Foundations of Cyber-Physical Systems: Theory and Practice with Node-RED

Komsan Kanjanasit
College of Computing
Prince of Songkla University

Preface

The growing convergence of computation, communication, and control with physical processes has given rise to **Cyber-Physical Systems (CPS)**. These systems now form the backbone of modern infrastructure and innovation, from autonomous vehicles and smart grids to intelligent healthcare devices and Industry 4.0 factories. Their impact is profound: CPS not only integrate hardware and software, but also transform how humans interact with the physical and digital worlds.

This book, *Introduction to Cyber-Physical Systems (CPS)*, is designed as both a foundational textbook and a hands-on guide for undergraduate and graduate students, engineers, and researchers. Its central goal is to provide a comprehensive introduction to CPS principles while emphasizing practical prototyping and experimentation.

Scope of the Book

The text is organized into twelve chapters, beginning with foundational concepts and architectures, progressing through sensing, actuation, networking, and control, and culminating in large-scale infrastructure systems, security, and case studies:

- **Chapters 1–2:** Introduce CPS foundations, architectures, and modeling approaches.
- **Chapters 3–6:** Cover sensing, actuation, communication, and discrete control algorithms.
- **Chapters 7–8:** Explore CPS prototyping systems, hardware/software co-design, and middleware.
- **Chapters 9–10:** Discuss infrastructure-scale CPS design, integration, and testing.
- **Chapter 11:** Addresses CPS security and privacy, emphasizing attack surfaces and defenses.
- **Chapter 12:** Concludes with case studies and future trends, including digital twins, edge AI, and sustainability.

Hands-On Node-RED Labs

A distinctive feature of this book is its emphasis on experiential learning. Each chapter is paired with a practical laboratory exercise implemented in **Node-RED**, an open-source, flow-based programming tool widely adopted for IoT and CPS applications. These labs:

- Reinforce theoretical concepts through simulation and prototyping.
- Use accessible hardware platforms such as ESP32, Arduino-compatible sensors, and actuators.
- Introduce communication protocols like MQTT and OPC UA.
- Encourage experimentation with dashboards, control logic, and security features.

By the end of the course, readers will have developed their own capstone CPS prototype, integrating the full pipeline from sensing to secure actuation.

Pedagogical Approach

The book combines:

- **Conceptual clarity:** Each chapter introduces key definitions, models, and frameworks.
- **Practical design:** Node-RED labs allow immediate application of theory.
- **Systemic perspective:** Case studies illustrate how CPS principles are applied in healthcare, transportation, manufacturing, and smart cities.
- **Forward-looking vision:** Future trends prepare students for emerging technologies in CPS.

Acknowledgments

This work is inspired by the rapid evolution of CPS research and practice across academia and industry. The integration of ESP32 platforms and Node-RED environments reflects the contributions of the open-source community, whose tools make CPS experimentation widely accessible. I am grateful to students and colleagues whose feedback shaped the content and structure of this book.

Intended Audience

The book is aimed at:

- Undergraduate and graduate students studying computer engineering, electrical engineering, computer science, or related fields.
- Practitioners seeking a structured introduction to CPS design and prototyping.
- Educators developing CPS courses with integrated theory and hands-on laboratories.

Closing Note

Cyber-Physical Systems are not only reshaping industries but also redefining society itself. By studying CPS, learners prepare to design the next generation of intelligent, connected, and secure systems. This book aims to equip readers with both the conceptual foundations and the practical skills to contribute to this exciting field.

Phuket, Thailand
September, 2025
Komsan Kanjanasit

Contents

List of Figures

List of Tables

Chapter 1

Foundations of Cyber-Physical Systems (CPS)

Chapter Overview

Cyber-Physical Systems (CPS) integrate computation, communication, and control with physical processes. This chapter builds the conceptual foundation for the course, clarifies how CPS differs from embedded systems and IoT, introduces representative applications (smart cities, healthcare, energy, transportation, manufacturing), and frames the analytical lenses we will use across the textbook (discrete-event/hybrid modeling, sensing and actuation chains, real-time constraints, and safety/security).

Learning Outcomes

After completing this chapter, students will be able to:

1. Define CPS and explain the interplay among sensing, computation, control, actuation, and communication.
2. Differentiate CPS from embedded systems, IoT solutions, and digital twins.
3. Describe the CPS layered architecture (perception–network–cyber–actuation–human).
4. Identify key design constraints: sampling, latency, jitter, reliability, and safety/security.
5. Explain the role of discrete control algorithms and hybrid (continuous–discrete) models in CPS.
6. Set up a basic Node-RED environment and implement a "Hello CPS" dashboard flow.

Prerequisites

Basic programming, introductory signals and systems, and comfort with microcontrollers (e.g., Arduino/ESP32) are recommended.

Key Terms

Cyber-Physical System, sensing chain, actuator, control loop, discrete-event system (DES), hybrid system, real-time constraint, latency budget, jitter, reliability, availability, safety, security, digital twin, MQTT, Node-RED.

1.1 What Is a Cyber-Physical System?

A CPS is a feedback system where computational elements observe and influence physical processes. Sensors capture the state of the plant (physical world), computation produces decisions (estimation, control, scheduling), and actuators apply inputs back to the plant. Communication networks connect distributed components across space and organizational boundaries.

CPS vs. Embedded Systems. An embedded system is a computer integrated into a device; a CPS extends this with *tight* feedback coupling to physical dynamics, often distributed over networks, with explicit timing, safety, and resilience guarantees.

CPS vs. Internet of Things (IoT). IoT emphasizes connectivity and data exchange; CPS emphasizes *control* and *closed-loop behavior*. IoT devices participate in CPS when their data and commands are orchestrated to meet control and timing requirements.

CPS vs. Digital Twins. A digital twin is a synchronized virtual representation of a physical asset/process. Digital twins enrich CPS with state estimation, prediction, and what-if analysis but are components/techniques rather than a CPS by themselves.

1.2 CPS Application Domains

- **Smart Manufacturing:** robotic cells, quality inspection, predictive maintenance.
- **Energy Systems:** microgrids, demand response, inverter control, fault detection.
- **Transportation/ITS:** traffic signal coordination, vehicle-to-infrastructure (V2I) safety.

- **Healthcare:** patient monitoring, closed-loop insulin delivery, rehabilitation exoskeletons.

- **Smart Buildings/Cities:** HVAC optimization, occupancy-aware lighting, safety alarms.

1.3 A Layered Reference Architecture

Figure 1.1 shows a canonical layered view used throughout the course.

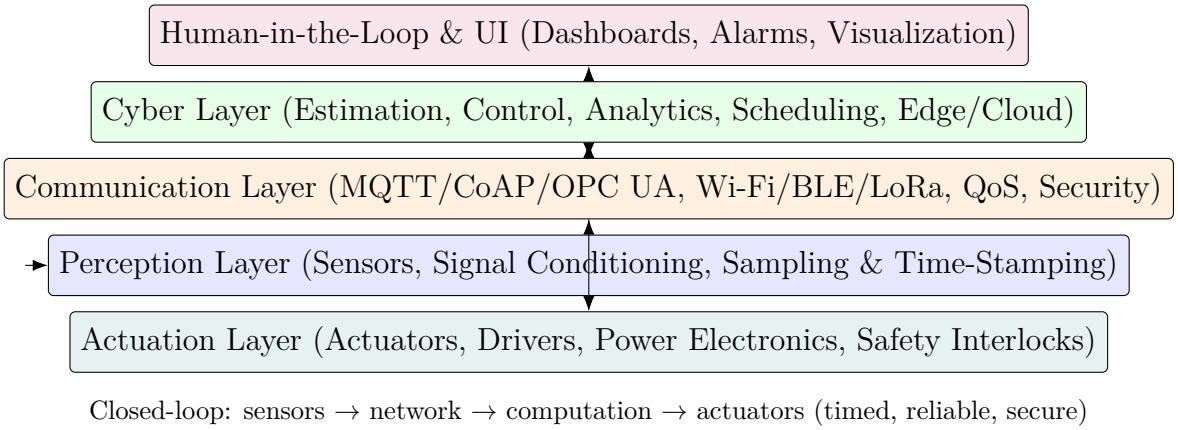

Figure 1.1: CPS layered stack used in this text.

1.4 Timing, Sampling, and Real-Time Constraints

CPS behavior depends critically on *timing*. Let T_s be the sampling period and D the end-to-end deadline from sensing to actuation. A simple latency budget is

$$D = d_{\text{sense}} + d_{\text{net}} + d_{\text{compute}} + d_{\text{act}},$$

where each term includes average and jitter components. For stability and performance:

$$T_s \leq D \qquad \text{and often} \qquad T_s \leq \frac{1}{10} T_{\text{dyn}},$$

with T_{dyn} a characteristic time constant of the physical process. Jitter variance should be kept small (e.g., via priority scheduling, QoS, or edge placement).

Reliability and Availability. Many CPS require >99.9% availability. Design patterns include redundancy (sensors, networks), watchdogs, fail-safes, and degraded modes.

1.5 Modeling Perspectives: Discrete, Continuous, and Hybrid

Discrete-Event Systems (DES) model events (arrivals, threshold crossings, mode switches). **Continuous-time** models capture physical dynamics (ODEs). **Hybrid systems** combine both (e.g., a thermostat: continuous temperature, discrete on/off control).

Example (Room Temperature Control). Continuous state $x(t)$ (temperature), discrete control $u \in \{\text{HEAT, OFF}\}$. A simple hybrid rule:

$$u(t) = \begin{cases} \text{HEAT}, & x(t) < x_{\text{set}} - \Delta, \\ \text{OFF}, & x(t) > x_{\text{set}} + \Delta. \end{cases}$$

The controller executes every T_s seconds; network and compute must meet $D \leq T_s$.

1.6 CPS Toolchain and Platforms

- **Edge hardware:** ESP32/Arduino, Raspberry Pi, industrial PLCs; optional FPGAs for hard real-time or acceleration.
- **Middleware & orchestration:** Node-RED for rapid prototyping; MQTT/OPC UA for messaging; containers for deployment.
- **Modeling/Simulation:** Modelica/Simulink for plant/control co-simulation; Python/-Matlab for analysis; digital twin dashboards.

1.7 Security and Safety at a Glance

CPS operate in the physical world; failures propagate beyond data loss. Minimum practices:

1. Mutual authentication and least-privilege credentials.
2. Transport security (TLS) and topic-level authorization (for MQTT).
3. Data validation (bounds, rate limits, plausibility checks).
4. Safety interlocks and emergency stop paths that bypass networks when needed.

1.8 Case Study: Smart Room Controller

A smart room monitors temperature (T), humidity (H), occupancy (O), and controls a fan and light. Requirements:

- Update T, H, O at 1 Hz; decision latency $D \leq 500$ ms.
- If O=0, reduce fan setpoint and turn off lights (energy saving).
- Fail-safe: if no sensor updates in 5 s, revert to safe defaults and alert.

We will develop this progressively in later chapters (sensing, rules/FSM, security, twin).

1.9 Node-RED Lab 1: Hello CPS World

Objectives

- Install Node-RED and create a basic dataflow: Inject → Function → Debug/Dashboard.
- Understand messages, topics, and deploy cycle.

Required Hardware/Software

PC with Node.js and Node-RED; a modern web browser. (No external hardware required.)

Pre-Lab Checklist

1. Install Node.js LTS. `node -v` should print a version.
2. Install Node-RED: `npm install -g node-red`.
3. Start: `node-red`. Browse to `http://localhost:1880`.
4. (Optional) Install dashboard nodes: Menu → Manage Palette → Install → `node-red-dashboard`.

Procedure

1. Drag an **inject** node (timestamp) → a **function** node (rename to *format*) → **debug** node.
2. In the *format* node, paste:

```
msg.payload = "Hello CPS: " + new Date().toISOString();
return msg;
```

3. Deploy. Click the inject button. Observe the Debug panel.
4. Add a **ui_text** node (dashboard) and connect from *format* to display the message.
5. (Optional) Replace inject with a repeating inject (every second) and observe streaming updates.

Expected Output

Debug panel and dashboard display strings like: *Hello CPS: 2025-09-10T00:00:00.000Z.*

Assessment Rubric (10 points)

2 Correct installation and launch of Node-RED.

3 Working flow (inject → function → debug).

3 Dashboard text displays formatted string.

2 Clear screenshot and brief explanation of message path.

Starter Flow (Importable JSON)


```
[{"id":"tab1","type":"tab","label":"Lab1 - Hello CPS"},
{"id":"inj1","type":"inject","z":"tab1","name":"tick","props":[{"p":"payload"}],
"repeat":"","crontab":"","once":false,"onceDelay":0.1,"topic":"",
"payloadType":"date","x":140,"y":120,"wires":[["fn1"]]},
{"id":"fn1","type":"function","z":"tab1","name":"format",
"func":"msg.payload = \"Hello CPS: \" + new Date(msg.payload).toISOString();\nreturn
"outputs":1,"noerr":0,"initialize":"","finalize":"","libs":[],
"x":330,"y":120,"wires":[["dbg1","ui1"]]},
{"id":"dbg1","type":"debug","z":"tab1","name":"debug","active":true,"tosidebar":true,
"console":false,"tostatus":false,"complete":"payload","statusVal":"",
"statusType":"auto","x":520,"y":90,"wires":[]},
{"id":"ui1","type":"ui_text","z":"tab1","group":"uigrp1","order":1,"width":0,"height"
"name":"","label":"Hello","format":"{{msg.payload}}","layout":"row-spread",
"x":520,"y":150,"wires":[]},
{"id":"uigrp1","type":"ui_group","name":"Lab1","tab":"uitab1","order":1,"disp":true,
"width":"6","collapse":false},
{"id":"uitab1","type":"ui_tab","name":"CPS Labs","icon":"dashboard",
```

"disabled":false,"hidden":false}]

1.10 Design Constraints and Trade-offs

CPS design balances competing goals:

Performance: rise time, tracking error, throughput.

Resource limits: CPU, memory, bandwidth, energy.

Timing: latency and jitter vs. sampling and control deadlines.

Safety/Security: enforcement of invariants, authentication, encryption, fail-safe modes.

Common trade-offs include pushing analytics to the edge to reduce latency/bandwidth, while keeping heavy models in the cloud for scalability.

1.11 From Prototype to Infrastructure-Scale CPS

Scaling from a single device to a campus/city system introduces:

- Heterogeneity (multi-vendor devices, protocols).
- Orchestration (naming, discovery, configuration).
- Observability (telemetry, logging, traces).
- Governance (policies, auditing, compliance).

Later chapters address these topics (security, digital twins, standards, and case studies).

Summary

CPS merge computation with the physics of the real world. Effective CPS engineering demands attention to timing, reliability, and safety/security, in addition to functionality. Node-RED provides a rapid-prototyping environment to explore CPS flows before integrating hardware and meeting stricter real-time constraints.

Review Questions

1. Define CPS and list the core subsystems involved in the CPS loop.
2. Explain how CPS differs from embedded systems and IoT in terms of goals and constraints.

3. Describe a latency budget and why jitter matters in closed-loop control.
4. Provide an example of a hybrid system from daily life and sketch its discrete logic.
5. What security measures are essential when deploying CPS in public infrastructure?

Exercises

1. (Modeling) For a temperature control loop with T_s=1 s, propose maximum permissible d_{net} and d_{compute} if d_{sense}=50 ms and d_{act}=50 ms, assuming D=T_s and 20% margin.
2. (Design) Draft a topic hierarchy for a smart-room CPS using MQTT (sensors, commands, events, alarms).
3. (Prototype) Extend Lab 1.9 to include a periodic inject (1 Hz) and a dashboard chart; discuss observed timing.

Further Reading

- E. A. Lee, "Cyber Physical Systems: Design Challenges," 2008.
- R. Rajkumar et al., "Cyber-Physical Systems: The Next Computing Revolution," 2010.
- P. Tabuada, *Verification and Control of Hybrid Systems*, 2009.
- OPC Foundation, *OPC UA Specifications*, latest edition.

Chapter 2

CPS Architectures and Models

Chapter Overview

This chapter explores the architectural frameworks and system models that underpin Cyber-Physical Systems (CPS). Building on the conceptual foundation from Chapter 1, we examine reference architectures (5C, IIRA, RAMI 4.0), layered abstractions, discrete-event and hybrid models, and co-design principles that align hardware, software, and communication. Practical work introduces modeling CPS data flow in Node-RED without physical hardware, emphasizing data injection, transformation, and visualization.

Learning Outcomes

After completing this chapter, students will be able to:

1. Describe layered CPS architectures and their functional decomposition.
2. Explain the 5C architecture (Connection, Conversion, Cyber, Cognition, Configuration).
3. Contrast IIRA (Industrial Internet Reference Architecture) with RAMI 4.0.
4. Model CPS processes using discrete-event (DE) and hybrid system representations.
5. Build a simple data flow model in Node-RED using inject, function, and dashboard nodes.

Key Terms

5C architecture, IIRA, RAMI 4.0, discrete-event system (DES), hybrid system, layered CPS model, co-simulation, Node-RED dataflow.

2.1 Reference Architectures for CPS

2.1.1 The 5C Architecture

The 5C architecture (Lee et al., 2015) is widely used in smart manufacturing and CPS engineering:

1. **Connection:** sensors and devices collect raw data.
2. **Conversion:** raw data are processed into useful information.
3. **Cyber:** information is aggregated, analyzed, and stored in cyber infrastructure.
4. **Cognition:** decision-making and knowledge generation.
5. **Configuration:** adaptive feedback to physical systems.

This stack captures both bottom-up (data-driven) and top-down (decision-driven) interactions.

2.1.2 IIRA (Industrial Internet Reference Architecture)

IIRA, developed by the Industrial Internet Consortium, specifies three viewpoints:

- **Business:** stakeholders, values, ROI, compliance.
- **Usage:** functional requirements, scenarios, workflows.
- **Functional:** capabilities organized into control, operations, information, and business domains.

IIRA emphasizes interoperability and cross-industry adaptability.

2.1.3 RAMI 4.0 (Reference Architectural Model Industry 4.0)

RAMI 4.0 defines a three-dimensional reference cube:

Hierarchy levels: from field devices to enterprise.

Layers: asset, integration, communication, information, functional, business.

Life cycle/value stream: type (design) to instance (runtime).

RAMI 4.0 stresses life cycle integration and standardized interfaces.

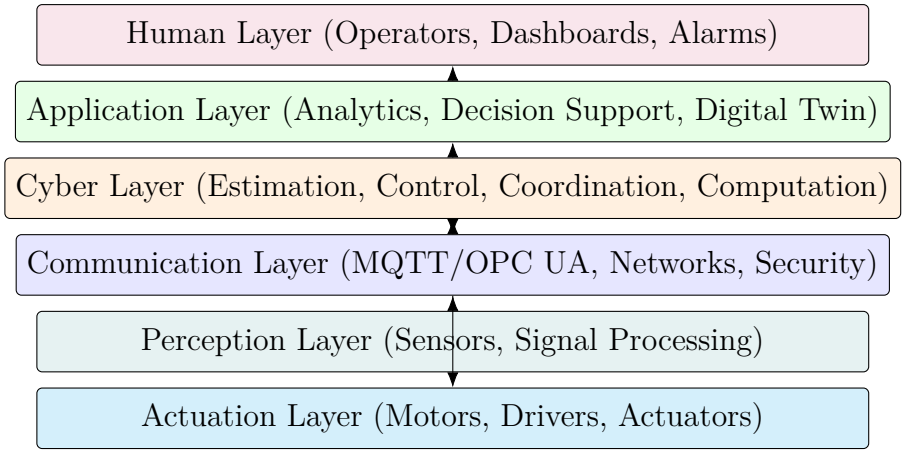

Figure 2.1: Simplified layered CPS model (perception to actuation).

2.2 Layered CPS Models

A simplified CPS layered model helps organize analysis:

2.3 System Models: DES, Continuous, and Hybrid

Discrete-Event Systems (DES). CPS often include events: packet arrivals, threshold crossings, alarms, or mode changes. DES model system state transitions triggered by discrete events.

Continuous Models. Physical plants (temperature, velocity, voltage) are modeled by differential equations.

Hybrid Models. Combine both perspectives. Example: traffic lights — continuous vehicle arrivals (flow), discrete switching of lights.

2.4 CPS Co-Design Principles

Key principles:

1. **Hardware-software co-design:** partitioning functions across microcontrollers, edge servers, and cloud.
2. **Timing-aware design:** allocate latency budgets across sensing, networking, computation, and actuation.
3. **Safety/security-first:** design invariants and fail-safe modes early.
4. **Scalable architecture:** anticipate growth in nodes and data volume.

2.5 Case Study: Traffic Light CPS

- **Sensors:** inductive loops or cameras detect cars.
- **Controllers:** compute phases based on arrival rates.
- **Actuators:** lights switch on/off.
- **Communication:** intersections share load information over network.
- **Model:** hybrid system with continuous vehicle arrivals and discrete FSM of traffic lights.

2.6 Node-RED Lab 2: Modeling CPS Data Flow

Objectives

- Model a CPS pipeline using Node-RED without physical devices.
- Use inject nodes as simulated sensors.
- Transform data using function nodes.
- Display actuator states on a dashboard.

Required Software

Node-RED (with dashboard package).

Procedure

1. Drag two **inject** nodes: label them “Sensor A” and “Sensor B”.
2. Connect both to a **function** node (Processing). Example code:

```
let v = Number(msg.payload);
msg.payload = {source: msg.topic, value: v, score: v*2};
return msg;
```

3. Connect function node to:
   - a **gauge** node (dashboard) showing score,
   - and a **debug** node to observe messages.
4. Deploy, trigger sensors, and observe dashboard.

Expected Output

When Sensor A injects value 42, the gauge displays 84. When Sensor B injects 17, the gauge shows 34. Debug panel prints JSON objects with source, value, score.

Assessment Rubric (10 points)

2 Correct creation of inject nodes with topics.

3 Function node transforms payload correctly.

3 Gauge displays processed score.

2 Debug messages consistent with inputs.

Starter Flow (Importable JSON)


```
[{"id":"tab2","type":"tab","label":"Lab2 - CPS Model"},
{"id":"injA","type":"inject","z":"tab2","name":"Sensor A","props":[{"p":"payload"},{"
"topic":"sensor/A","payload":"42","payloadType":"num","x":130,"y":100,"wires":[["fn2"
{"id":"injB","type":"inject","z":"tab2","name":"Sensor B","props":[{"p":"payload"},{"
"topic":"sensor/B","payload":"17","payloadType":"num","x":130,"y":160,"wires":[["fn2"
{"id":"fn2","type":"function","z":"tab2","name":"Processing","func":"let v = Number(m
{"id":"uiGauge","type":"ui_gauge","z":"tab2","group":"uigrp2","order":1,"gtype":"gage
{"id":"dbg2","type":"debug","z":"tab2","name":"debug","active":true,"tosidebar":true,
{"id":"uigrp2","type":"ui_group","name":"Lab2","tab":"uitab2","order":1,"disp":true,"
{"id":"uitab2","type":"ui_tab","name":"CPS Labs","icon":"dashboard","disabled":false,
```


Summary

CPS architectures organize sensing, computation, communication, and actuation into layered stacks, with reference models (5C, IIRA, RAMI 4.0) providing systematic guidance. Modeling frameworks (DES, hybrid systems) capture the interplay between discrete and continuous dynamics. Node-RED offers a rapid way to prototype CPS dataflows.

Review Questions

1. What are the five layers of the 5C architecture? Give an example for each.
2. How does IIRA's business/usage/functional viewpoint differ from RAMI 4.0's cube?
3. Distinguish between discrete-event, continuous, and hybrid CPS models.

4. Why is co-design (hardware + software) critical in CPS?

Exercises

1. Extend the Lab 2.6 flow by adding a third inject node (Sensor C). Apply a different processing rule (e.g., square the value) and display on a second gauge.
2. Sketch the RAMI 4.0 cube and map a smart factory use case onto it.
3. Model a traffic light CPS as a hybrid system with vehicle flow (continuous) and traffic lights (discrete FSM).

Further Reading

- L. Monostori, "Cyber-Physical Production Systems: Roots, Expectations and R&D Challenges," Procedia CIRP, 2014.
- Industrial Internet Consortium, *Industrial Internet Reference Architecture (IIRA)*, 2019.
- Plattform Industrie 4.0, *RAMI 4.0 Reference Model*, 2015.
- P. Derler, E.A. Lee, A. Sangiovanni-Vincentelli, "Modeling Cyber-Physical Systems," Proc. IEEE, 2012.

Chapter 3

Sensing and Data Acquisition in CPS

Chapter Overview

Cyber-Physical Systems (CPS) depend on sensors to capture the state of the physical world. Sensors convert physical phenomena (temperature, motion, pressure, light, vibration, biosignals) into measurable signals. This chapter explores sensing principles, signal conditioning, data acquisition (DAQ), sampling, noise, calibration, and distributed sensing systems. A practical lab demonstrates integrating a temperature or motion sensor with an ESP32 and visualizing data in Node-RED.

Learning Outcomes

After completing this chapter, students will be able to:

1. Classify different sensor types relevant to CPS applications.
2. Explain the sensing chain: transduction, conditioning, digitization, and communication.
3. Apply sampling theory and identify aliasing/noise issues.
4. Understand calibration and error sources in sensors.
5. Build a CPS pipeline with a physical sensor (ESP32 + LM73/MPU6050) and Node-RED dashboard.

Key Terms

Transducer, signal conditioning, ADC (Analog-to-Digital Converter), sampling rate, Nyquist theorem, quantization, noise, calibration, MQTT, Node-RED dashboard.

3.1 The Role of Sensing in CPS

Sensors are the "eyes and ears" of CPS. They capture state variables:

- **Environmental:** temperature, humidity, light, gas.
- **Mechanical:** acceleration, strain, pressure, vibration.
- **Biological:** heart rate, EEG, EMG.

Accurate sensing enables robust control and prediction. Errors propagate directly into actuation decisions.

3.2 The Sensing Chain

The sensing process involves multiple stages:

1. **Transduction:** physical stimulus → electrical signal (e.g., resistance change in thermistor).
2. **Conditioning:** amplification, filtering, linearization.
3. **Digitization:** ADC converts analog voltage to discrete numbers.
4. **Packaging/Communication:** values are sent via protocols (I^2C, SPI, UART, MQTT).

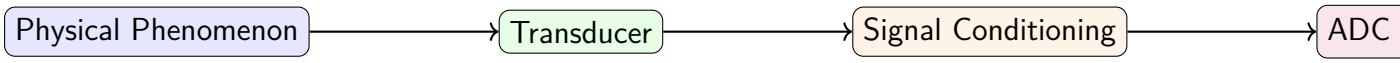

Figure 3.1: The sensing chain in CPS: from physical phenomenon to communication.

3.3 Sampling and Quantization

According to the Nyquist theorem, the sampling frequency f_s must satisfy:

$$f_s \geq 2f_{\max}$$

where $f_{\max}$ is the maximum frequency present in the signal. Otherwise, aliasing introduces distortion.

Quantization. An N-bit ADC provides 2^N levels. Quantization error is approximately $\pm\frac{1}{2}$ LSB.

Example. A 12-bit ADC with 3.3 V reference has step size:

$$\Delta = \frac{3.3}{2^{12}} \approx 0.8\,\text{mV}.$$

3.4 Noise and Calibration

- **Noise:** thermal, electromagnetic interference, quantization.
- **Filtering:** low-pass filters smooth high-frequency noise.
- **Calibration:** align sensor output with reference standards; essential for accuracy.

3.5 Distributed Sensing Systems

Modern CPS often rely on distributed sensor networks:

- Multiple sensor nodes share data via wireless protocols (Wi-Fi, BLE, LoRa).
- Synchronization ensures coherent state estimation.
- Data fusion (Kalman filter, sensor fusion algorithms) combines multiple readings.

3.6 Case Study: Inertial Sensing

The MPU6050 (accelerometer + gyroscope) illustrates distributed sensing:

- Provides acceleration (a_x, a_y, a_z) and angular velocity $(\omega_x, \omega_y, \omega_z)$.
- Used in drones, wearables, robotics.
- Often combined with magnetometer for full orientation (9-DOF).

3.7 Node-RED Lab 3: Sensor Integration with ESP32

Objectives

- Connect ESP32 to a physical sensor (LM73 or MPU6050).
- Publish sensor data via MQTT.
- Visualize data on Node-RED dashboard (gauge + chart).

Required Hardware/Software

- ESP32 DevKit board.
- LM73 (I^2C temperature sensor) or MPU6050 (accelerometer/gyro).
- Node-RED with dashboard and MQTT broker (e.g., Mosquitto).

Procedure

1. Wire sensor to ESP32 (I^2C: SDA $\rightarrow$ GPIO21, SCL $\rightarrow$ GPIO22).
2. Program ESP32 to read sensor and publish data to MQTT topic (`/cps/sensors/data`).
3. In Node-RED:
   - Add an **MQTT-in** node subscribed to `/cps/sensors/data`.
   - Connect to a **JSON** node (parse payload).
   - Display values on **gauge** and **chart**.
4. Deploy and observe real-time sensor data.

Expected Output

Dashboard shows:

- Temperature in degrees Celsius (LM73) or acceleration (MPU6050).
- Time-series chart updating at 1 Hz.

Assessment Rubric (10 points)

2 Correct hardware wiring of sensor to ESP32.

3 ESP32 publishes valid MQTT messages.

3 Node-RED displays live values on gauge and chart.

2 Debug panel shows structured JSON payload.

Starter Node-RED Flow (Importable JSON)

```
[{"id":"tab3","type":"tab","label":"Lab3 - Sensor → MQTT → Dashboard"},
{"id":"broker","type":"mqtt-broker","name":"LocalBroker","broker":"localhost","port":
{"id":"mqin","type":"mqtt in","z":"tab3","name":"Sensor data",
"topic":"/cps/sensors/data","qos":"0","datatype":"auto","broker":"broker",
"x":180,"y":120,"wires":[["json3","chart3","gauge3"]]},
{"id":"json3","type":"json","z":"tab3","name":"","property":"payload",
"action":"","pretty":false,"x":360,"y":80,"wires":[["gauge3"]]},
{"id":"gauge3","type":"ui_gauge","z":"tab3","group":"uigrp3","order":1,
"gtype":"gage","title":"Sensor Value","label":"units",
"format":"{{payload.value}}","min":"0","max":"100","x":560,"y":80,"wires":[]},
{"id":"chart3","type":"ui_chart","z":"tab3","group":"uigrp3","order":2,
"label":"Sensor Time Series","chartType":"line","legend":"false",
"xformat":"HH:mm:ss","interpolate":"linear","ymin":"0","ymax":"100",
"removeOlder":1,"removeOlderPoints":"300","removeOlderUnit":"3600",
"useOneColor":false,"x":380,"y":160,"wires":[[]]},
{"id":"uigrp3","type":"ui_group","name":"Lab3","tab":"uitab3","order":1,
"disp":true,"width":"8","collapse":false},
{"id":"uitab3","type":"ui_tab","name":"CPS Labs","icon":"dashboard"}]
```

Starter ESP32 Code (Arduino)

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>

const char* WIFI_SSID = "YOUR_WIFI";
const char* WIFI_PASS = "YOUR_PASS";
const char* MQTT_HOST = "192.168.1.10";
const int MQTT_PORT = 1883;

WiFiClient espClient;
PubSubClient mqtt(espClient);

void wifiConnect(){
  WiFi.begin(WIFI_SSID, WIFI_PASS);
  while(WiFi.status()!=WL_CONNECTED) delay(300);
}
```

```
void mqttConnect(){
  mqtt.setServer(MQTT_HOST, MQTT_PORT);
  while(!mqtt.connected()){
    mqtt.connect("esp32-sensor");
    delay(500);
  }
}

void setup(){
  Serial.begin(115200);
  Wire.begin();
  wifiConnect();
  mqttConnect();
}

void loop(){
  if(WiFi.status()!=WL_CONNECTED) wifiConnect();
  if(!mqtt.connected()) mqttConnect();

  // Example: fake sensor data
  int sensorVal = analogRead(34); // or replace with LM73/MPU6050
  char msg[64];
  snprintf(msg,sizeof(msg), "{\"value\":%d}", sensorVal);
  mqtt.publish("/cps/sensors/data", msg);

  mqtt.loop();
  delay(1000);
}
```

Summary

Sensing and data acquisition form the first step in the CPS feedback loop. Key challenges include noise, calibration, sampling, and distributed sensing coordination. Lab 3.7 demonstrates practical sensor integration with ESP32 and Node-RED.

Review Questions

1. What are the four main stages of the sensing chain?

2. Why is Nyquist sampling important, and what happens if violated?
3. Explain quantization error and give an example for a 10-bit ADC with 5V reference.
4. What is calibration and why is it important for CPS accuracy?

Exercises

1. Modify Lab 3.7 to publish two different sensor values in JSON (e.g., temperature and acceleration).
2. Apply a moving-average filter in Node-RED function node to smooth noisy readings.
3. Sketch a sensor fusion setup for an autonomous vehicle combining GPS, accelerometer, and gyroscope.

Further Reading

- J. Wilson, *Sensor Technology Handbook*, Elsevier, 2005.
- M. Bishop, "Pattern Recognition and Data Fusion in CPS," IEEE SMC, 2018.
- H. Kopetz, *Real-Time Systems: Design Principles*, Springer, 2011.

Chapter 4

Actuation and Control Interfaces in CPS

Chapter Overview

While sensors provide the "eyes and ears" of Cyber-Physical Systems (CPS), actuators are their "hands and muscles," enabling CPS to influence the physical environment. This chapter explores types of actuators, interfacing methods, and the principles of feedback control. We emphasize real-time constraints, safety interlocks, and communication pathways between cyber and physical domains. The practical lab demonstrates how to use Node-RED dashboards to control actuators (LEDs, motors, servos) via MQTT and ESP32.

Learning Outcomes

After completing this chapter, students will be able to:

1. Classify different types of actuators (electrical, mechanical, pneumatic, hydraulic).
2. Explain the interfacing chain from cyber commands to physical actuation.
3. Discuss feedback loops and timing constraints in actuation.
4. Implement basic Node-RED dashboards to control actuators via ESP32.
5. Recognize safety and reliability concerns in CPS actuation.

Key Terms

Actuator, driver, PWM (Pulse Width Modulation), H-bridge, servo, relay, feedback loop, latency, jitter, fail-safe.

4.1 Actuator Types and Principles

- **Electrical:** LEDs, DC motors, stepper motors, servo motors.
- **Mechanical:** valves, pumps, robotic arms.
- **Electro-mechanical:** relays, solenoids.
- **Others:** piezoelectric devices, MEMS actuators, pneumatic/hydraulic actuators.

Example. A DC motor converts electrical current into torque; a servo motor uses PWM input to set angular position.

4.2 The Actuation Chain

Similar to sensing, actuation involves a chain of stages:

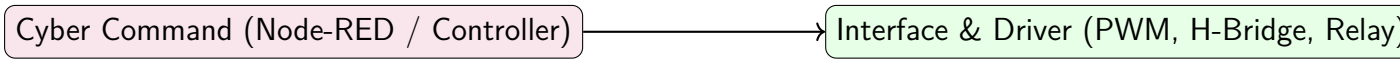

Figure 4.1: The actuation chain in CPS: from cyber command to actuator.

4.3 Feedback and Control Loops

Actuation rarely occurs open-loop; feedback ensures performance:

$$u(t) = f\Big(x_{\text{desired}}(t) - x_{\text{measured}}(t)\Big)$$

where $u(t)$ is the actuator command. Common controllers:

- **On/Off (Bang-Bang).** Example: thermostat.
- **Proportional (P).** Command proportional to error.
- **PID.** Combines proportional, integral, derivative actions.

4.4 Real-Time Constraints in Actuation

- **Latency:** command must reach actuator within D deadline.
- **Jitter:** variations in actuation timing can cause instability.
- **Resolution:** actuator may only respond at discrete steps (e.g., PWM duty cycles).

4.5 Safety and Reliability

- Hardware interlocks (emergency stop).
- Watchdog timers.
- Fail-safe states (default off).
- Redundancy for critical actuators.

4.6 Case Study: Smart Lighting Control

- **Sensor:** light-dependent resistor (LDR) measures brightness.
- **Actuator:** LED dimmer circuit driven by PWM.
- **Control:** Node-RED compares measured light with setpoint and adjusts PWM duty cycle.

4.7 Node-RED Lab 4: Actuator Control Dashboard

Objectives

- Use Node-RED dashboard to send actuator commands.
- Control LED (ON/OFF) and servo motor (angle) via ESP32.
- Understand MQTT-based cyber-to-physical actuation.

Required Hardware/Software

- ESP32 DevKit board.
- LED + resistor.
- Servo motor (SG90 or similar).
- Node-RED with MQTT broker (e.g., Mosquitto).

Procedure

1. In Node-RED, add:
   - **UI switch** node (topic `/cps/cmd/led`).
   - **UI slider** node (topic `/cps/cmd/servo`, 0–180).
   - Connect to **MQTT-out** nodes.
2. In ESP32 code, subscribe to these topics and control LED + servo accordingly.
3. Deploy flow and test by toggling switch and moving slider.

Expected Output

Dashboard controls LED ON/OFF and sets servo angle in real time. ESP32 executes commands via MQTT.

Assessment Rubric (10 points)

2 Dashboard UI elements created correctly.

3 MQTT messages sent from Node-RED to ESP32.

3 LED and servo respond accurately to commands.

2 Debug panel confirms correct payload structure.

Starter Node-RED Flow (Importable JSON)


```
[{"id":"tab4","type":"tab","label":"Lab4 - Actuator Control"},
{"id":"broker4","type":"mqtt-broker","name":"LocalBroker","broker":"localhost","port"
{"id":"uiSwitch","type":"ui_switch","z":"tab4","name":"LED","label":"LED",
"group":"uigrp4","order":1,"topic":"/cps/cmd/led","onvalue":"ON","offvalue":"OFF",
"x":150,"y":120,"wires":[["mqout4"]]},
{"id":"slider","type":"ui_slider","z":"tab4","name":"Servo","label":"Servo",
"group":"uigrp4","order":2,"min":0,"max":"180","step":1,"x":150,"y":180,"wires":[["fn
{"id":"fnServo","type":"function","z":"tab4","name":"pack angle",
"func":"msg.topic=\"/cps/cmd/servo\";\nmsg.payload={angle:Number(msg.payload)};\nretu
"x":340,"y":180,"wires":[["mqoutServo"]]},
{"id":"mqout4","type":"mqtt out","z":"tab4","name":"LED cmd","topic":"/cps/cmd/led",
"broker":"broker4","x":360,"y":120,"wires":[]},
{"id":"mqoutServo","type":"mqtt out","z":"tab4","name":"Servo cmd","topic":"/cps/cmd/
"broker":"broker4","x":540,"y":180,"wires":[]},
```

```
{"id":"uigrp4","type":"ui_group","name":"Lab4","tab":"uitab4","order":1,"disp":true,"
{"id":"uitab4","type":"ui_tab","name":"CPS Labs","icon":"dashboard"}]
```

Starter ESP32 Code (Arduino)

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <ESP32Servo.h>
#include <ArduinoJson.h>

const char* WIFI_SSID = "YOUR_WIFI";
const char* WIFI_PASS = "YOUR_PASS";
const char* MQTT_HOST = "192.168.1.10";
const int MQTT_PORT = 1883;

const int LED_PIN = 2;
const int SERVO_PIN = 14;

Servo myServo;
WiFiClient espClient;
PubSubClient mqtt(espClient);

void wifiConnect(){ WiFi.begin(WIFI_SSID, WIFI_PASS); while(WiFi.status()!=WL_CONNECT
void mqttConnect(){ mqtt.setServer(MQTT_HOST, MQTT_PORT); while(!mqtt.connected()){ m

void callback(char* topic, byte* payload, unsigned int len){
  String data((char*)payload,len);
  if(String(topic)=="/cps/cmd/led"){
    digitalWrite(LED_PIN, data=="ON"?HIGH:LOW);
  } else if(String(topic)=="/cps/cmd/servo"){
    StaticJsonDocument<128> doc;
    if(deserializeJson(doc,data)==DeserializationError::Ok){
      int angle = doc["angle"] | 90;
      myServo.write(constrain(angle,0,180));
    }
  }
}

void setup(){
```

```
  pinMode(LED_PIN, OUTPUT);
  myServo.attach(SERVO_PIN);
  wifiConnect();
  mqtt.setCallback(callback);
  mqttConnect();
}

void loop(){
  if(WiFi.status()!=WL_CONNECTED) wifiConnect();
  if(!mqtt.connected()) mqttConnect();
  mqtt.loop();
}
```

Summary

Actuators enable CPS to affect the physical world. The actuation chain includes cyber commands, drivers, and actuators, often within feedback loops. Real-time constraints, reliability, and safety interlocks are critical. Lab 4.7 demonstrates practical cyber-to-physical actuation with Node-RED and ESP32.

Review Questions

1. List three categories of actuators with examples.

2. Explain the role of PWM in actuator control.

3. Why are latency and jitter important in actuator commands?

4. Describe one fail-safe mechanism for actuators.

Exercises

1. Extend Lab 4.7 by adding a buzzer actuator controlled via MQTT.

2. Implement a simple bang-bang controller in Node-RED that turns LED ON if temperature >30°C (using sensor from Chapter 3).

3. Sketch a block diagram of a PID motor control loop for a CPS.

Further Reading

- H. K. Khalil, *Nonlinear Systems*, Prentice Hall, 2002.
- K. Ogata, *Modern Control Engineering*, Prentice Hall, 2010.
- D. E. Seborg et al., *Process Dynamics and Control*, Wiley, 2016.

Chapter 5

Communication and Networking in CPS

Chapter Overview

Communication networks form the "nervous system" of Cyber-Physical Systems (CPS). They connect sensors, actuators, controllers, and human interfaces across heterogeneous platforms and scales. This chapter introduces networking principles, CPS-specific requirements (latency, reliability, scalability, security), and key protocols used in IoT-enabled CPS. The practical lab implements an MQTT-based CPS pipeline with ESP32 sensor nodes and Node-RED dashboard.

Learning Outcomes

After completing this chapter, students will be able to:

1. Explain the role of communication in CPS closed-loop operation.
2. Describe wired and wireless communication options for CPS.
3. Compare IoT protocols (MQTT, CoAP, OPC UA, BLE, LoRa).
4. Identify networking challenges in CPS (latency, jitter, reliability, scalability).
5. Implement a distributed MQTT-based CPS network with Node-RED.

Key Terms

Latency, jitter, bandwidth, Quality of Service (QoS), MQTT, CoAP, OPC UA, BLE, LoRa, publish/subscribe, broker.

5.1 The Role of Communication in CPS

In CPS, sensing and actuation are linked by communication channels:

$$D = d_{\text{sense}} + d_{\text{net}} + d_{\text{compute}} + d_{\text{act}}$$

where d_{net} is often dominant. Network reliability and timing directly influence system safety and performance.

5.2 Wired vs. Wireless Communication

Wired. Ethernet, CAN bus, Modbus. Advantages: deterministic latency, high reliability. Limitations: poor mobility, high installation cost.

Wireless. Wi-Fi, Bluetooth/BLE, ZigBee, LoRa, 5G. Advantages: mobility, scalability. Limitations: interference, variable latency, power constraints.

5.3 IoT Protocols for CPS

- **MQTT (Message Queuing Telemetry Transport):** lightweight publish/subscribe protocol, ideal for CPS with many devices.
- **CoAP (Constrained Application Protocol):** RESTful over UDP, suited for constrained devices.
- **OPC UA:** industrial interoperability standard, rich semantics, security.
- **BLE (Bluetooth Low Energy):** low-power short-range links.
- **LoRa/LoRaWAN:** long-range, low-power, low-data-rate IoT connectivity.

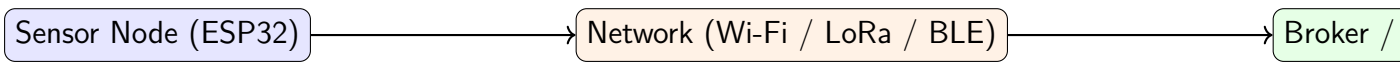

Figure 5.1: CPS communication chain: device → network → broker → dashboard.

5.4 Networking Challenges in CPS

- **Latency & Jitter:** must meet deadlines for stability.
- **Reliability:** packet loss → unsafe states.

- **Scalability:** hundreds/thousands of devices.
- **Security:** encryption, authentication, intrusion detection.

5.5 Case Study: Smart Grid Communication

Smart grids rely on distributed communication:

- Smart meters → edge gateway (via ZigBee/LoRa).
- Gateways → utility control (via LTE/5G).
- Control decisions → actuators (circuit breakers, inverters).

Networking requirements: <100 ms latency, high reliability, strong security.

5.6 Node-RED Lab 5: MQTT IoT Network

Objectives

- Set up an MQTT broker (Mosquitto).
- Publish sensor data from ESP32.
- Subscribe and visualize data in Node-RED.

Required Hardware/Software

- ESP32 board with sensor (e.g., LM73 or simulated analog input).
- MQTT broker (Mosquitto).
- Node-RED with dashboard.

Procedure

1. Install Mosquitto broker on PC or server.
2. Flash ESP32 with Arduino sketch that publishes sensor data to topic `/cps/temp/room`.
3. In Node-RED:
    - Add **MQTT-in** node (subscribe to `/cps/temp/room`).
    - Connect to **JSON** node.
    - Display on **chart** and **gauge**.
4. Deploy and observe streaming data.

Expected Output

Real-time temperature values plotted on dashboard chart and displayed on gauge.

Assessment Rubric (10 points)

2 MQTT broker installed and running.

3 ESP32 successfully publishes data.

3 Node-RED displays data on gauge and chart.

2 Debug panel confirms payload parsing.

Starter Node-RED Flow (Importable JSON)

```
[{"id":"tab5","type":"tab","label":"Lab5 - MQTT Network"},
{"id":"broker5","type":"mqtt-broker","name":"LocalBroker","broker":"localhost","port"
{"id":"mqin","type":"mqtt in","z":"tab5","name":"Room Temp","topic":"/cps/temp/room",
"qos":"0","datatype":"auto","broker":"broker5","x":150,"y":120,"wires":[["json5"]]},
{"id":"json5","type":"json","z":"tab5","name":"","property":"payload","action":"",
"pretty":false,"x":340,"y":120,"wires":[["gauge5","chart5","dbg5"]]},
{"id":"gauge5","type":"ui_gauge","z":"tab5","group":"uigrp5","order":1,
"gtype":"gage","title":"Room Temperature","label":"°C","format":"{{payload.t}}",
"min":0,"max":"50","x":540,"y":100,"wires":[]},
{"id":"chart5","type":"ui_chart","z":"tab5","group":"uigrp5","order":2,
"label":"Temperature Trend","chartType":"line","legend":"false","xformat":"HH:mm:ss",
"interpolate":"linear","ymin":"0","ymax":"50","removeOlder":1,"removeOlderPoints":"30
"removeOlderUnit":"3600","x":540,"y":160,"wires":[[]]},
{"id":"dbg5","type":"debug","z":"tab5","name":"debug","active":true,"tosidebar":true,
"complete":"true","x":540,"y":200,"wires":[]},
{"id":"uigrp5","type":"ui_group","name":"Lab5","tab":"uitab5","order":1,"disp":true,"
{"id":"uitab5","type":"ui_tab","name":"CPS Labs","icon":"dashboard"}]
```

Starter ESP32 Code (Arduino)

```
#include <WiFi.h>
#include <PubSubClient.h>

const char* WIFI_SSID = "YOUR_WIFI";
const char* WIFI_PASS = "YOUR_PASS";
const char* MQTT_HOST = "192.168.1.10";
```

```
const int MQTT_PORT = 1883;

WiFiClient espClient;
PubSubClient mqtt(espClient);

void wifiConnect(){
  WiFi.begin(WIFI_SSID, WIFI_PASS);
  while(WiFi.status()!=WL_CONNECTED) delay(300);
}
void mqttConnect(){
  mqtt.setServer(MQTT_HOST, MQTT_PORT);
  while(!mqtt.connected()){
    mqtt.connect("esp32-room");
    delay(500);
  }
}

void setup(){
  Serial.begin(115200);
  wifiConnect();
  mqttConnect();
}

void loop(){
  if(WiFi.status()!=WL_CONNECTED) wifiConnect();
  if(!mqtt.connected()) mqttConnect();

  float temp = 25.0 + (millis()%5000)/1000.0; // demo temperature
  char buf[64];
  snprintf(buf,sizeof(buf),"{\"t\":%.2f}",temp);
  mqtt.publish("/cps/temp/room", buf);

  mqtt.loop();
  delay(1000);
}
```

Summary

Networking in CPS connects sensors, actuators, and controllers across scales. Latency, jitter, reliability, and security are critical for safe and stable operation. MQTT provides a lightweight, scalable publish/subscribe mechanism, demonstrated in Lab 5.6.

Review Questions

1. What are the advantages and limitations of wired vs. wireless communication in CPS?
2. Compare MQTT and CoAP for CPS applications.
3. Explain how latency and jitter affect CPS control loops.
4. What role does an MQTT broker play in publish/subscribe systems?

Exercises

1. Modify Lab 5.6 to include two ESP32 sensor nodes publishing to different topics, and plot both on one dashboard chart.
2. Configure MQTT QoS levels (0,1,2) in Node-RED and observe differences in delivery.
3. Discuss security measures required for MQTT communication in CPS.

Further Reading

- A. Banks, R. Gupta, *MQTT Version 3.1.1*, OASIS Standard, 2014.
- OPC Foundation, *OPC Unified Architecture Specification*.
- E. Kovacs, "Standards and IoT Protocols for CPS," IEEE Communications Standards, 2019.
- H. Kopetz, *Real-Time Systems: Design Principles*, Springer, 2011.

Chapter 6

Discrete Control Algorithms in CPS

Chapter Overview

Cyber-Physical Systems (CPS) operate through control loops that map sensor data to actuator commands. Control algorithms are the "brains" of CPS, ensuring stability, performance, and safety. This chapter introduces discrete control principles, rule-based logic, finite state machines (FSMs), and classical controllers (P, PI, PID) in digital implementations. The practical lab demonstrates how to implement a rule-based control strategy in Node-RED, where sensor inputs trigger automated actuator responses.

Learning Outcomes

After completing this chapter, students will be able to:

1. Describe the role of discrete control algorithms in CPS.
2. Implement rule-based controllers using logic conditions.
3. Model system behavior using finite state machines.
4. Apply basic P/PI/PID controllers in a discrete-time setting.
5. Build automated CPS control flows in Node-RED with ESP32 sensors and actuators.

Key Terms

Control loop, setpoint, error signal, finite state machine (FSM), bang-bang control, proportional (P), integral (I), derivative (D), sampling period, stability.

6.1 From Continuous to Discrete Control

In continuous-time control:

$$u(t) = f\Big(x_{\text{desired}}(t) - x_{\text{measured}}(t)\Big)$$

CPS often implement this in discrete time with sampling interval T_s:

$$u[k] = f\Big(x_d[k] - x_m[k]\Big)$$

where k is the discrete time index. Digital controllers must respect timing constraints: $T_s \leq D$ (end-to-end deadline).

6.2 Rule-Based (Bang-Bang) Control

Concept. Actuators switch ON or OFF depending on thresholds.

$$u[k] = \begin{cases} \text{ON}, & x[k] > x_{\text{high}} \\ \text{OFF}, & x[k] < x_{\text{low}} \end{cases}$$

Example. A fan turns ON when temperature >30°C and OFF when <25°C.

6.3 Finite State Machines (FSMs)

FSMs capture CPS modes explicitly:

- **States:** OFF, IDLE, ACTIVE.
- **Transitions:** triggered by events (sensor threshold, timer).
- **Actions:** performed on entry/exit (turn LED ON).

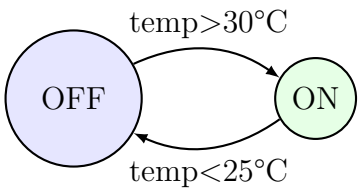

Figure 6.1: Finite State Machine (FSM) for thermostat control.

6.4 Classical Controllers (P, PI, PID)

Proportional Control (P).

$$u[k] = K_p e[k], \quad e[k] = x_d[k] - x_m[k]$$

Proportional–Integral (PI).

$$u[k] = K_p e[k] + K_i \sum_{i=0}^{k} e[i] T_s$$

Proportional–Integral–Derivative (PID).

$$u[k] = K_p e[k] + K_i \sum_{i=0}^{k} e[i] T_s + K_d \frac{e[k] - e[k-1]}{T_s}$$

Applications. PID controllers regulate motor speed, temperature, position, etc.

6.5 Timing and Stability Considerations

- Sample too slowly → unstable control.
- Communication jitter → oscillations or delayed response.
- Quantization → dead zones, limit cycles.

6.6 Case Study: Room Temperature Control

- **Sensor:** ESP32 reads temperature.
- **Controller:** Node-RED rule (IF temp >30°C, fan ON).
- **Actuator:** Fan connected to ESP32 pin.

This simple bang-bang control stabilizes temperature between 25–30°C.

6.7 Node-RED Lab 6: Automated CPS Control

Objectives

- Implement a threshold-based controller in Node-RED.
- Automatically trigger actuators based on sensor data.

- Visualize control actions on dashboard.

Required Hardware/Software

- ESP32 board with temperature sensor (LM73 or simulated analog).
- LED or fan actuator.
- Node-RED with MQTT broker.

Procedure

1. ESP32 publishes sensor values to topic `/cps/temp/room`.
2. In Node-RED:
   - Add **MQTT-in** node (subscribe to topic).
   - Add **JSON** node.
   - Connect to **function** node with logic:

     ```
     let t = Number(msg.payload.t);
     msg.topic = "/cps/cmd/fan";
     msg.payload = (t > 30) ? "ON" : "OFF";
     return msg;
     ```

   - Connect to **MQTT-out** node (to actuator).
   - Add **UI text** to show fan status.
3. Deploy and observe automatic control.

Expected Output

When temperature rises above 30°C, fan turns ON; when it drops below threshold, fan turns OFF. Dashboard displays fan status.

Assessment Rubric (10 points)

2 ESP32 publishes valid sensor data.

3 Node-RED implements correct IF-logic.

3 Actuator responds to conditions.

2 Dashboard shows fan status correctly.

Starter Node-RED Flow (Importable JSON)

```
[{"id":"tab6","type":"tab","label":"Lab6 - Auto Control"},
{"id":"broker6","type":"mqtt-broker","name":"LocalBroker","broker":"localhost","port"
{"id":"mqin6","type":"mqtt in","z":"tab6","name":"Temp in",
"topic":"/cps/temp/room","qos":"0","datatype":"auto","broker":"broker6",
"x":120,"y":120,"wires":[["json6"]]},
{"id":"json6","type":"json","z":"tab6","name":"","property":"payload",
"action":"","pretty":false,"x":290,"y":120,"wires":[["fn6"]]},
{"id":"fn6","type":"function","z":"tab6","name":"IF temp>30",
"func":"let t = Number(msg.payload.t);\nmsg.topic = \"/cps/cmd/fan\";\nmsg.payload =
"outputs":1,"x":470,"y":120,"wires":[["mqout6","status6"]]},
{"id":"mqout6","type":"mqtt out","z":"tab6","name":"Fan cmd",
"topic":"/cps/cmd/fan","qos":"","retain":"","broker":"broker6","x":650,"y":120,"wires
{"id":"status6","type":"ui_text","z":"tab6","group":"uigrp6","order":1,
"label":"Fan status","format":"{{msg.payload}}","x":650,"y":170,"wires":[]},
{"id":"uigrp6","type":"ui_group","name":"Lab6","tab":"uitab6","order":1,"disp":true,"
{"id":"uitab6","type":"ui_tab","name":"CPS Labs","icon":"dashboard"}]
```

Starter ESP32 Code (Arduino)

```
#include <WiFi.h>
#include <PubSubClient.h>

const char* WIFI_SSID="YOUR_WIFI";
const char* WIFI_PASS="YOUR_PASS";
const char* MQTT_HOST="192.168.1.10";
const int MQTT_PORT=1883;

const int FAN_PIN = 5;

WiFiClient espClient;
PubSubClient mqtt(espClient);

void wifiConnect(){ WiFi.begin(WIFI_SSID, WIFI_PASS); while(WiFi.status()!=WL_CONNECT
void mqttConnect(){ mqtt.setServer(MQTT_HOST,MQTT_PORT); while(!mqtt.connected()){ mq

void callback(char* topic, byte* payload, unsigned int len){
  String data((char*)payload,len);
  if(String(topic)=="/cps/cmd/fan"){
```

```
    digitalWrite(FAN_PIN, data=="ON"?HIGH:LOW);
  }
}

float readRoomTemp(){ return 25.0 + 10.0*sin(millis()/5000.0); } // demo

void setup(){
  pinMode(FAN_PIN,OUTPUT);
  wifiConnect();
  mqtt.setCallback(callback);
  mqttConnect();
}

void loop(){
  if(WiFi.status()!=WL_CONNECTED) wifiConnect();
  if(!mqtt.connected()) mqttConnect();

  float t=readRoomTemp();
  char buf[64];
  snprintf(buf,sizeof(buf),"{\"t\":%.2f}",t);
  mqtt.publish("/cps/temp/room",buf);

  mqtt.loop();
  delay(1000);
}
```

Summary

Discrete control algorithms form the computational core of CPS. Rule-based logic and FSMs handle events and thresholds, while P/PI/PID controllers regulate continuous processes. Lab 6.7 demonstrates automatic actuation in response to sensor data using Node-RED and ESP32.

Review Questions

1. What is the difference between continuous and discrete control?
2. How does a finite state machine represent CPS modes?
3. Write the discrete PID formula and explain each term.

4. Why are timing constraints important in digital control?

Exercises

1. Extend Lab 6.7 with hysteresis: fan ON >30°C, fan OFF <28°C.
2. Implement a three-state FSM in Node-RED for a smart light (OFF, DIM, BRIGHT).
3. Simulate a proportional controller in Node-RED: $u = K_p(x_d - x_m)$ with adjustable K_p via dashboard slider.

Further Reading

- K. Ogata, *Discrete-Time Control Systems*, Prentice Hall, 1995.
- E. A. Lee and S. Tripakis, "Foundations of CPS Modeling," ACM, 2012.
- H. K. Khalil, *Nonlinear Systems*, Prentice Hall, 2002.

Chapter 7

Prototyping Systems in CPS

Chapter Overview

Prototyping is a critical phase in Cyber-Physical Systems (CPS) engineering. It bridges theoretical models and final deployments, allowing engineers to validate concepts, test interactions between cyber and physical components, and iterate designs rapidly. This chapter examines prototyping methodologies, hardware and software platforms, hardware-in-the-loop (HIL) testing, and simulation tools. The practical lab demonstrates building a CPS prototype with both simulated and physical components using Node-RED and ESP32.

Learning Outcomes

After completing this chapter, students will be able to:

1. Explain the role of prototyping in CPS design workflows.
2. Identify common hardware and software platforms for CPS prototyping.
3. Distinguish between rapid prototyping and hardware-in-the-loop (HIL) testing.
4. Integrate real and simulated components into a prototype system.
5. Implement a Node-RED flow that synchronizes physical and virtual CPS states.

Key Terms

Prototyping, rapid prototyping, hardware-in-the-loop (HIL), co-simulation, Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), real-time testing.

7.1 The Role of Prototyping in CPS

CPS are complex, safety-critical systems; prototyping reduces risk by validating design choices early. Benefits:

- Explore different architectures quickly.
- Validate sensing, actuation, and networking performance.
- Identify integration issues before full-scale deployment.
- Enable iterative, user-centered design with feedback.

7.2 Prototyping Methodologies

Model-in-the-Loop (MIL). Mathematical models are tested in simulation (e.g., Simulink, Modelica).

Software-in-the-Loop (SIL). Software (controller code) is tested with simulated plant.

Hardware-in-the-Loop (HIL). Real hardware interacts with simulated plant/environment. Allows testing under realistic but safe conditions.

Rapid Prototyping. Using flexible tools (e.g., Arduino, ESP32, Raspberry Pi, Node-RED) to quickly assemble functional CPS prototypes.

7.3 Hardware Platforms for CPS Prototyping

- **Microcontrollers:** Arduino, ESP32 — cost-effective, widely supported.
- **Single-board computers:** Raspberry Pi — suitable for edge analytics.
- **FPGAs:** Xilinx/Intel — real-time deterministic control, hardware acceleration.
- **Industrial controllers:** PLCs — rugged, industry-standard.

7.4 Software Platforms and Tools

- **Node-RED:** flow-based prototyping for IoT/CPS.
- **MATLAB/Simulink:** plant and controller co-simulation.
- **Modelica:** object-oriented modeling of physical systems.

- **Docker/Kubernetes:** deploying cyber services in scalable ways.

7.5 Hardware-in-the-Loop (HIL) Prototyping

- Real controller hardware connected to a simulated environment.
- Enables testing dangerous scenarios safely.
- Example: EV battery management system controller tested against a simulated battery.

7.6 Case Study: Smart Traffic Light Prototype

- **Virtual:** Node-RED simulates car arrivals.
- **Physical:** ESP32 drives real LEDs representing traffic lights.
- **Integration:** Both systems exchange state via MQTT.

7.7 Node-RED Lab 7: Hardware-in-the-Loop CPS Prototype

Objectives

- Create a hybrid CPS prototype combining simulated and real components.
- Synchronize simulated sensor data with physical actuator responses.
- Validate consistency between virtual and physical CPS states.

Required Hardware/Software

- ESP32 board with LED (representing actuator).
- Node-RED with dashboard and MQTT broker.
- Simulated sensor inputs (Inject nodes in Node-RED).

Procedure

1. In Node-RED:
   - Use **inject** node to simulate temperature sensor.
   - Send values via function node to MQTT topic `/cps/sensors/real`.
   - Add a simulated path: another inject node → function → chart.
   - Add **MQTT-out** to control ESP32 LED (topic `/cps/cmd/led`).
2. ESP32 subscribes to actuator topic and drives LED ON/OFF.
3. Compare simulated and physical states on dashboard chart.

Expected Output

Dashboard shows both simulated and real states. ESP32 LED reflects simulated sensor threshold logic.

Assessment Rubric (10 points)

2 Correct Node-RED simulation of sensor data.

3 ESP32 actuator responds to control commands.

3 Dashboard compares simulated vs real states.

2 Consistency documented with screenshots.

Starter Node-RED Flow (Importable JSON)

```
[{"id":"tab7","type":"tab","label":"Lab7 - HIL Prototype"},
{"id":"broker7","type":"mqtt-broker","name":"LocalBroker","broker":"localhost","port"
{"id":"inj7","type":"inject","z":"tab7","name":"Simulated Sensor","props":[{"p":"payl
"payload":"{\"value\":25}","payloadType":"json","x":160,"y":100,"wires":[["chart7"]]}
{"id":"mqin7","type":"mqtt in","z":"tab7","name":"Real Sensor","topic":"/cps/sensors/
"broker":"broker7","datatype":"auto","x":160,"y":160,"wires":[["chart7"]]},
{"id":"chart7","type":"ui_chart","z":"tab7","group":"uigrp7","order":1,"label":"Sim v
"chartType":"line","legend":"true","xformat":"HH:mm:ss","interpolate":"linear","ymin"
"removeOlder":1,"removeOlderPoints":"300","removeOlderUnit":"3600","x":380,"y":130,"w
{"id":"uigrp7","type":"ui_group","name":"Lab7","tab":"uitab7","order":1,"disp":true,"
{"id":"uitab7","type":"ui_tab","name":"CPS Labs","icon":"dashboard"}]
```

Starter ESP32 Code (Arduino)

```
#include <WiFi.h>
#include <PubSubClient.h>

const char* WIFI_SSID="YOUR_WIFI";
const char* WIFI_PASS="YOUR_PASS";
const char* MQTT_HOST="192.168.1.10";
const int MQTT_PORT=1883;

const int LED_PIN = 2;

WiFiClient espClient;
PubSubClient mqtt(espClient);

void wifiConnect(){ WiFi.begin(WIFI_SSID, WIFI_PASS); while(WiFi.status()!=WL_CONNECTI
void mqttConnect(){ mqtt.setServer(MQTT_HOST,MQTT_PORT); while(!mqtt.connected()){ mq

void callback(char* topic, byte* payload, unsigned int len){
  String data((char*)payload,len);
  if(String(topic)=="/cps/cmd/led"){
    digitalWrite(LED_PIN, data=="ON"?HIGH:LOW);
  }
}

void setup(){
  pinMode(LED_PIN,OUTPUT);
  wifiConnect();
  mqtt.setCallback(callback);
  mqttConnect();
}

void loop(){
  if(WiFi.status()!=WL_CONNECTED) wifiConnect();
  if(!mqtt.connected()) mqttConnect();
  mqtt.loop();
}
```

Summary

Prototyping allows CPS engineers to test designs quickly and safely before full deployment. Rapid prototyping platforms (ESP32, Node-RED) and methods like hardware-in-the-loop reduce risks and accelerate iteration. Lab 7.7 demonstrates integrating simulated and physical CPS components.

Review Questions

1. What is the difference between MIL, SIL, and HIL testing?
2. Why is prototyping critical in CPS development?
3. Give two advantages and two limitations of rapid prototyping platforms like ESP32.
4. Explain how HIL can be used to test safety-critical CPS.

Exercises

1. Extend Lab 7.7 by adding a second actuator (servo motor) controlled by simulated data.
2. Build a prototype smart traffic light system combining simulated arrivals and physical LED lights.
3. Integrate a dashboard slider to control simulation parameters (e.g., temperature threshold).

Further Reading

- P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica*, Wiley, 2015.
- R. Rajkumar et al., "Cyber-Physical Systems: The Next Computing Revolution," IEEE DAC, 2010.
- L. Monostori, "Cyber-Physical Production Systems," Procedia CIRP, 2014.

Chapter 8

Software and Hardware Implementation in CPS

Chapter Overview

Software and hardware are tightly integrated in Cyber-Physical Systems (CPS). Sensors, actuators, and communication networks must be orchestrated through software that meets real-time constraints, while hardware platforms must be selected to balance performance, energy, and cost. This chapter explores embedded software development, operating systems, middleware, hardware architectures, and co-design strategies. The practical lab demonstrates a CPS traffic light prototype combining software logic (FSM) and hardware actuation (LEDs) with Node-RED and ESP32.

Learning Outcomes

After completing this chapter, students will be able to:

1. Explain the role of embedded software in CPS.
2. Differentiate between bare-metal and RTOS-based implementations.
3. Describe middleware platforms used for CPS integration.
4. Evaluate hardware options: MCUs, SBCs, and FPGAs.
5. Implement a combined software-hardware CPS prototype in Node-RED and ESP32.

Key Terms

Embedded software, RTOS, middleware, Node-RED, ROS, OPC UA, microcontroller (MCU), single-board computer (SBC), FPGA, hardware-software co-design.

8.1 Embedded Software in CPS

Bare-metal. Simple firmware runs directly on microcontrollers without operating system. Advantages: low overhead, deterministic timing. Limitations: limited scalability and maintainability.

RTOS-based. Real-Time Operating Systems (FreeRTOS, Zephyr, VxWorks) provide multitasking, task scheduling, and deterministic timing. Widely used in CPS for balancing concurrency and predictability.

8.2 Middleware for CPS

Middleware bridges distributed devices:

- **Node-RED:** rapid prototyping, dashboard, MQTT integration.
- **ROS/ROS 2:** robotics middleware for sensor/actuator integration.
- **OPC UA:** industrial interoperability, rich semantics, security.
- **DDS (Data Distribution Service):** real-time publish/subscribe for critical systems.

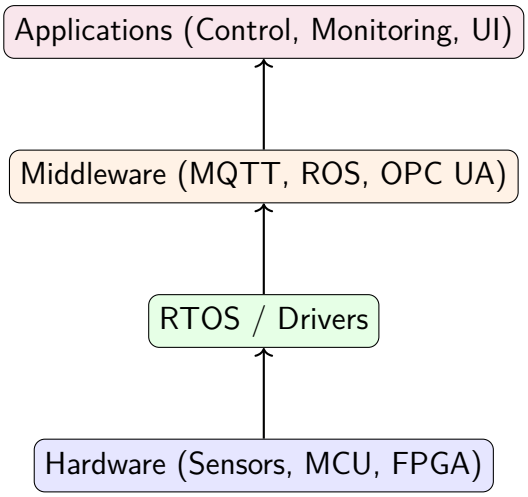

Figure 8.1: Software-hardware stack in CPS implementation.

8.3 Hardware Platforms

- **Microcontrollers (MCUs):** Arduino, ESP32 — lightweight, low power, real-time.

- **Single-board computers (SBCs):** Raspberry Pi — supports Linux, suitable for edge analytics.
- **FPGAs:** deterministic timing, hardware acceleration for AI/ML workloads in CPS.
- **Heterogeneous platforms:** combinations (e.g., Zynq SoCs) integrating CPUs and FPGAs.

8.4 Co-Design of Hardware and Software

Hardware-software co-design aligns system requirements with platform capabilities:

- Partition computation: edge vs cloud, CPU vs FPGA.
- Optimize timing: allocate tasks to meet latency budgets.
- Ensure safety/security: hardware watchdogs, software fail-safes.

8.5 Case Study: Smart Building HVAC Control

- **Hardware:** ESP32 + temperature sensors + relays.
- **Software:** Node-RED middleware, threshold/PID controllers.
- **Integration:** RTOS tasks manage sensing and actuation, MQTT synchronizes with cloud dashboard.

8.6 Node-RED Lab 8: Traffic Light CPS Prototype

Objectives

- Implement a finite state machine (FSM) traffic light controller.
- Combine simulated car arrivals with physical LED actuation.
- Demonstrate CPS integration of software logic and hardware.

Required Hardware/Software

- ESP32 board with 3 LEDs (red, yellow, green).
- Node-RED with MQTT broker.
- Dashboard nodes for visualization.

Procedure

1. ESP32 subscribes to MQTT topics for LED commands.
2. In Node-RED:
   - Inject “car detected” events.
   - Function node implements FSM logic (cycle R→G→Y→R).
   - MQTT-out nodes publish LED commands.
   - Dashboard shows LED states via UI indicators.
3. Deploy and observe traffic light cycle with both simulated and physical outputs.

Expected Output

ESP32 LEDs cycle according to FSM states. Node-RED dashboard displays real-time states.

Assessment Rubric (10 points)

2 Node-RED FSM implemented correctly.

3 MQTT messages control physical LEDs.

3 Dashboard shows state transitions.

2 Consistency between virtual and physical states.

Starter Node-RED Flow (Importable JSON)

```
[{"id":"tab8","type":"tab","label":"Lab8 - Traffic Light"},
{"id":"broker8","type":"mqtt-broker","name":"LocalBroker","broker":"localhost","port"
{"id":"car8","type":"inject","z":"tab8","name":"Car detected","props":[{"p":"payload"
"payload":"car","payloadType":"str","x":130,"y":100,"wires":[["fsm8"]]},
{"id":"fsm8","type":"function","z":"tab8","name":"FSM",
"func":"var s = context.get('s')||'RED';\nif (msg.payload==='car' && s==='RED'){s='GR
"x":330,"y":130,"wires":[["mqout8","uiText"]]},
{"id":"mqout8","type":"mqtt out","z":"tab8","name":"LED cmd","topic":"/cps/cmd/led",
"broker":"broker8","x":520,"y":130,"wires":[]},
{"id":"uiText","type":"ui_text","z":"tab8","group":"uigrp8","order":1,
"label":"State","format":"{{msg.payload}}","x":520,"y":90,"wires":[]},
{"id":"uigrp8","type":"ui_group","name":"Lab8","tab":"uitab8","order":1,"disp":true,"
{"id":"uitab8","type":"ui_tab","name":"CPS Labs","icon":"dashboard"}]
```

Starter ESP32 Code (Arduino)

```
#include <WiFi.h>
#include <PubSubClient.h>

const char* WIFI_SSID="YOUR_WIFI";
const char* WIFI_PASS="YOUR_PASS";
const char* MQTT_HOST="192.168.1.10";
const int MQTT_PORT=1883;

const int RED_PIN=16, YELLOW_PIN=17, GREEN_PIN=18;

WiFiClient espClient;
PubSubClient mqtt(espClient);

void wifiConnect(){ WiFi.begin(WIFI_SSID,WIFI_PASS); while(WiFi.status()!=WL_CONNECTE
void mqttConnect(){ mqtt.setServer(MQTT_HOST,MQTT_PORT); while(!mqtt.connected()){ mq

void callback(char* topic, byte* payload, unsigned int len){
  String data((char*)payload,len);
  digitalWrite(RED_PIN, data=="RED"?HIGH:LOW);
  digitalWrite(YELLOW_PIN, data=="YELLOW"?HIGH:LOW);
  digitalWrite(GREEN_PIN, data=="GREEN"?HIGH:LOW);
}

void setup(){
  pinMode(RED_PIN,OUTPUT);
  pinMode(YELLOW_PIN,OUTPUT);
  pinMode(GREEN_PIN,OUTPUT);
  wifiConnect();
  mqtt.setCallback(callback);
  mqttConnect();
}

void loop(){
  if(WiFi.status()!=WL_CONNECTED) wifiConnect();
  if(!mqtt.connected()) mqttConnect();
  mqtt.loop();
}
```

Summary

Software and hardware integration defines CPS functionality. Embedded software, RTOS, and middleware orchestrate hardware resources to meet real-time and safety constraints. Prototyping with Node-RED and ESP32 demonstrates the interplay of FSM logic and physical actuation. Lab 8.6 illustrates a traffic light CPS with both simulated and physical outputs.

Review Questions

1. Compare bare-metal and RTOS-based software implementations in CPS.
2. What role does middleware play in CPS integration?
3. List three hardware platforms for CPS prototyping and describe their strengths.
4. Why is hardware-software co-design essential in CPS?

Exercises

1. Extend Lab 8.6 with pedestrian crossing signals (add blue LED).
2. Modify FSM to include a timer for each state duration.
3. Implement traffic light control for two intersections and coordinate them via MQTT.

Further Reading

- FreeRTOS documentation: https://freertos.org
- OPC Foundation, *OPC UA Specifications*.
- ROS 2 documentation: https://docs.ros.org/en/foxy
- H. Kopetz, *Real-Time Systems: Design Principles*, Springer, 2011.

Chapter 9

Designing CPS Projects Incorporating Infrastructure Systems

Chapter Overview

Cyber-Physical Systems (CPS) increasingly underpin critical infrastructure such as smart cities, energy grids, water distribution, and intelligent transportation systems. Designing CPS projects at this scale requires careful integration of sensing, communication, computation, and actuation across heterogeneous platforms, while ensuring safety, security, and scalability. This chapter introduces methodologies for CPS infrastructure design, highlights domain-specific examples, and explores integration challenges. The practical lab demonstrates secure CPS communication with MQTT authentication and encryption.

Learning Outcomes

After completing this chapter, students will be able to:

1. Describe the role of CPS in infrastructure systems.
2. Identify domain-specific requirements for smart grids, transportation, and cities.
3. Apply layered design methodologies for infrastructure-scale CPS.
4. Recognize scalability, interoperability, and governance challenges.
5. Implement secure CPS communication using Node-RED and ESP32.

Key Terms

Infrastructure CPS, smart grid, intelligent transportation system (ITS), smart city, interoperability, scalability, security, privacy, governance.

9.1 CPS and Infrastructure Systems

Infrastructure CPS extends small-scale prototypes to city- or nation-wide systems:

- **Smart grids:** distributed energy resources, demand response, real-time monitoring.
- **Smart transportation:** vehicle-to-infrastructure (V2I) communication, adaptive signals, traffic safety.
- **Smart cities:** integrated lighting, waste management, environmental monitoring.
- **Water management:** real-time sensing of flow, pressure, and leakage.

9.2 Methodology for Infrastructure CPS Design

Step 1: Requirements Analysis. Identify functional, performance, and regulatory requirements.

Step 2: Layered Architecture. Adopt reference models (e.g., RAMI 4.0, IIRA, 5C).

Step 3: Integration Strategy. Choose standards (MQTT, OPC UA, CoAP) for interoperability.

Step 4: Security and Governance. Authentication, authorization, encryption, audit trails.

Step 5: Scalability Planning. Cloud/edge partitioning, containerization, orchestration (Kubernetes).

9.3 Challenges in Infrastructure CPS

- **Scalability:** thousands of devices, high data volume.
- **Interoperability:** multi-vendor, multi-protocol ecosystems.
- **Resilience:** fault tolerance, redundancy, recovery after cyber-attacks.

- **Governance:** data ownership, privacy, regulatory compliance.

9.4 Case Study: Smart City Lighting

- Sensors measure ambient light and occupancy.
- Node-RED aggregates data across districts.
- Actuators dim or brighten streetlights adaptively.
- Secure MQTT ensures only authorized controllers can issue commands.

9.5 Node-RED Lab 9: Secure CPS Communication

Objectives

- Implement secure MQTT communication in a CPS pipeline.
- Configure broker authentication and TLS.
- Validate that only authorized devices exchange messages.

Required Hardware/Software

- ESP32 board with sensor (temperature).
- Mosquitto broker configured with username/password and TLS certificates.
- Node-RED with MQTT nodes.

Procedure

1. Configure Mosquitto broker:

```
listener 8883
allow_anonymous false
password_file /etc/mosquitto/passwd
cafile ca.crt
certfile server.crt
keyfile server.key
```

2. ESP32 connects with credentials and publishes sensor data.
3. Node-RED MQTT-in subscribes with authentication enabled.
4. Dashboard visualizes data securely.

Expected Output

MQTT broker accepts only authenticated devices. Node-RED dashboard displays authorized sensor data. Unauthorized clients are rejected.

Assessment Rubric (10 points)

2 Broker configured with authentication and TLS.

3 ESP32 publishes securely.

3 Node-RED subscribes with credentials and displays data.

2 Unauthorized clients blocked.

Starter Node-RED Flow (Importable JSON)

```
[{"id":"tab9","type":"tab","label":"Lab9 - Secure CPS"},
{"id":"broker9","type":"mqtt-broker","name":"SecureBroker","broker":"localhost","port
"tls":"tls9","credentials":{"user":"student","password":"pass123"}},
{"id":"mqin9","type":"mqtt in","z":"tab9","name":"Temp in",
"topic":"/cps/temp/secure","broker":"broker9","datatype":"auto","x":160,"y":120,"wire
{"id":"gauge9","type":"ui_gauge","z":"tab9","group":"uigrp9","order":1,
"title":"Secure Temp","label":"°C","format":"{{payload.t}}","min":0,"max":"50",
"x":360,"y":120,"wires":[]},
{"id":"uigrp9","type":"ui_group","name":"Lab9","tab":"uitab9","order":1,"disp":true,"
{"id":"uitab9","type":"ui_tab","name":"CPS Labs","icon":"dashboard"},
{"id":"tls9","type":"tls-config","name":"TLS Config","cert":"","key":"","ca":"",
"verifyservercert":true}]
```

Starter ESP32 Code (Arduino, with TLS + Auth)

```
#include <WiFiClientSecure.h>
#include <PubSubClient.h>
#include <WiFi.h>

const char* WIFI_SSID="YOUR_WIFI";
const char* WIFI_PASS="YOUR_PASS";
const char* MQTT_HOST="192.168.1.10";
const int MQTT_PORT=8883;
const char* MQTT_USER="student";
const char* MQTT_PASSWD="pass123";
```

```
WiFiClientSecure espClient;
PubSubClient mqtt(espClient);

void wifiConnect(){ WiFi.begin(WIFI_SSID,WIFI_PASS); while(WiFi.status()!=WL_CONNECTE
void mqttConnect(){ mqtt.setServer(MQTT_HOST,MQTT_PORT); while(!mqtt.connected()){ mq

void setup(){
  Serial.begin(115200);
  wifiConnect();
  espClient.setInsecure(); // for demo; load CA cert in production
  mqttConnect();
}

void loop(){
  if(WiFi.status()!=WL_CONNECTED) wifiConnect();
  if(!mqtt.connected()) mqttConnect();

  float temp = 22.0 + (millis()%3000)/100.0;
  char buf[64];
  snprintf(buf,sizeof(buf),"{\"t\":%.2f}",temp);
  mqtt.publish("/cps/temp/secure", buf);

  mqtt.loop();
  delay(1000);
}
```

Summary

Infrastructure CPS projects extend prototypes into large-scale systems. Key considerations include scalability, interoperability, and governance. Security and privacy become paramount. Lab 9.5 demonstrates secure CPS communication, a critical foundation for smart infrastructure.

Review Questions

1. What domains of infrastructure benefit most from CPS?
2. Why is interoperability essential in infrastructure-scale CPS?

3. Describe three challenges in scaling CPS to smart city systems.
4. How does TLS improve security in CPS communication?

Exercises

1. Extend Lab 9.5 by adding a second sensor node and configure role-based authorization.
2. Design a CPS for smart water management, identifying sensors, actuators, and communication layers.
3. Research how OPC UA could be applied to an energy grid CPS.

Further Reading

- Industrial Internet Consortium, *Industrial Internet Reference Architecture (IIRA)*, 2019.
- Plattform Industrie 4.0, *RAMI 4.0 Reference Model*, 2015.
- P. Derler, E. A. Lee, "Modeling Cyber-Physical Systems," Proc. IEEE, 2012.
- S. Karnouskos, "Cyber-Physical Systems in Smart Grids," IEEE, 2011.

Chapter 10

CPS System Integration and Testing

Chapter Overview

Integration and testing are critical phases in Cyber-Physical Systems (CPS) engineering. As systems grow from prototypes to infrastructure-scale deployments, multiple hardware and software components must interact seamlessly. Rigorous testing ensures safety, reliability, and performance. This chapter covers system integration strategies, verification and validation techniques, testing frameworks, simulation tools, and fault injection methods. The practical lab demonstrates how to implement a digital twin in Node-RED to mirror CPS states for testing and monitoring.

Learning Outcomes

After completing this chapter, students will be able to:

1. Explain the importance of integration and testing in CPS lifecycles.
2. Distinguish between verification and validation.
3. Apply test-driven development (TDD) and continuous integration in CPS.
4. Use simulation and digital twin models for testing CPS components.
5. Build a Node-RED digital twin prototype to mirror CPS states.

Key Terms

System integration, verification, validation, test-driven development (TDD), continuous integration (CI), digital twin, fault injection, simulation, regression testing.

10.1 The Role of Integration in CPS

CPS often include heterogeneous components:

- Microcontrollers (ESP32, Arduino).
- Edge devices (Raspberry Pi).
- Middleware (Node-RED, OPC UA).
- Cloud platforms (AWS IoT, Azure IoT Hub).

Integration involves connecting these layers into a coherent whole. Poor integration leads to communication failures, timing issues, and unsafe behaviors.

10.2 Verification vs. Validation

- **Verification:** "Are we building the system right?" — checking against specifications.
- **Validation:** "Are we building the right system?" — checking against stakeholder needs.

10.3 Testing Approaches

Unit Testing. Test software modules in isolation.

Integration Testing. Test interactions between components.

System Testing. Evaluate full CPS, including hardware and network.

Acceptance Testing. Confirm system meets end-user needs.

10.4 Test-Driven Development (TDD) in CPS

1. Write test case for desired functionality.
2. Implement code to pass test.
3. Refactor, rerun tests, and repeat.

TDD improves reliability, though hardware dependencies complicate process. Simulators and emulators can assist.

10.5 Continuous Integration (CI)

CI tools (GitHub Actions, Jenkins) automatically build and test CPS software. Benefits:

- Detect regressions early.
- Ensure reproducibility.
- Support team collaboration.

10.6 Simulation and Digital Twins

Simulation. Run models of physical processes to test control strategies. Tools: MATLAB/Simulink, Modelica, NS-3.

Digital Twins. Digital replicas of physical CPS components synchronized in real-time. Used for monitoring, prediction, fault detection, and testing.

10.7 Fault Injection in CPS Testing

- Inject sensor noise or drop packets to test robustness.
- Simulate hardware failure (e.g., actuator stuck).
- Evaluate system response and fail-safe mechanisms.

10.8 Case Study: Smart Grid Digital Twin Testing

Utility companies use digital twins of substations to test load balancing algorithms. Failures can be simulated safely, validating robustness before deployment.

10.9 Node-RED Lab 10: Digital Twin of CPS State Mirroring

Objectives

- Build a digital twin that mirrors CPS states in Node-RED.
- Compare real ESP32 sensor/actuator data with twin state.
- Demonstrate integration testing via state consistency checks.

Required Hardware/Software

- ESP32 board with sensor (e.g., LM73).
- Node-RED with dashboard and MQTT broker.
- Database (optional, e.g., InfluxDB) for state history.

Procedure

1. ESP32 publishes sensor data and actuator state to MQTT topics.
2. In Node-RED:
    - Add **MQTT-in** nodes for real data.
    - Store real data in a state database or memory.
    - Mirror state to a dashboard digital twin.
    - Compare expected vs real states; flag mismatches.
3. Use dashboard charts to monitor live and twin states.

Expected Output

Dashboard displays both real sensor/actuator states and their digital twin mirrors. Discrepancies trigger alerts.

Assessment Rubric (10 points)

2 ESP32 publishes correct sensor/actuator data.

3 Node-RED twin mirrors states accurately.

3 Dashboard visualizes real vs twin states.

2 Alerts generated on mismatch.

Starter Node-RED Flow (Importable JSON)


```
[{"id":"tab10","type":"tab","label":"Lab10 - Digital Twin"},
{"id":"broker10","type":"mqtt-broker","name":"LocalBroker","broker":"localhost","port
{"id":"mqin10","type":"mqtt in","z":"tab10","name":"Sensor",
"topic":"/cps/sensors/room","broker":"broker10","datatype":"auto","x":150,"y":100,"wi
{"id":"fn10","type":"function","z":"tab10","name":"Mirror State",
"func":"flow.set('realState',msg.payload);\nmsg.topic='/cps/twin/room';\nreturn msg;"
```

```
"x":350,"y":100,"wires":[["mqout10","chart10"]]},
{"id":"mqout10","type":"mqtt out","z":"tab10","name":"Twin Pub","topic":"/cps/twin/ro
"broker":"broker10","x":540,"y":100,"wires":[]},
{"id":"chart10","type":"ui_chart","z":"tab10","group":"uigrp10","order":1,
"label":"Twin vs Real","chartType":"line","legend":"true","xformat":"HH:mm:ss",
"interpolate":"linear","ymin":"0","ymax":"50","removeOlder":1,
"removeOlderPoints":"300","removeOlderUnit":"3600","x":540,"y":160,"wires":[[]]},
{"id":"uigrp10","type":"ui_group","name":"Lab10","tab":"uitab10","order":1,"disp":tru
{"id":"uitab10","type":"ui_tab","name":"CPS Labs","icon":"dashboard"}]
```

Starter ESP32 Code (Arduino)

```
#include <WiFi.h>
#include <PubSubClient.h>

const char* WIFI_SSID="YOUR_WIFI";
const char* WIFI_PASS="YOUR_PASS";
const char* MQTT_HOST="192.168.1.10";
const int MQTT_PORT=1883;

WiFiClient espClient;
PubSubClient mqtt(espClient);

void wifiConnect(){ WiFi.begin(WIFI_SSID,WIFI_PASS); while(WiFi.status()!=WL_CONNECTE
void mqttConnect(){ mqtt.setServer(MQTT_HOST,MQTT_PORT); while(!mqtt.connected()){ mq

float readTemp(){ return 25.0 + (millis()%5000)/500.0; } // simulated

void setup(){
  Serial.begin(115200);
  wifiConnect();
  mqttConnect();
}

void loop(){
  if(WiFi.status()!=WL_CONNECTED) wifiConnect();
  if(!mqtt.connected()) mqttConnect();

  float t=readTemp();
```

```
  char buf[64];
  snprintf(buf,sizeof(buf),"{\"t\":%.2f}",t);
  mqtt.publish("/cps/sensors/room",buf);

  mqtt.loop();
  delay(1000);
}
```

Summary

System integration and testing ensure CPS operate safely and reliably in real-world conditions. Verification and validation, TDD, CI, and digital twins support systematic testing. Lab 10.9 illustrates how Node-RED can be used to mirror and validate CPS states against real data.

Review Questions

1. Differentiate between verification and validation in CPS.
2. Why is fault injection important for CPS testing?
3. How do digital twins support system integration testing?
4. What is the role of continuous integration (CI) in CPS software development?

Exercises

1. Extend Lab 10.9 with anomaly detection: trigger alarm if real–twin mismatch persists for more than 5 seconds.
2. Implement a test harness in Node-RED that replays recorded sensor data to validate control logic.
3. Discuss how fault injection could be applied to an autonomous vehicle CPS.

Further Reading

- B. Selic, “Verification and Validation of Cyber-Physical Systems,” IEEE, 2012.
- P. Tabuada, *Verification and Control of Hybrid Systems*, Springer, 2009.
- G. Fainekos et al., “Simulation-based Testing of CPS,” IEEE Trans. CAD, 2013.

- OPC Foundation, *Digital Twin Standards*, 2020.

Chapter 11

CPS Security and Privacy

Chapter Overview

Security and privacy are fundamental challenges in Cyber-Physical Systems (CPS), where cyber attacks can directly endanger physical processes and human safety. This chapter explores threat models, attack surfaces, and design principles for securing CPS. Topics include authentication, encryption, intrusion detection, anomaly monitoring, and privacy-preserving data handling. The practical lab demonstrates a Node-RED intrusion detection prototype that flags anomalous sensor data.

Learning Outcomes

After completing this chapter, students will be able to:

1. Identify major security and privacy threats in CPS.
2. Describe CPS-specific attack surfaces and vulnerabilities.
3. Apply fundamental security principles: confidentiality, integrity, availability (CIA).
4. Implement anomaly detection for CPS security monitoring.
5. Build a Node-RED intrusion detection flow that detects abnormal data patterns.

Key Terms

Attack surface, confidentiality, integrity, availability (CIA), intrusion detection system (IDS), anomaly detection, denial of service (DoS), man-in-the-middle (MITM), encryption, privacy-preserving CPS.

11.1 Threat Models in CPS

CPS combine cyber and physical components, introducing unique vulnerabilities:

- **Cyber attacks:** malware, phishing, buffer overflows.
- **Network attacks:** MITM, packet replay, denial of service (DoS).
- **Physical attacks:** tampering with sensors/actuators.
- **Privacy threats:** unauthorized access to sensitive data (e.g., health, location).

11.2 Attack Surfaces in CPS

- **Device layer:** microcontrollers, sensors, actuators.
- **Communication layer:** protocols (MQTT, CoAP, OPC UA).
- **Control layer:** logic (FSM, PID, AI-based controllers).
- **Application layer:** dashboards, mobile apps, cloud APIs.

11.3 The CIA Triad for CPS

Confidentiality. Ensure only authorized entities access data.

Integrity. Prevent unauthorized modification of data or commands.

Availability. Guarantee timely access to CPS functions even under attack.

11.4 Security Principles for CPS

- Defense in depth: multiple layers of protection.
- Principle of least privilege: minimal access rights.
- Fail-safe defaults: system defaults to safe mode under failure.
- Security by design: integrate from the beginning, not as afterthought.

11.5 Privacy in CPS

- Data minimization: collect only necessary data.
- Anonymization: remove identifiers from datasets.
- Consent and governance: respect legal/ethical data usage.

11.6 Case Study: Stuxnet

Stuxnet demonstrated the potential for cyber attacks on physical systems:

- Targeted PLCs in industrial control systems.
- Modified actuator behavior while masking malicious commands.
- Highlighted the need for monitoring both cyber and physical anomalies.

11.7 Defenses and Mitigations

- Strong authentication and encryption (TLS, certificates).
- Secure firmware updates with signatures.
- Intrusion detection systems (IDS).
- Redundancy and diversity (e.g., different sensors for same parameter).

11.8 Node-RED Lab 11: Intrusion Detection with Anomaly Detection

Objectives

- Implement an anomaly detection system for CPS data streams.
- Flag abnormal sensor values or communication patterns.
- Visualize alerts on Node-RED dashboard.

Required Hardware/Software

- ESP32 with sensor (temperature or simulated input).
- Node-RED with dashboard and MQTT broker.

Procedure

1. ESP32 publishes sensor values to `/cps/secure/temp`.
2. Node-RED subscribes and applies anomaly detection rule:

```
let t = Number(msg.payload.t);
if (t < 0 || t > 50) {
    msg.alert = "ANOMALY: out-of-range!";
} else {
    msg.alert = "Normal";
}
return msg;
```

3. Display sensor values and alerts on dashboard (gauge + text).

Expected Output

If sensor value exceeds threshold (e.g., 50°C), Node-RED dashboard shows an "ANOMALY" alert in red.

Assessment Rubric (10 points)

2 ESP32 publishes valid data.

3 Node-RED anomaly detection logic implemented.

3 Dashboard displays alerts correctly.

2 Screenshots/documentation provided.

Starter Node-RED Flow (Importable JSON)

```
[{"id":"tab11","type":"tab","label":"Lab11 - IDS"},
{"id":"broker11","type":"mqtt-broker","name":"LocalBroker","broker":"localhost","port
{"id":"mqin11","type":"mqtt in","z":"tab11","name":"Temp in",
"topic":"/cps/secure/temp","broker":"broker11","datatype":"auto","x":150,"y":120,"wir
{"id":"fn11","type":"function","z":"tab11","name":"Anomaly Detection",
"func":"let t=Number(msg.payload.t);\nif(t<0||t>50){msg.alert=\"ANOMALY\";}else{msg.a
"x":360,"y":120,"wires":[["gauge11","uiText11"]]},
{"id":"gauge11","type":"ui_gauge","z":"tab11","group":"uigrp11","order":1,
"title":"Temperature","label":"°C","format":"{{msg.payload.t}}","min":0,"max":"60",
"x":560,"y":100,"wires":[]},
```

```
{"id":"uiText11","type":"ui_text","z":"tab11","group":"uigrp11","order":2,
"label":"Alert","format":"{{msg.alert}}","x":560,"y":150,"wires":[]},
{"id":"uigrp11","type":"ui_group","name":"Lab11","tab":"uitab11","order":1,"disp":tru
{"id":"uitab11","type":"ui_tab","name":"CPS Labs","icon":"dashboard"}]
```

Starter ESP32 Code (Arduino)

```
#include <WiFi.h>
#include <PubSubClient.h>

const char* WIFI_SSID="YOUR_WIFI";
const char* WIFI_PASS="YOUR_PASS";
const char* MQTT_HOST="192.168.1.10";
const int MQTT_PORT=1883;

WiFiClient espClient;
PubSubClient mqtt(espClient);

void wifiConnect(){ WiFi.begin(WIFI_SSID,WIFI_PASS); while(WiFi.status()!=WL_CONNECTE
void mqttConnect(){ mqtt.setServer(MQTT_HOST,MQTT_PORT); while(!mqtt.connected()){ mq

float readTemp(){ return 20.0 + random(-10,40); } // simulates anomalies

void setup(){
  Serial.begin(115200);
  wifiConnect();
  mqttConnect();
}

void loop(){
  if(WiFi.status()!=WL_CONNECTED) wifiConnect();
  if(!mqtt.connected()) mqttConnect();

  float t=readTemp();
  char buf[64];
  snprintf(buf,sizeof(buf),"{\"t\":%.2f}",t);
  mqtt.publish("/cps/secure/temp",buf);

  mqtt.loop();
```

```
  delay(1000);
}
```

Summary

Security and privacy are essential for CPS safety and trustworthiness. Attack surfaces span devices, networks, and applications, requiring layered defenses. Privacy must be respected through governance and anonymization. Lab 11.8 demonstrates anomaly detection in Node-RED, an entry point to intrusion detection for CPS.

Review Questions

1. What are the major CPS attack surfaces?
2. Explain the CIA triad in the context of CPS.
3. Why is anomaly detection important in CPS security?
4. How can privacy-preserving design protect sensitive CPS data?

Exercises

1. Extend Lab 11.8 with a dashboard alarm (red indicator) that persists until reset.
2. Implement role-based access control (RBAC) for Node-RED dashboard users.
3. Research and present a recent cyber-physical attack (e.g., on power grids or vehicles).

Further Reading

- C. Alcaraz and S. Zeadally, "Critical Infrastructure Protection: Requirements and Challenges for CPS," IEEE Communications Magazine, 2015.
- Y. Mo et al., "Cyber-Physical Security of a Smart Grid Infrastructure," Proc. IEEE, 2012.
- National Institute of Standards and Technology (NIST), *Guide to Industrial Control Systems (ICS) Security*, NIST SP 800-82.
- E. A. Lee, "The Past, Present and Future of Cyber-Physical Systems: Security at the Forefront," 2015.

Chapter 12

CPS Case Studies and Future Trends

Chapter Overview

Cyber-Physical Systems (CPS) have already transformed critical industries and infrastructures worldwide. This chapter presents real-world case studies in healthcare, transportation, smart cities, and Industry 4.0, illustrating the diversity and impact of CPS. It also explores future research directions, including digital twins, AI integration, quantum CPS, and sustainability. The final lab challenges students to design a capstone CPS prototype that integrates sensing, networking, control, actuation, and dashboards in Node-RED.

Learning Outcomes

After completing this chapter, students will be able to:

1. Analyze real-world CPS case studies in multiple domains.
2. Identify key lessons and challenges from CPS deployments.
3. Discuss emerging CPS technologies and research directions.
4. Design and prototype an integrated CPS project as a capstone exercise.

Key Terms

Smart city, Industry 4.0, intelligent transportation system (ITS), digital twin, edge AI, quantum CPS, sustainability.

12.1 Case Studies in CPS

12.1.1 Healthcare CPS: Remote Patient Monitoring

- Wearable sensors monitor vital signs (ECG, heart rate, blood oxygen).
- Data transmitted via IoT gateways to hospital cloud servers.
- Doctors access real-time dashboards and alerts.
- Challenges: data privacy (HIPAA/GDPR), energy-efficient wearables.

12.1.2 Transportation CPS: Intelligent Traffic Systems

- Cameras and inductive loops detect vehicle flow.
- Adaptive traffic signals reduce congestion.
- Vehicle-to-Infrastructure (V2I) communication improves safety.
- Challenges: latency, security against spoofed signals.

12.1.3 Smart City CPS: Urban Infrastructure

- Sensors monitor air quality, waste bins, streetlights.
- Actuators optimize energy usage and resource management.
- City dashboards integrate multiple subsystems for decision support.
- Challenges: interoperability, governance, large-scale deployment.

12.1.4 Industry 4.0 CPS: Smart Factories

- Machines equipped with sensors for predictive maintenance.
- OPC UA ensures interoperability across vendors.
- Digital twins optimize production workflows.
- Challenges: legacy system integration, cybersecurity.

12.2 Future Trends in CPS

Digital Twins. Real-time replicas of physical systems used for predictive analytics, fault detection, and optimization.

Edge AI. Deploying AI models at the edge (on microcontrollers, FPGAs) for real-time decision-making without cloud latency.

Quantum CPS. Future CPS may exploit quantum communication and computation for ultra-secure and high-performance systems.

Sustainability. Green CPS architectures focus on energy efficiency, renewable integration, and circular economy principles.

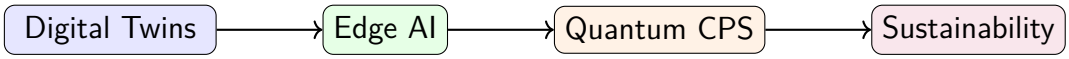

Figure 12.1: Future directions in CPS research and applications.

12.3 Node-RED Lab 12: Capstone CPS Project

Objectives

- Design and prototype a full CPS system integrating sensing, networking, control, and actuation.
- Apply security and scalability considerations.
- Demonstrate results through Node-RED dashboard visualization.

Required Hardware/Software

- ESP32 boards (1–3 units) with chosen sensors and actuators.
- Node-RED with MQTT broker and dashboard.
- Optional: database (InfluxDB) and visualization (Grafana).

Procedure

1. Define CPS application domain (e.g., smart home, healthcare, traffic system).
2. Select appropriate sensors and actuators for the domain.
3. Program ESP32 to publish sensor data and subscribe to control commands.
4. In Node-RED:
    - Integrate multiple MQTT topics.

- Implement control logic (rule-based, FSM, or PID).
- Create dashboard with gauges, charts, switches, and alerts.

5. Present prototype, including design rationale, system diagram, and results.

Expected Output

A working CPS prototype tailored to a chosen domain, visualized on Node-RED dashboard, with end-to-end functionality from sensing to actuation.

Assessment Rubric (20 points)

5 Clear problem definition and design rationale.

5 Correct integration of sensing, networking, control, and actuation.

5 Dashboard visualization and usability.

5 Innovation and scalability considerations.

Summary

This chapter presented real-world CPS case studies across multiple domains, highlighting lessons learned from deployments. It also discussed future trends: digital twins, edge AI, quantum CPS, and sustainability. Lab 12.3 challenges students to design their own CPS prototypes, synthesizing knowledge from the entire course.

Review Questions

1. Describe one CPS case study in healthcare and its main challenges.
2. What role do digital twins play in Industry 4.0 CPS?
3. Why is edge AI important for CPS?
4. How can CPS contribute to sustainability goals?

Exercises

1. Choose a CPS domain (e.g., smart home, smart grid, healthcare) and propose a new CPS solution. Draw its layered architecture.
2. Extend Lab 12.3 by adding anomaly detection for security monitoring.

3. Write a short report comparing two future CPS technologies (e.g., edge AI vs quantum CPS).

Further Reading

- R. Rajkumar et al., "Cyber-Physical Systems: The Next Computing Revolution," IEEE DAC, 2010.
- E. A. Lee, "The Past, Present and Future of Cyber-Physical Systems," Proc. IEEE, 2015.
- L. Monostori, "Cyber-Physical Production Systems: Roots and Challenges," CIRP Annals, 2014.
- M. Wollschlaeger et al., "The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things," IEEE, 2017.