

# Syntactic and Semantic Edge Interoperability


Tanzima Azad<sup>†</sup>   [ Griffith University | [tanzima.azad@griffithuni.edu.au](mailto:tanzima.azad@griffithuni.edu.au) ]

M A Hakim Newton<sup>†</sup>  [ The University of Newcastle | [mahakim.newton@newcastle.edu.au](mailto:mahakim.newton@newcastle.edu.au) ]

Jarrod Trevathan  [ Griffith University | [j.trevathan@griffith.edu.au](mailto:j.trevathan@griffith.edu.au) ]

Abdul Sattar  [ Griffith University | [a.sattar@griffith.edu.au](mailto:a.sattar@griffith.edu.au) ]

<sup>†</sup> These two authors are joint-first authors.

 School of Information and Communication Technology, Griffith University, 170 Kessels Road, Nathan, QLD 4111, Australia.

**Received:** 25 September 2024 • **Accepted:** 08 March 2025 • **Published:** 22 May 2025

**Abstract** The Internet of Things (IoT) has transformed various sectors, from home automation to healthcare, leveraging a multitude of sensors and actuators communicating through cloud, fog, and edge networks. However, the diversity in device manufacturing and communication protocols necessitates interoperable communication interfaces. Most existing IoT interoperability solutions often rely on cloud-based centralised architectures and suffer from latency and scalability issues. This work specifically focuses on scenarios where decisions need to be made with IoT edge devices in real-time, even in situations where there might be internet disruptions, low bandwidth, or no internet connection. While typical IoT interoperability solutions support edge devices, their reliance on cloud-based architectures makes them unsuitable for mission-critical applications, environmental monitoring, or water quality monitoring, where internet connectivity cannot be guaranteed. To tackle these challenges, the project InterEdge proposed a theoretical interoperability model supporting hierarchical decentralised communication between edge devices. The aforementioned framework has four levels to handle network, syntactic, semantic, and organisational aspects of interoperability. As part of the same project, this work focuses on the implementation of the syntactic and semantic levels of the aforementioned framework. This work involves tackling the implementation challenges, particularly considering key issues related to transmission latency and memory requirements. We have created profiles for edge devices and data formats to store their essential and extra information. Using the profiles, communications can be established and maintained seamlessly among edge devices. We have conducted a comparative analysis between InterEdge implementation and three other implementations of established open standards. The experimental results demonstrate that the syntactic and semantic levels of the implemented interoperability solution, InterEdge, significantly outperforms the existing open standards in terms of standard benchmarking metrics such as code size, memory usage, and response latency. The contribution of this paper lies in these implementation results, which provide concrete evidence of the superior performance of our proposed solution, InterEdge, thereby validating its efficacy in real-world IoT scenarios.

**Keywords:** Internet of Things, Middleware, Edge Devices, Syntactic Interoperability, Semantic Interoperability

## 1 Introduction

The *Internet of Things* (IoT) is a global network that allows uniquely identifiable devices to connect with each other using communication technologies. Typical IoT devices are *sensors* and *actuators*. Sensors can gather data from surroundings while actuators can perform actions. IoT has diverse applications such as home automation, smart cities, supply chains, healthcare, agriculture, and environmental monitoring. However, IoT faces a major challenge in ensuring seamless communication and data sharing between various IoT devices. An example modular farming system could have monitoring and controlling devices in charge of regulating heating, ventilation, and air conditioning. These devices might come from various vendors, supporting diverse network technologies, using different message formats, protocols, and terminologies that may not be fully compatible with each other. To deal with such scenarios, one needs an effective and scalable *interoperability* solution.

One key concern in this paper is to support emergency and

mission-critical services. These services need ultra-reliable and low-latency communication [Pitstick *et al.*, 2024; Bojadjevski *et al.*, 2018], preferably by using edge computing. Edge systems, by positioning computation closer to where data is generated and needed, offer significant advantages including reduced latency, optimised bandwidth, and enhanced resiliency and availability. These benefits are particularly valuable in highly uncertain and resource-constrained environments, such as those encountered by first responders, law enforcement, and military personnel, where timely decision-making is critical.

A significant body of research has been done to address interoperability issues in IoT. We later provide a detailed review. Briefly, these include Open Connectivity Foundation (OCF) [Park, 2017], Eclipse Hono [Kherbache *et al.*, 2022] and Eclipse Kapua [Kapua, 2018] by Eclipse IoT Working Group, EdgeX Foundry [Foundry, 2021], FIWARE [Cirillo *et al.*, 2019] Platform, Amazon Web Service (AWS) IoT [Pierleoni *et al.*, 2019], Microsoft Azure IoT [Klein, 2017], Web of Things (WoT) [Gyrard *et al.*, 2017] by W3C, IoTiv-

ity Project [Dang et al., 2017], H2020 IoF2020 Project [Verdouw et al., 2017], OneM2M [Park et al., 2016] Standard, and Sensor Observation Service (SOS) [Bröring et al., 2012] and SensorThings API [Liang et al., 2021] developed by the Open Geospatial Consortium (OGC).

Most of the above mentioned IoT interoperability solutions, except IoTivity, WoT, and Wasm, rely on cloud-based centralised architectures and suffer from latency and scalability issues. This limitation, along with the reliance on cloud-based architectures in other solutions, means that these existing works are unsuitable for scenarios where there is no internet connection, low bandwidth, or any connectivity issue. Cloud-based architectures exhibit higher latency than edge-based architectures, which provide lower latency to 92% of end-users, often by a significant margin of 10 to 100 milliseconds. Simulations show that while cloud-based systems experience exponential latency growth under constrained bandwidth and high loads, edge based systems maintain stable response times [Shukla et al., 2023; Charyyev et al., 2020; Maheshwari et al., 2018]. On the other hand, IoTivity, WoT, and Wasm can be implemented at the edge level. However, they require more powerful hardware and cannot support resource-constrained Class-0 devices like Arduino Uno R3. Moreover, IoTivity is limited to its own communication protocol and does not support others while WoT and Wasm rely on external gateways for data routing between devices.

Nevertheless, some recent work focuses on specific aspects. For instance, some studies concentrate solely on achieving semantic interoperability [Nagasundaram et al., 2024; Mofatteh et al., 2024; Paul et al., 2024], while others address only network interoperability [Pramukantoro et al., 2018; Marin et al., 2024] and some on organisation interoperability [del Campo et al., 2024].

The InterEdge project, which is focused on developing an interoperability solution for edge devices that operates entirely within an edge network, unlike typical cloud-based solutions [Rasheed, 2024]. As part of the project, in our previous published work [Azad et al., 2023], we have *theoretically* proposed a hierarchical decentralised *interoperability model* that supports both wired and wireless connectivity. The model uses controller devices to keep all communications exclusively within the edge network, even without the need for an internet connection if devices are connected via wired connections, and it can also function with low bandwidth. The model is particularly suited for scenarios where real-time decision-making is critical and cloud-based response latency is not acceptable. The model comprises bottom-to-top four levels: *network*, *syntactic*, *semantic*, and *organisational* level. The network level deals with various networking technologies to establish seamless connections among the edge devices. The syntactic level supports various data formats used in communication between the edge devices. The semantic level focuses on the meaning of the messages transmitted. The organisational level addresses platform and domain heterogeneity e.g. interoperability between a smart agriculture system and a supply chain management system. In order to realise the proposed edge interoperability model, in the same paper, we also have proposed an *interoperability framework* having five layers: *physical*, *profile*, *registration*, *service*, and *control* layer. Note the interoperability frame-

work is the organisation of the implementation aspects while the interoperability model comprise conceptual interoperability levels.

As part of InterEdge, in this work, we have *actually implemented* the syntactic and semantic levels of our aforementioned *theoretical* interoperability framework. Implementation of a theoretical framework often poses various technical challenges not innately addressed and also software development issues. This work involves tackling the implementation challenges, particularly considering key issues related to transmission latency and memory requirements. We have considered controllers, sensors and actuators as edge devices. These controllers, sensors, and actuators are Class-0 devices [Bormann et al., 2012], meaning they have limited memory and processing power. We have implemented device and controller profiles and registration processes to help establish connection and maintain. We have implemented data profiles to support syntactic and semantic interoperability levels of the interoperability model. These two levels are on top of the network level, which for this work has been implemented using wired connection. We also have implemented the controller program and the scheduler program to oversee the entire process: probing sender devices periodically, verifying conditions, and sending required messages from sender to receiver devices using routing algorithms.

This paper assumes that the network connectivity has already been established e.g. by using wired connection for the time being although InterEdge framework supports wireless connectivity as well. This allows the focus to be shifted to addressing syntactic and semantic interoperability. Implementing network interoperability involves enabling communication among devices using different network protocols such as Wi-Fi, Bluetooth, Zigbee, and others. The task requires addressing various key issues related to protocol compatibility, data transmission, network management, and security and authentication. All these make the task a substantial undertaking in itself. As such, the implementation of network interoperability is considered beyond the scope of this paper. This also makes addressing the challenge of enabling device connections using low internet bandwidth or limited connectivity out of the scope. Organisational interoperability is also out of scope because that requires to address complicated issues such as cross-domain interoperability, programming language diversity, and proprietary barriers.

Among many possibilities, we give an example application use case for this implementation. In a smart healthcare application, interoperability is crucial during a medical emergency, such as monitoring a patient's vital signs in an ambulance. Devices like heart rate monitors, oxygen sensors, and blood pressure cuffs must communicate in real time with the ambulance's onboard system. Syntactic interoperability ensures these devices share data in a compatible format, while semantic interoperability guarantees that the system correctly interprets the data (e.g., recognising a critical drop in oxygen levels and triggering an alert). This real-time, standardised communication ensures immediate medical decisions without relying on historical data or cloud processing.

We have compared InterEdge implementation with other three implementations of existing open standards—OGC SOS [Na and Priest, 2007], OGC SensorThings API [Liang

and Khalafbeigi, 2019], and EdgeX Foundry [Foundry, 2021]. Our experiments show that InterEdge implementation performs better in terms of code size, memory usage, and response latency than the three existing implementations.

To summarise, the contributions of this paper are below.

- Adding the implementation challenges to obtain an actual implementation of the syntactic and semantic interoperability levels of the theoretical interoperability model and framework InterEdge. [Azad et al., 2023].
- A comparison of our solution with three existing open standard interoperability solutions. Our solution significantly outperforms the others on class-0 microcontroller devices in terms of standard metrics such as code size, memory usage, and response latency.

The rest of the paper is organised as follows: Section 2 defines the terminologies used in this paper; Section 3 delves into existing solutions for IoT edge interoperability; Section 4 gives an overview of the interoperability framework in our previous paper [Azad et al., 2023]; Section 5 provides the implementation details of our interoperability solution for syntactic and semantic levels; Section 6 describes example edge network scenarios. Section 7 provides a comparative analysis of our proposed solution against three existing solutions. Lastly, Section 8 presents our conclusions and suggests potential avenues for future research.

## 2 Background

We cover some preliminary concepts needed for a clear understanding of the paper.

- IoT:** A network of interconnected devices to enable smart and automated systems to communicate and exchange data with each other.
- Cloud:** Remote servers on the internet to store, manage, and process data, allowing users to access and share information from anywhere.
- Edge:** Devices or systems that can read, process, and transmit data closer to the source of data generation.
- Data Read:** Getting data directly from sensors or devices.
- Data Write:** Sending data directly to actuators or devices.
- Data Transmission:** data read, data write, and exchange between controllers potentially via other controllers.
- Interoperability:** The capability of heterogeneous devices and applications to seamlessly collaborate, exchange data, and interpret data effectively, regardless of their manufacturers, underlying technologies, or protocols.

We use the same terminologies used in our previous paper on the hierarchical decentralised interoperability model [Azad et al., 2023]. To conveniently distinguish some aspects of the two papers, we introduce two new terminologies: *framework* and *implementation*. All these terms are below.

- Model:** A categorisation of interoperability: hierarchical or non-hierarchical, vertical or horizontal, or mixed.
- Level:** A category in a hierarchical or vertical interoperability model. A higher level with larger granularity sequentially depends on a lower one in the hierarchy.

**Class:** A category in a horizontal interoperability model. Classes do not mutually depend on each other.

**Solution:** An implementation framework that potentially implements an interoperability model. It also means an actual implementation of an interoperability model.

**Framework:** A potentially implemented solution.

**Implementation:** An actually implemented solution.

**Service:** An algorithmic method that implements a particular data or execution functionality for a solution.

**Layer:** A set of services sharing certain traits of functionalities within a vertical organisation of the services.

**Environment:** A connected network system of physical objects, devices, sensors, and actuators that interact.

**Interoperable Environment:** An environment that ensures seamless interoperability among its physical objects, devices, sensors, and actuators.

**Data and Message:** Data goes in or comes out of devices. Messages are data plus routing information and are circulated over the network.

## 3 Related Work

We refer to our previous paper [Azad et al., 2023] that covers cloud-based, fog-based, and edge-based i.e. all three types of IoT interoperability solutions. In this paper, we mainly focus on solutions that provide edge-level interoperability.

### 3.1 Generic Solutions

The Open Connectivity Foundation (OCF) [Park, 2017] uses open specifications and cloud access to promote interoperable communication between edge devices running on diverse platforms. Microsoft Azure IoT [Klein, 2017] and Amazon AWS IoT [Pierleoni et al., 2019] offer cloud services for various communication protocols and data formats to enable interoperability between edge devices. The OneM2M standard [Park et al., 2016] uses an M2M service layer embedded in hardware or software to allow interoperability between edge devices and applications. The Web of Things (WoT) [Gyrard et al., 2017] by W3C addresses edge-level interoperability by using cloud-based web standards that connect diverse IoT platforms. The FIWARE [Cirillo et al., 2019] platform provides cloud-based open-source components to enable seamless connectivity and data sharing among edge devices. Overall, aforementioned interoperability solutions use cloud-based centralised architectures and suffer from latency and scalability challenges.

The IoTivity project [Dang et al., 2017] by the Linux Foundation ensures secure and reliable edge level interoperability by using its own communication protocol. However, it cannot integrate other existing communication protocols. The H2020 IoF2020 [Verdouw et al., 2017] project addresses edge based interoperability with lightweight IoT solutions. However, it faces challenges in managing the complexity of coordination among its multi-layered solution.

To enhance interoperability in IoT, Sciallo et al. [2021] proposed the adoption of W3C WoT standards and develop the WoT Micro-Servient (WMS), a framework specifically designed for resource-constrained IoT devices. WMS en-

ables the deployment of native WoT applications on microcontrollers, enhancing performance by reducing latency and energy consumption compared to proxy-based solutions. However, the implementation presented in the paper is limited to devices connected via the same Wi-Fi network, despite the W3C WoT's capability to support various communication protocols, which are not demonstrated. The paper primarily focuses on achieving interoperability between two devices by enabling data exchange and understanding, but it does not explore communication across different network protocols or address cross-domain interoperability.

We later empirically compare InterEdge implementation with three recent existing interoperability solutions: SOS [Leibovici *et al.*, 2023], SensorThings API [Liang *et al.*, 2021], and EdgeX Foundry [Foundry, 2021]. SOS is a fully cloud-based solution. SensorThings API [Liang *et al.*, 2021] and EdgeX Foundry support both cloud and edge level interoperability. However, to support edge level interoperability, they still need cloud access. We further discuss these three.

The Sensor Observation Service (SOS) [Leibovici *et al.*, 2023], developed by the Open Geospatial Consortium (OGC), addresses interoperability challenges in IoT by providing a standardised web service. This service allows users to integrate sensor data from various sources using a common format. SOS also supports the management (addition, removal, querying) of sensors. However, its cloud-based architecture requires a stable network connection, which limits its effectiveness in areas with poor internet connectivity. Unreliable networks can lead to delays or failures in data transmission and retrieval, and the large XML data requests can be too demanding for low-bandwidth networks, further impacting performance. However, the SOS does not support tasking services for actuators or receivers, such as sending commands or data to any actuator or receiver.

Due to the overhead of using XML data in SOS, OGC has proposed a JSON based standard named SensorThings API [Liang *et al.*, 2021]. SensorThings API is designed for resource-constrained devices and has two main profiles: the Sensing Profile for sensors and the Tasking Profile for actuators. The Sensing Profile enables the organisation of observations into data streams, associating each observation with specific sensors and features of interest. The Tasking Profile, on the other hand, allows for the control and management (addition, removal, and querying) of actuators, specifying tasks and actions for devices to perform. Although SensorThings API is designed for edge devices, but it still faces implementation challenges with very limited processing power and memory, hindering its widespread adoption and integration across diverse IoT ecosystems.

EdgeX Foundry [Foundry, 2021] is an open-source framework designed to address interoperability challenges in IoT edge devices. It works as a translator among IoT devices and provides a common platform for connecting and managing the devices and applications, promoting seamless communication and data exchange. However, the framework's architecture requires consistent network connectivity for communication between devices and the central platform. Failure of connecting to the internet can disrupt the synchronisation of data between edge devices and cloud-based services, affecting the overall reliability and performance of the system.

### 3.2 Application Specific Solutions

While the above mentioned solutions provide a generic framework for interoperability in IoT, there are implementations of interoperability tailored to specific IoT applications, focusing on particular real-life challenges. These targeted approaches aim to address the unique requirements of domains such as healthcare, smart agriculture, and structural monitoring, where customised solutions are often necessary to achieve seamless integration and functionality.

The MAC4PRO architecture [Gigli *et al.*, 2023] introduces a sensor-to-cloud platform for Structural Health Monitoring (SHM), emphasising seamless integration of sensing and software technologies for accurate data read and transmission. However, its scope is predominantly tailored to the condition assessment of industrial and civil infrastructures. This focus limits its applicability to other domains, such as small-scale or specialised monitoring scenarios, where specific adaptations might be needed to address unique structural or operational requirements.

The GOLDIE [Hao and Schulzrinne, 2021] framework addresses IoT interoperability challenges by introducing a federated, location-based global directory for managing IoT metadata. Designed to support discoverability, geospatial queries, and global access, GOLDIE facilitates cross-domain services and enables efficient data sharing and resource discovery. While GOLDIE's hierarchical structure aligns with edge computing principles by enabling localised data read and transmission with reduced latency, it still relies on local directories deployed on servers. This introduces a dependency on centralised architecture, which can potentially limit scalability.

### 3.3 Contrasting InterEdge with Related Work

In Table 1, we compare this implementation of InterEdge's Syntactic and Semantic level interoperability levels with existing solution's Syntactic and Semantic level interoperability levels. The table shows that this work, WoT, Wasm, and IoTivity do not require cloud access to facilitate edge interoperability while the remaining works rely on cloud connectivity. The table also shows that WoT, Wasm, and IoTivity are not compatible with Class-0 devices and due to their implementation choices necessitate a minimum of Class-1 devices for operation. Moreover, their architectures do not explicitly address interoperability at distinct levels, such as syntactic and semantic interoperability levels, but rather approach the issue in a more general manner. Note that this work supports Class-0 devices and explicitly addresses interoperability at different levels, including syntactic and semantic interoperability. This implementation also has integrated controllers to provide services related to the two interoperability levels. Moreover, this work adopts a simple event-driven procedure based software architecture to maintain simplicity, reduce computational overhead, and to ensure efficiency.

## 4 Framework Overview

We provide a summary of the edge interoperability model and its associated framework proposed in our previous paper

[Azad et al., 2023]. We also briefly describe the control flow of the interoperability solution when it is being run.

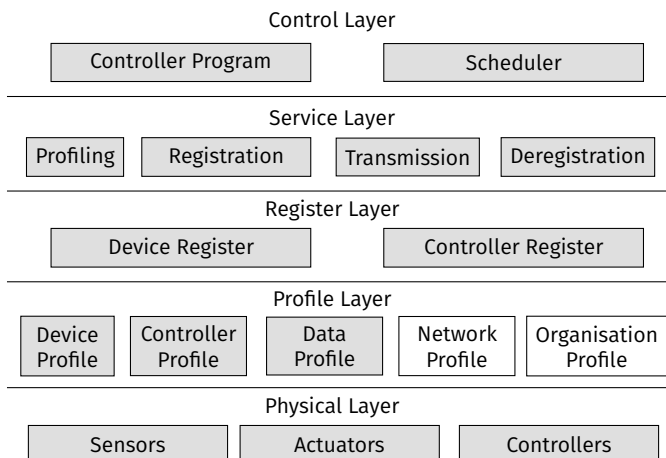
#### 4.1 Model and Framework

Table 2 left column shows the interoperability levels in the interoperability model. The model has four levels from bottom to top: *network*, *syntactic*, *semantic*, and *organisational*. The network level deals with networking technologies to establish seamless connections among edge devices. The syntactic level supports data formats used in communication. The semantic level provides the meaning of the messages transmitted. The organisational level addresses platform and domain heterogeneity present in the edge network.

**Table 2.** Edge Interoperability Model by [Azad et al., 2023]: the interoperability levels and their implementing profiles

Interoperability Level	Implementing Profile
Organisational	Organisation Profile
Semantic	Device Profile
Syntactic	Data Profile
Network	Network Profile

Figure 1 shows the layers in the associated implementation framework. The framework has five layers: *physical*, *profile*, *register*, *service*, and *control*. The physical layer holds all physical devices such as sensors, actuators, and controllers. The profile layer holds various types of profiles: device profile, controller profile, data profile, network profile and organisation profile. The register layer holds information about the instances such as devices and controllers that are *active* in the interoperable environment. The service layer has four types of services: *profiling*, *registration*, *transmission* and *deregistration*. The control layer contains a controller program and scheduler methods to perform continuous probing of the devices and other controllers to get data and messages and thus initiate subsequent tasks.



**Figure 1.** Edge Interoperability Framework [Azad et al., 2023], with the components implemented in this work highlighted by shaded boxes.

As shown in Table 2 right column, interoperability at various levels is achieved through specific profile parameters in our framework: network profile parameters for network level interoperability, data profile parameters for syntactic

level interoperability, device profile parameters for semantic level interoperability, and organisation profile parameters for organisational level interoperability. For this work, organisational profiles are not implemented while wired connection requires no network profile in the network level.

It is worth noting that to implement an interoperability level, all layers within the interoperability framework may need to be implemented. While the profile layer determines the specific standards and protocols required for a particular interoperability level, successful implementation may necessitate having functionalities in all layers of the framework.

#### 4.2 Program Control Flow

Prior to the execution of the controller program within the control layer, as illustrated in Figure 1, relevant profiles describing devices, controllers, and data are stored in the non-volatile memory. Also, the description of the edge environment is stored in the device and controller registers in the non-volatile storage. When the controller devices start operating, first the profiles and the registers are loaded from the non-volatile storage to the volatile storage. Then, the controller program is invoked. Within a loop, the controller program, uses the scheduler to schedule tasks and consequently execute. Later, we provide further details of the associated services, and the operational procedures of both the controller and scheduler programs and edge network design.

### 5 InterEdge Implementation

The actual implementation of the proposed theoretical framework is a significant contribution of this paper, as it not only validates the framework's applicability but also resolves critical challenges encountered during application. A key issue is the resource constraints of edge devices that are designed for mission-critical applications requiring real-time decision-making based on current data.

1. Initial attempts to implement the profile and registration layers using XML and JSON formats proved impractical due to their verbose, tag-based structure, resulting in excessive memory and processing overhead. The profiles and registers generated a total of 4KB of files for XML and 3KB of files for JSON, while the Arduino Uno R3, has only 1KB of electrically erasable programmable read-only memory (EEPROM). This limitation was addressed by adopting plain text files, significantly reducing the memory footprint and enabling efficient real-time operation.
2. Various existing interoperability implementations use various programming languages that include C/C++, Java, JavaScript, .NET, Python, and Node.js. Upon careful consideration, the InterEdge implementation is developed using the C/C++ programming language. The reason is to get a smaller executable program and greater speed. Also, C/C++ is often the primary language supported by various microcontrollers that include Arduino Uno R3 and ESP32. To get a smaller executable program, we have also adopted simple pro-

cedural programming instead of object-oriented or any other programming paradigms.

3. Creating device profiles and registers required an in-depth understanding of device operations. For syntactic interoperability, precise data formats were defined, while semantic interoperability involved embedding meaningful metadata during registration.

Overcoming above challenges demonstrates the framework's practicality and its potential to enhance edge-based IoT interoperability in resource-constrained environments.

Below we describe the implementation of the five layers of InterEdge. Although in some layers, we show the implementation for some commonly used devices, following the examples, future extensions could be easily incorporated for potential other devices. As shown in Figure 1, the components from the five layers that have been implemented are highlighted in shaded colour.

## 5.1 Physical Layer

The physical layer encompasses all the physical devices. We need controller, sensor, and actuator devices. Below we list the specific hardware used as these devices.

1. **Controller Devices:** Arduino Uno R3 microcontroller and ESP32 microcontroller
2. **Sensor Devices:** TMP36 temperature sensor and HC-SR04 ultrasonic motion sensor
3. **Actuator Devices:** Light Emitting Diode (LED) lights

The sensors and actuators are connected with controllers. The controllers are connected with each other to expand the edge network. All connections in this work are wired, although the framework [Azad et al., 2023] supports wireless connection. Arduino Uno R3 uses serial communication when connected via wire. For ESP32 the communication protocol we have used can have reliable communication up to 1 meter (3.3 feet), potentially up to 10 meters (33 feet) using I2C bus extensions. Moreover, for Arduino Uno R3, the communication protocol we have used can have reliable communication up to 15 meters (50 feet), but can be extended up to 50 meters (164 feet) using RS-232 transceivers.

## 5.2 Profile Layer

The profile layer stores detailed information about all the devices in the edge network. The information includes device type, manufacturer, configuration parameters, network protocols, data formats, and organisational affiliations.

### 5.2.1 Device Profile

We create device profiles for TMP36 temperature sensors, HC-SR04 ultrasonic motion sensors, and LED lights. These profiles include main parameters and further parameters for reading and writing data. The formula parameter in the tables is presented in postfix notation.

- **TMP36 Sensor:** Table 3 shows the parameter values for the device profile. The map function in the formula converts the raw analog readings from the temperature sensor into temperature values within the specified range of -40 to 125 degrees Celsius.

**Table 3.** Device profile for TMP36 temperature sensors

Parameter	Value
TypeID	Type001
TypeName	TMP36
Category	Sensor
SubCategory	Temperature
TransmissionMode	Read
Connectivity	Wired
DataType	Int
DataLength	3
MeasuringUnit	°C
NeedFormula	Yes
Formula	data 20 - 3.04 * 0 1023 -40 125 map
ReadMethod	ReadData

Table 4 shows the parameter values for the ReadData method.

**Table 4.** Values for ReadData method for TMP36

Feature	Parameter	Value
Initialization	noOfPin	1
	pins	TMP_PIN
	pinMode	INPUT
Read	pinMode	INPUT
	digitalWrite	NA
	Delay	NA

- **HC-SR04 Sensor:** Table 5 shows the the parameter values for the device profile. Table 6 shows the parameter values for the ReadData method.

**Table 5.** Device profile for HC-SR04 sensors

Parameter	Value
TypeID	Type002
TypeName	Ultrasonic
Category	Sensor
SubCategory	Distance
TransmissionMode	Read
Connectivity	Wired
DataType	Float
MeasuringUnit	CM
NeedFormula	Yes
Formula	data 0.034 * 2 /
ReadMethod	ReadData

- **LED Light:** Table 7 shows the parameter values for the device profile. The parameter values for the LED light device profile are shown in . Table 8 shows the parameter values for the WriteData.

**Table 6.** Values for ReadData method for HC-SR04

Feature	Parameter	Value
Initialization	noOfPin	2
	pins	TRIG_PIN, ECHO_PIN
	pinMode	[OUTPUT], [INPUT]
Read	pinMode	[], []
	digitalWrite	[LOW, HIGH, LOW]
	delayMicroseconds	[2,10], []
	pulseIn	[], [HIGH]

### 5.2.2 Controller Profile

We create controller profiles for Arduino Uno R3 and ESP32 microcontrollers. The controller profile provides detailed information about the controller: main parameters as well as configuration parameters.

**Table 7.** Device profile for LED lights

Parameter	Value
TypeID	Type003
TypeName	LED
Category	Sensor
SubCategory	Light
TransmissionMode	Write
Connectivity	Wired
DataType	Boolean
MeasuringUnit	Boolean
NeedFormula	No
WriteMethod	WriteData

**Table 8.** Values for WriteData method for LED

Feature	Parameter	Value
Initialization	noOfPin	1
	pins	LEDPIN
	pinMode	OUTPUT
Write	pinMode	OUTPUT
	digitalWrite	NA

- **Arduino Uno R3 Microcontroller:** Table 9 shows the parameter values for Arduino Uno R3 microcontrollers.

**Table 9.** Controller profile for Arduino Uno R3

Parameter	Value
TypeID	ContType001
TypeName	Arduino Uno R3
Category	AVR microcontroller
Connectivity	Wired
NetworkProfile	N/A
OrganisationProfile	N/A

- **ESP32 Microcontroller:** Table 10 shows the parameter values for ESP32 microcontrollers.

**Table 10.** Controller profile for ESP32

Parameter	Value
TypeID	ContType002
TypeName	ESP32
Category	Xtensa microcontroller
Connectivity	[Wired, Wi-Fi, Bluetooth]
NetworkProfile	N/A
OrganisationProfile	N/A

### 5.2.3 Data Profile

We create data profiles for each type of data to be read or written. However, microcontrollers Arduino Uno R3 and ESP32, sensors TPM36 and HC-SR04, and LEDs all deal with digital signals. The data profile is associated with a device when the device is registered to the network.

- **Digital Signal:** Table 11 shows the parameter values for the data profile.

**Table 11.** Data profile for Digital Signal

Parameter	Value
Id	Format001
Name	Digital Signal
Version	DS01
DataType	Digital Signal
Encoding	N/A

### 5.2.4 Network Profile:

As already mentioned before, in this work, for the time being, wired connection was assumed to be between edge devices. The microcontrollers used the serial communication. We did not implement any network interoperability in this work. Therefore, no network profile was created.

### 5.2.5 Organisation Profile:

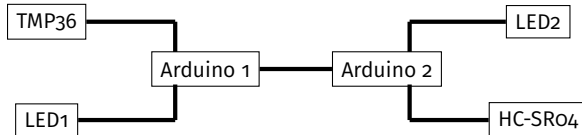
No organisational level interoperability was implemented in this work. Therefore, no organisation profile was created. To implement the syntactic and semantic level interoperability, only one type of microcontrollers (either Arduino Uno R3 or ESP32) and basic raw sensors (TMP36 and HC-SR04) and actuators (LEDs) were used in this work; all of these just use digital signals. Organisational profiles would have been needed if we would have used microcontrollers, sensors, and actuators from various manufacturing organisations together.

## 5.3 Register Layer

The register layer maintains a repository of active devices and controllers connected in the IoT network. Note that the registration information includes some fields that are also found in the profiles. This duplication stems from the subtle distinction that profiles often list a range of potential options for a device or controller while only a specific subset of these options might be actively utilised in a given connection to the environment. So the redundancy allows for flexibility, enabling the system to adapt to varied scenarios by selecting and employing the most relevant configurations.

Note that the registration layer integrates particular devices and controllers within the IoT environment by storing a predefined configuration in the controllers. Disconnected devices must re-register upon reconnection. This static setup enhances security, allowing only pre-registered devices to interact and requiring physical access for new registrations, thus preventing unauthorised access.

Since registration involves an edge environment, we show an example in Figure 2. The example has controller to device connections as well as controller to controller connections.



**Figure 2.** An example edge environment with two controllers (Arduino Uno R3), two sensors (TMP36 and HC-SR04), and two actuators (LEDs). All six devices to be registered.

### 5.3.1 Device Register

We create a device register for each instance of a device type and give a unique DeviceId. For example, two device registers with two unique DeviceIds will be created for two TMP36 sensors. For the four sensor and actuator devices in Figure 2, we will create four device registers: one for TMP36 temperature sensor, one for HC-SR04 ultrasonic distance sensor, and two for LED lights.

- **TMP36 Sensor:** Table 12 shows the parameter values of the device register for TMP36 Sensor.

**Table 12.** Device register for TMP36 Sensor

Parameter	Value
DeviceId	TMP36001
DeviceName	TMP36 Temperature Sensor
TransmissionMode	Read
Connectivity	Wired
DeviceProfile	Type001
DataProfile	Format001
NetworkProfile	N/A
OrganisationProfile	N/A
ActualValue	N/A
ConvertedValue	N/A

- **HC-SR04 Sensor:** Table 13 shows the parameter values of the device register for HC-SR04 Sensor.

**Table 13.** Device register for HC-SR04 Ultrasonic Sensor

Parameter	Value
DeviceId	HCSR04001
DeviceName	HC-SR04 Ultrasonic Sensor
TransmissionMode	Read
Connectivity	Wired
DeviceProfile	Type002
DataProfile	Format001
NetworkProfile	N/A
OrganisationProfile	N/A
ActualValue	N/A
ConvertedValue	N/A

- **LED Lights:** We have two LED lights in Figure 2. So we need one device register for each LED light. Table 14 shows the parameter values for the device register for LED1 (or LED2). In this case, the two device registers mainly differ by their DeviceIds.

**Table 14.** Device register of LED1 (or LED2) Light

Parameter	Value
DeviceId	LED001 (or LED002)
DeviceName	LED Lights
TransmissionMode	Write
Connectivity	Wired
DeviceProfile	Type003
DataProfile	Format001
NetworkProfile	N/A
OrganisationProfile	N/A
ActualValue	N/A
ConvertedValue	N/A

**Table 15.** Controller register for Arduino Uno 1

Parameter	Value
ControllerId	Arduino001
ControllerName	Arduino Uno R3
ConnectedDevices	Arduino001-TMP36001 Arduino001-LED001
ConnectedControllers	Arduino002
Connectivity	Wired
NetworkProfile	N/A
OrganisationProfile	N/A
SourceDevices	Arduino001-TMP36001
ConditionActions	data 30°C > high low ?
DestinationDevices	Arduino001-Arduino002-LED002
DeviceProbeIntervals	(00:00:05)
MessageBuffer	NULL
ControllerProbeIntervals	(00:00:05)

**Table 16.** Controller register for Arduino Uno 2

Parameter	Value
ControllerId	Arduino002
ControllerName	Arduino Uno R3
ConnectedDevices	Arduino002-HCSR04001 Arduino002-LED002
ConnectedControllers	Arduino001
Connectivity	Wired
NetworkProfile	N/A
OrganisationProfile	N/A
SourceDevices	Arduino002-HCSR04001
ConditionActions	data 15cm > high low ?
DestinationDevices	Arduino002-Arduino001-LED001
DeviceProbeIntervals	(00:00:05)
MessageBuffer	NULL
ControllerProbeIntervals	(00:00:05)



### 5.3.2 Controller Register

We create two controller registers for the two Arduino Uno R3. Each controller register gets its unique ControllerId. Note that we could replace the two Arduino Uno R3 microcontrollers with two ESP32 microcontrollers. As mentioned before, two different microcontrollers cannot be used simultaneously in this work, since we have not implemented organisational interoperability.

- **Arduino Uno R3 Controllers:** Table 15 and Table 16 show the parameter values for two controller registers. Notice that each controller keeps the other controller in its ConnectedControllers field.

## 5.4 Service Layer

The service layer encompasses four types of services: profiling, registration, transmission, and deregistration.

1. **Profiling:** We created the profiles for devices, controllers, and data. We uploaded them into the non-volatile memory of the respective microcontrollers. Then, the profiling function is run before the controller program is invoked to load these profiles from non-volatile memory into volatile memory.
2. **Registration:** We created registers for the devices and the controllers. We uploaded them into the non-volatile memory of the respective microcontrollers although the framework [Azad et al., 2023] supports registration information to be kept in the random access memory. After the profiling function is called before the controller program is invoked, the registration function is called to load the registers from non-volatile memory into volatile memory, ensuring seamless initialisation and operation with the active devices.
3. **Transmission:** Seven methods are used in transmission. ReadData method uses device registers, device profiles, and data profiles to connect to devices, read data, and process. NeedAction method uses controller registers to determine necessary actions based on conditions and artificial intelligence algorithms. WriteData method uses device registers, device profiles, and data profiles to write data to devices after processing. NeedRouting method uses controller registers to decide if data needs to be routed through other controllers. WriteMessage uses controller registers, and controller profiles, device registers, device profiles, and data profiles to create messages for transmission across the network. RouteMessage uses controller registers to send messages between controllers. ReadMessage uses controller registers to read messages from other controllers. We have implemented all the aforementioned methods as their algorithms are in our previous paper [Azad et al., 2023].
4. **Deregistration:** We remove the device information from the controller register when a device is no longer needed. In this implementation, deregistration takes place when updated controller register is uploaded to the non-volatile memory of the controller although the

framework [Azad et al., 2023] supports keeping registration information in the random access memory.

## 5.5 Control Layer

In the control layer, a ControllerProgram and scheduling methods such as InitSchedule, UpdateSchedule, DataReadingScheduled, and MessageReadingScheduled ensure continuous data and message retrieval from devices and controllers. Based on set intervals, the controller schedules and reads data using ReadData, determines actions with NeedAction, and decides routing with NeedRouting. Data is either written directly to devices with WriteData or routed through controllers using WriteMessage and RouteMessage. Controllers regularly check their MessageBuffer for new messages, handling them to ensure proper data delivery.

We have implemented all the aforementioned methods as their algorithms are in our previous paper [Azad et al., 2023]. Note that the interoperability framework in our previous paper [Azad et al., 2023] and in this implementation, the ControllerProgram within a loop continually update the schedule and as per the schedule, synchronously probes devices and performs actions. To support asynchronous scheduling of read or write actions, a synchronously probed switch with a very short time interval could be used as a source sensor and the required read or write actions on the destination device could be scheduled for immediate or next occurrence.

## 5.6 Software Architecture

This implementation employs event-driven programming. A loop in the controller program continually makes probes to the devices and other controllers to see whether an event is to be triggered and responded. For software organisation, this implementation just uses a procedure-based architecture, wherein all functionalities are implemented as straightforward procedures, avoiding the complexity and overhead associated with more intricate software architectures. This design choice aims to preserve simplicity and minimise computational overhead, thereby enhancing overall performance.

## 6 Implementing Edge Networks

We describe how to design an arbitrary edge network using InterEdge. Then, we show two scenarios: one with single controller and the other with multiple scenarios.

To evaluate our implementation, we consider two scenarios: one with just one controller needing no message routing and another one with multiple controller needing message routing. These two scenarios demonstrate the flexibility and scalability of our solution. A microcontroller usually can be connected to a maximum number of devices. So multiple microcontroller setup is particularly needed when managing comparatively large numbers of sensors and actuators.

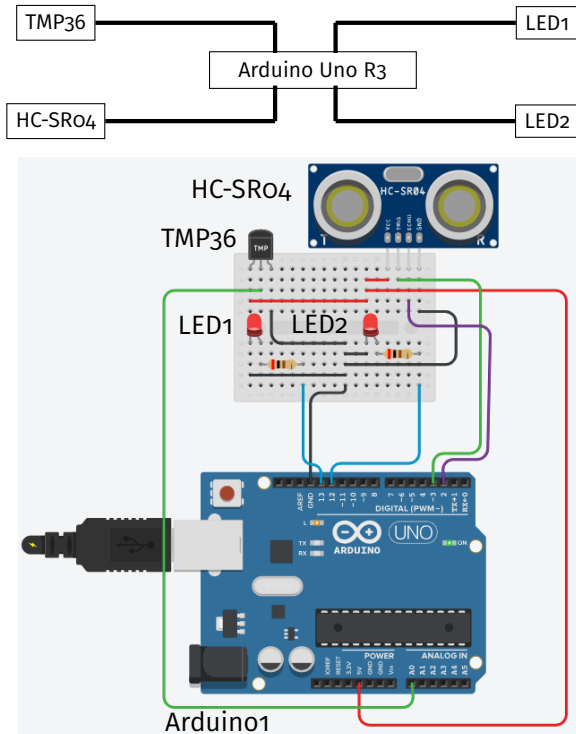
## 6.1 Generic Edge Network

Achieving interoperability is generalised by creating device and controller profiles for the device types. For instance, profiles for TMP36 sensors, HC-SR04 sensors, LED lights, Arduino Uno R3, and ESP32 are created in this implementation. They can now be used in any scenario involving these devices and controllers. However, if any new device type is to be considered, we will have to create a profile for that device.

To design an edge network, device and controller profiles are stored in the controllers's EEPROMs. Then, the controllers are registered in the network using the controller registers. This is followed by the sequential registration of the devices using the device registers. When a controller reaches its capacity, another controller is added to the network, and additional devices are registered with that. In an edge network, communication between devices works only when the connection is active. For wired connections, this is naturally maintained as the connection is consistently active. However, for wireless connections, such as Wi-Fi, the network would require the connection to remain active to facilitate communication among devices.

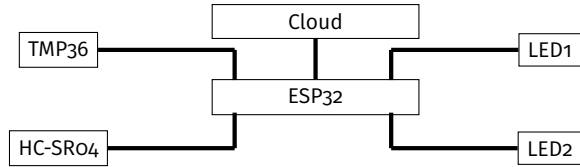
## 6.2 Single Controller Scenario

Figure 3 shows a controller (Arduino Uno R3) connected with two sensors (TMP36 and HC-SR04) and two LEDs.



**Figure 3.** Single Controller Scenario: Arduino Uno R3 connected with two sensors TMP36 and HC-SR04 and two LEDs. Top: schematic diagram and Bottom: simulator generated image.

In Figure 3, the controller reads the temperature from the temperature sensor TMP36. If the temperature value is greater than a threshold value then the controller sends an ON signal else an OFF signal to one LED. Also, the controller reads the distance from the ultrasonic distance sensor.



**Figure 4.** Single Controller Scenario: ESP32 connected with cloud, two sensors TMP36 and HC-SR04 and two LEDs.

If the distance value is greater than a threshold value than it sends an ON signal else an OFF signal to another LED. We can configure the controller to read data in regular intervals.

Note that the controller is typically equipped with sufficient processing capacity. It holds profiles, registers, a controller, and a scheduler program to perform essential functions such as data transformation, protocol translation, and service delivery for seamless interoperability. Further, note that the controller typically facilitates communication as a central intermediary since devices themselves may not have processing capacity. However, direct device-to-device communication is possible when devices have adequate processing capabilities i.e. the same physical device supports both device and controller functions. In such cases, device profiles and registers are maintained within the devices, and the controller's functionality is embedded in them, enabling decentralised communication and enhanced flexibility.

While Figure 3 shows our implementation, the three open standard solutions that we later compare with need cloud access as shown in Figure 4. Since Arduino Uno R3 microcontrollers do not support internet connection to the cloud, in this scenario an ESP32 or another microcontroller is used.

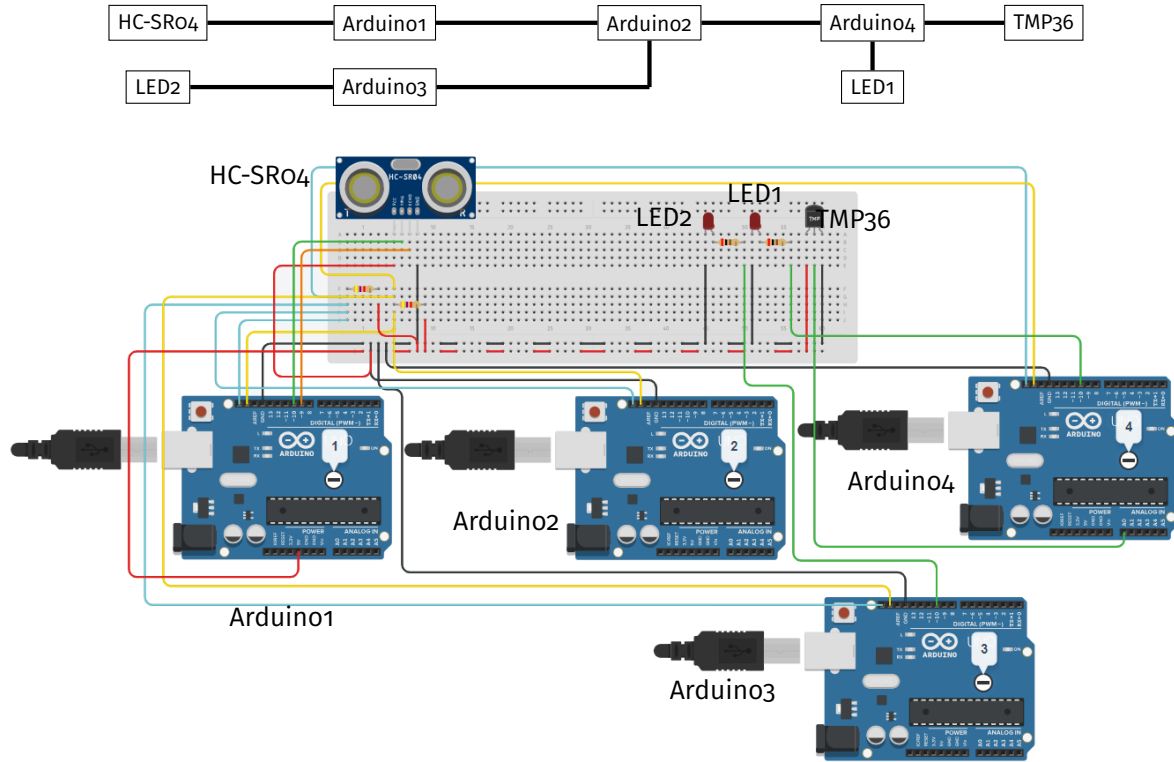
For our implementation, we use the online simulator Wokwi (<https://wokwi.com/>) to set up IoT edge networks. This simulator allows us to create and test our microcontroller configurations without the need for physical hardware, providing a flexible and accessible platform for development. Moreover, it allows us to design, simulate, and debug Arduino Uno R3 and ESP32 projects in a web-based environment. The programming languages in the simulator are variants of C++. Our implementation can be found from (<https://wokwi.com/projects/396638811657719809>).

## 6.3 Multiple Controller Scenario

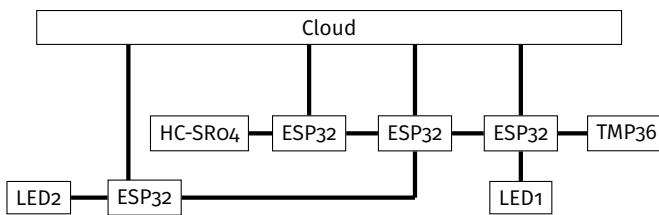
Figure 5 shows four controllers (all are Arduino Uno R3) connected with two sensors (TMP36 and HC-SR04) and two LEDs. All the controllers are replaced by ESP32 microcontrollers when we run experiments with ESP32 type.

In Figure 5, controller Arduino1 reads distance from the ultrasonic distance sensor HC-SR04 and if the distance is greater than a threshold value then it sends an ON signal else an OFF signal to LED1 connected with controller Arduino4 via another controller Arduino2. Also, controller Arduino4 reads temperature from the temperature sensor TMP36 and if temperature is greater than a threshold value then it sends an ON signal else an OFF signal to the LED connected with controller Arduino3 via another controller Arduino2. Table 17 shows how the routing information is stored in the controller registers.

While Figure 5 shows our implementation, the three open



**Figure 5.** Multiple Controller Scenario: Four Arduino Uno R3 microcontrollers are connected with two sensors TMP36 and HC-SR04 and two LEDs. Top: schematic diagram and Bottom: simulator generated image.



**Figure 6.** Multiple Controller Scenario: Four Controllers are connected with cloud, two sensors TMP36 and HC-SR04 and two LEDs.

standard solutions we later compare with need cloud access (and supporting microcontrollers such as ESP32) as in Figure 6.

**Table 17.** Routing information for data transmission in Figure 5. (Top) from HC-SR04 to LED1 and (Bottom) from TMP36 to LED2. Arduino1-4 are denoted by A1-4 and HC-SR04 by HC. Each row is stored in the controller register of the corresponding controller mentioned in the left most column named Controller; the other two columns Source Device and Destination Device are stored in the SourceDevices and DestinationDevices in the respective controller registers.

Controller	Source Device	Destination Device
A1	A1-HC	A1-A2-A4-LED1
A2	A2-A1-HC	A2-A4-LED1
A4	A4-A2-A1-HC	A4-LED1
Controller	Source Device	Destination Device
A4	A4-TMP36	A4-A2-A3-LED2
A2	A2-A4-TMP36	A2-A3-LED2
A3	A3-A2-A4-TMP36	A3-LED2

The Wokwi online simulator can support different types of microcontroller such as Arduino, ESP32, Raspberry Pi, but it

cannot support multiple controllers in one project. So for our multiple controller scenario, we have used another simulator <https://www.tinkercad.com/> and our implementation is available from <https://www.tinkercad.com/things/i8c4z51qvzQ-syntactic-n-semantic>.

## 6.4 Comparison with Other Implementations

We compare our implementation of the single and multiple controller scenarios described in the previous subsections with the implementation work to be required by other solutions such as WoT, Wasm, and IoTivity.

**Common Hardware Setup:** Each framework requires devices such as TMP36 and HC-SR04 sensors, and LED lights. Raw sensors lack processing power. So a gateway/controller (e.g., ESP32) is necessary to process sensor data, make decisions, and send commands to the LEDs. The Arduino Uno R3 is not compatible due to its limited resources.

**WoT Software Setup:** WoT uses JSON-LD-based Thing Descriptions (TD) to define device properties and actions. The ESP32 reads sensor data, exposes it via HTTP/MQTT, and controls LEDs based on predefined conditions. In multi-controller scenarios, users must manually configure data routing, as the WoT framework does not specify inter-controller communication.

**Wasm Software Setup:** Wasm compiled modules manage sensor readings and LED controls. The ESP32 loads and executes these modules, handling interactions with hardware.

Communication between modules or external systems requires custom implementation, as Wasm does not define data routing between multiple controllers.

**IoTivity Software Setup:** IoTivity represents sensors and LEDs as resources within its framework. The ESP32 registers these resources, processes CoAP requests for sensor data, and sends CoAP requests to control LEDs based on conditions. While resource discovery and interaction are managed by IoTivity, data routing between multiple controllers is not defined by the framework.

**Overall Comments:** From the above discussion, it is evident that this implementation of InterEdge has more advantages as it supports Class-0 devices, includes an integrated controller, and needs less custom coding. In contrast, WoT, Wasm, and IoTivity do not support Class-0 devices, lack the concept of an integrated controller, and need significant amount of coding. Implementation of data routing in these frameworks demands extra configuration and coding whereas InterEdge and this implementation seamlessly supports these functionalities within its framework design.

## 7 Experimental Results

We describe our evaluation metrics, competitor solutions, and provide performance analysis and discussion.

### 7.1 Evaluation Metrics

In the literature, a related paper [Jazayeri *et al.*, 2015] has evaluated the performance of four interoperable open standard solutions OGC PUCK over Bluetooth [O'Reilly *et al.*, 2012], TinySOS [Jazayeri *et al.*, 2012], SOS over CoAP [Jazayeri *et al.*, 2015], and OGC SensorThings API [Liang *et al.*, 2021]. In order to evaluate each solution, it has used a service prototype (i.e., server), a gateway, and a client. As metrics, it has used (i) code storage in read only memory (ROM, EEPROM), (ii) main memory (RAM) usage, (iii) request length of an operation, (iv) response size of an operation, (v) and response latency.

Our solution differs from existing solutions in terms of data communication. Existing solutions require making API calls with XML, JSON, or plain text data requests to the cloud and then retrieving data from the cloud. In contrast, our solution involves direct communication between the controller and the device, bypassing the cloud entirely. Consequently, metrics like request size and response size, which are relevant for cloud-based systems, are not applicable to our solution as there is no API call or cloud data transmission. In this implementation, we have considered wired connection only. Our solution eliminates the need for data requests by directly sending signals from the controller to the device. So we can evaluate our implementation with three metrics (i) ROM storage, (ii) RAM usage, and (iii) response latency. These metrics are suitable for our solution's architecture. This is because the elimination of API calls and cloud data transmission in our solution reflects a more streamlined

approach to edge device communication, tailored specifically for resource-constrained environments.

### 7.2 Competitor Solutions

We target our solution to work both on class-0 and class-1 edge devices [Bormann *et al.*, 2012]. Class-0 devices such as Arduino Uno R3 have only 32KB of random access memory (RAM) and 2KB of code space. Class-1 devices such as ESP32 have about 10KB of RAM and 100KB of code space.

We have run the three available open standard solutions—OGC SOS [Na and Priest, 2007], OGC SensorThings API [Liang and Khalafbeigi, 2019], and EdgeX Foundry [Foundry, 2021]. These solutions support edge interoperability, but using cloud access needing comparatively more resources. So they cannot operate on class-0 devices. We have used class-1 device ESP32 microcontroller for them. ESP32 has low-power consumption and integrated Wi-Fi and Bluetooth capabilities. This facilitates seamless connectivity, enabling devices to establish connections with local networks and internet. Note that SOS [Na and Priest, 2007] only supports data sensing operations. On the other hand, OGC SensorThings API [Liang and Khalafbeigi, 2019] and EdgeX Foundry [Foundry, 2021] support both data sensing and actuator tasking but in two different parts: Part 1 and Part 2 respectively. However, we cannot run their actuator tasking in Part 2 because of the unavailability of the program and documentations. So our comparison with the three open standards is mainly restricted within the data sensing operations.

We have selected the above-mentioned three solutions for comparison based on specific criteria to ensure a meaningful evaluation of our solution against existing standards.

- First, the selected solutions must support edge interoperability, which is essential for the targeted edge devices.
- Second, if a solution supports both edge and cloud interoperability, its architecture should be simple enough to be callable from a class-0 microcontroller device, reflecting the resource constraints typical of such environments Open Connectivity Foundation (OCF) [Park, 2017], OneM2M [Park *et al.*, 2016], FIWARE [Cirillo *et al.*, 2019].
- Lastly, the availability of open-source code was a crucial factor, enabling direct implementation and testing.

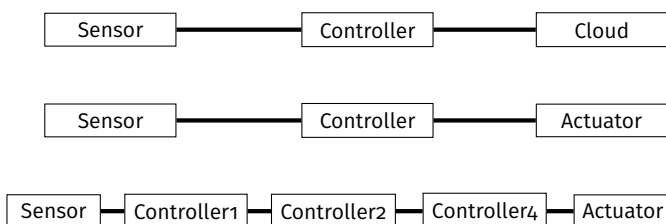
While there are other existing interoperable solutions that address interoperability in IoT edge devices, they do not fully meet all these criteria and, therefore, were not included in our comparison. Further, while the WoT by W3C [Gyrard *et al.*, 2017] provides an implementation that supports Arduino microcontrollers, the implementation is specifically designed for the Arduino Mega. The Arduino Mega requires 8KB of RAM, 256KB of Flash memory, and 4KB of EEPROM, which are significantly higher resource capacities compared to the Arduino Uno R3 used in our implementation of the InterEdge solution.

### 7.3 Evaluation Test Cases

Figure 7 presents the schematic diagram of the experimental setup used to compare our solution with three open standard solutions. Figure 7 (top) illustrates the schematic diagram for the three open standards. To evaluate the performance of our solution in alignment with the three open standards, our analysis focused on the process of reading sensor data. For the OGC SOS [Leibovici *et al.*, 2023], this involved a twofold procedure: the insertion of observations (Insert Observation) into the SOS cloud followed by their retrieval (Get Observation). These functions were assessed separately due to their distinct roles within the OGC SOS framework. Within the OGC SensorThings API [Liang *et al.*, 2021], evaluations were conducted on both the creation (Create Observation) and retrieval (Read Observation) of observations. Likewise, in EdgeX Foundry, we scrutinised the functionalities associated with sending data (Send Data) and reading data (Read Data). For convenience, throughout the rest of the paper, we will refer to SOS (Insert Observation) as SOS (Cloud Upload), SOS (Get Observation) as SOS (Cloud Download), SensorThings API (Create Observation) as SensorThings API (Cloud Upload), SensorThings API (Read Observation) as SensorThings API (Cloud Download), EdgeX (Send Data) as EdgeX (Cloud Upload), and EdgeX (Read Data) as EdgeX (Cloud Download). For the three standard solutions, ‘writing’ refers to reading data from the sensor, sending a request to the cloud to store the data, and receiving a success response (Cloud Upload). In contrast, for our solution, ‘writing’ refers to sending a command directly to an actuator. Similarly, for the three standard solutions, ‘reading’ involves sending a request to the cloud to retrieve previously stored data and successfully reading the data from the cloud (Cloud Download). For our solution, ‘reading’ refers to directly reading the sensor data.

Figure 7 (middle) depicts our implementation for comparison. In our setup, the controller directly reads data from the sensor and writes data to the actuator. The actuator is utilised solely for analysing our experimental results and is not employed in the three open standards. In our solution, where a sensor or actuator is directly connected to a microcontroller, the evaluation process involves only direct data retrieval.

Figure 7 (bottom) depicts our implementation for both read and write data in the multiple controller scenario. The online simulator utilised for implementing the multiple controller scenario supports only Arduino microcontrollers. Consequently, we were able to demonstrate the results of the multiple controller experiment exclusively using Arduino microcontrollers.



**Figure 7.** Test cases: (top) for data sensing for the three open standard solutions, (middle) for both data sensing and actuator tasking for our solution in a single controller scenario and (bottom) in a multiple controller scenario.

### 7.4 Performance Comparison

As mentioned before, we show the performance comparison using ROM usage, RAM usage, and response latency.

#### 7.4.1 ROM Usage

Table 18 shows the ROM usage by various solutions. The SOS (Cloud Upload) needs the most ROM in comparison with other implementations. This is because it needs to send a big chunk of data for insertion of the sensor data. The SOS (Cloud Download) and SensorThings API (Cloud Upload) take similar sizes of ROM. This is because the size of XML data is used for the SOS (Cloud Download) request and the size of json data used for SensorThings API (Cloud Upload) are similar. For SensorThings API (Cloud Download), EdgeX (Cloud Upload), and EdgeX (Cloud Download) use similar sizes of ROM. SensorThings API (Cloud Download) requires a bit less ROM than SOS (Cloud Download) because the size of data SensorThings API (Cloud Download) sends to retrieve sensor data is less than the size of data SOS (Cloud Download) sends to retrieve sensor data.

**Table 18.** ROM and RAM usage (in KB)

Solution Name	ROM	RAM
SOS (Cloud Upload)	886.509	45.864
SOS (Cloud Download)	883.745	45.848
SensorThings API (Cloud Upload)	883.885	45.864
SensorThings API (Cloud Download)	882.605	45.64
EdgeX (Cloud Upload)	882.685	45.656
EdgeX (Cloud Download)	882.477	45.640
InterEdge Single Controller (Read-Write) (ESP32)	274.992	35.012
InterEdge Single Controller (Read-Write) (Android)	6.966	0.801
InterEdge Multiple Controllers (Read-Write) (Android)	7.916	0.970

Sizes are for each controller in multiple controller scenarios

For our solution, we see that for single scenario the same code occupies more space in ESP32 than Arduino Uno R3 implementation. This is because the AVR libraries employed in Arduino Uno R3 are simple and small, which is in stark contrast to the larger and more intricate libraries tailored for ESP32 microcontrollers. Consequently, the utilisation of ESP32 necessitates a greater allocation of program memory due to the expanded functionality and feature set encapsulated within its libraries. The ROM and RAM usage for multiple controller scenario have been shown for each controller. The ROM and RAM usage in the multiple controller Arduino scenario are almost similar to, but slightly higher than, those in our single-controller Arduino implementation.

Nevertheless our solution exhibits a lower ROM footprint when compared to the other three open standards. Additionally, within the Arduino framework, our solution demonstrates a notably reduced utilisation of ROM resources. This observation underscores the efficiency and resource optimisation inherent in our solution, particularly within the context of Arduino-based implementations.

### 7.4.2 RAM Usage

Table 18 shows the RAM usage during the execution of each implementation. Since all three open standards primarily involve calling URLs with POST or GET requests, their RAM usage is quite similar. However, our ESP-based solution consistently utilises less RAM compared to the other three open standards. Additionally, the Arduino implementation of our solution requires remarkably small RAM resources, indicating its memory efficiency. In our multiple controller scenario, each controller requires slightly more RAM compared to what is required by each controller in our single controller scenario (Arduino).

### 7.4.3 Response Latency

We have recorded the end-to-end response latency in our study. Using a PC client, we conducted experiments to retrieve sensor data from both an ESP32 and an Arduino Uno R3 for our solution. Latency for the three open standards has been defined as the duration between the PC client sending a request to the cloud and receiving the response. In our scenario, response latency has been calculated from the moment the controller initiated data reading from a sensor until it completed reading the data. Additionally, for our solution, we have recorded the time taken to send a command to an actuator. In this case, response latency has been calculated from the moment the controller started checking the sensor data to determine if an action has been required until it finished sending the command to the actuator.

Figure 8 shows the response latency vs the number of readings/writings request for the competitor solutions. We see that the SOS (Cloud Upload) operation exhibited the longest response time. This can be attributed to the greater size of XML data transmitted compared to other implementations. For a lower number of sensor readings or writings, SensorThings API (Cloud Upload) demonstrated shorter response times than SOS (Cloud Download). However, for over 80 sensor readings or writings, SensorThings API (Cloud Upload) took longer. Notably, EdgeX implementations exhibited comparatively shorter response times than SOS and SensorThings API implementations. Specifically, EdgeX (Cloud Download) operation exhibited shorter response times than (Cloud Upload), as it only accessed the cloud once to retrieve all the data at once. Thus, any increase in time was solely due to the additional data retrieved for higher numbers of sensor readings. However, for EdgeX implementations, we do not mention any time when we are sending a data or which time data we are retrieving, we only send one data or two data or more and we retrieve one data or two data or more. In our multiple controller scenario, the WriteData operation exhibits a longer response time compared to the ReadData operation. This discrepancy can be attributed to the implementation specifics of the online simulator. The simulator employs a predefined method for invoking a microcontroller and for executing read or write operations to or from that microcontroller. Consequently, there is an inherent wait time associated with these operations, which contributes to the increased duration observed for the WriteData process. Both our ESP32 and Arduino single scenario imple-

mentations for read and write data are better than the existing solutions. Moreover, they exhibit similar response times, indicating similar performance across the two platforms.

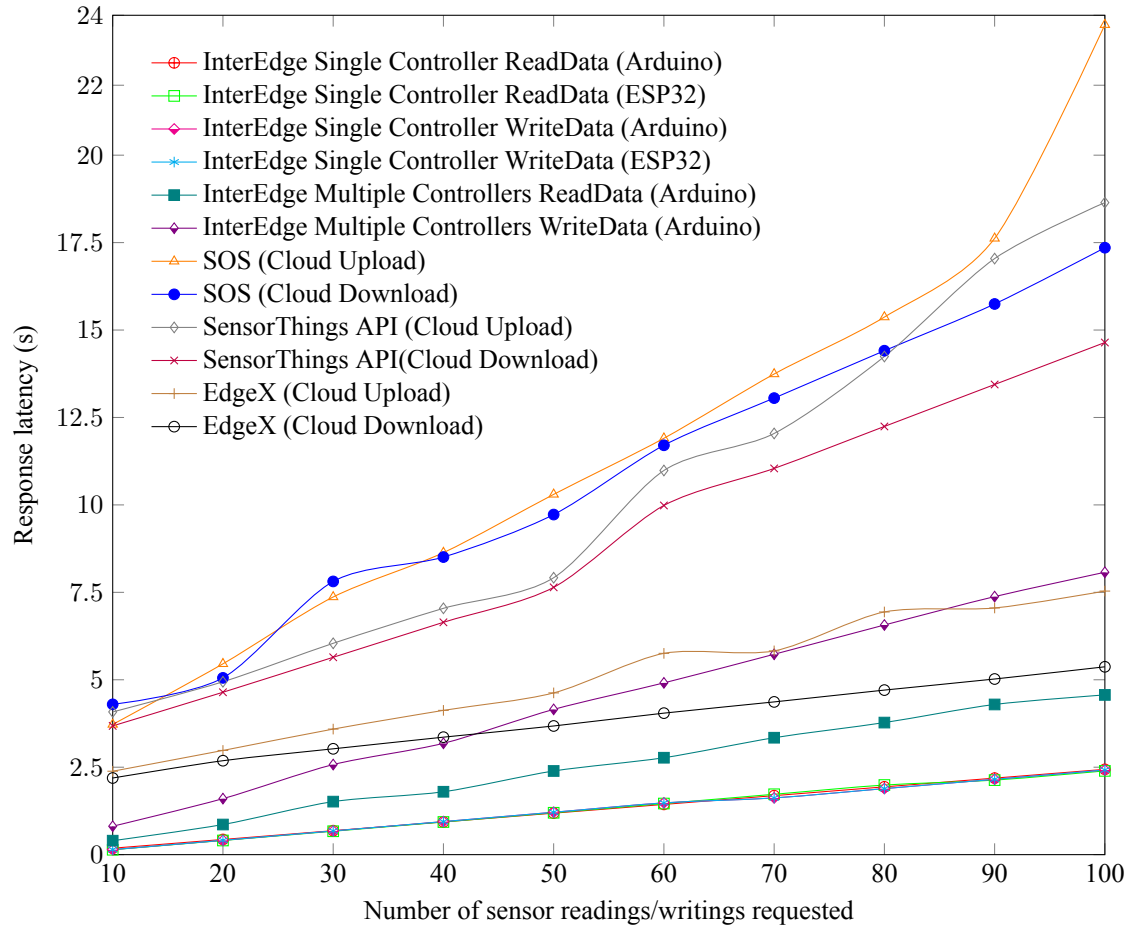
### 7.4.4 Scalability Performance

Table 19 shows running times of various operations. The profiles and registers are read once and saved in the RAM once when the program starts (setup). The per-unit times are for each device. So the total time for each operation will be setup time plus the number of devices multiplied by the per-unit time. The controller-to-controller operation is to transmit the message from one controller to the next one. The scheduling operations include initialisation (setup) before the controller program is run and then updating the schedule in each lap of the continually running loop (per-unit). For the same operation, in general, ESP32 microcontroller needs more execution time than Arduino Uno R3. This is because the ESP32 libraries require more memory. Table 20 shows the memory usage in bytes for each device.

**Table 19.** Execution time (milliseconds) for various operations. Setup times do not depend on the number of devices while per-unit time is for each device. For scheduling operation, setup is the initialisation before the loop and per-unit is for each schedule update. For ESP32, multiple controllers are not supported by the online simulator and so controller-controller time is not shown.

Operation	setup	per unit
<b>ESP32</b>		
Load Device Profile	11	368
Load Controller Profile	11	347
Load Data Profile	11	340
Load Device Register	11	375
Load Controller Register	11	392
Scheduling Operation	5	407
Read Data	11	2
Write Data	11	1
<b>Arduino Uno R3</b>		
Load Device Profile	6	285
Load Controller Profile	6	160
Load Data Profile	6	116
Load Device Register	6	327
Load Controller Register	6	430
Scheduling Operation	3	439
Read Data	6	2
Write Data	6	1
Controller-Controller	6	10





**Figure 8.** Response latency vs. number of sensor readings/writings. Note that the four lines for our single scenario solutions are almost coincidental on each other and are at lower part of the chart.

**Table 20.** Memory Usage (in bytes) per unit.

Category	Device	Size
Device Profile	TMP36	542
	LED	386
Controller Profile	Arduino Uno R3	275
	ESP32	284
Data Profile	Digital Signal	183
Device Register	TMP36	437
	LED	402
Controller Register	Arduino Uno R3	493
	ESP32	562

The scalability of the solution essentially depends on various factors such as resource availability, processing power, and the number of devices and controllers in the network. With the information in Tables 19 and 20, we can predict the solution's performance, given the numbers of various devices. This information is critical for making informed decisions about hardware selection, and network architecture design, ensuring that the IoT system can scale efficiently while maintaining optimal performance. Nevertheless, the most frequent operations are read, write, and controller-controller transmission. The execution times for these operations are  $\leq 10\text{ms}$ . Moreover, the memory requirement for various devices are in the range of hundreds of bytes. With these low number, the solution is expected to be scalable when the number of devices increases.

## 7.5 Overall Discussion

Our proposed solution demonstrates superior performance compared to the other three open standards across multiple metrics. In terms of ROM usage, our solution exhibits a significantly lower footprint, especially within the Arduino framework, highlighting its efficiency and resource optimisation. For RAM usage, our ESP-based implementation consistently utilises less memory, and the Arduino version requires remarkably minimal RAM resources, showcasing its memory efficiency. When it comes to response latency, our solution maintains comparable performance across both ESP32 and Arduino platforms, with notably shorter response times than the SOS and SensorThings API implementations. The RAM utilisation demonstrates near-constant behaviour for a specific controller type, while ROM usage exhibits variability dependent on the quantity of connected devices. Furthermore, the response latency maintains consistency across instances of the same controller type. The direct communication method employed in our solution eliminates the need for extensive data transmission, further enhancing its overall performance. Thus, our solution clearly outperforms the other standards in terms of ROM and RAM efficiency, as well as response latency. Since the scalability of our solution relies on resource availability, processing power, and the number of devices and controllers, it can efficiently scale as the number of devices increases.

## 8 Conclusion

In this paper, we have presented an implementation of our previously proposed hierarchical decentralised edge interoperability framework, which we have named InterEdge. Our solution shows significant strides in addressing the interoperability challenges within the Internet of Things (IoT) landscape. By adopting a hierarchical interoperability model and a layer-based implementation for seamless communication among resource-constrained IoT edge devices, we have successfully demonstrated enhanced efficiency, mobility, and flexibility in IoT interoperability. Our solution emphasis on local decision-making at the edge level, which distinguishes it from conventional cloud-based approaches, empowering edge devices to contribute actively to IoT developments. The contribution of this paper lies in the rigorous implementation and evaluation of the syntactic and semantic interoperability layers within the InterEdge framework. The results from our implementation clearly demonstrate the superior performance of InterEdge in comparison to existing open standards, particularly in terms of code size, memory usage, response latency, and scalability. Looking ahead, our focus remains on refining and expanding the implementation to accommodate evolving IoT requirements, particularly in implementing network and organisational interoperability, thereby advancing the landscape of IoT interoperability.

## Acknowledgements

### Authors' Contributions

Tanzima Azad: Conceptualized the study, developed the methodology, performed the analysis, and wrote the original draft of the manuscript.

MA Hakim Newton: Conceptualized the study, developed the methodology, and reviewed and edited the manuscript.

Jarrod Trevathan and Abdul Sattar: Supervised the research project, provided key insights, and helped with final approval.

### Competing interests

The authors declare that they have no competing interests related to the research, authorship, and/or publication of this article.

### Availability of data and materials

Our implementations are available from the following links.

- Implementation on **Wokwi** platform:  
<https://wokwi.com/projects/396638811657719809>
- Implementation on **Tinkercad** platform:  
<https://www.tinkercad.com/things/i8c4z51qvzQ-syntactic-n-semantic>

## References

Azad, T., Newton, M. H., Trevathan, J., and Sattar, A. (2023). Hierarchical decentralised edge interoperability. *IEEE Internet of Things Journal*. DOI: 10.1109/JIOT.2023.3340298.

- Bojadjevski, S., AnastasovaBojadjevski, N., Kalendar, M., and Tentov, A. (2018). Interoperability of emergency and mission critical iot data services. In *2018 26th Telecommunications Forum (TELFOR)*, pages 1–4. IEEE. DOI: 10.1109/TELFOR.2018.8611826.
- Bormann, C., Castellani, A. P., and Shelby, Z. (2012). Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67. DOI: 10.1109/MIC.2012.29.
- Bröring, A., Stasch, C., and Echterhoff, J. (2012). Ogc sensor observation service interface standard, version 2.0. Available at: <https://www.ogc.org/standards/sos/>.
- Charyyev, B., Arslan, E., and Gunes, M. H. (2020). Latency comparison of cloud datacenters and edge servers. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–6. DOI: 10.1109/GLOBECOM42002.2020.9322406.
- Cirillo, F., Solmaz, G., Berz, E. L., Bauer, M., Cheng, B., and Kovacs, E. (2019). A standard-based open source iot platform: Fiware. *IEEE Internet of Things Magazine*, 2(3):12–18. DOI: 10.1109/IOTM.0001.1800022.
- Dang, T.-B., Tran, M.-H., Le, D.-T., and Choo, H. (2017). On evaluating iotivity cloud platform. In *Computational Science and Its Applications-ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part V* 17, pages 137–147. Springer. DOI: 10.1007/978-3-319-62404-4\_10.
- del Campo, G., Saavedra, E., Piovano, L., Luque, F., and Santamaria, A. (2024). Virtual reality and internet of things based digital twin for smart city cross-domain interoperability. *Applied Sciences*, 14(7):2747. DOI: 10.3390/app14072747.
- Foundry, E. (2021). Why edgex. Available at: <https://www.edgexfoundry.org/software/platform/>.
- Gigli, L., Zyrianoff, I., Zonzini, F., Bogomolov, D., Testoni, N., Di Felice, M., De Marchi, L., Augugliaro, G., Mennuti, C., and Marzani, A. (2023). Next generation edge-cloud continuum architecture for structural health monitoring. *IEEE Transactions on Industrial Informatics*. DOI: 10.1109/tii.2023.3337391.
- Gyrard, A., Patel, P., Datta, S. K., and Ali, M. I. (2017). Semantic web meets internet of things and web of things. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 917–920. DOI: 10.1145/3041021.3051100.
- Hao, L. and Schulzrinne, H. (2021). Goldie: Harmonization and orchestration towards a global directory for iot. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE. DOI: 10.1109/info-com42981.2021.9488752.
- Jazayeri, M. A., Huang, C.-Y., and Liang, S. H. (2012). Tinysos: Design and implementation of interoperable and tiny web service for the internet of things. In *Proceedings of the First ACM SIGSPATIAL Workshop on Sensor Web Enablement*, pages 39–46. DOI: 10.1145/2451716.2451722.
- Jazayeri, M. A., Liang, S. H., and Huang, C.-Y. (2015). Implementation and evaluation of four interoperable open standards for the internet of things. *Sensors*, 15(9):24343–24373. DOI: 10.3390/s150924343.



- Kapua, E. (2018). Eclipse kapua. Accessed on June, 25. Available at: <https://www.eclipse.org/kapua>.
- Kherbache, M., Maimour, M., and Rondeau, E. (2022). Digital twin network for the iiot using eclipse ditto and hono. *IFAC-PapersOnLine*, 55(8):37–42. DOI: 10.1016/j.ifacol.2022.08.007.
- Klein, S. (2017). *IoT Solutions in Microsoft's Azure IoT Suite*. Springer. DOI: 10.1007/978-1-4842-2143-3.
- Leibovici, D. G., Santos, R., Hobona, G., Anand, S., Kamau, K., Charvat, K., Schaap, B., and Jackson, M. (2023). Geospatial standards. *The Routledge Handbook of Geospatial Technologies and Society*. DOI: 10.4324/9780367855765-7.
- Liang, S. and Khalafbeigi, T. (2019). Ogc sensorthings api part 2—tasking core, version 1.0. Available at: <https://docs.ogc.org/is/17-079r1/17-079r1.html>.
- Liang, S., Khalafbeigi, T., van Der Schaaf, H., Miles, B., Schleidt, K., Grellet, S., Beaufils, M., and Alzona, M. (2021). Ogc sensorthings api part 1: Sensing version 1.1. In *Open geospatial consortium*. DOI: 10.62973/18-088.
- Maheshwari, S., Raychaudhuri, D., Sesar, I., and Bronzino, F. (2018). Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 286–299. DOI: 10.1109/SEC.2018.00028.
- Marin, M. C., Cerutti, M., Batista, S., and Brambilla, M. (2024). A multi-protocol iot platform for enhanced interoperability and standardization in smart home. In *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE. DOI: 10.1109/CCNC51664.2024.10454663.
- Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2022). Webassembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, pages 3–8. DOI: 10.1145/3526059.3533618.
- Mofatteh, M. Y., Pirayesh, A., and Fatahi Valilai, O. (2024). A layered semantic interoperability framework for conflict resolution of semantic models in smart devices. In *Intelligent Systems Conference*, pages 425–445. Springer. DOI: 10.1007/978-3-031-66431-1\_30.
- Na, A. and Priest, M. (2007). Sensor observation service. version 1.0. Available at: <https://www.ogc.org/standards/sos/>.
- Nagasundaram, D., Manickam, S., Laghari, S. U. A., and Karuppayah, S. (2024). Proposed fog computing-enabled conceptual model for semantic interoperability in internet of things. *Bulletin of Electrical Engineering and Informatics*, 13(2):1183–1196. DOI: 10.11591/eei.v13i2.5748.
- O'Reilly, T., Toma, D., Del-Rio-Fernandez, J., and Headley, K. (2012). Ogc® puck protocol standard. *Open Geospatial Consortium (OGC), Wayland, MA, USA, OGC Candidate Encoding Standard*. Available at: <https://www.ogc.org/standards/puck/>.
- Park, H., Kim, H., Joo, H., and Song, J. (2016). Recent advancements in the internet-of-things related standards: A onem2m perspective. *Ict Express*, 2(3):126–129. DOI: 10.1016/j.ict.2016.08.009.
- Park, S. (2017). Ocf: A new open iot consortium. In *2017 31st international conference on advanced information networking and applications workshops (WAINA)*, pages 356–359. IEEE. DOI: 10.1109/WAINA.2017.86.
- Paul, A., Hazarika, B., Singh, K., Mumtaz, S., and Li, C.-P. (2024). Deep learning-based semantic interaction network: Advancing iot data modeling for interoperability. In *2024 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 135–140. IEEE. DOI: 10.1109/ICCWorkshops59551.2024.10615419.
- Pierleoni, P., Concetti, R., Belli, A., and Palma, L. (2019). Amazon, google and microsoft solutions for iot: Architectures and a performance comparison. *IEEE access*, 8:5455–5470. DOI: 10.1109/ACCESS.2019.2961511.
- Pitstick, K., Novakouski, M., Lewis, G. A., and Ozkaya, I. (2024). Defining a reference architecture for edge systems in highly-uncertain environments. *arXiv preprint arXiv:2406.08583*. DOI: 10.1109/ICSA-C63560.2024.00064.
- Pramukantoro, E. S., Bakhtiar, F. A., Aji, B., and Pratama, R. (2018). Middleware for network interoperability in iot. In *2018 5th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pages 499–502. IEEE. DOI: 10.1109/EECSI.2018.8752917.
- Rasheed, H. (2024). Consideration of cloud-web-concepts for standardization and interoperability: A comprehensive review for sustainable enterprise systems, ai, and iot integration. *Journal of Information Technology and Informatics*, 3(2). Available at: [https://www.researchgate.net/publication/382305757\\_Consideration\\_of\\_Cloud-Web-Concepts\\_for\\_Standardization\\_and\\_Interoperability\\_A\\_Comprehensive\\_Review\\_for\\_Sustainable\\_Enterprise\\_Systems\\_AI\\_and\\_IoT\\_Integration](https://www.researchgate.net/publication/382305757_Consideration_of_Cloud-Web-Concepts_for_Standardization_and_Interoperability_A_Comprehensive_Review_for_Sustainable_Enterprise_Systems_AI_and_IoT_Integration).
- Sciullo, L., Zyrianoff, I. D. R., Trotta, A., and Di Felice, M. (2021). Wot micro servient: Bringing the w3c web of things to resource constrained edge devices. In *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 161–168. IEEE. DOI: 10.1109/smartcomp52413.2021.00042.
- Shukla, S., Hassan, M. F., Tran, D. C., Akbar, R., Paputungan, I. V., and Khan, M. K. (2023). Improving latency in Internet-of-Things and cloud computing for real-time data transmission: a systematic literature review (SLR). *Cluster Computing*, pages 1–24. DOI: 10.1007/s10586-021-03279-3.
- Verdouw, C., Wolfert, S., Beers, G., Sundmaeker, H., and Chatzikostas, G. (2017). Iof2020: Fostering business and software ecosystems for large-scale uptake of iot in food and farming. In *The International Tri-Conference for Precision Agriculture in 2017*. Precision Agriculture Association New Zealand Hamilton (NZ). DOI: 10.5281/zenodo.1002903.

**Table 1.** Syntactic and Semantic edge interoperability: our implementation vs existing solutions

Interoperability Framework Implementation	Cloud Support Needed	Profile Format Language	Implementation	Supports Class-0 Devices?	Integrated Controller?	Explicit Syntactic Level?	Explicit Semantic Level?
Open Connectivity Foundation (OCF) [Park, 2017]	✓	RAML, JSON	C/C++, Java Client-Server Model	×	×	×	×
Microsoft Azure IoT [Klein, 2017]	✓	Apache Avro, JSON	.NET, Java, Python, Node.js, C/C++ Event-Driven Architecture	×	×	×	×
Amazon AWS IoT [Pierleoni et al., 2019]	✓	JSON, Apache Parquet	Python, Java, Node.js, C/C++, .NET Event-Driven Architecture	×	×	×	×
OneM2M [Park et al., 2016]	✓	XML, JSON, CBOR	Java, C/C++, Python, JavaScript Resource-Oriented Architecture	×	×	×	×
FIWARE [Cirillo et al., 2019]	✓	JSON	Java, Python, JavaScript Publish-Subscribe Pattern	×	×	×	×
SOS [Na and Priest, 2007]	✓	XML	Java, Python, .NET, JavaScript Web Service Oriented	×	×	×	×
SensorThings API [Liang and Khalafbeigi, 2019]	✓	XML	Java, Python, .NET, JavaScript RESTful Architecture	×	×	×	×
EdgeX [Foundry, 2021]	✓	XML	Go, C, Java, Python Microservices Architecture	×	×	×	×
Web of Things (WoT) [Gyrard et al., 2017]	×	JSON	JavaScript, C/C++, Java, Python Event-Driven Architecture	×	×	×	×
IoTivity [Dang et al., 2017]	×	XML	C/C++ Resource-Oriented Architecture	×	×	×	×
WebAssembly (Wasm) [Ménétrety et al., 2022]	×	JSON	Rust, C/C++, Go, AssemblyScript Component-Based Architecture	×	×	×	×
<b>This Work</b>	×	Plain TXT	C/C++ Procedural-Based Architecture and Event-Driven Architecture	✓	✓	✓	✓