Design and Implementation of RESTful Wireless Sensor Network Gateways Using Node.js Framework

Mehmet Fatih KARAGÖZ, MSc.
Aselsan Inc.
Image Processing Department
Ankara, Turkey
fkaragoz@aselsan.com.tr

Cevahir TURGUT, MSc.
Aselsan Inc.
Image Processing Department
Ankara, Turkey
cturgut@aselsan.com.tr

***Abstract*—With proliferation of Cloud technologies, the need for connecting wireless sensor networks to Cloud servers arose. With easy accessibility to the cloud and the increased processing power of cloud servers, it became possible to gather data from sensor networks at different locations and process large amount of sensor data. A new generation WSN gateways is needed for this purpose in order to connect the sensors to cloud servers. In this paper, a design and implementation method for RESTful WSN Gateways is proposed using a Javascript based framework, Node.js, which uses asynchronous programming model and can handle a large amount of network connections simultaneously. The gateway is designed as an Embedded Linux device, which can handle multiple accesses with both sensors and cloud servers. All communication APIs residing inside the gateway is designed to be RESTful, HTTP API for communication with cloud servers and Constrained Application Protocol (CoAP) for sensor communication. The proposed approach in this paper provides seamless integration between cloud applications and sensors with its well-defined standard API and minimizes the development time of WSN gateways.**

Keywords— WSN, gateway, CoAP, RESTful, Cloud, sensor, Node.js, IoT

I. INTRODUCTION

Nowadays, the idea of monitoring and controlling pervasive physical devices and sensors over internet is becoming popular with help of the recent achievements at internet connectivity, low power electronic devices and wireless communication methods. For interactive communication of wireless sensor devices over internet, a new communication protocol named Constrained Application Protocol (CoAP) is introduced [8]. However, it is not feasible to connect each sensor device to internet because of sensor endurance, power consumption and network congestion handling issues. A gateway device is needed between sensors and internet, which will handle communications with sensors and Cloud servers. In this paper this gateway is proposed as embedded device which communicates sensors in order to get their readings through CoAP and forward sensor data to Cloud servers through a RESTful interface. This proposed design enables users to access sensors and Internet of Things (IoT) devices over Cloud applications. This method transfers network congestion handling problem on sensors to Cloud application which can handle network congestion by the help of scalability and high I/O performance of Cloud technologies.

Cloud servers consist of a large number of computers connected to each other through a high speed network connection residing on internet [1]. This infrastructure provides us high performance computing and high storage capabilities. By means of these features, it becomes possible to connect different sensor networks and IoT devices at different locations, collect enormous amount of data from these devices, store and process these data using Cloud technologies.

Traditional designs propose gateways as a software component running on high performance computers [2][3]. With latest improvements on embedded processor technology and emergence of asynchronous and lightweight software frameworks; it became possible to design embedded gateway devices which can handle a large amount of network connections. The proposed gateway design is build on Node.js framework which depends on Chrome's JavaScript runtime and enables software developers to build fast and scalable network applications easily [4]. Thanks to the modularized design of this framework and available modules on RESTful Web servers and CoAP protocol, the implementation of this design can be performed rapidly and easily.

Representational State Transfer (REST) is an architectural style which has the properties like simplicity of interfaces, visibility of communication, modifiability and portability of components [6][11]. A RESTful API consists of four main HTTP request methods; GET, POST, PUT, DELETE. GET method is used for read operation, POST is for create operation, PUT is for update/modify operation and DELETE is for delete operation. By means of following this architectural style on gateway API design, Cloud application developers can communicate with sensor gateways through a well defined and familiar interface as a Web Service. The gateway should forward these requests taken from the RESTful interface to sensor devices. However, implementing a complete RESTful protocol on low power sensor devices is not feasible in means of power consumption and sensor life time. To overcome this problem, a lightweight UDP based protocol, CoAP, is being developed [7][8]. With the help of CoAP interface, the gateway can forward requests taken from Cloud servers to sensor devices conserving RESTful architecture.

ISBN 978-3-8007-3621-8

II. System Overview

In this section, we explain components residing on Cloud based Sensor Networks; sensor devices, Cloud servers, gateway devices, respectively, followed by a sample usage scenario.

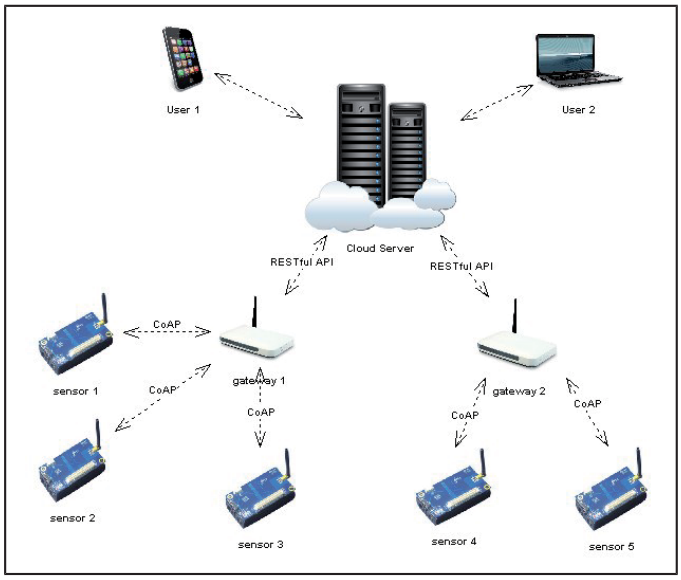

Fig. 1. Sample Usage Scenario

A. Sensor

Sensor devices are wireless low power electronic devices which are connected to gateways through a wireless radio link and gathers environmental data. These devices communicate with gateways using CoAP protocol. A CoAP server is residing on each of these sensor devices and gateway behaves as CoAP client. Gateway sends two types of CoAP requests. The first one is to get sensor information and available resources (a sensor device can contain different types of environmental measurements such as temperature, humidity, wind, etc.). The other one is to get data from available resources on the sensor.

B. Cloud

The main role of Cloud server is to get sensor data through all gateways composing WSN system. By courtesy of internet centric architecture of Cloud servers, gathering sensor data from WSNs at different locations is possible. This large amount of data collected from different gateway devices have to be stored, processed and analyzed. Since Cloud servers have high storage capabilities and high processing power, it is possible to realize storing and analyzing these data, which brings flexibility for data storage and ability to perform complex data mining algorithms [9][10].

Application running on Cloud server communicates with gateways through a RESTful HTTP API. A client-server based architecture exists, where Cloud application is a HTTP client and can be connected to multiple HTTP servers residing on gateways. The reason for using gateway as an HTTP server is providing the instantaneous reaction of commands taken from Cloud application. This approach reduces network traffic by removing the need for polling messages in the reverse scenario where Cloud application is HTTP server and gateways act as clients. Firstly, Cloud application gets information of gateways and their resources (connected sensors and available sensor resources). Secondly, Cloud application can manage resources of sensors through a RESTful API residing on gateways. Cloud application can send following requests to gateway for available resources:

- POST: Initialization of resource tracking
- PUT: Modification of tracking parameters
- DELETE: Termination of resource tracking
- GET: Reading resource data

Gathered, stored and analyzed sensor measurements can be served to WSN end users through any interface that can be implemented on Cloud servers. The comprehensive playground is available for end user application developers with various options on development environments, software frameworks and programming languages.

C. Gateway

The gateway device connects sensor devices to Cloud servers through internet. Our proposed gateway design is an embedded Linux device comprising a network software component built using Node.js framework that can handle a large number of connections with sensors and Cloud servers.

Gateway consists of two main network components, RESTful HTTP server to communicate with Cloud servers, CoAP clients to communicate with connected sensor devices. This design criteria requires the need for gateways to be continuously on-line devices. In our usage scenario, the network has a heterogeneous structure where the gateways are line-powered devices providing continuous connection to Cloud servers. The design and implementation details of these network components are mentioned at GATEWAY ARCHITECTURE.

D. Usage Scenario

A sample usage scenario is shown at “Fig. 1”. In this scenario, two different sensor groups are connected to two different gateway devices through CoAP interface. Cloud application running on Cloud servers is managing these sensor groups and gathering data from them through gateway devices and the RESTful API. Moreover, there are two clients connected to Cloud application to access WSN data. In this scenario, interface between Cloud application and client user applications can be implemented in various ways and implementation details of these interfaces are not given in this paper.

III. Gateway Architecture

A. Overview

Main architectural design goal behind the proposed gateway architecture is making an embedded gateway with scalable network communication interfaces as a bridge between low power sensor devices and Cloud in order to achieve data sharing of sensor networks by using advantages of Cloud technologies. We aimed to achieve this goal by

ISBN 978-3-8007-3621-8

designing a RESTful API for Cloud communication and CoAP API for sensor communication, which brings ease of integration between each other by conserving simplicity of interface and without sacrificing main functionalities.

Embedded Linux as an operating system running on an embedded device and Node.js as the software development framework are used to implement proposed gateway design. Node.js is Javascript based lightweight, asynchronous, event-driven network framework to build fast and scalable network applications [4].

"Fig. 2" depicts main architectural software components of gateway and communication interfaces of these components. Whenever gateway starts operating, firstly, it discovers sensors that communicate with this gateway. Secondly, gateway sends request messages to connected sensors to get general information and resources of these sensors. Finally, gateway waits for new connections from Cloud servers. Cloud server application sends POST request to gateway to initialize sensor resource tracking. Gateway starts to get data of resources on connected sensors after initialization of resource tracking. At this point, sensor network data is available at the gateway and Cloud application can achieve these data by sending GET requests to gateway for requested resource of requested sensor.

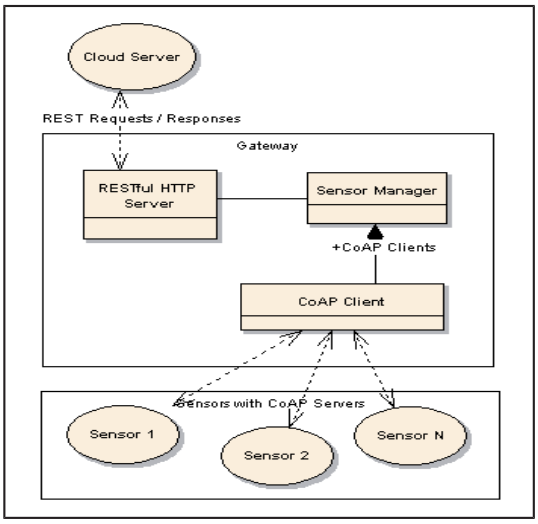

Fig. 2. Main Software Components of Gateway

B. Cloud Interface

Cloud server application does not need to know sensors of WSN system, it is sufficient to know gateways. It gets information about sensors connected to gateways and resources of each sensor through RESTful API which is described in TABLE I. These APIs are implemented using Node.js on our proposed gateway design. The communication between Cloud application and the gateway is built on JavaScript Object Notation (JSON) messages, which has a very clean object type format and can be parsed without effort using Node.js.

Gateway serves information of connected sensors to Cloud application through first API, GET method to get sensor names of gateway, defined in TABLE I. Gateway sends names of connected sensors as JSON.

TABLE I. RESTFUL HTTP API

Method		**API**	
		Parameters	***API Address***
1	GET	None	http://[<gateway_address>]:<port_number>/sensors/
2	GET	None	http://[<gateway_address>]:<port_number>/<sensor_name>/
3	POST	period	http://[<gateway_address>]:<port_number>/<sensor_name>/resource/<resource_name>/<resource_id>/
4	PUT	period	http://[<gateway_address>]:<port_number>/<sensor_name>/resource/<resource_name>/<resource_id>/
5	GET	None	http://[<gateway_address>]:<port_number>/<sensor_name>/resource/<resource_name>/<resource_id>/
6	DELETE	None	http://[<gateway_address>]:<port_number>/<sensor_name>/resource/<resource_name>/<resource_id>/

After Cloud application gets sensor names connected to gateway; it gets information about resources of each sensor by using second API, GET method to get resources of the sensor. Gateway responses to second API calls by sending information of requested sensor including name and location of sensor, number of resources residing at sensor with their types.

Cloud application uses third API, POST method to initialize of resource tracking (e.g. reading temperature from environment) on requested sensor of requested gateway. Parameters of resource tracking are passed to gateway through this API; as an example, data reading period of sensor is set by "period" parameter. After this API call, gateway configures related sensor and starts to read resource data of sensor using CoAP API calls. Parameters of resource tracking like "period" are modified using fourth API, PUT method. After initialization of resource tracking, Cloud application gets latest sensor data by periodical call of fifth API, GET method. Finally, resource tracking of sensor is finalized by calling the sixth API, DELETE method.

Express module of Node.js is used to implement RESTful HTTP server. All API calls are executed asynchronously. Server application in gateway takes API calls and whenever response becomes ready, server application sends response to Cloud application. This method enables us to use CPU of gateway in an efficient way since there is no blocking function calls. Moreover, "Express" module of Node.js eases defining REST API as seen in Fig. 3 from prototype implementation of our gateway design.

ISBN 978-3-8007-3621-8

```
//Load Express module and create server
var express = require('express');
var app = express();
//Create API 1, GET
app.get('/sensors', function(req, res){
    res.send(sensors);
});
//Create API 3, POST
app.post('/:sensorid/resource/:resourcetype/:resourcetypeid/',
    function(req, res){
        var result = performResourceInitialization(req);
        res.send(result);
    });
//Create API 6, DELETE
app.delete('/:sensorid/resource/:resourcetype/:resourcetypeid/',
    function(req, res){ //...
    });
//Create API 5, GET
app.get('/:sensorid/resource/:resourcetype/:resourcetypeid/',
    function(req, res){
        var result = getSensorResourceValue(req);
        res.send(result);
    });
//... other implementations
//Listen connections from Cloud application
app.listen(3000);
```

Fig. 3. Sample RESTful Webserver Code written using Node.js Express Module

C. Sensor Interface

In the proposed design, the communication between Cloud application and sensor devices is transparent, which is handled by the gateway. Gateway communicates to sensor devices through CoAP interface as a CoAP client. The implementation of gateway design is built using Node.js framework and for the CoAP client part, a Node.js module named "Node-coap" is used. All the communication between sensor devices and the gateway is also built on JSON type messages. Sensor devices in our proposed design are assumed to have a CoAP server with following GET type requests, given in TABLE II.

TABLE II. SENSOR COAP API

Method		API	
		Parameters	*API Address*
1	GET	None	coap://[<sensor_address>]/info/
2	GET	None	coap://[<sensor_address>]/resources/
3	GET	None	coap://[<sensor_address>]/resource/<resource_type>/<resource_type_id>

The sensor manager part of our gateway design keeps the information about sensors as an array, where the address information, general information about sensors and available resources of these sensors are kept. An example of how the sensor information is hold is given on Fig. 4. As it is seen on Fig. 4, a sample sensor has three information parts, "address", "info" and "resources". The "resources" are the available measurement types of a connected sensor and its sub elements, where a type of measurement can be accomplished by a number of same type sensors to achieve a more reliable measurement.

```
sensors["sensor1"] = {
    address: "10.0.0.10",
    info: {
        name: "sensor1",
        location: "Ankara"
    },
    resources: {
        humidity: {elements: [1,2,3]},
        temperature: {elements: [1,2]}
    }
};

sensors["sensor2"] = {
    address: "10.0.0.12",
    info: {
        name: "sensor2",
        location: "Istanbul"
    },
    resources: {
        temperature: {elements: [1]}
    }
};
```

Fig. 4. Sensor Data Structure on Gateway

When a new sensor device discovered by our gateway device, an empty device structure is initialized and the requests are fired to receive detailed information about that sensor device. Firstly, "info" request (first method in TABLE II. is sent to get the information about the sensor, and respectively the info part of that sensor is filled as seen in Fig. 5.

```
var req = coap.request(sensors[sensor_id].address + '/info');

req.on('response', function(res) {
    ParseSensorInfo(sensors[sensor_id], res);
});

req.end();
```

Fig. 5. CoAP "info" request to sensor device

Following that request, "resources" request (second method in TABLE II. is sent to get the information about available resources of sensor device and the appropriate part of sensor information table is filled as seen in Fig. 6.

```
var req = coap.request(sensors[sensor_id].address + '/resources');

req.on('response', function(res) {
    ParseSensorResources(sensors[sensor_id], res);
});

req.end();
```

Fig. 6. CoAP "resources" request to sensor device

After this operation, the new sensor device becomes ready to be served to Cloud application. Moreover, when the Cloud application initializes a resource tracking operation for a sensor device, a timer with a period defined by Cloud application is set and sensor data reading requests (third method in TABLE II. are called periodically. The readings are parsed periodically as seen in Fig. 7 and the measurement data is available for Cloud application with requested update period, which can be read by Cloud application through the HTTP API of the gateway device.

ISBN 978-3-8007-3621-8

```
var data_address = sensors[sensor_id].address + '/resource/' +
                   resource_type + '/' + type_id;

var req = coap.request(data_address);

req.on('response', function(res) {
    ParseSensorData(sensors[sensor_id], res);
});

req.end();
```

Fig. 7. CoAP data reading request to sensor device

Iç. Implementation and Results

In this section, we explain the prototype we implemented for performance measurements. The gateway prototype is implemented using an embedded board IGEP v2 [5] which has a ARM Cortex A8 processor running at 1Ghz and 512 MB of RAM. Node.js framework is cross-compiled and installed on this board with its modules “express” and “node-coap”, for RESTful HTTP server and CoAP client, respectively. Two main tests were performed on this prototype, one for testing the cloud interface of gateway application, and one for testing the CoAP based sensor communication.

Firstly, for the test of sensor communication, we implemented a CoAP server running on computers using Ubuntu 12.04 operating system, to simulate the sensor devices. As we mentioned in design part, complete CoAP API for this sensor simulator is implemented, which can send information, resources and measurements to our gateway prototype. For this performance test, the simulator sensor has a humidity sensor and the measurement period of gateway is altered to different values in order to measure the packet performance of software. Details of the packet performance test are given on TABLE III. Request period of sensor measurement readings are changed from 1000ms to 10ms and expected number of packets at sensor devices are compared with the actual number of incoming packets per second. Each test is performed for 600 seconds with JSON messages containing measurement values and average incoming packet numbers per second are measured. The JSON messages sent through the CoAP protocol. As we see on the TABLE III. , the gateway device can send the exact number of packets expected above 100ms period. Under this period value, success rate starts to decrease and we see %77 value at 10ms period. As we are working on an event-queue based framework, the event-queue would be crowded by decreasing the period and parsing operations increase the processing time for each response event in the event-queue. This increased processing time of events brings a lower success rate at lower periods. However, the gateway sensor communication works flawlessly below 20Hz measurement frequency. If there is a need for strict number of measurements from sensors, the proposed gateway design can be used below 20Hz without any issues (e.g. latency, packet loss). Furthermore, above 20Hz, the success rates are acceptable for most sensor network scenarios.

TABLE III. SENSOR INTERFACE PERFORMANCE

Request Period (ms)	Expected Packets	Incoming Packets	Success Rate
1000	1	1	%100
500	2	2	%100
250	4	4	%100
100	10	10	%100
50	20	19	%95
40	25	23	%92
25	40	36	%90
10	100	77	%77

For testing the RESTful cloud interface of the gateway device, we used Apache Bench v2.3 (ab) as the HTTP client running on computer using Ubuntu 12.04 operating system and we performed regression tests on gateway HTTP server. We used a total number of 10000 packets for each test with different concurrency levels, and we measured the average response time per request and number of lost packets for each request. For all test scenarios, the number of lost packets is zero. The gateway software and the RESTful HTTP server module “express” was able to handle all the requests and successfully answered these requests. The values of average response time at different concurrency levels are given in Fig. 8. Average response time starts with 4.5 ms at concurrency level 1, and for different concurrency level we observed similar response times between 3.5 – 4 ms. These results show that the cloud interface of our gateway design has a stable average response time at different request loads. When there is no concurrent connection (concurrency level 1), we observe the highest average response time, because the client waits for a request to completed before sending another and this round-trip time contributes to average response time.

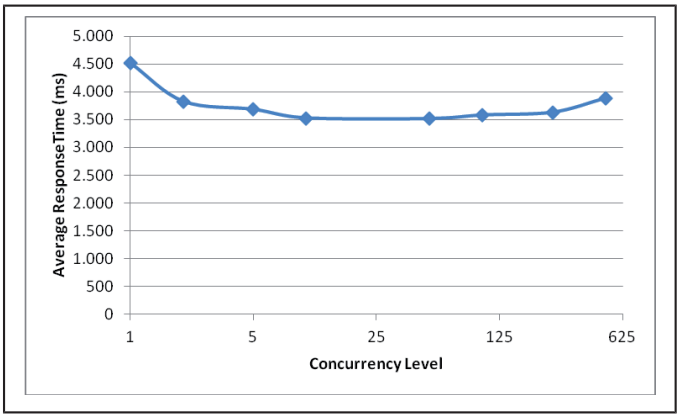

Fig. 8. Cloud Interface Performance Results

ç. Conclusion

By our proposed method, Different WSNs at different locations can be used by cloud applications and large amount of data belonging to multiple networks can be processed without issues. The gateway device creates a transparency between sensor devices and cloud application. Cloud application can set the properties of sensor measurement requests such as measurement period, however there is no

ISBN 978-3-8007-3621-8

direct communication between sensors and cloud application. This design brings the advantage of preventing the regression on low-performance sensor devices and moves this regression to high-performance gateway device which can handle this regression, thanks to asynchronous event-queue based software structure.

A well defined API is designed which brings seamless integration with cloud applications and minimized the development time. The cloud application developers, who are good at implementing web services, can use WSNs as web services through this well defined RESTful HTTP API. By courtesy of the CoAP interface of sensors, the development time inside the gateway design is minimized as the whole system is built on RESTful protocols at sensor and cloud interfaces.

REFERENCES

[1] M. Carroll, P. Kotzé, A. Merwe, "Securing Virtual and Cloud Environments", In I. Ivanov et al. Cloud Computing and Services Science, Springer Science, 2012, pp. 73-90.

[2] V. Trifa, S. Wieland, D. Guinard, T. Bohnert, "Design and Implementation of a Gateway for Web-based Interaction and Management of Embedded Devices", DCOSS, 2009.

[3] M.A. Islam, F. Belqasmi, R. Glitho, F. Khendek, "Implementing OMA RESTful location services in wireless sensor environments", IEEE Symposium on Computers and Communications (ISCC), 2012

[4] Node.js: http://www.nodejs.org/

[5] IGEP v2 embedded board: https://www.isee.biz/products/igep-processor-boards/igepv2-dm3730

[6] T. Erl, B. Carlyle, C. Pautasso, R. Balasubramanian, "SOA with REST", 1st ed, Prentice Hall, 2013, pp. 51-67.

[7] W. Colitti, K. Steenhaut, and N. De Caro. Integrating Wireless Sensor Networks with the Web. In Proc. IP+SN, Chicago, IL, USA, 2011.

[8] IETF CoRE Working Group, "Constrained Application Protocol (CoAP) Internet-Draft", ver. draft-ietf-core-coap-18, 2013.

[9] S. Li, L. Xu, X. Wang, J. Wang, "Integration of hybrid wireless networks in cloud services oriented enterprise information systems", Enterprise Information Systems, vol 6, issue 2, pp. 165-187, 2012.

[10] B. Shen, Y. Liu, X. Wang, "Research on data mining models for the Internet of Things", Image Analysis and Signal Processing, vol. 1, pp. 127-132, 2010.

[11] Fielding, Roy T., Taylor, Richard N., "Principled Design of the Modern Web Architecture", ACM Transactions on Internet Technology, vol. 2, issue 2, pp. 115–150, 2002.

ISBN 978-3-8007-3621-8