SemWoT: Integrating W3C Web of Things Devices into the Semantic Web

No Author Given

No Institute Given

Abstract. We introduce SemWoT, an approach to integrate Web of Things (WoT) devices into the Semantic Web. SemWoT defines two mappings: a mapping between different IoT protocols using the WoT abstraction as an intermediate representation, and a mapping to generate RDF mapping language (RML) templates based on semantic information to map non-RDF data to RDF in a transparent way. Additionally, we implement the SemWoT approach in a mediator, define an ontology called `aio` to describe interactions with the mediator, and specify interaction sequences to offer a RESTful Read-Write Linked Data interface. Our evaluation of the SemWoT mediator shows that it introduces a consistent overhead compared to direct access but offers the benefits of easy HTTP-based data access and provides semantically annotated data.

Keywords: Web of Things · Semantic Web · Read-Write Linked Data.

1 Introduction

Internet of Things (IoT) devices communicate with other devices, gateways or the cloud using various communication protocols depending on the application area, such as Bluetooth Low Energy (LE) in combination with a Generic Attribute Profile (GATT) or the Internet Protocol (IP) in combination with the Hypertext Transfer Protocol (HTTP) [1]. Connected devices also exchange data in different data formats, including JSON, plain text encoded in Unicode, and binary data. Because of the different communication protocols and data formats used, the IoT can be considered a heterogeneous field of connected devices.

One approach to overcoming the challenges of heterogeneity in the IoT domain is to create semantic interoperability, for instance, through the Web of Things (WoT) Architecture [12] developed by the World Wide Web Consortium (W3C). The WoT architecture achieves semantic interoperability by providing multiple building blocks. One building block is the semantic interface description in the form of RDF graphs [8]. The RDF graph describes available interaction affordances, grouped into `properties` that expose the internal state of devices, `actions` that handle long-running functions, and `events` that model asynchronous data pushes from devices. Additionally, the RDF graphs include semantic metadata annotations, such as information about units of measurement, and natural language annotations through comments or labels. A second building block of the WoT Architecture are the protocol binding templates [11] available for multiple IoT protocols, such as HTTP, Modbus, or Bluetooth LE [5]. The bindings define a mapping of concrete protocol methods, such as `HTTP GET` or `HTTP POST`, to abstract WoT methods, such as `readProperty` or `invokeAction`. A third building block is a software implementation called the WoT Scripting API [10]. The WoT Scripting API can parse semantic interface description graphs and use protocol bindings to allow users to implement applications independent of the protocol, using only the abstract WoT methods. In addition, the WoT Scripting API provides data format decoders to convert read IoT data into in-memory data structures of the WoT Scripting API implementation language.

Within a WoT context, IoT devices are in general called *Things* and semantic interface description RDF graphs are called *Thing Descriptions* (TDs).

1.1 Problem Statement

Despite the semantic interoperability provided by the WoT architecture, developing applications that use data generated by Things face two major challenges.

1. A user's end device running the application must support all desired IoT communication protocols, possibly through hardware adapters for protocols such as Bluetooth LE or ZigBee. In addition, the end device must be in close proximity to the Things, as some IoT protocols have limited range [17].
2. The WoT Scripting API only provides data format transformations for a subset of terms defined by JSON schema[1], which means that the Thing measurements do not provide any additional semantic or natural language annotations that may partially be present in the TD, such as units, provenance data, or comments, making processing of the IoT data more difficult.

To encourage the adoption of the Web of Things and the development of WoT applications, there is a need for an accessible, language-independent, and user-friendly interface that accepts and provides semantically annotated Thing data, effectively integrating Things into the Semantic Web.

1.2 Approach and Contributions

Therefore, we propose to use a mediator that provides a RESTful Read-Write Linked Data (RWLD) interface [2], allowing users to access and engage with the interaction affordances of a Thing, such as reading and writing properties, invoking actions, and subscribing to events. The mediator acts as an intermediary between different communication protocols, mapping them to a RESTful HTTP interface, while using Semantic Web technologies such as RDF and ontologies to transform diverse data formats into FAIR RDF data [18] using the RDF Mapping Language (RML) [4]. FAIR data means that the data is findable, accessible, interoperable, and reusable. Adherence to the FAIR data principles

[1] `https://www.w3.org/TR/wot-thing-description11/#sec-data-schema-vocabulary-definition`

is important and has been shown to facilitate the development of data analytics and artificial intelligence applications in both scientific research [16] and industry [6]. In general, the proposed mediator approach separates the logic of a user application that deals with information processing and control of Things from the need to support different communication protocols and codecs for parsing, encoding, and decoding different data formats. As a result, application development is simplified by using a common, language-independent interface that reuses established Semantic Web technologies.

The key contributions of our work are as follows:

- The introduction and formalization of mappings between different IoT protocols, as well as between semantics in TDs and RML mapping rules.
- The introduction of an ontology to interact with the mediators RWLD interface and the definition of interaction sequences for the mediator.
- The prototypical implementation and empirical evaluation of the performance overhead introduced by the mediator.

2 Running Example

To better illustrate the contributions of this work, we use an IoT sensor as a running example throughout the paper. The interactions offered by the IoT device and additional metadata are semantically described using a TD, making it a *Thing* in the WoT context. The TD graph in turtle serialization describing the API of the Thing is illustrated in Listing 1.1. The Thing is able to provide temperature measurements via the property affordance `temp` (line 16) and offers a user to activate or deactivate the precision mode using the action affordance `precisionMode` (line 27). The temperature measurement data is encoded in binary format (line 19), and the communication is based on Bluetooth LE in combination with the Generic Attribute Profile (GATT). GATT uses a server-client communication model and defines a way to structure data using services that contain characteristics that can be read using the `read` method, written using either `write without (w/o) response` or `write with (w/) response`, or subscribed to using the `notify` method. In this example, the IoT device acts as a GATT server (line 13), while the data consumer acts as a GATT client.

In the following sections, we demonstrate how to integrate the sensor Thing into the Semantic Web. Specifically, we show how to map the Thing's communication protocol, Bluetooth LE, to the Web's communication protocol, HTTP, and how to convert the binary data exchanged by the Thing, along with relevant metadata, into RDF, the data format for knowledge representation of the Semantic Web.

3 Related Work

The challenge of integrating IoT devices into the Web or Semantic Web has been explored in previous studies. This section reviews key contributions and highlights how our approach builds on and differs from existing methodologies.

Listing 1.1. Thing description of a sensor, introduced in the running example, shown in Turtle serialization. The API is based on an existing Xiaomi Flower Care sensor.

```
@prefix td: <https://www.w3.org/2019/wot/td#> .
@prefix jschema: <https://www.w3.org/2019/wot/json-schema#> .
@prefix hctl: <https://www.w3.org/2019/wot/hypermedia#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix qudt: <https://qudt.org/2.1/schema/qudt#> .
@prefix sbo: <https://freumi.inrupt.net/SimpleBluetoothOntology.ttl#> .
@prefix bdo: <https://freumi.inrupt.net/BinaryDataOntology.ttl#> .
@prefix sosa: <http://www.w3.org/ns/sosa/> .

[] a td:Thing ;
  td:title "Flower"@en ;
  td:description "A Xiaomi Flower Care Sensor."@en ;
  sbo:hasGATTRole sbo:Server ;
  td:hasPropertyAffordance [
    a jschema:NumberSchema ;
    td:name "temp" ;
    td:description "In degrees Celsius" ;
    sosa:hasFeatureOfInterest <http://ex.com/Room404> ;
    bdo:pattern "{temp}0000000023c00fb349b" ;
    bdo:variables [
        bdo:bytelength 2 ;
        bdo:scale 0.1 ;
    ] ;
    td:hasForm [ hctl:hasTarget "gatt://5C-85-7E-B0-25-EB/f9b3/805f9"^^xsd:anyURI ]
  ] ;
  td:hasActionAffordance [
    td:name "precisionMode" ;
    td:description "Enable precision mode."@en ;
    td:hasInputSchema [
        a jschema:IntegerSchema ;
        bdo:bytelength 1 ;
    ] ;
    td:hasForm [ hctl:hasTarget "gatt://5C-85-7E-B0-25-EB/f9b3/805f9"^^xsd:anyURI ]
  ] .
```

Käfer et al. [9] present two methods for connecting sensors and actuators to the Web. The first method involves a direct connection between devices and Web resources, using a REST server to provide resource information. The second method uses an intermediary with a sensor/actuator adapter and a client connector that represents the state of connected devices. Both approaches use web technologies and linked data to integrate IoT devices into applications.

In contrast, our approach not only integrates IoT devices into the Semantic Web using the standardized WoT framework, but also uses the information contained in the TDs to create a RESTful API with read-write linked data. This creates an easy-to-use and machine-readable interface that can be applied to different devices, improving interoperability and usability.

Noura et al. [13] present the WoTDL2API approach, which uses model-driven engineering and OpenAPI to automatically generate RESTful APIs for IoT device control and access. The WoTDL2API method achieves device and platform interoperability by transforming IoT devices into WoT devices by describing the interface and mapping the protocols to HTTP. The generated RESTful APIs are defined in custom ontologies, allowing seamless integration into the WoT without prior API standardization, thus supporting diverse devices with custom APIs.

Our approach goes beyond protocol mapping to include mapping various data formats to RDF. The semantic enrichment of raw data allows information to be directly processed and interpreted by both humans and machines. By incorporating RDF, we achieve a higher level of semantic interoperability, making data more accessible and usable across platforms and applications.

Building on the previous work, our method aims to provide a more comprehensive solution for IoT interoperability by not only mapping protocols to HTTP for easier data access, but also mapping the different data formats to RDF for easier data processing.

4 Mapping between Protocols using the Web of Things Abstraction

As a first step, we show how methods from an arbitrary input protocol can be translated to methods of an arbitrary output protocol via the WoT interaction methods using WoT protocol bindings. We formalize the mapping approach using set theory to ensure precision and clarity. Additionally, we provide for each definition concrete steps applied to the running example to illustrate the mapping of the input protocol HTTP to the output protocol Bluetooth LE, representing the interaction from the Web with the Bluetooth LE device.

For the mapping, we introduce three distinct sets I_P, W, and O_P:

Definition 1 (Input Protocol Methods). *Let I_P be the set of all interaction methods of the input protocol, where each element represents a different interaction method.*

Example 1. In the running example, we consider the HTTP interaction methods as input, as defined by the HTTP protocol bindings[2]. The set of input methods is given by $I_P = \{\texttt{GET}, \texttt{PUT}, \texttt{POST}, \texttt{GET+LP}\}$ where `LP` stands for long polling.

Definition 2 (WoT Interaction Methods). *Let W be the set of all WoT interaction methods*[3]*, where each element represents a different WoT interaction method.*

Example 2. As a next step, we define the WoT interaction methods as an intermediate representation to map from HTTP to Bluetooth LE. To demonstrate the approach, we focus on the four basic WoT interaction methods, i.e., $W = \{\texttt{readProperty}, \texttt{writeProperty}, \texttt{invokeAction}, \texttt{subscribeEvent}\}$.

Definition 3 (Output Protocol Methods). *Let O_P be the set of all interaction methods of the output protocol, where each element represents a different interaction method.*

Example 3. In the running example, our output set consists of the corresponding Bluetooth LE interaction methods [5], i.e., $O_P = \{\texttt{read}, \texttt{write w/o response}, \texttt{write w/ response}, \texttt{notify}\}$.

Given the sets I_P, W, and O_P, the correspondence to and from the WoT interaction methods is given by the functions f_1 and f_2, defined as follows:

Definition 4 (Input to WoT). *Let $f_1 : I_P \to M$ be the mapping function from the input protocol methods to the WoT interaction methods.*

Example 4. In the running example, the function f_1 describes the mapping from HTTP methods to WoT methods, therefore f_1 is defined as the set of ordered pairs $f_1 = \{(\texttt{GET}, \texttt{readProperty}), (\texttt{PUT}, \texttt{writeProperty}), (\texttt{POST}, \texttt{invokeAction}), (\texttt{GET+LP}, \texttt{subscribeEvent})\}$.

Definition 5 (WoT to Output). *Let $f_2 : M \to O_P$ be the mapping function from the WoT interaction methods to the output protocol methods.*

Example 5. In the running example, the function f_2 describes the mapping from WoT methods to Bluetooth LE methods [5], the function f_2 is therefore defined as the set of ordered pairs $f_2 = \{(\texttt{readProperty}, \texttt{read}),$
$(\texttt{writeProperty}, \texttt{write w/o response}), (\texttt{invokeAction}, \texttt{write w/ response}),$
$(\texttt{subscribeEvent}, \texttt{notify})\}$.

The mapping from the set of input protocol methods I_P to the set of output protocol methods O_P is given by the composite mapping function $h = f_2 \circ f_1$ defined as follows:

[2] https://w3c.github.io/wot-binding-templates/bindings/protocols/http/#http-default-vocabulary-terms

[3] https://www.w3.org/TR/wot-thing-description11/#table-well-known-operation-types

Definition 6 (Composite Mapping Function). *Let h be the mapping function from I_P to O_P, given as $h(i) = f_2(f_1(i)) \ \forall \ i \in I_P$, with f_1 and f_2 as defined in Definitions 4 and 5.*

Example 6. In the running example, the mapping is performed form the input protocol HTTP to the output protocol Bluetooth LE, which means, that the function h is given by following statements:

$$\begin{aligned} h(\texttt{GET}) &= f_2(f_1(\texttt{GET})) = f_2(\texttt{readProperty}) = \texttt{read} \\ h(\texttt{PUT}) &= f_2(f_1(\texttt{PUT})) = f_2(\texttt{writeProperty}) = \texttt{write w/o response} \\ h(\texttt{POST}) &= f_2(f_1(\texttt{POST})) = f_2(\texttt{invokeAction}) = \texttt{write w/ response} \\ h(\texttt{GET+LP}) &= f_2(f_1(\texttt{GET+LP})) = f_2(\texttt{subscribeEvent}) = \texttt{notify} \end{aligned}$$

The statements demonstrate that the composite mapping function h produces the correct mapping between the methods of the protocols HTTP and Bluetooth LE, consisting of the tuples:
$h = \{(\texttt{GET}, \texttt{read}), (\texttt{PUT}, \texttt{write w/o response}), (\texttt{POST}, \texttt{write w/ response}), (\texttt{GET+LP}, \texttt{notify})\}$.

The mapping is lossless and reversible as long as there exists a clear one-to-one correspondence between each protocol method and the respective abstract WoT interaction method.

5 Generating RML mapping rules using semantic information from WoT Thing Descriptions

After introducing an approach to map different protocols using the WoT methods as an intermediate representation in the previous section, we now present how to map the relevant information contained in WoT TDs to an RML mapping rule. The mapping rules allow various non-RDF data formats to be converted to RDF by specifying how the resulting RDF data should be structured and which ontologies should be used. By generating these RML mapping rules, we can transform data into RDF in a transparent and deterministic way.

First, we define a TD graph, which describes the API of an associated IoT device using property affordances, action affordances, and event affordances.

Definition 7 (Thing Description Graph). *Let P, A, and E be the sets of all property, action, and event affordances of a device, respectively. A Thing Description Graph, G_{TD}, is the union of these sets of interaction affordances: $G_{TD} = P \cup A \cup E$.*

Example 7. In the running example, we have one property affordance called `temp`, and one action affordance called `precisionMode`. Therefore the set P is given by $P = \{\texttt{temp}\}$ and the set A is given by $A = \{\texttt{precisionMode}\}$ while the set E is the empty set. Therefore, G_{TD} is given by $G_{TD} = P \cup A \cup E = \{\texttt{temp}\} \cup \{\texttt{precisionMode}\} \cup \emptyset = \{\texttt{temp}, \texttt{precisionMode}\}$.

Next, we define an interaction affordance I in the set of all interaction affordances, i.e. the Thing Description Graph G_{TD}. An interaction affordance I contains information about the datatype of the exchanged data and the set of additional semantic annotations.

Definition 8 (Interaction Affordance). *An interaction affordance I is a tuple, $I = (d, S)$, contained in a Thing Description Graph, $I \in G_{TD}$. Each interaction affordance I consists of:*

– *a datatype annotation $d \in D$, where $D = \{ArraySchema,\ BooleanSchema,\ NumberSchema,\ IntegerSchema,\ ObjectSchema,\ StringSchema,\ NullSchema\}$,*
– *a set of semantic annotations S containing all semantic annotations in triple format.*

Example 8. In the running example, the first interaction affordance I_1 (i.e. `temp`) contains the datatype element $d_1 =$ `NumberSchema` and the semantic annotation set $S_1 = \{($`[]`, `sosa:hasFeatureOfInterest`, `<http://ex.com/Room404>`$)\}$. Thus, I_1 is given as $I_1 = ($`NumberSchema`, $\{($`[]`, `sosa:hasFeatureOfInterest`, `<http://ex.com/Room404>`$)\})$.

After defining the Thing Description Graph and an interaction affordance, we now define the second part needed for the mapping: the RML mapping rule. In RML, so-called TriplesMaps define how non-RDF source data is mapped to RDF by specifying ontologies and the structure of triples. An RML mapping rule can contain multiple TriplesMaps.

Definition 9 (RML Mapping Rule). *Let R be the set of all TriplesMaps, where each TriplesMap describes how a different part of the output RDF data is generated. We call the set R the RML mapping rule.*

In SemWoT, we use a template-based approach, i.e., we have different TriplesMap templates that describe a mapping of Thing data using the SOSA/SSN ontology to model observations, the PROV-O ontology to model provenance information, and, if applicable, the QUDT ontology to model units of measurement. We define the RML template as follows:

Definition 10 (RML Mapping Rule Template). *Let T be the set of all TriplesMap templates t, where each TriplesMap template t describes how a different part of the output RDF data is generated.*

Example 9. In the running example, we have TripleMap templates for the measurement `result` t_r and the `semantic annotations` t_S. Therefore, in the running example T is given by $T = \{t_r,\ t_S\}$

Finally, we fill in the templates with the information for the corresponding interaction affordance. This process generates the final RML mapping rule, which can then be executed by an RML interpreter to produce RDF data.

Definition 11 (Generating RML Mappings). *Let $g_1 : I \times T \rightarrow R$ be the mapping function that combines the interaction affordance set with the mapping template set to create the final RML mapping rule.*

6 A SemWoT Mediator

Both the protocol mapping and the TD to RML mapping introduced in the previous sections are combined in our SemWoT approach. SemWoT integrates WoT devices into the Semantic Web by making data accessible via HTTP, regardless of the IoT protocol used by the device, and by making data available in RDF, regardless of the data format provided by the device.

In software development, the *Mediator* pattern [7] is used when a software component acts as a central interface between two otherwise incompatible components. This pattern mediates between system objects, allowing information to be exchanged despite data and functional incompatibilities. Therefore, we call a software implementing the SemWoT approach a SemWoT mediator.

A SemWoT mediator implements the mediator pattern with two interfaces: one for clients on the Semantic Web via a RESTful API that accepts and returns RDF data, and one for sensors and actuators using the WoT. Figure 1 shows the abstract representation of a SemWoT mediator.

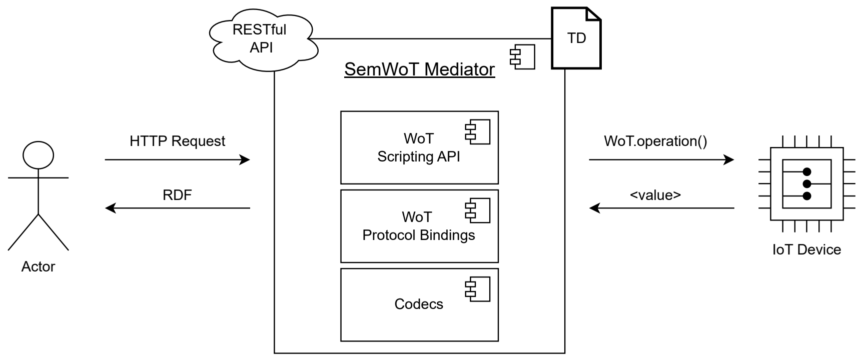

Fig. 1. Architectural structure of the SemWoT mediator

A SemWoT mediator uses the WoT Scripting API to interact with IoT devices through abstract WoT operations such as reading properties, writing properties, invoking actions, and subscribing to events [10]. The Scripting API uses a TD to create an instance of the Thing for these interactions. On the client side, the SemWoT mediator provides a RESTful API that allows HTTP requests to a Thing. The RESTful API is derived from the TD, with the access path consisting of the HTTP protocol scheme, followed by the authority consisting of mediator IP address and port, and finally the path consisting of sensor name, and affordance name [3]. For instance, in the running example, the URI of the `temp` property is given by "`http://<IP>:<port>/Flower/temp`".

To implement a RESTful Read-Write Linked Data interface on the SemWoT mediator, we need an ontology to describe tasks and interaction invocations, as well as a clear definition of interaction sequences for all interactions. The following subsections describe these aspects.

6.1 Interaction Invocation Ontology

We created a simple RDFS ontology to model the interaction patterns and functionalities in the context of a SemWoT mediator, following the LOT methodology [14] and used the Ontology Pitfall Scanner [15] for validation. A new ontology is required because the existing TD ontology only allows describing available interaction affordances, not that a concrete interaction affordance should be invoked, which we want to enable. The resulting ontology, called the Actionable IoT Ontology, uses the prefix `aio` and aims to describe the invocation of interaction affordances that a SemWoT mediator should perform. An RDF graph describing the activation of the precision mode with an input value of 1, based on the running example, is shown in listing 1.2.

Listing 1.2. The `aio` ontology describing the activation of the precision mode.

```
@prefix aio: <https://solidweb.me/anon/sbo/aio.ttl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

[] rdf:type aio:ActionInvocationInteraction ;
aio:hasInvocationInput [
 rdf:value "1"^^xsd:integer
 ] .
```

The central class in the `aio` ontology is `InteractionInvocation`, which has three subclasses to describe different types of interaction: the invocation of a writeProperty, the invocation of an Action, and the invocation of an Event subscription. readProperties do not need a separate class because the read operation can be invoked using an HTTP GET request without a body, and therefore no detailed descriptions are required. In addition to classifying interactions, the ontology also defines a property called `aio:hasStatus`, which is used to indicate the current state of a triggered WoT action. For example, after initialization, the status changes from *initializing* to *running*, and after completion, the status updates to *finished*. The ontology also associates interactions with their input and output information, such as datatypes and values, using the `hasInvocationInput` and `hasInvocationOutput` properties. Additional properties and classes have been reused from well-known ontologies such as SOSA/SSN and QUDT.

The Widoco documentation and the RDF data of the ontology can be found online[4].

[4] `https://solidweb.me/anon/sbo/aio.html`

6.2 Reading and Writing Properties

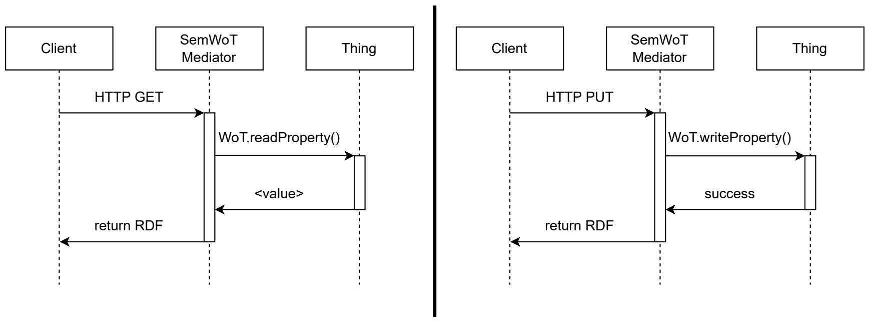

Fig. 2. Reading (left) and writing (right) of a WoT property using a SemWoT mediator as intermediary.

RESTful access to WoT properties allows information to be read from and written to a Thing. Property operations are near-instantaneous and can be pushed directly to the Thing through the mediator.

To read a WoT property of a Thing, a client sends an HTTP GET request to the mediator. The mediator maps the GET request to the Thing's protocol using the method introduced in section 4 and performs a read operation. Once the Thing returns the result, the mediator maps the data to RDF by filling in an RML mapping template using the approach described in section 5 and executes the mapping. The RDF data is then returned to the client via HTTP. The UML sequence diagram for reading a property is shown on the left side of Fig. 2.

To write a WoT property, a client sends an HTTP PUT request containing the task description in RDF using the `aio` ontology to the SemWoT mediator. The mediator extracts the task using SPARQL and translates the request into a write operation for the Thing's protocol. Once the data is successfully written, RDF data indicating a successful write operation is returned to the client. The UML sequence diagram for writing a property is shown on the right side of Fig. 2.

6.3 Invoking Actions

RESTful invocation of WoT actions is more complicated than simply reading or writing WoT properties, because WoT actions are long-running operations that do not return immediately.

To invoke a WoT action, a client sends therefore an HTTP POST request containing RDF data describing the action to be performed to the SemWoT mediator, which returns a location header. Meanwhile, the SemWoT mediator invokes the action on the Thing. The client can periodically check the status of

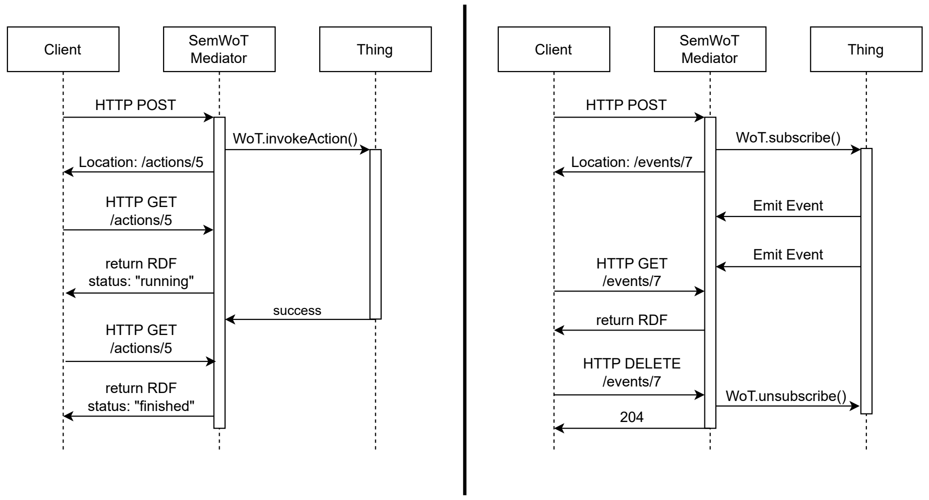

Fig. 3. Invoking a WoT action (left) and subscribing/unsubscribing from a WoT event (right) using the SemWoT mediator as an intermediary.

the action by polling the newly created RDF resource as defined by the location header. Once the WoT action has been successfully executed, the status in the RDF document changes from `running` to `finished`. The UML sequence diagram for invoking a WoT action is shown on the left side of Fig. 3.

6.4 Subscribing to Events

Subscribing and unsubscribing to WoT events in a RESTful manner is adjusted to handle the asynchronous nature of events. To subscribe to a WoT event, a client uses an HTTP POST request that contains RDF data that defines the specific event to subscribe to. The request creates a resource on the WoT mediator, and the client receives a new URI via the location header in the response. Meanwhile, the mediator subscribes to the specified event offered by a particular Thing. Once the Thing emits an event, the received data is mapped to RDF using RML, and the corresponding resource is updated. The client can use HTTP GET requests on the RDF resource to retrieve the latest emitted event data. To unsubscribe from the WoT event source, a client can send an HTTP DELETE request to the resource and the SemWoT mediator will unsubscribe from the event source of the Thing.

One risk associated with this approach is that events may be emitted more frequently than the client reads the RDF resources containing the event data. This can lead to missing event data from the client's perspective since only the latest event data is kept in the RDF resource. To mitigate the risk, the client needs to increase the polling rate.

The UML sequence diagram for subscribing and unsubscribing from a WoT event is displayed on the right side of Fig. 3.

7 Empirical Evaluation

In the empirical evaluation, we assess the request overhead introduced by the WoT mediator by comparing the execution time of direct access to a Thing's interaction affordances with access through the SemWoT mediator. To perform the evaluation, we implemented a SemWoT mediator as a prototype in JavaScript using the *express.js* framework to provide the RESTful API and *node-wot* to interact with Things using the WoT abstraction. Additionally, we implemented Things based on HTTP to allow easier evaluation of the performance. All code is open source and available at `https://anonymous.4open.science/r/SemWoT-Mediator-C518/`. For the hardware setup we used two Raspberry Pi 3Bs running Raspberry Pi OS, one to host the Things and one to host the mediator, and a notebook running Windows 10 as the client. All three devices were connected to the same wifi network.

We evaluated the performance of reading and writing properties, as well as invoking actions. However, we did not evaluate the performance of subscribing to events, because the relevant metric for events is not the subscription time, but the time from when the event is emitted by the device to when the data is available to the client. Due to the distributed nature of the system, precise time measurements were not possible, making it difficult to obtain consistent and reliable results. Therefore, we focused our evaluation on operations where timing could be accurately assessed.

Reading Properties The first diagram in Fig. 4 shows the response times for the `readProperty` operation. Direct access ranged from 14 to 23 ms, while access through the SemWoT mediator ranged from 41 to 60 ms.

Writing Properties The second diagram in Fig. 4 shows the `writeProperty` operation. Direct access response times ranged from 23 to 33 ms, while using the SemWoT mediator ranged from 39 to 52 ms.

Invoking Actions The third diagram in Fig. 4 shows the `invokeAction` operation, invoking a function with a processing time of 50 ms. Direct access times ranged from 71 to 82 ms, while using the SemWoT mediator increased response times to 112 to 129 ms.

Discussion The results show that while the SemWoT mediator introduces additional overhead in terms of response time due to processing tasks such as protocol translation and RML mapping generation and execution, the mediator maintains consistent performance. The overall consistency is beneficial for applications that require predictable timing. The trade-off between increased response times and the benefits of self-describing data and protocol abstraction provided by the SemWoT mediator must be considered based on the specific requirements of the application. For applications where minimal latency is critical, direct access may be preferable. However, for applications that require the flexibility of using HTTP to access a WoT device and benefit from the additional semantic

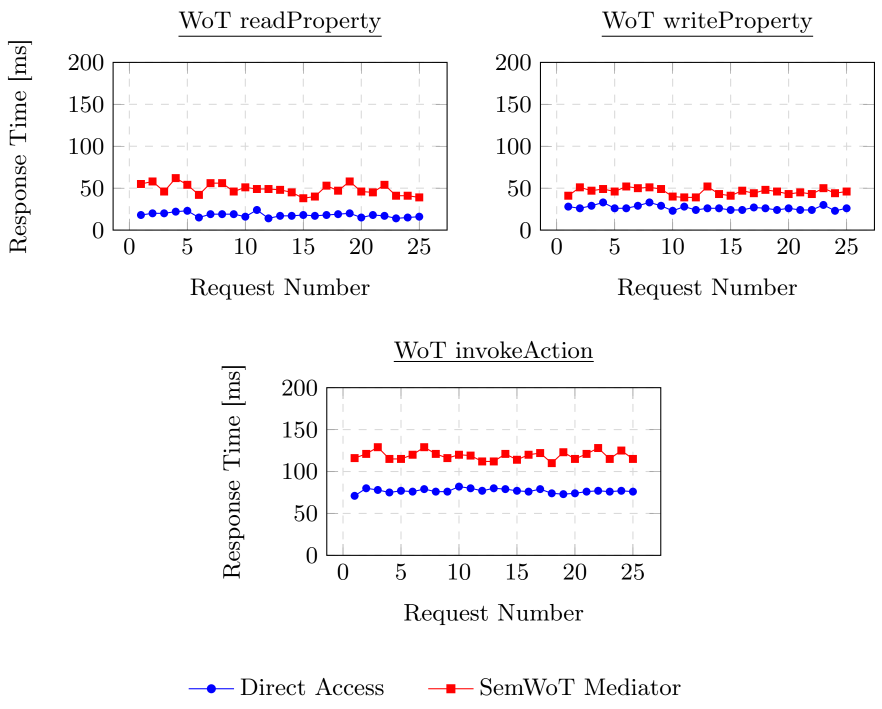

Fig. 4. Response times of different WoT operations comparing direct access to access using the mediator.

information provided by the SemWoT mediator, the additional overhead may be justified.

8 Conclusion and Future Work

In this paper, we introduced SemWoT, an approach that addresses the challenge of IoT data access by requiring only HTTP protocol support and eases further processing of gathered IoT data by making the data self-describing through the use of semantics. We developed a mediator that implements the mappings introduced by the SemWoT approach, defined interaction sequences for the mediator, and introduced the `aio` ontology to describe interaction invocations, enabling the direct integration of Things into the Semantic Web. Our evaluation showed that while the overhead added by the SemWoT mediator is relevant, it is consistent and does not fluctuate, providing a trade-off between ease of data access and processing versus request time. Future work will explore more advanced techniques for evaluating event data exchange performance, further optimizations to reduce mediator overhead, and the setup of a real-world, multi-sensor testbed for long-term testing.

References

1. Balaji, S., Nathani, K.o.: IoT technology, applications and challenges: a contemporary survey. Wireless personal communications **108**, 363–388 (2019)
2. Berners-Lee, T.: Read-write linked data. `https://www.w3.org/DesignIssues/ReadWriteLinkedData.html` (2018)
3. Berners-Lee, T., Fielding, R.T., Masinter, L.M.: Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Jan 2005). `https://doi.org/10.17487/RFC3986`, `https://www.rfc-editor.org/info/rfc3986`
4. Dimou, A., Vander Sande, M., Colpaert, P., et al.: RDF Mapping Language (RML). Specification proposal draft (2014)
5. Freund, M., Dorsch, R., Harth, A.: Applying the Web of Things Abstraction to Bluetooth Low Energy Communication. arXiv preprint arXiv:2211.12934 (2022)
6. Freund, M., Rott, J., Dorsch, R., et al.: FAIR Internet of Things Data: Enabling Process Optimization at Munich Airport. In: European Semantic Web Conference. Springer (2024)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH (1995)
8. Kaebisch, S., McCool, M., Korkan, E., Kamiya, T., Charpenay, V., Kovatsch, M.: Web of Things (WoT) Thing Description 1.1. `https://www.w3.org/TR/wot-thing-description/` (2023)
9. Käfer, T., Bader, S.R., Heling, L., Manke, R., Harth, A.: Exposing internet of things devices via rest and linked data interfaces. In: Proc. 2nd workshop semantic web technol. Internet Things. pp. 1–14 (2017)
10. Kis, Z., Peintner, D., Aguzzi, C., Hund, J., Nimura, K.: Web of Things (WoT) Scripting API. `https://www.w3.org/TR/wot-scripting-api/` (2023)
11. Koster, M., Korkan, E.: Web of Things (WoT) Binding Templates. `https://www.w3.org/TR/wot-binding-templates/` (2024)
12. Lagally, M., Matsukura, R., McCool, M., Toumura, K., Kajimoto, K., Kawaguchi, T., Kovatsch, M.: Web of Things (WoT) Architecture 1.1. `https://www.w3.org/TR/wot-architecture/` (2023)
13. Noura, M., Heil, S., Gaedke, M.: Webifying heterogenous internet of things devices. In: Web Engineering: 19th International Conference, ICWE 2019, Daejeon, South Korea, June 11–14, 2019, Proceedings 19. pp. 509–513. Springer (2019)
14. Poveda-Villalón, M., Fernández-Izquierdo, A., Fernández-López, M., García-Castro, R.: Lot: An industrial oriented ontology engineering framework. Engineering Applications of Artificial Intelligence **111**, 104755 (2022)
15. Poveda-Villalón, M., Gómez-Pérez, A., Suárez-Figueroa, M.C.: Oops!(ontology pitfall scanner!): An on-line tool for ontology evaluation. International Journal on Semantic Web and Information Systems (IJSWIS) **10**(2), 7–34 (2014)
16. Scheffler, M., Aeschlimann, M., Albrecht, M., et al.: FAIR Data Enabling New Horizons for Materials Research. Nature **604**(7907), 635–642 (2022)
17. Tosi, J., Taffoni, F., Santacatterina, M., et al.: Performance Evaluation of Bluetooth Low Energy: A Systematic Review. Sensors **17**(12), 2898 (2017)
18. Wilkinson, M.D., Dumontier, M., et al.: The FAIR Guiding Principles for Scientific Data Management and Stewardship. Scientific data **3**(1), 1–9 (2016)