SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES

(AUTONOMOUS)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

IV B.Tech – VII Semester

23CSE354C – INTERNET OF THINGS

LECTURE NOTES

(Units I – V)

Units Covered

UNIT I: INTRODUCTION TO IoT

UNIT II: PROTOTYPING IoT OBJECTS USING MICROPROCESSOR / MICROCONTROLLER

UNIT III: IoT ARCHITECTURE AND PROTOCOLS

UNIT IV: DEVICE DISCOVERY AND CLOUD SERVICES FOR IoT

UNIT V: UAV IoT

Prepared By

V.P.Manikandan

UNIT – I: INTRODUCTION TO IoT

Definition and Characteristics of IoT | Physical Design of IoT | IoT Protocols | IoT Communication Models | IoT Communication APIs | Communication Protocols | Embedded Systems | IoT Levels and Templates

Contents

UNIT – I: INTRODUCTION TO IoT

1.1 Introduction

The Internet of Things (IoT) is one of the most significant technological revolutions of the current era, extending the reach of the Internet beyond traditional computing devices such as desktops, laptops, and smartphones to everyday physical objects. These objects — commonly referred to as "things" — are embedded with sensors, actuators, processing ability, software, and network connectivity that enable them to collect, exchange, and act upon data. IoT connects the physical world with the digital world, allowing devices to sense their environment, communicate with each other, and make intelligent decisions with minimal human intervention.

The scope of IoT spans across numerous domains including smart homes, smart cities, healthcare, agriculture, industrial automation, transportation, and environmental monitoring. As the number of connected devices continues to grow into billions, understanding the fundamental building blocks of IoT — its architecture, protocols, communication models, and levels of deployment — becomes essential for designing efficient and scalable IoT solutions. This unit introduces these foundational concepts.

1.2 Definition and Characteristics of IoT

1.2.1 Definition

The Internet of Things can be defined as a network of physical objects ("things") embedded with sensors, software, and other technologies that enables these objects to collect and exchange data over the Internet without requiring human-to-human or human-to-computer interaction. Each "thing" is uniquely identifiable through its embedded computing system and is able to interoperate within the existing Internet infrastructure.

In simple terms, IoT can be summarized as: Things (physical devices) + Internet (connectivity) + Data (sensing, processing, and communication) = A smart, interconnected ecosystem capable of autonomous decision-making.

1.2.2 Characteristics of IoT

The key characteristics that distinguish an IoT system from a conventional networked system are described below:

- **Interconnectivity:** Anything can be interconnected with the global information and communication infrastructure.
- **Things-related services:** IoT is capable of providing thing-related services within the constraints of things, such as privacy protection and semantic consistency between physical things and their associated virtual representation.
- **Heterogeneity:** IoT devices are heterogeneous, built on different hardware platforms and networks; they interact through different networks and are able to interoperate across these networks.
- **Dynamic changes:** The state of devices changes dynamically — for example sleeping/waking up, connected/disconnected — and the number of devices can change dynamically too.
- **Enormous scale:** The number of devices that need to be managed and that communicate with each other is much larger than the devices connected to the current Internet.
- **Safety:** As we gain benefits from IoT, we must not forget about safety — as both the personal safety and the safety of our data need to be secured.

- **Connectivity:** Enables network accessibility and compatibility — accessibility is provided through network availability, while compatibility provides a common ability to consume and produce data.
- **Sensing:** IoT devices detect or measure changes in the surrounding environment using sensors and report the same to give a true picture of the physical world.
- **Intelligence:** The extraction of knowledge from generated data is very important — sensors generate data which must be analysed and processed to extract meaningful information.
- **Energy:** Since most IoT devices are battery-powered or energy-harvesting, energy efficiency in sensing, computation and communication is a critical design characteristic.

1.3 Physical Design of IoT

The physical design of IoT refers to the actual hardware components — the individual node level devices ('things'), such as sensors, actuators, and processing units, along with the protocols used for communication between them. A "Thing" in IoT can be any object equipped with sensors that is capable of collecting and transferring data over a network without manual intervention.

1.3.1 Generic Block Diagram of an IoT System

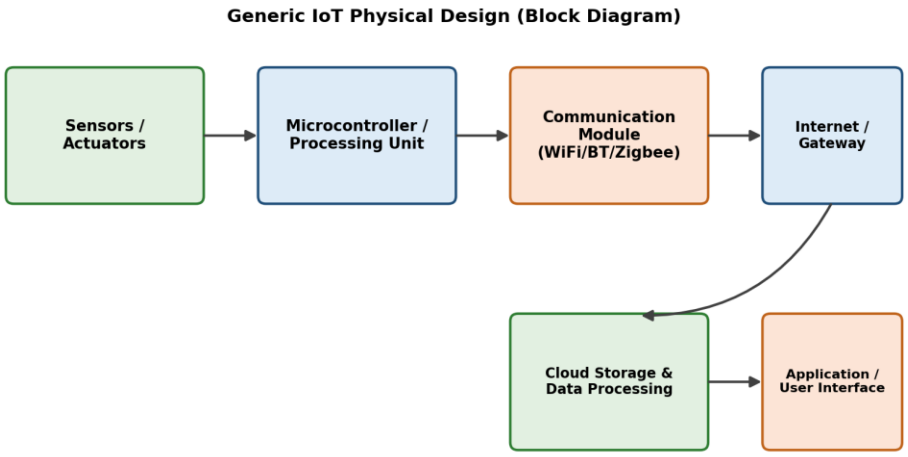

Fig. 1.1 — Generic Physical Design (Block Diagram) of an IoT System

As illustrated in Fig. 1.1, an IoT system typically comprises: (i) sensors/actuators that interact with the physical environment, (ii) a microcontroller or processing unit that reads sensor data and controls actuators, (iii) a communication module that transmits data using short-range or long-range wireless technologies, (iv) a gateway or Internet connection that bridges the local network to the wider Internet, and (v) cloud-based storage and processing along with an application/user-interface layer that presents actionable information to the end-user.

1.3.2 Things in IoT

A 'thing' in IoT could be a person with a heart-monitoring implant, a farm animal with a biochip transponder, an automobile with built-in sensors, or any object assigned an IP address that is provided with the ability to transfer data over a network. Things can be broadly classified as:

- Sensing things — collect data about physical parameters (temperature, humidity, motion, etc.)
- Actuating things — perform a physical action based on control signals (motors, relays, valves, etc.)
- Hybrid things — possess both sensing and actuating capability.

1.3.3 IoT Protocols — Layered Classification

IoT protocols define the physical design and are classified according to the layer they operate in, similar to the layered TCP/IP model. Table 1.1 summarizes protocols according to their functional layer; each is discussed in more detail in Section 1.4.

Layer	Function	Example Protocols
Link Layer	Physical addressing and framing of data over a physical medium	802.3 – Ethernet, 802.11 – WiFi, 802.16 – WiMax, 802.15.4 – LR-WPAN, 2G/3G/4G
Network Layer	Sending IP datagrams from source to destination and routing	IPv4, IPv6, 6LoWPAN
Transport Layer	End-to-end message transfer, independent of underlying network	TCP, UDP
Application Layer	Data formats exchanged between applications	HTTP, CoAP, WebSocket, MQTT, XMPP, DDS, AMQP

1.4 IoT Protocols

IoT Protocols enable devices, gateways, and servers to exchange information reliably despite the constraints of low power, low bandwidth, and lossy links typical of IoT deployments. The major protocols, organized layer-wise, are described below.

1.4.1 Link Layer Protocols

- 802.3 – Ethernet: Defines the physical layer and the MAC sub-layer of the data link layer for wired LANs.
- 802.11 – WiFi: Defines the standard for implementing wireless LAN in the 2.4 GHz, 5 GHz, and 60 GHz frequency bands.
- 802.16 – WiMax: Defines the standard for Wireless Broadband Access (WBA), providing high data rates over long distances.
- 802.15.4 – LR-WPAN: Defines the standard for Low-Rate Wireless Personal Area Networks, the basis for higher-layer protocols such as ZigBee and 6LoWPAN.
- 2G/3G/4G – Mobile Communication: Provides Internet access through the cellular network for devices requiring wide-area connectivity.

1.4.2 Network/Internet Layer Protocols

- IPv4: 32-bit addressing scheme, allowing about 4.3 billion unique addresses; still widely used but insufficient for the scale of IoT.
- IPv6: 128-bit addressing scheme providing an enormous address space (2^128 addresses), essential given the scale of IoT deployments.
- 6LoWPAN: Enables the use of IPv6 over low-power, low-rate wireless networks such as IEEE 802.15.4, by compressing IPv6 headers to suit constrained devices.

1.4.3 Transport Layer Protocols

- TCP (Transmission Control Protocol): A connection-oriented, reliable protocol that guarantees delivery, ordering and error-checking, at the cost of higher overhead.
- UDP (User Datagram Protocol): A connectionless, lightweight protocol without delivery guarantees, preferred for constrained IoT devices where speed matters more than reliability.

1.4.4 Application Layer Protocols

Protocol	Transport	Key Feature
HTTP	TCP	Request-response, widely used, but relatively heavy for constrained devices
CoAP	UDP	Constrained Application Protocol; lightweight REST-like protocol for constrained nodes
WebSocket	TCP	Full-duplex communication over a single, long-lived TCP connection
MQTT	TCP	Lightweight publish-subscribe protocol; ideal for low-bandwidth, high-latency networks
XMPP	TCP	Extensible Messaging and Presence Protocol; supports near real-time messaging
DDS	TCP/UDP	Data Distribution Service; high performance publish-subscribe protocol for M2M
AMQP	TCP	Advanced Message Queuing Protocol; supports point-to-point and publish-subscribe messaging

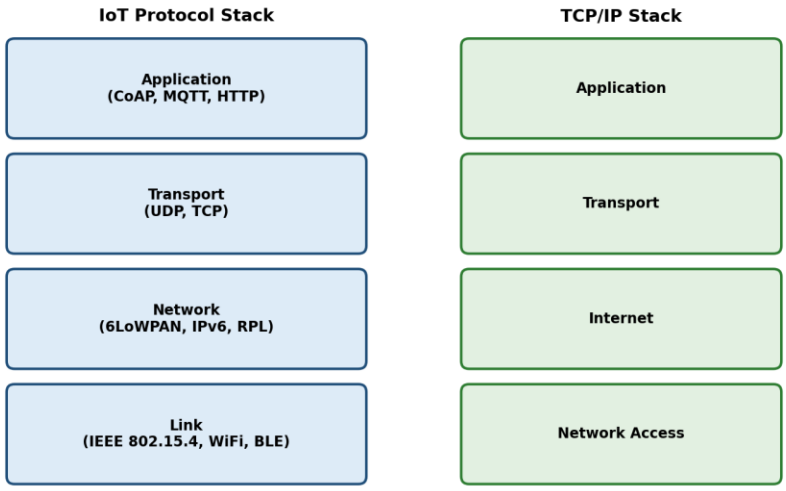

Fig. 1.2 — IoT Protocol Stack compared with the traditional TCP/IP Stack

1.5 IoT Communication Models

The Internet Architecture Board (IAB) identifies four common communication models used by IoT devices: Request-Response, Publish-Subscribe, Push-Pull, and Exemplar. Understanding these models helps in designing efficient and scalable communication for IoT applications.

1.5.1 Request-Response Model

In this model, the client sends requests to the server, and the server responds to the requests. When the server receives a request, it decides how to respond, retrieves the requested data, prepares a response, and sends it back to the client. This is a stateless model — each request-response pair is independent of the previous one.

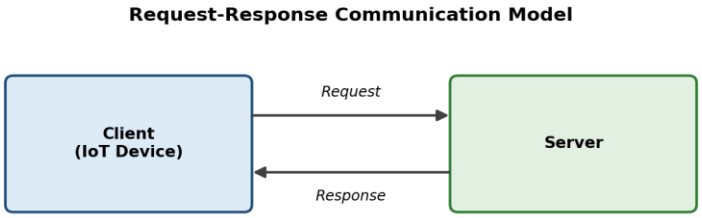

Fig. 1.3 — Request-Response Communication Model

1.5.2 Publish-Subscribe Model

This model involves publishers, brokers, and consumers (subscribers). Publishers are the source of data and send data to a topic managed by the broker; they are unaware of the consumers. Subscribers subscribe to topics of interest, and the broker forwards messages from publishers to all the appropriate subscribers. This model decouples publishers from subscribers, improving scalability.

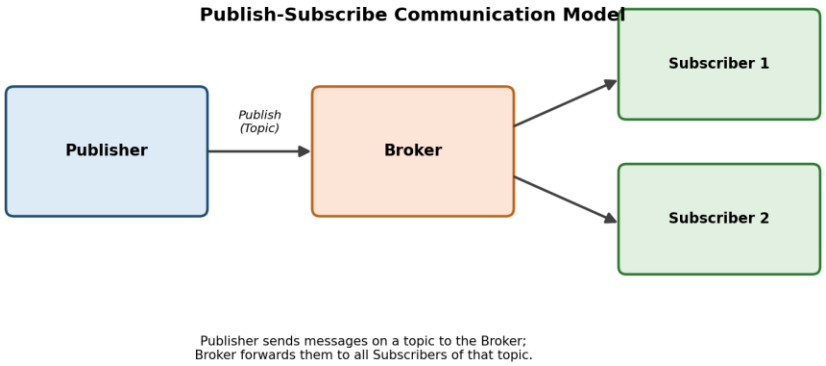

Fig. 1.4 — Publish-Subscribe Communication Model

1.5.3 Push-Pull Model

In the push-pull model, data producers push data into a queue, and consumers pull data from the queue. Queues act as a buffer, helping mediate issues that arise from data producers and consumers operating at different rates or being available at different times.

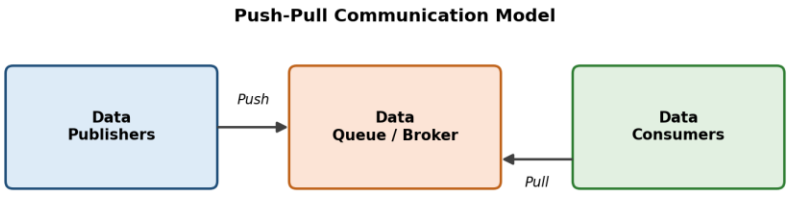

Fig. 1.5 — Push-Pull Communication Model

1.5.4 Exemplar Model

The Exemplar model represents a real-world combination of the above models — for example, an IoT device (sensor node) communicates with a REST-based web service which in turn interacts with a cloud backend; the results are then consumed by mobile or web applications. This model demonstrates end-to-end communication from constrained devices to end-user applications.

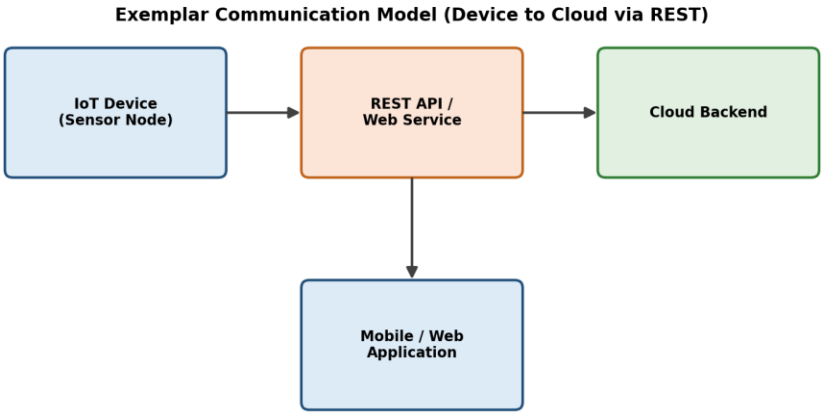

Fig. 1.6 — Exemplar Communication Model (Device-to-Cloud via REST)

1.6 IoT Communication APIs

IoT Communication APIs define how applications interact with IoT devices and services over a network. The two most widely used API styles for IoT communication models are REST-based and WebSocket-based APIs.

1.6.1 REST-based Communication APIs

Representational State Transfer (REST) is a set of architectural principles for designing loosely coupled applications that use the HTTP protocol for data communication. REST-based APIs follow the Request-Response communication model. In the REST architecture, a client sends a request to a server that hosts a resource, and the server responds with the current state of the resource.

Key principles of REST include:

- Resource-based: Everything is treated as a resource identified by a unique URI (Uniform Resource Identifier).
- Statelessness: Each request from a client contains all the information needed for the server to fulfil it; no client context is stored on the server between requests.
- Uniform interface: A fixed set of operations is used to manipulate resources (GET, POST, PUT, DELETE).
- Representation-oriented: A resource can have multiple representations, e.g., JSON, XML, or plain text.
- Cacheable: Responses must define themselves as cacheable or non-cacheable to improve client-side performance.

HTTP Method	CRUD Operation	Description
GET	Read	Retrieve the current state/representation of a resource
POST	Create	Create a new resource / sub-resource
PUT	Update	Update an existing resource (full replace)
DELETE	Delete	Remove a resource

1.6.2 WebSocket-based Communication APIs

WebSocket APIs allow bi-directional, full-duplex communication between clients and servers over a single, long-lived TCP connection. Unlike the request-response model, once a WebSocket connection is established, the server can push data to the client without waiting for a request, which is particularly useful for real-time IoT applications such as live sensor dashboards, alerting systems and telemetry streaming.

Feature	REST API	WebSocket API
Communication Model	Request-Response	Full-duplex / Bi-directional

Feature	REST API	WebSocket API
Connection	New connection per request (or reused via keep-alive)	Single persistent connection
Overhead	Higher (headers per request)	Lower after handshake
Best suited for	CRUD operations, infrequent updates	Real-time telemetry, live dashboards

1.7 Communication Protocols (Short-Range Wireless Technologies)

In addition to the layered protocol stack, IoT devices commonly rely on short-range wireless communication technologies to connect sensors and actuators to gateways. The most important of these technologies are described below and compared in Table 1.4.

1.7.1 RFID (Radio Frequency Identification)

RFID uses electromagnetic fields to automatically identify and track tags attached to objects. An RFID system consists of a tag (transponder), a reader, and an antenna. RFID tags may be passive (powered by the reader's signal) or active (battery-powered).

1.7.2 NFC (Near Field Communication)

NFC is a short-range (up to ~4 cm), high-frequency wireless communication technology that enables simple and secure two-way interactions between electronic devices, widely used for contactless payments and access control.

1.7.3 Bluetooth / BLE

Bluetooth is a wireless technology standard for exchanging data over short distances using UHF radio waves. Bluetooth Low Energy (BLE) is a power-optimized variant designed specifically for applications that need to run for long periods on small batteries, common in wearables and fitness trackers.

1.7.4 ZigBee

ZigBee is a specification built on top of IEEE 802.15.4 for a suite of high-level communication protocols used to create personal area networks with low-power digital radios, suited for home automation and industrial control applications requiring a low data rate and long battery life.

1.7.5 WiFi

WiFi (IEEE 802.11) provides higher data rates than the other technologies discussed here and is widely deployed, but at the cost of higher power consumption, making it suitable for IoT nodes that are mains-powered or otherwise not severely energy-constrained.

Technology	Range	Data Rate	Power Usage	Typical Application
RFID	Up to 100 m (active)	Low	Very Low / Passive	Asset tracking, inventory
NFC	~4 cm	424 kbps	Very Low	Payments, access control
Bluetooth / BLE	10 – 100 m	1 – 3 Mbps	Low	Wearables, health monitors
ZigBee	10 – 100 m	250 kbps	Very Low	Home automation, sensors
WiFi	50 – 100 m	Up to 1+ Gbps	High	Smart cameras, high-bandwidth IoT

1.8 Embedded Systems

An embedded system is a combination of computer hardware and software — and often additional mechanical parts — designed to perform a dedicated function, either as an independent system or as part of a larger system. Embedded systems form the computational core of most IoT devices.

1.8.1 Characteristics of Embedded Systems

- Application/domain specific — designed for a particular task rather than general-purpose computing.
- Reactive and real-time — must respond to external events within strict timing constraints.
- Low power consumption — many embedded IoT devices run on batteries or energy harvesting.
- Compact and cost-effective — minimal hardware footprint tailored to the intended function.
- Connected — increasingly equipped with communication modules for IoT connectivity.

1.8.2 Components of an Embedded System

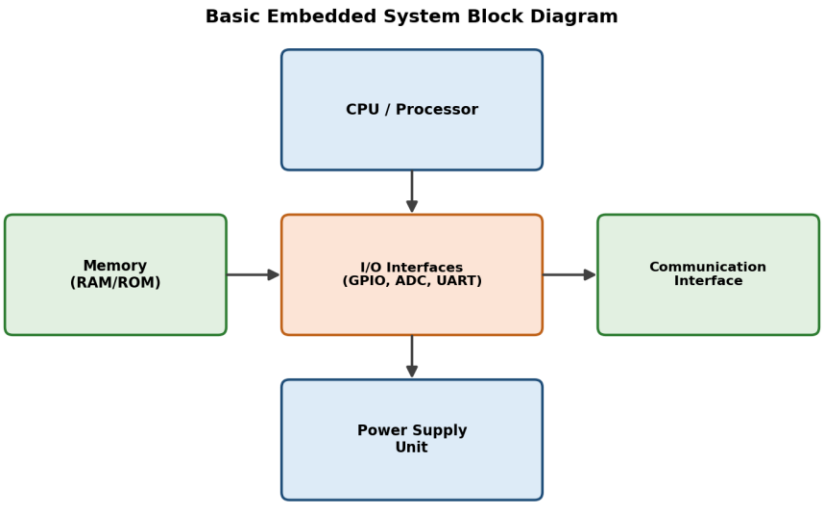

Fig. 1.7 — Basic Block Diagram of an Embedded System

As shown in Fig. 1.7, a typical embedded system consists of: a CPU/processor (microcontroller or microprocessor) that executes the control program; memory (RAM for runtime data, ROM/Flash for program storage); I/O interfaces (GPIO pins, ADC/DAC, UART, SPI, I2C) for interacting with sensors and actuators; a communication interface for network connectivity; and a power supply unit.

1.8.3 Types of Embedded Systems

- Stand-alone embedded systems — operate independently, taking input and producing output without a host computer (e.g., a digital thermostat).
- Real-time embedded systems — must complete a task within a specified time; classified as hard real-time (missing a deadline is catastrophic) or soft real-time (missing a deadline degrades quality but is tolerable).
- Networked embedded systems — connected to a network to provide output to attached devices, common in IoT deployments.
- Mobile embedded systems — small-sized systems designed for portability, such as digital cameras and wearables.

1.8.4 Popular IoT Prototyping Boards

Board	Processor Type	Key Feature
Arduino Uno	8-bit AVR Microcontroller	Simple, low-cost, large community support; ideal for beginners
Raspberry Pi	ARM-based Microprocessor (runs full OS)	Full Linux OS support; suited for computation-heavy IoT applications

Board	Processor Type	Key Feature
ESP8266 / ESP32	32-bit Microcontroller with built-in WiFi	Low-cost WiFi-enabled boards popular for connected IoT projects
NodeMCU	ESP8266-based development board	Lua/Arduino IDE programmable, integrated WiFi module

1.9 IoT Levels and Templates

An IoT system comprises several building blocks such as sensors/actuators, resources, controller service, database, web service, analysis component, and application. An IoT level is defined based on the number of components and how they are arranged in an IoT system. IoT levels and their associated deployment templates help system designers select the right architecture for a given application. The commonly discussed IoT levels are described below.

1.9.1 IoT Level-1

A Level-1 IoT system has a single node/device that performs sensing and/or actuation, stores data, performs analysis, and hosts the application. This is suitable for low-cost, low-complexity solutions where the data involved is not big and the requirements are not computationally intensive, for example a home automation system that turns on/off an appliance based on a sensor reading.

1.9.2 IoT Level-2

A Level-2 IoT system has a single node that performs sensing/actuation and local data storage, but the data analysis and storage are hosted on the cloud, enabling more efficient and faster processing of larger data volumes. Level-2 systems are typically used for applications where data is significant, but the number of nodes is small.

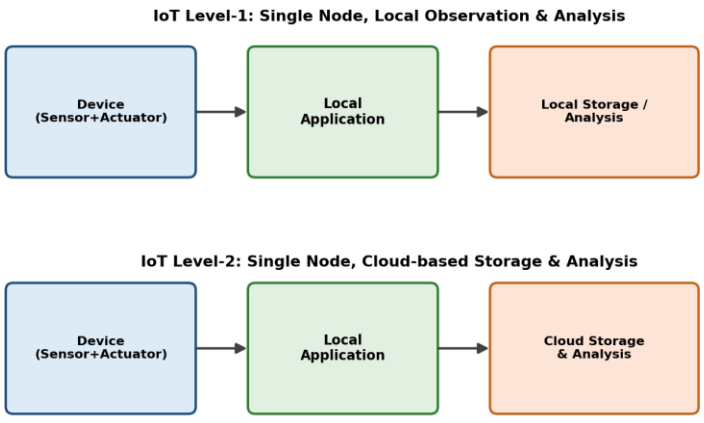

Fig. 1.8 — IoT Level-1 and IoT Level-2 Deployment Templates

1.9.3 IoT Level-3

A Level-3 IoT system has a single node that performs sensing/actuation and uses a local or cloud-based application, with data storage and analysis performed on the cloud. This level introduces web services for accessing and controlling the system remotely.

1.9.4 IoT Level-4

A Level-4 IoT system has multiple nodes performing local analysis. Data is collected from each node and stored/analysed locally or on the cloud. Level-4 contains local and cloud-based observer nodes which can subscribe to and receive information collected from the sensors in the IoT system.

1.9.5 IoT Level-5

A Level-5 IoT system consists of multiple end nodes and one coordinator node. The end nodes perform sensing and/or actuation, while the coordinator node collects data from the end nodes and sends it to the cloud for storage and analysis. This model is useful for applications where multiple sensor nodes need to be aggregated before transmission, reducing network overhead.

1.9.6 IoT Level-6

A Level-6 IoT system has multiple independent end nodes that perform sensing and/or actuation and send data directly to the cloud. The data collected is stored, and a big-data analytics component fetches the data and performs the analysis. This level is well-suited for large-scale deployments requiring high scalability, such as smart city sensor networks.

Level	Nodes	Storage & Analysis	Typical Use Case
1	Single node	Local	Simple home automation switch
2	Single node	Cloud	Weather monitoring station
3	Single node + web service	Cloud	Remotely accessible smart device
4	Multiple nodes	Local + Cloud (observer nodes)	Noise/traffic monitoring
5	Multiple end nodes + coordinator	Cloud	Forest fire detection network
6	Multiple independent nodes	Cloud (Big Data Analytics)	Smart city sensor networks

1.10 Important Formulae

The following formulae are commonly used while analysing sensing, sampling, transmission and power aspects of IoT systems. Each formula can be copied into calculations directly.

1.10.1 Nyquist Sampling Theorem

To accurately reconstruct an analog signal from its samples without aliasing, the sampling frequency must be at least twice the maximum frequency component present in the signal:

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

where f_s = sampling frequency, f_{max} = maximum signal frequency

1.10.2 Shannon–Hartley Channel Capacity

The theoretical maximum data rate (channel capacity) of a communication channel in the presence of noise is given by:

$C = B \times \log_2 (1 + S/N)$

where C = channel capacity (bps), B = bandwidth (Hz), S/N = signal-to-noise ratio

1.10.3 Data Rate / Throughput

The effective data rate transmitted over a given period is:

Data Rate (bps) = Total Bits Transmitted / Total Time (s)

1.10.4 Battery Life Estimation

A commonly used approximation to estimate how long a battery-powered IoT node will operate:

Battery Life (hours) = Battery Capacity (mAh) / Average Current Consumption (mA)

1.10.5 Duty Cycle

Many IoT sensor nodes alternate between active and sleep states to conserve power. The duty cycle is defined as:

Duty Cycle (%) = [Ton / (Ton + Toff)] × 100

where Ton = active time, Toff = sleep time

1.10.6 Power Consumption

The instantaneous electrical power consumed by a device is the product of voltage and current:

P = V × I

where P = power (Watts), V = voltage (Volts), I = current (Amperes)

1.10.7 Energy Consumed

The total energy consumed by a device over a period of operation:

E = P × t

where E = energy (Joules or Wh), P = power (W), t = time (s or h)

1.10.8 Free Space Path Loss (FSPL)

Used to estimate signal attenuation over a distance in wireless IoT links:

FSPL (dB) = 20 log10(d) + 20 log10(f) + 20 log10(4π/c)

where d = distance between transmitter and receiver, f = frequency, c = speed of light

1.10.9 Illustrative Numerical Example

Consider a battery-powered temperature sensor node with a battery capacity of 2000 mAh. The node draws 15 mA while active and 0.05 mA while in sleep mode. If the node operates with a duty cycle of 10% (i.e., it stays active for 10% of the time and sleeps for the remaining 90%), estimate its average current consumption and expected battery life.

Average Current = (Duty Cycle x Iactive) + [(1 - Duty Cycle) x Isleep]

= (0.10 x 15) + (0.90 x 0.05) = 1.5 + 0.045 = 1.545 mA

Battery Life = Battery Capacity / Average Current = 2000 / 1.545 = 1294.5 hours (approx. 54 days)

This example demonstrates why duty-cycling is a critical design technique in IoT sensor nodes: by keeping the radio and processor in sleep mode for most of the time, battery life can be extended from a few hours (if always active) to several weeks or months.

1.11 Summary

- IoT connects everyday physical objects to the Internet, enabling them to sense, communicate, and act with minimal human intervention.
- Key characteristics of IoT include interconnectivity, heterogeneity, dynamic changes, enormous scale, safety, connectivity, sensing, intelligence, and energy efficiency.
- The physical design of IoT covers the 'things' (sensors/actuators) and the layered protocols (Link, Network, Transport, Application) used for communication.
- Four communication models are used in IoT: Request-Response, Publish-Subscribe, Push-Pull, and Exemplar.

- REST-based and WebSocket-based APIs are the two major styles of IoT communication APIs.
- Short-range wireless technologies such as RFID, NFC, Bluetooth/BLE, ZigBee, and WiFi enable device-to-device and device-to-gateway communication.
- Embedded systems form the computational core of IoT devices, characterized by dedicated functionality, real-time operation, and low power consumption.
- IoT systems are classified into six levels (Level 1 to Level 6) based on the number of nodes and where storage/analysis is performed (locally or on the cloud).

1.12 Review Questions

Short Answer Questions

1. Define the Internet of Things.
2. List any five characteristics of IoT.
3. What is a 'thing' in the context of IoT? Give two examples.
4. Name the four layers used to classify IoT protocols.
5. What is the difference between TCP and UDP?
6. Define REST and list its key principles.
7. What is the difference between a REST API and a WebSocket API?
8. What is an embedded system? List its characteristics.
9. What is duty cycle and why is it important in IoT nodes?
10. List the six IoT levels.

Long Answer Questions

11. Explain the physical design of IoT with a neat block diagram.
12. Describe the four IoT communication models with diagrams, highlighting their advantages and disadvantages.
13. Compare REST-based and WebSocket-based communication APIs in detail.
14. Explain any five short-range wireless communication technologies used in IoT, with a comparison table.
15. Describe the components of an embedded system with a block diagram.
16. Explain IoT Levels 1 through 6 with suitable examples and diagrams.
17. Derive/explain the Shannon-Hartley theorem and its significance in IoT communication.
18. Explain the layered classification of IoT protocols with examples of protocols at each layer.

References

- Vijay Madisetti and Arshdeep Bahga, "Internet of Things (A Hands-on Approach)", 1st Edition, VPT, 2014.
- Jan Holler et al., "From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence", 1st Edition, Academic Press, 2014.
- Pethuru Raj, Anupama C. Raman, "The Internet of Things, Enabling Technologies and Use Cases", CRC Press.
- Cuno Pfister, "Getting Started with the Internet of Things", O'Reilly Media, 2011.

UNIT – II: PROTOTYPING IoT OBJECTS USING

MICROPROCESSOR / MICROCONTROLLER

Working Principles of Sensors and Actuators | Setting up the Board |

Programming for IoT | Reading from Sensors |

Communication through Bluetooth and Wi-Fi

Contents

UNIT – II: PROTOTYPING IoT OBJECTS USING MICROPROCESSOR / MICROCONTROLLER

2.1 Introduction

Prototyping is the process of building a working model of an IoT solution to validate design ideas before full-scale development. At the heart of every IoT prototype lies a microprocessor or microcontroller board that reads data from sensors, processes it, and controls actuators — while also communicating with other devices or the cloud. This unit covers the working principles of sensors and actuators, the process of setting up a development board, writing programs for IoT applications, reading data from sensors, and establishing communication using Bluetooth and Wi-Fi.

A typical prototyping workflow involves: selecting an appropriate board (Arduino, Raspberry Pi, NodeMCU, etc.), connecting sensors and actuators to the board, writing and uploading firmware/code, testing sensor readings, and finally enabling wireless communication so that the device can share data with other systems or the Internet.

2.2 Sensors — Working Principles and Characteristics

2.2.1 Definition

A sensor is a device that detects and responds to changes in a physical quantity such as temperature, pressure, light, humidity, or motion, and converts this physical phenomenon into a measurable electrical signal (usually voltage, current, or resistance) that can be read by a microcontroller.

2.2.2 Sensor Measurement Chain

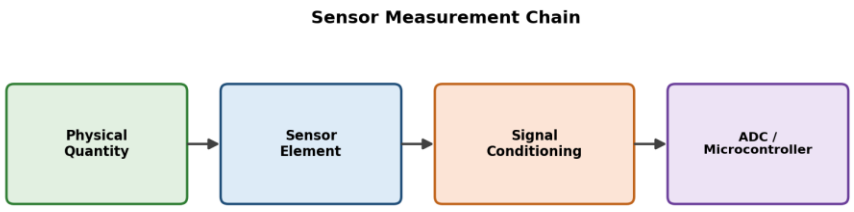

Fig. 2.1 — Sensor Measurement Chain: Physical Quantity to Digital Value

As shown in Fig. 2.1, a physical quantity is first detected by the sensing element, which produces a raw electrical signal. This signal is often weak or noisy, so it passes through a signal-conditioning stage (amplification, filtering, linearization). The conditioned analog signal is then converted to a digital value by an Analog-to-Digital Converter (ADC) before being processed by the microcontroller.

2.2.3 Classification of Sensors

- **Based on Output:** Analog sensors (produce a continuous signal, e.g., LDR, thermistor) and Digital sensors (produce discrete signals, e.g., DHT11, PIR).
- **Based on Power Requirement:** Active sensors (require external power to operate, e.g., ultrasonic sensor) and Passive sensors (generate signal from the measured quantity itself, e.g., thermocouple).
- **Based on Quantity Measured:** Temperature (LM35, DHT11), Light (LDR), Proximity (IR, Ultrasonic), Motion (PIR, Accelerometer), Gas (MQ series), Humidity (DHT22), and Pressure sensors (BMP180).

2.2.4 Characteristics / Performance Parameters of Sensors

- **Sensitivity:** The ratio of change in output to change in the measured input quantity.
- **Resolution:** The smallest change in input that a sensor can detect and represent in its output.
- **Accuracy:** How close the measured value is to the true/actual value of the quantity being measured.
- **Precision:** The degree of reproducibility of a measurement — how close repeated measurements are to each other.
- **Response Time:** The time taken by the sensor to respond to a change in the input quantity and reach a stable output.
- **Linearity:** The extent to which the sensor's output varies in direct proportion to the input.
- **Calibration:** The process of comparing the sensor output with a known reference standard and adjusting it to minimize measurement error.

2.2.5 Commonly Used Sensors in IoT Prototyping

Sensor	Quantity Measured	Output Type	Typical Application
LM35	Temperature	Analog	Weather station, HVAC control
DHT11 / DHT22	Temperature & Humidity	Digital	Smart agriculture, indoor climate monitoring
LDR (Light Dependent Resistor)	Light Intensity	Analog	Automatic street lighting
PIR Sensor	Motion / Infrared radiation	Digital	Intruder/motion detection, smart lighting
Ultrasonic (HC-SR04)	Distance / Proximity	Digital (pulse width)	Obstacle detection, parking assist
MQ-2 / MQ-135	Gas concentration	Analog	Gas leak detection, air quality monitoring
BMP180 / BMP280	Atmospheric Pressure	Digital (I2C)	Weather forecasting,

Sensor	Quantity Measured	Output Type	Typical Application
			altitude estimation

2.3 Actuators — Working Principles and Types

2.3.1 Definition

An actuator is a device that converts an electrical control signal from a microcontroller into physical motion or action — such as rotating a motor, opening a valve, or switching on a light. While sensors allow a system to perceive its environment, actuators allow it to act upon that environment, closing the sense-decide-act loop that defines an intelligent IoT system.

2.3.2 Classification of Actuators

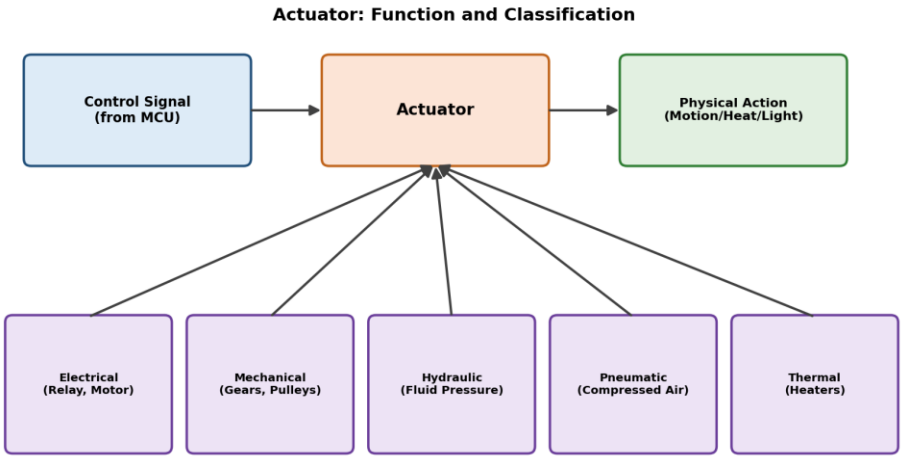

Fig. 2.2 — Actuator Function and Classification

- **Electrical Actuators:** Operate using electrical energy — examples include DC motors, servo motors, stepper motors, solenoids, and relays.
- **Mechanical Actuators:** Convert one form of motion into another using gears, pulleys, chains, and rack-and-pinion mechanisms.
- **Hydraulic Actuators:** Use pressurized fluid to generate force/motion; capable of producing very high forces, used in heavy industrial equipment.
- **Pneumatic Actuators:** Use compressed air to produce motion; fast-acting and commonly used in factory automation.
- **Thermal / Magnetic Actuators:** Respond to changes in temperature or magnetic fields, e.g., bimetallic strips in thermostats.

2.3.3 Commonly Used Actuators in IoT Prototyping

Actuator	Type	Control Signal	Typical Application
Relay Module	Electrical (switch)	Digital HIGH/LOW	Switching mains appliances ON/OFF
DC Motor	Electrical	PWM voltage	Fans, wheels in robotic IoT devices
Servo Motor	Electrical	PWM (angle control)	Camera positioning, robotic arms
Solenoid Valve	Electro-mechanical	Digital HIGH/LOW	Automatic water valves, irrigation
Buzzer / LED	Electrical (indicator)	Digital / PWM	Alerts and visual/audio indication

2.4 Setting up the Board

Before any sensor reading or actuator control can be programmed, the development board itself must be set up correctly. This section describes the general procedure using the Arduino Uno as a representative microcontroller board, and the Raspberry Pi as a representative microprocessor (single-board computer).

2.4.1 Arduino Uno Board Layout

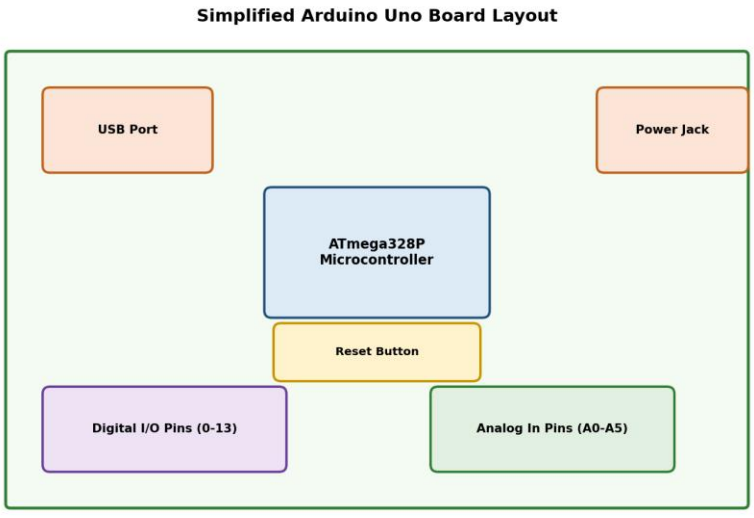

Fig. 2.3 — Simplified Arduino Uno Board Layout

The Arduino Uno is built around the ATmega328P microcontroller. Key components include the USB port (for programming and serial communication), the power jack (for external power supply), digital I/O pins 0-13 (some of which support PWM output), analog input pins A0-A5 (connected to the internal ADC), and a reset button.

2.4.2 Steps to Set Up an Arduino Board

- Install the Arduino IDE (Integrated Development Environment) from the official Arduino website on your computer.
- Connect the Arduino board to the computer using a USB cable.
- Open the Arduino IDE and select the correct board type from Tools → Board (e.g., Arduino Uno).
- Select the correct communication port from Tools → Port.
- Write or open a sketch (Arduino program) in the editor.
- Click the Verify button to compile the code and check for errors.
- Click the Upload button to transfer the compiled program to the board's microcontroller.
- Open the Serial Monitor (if required) to view sensor readings or debug messages in real time.

2.4.3 Steps to Set Up a Raspberry Pi

- Download and flash a Raspberry Pi OS image onto a microSD card using Raspberry Pi Imager or balenaEtcher.
- Insert the microSD card into the Raspberry Pi and connect a display, keyboard, and mouse (or configure headless access via SSH).
- Power on the Raspberry Pi and complete the initial OS setup (locale, WiFi, password).
- Update the package list and install required libraries, e.g., 'sudo apt update && sudo apt install python3-pip'.
- Enable required interfaces (GPIO, I2C, SPI, Camera) using 'sudo raspi-config'.
- Write Python (or other language) scripts to interface with sensors/actuators connected to the GPIO pins.

2.4.4 Arduino vs Raspberry Pi

Feature	Arduino (Microcontroller)	Raspberry Pi (Microprocessor)
Operating System	None (runs single program)	Full Linux-based OS
Processing Power	Low (8/16/32-bit MCU)	High (ARM Cortex multi-core CPU)
Power Consumption	Very Low	Higher
Programming Language	C / C++ (Arduino sketches)	Python, C++, Java, and more
Best suited for	Simple, real-time, single-task control	Computation-intensive, multi-tasking applications

2.5 Programming for IoT

Programming for IoT devices generally follows an event-driven, real-time model. For microcontroller boards like Arduino, code is structured around two mandatory functions: setup() (executed once at power-up/reset) and loop() (executed repeatedly). Fig. 2.4 illustrates this execution flow.

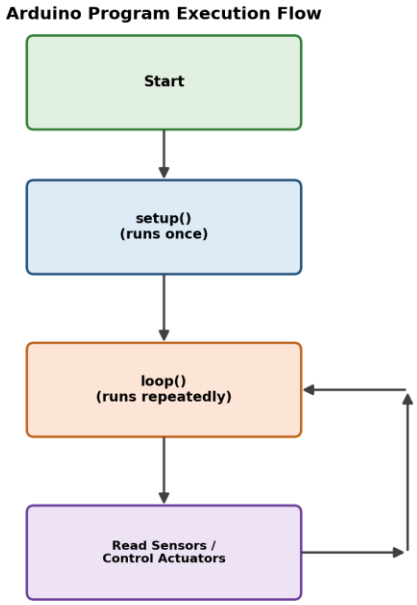

Fig. 2.4 — Arduino Program Execution Flow

2.5.1 Basic Structure of an Arduino Sketch

```
void setup() {
  // runs once: initialize pins, serial port, sensors
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
}

void loop() {
  // runs repeatedly: main program logic
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

2.5.2 Programming for Raspberry Pi (Python Example)

Raspberry Pi programs typically use the RPi.GPIO or gpiozero library in Python to control GPIO pins. The following example blinks an LED connected to GPIO pin 18:

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(18, GPIO.OUT)

while True:
    GPIO.output(18, GPIO.HIGH)
    time.sleep(1)
    GPIO.output(18, GPIO.LOW)
    time.sleep(1)
```

2.5.3 Key Programming Concepts for IoT

- **Pin Modes:** Digital pins can be configured as INPUT, OUTPUT, or INPUT_PULLUP depending on whether they read a sensor or drive an actuator.
- **Delays and Timers:** The delay() function pauses program execution for a set duration; millis() is used for non-blocking timing in more advanced sketches.
- **Interrupts:** Allow the microcontroller to immediately respond to an external event (e.g., a button press) without continuously polling in the loop.
- **Serial Communication:** Serial.begin(), Serial.print(), and Serial.read() are used to send/receive data over USB for debugging or communication with a host computer.

2.6 Reading Data from Sensors

Reading data from a sensor requires understanding whether the sensor produces an analog or digital signal, and applying the correct function calls to acquire and interpret that signal within the microcontroller program.

2.6.1 Reading Analog Sensors

Analog sensors are connected to the analog input pins (A0-A5 on Arduino Uno) and read using the analogRead() function, which returns a 10-bit value (0-1023) representing the input voltage between 0V and the reference voltage (usually 5V or 3.3V).

```
int sensorPin = A0;
int sensorValue = 0;

void setup() {
  Serial.begin(9600);
}

void loop() {
  sensorValue = analogRead(sensorPin);
  float voltage = sensorValue * (5.0 / 1023.0);
  Serial.print("Sensor Value: ");
```

```
  Serial.print(sensorValue);
  Serial.print("  Voltage: ");
  Serial.println(voltage);
  delay(500);
}
```

2.6.2 Analog-to-Digital Conversion (ADC)

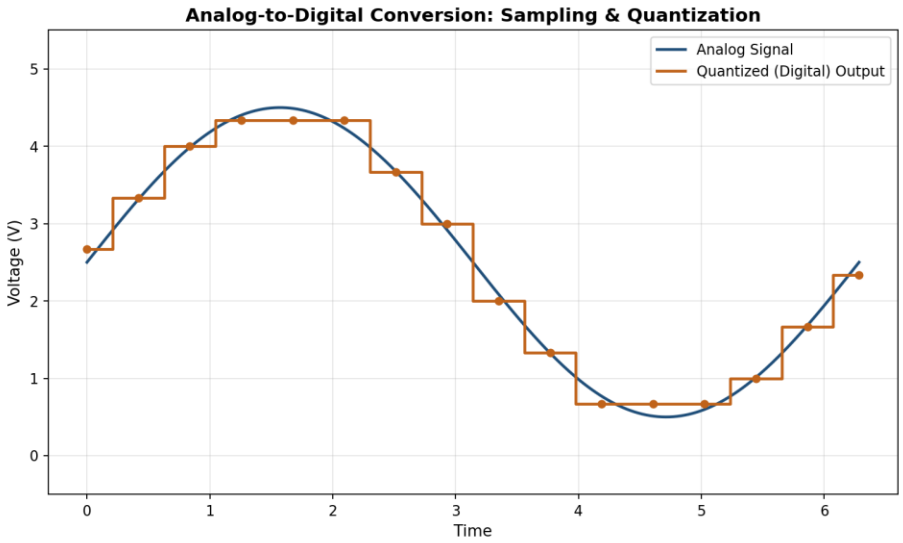

Fig. 2.5 — Analog-to-Digital Conversion: Sampling and Quantization

As shown in Fig. 2.5, the continuous analog signal is first sampled at discrete time intervals, and each sample is then quantized to the nearest representable digital level. The number of representable levels depends on the ADC's bit resolution (n): an n-bit ADC can represent 2^n distinct levels.

2.6.3 Reading Digital Sensors

Digital sensors output either a HIGH/LOW logic level (read using digitalRead()) or a timed pulse/serial protocol (e.g., DHT11 uses a single-wire timed protocol, while BMP280 uses I2C). Library functions abstract the low-level protocol so that the sensor value can be read with a simple function call.

```
#include <DHT.h>
#define DHTPIN 2
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(9600);
  dht.begin();
}
```

```
void loop() {
  float humidity = dht.readHumidity();
  float temperature = dht.readTemperature();
  Serial.print("Humidity: "); Serial.print(humidity);
  Serial.print("%  Temperature: "); Serial.print(temperature);
  Serial.println(" C");
  delay(2000);
}
```

2.7 Communication through Bluetooth

Bluetooth is a short-range wireless communication technology operating in the 2.4 GHz ISM band, widely used to connect IoT prototyping boards with smartphones, computers, or other nearby devices without requiring a WiFi access point. Bluetooth Low Energy (BLE) is a power-optimized variant of Bluetooth designed for devices that must run for long periods on small batteries.

2.7.1 Bluetooth Protocol Stack

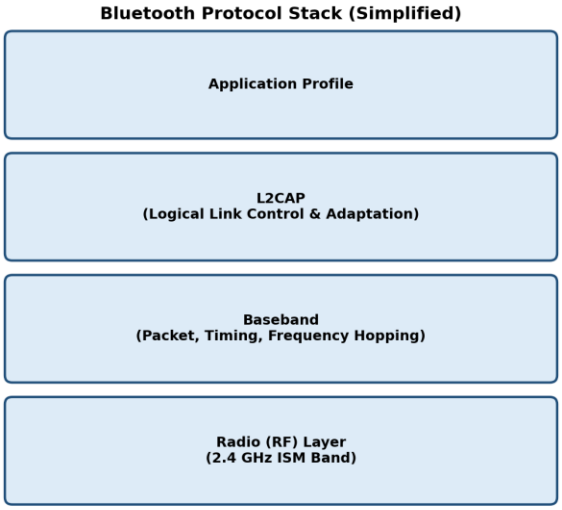

Fig. 2.6 — Simplified Bluetooth Protocol Stack

The Radio (RF) layer handles the physical transmission at 2.4 GHz. The Baseband layer manages packet formatting, timing, and frequency-hopping spread spectrum (FHSS) for interference resistance. L2CAP (Logical Link Control and Adaptation Protocol) provides multiplexing and segmentation/reassembly of data. The Application Profile layer defines how specific applications (e.g., Serial Port Profile, Health Device Profile) use the underlying stack.

2.7.2 Interfacing a Bluetooth Module (HC-05) with Arduino

The HC-05 is a popular Bluetooth module that communicates with Arduino over serial (UART) using its RX and TX pins, allowing wireless serial communication with a paired smartphone or computer.

```
#include <SoftwareSerial.h>
SoftwareSerial BTSerial(10, 11); // RX, TX

void setup() {
  Serial.begin(9600);
  BTSerial.begin(9600);
}

void loop() {
  if (BTSerial.available()) {
    char c = BTSerial.read();
    Serial.write(c);   // print data received via Bluetooth
  }
  if (Serial.available()) {
    char c = Serial.read();
    BTSerial.write(c); // send data via Bluetooth
  }
}
```

2.7.3 Bluetooth Classic vs Bluetooth Low Energy (BLE)

Feature	Bluetooth Classic	Bluetooth Low Energy (BLE)
Power Consumption	Higher	Very Low
Data Rate	Up to 3 Mbps	Up to 1-2 Mbps
Connection Setup Time	~100 ms	~3 ms
Typical Applications	Audio streaming, file transfer	Wearables, beacons, sensor telemetry

2.8 Communication through Wi-Fi

Wi-Fi (IEEE 802.11) is a wireless local area networking technology that allows IoT devices to connect directly to the Internet through a wireless access point (router), enabling continuous cloud connectivity for data logging, remote monitoring, and control.

2.8.1 Wi-Fi based IoT Communication Architecture

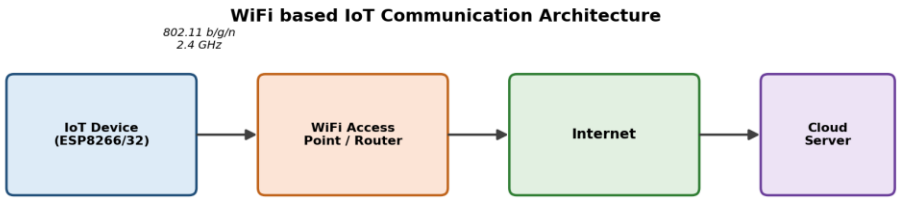

Fig. 2.7 — Wi-Fi based IoT Communication Architecture

As shown in Fig. 2.7, an IoT device fitted with a Wi-Fi module (such as the ESP8266 or ESP32) associates with a nearby Wi-Fi access point, which routes its traffic through the Internet to a cloud server for storage, processing, or remote access via a mobile/web application.

2.8.2 ESP8266 / ESP32 Wi-Fi Modules

The ESP8266 and ESP32 are low-cost System-on-Chip (SoC) modules with an integrated 32-bit microcontroller and built-in Wi-Fi (802.11 b/g/n), making them extremely popular for connected IoT prototyping. The ESP32 additionally includes Bluetooth/BLE support, dual-core processing, and more GPIO pins.

2.8.3 Connecting an ESP8266/ESP32 to Wi-Fi (Arduino IDE)

```
#include <ESP8266WiFi.h>

const char* ssid = "YOUR_WIFI_SSID";
const char* password = "YOUR_WIFI_PASSWORD";

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
```

```
  }
  Serial.println("WiFi connected");
  Serial.println(WiFi.localIP());
}

void loop() {
  // application logic: send sensor data to a server
}
```

2.8.4 Sending Sensor Data to the Cloud (HTTP GET Example)

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>

void sendData(float temperature) {
  HTTPClient http;
  String url = "http://api.example.com/update?temp=" + String(temperature);
  http.begin(url);
  int httpCode = http.GET();
  Serial.println(httpCode);
  http.end();
}
```

2.8.5 Bluetooth vs Wi-Fi for IoT Communication

Parameter	Bluetooth / BLE	Wi-Fi
Range	10 - 100 m	50 - 100 m (indoor)
Power Consumption	Very Low	High
Data Rate	Up to 2 Mbps (BLE)	Up to 1+ Gbps
Internet Connectivity	Requires a gateway/phone	Direct via access point
Best suited for	Wearables, proximity sensing	Continuous cloud-connected devices

2.9 Important Formulae

The following formulae are essential for analysing sensor readings, ADC conversion, and communication parameters while prototyping IoT objects. Each formula is presented so it can be copied directly for calculations.

2.9.1 Sensor Sensitivity

Sensitivity (S) = ΔOutput / ΔInput

where ΔOutput = change in sensor output, ΔInput = change in measured physical quantity

2.9.2 Percentage Error / Accuracy

Percentage Error (%) = |(Measured Value - True Value) / True Value| x 100

2.9.3 ADC Resolution (Step Size)

The smallest change in analog voltage that an n-bit ADC can detect, given a reference voltage Vref:

Resolution (Step Size) = Vref / 2^n

Example (Arduino Uno): 5V / 2^10 = 5 / 1024 = 4.88 mV per step

2.9.4 ADC Digital Output Code

The digital code produced by the ADC for an input voltage Vin:

Digital Code = round[(Vin / Vref) x (2^n - 1)]

2.9.5 Voltage from ADC Reading (Arduino analogRead)

Voltage (V) = analogRead() x (Vref / 1023)

For Vref = 5V: Voltage = analogRead() x (5.0 / 1023)

2.9.6 PWM Duty Cycle (Actuator Control)

Pulse Width Modulation is commonly used to control motor speed or LED brightness:

Duty Cycle (%) = (Pulse ON Time / Total Period) x 100

Average Output Voltage = Duty Cycle x Supply Voltage

2.9.7 UART Baud Rate and Bit Time

Bit Time (s) = 1 / Baud Rate

Example: At 9600 baud, Bit Time = 1/9600 = 104.16 microseconds

2.9.8 Bluetooth / Wi-Fi Data Transfer Time

Transfer Time (s) = Data Size (bits) / Data Rate (bps)

2.9.9 Worked Example: ADC Reading to Temperature (LM35 Sensor)

The LM35 temperature sensor outputs 10 mV per degree Celsius. Given an Arduino analogRead() value of 205 with a 5V reference, calculate the temperature.

Voltage = analogRead() x (5.0 / 1023) = 205 x (5.0/1023) = 1.002 V

Temperature (C) = Voltage (mV) / 10 = 1002 mV / 10 = 100.2 °C

2.9.10 Complete Prototyping Setup

Fig. 2.8 summarizes a complete IoT prototyping setup that brings together sensors, actuators, a microcontroller, and wireless communication to a cloud/mobile application — integrating all the concepts covered in this unit.

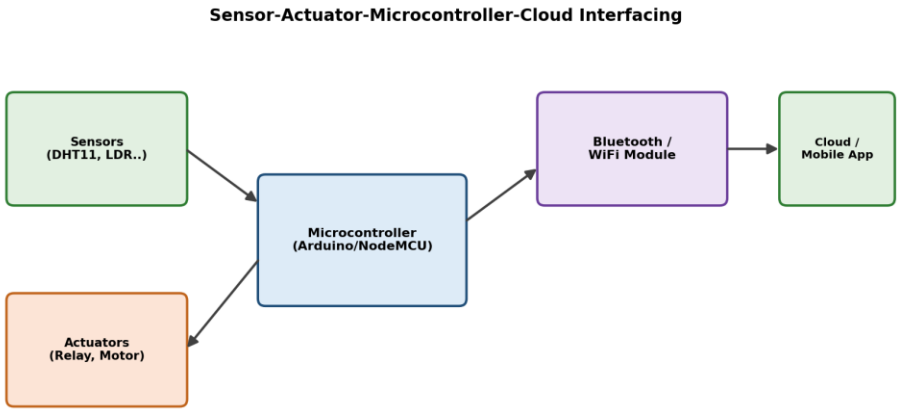

Fig. 2.8 — Sensor-Actuator-Microcontroller-Cloud Interfacing

2.10 Summary

- Sensors convert physical quantities into electrical signals; key parameters include sensitivity, resolution, accuracy, precision, response time, and linearity.
- Actuators convert electrical control signals into physical actions and are classified as electrical, mechanical, hydraulic, pneumatic, or thermal.
- Setting up a board (Arduino or Raspberry Pi) involves installing the IDE/OS, connecting the device, and configuring the correct board/port settings.
- Arduino programs use the setup()/loop() structure, while Raspberry Pi programs typically use Python with the GPIO library.
- Analog sensors are read using analogRead() and the ADC converts a continuous analog signal into discrete digital values based on its bit resolution.
- Bluetooth (and BLE) enables short-range, low-power wireless communication, ideal for wearables and proximity-based applications.
- Wi-Fi enables direct Internet connectivity for IoT devices via an access point, suited for continuous cloud-connected applications.
- Key formulae covered include sensitivity, ADC resolution and digital code, PWM duty cycle, UART bit time, and data transfer time calculations.

2.11 Review Questions

Short Answer Questions

19. Define a sensor and an actuator, giving one example of each.
20. List any five characteristics/performance parameters of a sensor.

21. What is the difference between an analog sensor and a digital sensor?
22. What is the function of the setup() and loop() functions in an Arduino sketch?
23. What is ADC? State the formula for ADC resolution.
24. Differentiate between Bluetooth Classic and Bluetooth Low Energy (BLE).
25. What is the purpose of the analogRead() function in Arduino programming?
26. List the steps to connect an ESP8266/ESP32 module to a Wi-Fi network.
27. Define PWM and state the formula for duty cycle.
28. What is the significance of the reference voltage (Vref) in ADC conversion?

Long Answer Questions

29. Explain the working principle of sensors with a neat block diagram of the sensor measurement chain.
30. Explain the classification of actuators with examples and a labelled diagram.
31. Describe the step-by-step procedure to set up an Arduino Uno board and upload a program.
32. Explain the process of reading data from analog and digital sensors with example code.
33. Explain the Bluetooth protocol stack and describe how an HC-05 module is interfaced with Arduino.
34. Explain the Wi-Fi based IoT communication architecture and demonstrate how an ESP8266 module sends sensor data to a cloud server.
35. Derive the formula for ADC digital output code and solve a numerical example converting an analogRead() value to a physical temperature reading.
36. Compare Bluetooth and Wi-Fi as communication technologies for IoT prototyping, highlighting range, power, and data rate trade-offs.

References

- Vijay Madisetti and Arshdeep Bahga, "Internet of Things (A Hands-on Approach)", 1st Edition, VPT, 2014.
- Cuno Pfister, "Getting Started with the Internet of Things", O'Reilly Media, 2011.
- Arduino Official Documentation: https://www.arduino.cc/
- Raspberry Pi Official Documentation: https://www.raspberrypi.org/

UNIT – III

IoT Architecture and Protocols

Topics Covered: Architecture Reference Model | Reference Model & Architecture |

IoT Reference Model | 6LoWPAN | RPL | CoAP | MQTT | IoT Framework - ThingSpeak

Table of Contents

3.1 Introduction to Unit III

Unit III deals with the architectural foundations of the Internet of Things and the key communication protocols that allow constrained devices to exchange data reliably over lossy, low-power networks. An IoT system connects an enormous number of heterogeneous devices — sensors, actuators, gateways, and servers — that differ in computing power, memory, energy budget, and network reach. Without a common architectural blueprint, building interoperable and scalable IoT systems becomes extremely difficult. This unit therefore begins with the IoT Architecture Reference Model (ARM), which provides a structured, technology-neutral way of describing IoT systems, and then narrows down to the concrete reference models (3-layer and 5-layer) that are commonly taught and implemented.

The unit further studies the protocol stack used specifically for constrained IoT networks — 6LoWPAN for IPv6 header compression over low-power wireless links, RPL for routing in Low-power and Lossy Networks (LLNs), CoAP as a lightweight RESTful application protocol, and MQTT as a publish/subscribe messaging protocol. Finally, the unit introduces ThingSpeak, an open cloud-based IoT analytics framework, as a practical case study of how these protocols and models come together in a real platform.

Learning Objectives

- Understand why a reference architecture is required for IoT systems.
- Describe the domains, functional groups and views of the IoT Architecture Reference Model.
- Compare the 3-layer, 5-layer and 7-layer IoT reference models.
- Explain the working, header format, and header-compression mechanism of 6LoWPAN.
- Explain RPL's DODAG construction, Rank computation, and control messages.
- Describe CoAP's request/response and messaging layers, message format and methods.
- Describe MQTT's publish/subscribe model, topics, and Quality of Service levels.
- Use ThingSpeak as an IoT analytics framework for data collection and visualization.

3.2 IoT Architecture Reference Model (ARM) — Introduction

An Architecture Reference Model (ARM) is an abstract, technology-independent framework that captures the essential concepts, relationships, and design principles common to a broad class of systems. For the Internet of Things, the ARM was developed to answer a fundamental problem: IoT systems are built by many different vendors, using many different technologies, for many different application domains (smart homes, smart cities, healthcare, agriculture, industry, and so on). Without an agreed set of concepts and rules, systems built independently cannot interoperate, and every new project has to reinvent basic design decisions from scratch.

3.2.1 Why an IoT Reference Architecture is Needed

- Interoperability: Devices and platforms from different manufacturers must be able to exchange data and services in a consistent manner.
- Reusability: Common design patterns (e.g., device registration, resource discovery) can be reused across projects instead of being redesigned.
- Scalability: A layered, modular architecture allows the system to grow from a handful of devices to millions of devices.
- Security & Trust: A reference model makes it possible to reason about security and privacy requirements consistently across domains.
- Guidance for Standardization: Reference models feed into standards bodies (IEEE, IETF, oneM2M, ITU-T) helping unify fragmented IoT ecosystems.

3.2.2 Reference Model vs Reference Architecture

It is important to distinguish two closely related terms used in IoT architecture design:

Term	Description
Reference Model	The highest level of abstraction. Describes the main concepts (Domain Model, Information Model, Communication Model, etc.) and how they relate — independent of any concrete technology.
Reference Architecture	A more concrete blueprint derived from the reference model. Provides architectural views (functional, information, deployment) and perspectives (security, performance) that guide implementation of a real system.

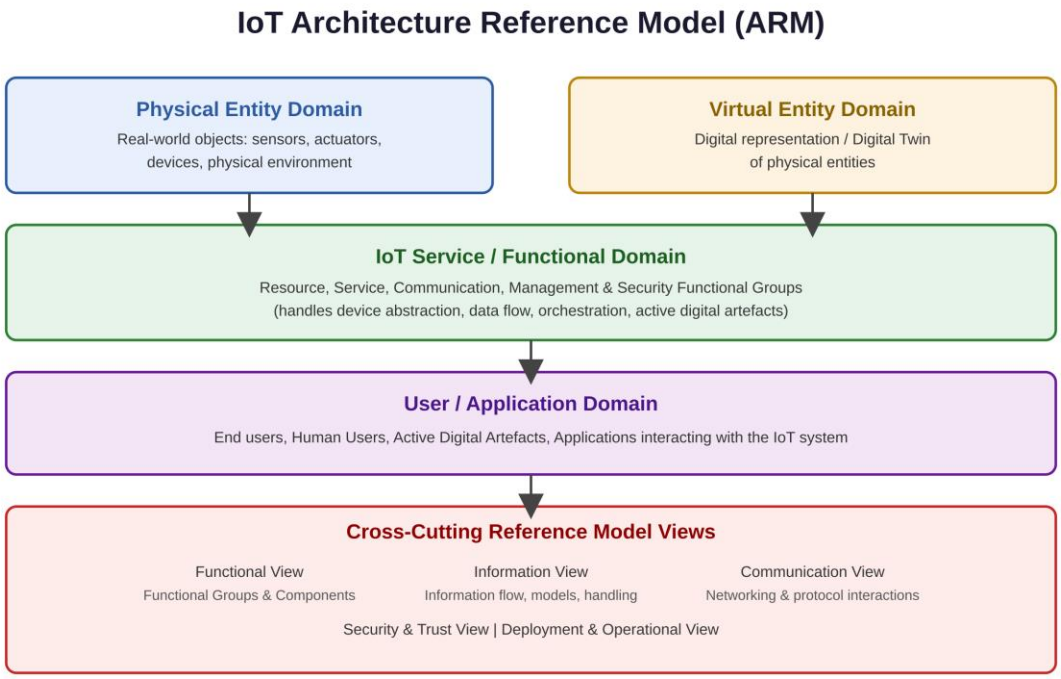

Fig 3.1: Domains and cross-cutting views of the IoT Architecture Reference Model

3.2.3 Domains of the IoT Domain Model

The IoT ARM Domain Model identifies the principal concepts and their relationships, grouped into four broad domains:

- Physical Entity Domain: Represents real-world objects of interest — a room, a vehicle, a person, a machine — that are monitored or controlled by the IoT system.
- Virtual Entity Domain: The digital counterpart (Virtual Entity / Digital Twin) of a physical entity, maintained inside the information system, which applications actually operate on.
- IoT Service / Functional Domain: The set of functional groups (Device, Communication, IoT Service, Virtual Entity, IoT Process Management, Service Organization, Security, Management) that realize the behaviour of the IoT system.
- User / Application Domain: Human users and Active Digital Artefacts (autonomous software agents) that consume IoT services through applications.

3.3 Reference Model and Architecture

3.3.1 Functional Groups (Functional View)

The IoT Reference Architecture organizes system behaviour into Functional Groups (FGs), each containing Functional Components (FCs) that expose well-defined interfaces:

Functional Group	Responsibility
Device FG	Sensing, actuation, tag reading, storage and processing at the device level.
Communication FG	Abstracts heterogeneous networking technologies and provides an end-to-end communication channel (Hop-to-Hop, Network, End-to-End Communication components).
IoT Service FG	Exposes and manages IoT Services associated with resources/devices, handles service description and discovery.
Virtual Entity FG	Manages Virtual Entities, keeps their association with physical entities and IoT services up to date.
IoT Process Management FG	Integrates business processes with IoT-related services, including process modelling and execution.
Service Organization FG	Composes, orchestrates and chains basic IoT services into higher level composite services.
Security FG	Provides authentication, authorization, identity management, trust, and key management.
Management FG	Configuration, fault, performance monitoring, and reporting of IoT system components.

3.3.2 Information View

The Information View describes how information about Virtual Entities and related IoT services flows and is represented, handled, and stored within the system. It defines the Information Model (attributes and metadata of Virtual Entities), and the lifecycle of information — from generation at the device, through aggregation and contextualization, to consumption by applications.

3.3.3 Communication View

The Communication View addresses how the various devices, gateways and servers communicate with each other, dealing with the constraints of low-power and lossy links. It typically distinguishes:

- Device-to-Device communication (D2D) — direct communication, often over short-range radios (Zigbee, BLE, 802.15.4).
- Device-to-Gateway communication (D2G) — devices report through a local gateway that performs protocol translation.
- Gateway-to-Cloud / Backend communication — the gateway relays aggregated data to cloud services over IP-based networks.

- Device-to-Cloud communication (D2C) — devices with IP capability communicate directly with cloud services.

3.3.4 Perspectives: Security, Trust, Performance, Deployment

Cutting across every layer/view are non-functional 'perspectives' that a designer must consider while instantiating the reference architecture into a concrete system: Security & Privacy (encryption, key management, access control), Trust & Reputation (validity of data and identities), Performance & Scalability (latency, throughput under massive device counts), and Deployment & Operational aspects (device provisioning, firmware updates, lifecycle management).

3.4 IoT Reference Model

While the ARM gives an abstract, domain-driven view, most textbooks and practical designs describe IoT systems using simpler layered reference models. The most common are the 3-layer, 5-layer, and 7-layer models.

3.4.1 3-Layer IoT Reference Model

This is the most basic and widely used model, especially for introductory understanding of IoT systems.

Layer	Function	Typical Technologies
Application Layer	Delivers application-specific services to the end user (smart home, smart health, smart agriculture, etc.)	Mobile/web apps, dashboards
Network Layer	Routes data collected from the perception layer over the Internet/intranet to processing systems	Wi-Fi, 3G/4G/5G, 6LoWPAN, RPL
Perception Layer	Physical sensors/actuators that collect and act on information from the environment	RFID, temperature/humidity sensors, cameras, GPS

3.4.2 5-Layer IoT Reference Model

The 5-layer model refines the 3-layer model by splitting the Network layer into Transport and Processing, and inserting a Business layer above the Application layer to capture the overall business logic and value-creation aspect of IoT deployments.

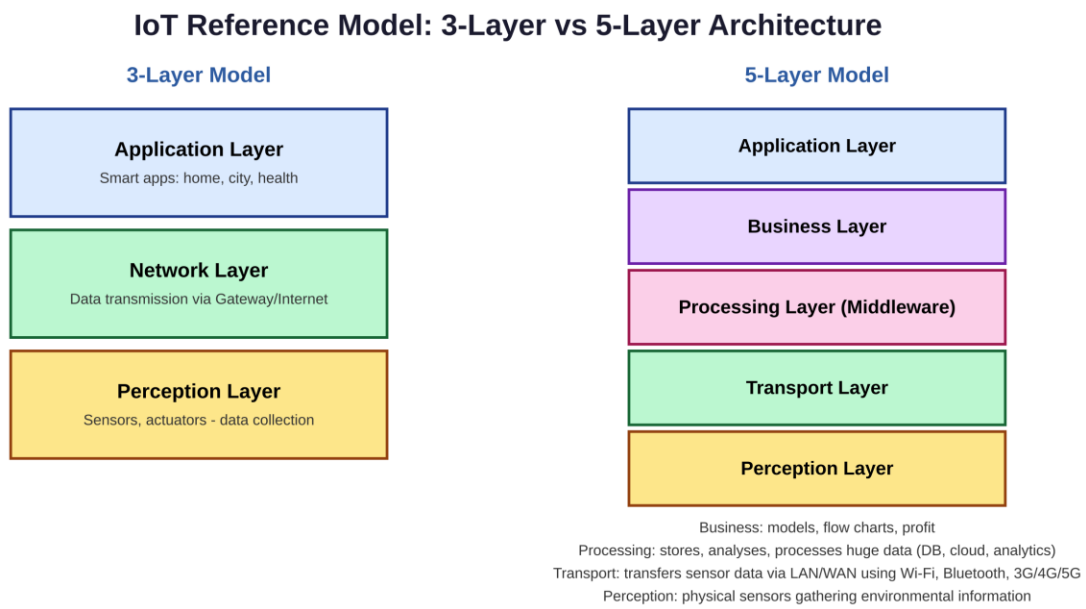

Fig 3.2: Comparison of the 3-Layer and 5-Layer IoT Reference Models

Layer	Description
Business Layer	Manages the overall IoT system: business models, flowcharts, profit models, and user privacy policies.
Application Layer	Delivers application-specific services (e.g. smart home automation, fleet management).
Processing Layer (Middleware)	Stores, analyses and pre-processes huge volumes of data coming from the transport layer, using databases, cloud computing and big-data modules.
Transport Layer	Transfers sensor data from the perception layer to the processing layer and vice-versa using Wi-Fi, Bluetooth, RFID, or cellular networks.
Perception Layer	The physical layer having sensors to sense the environment and gather information; equivalent to the 'things' in IoT.

3.4.3 7-Layer IoT World Forum Reference Model (Overview)

For completeness, industry consortia such as the IoT World Forum propose a more granular 7-layer model that separates edge computing, data accumulation, and collaboration/processes as distinct layers: (1) Physical Devices & Controllers, (2) Connectivity, (3) Edge (Fog) Computing, (4) Data Accumulation, (5) Data Abstraction, (6) Application, and (7) Collaboration & Processes (involving people). This model is

particularly useful for large enterprise and industrial IoT deployments where edge analytics and human collaboration play a significant role.

3.4.4 IoT Levels and Deployment Templates

An IoT system can additionally be described by an 'IoT Level', a template describing the number of physical entities/end-nodes, degree of local vs cloud analytics, and use of local or remote storage. Common levels described in reference textbooks include:

- Level 1: A single node performs sensing/actuation, analysis, and application hosting locally (e.g., a home automation controller).
- Level 2: A single node with cloud-based data storage/analysis for reduced local processing (e.g., a smart camera uploading footage to cloud analytics).
- Level 3: A single node with a remote application, no local application logic; the observer node streams data to a cloud application.
- Level 4: Multiple end nodes with local analysis, sending processed results to the cloud (distributed sensing with edge processing).
- Level 5: Multiple end nodes coordinated by an observer node that forwards data to the cloud for centralized storage/analysis.
- Level 6: Multiple independent end nodes, each sending data directly to the cloud (fully distributed collection and centralized processing).

3.5 IoT Protocols — Overview of the Protocol Stack

Constrained IoT devices operate under strict limits on memory, processing power, and battery life, and they typically communicate over Low-power and Lossy Networks (LLNs) using radios such as IEEE 802.15.4. Standard Internet protocols (full IPv6, HTTP/TCP) are too heavyweight for such devices, motivating a dedicated 'constrained' protocol stack, illustrated below.

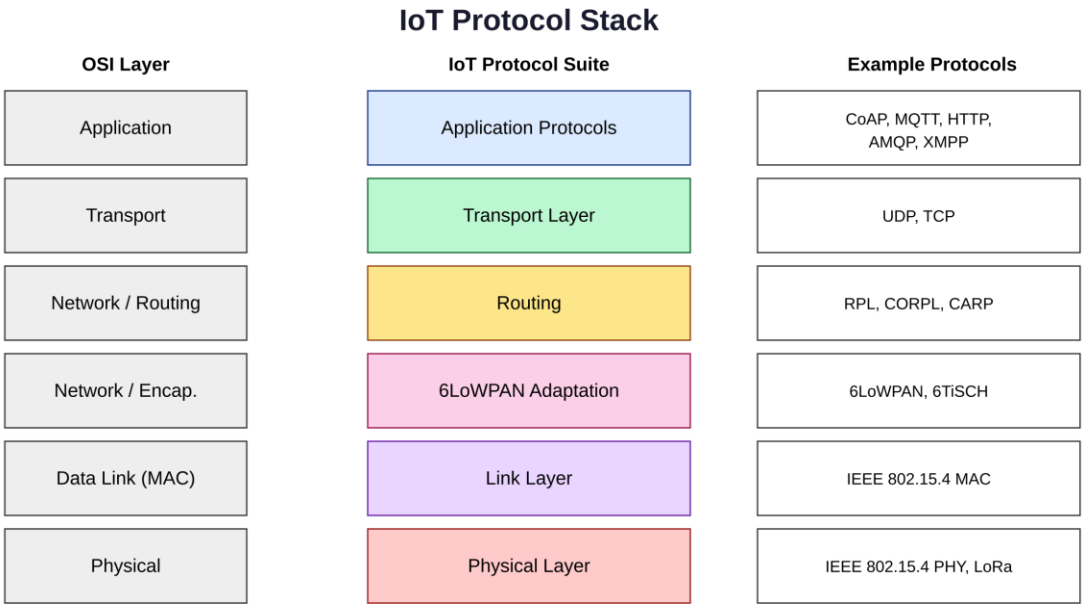

Fig 3.3: IoT Protocol Stack mapped against OSI layers

The remainder of this unit examines four key protocols from this stack in detail: 6LoWPAN (network/adaptation layer), RPL (routing layer), CoAP (application layer over UDP), and MQTT (application layer over TCP, publish/subscribe).

3.6 6LoWPAN — IPv6 over Low-Power Wireless Personal Area Networks

3.6.1 Introduction

6LoWPAN (IPv6 over Low-power Wireless Personal Area Networks) is a standard defined by the IETF (RFC 4944, updated by RFC 6282) that enables the transmission of IPv6 packets over IEEE 802.15.4-based networks, which are characterised by small frame sizes (127 bytes maximum), low bandwidth, and limited processing capability. 6LoWPAN acts as an adaptation layer sitting between the IPv6 network layer and the IEEE 802.15.4 MAC/PHY layers.

3.6.2 Need for 6LoWPAN

- A plain IPv6 header alone is 40 bytes; adding an 8-byte UDP header leaves very little space in a 127-byte 802.15.4 frame for actual application payload.
- IPv6 requires a Maximum Transmission Unit (MTU) of at least 1280 bytes, far larger than the 802.15.4 frame size, so fragmentation and reassembly are required.
- 6LoWPAN provides header compression, fragmentation/reassembly, and mesh addressing so that IPv6 can operate efficiently over such constrained links.

3.6.3 Functions of the 6LoWPAN Adaptation Layer

- Header Compression (HC): Compresses the IPv6 and UDP headers using context information shared between sender and receiver (stateless/stateful compression defined by IPHC — RFC 6282).
- Fragmentation and Reassembly: Splits IPv6 packets larger than the 802.15.4 frame size into multiple fragments at the source and reassembles them at the destination.
- Mesh Addressing / Routing Support: Supports mesh-under routing at the adaptation layer, complementing route-over routing done by protocols like RPL.
- Stateless Address Auto-configuration: Derives IPv6 interface identifiers from the 802.15.4 short/extended address.

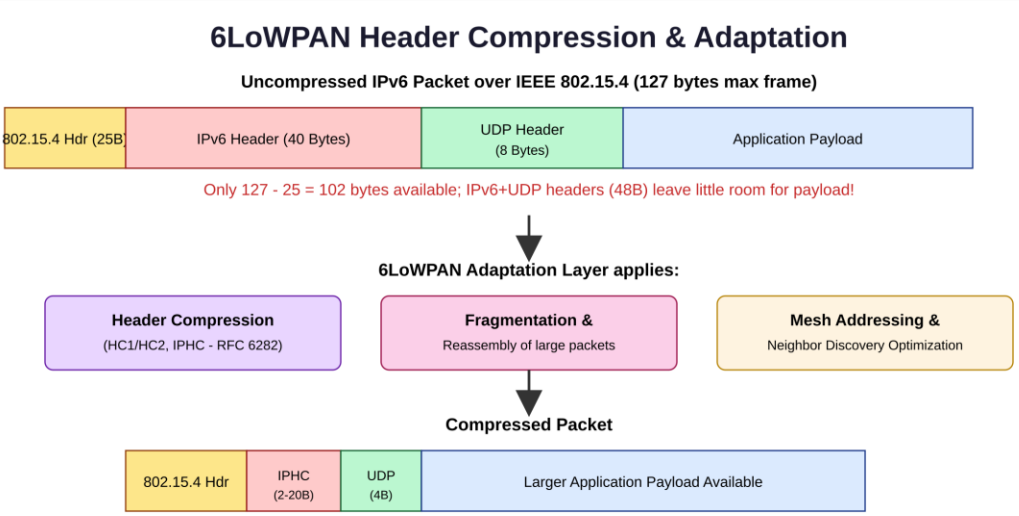

Fig 3.4: 6LoWPAN header compression and fragmentation over an IEEE 802.15.4 frame

3.6.4 6LoWPAN Frame Format Fields

A 6LoWPAN-encapsulated packet begins with a Dispatch byte identifying the type of header that follows, such as:

Pattern (binary)	Header Type
00xxxxxx	Not a 6LoWPAN frame (reserved)
01000001	Uncompressed IPv6 Addressing Header (IPv6)
01000010	LOWPAN_HC1 Compressed IPv6 Header (legacy)
01100xxx	LOWPAN_IPHC Header (RFC 6282, current standard)
11xxxxxx	Mesh Header

Pattern (binary)	Header Type
11000xxx	Fragmentation Header

3.6.5 Header Compression Ratio — Illustrative Calculation

The efficiency gained from 6LoWPAN header compression can be expressed with a simple compression-ratio formula:

Compression Ratio (%) = [(Original Header Size − Compressed Header Size) / Original Header Size] × 100

Example: For an IPv6 + UDP header of 48 bytes (40 + 8) compressed down to 6 bytes using IPHC with shared context, the compression ratio is:

Compression Ratio = [(48 − 6) / 48] × 100 = 87.5%

This dramatic reduction leaves far more of the 102-byte usable payload space (127-byte frame minus 25-byte MAC header) for actual application data, which is essential for constrained sensor networks.

3.7 RPL — Routing Protocol for Low-Power and Lossy Networks

3.7.1 Introduction

RPL (pronounced 'ripple', RFC 6550) is a distance-vector routing protocol designed by the IETF ROLL working group specifically for Low-power and Lossy Networks (LLNs) such as wireless sensor networks. LLNs are characterised by high packet loss, low throughput, and thousands of resource-constrained nodes, making traditional routing protocols like OSPF or RIP unsuitable.

3.7.2 Destination Oriented Directed Acyclic Graph (DODAG)

RPL organises the network topology as a Destination Oriented Directed Acyclic Graph (DODAG) — a tree-like structure rooted at a single sink node, commonly the LLN Border Router (LBR), which is usually the gateway to the Internet or a central controller. Every other node in the network selects one or more parents that are closer (in terms of a defined metric) to the DODAG root, and all traffic destined for the root flows up along parent links.

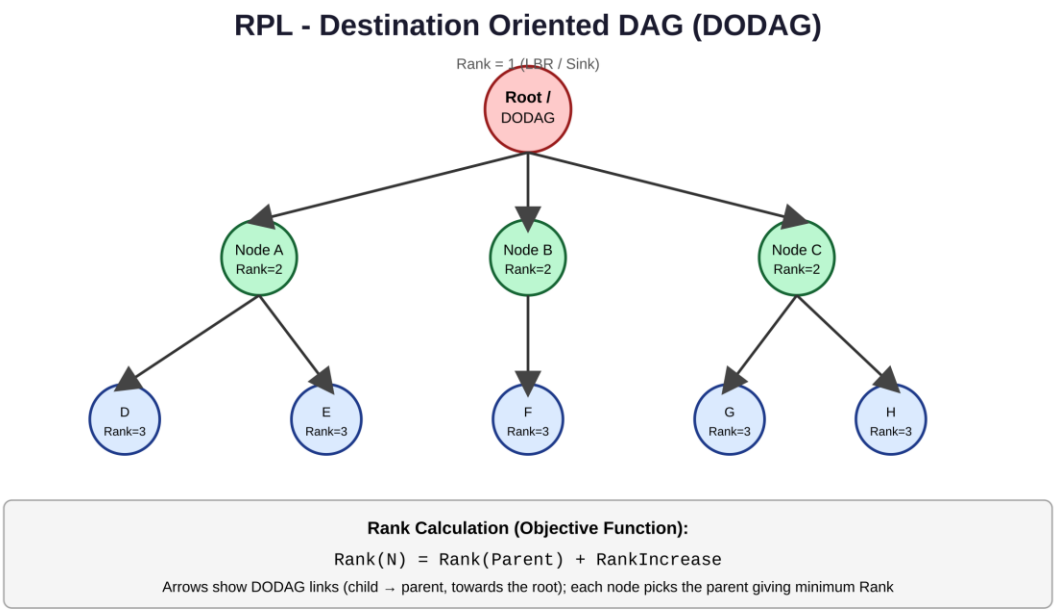

Fig 3.5: RPL Destination Oriented DAG (DODAG) topology with Rank values

3.7.3 Rank and Objective Function

Each node in a DODAG is assigned a Rank, a scalar value representing the node's relative position with respect to the DODAG root — Rank strictly decreases as one moves towards the root. Rank is computed using an Objective Function (OF), which defines the routing metric (e.g., hop count, expected transmission count ETX, energy) and how it is combined along a path:

Rank(N) = Rank(Parent) + RankIncrease

where RankIncrease is computed by the Objective Function based on the chosen routing metric (for example, using hop-count OF0, RankIncrease is typically a fixed increment per hop; using the Minimum Rank with Hysteresis Objective Function (MRHOF), RankIncrease reflects the link's ETX cost). A node may have several candidate parents but selects the Preferred Parent that yields the smallest Rank, subject to hysteresis to avoid frequent route flapping.

3.7.4 RPL Control Messages (ICMPv6)

Message	Purpose
DIS (DODAG Information Solicitation)	Sent by a node to solicit DIO messages from neighbours, used to discover nearby DODAGs.
DIO (DODAG Information Object)	Multicast by nodes already in the DODAG, carrying Rank, DODAG ID, and configuration parameters; used to construct/maintain the DODAG.
DAO (Destination Advertisement	Sent by a child to its parent (unicast, upward) to advertise reachability of

Message	Purpose
Object)	destinations, enabling downward routes.
DAO-ACK	Acknowledgement of a DAO message from parent to child.

3.7.5 Modes of Operation

- Storing Mode: Every intermediate node maintains routing tables for its sub-DODAG, allowing point-to-point routing without going through the root.
- Non-Storing Mode: Only the root maintains complete routing information (via Source Routing headers); intermediate nodes forward without local routing tables, trading memory for simpler node implementation.

3.7.6 Trickle Timer

RPL uses the Trickle algorithm to control how often DIO messages are (re)transmitted: the interval between transmissions doubles (up to a maximum) when the network is stable, and resets to a minimum whenever a topology change is detected. This keeps control-traffic overhead low in a stable network while enabling fast convergence after a change.

3.8 CoAP — Constrained Application Protocol

3.8.1 Introduction

The Constrained Application Protocol (CoAP, RFC 7252) is a specialized, lightweight, RESTful web-transfer protocol designed for constrained nodes and networks. Unlike HTTP, which runs over TCP, CoAP is designed to run over UDP, making it suitable for low-power devices where the overhead of establishing and maintaining a TCP connection would be prohibitive.

3.8.2 CoAP Architecture — Two-Layer Model

- Messaging Layer: Deals with UDP transport and asynchronous exchanges — handling duplication of messages and providing (optional) reliability using message types CON (Confirmable), NON (Non-confirmable), ACK (Acknowledgement) and RST (Reset).
- Request/Response Layer: Deals with the semantics of REST-style request/response interactions using methods (GET, POST, PUT, DELETE) and response codes, piggy-backed on top of the Messaging Layer.

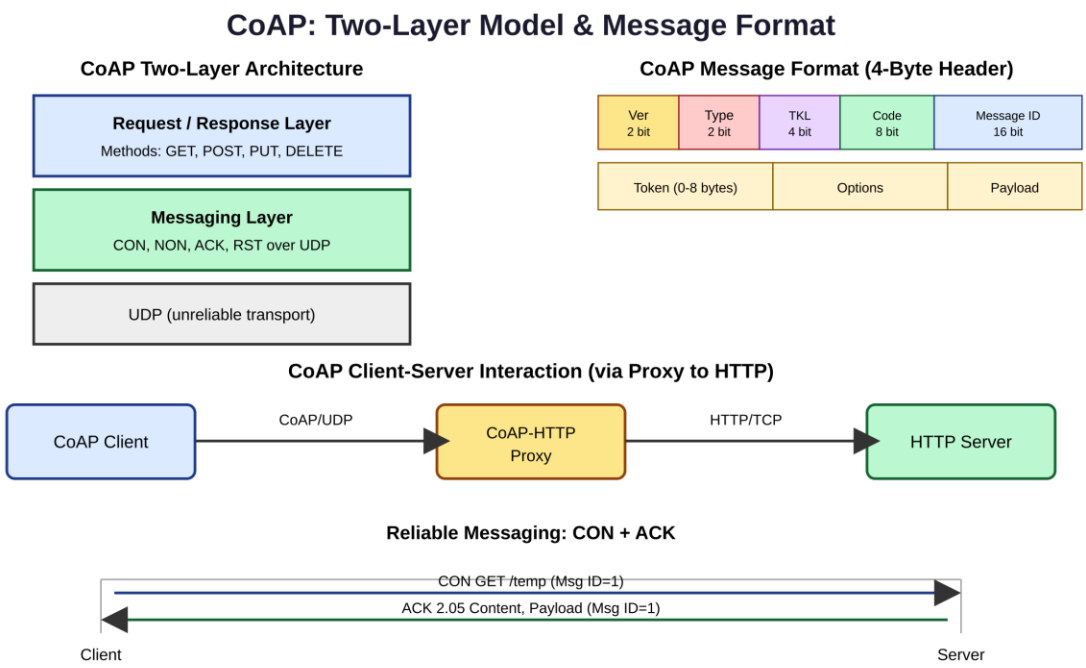

Fig 3.6: CoAP two-layer architecture, message header format, and reliable CON/ACK exchange

3.8.3 CoAP Message Format

Every CoAP message starts with a fixed 4-byte binary header, followed by a variable-length Token, Options, and an optional Payload marked off by an 0xFF byte:

Field	Size	Description
Version (Ver)	2 bits	CoAP protocol version, currently 1.
Type	2 bits	CON (0), NON (1), ACK (2), or RST (3).
Token Length (TKL)	4 bits	Length of the variable-length Token field (0-8 bytes).
Code	8 bits	Request method (e.g. 1=GET, 2=POST, 3=PUT, 4=DELETE) or response code (e.g. 2.05 Content, 4.04 Not Found).
Message ID	16 bits	Used for message de-duplication and for matching ACK/RST to a CON/NON message.

3.8.4 Message Types and Reliability

Because CoAP runs over unreliable UDP, it implements a simple stop-and-wait retransmission mechanism for messages sent as Confirmable (CON): the sender resends the message with an exponential back-off timer until an ACK is received or a maximum retransmission count is reached.

Timeout(n) = INITIAL_TIMEOUT × 2^n (with random jitter), n = retransmission attempt number

With default CoAP parameters (INITIAL_TIMEOUT ≈ 2 seconds, MAX_RETRANSMIT = 4), a CON message may be transmitted up to 5 times in total before being deemed failed, giving the sender a good chance of success even over lossy radio links.

3.8.5 CoAP Methods and Response Codes

Method / Code	Meaning
GET	Retrieve a representation of a resource.
POST	Create a new resource or trigger processing on the server.
PUT	Update / create a resource at a known URI.
DELETE	Delete the specified resource.
2.05 Content	Success — response contains a representation of the resource.
4.04 Not Found	Requested resource does not exist.
5.00 Internal Server Error	Server encountered an error fulfilling the request.

3.8.6 CoAP Features Summary

- RESTful interaction model similar to HTTP, but with a much smaller 4-byte binary header vs HTTP's verbose text headers.
- Built-in support for resource discovery via the well-known '/.well-known/core' URI.
- Observe option allows a client to subscribe to a resource and receive notifications on changes (a lightweight publish-like extension).
- Supports proxying and caching, and can interwork with HTTP through CoAP-HTTP proxies for integration with existing web infrastructure.
- Optionally secured using DTLS (Datagram TLS) as CoAPs, analogous to HTTPS for HTTP.

3.9 MQTT — Message Queuing Telemetry Transport

3.9.1 Introduction

MQTT is a lightweight, publish/subscribe based messaging protocol designed for constrained devices and low-bandwidth, high-latency, or unreliable networks. Originally developed by IBM, it is now an OASIS and ISO standard (ISO/IEC 20922) and is widely used in IoT applications such as remote monitoring, telemetry, and messaging between distributed sensors and cloud platforms.

3.9.2 Publish/Subscribe Architecture

Unlike CoAP's request/response model, MQTT decouples data producers from data consumers using a central MQTT Broker:

- Publisher: A client that sends (publishes) messages tagged with a Topic to the broker, without needing to know who (if anyone) is listening.
- Subscriber: A client that registers interest in one or more Topics with the broker and receives messages published to matching topics.
- Broker: The server that receives all published messages, filters them by topic, and forwards them to all subscribed clients; it also manages client sessions, authentication and persistence of retained messages.

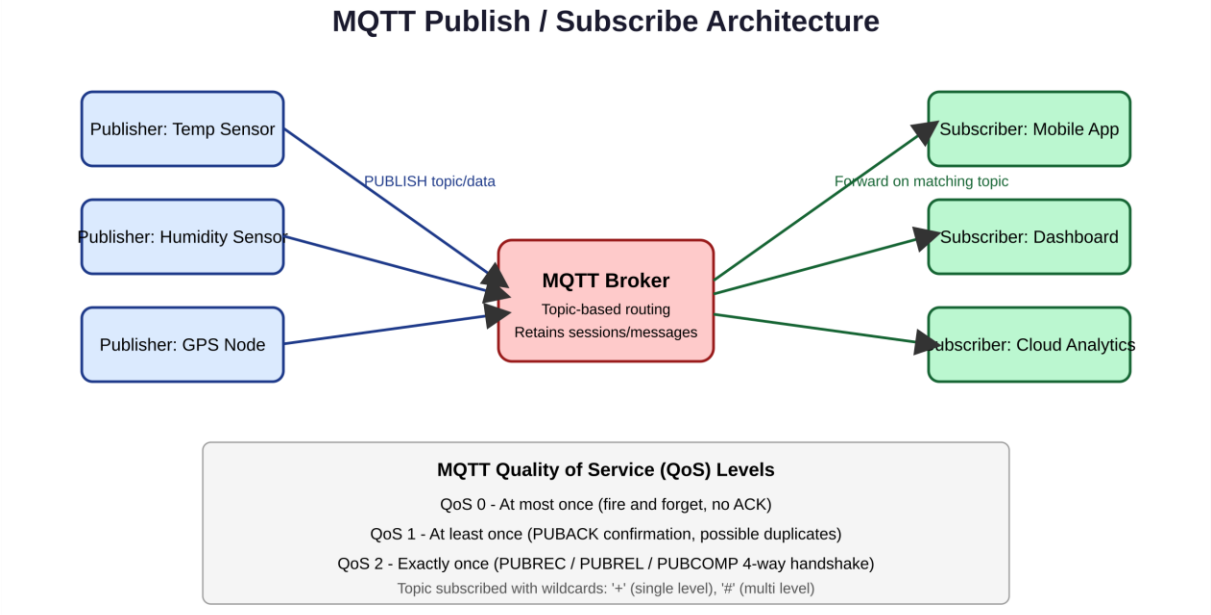

Fig 3.7: MQTT publish/subscribe architecture with broker-mediated topic routing and QoS levels

3.9.3 Topics and Wildcards

MQTT topics are hierarchical UTF-8 strings separated by a forward slash, e.g. 'home/livingroom/temperature'. Subscribers can use wildcards to match multiple topics:

Wildcard	Meaning	Example
+	Matches exactly one topic level	home/+/temperature matches home/livingroom/temperature and home/kitchen/temperature
#	Matches any number of levels (must be last character)	home/# matches all topics under 'home'

3.9.4 Quality of Service (QoS) Levels

MQTT defines three Quality of Service levels that determine the guarantee of message delivery between a client and the broker (and independently, between the broker and each subscriber):

QoS Level	Guarantee	Handshake
QoS 0	At most once (fire-and-forget; message may be lost, no duplicates)	PUBLISH only
QoS 1	At least once (guaranteed delivery; duplicates possible)	PUBLISH → PUBACK
QoS 2	Exactly once (guaranteed delivery, no duplicates)	PUBLISH → PUBREC → PUBREL → PUBCOMP

Higher QoS levels provide stronger delivery guarantees at the cost of higher latency and network overhead, which is an important design trade-off in battery-powered IoT deployments.

3.9.5 MQTT Packet Overhead

MQTT's fixed header is extremely small, contributing to its lightweight nature:

MQTT Fixed Header = 2 bytes (1 byte Control Packet type/flags + 1+ byte(s) Remaining Length)
Minimum PUBLISH Overhead ≈ Fixed Header (2B) + Topic Length (2B) + Topic Name (variable)

Compared to HTTP, where a simple request/response pair may involve several hundred bytes of text headers, MQTT's binary framing reduces the protocol overhead by an order of magnitude, which is significant for battery-powered sensors sending small, frequent updates.

3.9.6 Other Key MQTT Features

- Persistent (Clean Session = false) or Non-persistent sessions, allowing message queuing for offline clients.
- Retained Messages: the broker stores the last message on a topic and immediately delivers it to any new subscriber.

- Last Will and Testament (LWT): a message the broker publishes automatically if a client disconnects unexpectedly, useful for detecting device failures.
- Keep Alive: periodic PINGREQ/PINGRESP messages to detect broken connections.

3.9.7 CoAP vs MQTT — Quick Comparison

Aspect	CoAP	MQTT
Transport	UDP	TCP
Model	Request / Response (RESTful)	Publish / Subscribe
Header Size	4 bytes fixed	2 bytes fixed
Reliability	CON/ACK with retransmission	QoS 0 / 1 / 2
Typical Use	Device control, resource access	Telemetry, event broadcast, sensor data

3.10 IoT Frameworks — ThingSpeak

3.10.1 Introduction

ThingSpeak is an open-source, cloud-based IoT analytics platform developed by MathWorks that allows users to collect, visualize, and analyse live data streams from connected sensors and devices. It provides a simple REST and MQTT API for uploading data, and integrates directly with MATLAB for advanced analytics, making it particularly popular in academic and prototyping IoT projects.

3.10.2 Key Concepts

- Channel: The fundamental storage location in ThingSpeak, containing up to 8 data Fields, plus location and status fields, associated with a particular device or application.
- Write API Key: A per-channel secret key used by a device to authenticate when posting data to a channel.
- Read API Key: A key that allows applications or other users to read data from a (private) channel.
- MATLAB Analytics: Analysis code (using MATLAB) can be run directly within ThingSpeak on the collected channel data, without needing a separate MATLAB license.
- TimeControl / React: Enables scheduled or conditional execution of MATLAB analysis, or triggering actions (e.g., sending an alert) when incoming data satisfies a set condition.
- ThingTweet, ThingHTTP, TalkBack: Companion apps for tweeting updates, triggering external HTTP requests, and queuing commands back to devices.

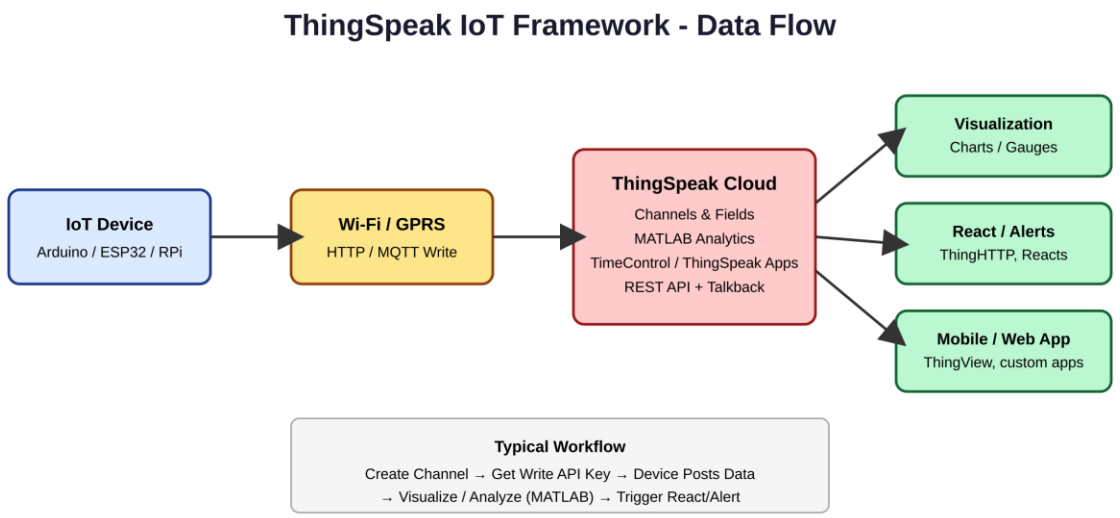

Fig 3.8: ThingSpeak IoT data-flow: from device to channel to visualization/analytics

3.10.3 Typical Workflow

- Create a ThingSpeak account and a new Channel, defining the Fields to be recorded (e.g., Field1 = Temperature, Field2 = Humidity).
- Note down the channel's Write API Key.
- Program the IoT device (e.g., ESP32/Arduino with Wi-Fi) to periodically send an HTTP GET/POST (or MQTT PUBLISH) request to the ThingSpeak update endpoint with the API key and field values.
- ThingSpeak stores each update as a timestamped entry and automatically renders default line charts for each field.
- Optionally, write MATLAB analysis code within the channel to compute statistics, detect anomalies, or generate custom visualizations.
- Configure a React to trigger an alert/action (e.g., email, ThingHTTP call) when a field value crosses a threshold.

3.10.4 Example REST API Call

A device updates a channel field using a simple HTTP GET request of the following form (illustrative, not to be treated as literal source code from any copyrighted material):

GET https://api.thingspeak.com/update?api_key=<WRITE_API_KEY>&field1=<value>

3.10.5 Advantages of ThingSpeak

- Free tier suitable for academic projects, prototyping, and small-scale deployments.
- Tight integration with MATLAB for signal processing, machine learning, and visualization without extra licensing.
- Simple REST and MQTT APIs make integration with almost any microcontroller (Arduino, ESP8266/32, Raspberry Pi) straightforward.

- Built-in support for public/private channels, enabling data sharing for research or community projects.

3.11 Solved Numerical Examples

Example 1: RPL Rank Calculation

A DODAG root R has Rank = 1. Using the hop-count based Objective Function OF0 with a fixed RankIncrease of 1 per hop (MinHopRankIncrease = 1), compute the Rank of a node that is 4 hops away from the root along its preferred parent path.

Rank(N) = Rank(Parent) + RankIncrease

Rank(hop1) = 1 + 1 = 2

Rank(hop2) = 2 + 1 = 3

Rank(hop3) = 3 + 1 = 4

Rank(hop4) = 4 + 1 = 5

Hence, a node 4 hops away from the DODAG root has a Rank of 5. Any candidate parent offering a smaller resulting Rank would be preferred, subject to the DODAG's configured hysteresis (hold-down) constant.

Example 2: 6LoWPAN Header Compression Ratio

A sensor node sends an application payload of 40 bytes. Without compression, the packet carries a 40-byte IPv6 header and an 8-byte UDP header. With 6LoWPAN IPHC compression (shared context for prefixes, elided fields), the combined header shrinks to 7 bytes. Calculate (a) the compression ratio and (b) the percentage of the 802.15.4 frame's usable payload (102 bytes after the 25-byte MAC header) occupied by headers in each case.

Original header size = 40 + 8 = 48 bytes

Compressed header size = 7 bytes

Compression Ratio = [(48 − 7) / 48] × 100 = 85.4%

Header share (uncompressed) = (48 / 102) × 100 = 47.1%

Header share (compressed) = (7 / 102) × 100 = 6.9%

The result shows that without compression nearly half of the usable frame is consumed by protocol headers, whereas with 6LoWPAN IPHC only about 7% is used — freeing up over 40 extra bytes for the application payload in every single frame.

Example 3: CoAP Confirmable Message Retransmission Timeline

A CoAP client sends a CON request with INITIAL_TIMEOUT = 2 seconds and MAX_RETRANSMIT = 4 (default parameters), doubling the timeout on every retry. Assuming no ACK is received until the final attempt, list the approximate times (ignoring random jitter) at which each transmission occurs.

Attempt 0 (original): t = 0s

Attempt 1: t = 0 + 2 = 2s

Attempt 2: t = 2 + 4 = 6s

Attempt 3: t = 6 + 8 = 14s

Attempt 4: t = 14 + 16 = 30s

If no ACK arrives after the 4th retransmission (5 transmissions total), the CoAP client gives up and reports the request as failed at approximately t = 30 seconds, having made 5 attempts in total across roughly 45 seconds of worst-case exponential back-off (including the final wait window).

Example 4: MQTT vs HTTP Overhead Comparison

A sensor sends a small JSON payload of 20 bytes representing a temperature reading. Compare the approximate total message size using (a) MQTT PUBLISH with QoS 0, and (b) a typical HTTP POST request with headers.

Protocol	Approx. Header/Framing Overhead	Total Message Size
MQTT (QoS 0)	≈ 2-5 bytes (fixed header + minimal variable header)	≈ 25 bytes
HTTP POST	≈ 200-400 bytes (request line + headers)	≈ 220-420 bytes

This comparison illustrates why MQTT (and CoAP) are preferred over plain HTTP for constrained IoT devices sending frequent, small telemetry updates — the protocol overhead using HTTP can be an order of magnitude larger than the actual payload itself.

3.12 Summary

- The IoT Architecture Reference Model (ARM) provides a technology-neutral framework of domains (Physical Entity, Virtual Entity, IoT Service, User) and views (Functional, Information, Communication) that guide the design of interoperable IoT systems.
- Practical designs commonly use the layered 3-layer (Perception, Network, Application) or 5-layer (Perception, Transport, Processing, Application, Business) IoT reference models.
- 6LoWPAN compresses IPv6/UDP headers and fragments packets so that full IPv6 connectivity can be achieved over constrained IEEE 802.15.4 links.
- RPL builds a Destination Oriented DAG rooted at a border router, using Rank and an Objective Function to select loop-free, efficient upward and downward routes in Low-power and Lossy Networks.

- CoAP is a RESTful protocol over UDP with a compact 4-byte header, supporting reliable Confirmable messaging and methods like GET/POST/PUT/DELETE.
- MQTT is a lightweight publish/subscribe protocol over TCP, using a central broker, hierarchical topics, and three QoS levels to balance reliability against overhead.
- ThingSpeak is a cloud IoT analytics framework that stores sensor data in Channels and offers MATLAB-based analytics, visualization, and alerting/reaction capabilities.

3.13 Important Questions

Short Answer Questions (2 / 3 Marks)

37. What is the need for an IoT Architecture Reference Model?
38. Differentiate between a Reference Model and a Reference Architecture.
39. List the layers of the 3-layer IoT reference model.
40. What is the purpose of the 6LoWPAN adaptation layer?
41. Define Rank in RPL.
42. What are the four CoAP message types?
43. List the three MQTT QoS levels.
44. What is a Channel in ThingSpeak?

Long Answer Questions (5 / 10 Marks)

45. Explain the domains and functional groups of the IoT Architecture Reference Model with a neat diagram.
46. Compare and contrast the 3-layer and 5-layer IoT reference models, explaining the responsibility of each layer.
47. Explain 6LoWPAN header compression with the aid of a diagram. Why is header compression essential for IoT networks?
48. Describe how RPL constructs a DODAG. Explain the roles of DIS, DIO, and DAO control messages.
49. Explain the CoAP two-layer architecture and message format. How does CoAP achieve reliability over UDP?
50. Explain the MQTT publish/subscribe model with a neat diagram, and describe the three QoS levels with their handshake mechanisms.
51. Compare CoAP and MQTT protocols on the basis of transport, architecture, and reliability.
52. Explain the architecture and working of the ThingSpeak IoT framework with a suitable case study/example.

UNIT – IV

DEVICE DISCOVERY AND CLOUD SERVICES FOR IoT

Lecture Notes

Unit IV – Syllabus Coverage

Table of Contents

4.1 Introduction to Device Discovery

Device discovery is the process by which an IoT device, gateway, or network management system identifies the presence, identity, and capabilities of other devices that are reachable within a network. It is the very first functional step in bringing a new sensor, actuator, or smart object into an operational IoT deployment, and it precedes registration, configuration, and data exchange.

In large-scale IoT deployments — smart homes, smart cities, industrial automation, and precision agriculture — hundreds or thousands of heterogeneous devices from different manufacturers, using different communication protocols (Zigbee, BLE, Wi-Fi, 6LoWPAN), must be located and catalogued automatically, without manual intervention. Device discovery mechanisms make this possible.

4.1.1 Why Device Discovery is Needed

- Plug-and-play operation: New devices should be usable immediately after being powered on, without manual IP configuration.
- Dynamic network topology: IoT networks are highly dynamic — nodes join, leave, sleep, and rejoin frequently.
- Interoperability: Devices from different vendors must expose their capabilities in a common, discoverable format.
- Scalability: Automatic discovery avoids the administrative overhead of manually configuring thousands of nodes.
- Fault tolerance: Discovery protocols can detect when a device has disappeared from the network and trigger recovery actions.

4.1.2 Types of Device Discovery

Type	Description	Example Protocol
Passive Discovery	Device listens for periodic advertisement/beacon frames sent by neighbours.	BLE Advertising, Zigbee Beacon
Active Discovery	Device actively broadcasts a discovery/query request and waits for responses.	mDNS query, SSDP M-SEARCH
Centralized Discovery	A central discovery/registry server maintains the list of all devices.	CoRE Resource Directory
Distributed Discovery	Devices discover each other directly using peer-to-peer broadcast/multicast.	mDNS/DNS-SD, UPnP

4.1.3 Device Discovery Process

A generic device discovery process, applicable across most IoT protocol stacks, proceeds through the following stages:

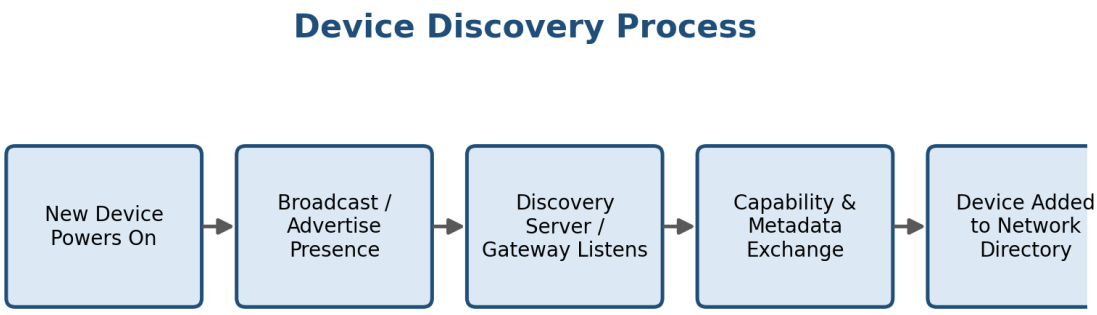

Fig 4.1 – Generic Device Discovery Process

- Advertisement / Beaconing: The new device broadcasts an advertisement packet containing its unique identifier and basic capability information.
- Listening: Gateways, coordinators, or discovery servers within radio range listen continuously or periodically for such advertisements.
- Capability Exchange: Once detected, a handshake occurs where the device shares detailed metadata — device type, supported protocols, sensor types, firmware version.
- Directory Update: The discovering entity (gateway/resource directory) adds an entry for the new device, making it visible to applications and other devices.

4.1.4 Discovery Probability and Timing – Key Formulae

In beacon-based discovery (e.g., BLE, Zigbee), the probability that a scanning device successfully detects an advertising device within a given scan window is an important design parameter. If the scanning device performs n independent scan attempts, and p is the probability that a single scan attempt overlaps with a beacon transmission, then the cumulative discovery probability is:

$$P_{discovery} = 1 - (1 - p)^n$$

Eq. 4.1 — Cumulative Discovery Probability

The expected number of scan attempts required for successful discovery (a geometric distribution) is given by:

$$E[N] = 1 / p$$

Eq. 4.2 — Expected Number of Attempts for Discovery

The expected discovery time can then be computed by multiplying the expected number of attempts by the beacon (advertising) interval, Tadv:

$$E[T_{discovery}] = E[N] \times T_{adv}$$

Eq. 4.3 — Expected Discovery Time

Worked Example

A BLE sensor advertises every Tadv = 100 ms. If the probability of a single successful overlap between the scan window and the advertisement is p = 0.25, then:

- Expected number of attempts, E[N] = 1 / 0.25 = 4 attempts
- Expected discovery time, E[T] = 4 × 100 ms = 400 ms
- Probability of discovery within 6 attempts: $P = 1 - (0.75)^6 \approx 1 - 0.178 = 0.822$ (about 82.2%)

4.2 Device Discovery Capabilities

"Discovery capabilities" refers to the specific technical functions that a discovery protocol or platform must provide to allow devices to be found, identified, and correctly interpreted by the rest of the network. These capabilities go beyond simply detecting that "something is there" — they must expose rich, structured information about the device.

4.2.1 Core Capabilities Expected of a Discovery System

- Identity Resolution: Providing a globally or locally unique identifier for each device (MAC address, UUID, URN, or IPv6 address).
- Capability Description: Exposing the resource types the device supports (e.g., temperature sensor, actuator, camera) typically using a resource description format such as CoRE Link Format or a JSON/JSON-LD descriptor.
- Service Advertisement: Advertising available services/endpoints (e.g., a device may expose /temperature, /humidity, /led as CoAP or REST resources).
- Reachability Information: Providing network-layer addressing information (IP address, port) so the device can actually be communicated with after discovery.
- Status/Liveness Reporting: Indicating whether the device is currently online, in sleep mode, or has failed, often via periodic keep-alive or heartbeat messages.
- Security Credentials Exchange: Sharing public keys, certificates, or pre-shared key identifiers needed to establish a secure session.

4.2.2 Common Discovery Protocols and Their Capabilities

Protocol	Layer	Key Capability
mDNS / DNS-SD	Application (Zero-configuration	Maps human-readable service names to IP

Protocol	Layer	Key Capability
	networking)	addresses without a DNS server.
SSDP (UPnP)	Application	Multicast-based discovery of UPnP devices and services on a LAN.
CoRE Resource Directory (RD)	Application (CoAP-based)	Central directory where constrained devices register their CoAP resources for lookup.
Bluetooth LE GAP/GATT	Link/Application	Advertising packets + GATT service/characteristic discovery.
Zigbee Device Object (ZDO)	Application	Network/service/binding discovery within a Zigbee PAN.

4.2.3 CoRE Link Format Example

The Constrained RESTful Environments (CoRE) Link Format (RFC 6690) is widely used to describe discoverable resources on a CoAP device. A typical response to a discovery request (GET /.well-known/core) looks like:

```
</sensors/temperature>;rt="temperature-c";if="sensor";ct=41, </sensors/humidity>;rt="humidity-c";if="sensor";ct=41, </actuators/led>;rt="led";if="actuator";ct=0
```

Here, rt indicates the resource type, if indicates the interface description, and ct indicates the Content-Format code understood by the resource.

4.3 Registering a Device

Once a device has been discovered, it must be registered with the managing platform — a gateway, network server, or cloud IoT platform — before it can participate in normal data exchange. Registration establishes a persistent, authenticated identity for the device inside the system's device registry/database, and allocates the resources (topics, endpoints, storage) the device will use.

4.3.1 Objectives of Device Registration

- Authenticate the device and confirm it is authorized to join the network.
- Create a permanent device profile/record containing metadata (type, owner, location, firmware version).
- Issue credentials (access tokens, certificates, or API keys) for future secure communication.
- Allocate application-level resources such as MQTT topics, database tables, or storage buckets.
- Enable device management operations — monitoring, firmware updates (OTA), remote configuration.

4.3.2 Step-by-Step Registration Process

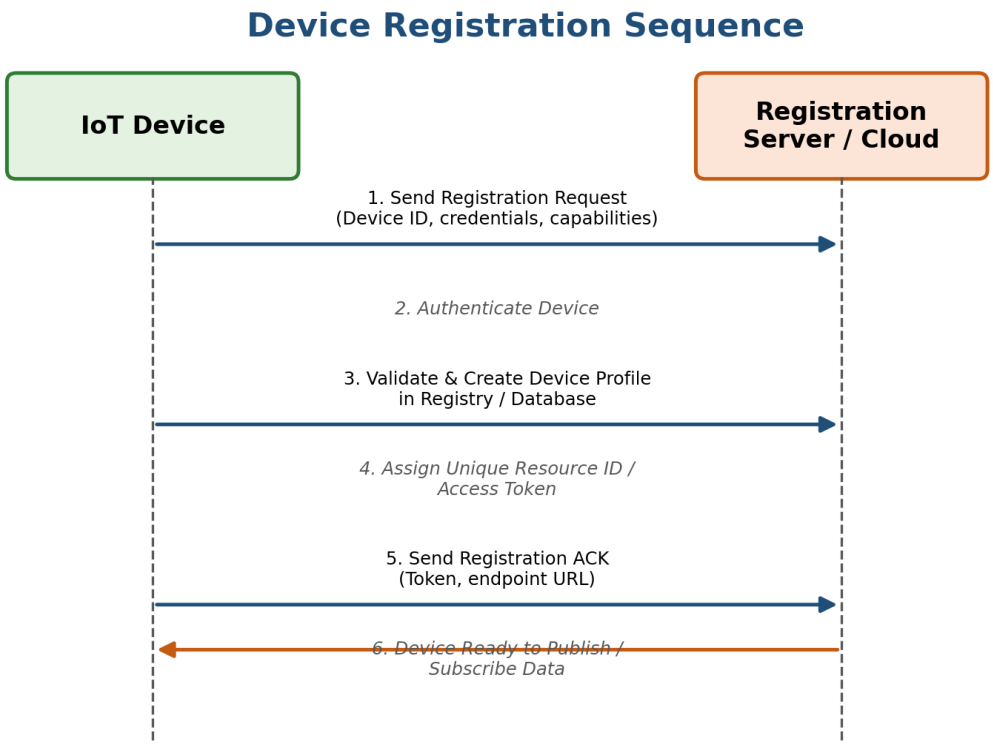

Fig 4.2 – Device Registration Sequence Diagram

- **Registration Request**: The device sends a registration request to the server, including its unique device ID, manufacturer credentials, and capability descriptor.
- **Authentication**: The server authenticates the request — this may use pre-shared keys, X.509 certificates, or OAuth 2.0 client-credential flows.
- **Validation and Profile Creation**: If authentication succeeds, the server validates the metadata and creates (or updates) a device profile entry in its registry database.
- **Resource Allocation**: A unique resource identifier is generated, and application resources (topics, buckets, dashboards) are provisioned.
- **Acknowledgement**: The server responds with an access token, refresh token, and the endpoint URL(s) the device should now use for data publishing.
- **Operational State**: The device transitions into an operational ("registered/active") state and begins normal telemetry exchange.

4.3.3 Typical Registration Payload (JSON Example)

```
{  "device_id": "SITAMS-TEMP-0231",  "device_type": "temperature_sensor",  "manufacturer": "Acme IoT",
"firmware_version": "2.1.0",  "protocol": "MQTT",  "public_key": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIB...",
"location": {"lat": 13.6288, "lon": 79.4192} }
```

4.3.4 Registration Scalability Formula

For a platform expected to onboard a large fleet of devices, the total registration time for N devices being registered sequentially through a single registration server with an average per-device registration time of Treg is:

$$T_{total} = N \times T_{reg}$$

Eq. 4.4 — Sequential Registration Time

If the server supports parallel registration through k concurrent worker threads/processes, the total time reduces approximately to:

$$T_{total} \approx (N \times T_{reg}) / k$$

Eq. 4.5 — Parallelized Registration Time

Worked Example: To register N = 10,000 devices at Treg = 150 ms each using k = 25 parallel workers: Ttotal ≈ (10,000 × 0.15) / 25 = 60 seconds, compared to 1500 seconds (25 minutes) if done sequentially.

4.4 Deregistering a Device

Deregistration is the controlled removal of a device's identity and associated resources from a network or cloud platform. It is triggered when a device is decommissioned, replaced, reported lost/stolen, or fails repeated liveness checks. Proper deregistration is essential for security (revoking credentials of a device that may be compromised) and for efficient resource management (freeing up storage, topics, and licenses).

4.4.1 Reasons for Deregistration

- Planned decommissioning at end-of-life of the device.
- Device replacement or hardware upgrade.
- Security compromise — credentials suspected to be leaked or device tampered with.
- Prolonged inactivity/loss of connectivity beyond a defined timeout threshold.
- Ownership transfer to a different user or organization.

4.4.2 Step-by-Step Deregistration Process

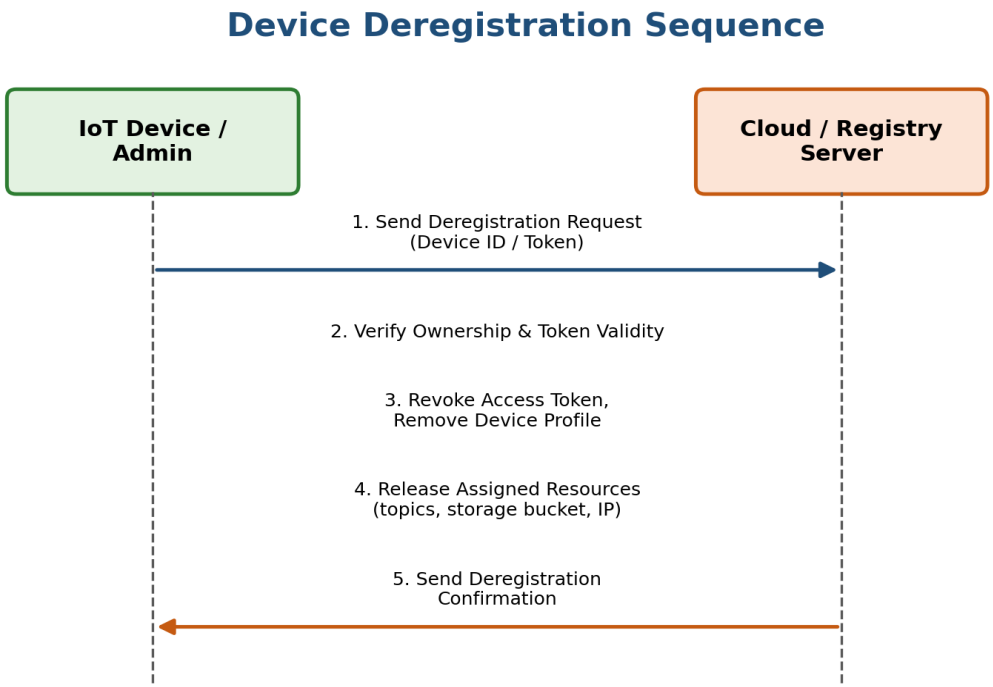

Fig 4.3 – Device Deregistration Sequence Diagram

- **Deregistration Request**: Initiated either by the device itself (graceful shutdown), by an administrator through a management console, or automatically by the platform after a timeout.
- **Ownership/Token Verification**: The platform verifies that the requester has the authority to deregister the specific device.
- **Revocation:** The device's access token/certificate is revoked immediately so it can no longer authenticate.
- **Profile and Resource Removal**: The device profile is deleted or archived, and allocated resources (MQTT topic, storage bucket, dashboard widget) are released.
- **Confirmation:** A deregistration confirmation is returned to the requester, and an audit log entry is created for traceability.

4.4.3 Graceful vs Forced Deregistration

Aspect	Graceful Deregistration	Forced Deregistration
Initiator	Device or authorized user	Platform / administrator, due to inactivity or security event
Data handling	Device flushes buffered data before leaving	May result in data loss for unsent telemetry
Token revocation	Immediate, coordinated with device	Immediate but unilateral

Aspect	Graceful Deregistration	Forced Deregistration
Typical trigger	End-of-life, planned maintenance	Missed heartbeat, security alert, policy violation

4.4.4 Liveness Timeout Formula

Platforms often use a heartbeat mechanism to detect silently disconnected devices and trigger automatic (forced) deregistration. If a device is expected to send a heartbeat every Th seconds, and the platform tolerates m missed heartbeats before declaring the device unreachable, the deregistration timeout is:

$$T_{timeout} = m \times T_h$$

Eq. 4.6 — Liveness / Deregistration Timeout

Worked Example: If Th = 30 seconds and the platform allows m = 3 missed heartbeats, then Ttimeout = 3 × 30 = 90 seconds. If no heartbeat is received within 90 seconds, the platform automatically initiates a forced deregistration workflow.

4.5 Introduction to Cloud Computing for IoT

IoT devices individually generate small amounts of data, but at scale — across thousands or millions of devices — the aggregate volume, velocity, and variety of data (often called the "3 Vs" of IoT data, along with veracity and value forming the "5 Vs") exceeds what can be stored or processed locally. Cloud computing provides the elastic, on-demand, pay-as-you-go infrastructure needed to ingest, store, process, and visualize this data.

4.5.1 Why IoT Needs the Cloud

- Elastic Scalability: Compute and storage resources can grow or shrink automatically with device count and data volume.
- Centralized Data Management: A single source of truth for data collected from geographically distributed devices.
- Advanced Analytics: Cloud platforms provide machine learning, big-data analytics, and visualization tools not feasible on constrained devices.
- Remote Device Management: Centralized dashboards for monitoring, configuring, and updating firmware on distributed devices.
- Cost Efficiency: Pay-as-you-go pricing avoids large upfront capital expenditure on data-center hardware.

4.5.2 NIST Definition of Cloud Computing — Five Essential Characteristics

- On-demand self-service — Users provision resources automatically without human interaction with the provider.
- Broad network access — Services accessible over the network via standard mechanisms (HTTP, REST).
- Resource pooling — Provider resources pooled to serve multiple consumers using a multi-tenant model.
- Rapid elasticity — Resources can be scaled out/in rapidly, sometimes automatically.

- Measured service — Resource usage is monitored, controlled, and reported, enabling pay-per-use billing.

4.6 Cloud Storage Models

Cloud storage refers to the models and mechanisms by which data generated by IoT devices is persisted in the cloud. The correct storage model must be chosen based on the type of data (structured/unstructured), the access pattern (frequent small writes vs bulk analytics reads), and cost constraints.

4.6.1 Types of Cloud Storage by Data Structure

Storage Type	Description	Typical IoT Use Case
Object Storage	Stores data as discrete objects (data + metadata + unique key) in a flat namespace; highly scalable.	Storing firmware images, raw sensor data dumps, images/video from IoT cameras (e.g., AWS S3, Azure Blob).
Block Storage	Data split into fixed-size blocks, attached to virtual machines like a hard disk.	Backing storage for databases/VMs running IoT stream-processing applications.
File Storage	Hierarchical file/folder structure accessed via network file protocols (NFS/SMB).	Shared configuration files, logs across multiple analytics VMs.
Time-Series Database	Optimized for storing/querying timestamped sensor readings.	Storing continuous telemetry — temperature, GPS coordinates, vibration data (e.g., InfluxDB, TimescaleDB).
NoSQL Document Store	Schema-less JSON-like documents.	Storing heterogeneous device metadata/configuration (e.g., MongoDB, DynamoDB).

4.6.2 Cloud Deployment Models

Independent of the storage type, the cloud infrastructure itself may be deployed in one of the following models:

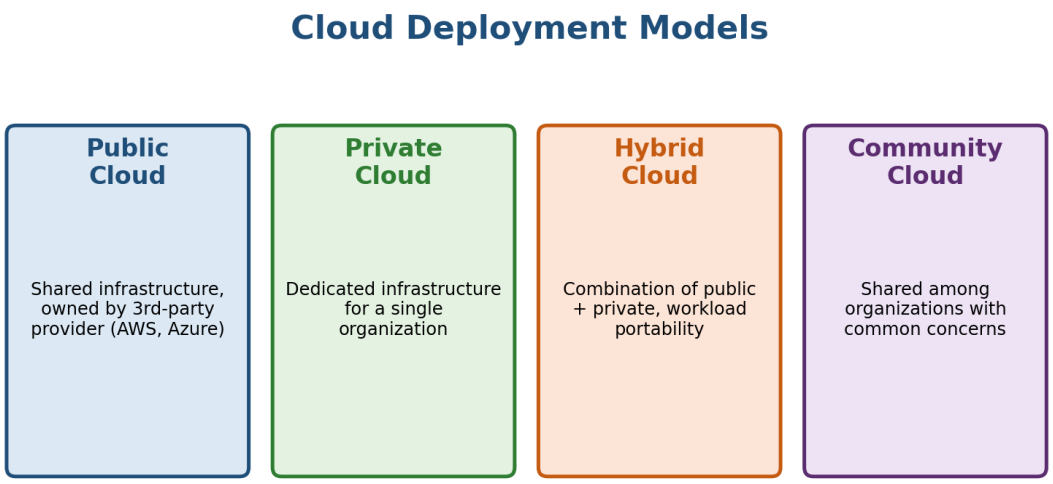

Fig 4.4 – Cloud Deployment Models

4.6.3 Storage Capacity Planning Formula

A fundamental design task is estimating how much cloud storage capacity is required for a given IoT deployment. If a single device generates a sample of size s bytes at a sampling frequency of f samples per second, the data rate per device is:

$$D = f \times s$$

Eq. 4.7 — Per-Device Data Rate (bytes/second)

For a deployment of N devices, each retained for a period of R seconds (retention window), the total required storage capacity is:

$$S_{total} = N \times D \times R = N \times f \times s \times R$$

Eq. 4.8 — Total Storage Capacity Requirement

Worked Example

Consider a smart-city deployment of N = 5,000 temperature sensors, each producing a sample of s = 64 bytes every 10 seconds (f = 0.1 samples/second), with a retention period of R = 30 days (2,592,000 seconds).

- Per-device data rate: D = 0.1 × 64 = 6.4 bytes/second
- Total storage: Stotal = 5000 × 6.4 × 2,592,000 = 82,944,000,000 bytes ≈ 82.94 GB

This calculation allows architects to correctly size storage buckets, choose pricing tiers, and estimate monthly cloud storage cost.

4.6.4 Cloud Storage Cost Formula

Given a provider's storage price rate Cs (cost per GB per month), and outbound data transfer price Ct (cost per GB transferred), the estimated monthly cost is:

$$Cost_{monthly} = (S_{total} \times C_s) + (S_{transfer} \times C_t)$$

Eq. 4.9 — Monthly Cloud Storage Cost

4.7 Cloud Service and Deployment Models

Cloud providers offer services at different levels of abstraction, commonly summarized as the "as-a-Service" stack. Understanding this stack is important because it determines how much of the infrastructure the IoT solution architect must manage versus how much is managed by the cloud provider.

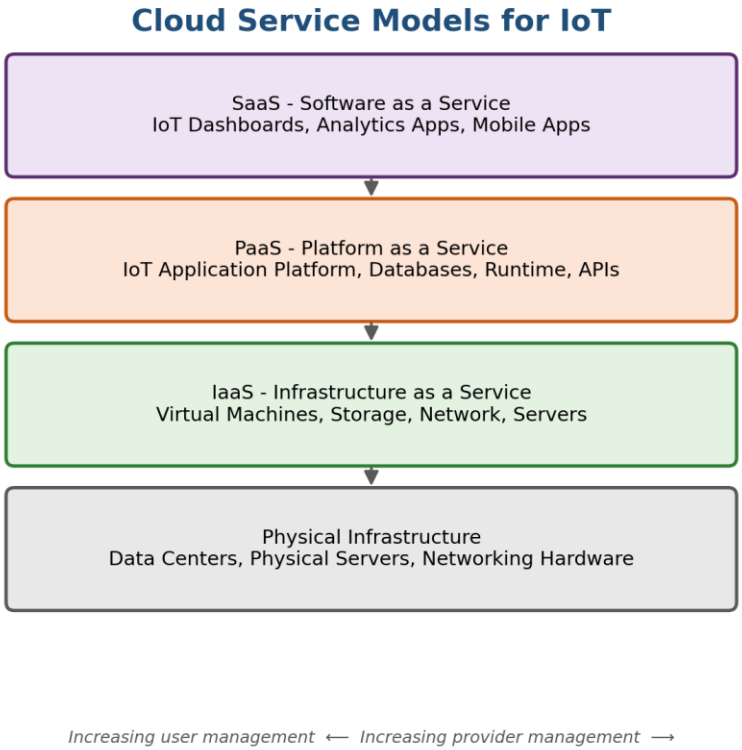

Fig 4.5 – Cloud Service Models Layered View

4.7.1 IaaS, PaaS and SaaS Compared

Model	What is Provided	What User Manages	IoT Example
IaaS	Virtual machines, storage, networking	OS, middleware, runtime, application, data	Renting VMs on AWS EC2/Azure VM to run an

Model	What is Provided	What User Manages	IoT Example
			MQTT broker
PaaS	Runtime environment, managed databases, APIs	Application code and data only	AWS IoT Core, Azure IoT Hub, Google Cloud IoT
SaaS	Fully managed application, ready to use	Configuration and data only	ThingSpeak dashboards, off-the-shelf IoT analytics apps

4.7.2 IoT Frameworks — ThingSpeak

ThingSpeak is an open, cloud-based IoT analytics platform (a SaaS/PaaS hybrid) that allows devices to send data to the cloud using simple REST or MQTT APIs, visualize live sensor data in configurable channels and fields, and execute MATLAB analytics code directly against stored data. Each ThingSpeak Channel can have up to 8 fields, and devices write to it using a simple HTTP GET/POST request containing a write API key.

```
GET https://api.thingspeak.com/update?api_key=YOUR_WRITE_KEY&field1=25.4&field2=61
```

4.7.3 Cloud Computing Cost-Benefit Formula

A simplified Total Cost of Ownership (TCO) comparison between on-premise infrastructure and cloud infrastructure over a period of T months can be expressed as:

$$TCO_{on\text{-}prem} = C_{capex} + (T \times C_{opex})$$

Eq. 4.10 — On-Premise TCO

$$TCO_{cloud} = T \times C_{cloud/month}$$

Eq. 4.11 — Cloud TCO

The cloud model is preferable whenever TCOcloud < TCOon-prem, which is typically true for early-stage or highly variable-load IoT deployments, since it avoids large capital expenditure (Ccapex).

4.8 Communication APIs for IoT–Cloud Integration

Communication APIs define the standardized way in which IoT devices, gateways, and cloud applications exchange data and commands. The choice of API/protocol depends on device constraints (power, memory), required message pattern (request/response vs publish/subscribe), and latency requirements.

4.8.1 REST-based APIs

Representational State Transfer (REST) is the most widely used architectural style for web-based communication APIs. It uses standard HTTP methods to operate on resources identified by URIs.

HTTP Method	Operation	Example
GET	Retrieve a resource's current state	GET /api/devices/1023
POST	Create a new resource	POST /api/devices
PUT	Update/replace an existing resource	PUT /api/devices/1023
DELETE	Remove a resource	DELETE /api/devices/1023

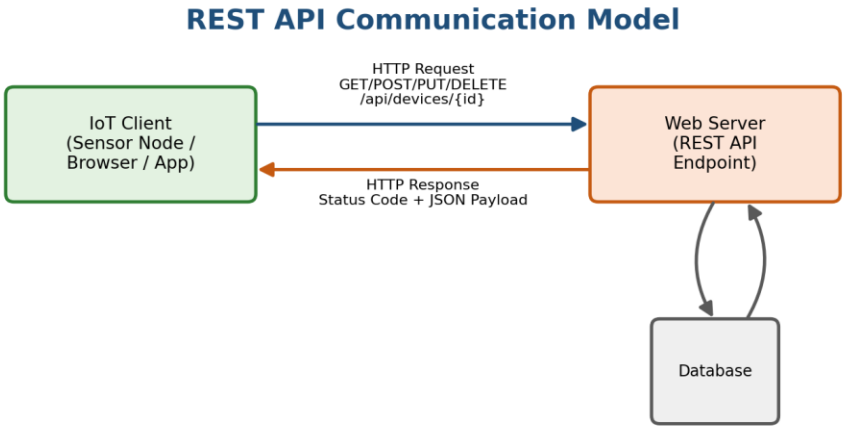

Fig 4.6 – REST API Request–Response Communication Model

4.8.2 WebSocket API

Unlike REST, which follows a request-response pattern, WebSocket provides a persistent, full-duplex connection between client and server, allowing the server to push data to the client without a new request each time — useful for real-time dashboards displaying live sensor readings.

4.8.3 Response Time / Latency Formula

The total response time experienced by an IoT client calling a cloud API is the sum of network propagation delay, transmission delay, server processing delay, and any queuing delay:

$$T_{response} = T_{prop} + T_{trans} + T_{proc} + T_{queue}$$

Eq. 4.12 — Total API Response Time

4.8.4 Little's Law for Server Queuing

Little's Law relates the average number of requests in a system (L), the average arrival rate of requests (λ), and the average time a request spends in the system (W):

$$L = \lambda \times W$$

Eq. 4.13 — Little's Law

Worked Example: If an IoT cloud API server receives an average of λ = 200 requests/second, and each request spends an average W = 0.05 seconds (50 ms) in the system, then the average number of requests present in the system at any time is L = 200 × 0.05 = 10 requests. This metric is used to size server thread pools and connection queues.

4.8.5 Throughput Formula

The throughput of a communication link or API endpoint is the amount of data successfully transferred per unit time:

$$Throughput = Total\ Data\ Transferred\ /\ Total\ Time\ Taken$$

Eq. 4.14 — Throughput

4.9 Web Server – Fundamentals

A web server is a software (and the hardware it runs on) that accepts requests from clients — typically over HTTP/HTTPS — and returns appropriate responses, which may be static files (HTML, images) or dynamically generated content (from a database query or sensor reading). In the context of IoT, both the cloud backend and, increasingly, the devices themselves can run web servers.

4.9.1 How a Web Server Works

- The client (browser, mobile app, or IoT device) establishes a TCP connection with the server, usually on port 80 (HTTP) or 443 (HTTPS).
- The client sends an HTTP request containing a method, a URI, headers, and (optionally) a body.
- The web server parses the request, and either serves a static resource directly or forwards the request to an application/backend layer (e.g., a REST API handler).
- The server returns an HTTP response with a status code (e.g., 200 OK, 404 Not Found, 500 Internal Server Error), headers, and a response body.
- The connection is closed or, in HTTP/1.1 and later, kept alive for subsequent requests.

4.9.2 Common Web Servers Used in IoT Backends

Web Server	Type	Typical Use
Apache HTTP Server	General-purpose, process/thread-based	Traditional dynamic web applications
Nginx	Event-driven, high-concurrency	Reverse proxy / load balancer in front of IoT APIs
Node.js (Express)	Asynchronous, event-loop based JavaScript runtime	Lightweight REST APIs for IoT dashboards
Flask / Django (Python)	WSGI-based application frameworks	Rapid prototyping of IoT backend services

4.9.3 HTTP Status Codes Relevant to IoT APIs

Code	Class	Meaning
200 OK	Success	Request processed successfully
201 Created	Success	New resource (e.g., new device) created successfully
400 Bad Request	Client Error	Malformed request syntax or invalid data
401 Unauthorized	Client Error	Missing or invalid authentication credentials
404 Not Found	Client Error	Requested resource/device does not exist
429 Too Many Requests	Client Error	Rate limit exceeded — common for constrained device APIs
500 Internal Server Error	Server Error	Unexpected server-side failure
503 Service Unavailable	Server Error	Server temporarily overloaded or under maintenance

4.10 Web Server for IoT (Constrained Devices)

Running a full-featured web server such as Apache directly on a resource-constrained IoT device (with as little as a few kilobytes of RAM) is impractical. Instead, IoT devices typically implement lightweight embedded web servers, or use a lightweight application protocol such as CoAP that mimics REST semantics while being optimized for constrained networks (6LoWPAN, LoRa).

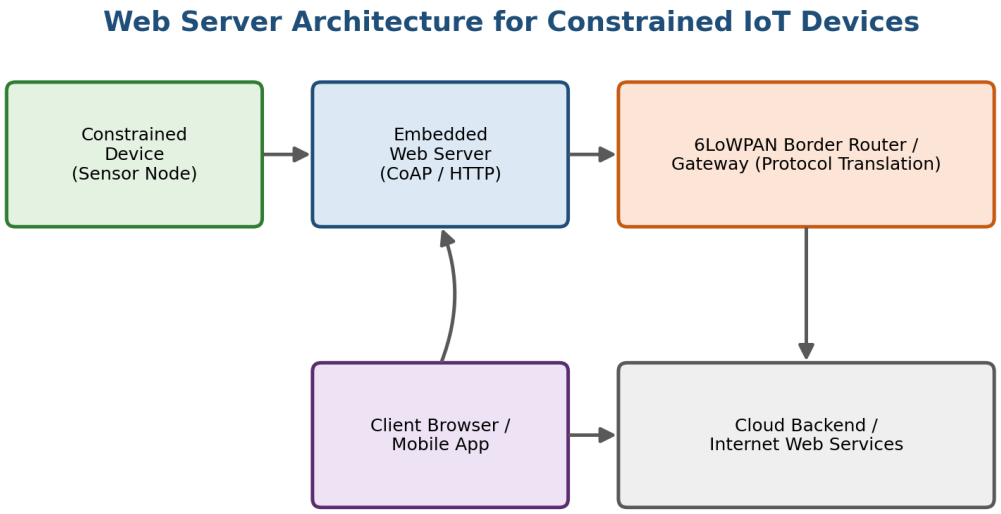

Fig 4.7 – Web Server Architecture for Constrained IoT Devices

4.10.1 Design Constraints for Embedded Web Servers

- Limited RAM/ROM: Some microcontrollers have less than 256 KB of RAM, so the web server's memory footprint must be minimal.
- Low Power Consumption: Devices are often battery-powered, so the server must minimize active radio and CPU time.
- Limited Bandwidth: Low-power wireless networks (802.15.4, LoRa) offer only tens of kbps, requiring compact message formats.
- Intermittent Connectivity: Devices may sleep for long periods, so the server design must tolerate delayed responses.

4.10.2 CoAP as a Lightweight Web Protocol

The Constrained Application Protocol (CoAP), defined in RFC 7252, is designed as a specialized web-transfer protocol for constrained nodes. It follows the REST model — using GET, POST, PUT, DELETE — but runs over UDP instead of TCP, and uses a compact binary header instead of verbose HTTP text headers, drastically reducing overhead.

Feature	HTTP	CoAP
Transport	TCP	UDP
Header size	Verbose, text-based (100s of bytes)	Compact binary header (4 bytes minimum)
Reliability	Provided by TCP	Optional, via CoAP Confirmable (CON) messages
Resource discovery	Not built-in	Built-in via /.well-known/core
Typical use	General web, cloud backends	Constrained sensor/actuator nodes

4.10.3 Overhead Reduction Formula

The relative overhead saving of CoAP compared to HTTP for a given payload can be expressed as a percentage reduction in total message size:

$$\%Reduction = [(H_{HTTP} - H_{CoAP}) / H_{HTTP}] \times 100$$

Eq. 4.15 — Header Overhead Reduction

Worked Example: If a typical HTTP request header is HHTTP = 300 bytes and the equivalent CoAP header is HCoAP = 20 bytes, the overhead reduction is: %Reduction = [(300 − 20)/300] × 100 = 93.3%. This is why CoAP is strongly preferred over HTTP for battery-powered, low-bandwidth sensor nodes.

4.10.4 Power/Energy Consumption Formula for Web Requests

The energy consumed by a constrained device to serve or send one web request/response can be approximated as the product of the average current drawn, the supply voltage, and the active (radio-on) time:

$$E = V \times I \times t$$

Eq. 4.16 — Energy Consumed per Transaction

Worked Example: A sensor node operating at V = 3.3 V, drawing I = 40 mA while its radio is active for t = 0.05 s to send a CoAP response, consumes E = 3.3 × 0.04 × 0.05 = 0.0066 J (6.6 mJ) per transaction. Minimizing t (through smaller CoAP headers) directly reduces energy consumption and extends battery life.

4.11 Case Study – Simple IoT Web Server

This case study demonstrates a minimal REST-style web server implemented using Python's Flask framework, representing a typical cloud-side endpoint that receives sensor readings from IoT devices and stores them, illustrating registration, data ingestion, and deregistration endpoints discussed earlier in this unit.

4.11.1 Sample Flask-based IoT Web Server Code

```
from flask import Flask, request, jsonify import uuid, time  app = Flask(__name__) devices = {}       # in-memory device registry readings = []       # in-memory data store  @app.route('/api/devices/register', methods=['POST']) def register_device():     data = request.get_json() device_id = str(uuid.uuid4())     devices[device_id] = {         "type": data.get("device_type"),         "registered_at": time.time(), "status": "active"     }     return jsonify({"device_id": device_id, "status": "registered"}), 201  @app.route('/api/devices/<device_id>/data', methods=['POST']) def push_data(device_id):     if device_id not in devices:         return jsonify({"error": "device not registered"}), 404 payload = request.get_json()     payload["device_id"] = device_id     payload["timestamp"] = time.time()     readings.append(payload) return jsonify({"status": "data received"}), 200  @app.route('/api/devices/<device_id>', methods=['DELETE']) def deregister_device(device_id):     if device_id in devices:         del devices[device_id]         return jsonify({"status": "deregistered"}), 200 return jsonify({"error": "device not found"}), 404  if __name__ == '__main__':     app.run(host='0.0.0.0', port=5000)
```

4.11.2 Explanation

- POST /api/devices/register implements the registration process from Section 4.3 — it creates a new device profile and returns a unique device_id.
- POST /api/devices/<id>/data implements the ongoing telemetry/data-ingestion channel used after successful registration.
- DELETE /api/devices/<id> implements the deregistration process from Section 4.4, removing the device profile.
- In a production cloud deployment, the in-memory dictionaries would be replaced with a persistent database (e.g., a NoSQL document store or time-series database, as discussed in Section 4.6).

4.12 Important Formulae – Summary

Eq.	Formula	Purpose
4.1	Pdiscovery = 1 − (1 − p)^n	Cumulative device discovery probability
4.2	E[N] = 1 / p	Expected number of scan attempts
4.3	E[Tdiscovery] = E[N] × Tadv	Expected discovery time
4.4	Ttotal = N × Treg	Sequential registration time
4.5	Ttotal ≈ (N × Treg) / k	Parallel registration time
4.6	Ttimeout = m × Th	Liveness / deregistration timeout
4.7	D = f × s	Per-device data rate
4.8	Stotal = N × f × s × R	Total storage capacity requirement
4.9	Costmonthly = (Stotal×Cs)+(Stransfer×Ct)	Monthly cloud storage cost

Eq.	Formula	Purpose
4.10	TCOon-prem = Ccapex + (T × Copex)	On-premise total cost of ownership
4.11	TCOcloud = T × Ccloud/month	Cloud total cost of ownership
4.12	Tresponse = Tprop+Ttrans+Tproc+Tqueue	Total API response time
4.13	L = λ × W	Little's Law – requests in system
4.14	Throughput = Data / Time	Communication link throughput
4.15	%Reduction = [(Hhttp−Hcoap)/Hhttp]×100	CoAP header overhead reduction
4.16	E = V × I × t	Energy consumed per transaction

4.13 Review Questions

Short Answer Questions

53. Define device discovery and explain why it is essential in IoT networks.
54. Differentiate between active and passive device discovery.
55. List the core capabilities a device discovery system must provide.
56. What is the CoRE Resource Directory and how does it support discovery?
57. Explain the step-by-step process of registering a new IoT device.
58. What triggers device deregistration? Distinguish graceful from forced deregistration.
59. Explain the five essential characteristics of cloud computing as defined by NIST.
60. Differentiate between object storage, block storage, and file storage with IoT examples.
61. Explain the IaaS, PaaS, and SaaS models with one IoT example each.
62. Describe how ThingSpeak can be used as an IoT cloud framework.
63. What is a communication API? Compare REST and WebSocket.
64. Explain how a web server processes an HTTP request, step by step.
65. Why is CoAP preferred over HTTP for constrained IoT devices?

Numerical / Problem-Solving Questions

66. A device advertises every 200 ms with a per-attempt discovery probability of 0.2. Calculate the expected number of attempts and expected discovery time.
67. Calculate the total storage required for 2,000 devices, each producing 32-byte samples every 5 seconds, retained for 15 days.
68. A registration server takes 200 ms per device. Compare the total registration time for 5,000 devices using (a) sequential processing and (b) 20 parallel workers.

69. If the heartbeat interval is 20 seconds and the platform allows 4 missed heartbeats, calculate the deregistration timeout.
70. An IoT cloud API server handles 150 requests/second with an average system time of 40 ms per request. Using Little's Law, find the average number of requests in the system.
71. If an HTTP header is 280 bytes and the equivalent CoAP header is 15 bytes, calculate the percentage overhead reduction.
72. A sensor operates at 3.0 V, draws 35 mA, and keeps its radio on for 0.06 s per transaction. Calculate the energy consumed per transaction.

Essay / Long Answer Questions

73. With a neat diagram, explain the complete lifecycle of an IoT device from discovery through registration to deregistration.
74. Discuss the different cloud storage models available for IoT data and justify which model is best suited for high-frequency sensor telemetry.
75. Explain the architecture of a web server designed for resource-constrained IoT devices, and discuss the trade-offs involved.
76. Compare and contrast cloud service models (IaaS, PaaS, SaaS) and cloud deployment models (public, private, hybrid, community) in the context of a smart-city IoT deployment.

UNIT – V

UAV IoT

Lecture Notes

Unit V – Syllabus Coverage

As per the course structure of 23CSE354C – Internet of Things, Unit V covers the following topics:

Table of Contents

5.1 Introduction to Unmanned Aerial Vehicles / Drones

An Unmanned Aerial Vehicle (UAV), popularly known as a drone, is an aircraft that operates without a human pilot on board. It is either controlled remotely by a human operator on the ground (Remote Piloted Aircraft, RPA) or flies autonomously using pre-programmed flight plans and onboard sensors/computers. A complete UAV system, including the aircraft, ground control station, communication link, and payload, is referred to as an Unmanned Aircraft System (UAS).

UAVs represent one of the most visible and rapidly growing branches of the Internet of Things because a modern drone is essentially a flying network of sensors, actuators, and communication modules — combining GPS, IMU, cameras, and wireless connectivity to sense its environment, make autonomous decisions, and relay data to the cloud in real time.

5.1.1 Brief History

- Early UAVs were developed primarily for military reconnaissance and target practice during the early-to-mid 20th century.
- Advances in miniaturized electronics, MEMS sensors, lithium-polymer batteries, and GPS in the 1990s–2000s enabled small, affordable civilian drones.
- Since the 2010s, multi-rotor drones combined with open-source autopilot software (ArduPilot, PX4) have driven explosive growth in commercial and hobbyist use.

5.1.2 Key Characteristics of a UAV

- Airframe: The physical structure supporting all components (fixed-wing, rotary-wing, or hybrid).
- Propulsion System: Motors, propellers, and Electronic Speed Controllers (ESCs) that generate thrust.
- Autopilot / Flight Controller: The onboard computer that reads sensor data and computes control commands.
- Sensors: GPS, IMU, barometer, compass, ultrasonic/LiDAR range finders, and cameras.
- Communication Link: RF telemetry, Wi-Fi, or cellular link connecting the UAV to a ground control station or cloud.
- Payload: Task-specific equipment — cameras, sensors, delivery packages, or spraying equipment.

5.1.3 Degrees of Freedom

A UAV in flight is described using six degrees of freedom (6-DOF): three translational (surge/x, sway/y, heave/z) and three rotational (roll, pitch, yaw) about its centre of gravity. All flight control algorithms are ultimately designed to regulate these six quantities.

5.2 Drone Types

UAVs are broadly classified based on their aerodynamic configuration into four major categories, each offering different trade-offs between endurance, payload capacity, hovering ability, and mechanical complexity.

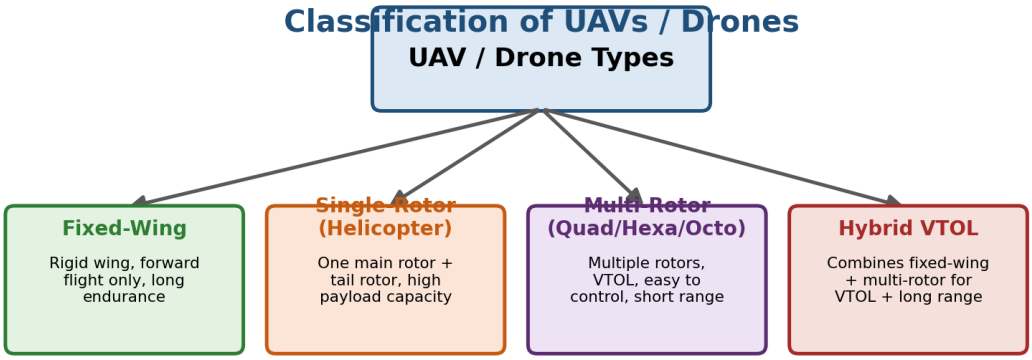

Fig 5.1 – Classification of UAVs / Drones

Type	Lift Mechanism	Advantages	Limitations
Fixed-Wing	Aerodynamic lift from a rigid wing, requires forward airspeed	Long endurance, large area coverage, energy efficient	Requires runway/launcher, cannot hover
Single-Rotor (Helicopter)	One large main rotor + tail rotor for anti-torque	Higher payload capacity, efficient hover	Mechanically complex, higher maintenance
Multi-Rotor (Quad/Hexa/Octocopter)	Multiple fixed-pitch rotors	VTOL, easy to control, stable hover, low cost	Short flight time, limited range and payload
Hybrid VTOL	Combines fixed-wing cruise with rotor-based vertical lift	VTOL capability + long-range fixed-wing efficiency	Complex design, higher cost, heavier

5.2.1 Multi-Rotor Sub-Types

- Tricopter — 3 rotors, requires a servo-actuated tail rotor for yaw control.
- Quadcopter — 4 rotors, the most common configuration, simple and stable.
- Hexacopter — 6 rotors, offers redundancy (can survive single motor failure).
- Octocopter — 8 rotors, used for heavy-lift and professional cinematography applications.

5.3 Applications of UAVs

Drones have moved rapidly from purely military use to a wide array of civil and environmental applications, driven by falling sensor costs and improvements in autonomy.

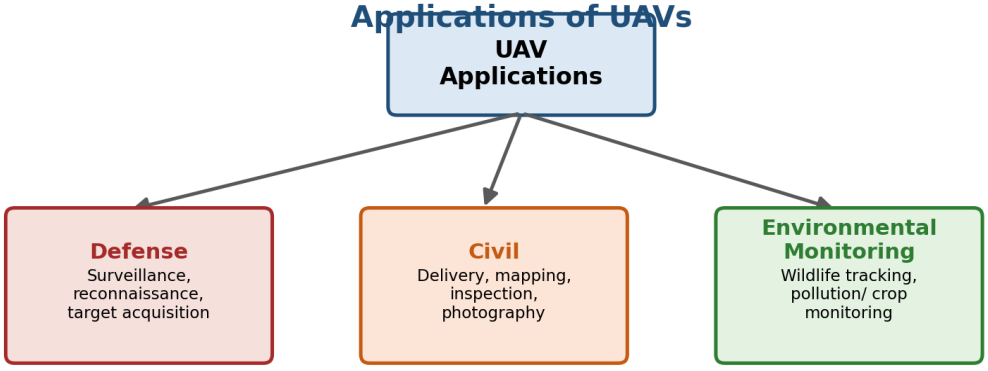

Fig 5.2 – Applications of UAVs

5.3.1 Defense Applications

- Intelligence, Surveillance, and Reconnaissance (ISR) missions over hostile or inaccessible terrain.
- Target acquisition and precision-guided munitions delivery.
- Border patrol and perimeter security monitoring.
- Combat search and rescue support.

5.3.2 Civil Applications

- Last-mile parcel delivery in urban and rural areas.
- Aerial photography, videography, and cinematography.
- Infrastructure inspection — power lines, bridges, wind turbines, pipelines.
- Precision agriculture — crop health monitoring, targeted spraying, yield estimation.
- Surveying and mapping using photogrammetry and LiDAR.
- Traffic monitoring and disaster-response damage assessment.

5.3.3 Environmental Monitoring Applications

- Wildlife population tracking and anti-poaching patrols.
- Forest fire detection and post-fire damage assessment.
- Air and water quality sampling in hard-to-reach locations.
- Glacier, coastline, and deforestation monitoring using time-series aerial imagery.

5.4 UAV Elements — Frame, Arms, Motors, ESC, Propellers, Battery

Every multi-rotor UAV is built from a common set of mechanical and electronic building blocks. Understanding each element and how it interacts with the others is essential to designing, building, or troubleshooting a UAV system.

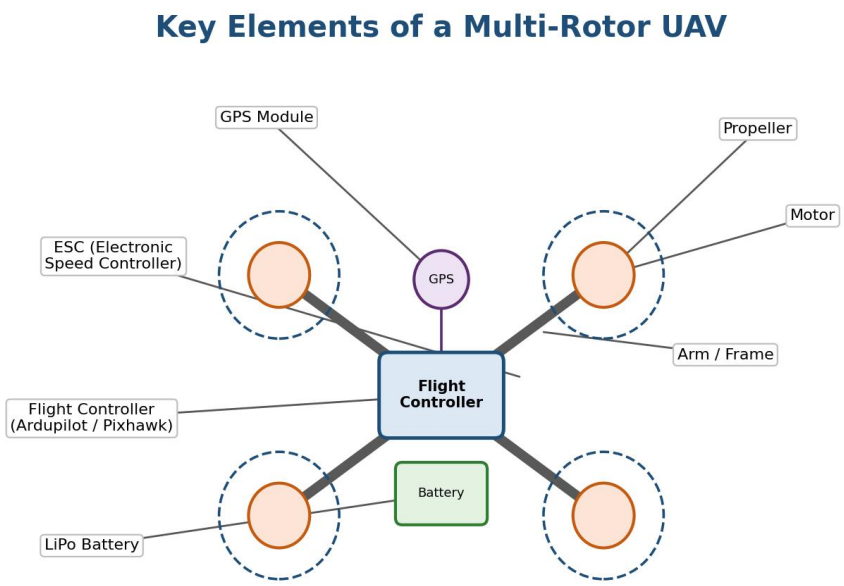

Fig 5.3 – Key Elements of a Multi-Rotor UAV

5.4.1 Frame and Arms

The frame (or chassis) is the structural backbone of the UAV, typically made from lightweight yet rigid materials such as carbon fibre, aluminium, or engineering plastics. The arms extend outward from the central frame and hold the motors at each end. Frame design directly affects the UAV's stiffness, vibration characteristics, and the

maximum propeller size it can accommodate. The distance between two diagonally opposite motors is called the wheelbase, and it determines the propeller clearance and overall stability.

5.4.2 Motors (BLDC Motors)

Multi-rotor UAVs almost universally use Brushless DC (BLDC) motors due to their high efficiency, favourable power-to-weight ratio, and long service life compared to brushed motors. A motor's speed constant, denoted Kv (RPM per volt), specifies how fast the motor will spin (unloaded) per volt applied:

$$RPM = K_v \times V$$

Eq. 5.1 — Motor Speed vs Applied Voltage

Worked Example: A motor rated at Kv = 900 RPM/V, powered by an 11.1 V (3S) LiPo battery, will spin at approximately RPM = 900 × 11.1 = 9,990 RPM under no load.

5.4.3 Electronic Speed Controller (ESC)

The ESC is the electronic circuit that converts the DC power from the battery into a three-phase AC signal to precisely control the speed and direction of a BLDC motor, based on a PWM control signal received from the flight controller.

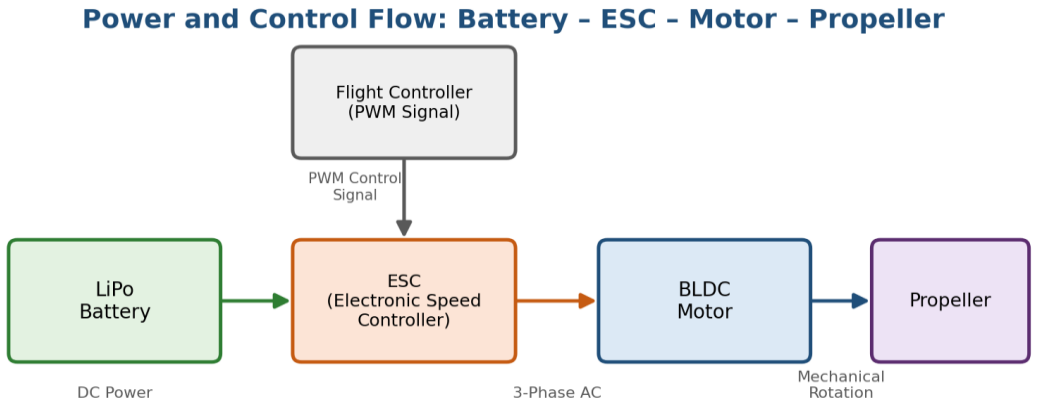

Fig 5.4 – Power and Control Flow: Battery – ESC – Motor – Propeller

- Receives a PWM signal (typically 1000–2000 microseconds pulse width) from the flight controller.
- Regulates the current delivered to each motor phase winding using electronic commutation.
- Provides Battery Elimination Circuit (BEC) output in some designs to power the flight controller/receiver.
- Common ESC protocols include standard PWM, OneShot125, Multishot, and the modern digital DShot protocol.

5.4.4 Propellers

Propellers convert the motor's rotational (mechanical) energy into thrust by accelerating air downward (Newton's third law). Propellers are specified by two numbers — diameter × pitch (in inches), e.g., a 10×4.5 propeller has a 10-inch diameter and a 4.5-inch theoretical pitch (forward distance travelled per revolution in an ideal fluid).

5.4.5 Propeller Thrust and Power Formulae

The static thrust generated by a propeller can be estimated using the propeller thrust coefficient equation:

$$T = C_T \times \rho \times n^2 \times D^4$$

Eq. 5.2 — Propeller Thrust

where CT is the thrust coefficient (dimensionless, depends on propeller design), ρ is air density (kg/m³), n is rotational speed (revolutions per second), and D is propeller diameter (m). The corresponding shaft power required is:

$$P = C_P \times \rho \times n^3 \times D^5$$

Eq. 5.3 — Propeller Power Required

5.4.6 Battery (LiPo) and Flight Time

Lithium Polymer (LiPo) batteries are preferred in UAVs for their high energy density and high discharge (C) rating. The flight time achievable is directly related to battery capacity and average current draw of the UAV:

$$t_{flight} = (Capacity_{(mAh)} \times 60) / I_{avg\ (mA)}$$

Eq. 5.4 — Flight Time Estimation

Worked Example: A UAV powered by a 5000 mAh battery, drawing an average current of 15,000 mA (15 A) during flight, gives an estimated flight time of: tflight = (5000 × 60) / 15000 = 20 minutes.

5.4.7 Thrust-to-Weight Ratio

A critical design parameter determining whether a UAV can lift off, hover stably, and manoeuvre aggressively is the thrust-to-weight ratio (TWR):

$$TWR = T_{total} / W_{total}$$

Eq. 5.5 — Thrust-to-Weight Ratio

A TWR of exactly 1.0 allows level hover; a TWR of 2.0 or higher is generally recommended for stable, responsive flight with reserve thrust for manoeuvring and wind disturbance rejection.

5.5 GPS in UAV Navigation

The Global Positioning System (GPS) is a satellite-based navigation system providing a UAV with its absolute position (latitude, longitude, altitude) and velocity anywhere on Earth with a clear sky view. GPS is essential for autonomous waypoint navigation, geofencing, return-to-home functionality, and position-hold flight modes.

5.5.1 Working Principle — Trilateration

A GPS receiver calculates its position by measuring the time delay between signal transmission from at least four satellites and their reception, converting this time delay into a distance (pseudorange) using the known speed of light:

$$d = c \times \Delta t$$

Eq. 5.6 — Pseudorange from Signal Travel Time

where c is the speed of light (≈3×10^8 m/s) and Δt is the measured signal travel time. With distances from four or more satellites of known position, the receiver's 3D position and clock offset are solved using trilateration (a system of simultaneous equations).

5.5.2 GPS Accuracy Factors

- Number of visible satellites — more satellites generally improve position accuracy.
- Dilution of Precision (DOP) — a measure of satellite geometry quality; lower DOP indicates better accuracy.
- Multipath effects — signal reflections off buildings/terrain can introduce position errors.
- Differential GPS (DGPS) and Real-Time Kinematic (RTK) corrections — used to improve accuracy from metres down to centimetres for precision applications like surveying.

5.5.3 Role of GPS in Flight Modes

Flight Mode	Role of GPS
Loiter / Position Hold	Maintains a fixed GPS position, compensating for wind drift
Auto (Mission)	Navigates automatically between pre-defined GPS waypoints
Return to Launch (RTL)	Uses stored home GPS coordinates to autonomously return and land
Geofencing	Restricts flight within a virtual GPS-defined boundary

5.6 IMU (Inertial Measurement Unit)

The Inertial Measurement Unit (IMU) is a critical onboard sensor package, typically combining a 3-axis accelerometer, a 3-axis gyroscope, and often a 3-axis magnetometer, that allows the flight controller to estimate the UAV's orientation (attitude) and angular rate at very high update frequencies — essential for real-time stabilization, since GPS alone updates too slowly (typically 5–10 Hz) for stable flight control.

5.6.1 IMU Components

Sensor	Measures	Role
Accelerometer	Linear acceleration along X, Y, Z axes	Determines tilt relative to gravity, detects vibration
Gyroscope	Angular velocity about X, Y, Z axes (roll, pitch, yaw rates)	Provides fast, short-term orientation change data
Magnetometer	Magnetic field strength along X, Y, Z axes	Provides absolute heading reference (compass)

5.6.2 Sensor Fusion — Combining GPS and IMU

Neither GPS nor IMU alone is sufficient for reliable UAV navigation: GPS is accurate over long time scales but slow and can be lost (e.g., under bridges or indoors); IMU is fast but accumulates drift error over time due to bias and integration error. Modern flight controllers therefore fuse both sources using an Extended Kalman Filter (EKF) or a simpler complementary filter.

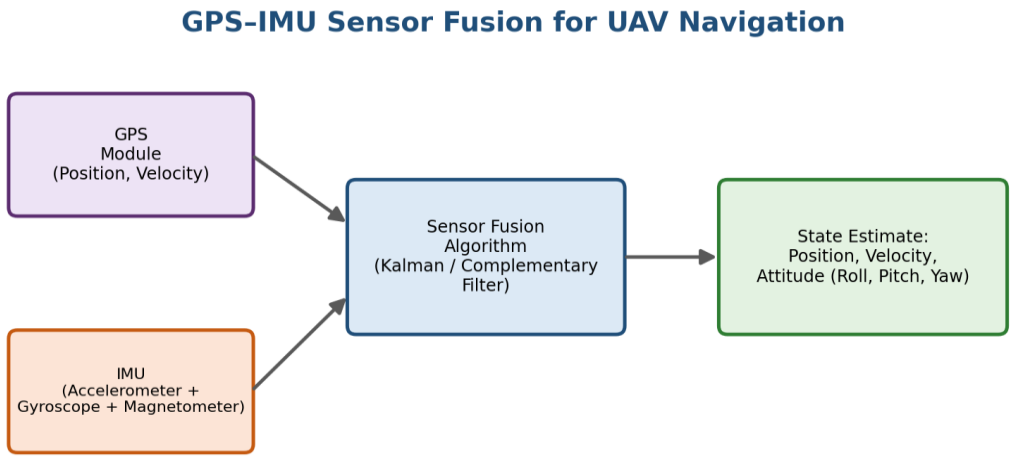

Fig 5.5 – GPS–IMU Sensor Fusion for UAV Navigation

5.6.3 Complementary Filter Formula

A simple and computationally light approach to fusing accelerometer and gyroscope data for attitude estimation is the complementary filter, which blends a high-pass filtered gyroscope angle with a low-pass filtered accelerometer angle:

$$\theta = \alpha(\theta_{gyro}) + (1 - \alpha)(\theta_{accel})$$

Eq. 5.7 — Complementary Filter for Attitude Estimation

where α (typically ≈0.98) is a weighting constant close to 1, giving greater short-term trust to the fast-responding gyroscope while the slower accelerometer term corrects long-term gyroscope drift.

5.6.4 Gyroscope Angle from Angular Rate

The gyroscope-based angle estimate is obtained by numerically integrating the measured angular rate ω over the sample time interval Δt:

$$\theta_{gyro} = \theta_{previous} + (\omega \times \Delta t)$$

Eq. 5.8 — Gyroscope Angle Integration

5.7 Ultrasonic Sensors in UAVs

Ultrasonic (sonar) sensors are used in UAVs primarily for precise altitude measurement at low heights (typically below 5–8 metres) and for short-range obstacle detection, complementing the barometer (which is less accurate close to the ground due to ground-effect pressure changes) and GPS (which lacks the resolution for fine height control during landing).

5.7.1 Working Principle

An ultrasonic sensor emits a short burst of high-frequency sound (typically 40 kHz, above human hearing range) and measures the time taken for the echo to return after reflecting off the ground or an obstacle. Distance is calculated from the measured round-trip time and the speed of sound in air:

$$d = (v_{sound} \times t) / 2$$

Eq. 5.9 — Ultrasonic Distance Measurement

where vsound is the speed of sound in air (≈343 m/s at 20 °C), t is the total round-trip travel time of the pulse, and division by 2 accounts for the pulse travelling to the object and back.

Worked Example: If the measured round-trip time for an echo is t = 6 milliseconds (0.006 s), the distance to the ground is: d = (343 × 0.006) / 2 = 1.029 m.

5.7.2 Advantages and Limitations

Advantages	Limitations
Low cost and low power consumption	Limited range (typically a few metres to ~8 m)
High accuracy at short range for precision landing	Affected by soft/absorptive surfaces (e.g., grass, foliage)
Works regardless of lighting conditions (unlike optical sensors)	Susceptible to noise from propeller wash and wind

5.8 UAV Flight Dynamics and Control

A multi-rotor UAV achieves controlled flight purely by varying the relative rotational speeds of its motors. Understanding how differential thrust produces roll, pitch, and yaw motion is fundamental to UAV control system design.

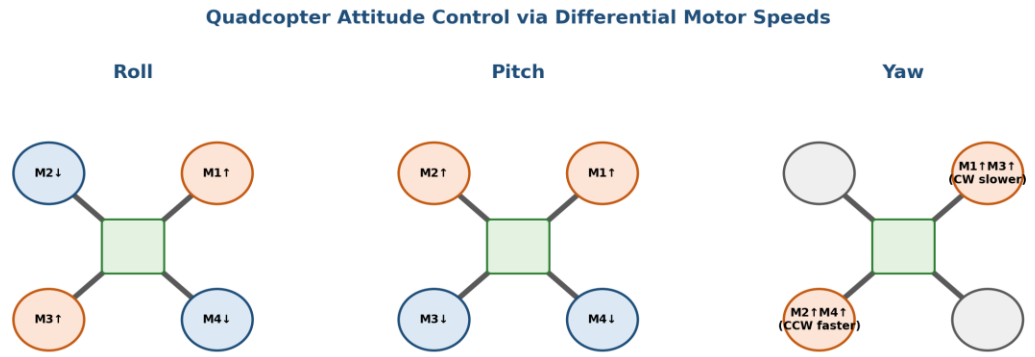

Fig 5.6 – Quadcopter Attitude Control via Differential Motor Speeds

- Roll (rotation about the longitudinal/front-back axis): achieved by increasing thrust on motors on one side while decreasing it on the opposite side.
- Pitch (rotation about the lateral/left-right axis): achieved by increasing thrust on front (or rear) motors while decreasing thrust on the opposite pair.
- Yaw (rotation about the vertical axis): achieved by exploiting the reaction torque of the propellers — increasing speed of the clockwise-spinning motor pair relative to the counter-clockwise pair (or vice versa), since adjacent motors spin in opposite directions to cancel net torque in steady hover.
- Throttle (vertical/altitude control): achieved by increasing or decreasing the average speed of all four motors simultaneously.

5.8.1 PID Control Loop

Flight controllers use a Proportional-Integral-Derivative (PID) control loop for each axis (roll, pitch, yaw) to compute the correct motor output needed to minimize the error between the desired (setpoint) attitude and the current measured attitude:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d (de(t)/dt)$$

Eq. 5.10 — PID Controller Output

where e(t) is the instantaneous error (setpoint − measured value), Kp, Ki, and Kd are the proportional, integral, and derivative gain constants respectively, and u(t) is the resulting control output (added/subtracted from baseline motor throttle).

5.8.2 Total Torque Equation for Roll/Pitch

The net rolling (or pitching) torque produced about the UAV's centre of gravity is the product of the thrust differential between opposite motor pairs and the moment arm (half the wheelbase length, L):

$$\tau = (T_1 - T_2) \times L$$

Eq. 5.11 — Roll/Pitch Torque

5.8.3 Range Equation for Fixed-Wing UAVs

For fixed-wing UAVs, the maximum achievable range is estimated using the Breguet range equation, which relates range to battery energy, aerodynamic efficiency, and aircraft weight:

$$R = (\eta \times E_{battery} \times (L/D)) / (W \times g)$$

Eq. 5.12 — Simplified Range Equation (Electric Fixed-Wing UAV)

where η is overall propulsive efficiency, Ebattery is the usable battery energy (Joules), L/D is the lift-to-drag ratio (aerodynamic efficiency), W is aircraft weight (kg), and g is gravitational acceleration (9.81 m/s²).

5.9 UAV Software — ArduPilot

ArduPilot is one of the most widely used open-source autopilot software suites, supporting multi-rotors (ArduCopter), fixed-wing aircraft (ArduPlane), ground rovers (ArduRover), and submarines (ArduSub). It runs on dedicated flight-controller hardware (such as the Pixhawk series) and provides a complete stack for sensor fusion, flight-mode management, navigation, and control.

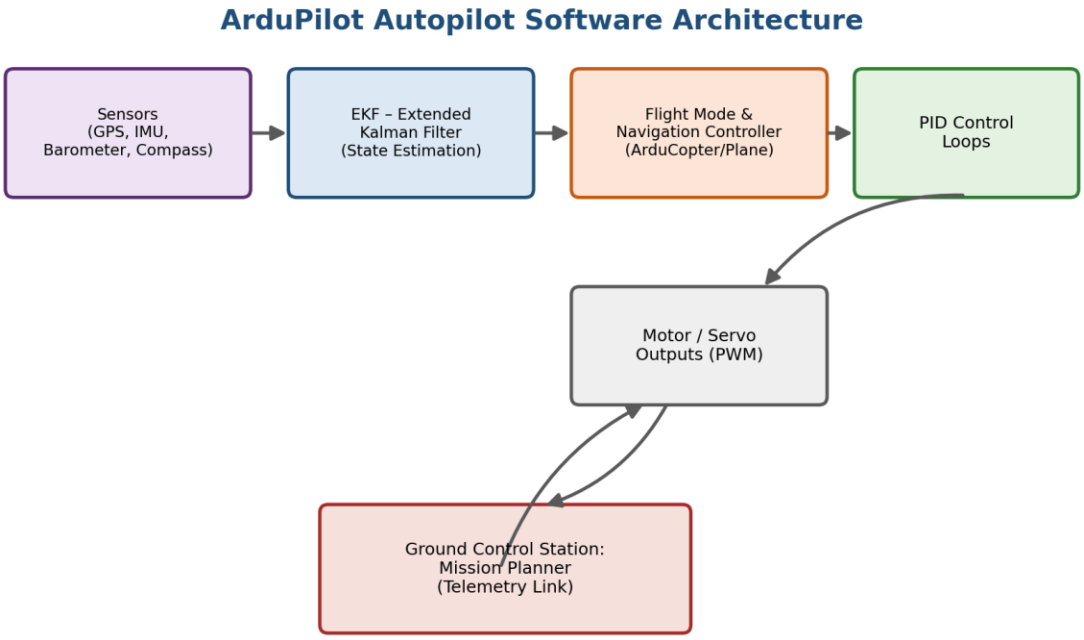

Fig 5.7 – ArduPilot Autopilot Software Architecture

5.9.1 Key Functional Blocks

- Sensor Drivers: Interface with GPS, IMU, barometer, compass, and rangefinder hardware.
- EKF (Extended Kalman Filter): Fuses all sensor data into a single, best estimate of position, velocity, and attitude.
- Flight Mode Manager: Implements dozens of flight modes (Stabilize, Loiter, Auto, RTL, Guided, and more), each with different levels of autonomy.
- Navigation Controller: Converts mission waypoints or pilot stick input into desired attitude/velocity targets.
- PID Control Loops: Compute the final actuator (motor/servo) commands needed to track the desired attitude and rates.
- MAVLink Communication: A lightweight messaging protocol used to communicate with ground control stations like Mission Planner.

5.9.2 Common ArduCopter Flight Modes

Flight Mode	Description
Stabilize	Pilot controls angle directly; auto-levels roll/pitch but pilot manages altitude
AltHold	Maintains altitude automatically using barometer/rangefinder, pilot controls horizontal movement

Flight Mode	Description
Loiter	Maintains both position (GPS) and altitude automatically; pilot can nudge position
Auto	Fully autonomous flight following a pre-loaded mission of GPS waypoints
RTL (Return to Launch)	Autonomously returns to and lands at the home/launch location
Guided	Accepts real-time position/velocity commands from a companion computer or GCS

5.10 Mission Planner — Ground Control Station

Mission Planner is a free, open-source Ground Control Station (GCS) application, developed for use with ArduPilot, that runs on a laptop or ground computer and communicates with the UAV over a telemetry radio link using the MAVLink protocol.

5.10.1 Key Functions of Mission Planner

- Mission Planning: Drawing and uploading autonomous flight paths (waypoints) on a satellite map.
- Real-Time Telemetry Display: Showing live UAV position, altitude, speed, battery voltage, and sensor health on a Heads-Up Display (HUD).
- Parameter Configuration: Tuning PID gains, failsafe behaviours, and hardware-specific settings.
- Firmware Installation: Flashing the latest ArduPilot firmware onto the flight controller board.
- Log Analysis: Downloading and analyzing detailed flight logs for post-flight diagnostics and PID tuning.
- Geofencing and Failsafe Setup: Defining safe flight boundaries and configuring automatic responses to signal loss or low battery.

5.10.2 Telemetry Link Budget

The reliable operating range of the telemetry link between the UAV and Mission Planner is governed by the Friis free-space transmission equation, which relates received power to transmitted power, antenna gains, wavelength, and distance:

$$P_r = P_t G_t G_r (\lambda / 4\pi d)^2$$

Eq. 5.13 — Friis Transmission Equation (Link Budget)

where Pr is received power, Pt is transmitted power, Gt and Gr are transmitter/receiver antenna gains, λ is the signal wavelength, and d is the distance between transmitter and receiver. This equation is used to estimate the maximum achievable telemetry/control range for a given radio system.

5.11 Internet of Drones (IoD)

The Internet of Drones (IoD) is an emerging paradigm that extends IoT principles to networks of UAVs, providing a layered architecture for coordinated access, navigation, and control of multiple drones sharing a common airspace and communication infrastructure — analogous to how the Internet of Things connects everyday objects, but adapted for the unique mobility, altitude, and safety requirements of aerial vehicles.

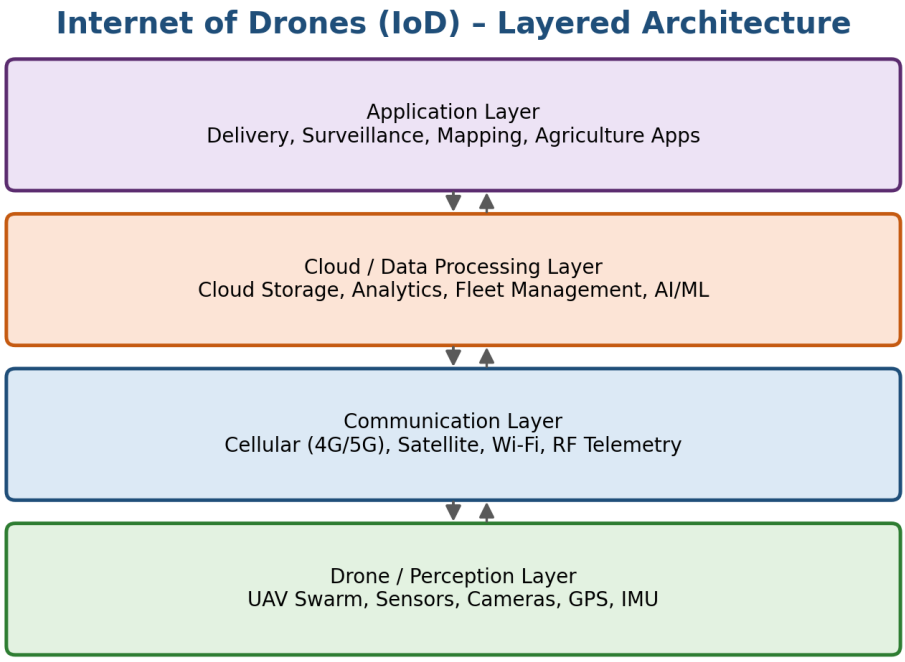

Fig 5.8 – Internet of Drones (IoD): Layered Architecture

5.11.1 IoD Layers

Layer	Function
Drone / Perception Layer	The physical UAV fleet with onboard sensors (GPS, IMU, cameras) collecting data and executing flight commands
Communication Layer	Provides connectivity between drones and ground/cloud systems using cellular (4G/5G), satellite links, Wi-Fi, or dedicated RF telemetry
Cloud / Data Processing Layer	Aggregates data from multiple drones, performs analytics, fleet management, path planning, and AI-based decision-making

Layer	Function
Application Layer	End-user services such as delivery tracking, surveillance dashboards, precision-agriculture reports, and mapping outputs

5.11.2 Key Challenges in IoD

- Airspace management and collision avoidance among multiple drones sharing the same volume of airspace.
- Security and authentication to prevent GPS spoofing, signal jamming, or unauthorized drone hijacking.
- Privacy concerns related to aerial surveillance and camera-equipped drones over populated areas.
- Regulatory compliance with national aviation authorities (e.g., DGCA in India, FAA in the USA) regarding altitude limits, no-fly zones, and remote pilot licensing.
- Energy and endurance constraints limiting continuous multi-drone operations without frequent recharging or battery swaps.

5.12 Case Study — FlytBase

FlytBase is a commercial software platform that enables fully autonomous, remote drone operations at scale, commonly used for infrastructure inspection, security surveillance, and industrial monitoring using "drone-in-a-box" docking stations that allow drones to operate without an on-site human pilot.

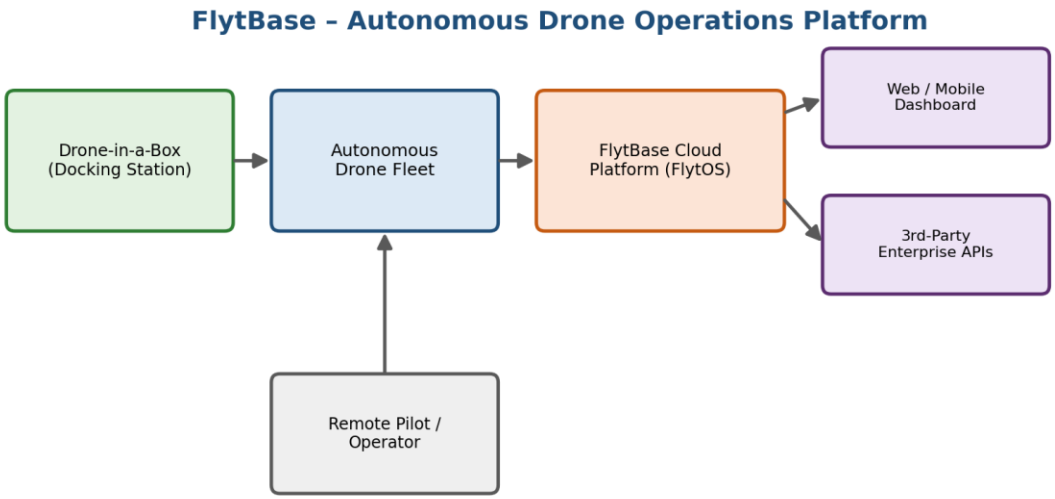

Fig 5.9 – FlytBase Autonomous Drone Operations Platform

5.12.1 Key Capabilities

- Beyond Visual Line of Sight (BVLOS) autonomous operations from a centralized remote command centre.
- Automated drone-in-a-box docking stations that handle takeoff, landing, and battery charging without manual intervention.
- Cloud-based fleet management dashboard for monitoring multiple drones across multiple sites simultaneously.
- Open APIs and SDKs (FlytOS) allowing integration with third-party enterprise systems (asset management, GIS, analytics platforms).
- AI-powered video/image analytics for automated defect detection, intrusion alerts, and inventory counting.

5.12.2 Typical Workflow

- A scheduled or event-triggered mission is initiated remotely through the FlytBase cloud dashboard.
- The docking station automatically opens, and the drone undocks, powers on, and begins the pre-programmed mission autonomously.
- The drone streams live telemetry, video, and sensor data back to the FlytBase cloud platform over a cellular or Wi-Fi link.
- On mission completion, the drone autonomously returns and lands precisely on the docking station, which recharges its battery for the next mission.
- Collected data is processed (analytics, defect detection) and made available to enterprise users through dashboards or third-party integrations.

5.12.3 Relevance to IoT and IoD Concepts

The FlytBase case study exemplifies nearly every concept covered in this unit and the previous unit: device discovery and registration of drones joining the fleet, cloud storage of telemetry and video data, REST/web APIs for third-party integration, and a layered IoD architecture connecting the physical drone layer to cloud analytics and end-user applications.

5.13 Important Formulae — Summary

Eq.	Formula	Purpose
5.1	$RPM = Kv \times V$	Motor speed vs applied voltage
5.2	$T = CT \times \rho \times n^2 \times D^4$	Propeller static thrust
5.3	$P = CP \times \rho \times n^3 \times D^5$	Propeller power required
5.4	$tflight = (Capacity \times 60) / Iavg$	Battery flight time estimation
5.5	$TWR = Ttotal / Wtotal$	Thrust-to-weight ratio

Eq.	Formula	Purpose
5.6	$d = c \times \Delta t$	GPS pseudorange from signal travel time
5.7	$\theta = \alpha \cdot \theta_{gyro} + (1-\alpha) \cdot \theta_{accel}$	Complementary filter attitude estimation
5.8	$\theta_{gyro} = \theta_{prev} + (\omega \times \Delta t)$	Gyroscope angle integration
5.9	$d = (v_{sound} \times t) / 2$	Ultrasonic distance measurement
5.10	$u(t) = K_p \cdot e(t) + K_i \int e(t)dt + K_d(de/dt)$	PID controller output
5.11	$\tau = (T1 - T2) \times L$	Roll/pitch torque
5.12	$R = (\eta \cdot E_{battery} \cdot (L/D)) / (W \cdot g)$	Fixed-wing UAV range equation
5.13	$P_r = P_t G_t G_r (\lambda / 4\pi d)^2$	Friis transmission equation (link budget)

5.14 Review Questions

Short Answer Questions

77. Define UAV and UAS. How do they differ?
78. List the six degrees of freedom of a UAV in flight.
79. Differentiate between fixed-wing, single-rotor, multi-rotor, and hybrid VTOL drones.
80. List two applications of UAVs each in defense, civil, and environmental monitoring domains.
81. What is the function of an Electronic Speed Controller (ESC) in a UAV?
82. Explain the significance of the Kv rating of a BLDC motor.
83. Why is GPS alone insufficient for stable UAV flight control?
84. List the three sensors typically combined in an IMU and state what each measures.
85. Explain the working principle of an ultrasonic distance sensor.
86. How does a quadcopter achieve yaw rotation using only fixed-pitch propellers?
87. What is the role of the EKF in ArduPilot's architecture?
88. List four key functions of Mission Planner as a ground control station.
89. Define the Internet of Drones (IoD) and list its four architectural layers.
90. What is a "drone-in-a-box" system, as used by FlytBase?

Numerical / Problem-Solving Questions

91. A BLDC motor has Kv = 1200 RPM/V and is powered by a 4S (14.8 V) LiPo battery. Calculate its no-load RPM.

92. A UAV uses a 4200 mAh battery and draws an average current of 12 A during flight. Estimate the flight time in minutes.
93. A UAV weighs 2.5 kg and its four motors together produce a maximum total thrust of 6.0 kg-force. Calculate the thrust-to-weight ratio.
94. An ultrasonic sensor measures a round-trip echo time of 8 ms. Calculate the distance to the ground, assuming the speed of sound is 343 m/s.
95. Using a complementary filter with $\alpha = 0.98$, $\theta_{gyro} = 15°$, and $\theta_{accel} = 12°$, calculate the fused attitude angle θ.
96. A telemetry transmitter radiates Pt = 100 mW with Gt = Gr = 2 (unity-scale gain), at a wavelength $\lambda = 0.125$ m (2.4 GHz) and a distance d = 500 m. Estimate the received power Pr using the Friis equation.
97. Two motors on one side of a quadcopter produce a combined thrust of 8 N, while the opposite pair produces 6 N. If the moment arm L = 0.25 m, calculate the resulting roll torque.

Essay / Long Answer Questions

98. With a neat labelled diagram, explain the major mechanical and electronic elements of a multi-rotor UAV.
99. Explain, with formulae, how differential motor speeds produce roll, pitch, and yaw motion in a quadcopter.
100. Discuss the role of GPS and IMU in UAV navigation, and explain why sensor fusion is necessary.
101. Describe the architecture of ArduPilot autopilot software and explain how it interacts with Mission Planner.
102. Explain the concept of the Internet of Drones (IoD) with its layered architecture and discuss the key challenges it faces.
103. Analyze the FlytBase case study and explain how it demonstrates the practical application of IoT/IoD concepts in autonomous drone operations.