Extending the Devices Profile for Web Services (DPWS) standard using a REST proxy

Ngoc Son Han, Soochang Park, Gyu Myoung Lee, Noel Crespi

Extending the Device Profile for Web Services (DPWS) standard using a REST Proxy

Son N. Han, Soochang Park, Gyu Myoung Lee, and Noel Crespi

Abstract

The OASIS standard Devices Profile for Web Services (DPWS) enables the use of Web services for service-oriented and event-driven Internet of Things (IoT) applications. DPWS has been proven to be an appropriate technology for implementing services on resource-constrained devices. However, the performance of these services has not been well investigated to realize DPWS features such as dynamic discovery and eventing mechanisms for IoT scenarios. Moreover, DPWS introduces considerable overhead due to the use of Simple Object Access Protocol (SOAP) envelopes in exchange messages. We extend the DPWS standard by using a Representational State Transfer (REST) proxy to tackle these problems, creating RESTful Web APIs to pave the way for developers to invest more in this technology.

Keywords: Internet of Things, DPWS, REST.

1. Introduction

We are witnessing the next major evolution of the Internet where millions of devices become connected to the Internet to create a new ecosystem called Internet of Things (IoT). IoT has recently gained momentum with the advancement in technology and the arrival of many commercial products that are penetrating our daily life. When it comes to IoT applications, especially for the integration into the Web, standards such as CoAP [1] and DPWS [2] are being developed to support the creation of a new generation of applications. The OASIS standard DPWS enables secure Web service capabilities on resource-constrained devices, which can be used for service-oriented and event-driven applications in the area of networked devices, the Internet of Things (IoT). DPWS has an architectural concept similar to World Wide Web Consortium (W3C) Web Service Architecture [3] but different in several ways to better fit in resource-constrained environments (limited computing power and network traffic) and event-driven scenarios. DPWS is based on Web Service Description Language[1] (WSDL) and Simple Object Access Protocol[2] (SOAP) to describe and communicate device services, but it does not require any central service registry such as Universal Description, Discovery and Integration[3] (UDDI) for service discovery. Instead, it relies on SOAP-over-UDP[4] binding and UDP multicast to dynamically discover device services. DPWS offers a publish/subscribe eventing mechanism, WS-Eventing[5], for clients to subscribe for device events, *e.g.,* a device switch is on/off or sensing when temperature reaches a predefined threshold. When an event occurs, notifications are delivered to subscribers via separate TCP connections.

These features, secure Web services, dynamic discovery, and eventing, are the main advantages of DPWS for event-driven IoT applications. Nevertheless, in fact, developers would face several problems when applying DPWS for Web-based IoT applications. The main concern is about the dynamic discovery in which the network range of UDP multicast messages is limited to the local subnet. Therefore, it is impossible to carry out this mechanism in a large network such as the Internet. With WS-Eventing, the establishment of separate TCP connections in case of delivering the same event notification to many different subscribers will generate a global mesh-like connectivity between all devices and subscribers (see Figure 1b). This requires high memory, processing power, and network traffic and thus consumes a considerable amount of energy in devices. Another issue is the overhead due to the data representation in XML format and multiple bidirectional message exchanges. It is not a problem when most DPWS devices currently communicate locally, but in a mass deployment of devices, these messages would generate heavy Internet traffic and increase the latency in device/application communication. Furthermore, W3C Web services use WSDL for service description and SOAP for service communication; the former, despite the fact that it is a W3C standard, requires much effort from developers to process poorly-structured XML data; the

[1] http://www.w3.org/TR/wsdl
[2] http://www.w3.org/TR/soap/
[3] http://uddi.org/pubs/uddi_v3.htm
[4] http://docs.oasis-open.org/ws-dd/soapoverudp/1.1/os/wsdd-soapoverudp-1.1-spec-os.html
[5] http://www.w3.org/Submission/WS-Eventing/

latter is mostly common in stateful enterprise applications, whereas recent Web applications are moving toward the core Web concepts referred as Representational State Transfer (REST) [4] by offering stateless, unified, and simple interfaces of RESTful Web APIs.

We propose the extension of DPWS standard using a REST proxy to solve these problems by providing the following features: (1) global dynamic discovery using WS-Discovery [6] in local networks; (2) proxy-based topology for publish/subscribe eventing mechanism; (3) dynamic REST addressing for DPWS devices; (4) RESTful Web APIs; and (5) WSDL caching. This REST proxy extension of DPWS unburdens Internet traffic by processing the main load in local networks. Also, the proxy can extend the dynamic discovery from locally to globally through RESTful Web APIs. Developers do not have to parse complex WSDL documents to get access to service descriptions; they can use RESTful Web APIs to control devices. Experiment results show a plain topology and substantial reductions in the overhead and latency when using our proposed proxy.

2. Web Services for the Internet of Things

The IoT is an ecosystem where all smart things or networked devices (i.e., sensors and actuators, embedded devices, electronic appliances, and digitally enhanced everyday objects) are connected using IP protocols to facilitate interoperability. It envisions an era of pervasive applications that are built on top of these networked devices. IoT scenarios require not only to have devices connected to the Internet but also seamlessly integrated into existing Internet infrastructure in which Web applications are predominant. The IoT could benefit from the Web service architecture like today's Web does by using the DPWS standard. DPWS brings W3C Web service technology into the era of networked devices by defining a set of specifications to provide a secure and effective mechanism for describing, discovering, messaging, and eventing of services for resource-constrained devices. DPWS uses WSDL to describe the device, Web Services Metadata Exchange[7] to define metadata about the device, and WS-Transfer[8] to retrieve the service description and metadata information about the device. Messaging is done by using SOAP, WS-Addressing[9], and MTOM/XOP[10] with SOAP-over-HTTP and SOAP-over-UDP bindings. It uses WS-Discovery for discovering a device (hosting service), WS-Eventing for setting up and managing subscriptions to the device events, and Web Services Policy[11] to define a policy assertion to indicate compliance of the device with DPWS.

Since its debut in 2004 by a consortium led by Microsoft, DPWS has become part of Microsoft's Windows Vista and Windows Rally (a set of technologies from Microsoft intended to simplify the setup and maintenance of wired and wireless networked devices), and has been developed in several research and development projects under the European Information Technology for European Advancement (ITEA) and Framework Programme (FP): SIRENA (02014 ITEA2), SODA (05022 ITEA2), SOCRATES (FP6), and on-going IMC-AESOP (FP7) and WOO (10028 ITEA2). Many technology giants such as ABB, SAP, Schneider Electric, Siemens, and Thales have been participating in these projects. As they have large market shares in electronics, power, automation technologies as well as enterprise solutions, their promotion of the DPWS technology promise a wide range of the future DPWS/IoT products. Schneider Electric and Odonata pioneered the implementation of DPWS leading to the early and open-source release of software stacks implementing DPWS in C and Java available at Service-Oriented Architecture for Device Website (SOA4D.org). Web Services for Devices initiative (WS4D.org) reinforces the implementation by providing and maintaining a repository to host several open-source stacks and toolkits for DPWS. In addition, many researches have been recently carried out to complete the technology. Experiment results show that DPWS is able to be implemented into (even) highly resource-constrained devices such as sensor nodes with reasonable ROM footprints [5]. Other technical issues of DPWS have also been explored such as encoding and compression [6], the integration with IPv6 infrastructure and 6LoWPAN [7, 8], the scalability of service deployment [9], and the security in the latest release of WS4D DPWS stacks.

[6] http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01
[7] http://www.w3.org/TR/ws-metadata-exchange/
[8] http://www.w3.org/Submission/WS-Transfer/
[9] http://www.w3.org/Submission/ws-addressing/
[10] http://www.w3.org/TR/soap12-mtom/
[11] http://www.w3.org/Submission/WS-Policy/

DPWS thus far has been widely used in automation industry, home entertainment, and automotive systems [10] and also applicable for enterprise integrations [11]. It satisfies many requirements for IoT applications such as resource-constrained, event-driven, and dynamic discovery; In the meantime, it can maintain the integration with the Internet and enterprises infrastructures. In addition, the strong support from the community is another reason to make it a promising technology for the future IoT. However, IoT systems containing a huge number of devices, in contrast to small numbers in industrial and home applications, cause some features of DPWS such as dynamic discovery and publish/subscribe eventing impossible in a global and mass deployment of devices. It is therefore necessary to extend DPWS to fit to IoT scenarios with several problems need to be resolved before DPWS can successfully arrive in the IoT domain. In the following sections, we are going to analyze DPWS problems with IoT and propose the extension of DPWS standard by using a REST proxy.

3. Use case

In the new ecosystem of networked devices, many IoT platforms are provided to build a new generation of Web-based applications aggregating these services. Peter, an IoT user, chooses a DPWS platform for his Web-based home automation system. He would like to make a module for controlling a newly-purchased DPWS heater. The heater is equipped with a temperature sensor, a switch, memory, a processor, and networking media, and is implemented with a hosted *Heater* service. *Heater* service consists of seven operations: (1) check the heater status (*GetStatus*), (2) switch the heater on/off (*SetStatus*), (3) get room temperature (*GetTemperature*), (4) adjust the heater temperature (*SetTemperature*), (5) add (*AddRule*), (6) remove (*RemoveRule*), and (7) get (*GetRules*) available policy rules for defining automatic operation of the heater.

Peter connects the heater to the network and tries to control it from his IoT application. We will follow Peter's development process to understand what challenges he can encounter when developing, deploying, and consuming the device from his IoT application and how the extended DPWS helps him to solve these problems. This use case illustrates a common case in several consumer applications when a new device joins the network.

4. REST Proxy Design

4.1. Global Dynamic Discovery

When an application tries to locate a device in a network, it sends a UDP multicast message (using the SOAP-over-UDP binding) carrying a SOAP envelope that contains a WS-Discovery *Probe* message with search criteria, *e.g.*, the name of the device. All the devices in the network (local subnet) that match the search criteria will respond with a unicast WS-Discovery *Probe Match* message (also using the SOAP-over-UDP binding). In our use case, it is the heater that sends *Probe Match* message containing network information. The application can send a series of other messages by the same means to invoke a required operation. At this point, Peter would realize that it is impossible for his IoT application to dynamically discover the heater because of the network range limit to local subnet of multicast messages.

If a REST proxy is applied, it allows the application to suppress multicast discovery messages and send a unicast request to the proxy instead. Then, the proxy can representatively send *Probe* and receive *Probe Match* messages to and from the network while the behavior of devices remains unmodified; they still answer to *Probe* message arriving via multicast. In networks with many changes in the device structure, where many *Probe* messages appear, the proxy can significantly unburden the Internet traffic.

REST proxy provides two RESTful Web APIs to handle the discovery as follows:

1) PUT http://123.456.789.1:8080/discovery: update the discovery with search criteria (*e.g.*, name of device)
2) GET http://123.456.789.1:8080/discovery: get the list of discovered devices

(123.456.789.1 is the IP address, 8080 is the port number of the proxy)

We also propose a repository in the proxy to maintain the list of active devices. The repository is updated when devices join and leave the network. In addition, the proxy performs a routine to periodically check the consistency of the repository, says every 30 minutes. For a proxy with 100 devices, the size of the repository is about 600 kb, so it is feasible for unconstrained machines used to host a proxy.

4.2. Publish/subscribe Eventing

To receive event notifications, Peter can subscribe his application directly to the heater by sending a SOAP envelope containing a WS-Eventing *Subscribe* message (using the SOAP-over-HTTP binding). The heater responds by sending a WS-Eventing *SubscribeResponse* message via the HTTP response channel. When an event occurs, the heater establishes a new TCP connection and sends an event notification to the subscriber. Therefore, in scenarios with many subscribers, it generates high level of traffic, requiring high resources, and causing devices to consume more energy. However, this publish/subscribe mechanism can be done through REST proxy to reduce the overhead of SOAP message exchanges and resource consumption, replacing global mesh-like connectivity by proxy-based topology (see Figure 1). One RESTful Web API is dedicated for event subscription; instead of sending a WS-Eventing *Subscribe* message, the application sends an HTTP POST request to the subscription resource as follows:

- POST http://123.456.789.1:8080/heater/event (parameter: application endpoint): subscribe to an event

The proxy, on behalf of applications, receives the event notification from the device and then disseminates these messages to the applications.

4.3. Dynamic REST Addressing

DPWS uses WS-Addressing to assign a unique identification for each device (endpoint address), independent from transport specific address. This unique identification is used with a series of message exchanges *Probe/ProbeMatch*, *Resolve/ResolveMatch* to get a transport address and then another series of messages are sent back and forth to invoke an operation. This process creates the overhead on the Internet. We define a mapping between a pair of DPWS endpoint/transport addresses and a single proxy URI, and thus replace several SOAP messages by simpler HTTP request/response messages. The mapping is carried out dynamically when a device is discovered. For example, in our use case of the DPWS heater:

Endpoint address: urn:uuid:800fa0d0-f5c0-11e2-80de-911c7defef4c
Transport address: http://123.456.789.10:4567/Heater
mapped to
URI: http://123.456.789.1:8080/Heater

The mapping is unique for each device service, and data are stored in the device repository of the proxy. The repository is also updated when there is a change in device status and/or periodically when the proxy runs its routine to check all the active devices.

4.4. RESTful Web APIs

As it is based on the above dynamic REST addressing mechanism, our REST proxy can generate a set of RESTful Web APIs associated with each device. It means that, instead of sending several SOAP-over-HTTP binding messages involving strict and large data formats, Peter can take advantage of the simple, familiar Web interfaces. The APIs consist of functions for discovery, subscription and service calls in REST architectural style. In order to generate these RESTful Web APIs from DPWS operations, we propose a design constraint on DPWS devices' implementation. It is based on the fact that most device services provide simple operations compared to normal Web services with complex input/output data structure. Our proposed constraint follows a simplified CRUD model ("create", "read", "update", "delete") to map between these services

and HTTP methods: *DPWS Operation Prefix → CRUD Action → HTTP Method*. Specifically, four CRUD actions are applied to map DPWS operations to HTTP methods as follows:

Get_ → READ → GET
Set_ → UPDATE → PUT
Add_ → CREATE → POST
Remove_ → DELETE → DELETE

Table 1 shows a list of RESTful Web APIs provided by a REST proxy for the heater device mapping with DPWS operations. Listing 1 is an example of request and response messages to get and return the status of the heater by using the proxy API *GET http://123.456.789.1:8080/heater*.

```
GET /heater HTTP/1.1
Host: 123.456.789.1:8080
Accept: text/html
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html
Transfer-Encoding: chunked
```

Listing 1. Request and response messages for obtaining the status of the heater.

4.5. WSDL Caching

When an application knows a device *hosted service* (representing device functionalities) endpoint address, it can ask that service for its interface description by sending a *GetMetadata Service* message. The service may respond with a *GetMetadata Service Response* message including a WSDL document. The WSDL document describes the supported operations and the data structures used in the device service. Some DPWS implementations (such as WS4D JMEDS[12]) provide a cache repository to store the WSDL document at runtime. After the application retrieves the WSDL file for the first time, the file can be cached for local usage in the subsequent occurrences within the life cycle of the DPWS framework (start/stop). This kind of caching mechanism would significantly reduce both the latency and the message overhead. Our DPWS proxy can provide WSDL caching not only at runtime but also permanently in a local database. The cache is updated along with the routine to maintain the device repository in proxy described in the dynamic discovery section.

5. Evaluation

5.1. Experiment Setup

We set up an experiment to evaluate the latency and overhead in two different scenarios: one uses our proposed REST proxy (Figure 1) and the other uses the original DPWS (Figure 1b). In both cases, there is an IoT application communicating with a DPWS device (a heater) to carry out the tasks of invoking the device *hosted service*. To replicate a realistic deployment of the IoT application, it is deployed on a server running the Tomcat[13] application server, using public Internet connection, and locating about 30 km away from the local network of the devices. The DPWS heater is implemented with a *hosted service* providing seven operations as shown in Table 1 (DPWS operations 3 to 9). These operations use simple command line messages to indicate the effect of each operation such as *"current status: on"* and *"new status updated: off"*. A REST proxy is implemented in Java using the Jersey[14] library on Tomcat for handling the nine RESTful Web APIs of the heater as shown in the Table 1. The IoT application uses RESTful Web APIs provided by the

[12] http://ws4d.org/jmeds/
[13] http://tomcat.apache.org/
[14] http://jersey.java.net/

REST proxy (Figure 1a) and the WS4D JMEDS library (Figure 1b) to carry out four functionalities provided by the DPWS heater: get heater status, set heater status, add new rule, and delete a rule.

5.2. Features Comparison

For the original DPWS communication, we exclude the preprocessing phase to discover the device information (endpoint and transport addresses). Round-trip time (RTT) and message size are measured for invoking operations only. It should be noted that the actual time of the whole process would be higher and varies according to implementation strategies. One can choose to have a device discovered and its services invoked in real-time; one can have the information about device stored and then only send requests to invoke the device service. The real RTTs and message sizes would be always higher than the ones using our proposed REST proxy.

With our proposed design of the REST proxy, the DPWS standard is extended to have new features as shown in the Table 2 that doesn't exist in DPWS. These new features including Global Discovery, Global Addressing, and RESTful Web APIs are required to realize the technology for IoT applications. The extension in the meantime preserves the publish/subscribe eventing mechanism of DPWS even with better messaging format.

5.3. Latency and Message Overhead

Latency evaluation presents the mean RTTs (Figure 2a) for an application to send requests and receive responses to consume four operations of the heater *hosted service* by using RESTful Web APIs from the proxy (PROXY) and by original DPWS operations in two situations when WSDL is cached (WSDL) and not cached (DPWS). The use of the proxy significantly improves the latency compared to the both cases of DPWS communication with WSDL cached and not cached, about 75% and 20% respectively. In many pervasive IoT scenarios requiring high responsiveness, reasonable delay would improve system performance and the user experiences.

Figure 2b shows the message sizes of requests (REQUEST) and responses (RESPONSE) in four RESTful Web APIs (PROXY) and their counterpart DPWS operations (DPWS) to fulfill the same tasks. In DPWS operations, the messages do not include WSDL documents as we assume that developers choose to cache these documents when designing their applications (real-time processing WSDL documents generates more messages). It shows a great improvement of message overhead when applying REST proxy. Especially when we consider real deployments of applications and devices in original DPWS communication, it is inevitable to avoid almost full-mesh connectivity (Figure 1b) compared to the simple and linear increments of HTTP traffic in the REST proxy scenario (Figure 1a).

6. Conclusion

DPWS was designed to be an appropriate technology for use in event-driven IoT applications thanks to features such as eventing and dynamic discovery, which cannot be supported natively with HTTP protocol. The key of these features is their use of SOAP-over-UDP multicast and SOAP-over-HTTP binding, which are, in practice, limited in network range and introduce considerable overhead by using SOAP envelopes. We have proposed the design of the REST proxy to extend the DPWS standard to better integrate it into the IoT applications and the Web world while maintaining its advantages. The experiment results show a significant improvement in reducing the latency and overhead as well as simplifying the global topology of using RESTful Web APIs. For the future usage of our REST proxy design, it will be necessary to establish a standard in designing DPWS device services for a variety of devices and to be used in the dynamic generation of RESTful Web APIs. Also, its adoption in many other scenarios with real-time constraints or highly dynamicity, such as in military applications and disaster monitoring, should be further investigated.

Acknowledgement

This work is supported by two European ITEA2 projects: 10028 “Web of objects” (WoO) and 11020 “Social Internet of Things: Apps by and for the Crowd” (SiTAC).

References

[1] Z. Shelby, K. Hartke, and C. Bormann, “Constrained Application Protocol (CoAP),” *IETF Internet Draft*, Jun. 2013.
[2] “Devices profile for web services version 1.1,” *OASIS Standard*, Jul. 2009.
[3] “Web services architecture,” W3C, W3C Working Group Note, Feb. 2004.
[4] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.
[5] C. Lerche, N. Laum, G. Moritz, E. Zeeb, F. Golatowski, and D. Timmermann, “Implementing powerful web services for highly resource-constrained devices,” in *2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2011, pp. 332–335.
[6] G. Moritz, D. Timmermann, R. Stoll, and F. Golatowski, “Encoding and compression for the devices profile for web services,” in *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, 2010, pp. 514–519.
[7] G. Moritz, F. Golatowski, D. Timmermann, and C. Lerche, “Beyond 6LoWPAN: Web services in wireless sensor networks,” *IEEE Transactions onIndustrial Informatics*, vol. 9, no. 4, pp. 1795–1805, Nov. 2013.
[8] I. Samaras, G. Hassapis, and J. Gialelis, “A modified DPWS protocol stack for 6LoWPAN-based wireless sensor networks,” *IEEE Transactions onIndustrial Informatics*, vol. 9, no. 1, pp. 209–217, Feb. 2013.
[9] X. Yang and X. Zhi, “Dynamic deployment of embedded services for dpws-enabled devices,” in *2012 International Conference on Computing,Measurement, Control and Sensor Network (CMCSN)*, 2012, pp. 302–306.
[10] T. Cucinotta, A. Mancina, G. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo and F. Rusina, “A Real-Time Service-Oriented Architecture for Industrial Automation,” IEEE Transactions on Industrial Informatics, vol. 5, no. 3, pp. 267-277, 2009.
[11] P. Spiess, S. Karnouskos, D. Guinard, D. Savio, O. Baecker, L. Souza, and V. Trifa, “Soa-based integration of the internet of things in enterprise services,” in *IEEE International Conference on Web Services (ICWS 2009)*, 2009, pp. 968 –975.

Biographies

Son N. Han is a Ph.D. student at Institut Mines-Telecom, Telecom SudParis. His research focuses on Internet Technologies. He has a M.Sc. in Computer Science from The University of Seoul. Contact him at son.han@it-sudparis.eu.

Soochang Park is a research associate at Institut Mines-Telecom, Telecom SudParis. His research focuses on Networking. He has a Ph.D. from Chungnam National University. Contact him at soochang.park@telecom-sudparis.eu.

Gyu Myoung Lee is adjunct associate professor at Telecom SudParis and at the Korea Advanced Institute of Science and Technology (KAIST). His research focuses on future networks and services. He has a Ph.D. from KAIST. Contact him at gm.lee@it-sudparis.eu.

Noel Crespi is a professor at Institut Mines-Telecom, Telecom SudParis. His research focuses on Web-Next Generation Network (Web-NGN) convergence and SaaS. He has a Ph.D. from Paris VI University. Contact him at noel.crespi@mines-telecom.fr.

Tables and Figures

Table 1. Proxy RESTful Web APIs for the heater device

No.	RESTful Web APIs	DPWS Operations	Parameters	Functionalities
1	GET http://123.456.789.1:8080/discovery PUT http://123.456.789.1:8080/discovery	Discovery	deviceName	List of devices Search for device(s)
2	POST http://123.456.789.1:8080/heater/event	Subscription		Subscribe to an event
3	GET http://123.456.789.1:8080/heater	GetStatus()		Get heater status
4	PUT http://123.456.789.1:8080/heater	SetStatus(String)	status	Set heater status
5	GET http://123.456.789.1:8080/heater/temp	GetTemp()		Get room temperature
6	PUT http://123.456.789.1:8080/heater/temp	SetTemp(int)	temperature	Adjust heater temperature
7	POST http://123.456.789.1:8080/heater/rules	AddRule(String)	rule	Add new rule
8	GET http://123.456.789.1:8080/heater.rules	GetRules()		List of rules
9	DELETE http://123.456.789.1:8080/heater/rules/{ruleID}	RemoveRule(int)	ruleID	Delete a rule

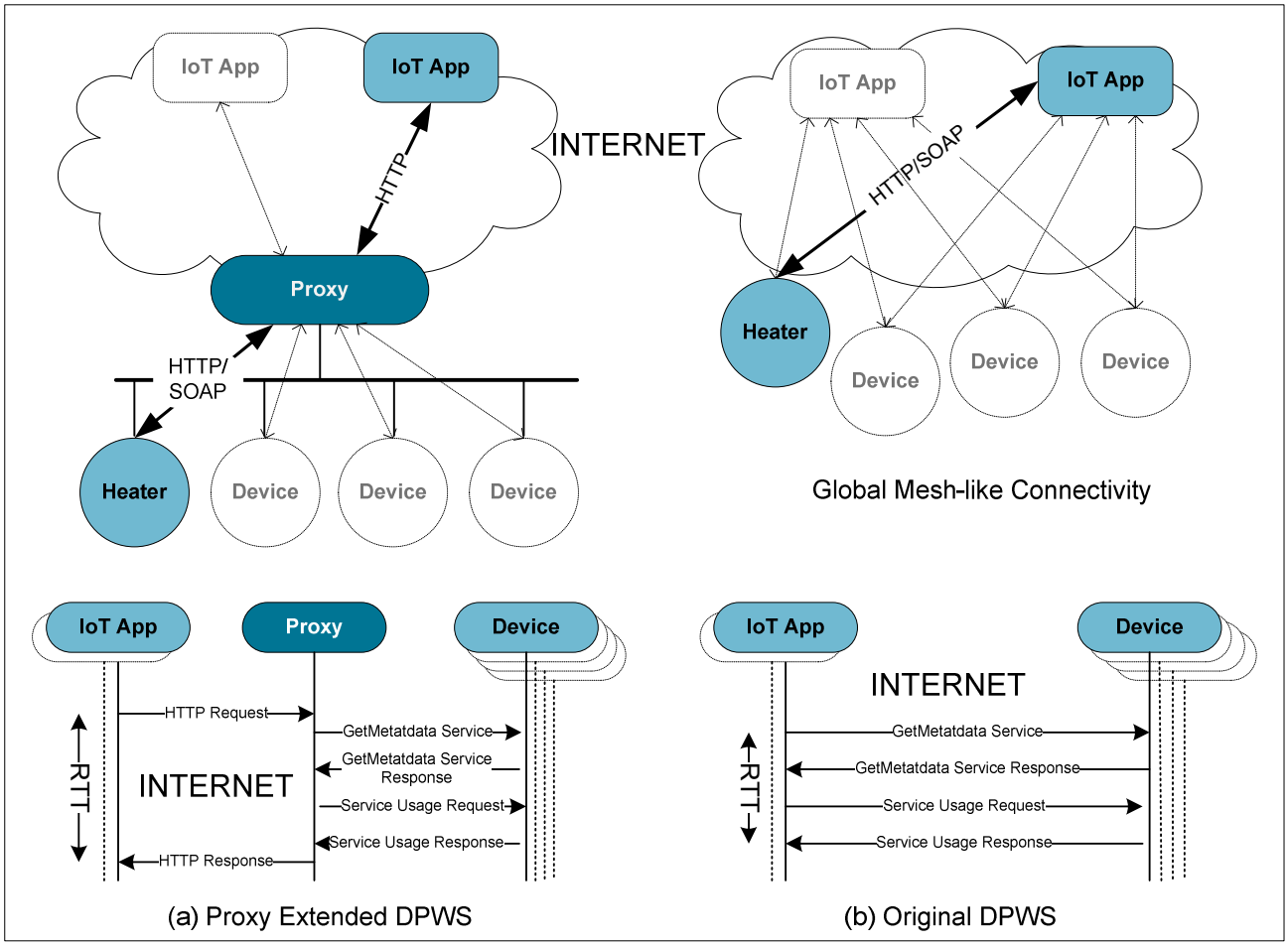
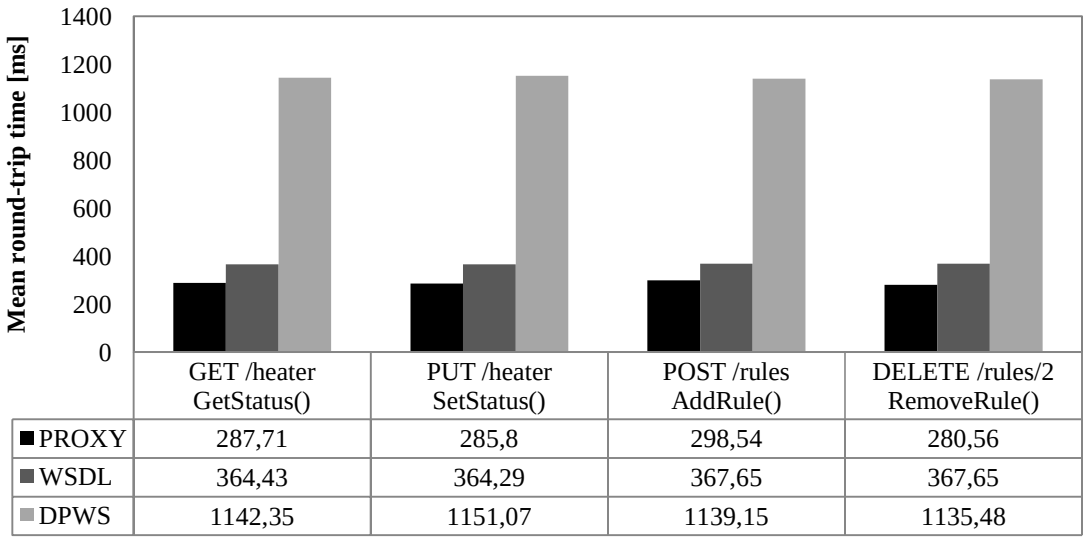
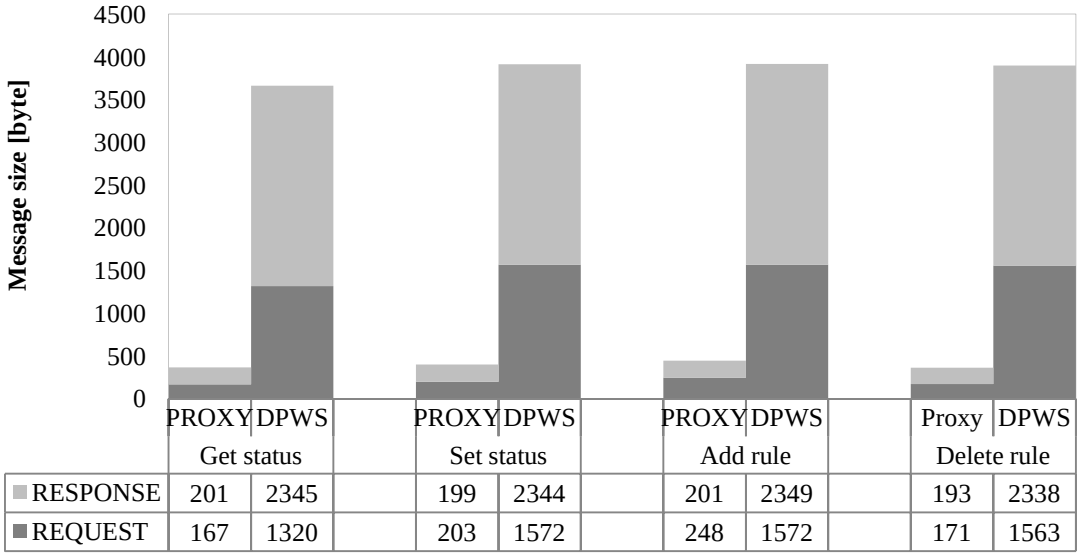


Figure 1. Experiment setup in two cases showing that original DPWS communication configures global mesh-like connectivity of HTTP/SOAP binding while our proposed scheme only configures proxy-based topology with local HTTP/SOAP binding. Consequently, the original DPWS introduces higher latency and overhead.

Table 2. Features comparison between DPWS and the proxy extended		
Features	**DPWS**	**Proxy**
Global Discovery	NO	YES
Publish-subscribe Eventing	YES	YES
Global Messaging	SOAP messages	HTTP methods
Global Topology	Mesh-like	Proxy-based
RESTful Web API	NO	YES
Configuration Module	NO	YES



(a) Mean round-trip time.



(b) Message size.

Figure 2. Mean round-trip time of 100 tests and /response message sizes when using REST proxy (Proxy) and original DPWS (DPWS) in four cases: GET /heater - GetStatus(), PUT /heater - SetStatus(), POST /rules – AddRule(), DELETE /rules/2 - RemoveRule().