Uludağ University Journal of The Faculty of Engineering, Vol. 23, No. 2, 2018 RESEARCH

DOI:10.17482/uumfd.313311

A NEW MESSAGE PROCESSING MECHANISM FOR INTERNET OF THINGS

Cengiz TOĞAY*

Received: 16.05.2017; revised:01.02.2018; accepted: 15.05.2018

Abstract: Data is the most valuable thing in Industry 4.0 Era. Devices in the industry produce a tremendous amount of data and they are stored or processed by the services. Because of the advantages of the cloud computing, now the data is on the Internet instead of the local servers. Therefore, thanks to the cloud computing, the data can be processed by third parties. There are two concerns about sharing the data; namely performance and privacy. For near real-time problems, the data should be sent to services as soon as possible. For privacy, only required and allowed part of the data payload should be shared among parties.Therefore, a processing mechanism is needed before sending the data to target nodes. In this study, a new mechanism is proposed for processing/filtering the data based on the target node needs and rules defined by the administrator.

Keywords: Internet of Things, Message Processing, Cloud Computing, Message Oriented Middleware

Nesnelerin İnterneti için Yeni Mesaj İşleme Mekanizması

Öz: Endüstri 4.0 çağında veri en önemli bileşen halini aldı. Endüstrideki cihazlar tarafından üretilen çok ciddi miktarda veriler servisler tarafından işlenmekte ya da saklanmaktadırlar. Bulut mimarilerinin sağladığı avantajlar nedeni ile veriler artık yerel sunucular yerine İnternet ortamında yer almaktadır. Bulut bilişim sayesinde, veriler dış servisler tarafından işlenebilmektedir. Burada iki temel konu ortaya çıkmaktadır. Bunlar performans ve gizliliktir. Gerçek zamanlı uygulamalar için veri servislere mümkün olduğunca çabuk bir şekilde ulaştırılmalıdır. Gizlilik konusunda ise yalnızca verinin gerekli ve izin verilen kısımları paylaştırılmalıdır. Bu nedenle, verilerin hedeflere gönderilmeden önce işleyecek bir mekanizmaya ihtiyaç bulunmaktadır. Bu çalışmada, yöneticilerin tanımladığı kurallar ile hedefin gereksinimlere bağlı olarak çalışan yeni bir veri işleme/filtreleme mekanizması önerilmektedir.

Anahtar Kelimeler: Nesnelerin İnterneti, Mesaj İşleme, Bulut Bilişim, Mesaj Yönelimli Ara katman

1. INTRODUCTION

Nowadays, cloud computing is getting more attention because of the flexibility, reliability, speed, performance, scalability, and maintenance costs. In a cloud environment, physical infrastructure is virtualized to provide hardware and software resources to users based on the demand. Complex systems which include various hardware and software components require cloud computing approaches to efficiently handle large amount of data. In industry, there is a tremendous number of devices and they produce data about status of themselves. The data is sent to servers to store and process. Because of the advantages of cloud computing, data is stored and processed by the servers located in the Internet. This approach creates opportunities for service providers for processing the data. For instance, a vendor/third party can provide a

* Uludag University, Faculty of Engineering, Department of Computer Engineering, 16059, Görükle, Bursa, Turkey
Correspondence Author: Cengiz Toğay (ctogay@uludag.edu.tr)

service utilizing machine learning algorithms to produce valuable information based on the data. Message Oriented Middleware (MOM) a middleware infrastructure is also getting more important for complex systems because of the communication flexibility among heterogeneous architectures (Curry, 2005). Senders and receivers are connected to a server which acts as an intermediary to publish and subscribe messages. Communication parties can be various types of components such as embedded control systems, gateways, software components etc. where they are working in different types of environments and operating systems. For these types of systems where they have to interconnect for creating a single system, a "broker" is needed. The broker should solve the communication problems among sides where they use different communication protocols. Broker usage also presents advantages for scalability and flexibility. Administrators can add new producers or consumers to the system without affecting the available parties. Another advantage to use brokers is anonymity for parties. All parties communicate through Brokers. A message sent by a producer is delivered to consumer through a Broker. The consumer only knows the payload of the message and some relevant headers. There can be no another information such as device IP or mac address etc. Same message can be delivered to more than one party where they subscribed to a topic. The Administrator of the Broker can authorize a customer and/or a publisher which topic or queue can be used by whom. However, there is no authority for the administrator on the payload of the message. The authorized messages are sent through to the available consumers. Brokers such as ActiveMQ (ActiveMQ, 2003) allows the consumers to define filters for which messages (for instance, "temperature > 30 Celsius") should be delivered themselves. Each message completely should be sent to each subscriber with the whole message payload.

Problems which we identified in this paper are 1) same message delivered to all subscribed parties without hiding/changing/inserting some items in data payload, 2) Data owner has not any authority on the data 3) third parties have to handle more data payload than required, 4) there is a privacy issue in terms of the data owner, and 5) extra process and network capacity is required. Therefore, in this paper, we introduce a new payload processing mechanism. The mechanism is implemented on the ActiveMQ Broker and messages are processed through JavaScript codes. Therefore, data payload can be processed to modify and validate. The JavaScript functions are applied for a specific subscriber and they can be inserted/deleted/modified during the runtime by the administrator.

2. RELATED WORK

2.1. ActiveMQ

There exist many hardware (Solace Systems, Tervel, etc.) and software as a service (IBM Webshere MQ and Microsoft Message Queuing), and open source (Rabbit MQ, HornetQ, ActiveMQ, OpenJMS, Qpid, etc.) middleware solutions. We selected and investigated in ActiveMQ open source project. In literature, there are some comparison of the Brokers such as, ActiveMQ is faster than RabbitMQ in terms of the message reception (the client sends message to server). However, RabbitMQ is faster in terms of the message producing (Ionescu, 2015). Also ActiveMQ has better performance than OpenMQ in terms of less memory usage and throughput especially in the small size payloads (Klein et al., 2015). Since, our investigated problems (such as Internet of Things) are mostly based on the message reception on the server side, we selected ActiveMQ instead of the Rabbit MQ and OpenMQ.

The ActiveMQ supports Java Messaging Service (JMS) message exchange standard and across various messaging protocols such as OpenWire, Message Queuing Telemetry Transport (MQTT, 2015), Advanced Message Queuing Protocol 1.0 (AMQP, 2014), The Simple Text Oriented Messaging Protocol 1.2 (STOMP, 2012), REST API, Ajax etc. JMS Publishers and Subscribers connect to Broker (JMS Server) through JMS API. The JMS API introduce an independence/portability to JMS clients from the JMS servers (ActiveMQ, RabbitMQ etc.).

ActiveMQ supports most of the messaging protocols preferred in IoT environments. One of the protocols named XMPP is not supported anymore in the ActiveMQ because of the performance issues. Instead of the XMPP, MQTT or AMQP can be utilized because of throughput and resource requirements (Yassein, et al., 2016).

Asynchronous communication is one of the popular approaches to cloud solutions because of the performance. The parties are not blocked during the communication. A publisher sends a message to the broker and it is assumed that message is processed or stored by a subscriber. Parties do not need to know about each other (availability status, IP address, any information about server etc.). There are some approaches (durable, non-durable, persistent, and nonpersistent) for dealing with the message by the Broker. If the message is persistent, the message is stored by the Broker until they are delivered to targets. There can be server crashes and connection failures between subscriber and Broker. When a message is delivered to at least one subscriber, the message is deleted from the Broker. For durable subscription that can be achieved with publish/subscribe domain, Broker saves messages for the durable subscriber that is not connected when a message arrives. When they are connected, all waiting messages except expired ones are sent to the subscriber. When the durable subscriber unsubscribes from the topic, waiting/saved messages are deleted from the Broker. If there is no subscriber to process received the message by the Broker, the messages are discarded.

JMS identifies two styles of messaging (point-to-point and publish/subscribe). The queue is an implementation of the point-to-point style. When a producer sends a message to the queue, Broker delivers the message once and only once to a single consumer based on the sort of round-robin style as depicted in Figure 1. Broker stores all the messages until they are delivered or expired. The topic is an implementation of the publish/subscribe style. Publishers send a message to a topic and Subscribers register to receive the message from the topic. Messages are delivered to all legitimate (active, durable, and based on the selectors) subscribers.

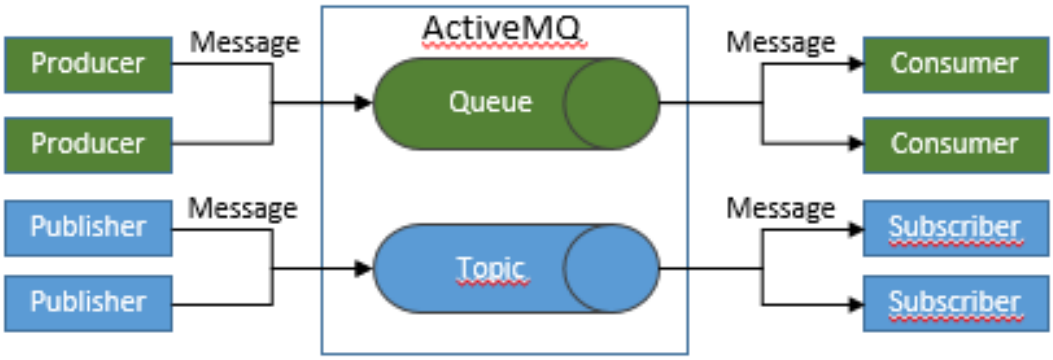

Figure 1:
Point to point (Queue) and Publish/Subscribe (Topic) styles

JMS Publisher sends (publishes) a message to the broker. The broker sends the same message to one or more JMS Subscriber which is called as consumer/subscriber. A JMS client can be producer and consumer at the same time. The JMS message can include any payload (text, binary, map, stream and object messages) as well as headers including message specific information (JMSDestination, JMSDeliveryMode, JMSExpiration, etc.).

Publishers can send any message to a specific topic/queue and brokers decide which message is sent to which consumers. When a consumer subscribes to the topic/queue, the message can be sent directly. However, consumers can specify which messages should be delivered by the broker through Message Selectors. The message selectors are used to filter messages by the Broker based on the payload. For instance, the consumer can set a selector for a

topic/queue so that the message has to include pressure field bigger than a specific value. It should be stressed that selector mechanism is used only to select messages on the Broker based on the specification that is defined by the subscriber. There is no enrichment or discarding items from the message payload.

2.2. Mozilla Rhino JavaScript Engine

As defined before, the proposed mechanism is based on the JavaScript language. Since ActiveMQ is implemented with Java Language. We utilized a JavaScript Engine namely Mozilla Rhino (Rhino, 1997) to execute JavaScript codes in ActiveMQ. JavaScript is an interpreted scripting language. It was introduced to use within browsers to provide more dynamic Web Pages in 1995. Mozilla Rhino allows the interactive execution of JavaScript code within a JVM implementation. In this article, we prefer to use the JavaScript because of five reasons: 1) it is an interpreted language, 2) it is well known by the developers 3) scripts can be easily executed by the Java environment 4) it supports the use of the java objects in the script 5) it can be modified during runtime without recompiling the project.

Mozilla Rhino JavaScript engine is selected because of following items:1) it is an open source project, 2) Java methods can be invoked from the JavaScript or JavaScript execution results can be used by the Java methods, 3) it is embedded as the default Java scripting engine in Java Standart Edition 6 and 7. The Rhino JavaScript engine is used for direct access to Java code from JavaScript for Celtix JavaScript/E4X service implementations (Vinoski, 2006). Host objects in JavaScript provide more capabilities through java classes. Since JavaScript is an interpreted language, codes should be interpreted each time when required. We implemented a management mechanism to utilize the JavaScript functions in terms of performance.

3. ARCHITECTURE STRUCTURE

Internet of things includes various nodes that include devices, gateways, and servers. All these nodes communicate with each other with standard protocols such as MQTT and AMQP. The produced data is sent to a target service (server) or another node (device or gateway) through a Broker. Our proposed filtering mechanism can be used for both publish-subscribe and point-to-point styles with different ways. In this article, we presented the approach based on the JMS publish-subscribe style. As depicted in Figure 2, a Publisher sends temperature and altitude values to ActiveMQ with the “sensor” topic. ActiveMQ sends the message which available subscribers for the “sensor” topic. As it can be seen in Figure 2 that modified payload delivered to “Subscriber 1” and “Subscriber 2” separately. The “Subscriber 1”. The ActiveMQ executes a script for “Subscriber 1” and altitude value is deleted from the message payload. Similarly, we can also modify the message payload with this mechanism. For instance, the Publisher also publishes the location as depicted in Figure 2 and “Istanbul” is delivered in the payload to the “Subscriber 1” as defined by the Publisher. However, “Istanbul” is modified with “Turkey” in message payload before deliver to the “Subscriber 2”. Therefore, the exact location is hidden from the “Subscriber 2”.

Since Java codes are faster than the JavaScripts, we implemented some useful Java methods in a library called by the JavaScript such as removing a specific key from the payload with values, encrypting the message or validating the payload for a specific template matching. JSUtils is a bridge for our library including various utility classes. Some of the classes and their functions are demonstrated in Figure 3. Such as isBinaryMessage in “UtilityFunctions” checks the message in terms of the BinaryMessages. The “remove” function in JSONUtility, deletes the specific property name from the JSON string. Another example, The “encrypt” function in the AES encrypts the message with a key and AES symmetric encryption algorithm. They can be extended as required.

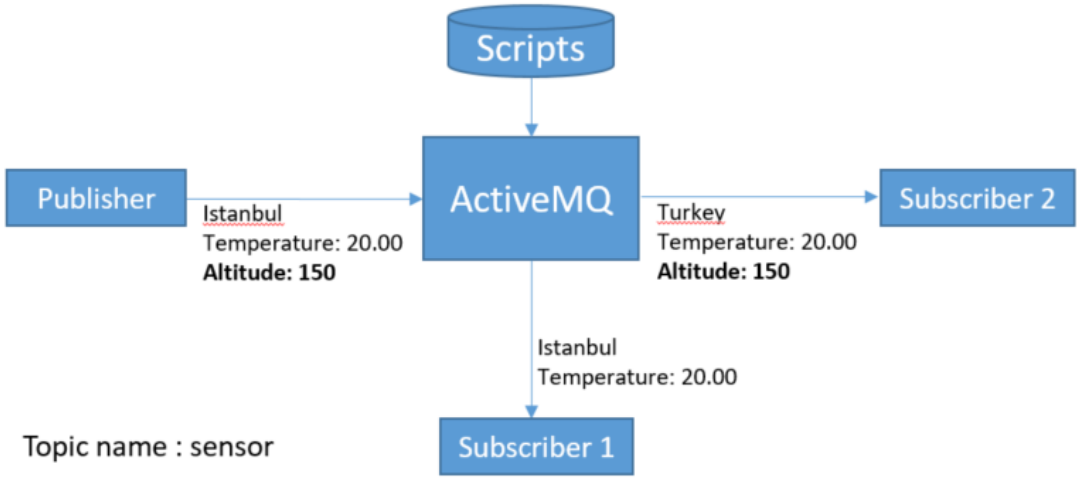

Figure 2:
The Architecture

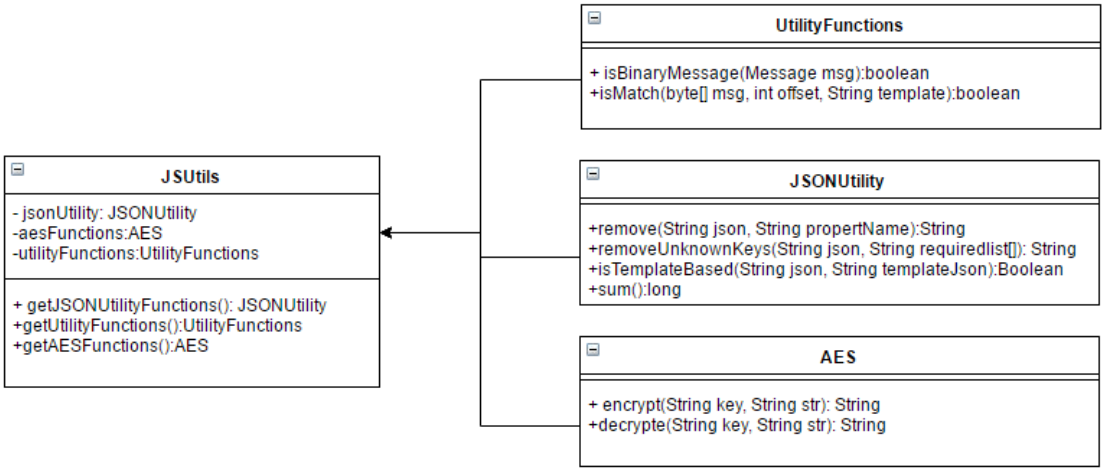

Figure 3:
JSUtil Library Classes

It should be noted that “Utils Classes” have to be compiled before the script is executed. There are two approaches for calling scripts. In the first approach, scripts are compiled before the execution of Java application for each time. This kind of approach have advantages on the memory consumption but disadvantages on the CPU utilization and throughput. As a second approach we proposed in this paper, JavaScript is compiled when needed and stored in the heap memory until it expires. We implemented the JSFunctionManager and related classes as depicted in Figure 4. Whenever a function is used, its lastAccessTime is updated. JSFunctionExpireControl that is a thread checks the expired functions and deletes them from the memory.

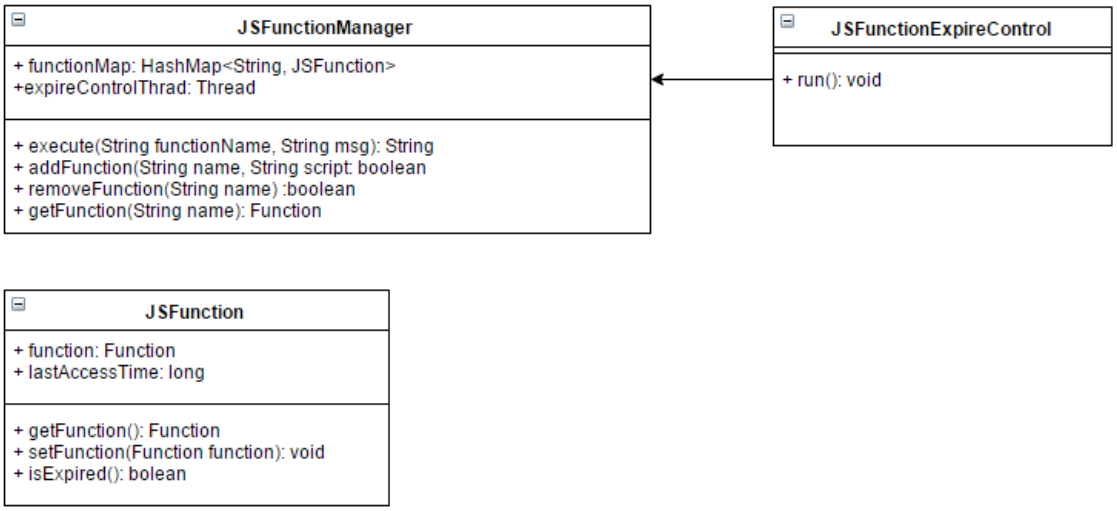

Figure 4:
JSFunction Classes

The library is utilized in the broker to apply rules to messages. For instance, AES encryption script defined in Figure 5 can be performed through java function for the "message" as depicted in Figure 6. Instead of the example "aeskey" in Figure 5, developers can use pre-shared keys in database or generate during login operation of the Broker. The script is added to the manager as an "evaluation_1". Since it is added to the manager, it can be used as many times as required without compilation. The "evaluation_1" script is executed by the manager utilizing the encryption function of the library for a message. The JSFunctionManager manages all JSFunctions in the platform depending on the memory and performance concerns.

```
String script = "function evaluation(message) {"
        + "return util.getAESFunctions().encrypt('aeskey',message);"
        +"}";
```

Figure 5:
JavaScript function encrypt a message

```
JSFunctionManager manager = new JSFunctionManager();
manager.addFunction("evaluation_1 ", script);
manager.execute("evaluation_1 ", message);
```

Figure 6:
Java code utilize the JavaScript command function

Through the proposed approach, subscribers are secured from the unexpected form of message payloads. Administrator can define a template for a publisher and if message payload is not congruent with the template it can be discarded. This kind of message may be formed by the malicious third party or the version of the publisher may be outdated and subscribers are not available anymore for this message payload. The isTemplateBased function in the JSONUtility is one of the useful functions, which validate the JSON string in case of a match with templateJSON as depicted in Figure 7. Since the template is matched, "true" is returned by the function.

```
JSFunctionManager manager = new JSFunctionManager();

String script = "function evaluation(x) {"
+"return util.getJSONUtilityFunctions().isTemplateBased(\"{\\\"temp\\\":\\\"\\\",”
+"\\\"pressure\\\":\\\"\\\"}\",x);}";

manager.addFunction("evaluate", script);

String exampleJSON = "{ \"temp\":25, \"pressure\":1000}";

String result = manager.execute("evaluate", exampleJSON);
```

Figure 7:
Template validation example

Another useful library class is the AES. A message can be encrypted or decrypted by AES class. As shown in Figure 8, the message ("plain text") is encrypted with "aeskey" defined in the script. The aeskey can be obtained from a database through utility classes.

```
JSFunctionManager manager = new JSFunctionManager();

String script = "function evaluation(x) {"
        + "return util.getAESFunctions().encrypt(\"aeskey\",x);}";

manager.addFunction("evaluate", script);

manager.execute("evaluate", "plain text");

String result = manager.execute("evaluate", "plain Text");
```

Figure 8:
Encryption example

4. TEST AND RESULTS

We tested and compared the proposed approach in terms of the performance. Tests are executed in a virtual machine (8192 MB, 4 Core 2.6 GHz). Our objective is only to compare the approached roughly to other approaches and executions are measured in terms of milliseconds and repeated ten times.

4.1. Case 1

We tested a string based replace operation for only “Java function”, “JavaScript function”, and “JavaScript function executes a Java function” as shown in Figure 9, Figure 10, and Figure 11 respectively. The functions, replace the “Istanbul” term with “Turkey”. The replace function is executed ten million times for the three approaches in ten cases. Java function, is completed in average 2782 milliseconds. Similarly, JavaScript function is completed on the average 5892 milliseconds. In this approach, string operation is completed as a native JavaScript operation. Lastly, JavaScript calls the replace function of our library takes on the average 6210 milliseconds. As depicted in Figure 12, replace function in library completed its task very close to the native replace function of JavaScript. Therefore, it is clear that host objects are performed as native functions. However, it should be noted that java function has a better performance than JavaScript.

```
public String replace(String str) {
      return str.replace("Istanbul", "Turkey");
   }
```

Figure 9:
Java function for String replace operation

```
String script = "function evaluation(x) {"
     + " return x.replace(\"Istanbul\", \" Turkey \");"
 + "}";
```

Figure 10:
Javascript function for String replace operation

```
 String script = "function evaluation(x) {"
          + " return util.getUtilityFunctions().replace(x);"
          + "}";
```

Figure 11:
JavaScript function utilize the library for String replace operation

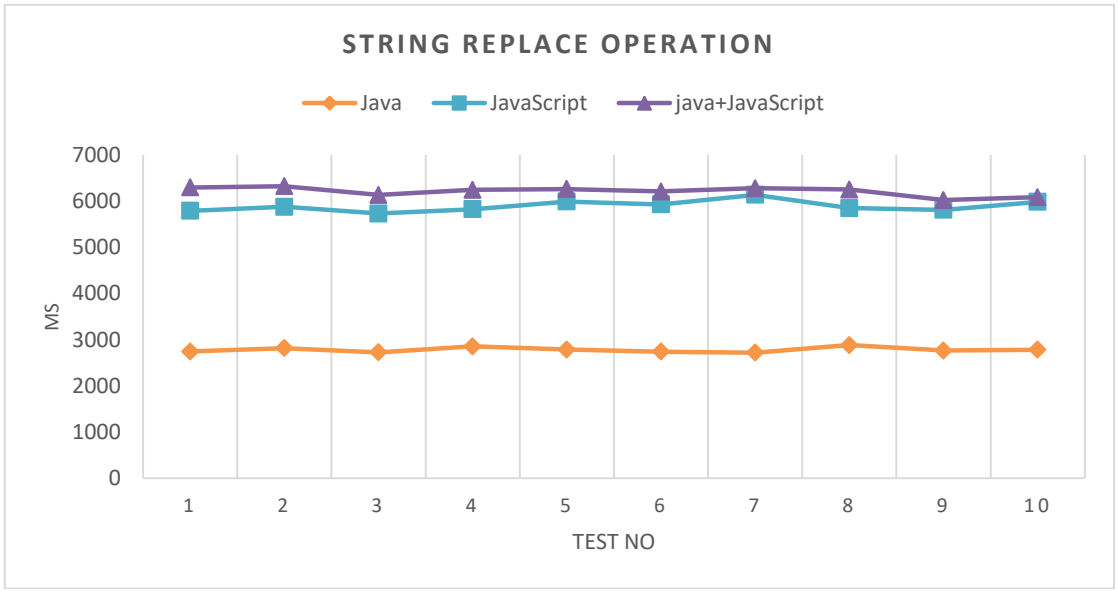

Figure 12:
Comparison of the String replace operation

4.2. Case 2

In this case, sum of the numbers is calculated through "java function in the library executed by the JavaScript" and "JavaScript function". Since there is no object or compiled function for this problem, it allows to evaluating performance of this approach. A function named "sum" is defined in the UtilityFunction as depicted in Figure 13. It calculates the sum of the 100.000.000 numbers. This operation is completed on the average in 7.8 milliseconds. The function is called by the JavaScript function as depicted in Figure 14. This operation is completed on the average in 51.7 milliseconds. JavaScript function as shown in Figure 15 completes the same problem on the average in 7986 milliseconds. As depicted in Figure 16, the library function is outperforms JavaScript function. Javascript function as depicted in Figure 15 also executed in Chrome Browser on the average in 2615 milliseconds. Our test results show that fundamental functions should be implemented in a Java library.

```
public long sum() {
    long sum = 0;
    for (int i = 0; i < 100000000; i++)      sum = sum + i;
     return sum;
}
```

Figure 13:
Java function in the library

```
String script = "function evaluation(x) {"
        + " return util.getUtilityFunctions().sum();"
        + "}";
```

Figure 14:
JavaScript function utilize the library for sum function

```
String script = "function evaluation(x) {"
        + "sum = 0;"
        + "for (i =0; i<100000000;i++) "
        + "      sum = sum+i; "
        + "return sum;"
        + "}";
```

Figure 15:
JavaScript function calculates the sum of 100.000.000 number

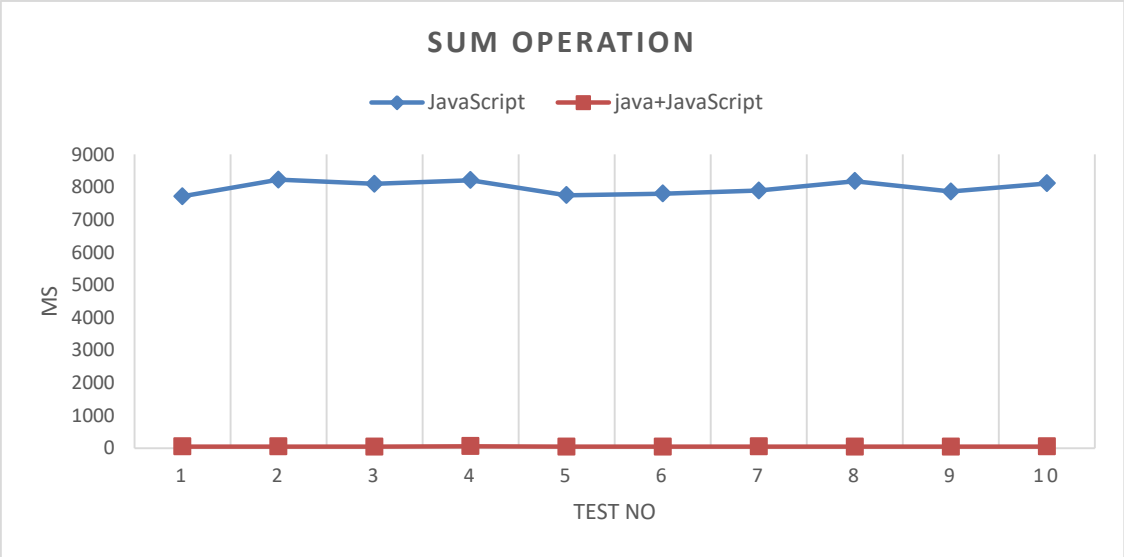

Figure 16:
Comparison of the Sum operation

5. CONCLUSION

Devices produce a tremendous amount of data and data should be processed and saved by some services. Because of performance, scalability, interdependency, and anonymity reasons Broker utilization is one solution for Internet of Things solution. The ActiveMQ open source server is used as a broker among producers and consumers. It is expected that devices should always produce the data without awareness of the consumers. But the produced data is not always congruent for the consumers. One approach can be the utilization of the pure ActiveMQ with a service that processes data and resends to consumers. Service should subscribe to all topics and resend them to a different topic which consumers know. One advantage of this approach is that there is no implementation on the ActiveMQ. There are four fundamental disadvantages; 1) consumers should subscribe to resend topics 2) messages travel one more node 3) service should have the store and forward capability for persistent and durable subscribers 4) there is extra latency for near real-time solutions.

Our tests show that Java functions have better performance than the JavaScript through Rhino Engine. If there is a fixed function applied to the message, processing mechanism should be implemented directly in the ActiveMQ. However, in this approach, modification, insertion, or deletion of the functions require compile and deploy operations.

In this paper, we proposed a new mechanism for processing the data through Rhino JavaScript engine in the ActiveMQ. When a message is delivered to the broker and if there is a defined JavaScript function for a specific message type, message payload is processed by the function before delivering to consumer. The JavaScript functions can be modified/inserted/deleted during runtime. Fundamental classes are implemented in Java environment and they can be used as a host objects in JavaScript. Our tests show that native JavaScript objects perform nearly as host objects and functions implemented in the library. Also, our library functions are outperformed than native JavaScript. Since JavaScript functions are interpreted during runtime; they should be as short as possible and they should utilize the java functions in the library.

ACKNOWLEDGEMENT

These results preliminary study of the project proposal applied to TUBITAK 1505 University Industry Collaboration Grant Program and the study is supported by EMKO Electronic A.Ş located in Bursa, Turkey.

REFERENCES

1. ActiveMQ, (2003). Apache ActiveMQ. Access address: http://activemq.apache.org (Accessed in: 01.05.2017)

2. AMQP, (2014), Standard OASIS Advanced Message Queuing Protocol AMQP Version 1.0 Access address: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=amqp (Accessed in: 01.05.2017)

3. Ionescu, V. M. (2015). The analysis of the performance of RabbitMQ and ActiveMQ. In 2015 14th RoEduNet International Conference - Networking in Education and Research, RoEduNet NER 2015 - Proceedings (pp. 132–137). Craiova Romania. http://doi.org/10.1109/RoEduNet.2015.7311982

4. Klein, A. F., ŞtefĂnescu, M., Saied, A., Swakhoven, K. (2015). An experimental comparison of ActiveMQ and OpenMQ brokers in asynchronous cloud environment. 2015 5th International Conference on Digital Information Processing and Communications, ICDIPC 2015, 24–30. http://doi.org/10.1109/ICDIPC.2015.7323001

5. MQTT, (2015), MQTT Version 3.1.1. Access address: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html, (Accessed in: 01.05.2017)

6. Rhino, (1997). Mozilla Rhino. Access address: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino (Accessed in: 01.05.2017)

7. STOMP (2012), The Simple Text Oriented Messaging Protocol 1.2. Access address: http://stomp.github.io (Accessed in: 01.05.2017)

8. Vinoski, S. (2006). Scripting JAX-WS. IEEE Internet Computing, 10(3), 91–94. http://doi.org/10.1109/MIC.2006.65