Prototyping and Evaluation of Sensor Data Integration in Cloud Platforms

Marcus Korsnes Morlandstø

Master's thesis in Software Engineering

Department of Computer science, Electrical engineering and Mathematical sciences,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

September 2022

Western Norway
University of
Applied Sciences

Abstract

The SFI Smart Ocean centre has initiated a long-running project which consists of developing a wireless and autonomous marine observation system for monitoring of underwater environments and structures. The increasing popularity of integrating the Internet of Things (IoT) with Cloud Computing has led to promising infrastructures that could realize Smart Ocean's goals. The project will utilize underwater wireless sensor networks (UWSNs) for collecting data in the marine environments and develop a cloud-based platform for retrieving, processing, and storing all the sensor data. Currently, the project is in its early stages and the collaborating partners are researching approaches and technologies that can potentially be utilized. This thesis contributes to the centre's ongoing research, focusing on the aspect of how sensor data can be integrated into three different cloud platforms: Microsoft Azure, Amazon Web Services, and the Google Cloud Platform. The goals were to develop prototypes that could successfully send data to the chosen cloud platforms and evaluate their applicability in context of the Smart Ocean project. In order to determine the most suitable option, each platform was evaluated based on set of defined criteria, focusing on their sensor data integration capabilities. The thesis has also investigated the cloud platforms' supported protocol bindings, as well as several candidate technologies for metadata standards and compared them in surveys. Our evaluation results shows that all three cloud platforms handle sensor data integration in very similar ways, offering a set of cloud services relevant for creating diverse IoT solutions. However, the Google Cloud Platform ranks at the bottom due to the lack of IoT focus on their platform, with less service options, features, and capabilities compared to the other two. Both Microsoft Azure and Amazon Web Services rank very close to each other, as they provide many of the same sensor data integration capabilities, making them the most applicable options.

Acknowledgements

I would like to thank my supervisors, Lars Michael Kristensen and Tosin Daniel Oyetoyan for supporting me the whole way, and providing feedback and guidance when I needed it. I also want to thank fellow Smart Ocean master student, Tore Berven for his collaboration on the development of the Real-Time Collector client interface, that we used separately for our own thesis goals.

Contents

1 Introduction **6**
1.1 Context . . . 7
1.1.1 Work packages . . . 7
1.1.2 Pilot Demonstrators . . . 8
1.1.3 Motivation . . . 8
1.2 Problem Description . . . 9
1.3 Methodology . . . 10
1.4 Outline . . . 11

2 Background **12**
2.1 Internet of Things . . . 12
2.1.1 IoT Elements . . . 13
2.1.2 General Layered IoT Architecture . . . 14
2.1.3 Internet of Underwater Things . . . 16
2.2 Cloud Computing . . . 17
2.2.1 Service Models . . . 17
2.2.2 IoT Cloud Platform Components . . . 18
2.2.3 Sensor-Cloud Infrastructure . . . 19
2.3 Application Layer for IoT . . . 21
2.3.1 Data Serialization Formats . . . 21
2.3.2 Communication Models . . . 22
2.3.3 Representational State Transfer (REST) . . . 23
2.4 Interoperability . . . 25
2.4.1 Interoperability Levels . . . 25
2.4.2 Metadata . . . 25
2.4.3 IoT Frameworks . . . 26

3 A Survey on Application Layer Protocols and Metadata Standards **28**
3.1 Application Layer Protocols . . . 28
3.1.1 HyperText Transfer Protocol (HTTP) . . . 28
3.1.2 Message Queuing Telemetry Transport (MQTT) . . . 29
3.1.3 Advanced Message Queuing Protocol (AMQP) . . . 29
3.1.4 Constrained Application Protocol (CoAp) . . . 30
3.1.5 WebSocket . . . 30
3.1.6 Protocol Summary . . . 31
3.2 Metadata Industry Standards . . . 32

3.2.1 IP for Smart Objects (IPSO) 32
3.2.2 Open Connectivity Foundation (OCF) 34
3.2.3 Open Platform Communications Unified Architecture (OPC UA) 36
3.2.4 Open Geospatial Consortium's Sensor Web Enablement (OGC SWE) 37
3.2.5 World Wide Web Consortium (W3C) 39
3.2.6 Metadata Summary 41

4 Cloud Platforms 44
4.1 Microsoft Azure 44
4.1.1 Azure IoT Overview 44
4.1.2 Azure Protocol Bindings, SDKs and APIs 46
4.2 Amazon Web Services 47
4.2.1 AWS IoT Overview 48
4.2.2 AWS Protocol Bindings, SDKs, and APIs 49
4.3 Google Cloud Platform 50
4.3.1 GCP IoT Overview 50
4.3.2 GCP Protocol Bindings, SDKs, and APIs 51
4.4 Cloud Platform Summary 52

5 Design and Implementation 55
5.1 Virtual Sensors and Client Interface 55
5.1.1 AADI Device Simulator 55
5.1.2 AADI Real-Time Collector 55
5.1.3 Client Interface 56
5.2 Prototype Design and Workflow 58
5.3 Prototype 1 - Microsoft Azure 58
5.3.1 Azure IoT Hub and Device Setup 59
5.3.2 Azure Implementation with Device SDK 60
5.3.3 Azure Deployment and Output 62
5.4 Prototype 2 - Amazon Web Services 63
5.4.1 AWS IoT Core and Device Setup 63
5.4.2 AWS Implementation with HTTPS 66
5.4.3 AWS Implementation with MQTT 68
5.4.4 AWS Deployment and Output 69
5.5 Prototype 3 - Google Cloud Platform 70
5.5.1 GCP IoT Core and Device Setup 71
5.5.2 GCP data integration with HTTPS 71
5.5.3 GCP Implementation with MQTT 76
5.5.4 GCP Deployment and Output 78

6 Evaluation 80
6.1 Evaluation Criteria 80
6.1.1 Interoperability 80
6.1.2 IoT Support 80
6.1.3 Device Connectivity 81
6.1.4 Data Management 81
6.1.5 Ease of Implementation 81
6.2 Cloud Platform Evaluations 81

6.2.1 Microsoft Azure Evaluation 81
6.2.2 Amazon Web Services Evaluation 83
6.2.3 Google Cloud Platform Evaluation 85

7 Conclusion and Future Work **88**
7.1 Summary 88
7.2 Research Questions 89
7.2.1 Research Question 1 89
7.2.2 Research Question 2 90
7.2.3 Research Question 3 90
7.3 Conclusion 91
7.4 Related Work 92
7.5 Future Work 92

A Source code **96**

B XML Sensor Data Sample from the AADI Device Simulator **97**

Chapter 1

Introduction

The Norwegian ocean industries are among the world leaders in operations and technology in offshore, petroleum and fisheries. The oceans are critical for food supply, climate regulation, transportation and energy production. As a result, there is a need for efficient and reliable ways of collecting data and monitoring changes in the marine environment. The SFI Smart Ocean project [182] has a goal to develop a flexible, wireless and autonomous marine observation system that will ensure sustainable use and monitoring of underwater environments, installations, and ocean resources.

Over the last decade, the use of Internet of Things (IoT) technology has been rapidly increasing and has opened for new approaches for creating smarter and more efficient systems and applications. The different IoT devices can sense and collect data from their surrounding environments and send the data to their corresponding IoT system via the Internet. Unfortunately, IoT also has many limitations when it comes to storage, scalability, computation, and data management. On the other hand, Cloud Computing provides massive computing and storage capabilities, high scalability, software services, and efficient data management. Therefore, IoT is now more often integrated with Cloud Computing to make up for its limitations. With this approach, it is possible to create a high-performance computing and storage infrastructure for real-time processing and storing of data from millions of IoT devices, that collaborates with several applications in a cloud environment among multiple organizations, enabling information sharing on a large scale [8].

There exists many extensions of the IoT-term that are being used to differentiate the functionalities and characteristics of "things". The Internet of Underwater Things (IoUT) [190] is a novel class of IoT, and is defined as "a complex system that consists of multiple heterogeneous sensor networks that can sense, monitor and identify underwater objects by wired or wireless communication." The platform to be developed in the Smart Ocean project will use the IoUT as a foundation, and integrate underwater wireless sensor networks (UWSN) for enabling more flexible and cost-effective approaches for multi-parameter monitoring in the ocean. In combination with Cloud Computing, the final Smart Ocean platform will support a complete data value chain from communication and processing, to storage capabilities, big data analytics and visualization [182]. The approach of integrating wireless sensor networks with Cloud Computing is commonly referred to as a sensor-cloud infrastructure [269].

There are still aspects in the Smart Ocean project that requires further research. One of the topics is investigating how sensor data can be integrated and handled in cloud platforms. There exist many candidate technologies and approaches, but not all will have the capabilities or meet the requirements needed to realize the SFI Smart Ocean's vision. In this thesis, we investigate and experimentally evaluate three selected cloud platforms. The selected cloud platforms are Microsoft Azure, Amazon Web Services, and Google Cloud Platform. We develop prototypes for each cloud platform, in order to test their sensor data integration capabilities. Relevant protocol bindings, service APIs and industry standards for sensor metadata are also be investigated. The main goal is to put forward potential technologies and solutions that can be used for sensor data integration in the future Smart Ocean platform.

1.1 Context

The SFI Smart Ocean is a centre for research-based innovation, hosted by the Department of Physics and Technology at the University of Bergen (UIB). The project involves active cooperation between several research and industry partners, as well as observers from national authorities [183]. The Western Norway University of Applied Sciences (HVL) is one of the research partners collaborating on the project. The work associated with this thesis contributes to the ongoing tasks in WP3 (see section 1.1.1) that are coordinated by HVL. The project is currently in its early stages and the project's research tasks and scientific methods are split into three different work packages and four pilot demonstrators.

1.1.1 Work packages

Each work package (WP) has their own activities and focus areas, where all the partners are split up to work and collaborate on one or more of these different packages. The three WPs are as follows [182, 184].

WP1 - Autonomous sensors and measurement strategies: This WP evolves around development of autonomous sensor technologies, and measurement methods and strategies for improving real-time monitoring of underwater environments and structures. The marine smart sensors will include embedded preprocessing and compression of data, acoustic modem compatibility, and smart operation for low energy use.

WP2 - Wireless network communication: This WP concerns the communication technology aspects of the project. It involves development of an acoustic underwater wireless technology platform that will be assembled to energy-efficient UWSNs for long-term marine monitoring. The hardware and software components also needs to be optimized in terms of limited battery capacity, efficiency, and reliability.

WP3 - Software technology and big-data middleware: This WP is coordinated by HVL. It includes development and implementation of the cloud-based Smart Ocean platform and will comprise a set of software frameworks that enables the integration of external and internal ocean data sources, data storage and processing, application deployment, and visualization of big-data sets. The platform will enable data spaces based on a uniform and standardized set of APIs and data formats. Its main function will be to receive and process the data produced by the heterogeneous underwater sensors.

1.1.2 Pilot Demonstrators

The pilot demonstrators (PDs) constitute an overarching work package where the objective is to establish in-sea facilities and environments for research and testing of the different components that is being developed under WP1-WP3 and demonstrate these system components under realistic operational conditions. Four PDs have currently been defined by Smart Ocean, but the number of test sites is presumed to be expanded during the project's life cycle [182, 184].

PD1 - Local scale environmental monitoring: A test site is set up at the Austevoll Research Station near Bergen. It provides a multipurpose local-scaled wireless network of autonomous sensors that will be used for monitoring oceanographic and seabed environmental parameters. Several modular rigs with sensors are being placed where a range of sensors and communication systems can be included. PD1 will contain and demonstrate all major Smart Ocean monitoring system components and functionalities. During the time of writing this thesis, PD1 is the only case study accessible while the other PDs will be available later.

PD2 - Mesoscale environmental monitoring: A real-time and scalable ocean multipurpose observing system will later be established as an extension of PD1 at Austevoll, paving the way for longer scale communication, geo-positioning, and mesoscale environmental monitoring, using acoustic tomography and passive acoustics.

PD3 - Integrity measurements offshore wind: Another test site will be established and used for research, development, testing, and demonstration of sensors for integrity monitoring of bottom-mounted and floating wind turbine structures.

PD4 - Integrity measurements oil and gas: Similar to PD3, a test site will be established and used for research, development, testing, and demonstration of sensors, but in this case it is for integrity monitoring of oil and gas installations.

1.1.3 Motivation

Realize the SFI Smart Ocean project will require a large amount of various resources. Creating a cloud-based IoT platform involves many challenging aspects. By developing frameworks and utilizing existing technologies, standards and cloud platforms, there is potential to simplify many core aspects of the development and implementation processes in the Smart Ocean project. Hence, trying to narrow it down to some of the most relevant and promising options can go a long way. This thesis contributes to the ongoing research in WP3 on sensor data integration in the cloud, existing metadata industry standards, protocol bindings, and service APIs. The software technologies being developed in WP3 are validated through the development of prototypes that are linked to the consortium pilot demonstrators and through the deployment of reference implementations that integrates with external systems and data services. Since PD1 is currently the only available case study, we have focused on developing prototypes that are tested at this test site. The physical underwater sensors at Austevoll are not yet available for experimental evaluation, during the writing of this thesis. Instead, virtual sensors are used in order to simulate the behavior and data streams to one of the physical sensors related to the PD1 case study.

1.2 Problem Description

In order to contribute to the research and determine optimal solutions, we have specified the objectives to achieve, and the scope of the thesis. This thesis investigates the three chosen cloud platforms, develop prototypes and evaluate their applicability in a Smart Ocean context. In order to establish interoperability between all the various underwater sensors and the future Smart Ocean platform, existing industry standards for metadata are also investigated to present possible solutions for universally representing the sensors' descriptions, data syntax and semantics. When integrating sensor data to the three cloud platforms, we need to negotiate access to each platform based on their supported protocol bindings and service APIs. Hence, we have investigated relevant communication protocols and APIs for determining the most applicable options. The research questions given for this thesis are as follows:

R1 How does common cloud platforms such as Microsoft Azure, Amazon Web Services and Google Cloud Platform handle sensor data integration? What are the associated and common concepts?

R2 What industry standards exists for sensor metadata, e.g. descriptions of the sensors, data syntax, and semantics?

R3 What kind of APIs and protocol bindings are required and relevant in order to produce data on the three cloud platforms?

To achieve any substantial results, we undertake surveys on the cloud platforms and existing research literature, related to the research questions above. Finding efficient solutions for sensor data integration depends on many different factors. We have examined and compared the different candidate technologies, and developed prototypes for testing the cloud platforms' sensor data integration capabilities, with the goal to identify the most suitable platform and technologies. The following are contributions of this thesis:

- Development of sensor data integration prototypes for each of the three cloud platforms.
- Present the applicable technologies and approaches for sensor data integration based on the findings.

- Identification of limitations and shortcomings based on the technologies' capabilities.

1.3 Methodology

To answer the research questions, we need a research method to evaluate the technologies and approaches. This thesis has used Brown and Wallnau's technology delta evaluation framework [30] as a foundation. The framework consists of three phases, a descriptive modeling phase, an experiment design phase, and lastly an experiment evaluation phase. Figure 1.1 shows a model of the framework.

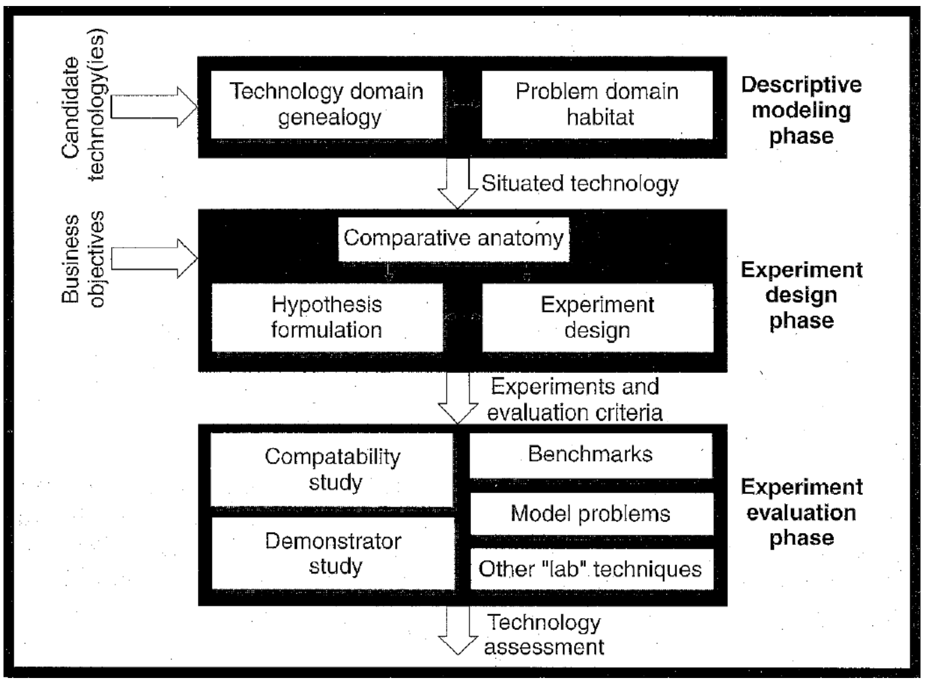


Figure 1.1: Technology Delta Evaluation Framework [30]

The descriptive modelling phase consists of discovering the impact and distinctive features of candidate technologies, while also understanding their historical and technological antecedents. We create requirements based on the technology features of interest and their relationship to usage contexts. These requirements form the basis for the experimental evaluation phase. For this thesis, the technologies we consider are metadata industry standards, IoT technology and cloud platforms.

The experiment design phase works as the planning phase. Here, we conduct an empirical comparative feature analysis for a more detailed investigation of the technologies and platforms. Our research questions are then investigated for gathering refutable evidence. In our case, we have compared the features, advantages, and disadvantages of the candidate technologies, and designed how the prototypes will be implemented for checking against the core requirements.

This is done in order to generate sufficient evidence for answering our research questions.

Finally, in the experimental evaluation phase we conduct experiments, gather evidence, and analyze it. From here, the research questions are answered. This means evaluating the sensor data integration capabilities of each cloud platform based on the prototypes' applicability and our theoretical findings. The information gathered are then used for evaluating each platform separately, according to a defined set of evaluation criteria. The thesis has decided on the following criteria: Interoperability, IoT support, device connectivity, data management, and ease of implementation.

1.4 Outline

The rest of the thesis outline are as follows:

Chapter 2 - Background: Covers theoretical background for concepts and literature relevant to sensor data integration in the three chosen cloud platforms.

Chapter 3 - A Survey on Application Layer Protocols and Metadata Industry Standards: Presents and compares relevant application layer protocols and metadata industry standards.

Chapter 4 - Cloud Platforms: Presents a detailed overview of the three chosen cloud platforms, and compares their features and services.

Chapter 5 - Design and Implementation: Describes the design and implementation of the prototypes developed for sensor data integration in the three cloud platforms, using Smart Ocean's pilot demonstrator 1 (PD1) as a case study.

Chapter 6 - Evaluation: Here, we evaluate the three cloud platforms based on the thesis' defined evaluation criteria.

Chapter 7 - Conclusion and Future Work: In the final chapter, we discuss our findings and results, and answer our research questions. Then we give a final conclusion and cover future work.

Chapter 2

Background

This chapter provides theoretical background for paradigms and technologies relevant for sensor data integration in the three cloud platforms that are under evaluation for the Smart Ocean project.

2.1 Internet of Things

The European Research Cluster on the Internet of Things (IERC) [248] defines IoT as "a dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual "things" have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network."

In short, the IoT is comprised of connecting "things" from the physical world to the Internet, bridging the gap between the physical and virtual worlds. The "things" in the IoT spans many different types of physical devices, appliances and machines that have capabilities to sense, actuate and monitor their different surroundings [173]. There are also things with edge computing capabilities, often referred to as edge nodes. Edge nodes are sensors and smart things that have the ability to perform local preprocessing on the collected data in order to improve data transmission over the network and offload some of the processing pressure in IoT system [267, 107].

The IoT builds upon the concept of machine-to-machine (M2M) communication which allows for information exchanges between two or more machines with minimal to no human intervention. While M2M relies on point-to-point connectivity and is generally designed for isolated network environments, IoT greatly expands on M2M's capabilities in terms of scalability, networking and interoperability. The IoT can interconnect with billions of devices over the Internet and the various types of data collected can be exchanged with other devices and applications, or be transferred to centralized servers or cloud-based applications for processing, storage and visualization [21]. In this section we describe the IoT's essential elements, general architecture, and its novel class, the Internet of Underwater Things (IoUT).

2.1.1 IoT Elements

IoT technology comes with many functionalities and in order to use them properly, six essential elements are required. These are identification, sensing, communication, computation, services and semantics [31, 11, 108, 68]. We describe these elements below.

Identification
With the IoT, it is important to ensure correct identification for each physical thing in the system. There mainly exists two identification processes: naming and addressing. Naming refers to the thing's specific name, and addressing refers to its unique IP address that are used for network communication. Several things can share the same name, but the addressing needs to be unique for each of them. Technologies such as electron products codes (EPC) and ubiquitous codes (uCode) [105] can be used for naming the devices. The methods for addressing typically involves using common internet protocols such as IPv4 [92], IPv6 [89] and 6LoWPAN [177] for assigning the unique IP addresses to each thing.

Sensing
Sensing in the IoT involves using physical things that have the capabilities to collect information from their surroundings and transferring it to corresponding endpoints where the information can be stored. The exists many different types of Internet-enabled things that provide sensing capabilities for collecting information.

Communication
One of the main purposes of IoT is enabling physical things to connect and communicate over the Internet. With communication, the things can transfer and receive various types of information. There exists many technologies that provides communication such as Bluetooth [34], WiFi [83], Ethernet [84], and cellular communications like 2G, 3G, and 4G [176, 93].

Computation
When the collected information has been transferred and received by the consuming endpoints, various hardware, software and cloud platform components can be utilized to perform computation on the information. For hardware, integrated circuit boards such as Arduino and Raspberry Pi can be used, whereas operating systems plays a major role for processing in software applications. Cloud platforms are also an important computational aspect of IoT as they provide high computation capabilities, including real-time processing.

Services
There mainly exists four types of services that can be provided by IoT systems. The first service is the Identity-Related Service, which is used for identification of the physical things that sends requests. The second service is the Information Aggregation Service that collects and processes all the information. The

third service is the Collaborative-Aware Service which makes decisions according to the collected information and sends back responses to the things. The last service is the Ubiquitous Service, which aims to deliver communication to the things, independent of time and place [70, 266].

Semantics
Semantics are contextual information that are used for discovering each physical thing in the system and extraction of their resources and modelling information. An IoT system must be able to interpret and accept all the receiving information and make appropriate decisions. They achieve this by using existing standards or specified descriptions, ontologies and schemas that ensures interoperability in the IoT system. This will be discussed in more detail in chapter 2.4 and chapter 3.2.

2.1.2 General Layered IoT Architecture

In this section we provide an overview of a general layered architecture for the IoT. There have been several architectures for the IoT that have been proposed over the years by different researchers, but due to the wide range of different IoT areas and scenarios, there exists no single architecture that can address all the requirements that each IoT area brings [67]. In [12], sixteen different IoT architectures were identified, and the studies done in [106] and [189] shows that most layered IoT architectures ranges between three to seven layers, with different levels of complexity. Many IoT companies are also aiming towards standardizing the complex IoT areas by releasing IoT reference architectures in order to provide strong starting points for developers that wants to build IoT solutions [189]. Reference architectures are recommended architectures that are derived from other more specific and concrete architectures [22].

For this overview, we focus on a well-known five-layer architecture [265, 196, 104, 245] that represents most of the common building blocks of an IoT system. In the early research stages, a basic three-layer architecture was introduced and consisted of these layers: The perception layer, network layer, and application layer. The five-layer architecture builds upon the three-layer variant and adds two more layers: The middleware layer and business layer, which better represent the current state of the IoT paradigm. Figure 2.1 shows the layered structure.

- **Perception Layer:** This is the physical layer which consists of all the "things" in the IoT system. The IoT devices senses and collects different physical parameters around their respectable environments. The main task of the perception layer is to perceive the collected information and convert it to digital signals which is then passed over to the network layer for transmission.
- **Network Layer:** This layer is responsible for securely and reliably transmitting the information gathered from the perception layer to the middleware layer for processing, either directly or via gateways. Many different networking protocols can be used, depending on whether the connection is wired or wireless.

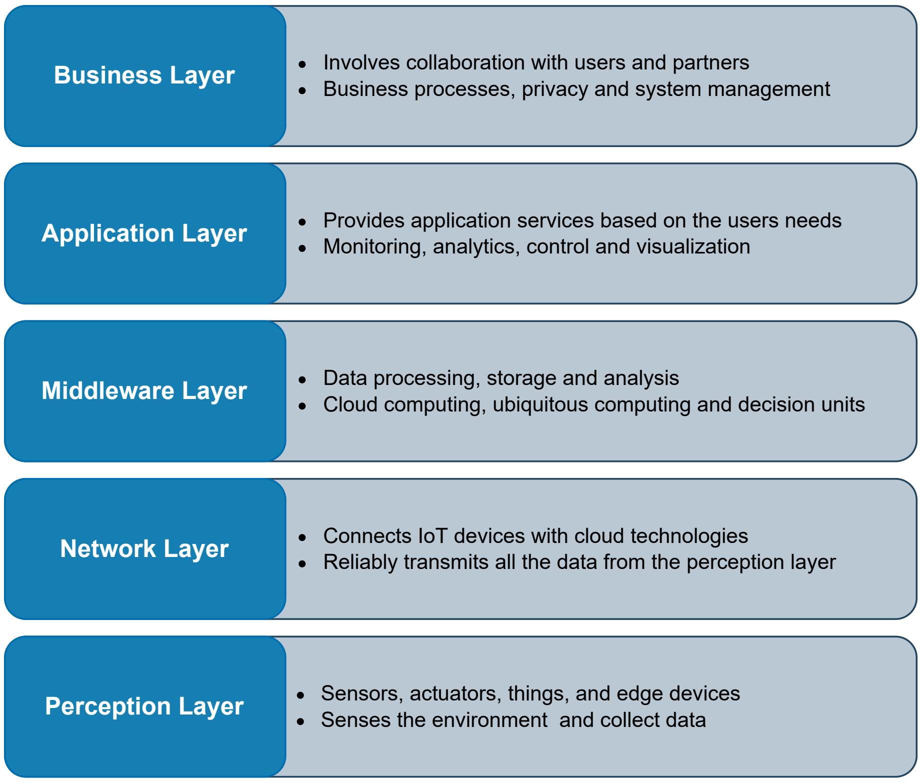

Business Layer
Involves collaboration with users and partners
Business processes, privacy and system management
Application Layer
Provides application services based on the users needs
Monitoring, analytics, control and visualization
Middleware Layer
Data processing, storage and analysis
Cloud computing, ubiquitous computing and decision units
Network Layer
Connects IoT devices with cloud technologies
Reliably transmits all the data from the perception layer
Perception Layer
Sensors, actuators, things, and edge devices
Senses the environment and collect data

Figure 2.1: The Five-Layer IoT Architecture

- **Middleware Layer:** Also known as the processing layer, the middleware layer connects all the components and data streams by linking to the devices or gateways in the network layer, and on the other side integrate with applications and services in the application layer. Its main tasks is to store, analyze, and process all the data that comes from the network layer. Due to the large volumes of data that often comes through, the middleware layer uses different technologies such as databases, Cloud Computing, Ubiquitous Computing, and big data processing for managing all the data.
- **Application Layer:** This layer is responsible for delivering applications and services to end users based on their needs. This can be anything from device monitoring and management, to data visualization, or report creation based on the processed information.
- **Business Layer:** The business layer is responsible for managing the overall IoT system and its applications and services that are communicating with each other. The data received from the application layer can be utilized for creating different business processes and models, in order to determine future actions, innovations and business strategies. This is when the owners and collaborative partners receives the value from the IoT.

2.1.3 Internet of Underwater Things

The Internet of Underwater Things (IoUT) [190] is a novel class of the IoT for underwater environments and will be the foundation for collecting data in the SFI Smart Ocean project. The functionalities and structure of the IoUT is similar to land-based IoT, but due to the unique characteristics and challenges of underwater environments, general design approaches for land-based IoT cannot directly be adopted. Underwater environments affects the sensors network coverage and data transmission capabilities. An IoUT system typically consists of underwater sensors, sinks that collects data from the sensors and forwards it to the surface, and different stations that receives the data. Figure 2.2 shows a general model of an IoUT system.

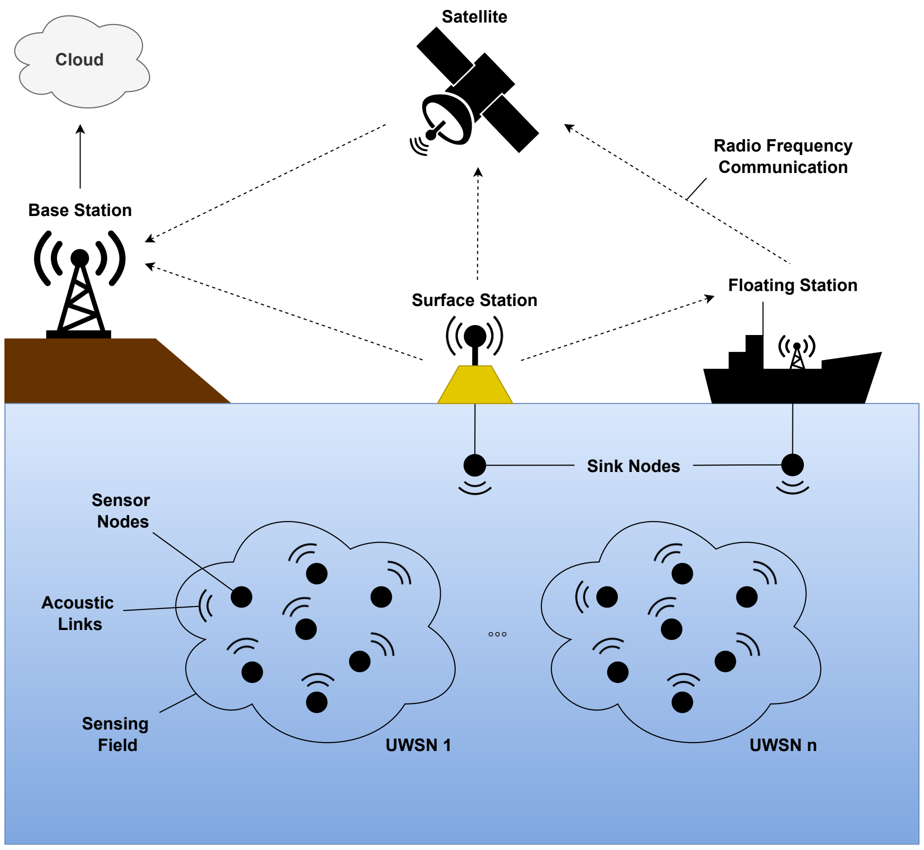


Figure 2.2: General Model of IoUT [111, 101]

Data collection in underwater environments nowadays is primarily done through monitoring from ships, buoys, and autonomous vehicles [183]. Underwater wireless sensor networks (UWSNs) are the main type of sensors that will be utilized for collecting underwater data in the Smart Ocean project. A general wireless sensor network (WSN) can be described as a network that contains distributed heterogeneous sensors, called nodes. These nodes cooperatively communicate through wireless links and collects information from their sensing field areas. The collected data is then sent to sinks that can either preprocess some of the data locally, or forward it via connected networks or gateways [7, 10]. In UWSNs, each sensor node are distributed across the underwater environments and use acoustic communication in order to sense, collect and transfer underwater information. Unfortunately, acoustic communication has high latency and low bandwidth [201]. It is especially important that these underwater sensor nodes avoid large headers in order to efficiently transfer the data. The collected data gets transmitted to near sinks connected to certain stations, which then either preprocesses or directly forwards the data to the nearest onshore base station where the data gets analysed, or sent to the cloud via the Internet for storage and processing purposes [82].

2.2 Cloud Computing

The U.S. National Institute of Standards and Technology (NIST) [250] defines Cloud Computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Cloud Computing provides several applications and services that offers computation, communication, and storage resources at low cost. This is a main reason why it has become a promising technology for supporting IoT systems [260]. The future Smart Ocean Platform will include a set of Cloud Computing resources in order to provide a secure and operational infrastructure that can manage, store and process all the incoming sensor data. In this section we present relevant service models, IoT cloud platforms, and a general overview of the sensor-cloud infrastructure.

2.2.1 Service Models

With Cloud Computing, there are several ways cloud providers can deliver services to end users. NIST defines three main ways: Infrastructure-as-a-Service, Platform-as-a-Service, and Software-as-a-Service [250].

Infrastructure-as-a-Service (IaaS)
IaaS is a Cloud Computing service model that only offers the most fundamental computing resources to the users, such as virtual machines, servers, storage, networking and hosting environments. The cloud provider controls the underlying cloud infrastructure, while the users are responsible for deploying the operating system and the applications that are going to be executed on the cloud service. The users are also responsible for development and updating the applications, as well as monitoring their operational statuses. IaaS offers the most freedom and control to the users, at the expense of less abstraction and more responsibility [20, 250, 57].

Platform-as-a-Service (PaaS)
PaaS is a higher level of abstraction then IaaS where the cloud provider manages everything from servers, networks, storage, and operating system. It controls

the underlying infrastructure as well as providing updates for the runtime environments and supporting services. This makes it simpler for the users, but they are still responsible for the development, configuration and management of the deployed applications on the cloud service. Nowadays, PaaS is often built around containers that virtualize the operating system, which only contains the necessary executables and configurations needed for running the applications [20, 250, 57].

Software-as-a-Service (SaaS)
With SaaS, entire applications run on the provider's cloud service. Users can access these applications by using a web browser, desktop client, or APIs that integrates with their applications. Since the cloud service provider manages both the application and data, users typically have very limited control over the application, with the exception of user-specific settings [20, 250, 57].

2.2.2 IoT Cloud Platform Components

The cloud itself is a collection of servers and databases that are located in data centers. A cloud platform is a comprehensive suite of integrated cloud services that can facilitate many of the common features needed for building diverse IT solutions. In context of the IoT, data collected from heterogeneous devices can be sent via the Internet and allocated to several cloud platform services to perform operations on them. There exists many different variants of cloud platforms provided by many different vendors, ranging from open-source communities to large commercial vendors such as Microsoft, Google, and Amazon, which currently holds the largest cloud market shares [256]. The scope of cloud offerings varies for each individual cloud platform, but a cloud platform utilized for the IoT typically consists of the following components: cloud gateway, digital twins, stream processing, storage, security and management, and business integration [174, 198, 1]. Figure 2.3 shows a high level architecture on how these components work together.

- **Cloud gateway:** A cloud gateway works as a control point that interface with connected IoT devices. Its main tasks is to acquire and route their incoming data to specified destinations and services in the cloud, using appropriate data ingestion techniques. The cloud platform's active rules and policies determine how individual data is routed.
- **Digital twins:** A digital twin is a synchronized, digital representation replica that simulate the state and behavior of a physical IoT device. These digital representations are stored in the cloud and are used to simulate and predict the behavior of their physical counterparts. This reduces the need to always rely on connecting to the physical devices directly, since the digital twins are always available in cloud.
- **Storage:** Storage services can store data for long-time archival purposes, or perform demanding analytics and machine learning processing on them. The databases are quite scalable in size and typically provide both relational SQL databases for structured data, and NoSQL databases for unstructured data. Supported queries and APIs are used to retrieve stored

data, which can then be to sent other connected cloud services and applications.

- **Stream Processing:** Stream processing services operate on data flows that are being routed with relatively low latency. These services can process incoming data in near real-time, or implement temporal storage for the most recent data sets and perform moderate analytics on them. The stream processes can involve multiple input streams and output streams. The output stream data can be forwarded to other cloud services and applications, using supported queries and APIs.
- **Security and management:** Cloud platforms need to arrange for secure device communication over the network, as well as security and management in the cloud environment. The platforms can provide identity and access management (IAM), rules and policy creation, secure monitoring, data protection and transport encryption. With device provisioning, each device gets an unique ID in combination with a certificate and public/private keys embedded in them for gateway authentication. The cloud's identity registry contains all the registered devices where they can be managed by the users.
- **Business integration:** The business oriented cloud services are the ones that produce value to the consumers, while most of the other components are mainly used for gathering insights about data. Business integration involve enterprise services that provide functionalities such as visualization, notification, and system management.

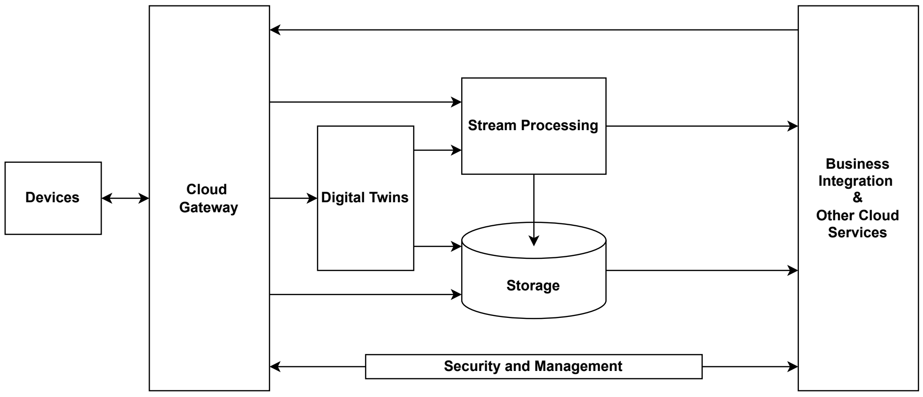


Figure 2.3: High Level Architecture of IoT Cloud Components [174, 1]

2.2.3 Sensor-Cloud Infrastructure

As mentioned earlier, the SFI Smart Ocean wants to realize the IoUT in their project, where UWSNs are utilized as the main IoT device type in the system. By integrating WSNs (or UWSNs) with Cloud Computing, we get what is commonly referred to as a sensor-cloud infrastructure [269]. While the sensor-cloud infrastructure is conceptually similar to many other IoT-cloud integration approaches, sensor-cloud emphasizes an approach for managing, organizing and monitoring multiple WSNs at the same time on a shared network.

The infrastructure consists of using Cloud Computing's resource offerings to create virtual representations of all the physical sensors. By creating virtual sensors, users do not need to worry about the physical sensors real locations, nor their different specifications. For standardization, the users creates virtual sensor templates for the different physical sensors that contains their data. While there can be many different types of physical sensors, many of them may also share several constraints. In these cases, users can also create templates for specified virtual sensor groups to add more flexibility. The templates are stored and processed in the cloud and can be dynamically and automatically provisioned when needed. Figure 2.4 shows the relationship between the physical sensors, virtual sensors and virtual sensor groups.

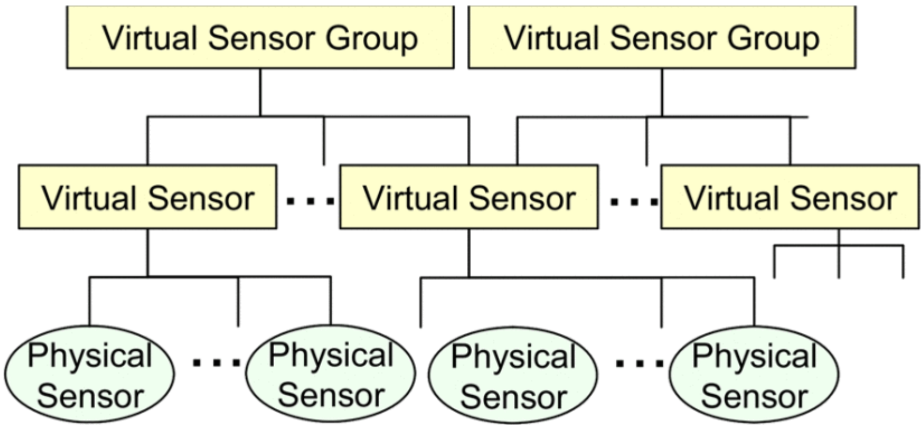


Figure 2.4: Relationship Between the Physical Sensors, Virtual Sensors and Virtual Sensor Groups [269]

When it comes to architecture and workflow, there exists several architectures [8], but they all mostly follow the same main principles. Figure 2.5 provides a general architecture that highlights how a sensor-cloud infrastructure can be utilized. The developers creates a solution framework of cloud services that abstracts much of the system environment's complexity. This include many of the cloud components that were presented in section 2.2.2. The clients access the sensor-cloud infrastructure's portal with different user-roles. The roles can be administrators who manages and monitors the infrastructure's services, sensor owners that can register, delete or monitor their sensors, or end users who requests and utilize various sensor data from the virtual sensors. Once they have entered the infrastructure's user interface, there are several available options the clients can choose to perform. Virtual sensor templates and their resources can be requested, registered or reserved via automatic provisioning. Sensor information can be retrieved for monitoring and visualization purposes, and the sensors virtual machines can be accessed for control. The sensor-cloud infrastructure provides virtualization, standardization, automation and monitoring of multiple WSNs in an interoperable manner.

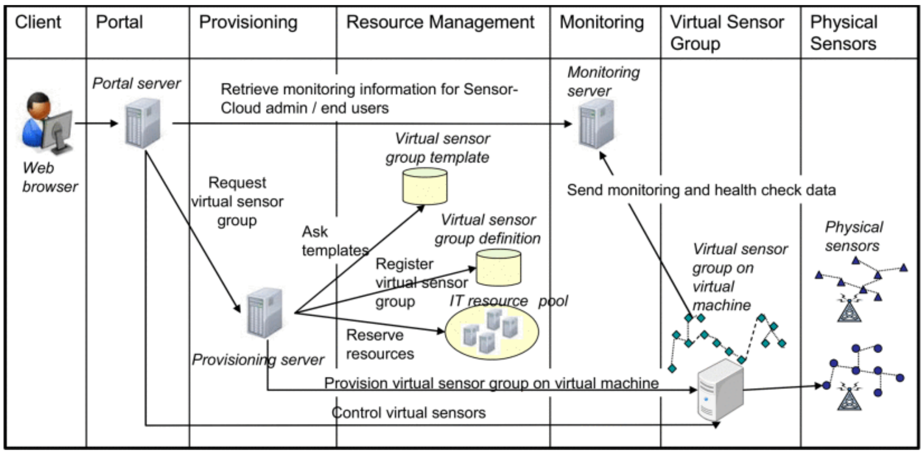

Client
Portal
Provisioning
Resource Management
Monitoring
Virtual Sensor Group
Physical Sensors
Web browser
Portal server
Retrieve monitoring information for Sensor-Cloud admin / end users
Monitoring server
Virtual sensor group template
Request virtual sensor group
Send monitoring and health check data
Ask templates
Virtual sensor group definition
Physical sensors
Register virtual sensor group
Virtual sensor group on virtual machine
IT resource pool
Reserve resources
Provisioning server
Provision virtual sensor group on virtual machine
Control virtual sensors

Figure 2.5: Sensor-cloud Architecture [269]

2.3 Application Layer for IoT

The application layer is the layer that lies on the very top of the Open Systems Interconnection (OSI) model and the simpler Transfer Control Protocol/Internet Protocol (TCP/IP) model [192, 9]. The application layer contains many different standards and protocols for handling the communication and message exchanges between the application entities in IoT systems [191]. In this section, we describe some of the application layer's most relevant standards for sensor data integration in the selected cloud platforms. The only exception is the application layer protocols which will get its own section in chapter 3.1.

2.3.1 Data Serialization Formats

In IoT systems, data serialization formats are used for encoding collected data into messages that can be exchanged and interpreted by receiving endpoints for storage and sharing purposes. These data serialization formats allows the information to be maintained and recovered in its original structure. Each format have a different effect on the device resource usage, processing, and communication efficiency. As a result, it is important to choose data formats that suits each IoT environment [191, 175]. This section presents some of the most widely used formats for the IoT.

Extensible Markup Language (XML)
XML [264, 28] is a commonly used text-based format for representing and sharing structured information. It is stored in a hierarchical format and its syntax rules are strict, meaning that XML documents will not get processed in case of errors. In XML, each information element is represented in a name-value pair (also called key-value pair) with the value contained between start-tags: <> and end-tags: </>. Here is an example: <name>value</name>. The name is the defined attribute or property name specified in the document, while the value represents its current content or state. There also exists a more compact representation of the XML format, called Efficient XML Interchange (EXI) [199].

The EXI format is more intended for resource-constrained environments as it converts XML messages to binary, optimizing the utilization and performance of the computational resources.

JavaScript Object Notation (JSON)
JSON [257, 55] is another widely-used text-based data serialization format. The data is stored in a hierarchical format and uses a syntax that consists of a small set of formatting rules with braces, brackets, colons, and commas. The syntax does not specify any complete data interchanges, but provides the framework needed for attaching the desired semantics. Just like XML, each information element is represented in a name-value pair, only that it is structured differently in JSON. Here is an example: {"name" : value}. The value can be either an object, array, integer, string, true, false, or null. There also exists another variant of the JSON format called JSON-LD [263]. JSON-LD is a JSON format for serializing Linked Data, which is a way to create interpretable data with strong relationships across the web.

Comma-Separated Values (CSV)
CSV is yet another text-based data serialization format. It is a format that is often used for storing and exchanging large amounts of tabulated data in various cloud platforms. As the format-name suggests, the values are separated by commas and are delimited by basic line breaks (CRLF). The name-value pairs can for example be described as: 'name1,name2,name3 CRLF' which contains: 'value1,value2,value3 CRLF'. CSV are supported by a wide range of database and spreadsheet programs, which makes it easy to import CSV files and work with the data, as well as exporting the files into other more efficient formats for processing. The CSV format is considered to be semi-structured as it cannot represent hierarchical data naturally. The relations between the data is usually handled by multiple CSV files that are linked through the use of column-contained foreign keys [170, 246].

2.3.2 Communication Models

When application layer entities communicate and exchange messages over a network, they are usually following the rules of a specified communication model. In this this section we present the two most relevant communication models for communication in the IoT.

Request-Response Model
Request-Response is a communication model that consists of a client and a server. The client sends requests to the server, while the server responds to each request. Based on the request, the server's response can provide the client with specified resources, or perform more internal functions within the system. Each pair of request-response messages is independent and the communication model works in a stateless manner, meaning that the server does not keep any session information. The client's request must contain all the necessary information that the server needs in order to process and respond to the request [21]. Figure 2.6 shows the request-response communication model.

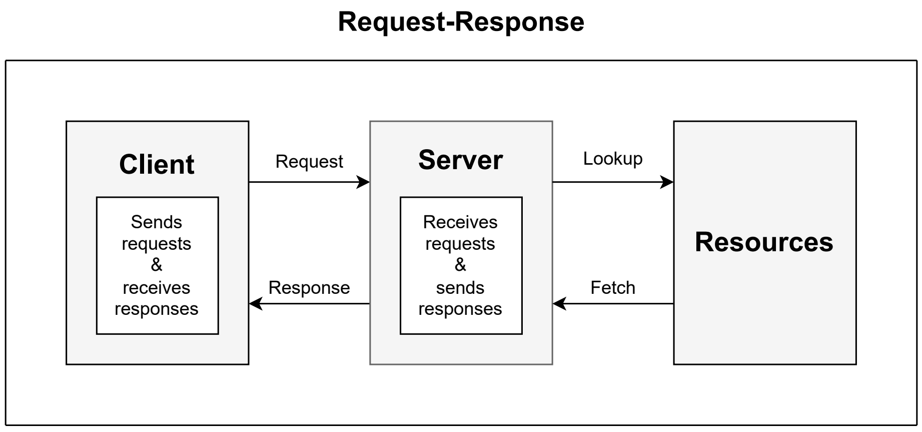

Request-Response
Client
Sends requests & receives responses
Request
Response
Server
Receives requests & sends responses
Lookup
Fetch
Resources

Figure 2.6: Request-Response Communication Model [21]

Publish-Subscribe Model
Publish-Subscribe is another communication model that consists of publishers, subscribers, and a broker. Here the publishers and subscribers are the clients, while the broker act as the server. The broker contains different topics that clients can either publish or subscribe to. Publishers sends messages to the broker which then forwards the messages to specific topics on the server. Other clients can subscribe to these topics and then only receive messages from these topics explicitly. Subscribers receives the messages the moment they are published on the server. Publishers and subscribers are also not aware of each other [21]. Figure 2.7 shows the publish-subscribe communication model.

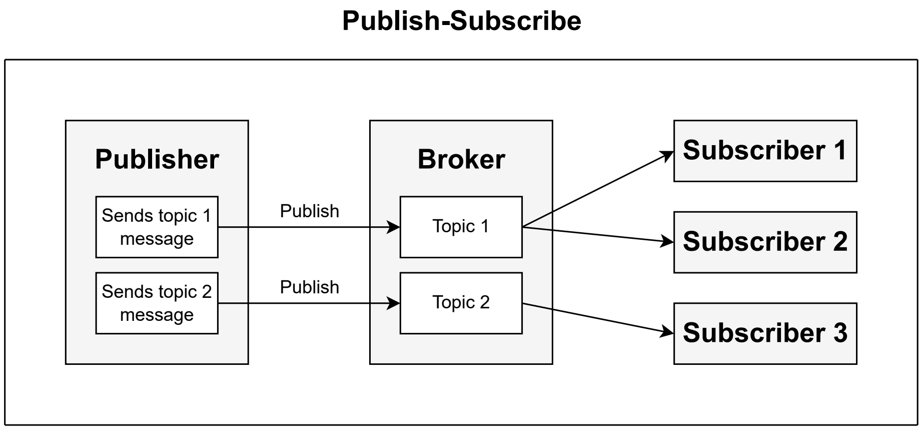


Figure 2.7: Publish-Subscribe Communication Model [21]

2.3.3 Representational State Transfer (REST)

An application programming interface (API) is a set of rules that define how applications or services can connect and communicate with each other through a documented interface. With the use of APIs, various data can be queried, parsed and exchanged in a simplified and secure manner [56]. REST is an architectural style for creating web services and web APIs, and is the primary design model for API utilization in IoT and Cloud Computing. The REST architecture was first introduced by Roy Fielding [61] in the year 2000. A REST API follows the request-response model, and interacts between a client and a server based on the resources available. The resources are addressed using Uniform Resource Identifiers (URIs), and are typically represented in text-based formats such as XML or JSON. REST APIs communicate using HTTP based commands such as POST, GET, PUT, and DELETE in order to perform the standardized CREATE, READ, UPDATE, and DELETE (CRUD) operations in general databases. The architecture also comes with six architectural constraints that needs to be followed in order for it to be declared as a RESTful API [58, 21].

- **Client-Server:** The principle behind this constraint is to ensure that the client and server are fully decoupled and independent of each other. The client should not know anything about the server and its content, and only be concerned with the information contained in the URI of each request. Similarly, the server should not know anything about the client besides the request and only be concerned with passing the requested information in the response.
- **Stateless:** By being stateless, each request needs to include all the information needed in order for it to be processed on the server-side. The server is not allowed to store any session data related to a client's request.
- **Cacheable:** The constraint requires the responses to label themselves as either cahchable or non-cachable, when possible. If a response is cachable, then the client is allowed to keep the response data for reusing purposes. The goal is to eliminate unnecessary interactions between the client and the server, increasing performance and scalability.
- **Layered System:** Between the client and server are different layers that the requests and responses goes through. The layers should be designed with separated concerns that do not effect the requests and responses in any way. The client and the server should also be agnostic to whether they communicate with the endpoint or an immediate layer.
- **Uniform Interface:** This constraint evolves around making sure that the communication method between the client and server is uniform. Requests that asks for the same resources should be identical, and each resource representation should only contain information that the client might need.
- **Code on Demand:** This is the only optional constraint. While the contents of the resources are usually static, sometimes the server's responses can contain scripts or executable code to run on the client-side. In these scenarios, the code should only be executed on-demand.

2.4 Interoperability

One of the main goals of the IoT is to achieve interoperability in their systems. However, achieving interoperability is also one of the IoT's biggest design challenges. The reason for this is that there exist no universal standard for representing data in the IoT. An IoT system can span many types of devices with different data syntax, descriptions and semantics which often leads to interoperability issues. There exists many definitions on interoperability, but most of them agree on same principles, which is being able to utilize, understand, and exchange information between system endpoints [261, 53]. This section provide an overview of relevant interoperability levels, metadata, and IoT frameworks.

2.4.1 Interoperability Levels

There are mainly three interoperability levels of relevance for IoT systems: technical interoperability, syntactical interoperability, and semantic interoperability [18, 261, 72]. We describe each of these three levels below.

Technical Interoperability
Technical interoperability is strongly related to hardware, software and platform components with machine-to-machine (M2M) communication capabilities. It is the interoperability level that enables heterogeneous IoT devices to communicate and exchange messages over a network, without necessarily understanding the meaning or content of the messages. This interoperability level is typically achieved be utilizing standardized communication protocols such as HTTP, MQTT, AMQP and CoAp (see chapter 3.1).

Syntactic Interoperability
Syntactic interoperability is related to how data is represented and how it can be retrieved in IoT systems. Data representation is usually realized through the use of standardized data formats such as JSON, XML and CSV, while standardized query languages can be utilized by users for retrieving the desired data.

Semantic Interoperability
Semantic interoperability ensures a common understanding of the information exchanged between IoT devices and system endpoints. The interoperability level facilitates machine understanding of data by providing contextual information for semantic data representations in the system. Consuming endpoints are then able to understand the data through the adherence to common specifications.

2.4.2 Metadata

For the Smart Ocean project, it is necessary to define universal metadata representations for the various underwater sensors in order to ensure interoperability. Metadata can in its simplest definition be described as "data about data". It can provide specified information about the content, context and structure of different resources, which adds value by making it easier to locate, retrieve, organize, analyze and manage each resource [195, 85].

Descriptions
Metadata descriptions defines specified content-information that are critical for establishing an accurate understanding of a resource. The main purpose of these descriptions is to facilitate discovery and identification of each resource. In context of the IoT, important resource descriptions can be information about telemetry data, events, states, locations, settings, and serial numbers, which provides the necessary information for distinguishing each resource in the system.

Semantics
Semantics describes the "meaning" of data, which expresses the representation of metadata in a contextual manner. They provide annotations for representing a resource's properties and capabilities, using a well-defined set of vocabulary terms. Semantics typically involves ontologies which defines a set of formal descriptions and relationships that are used to describe a specific domain, abstracting the heterogeneity of systems.

Data syntax
Data syntax specifies the rules of how metadata information is structured which allows it to be stored, transferred, and queried. The syntax declares the grammar, data relationships, and hierarchical design of metadata. The metadata information can be expressed in many different markup languages, programming languages, and serialization formats.

2.4.3 IoT Frameworks

Another way of dealing with interoperability is developing or utilizing existing IoT frameworks. An IoT framework provides an information model and other specifications in order to facilitate interoperability in IoT systems. These frameworks are typically designed for interfacing between the application layer and transport layer in order to abstract much of the system complexity. With this approach, applications and platforms are able to interact with the framework's data directly. IoT frameworks typically support several standardized communication technologies and message exchange techniques that all the IoT devices must follow, which simplifies how devices interconnect with the IoT system. With the following specifications, IoT frameworks have the potential to ensure the syntactic and semantic levels of interoperability. [32, 175]

- **Information model:** Are abstract data model representations that describe the information of the IoT devices. The information model usually includes information like data types, attributes, properties, relations, schema definitions and metadata descriptions that facilitates universal data representations in the IoT system.
- **Payload serialization:** The payload data needs to be serialized into a format that receiving endpoints are able to interpret. IoT frameworks often support a common set of data serialization formats. Examples are JSON and XML which uses a high-level syntax that represents the data elements in name-value pairs.
- **Protocol bindings:** Specifies the supported application layer protocols

that the IoT devices communicate over. The bindings may also include the operated port numbers that each protocol uses. Examples are HTTP, MQTT, AMQP and CoAp (see chapter 3.1).

- **Identification and discovery:** IoT frameworks needs to offer conventions for giving unique identifications for each IoT device, such as naming and addressing methods in order for them to be discovered in the system.
- **Security:** An IoT framework usually contains a specified security perimeter that all IoT devices must follow in order to successfully connect and communicate in the system. Depending on framework, the integrated security may involve several authorization, authentication and/or encryption capabilities.

Chapter 3

A Survey on Application Layer Protocols and Metadata Standards

Besides evaluating the selected cloud platforms, another goal is to investigate relevant protocol bindings and metadata industry standards that can potentially be used in the Smart Ocean project. This chapter presents a survey on relevant application layer protocols that can be used for sending sensor messages to the three cloud platforms, as well as metadata industry standards that can be used for creating interoperable sensor data representations.

3.1 Application Layer Protocols

The application layer protocols define how applications send messages over the network. The messages are encoded in the application layer and encapsulated in the transport layer for transportation, while port numbers are used for addressing [21]. In this section we describe some of the most common and relevant protocols that are utilized in the IoT.

3.1.1 HyperText Transfer Protocol (HTTP)

HTTP is the universal protocol utilized for the World Wide Web (WWW). It is based around the request-response model where the client uses different HTTP commands to send requests to a server. These commands are GET, POST, PUT, DELETE, HEAD, TRACE, OPTIONS, and CONNECT. For security, HTTP uses either the Transport Layer Security (TLS) protocol or the Secure Socket Layer (SSL) protocol, resulting in the secure version of HTTP, known as HTTPS. Universal Resource Identifiers (URIs) are used to identify HTTP's resources. Since REST API's uses HTTP commands for transferring data, the RESTful architecture can easily be implemented into IoT systems. For transportation, HTTP uses the Transmission Control Protocol (TCP) which guarantees reliable and successful delivery as long as the connection is not interrupted.

Unfortunately, HTTP is known for having large header sizes due to its textual format. In combination with the required TCP packets that comes with each request, makes HTTP a very heavy protocol. As many IoT devices often have limited bandwidth and battery-power, this makes HTTP not a suitable protocol for resource constrained environments. HTTP/1.1 [60] is currently the most widely adopted version of the protocol, but there exist a newer version named HTTP/2 [24]. This version is showing promising improvements over HTTP/1.1 with lower latency, full request-response multiplexing and compression of the header fields that reduces the header sizes. However, all the three cloud platforms evaluated for this thesis, still only supports the HTTP/1.1 version for sending IoT data to their cloud gateways [21, 54, 59, 23].

3.1.2 Message Queuing Telemetry Transport (MQTT)

MQTT is a machine-to-machine (M2M) message protocol based around the publish-subscribe model. Here, the clients are either publishers or subscribers, while the server is a message broker that coordinates all the messages for each topic. Just like HTTP, it runs over TCP and uses TSL/SSL for security. Commands such as CONNECT, DISCONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE, and CLOSE are utilized with this protocol. MQTT provides three levels of quality of service (QoS): level 0-2, which makes its message delivery quite reliable. These QoS levels are (0) at most once which is suitable if data loss is acceptable; (1) at least once which ensures delivery, but it may incur duplicates; and (2) exactly once that guaranties message delivery without duplicates. Which level to use depends on the requirements of each IoT system. The MQTT protocol is flexible and lightweight, and suits many diverse application scenarios. It is primarily used for devices that require efficient use of bandwidth and battery-power, making it an ideal protocol for IoT. Currently, the most widely adopted version is MQTT 3.1.1 [179], while MQTT 5.0 [180] is the newest version that comes with several new features and benefits over version 3.1.1. Unfortunately, the three evaluated cloud platforms are still primarily using version 3.1.1, with little to no support for version 5.0. The cloud platforms also only provide support for QoS level 0 and 1, and not level 2 [21, 268, 197, 54].

3.1.3 Advanced Message Queuing Protocol (AMQP)

AMQP is a communication protocol that is primarily designed for exchanging business-oriented messages between system endpoints. AMQP runs over TCP and supports the publish-subscribe model. The publish-subscribe approach is similar to MQTT, but the key difference is that the broker is divided into two main components: exchanges and queues. The exchange component receives publisher-messages which then distribute copies of the messages to queues for each topic. Subscribers connect to these queues and then receives the messages whenever they are available. Just like MQTT, it also supports the same three levels of QoS and uses TLS/SSL for security. It also supports the Simple Authentication and Security Layer (SASL) framework that provides additional authentication mechanisms and data security. The AMQP protocol provides a higher level of security and more messaging queuing capabilities then MQTT, but the protocol is also more complex and were designed for more general purpose usage rather then for IoT specifically. The protocol is commonly implemented for high performance messaging in server-based enterprise environments [181, 21, 197, 54].

3.1.4 Constrained Application Protocol (CoAp)

CoAp is a web transfer protocol optimized for machine-to-machine (M2M) applications. It has many of the same characteristics as HTTP, but is designed for resource-constrained environments. Like HTTP, CoAp follows the request-response model and is based on the REST model, using commands such as GET, PUT, POST, and DELETE. However, unlike HTTP, the protocol uses the User Datagram Protocol (UDP) instead of TCP for transportation which leads to more efficient transmissions, but also more unreliable messaging. CoAp defines a message layer that consists of four messaging types for dealing with the UDP transmissions: Confirmable (CON), Non-confirmable (NON), Acknowledgement (ACK), and Reset (RST). A request that requires reliable messaging uses a CON messages which contains a default timer and retransmits the request until the server responds with an ACK message, acknowledging successful delivery. In cases where the server is not able to process a CON message within the timeout, it responds with a RST message instead. For requests that does not require reliable messaging, a NON message is sent instead. These requests do not receive any response messages, but still offers detection for duplicates via its unique message ID. The CON and NON messages forms the QoS support for this protocol. For security, CoAp uses the Datagram Transport Layer Security (DTLS) protocol or IP Security (IPSec). With its low latency and efficient utility in resource-constrained environments, CoAp is a suitable protocol for IoT usage in scenarios that allows for unreliable messaging. [21, 247, 25, 197].

3.1.5 WebSocket

The WebSocket protocol supports full-duplex, bidirectional communication over a single socket. The main purpose of this protocol is to enable mechanisms for sending data through web applications, that needs two-way communication. The connection starts with the client initializing a handshake with the server to establish a session. As the handshake is similar to HTTP, the web servers are able to handle connections for both WebSocket and HTTP through the same port. However, the WebSocket protocol does not follow request-response model after the initial handshake, as the clients and servers exchange messages in an asynchronous full-duplex connection. The protocol message runs over TCP and provide secure sessions via TLS/SSL. The WebSocket protocol supports bidirectional communication, but also supports publish-subscribe messaging by using its sub-protocol called WebSocket Application Messaging Protocol (WAMP) [71]. While the protocol's use cases are limited to primarily web applications, it provides real-time communication, security, reliable transmission, and small header sizes. [21, 103, 110].

3.1.6 Protocol Summary

We have now described some of the most common application layer protocols that are used in IoT scenarios. Each protocol has its advantages and disadvantages when it comes to its use cases. In table 3.1, we give an overview of the features of each protocol.

Parameters	**HTTP**	**MQTT**	**AMQP**	**CoAp**	**WebSocket**
Transport	TCP	TCP	TCP	UDP	TCP
Header Size	Large, Undefined	2 Bytes	8 Bytes	4 Bytes	16 Bytes
Message Size	Large, Undefined	Up to max 256 MB	Undefined	Usually Small Enough to fit a Single IP Datagram	Undefined
Encoding	Textual	Binary	Binary	Binary	Textual/Binary
Communication Model	Request/ Response	Publish/ Subscribe	Publish/ Subscribe	Request/ Response	Bidirectional, Publish/ Subscribe
QoS	TCP Based	3 Levels	3 Levels	Confirmable or Non-confirmable	TCP Based
Security	TLS/SSL	TLS/SSL	TLS/SSL, IPSec, SASL	DTLS, IPSec	TLS/SSL
Standard	IETF [60]	OASIS [179, 180]	OASIS [181]	IETF [247]	IETF [110]

Table 3.1: Application Layer Protocol Overview

With HTTP, applications must repeat the headers with each request and response which increases latency. HTTP is a also a very heavy and resource-demanding protocol, which is not very ideal for efficiently transmitting IoT data between endpoints. Regardless, the HTTP protocol is still relevant due to the wide usage of REST API's in IoT systems, where HTTP commands are used. The CoAp protocol were explicitly designed for resource-constrained environments, making it an ideal protocol for the IoT. It offers small header sizes, asynchronous communication, and avoids unnecessary retransmissions unlike HTTP, which reduces latency and power consumption of IoT devices. However, the CoAp protocol's largest drawback is its messaging over UDP, making it not ideal for IoT scenarios where reliable messaging is a requirement. Both MQTT and AMQP uses publish-subscribe messaging with the same three QoS levels and provide small headers sizes, and reliable messaging via TCP. Both protocols are viable options for IoT, but MQTT was specifically designed for IoT, while AMQP were designed for more general purpose message queuing. AMQP provides more security then MQTT, but it is also a more complex protocol to implement. MQTT is a more simple and developer-friendly protocol, and offers slightly more efficient messaging then AMQP due to its smaller header size. Lastly, the WebSocket protocol does not have many utilization purposes in IoT beyond sending messages through web applications, where the protocol is mandatory. Other protocols such as MQTT and AMQP can also be binded with WebSocket and exchange messages, using their own rules on top of the WebSocket protocol.

3.2 Metadata Industry Standards

There exists several metadata industry standards that can provide sensor descriptions, data syntax, and semantics for the Smart Ocean project. In this section, we present some of the potential standards for usage in IoT scenarios.

3.2.1 IP for Smart Objects (IPSO)

The IPSO Working Group [252] is one of several collaborative working groups in the Open Mobile Alliance (OMA) SpecWorks [255]. The IPSO Working Group focuses on enabling communication between smart objects in IoT systems. It supports and manages a Smart Object Registry [253, 254] that consists of multiple defined smart objects. These smart object are meant to be used by anyone who require IoT-based metadata representations in their projects. The IPSO object model is designed to work with other RESTful enabled application layers and frameworks for enabling interoperable message exchanging between IoT devices and system endpoints. IPSO is also especially designed to work with the OMA Lightweight Machine-to-Machine (LWM2M) specification which provides security, management, interfaces and protocols such as CoAp or HTTP. Together, they enable users to develop entire IoT systems, using their comprehensive set of defined object models for providing semantic interoperability. Figure 3.1 shows a model of the general LWM2M and IPSO stack.

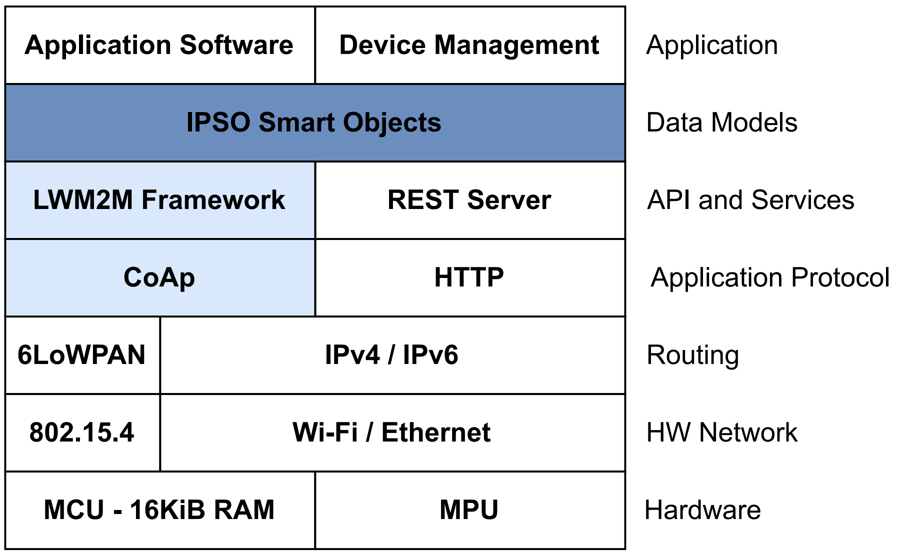


Figure 3.1: LWM2M and IPSO Stack

All of IPSO's objects are registered in the LWM2M OMA Naming Authority (OMNA) [254], and are mapped into a URI path structure that consist of three different components: Object ID, Instance ID, and Resource ID. These components are separated by a "/" character and are represented in the following form: Object ID/Instance ID/Resource ID. The Object ID contains the registered ID of a defined object in OMNA. The Instance ID represents a particular instance of the object, while the Resource ID represents observable properties of the object.

As an example, we describe IPSO's defined humidity object [251]. Humidity's Object ID is registered with the number 3304 in OMNA. The Instance ID are 0 as there currently is only one instance of the object. 5700 is humidity's defined Resource ID for representing the "Sensor Value" property. This results in the following URI: 3304/0/5700. Figure 3.2 shows humidity's object and resource definitions, including a simple usage example. Physical objects are able to link multiple defined smart objects together for diverse metadata representations such as temperature, humidity, voltage and over 50 other defined smart objects [96].

IPSO Humidity Object Definition

Name	Object ID	Object URN	Instances	Mandatory
Humidity	3304	urn:oma:lwm2m:ext:3304	Multiple	Optional

IPSO Humidity Resource Definitions

ID	Name	Operations	Instances	Mandatory	Type	Range or Enumeration	Units	Description
5700	Sensor Value	Read	Single	Mandatory	Float			Last or Current Measured Value from the Sensor
5701	Sensor Units	Execute	Single	Optional	String			Measurement Units Definition
5601	Min Measured Value	Read	Single	Optional	Float			The minimum value measured by the sensor since power ON or reset
5602	Max Measured Value	Read	Single	Optional	Float			The minimum value measured by the sensor since power ON or reset
5603	Min Range Value	Read	Single	Optional	Float			The minimum value that can be measured by the sensor
5604	Max Range Value	Read	Single	Optional	Float			The maximum value that can be measured by the sensor
5605	Reset Min and Max Measured Values	Read	Single	Optional				Reset the Min and Max Measured Values to Current Value

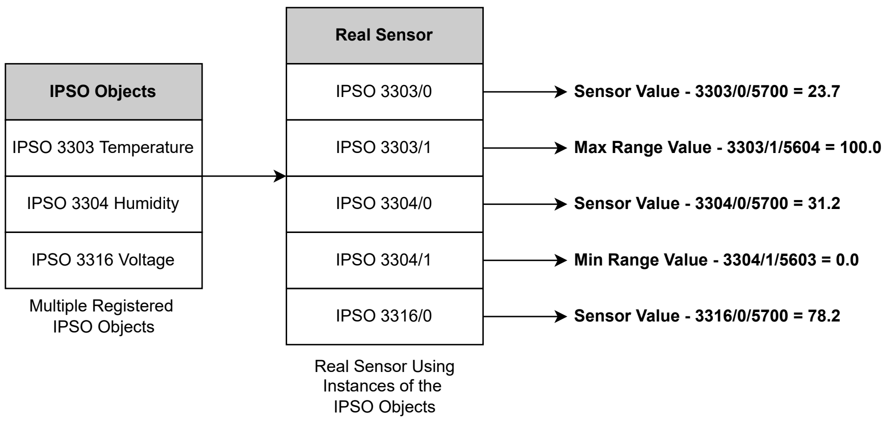

IPSO Objects
IPSO 3303 Temperature
IPSO 3304 Humidity
IPSO 3316 Voltage
Multiple Registered IPSO Objects
Real Sensor
IPSO 3303/0
IPSO 3303/1
IPSO 3304/0
IPSO 3304/1
IPSO 3316/0
Real Sensor Using Instances of the IPSO Objects
Sensor Value - 3303/0/5700 = 23.7
Max Range Value - 3303/1/5604 = 100.0
Sensor Value - 3304/0/5700 = 31.2
Min Range Value - 3304/1/5603 = 0.0
Sensor Value - 3316/0/5700 = 78.2

Figure 3.2: IPSO Humidity Object Definition and Usage Example [251]

3.2.2 Open Connectivity Foundation (OCF)

The Open Connectivity Foundation (OCF) [4] prescribes an entire IoT framework which currently consists of 19 specifications on their website [5], as of version 2.2.5. To mention a few, the core framework specifies the mandatory architectures, features, resources, and protocols for enabling IoT implementations with OCF. Another specification describes OCF's security aspects, while their bridging specification defines translations and mappings for establishing compatible bridges between OCF resources and other non-OCF standards. It also provides specifications for defined resource types and device types for the OCF resource model that will by our primary focus for this overview.

The OCF framework provides core functionalities for exchanging messages between applications and systems in an interoperable manner. It provides capabilities for identification and addressing, device discovery and management, defining resource models and more. In order to utilize the framework, the developers need to implement the required specification functionalities. OCF enables framework interactions between clients and servers where the servers contains instances of the relevant object representations, while the clients sends requests to a server for retrieving object states. Interactions between the clients and servers are performed using RESTful operations such as CREATE, READ, UPDATE, DELETE, and NOTIFY (CRUDN) that can be mapped to both the CoAP and MQTT protocols [2]. Figure 3.3 shows OCF's functional block diagram of the required functionalities.

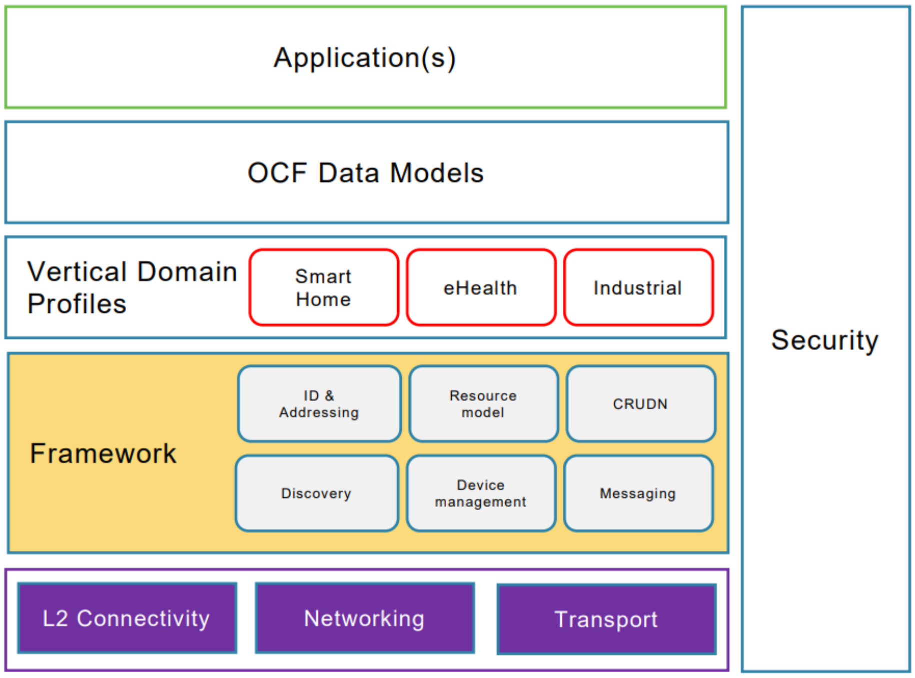


Figure 3.3: OCF Functional Block Diagram [2]

OCF's resource model provides abstractions for defining different resource types and device types that can operate in diverse application environments. OCF's current resource type specification [3] defines 180 different resource types that can be utilized for various metadata representations. The device specification consists of different device representations that define a list of the minimum required resource types that needs to be implemented for each specified device type. Each resource begins with the URI prefix: where "oic" is the reserved namespace for all OCF specifications.

An OCF resource type consists of several built-in properties: "rt", "if", "n", and "id". The first one, "rt", declares the specified ID of a resource type with the following URI prefix: "oic.r.{type name}", where "oic.r" is the reserved namespace for all OCF specified resource types. Next, "if" declares the the supported OCF interfaces for each resource type which are used to provide defined access rules for the API requests and responses. "n" is an optional property for declaring a human-readable name to the resource. Lastly, "id" is another optional property for declaring a unique identifier for a specific resource instance. Additional properties are enabled based on each defined resource type. All the resource properties are represented in name-value pairs and are stored in JSON schemas. As an example, figure 3.4 shows the definition of the humidity resource type, which has the following prefix "oic.r.humidity".

The Property definitions of the Resource with type "rt" = "oic.r.humidity"

Property name	Value type	Mandatory	Access mode	Description
rt	Array	No	Read Only	The Resource Type.
desiredHumidity	Integer	No	Read Write	The desired value for humidity.
humidity	Integer	Yes	Read Only	The current sensed value for humidity.
n	Array of strings, integers and boolean	No	Read Write	
id	Array of strings, integers and boolean	No	Read Write	
if	Array	No	Read Only	The OCF Interface set supported by this Resource.
desiredHumidity	Integer	Yes	Read Write	The desired value for humidity.

The CRUDN operations of the Resource with type "rt" = "oic.r.humidity"

CREATE	READ	UPDATE	DELETE	NOTIFY
	get	post		observe

Figure 3.4: OCF Humidity Resource Type Definition [3]

3.2.3 Open Platform Communications Unified Architecture (OPC UA)

OPC UA is a platform-independent standard that integrates a framework for enabling abstract message exchanges between systems and applications in diverse industrial domains. The standard provides an extensive specification on their website [66] that consists of multiple parts.

OPC UA's information model specification [65] provides flexible ways for digitally describing many different physical objects and processes, for representing the structure and semantics of data. OPC UA provides a basic unit of information that is referred to as a node, which is used to create every other related type of information in the model. These interconnected nodes are structured in an object-oriented manner, which forms the OPC UA address space model [63]. The OPC UA address space model defines a meta model that establishes the main rules for how OPC UA information models can be created and represented. The model defines base node class that extends a hierarchy of eight other optional node classes that inherits from it. These node classes are: Object, ObjectType, Variable, VariableType, DataType, ReferenceType, Method, and View. These node classes can be used for describing and representing all kinds of metadata and system information based on the users requirements. Each node class provide a set of both mandatory and optional attributes, while the relationships between the nodes are defined using references. Figure 3.5 shows a high level representation of the meta model.

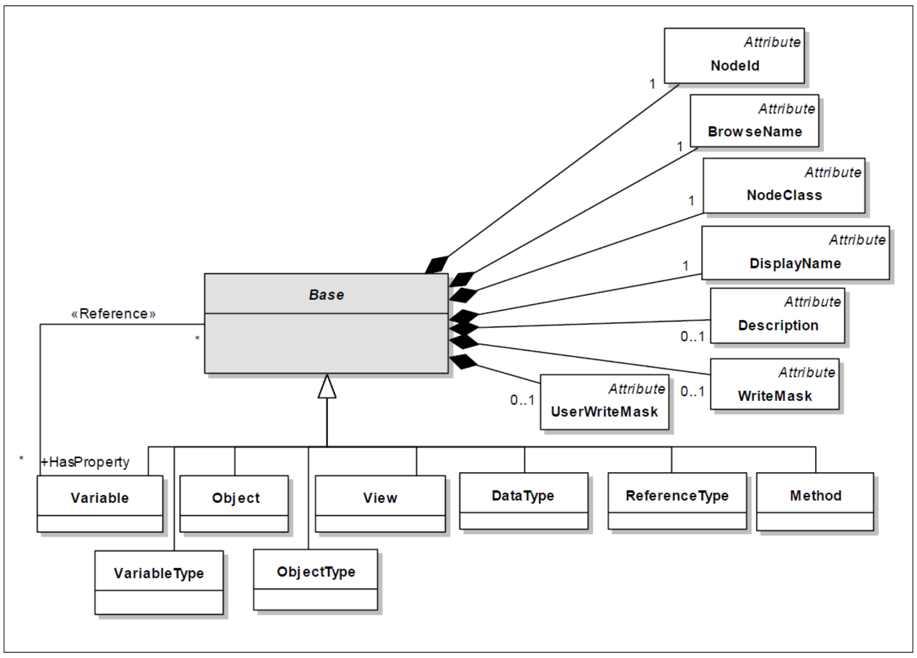


Figure 3.5: OPC UA Meta Model [63]

The OPC UA defines a client-server based architecture for enabling interoperable message exchanging. The defined address space model is contained in a

OPC UA Server application, which implements several services [64] and exposes the node-based information models, enabling resource discovery. An OPC UA Client can then send requests to the OPC UA Server and retrieve accessible data. Messages are usually sent via requests and responses using HTTP, but the OPC UA standard also provide options for sending messages using publishers and subscribes with MQTT or AMQP. For data encodings, OPC UA supports several formats, including binary, JSON and XML [62]. Figure 3.6 shows a general OPC UA Server architecture.

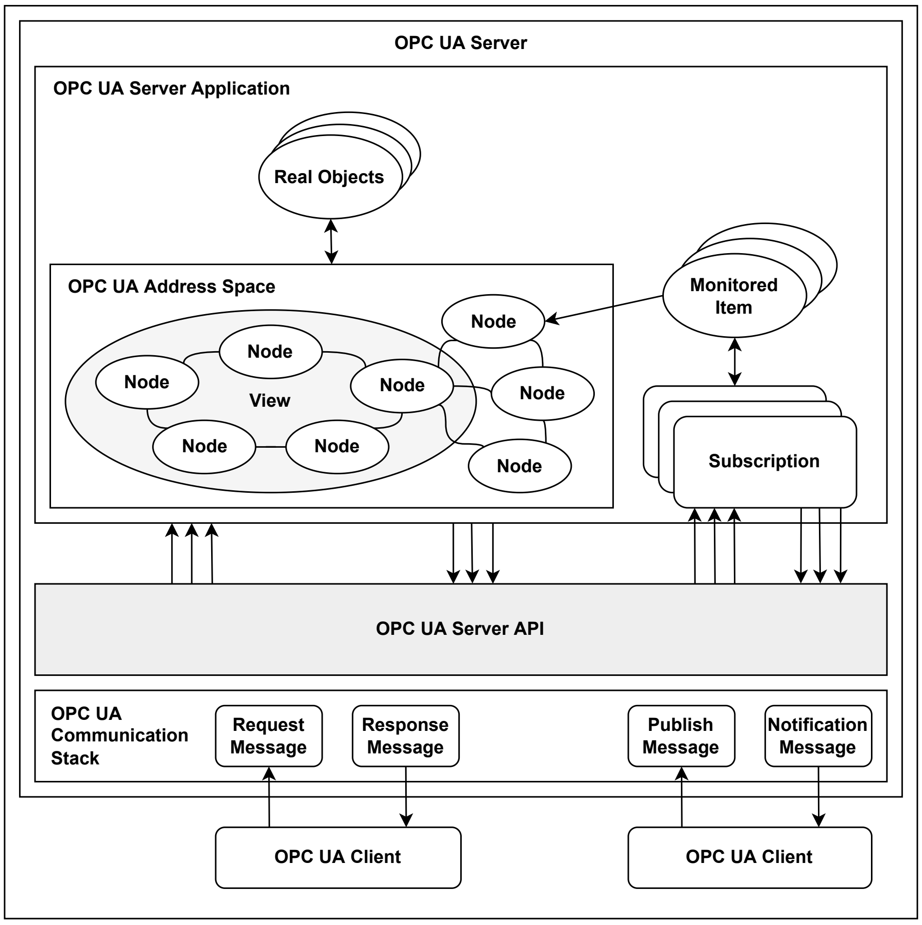


Figure 3.6: OPC UA Server Architecture [62]

3.2.4 Open Geospatial Consortium's Sensor Web Enablement (OGC SWE)

The Open Geospatial Consortium (OGC) [40] is an international industry consortium that develops standards for providing geographical information on the web. Among their available standards are the Sensor Web Enablement (SWE), which is a framework for standardizing the Sensor Web. The OGC SWE defines the Sensor Web as "web accessible sensor networks and archived sensor data that can be discovered and accessed using standard protocols and application programming interfaces" [194]. To achieve this, the OGC SWE has incorporated several open standards for data models, encodings, and service interfaces.

- **Sensor Model Language (SensorML):** SensorML [41] provides a framework for describing sensor systems and processes. SensorML treats all sensor system components as processes which can contain information such as inputs, outputs, parameters, methods and metadata descriptions. Its specification defines an information model and XML schema encodings that can be used for enabling discovery, task processing, and exploitation of observational data of sensors in the system. SensorML's official schemas can be found here: `http://schemas.opengis.net/sensorML/`.
- **Observations and Measurements (O&M):** O&M [36] provides abstract information models and XML schema encodings for describing and exchanging observations and measurements from sensors. An observation comprises of observing a feature of interest, defining appropriate feature properties and metadata, and procedures which are processes described in SensorML for producing the resulting measurement values. The O&M model information can be exchanged between different system endpoints. O&M's official schemas can be found here: `http://schemas.opengis.net/om/`.
- **SWE Common Data Model:** SWE Common [44] provides low-level data model definitions for exchanging sensor data between OGC SWE framework entities. It enables an XML encoding that should be utilized by the other OGC SWE standards to ensure better framework interoperability.
- **Sensor Observation Service (SOS)** SOS [42] is a web service interface for requesting, querying, and retrieving sensor observations, sensor metadata, and sensor system information. SOS's official schemas can be found here: `https://schemas.opengis.net/sos/`.
- **Sensor Planning Service (SPS)** SPS [43] is a web interface for sending user-driven requests to the sensor system. It can involve acquisition and tasking of sensors, or gathering information about other OGC SWE services in order to provide access to the data collected by tasked sensors. SPS's official schemas can be found here: `https://schemas.opengis.net/sps/`.
- **PUCK Protocol Standard:** PUCK [37] is a instrument protocol that can used for retrieving SensorML information and metadata from devices that enables the RS232 serial port or Ethernet connections.

Other OGC standards of interest are the SensorThings [38] standard, which is a RESTful API that provides interconnections between IoT systems and applications for tasking, managing and retrieving IoT data. Another standard is WaterML [39], which describes an information model for representing and exchanging observational underwater data, using other existing OGC standards. Lastly, the SWE Service Model [45] contains eight different packages with different data types that are commonly integrated with the other OGC SWE services.

The OGC SWE standards can cooperate in several ways for enabling diverse sensor system use case scenarios. An example of this is shown in figure 3.7. The

SOS and SPS services can be established based on SensorML sensor system descriptions. The sensor system retrieves SensorML metadata from sensors via the PUCK protocol, and register them to a catalog for enabling sensor discovery to client applications. The sensor system also registers to the SOS, and the SOS registers to the catalog. A client application #1 can for example send a request to SPS for tasking the sensor system to sample its sensors and publish their observations and measurements to SOS, using the O&M and SWE Common standards. A client application #2 can then connect to SOS and retrieve the observational data [19, 91, 259]. This is only some of the possible use cases that the OGC SWE standards can provide.

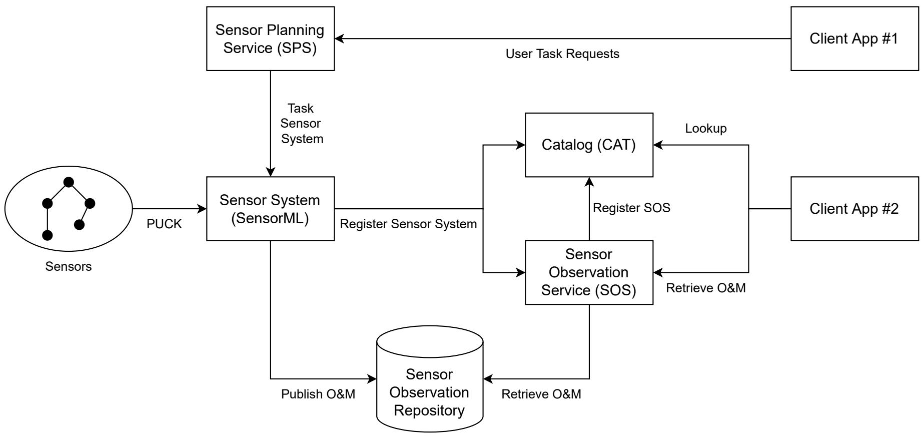


Figure 3.7: OGC SWE Functional Architecture and Interactions [19, 91, 259]

3.2.5 World Wide Web Consortium (W3C)

The World Wide Web Consortium (W3C) [48] is an international organization that focuses on developing standards for the World Wide Web. Several standards have been developed under their "Semantic Web" and "Semantic Sensor Network Incubator Group" that can be utilized for enabling semantic interoperability in IoT systems. Below, we describe the different standards.

Semantic Web
The Semantic Web [47] standards expands the capabilities on the web by providing meaning, ontologies and structure to web data. The standards are based around Linked Data [46], which consists of linking structured data with other data for facilitating machine understanding and relationships between web resources. The main standards of interest are RDF, RDFS, OWL, and SPARQL, which forms the Web of Data.

- **Resource Description Framework (RDF):** RDF [50, 88] defines graph-based models for representing semantic resources and their relationships on the web. The standard creates statements in form of triples, which contains the three following components: a subject, a predicate, and an object. An RDF triple can be seen as a directed graph where the subject and the object are nodes, while the predicate is the edge between them, directing against the object. Related sets of RDF triples forms the RDF graph models, linking together all the resources. RDF provides a vocabulary of 8 classes and 7 properties, and are are using URIs for identifiers and the XML format for data syntax. RDF's namespace can be found here: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`.

- **Resource Description Framework Schema (RDFS):** RDFS [29] is an extension of RDF's vocabulary that allows for taxonomic and hierarchical descriptions of RDF elements. RDFS provides 6 additional classes and 9 additional properties, giving a total of 12 classes and 16 properties for representing various resources and relationships. It also defines the domain and range of the RDF/RDFS properties. RDFS's namespace can be found here: `http://www.w3.org/2000/01/rdf-schema#`.
- **Web Ontology Language (OWL):** OWL [249, 49, 109, 185] is an ontology language that extends the vocabulary on top of RDF. However, compared to RDFS, OWL's vocabulary provides more expressive and logical descriptions to assign additional meaning to RDF triples in RDF graphs. OWL helps enriching RDF information with more meaningful semantics which can be useful in many scenarios. The standard's vocabulary consists of 17 classes and 23 properties. OWL's namespace can be found here: `http://www.w3.org/2002/07/owl#`.
- **SPARQL Protocol and RDF Query Language (SPARQL):** SPARQL [200, 51, 69] is an SQL-like query language used for querying RDF data. Its syntax and semantics supports aggregation, sub-queries, negation, property paths and several functions. The standard uses four query forms: SELECT, CONSTRUCT, ASK, and DESCRIBE. SELECT extracts values from RDF triples that matches a specified pattern, CONSTRUCT obtains an RDF graph and reconstructs it into a specified graph template, ASK returns a boolean based on whether the query pattern matches or not, and DESCRIBE obtains an RDF graph and describes its resources. Data formats such as JSON, XML and CSV/TSV can be utilized for serializing SPARQL results obtained from SELECT and ASK queries.

The combined usage of these standards gives users many flexible options for enabling semantic interoperability in web-based services and applications, which are commonly adapted in the domain of IoT. Figure 3.8 gives an example on how the standards can be utilized together, with the exception of OWL.

Semantic Sensor Network Incubator Group

The W3C's Semantic Sensor Network Incubator Group [35] has produced two ontology standards for enabling sensor-based semantics. These standards are the Semantic Sensor Network (SSN) and Sensor, Observation, Sample, and Actuator (SOSA) ontologies [86]. The SSN/SOSA ontologies were highly influenced by the OGC SWE's existing standards such as SensorML and O&M, as they already provided well-established annotations of sensors, observations and measurements. However, unlike the OGC SWE standards, the SSN/SOSA standards are fully integrated and compatible with the W3C's Semantic Web standards.

- **Semantic Sensor Network (SSN):** SSN [86] is an ontology language for describing sensors, observations, procedures, and properties for features of

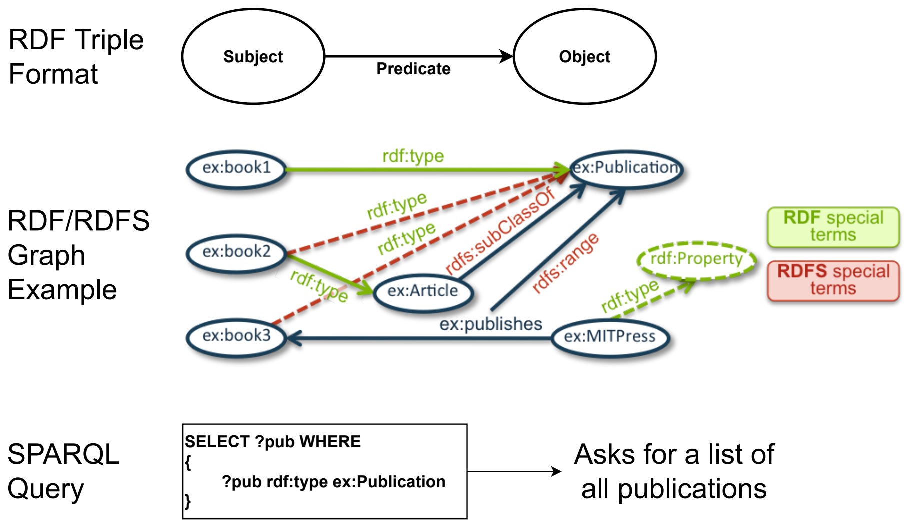


Figure 3.8: RDF, RDFS, and SPARQL Example [87]

interest. In the beginning, SSN was originally built around the Stimulus Sensor Observation (SSO) [95] design pattern, which provided a common ground for the SSN ontologies, as well as addressing the needs for light-weight semantics in Linked Data scenarios. In the newest versions, SSO is replaced by SOSA. Currently, the SSN ontology consists of 6 classes and 15 properties that can be utilized independently or collaboratively with SOSA's own classes and properties. They both provide different scopes with distinct ontologies that are suitable for different use cases. SSN's namespace can be found here: `https://www.w3.org/ns/ssn/`.

- **Sensor, Observation, Sample, and Actuator (SOSA):** SOSA [86] provides a light-weight ontology for modelling interactions between sensors, observations, samplings, and actuators. The ontology can either be used standalone, or together with SSN for broadening its use case scenarios. SOSA is an improved and expanded version of the SSO design pattern, providing a more modular ontology. SSO were limited to the scope of sensors and their observations, while SOSA extended the scope to also include classes and properties for for actuators and sampling. Currently, the SOSA ontology consists of 13 classes and 23 properties. SOSA's namespace can be found here: `https://www.w3.org/ns/sosa/`.

The SSN/SOSA ontologies defines several conceptual modules that covers all their main concepts. Figure 3.9 gives an overview of the conceptual ontology modules, as well as utilization of the main SSN/SOSA classes and properties from a observation perspective. The SSN components are shown in blue, while the SOSA components are shown in green.

3.2.6 Metadata Summary

The goal of all these standards is to accommodate for the lack universal metadata representations in the IoT and provide standardized ways for enabling

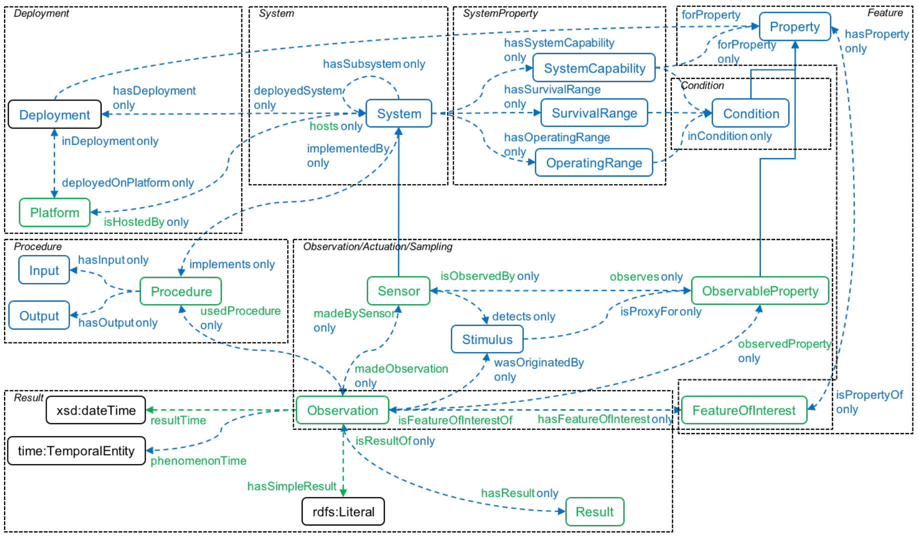


Figure 3.9: Overview of the SSN (Blue) and SOSA (Green) Classes and Properties [86]

syntactic and semantic interoperability to heterogeneous systems. Some provide information models with multiple object type definitions and semantic ontologies, while others provide entire IoT frameworks. The IPSO standard provide an object model that defines over 50 different smart objects. The standard is integrated as an information model layer on top of the LwM2M framework for providing more extensive metadata representations, enhancing the framework's semantic interoperability. The OPC UA provides an IoT framework that are mainly aimed towards industrial domains. The OPC UA's information model uses an object-oriented, node-based terminology for creating flexible and relational descriptions of different sensors and machinery. The OPC UA's specification provides services, mappings, security and supported communication technologies for abstract message exchanging between other applications and systems. The OCF also provides an entire IoT framework with multiple specifications that enable similar functionalities to OPC UA, allowing for abstract interactions and message exchanges with other applications and systems. The OCF's most recent resource type specification defines 180 different types of resources that can be used for defining diverse metadata representations of sensors and other IoT-based entities. For the OGC SWE, the SensorML, O&M, and SWE Common Data Model standards form their information model, while the framework's other standards provides service interfaces. The framework's standards can be integrated together in order to enable system architectures for requesting, tasking, managing, and retrieving various sensor data. The W3C's Semantic Web standards such as RDF, RDFS, OWL, and SPARQL provide the necessities for achieving semantic interoperability in web based heterogeneous systems, and can represent data in all kinds of different domains. For more IoT-specific domains, the SSN/SOSA standards can be applied to the Semantic Web standards, as they provide general-purpose ontologies for describing different aspects of sensors, observations and their processes. All the presented standards have both similar and different characteristics, but with the same purpose of virtually representing physical objects in an interpretative and contextual manner that facilities interoperable machine understandings.

Chapter 4

Cloud Platforms

This thesis evaluates three well established cloud platforms in order to assess their applicability and determine their potential use in the Smart Ocean project. In this chapter we give a detailed overview of the services, features and related concepts between the three cloud platforms.

4.1 Microsoft Azure

Microsoft Azure is a cloud platform developed by Microsoft, starting in 2010. Today, it is one of the largest and most popular cloud platforms on the market, with over 200 products on offering to assist users in developing their desired IT solutions [169]. In this section we give an overview of Microsoft Azure's offerings that are relevant to sensor data integration.

4.1.1 Azure IoT Overview

The Azure IoT is Microsoft's collection of relevant cloud services and applications that offers the components needed for building diverse IoT solutions on their platform. Figure 4.1 shows Microsoft's IoT reference architecture [151] with the recommended components, but no solution requires all of them. An Azure IoT solution involves (1) things that collects and sends data, (2) insights that can be gained based on the received data, and (3) actions that can be taken based on the acquired insights [126].

Things
The things are the physical devices or IoT applications that sends data to Azure. Azure IoT offers support for many types of IoT devices from different micro controller units running on either Azure RTOS [130] or Azure Sphere [131], to developer boards like Arduino or Raspberry Pi running on Azure's IoT developer kits [158]. Besides general IoT device support, Azure also offer support for various IoT edge devices by connecting them through edge gateways, using the Azure IoT Edge [121] service. These are devices that may be more resource-constrained, performs local preprocessing, or uses other communica-

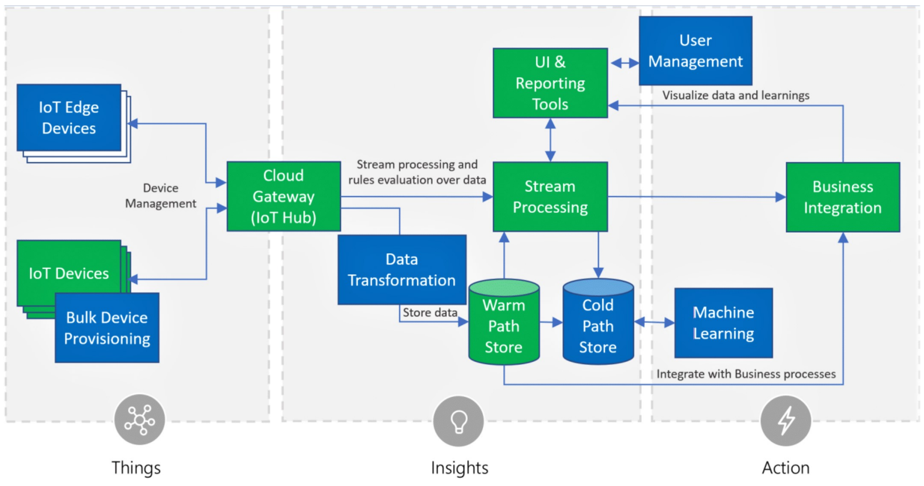


Figure 4.1: Azure IoT Reference Architecture [151]

tion protocols that are not supported by the platform and needs translation.

Cloud Gateway

The Azure IoT Hub [123] is Microsoft Azure's cloud gateway service that allows billions of IoT devices to securely connect and send data to the cloud. The service supports several connectivity options for IoT device integration, with both direct and indirect cloud gateway utilities, as well as edge gateways via the Azure IoT Edge service. The IoT Hub securely connects the devices via TLS encryption and provide both device-to-cloud (D2C) and cloud-to-device (C2D) messaging. The hub also provides device provisioning with per-device authentication, using either symmetric keys, X.509 certificates, or Trusted Platform Module (TPM). The Device Provisioning Service (DPS) [122] can also be integrated with IoT Hub in order to provide scalable and automatic provisioning without human intervention in a secure manner. In order to ingest received data to other cloud services and applications, the IoT Hub defines rules in an SQL-like query syntax to authorize where individual flows of data are routed. The IoT Hub also provide integration of digital twins for each cloud registered device. The digital twins are represented in JSON documents that contains desired and reported properties, tags, and state information of their physical counterparts [123]. The Azure Digital Twins [119] service allows users to design graph models of digital twins and further contextualize them in expandable, digital environments.

Insights

Once the IoT devices have successfully connected to the cloud, the received IoT data can then be processed and explored in several ways in order to gather insights about them [126]. The data typically gets routed either to a stream processing path, a warm storage path, or a cold storage path. The stream processing path is used for processing acquired data in near real-time. If incoming data flows gets processed with higher latency, then the data can either be stored in the warm storage path or cold storage path. The warm storage path temporarily stores recent data sets that can accommodate for longer delays and more detailed processing. The cold storage path is used for long-term storage of data, where time consuming analytics and batch processes can be performed on them. For the stream processing path, Azure provide several services that implements stream processing engines. Azure Stream Analytics [166] provides real-time analytics and processing of data from multiple data sources at once, while HDInsight [168] can be used for implementing several third-party frameworks for processing their data on the platform. For the warm storage path, the Azure Cosmos DB [116] service provide NoSQL databases for storing unstructured data, while the Azure SQL Database [132] service provide relational SQL databases for storing structured data. Lastly, for the cold storage path, services such as Azure Data Lake [137] or Azure Blob Storage [115] allows for archiving massive volumes of data. Meanwhile, services like Azure Machine Learning [125] or Azure Databricks [118] can be used to analyze the cold data. Additionally, Azure also offers multi-purpose services such as Azure Data Explorer [117] and Azure Time Series Insights [133] which provide both near real-time processing, storage, and analytics of large data volumes.

Actions

When insights have been gathered, different actions can be performed in order manage the cloud environment and produce value. Azure offers several business and managements services that can be integrated in the solutions. To mention a few, Power BI [135] is Microsoft's primary tool for visualizing data in Azure. Various services can connect and route their data to Power BI for data visualization, modeling, analysis, and creating business reports, using the service's interactive dashboards. For security, Azure Active Directory [113] is used for general identity and access management (IAM) in the cloud, while Microsoft Defender for IoT [153] provide a security environment for identifying IoT devices, and detecting their vulnerabilities and threats. Azure App Service [114] allows users to build and deploy their web applications and REST APIs on the platform, while Azure Monitor [127] provide general user-level monitoring and diagnostics of all the integrated applications and cloud services.

4.1.2 Azure Protocol Bindings, SDKs and APIs

Microsoft Azure provides several protocols, SDKs and APIs for enabling applications and systems to integrate and communicate with the platform. For device communication with IoT Hub, the platform supports HTTP 1.1 over native TLS, MQTT 3.1.1, and AMQP 1.0, where both the MQTT and AMQP protocol can optionally bind with WebSocket for communication via web applications [134]. HTTPS can be utilized via REST API posting, while MQTT and AMQP provide brokered messaging to specific topics. MQTT supports QoS level 0 and 1, but not level 2. In order for the devices to successfully send data to IoT Hub, the user must specify the chosen protocol's required endpoint and port number, serialize the payload data into a supported format, and add the device's registered authentication information in the message exchange. Table 4.1 shows Microsoft Azure's supported protocol bindings. Azure also technically supports other protocols via Azure IoT Edge, which allows protocols such as

Protocol	Authentication	Port	Endpoint
HTTPS 1.1	Symmetric Keys, X.509 Certificates, Trusted Platform Module	443	{iot-hub-name}/devices/{device-id}/ messages/events?api-version=2020-03-13
MQTT 3.1.1	Symmetric Keys, X.509 Certificates, Trusted Platform Module	8883	Client ID: {iot-hub-name}.azure-devices.net/, Topic ID: devices/{device-id}/messages/events
AMQP 1.0	Symmetric Keys, X.509 Certificates, Trusted Platform Module	5671	amqps://{device-id}@sas.{iot-hub-name}:{iot-hub-name} .azure-devices.net/devices/{device-id}/messages/events
MQTT over WebSocket	Symmetic Keys X.509 Certificates, Trusted Platform Module	443	Client ID: wss://{iot-hub-name}.azure-devices.net:433/ $iothub/websocket Topic ID: devices/{device-id}/messages/events
AMQP over WebSocket	Symmetic Keys X.509 Certificates, Trusted Platform Module	443	wss://{iot-hub-name}.azure-devices.net:433/ $iothub/websocket

Table 4.1: Microsoft Azure Protocol Bindings

CoAp or OPC UA Servers to be used via edge gateways and send data to IoT Hub. Legacy devices that uses other non-supported protocols can also connect to an indirect cloud gateway and be translated to a platform supported protocol before entering the direct cloud gateway.

Azure offers multiple service SDKs for developing back-end applications in many different languages [159], that can manage the user's cloud services. For the IoT Hub service, Azure offers both service SDKs that can be used for working with the hub via applications, and device SDKs [157] for facilitating the development of device client applications that can connect to the IoT Hub via Azure's supported protocols bindings. These clients can send D2C data, and optionally receive C2D data from the IoT Hub. Azure's device SDKs is available for the following languages: .NET, Python, Java, Node.js, C and embedded C [157]. Microsoft also provides an entire REST APIs reference documentation [129] for communicating and interacting with multiple cloud services. The REST API for IoT Hub [146] allows users to access and configure settings in the service, send device messages, update and retrieve device twins, and more.

4.2 Amazon Web Services

Amazon Web Services is currently the most popular cloud platform, holding the largest cloud market share of any other platform. The cloud platform started back in 2006 [208] with the introduction of its Simple Storage Service (S3) and has since spanned its offerings to over 200 widely adopted products [228]. In this section, we give an overview of Amazon's relevant cloud services and features for integrating sensor data to their platform.

4.2.1 AWS IoT Overview

Amazon Web Services provides multiple IoT services and general purpose services that can be integrated together for developing diverse IoT solutions on the platform. Amazon also provides a general IoT reference architecture, illustrated in their official AWS IoT Core developer guide [241]. Figure 4.2 shows the architecture, where the green components are the IoT-specified services, while the blue components showcase several general purpose services that can be integrated in the IoT solution.

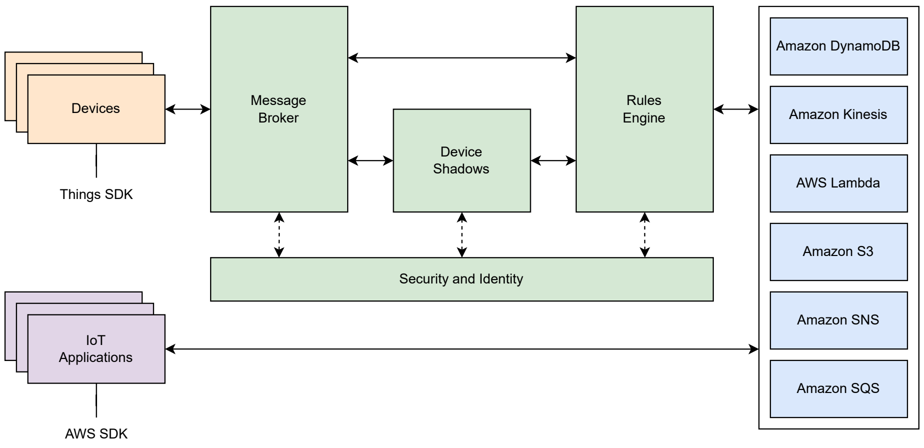


Figure 4.2: AWS IoT Core Architecture

Devices
Amazon provide several device and software options for connecting and sending data to the platform. AWS IoT ExpressLink [218] provides a comprehensive set of developed hardware modules from Amazon's collaborative partners, that can be easily integrated with the platform's services. The FreeRTOS [235] service provides a real-time operating system that can run on different microcontrollers for connecting to the platform's cloud gateways. For edge devices, the AWS IoT Greengrass [220] service provides open-source device software to run on devices or IoT client applications for enabling local preprocessing and message exchanging to the cloud via edge gateways. Meanwhile, users can deploy and manage the edge devices from the AWS portal. Besides these services, Amazon Web Services also offers multiple service SDKs for communication with different services, and device SDKs for simplifying the development of client applications for sending device data to to the cloud.

Cloud Gateway
AWS IoT Core [214] is Amazon's main service for securely connecting and managing IoT devices in the cloud, and forwarding the data to other connected cloud services and applications. IoT Core consists of a message broker that devices can publish messages to, based on their defined topics. Device shadows is IoT Core's name-variant for digital twins, which can be created for storing the reported and desired states of devices in JSON documents. Security and

identification is provided through device provisioning, authentication files, payload transit encryption, and AWS IAM [210]. The service offers support for several authentication formats, including X.509 certificates, SigV4, and custom authentication tokens using JSON Web Tokens (JWTs) or OAuth. Additional management and security can be integrated also be integrated with the use of AWS IoT Device Defender [215] and AWS IoT Device Management [216]. The rules engine defines the data ingestion rules for how individual flows of data are routed to other services, using a SQL-like language syntax.

Supporting Services
Amazon Web Services provides several relevant services for gathering insights and performing action on ingested data from IoT Core. AWS Lambda [224] can be used enabling serverless, on-demand execution of custom code for responding to triggered events in the system. Amazon Kinesis [204] provides real-time stream processing of different kinds of streams that are separated into four different modules. The modules involves analytics streams, data streams, video streams, and lastly data Firehouse which is used to transforming and prepare data streams to be loaded into data stores. Amazon DynamoDB [203] and Amazon Simple Storage Service (S3) [207] is the two main storage services that is used for IoT purposes. Amazon DynamoDB provides fast and flexible NoSQL databases for storing data both temporally and long-term. Amazon S3 provides a heavily scalable object storage that can store large data volumes for long-term archiving of data. Other services such as Amazon Simple Notification Service (SNS) [206] can be used for sending and receiving notifications from various system sources, while Amazon Simple Queue Service (SQS) [**aws**] can store data in distributed queues that can be retrieved by different applications and services. Finally, Amazon QuickSight [205] provide general utilities business processes and visualization of data that are queried from output streams and storage spaces.

4.2.2 AWS Protocol Bindings, SDKs, and APIs

Amazon Web Services supports the following protocols for sending messages to AWS IoT Core: HTTP 1.1 via native TLS, MQTT 3.1.1, and MQTT over WebSocket [234]. All message are sent to specified topics in the service's message broker. With the MQTT protocol, devices and IoT applications can publish and subscribe to topics with QoS level 0 or 1, while the HTTPS protocol only allows for publishing to topics via REST API posting. Devices and client connections needs to use the applied protocol's unique endpoint and supported port number. Additionally, they also need to input their provisioned device's authentication information. Table 4.2 shows the supported protocol bindings. Similar to Microsoft Azure, Amazon Web Services allows for utilization of other non-supported protocols via edge computing, using AWS IoT Greengrass and AWS IoT SiteWise. For SDK support, Amazon Web Services provides service SDKs for nine different languages, and IoT device SDKs for five different languages [242]. For API support, the majority are based on REST. Amazon provides an entire API reference document for IoT [223].

Protocol	Authentication	Port	Endpoint
HTTPS 1.1	SigV4, X.509 Certificates, Custom Authentication	443, 8883	https://{iot-core-name}:{port}/topics/{topic}?qos={0 or 1}
MQTT 3.1.1	X.509 Certificates, Custom Authentication	443, 8883	Client ID: {iot-core-name}, Topic ID: {topic}
MQTT over WebSocket	SigV4, Custom Authentication	443	wss://{iot-core-name}:433/mqtt

Table 4.2: Amazon Web Services Protocol Bindings

4.3 Google Cloud Platform

Google Cloud Platform is Google's suite of available Cloud Computing services. The platform came on the market in 2011, and has since become one of the most popular cloud platforms available, with over 100 products available. In this section we give an overview of the cloud platform, focusing on the sensor data integration aspects.

4.3.1 GCP IoT Overview

Similar to the two previous cloud platforms, Google Cloud Platform also provides an IoT reference architecture [75] that contains Google's most relevant cloud services for developing IoT solutions on their platform. Figure 4.3 shows Google's reference architecture. We now go through the main IoT components, following the flow of the architecture.

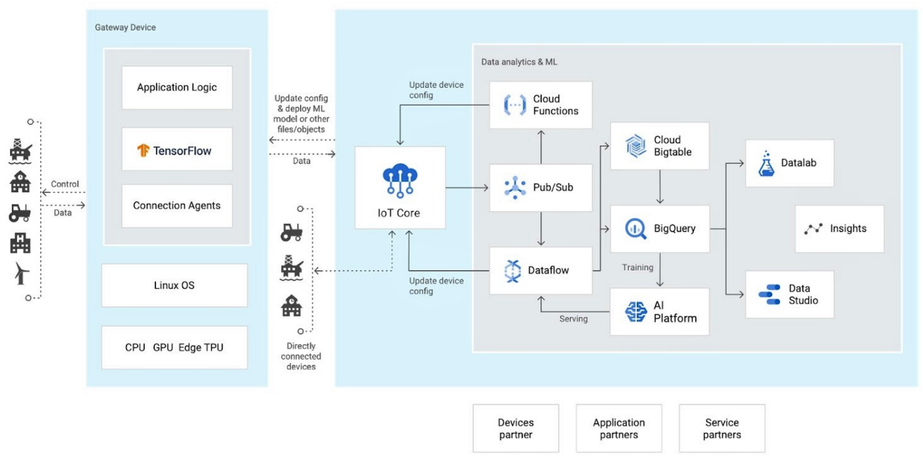


Figure 4.3: GCP IoT Reference Architecture [75]

Devices
Google Cloud Platform allows the use of several developer boards such as Arduino and Raspberry Pi for building IoT applications to send data. Google

in particular has a large focus when it comes to machine learning, and allows for cloud integration with the TensorFlow framework. Google has developed custom hardware chip called TensorFlow Processing Unit (TPU) [243] which run on TensorFlow for building and executing machine learning models on the platform. Google has also made an edge version of the chip, called Edge TPU [244] for running ML models on the edge with TensorFlow Lite, accelerating ML training in the cloud.

Cloud Gateway
In order to send IoT data to Google Cloud Platform, the devices and IoT applications need to negotiate access to the GCP IoT Core [75] service. The service provides two different protocol bridges that act as the gateway-options. IoT Core provide device provisioning via its device manager, and creates registries where related devices are stored. Each device gets a public/private key pair for authentication where X.509 client certificates are used as the public key, while the private key is needed for creating a the mandatory JWT. Google's Cloud Pub/Sub [77] service establishes brokered messaging capabilities for IoT Core. Each registry binds to created subscriptions and topic in order to enable an endpoint where messages can be stored.

Supported Services
In order to ingest data to other services, the platform provides pipelines that are offered by Cloud DataFlow [231]. DataFlow provide both routing of data to other service endpoints, as well stream processing capabilities. For storage of temporal data, Cloud Datastore [233] provide NoSQL databases, while Cloud BigTable [226] provides relational SQL databases. The Cloud Storage [229] service can be used for long-term archiving of data. BigQuery [225] enables high-scalable data querying to other services. It can route data to AI Platform [202] for performing machine learning, while Datalab [232] can be used for data exploration and analysis. Lastly, Google's Data Studio [230] provides dashboards for data visualization and general business integration.

4.3.2 GCP Protocol Bindings, SDKs, and APIs

In the Google Cloud Platform, devices can connect and send messages either through their HTTP bridge [78] or MQTT bridge [79]. Its supports HTTPS 1.1 over REST API posting for broker publishing, and both publishing and subscribing with MQTT 3.1.1. JWTs needs to be used for authentication over both protocol bridges. The Google Cloud Platform provides multiple service SDKs [227], but only provide device SDK support for Embedded C. For APIs, most of them are based around REST, but the Google Cloud Platform also offers gRPC API support for some of its cloud services, but in case of sensor data integration, REST APIs are still utilized as Google's IoT Core service only supports HTTP/1.1, and gRPC only works over HTTP/2. The main APIs of interest is the Cloud IoT API [73] and the Cloud Pub/Sub API [33] which is required in order to send messages to IoT Core and publish them to topics in Cloud Pub/Sub.

Protocol	Authentication	Port	Endpoint
HTTPS 1.1	JSON Web Tokens	443	https://cloudiotdevice.googleapis.com/v1/projects/{project-id}/locations/{cloud-region}/registries/{registry-id}/devices/{device-id}:publishEvent
MQTT 3.1.1	JSON Web Tokens, X.509 Certificates	443, 8883	Client ID: projects/{project-id}/locations/{cloud-region}/registries/{registry-id}/devices/{device-id}, Topic ID: devices/{device-id}/{topic}

Table 4.3: Google Cloud Platform Protocol Bindings

4.4 Cloud Platform Summary

We have now given an overview over each platform's most essential aspects for our thesis. In table 4.4 below, we compare the cloud platforms' most relevant features and services. While there are differences, they all fundamentally handle sensor data integration in very similar ways. Each platform has their own IoT service hub (IoT Hub/Core) which provide cloud gateway connections to the IoT devices, using their required protocol bindings and authentication mechanisms. IoT Hub/Core also provide device provisioning and general techniques for ingesting data to other services. All three platforms offers various general purpose services for providing data processing, storage, and business integration. Additionally, they also offer similar services for enabling serverless computing, application deployment, containers, and virtual machines. In terms of differences, the amount language and protocol support varies for each cloud platform. Microsoft Azure is the only platform that supports the AMQP protocol, while the two others are mainly limited to HTTPS and MQTT. Google Cloud Platform do not provide any options for digital twins, and have very limited device SDK support compared to the other two platforms. In terms of edge computing, both Microsoft and Amazon provide IoT-specified edge computing services on their platform. Google do also provide edge computing, but not any services that are specifically designed for IoT usage. Instead, Google are using the combined effort of the Anthos [209], Google Distributed Cloud Edge [239], and Google Kubernetes Engine [240] services for distributing managed hardware and software solutions that extends Google's cloud services to the edge. It uses containers and virtual machines for allowing users to run their desired cloud services closer to where their data is being produced and consumed. Amazon currently is the cloud market leader of the three, with Microsoft in second place, and Google Cloud Platform in third place. Amazon is also the most mature platform of the three as it started in 2006, and currently offers the most available cloud services of all the three platforms. Even so, Microsoft Azure is not far behind in terms of service offerings, and also provide many of the same features and capabilities for sensor data integration. Google Cloud Platform also provide many of the essential features and capabilities for enabling sensor data integration, but are also lacking in several areas when compared to the other two cloud platforms.

Parameters	Azure	AWS	GCP
Starting Year	2010	2006	2011

Service Models	IaaS, PaaS, SaaS	IaaS, PaaS, SaaS	IaaS, PaaS, SaaS
Device Software	Azure RTOS, Azure Sphere, Azure Plug and Play, Device SDKs	FreeRTOS, AWS IoT ExpressLink, Device SDKs	CPU/GPU/Edge TPU, Device SDKs
Cloud Gateway	Azure IoT Hub	AWS IoT Core	GCP IoT Core
IoT Message Protocols	HTTPS, MQTT, AMQP, MQTT over WebSocket, AMQP over WebSocket	HTTPS, MQTT, MQTT over WebSocket	HTTPS, MQTT
Serverless Computing	Azure Functions	AWS Lambda Functions	Cloud Functions
Device Authentication	Symmetric Keys, X.509 Certificates, Trusted Platform Module	X.509 Certificates, SigV4, OAuth, JSON Web Tokens	X.509 Certificates, OAuth, JSON Web Tokens
Security	TLS Encryption, Azure Active Directory	TLS Encryption, Amazon Cognito, AWS IAM	TLS Encryption, Google Cloud IAM
Device SDKs	.NET, Python, Java, Node.js, C, Embedded C	Python, Java, C++, JavaScript, Embedded C	Embedded C
API Integration	REST-based	REST-based	REST-based, gRPC-based
Service SDKs and APIs	.NET, Python, Java, C++, C, JavaScript/TypeScript, Go,	.NET, Python, Java, C++, JavaScript, Node.js, PHP, Ruby, Go	.NET, Python, Java, C++, Node.js, PHP, Ruby, Go

Storage	Azure SQL DB, Azure Cosmos DB, Azure Blob Storage, Azure Data Lake	Amazon S3, DynamoDB	Cloud Storage, Cloud BigTable, Cloud Datastore
Analytics	Azure Stream Analytics, Azure HDInsight, Azure Data Explorer, Time Series Insights	AWS IoT SiteWise, AWS IoT Analytics, Amazon Kinesis	BigQuery, Cloud Datalab, Cloud Dataflow
Visualization	Power BI	Amazon QuickSight	Looker, Google Data Studio
ML and AI	Azure Machine Learning, Azure Data Bricks	Amazon SageMaker, Amazon Augmented AI	Cloud ML Engine, AI Platform
Virtual Machine	Azure Virtual Machine	Amazon EC2	Google Compute Engine
Container	Azure Kubernetes Service	AWS Elastic Container, AWS Kubernetes Service	Google Kubernetes Engine
Deployment	Azure App Service	AWS Elastic Beanstalk	Google App Engine
Digital Twins	Azure Digital Twins	AWS IoT TwinMaker, AWS IoT Device Shadow	N/A
Edge Computing	Azure IoT Edge	AWS IoT Greengrass, AWS IoT SiteWise	Edge TPU, Anthos, Distributed Cloud Edge, Google Kubernetes Engine
Additional IoT Related Services	Azure IoT Central, MS Defender for IoT, Azure IoT Hub DPS, Azure Percept, Windows for IoT	Amazon SNS and SQS, AWS IoT Events, AWS IoT Things Graph, AWS IoT Defender, AWS IoT FleetWise, AWS IoT Device Management	Cloud Pub/Sub,

Table 4.4: Cloud Platform Comparison

Chapter 5

Design and Implementation

In this chapter, we go over the design and implementation aspects of the sensor data integration prototypes. Several client applications have been developed for each cloud platform, using SFI Smart Ocean's pilot demonstrator 1 as a case study. The GitHub link to all the code can be found in Appendix A.

5.1 Virtual Sensors and Client Interface

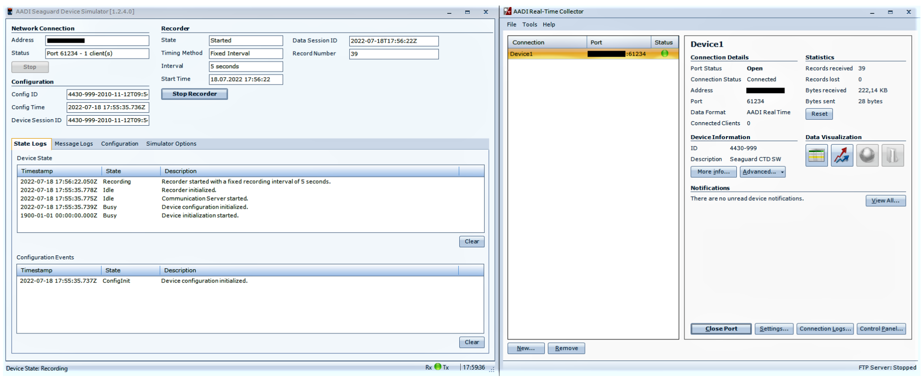
Aanderaa Data Instruments AS (AADI) [15] is one of the collaborating partners in the Smart Ocean project that are supplying with sensors and instrumentation for pilot demonstrator 1 (PD1). As mentioned in chapter 1.1.3, the physical underwater sensors for PD1 are not yet available for experimental prototyping. Instead, virtual sensors are used to simulate the physical sensors and instruments data streams. AADI have developed two applications that are being used for the sensor data integration prototypes: The AADI Device Simulator and the AADI Real-Time Collector. We have also developed a client interface that can communicate with these applications.

5.1.1 AADI Device Simulator

The AADI Device Simulator (DS) is an application that has the ability to simulate several of AADI's existing sensors and instruments. One of their offerings is the Seaguard platform [16] which spans multiple connected sensors and instruments that senses and collects various underwater parameters. The application simulates the Seaguard devices by producing the exact same data in XML formats. A new XML file is created every few seconds, where its attribute values are randomized each time. This is the virtual sensor messages that are going to be sent to the cloud platforms with our prototypes.

5.1.2 AADI Real-Time Collector

The second application is the AADI Real-Time Collector (RTC). The RTC is used to configure and control all the devices from AADI. The RTC can connect to each AADI device using their designated IP addresses and port numbers, and receive their data in real time. It can connect with the DS and receive its data as well. The RTC also allows other client applications to connect with the RTC through the Windows Communication Foundation (WCF) framework [138]. The clients can subscribe to a particular device and request its incoming messages. Depending on the scenario, these client applications can then visualize the data, store the data in a database, send the data to other applications, or in our case: send the data to different cloud platforms. Figure 5.1 shows the DS on the left and the RTC on the right.



Network Connection

Address	[redacted]
Status	Port 61234 - 1 client(s)

Configuration

Config ID	4430-999-2010-11-12T09:5
Config Time	2022-07-18 17:55:35.736Z
Device Session ID	4430-999-2010-11-12T09:5

Recorder

State	Started	Data Session ID	2022-07-18T17:56:22Z
Timing Method	Fixed Interval	Record Number	39
Interval	5 seconds		
Start Time	18.07.2022 17:56:22		

Device State

Timestamp	State	Description
2022-07-18 17:56:22.050Z	Recording	Recorder started with a fixed recording interval of 5 seconds.
2022-07-18 17:55:35.778Z	Idle	Recorder initialized.
2022-07-18 17:55:35.775Z	Idle	Communication Server started.
2022-07-18 17:55:35.739Z	Busy	Device configuration initialized.
1900-01-01 00:00:00.000Z	Busy	Device initialization started.

Configuration Events

Timestamp	State	Description
2022-07-18 17:55:35.737Z	ConfigInit	Device configuration initialized.

Device State: Recording

Connection	Port	Status
Device1	[redacted]:61234	●

Device1

Connection Details

Port Status	**Open**
Connection Status	Connected
Address	[redacted]
Port	61234
Data Format	AADI Real Time
Connected Clients	0

Statistics

Records received	39
Records lost	0
Bytes received	222,14 KB
Bytes sent	28 bytes

Device Information

ID	4430-999
Description	Seaguard CTD SW

Data Visualization

Notifications

There are no unread device notifications.

FTP Server: Stopped

Figure 5.1: AADI Device Simulator and AADI Real-Time Collector

5.1.3 Client Interface

The RTC application currently only support connections with .NET client applications that are enabling the WCF framework. The latest .NET version that fully supports the WCF framework is .NET Framework 4.8 [140]. Hence, the client interface as well as all our prototype applications are developed using this .NET version. All the projects are created using the Visual Studio IDE [165] and are contained in one solution [167]. The client interface consists of three separate projects that operate together for connecting and subscribing to the RTC devices, and forwarding the sensor messages to other applications.

RTC Client

The RTC client project consists of a console application that enables users to configure how they want to connect and subscribe to the RTC application, in order to retrieve the sensor messages. The client is capable of sending the acquired sensor messages to other applications in several ways. The client's main approach relies on the use of Microsoft Message Queuing (MSMQ) [150]. MSMQ enables an asynchronous queue that are decoupled from the applications. It allows applications to communicate across heterogeneous networks and systems that run at different times. The RTC client forwards its received sensor messages to this queue, where they are stored until other connecting applications retrieves them. All the projects in our Visual Studio solution uses developed methods for communicating with the MSMQ. This is how all our prototypes retrieves the sensor messages from the RTC client. The project utilizes the AADI.Realtime.Collector.ClientInterface.dll class library [17] that contains the classes and interfaces needed to successfully establish connection with the RTC. The RTC allows client applications to connect in one of the three following ways:

- **IPC Service:** The Inter Process Communication (IPC) [145] service is chosen if both the RTC application and client application are running on the same machine. IPC establishes a duplex (two-way) communication between the RTC and client interface through a named pipe [155]. This the most efficient way for connecting with the RTC.
- **TCP Service:** The TCP service is chosen if the RTC and client application are running on two separate machines on the same internal network. The client interface inputs the TCP service's IP address and port number in order to establish the connection with RTC.
- **ASMX Web Service:** The ASP.NET Web Service (ASMX) [112] exposes an interface that establishes a one-way connection over HTTP for enabling the RTC and the client application to communicate. This option is chosen if both applications are running on separate machines outside the local network. The client application inputs the ASMX Web Service's IP address and port number in order to establish connection the RTC.

Web API

A project containing a REST API has also been created. The web API serves as another option for sending the RTC client's retrieved sensor messages to other applications that are able to fetch its data. The project relies on MSMQ for retrieving the sensor messages, and then sends the data via HTTP POST commands. The web API also requires the Internet Information Service (IIS) [144]. IIS enables a web server running on the operating system, that hosts a WCF service for sending messages with the web API.

There are also another option for .NET applications that don't enable either IIS or MSMQ, to retrieve sensor data from the RTC client. They do this by integrating their own self-hosted web API, using the Open Web Interface for .NET (OWIN) [156]. OWIN enables web applications to be decoupled from the web server, and provides a pipeline-based middleware for handling the requests and responses between them. The RTC client creates a general message queue, using .NET's own Message class [149]. By entering the self-hosted API's specific hosting endpoint in the RTC client console, the OWIN-based applications are able to request the sensor messages from this queue.

Class Library

Another created project contains a class library called MessageHandling. It consists of multiple helper methods that are called by the different projects in the solution. The MessageHandling class contains methods for communicating with the MSMQ, creating message queue for OWIN applications, and serializing XML sensor messages to JSON. Some of the methods are utilized by the RTC client and web API, while others are utilized by the sensor data integration prototypes. The MessageHandling class library were created to facilitate the development process and provide more flexible functionality in the solution.

5.2 Prototype Design and Workflow

In this section, we describe the overall design and workflow between all the components. The DS and RTC applications are running locally on our computer. The DS produces virtual sensor messages every few seconds, while the RTC establishes connection with the DS through our computer's local IP address and default port number. The RTC registers a device entity that receives the sensor messages from the DS. While these two applications are actively running, we initialize the client interface projects in Visual Studio and our client console application will start up. After choosing the appropriate connection settings in the console, our client interface connects and subscribes to the RTC's established device and requests its sensor messages. The client interface then continuously forwards the newly retrieved sensor messages to the MSMQ, making them accessible to our prototypes. Each prototype consists of several client console applications that are designed for each platform-supported protocol binding. In our Visual Studio solution, we run our prototype of choice which collects the sensor messages from MSMQ, serializes them to JSON, and then sends them to the cloud platform's IoT message hub service (IoT Hub/Core). Each running prototype continues to collect and send virtual sensor messages from the MSMQ until the user manually quits the application. Figure 5.2 shows a high level model of the relationships between the components. For the rest of this chapter, we go over the prototype implementations for each cloud platform.

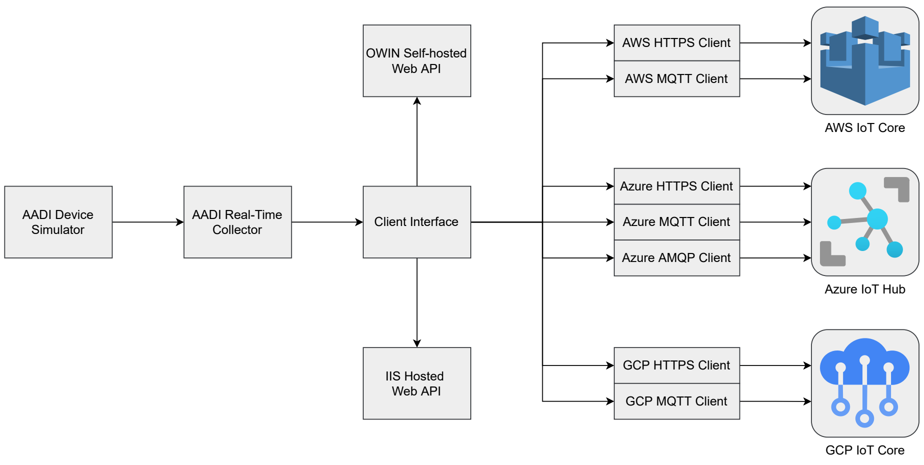


Figure 5.2: Model of the relationships between the different components

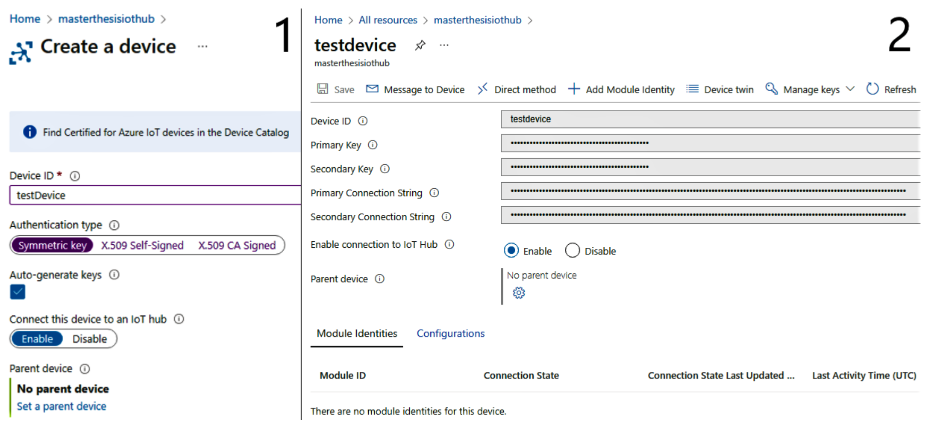
5.3 Prototype 1 - Microsoft Azure

The first prototype is sensor data integration with Microsoft Azure. The prototype consists of one client application that continuously retrieves sensor data from the client interface and sends it to Azure IoT Hub, either via HTTPS, MQTT, or AMQP. The client uses Azure's supported device SDK for .NET.

5.3.1 Azure IoT Hub and Device Setup

Before our prototype can send the sensor messages to Microsoft Azure, there are several prerequisites that needs to be fulfilled on the cloud platform. The user first and foremost, needs to create an Azure account and then log in to the Azure portal. Azure IoT Hub [123] is the cloud service we are going to be using for receiving the prototype's messages in the cloud. We first create a resource group [148], which is a container that holds all the services that are used in a particular solution. From here, we can start setting up our IoT Hub and input the required information [136]. In the basics section, we give our hub a name, selects our region, add our account subscription, and choose our created resource group. In the management section, we choose the desired pricing and scale tier. There are mainly two tiers, a standard tier that allows the user to take advantage of all the hub's features, and a cheaper basic tier that limits several features like IoT edge, C2D messaging, and digital twins. Each tier also provide different scaling levels that determine the extra cost and number of messages that can be sent to the hub per day [124]. After choosing the appropriate settings, we then review and create our IoT Hub.

Once we have our IoT Hub, we can start creating devices through IoT Hub device provisioning. We navigate into our IoT Hub and create a device. We give our device a name that will be the ID for that particular device. We also have several authentication options to choose from, including symmetric keys, self-signed X.509 certificate, or signed X.509 certificate authority (CA). For our setup, we choose the symmetric keys option with auto-generation on for the primary and secondary keys. After the device has been created, we can enter our device's information where we can see our primary and secondary key values, as well the primary connection string that we need for our prototype. Figure 5.3 shows the device creation and its information. We now have the prerequisites in order. Next we give an overview of the prototype implementation with the .NET device SDK.



Home > masterthesisiothub >

Create a device ···

Find Certified for Azure IoT devices in the Device Catalog

Device ID * ⓘ
testDevice

Authentication type ⓘ
[✓ Symmetric key] X.509 Self-Signed X.509 CA Signed

Auto-generate keys ⓘ
✓

Connect this device to an IoT hub ⓘ
[✓ Enable] Disable

Parent device ⓘ
No parent device
Set a parent device

Home > All resources > masterthesisiothub >

testdevice ···
masterthesisiothub

Save | Message to Device | Direct method | + Add Module Identity | Device twin | Manage keys | Refresh

Device ID ⓘ	testdevice
Primary Key ⓘ	••
Secondary Key ⓘ	••
Primary Connection String ⓘ	••
Secondary Connection String ⓘ	••
Enable connection to IoT Hub ⓘ	[✓ Enable] Disable
Parent device ⓘ	No parent device

Module Identities | Configurations

Module ID	Connection State	Connection State Last Updated ...	Last Activity Time (UTC)

There are no module identities for this device.

Figure 5.3: Azure Device Creation (1) and Device Information (2)

5.3.2 Azure Implementation with Device SDK

Microsoft Azure is the only cloud platform of the three that offers device SDK support for .NET. This simplifies the implementation process as the Microsoft.Azure.Devices.Client library [154] offers several classes, interfaces, enums and delegates that reduces the amount of code needed to successfully send sensor messages to our IoT Hub. If device SDKs were not available, we would have to utilize other open-source libraries for MQTT and AMQP, and language specific methods for developing REST API posting requests with HTTPS. Examples of this with HTTPS and MQTT are shown in the client implementations for Amazon Web Services in section 5.4, and Google Cloud Platform in section 5.5.

We begin by declaring several arguments. The DeviceClient class [139] contains methods for sending and receiving IoT Hub messages, while the iotHubName, deviceId and primaryKey strings contains the name of our IoT Hub, the name ID of our created device, and the device's related primary key. These string-values will be unique for each user. Then all of these values are used in the last string, creating the full connection string needed to authenticate our device with IoT Hub.

```
private static DeviceClient azureClient;

private static string iotHubName = "{your iot hub name}";
private static string deviceId = "{your device name}";
private static string primaryKey = "{device primary key}";

private static string connectionString = $"HostName={iotHubName}.
    azure-devices.net;DeviceId={deviceId};SharedAccessKey={
    primaryKey}";
```

The device SDK library's TransportType enum [164], allows us to select our communication protocol of choice for sending sensor messages to IoT Hub. We can choose between HTTPS 1.1, MQTT 3.1.1, AMQP 1.0, MQTT over WebSocket, or AMQP over WebSocket. All the needed functionality for each protocol is contained in this one line.

```
private static readonly TransportType protocol = TransportType.
    Mqtt;
//private static readonly TransportType protocol = TransportType.
    Http1;
//private static readonly TransportType protocol = TransportType.
    Amqp;
//private static readonly TransportType protocol = TransportType.
    Mqtt_WebSocket_Only;
//private static readonly TransportType protocol = TransportType.
    Amqp_WebSocket_Only
```

Next, we have the method developed for retrieving the XML messages from the client interface via MSMQ, serialize them to JSON, and then sending them to IoT Hub. The method uses the task asynchronous programming model [163] to reduce potential blocking of processes that needs to complete before other processes can continue, making the application more responsive. The method consists of a while-loop that will continue until the user manually quits the client. On line 9, we use the ReadMessagesFromClient method from our MessageHandling class library to collect all the current available messages in the

MSMQ. We create an array list that has the same length as the current amount of messages. If the MSMQ is empty, the loop waits for five seconds and tries again. If the MSMQ has available messages, each message will be serialized to JSON and transformed to a sequence of one to four bytes using UTF-8 encoding [141]. Then on line 28, we use the Azure's SendEventAsync method that sends the current JSON message to IoT Hub, before the loop starts over and repeats the process for the next sensor message. This method works as the basis for collecting and sending sensor messages in all the prototypes, with some adjustments to suit each client.

```
 1  private static async Task SendDeviceToCloudMessagesAsync(
        CancellationToken cts){
 2      string[] messages;
 3      string[] jsonMessage;
 4      int messageCounter = 0;
 5      int counter;
 6
 7      while (!cts.IsCancellationRequested){
 8          counter = 0;
 9          messages = MessageHandling.ReadMessagesFromClient();
10          jsonMessage = new string[messages.Length];
11
12          if (messages.Length == 0 || messages == null){
13              Console.WriteLine("There is no messages in the queue");
14              await Task.Delay(5000);
15          }else{
16              foreach (string sensorMessage in messages){
17                  // Create JSON message
18                  jsonMessage[counter] = MessageHandling.
                        SerializeToJson(sensorMessage);
19                  using var message = new Message(Encoding.ASCII.
                        GetBytes(jsonMessage[counter])){
20                      ContentType = "application/json",
21                      ContentEncoding = "utf-8",
22                  };
23                  counter++;
24                  messageCounter++;
25
26                  // Send the telemetry message
27                  await azureClient.SendEventAsync(message);
28                  Console.WriteLine($"{DateTime.Now} > Message {
                        messageCounter} sent" );
29                  await Task.Delay(5000);
30              }
31          }
32      }
33  }
```

We also create a method for validating our connection string. The IotHubConnectionStringBuilder class [147] takes our created connection string in as a parameter and and validates it based on the IoT Hub host name, device ID, and primary key. We get an exception error if the connection string is invalid.

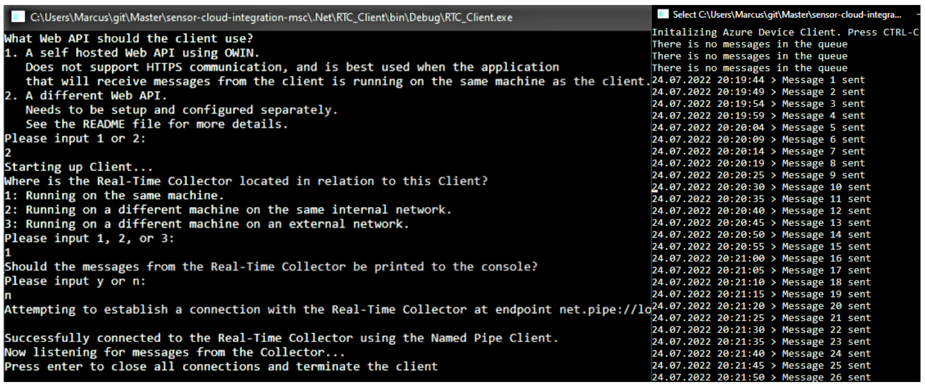
```
private static void ValidateConnectionString(string[] args){
    if (args.Any()){
        try{
            var cs = IotHubConnectionStringBuilder.Create(args[0]);
            connectionString = cs.ToString();
        }catch (Exception){
            Console.WriteLine($"Error: Unrecognizable parameter '{
                args[0]}' as connection string.");
            Environment.Exit(1);
        }
    }else{
        try{
            _ = IotHubConnectionStringBuilder.Create(
                connectionString);
        }catch (Exception){
            Console.WriteLine($"Error: Unrecognizable parameter '{
                connectionString}' as connection string");
            Environment.Exit(1);
        }
    }
}
```

Finally, the main method initializes all the methods. Our connection string gets validated before it gets passed in as a parameter in the DeviceClient class, together with our chosen protocol. We also set up a cancellation token that lets the user manually quit the client by typing CTRL-C in the console. When the DeviceClient has established a connection with our IoT Hub, our sensor messaging method will continue to send messages in a loop, until the user manually quits the client.

```
private static async Task Main(string[] args){
    Console.WriteLine("Initalizing Azure Device Client. Press CTRL-
        C to exit...");

    //This method validates our connection string as a parameter
    ValidateConnectionString(args);

    //The DeviceClient connects to the IoT hub using our connection
         string and chosen protocol
    azureClient = DeviceClient.CreateFromConnectionString(
        connectionString, protocol);

    //Sets up a condition to quit the client, press CTRL-C to quit
    using var cts = new CancellationTokenSource();
    Console.CancelKeyPress += (sender, eventArgs) => {
        eventArgs.Cancel = true;
        azureClient.CloseAsync();
        azureClient.Dispose();
        cts.Cancel();
        Console.WriteLine("Exiting...");
    };

    //Method for sending sensor data to IoT Hub
    await SendDeviceToCloudMessagesAsync(cts.Token);
}
```

5.3.3 Azure Deployment and Output

Before the prototype deployment, the DS and RTC applications must be running in the background, and actively produce sensor messages. In Visual Studio, we choose the projects that are going to run on startup. We select the client interface projects and the Azure client project. When we run the applications, we get two consoles displayed as shown in figure 5.4, the client interface console on the left, and the Azure client console on the right. In the client interface console we need to choose the IPC service, since both the client interface and the RTC are running on the same machine. If the console gives information that the RTC client successfully connected to the RTC, the client interface will then start receiving XML sensor messages and make them accessible for the Azure client via MSMQ. The Azure client then collects all the current messages in MSMQ and sends them to IoT Hub. New MSMQ messages are collected each time the loop repeats. If the Azure client successfully connected to IoT Hub, the console will write that the messages has been sent. The exact same process will be done for all the other prototypes as well.



```
C:\Users\Marcus\git\Master\sensor-cloud-integration-msc\.Net\RTC_Client\bin\Debug\RTC_Client.exe
What Web API should the client use?
1. A self hosted Web API using OWIN.
   Does not support HTTPS communication, and is best used when the application
   that will receive messages from the client is running on the same machine as the client.
2. A different Web API.
   Needs to be setup and configured separately.
   See the README file for more details.
Please input 1 or 2:
2
Starting up Client...
Where is the Real-Time Collector located in relation to this Client?
1: Running on the same machine.
2: Running on a different machine on the same internal network.
3: Running on a different machine on an external network.
Please input 1, 2, or 3:
1
Should the messages from the Real-Time Collector be printed to the console?
Please input y or n:
n
Attempting to establish a connection with the Real-Time Collector at endpoint net.pipe://lo

Successfully connected to the Real-Time Collector using the Named Pipe Client.
Now listening for messages from the Collector...
Press enter to close all connections and terminate the client
```

```
Select C:\Users\Marcus\git\Master\sensor-cloud-integra...
Initalizing Azure Device Client. Press CTRL-C
There is no messages in the queue
There is no messages in the queue
There is no messages in the queue
24.07.2022 20:19:44 > Message 1 sent
24.07.2022 20:19:49 > Message 2 sent
24.07.2022 20:19:54 > Message 3 sent
24.07.2022 20:19:59 > Message 4 sent
24.07.2022 20:20:04 > Message 5 sent
24.07.2022 20:20:09 > Message 6 sent
24.07.2022 20:20:14 > Message 7 sent
24.07.2022 20:20:19 > Message 8 sent
24.07.2022 20:20:25 > Message 9 sent
24.07.2022 20:20:30 > Message 10 sent
24.07.2022 20:20:35 > Message 11 sent
24.07.2022 20:20:40 > Message 12 sent
24.07.2022 20:20:45 > Message 13 sent
24.07.2022 20:20:50 > Message 14 sent
24.07.2022 20:20:55 > Message 15 sent
24.07.2022 20:21:00 > Message 16 sent
24.07.2022 20:21:05 > Message 17 sent
24.07.2022 20:21:10 > Message 18 sent
24.07.2022 20:21:15 > Message 19 sent
24.07.2022 20:21:20 > Message 20 sent
24.07.2022 20:21:25 > Message 21 sent
24.07.2022 20:21:30 > Message 22 sent
24.07.2022 20:21:35 > Message 23 sent
24.07.2022 20:21:40 > Message 24 sent
24.07.2022 20:21:45 > Message 25 sent
24.07.2022 20:21:50 > Message 26 sent
```

Figure 5.4: Application Deployment

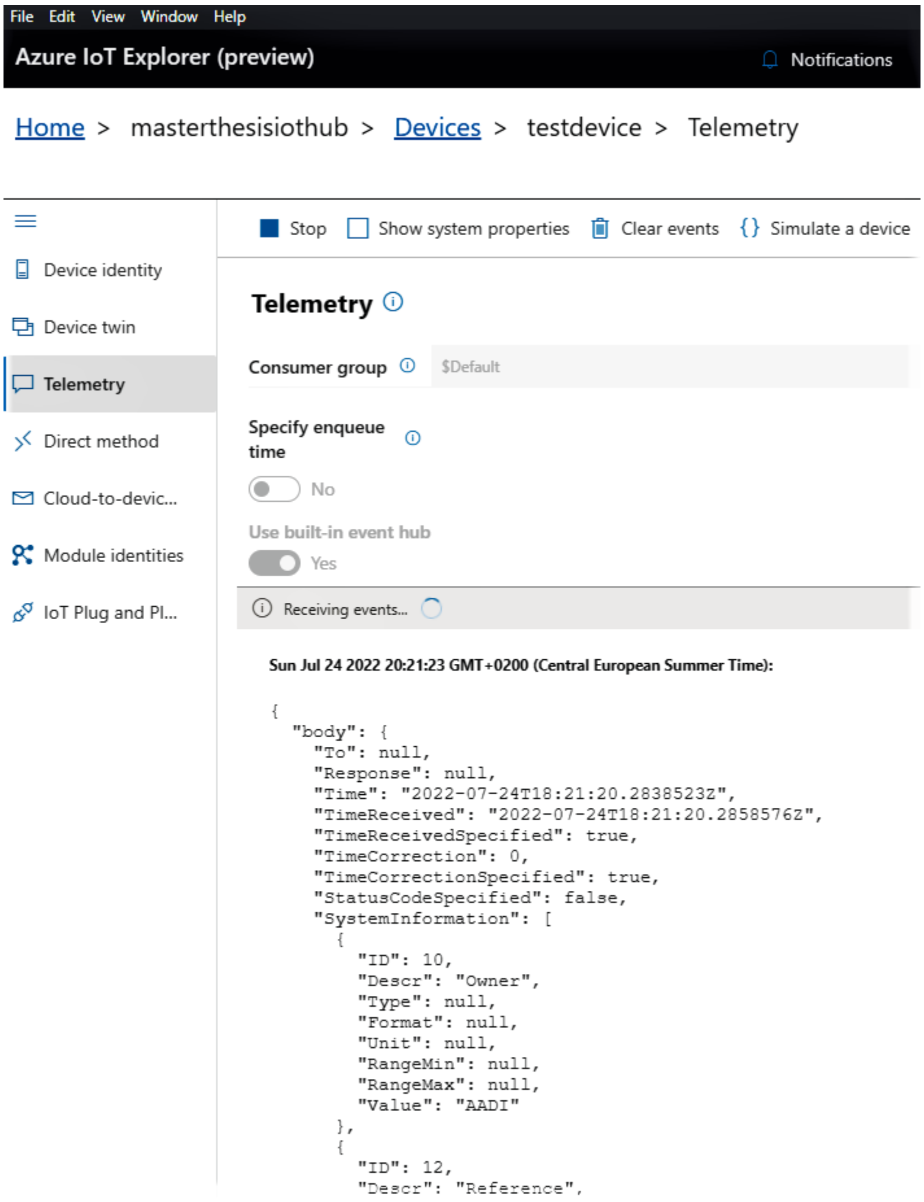
Azure IoT Hub has no direct way of displaying the received sensor messages in the portal, without routing them to other services. However, Microsoft offers a graphical tool, called the Azure IoT Explorer [143], that can be downloaded and installed on the user's computer. The tool allows us to connect with our IoT Hub and interact with the connected devices. We select our device and go to the telemetry tab. Here, we can see the sensor messages being received in real time while our Azure prototype is running. Figure 5.5 shows an image of the sensor messages being successfully integrated to Azure IoT Hub.

5.4 Prototype 2 - Amazon Web Services

In the second prototype, we develop clients for sending sensor messages to Amazon Web Services, using AWS IoT Core. Two client applications are created, one for HTTPS and one for MQTT. Since Amazon does not offer device SDK support for .NET, we need to utilize open-source MQTT libraries and create .NET methods for sending REST API posting requests with HTTPS.

5.4.1 AWS IoT Core and Device Setup

Once again, we need to do the necessary setup on the cloud platform before we can send our sensor messages. We first create an AWS account which automatically sign's us up for all services on the cloud platform, and then we log in to the AWS portal. In the search bar, we search for AWS IoT Core and enter the service. For our prototypes, we need our IoT Core service's cloud region and



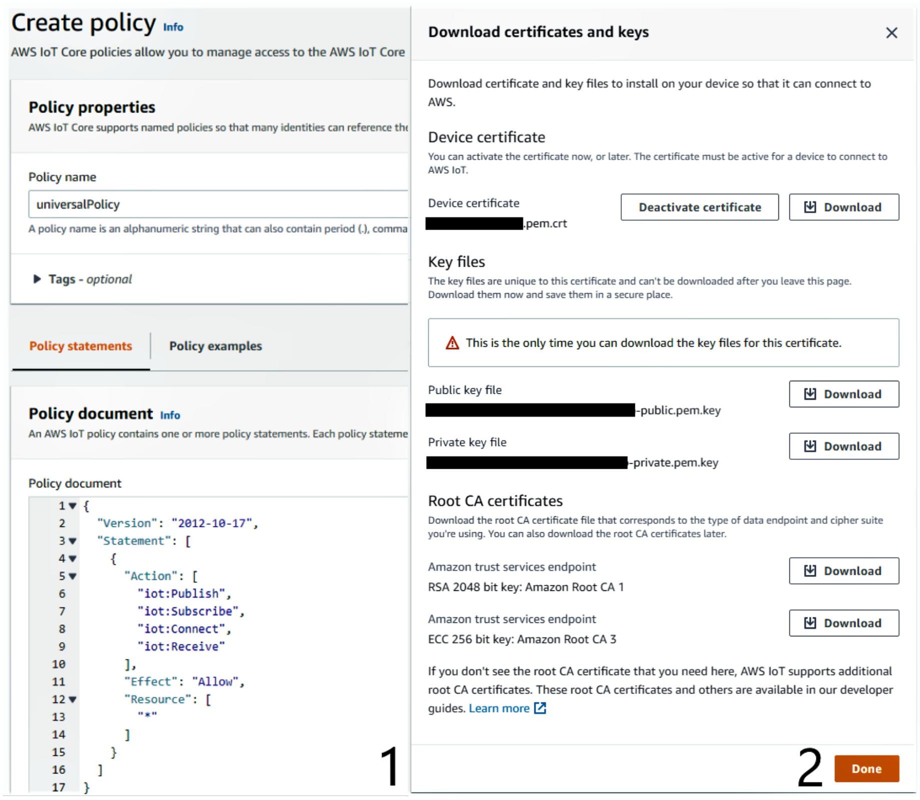
Sun Jul 24 2022 20:21:23 GMT+0200 (Central European Summer Time):

```
{
  "body": {
    "To": null,
    "Response": null,
    "Time": "2022-07-24T18:21:20.2838523Z",
    "TimeReceived": "2022-07-24T18:21:20.2858576Z",
    "TimeReceivedSpecified": true,
    "TimeCorrection": 0,
    "TimeCorrectionSpecified": true,
    "StatusCodeSpecified": false,
    "SystemInformation": [
      {
        "ID": 10,
        "Descr": "Owner",
        "Type": null,
        "Format": null,
        "Unit": null,
        "RangeMin": null,
        "RangeMax": null,
        "Value": "AADI"
      },
      {
        "ID": 12,
        "Descr": "Reference",
```

Figure 5.5: Azure Sensor Message Output

unique endpoint. In the right-top corner we choose the cloud region we want to use, and if we navigate to the settings option, we find the unique endpoint for our IoT Core service.

For device creation there are several options including registration of single devices, device groups, device types, and provisioning templates for multiple devices. For our setup we only create a single device. There are several properties that can be added to a device, but we only give the device a unique name ID and move on. Next we configure the device's authentication where users can either upload their own self-signed certificates or CA certificates, or auto-generate a certificate, public key, and private key, which is the option we choose. Lastly, we create a policy for managing the device's access. We name the policy and then add the following actions: iot:Publish, iot:Subscribe, iot:Connect, iot:Receive. When the created policy is added to the device, we then download the auto-generated certificate, public key, and private key, as well as Amazon's official CA root certificate: Amazon Root CA 1. Figure 5.6 shows the created policy and download page for the authentication files.



Create policy Info

AWS IoT Core policies allow you to manage access to the AWS IoT Core

Policy properties

AWS IoT Core supports named policies so that many identities can reference the

Policy name

universalPolicy

A policy name is an alphanumeric string that can also contain period (.), comma

Tags - *optional*

Policy statements | Policy examples

Policy document Info

An AWS IoT policy contains one or more policy statements. Each policy stateme

Policy document

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iot:Publish",
        "iot:Subscribe",
        "iot:Connect",
        "iot:Receive"
      ],
      "Effect": "Allow",
      "Resource": [
        "*"
      ]
    }
  ]
}
```

Download certificates and keys

Download certificate and key files to install on your device so that it can connect to AWS.

Device certificate

You can activate the certificate now, or later. The certificate must be active for a device to connect to AWS IoT.

Device certificate
[redacted].pem.crt
Deactivate certificate | Download

Key files

The key files are unique to this certificate and can't be downloaded after you leave this page. Download them now and save them in a secure place.

This is the only time you can download the key files for this certificate.

Public key file
[redacted]-public.pem.key
Download

Private key file
[redacted]-private.pem.key
Download

Root CA certificates

Download the root CA certificate file that corresponds to the type of data endpoint and cipher suite you're using. You can also download the root CA certificates later.

Amazon trust services endpoint
RSA 2048 bit key: Amazon Root CA 1
Download

Amazon trust services endpoint
ECC 256 bit key: Amazon Root CA 3
Download

If you don't see the root CA certificate that you need here, AWS IoT supports additional root CA certificates. These root CA certificates and others are available in our developer guides. Learn more

Done

Figure 5.6: AWS Device Policy Creation (1) and Authentication Files (2)

We add the downloaded authentication files to a separate folder. The files will have long generated names, but for simplicity we rename them to "certificate", while keeping "AmazonRootCA1" as it is. For our prototypes we need to reformat the file types due to the difficulty of formatting .PEM files in C. We manually reformat the downloaded files to the following file types:

- **Certificate:** certificate.cert.pem
- **Public key:** certificate.public.key
- **Private key:** certificate.private.key
- **Amazon root certificate:** AmazonRootCA1.crt

Then we use the reformatted files in the following OpenSSL command [238] that we execute in a terminal. This is done to chain the certificate, private key, and

Amazon root certificate into a .PFX client certificate file that our .NET clients can understand. The user also need to add a custom password for the client certificate in the command.

```
openssl pkcs12 -export -in certificate.cert.pem -inkey certificate.
    private.key -passout pass:{your certificate password} -out
    certificate.cert.pfx -certfile AmazonRootCA1.crt
```

The executed command outputs the client certificate: certificate.cert.pfx. The client certificate, its custom password, and the Amazon CA root certificate are the files we need for our prototype. We now have the required setup, and move on to the client implementations.

5.4.2 AWS Implementation with HTTPS

The HTTPS client publishes messages to a specified topic in IoT Core's message broker, using REST API posting. We define the following arguments that will be unique for each user, with the exception of the port number. They are all used for creating the client-specific endpoint and topic-specific request URL.

```
private static string clientCert = "certificate.cert.pfx";
private static string certPassword = "{your certificate password}";
private static string topicId = "{your topic name}}";
private static string iotCoreId = "{your iot core id}";
private static string cloudRegion = "{your cloud region}"
private static int httpsPort = 8443;

private static string endpoint = $"{iotCoreId}.{cloudRegion}.
    amazonaws.com";
private static string requestURL = $"https://{endpoint}:{httpsPort
    }/topics/{topicId}?qos=1";
```

The main method creates a X.509 certificate instance with the X509Certificate2 class [172], using our client certificate and password as input. Then we set up a condition for quitting the client by typing CTRL-C in the console, when running the application. Lastly, it initializes the method for sending sensor messages to the cloud platform. The method takes in our request URL, client certificate instance, and the cancellation token.

```
static void Main(string[] args){
    Console.WriteLine("Initalizing AWS HTTP client. Press CTRL-C to
         exit...");

    //Reads our client certificate and certificate password
    X509Certificate2 clientCertificate = new X509Certificate2(Path.
        Combine(AppDomain.CurrentDomain.BaseDirectory, clientCert),
         certPassword);

    //Sets up a condition to quit the client
    using var cts = new CancellationTokenSource();
    Console.CancelKeyPress += (sender, eventArgs) => {
        eventArgs.Cancel = true;
        cts.Cancel();
        Console.WriteLine("Exiting...");
    };

    SendDeviceToCloudMessages(requestURL, clientCertificate, cts.
        Token);
16 }
```

The sensor messaging method uses .NET's HttpWebRequest class [142] for creating and sending HTTPS posting requests to the AWS IoT Core. The method's structure is quite similar to Azure's sensor messaging method, with the main differences being the code between line 24-33. The method obtains the sensor messages from MSMQ, serializes them to JSON, and then byte encodes them into a UTF-8 format. For the HTTPS header, we pass in the payload's content-length and sets the content-type to application/json. We add the request URL for the URI, and also add our client certificate for the device authentication. Then we use .NET's System.IO.Stream class [162] for writing the request data, which transfers it to AWS IoT Core.

```
1  private static void SendDeviceToCloudMessages(string requestURL,
       X509Certificate2 clientCert, CancellationToken ct){
2      string[] messages;
3      string[] jsonMessage;
4      int messageCounter = 0;
5      int counter;
6      HttpWebRequest request;
7
8      while (!ct.IsCancellationRequested){
9          counter = 0;
10         messages = MessageHandling.ReadMessagesFromClient();
11         jsonMessage = new string[messages.Length];
12         if (messages.Length == 0 || messages == null){
13             Console.WriteLine("There is no messages in the queue");
14             Thread.Sleep(5000);
15         }else{
16             foreach (string sensorMessage in messages){
17                 // Create JSON message
18                 jsonMessage[counter] = MessageHandling.
                       SerializeToJson(sensorMessage);
19                 byte[] byteArray = Encoding.UTF8.GetBytes(
                       jsonMessage[counter]);
20
21                 counter++;
22                 messageCounter++;
23
24                 request = (HttpWebRequest) WebRequest.Create(
                       requestURL);
25                 request.Method = "POST";
26                 request.ContentLength = byteArray.Length;
27                 request.ContentType = "application/json";
28                 request.KeepAlive = true;
29                 request.ClientCertificates.Add(clientCert);
30
31                 Stream dataStream = request.GetRequestStream();
32                 dataStream.Write(byteArray, 0, byteArray.Length);
33                 dataStream.Close();
34
35                 Console.WriteLine($"{DateTime.Now} > Message {
                       messageCounter} sent");
36                 Thread.Sleep(5000);
37             }
38         }
39     }
40 }
```

5.4.3 AWS Implementation with MQTT

For the MQTT implementation, we are using the M2Mqtt [186] open-source library for developing our MQTT client in .NET. In order to successfully connect with IoT Core, the following arguments are required. Both the Amazon root certificate and the client certificate with its password is needed for authentication. Additionally, we need our device ID, our topic name, our IoT Core endpoint, and the MQTT port number which is 8883.

```
private static string caCert = "AmazonRootCA1.crt";
private static string clientCert = "certificate.cert.pfx";
private static string certPassword = "{your certificate password}";

private static string deviceId = "{your device name}";
private static string topic = "{your topic name}";
private static string iotCoreId = "{your iot core id}";
private static string cloudRegion = "{your cloud region}"
private static int mqttPort = 8883;

private static string endpoint = $"{iotCoreId}.{cloudRegion}.amazonaws.com";
```

For the main method, we first pass in both the Amazon root certificate, and our client certificate in order create an X.509 certificate instance of each of them. On line 11, we create our MQTT client using the MqttClient class from the M2Mqtt library, and pass in the required arguments and certificate instances. Next, we create our standard cancellation token for manually quitting the client. We connect our MQTT client AWS IoT Core by passing in our device ID as a parameter. Lastly, we initiate our sensor messaging method.

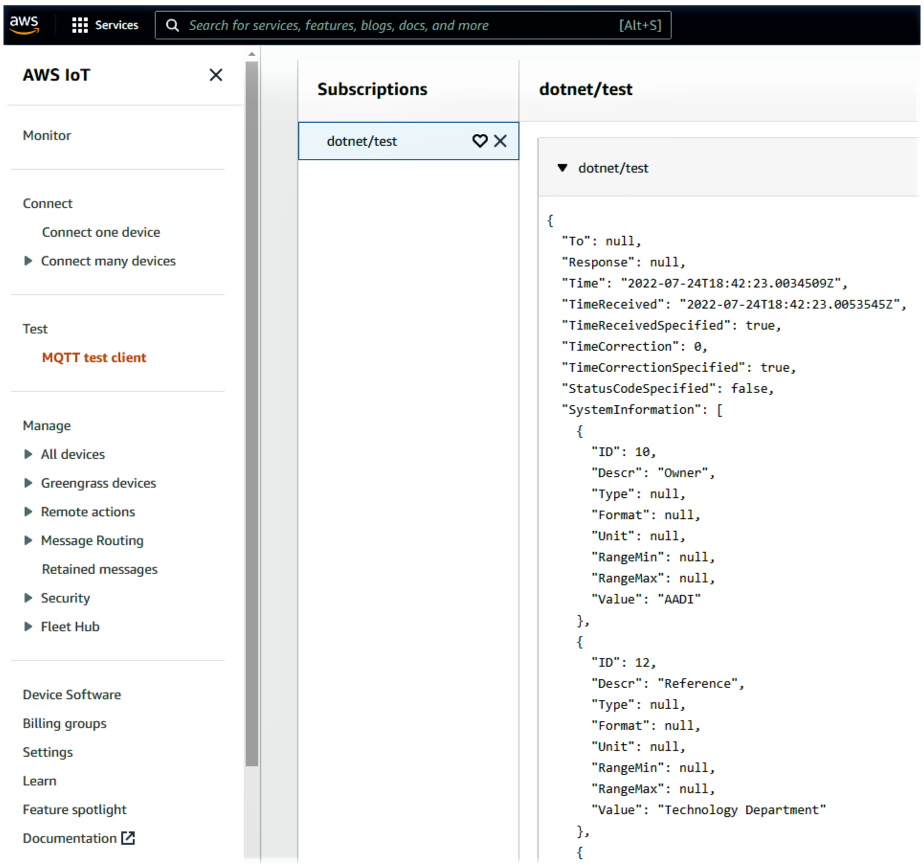
```
1  static void Main(string[] args){
2      Console.WriteLine("Initializing AWS IoT MQTT client. Press CTRL-C to exit...");
3  
4      //Reads the CA certificate
5      X509Certificate caCertificate = X509Certificate.CreateFromCertFile(Path.Combine(AppDomain.CurrentDomain.BaseDirectory, caCert));
6  
7      //Reads our client certificate and certificate password
8      X509Certificate2 clientCertificate = new X509Certificate2(Path.Combine(AppDomain.CurrentDomain.BaseDirectory, clientCert), certPassword);
9  
10     //Creates the MQTT client by passing in our arguments
11     var client = new MqttClient(endpoint, mqttPort, true, caCertificate, clientCertificate, MqttSslProtocols.TLSv1_2);
12 
13     //Sets up a condition to quit the client
14     using var cts = new CancellationTokenSource();
15     Console.CancelKeyPress += (sender, eventArgs) => {
16         eventArgs.Cancel = true;
17         cts.Cancel();
18         Console.WriteLine("Exiting...");
19     };
20 
21     //Connects the MQTT client our device in the cloud
22     client.Connect(deviceId);
23 
24      //The method for sending data to the cloud
25      SendDeviceToCloudMessages(cts.Token, client, topic);
26  }
```

The sensor messaging method collects the currently available messages from MSMQ, serializes the messages to JSON, and reformat the payload into bytes with UTF-8 encoding. On line 24, the MQTT client takes in our topic name and payload as parameters, and then publishes each message to the specified topic in IoT Core's message broker.

```
1   private static void SendDeviceToCloudMessages(CancellationToken cts
        , MqttClient mqttClient, string topic){
2       string[] messages;
3       string[] jsonMessage;
4       int messageCounter = 0;
5       int counter;
6
7       while (!cts.IsCancellationRequested){
8           counter = 0;
9           messages = MessageHandling.ReadMessagesFromClient();
10          jsonMessage = new string[messages.Length];
11          if (messages.Length == 0 || messages == null){
12              Console.WriteLine("There is no messages in the queue");
13              Thread.Sleep(5000);
14          }else{
15              foreach (string sensorMessage in messages){
16                  // Create JSON message
17                  jsonMessage[counter] = MessageHandling.
                        SerializeToJson(sensorMessage);
18                  byte[] byteArray = Encoding.UTF8.GetBytes(
                        jsonMessage[counter]);
19
20                  counter++;
21                  messageCounter++;
22
23                  // Send the telemetry message
24                  mqttClient.Publish(topic, byteArray);
25                  Console.WriteLine($"{DateTime.Now} > Message " +
                        messageCounter + " sent");
26                  Thread.Sleep(5000);
27              }
28          }
29      }
30  }
```

5.4.4 AWS Deployment and Output

The prototype deployment is done in the same way as the prototype deployment for Microsoft Azure. The DS and RTC applications must be actively running in the background. In Visual Studio, we choose the client interface projects and then either the HTTPS client or MQTT client to run on startup. We then run the chosen projects and get up the client interface console and the prototype client console. In client interface console we specify that both the RTC and client applications are running on the same machine. If the client interface successfully connected to the RTC, it will start receiving virtual sensor messages and forward them to the MSMQ. If the prototype client application successfully established connection with AWS IoT Core, the console then start printing messages in the console each time a new sensor message was sent to the cloud. The deployment is exactly the same as shown in figure 5.4 with Azure. Next, we head over to our IoT Core in the AWS portal and navigate to the MQTT test client. There we enter our topic name, which in our case were "dotnet/test", and click on the subscribe button. Our published messages will then be displayed as shown in figure 5.7.



```
{
  "To": null,
  "Response": null,
  "Time": "2022-07-24T18:42:23.0034509Z",
  "TimeReceived": "2022-07-24T18:42:23.0053545Z",
  "TimeReceivedSpecified": true,
  "TimeCorrection": 0,
  "TimeCorrectionSpecified": true,
  "StatusCodeSpecified": false,
  "SystemInformation": [
    {
      "ID": 10,
      "Descr": "Owner",
      "Type": null,
      "Format": null,
      "Unit": null,
      "RangeMin": null,
      "RangeMax": null,
      "Value": "AADI"
    },
    {
      "ID": 12,
      "Descr": "Reference",
      "Type": null,
      "Format": null,
      "Unit": null,
      "RangeMin": null,
      "RangeMax": null,
      "Value": "Technology Department"
    },
    {
```

Figure 5.7: AWS Sensor Message Payload Output

5.5 Prototype 3 - Google Cloud Platform

The third and final prototype is sensor data integration with Google Cloud Platform. The prototype consists of a HTTPS client and a MQTT client for sending sensor messages to GCP IoT Core. Google does not offer device SDK support for .NET, meaning that we again need to utilize open-source libraries for MQTT and REST API requests with HTTPS. The implementation approach is quite similar to the Amazon Web Services prototype, with some exceptions.

5.5.1 GCP IoT Core and Device Setup

For our setup, we must first create a Google Cloud Platform account and once logged in, we create a project with a proper name. From here, we can start setting up the necessary services. IoT Core [75] is the main service that we are going to use, but Google's Pub/Sub [77] service is also needed for creating topics and subscriptions in order to have an endpoint where the sensor messages can be published. In order to use these services, we first need to enable two API's in our account: The Cloud IoT API [73] and the Cloud Pub/Sub API [33].

We start by creating a topic and subscription on the Pub/Sub service. There are two types of topics: event-topics and state-topics. The event-topic is used for aggregation of the received sensor messages, routed from IoT Core. The state-topics are optional topics that can be created for monitoring the state of each created device, which usually involves configuration updates or reporting of device changes. The event-topic is the only type we are using for our prototype. After the topics are created, we create a subscription that we bind to our event-topic. The subscription is used for directing the sensor messages to different subscribers. The messages can either be pushed or pulled as needed.

For the IoT Core configuration, we must first create a registry. In the registry creation, we bind it to our created topic and select both the HTTP and MQTT protocol. When the registry creation is complete, we can start creating devices and add them to the registry. In order to send data to GCP IoT Core, we need to generate a device key pair to authenticate each created device. This is done by executing the following OpenSSL command in the Cloud Shell terminal [74].

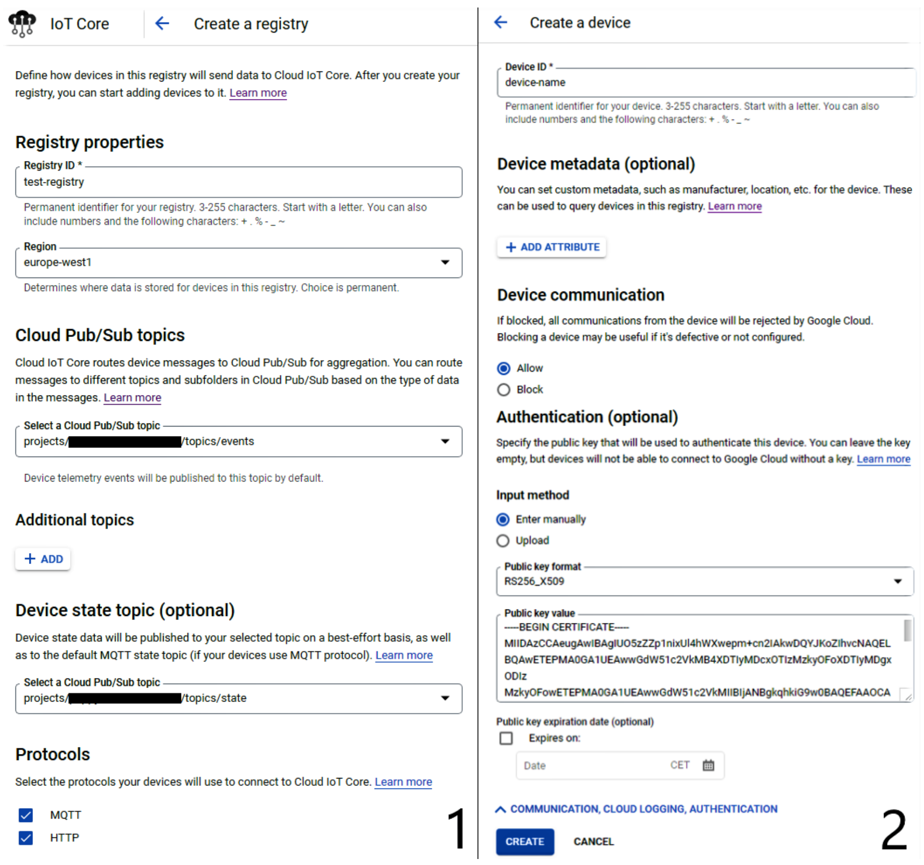
```
openssl req -x509 -newkey rsa:2048 -keyout rsa_private.pem -nodes -
    out \ rsa_cert.pem -subj "/CN=unused"
```

When the command is executed, we get an output with two .PEM files. A client certificate file: rsa_cert.pem, and a private key: rsa_private.pem. We also need to download Google's official certificate authority root file which can be found here: `https://pki.goog/roots.pem`. The content of the rsa_cert.pem file needs to be copied and added to the associated device in our IoT Core registry as a public key in the RS256_X509 format, as shown in figure 5.8. We now have the required GCP setup for sending messages to IoT Core. Next, we go over the .NET implementation for the HTTPS client and the MQTT client.

5.5.2 GCP data integration with HTTPS

The HTTPS client provides the code needed for publishing sensor messages over Google's HTTP bridge [78]. The HTTP bridge requires the use of the cloudiot-device REST resource [80] from the Cloud IoT API in order to enable device communication on the service. The cloudiotdevice REST resource provides a method called publishEvent [76] which is the method we need to use for publishing messages to GCP IoT Core. Below we have defined the necessary arguments for the client. Most of them are used for creating the publishEvent method's specific request URL.

```
private static string certPrivateKey = "rsa_private.pem";
private static string httpBridgeHostName = "cloudiotdevice.
    googleapis.com";
```


IoT Core ← Create a registry

Define how devices in this registry will send data to Cloud IoT Core. After you create your registry, you can start adding devices to it. Learn more

Registry properties

Registry ID *
test-registry
Permanent identifier for your registry. 3-255 characters. Start with a letter. You can also include numbers and the following characters: + . % - _ ~

Region
europe-west1
Determines where data is stored for devices in this registry. Choice is permanent.

Cloud Pub/Sub topics

Cloud IoT Core routes device messages to Cloud Pub/Sub for aggregation. You can route messages to different topics and subfolders in Cloud Pub/Sub based on the type of data in the messages. Learn more

Select a Cloud Pub/Sub topic
projects/[redacted]/topics/events
Device telemetry events will be published to this topic by default.

Additional topics

+ ADD

Device state topic (optional)

Device state data will be published to your selected topic on a best-effort basis, as well as to the default MQTT state topic (if your devices use MQTT protocol). Learn more

Select a Cloud Pub/Sub topic
projects/[redacted]/topics/state

Protocols

Select the protocols your devices will use to connect to Cloud IoT Core. Learn more

☑ MQTT
☑ HTTP

1

← Create a device

Device ID *
device-name
Permanent identifier for your device. 3-255 characters. Start with a letter. You can also include numbers and the following characters: + . % - _ ~

Device metadata (optional)

You can set custom metadata, such as manufacturer, location, etc. for the device. These can be used to query devices in this registry. Learn more

+ ADD ATTRIBUTE

Device communication

If blocked, all communications from the device will be rejected by Google Cloud. Blocking a device may be useful if it's defective or not configured.

● Allow
○ Block

Authentication (optional)

Specify the public key that will be used to authenticate this device. You can leave the key empty, but devices will not be able to connect to Google Cloud without a key. Learn more

Input method

● Enter manually
○ Upload

Public key format
RS256_X509

Public key value
```
-----BEGIN CERTIFICATE-----
MIIDAzCCAeugAwIBAgIUO5zZZp1nixUl4hWXwepm+cn2IAkwDQYJKoZIhvcNAQEL
BQAwETEPMA0GA1UEAwwGdW51c2VkMB4XDTIyMDcxOTIzMzkyOFoXDTIyMDgx
ODIz
MzkyOFowETEPMA0GA1UEAwwGdW51c2VkMIIBIjANBgkqhkiG9w0BAQEFAAOCA
```

Public key expiration date (optional)
☐ Expires on:
Date CET

^ COMMUNICATION, CLOUD LOGGING, AUTHENTICATION

CREATE CANCEL

2

Figure 5.8: GCP Registry Creation (1) and Device Creation (2)

```
private static string projectId = "{your project name}";
private static string cloudRegion = "{your iot core region}";
private static string registryId = "{your registry name}";
private static string deviceId = "{your device name}";
private static string topicId = "{your topic name}";

private string requestURL = $"https://{httpBridgeHostName}/v1/
    projects/{projectId}/locations/{cloudRegion}/registries/{
    registryId}/devices/{deviceId}:publishEvent";
```

The main method initializes a condition for quiting the client using CTRL-C, and the method for sending the sensor messages to GCP IoT Core. The method takes in the request URL, cancellation token, as well as a created JSON Web Token [81] which we will be showing next.

```
public static void Main(string[] args){
    Console.WriteLine("Initalizing GCP HTTP client. Press CTRL-C to
         exit...");

    //Sets up a condition to quit the client
    using var cts = new CancellationTokenSource();
    Console.CancelKeyPress += (sender, eventArgs) =>
    {
        eventArgs.Cancel = true;
        cts.Cancel();
        Console.WriteLine("Exiting...");
    };

    SendDeviceToCloudMessages(requestURL, JwtToken(certPrivateKey),
        cts.Token);
}
```

Unlike the clients developed for Amazon's prototype, each device must also prepare a JSON Web Token (JWT). JWT's are used for enabling time-limited device authentication for connecting to GCP IoT Core, and are mandatory for both HTTPS and MQTT. For HTTPS, the JWT must be included in the header of each HTTPS request. For MQTT, the JWT must be passed in the password field of the CONNECT message. The creation of a JWT for GCP IoT Core requires a JWT Web Signature [99] and a JWT header [100].

The JWT Web Signature must contain a secret key, which in our case is the generated certificate private key: rsa_private.pem, in combination with a set of the three following claims.

- **iat ("Issued At"):** This claim contains the exact timestamp when the JWT was created. The time must be specified in Unix seconds elapsed since the Unix epoch which is 00:00:00 UTC on January 1, 1970.
- **exp ("Expiration"):** This is the expiration timestamp when the JWT is no longer valid. It is created using the iat timestamp + the number of chosen seconds the JWT is valid. A JWT can only be valid for maximum 24 hours (86400 seconds).
- **aud ("Audience"):** This claim uses is a single string that contains our project ID where the devices are registered.

Next, the JWT header must include two fields, one field for the token type, and a digital signature algorithm for the second field. The token type is "JWT", while the digital signature algorithm can be either a Rivest-Shamir-Adleman (RSA) key [94, 98], or a Elliptic Curve key [102, 97] using SHA-256. In our case, we use an RSA key.

Following the method from line 3-19, we read the content of our private key using Bouncy Castle's PemReader class [27], and then use the RSAParemeters struct [161] for structuring the private key into a raw RSA key form. Then on line 22, we perform RSA encryption on the private key, using the RSACryptoServiceProvider class [160]. From line 24-34, we create the three required claims for the JWT web signature. Then from line 36-38, we create the JWT itself using the JavaScript Object Signing and Encryption (JOSE) [262] library for .NET. We pass in the developed claims, RSA key, and SHA-256 algorithm, and encode it to the the final JWT string we need. The exact same method is used for both the HTTPS client and the MQTT client.

```
1  private static string JwtToken(string certPrivateKey){
2
3      string privateKey = File.ReadAllText(certPrivateKey);
4
5      RSAParameters rsaParams;
```

```

    if (privateKey.IndexOf("-----BEGIN PRIVATE KEY-----") < 0){
        throw new NotSupportedException("Invalid private key was
            used.");
    }

    // Reads the private key file.
    using (var sr = new StringReader(privateKey)){
        var pemReader = new PemReader(sr);
        var KeyParameter = (AsymmetricKeyParameter) pemReader.
            ReadObject();
        var privateRsaParams = KeyParameter as
            RsaPrivateCrtKeyParameters;
        rsaParams = DotNetUtilities.ToRSAParameters(
            privateRsaParams);
        pemReader.Reader.Close();
        sr.Close();
    }

    //RSA encrypt private key
    RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
    rsa.ImportParameters(rsaParams);

    DateTime dtnow = DateTime.Now;
    long iat = ((DateTimeOffset)dtnow).ToUnixTimeSeconds(); //Unix
    long exp = iat + 3600; //Token expires in 3600 seconds

    //Creates the the three required claim fields.
    var claims = new Dictionary<string, object>(){
        {"iat", iat} //("Issued At")
       ,{"exp", exp} //("Expiration")
       ,{"aud", projectId} //("Audience")
    };

    string token = JWT.Encode(claims //The required claims
        , rsa //RSA privateKey
        , JwsAlgorithm.RS256); //Signature with SHA-256 algorithm

    return token;
}
```

We once again utilize the .NET HttpWebRequest class [142] for sending REST API posting requests to GCP IoT Core. In order to successfully send sensor data via the HTTP bridge, we need to strictly follow the requirements of the API's publishEvent method [76]. We have already defined the method's specified request URL, but the request body must contain the three following fields: a subFolder-string, a binaryData-string, and a gatewayInfo-object. In our case, we only need the subFolder and binaryData, since our device is not binded to any gateway in GCP IoT Core. The subFolder must contain our topic ID, while the binaryData contains our payload data which needs to be formatted in the base64-encoded binary format. After the information is added to the fields, the request body needs to be formatted into a JSON structure, and then formatted into bytes using UTF-8 encoding. We pass the request body's content-length in the header, as well as the content-type which is application/json. Lastly, we need to pass in our JWT in the header, specified in the following name-value pair form: "Authorization": "Bearer " + JWT. The final created request is then written to the dataStream method for transmission over GCP IoT Core's HTTP bridge.

```
private static void SendDeviceToCloudMessages(string requestURL,
    string token, CancellationToken cts){
    string[] messages;
    string[] jsonMessage;
    int messageCounter = 0;
    int counter;
    HttpWebRequest request;
    Payload payload;

    while (!cts.IsCancellationRequested){
        counter = 0;
        messages = MessageHandling.ReadMessagesFromClient();
        jsonMessage = new string[messages.Length];
        if (messages.Length == 0 || messages == null){
            Console.WriteLine("There is no messages in the queue");
            Thread.Sleep(5000);
        }else{
            foreach (string sensorMessage in messages){
                //Create JSON message
                jsonMessage[counter] = MessageHandling.
                    SerializeToJson(sensorMessage);

                //Encodes message to base64-string
                string base64data = Convert.ToBase64String(Encoding
                    .UTF8.GetBytes(jsonMessage[counter]));

                counter++;
                messageCounter++;

                //Creates the required request body
                payload = new Payload{
                    binaryData = base64data,
                    subFolder = topicId
                };
                string requestBody = JsonConvert.SerializeObject(
                    payload);
                byte[] byteArray = Encoding.UTF8.GetBytes(
                    requestBody);

                //Add the necessary settings to the request
                request = (HttpWebRequest)WebRequest.Create(
                    requestURL);
                request.Method = "POST";
                request.ContentLength = byteArray.Length;
                request.ContentType = "application/json";
                request.KeepAlive = true;
                request.Headers.Add("Authorization", "Bearer " +
                    token);

                // Publishing telementry data
                Stream dataStream = request.GetRequestStream();
                dataStream.Write(byteArray, 0, byteArray.Length);
                dataStream.Close();

                Console.WriteLine($"{DateTime.Now} > Publishing
                    Message " + messageCounter);
                Thread.Sleep(5000);
            }
        }
    }
}
```

5.5.3 GCP Implementation with MQTT

The MQTT client publishes sensor messages to GCP IoT Core over Google's MQTT bridge [79]. GCP IoT Core's broker listens over the 8883 port, using mqtt.googleapis.com as the the host name. It also requires the use of Google's official CA root file for authentication, and our private key for the JWT creation. The rest of the arguments is needed for creating the MQTT bridge's required client ID path and topic path.

```
private static string caCert = "roots.pem";
private static string certPrivateKey = "rsa_private.pem";
private static string mqttBridgeHostName = "mqtt.googleapis.com";
private static int mqttPort = 8883;

private static string projectId = "{your project name}";
private static string cloudRegion = "{your iot core region}";
private static string registryId = "{your registry name}";
private static string deviceId = "{your device name}";
private static string topicId = "{topic name}";

private static string clientId = $"projects/{projectId}/locations/{
    cloudRegion}/registries/{registryId}/devices/{deviceId}";
private static string deviceTopic = $"/devices/{deviceId}/{topicId}
    ";
```

Just like with Amazon's MQTT client, we again use the M2Mqtt [186] open-source library for developing our MQTT client in .NET. From line 5-14 we read the content of Google's CA root .PEM file, reformatting it to base64 binary, and stores it in a byte array. This is done so we can pass in the CA root file as parameter in the X509Certificate class [171]. Next we create the MQTT client using our newly created certificate instance, the MQTT bridge's supported host name and port, and TLS v1.2 for initiating the security protocol handshake. We also set up the usual cancellation token for manually quitting the client. Then we connect our MQTT client to IoT Core's MQTT bridge, by passing in our defined client ID path and created JWT, using the exact same JWT method that was introduced in the HTTP implementation. Finally, we initialize our sensor messaging method.

```
1  static void Main(string[] args){
2      Console.WriteLine("Initalizing GCP MQTT client. Press CTRL-C to
           exit...");
3
4      //Reads the content of the CA root file and formats the .PEM
           file to a base64 format, stored in a byte array
5      byte[] data;
6      using (FileStream fs = File.OpenRead(Path.GetDirectoryName(
           Assembly.GetEntryAssembly().Location) + "\\" + caCert)){
7          data = new byte[fs.Length];
8          fs.Read(data, 0, data.Length);
9          if (data[0] != 0x30){
10             //Using the PEM method on line 45-53
11             data = PEM("CERTIFICATE", data);
12         }
13         fs.Close();
14     }
15
16     //Creates X.509 certificate instance of the reformatted CA root
17     X509Certificate clientCertificate = new X509Certificate(data);
18
19      //Crates MQTT client using our certificate and arguments
20      MqttClient client = new MqttClient(mqttBridgeHostName, mqttPort
            , true, null, clientCertificate, MqttSslProtocols.TLSv1_2);
21
22      //Sets up a condition to quit the client
23      using var cts = new CancellationTokenSource();
24      Console.CancelKeyPress += (sender, eventArgs) => {
25          client.Disconnect();
26          eventArgs.Cancel = true;
27          cts.Cancel();
28          Console.WriteLine("Exiting...");
29      };
30
31      //Connects MQTT client to IoT Core with our client ID, username
             as null, and JWT for authentication
32      client.Connect(clientId, null, JwtToken(cert));
33
34      //Method for sending sensor messages to GCP IoT Core
35      SendDeviceToCloudMessages(client, deviceTopic, cts.Token);
36  }
37
38  //Method for base64 encoding Google's CA root file (roots.pem)
39  private static byte[] PEM(string type, byte[] data){
40      string pem = Encoding.ASCII.GetString(data);
41      string header = String.Format("-----BEGIN {0}-----\n", type);
42      string footer = String.Format("\n-----END {0}-----", type);
43      int start = pem.IndexOf(header) + header.Length;
44      int end = pem.IndexOf(footer, start);
45      string base64 = pem.Substring(start, (end - start));
46      return Convert.FromBase64String(base64);
47  }
```

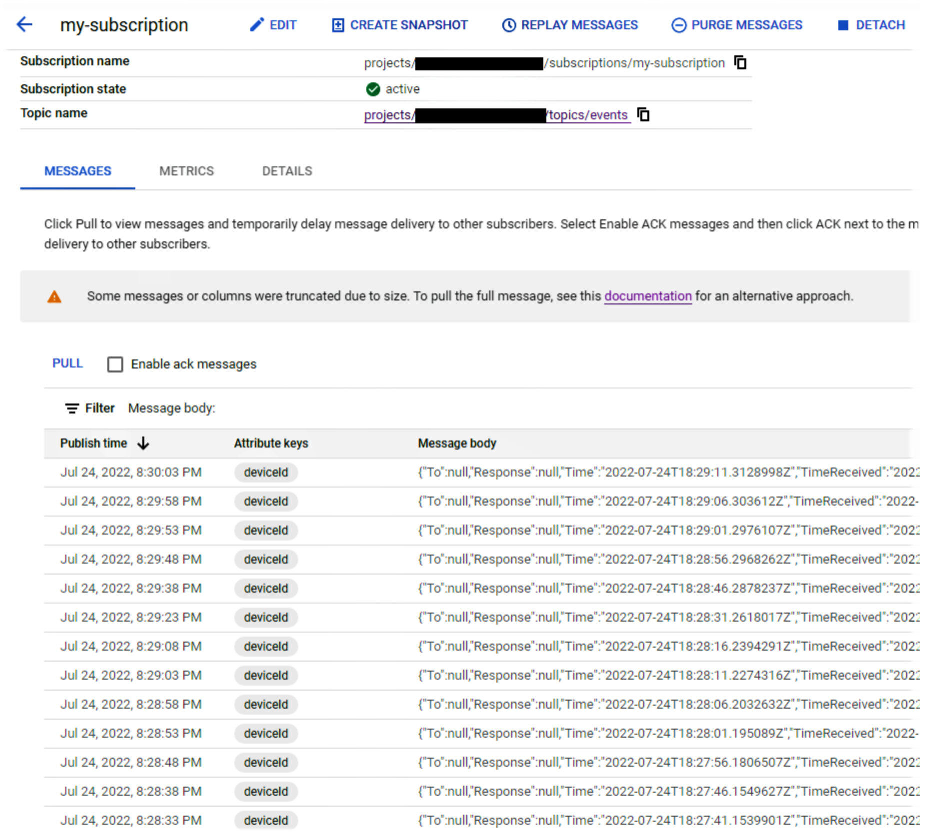
The sensor messaging method collects the currently available sensor messages from MSMQ, serialize them to JSON, and then formats the JSON message into bytes using UTF-8 encoding. On line 24, the MQTT client takes in our topic-path string, payload message, and declare that we use QoS level 1. Then each sensor message gets sent to IoT Core and published to our topic in Cloud Pub/Sub.

```
1   private static void SendDeviceToCloudMessages(MqttClient client,
        string deviceTopic, CancellationToken cts){
2       string[] messages;
3       string[] jsonMessage;
4       int messageCounter = 0;
5       int counter;
6
7       while (client.IsConnected && !cts.IsCancellationRequested){
8           counter = 0;
9           messages = MessageHandling.ReadMessagesFromClient();
10          jsonMessage = new string[messages.Length];
11          if (messages.Length == 0 || messages == null){
12              Console.WriteLine("There is no messages in the queue");
13              Thread.Sleep(5000);
14          }else{
15              foreach (string sensorMessage in messages){
16                  //Create JSON message
17                  jsonMessage[counter] = MessageHandling.
                        SerializeToJson(sensorMessage);
18                  byte[] bMessage = Encoding.UTF8.GetBytes(
                        jsonMessage[counter]);
```

```

                    messageCounter++;
                    counter++;

                    //Send the telemetry message
                    client.Publish(deviceTopic, bMessage, MqttMsgBase.
                        QOS_LEVEL_AT_LEAST_ONCE, false);
                    Thread.Sleep(5000);
                }
            }
        }
}
```

5.5.4 GCP Deployment and Output

Similar to the deployment of the prototypes for Azure and Amazon, we choose the client interface and then either the HTTPS client or MQTT client to run on startup in Visual Studio. AADI's DS and RTC applications must be running in the background as well. We initialize the console applications, write the required information in the client interface and if everything was successful, the clients writes in the console each time a new message was sent to IoT Core. The process is exactly the same as shown in figure 5.4 with Azure. While the clients are running, we head over to the GCP portal and navigate to our created subscription in the Pub/Sub service, and pull the received sensor messages. We then get all the current messages displayed as shown in figure 5.9.



my-subscription

Subscription name	projects/[redacted]/subscriptions/my-subscription
Subscription state	active
Topic name	projects/[redacted]/topics/events

Click Pull to view messages and temporarily delay message delivery to other subscribers. Select Enable ACK messages and then click ACK next to the m
delivery to other subscribers.

Some messages or columns were truncated due to size. To pull the full message, see this documentation for an alternative approach.

Message body:

Publish time	Attribute keys	Message body
Jul 24, 2022, 8:30:03 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:29:11.3128998Z","TimeReceived":"2022
Jul 24, 2022, 8:29:58 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:29:06.303612Z","TimeReceived":"2022-
Jul 24, 2022, 8:29:53 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:29:01.2976107Z","TimeReceived":"2022
Jul 24, 2022, 8:29:48 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:28:56.2968262Z","TimeReceived":"2022
Jul 24, 2022, 8:29:38 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:28:46.2878237Z","TimeReceived":"2022
Jul 24, 2022, 8:29:23 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:28:31.2618017Z","TimeReceived":"2022
Jul 24, 2022, 8:29:08 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:28:16.2394291Z","TimeReceived":"2022
Jul 24, 2022, 8:29:03 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:28:11.2274316Z","TimeReceived":"2022
Jul 24, 2022, 8:28:58 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:28:06.2032632Z","TimeReceived":"2022
Jul 24, 2022, 8:28:53 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:28:01.195089Z","TimeReceived":"2022-
Jul 24, 2022, 8:28:48 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:27:56.1806507Z","TimeReceived":"2022
Jul 24, 2022, 8:28:38 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:27:46.1549627Z","TimeReceived":"2022
Jul 24, 2022, 8:28:33 PM	deviceId	{"To":null,"Response":null,"Time":"2022-07-24T18:27:41.1539901Z","TimeReceived":"2022

Figure 5.9: GCP Sensor Message Output

Chapter 6

Evaluation

In this chapter, we are evaluating the three cloud platforms based on a defined set of evaluation criteria, focusing on their sensor data integration aspects.

6.1 Evaluation Criteria

Before we evaluate the cloud platforms, we define the chosen set of criteria for this thesis. The defined criteria are inspired by existing functional, non-functional, and architectural requirements for middleware technology and IoT platforms [6, 193, 52]. Some of these requirements are combined together or excluded due to irrelevance, in order to better suit them to the sensor data integration aspects we are focusing on. The final criteria are based around qualitative principles.

6.1.1 Interoperability

The cloud platforms should be able to communicate and exchange messages from various heterogeneous IoT devices and applications in an interoperable manner. The criterion evolves around how many of the interoperability levels defined in chapter 2.4.1, are supported by each cloud platform. Technical interoperability involves supported communication technologies for exchanging messages over the network. Syntactic interoperability consists of formatting and encoding technologies for interpreting and retrieving information. Lastly, semantic interoperability provides contextual data in order to ensure common understandings of the exchanged information between endpoints.

6.1.2 IoT Support

For this criterion, we want to find out the amount of IoT support each cloud platform offers. This is done to unravel how much each vendor prioritizes IoT on their platform, which will determine how applicable the cloud platforms are for sensor data integration purposes. The criteria focus on the amount of IoT-specific cloud services available on each platform, and their combined versatility of capabilities. Other relevant general-purpose cloud services are excluded.

6.1.3 Device Connectivity

This criterion focuses on how IoT devices are connected and managed in each cloud platform. This includes aspects such as device provisioning, security, and gateway utility. Device provisioning consists of registering each physical IoT device in the cloud, giving them unique identification and authentication. Security involves aspects such as rule-based authorization, identity and access management (IAM), and encryption of payload data. Gateway utility concerns the different type of gateways that IoT devices are able to connect with.

6.1.4 Data Management

In this context, data management refers to how each cloud platform manages data received from IoT devices. This includes how the data is ingested, processed, stored, and visualized, using their various services on offering. Data ingestion involves how flows of data are queried and routed from one or more sources to an endpoint. Data processing is typically done either through real-time stream processing or longer batch processes. For storage, the platforms usually provide different types of databases, such as relational SQL databases and NoSQL databases. Lastly, visualization services can query data from storage spaces and output streams, and graphically view the retrieved data for analysis and monitoring.

6.1.5 Ease of Implementation

This criterion evolves around how easy it is to implement the necessary code for sending IoT data to each cloud platform. This includes the amount of code that needs to be written by the developer, the extent of provided documentation and software tool support to simplify development, and the platform prerequisites needed to be set up in advance before data can be sent to each platform.

6.2 Cloud Platform Evaluations

In this section, we evaluate the three cloud platforms based on the evaluation criteria defined in section 6.1. Each cloud platform is evaluated separately.

6.2.1 Microsoft Azure Evaluation

Interoperability
Microsoft Azure supports several communication protocols, including REST API posting via HTTPS 1.1, and brokered messaging via MQTT 3.1.1 and AMQP 1.0 where both protocols also enable messaging over WebSocket. Azure also provides SDKs and APIs for multiple languages for integration between applications and systems, ensuring technical interoperability on the platform. When it comes to syntactic interoperability, the platform supports several serialization formats for encoding data, with the most favorable ones being JSON and CSV for IoT message exchanging. Azure provides SQL-like query languages and APIs that can be used for retrieving desired data, depending on the services in use. Unfortunately, Microsoft Azure does not have any direct support for semantic interoperability, as the responsibility is left to the developers to

define the information model for how the applications and cloud services are going interpret the exchanged data and metadata. As a result, Microsoft Azure only supports technical and syntactic interoperability.

IoT Support
After investigating Microsoft Azure's official website, we find nine IoT-specific cloud services for Azure. Azure IoT Hub [123] is the main IoT service that acts as the cloud gateway for securely connecting and managing billions of IoT devices. The service also provides device provisioning, ingestion of data, and digital twins. Azure IoT Hub Device Provisioning Service (DPS) [122] is a helper service for facilitating scalable and automatic device provisioning in the cloud. Azure Digital Twins [119] provides graph modelling of digital twins for contextualizing them in virtual environments. Azure IoT Central [120] enables a high-level environment for simplifying development of IoT solutions. It provides web application user interfaces that are integrated with IoT Hub and the other connected PaaS services for facilitating monitoring, management, template creation of devices. Azure IoT Edge [121] provide the edge computing capabilities on the platform, including container modules for enabling business logic on edge devices, an edge runtime environment for running and managing the edge devices and modules, and a cloud-interface for monitoring the edge devices. Microsoft Defender for IoT [153] enables a security environment across the user's IoT infrastructure that provides unified detection and protection of device threats. Azure RTOS [130] and Azure Sphere [131] provides different device software that can run on different developer boards and microcontrollers. Finally, Azure Percept [128] is a platform of different hardware, software, and service components for that simplifies development of Azure AI models and technologies on the edge, with options for integrating them to other IoT services.

Device Connectivity
In Microsoft Azure, IoT devices can connect to cloud gateways either directly or indirectly [151]. Direct gateway connections involve IP capable IoT devices that are using the platform's supported communication protocols. Indirect gateway connections involves either edge gateways, custom gateways, or both. Edge gateways involves devices that connect via Azure IoT Edge, while custom gateways can be utilized for devices that need protocol translations or customized processing before entering the direct cloud gateway. Azure IoT Hub performs device provisioning at a per-device level, where each device gets an unique ID, and authentication using either symmetric keys, X.509 certificates or TPMs. IoT Hub's DPS can be integrated for more scalable and automatic device provisioning. For security, the platform encrypts payload data with TLS, and defines rules for authorizing how devices can be ingested. The Microsoft Defender for IoT service can also be integrated for enabling unified threat protection for the IoT devices. Azure Active Directory provides the general IAM functionalities for access in cloud.

Data Management
Azure IoT Hub uses an SQL-like syntax language for writing the rules for how

individual data is ingested to endpoints. As mentioned in chapter 4.1.1, Azure provides options for both warm storage and cold storage. For real-time stream processing, Azure Stream Analytics is the most common option. If the streamed data flows have higher latency, Azure Cosmos DB can be used for storing warm data in NoSQL databases, while Azure SQL Database provide warm data storage in relational SQL databases. For cold storage, Azure Data Lake or Azure Blob Storage allows for storing massive volumes of data for long-term purposes. Meanwhile, services such as Azure Machine Learning or Azure Databricks can be used to perform batch processing and analytics on the cold data. Another option is the Time Series Insights service which provides both data aggregation, analytics, and storage capabilities. Output streams and stored data can be queried and routed to Power BI for visualization, using dashboards.

Ease of Implementation
Microsoft Azure is the only platform of the three that provided device SDK support for .NET, which simplified the implementation substantially. Azure's official GitHub provides repositories with device SDK sample code for each supported language [152], making it easy for developers to get started. On their official website, they also provide documentation for all of their available services, including general information, tutorials, and several code examples, depending on the service. For the prerequisites, we needed to register an Azure account, create a resource group, create an IoT Hub, and provision a device with symmetric keys authentication. These were the only mandatory steps needed for gathering the required authentication and endpoint information for our prototype. The prototype for Microsoft Azure only consisted of one client application since the device SDK's TransportType enum easily allows users to choose which communication protocol they want to use, by only changing one line in the code. Our final implementation of the prototype resulted in 150 lines of code in total.

6.2.2 Amazon Web Services Evaluation

Interoperability
Amazon Web Services supports both technical and syntactic interoperability, but not semantic interoperability, due to the platform being model agnostic. The developers are responsible for defining the information model for how the platform interprets the incoming data and metadata. For technical interoperability, the platform supports HTTPS 1.1 over REST API posting, MQTT 3.1.1 over brokered messaging, and MQTT over WebSocket for web applications. The platform also supports a large number of SDKs and APIs for integrating and communicating with the platform. For syntactic interoperability, Amazon Web Services provide SQL-like querying and APIs for retrieving data from various cloud services. For data representation, the platform supports many common serialization formats such as JSON, XML, and CSV. For IoT messaging, JSON seems to be the most favorable option.

IoT Support
Amazon Web Services lists thirteen IoT-specified services on their website. AWS IoT Core [214] is used for device provisioning, data ingestion, digital twin reg-

istration, and securely connecting billions of IoT devices to the platform. AWS IoT Device Management [216] can be integrated with AWS IoT Core for providing extended management capabilities over the devices, while AWS IoT Device Defender [215] can be integrated for simplifying device authentication and extend the device security and monitoring capabilities. AWS IoT TwinMaker [212] allows users to easily create digital twin representations of more real-world data entities. AWS IoT Greengrass [220] and AWS IoT SiteWise [222] provide the platform's edge computing capabilities. AWS IoT Greengrass provide edge runtime software for enabling IoT devices to preprocess data locally and still managing them in the cloud, while AWS IoT SiteWise can interface with industrial equipment and legacy protocols for collecting, analyzing, and monitoring their data at scale. AWS IoT Analytics [213] is an analytics service that provides stream processing, monitoring, and time-series storage for massive volumes of data. FreeRTOS [235] provide an open-source operating system to run on microcontrollers for creating resource-constrained devices that can connect and send data to the cloud. AWS IoT Events [217] simplifies the steps for detecting and responding to events from IoT devices and applications at scale. AWS IoT 1-Click [211] provides out of the box functionalities for easily associating devices with desired AWS Lambda Functions, enabling simple device-triggering of actions. AWS IoT RoboRunner [221] is robotics service that provides integration of different robot types and application building for managing them. AWS IoT ExpressLink [218] provides a catalog of hardware modules that were developed by AWS partners, that can easily be utilized and connected to the cloud. Lastly, AWS IoT FleetWise [219] is a specified service for easily collecting data from vehicles and transferring them to cloud. While the number of IoT-specific services are abundant, some of them are designed for very specific use cases, particularly the last five mentioned ones, i.e. AWS IoT Events, AWS IoT 1-Click, AWS IoT RoboRunner, AWS IoT ExpressLink, and AWS IoT FleetWise.

Device Connectivity

Amazon Web Services allows devices to connect to the cloud either through the use of supported protocol bindings or through enabled edge gateways. AWS IoT Core provides device provisioning for each device. AWS IoT Management can be integrated for extensive monitoring and management capabilities, including bulk registering and group organization of devices. For authentication, different X.509 certificates are typically used, but users can also create custom authentication tokens with JSON Web Tokens (JWTs) and OAuth. Optionally, SigV4 can be used for authentication with HTTPS and MQTT over WebSocket in particular. A defined policy must also be attached to each device before it is allowed to access the cloud. For security, payload messages in transit are encrypted through TLS, while AWS IAM can specify rules, roles and user access in the cloud. AWS IoT Defender can also be integrated for for enabling additional security.

Data Management

In Amazon Web Services, data ingestion is performed through AWS IoT Core's rules engine, using a SQL-like syntax language for defining the rules. Several rules can also be triggered based on specified events with AWS Lambda Functions, executing serverless code. For real-time stream processing, Amazon

Kinesis provide separated modules for data streams, analytics streams, video streams, and Data Firehose. For storage options, Amazon DynamoDB provide flexible, name-value NoSQL databases that can be used for both temporal and long-term storage, while AWS Simple Storage Service (S3) is an object storage with massive scalability capabilities for long-term storage of massive data volumes. For visualization purposes, Amazon QuickSight provide multiple interactive dashboards for analysis, business processes and graphical monitoring of queried output streams and stored data.

Ease of Implementation
The prototype developed for Amazon Web Services consisted of two client applications. One client for sending messages with HTTPS, and another client for sending messages with MQTT. The platform provides IoT device SDKs for five different programming languages, but unfortunately not for .NET. The HTTPS client were developed using .NET methods for REST API posting, while the MQTT client used the M2Mqtt open-source library. The platform's official website provides developer guides for the majority of their cloud services, including AWS IoT Core [214]. On their official GitHub [236], they provide documentation for all their SDKs and device SDKs, including code samples for the supported languages. Additionally, the platform also provides another GitHub [237] that is fully dedicated providing code samples for various use case scenarios, including traditional IoT client implementations in .NET. For the prerequisites, we needed to sign up for an account, configure AWS IoT Core, and provision a device. For authentication, we chose the option for auto-generating a X.509 certificate, public key, and private key set. We also needed to create and attach a policy to our device. We then downloaded the authentication files, including Amazon's official CA root file. For simplifying certificate conversion in .NET, we reformatted the downloaded files and then executed an OpenSSL command for chaining the authentication files to a .NET interpretable client certificate. Once the certificate conversion was in order, the implementation process was relatively straight forward due to their excellent code documentation. For the amount of code, the final HTTP client implementation only consisted of 92 lines of code, while the MQTT client only required 85 lines of code.

6.2.3 Google Cloud Platform Evaluation

Interoperability
Similar to the two previous platforms, the Google Cloud Platform only supports technical and syntactic interoperability, while the developers are responsible for establishing the information model for enabling semantic interoperability on the platform. For technical interoperability, the Google Cloud Platform supports HTTPS 1.1 publishing via REST API posting, and MQTT 3.1.1 for both publishing and subscribing to the broker. It also supports various SDKs and APIs for multiple languages that can be used for communicating and integrating with the platform, through different applications and systems. For syntactic interoperability the platform supports multiple common data formats, depending on the service. JSON is the most favorable option when serializing IoT payload data for transportation to the cloud. For data retrieval, the BigQuery service

can be used for querying desired data, including several APIs.

IoT Support
When it comes to Google Cloud Platform's IoT support, it technically only has one cloud service that is fully dedicated to IoT: IoT Core [75]. IoT Core's two main components are its device manager, and the two protocol bridges. The device manager registers and authenticates devices through provisioning, as well as monitoring them. The devices can securely connect to IoT Core via one of two protocol bridges in order to send messages to the platform. IoT Core is heavily reliant on the Cloud Pub/Sub service for providing brokered endpoint messaging and proper data ingestion to other services. However, Cloud Pub-/Sub is a general-purpose cloud service that is used for much more then only supporting IoT Core.

Device Connectivity
Devices can connect to Google Cloud Platform either via the HTTP bridge or MQTT bridge. Both bridges are broker-based with publishers and subscribers. In case of the HTTP bridge, messages can only be published. For device provisioning, IoT Core's device manager provides registry creation and authentication of devices. Each registry is bind to one or more Cloud Pub/Sub topics where devices publish their messages. Each device can be authenticated through the use of X.509 certificates and JWTs. Additionally, Google's official CA root file is also required for connections over the MQTT bridge. A public/private key pair can be created where the public key (client certificate) is added to the registered device in IoT Core, while the private key is needed for the creation of the JWT. For JWT's verification signature, it supports RSA or Elliptic Curve algorithms. For security, payload messages under transmission are encrypted using TLS 1.2. Google's Cloud IAM service is used for managing cloud access, while the Cloud DataFlow service enables pipelines for forwarding data to other services.

Data Management
In the Google Cloud Platform, data are ingested through pipelines, using Cloud DataFlow. In one pipeline, Google Cloud Functions can deploy serverless functions for automatic event-triggering in response to new published messages in Cloud Pub/Sub. In another parallel pipeline, the Cloud DataFlow can route flows of data to other service endpoints. DataFlow also provide unified stream processing and batch processing of data. For storage of low-latency data (warm data), the Google Cloud Platform offers Cloud Datastore for storage in NoSQL databases, while Cloud BigTable provides storage in relational SQL databases. For long-term storage, the Cloud Storage service can be used. BigQuery is a fully managed data warehouse with its main functionality being high-scalable data querying. It can route data to services such as the AI Platform for training ML models, or Datalab for exploration, transformation, and analysis of data. BigQuery can also route output streams or stored data to Google's Data Studio where users can visualize the data and perform business processes, using dashboards.

Ease of Implementation

For our Google Cloud Platform prototype, we developed an HTTPS client for publishing messages over the HTTP bridge, and an MQTT client for the publishing messages over the MQTT bridge. The platform only provides device SDK support for the Embedded C language, which meant that we needed implement the clients in more traditional ways. Fortunately, IoT Core's documentation on their official website [75] provide general information, guides, and example code for publishing data to IoT Core. The platform's official GitHub [188] also provide sample code for several languages, using open-source libraries for MQTT. For the platform prerequisites, we needed set up both the IoT Core and Cloud Pub/Sub service. In the Cloud Pub/Sub service, we needed to create a subscription and an event topic. In IoT Core, we created a registry for our devices, and then registered one device. For authentication we executed an OpenSSL command for creating a client certificate and private key pair, as shown in chapter 5.5.1. Then, the client certificate information needed to be added to our registered device as a public key. We also needed to download Google's official CA root file. For both client implementation, we needed to programmatically create a JWT and strictly follow the requirements of their respectable protocol bridge. The MQTT client in particular also needed to convert the CA root file in the code before being able to create a X.509 certificate instance of it, making the implementation rather challenging. Regarding the amount of code, the HTTP client ended up on 164 lines, while the MQTT client required 186 lines of code.

Chapter 7

Conclusion and Future Work

In the final chapter, we discuss the thesis' findings and results, and answer our research questions. Then we give our final conclusion and cover the future work that can be done further.

7.1 Summary

In this thesis, we have prototyped and experimentally evaluated the sensor data integration capabilities of three selected cloud platforms in a Smart Ocean context. A multi-purpose .NET client application has been developed with the ability to request sensor messages from the AADI Real-Time Collector and make them available to other applications, using MSMQ and web APIs. The three sensor data integration prototypes are able to receive the sensor messages from the client interface and successfully integrate them to all three cloud platforms, using their supported protocol bindings.

In the previous chapter, we evaluated the three cloud platforms based on a set of defined evaluation criteria. The results show that all three cloud platforms offers the same amount of interoperability levels with support for both technical and syntactic interoperability, but are lacking when it comes to semantic interoperability. Metadata can be stored and processed on the platforms, but they do not necessarily understand the meaning of the information. The survey done in chapter 3.2 highlights several metadata industry standards that can be utilized for integrating support for semantic interoperability in the cloud platforms. Regardless of the platform of choice, the developers needs to adapt to each platform's specific SDKs and APIs, and negotiate device access based on their specified authentication requirements.

When it comes to IoT support, Amazon Web Services offers the largest amount of services dedicated to IoT with thirteen IoT services. Microsoft Azure offers nine IoT services in total, while Google Cloud Platform only has one service fully dedicated to IoT. The platforms' main IoT service is their IoT Hub/Core service that provides the device provisioning, data ingestion and cloud gateway utilities needed for connecting and managing IoT devices on their platforms. However, Microsoft and Amazon's platform has the the upper hand due to the amount of other supportive IoT services that provides more flexible IoT capabilities then Google's platform. The Google Cloud Platform offers the least amount of protocol support and almost no device SDK support. It also currently have no options for digital twins, and its way of offering edge computing is much less elegant and intuitive compared to Azure IoT Edge and AWS IoT Greengrass. These findings clearly shows that Google does not prioritize IoT on the same level as Microsoft and Amazon does on their respective cloud platforms. Besides their IoT-specific cloud services, all three platforms provides several general cloud services that can aid in the processing, storage, and business integration aspects of the IoT solutions.

For the implementation aspects, all three cloud platforms offers great documentation both on their official GitHubs and websites, and provides a wide range of SDK and API support for many different programming languages. Microsoft Azure was the only platform of the three that offered device SDK support for .NET, while the prototypes for Amazon Web Services and Google Cloud Platform needed to be developed using .NET specific REST API methods for HTTPS and open-source libraries for MQTT. Regardless, all the prototyped client applications required less then 200 lines of code. The prototypes developed for the Google Cloud Platform ended up being the most challenging one due to the required JWT implementation and certificate conversion in .NET. The prototypes developed for Amazon Web Services were easier to implement then Google, mostly due to the fact that they did not require JWTs, and that we converted the certificates in advance and only needed to pass them in as they were in the code. The implementation would have definitely been simplified if we were using another programming language that supported Amazon's available device SDKs. The prototype developed for Microsoft Azure ended up being the easiest to implement, mostly due the use of the device SDK. If we were not using the device SDK, the difficulty level would probably be more similar to the two other prototype implementations.

All three cloud platforms offers the essential components needed for successful sensor data integration, but also lack in some areas depending on the platform of choice. We conclude that Google Cloud Platform is overall the most inferior platform option of three for sensor data integration. Both Microsoft Azure and Amazon Web Services lies on a very similar level where it can be a matter of preference, as they provide most of the same sensor data integration capabilities.

7.2 Research Questions

We have now presented all our research and results, and gained enough evidence to answer our research questions, which were described back in chapter 1.2.

7.2.1 Research Question 1

The first research question, R1, was about how the three cloud platforms handled sensor data integration and what their associated and common concepts were. In chapter 4, we gave an overview over each cloud platform and described their relevant services and features, as well as comparing their offerings in table 4.4. In chapter 5, we show the required platform setup and implementation needed to successfully send sensor messages to each cloud platform, using Smart Ocean's pilot demonstrator 1 as a case study. We then evaluated each cloud platform in chapter 6 using a set of defined evaluation criteria that were relevant to their sensor data integration aspects. As we have shown, all three cloud platforms handles sensor data integration in a very similar manner. Each cloud platform provides a main IoT service hub that acts as the cloud gateway where physical IoT devices can securely connect to. Once devices are authenticated and connected to the cloud, they can start sending data to the IoT service hub. From here, the data can be routed to other connected services for processing, storage, business integration, and other utilities. The amount of services utilized is entirely up to each user as they develop end-to-end IoT solutions based on their own requirements and needs. All three platforms offers the essential components needed for sensor data integration, but Microsoft Azure or Amazon Web Services are overall the better choices, while Google Cloud Platform lacks in several areas.

7.2.2 Research Question 2

The second research question, R2, was about what industry standards existed for sensor metadata, in terms of sensor descriptions, data syntax, and semantics. In chapter 3.2, we gave an overview of several metadata industry standards that can potentially be utilized in the Smart Ocean project. The investigated standards were: IPSO, OCF, OPC UA, OGC SWE, and W3C. The W3C's Semantic Web provide RDF and RDFS for resource descriptions in form of graph-based data models. The OWL standard can be used to semantically enrich the RDF data models, while the SQARQL standard enables querying of RDF data models. These standards can provide metadata for all kinds of web resources, while the SSN/SOSA provides sensor-oriented ontologies for the other Semantic Web standards. The OGC SWE framework offers SensorML for descriptions of both sensors and sensor systems, while the O&M standard provide schemas for observational data and measurements. IPSO provides a a data model with over 50 defined smart objects for representing IoT-based metadata. OCF's framework include a specification with 180 defined resource types. OPC UA provide a versatile, object-oriented data model for describing all kinds of metadata, using a flexible set of node-based classes and type definitions. For data syntax, all the standards supports either one or more common data formats such as JSON, XML, CSV, or binary for expressing the metadata.

7.2.3 Research Question 3

The third and final research question, R3, was about which APIs and protocol bindings were required and relevant in order to produce data to each of the three cloud platforms. In chapter 3.1, we conducted a survey on some of the most relevant application layer protocols for IoT scenarios. In chapter 4.1.3, 4.2.3, and 4.3.3, we highlight the supported protocol bindings and APIs for each respective cloud platform. The platforms' supported protocols are mainly HTTPS 1.1 over REST API messaging, and MQTT 3.1.1 over brokered messaging, while

Microsoft Azure in particular also additionally supports AMQP 1.0. Both Microsoft and Amazon also allows for WebSocket binding with their broker-enabled protocols. Other different non-supported protocols can also be utilized over edge computing, depending on the platform. Overall, MQTT (or optionally AMQP, in case of Microsoft Azure) is the most ideal IoT protocol to choose from as it is supported on all the three cloud platforms, offers reliable messaging via TCP, small header sizes, and efficient data transmission. HTTP as earlier mentioned, is not the most ideal IoT protocol, but it still has its purpose as the majority of the cloud platforms' APIs are based on REST implementations (see chapter 2.3.3), using HTTP commands for CRUD operations.

7.3 Conclusion

The internet of things and cloud computing paradigms are under continuous development and new technologies and solutions are constantly on the rise by devoted researchers and organizations. Even at their current evolution stages, their technology offerings shows great potential for aiding in the development of the future Smart Ocean Platform. The cloud platforms evaluated in this thesis are currently the three most popular cloud platforms on the market who have their own unique ways of offering services. However, they are only three of over 600 available IoT cloud platforms as shown in the 2021 report by IoT Analytics [13]. This means that there are potentially better cloud platforms to choose from. At the same time, Microsoft, Amazon and Google's cloud platforms holds the largest market shares for a reason. They are all flexible cloud platforms that offers a large set of various cloud services that can support the SFI Smart Ocean project in many ways, both for sensor data integration and other aspects.

Interoperability is still a major concern as there still does not exists any universal standard for dealing with these issues. The three cloud platforms offers both technical and syntactic interoperability, but lacks in semantic interoperability. As a result, the semantic interoperability must be addressed and established before the data enters the cloud gateways. The best course of action is to choose an ideal set of existing standards, protocols, and data formats that suits the organization's requirements, and create frameworks accordingly. The thesis highlights several potential metadata industry standards that can be utilized for addressing the semantic interoperability challenges in the Smart Ocean project.

As discussed in section 7.1, the Google Cloud Platform ranks at the bottom as the most inferior cloud platform of the three, based on our evaluation criteria. Both Microsoft Azure and Amazon Web Services are better options due to their higher focus on IoT, and more diverse capabilities, features, and service options for sensor data integration purposes. Which of the two is the superior one is not as clear-cut since they generally lie on a very similar level and provide most of the same IoT capabilities. Amazon Web Services is currently the cloud market leader and supports the largest amount of services in total, even though some of them are aimed towards very specific use cases. This is where the strategies differ between the two vendors, as Microsoft provide less services with more features, while Amazon provide more services with more condensed and specified features for each of them, adding more flexible options to the users. In the end, both Microsoft Azure and Amazon Web Services are recommended to be used in the Smart Ocean project. Nevertheless, all three cloud vendors are constantly introducing new features and capabilities to their platforms, which may or may not change their current standings in the future.

7.4 Related Work

The emergence of integrating IoT with Cloud Computing has opened up for many new promising architectures and approaches for developing more efficient systems, but it has also introduces several new challenges as well. There is currently an overwhelmingly large amount of competing standards, protocols, and cloud platforms to choose from when deciding on a technology stack for IoT-cloud integration. For aspects around the integration of the IoT with Cloud Computing, there exists several comprehensive surveys [26, 178] that provides overviews of common components, architectures, and approaches that demonstrates the effective complementary of the two paradigms. In order to help increasing semantic interoperability in IoT platforms, the IoT European Platforms Initiative (IoT-EPI) [14] has initiated seven different projects (AGILE, BIG IoT, INTER-IoT, VICINITY, SymbIoTe, bIoTope, and TagItSmart) that focuses on providing dynamic architectures and semantic interoperability across IoT platforms for multiple use case scenarios. There has also been several papers [197, 54, 23] that compares common communication protocols for the IoT, where MQTT comes forward as one of the most ideal protocols to choose from. In [187], the authors provide an overview of general IoT-cloud architectures, and performs a comparative analysis between Microsoft Azure, Amazon Web Services and Google Cloud Platform. It involved a comparative analysis of their parallel IoT services, as well as a performance analysis on their different costs and messaging times. For the IoT service analysis, they compared them using seven different key points. Their focus was not on declaring a winner, but rather highlight their characteristics and provide a comprehensible overview to developers so they can make an informative choice when considering a suitable platform for their IoT use cases.

7.5 Future Work

The future work of this thesis first and foremost relies on whether the SFI Smart Ocean chooses to go with one of the three evaluated cloud platforms in their project or not. There are as mentioned many other cloud platforms to choose from, but both Microsoft Azure and Amazon Web Services has shown to be solid options for sensor data integration in the cloud. When a cloud platform has been selected, the Smart Ocean developers needs to learn the specifications of the platform, and decide on an appropriate IoT solution for the project.

Regarding our developed prototypes, we were limited to using .NET Framework 4.8 due to the lack of compatibility options with the AADI Real-Time Collector. This is not ideal since .NET Framework 4.8 are at this point a legacy version, and the three cloud platforms' .NET support are mainly aiming towards the newer .NET Core versions. Fortunately, our developed client interface offers a work-around for avoiding these language restrictions. The acquired sensor data can be sent to other applications, using our developed web API. A solution is then to develop applications in other programming languages of choice that are able to fetch the web API data, and then utilize the data in these applications. The SFI Smart Ocean should also implement a graphical interface for the prototypes to allow users to input the required endpoint and authentication information for each device. For this thesis, the information were only hard-coded in, as we only prototyped with one device for each cloud platform. The MQTT protocol is shown to be the most applicable option of the supported protocols on the platforms. However, the fact that all three cloud platforms are still limited to MQTT 3.1.1 and not version 5.0, with only support for QoS level 0 and 1, and not 2, is rather disappointing. These limitations can possibly restrict potential MQTT messaging use cases on the platforms. A possible solution for enabling MQTT's full potential, is integrating a third-party MQTT broker such as HiveMQ [90]. This broker in particular is compliant with all three cloud platforms and provides full support for both version 3.1.1 and 5.0, with none of the limitations [258]. The use of device SDKs simplifies the device implementation processes on the platforms, but they also restricts several functionalities of each protocol specification.

There is also a lot more that can be done on the subject of metadata and interoperability. The thesis has presented several potential options for enabling semantic interoperability, but they have not been prototyped or tested in a Smart Ocean context. It will require further investigation in order to confidently determine the most suitable options for the project. In terms of cloud platform compatibility, the OPC UA standard in particular is already supported by Microsoft Azure and Amazon Web Services via their edge computing services, which indicates a promising place to start for further investigation. While edge computing has not been a major focus of this thesis, it will be more relevant as the Smart Ocean project progresses further. Utilizing edge nodes and local preprocessing of data before forwarding it to the cloud platform has major advantages. Once the SFI Smart Ocean enables edge nodes for experimentation in their project, further prototyping should be carried out for the chosen cloud platform's edge computing services.

List of Figures

1.1 Technology Delta Evaluation Framework [30] 10

2.1 The Five-Layer IoT Architecture 15
2.2 General Model of IoUT [111, 101] 16
2.3 High Level Architecture of IoT Cloud Components [174, 1] . . . 19
2.4 Relationship Between the Physical Sensors, Virtual Sensors and Virtual Sensor Groups [269] 20
2.5 Sensor-cloud Architecture [269] 21
2.6 Request-Response Communication Model [21] 23
2.7 Publish-Subscribe Communication Model [21] 23

3.1 LWM2M and IPSO Stack 32
3.2 IPSO Humidity Object Definition and Usage Example [251] . . . 33
3.3 OCF Functional Block Diagram [2] 34
3.4 OCF Humidity Resource Type Definition [3] 35
3.5 OPC UA Meta Model [63] 36
3.6 OPC UA Server Architecture [62] 37
3.7 OGC SWE Functional Architecture and Interactions [19, 91, 259] 39
3.8 RDF, RDFS, and SPARQL Example [87] 41
3.9 Overview of the SSN (Blue) and SOSA (Green) Classes and Properties [86] 42

4.1 Azure IoT Reference Architecture [151] 45
4.2 AWS IoT Core Architecture 48
4.3 GCP IoT Reference Architecture [75] 50

5.1 AADI Device Simulator and AADI Real-Time Collector 56
5.2 Model of the relationships between the different components . . . 58
5.3 Azure Device Creation (1) and Device Information (2) 59
5.4 Application Deployment 63
5.5 Azure Sensor Message Output 64
5.6 AWS Device Policy Creation (1) and Authentication Files (2) . . 65
5.7 AWS Sensor Message Payload Output 70
5.8 GCP Registry Creation (1) and Device Creation (2) 72
5.9 GCP Sensor Message Output 79

List of Tables

3.1 Application Layer Protocol Overview 31

4.1 Microsoft Azure Protocol Bindings 47
4.2 Amazon Web Services Protocol Bindings 50
4.3 Google Cloud Platform Protocol Bindings 52
4.4 Cloud Platform Comparison 54

Appendix A

Source code

The source code for the sensor data integration prototypes is available at this URL: `https://github.com/smartoceanplatform/sensor-cloud-integration-msc`.

For more detailed documentation regarding utilization of the Real-Time Collector client application itself, see the following URL: `https://github.com/smartoceanplatform/rtdc-integrator/blob/main/README.md`.

Appendix B

XML Sensor Data Sample from the AADI Device Simulator

```xml
<?xml version="1.0" encoding="utf-8"?>
<Device ID="4430-999" SessionID="4430-999-2010-11-12T09:54:50Z"
    Descr="Seaguard CTD SW" SerialNo="999" ProdNo="4430" ProdName="
    Seaguard CTD SW" DeviceType="Instrument" ProtocolVer="4" xmlns=
    "http://www.aadi.no/RTOutSchema">
    <Time>2022-01-16T16:03:13.2177045Z</Time>
    <TimeCorrection>0</TimeCorrection>
    <SiteInfo>
        <VerticalPosition>0</VerticalPosition>
        <Owner>AADI</Owner>
        <Reference>Technology Department</Reference>
    </SiteInfo>
    <Data SessionID="2022-01-16T11:47:16Z">
        <Time>2022-01-16T16:03:13.2177045Z</Time>
        <RecordNumber>3064</RecordNumber>
        <SensorData ID="SN100-0" Descr="System Parameters" SerialNo
            ="0" ProdNo="SN100" ProdName="System Node" ProtocolVer=
            "4">
            <Parameters>
                <Point ID="0" Descr="Battery Voltage" Type="VT_R4"
                    Format="" Unit="V" RangeMin="0" RangeMax="15">
                    <Value>2.124000</Value>
                </Point>
                <Point ID="1" Descr="Memory Used" Type="VT_I4"
                    Format="" Unit="Bytes" RangeMin="0" RangeMax="
                    12455936">
                    <Value>8806400</Value>
                </Point>
                <Point ID="2" Descr="Interval" Type="VT_I4" Format=
                    "" Unit="ms" RangeMin="" RangeMax="">
                    <Value>5006</Value>
                </Point>
            </Parameters>
        </SensorData>
        <SensorData ID="AN100-0" Descr="Analog Sensors" SerialNo="0
            " ProdNo="AN100" ProdName="Analog Sensors" ProtocolVer=
"4">
            <Parameters>
                <Point ID="0" Descr="Channel 1" Type="VT_R8" Format="%0.3f" Unit="V" RangeMin="-5" RangeMax="5">
                    <Value>4.643</Value>
                </Point>
                <Point ID="1" Descr="Channel 2" Type="VT_R8" Format="%0.3f" Unit="V" RangeMin="-5" RangeMax="5">
                    <Value>-3.010</Value>
                </Point>
                <Point ID="2" Descr="Channel 3" Type="VT_R8" Format="%0.3f" Unit="V" RangeMin="-5" RangeMax="5">
                    <Value>4.993</Value>
                </Point>
                <Point ID="3" Descr="Channel 4" Type="VT_R8" Format="%0.3f" Unit="V" RangeMin="-5" RangeMax="5">
                    <Value>4.296</Value>
                </Point>
            </Parameters>
        </SensorData>
        <SensorData ID="4117B-39" Descr="Pressure Sensor" SerialNo="39" ProdNo="4117B" ProdName="Pressure Sensor" ProtocolVer="4">
            <Parameters>
                <Point ID="0" Descr="Pressure" Type="VT_R4" Format="%0.3f" Unit="kPa" RangeMin="0" RangeMax="1000">
                    <Value>205.656</Value>
                </Point>
                <Point ID="1" Descr="Temperature" Type="VT_R4" Format="%0.3f" Unit="DegC" RangeMin="-5" RangeMax="35">
                    <Value>10.861</Value>
                </Point>
                <Point ID="2" Descr="Rawdata Pressure" Type="VT_I4" Format="%u" Unit="" RangeMin="" RangeMax="">
                    <Value>10250</Value>
                </Point>
                <Point ID="3" Descr="Rawdata Temperature" Type="VT_I4" Format="%d" Unit="" RangeMin="" RangeMax="">
                    <Value>30390</Value>
                </Point>
            </Parameters>
        </SensorData>
        <SensorData ID="4060-43" Descr="Temperature Sensor" SerialNo="43" ProdNo="4060" ProdName="Temperature Sensor" ProtocolVer="4">
            <Parameters>
                <Point ID="0" Descr="Temperature" Type="VT_R4" Format="%0.3f" Unit="DegC" RangeMin="-5" RangeMax="40">
                    <Value>10.862</Value>
                </Point>
            </Parameters>
        </SensorData>
        <SensorData ID="4319-62" Descr="Conductivity Sensor" SerialNo="62" ProdNo="4319" ProdName="Conductivity Sensor" ProtocolVer="4">
            <Parameters>
                <Point ID="0" Descr="Conductivity" Type="VT_R4" Format="%0.3f" Unit="mS/cm" RangeMin="0"
```

```
                    RangeMax="75">
                    <Value>30.133</Value>
                </Point>
                <Point ID="1" Descr="Temperature" Type="VT_R4"
                    Format="%0.3f" Unit="Deg.C" RangeMin="-5"
                    RangeMax="35">
                    <Value>10.866</Value>
                </Point>
            </Parameters>
        </SensorData>
        <SensorData ID="4648-19" Descr="Wave And Tide Sensor"
            SerialNo="19" ProdNo="4648" ProdName="Wave And Tide
            Sensor" ProtocolVer="4">
            <Parameters>
                <Point ID="0" Descr="Pressure" Type="VT_R4" Format=
                    "%0.3f" Unit="kPa" RangeMin="0" RangeMax="1000"
                    >
                    <Value>205.657</Value>
                </Point>
                <Point ID="1" Descr="Temperature" Type="VT_R4"
                    Format="%0.3f" Unit="DegC" RangeMin="-5"
                    RangeMax="35">
                    <Value>10.865</Value>
                </Point>
                <Point ID="2" Descr="Rawdata Pressure" Type="VT_I4"
                     Format="%u" Unit="" RangeMin="" RangeMax="">
                    <Value>78193</Value>
                </Point>
                <Point ID="3" Descr="Rawdata Temperature" Type="
                    VT_I4" Format="%d" Unit="" RangeMin="" RangeMax
                    ="">
                    <Value>30488</Value>
                </Point>
                <Point ID="29" Descr="Tide Pressure" Type="VT_R4"
                    Format="%0.3f" Unit="kPa" RangeMin="0" RangeMax
                    ="700">
                    <Value>205.654</Value>
                </Point>
                <Point ID="30" Descr="Tide Level" Type="VT_R4"
                    Format="%0.3f" Unit="m" RangeMin="0" RangeMax="
                    300">
                    <Value>10.633</Value>
                </Point>
                <Point ID="5" Descr="Sign. Height" Type="VT_R4"
                    Format="%0.3f" Unit="m" RangeMin="0" RangeMax="
                    10">
                    <Value>3.275</Value>
                </Point>
                <Point ID="6" Descr="Max Height" Type="VT_R4"
                    Format="%0.3f" Unit="m" RangeMin="0" RangeMax="
                    10">
                    <Value>4.055</Value>
                </Point>
                <Point ID="7" Descr="Mean Period" Type="VT_R4"
                    Format="%0.3f" Unit="s" RangeMin="0" RangeMax="
                    10">
                    <Value>6.625</Value>
                </Point>
                <Point ID="8" Descr="Peak Period" Type="VT_R4"
                    Format="%0.3f" Unit="s" RangeMin="0" RangeMax="
                    10">
                    <Value>6.652</Value>
```

```xml
                </Point>
                <Point ID="9" Descr="Energy Period" Type="VT_R4" Format="%0.3f" Unit="s" RangeMin="0" RangeMax="10">
                    <Value>6.547</Value>
                </Point>
                <Point ID="10" Descr="Mean Zero Crossing" Type="VT_R4" Format="%0.3f" Unit="s" RangeMin="0" RangeMax="10">
                    <Value>6.532</Value>
                </Point>
                <Point ID="11" Descr="Steepness" Type="VT_R4" Format="%0.3f" Unit="" RangeMin="0" RangeMax="0.2">
                    <Value>0.100</Value>
                </Point>
                <Point ID="12" Descr="Irregularity" Type="VT_R4" Format="%0.3f" Unit="" RangeMin="0" RangeMax="1">
                    <Value>0.501</Value>
                </Point>
            </Parameters>
        </SensorData>
    </Data>
</Device>
```

Bibliography

[1] The Industrial Internet Consortium (IIC). “The Industrial Internet Reference Architecture Version 1.9.” eng. In: (2019), pp. 1–58. URL: https://www.iiconsortium.org/pdf/IIRA-v1.9.pdf.

[2] Open Connectivity Foundation (OCF). *OCF Core Framework Specification 2.2.5.* URL: https://openconnectivity.org/specs/OCF_Core_Specification_v2.2.5.pdf. (accessed: 16.06.2022).

[3] Open Connectivity Foundation (OCF). *OCF Resource Type Specification 2.2.5.* URL: https://openconnectivity.org/specs/OCF_Resource_Type_Specification_v2.2.5.pdf. (accessed: 16.06.2022).

[4] Open Connectivity Foundation (OCF). *OCF Solving The IoT Standards Gap.* URL: https://openconnectivity.org/. (accessed: 16.06.2022).

[5] Open Connectivity Foundation (OCF). *OCF Specification 2.2.5.* URL: https://openconnectivity.org/developer/specifications/. (accessed: 16.06.2022).

[6] Preeti Agarwal and Mansaf Alam. “Open Service Platforms for IoT.” In: *Internet of Things (IoT): Concepts and Applications*. Ed. by Mansaf Alam, Kashish Ara Shakil, and Samiya Khan. Cham: Springer International Publishing, 2020, pp. 43–59. ISBN: 978-3-030-37468-6. DOI: 10.1007/978-3-030-37468-6_3. URL: https://doi.org/10.1007/978-3-030-37468-6_3.

[7] Ian F Akyildiz et al. “Wireless sensor networks: a survey.” In: *Computer networks* (2002), pp. 393–422. DOI: https://doi.org/10.1016/S1389-1286(01)00302-4.

[8] Atif Alamri et al. “A Survey on Sensor-Cloud: Architecture, Applications, and Approaches.” eng. In: (2013), pp. 1–18. DOI: https://doi.org/10.1155/2013/917923.

[9] Mohammed M Alani. “Guide to OSI and TCP/IP models.” In: (2014). DOI: https://doi.org/10.1007/978-3-319-05152-9.

[10] Afrah Ali and Alauddin Al-Omary. “Integrating Wireless Sensor Networks with Cloud Computing, a Survey.” In: *2020 International Conference on Data Analytics for Business and Industry: Way Towards a Sustainable Economy (ICDABI)*. IEEE. 2020, pp. 1–6. DOI: https://doi.org/10.1109/ICDABI51230.2020.9325681.

[11] Syed Arshad Ali, Manzoor Ansari, and Mansaf Alam. “Resource Management Techniques for Cloud-Based IoT Environment.” In: *Internet of Things (IoT): Concepts and Applications*. Ed. by Mansaf Alam, Kashish Ara Shakil, and Samiya Khan. Cham: Springer International Publishing,

2020, pp. 63–87. ISBN: 978-3-030-37468-6. DOI: 10.1007/978-3-030-37468-6_4.

[12] Fatma Alshohoumi et al. "Systematic review of existing IoT architectures security and privacy issues and concerns." In: *Int. J. Adv. Comput. Sci. Appl* (2019), pp. 232–251. DOI: http://dx.doi.org/10.14569/IJACSA.2019.0100733.

[13] IoT Analytcs. *2021 List of IoT Platforms Companies.* URL: https://iot-analytics.com/product/list-of-iot-platform-companies/. (accessed: 17.07.2022).

[14] Bröring Arne et al. "Advancing IoT platforms interoperability." In: (2018). URL: https://iot-epi.eu/wp-content/uploads/2018/07/Advancing-IoT-Platform-Interoperability-2018-IoT-EPI.pdf.

[15] Aanderaa Data Instruments AS. *About Aanderaa and Xylem.* URL: https://www.aanderaa.com/about. (accessed: 20.07.2022).

[16] Aanderaa Data Instruments AS. "SEAGUARD Platform." In: (2018). URL: https://www.aanderaa.com/media/pdfs/seaguard-platform.pdf.

[17] Aanderaa Data Instruments AS. "TD278 AADI Real-Time Programming Reference." In: (2011). URL: https://github.com/smartoceanplatform/rtdc-integrator/blob/main/Documentation/TD278%5C%20AADI%5C%20Real-Time%5C%20Programming%5C%20Reference.pdf%20(NB:%20Requires%20Access).

[18] Zoran B Babovic. "Semantic Internet of Things Platforms-Examples of Two Different Approaches for Cloud Environment." In: *2019 27th Telecommunications Forum (TELFOR).* IEEE. 2019, pp. 1–8.

[19] Zoran Babovic and Veljko Milutinovic. "Novel system architectures for semantic-based integration of sensor networks." In: *Advances in Computers.* Elsevier, 2013, pp. 91–183. DOI: https://doi.org/10.1016/B978-0-12-408091-1.00002-6.

[20] Arshdeep Bahga and Vijay Madisetti. *Cloud computing: A hands-on approach.* CreateSpace Independent Publishing Platform, 2013. ISBN: 9781494435141.

[21] Arshdeep Bahga and Vijay Madisetti. *Internet of Things: A Hands-On Approach.* VPT, 2014. ISBN: 9780996025515.

[22] T Bangemann et al. "Industrie 4.0-Technical Assets: Basic terminology concepts life cycles and administration models." In: *VDI/VDE and ZVEI* (2016).

[23] Cüneyt Bayılmış et al. "A survey on communication protocols and performance evaluations for Internet of Things." In: *Digital Communications and Networks* (2022). DOI: https://doi.org/10.1016/j.dcan.2022.03.013.

[24] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2).* RFC 7540. May 2015. DOI: 10.17487/RFC7540. URL: https://www.rfc-editor.org/info/rfc7540.

[25] Carsten Bormann. *CoAp - Overview.* URL: https://coap.technology/. (accessed: 17.03.2022).

[26] Alessio Botta et al. "Integration of cloud computing and internet of things: a survey." In: *Future generation computer systems* (2016), pp. 684–700. DOI: https://doi.org/10.1016/j.future.2015.09.021.

[27] Legion of the Bouncy Castle Inc. *Bouncy Castle C API.* URL: https://bouncycastle.org/csharp/index.html. (accessed: 20.07.2022).

[28] Tim Bray et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* URL: https://www.w3.org/TR/xml/.

[29] Dan Brickley, Ramanathan V Guha, and Brian McBride. "RDF Schema 1.1." In: (2014). URL: https://www.w3.org/TR/rdf-schema/.

[30] Alan W. Brown and Kirt C. Wallnau. ""A Framework for Evaluating Software Technology"." eng. In: (1996), pp. 39–49. DOI: 10.1109/52.536457.

[31] Muhammad Burhan et al. "IoT elements, layered architectures and security issues: A comprehensive survey." In: *Sensors* (2018), pp. 1–37. DOI: http://dx.doi.org/10.3390/s18092796.

[32] Sunil Cheruvu et al. "IoT Frameworks and Complexity." In: *Demystifying Internet of Things Security: Successful IoT Device/Edge and Platform Security Deployment.* Berkeley, CA: Apress, 2020, pp. 23–148. ISBN: 978-1-4842-2896-8. DOI: 10.1007/978-1-4842-2896-8_2. URL: https://doi.org/10.1007/978-1-4842-2896-8_2.

[33] Google Cloud. *Cloud Pub/Sub API.* URL: https://cloud.google.com/pubsub/docs/reference/rest/. (accessed: 20.07.2022).

[34] Mario Collotta et al. "Bluetooth 5: A concrete step forward toward the IoT." In: *IEEE Communications Magazine* 56.7 (2018), pp. 125–131. DOI: https://doi.org/10.1109/MCOM.2018.1700053.

[35] Michael Compton et al. "The SSN ontology of the W3C semantic sensor network incubator group." In: *Journal of Web Semantics* (2012), pp. 25–32. DOI: https://doi.org/10.1016/j.websem.2012.05.003.

[36] Open Geospatial Consortium. *Observations and Measurements.* URL: https://www.ogc.org/standards/om. (accessed: 03.05.2022).

[37] Open Geospatial Consortium. *OGC PUCK Protocol Standard.* URL: https://www.ogc.org/standards/puck. (accessed: 03.05.2022).

[38] Open Geospatial Consortium. *OGC SensorThings API.* URL: https://www.ogc.org/standards/swecommon. (accessed: 03.05.2022).

[39] Open Geospatial Consortium. *OGC WaterML.* URL: https://www.ogc.org/standards/waterml. (accessed: 03.05.2022).

[40] Open Geospatial Consortium. *Open Geospatial Consortium.* URL: https://www.ogc.org/. (accessed: 03.05.2022).

[41] Open Geospatial Consortium. *Sensor Model Language (SensorML).* URL: https://www.ogc.org/standards/sensorml. (accessed: 03.05.2022).

[42] Open Geospatial Consortium. *Sensor Observation Service.* URL: https://www.ogc.org/standards/sos. (accessed: 03.05.2022).

[43] Open Geospatial Consortium. *Sensor Planning Service (SPS).* URL: https://www.ogc.org/standards/sps. (accessed: 03.05.2022).

[44] Open Geospatial Consortium. *SWE Common Data Model Encoding Standard.* URL: https://www.ogc.org/standards/swecommon. (accessed: 03.05.2022).

[45] Open Geospatial Consortium. *SWE Service Model Implementation Standard.* URL: https://www.ogc.org/standards/swes. (accessed: 03.05.2022).

[46] The World Wide Web Consortium. *Linked Data.* URL: https://www.w3.org/standards/semanticweb/data. (accessed: 08.05.2022).

[47] The World Wide Web Consortium. *Semantic Web.* URL: https://www.w3.org/standards/semanticweb/. (accessed: 08.05.2022).

[48] The World Wide Web Consortium. *W3C - Leading the web to its full potential.* URL: https://www.w3.org/. (accessed: 08.05.2022).

[49] World Wide Web Consortium et al. “OWL 2 Web Ontology Language Document Overview (Second Edition).” In: (2012). URL: https://www.w3.org/TR/owl2-overview/.

[50] World Wide Web Consortium et al. “RDF 1.1 Concepts and Abstract Syntax.” In: (2014). URL: https://www.w3.org/TR/rdf11-concepts/.

[51] World Wide Web Consortium et al. “SPARQL 1.1 Overview.” In: (2013). URL: https://www.w3.org/TR/sparql11-overview/.

[52] Mauro AA da Cruz et al. “A reference model for internet of things middleware.” In: *IEEE Internet of Things Journal* (2018), pp. 871–883. DOI: https://doi.org/10.1109/JIOT.2018.2796561.

[53] Saikou Y Diallo et al. “Understanding interoperability.” In: *Proceedings of the 2011 Emerging M&S Applications in Industry and Academia Symposium.* 2011, pp. 84–91. URL: https://www.researchgate.net/publication/220954268_Understanding_interoperability.

[54] Jasenka Dizdarević et al. “A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration.” eng. In: (2019), pp. 1–29. DOI: https://doi.org/10.1145/3292674.

[55] ECMA. “Standard ECMA-404 The JSON Data Interchange Syntax.” In: (2017). URL: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf.

[56] IBM Cloud Education. *Application Programming Interface (API).* URL: https://www.ibm.com/cloud/learn/api. (accessed: 28.05.2022).

[57] IBM Cloud Education. *IaaS versus PaaS versus SaaS.* URL: https://www.ibm.com/cloud/learn/iaas-paas-saas. (accessed: 27.11.2022).

[58] IBM Cloud Education. *REST APIs.* URL: https://www.ibm.com/cloud/learn/rest-apis. (accessed: 28.05.2022).

[59] Asma Elmangoush. *Evaluating the features of http/2 for the internet of things.* 2017. URL: https://www.researchgate.net/publication/320453832_Evaluating_the_Features_of_HTTP2_for_the_Internet_of_Things.

[60] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.* RFC 7230. June 2014. DOI: 10.17487/RFC7230.

[61] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* University of California, Irvine, 2000.

[62] OPC Foundation. *OPC Unified Architecture Part 1: Overview and Concepts.* URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-1-overview-and-concepts/. (accessed: 16.06.2022).

[63] OPC Foundation. *OPC Unified Architecture Part 3: Address Space Model.* URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-3-address-space-model/. (accessed: 16.06.2022).

[64] OPC Foundation. *OPC Unified Architecture Part 4: Services.* URL: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-4-services/. (accessed: 16.06.2022).

[65] OPC Foundation. *OPC Unified Architecture Part 5: Information Model.* URL: `https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-5-information-model/`. (accessed: 16.06.2022).

[66] OPC Foundation. *OPC Unified Architecture Specification.* URL: `https://opcfoundation.org/developer-tools/specifications-unified-architecture`. (accessed: 16.06.2022).

[67] Paul Fremantle. "A reference architecture for the internet of things." In: *WSO2 White paper* (2015), pp. 02–04. URL: `https://docs.huihoo.com/wso2/wso2-whitepaper-a-reference-architecture-for-the-internet-of-things.pdf`.

[68] Ala Al-Fuqaha et al. "Internet of things: A survey on enabling technologies, protocols, and applications." In: *IEEE communications surveys & tutorials* (2015), pp. 2347–2376. DOI: `https://doi.org/10.1109/COMST.2015.2444095`.

[69] Steve Harris Garlik, Andy Seaborne, and Eric Prud'hommeaux. "SPARQL 1.1 Query language." In: (2013). URL: `https://www.w3.org/TR/sparql11-query/`.

[70] Matthew Gigli, Simon GM Koo, et al. "Internet of things: services and applications categorization." In: *Adv. Internet Things* (2011), pp. 27–31. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.432.6422%5C&rep=rep1%5C&type=pdf`.

[71] Crossbar.io Technologies GmbH. *The Web Application Messaging Protocol.* URL: `https://wamp-proto.org/index.html`. (accessed: 05.04.2022).

[72] Regel Gonzalez-Usach et al. "Interoperability in IoT." In: *Handbook of research on big data and the IoT*. IGI Global, 2019, pp. 149–173.

[73] Google. *Cloud IoT API.* URL: `https://cloud.google.com/iot/docs/reference/cloudiotdevice/rest`. (accessed: 20.07.2022).

[74] Google. *Create an IoT Core device registry.* URL: `https://cloud.google.com/iot/docs/create-device-registry`. (accessed: 20.07.2022).

[75] Google. *Google Cloud IoT Core documentation.* URL: `https://cloud.google.com/iot/docs/`. (accessed: 20.07.2022).

[76] Google. *Method: projects.locations.registries.devices.publishEvent.* URL: `https://cloud.google.com/iot/docs/reference/cloudiotdevice/rest/v1/projects.locations.registries.devices/publishEvent`. (accessed: 20.07.2022).

[77] Google. *Pub/Sub.* URL: `https://cloud.google.com/pubsub/`. (accessed: 11.06.2022).

[78] Google. *Publishing over the HTTP bridge.* URL: `https://cloud.google.com/iot/docs/how-tos/http-bridge`. (accessed: 20.07.2022).

[79] Google. *Publishing over the MQTT bridge.* URL: `https://cloud.google.com/iot/docs/how-tos/mqtt-bridge`. (accessed: 20.07.2022).

[80] Google. *Service: cloudiotdevice.googleapis.com.* URL: `https://cloud.google.com/iot/docs/reference/cloudiotdevice/rest`. (accessed: 20.07.2022).

[81] Google. *Using JSON Web Tokens (JWTs).* URL: `https://cloud.google.com/iot/docs/how-tos/credentials/jwts`. (accessed: 20.07.2022).

[82] MP Gopinath et al. "A secure cloud-based solution for real-time monitoring and management of Internet of underwater things (IOUT)." In:

Neural Computing and Applications (2019), pp. 293–308. DOI: https://doi.org/10.1007/s00521-018-3774-9.

[83] IEEE 802.11 Working Group. URL: https://www.ieee802.org/11/. (accessed: 02.07.2022).

[84] IEEE 802.3 Working Group. URL: https://www.ieee802.org/3/. (accessed: 02.07.2022).

[85] Sara Hachem, Thiago Teixeira, and Valérie Issarny. "Ontologies for the internet of things." In: *Proceedings of the 8th middleware doctoral symposium.* 2011, pp. 1–6. DOI: https://doi.org/10.1145/2093190.2093193.

[86] Armin Haller et al. "Semantic Sensor Network Ontology." In: (2017). URL: https://www.w3.org/TR/vocab-ssn/.

[87] Sandro Hawke et al. "SPARQL 1.1 Entailment Regimes." In: *World Wide Web Consortium (W3C)* (2013). URL: https://www.w3.org/TR/sparql11-entailment/.

[88] Patrick J. Hayes and Peter F. Patel-Schneider. "RDF 1.1 Semantics." In: (2014). URL: https://www.w3.org/TR/rdf11-mt/.

[89] Bob Hinden and Dr. Steve E. Deering. *Internet Protocol, Version 6 (IPv6) Specification.* RFC 2460. Dec. 1998. DOI: 10.17487/RFC2460. URL: https://www.rfc-editor.org/info/rfc2460.

[90] HiveMQ. *HiveMQ MQTT Broker.* URL: https://www.hivemq.com/hivemq/mqtt-broker/. (accessed: 08.08.2022).

[91] Jan Höller et al. "IoT Architecture – State of the Art." In: *From Machine-to-Machine to the Internet of Things.* Elsevier, 2018, pp. 145–165. DOI: https://doi.org/10.1016/B978-0-12-407684-6.00006-1.

[92] *Internet Protocol.* RFC 791. Sept. 1981. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791.

[93] Philip Irving and Pascal A Ochang. "Evolutionary analysis of GSM, UMTS and LTE mobile network architectures." In: *World Scientific News* 54 (2016), pp. 27–39. URL: https://sure.sunderland.ac.uk/id/eprint/7512/.

[94] Funda Ízdemir, Zeynep Ídemiş Ízger, et al. "Rivest-Shamir-Adleman Algorithm." In: *Partially Homomorphic Encryption.* Springer, 2021, pp. 37–41. ISBN: 978-3-030-87629-6. DOI: https://doi.org/10.1007/978-3-030-87629-6_3.

[95] Krzysztof Janowicz and Michael Compton. "The Stimulus-Sensor-Observation Ontology Design Pattern and its Integration into the Semantic Sensor Network Ontology." In: *SSN.* Citeseer. 2010. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.174.8881&rep=rep1&type=pdf.

[96] Jaime Jimenez, Michael Koster, and Hannes Tschofenig. "IPSO smart objects." In: *Position paper for the IOT Semantic Interoperability Workshop.* 2016. URL: https://www.omaspecworks.org/wp-content/uploads/2018/03/ipso-paper.pdf.

[97] Michael Jones. *JSON Web Algorithms (JWA) - Digital Signature with ECDSA.* RFC 7518. May 2015. DOI: 10.17487/RFC7518. URL: https://datatracker.ietf.org/doc/html/rfc7518#section-3.4.

[98] Michael Jones. *JSON Web Algorithms (JWA) - Digital Signature with RSASSA-PKCS1-v1*$_5$. RFC 7518. May 2015. DOI: 10.17487/RFC7518.

URL: https://datatracker.ietf.org/doc/html/rfc7518#section-3.3.

[99] Michael Jones, John Bradley, and Nat Sakimura. *JSON Web Signature (JWS).* RFC 7515. May 2015. DOI: 10.17487/RFC7515. URL: https://www.rfc-editor.org/info/rfc7515.

[100] Michael Jones, John Bradley, and Nat Sakimura. *JSON Web Signature (JWS) - JOSE Header.* RFC 7515. May 2015. DOI: 10.17487/RFC7515. URL: https://www.rfc-editor.org/info/rfc7515#section-4.

[101] Chien-Chi Kao et al. "A comprehensive study on the internet of underwater things: applications, challenges, and channel models." In: *Sensors* (2017), pp. 1–20. DOI: https://doi.org/10.3390/s17071477.

[102] Vivek Kapoor, Vivek Sonny Abraham, and Ramesh Singh. "Elliptic curve cryptography." In: *Ubiquity* 2008.May (2008), pp. 1–8. DOI: https://doi.org/10.1145/1386853.1378356.

[103] Vasileios Karagiannis et al. "A Survey on Application Layer Protocols for the Internet of Things." eng. In: (2015), pp. 9–18. URL: https://jesusalonsozarate.files.wordpress.com/2015/01/2015-transaction-on-iot-and-cloud-computing.pdf.

[104] Rafiullah Khan et al. "Future internet: the internet of things architecture, possible applications and key challenges." In: *2012 10th international conference on frontiers of information technology.* IEEE. 2012, pp. 257–260. DOI: https://doi.org/10.1109/FIT.2012.53.

[105] Noboru Koshizuka and Ken Sakamura. "Ubiquitous ID: standards for ubiquitous computing and the internet of things." In: *IEEE Pervasive Computing* (2010), pp. 98–101. DOI: https://doi.org/10.1109/MPRV.2010.87.

[106] Nallapaneni Manoj Kumar and Pradeep Kumar Mallick. "The Internet of Things: Insights into the building blocks, component interactions, and architecture layers." In: *Procedia computer science* (2018), pp. 109–117. DOI: https://doi.org/10.1016/j.procs.2018.05.170.

[107] He Li, Kaoru Ota, and Mianxiong Dong. "Learning IoT in edge: Deep learning for the Internet of Things with edge computing." In: (2018), pp. 96–101. DOI: https://doi.org/10.1109/MNET.2018.1700202.

[108] Gonçalo Marques, Nuno Garcia, and Nuno Pombo. "A Survey on IoT: Architectures, Elements, Applications, QoS, Platforms and Security Concepts." In: *Advances in Mobile Cloud Computing and Big Data in the 5G Era.* Ed. by Constandinos X. Mavromoustakis, George Mastorakis, and Ciprian Dobre. Cham: Springer International Publishing, 2017, pp. 115–130. ISBN: 978-3-319-45145-9. DOI: 10.1007/978-3-319-45145-9_5.

[109] Deborah L. McGuinness, Frank Van Harmelen, et al. "OWL Web Ontology Language Overview." In: (2004). URL: https://www.w3.org/TR/owl-features/.

[110] Alexey Melnikov and Ian Fette. *The WebSocket Protocol.* RFC 6455. Dec. 2011. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/info/rfc6455.

[111] Tommaso Melodia et al. *Advances in Underwater Acoustic Networking.* 2013. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.307.5052&rep=rep1&type=pdf.

[112] Microsoft. *ASMX Client with a WCF Service.* URL: https://docs.microsoft.com/en-us/dotnet/framework/wcf/samples/asmx-client-with-a-wcf-service. (accessed: 21.07.2022).

[113] Microsoft. *Azure Active Directory Documentation.* URL: https://docs.microsoft.com/en-us/azure/active-directory/. (accessed: 20.07.2022).

[114] Microsoft. *Azure App Service Documentation.* URL: https://docs.microsoft.com/en-us/azure/app-service/. (accessed: 20.07.2022).

[115] Microsoft. *Azure Blob Storage Documentation.* URL: https://docs.microsoft.com/en-us/azure/storage/blobs/. (accessed: 20.07.2022).

[116] Microsoft. *Azure Cosmos DB Documentation.* URL: https://docs.microsoft.com/en-us/azure/cosmos-db/. (accessed: 20.07.2022).

[117] Microsoft. *Azure Data Explorer documentation.* URL: https://docs.microsoft.com/en-us/azure/data-explorer/. (accessed: 18.07.2022).

[118] Microsoft. *Azure Databricks Documentation.* URL: https://docs.microsoft.com/en-us/azure/databricks/. (accessed: 20.07.2022).

[119] Microsoft. *Azure Digital Twins Documentation.* URL: https://docs.microsoft.com/en-us/azure/digital-twins/. (accessed: 29.07.2022).

[120] Microsoft. *Azure IoT Central documentation.* URL: https://docs.microsoft.com/en-us/azure/iot-central/. (accessed: 17.08.2022).

[121] Microsoft. *Azure IoT Edge documentation.* URL: https://docs.microsoft.com/en-us/azure/iot-edge/?view=iotedge-2020-11. (accessed: 18.07.2022).

[122] Microsoft. *Azure IoT Hub Device Provisioning Service (DPS) Documentation.* URL: https://docs.microsoft.com/en-us/azure/iot-dps/. (accessed: 20.07.2022).

[123] Microsoft. *Azure IoT Hub Documentation.* URL: https://docs.microsoft.com/en-us/azure/iot-hub/. (accessed: 29.07.2022).

[124] Microsoft. *Azure IoT Hub pricing.* URL: https://azure.microsoft.com/en-us/pricing/details/iot-hub/. (accessed: 18.07.2022).

[125] Microsoft. *Azure Machine Learning Documentation.* URL: https://docs.microsoft.com/en-us/azure/machine-learning/. (accessed: 20.07.2022).

[126] Microsoft. *Azure Maps Documentation.* URL: https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/iot. (accessed: 20.07.2022).

[127] Microsoft. *Azure Monitor Documentation.* URL: https://docs.microsoft.com/en-us/azure/azure-monitor/. (accessed: 20.07.2022).

[128] Microsoft. *Azure Percept documentation.* URL: https://docs.microsoft.com/en-us/azure/azure-percept/. (accessed: 17.08.2022).

[129] Microsoft. *Azure REST API reference.* URL: https://docs.microsoft.com/en-us/rest/api/azure/. (accessed: 27.07.2022).

[130] Microsoft. *Azure RTOS Documentation.* URL: https://docs.microsoft.com/en-us/azure/rtos/. (accessed: 20.07.2022).

[131] Microsoft. *Azure Sphere Documentation.* URL: https://docs.microsoft.com/en-us/azure-sphere/. (accessed: 20.07.2022).

[132] Microsoft. *Azure SQL Documentation.* URL: https://docs.microsoft.com/en-us/azure/azure-sql/?view=azuresql. (accessed: 20.07.2022).

[133] Microsoft. *Azure Time Series Insights documentation.* URL: https://docs.microsoft.com/nb-no/azure/time-series-insights/. (accessed: 18.07.2022).

[134] Microsoft. *Choose a device communication protocol.* URL: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-protocols. (accessed: 27.07.2022).

[135] Microsoft. *Connect to data in Power BI - documentation.* URL: https://docs.microsoft.com/en-us/power-bi/connect-data/. (accessed: 20.07.2022).

[136] Microsoft. *Create an IoT hub using the Azure portal.* URL: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-create-through-portal. (accessed: 18.07.2022).

[137] Microsoft. *Data Lake.* URL: https://azure.microsoft.com/en-us/solutions/data-lake/. (accessed: 18.07.2022).

[138] Microsoft. *Develop Service-Oriented Applications with WCF.* URL: https://docs.microsoft.com/en-us/dotnet/framework/wcf/. (accessed: 21.07.2022).

[139] Microsoft. *DeviceClient Class.* URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.devices.client.deviceclient?view=azure-dotnet. (accessed: 18.07.2022).

[140] Microsoft. *Download .NET Framework 4.8.* URL: https://dotnet.microsoft.com/en-us/download/dotnet-framework/net48. (accessed: 21.07.2022).

[141] Microsoft. *Encoding.GetBytes Method.* URL: https://docs.microsoft.com/en-us/dotnet/api/system.text.encoding.getbytes?view=netframework-4.8. (accessed: 18.07.2022).

[142] Microsoft. *HttpWebRequest Class.* URL: https://docs.microsoft.com/en-us/dotnet/api/system.net.httpwebrequest?view=net-6.0. (accessed: 20.07.2022).

[143] Microsoft. *Install and use Azure IoT explorer.* URL: https://docs.microsoft.com/en-us/azure/iot-fundamentals/howto-use-iot-explorer. (accessed: 18.07.2022).

[144] Microsoft. *Internet Information Services (IIS).* URL: https://www.iis.net/overview. (accessed: 21.07.2022).

[145] Microsoft. *Interprocess Communications.* URL: https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications. (accessed: 21.07.2022).

[146] Microsoft. *IoT Hub REST.* URL: https://docs.microsoft.com/en-us/rest/api/iothub/. (accessed: 27.07.2022).

[147] Microsoft. *IotHubConnectionStringBuilder Class.* URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.devices.client.iothubconnectionstringbuilder?view=azure-dotnet. (accessed: 18.07.2022).

[148] Microsoft. *Manage Azure resource groups by using the Azure portal.* URL: https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/manage-resource-groups-portal. (accessed: 18.07.2022).

[149] Microsoft. *Message Class.* URL: https://docs.microsoft.com/en-us/dotnet/api/system.messaging.message?view=netframework-4.8. (accessed: 21.07.2022).

[150] Microsoft. *Message Queuing (MSMQ).* URL: https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/ms711472(v=vs.85). (accessed: 21.07.2022).

[151] Microsoft. "Microsoft Azure IoT Reference Architecture Version 2.1." In: (2018). URL: https://download.microsoft.com/download/A/4/D/A4DAD253-BC21-41D3-B9D9-87D2AE6F0719/Microsoft_Azure_IoT_Reference_Architecture.pdf.

[152] Microsoft. *Microsoft Azure IoT SDKs.* URL: https://github.com/Azure/azure-iot-sdks. (accessed: 18.07.2022).

[153] Microsoft. *Microsoft Defender for IoT Documentation.* URL: https://docs.microsoft.com/en-us/azure/defender-for-iot/. (accessed: 20.07.2022).

[154] Microsoft. *Microsoft.Azure.Devices Namespace.* URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.devices.client?view=azure-dotnet. (accessed: 18.07.2022).

[155] Microsoft. *Named Pipes.* URL: https://docs.microsoft.com/en-us/windows/win32/ipc/named-pipes. (accessed: 21.07.2022).

[156] Microsoft. *Open Web Interface for .NET (OWIN) with ASP.NET Core.* URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/owin?view=aspnetcore-6.0. (accessed: 21.07.2022).

[157] Microsoft. *Overview of Azure IoT Device SDKs.* URL: https://docs.microsoft.com/en-us/azure/iot-develop/about-iot-sdks. (accessed: 27.07.2022).

[158] Microsoft. *Project Catalog - IoT DevKit.* URL: https://microsoft.github.io/azure-iot-developer-kit/docs/projects/. (accessed: 20.07.2022).

[159] Microsoft. *Project Catalog - IoT DevKit.* URL: https://azure.microsoft.com/en-us/downloads/. (accessed: 20.07.2022).

[160] Microsoft. *RSACryptoServiceProvider Class.* URL: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rsacryptoserviceprovider?view=net-6.0. (accessed: 20.07.2022).

[161] Microsoft. *RSAParameters Struct.* URL: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rsaparameters?view=net-6.0. (accessed: 20.07.2022).

[162] Microsoft. *Stream Class.* URL: https://docs.microsoft.com/en-us/dotnet/api/system.io.stream?view=net-6.0. (accessed: 20.07.2022).

[163] Microsoft. *Task asynchronous programming model.* URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/task-asynchronous-programming-model. (accessed: 18.07.2022).

[164] Microsoft. *TransportType Enum.* URL: https://docs.microsoft.com/en-us/dotnet/api/microsoft.azure.devices.client.transporttype?view=azure-dotnet. (accessed: 18.07.2022).

[165] Microsoft. *Visual Studio: IDE and Code Editor for Software Developers and Teams.* URL: https://visualstudio.microsoft.com/. (accessed: 21.07.2022).

[166] Microsoft. *Welcome to Azure Stream Analytics.* URL: https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-introduction. (accessed: 24.04.2022).

[167] Microsoft. *What are solutions and projects in Visual Studio?* URL: https://docs.microsoft.com/en-us/visualstudio/ide/solutions-and-projects-in-visual-studio?view=vs-2022. (accessed: 21.07.2022).

[168] Microsoft. *What is Azure HDInsight?* URL: https://docs.microsoft.com/en-us/azure/hdinsight/hdinsight-overview. (accessed: 24.04.2022).

[169] Microsoft. *What is Azure?* URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/. (accessed: 29.07.2022).

[170] Microsoft. *Working with CSV and JSON files for data solutions.* URL: https://docs.microsoft.com/en-us/azure/architecture/data-guide/scenarios/csv-and-json. (accessed: 04.08.2022).

[171] Microsoft. *X509Certificate Class.* URL: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.x509certificate?view=net-6.0. (accessed: 20.07.2022).

[172] Microsoft. *X509Certificate2 Class.* URL: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.x509certificate2?view=net-6.0. (accessed: 20.07.2022).

[173] Milan Milenkovic. "Chapter 1: Introduction and Overview." In: *Internet of Things: Concepts and System Design.* Cham: Springer International Publishing, 2020, pp. 1–25. ISBN: 978-3-030-41346-0. DOI: 10.1007/978-3-030-41346-0_1. URL: https://doi.org/10.1007/978-3-030-41346-0_1.

[174] Milan Milenkovic. "Chapter 4: Cloud." In: *Internet of Things: Concepts and System Design.* Cham: Springer International Publishing, 2020, pp. 109–153. ISBN: 978-3-030-41346-0. DOI: 10.1007/978-3-030-41346-0_4.

[175] Milan Milenkovic. "Chapter 6: IoT Data Models and Metadata." In: *Internet of Things: Concepts and System Design.* Cham: Springer International Publishing, 2020, pp. 201–223. ISBN: 978-3-030-41346-0. DOI: 10.1007/978-3-030-41346-0_6. URL: https://doi.org/10.1007/978-3-030-41346-0_6.

[176] Ajay R Mishra. *Fundamentals of Network Planning and Optimisation 2G/3G/4G: Evolution to 5G.* John Wiley & Sons, 2018. ISBN: 9781119331797.

[177] Gabriel Montenegro, Christian Schumacher, and Nandakishore Kushalnagar. *IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals.* RFC 4919. Aug. 2007. DOI: 10.17487/RFC4919. URL: https://www.rfc-editor.org/info/rfc4919.

[178] Motahareh Nazari Jahantigh et al. "Integration of internet of things and cloud computing: a systematic survey." In: *IET Communications* (2020), pp. 165–176. DOI: https://doi.org/10.1049/iet-com.2019.0537.

[179] OASIS. *MQTT Version 3.1.1.* Oct. 2014. URL: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[180] OASIS. *MQTT Version 5.0.* Mar. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[181] OASIS. *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0.* Sept. 2012. URL: https://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html.

[182] SFI Smart Ocean. "Flexible and cost-effective monitoring for management of a productive and healthy ocean [Unavailable]." eng. In: (2020), pp. 1–20.

[183] SFI Smart Ocean. *SFI Smart Ocean: About the centre.* URL: https://sfismartocean.no/partnership/. (accessed: 28.10.2021).

[184] SFI Smart Ocean. *SFI Smart Ocean: Work packages.* URL: https://sfismartocean.no/work-packages/. (accessed: 16.02.2022).

[185] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. “OWL Web Ontology Language Semantics and Abstract Syntax.” In: (2004). URL: https://www.w3.org/TR/owl-semantics/.

[186] Paolo Patierno. *M2Mqtt*. URL: https://github.com/eclipse/paho.mqtt.m2mqtt. (accessed: 20.07.2022).

[187] Paola Pierleoni et al. “Amazon, Google and Microsoft solutions for IoT: Architectures and a performance comparison.” In: *IEEE access* (2019), pp. 5455–5470. DOI: https://doi.org/10.1109/ACCESS.2019.2961511.

[188] Google Cloud Platform. *GitHub - Google Cloud Platform*. URL: https://github.com/GoogleCloudPlatform. (accessed: 26.08.2022).

[189] Sharad Pratap Singh et al. “A Survey on Internet of Things (IoT): Layer Specific vs. Domain Specific Architecture.” In: *Second International Conference on Computer Networks and Communication Technologies*. Ed. by S. Smys, Tomonobu Senjyu, and Pavel Lafata. Cham: Springer International Publishing, 2020, pp. 333–341. ISBN: 978-3-030-37051-0.

[190] Tie Qiu et al. “Underwater Internet of Things in Smart Ocean: System Architecture and Open Issues.” eng. In: (2020), pp. 1–11. DOI: 10.1109/TII.2019.2946618.

[191] Ammar Rayes and Samer Salam. “IoT Protocol Stack: A Layered View.” In: *Internet of Things from Hype to Reality: The Road to Digitization*. Cham: Springer International Publishing, 2022, pp. 97–152. ISBN: 978-3-030-90158-5. DOI: 10.1007/978-3-030-90158-5_5. URL: https://doi.org/10.1007/978-3-030-90158-5_5.

[192] Ammar Rayes and Samer Salam. “The Internet in IoT.” In: *Internet of Things from Hype to Reality: The Road to Digitization*. Cham: Springer International Publishing, 2022, pp. 35–62. ISBN: 978-3-030-90158-5. DOI: 10.1007/978-3-030-90158-5_2. URL: https://doi.org/10.1007/978-3-030-90158-5_2.

[193] Mohammad Abdur Razzaque et al. “Middleware for internet of things: a survey.” In: *IEEE Internet of things journal* (2015), pp. 70–95. DOI: https://doi.org/10.1109/JIOT.2015.2498900.

[194] Carl Reed et al. “Ogc® sensor web enablement: overview and high level architecture.” In: *2007 IEEE Autotestcon* (2007), pp. 372–380. DOI: https://doi.org/10.1109/AUTEST.2007.4374243.

[195] Jenn Riley. “Understanding metadata.” In: (2017), pp. 1–45. URL: http://groups.niso.org/higherlogic/ws/public/download/17446/Understanding%5C%20Metadata.pdf.

[196] Omar Said and Mehedi Masud. “Towards internet of things: Survey and future vision.” In: *International Journal of Computer Networks* (2013), pp. 1–17. URL: https://cs.brown.edu/courses/csci2270/archives/2017/papers/Towards_Internet_of_Things_Survey_and_Fu.pdf.

[197] Tara Salman and Raj Jain. “A Survey of Protocols and Standards for Internet of Things.” eng. In: (2019), pp. 1–20. DOI: https://doi.org/10.48550/arXiv.1903.11549.

[198] Amany Sarhan. “Cloud-based IoT platform: Challenges and applied solutions.” In: *Harnessing the Internet of Everything (IoE) for accelerated innovation opportunities*. IGI Global, 2019, pp. 116–147.

[199] John Schneider et al. *Efficient XML Interchange (EXI) Format 1.0 (Second Edition)*. URL: https://www.w3.org/TR/exi/. (accessed: 02.05.2022).

[200] Andy Seaborne and Eric Prud'hommeaux. "SPARQL Query Language for RDF." In: *World Wide Web Consortium (W3C)* (2008). URL: https://www.w3.org/TR/rdf-sparql-query/.

[201] Fatih Senel et al. "Self-deployment of mobile underwater acoustic sensor networks for maximized coverage and guaranteed connectivity." In: *Ad Hoc Networks* (2015), pp. 170–183. DOI: https://doi.org/10.1016/j.adhoc.2014.09.013.

[202] Amazon Web Services. *AI Platform.* URL: https://cloud.google.com/ai-platform/docs/technical-overview. (accessed: 29.08.2022).

[203] Amazon Web Services. *Amazon DynamoDB Documentation.* URL: https://docs.aws.amazon.com/kinesis/index.html. (accessed: 28.08.2022).

[204] Amazon Web Services. *Amazon Kinesis Documentation.* URL: https://docs.aws.amazon.com/dynamodb/index.html. (accessed: 28.08.2022).

[205] Amazon Web Services. *Amazon QuickSight Documentation.* URL: https://docs.aws.amazon.com/quicksight/index.html. (accessed: 28.08.2022).

[206] Amazon Web Services. *Amazon Simple Notification Service Documentation.* URL: https://docs.aws.amazon.com/sns/index.html. (accessed: 28.08.2022).

[207] Amazon Web Services. *Amazon Simple Storage Service Documentation.* URL: https://docs.aws.amazon.com/s3/index.html. (accessed: 28.08.2022).

[208] Amazon Web Services. *Amazon Web Services Launches.* URL: https://press.aboutamazon.com/news-releases/news-release-details/amazon-web-services-launches-amazon-s3-simple-storage-service. (accessed: 26.08.2022).

[209] Amazon Web Services. *Anthos Technical Overview.* URL: https://cloud.google.com/anthos/docs/concepts/overview. (accessed: 29.08.2022).

[210] Amazon Web Services. *AWS Identity and Access Management Documentation.* URL: https://docs.aws.amazon.com/iam/. (accessed: 26.08.2022).

[211] Amazon Web Services. *AWS IoT 1-Click Documentation.* URL: https://docs.aws.amazon.com/iot-1-click/index.html. (accessed: 25.08.2022).

[212] Amazon Web Services. *AWS IoT Analytics Documentation.* URL: https://docs.aws.amazon.com/iot-twinmaker/. (accessed: 25.08.2022).

[213] Amazon Web Services. *AWS IoT Analytics Documentation.* URL: https://docs.aws.amazon.com/iotanalytics/. (accessed: 25.08.2022).

[214] Amazon Web Services. *AWS IoT Core Documentation.* URL: https://docs.aws.amazon.com/iot/index.html. (accessed: 25.08.2022).

[215] Amazon Web Services. *AWS IoT Device Defender Documentation.* URL: https://docs.aws.amazon.com/iot-device-defender/. (accessed: 25.08.2022).

[216] Amazon Web Services. *AWS IoT Device Management Documentation.* URL: https://docs.aws.amazon.com/iot-device-management/index.html. (accessed: 25.08.2022).

[217] Amazon Web Services. *AWS IoT Events Documentation.* URL: https://docs.aws.amazon.com/iotevents/. (accessed: 25.08.2022).

[218] Amazon Web Services. *AWS IoT ExpressLink Documentation.* URL: https://docs.aws.amazon.com/iot-expresslink/index.html. (accessed: 25.08.2022).

[219] Amazon Web Services. *AWS IoT FleetWise Documentation.* URL: https://docs.aws.amazon.com/iot-fleetwise/index.html. (accessed: 25.08.2022).

[220] Amazon Web Services. *AWS IoT Greengrass Documentation.* URL: https://docs.aws.amazon.com/greengrass/index.html. (accessed: 25.08.2022).

[221] Amazon Web Services. *AWS IoT RoboRunner (Preview) Documentation.* URL: https://docs.aws.amazon.com/iotroborunner/. (accessed: 25.08.2022).

[222] Amazon Web Services. *AWS IoT SiteWise Documentation.* URL: https://docs.aws.amazon.com/iot-sitewise/?id=docs_gateway. (accessed: 25.08.2022).

[223] Amazon Web Services. "AWS IoT: API Reference." eng. In: (2022), pp. 1–1093. URL: https://docs.aws.amazon.com/iot/latest/apireference/iot-api.pdf.

[224] Amazon Web Services. *AWS Lambda Documentation.* URL: https://docs.aws.amazon.com/lambda/index.html. (accessed: 28.08.2022).

[225] Amazon Web Services. *BigQuery.* URL: https://cloud.google.com/bigquery/. (accessed: 29.08.2022).

[226] Amazon Web Services. *BigTable.* URL: https://cloud.google.com/bigtable/. (accessed: 29.08.2022).

[227] Amazon Web Services. *Cloud.* URL: https://cloud.google.com/sdk/. (accessed: 29.08.2022).

[228] Amazon Web Services. *Cloud computing with AWS.* URL: https://aws.amazon.com/what-is-aws/. (accessed: 26.08.2022).

[229] Amazon Web Services. *Cloud Storage.* URL: https://cloud.google.com/storage/. (accessed: 29.08.2022).

[230] Amazon Web Services. *Data Studio.* URL: https://marketingplatform.google.com/about/data-studio/. (accessed: 29.08.2022).

[231] Amazon Web Services. *DataFlow.* URL: https://cloud.google.com/dataflow/. (accessed: 29.08.2022).

[232] Amazon Web Services. *Datalab.* URL: https://cloud.google.com/datalab/docs/. (accessed: 29.08.2022).

[233] Amazon Web Services. *Datastore.* URL: https://cloud.google.com/datastore/. (accessed: 29.08.2022).

[234] Amazon Web Services. *Device communication protocols.* URL: https://docs.aws.amazon.com/iot/latest/developerguide/protocols.html. (accessed: 26.08.2022).

[235] Amazon Web Services. *FreeRTOS Documentation.* URL: https://docs.aws.amazon.com/freertos/index.html. (accessed: 25.08.2022).

[236] Amazon Web Services. *GitHub - Amazon Web Services.* URL: https://github.com/aws. (accessed: 26.08.2022).

[237] Amazon Web Services. *GitHub - AWS Samples.* URL: https://github.com/aws-samples. (accessed: 26.08.2022).

[238] Amazon Web Services. *Github - aws-iot-dotnet-publisher-http.* URL: https://github.com/aws-samples/aws-iot-dotnet-publisher-http. (accessed: 28.08.2022).

[239] Amazon Web Services. *Google Distributed Cloud Edge.* URL: https://cloud.google.com/distributed-cloud/edge/latest/docs/overview. (accessed: 29.08.2022).

[240] Amazon Web Services. *Google Kubernetes Engine.* URL: https://cloud.google.com/kubernetes-engine/. (accessed: 29.08.2022).

[241] Amazon Web Services. *How AWS IoT works.* URL: https://docs.aws.amazon.com/iot/latest/developerguide/aws-iot-how-it-works.html. (accessed: 26.08.2022).

[242] Amazon Web Services. *Tools to Build on AWS.* URL: https://aws.amazon.com/developer/tools/. (accessed: 26.08.2022).

[243] Amazon Web Services. *TPU.* URL: https://cloud.google.com/tpu/docs. (accessed: 28.08.2022).

[244] Amazon Web Services. *TPU Edge.* URL: https://cloud.google.com/edge-tpu/docs. (accessed: 28.08.2022).

[245] Pallavi Sethi and Smruti R Sarangi. "Internet of things: architectures, protocols, and applications." In: *Journal of Electrical and Computer Engineering* 2017 (2017). DOI: https://doi.org/10.1155/2017/9324035.

[246] Yakov Shafranovich. *Common format and MIME type for comma-separated values (CSV) files.* 2005. URL: https://www.rfc-editor.org/rfc/rfc4180.

[247] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP).* RFC 7252. June 2014. DOI: 10.17487/RFC7252.

[248] Ian G. Smith. *The Internet of Things 2012: New Horizons.* IERC - Internet of Things European Reasearch Cluster, 2012. ISBN: 9780955370793.

[249] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. "OWL Web Ontology Language Guide." In: (2004). URL: https://www.w3.org/TR/owl-guide/.

[250] Annie W Sokol, Michael D Hogan, et al. "Nist cloud computing standards roadmap." In: *NIST Special Publication* (2013). DOI: https://doi.org/10.6028/NIST.SP.500-291r2.

[251] OMA SpecWorks. *Humidity.* URL: https://devtoolkit.openmobilealliance.org/OEditor/LWMOView?url=https%5C%3a%5C%2f%5C%2fraw.githubusercontent.com%5C%2fOpenMobileAlliance%5C%2flwm2m-registry%5C%2fprod%5C%2fversion_history%5C%2f3304-1_0.xml. (accessed: 21.07.2022).

[252] OMA SpecWorks. *IPSO Smart Objects Working Group.* URL: https://omaspecworks.org/about/the-oma-specworks-work-program/ipso-smart-objects-working-group/. (accessed: 21.07.2022).

[253] OMA SpecWorks. *lwm2m-registry.* URL: https://github.com/OpenMobileAlliance/lwm2m-registry. (accessed: 20.07.2022).

[254] OMA SpecWorks. *OMA LightweightM2M (LwM2M) Object and Resource Registry.* URL: https://technical.openmobilealliance.org/OMNA/LwM2M/LwM2MRegistry.html. (accessed: 20.07.2022).

[255] OMA SpecWorks. *OMA SpecWorks.* URL: https://omaspecworks.org/. (accessed: 21.07.2022).

[256] Statista. *Cloud infrastructure services vendor market share worldwide from 4th quarter 2017 to 4th quarter 2021.* URL: https://www.statista.com/statistics/967365/worldwide-cloud-infrastructure-services-market-share-vendor/. (accessed: 03.07.2022).

[257] Ed. T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259.

[258] The HiveMQ Team. *Comparison of MQTT Support by IoT Cloud Platforms.* URL: https://www.hivemq.com/blog/hivemq-cloud-vs-aws-iot/. (accessed: 08.08.2022).

[259] Vlasios Tsiatsis et al. "Architecture and State-of-the-Art." In: *Internet of Things: technologies and applications for a new age of intelligence.* Academic Press, 2018, pp. 143–180. DOI: https://doi.org/10.1016/B978-0-12-814435-0.00019-5.

[260] Himani Tyagi and Rajendra Kumar. "Cloud Computing for IoT." In: *Internet of Things (IoT): Concepts and Applications.* Ed. by Mansaf Alam, Kashish Ara Shakil, and Samiya Khan. Cham: Springer International Publishing, 2020, pp. 25–41. ISBN: 978-3-030-37468-6. DOI: 10.1007/978-3-030-37468-6_2.

[261] Hans Van Der Veer and Anthony Wiles. "Achieving technical interoperability." In: *European telecommunications standards institute* (2008).

[262] Dmitry Vsekhvalnov. *Ultimate Javascript Object Signing and Encryption (JOSE).* URL: https://github.com/dvsekhvalnov/jose-jwt. (accessed: 20.07.2022).

[263] W3C. *A JSON-based Serialization for Linked Data.* URL: https://json-ld.org/spec/latest/json-ld/. (accessed: 02.05.2022).

[264] W3C. *XML Essentials.* URL: https://www.w3.org/standards/xml/core. (accessed: 28.05.2022).

[265] Miao Wu et al. "Research on the architecture of Internet of Things." eng. In: (2010), pp. 484–487. DOI: https://doi.org/10.1109/ICACTE.2010.5579493.

[266] Xing Xiaojiang, Wang Jianli, and Li Mingdong. "Services and key technologies of the Internet of Things." In: *Zte Communications* (2020), pp. 26–29. URL: http://zte.magtechjournal.com/CN/abstract/article_363.shtml.

[267] Wei Yu et al. "A survey on the edge computing for the Internet of Things." In: (2017), pp. 6900–6919. DOI: https://doi.org/10.1109/ACCESS.2017.2778504.

[268] Michael Yuan. *Getting to know MQTT.* URL: https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/. (accessed: 14.03.2022).

[269] Madoka Yuriyama and Takayuki Kushida. "Sensor-cloud infrastructure-physical sensor management with virtualized sensors on cloud computing." In: *2010 13th international conference on network-based information systems.* IEEE. 2010, pp. 1–8. DOI: https://doi.org/10.1109/NBiS.2010.32.