Node.DPWS: Efficient Web Services for the IoT

Konstantinos Fysarakis, Damianos Mylonakis, Charalampos Manifavas, Ioannis Papaefstathiou

Abstract. *Interconnected computing systems, in various forms, will soon permeate our lives, realizing the Internet of Things (IoT) and allowing us to enjoy novel, enhanced services that promise to improve our everyday life. Nevertheless, this new reality introduces significant challenges in terms of performance, scaling, usability and interoperability. Leveraging the benefits of Service Oriented Architectures (SOAs) can help alleviate many of the issues that developers, implementers and end-users alike have to face in the context of the IoT. This work presents Node.DPWS, a novel implementation of the Devices Profile for Web Services (DPWS) based on the Node.js platform. As such, Node.DPWS is the first DPWS library being made available to Node.js developers and can be used to deploy lightweight, efficient and scalable Web Services over heterogeneous nodes, including devices with limited resources. A performance evaluation on typical embedded devices validates the benefits of Node.DPWS compared to alternative DPWS toolkits.*

Keywords: M.3 Web Services; D.3.2.h Development tools; D.2m.a Software libraries; K.6.3.a Software development; C.2.6.f Standards C.3.h; Ubiquitous computing;

1. Introduction

The advent of pervasive computing devices, interconnected to form the IoT, brings forth various challenges. End-users typically do not possess the skills to configure and setup the devices that may be found in smart environments; in large-scale deployments, individually setting up devices is not even feasible. From the perspective of implementers, there is a need for rapid development and deployment, while simultaneously tackling issues of scaling and inherent limitations in terms of resources (CPU, memory, power etc.). These are exacerbated by interoperability concerns, as smart devices are built upon diverse hardware platforms and often operate over heterogeneous networks. Moreover, while existing networking mechanisms are updated to efficiently handle the vast population of these resource-constrained devices (e.g. IETF's 6LoWPAN for IPv6 over 802.15.4), higher level, machine to machine interactions, are often required to make use of the devices' full potential. The above have shifted researchers' and developers' focus on mechanisms that guarantee interoperability, providing seamless access to the various devices and their functional elements. SOAs have evolved from this need, providing interoperable, cross-platform, cross-domain and network-agnostic access to devices and their services. This approach has already been successful in business environments (e.g. [1]), as SOAs allow stakeholders to focus on the services themselves, rather than the underlying hardware, network technologies and architectures [2].

The DPWS [3] specification enables the adoption of a SOA-based approach on embedded devices with limited resources, allowing system owners to enjoy these benefits across the heterogeneous systems of the IoT ecosystem. Various libraries already exist to assist developers in creating and deploying DPWS devices, each targeting a specific platform and/or programming language. This work presents Node.DPWS, a novel implementation of the DPWS specification which focuses on the Node.js platform (http://nodejs.org/), also referred to as Node, a JavaScript-based runtime environment designed to maximize throughput and efficiency. The associated Node.DPWS libraries are the first to leverage the benefits of both DPWS and Node, allowing the creation of high performance, scalable and lightweight DPWS devices for the heterogeneous, often resource-constrained, platforms typically found in smart environments. Moreover, the libraries are easy to use and the devices can be defined with a minimum of code, reducing the development effort.

The article begins with an introduction to DPWS, including its key characteristics and advantages. Details about Node.DPWS follow, with a brief introduction to Node.js and a description of the presented DPWS libraries and their features, also providing sample code to demonstrate the implementation approach. Alternatives to Node.DPWS are presented on the sidebar. The performance of Node.DPWS is evaluated on actual embedded platforms, including a comparison to identical devices developed using the most mature DPWS toolkit currently available. The article concludes with some pointers regarding the future of Node.DPWS.

2. The Devices Profile for Web Services (DPWS)

DPWS was introduced by a Microsoft-led consortium in 2004 and is now an OASIS open standard (version 1.1 since July 2009, [3]). The DPWS specification defines a minimal set of implementation constraints to enable Web Service messaging on resource-constrained devices. It employs similar messaging mechanisms as the Web Services Architecture (WSA, [4]), with restrictions to complexity and message size, allowing platform- and language-neutral services, similar to those offered by traditional web services. The profile's architecture includes hosting and hosted services. Thus, a multifunctional device integrated into e.g. a smart home environment, may feature various hosted services (e.g. a temperature service, a door control service, a movement-sensing service) and a single hosting service which facilitates discovery of the hosted services.

DPWS was originally conceived and introduced as a successor to Universal Plug and Play (UPnP), but due to the lack of backward compatibility such a transition has not taken place. Instead, nowadays DPWS is actively pushed by industry stakeholders for large-scale deployments (commonplace in the IoT era) [5]. Like UPnP, DPWS is natively integrated into the Windows operating system, from Windows Vista onwards.

By allowing the migration of low level information (e.g. sensing data) into higher level contexts (e.g. business operations or knowledge extraction), new types of services are made possible; these are expected to be vital for future end-user as well as enterprise deployments, in a number of industry domains. The use and benefits of DPWS have already been studied extensively by researchers in the context of various fields, such as railway systems [6], industrial automation [7], eHealth [8], smart cities [9] and smart homes [10].

The main disadvantage of DPWS, at its current state, is the existence of numerous specifications (protocols, bindings etc.) which have not been consolidated yet, but the above are promising indicators for the future of the specification and its wider adoption.

2.1 DPWS libraries *(Sidebar)*

A survey on alternative to Node.DPWS APIs for DPWS development reveals a plethora of available solutions with diverse characteristics: the libraries contained within Microsoft's .NET Micro Framework (http://www.netmf.com), the Web Services for Devices (WS4D, http://ws4d.e-technik.uni-rostock.de) toolkits which include WS4D-uDPWS, WS4D-JMEDS, WS4D-Axis2 and WS4D-gSOAP, and, finally, the Service-Oriented Architecture for Devices (SOA4D, https://forge.soa4d.org) solutions which include DPWS-Core and DPWS4J. Information for the identified DPWS implementations is aggregated in Table 1.

Table 1. Overview of DPWS Toolkits

	.NET Micro Framework	**WS4D-uDPWS**	**WS4D-JMEDS**	**WS4D-Axis2**	**WS4D-gSOAP**	**DPWS-Core**	**DPWS4J**
Language	C#	C	Java	Java (Apache Axis2)	C	C	Java
CPU/OS	ARM	PC (VM), TelosB, AVR Raven	Java SE, Java CDC/CLDC, Android	Java SE	Linux, Windows, ARM	Linux, Windows	Java SE, Java CDC
DPWS 1.0	√	-	√	√	√	√	√
DPWS 1.1	√	√	√	-	Partial	Partial	-
IPv4	√	√	√	√	√	√	√
IPv6	-	√	√	-	Partial	√	-
License	Apache 2.0	FreeBSD	EPL	Apache 2.0	GPL/LGPL	LGPL	LGPL
Active	Yes	No	Yes	No	No	Yes	No

Nevertheless, when focusing on key features such as code portability, deployment on heterogeneous platforms, support for IPv6 (necessary for IoT applications) and active development and support of the tools, the valid options are actually fewer, with WS4D-JMEDS standing out as the most attractive choice. It is the most mature work of the WS4D initiative, providing a feature-rich platform which is being constantly updated and improved. It is, therefore, used as a benchmark to assess the potential benefits of Node.DPWS.

3. About Node.js

Node.js (https://nodejs.org) is a relatively new platform, introduced in May 2009 (in version v0.12.4, as of June 2015). It is an evented server-side implementation based on Google's V8 JavaScript engine. Both Node and the V8 engine are mostly implemented in C and C++, but Node's wrapper enhances the engine's basic features by allowing server-side deployment of JavaScript programs and the use of various C libraries, system calls, binary data manipulation and request handling. The core development concept was to create the building block for lightweight and scalable servers, providing an evented, non-blocking infrastructure for highly concurrent applications.

Node handles network input/output (I/O) operations in an evented, non-blocking fashion, while file I/O operations are handled asynchronously. This differentiates Node from typical implementations which follow the threaded model where for each new connection a thread is created, having inherent scaling issues. In Node, each new connection requires only a small heap allocation. Moreover, Node's executing thread cannot be blocked; in situations where this would normally happen (e.g. waiting for data from a remote database), the thread's runtime is utilized to serve other requests. The above result in very fast applications, which also scale well, even in the case of the resource-constrained devices (i.e. with no multi-core processors or large amounts of memory) typically embedded in smart environments. More details on Node's characteristics and code samples can be found in [11].

Research indicates that while Node.js offers significant benefits in terms of I/O performance and resource utilization, it is not as good at serving large static files [12]. This does not harm its applicability in developing fast, scalable network applications, and is expected to improve as the platform matures. Moreover, it is not an issue in the context of typical IoT applications, where end devices typically transmit low level information (e.g. sensing data) and receive commands on their functional elements (e.g. turn ON/OFF). Finally, some concerns regarding the security of Node.js applications can be attributed to the lack of a stable version and can be avoided by following security-conscious programming practices [13]. The platform is not inherently insecure and can safely be used in production-grade deployments.

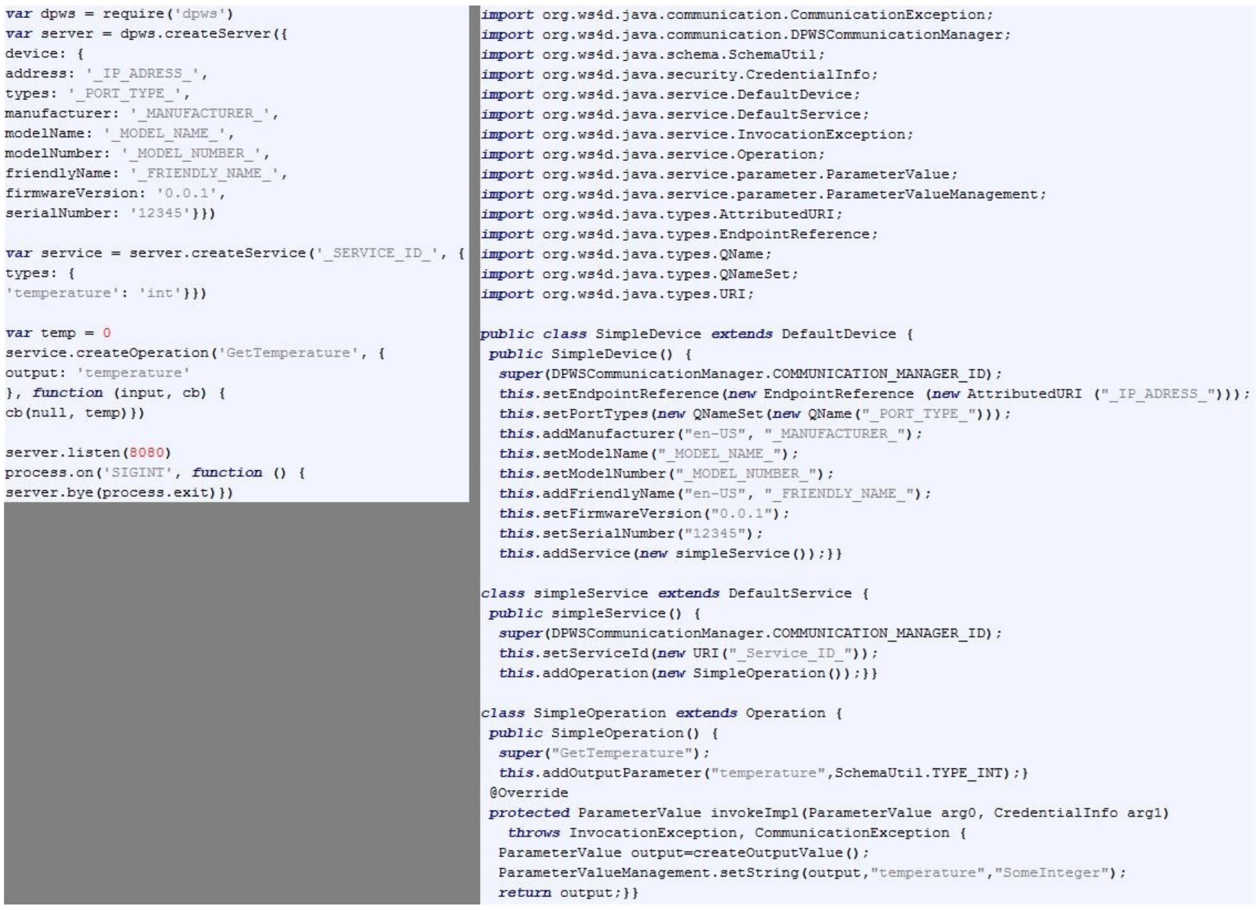
As Node.js addresses many of the issues with real-time and lightweight application communications, it has quickly gained the support of the developers' community (there is a variety of libraries already available) and of major stakeholders in the industry [14], including companies such as Google, Microsoft and Yahoo!. Popular websites such as Wikipedia, LinkedIn, eBay and Microsoft's Azure cloud platform already make use of Node.js, even though it has not reached a stable version yet; an indicator that there is a strong demand for Node's features and that its user base will continue to grow over time.

4. The Node.DPWS libraries

Node's characteristics are a good match for event-driven web services deployable on the embedded devices expected to be present in smart enterprise, industrial, domestic and other ubiquitous-computing-enhanced environments. Thus, it is an attractive solution for implementing the DPWS specification, to potentially deliver highly scalable DPWS devices, able to handle many clients concurrently, while having very low resource consumption.

Node.DPWS provides such an implementation of DPWS using Node.js, supporting both versions 1.0 (2006) and 1.1 (2009) of the specification. The developer is responsible for describing the device's attributes (e.g. manufacturer or device name), its supported services (e.g. Temperature service), operations (e.g. get current temperature) and events (e.g. overheating alerts), and the libraries will properly advertise them and match them to requests. More complex operations are also supported by the library; e.g. allowing clients to subscribe to temperature readings at set intervals or when certain events occur, adopting the WS-Eventing specification [15]. Node.DPWS also supports auto-discovery, by implementing WS-Discovery [16], a multicast discovery protocol to locate services (the main mode of discovery being a client looking for target services). Apart from discovering devices, the developed library facilitates replies to discovery requests, forwarding the developer-defined device details to requesting nodes whose queries match the device.

A key characteristic of Node.DPWS is its compact code, which is also easy to use. Operations can be defined through minimal code and the developer only has to add the device's information and its operations (along with their callback functions, adding the desired functionality to the device). To highlight this, the source code of a DPWS device compared to the same device implemented using WS4D-JMEDS is provided in Figure 1.



```
var dpws = require('dpws')
var server = dpws.createServer({
device: {
address: '_IP_ADRESS_',
types: '_PORT_TYPE_',
manufacturer: '_MANUFACTURER_',
modelName: '_MODEL_NAME_',
modelNumber: '_MODEL_NUMBER_',
friendlyName: '_FRIENDLY_NAME_',
firmwareVersion: '0.0.1',
serialNumber: '12345'}})

var service = server.createService('_SERVICE_ID_', {
types: {
'temperature': 'int'}})

var temp = 0
service.createOperation('GetTemperature', {
output: 'temperature'
}, function (input, cb) {
cb(null, temp)})

server.listen(8080)
process.on('SIGINT', function () {
server.bye(process.exit)})
```

```
import org.ws4d.java.communication.CommunicationException;
import org.ws4d.java.communication.DPWSCommunicationManager;
import org.ws4d.java.schema.SchemaUtil;
import org.ws4d.java.security.CredentialInfo;
import org.ws4d.java.service.DefaultDevice;
import org.ws4d.java.service.DefaultService;
import org.ws4d.java.service.InvocationException;
import org.ws4d.java.service.Operation;
import org.ws4d.java.service.parameter.ParameterValue;
import org.ws4d.java.service.parameter.ParameterValueManagement;
import org.ws4d.java.types.AttributedURI;
import org.ws4d.java.types.EndpointReference;
import org.ws4d.java.types.QName;
import org.ws4d.java.types.QNameSet;
import org.ws4d.java.types.URI;

public class SimpleDevice extends DefaultDevice {
 public SimpleDevice() {
  super(DPWSCommunicationManager.COMMUNICATION_MANAGER_ID);
  this.setEndpointReference(new EndpointReference (new AttributedURI ("_IP_ADRESS_")));
  this.setPortTypes(new QNameSet(new QName("_PORT_TYPE_")));
  this.addManufacturer("en-US", "_MANUFACTURER_");
  this.setModelName("_MODEL_NAME_");
  this.setModelNumber("_MODEL_NUMBER_");
  this.addFriendlyName("en-US", "_FRIENDLY_NAME_");
  this.setFirmwareVersion("0.0.1");
  this.setSerialNumber("12345");
  this.addService(new simpleService());}}

class simpleService extends DefaultService {
 public simpleService() {
  super(DPWSCommunicationManager.COMMUNICATION_MANAGER_ID);
  this.setServiceId(new URI("_Service_ID_"));
  this.addOperation(new SimpleOperation());}}

class SimpleOperation extends Operation {
 public SimpleOperation() {
  super("GetTemperature");
  this.addOutputParameter("temperature",SchemaUtil.TYPE_INT);}
 @Override
 protected ParameterValue invokeImpl(ParameterValue arg0, CredentialInfo arg1)
   throws InvocationException, CommunicationException {
  ParameterValue output=createOutputValue();
  ParameterValueManagement.setString(output,"temperature","SomeInteger");
  return output;}}
```

Figure 1. Creating a simple DPWS device using Node.DPWS (left) and the same device in WS4D-JMEDS (right).

In this example, the API's are used to expose a simple service providing temperature readings. In the case of Node.DPWS, the temperature operation is defined in a few lines of code: input/output types are specified and then a handler is provided (called whenever the operation is invoked). And while Node.js code is generally compact, this high level of abstraction would not be possible without Node.DPWS, as the developer would have to deal with the low level aspects of the specification's implementation, such as the communications (sockets etc.) and all the XML parsing and processing, in her code, choosing the most appropriate modules (e.g. server) where needed; a nontrivial task, especially for someone not familiar with the complexities of server-side JavaScript programming and the Node.js ecosystem. Further details on this can be gleaned by examining the freely available library sources, which also include samples to help familiarize developers with its functionality.

4.1. Performance evaluation

In order to assess the performance of Node.DPWS, we examined the behavior of a simple DPWS device featuring a "GetTemperature" operation which, when invoked, returned an integer value. Three versions of the device were developed: the Node.DPWS one and two versions using WS4D-JMEDS, one compiled using Java Standard Edition (SE) and the other following the Java Connected Device Configuration (CDC), part of the Java Platform Micro Edition (Java ME) designed for handheld and embedded systems.

The applications were deployed on Beaglebone (http://beagleboard.org/bone) embedded platforms (720MHz ARM Cortex-A8, 256MB RAM, Arch Linux ARM operating system), interconnected via wired Ethernet to minimize the network's impact. The test-bed also featured a client application to discover and query the DPWS devices, recording response times.

A total of 500 requests were issued from the benchmarking client (running on a desktop PC) to each of the three DPWS devices, while various aspects of their performance were being monitored. Figure 2 (left side) presents the recorded response time, i.e. the time that the client had to wait to get a response to its "GetTemperature" invocation. The Node.DPWS device responded faster than the WS4D-JMEDS-based implementations. Averaging the response times over the 500 requests reveals that the Node.DPWS device performed significantly better than both the WS4D-JMEDS devices, featuring an average response time of 24.44ms, i.e. 53% and 66.3% faster than the CDC and SE variants respectively. In an alternative representation of this performance gap, the Node.DPWS device was able to handle 40.92 requests per second, compared to 19.3 and 13.8 requests for the CDC and SE devices respectively.

Moving to the DPWS device itself, CPU and memory load were recorded during benchmarking. There were no significant differences in terms of CPU load: the average load while handling the test requests was recorded at 90.4%, 96.7% and 91.9% for Node.DPWS, CDC and SE respectively. The Node.DPWS device demonstrated better behavior in terms of the memory consumed during benchmarks. Its memory footprint was measured at 26440 bytes on average, which was 10% lower than its WS4D-JMEDS CDC counterpart (29387 bytes) and 18% lower than the SE one (32280 bytes).

To examine the behavior of the APIs on complex applications, the Policy-based Access Control (PBAC) framework presented in [9] was also evaluated, as it involves complex communication mechanisms, including automated discovery of devices, subscription and eventing. All of the framework's entities were developed using the investigated APIs, including the policy enforcement points (deployed on the Beaglebones) and the policy decision points and policy repositories (deployed on a desktop PC). Figure 2 (right side) presents the average response times for 500 requests to retrieve data via the pertinent (now access-control-protected) operation. The increased overhead of the access control mechanisms is evident, but the pattern formerly observed is maintained in this more demanding use case. The performance gap is not as evident as in the simpler scenario, as some of the delays are unrelated to the target device's implementation (e.g. the device has to wait for the policy decision point to retrieve the relevant policies and issue a decision on the access request before responding to the benchmark client), but Node.DPWS still outperforms the alternatives, followed by JMEDS CDC (response time increased by 22.51%) and JMEDS SE (increased by 32.63%).

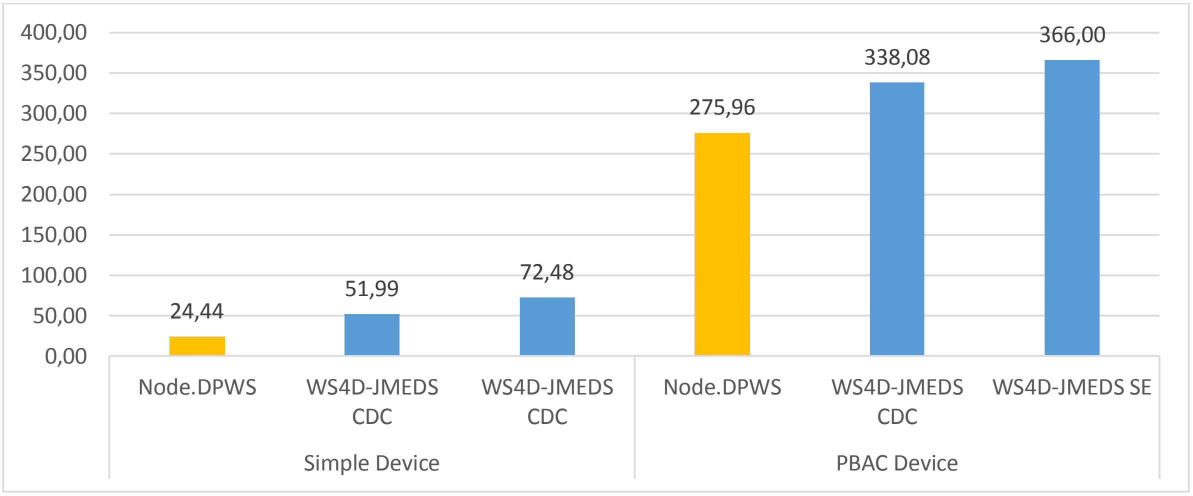


Figure 2. Average response time (in ms) for 500 requests on both the simple (left) and the policy-based access control implementation (right)

Another important aspect, especially considering battery-powered IoT devices, is energy consumption. To this end, voltage and current were monitored during benchmarks, using hardware interfaces on the Beaglebones. The monitored consumption excluded the USB Host port and expansion boards, but these were not actively used during testing, thus their overhead should be minimal compared to that of the other hardware components. The energy consumed to serve 500 requests, on each of the investigated scenarios and implementations, are presented in Figure 3. As with the other examined parameters, the energy consumption gains when developing the devices using Node.DPWS are significant.

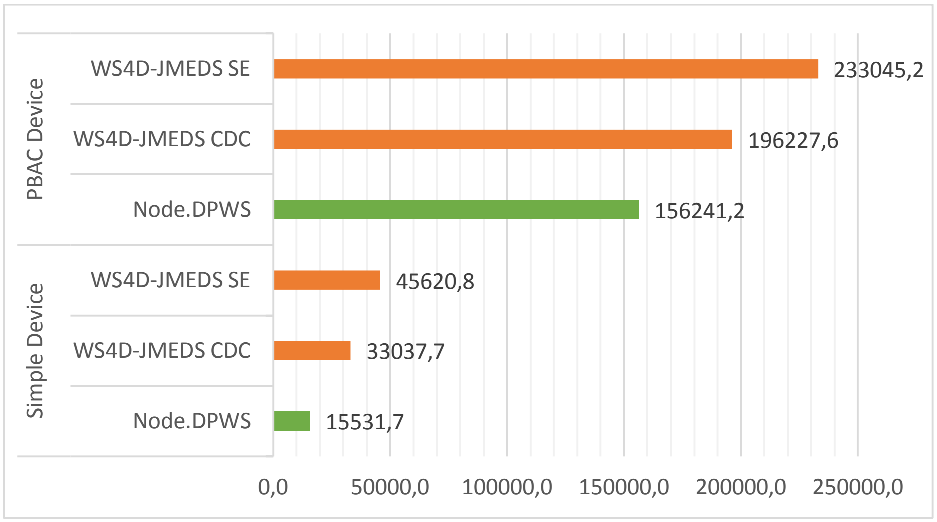


Figure 3. Energy (in mJ) required to handle 500 requests.

The advantages of adopting Node.DPWS are evident in the above results, but accurately quantifying the extent to which the noted differences can be attributed to the design of the investigated libraries or to the underlying environments (i.e. Node.js with its fast C/C++ -based engine and its event-driven, non-blocking design, set against the traditional Java environment) is nontrivial. Both libraries were designed with the intrinsic characteristics of the corresponding development environment in mind, so their behavior and features are, in great degree, interwoven with the platform they were built upon. Still, the scenarios were carefully chosen not to involve any unfair blocking requests (e.g. reading values from files), as this would exacerbate differences between the environments, simultaneously obfuscating differences between the libraries themselves.

Nevertheless, some design choices further differentiate the performance of the two libraries. The Elementtree XML parser chosen for Node.DPWS is more efficient and has a smaller memory footprint than the SAX parser used in JMEDS. Moreover, the Restify module employed in the presented library is an extremely lightweight HTTP server. Node.DPWS only binds services to a single developer-defined IP, while JMEDS automatically binds services to all available interfaces, thus consuming extra resources. Another key difference is that Node.DPWS only allows one instance of DPWS per device (still, all the desired elements can be exposed as different hosted services on this single hosting service). In contrast, JMEDS allows the deployment of multiple DPWS devices, an approach that adds complexity, as the middleware is responsible for managing the discovery, invocation and eventing mechanisms of all concurrently deployed devices, imposing extra processing overhead and delays.

5. Conclusions

In this article we demonstrated the benefits of leveraging the relatively novel Node.js platform to implement the DPWS specification in the form of Node.DPWS; an easy to use and lightweight set of libraries for creating and deploying DPWS devices on systems with limited resources. The performance assessment revealed that Node.DPWS outperforms the most attractive alternative currently available, the WS4D-JMEDS toolkit. The enhanced performance, scaling and compact code characteristics of the library show that there is significant room for improvement in the DPWS-related tools currently available. In the context of the IoT, It is, therefore, worthwhile to pursue further work on the Node.DPWS implementation, in order to enrich its libraries with more features, e.g. extending it to support other WS-* protocols (WS-Security being a good candidate).

The advent of the IoT leads to the modernization of software engineering, bringing it closer to the requirements of the ubiquitous computing world, with its plethora of heterogeneous, resource-starved end devices that will typically handle “small” (sensing) data. This intensifies the ever-present need of software practitioners to keep up to date with new tools and design approaches. Node.js is one such innovative platform and Node.DPWS aims to motivate researchers and developers alike to further explore the platform and the benefits of efficient, lightweight and scalable web services in IoT applications. Nonetheless, in software development one size does not fit all: the proposed library may be excellent for ubiquitous sensing devices but will have inherent limitations when used for backend devices handling large datasets. Thus, it is essential to examine use case scenarios at all development stages, selecting the appropriate tools for each application; an even more compelling argument for software practitioners maintaining an up-to-date and diverse skillset.

6. References

[1] S. Murer and C. Hagen, “15 Years of Service Oriented Architecture at Credit Suisse,” IEEE Softw., vol. 31, no. 6, pp. 9–15, 2014.

[2] N. Serrano, J. Hernantes, and G. Gallardo, “Service-Oriented Architecture and Legacy Systems,” IEEE Softw., vol. 31, no. 5, pp. 15–19, 2014.

[3] “Devices profile for web services, version 1.1,” 2009. [Online]. Available: http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.pdf. [Accessed: 05-Jun-2015].

[4] D. Booth et al., “Web Services Architecture,” W3C Working Group, 2004. [Online]. Available: http://www.w3.org/TR/ws-arch/. [Accessed: 05-Jun-2014].

[5] T. Nixon, “UPnP Forum and DPWS Standardization Status,” Rally Technologies, Windows Device and Storage Technologies Group, 2008. [Online]. Available: http://download.microsoft.com/download/f/0/5/f05a42ce-575b-4c60-82d6-208d3754b2d6/UPnP_DPWS_RS08.pptx. [Accessed: 05-Jun-2015].

[6] V. Venkatesh et al., “A Smart Train Using the DPWS-based Sensor Integration,” Res. J. Inf. Technol., vol. 5, no. 3, pp. 352–362, Mar. 2013.

[7] T. Cucinotta et al., “A Real-Time Service-Oriented Architecture for Industrial Automation,” IEEE Trans. Ind. Informatics, vol. 5, no. 3, pp. 267–277, Aug. 2009.

[8] D. Gregorczyk et al., "An Approach to Integrate Distributed Systems of Medical Devices in High Acuity Environments." 5th Workshop on Medical Cyber-Physical Systems. 2014.

[9] K. Fysarakis et al., “Policy-based Access Control for DPWS-enabled Ubiquitous Devices,” in Proc. 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014), pp.1-8, Barcelona, Spain, 16-19 Sept. 2014.

[10] A. Müller et al., "A secure service infrastructure for interconnecting future home networks based on DPWS and XACML." in Proc. of the 2010 ACM SIGCOMM workshop on Home networks. ACM, 2010.

[11] S. Tilkov and S. Vinoski, “Node.js: Using JavaScript to build high-performance network programs,” IEEE Internet Comput., vol. 14, pp. 80–83, 2010.

[12] I.K. Chaniotis, K.-I.D. Kyriakou, and N.D. Tselikas, “Is Node.js a viable option for building modern web applications? A performance evaluation study,” Computing, Springer, pp. 1-22, 2014.

[13] A. Ojamaa and D. Karl, “Security Assessment of Node.js Platform,” Information Systems Security, pp. 35–43, 2012.

[14] “Projects, Applications, and Companies Using Node.” [Online]. Available: https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node. [Accessed: 05-Jun-2015].

[15] D. Box et al., “Web Services Eventing (WS-Eventing),” W3C Member Submission, vol. 2009. pp. 1–34, 2006.

[16] T. Nixon et al., “Web Services Dynamic Discovery (WS- Discovery), version 1.1,” OASIS Standard Specification, 2009.

Biographies

Konstantinos Fysarakis is a PhD candidate at the Department of Electronic & Computer Engineering of the Technical University of Crete. His interests revolve around the security & dependability of embedded systems and the challenges that arise with the wider adoption of ubiquitous computing. He received an MSc in Information Security from Royal Holloway, University of London and is a member of the IEEE. Contact him at kfysarakis@isc.tuc.gr.

Damianos Mylonakis is a freelance software developer working for various industry clients. He is interested in the use of cutting-edge programming tools for the development of applications targeting resource-constrained platforms. He holds a BSc in Computer Science from the University of Crete. Contact him at i@danmilon.me.

Charalampos Manifavas is a professor at Department of Informatics Engineering of the Technological Educational Institute of Crete. His areas of interest include network and information security. He has fifteen years of experience in the public and private sectors as security engineer and consultant. He holds a PhD in Computer Science from the University of Cambridge. Contact him at harryman@ie.teicrete.gr.

Ioannis Papaefstathiou is a professor at the Department of Electronic & Computer Engineering of the Technical University of Crete. He is interested in the architecture and design of novel computer systems, concentrating on devices with highly constrained resources. He has a PhD in Computer Science from the University of Cambridge and an MSc from Harvard University. Contact him at ypg@mhl.tuc.gr.