Distributed Algorithms for Computer Networks

Chapter 4

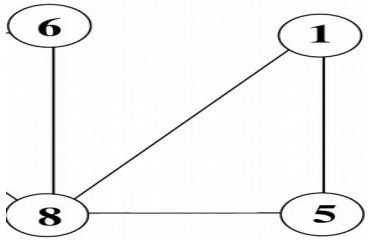
Computational Model

Dr. Amjad Hawash

Dr.Ahmed Awad

Computational Model Basics

- Aims to model the application software, middleware, and the network to deliver messages between different nodes of a distributed system.
- A distributed model is typically modeled as a graph whose vertices represent the computation nodes and the edges represent the links between them.
- A distributed algorithm runs at each node of the network graph and cooperates with other nodes to accomplish a common task.

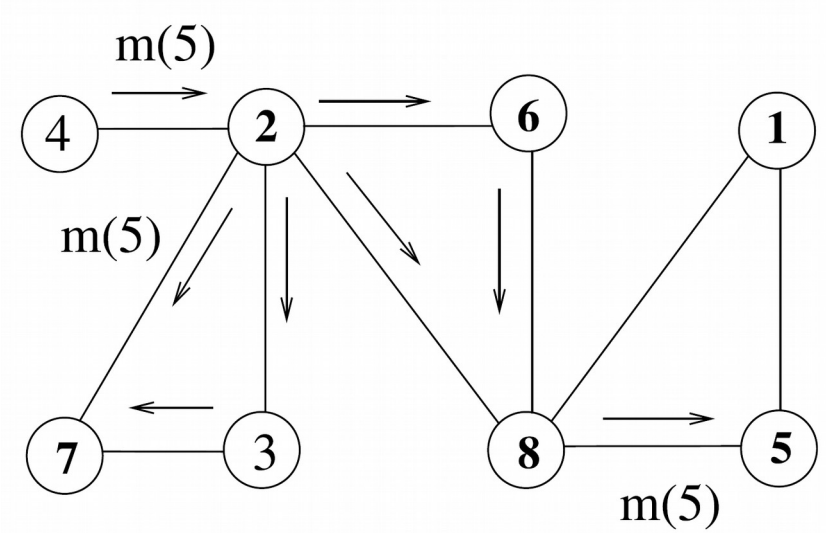


Simple Routing Algorithm

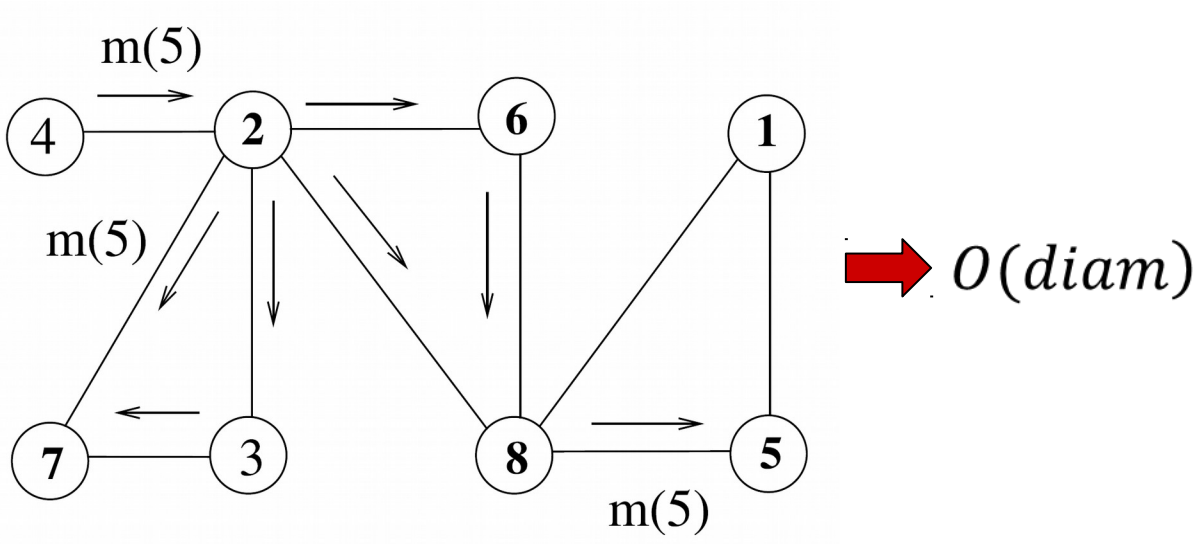
- Nodes only know their neighbors
- When a node receives a message

➔ Simply forward this message to all of its neighbors except the one it has received the message from.

➔ If the destination is one of its neighbors, it sends the message directly for it.



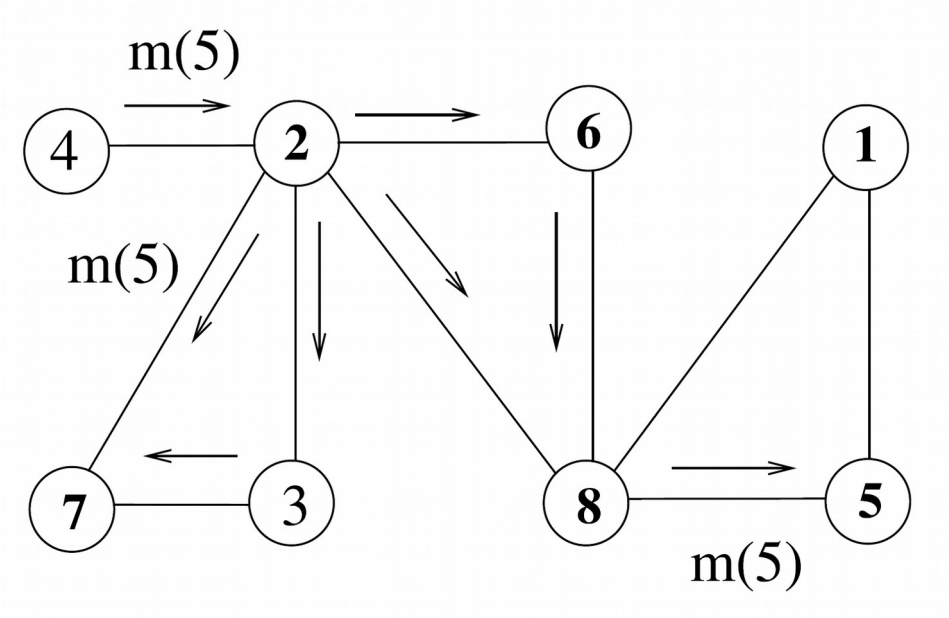
Simple Routing Algorithm (2)



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 Algorithm (3)

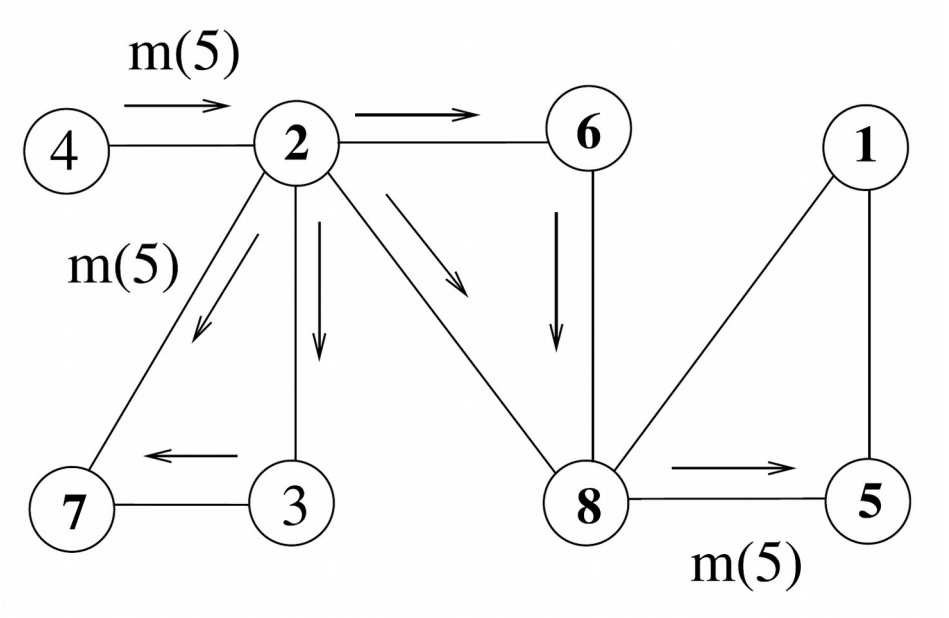


- Drawback of the algorithm:

  ➔ A node may send/receive the message more than once.

  ➔ The network will be flooded with duplicate messages.

Simple Routing Algorithm (4)



- Solution:

  ➔ Sequence number.

  Each node can check whether it has seen the seq value from some node before. If yes, it will be discarded.

Message Passing

- A process p_i at node i communicates with other processes by exchanging messages only.
- Each process p_i 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*.
- A 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.

Distributed Algorithm Code Segment

- Receiving a message.
- Based on the type of the message perform a specific action.
- The algorithm runs until some condition is met, for example:

  ➔ A specific message is received.

  ➔ Some Boolean variable becomes true.

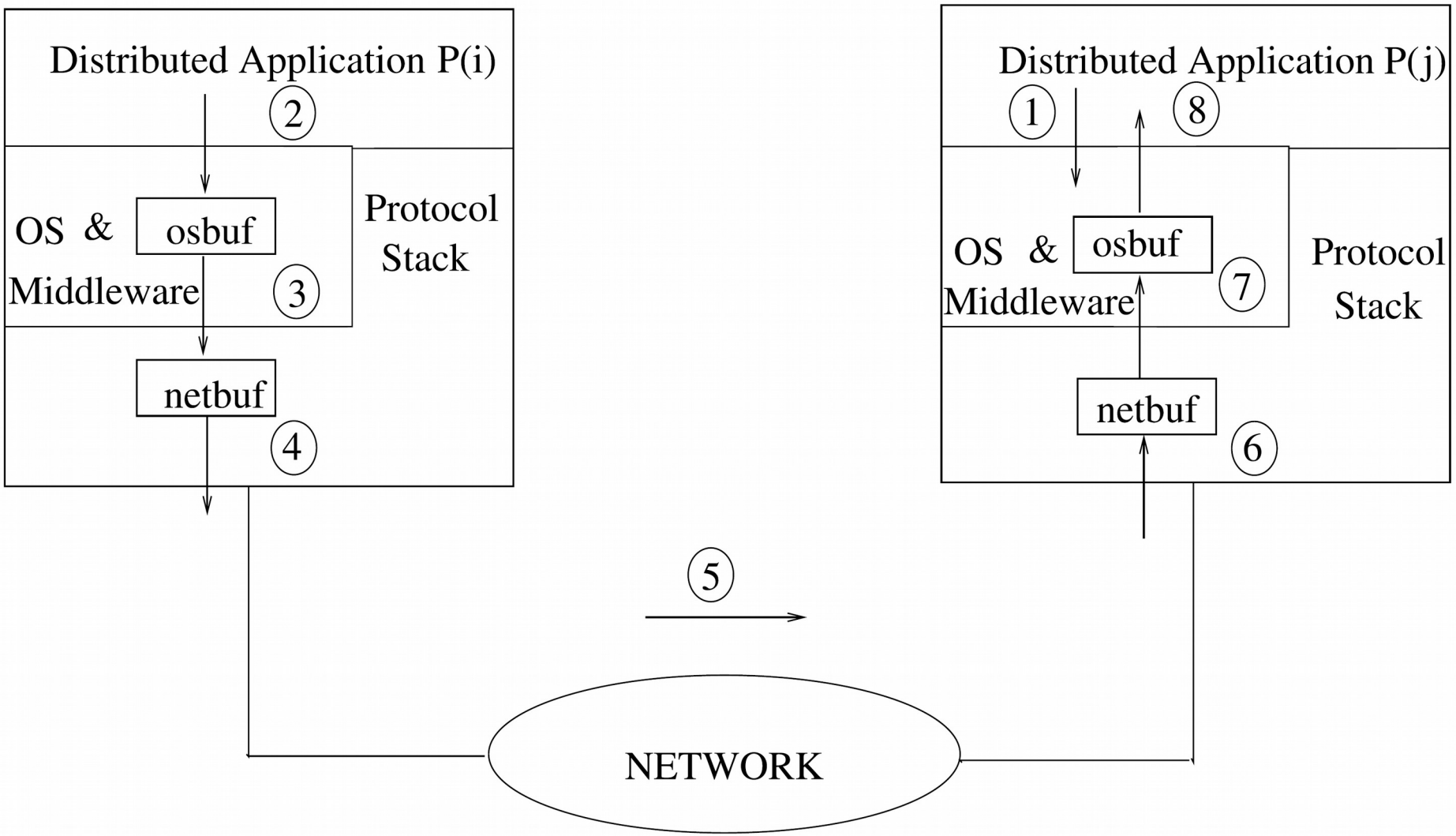
Distributed Algorithm Structure

- Algorithm running at node i:

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
```

Steps of Message Delivery



Steps of Message Delivery

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*.
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.
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.

Delivering a Sequence of Messages

- First-In-First-Out (FIFO) Delivery
  - Deliver the messages in **sequence** to the required node.
- Non-First-In-First-Out (Non-FIFO) Delivery
  - Deliver the messages in **random** order.
- Buffered Communication
  - Using buffers between the application, operating system, and the network protocol.
- Unbuffered Communication
  - The message is written/received directly to/from the network.

Finite State Machine (FSM)

- Mathematical model to design systems whose output depends on the **history** of their inputs and their **current states**.
- Has a number of states, and it can be only in one state at a time.
- Upon triggering an event, a FSM can change its current state.
- Represented as:
  - Directed graph.
  - State table.

Finite State Machine (FSM)

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

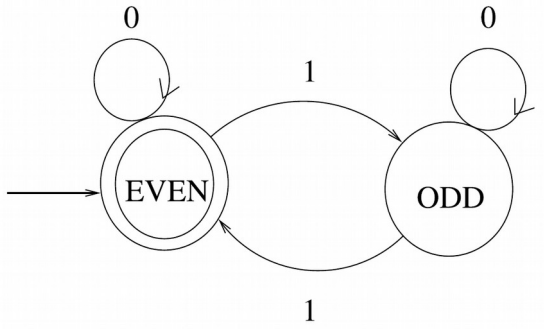
- I is a set of input signals
- S is a finite nonempty set of states
- $S_0 \in S$ is the initial start state
- δ is the state transition function such that $\delta : S \times I \rightarrow O$
- $O \in S$ is the set of output states

- Types of FSM:
  - **Moore Machine**: The output is the new decided state.
  - **Mealy Machine**: Provides an output and a new state as a result of being triggered by some action.

Moore Machine Example: Parity Check

- Takes a binary string as input.
- If the number of 1s is even, it will be in **EVEN** state.
- Otherwise, it will be in **ODD** state.

State Diagram

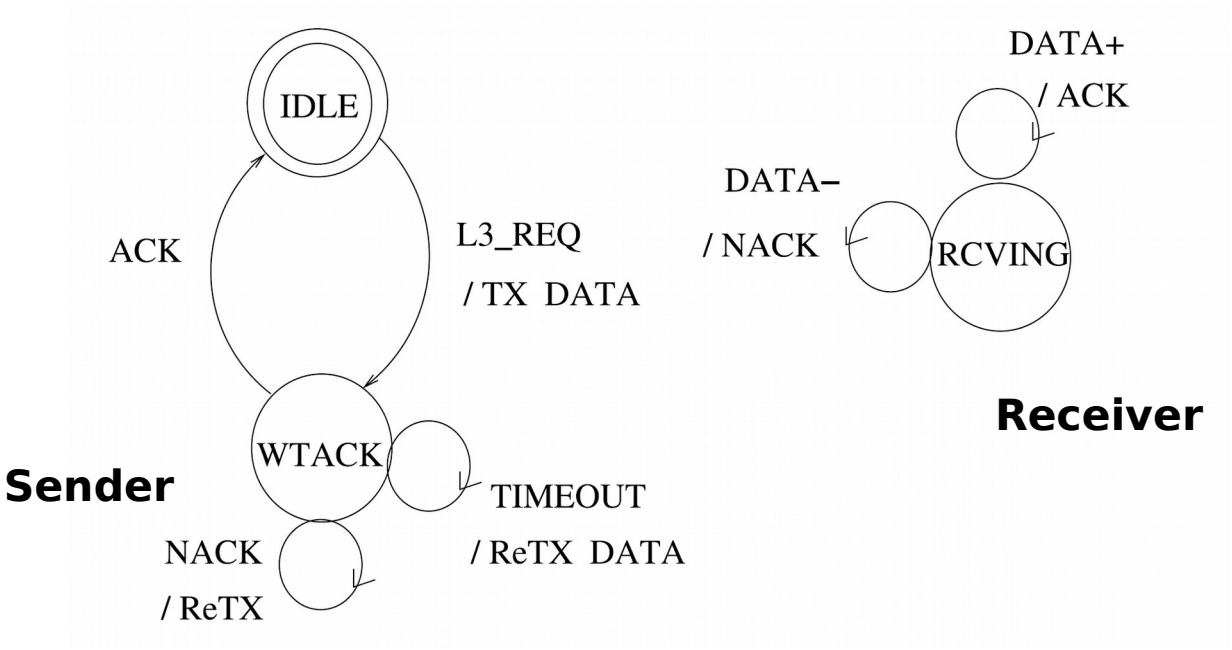


State Table

	0	1
ODD	ODD	EVEN
EVEN	EVEN	ODD

Mealy Machine Example: Data Link Protocol Design

- Stop and Wait Automatic Repeat Request (ARQ) :
  - Data link layer protocol.
  - Error control
  - Wait for acknowledgement from the receiver before sending the next frame.
  - A single frame in transmission at a time.
  - Duplicate packets are avoided using odd and even sequence numbers.



Mealy Machine Example: Data Link Protocol Design

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

Mealy Machine Example: Data Link Protocol Design

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
```

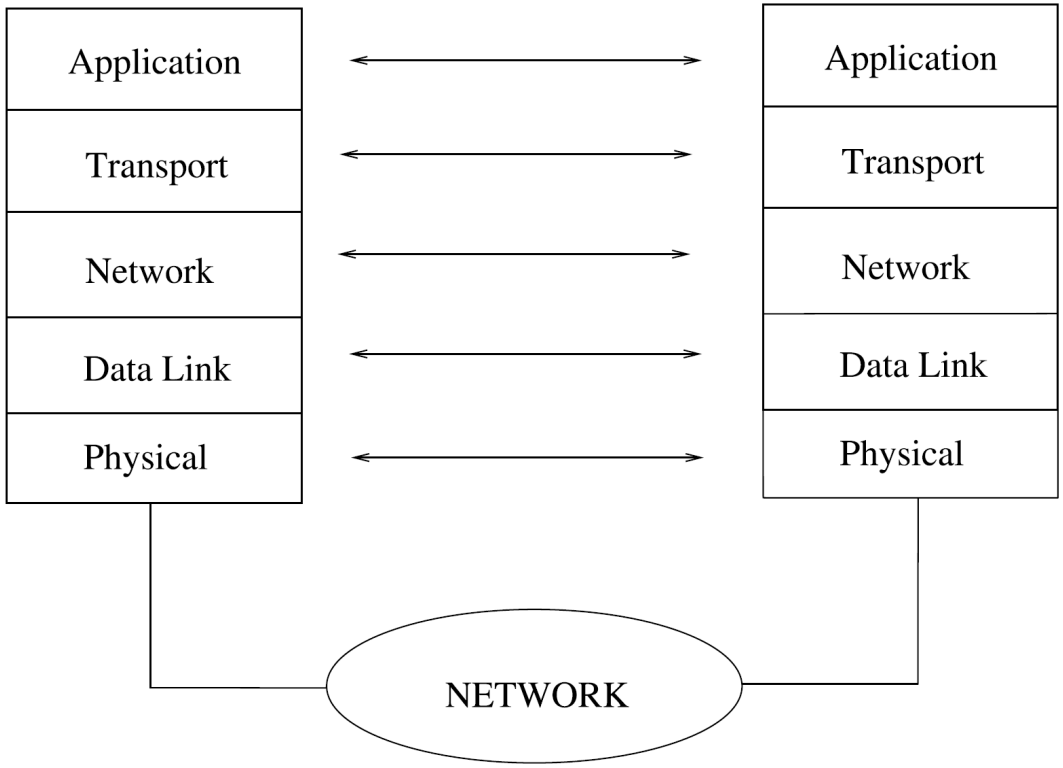
Synchronization- Hardware Level

Hardware level: Processors execute in lock step, and the next step of execution is not enabled until all nodes finish their current execution. This is called Single Instruction Multiple Data (SIMD) which is not feasible for distributed systems.

Multiple-Instruction-Multiple Data (MIMD): Do not reply on hardware synchronization and nodes work autonomously (More realistic in distributed computing systems).

Synchronization- Network Protocol Level

A typical network protocol at some layer at the sender side communicates with the same layer of the destination node by applying error checking codes, receiving ackonwledgements, and retransmissions.

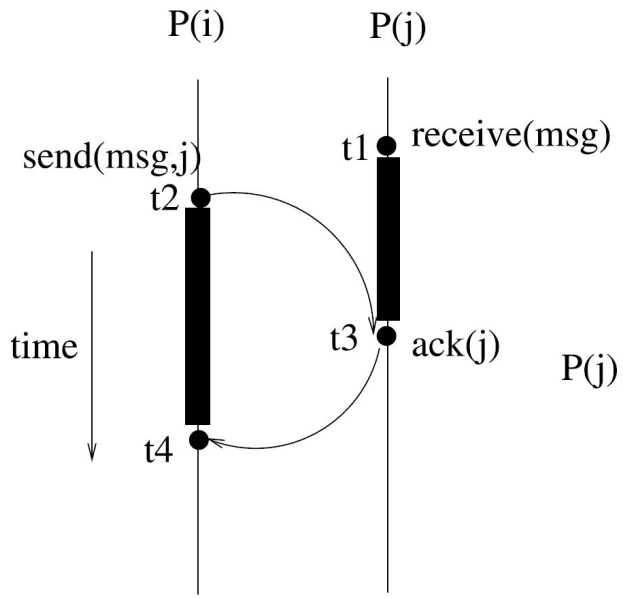


Communication Primitives

Blocking send: The sending process is blocked by the operating system until an acknowledgement from the destination is received to confirm its reception. In this case, the process is unblocked.

Blocking receive: The receiving process is blocked by the operating system if there are no messages av

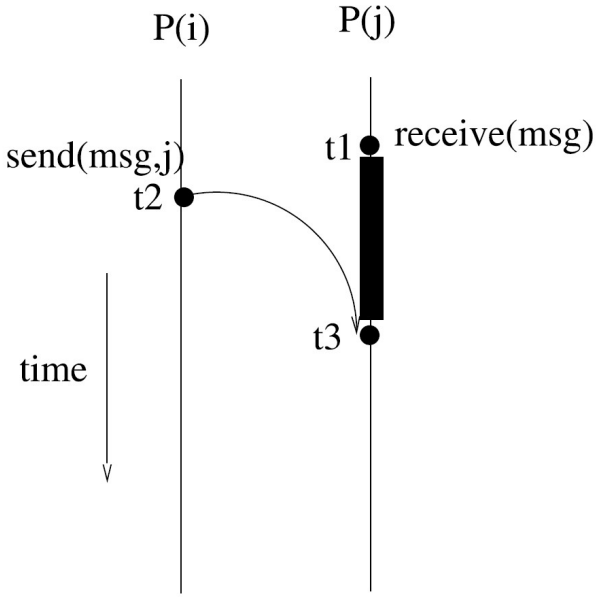
Blocking send



Communication Primitives (2)

Non-blocking send: The sending process continues processing after sending the message.

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



Non-blocking send

Mailboxes

Mailbox: A depository place for messages which is used for indirect inter-process communications. It is mainly used for **non-blocking receive**.

Mailbox is protected by a **semaphore** data structure which consists of an integer and a process queue.

- Two operations:
  - Wait
  - Signal

Mailboxes

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
```

Application Level Synchronization

- Known as **global synchronization**.
- Does not have any support from the OS or the hardware.
- Used in many applications.
- Achieved by the use of special protocol messages.
- **Lock-step** fashion in rounds as follows:
  - Start round.
  - Send messages.
  - Receive messages.
  - Perform computation.

  ➔ The next round can be only started after all messages from previous nodes are delivered and all computations have been performed.

Application Level Synchronization (2)

- **Root**: Central node that initiates a round, and gathers special message in an accumulated manner from all nodes, to start then a new round.

- **Single-Initiator Algorithm**: The distributed algorithm is started by a single designated process (initiator).

- **Concurrent-Initiator Algorithm**: The distributed algorithm is started by concurrent initiators.

- Ex ım:

	Single Initiator	Concurrent Initiator
Synchronous	SSI	SCI
Asynchronous	ASI	ACI

Execution Modes of Distributed Algorithms

- **Synchronous Single-Initiator (SSI):** Synchronous distributed algorithm started by a single initiator.
  - Easy to analyze.
  - Requires synchronization operation either at hardware level, middleware, or by the additional of special protocol control messages.
- **Asynchronous Single-Initiator (ASI):** Asynchronous distributed algorithm started by a single initiator.
  - Does not need any synchronization.
  - Termination conditions should be carefully designed.
- **Synchronous Concurrent-Initiator (SCI):** An algorithm that is executed synchronously under the control of concurrent initiators.
  - May require hardware support.
- **Asynchronous Concurrent-Initiator (ACI):** The most versatile type of algorithms.
  - Synchronization at certain points during execution may be complicated.

Performance Metrics of Distributed Algorithms

- **Time Complexity.**
- **Bit Complexity.**
- **Space Complexity.**
- **Message Complexity**

Time Complexity

- **For sequential algorithm:** The number of steps needed for the algorithm to finish.

- **For synchronous distributed algorithm:** The number of rounds required for the algorithm to finish in the worst case.

- **For asynchronous distributed algorithm:** The number of steps needed for the algorithm to finish in its worst case.

Bit Complexity

- The maximal length of a message communicated in the distributed system.

- The problem is when the message is large or is enlarged as traverses the network.

- For example, some applications require that a special message includes the node identifier for each node it traverses.

- ➔ Bit complexity ***O(nlogn)***

Space Complexity

- The maximum storage in bits required by the algorithm for local storage at a node.

- Important if a node holds large tables as in the case of routing algorithms.

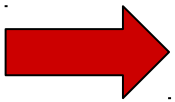
Message Complexity

- The number of messages exchanged in the communication for the distributed application.
- This cost is dominant because the time spent in message transmissions is order of magnitude higher than the time spent for local computations.
- Calculated by measuring the number of messages traverse the edges of the graph.

Example: SSI Algorithm

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
```



What is the message complexity????