Technological Educational Institute of Western Greece

Computer & Informatics Engineering Department

Design & Development of a message passing communication framework for heterogeneous Wireless Sensor Networks

Author: Antonopoulos Konstantinos

Supervisors:

Assistant Professor Voros Nikolaos

Department of Computer & Informatics Engineering

Technological Educational Institute of Western Greece

Research & Teaching Assistant Antonopoulos Christos

Department of Computer & Informatics Engineering

Technological Educational Institute of Western Greece

Master Thesis in Computer Science Engineering

February 2016

Abstract

Cyber Physical Systems (CPS) represent a relatively new research domain aiming to unite the physical and the ICT worlds. Therefore, it is anticipated to have a huge impact in a wide range of real life application scenarios. Incorporating different engineering domains such as Wireless Sensor Networks and Embedded systems, CPSs are characterized from high degree of heterogeneity regarding various aspects, such as communication, hardware and software solutions. Additionally, in order to be well accepted from end users an end-to-end, it is of paramount importance to exhibit high degree of configurability and flexibility so as to applicable in a diverse application scenarios. Aiming to address such objectives this work proposes a holistic end-to-end CPS communication infrastructure based on message passing communication technologies. Presented scheme offers homogeneous support to a wide range of WSN communication technologies while the system wide architecture is able to adjust to any application or platform peculiarities. Additionally, the whole architecture is implemented based on commercial off-the-shelf equipment thus demonstrating the infrastructure's degree of feasibility. Finally, end to end performance evaluation is performed highlighting, on one hand, applicability of the system to a wide range of real applications and, on the other hand, resource conservative behavior advocating integration to nowadays embedded systems.

Keywords: Cyber Physical System, Internet of Things, Wireless Sensor Networks, Message Passing Protocols, Communications, Infrastructure Design implementation, Performance Evaluation.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Cyber physical Systems

1.1.1 What is Cyber Physical Systems (CPS)

A Cyber Physical System [1] has a tight integration of cyber and physical objects. The term cyber objects refers to any computing hardware/software resources that can achieve computation, communication, and control functions in a discrete, logical, or switched environment. Also, physical objects refers to any natural or human-made systems that are governed by the laws of physics and operate in continuous time. It is believed that CPSs will transform how we interact with the physical world, just like the Internet transformed how we interact with one another. A CPS could be a system at multiple scales, from big smart bridges with fluctuation detection and responding functions, to autonomous cars, to tiny implanted medical devices. As a matter of fact, the ultimate purpose of using cyber infrastructure (including sensing, computing, and communication hardware/software) is to intelligently monitor (from physical to cyber) and control (from cyber to physical) our physical world.

1.1.2 Cyber Physical Systems Layout

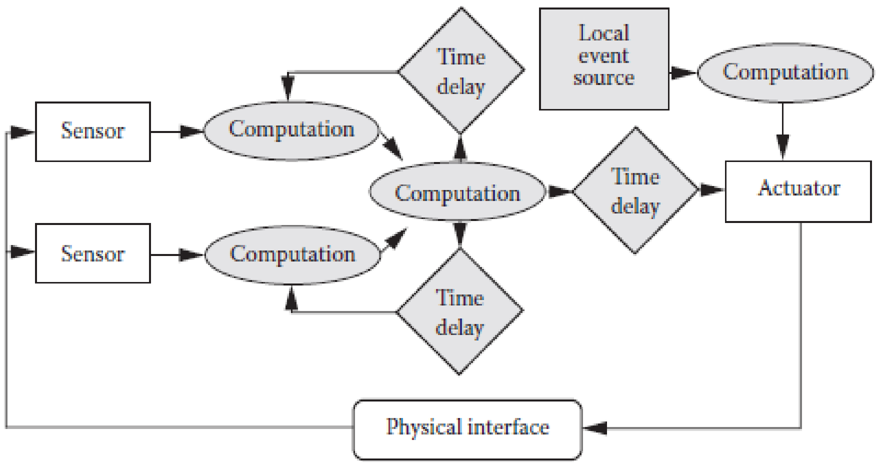


Figure 1. Simplified layout of a CPS

A CPS is the integration of computer processing with physical input/output. Embedded computers that are networked together utilize a series of feedback loops to oversee and control a variety of physical processes. A typical CPS has a basic layout consisting of interconnected sensors and actuators as shown in Figure 1. An actuator is simply a type of motor that moves or controls another mechanism. The sensors provide data taken from a physical object, which is, in turn, utilized by the actuator to perform a function. After data are collected by the sensors, a number of algorithms are performed and looped over again until a proper command is calculated and sent to the actuator. It is important to note that time delays must be accounted for when running computations.

1.1.3 CPS Challenges

Reliability and Uncertainty

Consumers expect CPSs to be reliable and consistent. Indeed, in many applications, such as medical systems, it is crucial that CPSs perform requested tasks on time and predictably. Unfortunately, current technologies leave much to be explored in this aspect. The behavior of electronic components is not perfectly consistent or predictable.

A disconnection often lies between program execution and physical requirements. A program has, essentially, 100% reliability in the sense that it will go through the exact same set of commands in exactly the same order every time it is run. However, physical systems rely not only on function but also on timing, and computer programs can be imperfect in this regard. This mismatch causes much uncertainty and unreliability and becomes a problem for CPSs.

Levels of Abstraction

Embedded systems, which CPSs fundamentally rely on, have different levels of abstraction. This is helpful in that it allows someone to work on one part of the system without having to understand or alter the rest of the system. For example, a programmer can change his or her code without having to worry about the actual electronic components that will execute the program's commands because the codes have been designed with certain abstraction; that is, they are suitable to any machines as long as the machine understands the code abstraction. However, the way in which this separation is currently implemented causes several orders of magnitude of timing precision to be lost. CPSs can be extremely time sensitive, and precise timing is very important. It is, however, quite encouraging that high precision is available on the most basic level digital circuits. This opens the possibility for more precise CPSs in the future.

Cyber-Physical Mismatch

Today, many frequently used programming languages do not have commands dealing with timing and synchronization. In many electronic products today, a delicate balance is reached between hardware and software, and even a slight change in either one can throw off this balance, thus causing errors. For example, airlines stock up on parts when installing a new hardware system because if part of the system is replaced with a non-identical part, the software will have to be retested and possibly changed. This is not only economically cumbersome, but it also keeps these consumers from having up-to-date technology. Thus, they are stuck using the old model until they use up all their stockpiled parts and can afford to catch up with the technology. So dividing a system into layers of abstraction, if not managed properly, may actually impede progress toward successful CPSs.

In a CPS today, the interaction and coordination between the physical elements and the cyber elements of a system are key aspects. In the physical world, one of the most dominant characteristics is its dynamics or the state of the system constantly changes over time. Alternatively, in the cyber world, these dynamics are more appropriately defined as a series of sequences that do not have temporal semantics. One of the greatest problems that engineers and researchers are faced with today is the point at which these two corresponding subsystems intersect one another. There are two basic approaches to analyzing this problem: cyberizing the physical (CtP), which is where cyber interfaces and properties are imposed on a physical system; and physicalizing the cyber (PtC), which is when software and cyber components are represented dynamically in real time.

Super dense timing

One of the greatest challenges that have been brought to the attention of many researchers is the notion of a uniform concept of time across all parts of a system operating simultaneously when correlating the cyber object to the physical object. There have been many problems in the dynamic systems with control processes that relate to synchronization, flocking, and formation control. Within the network of cyber components (such as sensors), a uniform concept of time cannot be realized.

Systems that utilize synchronization and time-triggered networks can approximate a time model, but imperfections must be included to accurately model the dynamics. The dynamics and natural uncertainties of the physical world are difficult to accurately model. The concept of time just does not mix well with the cyber world. Time is continuous in the real world but must become discrete in the cyber world.

1.2 Message passing protocols

1.2.1 What is messaging

Messaging is a technology that enables high-speed, asynchronous, highly reliable program-to-program communication. Programs communicate by sending packets of data called messages to each other. Channels, also known as queues, are logical pathways that connect the programs and convey messages. A channel behaves like a collection or array of messages, but one that is magically shared across multiple computers and can be used concurrently by multiple applications. A sender (or producer) is a program that sends a message by writing the message to a channel. A receiver (or consumer) is a program that receives a message by reading (and deleting) it from a channel.

The message itself is simply some sort of data structure—such as a string, a byte array, a record, or an object. It can be interpreted simply as data, as the description of a command to be invoked on the receiver, or as the description of an event that occurred in the sender. A message actually contains two parts, a header and a body. The header contains meta-information about the message—who sent it, where it's going, etc.; this information is used by the messaging system and is mostly (but not always) ignored by the applications using the messages. The body contains the data being transmitted and is ignored by the messaging system. In conversation, when an application developer who is using messaging talks about a message, he's usually referring to the data in the body of the message.

Asynchronous messaging architectures are powerful, but require us to rethink our development approach. As compared to the other three integration approaches, relatively few developers have had exposure to messaging and message systems. As a result, application developers in general are not as familiar with the idioms and peculiarities of this communications platform.

1.2.2 Messaging Systems

Messaging capabilities are typically provided by a separate software system called a messaging system or message-oriented middleware (MOM). A messaging system manages messaging the way a database system manages data persistence. Just as an administrator must populate the database with the schema for an application's data, an administrator must configure the messaging system with the channels that define the paths of communication between the applications. The messaging system then coordinates and manages the sending and receiving of messages. The primary purpose of a database is to make sure each data record is safely persisted, and likewise the main task of a messaging system is to move messages from the sender's computer to the receiver's computer in a reliable fashion.

The reason a messaging system is needed to move messages from one computer to another is that computers and the networks that connect them are inherently unreliable. Just because one application is ready to send a communication does not mean that the other application is ready to receive it. Even if both applications are ready, the network may not be working, or may fail to transmit the data properly. A messaging system overcomes these limitations by repeatedly trying to transmit the message until it succeeds. Under ideal circumstances, the message is transmitted successfully on the first try, but circumstances are often not ideal.

In essence, a message is transmitted in five steps:

1. **Create**, sender creates the message and populates it with data.
2. **Send**, sender adds the message to a channel.
3. **Deliver**, the messaging system moves the message from the sender's computer to the receiver's computer, making it available to the receiver.
4. **Receive**, receiver reads the message from the channel.
5. **Process**, receiver extracts the data from the message.

1.2.3 Why use Messaging

Benefits of using messaging:

- **Remote Communication**, Messaging enables separate applications to communicate and transfer data. Two objects that reside in the same process can simply share the same data in memory. Sending data to another computer is a lot more complicated and requires data to be copied from one computer to another.
- **Platform / Language Integration**, When connecting multiple computer systems via remote communication, these systems likely use different languages, technologies and platforms, perhaps because they were developed over time by independent teams. Integrating such divergent applications can require a demilitarized zone (DMZ) of middleware to negotiate between the applications, often using the lowest common denominator—such as flat data files with obscure formats. In these circumstances, a messaging system can be a universal translator between the applications that works with each one’s language and platform on its own terms, yet allows them to all communicate through a common messaging paradigm.
- **Asynchronous Communication**, Messaging enables a send and forget approach to communication. The sender does not have to wait for the receiver to receive and process the message; it does not even have to wait for the messaging system to deliver the message. The sender only needs to wait for the message to be sent. Once the message is stored, the sender is then free to perform other work while the message is transmitted in the background. The receiver may want to send an acknowledgement or result back to the sender, which requires another message, whose delivery will need to be detected by a callback mechanism on the sender.
- **Variable Timing**, with synchronous communication, the caller must wait for the receiver to finish processing the call before the caller can receive the result and continue. In this way, the caller can only make calls as fast as the receiver can perform them. On the other hand, asynchronous communication allows the sender to batch requests to the receiver at its own pace, and for the receiver to consume the requests at its own different pace. This allows both applications to run at maximum throughput and not waste time waiting on each other.
- **Throttling**, A problem with remote procedure calls is that too many of them on a single receiver at the same time can overload the receiver. This can cause

performance degradation and even cause the receiver to crash. Asynchronous communication enables the receiver to control the rate at which it consumes requests, so as not to become overloaded by too many simultaneous requests. The adverse effect on callers caused by this throttling is minimized because the communication is asynchronous, so the callers are not blocked waiting on the receiver.

- **Reliable Communication**, Messaging provides reliable delivery. The messaging system uses a store and forward approach to transmitting messages. The data is packaged as messages, which are atomic, independent units. When the sender sends a message, the messaging system stores the message. It then delivers the message by forwarding it to the receiver's computer. Storing the message on the sender's computer and the receiver's computer is assumed to be reliable
- **Disconnected Operation**, Some applications are specifically designed to run disconnected from the network, yet to synchronize with servers when a network connection is available. Messaging is ideal for enabling these applications to synchronize—data to be synchronized can be queued as it is created, waiting until the application reconnects to the network.
- **Mediation,** The messaging system acts as a mediator between all of the programs that can send and receive messages. An application can use it as a directory of other applications or services available to integrate with. If an application becomes disconnected from the others, it need only reconnect to the messaging system, not to all of the other messaging applications. The messaging system can be used to provide a high number of distributed connections to a shared resource, such as a database. The messaging system can employ redundant resources to provide high-availability, balance load, reroute.
- **Thread Management**, Asynchronous communication means that one application does not have to block while waiting for another application to perform a task, unless it wants to. Rather than blocking to wait for a reply, the caller can use a callback that will alert the caller when the reply arrives.

1.2.4 Challenges of Asynchronous Messaging

Asynchronous messaging is not the panacea of integration. It resolves many of the challenges of integrating disparate systems in an elegant way but it also introduces new challenges. Some of these challenges are inherent in the asynchronous model while other challenges vary with the specific implementation of a messaging system.

- **Complex programming model**, asynchronous messaging requires developers to work with an event-driven programming model. Application logic can no longer be coded in a single method that invokes other methods, but the logic is split up into a number of event handlers that respond to incoming messages.
- **Sequence issues**, message channels guarantee message delivery, but they do not guarantee when the message will be delivered. This can cause messages that are sent in sequence to get out of sequence.
- **Synchronous scenarios**, not all applications can operate in a send and forget mode. Therefore, many messaging systems need to bridge the gap between synchronous and asynchronous solutions.
- **Performance**, messaging systems add some overhead to communication. It takes effort to make data into a message and send it, and to receive a message and process it.

2. Background Information

2.1 Wireless communications protocols

2.1.1 Bluetooth

Bluetooth [2] is a short-range (10–100m) wireless link technology aimed at replacing cables that connect phones, laptops, PDAs, and other portable devices, developed by Bluetooth Special Interest Group (SIG).

The Bluetooth standard operates at 2.4 GHz in the ISM band with GFSK modulation. The FHSS technique is used to reduce the effect of radio frequency interferences on transmission quality. The Bluetooth devices sharing the same channel, form a network called piconet, with a single unit acting as a master, the other units acting as slaves. Up to eight devices constitute a piconet, with a master device coordinating access by a polling scheme.

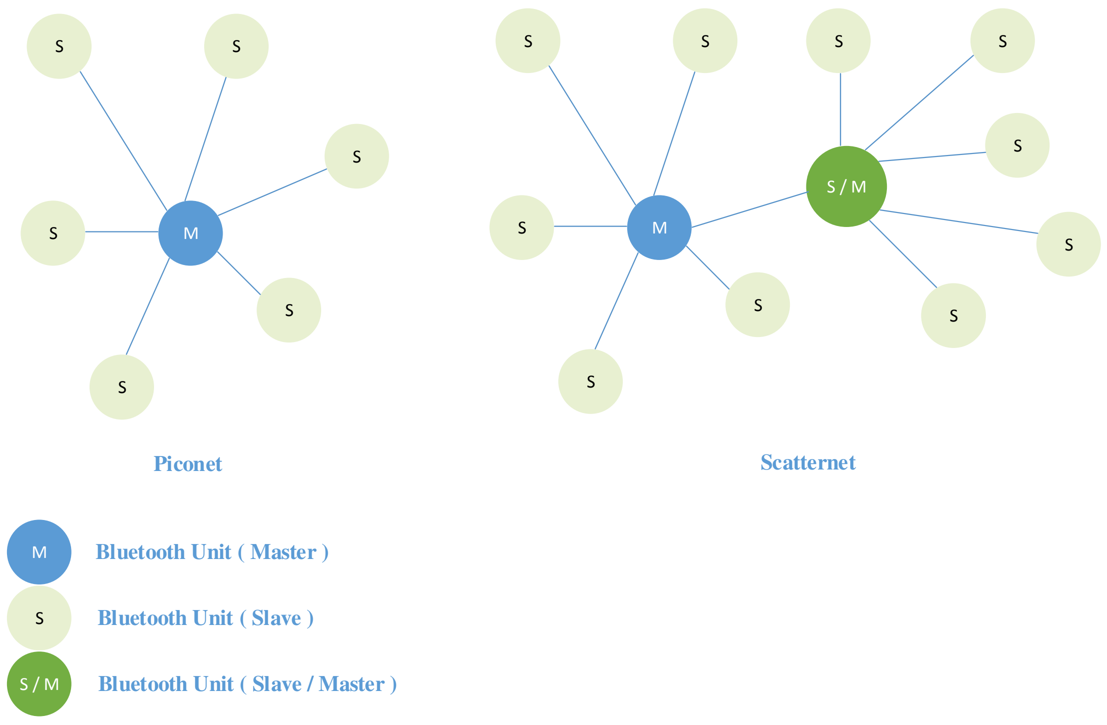


Figure 2. Bluetooth Piconet & Scatternet

In many cases multiple piconets can cover the same area, a unit can participate in two or more overlaying piconets by applying time multiplexing. To participate on the proper channel, it should use the associated master device address and proper clock offset to obtain the correct phase. A Bluetooth unit can act as a slave in several piconets, but only as a master in a single piconet: since two piconets with the same master are synchronized and use the same hopping sequence, they are one and the same piconet. A group of piconets in which connections consists between different piconets is called a scatternet.

The channel is represented by a pseudo-random hopping sequence in the 79 RF channels of 1-MHz width. The raw data rate is 1 Mbit/s. The hopping sequence is unique for the piconet and is determined by the Bluetooth master. The channel is divided into time slots, where each slot corresponds to an RF hop frequency. Consecutive hops correspond to different RF frequencies. The nominal hop rate is 1600 hops/s. A time division multiplexing (TDD) technique divides the channel into 625 μsecs slots and, with a 1-Mbit/s symbol rate, a slot can carry up to 625 bits.

Protocol Stack

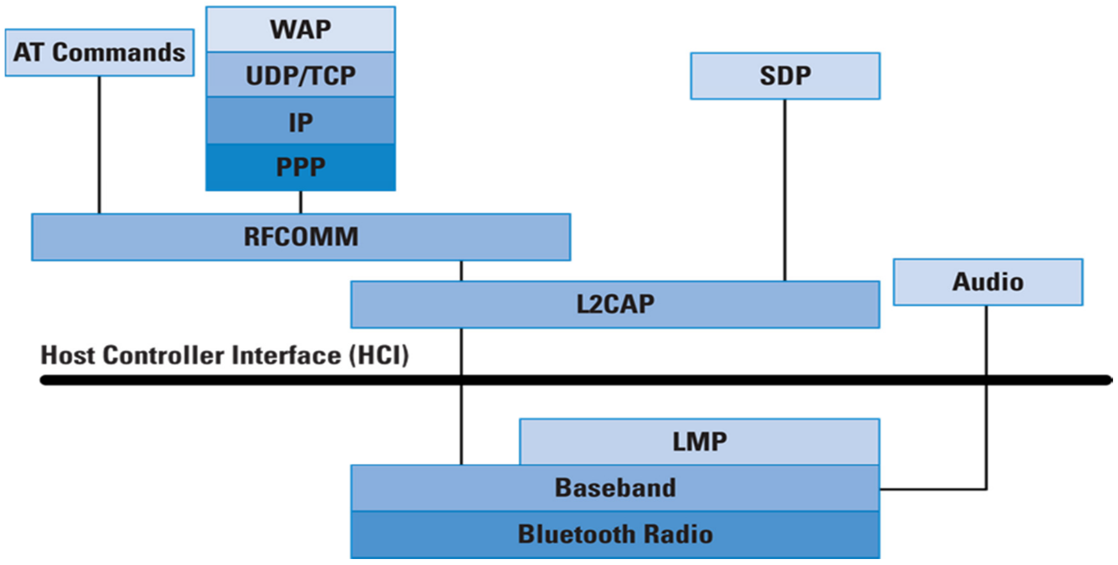


Figure 3. Bluetooth Stack

SDP

The service discovery protocol (SDP) provides a means for applications to discover which services are available and to determine the characteristics of those available services.

RFCOMM

The Radio Frequency Communications (RFCOMM) protocol is a reliable streams-based protocol. It provides roughly the same service and reliability guarantees as TCP. The Bluetooth specification states that it was designed to emulate RS-232 serial ports

L2CAP

The Logical Link Control and Adaption Protocol (L2CAP) is a packet-based protocol that can be configured with varying levels of reliability. The default maximum packet size is 672 bytes, but this can be negotiated up to 65,535 bytes after a connection is established. L2CAP can be compared with UDP, which is a best-effort packet-based protocol, but there are enough differences that the use cases for L2CAP are much broader than the use cases for UDP. Both are packet-based protocols, but L2CAP enforces delivery order.

HCI

The HCI provides a uniform interface method of accessing the Bluetooth hardware capabilities. The HCI Link commands provide the Host with the ability to control the link layer connections to other Bluetooth devices.

LMP

Used for control of the radio link between two devices, handling matters such as link establishment, querying device abilities and power control. Implemented on the controller.

2.1.2 802.15.4

802.15.4 [3] is a standard for wireless communication issued by the IEEE (Institute for Electrical and Electronics Engineers). The IEEE is a technical professional association that has written numerous standards to promote growth and interoperability of existing and emerging technologies. IEEE has published the standards that define communication in areas such as the Internet, PC peripherals, industrial communication and wireless technology.

802.15.4 was developed aiming towards low data rate, simple connectivity, connectionless data transfer, and battery dependant applications in mind. The

802.15.4 standard specifies that communication can occur in the 868-868.8 MHz, the 902-928 MHz or the 2.400-2.4835 GHz Industrial Scientific and Medical (ISM) bands. While any of these bands can technically be used by 802.15.4 devices, the 2.4 GHz band is more popular as it is open in most of the countries worldwide. The 868 MHz band is specified primarily for European use, whereas the 902-928 MHz band can only be used in the United States, Canada and a few other countries and territories that accept the FCC regulations.

The 802.15.4 standard specifies that communication should occur in 5 MHz channels ranging from 2.405 to 2.480 GHz. In the 2.4 GHz band, a maximum over-the-air data rate of 250 kbps is specified, but due to the overhead of the protocol the actual theoretical maximum data rate is approximately half of that. While the standard specifies 5 MHz channels, only approximately 2 MHz of the channel is consumed with the occupied bandwidth. At 2.4 GHz, 802.15.4 specifies the use of Direct Sequence Spread Spectrum and uses an Offset Quadrature Phase Shift Keying (O-QPSK) with half-sine pulse shaping to modulate the RF carrier. The graph below shows the various channels at the spacing specified by 802.15.4.

The 802.15.4 standard allows for communication in a point-to-point or a point-to-multipoint configuration. A typical application involves a central coordinator with multiple remote nodes connecting back to this central host.

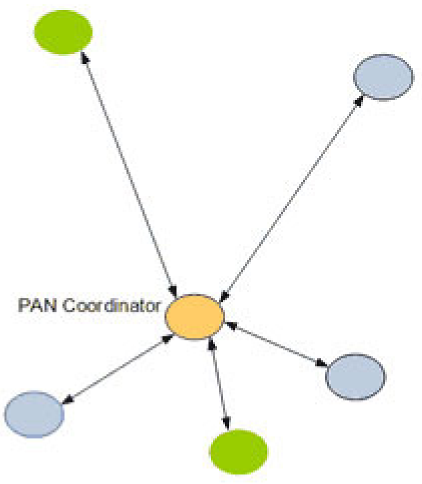


Figure 4. Typical 802.15.4 Topology

2.1.3 Zigbee

ZigBee [4] is a protocol that uses the 802.15.4 standard as a baseline and adds additional routing and networking functionality. The ZigBee protocol was developed by the ZigBee Alliance. The ZigBee Alliance is a group of companies that worked in cooperation to develop a network protocol that can be used in a variety of commercial and industrial low data rate applications. ZigBee is designed to add mesh networking to the underlying 802.15.4 radio. Mesh networking is used in applications where the range between two points may be beyond the range of the two radios located at those points, but intermediate radios are in place that could forward on any messages to and from the desired radios.

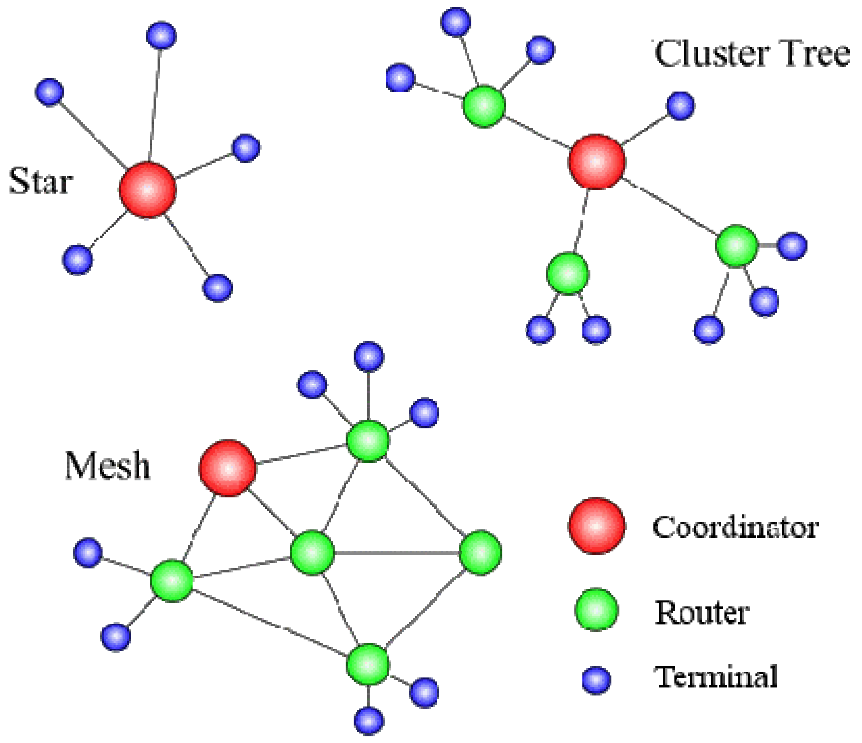


Figure 5. Zigbee Topologies

Devices in the ZigBee specification can either be used as End Devices, Routers or Coordinators. Routers can also be used as End Devices. Since the ZigBee protocol uses the 802.15.4 standard to define the PHY and MAC layers, the frequency, signal bandwidth and modulation techniques are identical.

Because ZigBee was designed for low power applications, it fits well into embedded systems and those markets where reliability and versatility are important but a high bandwidth is not. The lower data rate of the ZigBee devices allows for better sensitivity and range, but of course offers less throughput. The primary advantage of ZigBee lies in its ability to offer low power and extended battery life.

	Zigbee / 802.15.4	**Bluetooth**
Focus Application	Monitoring & Control	Device Connectivity
Batter life	Years	1 Week
Bandwidth	250kbps	720kbps
Typical Range	100+ meters	10-100 meters
Advantages	Low power, cost	Convenience

Table 1. Protocols Features

2.2 Operating systems

2.2.1 Linux

Linux [5] is a Unix-like and mostly POSIX-compliant computer operating system (OS) assembled under the model of free and open-source software development and distribution. Originally developed as a free operating system for personal computers based on the Intel x86 architecture, but has since been ported to more computer hardware platforms than any other operating system. It can also run on embedded systems which are devices whose operating system is typically built into the firmware and is highly tailored to the system.

2.2.2 TinyOS

TinyOS [6] is a lightweight operating system specifically designed for low-power wireless sensors. It differs from most other operating systems in that its design focuses on ultra-low-power operation. Rather than a full-fledged processor, TinyOS is designed for the small, low-power microcontroller motes have. Furthermore, has very aggressive systems and mechanisms for saving power.

TinyOS makes building sensor network applications easier. It provides a set of important services and abstractions, such as sensing, communication, storage, and timers. It defines a concurrent execution model, so developers can build applications out of reusable services and components without having to worry about unforeseen interactions. TinyOS runs on over a dozen generic platforms, most of which easily support adding new sensors. Furthermore, its structure makes it reasonably easy to port to new platforms. Applications and systems, as well as the OS itself, are written in the nesC language. nesC is a C dialect with features to reduce RAM and code size, enable significant optimizations, and help prevent low-level bugs like race conditions.

At a high level, TinyOS provides three things to make writing systems and applications easier:

- **Component model**, define how to write small, reusable pieces of code and compose them into larger applications.
- **Concurrent execution model**, define how components interleave their computations.
- **Application programming interfaces (API)**, services, component libraries and an overall component structure that simplify writing new applications and services.

2.3 Messaging Protocols

2.3.1 AMQP

The Advanced Message Queuing Protocol (AMQP) [7] is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns. It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

AMQP is divided up into separate layers.

Transport Layer

The AMQP Transport Layer defines a peer-to-peer protocol for transferring Messages between Nodes in the AMQP network. This layer is not concerned with the internal workings of any sort of Node, and only deals with the mechanics of unambiguously transferring a Message from one Node to another.

Nodes are named entities responsible for the safe storage and/or delivery of Messages. Messages can originate from, terminate at, or be relayed by Nodes. The AMQP Network consists of Nodes connected via Links.

A Link is a unidirectional route between two Nodes. Links attach to a Node at a Terminus. There are two kinds of Terminus: Sources and Targets. A Terminus is responsible for tracking the state of a particular stream of incoming or outgoing messages. Sources track outgoing messages and Targets track incoming messages. Messages may only travel along a Link if they meet the entry criteria at the Source.

As a Message travels through the AMQP network, the responsibility for safe storage and delivery of the Message is transferred between the Nodes it encounters. The Link Protocol manages the transfer of responsibility between the Source and Target. Nodes exist within a Container, and each Container may hold many Nodes. Nodes can be Producers, Consumers and Queues.

Messaging Layer

The transport layer defines a number of extension points suitable for use in a variety of different messaging applications. The messaging layer specifies a standardized use of these to provide interoperable messaging capabilities. This layer deals with

- Message format
- Delivery states for messages traveling between nodes
- Distribution nodes
- Sources and Targets

Transaction Layer

Transactional messaging allows for the coordinated outcome of otherwise independent transfers. This extends to an arbitrary number of transfers spread across any number of distinct links in either direction.

For every transactional interaction, one container acts as the transactional resource, and the other container acts as the transaction controller. The transactional resource performs transactional work as requested by the transaction controller. The transactional controller and transactional resource communicate over a control link which is established by the transactional controller.

Security Layer

Security Layers are used to establish an authenticated and/or encrypted transport over which regular AMQP traffic can be tunneled. Security Layers may be tunneled over one another (for instance a Security Layer used by the peers to do authentication may be tunneled over a Security Layer established for encryption purposes). The framing and protocol definitions for security layers are expected to be defined externally to the AMQP specification as in the case of TLS. An exception to this is the SASL security layer which depends on its host protocol to provide framing.

2.3.2 STOMP

STOMP [8] is a frame based protocol, with frames modelled on HTTP. A frame consists of a command, a set of optional headers and an optional body. STOMP is text based but also allows for the transmission of binary messages. The default encoding for STOMP is UTF-8, but it supports the specification of alternative encodings for message bodies.

A STOMP server is modelled as a set of destinations to which messages can be sent. The STOMP protocol treats destinations as opaque string and their syntax is server implementation specific. Additionally STOMP does not define what the delivery semantics of destinations should be. The delivery, or “message exchange”, semantics of destinations can vary from server to server and even from destination to destination. This allows servers to be creative with the semantics that they can support with STOMP.

A STOMP client is a user-agent which can act in two (possibly simultaneous) modes:

- As producer, sending messages to a destination on the server via a SEND frame
- As consumer, sending a SUBSCRIBE frame for a given destination and receiving messages from the server as MESSAGE frames.

STOMP is designed to be a lightweight protocol that is easy to implement both on the client and server side in a wide range of languages. This implies, in particular, that there are not many constraints on the architecture of servers and many features such as destination naming and reliability semantics are implementation specific.

2.3.3 MQTT

MQTT [9] is an extremely simple and lightweight messaging protocol. Its publish/subscribe architecture is designed to be open and easy to implement, with up to thousands of remote clients capable of being supported by a single server. These characteristics make MQTT ideal for use in constrained environments where network bandwidth is low or where there is high latency and with remote devices that might have limited processing capabilities and memory.

The MQTT protocol includes the following benefits:

- Extends connectivity beyond enterprise boundaries to smart devices.
- Offers connectivity options optimized for sensors and remote devices.
- Delivers relevant data to any intelligent, decision-making asset that can use it.
- Enable massive scalability of deployment and management of solutions.

MQTT minimizes network bandwidth and device resource requirements while attempting to ensure reliability and delivery. This approach makes the MQTT protocol particularly well-suited for connecting machine to machine (M2M), which is a critical aspect of the emerging concept of Cyber Physical Systems.

The MQTT protocol includes the following features:

- Open and royalty-free for easy adoption. MQTT is open to make it easy to adopt and adapt for the wide variety of devices, platforms, and operating systems that are used at the edge of a network.
- A publish/subscribe messaging model that facilitates one-to-many distribution. Sending applications or devices do not need to know anything about the receiver, not even its address.

- Ideal for constrained networks (low bandwidth, high latency, data limits, and fragile connections). MQTT message headers are kept as small as possible. The fixed header is just two bytes, and it's on demand, push-style message distribution keeps network utilization low.
- Multiple service levels allow flexibility in handling different types of messages. Developers can designate that messages will be delivered at most once, at least once, or exactly once.
- Designed specifically for remote devices with little memory or processing power. Minimal headers, a small client footprint, and limited reliance on libraries make MQTT ideal for constrained devices.
- Easy to use and implement with a simple set of command messages. Many applications of MQTT can be accomplished using just CONNECT, PUBLISH, SUBSCRIBE, and DISCONNECT.
- Built-in support for loss of contact between client and server. The server is informed when a client connection breaks abnormally, allowing the message to be re-sent or preserved for later delivery.

Basic concepts of MQTT

The MQTT protocol is built upon several basic concepts, all aimed at assuring message delivery while keeping the messages themselves as lightweight as possible.

i. **Publish & Subscribe,** The MQTT protocol is based on the principle of publishing messages and subscribing to topics, which is typically referred to as a publish/subscribe model. Clients can subscribe to topics that pertain to them and thereby receive whatever messages are published to those topics. Alternatively, clients can publish messages to topics, thus making them available to all subscribers to those topics.

ii. **Topics & Subscriptions,** Messages in MQTT are published to topics, which can be thought of as subject areas. Clients, in turn, sign up to receive particular messages by subscribing to a topic. Subscriptions can be explicit, which limits the messages that are received to the specific topic at hand or can use wildcard designators, such as a number sign (#) to receive messages for a variety of related topics.

iii. **Quality of Service levels,** MQTT defines three quality of service (QoS) levels for message delivery, with each level designating a higher level of effort by the server to ensure that the message gets delivered. Higher QoS levels ensure more reliable message delivery but might consume more network bandwidth or subject the message to delays due to issues such as latency.

iv. **Retained messages,** With MQTT, the server keeps the message even after sending it to all current subscribers. If a new subscription is submitted for the same topic, any retained messages are then sent to the new subscribing client.

v. **Clean sessions and durable connections,** when a MQTT client connects to the server, it sets the clean session flag. If the flag is set to true, all of the client's subscriptions are removed when it disconnects from the server. If the flag is set to false, the connection is treated as durable, and the client's subscriptions remain in effect after any disconnection. In this event, subsequent messages that arrive carrying a high QoS designation are stored for delivery after the connection is reestablished. Using the clean session flag is optional.

vi. **Wills,** when a client connects to a server, it can inform the server that it has a will, or a message, that should be published to a specific topic or topics in the event of an unexpected disconnection. A will is particularly useful in alarm or security settings where system managers must know immediately when a remote sensor has lost contact with the network.

Benefits of using MQTT

Using the MQTT protocol extends message queueing to tiny sensors and other remote telemetry devices that might otherwise be unable to communicate with a central system or that might be reached only through the use of expensive, dedicated networks. Network limitations can include limited bandwidth, high latency, volume restrictions, fragile connections, or prohibitive costs. Device issues can include limited memory or processing capabilities, or restrictions on the use of third-party communication software. In addition, some devices are battery-powered, which puts additional restrictions on their use for telemetry messaging.

MQTT was designed to overcome these limitations and issues and includes the following underlying principles:

- **Simplicity,** the protocol was made open so that it can be integrated easily into other solutions.
- **Use of a publish/subscribe model,** the sender and the receiver are decoupled. Thus, publishers do not need to know who or what is subscribing to messages and vice versa.
- **Minimal Maintenance,** features, such as automated message storage and retransmission, minimize the need for on-the-fly administration.
- **Limited on-the-wire footprint,** the protocol keeps data overhead to a minimum on every message.
- **Continuous session awareness,** by being aware of when sessions have terminated, the protocol can take action accordingly, thanks in part to a will feature.
- **Local message processing,** the protocol assumes that remote devices have limited processing capabilities.
- **Message persistence,** through the designation of specific QoS, the publisher can ensure delivery of the most important messages.
- **Agnostic regarding data types,** the protocol does not require that the content of messages be in any particular format.

2.3.4 MQTT-SN

MQTT-SN [10] is designed to be as close as possible to MQTT, but is adapted to the peculiarities of a wireless communication environment such as low bandwidth, high link failures, short message length, etc. It is also optimized for the implementation on low-cost, battery-operated devices with limited processing and storage resources.

MQTT-SN differences from MQTT,

i. The CONNECT message is split into three messages. The two additional ones are optional and used to transfer the Will topic and the Will message to the server.

ii. To cope with the short message length and the limited transmission bandwidth in wireless networks, the topic name in the PUBLISH messages is replaced by a short, two-byte long “topic id”. A registration procedure is defined to allow clients to register their topic names with the server/gateway and obtain the corresponding topic ids. It is also used in the opposite direction to inform the client about the topic name and the corresponding topic id that will be included in a following PUBLISH message

iii. "Pre-defined" topic ids and "short" topic names are introduced, for which no registration is required. Predefined topic ids are also a two-byte long replacement of the topic name, their mapping to the topic names is however known in advance by both the client's application and the gateway/server. Therefore both sides can start using pre-defined topic ids; there is no need for a registration as in the case of "normal" topic ids mentioned above. Short topic names are topic names that have a fixed length of two octets. They are short enough for being carried together with the data within PUBLISH messages. As for pre-defined topic ids, there is also no need for a registration for short topic names.

iv. A discovery procedure helps clients without a pre-configured server/gateway's address to discover the actual network address of an operating server/gateway. Multiple gateways may be present at the same time within a single wireless network and can co-operate in a load-sharing or stand-by mode.

v. The semantic of a "clean session" is extended to the Will feature, i.e. not only client's subscriptions are persistent, but also Will topic and Will message. A client can also modify its *Will* topic and *Will* message during a session.

vi. A new offline keep-alive procedure is defined for the support of sleeping clients. With this procedure, battery-operated devices can go to a sleeping state during which all messages destined to them are buffered at the server/gateway and delivered later to them when they wake up.

2.4 Network devices

2.4.1 Sensors / Actuators

A sensor [11] is an object whose purpose is to detect events or changes in its environment, and then provide a corresponding output, where actuator get this corresponding output and act if necessary. A sensor is a type of transducer; sensors may provide various types of output, but typically use electrical or optical signals.

Sensors utilize a wide spectrum of transducer and signal transformation approaches with corresponding variations in technical complexity. These range from relatively simple temperature measurement based on a bimetallic thermocouple, to the detection of specific bacteria species using sophisticated optical systems. There are no uniform descriptions of sensors or the process of sensing. In many cases, the definitions available are driven by application perspectives.

There are many types of sensors, some of them listed below.

- **Mechanical Sensors**

  Mechanical sensors are based on the principle of measuring changes in a device or material as the result of an input that causes the mechanical deformation of that device or material.

- **MEMS Sensors**

  The name MEMS is often used to describe both a type of sensor and the manufacturing process that fabricates the sensor. MEMS are three-dimensional, miniaturized mechanical and electrical structures, typically ranging from 1 to 100 mm, which are manufactured using standard semiconductor manufacturing techniques. MEMS consist of mechanical microstructures, micro-sensors, micro-actuators, and microelectronics, all integrated onto the same silicon chip.

- **Optical Sensors**

  Optical sensors work by detecting waves or photons of light, including light in the visible, infrared, and ultraviolet (UV) spectral regions. They operate by measuring a change in light intensity related to light emission or absorption by a quantity of interest. They can also measure phase changes occurring in light beams due to interaction or interference effects.

- **Semiconductor Sensors**

Semiconductor sensors have grown in popularity due to their low cost, reliability, low power consumption, long operational lifespan, and small form factor. This type of sensors are Gas Sensors, Temperature sensors.

- **Electrochemical Sensors**

An electrochemical sensor is composed of a sensing or working electrode, a reference electrode, and, in many cases, a counter electrode. These electrodes are typically placed in contact with either a liquid or a solid electrolyte. In the low-temperature range (<140° C), electrochemical sensors are used to monitor pH, conductivity, dissolved ions, and dissolved gases.

- **Biosensors**

Biosensors use biochemical mechanisms to identify and analyze of interest in chemical, environmental (air, soil, and water), and biological samples (blood, saliva, and urine). The sensor uses an immobilized biological material, which could be an enzyme, antibody, nucleic acid, or hormone, in a self-contained device. The biological material being used in the biosensor device is immobilized in a manner that maintains its bioactivity.

2.4.2 Message Brokers

A message broker is an intermediary module which translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. Message brokers are elements in telecommunication networks where programs (software applications) communicate by exchanging formally-defined messages. It mediates communication amongst applications, minimizing the mutual awareness that applications should have of each other in order to be able to exchange messages, effectively implementing decoupling. The main purpose of a message broker it to take incoming messages from applications and perform some action on them. Some of the action define below.

- Route messages to one or more of many destinations.
- Transform messages to an alternative representation.

- Perform message aggregation, decomposing messages into multiple messages and sending them to their destination, then recomposing the responses into one message to return to the user.
- Interact with an external repository to augment a message or store it.
- Invoke Web services to retrieve data.
- Respond to events or errors.
- Provide content and topic-based message routing using publish–subscribe pattern.

Well known message brokers are described below.

Really Small Message Broker

Really Small Message Broker (aka RSMB) [12] is a small server that uses MQ Telemetry Transport (MQTT) for lightweight, low-overhead messaging. It enables messaging to and from tiny devices such as sensors and actuators over networks that might have low bandwidth, high cost, and varying reliability. "Publishers" send messages to the broker, which then distributes the messages to the "subscribers" who have requested to receive those messages.

RSMB has a "bridge" that enables connections to other MQTT-capable servers; this bridge allows messages to be passed between RSMB instances as well as to other MQTT servers such as ActiveMQ. RSMB can run in embedded systems in order to provide a messaging infrastructure in remote installations and pervasive environments. Given Really Small Message Broker's low memory requirements, it can help extend the reach of the MQTT messaging infrastructure to the smallest components.

Mosquitto

Mosquitto [13] provides a lightweight server implementation of the MQTT and MQTT-SN protocols, written in C. The reason for writing it in C is to enable the server to run on machines which do not even have capacity for running a JVM. Sensors and actuators, which are often the sources and destinations of MQTT and

MQTT-SN messages, can be very small and lacking in power. This also applies to the embedded machines to which they are connected.

As well as accepting connections from MQTT client applications, Mosquitto has a bridge which allows it to connect to other MQTT servers, including other Mosquitto instances. This allows networks of MQTT servers to be constructed, passing MQTT messages from any location in the network to any other, depending on the configuration of the bridges.

ActiveMQ

ActiveMQ [14] is an open source, Java Message Service (JMS) 1.1–compliant, message-oriented middleware (MOM) from the Apache Software Foundation that provides high availability, performance, scalability, reliability, and security for enterprise messaging, which is licensed using the Apache License. The goal of ActiveMQ is to provide standards-based, message-oriented application integration across as many languages and platforms as possible. Implements the JMS spec and offers dozens of additional features and value on top of this spec.

2.4.3 Gateways

Gateways perform protocol translation between different networks. A gateway can operate at any network layer, and, unlike a router or a switch, a gateway can communicate using more than one protocol. PCs, servers, and M2M devices can function as gateways, although they are most commonly found in routers. In a sensor network, a gateway is responsible for interfacing the data from the sensor nodes to another network that uses a different protocol, and delivering commands back from that network to the nodes. Gateways work on OSI layers 4-7.

3. Network Architecture

As shown in the figure 6, above, the network architecture that we propose consists from multiple services and communication protocols.

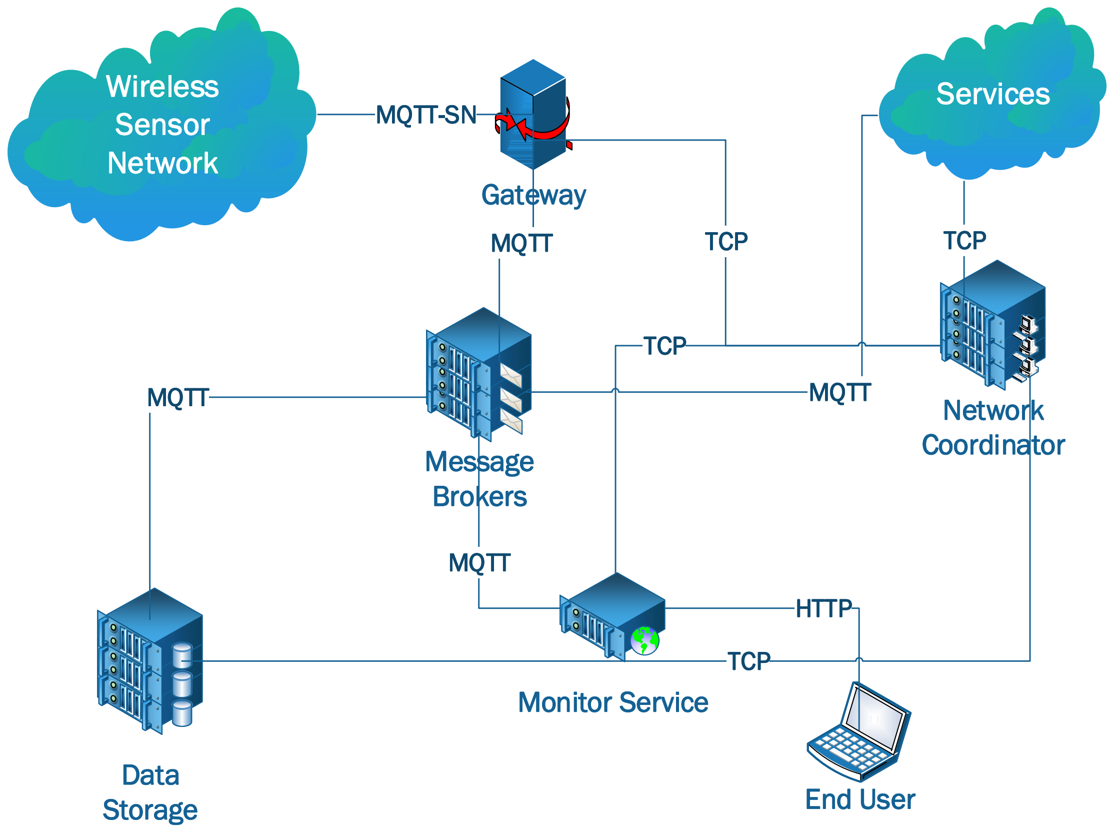


Figure 6. Network Architecture

In the wireless sensor network, we implement basic applications that communicate over Bluetooth [2] and 802.15.4 [3] with our gateway. Gateway is an intermediate component, vital, whose main scope is to translate data which come from WSN to appropriate format in order to transmit them over IP network. In the gateway we implement interfaces for the communication with the WSN, such as Bluetooth and CC2420 adapters, as well as interfaces for the communication with the IP network, such as Messaging client. Each service in the network must register itself, in order to inform other services about its presence. This achieved using the Network coordinator, which each service register itself, and store what it is offer to the network. Network coordinator will used from other services, to retrieve configurations, as well as for service discovery inside the network. The communication between gateways and the services in the IP network made through Message Broker, which is responsible to forward the messages to all interested parties. Each service must inform Message Broker about the data wants to receive. Data storage service, used as the basic storage, firstly for storing collected data from WSN, and then for any type data needs to be stored. Finally we implement the monitor service, in order to help end users, configure and monitor both WSN and IP network, through a simple web interface.

3.1 Communication Protocols

As depicted in figure 6 our network architecture, consists from multiple communication protocols in order to accomplish communication in a heterogeneous environment. On the wireless sensor network the technologies used from the sensors and actuators to communicate with gateway and in turn with the IP network are Bluetooth and 802.15.4 compliant technologies, and all these communications abstracted using the MQTT-SN protocol.

The Gateway (GW) needs many connections, in order to accomplish its tasks. In our architecture beyond the connection with the WSN, establish two more connections further, one with the message broker, which use the MQTT protocol for send and receive messages from the IP network and the WSN respectively, and the other with the network coordinator, explained in more detail in a later chapter, via a TCP connection. All the IP network components, except network coordinator, that want to send to and receive messages, must connect with message broker using the MQTT protocol. Finally the HTTP protocol used for connecting end users with the network in order to configure, parameterized and monitor the traffic on both WSN and IP network.

3.2 Gateway

Depending on how a gateway performs the protocol translation between MQTT-SN and MQTT, we can differentiate between two types of gateways, namely *transparent* and *aggregating* Gateways [10], as shown in Figure 7.

Transparent Gateway

For each connected MQTT-SN client a transparent GW will setup and maintain a MQTT connection to the MQTT server. This MQTT connection is reserved exclusively for the end-to-end and almost transparent message exchange between the client and the server. There will be as many MQTT connections between the GW and the server as MQTT-SN clients connected to the GW. The transparent GW will perform a "syntax" translation between the two protocols. Since all message exchanges are end-to-end between the MQTT-SN client and the MQTT server, all functions and features that are implemented by the server can be offered to the client. Although the implementation of the transparent GW is simpler when compared to the one of an aggregating GW, explained below, it requires the MQTT server to support a separate connection for each active client. Some MQTT server implementations might impose a limitation on the number of concurrent connections that they support.

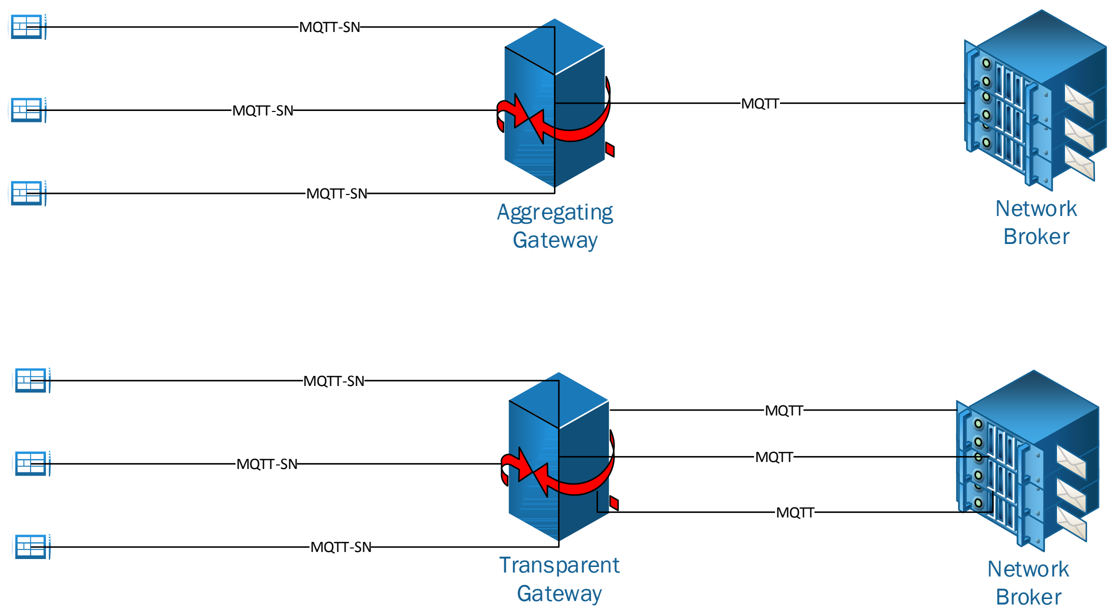


Figure 7. Gateway types

Aggregating Gateway

Instead of having a MQTT connection for each connected client, an aggregating GW will have only one MQTT connection to the server. All message exchanges between a MQTT-SN client and an aggregating GW end at the GW. The GW then decides which information will be given further to the server. Although its implementation is more complex than the one of a transparent GW, an aggregating GW may be helpful in case of WSNs with very large number of Sensors-Actuators because it reduces the number of MQTT connections that the server has to support concurrently.

3.3 Network Coordinator

In order to run large systems correctly and efficiently within these systems should have some sort of agreement among themselves. It is difficult to design a large system, and it's even harder when a collection of individual computing entities are programmed to function together. In this point we need a component inside our network that will be responsible for:

1. Configuration management
2. Network Rules
3. Detect of node leave / join
4. Services registration and discovery
5. Network tasks
6. Synchronization

With the presence of the network coordinator we can accomplish:

- **Resource sharing**: This refers to the possibility of using the resources in the system, such as storage space, computing power, data, and services from anywhere, and so on.
- **Extendibility**: This refers to the possibility of extending and improving the system incrementally.
- **Concurrency**: This refers to the system's capability to be used by multiple nodes at the same time.
- **Performance and Scalability**: This ensures that the response time of the system doesn't degrade as the overall load increases.
- **Fault Tolerance**: This ensures that the system is always available even if some of the components fail or operate in a degraded mode.
- **Abstraction through APIs**: This ensures that the system's individual components are concealed from the end users, revealing only the end services to them.

3.4 Network Broker

In our architecture network broker is the responsible component which receive and forward messages to all interested parties. The broker, we use in our infrastructure is Apache ActiveMQ [14].

ActiveMQ is an open source, Java Message Service (JMS) – compliant, message-oriented middleware (MOM) from the Apache Software Foundation that provides high availability, performance, scalability, reliability, and security for enterprise messaging. The goal of ActiveMQ is to provide standards-based, message-oriented application integration across as many languages and platforms as possible. ActiveMQ implements the JMS spec and offers dozens of additional features and value on top of this spec. One of the key characteristics of ActiveMQ is that support multiple messaging connectors, such as AMQP, STOMP, MQTT.

Some of the key features described below:

- **JMS Compliance**

  ActiveMQ is an implementation of the JMS 1.1 spec. The JMS spec provides important benefits and guarantees, including synchronous or asynchronous message delivery, once-and-only- once message delivery, message durability for subscribers, and much more.

- **Connectivity**

  ActiveMQ provides a wide range of connectivity options, including support for protocols such as HTTP/S, IP multicast, SSL, AMQP, STOMP, TCP, UDP, MQTT, and more. Support for such a wide range of protocols equates to more flexibility.

- **Pluggable persistence and Security**

  ActiveMQ provides multiple flavors of persistence and you can choose between them, default is KahaDB [15]. Also, security in ActiveMQ can be completely customized for the type of authentication and authorization that’s best for your needs. ActiveMQ also supports its own simple style of authentication and authorization using properties files as well as standard JAAS login modules.

- **Integration with application servers**

It’s common to integrate ActiveMQ with an application server, such as Apache Tomcat, Jetty, Apache Geronimo, and JBoss.

- **Many client APIs**

ActiveMQ provides client APIs for many languages besides just Java, including C/C++, .NET, Perl, PHP, Python, Ruby, and more. This opens the door to opportunities where ActiveMQ can be utilized outside of the Java world.

- **Clustering**

Many ActiveMQ brokers can work together as a federated network of brokers for scalability purposes. This is known as a network of brokers and can support many different topologies.

3.5 Services

Services [16] are small autonomous components in the network, which expose in our network what types of data can handle or process and what types of data can offer to the network, in order other services to use to accomplish a task. An example of such a service is the monitoring of the wireless sensor network. A service like that will be processing the data which receive and then inform all the interested services about the changes made. A service can have configurations, parameters and rules that can be defined in three different levels

1. **Network level**, all the configuration, parameters and rules will be fetched from a service that hold this type of information, like Network coordinator or Data Storage service.
2. **Service level**, all the configuration, parameters and rules will be defined inside the service.
3. **Mixed level**, some the configuration, parameters and rules will be fetched from a service that hold this type of information, and some of them will be defined inside the service.

The presence and discovery of a service in the network can be accomplished through the network coordinator. Each service must create an entry to network coordinator and it is responsible to define all the necessary information for other services to discover and communicate with it. A typical entry of a service in the coordinator may have

- **IP Address or Domain**, define the location of service.
- **Connection type**, define the type of the communication e.g. MQTT
- **Incoming message**, define the incoming message format, that can handle
- **Outgoing message**, define the format of the outgoing message.

Some of the services our network contains described below.

3.5.1 Data storage

Data storage component is a cluster of databases which will be used by the framework for storing and retrieving data or configurations that services in the network wants. An example of such data will be a list of topic ids of the MQTT-SN protocol, and the string representation of topic id to a topic name of the MQTT protocol. This type of data will be fetched from a gateway in order to match topic ids with the topic names between WSN and IP network, and then route the data to defined topic name.

3.5.2 Monitoring / Configuration

Monitoring-configuration component is web interface where end users, such as administrators, could configure both IP network and WSN. An example of such a configuration, is when an administrator insert to database a topic id, and a topic name for this id. Another possibility that will be provided from this component is to allow administrator to watch wireless sensor network traffic and measurements, and make all the necessary actions, such as parametrized network configurations and rules.

4. Infrastructure Implementation

4.1 Network Coordinator

In our network as a coordination service we use Apache Zookeeper. Apache ZooKeeper [17] is a software project of the Apache Software Foundation; it provides an open source solution to the various coordination problems in large distributed systems.

Apache ZooKeeper, as a centralized coordination service, is distributed and highly reliable, running on a cluster of servers called a ZooKeeper ensemble. Distributed consensus, group management, presence protocols, and leader election are implemented by the service so that the applications do not need to reinvent the wheel by implementing them on their own. On top of these, the primitives exposed by ZooKeeper can be used by applications to build much more powerful abstractions to solve a wide variety of problems.

Coordinator data model

ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical namespace of data registers. The namespace looks quite similar to UNIX filesystem. The data registers are known as znodes in the ZooKeeper nomenclature. The data in a znode is typically stored in a byte format, with a maximum data size in each znode of no more than 1 MB. A typical structure on Apache Zookeeper shown in figure below

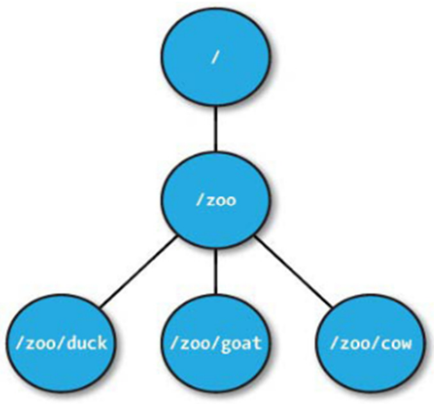


Figure 8. Zookeeper structure

Type of znodes

Zookeeper defines two types of znodes, *Persistent* and *ephemeral* which each of them can be defined as *sequential* or not. A sequential znode is assigned a sequence number by ZooKeeper as a part of its name during its creation. The value of a monotonously increasing counter (maintained by the parent znode) is appended to the name of the znode. The counter used to store the sequence number is a signed integer (4 bytes). It has a format of 10 digits with 0 (zero) padding. For example, look at /path/to/znode-0000000001. This naming convention is useful to sort the sequential znodes by the value assigned to them.

Persistent znodes

Persistent znodes have a lifetime in the Zookeeper's namespace until they are explicitly deleted. A znode can be deleted by calling the delete API call. It's not necessary that only the client that created a persistent znode has to delete it. Note that any authorized client of the ZooKeeper service can delete a znode. Persistent znodes are useful for storing data that needs to be highly available and accessible by all the components of a distributed application. For example, an application can store the configuration data in a persistent znode. The data as well as the znode will exist even if the creator client dies.

Ephemeral znodes

An ephemeral znode is deleted by the ZooKeeper service when the creating client's session ends. An end to a client's session can happen because of disconnection due to a client crash or explicit termination of the connection. Even though ephemeral nodes are tied to a client session, they are visible to all clients, depending on the configured Access Control List (ACL) policy. An ephemeral znode can also be explicitly deleted by the creator client or any other authorized client by using the delete API call. An ephemeral znode ceases to exist once its creator client's session with the ZooKeeper service ends. Hence, in the current version of ZooKeeper, ephemeral znodes are not allowed to have children. The concept of ephemeral znodes can be used to build distributed applications where the components need to know the state of the other constituent components or resources.

Coordinator Structure

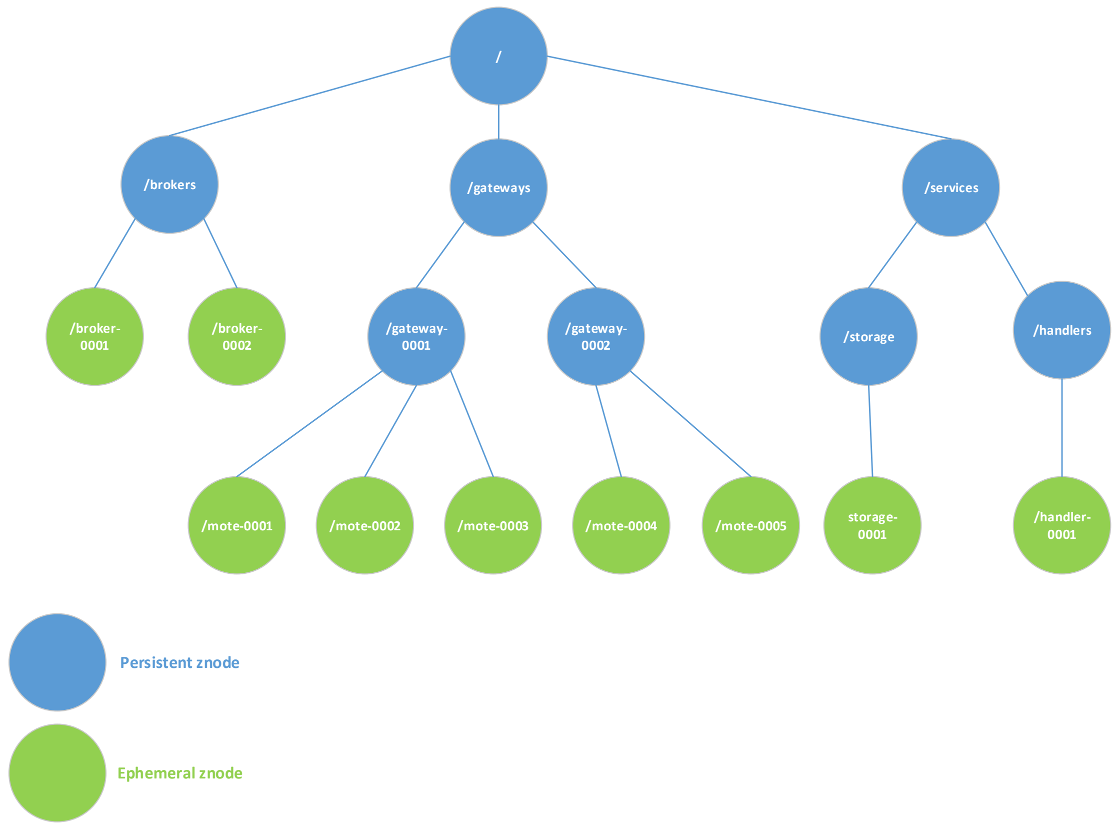


Figure 9. Network Coordinator Structure

As shown in figure 9, our structure consists of three main znodes, namely brokers, gateways and services, which cannot be deleted. Under the */brokers* znode will be the available network brokers, which every service or component can fetch them, and then decide with which broker wants to connect. Under the */gateways* znode, we create persistent znodes with the gateway name in format *gateway-<identifier>*, these znodes are created from the network administrators. When the gateway connect to our IP network, is responsible to change the status of the znode, in order to inform all other interested components of the network that this gateway is online. Each gateway znode has children which are sensors or actuators of a WSN. These ephemeral znodes are created when a WSN node connected with the gateway. Gateway is responsible to create, update and delete each znode. The last znode, */services,* as the name implies contains all the network available services. The first child of the */services* znode implies the type of service, for example, as shown in figure 9, znode */storage* contains available data storage services. The naming for each available service is just like gateway znode.

4.2 Network Broker

As we mentioned in previous chapter in our framework, as main Broker in the IP network we use Apache ActiveMQ [14]. In this chapter will be describe some of the configurations on the broker. The basic configuration of the broker located in activemq.xml file. As the suffix implies the data format of the configuration is in XML, where each parameter defined inside tags.

Transport Mechanism

Transport connectors are defined inside the transportConnectors tag. A typical connector definition appears below:

```
<transportConnector name="mqtt" uri="mqtt://0.0.0.0:1883"/>
```

In the definition above we enable the *mqtt* messaging protocol, which defined with the attribute *name*. The *uri* specifies the connector that should be used. In the preceding line, the 0.0.0.0 address means that ActiveMQ will listen on all interfaces and address on the server. For instance, if the address was 127.0.0.1, then only local connections will be allowed, and no one could be connect to the broker from another machine.

Authentication Mechanism

ActiveMQ allows us to define authentication mechanism in order to connect a component to the broker. The authentication mechanism is defined inside <*plugins*> tag, as shown above

```
<plugins>
  <simpleAuthenticationPlugin anonymousAccessAllowed="false">
     <users>
        <authenticationUser username=".." password="…" groups=""/>
     </users>
  </simpleAuthenticationPlugin>
</plugins>
```

The tag used for the authentication is simpleAuthenticationPlugin and with attribute anonymousAccessAllowed=”false” we inform broker not to accept anonymous connections. Inside the users tags we define the authentication credentials. *Groups* attribute is the groups of the user defined, which inform broker about the access rights of the defined users.

4.3 Gateway

Gateway is the main entrance of the data collected from sensors in the physical environment, which makes it, a fundamental component to the network. Data came for sensors, are asynchronously data, different types of data, from different communication technologies, that must be parsed, analyzed, translate and finally forwarded to the network. Gateway work as mediator, who implements adapters for parse the data from different communication technologies, such as Bluetooth and 802.15.4, analyze them and extract the useful information, create compatible messages from the information extracted and finally forward them to the network or not. Without the presence of the gateway will not be possible for the various communication technologies talk to each other.

4.3.1 Architecture

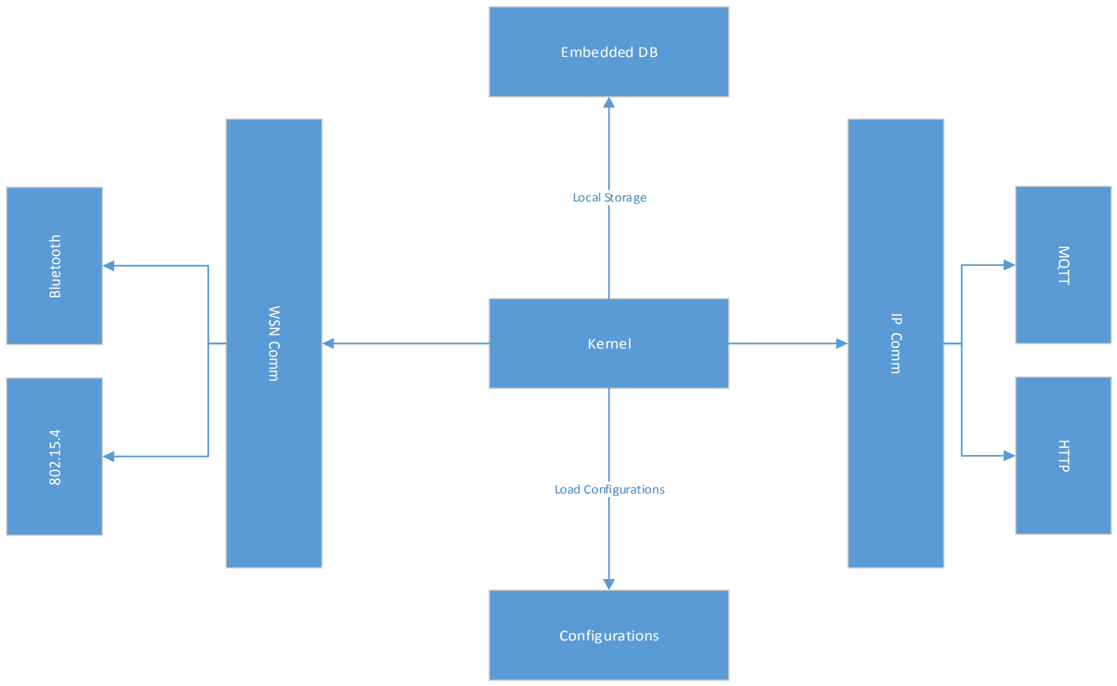


Figure 10. Gateway Architecture

Figure 10 show our gateway architecture. The gateway consists from five main components named Kernel, Configurations, Embedded DB, IP Communicator and finally the WSN Communicator. The Kernel component is the brain of the gateway, which is responsible for:

1. Loading the configurations from the *Configuration* component.
2. Initialize and hold the connection with the *Embedded DB*.
3. Initialize the communication with the IP Network and WSN, throw the components (*WSN Comm.* & *IP Comm.*).
4. Route data and messages between each main components.

4.3.2 Configurations

All the configurations for the gateway will be defined in the *gateway.json* file. The format of the configurations as indicated by the suffix of the file is JSON [20]. The reason that will choose JSON as configuration format instead of XML is that JSON is more readable than XML, lightweight and the configurations in both coordinator and Database are stored in JSON format, and we want to keep uniform schemas. Some of the basic configuration parameters of the gateway are described in the table below.

Parameter	Explanation	Values
gwid	Gateway identifier, Must be the same with that name that stored in Storage and Coordinator	gateway-0001
type	Define the type of gateway, for more info check *Chapter 3.2*	aggregate | transparent
coordinator	Information about the connection with the coordinator of the network	{ "host": "domain", "port": 2181, "username": "", "password": "" }
interfaces	List with connection interfaces for the WSN,	1. ble 2. bt

	can be more than one.	3. zigbee 4. 802154
connection_mode	Gateway connection mode to WSN, explained below.	1. force_connect 2. stand_by 3. invisible 4. closed
communication_protocols	List with connections to the backend network, can be more than one.	1. mqtt 2. amqp 3. stomp 4. websocket 5. soap 6. http
data_process	Define how the gateway will be handle the data from the sensor networks, explained below	1. pass_through 2. buffering 3. calculate
subscriptions	Contains a list with predefined subscriptions to topics or queues of the gateway	["/wsn/framework"]
edb	Embedded Database name	gatewayedb

Table 2. Gateway Configurations

Gateway connection mode

Gateway connection mode define how the gateway will behave with the wireless sensor network. As mentioned in the table above, gateway can be in four different connection modes.

1. **force_connect**, in this mode gateway will start searching for devices in the network, and when finds a mote will try to connect with it. This is the default connection mode for the gateway.
2. **stand_by**, the gateway will send a broadcast message in order to inform the network motes for its presence, and then will wait from the sensor motes to make a connection request.

3. **invisible**, in this mode the gateway will be online, but will not send a broadcast message in order to inform the network for its presence. Only the devices know its existence will try to connect with the gateway.
4. **closed**, in closed mode the gateway will be online nobody knows about its existence and no connections can be established.

Data processing

This configuration parameter will inform the gateway how to handle data derived from the wireless sensors network. Gateway have three different modes for the data processing.

1. **pass_through**, in this mode every incoming data will be pass directly to the IP network.
2. **buffering**, the incoming data will be stored locally to the gateway. In this mode the gateway will be defined at startup or will be switched when a connection with IP network lost.
3. **process**, in this mode gateway will be process the incoming data according to the given implementation, and then according to the rules defined will process the incoming data.

4.3.3 Embedded DB

Our solution provide an embedded DB for a series of actions that will persist in the gateway. Instead of using just files for this persistence we create a key-value database, which is based on mapDB [18]. MapDB is an embedded database engine for Java. It provides Maps and other collections backed by disk or off-heap memory storage. MapDB is not a database, but an engine. It is not complete solution, but a set of building blocks such as: memory allocators, caches, storages, indexes, transaction wrappers etc. This gives MapDB lot of flexibility and space for performance optimizations and small footprint.

We expose a very simple API for gateway internal components to use the database, which consists from five methods:

- **save**, add a new value to a collection.

- **update**, update an existing value in a collection.
- **delete**, delete an existing value in a collection
- **get**, get a specific value from a collection
- **fetch**, get a list of values from a collection

In the backend of the database we organize our data into collections. Collections are groups of key-value entry. For example, in order to keep track which sensors devices are connected to gateway, we have the devices collection.

The most important actions that will performed from the embedded DB are:

1. Store of the WSN devices and their properties.
2. Devices subscriptions.
3. Internal gateway rules, such as handlers for the data.
4. Buffering functionality for WSN sensor data.
5. Data formats, in case of need to read the data locally.
6. Routing rules.

4.3.4 IP Communicator

The IP communicator is responsible to set up the connections defined in the configurations of the gateway with the IP network, as name implies. As shown in figure 10, the gateway configured to set up two communications with the IP network, HTTP and MQTT, more than these connections can be define. For the HTTP communication the client will create an HTTP client and will wait for data sourced from the WSN in order to send to HTTP API. The creation of a MQTT connection, and in more general a message passing connection is more complicated than a simple HTTP client. Connection procedure shown in figure 11.

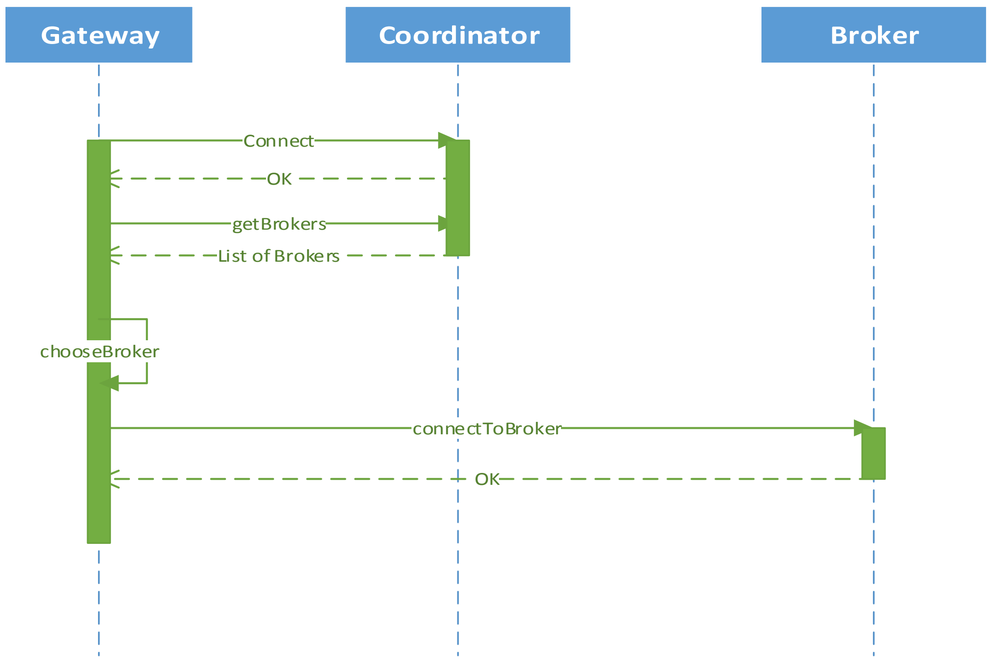


Figure 11. Message passing protocol connection procedure

After the connection of the gateway with the coordinator, gateway will fetch from coordinator all the available brokers, and in our case will fetch all the brokers that support MQTT protocol. Then will choose one broker from the list and will try to connect with it. The choice of the broker is based on a single ping, and the broker with the smallest response time will be selected.

4.3.5 WSN Communicator

The WSN Communicator is responsible to initialize gateway wireless interfaces based on the configurations. As shown in figured 10, the gateway configured to initialize two communications with the wireless sensor network, Bluetooth and 802.15.4. After the initialization of the interfaces its component is autonomous which will forward all the data sourced from the WSN to Communicator which is responsible, according to rules to forward or store the data. The same thing will happen when the data come from the other side, IP network. Its subcomponent of the WSN communicator is responsible, according to gateway *connection_mode* configuration of the gateway to start or not, sensor nodes discovery and connection.

4.4 Services

4.4.1 Services Architecture

For the definition of the services in the network we use the micro-services architectural patterns. With this type of architecture we can plug any component (service) on the network and be available directly to all others service to use it. To this help the presence of the coordinator, which every service register its self on it. Any new service wants to connect to the network, after its registration will fetch from the coordinator information that it needs for.

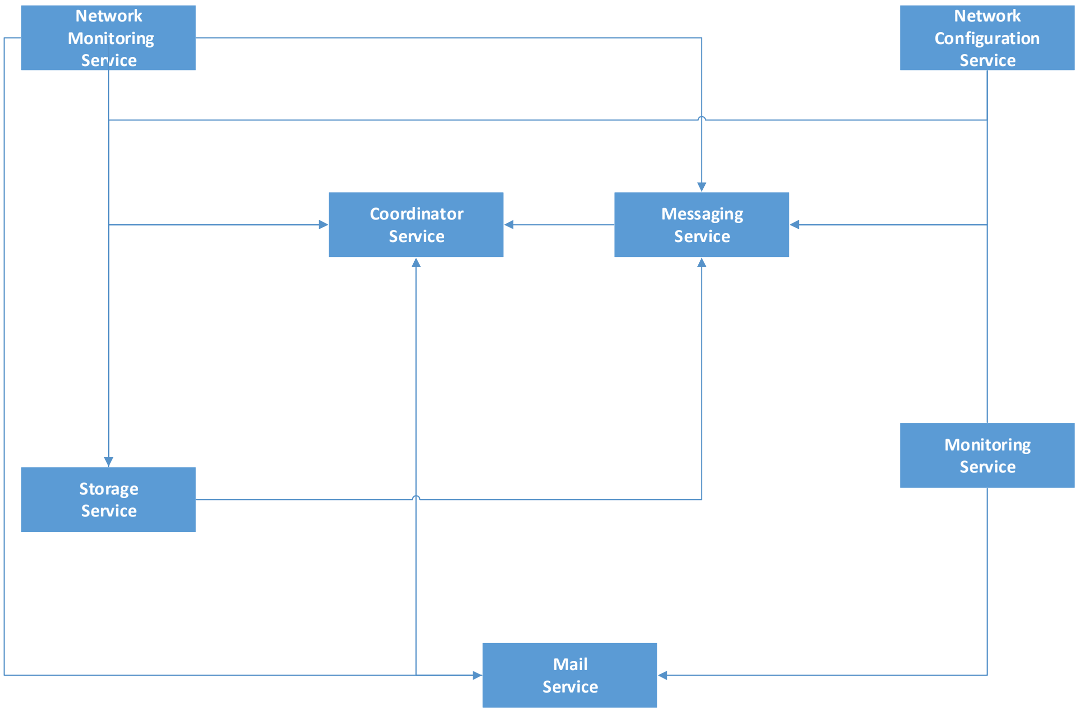


Figure 12. Microservices Architecture

For example, the monitoring service wants to send an email for an abnormal situation on the sensor network. After connecting to the coordinator service, will fetch all the services that provide email service, with the configurations that will be used later to send the email.

Figure 12 shows a set of services in our IP network. All this services are connected together using multiple communication protocols. Some services such as Mail service expose a simple HTTP API in order to other services to communicate with. Storage service is provided through MQTT message protocol. Finally other services such as Network monitoring service does not provide any communication interface for receiving data, but connect to other services in order to fetch data from them.

4.5.2 Data storage

In our storage service the main Database is MongoDB [19]. MongoDB is popular as it is fast and flexible with excellent community support. It is a document-oriented database, fitting in somewhere between Key-Value stores and traditional relational databases. MongoDB stores documents as BSON, which is effectively binary-encoded JSON [20]. When you run a query you get a JSON object returned (or a string in JSON format, depending on the driver). Look at the following code snippet for example:

{ "_id" : ObjectId("4ffbc45c35097b5a1583ad71"), "addr" : "001060AA36F8", "status" : 1}

So, a document is a set of keys (for example, addr) and values (for example, *001060AA36F8*). The _id entry is a unique identifier that the underlying MongoDB driver will—by default—create for each new document.

Some features of MongoDB described below

- **Document data model**, MongoDB's data model is document-oriented. A document is essentially a set of property names and their values. The values can be simple data types, such as strings, numbers, and dates.
- **Ad-hoc queries**, Ad hoc queries are easy to take for granted if the only databases you've ever used have been relational. But not all databases support dynamic queries.
- **Secondary indexes**, Secondary indexes in MongoDB are implemented as B-trees. B-tree indexes, also the default for most relational databases, are optimized for a variety of queries, including range scans and queries with sort clauses. By permitting multiple secondary indexes, MongoDB allows users to optimize for a wide variety of queries. You can create up to 64 indexes per collection.
- **Replication**, MongoDB provides database replication via a topology known as a replica set. Replica sets distribute data across machines for redundancy and automate failover in the event of server and network outages.
- **Speed and Durability**, Write speed can be understood as the volume of inserts, updates, and deletes that a database can process in a given time frame, where Read speed can be understood as the volume of read in a given time

frame. Durability refers to level of assurance that these write operations have been made permanent. Users control the speed and durability trade-off by choosing write semantics and deciding whether to enable journaling.

- **Scaling**, MongoDB use vertical scaling or scaling up. Vertical scaling has the advantages of being simple, reliable, and cost-effective up to a certain point.

Database Structure

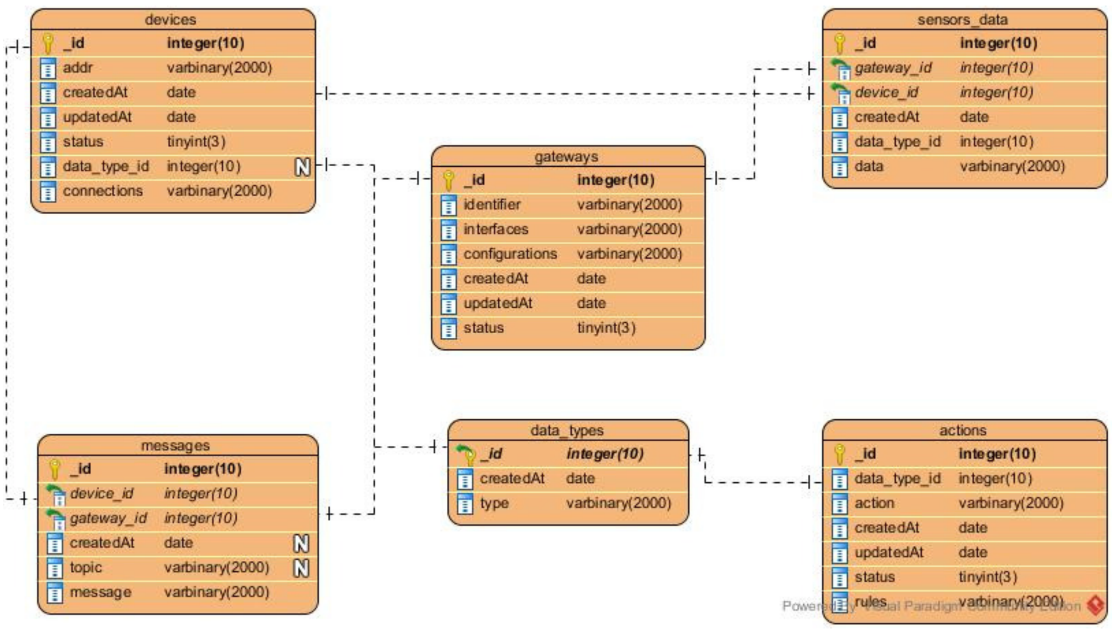


Figure 13. Database Structure

As depicted on figure 13, our database implementation is simple, it’s only contains 6 collections, we analyze each collection below,

- **data_types**, contain sensor data type, e.g. accelerometer.
- **devices**, contain all the known wireless sensor network devices, what is worth mentioning is the *data_type_id* and *connections* fields, where we identify which data the device carry(data_type_id) and which is the connections with our network, e.g. Bluetooth.
- **gateways,** contain all the gateways of our network. A typical document for a gateway shown in Table 2 below

_id	0001
identifier	gateway-0001

interfaces	["802154", "BT"]
configurations	{ "broker":"fetch", "connection_mode":"force_connect", "type" : "aggregating", "protocols":["mqtt","http"] }
createdAt	2015-10-10 23:00:00
updatedAt	2015-10-10 23:00:00
status	1 (Enabled)

Table 3. Gateway document

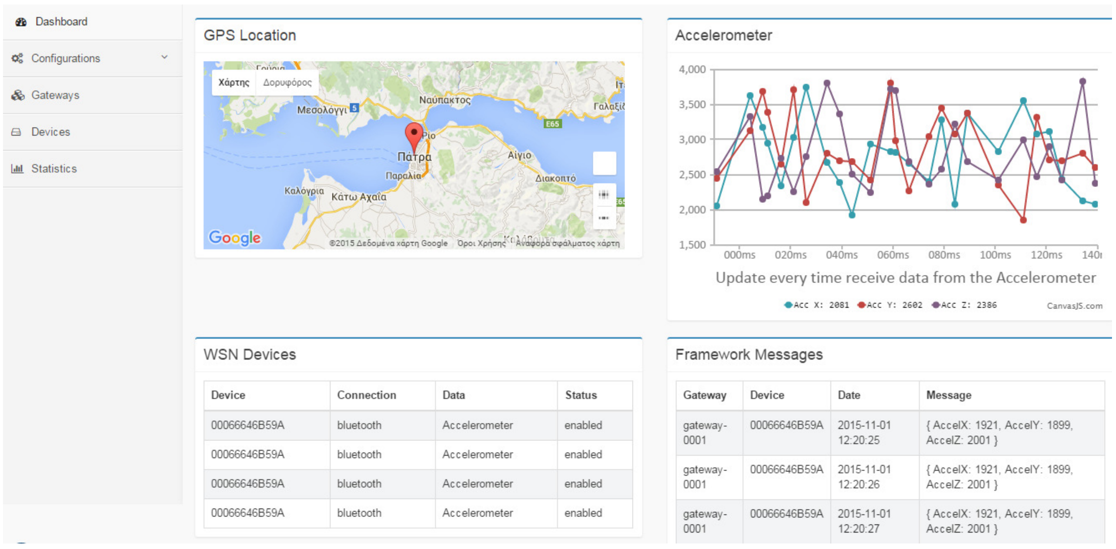
- **messages**, logging all the commands, configuration messages, parameterized messages that will be send to the network.
- **sensors_data**, storing all the data coming from the sensors in WSN.
- **actions**, contain rules how to handle different types of data. For example, assume that we have a sensor that send us data with the values of temperature. When the temperature goes down of 20 degrees (it will be defined in the rules fields of the document) make the action defined in the action field. A typical entry of such an action is shown in Table 3 below.

_id	11292812
data_type_id	5 (data_types → temperature)
action	send_email
createdAt	2015-10-10 23:00:00
updatedAt	2015-10-10 23:00:00
status	1 (Enabled)
rules	{ "minimum": 20, "maximum": 35 }

Table 4.Temperature rule document

4.5.3 Monitoring

After we have created entire network, it is time to implement some tools in order to monitoring, configuring and parameterized both wireless sensor network and IP network. For this purpose we have created a web-based platform in order to achieve this, as shown in the figure below.



WSN Devices

Device	Connection	Data	Status
00066646B59A	bluetooth	Accelerometer	enabled
00066646B59A	bluetooth	Accelerometer	enabled
00066646B59A	bluetooth	Accelerometer	enabled
00066646B59A	bluetooth	Accelerometer	enabled

Framework Messages

Gateway	Device	Date	Message
gateway-0001	00066646B59A	2015-11-01 12:20:25	{ AccelX: 1921, AccelY: 1899, AccelZ: 2001 }
gateway-0001	00066646B59A	2015-11-01 12:20:26	{ AccelX: 1921, AccelY: 1899, AccelZ: 2001 }
gateway-0001	00066646B59A	2015-11-01 12:20:27	{ AccelX: 1921, AccelY: 1899, AccelZ: 2001 }

Figure 14. Web Panel

Monitoring panel we will be contain everything we need in order customized our network configurations and monitoring network sensors. A sample of this web panel which is shown in the figure above contains the location of a configured wireless sensor network, panel *GPS location*, a streaming data of an accelerometer sensor from our network and finally the main messages of the network, panel *Framework messages,* and the available sensor devices. The technologies used for the implementation of the monitoring tools is HTML5, CSS3, and JavaScript.

5. Demos

In this chapter we will present two simple demos, with the functionality of our system. In the first demo we will describe in detail, the initialization process of core network services, and then we will present additional services that our system needs in order to provide the basic functionality which is the collection of data transmitted from wireless sensor networks. After setup the basic services in our system, we will present an easy to understand second demo, which will show how the system can send notifications according to rules that we define and the data collected from sensors.

5.1 System basic functionality initialization

In the first demo, we will describe a step by step service initialization process, in order to our system to be able to collect data from sensor motes in WSN.

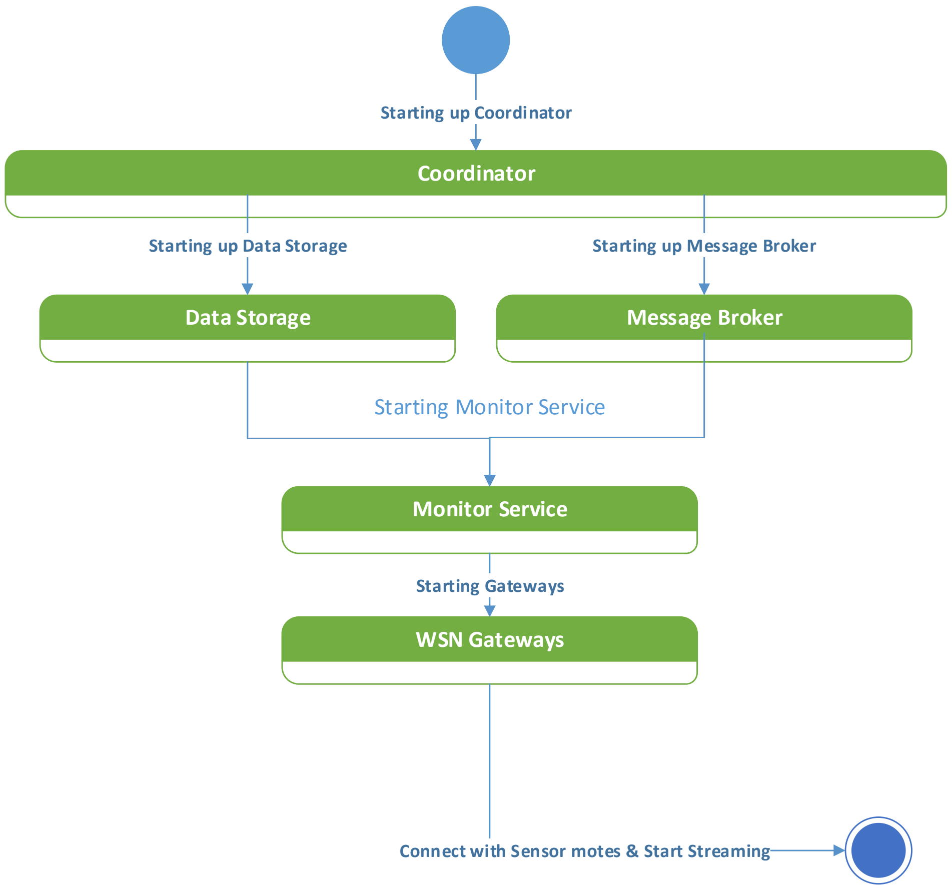


Figure 15. System startup procedure

The services that will start in this demo are:

1. Coordinator
2. Message Broker
3. Data Storage
4. Monitor Service
5. WSN Gateway

The first service that need to be started is the Coordinator. We need the coordinator, for the services to retrieve configurations in order to start operate inside the network. As reported in previous chapters coordinator contains configurations and service registration data. After starting the coordinator, we start the Message Broker, in order network services to publish and receive messages. Once the message broker is started, it's responsible to create an entry to the coordinator with information that will be used later by services. The most important information are, *host* and *supported protocols*. The host is the IP or domain name of the message broker, and supported protocols is the message passing protocols supported by this broker, like MQTT.

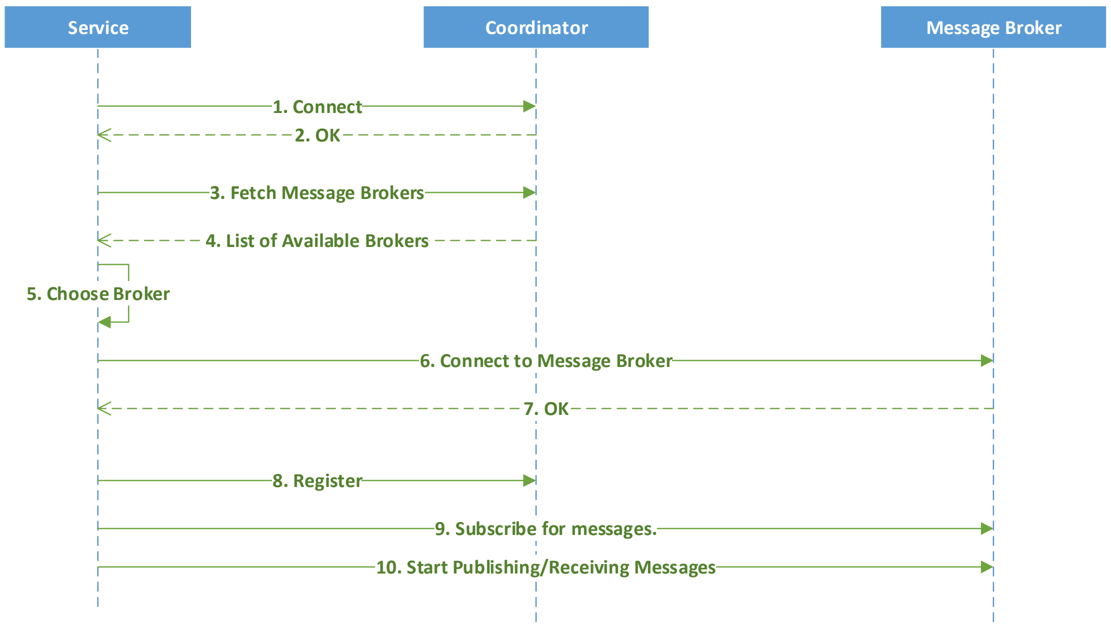

Service
Coordinator
Message Broker
1. Connect
2. OK
3. Fetch Message Brokers
4. List of Available Brokers
5. Choose Broker
6. Connect to Message Broker
7. OK
8. Register
9. Subscribe for messages.
10. Start Publishing/Receiving Messages

Figure 16.Typical service startup procedure

Finally we start the Data Storage service, which as the name implies, will be used to store collected data from sensors. This data will be delivered via the Message Broker. For this reason this service needs to connect to the Message Broker, but how this service knows the IP of the Message Broker; this is where we need the Coordinator.

Data Storage service fetch from the Coordinator the IP of a broker that support a protocol wanted from the service, e.g. MQTT, and then connects to the broker. After connection established with Message Broker each service is responsible to register itself to the Coordinator, to be available to any service wants to communicate with. One last thing that must be done from a service is to inform the Message Broker with the messages want to receive. The procedure described is depicted in figure 16. So far we start the network core services. Now we need to start two additional services in order to start collecting data. The first service is the Monitor service, which provide to end users a web interface from where we can monitor sensor devices and configure them. The starting process of monitor service it's the same as all the network services. Finally we start the gateways which connect our IP network with our heterogeneous wireless sensor network. From that moment onwards we can monitor and publish messages to WSN.

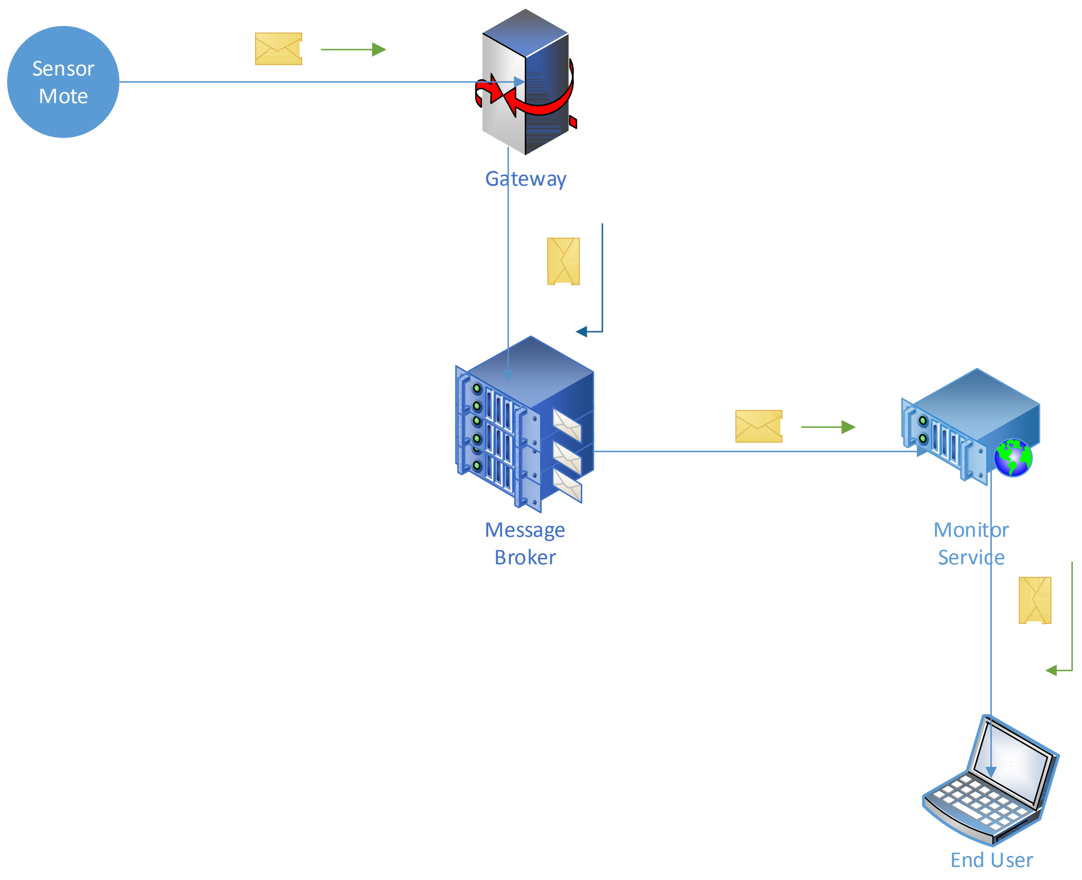


Figure 17. Typical message routing

A typical message route depicted in figure 17 below. A message sent from a sensor mote first received from the gateway, which in turn forward it to Message Broker. When Message Broker receive the message, checking its subscriptions and forward the message to interested parties, which in our demo is Monitor service. Finally, when the monitor services receives the message, show it to the end user.

5.2 Configure system notifications

From the previous demo, we have already setup all the necessary services for our network to operate. In this demo we will start a new service that will receive data from the Message Broker, analyze them, and checkout if the rules for the data is in the permissible limits, and if not triggers a notification to the network, in order for other services that handle notifications to perform, the configured actions. Firstly we will start a service, which handle data from the sensors, and to be more specific, handle accelerometer data. After the basic initialization, the service will fetch from the Data Storage, rules relative to accelerometer data.

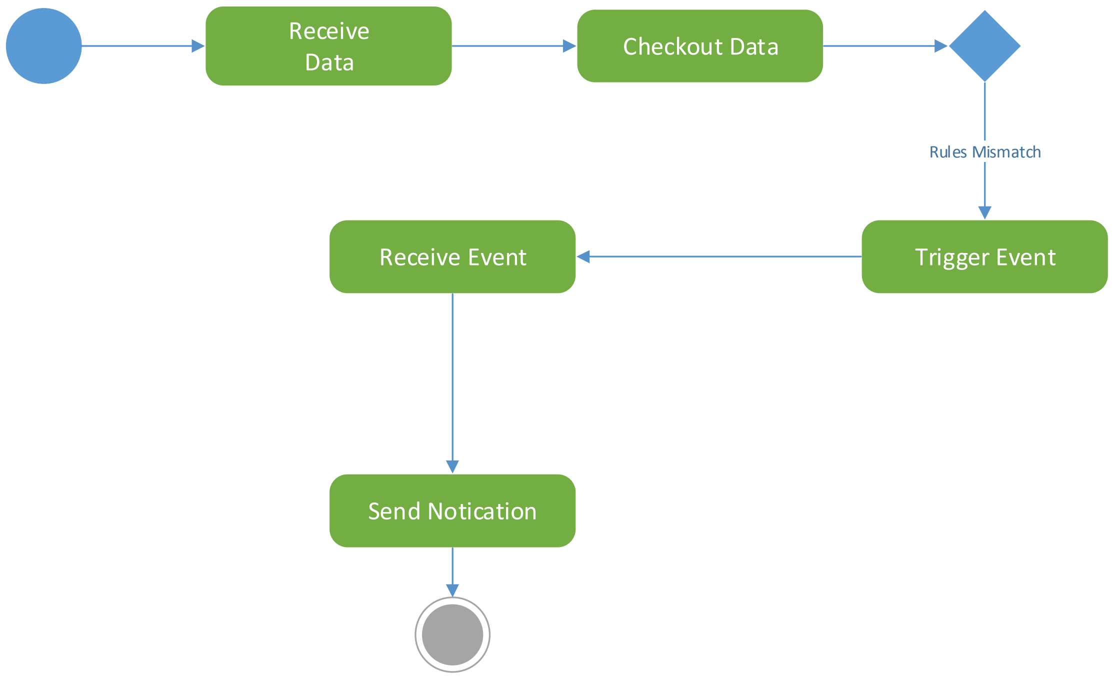


Figure 18. Notification process

Then we will start an additional service that listens for notifications events, and when receive such a notification just performs the action. In our demo the notification is a simple email that informs end users about the rules mismatch. In figure 18, depicted the process for sending a notification.

6. Performance Evaluation

In this section we evaluate the performance of our infrastructure implementation. Firstly, aiming to evaluate time constrained behavior and performance robustness, we measure the round trip delay of a message which is send from WSN until the respective response comes. Furthermore in order to evaluation the resource demands as well as scalability capabilities, we measure CPU and memory usage in the Gateway, which is a critical component in our infrastructure. Additionally, in order to reveal the effect of different technologies both Bluetooth based and IEEE 802.15.4 devices are utilized in conducted experiments.

6.1 Experimental setup

Aiming to offer useful, objective and comprehensive evaluation for our network infrastructure, the following network parameters are taken into consideration:

- Traffic data rate of 5 transmitted messages per second.
- 1 up to 6 concurrently transmitted nodes are considered.
- A data message size of 16 bytes typical for WSN applications.

In order to measurement the round trip delay of a message the following scenario implemented. Firstly we create a network service, which receives the measurement messages and without performing any manipulation, sent them back to the Message Broker. In the WSN, our sensors before send the message to the Gateway save the transmission time. When the Gateway receive the message forward it to Message Broker, which in turn send it to the network service we have created. As mentioned before, our service when receive the message send it back to Message Broker, which forward it again to the Gateway. In this point the Gateway, extract the payload from the message, in order to find the destination sensor mote, for send it. Finally when the sensor mote receive the message, calculate the delay, between the received and transmission message time.

The evaluation undertaken is conducted based on Shimmer platform [21], which offering both Bluetooth and 802.15.4, communication capabilities. Shimmer nodes' software stack is based on the open source TinyOS operating system. The Gateway is a standard x86 PC running Linux operating system.

6.2 Experimental results

Firstly we measure the delay only for devices transmitting messages through Bluetooth, then measure delay only for devices transmitting messages through 802.15.4, finally create a mixed network where half the sensors send messages through Bluetooth and the other half through 802.15.4.

Figure 19, below, the time it takes for a message to send from a device and return to it. As depicted when only Bluetooth devices transmit messages, the mean delay ranges from 120ms up to 150ms, when 6 devices transmit concurrently.

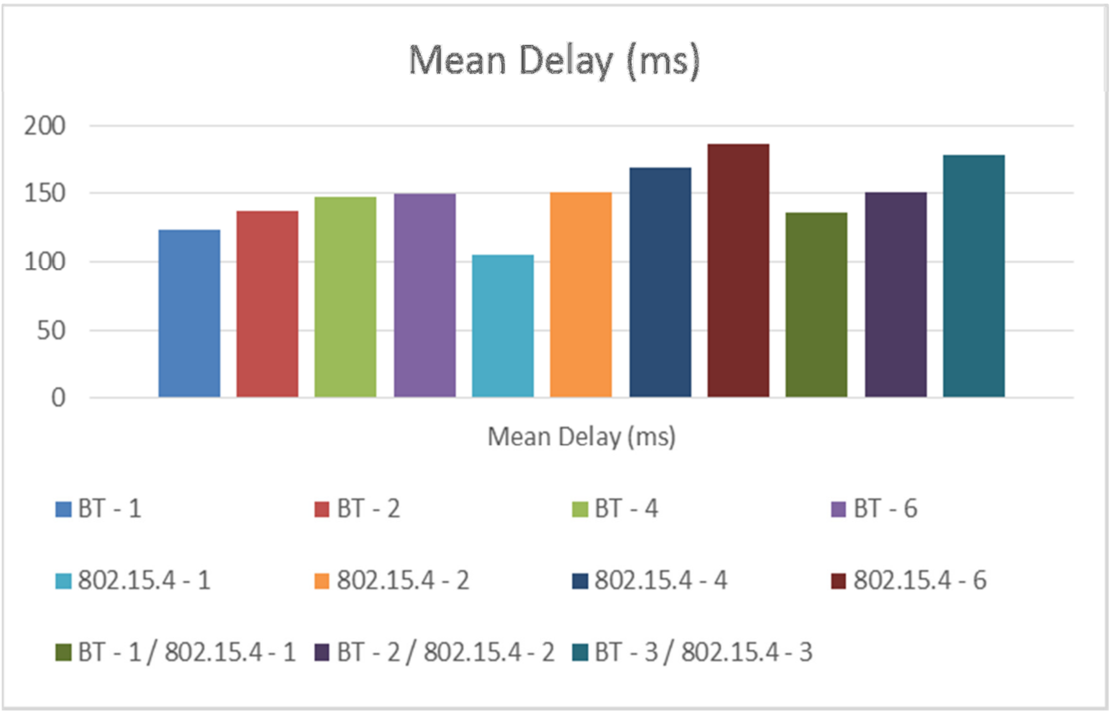


Figure 19. Messages Mean Delay

In the measurements where only 802.15.4 devices transmit messages, the results is identical with Bluetooth, on low traffic rate scenarios, where 2 device transmit concurrent, but when we increase the concurrent transmission devices to 4 and 6, we observer a slight increase, where the mean delay reach up to 185ms, when 6 devices transmit concurrently. Finally, in the mixed networking scenario, the results are almost identical, with the corresponding, when we have only Bluetooth or 802.15.4

transmitters. Ranges from 130ms up to ~180ms when 6 devices transmit concurrently. Respective measurements advocate the use of the proposed infrastructure even in demanding WSN applications. Additional, it exhibits considerable stability with respect to heterogeneous technologies and varying number of sensors.

Figure 20, below, depicts, depicts, the CPU usage of the Gateway, when sensors transmitting data. As shown the usage ranges mainly from 2% up 5% for all scenario, with some small spikes, who reach up to 14%, originates when Bluetooth devices transmit data.

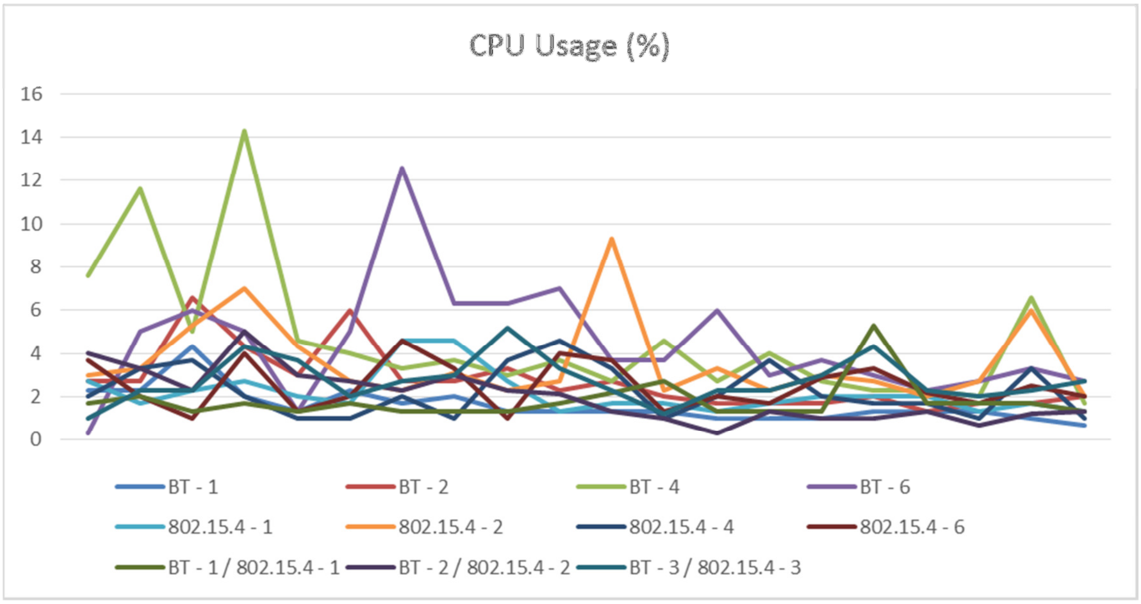


Figure 20. CPU Usage

Finally figure 21, below, depicts the memory usage of the Gateway, when sensors transmitting data. As shown the usage ranges from 60MB on low traffic scenarios, up to 80MB when high traffic scenarios imposed. Both memory and CPU utilization measurements indicate quite low respective demands which can be accommodated by nowadays embedded systems such as Raspberry PI or Intel's Edison based platforms.

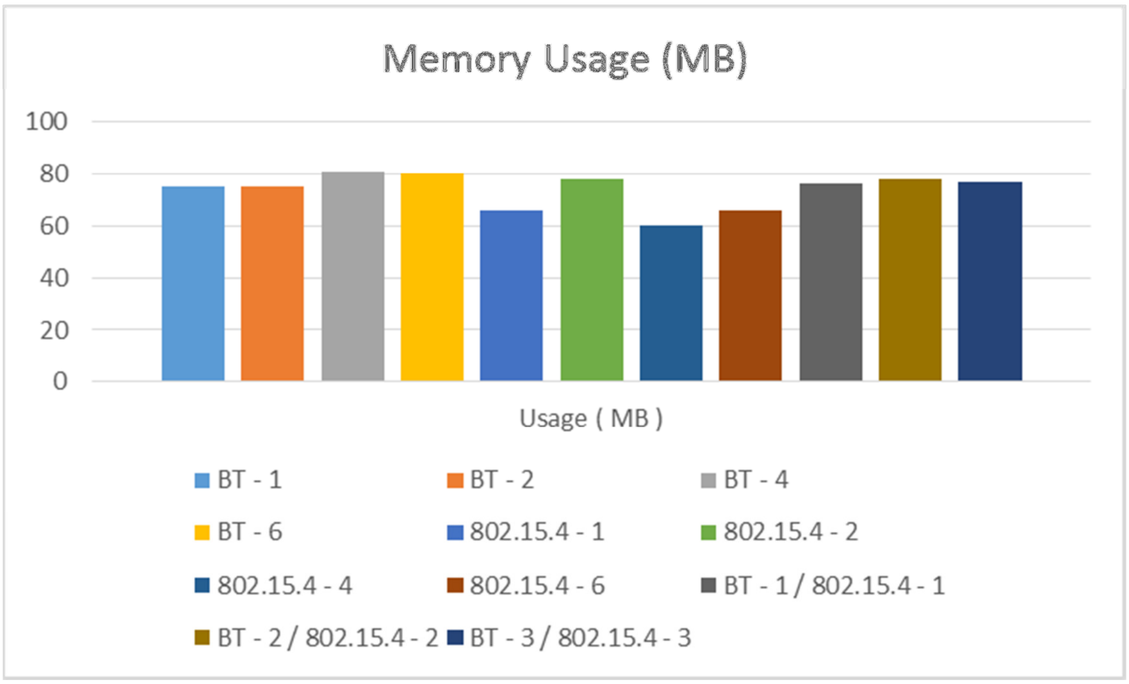


Figure 21. Memory Usage

7. Conclusions

Over the last few years Cyber Physical Systems appear as the most prominent research area able to unite the physical and cyber domains in the context of a wide range of different and diverse applications scenarios. However, in order for respective solutions to be truly useful, practical and widely utilized there is a critical need for holistic communication architectures tackling heterogeneity while exhibiting high degree of flexibility and configurability. Driven by this requirement this paper proposes a comprehensive end-to-end architecture based on message passing communication paradigm. In that respect implemented architecture includes wireless sensor networks as the last mile of a complete CPS systems but also considers backend aspects such as network coordination, storage facilities and end user interfaces. All these aspects comprise a complete, efficient and versatile novel architecture effectively addressing aforementioned requirements. Additionally, implemented solutions is evaluated considering different WSN communication technologies and workload patterns exhibiting in all cases time constrained and robust behavior. Furthermore, resource requirements are measured revealing conservative respective demands easily met by COTS nowadays embedded platforms. Finally we believe that such an effort can also serve as the foundation for further extensions and enhancement increasing the added value of such solutions in different application domains.

Bibliography

1. Fei Hu, Cyber Physical Systems - Integrated Computing and Engineering Design. CRC Press, 2014.
2. Bluetooth. Specifications of the Bluetooth Systems (SIG). 2001 : s.n. Version 1.1.
3. IEEE. 802.15.4 Specification. 802.15.4 Specification. [Online] IEEE. http://standards.ieee.org/about/get/802/802.15.html.
4. ZigBee Specification. 2008.
5. Linux. [Online] http://www.linux.org.
6. Levis Philip and Gay David. TinyOS Programming. 2009.
7. AMQP Specification v1.0. 2011.
8. STOMP Specifications. 2015. v1.2.
9. Banks Andrew and Gupta Rahul. MQTT Specifications v3.1.1. s.l. : OASIS, 2014.
10. Truong, Andy Stanford-Clark & Hong Linh. MQTT for Sensor Networks (MQTT-SN), Protocol Specification v1.2. 2013.
11. Michael McGrath, Cliodhan Scanail. Sensor Technologies. s.l. : Apress, 2013.
12. IBM. [Online] March 2013. https://goo.gl/VV1LNw.
13. Mosquitto. Mosquitto. [Online] September 2015. http://mosquitto.org/.
14. Bruce Snyder, Dejan Bosanac, and Rob Davies. ActiveMQ in Action. s.l. : Manning Publications, 2011.
15. KahaDB. ActiveMQ Kaha. [Online] 2015. http://activemq.apache.org/kahadb.html.
16. Newman Sam. Building Microservices. s.l. : O'Reilly, 2015.
17. Haloi, Saura. Apache Zookeeper Essentials. s.l. : Packt Publishing, 2015.
18. Kotek Jan. mapDB. [Online] 2014. http://www.mapdb.org/.
19. MongoDB. [Online] https://www.mongodb.org/.
20. JSON. [Online] http://www.json.org/.
21. Shimmer. [Online] http://www.shimmer-research.com/.
22. Gregor Hohpe. Enterprise Integration Patterns Design Building and Deploying Messaging Solutions. s.l. : Addison-Wesley, 2004.
23. Zookeeper: Wait-free coordination for Internet-scale system. P Hunt, M. Konar, F. Junqueira, B. Reed. s.l. : USENIXATC Proceedings of the 2010 USENIX conference on USENIX annual technical conference, 2010.
24. MQTT-S – A Publish/Subscribe Protocol For Wireless Sensor Networks. U Hunkeler, H. Truong, A. Stanford-Clark. 2008.
25. Enabling Publish/Subscribe Services in Sensor Networks. Duc A. Tran, Linh H. Truong. 2010.
26. Wireless Sensor Network Architecture for Smart Building. Adama, V. 2008.
27. Communication based on MQTT Protocol. Milojic, Milan. 2014.
28. A Comparative study of Wireless Protocols: Bluetooth, UWB, ZigBee and Wifi. J. Lee, Y. Su, Ch. Shen. 2006.
29. Nane Kratzke, About Microservices, Containers and their Underestimated Impact on Network Performance, CLOUD COMPUTING 2015 : The Sixth International Conference on Cloud Computing, GRIDs, and Virtualization
30. Aad Versteden, Erika Pauwels, Agis Papantoniou. An Ecosystem of User-facing Microservices supported by Semantic Models. Published at the 5th International USEWOD Workshop.
31. Orestis Evangelatos, Kasun Samarasinghe, Jose Rolim. Evaluating Design Approaches For Smart Building Systems.

Abbreviations

ACL – Access Controls Lists

AMQP – Advanced Message Queuing Protocol

API – Application Programming Interface

BSON – Binary JSON

CPS – Cyber Physical System

CSS – Cascading Style Sheets

DB – Database

DMZ – Demilitarized Zone

FCC – Federal Communications Commission

FHSS – Frequency Hopping Spread Spectrum

GPS – Global Positioning System

GSFK – Gaussian Frequency Shift Keying

GW – Gateway

HCI – Host Controller Interface

HTML – HyperText Markup Language

IoT – Internet of Things

IP – Internet Protocol

ISM – Industrial Scientific Medicine

JAAS – Java Authentication Authorization Service

JMS – Java Message Service

JSON – JavaScript Object Notation

LMP – Link Management Protocol

L2CAP – Logical Link Control and Adaptation Protocol

MEMS – Microelectromechanical Systems

MOM – Message Oriented Middleware

MQ – Message Queue

MQTT – MQ Telemetry Passport

MQTT-SN – MQ Telemetry Passport for Sensor Networks

M2M – Machine to Machine

OS – Operating System

O-QPSK – Offset Quadrature Phase Shift Keying

OSI – Open Systems Interconnection

PC – Personal Computer

PDA – Personal Digital Assistance

POSIX – Portable Operating System Interface

QoS – Quality of Service

QPSK – Quadrature Phase Shift Keying

RAM – Random-Access Memory

RF – Radio Frequency

RFCOMM – Radio Frequency Communications

RPC – Remote Procedure Call

RSMB – Real Small Message Broker

SASL – Simple Authentication and Security Layer

SIG – Bluetooth Special Interest Group

SDP – Service Discovery Protocol

STOMP – Streaming Text Oriented Message Protocol

TCP – Transmission Control Protocol

TDD – Time Division Multiplexing

TLS – Transport Layer Security

UDP – User Datagram Protocol

URI – Uniform Resource Identifier

URL – Uniform Resource Locator

WSN – Wireless Sensor Network

XML – Extensible Markup Language