A Holistic Architecture for the Internet of Things, Sensing Services and Big Data

David Tracey, Cormac Sreenan
Dept. Of Computer Science,
University College Cork,
Cork, Ireland

***Abstract*—**Wireless Sensor Networks (WSNs) increasingly enable applications and services to interact with the physical world. Such services may be located across the Internet from the sensing network. Cloud services and big data approaches may be used to store and analyse this data to improve scalability and availability, which will be required for the billions of devices envisaged in the Internet of Things (IoT). The potential of WSNs is limited by the relatively low number deployed and the difficulties imposed by their heterogeneous nature and limited (or proprietary) development environments and interfaces. This paper proposes a set of requirements for achieving a pervasive, integrated information system of WSNs and associated services. It also presents an architecture which is termed holistic as it considers the flow of the data from sensors through to services. The architecture provides a set of abstractions for the different types of sensors and services. It has been designed for implementation on a resource constrained node and to be extensible to server environments. This paper presents a 'C' implementation of the core architecture, including services on Linux and Contiki (using the Constrained Application Protocol (CoAP)) and a Linux service to integrate with the Hadoop HBase datastore.

***Index Terms*—Wireless Sensor Networks, Tuple Space, Information Model, Protocols, Cloud Computing, Big Data**

I. Introduction

Wireless Sensor Networks (WSNs) are being enabled by the increasing availability of sensors and advances in wireless technologies, hardware and the use of IP for connecting resource constrained devices. The use of micro IP stacks (and IPv6 over Low power Wireless Personal Access Networks (6LowPAN) [1] has enabled constrained devices to connect to the Internet in a so called "Internet of Things" (IoT). Definitions of IoT generally share the idea that it relates to the integration of the physical world with the virtual world of the Internet [2]. IoT is characterised by an interconnected set of individually addressed and constrained (possibly autonomous) devices in a distributed system, with sensing/active devices for physical phenomena, data collection, and applications using sensing, computation and actuation. There could potentially be billions of such devices connected across the Internet with predictions of 50 to 100 billion devices being connected to the Internet by 2020 [3].

WSNs have a (possibly large) number of devices with sensing capabilities, limited processing capability and wireless connectivity (allowing nodes to be deployed close to the phenomenon being observed) to other sensor or gateway nodes. WSN nodes exist to sense a particular entity, collect (and possibly parse or aggregate) the data and send the data to one or more destinations and ultimately to an application across a range of areas, e.g. environmental monitoring, surveillance and healthcare. Such deployments are usually dedicated and proprietary or specialized to optimise one particular aspect such as lifetime.

The availability of increased storage and processing power at a lower cost with greater bandwidth has enabled a range of Cloud Computing services. In terms of IoT, this allows more sources of data to be collected and for the data to be held for a longer time and to be processed by powerful cloud based applications and Big Data techniques, e.g. HBase and MapReduce. Big Data can be characterised by the 3 'Vs of Volume (size of the data), Variety (range in type and source of data) and Velocity (frequency of data generation) [4].

The constrained nature of WSN nodes in terms of processing power, memory and energy consumption makes it difficult to enable WSNs to be more easily deployed, developed and integrated with new Internet based services. A key challenge is to enable WSNs to become extensions of the Internet infrastructure, to take full advantage of Cloud and Big Data services [5] and be universally available, rather than isolated and relatively small islands of sensor networks. To address this challenge, this paper presents a set of architectural requirements, a resulting layered architecture and abstractions for the data exchange roles taken by services on WSN nodes and in the Cloud, supported by a novel protocol. It also evaluates an initial implementation of the architecture.

The remainder of this paper is organised as follows. We discuss prior work in section II and present a set of architectural requirements to meet the challenge above in Section III. Section IV presents the architecture, including its service abstractions, object library and introduces the message protocol. Sections V and VI present an initial implementation and evaluation of the architecture and its HBase integration. The paper concludes in Section VI.

II. Existing And Emerging Frameworks

This section outlines the current frameworks and approaches used in the Internet of Things, WSN software, Cloud Integration and Big Data. A recent survey shows that only 13 of 28 WSN systems surveyed have actually been implemented on hardware rather than run in simulators [6] and that there is still an absence of broad abstractions, which we propose later. Hence applications are often bound to a particular WSN technology and not easily portable as the

application developer must have detailed knowledge of each underlying technology.

A. Constrained Application Protocol and IoT

The Constrained Application Protocol (CoAP) has been developed by the Internet Engineering Task Force (IETF) and is targeted at the IoT area [7]. It is a standard for a specialized web transfer protocol for constrained nodes and constrained (e.g. low-power, lossy) networks. It is built on top of UDP and uses web concepts such as URIs and media formats for easy integration of such constrained environments into HTTP and it addresses issues such as the overhead of HTTP headers, XML parsing, TCP over lossy links and the handling of node duty cycles. It uses the REST architectural style [8], where resources (such as sensors) are represented in a number of formats and accessed by their Universal Resource Identifier (URI) using a limited set of verbs, such as GET, POST, PUT, DELETE in HTTP. The decoupled nature of this style facilitates application development and scalability.

B. Cloud Integration Approaches

The NIST has proposed three main Cloud service types/models of Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [10]. Sensing as a Service has been proposed, with elements of an IAAS solution [5], but more often as a PAAS. Commercial offerings such as cosm.com allow users to upload their sensor data using a defined set of attributes.

Sensor-Cloud [11] uses SensorML to describe sensor metadata and manages sensors via the cloud, rather than providing their data as a service. The OpenIoT [12] middleware platform comprises an IoTCloudController (provides SOAP Web services for sensor registration, discovery, subscription and control), a JMS style Message Broker, Sensors (with a module to publish to OpenIoT) and Clients (which subscribe to or consume sensor data). Another approach uses a data channel based on Java FileInputStream(), FileOutputStream() to hide the underlying network protocols and a Sensor Server on the wireless network's master node to filter sensor data and to deliver it to cloud services [13]. This approach is simple, but limited in its flexibility. Another integration approach uses a content-based pub-sub model for event publications and subscriptions for asynchronous data exchange, requiring a gateway at the edge of the cloud to receive sensor data, a Pub/Sub Broker to process and deliver events to registered users and a range of components to support SaaS applications [14].

These middleware approaches to cloud integration require specific application gateways/proxies at the edge of each wireless network and their own sensor data definition.

C. Big Data

The use of Big Data is well established commercially to analyse large amounts of data in order to make timely decisions, e.g. in retail for analysing consumer behaviour and preferences. This paper illustrates how seamlessly our holistic architecture can accommodate the use of Apache HBase to store sensor data. HBase uses the Hadoop Distributed File System (HDFS) and is a distributed, versioned, column-oriented, store, derived from Google BigTable. HBase stores data into tables, rows and cells. Rows are sorted by row key and each cell in a table is specified by a row key, column key and a version, with the content held as an un-interpreted array of bytes. We consider HBase suitable for WSN data not just because it is scalable and can store large amounts of replicated data, but because of its key value nature and flexible data access. The data access is provided by a rapid query using a get with a row key and a scan using an arbitrary combination of selected column family names, qualifier names, timestamp, and cell values. It also provides sparse tables, which is appropriate for cases where not all WSN nodes can provide all the columns defined. Columns belong to a particular column family and are identified by a qualifier. Column families must be declared at schema definition time, but individual columns can be added to a family at run time. The associated MapReduce model has been shown to be appropriate for processing sensor data [15].

D. WSN Software Frameworks

Programming WSN applications and nodes is time-consuming, error-prone and difficult requiring low level hardware and network knowledge, often using a vendor specific environment for particular hardware. Software Engineering concepts and higher level abstractions are required to improve the development process and ease the integration with other systems in order for wider deployment of WSNs [16] as part of the seamless, context aware environments envisaged in pervasive computing [17], where applications/services are interested in the sensed information, not the underlying hardware or wireless network. Special purpose operating systems like Contiki are used on more constrained nodes, while more powerful hardware platforms such as SUNSPOT have high level language support such as Java, but at the cost of more expensive hardware and higher power consumption. TinyDB [18] essentially considers the WSN as a distributed database and can be considered limited by its table based approach and relational queries, especially in terms of handling events. Middleware approaches such as Sensation[19] treat the sensor network as a whole as an information source similar to a database, with its middleware acting as an integration layer between applications and networks and a proxy with a prioi configuration for particular WSNs to hide device and network specifics. Agent based middleware requires particular node computational capability and the energy used by traffic for code mobility reduces node lifetime [20]. A data-centric approach such as directed diffusion has the potential of significant energy savings and relatively high performance, but it is tightly coupled to a query on demand data model where applications can accept aggregated data [21]. TeenyLIME [22] is another higher level approach, which is based on a shared memory space (tuple space), derived from Linda's [23] limited number of simple operations to insert, read, and withdraw tuples from a tuple space. TeenyLIME has been deployed in a real-world application and shown the usefulness of a tuple space approach in WSNs [24], but a node's local tuple space is only shared with the nodes within communication range.

III. Architecture Requirements

The objective of our architecture is to simplify the development, configuration and deployment issues to enable ubiquity of WSNs, easier interfacing to other networks and the easier development of generic and more powerful applications using sensor data. To meet this objective, we define the following architecture requirements:

1. It must be independent of particular node hardware, must handle a range of node functional capabilities and provide an extensible layered system able to handle the radio channel and environmental factors, within the required limits of power consumption.
2. It must provide abstractions for the basic operations required of a sensor node and the services using it, which map easily to a range of heterogeneous devices and higher level services.
3. It must clearly define the possible roles of nodes and any protocols must be sufficiently simple for low capability devices to participate. It is unreasonable to demand that all nodes have equal functionality, as this limits the ability to handle more powerful nodes. Nodes will, however, require a minimum level of functionality, e.g. forwarding data to a neighbour.
4. It must provide a consistent means to exchange sensor information independent of the underlying technology and provide specific support for the modelling of sensor data to allow integration into higher level systems. A sensor node should be able to advise other nodes and services of its sensing and platform capabilities.
5. It must be able to handle small, static networks and allow the system to adapt as the network grows/changes or encounters other networks and support applications discovering and collaborating without a centralized coordination facility.

The need for a more holistic approach can be seen in a remote healthcare monitoring scenario, where sensors connect to a central gateway in a house over a wireless network. The gateway is responsible for storing the data locally and uploading data to a central health monitoring site, possibly via a central gateway/proxy and cloud based services to analyse the data [25]. Such solutions often require sensor application and proxy design to handle data integration, network integration and security concerns. This lack of unified abstractions will become more problematic in this scenario as Wireless Body Area Networks are deployed, e.g. IEEE802.15.6 which allows up to 64 nodes on a body to connect via a central co-ordinator node. When large numbers of WSNs/BANs are deployed, treating these networks of nodes as peripheral devices and connecting them to the Internet via proxies or sinks will limit performance and scalability [26].

IV. The Holistic Architecture

This section proposes an architecture to meet the requirements from section II. The key principle underlying it is that all WSNs are primarily about delivering sensed data/events to one or more applications (periodically, on-demand or asynchronously) or commands to actuators from applications. The architecture meets the requirements in section II by using a number of service abstractions to model the different roles a service can perform, defined software layers and an object infrastructure to support information models. It uses a simple protocol based on Peer to Peer (P2P) concepts able to run on constrained nodes. The approach is termed as holistic because it considers the entirety of the data flow between sensor and service(s), supported by lower layers, rather than each layer specifying its own behaviour in isolation.

Figure 1 shows the layers in the architecture for nodes of different capability with their different roles, e.g. a node that only fulfills the forwarder role does not have a local instrumentation layer, but has an object space to store data from remote peers. It also shows how a HBase store is modeled as a sink service and how it would be exposed to constrained nodes using a hpp_endpoint. The Data Model Service Layer provides a high level abstraction for node data and it uses the object space to hold remote peer data and local data (if supported by the role), so simplifying the communication of data between sensor nodes and higher level applications. The local instrumentation (li) layer supports local data and provides an abstraction above device specific layers to map to the underlying node functions or data.

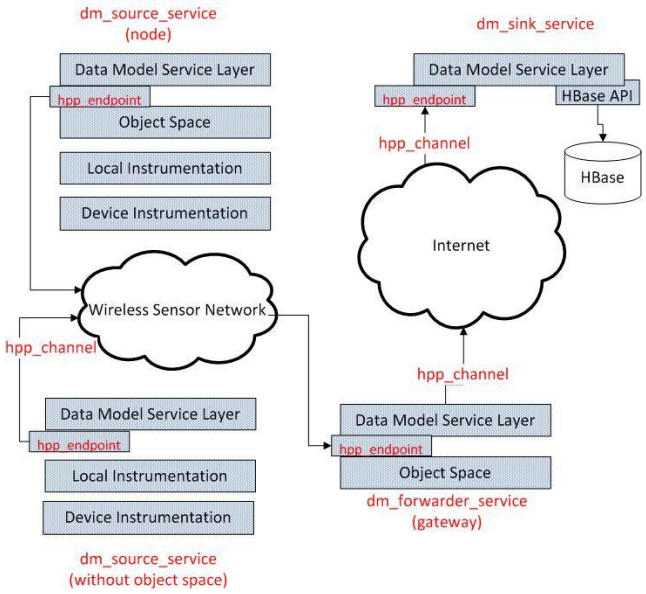

Fig. 1. Holistic Architecture

A. Service Abstractions and Data Model Service Layer

The architecture's Data Model layer uses a set of service roles to model the data flow and to abstract the lower layer interfaces for nodes and hide the underlying network and node specifics from the application developer. The Data Model (DM) Service layer abstracts the service capabilities using roles reflecting the nature of the data exchange. The defined roles support a range of capability with the following roles:

- DM_SINK_SRV (adds interest objects to its peers for data it wants)

- DM_SOURCE_SRV (sends its sensor data)
- DM_FORWARDER_SRV (forwards to peer services)
- DM_STORE_SRV (stores data from peer services)
- DM_MATCHER_SRV (provides results of advanced matching queries)
- DM_AGGREGATOR_SRV (aggregates data from peer services)

A node can have several roles according to its resources, e.g. a constrained node may only act as a DM_SOURCE_SRV, not storing its own data or a node may remove its capability as a DM_FORWARDER_SRV if low on remaining power. Source and sink roles can be seen in other flow based approaches such as Flume, used to deliver large amounts of log data in Web and Cloud Computing services. We have added the forwarder, aggregator and store roles for the capabilities of WSN nodes.

Services use the holistic peer-to-peer (hpp) protocol to exchange hpp messages using the hpp_endpoint and hpp_channel. A hpp service registers/deregisters instances of its objects (and their specific methods), its capabilities (in a template object) and its interests in other objects with the object space layer. These objects may be forwarded to remote peers and services must renew their object leases with their peers. A service's capabilities are thus advertised to other services, allowing a node to set its sensing and response timing based on the received interests, e.g. a sensor may be able to report every 15 minutes, but only sends a reading every hour based on what interests were provided by applications.

B. The Object Space Layer

The object library is a simple object-like infrastructure suitable for resource constrained devices with object functions to support a simple shared object store and associated API. It is used to store locally instrumented data and data received from other nodes for aggregation or other purposes. It is based on Linda's tuple space concepts. The decoupling in time and space of tuple space communication enables interactions where applications can be added or removed independently and do not have to be available simultaneously to transfer data between themselves. Our object library has been implemented in C and its main methods are objectAdd(), objectRemove(), objectGetByHandle(),objectGetByName(), objectLeaseRenew() and objectGetInstance().

The object space is non-prescriptive about the classes and instances it holds, except that it requires the use of a template to hold the type of each attribute of the object and its methods. An object structure represents an object held in the object store, with its template and each object has a lease, allowing for the space to remove objects if leases are not renewed. The template and instance are kept separately to allow for objects that represent a class (i.e. do not have instances) and to allow a range of object encodings. For resource constrained devices it also offers an efficient way of transferring them to other nodes, where the template (or a reference) can be sent once to another node prior to the encoded object. Templates are also used to define node capabilities on a model/object basis (i.e. to specify which properties of a standard object are instrumented). The definition of a template is transparent to the object store.

C. Local Instrumentation Layer

This layer hides the platform specific sensor implementations and provides get()/set() functions and method prototypes for node functionality such as power off. It also allows the use of C language features such as pointers to reduce memory usage. It also provides per attribute structures to allow only those object/sensor attributes supported by the node to be implemented and these can be built into higher level information models, e.g. an SNMP MIB table or CIM object.

D. The Holistic P2P Protocol (HPP) and Hpp Channel

A simple message protocol suitable for resource limited nodes has been developed to support interaction between the different service roles we have defined. It uses a hpp_channel between hpp_endpoints to provide a single API to run on top of various network and data link layers, so that applications do not require knowledge of the underlying network. It uses a limited set of message types in line with the operations of the object space. HPP has the characteristics of a P2P system at the application level as its hpp_channel and defined roles allow nodes to act in an autonomic and dynamic manner where nodes enter or leave the network and any node may initiate, manage or terminate a session with other nodes. It does not at present support node discovery (but can discover node capabilities) or overlay networks.

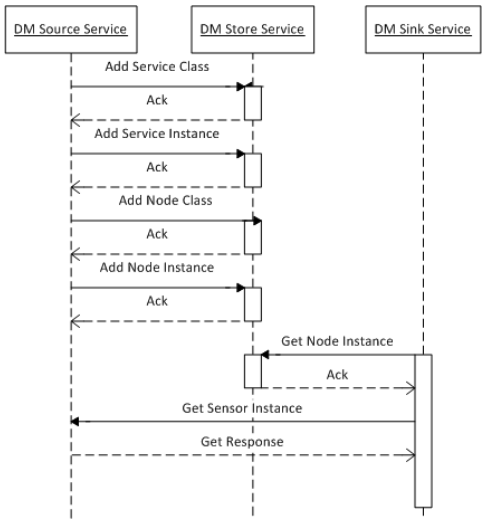

Fig. 2. Sample Service Interaction

HPP messages consist of blocks, always started by at least a Header block followed by other blocks for Address, Data and Credentials. Some messages may only hold a header block and every block has the same preamble of a Command, a block length and a block id, so a WSN node only has to receive the header block and parse the command to determine if it should process this message. During the Connection Phase, the messages are Hello, Attach and Detach and during the Data Phase, the messages are Get, Add, Remove, Get Response, Action, Notify and Acknowledgement. All nodes must support Hello, Attach, Detach, but nodes may support only Get/Get Response in the Data phase (shown in its capabilities). The command types map well to the REST approach, although

Action, Notify primitives have been added for the actuator and alert functionality of sensor devices.

The sequence diagram in Figure 2 shows an example message interaction (after Hello and not showing object lease renewal), where a source service (on a node) adds both its service and node class templates and instances to a store service, e.g. on a higher powered node. This store service is queried by a sink service for the node's capabilities and determines that there is a sensor on the node, which it then retrieves. Other interactions are possible, e.g. the source service adds its sensor class and instance to a store service (at a period matching the sensor reading update) so the retrieval by the sink service can use the store service's data for that node and not require additional transmission to the original source node.

V. IMPLEMENTATION

A. HPP Implementation

This section discusses the design and implementation issues encountered in an initial implementation using the CIM information model for sensor objects and storing this data in HBase. The implementation in 'C' includes the Data Model Service, Object Space and Local Instrumentation Layers shown in Figure 1 and a DM_SINK_SRV service written in Java to integrate with HBase. The 'C' code was implemented initially on Linux, using the hpp_service abstraction on top of the hpp channel abstraction to hide the specific network layer details. Testing was done using Linux based source nodes sending hpp messages to transfer their classes and instances to a specified number of remote nodes using a small number of functions, as the following code is all that is required for a service to start receiving messages from other services:

```
rv = hpp_endpoint_check(endpoint_ptr);
if (rv == 0) {
  channel_ptr = hpp_endpoint_accept(endpoint_ptr);
} else if (rv > 0 ) {
  hpp_endpoint_get_messages(endpoint_ptr);
} // timed out with no data, so loop again
```

The Linux code was then ported to Contiki running on a Sky WSN mote (emulated in Cooja), using the CoAP implementation. This implementation created objects and added them to the object space at different times as the node started up (and added dynamically later), e.g. the DM service class and instance objects were created at the start of the process, followed by the node class and instance and the local instrumented objects for led and temperature sensor. This showed the architecture and its abstractions worked across Linux and constrained nodes.

B. Data Model Service Layer

The initial Contiki implementation includes a number of custom CoAP "resources" on top of the data model layer, using the object space. For example, a DM_SOURCE_SRV service and node objects were implemented as key value pair objects be sent to another node such as a DM_STORE_SRV. Also, a CoAP resource was implemented for the creation of HPP objects dynamically. Classes and instances for red/blue/green leds, temperature sensor and node, using a subset of attributes from the CIM object, were also implemented. The following pseudo-code (not including error code) shows the service adding its own service class template and initialising its role(s):

```
uchar dm_register_dm_service(objectAttr_t *template_ptr,
    objectAttr_t *inst_ptr, objectAttr_t *inst_key_ptr) {

if (dm_srv_class_hdl == 0)
  dm_srv_class_hdl = dm_add_service_class(
                       &DMServiceTemplate,
                       DM_SERVICE_CLASSNAME);

hdl = dm_add_instance(.....);

if (service_role || DM_SOURCE_SRV) {
   dm_source_init(); // initialise my local instrumentation (li)
}                    // objectswith object store
if (service_role || DM_SINK_SRV) {
      dm_sink_init(); // add objects we are interested in to
}                     // object space on remote peers
if (service_role || DM_STORE_SRV) {
   dm_store_init(); // set up support for holding
}                   //instrumentation objects from peers
return (0);
}
```

The data model layer provides support functions on top of the object library; dm_initialise() and dm_add_class(), dm_add_instance(), dm_remove_instance() for local or remote sensor classes/instances. Retrieving object instances is done by dm_get_instance(inst_handle) or dm_find_instance(), which uses key values or particular attribute values according to the matching specified. Matching is implemented in the data model layer and not the object library (the contents of objects are transparent to it). A hpp Add message is sent to a remote node to add a class or instance, with the remote node calling setupTemplate() to process the class attributes received and then dm_add_class() or calling dm_add_instance() with the received instance attributes.

C. Local Instrumentation Layer

Locally instrumented data is implemented using an li_class_property for each property and an li_inst_property with the value. This per property approach aligns with the hardware/vendor specific implementations to access particular readings or data, e.g. to access sensor data by reading a value from a register or an API call like get_sensor_reading(). The li_class_property structure does not make any assumption about the object it is to be put in (it could appear in more than one) and can be combined into different classes for particular information models or be added into tables or key value stores such as HBase. A node's local instrumentation (li) classes and instances are added to its local object store and optionally converted into key value pairs for adding to other nodes.

Key and non-key properties are treated separately as many information models use keys to identify groups of data (rows in SNMP or HBase or object instances in CIM), but also because

resource constrained devices often set keys when the class is created and can be allocated then, whereas non-key data in an instance changes and may be read by a dynamic getter function.

D. HPP Integrated Erbium-CoAP Implementation on Contiki

The Linux implementations of the local instrumentation (li) layer, data model and object space, supporting libraries (memory utilities, doubly linked list, hash, lease) and the message building parts of the hpp protocol have been ported to Contiki as part of the pre-existing erbium-REST implementation example [9]. This approach allowed these items to be tested on hardware with a supporting REST infrastructure and for the port to use existing Contiki libraries. The code samples below show the integration itself was straightforward. The hpp message payload was simply added as CoAP payload using the call REST.set_response_payload(). It is expected that adding the hpp channel abstraction on top of the existing Contiki networking stack will not be difficult. The additional code required in Contiki compared to Linux consisted of:

- A Contiki call to initialize hpp_element. The simple call service_hdl = service_initialise(); was added to the Contiki main PROCESS to call the initialize code in the Linux hpp_service daemon to set up the service and node objects.
- Integrating with the REST code. This consisted of code to add the resource into the erbium resource handling list rest_activate_resource(&resource_hppnode) and the code to implement that resource. The CoAP resources were accessed via URLs using a suffix of hpp/[classname] and the node responded with the properties implemented in that hpp object as key value pairs in the CoAP payload, using multiple CoAP buffers. A RESOURCE macro is used to define a CoAP resource and the CoAP verbs such as get or put it handles, with a corresponding function to implement it called resource-name_handler. The handler below for the node object returns the node instance from the object space when queried over CoAP:

```
void hppnode_handler(…) {
   object_t *instObj_ptr = NULL;
   instObj_ptr = dm_find_instance(NODE_CLASS);
   hpp_send_object_resp(instObj_ptr, response, buffer);
}
```

- Adding a Resource for Hpp Objects. This allowed a URI like /nodeAddr/hpp/object?hdl=x to select an object by the handle allocated when it was created in the object space or to walk through the available objects, as shown by the following handler:

```
void hppobject_handler(…) {
   len = REST.get_query_variable(request, "hdl", &chdl);
   instObj_ptr = dm_find_object_by_handle(hdl);
   hpp_send_object_resp(instObj_ptr, response, buffer);
}
```

- Integrating with the Contiki hardware abstractions. This pseudo-code shows the li layer code wrapping the Contiki led calls and is called by a resource handler to set a led:

```
li_mote_method(int method_cap, int inst_id, int setting) {
uint8_t led = (uint8_t)inst_id;
if (method_cap == MOTE_CAP_LED_SET)
   if (setting == MOTE_LED_ON)
      leds_on(led); // Removed leds_off, leds_toggle code
}
```

E. Integration of Data From Contiki Based Node with HBase

We created a HBase table for each hpp class with a row for each instance. The tables have two column families named "key attributes" and "attributes" and a column family qualifier for each attribute. A row key consists of the hpp object's key attributes, node id and a timestamp.

A Java CoAP client (a DM_SINK_SRV) was written that connected to the desired WSN node via a socket to the CoAP Server on the Contiki rpl border router. It built a COAPPacket using COAPPacket(), called the serialize() method and sent it using the COAP libraries. It then passed the reply data and the HBaseConfiguration object it had created to writeToHBase().

The code extract below shows writeToHBase(). It assumes the table has already been created by an earlier hpp command to add the class and shows how the received hpp data as key value pairs is processed and written as a row to the HBase table for that class:

```
public static void writeToHBase(Configuration conf,
   String tableName, String hppData) {

  Map<String, String> keyKvs = getKeyMap(hppData);
  Map<String, String> attrKvs = getAttrMap(hppData);
  HBase admin = new HBaseAdmin(conf);
  HTable table = new HTable(conf, tableName);
  String rowKey = createRowKey(keyKvs);
  Put put = new Put(Bytes.toBytes(rowKey));
  // Add hpp data to column families
  addMapToHBasePut(put, keyKvs, "key attributes");
  addMapToHBasePut(put, attrKvs, "attributes");
   table.put(put);
   admin.close();
}
```

VI. Evaluation of Implementation

The initial implementation is evaluated in this section in terms of the abstractions used, the ability to map properties to objects or tables, HBase integration, the value of the initial Linux implementation and its memory use. It is planned to perform more objective tests in defined scenarios.

1) Abstractions

Evaluating abstractions can be done by ensuring that "end-user" and "WSN geek" are catered for [6]. The "end-user" is a domain expert concerned with using the WSN data and not with the network/node specifics, which the "WSN geek" is concerned with. We have shown examples where the end user is able to access the data simply with known CoAP Resources or objects or from the HBase store. The "WSN geek" has been provided with a cross-platform architecture using an object

space and data model layer with a local instrumentation layer for incorporating node specific functionality and capabilities. The code extracts show that these items made it straightforward for a node to implement objects from a rich information model on both a Linux and Contiki platform and to map to CoAP Resources. This also meets the design goal of the same abstractions giving a generic information infrastructure across heterogeneous platforms of different capability, even when used with delivery protocols other than the hpp protocol. The object space was also shown to easily map objects to specific CoAP REST resources and the hppobj resource above showed it also easily supported discovery and searches across the implemented objects.

The value of some of the service abstractions has been shown with a Java DM_SINK_SRV service that receives data as hpp key value pairs from Contiki and stores that data in HBase and also a DM_SOURCE_SRV that adds its classes and instances to specific remote nodes (via hpp add directly or in a CoAP PUT payload).

2) Object and Property Node Mapping

The sample code has shown that an attribute based implementation of the objects fits naturally with the low level specifics of the nodes and maps to CoAP REST resources, such as led and sensors and groupings of individual attributes, such as proposed in the IP for Smart Objects (IPSO) Application Framework [27]. The implementation showed that the approach of having a class object as a template with attribute descriptions and its instance object with attribute values was successful in three ways; it allowed selective use of attributes from CIM classes on constrained nodes (important for the many strings used in objects such as CIM_NumericSensor), it supported a set of abstractions in a COAP/REST environment and also allowed straightforward mapping of these attributes into a HBase store.

3) HBase Integration

In terms of data mapping, the hpp objects mapped cleanly to HBase tables and the use of a property per attribute mapped well to HBase columns. Furthermore, the approach of separate key and non-key properties could be mapped to separate HBase column families, allowing a HBase scan across all rows of key attributes as well as non-key attributes, rather than only being able to use the key attributes as instance identifiers. The hpp message primitives also mapped well to HBase functionality, e.g. the two column families defined for attributes allowed adding new objects with their attributes by creating a table (and its columns), which can be done dynamically on receiving a hpp Add message with the template class. Similarly, a hpp Add of an instance (at a given time) will result in a new row in the object's table. The architecture allowed hpp data on the node to be transported and stored in HBase, using CoAP, requiring no application level proxy and only requiring a proxy at the network level (the rpl border gateway).

4) Linux Implementation and Code Porting Issues

The approach of initially implementing on Linux allowed the design to be refined and the code to be debugged and tested more easily and rapidly, using the more advanced Linux development and debug environments. It also provided services on Linux that could integrate easily with those on constrained nodes. These benefits came at little cost in terms of the subsequent port to Contiki as most of the code did not require any changes, given the availability of standard C libraries in Contiki. The main code changes were to provide a revised Makefile, a simplified implementation of gettimeofday() used for object leases and to change the type of function parameters and structure members to reduce size (e.g. from int to char).

5) Memory Usage

It was necessary to remove parts of the erbium-CoAP code to create space for the hpp code. Retaining parts of the erbium and CoAP stack did allow using the CoAP transport and the Copper Browser plugin for testing. A more complete integration with CoAP would reduce the memory footprint and allow more hpp functionality to be included.

TABLE I. MEMORY USAGE OF REST EXAMPLE

	Original Erbium REST Code			***Erbium + HPP Code***		
	Code (%)	***Data(%)***	***Total (%)***	***Code (%)***	***Data(%)***	***Total (%)***
libc	8	0	7	9	0	8
core	9	3	8	7	2	6
Network	50	74	53	50	63	52
Platform	12	3	10	10	4	9
coap	17	17	17	11	12	11
rest	5	3	5	2	4	2
hpp	n/a	n/a	n/a	11	15	12

Table 1 shows the percentages (both applications varied by a few 100 bytes) of the available memory (10K RAM, 48K Flash) used for particular sections in the original er-rest-example application and for the modified application with hpp. The hpp application included resources for the hpp led and objects for Service, node and reduced CIM_AlarmDevice and CIM_NumericSensor. The REST engine and CoAP use a small amount of memory compared to networking, which is equivalent to that for the platform and core. It can be seen that the code and data usage of hpp is equivalent to that of CoAP, so that it is feasible for a constrained device.

VII. CONCLUSION

We have proposed a set of requirements for an architecture that reflects the characteristics of WSNs and would allow WSNs to be more widely deployed and more easily integrated with applications, including Big Data services to collect and analyse their data. We have proposed a holistic architecture with defined abstractions, software layers, a loosely coupled object space and a simple and flexible protocol. These abstractions also enabled the approach of developing the code initially on Linux and then porting to Contiki. We have also evaluated the architecture based on an initial implementation.

The first requirement has been met by showing that the architecture and abstractions can be relatively easily implemented on both constrained WSN nodes with acceptable

memory use and are also suitable for more capable devices and applications, e.g. on Linux. The second requirement has been met by providing abstractions for the basic operations of a sensor node and the services using it, e.g. the local instrumentation layer handled the underlying Contiki hardware libraries and the data model layer handled the REST resources. The third requirement has been met with the service roles, although only the source, sink and store roles have been implemented at this point. The fourth requirement has been met by showing the exchange of sensor information from the node to CoAP to HBase independent of the underlying technology.

Further work is planned to port the hpp channel abstraction to Contiki and to investigate further integration of hpp with the CoAP transport, to implement the other service roles in the architecture, as well as investigating the use of service capabilities/interests, particularly in terms of the interaction with Big Data services in the cloud to perform processing. It is also planned to investigate support for P2P overlays and the use of Distributed Hash Tables (DHT). It is also planned to perform larger scale tests with more nodes to verify the architecture meets the fifth requirement of being able to scale from small static networks to larger dynamic, heterogeneous environments and to show the benefits of the characteristics of the P2P and tuple concepts in the architecture (high scalability, redundancy, fault-tolerance and self-management).

In summary, this architecture has been shown to enable a holistic, high-level approach on constrained and powerful platforms and enable a straightforward integration with Contiki and HBase to store sensor data, requiring only simple message reformats without requiring semantic changes or application proxies in an infrastructure of nodes and services.

REFERENCES

[1] N. Kushalnagar, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals", RFC 4919

[2] S. Haller, "The Things in the Internet of Things", Internet of Things Conference 2010, http://www.iot2010.org/

[3] D.A Reed, D.D. Ganno., J.R. Larus., "Imagining the Future: Thoughts on Computing," Computer, vol. 45, no. 1, pp. 25-30, Jan. 2012.

[4] D. Laney, "Application Delivery Strategies", http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

[5] A. Zaslavsky, C. Perera, D. Georgakopoulos, "Sensing as a Service and Big Data", Proc. of International Conference on Advances in Cloud Computing, July 2012.

[6] L. Mottola and G. P. Picco. "Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. ACM Computing Surveys, 2010.

[7] Z. Shelby et al, "Constrained Application Protocol (CoAP)", Internet-Draft. draft-ietf-core-coap-12.

[8] R. Fielding "Architectural Styles and the Design of Network-based Software Architectures", Doctoral dissertation (2000),

[9] M. Kovatsch, S. Duquennoy, A. Dunkels, "A Low Power CoAP for Contiki", IEEE 8th International Conference on Mobile Adhoc and Sensor Systems (MASS), 2011

[10] P. Mell, T. Grance, "The NIST Definition of Cloud Computing", csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[11] M. Yuriyama and T. Kushida, "Sensor-Cloud Infrastructure - Physical Sensor Management with Virtualized Sensors on Cloud Computing,", Sept. 2010, pp. 1-8.

[12] G.C. Fox, S. Kamburugamuve, R.D. Hartman, "Architecture and Measured Characteristics of a Cloud Based Internet of Things API", International Conference on Collaboration Technologies and Systems (CTS), 2012,

[13] J. Melchor, M. Fukuda,. "A Design of Flexible Data Channels for Sensor-Cloud Integration", Proc. of the 2011International Conference on Systems Engineering, ICSENG2011

[14] M. Hassan, B. Song., E-N. Huh, "A Framework of Sensor - Cloud Integration Opportunities and Challenges", Proc. of the 3rd International Conference on Ubiquitous Information Management and Communication, ICUIMC '09

[15] C. Jardak, J. Riihijärvi, F. Oldewurtel, P. Mähönen, "Parallel Processing of Data from Very Large-Scale Wireless Sensor Networks", Proc. of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC10

[16] G. Picco, "Software Engineering and Wireless Sensor Networks: Happy Marriage or Consensual Divorce?", FoSER 2010.

[17] M. Weiser "The computer for the twenty-first century", Scientific American, September 1991 (reprinted in IEEE Pervasive Computing, Jan-Mar 2002)

[18] S. R. Madden, 'The Design and Evaluation of a Query Processing Architecture for Sensor Networks", Ph.D. Thesis. UC Berkeley. Fall, 2003

[19] T. Hasiotis et al, "Sensation: A Middleware Integration Platform for Pervasive Applications in Wireless Sensor Networks", 2nd European Workshop on Wireless Sensor Networks (EWSN), Istanbul, Turkey, January 2005.

[20] A. Boulis and M. B. Srivastava, "A Framework for Efficient and Programmable Sensor Networks", In Proc. of OPENARCH 2002, New York, June, 2002.

[21] C. Intanagonwiwat, R. Govinden, D. Estrin, J. Heidemann, F. Silva, "Directed Diffusion for Wireless Sensor Networking", IEEE/ACM Transactions on Networking, Vol 11, No. 1, February 2003

[22] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. "Programming wireless sensor networks with the TeenyLIME middleware", Proc. of the 8th Int. Middleware Conf., 2007

[23] D. Gelernter, "Generative communication in Linda", ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 7 Issue 1, Jan. 1985

[24] M. Ceriotti et al "Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment", Proc. of the 8th Int. Conf. On Information Processing in Sensor Networks (IPSN), 2009.

[25] X. Le et al , "Secured WSN-integrated Cloud Computing for u-Life Care", IEEE CCNC 2010.

[26] D. Clark et al, "Making the world (of communications) a different place.", ACM SIGCOMM CCR, 35(3):91–96, 2005.

[27] Z. Shelby et al, "The IPSO Application Framework", Internet-Draft. draft-ipso-app-framework-04, August 2012