Two-way Integration of Service-Oriented Systems-of-Systems with the Web of Things

Ivan Zyrianoff*, Lorenzo Gigli*, Federico Montori*†, Carlos Kamienski‡, Marco Di Felice*†,
* Department of Computer Science and Engineering, University of Bologna, Italy
† Advanced Research Center on Electronic Systems "Ercole De Castro", University of Bologna, Italy
‡ CMCC, Federal University of the ABC, Santo André, Brazil
Emails: {ivandimitry.ribeiro, lorenzo.gigli, federico.montori2, marco.difelice3}@unibo.it, cak@ufabc.edu.br

Abstract—**The Internet of Things (IoT) is nowadays affected by significant interoperability issues. One of the most popular countermeasures is the Web of Things (WoT), proposed recently in a consistent standardization effort. On the other hand, several IoT-oriented frameworks are already established in industrial scenarios and provide SOA-like features such as discovery and orchestration. In this paper, we study how to bridge these two worlds by proposing a tool that enables a two-way translation between a WoT ecosystem and a System-of-Systems composed of well-described Web services. We evaluate the efficiency and scalability of our solution over the Eclipse Arrowhead framework through a series of experiments that assess the scalability of our solution under realistic workloads.**

Index Terms—**IoT, WoT, SOA, Arrowhead framework, Interoperability, Performance Evaluation**

I. Introduction

The rapid growth of the Internet of Things (IoT) in the last few years is generating new and increasing demands over systems architectures, platforms, and smart applications [1]. Consequently, a vast number of novel technologies, applications, and devices emerged, creating an unprecedented fragmented and heterogeneous scenario [2]. The lack of interoperability in IoT is a well-known issue, and studies show that it is one of the main factors hindering the large-scale adoption of IoT-based systems [3].

The Web of Things (WoT) standard proposed by the W3C consortium is a promising solution to tackle the heterogeneity in the IoT and has attracted the attention of both academia and market [4]. The W3C WoT enables easy integration across IoT platforms and application domains through a descriptive approach, i.e., it aims at defining a uniform representation of the capabilities of a Thing rather than prescribing the way to implement it.

At the same time, in the context of Industry 4.0, the Eclipse Arrowhead project [5], based on Service-Oriented Architecture (SOA) concepts, has gained significant momentum. It aims to enable all of its users to work in a common and unified approach that interconnects several IoT-based local clouds, leading to high interoperability levels. Any RESTful API service can be easily added to the Arrowhead ecosystem and be part of the global Arrowhead cloud of Systems of Systems (SoS).

It is worth highlighting that the presence of multiple solutions for IoT interoperability in the market may further exacerbate the fragmentation issue since it may lead to separate data islands and vertical silos. On the other hand, several Web services and platforms do not comply with any specific standard. In fact, modifying each relevant IoT-based service to follow a given standard – such as the WoT – is a herculean task. Based on these considerations, we propose a method to seamlessly bridge Arrowhead-compliant services to a WoT ecosystem and vice-versa, opening a world of possibilities to new WoT-based services and interactions.

More in detail, the main contributions of this paper are:

- We propose and implement a middleware, namely, WoT-Arrowhead Enabler (WAE), capable of discovering and converting Arrowhead services into Web Things and vice-versa. Thus, seamlessly connecting both ecosystems.
- We propose a way to convert special services structured via OpenAPI Specification (OAS) into WoT Thing Descriptions and deploy those as fully functional Web Things.
- We validate the proposed solution through a series of performance analysis experiments that enlighten the scalability of the application when tested in a close-to-real environment under high workloads.

In the remainder of this paper, Section II presents the background and the related work. The architectural design of the proposed solution composes Section III. Section IV goes in-depth in the interactions of the WoT, the Arrowhead framework, and the WAE. The implementation details are in Section V, followed by the performance analysis of the tool in Section VI. Finally, Section VII concludes and proposes relevant future works.

II. Background and Related Work

A. Eclipse Arrowhead

Over the last years, the industrial context witnessed a steady shift from legacy monolithic systems to decoupled and service-oriented environments. Single systems are seen as atomic units of operation and interact seamlessly without a prefixed setting. This trend is one of the bases upon which Industry 4.0 has settled. The Eclipse Arrowhead Framework [5] is one of the most successful implementations of such a paradigm, bringing together loose coupling, late binding, and discovery capabilities. More in detail, Eclipse Arrowhead is built on top of the concept of “Local Clouds”, which are service-oriented architectures of Systems-of-Systems (SoS) managed by a set of Core Services [5]. Thereby, other systems are service providers or service consumers and refer to the Core Services as a central authority. Among the most critical Core Services, we mention the Service Registry, the Authorization, and the Orchestration modules.

In this paper, we deal primarily with the Service Registry (SR), that stores the services offered by each service provider in the local cloud as service records. Each service record contains the essential details for interacting with such service (*i.e.*, the endpoint and the service name) as well as additional details in case the service is annotated via a well-known standard (*e.g.*, OpenAPI or WSDL). Each service provider all its offered services independently all its offered services in the SR via its API, so that service consumers can subsequently fetch the necessary reference to the services of interest.

B. Web of Things

The recent W3C WoT architecture [4] extends already established Web Technologies to counter the inherently fragmented landscape of IoT. This approach enables interoperability through cross-domain applications and IoT Platforms. The core of the WoT architecture is the Web Thing (WT), defined as ”*physical or a virtual entity whose metadata and interfaces are described by a Thing Description (TD)*” [4]. The TD comprises metadata descriptions of the WT building blocks in a human- and machine-readable JSON-LD document. Figure 1 depicts the main WT architecture components encompassed by the TD:

Behavior: represents the overall application logic – *e.g.* the code of the handlers for the WT affordances.

Interaction Affordances: provide an abstract model of the WT interface in terms of properties (*i.e.*, the state variables of the WT), actions (*i.e.*, commands that can be invoked on the WT), and events (*i.e.*, notifications sent by the WT).

Data Schemas: describe the information model and payload structure exchanged between WTs and consumers during interactions.

Security Configurations: define the control access mechanisms to the affordances and manage the security metadata.

Protocol Bindings: map the Affordances to the network strategies (*e.g.*, the protocols) to communicate with the WT.

A run-time software named *Servient* implements the software object described by the TD. The Servient allows to host and *expose* a WT (*i.e.*, to make the TD available over a network) and to interact with a remote WT by *consuming* the TD and accessing the WT affordances. It also binds multiple protocols and data models to enable interactions with different platforms.

C. Related Work

IoT heterogeneity is a significant issue that prevents the emergence of large-scale IoT-based systems [6]. Consequently, the research community put efforts into bringing interoperability to the many facets of IoT. One of these attempts is the

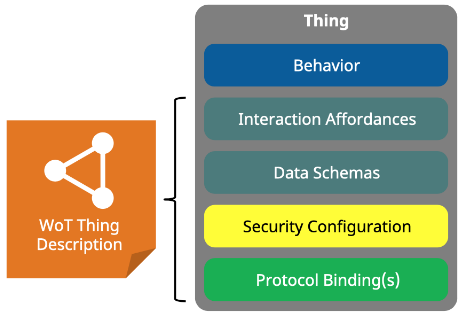

Fig. 1. W3C Web Thing architecture proposed in [4].

BIG IoT API [7] that enables interactions between different proprietary IoT platforms utilizing a novel approach for self-description and semantic annotation to adapt arbitrary IoT platforms.

The Arrowhead framework has been attracting academic attention for enabling IoT interoperability in between almost any IoT elements. Campos et al. [8] propose and implement the integration of IoT devices with the Arrowhead framework in the domain of industrial maintenance engineering. Campos-Rebelo et al. [9] propose a transparent protocol translator – *i.e.* a protocol proxy – for the Industrial IoT utilizing the Arrowhead framework to provide the operating environment for the translator. A similar approach was adopted by Moutinho et al. [10], who contribute to support the semantic compatibility verification and the generation of translators for XML messages, focusing on IoT message schemes.

Sciullo et al. [11] first introduced WAE and utilized it to perform the discovery and registration of WTs in Arrowhead SR. Our work represented substantial progress, enabling the dual integration of WoT and Arrowhead ecosystem, and made significant evolution on the architecture and interactions initially proposed. Additionally, WAE was re-implemented entirely.

III. Architectural Design

Although WoT is a potential solution to enable interoperability in IoT-based systems, not all systems can communicate directly with WTs, due to the strict WoT interfaces defined by the W3C [4]. Further, applications may not know the location of those WTs, unless implementing a discovery service. The Arrowhead SR solves both problems. It can expose the location and interface – as a REST API – of WTs. Thus, our application first automatically registers WTs as Arrowhead services; a detailed explanation of those interactions can be found at [11]. On the other hand, the Arrowhead ecosystem encompasses different services for several purposes and application domains. Hence, it will be a significant advantage for WoT-based applications to interact with those services. This feature enables seamless integration of both ecosystems, *i.e.*, from WoT to Arrowhead and vice-versa, to reduce the fragmentation issue among interoperability solutions previously mentioned.

Therefore, we propose the WAE, an application that spawns a WT proxy for each Arrowhead service. In this manner, WoT applications can interact with a variety of applications that do not follow the W3C standard architecture and interfaces [4]. In a typical Arrowhead implementation, there are numerous services registered onto the SR. Our proposal does not aim to convert all services to individual WTs since it would generate unnecessary computational resource usage. Instead, WAE monitors an array of services using a specific identifier. Whenever a new service is registered with the monitored identifier, our solution automatically detects it and attempts to convert it into a new WT. The instantiation of a WT into a service proxy requires mapping the service REST API into a TD, i.e., mapping REST endpoints to WoT affordances. We utilize the OAS of each service, when available, to convert that API documentation into a TD. The OAS defines a standard, language-agnostic description interface to RESTful APIs (the standard is also used in Swagger[1]). Any other API specification can be used for this purpose. We opted for OAS since it is widely adopted in commercial and academic applications. Clearly, the approach can be extended to other methods, provided that appropriate translators are developed.

Converting a REST API in a WoT TD is challenging since there is no exact match between HTTP methods and WoT affordances. However, we can identify some similarities between HTTP methods and WoT affordances: the *WoT property* and the *HTTP GET method* both aim to retrieve data from a specified resource; additionally, the *WoT action* and the *HTTP POST request* both submit data to a specified resource.

On top of these premises, our application reads the specification of a REST API and converts the GET endpoints into WoT *read-only* properties and POST endpoints into actions, thus, creating a minimal TD that can be used to instantiate a WT. Unfortunately, not all operations can be easily translated into TD since there are some mismatches between generic REST interfaces and the W3C WoT interface [4], such as:

- RESTful APIs support different parameters, as: path parameters (`/users/{id}`), query parameters (`/users?role=admin`), header parameters (`X-MyHeader: Value`) and cookie parameters. WoT interfaces do not support any of those parameters; consequently, endpoints that require them cannot be translated into a TD.
- The hierarchical tree structure of paths in a REST API does not have an equivalent in WoT. Hence, all endpoints are mapped as a plain affordance – *e.g.* a GET endpoint as */sensor/moisture/depth3* is translated to *sensor--moisture--depth* WoT property;
- PUT and DELETE endpoints do not have a direct correspondence to WoT affordances, therefore, such endpoints are not translated into a TD.

The translated TD is instantiated as a WT that acts as a proxy of the real service. When a property is queried or an action is evoked, the WT makes the correspondent request to the service. Then, it forwards the reply in a WoT-understandable way. The communication only involves the instantiated WT and the proxied service. Consequently, the interactions of the instantiated WTs are entirely decoupled from the WAE – they are managed as regular WTs –thereby, WAE is not a potential communication bottleneck or a single-point-of-failure for the system.

[1]`https://swagger.io/specification/`

All WTs created by WAE are automatically registered in a Thing Directory (*e.g.*, MODRON [12]), ensuring that the WTs can be discovered and managed within the WoT ecosystem. Moreover, MODRON allows users to search, list, and query the instantiated WTs in a friendly Web dashboard.

IV. Service Interaction

This section details the interactions between the software modules from an architectural standpoint to enable the two-sided integration: (*i*) the automatic discovery of new WT and their registration in Arrowhead SR, and (*ii*) the automatic discovery and conversion of Arrowhead services into WTs.

A. WT Discovery and Registration in Arrowhead

Figure 2 depicts the service interactions performed by the WAE to discover new WTs and register them in the Arrowhead SR. The WAE application periodically queries the Thing Directory, monitoring if a new WT was created. Figure 2 illustrates this 3-step process:

1) The WAE retrieves the list of all current WTs in the Thing Directory;
2) The WAE checks if each WT is registered and up-to-date in the Arrowhead SR. Hence, the WAE issues a GET request in the Arrowhead SR API with the metadata information for each WT. An empty reply means that the WT is not registered. Further, the WAE compares the TD of the WT with the one register in the Arrowhead. If any difference is detected, there is the need to update the WT in the Arrowhead.
3) The WAE registers or updates the WTs identified in the previous step.

B. Discovery and Conversion of Arrowhead Services into Web Things

Figure 3 depicts the service interactions performed by the WAE to filter and convert Arrowhead services into WTs. To this aim, the WAE periodically queries the Arrowhead SR and checks if a new service was created matching the service names it is currently monitoring. If so, the WAE instantiates a WT that acts as a proxy of that service. The detailed steps illustrated in Figure 3 are:

1) The user specifies one or more service names in a JSON format via the WAE API. Then, the WAE starts monitoring all services with such names.
2) The WAE obtains all services in Arrowhead SR and filters those that match the service names currently being monitored. Next, it checks if the filtered services have not been already deployed as WTs.

TABLE I
WAE's RESTful API endpoints

Name	Method	Description
`/arrowhead`	GET	returns metadata regarding the Arrowhead polling for the conversion of Arrowhead services to WTs
	POST	adds a new serviceName to be monitored and translated as a Web Thing
`/wotRepository`	GET	returns metadata regarding the Thing Repository polling for the discovery and registration of WTs to Arrowhead
`/management`	GET	returns the metadata in both /arrowhead and /wotRepository endpoints in a single object
`/health`	GET	returns the status of the service and the current uptime
`/api-docs`	GET	Swagger GUI interface of OAS specification
`/openapi`	GET	returns WAE OAS specification in JSON

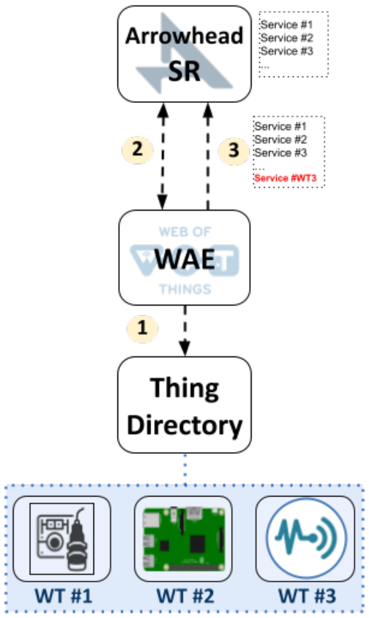

Fig. 2. Discovery and registration of WoT in the Arrowhead SR

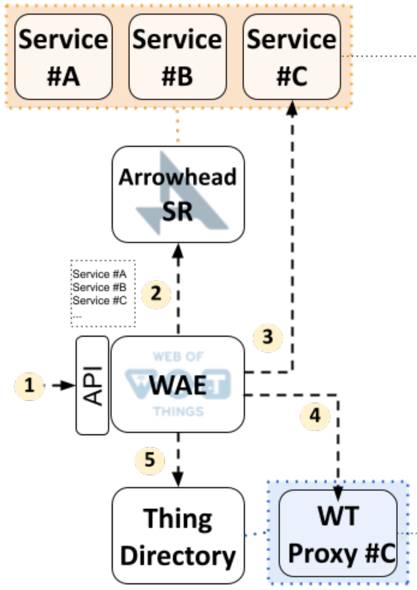

Fig. 3. Conversion of Arrowhead services into Web Things.

3) If the WAE identifies one or more applications deployed as WT, it gets the application OAS specification and converts it to a TD, applying the translation rules previously mentioned in Section III.
4) The WAE instantiates a WT that acts as a proxy of the real service with the converted TD.
5) The WAE registers the new WT in the Thing Directory.

V. Implementation

WAE is an open-source application (available at [13]) developed in JavaScript using the NodeJS v10 engine, entirely re-implemented from its previous design [11]. The *node-wot*[2] framework – the official W3C framework for the WoT – supports the creation of TDs and WTs. Our implementation follows best design practices and state-of-art technology:

- ***Modular***: the ecosystem of WoT and Arrowhead is composed of a multitude of different services independently interacting with each other, often, applications that composed those system need to be loosely coupled. Therefore, the WAE is also available as a lightweight virtualized Docker[3] container, operating without knowledge of the definitions of other components .
- ***Customizable***: the user can choose, through a configuration file, the WAE operation modes, either the standard W3C Thing Repository or MODRON [12]. Also, the user can configure polling intervals and chose if the WAE converts WT to Arrowhead services, convert Arrowhead services to WT, or both.
- ***Well Documented***: the application follows the directives of clean code [14]. The git repository provides information about the installation and use, the commits follow the conventions defined by the Google Angular developer team[4], and the WAE has a Swagger graphical interface intuitively exposing its OpenAPI documentation.
- ***Traceable Errors***: the WAE offers informative log messages using *pino*[5], with configurable log levels.
- ***Easy to Manage***: the application provides various management information through its API, such as the last time it polled data from Arrowhead SR and the Thing Directory, the number of converted WT to Arrowhead

[2]`github.com/eclipse/thingweb.node-wot`

[3]`docker.com`
[4]`gist.github.com/brianclements/841ea7bffdb01346392c`
[5]`https://github.com/pinojs/pino`

services, and vice-versa. Also, it exposes a `/health` endpoint, commonly used by third-party management tools.

Table I presents a complete list of WAE API endpoints and their descriptions.

VI. Performance Analysis

We conducted a performance analysis study with a twofold scope: (*i*) to validate WAE conversion of Arrowhead Services to WTs; (*ii*) to investigate the application's scalability in scenarios in which thousands of services need to be translated and instantiated.

For supporting the experiments, we developed a tool that generates REST APIs, detailed in Subsection VI-A. Also, we modeled and designed the experiments that are presented in Subsection VI-B. Finally, the results are included in Subsection VI-C.

A. Data Analysis: REST API Statistical Inference and OAS Generation

To run and control the experiments, we developed the OpenAPIGenerator [15], an open-source application for generating a configurable number of random OAS and exposing them in predetermined URIs. The OpenAPIGenerator also registers each generated OAS in the Arrowhead SR as a service.

A synthetic OAS needs to have size and complexity mimicking a real-world set of service APIs to approximate the experiments to a real scenario. Hence, we made statistical inferences in the public directory of REST API definitions available at APIs.guru[6] in OAS format. The APIs.guru directory filters out private and non-reliable APIs, thus consists of public, persistent, and helpful APIs – *i.e.*, that provide useful functions, not only for its owner. The dataset is composed of 2,283 different APIs, resulting in 52,203 endpoints and 77,171 methods.

Figure 4 depicts the occurrence percentage of each operation, *i.e.*, HTTP method, in the directory. We filter operations that are not IANA-valid HTTP methods [16] – specific to a particular domain or company – and methods that represent less than 0.5% of the total. The WAE translatable operations – GET and POST – correspond to 77.25% of the total. The histograms in Figure 5 depict the probability distributions of API Endpoints and GET and POST methods. Both distributions are similar and follow a long tail behavior. The x-axis in both histograms was limited to 100 to improve the graph visualization, encompassing 96.2% of the API Endpoints data and 95.6% of the GET and POST method data. From the WAE point of view, the OAS processing workload is tied to the number of GET and POST methods in the specification since those are translatable to WoT affordances. Thus, we utilize the dataset to create an empirical probability distribution of the occurrence of those operations. The OpenAPIGenerator uses such distribution to generate OAS in the experiments synthetically.

[6]`https://apis.guru/`

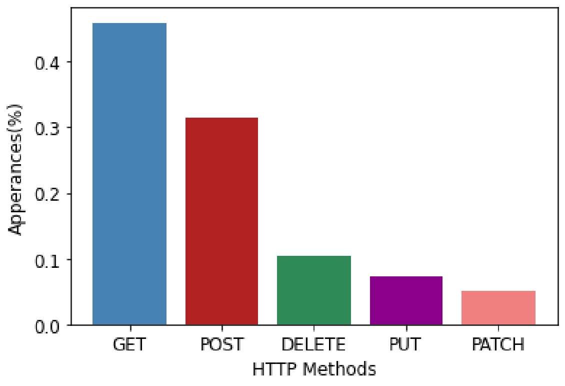

Fig. 4. Percentage of the valid HTTP Methods in the analysed dataset

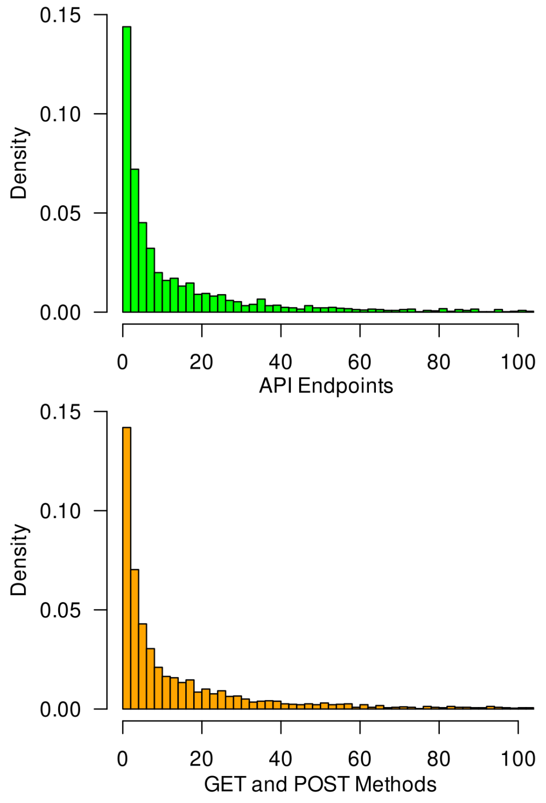

Fig. 5. Histograms of API Endpoints and GET and POST methods

B. Experimental Design

In a single server, we instantiated the three services utilized in the experiments: the WAE, the OpenAPIGenerator, and the Arrowhead SR, each virtualized as a Docker container. Preliminary to each experiment, the OpenAPIGenerator creates a set of OAS, exposes them, and registers them in the Arrowhead SR. A subgroup of the services has the same *service name*. Each experiment consists of the WAE converting the OAS of that subgroup of Arrowhead services to TDs and deploying them as WTs. We performed four experiments, varying the number of services of the subgroup, starting from a single

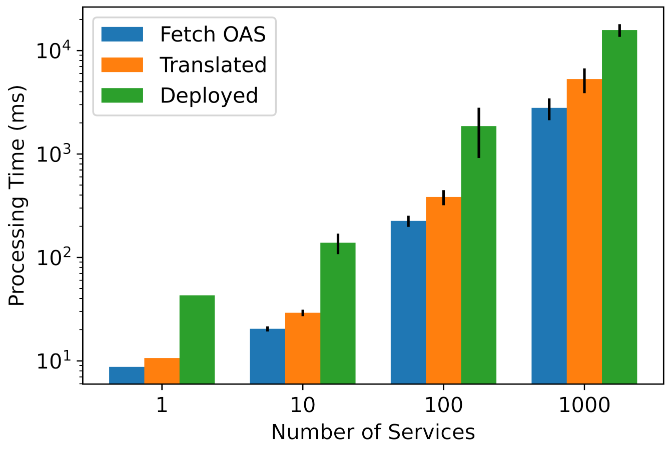

Fig. 6. Experimental Processing Times

service, then increasing by a factor of 10 for each experiment (*i.e.*, 1, 10, 100, 1,000). In every experiment, we record the time that the WAE takes to: (*i*) **fetch the OAS** once detects an Arrowhead service that needs to be converted; (*ii*) **translate** the OAS to a TD; (*iii*) **deploy** the service as a WT.

We set the initial time reference for all the recorded metrics as the last Arrowhead SR response time - the timestamp of the last poll operation performed by WAE.

C. Results

Figure 6 summarizes the key results of the performance analysis, depicting the processing time for all recorded metrics in the different evaluated workloads. The y-axis is expressed on a logarithmic scale. Overall, the experiments validated the WAE conversion of Arrowhead services to WTs and showcased that it can scale to convert a significant amount of services in a suitable time. The WAE can convert a batch of 1,000 services to WT in less than 16s, and, for a single service, it takes less than 50ms. The step taking more processing time in all workloads is the deployment of a new WT. This behavior is expected, especially for WTs that comply with the specifications of the W3C standard, thus needing to bind different protocols and support complex interactions. The fastest process is the translation, as it parses the OAS and maps it to another format via plain string manipulations. The process of fetching the OAS can potentially take more time in a real environment due to network latency issues (which we did not consider since the applications were deployed in the same server in the experiments) and workload issues since servers can take more time to respond to a request,*e.g.* due to the current use of computational resources.

VII. Conclusion

The paper proposes and implements a solution that seamlessly bridges WTs with Arrowhead services. Our application automatically converts REST API specifications to TDs and deploys them as WTs that proxy the services. Using this method, we introduce a multitude of new services to the WoT ecosystem. This paper presents a significant evolution of the application and architecture initially presented in [11].

Further, the proposed solution was evaluated regarding its scalability as we undergo experiments converting batches of services to WTs. As future works, we plan to enable the complete conversion of OAS to WT, thus, including the other HTTP methods and request parameters. Further, we will expand our performance analysis with more comprehensive experiments to measure the impacts of consuming a service through a WT proxy.

Acknowledgment

This research is funded by ECSEL, the *Electronic Components and Systems for European Leadership Joint Undertaking* under grant agreement No 826452 (Arrowhead Tools), supported by the European Union Horizon 2020 research and innovation programme and by the member states.

References

[1] L. Atzori, A. Iera, and G. Morabito, "Understanding the internet of things: definition, potentials, and societal role of a fast evolving paradigm," *Ad Hoc Networks*, vol. 56, pp. 122–140, 2017.
[2] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in internet of things: Taxonomies and open challenges," *Mobile Networks and Applications*, vol. 24, no. 3, pp. 796–809, 2019.
[3] E. Lee, Y.-D. Seo, S.-R. Oh, and Y.-G. Kim, "A survey on standards for interoperability and security in the internet of things," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1020–1047, 2021.
[4] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, and K. Kajimoto, "Web of things (wot) architecture," W3C recommendation, Apr. 2020. https://www.w3.org/TR/wot-architecture/.
[5] J. Delsing, *Iot automation: Arrowhead framework*. Crc Press, 2017.
[6] M. Noura, M. Atiquzzaman, and M. Gaedke, "Interoperability in internet of things: Taxonomies and open challenges," *Mobile Networks and Applications*, vol. 24, no. 3, pp. 796–809, 2019.
[7] A. Bröring, A. Ziller, V. Charpenay, A. S. Thuluva, D. Anicic, S. Schmid, A. Zappa, M. P. Linares, L. Mikkelsen, and C. Seidel, "The big iot api - semantically enabling iot interoperability," *IEEE Pervasive Computing*, vol. 17, no. 4, pp. 41–51, 2018.
[8] J. Campos, P. Sharma, M. Albano, L. L. Ferreira, and M. Larrañaga, "An open source framework approach to support condition monitoring and maintenance," *Applied Sciences*, vol. 10, no. 18, 2020.
[9] H. Derhamy, J. Eliasson, and J. Delsing, "Iot interoperability—on-demand and low latency transparent multiprotocol translator," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1754–1763, 2017.
[10] F. Moutinho, L. Paiva, J. Kopke, and P. Malo, "Extended semantic annotations for generating translators in the arrowhead framework," *IEEE Transactions on Industrial Informatics*, vol. 14, pp. 2760–2769, June 2018.
[11] L. Sciullo, L. Gigli, A. Trotta, and M. D. Felice, "Wot store: Managing resources and applications on the web of things," *Internet of Things*, vol. 9, p. 100164, 2020.
[12] C. Aguzzi, L. Gigli, L. Sciullo, A. Trotta, F. Zonzini, L. De Marchi, M. Di Felice, A. Marzani, and T. S. Cinotti, "Modron: A scalable and interoperable web of things platform for structural health monitoring," in *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pp. 1–7, IEEE, 2021.
[13] Ivan Dimitry Zyrianoff, "Wot-arrowhead adapter." https://github.com/UniBO-PRISMLab/wot-arrowhead-adapter, Accessed Jul. 22, 2021.
[14] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. USA: Prentice Hall PTR, 1 ed., 2008.
[15] Ivan Dimitry Zyrianoff, "Openapi generator." https://github.com/UniBO-PRISMLab/openAPIgenerator, Accessed Jul. 22, 2021.
[16] IANA, "Hypertext transfer protocol (http) method registry." https://www.iana.org/assignments/http-methods/http-methods.xhtml, Accessed Jul. 27, 2021.