



Titre: Multi-Protocol Interoperability Between Distributed Cyber-Physical Systems Towards Industry 4.0 Collaborative Optimization
Title:

Auteur: Md Sabbir Bin Azad
Author:

Date: 2022

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Azad, M. S. B. (2022). Multi-Protocol Interoperability Between Distributed Cyber-Physical Systems Towards Industry 4.0 Collaborative Optimization [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie.
Citation: <https://publications.polymtl.ca/10528/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/10528/>
PolyPublie URL:

Directeurs de recherche: Christophe Danjou
Advisors:

Programme: Maîtrise recherche en génie industriel
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

**Multi-Protocol Interoperability Between Distributed Cyber-Physical Systems
Towards Industry 4.0 Collaborative Optimization**

MD SABBIR BIN AZAD

Département de mathématiques et de génie industriel

Mémoire présenté pour l'obtention du diplôme de *Maîtrise ès sciences appliquées*

Génie industriel

Août 2022

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Multi-Protocol Interoperability Between Distributed Cyber-Physical Systems Towards Industry 4.0 Collaborative Optimization

présenté par **Md Sabbir Bin AZAD**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Fabiano ARMELLINI, président

Christophe DANJOU, membre et directeur de recherche

Matthieu BRICOGNE-CUIGNIERES, membre

DEDICATION

*To Who has taught by the pen. He has taught man
that which he knew not. . .*

(From the Holy Quran, Surah : 96, AL-Alaq, Verse : 4-5)

ACKNOWLEDGEMENTS

Foremost, I would like to express my heartiest gratitude to my research supervisor Dr. Christophe Danjou for the continuous support of my master's research study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance, advice, and wisdom helped me in all the time of research and writing of the thesis. I could not have imagined having a better supervisor and mentor for my master's research studies.

Besides my supervisor, I would like to thank rest of my thesis committee Prof. Fabiano Armellini, Matthieu Bricogne-Cuignieres for their encouragement, insightful comments.

I thank to all my friends and family for helping me survive all the stress and not letting me give up. In particular, I would like to thank my friends Shahab Hamdavi, Ferrakkem Bhuiyan, Moaz Abrar, Rifat Sobhan for their support and thoughtful suggestions. I am grateful to Fahim Shahriar for enlightening me with the research work.

I am grateful to my wife Razia Sultana for her immense sacrifice and continued support. Last but not the least, I would like to thank my parents Abul Kalam Azad and Nargish Azad for supporting me throughout my life.

RÉSUMÉ

L'industrie 4.0 est apparue comme une stratégie potentielle pour fournir une connectivité étendue dans l'environnement de production, qui évolue rapidement, combinée à une demande commerciale croissante et à une fabrication personnalisée de masse. La personnalisation de masse et les produits complexes nécessitent plus de données et une communication M2M plus adaptable qui facilite l'échange de données fluide et l'interaction entre les composants industriels dans la fabrication intelligente. L'intégration de dispositifs IoT industriels au profit de différents secteurs industriels nécessite simultanément une connectivité réseau étendue, une communication interopérable et une collaboration entre les machines en réseau. Bien que les problèmes techniques critiques liés à la connectivité réseau aient été correctement résolus, la technologie n'est pas prête pour une communication flexible et transparente entre des machines disparates. L'un des défis qui découle de ce développement est le besoin croissant de normes et de protocoles interopérables à différents niveaux de l'écosystème de fabrication. Compte tenu de l'infrastructure interopérable requise pour l'industrie 4.0, le document fournit une solution interopérable sécurisée et rentable pour les traducteurs multiprotocoles. La principale contribution de mémoire est une méthode pour cartographier les multi-protocoles IoT, y compris HTTP, MQTT, CoAP, WebSocket et Modbus TCP dans une passerelle à faible coût, ainsi que pour fournir une communication M2M interopérable en duplex intégral efficace et une intégration dans le cloud pour une compatibilité. plates-formes.

ABSTRACT

Industry 4.0 has emerged as a potential strategy to provide extensive connectivity in the production environment, which is rapidly evolving combined with rising commercial demand, mass personalized manufacturing. Mass customization and complicated products necessitate more data and more adaptable M2M communication that facilitates smooth data interchange and interaction between industrial components in smart manufacturing. Integrating industrial IoT devices to benefit different industry sectors simultaneously requires extensive network connectivity, interoperable communication, and collaboration among the networked machines. While critical technical issues with network connectivity have been properly addressed, the technology is not ready for flexible and seamless communication between disparate machines. One of the challenges that arises as a result of this development is the growing need for interoperable standards and protocols at various levels of the manufacturing ecosystem. Considering the interoperable infrastructure required for Industry 4.0, the research work provides a secure and cost-effective interoperable solution for multi-protocol translators. The key contribution of the research is a method for mapping IoT multi-protocols including HTTP, MQTT, CoAP, WebSocket, and Modbus TCP into a low-cost gateway, as well as providing effective full-duplex interoperable M2M communication and cloud integration for compatible platforms.

TABLE OF CONTENTS

DEDICATION	III
ACKNOWLEDGEMENTS	IV
RÉSUMÉ.....	V
ABSTRACT.....	VI
TABLE OF CONTENTS	VII
LIST OF TABLES	XI
LIST OF FIGURES.....	XII
LIST OF SYMBOLS AND ABBREVIATIONS.....	XV
LIST OF APPENDICES	XVIII
CHAPTER 1 INTRODUCTION.....	1
1.1 Research Questions	3
1.2 Thesis Structure.....	4
CHAPTER 2 STATE OF THE ART.....	5
2.1 Industrial Internet of Things(IIoT).....	5
2.2 Interoperable Communication between Cyber-Physical Systems	6
2.2.1 Technical Interoperability	6
2.2.2 Syntactic Interoperability	7
2.2.3 Semantic Interoperability	7
2.2.4 Cross-domain Interoperability.....	7
2.2.5 Horizontal Interoperability	7
2.2.6 Vertical Interoperability	8
2.3 Challenges in Implementation of Interoperability	9
2.4 Existing Interoperable Standards and their Limitations	10

2.4.1	OPC UA	10
2.4.2	MTConnect.....	11
2.4.3	Other Interoperable Approaches	12
2.5	Research Gap.....	13
2.6	IoT Gateway	14
2.7	IoT Communication Standards.....	17
2.7.1	MQTT.....	17
2.7.2	CoAP.....	18
2.7.3	HTTP.....	19
2.7.4	WebSocket	19
2.7.5	Modbus TCP	21
2.7.6	AMQP	21
2.8	Comparison of IoT Standard Protocols.....	22
2.9	Literary Review Conclusion.....	24
CHAPTER 3 RESEARCH METHODOLOGY		25
3.1	Research Objectives	25
3.2	Research Design.....	26
3.2.1	Protocol Selection Framework	26
3.2.2	Gateway Design	27
3.2.2.1	Data Formatting.....	28
3.2.2.2	Protocol Bridging	29
3.2.2.3	Interoperable communication among nodes and gateways.....	31
3.2.2.4	Data Process and Storage	32
3.3	Methodology Conclusion	33

CHAPTER 4	RESEARCH DEVELOPMENT	34
4.1	Multi-Protocol Gateway Development	35
4.2	Multiple Server Configuration on the Gateway	37
4.2.1	MQTT Broker Configuration	37
4.2.2	HTTP Server (Apache Web Server).....	37
4.2.3	Modbus TCP Server	38
4.2.4	CoAP Server Implementation	39
4.2.5	WebSocket Server Deployment	41
4.3	Node Microcontrollers for different Sensor Integration	42
4.3.1	Node1 as Raspberry Pi 3 and Sensor DHT22	42
4.3.2	Node2 as Arduino Uno Wi-Fi Rev2 and Sensor BME280	45
4.3.3	Node3 as ESP32 and Sensor as MQ-135	48
4.4	Communication between node Microcontrollers and the Sensors	52
4.4.1	Node1: Raspberry Pi 3 and DHT22 Communication.....	52
4.4.2	Node2: Arduino Uno Wi-Fi Rev2 with BME280 Sensor Communication	53
4.4.3	Node3: ESP32 with MQ-135 Gas Sensor Communication	54
4.5	Communication Protocol Selection for the Nodes	55
4.6	Communication between Node Microcontroller and Gateway	57
4.6.1	Node2 Data Received by Gateway over WebSocket Protocol	57
4.6.2	Node3 Data Transfer to Gateway over MQTT protocol	58
4.6.3	Node1 Data Transfer to Gateway over CoAP Protocol	59
4.6.4	Node1 Data Transfer to Gateway over Modbus TCP Protocol.....	60
4.6.5	Node1 Data Transfer to Gateway over HTTP Protocol	60
4.7	Data Collection and Storage.....	61

4.7.1	Data Store to Local and Cloud Database	62
4.7.2	KEPServerEX Data Logging and Communication.....	63
4.7.3	Data Store to Azure IoT Hub and Data Explorer Databases.....	65
4.8	Visualization of Real-Time Node Data in Web Application	67
4.9	Development Conclusion	67
CHAPTER 5	RESULTS AND DISCUSSION	69
5.1	Case Study: Implementation on ThingsBoard Platform	69
5.2	Configuration with ThingsBoard Platform	70
5.2.1	Gateway Configuration	70
5.2.2	ThingsBoard Configuration.....	73
5.3	Gateway Transferring Node1 data to ThingsBoard	75
5.4	Gateway Sending Node2 data to ThingsBoard	75
5.5	Gateway Publishing Node3 data to ThingsBoard	76
5.6	Real-time Visualization on ThingsBoard.....	77
5.7	Results	79
5.8	Limitations	81
CHAPTER 6	CONCLUSION AND RECOMMENDATIONS.....	82
6.1	Conclusion.....	82
6.2	Future Work	83
REFERENCES	84
APPENDICES	92

LIST OF TABLES

Table 2.1 Comparison of Different IoT Standards and Protocols.....	22
Table 4.1 Comparison of Efficient Gateways for Smart IoT Environment	35
Table 4.2 Comparison of Different Temperature and Humidity Sensors	43
Table 4.3 Comparison of Different Microcontroller Boards.....	45
Table 4.4 Different Air Quality Range with Status.....	51

LIST OF FIGURES

Figure 2.1 General architecture of device connected using MTConnect standard	12
Figure 2.2 Main characteristics of the IoT gateway	15
Figure 2.3 Overview structure of data transmission system using MQTT protocol	18
Figure 2.4 HTTP protocol over REST architecture	19
Figure 2.5 WebSocket over TCP sequence diagram	20
Figure 2.6 Client-Server architecture of basic Modbus TCP/IP communication	21
Figure 3.1 Protocol selection block diagram for applicable IoT systems	27
Figure 3.2 Example of systematized data format	28
Figure 3.3 Protocol bridging for multiple nodes with different protocols	31
Figure 3.4 Interoperable protocol communication among the nodes and gateways	32
Figure 4.1 Proposed multi-protocol gateway architecture implementation framework.....	34
Figure 4.2 Architecture of Raspberry Pi 4 Model B	36
Figure 4.3 Mosquitto MQTT broker running on terminal	37
Figure 4.4 Apache3 Raspbian version webserver installation confirmation.....	38
Figure 4.5 Register mapping for Modbus communication	39
Figure 4.6 CoAP requests for node sensors	40
Figure 4.7 CoAP payload observing from GPIO output.....	41
Figure 4.8 CoAP successful response code on message receiving	41
Figure 4.9 Pinouts of DHT22 temperature-humidity sensor.....	44
Figure 4.10 DHT22 sensor connection with node1 Raspberry Pi 3B	44
Figure 4.11 Arduino UNO Wi-Fi microcontroller with pinouts	46
Figure 4.12 BME280 Pressure-Altitude sensor pinouts.....	47
Figure 4.13 BME280 sensor connection wiring with Arduino UNO W-Fi microcontroller	48

Figure 4.14 ESP32-WROOM development board architecture and pinouts	49
Figure 4.15 Pinouts of MQ-135 environmental sensor	50
Figure 4.16 Sensitivity characteristics of MQ-135 environmental sensor	50
Figure 4.17 Experimental setup for node3 ESP32 and sensor MQ-135	52
Figure 4.18 Node1 raspberry Pi 3 receiving data from DHT22 temperature-humidity sensor.....	53
Figure 4.19 Node2 Arduino UNO receiving sensor data from BME280 sensor	54
Figure 4.20 Node3 ESP32 receiving sensor air quality data fromMQ-135 environmental sensor	55
Figure 4.21 Protocol selection interface for the nodes to receive data on the gateway	55
Figure 4.22 Protocol assigning from gateway to nodes for sending payloads	56
Figure 4.23 Node2 data received by gateway over WebSocket protocol	57
Figure 4.24 Node3 air quality data received by gateway over MQTT protocol	58
Figure 4.25 Node1 data received by gateway over CoAP protocol	59
Figure 4.26 Node1 data received by gateway over Modbus TCP/IP protocol.....	60
Figure 4.27 Node1 data received by gateway over HTTP protocol.....	61
Figure 4.28 Table structure for data storage in gateway local database	62
Figure 4.29 Data storage table in gateway local database.....	62
Figure 4.30 Data sent from gateway to cloud database.....	63
Figure 4.31 Gateway sending node3 air quality data to KEPServerEX.....	64
Figure 4.32 KEPServerEX data logging in Excel	65
Figure 4.33 Gateway sending temperature humidity data to Azure IoT Hub	66
Figure 4.34 Data ingestion to Azure Data Explorer database	66
Figure 4.35 Temperature humidity real-time data visualization in Azure IoT web application....	67
Figure 5.1 Cloud based ThingsBoard platform integration with muti-protocol gateway	70
Figure 5.2 Configuration for MQTT mapping in JSON format.....	71

Figure 5.3 Gateway-ThingsBoard configuration parameters	72
Figure 5.4 Sensor data formatted to JSON data format for message payload	72
Figure 5.5 Gateway configuration as device on ThingsBoard cloud platform	73
Figure 5.6 Gateway access token credentials generated on ThingsBoard	74
Figure 5.7 Gateway publishing Node1 data to ThingsBoard Cloud	75
Figure 5.8 Gateway publishing node1 data to ThingsBoard cloud	76
Figure 5.9 Gateway publishing node3 data to ThingsBoard cloud	77
Figure 5.10 Latest telemetry received by the device on ThingsBoard cloud platform	78
Figure 5.11 Real-time data visualization dashboard on ThingsBoard platform	79

LIST OF SYMBOLS AND ABBREVIATIONS

CPSs	Cyber Physical Systems
IoT	Internet of Things
IIoT	Industrial Internet of Things
ARM	Advanced RISC Machine
RPi	Raspberry PI
CCGX	Color Control GX
OS	Operating System
M2M	Machine-to-Machine
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
MQTT	MQ Telemetry Transport
CoAP	Constrained Application Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
SCTP	Stream Control Transmission Protocol
AMQP	Advanced Message Queuing Protocol
FTP	File Transfer Protocol
JSON	JavaScript Object Notation
CSV	Comma Separated Values
TSV	Tab-Separated Values
ORC	Optimized Row Columnar
OPC UA	OPC Unified Architecture
COM	Component Object Model

DCOM	Distributed Component Object Model
CAN	Controller Area Network
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
SSL	Secure Sockets Layer
SASL	Simple Authentication and Security Layer
TLS	Transport Layer Security
QoS	Quality of Service
XML	Extensible Markup Language
PLC	Programmable Logic Controller
HMI	Human Machine Interface
SDN	Software Defined Network
Git	Global Information Tracker
AI	Artificial intelligence
HTML	HyperText Markup Language
RFID	Radio-frequency Identification
RTU	Remote Terminal Unit
REST	Representational State Transfer
API	Application Programming Interface
UI	User Interface
LAN	Local Area Network
LTE	Long-Term Evolution
GPIO	General-Purpose Input/Output
PWM	Pulse-Width Modulation

RAM	Random Access Memory
MCU	Microcontroller Unit
PPM	Parts Per Million
hPa	Hectopascal
k Ω	Kilo Ohms
μ A	Micro Ampere
AQI	Air Quality Index
SQL	Structured Query Language
VCC	Voltage Common Collector
GND	Ground
SDA	Serial Data
SCL	Serial Clock

LIST OF APPENDICES

Appendix A	NODE devices code and algorithm	92
Appendix B	Gateway code and algorithm	99

CHAPTER 1 INTRODUCTION

Industry 4.0, fourth industrial revolution brought about by introduction of IoT and CPSs (Kagermann, Helbig, Hellinger, & Wahlster, 2013), has emerged as a promising approach to provide extensive connectivity in manufacturing environment (Li, Lai, & Poor, 2012). The evolution of smart manufacturing technology is changing rapidly, and coupled with increasing commercial demand, laminated modeling shows many advantages in providing customized and specifically designed products (Meng, Y., Yang, Chung, Lee, & Shao, 2018). Mass customized and complex products leads to a greater need of information and more flexible automation solutions (ElMaraghy, 2005). This flexibility and more advanced information processing requires more intelligence in the system. It's also designed for human workers, therefore it requires smart factory (Zuehlke, 2008). Smart manufacturing relies on machine-to-machine communication, which makes it possible for manufacturing items like devices, machines, systems, and people to communicate and share data easily which allows for dynamic configuration and autonomous collaboration between them (Lu & Asghar, 2020). However, the challenges in linked machine collaboration, interoperability, and communication still exist (Meng, Z. Z., Wu, & Gray, 2017). Although the crucial technical issues of network connectivity (Wollschlaeger, Sauter, & Jasperneite, 2017) have been addressed adequately, the technologies are not ready for communication between heterogeneous machines in a flexible and seamless manner (Meng, Z. Z. et al., 2017). One of the challenges that arises as a result of this evolution is the growing need for interoperability at various levels of the manufacturing ecosystem (Zeid, Sundaram, Moghaddam, Kamarthi, & Marion, 2019).

Manufacturing ecosystems have evolved into interconnected networks of automation devices, services, and businesses as a result of recent advancements in manufacturing technology, including cyber-physical systems, the industrial internet, artificial intelligence (AI), and machine learning (Zeid et al., 2019). Cyber-Physical Systems (CPS) are collections of physical and computer components that are integrated with each other to operate a process safely and efficiently. The concept of industrial wireless CPS is based on the interaction between the cyber and physical worlds, including industrial wireless devices and physical components (e.g., sensors and actuators), and cyber components (e.g., processing and storage devices) (Pan et al., 2019). Difficulty in the communication of cyber physical systems, such as machines, sensors and devices, insufficient

middle-ware interfaces, or APIs to deploy heterogeneous manufacturing resources are challenges to enable interoperable cloud manufacturing applications. Furthermore, the rapid advancement of machine learning algorithms and computing power has created new opportunities to assist manufacturing processes and decisions with additional data insights (Zeid et al., 2019).

Industry 4.0 initiative (W., 2014) aims to develop efficient and low-cost production with flexible workflows for producing high-quality personalized products at low costs. Industry 4.0 uses Cyber-Physical Systems (CPS) in its highly intelligent and flexible manufacturing process. In particular, manufacturing automation needs personalized-product-based manufacturing process automation and vertical integration of manufacturing systems. Both collectively form dynamic end-to-end engineering integration. For organizations, such integration can lead to better collaboration between different roles and functions (Schuh, Potente, Varandani, Hausberg, & Fränken, 2014). Increased knowledge sharing and co-operation can also decentralize decision-making and increase the autonomy of individuals (Mattsson, Karlsson, Fast-Berglund, & Hansson, 2014). Furthermore, system integration is an enabler to implement IoT, CPS, and Smart Factories (Hermann, Pentek, & Otto, 2015). Successful system integration requires good strategies for managing system heterogeneity and middleware connectivity. However, integrating new devices to benefit different industry sectors simultaneously requires significant challenges as part of what is being called the Industrial Internet of Things (IIoT) (Serpanos & Wolf, 2018). IIoT devices have the following unique characteristics such as low processing power and storage capacity, narrow data download, low bandwidth, and limited battery life (Sisinni, Saifullah, Han, Jennehag, & Gidlund, 2018). Given the ubiquity of these devices and facing such limitations, it is necessary to develop new types of communication protocols designed to deal with these limitations (Garrocho et al., 2020). Generally, the used protocols are based on communication through cloud and between machines (Kshetri, 2017). A scalable interoperability solution needs the ability to automatically (without much effort) adjust the semantic relationships of dynamic information systems. Semantic interoperability must be achieved in interworking solutions to provide a common meaning for the data exchanged by heterogeneous devices, even if the heterogeneous devices belong to different domains (Cavalieri, 2021). Different communication protocols are employed in IoT, e.g., HTTP, CoAP, MQTT, Web Sockets, AMQP, among others. The main driving force for the design of such protocols is the hardware limitations of embedded devices, which impede the use of traditional

network protocols. Communication protocol integration enables interoperability between multiple devices and services and, one possible solution is to design a multi-protocol strategy (Desai, Sheth, & Anantharam, 2015). Though existing standards such as MTCConnect, OPC-UA enable industrial object specifications and information-rich M2M communications, the information models generated by these standards are not semantically defined, making semantic understanding and intelligent decision-making difficult (Grangel-González, 2017). Therefore, it is important to identify the gap between the current state of information and communication systems for manufacturing operations and what is required to achieve the future interconnected heterogeneous systems of autonomous entities. In aforementioned situation, an IoT system which has capability of interchanging between access protocols may overcome the said challenges in interconnected heterogeneous systems.

1.1 Research Questions

The problematic is divided into two research questions (Q1 and Q2). The first question Q1 addresses how heterogeneous devices can be connected and shared information with each other by means of structuring any data with different access standards. The second question Q2 directs to make a low-cost interoperable system for collaborative M2M interoperable optimization in small and medium enterprises.

Q1: What are the advantages of an IoT interoperable system to assist interconnecting heterogeneous devices with different access standards?

It has been difficult to successfully integrate interoperability among different manufacturing operations and processes to accomplish data-driven monitoring, prediction, control, and optimization. The intention of this question is to distinguish the outcomes of interoperable IoT system that enables machine to machine communication between manufacturing items like machines, devices, systems etc.

Q2: What IoT solution can be provided to make this interoperable system cost effective for small and medium enterprises?

Large enterprises use advanced levels of interoperability management methods and tools to handle complexity more comprehensively. Conversely, SMEs deals with the obstacles of autonomous

interoperability between business, manufacturing functionalities and distributed control systems. The reason for this question is to identify cost-effective interoperable IoT systems for SMEs to move to CPS-based automation paradigms with digital modularity and interoperability during real-time end-to-end integration

1.2 Thesis Structure

This thesis is structured as follows: Chapter 2 discusses interoperability standards, finds the gap among current interoperable solutions and challenges implementing interoperable communication, it also addresses different communication protocols and their characteristics ; Chapter 3 describes the research objectives and steps taken to accomplish the objectives with research design strategies Chapter 4 demonstrates development of the proposed interoperable system, communication with different devices, sensors and integration of different cloud and industrial databases ; Chapter 5 tests feasibility of deployment model and discusses results and limitations of the proposed gateway; finally, Chapter 6 discusses conclusions and future work.

CHAPTER 2 STATE OF THE ART

Design and development of interchanging and interoperable structure between IoT standards to achieve seamless connectivity in heterogeneous systems require consideration of many areas including industrial Internet of Things connectivity and accessibility, exchanging information between standards and format, analyzing interoperable requirements and limitations. This research project will involve setting up an IoT interoperable structure for heterogenous device connectivity. Before doing so, this chapter will start with presenting a literature review on interoperable communication and their challenges in implementation, existing interoperable solutions, and their limitations. The second part of this literature review will discuss gateway based interoperable approaches, characteristics, advantages and disadvantages of communication standards and their comparison.

2.1 Industrial Internet of Things(IIoT)

The Industrial Internet of Things (IIoT) has emerged as a general concept of the application of the Internet of Things to the industrial sector. IIoT primarily refers to an industrial framework in which many devices or machines are connected and synchronized using Internet platform technology in the context of the machine-to-machine and Internet of Things (Sisinni et al., 2018). Compared to the Internet of Things (IoT) in the private sector, the focus of the industrial sector is on networking the machine and end-to-end process chains. In fact, this is a generalization of Industry 4.0 and seems to focus on the efficiency of industrial processes. Industrial IoT allows manufacturers to digitize almost any part of their business. The fundamental characteristics of IIoT lead to requirements that need to be met by the reference architecture. Key requirements identified by the International Telecommunication Union (ITU) include interoperability, identity-based connectivity, network and service autonomy, location-based service integration, security and privacy, as well as capabilities for management of things and services, including plug and play (Serpanos & Wolf, 2018).

2.2 Interoperable Communication between Cyber-Physical Systems

Advances in embedded systems and information and communication technology have further expanded the adoption of wireless connectivity within CPS. CPS represents the edge of the Internet of Things (IoT). This is a vision that allows people and things to connect anytime, anywhere, to anything, to anyone, ideally through any path / network or service (Vermesan et al., 2009). However, due to the rising complexity that is beyond human grasp, gadgets without the capacity to adjust themselves to the environment will be unable to dynamically participate in the production line in Industry 4.0. As a result, "intelligent interoperability" is crucial in "enabled the enabler" and ensuring that research projects in the field of IoT play a prominent role in Industry 4.0 (Lelli, 2019). In other words, the device must have the ability to describe itself in terms that both machines and humans can understand. Therefore, they facilitate an implicit or explicit semantic description of themselves (Hermann et al., 2015).

The European Telecommunication Standards Institute (ETSI) and the European Interoperability Framework (EIF) define four levels of interoperability in complex systems. Technique, syntax, semantic, and cross-domain interoperability (Izza, 2009). In addition, to unlock the full potential of IoT vision, we need standards that enable horizontal and vertical interoperability, operation, and programming across devices and platforms, regardless of model or manufacturer (Hatzivasilis et al., 2018).

2.2.1 Technical Interoperability

Technical interoperability is generally linked with hardware or software components, systems, and platforms that enable machine-to-machine communication, and is frequently concentrated on communication protocols and the infrastructure required for those protocols to function (Kubicek, Cimander, & Scholl, 2011). The main limitation in achieving technical interoperability between heterogeneous systems is the issue of old interoperability between old and new systems. The technical interoperability specification is by listing a list of existing standards that interfaces, interconnect services, data integration services, data exchanges, and communication protocols must use to achieve interoperability. It will be configured. This approach works if interoperability between stations are considered, but if end users want to easily customize their stations to meet different operational needs, more actions are required (R.L., 2005).

2.2.2 Syntactic Interoperability

For industrial systems, syntactic interoperability entails establishing generic structures for data flow across heterogeneous systems and components at various levels from multiple suppliers and platforms (Givehchi, Landsdorf, Simoens, & Colombo, 2017). Two or more systems can interact and share data through syntactic interoperability, but the interface and programming languages must be compatible. Manufacturing equipment must be able to parse messages to accurately decode the message to its pieces, such as message content, language, and sender, to establish syntactic interoperability at the industry level.

2.2.3 Semantic Interoperability

Interoperability at the semantic level includes the technology needed to enable communication platforms to share the meaning of information. Interoperability between the components of a large distributed system is the ability to exchange services and data with each other. Semantic interoperability ensures an agreement and common understanding between system requesters and providers, such as messaging protocols, procedure names, error codes, and argument types. Semantic interoperability in heterogeneous industrial systems is a promising approach to addressing the complexity of multi-vendor and multi-technology systems (Loskyll, 2012).

2.2.4 Cross-domain Interoperability

The new IoT platform provides a heterogeneous way to access things and their data. Cross-domain interoperability allows to build an IoT ecosystem with cross-platform, cross-standard, and cross-domain IoT services and applications. Such interoperability requires the extraction of data and services to obtain a common subset of information and services in the cooperating domain. This includes, for example, business process interoperability (BPI). This allows systems in different domains to integrate and communicate with each other using a well-defined standard business language (Honkola, Laine, Brown, & Oliver, 2009).

2.2.5 Horizontal Interoperability

Horizontal interoperability enables network-independent open standards and networking with a variety of existing vertical M2M systems on IoT platforms. Open standards are an important tool

for providing interoperability between and within different domains. Horizontal integration involves coordinating information flows and systems across different systems so that all data can be accessed and analyzed on a single platform. The main function of this type of horizontal platform is to enable the development of services that are independent of the underlying heterogeneous network devices. Most solutions only provide cross-domain compatibility and typically act as a closed silo with a narrow application focus that imposes specific data formats and interfaces. Mechanisms for resolving these issues and achieving horizontal interoperability include gateway proxies for messaging protocols. The multi-protocol ecosystem allows gateways to extend their capabilities and interact with devices that support a variety of protocols (Hatzivasilis et al., 2018).

2.2.6 Vertical Interoperability

Vertical Interoperability is the capability of manufacturing enterprises to exchange technical and enterprise information in a comprehensible manner. SmartFactoryKL, an EU-funded initiative, demonstrated one of the fewest vertical integrations of information technology in an organization with the shop floor (Weyer, Schmitt, Ohmer, & Gorecky, 2015). By designing a standardized plug-and-play multivendor interface, the research and development took a step toward Industry 4.0 and decentralized corporate integration. To accomplish vertical integration, the project employs technologies such as RFID, web services, and OPC UA (Weyer et al., 2015). Multivendor interfaces, on the other hand, necessitate multi-vendor protocol integration to monitor, manage, and process data at all phases of development.

Facilitating product interoperability in a multi-vendor, multi-network, and multi-service context is one of the main goals of the creation of interoperability standards. Multiple standards from several organizations that create standards are frequently the foundation of complex goods and systems. Users benefit from a far wider range of products thanks to interoperability, and manufacturers can take advantage of the economies of scale that a larger market delivers. Therefore, interoperability is essential to the success of contemporary technologies, and market demand has made sure that interoperability has a central place in standardization.

2.3 Challenges in Implementation of Interoperability

Given the complexity of the process, the factors that influence interoperability should be multivariate (Zeid et al., 2019). Key interoperability barriers include data inconsistencies, scalability, inconsistent data formats or standards, connectivity in the IoT space, and increased operational costs when using and installing various commercially available products.

- Data from heterogeneous sources can lead to data layer inconsistencies and requires more resources to optimize unstructured data (Kadadi, Agrawal, Nyamful, & Atiq, 2014). In the heterogeneous environment, one of the main problems with schema integration is resolving data inconsistencies that may exist in different data sources of semantically identical data. This semantically identical data resolves data representation conflicts when represented differently in different data sources.
- Integrating new data from multiple resources with data from legacy systems creates scalability issues (Kadadi et al., 2014). To connect non-interoperable devices and applications, custom own developed middleware is needed. This is a time-consuming process and needs to be updated as new components are integrated. Introducing different types of sensors and embedded systems is difficult.
- There are no rules or standards set at the application level. That cannot combine or complement the data collected from different sensors and devices. It is not possible to integrate devices from different manufacturers. No network infrastructure has multiple communication protocols such as MQTT, CoAP, HTTP, Modbus, and software from multiple vendors that always connect different devices and networks.
- There is no common data format syntax for semantic level integration and interoperable peer-to-peer communication between CPS for effective M2M information exchange and faster action plans for sustainable Industry 4.0 manufacturing.
- Service providers are associated with and adhere to IoT devices or software provided by a single vendor. This can result in higher operating costs later and can lead to problems with product functionality and stability.

2.4 Existing Interoperable Standards and their Limitations

Few years after the initial development of the Internet, several methods, and frameworks for exchanging data between machines and applications were proposed. GE Fanuc Automation has developed “Cimplicity”, which enables factory and plant monitoring with XML-based data via WebViewScreen. This service can also generate and display alerts (Waurzyniak, 2001). Another application, called FactoryFlow, has been developed by UGS to provide offline access to factory floor processes, plans, and simulation data. MDSI has developed an application called OpenCNC for the purpose of accessing data on CNC machines from the Internet. This application is a software-only machine tool controller that installs on a Windows PC to collect real-time data and publish it to a database. This gives the user access to the OpenCNC "Important Events" file created from the collected PLC data (Wang, Orban, Cunningham, & Lang, 2004). The development of custom applications and interfaces such as Cimplicity, WebView, FactoryFlow, and OpenCNC has led to the initiative to develop open communication standards that allow Internet connectivity to manufacturing facilities ("MTConnect is the communication standard of choice for manufacturing," 2017). Existing standards, such as MTConnect, OPC-UA, and AutomationML, enable M2M communication with a wealth of industrial object specifications and information, so the information model generated from these standards is semantically undefined. Understanding meaning and intelligent decision-making are challenges (Lu & Asghar, 2020). Also, there are several solutions (proposed or implemented) with the aim of increasing IoT interoperability which still lack full-duplex (connectivity in both direction) integration.

2.4.1 OPC UA

The OPC UA (OPC Unified Architecture) technology was created to provide genuine unified connection based on a safe and easy platform to allow corporate interoperability and address enterprise integration difficulties (Mai, Vu, & Myeong-Jae, 2011). OPC extends the very successful OPC communication protocol, which is utilized in horizontal integration transferring data across automation systems and vertical integration transferring data between different layers of industrial automation (de Souza et al., 2008). OPC Server is built on COM / DCOM component technology, which makes it not only dependent on the Windows platform, but also has several technical flaws in data transfer and security across the network. OPC UA provides a standard and

comprehensive address space, service model, and security model, allowing a single OPC UA server to combine data, alarms, and events, as well as historical data, into its address space and employ a set of uniform services to provide them with an external interface. In comparison to earlier OPC specifications, OPC UA included more new capabilities, such as a sophisticated data structure, a unified address space, platform independence, and improved security (Qiao & Feng, 2011).

With OPC UA, the server typically maintains a complete information model. Whether the information space is divided into different namespaces and information models, or almost all OPC UA servers use the same information model, each server provides all the namespaces it needs. This is both an advantage and a disadvantage. The advantage is that the server is completely self-contained and applications that use this information only need to communicate with this one server. This reduces communication effort and client complexity. On the other hand, this has many drawbacks. Especially for small devices, the additional information model introduces high overhead due to additional memory requirements. In fact, the memory required for the information model is one of the major obstacles to deploying OPC UA in inexpensive microcontrollers. (Iatrou & Urbas, 2016a, 2016b). Objects Linking and Embedding for Process Control-Integrated Architecture (OPC UA) is an industrial communication framework that is being heavily promoted for the integration of distributed systems. It has many promising properties, but there are still situations where other communication protocols have advantages (Derhamy, Rönholm, Delsing, Eliasson, & Deventer, 2017). The HTTP and TCP protocols are supported by OPC UA. Message-based security is used by the OPC UA technology, which implies messages may be sent through HTTP and TCP ports. OPC UA is unable to work in a multiprotocol environment. For OPC UA clients, gaining access to non-OPC UA services is also an issue.

2.4.2 MTConnect

MTConnect is a protocol created by the Association for Manufacturing Technology (AMT) that provides open, royalty-free device connectivity standards and technologies, as well as simple software or firmware devices. Transfers data over the network using the Internet Protocol for broader interoperability between numerical control devices. The adapter obtains data from the machine and passes it on to the agent. The data is stored on a TCP server, and the agent offers a REST interface, which allows the data to be obtained through HTTP request-reply. As a result,

many users can have access to the machine's real-time data collection. Basic architecture of MTConnect device connection process is illustrated in Figure 2.1

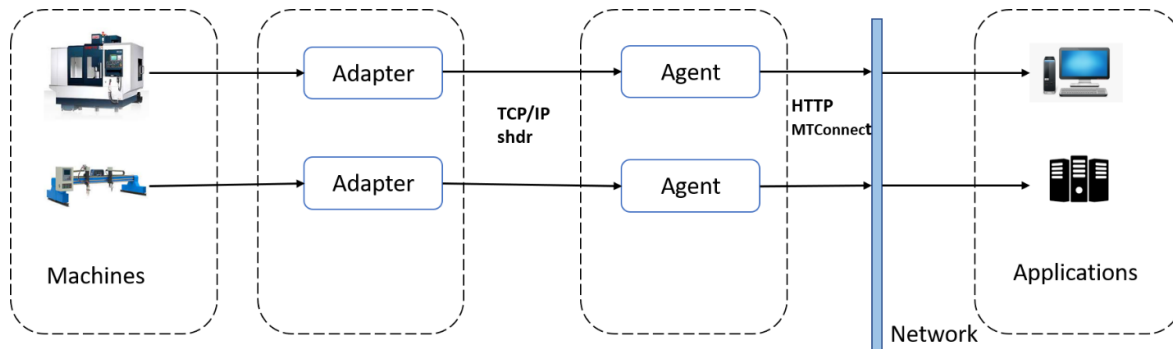


Figure 2.1 General architecture of device connected using MTConnect standard

Although MTConnect provides many benefits as a standard, it also has several limitations. Communication between numerical controls involves bidirectional data flow. However, MTConnect is a read-only protocol because the data comes only from the device. Both client-side and server-side devices are required to achieve bidirectional information flow between devices (Xiao, Huang, & Zhao, 2018). An important area of Industry 4.0 is the ability to communicate between machines. However, MTConnect cannot communicate with each other or read information from other devices. This is due to the nature of data exchange in the architecture of this protocol, as the adapter can only communicate and provide local variables for the device. Therefore, it lacks functionality as a standalone program and requires the use of another service (Parto, 2017).

MTConnect currently only provides data in XML format. XML is a human-readable and machine-readable web format. However, this format is so large and so complex to analyze that it makes it difficult to develop and integrate IoT applications. JSON, on the other hand, is not only human and machine readable, but also lightweight and very easy to parse. Therefore, most web apps and APIs communicate in JSON, and MTConnect is not a viable option for integrating these services (Parto, 2017).

2.4.3 Other Interoperable Approaches

H. Derhamy et al. developed a multi-protocol solution for IoT interoperability issues (Derhamy, Eliasson, & Delsing, 2017). The solution includes protocol implementation translators based on

Service Oriented Architecture (SOA), intermediate format to reduce the number of translations necessary. The system also detects protocol incompatibilities and perform the translation. Barros et al. introduced Internet of Things Multiprotocol Message Broker (IoTM2B) strategy to integrate various communication protocols such as HTTP, MQTT and CoAP and their performance evaluation based on two scenarios, machine-to-machine (M2M) protocols Communication and cloud-based environment (Barros et al., 2019). This strategy extends IoT DSM to provide integration with embedded devices Via various protocols. Derhamy et al. proposed interoperability solution consists of a multi-protocol translator that is injected into the service exchange on demand (Derhamy, Rönholm, et al., 2017). The main contribution of this research is to suggest ways to map OPC UAs to intermediate formats. Intermediate formats can be mapped to other standard IoT protocols such as CoAP, HTTP, and MQTT.

2.5 Research Gap

Given the increasing importance of interoperability in IIoT, solutions that can automatically integrate and analyze data across systems are essential. Interoperable solutions facilitate the rapid creation of IoT applications across platforms and domains, eliminating the need for application developers to own or operate an IoT infrastructure or platform (Zarko, 2019). Typical architecture in production environments includes numerous devices, sensors, and gateways that potentially communicate via different protocols. This sort of design works well when there are only a few systems to integrate. However, it leads to a difficult-to-maintain architecture when there are a larger number of components since the systems are connected point-to-point and the components are hardwired together. Modern architectures require more flexibility. Many manufacturing companies look for interoperability, adaptability, flexibility, and ease of implementation for the cyber-physical systems. The information exchange between the heterogeneous systems imposes the need for an interoperability solution.

Thus, a functional decomposition of such solutions leads us to the following main issues. Different systems use different communication mechanisms (OPC UA, MQTT, HTTP, CoAP) to provide or consume data. The system uses a variety of data formats such as OPC UA data models, plain bytes, tabular CSV, and Excel files. Different data semantics coexist. In fact, each system has its own semantics in generating or interpreting data. Communication protocols for propagating data, such

as request / response and publish / subscribe, use different interaction paradigms. The main issue with current interoperable solutions is that there isn't a method that fits well with integration of IoT protocols in a gateway and effective interoperable communication to interconnect the sensors, IIoT devices, and machines and cloud integration for compatible platforms.

To overcome the challenges with current interoperable solutions, we noticed from the literature review that an IoT gateway would play a vital role to enable data communication using different network protocols which can establish bridging among heterogenous devices.

2.6 IoT Gateway

With the rapid development of the Internet of Things (IoT), there is an increasing demand for ubiquitous connectivity to integrate multiple heterogeneous networks such as Zigbee ad hoc networks, wireless LANs, and wired networks. In general, IoT Gateway bridges various sensor and discovery domain networks with public or local area networks to support communication with different communication protocols and data formats. Therefore, an IoT gateway can connect multiple nodes with multiple sensors through different networks, it also performs many other tasks such as this IoT gateway performs protocol translation, aggregating all data, local processing, and filtering of data before sending it to application domain, locally storing data and providing device security. At the same time, a gateway becomes an ideal device for network management functions, since while exchanging messages with the sensor nodes, it can map the network and establish a comprehensive knowledge of the network(Wilder, Jose, Harold, & Alvarado, 2021). Figure 2.3 exemplifies main characteristics of the IoT gateway(Wilder et al., 2021).

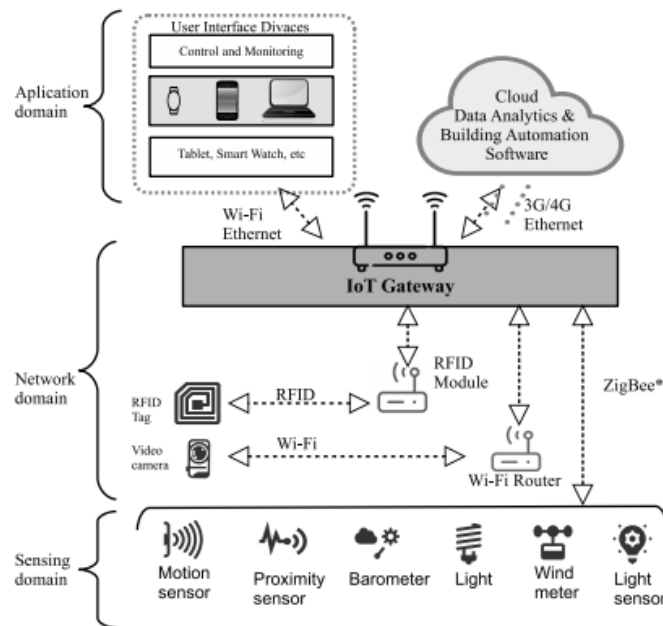


Figure 2.2 Main characteristics of the IoT gateway

Reprinted from (Wilder et al., 2021)

IoT gateways have been developed for industrial IoT applications. For example, (Dionisio, Malhao, & Torres, 2020) have developed a versatile, based on the Raspberry Pi platform. It can monitor critical parameters of the shop-floor factory through open-source software, both for smartphone and Desktop / Laptop computers, as well as storing data for remote analysis. They presented a gateway based on Raspberry Pi firmware and the OPC UA protocol for data transmission. They created an OPC UA to MQTT conversion module for connecting shop floor equipment and devices to an external cloud server and database.

Another self-configuration supported gateway is the one proposed in (Kang & Choo, 2018) which is designed for in-home-scale environments. They used the IoTivity framework to create the test bench. The proposed testbed's three server devices are based on a Raspberry Pi embedded system. The goal of this gateway is to ensure interoperability between devices that don't have IP-based communication capabilities. This gateway employed the CoAP (Constrained Application Protocol) protocol for device-to-device (D2D) communications to achieve this goal.

As a possible solution to interoperability problem caused by the heterogeneity of IoT devices (Wilder et al., 2021) proposed a IoT gateway. They established gateway acts as the hub of a paradigm in which multiple wireless nodes can send data using a variety of communication protocols, including Wi-Fi, Bluetooth, ZigBee, and Ethernet. They used the Samsung Artik 1020 development kit to create the gateway, which is a high-performance, multi-protocol embedded board with Bluetooth, ZigBee, and Wi-Fi wireless communication capabilities. They employed a wireless node made up of six sensors that measured environmental temperature, relative humidity, sun radiation, wind speed, rainfall level, and wind direction. The wireless nodes use ZigBee, Wi-Fi, and Bluetooth to send data from sensors to the gateway.

According to the needs of the IoT ecosystem and the current requirements of the IoT applications, IoT gateway must have the option to choose different protocols communication which need to be selected based on their applications and capabilities. Each of these protocols have distinct features and capabilities, which complex the identification of a protocol suitable for specific use cases. The CoAP protocol's hibernate architecture and binary data format make it ideal for applications related to automation, mobile phones, microcontrollers, and more. Another protocol widely used in IoT applications is MQTT. This is recommended for network scenarios where bandwidth consumption needs to be reduced and the processing and storage capacity of the devices involved in the communication is low (Wilder, Jose, Harold, & Alvarado, 2021). Like CoAP, WebSocket protocol's standard connectivity helps simplify many of the complexities and difficulties involved in the operation of bi-direction communication. This protocol can be applied to IoT networks where data is continuously communicated between multiple devices. However, one problem is that industrial and manufacturing sensors are primarily connected to programmable logic controllers (PLCs) to collect large amounts of sensor data and send it to communication systems (John & Vorbröcker, 2020). To enable IoT connectivity for these sensors connected to PLCs, the controller needs to configure with Modbus TCP protocol. There are some other devices which do not support any of these protocols and require to transfer large amounts of data. For these devices, HTTP protocol is widely used. For example, manufacturing and 3-D printing rely on the HTTP protocol due to the large amounts of data it can publish. Based on the capabilities and requirements of the sensor data transmission, users set access protocols for the bidirectional communication between nodes and gateway. In order to design interoperable connectivity, proper protocol needs to be

addressed initially in terms of seamless communication among different distributed cyber physical systems. We performed a study to for a better understanding of the industry specific communication protocols to design a protocol selection framework for the interoperable IoT gateway.

2.7 IoT Communication Standards

The Internet of Things covers a wide range of industries and use cases, from single constrained devices to large-scale cross-platform deployments of embedded technologies and cloud systems that connect in real time. A key issue here is the architecture and platform used by the machines and software packages. A better understanding of the subject can be achieved by studying industry-specific communication protocols and their respective logical semantics (Zeid et al., 2019). Based on the IoT devices and their applications, there are different application-layer IoT protocols such as MQTT, CoAP, AMQP, HTTP etc.

2.7.1 MQTT

MQTT, which was first launched in 1999, is one of the earliest M2M communication protocols. It was created by IBM's Andy Stanford-Clark and Arlen Nipper (Eurotech). It's a lightweight M2M messaging protocol built for limited networks that uses publish/subscribe messaging (Bandyopadhyay & Bhattacharyya, 2013). Designed to support remote monitoring, it provides low latency, secure messaging, and efficient delivery of data to one or more recipients over vulnerable networks. It is TCP-based and asynchronous and can integrate a publish-subscribe communication model.

Advantages of data transfer MQTT is good, reliable, easy to build. It uses less network resources even in conditions of unstable (Uy & Nam, 2019). The overall structure of the data transmission system using the MQTT protocol is shown in Figure 2.3

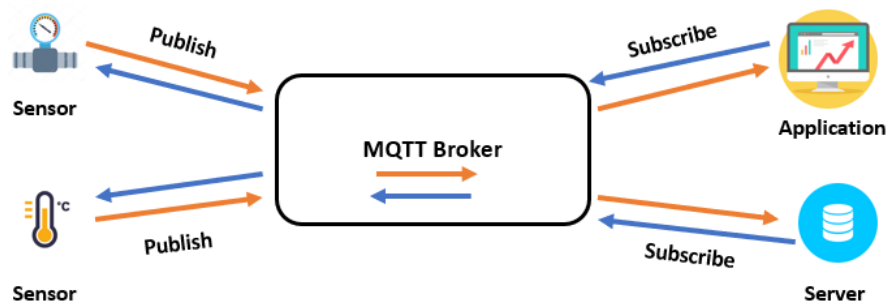


Figure 2.3 Overview structure of data transmission system using MQTT protocol

The MQTT client publishes the message to the MQTT broker. This broker can be subscribed to by other clients and retained for future subscriptions. Each message is published to one address. Customers can subscribe to multiple topics and receive all messages published on each topic. If the message is sent to the broker but the client has not yet subscribed, the packet is not stored in the broker and waits for the client to send it. Another great feature of MQTT is the three levels of quality of service (QoS) to ensure the delivery of messages. QoS Level: QoS0-Maximum once, QoS1-At least once, QoS2-Exactly once.

2.7.2 CoAP

Constrained Application Protocol (CoAP) is a service layer protocol used by resource-constrained low-power sensors and devices connected over a lossy network, especially when there are many sensors and devices on the network. CoAP is one of the most popular IoT communication protocols, especially in the context of advanced metering and distributed intelligence applications, as it extends the scope of HTTP to restricted devices (Silva, Carvalho, Soares, & Sofia, 2021). Many manufacturers use CoAP in their IoT devices since it is lightweight and energy efficient. CoAP is based on a client/server approach and uses REST to increase interoperability. Its design has been carefully worked to fit constrained devices in terms of battery, memory, storage. Running over UDP. Its lightweight design makes it a promising protocol for embedded devices. While widely available and highly interoperable, also providing inbuilt support for content negotiation and discovery, CoAP remains a one-to-one protocol based on a client/server model (Silva et al., 2021).

The methods supported in CoAP are based on the RESTful structure which are listed as follows:

- GET: operation to retrieve representation in resource identified by the URI request.
- POST: request that the server build a new subordinate resource with the parent URI specified.
- PUT: requests that the enclosed message content be used to update/create the resource indicated by the request URI.
- DELETE: requests resource identified by the request URI to delete. URI in CoAP is similar to HTTP, where the unsecure URL starts with CoAP:// and the secure URI is CoAPs:// (Alghamdi, Lasebae, & Aiash, 2013)

2.7.3 HTTP

HTTP supports a RESTful web architecture for requests / responses. Like CoAP, HTTP uses a Universal Resource Identifier (URI) instead of a topic. The server uses the URI to send the data, and the client uses the URI to receive the data. HTTP is a text-based protocol that does not define the size of headers and message payloads, but depends on the web server or programming technology (Naik, 2017). HTTP uses TCP as a default transport protocol and TLS/SSL for security. The HTTP protocol uses the GET, POST, PUT, and DELETE commands to exchange/remove data between the client and server. The REST architecture is shown in Figure 2.4 below.

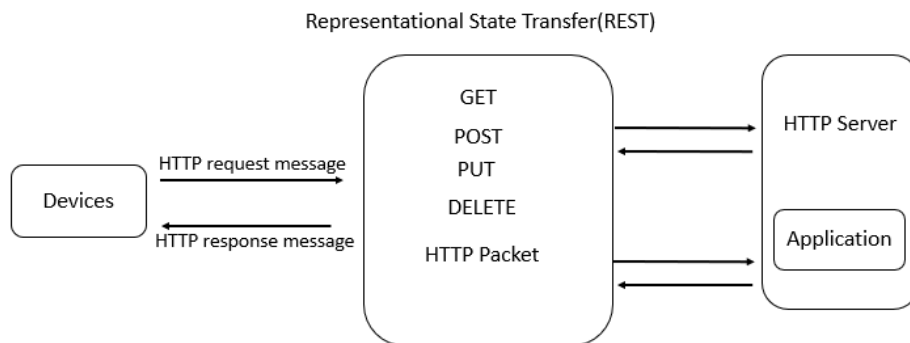


Figure 2.4 HTTP protocol over REST architecture

2.7.4 WebSocket

The WebSocket protocol is an application layer protocol designed for continuous data exchange between clients and servers. This enables bidirectional data transfer in web sessions and

asynchronous communication, which is considered a viable alternative to HTTP polling. This means that both sides can send data at any time while the connection is established. The protocol is divided into two parts: handshake and data transmission. In the handshake, the client and server basically establish initial communication over HTTP and the port. The default is 80. In this first communication, the client requests a communication type update. It can exchange data using the WebSocket protocol after the validation is validated.

Since the WebSocket is a TCP-based protocol, it requires a TCP connection to be established between client and server before any WebSocket-based interaction can occur. Handshaking a TCP connection requires three messages to be exchanged between client and server. Three-Way handshake or a TCP 3-way handshake is a three-step process that requires both the client and server to exchange synchronization and acknowledgment packets before the real data communication process starts. At this point, two peers that have direct access to the plain TCP protocol can start sending application-specific payload data to each other. However, for WebSocket-based communication, WebSocket session need to be established. To establish a session, the client sends a WebSocket upgrade request to the server, which responds with a WebSocket upgrade response. From this point on, even web-based clients and servers can send and receive data in asynchronous full-duplex mode (Skvorc, Horvat, & Srblijic, 2014). Sequence diagram of a WebSocket session over TCP protocol is shown below in Figure 2.5

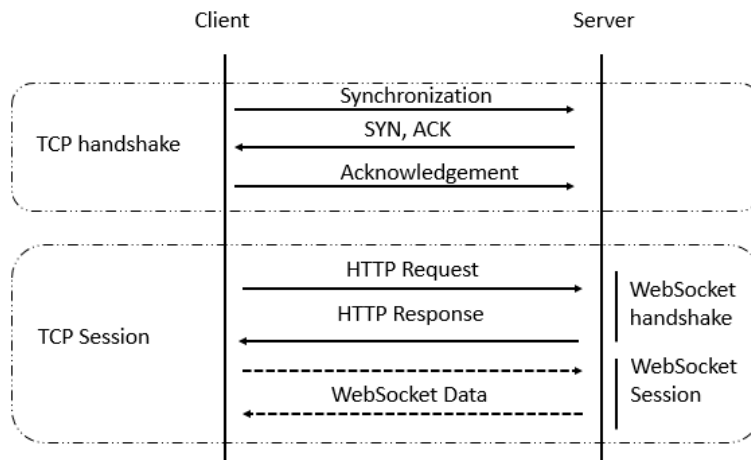


Figure 2.5 WebSocket over TCP sequence diagram

Reprinted from (Skvorc et al., 2014)

2.7.5 Modbus TCP

Modbus TCP / IP is one of the basic Modbus variations aimed at monitoring and controlling automated devices and is a simple vendor-independent communication protocol. Modbus TCP clients and servers listen and receive data over port 502. The Modbus / TCP protocol provides a client / server mode for communication between devices on an Ethernet network. There are four types of mode messages: Modbus request, Modbus acknowledgment, Modbus display, and Modbus response. A Modbus request is a message sent by a client to initiate an event. The Modbus display is a request message received from the server. A Modbus response is a response message sent by the server. A Modbus acknowledgment is a response message received from a client. The client-server architecture of basic Modbus TCP/IP communication is shown in Figure 2.6

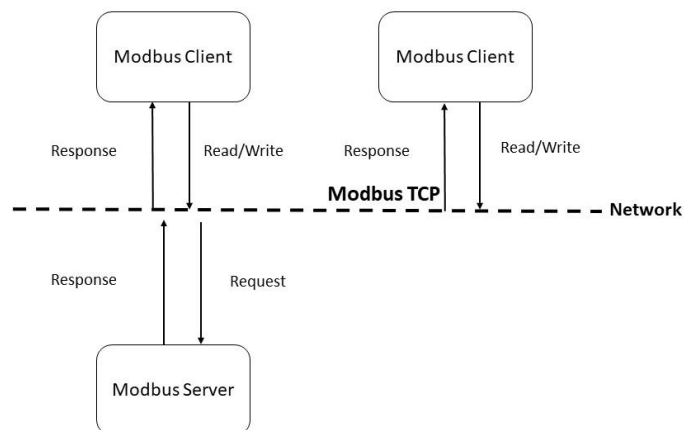


Figure 2.6 Client-Server architecture of basic Modbus TCP/IP communication

2.7.6 AMQP

The Advanced Message Queuing Protocol (AMQP) is a lightweight M2M protocol, designed for reliability, security, provisioning and interoperability (Foster, 2017). It follows a well-understood practices of data framing, client / server option negotiation, and connection processing. AMQP currently assumes stream-based transport (usually TCP). Sends sequential frames between channels, allowing multiple channels to share a single TCP connection. Therefore, communication

between the client and the broker is connection oriented. Reliability is one of AMQP's core features and provides two preliminary quality of service (QoS) levels for message delivery: Unsettle format (not reliable) and settle format (reliable).

2.8 Comparison of IoT Standard Protocols

This section presents a comparative analysis of the four widely accepted and emerging messaging protocols for IoT systems MQTT, CoAP, AMQP, HTTP, WebSocket and Modbus TCP based on several criteria to introduce their characteristics comparatively. The comparative study of these IoT standards and protocols is shown in Table 2.1.

Table 2.1 Comparison of Different IoT Standards and Protocols

Aspects	HTTP	CoAP	MQTT	AMQP	WebSocket	Modbus TCP
Communication approach	Client/Server	Client/Server	Client/ Broker, Client/Server	Client/ Broker, Client/Server	Client/Server	Client/Server
Messaging pattern	Request/Response	Request/Response	Publish/Subscribe Request/Response	Publish/Subscribe	Request/Response	Request/Response
Transport protocol	TCP	UDP, SCTP	TCP	TCP, SCTP	TCP	TCP
Security	TLS/SSL	DTLS	TLS/SSL	TLS/SSL SASL	TLS/SSL	

Table 2.1 Comparison of Different IoT Standards and Protocols (cont'd and end)

QoS/Reliability	Reliable (over TCP)	Confirmable /Non- Confirmable message	Guarante ed Message Arrival	Broker Redunda ncy	Reliable	Limited
Default Port	80/443	5683/5684	1883/888 3	5671/567 2	80/443	502
Binary Payload	No	Yes	Yes	Yes	Yes	No
Method	Get, Post, Put, Patch, Delete	Get, Post, Put, Delete	Publish, Subscribe	Publish, Consume	Bi- directional communica tion	Read/ Write Request
Message Size	Undefined	Small and Undefined	Maximu m 260MB	Undefine d	Should not exceed 64KB	Maximu m 255 bytes
Data Persistency	No	No	Yes	Yes	Yes	No

These messaging protocols are very extensive and different from each other because they have been evolved through different processes and needs. Interoperable integration and uniform access of these different IoT standards are required to provide seamless connectivity with different type of CPSs, devices, resources, and applications. Furthermore, organizations and current cloud-based

platforms provide expensive interoperable and compatibility tools which are costly investment for small manufacturing companies. Therefore, it is important to propose a communication protocol selection framework and a low-cost interoperable IoT system to interconnect IoT objects and heterogenous devices and machines with different access protocols to support interoperability across the small and medium enterprises.

2.9 Literary Review Conclusion

Chapter 2 presented challenges implementing interoperable communication, existing interoperable systems, and their limitations such as OPC UA, MTConnect. It also presented gateway based interoperable systems facing IoT challenges for different protocols. The chapter also discussed multiple access protocols and their characteristics to find suitable protocol for different applications and machine to machine communication. These concepts had to be reviewed to understand the objectives which will be listed in the next chapter.

CHAPTER 3 RESEARCH METHODOLOGY

It is clear that the main challenges of the current communication architecture are the smooth integration and interoperability of disparate communication standards, which are already supported by Internet of Things devices and sensors created expressly for certain purposes at various periods. However, it is important to develop an integrated solution, where heterogeneous CPSs, sensors, devices are uniformly made discoverable, given the ability to connect with other entities, and closely integrated with Internet infrastructure and services which will enable M2M data optimization and robustness in manufacturing industries. To overcome the challenges developing an interoperable solution, the problematic need to be addressed. This chapter proposed the research objectives and research design based on the problem statements and literature review defined in the previous two chapters.

3.1 Research Objectives

The objective of this research project is to demonstrate an IoT interoperable system for M2M communications among heterogeneous cyber physical systems with low computational capabilities. The system includes a protocol selection framework for connecting different devices and systems with different applications and capabilities and cost-effective gateway for connecting heterogeneous devices or nodes with different access protocols, bridge the communication and perform the conversion. For industrial communication in automation technology, industrial networking protocols such as HTTP/HTTPS, MQTT, Modbus RTU & Modbus TCP/IP, CoAP, WebSocket, AMQP are widely used. To demonstrate an interoperable system to communicate with devices with different protocols, a common platform is required which can access multiple protocols.

Using the above data as inputs and studied variables, we will be working towards two methods.

- Defining a protocol selection framework among HTTP, MQTT, CoAP, WebSocket Modbus TCP protocol based on the capabilities and requirements of the device and sensor data transmission.
- Developing the interoperable IoT multiprotocol conversion system in a low cost IoT gateway with above mentioned common access protocols for M2M communications among heterogeneous cyber physical systems with low computational capabilities.

The next section will present the methods and design procedure used to accomplish the objectives set above. The objectives indicate to design an interoperable middleware that can connect multiple devices, nodes and sensors with different access protocols and enable duplex communication between them. According to this vision, the next section presents research framework of this work consisting into the design of protocol selection framework and interoperable gateway architecture for seamless interoperable connectivity between heterogenous devices and machines via access communication protocols.

3.2 Research Design

Considering the interoperable infrastructure required for Industry 4.0, this thesis presents an interoperable solution of multiprotocol gateway. The objective of the proposed platform is to develop an interoperable system which is able to connect heterogenous devices with different protocol, process and store the data, exchange the data to different machines and to the cloud. The main contribution of the research is proposing protocol selection framework among HTTP, MQTT, CoAP, WebSocket, Modbus TCP and low-cost IoT gateway for developing effective interoperable M2M communication and cloud integration for compatible platforms. The contents in this section are categorized in two levels of protocol selection framework and gateway. The protocol selection framework presents a diagram for selecting different protocols based on the user end application requirements. The gateway section shows the general architecture of the platform and explains the structure of data exchange, data formatting and protocol bridging within the platform.

3.2.1 Protocol Selection Framework

To facilitate safe and high-speed data transfer among end IoT devices, protocol selection framework is designed from the understanding and study about communication protocols discussed in literature review among MQTT, CoAP, HTTP, WebSocket and Modbus TCP for the nodes and/or sensors to send data to the gateway. The diagram provides general automated system which allows users to select the suitable protocol for the application in regards of their standards and data transmission capabilities. In the Figure 3.1, protocol selection block diagram is shown to select the appropriate protocol for the IoT devices.

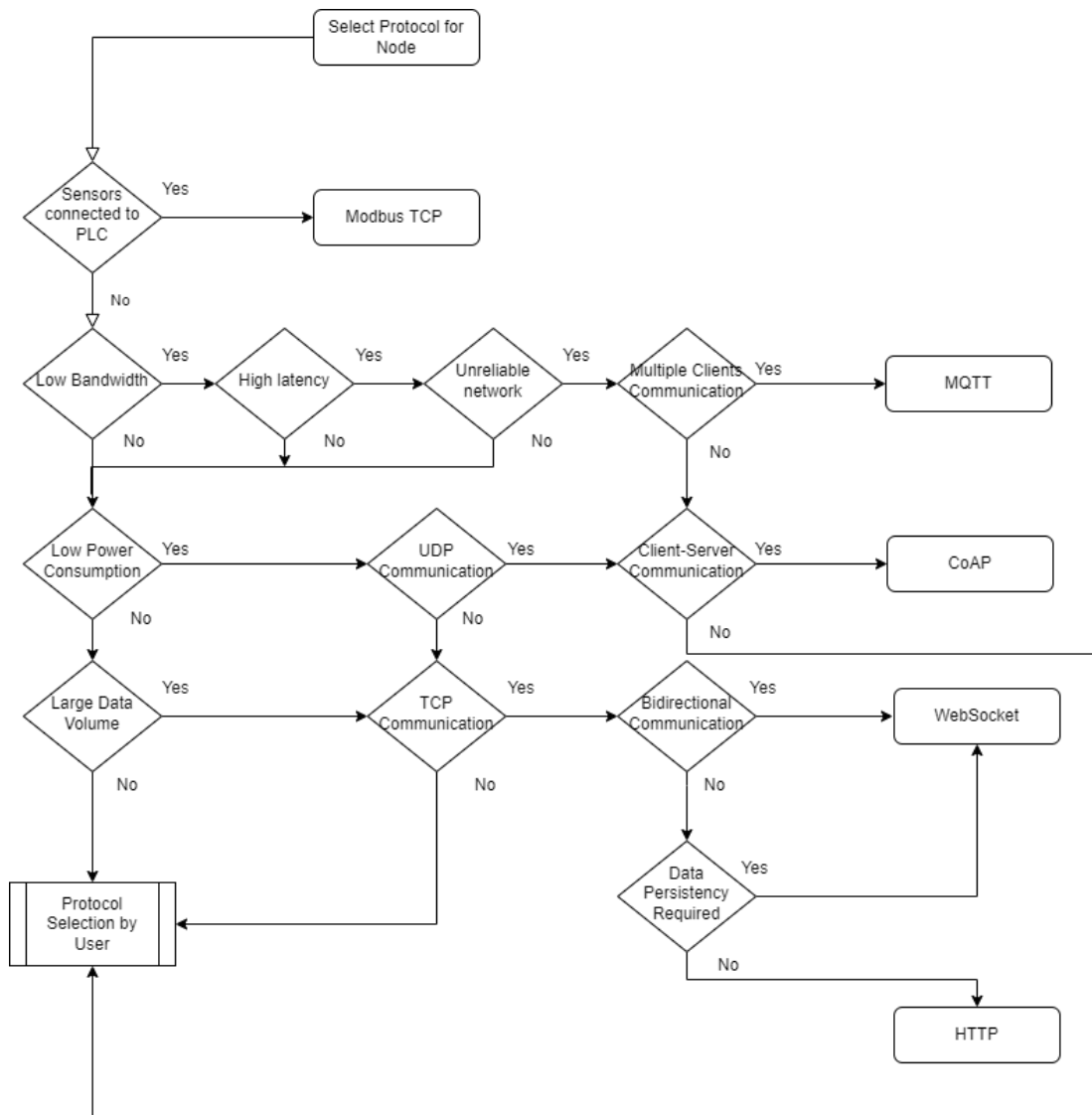


Figure 3.1 Protocol selection block diagram for applicable IoT systems

3.2.2 Gateway Design

The second part of the proposed objective is the middleware gateway which is a high performance and multi-protocol embedded board that is also accessible through LAN network and wireless communications. This gateway provides wide range of access capability, protocol interworking and interoperable managing and controlling of the sensor nodes. The gateway uses wireless communication protocol (e.g., Zigbee, Bluetooth, Wi-Fi) and LAN network to acquire the packet from the heterogenous sensor nodes, and use the 3G/4G, DSL and other network interfaces to send

the packets to the database server and Internet. The architecture of the gateway is defined with four modules. These modules are the key functions of the embedded communication.

3.2.2.1 Data Formatting

This module is responsible for formatting the collected data into a standard format and finally sending it to the database server. Heterogeneous nodes and sensors send data over different protocols, so they send data in different data formats. The gateway uses JSON format to systematize the representation of the data from the nodes and sensors. This format has important advantages such as simplicity and low resource consumption (Wilder et al., 2021). An example of the representation of the data recorded by the temperature and humidity sensor, in JSON format, is shown in Figure 3.2. As can be seen in this figure, the first three parameters are ‘node-id’, ‘protocol’ and ‘device’, which correspond to the identification of the remote node, connected sensors and device with the node and the communication protocol by which the sensor device send the data.

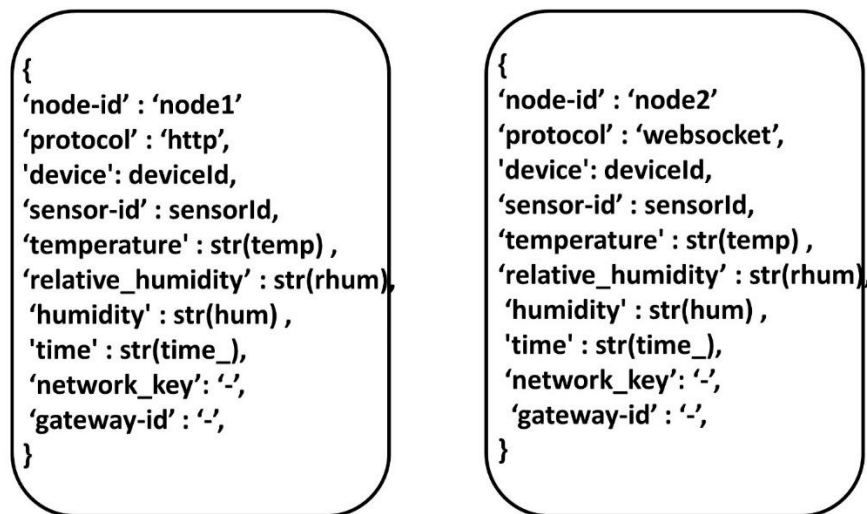


Figure 3.2 Example of systematized data format

The following parameters are sensor device’s fields ‘sensor-id’, ‘temperature’, ‘relative humidity’, ‘humidity’ and ‘time’, which collect the sensor information and data with the time. There are some other parameters for example ‘network_key’ which will be used to identify the communication network as Bluetooth, Wi-Fi, or others. Besides, the parameter ‘gateway-id’ which will be used

later to identify the gateway. The gateway identification will be useful when there is multiple gateways transferring information among each other.

3.2.2.2 Protocol Bridging

The proposed IoT gateway acts as a bridge between different protocols, mainly between HTTP, MQTT, CoAP, WebSocket and Modbus TCP. The gateway continuously being ready for listening for these multiprotocol connection requests and message payload with the standardized format.

In my proposed system, the gateway is acting as server with MQTT broker software contains and facilitates the communication from different nodes transferring messages from publisher to subscriber and subscriber to publisher. The broker installed on the gateway was Mosquitto, a well-known broker that implements several versions of the MQTT protocol and is a relatively lightweight software message broker (Light, 2017) and it has low power profile. In this regard, no data is initially sent through the Internet which is also an advantage in terms of security. This provides a minimal way of communication that does not require any cloud-based broker. The proposed bi-directional gateway can transfer the sorted data from the database and send to the application client and to the cloud with required protocol. Three threads are opened in the main process. The first one is to listen and accept on the multiple node connections. The second one is to read the commands from the software user interface module and read data from the IOT Gateway and parse the data, then send it to the MQTT broker for publishing. The third one is to show the received data on a webpage using Paho provided MQTT JavaScript client. It is also responsible for receiving configuration parameters that users can enter through the user interface and send to nodes and other gateways. In this case, in communication between gateways, the connected node and sensor act as subscribers and the gateway acts as publisher.

In the gateway, WebSocket communication technology also is adopted in MQTT broker as WebSocket provides full-duplex communication channels over a single TCP/IP connection. When the gateway starts, it creates a server socket which uses a particular port 80 for regular WebSocket connections. WebSocket server in the gateway, becomes ready for listening nodes configured with WebSocket protocol. When user configure node with WebSocket protocol for data transfer, the node creates client socket and tries to establish a communication link to the gateway server using its IP address and port number.

Another well-known protocol for communication between nodes and gateway servers is HTTP, which the user can configure. The server is installed on the local area network's IoT gateway. This server represents the data gateway server to which data from the nodes would be transferred. On the IoT gateway, an Apache web server is deployed and configured. HTTP is a request-response protocol that communicates between a client and a server. In the system, a node containing sensor devices makes an HTTP request to the server, which is subsequently answered by the server. The data given to the server via POST is saved in the HTTP request's request body. As a result, node sends data to the server gateway.

Another highly interoperable protocol for embedded devices with increased levels of security is CoAP. CoAP combined with size-optimized and reliable datagram communication. CoAP has two sublayers. H. Messaging sublayer and request / response sublayer. CoAP provides a URI like (coap://). This task installs a secure implementation of the CoAP server on the IoT gateway. Python 3 and the Aiocoap library are used to install the CoAP server on the gateway. In the proposed system, both the client and the server are co-located, communicate over the same network, and the gateway acts as a local server. A node that acts as a CoAP client can use a browser add-on (Copper (Cu) CoAP User-Agent) to send data to the server on a specific port 5683. The IoT gateway, which acts as a server, listens on specific ports for data received from clients. Nodes that use the CoAP communication protocol detect that the server is running at a gateway IP address that uses a particular port. After finding the service, the node sends a GET or PUT request to access the server. The gateway server reviews these requests and operates accordingly.

Modbus TCP server is also provided by the IoT gateway. A related feature of Modbus TCP is that it is supported by both proprietary and open source hardware / software, allowing different devices to seamlessly exchange the information made possible by this protocol (González, Calderón, & Portalo, 2021). The ability to interconnect older-type equipment for instance PLC plays an important role for industrial machine to machine communication. The IoT gateway uses pyModbusTCP, a Modbus/TCP client library for Python which gives access to Modbus/TCP server through the Modbus Client object. Modbus TCP/IP clients and server listen and receive Modbus data via port 502. When user set Modbus TCP protocol for the node, node acts as client and it initiates the communication by sending request to the IoT gateway server to transfer data. The IoT gateway server is aimed to build up the peer-to-peer communication that encapsulate/de-

encapsulate the incoming serial frames/TCP packages. These incoming serial data frames are read and written as holding register. The type of register being addressed by a Modbus request is determined by the function code. Once a connection is established, the gateway server imparts the node with the queried information until the node finishes the connection. Protocol bridging demonstration is shown below in Figure 3.3.

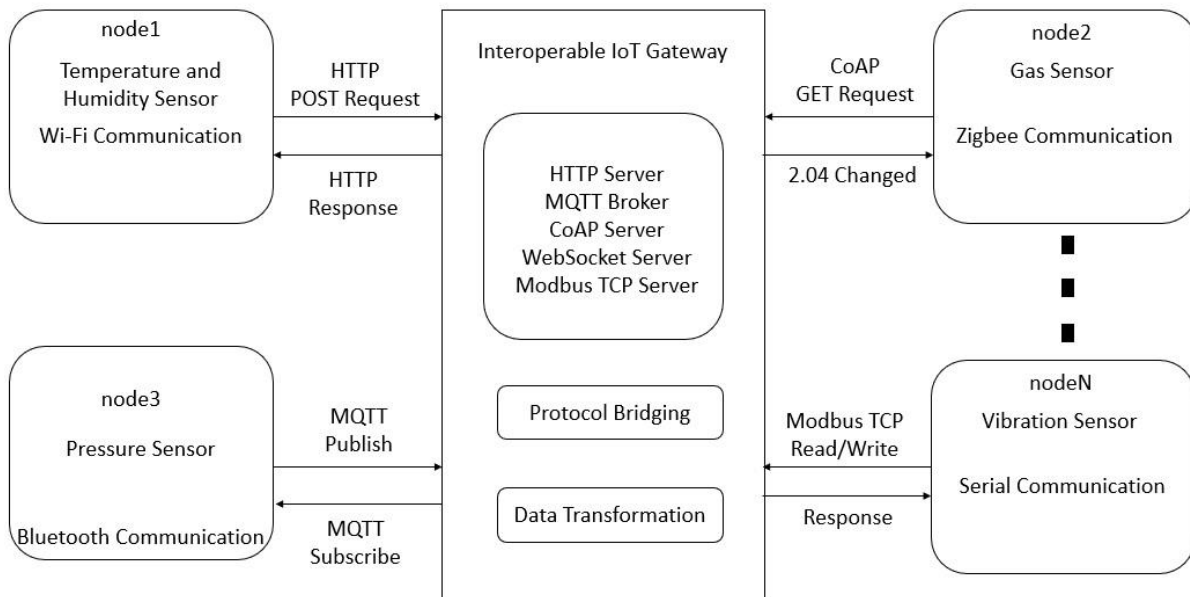


Figure 3.3 Protocol bridging for multiple nodes with different protocols

3.2.2.3 Interoperable communication among nodes and gateways

In the protocol bridging section, it is showed that multiple nodes with multiple protocol mainly MQTT, HTTP, WebSocket, CoAP and Modbus TCP send data simultaneously to the proposed IoT gateway. Another important function of the architecture is that the IoT gateway is able to communicate with the nodes and other gateways with the same protocol that nodes use to communicate with the gateway or different protocol in terms of the data type and data transfer requirements. For instance, node1 uses CoAP protocol to send data to IoT gateway. For sending payload from IoT gateway to node1, user can set any of MQTT, HTTP, WebSocket, CoAP and Modbus TCP protocol to communicate simultaneously. For the communication between multiple gateways, this protocol interoperability might play a significant role in industrial machine to

machine communication. An example is given below in Figure 3.4 to illustrate the interoperable protocol communication among the nodes and gateways in the proposed architecture.

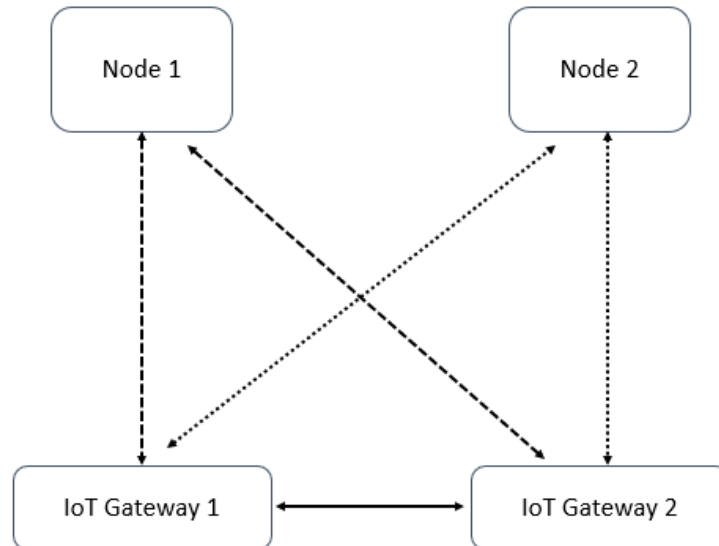


Figure 3.4 Interoperable protocol communication among the nodes and gateways

3.2.2.4 Data Process and Storage

The MySQL database management system can be used to record the time it takes to send and receive messages transmit over the network, where all data is stored after each session and record data is calculated. In the IoT gateway, in the local network can be configured so that when the nodes transmit data over this multiple protocol, after the data standardization process is done, primarily data are stored in local database with protocol-id and other values to identify the data for cloud and other database data transfer. Gateway data formatting methods play an important role in addressing the challenges of data integration from a variety of devices and systems for general systematization and representation in JSON format. To handle huge amounts of data and address data integration and data interoperability challenges, multiple databases can be integrated for high reliability and fast access to data. By adopting cloud-native services, developers can also take advantage of advanced technologies such as AI, machine learning (ML), and natural language processing (NLP).

3.3 Methodology Conclusion

In the research methodology chapter, we presented the steps taken to reach research objectives. An IoT protocol selection framework has been designed for the users to select suitable protocol for the application based on hardware capabilities and data transmission requirements. Also, we presented design of interoperable IoT gateway consisting of four modules including data formatting, protocol bridging, interoperable communication among nodes or devices and gateways, and data processing with storage.

CHAPTER 4 RESEARCH DEVELOPMENT

In order to evaluate the proposed solution for interoperability in Chapter 3, an IoT gateway architecture was implemented. The implemented system involves communication among IoT gateway and three remote wireless nodes. The nodes are connected with three different sensors. The gateway is designed to connect any IoT and industrial sensors. The three nodes and sensors are taken based on the availability and cost effectiveness. The solution proposed in this research aims at enabling interoperable connectivity from heterogenous devices and data acquired from different communication protocols and also extending these networks towards the IoT universe. The purpose of the application is to develop interoperability gateways which provides a reliable and uniform way of industry 4.0 cyber physical systems for easy interoperability. The proposed architecture is designed, implemented, and verified to evaluate quality of service of the gateway. To this purpose, the following modules have been implemented within the application: This chapter is consisting of 6 parts: (i) multi-protocol gateway development(ii)multi-protocol server integration (iii)node microcontrollers and sensors integration (iv)nodes to gateway interoperable communication (v) bi-directional communication among nodes and gateways (vi) data storage to multiple databases. Figure 4.1 shows the implementation details of the architecture in this project.

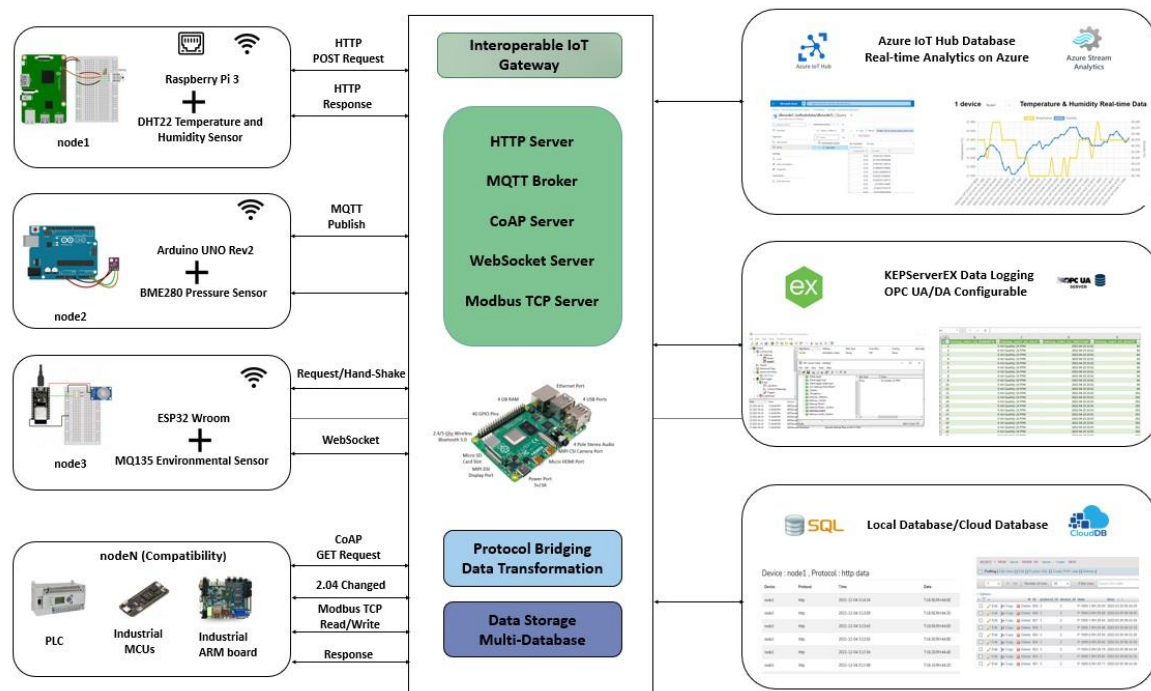


Figure 4.1 Proposed multi-protocol gateway architecture implementation framework

4.1 Multi-Protocol Gateway Development

For the development of the gateway, Raspberry Pi 4 development board was used, which is a high performance and multi-protocol embedded board. Initially, multiple single board computers were compared to find the cost efficient and complex task management compatible gateway. The table 4.1 is provided to show the comparison among Raspberry Pi 4, Raspberry Pi 3, Arduino Mega, Beagle Bone and Intel Galileo.

Table 4.1 Comparison of Efficient Gateways for Smart IoT Environment

	Raspberry Pi 4	Raspberry Pi 3	Beagle Bone	Intel Galileo
Processor	Quad core 64-bit ARM-Cortex A72	Broadcom BCM2837 64bit Quad Core	ARM Cortex-A8	Quark SoC X1000, 32-bit Intel
Frequency	1.5GHz	1.2GHz	1GHz	400MH
RAM	4GB	1GB	512MB	512 KB on-chip SRAM 256Mb DRAM
Operating System	Raspbian, Debian, Fedora, ARCH Linux ARM, RISC OS, Ubuntu Core et.	Raspbian, Debian, Fedora, ARCH Linux ARM) and FreeBSD Etc.	Android, Debian, Angstrom, Yocto, Fedora, Ubuntu etc.	Arduino, Linux distribution for Galileo, Rocket etc.
Power	15.3W	10W	15W	15W

Table 4.1 Comparison of Efficient Gateways for Smart IoT Environment (cont'd and end)

Cost	70 CAD	45 CAD	70 CAD	100 CAD
------	--------	--------	--------	---------

Based on the comparison of multiple single board computer, Raspberry Pi 4 has been selected as IoT gateway to implement the proposed architecture. As in the research work, it is proposed that heterogeneous nodes with different sensors communicate with the IoT gateway over different protocols and for the interoperable communication compatible functionalities, Raspberry Pi 3 Model B is preferred. Raspberry Pi is proposed as IoT gateway because of its low cost, high processing capability, sufficient amount of random-access memory (RAM), 40 input/output GPIO pins, RJ45 port and Wi-Fi connectivity for smart Internet of Things environment. In this research, MQTT broker, HTTP, WebSocket, CoAP, ModbusTCP server are implemented in Raspberry Pi 4. The Pi 4 is continuously being ready for listening for these multiprotocol connection requests and message payload with the standardized format through 802.11 b/g/n/ac Wireless LAN network. Figure 4.2 shows the Raspberry Pi 4 4GB model which is used in this research for the proposed system development.

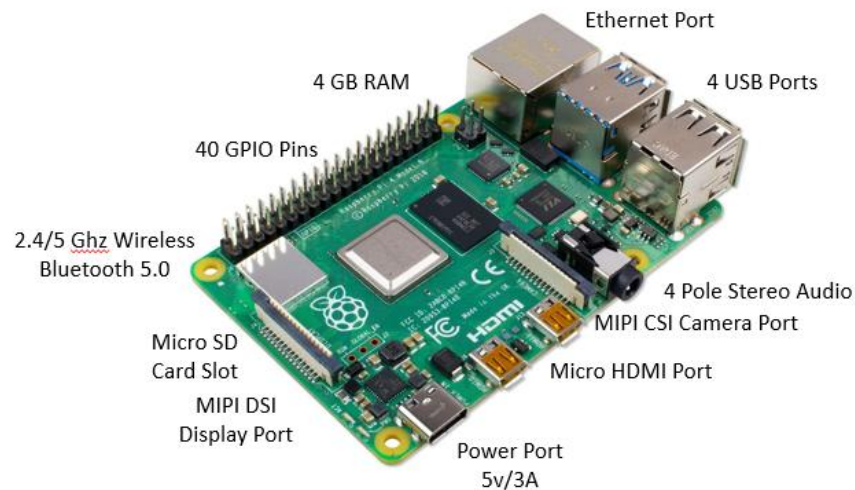


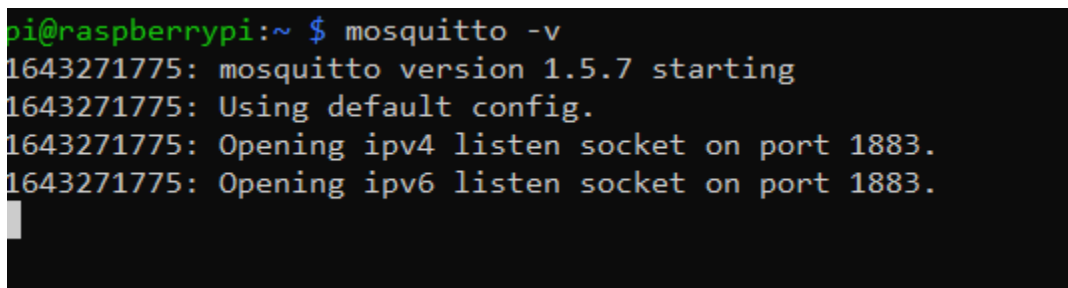
Figure 4.2 Architecture of Raspberry Pi 4 Model B

4.2 Multiple Server Configuration on the Gateway

MQTT, HTTP, CoAP, Modbus TCP and WebSocket server are integrated in the gateway. In our experiments we use a high speed, 32 GB class 10 (30 MB/s), micro-SD card for storage. This ensures that the storage device is not the system bottleneck. Raspbian OS has been installed in the system to configure all the servers and install the required software and libraries.

4.2.1 MQTT Broker Configuration

The software that is being used here is Mosquitto MQTT broker software, which is installed on the Raspberry pi by using command “sudo apt-get install mosquitto mosquitto-clients”. By installing it on the raspberry pi and starting it as a MQTT broker which handles the messages. Installation set up the broker and allow it to start on boot. After installation series of commands in the terminal of Raspberry username and password can be set and when everything is established, Mosquitto software broker will be started by using command “mosquitto /etc/mosquitto/mosquitto.conf”. And by following this sequence the clients are ready to connect to the broker to publish or subscribe the topic. After these installations, following command “mosquitto -v” on the terminal shows Mosquitto running on the system and in what ports the server is ready to listen connection requests. Figure 4.3 illustrates the correct installation and start-up Mosquitto MQTT broker on the terminal.



```
pi@raspberrypi:~ $ mosquitto -v
1643271775: mosquitto version 1.5.7 starting
1643271775: Using default config.
1643271775: Opening ipv4 listen socket on port 1883.
1643271775: Opening ipv6 listen socket on port 1883.
```

Figure 4.3 Mosquitto MQTT broker running on terminal

4.2.2 HTTP Server (Apache Web Server)

The Apache web server (HTTP server) is installed on the Raspberry Pi 4 IoT gateway. Apache can communicate between nodes and the server over the HTTP and HTTPS web protocols. Apache server has been installed with the command “sudo apt install apache2 -y”. The Apache server uses

HTTP (Hyper Text Transfer Protocol) to distribute website services online and supports four application profiles: Apache, Apache Full, Apache Secure, and OpenSSH. The Apache profile opens the port 80 (http traffic), while Apache secure opens only port 443 (SSL/TLS traffic). Because in this work both HTTP and HTTPS is used, Apache Full is enabled as it opens both port 80 and port 443. Figure 4.4 demonstrates apache2 Raspbian version running on the gateway.

```
pi@raspberrypi:~ $ apache2 -v
Server version: Apache/2.4.38 (Raspbian)
Server built:   2021-09-30T03:50:49
```

Figure 4.4 Apache3 Raspbian version webservice installation confirmation

4.2.3 Modbus TCP Server

Modbus is more suitable for rapid system development because it allows data and commands to pass between the two devices without requiring the knowledge of how data is processed or how outside communication is implemented. For Modbus TCP/IP communication, an open-source and full Modbus protocol called “pyModbus” is used. It works as fully implemented Modbus server and supports read/write on discrete and register. In the proposed communication architecture, the Raspberry Pi is configured as the Modbus TCP server. In the Raspberry Pi, a Modbus TCP/IP context must be generated using its IP address before trying to create a connection to the nodes. Before data communication over nodes, the gateway will create a register map for all data types with desired size. 30 registers are written initially for receiving read/write requests from the clients. More registers can be added for further development. Since the server cannot perform read or write operations in Modbus TCP/IP protocol, it only accepts read or write requests from the client and replies to a message to the client once the operation has been completed. Mapping of the holding registers for Modbus communication are illustrated in Figure 4.5.

```

if c.is_open():
    regs = c.read_holding_registers(0, 30)
    if regs:
        payload1 = str(regs[9])
        payload1 += "-" + str(regs[8])
        payload1 += "-" + str(regs[7])
        payload1 += " " + str(regs[4])
        payload1 += ":" + str(regs[5])
        payload1 += ":" + str(regs[6])
        payload1 += "|T:" + str(regs[2]/100)
        payload1 += ",RH:" + str(regs[3]/100)
        payload1 += "|node" + str(regs[1])
        if(exPayload1 != payload1):
            client_server.publish(mqtt_pub_topic1, payload1)
            exPayload1 = payload1

        payload2 = str(regs[19])
        payload2 += "-" + str(regs[18])
        payload2 += "-" + str(regs[17])
        payload2 += " " + str(regs[14])
        payload2 += ":" + str(regs[15])
        payload2 += ":" + str(regs[16])

```

Figure 4.5 Register mapping for Modbus communication

All the data that need to be exchanged between the gateway and nodes must be converted to registers as holding registers and input registers will be used. Each variable, regardless of integer or floating point, will be represented by 2 registers. It is easy to convert integer variables to such format. For floating points, they can be converted to integers just by multiplying the absolute value by 100 in order to preserve 2 decimals. The precision of the data can be easily adjusted based on either nodes requirement or data size limitation set by the Modbus register. Actually, both the gateway and the nodes are presented as actual data format for each Modbus register.

4.2.4 CoAP Server Implementation

In this work, the aiocoap Python CoAP library was used to implement the CoAP protocol (Maciej Wasilak & Amsüss, 2014). Simple CoAP server is installed with a single resource. The data of the nodes is stored in this resource. The first access method is PUT, which allows the node linked to the sensors to send data to the server running on the Raspberry Pi 4 IoT gateway. Furthermore, the GET method enables nodes with actuators to register with the resource, allowing nodes to be alerted when the server begins any feedback command or data transfer.

The PUT method requests that the resource identified by the request URI be updated or added with the enclosed payload. The message is triggered by assigning a PUT request to the resource. This

allows the process to notify the CoAP server of state changes in a simple and standard way. The resource is implemented as a special PUT handler that updates the resource status according to the PUT payload and triggers the delivery of the payload to the publisher. CoAP messages use two types of identifiers, message identifiers. This allows messages to be paired with acknowledgments and tokens for more general purposes. Figure 4.6 illustrates CoAP payload requests function for transferring from node microcontroller to Raspberry Pi 4 IoT server gateway.

```
class function_for_put_req(resource.Resource):
    def __init__(self):
        super().__init__()

        self.handle = None

    def notify(self):
        self.updated_state()
        self.reschedule()

    def reschedule(self):
        self.handle = asyncio.get_event_loop().call_later(5, self.notify)

    async def render_put(self, request):
        print('From Node put payload: %s' % request.payload)
        data_ = request.payload.decode("ascii")
        data_ = data_.split("@")
        return aiocoap.Message(payload=ret)
```

Figure 4.6 CoAP requests for node sensors

The GET method gets information about the resource identified by the request URI. A node with an actor waiting for a protocol response. When the server sends the payload with the response device ID, the node receives the request and checks the current status of the connected actuators communicating through the node's GPIO pins. In this task, the LEDs on the receiving node are connected via GPIO. For each payload status change request from the server, the node acts as a controller by changing GPIO.output (relay_pin, GPIO.HIGH) or GPIO.output (relay_pin, GPIO.LOW) according to the monitoring resource status change request. In Figure 4.7 trigger observe payloads are illustrated and invoked automatically by the GPIO library when the value on targeted GPIO pin changes.

```

async def main():
    protocol = await Context.create_client_context()

    request = Message(code=GET, uri="coap://" + ip + "/" + deviceId, observe=0)

    pr = protocol.request(request)

    r = await pr.response
    print("First response: %s\n%r"%(r, r.payload))

    async for r in pr.observation:
        print("Next result: %s\n%r"%(r, r.payload))

        pr.observation.cancel()
        break

    global state
    state = int(r.payload)
    if(state == 1):
        GPIO.output(relay_pin, GPIO.HIGH)
    else:
        GPIO.output(relay_pin, GPIO.LOW)

```

Figure 4.7 CoAP payload observing from GPIO output

The response obtained in the main function is a message like the request message, just that it has a different code. The response code is denoted in Python with some utility functions. For instance, in this work, successful response code is provided in Figure 4.8 is 2.04 which embeds in the successful 2.00 group.

```

2022-01-28 08:37:10 | T:17.80, RH:33.00
coap
4
Result: 2.04 Changed
'Message Received'

```

Figure 4.8 CoAP successful response code on message receiving

4.2.5 WebSocket Server Deployment

As WebSocket enables bidirectional communication in real time over the web, WebSocket server will be deployed in the gateway for horizontal interoperable communication. Node.js file is created to open the requested file and return the content to the client. Later on, socket.io is installed which is the client side WebSocket “library” needed to connect to a WebSocket server. It’s WebSocket

API will be used to communicate between server and the clients. After enabling Node.js html server, http does the handling requests and serving content and URL helps to parse requested URLs. The data is converted using the built-in Python JSON library. To make the request another package was imported on the script named "requests". With the help of requests.post() method the formatted payload was sent to the server and the response came from the server was also printed to the console. The script will wait for a specified number of time and then will repeat the process from the beginning.

4.3 Node Microcontrollers for different Sensor Integration

In our proposed framework, we have showed that we can add multiple nodes which can access to the gateway. For the development, we demonstrated three different nodes which establish communication with the gateway with different communication protocol. These nodes are embedded with three different sensors to present data communication among the nodes and gateway. Here in this research work, we are using the nodes so that sensors send the data it contains. Wi-Fi interface and GPIOs in microcontroller on the SoC can be used for general purpose as well, so our sensors and actuators can be directly connected to node.

4.3.1 Node1 as Raspberry Pi 3 and Sensor DHT22

The Raspberry Pi3 Model B has been selected as the wireless sensor node to provide an intelligent solution for real-time and efficient communication with the gateway. Equipped with the Broadcom BCM2387 chipset and 2GHz quad-core ARM Cortex-A53 (64-bit), the Raspberry Pi 3 Model B is an intelligent node for IEEE 802.11 b / g / n Wi-Fi and IEEE 802.11 Bluetooth communication. The processor features 1GB of LPDDR2 memory and a 40-pin GPIO header on the Pi, providing access to 27 GPIO, UART, I 2C, SPI, 3.3 and 5V sources. Each pin in the GPIO header is the same as its predecessor, Model B +. The sensor is connected via the GPIO pin of the microprocessor.

Accordingly, on this research, the sensor used is digital-output relative humidity & temperature sensor/module. Initially, temperature and humidity sensors which are widely used in IoT applications were compared: DHT22, DHT11 and DS18B20. The Table 4.2 is provided below to show the comparison:

Table 4.2 Comparison of Different Temperature and Humidity Sensors

Sensor	DHT22	DHT11	DS18B20
Measures	Temperature Humidity	Temperature Humidity	Temperature
Temperature Range	-40 to 80°C	0 to 50°C	-55 to 125°C
Humidity Range	0 to 100%	20 to 90%	~
Supply Voltage	3 to 6 VDC	3 to 5.5 VDC	3 to 5.5V VDC
Accuracy	+/- 0.5°C	+/- 2°C	+/-0.5°C
Communication	Digital via Single Bus	Digital via Single Bus	Digital via Single Bus
Minimum Response Time	2 Seconds	6 seconds	<10 seconds
Price Per Unit	7.29 CAD	6.32 CAD	7.79 CAD

The DHT22 has been selected because it has low power consumption, low price, better accuracy, fast response time and an acceptable humidity and temperature range than the other three sensors. DHT22 which is capacitive-type humidity and temperature sensor, utilizes exclusive digital-signal-collecting-technique and humidity sensing technology, assuring its reliability and stability. Figure 4.9 shows the module of the temperature and humidity sensor used in this research.

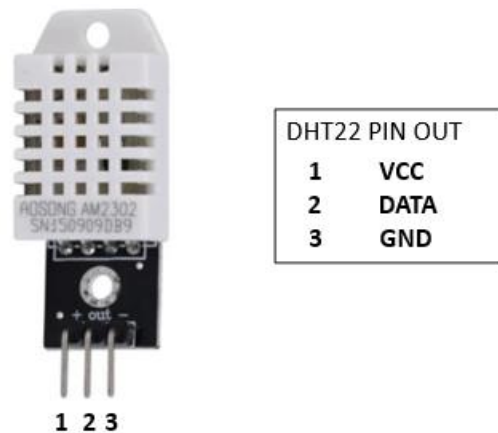


Figure 4.9 Pinouts of DHT22 temperature-humidity sensor

The temperature and humidity data are processed by the node (Raspberry Pi 3). When communication between RPi3 and DHT22 initiates, program of the node transforms voltage level of DATA BUS from high to low level and this process takes at least 1ms to ensure DHT22 could detect RPi's signal, then RPi waits 20 to 40us for DHT22's response. In this research, DHT22 sensor is connected using the Raspberry Pi3 through GPIO pins. The DHT 22 sensor has three pins: VCC, signal and GND. V cc pin is connected to Raspberry Pi GPIO (General Purpose Input/Output) pin of 3V, the signal pin is connected to GPIO 04 pin and GND pin of the sensor to the GND GPIO pin of the Raspberry Pi. The connection between DHT22 sensor and node raspberry pi 3B is shown in Figure 4.10.

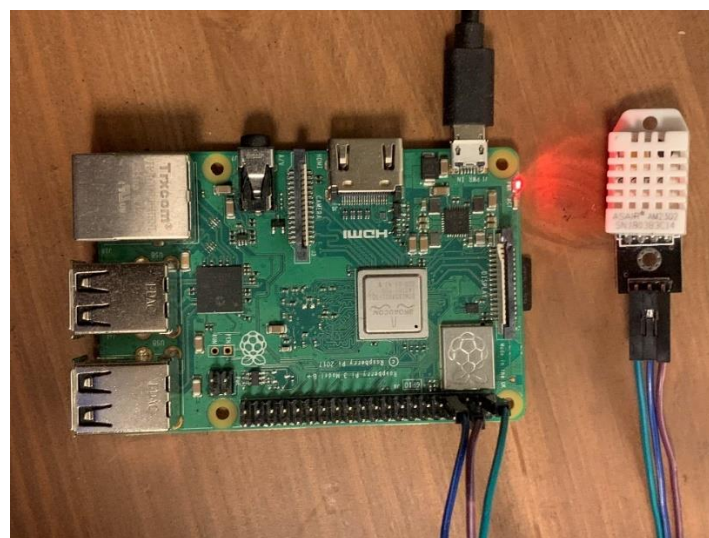


Figure 4.10 DHT22 sensor connection with node1 Raspberry Pi 3B

4.3.2 Node2 as Arduino Uno Wi-Fi Rev2 and Sensor BME280

For the integration of sensor with node 2, Arduino Uno Wi-Fi R2 is used which is ATmega4809 based microcontroller board integrated with Wi-Fi and Bluetooth module. Due to the presence of Wi-Fi connection ability, Arduino Uno Wi-Fi R2 is widely used for high performance and cost effective IoT applications. The Table 4.3 is provided to show the comparison different Arduino microcontroller boards to select the most efficient one for node2 application.

Table 4.3 Comparison of Different Microcontroller Boards

	Arduino Uno Wi-Fi R2	Arduino Mega 2560	Arduino Nano 33 BLE	Arduino Due
Processor	ATmega4809	ATmega2560	nRF52840 microcontroller	ATSAM3X8E Cortex-M3
Clock Speed	16MHz	16MHz	16MHz	84MHz
Memory	6KB SRAM, 48KB flash, 256 bytes EEPROM	256 KB of flash memory 8 KB of SRAM and 4 KB of EEPROM	32KB of program memory, 1KB of EEPROM, 2KB of RAM	512KB of ROM and 96KB RAM
Pins	14 digital I/O pins, 5 PWM channels, 6 analog inputs	16 analog inputs, 15 PWM channels	14 digital I/O, 6 analog inputs	54 digital I/O pins, 12 PWM channels, 12 analog inputs, and 2 analog outputs
Communication	I2C, SPI, UART	I2C, SPI, UART	I2C, SPI, UART	UART, USARTS, USB, I2C, SPI, CAN
Wireless Connectivity	Built in Wi-Fi, Bluetooth	None	Bluetooth	None

Table 4.3 Comparison of Different Microcontroller Boards (cont'd and end)

Shield Compatibility	5V	5V	3.3V	3.3V
Cost	53 CAD	48 CAD	28 CAD	48 CAD

Based on the comparison of different Arduino board, Arduino Uno Wi-Fi module has been chosen for implementation of node2 with sensor for its built-in Wi-Fi and 5V shield capability. The Arduino Uno Wi-Fi Rev2 is based on the 8-bit ATmega4809 microcontroller, and it has NINA-W102 u-Blox series Wi-Fi and Bluetooth module for wireless connectivity and ATECC608A high-security cryptographic microchip accelerator for implementing various authentication and encryption protocols. The microcontroller module is a self-contained SoC with a built-in TCP/IP protocol stack for network connectivity. The board has 14 digital input/output pins, 5 PWM outputs, 6 analogue inputs, a 16 MHz ceramic resonator, a USB connection, a power connector, an ICSP header, and a reset button, making it a strong controller for implementing communication through the gateway. The pinouts of the Arduino UNO Wi-Fi microcontroller are shown in Figure 4.11.

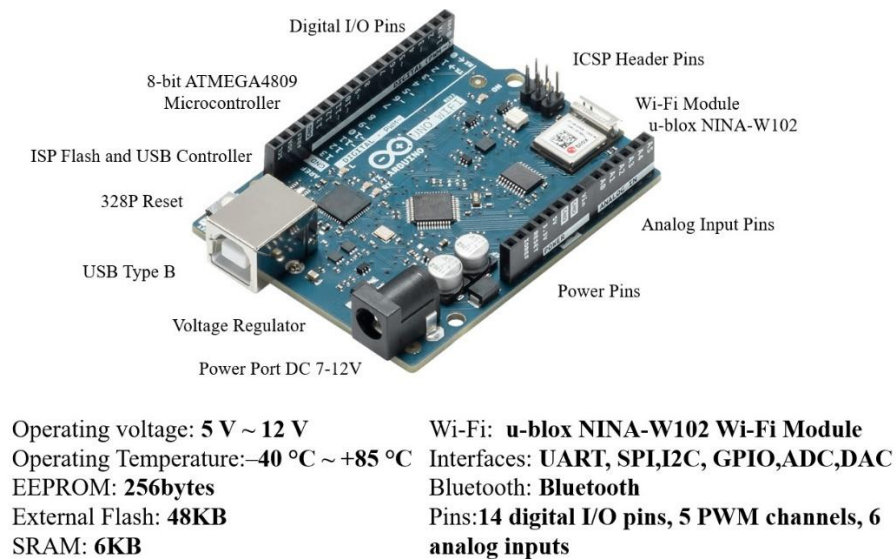


Figure 4.11 Arduino UNO Wi-Fi microcontroller with pinouts

The proposed system uses Arduino IDE (Integrated Development Environment) software to program the Arduino Uno Wi-Fi board. The Arduino IDE is an editor for compiling signal processing algorithms, compiling them into binary files, and then downloading them to the Arduino MCU via a USB serial port connection. Algorithm programming with a comprehensive suite of high-level, easy-to-use controls for I / O interfaces and peripheral configurations relies entirely on the connection and mapping of sensors to microcontrollers. The code is written in the IDE and uses the C / C ++ language to embed communication with the sensor. For this task, the Arduino microcontroller is programmed as a WebSocket client. The board collects the sensor's real-time measurements via the I / O interface and exposes the real-time data to the gateway server via socket communication.

For the integration of node2 sensor, bme280 is used which is an environmental sensor, which was designed for applications where size and low power consumption are crucial constraints. The sensor is able to measure pressure, humidity, temperature. The sensor supports performance requirements for emerging applications such as context awareness, and high accuracy over a wide temperature range. An additional advantage of the sensor is that the response time is extremely fast especially in the pressure measurement at very low noise. As a result, in this research work for node2 we are working only with the pressure data. Figure 4.12 demonstrates the pinout for the BME280 Pressure sensor.

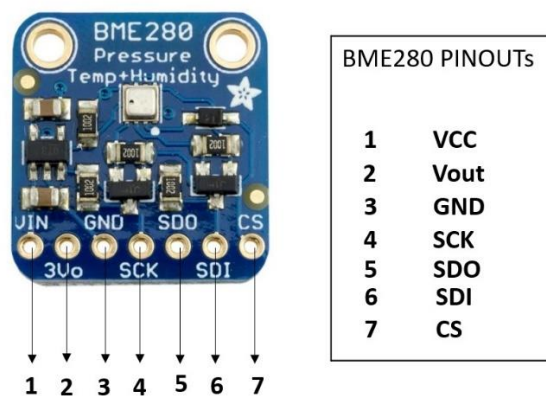


Figure 4.12 BME280 Pressure-Altitude sensor pinouts

In this research development, bme280 sensor's VCC input is connected through 5V output pin of Arduino UNO rev2. SDA (I2C1 Data) of the MCU is connected to sensor's SDA pin. The SDA

includes a fixed pull-up from 1.8k Ω to 3.3V. This is suitable for I2C bus communication. Since the sensor communicates over the I2C bus, the implementation of the wire library enables the I2C pins on the MCU. The Wire library uses a 7-bit I2C address that identifies the sensor device. The library uses a 32-byte buffer. Therefore, all communications must be within this limit. Figure 4.13 illustrates the setup of node2 with Arduino Uno W-Fi board and BME280 sensor.

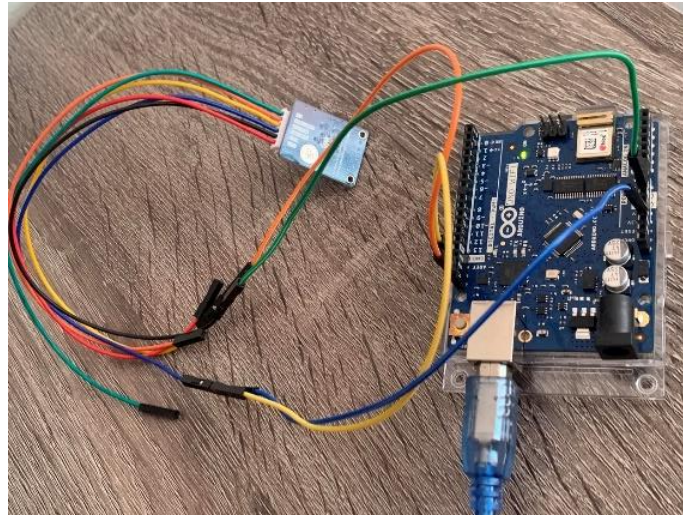
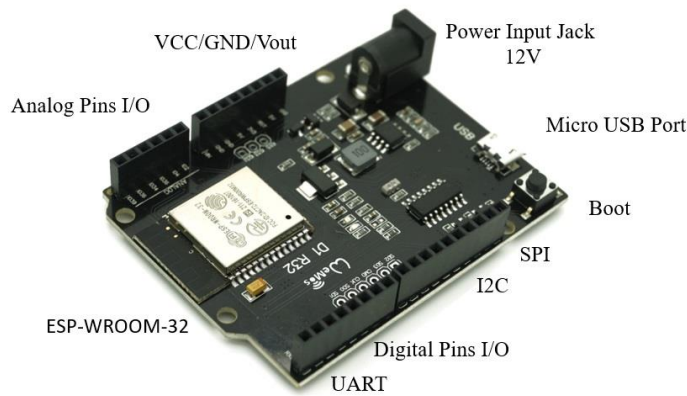


Figure 4.13 BME280 sensor connection wiring with Arduino UNO W-Fi microcontroller

4.3.3 Node3 as ESP32 and Sensor as MQ-135

For 3rd sensor integration, we used the ESP32-WROOM-32 board as node3. It is a high-performance general-purpose Wi-Fi and Bluetooth module used to implement a variety of applications, from low power sensor networks to the most demanding IoT applications. The Esp32 includes two low-power 32-bit Xtensa LX6 microprocessors with a chip quiescent current of less than 5 μ A, making it suitable for low-battery and portable electronics applications. This board was chosen for this development because it has 802.11 b / g / n (802.11n to 150 Mbit / s) compliant Wi-Fi capabilities. The board has analog and digital GPIOs that can be used to integrate the sensor with the analog output. Figure 4.14 demonstrates the architecture and GPIO pinout of the esp32 development board.



Operating voltage: **3.0 V ~ 3.6 V**
 Operating Channel: **2412 ~ 2484 MHz**
 Operating Temperature: **-40 °C ~ +85 °C**
 Booting ROM: **448KB**
 External Flash: **4MB**

Wi-Fi: **802.11 b/g/n (802.11n up to 150 Mbps)**
 Interfaces: **UART, SPI, I2C, GPIO, ADC, DAC**
 Bluetooth: **Bluetooth v4.2 BR/EDR**
A-MPDU and A-MSDU aggregation

Figure 4.14 ESP32-WROOM development board architecture and pinouts

In this research, ESP32 microcontroller is programmed with the Arduino software integrated development environment (IDE). The program is written in C++ and uploaded in the esp32 microcontroller through the open-source Arduino Software via USB cable. In this work, the ESP32 microcontroller is programmed as an MQTT client. The board collects the measured real-time values of the sensor, displays the values on the Arduino IDE Serial Monitor, and continuously publishes real-time data to the gateway server.

For the integration of node3 sensor, MQ-135 gas sensor has been deployed. Widely used in air quality meters, the MQ-135 sensor is suitable for detecting ammonia (NH₃), sulfur (S), benzene (C₆H₆), CO₂, smoke, and other harmful gases. The unit of air pollution is PPM (parts per million). To measure PPM gas, analog pins require to be used. The analog TTL operates and operates at 5 volts, so it can be easily integrated into node3's ESP32 MCU. Figure 4.15 shows the pinout of the MQ-135 environmental sensor.

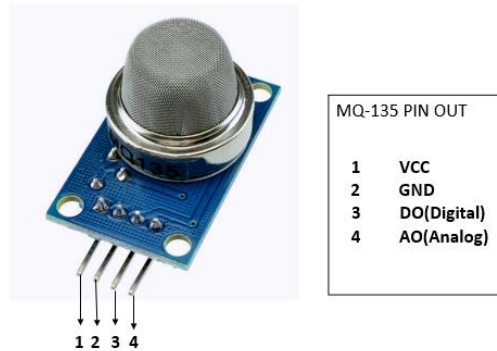


Figure 4.15 Pinouts of MQ-135 environmental sensor

The sensor which has very less latency is reliable and cost-effective implementation for node3 sensor device. The MQ-135 gas sensor uses SnO_2 , which has high resistance in clear air, as the material of the gas sensor. As the amount of harmful gas increases, so does the resistance of the gas sensor. Figure 3 is an excerpt from the MQ-135 data sheet, showing the typical sensitivity characteristics of the MQ-135 to a variety of gases measured at temperatures of 20°C . Humidity: 65%, O_2 concentration: 21%, $R_L = 20\text{ k}\Omega$, R_o = Sensor resistance at 100 ppm NH_3 in clean air, R_s = Sensor resistance at various gas concentrations. According to the sensitivity characteristics, R_s / R_o is the resistance ratio. The resistance R_L can be identified when the resistance sensor changes depending on the gas concentration (R_O) (Kant & Bhattacharya, 2017). Sensitivity characteristics of MQ-135 sensor is illustrated in the figure 4.16 below.

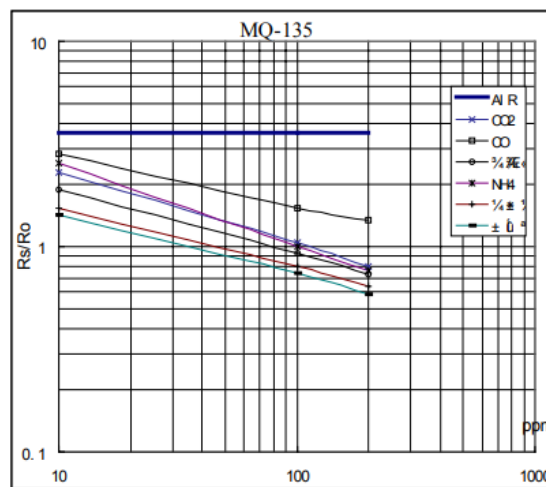


Figure 4.16 Sensitivity characteristics of MQ-135 environmental sensor

MQ135 Air Quality Sensor that can detect the level of various air pollutants. The AQI is an index for reporting daily air quality. It depicts how clean or polluted the air is, and what associated health effects might be a concern. The AQI is divided into six categories. Each category corresponds to a different level of health concern. Table 4.4 illustrates values of different Air Quality index and air quality status (Kinnera, Subbareddy, & Luhach, 2019).

Table 4.4 Different Air Quality Range with Status

Range (PPM)	Status
0-50	Good
51-100	Moderate
100-150	Unhealthy for Sensitive Groups
151-200	Unhealthy
201-300	Very Unhealthy
301-500	Hazardous

In this proposed development, MQ-135 sensor's VCC input is connected through 5V output pin of ESP32 WROOM board. Sensor's analog pin A0 is connected through the Analog pin of the microcontroller. ESP32 pulls the analog data of the sensor through the Analog pin A035. The GND Pin (Ground) of the sensor is connected with the MCU accordingly. The raw data fetched from the sensors is properly converted to PPM in the embedded code developed in Arduino IDE for ESP32 MCU. Figure 4.17 illustrates the experimental setup of node3 with ESP32 Wroom board and MQ-135 gas sensor.

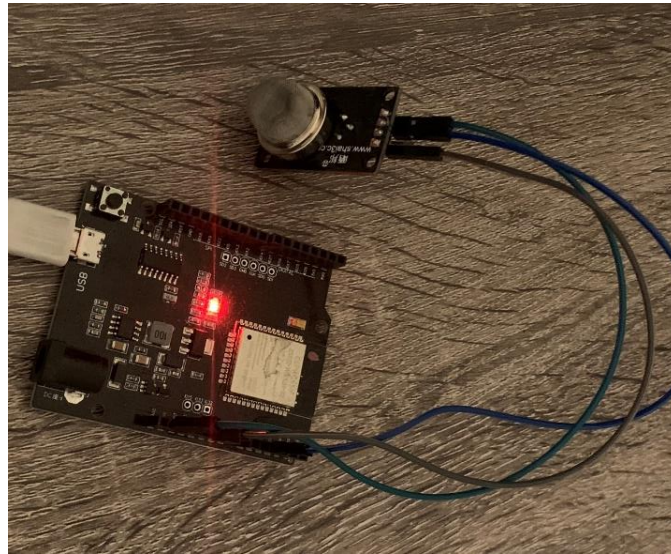


Figure 4.17 Experimental setup for node3 ESP32 and sensor MQ-135

4.4 Communication between node Microcontrollers and the Sensors

For the development of the proposed design, three different nodes and three different sensors are used. Sensors are connected through the GPIO pins and I/O interface of the node Microcontrollers. Three sensors DHT22 sensor, BME280 sensor, MQ-135 are connected to three node microcontrollers Raspberry Pi 3, Arduino Uno Rev2 and ESP32. These sensors collect the data from surroundings like temperature, humidity, pressure, air quality and send these accumulated data to nodes. The node microcontroller and microprocessor work on these data, process it and transmit the obtained results to gateway server through their communication protocol

4.4.1 Node1: Raspberry Pi 3 and DHT22 Communication

For node1, Temp-humid.py is a python script which is used for monitoring temperature and humidity using the DHT22 sensor. The DHT22 Temperature/Humidity sensor is connected through the GPIO pins of the node RPi 3. Adafruit GPIO Python library and the Adafruit DHT22 library are used to receive the data from the sensor. Figure 4.18 illustrates the sensor value of the DHT22 temperature humidity sensor acquired by the node Raspberry Pi on the command line.

```

pi@raspberrypi:~ $ sudo python temp-humid.py
Temp=21.5*C Humidity=44.2%
Temp=21.6*C Humidity=44.4%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.5*C Humidity=44.3%
Temp=21.6*C Humidity=44.4%
Temp=21.5*C Humidity=44.3%
Temp=21.6*C Humidity=44.3%

```

Figure 4.18 Node1 raspberry Pi 3 receiving data from DHT22 temperature-humidity sensor

4.4.2 Node2: Arduino Uno Wi-Fi Rev2 with BME280 Sensor Communication

In node2, Arduino Uno Wi-Fi Rev2 gathers data from the sensor BME280 using `pressure_altitude.ino` script on the Arduino IDE. The code has been run by including the needed libraries: the `Wire` library to use I2C, and the `Adafruit_Sensor` and `Adafruit_BME280` libraries to interface with the BME280 sensor. The `Wire` library implementation uses a 32-byte buffer; therefore, any communication should be within this limit. A variable called `SEALEVELPRESSURE_HPA` is created to save the pressure at the sea level in hectopascal. In the Arduino script, `bme.readPressure()` reads barometric pressure in hPa (Hectopascal = millibar) and `bme.readAltitude(SEALEVELPRESSURE_HPA)` estimates approximate altitude in meters based on the pressure at the sea level. To enable Wi-Fi connection with the network, `<WiFiClient.h>` library was set in the script on Arduino IDE. `<ArduinoHttpClient.h>` and `<Arduino-WebSocket-Fast.h>` libraries are added to the script to communicate through WebSocket protocol. Figure 4.19 shows the pressure and altitude acquired by the sensor on COM port 5 of the Arduino IDE.

The image shows a screenshot of an Arduino IDE. On the left, the code for a BME280 sensor is displayed. The code includes a `loop()` function that calls `printValues()` and a `printValues()` function that reads pressure and altitude data from the BME280 sensor and prints them to the serial monitor. The code is as follows:

```

pressure_altitude
}

void loop() {
  printValues();
  delay(delayTime);
}

void printValues() {

  Serial.print("Pressure = ");
  float MQ135_data=bme.readPressure() / 100.0F;
  float BME280_data=bme.readAltitude(SEALEVELPRESSURE_HPA);
  Serial.print(bme.readPressure() / 100.0F);
  Serial.println(" hPa");
  String pre= ("Pressure: ");
  String alt= ("Approx. Altitude: ");
  delay(2000);
  Serial.print("Approx. Altitude = ");
  Serial.print(bme.readAltitude(SEALEVELPRESSURE_HPA));
  Serial.println(" m");

  Serial.println();
}

```

On the right, the serial monitor window for COM5 is open, showing the output of the code. The output consists of several lines of data, including pressure in hPa and approximate altitude in meters. The output is as follows:

```

Approx. Altitude = 79.33 m
Pressure = 1003.75 hPa
Approx. Altitude = 79.38 m
Pressure = 1003.75 hPa
Approx. Altitude = 79.56 m
Pressure = 1003.75 hPa
Approx. Altitude = 79.29 m
Pressure = 1003.74 hPa
Approx. Altitude = 79.46 m
Pressure = 1003.76 hPa

```

At the bottom of the serial monitor window, there are two checkboxes: Autoscroll and Show timestamp.

Figure 4.19 Node2 Arduino UNO receiving sensor data from BME280 sensor

4.4.3 Node3: ESP32 with MQ-135 Gas Sensor Communication

For node3, MQ135 Air Quality sensor has been interfaced with ESP32 Wroom MCU. For compilation, AirQualityIndex.ino script has been written on Arduino IDE to communicate with the sensor and ESP32. Analog input A035 of the ESP32 was enabled to receive analog data from the sensor module. MQ-135 sensor module measured gas concentration in PPM. Multiple thresholds were added according to Air Quality Index chart (Kinnera et al., 2019) to determine the air quality status for the sensor data. `<wifi.h>` library with `setup()` function are included to start a connection to Wi-Fi network. As the sensor was configured to communicate with MQTT protocol through node, `<PubSubClient.h>` library has been added to the script. The PubSubClient library provides a client for publishing/subscribing message with a server that supports MQTT. Figure 4.20 shows air quality index received by the node on port COM4.

```

AirQualityIndex $
void setup() {
  Serial.begin(9600);
  delay(1000);
}

void loop() {
  delay(3000);
  int MQ135_data = (analogRead(anPin));
  Serial.print("Air Quality Index: ");
  Serial.print(MQ135_data);
  delay(500);
  Serial.print(" PPM");
  delay(2000);

  if(MQ135_data < MQ135_THRESHOLD_1){
    Serial.println(" Good Air");
    delay(500);
  }
  else if(MQ135_THRESHOLD_1 < MQ135_da
    Serial.println(" Moderate Air");
    delay(500);
  }
  else if(MQ135_THRESHOLD_2 < MQ135_data < MQ135_THRESHOLD_3){
    Serial.println(" Unhealthy Air for Sensitive Groups");
    delay(500);
  }
  else if(MQ135_THRESHOLD_3 < MQ135_data < MQ135_THRESHOLD_4){
    Serial.println(" Unhealthy Air");
    delay(500);
  }
}

```

COM4

```

Air Quality Index: 29 PPM Good Air
Air Quality Index: 29 PPM Good Air
Air Quality Index: 31 PPM Good Air
Air Quality Index: 30 PPM Good Air
Air Quality Index: 31 PPM Good Air
Air Quality Index: 30 PPM Good Air
Air Quality Index: 30 PPM Good Air
Air Quality Index: 29 PPM Good Air
Air Quality Index: 29 PPM Good Air
Air Quality Index: 31 PPM Good Air
Air Quality Index: 29 PPM Good Air
Air Quality Index: 30 PPM Good Air
Air Quality Index: 30 PPM Good Air
Air Quality Index: 29 PPM Good Air
Air Quality Index: 30 PPM Good Air

```

Autoscroll Show timestamp

Figure 4.20 Node3 ESP32 receiving sensor air quality data fromMQ-135 environmental sensor

4.5 Communication Protocol Selection for the Nodes

User assign communication protocol for the nodes connected with sensors to send data to the gateway. This configuration sets in what communication protocol the assigned node will communicate with the gateway. As the gateway is developed to receive any data over the MQTT, HTTP, CoAP, Modbus TCP/IP and WebSocket protocol, to visualize the data communication protocols need to be selected on the user interface. Figure 4.21 demonstrates protocol selection option for the nodes from the user interface.

Device Name	Data	Select Protocol
node1		Select
node2		Select
node3		Select

Protocol selection dropdown menu options:

- Select
- http
- MQTT
- Modbus_TCP
- WebSocket
- Coap

Figure 4.21 Protocol selection interface for the nodes to receive data on the gateway

Node1 is associated with Raspberry Pi. Due to node1's processing power capabilities as per the protocol selection framework demonstrated in previous chapter, HTTP client, CoAP client and Modbus TCP Client has been configured in the node1. Therefore, Node1 is able to communicate with any of these three protocols by enabling communication from the interface. Node1 was selected as HTTP, CoAP and Modbus TCP protocol separately from server end. As node2 is configured as WebSocket client, from the user interface of gateway WebSocket has been selected as data communication protocol. In this regard, gateway server was worked as WebSocket server and the node2 was worked as WebSocket client. Here, node3 is configured as MQTT client, so that node3 is able to publish messages to MQTT broker through MQTT protocol. From the user interface, MQTT protocol is selected for node3 to publish sensor data to the gateway server. Moreover, another significant function of the proposed system is that the gateway server is also able to interconnect with the nodes and other gateways with the same protocol that nodes use to communicate with the gateway or different communication protocol in terms of the data type and data transfer requirements.

Based on the capabilities of our proposed architecture, the gateway is able to run bi-directional communication. As a result, users are able to set access protocols for the communication between nodes and gateway. The software platform provides interface which allows users to select the suitable protocol for gateway to nodes payload transfer in regards of their standards and data transmission capabilities. For gateway to nodes communication, the interface let users to integrate nodes, also it allows users to set communication protocol required for the gateway to send payloads to nodes for machine control, feedback control and alarms. Figure 4.22 demonstrates the interface where user can set communication protocol for server to communicate with node's working access protocol.

Send Data

Device Name	Payload	Select Protocol	Action
node1	Select ▾	Select ▾	Send Payload
node2	Select ▾	http MQTT Modbus_TCP Websocket Coap	Send Payload
node3	Select ▾		Send Payload

Figure 4.22 Protocol assigning from gateway to nodes for sending payloads

4.6 Communication between Node Microcontroller and Gateway

The nodes are connected to the gateway. When user assign a protocol for the node to gateway communication. Node transfer sensor data to the gateway with the assigned protocol. In this development, node2 Arduino Uno with bme280 pressure sensor send data over WebSocket Protocol so that we enabled WebSocket protocol for gateway Node2 end protocol selection configuration. As node3 ESP32 send data over MQTT protocol, we permitted MQTT broker subscribing for node3 in the configuration end. For the experiment, we configured node1 communicating with HTTP, CoAP and Modbus TCP protocol so that we enabled our server end by assigning node1 with this protocol respectively.

4.6.1 Node2 Data Received by Gateway over WebSocket Protocol

For node2, we configured WebSocket as communication protocol as per the framework since our requirement is to establish bidirectional communication between the node2 Arduino Uno and the gateway. When the gateway is configured to receive data from node2 over WebSocket, it creates a server socket which uses a particular port 80 for regular WebSocket connections. Here, WebSocket server in the gateway, becomes ready for listening nodes configured with WebSocket protocol. In the experiment, node2 created client socket and tries to establish a communication link to the gateway server using its IP address and port number 80. When the communication established between node 2 and gateway server, server received pressure and altitude from node2 Arduino Uno Wi-Fi and showed the data to the interface. Figure 4.23 demonstrated node2 data received by the gateway over WebSocket protocol.

Device Name	Data	Select Protocol
node1		Select <input type="button" value="v"/>
node2	Pressure: 1012.73hpa , Approx. Altitude: 4.35meter	Websocket <input type="button" value="v"/>
node3		Select <input type="button" value="v"/>

Figure 4.23 Node2 data received by gateway over WebSocket protocol

4.6.2 Node3 Data Transfer to Gateway over MQTT protocol

Node 3 is configured to send data over MQTT protocol according to protocol selection framework designed in Chapter 3. Node3 device is esp32 which uses low bandwidth, the sensor connected to node 3 communicates with high latency and due to unreliable network characteristics, node 3 is configured to communicate over MQTT protocol. To establish connection from node 3 to gateway, it was selected from the interface so that gateway enables mosquitto subscribing to receive data from the sensor. Here, the gateway is devising as MQTT broker which facilitates the communication from node3 transferring messages from publisher to subscriber and subscriber to publisher. Node3 ESP32 with environmental sensor provides air quality data in ppm unit. In regards, reading data from sensor and sending data to the gateway broker, PubSubClient MQTT library is used on the node3 end. The PubSubClient library enables publish/subscribe messaging with a server that supports MQTT. In order to distinguish data sent by the node3, a topic string “/client/node2/mqtt” is used. Here, client node3 is publishing Air Quality values using different string for each value. On the other side, the broker gateway is configured to subscribe to the topic which node3 is using to publish the values. After node3 connects to the gateway broker, it starts a loop reading from the environmental sensor every 10 seconds and publishing the value to the topic “/client/node3/mqtt”. After receiving data on the gateway, data is sent to the web interface for demonstrating the real-time values and transforming data for storing into databases. In addition, for encrypted communication between the gateway broker and the client node3, TLS (Transport Layer Security) and SSL (Secure Sockets Layer) encryption is used. Figure 4.24 illustrated node3 sensor MQ-135 air quality data received by the gateway over MQTT protocol.

Device Name	Data	Select Protocol
node1		Select ▾
node2		Select ▾
node3	"AirQuality": 23 PPM	MQTT ▾

Figure 4.24 Node3 air quality data received by gateway over MQTT protocol

4.6.3 Node1 Data Transfer to Gateway over CoAP Protocol

Node1 Raspberry Pi 3B will use the AdaFruit DHT library methods to retrieve the DHT22 sensor's current temperature and humidity readings. Also, data will be formatted into JSON in order to be transmitted to the gateway. After the gateway is configured to receive data from node1 over CoAP protocol, CoAPthon Python library the script is activated to create a CoAP endpoint on the gateway. The CoAPthon library is a Python implementation of the CoAP protocol. This package contains a helper client class that uses the CoAP path and port during initialization. The object is created on the node with the path to the gateway endpoint and the default CoAP port 5683. This CoAP client object is used to send a POST message containing data to the gateway. To retrieve the readings from the DHT22 sensor, the Adafruit DHT22 library was imported to the python script of node1. This library contains a `read_retry ()` method that will attempt to read the temperature and humidity data from the DHT22 sensor and return values as floating-point decimals; if no reading is available, it will try again up to a specified number of retries, defaulting to fifteen. This default limit is specified by the CoAPthon package. Once the sensor data is returned, the payload is constructed. The payload consists of the temperature and humidity data formatted into a JSON object. This data is converted using the built-in PythonJSON library. A CoAP message is sent to the gateway using the formatted payload. The response message from node1 is output to the console. As soon as the gateway server receives the data, the data is processed, saved, and the interface dashboard is updated. The device is identified by a device access token sent as part of the request. The dashboard in Figure 4.25 displays values of temperature(T), relative humidity (RH) and timestamp.

Device Name	Data	Select Protocol
node1	2022-05-11 03:54:19 T:25.30,RH:49.10	Coap <input type="text"/>
node2		Select <input type="text"/>
node3		Select <input type="text"/>

Figure 4.25 Node1 data received by gateway over CoAP protocol

4.6.4 Node1 Data Transfer to Gateway over Modbus TCP Protocol

To demonstrate Modbus TCP communication between the client and gateway, node1 was configured as Modbus client. The communication model between node1 and gateway is client-server and the physical medium for exchanging data is the wireless network. In the experimental setup, node1 Raspberry Pi 3 works as a Modbus TCP client that transfers temperature and humidity data to gateway which works as Modbus TCP server. The core of the configuration node1 is capable of initiating a TCP connection with the gateway server, forming, and sending Modbus request, receiving, and interpreting the response, and maintaining or closing the TCP connection. For full read/write protocol on discrete and register and payload builder/decoder utilities PyModbusTCP library has been used. Server was set up to hold all the discrete inputs, coils, holding registers and input registers. Node1 was defined as ModbusTcpClient with the local IP address 192.168.0.106/24 and port 502 of the server gateway. As the gateway had register map for all data types with desired size, these registers received read/write requests from the node client. The precision of the data was adjusted based on data size limitation set by the Modbus register. By selecting Modbus TCP protocol for node1 from the dashboard, Figure 4.26 showed datetime, temperature and humidity value that is received from the node1 client.

Device Name	Data	Select Protocol
node1	2022-5-11 3:56:12 P:25.3,RH:49.2	Modbus_TCP ▾
node2		Select ▾
node3		Select ▾

Figure 4.26 Node1 data received by gateway over Modbus TCP/IP protocol

4.6.5 Node1 Data Transfer to Gateway over HTTP Protocol

In the experiment, for the demonstration of node sending data over HTTP protocol to the gateway, node1 Raspberry Pi 3B was also configured as client. Node1 has on-board Wi-Fi connectivity that helps in making wireless communication a successful attempt. The proposed design uses an Apache web server to create a local HTTP server. When a website is created, a server is required to host the website. PHP helps interpret the page and makes the page available to the client when the

request is generated. In this scenario, a Python script is written for client / server communication within the localhost. The node client sent a request message to the gateway to send temperature and humidity data, and the gateway processed it and responded to the request message to the node. A POST request is sent, so the body content type is specified. The content type is sent in the request as the header specified by calling the `addHeader` method of the HTTP client object. This method received the name of the header as the first input and its value as the second input. Node1 then used the POST method to send the request to the HTTP client object, passing the body of the request as a string as input. When data was successfully sent from node1 to the gateway, the gateway returned an HTTP response code to node1. The web interface in Figure 4.27 shows how the gateway received temperature and humidity data from the node1 endpoint over the HTTP protocol.

Device Name	Data	Select Protocol
node1	2022-05-11 03:52:43 T : 25.30, RH :48.60	http ▾
node2		Select ▾
node3		Select ▾

Figure 4.27 Node1 data received by gateway over HTTP protocol

4.7 Data Collection and Storage

In this research, the gateway is a platform which provides various services exclusively targeted for building IoT applications. It handles real-time data collection, visualizes the collected data, has ability to store data to databases and create APIs and communication hub for other devices. Here, node1, node2 and node3 connected to the dht22 temperature-humidity sensor, bme280 pressure-altitude sensor, MQ-135 air quality sensor send data to the gateway using their communication protocols. The gateway is designed to receive the data, process data in JSON format and store incoming data to databases. Three databases are presented to visualize and process for other server and device communications. We divided the databases into (i) local and cloud database (ii) KEPServerEX data logging and communication and (iii) Azure IoT hub databases.

4.7.1 Data Store to Local and Cloud Database

After processing data received from the nodes, the gateway initially send data to MariaDB which is a Linux-based relational database management server compatible with MySQL. These data are inserted into the database using MariaDB connector whose table structure is shown in Figure 4.28. The table structure used in this study to receive data from different nodes as device_id and protocols as protocol_id to insert configuration values. The application is configured as a background process and runs 24 hours nonstop and producing 17280 rows of data each day from each node. Data reading interval can be modified from the interface.

```
MariaDB [server]> SHOW COLUMNS FROM data;
```

Field	Type	Null	Key	Default	Extra
id	bigint(20) unsigned	NO	PRI	NULL	auto_increment
protocol_id	bigint(20) unsigned	NO	MUL	NULL	
device_id	bigint(20) unsigned	NO	MUL	NULL	
data	text	YES		NULL	
time	datetime	YES		NULL	
created_at	timestamp	YES		NULL	
updated_at	timestamp	YES		NULL	

Figure 4.28 Table structure for data storage in gateway local database

In the framework, different protocols are assigned with a number. Here, MQTT, HTTP, ModbusTCP, CoAP and WebSocket are assigned with protocol_id 1, 2, 3, 4 and 5 respectively. Data received from different nodes with different protocols are stored in local databases in MariaDB server. The following result in Figure 4.29 highlights node1 temperature humidity data with CoAP protocol received in MariaDB database.

```
MariaDB [server]> SELECT * FROM data where protocol_id=4 && device_id=1;
```

id	protocol_id	device_id	date-data	time
67	4	1	T:16.90,RH:30.60	2021-12-04 02:42:11
68	4	1	T:16.90,RH:30.90	2021-12-04 02:42:28
69	4	1	T:16.90,RH:31.00	2021-12-04 02:42:46
70	4	1	T:17.00,RH:31.20	2021-12-04 02:43:04
71	4	1	T:17.00,RH:31.30	2021-12-04 02:43:21
72	4	1	T:17.00,RH:31.40	2021-12-04 02:43:41
73	4	1	T:17.00,RH:31.40	2021-12-04 02:43:59
74	4	1	T:17.00,RH:31.50	2021-12-04 02:44:17
75	4	1	T:17.00,RH:31.80	2021-12-04 02:44:34

Figure 4.29 Data storage table in gateway local database

The data comes in the gateway is sent into the database created in cloud server. This table shown can be fetched from web browser/PhpMyAdmin by using the cloud web address. In this sense, each time the data of the nodes get updated, or requests made by the client, data are received by the gateway. The gateway then establishes connection with the cloud database and attends the requests through the APIs, which can modify a record in the table, read or add a new entry. In this way, it is possible to view the data in the remote web interface or to register the users to the application. The data can be retrieved from the cloud server when needed to send any signal to nodes and other gateways. The sensor values transferred to the cloud database from the gateway are shown in Figure 4.30.

```
SELECT * FROM `data` WHERE protocol_id=4 && device_id=1;
```

Options: Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

1 > >> | Show all | Number of rows: 25 | Filter rows: Search this table | Sort by key:

Options		id	protocol_id	device_id	date-data	time	created_at	updated_at
<input type="checkbox"/>	Edit Copy Delete	67	4	1	T:16.90,RH:30.60	2021-12-04 02:42:11	NULL	NULL
<input type="checkbox"/>	Edit Copy Delete	68	4	1	T:16.90,RH:30.90	2021-12-04 02:42:28	NULL	NULL
<input type="checkbox"/>	Edit Copy Delete	69	4	1	T:16.90,RH:31.00	2021-12-04 02:42:46	NULL	NULL
<input type="checkbox"/>	Edit Copy Delete	70	4	1	T:17.00,RH:31.20	2021-12-04 02:43:04	NULL	NULL
<input type="checkbox"/>	Edit Copy Delete	71	4	1	T:17.00,RH:31.30	2021-12-04 02:43:21	NULL	NULL
<input type="checkbox"/>	Edit Copy Delete	72	4	1	T:17.00,RH:31.40	2021-12-04 02:43:41	NULL	NULL
<input type="checkbox"/>	Edit Copy Delete	73	4	1	T:17.00,RH:31.40	2021-12-04 02:43:59	NULL	NULL
<input type="checkbox"/>	Edit Copy Delete	74	4	1	T:17.00,RH:31.50	2021-12-04 02:44:17	NULL	NULL

Figure 4.30 Data sent from gateway to cloud database

4.7.2 KEPServerEX Data Logging and Communication

KEPServerEX v6 was used as OPC Server to connect the gateway with HMI and Data Logger. To provide customers with a single source for industrial data, KEPServerEX uses OPC (Open Platform Communications) and many connection protocols. HMI serves as an OPC Client, requesting data from or sending orders to the hardware via the OPC server. The data acquisition middleware employed KEPServerEX with a MQTT client that could subscribe to data from various nodes on the gateway. To monitor, manage, and connect with PLCs and other devices, the data acquisition

middleware is linked to other Modbus, OPC driver platforms. In this experiment, the gateway collects data from various nodes and edge devices and sends it to Kepware's KEPServerEX OPC server MQTT client. Figure 4.31 demonstrates that data from the node3 ESP32 Wi-Fi microcontroller was being able to be accumulated in KEPServerEX OPC Server using MQTT protocol and subsequently transferred Air Quality data into Microsoft Access Data Logger. Then, node3 data from the KEPServerEX OPC Server were logged and recorded in the Microsoft Access database. The configured tag names were displayed on the OPC quick client.

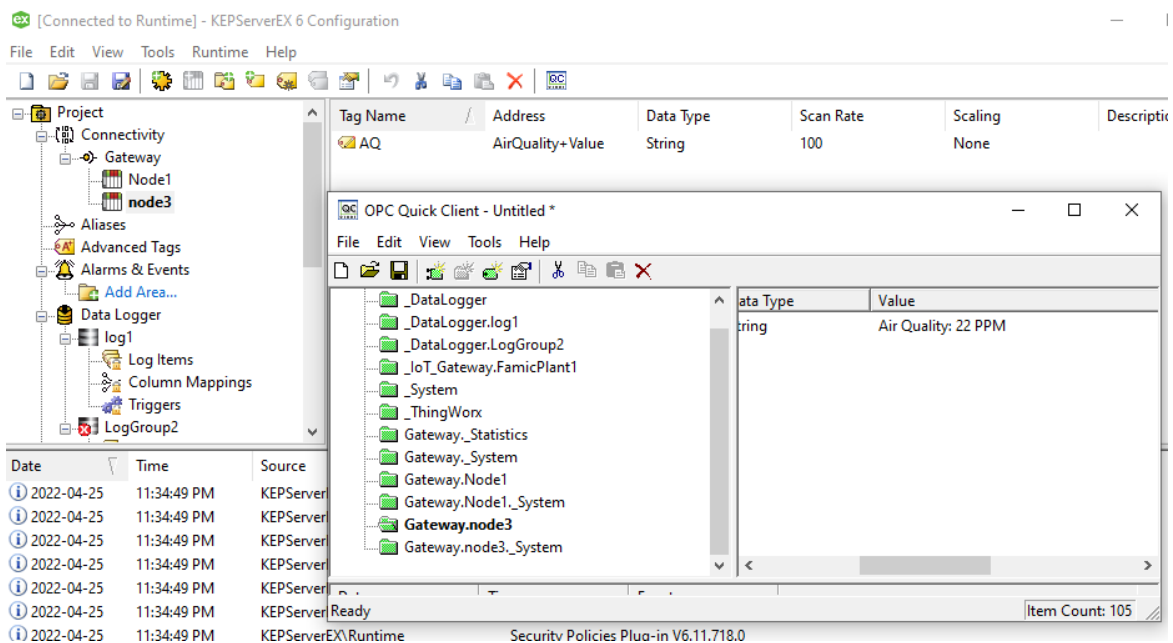


Figure 4.31 Gateway sending node3 air quality data to KEPServerEX

The collected data is sent to the database processing layer via the OPC protocol and to the display layer via the MQTT communication protocol. The presentation layer uses the client to complete the display analysis of the data stored on the server according to the appropriate logic code. This task is configured to access the Microsoft Access database and store the data in Microsoft Excel. Changes in this air quality index are recorded in a Microsoft Excel file. The designed gateway data logging system can be stored in local and cloud databases and used to communicate with other KEPServerEX drivers and industrial processes. From the Figure 4.32, we demonstrated Air Quality data with the corresponding timestamp that received from the MQTT client through the KEPServerEX.

A	B	C	D	E
id	Gateway_node3_AQ_NUMERICID	Gateway_node3_AQ_VALUE	Gateway_node3_AQ_TIMESTAMP	Gateway_node3_AQ_QUALITY
1	0	Air Quality: 23 PPM	2022-04-25 23:52	64
2	0	Air Quality: 23 PPM	2022-04-25 23:52	64
3	0	Air Quality: 23 PPM	2022-04-25 23:52	64
4	0	Air Quality: 23 PPM	2022-04-25 23:52	64
5	0	Air Quality: 23 PPM	2022-04-25 23:52	64
6	0	Air Quality: 23 PPM	2022-04-25 23:52	64
7	0	Air Quality: 23 PPM	2022-04-25 23:52	64
8	0	Air Quality: 23 PPM	2022-04-25 23:52	64
9	0	Air Quality: 23 PPM	2022-04-25 23:52	64
10	0	Air Quality: 23 PPM	2022-04-25 23:52	64
11	0	Air Quality: 23 PPM	2022-04-25 23:52	64
12	0	Air Quality: 23 PPM	2022-04-25 23:52	192
13	0	Air Quality: 23 PPM	2022-04-25 23:52	192
14	0	Air Quality: 23 PPM	2022-04-25 23:52	192
15	0	Air Quality: 23 PPM	2022-04-25 23:52	192
16	0	Air Quality: 23 PPM	2022-04-25 23:53	192
17	0	Air Quality: 23 PPM	2022-04-25 23:53	192
18	0	Air Quality: 23 PPM	2022-04-25 23:53	192
19	0	Air Quality: 23 PPM	2022-04-25 23:53	192
20	0	Air Quality: 23 PPM	2022-04-25 23:53	192
21	0	Air Quality: 23 PPM	2022-04-25 23:53	192
22	0	Air Quality: 23 PPM	2022-04-25 23:53	192

Figure 4.32 KEPServerEX data logging in Excel

4.7.3 Data Store to Azure IoT Hub and Data Explorer Databases

Microsoft's Azure IoT Hub service enables two-way communication between IoT devices and Azure. In this experiment, Azure IoT Hub environment will be deployed and connected through a gateway with nodes to the cloud platform. This environment uses Azure Data Explorer and other Azure IoT suites to provide messaging capabilities between devices and cloud services. Azure Data Explorer is a fast, scalable data exploration service for log and telemetry data that allows to merge, store, and analyze heterogeneous data. Azure IoT Hub was created to get the connection string for the gateway. A free tier was selected for testing and evaluation purposes in order to use the free subscription. With free subscription, 500 devices can be connected to the hub and transferred up to 8,000 messages per day. After retrieving the primary connection string, the gateway establishes communication with the IoT hub and sends data to the hub using the MQTT protocol. The gateway relayed temperature and humidity data to the Azure IoT hub, as illustrated in Figure 4.33, for demonstration purposes.

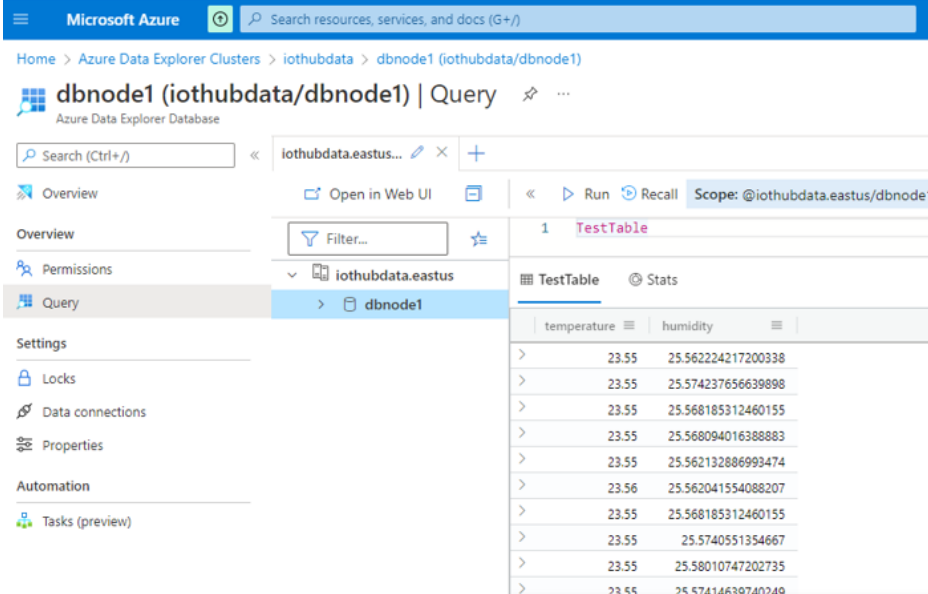
```

pi@raspberrypi:~/azure-iot-samples-node/iot-hub/Tutorials/RaspberryPiApp $ sudo node index.js 'HostN
redAccessKey=7dAvG1YV47KKJpL9YtyhV+WKCVR6gO+keeBiW+s6rnc='
[Device] Using MQTT transport protocol
Found BMx280 chip ID 0x60 on bus i2c-1, address 0x77
[Device] Sending message: {"messageId":1,"deviceId":"Raspberry Pi Node","temperature":21.39,"humidit
[Device] Message sent to Azure IoT Hub
[Device] Sending message: {"messageId":2,"deviceId":"Raspberry Pi Node","temperature":21.41,"humidit
y":25.216436210407775}
[Device] Message sent to Azure IoT Hub
[Device] Sending message: {"messageId":3,"deviceId":"Raspberry Pi Node","temperature":21.42,"humidity
:25.216096730871815}
[Device] Message sent to Azure IoT Hub
[Device] Sending message: {"messageId":4,"deviceId":"Raspberry Pi Node","temperature":21.42,"humidity
:25.164999804138866}

```

Figure 4.33 Gateway sending temperature humidity data to Azure IoT Hub

Connecting Azure Data Explorer table to IoT hub, TestTable table has been mapped for the node1 incoming data. Azure Data Explorer includes database functionality for receiving and storing data. By grouping and aggregating the data, the data is converted to JSON format and stored according to different categories via a simple JDBC storage method. TestTable was created to map incoming data to the temperature, humidity, and JSON data types of the table columns. The supported data formats for transferring data to Azure Data Explorer tables are Avro, CSV, JSON, MULTILINE JSON, ORC, TSV, TXT etc. Figure 4.34 demonstrates temperature humidity data receiving on the Azure Data Explorer database.



The screenshot shows the Azure Data Explorer interface for a database named 'dbnode1 (iothubdata/dbnode1)'. The 'Query' tab is active, showing a table named 'TestTable' with two columns: 'temperature' and 'humidity'. The table contains 10 rows of data, each with a temperature value and a corresponding humidity value.

temperature	humidity
23.55	25.562224217200338
23.55	25.574237656639898
23.55	25.568185312460155
23.55	25.568094016388883
23.55	25.562132886993474
23.56	25.562041554088207
23.55	25.568185312460155
23.55	25.5740551354667
23.55	25.58010747202735
23.55	25.57414639740249

Figure 4.34 Data ingestion to Azure Data Explorer database

4.8 Visualization of Real-Time Node Data in Web Application

The Azure App Service Web Apps feature provides a Platform as a Service (PAAS) for hosting web applications. Azure App Service supports web applications developed in many popular languages. In this case, it will be deployed to a Linux infrastructure-based gateway. To visualize node data in a web application, environment variables require to be configured. In order to read the data from the Azure IoT hub, the web app had to create a connection string and a consumer group name. Deploying code to App Service, Git, and FTP uses user-level deployment credentials. Figure 4.35 shows deployment to show temperature & humidity real time data on web page <https://temhum.azurewebsites.net>. When the node sends data to the gateway, the gateway processes it and sends it to Azure IoT hub for a plot of the gateway's 50 most recent readings.

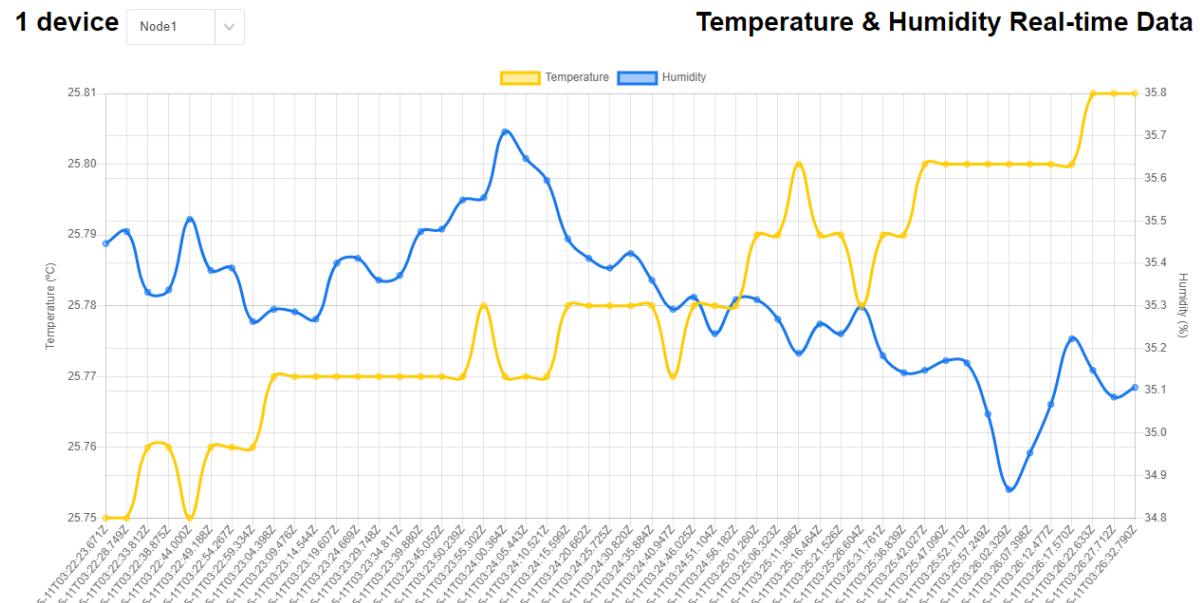


Figure 4.35 Temperature humidity real-time data visualization in Azure IoT web application

4.9 Development Conclusion

In this research development chapter, we implemented IoT multi-protocol gateway which can accept client-originated messages in various protocols on a low-cost microprocessor and demonstrated protocol interoperability among three different nodes and three different sensors. Temperature, humidity, pressure, altitude, and air quality sensors are used to illustrate wide range

of sensor data transmission in the proposed gateway. Other industrial and low powered sensors can be used as well. We also presented multiple server integration and communication for effective protocol conversion and bridging in the gateway. The last part of the development showed integration of different cloud and industrial access databases such as Azure and KEPServerEX with different data formats.

CHAPTER 5 RESULTS AND DISCUSSION

Interoperable IoT multiprotocol conversion system has been implemented in a low cost IoT gateway with multiple nodes via 5 industry standard access protocols in Chapter 4. In this section, a case study will be demonstrated to analyze a workflow involving data collection via nodes with sensors, gateway integration via a parametric control mechanism, and visualization modules to facilitate management and monitoring, with the goal of testing feasibility of the interoperability concept of integrating any platforms with the developed multi-protocol gateway. The objective is to test feasibility of the developed multi-protocol gateway by a case study to demonstrate interoperable access to any IoT open-source platform that connects via any of the 5 industry standard protocols (MQTT, CoAP, HTTP, Modbus TCP and WebSocket) and supports cloud implementations.

5.1 Case Study: Implementation on ThingsBoard Platform

The case study refers to implementation communication between gateway and ThingsBoard IoT platform. ThingsBoard is an open-source IoT platform built on the Java 8 platform that functions as an IoT gateway between registered devices communicating via HTTP, CoAP, and MQTT protocols to collect, analyse, visualise, and manage data (Paolis, Luca, & Paiano, 2018). ThingsBoard uses a powerful server-side API to securely provision, monitor, and control Internet of Things entities. Establish connections between devices, resources, customers, and other entities. The platform is designed to collect and store telemetry data in a fault-tolerant and scalable manner. There are built-in or custom widgets and customizable dashboards for visualizing data. Customers can view and use these dashboards. Specifies the data processing rule chain. Not only does this allow to transform and normalize device data, but it also triggers alerts based on incoming telemetry events, attribute updates, device inactivity, and user activity (Casillo et al., 2021). It provides a ready-to-use IoT solution for server-side infrastructure for a variety of IoT applications in the cloud or on-premises and presently supports three database options: SQL, NoSQL, and hybrid databases. These databases are used by the ThingsBoard platform to store entities such as devices, assets, dashboards, users, alerts, and telemetry data such as attributes, time series sensor readings, statistics, and events. The security features of ThingsBoard consist of company-recommended

encryption algorithms, including SSL, and a sort of tool registration credentials, including the acquisition of X.509 certificates and access tokens. (Henschke, Wei, & Zhang, 2020).

5.2 Configuration with ThingsBoard Platform

ThingsBoard cloud infrastructure is proposed to demonstrate smooth protocol integration with multi-protocol enabled gateway. The gateway using Raspberry Pi 4 acquire data from different sensor and IoT devices, transform, convert, and transfer the data to ThingsBoard IoT platform to monitor and visualise sensors data. In this experiment, the gateway used MQTT protocol to send the node and sensor data formatted as JavaScript Object Notation (JSON) to the ThingsBoard cloud endpoint at regular intervals. MQTT is a lightweight protocol and has a smaller header size per message than HTTP, so it was prioritized over HTTP in the project. HTTP, a heavy protocol, requires more overhead and message size than MQTT. ThingsBoard is configured to monitor and visualise data by creating IoT Dashboards and updating in real-time. Figure 5.1 illustrates the integration details between the gateway and the ThingsBoard platform.

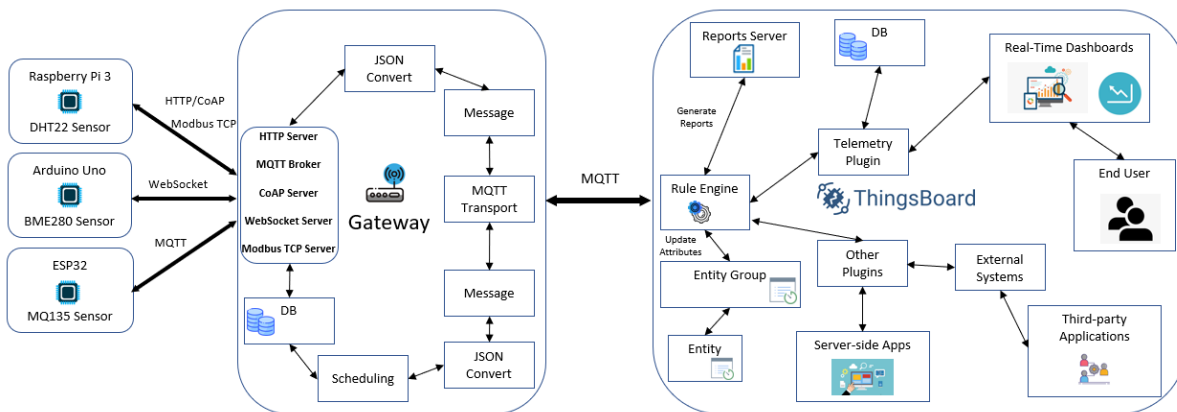


Figure 5.1 Cloud based ThingsBoard platform integration with multi-protocol gateway

5.2.1 Gateway Configuration

The multi-protocol gateway receives sensor data from three different nodes with different protocols, process the data before sending to ThingsBoard Platform. Three different nodes Raspberry Pi 3, Arduino Uno and ESP32 embedded with DHT22 temperature-humidity sensor, BME280 atmospheric pressure, altitude sensor, and MQ135 air quality sensor send sensor values to the proposed multi-protocol gateway with Modbus TCP, WebSocket and MQTT protocol respectively. The gateway is configured to receive and store the data to the local database and cloud

database. Gateway ThingsBoard packages has been installed in the gateway. mqtt.json connector file has been created to map the sensor data and values in the JSON format. By commenting out hashes in the configuration, a connector is activated. A name, type, and configuration file parameters are required for each connector. It is feasible to obtain multiple connectors can be active at the same time if the files are specified with different name and configurations. Figure 5.2 shows the mapping strategies for the mqtt.json connector.

```

"mapping": [
  {
    "topicFilter": "/sensor/data",
    "converter": {
      "type": "json",
      "deviceNameJsonExpression": "${serialNumber}",
      "deviceTypeJsonExpression": "${sensorType}",
      "timeout": 60000,
      "attributes": [
        {
          "type": "string",
          "key": "model",
          "value": "${sensorModel}"
        },
        {
          "type": "string",
          "key": "${sensorModel}",
          "value": "on"
        }
      ]
    }
  }
],

```

Figure 5.2 Configuration for MQTT mapping in JSON format

The gateway uses an access token to access the web interface of the ThingsBoard cloud server. The Device information tab contains the access token that is used to authenticate the gateway. The file 'tb_gateway.yaml' in the configuration folder for the ThingsBoard platform on the gateway is used to configure the connection to the ThingsBoard server. The hostname or IP address of the ThingsBoard server, as well as the port of the MQTT service on the server, are defined in this main configuration file. The access token is pasted underneath the security label as seen in Figure 5.3. Memory storage is used for storing incoming data before being sent to the server. Telemetry sent by the gateway for logging in the platform is inserted into the SQLite database table before being transferred to the ThingsBoard. No other relational database, such as PostgreSQL or MySQL, was used for this task because only one table was needed to store the data. SQLite is easy to set up and

manage. SQLite only needs database files, so it doesn't need a server. SQLite and data file path `./data/data.db` file have been initiated to store the data before sending to the platform.

```
GNU nano 3.2 /etc/thingsboard-gateway/config/tb_gateway.yaml
thingsboard:
  host: thingsboard.cloud
  port: 1883
  remoteShell: false
  remoteConfiguration: false
  statsSendPeriodInSeconds: 3600
  minPackSendDelayMS: 0
  checkConnectorsConfigurationInSeconds: 60
  security:
    accessToken: UEUAObQdVgtGjbrDXj80
  qos: 1
storage:
  type: memory
  read_records_count: 100
  max_records_count: 100000
```

Figure 5.3 Gateway-ThingsBoard configuration parameters

To deliver data to the ThingsBoard endpoint, a Python script using `mqtt paho` and `JSON` libraries is loaded. Figure 6.1 shows how the proposed gateway communicates node data to ThingsBoard over MQTT as JSON strings, where measured parameter values are represented by key-value pairs. Once the gateway receives sensor data, the payload is constructed. The payload consists of the temperature, humidity, pressure, altitude, and air quality data formatted into a JSON object, shown in Figure 5.4. This data is converted using the built-in Python `JSON` library. The JSON string of a typical client message could be:

<pre>{ "Temperature": 26.70, "Humidity": 79.20 }</pre>	<pre>{ "Pressure": 1002.82hpa, "Altitude": 87.17meter }</pre>	<pre>{ "AirQuality": 22ppm }</pre>
<p>Node1 RPI 3 DHT22 Temperature Humidity Sensor Values in JSON format</p>	<p>Node2 Arduino Uno BME280 Pressure Altitude Sensor Values in JSON format</p>	<p>Node1 ESP32 MQ135 Air Quality Index Sensor Values in JSON format</p>

Figure 5.4 Sensor data formatted to JSON data format for message payload

5.2.2 ThingsBoard Configuration

ThingsBoard support device management features using Web UI and REST API. With administrator access gateway_device01 is created which bridges communication through the proposed gateway. Device ID 11e62fa0-e690-11ec-a502-79978f9d7342 is generated for the gateway_device01. It is also configured as gateway in the ThingsBoard platform which is depicted in Figure 5.5. The device is also assigned to certain customer which allows Customer users to fetch device data using REST APIs or Web UI.

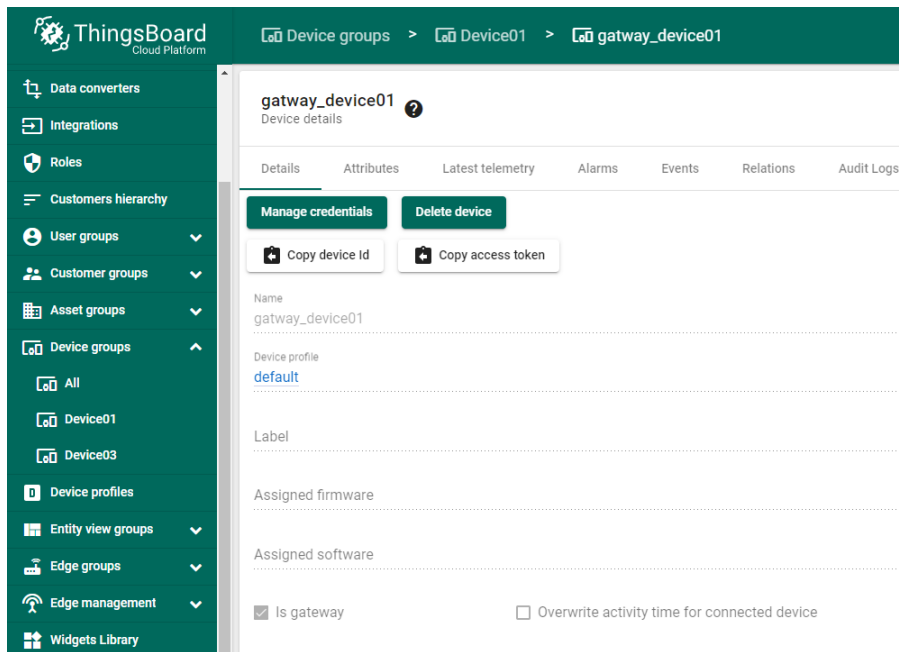
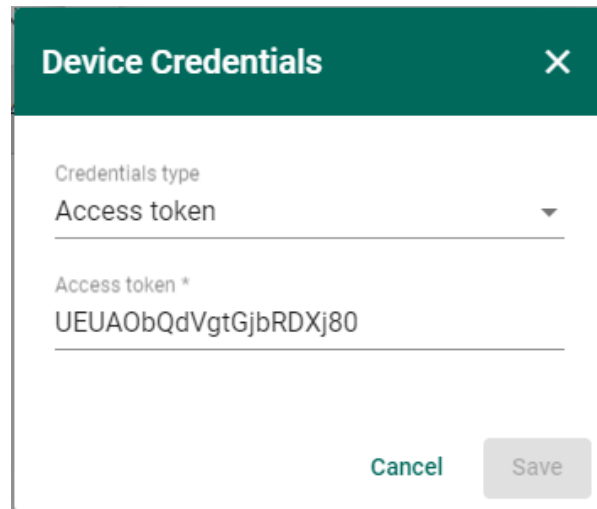


Figure 5.5 Gateway configuration as device on ThingsBoard cloud platform

The standard treatment of data sent to the telemetry endpoint include identifying the device delivering the data, storing the values, and updating any dashboards associated to that device. The device is recognised by the device access token provided in the request. The ThingsBoard access token utilised for the gateway in this study is shown in Figure 5.6. In this work, we'll use \$ACCESS TOKEN, which stands for access token device credentials. In the username field, the application must send a MQTT CONNECT message with the username \$ACCESS TOKEN. The following are possible return codes and their causes during the connect sequence: (i) 0x00 Connected - Successfully connected to ThingsBoard MQTT server. (ii) 0x04 Connection Refused, bad username or password - Username is empty. (iii) 0x05 Connection Refused, not authorized - Username contains invalid \$ACCESS_TOKEN.



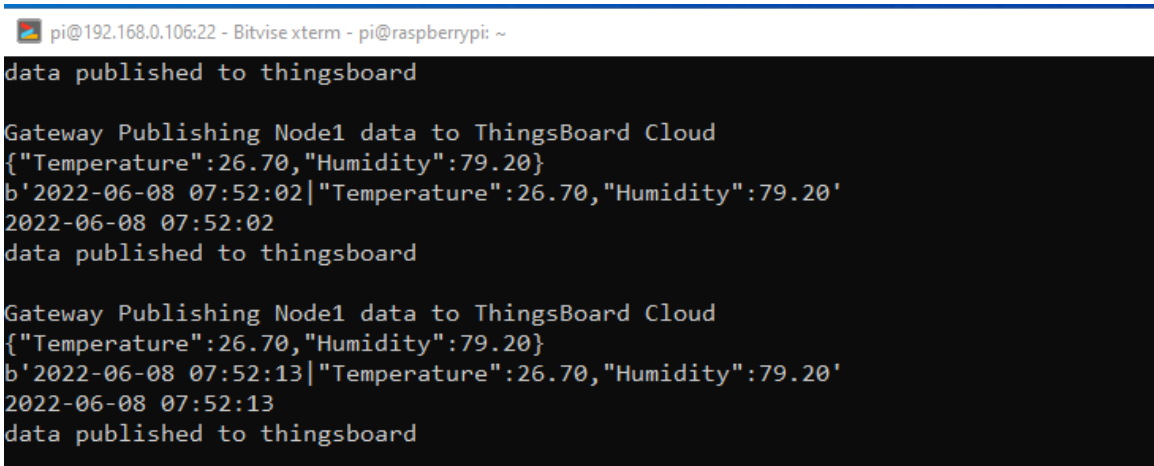
The image shows a 'Device Credentials' dialog box. The title bar is dark green with the text 'Device Credentials' and a close button (X). The main content area is white and contains a dropdown menu for 'Credentials type' with 'Access token' selected. Below this is a text input field for 'Access token *' containing the value 'UEUAObQdVgtGjbRDXj80'. At the bottom right, there are two buttons: 'Cancel' and 'Save'.

Figure 5.6 Gateway access token credentials generated on ThingsBoard

When the data is received by the ThingsBoard cloud, it is processed by the ThingsBoard cloud's rules engine. Data handling has not been subjected to any additional rules for this investigation. ThingsBoard allows defining rules to apply to incoming messages and message processing plugins. Filters for receiving messages, processors for adding metadata to messages, and actions for converting messages to new custom messages passed to plugins are all part of the rule engine. These functions can be used to perform some basic data processing activities, but not more complex steps. Aggregating data over time is not easy because it cannot track previously received values. Also, each rule script only allows access to values received from a single device, so it does not allow aggregation of values received from multiple devices at the same time. ThingsBoard also offers the possibility to create several containers called assets to reconstruct the data to upload the results of the data processing. ThingsBoard provides a REST API for managing entities such as devices and assets, retrieving data from telemetry, and a REST plugin for sending HTTP requests to external endpoints. Each access token is used to provide bridging between the gateway device and the ThingsBoard. Next, a connection to the MQTT broker is established through the Connection feature of ThingsBoard. The MQTT system is an API endpoint that allows telemetry to be uploaded to ThingsBoard. Finally, the telemetry is uploaded to the ThingsBoard using the MQTT publishing feature.

5.3 Gateway Transferring Node1 data to ThingsBoard

The proposed multiprotocol gateway receives DHT22 sensor temperature-humidity data transferred by Node1 Raspberry Pi 3 through Modbus TCP protocol and publish the data to ThingsBoard cloud over MQTT protocol. Since the gateway cannot perform read or write operations for Modbus TCP/IP protocol, it accepts read/write requests and replies to a message to node1 client. As the gateway has register map for all data types with desired size, these registers receive read/write requests from the node1 Raspberry pi 3. For full read/write protocol on discrete and register and payload builder/decoder utilities PyModbusTCP library has been used. The precision of the data is changed based on the Modbus register on the gateway's data size constraint. “`json.dumps(data)`” is used to convert the sensor data to JSON format. To publish telemetry data to ThingsBoard server node, gateway publish message to the following topic: “`v1/devices/me/telemetry`”. Access token 'UEUAObQdVgtGjbRDXj80' and publishing broker “`thingsboard.cloud`” and port 1883 are also initialized to publish data into the platform. Figure 5.7 demonstrates gateway publishing sensor data receiving from node1 to the ThingsBoard platform.



```

pi@192.168.0.106:22 - Bitvise xterm - pi@raspberrypi: ~
data published to thingsboard

Gateway Publishing Node1 data to ThingsBoard Cloud
{"Temperature":26.70,"Humidity":79.20}
b'2022-06-08 07:52:02|"Temperature":26.70,"Humidity":79.20'
2022-06-08 07:52:02
data published to thingsboard

Gateway Publishing Node1 data to ThingsBoard Cloud
{"Temperature":26.70,"Humidity":79.20}
b'2022-06-08 07:52:13|"Temperature":26.70,"Humidity":79.20'
2022-06-08 07:52:13
data published to thingsboard

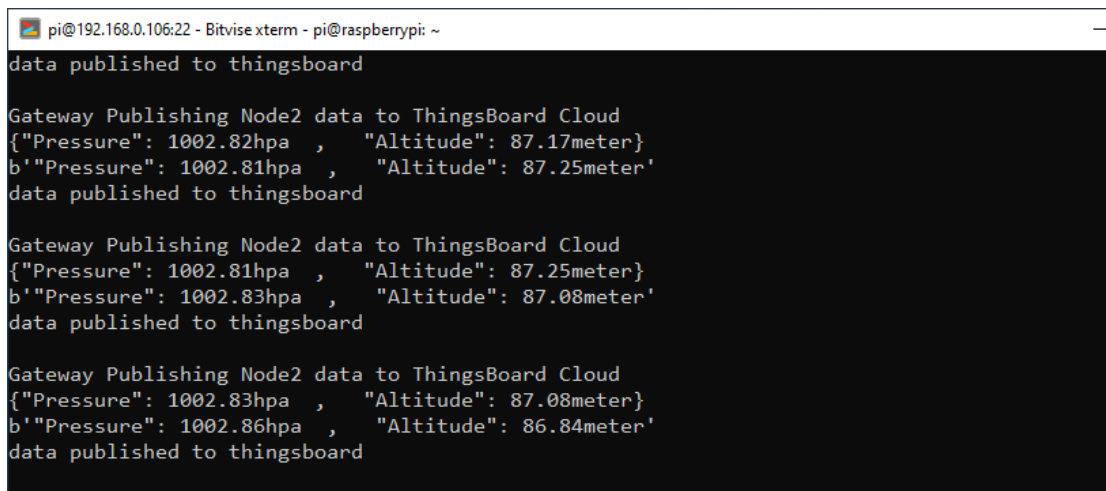
```

Figure 5.7 Gateway publishing Node1 data to ThingsBoard Cloud

5.4 Gateway Sending Node2 data to ThingsBoard

The proposed multiprotocol gateway receives BME280 sensor pressure-altitude data transferred by Node2 Arduino UNO Wi-Fi through WebSocket protocol and publish the data to ThingsBoard cloud over MQTT protocol. WebSocket API is used to receive sensor data from the nodes to the gateway. With the help of `requests.post ()` method the formatted payload is received and stored to

the local database. The data is converted using the built-in Python JSON library. In this experiment, WebSocket server in the gateway, becomes ready for listening data from node3 which communicates over WebSocket protocol. Node2 Arduino Uno creates client socket and tries to establish a communication link to the gateway server using its IP address and port number 80. When the communication established between node 2 and gateway server, gateway receives pressure and altitude values from node2 Arduino Uno Wi-Fi. Gateway enables publishing pressure and altitude telemetry data real-time to publish message to the following topic: “v1/devices/me/telemetry” on ThingsBoard platform. The gateway also transfers data stored in the database based on the scheduling policy set by the administrator. To publish data into the platform, the access token and publishing broker "thingsboard.cloud" are also set up, as well as port 1883. Figure 5.8 shows the gateway sending sensor data from Node1 to the ThingsBoard platform.



```

pi@192.168.0.106:22 - Bitvise xterm - pi@raspberrypi: ~
data published to thingsboard

Gateway Publishing Node2 data to ThingsBoard Cloud
{"Pressure": 1002.82hpa , "Altitude": 87.17meter}
b'"Pressure": 1002.81hpa , "Altitude": 87.25meter'
data published to thingsboard

Gateway Publishing Node2 data to ThingsBoard Cloud
{"Pressure": 1002.81hpa , "Altitude": 87.25meter}
b'"Pressure": 1002.83hpa , "Altitude": 87.08meter'
data published to thingsboard

Gateway Publishing Node2 data to ThingsBoard Cloud
{"Pressure": 1002.83hpa , "Altitude": 87.08meter}
b'"Pressure": 1002.86hpa , "Altitude": 86.84meter'
data published to thingsboard

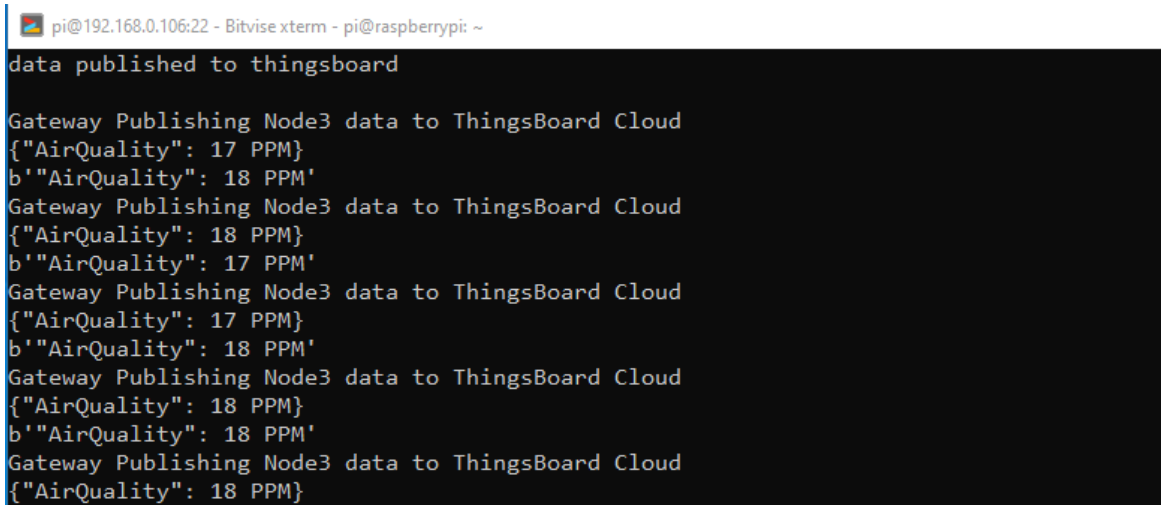
```

Figure 5.8 Gateway publishing node1 data to ThingsBoard cloud

5.5 Gateway Publishing Node3 data to ThingsBoard

The proposed multiprotocol gateway receives MQ135 environmental sensor air quality index data transferred by Node3 ESP32 through MQTT protocol and publish the data to ThingsBoard cloud over MQTT protocol. The PubSubClient library enables publish/subscribe messaging on the gateway that supports MQTT. The gateway acting as MQTT broker is configured to subscribe to the topic which node3 is using to publish the values. After node3 connects to the gateway, it starts a loop reading from the environmental sensor every 10 seconds by subscribing the value to the topic “/client/node3/mqtt?”. After receiving data on the gateway, data is sent to the to publish

telemetry data to ThingsBoard cloud to the topic “v1/devices/me/telemetry”. Before publishing sensor values to ThingsBoard platform, “json.dumps(data)” is used to process the sensor data to JSON format. paho.Client(“control1”), access token 'UEUAObQdVgtGjbRDXj80' and publishing broker “thingsboard.cloud” and port 1883 are also set to publish data into the platform. Figure 5.9 demonstrates gateway publishing air quality sensor data receiving from node3 to the ThingsBoard platform.



```

pi@192.168.0.106:22 - Bitvise xterm - pi@raspberrypi: ~
data published to thingsboard

Gateway Publishing Node3 data to ThingsBoard Cloud
{"AirQuality": 17 PPM}
b'"AirQuality": 18 PPM'
Gateway Publishing Node3 data to ThingsBoard Cloud
{"AirQuality": 18 PPM}
b'"AirQuality": 17 PPM'
Gateway Publishing Node3 data to ThingsBoard Cloud
{"AirQuality": 17 PPM}
b'"AirQuality": 18 PPM'
Gateway Publishing Node3 data to ThingsBoard Cloud
{"AirQuality": 18 PPM}
b'"AirQuality": 18 PPM'
Gateway Publishing Node3 data to ThingsBoard Cloud
{"AirQuality": 18 PPM}

```

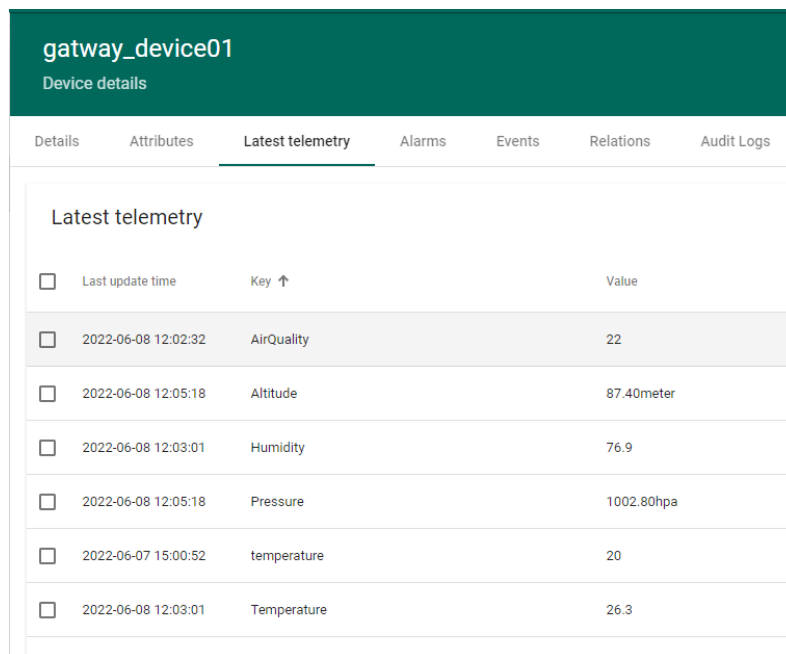
Figure 5.9 Gateway publishing node3 data to ThingsBoard cloud

5.6 Real-time Visualization on ThingsBoard

ThingsBoard's integration APIs allow custom applications to be built, and they use their own data visualisation tools. The software's complex stack technology ensures seamless performance, while its error-free data analytics provide real-time insights into the device usage patterns. Using the MQTT transport protocol, the multiprotocol gateway sends all sensor information to the ThingsBoard cloud platform. The platform receives data under topic “v1/devices/me/telemetry” and stores in SQL(PostgreSQL) databases which stores entities (devices, assets, customers, dashboards, etc.) and telemetry data (attributes, time series sensor readings, statistics, events). SQL storage is used as the experiment is considered to receive less than 5000 data points per second. The latest time series data values are queried within a specified time range using flexible aggregation. When the

message is successfully saved to the rule engine queue specified in the device profile, the device that delivers the message containing the time series data to the ThingsBoard receives an

acknowledgment. ThingsBoard can receive multiple telemetry data independently from multiple devices at the same time, but here a gateway is configured to run the experiment. Devices configured as Gateway_device01 on the ThingsBoard platform receive telemetry data from the multi-protocol gateway. As seen in Figure 5.10, gateway is sending all the Temperature 26.3 degree Celsius, humidity 76.9%, atmospheric pressure 1002.82 hPa, altitude 87.40 meter, and air quality index 22 values which are received by the gateway_device01 device on the ThingsBoard platform with timestamp.



gateway_device01			
Device details			
Details	Attributes	Latest telemetry	Alarms
Latest telemetry			
<input type="checkbox"/>	Last update time	Key ↑	Value
<input type="checkbox"/>	2022-06-08 12:02:32	AirQuality	22
<input type="checkbox"/>	2022-06-08 12:05:18	Altitude	87.40meter
<input type="checkbox"/>	2022-06-08 12:03:01	Humidity	76.9
<input type="checkbox"/>	2022-06-08 12:05:18	Pressure	1002.80hpa
<input type="checkbox"/>	2022-06-07 15:00:52	temperature	20
<input type="checkbox"/>	2022-06-08 12:03:01	Temperature	26.3

Figure 5.10 Latest telemetry received by the device on ThingsBoard cloud platform

Customizable IoT dashboards can be created using the Web UI. Numerous widgets that visualise data from multiple devices may be included in an IoT dashboard. Dashboard is created from the Dashboard group. ThingsBoard gives developers access to a large library of pre-set widgets organized into macro categories such as time series, recent values, controls, alarms, and static widgets. Different types of graphical solutions are available in each category, including charts and tables, maps, simple HTML maps, GPIO (general purpose input / output) controllers, analog and digital gauges. Entity aliases determines specific devices and assets to display on the dashboard. A single entity is configured for displaying gateway_device01 data to the widget library. ThingsBoard also allows users to create comprehensive dashboards for data visualisation that are updated in real-time and can be modified with over 30 widgets. Here, analog gauges and digital

gauges are used to show graphical visualization for temperature, humidity, atmospheric pressure, altitude, and air quality index. All these widgets show float value of the sensors. The dashboard in Figure 5.11 has five display widgets: one showing the last temperature reading, one showing all the last percentage humidity readings for the previous hour, one showing the last atmospheric pressure reading, one showing last altitude and air quality index readings at Cote des Neiges area in Montreal, Canada. This type of widgets uses values of entity attributes or time series as a data source. Different widgets can be set for same entity attribute to demonstrate wide range of visualization interface.

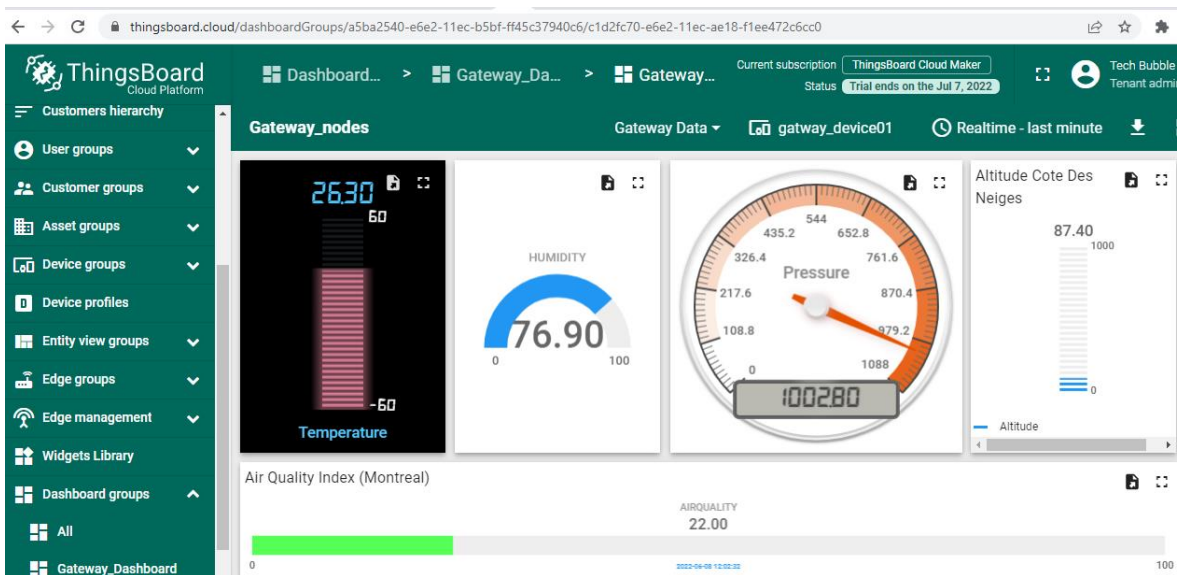


Figure 5.11 Real-time data visualization dashboard on ThingsBoard platform

5.7 Results

As discussed in the 2.5 Research Gap, different interaction paradigms are used in communication protocols for data propagation, such as request/response and publish/subscribe. The typical architecture in production environments consists of a large number of gadgets, sensors, and gateways that might converse via various protocols. The fundamental problem with current interoperable solutions is that there isn't a good way to integrate many IoT protocols into a gateway, have efficient interoperable communication to connect the sensors, IIoT devices, and machines, and integrate the cloud for platforms that are compatible. To overcome these challenges, we proposed an interoperable IoT system in a low-cost gateway that can communicate with different and heterogenous devices. To demonstrate communication between the gateway, connected

devices and platforms, we have taken three different embedded devices Raspberry Pi3, Arduino Uno Wi-Fi and ESP32. We also have used three different sensors DHT22, BME280 and MQ135 embedded with the above mentioned three devices. Our feasibility study of interoperable communication includes following approaches.

- Different devices with different protocols will transfer sensor data to gateway.
- Gateway receives data from different protocol, bridges communication and transfer to database
- Gateway transfers sensor data to third party cloud platform with platform specific protocol

The gateway receives data from three different embedded device Raspberry Pi 3, Arduino Uno Wi-Fi and ESP32 over three different protocols such as Modbus TCP, WebSocket and MQTT. The gateway receives the signal, processes, and transfers the sensors values real time into ThingsBoard IoT platform over MQTT protocol. To address the IoT Interoperability challenges, the gateway can provide the strategy of using different protocol integration, and data conversion, as well as integrating real-time data analysis with Kafka and Spark platforms on ThingsBoard platform for big data analytic applications. Integrating different protocol-enabled devices into each system enables the scalability, automation, and flexibility of Industry 4.0 manufacturing systems. Performing feasibility experiment with the proposed and developed interoperable IoT gateway and open source ThingsBoard IoT platform concludes following results.

- The IoT gateway communicates with any device with any of the 5 protocols such as MQTT, CoAP, HTTP, WebSocket and ModbusTCP
- IoT gateway works as a hub and receives data from different devices with any of the above mentioned 5 protocols
- The gateway transforms the data with JSON format, communicates with the third party ThingsBoard platform
- As ThingsBoard communicates via MQTT protocol, gateway transfers sensors data to the platform with MQTT protocol.
- Gateway also receives signal and input from the third party IoT platform. Thus, both way duplex communication is achieved. . This communication is bidirectional as it allows two-way communication with the gateway.

Successful integration of the multi-protocol gateway to the open source IoT ThingsBoard platform supports interoperable data exchanges compatibility of the proposed gateway which succeeds overcoming the challenges described in research gap. This multi-protocol gateway performs data formatting, protocol bridging per gateway architecture mentioned in the research design in chapter 3. The IoT interoperable gateway system will solve the interoperable challenges described in the literature review. Developing interoperable IoT architecture on a low-cost hardware which also satisfies RQ1 and RQ2 discussed in 1.1 Research Questions, and it could be used as interoperable middleware in small and medium enterprises. Interoperable integration and uniform access of these different IoT standards are provided to ensure seamless connectivity with different type of CPSs, devices, resources, and applications.

5.8 Limitations

Although the proposed IoT gateway can be used as a potential connectivity option for connecting heterogenous devices, machines, sensors, low powered CPSs, it has some limitations. One problem is while there are many other protocols used in the industrial communication, it can communicate through only 5 protocols MQTT, CoAP, HTTP, WebSocket and ModbusTCP. IoT systems are interconnected and communicate through networks. Therefore, despite all security measures, the system remains largely uncontrollable and can lead to various types of network attacks especially more prone to overall destruction in DoS attacks. The hardware that is used to implement the gateway is low powered and low processing power. For connecting more than 30 devices and performing more complex tasks, Raspberry Pi IoT gateway won't be useful because it doesn't have the resources and the capacity to help due to its limited capacity and capabilities. In that case, for more connected devices, powerful gateway is required to implement the proposed interoperable system.

CHAPTER 6 CONCLUSION AND RECOMMENDATIONS

This chapter provides a summary on the approach adopted by this research work, major contributions and points out some areas and scope of future work.

6.1 Conclusion

For interoperable M2M communication between heterogenous devices, we proposed and developed an interoperable middleware gateway by enabling multi-protocol integration to maintain robustness and immediate action plan towards sustainable Industry 4.0 manufacturing. In this research, we introduced an interoperable middleware gateway based IoT solution which can effectively handle IoT data from multi-protocol enabled devices, and transmit data to interconnect IoT objects, applications, heterogenous devices and machines with different access protocols. For collaborative M2M optimization with intelligent adaptation and integration at semantic level, we discussed findings of common access protocols to structure any data so that the manner of processing the information will be interpretable among the cyber-physical systems. This research presented protocol selection framework which allows users to select the suitable protocol for the application in regards of their standards and data transmission capabilities to accelerate safe and high-speed data transfer among end IoT devices. The research work implemented IoT multi-protocols such as HTTP, MQTT, CoAP, WebSocket, and Modbus TCP enabled low-cost gateways for effective full-duplex interoperable M2M communication and cloud integration among cyber-physical systems. The research also introduced data formatting method in the gateway to overcome data integration challenges from different devices and systems for a common systematization and representation in JSON format. To accommodate sheer scope of data and to address data integration and data interoperability challenges, three data storages and databases are presented for high reliability and rapid access to data. This is achieved by deploying the proposed Middleware gateway to local and cloud database, OPC Server based KEPServerEX to connect HMI and Data Logger, and Microsoft Azure IoT hub databases. In this research, we evaluated the potential gain of deploying the middleware gateway in a case study to provide a real-time cloud based integration and visualization module that facilitates sensor-based data collection between nodes, gateway integration with parametric control mechanisms and, interoperability management and to examine the potential benefits of analyzing the included workflow with the goal of validating the

interoperability concept of integrating any platforms via any of the industry standard protocols MQTT, CoAP, HTTP, Modbus TCP and WebSocket with the developed multi-protocol gateway.

6.2 Future Work

There can be some future work to extend the functionality and connectivity of the multi-protocol gateway. Future development can be expanding the supported OPC UA service and AMQP, CAN, CC-Link protocols implementation and PLC and other controllers bridging in the gateway. Future experimental tests will evaluate the performance of data analytics systems in terms of responsiveness, flexibility, and scalability in large, real-world scenarios. Implementing large number of multi-protocol gateways and devices in mesh network with scalability would be useful for long range Industrial IoT networks and large industrial manufacturing plants. Furthermore, fog and edge computing paradigm for data processing and AI techniques can be included in future work by examining the gateway module's connection, data transmission and synchronization capabilities. Future study could concentrate on delivering IoT security via the IoT Edge gateway, which is one of the primary difficulties facing IoT networks. The protocols and standards for IoT communication are continually changing. As a result, additional research into other protocols to include in the suggested middleware solution will be useful.

REFERENCES

- Adamson, G., Wang, L., Holm, M., & Moore, P. (2017). Cloud manufacturing – a critical review of recent development and future trends. *International Journal of Computer Integrated Manufacturing*, 30(4-5), 347-380. DOI: [10.1080/0951192X.2015.1031704](https://doi.org/10.1080/0951192X.2015.1031704)
- Alghamdi, T., Lasebae, A., & Aiash, M. (2013). Security Analysis of the Constrained Application Protocol in the Internet of Things. *2013 Second International Conference on Future Generation Communication Technology (FGCT)*. DOI: [10.1109/FGCT.2013.6767217](https://doi.org/10.1109/FGCT.2013.6767217)
- Bandyopadhyay, S., & Bhattacharyya, A. (2013). Lightweight Internet protocols for web enablement of sensors using constrained gateway devices. *2013 International Conference on Computing, Networking and Communications (ICNC)* 334-340. DOI: [10.1109/ICCNC.2013.6504105](https://doi.org/10.1109/ICCNC.2013.6504105)
- Barros, V., Junior, S., Bruschi, S., Monaco, F., & Estrella, J. (2019). *An IoT Multi-Protocol Strategy for the Interoperability of Distinct Communication Protocols applied to Web of Things*. DOI: [10.1145/3323503.3349546](https://doi.org/10.1145/3323503.3349546)
- Casillo, M., Colace, F., Santo, M. D., Lorusso, A., Mosca, R., & Santaniello, D. (2021). *VIOT_Lab: A Virtual Remote Laboratory for Internet of Things Based on ThingsBoard Platform*. Paper presented at the 2021 IEEE Frontiers in Education Conference (FIE). Retrieved from <https://ieeexplore.ieee.org/document/9637317/>
- Cavalieri, S. (2021). Semantic Interoperability between IEC 61850 and oneM2M for IoT-Enabled Smart Grids. *Sensors*, 21(7). DOI: [10.3390/s21072571](https://doi.org/10.3390/s21072571)
- De S., L. M. S., Spiess, P., Guinard, D., Koehler, M., Karnouskos, S., & Savio, D. (2008). SOCRADES: A web service based shop floor integration infrastructure. In *The Internet of Things. Lecture Notes in Computer Science* (Vol. 4952): Springer, Berlin, Heidelberg.
- Derhamy, H., Eliasson, J., & Delsing, J. (2017). IoT Interoperability—On-Demand and Low Latency Transparent Multiprotocol Translator. *IEEE Internet of Things Journal*, 4(5), 1754 - 1763. DOI: [10.1109/JIOT.2017.2697718](https://doi.org/10.1109/JIOT.2017.2697718)

Derhamy, H., Rönholm, J., Delsing, J., Eliasson, J., & Deventer, J. v. (2017). *Protocol interoperability of OPC UA in service oriented architectures*. Paper presented at the 2017 IEEE 15th International Conference on Industrial Informatics (INDIN), Emden, Germany. Retrieved from <https://ieeexplore.ieee.org/document/8104744/>

Desai, P., Sheth, A., & Anantharam, P. (2015, Jun 27-Jul 02). *Semantic Gateway as a Service architecture for IoT Interoperability*. Paper presented at the IEEE 3rd International Conference on Mobile Services MS, New York, NY (pp. 313-319).DOI: [10.1109/MobServ.2015.51](https://doi.org/10.1109/MobServ.2015.51)

Dionisio, R., Malhao, S., & Torres, P. (2020). Development of a Smart Gateway for a Label Loom Machine using Industrial IoT Technologies. *International Journal of Online and Biomedical Engineering (iJOE)*, 16(4), 6-14. DOI: [10.3991/ijoe.v16i04.11853](https://doi.org/10.3991/ijoe.v16i04.11853)

Elattar, M., Wendt, V., & Jasperneite, J. (2017). *Communications for Cyber-Physical Systems*. Springer Series in Wireless Technology: Springer, Cham.

ElMaraghy, H. A. (2005). Flexible and reconfigurable manufacturing systems paradigms. *International Journal of Flexible Manufacturing Systems*, 17(4), 261-276. DOI: [10.1007/s10696-006-9028-7](https://doi.org/10.1007/s10696-006-9028-7)

Foster, A. (2017). Messaging technologies for the industrial internet and the internet of things whitepaper. *PrismTech*.

Garrocho, C. T. B., Klippel, E., Machado, A. V., Ferreira, C. M. S., Cavalcanti, C., & Oliveira, R. A. R. (2020, Nov 23-27). *Blockchain-based Machine-to-Machine Communication in the Industry 4.0 applied at the Industrial Mining Environment*. Paper presented at the 10th Brazilian Symposium on Computing Systems Engineering (SBESC), Florianopolis, Brazil (pp. 1-8).DOI: [10.1109/SBESC51047.2020.9277852](https://doi.org/10.1109/SBESC51047.2020.9277852)

Givehchi, O., Landsdorf, K., Simoens, P. T. W., & Colombo, A. W. (2017). Interoperability for industrial cyber-physical systems : an approach for legacy systems. *IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS*, 13 3370–3378. DOI: [10.1109/TII.2017.2740434](https://doi.org/10.1109/TII.2017.2740434)

González, I., Calderón, A. J., & Portalo, J. M. (2021). Innovative Multi-Layered Architecture for Heterogeneous Automation and Monitoring Systems: Application Case of a Photovoltaic Smart Microgrid. *Sustainability* 2021, 13(4), 2234. DOI: [10.3390/su13042234](https://doi.org/10.3390/su13042234)

Grangel-González, I. (2017). *Semantic Data Integration for Industry 4.0 Standards*. Paper presented at the European Knowledge Acquisition Workshop. Retrieved from https://link.springer.com/chapter/10.1007/978-3-319-58694-6_36

Guinard, D., & Trifa, V. (2016). *Building the web of things: with examples in node.js and raspberry pi*: Manning Publications Co.

Hatzivasilis, G., Askoxylakis, I., Alexandris, G., Anicic, D., Bröring, A., Kulkarni, V., . . . Spanoudakis, G. (2018). *The Interoperability of Things: Interoperable solutions as an enabler for IoT and Web 3.0*. Paper presented at the 2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Barcelona, Spain. Retrieved from <https://ieeexplore.ieee.org/document/8514952/>

Henschke, M., Wei, X., & Zhang, X. (2020). *Data Visualization for Wireless Sensor Networks Using ThingsBoard*. Paper presented at the 2020 29th Wireless and Optical Communications Conference (WOCC). Retrieved from <https://ieeexplore.ieee.org/document/9114929/>

Hermann, M., Pentek, T., & Otto, B. (2015). *Design Principles for Industrie 4.0 Scenarios: A Literature Review*.

Honkola, J., Laine, H., Brown, R., & Oliver, I. (2009). Cross-Domain Interoperability: A Case Study. *Smart Spaces and Next Generation Wired/Wireless Networking* 22-31. DOI: [10.1007/978-3-642-04190-7_3](https://doi.org/10.1007/978-3-642-04190-7_3)

Iatrou, C. P., & Urbas, L. (2016a). Efficient OPC UA binary encoding considerations for embedded devices. *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)* 1148-1153. DOI: [10.1109/INDIN.2016.7819339](https://doi.org/10.1109/INDIN.2016.7819339)

Iatrou, C. P., & Urbas, L. (2016b). *OPC UA hardware offloading engine as dedicated peripheral IP core*. Paper presented at the 2016 IEEE World Conference on Factory Communication Systems (WFCS), Aveiro, Portugal. Retrieved from <https://ieeexplore.ieee.org/document/7496520/>

Izza, S. (2009). Integration of industrial information systems: from syntactic to semantic integration approaches. 3 1-57. DOI: [10.1080/17517570802521163](https://doi.org/10.1080/17517570802521163)

John, T., & Vorbröcker, M. (2020). *Enabling IoT connectivity for ModbusTCP sensors*. Paper presented at the 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). Retrieved from <https://ieeexplore.ieee.org/document/9211999/>

- Kadadi, A., Agrawal, R., Nyamful, C., & Atiq, R. (2014). Challenges of data integration and interoperability in big data. *2014 IEEE International Conference on Big Data (Big Data)* 38-40. DOI: [10.1109/BigData.2014.7004486](https://doi.org/10.1109/BigData.2014.7004486)
- Kagermann, H., Helbig, J., Hellinger, A., & Wahlster, W. (2013). *Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry; final report of the Industrie 4.0 Working Group*: Forschungsunion.
- Kang, B., & Choo, H. (2018). An experimental study of a reliable IoT gateway. *ICT Express*, 4(3), 130-133. DOI: [10.1016/j.ict.2017.04.002](https://doi.org/10.1016/j.ict.2017.04.002)
- Kant, R., & Bhattacharya, S. (2017). Sensors for Air Monitoring. In *Environmental, Chemical and Medical Sensors* (pp. 9- 30): SpringerLink.
- Kinnera, B. K. S., Subbareddy, S., & Luhach, A. (2019). IOT based Air Quality Monitoring System Using MQ135 and MQ7 with Machine Learning Analysis. *Scalable Computing: Practice and Experience*, 20 599- 606. DOI: [10.12694/scpe.v20i4.1561](https://doi.org/10.12694/scpe.v20i4.1561)
- Kshetri, N. (2017). Can Blockchain Strengthen the Internet of Things? *IT Professional*, 19(4), 68-72. DOI: [10.1109/MITP.2017.3051335](https://doi.org/10.1109/MITP.2017.3051335)
- Kubicek, H., Cimander, R., & Scholl, H. (2011). Layers of Interoperability. In *Organizational Interoperability in E-Government* (pp. 85-96): Springer, Berlin, Heidelberg.
- Lelli, F. (2019). Interoperability of the Time of Industry 4.0 and the Internet of Things. *Future Internet*, 11(2), 36. DOI: [10.3390/fi11020036](https://doi.org/10.3390/fi11020036)
- Li, H. S., Lai, L. F., & Poor, H. V. (2012). Multicast Routing for Decentralized Control of Cyber Physical Systems with an Application in Smart Grid. *Ieee Journal on Selected Areas in Communications*, 30(6), 1097-1107. DOI: [10.1109/JSAC.2012.120708](https://doi.org/10.1109/JSAC.2012.120708)
- Light, R. (2017). Mosquitto: server and client implementation of the MQTT protocol. *The Journal of Open Source Software*, 2. DOI: [10.21105/joss.00265](https://doi.org/10.21105/joss.00265)
- Loskyll, M. (2012). Towards Semantic Interoperability in Industrial Production. In *Semantic Interoperability: Issues, Solutions, Challenges* (pp. 71-104).

Lu, Y. Q., & Asghar, M. R. (2020). Semantic communications between distributed cyber-physical systems towards collaborative automation for smart manufacturing. *Journal of Manufacturing Systems*, 55 348-359. DOI: [10.1016/j.jmsy.2020.05.001](https://doi.org/10.1016/j.jmsy.2020.05.001)

Maciej Wasilak, & Amsüss, C. (2014). chrysn/aiocoap. Retrieved from <http://github.com/chrysn/aiocoap/>

Mai, S., Vu, V. T., & Myeong-Jae, Y. (2011). *An OPC UA client development for monitoring and control applications*. Paper presented at the Proceedings of 2011 6th International Forum on Strategic Technology, Harbin, China. Retrieved from <https://ieeexplore.ieee.org/document/6021120/>

Mattsson, S., Karlsson, M., Fast-Berglund, A., & Hansson, I. (2014, Apr 28-30). *Managing production complexity by empowering workers: six cases*. Paper presented at the 47th CIRP Conference on Manufacturing Systems, Univ Windsor, Windsor, Canada (Vol. 17, pp. 212-217). DOI: [10.1016/j.procir.2014.02.041](https://doi.org/10.1016/j.procir.2014.02.041)

Meng, Y., Yang, Y., Chung, H., Lee, P.-H., & Shao, C. (2018). Enhancing Sustainability and Energy Efficiency in Smart Factories: A Review. *Sustainability*, 10(12). DOI: [10.3390/SU10124779](https://doi.org/10.3390/SU10124779)

Meng, Z. Z., Wu, Z. P., & Gray, J. (2017). A Collaboration-Oriented M2M Messaging Mechanism for the Collaborative Automation between Machines in Future Industrial Networks. *Sensors*, 17(11). DOI: [10.3390/s17112694](https://doi.org/10.3390/s17112694)

Mourad, M. H., Nassehi, A., Schaefer, D., & Newman, S. T. (2020). Assessment of interoperability in cloud manufacturing. *Robotics and Computer-integrated Manufacturing*, 61. DOI: [10.1016/j.rcim.2019.101832](https://doi.org/10.1016/j.rcim.2019.101832)

Naik, N. (2017). *Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP*. Paper presented at the 2017 IEEE International Systems Engineering Symposium (ISSE). Retrieved from <https://ieeexplore.ieee.org/document/8088251/>

Pan, F., Pang, Z., Wen, H., Luvisotto, M., Xiao, M., Liao, R., & Chen, J. (2019). Threshold-Free Physical Layer Authentication Based on Machine Learning for Industrial Wireless CPS. *IEEE Transactions on Industrial Informatics*, 15(12), 6481-6491. DOI: [10.1109/TII.2019.2925418](https://doi.org/10.1109/TII.2019.2925418)

Paolis, L. T. D., Luca, V. D., & Paiano, R. (2018). *Sensor data collection and analytics with thingsboard and spark streaming*. Paper presented at the 2018 IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS). Retrieved from <https://ieeexplore.ieee.org/document/8405822/>

Parto, M. (2017). *A secure MTConnect compatible IoT platform for machine monitoring through integration of fog computing, cloud computing, and communication protocols*. (Georgia Institute of Technology). Retrieved from <https://smartech.gatech.edu/bitstream/handle/1853/59283/PARTODEZFOULI-THESIS-2017.pdf>

Qiao, L., & Feng, L. (2011). *The future of the device integration: Field device integration*. Paper presented at the 2011 IEEE 2nd International Conference on Software Engineering and Service Science, Beijing. Retrieved from <https://ieeexplore.ieee.org/document/5982422/>

R.L., C. (2005). Toward technical interoperability in telemedicine. *Telemed J E Health*, 11 384–404. DOI: [10.1089/tmj.2005.11.384](https://doi.org/10.1089/tmj.2005.11.384)

Schuh, G., Potente, T., Varandani, R., Hausberg, C., & Fränken, B. (2014). Collaboration Moves Productivity to the Next Level. *Procedia CIRP*, 17 3-8. DOI: [10.1016/j.procir.2014.02.037](https://doi.org/10.1016/j.procir.2014.02.037)

Serpanos, D., & Wolf, M. (2018). Industrial Internet of Things. In *Internet-of-Things (IoT) Systems: Architectures, Algorithms, Methodologies* (pp. 37-54). Cham: Springer International Publishing.

Shang, C., & You, F. (2019). Data Analytics and Machine Learning for Smart Process Manufacturing: Recent Advances and Perspectives in the Big Data Era. *Engineering*, 5(6), 1010-1016. DOI: [10.1016/j.eng.2019.01.019](https://doi.org/10.1016/j.eng.2019.01.019)

Shrouf, F., Ordieres, J., & Miragliotta, G. (2014). Smart factories in Industry 4.0: A review of the concept and of energy management approached in production based on the Internet of Things paradigm. *2014 IEEE International Conference on Industrial Engineering and Engineering Management* 697-701. DOI: [10.1109/IEEM.2014.7058728](https://doi.org/10.1109/IEEM.2014.7058728)

Silva, D., Carvalho, L. I., Soares, J., & Sofia, R. C. (2021). A Performance Analysis of Internet of Things Networking Protocols: Evaluating MQTT, CoAP, OPC UA. *Applied Sciences*, 11(11). DOI: [10.3390/app11114879](https://doi.org/10.3390/app11114879)

Sisinni, E., Saifullah, A., Han, S., Jennehag, U., & Gidlund, M. (2018). Industrial Internet of Things: Challenges, Opportunities, and Directions. *Ieee Transactions on Industrial Informatics*, 14(11), 4724-4734. DOI: [10.1109/TII.2018.2852491](https://doi.org/10.1109/TII.2018.2852491)

Skvorc, D., Horvat, M., & Srbljic, S. (2014). *Performance evaluation of WebSocket protocol for implementation of full-duplex web streams*. Paper presented at the 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia. Retrieved from <https://ieeexplore.ieee.org/document/6859715/>

Uy, N. Q., & Nam, V. H. (2019). *A comparison of AMQP and MQTT protocols for Internet of Things*. Paper presented at the 2019 6th NAFOSTED Conference on Information and Computer Science (NICS). Retrieved from <https://ieeexplore.ieee.org/document/9023812/>

Vermesan, O., Friess, P., Guillemin, P., Gusmeroli, S., Sundmaecker, H., Bassi, A., Doody, P. (2009). Internet of Things Strategic Research Roadmap. *The Cluster of European Research Projects, Tech. Rep.*

W., M. (2014). *Industrie 4.0: Smart Manufacturing for the Future*. Berlin: Retrieved from <http://www.gtai.de/GTAI/Navigation/EN/Invest/Service/publications,did=917080.html>

Wang, L., Orban, P., Cunningham, A., & Lang, S. (2004). Remote real-time CNC machining for web-based manufacturing. *Robotics and Computer-integrated Manufacturing*, 20 563-571. DOI: [10.1016/J.RCIM.2004.07.007](https://doi.org/10.1016/J.RCIM.2004.07.007)

Wang, L., Törngren, M., & Onori, M. (2015). Current status and advancement of cyber-physical systems in manufacturing. *Journal of Manufacturing Systems*, 37 517-527. DOI: [10.1016/j.jmsy.2015.04.008](https://doi.org/10.1016/j.jmsy.2015.04.008)

Waurzyniak, P. (2001). Electronic intelligence in manufacturing. *Manufacturing Engineering*, 127 44-44.

Weyer, S., Schmitt, M., Ohmer, M., & Gorecky, D. (2015). Towards Industry 4.0 - Standardization as the crucial challenge for highly modular, multi-vendor production systems. *IFAC-PapersOnLine*, 48(3), 579-584. DOI: [10.1016/j.ifacol.2015.06.143](https://doi.org/10.1016/j.ifacol.2015.06.143)

Wilder, C., Jose, M., Harold, P., & Alvarado, J. D. (2021). Internet of things: a multiprotocol gateway as solution of the interoperability problem. In Bonaventuriana (Ed.), *Mechatronics, Electronics and Telecommunications Advances Towards Industry 4.0* (pp. 24): arXiv.

Wollschlaeger, M., Sauter, T., & Jasperneite, J. (2017). The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0. *IEEE Industrial Electronics Magazine*, 11(1), 17-27. DOI: [10.1109/MIE.2017.2649104](https://doi.org/10.1109/MIE.2017.2649104)

Xiao, W., Huang, H., & Zhao, G. (2018). Communication methodology between machine tools using MTConnect protocol. *MATEC Web of Conferences*, 175. DOI: [10.1051/mateconf/201817503066](https://doi.org/10.1051/mateconf/201817503066)

Zarko, I. P. (2019). Why Interoperability Matters to Your or Any IoT Solution. *2019 15th International Conference on Telecommunications (ConTEL)* 1-1. DOI: [10.1109/ConTEL.2019.8848558](https://doi.org/10.1109/ConTEL.2019.8848558)

Zeid, A., Sundaram, S., Moghaddam, M., Kamarthi, S., & Marion, T. (2019). Interoperability in Smart Manufacturing: Research Challenges. *Machines*, 7(2). DOI: [10.3390/machines7020021](https://doi.org/10.3390/machines7020021)

Zuehlke, D. (2008). SmartFactory – from Vision to Reality in Factory Technologies. *IFAC Proceedings Volumes*, 41(2), 14101-14108. DOI: [10.3182/20080706-5-KR-1001.02391](https://doi.org/10.3182/20080706-5-KR-1001.02391)

APPENDIX A NODE DEVICES CODE AND ALGORITHM

A.1 Node1 Raspberry Pi 3B Sending Temperature-Humidity Data to Gateway over HTTP, CoAP and ModbusTCP Protocol

```

import time
import paho.mqtt.client as mqtt
from pyModbusTCP.client import ModbusClient
import Adafruit_DHT
import datetime
import requests
import urllib

import logging
import asyncio
from aiocoap import *

DHT_SENSOR = Adafruit_DHT.DHT22
DHT_PIN = 4

ip = "192.168.0.106"

deviceId = 'node1'
client_server = mqtt.Client(deviceId)
client_server.username_pw_set(username="project", password="A_project_b")
host_server = ip

flag = 0
protocol_selection = 1
protocol_sub_topic = "/client/"+deviceId
mqtt_pub_topic = "/client/"+deviceId+"/mqtt"

websocket_pub_topic = "/client/"+deviceId+"/websocket"

http_url = "http://" + ip + "/http?"

SERVER_HOST = ip
SERVER_PORT = 502
c = ModbusClient()
c.host(SERVER_HOST)
c.port(SERVER_PORT)

logging.basicConfig(level=logging.INFO)

delay_time = 10

def on_message(client, userdata, message):
    if(message.topic == protocol_sub_topic):
        print("message received " ,str(message.payload.decode("utf-8")))
        print("message topic=",message.topic)
        print("message qos=",message.qos)

```

```

    print("message retain flag=",message.retain)

    global protocol_selection
    protocol_selection = int(str(message.payload.decode("utf-8")))

def protocol_selection_loop():

    if(protocol_selection == 2):

        print("http")

    elif(protocol_selection == 3):
        print("modbus")
    elif(protocol_selection == 4):
        print("coap")
    else:
        print("invalid")

def take_time():
    # print("reading time") #-----
    time_now = datetime.datetime.now()
    time_now = str(time_now)
    time_now = time_now.split(".")
    time_now = str(time_now[0])
    # print("from take time: "+ str(time_now))
    return time_now

async def main_put(str):

    context = await Context.create_client_context()
    # context = self.get_context_data(object=self.object)

    await asyncio.sleep(2)

    payload = str.encode("ascii")
    request = Message(code=PUT, payload=payload, token=None,
uri="coap://"+ip+"/put_request")

    response = await context.request(request).response

    print('Result: %s\n%r'%(response.code, response.payload.decode("ascii")))

def on_connect(client_server, userdata, flags, rc):
    global flag;
    flag = 1
    print("connected ok")

def on_pub (client_server, userdata, result):
    print("data published")

def on_disconnect(client, userdata, rc):
    global flag;
    flag = 0

```



```

print("disconnected")

while True:
    try:
        humidity, temperature = Adafruit_DHT.read_retry(DHT_SENSOR, DHT_PIN)
        if (flag == 0):
            client_server.connect(host_server, port=1883, keepalive=60,
bind_address="")
            client_server.on_message = on_message
            client_server.on_connect = on_connect
            client_server.on_disconnect = on_disconnect
            client_server.on_publish = on_pub
            client_server.subscribe(protocol_sub_topic)
            client_server.loop_start()

            if humidity is not None and temperature is not None:

                data_send =
"\Temperature\":"+str("{:.2f}".format(temperature))+","++"\Humidity\":"+str(
"{:.2f}".format(humidity))
#         data_send = "\Temperature\":"+str("{:.2f}".format(temperature))
temp = str("{:.2f}".format(temperature))
hum = str("{:.2f}".format(humidity))
payload = take_time()
payload += "|"
payload += data_send
print(payload)

                protocol_selection_loop()

                #Publish data over HTTP
                if(protocol_selection == 2): X
                    print("2")
                    temp_url = http_url
                    time_ = take_time()

                    f = { 't' : str(temp) , 'rh' : str(hum) , 'time' : str(time_) ,
'device': deviceId }
                    payload = urllib.parse.urlencode(f);
                    temp_url += payload
                    print(temp_url)

                    x = requests.get(temp_url)
                    print(x.status_code)
                    if(x.status_code == 200):
                        print(x.text)

                    else:
                        print("failed")

                    # time.sleep(delay_time)
                    #Publish data over Modbus TCP
                elif(protocol_selection == 3):
                    print("3")

```

```

        if not c.is_open():
            if not c.open():
                print("unable to connect to
"+SERVER_HOST+": "+str(SERVER_PORT))
            new_time = take_time()
            new_time = new_time.split(" ")

            date1 = new_time[0]
            date1 = date1.split("-")

            time1 = new_time[1]
            time1 = time1.split(":")

        if c.is_open():

            temp = int(float(temp) * 100)
            hum = int(float(hum) * 100)

            if (deviceId == 'node1'):
                is_ok = c.write_single_register(1, 1)
                is_ok = c.write_single_register(2, temp)
                is_ok = c.write_single_register(3, hum)
                is_ok = c.write_single_register(4, int(time1[0]))
                is_ok = c.write_single_register(5, int(time1[1]))
                is_ok = c.write_single_register(6, int(time1[2]))
                is_ok = c.write_single_register(7, int(date1[2]))
                is_ok = c.write_single_register(8, int(date1[1]))
                is_ok = c.write_single_register(9, int(date1[0]))

            #Publish data over CoAP
        elif(protocol_selection == 4):
            print("4")
            payload += "@"+deviceId
            asyncio.get_event_loop().run_until_complete(main_put(payload))

            # time.sleep(delay_time)

        else:
            print("invalid")

            time.sleep(delay_time)

    else:
        print("Failed to get data from DHT22")
        # 2021-11-01 11:42:46|T:25.00,RH:72.60

except Exception as e:
    print("Exception Message: {} ".format(e))

```

A.2 Node2 Arduino UNO Wi-Fi Sending BME280 Sensor Pressure and Altitude Data to Gateway over WebSocket Protocol

```

#include <WiFi.h>
#include <WebServer.h>
#include <WebSocketsClient.h>
#include <ArduinoJson.h>
#include <Wire.h>
#include <SPI.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
#define BME_SCK 13
#define BME_MISO 12
#define BME_MOSI 11
#define BME_CS 10
// Wifi Credentials
const char* ssid = "MagicMan"; // Wifi SSID
const char* password = "Sabbir1234"; //Wi-Fi Password
WebSocketsClient webSocket; // websocket client class instance
StaticJsonDocument<100> doc; // Allocate a static JSON document

void setup() {
  // Connect to local WiFi
  WiFi.begin(ssid, password);
  Serial.begin(9600);
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(500);
  }
  Serial.println();
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP()); // Print local IP address
  delay(2000); // wait for 2s
  //address, port, and URL path
  webSocket.begin("192.168.0.106", 81, "/");
  // webSocket event handler
  webSocket.onEvent(webSocketEvent);
  // if connection failed retry every 5s
  webSocket.setReconnectInterval(5000);
}

void loop() {
  webSocket.loop(); // Keep the socket alive
}

void webSocketEvent(WStype_t type, uint8_t * payload, size_t length) {
  if (type == WStype_TEXT)
  {
    DeserializationError error = deserializeJson(doc, payload); // deserialize
incoming Json String
    if (error) { // Print erro msg if incomig String is not JSON formated
      Serial.print(F("deserializeJson() failed: "));
      Serial.println(error.c_str());
      return;
    }
    const String pin_stat = doc["PIN_Status"];
    const float BME280_Pdata=bme.readPressure() / 100.0F;
    const float BME280_Adata=bme.readAltitude(SEALEVELPRESSURE_HPA);
  }
}

```

```

    // Print the received data for debugging
    Serial.print(String(pin_stat));
    Serial.print(String(BME280_Pdata));
    Serial.println(String(BME280_Adata));
}
}

```

A.3 Node3 ESP32 Sending MQ135 Sensor Air Quality Data to Gateway over MQTT Protocol

```

#include <WiFi.h>
#include <PubSubClient.h>
#include <SPI.h>
#include <WiFiClientSecure.h>
#define MQ135_THRESHOLD_1 300

const int anPin = 35; //set analog pin
const char* ssid = "MagicMan";//WIFI ssid
const char* password = "Sabbir1234";//Wifi password
const char* mqtt_server = "192.168.0.106";//mqtt server
const char* mqtt_username = "project";
const char* mqtt_password = "A_project_b";
const char* mqtt_topic = "/client/node3/mqtt";
const char* clientID = "client_RFID"; // MQTT client ID
WiFiClient wifiClient;
//WiFiClientSecure wifiClient;
PubSubClient client(mqtt_server, 1883, wifiClient);

void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("message arrived : ");
  Serial.println(topic);

  Serial.print("messahe: ");
  for (int i = 0; i < length; i++) {
    Serial.println((char)payload[i]);
  }
  Serial.println();
  if (String(topic)==" /device/node3/mqtt") {
    Serial.print("hagu");
    if (topic=="1") {
      Serial.println("one");
    }
    else if (topic=="0"); {
      Serial.println("zero");
    }
  }
  Serial.println(".....");
}

void connect_MQTT(){
  if (client.connect(clientID, mqtt_username, mqtt_password)) {
    Serial.println("Connected to MQTT Broker!");
    client.subscribe("/device/node2/websocket");
  }
}

```

```

    else {
        Serial.println("Connection to MQTT Broker failed...");
    }
}

void setup_wifi() {
    delay(10);
    // We start by connecting to a WiFi network
    Serial.println();
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    randomSeed(micros());
    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
}

void setup() {
    Serial.begin(9600);
    delay(1000);
    setup_wifi();
    delay(500);
    connect_MQTT();
    Serial.setTimeout(2000);
    client.setCallback(callback);
    client.subscribe("/device/node2/websocket");
    Serial.println("done");
}

void loop() {
    //connect_MQTT();
    client.subscribe("/device/node2/websocket");
    delay(3000);
    int MQ135_data = (analogRead(anPin))/11.5;
    Serial.print("AirQuality Index: ");
    Serial.print(MQ135_data);
    delay(500);
    Serial.println("PPM");
    String ppm = " PPM";
    String qual= ("\AirQuality\": ");
    delay(2000);
    //client.publish(mqtt_topic, String(qual+MQ135_data+ ppm).c_str());
    client.publish(mqtt_topic, String(qual+MQ135_data).c_str()); //publish MQ135
    data to broker
    Serial.println("Published mama");
}

```

APPENDIX B GATEWAY CODE AND ALGORITHM

B.1 KEPServerEX Data Logging and Communication on Gateway from Node 3 ESP32 over MQTT Protocol

```

import paho.mqtt.client as mqtt
import paho.mqtt.client as mqtt
import requests
import sys
import MySQLdb
import mysql.connector
from datetime import date, datetime, timedelta
import time
import json

MQTT_ADDRESS = '192.168.0.106'
MQTT_USER = 'project'
MQTT_PASSWORD = 'A_project_b'
MQTT_TOPIC = "/client/node3/mqtt"
user_mysql = "admin"
pwd_mysql = "raspberry"

topic = "AirQuality"

openDoor = json.dumps({"val1": "2500"})
client_id = "rfidManager"
topic = "AirQuality"

openDoor = json.dumps({"val1": "2500"})

def on_connect(client, userdata, flags, rc):
    print('Connected with result code ' + str(rc))
    client.subscribe(MQTT_TOPIC)

def on_message(client, userdata, message):
    message.payload = message.payload.decode("utf-8")
    print(str(message.payload))
    p= str(message.payload)
    data = {"Value" : p}

```

```

    imei= json.dumps(data)
    publish(client, imei)
    #b=open("/home/pi/abc.txt", "a")
    #b.write(str(msg.payload))
def publish (client, imei):
    client.publish(topic, imei)

def main():
    mqtt_client = mqtt.Client()
    mqtt_client.username_pw_set(MQTT_USER, MQTT_PASSWORD)
    mqtt_client.on_connect = on_connect
    mqtt_client.on_message = on_message
    #    mqtt_client.on_publish = on_publish

    mqtt_client.connect(MQTT_ADDRESS, 1883)
    mqtt_client.loop_forever()

if __name__ == '__main__':
    print('MQTT to InfluxDB bridge')
    main()

```

B.2 Gateway Transferring Data to Local MySQL Database

```

import MySQLdb
import paho.mqtt.client as mqtt
from time import sleep

client1_http = "/client/node1/http"
client1_modbus = "/client/node1/modbus"
client1_coap = "/client/node1/coap"
client2_websocket = "/client/node2/websocket"
client3_mqtt = "/client/node3/mqtt"

ip = "192.168.0.106"

deviceId = 'server_db_saver'
client_server = mqtt.Client(deviceId)
client_server.username_pw_set(username="project", password="A_project_b")
host_server = ip

```

```
flag = 0
```

```
db = MySQLdb.connect(host="localhost",user="admin", passwd="!shahed@",db="server")
cur = db.cursor()
```

```
def on_message(client, userdata, message):

    print("message topic=",message.topic)
    msg = str(message.payload.decode("utf-8"))
    print(msg)
    png = msg.split("|")
    time = str(png[0])
    print(time)
    print (png)
    data = str(png)
    print(data)

    if(message.topic == client1_coap):
        protocol = "4"
        node = "1"
        cur.execute('INSERT INTO data(protocol_id,device_id,data,time)
VALUES (%s,%s,%s,%s)', (protocol,node,data,time))

    elif(message.topic == client1_modbus):
        protocol = "3"
        node = "1"
        x = cur.execute('INSERT INTO data(protocol_id,device_id,data,time)
VALUES (%s,%s,%s,%s)', (protocol,node,data,time))
        print(x)

    elif(message.topic == client1_http):
        protocol = "2"
        node = "1"
        cur.execute('INSERT INTO data(protocol_id,device_id,data,time)
VALUES (%s,%s,%s,%s)', (protocol,node,data,time))

    elif(message.topic == client2_websocket):
        protocol = "5"
        node = "2"
```



```

        cur.execute('INSERT INTO data(protocol_id,device_id,data) VALUES(%s,%s,%s)',(protcol,node,data))

    elif(message.topic == client3_mqtt):
        protcol = "1"
        node = "3"

        cur.execute('INSERT INTO data(protocol_id,device_id,data,time)
VALUES(%s,%s,%s,%s)',(protcol,node,data,time))
        db.commit()

def on_connect(client_server, userdata, flags, rc):
    global flag;
    flag = 1
    print("connected ok")

def on_disconnect(client, userdata, rc):
    global flag;
    flag = 0
    print("disconnected")

while True:
    if (flag == 0):
        client_server.connect(host_server, port=1883, keepalive=60, bind_address="")
        client_server.on_message = on_message
        client_server.on_connect = on_connect
        client_server.on_disconnect = on_disconnect
        client_server.subscribe(client1_http)
        client_server.subscribe(client1_modbus)
        client_server.subscribe(client1_coap)
        client_server.subscribe(client2_websocket)
        client_server.subscribe(client3_mqtt)
        client_server.loop_start()

    sleep(1)

```

B.3 Gateway Transferring Node 1 Data to ThingsBoard Cloud Platform

```

import paho.mqtt.client as paho           #mqtt library
import paho.mqtt.client as mqtt
import paho.mqtt.client as mqtt
import os
import json
import time
from datetime import datetime
from time import sleep
ACCESS_TOKEN='UEUAObQdVgtGjbRDXj80'     #Token of your device
broker="thingsboard.cloud"              #host name
port=1883                                #data listening port
MQTT_ADDRESS = '192.168.0.106'
MQTT_USER = 'project'
MQTT_PASSWORD = 'A_project_b'
MQTT_TOPIC = "/client/node1/mqtt"

def on_connect(client, userdata, flags, rc):
    print('Connected with result code ' + str(rc))
    client.subscribe(MQTT_TOPIC)

def on_publish(client,userdata,result):   #create function for callback
    print("data published to thingsboard \n")
    pass
client1= paho.Client("control1")         #create client object
client1.on_publish = on_publish          #assign function to callback
client1.username_pw_set(ACCESS_TOKEN)    #access token from thingsboard device
client1.connect(broker,port,keepalive=60)#establish connection

#while True:
def on_message(client, userdata, message):
    payload= message.payload.decode("utf-8")
    print(str(message.payload))
    png = payload.split("|")
    time = str(png[0])
    print(time)
    data="{ "
    data+= str(png[1])

```

```
data+="}"
ret= client1.publish("v1/devices/me/telemetry",data) #topic-v1/devices/me/telemetry
print("Gateway Publishing Nodel data to ThingsBoard Cloud")
print(data);
# time.sleep(5)

def main():
    mqtt_client = mqtt.Client()
    mqtt_client.username_pw_set(MQTT_USER, MQTT_PASSWORD)
    mqtt_client.on_connect = on_connect
    mqtt_client.on_message = on_message
#    mqtt_client.on_publish = on_publish

    mqtt_client.connect(MQTT_ADDRESS, 1883)
    mqtt_client.loop_forever()

if __name__ == '__main__':
    print('MQTT to InfluxDB bridge')
    main()
```