SoftwareX 13 (2021) 100661

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

WLDT: A general purpose library to build IoT digital twins

Marco Picone *, Marco Mamei, Franco Zambonelli

Department of Sciences and Methods for Engineering, University of Modena and Reggio Emilia, Italy

ARTICLE INFO

Article history:
Received 3 August 2020
Received in revised form 7 January 2021
Accepted 7 January 2021

Keywords:
Internet of Things
Digital twin
Library
Software agent

ABSTRACT

Digital twins are virtual software copies of physical objects and systems, and represent a strategic technology enabler to support Internet of Things devices and systems. Existing software frameworks for digital twins mainly operate in the cloud and are based on platform-specific solutions, harming interoperability and adaptability. However, it is getting recognized that Internet of Things and digital twins architectures can take advantage of microservices and platform-independent distributed architectures (also on the edge), promoting higher scalability, adaptability, and interoperability. In this context, we introduce WLDT (White Label Digital Twins), a general-purpose library that allows developers to create digital twins in terms of modular, adaptable, and inter-operable software agents. Among different features, WLDT library supports multiple standard protocols, caching, software processing pipeline and metrics monitoring.

Code metadata

Current code version	v1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-20-00018
Code Ocean compute capsule	*None*
Legal Code License	GPL-3.0
Code versioning system used	Git
Software code languages, tools, and services used	Java and Android
Compilation requirements, operating environments & dependencies	Java 8 or greater
If available Link to developer documentation/manual	https://github.com/wldt/wldt-core
Support email for questions	marco.picone@unimore.it

1. Motivation and significance

In the Internet of Things (IoT) arena, a Digital Twin (DT) is a comprehensive software representation of an individual physical object (e.g., a single IoT device or a more complex composite machinery), reflecting properties, conditions, and behavior of the physical object through models and data [1]. A physical object and its associated DT mutually communicate and collaborate with each other through bidirectional interactions related, for example, to telemetry or to incoming commands and configurations from external applications.

Since its introduction [2], the DT concept has proved very effective, and has been adopted in a variety of uses cases and application scenarios [3–5]. Gartner identifies DTs as one of the top 10 strategic trends and the forecast previews is that half of all corporations might be using them by 2021 [6,7]. One of the key benefits of DTs is to provide a solid, standard and scalable abstraction layer on top of physical assets, allowing authorized applications to easily and securely interact with a device without the need to be aware of the complexity related to data collection and networking. Unfortunately, in the design and implementation of IoT systems based on DTs, current technologies and libraries exhibit several shortcomings.

The lack of standards or common agreements for DTs design and development has led to the proliferation of several platform-specific solutions: IBM DTs are different from AWS ones, called “shadows” and from MS Azure ones, called “replicas”. Authors in [7] emphasizes that the potentials of DTs are harmed by the existing fragmentation and heterogeneity, where each model is built from scratch without common methods, standards or norms.

* Corresponding author.
E-mail address: marco.picone@unimore.it (Marco Picone).

https://doi.org/10.1016/j.softx.2021.100661

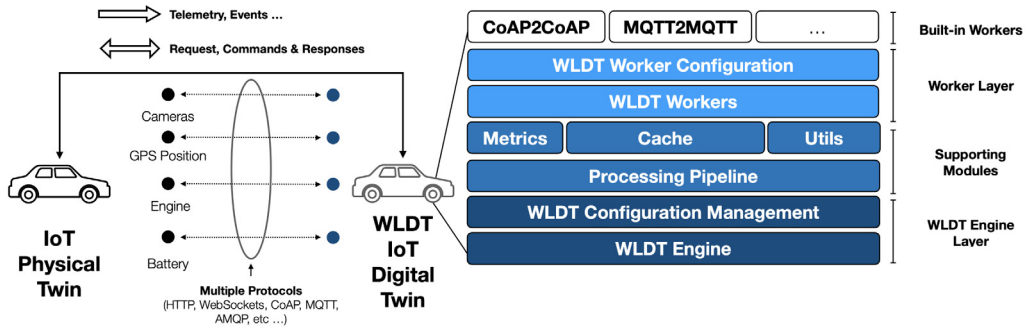

Fig. 1. Basic structure and main modules of White Label Digital Twin.

Open source organizations and consortiums together with industries (e.g., the IoT Eclipse Foundation[1]) proposed their open platforms for DT management. The Eclipse Ditto project[2] represents the state of the art of open source frameworks for DTs management and orchestration. It has been designed to be executed in the cloud and to simplify the backend management of DTs through APIs and SDKs (Software Development Kits), by targeting already connected things, customer applications and services. However, such frameworks still lacks flexibility and modularity. In fact, they are mainly focused on a monolithic vision of DTs, where the entire complexity of a physical object is managed by a single software entity, without the possibility to handle each DT's feature and task as an independent and flexible agent. Customization and adaptation implies direct interventions on the physical object or on intermediate modules (e.g., gateways and hubs) when the smart object's software update is unfeasible.

To overcome the limitations and constraints of existing DTs solutions, we developed a new Java library, called WLDT (White Label Digital Twin), designed to maximize modularity, re-usability and flexibility. In particular, WLDT focuses on the simplification of the design and development of DTs, and provides a set of core features and functionalities for their widespread adoption in multiple application scenarios. WLDT integrates a multi-threading core engine that is able to run multiple independent components at the same time, so as to effectively shape the behavior of each DT and its relationship with the physical counterpart. A set of built-in IoT features and modules provide an out-of-the-box mirroring for smart objects using both Message Queue Telemetry Transport (MQTT) [8] and/or Constrained Application Protocol (CoAP) [9]. Furthermore, the internal software processing pipeline system allows to dynamically customize the management for incoming and outgoing packets in order to adapt the behavior to the target use case and the physical counterpart. The internal caching system makes it possible to quickly store and retrieve operational data, thus improving performances and reducing the communication response time. WLDT has been also designed and developed with the characteristic of modeling each DT as an independent and autonomous software component. It enables to design microservices oriented IoT architectures [10,11], thus overcoming the limitations existing monolithic and legacy solutions by decoupling the responsibilities among multiple independent components.

The aim of WLDT is to become an enabling building block for the design and development of DT-driven IoT applications. The main users and key actors envisioned to interact with the proposed library are mainly software developers operating in IoT ecosystems, both at the edge and the cloud level. WLDT can be integrated and used without any further development or personalization by relying on the built-in MQTT and CoAP workers or can be extended through the creation of new modules and connectors to support specific target communication protocols or data flows. Any additional module can be re-used across multiple deployments thanks to the provided modular and configurable software architecture.

2. Software description

A DT implemented with the WLDT library is an independent software agent implementing all the features and functionalities of its physical counterpart. It can be deployed and executed in the cloud or at the level of edge computers. As illustrated in Fig. 1, a DT can be attached to a physical thing in order to create and maintain its virtualized replica by mirroring existing resources and extending the provided functionalities through additional modules and components.

2.1. Software architecture

The architectural layers presented in Fig. 1 schematically depicts existing components and how they are organized in the WLDT core. The basic layer of the solution is the "WLDT Engine" designed to handle and orchestrate available and active modules – denoted as Workers – defining the behavior of the DT. A Worker is the active module of the library and it is designed to implement a specific DT's task or feature related for example to the synchronization with the original physical counterpart through a target IoT protocol. WLDT Workers' implement all the available communication protocols supported by a DT involving both standard and well known protocols such as CoAP, MQTT, HTTP or WebSocket. Legacy protocols may be also supported in specific IoT deployments through the implementation of dedicated modules. Each worker is responsible to handle both Request/Response or Pub/Sub communication paradigms and the synchronization task required to manage both incoming and outgoing packets. Both the WLDT engine and the workers are characterized by multiple configuration options in order to easily change and adapt the DT behavior according to the target deployment and use case. The WLDT Configuration Manager is responsible to handle engine's parameters associated to DT's unique identifier, namespace, startup delay and the usage of the internal metrics and monitoring system. On the other hand each Worker can define its own personalized configuration through the WLDT Worker Configuration layer in order to retrieve the operational parameters useful for the implementation of its behavior (e.g., physical device endpoints, target Pub/Sub topics and RESTful resources).

Since the aim of the library is to support scalability and extensibility, the possibility for developers to quickly define dynamic behaviors into existing or new WLDT workers has been introduced in the library through the "Processing Pipeline" layer.

[1] Eclipse IoT - https://iot.eclipse.org/.

[2] Eclipse Ditto - https://www.eclipse.org/ditto/.

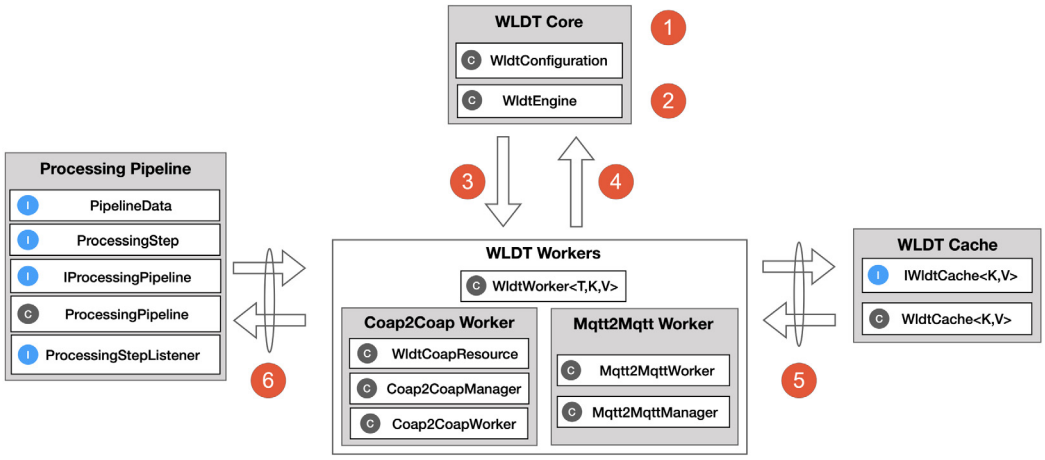

Fig. 2. Relationships between WLDT core objects and workers. The Engine configures and executes the workers responsible to shape DT's behavior by exploiting also the internal caching and software processing pipeline modules.

Developers can define a list of personalized software processing steps sequentially executed by the target worker and dedicated for example to the management of domain-specific incoming/outgoing packets, the integration with an external third party services or to data format translation and adaptation. These steps can be also dynamically loaded and re-used across multiple DT instances to maximize code re-usability. Furthermore, to support development activities, the library provides also an internal caching system where each module or entity can create its internal cache with a simplified and unified solution. The relationship among provided components is shown in Fig. 2 and can be summarized as follow: (i) The WLDT engine starts with an initial configuration and the associated list of workers that should be executed to shape DT's behavior; (ii) it instantiates the specified workers with the associated target configuration and the required processing pipeline (if needed); (iii) the engine executes each worker on an independent thread; (iv) active workers can send callbacks and notifications related to their operational phases (start, stop, error, warning, etc.) to the core engine; (v) they can write and read data from and to the internal caching system in order to support their implementation and (vi) workers can also use configured Processing Pipelines to customize their activities and the adaptation of incoming and outgoing messages.

2.2. Software functionalities

This section details some of the main functions of the WLDT library.

2.2.1. Internal data caching system

The library provides an internal shared caching system that can be adopted by each worker specifying the typology of its cache in terms of key and value class types. The interface `IWldtCache<K,V>` defines the methods for a WLDTcache and a default implementation is provided by the library through the class `WldtCache<K, V>`. Each cache instance is characterized by an string identifier and optionally by an expiration time and a size limit. An instance can be directly passed as construction parameter of a worker or it can be internally created for example inside a processing pipeline to handle cached data during data synchronization.

2.2.2. Processing pipelines

The Processing Pipeline is a configurable chain of software steps implemented and organized by the developer in order to personalize the DT's actions through the WLDT library. Each step can be re-used across multiple pipelines in order to maximize re-usability and modularity. A pipeline and its steps are defined through the interface `IProcessingPipeline` and the class `ProcessingStep`. Main methods to work and configure the pipeline are: `addStep()`, `removeStep()` and `start()`. The `ProcessingStep` and `PipelineData` classes are used to describe and implement each single step and to model the data passed through the chain. A step takes as input an initial `PipelineData` value and produces as output a new one of the same type. Two listeners classes have been also defined (`ProcessingPipelineListener` and `ProcessingStepListener`) to notify interested actors about the status of each step and/or the final results of the processing pipeline through the use of methods `onPipelineDone(Optional<PipelineData> result)` and `onPipelineError()`.

2.2.3. Monitor metrics and performance

The library allows the developer to easily define, measure, track and send to a local or remote collector all the application's metrics and logs. This information can be also useful to dynamically balance the load on active DTs operating on distributed clusters or to detect unexpected behaviors or performance degradation. The library implements a singleton class called `WldtMetricsManager` exposing the methods `getTimer(String metricId, String timerKey)` and `measureValue(String metricId, String key, int value)` used to track elapsed time of a specific processing code block or with the second option to measure a value of interest associated to a key identifier. The WLDT metric system provides by default two reporting option allowing the developer to periodically save the metrics on a local CSV file or to send them directly to a Graphite[3] collector node.

2.3. MQTT to MQTT worker

The first built-in worker is implemented through the class `Mqtt2MqttWorker` providing a configurable way to automatically synchronize data between the physical and the digital entities over MQTT. An MQTT physical device can be at the same time a data producer or consumer for example to publish telemetry data and to receive external commands at the same time. Developers can use up to four different types of topics inspired by the categorization provided in the open source projects Eclipse Hono[4] and Ditto[5] (also through a template placeholders engine[6])

[3] Graphite Metrics - https://graphiteapp.org/.
[4] Eclipse Hono - https://www.eclipse.org/hono/.
[5] Eclipse Ditto - https://www.eclipse.org/ditto/.
[6] Mustache template engine - https://mustache.github.io/.

to dynamically synchronize topics according to available device and resource information. As illustrated in the following example, available topics typologies belong to *telemetry*, *events*, *command requests* and *command responses* allowing the granular mirroring of a physical device through the topics mapping.

```
Mqtt2MqttConfiguration mqttConf = new
    Mqtt2MqttConfiguration();
mqttConf.setOutgoingClientQoS(0);
mqttConf.setDestinationBrokerAddress("127.0.0.1");
mqttConf.setDestinationBrokerPort(1884);
mqttConf.setDeviceId("id:97b904ada0f9");
mqttConf.setDeviceTelemetryTopic("telemetry/{{
    device_id}}");
mqttConf.setEventTopic("events/{{device_id}}");
mqttConf.setBrokerAddress("127.0.0.1");
mqttConf.setBrokerPort(1883);

WldtEngine eng = new WldtEngine(new WldtConfiguration
    ());
eng.addNewWorker(new Mqtt2MqttWorker(eng.getWldtId(),
     mqttConf));
eng.startWorkers();
```

Listing 1: Example a WLDT implementation using the built-in MQTT to MQTT worker to automatically create a DT of an existing MQTT physical object

2.4. CoAP to CoAP worker

The second core built-in IoT worker is dedicated to the seamless mirroring of standard CoAP physical objects. The CoAP protocol through the use of CoRE Link Format [12] and CoRE Interface [13] provides by default both resource discovery and descriptions. It is possible for example to easily understand if a resource is a sensor or an actuator and which RESTful operations are allowed for an external client. Therefore, a WLDT instance can be automatically attached to a standard CoAP object without the need of any additional information. As illustrated in the following example, the class `Coap2CoapWorker` implements the logic to create and keep synchronized the two counterparts using standard methods and resource discovery through the use of "/.well-known/core" URI in order to retrieve the list of available resources and mirror the corresponding digital replicas.

```
Coap2CoapConfiguration coapConf = new
    Coap2CoapConfiguration();
coapConf.setResourceDiscovery(true);
coapConf.setDeviceAddress("127.0.0.1");
coapConf.setDevicePort(5683);

WldtEngine wldtEngine = new WldtEngine(new
    WldtConfiguration());
wldtEngine.addNewWorker(new Coap2CoapWorker(coapConf)
    );
wldtEngine.startWorkers();
```

Listing 2: Example a WLDT implementation using the built-in CoAP to CoAP worker to automatically create a DT of an existing CoAP physical object

2.5. Experimental evaluation

In order to understand the performance of the proposed WLDT library and its current implementation a group of experiments have been defined focusing on measuring: (i) the introduced delay compared with the main state of the art reference; (ii) computational and memory costs; and (iii) modularity and development complexity. Conducted tests have been performed on a local Linux Edge Node equipped with an i7 Intel CPU and 16GB of RAM. As first evaluation, we compared the WLDT library with the Eclipse Ditto project measuring the introduced overhead delay for both MQTT and CoAP smart objects. Evaluated configurations take into account both objects that can directly integrate Ditto's SDK and things that cannot be customized requiring an intermediate module to communicate with. External consumer applications has been also implemented to test and measure the performance of the bidirectional communication through the DTs.

Fig. 3(a) shows the introduced delay as a function of the message rate with a fixed Payload size of 100 Bytes. On average the overhead introduced by Ditto is significantly higher with respect to the performance obtained by WLDT DTs both for MQTT and CoAP. The same trend is also confirmed by Fig. 3(b) considering instead the Payload size variation and a fixed message rate of 10 msg/s. The obtained performance are attributable to the fact that the Ditto framework has been designed to provides a set of extensive and inventory-oriented features for DTs management and communication with a structured storage and multiple architectural layers. This monolithic design introduce a relevant overhead if compared with the effective one-to-one DT mirroring provided by the presented library. Ditto remains excellent for a centralized DT management and it can potentially work side-by-side with the WLDT library in order to combine the advantages of both solutions.

Graphs in Figs. 3(c) and (d) analyze the CPU and the memory heap usages for CoAP and MQTT DT instances taking into account different configurations in order to understand how and if the performance will be affected over a period of approximately 15 min and a continuous rate of communications and data exchanges for devices with 10 distinct IoT resources. Presented results illustrate how DTs instances efficiently mirror a physical IoT device consuming a limited amount of memory (8 Mbytes for MQTT and 10 Mbytes for CoAP) and computational resources allowing to execute multiple DTs on the same computing infrastructure.

The presented library has been also successfully experimented for the creation of IoT DTs during the victorious Droidcon MEC (Multi-access Edge Computing) Hackathon 2020[7] in the context of an innovative Smart Cities experimentation. The library has been adopted to implement DTs of Road Side Units (RSU) and moving vehicles responsible to uniform data formats from heterogenous sources and bi-directionally communicate with an Edge Traffic Information System (E-TIS). With the aim to illustrate WLDT development complexity, Figs. 3(e) and (f) respectively report the required number of lines of code and the associated size footprint related to vehicle and RSU DTs and a shared Sensor Measurement Lists (SenML) data management module with respect to the size of the core library. Results shows how, thanks to the presented modular architecture, it is possible to easily and effectively digitalize a physical entity extending also its behavior overcoming the limitations and the heterogeneity of deployed legacy physical devices.

3. Illustrative examples

In the following subsections we present three illustrative and really implemented examples in order to highlight the core functionalities of the presented library. All the illustrated solutions have been developed and tested through the use of the WLDT library and have been also released as open source reference projects in the official GitHub organization and repositories.

[7] Droidcon MEC Hackathon 2020 - https://it.droidcon.com/2020/hackathon/.

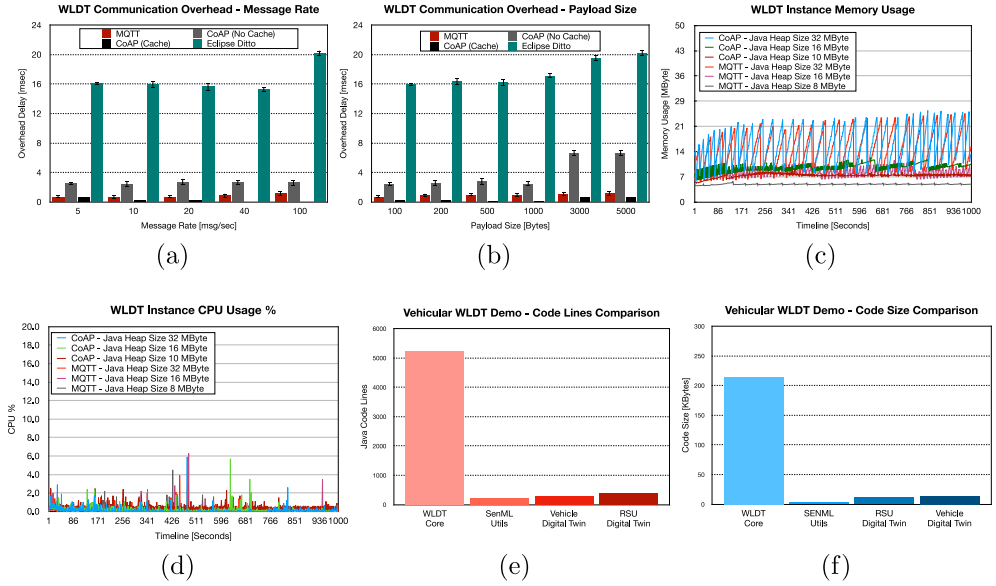

Fig. 3. (Average WLDT Communication Overhead with respect to Message Rate (a) and Payload size (b). Memory (c) and CPU (d) consumption of a WLDT instance. Code statistics in terms of code lines (e) and size (f) to digitalize MQTT IoT objects in a vehicular use case.

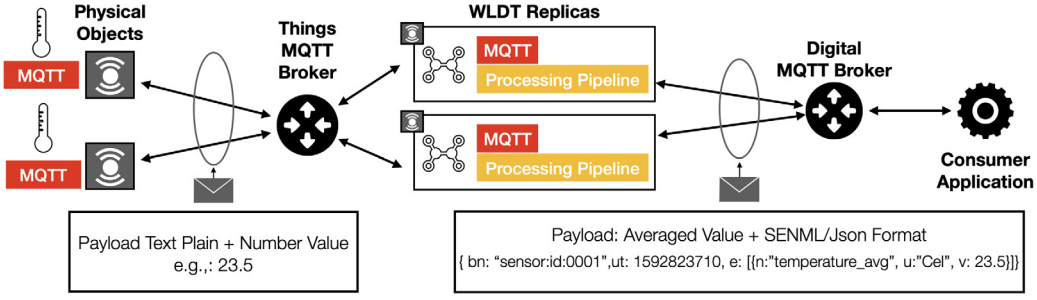

Fig. 4. Example a MQTT WLDT instance with a software pipeline processing raw values and forwarding the resulting as SenML packets.

3.1. MQTT to MQTT & processing pipeline

The first example, depicted in Fig. 4, focuses on an application scenario where multiple physical IoT MQTT objects (e.g., associated to temperature sensors) are mirrored into DTs allowing the protection of the core layer by means of decoupling the physical and the digital counterparts. The external direct access to the real device will be limited and the interaction with applications and consumers is instead securely handled by the virtual replicas. Furthermore, the scenario takes into account a custom processing pipeline on each DT instance in order to average received raw values and forward them in a standard format using the SenML data format [14]. The processing pipeline considers two independent and dedicated processing steps: (i) `MqttAverageProcessingStep`: uses the internal caching system to keep a buffer of n samples producing a new output value with the averaged value of the received measurements; and (ii) `MqttSenmlBuilderProcessingStep`: formats incoming data from the previous step using SenML+Json, the obtained result is used by the MQTT worker and forwarded to the external broker on a dedicated topic. Both involved steps work with a `MqttPipelineData` class implementing the interface `PipelineData` maintaining the message payload and the original target topic across all the processing chain.

3.2. Legacy protocol worker - Philips Hue lights

This second example, depicted in Fig. 5, focuses on the creation of a custom worker to seamlessly mirror physical Philips Hue[8] light bulbs into their digital and standard CoAP replicas. The Philips solution provides a set of legacy HTTP APIs different from IoT standard protocols in terms of communication and data format. Through the creation of a dedicated `PhilipsHueLightWorker` class it is possible to implement digital replicas retrieving all the information from the physical objects and exposing them as standard CoAP resources for a standard IoT interoperability with external applications and consumers. Each lights is digitalized as an independent CoAP resource and exposed to the external world through CoRE Link Format, CoRE Interface and SenML. External requests are directly handled by the digital replicas by forwarding them to the devices or using the local caching system to reduce the response time and the communication load on the physical objects.

3.3. White Label Digital Twins real-time monitoring

As illustrated in Fig. 6, the third example considers a use case where multiple independent DTs are operating in the same environments using the WLDT library and its metrics layer. The developer can define its own metrics inside each worker and through the engine's configuration how they should be measured and collected using both CSV files and/or a Graphite reporter. In the described example, involved metrics are automatically sent

[8] Philips Hue Platform - https://www2.meethue.com.

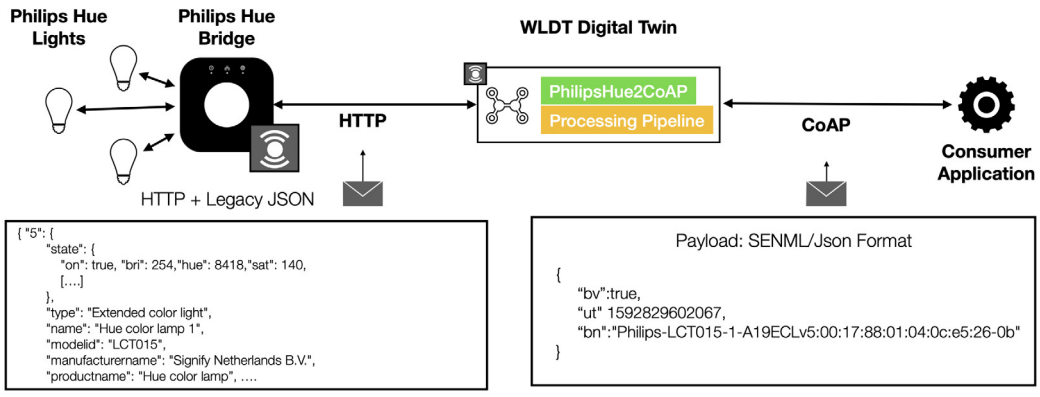

Fig. 5. Example of a WLDT instance mirroring Philips Hue lights through the standard use of CoAP and SenML.

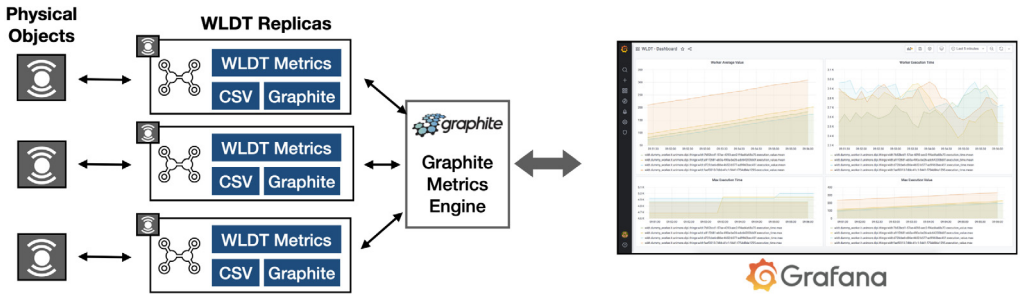

Fig. 6. Example of multiple WLDT instances with the metrics module enabled. All metrics are sent both to CSV filed and tho a Graphite Engine in order to be visualized on a Graphana reporting Dashboard.

to a Graphite collector active in the local network and visualized through the use of the Grafana[9] dashboard and reporting tool. Without any customization and additional dedicated layer, each single DT will be automatically monitored in real-time allowing to properly orchestrate the available services and detect performance degradation or anomaly situations.

4. Impact

WLDT represents a step towards the creation of independent, modular and intelligent IoT DTs. The library allows developers to easily create DTs for already deployed or new physical IoT smart objects without the need of directly operating on the device or being bound to a monolithic core and with the flexibility to adapt and customize the behavior according to the need of the target applications scenario.

Furthermore, thanks to the proposed solution, DTs can be easily designed to support a standard collaboration in terms of connection, data management and processing. The library provides an highly modular design, allowing an easy integration into new or existing business applications with the peculiar characteristic to be used both in the cloud and on the edge also through microservices and containerized deployments. WLDT overcomes the existing lack of DT models or common development approaches that are actually forcing the development of legacy solutions through monolithic and centralized layers. The real possibility to create a general purpose software agent attachable to a physical object to automatically clone it into its digital replica enables new scalable, distributed and extensible architectures for the real *autonomy* and *collaboration* among things and services.

Nevertheless, a software agent oriented vision for DTs follows also the microservice technological trend allowing applications to be orchestrated among multiple edge and cloud distributed computation facilities taking also advantage of dynamic and software controlled networking. A containerized WLDT enabled DT has the possibility to easily migrate or be cloned to one or multiple locations in order to be close to the data and the applications reducing latency and improving performance for example in 5G MEC (Multi-access Edge Computing) infrastructures [15,16].

WLDT has been publicly released with this software publication and it has been already used, in collaboration with other researchers and Universities to carry out different experimentation and research activities related in particular to IoT and edge computing. The WLDT library has been experimented within: (i) the POLIS-EYE project[10] to support and standardize IoT data acquisition from presence and traffic sensors; and (ii) the Bosch Smart Parking Pilot in Mantua[11] (Italy) to digitalize and virtualize physical parking smart objects. Both application scenarios allowed to show the importance of building an efficient, standard and flexible abstraction layer on top of physical devices in order to simplify and support application development and business logics. Furthermore, as previously introduced in Section 2.5 the WLDT has been also successfully experimented on Edge computing MEC infrastructures showing its modularity and the reduced implementation cost. The developers having worked with the library have explicitly appreciated the built-in support for IoT standard protocols, modularity, flexibility and the reduced amount of code to be written. WLDT will help to support new projects from both academia and industries related to the creation of new IoT cyber physical interaction forms and applications.

[9] Grafana - https://grafana.com/.

[10] POLIS-EYE Official Website - https://www.poliseye.it/.

[11] Bosch - https://www.bosch-press.it/pressportal/it/it/press-release-41216.html.

5. Conclusions

WLDT is a novel, powerful, modular and flexible library that can be adopted and used to create IoT DTs in multiple heterogeneous application scenarios. It can be used for automatically mirroring standard IoT smart objects (e.g., through MQTT and CoAP) or custom and legacy devices. Personalization is supported by the possibility to define custom processing pipeline to handle incoming and outgoing data, by a modular internal caching system and by the built-in support for metrics and logging.

We hope that WLDT can become a common and widespread tool for researchers and developers to design and implement their own DT-oriented solutions and services. As WLDT is an ongoing project, we hope that developers and researchers will join it to contribute to the codebase, thus speeding up its evolution and extending the range of provided features.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Work supported by: (i) POR-FESR 2014–2020 Project: POLIcy Support systEm for smart citY data governancE - POLIS-EYE. PG/2018/631990. CUP E21F18000200007 and (ii) Italian MIUR, PRIN 2017 Project “Fluidware”, N. 2017KRC7KT.

References

[1] Minerva R, Lee GM, Crespi N. Digital twin in the IoT context: A survey on technical features, scenarios, and architectural models. Proc IEEE 2020;108(10):1785–824. http://dx.doi.org/10.1109/JPROC.2020.2998530.

[2] Grieves M. Product lifecycle management: driving the next generation of lean thinking. New York: McGraw-Hill Education; 2005.

[3] Haag S, Anderl R. Digital twin – proof of concept. Manuf Lett. 2018;15. http://dx.doi.org/10.1016/j.mfglet.2018.02.006.

[4] Glaessgen E, Stargel D. The digital twin paradigm for future NASA and U.S. air force vehicles. In: 53rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics and materials conference 20th AIAA/ASME/AHS adaptive structures conference 14th AIAA; 2012. p. 1818.

[5] Tao F, Cheng J, Qi Q, Zhang M, Zhang H, Sui F. Digital twin-driven product design, manufacturing and service with big data. Int J Adv Manuf Technol 2018;94(9–12):3563–76.

[6] Botkina D, Hedlind M, Olsson B, Henser J, Lundholm T. Digital twin of a cutting tool. Procedia CIRP 2018;72:215–8. http://dx.doi.org/10.1016/j.procir.2018.03.178.

[7] Tao F, Qi Q. Make more digital twins. Nature 2019;573(7775):490–1. http://dx.doi.org/10.1038/d41586-019-02849-1.

[8] MQTT Version 3.1.1. 2014, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[9] Shelby Z, Hartke K, Bormann C. The Constrained Application Protocol (CoAP). 2014, RFC 7252, http://dx.doi.org/10.17487/RFC7252.

[10] Balalaie A, Heydarnoori A, Jamshidi P. Microservices architecture enables DevOps: Migration to a cloud-native architecture. IEEE Softw 2016;33(3):42–52. http://dx.doi.org/10.1109/MS.2016.64.

[11] Busanelli S, Cirani S, Melegari L, Picone M, Rosa M, Veltri L. A sidecar object for the optimized communication between edge and cloud in internet of things applications. Future Internet 2019;11(7). http://dx.doi.org/10.3390/fi11070145.

[12] Shelby Z. Constrained RESTful Environments (CoRE) Link Format. Tech. Rep. 6690, 2012, http://dx.doi.org/10.17487/RFC6690.

[13] Shelby Z, Koster M, Groves C, Zhu J, Silverajan B. Reusable interface definitions for constrained restful environments. Internet-Draft draft-ietf-core-interfaces-14, IETF Secretariat; 2019, https://tools.ietf.org/html/draft-ietf-core-interfaces-14.

[14] Jennings C, Shelby Z, Arkko J, Keränen A, Bormann C. Sensor Measurement Lists (SenML). 2018, RFC 8428, http://dx.doi.org/10.17487/RFC8428.

[15] Samanta A, Tang J. Dyme: Dynamic microservice scheduling in edge computing enabled IoT. IEEE Internet Things J 2020;7(7):6164–74. http://dx.doi.org/10.1109/JIOT.2020.2981958.

[16] Mach P, Becvar Z. Mobile edge computing: A survey on architecture and computation offloading. IEEE Commun Surv Tutor 2017;19(3):1628–56. http://dx.doi.org/10.1109/COMST.2017.2682318, arXiv:1702.05309.