Flexible IoT Middleware for Integration of Things and Applications

Joseph Boman
Department of Computer Science
University of Southern California
Los Angeles, USA
Email: jboman@usc.edu

Jonathan Taylor
Department of Computer Science
Tougaloo College
Tougaloo, USA
Email: joncody2012@gmail.com

Anne H. Ngu
Department of Computer Science
Texas State University
San Marcos, USA
Email: angu@txstate.edu

***Abstract*—The Internet of Things (IoT) is a rapidly growing system of physical sensors and connected devices, enabling an advanced information gathering, interpretation and monitoring. However, IoT must be supported by a middleware that allows IoT consumers and IoT application developers to interact in a user-friendly way, despite the differences in each user's perspective of IoT system. To that end, our software attempts to bridge the gap between IoT consumers and IoT application developers. Through the coupling of GSN (an existing open source IoT middleware), Firebase (a cloud storage service), and an IoT data interpreter developed by us, we have created a software system that takes the first step towards an ubiquitous middleware for IoT.**

I. Introduction

Internet of Things (IoT) is a domain that represents the next most exciting technological revolution since the birth of the Internet. IoT will bring endless opportunities and impact every corner of our planet. With IoT, we can build smart cities where parking space, urban noise, traffic congestion, street lighting, irrigation, and waste can be monitored in real time and managed more effectively. We can build smart homes that are safe and energy-efficient. We can build smart environments that automatically monitor air and water pollution and enable early detection of earthquake, forest fire and many other devastating disasters. IoT can transform manufacturing, making it leaner and smarter. According to CBS news, we've had nearly 600 bridge failures in the country since 1989. A large number of bridges in every state are really a danger to the traveling public. IoT can monitor the vibrations and material conditions in bridges (as well as buildings and historical monuments) and provide early warning that would save numerous human lives.

While Internet of Things (IoT) offers numerous exciting potentials and opportunities, it remains challenging to effectively manage the various heterogeneous components that compose an IoT application in order to achieve seamless integration of the physical world and the virtual one. In this paper, we investigate a generic IoT middleware system equipped with a set of tools, supporting streamlined development of IoT applications and their convenient maintenance, and extension. This middleware system leverages a cloud storage service (Firebase [1]) for sharing and consumption of IoT data and an open source sensor data management system (GSN [2]) for data acquisition from a variety of sensors. Furthermore, in using Firebase to store the data received from the sensors and a general purpose data interpreter for the collected data on Firebase, this middleware enables the deployment of any third party application that can interface with Firebase. To demonstrate the capability of this middleware, we have created two simple applications: a web application that enables a user to visualize the status of the physical devices that the sensors monitor, and a generic notification application (utilizing JESS - a Java based rule engine) which can post interpreted data collected for physical devices to social media such as Twitter, Facebook, or send messages to an e-mail address when certain conditions are detected by the sensors.

Existing IoT middleware falls into two categories in general. The first category allows users to add on as many sensors as they desire, and then gives users some tools (simple App or Web browser) to view the raw data that the sensors are collecting, but usually has limited functionalities when it comes to interfacing with other applications or interpreting the data. The second category limits the user on the type and the number of sensors that they can utilize, but enables the user to interpret the collected data - since possible use cases can be determined and programmed in a-priori - and to interface with many third party applications, usually

through some cloud storage services. Most of the existing IoT middleware which focused on supporting a large variety of sensors tend to fall in the first category, while IoT middleware focused on consumer usability falls into the second category. We analyzed the infrastructure of both existing commercial and academic research IoT middleware and by combining and extending some of their components, enabled IoT consumer to both add as many sensors as they would like and to consume IoT data in a flexible way. In our framework, data collected from sensors not only can be monitored in realtime, but also automatically converted into actionable information by our data interpreter based on trigger values or conditions preset by the user. This gives contextual information about the physical devices and enables IoT applications to be developed by accessing those high-level contexts independent of low level physical properties of the sensors or devices.

We designed and implemented a sample *smart home* web application prototype and a sample third party application that post messages to social media or send email notificatons to showcase our IoT middleware.

The remainder of this paper is organized as follows: a discussion of related work that currently exists in the field of the Internet of Things is presented in section II, an overview and in depth examination of the middleware that we created is described in section **??** followed by a discussion about the shortcomings and advantages of our software. The conclusions that we have drawn from creating and examining this middleware is presented in section IV. Finally, the future work that will need to be done in regard to our IoT middleware prototype in order to make it into a practical system is discussed in section V.

II. Related Work

Paraimpu

Paraimpu is a REST-based web-service that connects physical things/sensors to the web. Its main emphasis is on sharing of IoT data on the Web. You can interconnect most of your sensors to appliances or social media such as Facebook and Twitter [3]. Paraimpu also allows other people to integrate your IoT data with their own Things via a rule engine. However, the Paraimpu rule engine is very simplistic. The if/then rule format is what is used and it is not possible to connect multiple actuators or sensors with each other. A sensor must only be attached to an actuator and if you want to have a second actuator for the the same connection, you must make a separate rule to achieve it. Another problem with Paraimpu is that because the site is still in development, you are limited to certain number of sensors and actuators at one time on the site [3]. Finally, there is a limited type of sensors that are currently supported and if you want more type of sensors to be added, you have to use a third party web service Xively to do so.

Xively

Xively, along with LogMeIn, is a public cloud service that emerged from the IoT movement. The company has many tools and resources that developers can use to connect their sensors and collect data from those sensors [4]. If a developer comes across problems when using Xively API, there is also a large community of developers that can help. Xively overall mission is to help developers and companies to design physical sensors and connect them to their IoT cloud service quickly and simply.

Xively is basically used just for connecting the sensors that you want to Xively's cloud. If you want a rule engine that can interpret the data, you will either have to develop one yourself or find one that is compatible with Xively. The main goal for Xively is ultimately connecting the sensors you need to their cloud, where you can pull data from them easily from anytime and anywhere. There is no streamlined supported for developing actuators that can be used to control the sensors. Even though adding sensors is supposed to be simple using Xively, the API that is provided can be difficult to use, especially when using a sensor that is not already supported by the site. Because the libraries are in beta version, only experienced developers can set up actuators or add new sensors.

Google Nest

Google Nest consists of two fully integrated devices that contain sensors and processors to interpret the data collected. These devices are a thermostat and a smoke detector. Once the data is interpreted, the devices send the results as a JSON document and host the document on Firebase. Other applications can then read the data from the JSON document or make changes to certain sections of the document, in order to react appropriately to the readings that the sensors have found [5].

While Google Nest provides an excellent method of interfacing, through Firebase, and the API makes it easy for any 3rd party application to connect and utilize the data from Google Nest's devices, the fact that the devices are integrated with their software that interprets it means that to add any devices or sensors to the system requires

an inordinate amount of work. Essentially, unless Google Nest decides to add more devices for a consumer to purchase, the system is limited to just the smart thermostat and smoke detector. However, despite this shortcoming, Google Nest is still a big step forward towards the fully IoT integrated home, and we draw upon their ideas for using a hosted and standard-based JSON document server, Firebase, for the storage and control of sensor data.

Global Sensor Network

Global Sensor Network (GSN) is a middleware that enables users to more easily integrate with various different types of sensors. Primarily, GSN handles the thread management and data storage aspects of dealing with sensors. The users must create an XML file and a wrapper for each different sensor that they want to connect to the network [2]. The XML file tells GSN about the basics of the sensor: what kind of data (numerics, spatial) it will be giving to the system, any parameters (such as how often to ping the sensor to get data), and which wrapper or virtual sensor to use with the physical sensor. The wrapper tells GSN how to connect (e.g. which network protocol to use) to the sensor when it is first initialized, what to do in order to get data from the sensor, and what to do with the data when it is received from the sensor. These two things must be created for every new sensor that one wishes to connect to GSN. Currently, there is no easy way for users to create new XML files or wrappers, besides taking the existing ones and modifying them or writing them from scratch. Furthermore, while GSN stores the data from the sensors in an SQL database, it doesn't do anything to the data other than display it on a local web application, and the only data displayed is the raw sensor data - with no interpretation [2].

Ninja Blocks

Ninja Blocks consists of a system based around a central control hub that has the ability to control any devices that are plugged into smart power sockets. The focus of the system is control rather than sensing, and as such, there are a limited number and type of sensors that can be purchased from Ninja Blocks, however, the system is able to interface with a variety of other products from other companies, including Google Nest and many others [6]. Furthermore, the system can be monitored and controlled from a mobile application as well as the central hub. The main limitation of the system is that automation is rather limited. While you can receive notifications via the mobile app and turn on and off devices for it, you cannot set up rules that will monitor certain conditions or patterns, limiting the scope of what you can do with Ninja Blocks. However, despite this, the system as a whole is one of the best consumer based IoT systems currently available.

Twine

Twine is a consumer based IoT system that consists of a hub that contains some simple sensors in it and a web interface that allows the user to create rules that will message social media based upon certain values of the sensors. In addition, other sensors can be purchased and added to the system in order to allow Twine to detect more variety of sensors [7]. Despite the simplicity of Twine, it does fall short in a variety of areas. One such area is the matter of external sensors. Firstly, the sensors must be purchased from the company Supermechanical, which limits the type of sensors that can be purchased. Secondly, the sensor hub limits the number of sensors that can be attached, since there are a limited number of ports to plug in external sensors, and the sensors are all wired sensors. Furthermore, since all of the sensors are wired, the distance that any sensor can be placed away from the hub is limited. However, despite all of this, the web interface is still an excellent example of a rule engine - they allow the user to set up as many rules as they want, with multiple trigger conditions and responses - even with the limited choices for actions (social media messages only).

Smart Things

Smart Things is largely based around a mobile phone application and a central hub. Sensors and actuators can be purchased from Smart Things or other vendors, provided that those devices are on the list of devices that the Smart Things system is able to work with [8]. While the library of available devices is extensive and growing, the system as a whole is designed around the idea of the smart home, and so is not very adaptable to other environments, as almost all the sensors and actuators are designed to use in a standard home. Furthermore, since the devices must be set up by Smart Things before their application can use them, a user does not have the ability to get any devices they desire, and most of the devices that Smart Things has interfaced with are relatively expensive. Finally, Smart Things is based entirely around the mobile application and has no web interface, preventing users from modifying rules and receiving data from a computer.

SmartHome

The SmartHome system is a prototype for a smart home where the user can add any number of sensors and create rules around them to interact with them. Utilizing an internally developed system called sensor hive, SmartHome enabled the use of Phidgets sensors to monitor a variety of household devices. Once the devices are connected, a web interface allows the user to see when the devices are on or off, as well as the history of use of those devices [9]. This system, however, had a relatively simple rule system and the sensor hive could only interface with Phidgets sensors. These two aspects greatly limited the benefits of the system overall, but the web user interface that was used in SmartHome is a remarkably simple one that still accomplishes the goal of getting the information to the user in a way that is useful and intuitive. Therefore, we took a great deal of inspiration from the design of their web interface for visualizing the sensor data. To overcome the limitation of sensor hive, we adopted the GSN server for connecting to a variety of sensors.

III. System Architecture

When we began to build our IoT middleware, the question arose as to which language to write it in, and after examination, we narrowed it down to either C#, following in the footsteps of the SmartHome system, or Java, following in the footsteps of GSN [2]. In the end, we decided to use GSN for sensors connection, which restricted us to using Java. However, this turned out to be beneficial, since Firebase has no current API for interfacing with C# [1]. To utilize Firebase as the storage system for our IoT data, we have to modify GSN so that it will forward its data to Firebase instead of simply storing it in its SQL database. We also developed the Data Interpreter to pull the data from Firebase and utilize it to draw conclusions regarding the current status of the sensors. Our data interpreter is similar to the processors available in Google Nest, but have the advantage that it is an independence unit that can be extended by developers. Finally, we created two sample applications that would be used to demonstrate how other applications could connect to Firebase and still remain separate software systems. Figure 1 shows the architecture of our prototype IoT middleware system, where the arrows represent the flow of data through the system. Note that the Data Interpreter is a separate component from GSN.

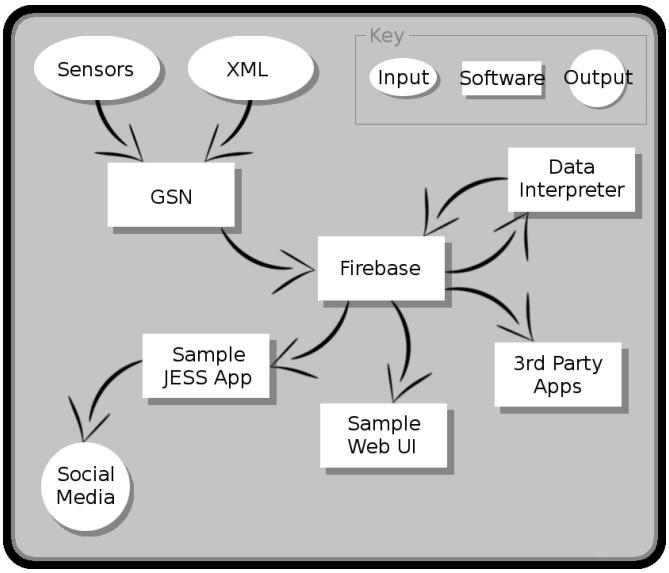

Fig. 1. The Data Flow of our prototype

Inputs

The two inputs to the overall system are the sensors themselves (i.e. the wrappers) and an XML file created by the user that describe the main properties of the sensors. The sensors are relatively self explanatory; each one collects some sort of data and then sends it to GSN - either on a regular basis, or when prompted by GSN. These sensors can be of any type, so long as the data they send is in a numerical format. In the future, this could be changed, however, at the moment, the type of sensors that we experiment with are restricted to collecting numeric data.

The XML file, on the other hand, can get a bit complicated, since it must contain a great deal of data, either for GSN or the Data Interpreter to work properly. Firstly, the XML file needs to list the virtual sensor class for GSN to represent the sensor with (in most situations, the Bridge Virtual Sensor is the one to use) and the name of the wrapper class to use to connect with the sensor and deal with the data. For GSN, the XML file must also include a descriptor of the data that the sensor will be creating - which tells GSN the name of the data being passed to it and the type of data [2]. In addition, for the Data Interpreter and Firebase, six values must be included for each separate numeric value that the sensor will be generating. First, a unique ID, which must be different for every value. Second, the name of the device that the sensor is monitoring, whether it is an actual device, such as a microwave, or something else, such as a doorway. The last four values required are the

trigger values for the device. These values tell the Data Interpreter when the device is on or off, and consist of the following: an on-floor value, an on-ceiling value, an off-floor value, and an off-ceiling value. When the sensor value is less than a ceiling value, then the device is on or off, respectively, and when the sensor value is above the floor value, the same is true. The reason both on and off need a high and low trigger value is because some devices will be on when the sensor value is under a threshold (such as a fridge and temperature sensor), while others will be on when their sensor value are above a threshold (such as lights and a light sensor). In order to prevent mistakes, use the value "-1" if a sensor does not have a particular trigger value. Finally, any values that the user wishes to be able to access in the wrapper class should be placed here as well, rather than hard-coding them into the wrapper.

GSN

The Global Sensor Network serves two main purposes in our code; it connects to the sensors and collects the data from them, and it forwards that data to Firebase in a generic format. In order to interface with the sensors, the user needs to create a wrapper class that inherits from AbstractWrapper which tells the system how to connect to the sensor as well as what to do to get data from the sensor. The wrapper must contain some relatively generic functions that it inherits from AbstractWrapper, however, the most important two functions are initialize and run. Initialize is called when the system is starting up, and should tell GSN how to connect to the sensor. It is here that the sensor should be checked to verify that it is connected, and the sensor should be added to the data uploader, so it can upload it. The run function is called on a regular basis, as each wrapper has a unique thread that will perform the run function over and over [2]. It is here that the code for pinging the sensor to collect information should be placed, as well as the code for sending the new sensor values to the data uploader.

The data uploader is part of GSN, and utilizes a singleton model, so all of the wrappers have access to it and multiple upload requests will not cause the system to repeat the same request. The uploader collects the data that the sensors are giving to it, and on a regular basis, forwards that data up to Firebase. A local copy of the data is stored in the data uploader class, to allow it to determine which sensor to assign the new data to. Currently, initializing the data uploader initializes the Data Interpreter, however, this is just a workaround to make the software easier to operate.

Firebase

Firebase is a cloud storage service that we have leveraged in order to enable our middleware to be decoupled and interface with 3rd party applications [1]. We decided to use Firebase upon an analysis of Google Nest's API. They utilized Firebase to allow developers to create mobile and web applications that used the data generated by their smart thermostat and smoke detector, without having to adjust to the specific format of data they were generating. Figure 2 shows the web interface where data is stored on Firebase and can be edited. Since Firebase stores data in a JSON format, many applications can already interpret that format, but Firebase also allows developers to create listeners for specific sections of the JSON document that will fire when data is changed, added, removed or moved. This means that creating a mobile or web application that will interface is incredibly simple. Furthermore, since Firebase enables users to add authentication to their own personal Firebase, users can rest assured that their data is protected from malicious attackers.

Fig. 2. The web interface for Firebase, showing the data from our system in a JSON format

There are a few potential drawbacks to Firebase, however. Firstly, since the system utilizes its own scheduling system for activating the listeners, it may be the case that when a large number of sensors and applications are connected to the system, the delay in firing the listeners could result in data being received that is no longer

current [1]. Also, Firebase only allows a free account to utilize 500 MB of data. While that is a large amount, and certainly enough for most situations, if users wish to store history for their devices, they would certainly end up using more than that, given enough time. All in all, however, as cloud storage services for IoT go, Firebase seems to be the best currently available, which is why we have chosen to use it.

Data Interpreter

The Data Interpreter pulls all of the sensor data down from Firebase, and then using the devices that were listed as being monitored by each sensor in the XML file, determines which sensors need to conclude that each device is on or off before the interpreter can actually determine that the device is on or off. It creates Firebase entries for each one of the devices that are listed as being monitored and uploads their initial state. Following that, no changes will be made unless the state of the device changes - whether the change is the device going from on to off, off to on, or simply changing location. When such an event happens, the interpreter will upload the new values to Firebase, and update the pastValues list. Currently, the pastValues list only stores up to ten values, since we felt that it would be safer to err on the side of less values, especially since we want the system to have the ability to scale with a large number of devices. Furthermore, due to the implementation of Firebase, the list of pastValues is wiped clean every time the software is restarted. This will need to be fixed, however, the data is still stored locally via GSN in an SQL database, so it exists and is accessible, just not in the cloud as we would have preferred.

Sample JESS App

The JESS Application is a sample application that demonstrates the way that our system can be used to allow a user to create rules that will trigger actuators. In the case of the JESS Application, the rules themselves are stored on Firebase in a specific format, however, in the future, the rules would be native, written through a GUI that would enable the user to drag and drop sensors, devices, and actuators to combine them into rules. JESS is a rule engine based in Java that utilizes the Rete algorithm in order to facilitate a large quantity of rules [10]. The JESS Application pulls the rules, sensor data, and device data from Firebase, and parses the rules into the JESS format, which is similar to Lisp. Once that is done, the rule is passes to the JESS Rete engine, which stores the rules and the data from the sensors and devices.

The application currently only has three different actuators: sending an email, posting a tweet, and sending a message to a twitter account. We attempted to interface Facebook with it as well, however, the difficulty and lack of an official Java API forced us to abandon that. These actuators can be used on separate rules or on the same rule. Furthermore, the rules currently do not support 'or' functionality, which is a failure of the parser - when it was created, it did not have the ability to make separate triggers into 'or' statements, and so it was not implemented. The system as a whole is able to support it, however, it would require a rewrite of the parser or a different method of inputting the rules into the system. Overall, the JESS Application demonstrates the benefits of decoupling the IoT system, since it is able to run independently of the main system, and can actually be run on a completely separate network, provided that it has access to Firebase and the proper authentication to read the Firebase data.

Sample Web UI

The Web UI was largely based on the web interface that was part of the original SmartHome software system, however, it is not the same [9]. Figure 3 shows the Web UI with the mouse hovering over the microwave. The primary purpose of the Web UI is to display the data provided about the devices in a format that is easy for users to interpret in a short period of time. To that end, we emphasized the visual aspects of the system. By making the interface picture based, the system as a whole becomes far easier to interpret. Each device has mouse-over text that displays the name of the device or the current status and the time that the status changed to the current status. Furthermore, since the JESS Application posts data to Twitter, we included the Twitter timeline in order to allow a user to see it without having to search for the specific user.

The web interface is an example of a 3rd party application that a user could choose to add to their personal IoT system. Not every system in the Internet of Things would need or want a visualization for a kitchen, or even a visualization of the system at all, however the benefit of our system is that since it is decoupled, this component can easily be replaced or removed entirely and this removal has no effect on the rest of the system which will work the exact same way. This customization enables each user to personalize their own IoT system without sacrificing anything because they want or do not

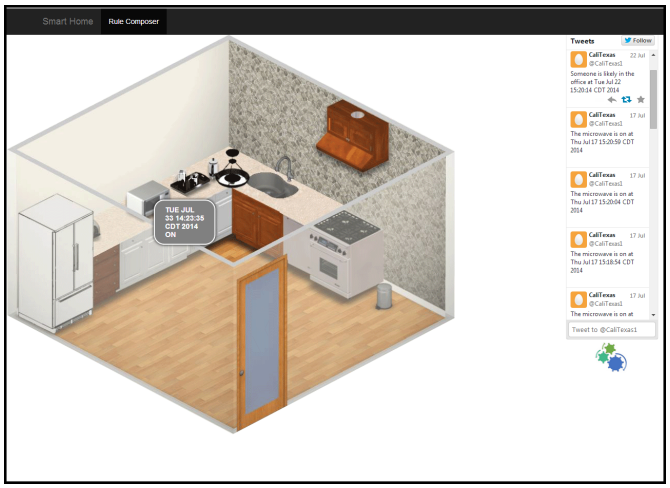

Fig. 3. The web interface that displays the data in a user friendly format

want a particular feature, which betters the idea of the Internet of Things as a whole.

Benefits of our System

- Through the use of GSN and its virtual sensors, our system has the ability to interface with virtually any sensor, enabling any user to add any number and type of sensors.
- The Data Interpreter enables the system to automatically infer the status of any device, based solely on the raw data that the system is receiving from the various sensors attached to it, and the trigger values that the user inputs in the XML file. This allows the system to provide useful real-world information, as opposed to just providing raw data.
- Since the data and information is stored on Firebase, the system as a whole has the ability to expand, interfacing with 3rd party applications, regardless of whether the application is native, mobile, web, or located anywhere else. Furthermore, since Firebase can require authentication, each user can protect their data and information and only give it to the applications which they want to allow to access their data.

Shortcomings of our System

- Since adding a new sensor requires creating an XML file and a wrapper in order to interface with GSN, adding sensors is time consuming and requires the user to have a basic knowledge of programming in Java and writing XML documents.
- Currently, the Data Interpreter is only able to determine whether a device is on or off, and not how long it has been in whatever state it has been in. Also, with devices that have states other than on and off, such as a vent that can be partially open, the Data Interpreter cannot detect that.
- Due to the Data Interpreter, sensors must all provide a numerical value, and the user must determine the trigger values for whatever device the sensor is monitoring before they can use the sensor/device combo with the system if they wish to extract meaningful data.
- Since the ”3rd party” applications we designed were meant as examples, they have a number of shortcomings, from the JESS Application’s rules being stored on Firebase instead of locally, and the Web UI not showing more than the current value for any device.

IV. Conclusion

While the Internet of Things is certainly a long way from becoming ubiquitous, it becomes closer and closer with each passing day. The future of the Internet of Things will consist of a variety of sensors loosely connected into a network that forwards data to some sort of cloud storage service, which will make the data available to either all users, or all users who have the proper authentication. Once the data is in the cloud, the users who pull it down can interpret the data, and send their interpretations back to the cloud, perhaps in the same location or a different one. Finally, other applications or the same ones will use interpreted data to fire actuators or send messages to various final destinations.

This system may appear to be complicated, however, it is one of the most promising models for the Internet of Things. By decoupling everything, it gives the system the ability to recover from shocks - if the sensor network goes down, the storage service and 3rd party applications can continue to operate. Furthermore, it allows any applications and sensors to be connected to the system without having to make sure that all other parts of the system can interface with them. Our system, although simplistic, is a proof of concept of this, and also takes the first steps to making such a system a reality. The future is a lot closer than it used to be, and our system is a basic glimpse of what such a future might hold.

V. Discussion and Future Work

The shortcomings that were discussed above need to be remedied before the system as a whole can move forward and become user friendly to the average user. Some of these fixes are things that will be explored by

our software and personal future work, while other parts of it will need to be changes made by people working on the other parts of our system. One of the largest changes that will need to be explored is the process of adding a sensor to GSN. While some progress has been made in this regard [11], the system as a whole is overly complicated, and some method needs to be found that will simplify the process, whether that involves providing some online resource where users can share their XML files and wrappers, or some form of automatic wrapper generation. Furthermore, Firebase must be explored to determine how well the system can scale - unfortunately, as this would require greater processing and number of sensors than we have access to, we cannot really explore this question.

In regards to our system, there are a few changes that need to be examined. Primarily, a native rule composer should be created that will allow a user to create rules easily through a GUI and fire rules based on the information on Firebase without having to run a separate application or write the rules on Firebase via a specific format. Furthermore, the Data Interpreter needs to be expanded, to add conclusions about states other than on and off, such as movement, partial states, and time of states. Next, a better method of storing the history of sensor readings and history of devices needs to be developed. And finally, the system as a whole needs to be made user friendly, whether this involves creating a GUI for various parts or simply making as much as possible as intuitive as possible. Overall, while this system is a step forward, there is still a great deal of work necessary in order to make it into a system that can be used by anyone and improves the quality of life for the average person.

Acknowledgement

We thank the National Science Foundation for funding the research under the Research Experiences for Undergraduates Program (CNS-1358939) at Texas State University to perform this piece of work and the infrastructure provided by a NSF-CRI 1305302 award.

We thank Charith Perera for discussing the benefits and shortcomings of GSN with us, and answering the questions we had in regard to the papers he had authored.

We thank Craig Smith for granting us a free research based academic license for JESS.

REFERENCES

[1] "Firebase documents," https://www.firebase.com/docs/.

[2] GSN Team, *Global Sensors Networks*, 2009.

[3] A. Pintus, D. Carboni, and A. Piras, "Paraimpu: A platform for a social web of things," in *Proceedings of the 21st International Conference Companion on World Wide Web*, ser. WWW '12 Companion. New York, NY, USA: ACM, 2012, pp. 401–404. [Online]. Available: http://doi.acm.org/10.1145/2187980.2188059

[4] M. Köhler, D. Wörner, and F. Wortmann, "Platforms for the internet of things–an analysis of existing solutions," (Forthcoming).

[5] "Google nest," https://developer.nest.com/documentation.

[6] "Ninja blocks," https://ninjablocks.com/.

[7] "Twine," http://supermechanical.com/twine/.

[8] "Smart things," http://www.smartthings.com/.

[9] Lina Yao, Quan Z. Sheng, Anne H.H. Ngu, Byron Gao, "Sensing Your Surroundings: Enabling Web-based Management of Internet of Things," 2014.

[10] S. Liang, P. Fodor, H. Wan, and M. Kifer, "Openrulebench: An analysis of the performance of rule engines," in *Proceedings of the 18th International Conference on World Wide Web*, ser. WWW '09. New York, NY, USA: ACM, 2009, pp. 601–610. [Online]. Available: http://doi.acm.org/10.1145/1526709.1526790

[11] C. Perera, A. Zaslavsky, C. Liu, M. Compton, P. Christen, and D. Georgakopoulos, "Sensor search techniques for sensing as a service architecture for the internet of things," *Sensors Journal, IEEE*, vol. 14, no. 2, pp. 406–420, Feb 2014.