Chapter 20: Unicast Routing

Outline

20.1 INTRODUCTION

20.2 ROUTING ALGORITHMS

20-1 INTRODUCTION

Unicast routing in the Internet, with a large number of routers and a huge number of hosts, can be done only by using <u>hierarchical routing</u>: routing in <u>several steps</u> using <u>different routing algorithms</u>.

In this chapter, we discuss the general <u>concept</u> and <u>algorithms</u> of unicast routing.

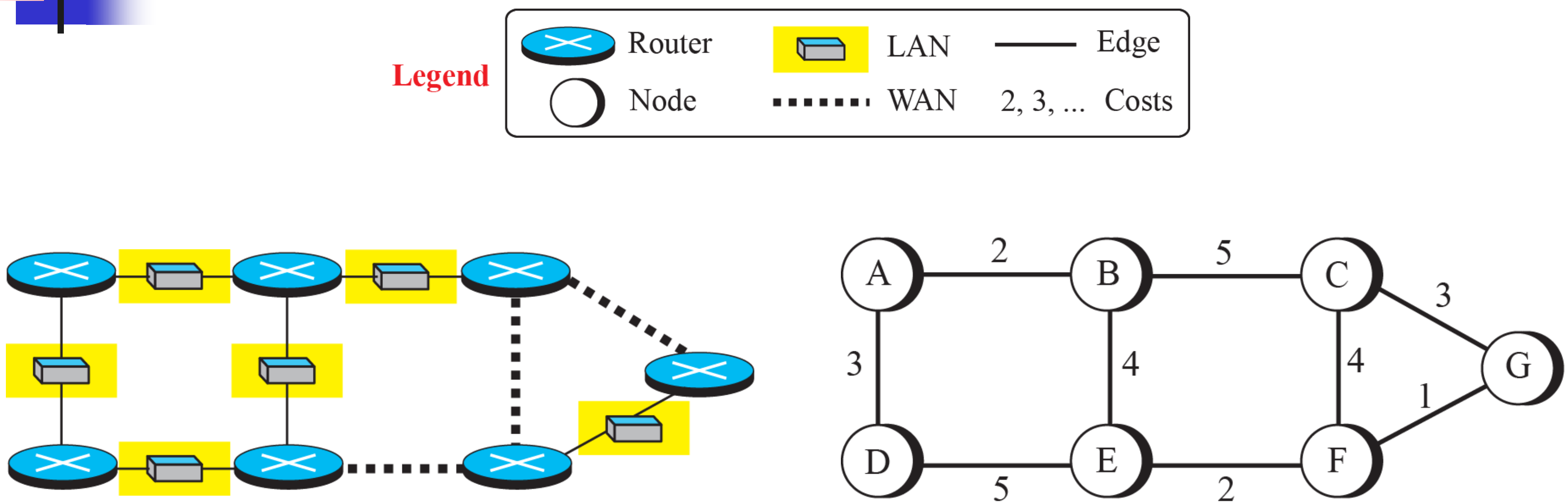
20.1.1 General Idea

In unicast routing, a packet is routed, hop by hop, from its source to its destination with the use of forwarding tables.

To find the best route, an internet can be modeled as a graph. A graph is a set of nodes and edges that connect the nodes. To model an internet as a graph, we can represent each router as a node and each network between a pair of routers as an edge.

In fact, an internet can be modeled as a weighted graph, in which each edge is associated with a cost. If there is no edge between the nodes, the cost is infinity.

20.1.2 Least-Cost Routing



a. An internet

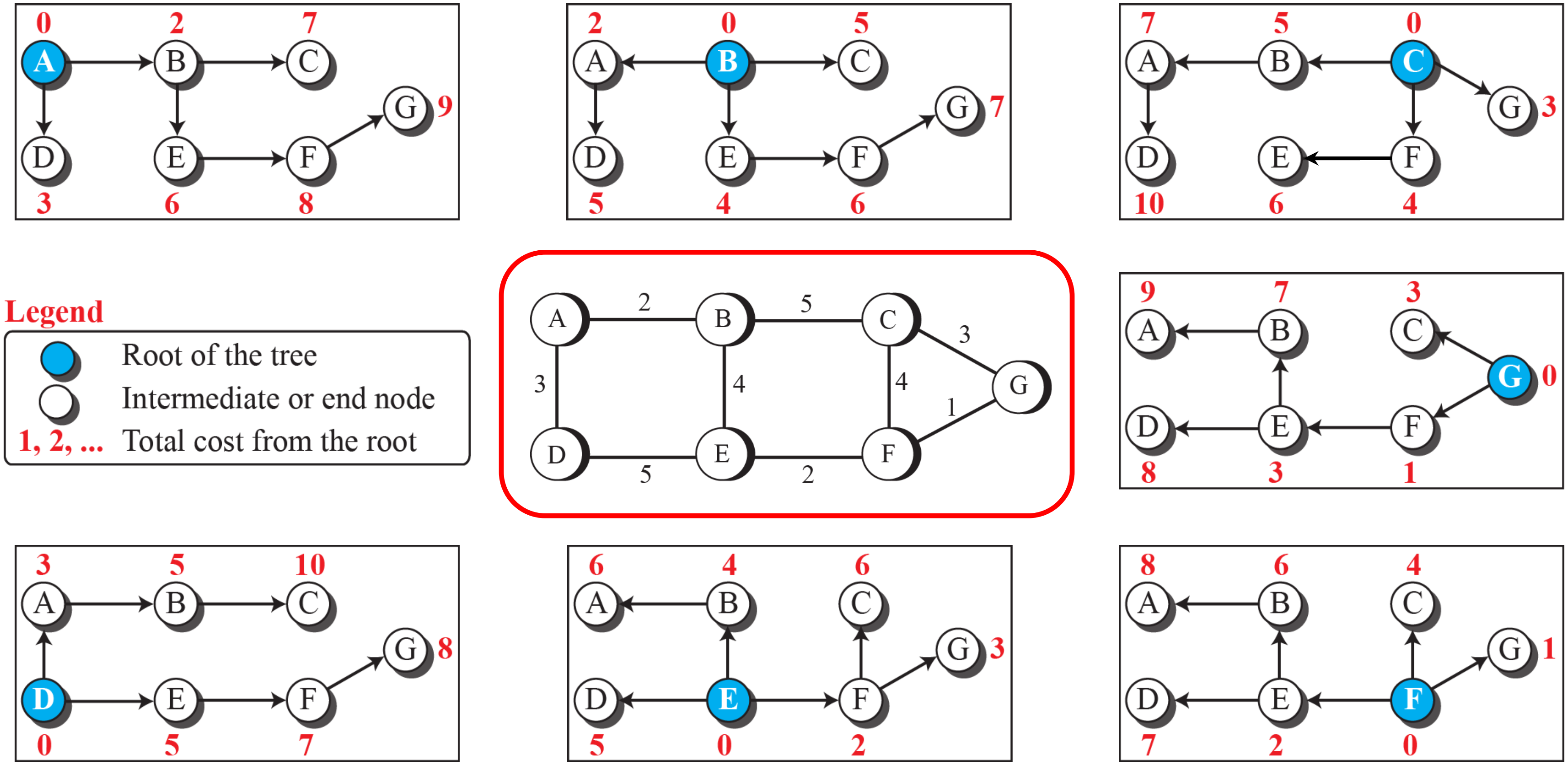
b. The weighted graph

One of the ways to determine the best route from the source router to the destination router is to find the least cost between the two in the weighted graph representation.

In other words, the source router chooses a route to the destination router in such a way that the total cost for the route is the least cost among all possible routes.

Figure 20.2: ***Least-cost trees***

A <u>least-cost tree</u> is a tree with the <u>source router as the root</u> that spans the whole graph (visits all other nodes) and in which the <u>path</u> <u>between</u> the <u>root</u> and <u>any other node</u> is the <u>shortest</u>. Note, if there are N routers in an internet, there are N least-cost trees (one shortest-path tree for each node).



Properties:

(1) The <u>cost</u> of the <u>least-cost routes</u> is the <u>same in both directions</u> (from inverse trees), i.e., [A → G (using least-cost tree A)] = [G → A (using least-cost tree G)] = 9.

(2) The <u>cost</u> of using a <u>combination of routes</u> with <u>different least-cost trees</u> is the <u>same</u> as using a <u>single least-cost tree</u>, i.e., [A → E (using least-cost tree A) + E → G (using least-cost tree E)] = [A → G (using least-cost tree A)] = 9.

20-2 ROUTING ALGORITHMS

Several routing algorithms have been designed in the past. The differences between these methods are in the way they <u>interpret</u> the <u>least cost</u> and the way they <u>create</u> the <u>least-cost tree</u> for each node.

In this section, we discuss <u>distance-vector routing</u> and <u>link-state routing</u>.

20.2.1 Distance-Vector Routing

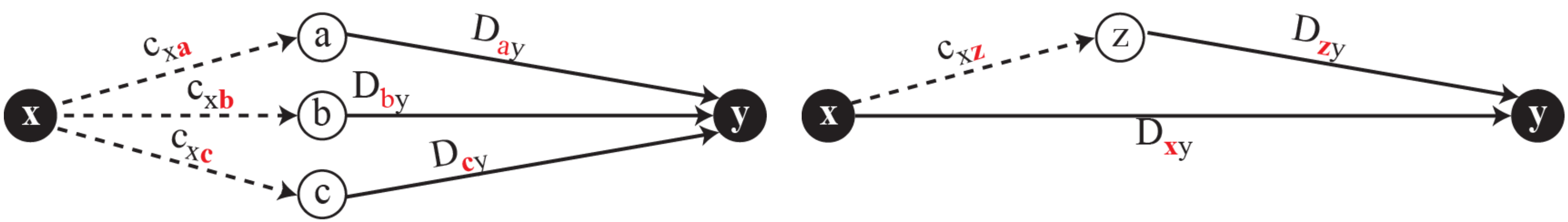
In distance-vector routing, the first thing <u>each node</u> creates is its own <u>least-cost tree</u> with the <u>rudimentary information</u> it has about its <u>immediate neighbors</u>. The <u>incomplete trees</u> are <u>exchanged</u> between <u>immediate neighbors</u> to make the trees <u>more and more complete</u> and to represent the whole internet.

In distance-vector routing, a <u>router</u> continuously <u>tells</u> all of its <u>neighbors</u> what it knows about the <u>whole internet</u> (although the knowledge can be incomplete).

Figure 20.3: Graphical idea behind Bellman-Ford equation

At the heart of distance vector routing is the Bellman-Ford equation. This equation is used to find the least cost between a source node, x, and a destination node, y, through some intermediary nodes (a, b, c, …) where the costs between the source and the intermediary nodes and the least costs between the intermediary nodes and the destination are given.

The following shows the general case in which D_{ij} is the shortest distance and c_{ij} is the cost between nodes i and j.



a. General case with three intermediate nodes

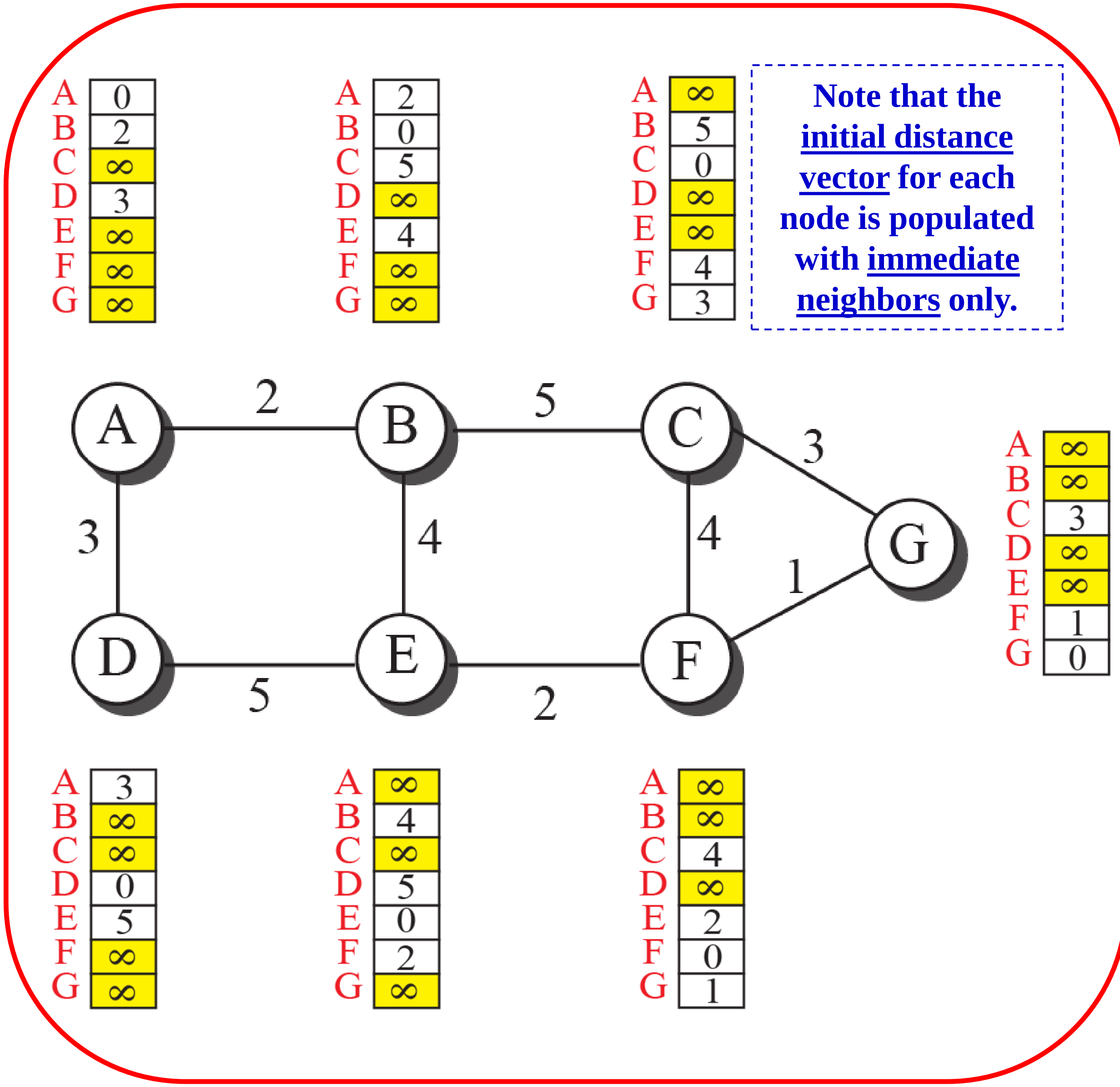
b. Updating a path with a new route

$$D_{xy} = \min\{(c_{xa} + D_{ay}), (c_{xb} + D_{by}), (c_{xc} + D_{cy}), \ldots\}$$

$$D_{xy} = \min\{D_{xy}, (c_{xz} + D_{zy})\}$$

Figure 20.5: *Distance vectors (one-dimensional arrays)*

Initial Distance Vectors



Distance Vector for Node A

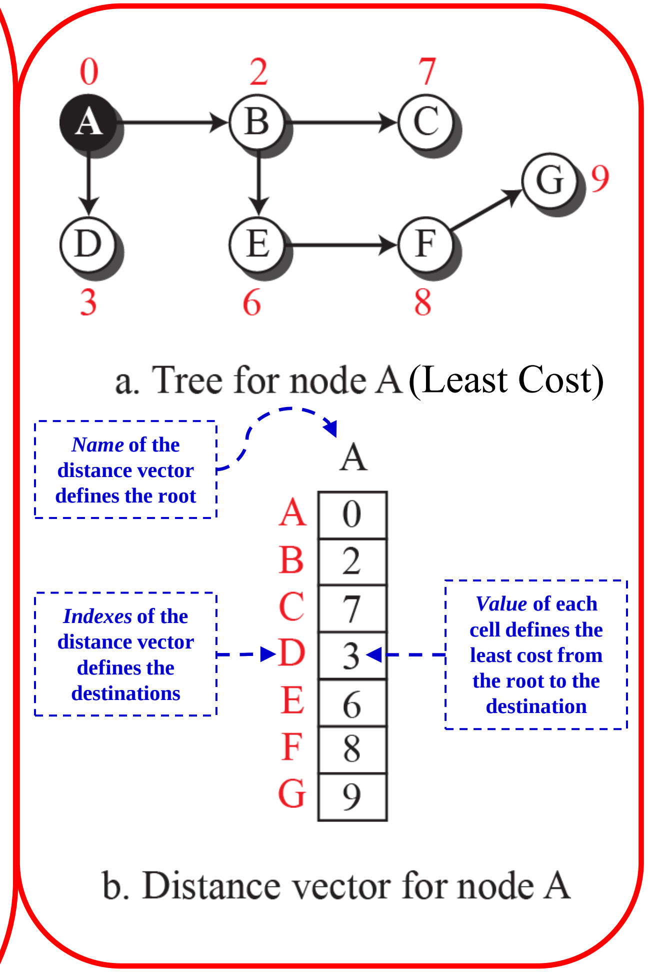
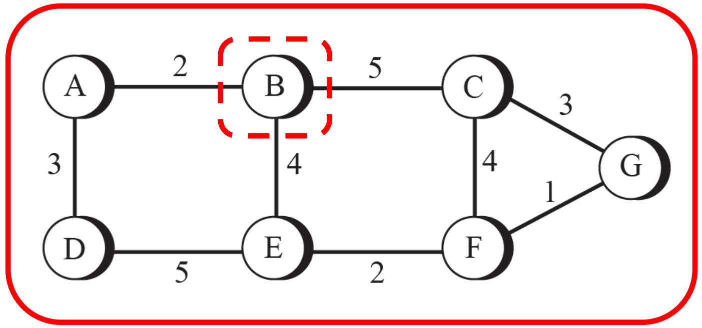


Figure 20.6: Updating distance vectors



	New B	Old B	A
A	2	2	0
B	0	0	2
C	5	5	∞
D	5	∞	3
E	4	4	∞
F	∞	∞	∞
G	∞	∞	∞

B[ ] = min (B[ ] , 2 + A[ ])

a. First event: B receives a copy of A's vector.

	New B	Old B	E
A	2	2	∞
B	0	0	4
C	5	5	∞
D	5	5	5
E	4	4	0
F	6	∞	2
G	∞	∞	∞

B[ ] = min (B[ ] , 4 + E[ ])

b. Second event: B receives a copy of E's vector.

Note:
X[ ]: the whole vector

After each node is updated, it immediately sends its updated vector to all neighbors.

Table 20.1: Distance-Vector Routing Algorithm

```
Distance_Vector_Routing ( )
{
    // Initialize (create initial vectors for the node)
    D[myself] = 0
    for (y = 1 to N)
    {
        if (y is a neighbor)
            D[y] = c[myself][y]
        else
            D[y] = ∞
    }
    send vector {D[1], D[2], …, D[N]} to all neighbors
    // Update (improve the vector with the vector received from a neighbor)
    repeat (forever)
    {
        wait (for a vector Dw from a neighbor w or any change in the link)
        for (y = 1 to N)
        {
            D[y] = min [D[y], (c[myself][w] + Dw[y])]        // Bellman-Ford equation
        }
        if (any change in the vector)
            send vector {D[1], D[2], …, D[N]} to all neighbors
    }
} // End of Distance Vector
```

20.2.2 Link-State Routing

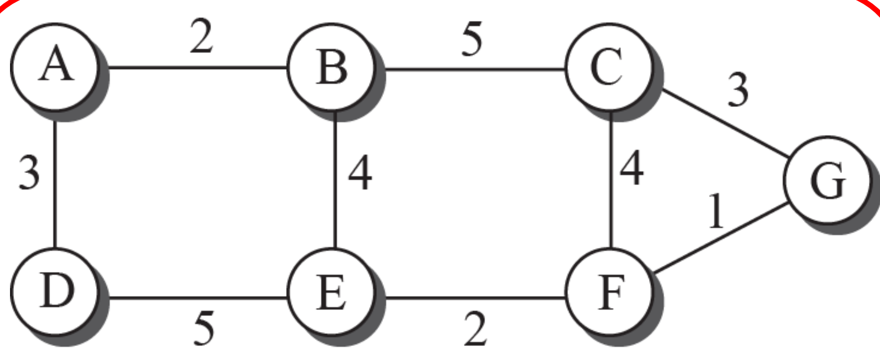
In link-state routing, the term <u>link-state</u> is used to define the <u>characteristic of a link</u> (an edge) that represents a network in the internet.

In this algorithm, the cost associated with an edge defines the state of the link. <u>Links</u> with <u>lower costs</u> are <u>preferred</u> to links with higher costs; if the <u>cost</u> of a link is <u>infinity</u>, it means that the link <u>does not exist</u> or has been <u>broken</u>.

In link-state routing, each <u>router</u> <u>tells</u> the <u>whole internet</u> what it knows about its <u>neighbors</u>.

Figure 20.8: ***Example of a link-state database***

To create a least-cost tree, each node needs to have a complete map of the network. The collection of states for all links is called the link-state database (LSDB), represented as a two-dimensional array (matrix). There is only one LSDB for the whole internet; each node needs to have a duplicate of it to be able to create the least-cost tree.



a. The weighted graph

	A	B	C	D	E	F	G
A	0	2	∞	3	∞	∞	∞
B	2	0	5	∞	4	∞	∞
C	∞	5	0	∞	∞	4	3
D	3	∞	∞	0	5	∞	∞
E	∞	4	∞	5	0	2	∞
F	∞	∞	4	∞	2	0	1
G	∞	∞	3	∞	∞	1	0

b. Link state database

Figure 20.9: ***LSPs created and sent out by each node to build LSDB***

The LSDB is created by a process called flooding. Each node requests and collects two pieces of information in a link-state packet (LSP) from its immediate neighboring nodes: the identity of the node and the cost of the link. When a node receives a LSP on one of its interfaces, it compares it with the copy it may already have. If it's newer, it keeps the newer copy and then sends it out to each of its interfaces except the one from which the LSP arrived. After receiving all new LSPs, each node creates the comprehensive LSDB.

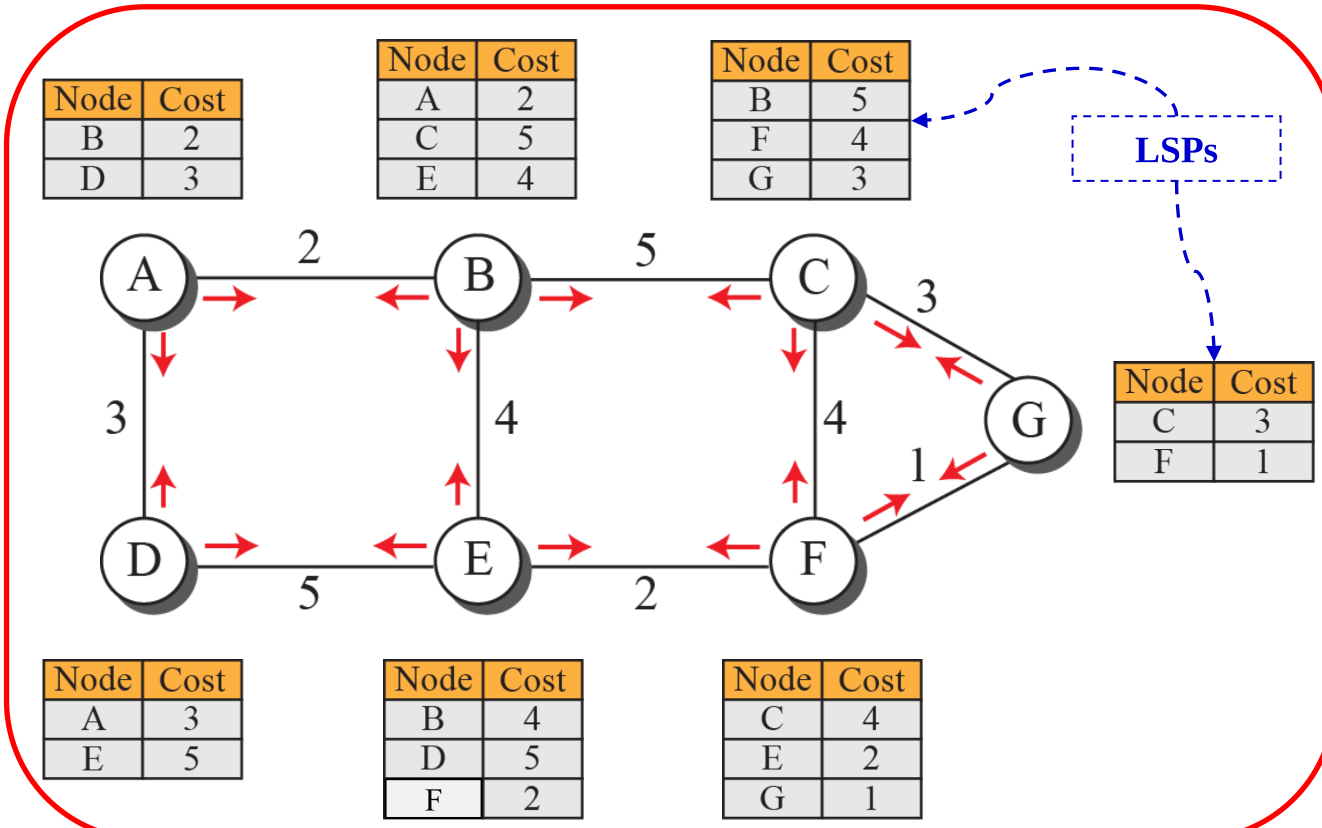


Figure 20.10: Least-cost tree

To create the <u>least-cost tree</u> for itself, each node needs to run the iterative <u>Dijkstra's algorithm</u>:

(1) The node (e.g., Node A below) chooses itself as the root of the tree, creating a tree with a single node, and sets the total cost of each node based on the information in the LSDB.

(2) The node selects one node, among all nodes not in the tree, which is closest to the root, and adds this to the tree. After this node is added to the tree, the cost of all other nodes not in the tree needs to be updated because the paths may have been changed.

(3) The node repeats step 2 until all nodes are added to the tree.

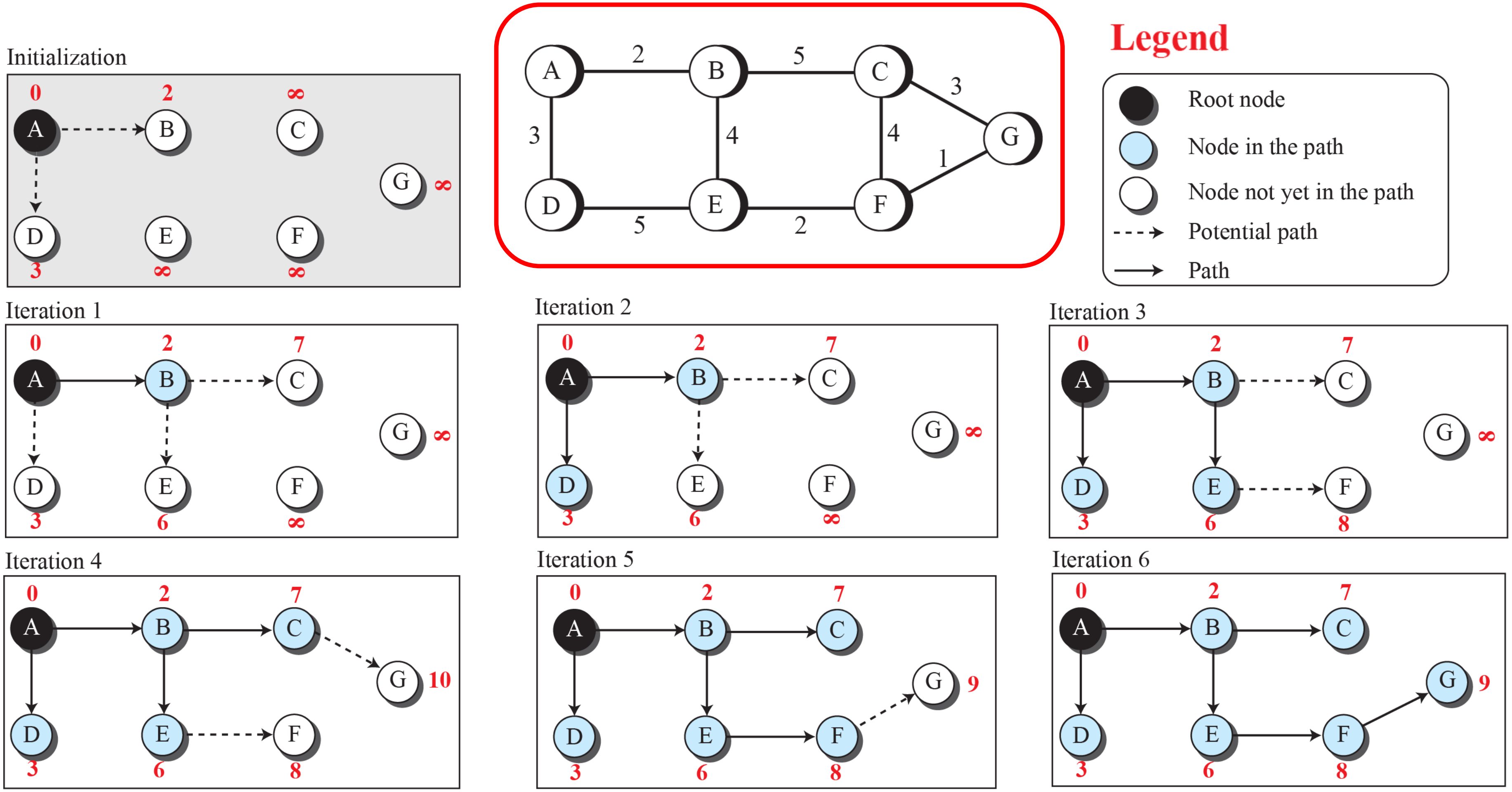


Table 20.2: Dijkstra's Algorithm

```
Dijkstra's Algorithm ( )
{
      // Initialization
      Tree = {root}                        // Tree is made only of the root
      for (y = 1 to N)                     // N is the number of nodes
      {
            if (y is the root)
                  D[y] = 0                 // D[y] is shortest distance from root to node y
            else if (y is a neighbor)
                  D[y] = c[root][y]        // c[x][y] is cost between nodes x and y in LSDB
            else
                  D[y] = ∞
      }
      // Calculation
      repeat
      {
            find a node w, with D[w] minimum among all nodes not in the Tree
            Tree = Tree ∪ {w}              // Add w to tree
            // Update distances for all neighbor of w
            for (every node x, which is neighbor of w and not in the Tree)
            {
                  D[x] = min{D[x], (D[w] + c[w][x])}
            }
      } until (all nodes included in the Tree)
} // End of Dijkstra
```

Chapter 24: Transport-Layer Protocols

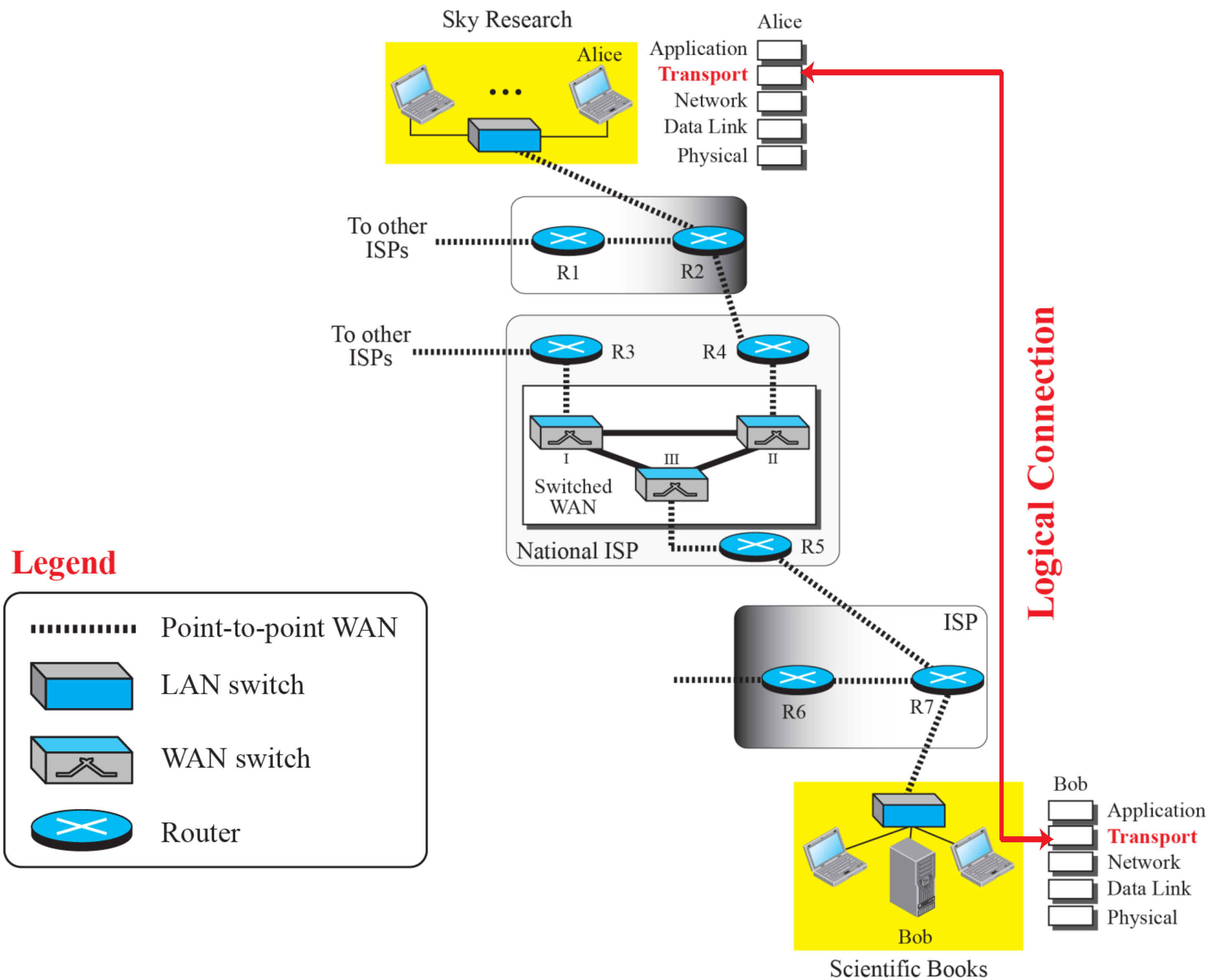
Outline

24-1 INTRODUCTION

The transport layer is located between the application layer and the network layer. It provides a <u>process-to-process communication</u> between two application layers, one at the local host and the other at the remote host.

Communication is provided using a <u>logical connection</u>.

Figure 23.1: Logical connection at the transport layer



23.1.1 Transport-Layer Services

The transport layer is located between the network layer and the application layer. It is responsible for providing services to the application layer and receives services from the network layer.

***Process-to-Process Communication**: as opposed to host-to-host communication in the network-layer, a process is an application-layer entity (running program).*

***Encapsulation and Decapsulation**: process sends message to the transport layer and the sender adds the transport-layer header (encapsulation). Packets at the transport layer are called user datagrams / segments. Decapsulation happens at the receiver.*

***Flow Control**: balance between production and consumption rates.*

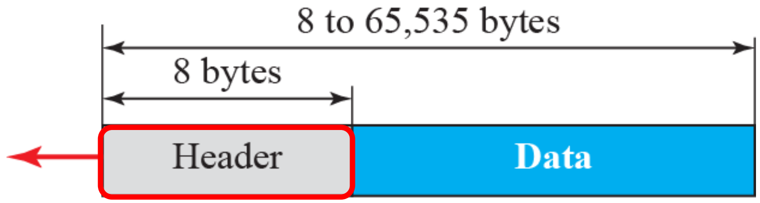
***Error Control**: the transport layer is responsible for (a) detecting and discarding corrupted packets, (b) keeping track of lost and discarded packets and resending them, (c) recognizing duplicate packets and discarding them and (d) buffering out-of-order packets until missing packets arrive.*

***Congestion Control**: implements mechanisms and techniques to keep the load below the capacity.*

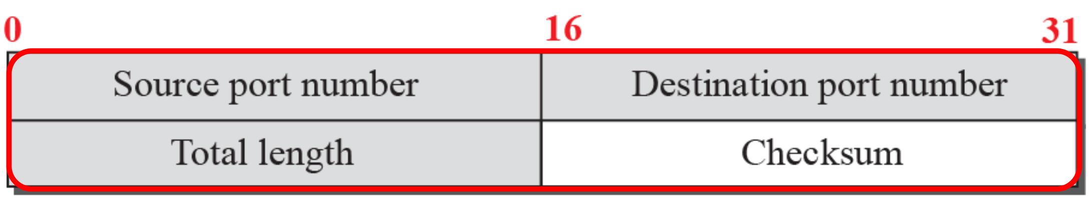
24-2 UDP

The User Datagram Protocol (UDP) is a connectionless, unreliable transport protocol. However, UDP is a very simple and efficient protocol using a minimum of overhead.

UDP packets, called user datagrams, have a fixed-size header of 8 bytes made of four fields, each of 2 bytes. The first two fields define the source and destination port numbers, the third field defines the total length of the user datagram, header plus data. The last field can carry the optional checksum.



a. UDP user datagram

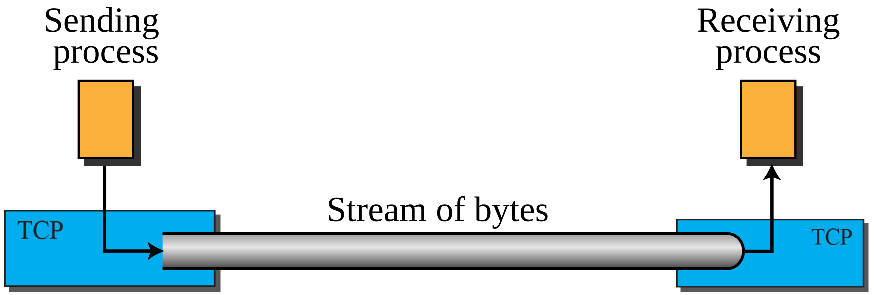


b. Header format

24-3 TCP

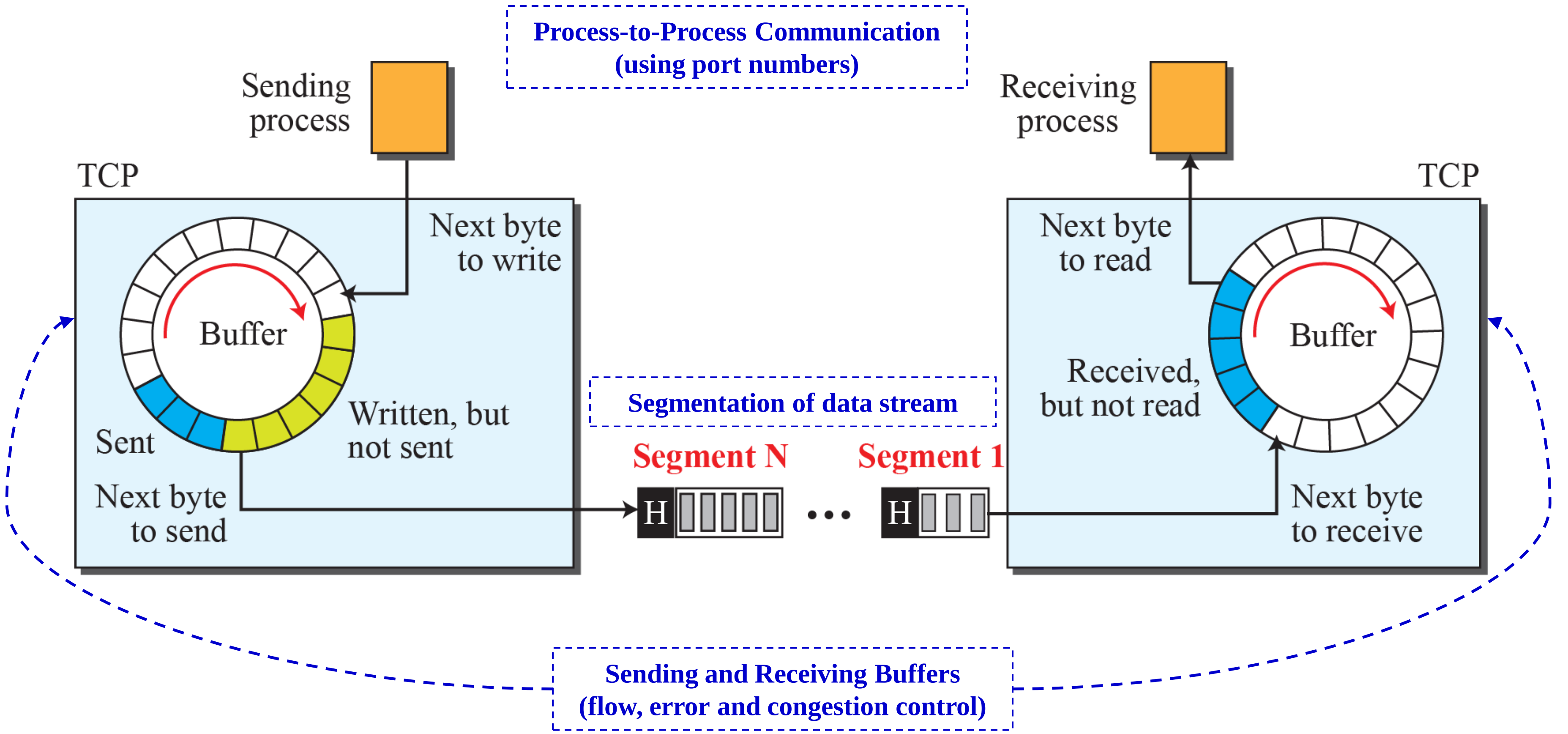
Transmission Control Protocol (TCP) is a <u>connection-oriented</u>, <u>reliable</u> protocol.

TCP, unlike UDP, is a <u>stream-oriented</u> protocol. It allows the sending process to deliver <u>data</u> as a <u>stream of bytes</u> and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary tube that carries their bytes across the Internet.



24.3.1 TCP Services

Let's describe the services offered by TCP protocol (connection-oriented, reliable, full-duplex) to the processes at the application layer.



24.3.2 TCP Features

To provide the services, TCP keeps track of the segments being transmitted or received (for flow and error control) using two fields: sequence number and the acknowledgement number. The bytes of data being transmitted in each connection are numbered by TCP. Numbering is independent in each direction (for full-duplex communication).

***Sequence number**: TCP assigns a sequence number to each segment that is being sent. The sequence number of the first segment is the initial sequence number (ISN) and is an arbitrarily generated random number [0, 2^{32}-1]. The sequence of any other segment is the sequence number of the previous segment plus the number of bytes carried by the previous segment, i.e., first byte of data in the segment.*

***Acknowledgement number**: the value of the acknowledgement number in a segment defines the number of the next byte a party expects to receive. The acknowledgement number is cumulative.*

Example

Suppose a TCP connection is transferring a file of 5,000 bytes. The first byte is numbered 10,001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1,000 bytes?

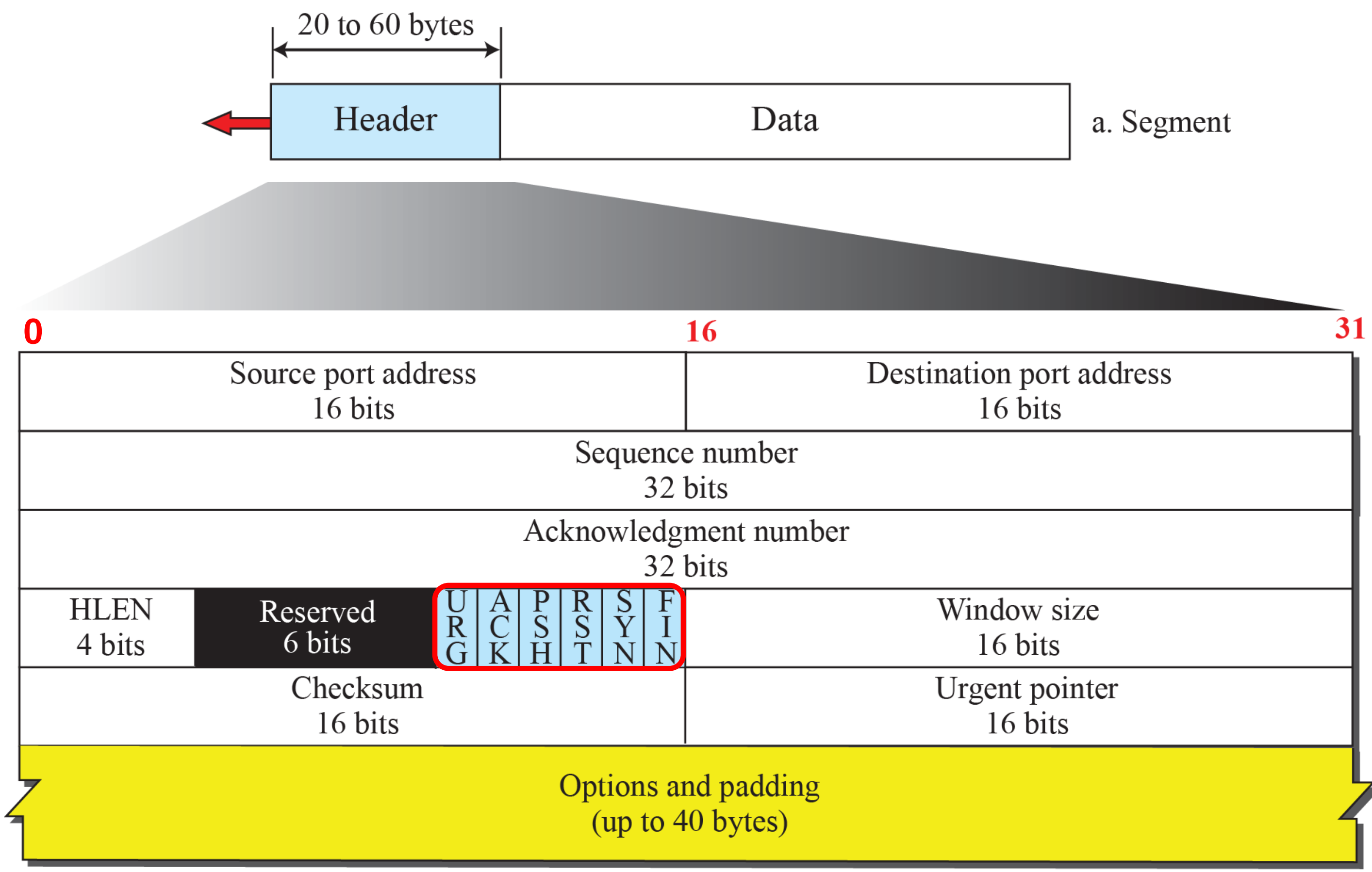
Solution

The following shows the sequence number for each segment. The value in the sequence number field of a segment defines the number assigned to the first data byte contained in that segment.

Segment 1	→	Sequence Number:	10,001	**Range:**	10,001	to	11,000
Segment 2	→	Sequence Number:	11,001	**Range:**	11,001	to	12,000
Segment 3	→	Sequence Number:	12,001	**Range:**	12,001	to	13,000
Segment 4	→	Sequence Number:	13,001	**Range:**	13,001	to	14,000
Segment 5	→	Sequence Number:	14,001	**Range:**	14,001	to	15,000

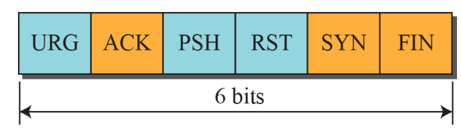
24.3.3 Segment

A packet in TCP is called a segment.



b. Header

Note that HLEN is in units of 4-bytes.



URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection

24.3.4 A TCP Connection

TCP is <u>connection-oriented</u>. All of the segments belonging to a message are then sent over this <u>logical path</u>. Using a single logical pathway for the entire message <u>facilitates</u> the <u>acknowledgment</u> process as well as <u>retransmission</u> of damaged or lost frames.

In TCP, connection-oriented transmission requires three phases: <u>connection establishment</u>, <u>data transfer</u> and <u>connection termination</u>.

Figure 24.10: Connection establishment

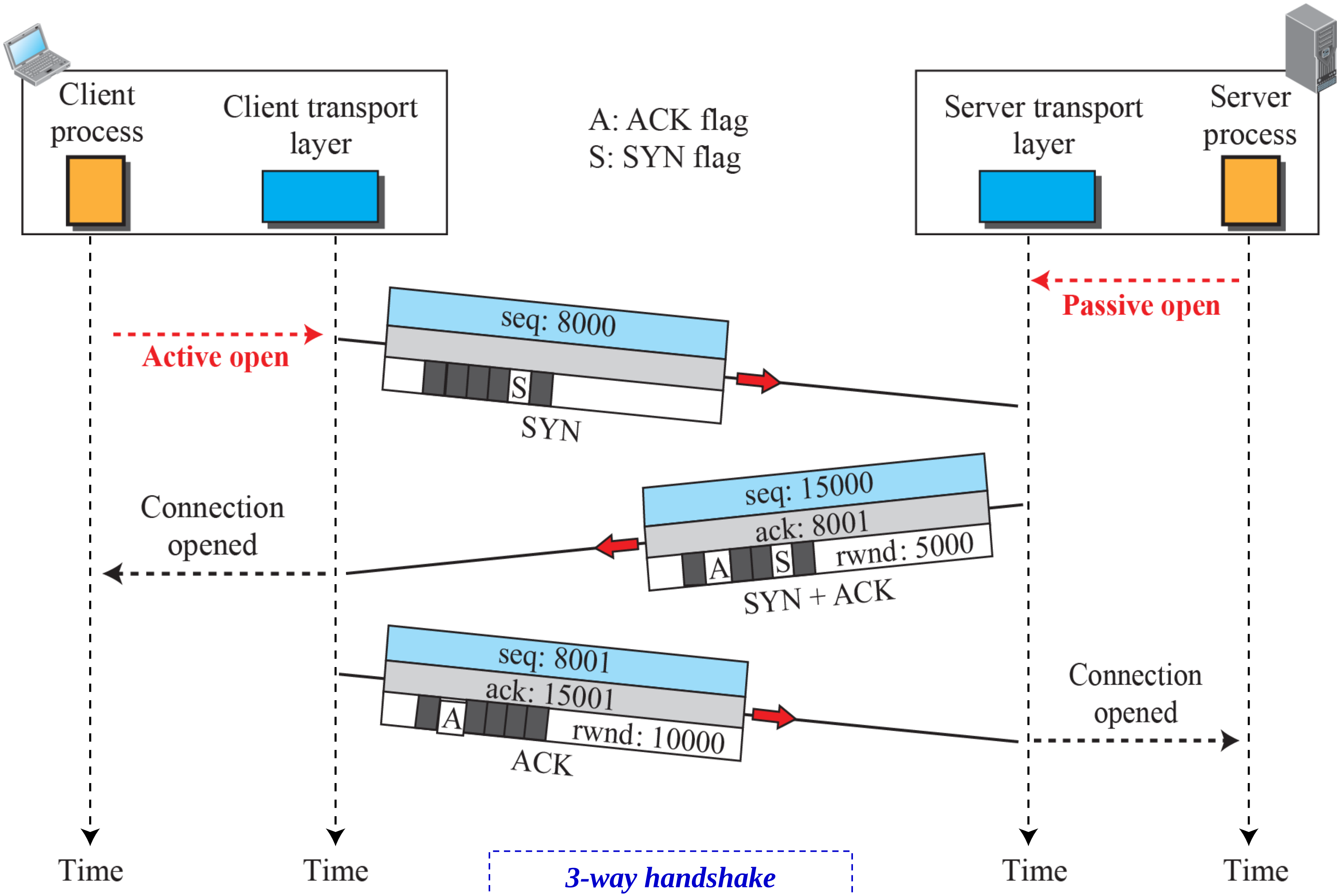


Figure 24.11: Data transfer

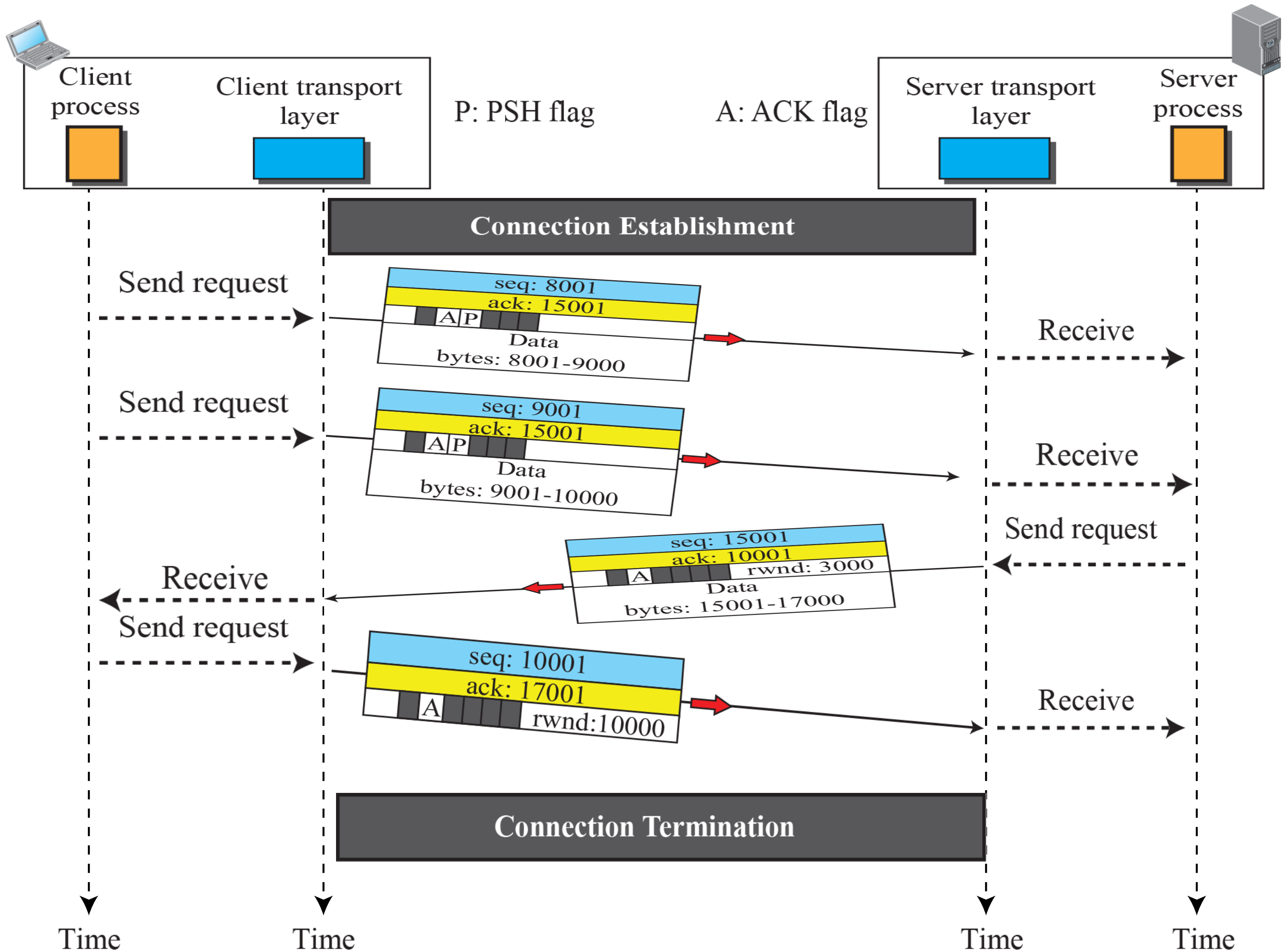


Figure 24.12: Connection termination

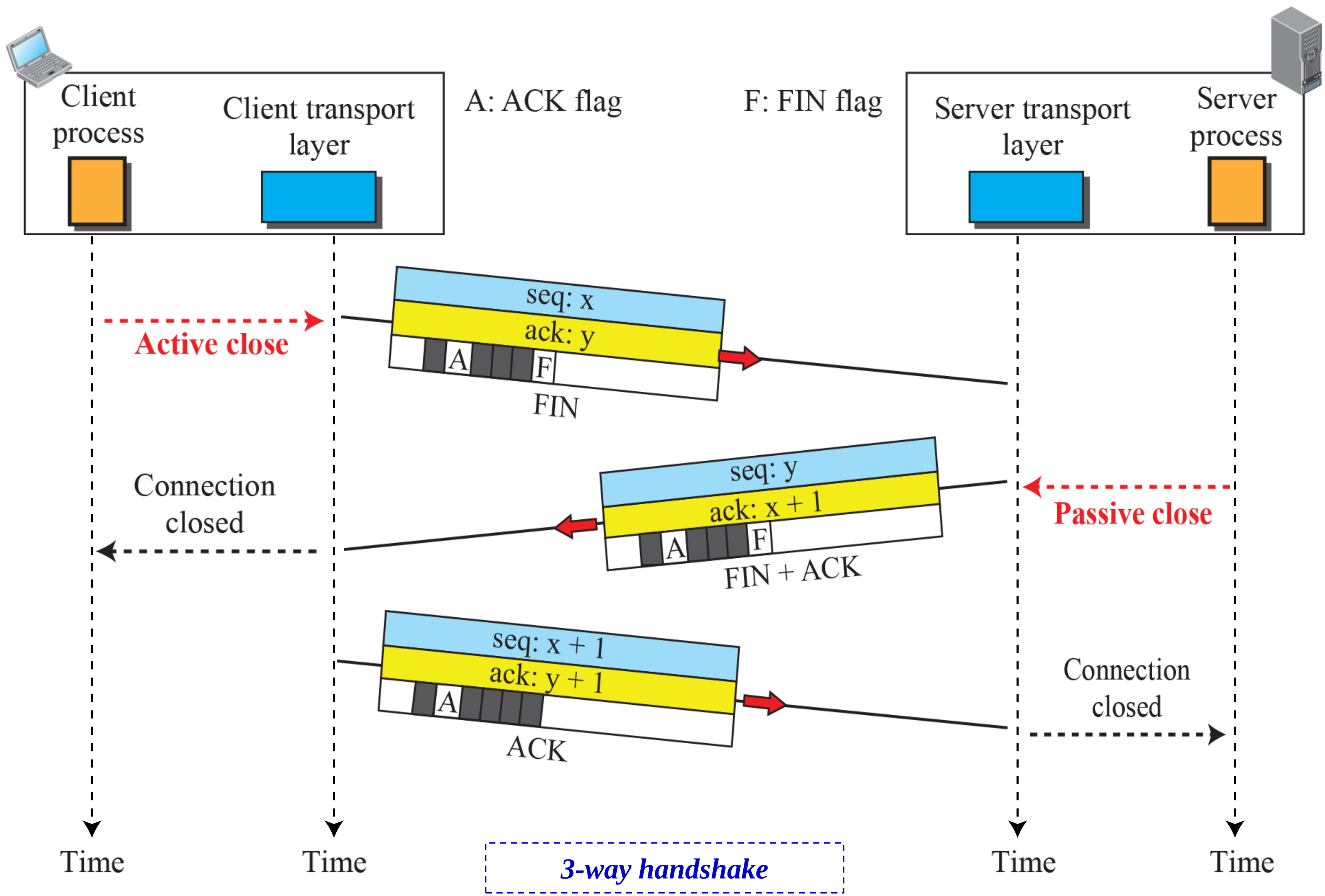


Figure 24.13: Half-close

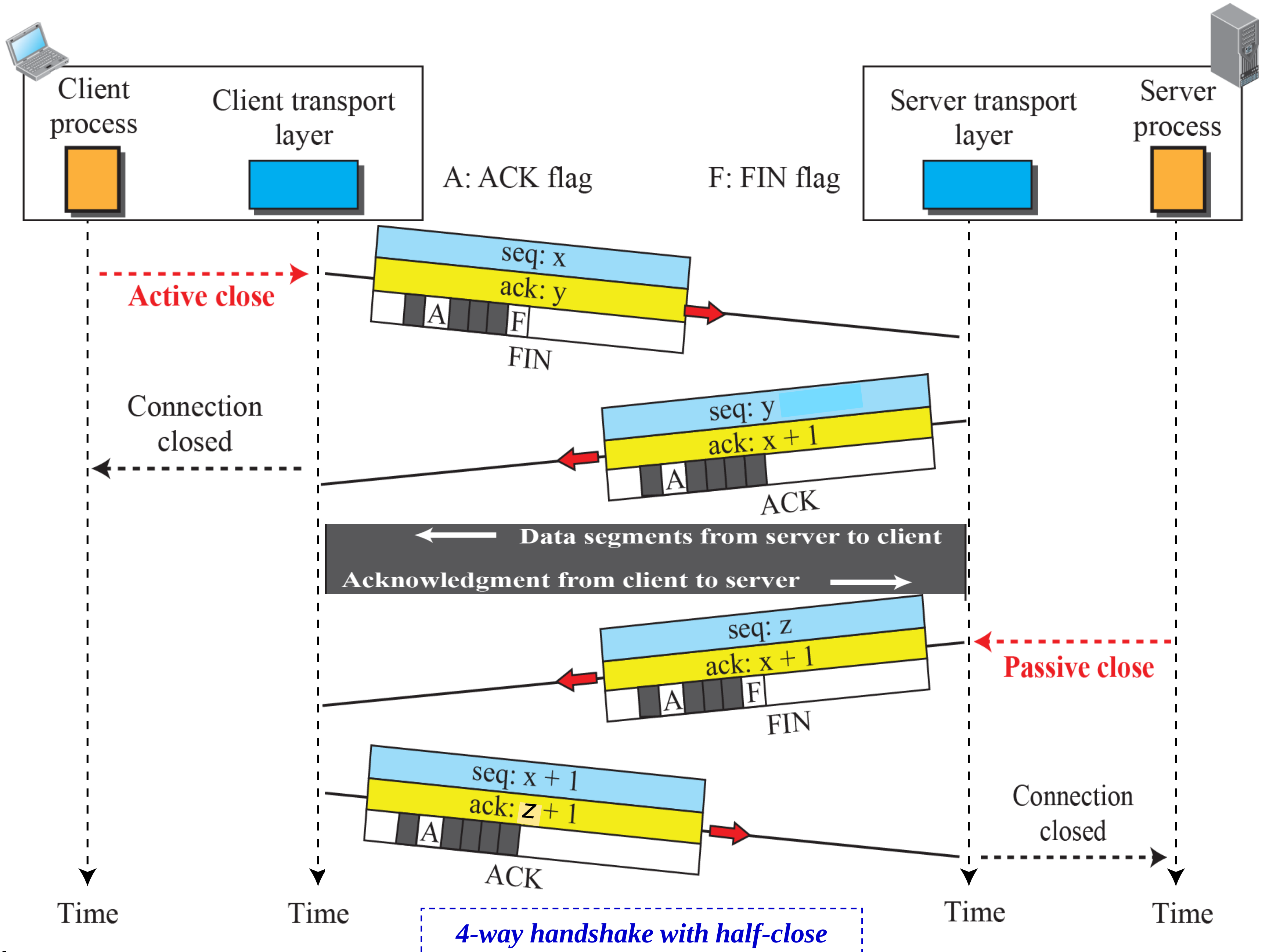
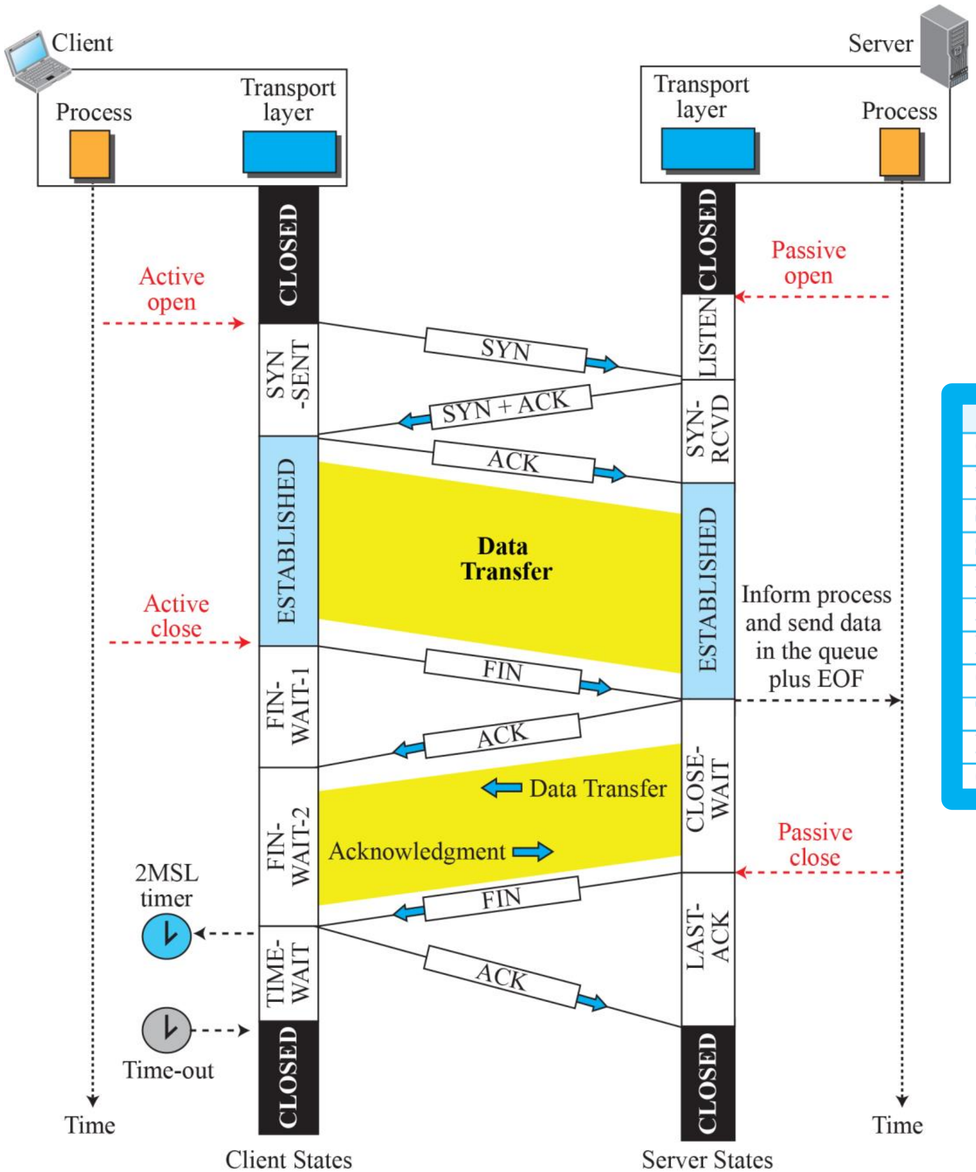


Figure 24.16: Time-line diagram for a common scenario



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

24.3.5 State Transition Diagram

To keep track of all the <u>different events</u> happening during <u>connection establishment</u>, <u>connection termination</u> and <u>data transfer</u>, TCP is specified as a <u>finite state machine</u> (FSM).

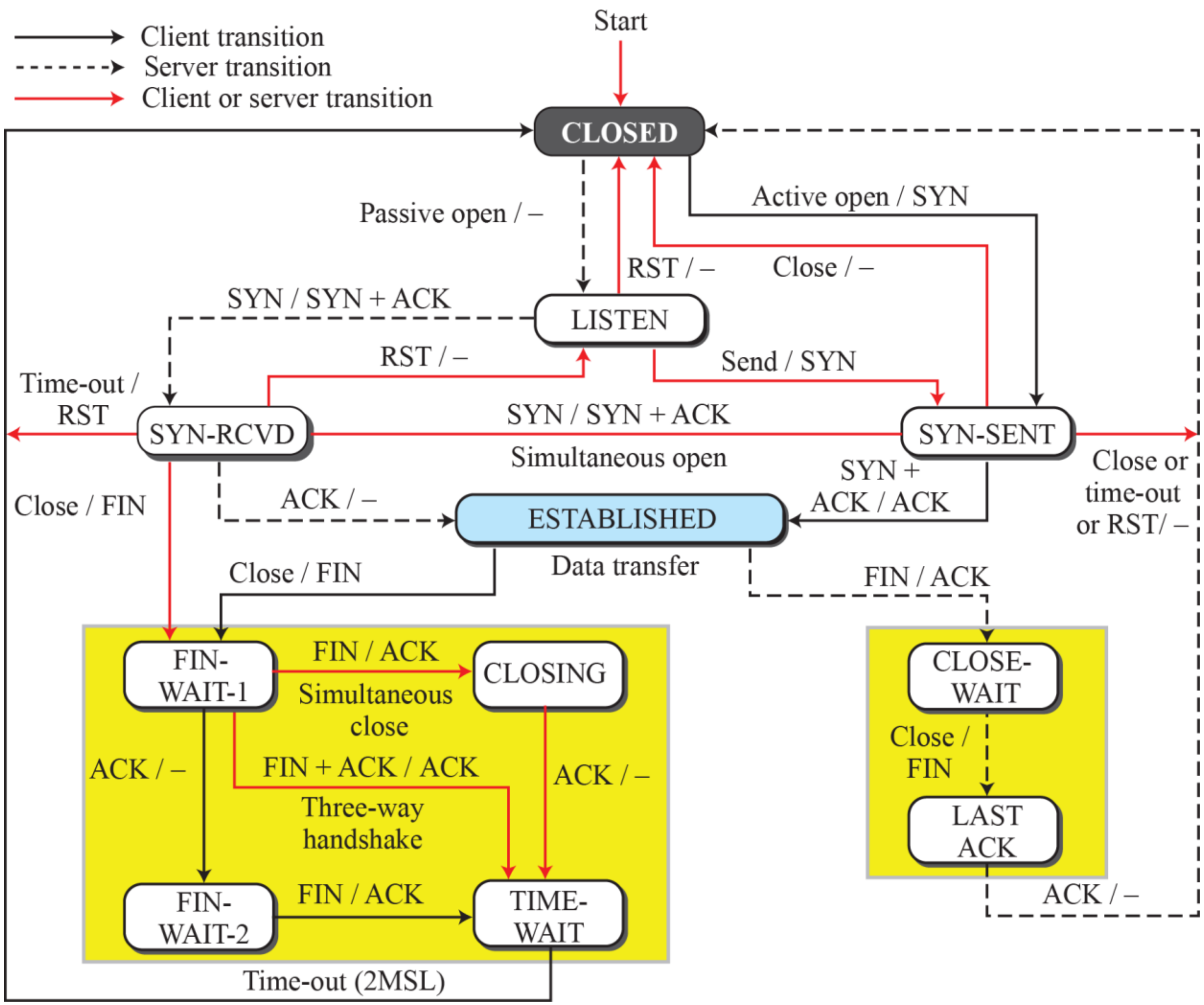


Figure 24.22: Simplified FSM for the TCP sender side

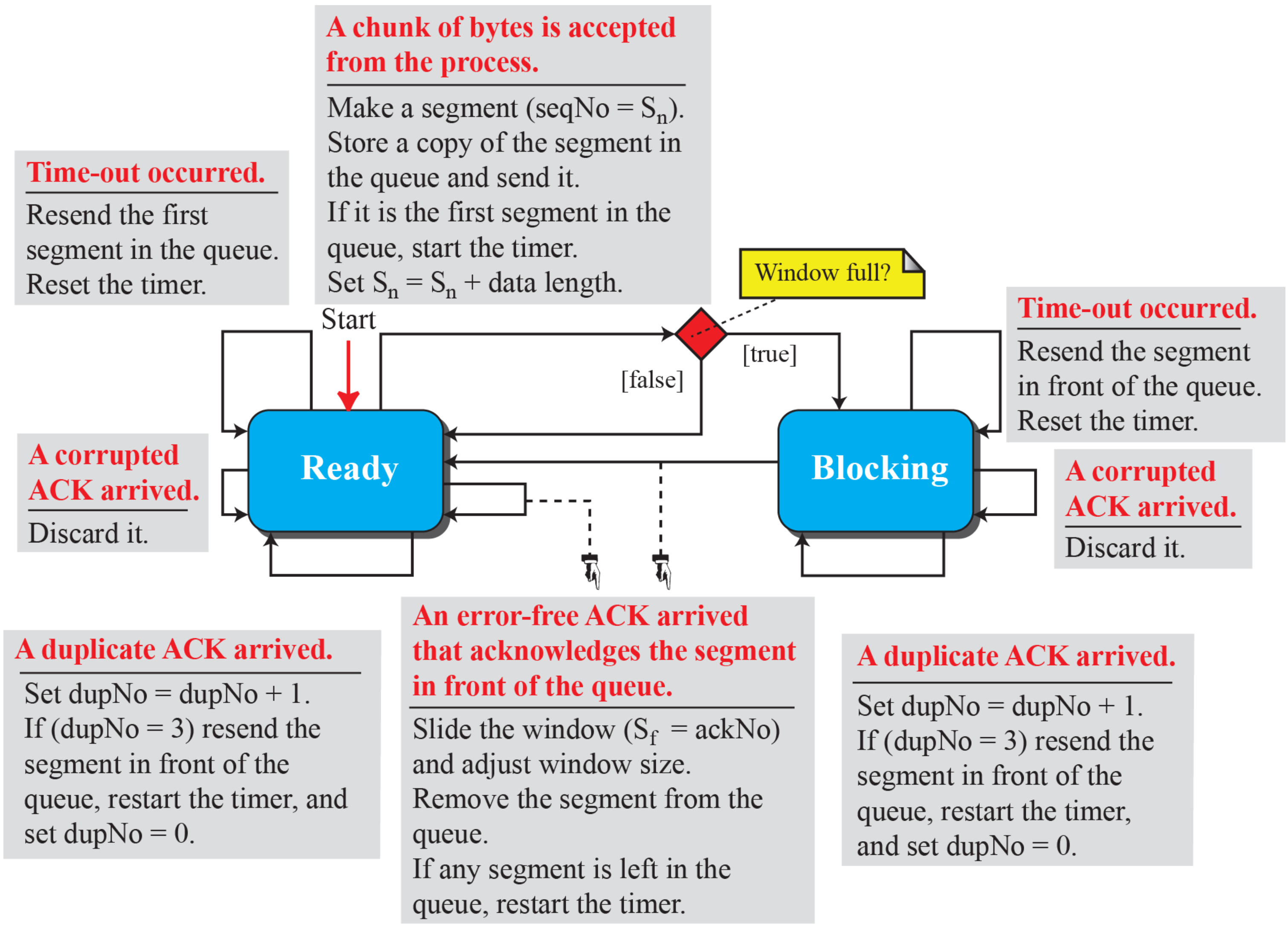
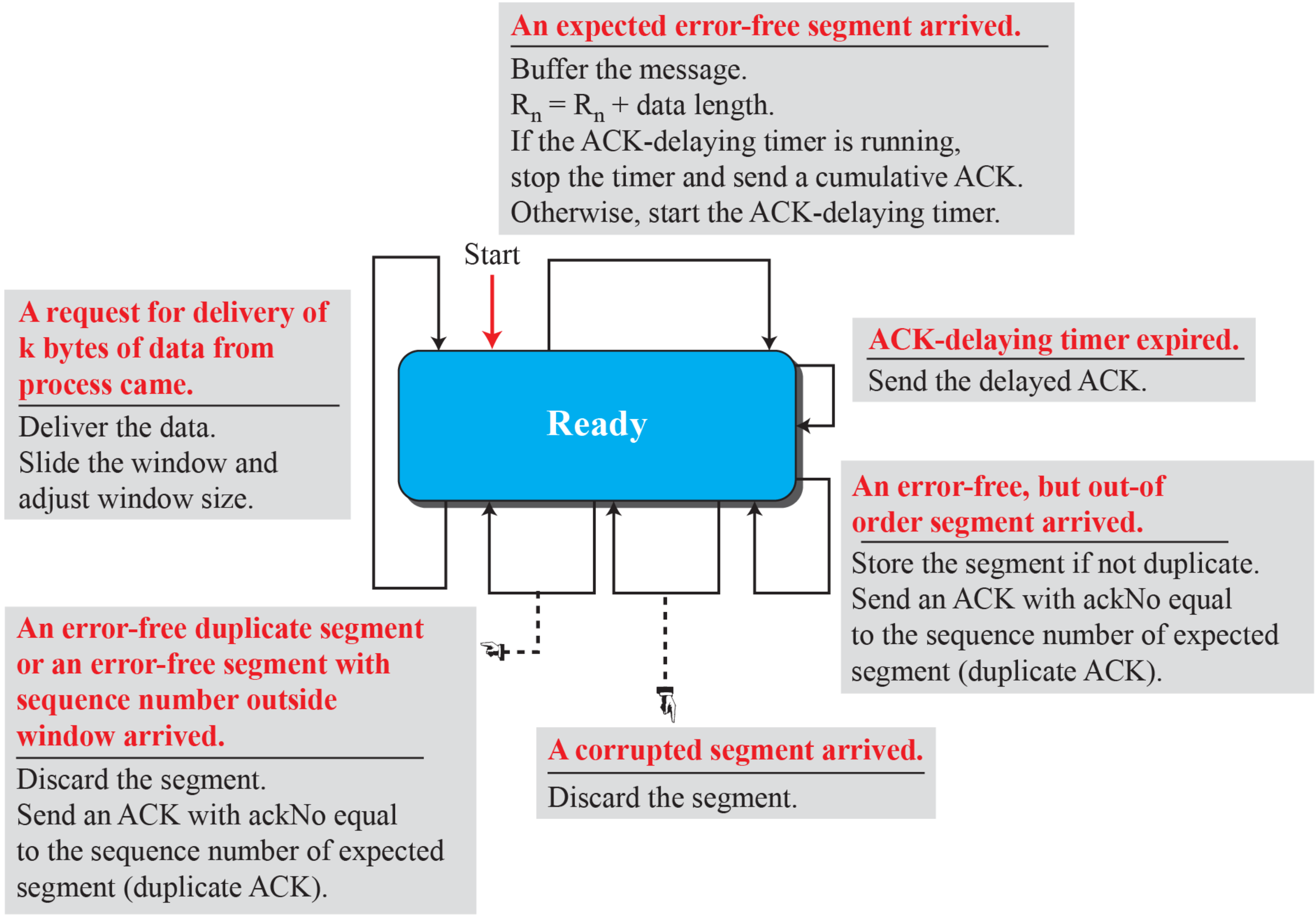


Figure 24.23: *Simplified FSM for the TCP receiver side*



"That's all Folks!"