Internet Of Things

Christian Rossi

Academic Year 2024-2025

Abstract

The course provides an overview of the four main components of IoT systems: sensors, communication technologies, management platforms, and data processing and storage platforms for sensor data. In the first part, the course covers the characteristics of the hardware components of sensor nodes (microcontrollers/microprocessors, memory, sensors, and communication devices). It then delves into communication technologies used in IoT systems, distinguishing between short-range solutions (ZigBee, 6LoWPAN) and long-range solutions (LoRaWAN, NB-IoT). Finally, the course focuses on application-level protocols for IoT systems (COAP, MQTT) and the analysis of IoT management platforms. The course includes hands-on development activities and is delivered through flipped classroom and/or blended learning formats.

Contents

CHAPTER 1

Introduction

1.1 Internet

Definition (*Internet*)**.** The Internet is a global network that connects various types of networks, enabling communication and data exchange.

Traditionally, the internet was primarily used for fixed, stationary clients accessing well-defined services. However, modern internet usage has shifted significantly with the rise of mobile clients. These mobile devices, often equipped with sensing and actuating capabilities, are no longer just consumers of information and services.

Technological advancements Several breakthroughs have paved the way for the rapid growth of the Internet of Things. The miniaturization of hardware, including CMOS technology, microelectromechanical systems, and advancements in materials and circuits, has enabled the development of compact yet powerful smart devices. At the same time, improvements in energy solutions, such as fuel cells and energy harvesting techniques, have enhanced the efficiency and autonomy of these devices. Increased mobility has further expanded the reach and functionality of Internet of Things applications.

In parallel, communication protocols have evolved to support low-power wireless technologies, ensuring efficient and reliable connectivity. The widespread adoption of cloud computing has also played a crucial role, providing scalable architectures and vast processing power. Additionally, the rise of artificial intelligence, particularly deep learning and generative AI, has unlocked new possibilities for intelligent data analysis, automation, and decision-making within Internet of Things ecosystems.

1.1.1 Internet of Things

Definition (*Internet of Things*)**.** The Internet of Things is a worldwide network of uniquely addressable interconnected objects, based on standard communication.

The Internet of Things is based on:

- *Smart objects*: devices embedded with sensors, actuators, and connectivity
- *Data*: continuous collection and processing of information

- *Pervasiveness*: seamless integration into everyday life
- *Seamless communication*: reliable and efficient interaction between devices, networks, and services

The Internet of Things primarily consists of connected low-cost endpoints, such as consumer devices and everyday smart objects, which focus on accessibility and widespread adoption.

1.1.2 Industrial Internet of Things

Definition (*Industrial Internet of Things*)**.** The Industrial Internet of Things refers to a network of interconnected sensors, instruments, and devices integrated with industrial computing applications, including manufacturing, energy management, and automation.

The Industrial Internet of Things consists of connected industrial assets that are typically medium to high-cost. These devices are more expensive but also more responsive, playing a critical role in industrial automation, manufacturing, and energy management.

Cybersecurity is a central concern in the Industrial Internet of Things, where even minor disruptions can have severe consequences. Unlike consumer Internet of Things, Industrial Internet of Things systems must operate with continuous availability, robustness, and resiliency, ensuring that industrial processes remain uninterrupted.

Industrial Internet of Things environments often coexist with a significant amount of legacy operational technologies such as SCADA systems, Programmable Logic Controllers, and Distributed Control Systems. These legacy systems, designed for reliability rather than cybersecurity, introduce additional challenges in securing industrial networks.

While usability and user experience are critical in consumer Internet of Things, they are not primary concerns in Industrial Internet of Things. Instead, the focus is on system integrity, fault tolerance, and maintaining operational continuity in complex industrial ecosystems.

1.1.3 Building blocks

The Internet of Things endpoints require strong security and reliability to ensure they operate safely and effectively within a network. These devices are not just about connectivity; they depend on a combination of smart objects, reliable connectivity, data collection, and advanced analytics to function properly.

The security of Internet of Things endpoints is critical, as these devices often handle sensitive data and are vulnerable to cyber threats. Ensuring reliability ensures that these devices can perform their tasks without interruption, providing accurate data and seamless communication within the system.

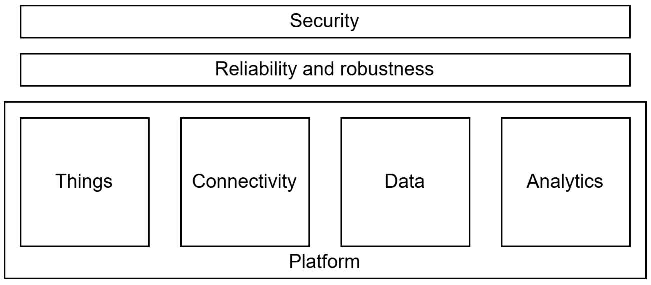

Figure 1.1: Internet of Things building blocks

1.2 Hardware

Definition (*Sensor node*)**.** A sensor node (or mote) is a device with several core capabilities:

- Sensing external phenomena, such as temperature, humidity, or pressure.
- Processing information collected by the sensors.
- Storing the gathered data.
- Communicating with other sensor nodes or external devices.

An actuator performs the following tasks: receiving input signals from control devices, processing and storing information, and acting on the industrial process, executing commands to modify conditions based on the input data.

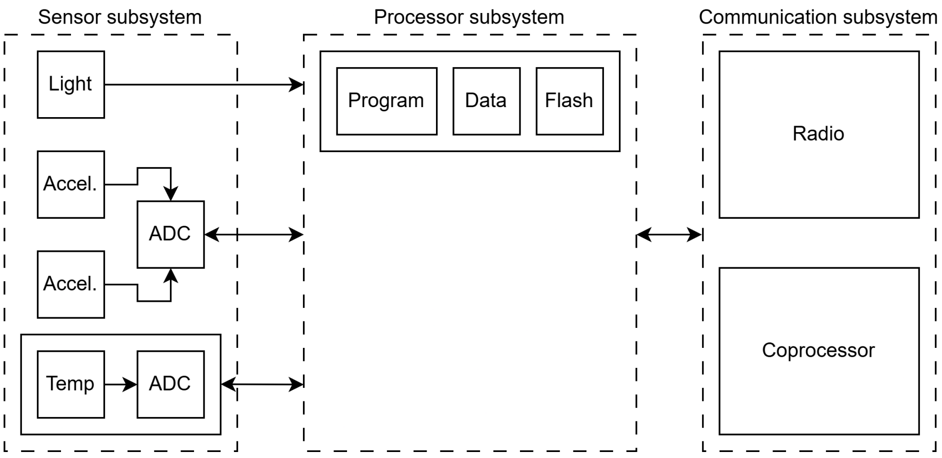

Figure 1.2: Sensor node architecture

1.2.1 Processor

The processor subsystem of a sensor node is often designed based on the SHARC architecture, though there are various alternatives for the individual components. These alternatives offer flexibility in terms of processing power, energy consumption, and other factors critical to specific use cases.

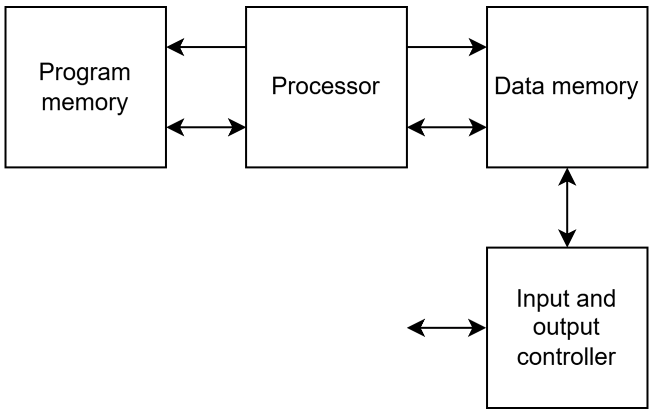

Figure 1.3: Processor architecture

A microcontroller is generally used as the processor in sensor nodes.

Definition (*Microcontroller*)**.** A microcontroller is a single integrated circuit designed for a specific application.

A microcontroller is usually compose by a Central Processing Unit and a clock generator (oscillator with quartz timing crystals). It is usually equipped with RAM, flash memory and an EEPROM. It is connected with a serial BUS, I/O interfaces and analogic and digital converters. While microcontrollers are flexible and low-cost, they can compromise speed in certain use cases.

Definition (*Digital Signal Processor*)**.** A Digital Signal Processor is a specialized microprocessor optimized for processing discrete signals using digital filters.

Digital Signal Processors excel at performing complex mathematical operations with extremely high efficiency. They can process hundreds of millions of samples per second, providing real-time performance. While they are well-suited for data-intensive operations, they are less flexible than microcontrollers.

Definition (*Application Specific Integrated Circuit*)**.** An Application Specific Integrated Circuit is a custom-designed integrated circuit tailored for a specific application.

ASICS offer high speed and can be tailored for specific tasks, but they come with high development costs and limited flexibility once designed.

Definition (*Field Programmable Gate Array*)**.** A Field Programmable Gate Array has a high-level architecture similar to ASICs but allows for some degree of reconfigurability after manufacturing.

Field Programmable Gate Arrays offer high-speed performance, supporting parallel programming, and moderate reconfigurability. However, they are more complex and costly than microcontrollers or Digital Signal Processors.

1.2.2 Sensor

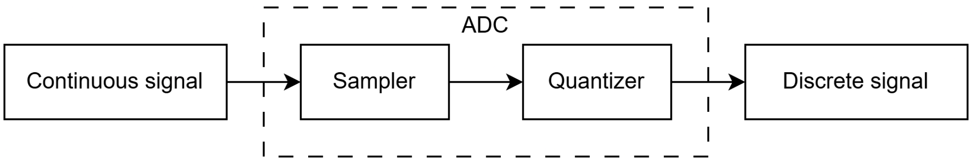

Figure 1.4: Analog to digital converter

Sampling The Nyquist ADC technique involves reading a time-continuous signal at specific points in time. The sampling rate, or bandwidth, is the inverse of the sampling interval:

$$f_s = \frac{1}{T}$$

The key idea is that if the sampling frequency is properly set, the original signal can be losslessly reconstructed from its samples.

Theorem 1.2.1 (Nyquist theorem)**.** *Given the signal bandwith B, we have that the sampling frequency must be chosen as:*

$$f_s = 2B$$

Quantization In quantization, the input voltage V_{in} is approximated by a digital codeword. An ideal quantizer maps input to output with the smallest variation in the input causing a change in the codeword.

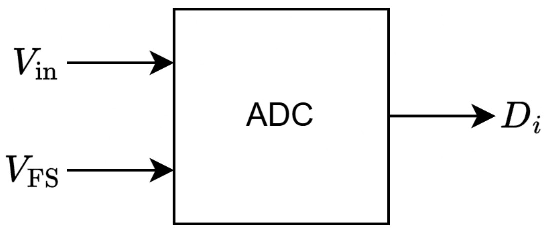

Figure 1.5: Quantization

The resolution refers to the smallest input variation that causes a change in the codeword:

$$\text{LBS} = \frac{V_{\text{FS}}}{2^n}$$

Here, V_{FS} is the full-scale voltage and n is the number of bits of resolution.

Quantization involves discretizing the continuous amplitude of the sampled signal. The quantization error occurs when the output voltage either overestimates or underestimates the input voltage. This error decreases as the resolution increases.

Hardware possibilities The IoT hardware landscape is vast, fragmented, and heterogeneous, with a wide variety of options in terms of CPU types, connectivity, storage, and sensing peripherals. Many end-devices and sensors are now capable of running full operating systems, enabling more complex applications.

There is a clear distinction between the application layer and the hardware control layer, allowing for greater flexibility in how systems are designed and deployed. However, there is also a battle between different types of operating systems in this space, including: commercial RTOS, open-source RTOS, and non-RTOS solutions.

1.2.3 Processor power

The power dissipation of the CPU is due to several factors, including:

$$P_{\text{p}} = P_{\text{dyn}} + P_{\text{sc}} + P_{\text{leak}}$$

Here, P_{dyn} is the power consumed by the work done (dynamic power), P_{sc} is the power lost due to short circuits, and P_{leak} is the power lost due to leakage. The dynamic power consumed during operation is given by:

$$P_{\text{dyn}} = CfV^2$$

Here, C is the capacitance, f is the frequency, and V is the voltage.

Local data processing is crucial in minimizing power consumption, especially in multi-hop networks, where power efficiency is key.

1.2.4 Sensor and actuator power

The absence of cables in wireless sensors and actuators means no wired power or connectivity. This presents unique challenges, especially in terms of energy efficiency, which becomes a must in these systems.

A sensor node typically operates with a limited power source, and its lifetime directly depends on the battery lifetime. The goal is to maximize the energy provided while minimizing the cost, volume, weight, and recharge requirements. However, the problem arises when recharging or battery replacement becomes impractical or too expensive.

There are two main types of batteries used:

- Primary batteries, which are not rechargeable.
- Secondary batteries, which are rechargeable, but only make sense when paired with some form of energy harvesting.

Guidelines To extend battery life, one of the key strategies is to switch off the radio as soon as possible, since radios consume significant power. The power consumption of short-range wireless communication devices remains roughly the same whether the radio is transmitting, receiving, or just idle and listening for potential signals.

Circuit power is primarily dominated by the core components, rather than large amplifiers. The radio must be listening to receive data, even if transmission is infrequent. Listening is often continuous, meaning the total energy consumption is dominated by the power used during idle listening.

Power cycle The power cycle of an IoT device consists of sleep and active states (wake-up/work). During the sleep state, power consumption is minimal, with only essential components running, resulting in some leakage. The average power consumption is then defined as:

$$P_{\text{avg}} = f_{\text{sleep}} P_{\text{sleep}} + f_{\text{wakeup}} P_{\text{wakeup}} + f_{\text{work}} P_{\text{work}}$$

Here, f_{sleep}, f_{wakeup}, and f_{work} are the fractions of time spent in sleep, wake-up, and work states, respectively.

The lifetime of the device is given by:

$$\text{lifetime} = \frac{\text{energy store}}{P_{\text{avg}} - P_{\text{gen}}}$$

Here, P_{gen} is the power generated.

Transmission consumption When data needs to be sent, the device first wakes up and then performs the actual transmission. The total energy consumption for this process is given by:

$$E_{tx} = P_{tx}(T_{wu} + T_{tx}) + P_0 T_{tx}$$

Here, P_{tx} is the power consumed by the transmitter, P_O is the output power of the transmitter,T_{tx} is the time taken to transmit a packet, and T_{wu} is the wake-up time.

Reception consumption When the device needs to receive data, it first wakes up and then performs the reception. The total energy consumption in this case is:

$$E_{tx} = P_{rx}(T_{uw} + T_{rx})$$

Here, P_{rx} is the power consumed by the receiver, T_{rx} is the time taken to receive a packet, and T_{wu} is the wake-up time.

Emitted power The emitted power is often a tunable parameter, and it is generally considered good practice to set it to the lowest value that still allows for reliable reception. The quality of the reception process is typically measured using metrics like:

- *Bit Error Rate*: the fraction of bits that are incorrectly received.
- *Packet Error Rate*: the fraction of packets that are not received correctly.

The relationship between BER and PER, for a packet of length l with independent errors, is given by:

$$\text{PER} = 1 - (1 - \text{BER})^l$$

Both BER and PER are influenced by the level of noise in the transmission and reception channels, which in turn is determined by the transmitted and received power. The Signal-to-Interference-plus-Noise Ratio is a key factor in determining this quality, and is calculated as:

$$\text{SINR} = 10 \log_{10} \left(\frac{P_{\text{recv}}}{N_0 + \sum_{i=1}^{k} I_i} \right)$$

Here, N_0 is the thermal noise (KTB), P_{recv} is the received power, and I_i are the interference contributions from other signals. Given the SINR and the specific modulation of the channel, BER can be computed.

Receiver sensitivity Each receiver is characterized by a sensitivity parameter, which is the minimum input signal power required for the receiver to demodulate the data correctly. Knowing this sensitivity, the required emitted power at the transmitter can be determined by inverting the propagation law of the communication channel.

Sensor power The power consumption due to sensing is highly dependent on the type of sensor used. A rough model for the power consumption of an Analog-to-Digital Converter can be expressed as:

$$P_s \sim f_s 2^n$$

Here, f_s is the sampling frequency, and n is the resolution of the ADC (in bits).

1.2.5 Design guidelines

Avoid full operation all the time. If there's no active task, switch to power-safe modes to preserve battery life.

Use power-aware operating systems that dim displays, enter sleep mode during idle times, and implement power-aware scheduling. Enable radios to forward packets at a lower power level while keeping the rest of the node in a sleep mode. Take advantage of performance-energy trade-offs within the communication subsystem by optimizing neighbor coordination and selecting appropriate modulation schemes.

1.3 Communication

In the Internet of Things, various devices and systems need to exchange information seamlessly. Industrial environments present unique challenges for communication networks. eyond environmental challenges, industrial networks must meet strict performance requirements. Timeliness is critical, as real-time data transmission ensures efficient and safe automation. Deterministic network behavior guarantees predictable response times, while reliability minimizes downtime and operational risks.

Designing a communication network for industrial systems requires careful planning. Network size and the number of connected devices determine the scale of infrastructure. The mobility of nodes, whether fixed or moving, influences the choice of communication protocols. Quality of Service constraints, including latency, throughput, and reliability, must align with operational demands. Integration with existing systems poses challenges, as legacy infrastructure must seamlessly connect with modern technologies. Budget constraints also play a role, influencing technology selection and implementation strategies.

Industrial communication networks handle various types of data traffic.

Definition (*Cyclic traffic*)**.** Cyclic traffic involves periodic transmissions, ensuring continuous process monitoring.

Definition (*Acyclic traffic*)**.** Acyclic traffic occurs in response to unpredictable events.

Definition (*Multimedia traffic*)**.** Multimedia traffic includes images, video streams, and other data-intensive transmissions.

Definition (*Backhand traffic*)**.** Backhand traffic aggregates data flows from multiple sources, consolidating information for centralized processing and analysis.

1.3.1 Key performance indicators

Definition (*Throughput*)**.** Throughput refers to the rate at which data can be transmitted over a network link, typically measured in bits per second or bytes per second.

The actual throughput depends on the type of link being used, as different technologies offer varying transmission capacities.

Definition (*Delivery time*)**.** Delivery time represents the total time required to transfer a service data unit from the source to the destination.

Measured in seconds, it is influenced by multiple factors, including transmission time, propagation time, and the execution time of network protocols.

Definition (*Minimum cycle time*)**.** Minimum Cycle Time defines the shortest duration needed to complete one full cycle within a control loop.

This metric is crucial in real-time systems, where consistent and predictable execution times are essential for maintaining stability and efficiency.

Definition (*Jitter*)**.** Jitter measures the precision and reliability of periodic operations, particularly in time-sensitive applications.

Variability in timing can lead to performance issues, making jitter a critical factor in industrial and communication networks that require consistent and predictable timing.

1.3.2 Technology

Defining a communication technology involves several key aspects, starting with the physical infrastructure.

This includes hosts, such as sensors, actuators, PLCs, and SCADA systems, which serve as traffic endpoints. Communication links, whether fiber optics, wireless, or wired connections, facilitate data transmission. Network nodes, including access points, switches, routers, and gateways, manage traffic flow and interconnect various components.

IoT connectivity rarely relies on a single-hop or single-technology approach. Instead, it integrates multiple heterogeneous technologies to ensure seamless communication across diverse environments. The classification of communication technologies often depends on their range.

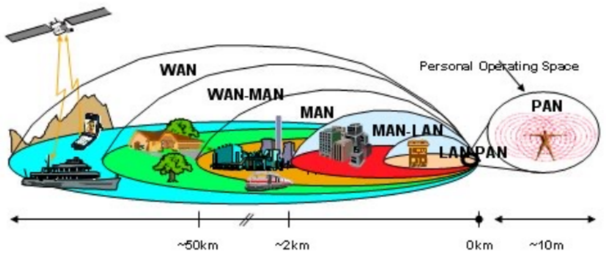

Figure 1.6: Connection range

Network topology also plays a significant role in communication technology classification. Ring topology provides a structured flow of data, while linear chains connect devices sequentially. Tree topology enables hierarchical communication, whereas star topology centralizes connections around a single node. Mesh networks offer robust, decentralized connectivity, enhancing reliability and redundancy in complex systems.

1.3.3 Protocols

Communication protocols define the set of rules governing the exchange of information between two entities. These rules specify aspects such as message format, connection setup and teardown procedures, and the semantics of the transmitted data. In IoT communication networks, protocols facilitate packet-based data exchange, similar to how the Internet operates.

Most communication protocols follow a layered architecture, where each layer provides specific services and functionalities to the layers above it. This structured approach enhances interoperability, modularity, and scalability in network communication. The primary layers include:

- *Application layer*: handles messages exchanged between applications, enabling services such as HTTP, MQTT, and CoAP.
- *Transport layer*: manages end-to-end communication through segmentation and reassembly, with protocols like TCP and UDP.
- *Network layer*: routes packets across different networks, using IP-based addressing and routing mechanisms.
- *Data link layer*: organizes data into frames, ensuring reliable point-to-point or point-to-multipoint transmission.

- *Physical layer*: deals with the actual transmission of raw bits over physical media such as wired or wireless connections.

CHAPTER 2

Application layer

2.1 Introduction

Definition (*Application*)**.** An application refers to a program running on a host that can communicate with another program over a network.

Applications enable machines, sensors, and control systems to exchange data, forming the foundation of smart manufacturing and Industry 4.0.

However, communication technologies in Industry 4.0 remain highly fragmented. Different protocols and languages coexist within factories, often creating compatibility challenges. Many field-level and factory-level technologies do not natively support Internet-based communication, making seamless data exchange difficult.

A practical solution to this fragmentation is the adoption of Service-Oriented Architectures, which facilitate standardized and scalable communication. SOA-based communication in Industry 4.0 generally follows two key models:

- *Client and server model*: a client requests data from a server, much like how web browsers access information. Common protocols include HTTP and CoAP.
- *Publish and subscribe model*: instead of direct requests, devices publish data to a central broker, which then distributes updates to subscribed clients. This approach is more efficient for real-time and event-driven communication, with protocol such as MQTT being widely used.

2.2 Hyper Text Transfer Protocol

Web pages are composed of multiple resources, including HTML documents, images, scripts, and other embedded elements. Each resource is uniquely identified by a Uniform Resource Locator (URL) or a Uniform Resource Identifier (URI), allowing clients to locate and access them over a network. When a client requests a resource using a URL, the request is resolved through domain name resolution (DNS), followed by the retrieval of the requested data from the corresponding web server.

On the web, resources are managed by servers, identified through URIs, and accessed synchronously by clients using a request/response paradigm. This model aligns with the Representational State Transfer (REST) architectural style, which enables scalable and stateless interactions between clients and servers.

2.2.1 Communication

HTTP follows a client-server architecture, where communication occurs as follows:

- The client sends an HTTP request for a specific resource, identified by its URI.
- The server processes the request and responds with the requested resource or an appropriate status message.

HTTP is a stateless protocol, meaning each request is independent, and the server does not retain memory of previous interactions. This simplifies communication but requires additional mechanisms for maintaining user state when necessary.

HTTP requests are human-readable (ASCII-encoded) and follow a structured format. The request includes essential components such as the method, the target resource (URI), and optional headers containing metadata.

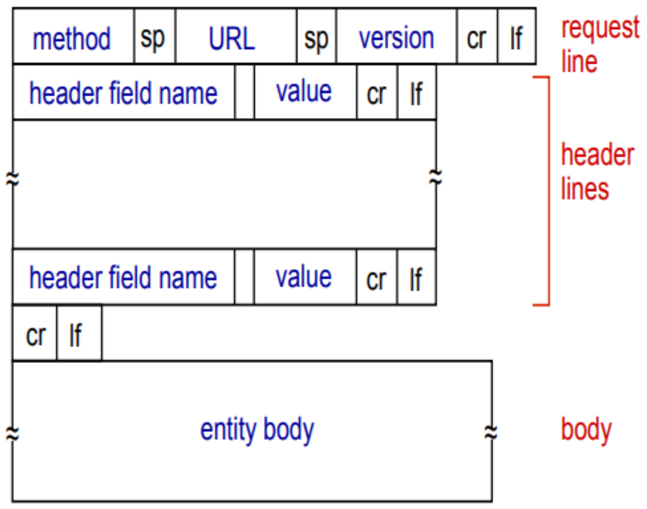

Figure 2.1: HTTP request

HTTP responses follow a similar format, including a status code that indicates the outcome of the request:

- *Informational* (1xx): request received and processing continues.
- *Success* (2xx): the request was successfully processed.
- *Redirection* (3xx): further action is needed to complete the request.
- *Client-side error* (4xx): the request contains incorrect syntax or cannot be fulfilled.
- *Server-side error* (5xx): the server encountered an issue while processing the request.

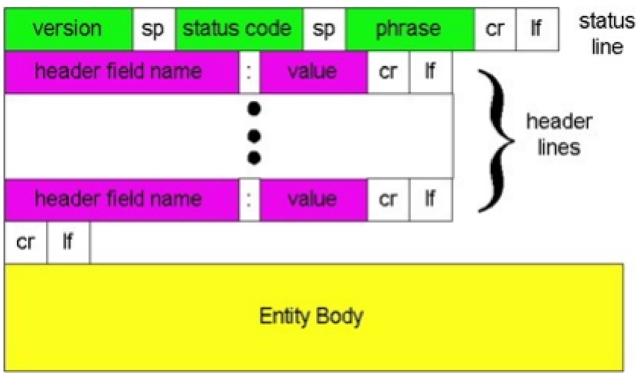

Figure 2.2: HTTP request

2.3 Constrained Application Protocol

The Constrained Application Protocol (CoAP) is designed to enable web-based services in constrained wireless networks, where traditional web solutions may not be feasible due to hardware and network limitations. These constraints include: low-power devices, limited memory and processing capabilities, and energy-efficient, low-bandwidth networks. To overcome these challenges, CoAP redesigns web-based communication to suit constrained environments while maintaining compatibility with standard web technologies.

Features CoAP is an embedded web transfer protocol optimized for lightweight, efficient communication. It introduces several features tailored for constrained devices:

- CoAP URI scheme `coap://`.
- Asynchronous transaction model which supports non-blocking communication.
- UDP binding with reliability and multicast support.
- RESTful methods.
- URI support.

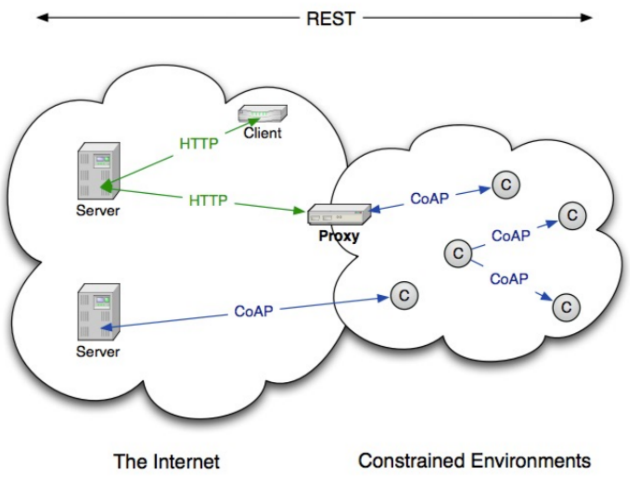

Figure 2.3: CoAP architecture

CoAP plays a critical role in IoT ecosystems, enabling lightweight, low-latency communication between resource-constrained devices while maintaining compatibility with traditional web technologies. Its UDP-based nature makes it well-suited for low-power wireless networks such as 6LoWPAN, LPWAN, and LoRaWAN, where energy efficiency and bandwidth conservation are key priorities.

Proxying and caching CoAP proxies act as intermediaries between clients and servers, forwarding requests and responses. They help with load balancing, security, and NAT traversal, improving network efficiency and scalability. Caching: CoAP supports caching responses to reduce repeated network requests. Clients and proxies can store responses and reuse them if the resource hasn't changed, saving bandwidth and improving speed.

2.3.1 Messages

CoAP primarily relies on UDP for lightweight communication, making it ideal for constrained networks. Message exchange occurs between endpoints with a compact 4-byte header, ensuring minimal overhead. Each message contains a 16-bit Message ID, enabling mechanisms for both reliable and unreliable transmission:

- *Confirmable messages*: require acknowledgment (ACK) or may trigger a Reset Message (RST) if lost. A stop-and-wait retransmission strategy with exponential backoff ensures reliability.
- *Non-confirmable messages*: sent without requiring acknowledgment, suitable for non-critical or frequent updates.
- *Duplicate detection*: both confirmable and non-confirmable messages use Message IDs to identify and discard duplicates.

CoAP follows a RESTful architecture, embedding request and response semantics directly into its messages. Each message can carry: methods, response code and options.

```
  0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.4: CoAP message

The CoAP 4-byte message header consists of several key fields:

- *Version* (`Ver`): CoAP version.
- *Type* (`T`): defines the message type. The message can be: confirmable (`CON`), non-confirmable (`NON`), acknowledgment (`ACK`), and reset (`RST`).
- *Token length* (`TKL`): indicates the number of bytes used for the token field.
- *Code*: represents either a request method (1 to 10) or a response code (40 to 255).
- *Message ID*: a 16-bit identifier for tracking requests and responses.

- *Token* (optional): used for request-response matching, especially in asynchronous communication.

2.3.2 Packet loss

CoAP employs a stop-and-wait retransmission mechanism to handle packet loss in unreliable networks. This approach ensures that lost messages are resent while maintaining low overhead.

When sending a confirmable message, the sender expects an acknowledgment or a reset message. If no response is received within a timeout period, the request is retransmitted.

```
Rand [ACK_TIMEOUT, ACK_TIMEOUT x ACK_RANDOM_FACTOR] ([2s, 3s])
```

If no response is received after the initial timeout, and the transmission counter is less than `MAX_RETRANSMIT(4)`: the transmission counter is increased, the timeout value is doubled (exponential backoff), and the message is retransmitted. This process continues until one of the following conditions is met:

- An `ACK` is received, confirming successful delivery.
- An `RST` message is received, indicating rejection.
- The transmission counter exceeds `MAX_RETRANSMIT` (4 retransmissions).
- The total attempt duration surpasses `MAX_TRANSMIT_WAIT` (93 seconds), at which point the transmission is aborted.

2.3.3 Observation

In the traditional REST paradigm, data is fetched through explicit pull requests, where the client continuously queries the server for updated information. In Wireless Sensor Networks, however, data is often periodic or triggered by specific events.

CoAP introduces the concept of Observation to address this issue. Observation allows clients to subscribe to resources and receive updates automatically whenever the resource's state changes, without having to repeatedly poll the server. This is more efficient, as it reduces unnecessary requests and bandwidth consumption.

CoAP clients use the `observe` option in their requests to subscribe to a resource. When the resource state changes, the server sends an update to the client. The client is notified asynchronously, receiving updates as they occur without needing to send explicit requests.

2.3.4 Block transfer

In networks using IPv6, large payloads are often fragmented at the lower layers, which can be inefficient. This can create overhead when transferring large data sets. CoAP uses Block Transfer at the application layer to handle large messages by splitting them into smaller blocks, avoiding fragmentation at lower layers. Block Transfer adds an option to CoAP messages, including:

- `nr`: incremental block number within the original data.
- `m`: a flag indicating if more blocks are expected.
- `sz`: block size, which determines the chunk size for each transfer.

2.3.5 Discovery

CoAP supports resource discovery, allowing clients to discover and interact with available resources on CoAP servers. The goal is to enable clients to discover the links hosted by a CoAP (or HTTP) server, including details like the URL, resource type, content type, and supported operations. The server returns resource details in a link-header format, which includes:

- *URL*: the resource location.
- *Relation*: describes the relationship between resources.
- *Type*: specifies the resource type.
- *Content-type*: indicates the format of the resource.
- *Interface*: describes the access protocol.

2.4 Message Queuing Telemetry Transport

MQTT is a Client-Server, publish/subscribe messaging protocol designed for lightweight and efficient communication, especially in constrained environments like IoT. It is ideal for scenarios requiring low bandwidth, simple implementation, and support for Quality of Service. The main features are:

- *Lightweight and bandwidth-efficient*: MQTT is designed to be simple to implement, particularly at the sensor side, with minimal overhead.
- *QoS support*: MQTT supports different levels of QoS to ensure reliable message delivery, providing flexibility in network conditions.
- *Data-agnostic*: it can handle a variety of data types without being tied to any specific format.
- *Session awareness*: MQTT maintains session state to ensure continuous communication even if the client temporarily disconnects.

In MQTT, clients do not communicate directly with each other. Instead, they publish messages to topics, and other clients subscribe to those topics. A single client can publish a message, and all clients subscribed to that topic will receive the message. Unlike CoAP's pull-based request/response model, MQTT uses a push model where updates are automatically sent to clients when new information is available.

Topics Topics are used to categorize and direct messages. Clients can publish or subscribe to these topics. Topics are hierarchical, with levels separated by a slash.

Keepalive Keepalive ensures that the connection between the client and the broker remains active. If the client does not send any other messages during the keepalive period, it sends a `PINGREQ` message to the broker to check the connection. The broker responds with a `PINGRESP`. Both `PINGREQ` and `PINGRESP` messages have a null payload. It is the client's responsibility to maintain the keepalive, ensuring that the session stays alive and messages can be delivered even if no new data is sent.

2.4.1 Connection

In MQTT, each client establishes a single connection to the MQTT broker for communication. This connection supports push capabilities, making it efficient for IoT applications where devices need to receive updates in real-time.

MQTT can also work even through firewalls or NAT devices, thanks to its use of TCP and well-defined protocols, allowing messages to pass with minimal restrictions. When a client connects to the broker, it sends a CONNECT message with several fields:

- `clientId`: a unique identifier for the client.
- `cleanSession`: if true, the broker will not retain any session state when the client disconnects. If false, it retains the session state for later reconnect.
- `username` (optional): a username for authentication.
- `password` (optional): a password for authentication
- `lastWillTopic` (optional): a topic where the broker will send a Last Will message if the client unexpectedly disconnects.
- `lastWillQoS` (optional): the QoS level for the last will message.
- `lastWillMessage` (optional): the actual last will message.
- `keepAlive`: the time (in seconds) the client allows the connection to remain idle before the broker expects a message to maintain the session.

When the broker responds to the CONNECT message, it sends a CONNACK message with the following fields:

- `sessionPresent`: indicates whether the client's session was retained.
- `returnCode`: the status code that tells the client the result of the connection attempt:
  - 0: successful connection.
  - 1: unacceptable version.
  - 2: the client ID is invalid or already in use.
  - 3: the broker is not available.
  - 4: authentication failed.
  - 5: the client is not allowed to connect.

2.4.2 Publisher

In MQTT, the publish message is used by the publisher to send data to the broker. The publish message includes the following fields:

- `packetId`: a unique identifier for the message.
- `topicName`: the topic to which the message is published.
- `QoS`: the Quality of Service level indicating the delivery guarantee.

- `retainFlag`: indicates whether the message should be retained by the broker.
- `Payload`: the actual data being sent in the message.
- `dupFlag`: indicates whether the message is a duplicate.

The Quality of Service of a pusblisher can be:

- *0* (at most once): the message is delivered at most once, with no guarantee of delivery. It is a best-effort transfer based on the reliability of the underlying transport protocol.
- *1* (at least once): the message is guaranteed to be delivered, but it may be delivered multiple times. The client stores the message and retransmits it until the broker acknowledges receipt. Once the broker receives the publish message, it sends a puback message back to the client to acknowledge receipt. This message includes the packetId of the message being acknowledged
- *2* (exactly once): ensures that the message is delivered exactly once, and involves a 4-step handshake:
    1. *Publish reception* (broker): the MQTT broker processes the packet and sends a PUBREC message back. The packetId is stored locally to avoid duplicate processing.
    2. *Pubrec reception* (client): upon receiving the PUBREC message, the client discards the original packet and sends a PUBREL message to the broker.
    3. *Pubrel reception* (broker): the broker clears any current state and sends a PUBCOMP message to confirm the delivery of the message.
    4. *Pubcomp reception* (broker): the client receives the PUBCOMP message.

2.4.3 Subscriber

In MQTT, the subscribe message is used by the client to subscribe to one or more topics. The fields in the subscribe message include:

- `packetId`: a unique identifier for the subscription request.
- `QoS1`: the Quality of Service level for the first topic.
- `Topic1`: the first topic the client wants to subscribe to.
- `QoS2`: the Quality of Service level for the second topic.
- `Topic2`: the second topic the client wants to subscribe to.

Once the broker receives the subscribe message, it responds with a suback message, which contains the following fields:

- `packetId`: the unique identifier for the subscription.
- `returnCode`: the result of the subscription request, which is returned for each topic in the subscription.

Unsubscribe To unsubscribe from topics, the client sends an unsubscribe message with the following fields:

- `packetId`: a unique identifier for the unsubscribe request.
- `Topic1`: the first topic to unsubscribe from.
- `Topic2`: the second topic to unsubscribe from.

When the broker processes the unsubscribe request, it sends an unsuback message with the following fields:

- `packetId`: the unique identifier for the unsubscribe request.
- `returnCode`: a return code for each topic in the unsubscribe request, indicating whether the unsubscribe action was successful.

2.4.4 Session

In MQTT, the session management behavior varies depending on whether the client uses a persistent session or a non-persistent session.

Non-persistent session In non-persistent sessions, when a client disconnects, all client-related information, such as subscriptions and QoS pending messages, is cleared from the broker. This means that when the client reconnects, it needs to re-subscribe to topics and will not receive messages sent during its disconnection.

Persistent session In a persistent session, both the client and the broker maintain the session state even if the client disconnects. This ensures that the client can resume its communication without losing its subscription information or missing messages. The session state includes:

- *Broker's responsibilities*: maintain the existence of the session, even if there are no active subscriptions, store all subscriptions associated with the client, retain messages in QoS 1 or QoS 2 that were not acknowledged by the client, store new QoS 1 or QoS 2 messages that were published while the client was offline, so the client can receive them upon reconnection, and keep track of all QoS 2 messages that were sent but not yet acknowledged by the client.
- *Client's responsibilities*: store any QoS 1 or QoS 2 messages that were sent by the broker but not yet acknowledged by the client and retain QoS 2 messages that were sent but not yet acknowledged by the broker.

With a persistent session, even if the client is offline, the broker will hold the state and deliver any relevant messages when the client reconnects.

2.4.5 Messages

Retained messages In MQTT, publishing and subscribing are asynchronous operations. This means that a client subscribing to a topic might not receive any message until another client publishes something to that topic.

Retained messages are publish messages where the retained flag is set to one. These messages are stored by the broker, and whenever a new client subscribes to the topic (or a matching topic pattern), the broker immediately sends the last retained message on that topic to the subscribing client. This ensures that subscribers receive the latest available message, even if no new message has been published.

Last will messages The Last Will and Testament (LWT) feature in MQTT allows a client to notify other clients about an unexpected disconnect. Here's how it works:

- *Client setup*: when a client connects to the broker, it can specify a last will message, which is a message that the broker will send to other clients if the client unexpectedly disconnects.
- *Broker's responsibility*: the broker stores the LWT message and only sends it if it detects that the client has disconnected unexpectedly.
- *Message delivery*: when the client experiences a hard disconnection, the broker sends the stored LWT message to all subscribed clients on the specified topic, alerting them of the disconnect.
- *Graceful disconnect*: if the client disconnects gracefully (by sending a disconnect message), the broker will discard the LWT message and not deliver it.

2.4.6 MQTT for Sensor Networks

MQTT-SN (MQTT for Sensor Networks) is a variation of the standard MQTT protocol designed to better suit the constraints of sensor networks and environments with limited resources. Here are the main differences compared to the standard MQTT protocol:

- *Extended architecture*: MQTT-SN introduces the concept of Gateways and Forwarders. Gateways connect the sensor network to the traditional MQTT broker, allowing for communication between sensors and other MQTT clients. Forwarders are used to help route messages across different networks.
- *New gateway discovery procedures*: MQTT-SN includes a new process for discovering gateways. This allows devices in a sensor network to automatically find and connect to the appropriate gateway for communication with the MQTT broker. The protocol defines messages specifically for this discovery process.
- *Compressed messages*: some messages in MQTT-SN are more compressed than in MQTT, making them smaller and more suitable for low-bandwidth and low-power networks. This helps to optimize message transfer, particularly when dealing with resource-constrained devices.
- *Extended keepalive procedures*: MQTT-SN supports extended keepalive procedures to handle clients that may be in sleep mode. These procedures allow clients to remain connected and avoid disconnection during periods of inactivity, thus improving the reliability of communication in energy-constrained environments.