WoT2Pod: An Architecture enabling an Edge-to-Cloud Continuum

Michael Freund
Fraunhofer Institute for Integrated Circuits IIS
Nürnberg, Germany
michael.freund@iis.fraunhofer.de

Justus Fries
Fraunhofer Institute for Integrated Circuits IIS
Nürnberg, Germany
justus.fries@iis.fraunhofer.de

Rene Dorsch
Fraunhofer Institute for Integraded Circuits IIS
Nürnberg, Germany
rene.dorsch@iis.fraunhofer.de

Peter Schiller
Friedrich-Alexander-Universität Erlangen-Nürnberg
Nürnberg, Germany

Andreas Harth
Fraunhofer Institute for Integrated Circuits IIS
Nürnberg, Germany

ABSTRACT

We present WoT2Pod, an architecture that provides a uniform API for data access and control of IoT devices at the edge and in the cloud, creating a seamless continuum between these environments. The uniform API is based on Resource Description Framework (RDF) graphs, Linked Data Platform (LDP) containers, and HTTP. The API is created by a middleware that uses the Web of Things to interact with devices, maps collected IoT data to RDF, provides the most recent data at the edge, and stores data for long-term storage and sharing with third parties in a Solid Pod in the cloud layer. We establish a formalism and derive equations for predicting latency, and use them to evaluate different polling and pushing-based methods for exchanging data between components. Our results underscore that interactions at the edge are consistently faster than those in the cloud, with data access 65% faster and device control 70% faster. We found that a strategy that relies entirely on pushing data provides the most optimal latency results.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; **Sensors and actuators**; • **Information systems** → **Semantic web description languages**.

KEYWORDS

Internet of Things, Web of Things, Solid Project, Architecture

ACM Reference Format:
Michael Freund, Justus Fries, Rene Dorsch, Peter Schiller, and Andreas Harth. 2023. WoT2Pod: An Architecture enabling an Edge-to-Cloud Continuum. In *The International Conference on the Internet of Things (IoT 2023), November 07–10, 2023, Nagoya, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3627050.3627063

IoT 2023, November 07–10, 2023, Nagoya, Japan

ACM ISBN 979-8-4007-0854-1/23/11.
https://doi.org/10.1145/3627050.3627063

1 INTRODUCTION

Different types of applications dealing with Internet of Things (IoT) devices and the data they generate have different requirements. Latency-sensitive applications that implement feedback loops require near real-time access to current data and the ability to quickly send control commands to selected devices. These requirements necessitate processing at the network edge due to its proximity to IoT devices [11]. In contrast, data-intensive applications that support high-level business decisions and process orchestration through long-term data analysis or machine learning tasks require access to significant amounts of historical data, typically stored in the cloud for scalability and ease of access [6].

Meeting these application requirements presents challenges.

First, the heterogeneity of IoT devices, with each device potentially using different IP-based or non-IP-based connectivity standards and communication protocols, complicates direct access to measurement data and control of the devices [19]. The Web of Things (WoT) architecture introduced by the World Wide Web Consortium (W3C) aims to alleviate this challenge by serving as a metamodel for describing existing interfaces [15] of *Things* with a *Thing Description* (TD) [16], enabling seamless communication.

Second, managing IoT data is challenging due to its multi-source nature, heterogeneity, and weak semantics [4]. Efficient management of such data often requires the use of data fusion techniques and metadata enrichment to facilitate future queries [23]. It is also necessary to ensure the long-term storage of data and to provide secure, privacy-preserving methods of data sharing [3]. The Social Linked Data (Solid) project [22], which provides a decentralized approach to managing self-describing data on the Web in so-called Pods, offers a promising solution for long-term data storage and secure data sharing.

Finally, the APIs used to access data in the cloud are often different from those used at the edge and from those used to control devices. This requires the implementation and use of different interaction algorithms in applications depending on the operation being performed.

These challenges make IoT application development complex, as applications must deal with different device protocols, process heterogeneous data with weak semantics, and adapt to different cloud and edge APIs for data access and device control. The lack of a unified approach that addresses all these challenges increases the application development effort. While initiatives such as the

Solid project and the Web of Things (WoT) provide pieces of the solution, a more comprehensive approach is essential. This is where the potential of a distributed edge middleware solution that uses the strengths of existing initiatives can become relevant.

Using the WoT abstraction model, the middleware can map various IP-based and non-IP-based communication standards and protocols to the more universal framework of IP and HTTP [8, 17]. In addition, a middleware can standardize the various data encodings used by IoT devices into a self-describing data format. This enables the creation of an API at the edge that standardizes data access and device interaction with HTTP, and simplifies data management and interpretation with a self-describing data format. In addition, the middleware can send data to Solid Pods to enable secure data sharing with third parties and long-term analytics. This allows the middleware to generate a uniform API structure both at the edge and in the cloud, creating a seamless edge-to-cloud API continuum.

Such a middleware approach shifts complexity - e.g., standardization of data encoding and harmonization of protocols - from application development to infrastructure. This shift can increase initial design complexity and extend deployment time. However, by introducing guidelines and best practices in the form of an architecture and by evaluating the potential advantages and disadvantages of different data exchange strategies, the initial deployment time can be reduced again.

In this work, we propose such an architecture, called WoT2Pod, which consists of a decentralized edge middleware that uses Semantic Web technologies to combine the interoperable device interactivity of WoT with the storage and sharing aspects of the Solid project. The middleware is driven by an algorithm that generates a unified Linked Data Platform (LDP) and HTTP-based API that provides Resource Description Framework (RDF) data both at the edge and in the cloud. The middleware consists of three components: an edge orchestrator, mediators, and intermediaries. The orchestrator not only manages WoT2Pod's setup and teardown processes, but also autonomously discovers devices based on their semantic API descriptions. In addition, the orchestrator creates a mediator and an intermediary for each device it finds. Each mediator periodically collects all available data from the corresponding IoT device, maps this data in RDF format, makes a small amount of the most recent data accessible via HTTP at the edge, and transfers the data to a Solid Pod in the cloud for long-term storage if needed. The intermediary's role is to pass control commands from the Solid Pod to the Mediator.

The key contributions of our work are:

- A middleware architecture that leverages Semantic Web technologies to connect the Web of Things and the Solid project and create a unified API for the edge and the cloud.
- An algorithm that maps device data to RDF and creates a URI structure for APIs based on relationships that reflect the device hierarchy.
- A formalism and equations for estimating the latency of different data exchange strategies and, based on this, a latency comparison of the strategies. We also demonstrate the practical applicability of our work with a deployment in an airport baggage handling system.

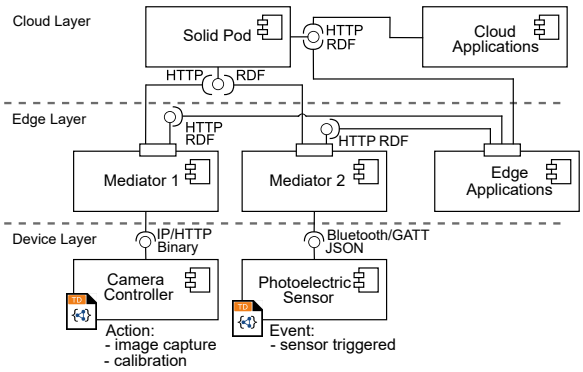

Figure 1: WoT2Pod provides a uniform HTTP and RDF based API for accessing data at the edge and in the cloud.

2 EXAMPLE SCENARIO

An airport wants to improve transparency of its baggage handling by collecting data on baggage size and color. The airport deploys a camera system consisting of RGB-D stereo vision cameras managed by a composite camera controller that provides binary data over HTTP. A photoelectric sensor, which sends data in JSON format, is connected via Bluetooth Low Energy (LE) and serves as the trigger for this IoT system. The camera controller and the photoelectric sensor are both described by a Thing Description.

The camera controller has an action for *depth sensor calibration* with no output and an action for *image capture* triggering, which provides a 3D point cloud from the merged camera images. The sensor is designed to emit *trigger* events when baggage interrupts its light beam.

The airport's IT department employs WoT2Pod to facilitate application development, as seen in Figure 1. WoT2Pod provides a uniform API across both the edge and the cloud, enabling control over Things and access to Thing data in RDF format.

Using WoT2Pod's uniform API, the airport develops an edge-based control application that creates a feedback loop. This application interfaces with the mediator of the photoelectric sensor using HTTP and RDF to determine if baggage triggers the sensor. When the sensor is activated, the application sends an RDF graph via HTTP to the camera controller's mediator, which triggers the camera controller to simultaneously capture images, enabling the application to control and orchestrate Things in near real-time.

In parallel, a separate cloud-based application processes the annotated data in the Pod, using HTTP and RDF to access and analyze the 3D point clouds to determine the size and color of the transported luggage. The results are stored back into the Pod as new RDF triples, illustrating how WoT2Pod enables intensive data processing tasks in the cloud.

Finally, an edge-based visualization application is deployed. The application uses HTTP and RDF to access the latest images from the edge, historical images from the cloud, and additional process data from the internal KG to derive performance indicators and provide image visualizations. This demonstrates WoT2Pod's ability to enable applications to access both near real-time and historical data across the edge and cloud.

WoT2Pod facilitates the development of diverse applications, enabling near real-time Thing control at the edge, supporting intensive data processing in the cloud, and providing data access across the edge and cloud. All these capabilities are delivered through a uniform API that leverages LDP container structures to expose RDF data over HTTP.

3 RELATED WORK

Data in its raw, unprocessed form often has limited utility, but the value is amplified when data is organized and structured. Wise *et al.* [24] suggest to use Linked Data to improve findability, the HTTP protocol for accessibility, ontologies and RDF for interoperability, and provenance metadata to enhance data reusability. Our approach builds on these strategies by mapping IoT data to RDF and using the Solid project, which is based on Semantic Web technologies and incorporates the LDP specification with added access control for long-term storage. Solid has proven beneficial in various contexts, including government [12] and industrial applications [13]. Additionally, RDF has been recognized as a tool for integrating diverse data sources [1] and serves as the foundational data model for Knowledge Graphs (KGs) [14]. RDF data enables RDFS/OWL based reasoning to infer implicit information [21] and also allows the interconnection of multiple RDF data sources, such as KGs, Linked Open Data, and Solid Pods using Web Linking [18], thereby combining distributed data sources.

IoT platforms such as Home Assistant[1] are widely used in home automation. Home Assistant is primarily designed to operate in local networks and allows the integration of various devices directly into its platform. This interaction is managed by special plug-ins called *integrations*. For example, the Zigbee home automation integration allows communication with Zigbee compatible devices. All data collected within Home Assistant is stored in a relational database, and both device control and data retrieval are available through an API based on JSON and HTTP. In contrast to integrations, we use the WoT TD to interact with devices. The TD is a standardized descriptor connecting devices to a broader web ecosystem and treats them like web resources. Another distinction of WoT2Pod is that the API provides self-describing RDF data, which provides a more semantically transparent perspective than traditional JSON representations. By using Solid Pods in the cloud, WoT2Pod also enables the storage of large amounts of data and the secure sharing of collected data with authenticated third parties.

Noura *et al.* presented WoTDL2API [20], a tool designed to create a RESTful API for devices using non-IP based communication, allowing easier interaction with these devices. The devices must be described using the WoTDL2API ontology. In contrast to this approach, we rely on the standardized WoT abstraction to access the devices. Furthermore, we map the collected data to RDF, which allows applications to access all data and control all devices through an API built on a single encoding (RDF) and protocol (HTTP).

Fries *et al.* [10] introduced an IoT architecture that integrates WoT with Solid Pods. It includes an edge orchestrator that discovers Things, transfers TDs, metadata, and RDF-mapped raw data into LDP containers, and handles a task queue within the Pod. The orchestrator manages properties and actions, but has no event support or data access at the edge. In contrast, we use edge mediators to map Thing data to RDF, provide near real-time data access at the edge, and support for properties, actions, and events.

[1]https://www.home-assistant.io/

4 DETAILED OVERVIEW OF WOT2POD

Our proposed WoT2Pod architecture spans three layers: the *device* layer (subsection 4.1), the *edge* layer (subsection 4.2), and the *cloud* layer (subsection 4.3). Both the edge and cloud layers provide a uniform API for controlling Things and accessing data created by a middleware at the edge. The APIs can be used by applications (subsection 4.4) deployed at the edge or in the cloud.

4.1 Device Layer

The *device* layer consists of sensors that collect data about the environment and actuators able to interact with the physical world. The interaction affordances, relationships, and static data of these embedded devices are described by a Thing Description. The TD offers an abstraction for the embedded devices, converting them into Things within the WoT context. There are two types of Things in the device layer: standalone Things and composite Things. Standalone Things are unique entities that do not encapsulate nested sub-Things. On the other hand, composite Things are formed by aggregating multiple standalone Things or composite Things. Composite Things control sub-Things, enabling them to perform more complex tasks.

4.2 Edge Layer

The *edge* layer forms the middle layer of the WoT2Pod architecture, consisting of our middleware that includes an edge orchestrator and possibly multiple mediators and intermediaries.

4.2.1 ***Edge Orchestrator.*** The edge orchestrator is hosted at the edge of the network and controls the setup and teardown process. The edge orchestrator has write access to Pods, discovers Things, and spawns mediators and intermediaries. Therefore, an edge orchestrator acts as both a client and a WoT Discoverer [5]. The discovery mechanism employed by the edge orchestrator is tailored to the specific use case and the constraints of the host machine. For instance, it may be based on direct methods such as user input, well-known URIs, DNS-based mechanisms [5], or link-following approaches [9].

4.2.2 ***Intermediary.*** Intermediaries are straightforward client software components designed to poll data from one server and push it to another. In our approach, we employ these intermediaries to either relay control commands from Pods in the cloud to mediators at the edge or to poll data from mediators and subsequently push it to Pods.

4.2.3 ***Mediator.*** Mediators are implemented as servients and use the WoT Scripting API[2] to consume TDs. By using the Scripting API, all already supported protocol bindings can be mapped to the abstract Thing interaction model[3], so that protocols like MQTT or Bluetooth LE can be mapped to HTTP.

[2]https://www.w3.org/TR/wot-scripting-api/
[3]https://www.w3.org/TR/wot-binding-templates/

Mediators perform three primary functions: collecting data from readable properties or events, relaying write-property or invoke-action operations to Things, and providing data access at the edge.

The mediator's first function is to collect, annotate, and map data to RDF, which may include provenance data, units, or other information extracted from the TD. Various methods such as templates, mapping tools, or reasoning can be used for the data mapping [2]. An example using SOSA/SSN[4] to describe observations and PROV-O[5] for provenance data is provided in Listing 1.

The second function is related to Thing control. Here, the mediator expects applications to create a new resource, containing control commands encoded as an RDF graph. This graph describes the operation, any required input, the target Thing, and the action to invoke, as illustrated in Listing 2.

In addition, the mediators provide access to a small amount of the latest annotated RDF data, structured in LDP containers, over HTTP. If the collected data requires long-term storage, the mediators can transfer it to the cloud. For ease of access, the mediators' API is described by a dynamically generated OpenAPI description.

Listing 1: Filled-in RDF graph template describing an observation and provenance data.

```
@prefix sosa: <http://www.w3.org/ns/sosa/> .
@prefix qudt: <http://qudt.org/vocab/quantity#> .
@prefix unit: <http://qudt.org/vocab/unit/> .
@prefix prov: <http://www.w3.org/ns/prov#> .
@prefix : <http://example.org#>.

:observation1 a sosa:Observation ;
  sosa:hasResult :result1 ;
  sosa:madeBySensor :sensor1 ;
  prov:wasGeneratedBy :activity1 .

:result1 a qudt:QuantityValue ;
  qudt:numericValue 25 ;
  qudt:unit unit:DEG_C .

:sensor1 a sosa:Sensor ;
  sosa:observes :temperature .

:activity1 a prov:Activity ;
  prov:used :sensor1 ;
  prov:generated :observation1 .
```

Listing 2: RDF graph to invoke action *calibrate* on sensor1.

```
@prefix rdfs:
    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix : <http://example.org#>.
@prefix ex: <http://example.ontology.org#>.

[] a ex:actionInvocation ;
  ex:forAction "calibrate" ;
  ex:forThing :sensor1 ;
  rdfs:label "Invoke calibration action" .
```

[4]https://www.w3.org/TR/vocab-ssn/
[5]https://www.w3.org/TR/prov-o/

4.3 Cloud Layer

The *cloud* layer hosts the Solid server, alongside the Pods used for long-term data storage. Given the geographic distance separating the Pods from the Things that generate the data, near real-time data access is not feasible. However, owing to the cloud's unrestricted nature, the Pods can accommodate substantial volumes of historical data and enable data sharing with other business units and external enterprises through integrated functionalities in Solid. The stored data in RDF format comprises annotated Thing data supplied by the mediators as depicted in Listing 1. The Pod provides the annotated data as RDF resources, organized in LDP containers, accessible via an HTTP API. The access control list for each created LDP container can be modified independently, enabling fine-grained access control to the data and control commands off each Thing. Control commands dispatched to the Pod are relayed to the corresponding mediator through intermediaries.

4.4 Applications

Applications control Things, retrieve, process, and visualize data. Applications may also be implemented as Solid Apps, programs equipped with WebID authentication support[6], which can pull, process, and store shared data from Solid Pods.

Depending on the application's needs, it can access different RDF data sources, including current data from edge mediators and historical data from Pods hosted in the cloud. To control Things, applications send RDF graph commands to either the associated mediator or the Pod. As a result, applications only need HTTP and RDF compliance to manage all data sources and Things. The APIs ensure consistent structure and uniform data encoding, simplifying data access and device control, and eliminating the need for applications to change internal logic for different sources. Applications can be deployed at the edge or in the cloud, enabling feedback loops, data processing, and visualization depending on the use case.

4.5 Uniform API Algorithm

When an instance of the WoT2Pod architecture is initialized, the edge orchestrator begins by attempting to discover all Things within the device layer. Following the identification, the edge orchestrator initiates the creation of a LDP container structure within a Solid Pod. The container structure mirrors the Thing hierarchy and is created by parsing static semantic data contained within discovered TDs. The semantic data could include meronym annotations or link relation types.

For instance, the URIs for the LDP container structure can be generated using Algorithm 1, which uses the SOSA/SSN properties - `sosa:hosts` and `sosa:isHostedBy` - to semantically denote meronym relationships. The algorithm takes as input a base URI string B and a map M. In M, each Thing, identified by the index i, has its unique URI uri_i as the key and the corresponding Thing Description object td_i as the value; the algorithm processes M to produce a set of generated URIs, U.

The algorithm executes three primary steps:

(1) The algorithm creates a tree structure by initiating TreeNode instances for each key-value pair in M, associating each

[6]https://solid.github.io/webid-profile/

TreeNode with the URI uri_i and its corresponding TD td_i. These TreeNodes form the backbone of our Thing hierarchy representation (lines 4 - 7).

(2) The algorithm establishes the tree's hierarchy by setting up parent-child relationships between TreeNodes, guided by the `sosa:hosts` and `sosa:isHostedBy` properties within the TDs (lines 8 - 22).

(3) Finally, the algorithm generates unique URIs for each TreeNode by traversing the tree and augmenting the URI of each parent node with the `title` property and the affordance name contained in the associated TD, mirroring the node's position within the tree's hierarchy (line 23, but function not depicted in Algorithm 1).

The output set U contains the URIs for all LDP containers and LDP resources which need to be materializes within the Pod by the edge orchestrator. To make the algorithm more tangible, we have provided an example implementation in JavaScript[7].

Once the LDP container structure is set up in the Pod through the algorithm, the edge orchestrator starts spawning intermediaries and mediators for each device. Each edge mediator also employs this algorithm to construct its local API, making the algorithm crucial in forming an edge-to-cloud data continuum with uniform APIs for data access and Thing control.

Algorithm 1 Uniform API Algorithm

```
1:  function GENERATEUNIFORMAPI(M, B)
2:      treeNodes ← {}
3:      rootNodes ← []
4:      for each uri in M do
5:          TNode ← new TreeNode(uri, M[uri])
6:          treeNodes[uri] ← TNode
7:      end for
8:      for each uri in treeNodes do
9:          TNode ← treeNodes[uri]
10:         urisOfChilds ← QUERYTD(TNode.td, 'hosts')
11:         for each uriOfChild in urisOfChilds do
12:             TNode.ADDCHILD(treeNodes[uriOfChild])
13:         end for
14:         urisOfParents ← QUERYTD(TNode.td, 'isHostedBy')
15:         if LENGTH(urisOfParents) > 0 then
16:             for each uriOfParent in urisOfParents do
17:                 TNode.SETPARENT(treeNodes[uriOfChild])
18:             end for
19:         else
20:             PUSH(rootNodes, TNode)
21:         end if
22:     end for
23:     return CREATEURIS(rootNodes, B)
24: end function
```

5 EVALUATION

To evaluate the architecture, we first formulate equations to estimate the latency of the system (subsection 5.1). We then verify these equations through a simulation based on virtual Ethernet connections, before proceeding to evaluate different data exchange strategies using the equations (subsection 5.2).

We consider only HTTP 1.1 communication over wired connections, excluding other IoT communication standards like MQTT or Bluetooth LE, and assume non-persistent HTTP connections, requiring a new TCP connection for each data exchange, to estimate performance under worst-case conditions.

Table 1: Symbols denoting end-to-end latency based on component interaction and communication approach.

Symbol	Meaning
$L_{X \rightleftharpoons Y}$	Latency when X polls Y
$L_{X \leftrightharpoons Y}$	Latency when Y polls X
$L_{X \rightharpoonup Y}$	Latency when X pushes to Y
$L_{X \leftharpoonup Y}$	Latency when Y pushes to X

5.1 Describing Latency

We introduce a formalism shown in Table 1 to describe the end-to-end latency of the architecture in various scenarios. For example, the latency when component X polls Y is denoted as $L_{X \rightleftharpoons Y}$. More complex scenarios, such as the latency between X and Z, with Y polling X and pushing to Z, can be expressed as $L_{X \leftrightharpoons Y \rightharpoonup Z}$.

We now formulate equations to estimate the latency between two components, X and Y, depending on whether they use polling or pushing-based communication.

As the communication between X and Y is HTTP-based, a TCP connection must be established before data exchange — regardless of whether the communication approach is based on polling or pushing. The TCP connection is set up using the Three-Way handshake [7], as illustrated in Figure 2a.

The handshake begins when X sends a `SYN` packet to Y. In response, Y sends back a `SYN-ACK` packet. X then acknowledges with an `ACK` packet, thereby establishing the TCP connection. Note that X does not have to wait for the `ACK` packet to arrive at Y to establish the TCP connection. The TCP teardown, which has no impact on latency, is excluded from our consideration.

TCP packet processing times at both X and Y are below 0.2 ms in our measurements. Since our focus is on estimating latency, we consider these durations to be negligible. Thus, we approximate the duration of a TCP handshake by a single round-trip time, represented as RTT_{XY}.

Once the TCP connection is established, X and Y can communicate using, for example, a polling-based HTTP interaction, i.e., $L_{X \rightleftharpoons Y}$. The exchanged packets and required processing times in a polling-based scenario are depicted in Figure 2b. X sends an HTTP GET request to Y. Y processes the received request, needing time $t_{req(Y)}$ for the processing operation, and then responds with the requested data. Upon receiving the data, X must processes it, an operation that takes time $t_{data(X)}$ to finish. The times $t_{req(Y)}$ and $t_{data(X)}$ are typically larger than 1 ms. Since latency is typically measured in milliseconds, durations greater than 1 ms are significant and cannot be considered negligible. Consequently, the total time until data is available at X when using HTTP consits of RTT_{XY},

[7] https://github.com/FreuMi/uriGeneration

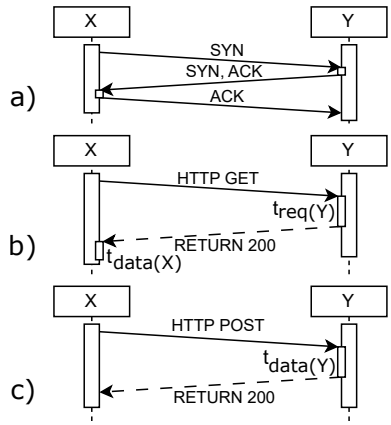

Figure 2: UML component diagram illustrating a) TCP connection establishment, b) HTTP polling-based, and c) push-based data exchange.

$t_{req(Y)}$, and $t_{data(X)}$. The polling interval of X, represented as PI_X, also contributes to $L_{X \rightleftharpoons Y}$. In a worst-case scenario, if the data is not available during the first poll, X has to wait for a complete polling interval before sending the next request. Consequently, $L_{X \rightleftharpoons Y}$ can be represented by equation 1, which takes into account both the TCP handshake and the HTTP polling-based data exchange. Equation 1 is also applicable in the case of $L_{X \leftrightarrows Y}$, since it only requires the exchange of X and Y positions and the inversion of the arrow symbol, yielding $L_{Y \rightleftharpoons X}$. All symbols used are defined in Table 2.

$$L_{X \rightleftharpoons Y} = RTT_{XY} + RTT_{XY} + t_{req(Y)} + t_{data(X)} + PI_X \quad (1)$$

$$= 2 \cdot RTT_{XY} + t_{req(Y)} + t_{data(X)} + PI_X \quad (2)$$

When X and Y exchange data using a push-based interaction, i.e. $L_{X \rightharpoonup Y}$, instead of a polling-based interaction, the HTTP packets exchanged are slightly different, as shown in Figure 2c.

X initiates the data exchange by sending an HTTP POST request to Y. Upon receipt, Y processes the incoming data, an operation that takes $t_{data(Y)}$ to complete. Then Y sends a response back to X. Since the data is already at Y and there is no need to wait for the response to reach X, the response does not contribute to latency. Therefore, in a scenario with symmetric latency between X and Y, only half of RTT_{XY} needs to be considered. Consequently, $L_{X \rightharpoonup Y}$ is described by equation 3, taking into account both the TCP handshake and the HTTP request. Equation 3 can also be applied in the case of $L_{X \leftharpoonup Y}$, as it only requires the swapping of X and Y and the inversion of the arrow, resulting in $L_{Y \rightharpoonup X}$.

$$L_{X \rightharpoonup Y} = RTT_{XY} + 0.5 \cdot RTT_{XY} + t_{data(Y)} \quad (3)$$

$$= 1.5 \cdot RTT_{XY} + t_{data(Y)} \quad (4)$$

We can use the derived equations 2 and 4 to calculate the latency of a combination of interactions, for instance $L_{X \leftrightarrows Y \rightharpoonup Z}$. This can be done by summing $L_{X \leftrightarrows Y}$ and $L_{Y \rightharpoonup Z}$. Once values are inserted into equation 6, we can estimate the latency for this scenario.

$$L_{X \leftrightarrows Y \rightharpoonup Z} = L_{X \leftrightarrows Y} + L_{Y \rightharpoonup Z} \quad (5)$$

$$= 2 \cdot RTT_{XY} + 0.5 \cdot t_{req(X)} + t_{data(Y)} + PI_Y + 1.5 \cdot RTT_{YZ} + t_{data(Z)} \quad (6)$$

We validate the derived equations for polling, pushing, and a combination of multiple communication approaches using a simulation based on virtual Ethernet connections and JavaScript code[8]. All RTTs are set to 100 ms, while processing times are determined by the runtime of the software used.

As shown in Figure 3, the deviation between the total measured runtime and the value calculated by the equation is approximately 7 ms for $X \rightleftharpoons Y$ and $X \rightharpoonup Y$, and about 14 ms for $X \leftrightarrows Y \rightharpoonup Z$. The latter is expected, given that it combines the previous two scenarios. Notably, these offsets remain constant even as latency increases.

The offset of 7 ms is due to the inaccuracy in measuring t_{data} and t_{req}. This is because the built-in HTTP module is compiled into the node.js binary, which prevents timing measurements at the lowest level. The total relative error between the equation and the simulation is about 3%. Despite this small discrepancy, the simulation confirms the correctness of our derived equations and the accuracy of our models.

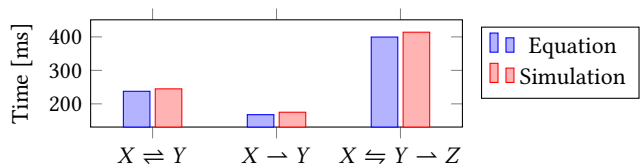

Figure 3: Comparison between theoretical equations (blue) and empirical simulation results (red) for a RTT of 100ms.

Table 2: Symbols denoting processing times, round trip times, and polling intervals.

Symbol	Meaning
RTT_{XY}	Round Trip Time between X and Y
$t_{req(X)}$	Time to process request at X
$t_{data(X)}$	Time to process received data at X
PI_X	Polling interval of X

Table 3: Symbols representing various components and their implementation.

Symbol	Meaning	Implementation
T	Thing	Servient
M	Mediator	Servient
P	Solid Pod	Server
I	Intermediary	Client
A	Application	Client or Servient

5.2 Theoretical Latency Analysis of WoT2Pod

In this subsection we compare the latency in two scenarios using the equations derived in the previous subsection. For the equations

[8] https://github.com/FreuMi/WoT2PodSim

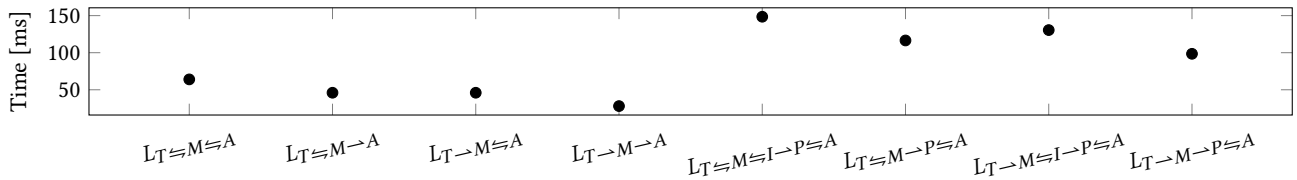

Figure 4: Data transfer time from Thing to application, depending on the interaction approach and whether the data is accessed at the edge or in the cloud.

we use the formalism introduced in Table 1 in combination with the symbols in Table 2 and Table 3. We consider the two following scenarios: **S_1)** Thing data to application, and **S_2)** control commands from application to Thing. Hereafter, we refer to the components at the edge as mediators and the component in the cloud as Pod. However, the latency equations generally apply to all components at the edge or in the cloud and not only to the ones used in WoT2Pod.

5.2.1 ***Scenario S_1: Thing Data to Application.*** Scenario S_1 includes several variations based on the communication approaches used. Things can provide data to mediators in two ways: as a read property via polling, or as an event via pushing. Both approaches can also be used to transfer data between the mediator and the Pod. Because the Pod is implemented as a server, it lacks the ability to poll on its own. Therefore, the Pod requires another intermediary agent that can poll the mediator and push data to the Pod.

Applications can access data from either a mediator or a Pod. If the application is implemented as a client, it can use polling to retrieve data. But, if the application is implemented as a servient, it has the ability to receive data that is pushed. Notably, because the Pod is implemented as a server, direct data pushing to the application is not possible. Instead, an additional intermediary agent would be required to facilitate data pushing from the Pod to the application. We did not consider this approach because the additional intermediary would only add complexity without improving performance or allow the separation of client and server implementations.

The application interacting with the mediator has four possible variants $L_{T\leftrightarrows M\leftrightarrows A}$, $L_{T\leftrightarrows M\rightharpoonup A}$, $L_{T\rightharpoonup M\leftrightarrows A}$, and $L_{T\rightharpoonup M\rightharpoonup A}$.

The application interacting with the Pod has also four variants $L_{T\leftrightarrows M\leftrightarrows I\rightharpoonup P\leftrightarrows A}$, $L_{T\leftrightarrows M\rightharpoonup P\leftrightarrows A}$, $L_{T\rightharpoonup M\leftrightarrows I\rightharpoonup P\leftrightarrows A}$, and $L_{T\rightharpoonup M\rightharpoonup P\leftrightarrows A}$. As mentioned above, the additional four variants in which the Pod pushes data to the application are not included.

Figure 4 presents the latency of each variant, calculated using hypothetical values. We have assumed that all processing times (t_{data} and t_{req}) equal 5 ms, with polling intervals set at 10 ms. In terms of latency, we assume that device-to-edge and edge-to-edge latencies are 6 ms. For edge-to-cloud or cloud-to-cloud latencies, we reference the average provided by the FCC's Measuring Broadband America[9] for cable connections, which stands at 17 ms. In our setup, applications, intermediaries, and mediators are deployed at the edge, while the Pod is hosted within the cloud.

Figure 4 demonstrates the consistent speed advantage of data access at the edge, with the worst-case scenario still being 65% faster than the best case in the cloud. Furthermore, the figure emphasizes the general superiority of push-based interaction over polling-based interaction in terms of latency. This result was expected based on the equations describing the interactions.

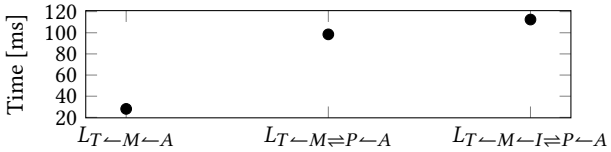

Figure 5: Control command transfer time from application to Thing, depending on the interaction approach and whether the command is sent to the edge or to the cloud.

5.2.2 ***Scenario S_2: Control Commands from Application to Thing.*** Scenario S_2 outlines the time to relay control commands from applications to Things. Applications can either send commands directly to edge mediators or to the the Pod in the cloud. Communication is always push-based as both Thing and Pod are implemented as servers.

Therefore, two combinations are possible if the application sends control commands to the Pod: $L_{T\leftarrow M\rightleftharpoons P\leftarrow A}$ and $L_{T\leftarrow M\leftarrow I\rightleftharpoons P\leftarrow A}$. There is also one other possible combination if the application sends control commands directly to the mediator: $L_{T\leftarrow M\leftarrow A}$.

Figure 5 presents the latency for each of the three variants, calculated using the values provided in 5.2.1. As depicted, sending control commands directly to the edge is approximately 70% faster than employing the quickest cloud variant. This outcome aligns with our expectations, given that the RTT at the edge is significantly shorter than the RTT to the cloud.

6 DEPLOYMENT

A real-world implementation was conducted at Munich Airport, one of the largest airports in Europe. The airport already uses a KG for data management and wants to gain more insight into its baggage handling process using IoT devices, similar to the example scenario described in this paper.

Our prototype is installed at a check-in counter and uses two industrial PCs: one hosting an edge orchestrator and intermediaries, and the other hosting mediators. The setup also includes a Pod hosted on a Community Solid Server[10] in the cloud. The prototype features three Intel Realsense D435i stereo vision cameras connected to a camera controller, along with a photoelectric sensor managed by an Arduino. Both the Arduino and the camera controller are described by TDs. The Things have associated mediators

[9] https://www.fcc.gov/general/measuring-broadband-america

[10] https://github.com/CommunitySolidServer/CommunitySolidServer

that not only allow control of the Things using actions or write properties, but also receive events, and poll read properties. The mediators are configured to retain the last ten sensor data points and images, as well as to push data for long-term storage to a Pod.

A control application operating at the edge implements a feedback loop, checks if luggage has triggered the sensor, and sends an RDF graph describing the `captureImages` action to the camera controller's mediator to invoke the action to capture images. Additionally, we deployed two cloud-operating applications used for processing of data in the Pod. One application estimates luggage dimensions using computer vision algorithms and depth images, while the other application uses K-Means clustering to determine the most predominant luggage colors from the RGB images.

We also deployed a web-based application that enables a process expert to visualize historical image data stored in the Pod and the near-real time images on mediators, along with existing process data from the internal KG. The airport's existing use of KGs facilitated the integration of the newly gathered process data stored in the Pod, as all data is in RDF format and can be linked seamlessly.

The check-in counter processes about 2,700 pieces of luggage each day, and for every piece of luggage, the camera system produces six images: three color images and three depth images. Each image is approximately 600 kilobytes in size, resulting in a total daily data generation of around 9.7 gigabytes. The prototype system has been in operation for a little over two months and has accumulated a total of about 600 gigabytes.

7 CONCLUSION AND FUTURE WORK

This paper began by identifying three challenges that complicate the development of IoT applications: the diverse nature of IoT devices, the task of managing multi-source and semantically weak data, and the disparate APIs used at different device, edge, and cloud layers. In response, we proposed WoT2Pod, an architecture consisting of a decentralized middleware that integrates the Web of Things and the Solid project using Semantic Web technologies. Wot2Pod enables the creation of a uniform API based on HTTP, LDP and RDF at the edge and the cloud. The middleware uses an algorithm to create this uniform API and was deployed at an airport where we developed three types of applications. The evaluation showed that latency can vary depending on the combination of data exchange strategies used (push or polling) and whether the data is accessed at the edge or in the cloud. We found that access at the edge is faster than access in the cloud in all cases, and that the push-only approach is the fastest for getting data from things to applications and vice versa. However, more research is needed to investigate how the proposed architecture scales in larger installations.

Future work will focus on analyzing the scalability of the architecture, evaluating latency when using wireless communications such as Bluetooth LE and other protocols, and improving application composition. We are also looking to expand our real-world deployments.

ACKNOWLEDGMENTS

This work was funded by the Bayerisches Verbundforschungsprogramm (BayVFP) des Freistaates Bayern through the KIWI project (grant no. DIK0318/03).

REFERENCES

[1] Jakob Beetz and André Borrmann. 2018. Benefits and limitations of linked data approaches for road modeling and data exchange. In *Advanced Computing Strategies for Engineering: 25th EG-ICE International Workshop 2018, Lausanne, Switzerland, June 10-13, 2018, Proceedings, Part II 25*. Springer, 245–261.

[2] Pieter Bonte and Femke Ongenae. 2023. Towards Cascading Reasoning for Generic Edge Processing. (2023).

[3] John Byabazaire, Gregory O'Hare, and Declan Delaney. 2020. Using trust as a measure to derive data quality in data shared IoT deployments. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 1–9.

[4] Hongming Cai, Boyi Xu, Lihong Jiang, and Athanasios V Vasilakos. 2016. IoT-based big data storage systems in cloud computing: perspectives and challenges. *IEEE Internet of Things Journal* 4, 1 (2016), 75–87.

[5] Andrea Cimmino, Michael McCool, Farshid Tavakolizadeh, and Kunihiko Toumura. 2023. Web of Things (WoT) Discovery. https://www.w3.org/TR/wot-discovery/.

[6] Michele De Donno, Koen Tange, and Nicola Dragoni. 2019. Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog. *Ieee Access* 7 (2019), 150936–150948.

[7] Wesley Eddy. 2022. Transmission Control Protocol (TCP). RFC 9293. https://doi.org/10.17487/RFC9293

[8] Michael Freund, Rene Dorsch, and Andreas Harth. 2022. Applying the Web of Things Abstraction to Bluetooth Low Energy Communication. *arXiv preprint arXiv:2211.12934* (2022).

[9] Michael Freund, Justus Fries, Thomas Wehr, and Andreas Harth. 2023. Generating Visual Programming Blocks based on Semantics in W3C Thing Descriptions. (2023).

[10] Justus Fries, Michael Freund, and Andreas Harth. 2023. A Solid Architecture for Machine Data Exchange with Access Control. (2023).

[11] Mian Guo, Lei Li, and Quansheng Guan. 2019. Energy-efficient and delay-guaranteed workload allocation in IoT-edge-cloud computing systems. *IEEE Access* 7 (2019), 78685–78697.

[12] Jonni Hanski, Pieter Heyvaert, Ben De Meester, Ruben Taelman, and Ruben Verborgh. 2023. Distributed Social Benefit Allocation using Reasoning over Personal Data in Solid. (2023).

[13] Daniel Henselmann, Karina Kolinsky, Sebastian Schmid, Daniel Schraudner, Andreas Both, and Andreas Harth. 2022. Solid Proof of Concept in an Enterprise Loan Request Use Case. (2022).

[14] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. 2021. Knowledge graphs. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–37.

[15] Michael Jacoby and Thomas Usländer. 2020. Digital twin and internet of things—Current standards landscape. *Applied Sciences* 10, 18 (2020), 6519.

[16] Sebastian Kaebisch, Michael McCool, Ege Korkan, Takuki Kamiya, Victor Charpenay, and Matthias Kovatsch. 2023. Web of Things (WoT) Thing Description 1.1. https://www.w3.org/TR/wot-thing-description/.

[17] Michael Lagally, Ryuichi Matsukura, Michael McCool, Kunihiko Toumura, Kazuo Kajimoto, Toru Kawaguchi, and Matthias Kovatsch. 2023. Web of Things (WoT) Architecture 1.1. https://www.w3.org/TR/wot-architecture/.

[18] Mark Nottingham. 2017. Web Linking. RFC 8288. https://doi.org/10.17487/RFC8288

[19] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. 2019. Interoperability in internet of things: Taxonomies and open challenges. *Mobile networks and applications* 24 (2019), 796–809.

[20] Mahda Noura, Sebastian Heil, and Martin Gaedke. 2019. Webifying heterogenous internet of things devices. In *Web Engineering: 19th International Conference, ICWE 2019, Daejeon, South Korea, June 11–14, 2019, Proceedings 19*. Springer, 509–513.

[21] Axel Polleres, Aidan Hogan, Renaud Delbru, and Jürgen Umbrich. 2013. RDFS and OWL reasoning for linked data. In *Reasoning Web International Summer School*. Springer, 91–149.

[22] Andrei Vlad Sambra, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitri Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee. 2016. Solid: a platform for decentralized social applications based on linked data. *MIT CSAIL & Qatar Computing Research Institute, Tech. Rep.* (2016).

[23] Lidong Wang. 2017. Heterogeneous data and big data analytics. *Automatic Control and Information Sciences* 3, 1 (2017), 8–15.

[24] John Wise, Alexandra Grebe de Barron, Andrea Splendiani, Beeta Balali-Mood, Drashtti Vasant, Eric Little, Gaspare Mellino, Ian Harrow, Ian Smith, Jan Taubert, et al. 2019. Implementation and relevance of FAIR data principles in biopharmaceutical R&D. *Drug discovery today* 24, 4 (2019), 933–938.