Received August 18, 2016, accepted September 22, 2016, date of publication October 7, 2016, date of current version November 8, 2016.

Digital Object Identifier 10.1109/ACCESS.2016.2615181

Web Performance Evaluation for Internet of Things Applications

ZORAN B. BABOVIC[1], (Student Member, IEEE), JELICA PROTIC[2], AND VELJKO MILUTINOVIC[2], (Fellow, IEEE)
[1]Innovation Center, School of Electrical Engineering, University of Belgrade, 11000 Belgrade, Serbia
[2]School of Electrical Engineering, University of Belgrade, 11000 Belgrade, Serbia

Corresponding author: Z. Babovic (zbabovic@etf.bg.ac.rs)

This work was supported in part by EU FP7 ProSense Project under Project 205494 and in part by the Serbian Ministry of Education and Science under Project III44006.

ABSTRACT An area of intensive research under the umbrella of the Internet of Things (IoT) has resulted in intensive proliferation of globally deployed sensor devices that provide a basis for the development of different use-case applications working with real-time data and demanding a rich user interface. Overcoming the lack of the standard HTML platform, HTML5 specifications WebSocket and Canvas graphics strongly supported the development of rich real-time applications. Such support has been offered by browser plug-ins such as Adobe Flash and Microsoft Silverlight for years. In order to provide a deep insight into IoT Web application performance, we implemented two test applications. In the first application, we measured latencies induced by different communication protocols and message encodings, as well as graphics rendering performance, while comparing the performance of different Web platform implementations. In the second application, we compared Web performance of IoT messaging protocols such as MQTT, AMQP, XMPP, and DDS by measuring the latency of sensor data message delivery and the message throughput rate. Our tests have shown that although Adobe Flash has the best performance at the moment, HTML5 platform is also very capable of running real-time IoT Web applications, whereas Microsoft Silverlight is noticeably behind both platforms. On the other hand, MQTT is the most appropriate messaging protocol for a wide set of IoT Web applications. However, IoT application developers should be aware of certain MQTT message broker implementation shortcomings that could prevent the usage of this protocol.

INDEX TERMS Real-time systems, web and internet services, performance evaluation, sensor systems and applications.

I. INTRODUCTION

Recent research efforts conducted within the Internet of Things (IoT) vision [1]–[3] create different sets of technologies for collecting real-world observations by connecting sensors, actuators, Radio-Frequency Identification (RFID) tags, and mobile phones on the Web [4]. These enabling technologies offer development of wide-class applications from domains such as the smart environment, which includes smart homes, cities, offices, and industrial environments; transportation and logistics; healthcare; environmental monitoring and others. All of these applications combine real-time sensor data with either historical sensor measurements or even personal and social network data and sometimes have strict requirements regarding network performance, such as latency and throughput. For instance, in industrial automation, latency requirements could be very rigorous because the update frequency in closed-loop control could vary between 10ms and 100ms [5]. Many analyses show that we can expect a tremendous growth of deployed interconnected sensor devices in our environment by 2020-2021, reaching the number of around 21 billion or up to 28 billion deployed devices as forecast by Gartner [6] and Ericsson analysts [7] respectively. Accordingly, the number of IoT applications and developers is rapidly growing, and VisionMobile predicts that the number of IoT developers will reach 4.5 million by 2020 [8].

Conventionally, real-time messaging applications have been implemented as desktop-based applications, relying on the socket connection of the underlying operating system and standard graphic library. Standard HTML Web applications have proved to have a number of advantages over desktop-based applications such as better portability, easier deployment, automatic update and maintenance for all clients. However, HTML lacks in providing support for real-time applications. In contrast, Web frameworks based on browser plug-in platforms have been overcoming these limitations of HTML for years. This support was first offered by the Java Applet plug-in, which is a technology that enabled execution of Java desktop applications within a Web browser. Much higher popularity was achieved by browser plug-ins, such as Adobe Flash and Microsoft Silverlight, which provided Web-based frameworks enriched with components for graphics rendering and support for real-time messaging communication and multimedia content delivery. The result was a proliferation of multimedia content on the Web and highly demanded real-time messaging applications. Recently, the support and interest for browser plug-ins are fading out because application providers do not want to be dependent on any plug-in providers, and browsers on some mobile platforms including iOS do not support these plug-ins.

Over time, some HTTP based techniques such as Comet [9] improved support for asynchronous data delivery on the standard HTML platform, but the full development of rich real-time applications was enabled by the introduction of HTML5 technologies like WebSocket [10] and Canvas graphics element [11]. As WebSocket implementations became mature, a number of Web clients for messaging protocols have been available, both open-source and proprietary. These protocols, established as IoT application layer protocols, enable publish/subscribe interaction model between sensors data providers and consumers thus offering a scalable architecture based around message brokers.

Having recognized advances in both IoT domain and the Web, our motivation in this paper is to provide a deep insight for software developers into the Web performance of IoT applications, by analyzing all components that have an impact on overall performance. Therefore, we developed two test applications in order to measure related latencies and the throughput rate of the sensor data message delivery.

The outcomes of this work are the following:

- We provide a comprehensive overview of technologies that could be utilized during the development of IoT Web applications. We describe communication protocols that included pull-based and push-based techniques, IoT application layer messaging protocols, message encodings containing text and binary message formats, and graphics rendering on the Web.
- We conducted a detailed survey of the related work that covers real-time messaging, both for IoT domain and on the Web in general.
- We performed the measurement of real-time performance of IoT Web application executed on HTML5, Adobe Flash and Microsoft Silverlight platforms. We analyzed latencies induced by employing different communication protocols such as HTTP long polling, HTTP streaming, and both TCP socket and WebSocket, as well as different message encodings like XML, JSON, and platform-supported binary formats. In addition, we measured the performance of dynamic graphics rendering that represents data visualization elements. In another test application, we measured the latencies of sensor data propagation, and the message throughput rate achieved by IoT application layer protocols MQTT, AMQP, XMPP, and DDS.

II. TECHNOLOGY OVERVIEW

In this section, we will give an overview of typical design alternatives from the application developers' perspective, important for developing Internet of Things real-time applications on the Web. These design alternatives belong to communication protocols, message encoding formats and at the end, the entire Web platform.

A. COMMUNICATION PROTOCOLS

We could identify three interaction models of communication between data producers and data consumers in the context of Web applications. The first model is request-reply interaction, which is also referred to as *pull-based data access, or synchronous data delivery,* and is typical for a service-oriented architecture. This model assumes that clients issue requests or queries to a service provider for specific data, and the service provider replies with appropriate data. IoT application developers are able to choose one of the following approaches:

Representational State Transfer (REST) – is an architectural style in which a client issues the standard HTTP request, choosing one of the methods such as GET, POST, PUT, and DELETE, and a server responds with appropriate data. The REST architectural style denotes interaction between a client and a server based on resources that are accessed using the HTTP protocol, addressed by URI, and represented by XML, HTML, or JSON formats.

Standard Web Service—a Web service aims to provide communication interoperability among different software platforms where the interface is described by Web Services Description Language (WSDL),[1] and messages are exchanged through the Simple Object Access Protocol (SOAP),[2] which is based on the HTTP protocol and XML data format. However, the standard Web service can be used only from the browser's plug-ins, and not from the standard HTML-JavaScript based platform.

Remote Object Method Call—this method is available in various Web frameworks Application Programming Interfaces (API), and is based on the standard HTTP request. The idea is to provide a proxy object with certain methods on the client side, which represents the remote object. Invocation of a method of a proxy object is propagated to the remote object. The common approach is that the client provides a callback object/function that is called when the operation is completed. This approach is suitable if the client and the server have compatible programming platforms—if

[1] http://www.w3.org/TR/wsdl
[2] http://www.w3.org/TR/soap/

client objects can actually be mapped to server objects and vice versa.

Constrained Application Protocol (CoAP) [12] targets Machine to Machine (M2M) communication with the aim of providing request-reply interaction model like REST to constrained devices and environment. CoAP can be easily translated to more resource demanding HTTP thus enabling integration of wireless sensor networks (WSN), for example, with the Web through proxies. CoAP is based on UDP transport, and supports reliable unicast, as well as best-effort multicast connections. CoAP has a low-overhead and its messages must fit into a single IP datagram, which in the case of IEEE 802.15.4 based protocols (e.g. 6LoWPAN WSNs) produces 127 bytes long messages. As an extension of standard REST style, CoAP allows clients to issue request for observing specific server's resource, which results in receiving asynchronous notifications about the resource from the server.

Push-based data propagation, or asynchronous data delivery, is a client–server interaction model which allows the server to send data to clients immediately upon their availability, e.g. upon arrival at server. Several techniques exist:

Long-Polling—A client sends a HTTP request and waits for the data from a Web server. The server holds the connection until new data are available, a timeout event arises, or the client disconnects. Upon data arrival, the server sends data to the client, and the client initiates a new HTTP request again. Therefore, the server is able to send data to the users at any time because there is always a pending request. The benefit of the long-polling technique is the use of a standard port that is not blocked by firewalls; it is robust and works together with the proxy server. The disadvantage is the allocation of a connection per client even if the data are not transferred.

HTTP Streaming—A Web server does not terminate the response message or connection as usual, but rather keeps it open and only appends new data to the response message. This can be implemented through XHR multipart steaming and XHR iFrame streaming [13]. XHR multipart streaming utilizes the HTTP content type 'multipart,' which enables the server to send data in multiple pieces. XHR iFrame streaming allows data to be sent in multiple <script> tags. To prevent a large size of the response message at the client side, the connection must be terminated, and a new HTTP request should be issued periodically.

Socket connections—Sockets are a TCP-based technology for providing bidirectional network communication over a single connection. HTML5 specifications introduced the *WebSocket* protocol [10], which enables communication over sockets from Web browsers. An establishment of a WebSocket connection is initiated by upgrading an HTTP request. After the connection is established, there is no need for header exchange between the parties, so the control data overhead is minimal.

In addition to these basic approaches, we describe *a publish-subscribe interaction model* which essentially employs communication primitives from previously described approaches. This model enables the client (subscriber) to subscribe on data that are associated with a certain topic and to receive the data on that topic published by other clients (publishers). The broker is a server component responsible for delivering messages between the publisher and the subscriber, ensuring quality of services, persisting messages and similar functionalities. Several messaging protocols have been used as IoT application protocols.

Message Queue Telemetry Transport (MQTT) Reference [14] was designed by IBM in 1999 for lightweight M2M communication with the goal of providing a publish-subscribe messaging protocol with as minimal as possible bandwidth requirements, code footprint size, power consumption and message data overhead. Topics in MQTT have hierarchical names separated with a slash (/), for instance *wsn1/sensors/temperature/temp1*. Clients are allowed to use wildcards while subscribing to topics in order to easily match multiple topics. There are three levels of Quality of Service in MQTT. There is a variation of this protocol named MQTT for Sensor Networks (MQTT-SN) that is intended for use on embedded devices working on non-TCP/IP networks such as ZigBee.

Advanced Message Queuing Protocol (AMQP) Reference [15] is a binary, open standard protocol for high-performance messaging middleware, primarily designed for enterprise environment, but it is used in various application areas. AMQP 1.0 [16] is the current version and it is a wire-level protocol that defines message format with common data types, whereby additional meta-data could be provided for data interpretation, thus achieving interoperability between different vendors. The protocol ensures reliable communication with three-modes of message-delivery: at-most-once, at-least-once, and exactly-once. Unlike the AMQP 1.0, AMQP 0.9.1 version assumes the model in which messages are published to *exchanges,* and according to *binding* rules messages are forwarded to *queues* and further delivered to clients who are subscribed to those *queues.* Depending on the exchange type, there are four possible ways of routing messages between publishers and consumers.

Extensible Messaging and Presence Protocol (XMPP) Reference [17] is a set of technologies for real-time messaging having in its core XML streaming technology. The protocol was developed in 1999 by Jabber open-source community for instant messaging (IM) applications. The protocol specifications contain the core specification standardized by Internet Engineering Task Force (IETF) [18] and over 300 extensions published through XMPP Extension Protocols (XEPs) which cover various purposes such as publish/subscribe messaging (XEP-0060: Publish-Subscribe,[3]) sensor data exchange (XEP-0323: Internet of Things - Sensor Data,[4]) multi-user chatting etc. In XMPP, clients exchange XML messages called *stanzas*. There are three basic stanza types: *message*, *presence*, and *iq* (info/query). The *message stanza* is

[3]http://xmpp.org/extensions/xep-0060.html
[4]http://xmpp.org/extensions/xep-0323.html

TABLE 1. Comparison of IoT application layer messaging protocols.

Protocol	Initial Target	Standard	Transport	Pub/Sub Model	Message Delivery Reliability		
					Without Confirmation	With Confirmation	Exactly-once
COAP [12]	REST on Constrained Devices	IETF RFC 7252	UDP	Observing feature	+	+	-
MQTT [14]	Lightweight M2M	OASIS Standard	TCP	Hierarchical topics	+	+	+
AMQP [15]	Enterprise apps	ISO and IEC	TCP	4 Exchange types: Direct, Fan Out, Topic, Header	?	+	+
XMPP [17]	Instant Messaging	IETF RFC 6120, 6121	TCP	Node in pub/sub plug-in	-	-	-
DDS [23]	High Performance Apps	Object Management Group (OMG)	UDP, TCP	Typed topics	---------------- 23 QoS policies ---------------		

a root element and contains message payload (in the child element *body*) as well as information about sender (*from*), receiver (*to*), message *type* and *id*. In a publish/subscribe messaging model, a *node* represents a topic. The protocol has come into the focus of IoT domain [19], [20] after appearances of lightweight implementations for constrained devices, i.e. μXMPP [21] and XMPP client for mbed [22].

Data Distribution Service (DDS) Reference [23] is an open standard middleware communication protocol. DDS proposes serverless architecture for high-performance interoperable data sharing using Data-Centric Publish-Subscribe (DCPS) model. This model assumes typed interfaces by allowing DDS *participants* to define *topics* of certain data types that correspond to data types of data objects which applications want either to publish or receive. Applications use *DataWriter* of the given data type to publish data objects to certain topic over *Publishers* component, whereas *DataReader* of the given data type is used for receiving data objects over *Subscriber* component. Dynamic discovery of DDS *participants* is the matching of their publications and subscriptions based on topics of the same name and data type. Interface Definition Language (IDL) is used for defining data types. QoS can be defined at the level of Publishers/Subscribers as well as the level of DataWriters/DataReaders. DDS protocol specifies use of multicast UDP within LAN, and TCP transport for communication over WAN. However, DDS specification defines APIs in C++ and Java, and proprietary implementations are mostly available,[5] whereas open-source implementation of DDS is only provided in OpenDDS[6] project.

In Table 1 comparison data of IoT application layer messaging protocols are presented.

B. MESSAGE ENCODINGS

In client–server communication, transferred data are encoded in certain message formats producing different message sizes and requiring certain encoding/decoding times, whereas complex schemes allow the full object graph serialization or the provision of meta-data and thus enable interoperability.

[5]http://portals.omg.org/dds/dds-resources/

[6]http://opendds.org/

Generally, we can classify message formats into text-based and binary message formats. In both approaches, upon receiving the data, the client parses and converts it into the appropriate internal object structure. After that, the data are ready for further processing or visualization.

Text-based message encodings keep data in a human-readable format. The most popular examples are *Extensible Markup Language (XML)*[7] and *JavaScript Object Notation (JSON)*.[8] XML is a common format for interchanging data over the Internet. It has separated data and metadata and may refer to grammar definitions (schema), which are used for checking the validity of an XML document. JSON contains data expressed as name–value pairs in typical programming notation, but it is language independent. IoT applications that provide semantic data together with raw sensor data in order to implement intelligent services [4] use text-based serialization formats as a representation of *Resource Description Framework (RDF)*[9] data model. RDF is a Semantic Web technology aimed at expressing concepts and their relationships defined in ontologies of certain domains. Such interlinked concepts can be represented as a graph, and a basic RDF data item is a triple in the form <subject, predicate, object>, which is actually the edge between two nodes in the graph. Text-based serialization syntaxes of RDF rely on XML and JSON—i.e., RDF/XML[10] and RDF/JSON[11] specifications. Other examples include N-Triples[12] (and its more generalized format N3), which attempt to keep messages more compact by extracting a predicate that is common for more RDF triples, whereas the *Terse RDF Triple Language (Turtle)*[13]is efficient for expressing queries in SPARQL,[14] which is W3C's recommendation language for querying RDF data.

Binary message formats are designed with specific goals to reduce message size and/or efficiently encode some

[7]http://www.w3.org/XML/

[8]http://json.org/

[9]http://www.w3.org/RDF/

[10]http://www.w3.org/TR/REC-rdf-syntax/

[11]https://dvcs.w3.org/hg/rdf/raw-file/default/rdf-json/index.html

[12]http://www.w3.org/TR/n-triples/

[13]http://www.w3.org/TeamSubmission/turtle/

[14]http://www.w3.org/TR/rdf-sparql-query/

object graph structures, but they lack interoperability. Many programming languages typically have their default object serialization using binary encodings—e.g., Java object serialization,[15] but there are also more advanced approaches such as Kryo framework,[16] used in many high-performance applications. Some text-based message formats, such as XML and JSON, have their improved binary versions. For instance, *Efficient XML Interchange* (EXI) [24] format was designed to convert XML messages to binary ones and thus reduce message size and bandwidth in the resource-constrained environment which is important for IoT domain. In 2011, World Wide Web Consortium adopted EXI format as a recommendation [25]. Binary JSON formats such as BSON,[17] BJSON,[18] and UBJSON,[19], aim to be used for high-performance application and enable easy parsing and manipulation of binary messages while relying on simplicity of schema-less data organization of JSON. By avoiding text-processing, these formats reduce message size and improve message processing compared with the standard JSON.

Binary message formats are also used when application-specific communication is employed and the data are marshaled according to internal specifications that should enable easier conversion and better performance. In such situations, vendors provide their own implementation of binary encodings. Protocol Buffers (PBF) [26] is a compact binary format developed by Google and is based, among other techniques, on variably sized encoding of integers. Open-source binary message formats Colfer[20] and Protostuff,[21] which are based on ideas of PBF, further improve its speed and message size performance [27]. Adobe's Action Message Format (AMF) [28] is designed to represent object-graph containing properties as key-value pairs. Research efforts related to semantic data streams investigate involvement of compression techniques on RDF stream data to achieve both space savings and better processing time. For instance, RDSZ [29] and work by Joshi *et al.* [30] are examples of such approaches. The technique HDT [31] produces a binary message from RDF data by utilizing RDF graph structure properties by separately encoding terms and triples from a RDF graph.

C. GRAPHICS RENDERING

In the Internet of Things application domain, visualization of gathered sensor data may include drawing curves, charts, animation of some polygons that represents changes of certain parameters, or tracking the location of some objects using maps of regions or cities in the background. The HTML5 specification introduced the canvas element used for dynamic rendering of 2D graphics. This feature was present in particular browser implementations for a while but was not standardized. This incompatibility limited Web application developers in creating portable applications. For more advanced 3D and 2D graphics application development, there is the Web Graphics Library (WebGL), which is a JavaScript API running on the HTML5 canvas element, allowing the Graphics Processing Unit (GPU) to accelerate image processing and effects. Moreover, browser plug-ins offered strong support for vector and bitmap graphics.

The goal of this work is to provide IoT developers with the insight into the impact which each of the described technologies has on the performance of real-time IoT Web applications. However, before describing testing environment and architecture, we would review the previous related work that covers some of the issues we are dealing with.

III. RELATED WORK

This section contains related work addressing Internet of Things application layer protocols, performances of real-time sensor data delivery, as well as more general Web-based real-time applications with the focus on the impact of communication protocols, message encoding efficiency, and graphic rendering performance on the Web.

Several surveys have provided description and comparative analysis of IoT application layer protocols without providing real data measurements [3], [32]–[34]. All these surveys identified CoAP, MQTT, XMPP, AMQP and REST services as the most representative protocols, whereas the authors in [3] and [34] include DDS to this group. In addition, the authors in [32] described Web service related specifications and also a set of Open-Geospatial Consortium's (OGC) Sensor Web Enablement (SWE) [35] sensor data services and standards. The authors in [33] considered protocols' reliability and security issues, as well as their suitability for constrained devices. As a part of testing the feasibility of Session Initiation Protocol (SIP) as an alternative protocol for M2M, the authors in [34] performed the deep analysis of the message header structure in each surveyed protocol. Also, they dedicated special attention to suitability of JSON and Google's Protocol Buffers data formats for M2M. In [3] the authors proposed three IoT application scenarios and discussed which of the described protocols could be used efficiently to fulfill functional requirements of such applications.

Several works experimentally tested the most popular IoT application layer protocols, typically comparing two selected protocols. In [36] the authors compared MQTT and CoAP by creating a middleware component in order to perform testing. They found that MQTT has a smaller latency for smaller packet loss than CoAP, and in contrast, higher latency than CoAP for higher packet loss.

In [37] the authors tested performance of XMPP with three different communication techniques. They found out that WebSocket version outperforms HTTP polling with the 100ms polling interval and long-polling versions. In the client-server round-trip test case in a LAN environment,

[15]http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html

[16]https://github.com/EsotericSoftware/kryo

[17]http://bsonspec.org/

[18]http://bjson.org/

[19]http://ubjson.org/

[20]https://github.com/pascaldekloe/colfer

[21]http://www.protostuff.io/

WebSocket based version reached the rate of 721 round-trips per second, whereas a long-polling version performed only 18 round-trips per second and HTTP polling performed below 10 round-trips per second. By increasing the message payload size, the round-trip rate was constant for both HTTP based techniques, but for WebSocket version the rate was decreasing, to 437 round-trips for 1000 bytes long messages. The authors explained that HTTP based techniques have to create a new TCP connection per every message sent to the server, whereas WebSocket based version uses the same connection during the entire test.

The authors in [38] compared AMQP protocol with the RESTful Web service. In three experimental scenarios, they measured the average number of sent messages by user application in 1s over AQMP protocol and RESTful, while the messages are stored in the database. AMQP achieved the average rate of 211.5 msg/s, while REST reached 125.9 msg/s. When messages are not stored in the database, AMQP reached 226.6 msg/s.

Several authors tested real-time message delivery using WebSocket and HTTP based techniques. Pimental and Nickerson [39] analyzed the behavior of WebSocket, HTTP polling and long-polling protocols in an application for receiving low-volume sensor data produced by a wind sensor at a 4Hz rate. They were interested in latencies induced by each protocol measured by clients on four locations in the world—one local in Canada, and three others on different continents. Test results show that HTTP polling average latency is between 2.3 and 4.5 times higher than either WebSocket or long-polling. However, WebSocket and long-polling have similar latencies, except for longer distances (Canada-Japan), whereas WebSocket latency is significantly lower by a factor of 3.8 to 4. Similar work was presented in [40] by Ma and Sun. They compared long polling, ActiveX socket, Flash socket, and WebSocket communication by measuring a round-trip response time in real-time monitoring system for remote intelligent buildings. Their experimental setup included high-speed network conditions when both a server and clients were residing in China, and low-speed network conditions, when the server was outside China. Measurements showed that WebSocket has the shortest response time, with very small difference to the FlashSocket, whereas long polling and ActiveX socket have at least twice higher latency.

In [13] Gutwin et al. conducted a comparison of HTML5 real-time communication technologies including long-polling, HTTP streaming, and WebSocket with Java applet plug-in socket implementation, in the context of groupware applications. They measured the maximum number of sent/received 500-byte messages without any payload in 1 s. All tests were performed in three network environments: LAN, MAN and WAN. The results show that TCP socket protocol outperforms long-polling and HTTP streaming in all network settings. In addition, the browser's WebSocket implementation achieves a higher message rate than the Java applet socket in a WAN environment by a factor of four—i.e., 5113 to 1245 messages.

Comprehensive comparison of message encodings for Java Virtual Machine (JVM) with the focus on the message creation, serialization, and deserialization times, as well as message size is provided in [27]. According to different benchmarks, Google Protocol Buffers enhanced message encodings such as Colfer and Protostuff, as well as Kryo, achieved the best results in most aspects.

Finally, the last research we would like to highlight was done by Hoetzlein [41], who developed a test suite in order to answer which rich Internet application framework gives the best performance for rendering online 2D graphics, used for dynamic online data visualizations, applications, and games. In addition to other frameworks, HTML5 and Flex rendering implementations in the most popular browsers—i.e., Chrome, Internet Explorer and Firefox—were compared. The conclusion that is drawn from the test results is that graphics performance depends on which browser is used. HTML5 renders faster than Flex in Firefox, but in the other two browsers, the opposite is true: Flex renders faster than HTML5. In addition, Flex rendering in Firefox is significantly slower than in the other two browsers.

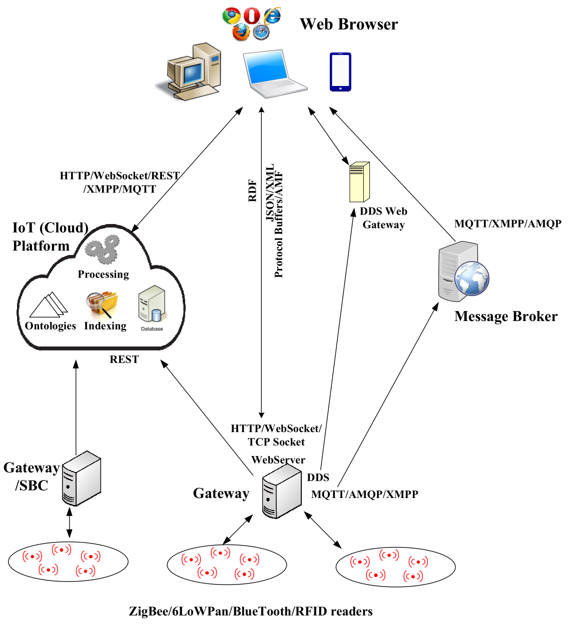


FIGURE 1. A generic architecture of internet of things web applications.

IV. GENERIC ARCHITECTURE OF IoT WEB APPLICATIONS

We created generic architecture of IoT Web applications (shown in Figure 1) with the aim of covering wide spectra of application scenarios from different domains including healthcare, smart home, environmental monitoring, smart cities, transportations and logistics, etc. In such diversity of applications, we can identify the common data collection patterns as follows [4]:

•*Acquisitional data collection* assumes data collection at defined time intervals. Clients can poll a server on equal time intervals for new data, or data can be pushed from the server to the interested clients when new data are available.

•*Event-driven data collection* is a consequence of the detection of the defined event—e.g., when measured data values satisfy some conditions. Because these events are asynchronous, users can be notified by the push-based data propagation.

•*Real-time data streaming* contains data sampled at a specific time rate. Users can be interested in obtaining raw, unprocessed data that demand significant communication bandwidth or either aggregated or processed data. Users should be informed about the data rate to avoid overload and to determine if they are able to interpret them reliably.

There are basically three types of organizations based either on a Web server, IoT platform, or a message-broker. For the scope of this work, the generic architecture is separated in three layers, the data or perception layer, the processing layer and the application layer.

The data layer contains heterogeneous devices as the sources of real-world observations. Sensor nodes are devices equipped with certain sensors and/or actuators. They have constrained resources including energy sources, computing and communication capacities. They are typically interconnected by wireless connection in either a star or a peer-to-peer topology, by employing either IEEE 812.15.4 related protocols like ZigBee and 6LoWPAN, or BlueTooth, constituting wireless sensor and actuator network (WSAN). These nodes can run lightweight software for serving data requests such as CoAP, and XMPP [21], [22] for enabling end-to-end communication and pull-based interaction with data consumers residing in WAN. However, such communication has to be routed through a sink or a gateway component. Also, some sensors could be directly attached to a single board computer (SBC) through USB interface or mobile phone [42] which could publish sensor observations to clients over WiFi, GPRS or any other wireless or wire connections. RFID readers are also source of sensor observations based on RFID tags.

The gateway component is a hardware device more capable than sensor nodes in terms of energy, processing power, and communication capabilities. It can vary from small board hardware devices running embedded Linux such as Raspberry Pie,[22] to full-size desktop computers. Depending on the chosen organization, the gateway could perform different functions such as protocol translation between IPv4/v6 networks and 6LoWPAN based WSN, device management, collection of sensor data from nodes and subsequent publishing of gathered data to either IoT platform or message-brokers, or running middleware software that hides WSN specifics. The gateway could also serve requests for sensor data from interested parties by exposing functionalities through appropriate service interfaces, as described in Section II. The pull-based model, offered through REST style, or as a standard HTTP or Web service interface, is typically used for getting the most recent sensor data, or for accepting clients' queries.

[22]https://www.raspberrypi.org/

On the other hand, push-based data propagation is enabled over TCP, WebSocket or HTTP based connections and could be useful for event-driven data provision, or for streaming sensor data to clients. These services can be hosted by employing a standard application server, or by running standalone communication libraries and platforms such as Netty[23] or Node.js.[24] This approach is suitable when in-house or custom IoT solution is deployed and when specific message encoding is used. In another organization, the gateway runs a message-broker server and delivers sensor data messages to subscribed clients residing in LAN/WAN using some of IoT messaging protocols.

The central component in the processing layer is an IoT platform which includes a wide class of solutions which perform several middleware functionalities, the sensor data processing being the major one. An IoT platform could be integrated with a cloud solution [3], [43]–[45], and expose its features usually through Platform as a Service (PaaS) model. Cloud solutions offer almost unlimited storage and computational resources and are thus able to support solutions dealing with IoT big data. Typically, an IoT platform stores received sensor data in order to combine them later with more recent data, performs data fusion with data originated from different sources, applies data mining algorithms in order to classify data or predict new data, or detects anomalies of new coming data. The authors in [44] surveyed a number of big data platforms, for instance Apache Spark[25] which is currently the leading one, as well as high-performance messaging and real-time processing tools and platforms (Apache Kafka,[26] Apache Storm,[27] and Spark Streaming,[28]) which can be deployed within a cloud in order to support such functions and build an efficient and scalable IoT platform.

Some solutions transform raw sensor data into higher abstract formats, by respecting some syntactic rules or by supplying raw data with meta-data, thus creating semantic sensor data representations in RDF. Such semantic based architectures offer intelligent services by exposing SPARQL endpoint. They also aim to enable horizontal IoT solutions by integrating WSAN using concepts defined in the ontology network. The base ontology covers the sensor data domain and it is typically Semantic Sensor Network (SSN) ontology proposed by World Wide Web Consortium (W3C). XGSN [46] is an example of such a solution, which is incorporated in the OpenIoT platform.[29] An IoT platform could also integrate a continuous query processing engine, which performs real-time processing over received data thus producing complex, aggregated or derived data to which user should be alerted if certain conditions are fulfilled. An IoT platform exposes its services through several

[23]http://netty.io/
[24]https://nodejs.org
[25]http://spark.apache.org/
[26]http://kafka.apache.org/
[27]http://storm.apache.org/
[28]http://spark.apache.org/streaming/
[29]https://github.com/mlot/openiot-platform

protocols and standards. According to [3], [43], and [45], popular IoT cloud based platforms include some of IoT application protocol message brokers. For instance, Xively[30] has support for REST and MQTT protocols, while Nimbits[31] supports REST and XMPP protocols. OGC SWE provides a set of specifications [35] which define several interfaces based on standard Web services and REST, as well as data syntax specifications in XML (Observation & Measurements – O&M). OGC SWE specifies services such as retrieving observed data (Sensor Observation Service - SOS), alerting of available sensor data (Sensor Event Service - SES), tasking for new sensor data (Sensor Planning Service - SPS), which are suitable for hosting within an IoT platform. The latency analysis of IoT (cloud) platform heavily depends on employed components and it is beyond the scope of this paper.

If sensor data processing is not necessary, a scalable architecture can leverage one or multiple message brokers which distribute sensor data messages to subscribed clients. They run either MQTT, XMPP, or AMQP protocol, while DDS based architecture does not require the message broker, although specific discover and routing components exist in order to match appropriate publisher and subscriber clients. In addition, for Web clients, particular DDS implementations require gateway component in order to convert communication to WebSocket. As described in the Section 2, different QoS could be specified regarding the message delivery reliability. Moreover, the message broker could persist messages if the messages are not delivered to all subscribed clients.

The application layer resides among Web clients, which run either on standard HTML5 platform, Adobe Flash, or Microsoft Silverlight. All platforms support HTTP clients and either WebSocket or TCP based socket connections, used for asynchronous data delivery. However, IoT messaging Web clients are in most cases implemented as JavaScript clients who directly interact with the IoT platform, the message brokers, or the gateway's Web server.

In the next section we will describe two test applications that we developed in order to evaluate Web performance of IoT applications.

V. TEST APPLICATIONS DESIGN AND DATA MODEL

In accordance with the generic architecture described in the previous section, we selected two approaches in order to test IoT application Web performance. The first test application represents an architecture in which a Web server resides on the gateway and provides sensor data to Web clients over asynchronous push-based communication, such as WebSocket, long polling, HTTP streaming etc. Web clients run on three different Web platforms: HTML5, Adobe Flash, and Microsoft Silverlight. The aim of this test is to measure the data propagation latency by analyzing the impact of communication protocol, message encoding, and graphics rendering as well as general performance of the selected Web platforms.

In the second test application, a publish/subscribe interaction model is used by employing IoT messaging protocols such as MQTT, XMPP, AMQP and DDS. We simulated an architecture in which a sensor data provider is located on the gateway component and it publishes messages to a message broker which further delivers messages to a JavaScript client running on the standard HTML platform. The goal of the second test is to compare latencies and the message throughput rate of IoT messaging protocols.

The use-case scenario was taken from a public district heating system, in which deployed sensors across the distribution network provide information about water temperature, pressure, water flow in pipes, energy consumption, and others. Similar data and usage patterns can be found in many Internet of Things applications—e.g., in health and sports applications [47]. All sensor data messages are collected at the gateway component which is a standard desktop computer. Sensor data are distributed to interested clients based on the selected approach.

Our sensor data model is influenced to some extent by the model proposed by OGC SWE specification [35] (see Figure 2-a). A sensor data message contains sensor observations from one or several sensor nodes. A sensor observation contains the following data: sensor URI, sensor type, measured value, value unit, observation time, and optionally latitude and longitude values of sensor location. Following the adopted sensor data model, we created XML and JSON messages, and we configured data structures for generation of Protocol Buffers binary message formats (see Figure 2-b,c,d).

Sensor observations are generated at a rate of 1Hz. We typically performed tests for 100 messages, which means that one test iteration lasts 3.33 minutes. There are three test parameters that can be chosen in applications: the communication protocol, encoding format, and number of monitored sensor nodes, which proportionally increases the sensor data message size.

The client application has several important blocks, which are depicted in the sequence diagram shown in Figure 3:

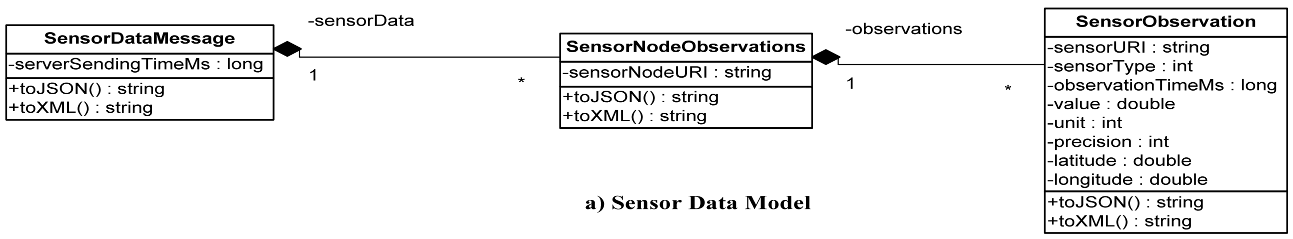
Interaction Manager is responsible for receiving data message over the selected protocol. Typically, its functionality is delegated to the appropriate framework that implements the communication protocol on the top of either WebSocket or HTTP based techniques, e.g. long-polling.

Message Decoding – Depending on the used message format, the appropriate function is called to decode the received message. Some of the used functions are built-in into the Web browser or browser's plug-in, or we used some of the available open-source frameworks.

Data Processing – If it is necessary, received sensor data would be processed by applying algorithms or by combining more recent data with the historical sensor data. If this function requires significant computation time, it should be executed in the separate thread in order to enable better responsivity of the Web client.

[30]https://xively.com/

[31]http://bsautner.github.io/com.nimbits/



```xml
<sensorData>
  <sensorNodeObservations>
      <sensorNodeURI>sensor.net/sn0</sensorNodeURI>
      <sensorObservation>
            <sensorURI>sensor.net/sn0/flowSensor2</sensorURI>
            <sensorType>5</sensorType>
            <observationTimeMs>1470852496994</observationTimeMs>
            <val>53.34045686695581</val>
            <unit>5</unit>
      </sensorObservation>
       ...
      <sensorObservation>
       ...
      </sensorObservation>
  </sensorNodeObservations>
  <serverSendingTimeMs>1470852496995</serverSendingTimeMs>
</sensorData>
```

b) XML Sensor Data Message

```json
{ "sensorData":
  [{ "sensorNodeURI":"sensor.net/sn0",
    "observations":
              [{"sensorURI":"sensor.net/sn0/flowSensor2",
                "sensorType":5,
                "observationTimeMs":1470852496994,
                "val":53.34045686695581,
                "unit":5 },
               ...
               {"sensorURI":...
              }]
  }],
  "serverSendingTimeMs":1470852496995
}
```

c) JSON Sensor Data Message

```
message SensorDataMessageProtoBuf {
            required uint64 server_sending_time = 1;
            repeated SensorNodeObservationsProtoBuf sensor_data = 2;
}
message SensorNodeObservationsProtoBuf {
            required string sensor_node_uri = 1;
            required uint64 server_sending_time = 2;
            repeated SensorObservationProtoBuf observations = 3;
}
```

```
message SensorObservationProtoBuf {
            required string sensor_uri = 1;
            required int32 sensor_type = 2;
            required uint64 observation_time_ms = 3;

            double value = 4;
            optional int32 unit = 5;
            optional int32 precision = 6;

            optional double latitude = 7;
            optional double longitude = 8;
}
```

d) Google Protocol Buffers
Data Definition

FIGURE 2. a) Sensor data model, b) XML, c) JSON message encodings, and d) Google protocol buffers data definition.

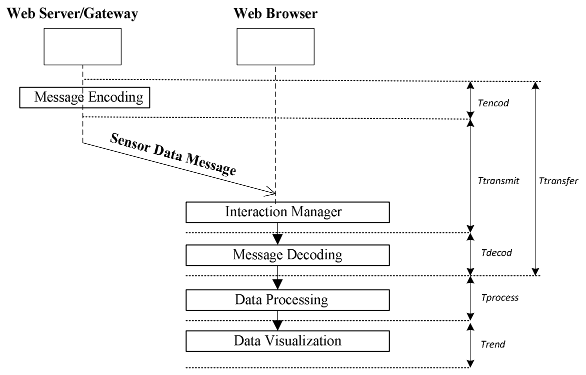


FIGURE 3. Sequence diagram with relevant times for measuring latencies.

Data Visualization – In our test, sensor data are visualized by simple vertical bars, where bar height represents the ratio of the current value to the maximum possible value.

For performance evaluation, the total time elapsed from the moment when data are available on the server/gateway to the moment when a user is able to see the relevant visual interpretation of data is used. This is expressed by Equation (1).

$$Ttotal = Ttransfer + Tprocess + Trender \quad (1)$$

$$Ttransfer = Ttransmit + Tdecod \quad (2)$$

The interpretation of Equation (1) is the following: *Ttransfer* denotes the time needed for transferring data message from the server to the client including encoding and decoding, *Tprocess* represents the time necessary for data processing at the client side, and *Trender* is the time required for rendering graphical elements on the client used for data interpretation. Equation (2) describes components of *Ttransfer*: *Ttransmit* include the time required for message encoding at the server side as well as the time spent by data in the transmission from the server to the client. We have not separately analyzed these times, because the encoding time is sometimes hard to measure on some platforms, since the encoding is performed within the used library and it is a part of the protocol stack. *Tdecod* represents the time needed for parsing the received data and its conversion to an internal data object.

The selection of a certain protocol or message format has an impact on a particular part of the total latency time *Ttotal*.

For instance, the choice of the protocol affects *Ttransmit*, whereas the choice of the message format affects all components of *Ttransfer*. In addition, different combinations of protocols and message formats may give unexpected values of *Ttransfer*. The specific Web platform implementation has a dominant impact on each particular time and hence on *Ttotal*. In our testing, we ignored *Tprocess* time after message decoding, because it heavily depends on applied algorithms, but in the real applications that time cannot be neglected in the latency analysis.

VI. IMPLEMENTATION AND EXPERIMENT SETUP

A. WEB SERVER BASED TEST APPLICATION

We have developed the same client test application for each selected Web platform, i.e. HTML5, Adobe Flash, and Microsoft Silverlight.

HTML5-JavaScript Web application was developed using the Google Web Toolkit (GWT)[32] framework. The GWT is an open source, client-side development toolkit, which enables developers to write code in the Java programming language that is cross-compiled into the JavaScript language and executed in a Web browser. The push-based communication, including WebSocket and long-polling protocol, is implemented by employing the Atmosphere 2.3.0 framework[33] because it hides incompatibilities of both browsers and Web servers. The server side code is implemented in Java and runs within Apache Tomcat Web server[34] (version 8.0.21). We included five message formats, both text based and binary, to test this platform. Two text-based formats are the standard ones, XML and JSON. Three binary formats are the following. First, the GWT serializer is a default binary format for serializing data over the GWT Remote Procedure Calls (RPC) mechanism. Second, Google Protocol Buffers (PBF) [26] was already mentioned as an efficient compact binary format developed by Google. Because there is no official PBF implementation for either JavaScript or GWT, we have used dcodeIO's PBF JavaScript implementation.[35] Third, we have included the PBF string format with the aim to reduce the overhead while streaming binary message formats using UTF-8 strings by the GWT. Hence we repacked the binary data of raw PBF into a UTF-8 string by combining two adjacent bytes into the one UTF-8 code while avoiding invalid UTF-8 code points, containing surrogate codes from D800 to DFFF.

Adobe's Flash platform[36] interprets Flash code, which provides raster graphics, multimedia streaming, animation, and interface functionality. The main programming language for developing Flash applications is ActionScript,[37] an object-oriented script language. Adobe provided an extended programming model through the Flex development environment, and this framework was donated to the Apache Software Foundation in 2011.[38] We used Apache Flex version 4.14 for our application. The server side of the Adobe Flash application is implemented in Java, runs within the Apache Tomcat Web server (version 7.0.19), and uses the Adobe Digital Enterprise Platform 4.6 framework[39] for implementation of real-time messaging with the Flex client. Our Flex application supports long-polling, HTTP streaming, and two socket protocols. The first socket protocol is the New I/O (NIO) socket protocol, which is based on a scalable socket server, and the second is the Real-Time Messaging Protocol (RTMP), which is designed for high-performance streaming of video, audio, and data over the Internet. Action Message Format (AMF) [28], Adobe's binary message format designed for high-performance real-time messaging applications, is used as a binary message format. XML is a representative of text message formats.

Microsoft Silverlight[40] is a free, cross-browser, cross-platform, and cross-device browser plug-in engine released in 2007 for the delivery of rich Internet applications. Our Silverlight application is powered by the Windows Communication Foundation (WCF)[41] service implementation of long-polling and socket protocols. These WCF services are deployed on Microsoft Internet Information Server (IIS) 7 Web server[42] and executed via the .NET 4.5.2 framework. Both the client and the server side code are implemented in C#. WCF is based on the Simple Object Access Protocol (SOAP) messaging protocol, which uses the XML encoding format. JSON message encoding was implemented via WCF's official JSON serializer. The client side was developed using Microsoft Silverlight 5 SDK.

The server machine is powered by an Intel i5-3320M CPU running at 2.60GHz, with 4GB of RAM, equipped with a Gigabit Ethernet network card. To ensure the same environment for all three platforms, we installed Web servers on the Windows 7 operating system (OS) because Windows is the only OS supported by IIS for use as a Web server for Silverlight applications. We performed testing with clients also running Windows 7 and residing in the server's LAN environment. We evaluated two clock synchronization protocols for use in our environment: Precise Time Protocol (PTP) [48] and Network Time Protocol (NTP) [49]. We eliminated PTP because we did not find an appropriate implementation for Windows. In addition, we have also eliminated NTP because the clock offset on our server fluctuated from a few milliseconds to approximately 80ms compared with the nearest NTP time servers. Finally, for server–client clock synchronization, we chose the Domain Time II tool developed by Greyware Automation Products.[43] Using this application, the

[32]http://www.gwtproject.org
[33]https://github.com/Atmosphere/atmosphere
[34]http://tomcat.apache.org/
[35]https://github.com/dcodeIO/ProtoBuf.js
[36]http://www.adobe.com/software/flash/about/
[37]http://www.adobe.com/devnet/actionscript.html
[38]http://flex.apache.org/
[39]http://help.adobe.com/en_US/dataservicesjee/4.6/Developing/index.html
[40]https://www.microsoft.com/silverlight/
[41]https://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx
[42]https://www.iis.net/
[43]https://www.greyware.com/software/domaintime/

TABLE 2. **HTML5 application test results (failed tests are not inserted in the table).**

Protocol	Long Poll.	Long Poll.	Long Poll.	Long Poll.	Long Poll.	Web Sock.	Web Sock.	Web Sock.	Web Sock.	Web Sock.	Long Poll.	Long Poll.	Long Poll.	Long Poll.	Long Poll.	Web Sock.	Web Sock.	Web Sock
Encoding	XML	JSON	GWT -SER	PBF	PBF-STR	XML	JSON	GWT -SER	PBF	PBF-STR	XML	JSON	GWT -SER	PBF	PBF-STR	JSON	GWT -SER	PBF-STR
Nodes	1	1	1	1	1	1	1	1	1	1	5	5	5	5	5	5	5	5
Payload Size [chars]	2322	1478	1153	1966	272	2322	1478	1153	1966	272	11281	7163	4485	9578	1343	7163	4485	1343
Ttotal [ms]	8.51	**6.05**	6.52	8.32	7.28	6.76	**5.74**	6.09	7.78	6.55	13.81	**7.51**	9.26	16.22	9.91	**7.22**	7.83	9.69
Ttransfer [ms]	7.98	**5.65**	6.13	7.89	7.05	6.33	**5.33**	5.75	7.42	6.07	13.36	**7.17**	8.90	15.68	9.39	**6.89**	7.45	9.34
Ttransmit [ms]	6.25	**5.54**	5.51	5.68	6.04	4.96	**5.18**	5.37	5.82	5.16	8.03	**6.68**	7.73	9.36	5.49	**6.28**	6.13	5.47
Tdecod [ms]	1.73	**0.11**	0.59	2.21	1.01	1.37	**0.15**	0.36	1.60	0.91	5.33	**0.49**	1.10	6.32	3.90	**0.61**	1.27	3.87
Trender [ms]	0.53	0.40	0.39	0.43	0.23	0.43	0.41	0.34	0.36	0.48	0.45	0.34	0.36	0.54	0.52	**0.33**	0.38	0.35

client–server clock offset was less than 0.5ms during all our testing. All applications were executed within the Google Chrome Web browser.[44]

B. IoT MESSAGING TEST APPLICATION

Since our intent was to test Web performance of IoT application layer messaging protocols, we selected pure JavaScript implementation for application clients, choosing the WebSocket for the underlying protocol for communication with a message broker, thus limiting our test case only to the HTML5 platform. On the other hand, we selected pure Java implementation for sensor data publisher clients in order to ensure the best possible portability. We here provide the list of used client implementations and message brokers per protocol.

For the *MQTT* protocol, we used components provided by Eclipse foundation, i.e. the message broker is Mosquitto,[45] while JavaScript and Java clients are developed within Paho project.[46] Mosquitto is implemented in C++ and it is distributed as a binary executable, but in order to get WebSocket support, the user should build Mosquitto with third party library libwebsocket.[47] In one test case, we also used a HiveMQ[48] message broker. We set MQTT QoS to work without acknowledge messages (QoS level 0).

XPPP test environment is based around OpenFire server,[49] and Smack[50] Java library, both provided by Ignite Realtime community site. We used Strophe.js[51] JavaScript library, which has a plenty of available plug-ins,[52] and for this test we used pubsub.js.

AMQP tests were performed with Apache Qpid 6.0.3[53] Java message broker. On the publisher side, we employed Rabbit Java client,[54] and as a JavaScript application client, we used patched Kaazing JavaScript AMQP client.[55] The patch was necessary because the original library was designed to communicate through a separate gateway component with an AMQP broker. The used clients implement AMQP 0.9.1 protocol version, which is not identical to the latest 1.0.

OMG's *DDS* specification allows for participants to communicate with TCP and UDP transports, and WebSocket is not supported by the standard. However, the Prismtech's Vortex platform provides a JavaScript client[56] for a DDS protocol, although with the use of additional dedicated standalone Java server which converts communication to the client via WebSocket. Thus we based the DDS messaging test on Prismtech's Vortex components.[57]

The client application is identical as in the previous test application, except for the fact that the communication part relies on messaging protocols' JavaScript implementations. In this test setup, we did not need to use any clock synchronization mechanism between computers running clients and message brokers, because we put both clients to run on the same computer, while message brokers ran on the server computer. In that way, we actually measured the latency of message round-trip time. The sensor data were also generated at the interval of 1s. Messages contain sensor data from one up to five sensor nodes and they are encoded using JSON format.

VII. RESULTS AND DISCUSSION

A. HTML5 APPLICATION

The test results for the HTML5 application are presented in Table 2 and Figure 4. The string representation of Protocol Buffers (PBF) produces significantly shorter messages than any other message encodings used for all three platforms. In contrast, the transfer time of such messages is higher than for JSON and GWT serializer formats. The explanation for this is that although the repacking of PBF to and from a string is a simple operation, it consumes time on both the server and client sides. Note that the Atmosphere-GWT internal conversion of the PBF message produces a larger message size than

[44]https://www.google.com/intl/en/chrome/browser/
[45]https://mosquitto.org/
[46]http://www.eclipse.org/paho/downloads.php
[47]https://libwebsockets.org/
[48]http://www.hivemq.com/
[49]https://www.igniterealtime.org/projects/openfire/
[50]https://www.igniterealtime.org/projects/smack/index.jsp
[51]http://strophe.im/strophejs/
[52]https://github.com/strophe/strophejs-plugins
[53]https://qpid.apache.org/
[54]https://www.rabbitmq.com/java-client.html
[55]https://github.com/afranchuk/javascript.client
[56]http://www.prismtech.com/vortex/vortex-web
[57]http://www.prismtech.com/vortex/overview

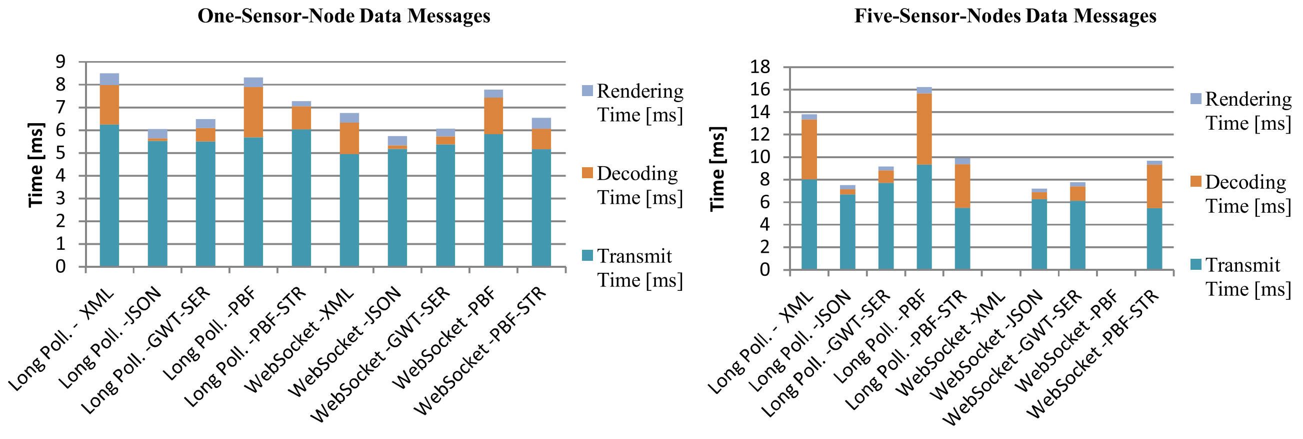


FIGURE 4. HTML5 application test results for one and five-sensor-nodes data message.

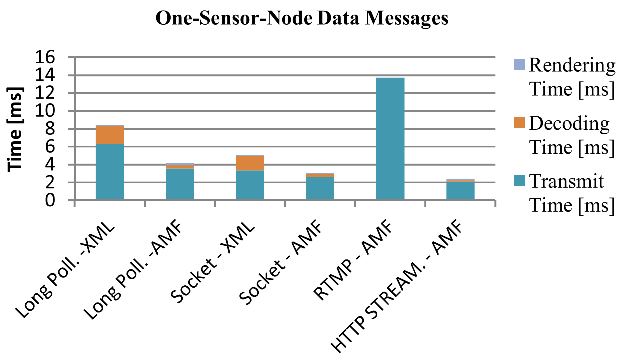
TABLE 3. Adobe flash application Test Results.

Protocol	Long-Polling	Long-Poling	Socket	Socket	RTMP	HTTP Stream.	Long-Polling	Long-Polling	Socket	Socket	RTMP	HTTP Stream.
Encoding	XML	AMF	XML	AMF	AMF	AMF	XML	AMF	XML	AMF	AMF	AMF
Nodes	1	1	1	1	1	1	5	5	5	5	5	5
Payload Size [chars]	2484	995	?	820	**746**	820	4838	2172	?	1997	**1924**	1997
Ttotal [ms]	8.43	4.16	5.07	3.05	13.67	**2.41**	12.69	5.86	8.65	5.03	15.12	**4.56**
Ttransfer [ms]	8.3	3.92	4.95	2.92	13.66	**2.23**	12.55	5.69	8.44	4.93	15.05	**4.45**
Ttransmit [ms]	6.34	3.58	3.35	2.58	13.66	**2.11**	7.38	4.91	3.44	4.85	13.89	**4.29**
Tdecod [ms]	1.96	0.34	1.6	0.34	**0**	0.12	5.17	0.78	5.00	**0.08**	1.16	0.16
Trender [ms]	0.13	0.24	0.12	0.13	0.01	**0.18**	0.14	0.17	0.21	0.1	0.07	**0.11**

the original PBF by a factor of approximately 3.6 to create a UTF-8 string used for transfer over an HTTP connection. This was our motivation to manually repack PBF in a shorter string by a factor of more than 7. An additional problem with the PBF message format is that all used frameworks—i.e., GWT, Atmosphere, and dcodeIO—use their own representation of long integers and byte arrays, which induce additional overhead for preparing the data format for dcodeIO PBF buffer. Test results show that JSON messages have the lowest latencies in our HTML5 testing regardless of the protocol that is employed. The primary reason for this is the browsers' optimization for decoding JSON messages because that operation is not interpreted but directly executed. In the case of other messages, the Web browser interprets JavaScript-specific decoding code. We can conclude that in an environment with high-speed communication, extremely short message formats do not provide any benefit if their encoding/decoding is not optimized. WebSocket has lower latency than long-polling, particularly because of the header overhead of long-polling. The difference in *Ttransmit* of these protocols for the same message encoding is very small, typically less than 1ms. The graphic rendering time is less than 1ms in all tests, so we can conclude that HTML5 canvas implementation provides more than satisfactory performance regarding standard Internet of Things application requirements. The tests for five-sensor-nodes over WebSocket with XML and PBF encodings have failed, because the framework does not accept messages larger than 10K bytes for sending over WebSocket.

B. ADOBE FLASH APPLICATION

The Adobe Flash platform test results contain data for four different protocols and two message encodings. The results are shown in Table 3 and Figure 5. All measured times are noticeably lower compared with other platforms. The only exception is the latency when the RTMP protocol is employed. One should recall that RTMP is designed for real-time deliverance of video and audio files when buffering must be employed, inducing the constant latency necessary for buffer filling. The communication with AMF performs better than with XML in all combinations, and there is noticeably very low decoding time for AMF messages: in the case of one-sensor-node message, the decoding time was 0ms, i.e. it was below the timer resolution (1ms) in all test iterations.



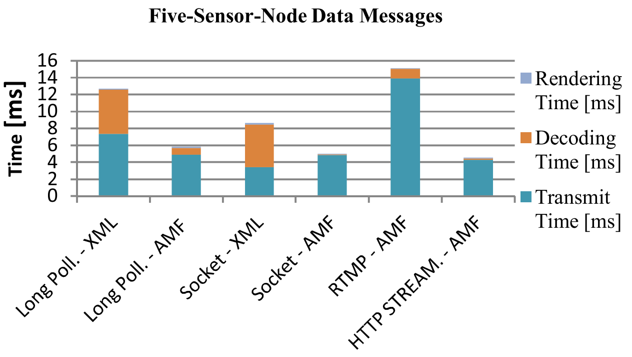


FIGURE 5. Adobe flash application test results for one and five-sensor-nodes data message.

TABLE 4. Microsoft silverlight application test results.

Protocol	Long-Polling	Long-Polling	Socket	Socket	Long-Polling	Long-Polling	Socket	Socket
Encoding	XML	JSON	XML	JSON	XML	JSON	XML	JSON
Nodes	1	1	1	1	5	5	5	5
Payload Size [chars]	6455	2631	6227	2403	27350	10002	27301	9774
Ttotal [ms]	14..13	15..76	**13..16**	13.97	17.06	24.09	**15.55**	22.55
Ttransfer [ms]	13.19	15.69	**12.23**	13.87	16.59	23.78	**15.38**	22.21
Ttransmit [ms]	13.1	13.64	**12.13**	11.89	14.46	15.42	**13.49**	13.9
Tdecod [ms]	0.09	2.05	**0.1**	1.98	2.13	8.36	**1.89**	8.31
Trender [ms]	0.94	0.07	0.93	0.1	0.47	0.31	0.17	0.34

The AMF highest average decoding time for one sensor node message is 0.34ms and that for five sensor node message is 1.16ms. NIO socket implementation performs better than the long-polling protocol in all cases. Surprisingly, HTTP streaming is the most efficient protocol considering latencies, although the difference in its *Ttransmit* time compared with the socket protocol is relatively small. Thus, the decision on the most appropriate protocol for real-time messaging on this platform depends mostly on other network settings such as firewalls, for example. However, some of our tests with HTTP streaming failed when additional header data were included for measuring times and message sizes.

C. MICROSOFT SILVERLIGHT APPLICATION

In the case of the Silverlight test application, there are four combinations of protocols and message encodings. The Silverlight application showed the largest message sizes among all tested applications. The reason for this is the selection of SOAP as the core messaging protocol in the WCF architecture. Although the WCF architecture offers a very flexible model with the ability for developers to define service contracts even on the client side, the cost for that is evident. The socket protocol together with XML message encoding achieves the lowest total latency time (see Table 4 and Figure 6). Although JSON message encodings produce 2.4 to 2.8 times shorter messages than XML encodings, the decoding time for XML messages is approximately 4 times smaller than for JSON messages in the case of five sensor node messages (1.89ms to 8.31ms). We can only conclude that Microsoft has a much better optimization of data serialization/deserialization operation for XML encodings than for JSON within the .NET library. The socket protocol performs better than long-polling (*Ttransmit* values for XML messages are 12.13ms to 13.10ms and 13.49ms to 14.46ms, respectively), but the impact on the total latency time is smaller due to the message encoding. We can also notice that message sizes for long-polling are approximately 300 bytes longer than with the socket protocol. The rendering time is always less than 1ms, which is similar to other platforms.

D. MESSAGE ENCODING COMPARISON

Message size values produced by all message encodings on the tested platforms are presented in Table 5 and Figure 7. Protocol Buffer String creates the shortest message among all message encodings. In the case of five sensor nodes, Adobe's AMF encoding produces a 50% longer message, 1343 characters vs. 1997 characters, but this is smaller than any other

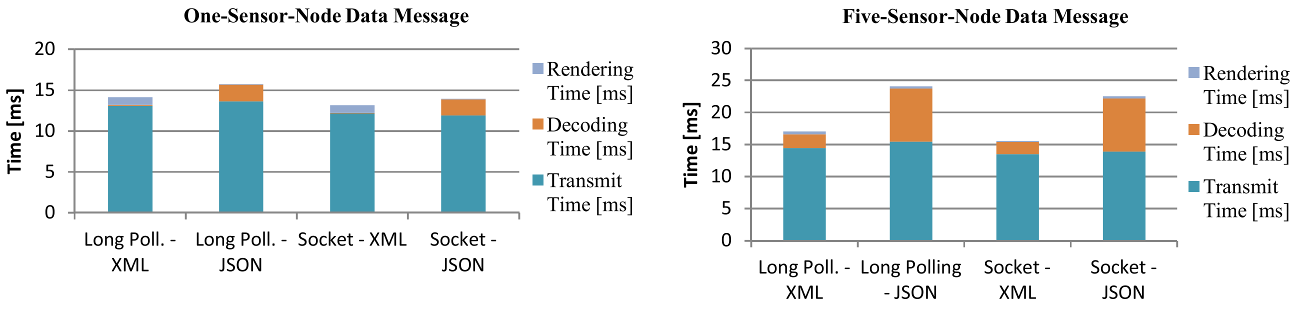


FIGURE 6. **Microsoft silverlight application test results for one and five-sensor-nodes data message.**

TABLE 5. **Message sizes for all message encodings.**

Message format	Message Payload Size – one sensor node [chars]	Message Payload Size – five sensor nodes [chars]
XML (HMTL5)	2322	11281
JSON (HTML5)	1478	7163
GWT Serializer (HTML5)	1153	4485
Google Protocol Buffer (HTML5)	1966	9578
Google Protocol Buffer String (HTML5)	**272**	**1343**
XML (Adobe Flash- Long-Polling)	2484	4838
AMF (Adobe Flash-Socket)	820	1997
XML (Microsoft Silverlight – Socket)	6227	27301
JSON (Microsoft Silverlight – Socket)	2403	9774

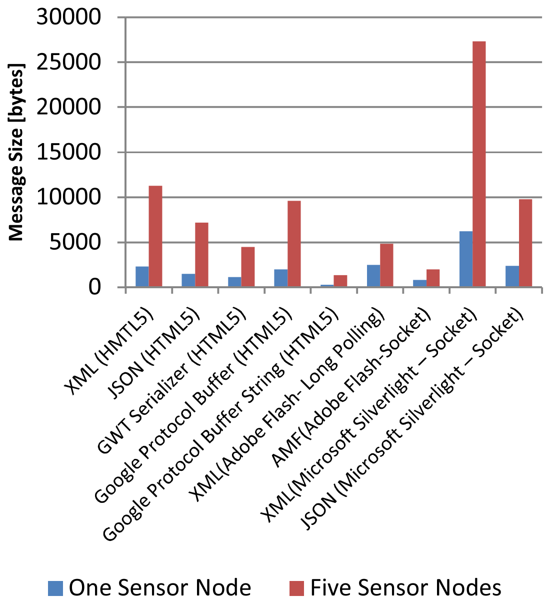


FIGURE 7. **Message payload sizes for all message encodings.**

message encodings. For one sensor node, the difference between PBF String and AMF message size is three times, 272 vs. 820. Furthermore, in the case of five-sensor-node messages, the XML message size on Adobe Flash is much smaller than XML on the HTML5 platform, 4838 characters vs. 11,281 characters, whereas in the case of one-sensor-node messages, XML messages on both platforms have similar sizes. This implies that header overhead in Adobe's message stack is significant when message payload is small, but the header remains compact for larger message payloads. However, the message header overhead is significantly higher for the Silverlight platform than for other platforms because its XML and JSON messages have much larger sizes than on other platforms.

E. OVERALL COMPARISON OF WEB PLATFORM PERFORMANCES

The best test results per platform are shown in Table 6 and Figure 8. Based on our tests, the winning combination is an Adobe Flash application based on the HTTP streaming protocol and AMF message encoding. This combination achieved the shortest latency and consequently offers communication with the highest message rate: for the one-sensor-node message use case, the average total latency is 2.41ms, whereas for five-sensor-node messages, the latency is 4.56ms. The HTML5 application is the second best, with latencies of 5.74ms and 7.22ms for one- and five-sensor-node messages, respectively. A winning combination in the HTML5 case is the WebSocket protocol together with the JSON message format, primarily because of the browser's optimization of JSON message decoding. The test results for the Silverlight application are significantly poorer than for the other two platforms (13.16ms and 15.55ms);as we discussed, we blame this on the WCF internal architecture based on SOAP messages, which induce huge message size overhead.

Generally, the communication protocol has less of an impact than the message encoding for a given platform. We can explain these results by the fact that owing to high-speed communication in LAN, there is a small difference in transmission time for a certain range of message sizes.

TABLE 6. The best test results per platform for one and five-sensor-node messages.

Platform	HTML5	Adobe Flash	Microsoft Silverlight	HTML5	Adobe Flash	Microsoft Silverlight
Protocol	WebSocket	HTTP Streaming	Socket	WebSocket	HTTP Streaming	Socket
Encoding	JSON	AMF	XML	JSON	AMF	XML
Sensor Nodes	1	1	1	5	5	5
Message Payload Size	**1478**	1639	6227	7163	3994	27301
Ttotal [ms]	5.74	**2.41**	13.16	7.22	**4.56**	15.55
Ttransfer [ms]	5.33	**2.23**	12.23	6.89	**4.45**	15.38
Ttransmit [ms]	5.18	**2.11**	12.13	6.28	**4.29**	13.49
Tdecod [ms]	0.15	0.12	**0.1**	0.61	**0.16**	1.89

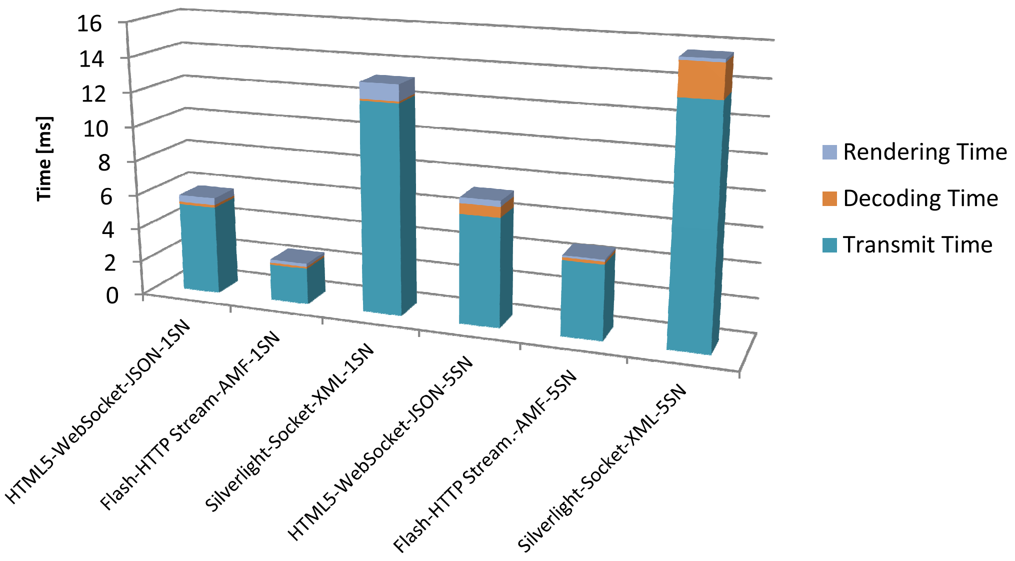


FIGURE 8. The best test results per platform for one and five-sensor-node messages.

However, we can notice that the implementation of message encoding/decoding significantly varies among platforms, despite the same processing requirement. The best illustration for this is the message decoding time for XML and JSON messages. Whereas Web browsers have the most optimized JSON implementation, Silverlight decodes XML messages most rapidly.

This leads us to the conclusion that each platform has the lowest latencies with its favorite message encodings. We have not achieved good results by employing extremely short encoding in HTML5 applications, mainly because of suboptimal implementation resulting from the frameworks mixture. The socket protocol is mainly better than long-polling for all platforms, but not significantly. Interestingly, the lowest latency in the test has been shown by the HTTP streaming protocol on the Adobe Flash platform. This is because it is one-way communication with relatively small header overhead. The tests have also shown that our Internet of Things application requirement for graphics rendering is not too demanding, and the differences among platforms are negligible.

F. WEB PERFORMANCE TEST OF IoT MESSAGING PROTOCOLS

In the first test case we measured the latency of message transfer from a publisher to a subscriber, which is a *Ttransfer* time as described in Section V and refers to the timestamp after message decoding on the client side.

Measured times are shown in Table 7 and Figure 9. The shortest latency is produced by the MQTT protocol, followed by AMQP, while the difference between XMPP and DDS are negligible. For the one-sensor-node message test case, MQTT achieved latency of only 2.53ms. The latencies proportionally grow with the increase of the message sizes. The results are quite predictable. MQTT as very lightweight protocol, induce small overhead in message handshaking protocol,

TABLE 7. The latency measurements of IoT web application messaging protocols.

Protocol	Number of Sensor Nodes	1 Sensor Node	2 Sensor Nodes	3 Sensor Nodes	4 Sensor Nodes	5 Sensor Nodes
	Message Payload Size [chars]	1475-1477	2896-2901	4316-4321	5734-5746	7157-7169
MQTT	*Ttransfer* [ms]	**2.53**	**3.38**	**3.65**	**3.84**	**4.33**
AMQP	*Ttransfer* [ms]	3.99	4.25	4.61	4.82	4.93
XMPP	*Ttransfer* [ms]	4.11	4.97	5.28	5.89	6.34
DDS	*Ttransfer* [ms]	4.3	5.04	5.48	5.56	5.72

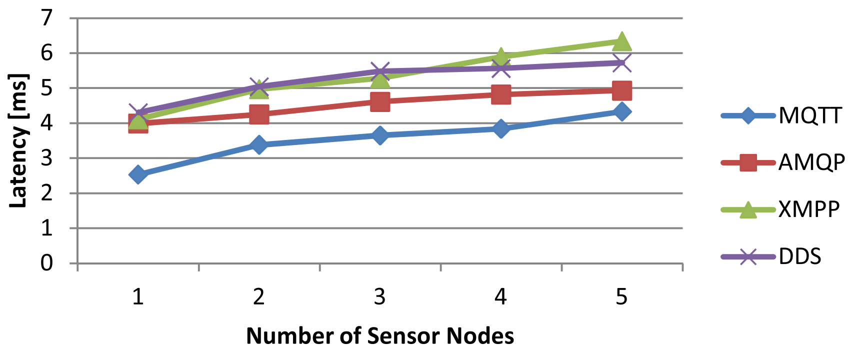


FIGURE 9. The latency measurement of IoT web application messaging protocols.

while AMQP based on binary message encodings efficiently handles sensor data messages. XMPP obviously produces higher overhead than the first two protocols by embedding messages in XML stanza structure. Although DDS uses the typed channel for communication, the impact of protocol conversion done by intermediate Java server cannot be ignored.

All measured times are of few milliseconds, while the timer resolution is 1ms, so even though the results are average values of a relatively large number of transferred messages, the results are approximated values. In order to minimize error due to the timer resolution, as well as to measure the maximum throughput capacity of each protocol, we conducted another test in which we measured the number of round-trip transferred messages, with the following scenario: the publisher publishes a message and upon receiving it, the subscriber immediately sends back that message to the publisher. The process continues as the publisher also immediately re-sends the same message to the subscriber etc. The results show the number of passed round-trip messages in the interval of 100s. In other words, we got that:

$$Ttransmission[s] = 100/2 * number\ of\ passed\ messages$$

Ttransmission time is different than the *Transmit* time analyzed in the Section V, because the message encoding time is not included. Our initial attempt was to perform the new test in the same configuration as in the previous case, with Java and JavaScript clients. However, the DDS test failed, because its Java client did not receive messages from the JavaScript client. Fortunately, the combination of both Java clients, as well as both JavaScript clients worked fine, so we repeated the test with the same configuration for three other protocols in order to appropriately compare the results which are shown in Table 8 and Figure 10.

These tests indeed show very different results. In the case of AMQP and XMPP protocols, the throughput rate values are aligned with the latency test: AMQP achieves slightly higher throughput message rate than XMPP, 229.38 msg/s to 187.87 msg/s. However, we have to emphasize that results for AMQP protocol significantly varied if tests were repeated several times, and the best results were measured immediately after the message broker started. For MQTT protocol, the message throughput rate in the test case of Java-Java clients is as expected and it is significantly higher than for both AMQP and XMPP protocols with 302.48 msg/s, comparing to 266.97 msg/s and 196.04 msg/s respectively. However, in our default test case with Java<->JavaScript clients as well as for JavaScript<->JavaScript clients, the results for MQTT are very poor, below 10 messages per second, i.e. 9.85 msg/s and 3.33 msg/s. The reason for this anomaly lies in the internal event-loop architecture of Mosquitto (we tested versions up to 1.4.8.) which affects the performance of sending data over WebSocket (libwebsocket library version 2.0.2. was used) and it requires the re-implementation of the main event loop. We ran the same test using the HiveMQ broker in

TABLE 8. The message throughput rate measurements of IoT web application messaging protocols.

Protocol	Java – JavaScript clients		Java – Java clients		JavaScript – JavaScript clients	
	Message Throughput Rate [msg/s]	*Ttransmission* [ms]	Message Throughput Rate [msg/s]	*Ttransmission* [ms]	Message Throughput Rate [msg/s]	*Ttransmission* [ms]
MQTT(Mosquitto)	9.85	50,76	302.48	1,65	3,33	150,15
MQTT(HiveMQ)	177.37	2,82	194.60	2,57	**354,64**	**1,41**
AMQP*	**229.38**	**2,18**	266.97	1,87	208,97	2,39
XMPP	187.85	2,66	196.04	2,55	195,63	2,56
DDS	X	X	**463.94**	**1,08**	181,88	2,75

* See in Text.

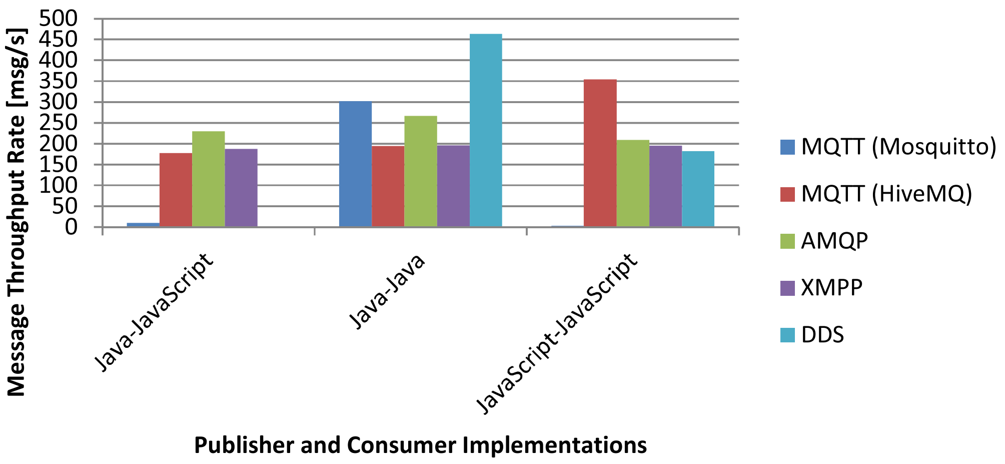


FIGURE 10. The message throughput rate measurements of IoT web application messaging protocols.

order to check correctness of Paho JavaScript MQTT library, and the message throughput rate of JavaScript<->JavaScript clients is the highest of all tested protocols, reaching 354.60 msg/s! However, in the latency test, the HiveMQ based setup showed noticeably higher latency than the Mosquitto broker based test. The second unexpected result is more than 50% higher message throughput rate for the DDS protocol than for the second best protocol, which is MQTT, in the configuration with Java-Java clients, 463.94 msg/s to 302.48 msg/s. In that case, the direct end-to-end TCP connection is established between clients, without any discovery or routing component as a mediator. The result for setup with JavaScript-JavaScript DDS clients is lower than for all other protocols, 181.88 msg/s. As in the latency test, this is the consequence of the protocol conversion that is done by the Java standalone server component, which takes time.

We can notice that the best transfer time for transferring a message in one-sensor-node test case for HTML5 platform shown in Table 6 is 5.33ms, which is significantly higher than any transfer time for messaging protocols shown in Table 7, ranging from 2.53ms to 4.3ms. Although the first test has less precise time measurement because the clock synchronization mechanism adds certain error, we can conclude that message delivery using publish/subscribe messaging protocols over message brokers has more efficient implementation and induces a slightly lower latency than conservative implementation of asynchronous message delivery through an application server protocol stack.

As a general remark, the variety of JavaScript client and message broker implementations make the decision of messaging protocol for IoT Web applications far from straightforward. The bug of Mosquitto message broker, resulting in the poor performance while employing JavaScript client and WebSocket communication, prevents from recommending the MQTT for any IoT Web applications scenario. On the other hand, the performance of MQTT based on a HiveMQ message broker is quite similar to other protocols. In general, every protocol has certain disadvantages: MQTT performances strongly depend on the used message broker; DDS requires the use of the additional dedicated server component for a Web client; AMQP has the highest fluctuation of message throughput rate among all protocols; although XMPP does not have any concrete drawbacks, its measured times are never among the best for any test case.

On the other hand, when comparing the performance of Java clients using TCP transport, DDS is the most efficient IoT messaging protocol, offering from 50% to more than 200% higher message rate than other protocols.

VIII. CONCLUSION

Having performed these tests, we conclude that HTML5 has become a mature platform for implementing Internet of Things real-time messaging applications. It is true that the Adobe Flash platform achieved the shortest latency among all tested platforms, but this difference is small. We can blame this on the poor support for binary message formats in the HTML5+JavaScript platform, which is the key to the efficient performance of Adobe Flash applications. We attempted to overcome this limitation by repacking Protocol Buffer messages into a string, but, unless we obtained the shortest message sizes, we did not improve the latency, mainly because of the data representation incompatibilities of the used frameworks. However, by embedding the appropriate implementation of binary messages into Web browsers, we can expect lower latencies in real-time messaging applications running on the HTML5 platform and hence obtain a more efficient platform. In other analyzed aspects of interest such as communication protocols and graphics rendering, HTML5 performs equally well as the Adobe Flash and Microsoft Silverlight platforms. WebSocket has achieved slightly better results than long-polling, as expected. HTML5 graphical performance is more than capable of satisfying the requirements of typical Internet of Things applications.

If we also consider the much better support for HTML5 than other compared platforms, especially on mobile phones and tablets, it is clear that this platform has a better future in the domain of Internet of Things applications. This suggests that application providers should start moving existing applications from Adobe Flash or Microsoft Silverlight platforms to HTML5.

The Web performance test of IoT messaging protocols has revealed that protocols have certain shortcomings as a result of message broker and JavaScript client implementations. It is up to the IoT application developers to carefully analyze use-case scenario and applications' critical requirements and choose the appropriate messaging protocol. Our study could be a source of valuable information, since we have identified several cases of which application developers should be aware. Generally, we can recommend MQTT as a protocol which can fulfill IoT Web application requirements in most scenarios, except the case when the Mosquitto message broker is used and the JavaScript client should publish messages over WebSocket.

REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[2] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of Things: Vision, applications and research challenges," *Ad Hoc Netw.*, vol. 10, no. 7, pp. 1497–1516, Sep. 2012.

[3] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 4th Quart., 2015.

[4] Z. Babovic and V. Milutinovic, "Novel system architectures for semantic-based integration of sensor networks," *Adv. Comput.*, vol. 90, pp. 91–183, Dec. 2013.

[5] J. Åkerberg, M. M. Gidlund, and M. Björkman, "Future research challenges in wireless sensor and actuator networks targeting industrial automation," in *Proc. 9th IEEE Int. Conf. Ind. Inform. (INDIN)*, Jul. 2011, pp. 410–415.

[6] R. van der Meulen. (Nov. 2015). *Gartner Says 6.4 Billion Connected 'Things' Will Be in Use in 2016, Up 30 Percent From 2015*, accessed on Jul. 20, 2016. [Online]. Available: http://www.gartner.com/newsroom/id/3165317

[7] (Nov. 2015). *Ericsson Mobility Report*, accessed on Jul. 20, 2016. [Online]. Available: http://www.ericsson.com/res/docs/2015/mobility-report/ericsson-mobility-report-nov-2015.pdf

[8] M. Asay. (Jun. 2014). *The Internet of Things Will Need Millions of Developers By 2020*, accessed on Jul. 20, 2016. [Online]. Available: http://readwrite.com/2014/06/27/internet-of-things-developers-jobs-opportunity

[9] *COMET*, accessed on Jul. 20, 2016. [Online]. Available: http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser

[10] I. Fette and A. Melnikov, "The WebSocket protocol," IETF Internet Draft, Work in Progress, Dec. 2011.

[11] (Jul. 20, 2016). *HTML Canvas 2D Context*. [Online]. Available: http://www.w3.org/TR/2dcontext/

[12] Z. Shelby, K. Hartke, and C. Bormann, *The Constrained Application Protocol (CoAP)*, document RFC 7252, Jun. 2014, accessed on Jul. 20, 2016. [Online]. Available: https://tools.ietf.org/html/rfc7252

[13] C. Gutwin, M. Lippold, and T. C. N. Graham, "Real-time groupware in the browser: Testing the performance of Web-based networking," in *Proc. ACM Conf. Comput. Supported Cooperat. Work*, 2011, pp. 167–176.

[14] *Message Queue Telemetry Transport, MQTT*, accessed on Jul. 20, 2016. [Online]. Available: http://mqtt.org/

[15] *Advanced Messaging Quieing Protocol*, accessed on Jul. 20, 2016. [Online]. Available: https://www.amqp.org/

[16] OASIS, Burlington, MA, USA. *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0*, accessed on Jul. 20, 2016. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=amqp

[17] *Extensible Messaging and Presence Protocol (XMPP)*, accessed on Jul. 20, 2016. [Online]. Available: http://xmpp.org/

[18] P. Saint-Andre. (2011). *Extensible Messaging and Presence Protocol (XMPP): Core*, accessed on Jul. 20, 2016. [Online]. Available: http://www.rfc-editor.org/info/rfc6120

[19] S. Bendel, T. Springer, D. Schuster, and A. Schill, "A service infrastructure for the Internet of Things based on XMPP," in *Proc. IEEE Int. Conf. Pervas. Comput. Commun. Workshops (PERCOM Workshops)*, 2013, pp. 385–388.

[20] M. Kirsche and R. Klauck, "Unify to bridge gaps: Bringing XMPP into the Internet of Things," in *Proc. IEEE Int. Conf. Pervas. Comput. Commun. Workshops (PERCOM Workshops)*, Mar. 2012, pp. 455–458.

[21] A. Hornsby and E. Bail, "μXMPP: Lightweight implementation for low power operating system Contiki," in *Proc. Int. Conf. Ultra Modern Telecommun. Workshops (ICUMT)*, Oct. 2009, pp. 1–5.

[22] *XMPP Client for mbed*, accessed on Jul. 20, 2016. [Online]. Available: https://developer.mbed.org/cookbook/XMPPClient

[23] *Data Distribution Service (DDS)*, accessed on Jul. 20, 2016. [Online]. Available: http://portals.omg.org/dds/

[24] P. Waher and Y. Doi, *Efficient XML Interchange (EXI) Format*, document XEP-0322, 2013.

[25] T. Kamiya and J. Schneider, *Efficient XML Interchange (EXI) Format 1.0*, document Rec. REC-Exi-20110310, World Wide Web Consortium, Cambridge, MA, USA, 2011.

[26] *Google Protocol Buffer*, accessed on Jul. 20, 2016. [Online]. Available: https://developers.google.com/protocol-buffers/

[27] *JVM-Serializers Benchmark*, accessed on Jul. 20, 2016. [Online]. Available: https://github.com/eishay/jvm-serializers/wiki

[28] *Action Message Format—AMF 3*, accessed on Jul. 20, 2016. [Online]. Available: http://www.adobe.com/content/dam/Adobe/en/devnet/amf/pdf/amf-file-format-spec.pdf

[29] N. Fernández, J. Arias, L. Sánchez, D. Fuentes-Lorenzo, and Ó. Corcho, "RDSZ: An approach for lossless RDF stream compression," in *Proc. Extended Semantic Web Conf. (ESWC)*, vol. 8465, 2014, pp. 52–67.

[30] A. K. Joshi, P. Hitzler, and G. Dong, "Logical linked data compression," in *The Semantic Web: Semantics and Big Data*, vol. 7882, 2013, pp. 170–184.

[31] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias, "Binary RDF representation for publication and exchange," *J. Web Semantics*, vol. 19, pp. 22–41, Mar. 2013.

[32] V. Gazis *et al.*, "A survey of technologies for the Internet of Things," in *Proc. Int. Wireless Commun. Mobile Comput. Conf. (IWCMC)*, 2015, pp. 1090–1095.

[33] V. Karagiannis, P. Chatzimisios, F. Vazquez-Gallego, and J. Alonso-Zarate, "A survey on application layer protocols for the Internet of Things," *Trans. IoT Cloud Comput.*, vol. 3, no. 1, pp. 11–17, 2015.

[34] P. Masek *et al.*, "Implementation of true IoT vision: Survey on enabling protocols and hands-on experience," *Int. J. Distrib. Sensor Netw.*, vol. 12, no. 4, p. 8160282, 2016.

[35] A. Bröring *et al.*, "New generation sensor Web enablement," *Sensors*, vol. 11, no. 3, pp. 2652–2699, 2011.

[36] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of MQTT and CoAP via a common middleware," in *Proc. IEEE 9th Int. Conf. ISSNIP*, Apr. 2014, pp. 1–6.

[37] M. Laine and K. Säilä, "Performance evaluation of XMPP on the Web," Aalto Univ., Aalto, Finland, Tech. Rep., 2012.

[38] J. L. Fernandes, I. C. Lopes, J. J. P. C. Rodrigues, and S. Ullah, "Performance evaluation of RESTful Web services and AMQP protocol," in *Proc. 5th ICUFN*, 2013, pp. 810–815.

[39] V. Pimentel and B. G. Nickerson, "Communicating and displaying real-time data with WebSocket," *IEEE Internet Comput.*, vol. 16, no. 4, pp. 45–53, Jul. 2012.

[40] K. Ma and R. Sun, "Introducing WebSocket-based real-time monitoring system for remote intelligent buildings," *Int. J. Distrib. Sensor Netw.*, vol. 9, no. 12, 2013, Art. no. 867693.

[41] R. C. Hoetzlein, "Graphics performance in rich Internet applications," *IEEE Comput. Graph. Appl.*, vol. 32, no. 5, pp. 98–104, Sep./Oct. 2012.

[42] A. Kos, S. Tomažič, and A. Umek, "Evaluation of smartphone inertial sensor performance for cross-platform mobile applications," *Sensors*, vol. 16, no. 4, p. 477, 2016.

[43] A. Botta, W. Donato, V. Persico, and A. Pescapé, "Integration of cloud computing and Internet of Things: A survey," *Future Generat. Comput. Syst.*, vol. 56, pp. 684–700, Mar. 2016.

[44] M. Díaz, C. Martín, and B. Rubio, "State-of-the-art, challenges, and open issues in the integration of Internet of Things and cloud computing," *J. Netw. Comput. Appl.*, vol. 67, pp. 99–117, May 2016.

[45] J. Mineraud, O. Mazhelisb, X. Suc, and S. Tarkoma, "A gap analysis of Internet-of-Things platforms," *Comput. Commun.*, vols. 89–90, pp. 5–16, Sep. 2016.

[46] J. P. Calbimonte, S. Sarni, J. Eberle, and K. Aberer, "XGSN: An open-source semantic sensing middleware for the Web of Things," in *Proc. 7th Int. Workshop Semantic Sensor Netw.*, Riva del Garda, Italy, Oct. 2014, pp. 51–66.

[47] A. Kos, S. Tomažič, and A. Umek, "Suitability of smartphone inertial sensors for real-time biofeedback applications," *Sensors*, vol. 16, no. 3, p. 301, 2016.

[48] D. Mills. *IEEE 1588 Precision Time Protocol (PTP)*, accessed on Jul. 20, 2016. http://www.eecis.udel.edu/~mills/ptp.html

[49] D. Mills *et al.*, *Network Time Protocol Version 4: Protocol and Algorithms Specification*, document IETF RFC 5905, Jun. 2010, accessed on Jul. 20, 2016. [Online]. Available: http://tools.ietf.org/html/rfc5905

ZORAN B. BABOVIC (S'13) received the M.Sc. degree in electrical engineering University of Belgrade, The School of Electrical Engineering, Serbia, in 2004, where he is currently pursuing the Ph.D. degree. He has been working on several research and software development projects, in cooperation with leading EU Institutes and USA/U.K. companies, such as IPSI Fraunhofer Institute, Germany, Storage Tek, USA, Dow Jones, USA, Maxeler, U.K., in the domain of multimedia, data management, and real-time software systems. Since 2006, he has been a Research Associate with the Innovation Center, University of Belgrade, The School of Electrical Engineering, Serbia. He participated in four EU FP6 and FP7 research projects in the domain of sensor networks, data mining, and computer architecture, as well as in four innovation and research projects funded by the Serbian Ministry of Education, Science and Technological Development. He has authored several conference and journal papers, and gave numerous talks at conferences in Europe.

JELICA PROTIC received the Ph.D. degree in electrical engineering from the University of Belgrade. She is currently an Associate Professor of Computer Engineering and Informatics with University of Belgrade, The School of Electrical Engineering. She was a Principal Designer in pioneer projects of networking proprietary industrial computers. With Milo Tomasevic and Veljko Milutinovic, she co-authored *Distributed Shared Memory: Concepts and Systems* (IEEE CS Press, 1997) and presented numerous pre-conference tutorials on this subject. She also conducted research in the domain of wireless sensor networks. She has long term experience in teaching a diversity of courses in programming languages, as well as the development of various educational software tools. Her research interests include distributed systems, consistency models, computer networks, and all aspects of computer-based quantitative performance analysis and modeling.

VELJKO MILUTINOVIC (M'82–F'03) received the Ph.D. degree in electrical engineering from University of Belgrade in 1982. In 1980, for about a decade, he was with the Purdue University, West Lafayette, IN, USA, as a Faculty Member. He has co-authored the architecture and design of the world's first DARPA GaAs microprocessor. Since 1990, after returning to Serbia, he has been a Faculty Member with University of Belgrade, The School of Electrical Engineering, where he is teaching courses related to computer engineering, sensor networks, and data mining, and also took part in teaching with the University of Indiana in Bloomington, Purdue, Harvard, and MIT. After year 2000, he participated in several FP6 and FP7 projects through collaboration with leading universities and industries in the EU/USA, including Microsoft, Intel, IBM, Ericsson, especially Maxeler. He has lectured by invitation to over 100 European universities. He has authored about 80 papers in SCI journals and about 20 books with major publishers in the USA. He is a member of Academia Europaea.