2016 IEEE First International Conference on Internet-of-Things Design and Implementation

Unified Service-oriented Access for WSNs and Dynamically Deployed Application Tasks

Theodoros Fronimos[1], Spyros Lalis[1,2], Manos Koutsoubelias[1,2], Thomas Bartzanas[1]

[1]Institute for Research and Technology Thessaly
Centre for Research and Technology Hellas (CERTH)
Volos, Greece
{frontheo,bartzanas}@mail.ireteth.certh.gr

[2]Electrical and Computer Engineering Department
University of Thessaly
Volos, Greece
{emkouts,lalis}@uth.gr

Abstract—**For good reasons of interoperability and flexibility, wireless sensor networks (WSNs) are accessed by remote clients via service-oriented interfaces. At the same time, it can be useful to let clients deploy custom sensing and processing tasks directly on the sensor nodes. However, this raises the issue of how the remote client application can interact with such a task, in parallel to conventional access of the WSN. We propose an approach for adopting a unified web-based interface for both information flows. This way, WSN clients do not have to deal with different interface technologies. Moreover, the gateway of the WSN can handle both flows using the same protocol translation engine. The paper presents our design, briefly discusses a first prototype implementation, and shows an indicative usage example.**

Keywords—wireless sensor networks; middleware; service-oriented access; integration; Internet of Things

I. INTRODUCTION

Wireless sensor networks (WSNs) are a key component of the so-called Internet of Things (IoT), as they can support a broad range of monitoring and control applications. The integration of WSNs into the wider Internet-based landscape emerges as a challenging issue. To this end, suitable open interfaces are required, which will allow WSNs to be accessed from different external systems in a flexible way.

At the same time, a large body of work proposes WSNs that can be programmed in a dynamic way, by injecting application-specific code that runs directly on the nodes of the WSN. Such code can access the onboard sensors, process the measurements locally, and send data to external systems only when this is actually useful for the application. This approach can greatly reduce traffic in the WSN. Given that computing is much cheaper than communication, this can also prolong the lifetime of battery-powered nodes.

The ability to program a WSN introduces a second dimension to the aforementioned challenge of WSN access. On the one hand, several remote application clients may issue queries to the WSN, and receive the data that is produced by the sensor nodes, through a well-defined interface. The different types and semantics for the data that can be produced as a response to such queries are known in advance, and are part of the WSN interface. On the other hand, certain application clients may wish to inject their own sensing and processing code into the WSN, and receive the data that is produced by it. However, this code may have to be controlled via custom commands and can produce custom data, which in the general case are known only to the application. The question is how to support both “worlds” at the same time.

In this paper, we present an end-to-end solution to this problem. At the WSN nodes, we use a middleware layer, which supports conventional sensor queries as well as the dynamic installation and execution of application tasks on the node. At the WSN gateway, a mediator engine supports a service-oriented access of the WSN, but also the installation of and interaction with application-specific tasks. Based on appropriate XML descriptions, the mediator engine can handle, in a uniform way, both conventional queries and task-specific commands, and return the data that is produced in each case to the respective clients. Notably, it is possible to have several outstanding queries and tasks running in the WSN concurrently to each other.

The rest of the paper is structured as follows. Section II gives an overview of related work. Section III describes the high-level architecture of our approach. Section IV presents the interface of the WSN towards external clients. Section V describes the programming model supported by the WSN node middleware. Section VI provides some details about our current implementation, and Section VII gives a concrete application example. Section VIII concludes the paper.

II. RELATED WORK

Numerous publications address the issue of creating high-level abstractions for sensor networks from different points of view. Proposals such as TinyDB [1] and Cougar [2] follow a macro-programming approach by abstracting the whole network as a single database on which the user can perform complex queries. The main drawback is that the user has no control over in-network operations, as their optimization relies solely on the underlying middleware.

Contrary to macro-programming, virtual machine based approaches allow users to distribute processing among network nodes, in the form of mobile code. In middleware systems like Agilla [3] and Maté [4], mobile agents carrying user defined programs are injected or flooded throughout the sensor network. Programs are composed of special, low-level instructions (opcodes) following an assembly-like style. The overhead for performing application updates is much lower compared to methods that substitute the whole firmware image. However, application programming tends to be tedious and error prone.

To simplify WSN programming, TENET [5] introduces a set of predefined high-level operations, called tasklets, which are supported by the nodes of the WSN and can be executed on demand. The programmer can then compose application-specific tasks in a very simple way, as a sequence of such operations. A task is encoded as a character string, sent to the sensor nodes for execution. Servilla [6] introduces a C-like language for the same purpose. However, in both cases, the code of all tasklets must be part of the pre-installed node firmware. POBICOS [7] allows application-specific agents to be written in C, and supports a fully dynamic placement and migration of such agents in the WSN.

A recent trend are service-oriented architectures based on web services for which support is readily available in most popular programming languages. In [8], nodes publish their basic functionalities as services to a base station, which in turn provides the necessary abstractions in order to be easily manipulated by a high level programming language of the user's choice. TinyWS [9] allows direct HTTP access on implemented node services through a SOAP processing engine residing on top of the platform's OS, while the various implementations of the CoAP protocol [10] follow a RESTful design. While easy to use, these frameworks do not support any intra-network processing.

Combining elements of the above approaches seems a more promising strategy. In [11] authors integrate TENET in a WSN-SOA to support task deployment and collaboration between nodes as a service. In USEME [12], applications can be composed via a declarative language which specifies nodes, groups and services; however, service implementation depends on the underlying OS. ProFun TG [13] is based on a declarative network programming model that utilizes abstract task graphs and dataflows. A visual IDE is provided for handling tasks, but the user is also given the option of programming application-specific tasks in C or SEAL [14]. Task allocation and data handling is feasible through JSON requests over HTTP.

While the task-based approaches in [11] [13] and the multi-layered software architecture in [15] offer advanced programming features, user-defined software must be part of the initial firmware. In contrast, our work focuses on providing easy to use interfaces that allow flexible data manipulation, node reconfiguration and task reprogramming, even after deployment.

III. System Architecture

The high-level system architecture of our approach is depicted in Fig. 1, highlighting the most important elements. The WSN consists of several sensor nodes and a gateway. The gateway provides an interface to external clients so that they can access/control the WSN sensor nodes and receive data from them. In addition, through this interface, clients can deploy application-specific tasks and interact with them.

The WSN gateway acts as a mediator between external clients and the WSN. It maps client requests to WSN-level command packets that are sent to the sensor nodes. Conversely, it maps the WSN-level data packets received from sensor nodes to replies for the respective clients. Sensor nodes run suitable middleware that handles requests coming

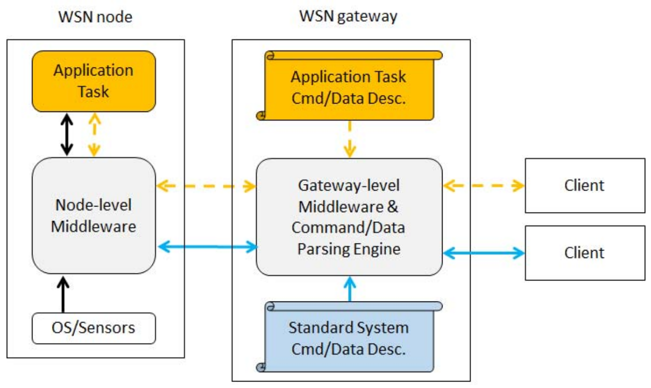
WSN node
WSN gateway
Application Task
Application Task Cmd/Data Desc.
Node-level Middleware
Gateway-level Middleware & Command/Data Parsing Engine
Client
Client
OS/Sensors
Standard System Cmd/Data Desc.

Figure 1. High-level system architecture.

from the gateway and sends back replies and data. The middleware is also responsible for installing and running application tasks on the sensor node, as well as for facilitating their interaction with the external application systems, via the gateway.

The translation between the client-level protocol and the WSN-level protocol is performed based on suitable metadata descriptions. This holds both for the standard (fixed) client functions, as well as for the (open) interaction between the tasks and the client applications that deployed them. The former metadata descriptions are part of the gateway firmware, whereas the latter are registered with the gateway before installing a task. As indicated in Fig. 1, this makes it possible for the gateway to use the same mediator engine for both cases, and for clients to interact with the WSN in a unified way.

IV. WSN Client Interface

The interface provided by the WSN gateway to external clients follows a web-based design. This way, the WSN can be accessed in a platform-neutral way, from any computer with a connection to the Internet. The WSN interface consists of two parts: the standard interface and the task interface. The standard interface allows remote clients to control the operation of the sensor nodes, issue sensor queries and receive sensor measurements, without using a task. The task interface is used by external application systems to install tasks into the WSN, control their operation, and receive the data produced by them. Below, we discuss the most important services in more detail. Note that selected services could be enabled on a per-client basis, depending on the access rights given to it by the gateway administrator.

A. Sensor Node Configuration Services

The ability to fine tune system parameters at a post-deployment stage can be very important in achieving an efficient operation and monitoring of the WSN. For this purpose, the gateway provides a set of configuration services, e.g., for setting the transmission power of a sensor node, or the frequency of liveness messages. The latter contain information about the current status of a node, including the transmission power, the battery voltage, the quality of the wireless link with the gateway, the tasks installed, and elapsed time since the last node reboot/reset. This information is collected at the gateway, to keep an updated view of the WSN, which can be retrieved by external clients/users via a corresponding request.

B. *Sensor Access Services*

There are two options for obtaining measurements from sensor nodes. The first one is by sending a one-off query request, to which a reply is produced promptly. An example is given in Table I, for a query targeting the temperature and humidity sensors of a node (the entire WSN can be addressed using a broadcast address in the request).

TABLE I. SERVICE REQUESTS/REPLIES FOR SENSOR QUERIES

one-off query	query?addr0=12&addr1=25&sensors=temp,hum
	{addr:[12,25],rssi:60,lqi:107,deluge:'off',version:0, tx_power:31,time:'2015:09:18:23:36:58', vals:{bat:3.1, temp:24.35,hum:56.267}}
periodic query	{addr:[12,25],request:'subscribe', service:'periodic' rates:{temp:60,hum:120}}
	{addr:[12,25], …, vals:{bat:3.1, temp:24.35,hum: 56.267}} {addr:[12,25], …, vals:{bat:3.1, temp:24.35}}

The second option is to submit a long-lived query for periodic sensor measurements at a fixed rate. In this case, the client opens a web-socket connection and sends a JSON object which defines the sensors to be sampled and their sampling rate. The client then receives over the same socket the data produced, in the form of JSON objects. The gateway keeps the query active as long as the respective web-socket connection remains open, otherwise the query is cancelled. As an example, Table I shows a query request for the temperature and humidity sensors, and the data objects that will be produced in response. Note that some data objects will only contain temperature values, as the temperature is sampled two times more frequently than humidity.

The gateway multiplexes queries issued concurrently by different clients. It will forward a new client query to the sensor nodes only when the query cannot be satisfied based on the queries that are already running. In a similar vein, it de-multiplexes the data produced by the sensor nodes to the respective clients, ensuring that each client will receive the data it requested. Moreover, the gateway caches the values recently received from sensor nodes, just in case these can be re-used to satisfy new queries.

C. *Task Installation Services*

The gateway offers a set of services so that external clients can dynamically deploy application tasks in the WSN. This is done following a stepwise process.

The first step is for the client to upload on the gateway the task binary, along with task-specific descriptions for the configuration commands that can be handled by the task and the data objects that are produced by it – there are XML files that must follow a specific format, described in the sequel.

Next, the client activates the installation mode on the target sensor nodes, and then triggers the actual task installation process. Finally, when done, the installation mode has to be deactivated. Table II shows indicative service requests for the last three steps of the process.

TABLE II. SERVICE REQUESTS FOR TASK INSTALLATION

activate	startinstall?addr0=12&addr1=25&file=test&size=2048
install	install?file=test&size=2048
deactivate	stopinstall?addr0=12&addr1=25

Note that code installation must be activated and deactivated in an explicit way. This is to avoid any background network traffic (and energy cost) that might be incurred by the respective WSN-level service, as long as it remains active.

D. *Task Configuration and Data Services*

Once an application task is installed on a sensor node, the client can interact with it, through the gateway, via the corresponding configuration and data services. Note that each task can have its own application-specific configuration parameters and produce its own application-specific types of data. These are not known in advance – contrary to the available sensors of the WSN nodes, which do not change in time. For this reason, the respective services are left open, and their syntax is defined via suitable description files.

More specifically, the task configuration service takes as parameters the node address and the identifier of the command, followed by zero or more task-specific parameters. The name of the configuration command identifier, the number and type of its parameters, and the order in which these should appear in the client request, are specified via an XML description, formatted according to the DTD in Table III. The gateway uses this description to parse client requests and verify that they conform to the specification, before it maps them to WSN-level packets that will be sent to the sensor nodes.

TABLE III. DTDS FOR TASK-SPECIFIC XML DESCRIPTION FILES

Application Task Commands	<!ELEMENT commands (command*)> <!ELEMENT command (uname,code,par*)> <!ELEMENT uname (#PCDATA)> <!ELEMENT code (#PCDATA)> <!ELEMENT par (uname,valtype)> <!ELEMENT valtype (#PCDATA)>
Application Task Data Objects	<!ELEMENT datamessages (datamsg*)> <!ELEMENT datamsg (uname,code,data*)> <!ELEMENT uname (#PCDATA)> <!ELEMENT code (#PCDATA)> <!ELEMENT data (uname,valtype,size?)> <!ELEMENT valtype (#PCDATA)>

Similarly, the data that can be produced by the application task comes in the form of JSON objects, whose structure is defined via a corresponding XML file, following the DTD in Table III. Each *datamsg* element represents a separate data object, which can include one or more *data* elements; optionally, a *size* element can be used to indicate an array of values of the same type and name. Importantly, the order of elements inside the data object defines the order in which the application task is expected to serialize the respective values in the WSN-level packet sent to the gateway, and the naming convention for the data items in the JSON object that will be ultimately delivered to the client.

The client can retrieve the data objects produced by the task in two different ways. The first option is to poll the gateway to receive a file with the data that was produced by the task up to that point. The corresponding service request takes as a parameter the node address and a flag indicating whether the contents of the file should be erased after the transfer completes. The second option is for the client to be notified when the task produces new data, via a web-socket, in the same way this is done for periodic sensor queries.

E. Sensor Node Middleware

The middleware running on the sensor nodes implements the access services provided by the WSN gateway to external clients, on the node itself. The interaction between the sensor nodes and the gateway occurs via a suitable WSN-level protocol that uses a binary encoding for compactness.

One of the main responsibilities of the middleware is to enable the dynamic installation and execution of application tasks via a suitable runtime environment (see next section). Moreover, whenever possible, the middleware bundles together data produced from different sensor queries and tasks that run locally, in a single packet. This can greatly reduce the number of individual packet transmissions in the WSN, which in turn increases reliability and saves energy.

V. Task Programming Model

Application tasks correspond to independent, event-driven software components – tasks do not have a control thread of their own. The task has three event handlers: *on_init*, *on_sensordata* and *on_command* (see Table IV). The *on_init* handler is called right after task installation, to let the task perform initialization actions. The middleware invokes the *on_sensordata* handler as soon as sensor measurements become available, so that the task can process them as needed. The sensors for which values are available and the corresponding measurements are passed to the task. Finally, *on_command* is called when the external client issues a configuration command for the task, passing the identifier of the command and a buffer with marshalled values for each of the command parameters, as per the corresponding XML description.

It is the responsibility of the application programmer to provide these handlers. The middleware offers a set of primitives that can be invoked from within the task in order to implement the desired functionality, listed in Table IV.

The *set_sensors* primitive is used to specify the sensors the task wishes to use and the sampling rate for each one of them. This is typically done from within the *on_init* handler. Based on this setting, the middleware periodically performs the desired low-level sensor access, and up-calls the *on_sensordata* handler to forward the values to the task.

The task must store the data it wishes to send in a buffer which it has to manage itself. This can be a global variable that persists across handler invocations, or a temporary variable used in the context of a single handler invocation.

TABLE IV. Event Handlers and API for Application Tasks

Event handlers	
on_init(void)	Initialize task state
on_sensordata(int mask, float *vals[])	Handle new sensor values
on_command(char id, char *buf)	Handle config. command
Sensor primitives	
set_sensors(int mask, int rates[])	Set the sensors to use and their sampling rates
Outgoing data primitives	
setType(char *buf, int type)	Set message buffer type

writeUInt8(char *buf, int *pos, int v)
writeUInt16(char *buf, int *pos, int v)
…
writeFloat(char *buf, int*pos, float f)
	Data marshalling routines

send(char *buf, int len)	Send message buffer
Incoming command primitives	

readUInt8(char *buf, int *pos, char *v)
readUInt16(char *buf, int *pos, int *v)
…
readFloat(char *buf, int*pos, float *f)
	Data unmarshalling routines

The type of the message buffer is set via *setType*, to one of the data object identifiers defined in the XML description. The payload is marshalled into the buffer using the corresponding *write** primitives. The programmer is responsible for writing data in the order specified in the XML description, so that data can be properly read/decoded by the gateway. To save space in the WSN-level packets, the marshalling routines do not add any extra type information. The only type information that is available in the packet is the data object identifier. Finally, the message buffer is sent over the network via the *send* primitive.

A similar approach, in the reverse direction, is followed for incoming configuration commands. In this case, the task retrieves the corresponding parameter values via the *read** unmarshalling primitives. Again, this must be done in the order given in the corresponding XML description.

VI. Implementation Details

We have implemented the node middleware on top of Contiki [16], a well-documented open source operating system ported to a wide range of hardware platforms. Contiki also supports different protocols such as ContikiMAC [17] and Rime [18], that are reliable and energy efficient.

Application tasks are written in C, following the model described in the previous section. Task execution is based on the Contiki programming model [19]. Each task is a Contiki process, executed by the kernel along with other system-level processes, and communicates with them via the Contiki IPC mechanisms. However, all Contiki-specific aspects are hidden from the programmer, who only needs to write the three event handlers using the task API. The code is then passed through a pre-processor, which generates the full Contiki program.

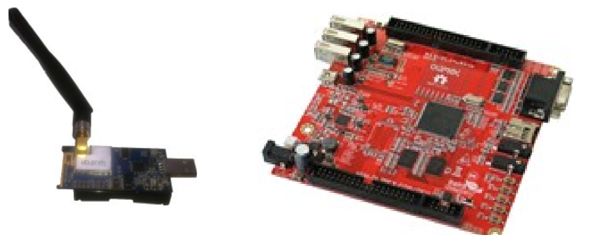
(a) (b)

Figure 2. Hardware of the system prototype: (a) node; (b) mini PC.

Figure 3. Wireless sensor node in a greenhouse.

Using the existing Contiki development tool chain, this code is converted into a dynamically loadable binary (ELF). Contiki provides support for loading ELF code on RAM via a procedure known as relocation [20]. The transfer of an ELF file from the WSN gateway to one or more sensor nodes is done via the Deluge protocol [21]. Finally, the ELF files are stored on the flash memory of the sensor node (from where they can then be retrieved by the ELF-loader) using the Coffee file system [22].

As a hardware platform for the sensor nodes we use TelosB nodes from AdvanticSYS (Fig. 2a), with 48K of ROM and 10K of RAM. The WSN gateway software is developed based on the Python CherryPy framework, and runs on a mini-PC (Fig. 2b) on top of a Linux environment. The interface to the WSN is a sensor node that is connected to the mini-PC over serial, which plays the role of the "data sink" for the WSN. In turn, the mini-PC can be accessed from the Internet over a wireless or wired connection, depending on the setup. Our prototype currently supports a star network topology, where every sensor node has a direct wireless link to the sink node. Given the rich networking support of Contiki, it is rather straightforward to extend our implementation to support multi-hop topologies.

VII. APPLICATION EXAMPLE

We give a concrete example for an application that deploys a task into the WSN placed in a greenhouse (Fig. 3), to monitor environmental parameters that can affect crop health and performance. This information feeds into a model that controls the greenhouse environment and watering process in order to achieve optimal crop growth conditions. One of the parameters of the model is the Vapor Pressure Deficit (VPD) value inside a greenhouse. This is calculated as a function of temperature (T) and relative humidity (RH): $(0.611*\exp(17.27*T/(T+237)))*RH/100$. Let us assume that the model does not need to receive updates on this value, as long as there is no large deviation compared to the old value.

The conventional way would be to periodically query the sensor nodes for T and RH, in the spirit of the service examples given in Table I, and then calculate the VPD on the machine that runs the model. But this would be inefficient, especially if the VPD value changes rarely.

Instead, one can use a task that performs this calculation locally, and sends updates only when the current VPD differs significantly from the last reported value. To increase flexibility, the task can have a configurable deviation threshold. Listing 1 gives the code for the application task, and Table V provides the XML descriptions for the configuration command supported by the task and the data that is produced by it.

```
#define SETTHRESHOLD_CMD 1
#define UPDATE_MSG  1
#define DEFAULT_DIFF 0.5
#define TEMP_SENSOR 1
#define HUM_SENSOR 3

static float diff=DEFAULT_DIFF, tmp,hum,lastvpd;
static char mbuf[1+sizeof(float)];

float calcVPD(float t, float rh){ … }
int getPos(int bit, int mask) { …}

on_init() {
   setType(mbuf,UPDATE_MSG);
   int rates[]={60,60};
   set_sensors(2, TEMP_SENSOR | HUM_SENSOR,rates);
}

on_command(int id, char *buf) {
   int pos=0;
   if (id == SETTHRESHOLD_CMD)
       readFloat(buf, &pos, &diff);
   }
}

on_sensordata(int mask, float vals[]){
   int pos,update=0; float vpd;

   pos = getPos(TEMP_SENSOR,mask);
   if (pos > 0) { tmp = vals[pos]; update = 1; }
   pos = getPos(HUM_SENSOR,mask);
   if (pos > 0) { hum = vals[pos]; update = 1; }
   if (update) {
      vpd = calcVPD(tmp,hum);
      if (abs(vdp – lastvpd) > diff) {
         pos = 1;
         writeFloat(mbuf,&pos,vdp);
         send(buf,pos);
         lastvdp = vpd;
      }
   }
}
```

Listing 1. Code for the VPD task

TABLE V. XML DESCRITPIONS FOR THE VPD TASK

Task-specific command	`<command>` `<uname>setthreshold</uname>` `<code>1</code>` `<param>` `<uname>absdiff</uname>` `<valtype>float</valtype>` `</param>` `</command>`
Task-specific data object	`<datamsg>` `<uname>update</uname>` `<code>1</code>` `<data>` `<uname>vpd</uname>` `<valtype>float</valtype>` `</data>` `</datamsg>`

The task code is 42 lines long, and compiles into a binary of roughly 2.200 bytes, out of which about 900 is the code of the API added by the pre-processor. Assuming the task runs on the sensor node with address *12.25*, the client can set/change the notification threshold via a request `http://server/taskcmd?adrr0=12&addr1=25&cmd=setthreshold&absdiff=0.95`, and receive data in the form `{addr:[12,25],time:'2015:09:18:23:36:58',update:{vpd:1.45}}`.

It is easy to imagine different versions of such a task, with a more complex application logic and commands for setting additional configuration parameters, which can also produce more than one types of data objects. Note that it is trivial for the external application to replace the task at any point in time, e.g., to correct a bug, or to use a more advanced version.

VIII. CONCLUSION

We presented a unified service-oriented access to a WSN for conventional queries as well as for the interaction with application-specific tasks deployed in the WSN dynamically. This greatly increases flexibility, without introducing extra interfacing complexity for the clients of the WSN.

One of the enhancements we wish to pursue in the future is to extend the task programming model to support application-specific in-network processing and multi-hop data forwarding within the task itself. We also want to experiment with mechanisms for adapting the operation of the sensor node, in an application-aware manner, to further reduce the energy consumption of sensor nodes.

ACKNOWLEDGMENT

This work was funded in part by the General Secretariat for Research and Technology of Greece, action KRIPIS, project “Smart Pole for Specialization and Development of Thessaly: Research, Innovation, Strategies”.

REFERENCES

[1] S. R. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong. “TinyDB: an acquisitional query processing system for sensor networks,” ACM Trans. Database Syst. 30(1), pp. 122-173, 2005.

[2] Y. Yong and J. Gehrke. “Query Processing in Sensor Networks.” CIDR. 2003, pp.233-244.

[3] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn and K. R. Walker. “Agile application-aware adaptation for mobility,” SIGOPS Oper. Syst. Rev. 31(5), pp. 276-287, 1997.

[4] P. Levis and D. Culler. “Maté: a tiny virtual machine for sensor networks,” SIGARCH Comput. Archit. News, 30(5), pp. 85-95, 2002.

[5] O. Gnawali, B. Greenstein, K. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan and E. Kohler. “The tenet architecture for tiered sensor networks,” 4th Intl Conf on Embedded Networked Sensor Systems, pp. 153-166, 2006.

[6] C.-L. Fok, G.-C. Roman and C. Lu. “Servilla: A flexible service provisioning middleware for heterogeneous sensor networks,” Sci. Comput. Program. 77(6), pp. 663-684, 2012.

[7] N. Tziritas, G. Georgakoudis., S. Lalis, T. Paczesny, J. Domaszewicz, P. Lampsas and T. Loukopoulos. “Middleware mechanisms for agent mobility in wireless sensor and actuator networks,” 3rd Intl Conf on Sensor Systems and Software, pp. 30-44, 2012.

[8] E. Avilés-López, J. Antonio García-Macías. “TinySOA: a service-oriented architecture for wireless sensor networks,” Service Oriented Computing and Applications, 3(2), pp. 99-108, 2009.

[9] N. B. Priyantha, A. Kansal, M. Goraczko and F. Zhao. “Tiny web services: design and implementation of interoperable and evolvable sensor networks,” 6th ACM Conf on Embedded Network Sensor Systems, pp. 253-266, 2008.

[10] M. Kovatsch, S. Duquennoy and A. Dunkels. “A low-power CoAP for Contiki,” IEEE Intl Conf on Mobile Adhoc and Sensor Systems, pp. 855-860, 2011.

[11] B. Le Corre, J. Leguay, M. Lopez-Ramos, V. Gay and V. Conan. “Service Oriented Tasking System for WSN,” IEEE Intl Conf on Developments in E-Systems Engineering, pp. 64-69, 2010.

[12] E. Caete, J. Chen, M. Diaz, L. Llopis and B. Rubio, “USEME: A Service-Oriented Framework for Wireless Sensor and Actor Networks,” 8th Intl Workshop on Applications and Services in Wireless Networks, pp. 47-53, 2008.

[13] A. Elsts, F. H. Bijarbooneh, M. Jacobsson and K. Sagonas. “ProFuN TG: A Tool for programming and managing performance-aware sensor network applications,” 40th IEEE Intl Workshop on Local Computer Networks, pp. 751-759, 2015.

[14] A. Elsts, J. Judvaitis and L. Selavo, “SEAL: a domain-specific language for novice wireless sensor network programmers,” EUROMICRO SEAA, pp. 220–227, 2013.

[15] G. Fortino, A. Guerrieri, G. M. O'Hare and A. Ruzzelli, “A flexible building management framework based on wireless sensor and actuator networks.” Journal of Network and Computer Applications, 35(6), pp. 1934-1952, 2012.

[16] A. Dunkels, B. Gronvall and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” 1st IEEE Workshop on Embedded Networked Sensors, pp. 455-462, 2004.

[17] A. Dunkels. "The ContikiMAC radio duty cycling protocol,” SICS Technical Report T2011:13, 2011.

[18] A. Dunkels. “Rime - a lightweight layered communication stack for sensor networks,” European Conf on Wireless Sensor Networks, poster, 2007.

[19] A. Dunkels, O. Schmidt, T. Voigt and M. Ali, “Protothreads: Simplifying event-driven programming of memory-constrained embedded systems,” 4th ACM Conf on Embedded Networked Sensor Systems, pp. 29-42, 2006.

[20] A. Dunkels, N. Finne, J. Eriksson and T. Voigt, “Run-time dynamic linking for reprogramming wireless sensor networks”, 4th ACM Conf on Embedded Networked Sensor Systems, pp. 15-28, 2006.

[21] J. W. Hui and D. Culler. “The dynamic behavior of a data dissemination protocol for network programming at scale,” 2nd Intl Conf on Embedded Networked Sensor Systems, pp. 81-94, 2004.

[22] N. Tsiftes, A. Dunkels, Z. He and T. Voigt, “Enabling large-scale storage in sensor networks with the coffee file system,” 8th ACM/IEEE Intl Conf on Information Processing in Sensor Networks, pp. 349-360, 2009.