Things Object Notation as a Communicational Light-weighted Language for IoT Devices

Khazam A. Alhamdan, Mohammad A. Alkandari

Computer Engineering Department, Kuwait University, Kuwait
Khazam.Alhamdan@grad.ku.edu.kw, m.kandari@ku.edu.kw

Abstract

IoT interoperability is a major challenge in the emerging IoT field. This paper introduces a novel and light weighted language for the IoT devices to use. TON (Things Object Notation) is a language which was built on top of JSON (JavaScript Object Notation) to accommodate the needs of IoTs' interoperability. This research improves two aspects of JSON (1) compactness and (2) commanding. Currently, a lot of communicational pollution such as wireless data and noise emerge from the explosion of the IoTs. Thus, a minimum information/data is needed to be sent. In addition, since most practitioners are hobbyists, a one language/protocol that passes information and commands will be more convenient. Further, this study provides three use cases and shows comparison between using JSON and TON as a mean to send data and information.

Keywords: IoT, Interoperability, JSON, Object notation

1 Introduction

The objective of this research is to provide an easy and standardized language for the IoT practitioners to use. A standard language for the communication between IoTs is necessary as the diversity of devices and applications is widely spread, which in turn raises the challenge of the IoTs' interoperability [1]. Many researchers tackled the problem from different perspectives and majorly, they proposed solutions hang around the web technologies. This is a good way to start attacking the problem as the web solved similar problems of having multiple servers speak different protocols and languages, and hence the w3 standard emerged [2]. For interoperability and sending commands to IoT system using the web technologies, researchers tend to use REST [6] (REpresentational State Transfer) to implement their API and approach [3-4, 10].

Using multiple protocols and languages expected to be not very helpful for practitioners. One of the major problem with IoT interoperability and then scalability is that practitioners and hobbyists especially, those who are developing independently and do not use structured languages that is called CL (Casual Language). For an instance, a developer tests an IoT device which measures the temperature of a room, may send the row measurement data from the sensor to the monitor or the logging server without any structure or labeling [11].

Others use middleware to accomplish universality and guarantee interoperability which might be a good choice for companies and industries. However, those middleware are usually cloud based which is sometimes not a very good choice for normal users [3].

So, this study introduces TON, which is based on the well tested, trusted and structured JSON notation. TON introduces some techniques to compact the overhead data of JSON and also introduces some new features to assists the use of a single language by the developers. TON introduces the class idea to JSON to compact the information and removes the overhead data that usually common between multiple objects and still maintaining ease of use.

Our research approach was to review the available solutions and analyze them, and then extract the requirements need to be in TON, and design our solution based on the extracted requirements. Finally, validate our approach by presenting some use cases and analyze them.

In the followings, Section 2 shows some related works. In Section 3, requirements and some design choices are introduced. In Section 4, background information about JSON is shown. In Section 5, the proposed design and solution are demonstrated. In Section 6, some use cases are synthesized and analyzed. Section 7 discusses and differentiates our work from others. Finally, section 8 concludes the research paper.

2 Related Works

This section provides a review of some IoT papers and applications as well as investigates some solutions and hint of requirements to be integrated into the TON language.

*Corresponding Author: Mohammad A. Alkandari; E-mail: m.kandari@ku.edu.kw

DOI: 10.3966/160792642019122007013

2.1 Individual

IoTs can comes in many flavors, IoTs can be an individual device or a system of devices and a coalition of devices. In reference [11] Vidrascu et al. have developed an IoT device that monitors the temperature of a room and its humidity. It is easy to pass the sensors data casually using a CL language to run the system, for example having a key-value pairs sent to the monitoring system for each sensor will be enough to interpret the data and control the system. It is good to notice here the IoT device do not have much of control over the components since they are inputs components and might be controlled at the beginning of the system boot, but usually not controlled after the system is started.

2.2 Coalition

Coalition is another flavor of IoT where devices collaborate to accomplish some objectives. In reference [12] Tsiropoulou et al. suggested a framework for coalitions and how the status of the group is maintained and communicated, where a coalition needs to communicate its state to maintain a good coalition formation. This system is not very complicated in term of events, data, and functionalities and thus can make use of a TON as a communication language.

2.3 Middleware

Middleware is a common solution for interoperability and communication. In the survey [3], the authors discussed a lot of other type of IoT middleware which can be used to connect an IoT system. Some of large IoT systems can be of type real-time IoT application, and WSN (Wireless Sensor Network).

In paper [13] Mayer et al. proposed a semantic system that interacts and acts as a middleware between the user and the IoT device. Their system implemented a reasoning engine that tries to reason about data and like suggesting and discovering the correct presentation of different functionalities and services provided by an IoT device. This approach has two main problems, first they heavily depending on third parties (which manufactures the IoTs) to be very cooperative. Second, the reasoning engine cannot be efficient thanks to the wide range of functionalities and their varieties as different IoTs can provide, which what we think is the sematic of functionalities should be taken as is and not try to reasoning them.

In paper [14], Datta et al. developed a system that is set in the middle of IoTs and users provide sematic information to the user and some discovery services. They have used SenML (Sensor Markup Language) which is a descriptive language (like HTML for web pages and XML for data) to provide and polish information collected from sensors by adding some information to the sensed data to make it more semantically understandable by other devices. So, they provide a good structuring for data, and also, they have developed a way to manage a non-smart device that are not IoTs in concept, they might be passive sensors like light sensors.

SenML is developed to carry information for the devices with limited resources. In reference [15] Su et al. introduced how one can transform a SenML data into RDF (Resource Description Framework) model. This means the data carrier can be elevated from a simple inferior representation into a superior representation easily. RDF is known to be constructed of three main components that make a statement which is subject, object, and predicate. Subject is the data, object is the information and the semantic of it, and predicate is the relation that connects the subject with the object [7] (e.g. subject = "earth", object = "solar system", predicate = "in", thus, <earth, in, solar system>). RDF usually carried on top of other data representation structure such as JSON [7]. Our research is to describe TON which is based on JSON, so as a biproduct TON can carry the RDF semantic too.

2.4 Semantics

Semantics is the way IoTs deliver meanings of their functionalities, state, and information. In paper [4] Kiljander et al. have discussed multiple models for interoperability and they have divided the interoperability into two levels simplifying and compacting the model by connectivity and semantic levels. While other researchers like Tolk et al. in [16] suggested six different levels which include syntactic and semantic levels, and Pantsar-Syväniemi et al. included semantic too [5]. Those emphasis the importance of the semantic problem and its interpretation. Lastly, Lappeteläinen et al. differentiate semantic and information levels to be different and distinct challenges [17].

In [18], Maarala et al. described a methodology to aggregate the data collected from IoT devices and storing them semantically using RDF model concept to establish the semantic relation in the data collected.

2.5 ARM (Architecture Reference Model) of IoT

To establish a good understanding of the possible communication between IoTs, one may look at the ARM of IoT and what is the state of the art of the IoT field. There are a lot of ARMs of IoT, a good and general ARM is provided by Bassi et al. [8]. This model describes the different relations and communications between a user and an IoT device. Their model will allow us to capture the information passed back and forth between IoTs. Therfore, This paper can define concretely the requirements needed to be in mind when we develop TON.

3 Requirements

This section presents the requirements, which were derived from previous research studies, to be used to develop TON. TON should capture the following requirements:

(1) *Formality*: information passed should be interpreted the same for any interpreter (e.g. an IoT). That is if information describes a service of tuning a light intensity with a value from 0 to 10, should not be interpreted as something else, like turning on/off the light.

(2) *Completeness*: any information need to be sent can be fully sent in a single TON object. That is, a user does not need to describe a service in multiple TON, but TON can encapsulate the information.

(3) *Classifiably*: a TON object can be classified as Request, Notification, or Actuation (those cases based on [8]). That is, a user can request information (e.g. sensor data), receive notification data (e.g. real time temperature data), or perform an action via an actuator (e.g. turn off the light). This can be achieved by adding a label to the TON object.

(4) *None-Prior Knowledge*: no prior knowledge is required for the two communicators to know each other in order to make sense of TON.

(5) *Compactness*: to have the size small as possible.

There are other requirements that users do not want to emphasize much since they can be added by the TON user and not enforced by the language and they are:

(6) *Identifiability (optional)*: an IoT may add identification information to the TON to identify itself.

(7) *Description (optional)*: an IoT may add a description to the TON as a metadata to describe itself. This can be handy if the IoT is used by a human or smart entity so they can make meaning of the IoT from its description. An example is an IoT light that give itself a description of being light, and then if a controller device (e.g. smartphone) displayed the description the end user (human), he can interact soundly with the IoT without a prior knowledge of the smartphone application nor the IoT implementation. An analogy is a group of applicants (IoTs) send their CVs (TONs) to an employer (smartphone), and without a prior knowledge, the employer can interact with the applicants properly and efficiently.

(8) *Greeting*: a greeting object to retrieve public service information that can be provided by an IoT. This one can be implemented by the developer on the first interaction between the IoT and the controller device.

4 Background

The base language that is used to build our TON upon is JSON. So, this section presents the basic building blocks of JSON and how a JSON object is constructed.

JSON object is constructed as a hierarchy of key-value pairs. (See Figure 1). Every JSON object starts with a left curly bracket and end with right curly bracket to enclose the data carried inside the JSON object.

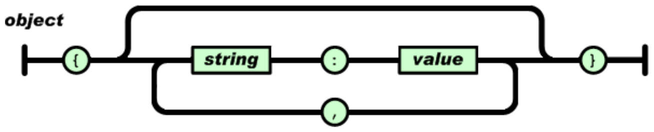

Figure 1. This figure is captured from the JSON documentation [9] - JSON object representation

A key used in JSON is a string of characters enclosed in double quotes character. Some special characters represented as the scape character then that special character or another character represent it. Examples for special characters are: new line, tab, null, double quotes, and a Unicode character and they can be represented as (\n, \t, \0, \”, and \u#### with 4 hexadecimal digit number) (see Figure 2).

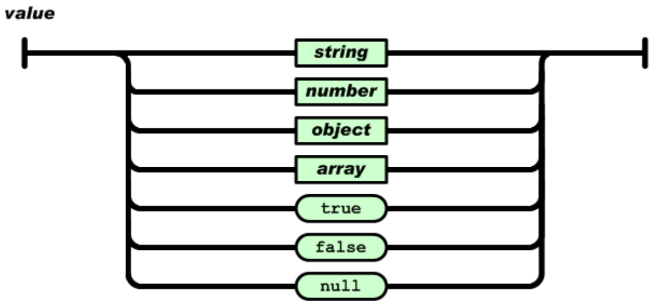

Figure 2. This figure is captured from the JSON documentation [9] - JSON value representation

A value in JSON is either a string, keyword (true, false, null), array of values, number, or object (see Figure 3). An array of values is a set of values enclosed by square brackets and separated by commas.

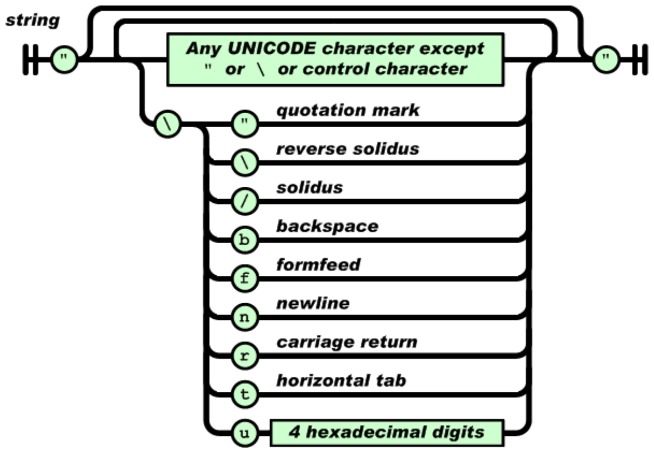

Figure 3. This figure is captured from the JSON documentation [9] - JSON String representation

5 Design & Proposed Solution

This section introduces the TON language and how it is developed from the requirements in the previous section. Moreover, this section argues about the satisfaction of those requirements.

5.1 Syntax

Generally, languages like JSON and XML are formal in their syntax, so considering one of them would not be a bad choice for formality requirement.

JSON uses curly brackets to indicate an object where objects can be nested. JSON is based on the idea of key-value data structure so an object will contain a key "K" and that key will be associated with a value, which can be a string, JSON object, array, number, Boolean (true, false) or noting (represented as null). Further, array is indicated by square brackets and it contains values (the same as the values mentioned previously). These specifications allow JSON to represent any data without losing the formality (see [9] for more details).

An example for a JSON object is shown in Figure 4. A JSON object carries information of a car with four wheels, colored scarlet and other specifications.

```
{
  "car": {
    "wheels" : ["front right","front left", "rear right","rear left"],
    "doors" : ["front right","front left", "rear right","rear left"],
    "roof" : true,
    "convertible": false,
    "color" : [145,13,9],
    "color-name" : "scarlet",
    "engine-size" : [5.2, "liter"],
    "engine-cylinders": 6,
    "weight": [1700,"Kg"]
  }
}
```

Figure 4. JSON example for a car object specification

On the other hand, XML can represent the same data using tags. XML as a markup language uses tags to specify the parents' objects and the child ones. So, for XML to represent our car object, it will create a tag <car> then add the other keys (i.e. <wheels>, <doors>, … etc.) enclosed by the car tag, and it finishes after the termination tag </car> (see Figure 5).

```
<root>
   <car>
      <color>
         <element>145</element>
         <element>13</element>
         <element>9</element>
      </color>
      <color-name>scarlet</color-name>
      <convertible>false</convertible>
      <doors>
         <element>front right</element>
         <element>front left</element>
         <element>rear right</element>
         <element>rear left</element>
      </doors>
      <engine-cylinders>6</engine-cylinders>
      <engine-size>
         <element>5.2</element>
         <element>liter</element>
      </engine-size>
      <roof>true</roof>
      <weight>
         <element>1700</element>
         <element>Kg</element>
      </weight>
      <wheels>
         <element>front right</element>
         <element>front left</element>
         <element>rear right</element>
         <element>rear left</element>
      </wheels>
   </car>
</root>
```

Figure 5. XML example for a car object specification

You can immediately observe that the XML language most of the time will have a size greater than the JSON object, for the same object specifications. So, we will go with something similar to JSON as we need to reduce those data overhead. Hence, the TON language is proposed and built based on JSON syntax.

5.2 Formality & Completeness

Using JSON model as it is, gives us Formality and Completeness, where all recipients will interpret the data the same with the same semantic, and all data can be sent as one object (object can contain nested objects). So let's define our structure, TON object is always surrounded by curly brackets (i.e. "{"for the beginning of TON, and"}" for the end of TON), and also, TON embedded JSON inside it. This ensures Formality and Completeness.

So, the proposed language is structured as in Figure 6. The figure demonstrates three possible forward flow of constructing a TON object, a path with [sub-JSON], a path starts with [label], and a path starts with a hash (#) which is the defining path. The sub-JSON path is exactly a JSON structure without the curly brackets in the beginning and in the end (i.e. if a JSON is ({"x" :{"num" :76}}), then sub-JSON is only ("x" :{"num" :76})), that is because TON already got a beginning and ending curly brackets.

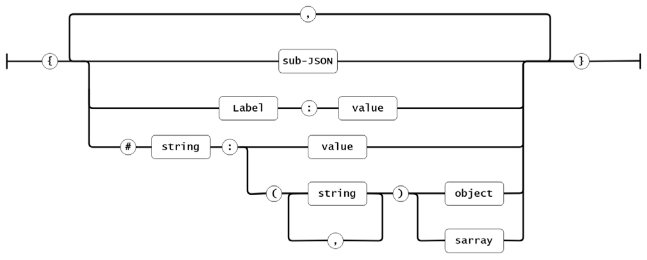

Figure 6. TON structure

The second path, the path of label is a novel addition to JSON so it can have some short-cut and fast executing for some IoTs functionalities such as Requesting data from an IoT and order an IoT to

Actuate an actuator to do something. Other labels can be introduced as the model is flexible. An example of label usage is shown in Figure 7.a, for requesting the temperature from an IoT device, and in Figure 7.b is an actuation order to some device with ID=107. More labels will be discussed in the sections follows.

```
{Request:"Temprature"}
```

(a) shows an example for TON using a label Request to request the Temperature data

```
{Actuate:{"ID":107,
          "Temprature":30,
          "unit":"C",
          "Dry":false
         }
}
```

(b) shows TON using a label Actuate to send the input data required to set the device with ID 107 to 30 degree Celsius and Dry mode being off

Figure 7.

The third path which starts with (#) is the path of defining classes, which allow some sort of objects like those of Object-oriented programming languages. However, the hash (#) is used to define a class for the set of data that are common between different IoTs, and thus can be gathered to minimize the size of the object need to be sent through the network. This novel way is added upon JSON. To define something, a user can start with a hash # and then a string that is representative of the thing he/she is defining, and then the user can come into a fork of two, either using a value directly (a value is defined in the next section) or having a set of inputs to initialize the object/array this definition is building. An example of defining lamp object is illustrated in Figure 8. The TON shown in Figure 8 shows how a lamp is defined and used by other key-value pairs.

```
{#"Lamp":{"Description":"LED",
          "Color":"White",
          "Temprature":50,
          "Temprature-unit":"C"
         },
  "L1":&"Lamp",
  "L2":&"Lamp"
}
```

Figure 8. TON defining of a class and using it. L1 and L2 are now two Lamps with description LED, color White, and Temperature of 50C

Another example for lamps definition is shown in Figure 9, where here the Lamp class takes inputs to initialize some values inside the object. The order of the input at the definition does not need to match the order they are inside the object; the matching is based on the keys inside the class. However, the order should match when using the class as a value for some key. This represents the second fork of the defining path in TON.

```
{#"Lamp"("Color","Description"):{
         "Description":"LED",
         "Color":"White",
         "Temprature":50,
         "Temprature-unit":"C"
        },
  "L1":&"Lamp"("Red","LED"),
  "L2":&"Lamp"("Yallow","Neon"),
  "L3":&"Lamp",
  "L4":{"Name":"Living Room",
        "Light":&"Lamp"("White","LED")
       }
}
```

Figure 9. TON defining of a class and using it. L3 is now a Lamp with description LED, color White, and Temperature of 50C, L1 is a Red LED, L2 is a Yellow Neon, and L4 is an object which encapsulate a lamp

By using the labels we satisfy the classifying requirement and compactness requirement by introducing the define functionality to the language.

5.3 Values

First, sarray is an array of strings, and its structure is shown in Figure 10. This was introduced just to simplify the TON diagram.

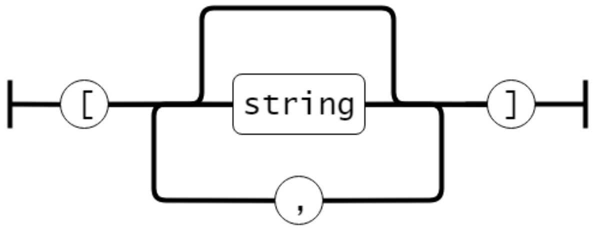

Figure 10. Sarray, an array of string

Second, Label component, which is a set of reserved words used to classify and categorize TONs by operation. There are currently three labels Request, Actuate, and Notification (other labels might be added as necessary in the future) [8]. Those labels are not contained in a double-quote like strings but used as is (see Figure 7).

Third, object component, which can be any TON object. This allows TON objects to be nested to carry more sophisticated information about the IoT and the IoT system.

Fourth, value component, which is the same value component of JSON but with some extra types of values. For instance, the classes are not defined in

JSON, so in order for TON to use them, a value that refer to a class should be integrated with JSON value (see Figure 11).

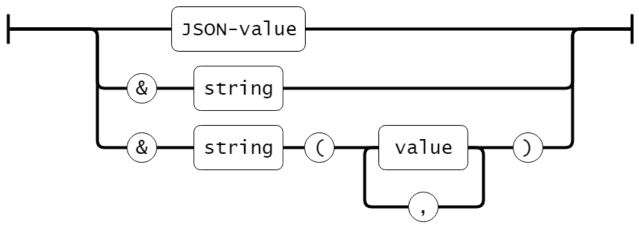

Figure 11. TON value, JSON-value is the value structure of JSON, the second path is using the default construction of a class, the third path is using a class with input values

6 Validation

In our validation, we will show some expected theoretical analysis, and we are going to compare our construction of TON against JSON using some use cases and see how much improvement we can be achieved.

6.1 Use cases Comparisons

In this part of the validation section, TON, and JSON were constructed for synthesized use cases. The first use case will be a comparison between JSON and TON for a simple IoT device. For the second use case, the two contexts are compared, (1) direct communication with the IoTs, where the user talks and command the IoT without any middleware, (2) the user talks to a middleware who talks and manage the IoT system. The third use case is a coalition of IoTs communicating with each other.

6.1.1 Room Monitor IoT

A simple IoT that monitor a room humidity, temperature, time, and aware of its power supply (see Figure 12) a bit similar to the work done in [11]. The device collects the humidity, and temperature data of a room with some time stamp. To retrieve the data appropriately, the IoT device will encapsulate the data in some data format like JSON, or XML. So here we are going to compare JSON against TON.

The data retrieved from this device is the temperature, humidity, power supply, and the time-stamp of each reading. Moreover, these can be represented in JSON as shown in Figure 13. Figure 14 shows how the data can be represented in TON. In this example, we are assuming that three readings will be returned.

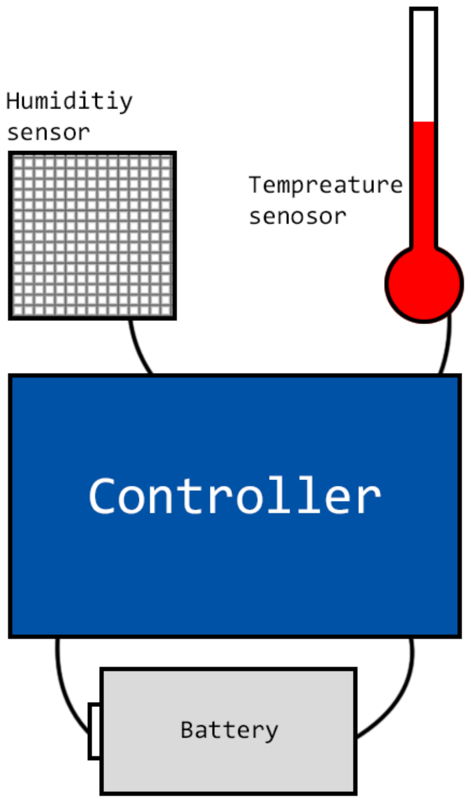

Figure 12. a simple IoT device that monitor the humidity and temperature of a room

```
{
  "T": [{"timestamp": "1/1/2018 09:00",
       "degree":21,
       "unit":"C"},
       {"timestamp": "1/1/2018 09:10",
       "degree":21.5,
       "unit":"C"},
       {"timestamp": "1/1/2018 09:20",
       "degree":22,
       "unit":"C"},
       {"timestamp": "1/1/2018 09:30",
       "degree":22.5,
       "unit":"C"},
       {"timestamp": "1/1/2018 09:40",
       "degree":22,
       "unit":"C"},
       {"timestamp": "1/1/2018 09:50",
       "degree":22,
       "unit":"C"},
       {"timestamp": "1/1/2018 10:00",
       "degree":20,
       "unit":"C"},
       {"timestamp": "1/1/2018 10:10",
       "degree":20,
       "unit":"C"}]
}
```

Figure 13. Data encapsulated in JSON

```
{ #"s":("degree","timestamp"){
    "timestamp": "1/1/2018 00:00"
    "degree":21,
    "unit":"C"
  },

  "T": [&"s"(21,"1/1/2018 09:00"),
    &"s"(21.5,"1/1/2018 09:10"),
    &"s"(22,"1/1/2018 09:20"),
    &"s"(22.5,"1/1/2018 09:30"),
    &"s"(22,"1/1/2018 09:40"),
    &"s"(22,"1/1/2018 09:50"),
    &"s"(20,"1/1/2018 10:00"),
    &"s"(20,"1/1/2018 10:10")]
}
```

Figure 14. data encapsulated with TON

As seen, a lot of overhead can be eliminated when using the introduced defining operation in TON. If we counted the number of characters for TON and JSON, we will find them 309 and 452 respectively (excluding the whitespaces). Further, if we focus on the dynamic part of the object (array contents) and calculated the eliminated percentage, we will have 52% eliminated (52% saves).

6.1.2 Smart House

A smart-house-holder has a room with multiple smart IoT devices such as TV, Lamp, ceil lamps, A/C, and a Heater (see Figure 15). A user will need to communicate with those IoTs to retrieve data and adjust them. So, we will show how those data can be communicated using two scenarios, one with direct communication with the IoTs, and the other one is with a middleware/middle-IoT.

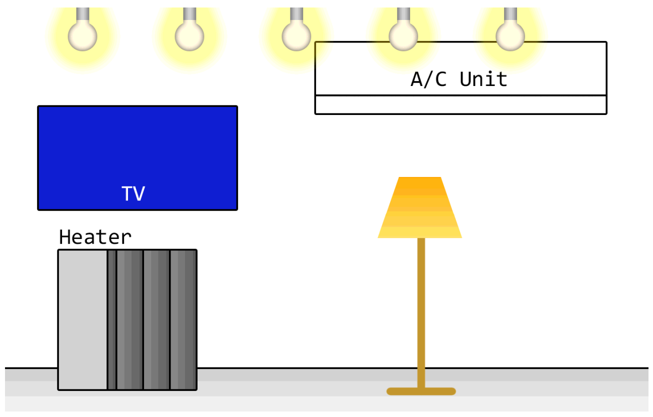

Figure 15. A sample smart house and what devices can be made smart

The smart house has a smart TV, A/C Heater, Lamp, and a set of ceil lamps. Two scenarios were demonstrated, (1) the user communicates with each individual, (2) there would be a middleware which will manage the IoTs.

The scenario of having a middleware has been analyzed, in which the user will communicate with rather than every individual IoT. The TON object shown in Figure 16 encapsulates all the information of the whole IoT system provided by the middleware. The JSON object was not included because it was easy to construct and it could take a huge portion of the page.

```
{ #"Lamp":("intensity","color","type","working"){
    "intensity":10,
    "intensity-unit":"lx",
    "color":"white",
    "type":"LED",
    "working":true,
    "action":["ON/OFF","tune"]
  },

  "TV":{
    "POWER":false,
    "Channel":0,
    "Channel-name":null,
    "volume":0,
    "action":["ON/OFF","tune"]
  }

  "AC":{
    "POWER":true,
    "Dry":false,
    "degree":23,
    "degree-unit":"C",
    "action":["ON/OFF","tune"]
  }

  "Heater":{
    "POWER":false,
    "degree":0,
    "unit":"C",
    "action":["ON/OFF","tune"]
  }

  "Floor-Lamp":&"Lamp"(19,"white","LED",true),
  "Ceil-Lamp":[&"Lamp"(10,"white","LED",true),
  &"Lamp"(10,"white","LED",true),
  &"Lamp"(10,"white","LED",true),
  &"Lamp"(10,"white","LED",true),
  &"Lamp"(10,"white","LED",true)]
}
```

Figure 16. Smart house middleware TON object

6.1.3 System of Drones

A coalition/swarm of drones (treated as IoTs) and how the data between them are passed (see Figure 17). We will look at this use case not as a user perspective, but as the drone leader perspective. In Figure 18, we show how drones information encapsulated, in Figure 19 the leader order some recipient drone to move to some location (x,y) via labeling the TON object as Actuate object.

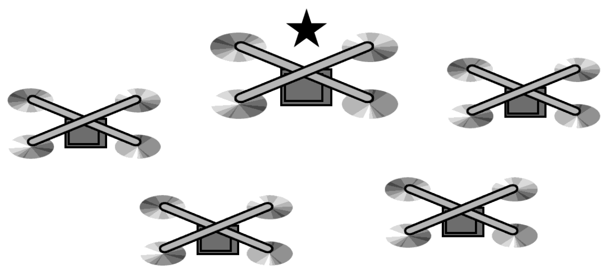

Figure 17. A system of drones, and the drone with star is the leader

```
{ "location":[0,0],
  "altitude":100,
  "lenght-unit":"m",
  "POWER":true,
  "action":["change location","chage altitude","ON/OFF"]
}
```

Figure 18. Drone information in TON and JSON

```
{Actuate:{"change location",[10,9]}}
```

Figure 19. Drone leader message encapsulated in TON to order a drone to move to the location (10,9) using the Actuate label of TON

6.2 Theoretical Comparison

It is easy to notice that TON is better than JSON in compacting data, as TON encapsulate JSON, which obviously gives TON the JSON power plus whatever tools we are designing for it (i.e. classes and labels). So, TON is either as good as JSON or better.

Tshe improvement gained by reducing the overhead repeated strings (keys) via introducing the classes and defining can be calculated as the following:

$$JSON(N) = N \cdot \sum key + value \quad (1)$$

$$TON(N) = N \cdot value' + \sum key + value$$

$$R(N) = \frac{JSON(N) - TON(N)}{JSON(N)}$$

$$= 1 - \frac{N \cdot value' + \sum key + value}{N \cdot \sum key + value}$$

$$= 1 - \frac{N \cdot value'}{N \cdot \sum key + value} - \frac{\sum key + value}{N \cdot \sum key + value} \quad (2)$$

$$R(N) = C - \frac{1}{N} \quad (3)$$

Where N is the number of objects sharing the same properties, key-vlaue pairs are the key and value pairs which define properties in a JSON object, value’ is the values needed to initialize an object in TON, and R(N) is the ratio saved of TON(N) from JSON(N) of N objects. C is a constant that represents the constants found in the last equation.

If R(N) is positive then defining a class and using it in TON will reduce the size of the overall object. If it is negative, this means it requires more space than a regular JSON. We can also see that the ratio of value’ to the sum of all key-value pairs can be negligibly small to 1. Thus, C can be safely assumed to be 1.

$$R(N) = 1 - \frac{1}{N} \quad (4)$$

Now we can see clearly that the amount saved by using classes is inversely proportional to the number of objects created (N). In another word, the size of JSON is linearly increased with respect to TON’s size.

6.3 Theoretical Growth

In this subsection, a general formula is derived, which describes how TON’s size grows against JSON.

A room is a set of objects $\mathbb{B}$, we can define a class for any given object, and thus the set of all classes is $\mathbb{C}$. A subset β of $\mathbb{B}$ is the set of objects that belongs to the same class $c_i \in \mathbb{C}$. Now, we can calculate the size used by TON S($\mathbb{B}$) as:

$$S(\mathbb{B}) = \sum_{i=0}^{\infty} |\beta_i| \cdot Size(c_i) \quad (5)$$

$$S(\mathbb{B}) = S(N, n_0, ..., n_{N-1}, t_0, ..., t_{N-1}) = \sum_{i=0}^{N-1} n_i \cdot t_i \quad (6)$$

Which goes through every possible class $c_i \in \mathbb{C}$. However, we can reduce and limit those classes to only the needed ones (e.g. if a room does not have a TV object, we do not need a class to represent TVs) $\mathbb{C}' \subseteq \mathbb{C}$.

Let $|\mathbb{C}'| = N$, $|\beta_i| = n_i$, and $Size(c_i) = t_i$ which represent the size needed to define an object of class c_i.

As if we want to plot how the function behave there will be exponentially may possible value S can have, so the Figure 20 represents the domain of which $n_i \cdot t_i$ can be picked from.

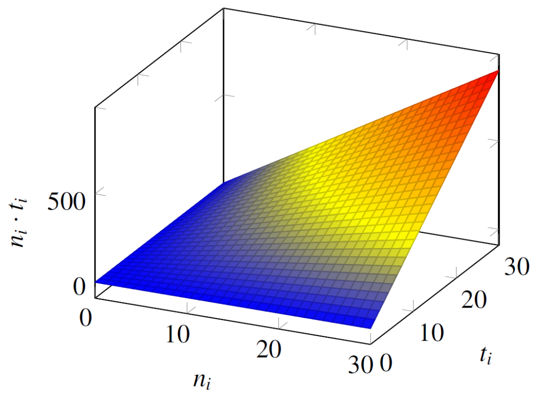

Figure 20. The domain of which S($\mathbb{B}$) takes the summation values from

The domain of TON sum can be compared with the domain of the equivalent JSON sum using the relation R(N') as follows (Notice N' of R(N') is number of objects, while the latter N is number of classes):

If we would draw the JSON summation values domain we will get Figure 21.

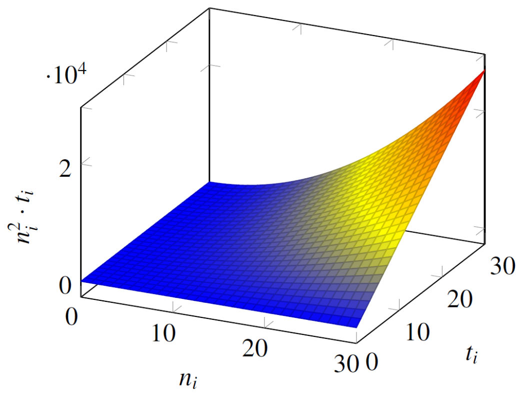

Figure 21. The domain of which JSON (N') takes the summation values from

We can see from equation (6) and (7) that JSON grows quadratically in comparison with TON which grows linearly with respect to number of objects in the set $\mathbb{B}$.

6.5 Experimental Validation

In this sub-section we experimentally compare TON to JSON and in an experiment that mimic a real scenario. We will assume an IoT that measures the temperature of a room and create an object to store it along with timestamp and the temperature unit (e.g. see Figure 13 and Figure 14).

The temperature data used in the experiment is randomly generated to be in the range from 20.0 to 24.9 inclusive. The temperatures were measured each minute, then we have generated 2 days' worth of data, then 7 days then 12 days, adding 5 till day 57 inclusive.

After that the JSON and TON objects were compressed using gzip compression (as gzip is the HTTP common practice for compression). The following Table 1 shows the different datapoint we have found:

This experiment yielded the following:

Table 1. TON and JSON data sizes in KB after compression, and the slope of their growth

Days	TON.gzip	JSON.gzip	TON slope	JSON slope
2	10	11	0	0
7	32	38	4.4	5.4
12	53	65	4.2	5.4
17	74	92	4.2	5.4
22	96	120	4.4	5.6
27	117	147	4.2	5.4
32	138	174	4.2	5.4
37	160	201	4.4	5.4
42	181	228	4.2	5.4
47	203	255	4.4	5.4
52	224	282	4.2	5.4
57	245	310	4.2	5.6

(1) JSON and TON became so light that the slope difference between TON growth size and JSON growth size become less.

(2) TON performance surpassed JSON by having smaller growth slope than JSON.

(3) Without compression JSON objects have twice the size of their correspondent TON objects (see Figure 22).

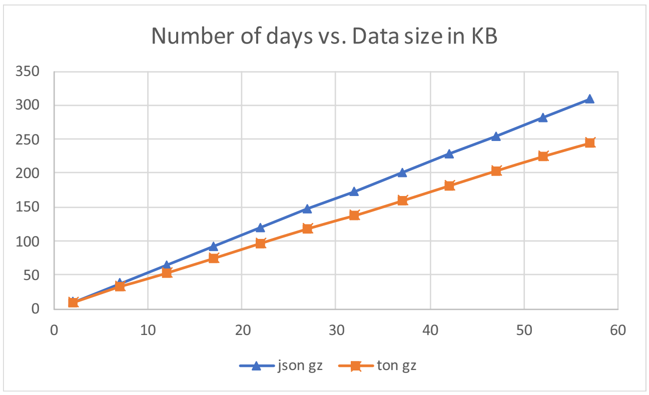

Figure 22. The linear growth of JSON and TON both compressed using gzip vs. number of days

6.5 Summary

We can notice in Table 2 that the data size saved increases as the number of repeated structures appears. For an instance, the number of lamps in the smart house allow TON to save 32.7% of the total size of a normal JSON object in spite of the verity of the IoT devices (TV, AC, ... etc.). And when we removed the static information and concentrated on the dynamic ones (the array) in use case 1, we managed to save 52% of the array size. This achievement occurs because of using the classes defined in TON which removes the extra keys that are just overhead that can be implied by introducing the data order to JSON (since JSON does not care about data ordering, it cares about the key-value concept only).

Table 2. Data size of different use case scenarios

Use case	TON	JSON	$\frac{JSON - TON}{JSON} \times 100$
Case 1/full size	309	452	31.6%
Case 1/array size	214	446	52%
Case 2/full size	633	941	32.7%
Case 3/full size	The same	The same	0%

7 Discussion

To differentiate our work from others, in this section, we will discuss different approaches to carry information around in the IoT field.

In reference [13] Mayer et al. had developed a middleware that allow IoTs to communicate with each other. Their middleware is built to use REST principles to carry the commands from a device to another. The semantic carrier they've used is RDF which allows them to pass information semantically between IoTs. The thing is that we tackle where their model cannot, is passing information with and without semantic. As they are using RDF model, they need to have a semantic database to eventually give semantics to the data. So, if there semantic is not defined in the database, the system cannot carry the information as required by the model.

In reference [14] Datta et al. they developed a middleware device that connect IoTs, smart devices, none-smart devices, and smartphones with each other. Their idea is that devices provide their functionalities and semantics upon registration in the middleware, so afterward whenever a smartphone need to communicate with IoTs it requests the semantic data from a local database inside the middleware so then they can communicate with the IoT system. The drawback of the system is that, they use a limited language to represent data. They have used SenML to represent sensors data to pass them to the middleware and also they have used the same language to communicate with the actuators. SenML is designed to be a light-weighted language and specially to deliver sensors data not to deliver command data. Not only our TON language can encapsulate SenML, but also uses labels to deliver data and commands easily and designed to do such job.

8 Conclusion

In this paper, we introduced new concepts to be adapted and developed upon JSON. First one is compactness, where overhead data was reduced by defining a class and use it as a template for other objects. Second contribution is to give TON the ability to carry commands from a device to another by labeling the TON object with Actuate, Request, and Notification. Other labels can be implemented.

Our experiments shown a remarkable improvement on the compressed size of the data need to be sent where the growth's slope was reduced from 5.4 to 4.3, by around 20% of the corresponding compressed JSON size.

For future work, we are thinking to study more IoT applications and see what kind of labels we can add to the language. Introducing more structures to TON similar to programming languages such as loops, which might be good for reducing the size of data even more. An example for a good use of loops is sending a musical note for an IoT that works as a musical instrument. Other future work we think worth of investigation is the wireless data pollution caused by the increase amount of data transferred by devices.

References

[1] F. Zambonelli, Key Abstractions for Iot-Oriented Software Engineering, *IEEE Software*, Vol. 34, No. 1, pp. 38-45, January-February, 2017.

[2] D. Zeng, S. Guo, Z. Cheng, The Web of Things: A Survey, *Journal of Communications*, Vol. 6, No. 6, pp. 424-438, September, 2011.

[3] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, S. Clarke, Middleware for Internet of Things: A Survey, *IEEE Internet of Things Journal*, Vol. 3, No. 1, pp. 70-95, February, 2016.

[4] J. Kiljander, A. D'elia, F. Morandi, P. Hyttinen, J. Takalo-Mattila, A. Ylisaukko-Oja, J. P. Soininen, T. S. Cinotti, Semantic Interoperability Architecture for Pervasive Computing and Internet of Things, *IEEE Access*, 2014, Vol. 2, pp. 856-873, August, 2014.

[5] S. Pantsar-Syväniemi, A. Purhonen, E. Ovaska, J. Kuusijärvi, A. Evesti, Situation-based and Self-adaptive Applications for the Smart Environment, *Journal of Ambient Intelligence and Smart Environments*, Vol. 4, No. 6, pp. 491-516, January, 2012.

[6] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*, O'Reilly Media, Inc, 2011.

[7] G. Klyne, J. J. Carroll, B. McBride, *RDF 1.1 Concepts and Abstract Syntax*, W3C Recommendation, 2014.

[8] A. Bassi, M. Bauer, M. Fiedler, T. Kramp, R. Kranenburg, S. Lange, S. Meissner, *Enabling Things to Talk*, Springer-Verlag Berlin Heidelberg, 2013.

[9] J. ECMA, *404 the Json Data Interchange Standard*, ECMA International, 2016.

[10] D. Guinard, V. Trifa, E. Wilde, A Resource Oriented Architecture for the Web of Things, *2010 Internet of Things (IOT)*, Tokyo, Japan, 2010, pp. 1-8.

[11] M. G. Vidrascu, P. M. Svasta, Embedded Software for IOT

Bee Hive Monitoring Node, *2017 IEEE 23rd International Symposium for Design and Technology in Electronic Packaging (SIITME)*, Constanta, Romania, 2017, pp. 183-188.

[12] E. E. Tsiropoulou, S. T. Paruchuri, J. S. Baras, Interest, Energy and Physical-aware Coalition Formation and Resource Allocation in Smart IoT Applications, *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, Baltimore, MD, USA, 2017, pp. 1-6.

[13] S. Mayer, N. Inhelder, R. Verborgh, R. Van de Walle, F. Mattern, Configuration of Smart Environments Made Simple: Combining Visual Modeling with Semantic Metadata and Reasoning, *2014 International Conference on the Internet of Things (IOT)*, Cambridge, MA, USA, 2014, pp. 61-66.

[14] S. K. Datta, C. Bonnet, N. Nikaein, An IoT Gateway Centric Architecture to Provide Novel M2M Services, *2014 IEEE World Forum on Internet of Things (WF-IoT)*, Seoul, South Korea, 2014, pp. 514-519.

[15] X. Su, H. Zhang, J. Riekki, A. Keränen, J. K. Nurminen, L. Du, Connecting IoT Sensors to Knowledge-based Systems by Transforming SenML to RDF, *5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014)*, Hasselt, Belgium, 2014. pp. 215-222.

[16] A. Tolk, Composable Mission Spaces and M&S Repositories-applicability of Open Standards, *Spring Simulation Interoperability Workshop*, Crystal, VA, USA, 2004, pp. 55-68.

[17] A. Lappeteläinen, J. M. Tuupola, A. Palin, T. Eriksson, Networked Systems, Services and Information the Ultimate Digital Convergence, *1st International NoTA Conference*, Helsinki, Finland, 2008, pp. 1-7.

[18] A. I. Maarala, X. Su, J. Riekki, *Semantic Data Provisioning and Reasoning for the Internet of Things, 2014 International Conference on the Internet of Things (IOT)*, Cambridge, MA, USA, 2014, pp. 67-72.

Biographies

Khazam A. Alhamdan is a Master student in Kuwait University, College of Engineering & Petroleum, in Kuwait, where he received his bachelor degree in Computer Engineering in 2016.

Mohammad A. Alkandari is an Assistant Professor of Computer Engineering at Kuwait University, Kuwait, where he has been on the faculty since 2012. He received his Ph.D. degree in Computer Science at College of Engineering from Virginia Tech. He is a researcher in Software Engineering, and Human-Computer Interaction.