2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA) | 978-1-6654-0032-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/IISA52424.2021.9555578

# Web of Things Functionality in IoT: A Service Oriented Perspective

Aimilios Tzavaras
*School of Electrical and Computer Engineering*
*Technical University of Crete (TUC)*
Chania, Crete, Greece
atzavaras@isc.tuc.gr

Euripides G.M. Petrakis
*School of Electrical and Computer Engineering*
*Technical University of Crete (TUC)*
Chania, Crete, Greece
petrakis@intelligence.tuc.gr

***Abstract*—The Web Thing Model of W3C defines a framework for integrating Things (e.g. devices) in the Web. It defines an information representation for Things based on JSON along with a set of RESTful services that enable access of Things on the Web. The Web Thing Model service (WTMs) is a novel implementation of the model and is compared against existing implementations selected from the Web. JSON Thing Descriptions (TD) and the operations supported by the REST API are the criteria to determine whether an implementation of Web Thing Model is according to the W3C model. We show that WTMs is the most complete implementation based on the requirements of the model. As a proof of concept, we show how WTMs can be integrated within a Service Oriented Architecture (SOA) for the IoT in order to support full-fledged Web Thing Model functionality.**

***Keywords*-Web services; IoT; Web of Things, Web Thing Model; Service Oriented Architecture;**

## I. Introduction

The Web of Things (WoT) [1] concept aims at unifying the world of Things connected on the Web. The WoT concept has received considerable attention by IoT vendors and by many investigators. The WoT Working group[1] is an ongoing effort to create standards-track specifications and test suites. The results of this effort are summarized by WoT abstract architecture [2] which suggests a list of possible operations to be supported by an WoT implementation[2]. Not all operations are supported by an implementation. The Semantic Web of Things (SWoT) [3] is the semantic extension of WoT that allows Things to become machine discoverable on the Web using Semantic Web tools such as SPARQL.

The Web Thing Model [4] of W3C, although not a standard (yet), it is more concrete and specifies the requirements that software or hardware vendors (who create products for the Web of Things) should meet. The Web Thing Model defines (a) the information model for Thing Descriptions (TD) based on JSON and, (b) a set of actions (i.e. services) together with their corresponding RESTful interface for accessing Things on the Web. The term Thing may refer to any device; a common temperature or pressure sensor, a window actuator, a smart TV, a Wi-Fi connected garage door or a smart car. Things become JSON objects representing their identity (i.e. a URI), properties, functionality (e.g. actions that Things may execute) and state information. They expose a RESTful interface on the Web in order to facilitate their interaction with other Things and services over the Web. Things are published on the Web (i.e. expose their identity, properties and functionality) so that they can be accessed by other services and be reused in Web applications.

WoT is far from being a reality to date. There is no universal application layer protocol to enable things and services to communicate. Instead, real-world devices may implement any of a wide range of application specific protocols (e.g. Bluetooth, MQTT, LoRa, ZigBee etc.). Web Thing model suggests that Things are accessed using Web protocols (notably HTTP). WoT does not depend on peculiarities of IoT protocols that would require an extra layer of complexity in an implementation. Ideally, WoT requires that Things receive (each one) an IPv6 address and have a Web server installed. However, this is not always possible especially for resource constraint devices. Although lightweight Web servers can be embedded in small devices, IoT devices usually feature limited resources and the solution is not optimal in terms of autonomy and cost. A workaround to this problem is to deploy a Web proxy on a server (or on a gateway) that keeps the virtual image of each Thing (e.g. a JSON representation). Web proxy implements a directory (e.g. a database) with all Things (i.e. instances, their types, descriptions and services supported). Things become part of the Web and can be accessed via their Web Proxy (i.e. they can be published, consumed, aggregated, updated and searched for).

The contributions of this work are summarized in the following: (a) we propose Web Thing Model service

[1]https://www.w3.org/WoT/wg/
[2]https://www.w3.org/WoT/IG/wiki/Implementations

978-1-6654-0032-9/21/$31.00~©~2021 IEEE

(WTMs), our implementation of Web Thing Model according to the requirements of Web Thing Model, (b) existing Web of Thing implementations are reviewed and compared with WTMs taking the requirements of W3C into consideration, (c) we show how WTMs can be integrated within a Service Oriented Architecture (SOA) for the IoT in order to support full-fledged WoT functionality.

The three candidate Web Thing Model implementations (namely, *Thingweb.node-wot*, *Webofthings.js* and WTMs) are compared based on their capacity to support the entire set of operations of the Web Thing Model specification. The comparison and the discussion followed, revealed that the WTMs is the only implementation that fully complies with the specification. It is realized as a RESTful Web service that communicates (i.e. with other services or clients) over HTTP in the cloud (and the Web) and implements the requirements, and all operations suggested by the Web Thing Model.

The performance of WTMs is evaluated in a smart city scenario [5]. The results of this evaluation reveal that WTMs is capable of responding in real-time under stress (i.e. with thousands of requests out of which many are executed in parallel). Finally, in order to convey Thing specific functionality and information (i.e. Thing identifier, descriptions and payloads) from the Web to an IoT application, WTMs is implemented as a proxy service of iSWoT [6], a prototype IoT Service Oriented Architecture (SOA) deployed on Google Cloud Platform (GCP). Cloud is the ideal environment for IoT applications deployment due to reasons related to its affordability (no up-front investment, low operation costs), scalability, easy maintenance and accessibility.

The Web Thing proxy service is presented in Sec. II. The proposed *WTMs* and selected reference implementations of the model are discussed and compared in Sec. III and Sec. IV respectively. Incorporation of the *WTMs* proxy in iSWoT architecture are discussed in Sec. V followed by conclusions in Sec. VI.

## II. Web Thing Proxy

The interconnection of Things to an application is commonly supported by sensor specific protocols (e.g. Bluetooth, Zigbee, Lora etc.) rather than by HTTP directly. It is plausible to assume that Things connect to a protocol translation service whose role is to convey Thing related data (i.e. identifier, description and payload) to the application using HTTP and JSON. This service runs on a gateway but it can be deployed within a proxy in the cloud (or server) equally well. IDAS backend IoT management[3] is a reference implementation of this service. It is the only service which is affected by the property of Things (e.g. a sensors) to use a specific protocol. Following IDAS, Things register to a Publication and Subscription service [7] in order to make their information available to users or other services based on subscriptions. Each time a Thing registers to Web Thing Proxy, a new entity is created in the Publication and Subscription service. Each time new data about a Thing becomes available, its corresponding component in Publication and Subscription service is updated and a notification is sent to all subscribed entities (i.e. users or services). It works in conjunction with any Publication and Subscription service such as ORION Context Broker[4], Scorpio[5] or RabbitMQ[6]. The database of the service (i.e. a MongoDB) is used to store all data about Things (i.e. descriptions, properties, measurements, actions, action executions and, subscriptions to Things). These data can be retrieved, updated or deleted by the Web Thing Model service. Notice that, Context Broker holds only the most recent information. History (past) data are forwarded to a history database (not part of Web Thing proxy). Fig. 1 illustrates the most essential parts (i.e. services) of the Web Thing proxy and its API interface.

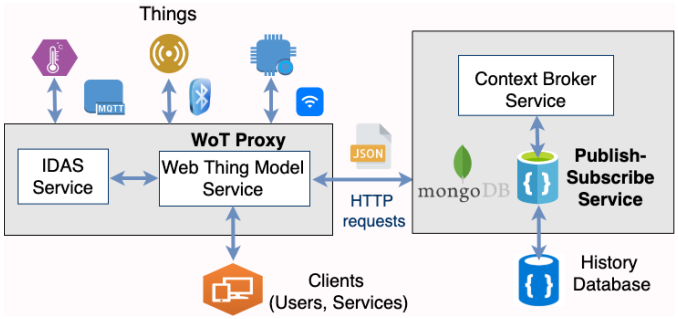


Fig. 1: Web Thing proxy service.

According to Web Thing Model specification, a Thing is identified by its resources, namely, a Web Thing Resource, a Model Resource, a Properties Resource, an Actions Resource (as long as the Thing supports actions), a Things Resource and a Subscriptions Resource. The Web Thing Resource is the root resource of the Web Thing model and it is used in order to provide a short and abstract description of a Web Thing. The Model Resource is meant to describe the values of Things properties as well as the actions that can be performed on Things. The Thing model is a more detailed description and it may also include helpful links (e.g. a link to a documentation file). The Properties Resource defines the properties of a Thing in general (e.g. pressure, humidity) and describes measurements related to a Thing property (e.g. temperature values provided by a temperature sensor) or the internal state of a Thing (e.g. the state of a

[3]https://fimac.m-iti.org/5a.php

[4]https://fiware-orion.readthedocs.io/en/master/
[5]https://github.com/ScorpioBroker/ScorpioBroker
[6]https://www.rabbitmq.com

smart door). The Actions Resource defines the allowed actions on a Thing, such as execution commands (e.g. a command sent by a client to a window actuator to open). The Things Resource is different from the Web Thing resource and it is used to describe specific operations on Things such as, registration of new Things to an application. Finally, the Subscriptions Resource is used to describe subscriptions to Web Things (especially to their actions and properties). For example, users and services can subscribe to the humidity property of a specific sensor and get notified on any changes of this information (i.e. new humidity value).

The data exchange format for Web Thing resources is based on JSON. The exact formatting depends on the particular services of the proxy service (e.g. NGSI[7] in the case of IDAS). The Web Thing Model introduces a list of operations which (in part or in full) can be offered by a Thing. In relation to Web Thing Resource, the model may allow an operation that retrieves or updates the description of a Thing in JSON. Similarly, for Thing Model (or for Properties Resource), Web Thing model may allow operations that retrieve or update the properties or actions (their values or state information respectively) on a Thing. In relation to Actions Resource, the model may allow an operation that retrieves the actions that a Thing may perform and, in addition, an operation for sending a command to a Thing to execute an action and operations to retrieve past action executions. In relation to Things Resource, the model may allow an operation that registers a Thing or an operation that retrieves all registered Things. Finally, in relation to Subscriptions Resource, the model may allow an operation that creates a new subscription to a Web Thing resource or, an operation that retrieves or updates the information of an existing subscription.

## III. Web Thing Model service

Web Thing Model service (WTMs) is an autonomous RESTful service in Python Flask and implements all the Web Thing Model operations on Things using HTTP. It provides a RESTful Web service designed to support all operations and use the same JSON payloads, API endpoints and response codes as described in the W3C specification. Specifically, WTMs supports all operations for retrieving and updating Thing descriptions and their properties, as well as all Thing model operations. It implements functions that send a command to a Thing (i.e. an actuator) to execute or retrieve actions and action executions, as well as functions that create, retrieve and delete subscriptions on Web Thing resources. In the following discussion, a hypothetical Smart Door (i.e. an actuator device) is used in most examples. It provides information on its state (i.e. open, closed or locked) and can receive an action command to open, close, lock or unlock. The DHT22 sensor[8] is also mentioned in some examples. In the following, WTMs operations are grouped by resource.

### A. Web Thing Resource

The first operation retrieves a Thing and is realized by issuing an HTTP GET request to the root URL of a Thing. The root URL of a Thing is its IP address and default port that follows the IP (e.g. http://34.122.93.207:5001/MySmartDoor in the case of the smart door). To retrieve the TD, an HTTP GET request is sent to Context Broker service, where the TD is stored. The JSON representation of the Thing contains its identifier (e.g. MySmartDoor) and possibly a characteristic name, a description, the date when the Thing entity was created and any other information given by the Thing owner. A Thing can be registered by its owner, who is responsible for setting the Thing description. A 200 OK response code is also returned as long as the operation is successful, similar to any other retrieval operation of the model.

The function of updating a Thing description, requires sending an HTTP PUT request to the root URL of a Thing, containing a JSON object in the request body. This object may contain new values for any of the Thing's attributes (except the identifier which cannot be updated). The operation is realized by sending an HTTP PATCH request to the Context Broker in order to update the existing Thing description by changing its attributes. A 204 NO CONTENT response code is returned if the operation is successful.

### B. Model Resource

The first operation intends to retrieve the model of a Thing by issuing an HTTP GET request to the /model endpoint of the root URL of a Thing. The request is forwarded to Context Broker service where the description of the Thing model is stored (i.e. using /v2/entities/MySmartDoor?type=Model instead of retrieving the entity that stores the TD by using /v2/entities/MySmartDoor?type=Thing). The JSON representation returned contains the identifier of the Thing along with any attribute which describes the model of the Thing (i.e. its properties, actions, etc.). Compared to TD, it is a more detailed description of the Thing.

The second operation intends to update the model of a Thing (i.e. update attributes of the Thing model description), by issuing an HTTP PUT request to the /model endpoint of the root URL of a Thing. The new attribute values are included in the request body (except identifier attribute that cannot be updated). The

[7]https://www.fiware.org/2016/06/08/fiware-ngsi-version-2-release-candidate/

[8]https://www.adafruit.com/product/385

operation is realized by forwarding an HTTP PATCH request to Context Broker service. A 204 NO CONTENT response is returned if the operation is successful.

### C. Properties Resource

The first operation intends to retrieve a list of properties. It is realized by issuing a GET request to the /properties endpoint of the root URL of a Thing. For example, in order to retrieve the properties of the smart door, an HTTP GET request is sent to the properties endpoint of the root URL of the smart door. A JSON array describing the Thing properties is returned. The array contains a JSON object for each specific property of the Thing. This object includes a short description of the property (e.g. identifier, name and any other attribute set by the Thing owner) and the current (i.e. last) measurement value of the Thing (e.g. last humidity value of DHT22) or the internal state of the Thing (e.g. the state of the door). So this operation returns a conceptual description as well as the last measured or state value for each property of the Thing. In the case of a smart door, only information about the state property of the Thing will be returned; in the case of DHT22 both temperature and humidity information will be included in the response JSON array.

The second operation on Properties Resource intends to retrieve the current value of a property. It requires sending an HTTP GET request to the /properties endpoint of the root URL of a Thing followed by the specific Thing property name as a path parameter (e.g. rootURL/properties/state for the smart door or rootURL/properties/temperature for DHT22). The Context Broker service allows storing only the most recent measurement values of a sensor in the case of DHT22 or, the last observed state of the smart door. The implementation can be modified to return the most (e.g. 10) recent values. Past measurements can be also stored in a history database (as shown in Fig. 1).

An update operation on the Properties Resource (e.g. update a specific property) requires sending an HTTP PUT request to the /properties endpoint of the root URL of a Thing followed by the name of the property. The new property value(s) and the new timestamp are included in the request body. For example, an HTTP PUT request is used to update the state property of the smart door (e.g. by changing its value to open). This is a property update operation which is different from an action (e.g. open the door) and not an action execution command to open the door. In response, WTMs forwards an HTTP PUT request to Context Broker service in order to update the existing property values and the property timestamp. A 204 NO CONTENT response and a header containing the property URL path are returned if the operation is successful.

The last operation on Properties Resource, updates multiple properties at once, by sending an HTTP PUT request to the /properties path of the root URL of a Thing followed by a path name. This operation is used in order to update the values of multiple properties of a specific Thing (e.g. the temperature and the humidity of DHT22) using a single HTTP request. The request body contains a JSON array with the new value and the new timestamp of each property. In turn, WTMs issues an HTTP POST request to Context Broker service in order to update the existing property values. The service utilizes the batch update operation of the Context Broker, which allows to create or update several entities with a single request. A 204 NO CONTENT response and a header containing the /properties endpoint of the root URL of the Thing are returned provided that the operation is successful.

### D. Actions Resource

The first operation retrieves a list of actions by issuing an HTTP GET request to the /actions endpoint of the root URL of a Thing. This operation is meant to return an array of descriptions for the actions that the Thing may perform (e.g. open, close, lock, unlock for the smart door). In turn, the WTMs issues an HTTP GET request to the Context Broker in order to retrieve the entity that holds this information in the form of a JSON array (i.e. containing an identifier and a name for each possible action). This information can be set by the Thing owner.

The second operation intends to retrieve all recent executions of a specific action and issues an HTTP GET request to the /actions endpoint of the root URL of a Thing followed by the specific action name as a path parameter. For example, a GET rootURL/actions/{actionName} (e.g. rootURL/actions/lock) will return an array of the recent executions of a specific action on the smart door (e.g. locking the smart door device), including information about the status of the action execution and a timestamp. This information can be retrieved by issuing a HTTP GET request to the Context Broker service. A JSON array is returned containing a separate object for each action execution. A time delay value might have been sent in the request body of a specific action execution, in order to schedule the execution of the action at a later time. The value attribute can be omitted in commands for opening, closing, locking and unlocking the smart door.

The next operation in meant to execute an action. An HTTP POST request is sent to the /actions endpoint of the root URL of a Thing followed by an action name in a path parameter (e.g. POST rootURL/actions/lock in order to lock the smart door). WTMs generates a unique identifier for each execution of an action which is stored in the Context Broker and can be used to retrieve the action. In fact, a random integer number is generated and its value is used to identify the action. The last operation retrieves the status of an action using its identifier as a

path parameter. An HTTP GET request is sent to the /actions endpoint of the root URL of a Thing followed by the name of the action and the execution identifier as a path parameter (e.g. GET rootURL/actions/lock/156). The operation is forwarded to the Context Broker where the specific action execution is stored.

### *E. Things Resource*

The first operation retrieves the list of Things registered to the proxy. It requires sending an HTTP GET request to the /things endpoint of the root URL of a Thing. WTMs forwards an HTTP GET request to the Context Broker service in order to retrieve all the entities of type Thing (i.e. GET v2/entities? typeThing). A JSON array with the description of Things registered to the proxy is returned.

The second operation registers a new device in a specific infrastructure (and the proxy). The operation requires sending an HTTP POST request to the /things endpoint of the root URL of a Thing. The request body must contain the TD of the new Thing. Following this request, WTMs issues an HTTP POST request with the new TD to the Context Broker. A 204 NO CONTENT response and a header containing the root URL of the Thing are returned as long as the operation is successful.

### *F. Subscriptions Resource*

These are operations for handling subscriptions to Things. The first operation creates a new subscription, so a user or service may subscribe to a specific Web Thing resource. A client (i.e. user or service) may subscribe to specific Thing properties or actions. According to Web Thing model, subscriptions are ideally supported using custom callbacks (i.e. Webhooks) which are naturally supported by Websocket protocol. In WTMs in particular, subscriptions are also realized via Webhooks and specifically using HTTP and the subscription mechanism of Context Broker service. An HTTP POST request is required in order to create a new subscription. More specifically, WTMs issues an HTTP POST request to Context Broker (i.e. to its /v2/subscriptions endpoint) in order to store the new subscription. The subscription request body is set by the subscriber (client or service). A response header containing the subscription identifier is returned as long as the operation is successful. In addition, an HTTP PATCH request is also sent to the database of Context Broker in order to store the new subscription identifier as a separate entity. A 200 OK response is then returned as long as the whole operation is successful. In the following, as a result of successful subscription operation, the subscribed user or service gets notified (i.e. receiving asynchronous notifications) on changes of Thing's state information (e.g. new temperature value). The subscription identifier is a 24-digit hexadecimal number, generated by the subscription mechanism of Context Broker service.

The second operation retrieves a list of subscriptions made to a specific Thing or Web Thing resource. The operation issues an HTTP GET request to the /subscriptions endpoint of the root URL of a Thing. WTMs in particular, issues an HTTP GET request to the Context Broker service where the subscriptions are stored. A JSON array containing all these subscriptions is returned. According to Web Thing model, all stored subscriptions are retrieved (not only the ones made to a specific Web Thing resource). However, WTMs enables retrieval (in a JSON array) of the subscriptions that refer to a specific resource (i.e. Thing). For example, in order to retrieve the subscriptions made to a registered smart door, an HTTP GET request is issued (i.e. GET http://34.122.93.207:5001/subscriptions).

The next operation is meant to retrieve a specific subscription using its subscription identifier as a path parameter. The operation requires sending an HTTP GET request on /subscriptions endpoint of the root URL of a Thing followed by subscription identifier as a path parameter (e.g. GET rootURL/subscriptions/5a82be4d093af1b95ac0f730). Following this, WTMs forwards the request to Context Broker service (i.e. to /v2/subscriptions/5a82be4d093af1b95ac0f730 service endpoint) where the specific subscription is stored. A JSON representation of the subscription is returned in response.

The last operation deletes a subscription using its subscription identifier. The operation issues HTTP DELETE request to the /subscriptions endpoint of the root URL of a Thing followed by the subscription identifier as a path parameter. WTMs in particular, issues an HTTP DELETE request to Context Broker service where the specific subscription is stored (i.e. DELETE /v2/subscriptions/identifier). The subscription is removed and a 200 OK response header is returned.

## IV. Web Thing Model Reference Implementations

Thingweb node-wot[9] is an implementation of WoT Scripting API[10] and enables the implementation Thing operations using a JavaScript API similar to the Web browser APIs. It provides an API Interface that allows scripts to interact with Things using Web protocols such as HTTP, HTTPS, CoAP, MQTT, Websockets. Not all WoT operations are supported by all protocols. Webofthings.js[11] is a lightweight and extensible implementation of WoT operations in Node.js for HTTP and Websockets protocols. Similar to WTMs, it is a typical

[9]https://github.com/eclipse/thingweb.node-wot
[10]https://www.w3.org/TR/wot-scripting-api/
[11]https://github.com/webofthings/webofthings.js/

implementation of the WoT operations suggested in the original WoT resource [1].

Thingweb node-wot and Webofthings.js are representative open-source implementations which are compared with WTMs, in terms of completeness: to what extent an implementation supports all operations foreseen by Web Thing Model specification (rather than those suggested by WoT). There is a lot of ambiguity as to what operations and protocols should be supported in WoT and, existing implementations differ significantly in both operations and protocols supported. The relevant comparison would open broader discussion about equivalence of operations in different contexts.

Similar to the Thingweb node-wot, some functions of Web Thing Model have not been implemented. For instance, the operation for updating a Thing description and the operation of updating multiple Thing properties at the same time using a single HTTP PUT request are not supported. As mentioned in Sec. III, users or services should be able to subscribe to Web Thing resources (e.g. properties, actions) and get notified on any new values or value changes. Webofthings.js supports the creation of subscriptions using the Websocket protocol. However, the rest of the subscription operations have not been implemented (i.e. would need a context broker to implement such operations on subscriptions). Similar to Thingweb.node-wot, operations on subscriptions (e.g. the retrieval of a list of subscriptions or of information about a specific subscription and the deletion of a subscription) are not supported. In contrast to both frameworks, WTMs aims to fully implement the Web Thing Model and attempts to realize the functionality of all services as described in the model. Table I summarizes the results of this comparison.

TABLE I: Comparison of Web Thing reference implementations.

| Operation | Web Thing Model service (WTMs) | Thingweb node-wot | Web-ofthings.js |
|---|---|---|---|
| *Retrieve a Web Thing* | √ | √ | √ |
| *Update a Web Thing* | √ | | |
| *Retrieve the model of a Thing* | √ | | √ |
| *Update the model of a Thing* | √ | | |
| *Retrieve a list of properties* | √ | √ | √ |
| *Retrieve the value of a property* | √ | √ | √ |
| *Update a specific property* | √ | √ | √ |
| *Update multiple properties at once* | √ | √ | |
| *Retrieve a list of actions* | √ | √ | √ |
| *Retrieve recent executions of an action* | √ | | √ |
| *Execute an action* | √ | √ | √ |
| *Retrieve the status of an action* | √ | √ | √ |
| *Retrieve a list of Web Things* | √ | √ | |
| *Add a Web Thing to a gateway* | √ | √ | |
| *Create a subscription* | √ | √ | √ |
| *Retrieve a list of subscriptions* | √ | | |
| *Retrieve information of a subscription* | √ | | |
| *Delete a subscription* | √ | | |

### A. Performance

WTMs is deployed using Docker in a medium flavor (2 vCPUs, 4,096Gb RAM and 20Gb SSD disk drive capacity) Virtual Machine (VM) in the Google Cloud Platform (GCP). Individual services (i.e. all WoT Model services, Orion Context Broker service and a MongoDB) are deployed within a Docker Engine as containers. Experimental results are taken using simulated (but realistic) data obtained from software simulating the operation of physical devices at a home and in a city environment. This allowed shifting the emphasis of the work from device specific functionality (e.g. IoT transmission protocols and vendor specific device functionality), to the actual performance of WTMs. To generate a large data set, the software produces pseudo-random measurements in the same value range and form as a real sensor.

WTMs is tested in a smart city scenario with 1,000 actuator Things that provide observations of two properties (e.g. temperature and pressure). Each Thing is capable of executing actions and receive subscriptions. Context Broker service stores 7,000 data entities in total comprising information about Things and hypothetical users. More specifically, the MongoDB of the Context Broker services comprises 1,000 Web Thing descriptions, 1,000 Thing model descriptions, 2,000 Thing properties (1,000 for each of the two observable properties), 1,000 possible Thing actions, 1,000 Web Thing action executions and, 1,000 subscriptions to Thing resources.

Apache Benchmark (AB)[12] is used to to stress WTMs with 1,000 simultaneous requests (for each operation) for varying values of concurrency representing operations executing in parallel. Table II summarizes the performance (i.e. response time) of the most representative WoT service requests for three values of concurrency. All measurements of time are averages over 1,000 re-

[12]https://httpd.apache.org/docs/2.4/programs/ab.html

TABLE II: Performance of WoT Model service

| Service request | Response Time (ms), c=1 | Response Time (ms), c=100 | Response Time (ms), c=200 |
|---|---|---|---|
| *Retrieve a list of properties* | 41.84 | 10,59 | 10.77 |
| *Update the model of a Thing* | 44.70 | 13.53 | 13.65 |
| *Update a specific property* | 47.88 | 16.73 | 17.10 |
| *Update multiple properties* | 59.62 | 28.43 | 28.75 |
| *Retrieve a list of Web Things* | 53.37 | 20.47 | 20.81 |
| *Retrieve a list of actions* | 43.07 | 8.74 | 8.84 |
| *Retrieve recent executions of an action* | 41.37 | 9.46 | 9.59 |
| *Retrieve a list of subscriptions* | 51.00 | 18.73 | 29.47 |
| *Execute an action* | 45.54 | 16.47 | 17.85 |

quests. In all cases, the performance of WoT requests improves with the simultaneous execution of requests (i.e. the Apache HTTP server switches to multitasking) reaching their lowest values for concurrency = 100. Even with concurrency = 200 the average execution time per request is close to real-time in most cases. All requests are forwarded to the MongoDB service of Context Broker service (in fact MongoDB is part of the service). The request that updates multiple properties (two properties in our case) is slower than a simple read (i.e. GET) request.

## V. iSWoT System

iSWoT [6] is an example architecture that incorporates WTMs proxy service. Building upon principles of SOA design, iSWoT is implemented as a composition of RESTful micro-services communicating over HTTP in the cloud. iSWoT is a Semantic WoT architecture integrating the following desirable characteristics: (a) it is highly configurable and modular and supports generation of fully customizable applications by re-using services and devices; (b) leveraging flow-based programming combined with SWRL (i.e. a rule-based language for ontologies in OWL), new applications can be generated with the aid of user-friendly interfaces; (c) all services are re-usable, implement fundamental functionality and offer a public interface allowing secure connections with other services (even third party ones); (d) it is also interoperable and expandable (i.e. services can be added or removed) while being secure by design: all services are protected by an OAuth2.0[13] mechanism. Access to services is granted only to authorized users (or authorized services) based on user roles and access policies.

[13]https://oauth.net

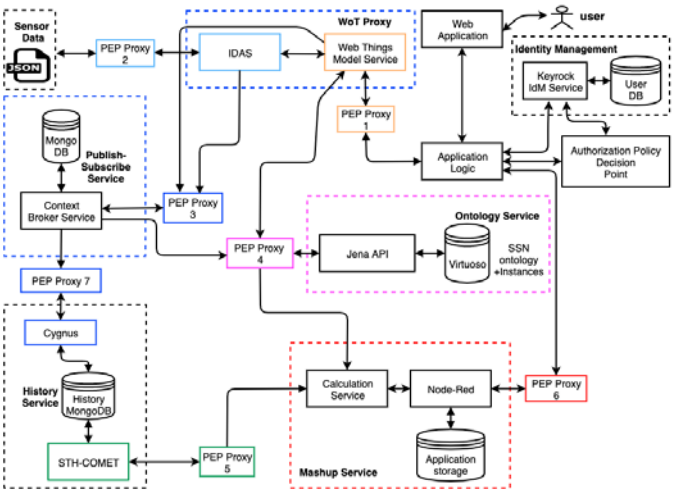


Fig. 2: iSWoT architecture.

Fig. 2 illustrates iSWoT's architecture. Besides WoT proxy, the following groups of services are identified:

*a) Publication and Subscription:* Things registered to WoT Proxy can publish information to this service. It receives measurements from devices registered to IDAS service and makes this information available to other services and users based on subscriptions. Each time a new sensor registers to WoT Proxy, a new entity is created in the service (and also to the ontology). Each time a Thing value (e.g. sensor measurement) becomes available, this component is updated and a notification is sent to entities subscribed to the Thing. The service holds the most recent values from all registered sensors in a MongoDb database. History (past) measurements are forwarded to History database.

*b) History database:* Collects data flows from Publication and Subscription service. The time series created from the history of data are stored in a MongoDB as (a) raw (unprocessed) values as received from devices and, (b) aggregated (processed) values. More specifically, maximum, minimum and average values over predefined time intervals (e.g. every hour, day, week etc.) are stored. This service is implemented using Cygnus[14]. The History database is implemented using MongoDB. STH-COMET[15] implements a query interface for MongoDB.

*c) Ontology:* It is the knowledge base component of iSWoT that handles IoT general purpose and application specific knowledge. The term Thing stands either for physical entities (e.g. Web devices) or for virtual entities (i.e. definitions of Thing categories). Physical entities are related (i.e. are instances) of virtual entities in SOSA ontology [8]. iSWoT supports seven sensors types (i.e. atmospheric pressure, temperature, humidity, luminosity, precipitation, human presence at home and wind speed).

[14]https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Cygnus
[15]https://fiware-sth-comet.readthedocs.io/en/latest/index.html

All are defined as subclasses of SOSA class Sensor.

*d) Application Mashup:* This service facilitates development of new applications by re-using information residing in History database and the ontology. The service is realized with the aid of Node-Red[16], an open-source flow-based programming tool for the IoT. Applications created are stored as a JSON entity in Application storage (i.e. a MongoDb database).

*e) Application logic:* Application logic orchestrates, controls and executes services running in the cloud. When a request is received (from a user or service), it is dispatched to the appropriate service. User requests are issued on the Web interface. First, a user logs in to iSWoT with a login name and password. The user is assigned a role (by the cloud administrator) and receives a token encoding his or her access rights (i.e. authorization to access iSWoT services). This is a responsibility of the User identification and authorization service. Each time application logic dispatches the request to another service, the token is attached to the header of the request. It is a responsibility of the security mechanism to approve (or reject) the request. In iSWoT, all public services are protected by a security mechanism. iSWoT users can access the system using a Web interface. Application owners can issue requests for available devices and subscribe to selected devices; customers can issue queries to select applications available for subscriptions.

*f) Security:* Implements access control services based on user roles and access policies. Initially, users register to iSWoT to receive a login name, a password and a role (i.e. customer, application owner, or infrastructure owner) encoding user's access rights. This is a responsibility of the cloud administrator. Once a user is logged-in, she/he is assigned an OAuth2 token encoding her or his identity. The token remains active during a session. A session is initialized at login and remains active during a time interval which is also specified in advance. User respective access rights are described by means of XACML[17] (i.e. a vendor neutral declarative access control policy language based on XML). Keyrock identity manager[18] is an implementation of this service. For each user, a XACML file is stored in Authorization Policy Decision Point (PDP)[19] service; user profile information is stored in User database.

Services offering a public interface are protected by a security mechanism (i.e. they do not expose their interface to the Web without protection). This security mechanism is realized by means of Policy Enforcement Proxy (PEP)[20] service. Each public service is protected by a separate PEP service. It is a responsibility of this service to approve or reject a request to the protected service. Each user request is forwarded to Application logic which dispatches the request to the appropriate service.

[16]https://nodered.org

[17]https://fiware-tutorials.readthedocs.io/en/latest/administrating-xacml/index.html

[18]https://keyrock.docs.apiary.io/#reference/keyrock-api/role

[19]https://authzforce-ce-fiware.readthedocs.io/en/release-5.1.2/

[20]https://fiware-pep-proxy.readthedocs.io/en/latest/

## VI. Conclusions

Web Thing Model service (WTMs) is our proposed implementation of the Web Thing Model specification of W3C. Compared to existing implementations, WTMs is complete (i.e. implements all Web Thing Model model operations) while being more flexible in certain cases allowing WoT operations to address certain Thing properties rather then handling the Thing Descriptions as a unit. In relation to future work, HTTPS protocol will eventually replace HTTP as a secure solution for the transmission of confidential information. Incorporating trust evaluation mechanisms for dealing with IoT risks due to malicious behavior of IoT [9] would be an important add-on to the implementation.

## References


[1] D. Guinard and V. Trifa, *Building the Web of Things*. Greenwich, CT, USA: Manning Publications Co., 2016. [Online]. Available: https://webofthings.org/book/

[2] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, and K. Kajimoto, "Web of Things (WoT) Architecture," 4 2020, w3C Recommendation. [Online]. Available: https://www.w3.org/TR/wot-architecture/

[3] A. Rhayem, M. B. A. Mhiri, and F. Gargouri, "Semantic Web Technologies for the Internet of Things: Systematic Literature Review," *Internet of Things*, vol. 11, pp. 1–22, 9 2020. [Online]. Available: https://doi.org/10.1016/j.iot.2020.100206

[4] "Web Thing Model," Aug. 2015, w3C member submission. [Online]. Available: http://www.w3.org/Submission/wot-model/

[5] G. Myrizakis and E. G. Petrakis, *"iHome: Secure Smart Home Management in the Cloud and the Fog*. IOS Press, Advanced in Parallel Computing, 2020, pp. 237–263. [Online]. Available: https://ebooks.iospress.nl/volumearticle/53830

[6] S. Botonakis, A. Tzavaras, and E. G. Petrakis, "iSWoT: Service Oriented Architecture in the Cloud for the Semantic Web of Things," in *Advanced Information Networking and Applications (AINA 2020)*, Cham, Mar. 2020, pp. 1201–1214. [Online]. Available: https://doi.org/10.1007/978-3-030-44041-1_103

[7] M. O'Riordan, "Everything You Need To Know About Publish/Subscribe," online, 2021, the ABLY platform. [Online]. Available: https://ably.com/topic/pub-sub

[8] K. Janowicz, A. Haller, S. J. Cox, D. L. Phuoc, and M. Lefrançois, "SOSA: A Lightweight Ontology for Sensors, Observations, Samples, and Actuators," *Journal of Web Semantics*, vol. 56, pp. 1–10, May 2019. [Online]. Available: http://dx.doi.org/10.1016/j.websem.2018.06.003

[9] T. Wang, S. Zhang, A. Liu, Z. A. Bhuiyan, and Q. Jin, "A Secure IoT Service Architecture with an Efficient Balance Dynamics Based on Cloud and Edge Computing," *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 4831–4843, 6 2019.