Universidade de Aveiro
2021

João Miguel
Soares Ferreira

Desenvolvimento de um Sistema de Gestão Técnica Centralizado

Development of a Centralized Building Management System

Universidade de Aveiro
2021

João Miguel
Soares Ferreira

Desenvolvimento de um Sistema de Gestão Técnica Centralizado

Development of a Centralized Building Management System

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Paulo Pedreiras, Professor Auxiliar do da Universidade de Aveiro, e do Doutor Pedro Fonseca, Professor Auxiliar do Departamento de Eletrónica Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president	Prof. Doutor Telmo Reis Cunha Professor Associado da Universidade de Aveiro
vogais / examiners committee	Prof. Doutor António Luis Ferreira Marques Professor Adjunto do Instituto Superior de Engenharia de Coimbra
	Prof. Doutor Paulo Bacelar Reis Pedreiras Professor Auxiliar da Universidade de Aveiro

agradecimentos / **acknowledgements**

Em primeiro lugar agradeço ao meu orientador, Professor Paulo Pedreiras, assim como ao co-orientador, Professor Pedro Fonseca, pela oportunidade dada para desenvolver este trabalho, assim como as suas orientações e sugestões que muito contribuíram para a realização desta dissertação.
Agradeço à empresa Diferencial, e em particular ao Engenheiro António Tavares, pela disponibilidade e pelos conhecimentos da área partilhados comigo, bem como ao Instituto de Telecomunicações por me acolher e pelas condições que me proporcionou.
À minha família, em especial aos meus pais, à minha irmã e aos meus avós, agradeço todo o apoio dado durante estes anos de estudo.
Deixo um agradecimento especial à minha namorada, por todo o apoio moral e motivação que me guiaram até ao final deste percurso.
Aos meus amigos, um obrigado por todas a discussões e diversões proporcionadas, que tornaram os dias mais leves.

Palavras Chave Internet of Things (IoT), Automação Industrial, Automação de Edifícios, Sistema de Gestão Centralizado

Resumo

Um Sistema de Gestão Centralizado tem por objetivo aumentar a comodidade e conforto dos utilizadores de um edifício, ao mesmo tempo que tenta reduzir os consumos energéticos do mesmo. Para isso, torna-se necessário integrar sensores e atuadores para controlar e recolher informação acerca dos processos físicos existentes. Nestes processos estão incluídos a iluminação e temperatura de, por exemplo, uma sala, ou até controlo de acesso. Esta informação, após processamento, permite, de uma maneira mais inteligente e eficiente, controlar os sistemas eletrónicos e mecânicos de um edifício, tais como os sistemas de AVAC ou iluminação, tentando, simultaneamente, diminuir gastos energéticos. O aparecimento do IoT, tornou possível o aumento do número de dispositivos de baixo nível nestes sistemas, graças à redução de custo e aumento de performance e conectividade que estes têm sofrido. Para melhor usufruir deste paradigma, é necessário um sistema moderno, com capacidade de conexão multi-protocolo e ferramentas para processamento e apresentação de informação. Neste sentido, fez-se um estudo das tecnologias mais relevantes da área da automação industrial e de edifícios, de modo a definir uma arquitetura moderna compatível com IoT e a escolher as plataformas de software que a constituem. InfluxDB, EdgeX Foundry e Node-Red foram as tecnologias escolhidas para a base de dados, gateway e dashboard, respetivamente, por serem as que mais se aproximaram dos requisitos definidos. Assim, foi desenvolvido um demonstrador que permitiu verificar o funcionamento de um sistema com a utilização destas tecnologias, assim como avaliar a performance da plataforma EdgeX em termos de jitter e latência. Verificou-se a partir dos resultados obtidos, que embora versátil e completa, esta plataforma ficou aquém do que se pertendia, tanto para aplicações real-time, como para as que necessitem de uma taxa de leitura de sensores elevada.

Keywords Internet of Things (IoT), Industrial Automation, Building Automation, Building Management System

Abstract

A building management system has user confort and comodity, as well as reduction of energy consumption, as its main goals. To accomplish this, it is necessary to integrate sensors and actuators as to control and retrieve information about the physical processes of a building. These processes include control over ilumination and temperature of a room, and even access control. The information, after processed, allows a more inteligent and efficient way of controling eletronic and mechanical systems of a building, such as HVAC and ilumination, while also trying to reduce energy expenditure. The emergence of IoT allowed to increment the number of low level devices on these systems, thanks to their cost reduction, increased performance and improved connectivity. To better make use of the new paradigm, it is required a modern system with multi-protocol capabilities, as well as tools for data processing and presentation. Therefore, the most relevant industrial and building automation technologies were studied, as to define a modern, IoT compatible, architecture and choose its constituting software platforms. InfluxDB, EdgeX Foundry and Node-Red were the selected technologies for the database, gateway and dashboard, respectively, as they closely align with the requirements set. This way, a demonstrator was developed in order to assess a systems's operation, using these technologies, as well as to evaluate EdgeX's performance for jitter and latency. From the obtained results, it was verified that, although versatile and complete, this platform underperforms for real-time applications and high reading rate workloads.

Contents

List of Figures

List of Tables

Glossary

API	Application Programming Interface
CLI	Command Line Interface
DALI	Digital Addressable Lighting Interface
DBMS	Database Managemen System
ERP	Enterprise Resource Planning
IoT	Internet of Things
MES	Manufacturing Execution System
PLC	Programmable Logic Controller
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
IFTTT	If This Than That
IP	Internet Protocol
IT	Information Technologies
OPC-UA	OLE for Process Control - Unified Architecture
HMI	Human-Machine Interface
OPC	OLE for Process Control
REST	Representational State Transfer
SDK	Software Development Kit
SQL	Structured Query Language
DDC	Direct Digital Controller
IA	Industrial Automation
BAS	Building Automation System
SDK	Software Developer Kit
UI	User Interface

Chapter 1

Introduction

Building Automation System (BAS), or Building Management Systems, have been implemented for quite some time now. They initially appeared as a way of managing HVAC systems, in order to optimize the resources consumed by those systems. They have since evolved to integrate those mechanical infrastructures with digitalised equipment. It now needs to manage a whole range of services, from access control and security to illumination control, be it through light fixtures or motorized blinds. These elements have, as a target, energy efficiency optimization, while also improving the comfort of the users, which can positively impact the productivity of a Company.

At the same time, recent technologies born in the Information Technologies (IT) world, have permeated onto the Industrial and Building automation field. The most impactful of the recent years is the Internet of Things (IoT) paradigm. This brought new, low cost electronics with high degrees of connectivity to sensors and actuators. As the cost lowered, a higher amount of these devices could be installed, giving way to what is called of Big Data. A high number of these devices can generate enormous amounts of data, and thus better inform a centralized decision making element. This data, if correctly analysed, allows insight on building occupation allowing the possibility of directing its resources to where they are most needed. It also allows exploring sensors' behaviour, as to increase the predictability of problems in the system due to a component malfunction. Functionality like this can alert the system manager to future problems, and plan an intervention ahead of time, reducing the possible costs by eliminating catastrophic problems and reducing the down time of the system.

The goal of this dissertation is to review recent technologies targeted at BAS, as well as to define an architecture for it. This system needs to have the possibility of integrating a wide range of equipment, be it individual sensors with recent communication protocols or legacy machines. In order to achieve this, a review of a set of key technologies is made, along with new software frameworks, specifically developed for Industrial and Building Automation.

1.1 How the document is structured

The present document is structured in the following way:

- Chapter 2 presents a review of a set of key technologies for Industrial and Building automation. These include the industrial automation architecture and some relevant communication protocols, namely Modbus, EtherCat, DALI and BACnet. An introduction to Containerization is also made, with Docker being the most relevant implementation. Relational, NoSQL and time series databases are presented too. The chapter closes with four different gateway software offerings and a brief overview of REST APIs.
- Chapter 3 reviews current and previous generation architectures for BAS. The architecures discussed were found on literature and are divided into two categories. The first, concerns the architecture based on the Industrial Automation (IA) hierarchical structure. The second encompasses solutions directly targeting IoT based BAS.
- Chapter 4 introduces the possible use cases of the systems, followed by the definition of its requirements. The system architecture is then defined, and the requirements for each element are drawn.
- Chapter 5 dicusses an implementation of the system. First, the demonstrator is defined and described. Then, the technologies for the gateway and the database were selected through a feature set comparison. The databases are further evaluated through some performance comparisons found on literature. These technologies are then described more thoroughly and their configuration and setup is shown.
- Chapter 6 shows the tests performed in order to evaluate the system. These mainly focus the gateway system, EdgeX Foundry. Firstly, the tests' setups are introduced. Then, the metrics evaluated are shown and discussed. Afterwards, the results are presented and discussed.

1.2 Contributions

This dissertation contributed with the research and definition of an architecture for BAS, as well as gathering knowledge and evaluating different technological offerings for these systems.

Chapter 2

Background

This chapter introuces background information required for a better understanding of the work developed during this dissertation. It begins by presenting the common structure for organizing a factory floor, with special focus on the network technologies used in this type of environments. Next, some protocols specially suited for Industrial and Building Automation are introduced, namely EtherCAT, BACnet, Modbus and DALI. Lastly presented is an overview of some software solutions for Containerization, Databases and Gateways, as well as a brief discussion on what is their functionality.

2.1 Factory hierarchy and its networks

To better understand how to incorporate a solution for the monitoring and management of manufacturing processes, or to interact with low-level devices so to acquire status information of sensors and actuators, it is important to review how the elements on a manufacturing process are structured.

Industrial automation continues to evolve and as such it requires multiple IT-systems to cope with the corresponding increase in the manufacturing systems complexity. So, to approach this problem in a simpler manner, different applications are separated and categorized in a functional, hierarchical manner. This takes the shape of what is known as the automation pyramid [1], pictured in Figure 2.1. This representation of the manufacturing processes is defined in different standards, being [2] the one most commonly refered to. This standard defines five layers, each one composed of groups of devices with specific functionality on the manufacturing process. Communication between the different layers is not normalized, as requirements and constrains are different. The hierarchical structure follows this categorization. So, the layers grow from the equipment that directly interacts and transforms the raw materials into products, which have stricter requirements in terms of critical, and timely information, to systems that oversee all the running processes. These sit at the top of the pyramid and have more relaxed requirements comparing with the lower levels, but interact with a higher volume of data.

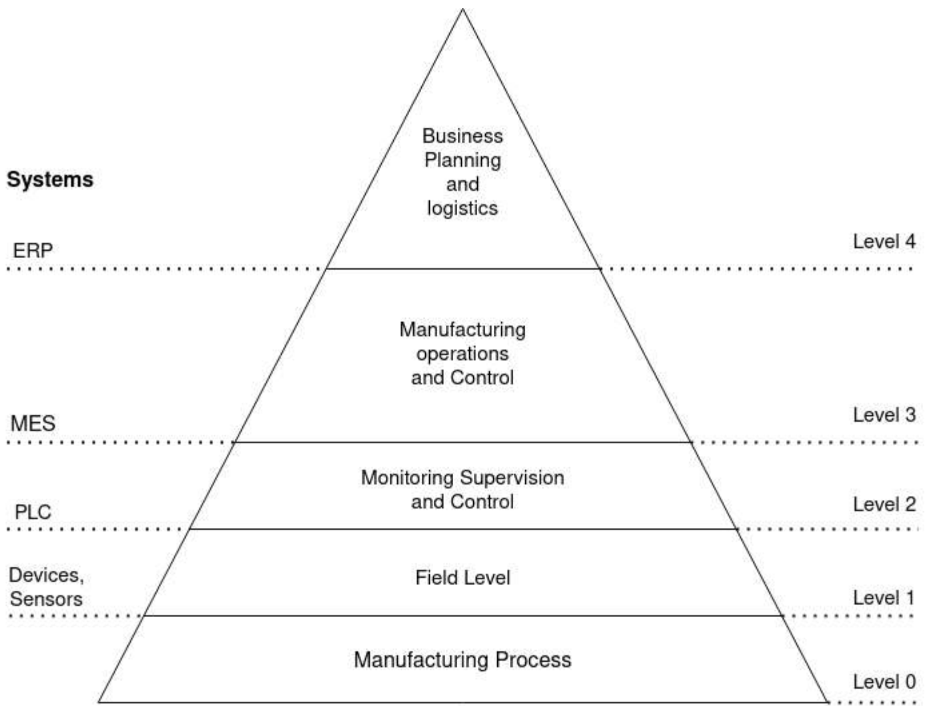

Systems
ERP
MES
PLC
Devices, Sensors
Business Planning and logistics
Manufacturing operations and Control
Monitoring Supervision and Control
Field Level
Manufacturing Process
Level 4
Level 3
Level 2
Level 1
Level 0

Figure 2.1: Industrial Automation Pyramid. Adapted from [1]

2.1.1 Manufacturing Process and Field Level

The first two levels of the automation pyramid are the manufacturing process and the Field Level. The first concerns the actual production, the machines and systems that transform materials in new products with added value.

The Field Level is where actuators act on the moving parts of the machine and sensors capture their valuable state. These elements are connected to higher levels frequently through fieldbuses like Modbus, Profibus and EtherCAT. Historically this layer was filled with proprietary protocols, many using robust low bitrate transport mechanisms such as EIA-485/422. However, recently, IT technology has been infiltrating. EtherCAT and Profinet are two prominent protocols based on Ethernet, bringing its versatility to the fieldbus layer. But still, limitations exist. Both of those protocols make proprietary modifications to the Ethernet standard to allow high performance real-time connections to critical machinery. Although they provide great utility for the applications they are destined, they do not have good interoperability capabilities.

2.1.2 Monitoring Supervision and Control

This layer includes systems that make decisions based on information from the Field Level devices, as well as software solutions that supervise and store information from the various manufacturing cells in a factory.

Here, sensors and actuators connect to Programmable Logic Controller (PLC) and Remote Terminal Unit (RTU) and close the control loop of the different factory floor machines. PLCs have limited processing capabilities but are very versatile, with modular I/O cards and dedicated add-ons for communication. Furthermore, their robustness and ease of development make them appellative to system integrators. Alternatives include the RTU, which have more restricted applications, as usually they come pre-programmed for a specific task, as well as microcontroller-based solutions.

Supervisory Control and Data Acquisition (SCADA) systems, also categorized in this level, retrieve and store information from the various machines, normaly displayed in a Human-Machine Interface (HMI). These elements are connected through Ethernet based protocols, which, depending on the requirements, can be Fieldbuses like the ones described before, or Internet Protocol (IP) based solutions like OLE for Process Control (OPC). The communication with the next level, the Manufacturing Operations and Control is typically made through IP technologies.

2.1.3 Manufacturing Operations and Control and Business Planning and Logistics

The top layers of the Automation pyramid focus more on the logistic control of the manufacturing process. Functions such as managing work sequences, parts lists and production receipts are delegated to solutions found in the Manufacturing Operations and Control level [1]. The most common solution used for this job is a Manufacturing Execution System (MES). It works in a day-to-day or a shift basis and coordinates all the manufacturing procedures needed to develop the target product.

For Business Planning and Logistics, Enterprise Resource Planning (ERP) tools are used to define long-term production utilization as well as energy demand for the entire factory [1]. Both of these levels use more IT based solutions, as the data flowing is of a bigger volume, and there is no need for solutions with strict time delivery, nor critical control loops.

2.2 Industrial and Building Automation protocols

As mentioned in Section 2.1, due to the requirements of the lower levels of the automation pyramid in terms of performance, Ethernet based protocols are quite different from the ones used in the IT domain. As such, it is relevant to give information about some protocols used on the industrial automation domain, represented by Modbus and EtherCAT and the building automation domain, represented by Digital Addressable Lighting Interface (DALI) and BACnet. This section serves as a short introduction to these technologies.

2.2.1 Modbus

Modbus is a communication protocol initially developed by Modicon and is now an open standard maintained by MODICON-ISA. The original application of this protocol was for PLCs to communicate between each other, but it can now be found connecting low level devices, such as sensors to controllers.

The Modbus protocol is defined to be implemented on top of various asynchronous serial transmissions media, the most notable being EIA-232 and EIA-485 and also over TCP/IP Ethernet. It defines an application layer with high parallelism with PLC memory structure.

The data space is divided into Coils, Discrete Inputs, Input Registers and Holding Registers, with the object type of the first two being a single-bit value and the others a 16-bit word.

The communication models are Master-Slave for serial connections and Server-Client for TCP/IP. As an example, in a network with a machine controlled by a PLC and an HMI to display the machine information, the PLC would be a server/slave, responding to requests published by the client/master, in this case the HMI.

2.2.2 EtherCAT

Ethernet for Control Automation Technology (EtherCAT) is an industrial communication protocol targeted at field level connections. Developed by Beckhoff, it is a real-time isochronous technology specially suited to applications requiring short-cycle times and low communication jitter [3]. Currently it is managed by the EtherCAT Technology Group (ETG) and shared as an open standard.

The communication model is based on a Master-Slave relationship, with the Master being the only element in the network capable of initiating a data stream to which the slave devices respond. This prevents unpredictable delays and thus increases the deterministic capabilities of the protocol. Although it is based on Ethernet physical medium, the slave nodes are arranged to represent an open ring. A telegram produced by the Master, runs through every slave sequentially. These respond to the request by adding the required data to the telegram as it passes. The last element of the network detects an open port and sends back the telegram, now fully populated with information from slave devices. Even though this mechanism greatly increases the performance of the network, it comes at the cost of custom Ethernet controllers on the slaves, reducing the interoperability with other industrial protocols. To address this, EtherCAT has a mailbox service which effectively allows packets from other technologies, like standard Ethernet packets, to circulate the network, at the cost of lower bandwidth, so to not compromise the network's performance.

2.2.3 DALI

DALI is a Building Automation protocol, trademark of the DiiA (Digital Illumination Interface Alliance) and developed specifically for illumination control. It is the successor of 1-10V/0-10V lightning control systems used for light switching and dimming and adds functionalities such as scene definition and control and fault detection. This protocol is defined in the IEC 62386 standard.

A DALI network is compose of an application controller, the local network controller, control devices, for example LED drivers or electrical ballasts, as well as input gear such as switches and presence sensors. Data and power flow from the application controller to the other network devices through the same electrical wire pair. This feature and the number of possible topologies (line, star, ring, tree and mixed) allow for a simple and versatile system installation. The limitations that come from this are a quite slow datarate on the bus (1200bps) and a maximum of 64 control gear units for each application controller drop [4].

The protocol defines a master-slave relationship between the application controller and the other network devices. As such, the only element capable of initiating data exchange is the application controller. Although the standard also defines multi-master mode of operation, only a small number of devices actually implement it.

2.2.4 BACnet

Building Automation and Control network (BACnet) is a protocol developed by the American Society of Heating and Air-Conditioning Engineers (ASHRAE) for building automation applications, specially suited for comunication between the automation layer and the management layer [4].

It is an object-oriented protocol, with objects describing devices in the network. Devices in the network always have at least one object associated to them (Device Object), but others can also be added, from a pool of 49 standard objects. Object properties can be read, written, or accessed through the corresponding services. Information exchanged on the network is made through services. Through them, operations over devices on the network, such as data read and write, are made. Although this is the base of a Server-Client relationship, devices do not have fixed roles, allowing direct communication between any node.

The standard defines the possibility for data transport over more than one data link and physical layer protocols. BACnet can be transported over IP Ethernet (BACnet/IP), ARCNET, EIA-482 Master-Slave/Token-Passing (MS-TP), EIA-232 for point-to-point connections and LonTalk, an Echelon proprietary protocol, defined for layers 1 and 2 of the OSI model.

2.2.5 MQTT

MQTT [5] is a publish/subscribe messaging transport protocol, using TCP/IP, that strives to be simple and easy to implement while being lightweight enough to be integrated into constrained applications. It is royalty-free since 2010 and an OASIS officially approved standard since 2014 [6]. Although multi-purpose, it is notably well suited for machine-to-machine applications and IoT use cases, as client libraries have a small code footprint, thus capable of being integrated into constrained devices, and high bandwidth efficiency.

As mentioned above, it is based on a publish/subscribe communication model. This tries to separate the traditional direct connection found in a client-server model. In MQTT's case, a client is connected to a broker and publishes its information with a specific topic attached. The broker, upon reception of the message, sends it to the clients which had previously subscribed to that same topic. This element, the broker, is the entity controlling the various connections in the network. It registers which clients are connected, and to which topics they publish or are subscribed to. It also receives all messages and filters them, determining the path they need to follow. The topic is a subject-based filtering mechanism. It identifies a message through the content of its information and not through the sender or destination identification. Effectively, the clients do not need to be aware of the existence of one another to establish a connection. The advantage of this mechanism is clear to see in terms of scalability, especially when compared to a client-server connection. Operations in the broker can be highly parallelized and messages are processed in an event-driven way.

2.3 Containerization

A container is a lightweight virtualization technology used to build self-contained applications for great portability. They are lightweight in comparison to traditional hypervisors, as they do not require neither hardware virtualization nor device driver virtualization [7]. In practice, this means that a system that is limited to a low number of virtual machines, due to their inherent higher resource cost, can run a much higher number of containers.

Applications are containerized, meaning they are built and packaged together with their respective configuration files, libraries and dependencies. They run on the same operating system kernel as the underlying one. More simply put, if an application is to be run on Linux the platform running the container also needs to be a Linux based system. This abstraction also implies that containers work as isolated processes at the operating system level.

2.3.1 Docker

Docker [8] is an open source containerization platform that has been increasing rapidly its user base due to its ease of use and being a quicker option to build, deploy and manage containers [9]. It provides the tools needed to package, instantiate and manage containers in a simple manner.

Docker compose

Docker compose is a tool that allows the deployment and management of a collection of containers. It has automatic image fetching, from Docker hub, and instantiation, all configured and described in a YAML (YAML ain't a Markup Language) file. It also allows network configuration and file system configuration (volumes).

2.4 Databases

A database is an organized collection of data, usually arranged for high speed search and retrieval. To organize data, Database Managemen System (DBMS) are used, each one designed for a certain specific requirement set. They vary in structure and architecture, with the main constraints between them being measured in terms of data insertion, data retrieval and search times as well as space occupation and transaction reliability.

A high number of database management systems exist in the market, being generally distinguished in two big groups: relational databases and NoSQL, schema-less databases. The first ones are the traditional and more mature solutions, with MySQL being one of the most well-known representatives. These databases are structured in a tabular manner, and the data inserted must follow a predefined table schema. To deal with the information stored, or to be stored, they use a Structured Query Language (SQL), with this acronym being their moniker. "Not only SQL" (NoSQL) databases started to be used as an alternative to relational ones, specially for IoT applications. The rigid data structure of relational DBMSs can, in applications with a high volume of unstructured data, hinder their implementation [10]. Narrowing the information to be stored into a collection of time series vectors, a new class of database can be drawn: time series databases. These use timestamps as the primary index of the data storage and have specific functionalities for its management.

Three databases, each representing one of these groups, will be discussed further ahead in this document. For now, a short presentation of each one is made.

2.4.1 MySQL

MySQL [11] is an open source database management system project, currently owned by Oracle. It has a GPLv2 licensing but the company also provides commercial licenses with an increased number of features. This DBMS has a relational data model, thus needing a strict schema to structure all the stored data. Data transactions are made through SQL, with small differences from the standard implementation, and are ACID compliant.

ACID properties ensure that transactions do not compromise data consistency and correctness [12]. There are four properties, Atomicity, Consistency, Isolation and Durability. Atomicity states that a transaction is a process that is indivisible. If a step fails, then the transaction is discarded. Consistency means that if a transaction fails the database needs to be roledback to a previous state, so that data is not corrupted and consistency is maintained. Isolation implies that transactions are independent and occur without interferences. Finally, Durability is a property that states that data transfered after a transaction is stored even if a system failure occurs.

2.4.2 MongoDB

MongoDB [13] is a relatively recent project, with its inception dating 2007, governed by the Mongo developed SSPL (Server Side Public License). As with MySQL, it is open source with possibility of commercial licensing. Using a document data store model, it favors ease of development and scaling. The document structure used is a custom binary JSON-like file (BSON), and CRUD (Create, Read, Update, Delete) operations on the data are made through a custom query language. A MongoDB server can be distributed by various machines through sharding. Sharding is a database architecture pattern consisting on the division of a table in to several tables, known as partitions [14]. These partitions can then be distributed through different machines. This way, it is possible to have a single table distributed through various machines, simplifying the process of horizontal scaling. Horizontal scaling is the capability of increasing a database performance and storage capacity through the addition of more compute nodes, instead of adding more resources, such as RAM or storage capacity, to a single system. MongoDB also supports high availability through a master-slave model.

2.4.3 InfluxDB

InfluxDB [15] is the most recent technology out of the DBMS presented in this document, with its initial release in 2013. It is a purposely built time-series database, with an open-source MIT license and optional commercial support with extended features. It has a schema-less data model, with emphasis on data typical of sensor networks, with the main table relationship being a measurement and its corresponding timestamp. With this, and the timestamp begin a dedicated table index, the performance of the DBMS can be greater than traditional technologies. Apart from this, retention policies allow for an efficient use of disk space, allowing Influx to be programmed to eliminate older, not needed, data. As with the others DBMS it offers cluster configuration for data replication by using more than one InfluxDB instance. Data transactions are achieved through a SQL-like language called InfluxQL, accessed through an HTTP Application Programming Interface (API).

2.5 Gateways

On an Industrial environment, implementing a solution incorporating hardware from different vendors, be it sensors or more intelligent devices like PLCs, can be quite challenging. A high number of vendors exist, and many have proprietary communication protocols, being difficult to integrate them directly in a system. A gateway, in the context of IoT, can be considered a platform, be it hardware, software or a mix of both, that directly tries to solve the problem of heterogeneity between the different existing protocols in a factory.

In this document, the concept of gateway is presented as an element that has the capability of interacting with different networks and can translate the data from their nodes to a unified and normalized notation. The reverse also applies, meaning that the endpoint of the gateway also needs to have the features to transport data to the nodes.

Next, four different software gateway frameworks will be presented, namely Eclipse Kura, EdgeX Foundry, macchina.io and Node-Red.

2.5.1 Eclipse Kura

Eclipse Kura [16] is an Eclipse IoT project, which provides a platform to build IoT Gateways based on Java and OSGi containers. It allows remote management of these systems and offers APIs for the creation of particular IoT solutions. It runs on top of a Java Virtual Machine and uses OSGi to make it easy to write reusable software building blocks. The framework is distributed in various ways: as a Docker container, as an installer for various hardware platforms (ARM, x86) and as an Eclipse project. It is released under the EPL-1.0 license.

This framework was initially developed by Eurotech and is now open-source and part of the Eclipse Foundation. This company is still the main contributor to the project, but others, such as Intel and Red Hat, have contributed as well. Eurotech also has a commercial stack based on the software presented here.

At the moment Kura has drivers for OLE for Process Control - Unified Architecture (OPC-UA), Siemens S7 PLC protocol and GPIO. While it is possible to develop other drivers, a framework or guidance for implementing one with the appropriate interface is missing. It is also important to indicate that documentation on this topic is poor.

2.5.2 EdgeX Foundry

EdgeX Foundry [17] is an open source project initiated by Dell with the objective of providing a vendor-neutral, open framework for IoT edge computing. It targets industrial IoT and features loosely coupled microservices and platform-independence, and can accommodate both IP and non-IP based communications. It has the advantage of being modular thanks to the microservice based architecture, meaning the system can be customized to the specific problem at hand and thus have a smaller footprint. The standard way to run EdgeX (what is called user approach) is through Docker containers, with docker-compose to start and manage the various micro services. Other deployment schemes, such as Canonical's snaps, are available.

The project was originally created by Dell in 2015 and made available open-source in 2017. The company is still the leading contributor but the community surrounding the project is growing, with various companies having already developed solutions using this framework, as well as tools to extend it [5]. It is a Linux Foundation project since 2017.

At the moment EdgeX has device services for Modbus, SNMP and MQTT with OPC-UA, BACnet and CAN support being added in the future (these are being developed by IOTech). As for the development of Device Services, the project has a Software Developer Kit (SDK) (two versions, one in Go and other in C) available to help the deployment of custom solutions. In addition, the documentation for this topic is detailed, referencing the functionality a device service needs to have, in order to communicate with the rest of the system.

2.5.3 macchina.io

macchina.io [18] is an SDK for edge device development, made by Applied Informatics GmbH for IoT gateways and connected embedded devices. Its core is open-source, but the company provides add-ons, as well as support and other perks with its paid licensing scheme. It is targeted for deployment on Linux capable hardware platforms with low resources.

The core of the system was developed in C++, dependent on the POCO C++ libraries (open-source, from the same company) for network and internet-based applications. Above it lies the Google V8 JavaScript engine, allowing the creation of applications in the Javascript programming language using the resources created in C++. The core is comprised of several bundles, according to the POCO OSP specifications (similar in concept to OSGi). To run the system the only available way is to build the code from Github. This project is licensed under Apache 2.0.

This SDK is closely tied to Applied Informatics commercially licensed software, with the company being the foremost contributor to the project. Its initial version 0.1.0 dates 2015 and is now on 0.11.0.

The open-source version of this framework supports Modbus and MQTT. These modules also have their code freely available. The commercial version adds OPC-UA, Siemens S7 protocol and CANopen support to the list. Available in the documentation is a small introduction on how to develop an OSP bundle, with no indication on the interface and functionality needed to interconnect a device driver with the rest of the system.

2.5.4 Node-Red

Node-Red [19] is a platform-independent open-source project, initially developed by IBM's Emerging Technologies group and now part of the OpenJS Foundation. It is based on Node.js, an event-driven JavaScript runtime framework for scalable network applications [20]. The project main selling point is the ease of use visual programming interface. This allows a flow-based programming paradigm, constituted by various nodes which operate on the flowing data. It can be installed through Node.js Package Manager (NPM), Docker or using the provided shell scripts for installation on a Raspberry Pi, a Beaglebone or other platforms. It is available under the Apache 2.0 license.

As stated previously, this project was initially developed by IBM's Emerging Technologies group and has since grown to be the most used platform from the ones presented here. This means good documentation, be it from the project itself or problems reported in forums and articles around the internet. Also, thanks to its success, NPM has available from the Node-Red's community a vast collection of nodes for various functionalities.

NPM has available from the community a vast number of interfaces and provides, of interest for a gateway project, protocols such as Modbus, OPC-UA, MQTT and others. Their source code is also available. To create custom nodes, one must develop three essential files: a json file with the node description, a JavaScript file describing its functionality and an HTML file to describe its user interface on Node-Red dashboard.

2.6 REST Api

Representational State Transfer (REST) is a service architectural style, originally for distributed systems. It is resource oriented and presents a set of restrictions when using web standards [21]. REST principles are addressable space, uniform and constrained interface, representation-oriented, stateless communication and Hypermedia as The Engine of Application State (HATEOS) [22].

Addressable space corresponds to the abstraction of information and data through a Uniform Resource Interface (URI), making every object and resource in a service reachable through an URI.

Uniform and constrained interface means that the only methods to manipulate resources are the ones available on the protocol distributed on the service. Representation-oriented means that resource and objects representations are exchanged between client and server, which can be XML, JSON or other formats. Stateless communication to a server means that client session information is not stored, which makes stateless applications more scalable. Finally, Hypermedia As The Engine Of Application State (HATEOAS) defines that the information exchanged between client and server is provided dynamically through hypermedia.

A system that follows a REST architectural style, tends to have good interoperability and modifiability. As such, an API based on these principles, focuses on providing a unified interface to manage network applications, based on the Hyper Text Transmision Protocol. This is especially suited to web services, which when REST principles are applied, are called RESTful services. REST APIs take advantage of HTTP to process data requests, the most common methods being GET, to retrieve data from a resource, POST, to create a new resource, PUT, to update an existing resource and DELETE, to delete a resource [23].

2.7 Summary

This chapter introduced various technologies and standards relevant for the work developed and presented in this document.

It began with an introduction to the general organization of a factory plant, the industrial automation pyramid. It addressed the categorization of the different industrial systems and the networks that interconnect them. Next, a couple of industrial and building automation protocols were presented, demonstrating the market heterogeneity. After this, gateway and DBMS solutions relevant for the work developed were introduced, detailing how each of them work and their applications. Later in the document these technologies will be compared more thoroughly. Auxiliary context was also given for Containerization and REST APIs. Both of these technologies are used by other elements of the developed systems. Furthermore, these are relevant technologies for modern web-based distributed systems and, as such, relevant to this dissertation.

The next chapter takes some of this knowledge and presents the current solutions for implementing a system to integrate various sources of information and present them in a centralized platform.

Chapter 3

Building Automation Systems

To improve the general comfort and well-being of buildings' users, as well as to reduce the cost of wasted energy in heating and illumination, be it in a work environment or a domestic setting, BAS appeared as a means to better control building's mechanical and electrical equipment. They were initially implemented to efficiently control and manage HVAC systems [24], but have since grown to address the multitude of services present on modern building facilities. These services range from the already mentioned HVAC to illumination control, presence monitoring and even access control and security. Energy consumption profiling represents another service of great interest in one of these systems. It provides BAS with a way to collect and process user behaviour, to rationalize energy consumption [25]. All these elements favour high knowledgeable decisions that directly improve the energy efficiency of the building, as well as its comfort and usefulness to the users.

As electronic devices evolved to consume less power and deliver more processing capabilities at a lower cost, more options appeared to integrate these devices into new monitoring scenarios. Sensors and actuators can now be equipped with Internet stacks, that allows them to communicate with each other and form networks [26]. This forms the Internet of Things. IoT opens the possibility for new use cases and increased information collection on BAS, further optimizing the energy management of buildings.

3.1 Architectures

Building Automation Systems are complex infrastructures. They integrate sensors, actuators and other independent subsystems with general or application-specific controllers and present the information captured in a centralized platform for management and analytics. As such, it is advantageous to decompose the system into hierarchical levels. The following sections present two approaches for BAS organization. The first one describes an architecture based on a simplified version of the Industrial Automation architecture presented in Chapter 2. The second one discusses two different architectures found in literature that address the new paradigm of IoT in BAS.

3.1.1 Based on the Industrial Automation Hierarchy

As mentioned before, implementations of building automation systems may closely follow the industrial automation model. As such, the BAS architecture is organized into a hierarchy with three levels, the field level, the automation level and, at the top, the management level, as depicted in Figure 3.1. Implementations of this architecture in BAS differ from IA in the sense that building services functionalities are not clearly defined for each level, but are instead implemented in a cross-level fashion [24].

Regarding the devices present in each level, sensors and actuators are categorized into the field level, just like in the IA architecture. But, for some scenarios, these devices can communicate directly with each other. Such is the case of illumination, as switches directly control the light actuators, without the need of a more capable controller. Protocols implemented in this level are DALI, LONworks and KNX, for example. In the automation level, controllers, commonly represented by Direct Digital Controller (DDC), are used to implement complex control schemes. For instance, these controllers may take into consideration weather reports which, in conjunction with weather and temperature sensors, allow to better control HVAC functionality [27]. This level is also referred to as a bridge level [24]. The systems in this layer aggregate and collect the information from the lower layer and send it to the management layer. These layers are connected by an IP-based network, with BACnet being the most commonly deployed communication protocol [27]. The top-level collects and logs all the data produced by the devices interacting with the physical processes. SCADA systems are used for this purpose and retain the data from the entire system, making it available to the management subsystem. Furthermore, workstations with graphical user interfaces are used to present the information captured. These machines can also exist outside of the local network, allowing remote control of the system functionalities.

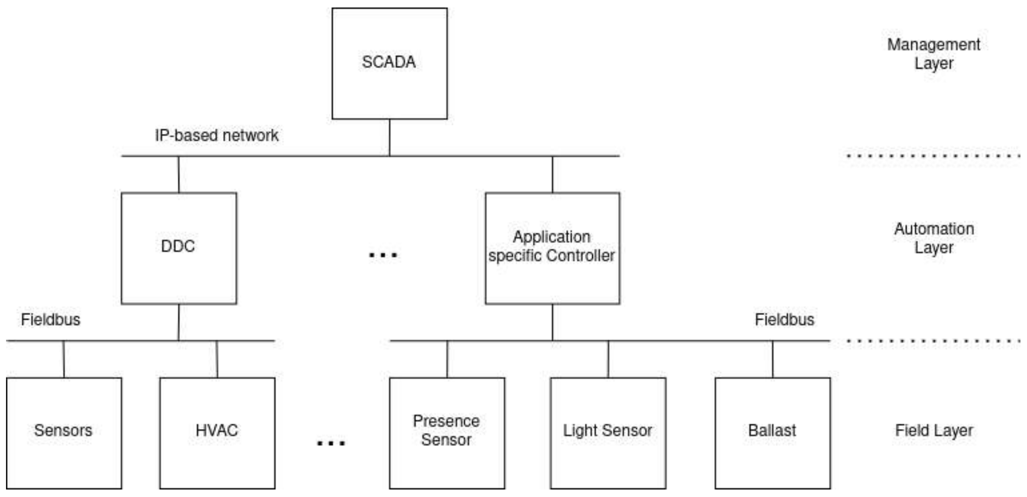


Figure 3.1: BA Architecture based on IA

3.1.2 Based on an IoT platform

An architecture for an IoT based BAS is not as straightforward as the one presented before, especially due to the lack of agreement on a standardized structure [28]. The authors on this paper present architectures from various literature sources, pointing out the lack of architectural uniformity which limits the pace at which new solutions appear. This leads developers into designing custom architectures, specifically targeted at an application, instead of a normalized, standard architecture. In addition, new architectures are developed as extensions of legacy designs, incorporating new functionalities and capabilities into already established technologies. This happens due to the conservative nature of the automation field, where robustness and quality of service are of high importance. It is hard to come up with completely new solutions, as legacy products still need to be supported. So, new systems must also be able to integrate older, heterogeneous and even proprietary protocols, such as the ones mentioned in the previous sections, with new standardized protocols, like MQTT and AMQP.

The authors in [28] propose a general architecture, with special thought put into the security aspect of the system, defending the need for a security framework in all of the seven layers that comprise the Open Systems IoT Reference Model (OSiRM), shown in Figure 3.2. This architecture has at the top, the applications layer, encompassing the potential applications of the system. Below it, the Data Analytics and Storage layer includes all the software needed to present, aggregate and process the information that comes from lower layers. The data aggregation layer follows, and it includes functionality to collect data generated from the various distributed sources that interact with the physical processes. The following layer, the Fog Networking layer, describes the type of communication infrastructure used to link the cyber-physical systems to the topmost layers. These systems are accounted for in the Data Acquisition layer. In this context, cyber-physical systems are electronic devices like sensors, actuators or other embedded devices, that interact and transform physical measurements and actions to the digital domain. The last layer, the Things layer, represents the "things" which can be automated. Effectively represent the physical processes or standalone systems that can be digitalized and integrated into a smart building system.

This architecture is quite complex and extensive, due to the authors focus on presenting a security framework for each one of the hierarchical layers described. This structure can be simplified to only take into consideration the system functionality. This leads to three relevant layers. At the top, there is the Data Analytics and Storage layer, where the building management functionality sits. Bellow it lies the Data Aggregation layer, that converges all the information coming from the lower layer to the top. The bottom layer, the Things layer, merges the bottom two layers of the initial structure and includes the devices that interact directly with the physical world.

This simplification also brings this architecture closer the solution proposed by Lilis, et al. [29]. For the openBMS system proposed, they present a hierarchy with three layers, pictured in Figure 3.3. The Intelligence and Maintenance layer, at the top, encompasses all the decisions, the data aggregation and its unified representation. The middle layer, called the Middleware layer, bridges the physical with the cyber systems, the so-called cyber-physical

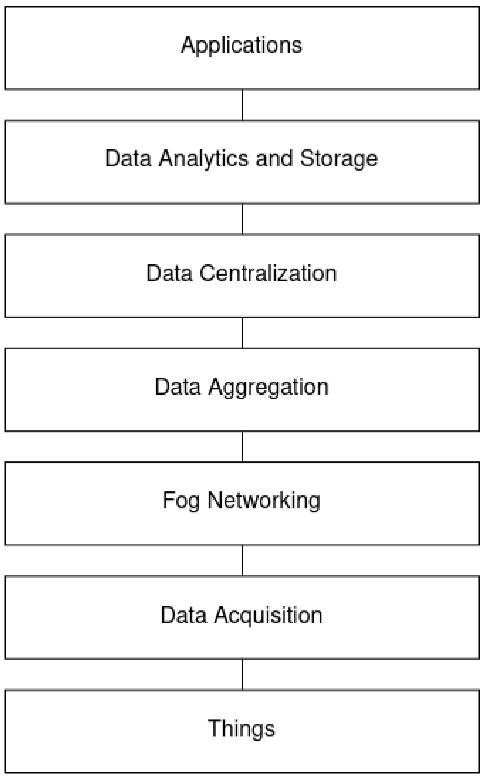


Figure 3.2: Building Automation architecture adapted from [28]

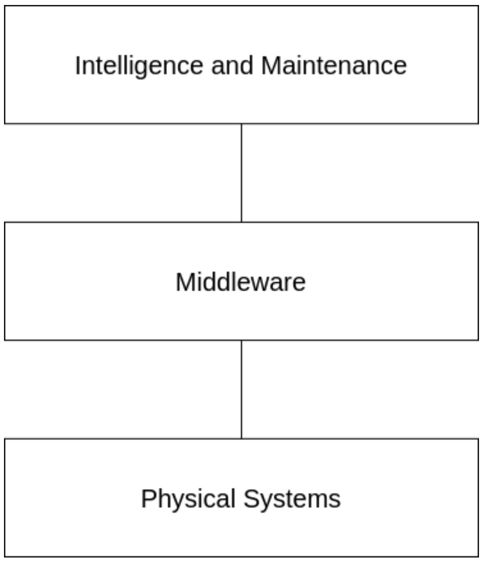


Figure 3.3: Building Automation architecture adapted from [29]

systems. The bottom layer consists of the physical systems' entities, from a logical point of view. Low power electronics form sensors and actuators networks that interact with the physical processes on a building.

3.2 Summary

In this chapter, Building Automation Systems were presented. It began by showing a definition of these types of systems, their goals and functionalities. Afterwards, the current IoT paradigm in the automation field was introduced, showing why the proliferation of low cost but network capable "things", bring more capabilities to these systems. At the core of the discussion lies the possible architectures for these systems. This discussion is relevant because of the inherent complexity that these systems exhibit. Two approaches were explored.

The first approach shows the more mature and well-established architecture based on the Industrial Automation hierarchical pyramid. This is a three-layered architecture, with a field-level for sensors and actuators, an automation layer for controllers, such as DDCs, and a Management Layer with SCADA systems for data aggregation and analysis.

The second approach is an IoT based solution, where two architectures were discussed. A complex seven-layer hierarchy focused on security features and a simpler three-layered solution. Simplifying the complex architecture, focusing the representation into a functional structure, leads to a three-layer structure, similar to the second IoT based architecture presented.

It was concluded that these architectures can merge into one, with the top-level encompassing the data storage, as well as presentation, analysis and management functionalities and an intermediate layer with control and integration functionality, aggregating the information from sensors and actuators, present at the bottom layer. This structure is the one selected as the base of the system developed in the context of this document, and its architecture will be discussed in the following chapter.

Chapter 4

System Architecture

This chapter introduces the overall structure of the developed system. First, the possible Use Cases for the system are presented, from which a set of design requirements are derived. Secondly, the architecture is presented, followed by the discussion of its composing blocks, along with their functional and non-functional requirements. From these, a set of comparison terms were defined, in order to select the most adequate software solutions to use, from the ones introduced in Chapter 2.

4.1 Use Cases

The system was designed to act upon a set of frequent situations in a Building Automation setting. To more clearly understand its requirements, these Use Cases were divided into two categories, based on actors and their interactions with it. The first is the management situation, where the actor, System Manager, will maintain the system upon deployment. As such, it needs to access the data from the entire system, in order to define control patterns and evaluate the energy consumption of the building. In addition to this, it will also need access to the lower-level devices, so as to, for example, turn on and off light fixtures, from a centralized workstation. The second situation concerns the system integrator. This actor has the job of installing and adapting the base framework into specific applications. Therefore, it requires tools to integrate the low-level devices onto the platform.

4.1.1 Management case

As mentioned, the actor in the management case is the System Manager. This entity needs to perform a set of actions over the infrastructure installed in a building, requiring a collection of functionalities for it to react accordingly. As such, it was defined that the actor shall interact with a management interface. This interface needs to make available all the information collected from the various sensors scattered in a building, presenting this data in graphs or other forms of data representation. It shall also make available buttons and triggers to interact with the actuators connected. Furthermore, the collected data shall be processed and further analysed, so the system must allow the user to program processing schemes over the information stored. With this information, four requirements can be defined:

- Centralized access to the information from all the network endpoints of the system;
- Data presentation in the form of graphs, dials, etc;
- Data proccessing capabilities, to define control schemes and analyse the data stored;
- Toolboxes with triggers to interact with actuators.

4.1.2 Integration case

This case assumes an engineer as the actor, deploying a solution for a building automation system. It has the job of integrating a diverse set of products required for the application in question. To accommodate these devices, and as a way to improve the versatility of the system, it needs an infrastructure to quickly support different protocol device drivers. This rules out the need to acquire new hardware or software solutions, each time a sensor or actuator is added to the system, even when they communicate through different network protocols. The integrator also needs to be given freedom to deploy several aggregation terminals throughout a building. As such, all the information produced from the low-level devices should be collected into a centralized infrastructure. In sum, the requirements for this Use Case are the following:

- Tools to integrate devices with diverse communication protocols;
- Centralized point of data storage, to connect various aggregation units.

4.2 General Architecture

The discussion from the previous chapter allows the definition of the system architecture. This architecture, represented in Figure 4.1, is constituted by three layers, organized in hierarchical order.

The top-level is reserved for the management software and functionality that process and analyse the data from the entire system. It can be represented as a workstation that presents the information available in the system and allows further analysis and processing to be made over that data. Adding to data processing and presentation, interaction with lower-level devices is also contemplated here. In this layer, a data store facility is also categorized.

The middle layer aggregates all the information from the various heterogeneous subsystems present on the field layer. To address this, a gateway based solution was defined, so to allow the normalization of the information produced by the devices in the layer bellow.

Finally, the field layer encompasses all the sensors and actuators a system of this kind can have. These devices represent off-the-shelf products, or custom ones, that communicate typically through industrial or building automation communication protocols, such as Modbus [30], EtherCAT [31] and DALI [32], to name a few.

This dissertation focuses predominantly on software solutions present in the management and middleware layer. The following sections introduce them and define the requirements needed in this context.

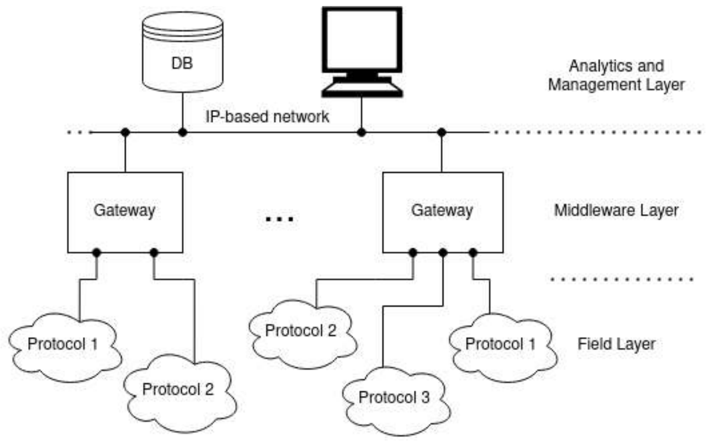


Figure 4.1: System architecture

4.3 Gateway

The gateway is the element that allows the aggregation of the different network devices available in the field level, retrieving the data from the various sensors and actuators and storing or forwarding it to the higher levels. As a result, it needs to possess the required technology to understand the various protocols available in a Building Automation setting. These can be hardware related, as there can exist physical layer differences between these methods of communication, and software related, defining how the information is structured. This dissertation focuses on a software-based gateway solution. The technologies evaluated were introduced in chapter 2 and will be further compared in Chapter 5.

For the developed system, there is a need to integrate various heterogeneous protocols from the building and industrial automation area, as discussed in the Integration Use Case. Besides, the chosen framework needs to be capable of allowing the addition of new protocol stacks, without changing the entire software structure of the gateway. Furthermore, as it was decided that this solution shall be open source, there needs to be a consideration about the type of license the software has, with the more relevant solutions being the ones with highly permissive licensing schemes.

It should additionally allow for easy application creation, meaning that the software needs to include modules to perform decisions on the flowing information. This allows the implementation of lower level, and thus lower latency, control schemes, important to the target applications of the developed system. Furthermore, connection to external databases is also something the chosen platform should already support.

Putting these requirements into a representation of the chosen framework leads to the gateway structure pictured in Figure 4.2. Device drivers establish the connection to the cyber-physical devices. The idea is that these elements are interchangeable and can be added to the system as needed. If, for instance, the final deployment requires connections to three different protocol networks, then the three corresponding software stacks should exist.

As these stacks can have quite different interfaces and APIs, a normalization agent is needed. It has the job of exposing the different network interfaces to the northbound devices

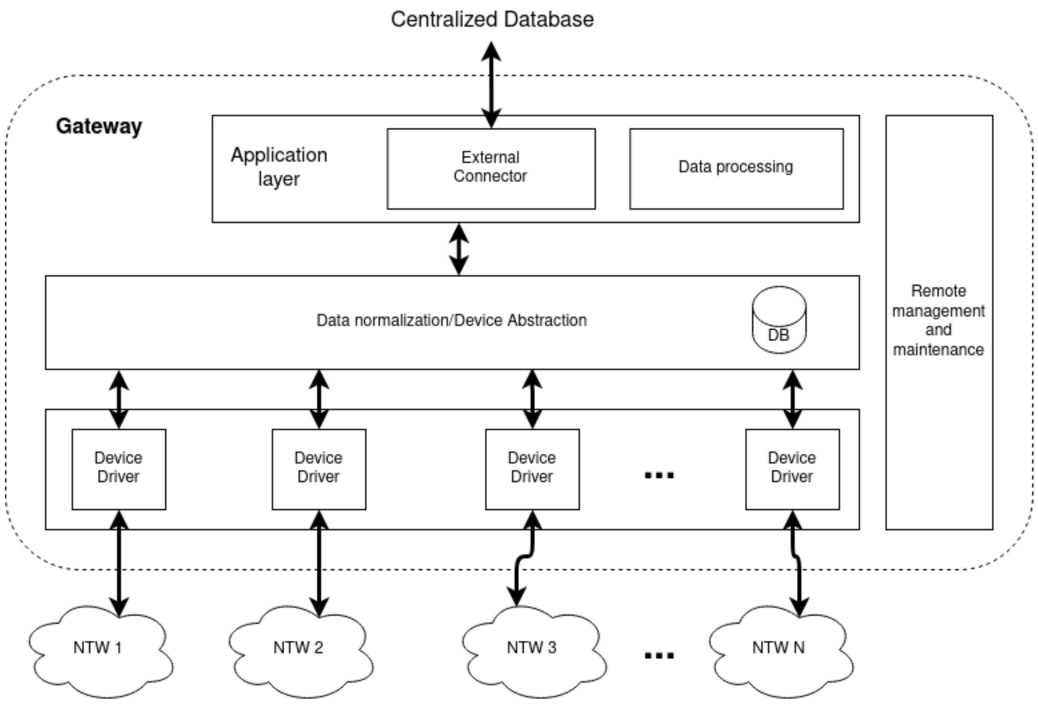


Figure 4.2: Gateway architecture

through a unified interface. Furthermore, the data captured in the gateway should also comply with this normalization. As so, the data can be structured in a set of values, answering to the following key questions:

- Who produced the information?
- What type of data is being read/written?
- What is the actual value read or to be written?

Now, with a unique dataset, data can be processed, independent from where it came. In order to fullfill this, the gateway should be equiped with a data processing module. This allows the creation of processing schemes to act on the flowing information, in order to select data to be sent upstream, or close control loops with the low level devices.

To export the data to the Management Layer, an export mechanism needs to be supported. It will allow the connection to the centralized data store facility. As so, the export module should already have the needed database connectors or protocols, to accomodate this. All of these individual modules should be interconnected through a management and maintenance platform with remote access capabilities. Its through here that the modules are configured and information about their state is retrieved.

As a final remark, a database should also be embedded in the gateway. In situations where there is a poor connection to the northbound devices, the data collected by this subsystem needs to be stored, at least during the time that the connection is down, so that information is not lost. Once the connection to the northbound is reestablished, the data cached is flushed to the central database and operation resumes normaly.

4.4 Database

The system must allow various gateways to be connected to a centralized data store facility. As such, it is relevant to define a database management software to structure the data captured by the field level devices and make it available to the top layer. The main requirement for this subsystem is the capability to handle the type of data that will be circulating in the system. The information generated at the field level is typically structured as a time series, a pair consisting of the value that was measured and its corresponding timestamp. As so, the database management system must be capable of dealing with this kind of data with a high level of performance in terms of query response times.

Three types of DBMS where presented in Section 2.4 from Chapter 2. Relational databases where shown as the more mature and robust solution, focusing primarily on high degrees of assurance that data transactions are completed. Their main drawback is the fixed data schema, which forces the data to be structured *a priori*, in a predefined manner. This can limit the potential scalability of the database, specially as a solution for the problem at hands. The system being developed needs to be able to adapt to new situations and can, as such, be expanded multiple times during its lifecycle. This means that new devices need to be supported quickly and at a low cost. A fixed schema database makes this process difficult, as it needs to be adapted to accept the data generated by the new devices, since they can have different properties than the ones already installed.

Document-based DBMSs target, in a way, these drawbacks by getting rid of the fixed data schema. Theses databases, integrated into the NoSQL category, accept flexible structures of data, mainly in the form of JSON files or similar, organizing the information in documents, instead of the table model of relational databases. This way, the database does not need to be configured beforehand to accept new data models. What document-based DBMSs loose is performance, having slower injection, retrieval and search times in comparison to relational databases, in certain situations. They also have more relaxed requirements in terms of transaction robustness.

The final group of possible solution are the time series databases, a subset of NoSQL databases. These systems have a similar approach to the document-based ones, by having no predefined data schema. As so, good scalability is assured. Furthermore, by adapting the type of data to be stored into sets of values and timestamps, these systems can use the time marks as keys to the rows of data they are associated with. This way, search times are drasticaly reduced, as well as injection and retrieval times, making it possibly the solution with the best performance for the case in hands.

4.5 Management Interface

The management interface is the point of contact between the building administrator and the network of devices present in a BAS. This is a centralized platform, where the entire data generated in the system is presented and high-level control and administration are made. This platform needs to connect to the database management system to retrieve the data to display and process. It also needs to interact with the gateways, to pass the commands to the actuators, and thus closing the control loop with the sensors. It is also here where more complex control schemes can be implemented, making it possible to, for example, implement data processing and analysis to infer the state of utilization of the building in terms of energy consumption. To address this, a couple of requirements were defined. The first is that the chosen platform must have a set of data presentation toolboxes, in order to chose the type of graph that best represents the information displayed. Secondly, the user needs to have tools to analyse the data presented. Included in these tools should be, essentialy, mathematical operators to calculate, for example, averages, maximums and minimums over the data being captured. It can also include alarmistic capabilities, allowing the definition of alarms when certain values exceed preset user defined thresholds. Toolboxes with buttons, rotary dials, sliders and other means of input are also needed, to display triggers and controllers for the actuators.

All the information to be displayed in the management software comes from the database. This is the data produced by the low-level devices and sent upstream through the gateway to the database. As such, this software needs to have the required connector to interact with the centralized database.

For the downstream data, or the commands sent from the management to the actuators, two approaches are possible. The first is a direct connection to the gateways. This would be a low latency solution, at the cost of loosing the commands and data sent to the southbound devices logs. Having all the data available is of extreme importance for the system being developed, as this information can help diagnose failures in the system. As such, a better solution is to send the commands to the database, which are then retrieved by the gateways. This way, there is always a record of the transactions made between the management system and the gateways. To support this, the gateways need to make available metadata from the devices connected to them, which should include the commands available for each device driver. The global metadata of the devices available should also be present in the centralized datastore facility, and the gateways should share this information, either periodicaly or when changes occur.

4.6 Summary

This chapter introduced the structure defined for the system developed in the context of this dissertation. Initially, two possible Use Cases were presented, one from the management standpoint and another from the integration point of view. These Use Cases lead to a set of core requirements the system needs to comply with. First, it must be possible to integrate devices with different communication protocols into one unified notation. For this a gateway based solution was chosen.

This device was presented with its possible architecture, derived from the defined requirements. This architecture incorporates the device drivers for each possible network protocol to be used, which can be interchangeable and expanded. The data captured by these software stacks are then normalized into a unified notation. This is fullfield in the Data Normalization block. This block also supports an embedded database as a system cache, in the case of a connection failure to the centralized database. Communication to the database is made through an external connector and alongside it a data processing block operates over the information on the gateway, defining, for example, control loops between sensors and actuators. The gateway should also have a remote management and maintenace interface block.

To allow devices to be distributed in a building more easily, a centralized databased was defined. This allows the integrator to distribute the field layer devices through various gateways, all connected to one data store. It also supports the record of all the information that flows on the system, meaning that it not only stores the data captured by the sensors, but also the commands sent to the actuators and all the gateway and low-level devices metadata. Three different types of DBMSs were presented, with the time series database being the one that best fits the requirements defined.

A Management Interface was also defined. This element processes the information present on the database and displays it on a user interface. It also has buttons and triggering elements to interact with all the devices in the system.

The next chapter presents what was developed, how the system was implemented and the technologies chosen. This choice comes from the comparison made between the technologies initially presented in Chapter 2, with the terms defined by the requirements discussed here.

Chapter 5

Implementation

The main focus of this chapter is the discussion of possible technological solutions for the requirements presented in the Chapter 4. Furthermore, it also shows the final demonstrator and explains how the selected technologies were integrated with each other.

The first section introduces the developed system. The work realized is explained by presenting the various architectural building blocks of the system and the solutions chosen for each.

Next, the technological decisions are explained. Each system block is discussed, showing how and why the frameworks chosen were the most adequate ones. These sections first introduce the comparison terms, derived from the requirements set in Chapter 4. Then, the projects are compared with each other in respect to each one of those terms. The sub-sections close with the overall comparisons and decisions for the choices made.

The chapter ends with a description of how each solution was deployed. This ranges from a more indepth explanation for each and how they were configured and programmed. EdgeX is introduced with more detail, presenting its microservice architecture and their functionality. It then continues with the necessary modifications and configurations made to the framework. Following, Influx is discussed, showing the configured data structure for the system. Finally, the use of Node-Red as the base for the Dashboard and the block interconnection is discussed and its implementation explained.

5.1 What was made

The system developed is a proof-of-concept for a Building Management system. It serves as a demonstrator and a test bed for the technologies discussed in this chapter.

Picking up on what was discussed in Chapter 4, the developed system is represented as a three layered architecture. The top layer, the management layer, comprises the management interface and the centralized database. The middle layer corresponds to the gateway, the industrial protocol aggregator. The lower layer is where the devices that interact with the physical processes are categorized. As such, the final system, represented in Figure 5.1 has a Dashboard, developed in Node-Red, for data presentation and interaction with the low level devices, working alongside an InfluxDB datastore. A software-based gateway framework,

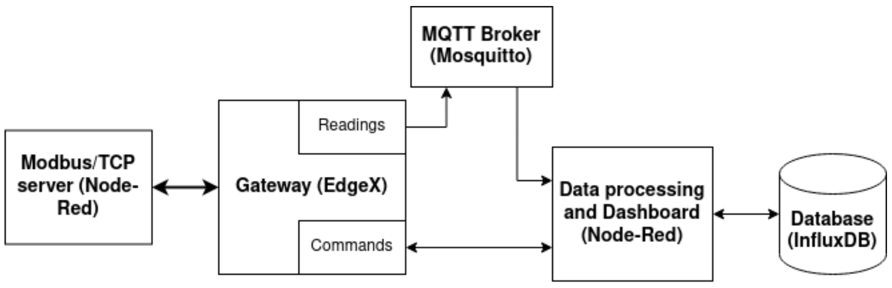


Figure 5.1: System block diagram

EdgeX Foundry, collects all the information from the field level devices. These were simulated with a virtual Modbus device.

These elements are interconnected in the following way. The virtual device, implementing a Modbus server, is connected to EdgeX via Modbus/TCP protocol. Its data is polled by EdgeX and periodically sent to an intermediate point, called the block interconnection, via MQTT. This was developed in Node-Red and bridges EdgeX with the database and dashboard. It selects and processes the data comming from the gateway directing it to the corresponding dashboard element and measurement database. Eclipse Mosquitto [33] was used as the MQTT broker.

5.2 Technologies used

To choose the more correct and adequate technologies for the gateway, the database and the management interface, a research was made on the most popular solutions for each category, in the context of IoT. The following section presents the comparison made between those technologies. It first introduces the comparison terms, presenting the information found for each framework. After a brief discussion, the results of the comparison are presented in the form of a table, with the corresponding discussion at the end of each sub-section.

5.2.1 Gateway

Various solutions for gateway systems exist today. Off-the-shelf products mainly focus on translating data from one protocol to another, and are clearly not the ideal choice for the system here presented. As such, the focus of the research was open-source software-based frameworks that are in-line with the requirements defined in the previous chapter. These solutions were decided to be open-source as they offer more flexibility for customization to specific use cases, if needed. For this to be possible, and more versatile, it is also important that the open-source licensing schemes used by the project owners have highly permissive licensing. The systems here presented all match this requirement, as they use Apache-2.0 and EPL-1.0 (Eclipse Public License).

Four solutions were chosen to be investigated and compared in this dissertation. They are EdgeX Foundry [17], Node-Red [19], macchina.io [18] and Eclipse Kura [16]. These were already introduced in Chapter 2. Here, a comparison will be made between them, in order to choose the most appropriate framework for the Building Automation system being discussed in this dissertation. They will be compared in six terms. These were defined based on the requirements expressed in Chapter 4. They are:

- Supported Industrial protocols
- Tools to develop new device drivers
- Data processing tools
- External database connections
- Embedded database
- Remote management and maintenace

Supported Industrial Protocols

This entry compares the various offers for industrial and building protocols available for each framework. This was evaluated in terms of which protocols these solutions support out-of-the-box. All have support for BACnet, Modbus and MQTT, but in this department, Node-Red takes the advantage, mainly due to the openness of the solution and the size of its community. Although there are nodes for the protocols presented, their performance and functionality was not evaluated. These can sometimes be incomplete or unsupported after short periods of time, as they are products of the community surrounding the project and they are not subjected to submission policies. macchina.io and EdgeX have a similar number of supported, open source, device drivers, with macchina.io having support to CANOpen, OPC-UA and Siemens S7 protocol on the paid version. Eclipse Kura has the least amount of supported protocols, having only OPC-UA and Siemens S7.

Tools to develop new device drivers

Here, the frameworks were evaluated in terms of capabilities and documentation to create and add support for new device drivers. EdgeX Foundry is the most complete solution in this case, as it has a device driver SDK specifically for this purpose. It provides the developer with APIs that allow the integration of protocol software stacks with the gateway. Futhermore, it is also the platform that has the most clear documentation regarding this issue. Node-Red follows, having also good documentation on how to develop new nodes, although not to the extent of the solution presented by EdgeX. Both macchina.io and Eclipse Kura were considered as not having the tools for this purpose. This is due, essentialy, to poor documentation and unclear mechanisms for a developer to add support for new protocols.

Data processing tools

This element compares the features each framework has to develop data processing applications. Data processing applications are definitions that can be added to the framework, operating over the flowing data. It is of high importance to be able to add control loops at the gateway level. In respect to this, Node-Red is the most complete solution. The primary objective of this platform is to offer a low code programming platform to process data from source to destination. As such, it offers many nodes that allow the definition of control rules, needed for the system developed. Eclipse Kura comes next, with its Kura wires. This is in many ways similar to the dataflow programming model in Node-Red. Nodes represent assets from the system, like sensors, actuators or PLCs and can be connected between them. There can also be nodes which implent operations over the flowing information. EdgeX is not as featureful as the other frameworks here presented. It supports a Rules Engine, allowing the developer to define, for example, control rules over the data produced, to trigger actions on other devices connected to the framework. The new version of the Export Services, the Application Services, will also allow some processing when sending data to external endpoints. macchina.io has a different approach. It leverages Javascript to build applications that interact with the various modules on the system, be it a device or webservice. As such, the data processing capabilities is not on par with the previous ones, at least in terms of its usability, as any processing required needs to be built from the ground up as an application. With this, it is possible to see that the two systems with the most complete solutions for data processing are Node-Red and Eclipse Kura, followed by EdgeX and macchina.io.

External database connections

Here, methods for connecting to external databases are compared. Eclipse Kura and Node-Red are the most complete solutions. Kura has support for Apache Camel, an integration framework, which allows created data to be easily transfered to a determined endpoint, in this case, the database. Apache Camel has support for the three databases covered in this dissertation. Node-Red, on the other hand, has many nodes for database connection available from the community, with these three included. EdgeX could, through the Application Service SDK, support any needed database, but doesn't have any out-of-the-box connector available to use. The version used during the development of the system didn't have this feature built-in, instead relying on the Export Service, supporting REST, MQTT or 0MQ connections. macchina.io supports the use of Redis and SQLite only.

Embedded database

As for the support of an embedded database, every framework has some kind of connection available, but only EdgeX and Kura have it integrated, by default, into the framework. EdgeX has two options, Redis which is the new default and MongoDB. Kura has the H2 database integrated. The other two, while having the connectors available, do not have the database integrated on the deployment package. This would need to be done when implementing either platform.

Remote management and maintenance

All the platforms have mechanisms for remote interaction and management, although with different approaches. macchina.io makes available a web User Interface (UI) that allows device management, testing and application programming through an embedded text editor. It is also through this interface that the bundles that comprise the framework deployment are managed. These bundles can represent device drivers and custom applications, for example. Node-Red approach is similar, as its interface is a web UI with the main programming environment. It allows setting up flows that define the applications running and to be run. EdgeX has two accessible interfaces, one for the configuration and registry through Consul, and another to manage an instance of EdgeX. Consul displays information from all the microservices running, presenting them listed with their operational state. The EdgeX UI allows the management of EdgeX objects and monitors the EdgeX data flow. It also allows the configuration of various microservice functionalities, such as defining scheduling schemes and instantiate device drivers. It should be noted that this UI is for demonstration only and is not advised to be used in a production setting.

Comparison

A condensed view of all of the information gathered concerning the comparison terms layed out previously is presented on Table 5.1. It is clear to see that, for the context being here discussed, Eclipse Kura and macchina.io are quite limited in their offering of supported protocols out-of-the-box and tools available to integrate new device drivers into the ecosystem. Node-Red and EdgeX Foundry are more advantageous in this regard, with more support for higher number of Industrial and Building automation protocols. Furthermore, they have good documentation on how to integrate existing protocol stacks into these products, with EdgeX having an advantage over Node-Red, due to the examples and extensive descriptions documented. The determining feature between these two alternatives is the existance of database connectors and the presence of an embedded database. Both were defined in Chapter 4 as important requirements for a gateway system in the context of a BAS. Preference was given to the solution with an embedded database. This is more advantageous from the point-of-view of integration, as more work, and thus time, is needed to integrate an embedded database onto Node-Red than to support a way to connect a database to EdgeX. As so, EdgeX was the chosen framework to use as a gateway solution for the work developed in the context of this dissertation.

Feature	Eclipse Kura	Node-Red	EdgeX Foundry	macchina.io
Supported Protocols	OPC-UA, Siemens S7	OPC-UA, BACnet, Modbus, Siemens S7, MQTT	OPC-UA, BACnet, Modbus, MQTT	OPC-UA (paid), Modbus, Siemens S7 (paid), MQTT, CANopen (paid)
Tools to develop new device drivers	No	Yes	Yes	No
Data Processing tools	Yes	Yes	Yes	Yes
External database connections	Yes	Yes	No	Yes
Embedded database	H2 Database	No	Redis/MongoDB	No
Remote management and maintenace	Yes	Yes	Yes	Yes

Table 5.1: Gateway frameworks comparison

5.2.2 Database

This dissertation also focused on finding a suitable database for the developed system. Although the research made did not go too much in-depth, three popular databases were compared. MySQL, MongodDB and InfluxDB were selected, each one representing a different data storage model. This comparison was made in terms of performance characteristics collected from a bibliographic survey and considering also the following features:

- Data model
- Query language
- Retention policies
- High availability

The choice of these three DBMS was decided by researching the most popular solutions for each category. The research used the Google search engine to get their popularity index. It was measured in terms of the number of page hits each search returned. The database sample is presented in Table 5.2 and features three examples for each database kind. SQL databases represent databases with a structured query language interface. These are usually the more traditional relational databases. NoSQL are databases that, contrary to the relational data model, do not have a strict data schema, and usually do not use a structured query language. TSDBMS are databases targeted specifically at time series data. This research was made on the 30th of March of 2020 and the query terms used followed the structure presented in Table 5.3.

The results from Table 5.4 show MySQL as the most popular solution for SQL type databases. For NoSQL, the number of pages in Table 5.5 is superior for MongoDB, and in Table 5.6 the one with the bigger result is InfluxDB. With this, the database samples is reduced from nine elements to these three: MySQL, MongoDB and InfluxDB. These are the targets of the comparison that follows.

SQL	NoSQL	TSDBMS
SQLite	Apache HBase	RRDtool
PostgreSQL	Apache Cassandra	OpenTSDB
MySQL	MongoDB	InfluxDB

Table 5.2: Database sample by type

“database name” “database” “iot”

Table 5.3: Search query format

Name	Results	Query
MySQL	11 900 000	"mysql" "database" "iot"
PostgreSQL	6 760 000	"postgresql" "database" "iot"
SQLite	582 000	"sqlite" "database" "iot"

Table 5.4: SQL databases search result number

Name	Results	Query
MongoDB	5 280 000	"mongodb" "database" "iot"
Apache Cassandra	167 000	"apache cassandra" "database" "iot"
Apache HBase	88 400	"apache hbase" "database" "iot"

Table 5.5: NoSQL databases search result number

Name	Results	Query
InfluxDB	140 000	"influxdb" "database" "iot"
OpenTSDB	64 800	"opentsdb" "database" "iot"
RRDtool	49 600	"rrdtool" "database" "iot"

Table 5.6: Time series databases search result number

Data model

This term compares the various data models of the different DBMSs. InfluxDB is a Key-Value database, so it works as a kind of associative array. It has unique key fields, which, in this case are timestamps indicating, for example, the time of the measurement. The value field can be a collection of tags and numeric data. For this database, it can be separated into groups, field keys and measures. Field keys represent the context of the measurement, the sensor from where the value was read and its units, while the measure field holds the actual reading. Document-based databases, such as MongoDB, store the information into objects with an arbitrary number of attributes. These can be JSON-like structures, allowing the existance of objects with a different number of attributes. MySQL is a relational database. It has a logical data structure similar to that of a table, with unique row identification. Furthermore, different tables can be logically associated by indexing keys.

Query language

Here, the methods to interact with the databases are evaluated. MySQL, as the name sugests, uses a Structured Query language. This is a set of easy-to-understand phrases that allow to query the database in a flexible manner. Through simple expressions, complex search queries over all the data stored on the database can be made. Influx has two approaches. The first is similar in concept to SQL, although with some differences to accomodate specific characteristics of the database. The second is an inhouse developed scripting language based on Javascript called Flux. MongoDB doesn't have a query language per se. It relies on programming language specific APIs to interact with it.

Retention policies

Retention policies define the time that data stays on the main storage facility. It helps to clear up space when the information stored loses its value. MySQL doesn't have this feature built-in. As so, a third party element is needed to workaround this. MongoDB has a Time To Live (TTL) setting for this purpose. The TTL Monitor is a thread, separated from the server thread, that keeps notice of TTL indexes and deletes the corresponding document when it expires. This is configured by indicating how many seconds the data should live or by defining a specific clock time for the data to be eliminated. InfluxDB also has the capabilities of defining retention policies. This is defined for measurements and tells the server to compare the timestamp associated with the measurement and the one registered on the system. If that period is higher than the one defined by the retention policy the data is eliminated. In addition, it also allows to define continuous querys (InfluxQL feature) that define downsampling routines over the data stored.

High availability

High availability concerns the capability of adding redundancy to the databases. This means that the DBMSs support infrastructures that allow multiple server instances to have copies of the same data set. Not only this, but a highly available system should also be capable of recovering in case of failure. All of the three solutions here presented have features for data replication. This allows, for example, a setup of two or more database servers, spread out into different hardware, locally or remotely, synchronized to the same data. When an insertion is made on the main server, automatic mechanisms synchronize all the databases, so the data is consistent across them. Through this redundancy, the database deployment is more robust to hardware, operating system and application failures. MySQL and MongoDB provide data replication capabilities on the open source version, while InfluxDB only provides this on the commercial product. In the context of this dissertation, as the focus is on open source solutions, InfluxDB is considered as not having high availability features.

Comparison

From the analysis of the features of the database solutions, condensed into Table 5.7, it is clear that all three databases have similar features, apart from the data model and the type of interaction. As mentioned in Chapter 4, in the requirement analysis, a time series DBMS would be the solution that more closely corresponds to the application needs. Furthermore, for the required workload, it is capable of higher performance compared to the other two.

Performance characteristics between MySQL and Mongo were evaluated by Sharvari Rautmare and D. M. Bhalerao [34]. The article presents a test workload similar to the one present here, with focus on time series data. Performance was evaluated in terms of Select and Insert querys and showed that there was not a clear winner. MongoDB and MySQL performed similarly, with MongoDB taking the advantage in insertion response times. It is also noted that overall MySQL responses were more stable than MongoDB. Sergio Di Martino et al. [35] made a similar comparison between NoSQL databases, namely Cassandra, MongoDB and InfluxDB. The authors show that, on average, InfluxDB performed better than the competitors. They were evaluated in terms of batch-ingestion time, retrieval time and disk usage. The retrieval times were evaluated from two angles: one through indexed search and the other through non-index search. This means that, for a time series data set, a search was made over the timestamps associated with the measurement or through a non-temporal atribute. From the analysis of these two references it is possible to extrapolate that InfluxDB is the more adequate solution for the system being developed.

Feature	InfluxDB	MongoDB	MySQL
Data model	Key-Value	Document-based	Relational
Query language	InfluxQL/Flux	No	SQL
Retention Policies	Yes	Yes	No
High availability	No	Yes	Yes

Table 5.7: Database Comparison

5.2.3 Dashboard

A thorough research was not made to evaluate a dashboard solution for the system. Node-Red was the framework used to developed a graphical interface, as it was also used to simulate the Modbus server (discussed in the following section) and aligned suficiently enough with the requirements indicated in Chapter 4.

Node-Red has available a node collection specifically designed for dashboard creation. It has toolboxes for data presentation, in the form of time graphs, gauges and bar graphs and a collection of interactive buttons and sliders. Furthermore, a high number of nodes to connect with the most popular databases, as well as an out-of-the-box MQTT client and robust capability for developing data processing flows put this framework in-line with the requirements set in the previous chapter.

5.3.1 The industrial protocol used

Modbus was the chosen protocol to evaluate the system. It is supported by every framework presented earlier and is popular in both Industrial and Building automation. Furthermore, it is an open standard protocol, meaning the standard is published openly and, as such, many device simulation platforms are available on the web. For this purpose, Node-Red was used to emulate a Modbus server. This server emulates a device, for example a PLC, that has a set of registers with values read from sensors, or to be written to actuators.

The device in question was modeled after a Dent PowerScout unit, model 3037, used in the example provided by EdgeX, showing how to setup the Modbus Device Service [36]. The device implemented has the interface presented in Table 5.8. This table shows the mapping of the Modbus server registers that can be read. The Current, Power and Voltage are read-only holding registers while the Switch coils can be written and read. The Modbus server deployed is pictured in Figure A.2.

Type of Register	Register Number	Register Address	Measurement
Holding Register	10	9	Current
Holding Register	4004	4003	Power
Holding Register	4018	4017	Voltage
Coil	2	1	Switch

Table 5.8: SQL databases search result number

5.3.2 EdgeX

EdgeX Foundry was the chosen platform for the gateway. It is a framework built specifically to interconnect "things" to IT systems [37]. It focuses on interroperability and aims to have a platform agnostic distribution. It is composed of a set of loosely coupled microservices, each one adding specific functionality which can be opt-out if the hardware has low-resources. EdgeX is also designed to be a reference implementation and encourages development of custom implementations of these services. Throughout the development of this system, the EdgeX version used was Fuji (1.1).

Microservice Architecture

The architecture of the framework is presented in Figure 5.2. In it is easy to see the various microservices that compose the system.

EdgeX separates sensors and actuators into the South side and the IT systems into the North side. The core services connect these two sides. These include the Core Data, Command, Metadata and the Registry and Config services. The core data is called by the Device Service to store the data captured from the South side devices. This service has an embedded Database, Redis by default, to store the data received persistantly until this is sent to the North side services and exported, in the context of this dissertation, to the centralized database. This mechanism can also be deactivated, if the application only requires data streaming from sensors. The Command service serves as a central control unit that is capable of sending commands to the other running services. It can, for instance, be asked to request a reading from the Device service, or configure the Rules Engine and other analytics services available. It receives requests from outside sources, such as a Cloud infrastructure that, for example, needs to configure a Device, or other services from the EdgeX collection. The Metadata service has persistant information about all the devices connected to EdgeX. It stores which devices are connected, how their information is organized, its type and how to command them. This information is stored, by default, in the embedded Redis server. Lastly, the Registry and Config service makes available the location and status of all EdgeX microservices. It also makes available a repository of initialization and operating values for the services configuration properties. This service takes leverage of Hashicorp's Consul platform [38].

The Device services are what allow EdgeX to interact with the low level devices. They translate the infomation that arrives from the multitude of protocols available into one unified notation, used throughout EdgeX services. These services also have to manage the commands and requests received and send them to the devices. Apart from this, they are also the way to manage the state of the sensors, actuators or higher level equipment, change their configuration and discover newly connected devices. The Device service is in fact a software abstraction layer, adapting a protocol driver/firmware into to the EdgeX echosystem. The EdgeX developers provide an SDK to ease the integration of proprietary or non-supported protocols into a EdgeX Device service.

To send the data captured to external storage or analytics, up until version 1.1 (Fuji) EdgeX had available the Export Services. Client Registration and Distribution allow the registration of external connections, be it a cloud infrastructure or the centralized database used in this work, and on-gateway services. The Client Registration allows for the registration of clients, storing their connection properties and the data they are subscribing to in the embedded database. The Export Services take the information registered, applies the needed data filters and sends the data to the endpoint. Two protocols are supported, a REST API through HTTP and MQTT. From version 1.2 (Geneva) onwards, the Export services were deprecated and substituted by the Application Services. These allow the definition of processing paths for data generated on EdgeX and exporting them to a custom endpoint. The applications services change the registration approach used before to a configuration based one. For this, EdgeX makes available an Application Service SDK for developers to deploy custom connectors adjusted to their needs.

The Supporting Services offer a collection of functions that include If This Than That (IFTTT) processing through a Rules Engine service, command and operation scheduling through the Scheduler Service, logging functionality, and alarmistic and notification events. The Rules Engine service implements the Drools [39] platform at its core and allows setting up event triggering mechanisms. This can be a rule set on sensor values that if met, triggers an actuation command on a south device. This service can receive data from the Export Services, on which the rules engine would be registered, or it can receive the information

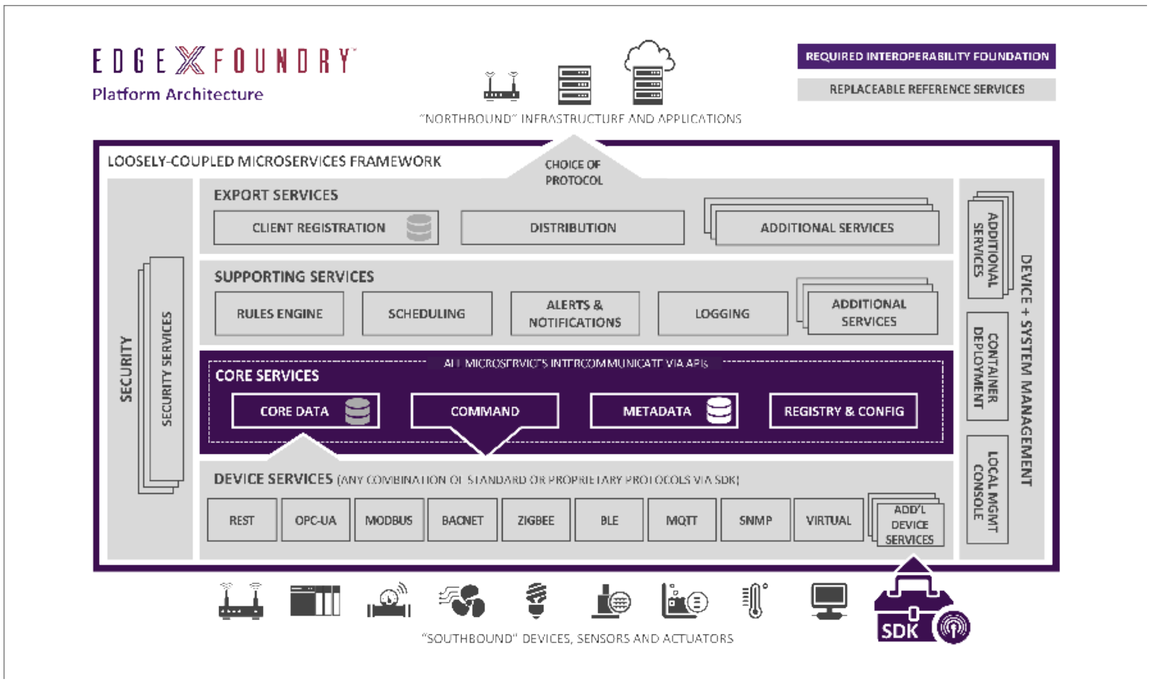


Figure 5.2: EdgeX microservice architecture taken from [40]

directly from the Core Data service, lowering the response latency. The scheduler service, by default, has the functionality of periodically calling a Scrubber service to clean old, exported data. It can also be configured to trigger commands on any EdgeX service. It can, for instance, be configured the send a read command to a device service every second. The Alerts and Notifications Service sends information to other services when events occur. It is used, for example, to alert that a measurement exceeded a given threshold, defined in the Rules Engine Service. Through the system management services it can also alert on some EdgeX module malfunctioning to the elements subscribed to this information. It communicates to on-gateway services and external processes. At the moment, it can redirect these notifications and alerts through e-mail and REST callback channels.

Adding to the services described before, there are also the Device and System Management and the Security services. The first takes care of managing all the EdgeX services, from starting and stoping to retrieving service resource consumption. This information can then be accessed from external management tools. The second adds a collection of security features horizontally to every EdgeX Service. These include secrets creation storage, to store access passwords, certificates and others, limit remote access to all of EdgeX services REST API and user account creation.

As stated over this explanation, these microservices communicate mainly through RESTful HTTP APIs, both for outside access, as well as in between them. In addition, data present in the Core Data is distributed to the Export Service via a ZeroMQ bus. This bus can also be used to send data to the Rules Engine when the default REST API is not performant enough in terms of latency or volume of information.

Deployment

The main deployment platform used to distribute EdgeX is Docker. This is the default route for users that want to evaluate the complete framework. Each microservice is distributed as a Docker container, available from EdgeX's Docker Hub repository. The easiest way to launch an instance of all of the needed services is through Docker Compose. This allows the setup of a configuration file that describes the services to be launched, the virtual networks, if needed, the shared memory locations, through volumes, and the start-up order. Docker compose takes care of downloading the images from the repository, creating the containers and initializing all the services. This method is available for x86-64 and arm64 cpu architectures. This was the prefered method for running the gateway framework. The docker-compose file used can be seen in Listing A.1.

For the system here developed, this gateway framework was deployed in two infrastructures. The first was a simulated environment, in a Virtual Machine running Ubuntu 18.04 with 1GB of RAM and two cpu cores. The second was on a Raspberry PI model 3B running the server edition of Ubuntu 20.04, for the ARM 64-bit architecture. The deployment was made mainly through Docker, using the Docker Compose file present in EdgeX github project [41], although modified to not use MQTT, but instead use the Modbus Device Service. Changing the database service used was also necessary, as the default at the time was MongoDB and not Redis.

Modbus device driver setup

As stated before, the Modbus protocol was used for the demonstrator in question. EdgeX has a Device Service already created to talk to Modbus devices. The setup can be made in two ways, before instatiation, by defining the configuration files that describe the device to be connected and through the Core Metadata service REST API.

There are two configuration files, one with the device service properties, a configuration TOML (Tom's Obvious, Minimal Language) file and a device profile YAML file. The first describes the intrinsic characteristics of the service, in respect to its interface with EdgeX, its name and application port number, the Core services ports, as well as the Modbus devices to which it connected. The device profile describes the properties of one of the connected devices. In the case of a Modbus device, it is comprised by the list of commands and resources associated with the device and the mapping between the REST API and the Modbus registers. These files, when referenced in the Docker Compose file, load automatically when EdgeX boots.

The second method is by describing the information indicated above in JSON format and sending it to the Core Metadata service through its REST API. This requires an HTTP POST request. It allows the integration of new devices through remote requests to an online system, elevating the flexibility of this framework. Furthermore, all the devices can be managed by this method, not only when adding new equipment, but also when removing the old one.

An accessible device in a Modbus/TCP network is seen as a server. The device service should, as such, be viewed as a Modbus client, the actor that accesses the information stored on the device. The way this was implemented was through the use of scheduling features to periodically poll the Modbus server for the data needed. This can be made in two ways, by setting a scheduler available in the Device Service itself or through the Scheduling Service. The second approach is more advisable, as will be seen in the performance discussion in Chapter 6.

The scheduling is configured for the Device Service, in the TOML file mentioned above. In there it is possible to define the frequency of the job and the resource accessed, when describing the connected device. It is also possible to set a flag indicating to read the resource on value change, although this was not explored in this dissertation. The Device Service configuration file, for this setup, is present in Listing A.2.

To configure the scheduler service one needs to first setup an interval. This registers a timer with a name and a given frequency. Then it is possible to associate this timer with a resource in what is called an interval action. This registers the timer with the corresponding device driver resource to be read, or command to be sent. This is configured directly on the scheduler service through its REST API, similarly to the other services. The JSONs sent to the Scheduler Service for configuration are shown Listings A.4 and A.5.

Data exportation setup

The data exportation configuration is similar to what was presented for the device service. For the setup here presented, the method for exporting data was through the MQTT protocol. This was setup through the Client Registry Service API, sending an HTTP POST with the information concerning the name of the topics to be published and the broker IP address. This information is sent in JSON format and put on the HTTP request body. Being a REST API, it allows remote configuration from processes outside of the gateway. This API supports reading the configurations (GET request), add new a client (POST request) and deleting clients (DELETE request). Export Service configuration JSON can be seen in Listing A.6.

5.3.3 InfluxDB

InfluxDB was the prefered database to be deployed in the system at issue. It was deployed in a Docker container, in the same computer as the Modbus server, the MQTT broker and the block interconnection. The database was created through InfluxDB Command Line Interface (CLI). After creation it is ready to accept information. This is sent by Node-Red, after being processed into the correct data structure, seen in Table 5.9. This information is associated with a Measurement in the database, with the device name being a tag and the timestamp the registered unix epoch time, inserted when the measurement was taken on EdgeX.

The communication with Influx is made through its HTTP interface. This allows to write queries in InfluxQL (similar to a structured query language) to interact with the database.

Timestamp	Device name	Value

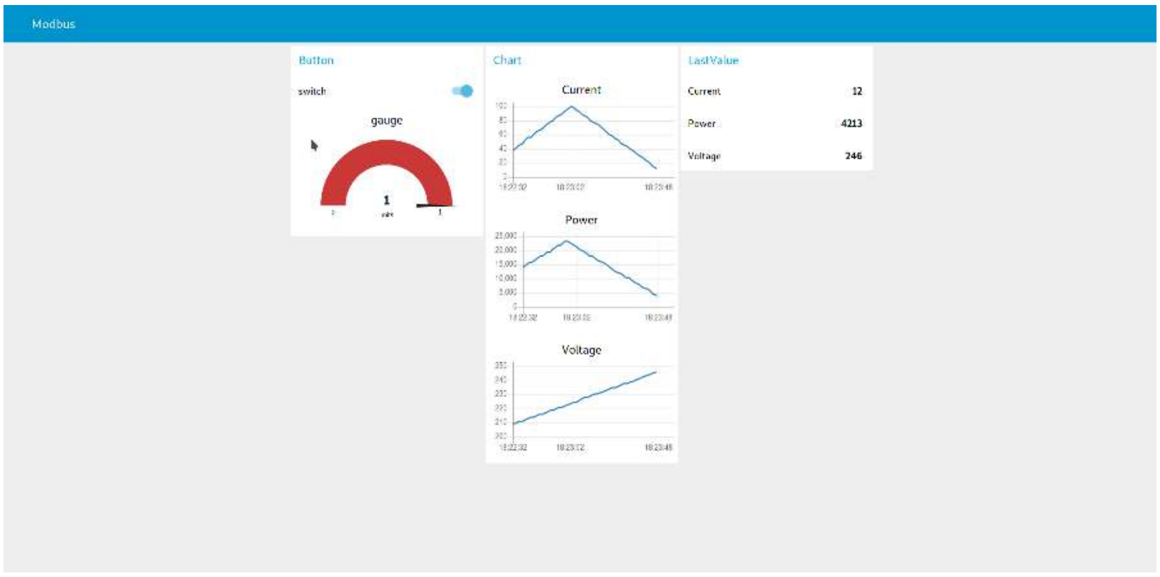
Table 5.9: Influx data scheme

5.3.4 Block interconnection

Node-Red was also used to interconnect the gateway with the database. Data from all devices is exported from EdgeX through MQTT in a single topic. A program was made to process the data arriving from EdgeX and insert it into the correct format for the database. The insertion on the database was made through an InfluxDB node that accepts the data to be injected in JSON form [42]. The information injected is associated with a database session and a measurement on the side of Influx.

5.3.5 Dashboard

A dashboard was created in order to display the information being captured from EdgeX and interact with the connected virtual device. The dashboard, pictured in Figure 5.3, presents the information read from the Modbus server discussed earlier. It is divided into three columns. The left one is consituted by a button with an associated gauge. This is used to simulate an actuator, and its response. So, when the button state changes, it sends a message reporting it to EdgeX, which inserts the value read onto the Modbus device. The one in the middle has time graphs displaying the values that are being collected. Finally the right column presents the last value read for each measurement. All the values displayed, are periodically read and updated on the dashboard. Figure A.1 depicts the Node-Red flow for data processing and dashboard.



Current	12
Power	4213
Voltage	246

Figure 5.3: Node-Red based dashboard

5.4 Summary

This chapter presented the system as it was implemented in practice. It begins by showing the block diagram of the system, indicating the two approaches made for the system evaluation. It then followed to a discussion comparing the possible solutions found for the gateway and the database.

Four frameworks were evaluated, EdgeX Foundry, Eclipse Kura, macchina.io and Node-Red. These were mainly compared through their feature set, from which the relevant terms were the supported protocols, the availability of tools to integrate non-supported device drivers, data processing mechanisms, connection to external endpoints, presence of an embedded database and remote management and maintenance capabilities. From the four, EdgeX was chosen as the most featureful solution, due to its more complete out-of-the-box protocol support and existance of a Device Service SDK, facilitating the addition of unsupported protocols to the framework.

As for the databases, three representatives of three different databases types, namely SQL, NoSQL and Time series, were initially selected. This sample was reduced to three, one for each category, by ranking them by their popularity index, measured through the number of site hits from a Google search engine query. MySQL, MongoDB and InfluxDB were selected and directly compared in terms of feature set, namely the kind of data model, query language, retention policies and high availability. The chosen platform was InfluxDB as it fits nicely with the requirements presented in Chapter 4. Furthermore, a performance comparison was made through two articles, one comparing Mongo and MySQL and the second comparing Cassandra, MongoDB and Influx. The results pointed to Influx having better performance than MongoDB, and thus better performance than MySQL. So, InfluxDB was chosen as the centralized DBMS for the rest of the work.

Next, the technologies used, along with their setup and deployment, were described. The first to be discussed was EdgeX, where its microservice architecture was presented, showing the job done by each service and how they interact with each other. Then, the configurations made to the Device, Scheduler and Export services were presented. The same approach was followed for the InfluxDB description, where the data scheme was presented, as well as for the block interconnection, which bridges the gateway and the database by processing the information exported by EdgeX into the required Influx data structure. Finaly the system dashboard, made with Node-Red, was shown indicating the elements present there, as well as how they were integrated with the system.

The following chapter presents the performance evaluation made to the EdgeX platform.

Chapter 6

Tests and Measurements

The performance evaluation focused mainly on the gateway framework EdgeX. This is the component that can have the most impact on the overall data throughput of the system, limiting the possible aplications for the platform here discussed. The metrics evaluated have to do with the reading scheduling capabilities available on the framework. As the chosen protocol for the demonstrator was Modbus/TCP, it is important to evaluate how the system manages the scheduling of reading requests. This protocol defines a Client-Server relationship between the Modbus device, the server, and the client, in this case, the EdgeX Device Service. To retrieve information from the server, the client must periodically request a reading. The performance evaluation of the request scheduling gives enough insight to make decisions on possible applications for this platform. A poor performance for this metric can invalidate its use for hard or even soft real-time applications. Latency is also something that needs to be taken into consideration. In this case, it measures the time it took for the collected information to reach the database. This metric can be used to evaluate situations were response time from the system is expectably low. As a simple example, it measures the time the information produced by a switch takes to be received in the database, evaluated by the management software and sent back to a lamp. If the time is too long, the activation of the lamp, when triggering the switch, can take enough time that the human eye stops perceiving this action as instantaneous.

This chapter is structured in the following way. First the test setups are introduced. This section discusses how the various elements of the system were deployed and interconnected. Then the tests carried are explained, from the metrics evaluated to how the information was retrieved. Following, the results from the test are depicted in a collection of graphs and an analysis is made for each. Wrapping up the chapter is a discussion on the overall results obtained, defining some of the bottlenecks and comments on which applications this system is not suited for.

6.1 The test setup

The setup used to test EdgeX was based on the structure introduced in the previous chapter. It is comprised by the Modbus device simulator, EdgeX as the gateway and an Influx database.

For the purpose of the test, the chosen device was a simplified version of the previously presented. It had only one measurement to be read, the Current, configured as before. The other platforms were configured acordingly, with Influx now storing two sets of time values, one marked by EdgeX when reading from the Modbus server, and another registered when the data is processed in Node-Red, right before insertion in the database. The Modbus server register was updated every second with an incremental value of one unit, alowing to quickly assess if data was loss.

Four tests were made in total, all with the system configured as above. What varied between the runs was the deployment pattern and the scheduling method used in EdgeX to retrieve data from the Modbus device. EdgeX allows two ways of registering timed events to collect data from a Device Service. One is through the Device Service itself, where the frequency of a task is registered on the device list of its configuration file. The second approach is through the scheduling service. This allows the definition of general purpose timers, which can then be associated with a command from any of the EdgeX's microservices. For all the tests run, the polling period was one second. Both of these approaches trigger a reading from the Device Service, which collects data, marks it and sends it to the Core Data microservice. This is then exported to the Export Service, which publishes the data to an MQTT topic.

As for the deployment schemes, there were three variations. The initial tests were run in a virtual machine, with two CPU cores and 1GB of the system resources allocated to it, pictured in Figure 6.1. The Operating System (OS) installed was Ubuntu 18.04.4. This machine run EdgeX through Docker and used the Device Service auto events feature to schedule device readings. The other tests, aggregated in Figure 6.2, were all run in a Raspberry Pi, model 3B, with Ubuntu Server 20.04 for ARM 64-bit CPUs. This version was selected, instead of the Raspbian distribution, or the same version OS as used in the virtual machine, because the Docker images available for EdgeX were only available for 64-bit OS versions, which these two Linux distributions did not have available.

The MQTT broker used, Mosquitto, as well as the Node-Red instance, which contains flows for the injection of data onto InfluxDB and for the Modbus server simulator, run in the same machine as the virtual machine. This machine has 8GB of RAM, and an hyper-threaded Intel i5 CPU with four cores, totaling eight threads.

Influx was configured in the same way as in Chapter 5. It run in a Docker container, and the only definition made was the creation of the database.

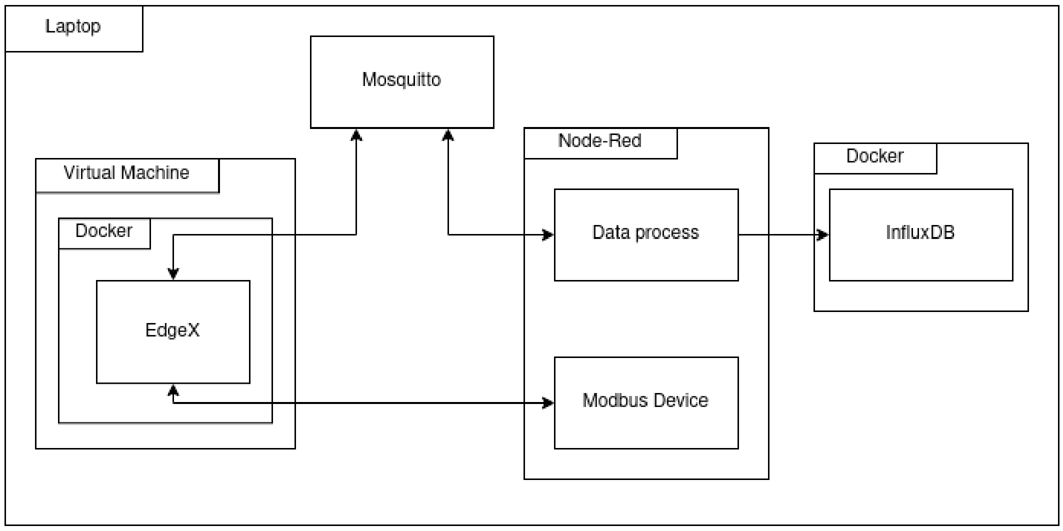


Figure 6.1: Test setup with EdgeX running on a Virtual Machine

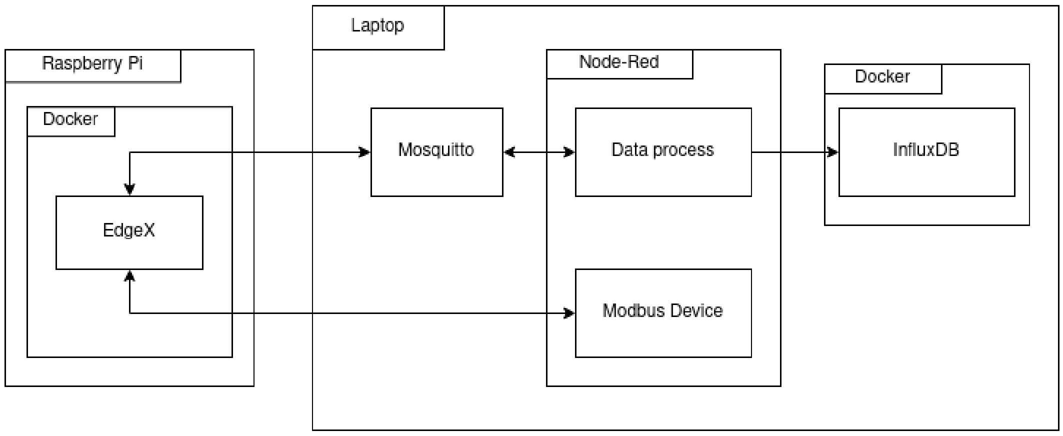


Figure 6.2: Test setup with EdgeX running on a Raspberry Pi

6.2 The evaluated metrics

The tests run concern the evaluation of EdgeX's performance. To fulfill this, two sets of tests were performed. One evaluates the possible jitter associated with a scheduled task and the other analyses the latency characteristics of the framework. To better understand what was measured and how to retrieve this information, a sequence diagram demonstrating the flow of information for one reading request is depicted in Figure 6.3. The first action is the trigger from the EdgeX scheduler, be it the Device Service Auto-Event or the Scheduler Service. This trigger activates a read request to the Modbus device, which responds accordingly, with a message containing a reading from the requested register. When this message arrives at EdgeX, a timestamp is added, registering the time of the reading. This message is then sent to the Core Data Service, which is exported through the Export Service, publishing the information to the topic "EdgeXDataTopic". The data-processing functionality having already subscribed to the same topic, receives the message, processes it and injects it onto the database. At the end of the processing pipeline another timestamp is added. This gives the possibility to

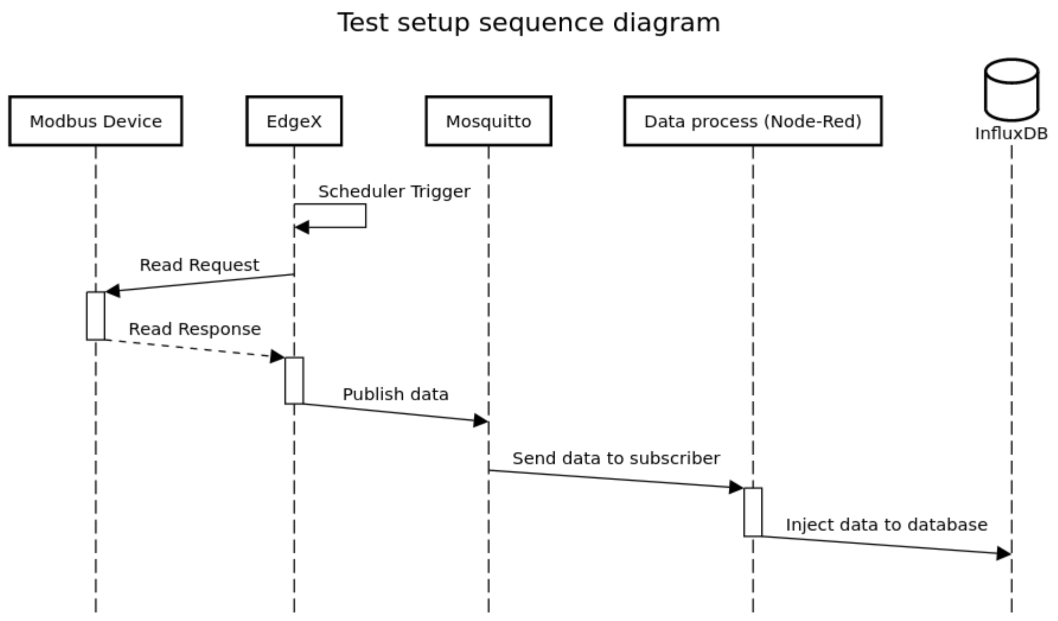


Figure 6.3: Sequence diagram of an EdgeX automatic reading

measure the latency between the reading of the Modbus register and the insertion on the database. The data injected onto the database is also timestamped by InfluxDB. From here on out, the difference between this last timestamp and the one registered by EdgeX is used for the latency analysis. The jitter and periodicity analysis was made using the timestamp provided by EdgeX.

6.3 How the information was retrieved

Matlab was used to facilitate the processing of the information collected in these tests. The data was retrieved directly from Influx, through its HTTP interface. Wireshark was used to monitorise the packets retrieved by EdgeX. This took advantage of the fact that for each measurement read, the Modbus/TCP connection is restarted, meaning that a session is open each time a reading is requested to the server. This allowed to quickly retrieve the number of readings requested, to assure that packets were not being lost.

The data structures stored in the Database are presented in Tables 6.1 and 6.2. The first corresponds to the data being updated in the Modbus server. This was used to set a baseline performance, to which EdgeX should correspond. The second is the information inserted onto Influx, read by EdgeX and processed on Node-Red. These structures coexist on a single database instance, on different measurements. For the Modbus device update structure, the Influx timestamp is the time value registered when a sample is inserted in the database, while the Node-Red timestamp is registered after a new value is generated, before being inserted in the database. For EdgeX readings the approach was similar. The Influx timestamp is collected when a sample is stored in the database. The EdgeX timestamp is marked in the Device Service, right after the connection with the Modbus server. Finally, the Node-Red

Modbus device updates			
Influx timestamp	Value updated	Node-Red timestamp	Sensor name

Table 6.1: Influx data structure for Modbus device updates

EdgeX readings				
Influx timestamp	EdgeX timestamp	Sensor name	Node-Red timestamp	Value read

Table 6.2: Influx data structure for EdgeX readings

timestamp is added to the structure at the end of the processing pipeline, right before data is sent to the database.

The data stored in the database was retrieved through Matlab, which connected to InfluxDB through its HTTP interface and InfluxQL query language. To fetch the period between the samples and the latency, the information received was processed and presented in plot and histogram forms, depicted in the following sections. The dataset used for the period analysis was retrieved by backwards difference calculation with all the sampled data. The Node-Red timestamps were used as a control, to set a baseline for the test. The EdgeX analysis used the timestamps registered when the Modbus device was read, at the Device Service. For the latency analysis, a difference was made between the timestamps registered in the database and the ones registered at the Device Service, thus allowing an aproximate evaluation of the time it takes for the data generated by EdgeX to arrive.

6.4 The results

The results presented in the following sections correspond to the data retrieved from four different experiences. The differences between them concern the method for task scheduling on EdgeX and the deployment scheme. The tests run were the following:

- EdgeX with read request configured on Device Service (Auto-events), running on Docker in a Virtual Machine
- EdgeX with read request configured on Device Service (Auto-events), running on Docker in a Raspberry Pi
- EdgeX with read request configured on Scheduler Service, running on Docker in a Raspberry Pi
- EdgeX with read request configured on Scheduler Service, running natively in a Raspberry Pi

These are divided into two sections, the first being concerned with the Auto-Events tests and the other with the Scheduler Services. From these tests, further metrics were also retrieved. The first is the Control measurement . This sets up a baseline and presents the periodicity, and its consistency, with which the data is being updated in the Modbus device. The second is the latency measured from the instant the data is captured in EdgeX's Device Service to the moment the data is injected in the database.

Each test was run three times. The images in this chapter correspond to only one run. The other results are present in Appendix B.

6.4.1 Control

The first graphs, presented in Figures 6.4 and 6.5, correspond to a baseline evaluation of the values updated on the Modbus device register. The first figure corresponds to a plot of the difference between samples. This was retrieved through the timestamps associated to the collected measurements. Table 6.3 has properties of the data presented. From this, it is clear to see that the Modbus register is updated in average with 1.001s (σ = 0.002s) between updates, with a jitter in the order of 0.008s. The variance is of $2\mu s$. The dataset in question was measured when EdgeX performance was evaluted with the Device Service configured for Auto-Events, deployed on Ubuntu in a Virtual Box, through Docker. Other measures were taken, with the results present in the Anexes.

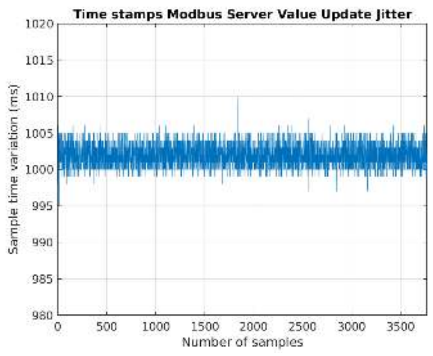


Figure 6.4: Time difference between Modbus register update

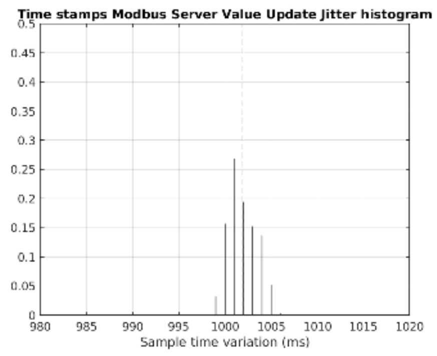


Figure 6.5: Histogram of time difference between Modbus register update

Average (ms)	1001.911
Standard Deviation (ms)	1.575
Variance (ms^2)	2.480
Jitter Interval (ms)	$[-6.911, 8.089]$

Table 6.3: Control dataset properties

6.4.2 Scheduling reading requests through Device Service auto-events

The first experience made was with EdgeX's Modbus Device Service configured with Auto-Events enabled, with a polling period of $1s$. The data collected is presented in a cartesian plot in Figure 6.6 and an histogram in Figure 6.7, with the timestamps collected processed through a backward finite difference function. This allows the evaluation of the jitter between read requests. From the histogram, for which its properties are presented in Table 6.4, it is seen that it is far from the gaussian curve expected for this kind of functionality. This test was executed on a Virtual Machine, which was thought of as the element that could provoke this unwanted behaviour.

To verify this, the same test was executed on physical hardware. This attempted to eliminate the possibly existing overhead due to the hardware virtualization process of an hypervisor. The results are presented in Figures 6.8 and 6.13, with its statistical properties

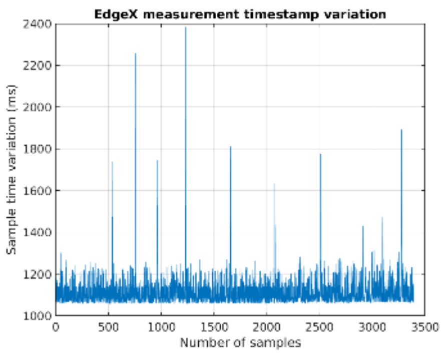


Figure 6.6: Time difference between Modbus readings (EdgeX configured for Device Service auto-events on Virtual Machine)

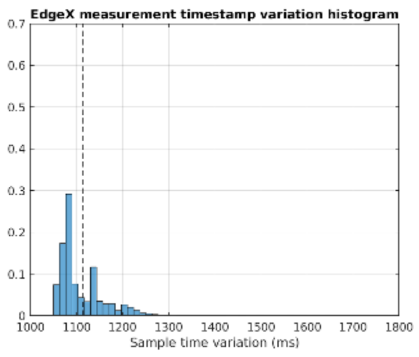


Figure 6.7: Histogram of time difference between Modbus readings (EdgeX configured for Device Service auto-events on Virtual Machine)

Average (ms)	1113.54
Standard Deviation (ms)	70.23
Variance (ms^2)	4932.75
Jitter Interval (ms)	$[-57.76, 1271.32]$

Table 6.4: Auto-events VM dataset properties

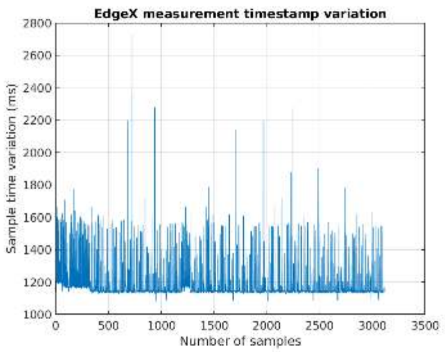


Figure 6.8: Time difference between Modbus readings (EdgeX configured for Device Service auto-events on Raspberry Pi through Docker)

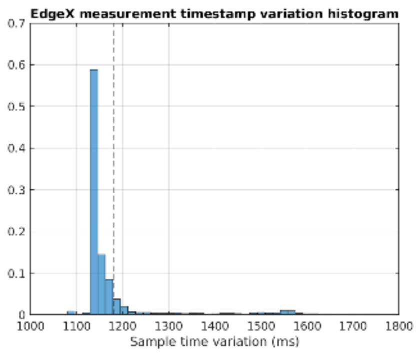


Figure 6.9: Histogram of time difference between Modbus readings (EdgeX configured for Device Service auto-events on Raspberry Pi through Docker)

presented in Table 6.5. Although the expectation was to have a more well behaved histogram, aproaching a gaussian distribution, the results obtained matched closely with the ones presented previously, with a slight penalty in terms of jitter. This means that this problem has its origin in the software and not on the deployment infrastructure. This was confirmed after some resarch made on EdgeX Foundry Github repository. In issue [43] is mentioned that the Auto-Events functionality waits for the execution of the reading request to end, adding this time to the wait period and thus increasing the total response time of the task execution. This mechanism justifies the results obtained in these experiments.

Average (ms)	1180.26
Standard Deviation (ms)	114.31
Variance (ms^2)	13066.89
Jitter Interval (ms)	$[-99.87, 1546.33]$

Table 6.5: Auto-events RPi dataset properties

6.4.3 Scheduling reading requests through Scheduler Service

As mentioned in the previous section, configuring the reading task schedule through the Auto-Events functionality present in the Device Service lead to underperformance. The resulting distribution obtained, presented on a histogram, was far from the gaussian like distribution expected. As such, there was the need to evaluate a different approach for this. The Scheduler Service was configured with the same characteristics as the previous experiences, with a $1s$ periodicity between reading requests. The data captured in Influx is presented in the form of a plot in Figures 6.10 and 6.12 while histograms are pictured in Figures 6.11 and 6.13. The statistical properties of the datasets used are indicated in Tables 6.6 and 6.8. The first pair of graphs are the results for the experience run for EdgeX deployed through Docker, while the second one corresponds to the experience with EdgeX running directly on the OS. Both of these were run on the same Raspberry Pi, with Ubuntu 20.04 Server as the chosen OS.

It is clear to see that both results approach the expected gaussian distribution. Most of the readings made, arrived with a $1s$ average period ($\sigma = 0.07s$), matching the configured period. On the other hand, some packets arrived with high jitter values of almost 50%, more or less, than the targeted period. Comparing the two deployment schemes, it is possible to see that, altough having overall a similar outcome, EdgeX running directly on the OS had better results than the deployment through Docker. This is expected, as Docker adds an extra layer of virtualization to the system, which, as seen, negatively impacts the performance of the various microservices.

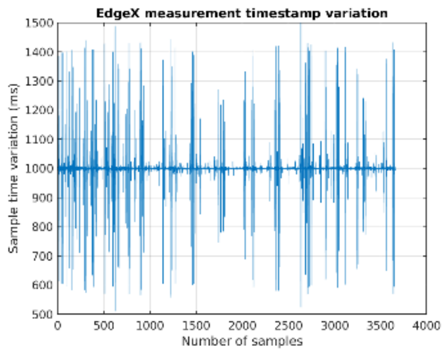


Figure 6.10: Time difference between Modbus readings (running directly on the OS on Raspberry Pi)

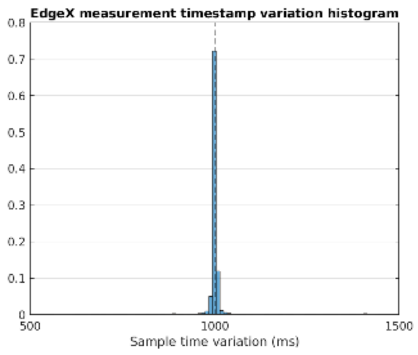


Figure 6.11: Histogram of time difference between Modbus readings (running directly on the OS on Raspberry Pi)

Average (ms)	999.99
Standard Deviation (ms)	73.56
Variance (ms^2)	5410.93
Jitter Interval (ms)	$[-489.37, 499.05]$

Table 6.6: Scheduler RPi baremetal dataset properties

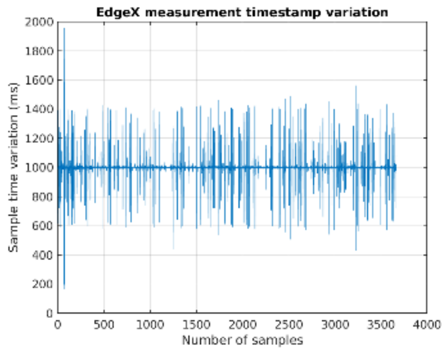


Figure 6.12: Time difference between Modbus readings (EdgeX deployed through Docker on Raspberry Pi)

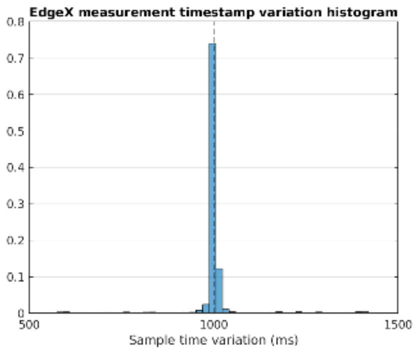


Figure 6.13: Histogram of time difference between Modbus readings (EdgeX deployed through Docker on Raspberry Pi)

Average (ms)	1000.02
Standard Deviation (ms)	90.65
Variance (ms^2)	8217.44
Jitter Interval (ms)	$[-835.13, 955.23]$

Table 6.7: EdgeX RPi Docker dataset properties

6.4.4 Latency

The latency test was run on the same setup as the previous experiences. This evaluated the time it took for information to be transferred from the EdgeX Device Service to the Database. Figures 6.14 and 6.15 present the results for the tests run with EdgeX running directly on the OS and through Docker respectively. The information is presented in a Histogram, with the statistical properties of the dataset for both runs present in Table 6.8.

As expected, the results for EdgeX running on top of the OS were better than the ones for the Docker deployments, although by an 8% margin. This shows that, for this metric, Docker does not have an impact as big as for the period variation and jitter. The average values for latency of approximalty $0.07s$ and $0.08s$ are enough for some applications, such as turning on a light, but more time constrained, and performant, control loops would be difficult to implement.

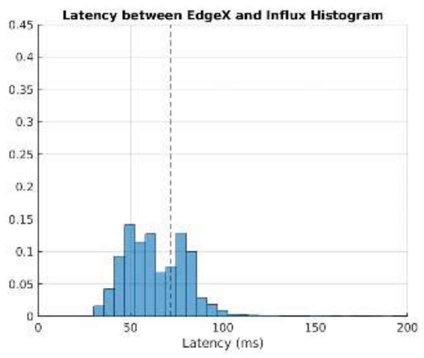


Figure 6.14: Latency from EdgeX to InfluxDB (running directly on the OS on Raspberry Pi)

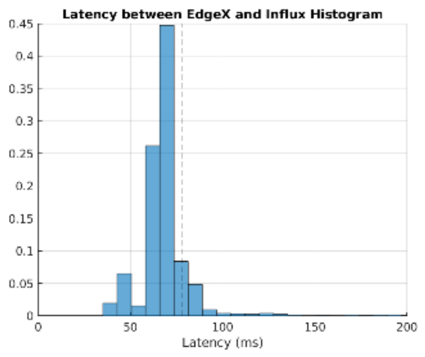


Figure 6.15: Latency from EdgeX to InfluxDB (EdgeX deployed through Docker on Raspberry Pi)

Metric	Baremetal	Docker
Average (ms)	71.46	78.23
Standard Deviation (ms)	52.94	60.97
Variance (ms^2)	2802.39	3717.56

Table 6.8: Latency test dataset propreties

6.5 Conclusions

The EdgeX Foundry platform chosen to be the gateway for the BAS had its performance evaluated in this chapter. This begun by exposing the tests made, the periodicity of a scheduled task and the latency. From these tests, it was possible to retrieve information on jitter characteristics of the framework, its performance and adequacy to be integrated in future real-world deployments. These tests were run for four setups, two with the Modbus device reading task scheduled by the Device Service and the other two by the Scheduler Service. Furthermore, EdgeX was deployed in three ways, two running through Docker in a Virtual Machine and on a Raspberry Pi and another running natively on the OS of the Raspberry Pi. The tests to evaluate the performance of the platform in terms of read request jitter were run for four setups, two with Auto-Events, on Virtual Machine and Raspberry Pi and the other two in the Raspberry Pi with EdgeX running through Docker and natively. Latency tests were only run for the Raspberry Pi setups with the task scheduling handled by the Scheduler Service.

The first tests, with the Device Service configured for Auto-Events with 1s period between read requests, resulted in performance metrics far from the expected. With an average of 1.11s for the Virtual Machine deployment and 1.18s for the Raspberry Pi, it distances by more than 100ms from the target 1s. Furthermore, the histograms for the distribution of time period between samples, show a behaviour quite different from what was expected, which is a behaviour more similar to that of a gaussian distribution. This may be do to the fact that the scheduler waits for the defined task period and for the completion of the task. Therefore, the total time between reads would be the period plus the time it took to retrieve the information.

This will always push the period over the expected 1s and justifies the results obtained.

To try and overcome this issue, the framework was also tested with the Scheduler Service configured to call the read request task every second. Both of the experiences made had EdgeX running on a Rasbperry Pi, either through Docker or directly on the OS. The analysis of the metrics painted out a picture different than the one presented beforehand. These results closely align with the expectations: the histogram is similar to a gaussian distribution and the averages are on point with the 1s target. As such, the implementation is different than before, and seems that there is a separate task triggering more consistenly the calls to the read requests, without waiting for it to finish its job. Nonetheless, the results revealed a high value of jitter, with the test where EdgeX run directly on the OS having a lower value than the deployment on Docker. The first part indicates that this platform, for the setups here presented, does not have enough performance to run tasks at a low rate of 1Hz. The jitter figures drift from the average value, at max, 140% for Docker and 86% for baremetal. With values as high as these some measurements were not read while others were read twice, depending if the jitter is positive or negative. In addition to this, these results were not very consistent, making this platform not suited for high frequency operation. Therefore, time critical applications should not be considered for this framework, as the period cannot be guaranteed.

The next chapter sums up the research and work done throughout this dissertation. It tries to give some advise when chosing a platform for BAS and presents some conclusions about the platforms evaluated. It also discusses some of the possible future work to continue this research.

Chapter 7

Conclusion

In this dissertation, an architecture for a Building Management System was selected, along with a selection of core technologies to support it. The architecture chosen was a three layered hierarchical structure with the field devices at the lowest layer, a middle layer with a set of gateways and a management layer comprised of a dashboard and a centralized Database. EdgeX Foundry was the software framework selected for the gateway, while InfluxDB was used for the database and Node-Red to build a Dashboard. Tests were run on a use case with a simulated Modbus device, specifically targeted at evaluating scheduling capabilites of the gateway platform, as well as its latency.

An evaluation was made of a set of technologies for the database, the dashboard and the gateway. The chosen software was Influx for the database, EdgeX for gateway and Node-Red as the dashboard, as stated before. The database was compared with other two, different DBMSs. They were MySQL, a relational database, and MongoDB, a document-oriented, NoSQL database. Influx was selected due to being built specifically to manage time series based data structures. This is the core of the information present in a BAS. As such, this data management system was the most adequate for the developed system, having features such as retention policies to efficiently manage storage space, as well as being the most performant of the three, for the workload here demonstrated.

EdgeX was the selected framework for the gateway system. This element needs to aggregate different Industrial and Building Automation protocols into a unified notation. This platform brought more advantage for this task than the evaluated competitors. It allows multiple device drivers to be used simultaneously, while also giving tools to quickly support different protocols, thus giving more flexibility on system deployment and support for a mix of legacy and modern field device solutions.

This platform was also tested for two performance metrics, task scheduling capabilities and latency. The first evaluated the periodicity with which EdgeX could be programmed to retrieve data from, in this case, a simulated Modbus device. Two possibilities were tested, one with the scheduling made on the Device Service and the other on the Scheduler Service. It was the latter that offered the better performance. The results showed its periodicity was closer to the expected gaussian distribution, although the jitter obtained was quite high. This indicates that the platform is not yet prepared to handle the typical time critical applications present in Industrial Automation. It works better for applications with high polling periods, that do not need a low reactivity time, due to the latency values observed. It is also relevant to point out that, altough small, there was an advantage in running EdgeX directly on the operating system, instead of in a set of Docker containers, as the jitter was lower. But this could be overturned with more capable hardware.

7.1 Future Work

The work presented in this document, allowed an insight on the upcoming technologies that bridge the new IoT based devices with old, legacy equipment and agglomerates them in a modern, versatile Building Management System. More indepth research can be made, in order to continue and improve this system. The following list presents some of the thematics which should be considered when continuing this work:

1. Evaluate other dashboard software frameworks
2. Upgrade EdgeX to the most recent version, which substitutes the Export Services with the Application Services
3. Add support for more industrial and building automation protocols to EdgeX
4. Test more thoroughly the gateway platform, especially for setups with higher loads
5. Evaluate the platform in terms of its security features

Appendices

Appendix A

Configuration files and software developed for the system

A.1 EdgeX configurations

A.1.1 Docker compose file

Listing A.1: docker-compose.yaml

```
# /*******************************************************************************
# * Copyright 2018 Dell Inc.
# *
# * Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except
# * in compliance with the License. You may obtain a copy of the License at
# *
# * http://www.apache.org/licenses/LICENSE-2.0
# *
# * Unless required by applicable law or agreed to in writing, software distributed under the License
# * is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
# * or implied. See the License for the specific language governing permissions and limitations under
# * the License.
# *
# * @author: Jim White, Dell
# * EdgeX Foundry, Fuji, version 1.1.0
# * added: Jun 30, 2019
# *******************************************************************************/

# NOTE: this Docker Compose file does not contain the security services - namely the API Gateway and Secret Store

version: '3.4'

# all common shared environment variables defined here:
x-common-env-variables: &common-variables
  EDGEX_SECURITY_SECRET_STORE: "false"
  edgex_registry: consul://edgex-core-consul:8500
  Clients_CoreData_Host: edgex-core-data
  Clients_Logging_Host: edgex-support-logging
  Logging_EnableRemote: "true"

volumes:
  db-data:
  log-data:
  consul-config:
  consul-data:
  portainer_data:

services:
```

```
volume:
  image: edgexfoundry/docker-edgex-volume:1.1.0
  container_name: edgex-files
  networks:
    - edgex-network
  volumes:
    - db-data:/data
    - log-data:/edgex/logs
    - consul-config:/consul/config
    - consul-data:/consul/data

consul:
  image: consul:1.7
  ports:
    - "8400:8400"
    - "8500:8500"
  container_name: edgex-core-consul
  hostname: edgex-core-consul
  networks:
    edgex-network:
      aliases:
          - edgex-core-consul
  volumes:
    - log-data:/edgex/logs
    - consul-config:/consul/config
    - consul-data:/consul/data
  depends_on:
    - volume

config-seed:
  image: edgexfoundry/docker-core-config-seed-go:1.1.0
  command: ["--profile=docker", "--cmd=/edgex/cmd-redis"]
  container_name: edgex-config-seed
  hostname: edgex-core-config-seed
  networks:
    edgex-network:
      aliases:
          - edgex-core-config-seed
  environment:
    <<: *common-variables
  volumes:
    - log-data:/edgex/logs
    - consul-config:/consul/config
    - consul-data:/consul/data
  depends_on:
    - volume
    - consul

redis:
  image: redis:5.0.5-alpine
  ports:
    - "6379:6379"
  container_name: edgex-redis
  hostname: edgex-redis
  networks:
    - edgex-network
  volumes:
    - db-data:/data
  depends_on:
    - volume

logging:
  image: edgexfoundry/docker-support-logging-go:1.1.0
  ports:
    - "48061:48061"
  container_name: edgex-support-logging
  hostname: edgex-support-logging
  networks:
    - edgex-network
  environment:
    <<: *common−variables
  volumes:
    − log−data:/edgex/logs
    − consul−config:/consul/config
    − consul−data:/consul/data
  depends_on:
    − config−seed
    − volume

system:
  image: edgexfoundry/docker−sys−mgmt−agent−go:1.1.0
  ports:
    − "48090:48090"
  container_name: edgex−sys−mgmt−agent
  hostname: edgex−sys−mgmt−agent
  networks:
    − edgex−network
  environment:
    <<: *common−variables
  volumes:
    − db−data:/data/db
    − log−data:/edgex/logs
    − consul−config:/consul/config
    − consul−data:/consul/data
    − /var/run/docker.sock:/var/run/docker.sock
  depends_on:
    − logging

notifications:
  image: edgexfoundry/docker−support−notifications−go:1.1.0
  ports:
    − "48060:48060"
  container_name: edgex−support−notifications
  hostname: edgex−support−notifications
  networks:
    − edgex−network
  environment:
    <<: *common−variables
  volumes:
    − log−data:/edgex/logs
    − consul−config:/consul/config
    − consul−data:/consul/data
  depends_on:
    − logging
    − redis

metadata:
  image: edgexfoundry/docker−core−metadata−go:1.1.0
  ports:
    − "48081:48081"
  container_name: edgex−core−metadata
  hostname: edgex−core−metadata
  networks:
    − edgex−network
  environment:
    <<: *common−variables
  volumes:
    − log−data:/edgex/logs
    − consul−config:/consul/config
    − consul−data:/consul/data
  depends_on:
    − logging
    − redis

data:
  image: edgexfoundry/docker−core−data−go:1.1.0
  ports:
    − "48080:48080"
```

```
      - "5563:5563"
    container_name: edgex-core-data
    hostname: edgex-core-data
    networks:
      - edgex-network
    environment:
      <<: *common-variables
    volumes:
      - log-data:/edgex/logs
      - consul-config:/consul/config
      - consul-data:/consul/data
    depends_on:
      - logging
      - redis

  command:
    image: edgexfoundry/docker-core-command-go:1.1.0
    ports:
      - "48082:48082"
    container_name: edgex-core-command
    hostname: edgex-core-command
    networks:
      - edgex-network
    environment:
      <<: *common-variables
    volumes:
      - log-data:/edgex/logs
      - consul-config:/consul/config
      - consul-data:/consul/data
    depends_on:
      - metadata

  scheduler:
    image: edgexfoundry/docker-support-scheduler-go:1.1.0
    ports:
      - "48085:48085"
    container_name: edgex-support-scheduler
    hostname: edgex-support-scheduler
    networks:
      - edgex-network
    environment:
      <<: *common-variables
    volumes:
      - log-data:/edgex/logs
      - consul-config:/consul/config
      - consul-data:/consul/data
    depends_on:
      - metadata
      - redis

  app-service-rules:
    image: edgexfoundry/docker-app-service-configurable:1.0.0
    ports:
      - "48100:48100"
    container_name: edgex-app-service-configurable-rules
    hostname: edgex-app-service-configurable-rules
    networks:
      edgex-network:
        aliases:
          - edgex-app-service-configurable-rules
    environment:
      <<: *common-variables
      edgex_service: http://edgex-app-service-configurable-rules:48100
      edgex_profile: rules-engine
      Service_Host: edgex-app-service-configurable-rules
      MessageBus_SubscribeHost_Host: edgex-core-data
    depends_on:
      - consul
      - logging
      - data

  # rulesengine:
  # image: edgexfoundry/docker-support-rulesengine:1.1.0
  # ports:
  # - "48075:48075"
  # container_name: edgex-support-rulesengine
  # hostname: edgex-support-rulesengine
  # networks:
  # edgex-network:
  # aliases:
  # - edgex-support-rulesengine
  # depends_on:
  # - app-service-rules

  export-client:
    image: edgexfoundry/docker-export-client-go:1.1.0
    ports:
      - "48071:48071"
    container_name: edgex-export-client
    hostname: edgex-export-client
    networks:
      - edgex-network
    environment:
      <<: *common-variables
    volumes:
      - db-data:/data/db
      - log-data:/edgex/logs
      - consul-config:/consul/config
      - consul-data:/consul/data
    depends_on:
      - data

  export-distro:
    image: edgexfoundry/docker-export-distro-go:1.1.0
    ports:
      - "48070:48070"
      - "5566:5566"
      - "1883:1883"
    container_name: edgex-export-distro
    hostname: edgex-export-distro
    networks:
      - edgex-network
    volumes:
      - db-data:/data/db
      - log-data:/edgex/logs
      - consul-config:/consul/config
      - consul-data:/consul/data
    depends_on:
      - export-client
    environment:
      <<: *common-variables
      EXPORT_DISTRO_CLIENT_HOST: export-client
      EXPORT_DISTRO_DATA_HOST: edgex-core-data
      EXPORT_DISTRO_CONSUL_HOST: dgex-config-seed
      EXPORT_DISTRO_MQTTS_CERT_FILE: none
      EXPORT_DISTRO_MQTTS_KEY_FILE: none

#################################################################
# Device Services
#################################################################

  # device-virtual:
  # image: edgexfoundry/docker-device-virtual-go:1.1.1
  # ports:
  # - "49990:49990"
  # container_name: edgex-device-virtual
  # hostname: edgex-device-virtual
  # networks:
```

```
# edgex−network:
# aliases:
# − edgex−device−virtual
# volumes:
# − db−data:/data/db
# − log−data:/edgex/logs
# − consul−config:/consul/config
# − consul−data:/consul/data
# depends_on:
# − data
# − command

# device−random:
# image: edgexfoundry/docker−device−random−go:1.1.1
# ports:
# − "49988:49988"
# container_name: edgex−device−random
# hostname: edgex−device−random
# networks:
# − edgex−network
# volumes:
# − db−data:/data/db
# − log−data:/edgex/logs
# − consul−config:/consul/config
# − consul−data:/consul/data
# depends_on:
# − data
# − command
#
# device−mqtt:
# image: edgexfoundry/docker−device−mqtt−go:1.1.1
# ports:
# − "49982:49982"
# container_name: edgex−device−mqtt
# hostname: edgex−device−mqtt
# networks:
# − edgex−network
# volumes:
# − db−data:/data/db
# − log−data:/edgex/logs
# − consul−config:/consul/config
# − consul−data:/consul/data
# depends_on:
# − data
# − command

device−modbus:
  image: edgexfoundry/docker−device−modbus−go:1.1.1
  ports:
    − "49991:49991"
    − "10502:10502"
  container_name: edgex−device−modbus
  hostname: edgex−device−modbus
  networks:
    edgex−network:
      aliases:
        − edgex−device−modbus
  privileged: true
  volumes:
    − db−data:/data/db
    − log−data:/edgex/logs
    − consul−config:/consul/config
    − consul−data:/consul/data
    − /home/jferreira/EDGEX_GIT/edgex_docker/modbus:/custom−config
  depends_on:
    − data
    − command
  entrypoint:
    − /device−modbus
```

```
      − −−registry=consul://edgex−core−consul:8500
      − −−confdir=/custom−config
  #
  # device−snmp:
  # image: edgexfoundry/docker−device−snmp−go:1.1.1
  # ports:
  # − "49993:49993"
  # container_name: edgex−device−snmp
  # hostname: edgex−device−snmp
  # networks:
  # − edgex−network
  # volumes:
  # − db−data:/data/db
  # − log−data:/edgex/logs
  # − consul−config:/consul/config
  # − consul−data:/consul/data
  # depends_on:
  # − data
  # − command

#################################################################
# UIs
#################################################################
  # ui:
  # image: edgexfoundry/docker−edgex−ui−go:1.1.0
  # ports:
  # − "4000:4000"
  # container_name: edgex−ui−go
  # hostname: edgex−ui−go
  # networks:
  # − edgex−network
  # volumes:
  # − db−data:/data/db
  # − log−data:/edgex/logs
  # − consul−config:/consul/config
  # − consul−data:/consul/data
  # depends_on:
  # − data
  # − command

#################################################################
# Tooling
#################################################################

  portainer:
    image: portainer/portainer
    ports:
      − "9000:9000"
    command: −H unix:///var/run/docker.sock
    volumes:
      − /var/run/docker.sock:/var/run/docker.sock
      − portainer_data:/data
    depends_on:
      − volume

networks:
  edgex−network:
    driver: "bridge"
```

A.1.2 Modbus Device Service configuration file

Listing A.2: configuration.toml

```
[Service]
Host = "edgex−device−modbus"
Port = 49991
ConnectRetries = 3
Labels = []
```

```
OpenMsg = "device modbus started"
Timeout = 5000
EnableAsyncReadings = true
AsyncBufferSize = 16

[Registry]
Host = "edgex-core-consul"
Port = 8500
CheckInterval = "10s"
FailLimit = 3
FailWaitTime = 10
Type = "consul"

[Logging]
EnableRemote = false
File = "/custom-config/device-Modbus.log"

[Writable]
LogLevel = "DEBUG"

[Clients]
  [Clients.Data]
  Name = "edgex-core-data"
  Protocol = "http"
  Host = "edgex-core-data"
  Port = 48080
  Timeout = 50000

  [Clients.Metadata]
  Name = "edgex-core-metadata"
  Protocol = "http"
  Host = "edgex-core-metadata"
  Port = 48081
  Timeout = 50000

  [Clients.Logging]
  Name = "edgex-support-logging"
  Protocol = "http"
  Host = "edgex-support-logging"
  Port = 48061

[Device]
  DataTransform = true
  InitCmd = ""
  InitCmdArgs = ""
  MaxCmdOps = 128
  MaxCmdValueLen = 256
  RemoveCmd = ""
  RemoveCmdArgs = ""
  ProfilesDir = "/custom-config"


# Pre-define Devices
[[DeviceList]]
  Name = "Modbus-TCP-test-device"
  Profile = "Network-Power-Meter"
  Description = "This device is a product for monitoring and controlling digital inputs and outputs over a LAN."
  labels = [ "Air conditioner","modbus TCP" ]
  [DeviceList.Protocols]
    [DeviceList.Protocols.modbus-tcp]
      Address = "192.168.100.1"
      Port = "10502"
      UnitID = "1"
  [[DeviceList.AutoEvents]]
    Frequency = "1s"
    OnChange = true
    Resource = "Readings"
```

A.1.3 Modbus device profile

Listing A.3: DENT.Mod.PS6037.profile.yaml

```
# DENT.Mod.PS6037.profile.yaml
name: "Network−Power−Meter"
manufacturer: "Dent Instruments"
model: "PS3037"
description: "Power Scout Meter"
labels:
  − "modbus"
  − "powerscout"
deviceResources:
    −
        name: "Current"
        description: "Current Reading"
        attributes:
            { primaryTable: "HOLDING_REGISTERS", startingAddress: "10" }
        properties:
            value:
                { type: "UINT16", readWrite: "R", size: "1", scale: "1", minimum: "0", maximum: "65535", defaultValue: "0" }
            units:
                { type: "String", readWrite: "R", defaultValue: "A" }
    −
        name: "Power"
        description: "Current Reading"
        attributes:
            { primaryTable: "HOLDING_REGISTERS", startingAddress: "4004" }
        properties:
            value:
                { type: "UINT16", readWrite: "R", size: "1", scale: "1", minimum: "0", maximum: "65535", defaultValue: "0" }
            units:
                { type: "String", readWrite: "R", defaultValue: "W" }
    −
        name: "Voltage"
        description: "Current Reading"
        attributes:
            { primaryTable: "HOLDING_REGISTERS", startingAddress: "4018" }
        properties:
            value:
                { type: "UINT16", readWrite: "R", size: "1", scale: "1", minimum: "0", maximum: "65535", defaultValue: "0" }
            units:
                { type: "String", readWrite: "R", defaultValue: "V" }
    −
        name: "Switch"
        description: "On/Off , 0−OFF 1−ON"
        attributes:
            { primaryTable: "COILS", startingAddress: "2" }
        properties:
            value:
                { type: "BOOL", readWrite: "RW", scale: "1", minimum: "0", maximum: "1", defaultValue: "0"}
            units:
                { type: "String", readWrite: "R", defaultValue: "On/Off"}


deviceCommands:
    −
        name: "Readings"
        set:
            − { index: "1", operation: "set", object: "Current" }
            − { index: "3", operation: "set", object: "Power" }
            − { index: "4", operation: "set", object: "Voltage" }
        get:
            − { index: "1", operation: "get", object: "Current" }
            − { index: "3", operation: "get", object: "Power" }
            − { index: "4", operation: "get", object: "Voltage" }
            − { index: "5", operation: "set", deviceResource: "Switch", mappings: { "true":"ON","false":"OFF"}}
    −
        name: "Switch"
        get:
            − { index: "1", operation: "set", deviceResource: "Switch", mappings: { "true":"ON","false":"OFF"}}
        set:
            − { index: "1", operation: "set", deviceResource: "Switch", mappings: { "OFF":"false","ON":"true"}}
coreCommands:
    −
        name: "Readings"
        get:
            path: "/api/v1/device/{deviceId}/Readings"
            responses:
                −
                    code: "200"
                    description: "Get Readings"
                    expectedValues: ["Current","Power","Voltage",Switch]
                −
                    code: "503"
                    description: "service unavailable"
                    expectedValues: []
        put:
            path: "/api/v1/device/{deviceId}/Readings"
            parameterNames: ["Current","Power","Voltage"]
            responses:
                −
                    code: "204"
                    description: "Set values"
                    expectedValues: []
                −
                    code: "503"
                    description: "service unavailable"
                    expectedValues: []
    −
        name: "Switch"
        get:
            path: "/api/v1/device/{deviceId}/Switch"
            responses:
                −
                    code: "200"
                    description: "Get the Switch"
                    expectedValues: ["Switch"]
                −
                    code: "500"
                    description: "internal server error"
                    expectedValues: []
        put:
            path: "/api/v1/device/{deviceId}/Switch"
            parameterNames: ["Switch"]
            responses:
            −
                code: "204"
                description: "Set the Switch"
                expectedValues: []
            −
                code: "500"
                description: "internal server error"
                expectedValues: []
```

A.1.4 Scheduler Service configuration (JSON)

Listing A.4: Timer definition

```
{
        "name":"timer1s",
        "start":"20200101T000000",
        "end":"",
        "frequency":"PT1S"
}
```

Listing A.5: Association of timer and event

```
{
        "name":"get−Modbus−Current",
        "interval":"timer1s",
        "target":"Modbus−TCP−test−device",
        "protocol":"http",
        "httpMethod":"GET",
        "address":"localhost",
        "port":49991,
        "path":"/api/v1/device/name/Modbus−TCP−test−device/Readings"
}
```

A.1.5 Export Service configuration (JSON)

Listing A.6: MQTT broker registration

```
{
    "name": "MQTTMosquitto",
    "addressable": {
        "name": "mqtt_mosqconnection",
        "protocol": "TCP",
        "address": "192.168.100.1",
        "port": 1883,
        "publisher": "EdgeXExportPublisher",
        "topic": "EdgeXDataTopic"
    },
    "format": "JSON",
    "enable": true,
    "destination": "MQTT_TOPIC"
}
```

A.2 Node-Red flows

A.2.1 Data processing

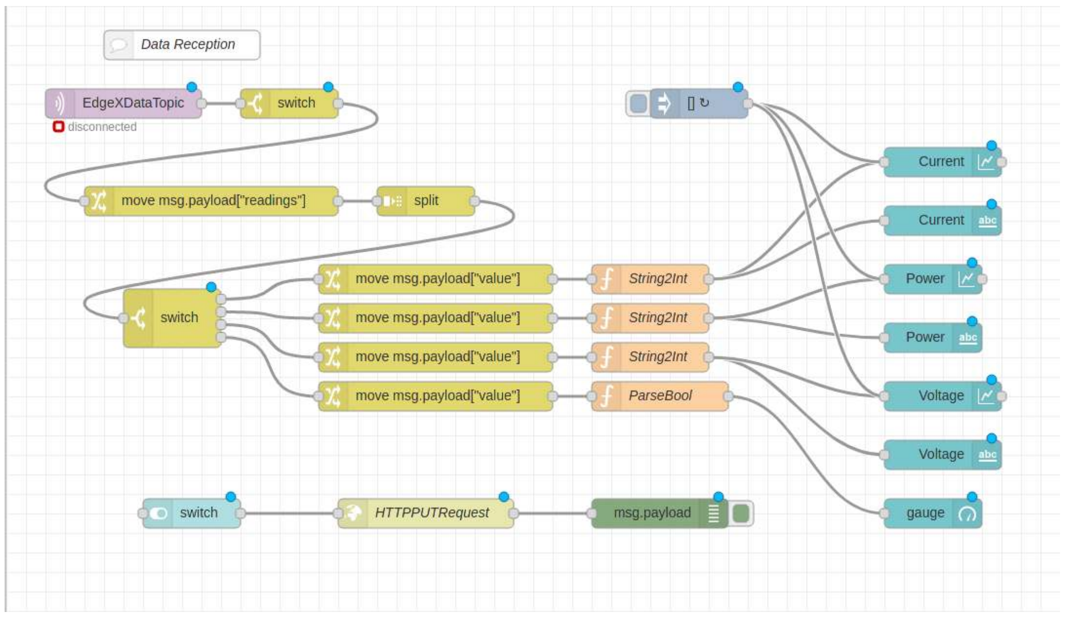


Figure A.1: Node-Red flow with the processing done to the data received form EdgeX and presentation on Dashboard

A.2.2 Modbus Server

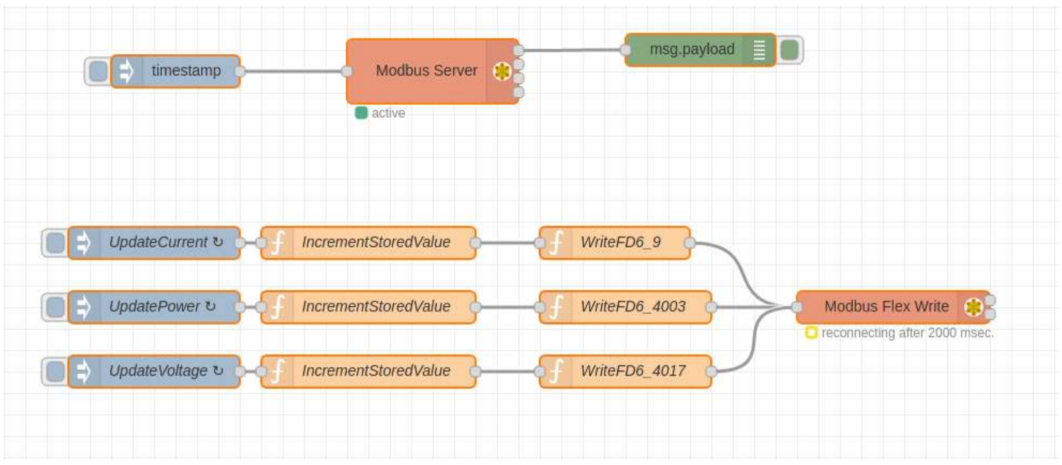


Figure A.2: Simulated Modbus device

Appendix B

Results from other runs

B.1 Modbus updates

B.1.1 Second run

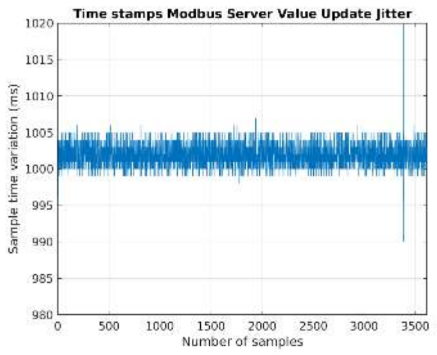


Figure B.1: Time difference between Modbus register update

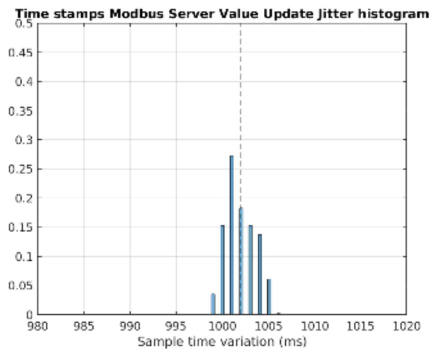


Figure B.2: Histogram of time difference between Modbus register update

Average (ms)	1002.019
Standard Deviation (ms)	5.653
Variance (ms)	31.957
Jitter Interval (ms^2)	$[-12.019, 325.981]$

Table B.1: Control dataset properties

B.1.2 Third run

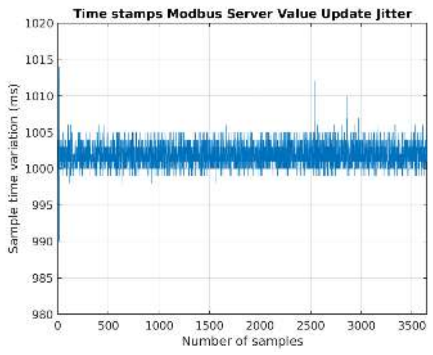


Figure B.3: Time difference between Modbus register update

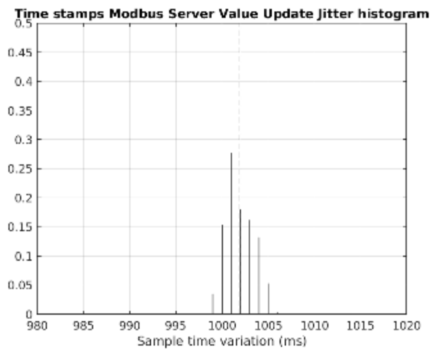


Figure B.4: Histogram of time difference between Modbus register update

Average (ms)	1001.914
Standard Deviation (ms)	1.607
Variance (ms^2)	2.584
Jitter Interval (ms)	$[-11.914, 12.086]$

Table B.2: Control dataset properties

B.2 EdgeX deployed on Virtual Box with Device Service scheduling

B.2.1 Second run

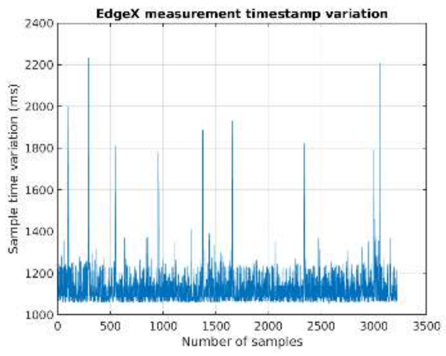


Figure B.5: Time difference between Modbus readings (EdgeX configured for Device Service auto-events)

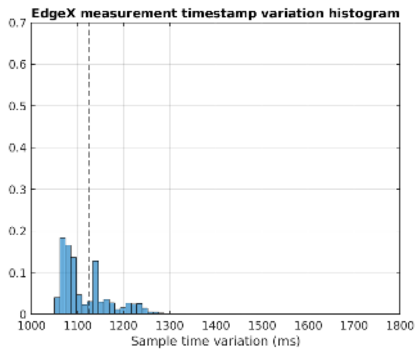


Figure B.6: Histogram of time difference between Modbus readings (EdgeX configured for Device Service auto-events)

Average (ms)	1125.76
Standard Deviation (ms)	87.60
Variance (ms^2)	7673.41
Jitter Interval (ms)	$[-71.12, 1109.34]$

Table B.3: Auto-events VM dataset properties

B.2.2 Third run

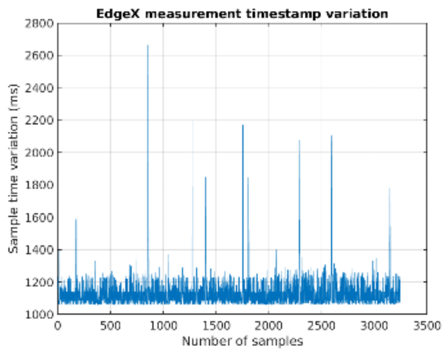


Figure B.7: Time difference between Modbus readings (EdgeX configured for Device Service auto-events on Virtual Machine)

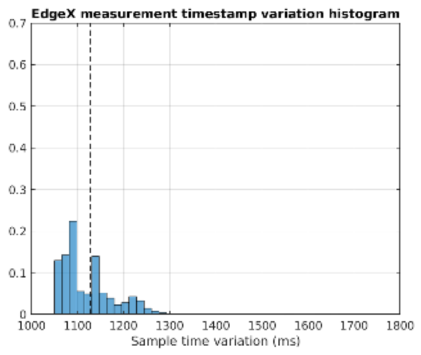


Figure B.8: Histogram of time difference between Modbus readings (EdgeX configured for Device Service auto-events on Virtual Machine)

Average (ms)	1128.17
Standard Deviation (ms)	87.17
Variance (ms^2)	7599.19
Jitter Interval (ms)	$[-71.29, 1536.19]$

Table B.4: Auto-events VM dataset properties

B.3 EdgeX deployed on Raspberry Pi through Docker with Device Service scheduling

B.3.1 Second run

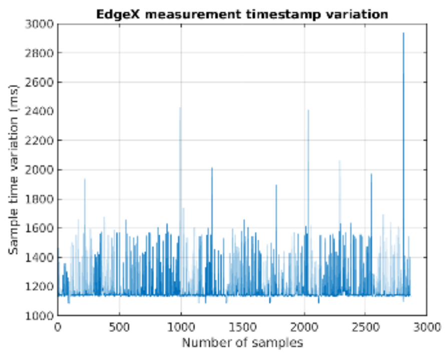


Figure B.9: Time difference between Modbus readings (EdgeX configured for Device Service auto-events on Raspberry Pi through Docker)

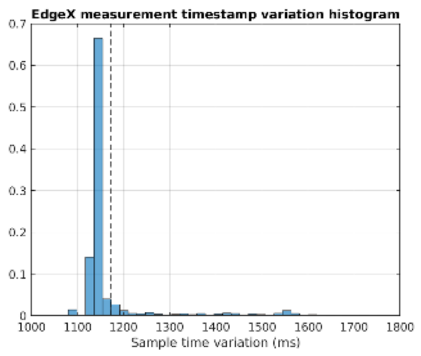


Figure B.10: Time difference between Modbus readings (EdgeX configured for Device Service auto-events on Raspberry Pi through Docker)

Average (ms)	1172.73
Standard Deviation (ms)	109.01
Variance (ms^2)	11883.83
Jitter Interval (s)	$[-89.77, 1766.22]$

Table B.5: Auto-events RPi dataset properties

B.3.2 Third run

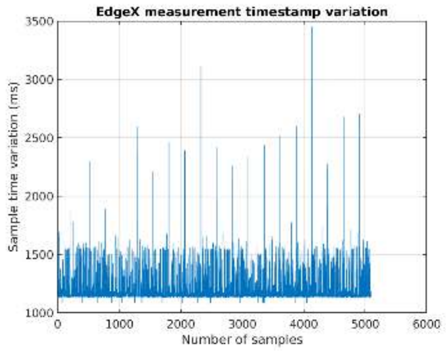


Figure B.11: Time difference between Modbus readings (EdgeX configured for Device Service auto-events on Raspberry Pi through Docker)

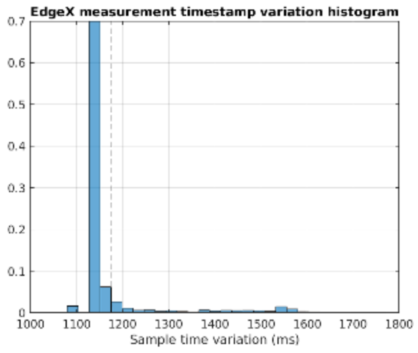


Figure B.12: Time difference between Modbus readings (EdgeX configured for Device Service auto-events on Raspberry Pi through Docker)

Average (ms)	1176.12
Standard Deviation (ms)	127.46
Variance (ms^2)	16246.44
Jitter Interval (ms)	$[-94.42, 2275.19]$

Table B.6: Auto-events RPi dataset properties

B.4 EdgeX deployed on Raspberry Pi directly on OS with Scheduler Service scheduling

B.4.1 Second run

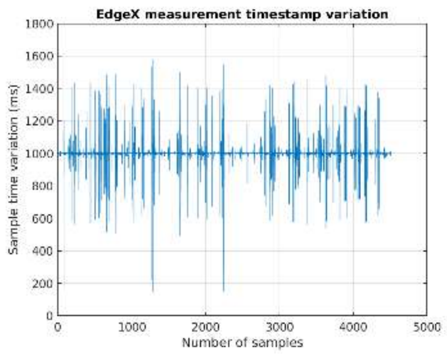


Figure B.13: Time difference between Modbus readings (running directly on the OS on Raspberry Pi)

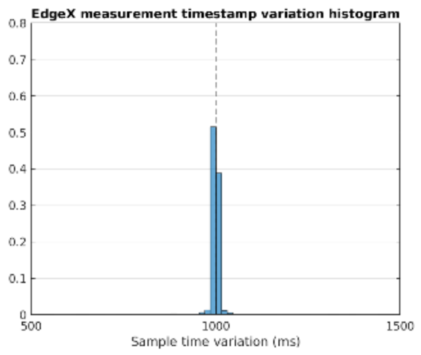


Figure B.14: Histogram of time difference between Modbus readings (running directly on the OS on Raspberry Pi)

Average (ms)	1000.00
Standard Deviation (ms)	71.48
Variance (ms^2)	5109.51
Jitter Interval (ms)	$[-854.60, 645.54]$

Table B.7: Scheduler RPi baremetal dataset properties

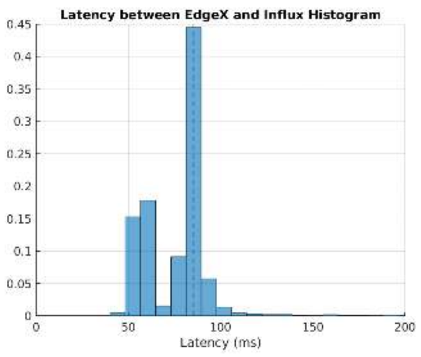


Figure B.15: Latency from EdgeX to InfluxDB (running directly on the OS on Raspberry Pi)

Metric	Baremetal
Average (ms)	85.19
Standard Deviation (ms)	60.11
Variance (ms^2)	3612.94

Table B.8: Latency test baremetal dataset properties

499

B.4.2 Third run

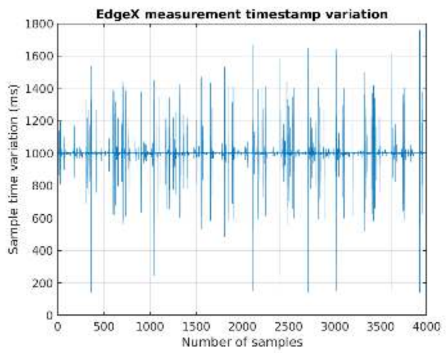


Figure B.16: Time difference between Modbus readings (running directly on the OS on Raspberry Pi)

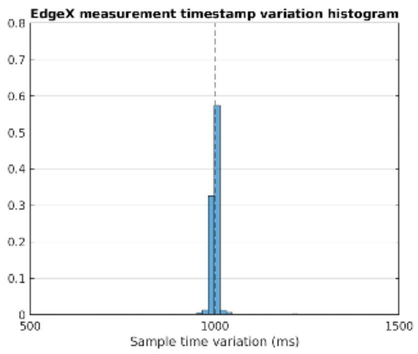


Figure B.17: Histogram of time difference between Modbus readings (running directly on the OS on Raspberry Pi)

Average (ms)	1000.00
Standard Deviation (ms)	81.87
Variance (ms^2)	6702.88
Jitter Interval (ms)	$[-859.67, 759.21]$

Table B.9: Scheduler RPi baremetal dataset properties

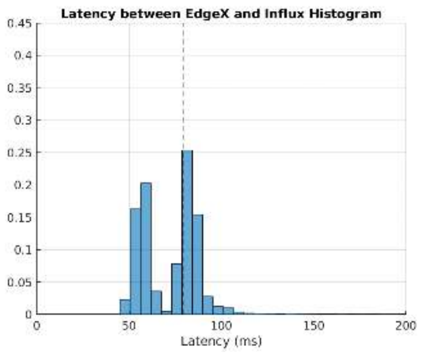


Figure B.18: Latency from EdgeX to InfluxDB (running directly on the OS on Raspberry Pi)

Metric	Baremetal
Average (ms)	79.53
Standard Deviation (ms)	50.96
Variance (ms^2)	2597.16

Table B.10: Latency test baremetal dataset properties

B.5 EdgeX deployed on Raspberry Pi through Docker with Scheduler Service scheduling

B.5.1 Second run

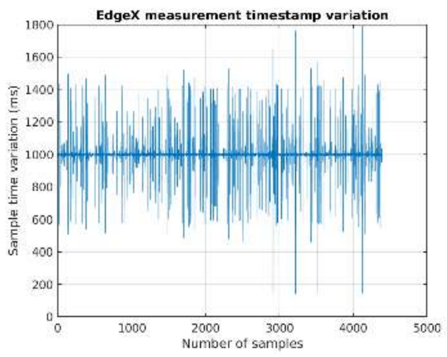


Figure B.19: Time difference between Modbus readings (running through Docker on Raspberry Pi)

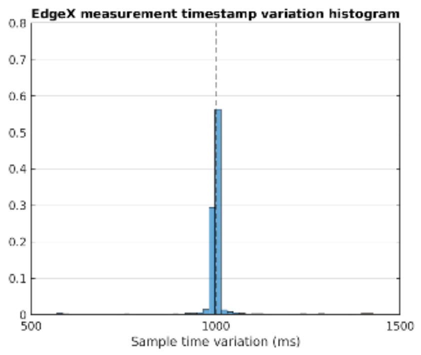


Figure B.20: Histogram of time difference between Modbus readings (running through Docker on Raspberry Pi)

Average (ms)	1000.00
Standard Deviation (ms)	95.88
Variance (ms^2)	9193.60
Jitter Interval (ms)	$[-858.92, 789.49]$

Table B.11: Scheduler RPi Docker dataset properties

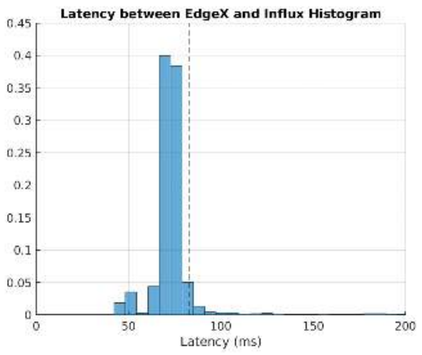


Figure B.21: Latency from EdgeX to InfluxDB (running through Docker on Raspeberry Pi)

Metric	Baremetal
Average (ms)	82.95
Standard Deviation (ms)	60.13
Variance (ms^2)	3615.65

Table B.12: Latency test Docker dataset properties

B.5.2 Third run

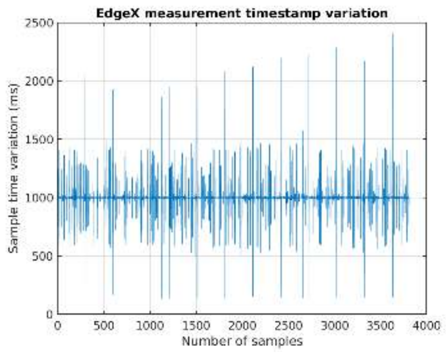


Figure B.22: Time difference between Modbus readings (running through Docker on Raspberry Pi)

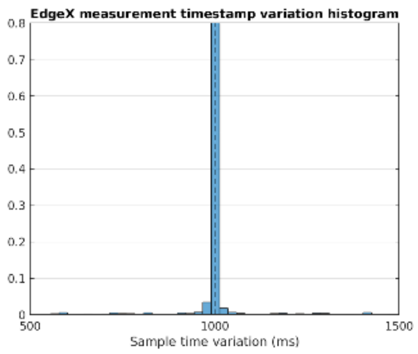


Figure B.23: Histogram of time difference between Modbus readings (running through Docker on Raspberry Pi)

Average (ms)	999.99
Standard Deviation (ms)	122.31
Variance (ms^2)	14958.55
Jitter Interval (ms)	$[-867.18, 1407.02]$

Table B.13: Scheduler RPi Docker dataset properties

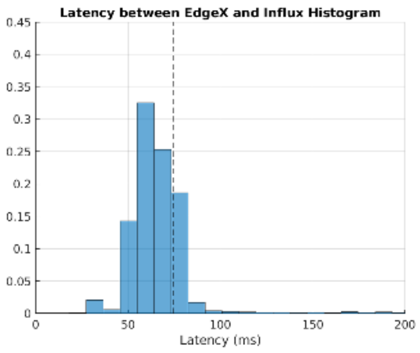


Figure B.24: Latency from EdgeX to InfluxDB (running through Docker on Raspeberry Pi)

Metric	Baremetal
Average (ms)	74.61
Standard Deviation (ms)	58.20
Variance (ms^2)	3386.68

Table B.14: Latency test Docker dataset properties

References

[1] M. F. Körner, D. Bauer, R. Keller, M. Rösch, A. Schlereth, P. Simon, T. Bauernhansl, G. Fridgen, and G. Reinhart, "Extending the automation pyramid for industrial demand response", in *Procedia CIRP*, vol. 81, Elsevier B.V., 2019, pp. 998–1003. DOI: `10.1016/j.procir.2019.03.241`.

[2] ISO Central Secretary, "Enterprise-control system integration — Part 1: Models and terminology", en, International Electrotechnical Commission, Geneva, CH, Standard ISO/IEC 62264-1:2013, 2013. [Online]. Available: `https://www.iso.org/standard/57308.html`.

[3] B. M. Wilamowski and J. David Irwin, "The Industrial Electronics Handbook. Second Edition: Industrial Communication Systems", Tech. Rep., 2011.

[4] O. Nývlt, "Buses, Protocols and Systems for Home and Building Automation", *Tecnolab*, 2011. [Online]. Available: `http://www.tecnolab.ws/pdf/Buses,%Protocols%and%Systems%for%Home%and%Building%Automation.pdf`.

[5] MQTT, *MQTT: The Standard for IoT Messaging.* [Online]. Available: `https://mqtt.org/%20http://mqtt.org/` (visited on 01/30/2021).

[6] *OASIS Message Queuing Telemetry Transport (MQTT) TC | OASIS.* [Online]. Available: `https://www.oasis-open.org/committees/tc%7B%5C_%7Dhome.php?wg%7B%5C_%7Dabbrev=mqtt` (visited on 01/30/2021).

[7] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison", in *2015 IEEE International Conference on Cloud Engineering*, 2015, pp. 386–393.

[8] S. Hykes, *Empowering App Development for Developers | Docker*, 2013. [Online]. Available: `https://www.docker.com/` (visited on 07/16/2020).

[9] IBM Cloud Education, *What is Docker? | IBM*, 2020. [Online]. Available: `https://www.ibm.com/cloud/learn/docker` (visited on 07/16/2020).

[10] S. Di Martino, L. Fiadone, A. Peron, V. N. Vitale, and A. Riccabone, "Industrial Internet of Things: Persistence for Time Series with NoSQL Databases", *Proceedings - 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2019*, pp. 340–345, 2019. DOI: `10.1109/WETICE.2019.00076`.

[11] *MySQL.* [Online]. Available: `https://www.mysql.com/` (visited on 07/16/2020).

[12] GeeksForGeeks, *ACID Properties in DBMS - GeeksforGeeks*, 2017. [Online]. Available: `https://www.geeksforgeeks.org/acid-properties-in-dbms/` (visited on 08/02/2020).

[13] *The most popular database for modern apps | MongoDB.* [Online]. Available: `https://www.mongodb.com/` (visited on 07/16/2020).

[14] *Understanding Database Sharding | DigitalOcean.* [Online]. Available: `https://www.digitalocean.com/community/tutorials/understanding-database-sharding` (visited on 08/02/2020).

[15] *InfluxDB Open Source Time Series Database | InfluxDB | InfluxData.* [Online]. Available: `https://www.influxdata.com/products/influxdb-overview/` (visited on 07/16/2020).

[16] *Eclipse Kura | The Eclipse Foundation.* [Online]. Available: `https://www.eclipse.org/kura/` (visited on 02/13/2020).

[17] *Home - EdgeX Foundry.* [Online]. Available: `https://www.edgexfoundry.org/` (visited on 02/13/2020).

[18] Macchina.io, *macchina.io - IoT Edge Device Software Development and Secure Remote Access Solutions*, 2019. [Online]. Available: `https://macchina.io/` (visited on 02/13/2020).

[19] *Node-RED*. [Online]. Available: `https://nodered.org/` (visited on 02/13/2020).

[20] Node.js, *About | Node.js*, 2017. [Online]. Available: `https://nodejs.org/en/about/` (visited on 07/16/2020).

[21] R. T. Fielding, "REST: architectural styles and the design of network-based software architectures", Doctoral dissertation, University of California, Irvine, 2000. [Online]. Available: `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[22] *1. Introduction to REST - RESTful Java with JAX-RS [Book]*. [Online]. Available: `https://www.oreilly.com/library/view/restful-java-with/9780596809300/ch01.html` (visited on 07/16/2020).

[23] *REST APIs: An Introduction | IBM*. [Online]. Available: `https://www.ibm.com/cloud/learn/rest-apis?mhsrc=ibmsearch%7B%5C_%7Da%7B%5C&%7Dmhq=REST` (visited on 07/16/2020).

[24] T. Sauter, S. Soucek, W. Kastner, and D. Dietrich, "The evolution of factory and building automation", *IEEE Industrial Electronics Magazine*, vol. 5, no. 3, pp. 35–48, 2011, ISSN: 19324529. DOI: `10.1109/MIE.2011.942175`.

[25] F. C. Ferreira, "Gestão técnica centralizada - implementação num edifício do tipo hospitalar", 2017.

[26] P. Domingues, P. Carreira, R. Vieira, and W. Kastner, "Building automation systems: Concepts and technology review", *Computer Standards and Interfaces*, vol. 45, pp. 1–12, 2016, ISSN: 09205489. DOI: `10.1016/j.csi.2015.11.005`. [Online]. Available: `http://dx.doi.org/10.1016/j.csi.2015.11.005`.

[27] T. Mundt and P. Wickboldt, "Security in building automation systems - A first analysis", in *2016 International Conference on Cyber Security and Protection of Digital Services, Cyber Security 2016*, IEEE, 2016, pp. 1–8, ISBN: 9781509007097. DOI: `10.1109/CyberSecPODS.2016.7502336`.

[28] D. Minoli, K. Sohraby, and B. Occhiogrosso, "IoT Considerations, Requirements, and Architectures for Smart Buildings-Energy Optimization and Next-Generation Building Management Systems", *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 269–283, 2017, ISSN: 23274662. DOI: `10.1109/JIOT.2017.2647881`.

[29] G. Lilis, G. Conus, N. Asadi, and M. Kayal, "Towards the next generation of intelligent building: An assessment study of current automation and future IoT based systems with a proposal for transitional design", *Sustainable Cities and Society*, vol. 28, pp. 473–481, 2017, ISSN: 22106707. DOI: `10.1016/j.scs.2016.08.019`. [Online]. Available: `http://dx.doi.org/10.1016/j.scs.2016.08.019`.

[30] *The Modbus Organization*. [Online]. Available: `https://modbus.org/` (visited on 11/28/2020).

[31] *EtherCAT Technology Group | HOME*. [Online]. Available: `https://www.ethercat.org/default.htm` (visited on 11/28/2020).

[32] *Home - Digital Illumination Interface Alliance*. [Online]. Available: `https://www.dali-alliance.org/` (visited on 11/28/2020).

[33] *Eclipse Mosquitto*. [Online]. Available: `https://mosquitto.org/` (visited on 12/12/2020).

[34] S. Rautmare and D. M. Bhalerao, "MySQL and NoSQL database comparison for IoT application", *2016 IEEE International Conference on Advances in Computer Applications, ICACA 2016*, pp. 235–238, 2017. DOI: `10.1109/ICACA.2016.7887957`.

[35] S. Di Martino, L. Fiadone, A. Peron, V. N. Vitale, and A. Riccabone, "Industrial Internet of Things: Persistence for Time Series with NoSQL Databases", *Proceedings - 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2019*, pp. 340–345, 2019. DOI: `10.1109/WETICE.2019.00076`.

[36] *6.3. Modbus - Adding a Device to EdgeX — EdgeX documentation*. [Online]. Available: `https://fuji-docs.edgexfoundry.org/Ch-ExamplesAddingModbusDevice.html` (visited on 12/09/2020).

[37] *Introduction - EdgeX Foundry Documentation*. [Online]. Available: `https://docs.edgexfoundry.org/1.3/` (visited on 12/11/2020).

[38] *Consul by HashiCorp.* [Online]. Available: `https://www.consul.io/` (visited on 12/08/2020).

[39] *Drools - Business Rules Management System (Java™, Open Source).* [Online]. Available: `https://www.drools.org/` (visited on 12/08/2020).

[40] *2. Introduction — EdgeX documentation.* [Online]. Available: `https://fuji-docs.edgexfoundry.org/Ch-Intro.html` (visited on 12/07/2020).

[41] *Edgexfoundry/demo-grove-pi.* [Online]. Available: `https://github.com/edgexfoundry/demo-grove-pi` (visited on 12/09/2020).

[42] *node-red-contrib-influxdb (node) - Node-RED.* [Online]. Available: `https://flows.nodered.org/node/node-red-contrib-influxdb` (visited on 12/09/2020).

[43] *AutoEvents interval may be inaccurate · Issue #517 · edgexfoundry/device-sdk-go.* [Online]. Available: `https://github.com/edgexfoundry/device-sdk-go/issues/517` (visited on 12/20/2020).