International Conference on Cloud of Things and Wearable Technologies 2018 [ICCOTWT 2018]

ISBN	978-93-88122-00-9
Website	iccotwt.com
Received	26 – May– 2018
Article ID	ICCOTWT012

VOL	01
eMail	iccotwt@asdf.res.in
Accepted	26 - June – 2018
eAID	ICCOTWT.2018.012

User Customizable IoT Systems Using Self-Aware Sensors and Self-Aware Actuators

Trusit Shah[1], S Venkatesan[2], Harsha Reddy[3], Somasundaram Ardhanareeswaran[4] and Vishwas Lokesh[5]
[1,2,3,4,5]Department of Computer Science, The University of Texas at Dallas, Richardson, Texas, USA

Abstract— *Many IoT devices have been designed, built and deployed enabling users to monitor and control their systems through the Internet. Many of these IoT devices are rigid with no easy ways for users to customize. We have designed an IoT architecture to enable IoT devices to enable user customization. Customization is performed in two ways: hardware customization and software customization. In hardware customization, a user can add/remove IoT sensors/actuators, which are made self-aware by our technique, to/from the IoT system without knowing any hardware details of those sensors/actuators. For software customization, the user can create tasks without writing any code. A self-aware sensor/actuator is a sensor/actuator with an additional processing element and related components. For software customization, we have designed a rule engine, which converts user-desired actions into computer code. We have tested this architecture in several real-life scenarios.*

INTRODUCTION

Advancements in embedded technology has led to several IoT solutions being offered to various vertical markets. These offerings have resulted in many advantages for the users. Going ahead, many more devices will become "Web enabled" and join the IoT ecosystem [1]. Current IoT solutions, offered by many different providers, typically have a proprietary system including User interfaces (both smart phone and web). The lack of standardization for the IoT products makes them rigid and limits the user's ability to customize the product. The heterogeneous environment for different applications and interoperability between different types of network protocols are some of the major challenges in standardizing IoT products [3].

Many researchers have proposed different ways to standardize IoT products. Some attempts in this effort include protocol standardization. IETF has proposed protocols such as 6LoWPAN and Co AP for the constrained devices used in the IoT environment [2]. Some cloud based IoT architectures have also been recommended by the researchers to standardize the IoT architecture using cloud-based services [4] [5]. In these solutions, the cloud service handles all the types of sensors and actuators in a standardize way. Most of these IoT architectures are more focused towards standardization of IoT systems (and not customization of the IoT system at the user level). As these architectures follow some standards, they provide flexibility to the developers to rapidly create new IoT systems.

In the current IoT systems, if the user wants to customize the software part of the IoT system, the user must reprogram that IoT system. For example, if a user currently using a proprietary system to monitor temperature at a warehouse needs to monitor humidity also, he/she would have to either buy a humidity sensor from the same provider and ask them to reprogram the IoT system supporting new sensor or buy an entirely new system.

In this paper, we describe a way to build an IoT system that is user customizable. Our architecture consists of four components: Self-aware sensors/actuators, an IoT gateway device (or IoT device), a server and a user interface. The self-aware sensor/actuator is connected to the IoT device using a known user interface such as USB, Wi-Fi or Bluetooth. The server communicates with IoT device and the user interface either via REST calls or over MQTT service. Any sensor/actuator can be converted into self-aware sensor/actuator using our methodology.

Our architecture focuses on user level customizations: if the user wants to deploy a new IoT system or make a change to an existing system, the user can do it without having any knowledge of and expertise in hardware or programming.

Our architecture enables the user to build an IoT system using few clicks from the user interface provided by us. For example, the user can setup a task such as "If the temperature is greater than 50 °F and humidity is 75%, turn on the Fan controller" and make changes to the IoT system easily whenever requirements change.

The paper is arranged in the following way. First, we discuss the related work in section 2. Section 3 covers the overall system architecture followed by hardware customization of our architecture in Section 4. In section 5, we describe the software customization. Section 6 describes the implementation details of the architecture. Section 7 presents a performance analysis of our architecture.

PREVIOUS WORK

There have been several ways that researchers have conceptualized the idea of user customizable IoT architecture. Kortuem, G. et al. discuss the awareness of smart IoT objects [17]. In that paper, the authors describe how a normal sensor or actuator can be aware of its surrounding and can perform tasks according to the environmental change. They have presented ways using which a developer can pre-program the IoT sensors/actuators with environmental constraints. The user doesn't have any control over customization of an IoT device.

Several approaches have been presented on the unification of IoT devices. JADE architecture is an example of one such architecture where the developer can configure and customize the IoT service [18]. JADE provides an easy way to create an IoT device using their framework. The developer needs to write some code to define the IoT system and JADE creates the IoT system from the defined script. A restful service is created by Stirbu et. al. [19] for unified discovery, monitoring and controlling of smart things. Sarar et. al [21] have introduced an IoT architecture with virtual object layer. This virtual object layer is responsible for unifying heterogeneous IoT hardware. Alam et. al [22] have proposed an architecture named SenaaS, which creates a virtual layer of IoT on the cloud. Here, IoT sensors are considered as sensor as a service and it hides all the hardware specific details of sensor to the user. Kiran et. al [23] have designed a rule based IoT system for remote healthcare application. They have created a virtual software layer to execute rules on the sensor values. As their work focuses only on a single application (healthcare), they don't have any virtualization on hardware. There has been similar work done on rule-based IoT systems. An If-Then based rule implementation architecture is explained by Zeng, D. et. alb [20]. The authors discuss the user configurable triggers for IoT systems.

Popular cloud services such as Amazon AWS, IBM Watson, ATT M2x have created a sensor as service cloud platform for IoT systems [15, 16]. These architectures are customizable at developer level. A user has the ability to configure few thresholds but the user cannot customize full IoT System.

SYSTEM ARCHITECTURE

Researchers have proposed different ways to develop an IoT architecture. These architectures can be classified into two types. In the first type, the IoT sensors/actuators directly communicates with the server over the internet [12]. In the second type, all the sensors/actuators are connected to a gateway device and that gateway device communicates with the server using a WAN interface [6,7,8,9]. The first type is suitable for the application where the number of sensors and actuators are low and/or when the network connectivity is poor.

We have designed two different architectures to demonstrate user customizable IoT system: IoT gateway-based architecture and standalone self-aware architecture. The IoT gateway-based architecture has four main components: One or more self-aware sensor(s)/actuator(s), an IoT device, a server (on the cloud) and a user interface. Fig. 1 shows the different components and their interactions.

Each self-aware sensor/actuator consists of a sensor/actuator with an additional processing element and related support components. The processing element stores basic details about the sensor/actuator, such as the id of sensor/actuator, type of sensor/actuator, parameters of sensor/actuator, etc. When the self-aware device is connected to the IoT device, all the details of sensor/actuator is communicated to the IoT device. The IoT device uploads these details to the server and obtains further instructions from the server on how to handle the newly connected sensor/actuator. Any number of self-aware sensors/actuators can be connected to the IoT device. The user can monitor all the connected sensors/actuators through the user interface and create tasks for those connected sensors/actuators.

The standalone self-aware architecture doesn't have the IoT gateway. The self-aware device (which can be either a self-aware sensor or a self-aware actuator) directly communicates with the server. The self-aware device must contain a network interface such as Wi-Fi/Cellular/Satellite modem to communicate with the server. In this case, the software (that processes the task/rule) runs on the server. Fig. 2 shows the representation of this architecture.

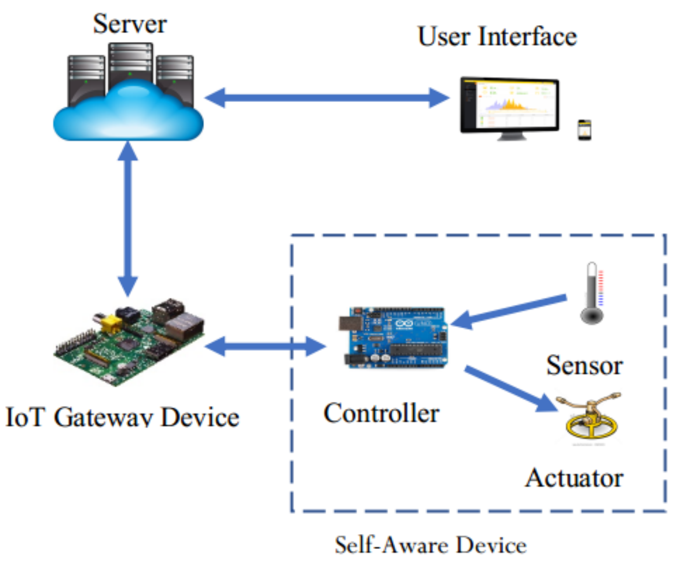

Figure 2 Gateway based self-aware architecture

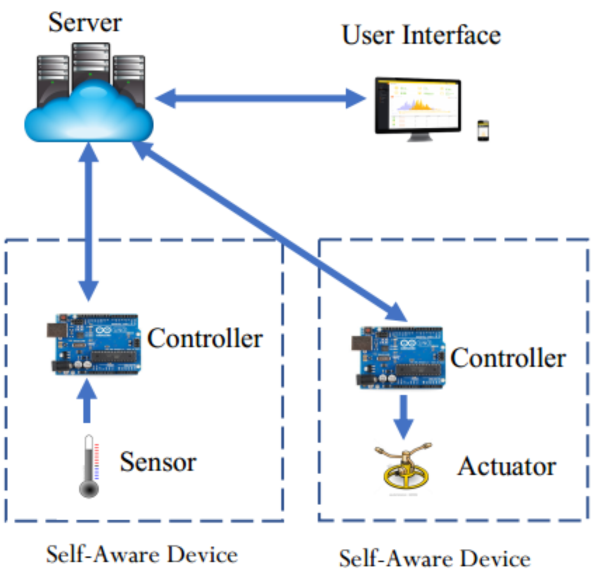

Figure 1 Standalone self-aware architecture

HARDWARE CUSTOMIZATION

We present the details of hardware customization assuming the gateway-based architecture is used. The goal of hardware customization is to enable the user to add or remove sensors/actuators as the needs change (instead of replacing the whole IoT system). To achieve such a goal, every time a new self-aware sensor/actuator is connected to the IoT gateway, the self-aware sensor/actuator should be able to identify itself to the gateway. The sensor/actuator knows the basic details about itself and, once connected to the IoT gateway, it communicates those details to the IoT gateway device. The following section gives the details of self-aware sensor/actuator. We use the term developer to denote a person who is an expert in hardware/software and can create a self-aware device (sensor or actuator). The term user refers to the end user of the IoT system with no expertise in hardware/software.

A. Self-Aware Sensor/Actuator

A self-aware sensor/actuator is basically a sensor/actuator with additional components to make that sensor/actuator self-aware. A self-aware sensor/actuator typically has the following components.

- Sensor/Actuator
- Processing Element
- Driver circuit for an actuator
- External memory (if processing element doesn't have sufficient memory)
- Communication interface to communicate with IoT gateway device
- Power supply (if sensor/actuator can't be powered by IoT gateway device)

B. Converting a Sensor/Actuator into a Self-Aware Sensor/Actuator

Steps to convert an arbitrary sensor/actuator into self-aware sensor/actuator:

1. Select suitable processing element.
2. Interface sensor/actuator to processing element.

3. Store parameters of sensor/actuator in the memory.
4. Create a communication interface for communication between a self-aware device and the IoT gateway device connected to it.
5. Create and embed self-aware device discovery code on IoT gateway device and self-aware device to detect connected nearby (if wirelessly connected) self-aware devices.

Select Suitable Processing Element

Depending on the type of sensor/actuator, the developer selects the processing element. For example, if the type of sensor is analog, it is advisable to select a processing element which has an inbuilt analog to digital converter. Some sensors such as crack detection sensor or fingerprint sensor require a complex algorithm to read sensor value, and hence a processing element with significant memory and processing power is preferable. The power consumption of processing element is also important, as some installations may require a battery powered solution. For those applications, processing elements with power saving capabilities should be selected.

Interface Sensor/Actuator to Processing Element and Program Parameters of Sensor/Actuator into Memory

The developer interfaces the sensor/actuator to the processing element depending on the type of the sensor/actuator. After that the developer programs the parameters of sensor/actuator into the memory. The parameter can be of two types: fixed parameters and user configurable parameters. Fixed parameters don't change over time (such as the unique id of a sensor/actuator, type of a sensor/actuator or manufacturing date of sensor/actuator). User configurable parameters are the parameters which the user can define according to the application (for example, the time interval between two consecutive sensor readings). The user can define these user configurable parameters from the user interface.

Communication Interface between Self-Aware Device and IoT Gateway Device

The communication interface between the self-aware device and the IoT device should be user-friendly and known to the user. WiFi, USB, and Bluetooth are some examples. The processing element is connected to one such communication interface to communicate with self-aware sensor/actuator. The IoT device and self-aware sensor/actuator communicate using two methods: query-response method and interrupt method. In the query- response method, the IoT device queries the self-aware sensor/actuator and self-aware sensor/actuator responds it. In the interrupt method, the IoT device turns on the interrupt mode where the self-aware device sends messages to the IoT device without any query. We have developed a library for the query response model in embedded-c which is suitable for most embedded hardware used for self-aware device.

Self-Aware Device Discoverable Code for IoT Gateway Device

An IoT gateway device should be able to discover all the self-aware devices near it (for wireless connection) and all the devices connected to it (for wired connection). For different types of communication protocols, the method of discovering self-aware sensor/actuator changes. For example, for USB, the developer just needs to check the/dev/tty USB ports for checking connected USB device. For Wi-Fi, a multicast signal with a certain message can be sent and all the nearby self-aware devices respond back. The developer should write code to enable discovery for all the communication interfaces available on that IoT device. The developer also writes the code for self-aware device, so that it responds back to the IoT device's device discovery query.

C. Self-Aware Virtual Sensor

Virtual sensors are sensors that are not physically connected to the IoT system. For example, weather feed from the weather API and time from the NTP server are some of the virtual sensors we have incorporated into our implementation. We have created a virtual sensor API in our architecture, which takes virtual sensors as input and attaches a unique id, sensor type and other parameters to make it self-aware.

D. Validation of Sensor/Actuator Value

Validation of sensor values is an important feature of the self-aware architecture. As the sensors/actuators are self- aware, they know their typical range of readings of the sensor outputs. These will also be stored as part of the sensor-specific data. If any sensed value is out of this range, the self-aware sensor notifies the user about the deviation. For example, a ds1820 temperature sensor is connected to the self-aware device and the maximum value ds1820 temperature sensor can have is 125 ℃ [10]. If the sensor reads more than 125 ℃, the self-aware sensor itself needs to generate an error message (in addition to possibly sending the error reading to the server). Another example of self-validation is related to a self-aware actuator. Suppose we have a relay that controls a compressor as the actuator, which has been made self-aware. For compressor longevity, the compressor should not be turned on and off frequently. There should be a minimum elapsed time between two successive times when it is turned on [11]. The self-aware actuator knows this constraint and, if it receives too many commands for turning on and turning off the relay (actuator), it disregards the received commands and generates an error to inform the user. This property makes sure that neither the user nor the server needs to be aware of actuator-specific constraints. Instead, the self-aware device has the knowledge of the constraints and has checks and balances built in.

E. Working Status of Sensor/Actuator

It is important to know if the connected sensors/actuators are actually working or not. For a sensor, we can detect its working condition by its current value. If a sensor stops sending values or sends values that are out of range, we can conclude that that sensor is not working properly assuming that the communication channel is not faulty. This type of property only works on sensors which give analog values (e.g. temperature sensor or pressure sensor which gives output over a range). It is not possible to detect working status for some of the digital sensors (for example touch sensor which gives either 1 or 0 as its output). We cannot decide whether the sensor is stuck at a single value or it is providing normal input. We can define such analog and digital sensors in the device property part of the unique id of the sensor and notify the user whether the user can get the working status of the sensor or not.

Checking the health of an actuator is harder than that of a sensor. Many actuators may not provide any feedback to the processing element. However, this can be rectified by using auxiliary parts. For example, we can connect an appropriate new sensor to the actuator and retrieve values from the sensor. Using the output of the new sensor, we can check whether the actuator is working or not. Failure of the feedback sensor can raise a false alert. Fig.3 explains the working of this feedback mechanism.

V. Software Customization

Our architecture provides user customization of software where the user can create tasks/rules and set the user defined parameters for self-aware sensors/actuators. For example, the user can set how frequently the user wants to read the sensor values as a user-defined parameter.

Rules are divided into two main components:

- Trigger Condition: When the trigger condition is true, the rule is executed by either IoT gateway device or the server. A single rule can have multiple trigger conditions. When a rule has more than one trigger condition, the rule is executed when all the trigger conditions are true or a specific combination of triggers occurs.
- Actions: This represents the actions to be taken when the conditions are true. Some examples of actions our architecture supports are as follow.
  1. Send Text Message/App Notifications.
  2. Send Email
  3. Make phone calls and play notification message
  4. Turn on/off Actuator
  5. Turn on/off Main Power

More actions are user customizable and can be added to the system.

A single rule can have more than one action. All the actions are executed when the rule is true. An example of full rule is: "If (temperature > 82 && humidity > 40) then "turn on" the fan controller and send message & email" This rule will turn on fan controller (a self-aware actuator) and send out the notifications when the value of temperature (a self-aware sensor) is more than 82 and the value of humidity (a self-aware sensor) is more than 40. While this is a simple if then else type of rule more complex rules are also possible.

A. Execution of Rule

A rule can be executed either on the server or on the IoT device. For the standalone self-aware architecture, a rule must run on server because the standalone self-aware devices may not be capable of running the rule engine. For the gateway-based architecture, a rule can run either on the server or on the IoT gateway device. When all the self-aware sensors/actuators attached with the rule are connected to the same IoT gateway device, the rule can be executed on the IoT gateway device. When the self-aware sensors/actuators are connected to different devices, the rule is executed on the server. When a rule is executed on the server, slow or intermittent internet connectivity can cause problems in execution of the rule. When a user creates a rule, the rule is stored in our database in the form of different tables. Execution of the rule is done as follows:

If the server is executing the rule, it fetches the rule from the database and sends it to the rule engine. The rule engine is a service in our architecture, which takes the rule as input, fetches sensor values related to the rule and outputs appropriate commands to the actuator. If the IoT device is executing the rule, it fetches the rule from the server using REST calls and after that, it sends the rule to the rule engine running on the IoT device. The architecture of Rule Engine is shown in Fig. 4.

The code executed by the IoT device or the server is generated automatically by our backend. The user doesn't write any code.

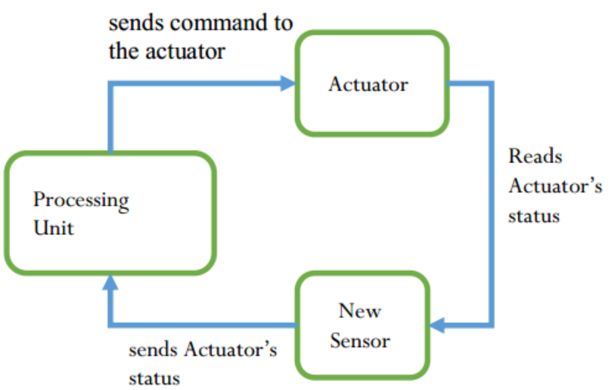

Figure 3 Feedback system for an actuator

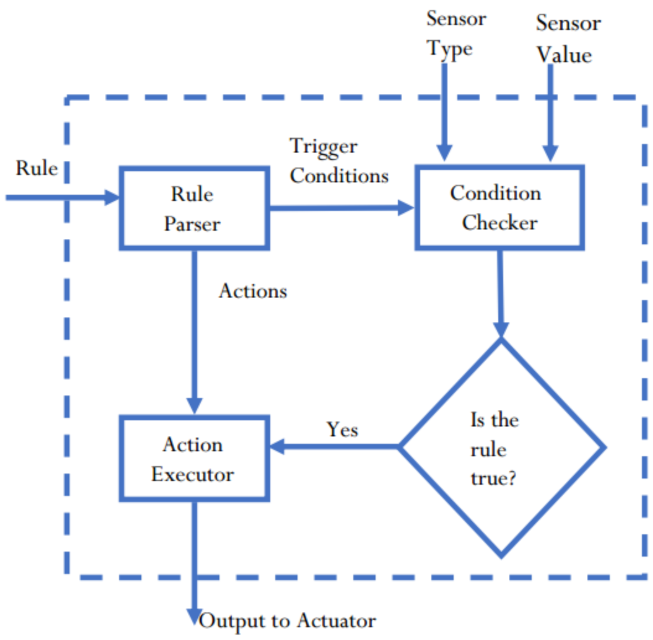

Figure 4 Rule execution block diagram

IMPLEMENTATION

We have implemented a basic prototype for such a self-aware sensor/actuator architecture. This implementation consists of four basic components: IoT gateway Device, self-aware sensor/actuator, server and user interface.

A. Server

We have used a server running CentOS Linux by Digital Ocean. We have used JAVA Spring MVC framework as our main web server. REST calls are used to communicate with the server. All the self-aware sensor and actuator details are stored in the database at our server, once they become active. Hibernate platform is used to communicate with the database from the web server.

B. IoT Device

We have used a Raspberry Pi as the IoT device. We have created four threads to perform the following tasks independently:

- Detect any new USB device connection or removal of a self-aware device.
- Communicate with connected self-aware sensors/actuators via USB. When there are more than one sensors/actuator, the IoT device communicates with them one by one.
- Get push notification from the server via MQTT.
- Run the rule engine.

When any newly connected device is detected by the first thread, the thread checks for the type of new device connected to it and if the new device is a self-aware device, it configures it and adds it to the list of connected devices.

According to the constraints of self-aware sensor/actuator, the second thread will communicate with the sensor and upload data through the services provided by the server.

If the user wants to manually turn on/off the actuator, the user can send a request to the server via the user interface. The server sends the same request to the appropriate IoT device using MQTT notification. The third thread running on IoT device receives this request and sends the appropriate command to turn on/off the actuator.

If all the self-aware sensors and self-aware actuators related to a rule are attached to the same IoT device, the last thread of IoT device code executes the rule and sends appropriate actions to the self-aware actuator.

C. Self-Aware Device

For proof of concept, an Arduino board has been used to build the self-aware sensor (or actuator). [A custom hardware will make the device minimal and aesthetically appealing, but will be time-consuming to build]. Temperature sensor ds1820 is connected to it. The sensor ds1820 works on 1 wire protocol. When the Arduino is connected to the Raspberry Pi it sends all details such as unique id, sensors connected to it, the range of the sensor, sensor's other constraints, etc. All these details are stored in nonvolatile memory of Arduino. After getting all these values, the Raspberry Pi will periodically request values from Arduino using AT commands. The Arduino will read the value from ds1820 via 1 wire protocol, and send the value to the pi. Before sending data to the pi, the Arduino performs preliminary check on the data to ensure that the data being provided to the Raspberry Pi is valid.

D. User Interface

We have created a simple web interface. It consists of following web pages.

- Summary Page: It displays basic details of the user. It also displays all the IoT devices owned by the user. The user can import a new IoT device and delete an existing IoT device from this page.
- Rule Page: User can create a new rule from this page. The user can select the IoT device to "program", see what sensors and actuators are connected to it, and then from a set of pull-down menus, select the desired behavior. This page is responsible for taking all the data from the user to create the needed tables which will be used by our backend to automatically create code to implement the rule.
- Data Display Page: All the sensor values are displayed on this page.

E. Actual Implementation

We have converted the following sensors and actuators into self-aware sensors and self-aware actuators.

Sensors:

- Temperature sensor
- Soil moisture sensor
- Water level sensor
- Ultrasonic sensor
- Barcode reader

Actuators:

- Sprinkler controller
- Fan controller
- Magnetic Lock controller

We have also deployed one system at a local organic farm named Profound Microforms and the system has been operating well for the past 9 months. The system consists of two self-aware sensors measuring air and water temperature respectively. The user can customize text message alert or email alert on this system. The farmer at Profound Microforms have customized alerts according to their requirements. For example, during winter they set an alert for the air temperature value greater than 70 °F and during summer they set an alert for the air temperature value greater than 85 °F.

PERFORMANCE ANALYSIS

We measured timing evaluation and power evaluation for the performance analysis on an actual proof of concept implementation. As the self-aware architecture requires additional processing element, it delays the end to end communication and it also requires more power. We performed our analysis on a Raspberry Pi as the IoT device and Arduino as a self-aware processing element. We used Java as the programming language for Raspberry Pi and C for Arduino.

A. Timing Analysis

We checked timing analysis by performing end to end communication on following two architectures. Basic IoT architecture, where the sensor is connected directly to IoT device and self-aware IoT architecture, where the sensor is connected to IoT gateway device through the self-aware device. Self-aware architecture introduces an additional communication delay between IoT device and self-aware device. This additional delay varies based on data rate used between the IoT device and the self-aware device. In our analysis, we have connected the IoT device with the self-aware device using USB protocol. We performed timing analysis on temperature sensor which contains a payload of 3 bytes from self-aware device to IoT device. We performed timing analysis for 100 cycles and took an average for analysis. Fig. 5 shows the analysis.

B. Power Analysis

Power consumption is divided into 3 main parts. Power consumed by IoT device, power consumed by self-aware device and power consume by sensor/actuator. Power consumed by self-aware devices is an additional power consumed in our architecture. The power consumption of self-aware device depends on the clock frequency of the self-aware sensor/actuator, communication data rate between the self-aware device and the IoT device, communication frequency (how often) between the self-aware device and IoT device and power saver mode used in the self-aware device. We theoretically calculated the power consumption using the following assumptions: the self-aware device has sleep mode and it wakes up periodically to sense the reading and sends it to the IoT device. Communication happens over USB with 9600 baud rates. Clock Frequency and voltage used are 16MHz and 5V respectively. For our configuration, current consumption during wake-up mode is 19.9 mA and sleep mode is 3.14 mA [13, 14]. Power consumptions for various duty cycles are shown in Table 2. Power consumption is computed using the following equation:

$$P = \frac{ts * Vin * I(sleep) + tw * Vin * I(wake\ up)}{ts + tw}$$

Vin = input voltage
ts = sleep time
tw = wake up time
I(sleep) = current consumption during sleep time
I (wake up) = current consumption during wake-up timeP = Power consumed

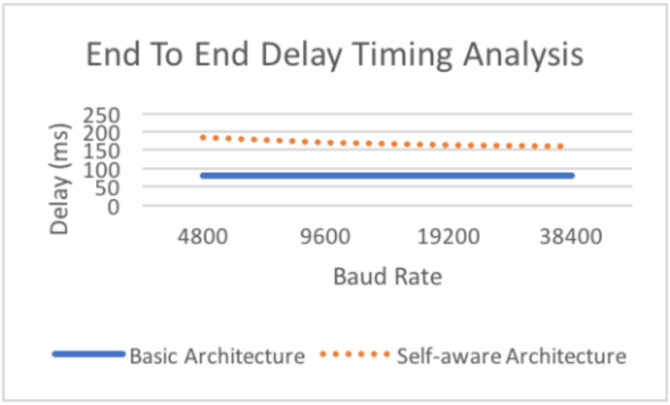

Figure 5 Timing Analysis

Wakeup time	Sleep time	Power consumption
1 second	59 seconds	17.09 mW
1 second	29 seconds	18.49 mW
1 second	9 seconds	24.08 mW
1 second	0 second (orno sleepmode)	99.5 mW

Table 1 Power Consumption

CONCLUSION

We have designed and implemented a user configurable IoT architecture using which a user can add (or remove) one or more self-aware sensor/actuator to (or from) the IoT system without knowing any hardware knowledge. The user can also define rules that will govern the execution of the IoT system, without having to write any software. This architecture enables a novice user to build a highly customized IoT system. We have built several proof of concept prototypes to validate the idea.

Security is an import requirement of IoT systems. Numerous approaches are possible for this task. Implementing a suitable security protocol for our framework is an important next step. We are working on this and related issues.

REFERENCES

1. Khan, R., Khan, S. U., Zaheer, R., & Khan, S. (2012, December). Future internet: the internet of things architecture, possible applications and key challenges. In Frontiers of Information Technology (FIT), 2012 10th International Conference on (pp. 257-260). IEEE.
2. Ishaq, Isam, et al. "IETF standardization in the field of the internet of things (IoT): a survey". Journal of Sensor and Actuator

Networks 2.2 (2013): 235-287.

3. Bandyopadhyay, Debasis, and Jaydip Sen. "Internet of things: Applications and challenges in technology and standardization". Wireless Personal Communications 58.1 (2011): 49-69.
4. Tao, F., Cheng, Y., Da Xu, L., Zhang, L., & Li, B. H. (2014). CCIoT-CMfg: cloud computing and internet of things-based cloud manufacturing service system. IEEE Transactions on Industrial Informatics, 10(2), 1435-1442.
5. Alam, Sarfraz, Mohammad MR Chowdhury, and Josef Noll. "Senaas: An event-driven sensor virtualization approach for internet of things cloud". Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on. IEEE, 2010.
6. Tan, Lu, and Neng Wang. "Future internet: The internet of things". 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE). Vol. 5. IEEE, 2010.
7. Datta, Soumya Kanti, Christian Bonnet, and Navid Nikaein. "An IoT gateway centric architecture to provide novel M2M services". Internet of Things (WF-IoT), 2014 IEEE World Forum on. IEEE, 2014.
8. Ren, J., Guo, H., Xu, C., & Zhang, Y. (2017). Serving at the Edge: A Scalable IoT Architecture Based on Transparent Computing. IEEE Network, 31(5), 96-105.
9. Hada, Hisakazu, and Jin Mitsugi. "EPC based internet of things architecture". RFID-Technologies and Applications (RFID-TA), 2011 IEEE International Conference on. IEEE, 2011.
10. https://datasheets.maximintegrated.com/en/ds/DS18S20.pdf
11. http://www.ni.com/white-paper/2774/en/
12. Yashiro, T., Kobayashi, S., Koshizuka, N., & Sakamura, K. (2013, August). An Internet of Things (IoT) architecture for embedded appliances. In Humanitarian Technology Conference (R10-HTC), 2013 IEEE Region 10 (pp. 314-319). IEEE.
13. http://www.home-automation-community.com/arduino-low-power-how-to-run-atmega328p-for-a-year-on-coin- cell-battery/.
14. http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf
15. https://m2x.att.com/
16. https://aws.amazon.com/iot/
17. Kortuem, G., Kawsar, F., Sundramoorthy, V., & Fitton, D.(2010). Smart objects as building blocks for the internet of things. IEEE Internet Computing, 14(1), 44-51.
18. Ghosh, Debashish, Fan Jin, and Muthucumaru Maheswaran. "JADE: A unified programming framework for things, web, and cloud". Internet of Things (IOT), 2014 International Conference on the. IEEE, 2014.
19. Stirbu, Vlad. "Towards a restful plug and play experience in the web of things". Semantic computing, 2008 IEEE international conference on. IEEE, 2008.
20. Zeng, D., Guo, S., Cheng, Z., & Pham, A. T. (2011, September). IF-THEN in the internet of things. In 2011 3rd international conference on awareness science and technology (iCAST) (pp.503-507). IEEE.
21. Sarkar, C., Nambi, S. A. U., Prasad, R. V., & Rahim, A. (2014, March). A scalable distributed architecture towards unifying IoT applications. In Internet of Things (WF-IoT), 2014 IEEE World Forum on (pp. 508-513). IEEE.
22. Alam, S., Chowdhury, M. M., & Noll, J. (2010, November). Senaas: An event-driven sensor virtualization approach for internet of things cloud. In Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on (pp. 1-6). IEEE.
23. Kiran, M. S., Rajalakshmi, P., Bharadwaj, K., & Acharyya, A. (2014, March). Adaptive rule engine based IoT enabled remote health care data acquisition and smart transmission system. In Internet of Things (WF-IoT), 2014 IEEE World Forum on (pp. 253-258). IEEE.