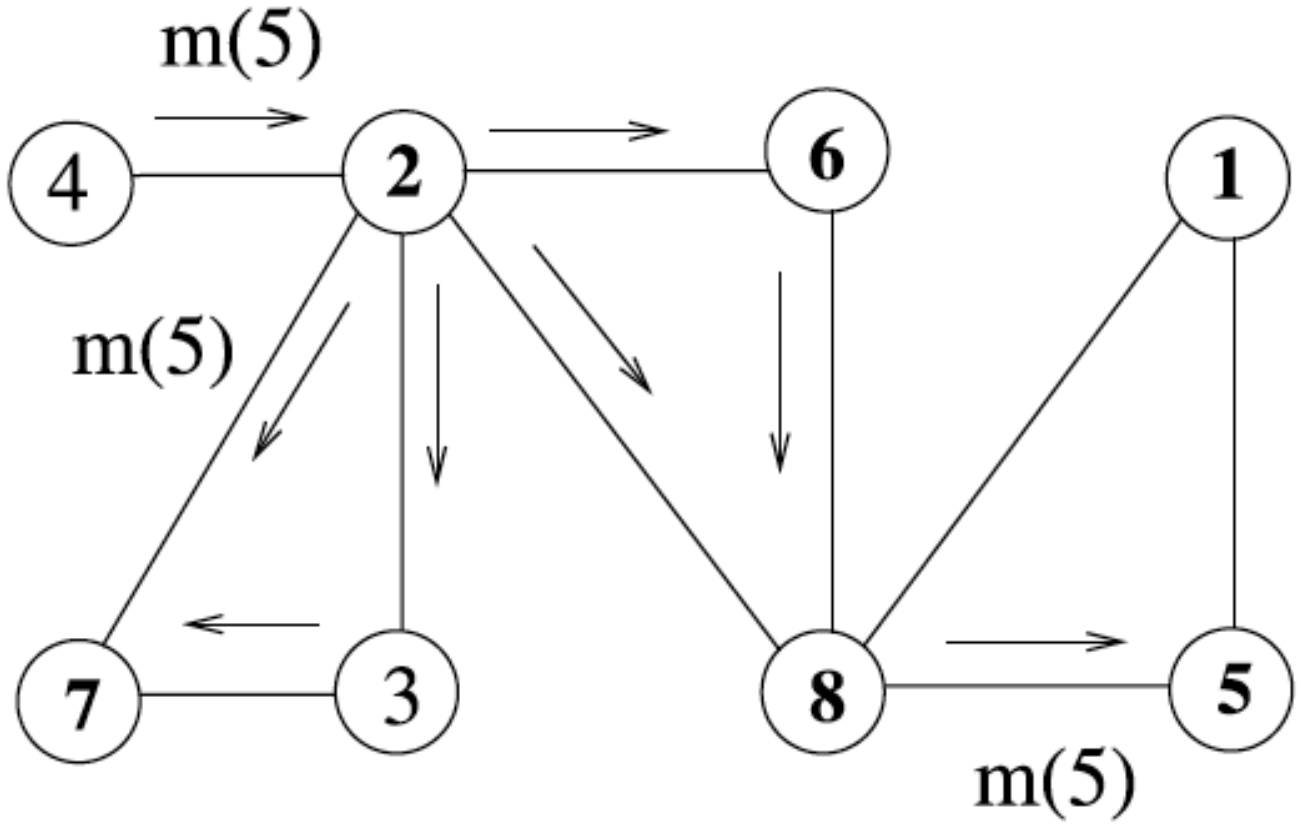
The Computational Model

Algorithm 3.1 Simple Routing Algorithm

```
1: int i, j                                   ▷ i is this node, j is the sender of the message
2: message types m(sender, dest)
3: while true do
4:     receive m(j,d)                         ▷ receive message with destination d from neighbor j
5:     if d ∈ Γ(i) then                       ▷ if destination is a neighbor
6:         send m(i, d) to d                  ▷ send message to the neighbor
7:     else send m(i, d) to Γ(i) \ {j}        ▷ else send it to all neighbors except the sender
8:     end if
9: end while
```

Simple Routing Operation



Message Passing

- Messages are crucial for the correct operation of a distributed algorithm.
  - A process p_i at node i communicates with other processes by exchanging messages only.
  - Each process pi has a state $s_i \in S$, where S is the set of all its possible states.
  - A configuration of a system consists of a vector of states as $C = [s_1, \ldots, s_n]$.
  - The configuration of a system may be changed by either a message delivery event or a computation event .

Message Passing

- Messages are crucial for the correct operation of a distributed algorithm.
  - A configuration of a system consists of a vector of states as $C = [s_1, \ldots, s_n]$.
  - Distributed system continuously goes through executions as $C_0, \phi_1, C_1, \phi_2, \ldots$
  - where ϕ_i is either a computation or a message delivery event.

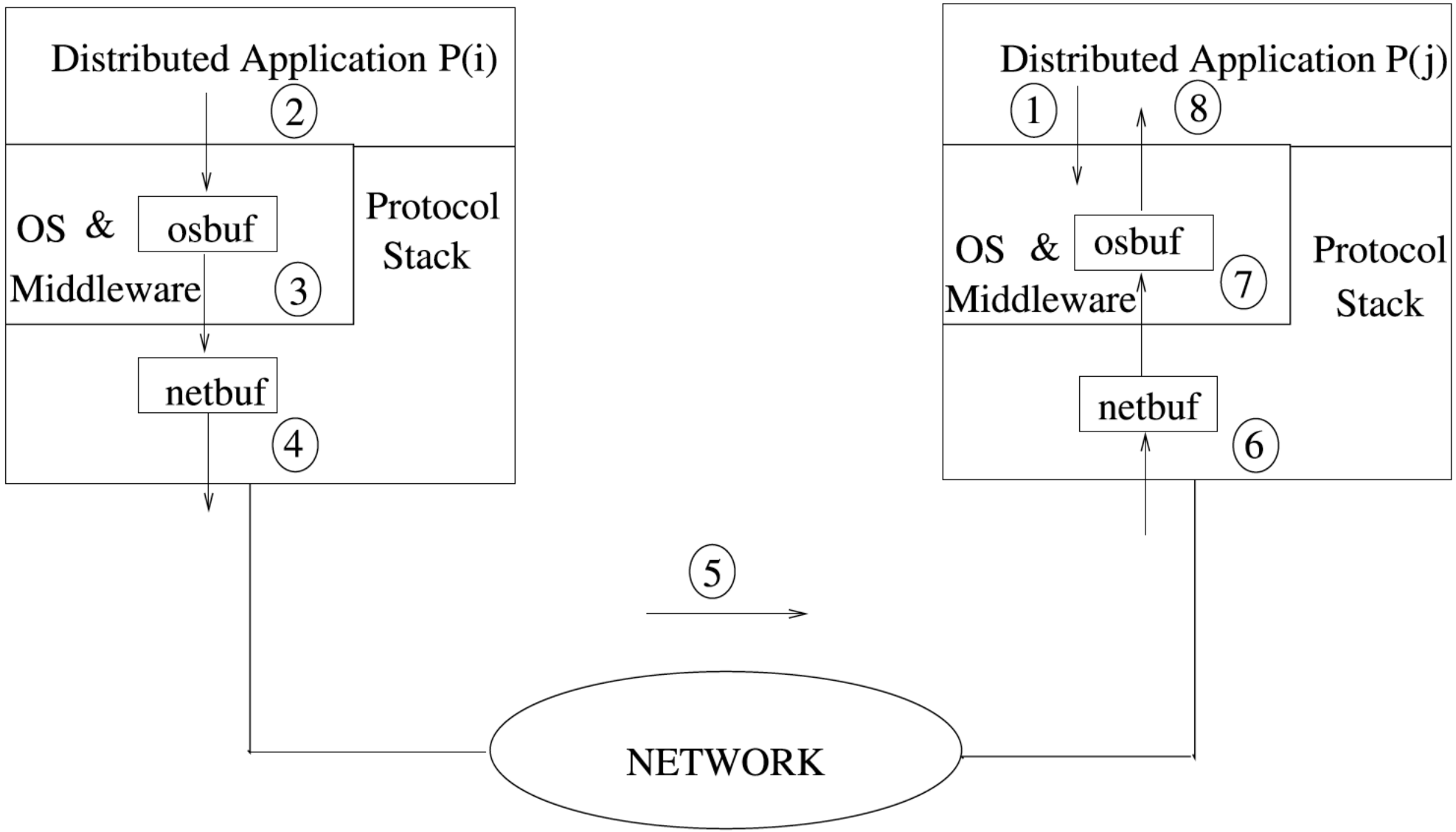
Message Passing Cont’

- A typical distributed algorithm code segment involves receiving a message and, based on the type of this message, performs a specific action as shown in Algorithm 3.2 .

- Typically, the distributed algorithm runs until some condition is met, for example, a specific message is received, or a boolean variable becomes true.

Algorithm 3.2 Distributed Algorithm Structure

```
1: while condition do
2:     receive msg(j)
3:     case msg(j).type of
4:             type_A   :   Action_A
5:             type_B   :   Action_B
6:             type_C   :   Action_C
7: end while
```


Distributed Application P(i)
2
OS & Middleware
osbuf
3
Protocol Stack
netbuf
4
5
NETWORK
Distributed Application P(j)
1
8
OS & Middleware
osbuf
7
Protocol Stack
netbuf
6

Message Passing Cont’

- The following steps are typically performed when the message msg(j, type) is delivered from a distributed application process (algorithm) P(i) at source node i to a distributed application process P(j) at destination node j as shown in Fig. 3.2 .

- 1. Receiving process P(j) executes receive(msg) and is blocked by its local operating system at node j since there are no any messages.

- 2. Sending process P(i) prepares the message by filling data , destination process , node identifiers , and type fields and invokes operating system primitive **send(msg, j)** , which copies **msg** to the operating system buffer ***osbuf*** .

Message Passing Cont’

- **3.** The operating system copies ***osbuf*** to the network buffer ***netbuf*** and invokes the communication network protocol.

- **4.** The protocol appends error checking and other control fields to the message and provides the delivery of the message to the destination node network protocol by writing contents of ***netbuf*** to the network link.

- **5.** The network delivers the data packet, possibly by exchanging few messages and receiving acknowledgements.

Message Passing Cont’

- **6.** The receiving network protocol at node j writes the network data to its buffer ***netbuf*** and signals this event to the operating system.
- **7.** The receiving node’s operating system copies data from ***netbuf*** to ***osbuf*** and unblocks the receiving process P(j) , which was blocked waiting for the message.
- **8.** P(j) is awaken and proceeds its processing with the received data.

Finite-State Machines

- A finite-state machine (FSM) or finite-state automaton is a mathematical model to design systems whose output depends on the history of their inputs and their current states, in contrast to functional systems where the output is determined by the current input only.

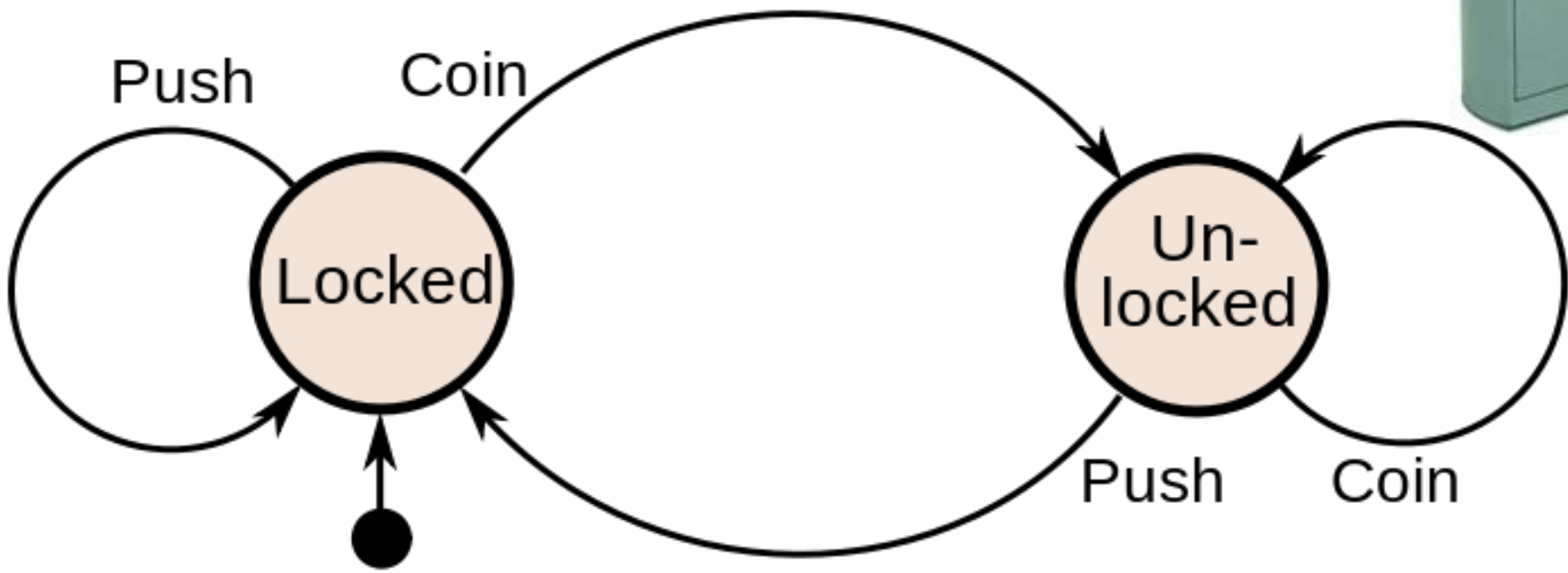
- It can only be at one state at any time called its current state .

- Upon a triggering by an event or a condition, an FSM may change its current state.

Finite-State Machines

- A deterministic FSM is a quintuple (I, S, S_0, δ, O) where

  - I is a set of input signals
  - S is a finite nonempty set of states
  - $S_0 \in S$ is the initial start state
  - δ is the state transition function such that $\delta : S \times I \rightarrow O$
  - $O \in S$ is the set of output states


Push
Coin
Locked
Un-
locked
Push
Coin

Turnstile State Table

Current State	Input	Next State	Output
Locked	coin	Unlocked	Unlocks the turnstile so that the customer can push through.
	push	Locked	None
Unlocked	coin	Unlocked	None
	push	Locked	When the customer has pushed through, locks the turnstile.

Finite-State Machines

- Moore Machine
- Mealy Machine

Finite-State Machines

- Moore Machine

  - As an example of Moore Machine, let us design an FSM that inputs a binary stringof the form 001101010. . . and at any point, determines its state based on the number of 1s received up to that point as ODD if this number is an odd number and EVEN otherwise.

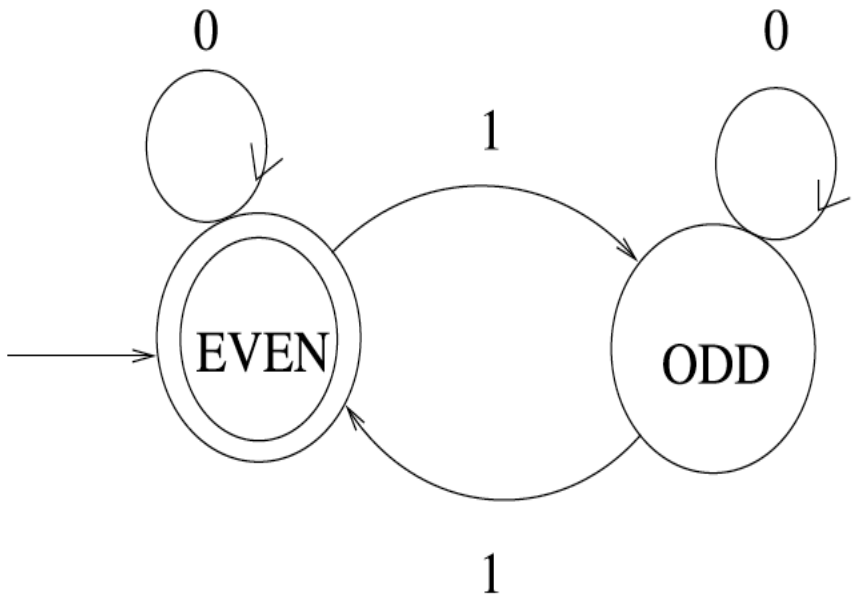


Fig. 3.3 FSM diagram of the Parity Checker

Table 3.1 State table for the Parity Checker

	0	1
ODD	ODD	EVEN
EVEN	EVEN	ODD

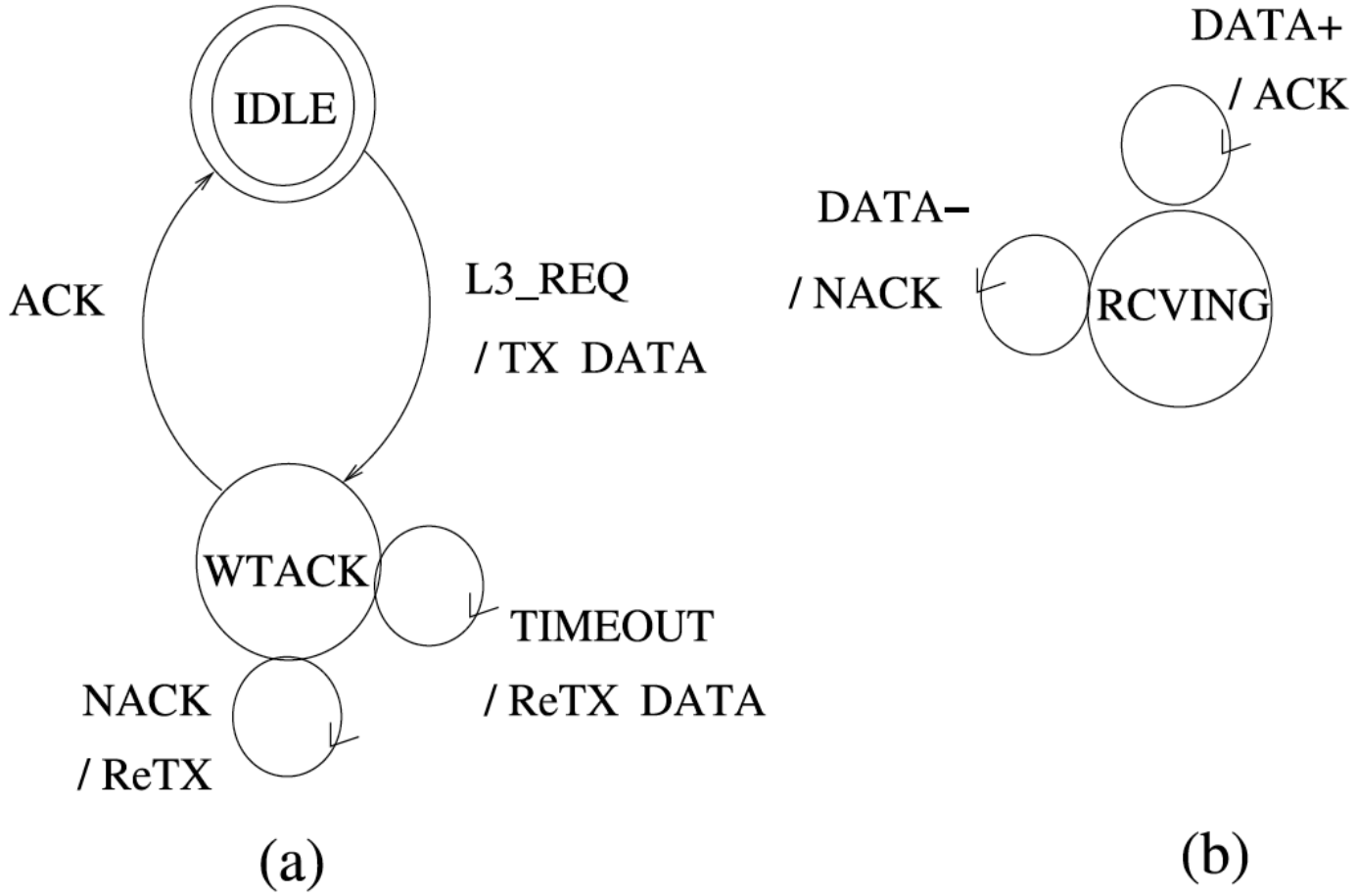
Finite-State Machines

- Mealy Machine : ex. Data Link Protocol Design :Stop and Wait Automatic Repeat Request (ARQ).

Finite-State Machines

Mealy Machine : ex. Data Link Protocol Design :Stop and Wait Automatic Repeat Request (ARQ).

Fig. 3.4 FSM diagrams of the data link protocol; (**a**) sender, (**b**) receiver



Finite-State Machines

Mealy Machine : ex. Data Link Protocol Design :Stop and Wait Automatic Repeat Request (ARQ).

Table 3.2 State table for data link protocol sender

	L3_REQ	ACK	NACK	TIMEOUT
IDLE	Act_00	NA	NA	NA
WTACK	NA	Act_11	Act_12	Act_13

Algorithm 3.3 Data Link Sender

```
1:  set of states IDLE, WTACK
2:  int currstate ← IDLE, curr_seqno ← 0
3:  fsm_table[2][3] ← action addresses

4:  procedure Act_00(frame)                          ▷ send frame first time
5:      frame.type ← DATA                            ▷ set type
6:      frame.seqno ← curr_seqno                     ▷ insert sequence number
7:      frame.error ← calc_error(frame)              ▷ calculate and insert error code
8:      send(frame) to receiver
9:      currstate ← WTACK
10: end procedure
11:
12: procedure Act_11(frame)                          ▷ frame received correctly
13:     currseq ← (currseq + 1) mod 2                ▷ increment sequence number
14:     respond to L3                                ▷ notify Layer 3
15:     currstate ← IDLE
16: end procedure
17:
```

```
18: procedure Act_12(frame)                                   ▷ re-transmission
19:     if error_count ≤ MAX_ERR_COUNT then ▷ if maximum error count is not reached
20:         frame.type ← DATA                                          ▷ set type
21:         currseq ← (currseq − 1) mod 2                ▷ set seqno to the old one
22:         frame.seqno ← curr_seqno                     ▷ insert sequence number
23:         frame.error ← calc_error(frame)        ▷ calculate and insert error code
24:         send(frame) to receiver
25:     else send error_report to Layer 3          ▷ report delivery error to upper layer
26:     end if
27: end procedure
28:
29: while true do                                                ▷ Sender main code
30:     receive msg
31:     call fsm_table[currstate][msg.type]  ▷ go to action specified by currstate and msg type
32: end while
```

Synchronization

- Hardware level
  - ❖Lock
    - SIMD

- Network Level
  - MIMD

- Network Itself

- There are also possibilities of synchronization at **operating system level**, **middleware level**, or the **distributed application level.**

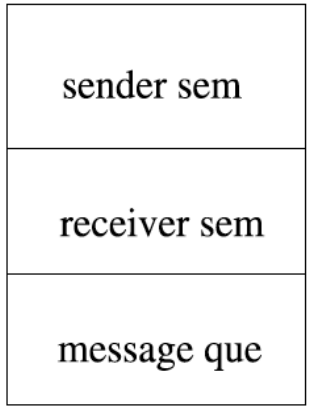


Fig. 3.7 The mailbox data structure

Algorithm 3.4 Interprocess Communication by Mailboxes

```
 1: procedure send_mbox(mbox_id)
 2:     wait on mailbox send semaphore
 3:     append message to mailbox message queue
 4:     signal mailbox receive semaphore
 5: end procedure
 6:
 7: procedure receive_mbox(mbox_id)
 8:     wait on mailbox receive semaphore
 9:     receive message from mailbox message queue
10:     signal mailbox send semaphore
11: end procedure
```

Communication Primitives

- Synchrony at operating system level is accomplished by carefully designing the communication primitives so that the sender and the receiver may be blocked or not by the operating system to yield the required behavior.
  - blocking send: The sending process is blocked by the operating system until an acknowledgement from the destination is received to confirm that the receiver has received the message, in which case the sender is unblocked.
  - Non-blocking send: The sending process continues processing after sending the message.

Communication Primitives

- blocking receive: The receiving process is blocked by the operating system if there are no messages available to it.

- non-blocking receive: The receiving process checks if it has any pending messages, and if there is a message, receives it. In any case, it continues processing.

Blocking – Non blocking

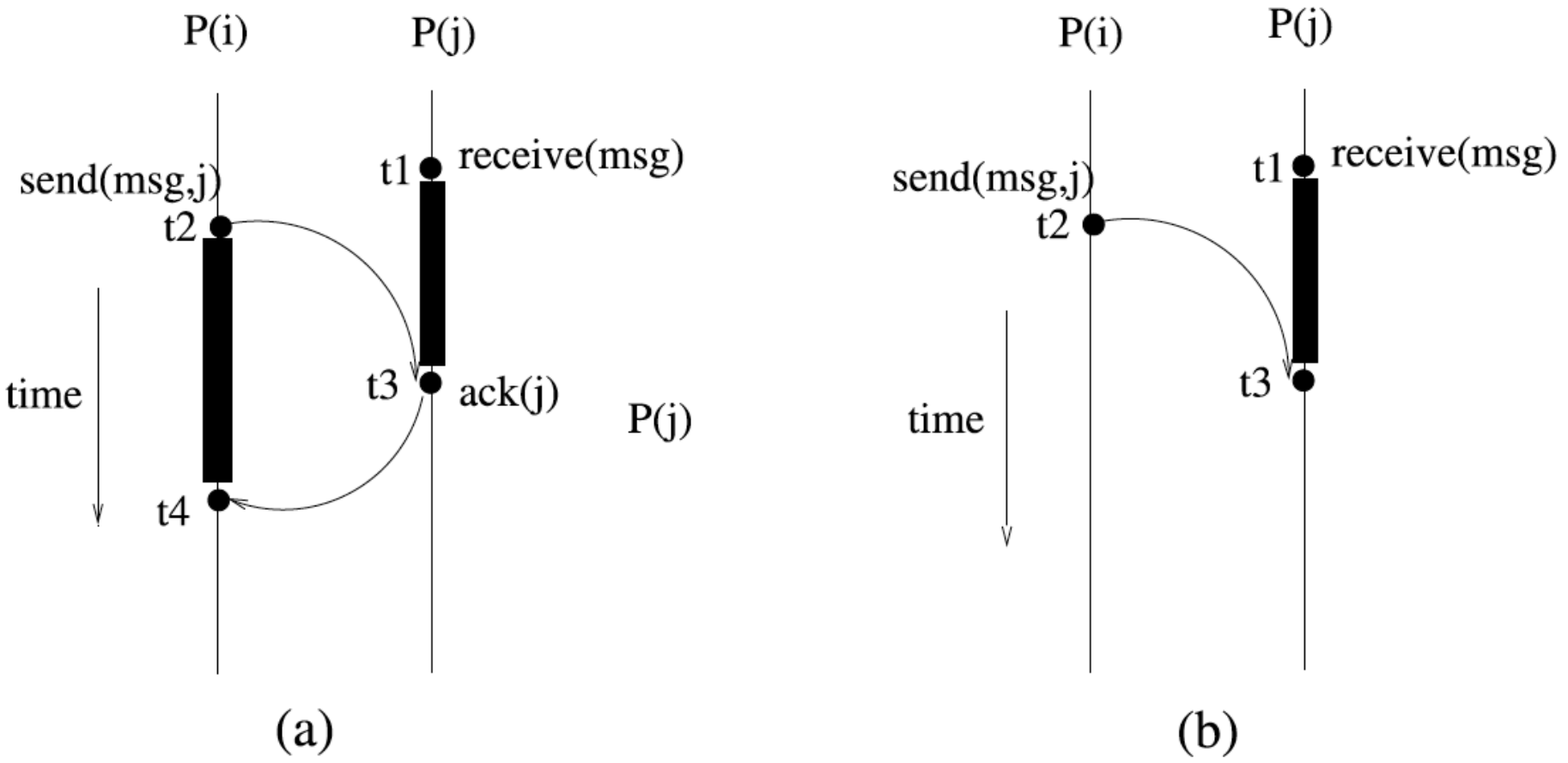


Fig. 3.6 (**a**) Blocking send and blocking receive. (**b**) Nonblocking send and blocking receive

Application Level Synchronization

- We will look into the synchronization among all processes at the application layer, and we will call this synchronization the global synchronization, which does not have any support from the operating system or the hardware.
- In some applications, support from hardware or a special middleware module called synchronizer , which provides synchrony among the processes, may be a better choice than a synchronizing protocol as in the case of concurrently initiated synchronous algorithms.

Application Level Synchronization

- May be achieved with network messages

1. Start round
2. **send** message(s)
3. **receive** message(s)
4. perform computation

Classification of distributed algorithms

	Single Initiator	Concurrent Initiator
Synchronous	SSI	SCI
Asynchronous	ASI	ACI

Performance Metrics

- Time
- Bit
- Space
- Message

Performance Metrics

- Time
  - Time(Seq _Alg)
  - Time(Synch _Dist)
  - Time(Asynch _Dist)

- For example, sending a message in a network modeled by a graph G(V,E) may require (n − 1) steps for it to reach the farthest node.

Performance Metrics

- Bit
  - For a distributed algorithm, we will mostly be interested in the maximum length of a message communicated.
  - In general, this will not be a problem unless the message is large or is enlarged as it traverses the network.

Performance Metrics

- Space Complexity
  - Space complexity of an algorithm specifies the maximum storage in bits required by the algorithm for local storage at a node.
  - This may be important if a node holds large tables as in the case of routing algorithms.

Performance Metrics

- Message Complexity
  - Number of messages exchanged is a good indicator of the cost of communication for the distributed application.
  - The cost incurred during the message communication of a distributed algorithm is often considered as the dominant cost of the algorithm since time spent in message transmissions is orders of magnitude higher than the time spent for local computations.

Algorithm 3.5 Sample_SSI

```
 1: boolean finished, round_over ← false
 2: message type round, info, upcast
 3: while ¬round_over do
 4:     receive msg(j)
 5:     case msg(j).type of
 6:             round:    send info(i) to all neighbors
 7:                       receive info from all neighbors
 8:                       do some computation, finished ← true
 9:             upcast:   if upcast received from all children and finished then
10:                           send upcast to parent
11:                           round_over ← true, finished ← false
12: end while
```

Analyses of Algorithm 3.5

- Each round requires *2(m+n-1)* messages;
- Number of rounds depend on the diameter d. Time O(d)
- So in total *2d(m+n-1)* messages
- T may have n − 1 as its maximum depth.
- *2n(n − 1)* time steps for *n − 1* rounds