Simple RESTful Sensor Application Development Model Using CoAP

Girum Ketema Teklemariam, Jeroen Hoebeke, Floris Van den Abeele, Ingrid Moerman, Piet Demeester
Department of Information Technology
Gent University – iMinds
Gent, Belgium
{firstname.lastname} @intec.ugent.be

Abstract— *The wireless communication capability of sensors and actuators made them suitable for several automation solutions which involve sensing physical properties and acting upon them. These days, gateway or cloud based sensor/actuator interaction models are widely used. In this model, every sensor/actuator interaction goes through the gateway or via the cloud. In order to realize the true Internet of Things philosophy where everything is interconnected, direct interactions between sensors and actuators, also called bindings, are important. In addition to this, alternative IoT application development models which facilitate application development and improve efficiency are required. In this paper, we introduce a CoAP based sensor/actuator binding solution where a 3rd party is responsible for setting up the binding, but is not involved in any of the further interactions. As binding creation and execution is fully based on RESTful CoAP interactions, very flexible bindings between any two devices can be created. Further, the binding concept is extended into the RESTlet concept for introducing (pre-)processing into the sensor/actuator interactions. RESTlets are small application building blocks with internal processing logic and RESTful interfaces for input, control and output. In this paper we present how IoT applications can be created by binding different RESTlets to each other and to sensors and actuators. We implemented these solutions in CoAP++ and Contiki and evaluated the implementation by taking different measures such as delay, memory footprint, and packet size.*

Keywords: IoT application development; Sensor/Actuator bindings; CoAP; Observe Option

I. INTRODUCTION

The communication capability that was recently added to sensor and actuator nodes has made them the future corner stones of the Internet. There are a huge number of applications that make use of sensors and actuators [1][2][3]. However, the networks that interconnect these nodes are usually characterized as Low power and Lossy Networks (LLNs) due to the unreliability of the network. This distinctive feature is basically the result of the constrained nature of the nodes (in terms of processing speed, memory, and power) and their limited radio communication capacity. For a long time, this limited the direct accessibility of these nodes from the Internet. As a way out, many vendors used gateways as intermediaries for all communication between the nodes and the external world. In addition, the communication protocols used in the LLNs were proprietary, restricting interoperability of different solutions even further.

Aware of these limitations, IETF established different working groups to address the limiting factors of constrained devices so that the nodes could run standard network protocols and become accessible directly from any network. 6LoWPAN [4] introduced an adaptation layer just below the IP layer so that IPv6 packets can be successfully transmitted through a wireless sensor and actuator network. IETF also focuses on the application layer to provide lightweight application protocols suitable for constrained devices. The Constrained Application Environment (CoRE) working group was tasked with coming up with such protocols and guidelines. The Constrained Application Protocol (CoAP) [5] is one of the achievements of this working group. The protocol is a lightweight counterpart of the HTTP protocol. The CoAP protocol allows communication with constrained devices in a RESTful way. The working group is also working on further extensions of the protocol. One such extension is the observe option that easily allows monitoring of resource states on sensors. These and other related protocols allow users from the Internet to interact directly with the constrained devices.

In this paper, we present a CoAP-based simple and flexible way to realize direct interactions between sensors and actuators, called *binding*. In addition, we introduce the concept of *RESTlets*, which are IoT application building blocks with inputs, control parameter, basic processing logic and outputs. Bindings are then used as the glue between the RESTlets, sensors and actuators to create basic IoT applications. . The main contribution of this work is twofold the first of which is a novel mechanism to enable direct sensor and actuator interaction from any network by eliminating the need for the intermediary watching over every interaction. The second contribution is a new RESTful application development model based on the binding and RESTlet concept. The concepts described in this paper all build upon the same protocol, CoAP, and RESTful mechanisms to achieve goals ranging from simple sensor-actuator associations to IoT application development.

The next section describes the sensor/actuator direct interaction challenges followed by the current IoT application development issues which motivated us to propose the solutions presented in this work. Section four briefly discusses the protocol that lies at the heart of the proposed systems, CoAP, and two of its extensions, Observe and Conditional Observe. Section five discusses the binding solution and section six describes RESTlets. Section seven and eight elaborate on the implementation and evaluation of the binding and the RESTlet concepts. Related work will be discussed in Section nine while Section ten concludes the paper by indicating future work.

II. CHALLENGES OF SENSOR/ACTUATOR BINDING

In order to take full advantage of the communication capability of sensor and actuator nodes, it is important to make them accessible from the Internet. Different solutions have been proposed to do so. Most of the solutions use third-party devices, usually a gateway or a cloud service, to control sensor/actuator interactions. This device or service handles the collection of sensor events and generation of actuator triggers. In other words, every sensor sends its data to the 3rd party and the 3rd party generates the appropriate triggers. Given the possibility of interconnecting everything in the current Internet of Things setup, we can see several limitation of this approach. First, users may need to initiate and control sensing and actuation from any device or any network. For instance, users may want to directly change the lighting settings of a home automation system from the Internet using their smartphone. Second, the intermediate node has to be always online to provide the required binding functionality. If the device or service fails, the sensor/actuator interaction will be disrupted making the device or the service a single point of failure for the whole network. In large networks, where several sensors and actuators are engaged in frequent interactions, sending all packets to the gateway or cloud service may introduce additional delays or network congestions.

Direct interaction of sensors and actuators without the involvement of a third party is an alternative that overcomes most of the aforementioned issues. One such solution is pre-programming the sensor/actuator bindings when deploying the network and reprogramming them whenever the bindings have to be changed. This solution eliminates the intermediary from the network but it is inflexible and may not be applicable for all use cases. Another solution is creating bindings by putting them in physical proximity and initiating a coupling procedure. This solution also works for initial setup but lacks the flexibility of changing the binding thereafter. Other solutions only allow bindings between devices that have well-defined interfaces [7], limiting the Internet of Things vision that every device can interact with any other device.

In this paper, we propose a CoAP based flexible sensor/actuator binding solution that resolves the limitations discussed above. The proposed solution allows direct sensor/actuator interactions, thereby removing the dependence on gateways or cloud services to coordinate the interaction between these constrained nodes. In addition, interfaces for easy manipulation of bindings will make creation of and control over bindings easy and flexible. This concept makes sensor and actuator nodes smarter as they can directly interact with other components. In addition, as we will show later, the concept can be extended to facilitate the development of IoT applications. This development exhibits several challenges, as we will discuss in the following section.

III. IoT Application Development

Different solutions that make use of networked sensors and actuators may require some form of (pre-)processing to be applied on sensor data before a decision can be made whether an actuator should be triggered or not. An example of such processing could be counting the number of events from different sensors before triggering an actuator. In many cases, this intelligence is implemented at the network gateway or in the cloud. Here, every sensor's data is sent to the gateway or the cloud for further processing, after which a decision is sent back down in the LLN to the actuator. This approach poses similar problems as the binding of sensors and actuators. WS-* or HTTP based RESTful mechanisms are usually used to develop the IoT applications at the gateways or in the cloud. Processing often takes place outside the sensor network requiring an always-on intermediary no matter how trivial the required processing is. In addition, processing often requires programming of all RESTful interactions and processing logic, limiting reuse of processing logic across IoT applications. An interesting alternative to this approach is a model that provides reusable and small application building blocks that can be placed anywhere in the network (at the gateway, in the cloud or in the LLN) and use RESTful mechanisms to interconnect these components to perform the desired processing or build an application. Some initiatives have appeared already that aim to break down IoT applications into small units [21], but they do not achieve a complete separation between processing logic and the RESTful interactions.

In this paper, we propose a simplified IoT application development model based on such application building blocks, which we call *RESTlets* and use bindings to interconnect the RESTlets, sensors and actuators, effectively adding intelligence to normal CoAP-based sensor/actuator interactions.

IV. CoAP, Observe and Conditional Observe

The Constrained Application Protocol (CoAP) [4] is an IETF proposed standard suitable for machine-to-machine or IoT interactions. The protocol works in a similar way as HTTP and implements a minimal subset of REST. Consequently, a mapping between both protocols is possible. CoAP uses the same methods as HTTP when sending requests from clients to servers, namely GET, PUT, POST, DELETE. Since TCP is too resource intensive for constrained devices, CoAP uses UDP with confirmable messages at the transport layer. In a normal client/server interaction, the CoAP client sends a request to a specific resource on a server by using one of the four methods and the server responds with the current representation of the resource of interest (for GET requests) or the appropriate response for the other mechanisms. Fig. 1 shows a typical client/server interaction where the client sends a GET request to receive the current temperature value on the server, represented by the resource /s/t. In the example, the server responds with the latest value, in this case 22.

For resource monitoring applications, clients need to have an up-to-date representation of data from servers. Sending periodic requests to servers (polling) is not an optimal solution for constrained devices. Observe [8] is an interesting extension of the CoAP protocol where clients inform their interest of getting an up-to-date resource representation from servers. After that, servers send

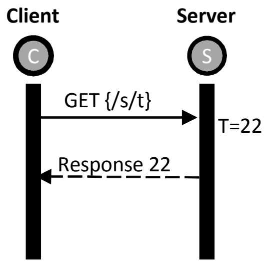


Figure 1: CoAP Operation

notifications whenever resource states change. To establish this observation relationship, the same GET method is used with the Observe option included in the first request (Fig 2). Upon reception of this request, the server notifies the current state of the resource to the client and registers the client for further notification of events. This is an interesting optimization of the protocol that avoids continuous listening for changes. However, there is still a room for further optimization of this approach. In many applications, every state change might not be significant enough to take action. In such cases, clients will drop the packets after comparing the payload against a specific threshold. Dropping packets that are generated by a constrained device and that have travelled through a constrained network is not so optimal. Therefore, further optimization can be obtained through Conditional Observation [9] where clients also send notification criteria when they register for observation. This means that servers will only notify clients if the resource state meets the criteria stated upon registration. Detailed implementation and evaluation of Conditional observation is given in [10]. In Figure 3, the client mentions that it is interested to be notified only when the temperature change results in a value less than 22.

V. DIRECT SENSOR AND ACTUATOR BINDINGS

Direct interaction of sensor and actuator nodes is advantageous for easy deployment, independent operation and management of wireless sensor/actuator networks. In this section we use the interaction of electric light bulb (actuator) and a switch (sensor) in a home automation system as a simple use case. In such systems, when the switch is pressed, the node triggers the actuator to turn on or off the light. To realize this, a traditional gateway-based system that uses RESTful services may be implemented using CoAP with observe option. The initiator, usually the gateway, registers at the sensor (in this case the switch) to be notified whenever the state of a resource representing button presses changes by sending a (conditional) observe request. Whenever such an event occurs, the sensor notifies the gateway by including the current values as payload. The gateway then triggers the actuator to switch the light on or off. In this case, every interaction between the sensor and the actuator is mediated by the gateway.

In our solution, any device connected to the Internet, for example a smartphone, may initiate the binding. The process starts when the initiator sends a GET request to the sensor along with the observe option to establish the binding. However, additional options have to be included in the request to inform the sensor that this is a binding request (not a regular observation request between the sensor and the initiator). Four new options are introduced to carry all the binding related information in the request. BIND_URI_HOST option carries the IPv6 address of the actuator to be notified and BIND_URI_PORT option, if present, indicates the UDP port of the actuator. If not present, the default CoAP server port number is assumed. The third option, BIND_URI_PATH contains the path to the resource of interest on the actuator. Whenever an event occurs, the sensor sends a PUT request to the resource on the actuator identified by the three new options mentioned above. The payload of the PUT request may also be specified by including the BIND_PAYLOAD option, a fourth newly introduced option. If this option is not provided, the current sensor reading will be used.

To summarize, the initiator sends a CoAP GET request to the sensor by specifying the binding information (the four new options) along with the observe option. Upon receipt, the sensor registers the actuator as observer and sends a PUT request to coap://[BIND_URI_HOST]:[BIND_URI_PORT]/BIND_URI_PATH whenever state changes occur. The payload of the request could be BIND_PAYLOAD or the current sensor value. The actuator may take different actions based on the payload. It is also possible to provide observation criteria as per the conditional observe draft. Once the binding relationship has been established, the initiator is no longer involved in further communications between the sensor and the actuator.

For easy management of binding relationships, the sensor may expose its active binding relationships through the

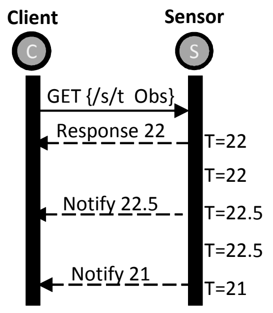


Figure 2: Observe Option

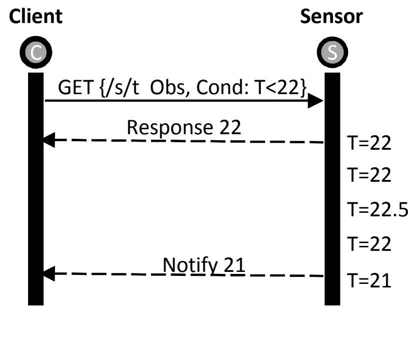


Figure 3: Conditional Observation

/binding resource so that users may modify existing relationships from the Internet via a RESTful interface.

VI. RESTLETS

In this section, we present the concept of RESTlets, small IoT application building blocks. RESTlets are modeled with a set of inputs, control parameters, processing logic and an output (Fig. 4). The inputs may be sensor readings or outputs of other RESTlets, which will be further processed to produce the desired output. The real power of the RESTlet concept lies in the processing logic. Depending on application requirements, the processing logic could be as simple as a negation operation, where the output is the logical inverse of the input, or as complex as an SMS module. The control parameters are configurable values such as phone numbers for SMS applications or a value threshold for a RESTlet which implements a simple less than (<) operator. Inputs, controls, and outputs can have any data type or representation (e.g. JSON, SenML). In fact, RESTlets that convert between different representation formats or data types may also be defined. The number of inputs and control parameters varies, depending on the type or the RESTlet. The RESTlets may be instantiated as many times as possible once they have been defined and implemented. This results in a number of new resources that represent input, output and control.

After the basic application building blocks, or RESTlets, have been defined, the desired IoT application can be programmed by dynamically instantiating the required RESTlets using the CoAP POST method and by binding the different components such as sensors, actuators, and the instantiated RESTlets. Fig. 5 and Fig. 6 show how an application that triggers an actuator when it gets values from two sensors can be implemented using RESTlets. The application requires an AND RESTlet which outputs 1 when both inputs are 1. This logic is programmed once and can be reused as many times as desired. Whenever required, the RESTlet is created by sending a POST request to the node that hosts the RESTlet by specifying its name. In Fig. 5, the two inputs of the RESTlet are connected to the two sensors and the output is connected to the actuator. These interconnections are actually binding relationships created by sending GET requests with the binding options to the different components of the application as shown in Fig 6.

The interaction between sensor and actuator nodes after a simple binding relationship is usually change/trigger interaction. This means, when a sensor value changes a trigger is sent to the actuator. By using RESTlets, intelligence can be added to these simple interactions, which

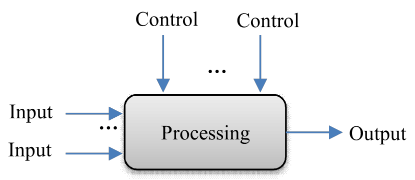
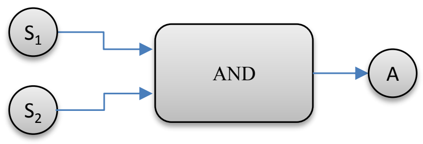


Figure 4: RESTlet Block Diagram



Information	Information	Information
Host: [SENSOR1/2]	Host: [HOST]	Host: [ACTUATOR]
Resource: s/m	Resource: restlet	Resource: a/lt

Figure 5: Block Diagram showing an AND RESTlet

Programming Instructions

```
POST [HOST]/restlet   PAYLOAD: "RN=AND"
                      [RESPONSE: Location-Path = /restlet/2334]
GET [SENSOR1]/s/m  BINDURI: [HOST]/restlet/2334/input/0
GET [SENSOR2]/s/m  BINDURI: [HOST]/restlet/AND_1/input/1
GET [HOST]/restlet/AND_1/output  BINDURI: [ACTUATOR]/a/lt
```

Figure 6: Programming instructions to realize the application

is important to achieve simplified sensor application development using solely CoAP and RESTful mechanisms.

VII. IMPLEMENTATION

A. *Implementation of Bindings*

We used Erbium in Contiki 2.6 to implement bindings on constrained devices [11]. The non-constrained devices were programmed in CoAP++, our own C++ implementation of CoAP and several extensions. The four new options, namely BIND_URI_HOST, BIND_URI_PORT, BIND_URI_PATH, and BIND_PAYLOAD were added to the list of options supported by Erbium and CoAP++. Both sensor and actuator nodes were Zolertia (Z1) nodes simulated in Cooja running Erbium. The initiator runs the CoAP++ code.

B. *Implementation of RESTlets*

To prove the feasibility of the RESTlet concept, we implemented the RESTlets on the gateway using CoAP++. The RESTlets were modeled as C++ classes with their inputs, outputs and control parameters represented as CoAP resources. The number of inputs and controls differs based on the type of RESTlet. As described in the previous sections, the core component of the RESTlets is the intelligence built into them in the form of member functions of the classes. Depending on the RESTlet type, the functions define what to do when an input arrives, when the output is updated or when a timer expires. For all RESTlet types, there is an internal wiring between the RESTlet's input and output variables in such a way that changes to one of the inputs may trigger an update to output values.

VIII. EVALUATION

A. *Evaluation of Bindings*

The basic scenario used for evaluating the binding concept is the interaction between a resource on a light switch (as sensor), identified by /gpio/btn, and a light bulb (as actuator), identified by /lt/on. Pressing the switch is simulated by reading values from a random sequence of 100 0's and 1's. If there is a transition from 0 to 1 or vice versa in subsequent readings, this indicates a button press which will trigger a notification to be sent to the observers. We also used different network topologies to see the impact on performance (Fig. 7). In all cases, we used RPL [12] as routing protocol in the constrained network. All tests were run 10 times for each topology. We compared memory footprint, transmission delay, and packet size of the binding solution against the CoAP gateway-based solution.

1) *Memory Footprint*

The original Erbium code has been modified to support bindings. The modifications include defining, serializing and parsing the new options; and extending the observation table to store the binding information; and a mechanism to check, update and delete bindings through the */binding* resource. All these changes require additional memory space mainly in the code (text) segment and the BSS area. For instance, the Code segment for the gateway-based solution was 48,434bytes and increased to 51,160bytes to support the binding solution. Similarly, the Data and BSS sections also showed a slight increase from 362bytes and 5,760bytes to 380 and 5894 bytes, respectively (considering only 1 observer).

However, this approach has also its own limitation. The memory requirement, specifically the BSS region, of both solutions increases when the number of observers increases. For instance, in our experiment every additional observer requires an additional 232 bytes and 266 bytes, respectively, for the gateway-based and the binding solutions. As memory is scarce in constrained devices, this will limit the number of observers allowed to register at the same time and thus the number of simultaneous bindings that can be supported. Here the gateway solution has an advantage since it may achieve scalability by aggregating multiple observe requests at the gateway avoiding one to one relationships between multiple actuators and a sensor.

2) *Communication Delay*

We calculated the time difference between the occurrence of an event at the sensor and the reception of the PUT packet at the actuator to compute the communication delay. We repeated the test for all three topologies. In our experiment, the gateway-based solution resulted in higher delay in all three topologies. In case of the gateway-based solution, every notification goes all the way to the gateway

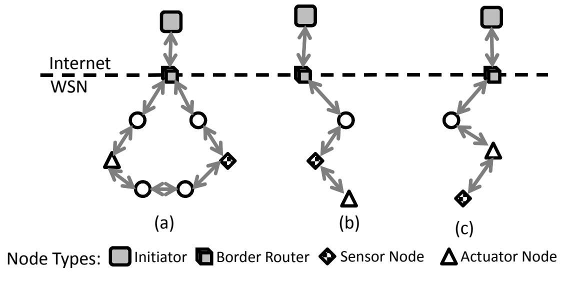

Internet
WSN
(a)
(b)
(c)
Node Types: Initiator Border Router Sensor Node Actuator Node

Figure 7: Topologies - (a) Two hops away (b) Sensor in the Middle (c) Actuator in the Middle

and actuator triggers are sent all the way down to the actuator even if the sensor and actuator are very close in the routing path. This explains the higher communication delay of the gateway-based solution.

3) *Packet size*

Packet sizes larger than the MTU of LLNs may result in fragmentation of packets which leads to sub-optimal solutions. For IEEE 802.15.4 based LLNs, the maximum packet size is 127 bytes [19]. The direct binding solution uses larger packets to establish the relationships, as it has to include the binding information in the request. However, if we use a reasonable resource path, as indicated by the IPSO Application Framework [13], and reasonable sized payload, this size does not exceed this limit. Moreover, this request is sent only once when we want to establish the relationship. Packet sizes of subsequent interactions between sensors and actuators are the same for both solutions. From this, we can conclude that the binding solution does not have a significant impact on the packet size to the extent that affects communication in the network.

B. *Evaluation of Restlets*

1) *Programming Complexity*

We used a lifestyle monitoring application as an example to perform the evaluation. The application has to toggle an alarm light in the house when the resident is not active for 24 hours. The resident is considered to be active when 2 motion sensors (e.g. one in the living room and the other in the hallway) together generate more than 10 signals or when the refrigerator door is opened and closed at least 2 times during a 24-hour period. The respective sensor resources are identified by [S_H]/s/m (motion sensor in the hallway), [S_L]/s/m (motion sensor in the living room) and [S_F]/s/r (the magnetic contact sensor on the fridge door). The application was built twice, once without the RESTlet concept and once with the RESTlet concept.

For the non-RESTlet application development, we employed a RESTful approach using CoAP with the observe option where the gateway establishes an observation relationship with each sensor by sending a CoAP GET request with (conditional) observe option. Whenever the gateway receives notifications from the sensors, it executes a sequence of code to realize the desired result. Fig. 8 shows high-level code that should be executed to realize the application under consideration.

The RESTlet approach makes use of 5 RESTlets to achieve the same result, as indicated in the block diagram shown in Fig. 9. The output of the two motion sensors is used as input to the COUNTER RESTlet which increment its output value whenever it receives a new input. The ISLARGER RESTlet takes the output of the first COUNTER RESTlet and produces 1 if the input is larger than 9 and 0 otherwise. Similarly, the output of the reed sensor on the fridge is fed into the counter, which, in turn is connected, to another ISLARGER RESTlet. The OR RESTlet accepts two inputs and performs a logical OR. The output of the OR

Step 1: Initialization

```
numberOfTimesMovementDetected = 0
numberOfTimesFridgeOpened  = 0
StartTimer(86400s)
```

Step 2: Create Observe relationship with sensors

```
GET [S_H]/s/m, obs
GET [S_L]/s/m, obs
GET [S_F]/s/r, obs
```

Step 3: Create the Programming Logic

```
if ( EVENT == "MOTION_HALL_OBSERVE“ ||
     EVENT == "MOTION_LIVING_OBSERVE")
{
        numberOfTimesMovementDetected++;
         if (numberOfTimesMovementDetected >= 10)
         {
            // start again for new interval
            numberOfTimesMovementDetected = 0;
            numberOfTimesFridgeOpened = 0;
         }
         restartTimer(86400s);
    }
    else if (Event == "REED_FRIDGE_OBSERVE")
    {
         if (lastStatus == “CLOSED")
         {
            numberOfTimesFridgeOpened++;
            if (numberOfTimesFridgeOpened >= 2)
            {
               // start again for new interval
               numberOfTimesMovementDetected = 0;
               numberOfTimesFridgeOpened = 0;
            }
           restartTimer(86400s);
         }
    }
}
```

Figure 8: non-RESTlet RESTful Application Code

RESTlet is used to trigger the Actuator A. In the whole system, there are 2 different types of control parameters, time based and value based (in the figure, TT and VT respectively). It is interesting to note that the output of the counters is a non-negative integer while the output of the remaining RESTlets is Boolean.

To create the desired application, we send 5 POST request to the /restlet resource of the HOST chosen to store the RESTlets. This will dynamically create the RESTlets and their input, output and control resources. In this example, we used the RESTlet type followed by a number to be used as unique id for the RESTlet resources. For instance, the first counter is COUNTER_1 and the second is COUNTER_2. If the resources are successfully created, the Location-Path option of the response contains the base location of the created input and output resources. The resources are represented as restlet-id/resource-type/resource-number. For example, COUNTER_1/input/0 refers to the first input of the COUNTER_1 RESTlet.

The next step of the programming is binding the output of one component to the input of another component by sending GET requests to the different hosts. It is interesting to note that the outputs of the two sensors are all bound to the single input of the counter because of the requirement of the application. If the outputs of each sensor had to be treated separately, each sensor output would have been bound to different counters. In this case, we would have more instantiations of COUNTER RESTlets in our application without requiring additional programming.

This approach has several advantages. One of the advantages is the simplification of application development. Most of the application logic is already implemented in the RESTlets. As the processing logic of the RESTlets can be very basic logical (AND, OR, NOT, XOR …) or arithmetic operations (COUNTER, ADD, SUBTRACT, MULTIPLY, DIVIDE …), building applications will be as simple as sending RESTful messages to create bindings between the different components (Sensors, RESTlets, and Actuators). In addition, some general purpose complex modules can also be modeled to be used by most applications. Examples of such modules include an SMS module that sends text to a preconfigured number and a WriteToDatabase Module which sends outputs to a database on a specific host.

The other advantage of the RESTlets approach is the flexibility of placement. Based on the application requirements and the complexity (or simplicity) of the RESTlets, they may be placed in the cloud, the sensor network gateway or in the LLN. It is also possible to place different RESTlets of the same application at different places or on different devices. This flexibility in placement of the RESTlets is important to optimize different aspects of the resources in the sensor network. Putting all RESTlets in the sensor network reduces the traffic flow to the gateway, and hence, reduces traffic congestion at the uplink nodes and improves delay. However, this might introduce additional processing and memory overhead on constrained devices. A better alternative may be putting simple RESTlets in the constrained network and complex RESTlets at the gateway or in the cloud. This way we may balance traffic congestion and resource utilization of the resources. Alternatively, we may also use more capable nodes (with more memory and processing speed) in the sensor network to host the RESTlets. However, optimal RESTlet placement is outside the scope of this paper.

This solution uses a CoAP based RESTful application development model by breaking down applications into small and manageable units and interconnecting those units

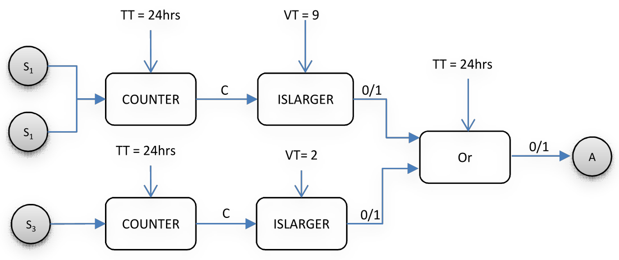


Figure 9: Block Diagram Showing Binding of RESTlets

Step 1: Create the necessary RESTlets.

```
POST [Host]/restlet       Payload: “RN=COUNTERS;TT=86400;”
            [Response: Location-path = /restlet/COUNTERS_1]

POST [Host]/restlet       Payload: “RN=COUNTERS;TT=86400;”
            [Response: Location-Path = /restlet/COUNTERS_2]

POST [Host]/restlet       Payload: “RN=ISLARGER;VT=9;”
            [Response: Location-Path = /restlet/ISLARGER_1]

POST [Host]/restlet       Payload = “RN=ISLARGER;VT=2;”
            [Response: Location-Path = /restlet/ISLARGER_2]

POST [Host]/restlet       Payload = “RN=OR;”
            [Response:  Location-path = /restlet/OR_1]
```

Step 2: Create the bindings to interconnect components.

```
GET [S1]/s/m              BIND_URI = [HOST]//restlet/COUNTERS_1/input/0
GET [S2]/s/m              BIND_URI = [HOST] /restlet/COUNTERS_1/input/0
GET [S3]/s/d              BIND_URI = [HOST] /restlet/COUNTERS_2/input/0
GET [HOST]/restlet/COUNTERS_1/output BIND_URI = [Host]/restlet/ISLARGER_1/input/0
GET [HOST]/restlet/COUNTERS_2/output BIND_URI = [Host] /restlet/ISLARGER_2/input/0
GET [HOST]/restlet/ISLARGER_1/output BIND_URI = [Host] /restlet/OR_1/input/0
GET [HOST]/restlet/ISLARGER_2/output BIND_URI = [Host] /restlet/OR_1/input/1
GET [HOST]/restlet/OR_1/output                  BIND_URI = [A] /a/toggle
```

Figure 10: Application Code for RESTlets

using REST mechanisms. Using the same protocol, CoAP and the same mechanisms (GET, PUT and POST) to realize simple sensor/actuator bindings and IoT application development is also an added advantage.

2) *Memory Requirement*

In our RESTlet model, the actual memory requirement of an application depends on the number and type of RESTlets used to realize the application. For instance, the RESTlets we created for experimentation are the basic application building blocks such as logical AND, logical OR, and counters, which have a minimum of 340 bytes and a maximum of 384 bytes. The memory requirement of an application increases as more RESTlets are being used. For example, the lifestyle monitoring application discussed above used 1724 bytes. The whole amount of memory might be taken from one device or it might be distributed among different devices in the network.

There is a trade-off between putting all RESTlets on the same machine and distributing them among multiple nodes. If all RESTlets are defined on one device, the memory requirement will be higher, particularly for devices hosting multiple applications that involve several RESTlets. On the other hand, the traffic flow will be almost non-existent as most of the binding execution stays within the same device. Distributing the RESTlets among multiple devices reduces the per-device memory requirement but increases the traffic in the network. One of our future works is experimenting with different applications to suggest optimal placements of the RESTlets.

3) *Processing Time*

In the RESTlet approach, the total processing time of an application to perform a given task is the sum of the processing time of every RESTlet code and the transmission of CoAP packets between RESTlets. The transmission time is also computed as the sum of the packet processing time and the radio communication time. If all RESTlets are on the same device the radio communication time is non-existent. Therefore, the total processing time is the sum of processing time of the RESTlets code and the packet processing time. With the appropriate cross-layer optimization, the packet processing time could be reduced to 0. This could make the processing overhead of the RESTlet approach smaller. Cross-layer optimization is also part of our future work.

IX. RELATED WORK

There are different works that address the association of sensor and actuator nodes. Zigbee End Device binding [7] is one of the notable works that addresses device bindings. [7] states that devices with a similar profile can be dynamically bound by the ZigBee coordinator if they meet specific requirements such as matching cluster IDs. This solution puts a rather stringent requirement on the nodes making its flexibility quite limited. The CoRE Interfaces draft [14], also mentions the concept of bindings in the context of CoAP. In this context, a binding is called the abstract relationship between two resources. The mechanism proposed in the draft allows end devices to establish a binding relationship through discovery mechanisms or through human intervention and then synchronize the content of their resources. Three binding methods, namely polling, observe and push, are defined to achieve this synchronization. The observe method creates an observation relationship between the end points and every notification copies the content of the resource to the observer. This solution has its advantages as it provides a generic solution that can be used in interface descriptions. However, the solution focuses on synchronizing the contents of two resources on different end devices. It is not possible to execute a specific action on the other device. Additional programming logic is still required to send the appropriate trigger to the same or different actuator.

There are also a number of works on IoT application development models. Some developers prefer WS-* such as SOAP requests and responses transmitted over the network using HTTP for IoT applications while others suggest RESTful approaches [15]. Based on a research conducted on developers, [16] concludes REST to be easier to program smart objects. One of the RESTful approaches is the Actinium runtime container which exposes Java Scripts, configurations and their management through a RESTful programming interface using CoAP [21]. The proposed architecture breaks large programs into smaller apps for reusability as our system does. However, there are a number of differences from our work. First, the apps (scripts) contain the CoAP requests, whereas RESTlets are just processing units and the link between RESTlets, sensors and actuators is made via the binding process. Second, this approach requires the devices to understand and execute the scripting language which is hard to apply in constrained devices. Finally, the core of the architecture, the runtime container, must be run in a non-constrained environment while our solution can fully be decentralized. The other

RESTful approach for IoT application development is Thing Broker [17], a platform that provides a Twitter-based RESTful interface for IoT application development. This approach uses "things" (e.g. sensors, data, computers, etc.) and "events" for application development. The whole world is considered to be composed of *things,* and *events* are associated with things. When a new event is generated by a *thing*, its data will be available to its followers. This approach is considerably different from our approach as it uses a high level abstraction of devices, data and events while we focus on loose coupling of processing of data and devices. LooCi [18] is another component and binding model for IoT applications. It uses an event-based binding model and standardized event types that allow easy component interactions and re-use of components. This approach uses RPC for communication.

X. Conclusion and Future Work

In this paper we presented how the CoAP protocol is extended to implement direct bindings of any two CoAP-enabled devices using a third party device. The two devices continue communicating with each other without involvement of the third party. As binding creation is entirely based on CoAP, it creates flexible communication between any two communicating devices contributing to the vision of a network of everything. We further extended the binding concept to add intelligence to the interaction of nodes by augmenting processing logic to the interactions. These entities, called RESTlets, can be used as building blocks for simple IoT applications. RESTlets take input from sensors or other RESTlets, process them and generate output which, in turn, will serve as input for other RESTlets or as a trigger to actuators. By dynamically creating RESTlets and binding inputs with outputs, the desired IoT application can be created without explicit coding for each application. The simplicity of the RESTlets allow them to be distributed throughout the network to improve efficiency.

There are a number of optimizations that are planned for the binding concept as well as the RESTlets. We plan to work on cross-layer optimization solutions such as modifying the routing protocol, MAC protocol or the RDC protocol to be aware of active bindings in order to further improve the performance of bindings. These improvements of bindings also improve RESTlet interactions. Suggesting optimal distribution of RESTlets in the network and implementing RESTlets on constrained devices will also be a topic of our future work. In addition, identifying suitable RESTlet content formats will also be part of our future work. A Binding Directory, a resource-directory like entity [20], which stores all active bindings, will also be developed to enable easy management of bindings and debugging of RESTlet based applications.

Acknowledgment

This work has been supported by the VLIR-UOS, through the Inter University Cooperation program at Jimma University (IUCJU), Ethiopia.

References

[1] A. Z. Alkar, U. Buhur, "An Internet Based Wireless Home Automation System for Multifunctional Devices," Consumer Electronics, IEEE Transactions on (Volume:51 , Issue: 4), 2005

[2] Vehbi C. Gungor, Gerhard P. Hancke, "Industrial Wireless Sensor Networks: Challenges, Design Principles, and Technical Approaches," IEEE Trans. on Ind. Elect., VOL. 56, NO. 10

[3] V. C. Gungor, B. Lu, G. P. Hancke, "Opportunities and Challenges of Wireless Sensor Networks in Smart Grid," IEEE Trans. on Ind. Elect., VOL. 57, NO. 10, p. 3557, 2010

[4] N. Kushalnagar, G. Montenegro, C. Schumacher, "RFC4919: IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs):Overview, Assumptions, Problem Statement, and Goals," IETF , August 2007.

[5] Z. Shelby, K. Hartke, and C. Bormann, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-18 (work in progress), IETF, June 2013.

[6] I. Ishaq, et al., "IETF Standardization in the Field of the Internet of Things (IoT): A Survey," Journal of Senor Actuator Networks, 2013.

[7] ZigBee Alliance, "ZigBee Specifications r13," 2006.

[8] K. Hartke, "Observing Resources in CoAP (draft-ietf-core-observe-18)," work in progress, IETF, 2014.

[9] L. Shi, J. Hoebeke, F. Van den Abeele, and A. Jara, "Conditional observe in CoAP (draft-li-core-conditional-observe-04)," June 2013.

[10] G. K. Teklemariam, J. Hoebeke, I. Moerman, P. Demeester, "Facilitating the creation of IoT applications through conditional observations in CoAP," EURASIP Journal on Wireless Communications and Networking, 2013:177

[11] M. Kovatsch, S. D, "A Low-Power CoAP for Contiki," Proceedings of the 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2011), Valencia

[12] T. Winter, P. Thubert, and et al., "RPL: Routing Protocol for Low Power and Lossy Networks, RFC6550," March 2012.

[13] Z. Shelby, C. Chauvenet, "The IPSO Application Framework (draft-ipso-app-framework-04)," IPSO Alliance , August 2012.

[14] Z. Shelby, "CoRE Interfaces (draft-shelby-core-interfaces-05), (work in progress)" March 2013

[15] C. Pautasso, O. Zimmermann, F. Leymann, "RESTful Web Services vs. 'Big' Web Services: Making the Right Architectural Decision," WWW 2008, April 2008, Beijing.

[16] D. Guinard, I. Ion, and S. Mayer, "In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers' Perspective," Mob. & Ubiq. Sys: Computing, Networking, and Services Lecture Notes, Volume 104, 2012, pp 326-337

[17] R. A. P. Almeida, R. Calderon, et. al, "Thing Broker: A Twitter for Things," UbiComp'13, September 8–12, 2013, Zurich, Switzerland

[18] D. Hughes, K. Thoelen, et. al, "LooCI: a Loosely-coupled Component Infrastructure for Networked Embedded Systems," MoMM2009, December 14–16, 2009, Kuala Lumpur, Malaysia

[19] IEEE Computer Society, "IEEE Std. 802.15.4-2006", October 2006

[20] Z. Shelb, C. Bormann, S. Krco, "CoRE Resource Directory," draft-ietf-core-resource-directory-01, December 2013

[21] M. Kovatsch, M. Lanter, S. Duquennoy, "Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications," Internet of Things (IoT) 3rd International Conference, 2012