recv: FIN *send*: ACK
app: **CLOSE** *send*: FIN
CLOSE-WAIT
app: **CLOSE** *send*: FIN
LAST-ACK
recv: ACK *send*: ---
Passive Close
FIN-WAIT-1
recv: FIN *send*: ACK
CLOSING
recv: ACK *send*: ---
recv: FIN, ACK *send*: ACK
recv: ACK *send*: ---
FIN-WAIT-2
recv: FIN *send*: ACK
TIME-WAIT
Timeout=2MSL *delete* TCB
Active Close

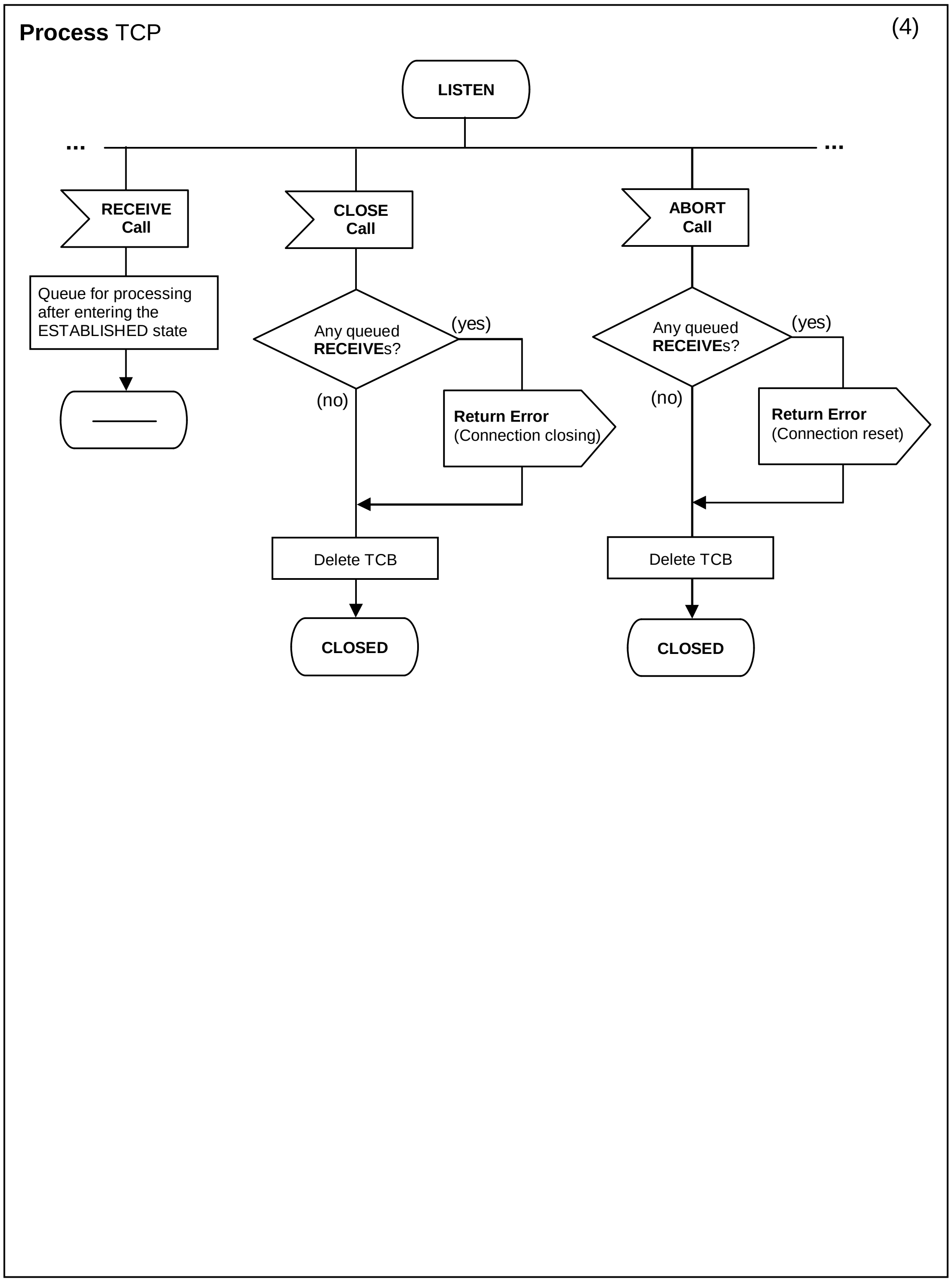


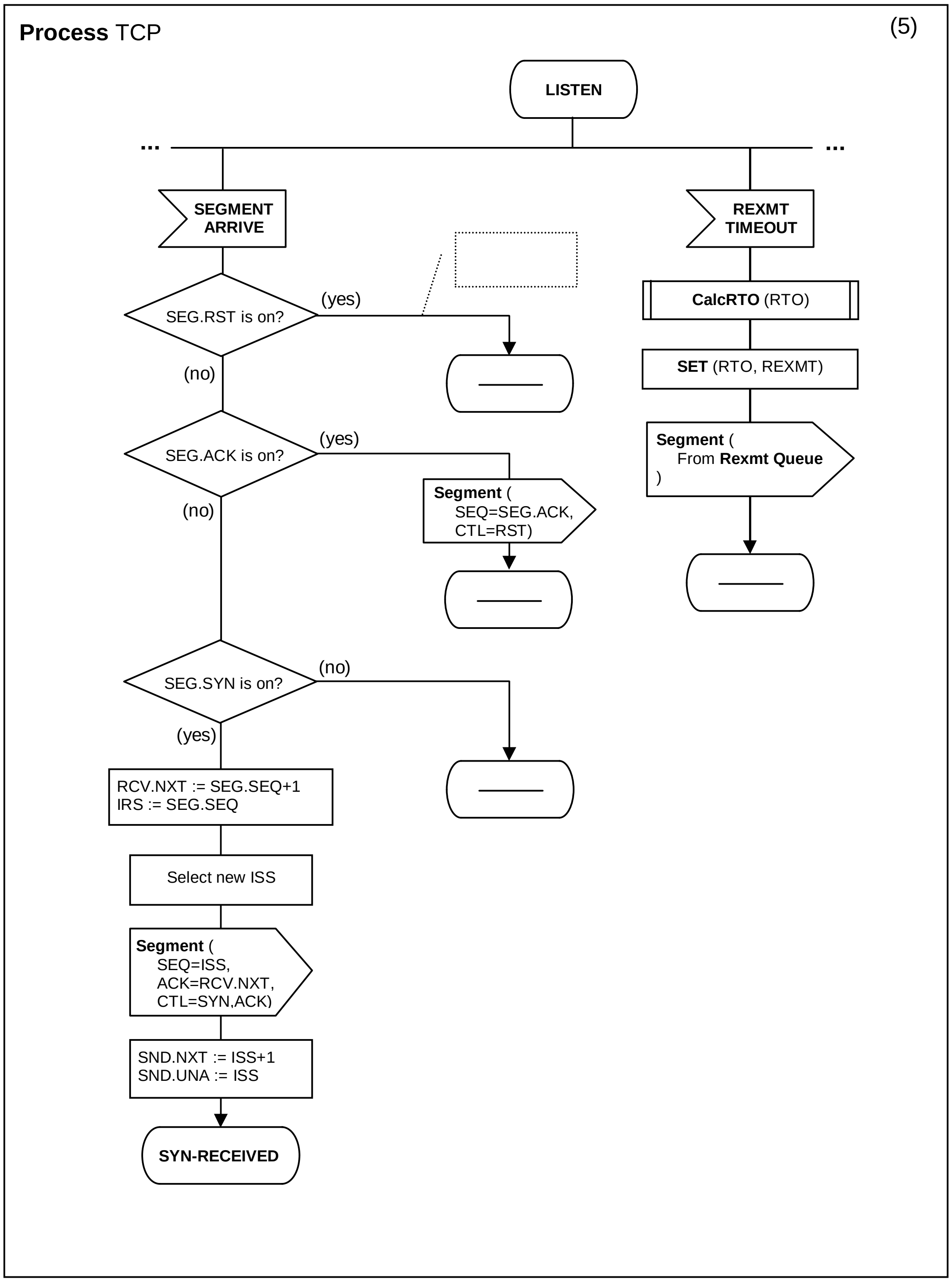
Process TCP
(1)
CLOSED
...
...
OPEN Call
SEND Call
Passive open: Do not need to specify foreign socket—can be filled by the incoming SYN.
(no)
Active open?
(yes)
Create and initialize TCB
LISTEN
Return Error (Connection does not exist)
Foreign socket specified?
(yes)
(no)
Return Error (foreign socket not specified)
Create and initialize TCB
Select new ISS
Segment (SEQ=ISS, CTL=SYN)
SND.UNA := ISS
SND.NXT := ISS+1
SYN-SENT

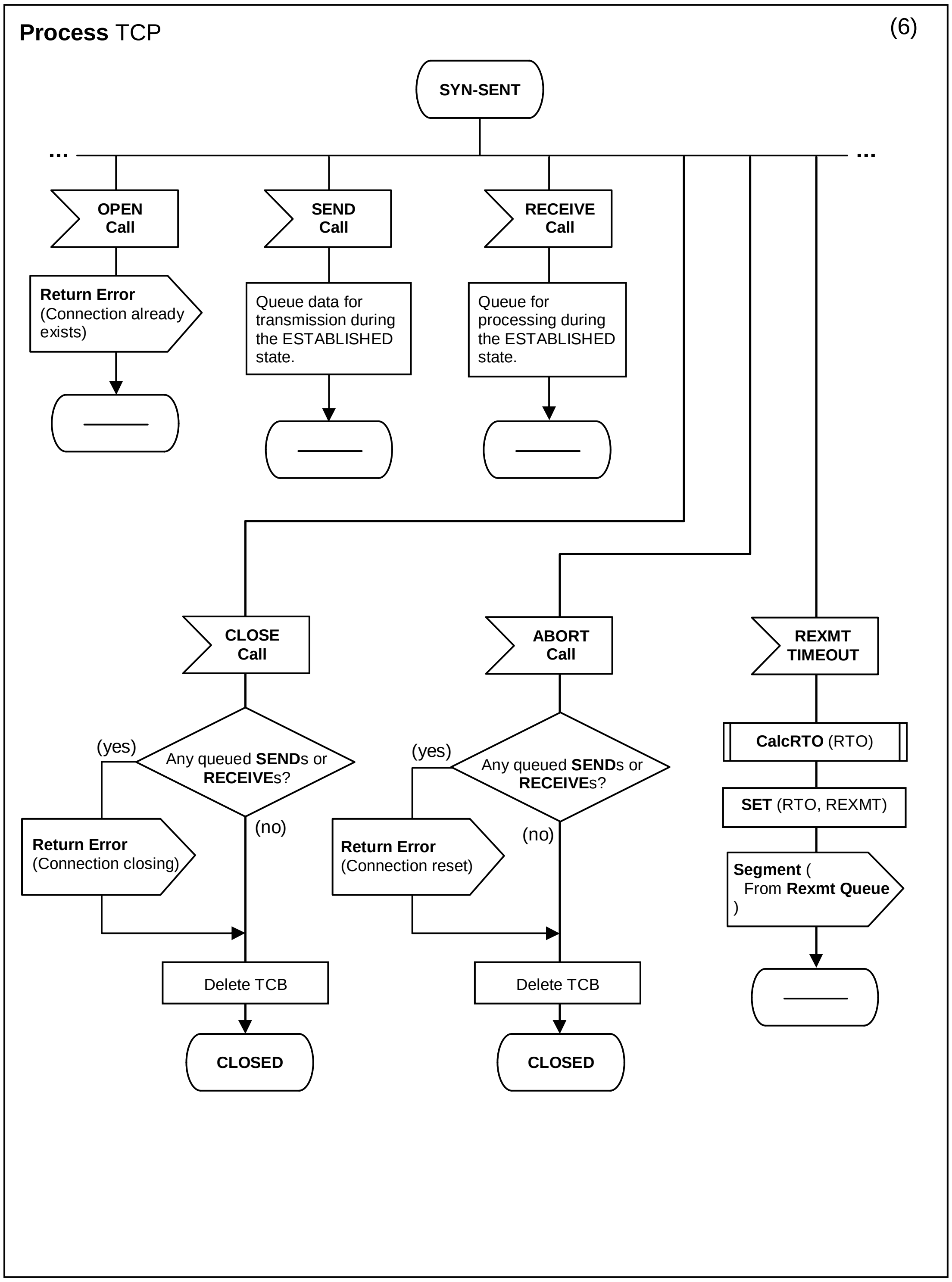

Process TCP
(2)
CLOSED
...
...
RECEIVE Call
CLOSE Call
ABORT Call
SEGMENT ARRIVES
Return Error (Connection does not exist)
Return Error (Connection does not exist)
Return Error (Connection does not exist)
SEG.ACK is on?
(no)
(yes)
Segment (SEQ=SEG.SEQ+SEG.LEN, CTL=RST,ACK)
Segment (SEQ=SEG.ACK, CTL=RST)

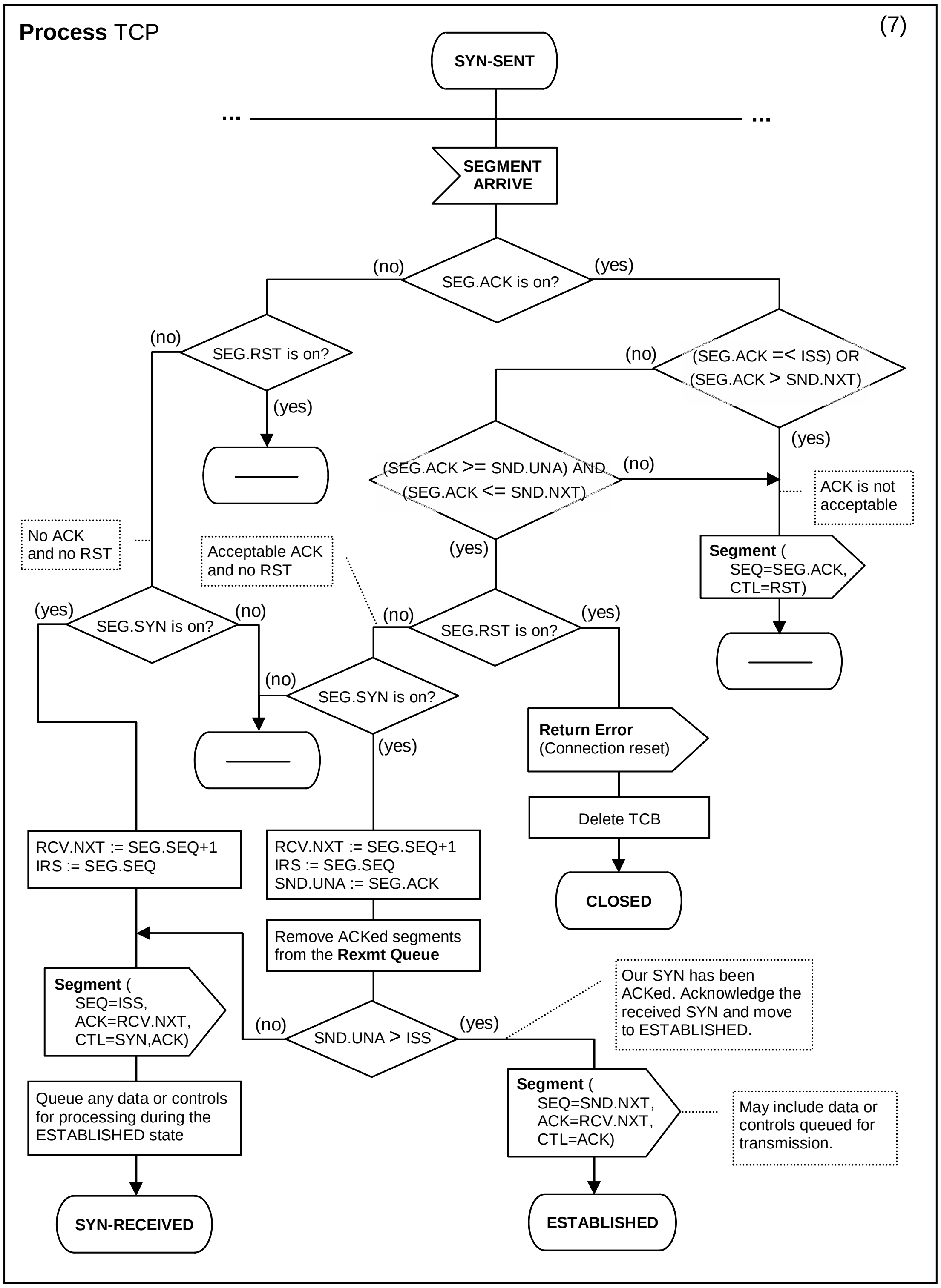


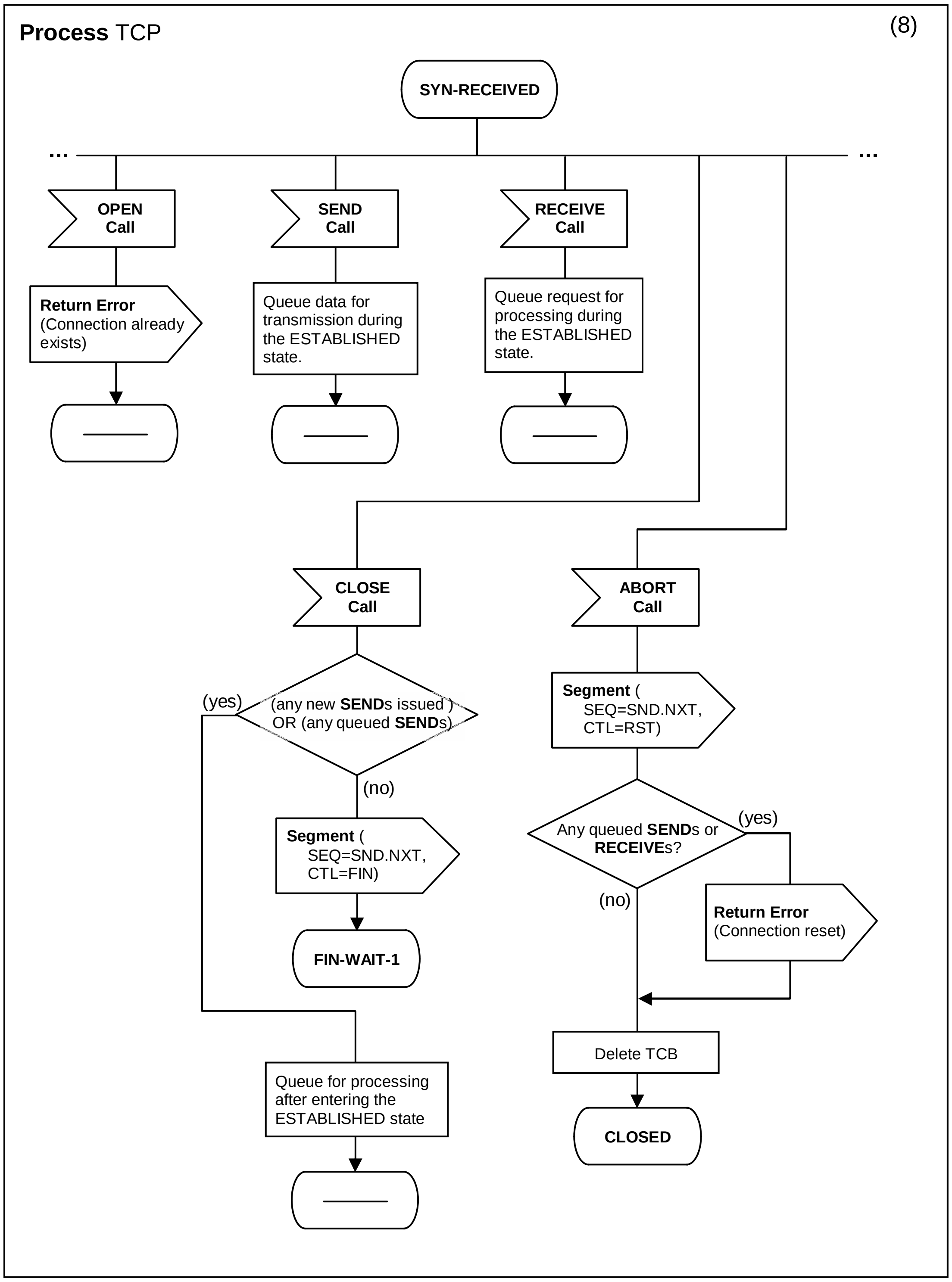
Process TCP
(3)
LISTEN
...
...
OPEN Call
Foreign socket specified?
(yes)
(no)
Change connection from passive to active.
Return Error (foreign socket not specified)
Fill foreign socket information in TCB
Select new ISS
Segment (SEQ=ISS, CTL=SYN)
SND.UNA := ISS
SND.NXT := ISS+1
SYN-SENT
SEND Call
Foreign socket specified?
(yes)
(no)
Change connection from passive to active.
Return Error (foreign socket not specified)
Fill foreign socket information in TCB
Queue data (if any) on the sender's queue for transmission during the ESTABLISHED state.
Select new ISS
Segment (SEQ=ISS, CTL=SYN)
SND.UNA := ISS
SND.NXT := ISS+1
SYN-SENT


Process TCP
(4)
LISTEN
...
...
RECEIVE Call
Queue for processing after entering the ESTABLISHED state
CLOSE Call
Any queued **RECEIVE**s?
(yes)
(no)
Return Error (Connection closing)
Delete TCB
CLOSED
ABORT Call
Any queued **RECEIVE**s?
(yes)
(no)
Return Error (Connection reset)
Delete TCB
CLOSED

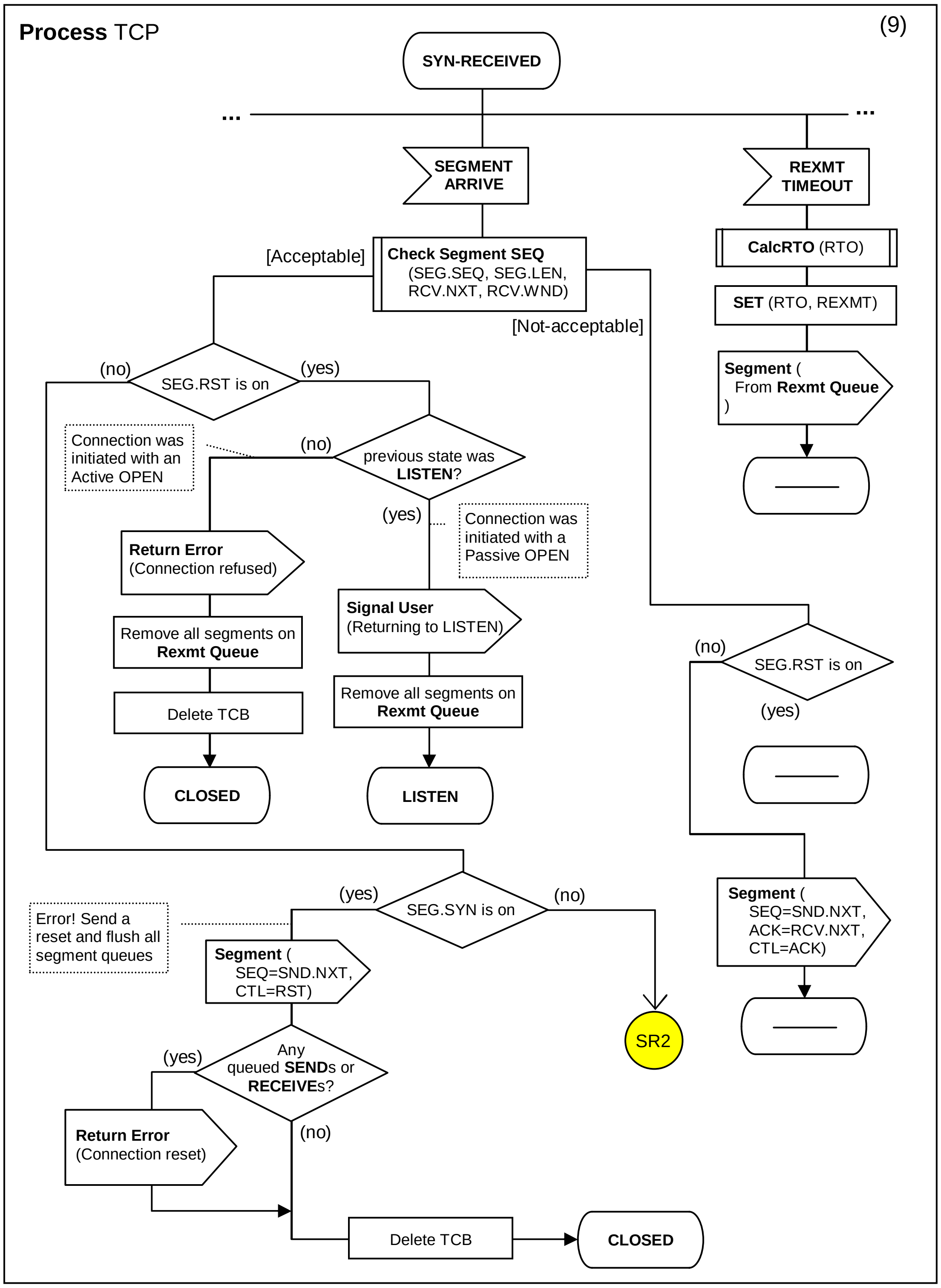

Process TCP
(5)
LISTEN
...
...
SEGMENT ARRIVE
REXMT TIMEOUT
SEG.RST is on?
(yes)
(no)
CalcRTO (RTO)
SET (RTO, REXMT)
SEG.ACK is on?
(yes)
(no)
Segment (From Rexmt Queue)
Segment (SEQ=SEG.ACK, CTL=RST)
SEG.SYN is on?
(no)
(yes)
RCV.NXT := SEG.SEQ+1
IRS := SEG.SEQ
Select new ISS
Segment (SEQ=ISS, ACK=RCV.NXT, CTL=SYN,ACK)
SND.NXT := ISS+1
SND.UNA := ISS
SYN-RECEIVED

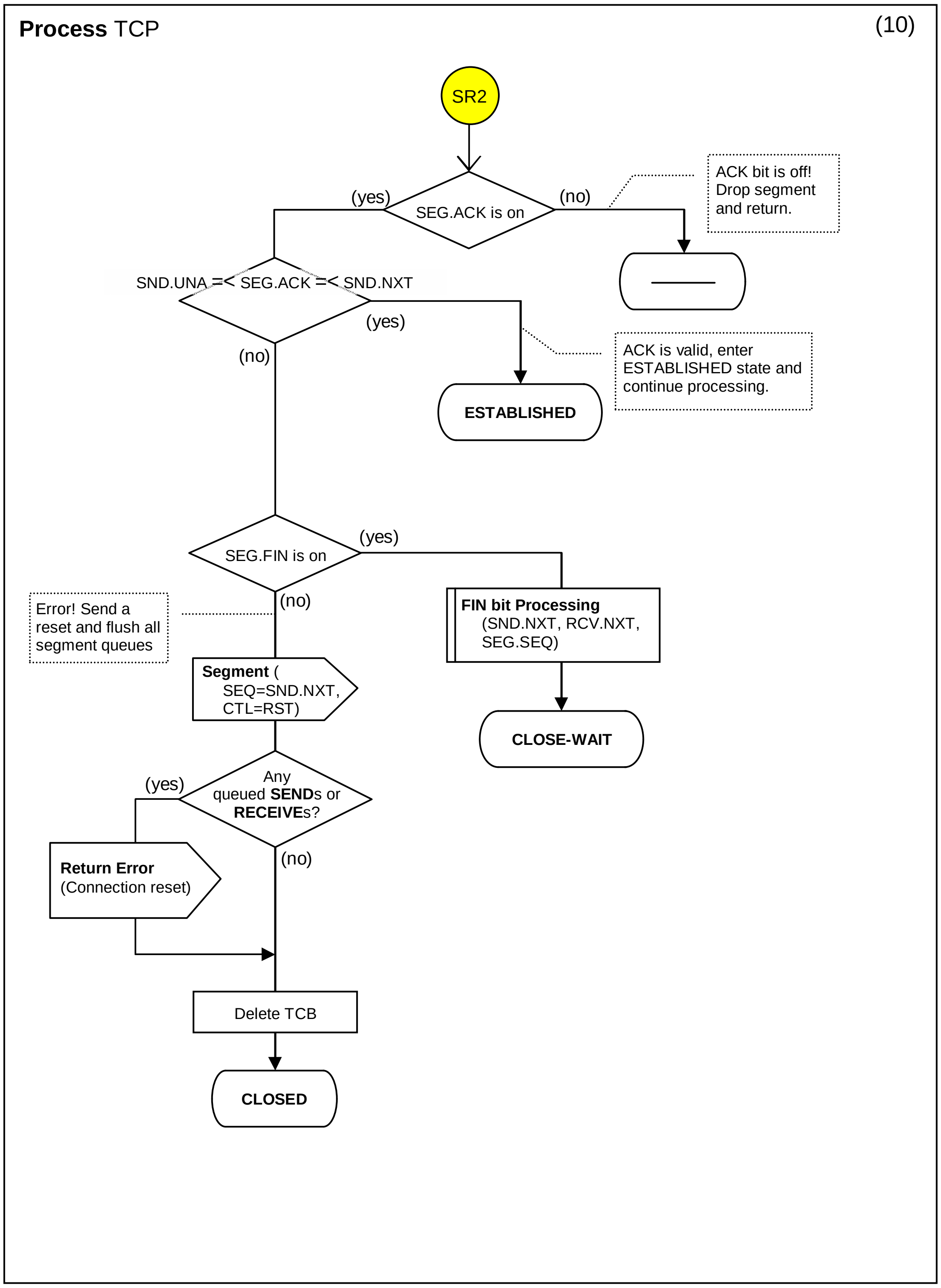

Process TCP
(6)
SYN-SENT
...
...
OPEN Call
Return Error (Connection already exists)
SEND Call
Queue data for transmission during the ESTABLISHED state.
RECEIVE Call
Queue for processing during the ESTABLISHED state.
CLOSE Call
Any queued SENDs or RECEIVEs?
(yes)
(no)
Return Error (Connection closing)
Delete TCB
CLOSED
ABORT Call
Any queued SENDs or RECEIVEs?
(yes)
(no)
Return Error (Connection reset)
Delete TCB
CLOSED
REXMT TIMEOUT
CalcRTO (RTO)
SET (RTO, REXMT)
Segment (From Rexmt Queue)

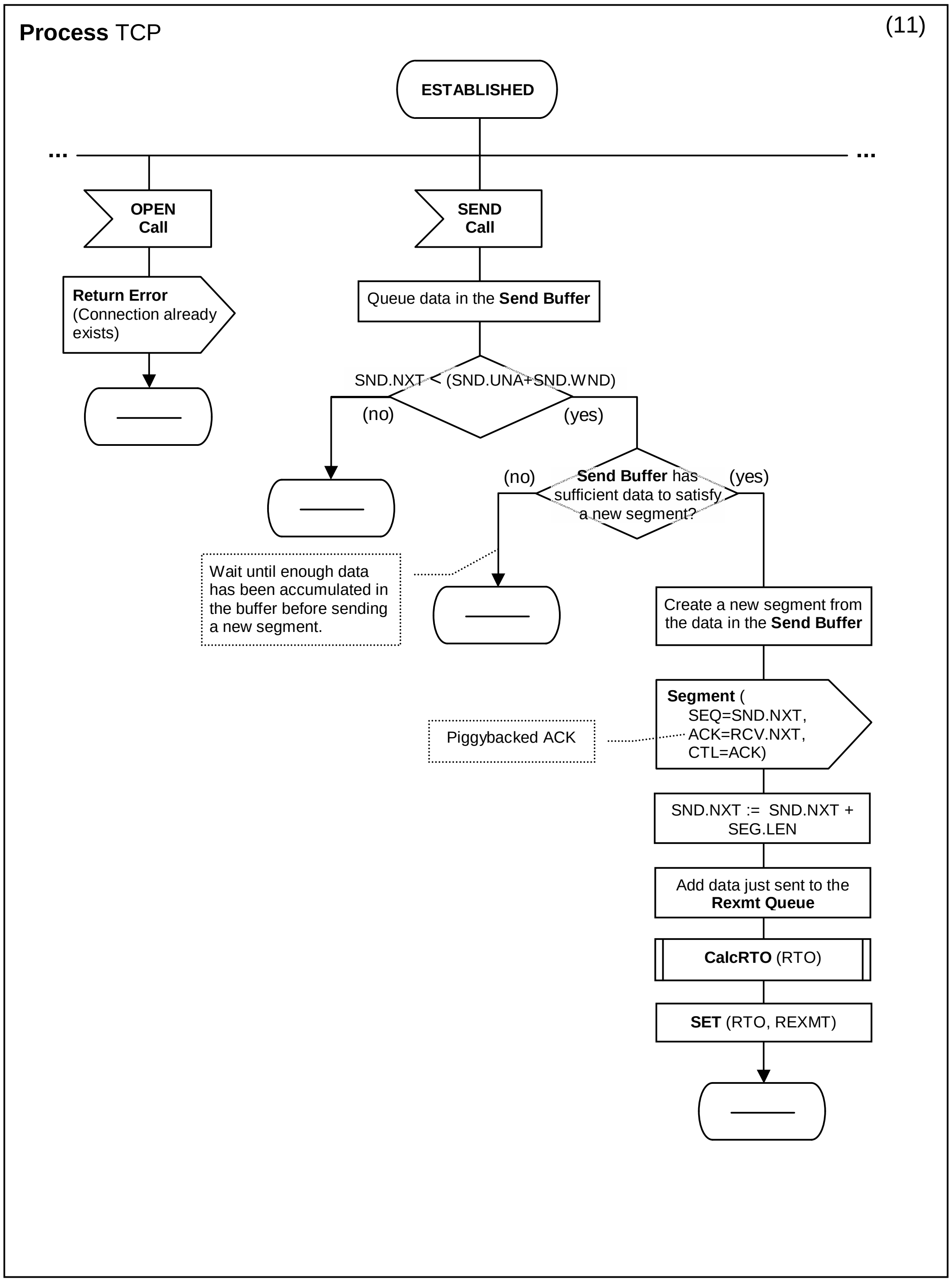

Process TCP
(7)
SYN-SENT
...
...
SEGMENT ARRIVE
SEG.ACK is on?
(no)
(yes)
SEG.RST is on?
(no)
(yes)
(SEG.ACK =< ISS) OR (SEG.ACK > SND.NXT)
(no)
(yes)
(SEG.ACK >= SND.UNA) AND (SEG.ACK <= SND.NXT)
(no)
(yes)
ACK is not acceptable
Segment (SEQ=SEG.ACK, CTL=RST)
No ACK and no RST
Acceptable ACK and no RST
SEG.SYN is on?
(yes)
(no)
SEG.RST is on?
(no)
(yes)
SEG.SYN is on?
(no)
(yes)
Return Error (Connection reset)
Delete TCB
CLOSED
RCV.NXT := SEG.SEQ+1
IRS := SEG.SEQ
RCV.NXT := SEG.SEQ+1
IRS := SEG.SEQ
SND.UNA := SEG.ACK
Remove ACKed segments from the Rexmt Queue
Segment (SEQ=ISS, ACK=RCV.NXT, CTL=SYN,ACK)
SND.UNA > ISS
(no)
(yes)
Our SYN has been ACKed. Acknowledge the received SYN and move to ESTABLISHED.
Queue any data or controls for processing during the ESTABLISHED state
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
May include data or controls queued for transmission.
SYN-RECEIVED
ESTABLISHED

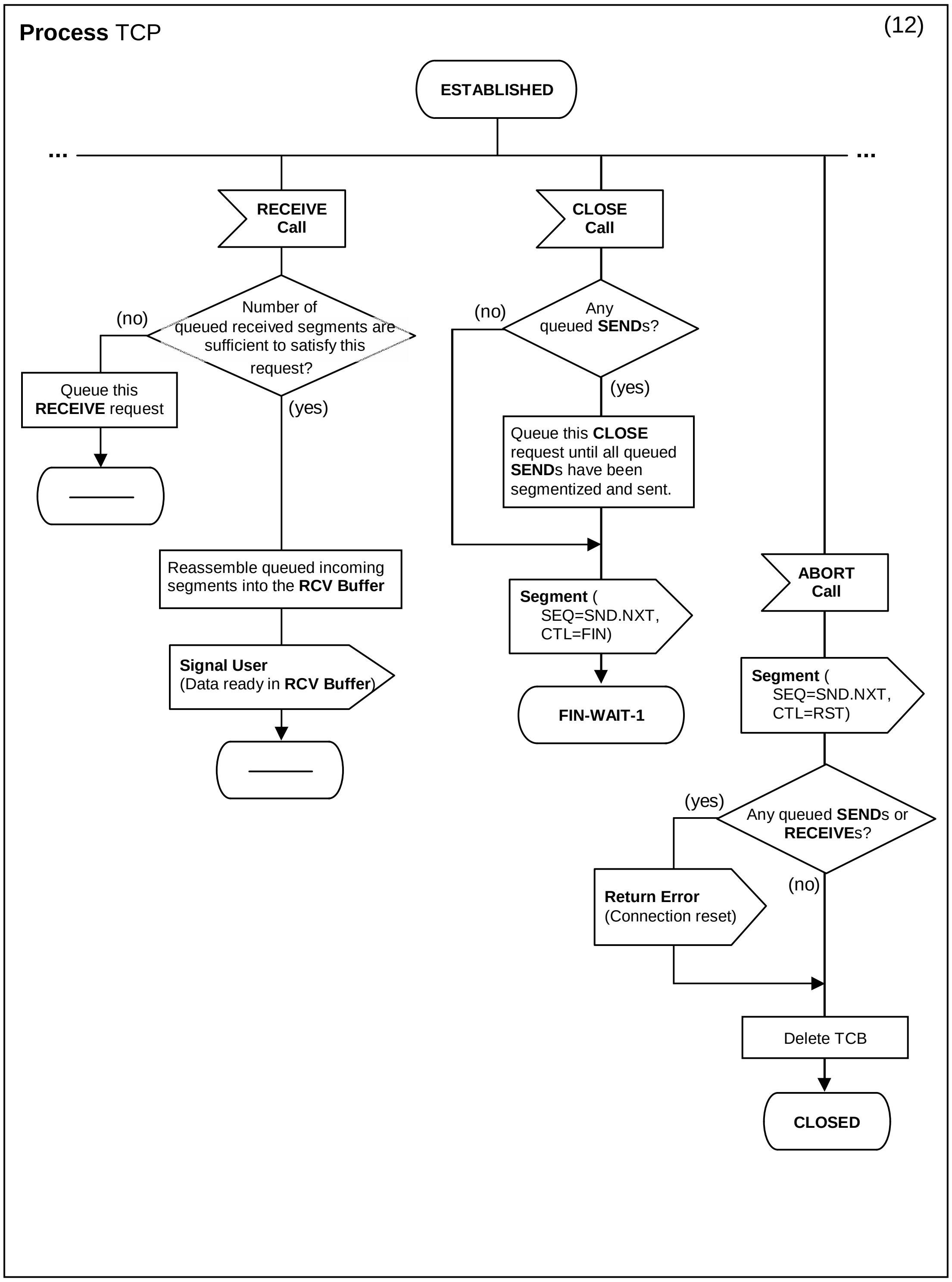


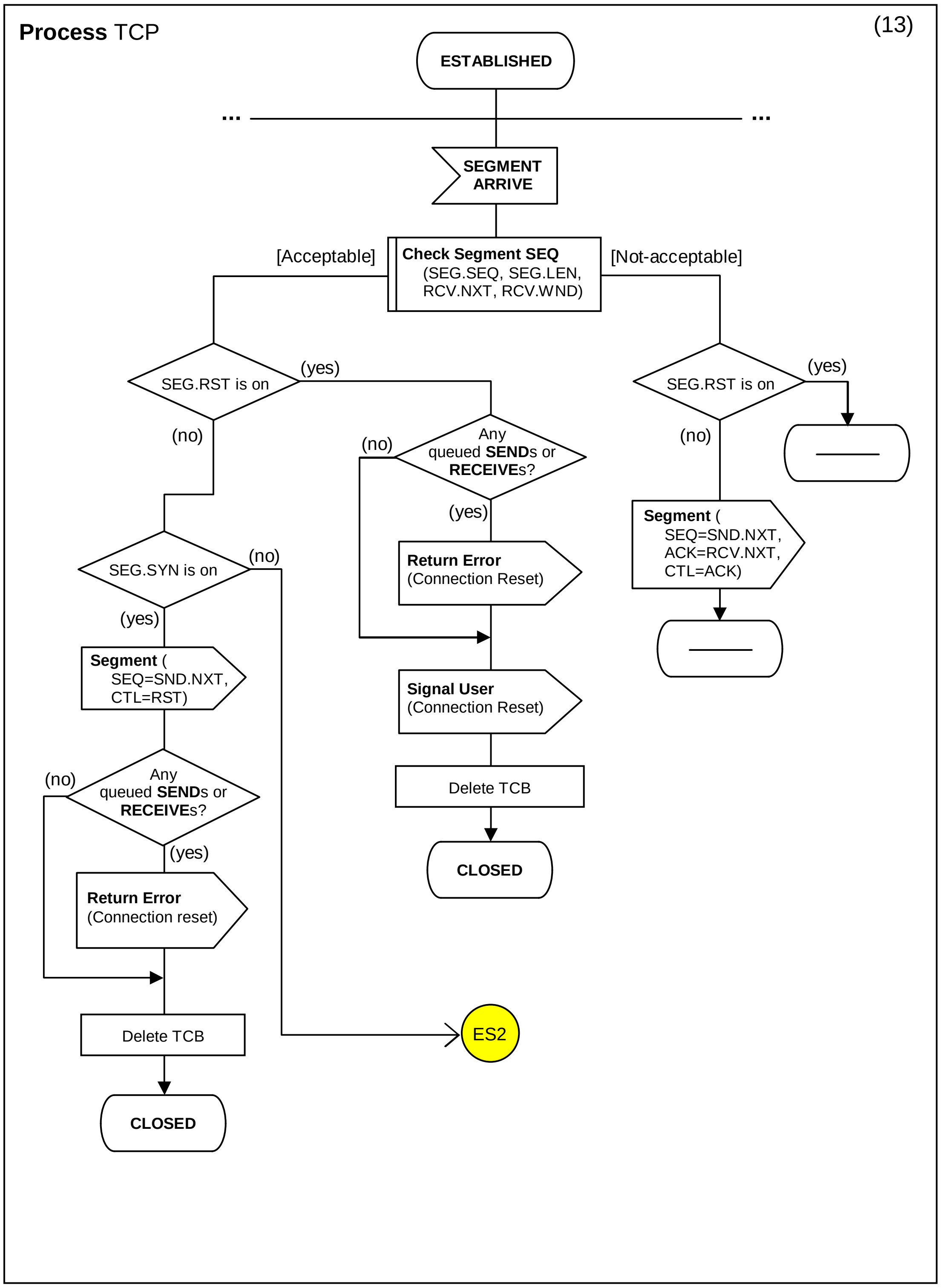
Process TCP
(8)
SYN-RECEIVED
...
...
OPEN Call
Return Error (Connection already exists)
SEND Call
Queue data for transmission during the ESTABLISHED state.
RECEIVE Call
Queue request for processing during the ESTABLISHED state.
CLOSE Call
(any new SENDs issued) OR (any queued SENDs)
(yes)
(no)
Segment (SEQ=SND.NXT, CTL=FIN)
FIN-WAIT-1
Queue for processing after entering the ESTABLISHED state
ABORT Call
Segment (SEQ=SND.NXT, CTL=RST)
Any queued SENDs or RECEIVEs?
(yes)
(no)
Return Error (Connection reset)
Delete TCB
CLOSED

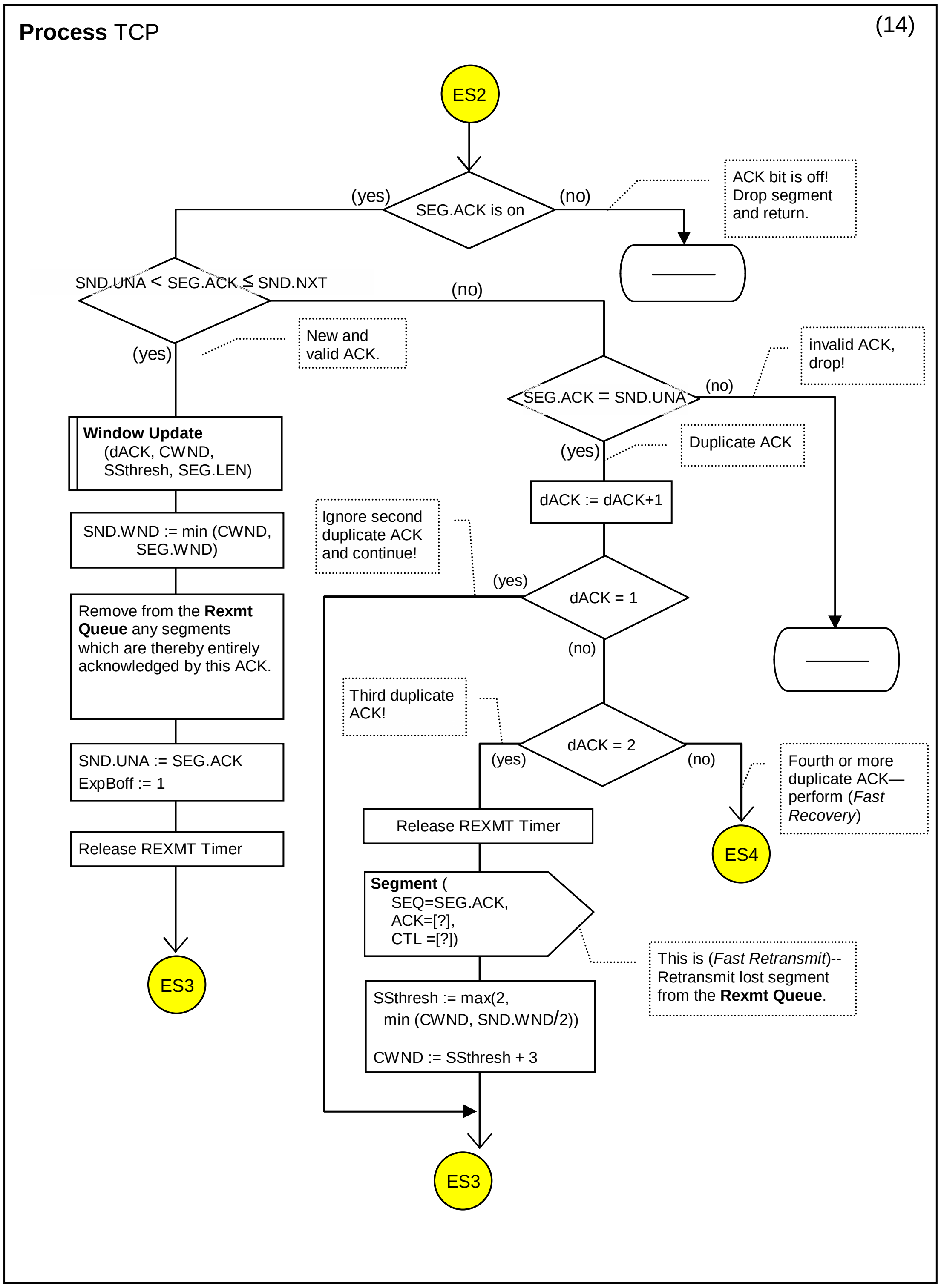

Process TCP
(9)
SYN-RECEIVED
...
...
SEGMENT ARRIVE
REXMT TIMEOUT
CalcRTO (RTO)
SET (RTO, REXMT)
Segment (From Rexmt Queue)
[Acceptable]
Check Segment SEQ (SEG.SEQ, SEG.LEN, RCV.NXT, RCV.WND)
[Not-acceptable]
(no)
SEG.RST is on
(yes)
Connection was initiated with an Active OPEN
(no)
previous state was LISTEN?
(yes)
Connection was initiated with a Passive OPEN
Return Error (Connection refused)
Remove all segments on Rexmt Queue
Delete TCB
CLOSED
Signal User (Returning to LISTEN)
Remove all segments on Rexmt Queue
LISTEN
(no)
SEG.RST is on
(yes)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
(yes)
SEG.SYN is on
(no)
Error! Send a reset and flush all segment queues
Segment (SEQ=SND.NXT, CTL=RST)
SR2
(yes)
Any queued SENDs or RECEIVEs?
(no)
Return Error (Connection reset)
Delete TCB
CLOSED


Process TCP
(10)
SR2
SEG.ACK is on
(yes)
(no)
ACK bit is off! Drop segment and return.
SND.UNA =< SEG.ACK =< SND.NXT
(yes)
ACK is valid, enter ESTABLISHED state and continue processing.
ESTABLISHED
(no)
SEG.FIN is on
(yes)
FIN bit Processing (SND.NXT, RCV.NXT, SEG.SEQ)
CLOSE-WAIT
(no)
Error! Send a reset and flush all segment queues
Segment (SEQ=SND.NXT, CTL=RST)
Any queued SENDs or RECEIVEs?
(yes)
Return Error (Connection reset)
(no)
Delete TCB
CLOSED

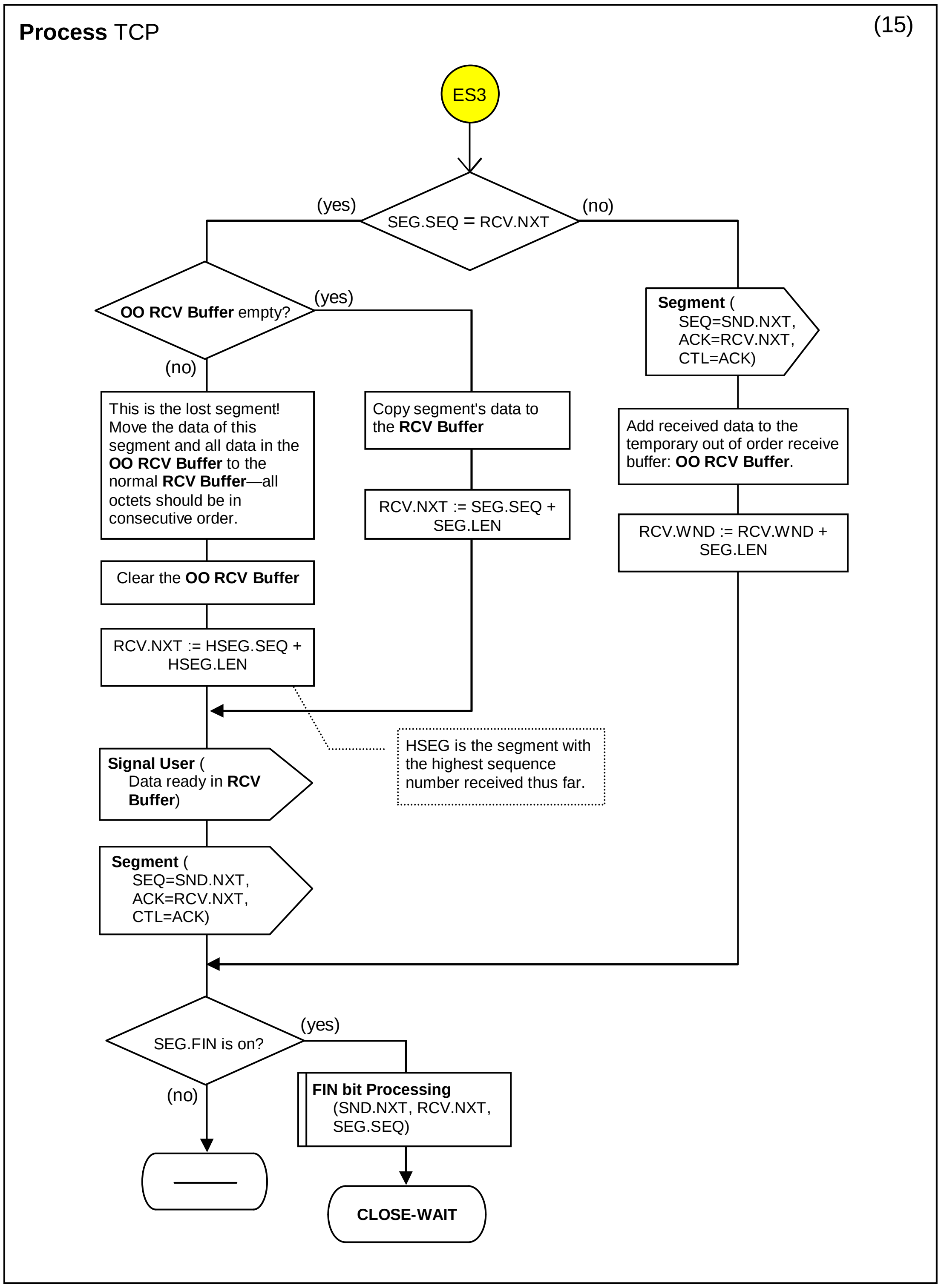

Process TCP
(11)
ESTABLISHED
...
...
OPEN Call
Return Error (Connection already exists)
SEND Call
Queue data in the Send Buffer
SND.NXT < (SND.UNA+SND.WND)
(no)
(yes)
Send Buffer has sufficient data to satisfy a new segment?
(no)
(yes)
Wait until enough data has been accumulated in the buffer before sending a new segment.
Create a new segment from the data in the Send Buffer
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
Piggybacked ACK
SND.NXT := SND.NXT + SEG.LEN
Add data just sent to the Rexmt Queue
CalcRTO (RTO)
SET (RTO, REXMT)


Process TCP
(12)
ESTABLISHED
...
...
RECEIVE Call
Number of queued received segments are sufficient to satisfy this request?
(no)
(yes)
Queue this RECEIVE request
Reassemble queued incoming segments into the RCV Buffer
Signal User (Data ready in RCV Buffer)
CLOSE Call
Any queued SENDs?
(no)
(yes)
Queue this CLOSE request until all queued SENDs have been segmentized and sent.
Segment (SEQ=SND.NXT, CTL=FIN)
FIN-WAIT-1
ABORT Call
Segment (SEQ=SND.NXT, CTL=RST)
Any queued SENDs or RECEIVEs?
(yes)
(no)
Return Error (Connection reset)
Delete TCB
CLOSED

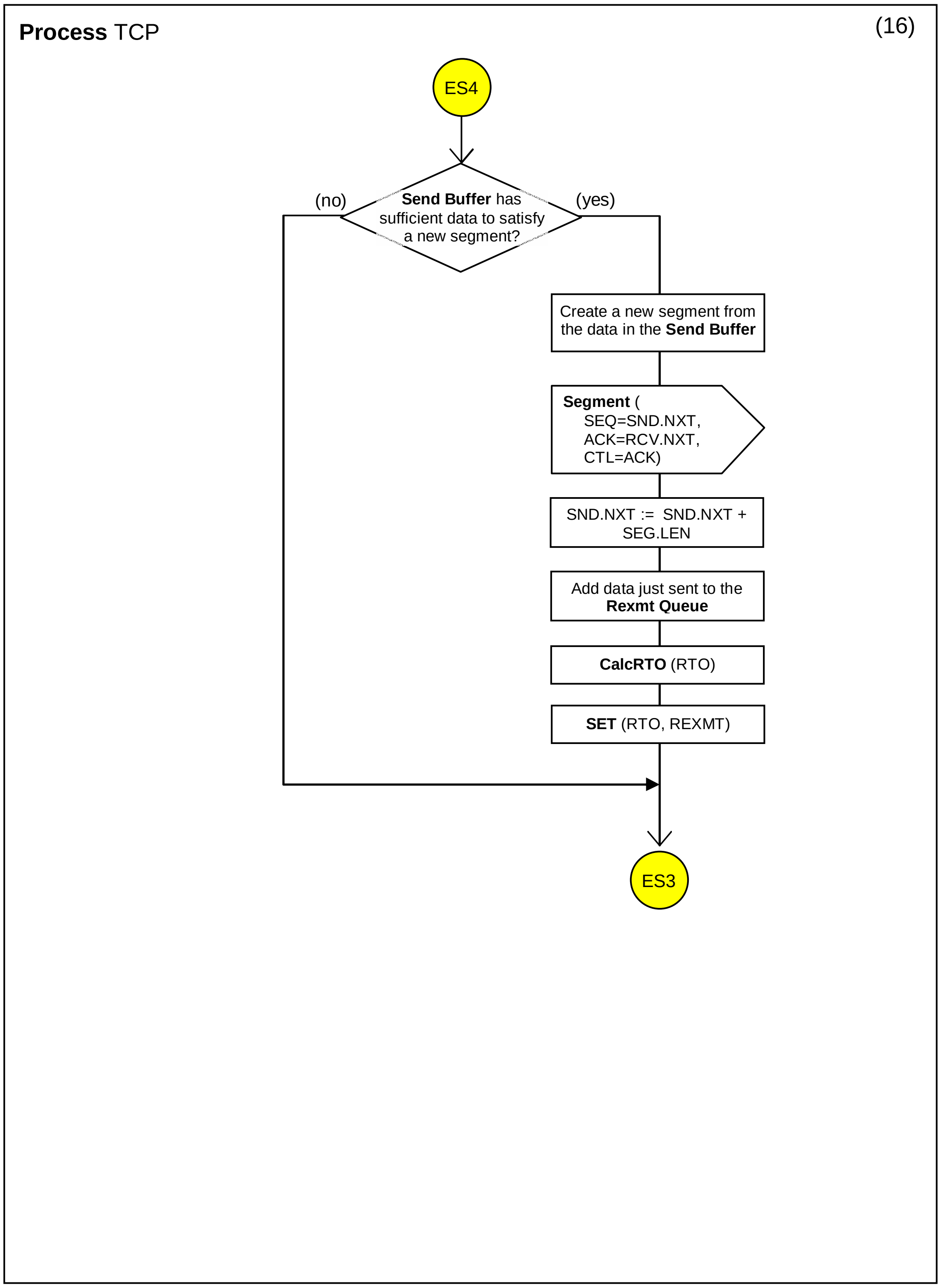

Process TCP
(13)
ESTABLISHED
...
...
SEGMENT ARRIVE
[Acceptable]
Check Segment SEQ (SEG.SEQ, SEG.LEN, RCV.NXT, RCV.WND)
[Not-acceptable]
SEG.RST is on
(yes)
(no)
Any queued SENDs or RECEIVEs?
(no)
(yes)
Return Error (Connection Reset)
Signal User (Connection Reset)
Delete TCB
CLOSED
SEG.SYN is on
(no)
(yes)
Segment (SEQ=SND.NXT, CTL=RST)
Any queued SENDs or RECEIVEs?
(no)
(yes)
Return Error (Connection reset)
Delete TCB
CLOSED
ES2
SEG.RST is on
(yes)
(no)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)

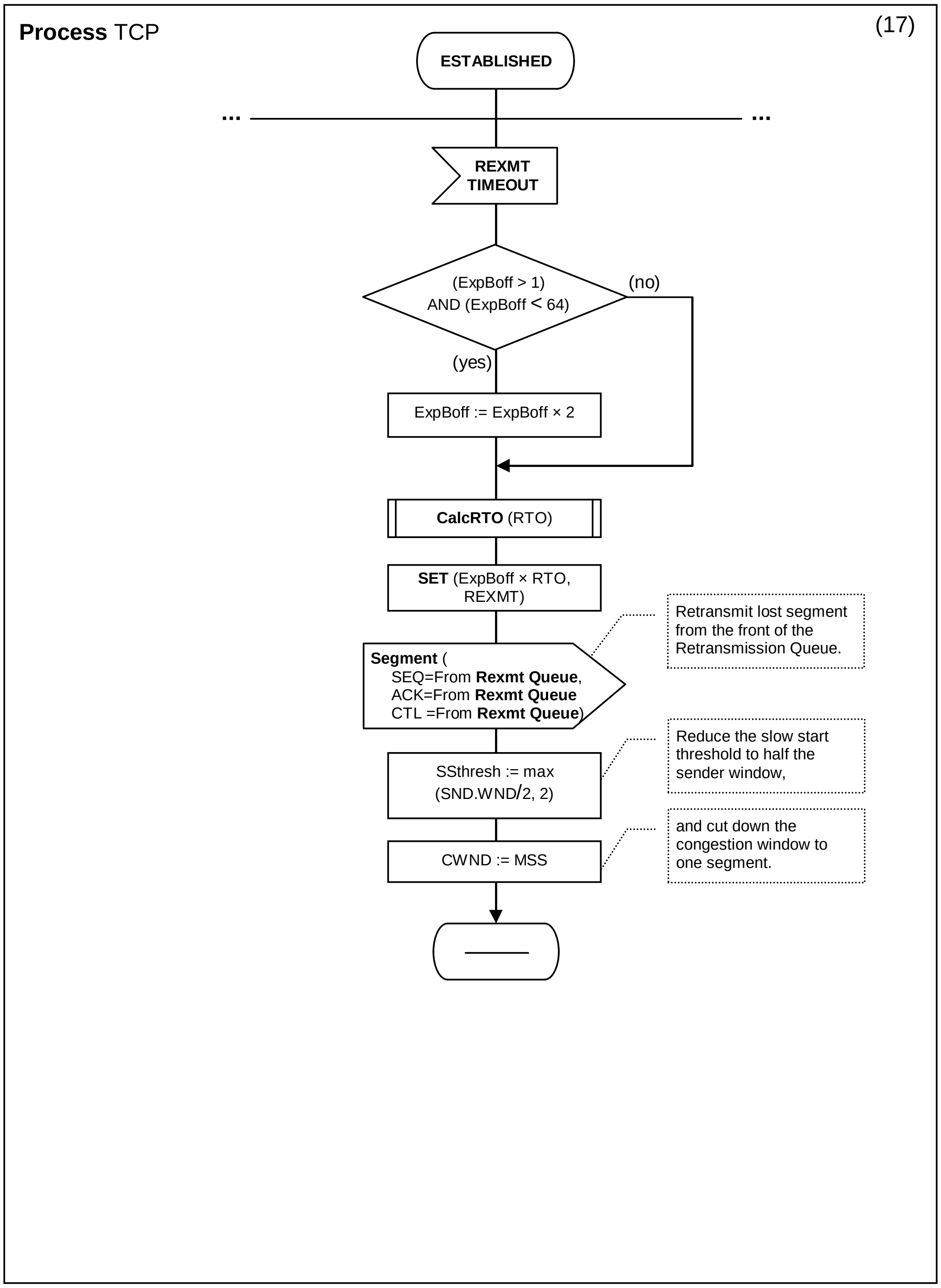


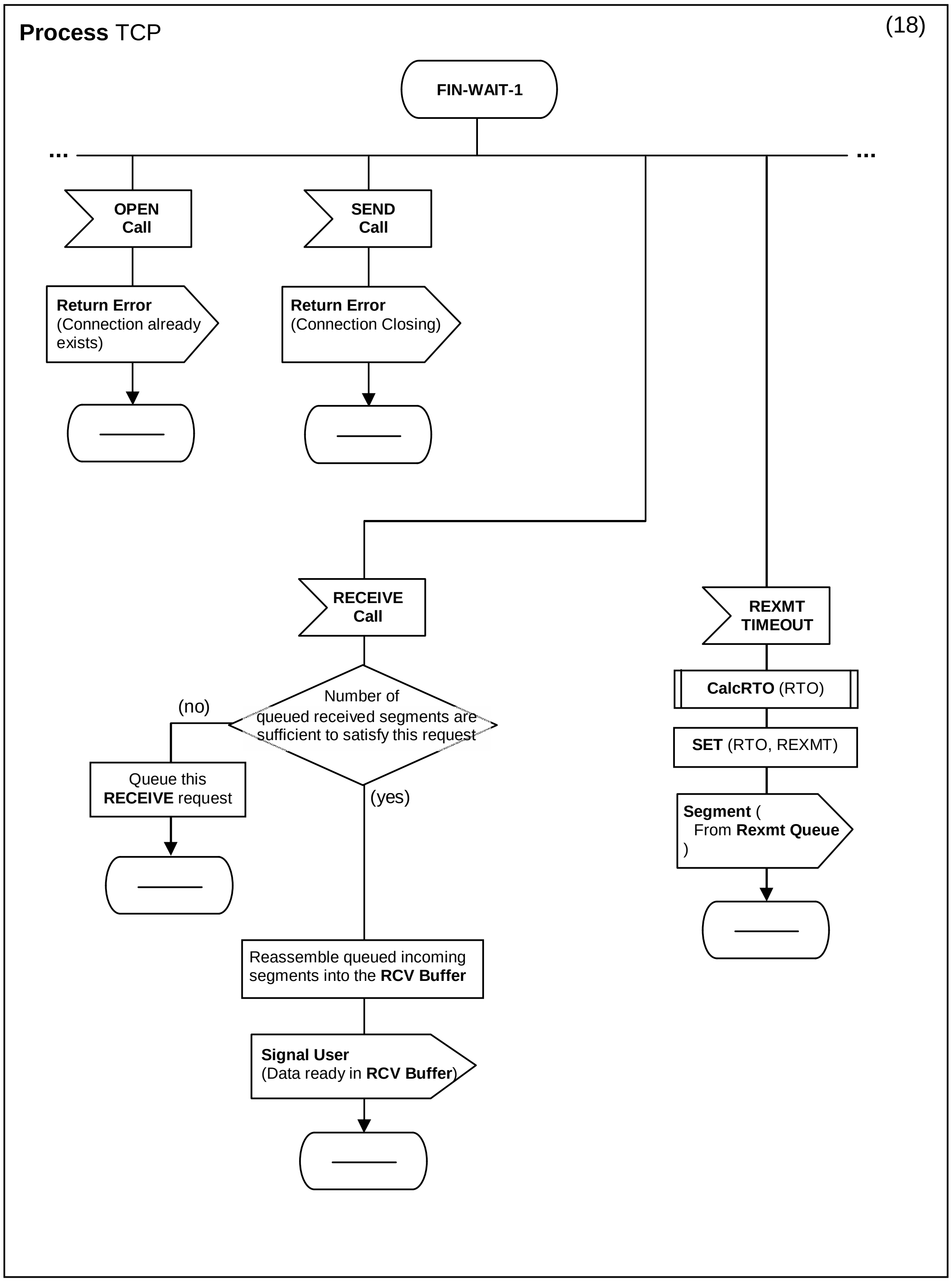
Process TCP
(14)
ES2
SEG.ACK is on
(yes)
(no)
ACK bit is off! Drop segment and return.
SND.UNA < SEG.ACK ≤ SND.NXT
(no)
(yes)
New and valid ACK.
SEG.ACK = SND.UNA
(no)
invalid ACK, drop!
(yes)
Duplicate ACK
Window Update (dACK, CWND, SSthresh, SEG.LEN)
dACK := dACK+1
SND.WND := min (CWND, SEG.WND)
Ignore second duplicate ACK and continue!
(yes)
dACK = 1
(no)
Remove from the **Rexmt Queue** any segments which are thereby entirely acknowledged by this ACK.
Third duplicate ACK!
dACK = 2
(yes)
(no)
Fourth or more duplicate ACK—perform (*Fast Recovery*)
SND.UNA := SEG.ACK
ExpBoff := 1
Release REXMT Timer
ES4
Release REXMT Timer
Segment (SEQ=SEG.ACK, ACK=[?], CTL =[?])
This is (*Fast Retransmit*)-- Retransmit lost segment from the **Rexmt Queue**.
ES3
SSthresh := max(2, min (CWND, SND.WND/2))
CWND := SSthresh + 3
ES3

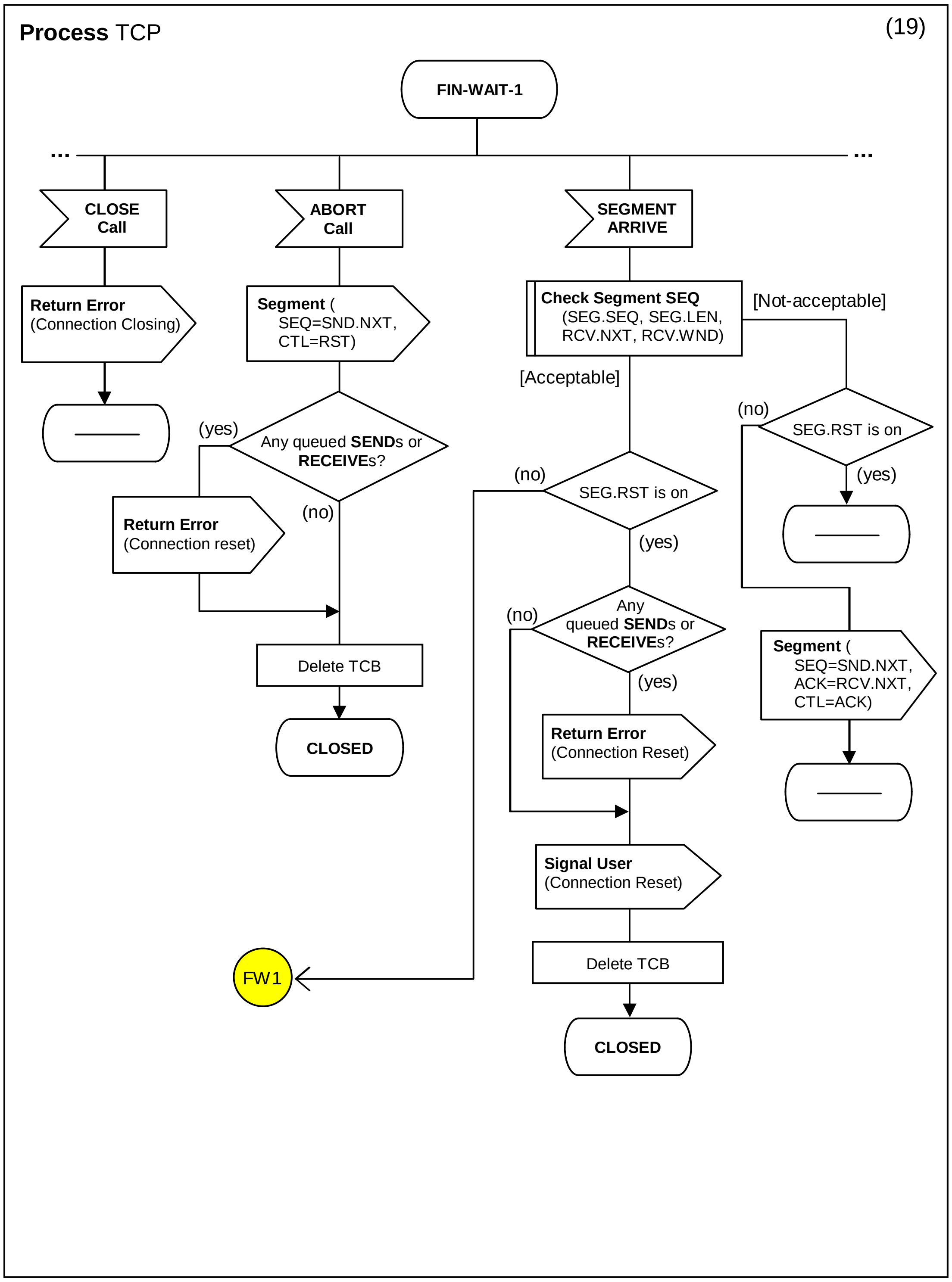


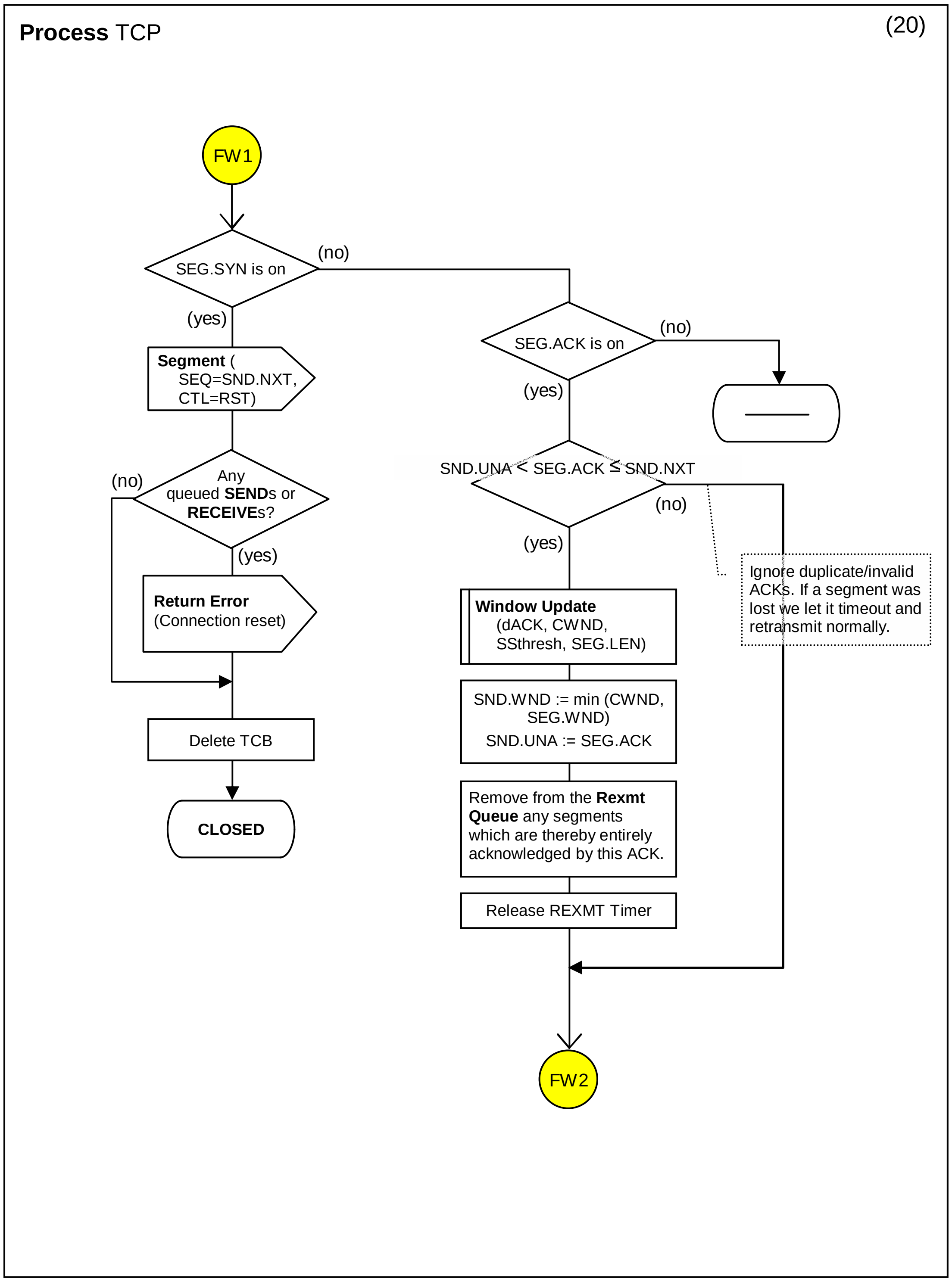
Process TCP
(15)
ES3
SEG.SEQ = RCV.NXT
(yes)
(no)
OO RCV Buffer empty?
(yes)
(no)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
This is the lost segment! Move the data of this segment and all data in the OO RCV Buffer to the normal RCV Buffer—all octets should be in consecutive order.
Copy segment's data to the RCV Buffer
Add received data to the temporary out of order receive buffer: OO RCV Buffer.
RCV.NXT := SEG.SEQ + SEG.LEN
RCV.WND := RCV.WND + SEG.LEN
Clear the OO RCV Buffer
RCV.NXT := HSEG.SEQ + HSEG.LEN
HSEG is the segment with the highest sequence number received thus far.
Signal User (Data ready in RCV Buffer)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
SEG.FIN is on?
(yes)
(no)
FIN bit Processing (SND.NXT, RCV.NXT, SEG.SEQ)
CLOSE-WAIT

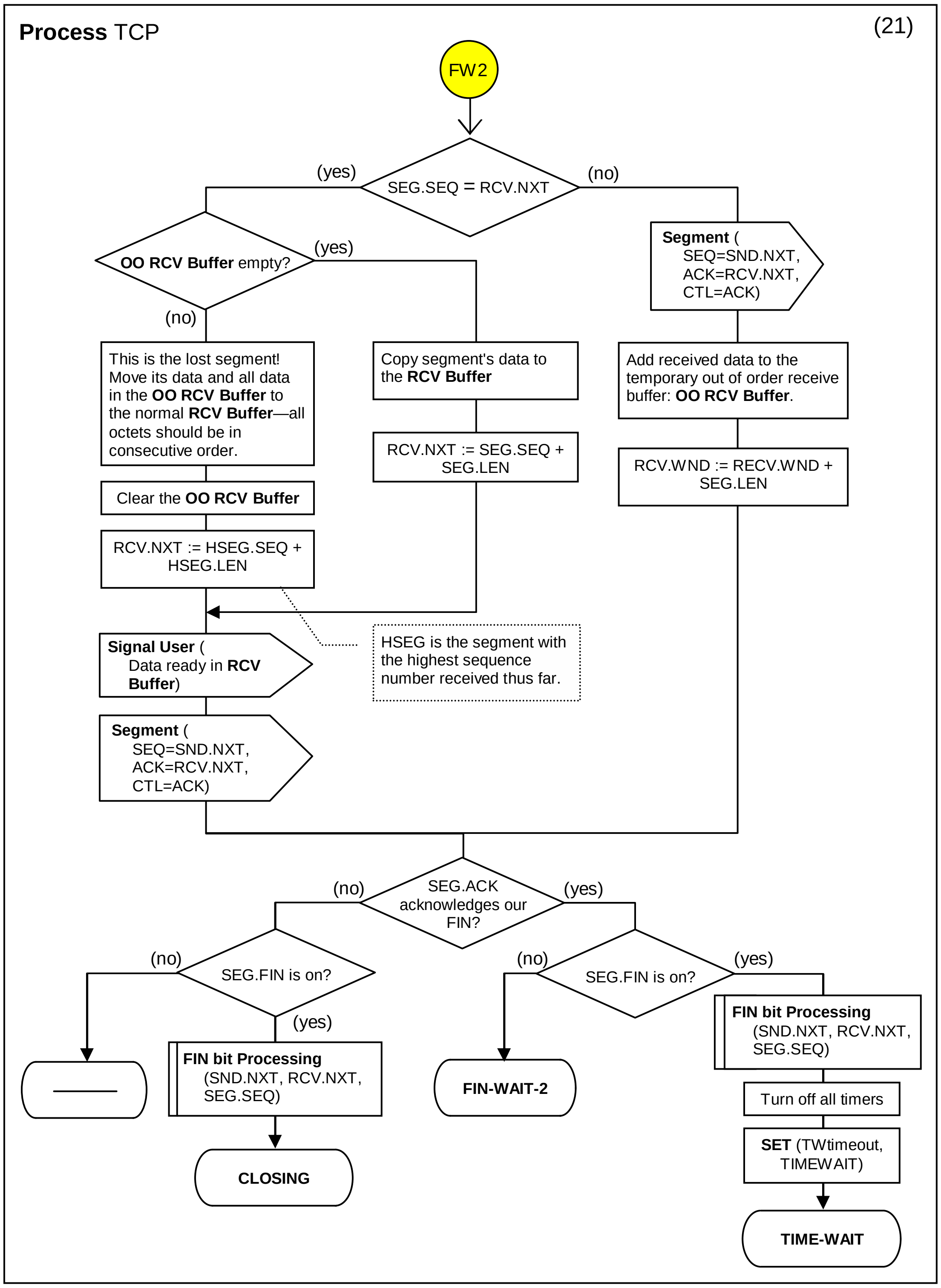

Process TCP
(16)
ES4
Send Buffer has sufficient data to satisfy a new segment?
(no)
(yes)
Create a new segment from the data in the **Send Buffer**
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
SND.NXT := SND.NXT + SEG.LEN
Add data just sent to the **Rexmt Queue**
CalcRTO (RTO)
SET (RTO, REXMT)
ES3

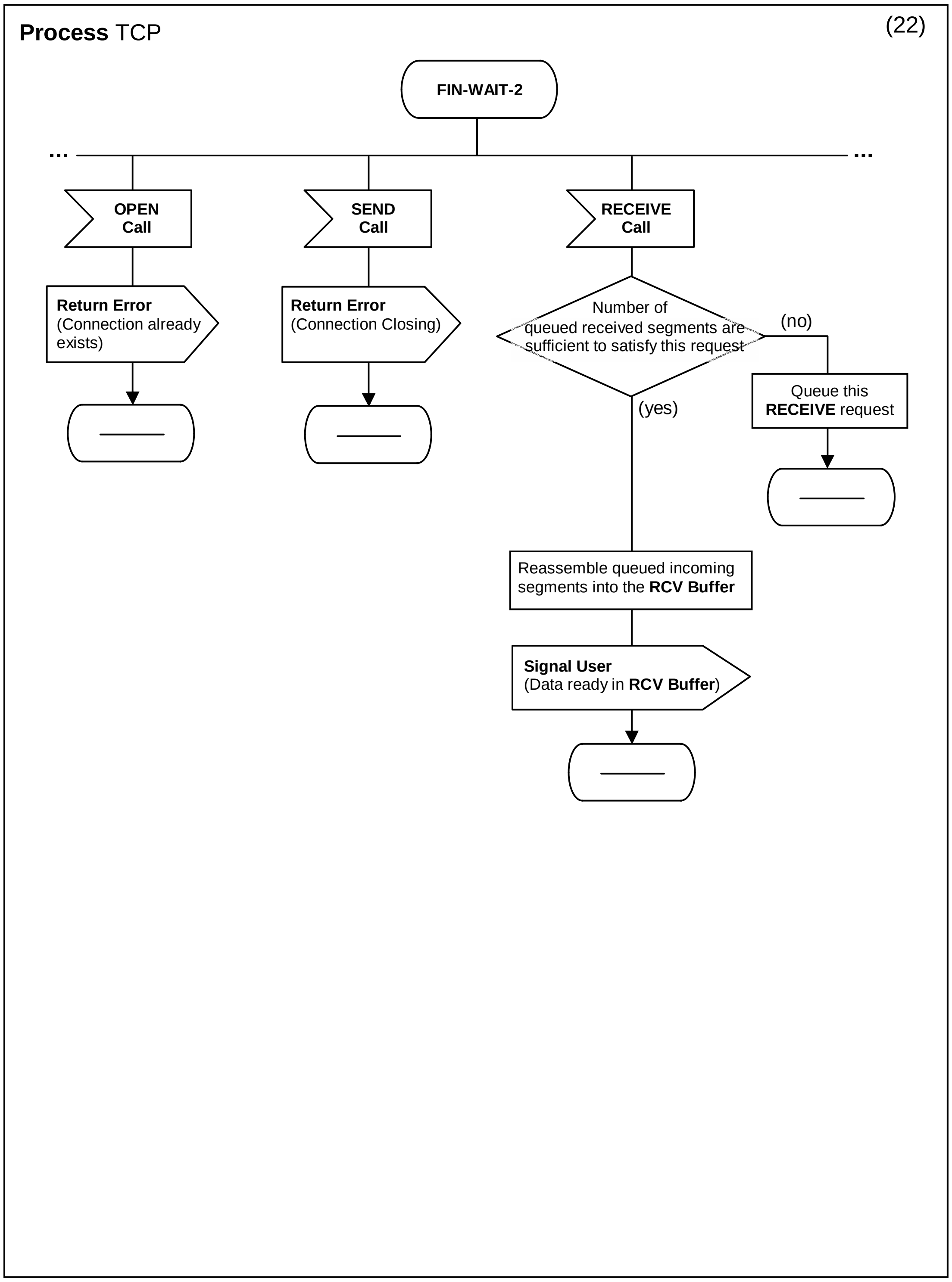

Process TCP
(17)
ESTABLISHED
...
...
REXMT TIMEOUT
(ExpBoff > 1) AND (ExpBoff < 64)
(no)
(yes)
ExpBoff := ExpBoff × 2
CalcRTO (RTO)
SET (ExpBoff × RTO, REXMT)
Segment (SEQ=From Rexmt Queue, ACK=From Rexmt Queue CTL =From Rexmt Queue)
Retransmit lost segment from the front of the Retransmission Queue.
SSthresh := max (SND.WND/2, 2)
Reduce the slow start threshold to half the sender window,
CWND := MSS
and cut down the congestion window to one segment.
—

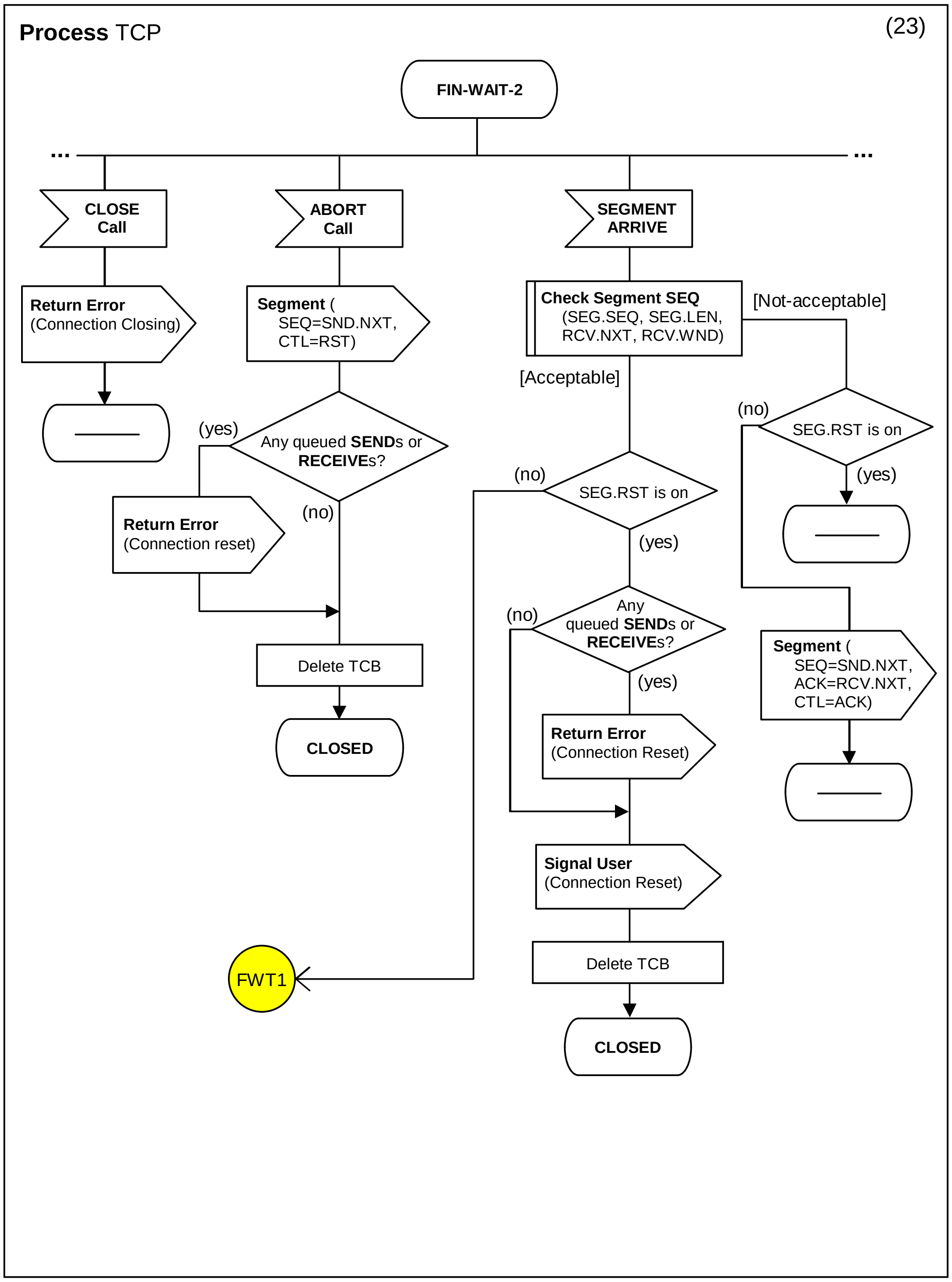

Process TCP
(18)
FIN-WAIT-1
...
...
OPEN Call
SEND Call
Return Error (Connection already exists)
Return Error (Connection Closing)
RECEIVE Call
REXMT TIMEOUT
Number of queued received segments are sufficient to satisfy this request
(no)
(yes)
Queue this RECEIVE request
CalcRTO (RTO)
SET (RTO, REXMT)
Segment (From Rexmt Queue)
Reassemble queued incoming segments into the RCV Buffer
Signal User (Data ready in RCV Buffer)


Process TCP
(19)
FIN-WAIT-1
...
...
CLOSE Call
ABORT Call
SEGMENT ARRIVE
Return Error (Connection Closing)
Segment (SEQ=SND.NXT, CTL=RST)
Check Segment SEQ (SEG.SEQ, SEG.LEN, RCV.NXT, RCV.WND)
[Not-acceptable]
[Acceptable]
(yes)
Any queued SENDs or RECEIVEs?
(no)
Return Error (Connection reset)
Delete TCB
CLOSED
(no)
SEG.RST is on
(yes)
(no)
Any queued SENDs or RECEIVEs?
(yes)
Return Error (Connection Reset)
Signal User (Connection Reset)
Delete TCB
CLOSED
FW1
(no)
SEG.RST is on
(yes)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)

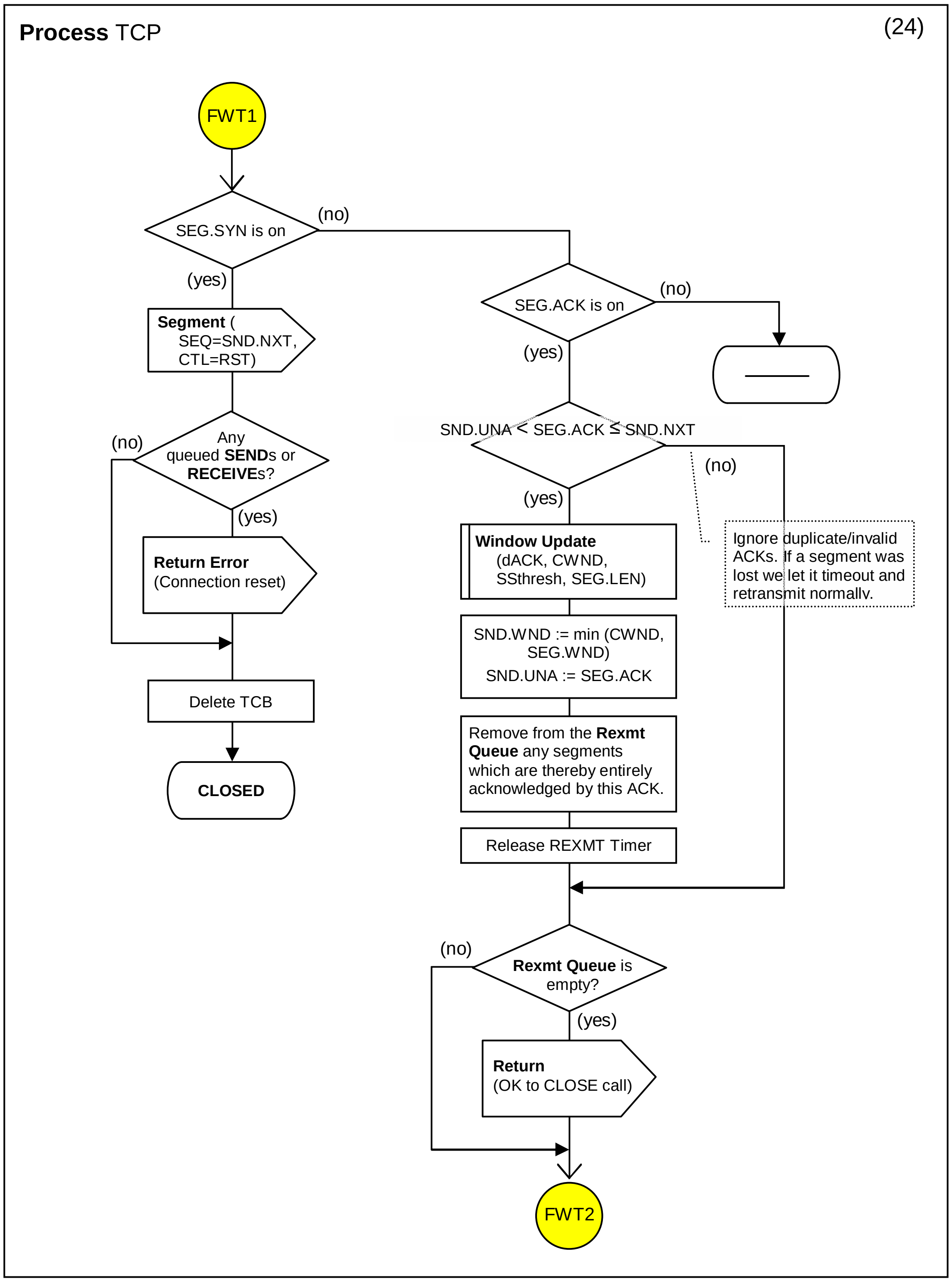

Process TCP
(20)
FW1
SEG.SYN is on
(no)
(yes)
Segment (SEQ=SND.NXT, CTL=RST)
(no)
Any queued SENDs or RECEIVEs?
(yes)
Return Error (Connection reset)
Delete TCB
CLOSED
SEG.ACK is on
(no)
(yes)
SND.UNA < SEG.ACK ≤ SND.NXT
(no)
(yes)
Ignore duplicate/invalid ACKs. If a segment was lost we let it timeout and retransmit normally.
Window Update (dACK, CWND, SSthresh, SEG.LEN)
SND.WND := min (CWND, SEG.WND)
SND.UNA := SEG.ACK
Remove from the Rexmt Queue any segments which are thereby entirely acknowledged by this ACK.
Release REXMT Timer
FW2


Process TCP
(21)
FW2
SEG.SEQ = RCV.NXT
(yes)
(no)
OO RCV Buffer empty?
(yes)
(no)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
This is the lost segment! Move its data and all data in the **OO RCV Buffer** to the normal **RCV Buffer**—all octets should be in consecutive order.
Copy segment's data to the **RCV Buffer**
Add received data to the temporary out of order receive buffer: **OO RCV Buffer**.
RCV.NXT := SEG.SEQ + SEG.LEN
RCV.WND := RECV.WND + SEG.LEN
Clear the **OO RCV Buffer**
RCV.NXT := HSEG.SEQ + HSEG.LEN
HSEG is the segment with the highest sequence number received thus far.
Signal User (Data ready in **RCV Buffer**)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
SEG.ACK acknowledges our FIN?
(no)
(yes)
SEG.FIN is on?
(no)
(yes)
SEG.FIN is on?
(no)
(yes)
FIN bit Processing (SND.NXT, RCV.NXT, SEG.SEQ)
FIN bit Processing (SND.NXT, RCV.NXT, SEG.SEQ)
——
CLOSING
FIN-WAIT-2
Turn off all timers
SET (TWtimeout, TIMEWAIT)
TIME-WAIT

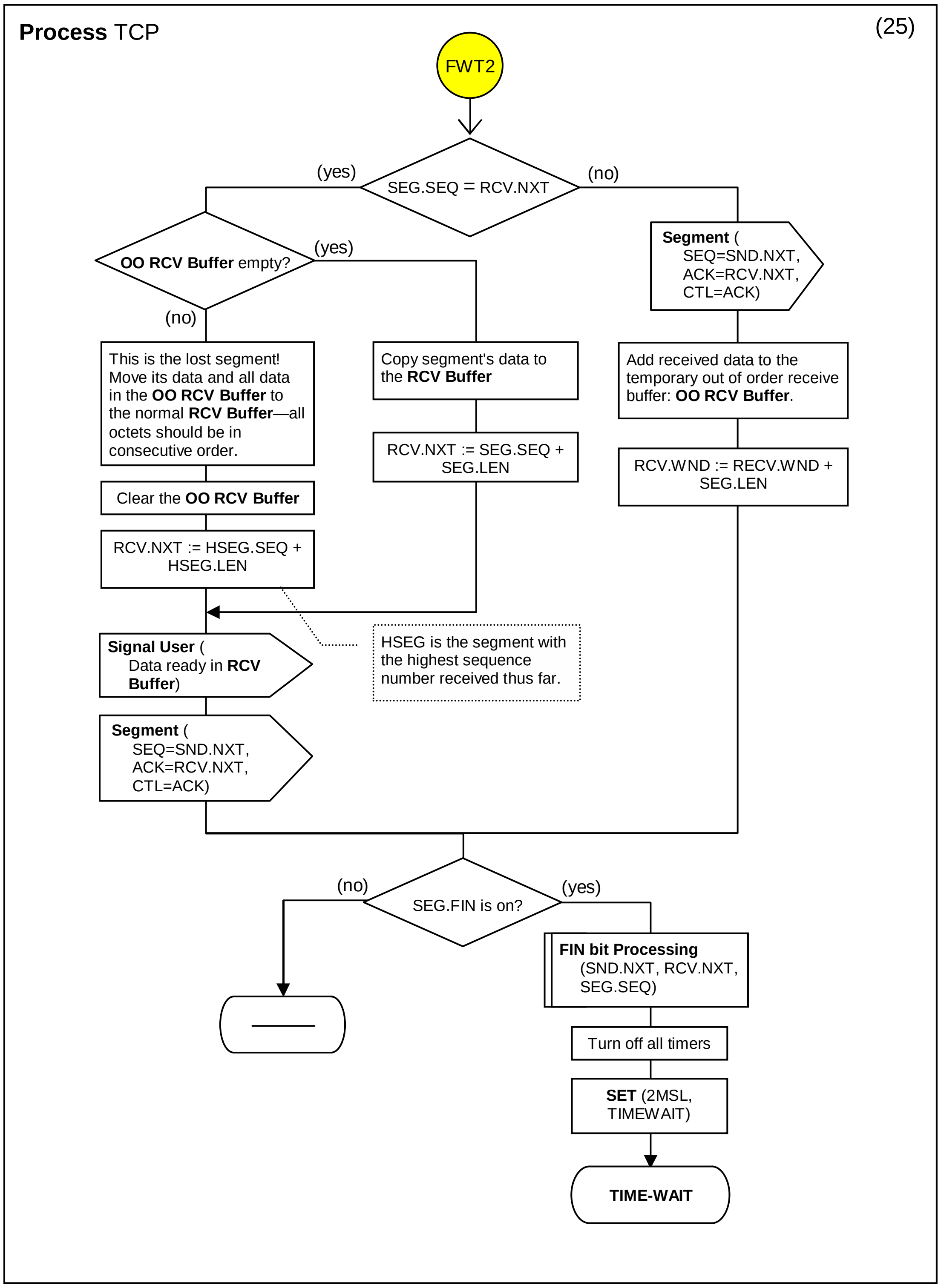

Process TCP
(22)
FIN-WAIT-2
...
...
OPEN Call
SEND Call
RECEIVE Call
Return Error (Connection already exists)
Return Error (Connection Closing)
Number of queued received segments are sufficient to satisfy this request
(no)
(yes)
Queue this RECEIVE request
Reassemble queued incoming segments into the RCV Buffer
Signal User (Data ready in RCV Buffer)

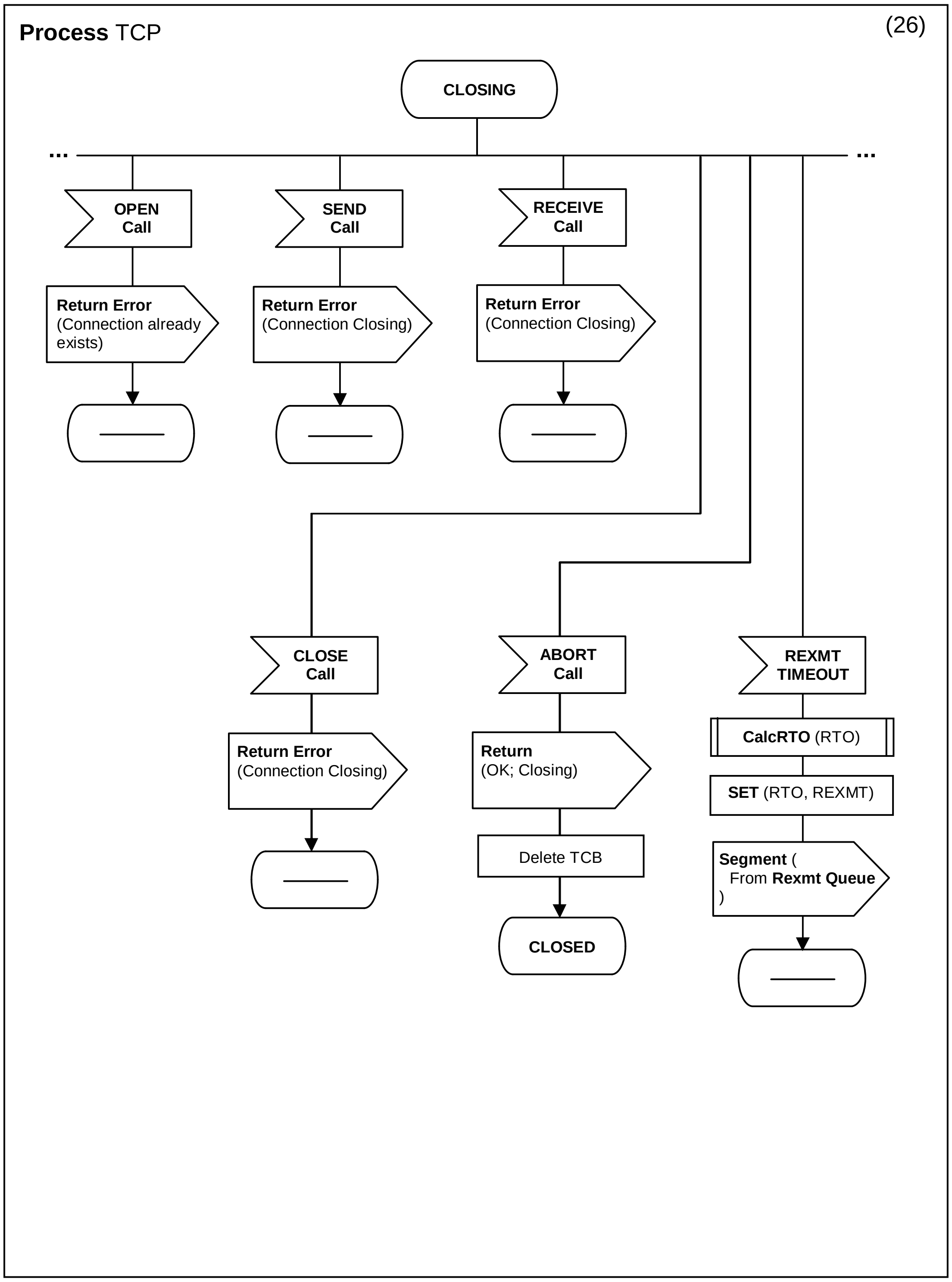


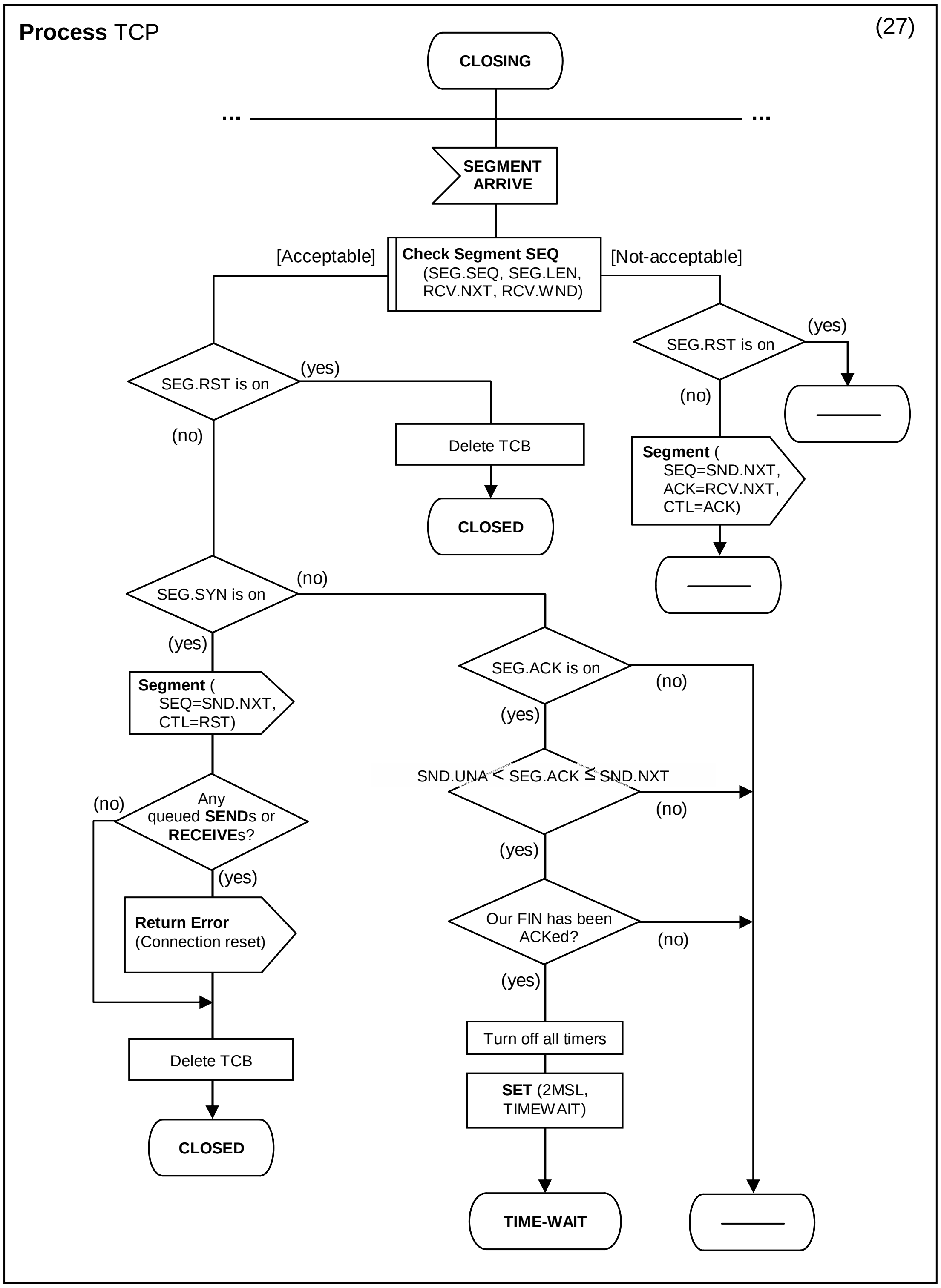
Process TCP
(23)
FIN-WAIT-2
...
...
CLOSE Call
ABORT Call
SEGMENT ARRIVE
Return Error (Connection Closing)
Segment (SEQ=SND.NXT, CTL=RST)
Check Segment SEQ (SEG.SEQ, SEG.LEN, RCV.NXT, RCV.WND)
[Not-acceptable]
[Acceptable]
(yes)
Any queued SENDs or RECEIVEs?
(no)
Return Error (Connection reset)
Delete TCB
CLOSED
(no)
SEG.RST is on
(yes)
(no)
Any queued SENDs or RECEIVEs?
(yes)
Return Error (Connection Reset)
Signal User (Connection Reset)
Delete TCB
CLOSED
(no)
SEG.RST is on
(yes)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
FWT1

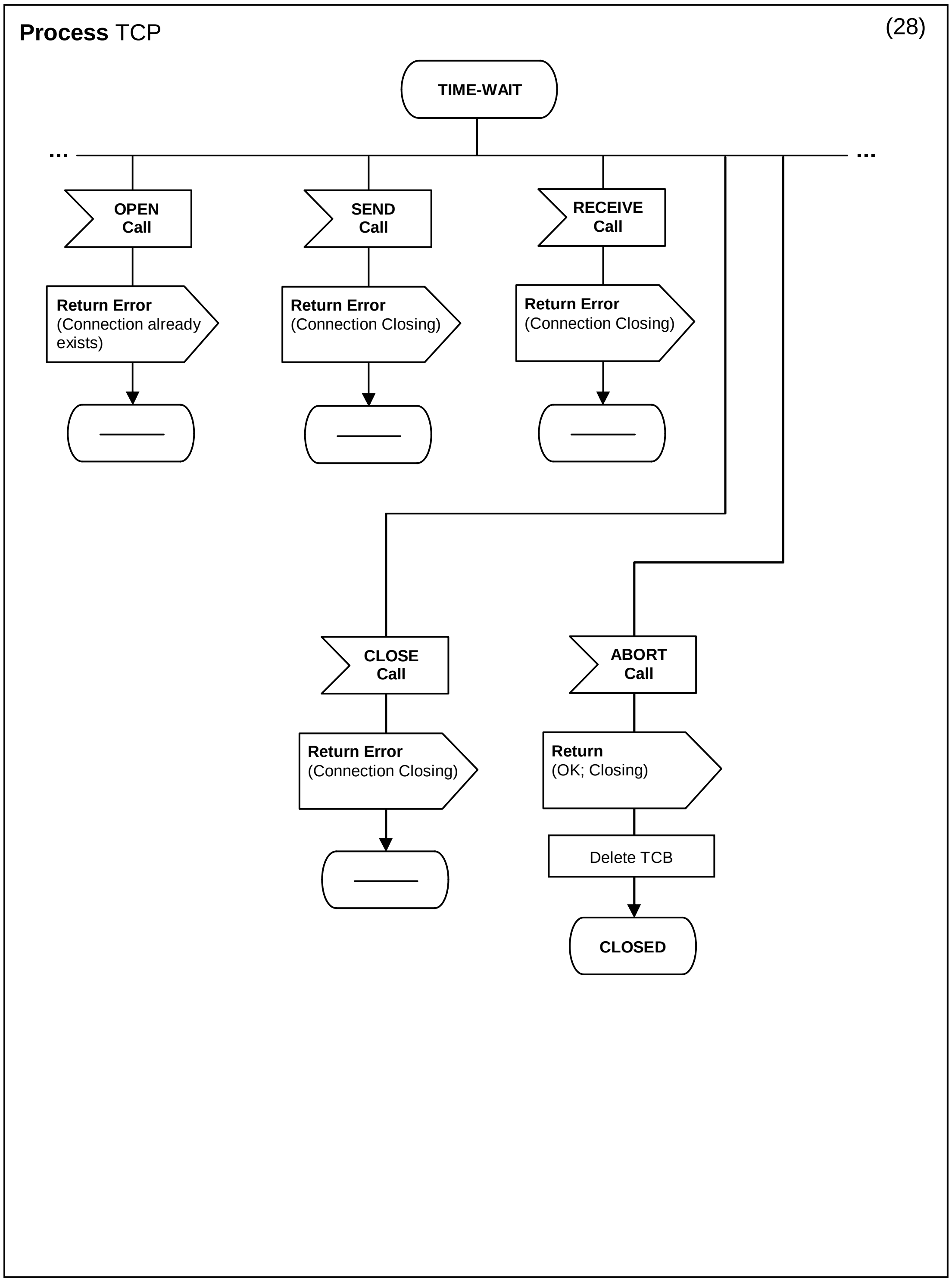


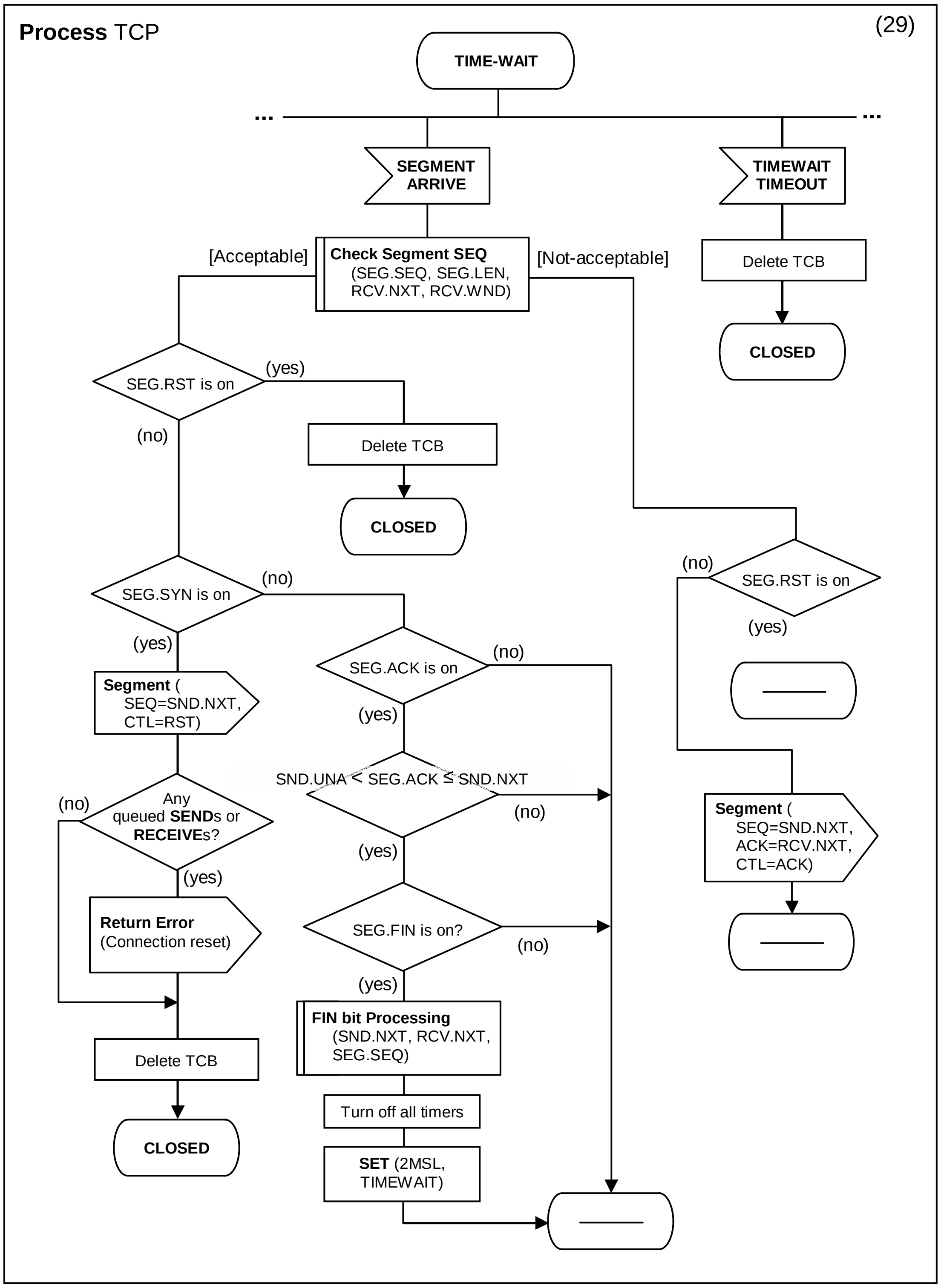
Process TCP
(24)
FWT1
SEG.SYN is on
(no)
(yes)
Segment (SEQ=SND.NXT, CTL=RST)
Any queued SENDs or RECEIVEs?
(no)
(yes)
Return Error (Connection reset)
Delete TCB
CLOSED
SEG.ACK is on
(no)
(yes)
SND.UNA < SEG.ACK ≤ SND.NXT
(no)
(yes)
Ignore duplicate/invalid ACKs. If a segment was lost we let it timeout and retransmit normally.
Window Update (dACK, CWND, SSthresh, SEG.LEN)
SND.WND := min (CWND, SEG.WND)
SND.UNA := SEG.ACK
Remove from the Rexmt Queue any segments which are thereby entirely acknowledged by this ACK.
Release REXMT Timer
Rexmt Queue is empty?
(no)
(yes)
Return (OK to CLOSE call)
FWT2


Process TCP
(25)
FWT2
SEG.SEQ = RCV.NXT
(yes)
(no)
OO RCV Buffer empty?
(yes)
(no)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
This is the lost segment! Move its data and all data in the OO RCV Buffer to the normal RCV Buffer—all octets should be in consecutive order.
Copy segment's data to the RCV Buffer
Add received data to the temporary out of order receive buffer: OO RCV Buffer.
RCV.NXT := SEG.SEQ + SEG.LEN
RCV.WND := RECV.WND + SEG.LEN
Clear the OO RCV Buffer
RCV.NXT := HSEG.SEQ + HSEG.LEN
HSEG is the segment with the highest sequence number received thus far.
Signal User (Data ready in RCV Buffer)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
SEG.FIN is on?
(no)
(yes)
FIN bit Processing (SND.NXT, RCV.NXT, SEG.SEQ)
Turn off all timers
SET (2MSL, TIMEWAIT)
TIME-WAIT

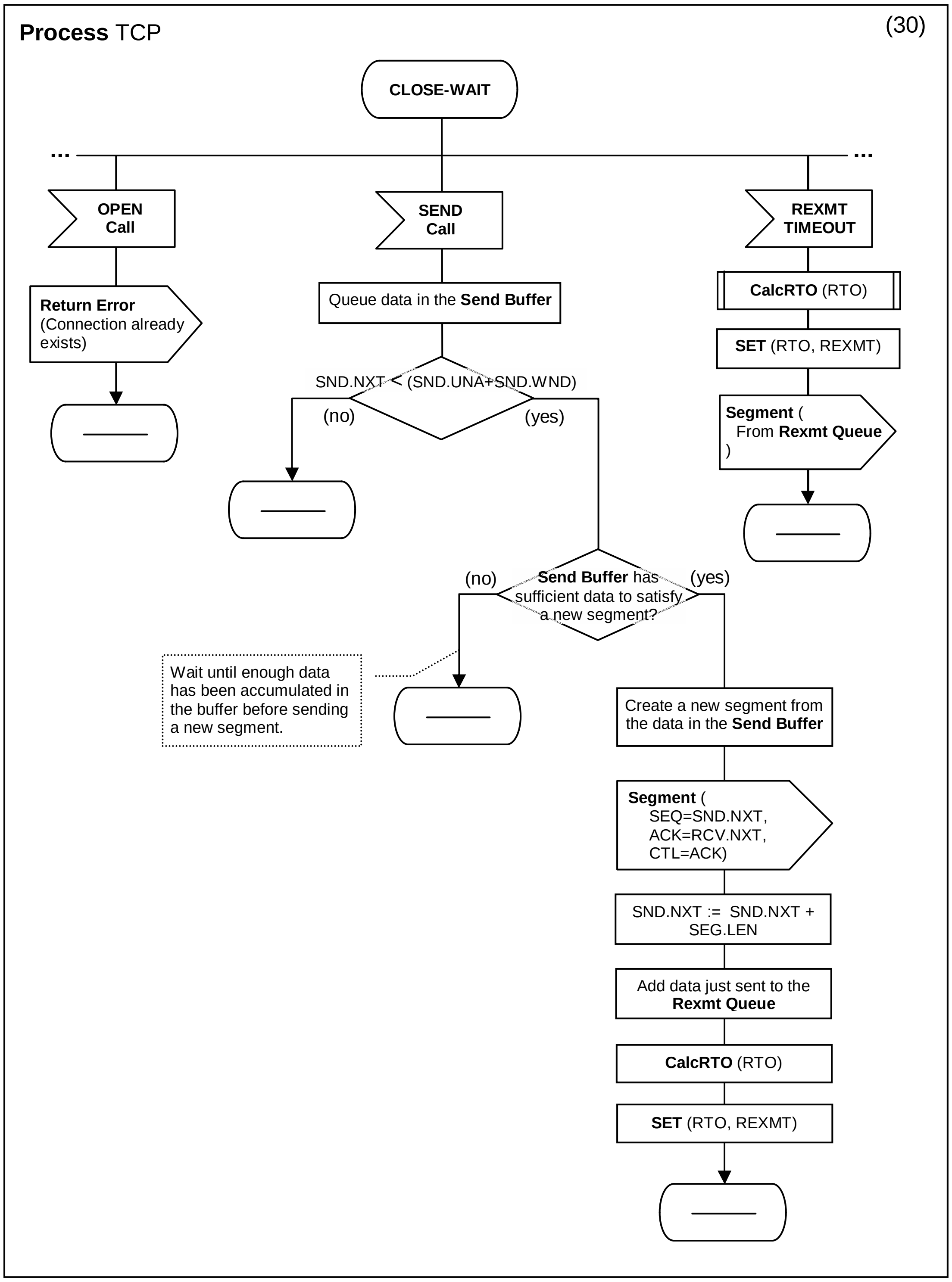

Process TCP
(26)
CLOSING
...
...
OPEN Call
SEND Call
RECEIVE Call
Return Error (Connection already exists)
Return Error (Connection Closing)
Return Error (Connection Closing)
CLOSE Call
ABORT Call
REXMT TIMEOUT
Return Error (Connection Closing)
Return (OK; Closing)
Delete TCB
CLOSED
CalcRTO (RTO)
SET (RTO, REXMT)
Segment (From Rexmt Queue)

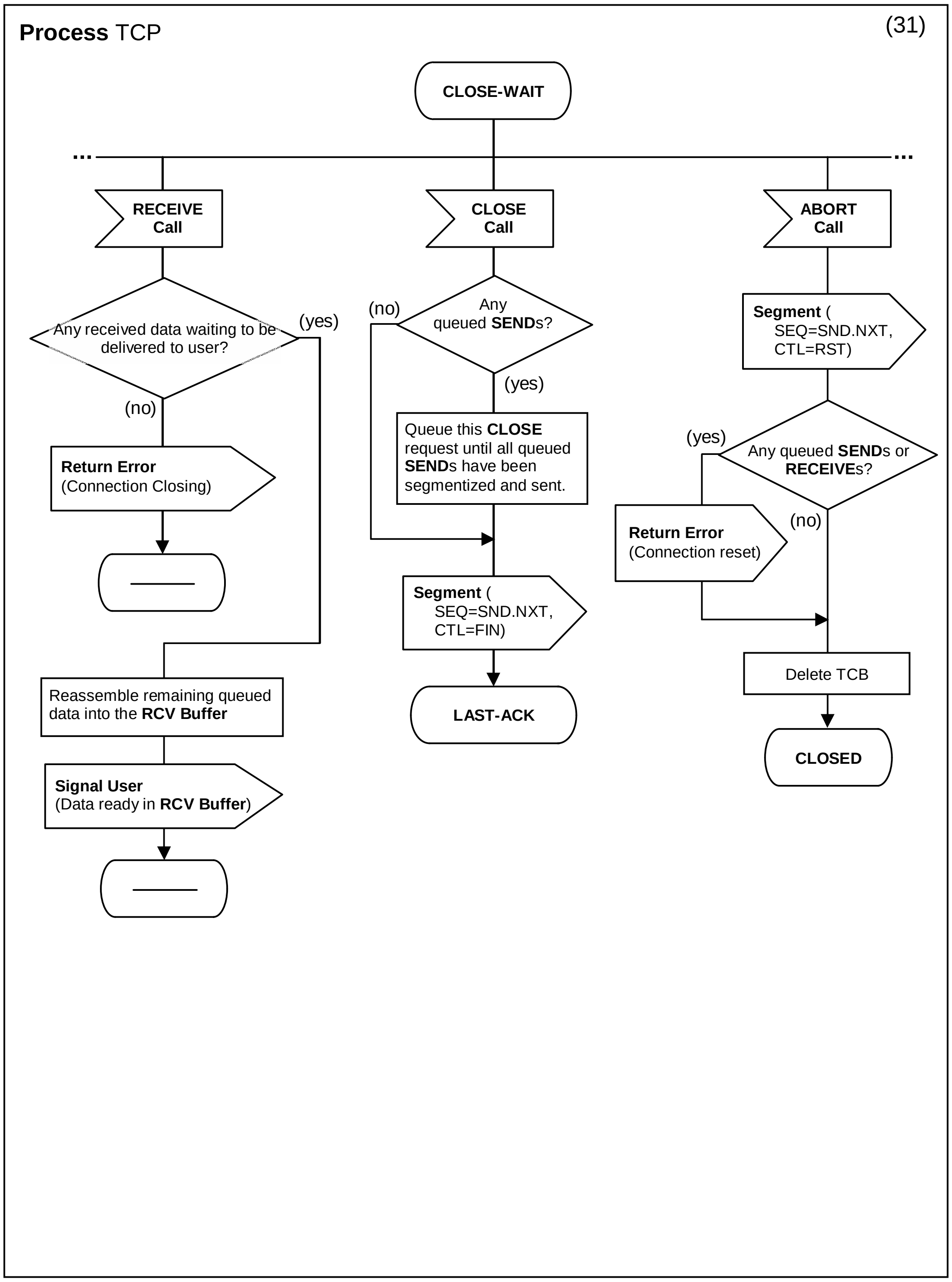

Process TCP
(27)
CLOSING
...
...
SEGMENT ARRIVE
[Acceptable]
Check Segment SEQ (SEG.SEQ, SEG.LEN, RCV.NXT, RCV.WND)
[Not-acceptable]
SEG.RST is on
(yes)
(no)
Delete TCB
CLOSED
SEG.RST is on
(yes)
(no)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
SEG.SYN is on
(no)
(yes)
Segment (SEQ=SND.NXT, CTL=RST)
SEG.ACK is on
(no)
(yes)
SND.UNA < SEG.ACK ≤ SND.NXT
(no)
(yes)
Any queued SENDs or RECEIVEs?
(no)
(yes)
Return Error (Connection reset)
Our FIN has been ACKed?
(no)
(yes)
Turn off all timers
SET (2MSL, TIMEWAIT)
Delete TCB
CLOSED
TIME-WAIT

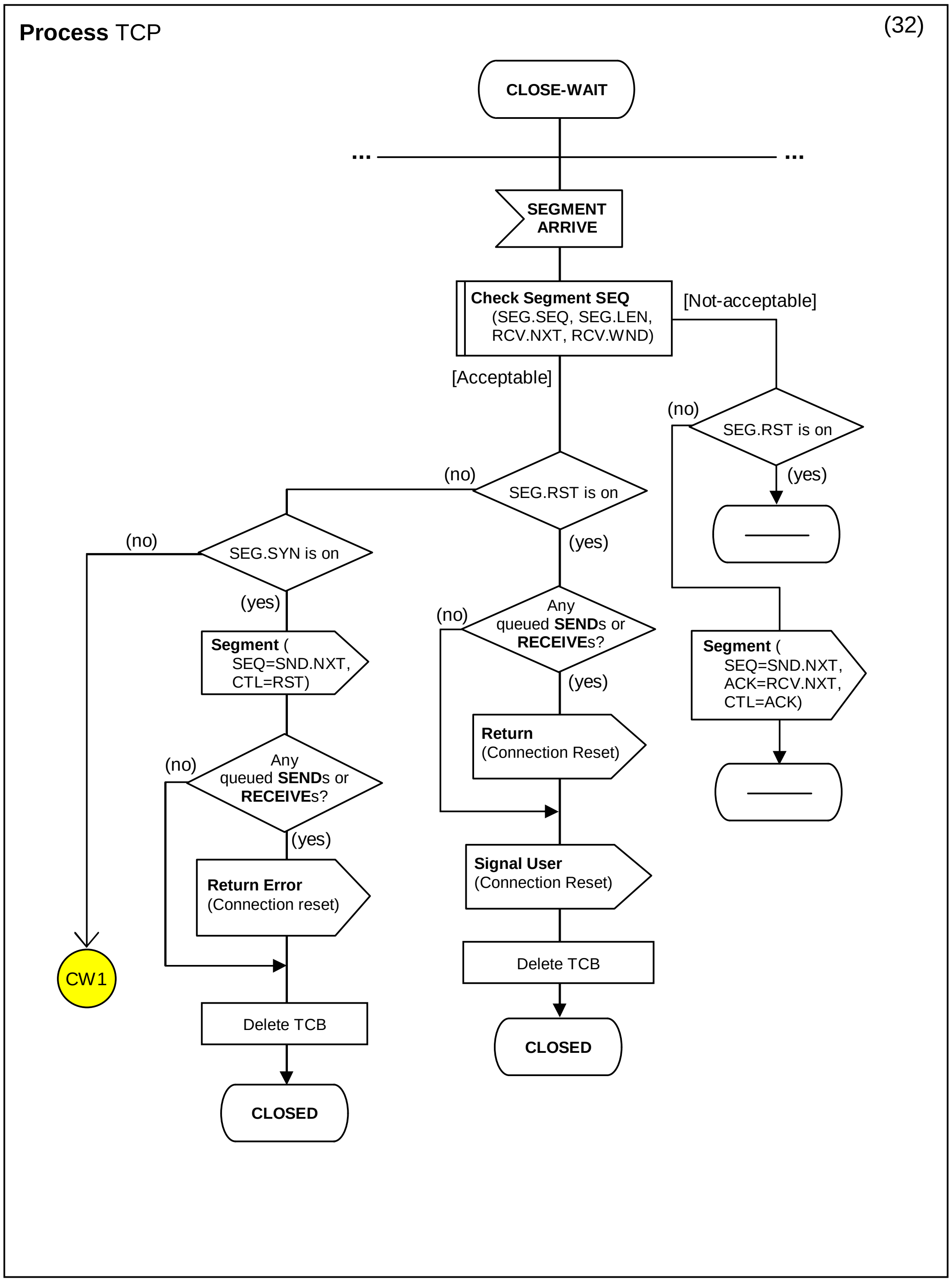

Process TCP
(28)
TIME-WAIT
...
...
OPEN Call
SEND Call
RECEIVE Call
Return Error (Connection already exists)
Return Error (Connection Closing)
Return Error (Connection Closing)
CLOSE Call
ABORT Call
Return Error (Connection Closing)
Return (OK; Closing)
Delete TCB
CLOSED

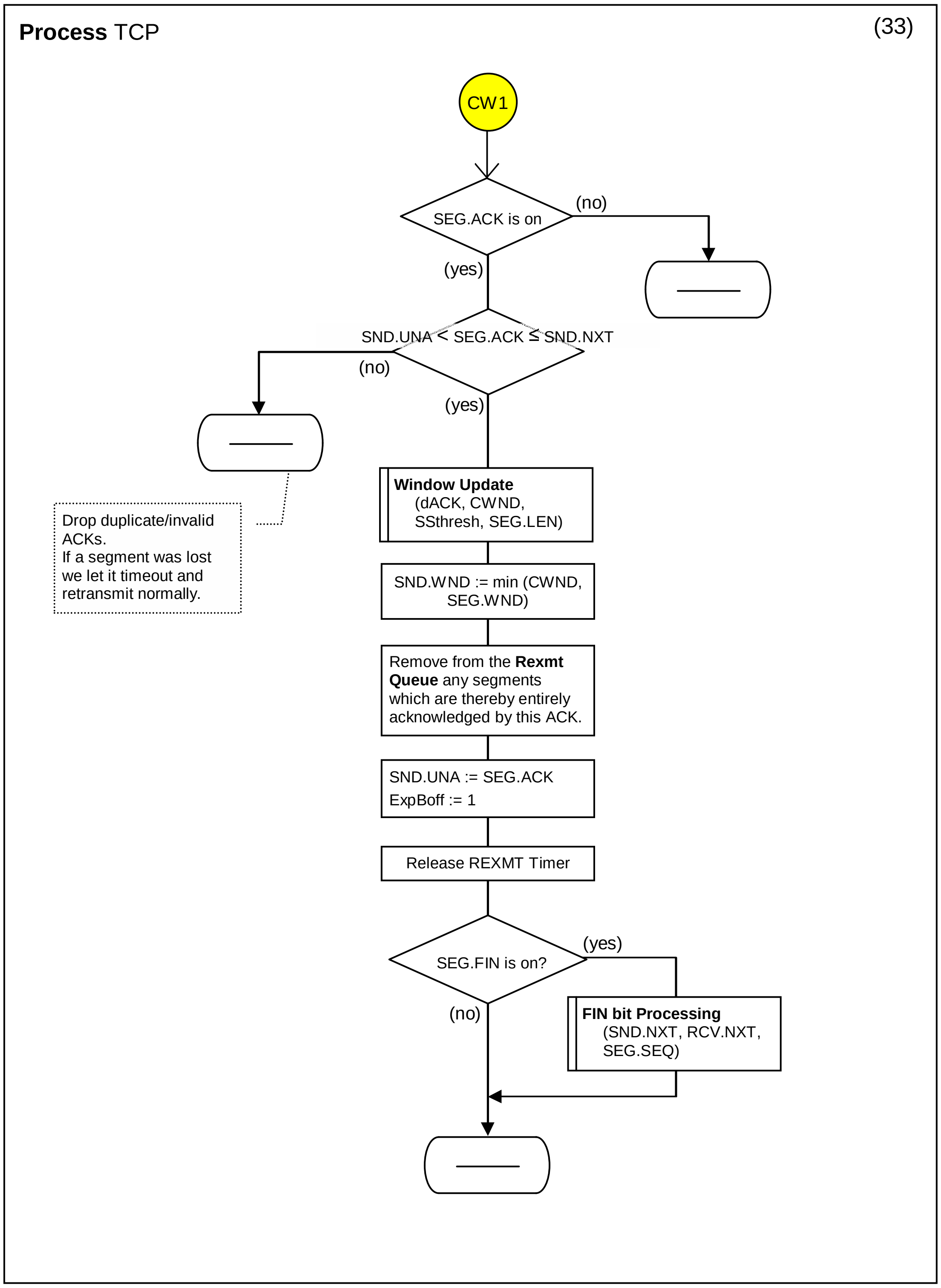


Process TCP
(29)
TIME-WAIT
...
...
SEGMENT ARRIVE
TIMEWAIT TIMEOUT
[Acceptable]
Check Segment SEQ (SEG.SEQ, SEG.LEN, RCV.NXT, RCV.WND)
[Not-acceptable]
Delete TCB
CLOSED
SEG.RST is on
(yes)
(no)
Delete TCB
CLOSED
SEG.RST is on
(no)
(yes)
SEG.SYN is on
(no)
(yes)
Segment (SEQ=SND.NXT, CTL=RST)
SEG.ACK is on
(no)
(yes)
SND.UNA < SEG.ACK ≤ SND.NXT
(no)
(yes)
Any queued SENDs or RECEIVEs?
(no)
(yes)
Return Error (Connection reset)
SEG.FIN is on?
(no)
(yes)
FIN bit Processing (SND.NXT, RCV.NXT, SEG.SEQ)
Delete TCB
CLOSED
Turn off all timers
SET (2MSL, TIMEWAIT)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)

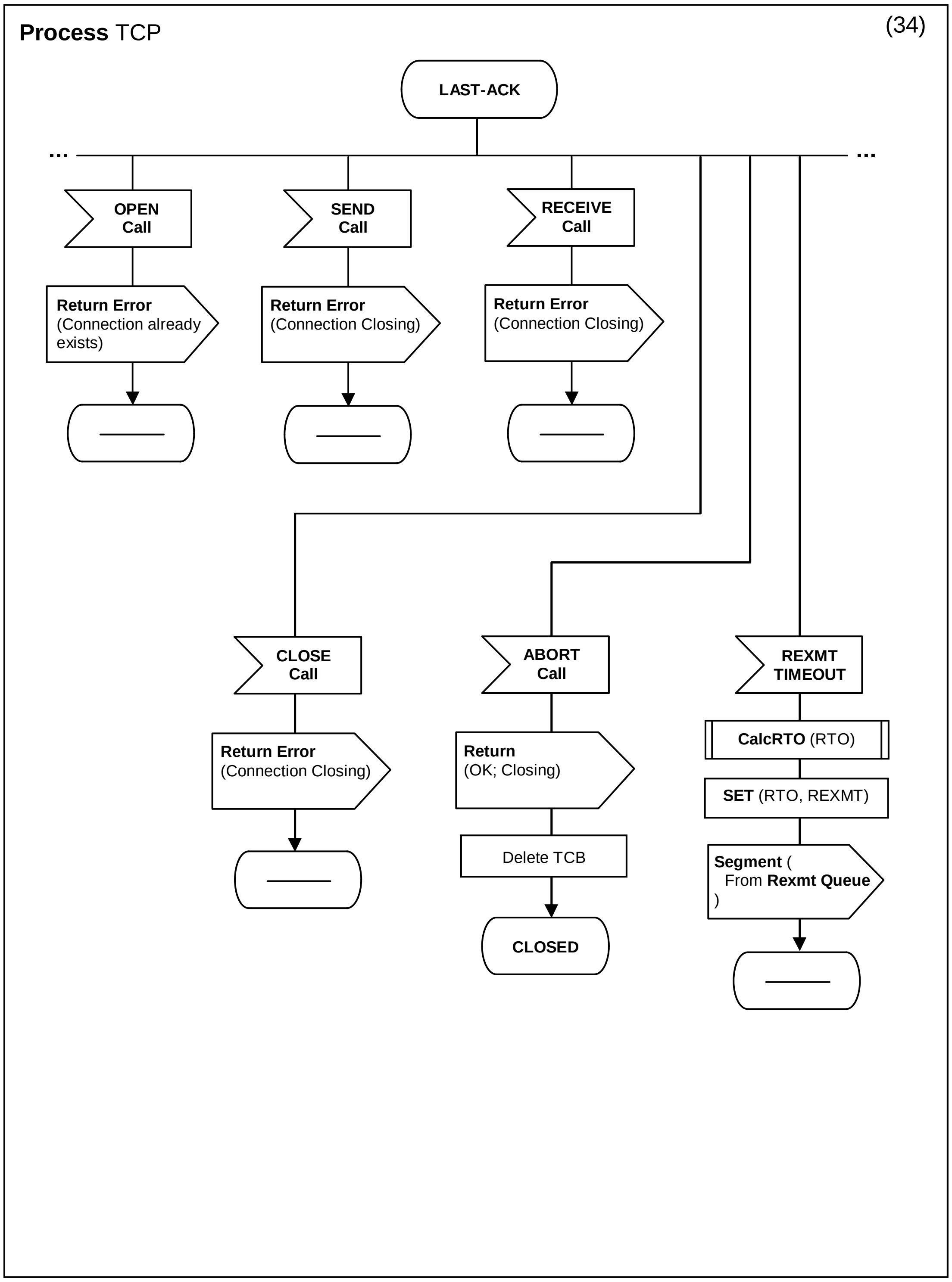

Process TCP
(30)
CLOSE-WAIT
...
...
OPEN Call
SEND Call
REXMT TIMEOUT
Return Error (Connection already exists)
Queue data in the Send Buffer
SND.NXT < (SND.UNA+SND.WND)
(no)
(yes)
CalcRTO (RTO)
SET (RTO, REXMT)
Segment (From Rexmt Queue)
(no)
Send Buffer has sufficient data to satisfy a new segment?
(yes)
Wait until enough data has been accumulated in the buffer before sending a new segment.
Create a new segment from the data in the Send Buffer
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
SND.NXT := SND.NXT + SEG.LEN
Add data just sent to the Rexmt Queue
CalcRTO (RTO)
SET (RTO, REXMT)

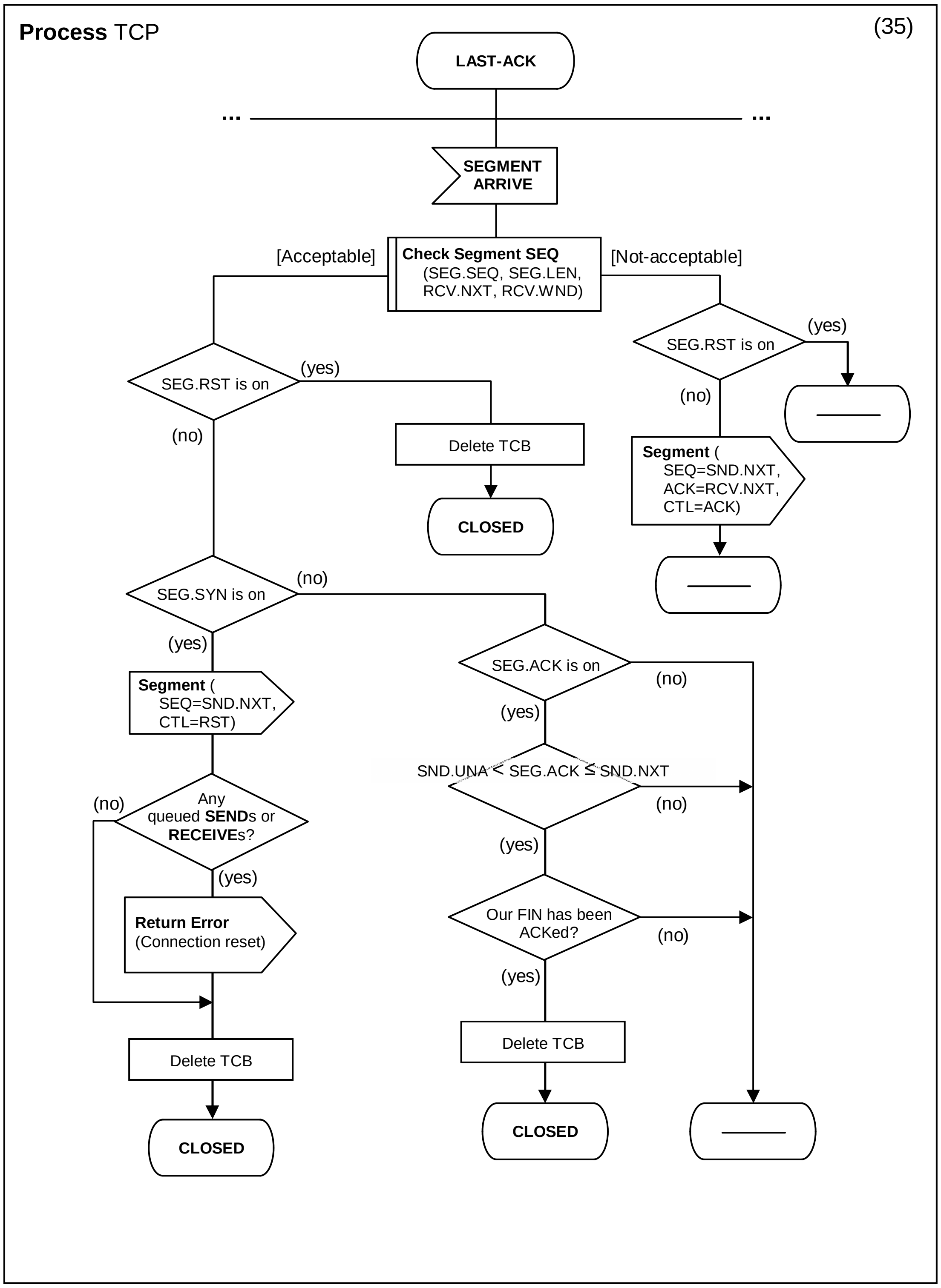

Process TCP
(31)
CLOSE-WAIT
...
...
RECEIVE Call
CLOSE Call
ABORT Call
Any received data waiting to be delivered to user?
(yes)
(no)
Return Error (Connection Closing)
Reassemble remaining queued data into the RCV Buffer
Signal User (Data ready in RCV Buffer)
Any queued SENDs?
(no)
(yes)
Queue this CLOSE request until all queued SENDs have been segmentized and sent.
Segment (SEQ=SND.NXT, CTL=FIN)
LAST-ACK
Segment (SEQ=SND.NXT, CTL=RST)
Any queued SENDs or RECEIVEs?
(yes)
(no)
Return Error (Connection reset)
Delete TCB
CLOSED


Process TCP
(32)
CLOSE-WAIT
...
...
SEGMENT ARRIVE
Check Segment SEQ (SEG.SEQ, SEG.LEN, RCV.NXT, RCV.WND)
[Not-acceptable]
[Acceptable]
(no)
SEG.RST is on
(yes)
(no)
SEG.RST is on
(yes)
(no)
SEG.SYN is on
(yes)
Segment (SEQ=SND.NXT, CTL=RST)
(no)
Any queued **SEND**s or **RECEIVE**s?
(yes)
Return Error (Connection reset)
Delete TCB
CLOSED
CW1
(no)
Any queued **SEND**s or **RECEIVE**s?
(yes)
Return (Connection Reset)
Signal User (Connection Reset)
Delete TCB
CLOSED
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)

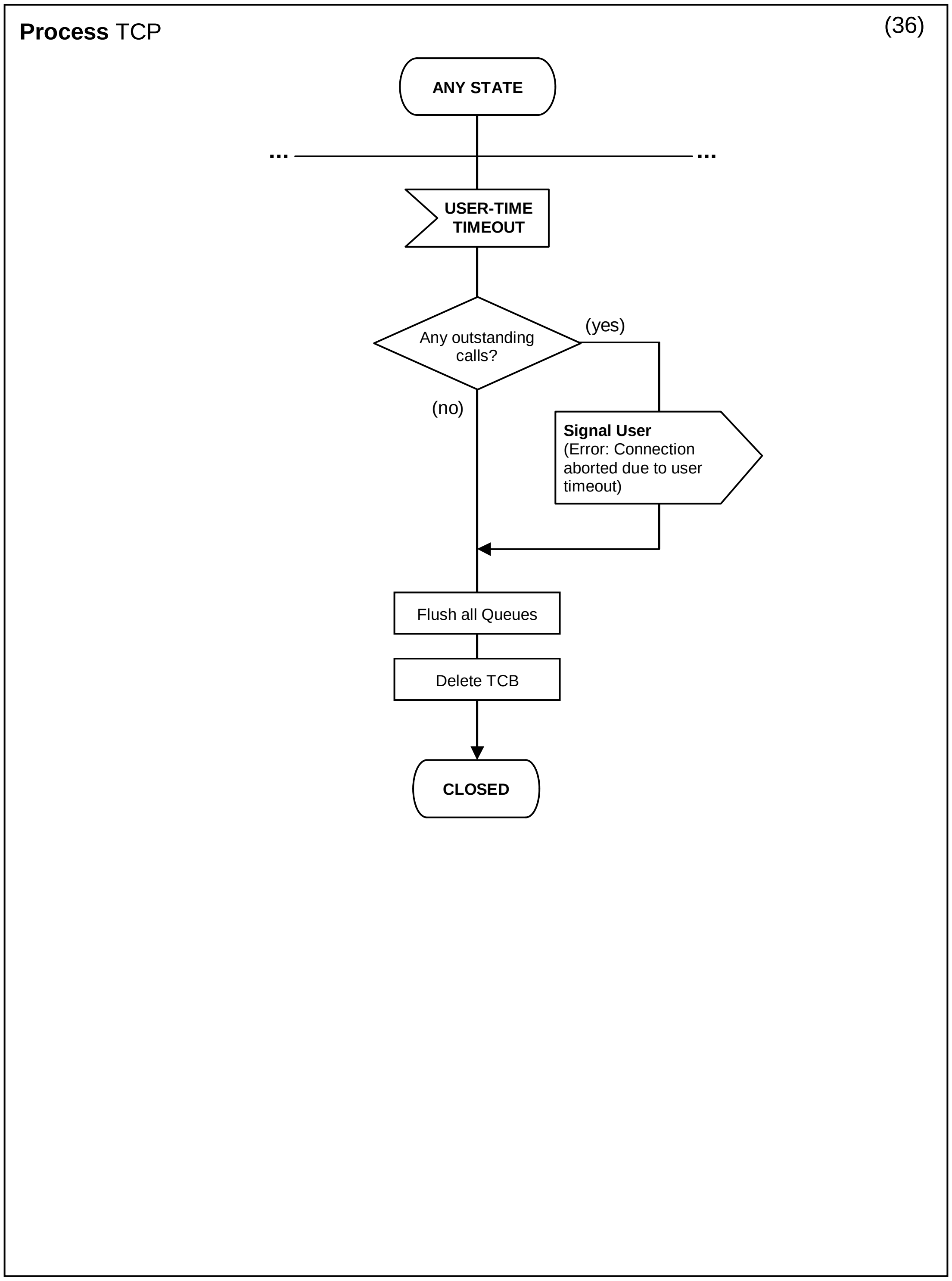


Process TCP
(33)
CW1
SEG.ACK is on
(no)
(yes)
SND.UNA < SEG.ACK ≤ SND.NXT
(no)
(yes)
Drop duplicate/invalid ACKs.
If a segment was lost we let it timeout and retransmit normally.
Window Update (dACK, CWND, SSthresh, SEG.LEN)
SND.WND := min (CWND, SEG.WND)
Remove from the Rexmt Queue any segments which are thereby entirely acknowledged by this ACK.
SND.UNA := SEG.ACK
ExpBoff := 1
Release REXMT Timer
SEG.FIN is on?
(yes)
(no)
FIN bit Processing (SND.NXT, RCV.NXT, SEG.SEQ)

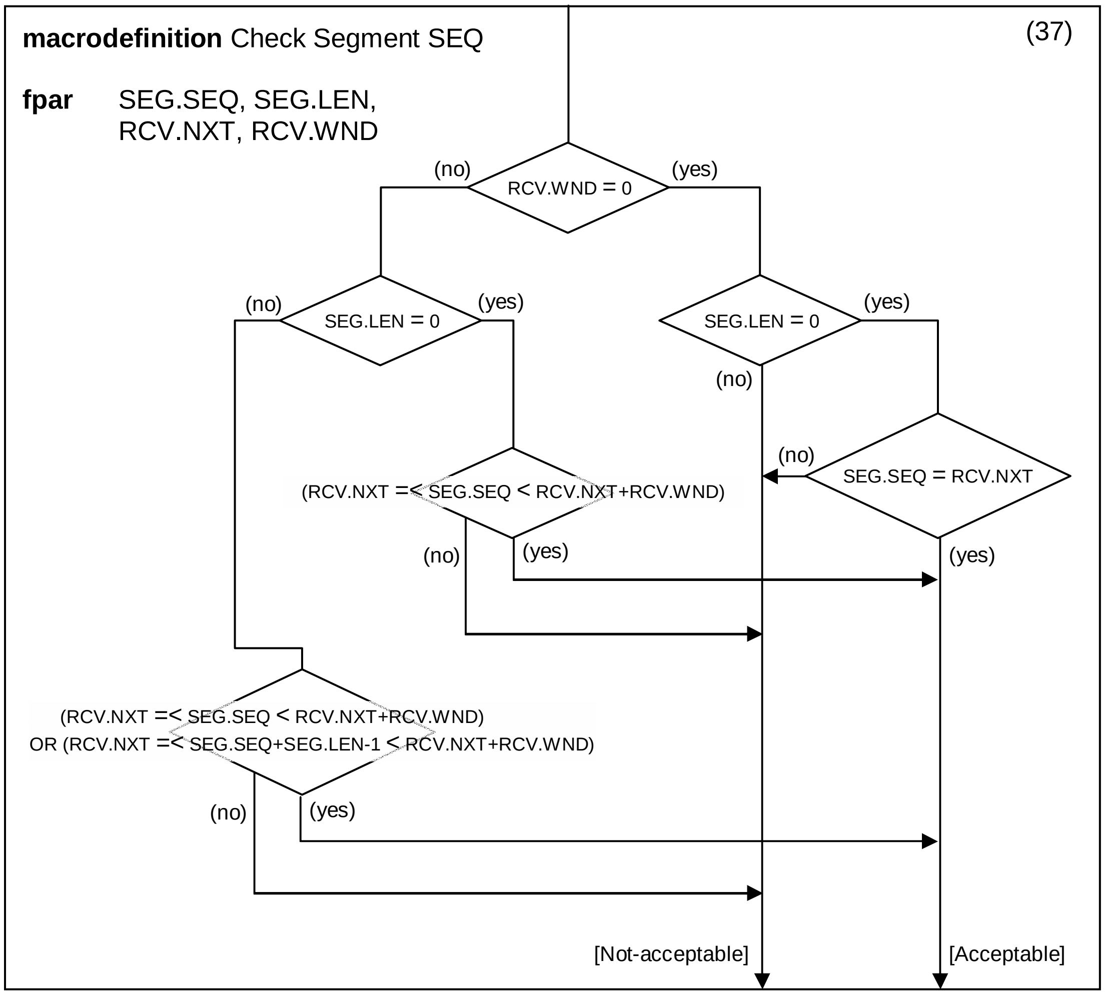

Process TCP
(34)
LAST-ACK
...
...
OPEN Call
SEND Call
RECEIVE Call
Return Error (Connection already exists)
Return Error (Connection Closing)
Return Error (Connection Closing)
CLOSE Call
ABORT Call
REXMT TIMEOUT
Return Error (Connection Closing)
Return (OK; Closing)
Delete TCB
CLOSED
CalcRTO (RTO)
SET (RTO, REXMT)
Segment (From Rexmt Queue)

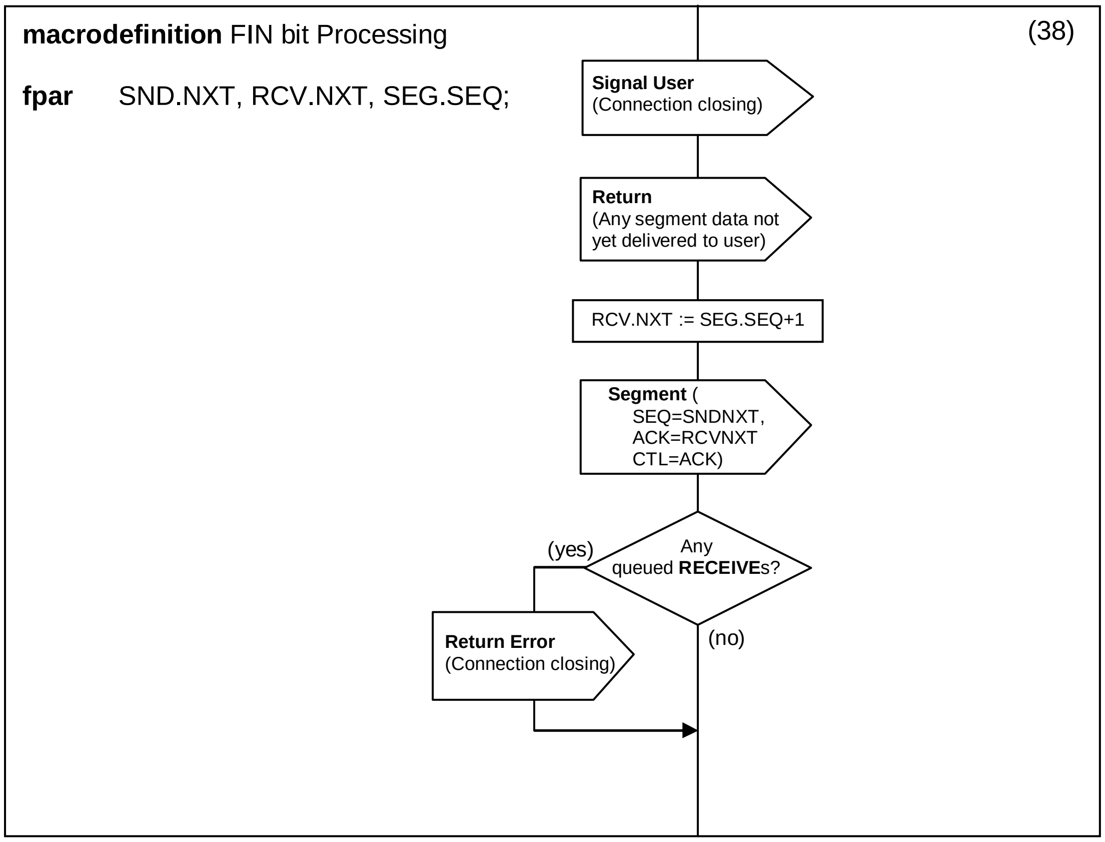


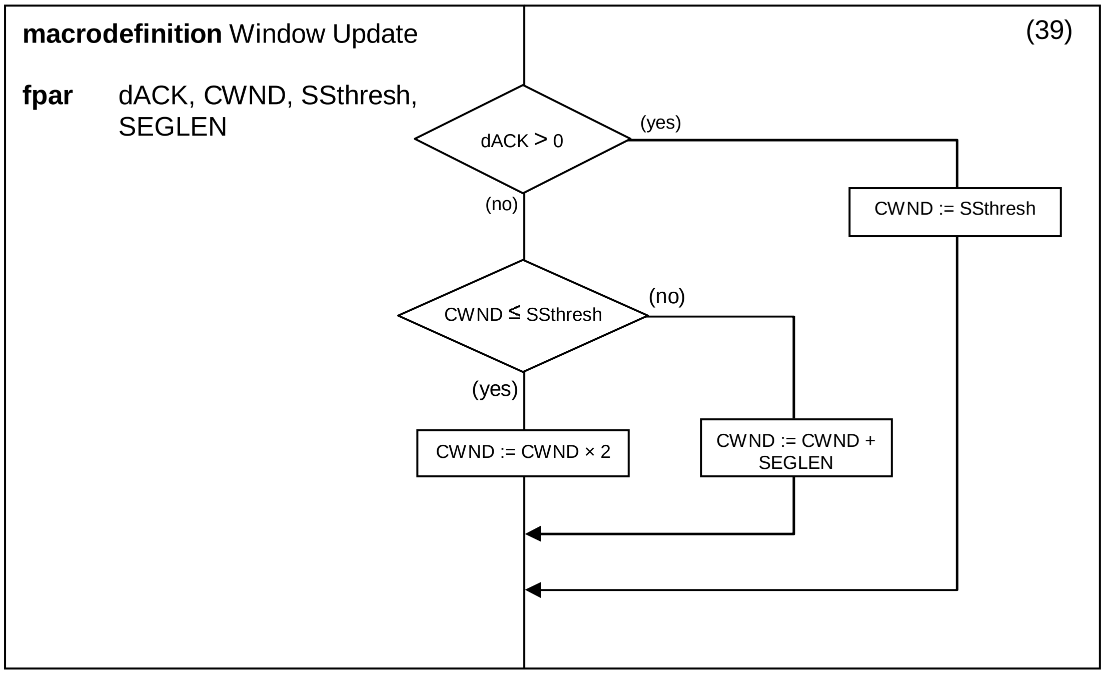
Process TCP
(35)
LAST-ACK
...
...
SEGMENT ARRIVE
Check Segment SEQ (SEG.SEQ, SEG.LEN, RCV.NXT, RCV.WND)
[Acceptable]
[Not-acceptable]
SEG.RST is on
(yes)
(no)
Delete TCB
CLOSED
SEG.RST is on
(yes)
(no)
Segment (SEQ=SND.NXT, ACK=RCV.NXT, CTL=ACK)
SEG.SYN is on
(no)
(yes)
Segment (SEQ=SND.NXT, CTL=RST)
Any queued SENDs or RECEIVEs?
(no)
(yes)
Return Error (Connection reset)
Delete TCB
CLOSED
SEG.ACK is on
(no)
(yes)
SND.UNA < SEG.ACK ≤ SND.NXT
(no)
(yes)
Our FIN has been ACKed?
(no)
(yes)
Delete TCB
CLOSED



Process TCP
(36)
ANY STATE
...
...
USER-TIME TIMEOUT
Any outstanding calls?
(yes)
(no)
Signal User
(Error: Connection aborted due to user timeout)
Flush all Queues
Delete TCB
CLOSED


macrodefinition Check Segment SEQ
(37)
fpar SEG.SEQ, SEG.LEN, RCV.NXT, RCV.WND
RCV.WND = 0
(no)
(yes)
SEG.LEN = 0
(no)
(yes)
SEG.LEN = 0
(yes)
(no)
SEG.SEQ = RCV.NXT
(no)
(yes)
(RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND)
(no)
(yes)
(RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND)
OR (RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND)
(no)
(yes)
[Not-acceptable]
[Acceptable]


macrodefinition FIN bit Processing
(38)
fpar SND.NXT, RCV.NXT, SEG.SEQ;
Signal User (Connection closing)
Return (Any segment data not yet delivered to user)
RCV.NXT := SEG.SEQ+1
Segment (SEQ=SNDNXT, ACK=RCVNXT CTL=ACK)
Any queued **RECEIVE**s?
(yes)
(no)
Return Error (Connection closing)


macrodefinition Window Update
(39)
fpar dACK, CWND, SSthresh, SEGLEN
dACK > 0
(yes)
(no)
CWND := SSthresh
CWND ≤ SSthresh
(no)
(yes)
CWND := CWND × 2
CWND := CWND + SEGLEN

3.4. Variable Classifications

We have also classified variables along every transition from any state to every other state as follows:

1- Conditional variables: those that were used inside decision symbols along the transition,
2- Read-from Variable: those that appeared on the right-hand-side of an assignment along the transition, and
3- Write-to Variables: are those that appeared on the left-hand-side of an assignment along the transition.

The three types are shown in tables 1, 2, and 3 respectively.

Table 1. Conditional variables

	CLOSED	**LISTEN**	**SYN-RCVD**	**SYN-SENT**	**ESTAB**	**FIN-WAIT1**	**FIN-WAIT2**	**CLOSING**	**TIME-WAIT**	**CLOSE-WAIT**	**LAST-ACK**
CLOSED		UCallsQ	UCallsQ RCV.WND RCV.NXT PrevState [SEG]	UCallsQ SND.UNA SND.NXT ISS [SEG]	UCallsQ RCV.NXT RCV.WND [SEG]	UCallsQ RCV.NXT RCV.WND [SEG]	UCallsQ RCV.NXT RCV.WND [SEG]	UCallsQ RCV.NXT RCV.WND [SEG]	UCallsQ RCV.NXT RCV.WND [SEG]	UCallsQ RCV.NXT RCV.WND [SEG]	UCallsQ RCV.NXT RCV.WND SND.UNA SND.NXT [SEG]
LISTEN			RCV.WND RCV.NXT PrevState [SEG]								
SYN-RCVD		[SEG]		[SEG]							
SYN-SENT											
ESTAB			RCV.WND RCV.NXT SND.UNA SND.NXT [SEG]	ISS SND.NXT SND.UNA							
FIN-WAIT1			UCallsQ		UCallsQ						
FIN-WAIT2						ORBuff SND.UNA SND.NXT RCV.NXT RCV.WND [SEG]					
CLOSING						ORBuff SND.UNA SND.NXT RCV.NXT RCV.WND [SEG]					
TIME-WAIT						ORBuff SND.UNA SND.NXT RCV.NXT RCV.WND [SEG]	ORBuff RexQ SND.UNA SND.NXT RCV.NXT RCV.WND [SEG]	UCallsQ SND.NXT SND.UNA RCV.NXT RCV.WND [SEG]			
CLOSE-WAIT			RCV.WND RCV.NXT SND.UNA SND.NXT [SEG]		SND.NXT SND.UNA RCV.NXT RCV.WND ORBuff [SEG]						
LAST-ACK										UCallsQ	

Table 2. Read-from Variables

	CLOSED	LISTEN	SYN-RCVD	SYN-SENT	ESTAB	FIN-WAIT1	FIN-WAIT2	CLOSING	TIME-WAIT	CLOSE-WAIT	LAST-ACK
CLOSED		UCallsQ	UCallsQ RexQ SND.NXT [SEG]	UCallsQ	UCallsQ SND.NXT [SEG]	UCallsQ SND.NXT [SEG]	UCallsQ SND.NXT [SEG]	UCallsQ SND.NXT [SEG]	UCallsQ SND.NXT [SEG]	UCallsQ SND.NXT [SEG]	UCallsQ SND.NXT [SEG]
LISTEN											
SYN-RCVD		RCV.NXT ISS [SEG]		ISS RCV.NXT RexQ [SEG]							
SYN-SENT	ISS [SEG]	ISS [SEG]									
ESTAB			[SEG]	SND.NXT RCV.NXT RexQ [SEG]							
FIN-WAIT1			SND.NXT								
FIN-WAIT2						ORBuff UCallsQ SND.NXT RCV.NXT RCV.WND CWND SSthresh [SEG]					
CLOSING						ORBuff UCallsQ SND.NXT RCV.NXT RCV.WND CWND SSthresh [SEG]					
TIME-WAIT						ORBuff UCallsQ SND.NXT RCV.NXT RCV.WND CWND SSthresh [SEG]	ORBuff UCallsQ SND.NXT RCV.NXT RCV.WND CWND SSthresh [SEG				
CLOSE-WAIT			UCallsQ SND.NXT RCV.NXT [SEG]		UCallsQ SSthresh dACK SND.NXT RCV.NXT RCV.WND [SEG]						
LAST-ACK										UCallsQ SND.NXT [SEG]	

Table 3. Write-to Variables

	CLOSED	LISTEN	SYN-RCVD	SYN-SENT	ESTAB	FIN-WAIT1	FIN-WAIT2	CLOSING	TIME-WAIT	CLOSE-WAIT	LAST-ACK
CLOSED		TCB	TCB	TCB	TCB	TCB	TCB	TCB	TCB	TCB [SEG]	TCB [SEG]
LISTEN	TCB										
SYN-RCVD		RCV.NXT IRS ISS SND.NXT SND.UNA [SEG]		RCV.NXT IRS [SEG]							
SYN-SENT	ISS SND.UNA SND.NXT TCB [SEG]	ISS SND.UNA SND.NXT [SEG] TCB									
ESTAB				RexQ RCV.NXT IRS SND.UNA [SEG]							
FIN-WAIT1											
FIN-WAIT2						RexQ SND.WND SND.UNA CWND					
CLOSING						RexQ SND.WND SND.UNA CWND ORBuff RBuff RCV.NXT [SEG]					
TIME-WAIT						RexQ SND.WND SND.UNA CWND RCV.NXT [SEG]	ORBuff RBuff SND.WND SND.UNA RCV.WND RCV.NXT [SEG]				
CLOSE-WAIT					ORBuff RCV.WND RCV.NXT CWND dACK [SEG]						
LAST-ACK										[SEG]	

4. TCP versions

While in this report we have modeled the congestion control mechanism of TCP Reno, more recent versions like TCP Vegas [Bra95] can also be modeled simply by replacing the bandwidth estimation scheme. In this section we briefly discuss the bandwidth estimation scheme of both TCP Reno and TCP Vegas and then we show a quick recipe to convert our Reno model to a Vegas one. A thorough comparison between TCP Reno and Vegas can be found in [Mo99].

4.1. TCP Reno

TCP Reno induces packet losses to estimate the available bandwidth in the network. While there are no packet losses, TCP Reno continues to increase its window size by one during each round trip time. When it experiences a packet loss, it reduces its window size to one half of the current window size. This is called AIMD (Additive Increase and Multiplicative Decrease). The congestion avoidance mechanism adopted by TCP Reno can cause periodic oscillation in the window size due to the constant update of the window size, which leads to an oscillation in the round trip delay of the packets.
The rate at which each connection updates its window size depends on the round trip delay of the connection. Hence, the connections with shorter delays can update their window sizes faster than other connections with longer delays, and thereby steal an unfair share of the bandwidth. As a result, it has been shown that TCP Reno exhibits an undesirable bias against the connections with longer delays [Flo91].

4.2. TCP Vegas

TCP Vegas adopts a different bandwidth estimation scheme. It uses the difference between the *Expected* and *Actual* flow rates to estimate the available bandwidth in the network. When the network is not congested, the actual flow rate will be close to the expected flow rate. Otherwise, the actual flow rate will be smaller than the expected flow rate. Using this difference in flow rates, TCP Vegas estimates the congestion level in the network and updates the window size accordingly. Note that this difference in the flow rates can be easily translated into the difference between the window size and the number of acknowledged packets during the round trip time, using the equation:

$$Diff = (Expected - Actual) \times BaseRTT$$

Where *Expected* is the expected rate, *Actual* is the actual rate, and *BaseRTT* is the minimum round trip time. The algorithm can be outlined as follows:

1. First, the source computes the expected flow rate $Expected = \frac{CrWND}{BaseRTT}$, where *CrWND* is the current window size and *BaseRTT* is the minimum round trip time.

2. Second, the source estimates the actual flow rate by using the actual round trip time according to $Actual = \frac{CrWND}{AcRTT}$, where *AcRTT* is the actual round trip time of a packet.

3. Using the expected and actual flow rates, the source computes the estimated backlog in the queue using $Diff = (Expected - Actual) \times BaseRTT$.

4. Based on *Diff*, the source updates its window size as follows:

$$CrWND = \begin{cases} CrWND + 1 & \text{if } Diff < \alpha \\ CrWND - 1 & \text{if } Diff > \beta \\ CrWND & \text{otherwise} \end{cases}$$

Here we propose a quick recipe to upgrade the existing EFSM/SDL model from TCP Reno to TCP Vegas:

1. Remove all TCP Reno extensions from the model—i.e., return it back to the basic RFC 793 standard.
2. Introduce the following variables in the EFSM:
   - *Actual*: actual flow rate
   - *Expected*: expected flow rate
   - *BaseRTT*: minimum roundtrip time
   - *AcRTT*: Actual roundtrip time
   - *CrWND*: current window size
   - *Diff*: Estimated queue backlog
3. In any state, when a (SEGMENT ARRIVE) event that acknowledges new data occurs, apply the procedure described above to update the current window size (*CrWND*).

5. Conclusion

In this document we presented an EFSM/SDL description of the original TCP standard in RFC 793 plus the congestion control enhancements proposed in [Jac88, Jac90]. We made several simplifying assumptions to make the model as concise and compact as possible and we also used macros and procedures to model repeated functionalities and to hide irrelevant details. The structured and object-oriented features of the SDL notation allows extending and/or modifying this model to describe advanced features or other TCP versions—we have shown one such case based on TCP Vegas.

6. References

[All99] Allman M., Paxson V., and Stevens W., “TCP Congestion Control,” RFC 2581, April 1999.

[Bra95] Brakmo L.S., Peterson L.L., “TCP Vegas: end to end congestion avoidance on the global Internet,” IEEE Journal on Selected Areas in Communications, 13(8):1465-80, October 1995.

[Byu01] Byun Y., Sanders B., and Keum C-S., "Design Patterns of Communicating Extended Finite State Machines in SDL," 8th Conference on Pattern Languages of Programs (PLoP'01), 2001.

[Ell97] Ellsberger J., Hogrefe D., and Sarma A., "SDL: Formal Object-Oriented Language For Communicating Systems," Prentrice Hall, Harlow, England, 1997.

[Flo91] Floyd S. and Jacobson V., “Connection with multiple Congested Gateways in packet-Switched Networks, Part1: One-way Traffic,” ACM Computer Communication Review, 21(5):30-47, August 1991.

[Jac88] Jacobson V., “Congestion Avoidance and Control,” Computer Communication Review, vol. 18, no. 4, pp. 314-329, Aug. 1988.

[Jac90] Jacobson V., “Modified TCP Congestion Avoidance Algorithm,” end2end-interest mailing list, April 1990.

[Mo99] Mo J., La R., Anantharam V. and Walrand J., "Analysis and Comparison of TCP Reno and Vegas," Proc. INFOCOM'99, Mar 1999.

[Pos81] Postel J., “Transmission Control Protocol,” RFC 793, September 1981.

[SDLfrm] SDL Forum Society. SDL specification (z.100 11/99). http://www.sdl-forum.org.

[Ste94] Stevens W. R., “TCP/IP Illustrated, Volume 1: The Protocols,” Addison-Wesley, 1994.

[Zag05] Raid Zaghal, “Interactive Protocols for extensible networking,” Ph.D. Dissertation, Kent State University, August 2005.