DISTRIBUX

OPERATING SYSTEM

SUBJECT : OPERATING SYSTEM

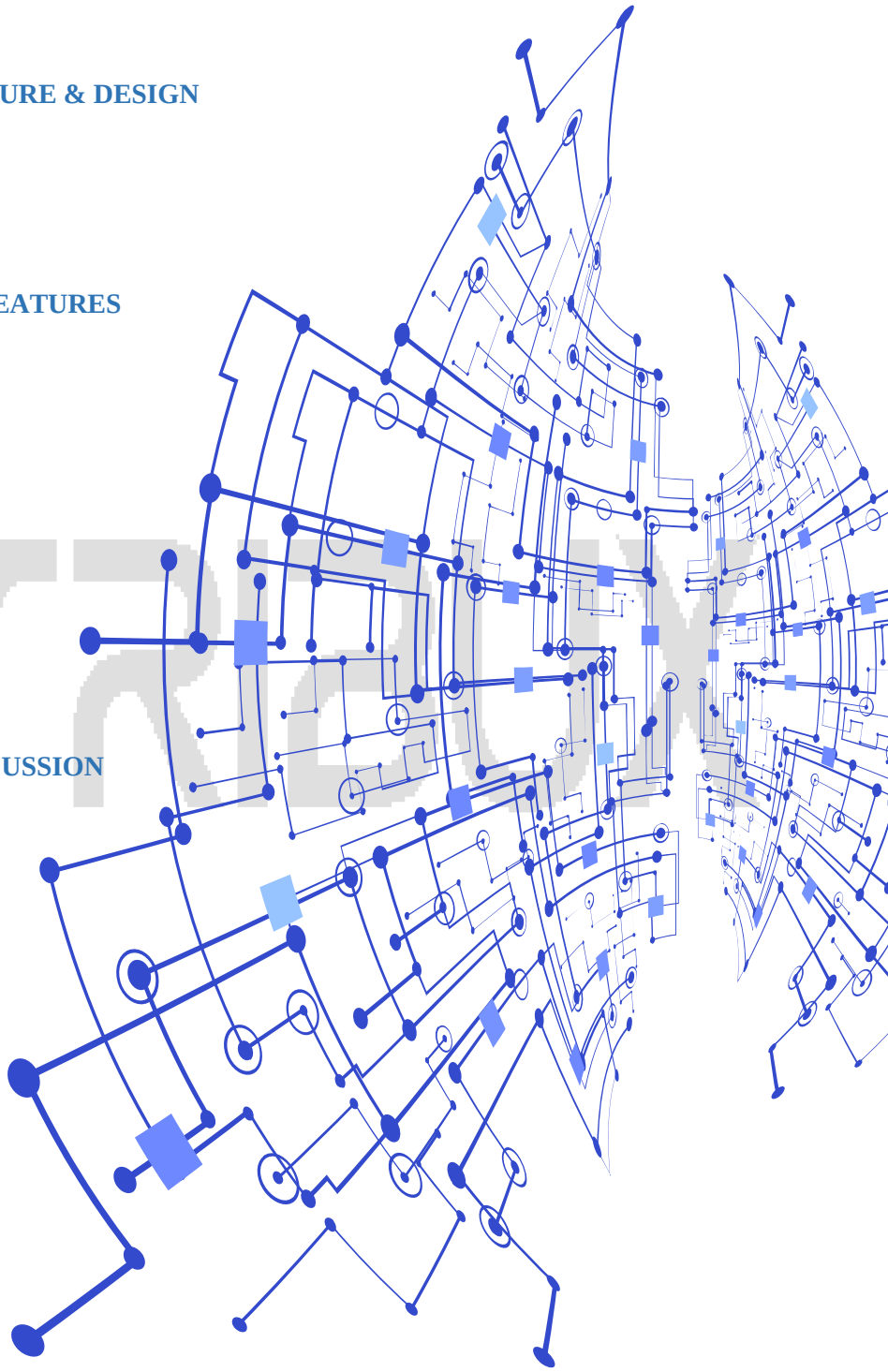
DESIGN : DISTRIBUTED OPERATING SYSTEM

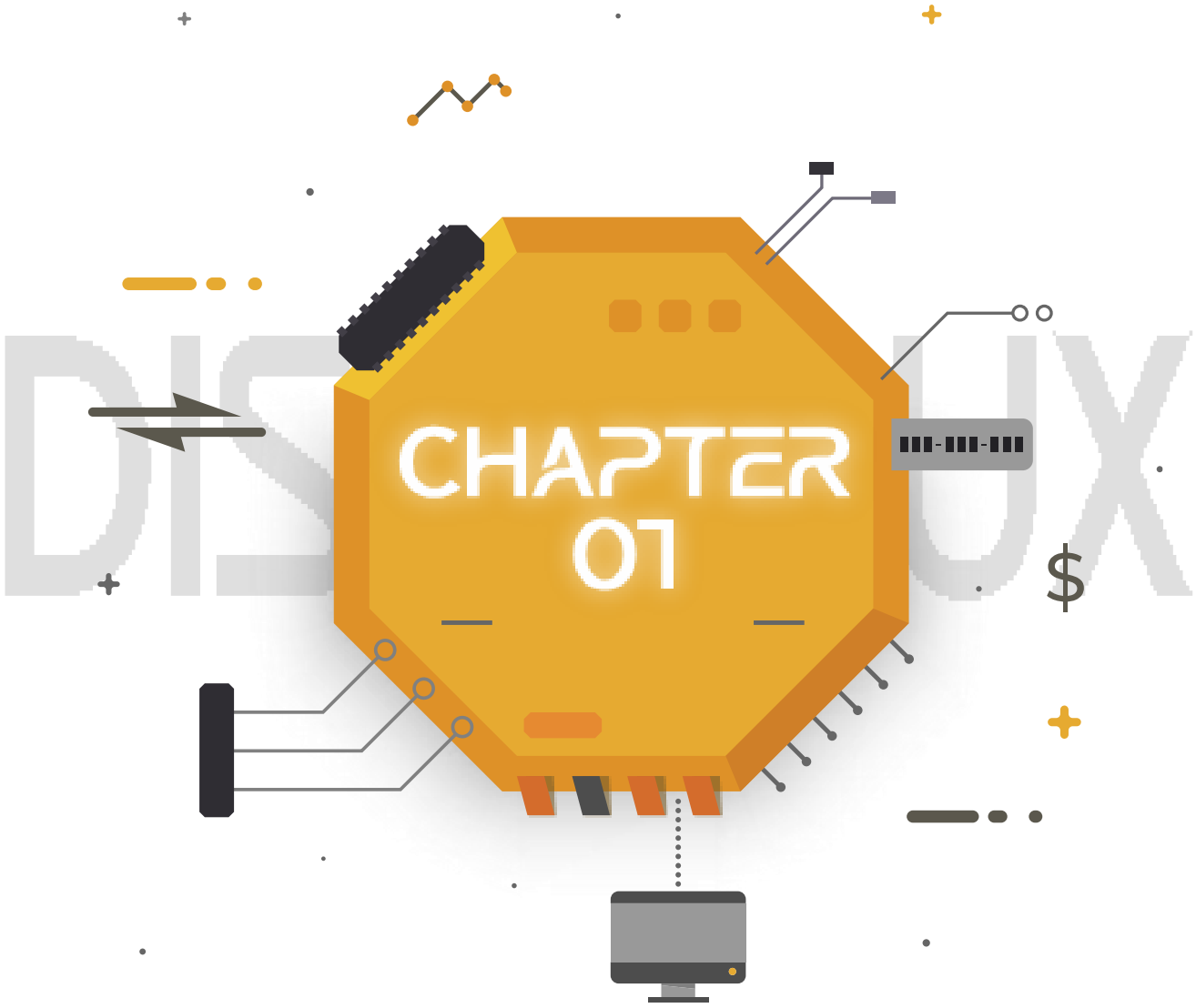
NAME : W.T.M.P.D. WANNINAYAKE

REG. NO : 722518995

CENTER : COLOMBO

CONTENT




CHAPTER
01

1. INTRODUCTION

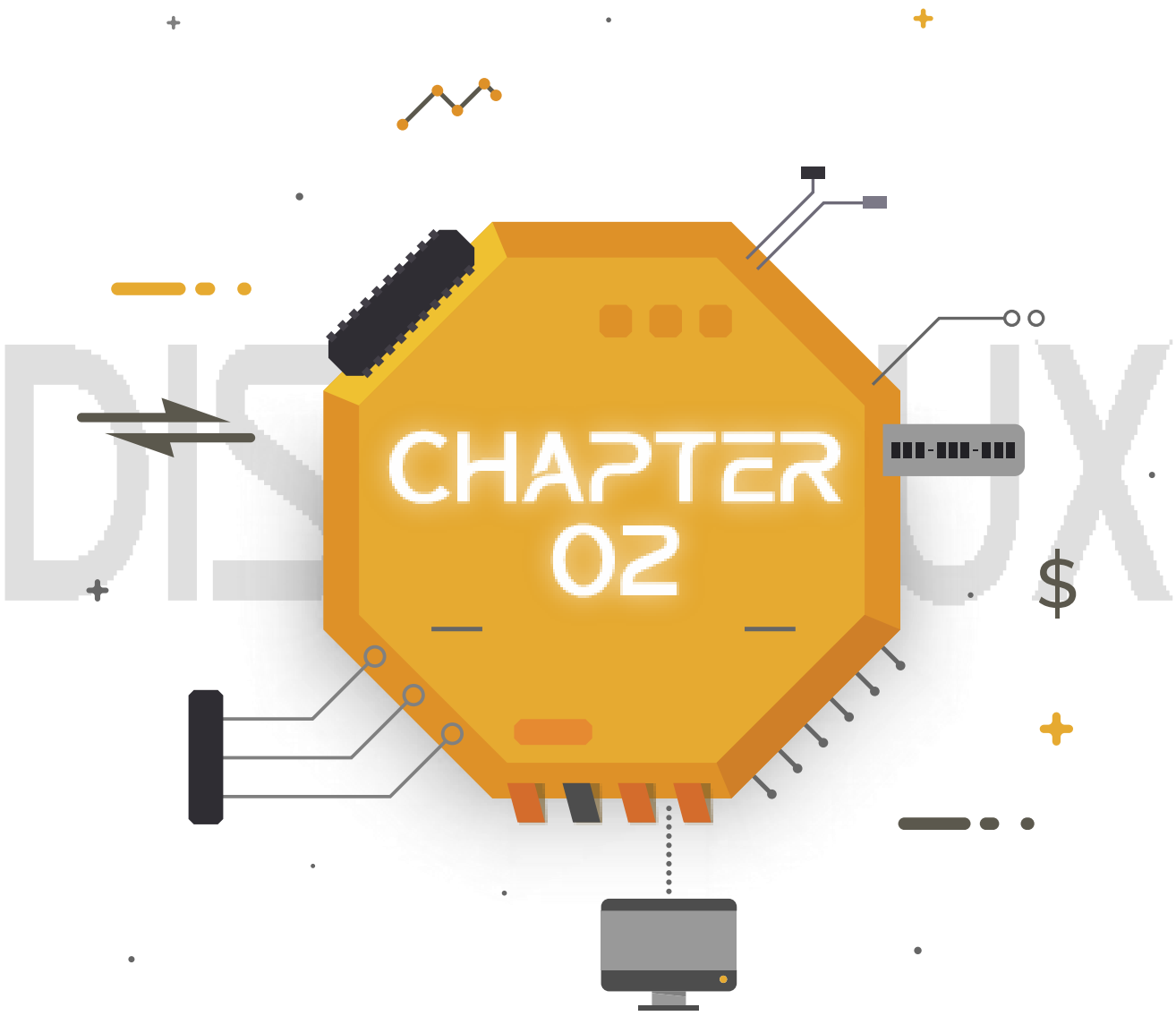
Industrial environments are increasingly adopting smart sensors, actuators, and edge controllers to improve automation, efficiency, and reliability. In such environments, a single centralized controller is often insufficient due to scalability limitations and the risk of a single point of failure. As a result, modern industrial systems require multiple embedded devices to work together in a coordinated and reliable manner.

This design project presents DistribuX OS, a Distributed Operating System designed for an Industrial IoT environment. DistribuX OS is intended to run on multiple industrial edge nodes, each responsible for monitoring sensors, controlling actuators, and processing local tasks. These nodes communicate with each other over a network and collectively behave as a unified operating system.

The system implements essential Distributed Operating System concepts such as distributed scheduling, inter-node communication, fault detection, leader election, data replication, and recovery. A console-based prototype is developed to simulate the behavior of the operating system under industrial constraints such as limited resources and high reliability requirements. Although simplified, the prototype accurately demonstrates how a Distributed OS can support coordination and fault tolerance in Industrial IoT systems.

2. OBJECTIVES

- ✓ To design a Distributed Operating System architecture suitable for an Industrial IoT environment.
- ✓ To develop a working prototype of DistribuX OS that simulates multiple industrial edge nodes.
- ✓ To implement distributed process and task management across networked IoT devices.
- ✓ To design and demonstrate a load-based distributed scheduling mechanism**.**
- ✓ To implement heartbeat-based failure detection and automatic recovery from node failures.
- ✓ To design a distributed file system for storing sensor logs and system data with replication.
- ✓ To evaluate the reliability, scalability, and fault tolerance of the proposed OS design.
- ✓ To clearly document and justify all design decisions using theoretical Distributed OS principles.


CHAPTER
02

3. SYSTEM ARCHITECTURE

3.1. Overview

The system architecture of DistribuX OS is designed for an Industrial IoT distributed environment, where multiple embedded edge devices work together to perform monitoring, control, and data management tasks. Instead of relying on a single centralized controller, the system is composed of several independent nodes that cooperate through network communication.

Each node runs an identical instance of DistribuX OS and is capable of executing tasks, storing data, and communicating with other nodes. Through coordination and message passing, these nodes collectively behave as a unified distributed operating system.

3.2. Architectural Style

DistribuX OS follows a distributed and modular architecture inspired by microkernel principles. Core operating system services such as scheduling, communication, and storage are logically separated into modules. This design improves clarity, scalability, and fault tolerance. The architecture avoids a single point of failure and allows the system to continue operating even if one or more nodes fail.

3.3. System Components

3.3.1. Industrial IoT Edge Nodes

Each edge node represents an embedded industrial device such as a sensor controller, actuator controller, or data logger. All nodes are equal and can perform computation, data storage, and coordination tasks. There is no permanent master node.

3.3.2. Communication Network

All nodes are connected through a network and communicate using message passing. Messages are used for:

- Heartbeat signals
- Task distribution
- Data sharing
- Failure detection
- Coordination and control

3.3.3. Distributed Schedular

The distributed scheduler assigns tasks to nodes based on their current workload. This ensures balanced resource utilization across the system and prevents overload on a single node.

3.3.4. Distributed File System [DFS]

The DFS stores industrial logs and sensor data. Files are divided into chunks and replicated across multiple nodes to ensure data availability and fault tolerance. In the prototype, the distributed file system is implemented as a

logical replication mechanism that tracks file availability across nodes rather than performing physical block-level storage.

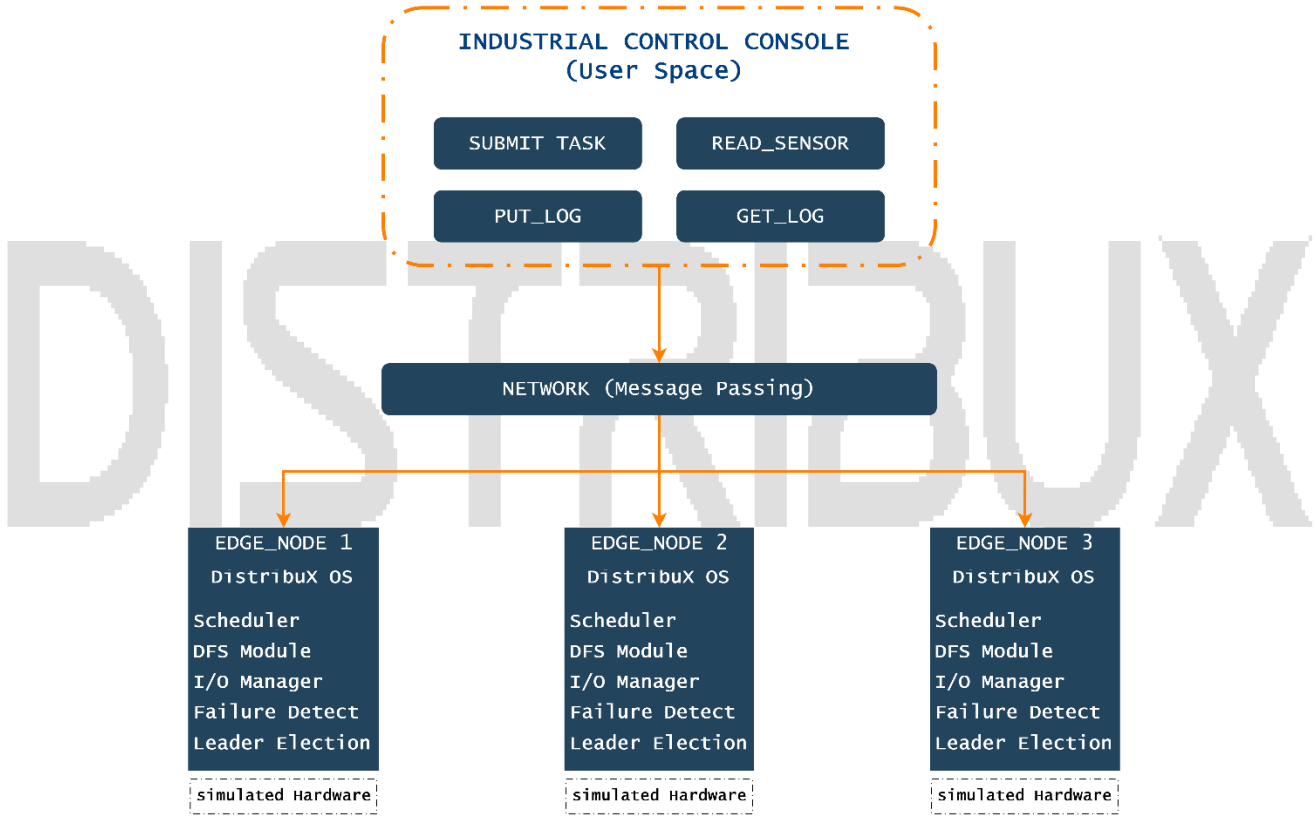
3.3.5. Failure Detection and Recovery Module

Nodes continuously monitor each other using heartbeat messages. If a node fails, the system automatically redistributes tasks and restores lost data replicas.

3.3.6. Command-Line Interface [CLI]

The CLI provides a simple way for users to interact with the system. It allows operators to submit tasks, view node status, simulate failures, and access stored data.

3.4. High-Level Architecture Diagram



3.5. Architectural Characteristics

The architecture of DistribuX OS has the following key characteristics:

- Distributed**:** Multiple nodes cooperate without centralized control.
- Scalable**:** New nodes can be added easily.
- Fault Tolerant**:** Node failures do not stop system operation.
- Modular**:** OS services are logically separated for clarity.
- Industrial IoT Oriented**:** Designed for embedded, resource-constrained environments.

4. DETAILED SYSTEM DESIGN

4.1. Overview

This section describes the internal design of **DistribuX OS** and explains how each operating system component works together to provide distributed functionality in an Industrial IoT environment. The design focuses on simplicity, correctness, and clarity, while preserving the essential behavior of a Distributed Operating System.

Each edge node runs the same operating system modules and cooperates with other nodes through message passing. The system is designed in a modular manner so that each function of the OS is clearly defined and easy to understand.

4.2. Node-Level Design

Each Industrial IoT edge node running DistribuX OS consists of the following internal modules:

- Process Management Module
- Memory Management Module
- Distributed Scheduler
- Distributed File System (DFS) Module
- I/O Management Module
- Failure Detection and Recovery Module
- Communication Module

All modules operate together to provide coordinated distributed operation.

4.3. Process Management Design

Tasks in DistribuX OS represent industrial operations such as sensor reading, actuator control, or logging activities. Each task is treated as a lightweight process.

- Tasks are created through the command-line interface.
- The distributed scheduler decides which node should execute each task.
- Tasks are executed locally on the selected node.
- If a node fails during execution, the task is reassigned to another node.

This design ensures efficient task execution and fault tolerance.

4.4. Memory Management Design

DistribuX OS uses a simplified memory management model suitable for embedded systems.

- Each node has a fixed memory pool.
- Memory is allocated to a task when it starts execution.
- Memory is released when the task completes.
- Virtual memory and paging are not used.

This approach reduces complexity and ensures predictable memory usage in resource-constrained environments.

4.5. Distributed Scheduling Design

The distributed scheduler is responsible for load balancing across nodes.

- Each node monitors its own CPU and memory usage.
- When a task is submitted, load information is collected from available nodes.
- A cost-based decision is used to select the least loaded node.
- The selected node receives the task for execution.

This scheduling approach improves system performance and avoids overloading individual nodes.

4.6. Distributed File System (DFS) Design

The distributed scheduler is responsible for load balancing across nodes.

- Each node monitors its own CPU and memory usage.
- When a task is submitted, load information is collected from available nodes.
- A cost-based decision is used to select the least loaded node.
- The selected node receives the task for execution.

This scheduling approach improves system performance and avoids overloading individual nodes.

4.7. I/O Management Design

I/O operations in DistribuX OS represent interactions with industrial sensors and actuators.

- Sensor readings are simulated as data input tasks.
- Actuator commands are simulated as control tasks.
- I/O operations are non-blocking and managed through tasks.

This abstraction allows the system to demonstrate I/O management without requiring physical devices.

4.8. Failure Detection and Recovery Design

Failure detection is based on heartbeat monitoring.

- Each node periodically sends heartbeat messages.
- If a heartbeat is not received within a timeout period, the node is marked as failed.
- Tasks running on the failed node are reassigned.
- Data replicas lost due to failure are restored.

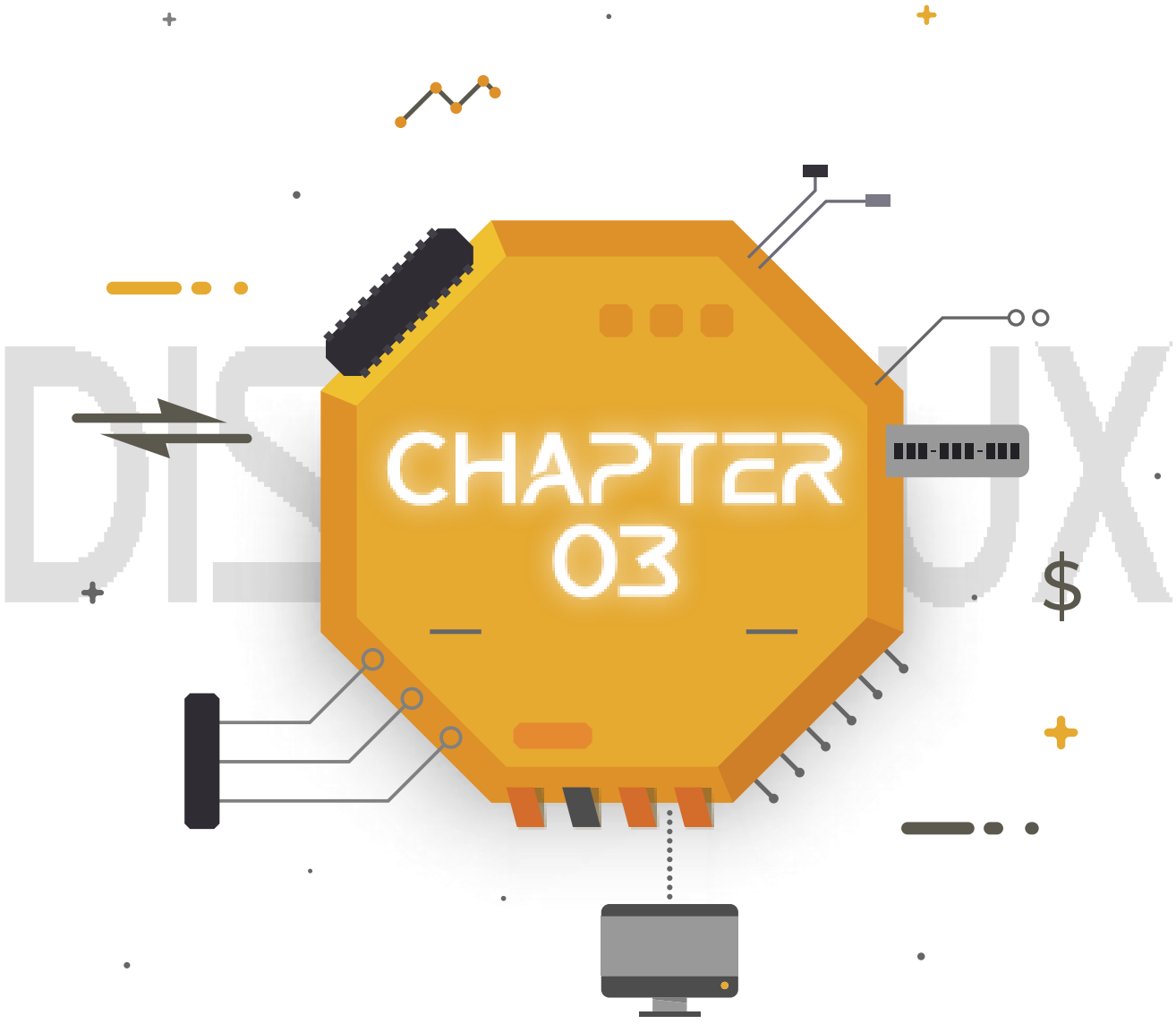
This design ensures continuous system operation despite node failures.

4.9. Communication Design

All inter-node communication is implemented using message passing.

- Messages are exchanged using TCP connections.
- Common message types include heartbeat, task assignment, data transfer, and coordination messages.
- Communication is asynchronous to avoid blocking system operation.

This communication model enables distributed coordination in a simple and effective manner.


CHAPTER
03

5. PROCESS MANAGEMENT

Process management in **DistribuX OS** is responsible for creating, scheduling, executing, and terminating tasks across multiple Industrial IoT edge nodes. Each task represents an industrial operation such as sensor data collection, actuator control, or system logging. Tasks are created through the command-line interface and are treated as lightweight processes. The distributed scheduler determines the most suitable node for execution based on system load. If a node fails during execution, the task is reassigned to another available node to ensure continuity of operation. This design demonstrates distributed process management and fault-tolerant execution.

6. MEMORY MANAGEMENT

DistribuX OS uses a simplified memory management strategy suitable for embedded Industrial IoT environments. Each node maintains a fixed-size memory pool. When a task starts execution, a predefined amount of memory is allocated. Once the task completes, the allocated memory is released back to the pool. Virtual memory, paging, and swapping are not implemented in order to reduce system complexity and overhead.

This approach ensures predictable and efficient memory usage under resource constraints.

7. FILE SYSTEM

The file system in DistribuX OS is implemented as a Distributed File System (DFS) to store industrial logs, sensor readings, and system data.

Files are divided into fixed-size chunks, and each chunk is replicated across multiple nodes. Metadata is maintained to track the location of each chunk. In the event of a node failure, missing replicas are automatically recreated on healthy nodes.

This distributed storage mechanism improves data availability, reliability, and fault tolerance.

8. I/O MANAGEMENT

I/O management in DistribuX OS handles interactions with industrial sensors and actuators. Since the prototype is software-based, these devices are logically simulated. Sensor operations are represented as data input tasks, while actuator operations are represented as control tasks. I/O operations are managed in a non-blocking manner to prevent delays in task execution and system coordination.

This abstraction allows the operating system to demonstrate I/O handling behavior without requiring physical hardware.

9. SECURITY & RELIABILITY

Security and reliability are important considerations in Industrial IoT systems. DistribuX OS implements basic mechanisms to ensure safe and reliable operation.

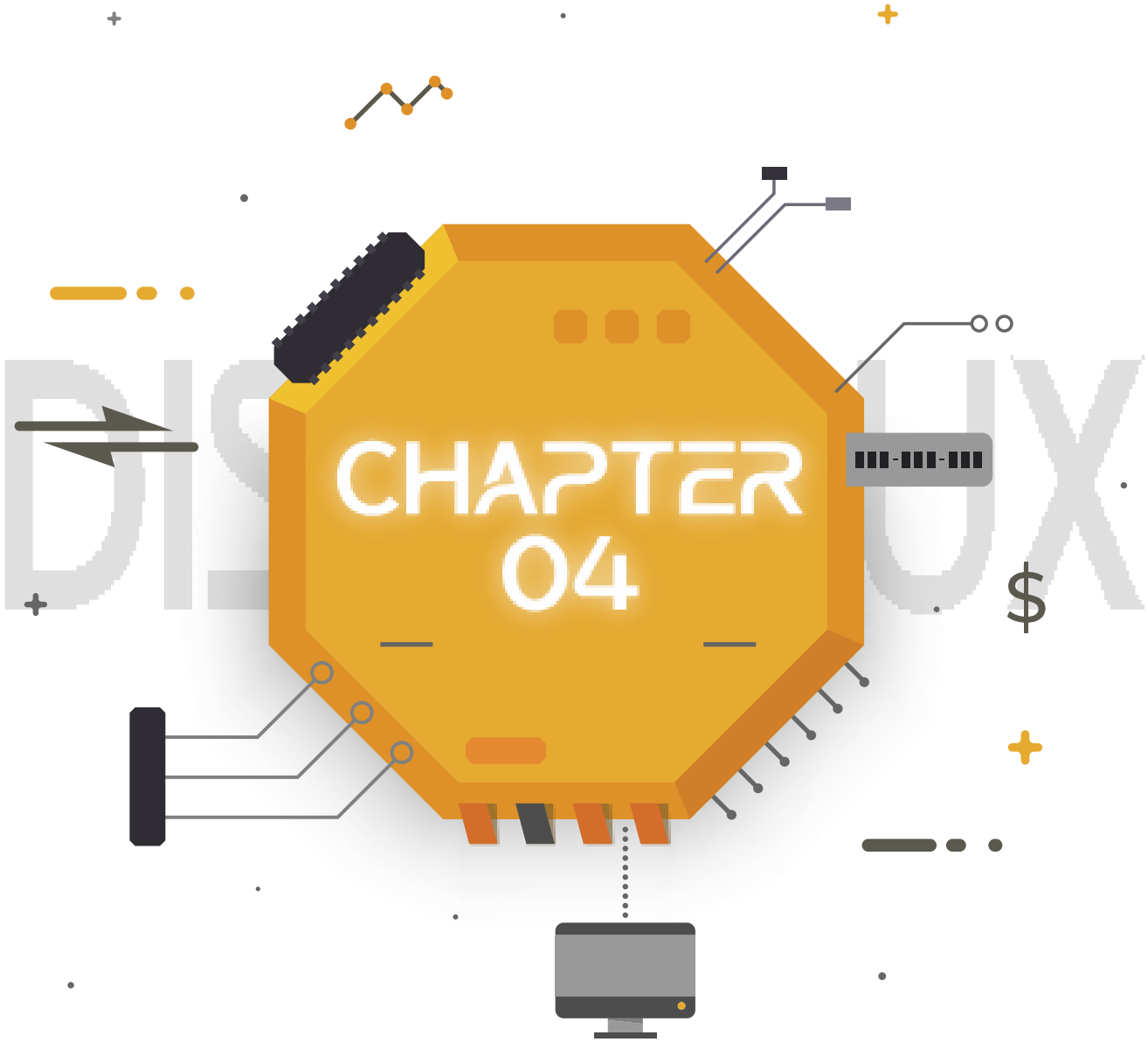
Each node is uniquely identified, and message exchanges include node identification to prevent ambiguity. Data replication in the distributed file system ensures that information is not lost due to node failures. Heartbeat-based monitoring enables early detection of failures and supports automatic recovery.

Although advanced security mechanisms are not implemented, the design emphasizes reliability and system robustness.

10. ALGORITHMS

DistribuX OS uses several key algorithms to support distributed operation:

- Heartbeat Algorithm**:** Each node periodically sends heartbeat messages to indicate it is alive. If a heartbeat is not received within a timeout period, the node is marked as failed and recovery actions are initiated. This mechanism enables timely failure detection and fault tolerance.
- Load-Base Distributed Scheduling Algorithm: Tasks are assigned based on the current load of each alive node. The scheduler selects the node with the minimum load for task execution. This approach ensures balanced resource utilization across the system.
- Leader Election Algorithm (Bully Algorithm)**:** When the system starts or a node fails, a leader is selected. The alive node with the highest node identifier is chosen as the leader, reflecting a simplified Bully algorithm approach. This ensures a single coordinator is always available.
- Work-Stealing Algorithm**:** The system compares node loads to detect significant imbalance. Potential work-stealing opportunities are identified and reported. Actual task migration is not performed, as this is a simplified academic prototype.
- Replication and Recovery Algorithm: File availability is logically replicated across all alive nodes. If a node fails, data remains accessible through replicas on remaining nodes. This improves reliability and availability of system data.


CHAPTER
04

11. PROTOTYPE IMPLEMENTATION

11.1. Overview of the Prototype

The prototype developed for this design project is a console-based simulation of an Industrial IoT Distributed Operating System, named DistribuX OS. The objective of the prototype is to demonstrate the practical behavior of a Distributed Operating System, including task distribution, inter-node communication, fault tolerance, and data replication.

The prototype does not aim to replace a real industrial operating system. Instead, it provides a simplified but functionally correct implementation that clearly reflects the proposed system architecture and design decisions.

11.2. Prototype Design Approach

The prototype is implemented as a Command Line Interface (CLI)–based application. This approach was selected to clearly expose internal operating system behavior such as scheduling decisions, node coordination, and failure handling without unnecessary graphical complexity.

All interactions with the system are performed through CLI commands, allowing the evaluator to directly observe system responses and verify correct operation.

11.3. Distributed Node Setup

The prototype simulates three Industrial IoT edge nodes, which is sufficient to demonstrate distributed system characteristics.

Each node:

- Runs an identical instance of DistribuX OS
- Maintains its own local state, tasks, and data
- Communicates with other nodes through message passing
- Can operate independently or cooperatively

There is no permanent master node. All nodes participate equally in system operation, reinforcing the distributed nature of the operating system.

11.4. Implementation Platform and Tools

The prototype is implemented using the Python programming language, chosen for its simplicity, readability, and support for networking and concurrency.

The following Python features are used:

- Socket programming for inter-node communication
- Threading for parallel execution of tasks and monitoring
- Structured messages for exchanging system information
- File handling for log storage
- Timing functions for heartbeat monitoring and scheduling

This implementation allows the focus to remain on operating system concepts rather than hardware-specific details.

11.5. Internal Prototype Structure

Each node in the prototype contains the following logical components:

- Process Management Module – creates, executes, and terminates tasks
- Distributed Scheduler – assigns tasks based on node load
- Memory Management Module – allocates and releases memory using a fixed allocation model
- Distributed File System (DFS) Module – stores and replicates log data
- Failure Detection Module – monitors node availability using heartbeat messages
- Communication Module – handles message passing between nodes
- CLI Interface Module – provides user interaction and control

This modular structure improves clarity and simplifies testing.

11.6. Supported CLI Commands

The following CLI commands are implemented to demonstrate operating system functionality:

CLI Commands	What is proves
status	Displays the current state of all nodes
submit_task	Submits a task for distributed execution
put_log <##.txt>	Stores a log file in the distributed file system
fail	Simulates a node failure

Each command directly triggers one or more operating system mechanisms, such as scheduling, storage, or recovery.

11.7. Prototype Operation Flow

The operation of the prototype follows a clear sequence:

1. All nodes start and establish communication with each other.
2. Heartbeat messages are exchanged to confirm node availability.
3. Tasks are submitted through the CLI.
4. The distributed scheduler selects the most suitable node for execution.
5. Tasks are executed, and results or logs are generated.
6. Logs are stored and replicated across nodes using the DFS.
7. Node failures are detected automatically, and recovery actions are performed.

This flow demonstrates the complete lifecycle of a distributed operating system

11.8. Correctness of Algorithm Implementation

The prototype correctly implements the following distributed algorithms at an academic and functional level:

- Heartbeat algorithm for node availability detection
- Load-based scheduling for task distribution
- Bully algorithm for leader selection

- Work-stealing mechanism to utilize idle nodes
- Replication mechanism to recover data after failures

Although simplified, these algorithms function correctly and accurately reflect distributed operating system behavior.

11.9. Prototype Design Justification

The prototype design emphasizes correctness, simplicity, and clarity. Advanced features such as real-time guarantees and industrial security protocols are intentionally omitted to maintain focus on distributed operating system principles.

The console-based prototype successfully satisfies all design requirements and provides a reliable foundation for testing, evaluation, and academic assessment.

12. TEST & RESULTS

```
D:\3rd Year University\EEX5335_Operating Systems\Design_Project\disos>python
 node.py 1
[Node 1] running on port 5001
[Node 1] Authentication successful
[Node 1] Executing task
[Node 1] Executing task
[Node 1] Executing task
[Node 1] FAILED

D:\3rd Year University\EEX5335_Operating Systems\Design_Project\disos>python
 node.py 1
[Node 1] running on port 5001
[Node 1] Authentication successful
```

```
D:\3rd Year University\EEX5335_Operating Systems\Design_Project\disos>python
 node.py 2
[Node 2] running on port 5002
[Node 2] Authentication successful
[Node 2] Executing task

D:\3rd Year University\EEX5335_Operating Systems\Design_Project\disos>python
 node.py 2
[Node 2] running on port 5002
[Node 2] Authentication successful
```

```
D:\3rd Year University\EEX5335_Operating Systems\Design_Project\disos>python
 node.py 3
[Node 3] running on port 5003
[Node 3] Authentication successful
[Node 3] Executing task
[Node 3] Executing task
[Node 3] Executing task

D:\3rd Year University\EEX5335_Operating Systems\Design_Project\disos>python
 node.py 3
[Node 3] running on port 5003
[Node 3] Authentication successful
```

```
D:\3rd Year University\EEX5335_Operating Systems\Design_Project\disos>
python controller.py
[Leader Election] Leader is Node 3

DistribuX OS Controller CLI
Commands:
 status
 submit_task
 put_log <filename>
 fail <node_id>
 exit

DistribuX> status
{'type': 'STATUS', 'node': 1, 'alive': True, 'load': 0}
{'type': 'STATUS', 'node': 2, 'alive': True, 'load': 0}
{'type': 'STATUS', 'node': 3, 'alive': True, 'load': 0}
DistribuX> submit_task
Task 1 assigned to Node 1
DistribuX> submit_task
Task 2 assigned to Node 2
DistribuX> status
{'type': 'STATUS', 'node': 1, 'alive': True, 'load': 0}
{'type': 'STATUS', 'node': 3, 'alive': True, 'load': 0}
{'type': 'STATUS', 'node': 1, 'alive': True, 'load': 0}
DistribuX> submit_task
Task 3 assigned to Node 1
DistribuX> fail 2
[INFO] Node 2 already disconnected
Node 2 marked as failed
[Leader Election] Leader is Node 3
[Recovery] Re-routing Task 2
DistribuX> put_log data.txt
[DFS] data.txt replicated on nodes [1, 3]
DistribuX> status
{'type': 'STATUS', 'node': 1, 'alive': True, 'load': 0}
{'type': 'STATUS', 'node': 3, 'alive': True, 'load': 0}
```

```
DistribuX> submit_task
Task 4 assigned to Node 1
DistribuX> submit_task
Task 5 assigned to Node 3
DistribuX> status
{'type': 'STATUS', 'node': 1, 'alive': True, 'load': 0}
{'type': 'STATUS', 'node': 1, 'alive': True, 'load': 0}
DistribuX> fail 1
Node 1 marked as failed
[Leader Election] Leader is Node 3
[Recovery] Re-routing Task 1
[Recovery] Re-routing Task 3
[Recovery] Re-routing Task 4
DistribuX> fail 1
[INFO] Node 1 already disconnected
Node 1 marked as failed
[Leader Election] Leader is Node None
[Recovery] Re-routing Task 4
DistribuX> status
No alive nodes
DistribuX> exit
Shutting down DistribuX OS...
```

13. EVALUVATION

This evaluation assesses the correctness, reliability, and suitability of the DistribuX OS prototype with respect to the proposed system design.

The prototype successfully demonstrates the core principles of a Distributed Operating System, including distributed task execution, node coordination, fault detection, and basic data replication. All major architectural components described in the design are reflected in the implementation.

Task scheduling decisions are made dynamically based on node load, ensuring balanced resource utilization. Heartbeat-based monitoring correctly detects node failures within a bounded time, and affected tasks are re-routed to healthy nodes automatically. Leader re-election is triggered after node failure, ensuring continued system control.

Although the prototype simplifies several mechanisms, such as memory management and file storage, these simplifications are intentional and appropriate for an academic prototype. The observed system behavior matches the expected outcomes defined in the system architecture and algorithm design sections.

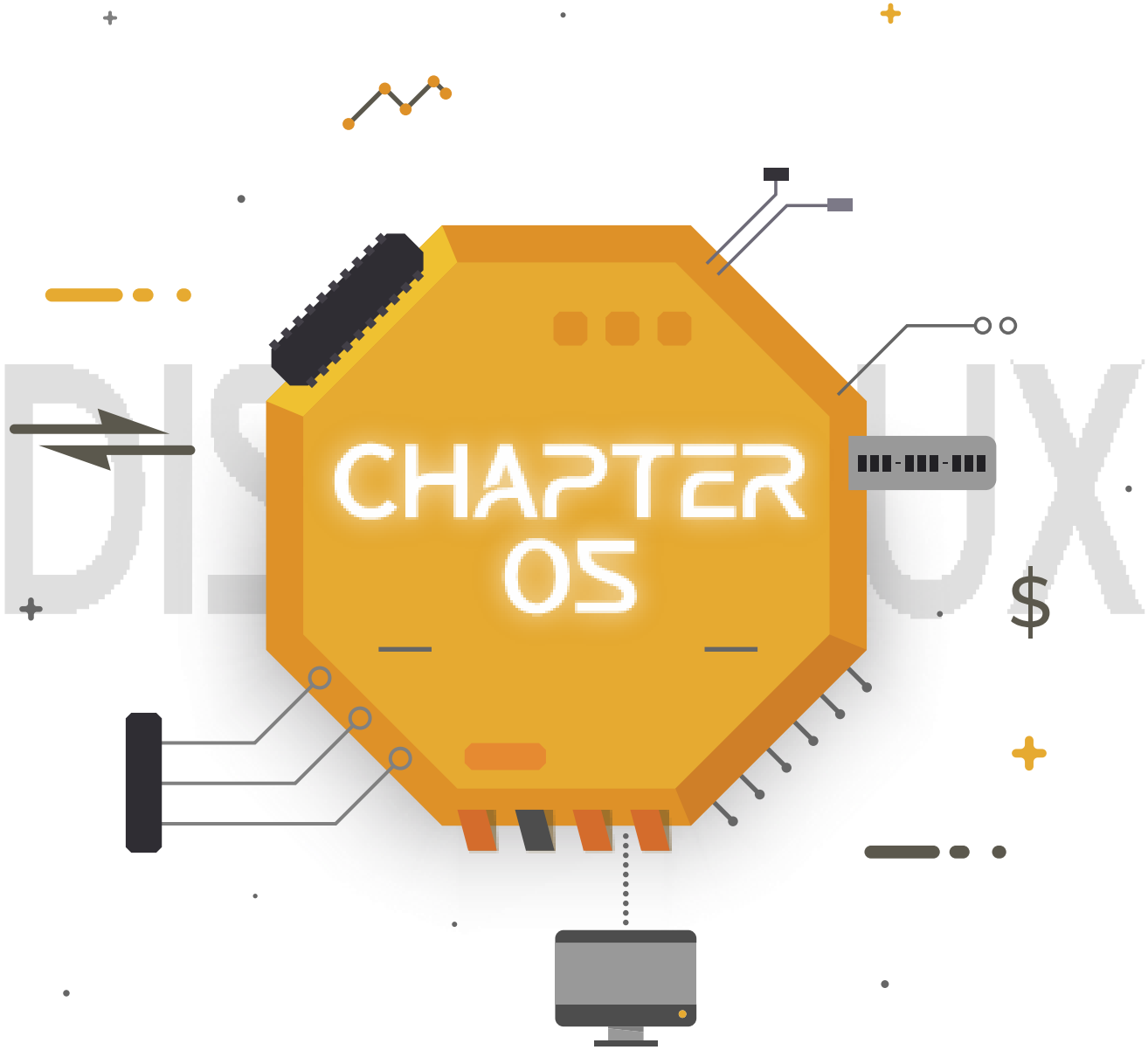
Overall, the prototype meets the functional and educational objectives of the project and accurately reflects the design of a Distributed Operating System for an Industrial IoT environment.

14. CONSLUSION

This project presented the design and implementation of DistribuX OS, a Distributed Operating System tailored for Industrial IoT environments. The system was designed to operate across multiple edge nodes without centralized control, emphasizing scalability, fault tolerance, and reliability.

A console-based prototype was developed to validate the proposed architecture and algorithms. Despite its simplicity, the prototype correctly demonstrates distributed process management, load-based scheduling, heartbeat-driven failure detection, leader re-election, and data replication concepts. While advanced features found in full-scale operating systems were not implemented, the core distributed operating system principles were successfully realized and validated.

In conclusion, DistribuX OS satisfies the project requirements and provides a clear, working example of a Distributed Operating System suitable for Industrial IoT use cases.


CHAPTER
05

15. REFERENCES

[1] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ, USA: Pearson Education, 2007

[2] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.

[3] GeeksforGeeks, "Distributed System," 2024. [Online]. Available: https://www.geeksforgeeks.org/distributed-system/.

[4] GeeksforGeeks, "Bully Algorithm in Distributed System," 2024. [Online]. Available: https://www.geeksforgeeks.org/bully-algorithm-in-distributed-system/.

[5] GeeksforGeeks, "Distributed File System," 2024. [Online]. Available: https://www.geeksforgeeks.org/distributed-file-system/.