Web Services Framework for Wireless Sensor Networks

Mark Allen Gray and Philip Newsam Scherer
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County (UMBC)
mgray2@umbc.edu, pscher1@umbc.edu

***Abstract* — The recent proliferation of machine-to-machine service-oriented computing and the emergence of cloud computing platforms and services provides promising new capabilities for wireless sensor networks. A Wireless Sensor Network (WSN) by itself is heavily constrained to low-power usage resulting in low compute and storage capacity. Also, there are problems with aggregating sensor data from multiple WSN deployments for the purposes of creating and sharing sensor data in "big data" form and for sensor data fusion algorithm development. Researchers working on applications that require sensor data for modeling and prediction can simulate that data but testing their models against real-world sensor data and deploying their applications on real-time sensor data streams are repeating challenges. In this paper, we propose a web service framework that addresses and overcomes many of these common problems for users of WSNs. We describe the architecture of the framework and the REpresentational State Transfer (REST) Application Program Interface (API) for accessing framework resources. The results from our initial implementation demonstrated the framework operation over a continuous 175 hour data collection window and successfully presented statistics of processed streaming weather sensor data averaged over this entire data record.**

Keywords* — *Web Services; Service Oriented Architecture; SOA; Wireless Sensor Network; WSN; REST; Cloud Computing.

I. INTRODUCTION

The motivation for this research is the integration of wireless sensor networks with cloud services to operate on "big data' systems and provide access to computationally intensive compute resources. The fundamental requirements of the project were to create a web service that:

(1) Operates on big data,
(2) Provides a computationally intensive service,
(3) Hosts the data and compute resources in a cloud, and
(4) Implements a service oriented architecture.

Our approach was to meet these requirements by creating a web services framework for wireless sensor networks that addresses some of the challenges in that domain. A Wireless Sensor Network (WSN) by itself is heavily constrained to low-power usage resulting in low compute and storage capacity [1]. Also, there are problems with aggregating sensor data from multiple WSN deployments for the purposes of creating and sharing sensor data in "big data" form and for sensor data fusion algorithm development [2][3]. Researchers working on applications that require sensor data for modeling and prediction can simulate that data but testing their models against real-world sensor data and deploying their applications on real-time sensor data streams are repeating challenges [4][5]. Our web services framework (herein after referred to as the "framework") addresses these challenges.

In this paper, we first provide an overview of the framework in Section II and follow that with use case descriptions in Section III and related work in Section IV. We then describe the architecture of the framework and our initial implementation in Section V with a description of the results of our demonstration in Section VI. We conclude the paper with a description of future work in Section VII, a conclusion summary in Section VIII, acknowledgments in Section IX, and a list of references in Section X.

II. SERVICE DESCRIPTION

The web service that we provide is a framework for WSN data collection and processing in a cloud. The framework incorporates a service-oriented architecture (SOA) for distributed computing [6] and a REpresentational State Transfer (REST) [7] Application Program Interface (API) for machine-to-machine communication. To demonstrate the operation, a test case WSN is implemented and included as an example of using the framework. The primary components of the framework are:

(1) REST API
(2) REST Process Server
(3) Hyper Text Transfer Protocol (HTTP) Client Server
(4) Example Sensor Server
(5) Example Data Processing

The test case WSN used collects weather data from temperature, pressure, and humidity sensors. The sensor data is aggregated, time stamped, location stamped, and streamed into the framework where it is recorded to cloud storage resources and made available to users on-demand for inspection or for processing on cloud computing resources.

The entire system is illustrated in Figure 1. There are two basic types of users: data producers and data consumers. Data producers are users that deploy WSNs and add them to the system. When they add a WSN to the system they can choose to make the data recorded from their WSN private, shared in a group, or shared with the public. Data consumers are users that wish to consume data shared by the data producers. A data producer is, by default, a data consumer of their own WSN data and of any shared data from other data producers.

 ISBN: 978-1-61208-337-7

The REST process server forms the core of the system. It implements the API to all of the framework's web services, abstracting the services into a set of resources with operations on those resources and encapsulating all cloud resources comprising the framework. Access to the API requires an API key. An administrator that deploys and maintains a system that uses the framework will allocate an admin API key. Only users with the admin API key can add users to the system.

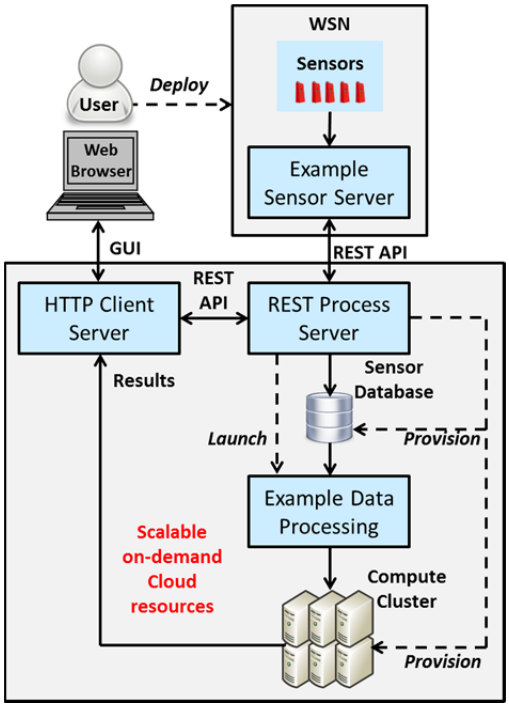

Figure 1. Framework System Components.

Users are added to the system through the HTTP client server. The HTTP client server implements a typical web portal Graphical User Interface (GUI) with user account signup and email verification which uses the admin API key to create the user. The HTTP client server is a user of the REST API. Once authenticated, users use their username and password to log in to their account. Each user has a user profile with an associated workspace and a dashboard for interfacing to the system.

Data producers will use their account to install their WSNs into the system. The account dashboard contains functions to add, modify, and remove a WSN. A WSN comprises a set of sensors and each sensor comprises one or more channels of data. For each WSN, sensor, and channel added to the system, the REST process server will allocate and return a Universally Unique IDentifier (UUID). The data producer will use their assigned API key and these UUIDs in their sensor server program for streaming their WSN sensor data into the system over the REST API to the sensor database. An example sensor server program written in Python [8] is included with the framework illustrating the use of the REST API for these purposes.

Data consumers will use their account to discover and use publically available WSN data or to subscribe to a group share. The account dashboard contains functions providing different views of WSN data including live sensor data being collected, recorded data in the sensor database, or the application of a data processing function to the data and a display of the results.

The system is currently designed with one built-in data processing function; an example data processing program is included. Future work will add the capability for data producers and data consumers to create a library of data processing functions and select the function to apply to a recorded dataset or live data. Additionally, the compute node type and number of nodes in the compute cluster running the data processing program will be user selectable.

All of the components in Figure 1 that are identified as "cloud resources" are deployed on a cloud platform. From the user's point of view, these resources are virtual and elastic. The elasticity of a cloud platform allows the system to scale up and scale down as demands require. For this project, these resources, due to schedule and budget constraints, were allocated on the UMBC BlueGrit computing system [9]. Future work will migrate the system to a commercial cloud platform for reliability and scalability testing purposes on a production cloud, for example Amazon Web Services (AWS) Elastic Compute Cloud (EC2).

III. USE CASES

The analysis, development, and deployment of wireless sensor network technologies are well-established in both academia and industry with applications in military, surveillance, environmental, industrial, transportation, healthcare, agricultural, home, and other many other use cases. Our framework extends these established use cases to address the following problems for hobbyists, researchers, and commercial enterprises:

(1) Aggregating sensor data from multiple WSN deployments,
(2) Creating and sharing sensor data in "big data" form,
(3) Providing a source of sensor data for sensor data fusion algorithm development,
(4) Replacing simulation data with real-world data in modeling and prediction algorithms, and
(5) Deploying algorithms against real-world real-time sensor streams in a cloud.

A. Hobbyists

WSN hobbyists could deploy the web services framework on a public cloud platform to manage the aggregation of their WSN generated data providing centralized access to their data from any Internet connected device. This would allow hobbyists to globally share their data with other hobbyists in a controlled system with authenticated users and managed access permissions. In addition to sharing data, hobbyists could share their sensor data processing functions and generally collaborate with each other on all aspects of their WSN interests. Public cloud platforms often offer free services for usage rates under

 ISBN: 978-1-61208-337-7

thresholds that would meet the requirements for the hobbyist use case.

B. Researchers

WSN academic, commercial, or military researchers creating intellectual property (IP) or other sensitive information could deploy the web services framework on a private (or community) cloud platform to manage the aggregation of their WSN generated data providing centralized access within their organization. This would allow the research team to collaborate amongst themselves or with other collaborative teams within their organization through a controlled system with authenticated users and managed access permissions. In addition to sharing data, researchers could share their sensor data processing functions and generally collaborate with each other on all aspects of their WSN research. As real-world data collects and builds in the sensor database, researchers across the organization could use the data in sensor data fusion and modeling and prediction algorithms.

C. Commercial

A production deployment of the web services framework on a commercial cloud platform could monetize the services and create value for the stakeholders. The service-oriented architecture is scalable over an elastic cloud infrastructure providing the service elasticity required for commercial service deployments. In this scenario, the cost to maintain the service scales up and down as the user demands scale up and down. Usage is on-demand with pay-as-you-go billing. Users on a commercial deployment could collaborate in the same way as described for hobbyists and researchers. The framework could be extended to support multiple cloud platforms with different price points that the user would choose or the user could provide the framework with the access credentials to cloud resources that they already have accounts with, in which case usage against those accounts would accrue against those accounts and a service fee would be added to monetize the transaction for the stakeholders.

IV. RELATED WORK

In this section, we look at current research and commercially deployed products that are related to web services for wireless sensor networks.

A. WSN Middleware

There is current academic research in the creation of WSN middleware primarily focused on the virtualization of WSN resources in a similar way that cloud computing offers virtualization of data and compute resources. One notable project is called “Serviceware” [10]. Serviceware is a service-oriented architecture of middleware that runs over the embedded WSN devices providing virtualization of the hardware in the form of services to multiple users concurrently. The motivation here is to drive down the cost of deploying, managing, and maintaining large-scale WSNs by maximizing the utility of the WSN resources to a broader user base and applications through infrastructure sharing. The authors note that maximizing WSN device utility also increases power consumption and further research is required to analyze the utility gains against the need to replace batteries more frequently.

B. SensorCloud

SensorCloud [11] is an existing commercially available proprietary product offering similar services as our web services framework for WSNs. Customers sign up for an account, choose a level of service with associated cost, receive an API key, and use the key to write code on their Internet connected sensor network devices that use their REST API. Like our REST API, users can get, add, update, and remove sensors and channels from their account and stream their sensor data to their account where it is stored in a database for query, retrieval, visualization, and analysis using data processing functions supplied by the user.

Unlike SensorCloud, our entire framework, including the front-end web portal and the back end REST server, will be open source and operate on top of open source web service software stacks. Additionally, our front-end web portal provides a user interface to get, add, update, and remove WSNs, sensors, and channels. For each resource added, a UUID is assigned and the user simply uses the UUID in their code. All of this can also be done through our REST API in the same way one would if using SensorCloud. Further, each WSN in our framework has Global Positioning System (GPS) location and altitude information and each sensor attached to a WSN has X,Y,Z grid coordinates relative to the GPS location and altitude. Streamed sensor samples include both time and location data supporting mobile wireless sensor networks. A feature that SensorCloud includes that we currently have not specified is the ability to define Short Message Service (SMS) and email alerts when certain user-defined conditions are detected.

C. Google’s Data Sensing Cloud

At the 2013 Google I/O Developer’s Conference in the San Francisco Moscone Center, Google implemented a version of the O'Reilly Data Sensing Lab, a collaborative project between O’Reilly Media and some of their partners. Google’s Data Sensing Lab deployed a 525 node, wireless sensor network at the conference feeding over 4000 continuous streams of sensor data into the Google Cloud Platform with Google Cloud Datastore for sensor data recording and Google Compute Engine for sensor data processing with results presented through a web application. Sensing consisted of temperature, humidity, noise, light, motion, and pressure to analyze the general atmosphere and traffic patterns of conference attendees throughout the conference’s changing of events and agenda. A Google representative at the conference stated “We think about data problems all the time and this looked like an interesting big data challenge that we could try to solve.” [12]

The fundamental architecture of Google’s project is very similar to our web services framework, although their focus was not in developing and demonstrating the required web services with an API, but on raising awareness and interest in hobbyists to build sensor nodes (the “lab” part of the project) and connecting to, and using, their cloud services.

 ISBN: 978-1-61208-337-7

D. Xively Cloud Services

Xively is building a business around services for the Internet of Things (IoT). User's develop and deploy their IoT products into the Xively "Connected Object Cloud" using Xively development tools, directory services, and data services through their API. The Xively API is a REST interface providing developers with web services to stream and record their sensor data to Xively servers and connect to other objects in the Xively cloud. Users, for a fee, can connect to those applications and embed the results in their websites or stream the data, for a fee, into their applications using the Xively API. The user relationships within this cloud ecosystem form a marketplace for real-time sensor fee-based data trading between connected devices and applications. "This sort of common platform is exactly what the Internet of Things really needs. Xively and similar platforms like Open.Sen.se will make it much easier and faster for unrelated devices to connect with each other and start delivering on the promise of smart homes, intelligent devices services and similar long-promised notions." [13]

An interesting capability here is the ability of sensor applications to use each other's data. The addition of REST services to our framework for the data processing function to use the data from other WSN sources would move our framework into the realm of this IoT paradigm. Whereas the Xively service is a closed proprietary deployment on Xively cloud resources, our framework, when deployed, will be open source for deployment on any cloud platform.

E. IBM InfoShpere Streams

Typical processing of big data resulting from sensor networks is performed on data that has been collected into a database where it is later queried, extracted, and analyzed. In the IBM InfoSphere Streams ("Streams") architecture, real-time sensor data streams are analyzed on a high performance computing platform before storing to a database. In this paradigm, data analysis is continuous, resulting in a continuous stream of low-latency real-time results for trend prediction, accelerating user responses to critical real-time events. A motivation for business applications is to address global economic competition; a motivation for government applications is to address global cybersecurity threats. Other example applications include telecommunications, financial services, healthcare, transportation, environmental, insurance, and utilities. Streams can consume data from satellites, sensors, cameras, news feeds, and a variety of other sources including traditional databases and Hadoop systems. In summary, Streams can process huge volumes and varieties of real-time data from diverse sources with very low latency, providing decision makers with the relevant and timely information they need [14].

An interesting capability here is the ability to process the sensor data in real-time before recording to a database. The addition of REST services to our framework for the data processing function to be a applied to the data either before or after recording to the database would extend the our framework to provide a similar capability. Whereas the Streams service is a closed proprietary deployment on IBM cloud resources, our framework, when deployed, will be open source for deployment on any cloud platform.

V. Framework Architecture

The top-level architectural components and interfaces comprising our web services framework for wireless sensor networks are illustrated in Figure 1 and listed here:

(1) REST API
(2) REST Process Server
(3) HTTP Client Server
(4) Example Sensor Server
(5) Example Data Processing

As described previously, there are two types of users: (1) data producers that deploy one or more WSNs and install them into the system for private, group, or public use and (2) data consumers that subscribe to and use WSN data shared by data producers.

The REST API forms the interface to the core services provided by the REST process server. The REST process server abstracts a set of resources that it manages and allows users to use those resources through REST request messages. All user API keys and cloud resources including the sensor database and compute cluster are allocated and managed by the REST process server.

The HTTP client server provides a web portal for users to create an account in the system and acquire an API key for using the REST API. The portal also implements a dashboard of functions for data producers to get, add, modify, and delete the resources representing their WSNs. Additionally, with their API key, users can perform these WSN administration functions directly from programs they may write.

The example sensor server runs on a WSN gateway node. It collects sensor samples from the attached sensor nodes and streams them to the REST process server where they are recorded to the sensor database. Users can access the live data or recorded data through the HTTP client server's web portal or from their programs. Live data and recorded datasets can be displayed. A user provided data processing function can be applied to datasets and the results displayed.

The test case WSN used in the project collects weather data from temperature, pressure, and humidity sensors. The sensor data is aggregated, time stamped, location stamped, and streamed into the framework where it is recorded to cloud storage and made available to users on-demand for inspection or for processing on a compute cluster.

A. REST API

The REST API forms the programming interface to the framework's set of web services. A RESTful [15] interface has client and server roles where clients initiate requests to servers and the servers process the requests and return responses. The requests and responses are formed into messages. The server manages resources that are addressable through the client requests. The representation of a resource and its state is captured in a document within the messages, typically in eXtensible Markup Language (XML) or

 ISBN: 978-1-61208-337-7

JavaScript Object Notation (JSON). Some REST interfaces support one or the other or both. The current specification of our framework uses the JSON format for the client request messages and the server response messages.

Requests and responses are typically processed using HTTP. Clients initiate requests using the HTTP request methods GET, POST, PUT, and DELETE. The client session transitions through its states based on the resource information returned from the server. This design pattern frees the server from the complexity of maintaining client and user interface state information, simplifying server logic and increasing server robustness, reliability, and scalability.

The framework's web services are abstracted into a set of resources where each resource is addressed by a unique Uniform Resource Locator (URL) and the operations on a resource are defined by the HTTP methods requested against the URLs. For GET methods, query parameters attached to the URL further define the data requested from the resource. For POST and PUT methods, the HTTP message body will contain the data to be transferred to the resource. The framework's root URL for accessing resources and services is: https://<FQDN>/api/<key>. FQDN indicates a Fully Qualified Domain Name, for example, www.example.com. Figure 2 illustrates the hierarchy of the resources comprising the framework and following is a description of each.

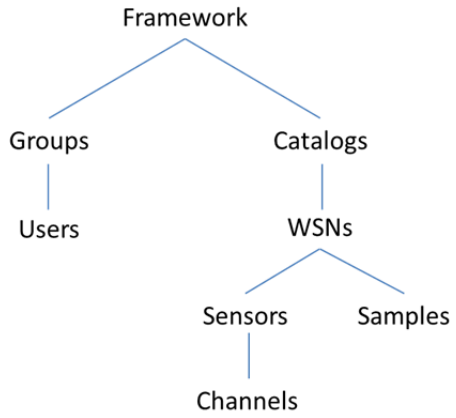

Figure 2. Framework Resource Hierarchy.

(1) Groups – A group resource is created by an admin user. A group is used to manage a group of users that share access to the same framework managed resources. Group services include listing all groups, creating and deleting a group, getting and updating a group's attributes, and adding and removing users to/from a group. The group resource was not implemented in the concept demonstration. The group resource URL relative to the root is: /groups/<groupid>.

(2) Users – Anyone with programmatic access to the REST API is a user. A program that sends a request message to the REST API must provide a user API key in the request message URL. API keys are created by an admin user that has an admin API key. The admin user makes a request on the REST API to create a user, and a unique user API key is returned. The HTTP client server web portal automates this process through the user account sign up process. User services include listing all users, creating and deleting a user, and getting and updating a user's attributes. The user resource URL relative to the root is:
/groups/<groupid>/users/<username>

(3) Catalogs – All resources offered to a user by the framework are organized into a hierarchy. At the top of this hierarchy is the catalog resource. The framework is currently architected with a single catalog, however, it can be extended to provide multiple catalogs for deployments that may wish to implement a "marketplace" of disparate catalogs distinguished, for example, by different legal agreements and terms and conditions of service. Catalog services include listing all catalogs, creating and deleting a catalog, getting and updating a catalog's attributes, and adding and deleting WSNs to/from a catalog. Catalog services were not implemented in the concept demonstration. The catalog resource URL relative to the root is: /catalogs/<catalogid>.

(4) WSNs – The top-level resource in a catalog is a wireless sensor network. A user that creates a WSN is considered a "data producer" and the "owner" of the WSN. The user can choose to make their WSN private, shared within a group of users, or shared with the public (all users). In the framework architecture the "sensor server" runs on a typical WSN gateway device where the WSN sensor devices stream their data to the WSN gateway for local storage or transmission over a network, in this case, transmission to the REST process server over the Internet. The WSN resource encapsulates information about the WSN gateway where the sensor server software will run. For example the initial GPS coordinates of the WSN gateway are specified when the WSN resource is created and they can be updated from time to time for a mobile WSN. WSN services include listing all WSNs available to the user, creating and deleting a WSN, getting and updating a WSN's attributes, adding and removing sensors to/from a WSN, recording samples to the WSN's sensor database, viewing live data from the WSN, viewing the WSN's recorded data, and applying a data processing function to a recorded WSN dataset. Applying data processing to the live stream was not implemented in the concept demonstration. The WSN resource URL relative to the root is:
/catalogs/<catalogid>/wsns/<wsnid>

(5) Sensors – The sensor resource encapsulates the identity, location, and sampling information about the sensor data channels that it physically contains and serves. Information about each sensor is captured when a sensor is created and it can be updated from time to time, for example the sampling frequency and the location for a sensor in a mobile WSN. Sensor services include listing all sensors owned by the user, creating and deleting a sensor, getting and updating a sensors's attributes, and adding and removing channels to/from a sensor. The sensor resource URL relative to the root is:
/catalogs/<catalogid>/wsns/<wsnid>/sensors/<sensorid>

(6) Channels – Sensor channels are the sources of sensor data in the framework. The channel resource encapsulates the data type and data unit for a sensor channel. For example a data type could be "Temperature" and the data unit could be "Celsius". Channel services include listing all channels owned by the user, creating and deleting a channel, and

 ISBN: 978-1-61208-337-7

getting and updating a channels's attributes. The channel resource URL relative to the root is: /catalogs/<catalogid>/wsns/<wsnid>/sensors/<sensorid>/channels/<channeleid>.

(7) Samples – A sample resource is the only resource that is not created by the framework (unless a simulation capability were added). A sample is the set of sensor channel values captured at any point in time, and location, by the sensor server according to the sample set configuration and the sample frequency. The sample set can be all of the sensors connected to the sensor server or a subset. The sensor server collects the sample set, adds a time stamp, adds a location stamp (GPS location and altitude and local grid location and altitude relative to GPS), and posts the data to a WSN.

B. REST Process Server

The REST process server component of the framework identified previously in Figure 1 is implemented in our concept demonstration with the software stack illustrated in Figure 3 below. At the top of the stack there are the framework's web services that we developed and integrated with the lower layers. The web services layer uses the services of the lower layers to implement all resources and services exposed by the REST API. Internally, it manages the sensor database and compute cluster for each WSN and schedules the data processing functions and returns results as directed by REST requests. Views of live sensor data, recorded sensor data, and processed sensor data are composed by the REST process server and returned in REST responses in JSON format.

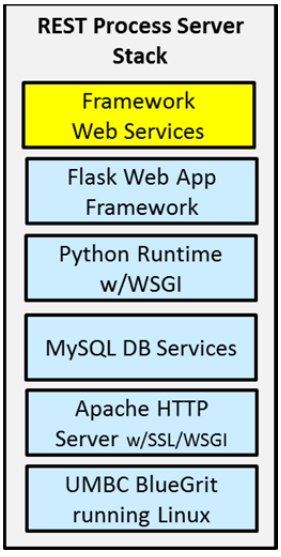

Figure 3. REST Process Server Stack.

For the concept demonstration, we utilized the resources of the BlueGrit computing platform at UMBC. The REST process server stack is built on Linux using open-source software. At the foundation is Apache HTTP Server. Secure Sockets Layer (SSL) encryption was enabled and utilized to secure all messages passing over the REST API. For database services, MySQL [16] was used. One database serves the framework and one database serves each WSN added to the system. Python was chosen for development of the framework's web services and Flask [17] was chosen to provide the required web application framework for deploying our REST web services. Flask is open source and implements the Web Server Gateway Interface (WSGI) 1.0 specification.

C. HTTP Client Server

The HTTP client server component of the framework identified previously in Figure 1 is implemented in our concept demonstration with the software stack illustrated in Figure 4 below. At the top of the stack is the framework's web portal that we developed and integrated with the lower layers. The web portal layer uses the services of the lower layers to implement the graphical user interface where users sign up for accounts, login into their account, and use a dashboard to manage their WSN deployments and to access and create views of WSN data and to process data and view the results. Internally, the web portal makes REST API requests to the REST process server on behalf of the user. The code that implements this interface is encapsulated in a PHP module that we installed into Drupal [18]. Drupal is a modular open-source Content Management System (CMS) framework written in PHP Hypertext Preprocessor (PHP).

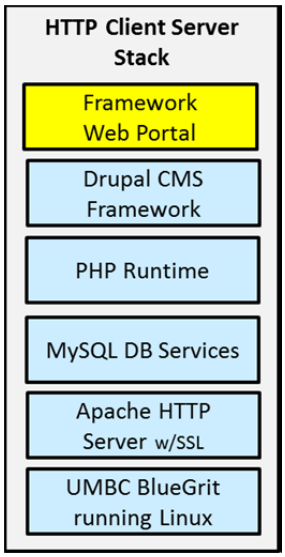

Figure 4. HTTP Client Server Stack.

For the concept demonstration, we again utilized the resources of the BlueGrit computing platform at UMBC for the HTTP client server, although there is no requirement that this server and the REST process server be on the same platform as long as they are both connected to the Internet. The HTTP client server stack is also built on Linux using open-source software. Also, at the foundation, is Apache HTTP Server. SSL encryption was enabled and utilized to secure all information passing between the user's web browser and the web portal. For database services, MySQL was used. A single Drupal database holds all the web portal

 ISBN: 978-1-61208-337-7

content and the module that we added to Drupal adds a table to that database.

The use of Drupal for the web portal immediately solves the problem of putting a "web face" on the service without reinventing all of the wheels that comprise a professional user-friendly dynamic website, which is a fundamental requirement that we have established for our framework. And because Drupal is modular, you install and use what you need. Only the core set of five Drupal modules (System, User, Node, Block, and Filter) and two contributed modules for enabling SSL, are required for the framework's web portal. Figure 5 is a screenshot of the web portal home page.

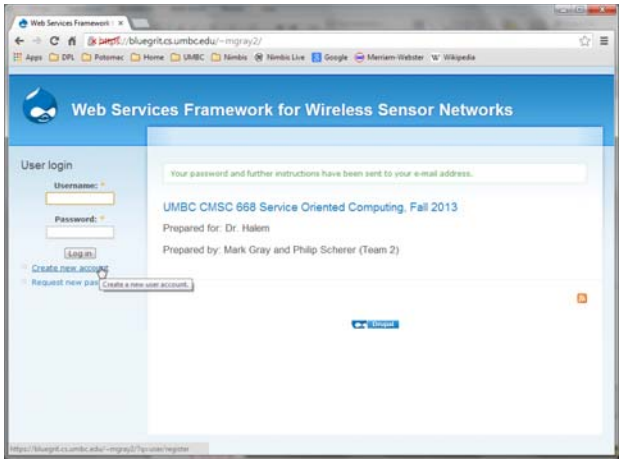

Figure 5. Web Portal Home Page.

The second and more important problem that Drupal immediately solves is the user signup and account management problem. User signup, authentication, login, and password management are entirely implemented with the core User module. When user authentication is complete and a user is created in the web portal, a REST request with the admin key is sent to the REST process server and an API key is allocated, returned, and made available to the user through their account on the web portal.

In addition to meeting these two fundamental requirements (a professional web face and user account sign up), a deployment of the framework's web portal could leverage the work from thousands of contributed Drupal modules, depending on the specific needs of the use case. For the hobbyist and researcher use cases identified previously, a deployment for these users could add profiles, forums, blogs, and other social networking tools for user interest discovery and collaboration. For the commercial use case, the open-source Ubercart [19] suite of Drupal modules could be added which comprise a complete end-to-end ecommerce workflow that integrates with several payment processing service providers. The commercial developers can create a catalog, add products, add terms and conditions, and build a shopping cart that buyers can take to checkout where their services are deployed. As of November 11, 2013 the Drupal developer community reached 30,000 with over 24,000 contributed modules [20].

D. Example Sensor Server

The example sensor server component of the framework identified previously in Figure 1 is implemented in our concept demonstration with the software stack illustrated in Figure 6 below. In this example test case, the sensor server is a "weather sensor server". This example is intended to be the "hello, world" for an initial test of a sensor server in a framework deployment. As such, it does not rely on actual physical sensors and the associated problems of procuring, installing, and getting the sensors to work just to test the framework. Instead, we rely on sensors that are already deployed with their data available to us on a REST API which we will inject into our system as if it were data collected on a deployed WSN. We used the Weather Underground (Wunderground) REST API [21] on a test account we setup that utilized their free level of service which was sufficient for both integration testing and the demonstration without exceeding the free usage levels.

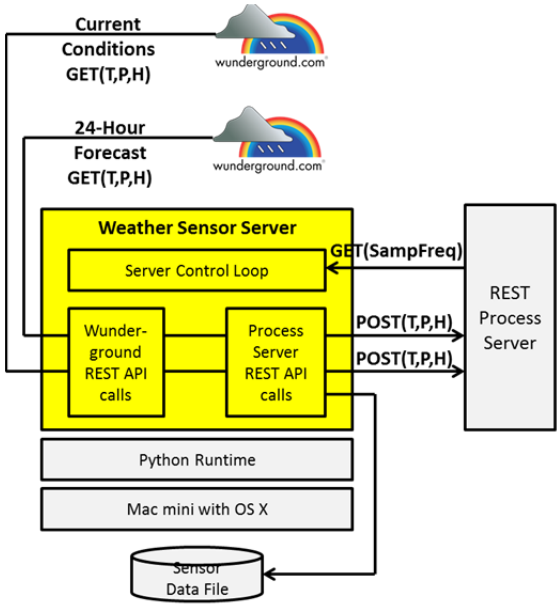

Figure 6. Example Sensor Server Stack.

Our weather sensor server was written in Python and executed on a Mac mini with OS X. A main control loop first updates its sampling frequency by a query to the REST process server where all WSN configuration parameters are maintained. It then sleeps for a time equal to the sample period. When it awakens it makes two calls to the Wunderground API for the current GPS location (in our demonstration the location is static): (1) the current weather conditions for the current location and (2) the 24-hour forecast for the current location. From the JSON responses, the temperature (T), pressure (P), and humidity (H) are extracted for both cases. The current(T, P, H) represent sensor 1 with three channels of data and the forecast(T, P, H) represent sensor 2 with three channels of data. Using the test user's API key and the resource IDs assigned by the web portal, two REST requests are made to the REST process server: (1) a POST of the current(T, P, H) with a timestamp

 ISBN: 978-1-61208-337-7

equal to the current time and (2) a post of the forecast(T, P, H) with a timestamp equal to the current time plus 24 hours. In our demonstration, we set the sample frequency to 12 times per hour, or once every five minutes and collected data for 151 hours. In addition to pushing the data out on the REST API, we also logged it to a text file in Comma Separated Value (CSV) format for testing and integration.

E. Example Data Processing

The example data processing component of the framework identified previously in Figure 1 is implemented in our concept demonstration with the program illustrated in Figure 7 below. In this example test case, the data processing program is a "weather data processing" program. Like the example sensor server, this example is intended to be the "hello, world" for an initial test of a data processing program in a framework deployment.

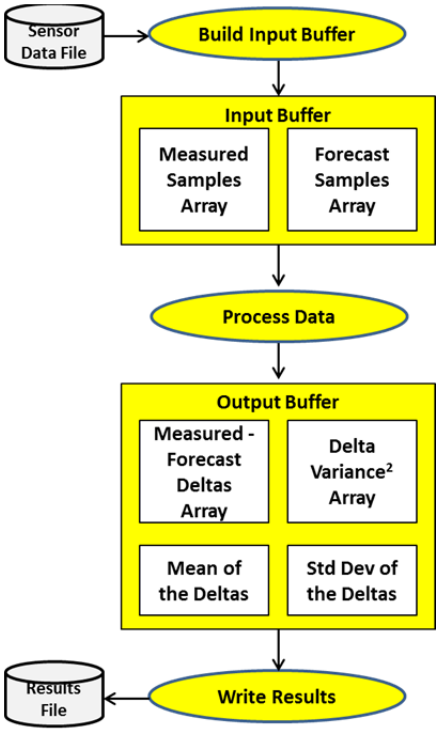

Figure 7. Example Data Processing Program.

The input to the data processing program is a sensor data file containing a dataset extracted from the sensor database. In a completely integrated system, the sensor data file would be pulled from the sensor database by the REST process server after a request to apply the data processing program to the data. The REST process server would place the file into a shared file system available to a compute cluster and launch the data processing program on that cluster. In our concept demonstration, we took the sensor data file that was recorded on the sensor server and uploaded it to a BlueGrit compute blade and executed the data processing program on the data.

The data processing pipeline shown in Figure 7 illustrates a general input -> process -> output dataflow. For this test case, the sensor data file contains 175 hours of data collected on two sensors with 12 samples collected per hour per sensor. Each sample contains a timestamp and the current temperature, pressure, and humidity for that sensor. Each sensor 1 sample contains the actual temperature, pressure, and humidity at that time. Each sensor 2 sample contains the 24-hour forecasted temperature, pressure, and humidity for that time. The "build input buffer" function averages the 12 samples for each one hour time slot and creates two arrays indexed by hour; the "measured samples array" from the sensor 1 samples and the "forecast samples array" from the sensor 2 samples. The data in the input buffer is then processed.

The "process data" function consists of four computations with the results of each computation saved to the output buffer. They are:

(1) Compute the deltas between measured and forecasted
(2) Compute the arithmetic mean over the deltas array
(3) Compute the delta variance^2 array
(4) Compute the standard deviation

The "write results" function summarizes and formats the contents of the output buffer and writes it to a text file where the REST process server picks it up.

Note that the first 24 hours (hour 0 through hour 23) of the 175 hours of data collected have no 24-hour forecast values for comparison and the last 24 hours (hour 151 through hour 174) have no measured values, therefore, the actual computable dataset is 127 hours of data from hour 24 to hour 150.

VI. PROJECT RESULTS

We successfully completed the initial design and implementation of each framework system component and demonstrated the functionality of each component separately, with partial integration of the HTTP client server with the REST process server. The following demonstrations of the REST API specification were presented:

(1) User account creation
(2) User dashboard walkthrough
(3) Sensor server demonstration (of live data)
(4) Sensor data processing (of recorded data)

Implementation consisted of installing and configuring the open-source components of the server stacks and software development of key components. The web services in the REST process server stack consisted of a Python application that we developed (about 1000 lines) and installed on Flask. The web portal in the HTTP client server stack consisted of a PHP module (about 1100 lines) that we developed and installed in Drupal. The example sensor server consisted of a Python application that we developed (about 130 lines) and installed on an Internet connected Mac mini. The example data processing program consisted of a C program that we developed (about 900 lines) and executed on a BlueGrit compute blade. The computation results of the data processing program are presented in Figure 8. A performance comparison between a Windows laptop and a BlueGrit blade is presented in Figure 9.

 ISBN: 978-1-61208-337-7

```
Data window begin time       = Tue Nov 26 12:06:49 2013
Data window end time         = Tue Dec 03 18:40:59 2013
Data window hours total      = 175
Forecast range (hours)       = 24
Hours of data processed      = 127
Number of samples processed = 9144
Mean of Temperature Deltas   = -0.731508 Fahrenheit
Mean of Pressure Deltas      = -0.013176 inHg mslp
Mean of Humidity Deltas      = 3.023204 %rH
StDv of Temperature Deltas   = 2.135467 Fahrenheit
StDv of Pressure Deltas      = 0.046519 inHg mslp
StDv of Humidity Deltas      = 8.797242 %rH
```

Figure 8. Data Processing Computation Results.

Windows 7 Laptop, Intel Core i7 @ 2.7Ghz, 8GB RAM	Performance on UMBC BlueGrit Intel01 Blade
= 0.026 seconds	***= 0.015 seconds***
= 2.84 usecs/sample	***= 1.64 usecs/sample***

Figure 9. Data Processing Performance Results.

VII. FUTURE WORK

Looking at this project as a set of sequential phases, this initial phase represents a three month concept study culminating in a concept demonstration which we have documented in this paper. Future work would address incomplete areas in the concept study and include research on new capabilities.

A. *Incomplete areas to address*

- Complete the integration of the concept study components
- Demonstrate parallel computation on the sensor data
- Implement the group resource and services
- Demonstrate on a commercial public cloud (Amazon)

B. *New capabilities to research*

- Cloud scalability, elasticity, load balancing
- Mobile WSN demonstration
- Data processing library creation and sharing
- Virtualization middleware on the sensor server and sensors
- Data processing on multiple input sources and types
- Data processing on live sensor streams
- Compute instance type and cluster size selection
- Network Protocol Time (NTP) on the sensor server

VIII. CONCLUSION

In conclusion, we successfully demonstrated an architecture and initial implementation of a web services framework for wireless sensor networks. A test case WSN was simulated on a Mac mini pulling actual real-time weather data from the Wunderground REST API and feeding it into the framework. Each framework component was individually constructed, tested, and demonstrated. The cloud storage and compute resources were provisioned from the UMBC BlueGrit computing platform. Future work includes end-to-end integration and testing of all components, the demonstration of parallel computation, the implementation of user groups, and a demonstration on a commercial public cloud. Other future work includes several areas of research, notably cloud scalability, mobile WSN, data processing libraries and sharing, and virtualization middleware extending the concept of cloud computing into the WSN domain.

IX. ACKNOWLEDGMENT

The authors thank Dr. Milton Halem for his direction and encouragement over the course of this project and his teaching assistant Lawrence Sebald for his support in the installation and configuration of various software services on the UMBC BlueGrit computing platform.

X. REFERENCES

[1] W. Dargie and C. Poellabauer, Fundamentals of Wireless Sensor Networks: Theory and Practice, Wiley, 2010.

[2] A. Cuzzocrea and G. Fortino, “Managing Data and Processes in Cloud-Enabled Large-Scale Sensor Networks: State-Of-The-Art and Future Research Directions”, 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp. 583-588.

[3] R. Govindan, J. M. Hellerstein, W. Hong, S. Madden, M. Franklin, and S. Shenker, “The Sensor Network as a Database”, University of Southern California, 2002, pp. 1-8.

[4] D. Tracey and C. Sreenan, “A Holistic Architecture for the Internet of Things, Sensing Services and Big Data”, 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp. 546-553

[5] S.K. Dash, S. Mohapatra, and P.K. Pattnaik “A Survey on Applications of Wireless Sensor Network Using Cloud Computing”, International Journal of Computer Science & Emerging Technologies, Volume 1, Issue 4, December 2010, pp. 50-55.

[6] M. P. Singh and M. N. Huhns, Service Oriented Computing: Semantics, Processes, and Agents, Wiley, 2005.

[7] Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, Dissertation, University of California, Irvine, 2000.

[8] Python. [Online]. Available: www.python.org. Retrieved Dec 2013.

[9] BlueGrit. [Online]. Available: http://bluegrit.cs.umbc.edu/userdocs.php . Retrieved December 2013.

[10] S. Rea, M. S. Aslam, and D. Pesch, “Serviceware - A Service Based Management Approach for WSN Cloud Infrastructures”, 10th IEEE International Workshop on Managing Ubiquitous Communications and Services 2013, San Diego, March 2013, pp. 133-138.

[11] SensorCloud. [Online]. Available: http://www.sensorcloud.com/system-overview. Retrieved Dec 2013.

[12] K. Fogarty, “Google's Wireless Sensors: Big Data or Big Brother?”, www.networkcomputing.com, May 22, 2013. Retrieved Dec 2013.

[13] B. Proffitt, “Xively Actually Connects Things to the Internet of Things”, www.readwrite.com, May 14, 2013. Retrieved Dec 2013.

[14] R. Rea, “IBM InfoSphere Streams, Redefining real-time analytics processing”, IBM Software, Thought Leadership White Paper, May 2013, pp. 1-8.

[15] L. Richardson and S. Ruby, RESTful Web Services, O'Reilly Media, 2007.

[16] MySQL. [Online]. Available: www.mysql.com. Retrieved Dec 2013.

[17] Flask. [Online]. Available: http://flask.pocoo.org. Retrieved Dec 2013.

[18] Drupal. [Online]. Available: https://drupal.org. Retrieved Dec 2013.

[19] Ubercart. [Online]. Available: http://www.ubercart.org. Retrieved Dec 2013.

[20] S. Choudhury, “30,000 Developers in Drupal.org and growing…”, [Online]. Available: https://drupal.org/node/2133153. Retrieved Dec 2013.

[21] Weather Underground (Wunderground) API. [Online]. Available: http://www.wunderground.com/weather/api/. Retrieved Dec 2013.

 ISBN: 978-1-61208-337-7