A System of Systems Architecture for the Internet of Things exploiting Autonomous Components

Basil Nikolopoulos*
Alexandros C. Dimopoulos
Mara Nikolaidou
George Dimitrakopoulos
Dimosthenis Anagnostopoulos

Department of Informatics & Telematics,
School of Digital Technology,
Harokopio University, Athens, Greece
E-mail: {basil, alexdem, mara, gdimitra, dimosthe}@hua.gr
* Corresponding authors

Abstract: As the Internet of Things (IoT) becomes more popular, supporting systems and their components become more complex and largely heterogeneous. This paper discusses on a System of Systems (SoS) architecture for IoT systems composed by autonomous components. The proposed architecture focuses on a middleware transforming sensor services to REST services, for the development of mixed-criticality applications. The middleware consisting of autonomous aggregation software running on commodity multi-core devices, such as Raspberry Pi. Self-management policies applied are discussed in the paper. The analysis of a smart building system, developed as a use case, provides solid evidence that such an architecture is realistic and can lead to highly competitive systems.

Keywords: IoT; SoS; Edge Devices, Aggregation Software; Autonomous components; self-management policies

Biographical notes:

Basil Nikolopoulos is a Ph.D. student at Harokopio University of Athens, faculty of Digital Technology. He earned his University degree (2009) on Informatics and Telematics and his M.Sc. (2013) on Computational and Internet Technologies and Applications. His research is around the area of Context-aware IoT and Autonomic Approaches targeting Multi-critical Applications in the IoT. He is currently working as a software engineer in the Cyber Industrial Intelligence Industry, while also working at his recently start up business. He has numerous articles published in international scientific conferences.

Alexandros C. Dimopoulos received in 2004 the diploma of Electrical and Computer Engineering from the National Technical University of Athens. At the same institution, he completed in 2009 his Ph.D. entitled âŁœEfficient Embedded SystemsâŁž, implementing a platform for the automated creation of hardware. He is a postdoc researcher in the BSRC Alexander Fleming, in the field of bioinformatics, while in parallel he teaches undergraduate and graduate courses at the department of Informatics and Telematics of Harokopio University, as an adjunct lecturer. His research interests cover the areas of software/hardware co-design, of Embedded Systems, of parallel computing and of bioinformatics applications with next generation sequence data. Dr. Dimopoulos has eleven

publications in international journals and more than twenty-five in international scientific conferences.
MARA
George Dimitrakopoulos is an assistant professor at the department of Informatics and Telematics of Harokopio University of Athens. He received his diploma in electrical and computer engineering from NTUA (2002) and his Ph.D. on spectrum and radio resources management from University of Piraeus (2007). He has been participating in numerous internationally funded RD projects and he is the author of more than 125 publications. His research interests revolve around intelligent transport systems, cognitive networks and 5G communication technologies and related applications.
Dimosthenis Anagnostopoulos is a Professor in the Department of Informatics and Telematics of Harokopio University of Athens in the area of Information Systems and Simulation. He served as Rector of Harokopio University (9/2011 âŁ“ 1/2016) and was also elected Chair of the Greek RectorsâŁ™ Conference (1/2014-6/2014). He is currently the Dean of Faculty of Digital Technology. He is a visiting Professor at Sussex University, UK and an Associate Editor of Requirements Engineering (Springer). His research interests include Information Systems, eGovernment, Semantic Web and Web Services, Modeling and Simulation, Business Process Modeling.

1 Introduction

In the last few years, more and more devices of everyday usage are becoming connected to the Internet, even devices that traditionally did not have any connections. The Internet of Things (IoT) promotes the interconnection of such devices, integrating them into a large network capable of offering numerous services to the end user. IoT technology is applied in different areas, from e-health [1] and transportation [2] to smart city [3] and building operation [4].

As IoT services become more popular, clearly the systems supporting them become more complex. There are numerous efforts [5] to promote distributed architectures for implementing complex IoT systems, ensuring their scalability. Following recent trends in Edge [6] and Fog [7] computing for IoT, the concept of providing an intermediate computing layer operating closer to the sensors and user devices promotes scalability, availability and performance. Instead of gathering raw data from sensors in centralised processing nodes, e.g. cloud servers, and centrally making all decisions regarding the configuration of an IoT system, there is an alternative approach to add device controlling nodes, called aggregators or gateways, to aggregate data from all the available devices and offer corresponding service-oriented architecture (SOA) services. [8]. The existence of an aggregator helps heterogeneous interconnected devices to collaborate seamlessly, while the autonomous operation of such components, utilizing the context they operate in [9], ensures the scalability of IoT systems, thus is of great importance. These aggregator services may run in commodity multi-core devices, e.g. Raspberry Pis, to control the sensors and the provided services to IoT user applications. The aggregators contribute to the enhancement of transparency, since device-specific details are hidden from the IoT application users.

In general, every portable or embedded implementation is bound by a major restrain, i.e. limited power supply. Typically, a wireless sensor network (WSN) [10] is bounded by the finite power source of its consisting sensor nodes. It has been proven [11] that in terms of energy consumption it is much more expensive than data processing: the required energy

for transmitting a single bit is equal to that needed for executing thousands of operations inside the node [12]. In addition, the power consumption of a processing unit is directly linked to the number of processing elements (i.e. cores) that are active.

Usually, the existence of an aggregator contributes in the energy conservation of each sensor and the whole network itself [13]. The aggregator unit has the responsibility of collecting data from each sensor; not necessarily following an identical procedure for all sensors. Although, in many cases [5, 9] aggregators become intelligent and may obtain some self-configuration features, there is a need for a generic approach towards a software architectural framework for implementing self-management policies in a unified and extendable fashion. The authors have previously presented the design and deployment of smart building SOA systems consisting of autonomous, cognitive components, targeting self-configuration and self-optimization features [14]. Moreover, in [15, 16] a context-aware, smart building management multi-core architecture, for tackling mixed criticality applications in an IoT context was presented.

In the currently presented work, the aggregator is implemented as autonomous, based on specific self-X properties, via the exploitation of which the end user of the system continues to receive all the requested services as expected; however, the aggregator controls more efficiently the flow of information and, if possible, limits the number of transmitted data. In a nutshell, based on the criticality of a service and the existence or absence of cached information, the aggregator controls the way the sensor measurements are going to be read, driven by the overall energy consumption. Within the aggregator, the number of active computer cores significantly defines the power consumption of the aggregator itself. The deployment of applications that follow a multi-critical scheme, allows the aggregator to organize the requests based on their criticality and apply load-balancing techniques in an attempt to keep the active cores as loaded as possible, while suspending needless cores.

As a proof of concept, of the proposed architecture, a case study is used, where the issue of power consumption is tackled while maintaining the system performance at very satisfying level. More specifically, a two-fold strategy regarding energy efficiency is followed. At the first level, the transmission model is enhanced by limiting the number of not imperative communications and thus increasing the power source duration. And in the second level, the number of active cores is controlled and according to the system's needs all non-essential cores are not used and thus their power consumption is drastically reduced. The decrease in power consumption is accumulated from both levels, while the quality of service is preserved within the acceptable levels. Moreover, the decisions regarding the most energy efficient policy are imposed automatically by the proposed implementation, in a seamless for the user way.

Hence, in the rest of the paper the proposed architecture is throughly presented, in an attempt to highlight its automocity, combined with the underlying self manged policies. In Section 2 a quick introduction of the necessary background is given. In Section 3 the details of the proposed architecture are introduced from a more abstract perspective, while in Section 4 an in-depth analysis of the architecture's core module is given. In Section 5 the proposed implementation is described through a use-case taken from the smart building area and the experimental results of this scenario are presented in Section 6. Finally, Section 7 concludes the discussion on the design and deployment of the presented architecture.

2 Related Work

In the following subsections and in order to maintain the paper as self-contained as possible, a quick introduction of the necessary background is given, along with a brief review of the related work.

2.1 Service-Oriented Architecture (SOA) for IoT

In the IoT era more and more devices are becoming Smart Devices, while the number and variety of resources available in the field of IoT have increased dramatically. These resources (deployed devices) are inevitably heterogeneous and differ in many aspects [5] and this certainly increases the difficulty in managing the derived systems and thus makes it inefficient to manually access and control them. A solution to this heterogeneity is the usage of service-oriented architectures (SOA), providing an interoperable way of communication. However, SOA concepts were originally designed for dealing mainly with few, complex and mainly static enterprise services [17]. Therefore, the trend is to create SOA web services, allowing the horizontal and vertical collaboration among IoT devices, online services, users, etc. [17]. Based on the very needs, available technology and applied specifications, various models have been presented for creating multi-layer SOAs. The model presented by the International Telecommunication Union (ITU) consists of five different layers [18] while other researchers propose either three layers [5] or four [19]. Typically, a web service model consists of a service provider, a service registry and a service consumer [20]. For a client application (service consumer) to connect to such a server, a network address and port is essential to be known [21], as well as information on how to communicate (protocol, language, and mechanisms to use). This information can be obtained using a registry service, such were UDDI (Universal Description, Discovery and Integration) which were the materialization of the SOA registry component for publishing and discovering Web services [22] or other more recent forms of resource discovery [23].
The IoT systems are becoming cheaper, smaller in size and more capable as time passes. One of the most promising capability that augments the potentials of the IoT is Context. Context aware systems are systems that can handle the context information and use it to their gain. In general, context-aware computing has been introduced as a key feature in IoT systems over the last years and a lot of work has been done that demonstrate the importance of context awareness. Some early works like CoolTown [24] and work by Henricksen et al. [25] highlight the importance of context aware computing. In order to successfully implement and manage context models, many different techniques have been presented in the literature [26]. Another aspect of the IoT era is the multi-criticality of the applications that will be deployed. A multi-criticality system is a system where every service, or job of the system is characterised by a level of importance [27]. It must be noted that criticality in IoT applications is usually not real-time but quasi real time, i.e. the responses/decisions are not expected in milliseconds or microseconds but can be reported in quite few milliseconds or even seconds. The latter response time is not considered a problem for most of the IoT applications.

2.2 Aggregation

Furthermore, instead of gathering raw data from sensors in centralised processing nodes, e.g. cloud servers, and centrally making all decisions regarding the configuration of an IoT

system, there is an alternative approach to add device controlling nodes, called aggregators or gateways, to aggregate data from all the available devices and offer corresponding SOA services. [8]. The existence of an aggregator helps heterogeneous interconnected devices to collaborate seamlessly. If no aggregator was to be used, then a standardization of the APIs used by all devices would be necessary. The latter is very difficult - if not impossible - given the large number of different manufactures at a virgin field such as IoT, where each additional function a manufacturer adds is considered to be added value of the specific device [28].

The existence of aggregators or gateways has also been adopted by potential standards, as Edge [6] and Fog [7] computing for the IoT, providing an intermediate computing layer operating closer to sensors and user devices. Such architectures have already been adopted by specific IoT application areas, as for example for smart-health services [29]. Following the proposed concepts of Edge or Fog nodes, aggregator services may run in multi-core devices, (e.g. Raspberry Pis), to control sensors and provided services to IoT user applications entering their area of control. Aggregators may contribute to the enhancement of transparency, since they hide device-specific details form IoT application users. The aggregator unit has the responsibility of collecting data from each sensor; not necessarily following the same procedure for all sensors. To deal with the additional complexity introduced, aggregators should operate in an autonomous fashion and become self-managed, based on the context they operate in. Moreover, the existence of an aggregator may contribute in the energy conservation of each sensor and the whole network itself [13].

2.3 Smart building management and energy efficiency

According to Eurostat, the consumption of energy in the EU that corresponds to buildings is around 40% [30]. Thus, a large initiative of Europe 2020 strategy regards energy efficiency measures in buildings [31]. The energy consumption is controlled via a BMS (Building Management System) capable of controlling lighting, heating units and other energy hungry subsystems. A major issue for such systems is the lack of standardization for the metadata needed in identifying sensors and hence it has to be done in a great extend manually. There are different proposals in the literature such as Zodiac [32] that automatically classifies, names and manages sensors based on active learning from sensor metadata or the one in reference [33] that uses knowledge from expert-provided examples. Other approaches such as the one in reference [34], utilize linguistic and semantic techniques for computing similarity values between labels of sensors and BMS inputs. While IMPReSS SDP [35], aims at rendering the development of BMS less complex and more cost effective by allowing external developers to create energy management services that exploit its architecture.

An IoT implementation that tackles with building energy efficiency must find ways to reconcile the conflicting concepts of energy efficiency and occupant comfort. Such an implementation commonly relies on a WSN, since WSNs have proven to be very valuable for monitoring and surveillance in various application fields, from monitoring health [36] and natural environment [37] to agriculture [38].

However, since a WSN is bounded by the finite power source of its consisting sensor nodes, a major effort has been given in the direction of developing energy-saving techniques. These techniques can be divided into two significant categories: the ones that focus on the energy efficiency of the network itself and those that focus on finding ways to reduce the frequency of the costly transmissions.

For the first category, various schemes exist in the literature, implementing different low-

power communications protocols such as ZigBee [39], IEEE 802.15.6 [40], 6LoWPAN [41] and many more. Regarding the second category, the efficiency is achieved by reducing the energy spent by the sensors. Since it is common for measured values to change slowly with time, one major class [42, 43] for energy efficient data acquisition aims at reducing the number of acquisitions, especially since some sensors require considerable power to perform their sampling or to convert them using power-hungry A/D converters [44]. Other techniques for energy efficient sampling are also abundant in the literature, such as hierarchical [45] or model-based active sampling [46], each one with its merits and flaws frequently contradicting with performance metrics such as latency and reliability [47].
In the following, we will be dealing with the second category in an attempt to increase the energy efficiency of the system, while keeping the performance at the same levels. We consider that the nature of the existing network is given at most case, i.e. the sensors used are based on commodity hardware and hence the implemented protocol is already imposed.

2.4 IoT Architectural Frameworks

Currently hundreds of IoT platforms are offered in the market, heavily affecting our everyday life. Since we are still at the beginning of the IoT era, neither dejure or defacto standards are applied yet. There are various efforts on new IoT application development using existing or newly introduced protocols for the control of the hardware, for the communication scheme, etc. The idea of controlling and coordinating various IoT elements through an IoT framework comes naturally, allowing a high-level implementation that hides the low-level technical complexity from the developer and the users. However, still the commercial frameworks offered are tightly connected with big market players, e.g. AWS IoT (Amazon), ARM Bed (ARM), Azure IoT Suite (Microsoft), Brillo/Weave (Google), Calvin (Ericsson), HomeKit (Apple), Kura (Eclipse) and SmartThings (Samsung) [48]. Moreover, the existence of numerous frameworks defeats the idea itself of a framework, since still no homogeneity is achieved.
However, IoT systems consist of numerous components of diverse complexity, communicating using different protocols and integrated in different levels of detail. They are built over the chaotic sensor network world. To that end, a number of orchestrators and monitors come into play to add to the number of IoT components. On top of this, services and applications are developed.
Eventually, IoT systems are moving towards becoming systems of systems. This trend is apparent in [49] and [50], where multiple systems and technologies come into play when constructing an IoT System. In [49] we can see a division of the architecture in the following layers from lower to higher levels: Sensing Layer, Network Layer, Service Layer, Interface Layer.
The sensing layer is integrated with existing hardware (RFID, sensors, actuators, etc.) to sense/control the physical world and acquire data. The networking layer provides basic networking support and data transfer over wireless or wired network. The Service layer creates and manages services. It provides services to satisfy user needs. The Interface layer provides interaction methods to users and other applications. Another common trend is providing an interface to communicate with the sensors and embedded devices, over http RESTful services. This is described in [50] where RESTful services are provided for interacting with the sensor networks, and embedded devices. In addition, DPWS [51] is used for network discovery of smart devices and the management of their services.

3 IoT System Architecture

Based on the current trends of IoT Systems, we have designed the multi-layered system architecture of in Fig. 1. The main layers of our architecture and the components in each are the following:

- Sensor networks layer where sensors and as an extension sensor networks reside
- Aggregation Layer where Aggregator Units exist
- Service Routing sublayer where Registry Units exist
- Services and Entities Layer where Service Entities and End User applications are encompassed

This layering has been tailored so as to create two separate worlds. The first world consists of heterogeneous sensor networks. These networks that get formed in this layer, can be either wired or wireless, and can be either sensor nodes, smart devices, embedded devices or any other device that can provide a service and has a way of communicating with an aggregator unit. It is the aggregator units' obligation to be able to cater for as many of sensor networks as possible.
Aggregator units, in turn, have the role of turning the chaotic world of WSNs into a more structured one by providing the whole of the actions that can be performed on the WSNs, over a RESTful API. We name our API as the "Multipurpose Unified JSON API", since by using this API, one can perform any action available in a WSN, regardless of the nature of the action. An action may be a sensing of a environmental variable, an action on an actuator, turning on or off a switch etc.
The third layer of our architecture encompasses all high-level entities that use the services provided by an aggregator unit. These services can be plain services that call services from the aggregators and return the result to a user, or be more complex, e.g. use other services'

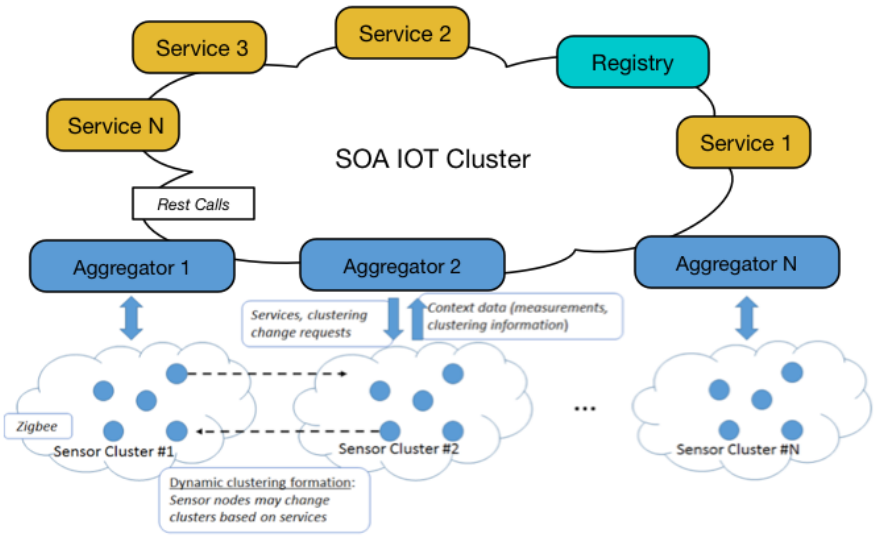

Figure 1: The envisioned architecture's high-level view

results in order to tailor another result, which in turn can be used either as a response to an end user or as input to another service. An important note about this layer, is that it encompasses the end user applications as well.
Between the aggregation and the service layer, there exists a space where we include the Registry Unit. The Registry Unit acts as a bridge between aggregator units and services. It is in essence a service registry where aggregators register their services and service entities can look up any information they need about aggregator services.
It has been deemed necessary [27] for a SoS System, let alone an IoT System, to be able to cater for Multi-criticality. Multi-criticality means that service provision is prioritized based on how important a service call is. Our architecture caters for multi-criticality, by allowing callers of a service to define how important their call is. The aggregator units in turn, take into consideration this extra information and use it to prioritize their service queue.
An important aspect of SoS systems as described in [52] and [53] that we wanted to research, is the aspect of autonomicity. The benefits of autonomicity are numerous, ranging from faster response times and higher QoS to self healing and disaster aversion. For our system to be considered autonomous, a set of self-Configuration properties has to be fulfilled.
A stand alone feature of our architecture, as well as a part of the autonomicity features, is load balancing. Load balancing in our architecture happens in two layers. We balance the load of the aggregator's CPU in its cores, and we dictate the way sensors communicate in our sensor layer (pushing data or pulling data) in order to better distribute load and achieve a level of energy efficiency.
We will now provide a high-level analysis of each component of our envisioned architecture, while more details regarding the sensors and the aggregator will be discussed in section 4.

3.1 Sensors

The sensor components in our architecture are considered to be heterogeneous and communicating with different protocols. As such, the architecture is able to cater for heterogeneous sensor networks, while also providing a seamless experience for the end user. The sensor networks are to communicate with one or multiple aggregator units, in order to report their readings. An important aspect of sensor-aggregator communication is the way the sensors report readings to an aggregator: either by constantly pushing data from the sensor to the aggregator, or by having a call/response relation, where the aggregator requests data from the sensor and the sensor responds.

3.2 Aggregators

The aggregator unit is the bridging component, of our SOA architecture, between the sensor networks (WSN/SN) and the higher-level components of the system. It provides the transparency layer that translates the actions available on the WSN, to RESTful services over HTTP protocol. This layer of transparency is critical as it removes the extra load of forcing users to use specific hardware, software, and communication protocols in order to communicate with the WSN, as this is, entirely handled by the aggregator units. We define an aggregator unit as an autonomic unit that can provide as a bare minimum the following features:

- Ability to communicate with at least one WSN and keep constant track of it
- Register and Update Services to the Registry Unit

- Effectively serve requests, according to its stated policies

In addition to the above, we also propose that an extra set of functions should be available and provided by the aggregator unit. These include, but are not limited to:

- Provision of aggregate functions (e.g. provide the mean temperature from a group of sensors)
- Context Reasoning (e.g. provide services per room/per item/per entity e.t.c.).

3.3 Registry

The registry unit constitutes a simplified DNS-like service as presented in Fig. 2. Any component may poll the registry unit so as to get a list of available components (either aggregators or application/service components). Moreover, it may ask for available services of a specific component. A registry entry consists of: the component description, its IP, and the component's services, followed by a short description of each service. For example, an application component may poll the registry with an aggregator id and a sensor id as HTTP parameters. The registry should let the application component know about the available services the aggregator offers for that specific sensor. Such services might be for instance, a switch toggle, a specific sensor reading or a battery level.
Aggregators are the only components responsible for posting the data to the registry unit via a registration procedure. This procedure is done over HTTP, using valid HTTP verbs. A component is also responsible for renewing its registry records when a service/sensor node is no longer available. The registry unit can opt to delete any component and the services it offers, denoted with the unique ID, assigned to it at the register process, if the component has not renewed its records in a while. The services provided are fully RESTfull and therefore can easily be manipulated and used. Moreover, all reply messages are in JSON format to allow easier manipulation of the data received. For a developer to implement an application

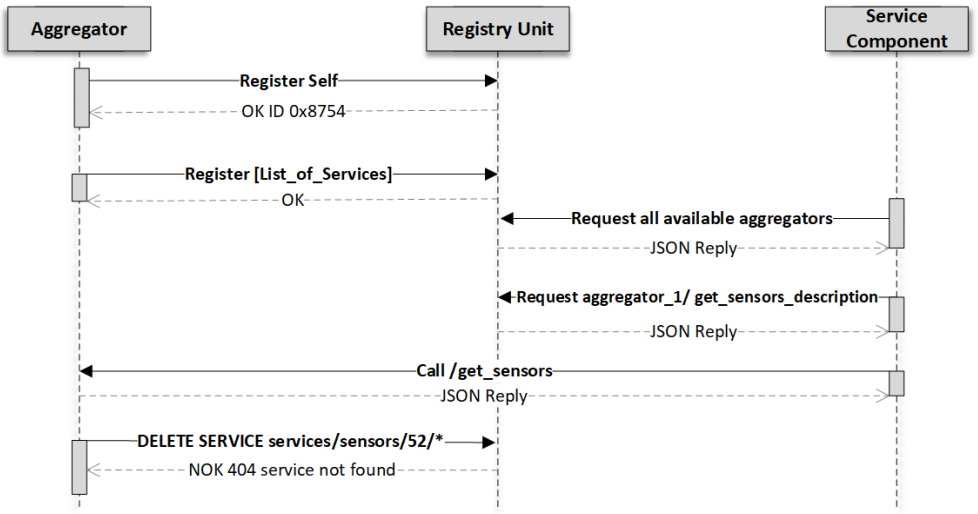

Figure 2: **High-level communication flows between Architecture components and the registry unit**.

using the registry unit, she can simply poll the registry for the services' descriptions and start using them through their endpoint component.
The registry unit acts as discovery service for the users and a registry service for the autonomous components currently available. As such, no information about the underlying WSN and sensor types is maintained in the registry. Aggregators are responsible for the correct documentation of their services and that allows us to consider a layer of abstraction over the WSN.

3.4 Services

Service components provide a service to an end user, either by directly communicating with an aggregator unit, after consulting the registry, or by using multiple existing services to aggregate data that are provided through these services and then compile a new service. As such, services have a dual nature as they can be considered either as end user components or as middleware that provide the foundation for other services. For a user to take advantage of any of the sensors and actuators unified under the aggregator, a specific service must be available. Every service is to be executed outside the aggregator, on an external processor, i.e. a computer or even a smart phone. Obviously, the possible services are numerous and only limited by the available sensors, actuators, and user creativity on how to combine them. A typical service does not have direct communication with the WSN or any subpart of the aggregator, but only with a dedicated component, as it will be explained in the next Section. Hence, the user via a service can utilize any subpart of the aggregator but without directly accessing it.
It is imperative for every service to advertise its existence by enrolling itself to the registry unit, so that the users can locate and use it. Every service contacts the SPU making requests with different levels of criticality, leading to various degrees of load. Thus, the aggregator monitors all the service requests and tries to satisfy all requests while retaining self-configuration and self-adaptation. However, under specific load condition the aggregator may not guarantee the timely execution of every service request. Finally, the communication between a service and the aggregator is achieved via a REST/JSON API.

4 Aggregator

The most important aspect of the aggregator unit is its autonomicity. The aggregator unit is an autonomous entity, in the sense that it is capable of making decisions that improve its performance and functionality. In order to achieve its autonomicity, it takes into account the context it is able to gather for the environment it is operating in, such as available nearby units, sensors' status etc. For the autonomicity to be actually achieved and implemented, we have defined a number of policies to be enforced by the aggregator unit, that can be grouped under two categories Self-Configuration Policies and Self-Adaptation Policies.
Self-X properties are the sum of properties that define the aggregator unit as autonomous. With Self-X properties being the goal, the Self-X policies are the means with which to achieve the goal. Each of the aforementioned set of policies aims at achieving a different aspect of autonomicity, as suggested by the names. We will refer to the set of policies stated to pertain autonomicity as Self-X policies.

4.1 Self-X properties and Self-X policies

Each set of Self-X policies, aims at implementing a Self-X property of the aggregator. The combination of these Self-X properties, give the aggregator unit its autonomicity. In order to be able to properly implement and enforce the policies, the aggregator unit needs to manage a number of resources. The main resources the aggregator manages are

- The aggregator's energy consumption and/or battery power
- The sensors' energy consumption and/or battery power
- The aggregator unit's CPU cores
- The sensors under the aggregator's influence themselves (i.e. accept or remove sensor from the aggregator's influence)

In addition, in order to be able to manage these resources, the aggregator unit must be able to self-monitor itself. To that end, a number of variables is monitored by the aggregator unit constantly, named "Monitored Variables". In the following, each Self-X property will be presented and establish why it is important for the autonomicity of the aggregator unit.

4.1.1 Self-Configuration

According to [54] self-configuration is defined as a system whose "components should either configure themselves such that they satisfy the specification or capable of reporting that they cannot". To achieve a level of self-configurability, the aggregator unit monitors a number of variables, relating to itself, its components, and the sensors it is managing at the time. The aggregator unit should be able to self-monitor itself and reconfigure its hardware, its software, and the way it operates on the fly. Described in the following are three self-configurability policies we propose and define:
The first policy is the **Criticality Level Management policy**. This policy defines priorities for all received service requests from users, by using "Criticality levels". The requests are then to be prioritized according to their criticality level and therefore, it is ensured that requests with a higher-level of criticality are to be serviced with a higher priority compared to requests with lower priority. Based on the work of [55], one way to tackle the aforementioned quantization is to define discrete levels of criticality. The authors of [55] define 5 levels of criticality in their work, Level-A to Level-E, with E being the lowest.
For our policy we will be quantizing the criticality of the requests in six levels level 1 to level 6, with level 1 being the lowest. In accordance with [55], we use the five proposed criticality levels, and add an extra criticality level of highest importance, that is to be used only by the aggregator units, to communicate emergency messages.
By applying the same logic explained in [55], six distinct lists were implemented that map to the six different criticality levels. At the event of the aggregator receiving a service request, the latter is sorted and added to the appropriate criticality level list. For each list a different scheduling algorithm is implemented for managing the requests in it: for levels 6 and 5 the partitioned preemptive EDF is proposed, for levels 4 and 3 the global preemptive EDF, and finally global best-effort for levels 2 and 1. The lists are checked for tasks from highest to lowest criticality and tasks are processed according to the respective scheduling algorithm. We name this algorithm for managing criticality levels as CAFIFO, or criticality aware FIFO.
The second Self-Configuration Policy regards the "**Resource Management**". The policy

aims at minimizing the energy footprint of the aggregator units. To that end, the aggregator unit constantly monitors the Monitored Variables. The main resource the aggregator manages by enforcing this policy is the per CPU core load. As it is apparent, one of the Monitored Variables is the per CPU core load. The policy is based on the premise that each request to be serviced, is serviced in a separate thread, and that each thread spawned to service a request is of similar complexity and thus stresses the CPU core in a linear manner. If a core's load is above a specific threshold, then it is regarded as overloaded. If, on the contrary, it is below a specific threshold then it is considered under-utilized. Based on the above, the aggregator unit takes the following actions:

- If a core is underutilized, then it should be shut down to preserve energy. Note that it is shut down when its last remaining threads have finished executing.
- If a core is overloaded, then any incoming threads will be dispatched to another core of the CPU. If no other core is running, then if possible turn one on.
- The next thread to be assigned to a core, is to be assigned to the core with the least processing load on execution time.
- If all cores are overloaded, the aggregator enters "overloaded mode". The aggregator exits "overloaded mode" when at least one core's load drops below the overloaded threshold.

The third policy of the Self-configuration Policies, is the "**Aggregator Service QoS Preservation**". This policy aims at maintaining the quality of service (QoS) provided by the aggregator units across our architecture. The policy is activated, and thus enforced, when the unit is in overloaded mode. Then the aggregator unit is under great stress and the provided QoS starts to decline. To prevent this, the aggregator can decide to ask nearby aggregator units, if they can take over a sensor that was initially under the original aggregator's supervision.
To decide on which sensor is to be "migrated" over to a neighboring aggregator unit, a simple metric is used. The metric takes into consideration the highest criticality level a sensor was called to service recently and the amount of calls it received in that time. The amount of time that this metric uses is the period the aggregator unit monitors its variables, which is called a "monitoring quantum". Monitoring quantum will be discussed in detail in the following section.
After a sensor unit to be migrated is found, then the aggregator unit communicates with neighboring aggregator units, using the criticality level 6, to inquire if they are able to handle the sensor unit it decided to migrate. The neighboring aggregator units are then to assess their load levels and decide upon whether they can handle accepting the sensor in question or not. After a decision has been made, the aggregator units are to reply to the originator aggregator unit. Should a suitable candidate be found, the aggregator unit will inform the sensor of the change, if it needs to, and pass the needed information to the other aggregator unit. If a candidate is not found, then the aggregator tries again with another sensor unit, until it exhausts its sensor list. This will continue until the aggregator unit exits the "overloaded mode", at which point it will continue its normal operation.

4.1.2 Self-Adaptation

According to [56], self-adaptive software is software that "evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the

software is intended to do, or when better functionality or performance is possible". As defined in [57], the self-adaptation of a system has to do with the system's ability to accommodate resource variability, changing user needs, and system faults. Our proposed system is self-adaptable in the sense that it can change the way it communicates with the sensor units depending on the current needs.

To meet the goal of self-adaptation we propose one policy, the **"Sensors' Communication Protocol"**. Sensor modules usually support data collection through push and pull methods, having the active method determined by users of the WSN. Push method means that the sensor module constantly pushes data to the WSN, and the aggregator unit collects the data as they are emitted. On the contrary, a pull method of communication means that the sensor module only emits data when polled to act so. Each method has its advantages and drawbacks, with push method being more energy consuming but providing fresh data constantly, and push vice versa.

The aggregator unit is able to command a sensor module to change its communication method based on the current context. On the event of a request with criticality level 5, the aggregator will command the sensor involved to switch to push mode. Since push mode is energy costly we have implemented a leasing scheme for how long a sensor will stay in push mode. When a sensor changes to push method, the sensor is asked to use the push method of communications for a certain time period, up to a minute. After that time period, the sensor reverts back to pull method of communications. If a request is received with a criticality level 5 for the same sensor within the mentioned time period, then the lease is renewed for double the previous time span, up to a minute.

The pull method is the default option for our WSN and is utilized by medium and low criticality requests. Requests of criticality Levels 1 to 4 will be serviced by using a pull method of communications. In essence the aggregator unit processes a request, it communicates with the sensor responsible for providing the data needed and then the sensor responds with the data. Although this is more energy efficient, it adds some delay overhead, as the user has to wait for the sensor to respond in order to get the fresh data.

Finally, another available option for requests of Criticality Levels 1 and 2 is a caching method. Caching can be used to tackle the communication overhead of the pull method for low criticality requests. The aggregator is able to cache data of every reading and timestamp them. Whenever a sensor sends data to the aggregator, retrieved either by pull or push methods, these data are stored in the shared memory module of the aggregator. On the event of receiving a request of Criticality level 1 or level 2, and the data requested have already been retrieved within a certain time span, then the cached data will be reported to the user. By employing caching techniques, we achieve greater energy preservation both on the side of the sensor modules and the aggregator unit while simultaneously maintaining a healthy level of QoS.

4.2 Architectural Implementation

The software architecture used for the implementation of the aggregator unit, is comprised of a number of modules, each of which provides a different utility to the aggregator. Not all of the modules are essential for the aggregator's basic operations. The extra modules contribute to the self-management, the autonomicity, and the self-monitoring of the unit. The modules function independently and communicate one another employing a common memory space. The software modules that provide the minimum needed functionality are:

- Control Unit (MCU)

- Service Provision Unit (SPU)
- Sensors Communication Unit (SCU)
- Shared Memory Unit (SHM)

On top of the above, there exists a software entity that is not literally a module but is essential for the operation of the unit: the Request Execution Threads (RET). The units that cater for the autonomicity of the unit are:

- Monitoring Unit (MON)
- Decision Making Unit (DMU)

All modules are depicted in Fig. 3, where a high-level representation is given, as well as the way they communicate. As it is easily seen from Figure 3, all modules communicate through the SHM; the direction of each arrow shows if the communication between the SHM and a given module is one or both ways.
Moreover, a conceptual structure of the modules and the communication flows among them can be seen in the class diagram of Fig. 4. The idea behind the modularisation of the software of the aggregator unit originates on our premise to create an aggregator unit that would easily adapt to any WSN with minimal changes. Given our proposed architecture, the only sensor dependent module is the Sensors Communication Unit. In the following paragraphs we will provide descriptions of the functionalities provided by each module, starting with the modules needed for the basic functionality, and then carrying on to the more complex modules catering for the autonomicity of the aggregator.

Control Unit (MCU)

The MCU is the unit that initializes and brings up the rest of the software modules and generally controls aspects of the aggregator unit, e.g. turn on/off cores etc. An added responsibility of the Control Unit is to keep the data fresh regarding the aggregator unit on the

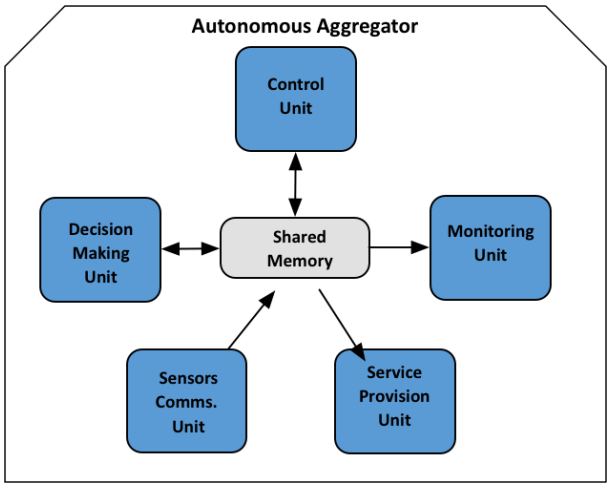

Figure 3: **High-level representation of the proposed implementation**

registry. This entails registering, updating, and deleting services available as well as update any relevant information regarding provided services on the registry. The Control Unit spawns Request Execution Threads which implement the asynchronous communication between the Control Unit and the Sensors Communication Unit. Finally, the Control Unit is responsible for queueing requests received from the SPU in their respective Request Queue according to their Criticality Level.

Service Provision Unit (SPU)

The SPU provides a public interface to be used by users. This public interface, named Multipurpose Unified JSON API, is common across all aggregator units, and should be enforced, in order to provide a single method of communication in our architectural system. By using the API, a user is able to request a service from an aggregator unit. The service provided requests by the API can be categorized in three groups:

- Sensing: Sensing requests are requests that ask for an action to be performed on the WSN. This action can be either a sensing request, as in to ask a sensor to sense an environmental variable, or for an actuator to be engaged etc. The aggregator should handle the overhead of simplifying complex requests that ask for a sensing request of a conceptual area. Grouping areas include rooms, floors, buildings, and so on. On the event of a complex request an aggregator may need to contact a number of sensors instead of simply contacting one.
- Description: Description Requests provide the user with a description of a available service. The answer is in accordance to the API and can be used by other autonomous entities to further improve their provided services, or from end user devices to provide a human readable description of the service.
- Entity: Entity requests provide users with information about the existence, or lack of sensors and smart devices under the aggregator unit's influence area.

All requests received by the SPU have a Criticality Level, defined at request time by the user. After a request has been received and processed, it is forwarded to the Control Unit to be sorted and handled accordingly. When a request has been properly handled and an answer has been provided, then the SPU handles the communication with the user to provide a response.

Sensors Communication Unit (SCU)

The SCU is the only software module that is sensor dependent. Its implementation varies, depending on the way the WSN communicates, but the interface provided to the rest of the modules must be common, regardless the underlying implementation. The Unit must be able to report any incoming messages to the Control Unit, as well as push any outgoing messages towards the WSN. The messages' nature and variety is defined by the WSN. The unit is not constantly executed in an attempt to preserve energy; it remains dormant until an action is required on its side.

Request Execution Thread (RET)

The RET is not a module but rather a formal representation of the software structure handling the asynchronous communication between the Control Unit and the Sensors Communication

Unit. RETs are spawned by the Control Unit whenever a request is received from the SPU, and its existence symbolizes the existence of an outstanding, and yet to be serviced, request from a user. When a RET is spawned, it is not immediately executed, rather queued, as described in section 4.1.1, according to the criticality level of the request. When the Request Execution Thread is to be executed, it is then removed from the queue. On execution the Request Execution Thread will check the nature of the request and act accordingly, by communicating with the Sensors Communication Unit, or retrieving data from the Shared Memory Unit. When the Request Execution Thread has collected all the necessary data it will contact the SPU with the response to be relayed to the initiator of the request.

Shared Memory Unit (SHM)

The SHM acts as a common information repository for all modules. Different modules are able to store information in Key Value pairs, which remain public within the aggregator unit, and is available for every other module to access and modify. It is possible to store the policies that the unit has to enforce in the SHM during startup of the system, and therefore modify the said policies on runtime.
The SHM plays an integral role in enabling self-management for the aggregator unit, as the Monitoring Unit and the Decision Making unit, which will be discussed in the following paragraphs, both use the SHM in order to monitor the status of the system, as well as monitor and analyze historic data that may be needed in order to implement policies. Since the nature of the system is multicore and multithreaded, it is apparent that the control over the access and modification of the information in the SHM is important. In systems as the one proposed, race conditions and deadlocks are a major concern. To tackle this issue and retain the integrity of the information, a semaphore system is proposed with a common FIFO queueing mechanism for each action to be performed on the data. Further research on the way the queue is to be handle could be done, implementing a CAFIFO-like algorithm where memory access requests are sorted based on the importance of the request. The scope of the research should include performance aspects and system stability among others.

Monitoring Unit (MON)

The MON monitors the overall status of the aggregator, and creates metrics that help with the management of the unit. These metrics are taken into consideration by the Decision Making Unit in order to conduct its operations. This module measures available resources and a number of other system variables of interest, as per policies' directives. The resulting data are stored in the Shared Memory Unit. The Monitoring Unit conducts its monitoring on a preset time quantum, the Monitoring Quantum, which we propose as 1 second. The proposed monitored variables,as derived by the set of policies we enstate, are shown in Table 1.
As Monitored Variables are closely related to the enstated policies asked to be implemented by the aggregator unit, they are subject to change, with policy changes. Adding to the monitoring capabilities, the monitoring unit is responsible for invoking the Decision Making Unit, on severe unit status changes defined by enstated policies.. The thresholds for when the Decision Making Unit should be invoked is defined by the System Policies.

Decision Making Unit (DMU)

The DMU is an advisory unit that given a system status, will advise on actions that should be taken. It must be emphasized that the actions themselves are carried out by the Control

Unit, the DMU only advises on which actions to perform. The actions to be performed, are dictated by the policies that the aggregator unit follows. As previously hinted, the Unit is not constantly running, but rather invoked by the Monitoring Unit when needed. After taking into consideration the system status, a set of actions is to be given as feedback to the Control Unit, the advice is packed in a single, JAVA in our implementation, object and given to the Control Unit to use. In order to decide on a course of action, the Unit has its policies implemented in CEP form.

Example of policy invocation

In order to better understand how the self-management enabling units work, we will now describe how an aggregator unit implements the resource management policy we have described in section 4.1.1 by focusing on the actions taken by the MON the SHM and the DMU. The resource management policy, is the policy that is responsible for turning on and off CPU cores, according to the system load. If a core's load is above a specific threshold, then it is regarded as overloaded. If, on the contrary, is below a specific threshold then it is considered under-utilized. If a core is underutilized, then it is shut down to preserve energy. On the other hand, if a core is overloaded, then any incoming threads will be set to be executed on another core of the CPU. If no other core is running, then if possible, one is turned on. If all cores are overloaded, the aggregator enters "overloaded mode". The aggregator exits "overloaded mode" when at least one core's load drops below the overloaded core level threshold.
In each monitoring quantum the MON monitors a number of monitoring variables. We have proposed a monitoring quantum of 1 second, although this is up to the implementer's choice. In this example we will examine the actions taken by the aggregator's units when a core is overloaded. By monitoring these variables, the MON is able to render a core as Overloaded. If during a monitoring check a core is found to be overloaded, then the DMU is awaken given as information that it was awaken because core X is overloaded. After that, the monitoring unit continues its operations normally, and will recheck the system's status on the next monitoring quantum.
The variables that are monitored as well as the limits that will be checked against are listed in Table 2. The limits and values stated in the table have been deduced after executing stress

Current System Load
Current Running Cores
Current Overloaded Cores
Current Available Cores
Number of requests per minute
Available hardware
Available hardware's status (i.e. Battery levels)
Available sensors (including all directly deduced information i.e. communication method)
Available services
Sensor's health and sanity (battery levels, validity of data produced)
Neighboring Aggregator Units
Overloaded Mode
Under-Utilized level
Overloaded Level

Table 1 Proposed monitored variables as derived by the set of policies we enstate

Variable name	Value
Current System Load [CoreId,Load]	[(1,85%),(2,90%),(3,0%),(4,0%)]
Current Running Cores	2
Current Overloaded Cores	2
Current Available Cores	4
Overloaded Mode	false
Under-Utilized level	7%
Overloaded Level	80%

Table 2 Monitored Variables and limits established from the aggregator stress tests.

tests on the configuration discussed in section 5.
When the DMU receives the message from MON, it will try to enforce the resource management policy. Internally, the policies are written in CEP Syntax. The resource management policy, for example, in CEP is the following:

```
for(x) x.getLoad>OverLoadLevel :- controlUnit.setAsOverloaded(x)
for(x) x.getLoad<UnderUtilizedLevel :- controlUnit.setAsUnderUtilized(x)
for(x) (x.isUnderUtilized) and (x.RunningThreads = 0) :-
controlUnit.shutdown(x)
(OverloadedCores = RunningCores) and (RunningCores !=
AvailableCores) :- controlUnit.activateNewCore
(OverloadedCores = RunningCores) and (RunningCores =
AvailableCores) :- controlUnit.enterOverloadedMode
(Overloaded = true) and (OverloadedCores != RunningCores) :-
controlUnit.enterOverloadedMode
```

After the DMU is done checking the system status, it regards that there are 2 overloaded cores, and 2 idle cores. Given this scenario, the DMU will decide that another core should be commissioned and the load between the cores should be rebalanced.information is then forwarded to the MCU. After this course of action, the DMU returns to its hibernation.
The MCU receives the advice from the DMU and performs the actions described. It then continues its operations as normally.

5 The proposed Implementation

The proposed autonomous aggregator architecture was adopted to implement the aggregators integrated within SOA infrastructure for smart building applications, constructed in the framework of EMC^2 ARTEMIS Joint project. For a thorough presentation, the reader is referred to [15].
In a nutshell, two discrete sensor technologies using different communication protocols were integrated within Smart Building SOA architecture. The different families provide both similar and unique functionality and were set up to monitor a single room. To seamlessly integrate both technologies within Smart Building infrastructure, two aggregator units were deployed on different hardware (a Raspberry Pi 2 and a Raspberry Pi 3). Both units run the same distribution of GNU/Linux Raspbian Stretch. The aggregator software was developed in JAVA and therefore, the only prerequist for porting our implementation on a different platform than that of Raspberry Pi, is the existence of a Java Virtual Machine (JVM) implementation. Hence, our proposed implementation is hardware independent and can be executed without any alterations on virtually any hardware having a JVM and a

network adapter.
The aggregator software acts as a middleware, hiding from the services all sensor related features, such as communication protocols, and support a common REST API over HTTP protocol for all sensors. The same aggregator software is running on both Raspberry devices, independently of the sensors controlled by it since the sensor communication unit integrated the same libraries for both sensor families, thus any sensor regardless its family, may be controlled by both aggregators in real-time. The requests each aggregator may serve, utilizing its sensors, are registered to the Registry Unit, which in turn, is used by IoT Services running in the Registry's zone to find the proper aggregator to execute the service request. The Registry Unit was deployed on the Raspberry Pi 2 and communicated over Ethernet with the rest of the network. In this paper we focused mainly on the implementation of the aggregator module and its autonomy features, for more details on the provided aggregator API and the implementation of the Registry Unit see [14] and [15].

5.1 Services for the Smart Building Demonstrator

In order to highlight the potential offered by the presented architecture, a couple of Smart Building services were implemented. Real-life services and at the same time easy to walk through, in this attempt to demonstrate them. For each of the presented services a simple UI was implemented, which displays all the actions taken from the service side and allows their logging.
The services presented in-depth in the following are:

- S1 - Temperature control: the temperature at a specific location is set to a designated value
- S2 - Fire watchdog: a constant check for fire in a specific location
- S3 - Person's whereabouts: a specific person is located based on the unique Bluetooth Tag of a device she is carrying

S1 - Temperature control

This rather straightforward service offers the ability to control the temperature of a specific location during the winter. The user can set a desired temperature and the client will ensure it will be applied, with the aid of one or more temperature sensors and one or more actuators that control heating units at the premises; obviously if the heating units are replaced with cooling ones, a similar service can be used during the summer. The specific service is not assigned high criticality, since the changes in temperature do not take place suddenly but instead gradually. Moreover, the late response in a temperature change does not lead to irreversible effects; hence S1 invokes the aggregator applying criticality of level 2. In this scenario, the current temperature is displayed on the UI and the user sets a higher desired temperature. This procedure - from the client point of view - follows the steps described below:

1. Request all available services using *GetAvailableServices()* and verify that the temperature measurement and control of the specific location is indeed among the provided services
2. Request the current temperature from the SPU with criticality level 2, via the **GetTemp()** method

(a) If the temperature is below the user defined threshold, connect to the SPU and send the request for turning on the heating units

(b) Else, request from the SPU to turn off the heating units.

It is noted that for efficiency reasons, there is a minimum threshold of at least 1 degree Celsius in the difference between the desired and the actual temperature, for the service to start or stop the operation of the heating units.
From the aggregator point of view, the steps followed for the procedure are:

1. In response to the client *GetAvailableServices()* request, the SPU connects to the SHM and using the *Get()* method collects all the offered services and returns them to the client, via the *SetResponse()* method
2. In response to the client request for the current temperature, the SPU follows a series of steps in order to return to the client the mean temperature of the specific location:

   (a) The SPU forwards the request to the CU, which in turn communicates with the DMU, asking advice on how to handle the request

   (b) Based on the DMU response, the CU will start a Request Execution Thread which will communicate with the SCU and eventually with the WSN

   (c) The SCU unit forwards the WSN response to the Request executing thread, which in turn returns the response to the CU

   (d) Finally, the CU renews any Shared Memory variables needed, then wraps the response as needed and forwards it to the SPU

3. The client request to turn on/off the heating units will be forwarded to the CU, that will start a new thread which will order (via the SCU) the actuators to turn on/off the heating units

S2 - Fire watchdog

The specific service S2 constantly searches for indications of fire, by measuring CO_2 concentration, temperature levels and ambient light levels in a specific location. The rational is that if the CO_2 levels are above a specific threshold and at the same time the temperature is abnormally high and the light levels are also high then it is safe to assume that a fire has started.
The specific service operates in three distinct criticality levels, between a medium (3) and a high (5) criticality level, since it is crucial for the watchdog not to be delayed, especially when there are indications of fire. When no sign of any irregularity is read, the service runs at criticality level 3, monitoring the room temperature. If the temperature rises above a preset threshold, the service increases its criticality level to 4 and starts monitoring CO_2 levels as well. If the CO_2 levels also exceed a preset threshold, then once more the service increases its criticality level to 5 and starts to also read the ambient light levels. Should the latter also rise above a preset threshold, the fire alarm is sounded to warn the user and the sprinklers are activated. In case any of the measured values fall below the designated threshold, then the service returns to the initial criticality level of 3.
Similar to service S1, the calls from the client point of view are serviced as follows:

1. Request all available services using *GetAvailableServices()* and verify that the temperature, CO_2, and ambient light level measurements as well as the control of the fire alarm and the sprinklers are indeed among the provided services

2. Request the max temperature using the **GetTemp()** method and criticality level 3

   (a) If the temperature is below the user defined threshold, go to step 3 and request the max temperature once more

   (b) Else, request the CO_2 levels using criticality level 4

      i. If the CO_2 levels are below the user defined threshold, go to step 3 and request the max temperature once more

      ii. Else, request the ambient light levels using criticality level 5

         i. If the ambient light levels are below the user defined threshold, go to step 3 and request the max temperature once more

         ii. Else, sound the alarm and activate the sprinklers using criticality level 5

S3 - Person's whereabouts

The S3 service tries to locate a person's whereabouts, through the usage of the Bluetooth sensors, by locating the Bluetooth Tag (e.g. cellphone's Bluetooth) of a specific device paired with that person. The Bluetooth sensors can constantly scan their vicinity for available Bluetooth devices, which are distinguished based on their Bluetooth Address (BD_ADDR), similarly to MAC address in computer networks. The service can operate in two modes:

- mode 1, where the Bluetooth Tag is sought just once
- mode 2, where the Bluetooth Tag is sought constantly and once found, the S3 user is informed

The specific service operates in two distinct criticality levels, depending on the operation mode. In mode 1 criticality level 4 is used since the user in question may be on the move and hence, it is wanted to be located as fast as possible in order to avoid the undesirable situation to read multiple times the same Bluetooth Tag by different sensors along its route. Contrary, in mode 2 criticality level 2 is used in the rationale that it is more efficient once the Bluetooth Tag is located, then the user to re-initiate S3 at operating mode 1 and actively locate the Bluetooth Tag. For the needs of the specific service, running at mode 1, the client utilizes the following calls:

1. Request all available services using *GetAvailableServices()* and Verify that the Bluetooth sensors readings are indeed among the provided services
2. Request from the SPU the Bluetooth readings using the **GetBTInfo()** method and criticality level 2

6 Experimental Results

In order to assess the capabilities of our system and highlight its efficiency, a case study using all three presented services was tested, evaluating both the system's performance and energy efficiency. Various experiments were executed on the demonstrator hardware, with different loads, criticalities and number of parallel services executed, in order to test the proposed system. The results of a typical execution that thoroughly depicts the system's behavior are presented in the following subsections.

6.1 Resource Management

The first policy presented is related to the switching on and off of CPU cores, as defined by the Resource Management policy, in regards to the system's load. The policy requires for preset levels of load in order to set thresholds for states. In order to measure the max load per core, we conducted stress tests on our hardware, and defined that each core was capable of running 40 threads concurrently at max. With this metric defined as the 100% of load a core can achieve, we defined the Under-Utilized level at 3 threads and the Overloaded level at 32 threads. In Fig. 6 the relation between the percentage of energy saved on the Raspberry Pi and the active cores is depicted. As expected, the more cores that are active, the more power is consumed and hence once all cores are active no energy saving is achieved. The maximum achieved savings were 75% of the Raspberry's CPU power consumption.
In Fig. 7, the relation between the response times and the performance is shown, which follows an interesting pattern. On the top chart, the solid gray line represents the average high criticality request response time per interval and the solid black line the average per-interval response time for all request. The dashed lines represent the total average response time for high-criticality requests in black color as well as the average response time for all requests in gray color. On the bottom figure, the number of active cores along with the number of overloaded cores is shown. While the system remains in normal status, i.e. not overloaded, response times remain mostly constant, and around a similar time span. It is only when the system becomes overloaded that it becomes unstable and response times start rising fast. Moreover, a spike exits in the requests' response time after the overload of the system. That is due to leftover low-level requests, that have arrived before the overload of the system, and could not be handled until after the system was back to normal status.

6.2 Sensor communication protocol

The second presented policy, regards the way the sensors communicate. The policy chooses between a push or pull communication model. The results of this policy are presented in Fig. 8. In the top chart, the pushing or pulling mode of the sensor is depicted by the background of the chart; the dark background indicates pushing while the lighter one pulling. Additionally, the average response time for high criticality requests as well as the per-interval response time for high criticality requests is shown. On the bottom chart, the number of active cores along with the number of overloaded cores is presented. When the sensor turns to pushing mode, the average response time drops significantly, and only increases as the system overloads and becomes unable to handle the load. It is important to note that pushing is costlier than pulling data, i.e. while the sensor is pushing data it consumes on average 75-100% more power.

6.3 Criticality Level Management

The third presented policy regards Criticality Level Management. In this policy, the aggregator responds to low criticality requests with cached responses. In Fig. 9 it is depicted that under normal circumstances cached responses comprise more than 50% of the communication while, when the system starts becoming overloaded, cached responses drop to about 40% of total communications. Hence, the relation between cached responses and the average response time for all requests is shown. In Fig. 10 the number of completed requests is presented in relation to the number of completed requests from cache, per interval, and the per-interval savings of the WSN in percentages. It is noted that sensor power savings

from cached responses reached a peak of more than 400% and averaged at about 71% per interval.

7 Conclusions

A System of Systems (SoS) architecture towards the Internet of Things exploiting the autonomous operation of aggregator devices was presented in the paper. The heart of the system, i.e. the aggregator, offers transparency, by hiding sensor-specific details from the IoT application users and thus allowing the successful interconnection of heterogeneous IoT devices. The aggregator is autonomous in the sense that self-configuration and self-optimization features are supported, while aggregation software may run on commodity multi-core devices, such as the Raspberry Pis.

A smart building system was built using the proposed architecture as a case study. The analysis of the use case provided solid evidence that such an architecture is realistic and can lead to highly competitive systems. Indeed, the system provides the desired performance, while reducing the overall system's energy consumption, both in terms of the existing WSN and the aggregator component itself. To achieve these goals, the aggregator is integrated with SOA, makes efficient usage of IoT multi-core systems, and applies multi-critical policies. By ranking the offered services, according to their criticality, the aggregator controls the sensors behavior, in an attempt to reduce the overall energy consumption. Moreover, the power consumption of the aggregator itself is controlled via using the optimal number of active computer cores, using load balancing techniques and keeping the active cores as loaded as possible, while suspending needless ones.

Our next endeavor will be the extension of the system, allowing different aggregators to share even more data, such as their policies and experiences without enforcing any centralized control in decision making. Moreover, the usage of context aware algorithms such as CAFIFO will be explored, for tasks such as accessing the Shared Memory Module. Undoubtedly, larger scale scenarios will be sought from various areas - not exclusively from smart building system.

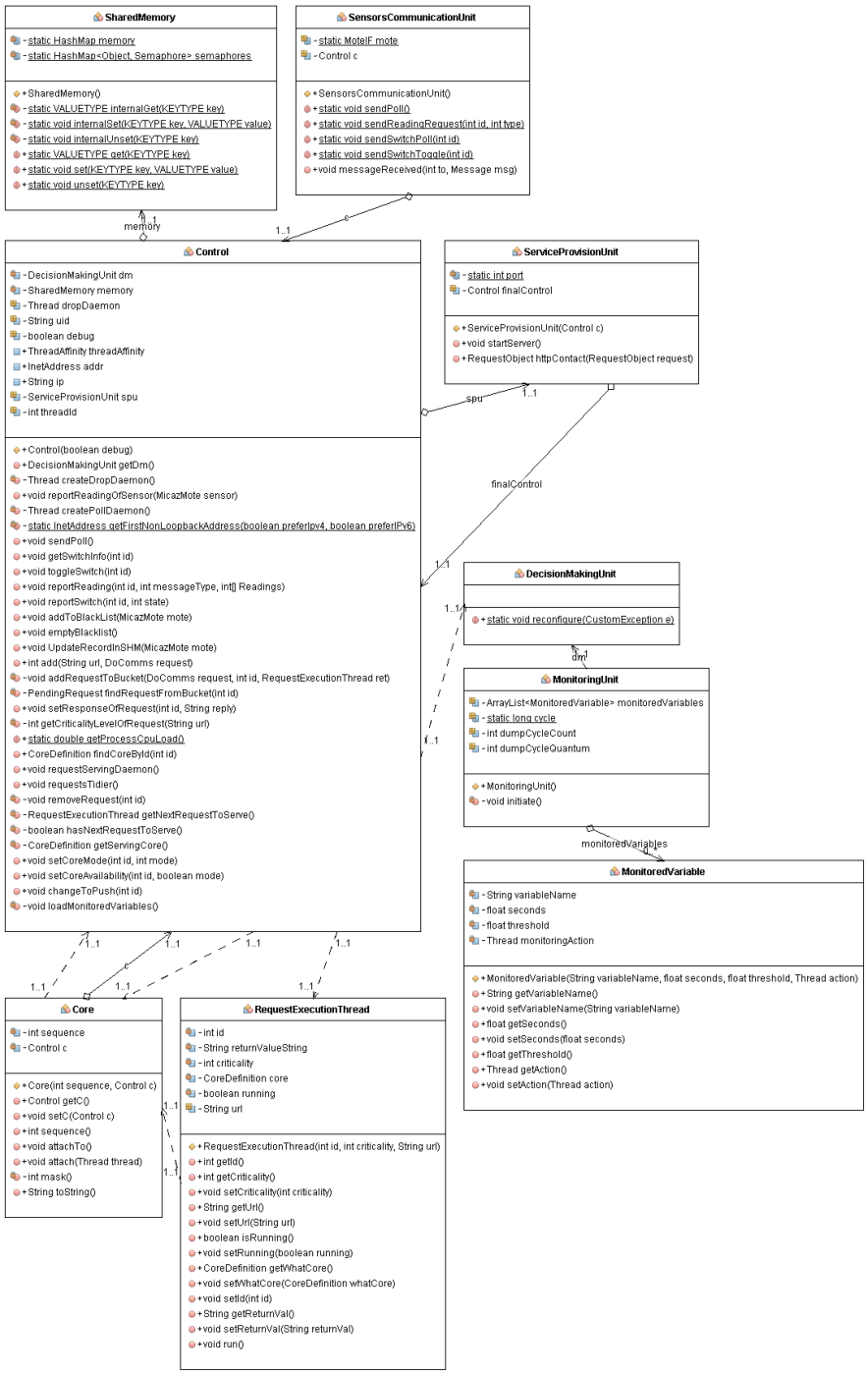

Figure 4: **Class Diagram depicting the communication flows between the Aggregator's modules**

Figure 5: Screenshot of the UI used for the fire watchdog. In the specific service, no user input is needed and only debug messages regarding the service execution and sensors levels are outputted on the UI

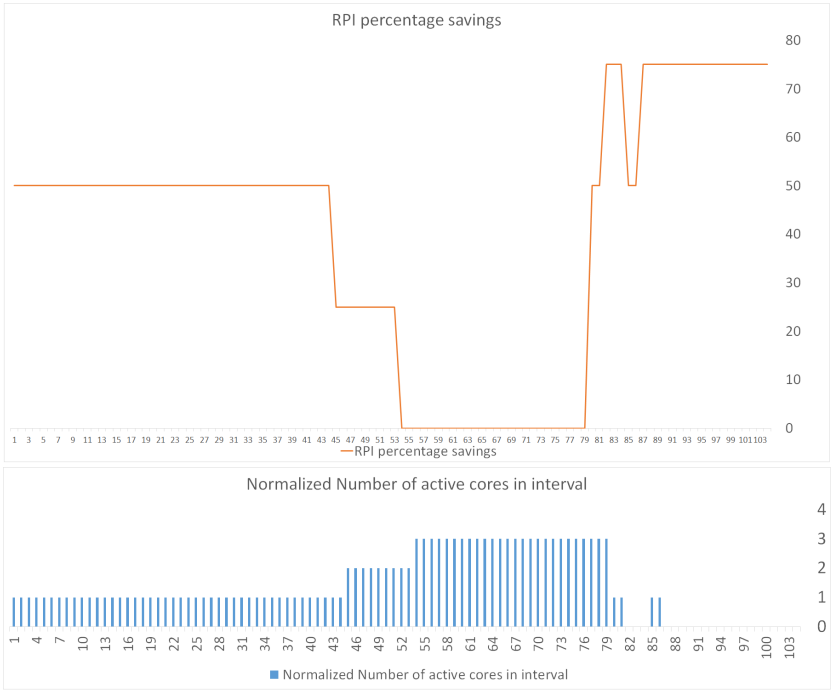

Figure 6: **Relation between the percentage of energy saved on the CPU, and the currently running cores**.

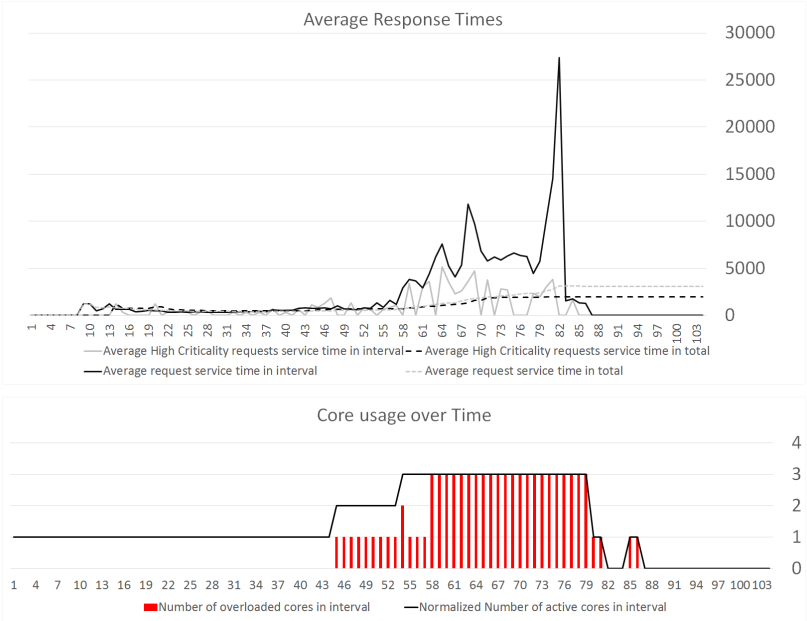

Figure 7: **Relation between the percentage of energy saved on the CPU, and the currently running cores**.

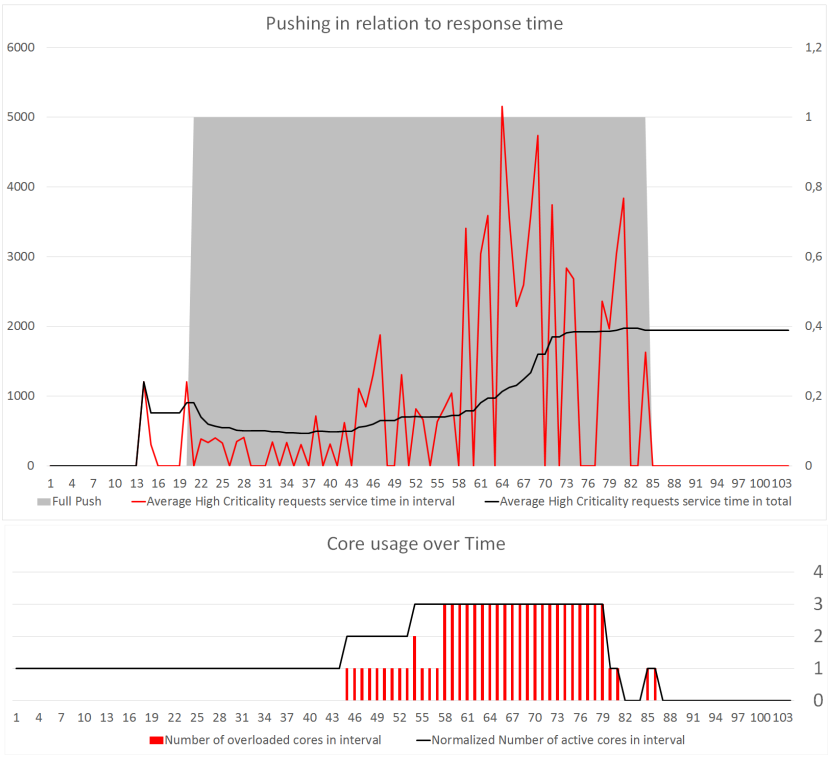

Figure 8: **Relation between the sensor protocol, the high criticality response times and the system's load status** .

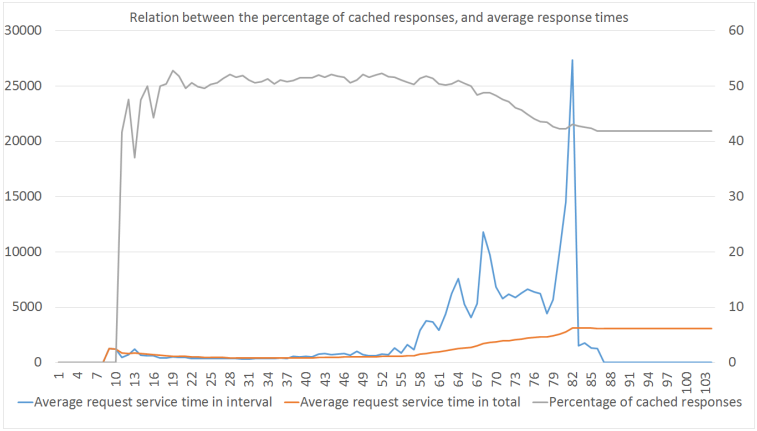

Figure 9: **Relation between the percentage of cached responses, and average response times**.

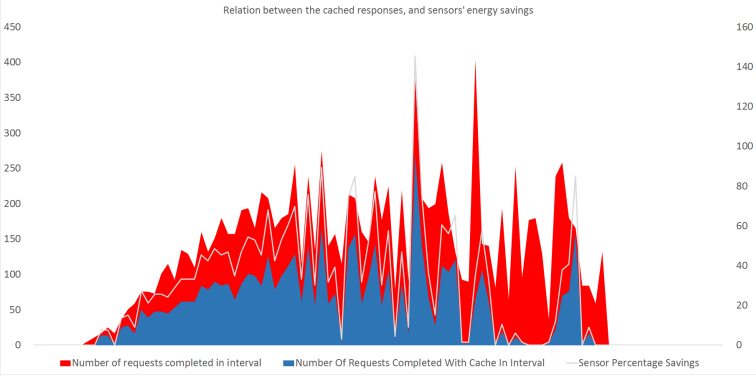

Figure 10: **Relation between the cached responses, and sensors' energy savings**.

Abbreviations

BMS	Building Management System
DMU	Decision Making Unit
IoT	Internet of Things
ITU	International Telecommunication Union
JVM	Java Virtual Machine MCU
Control Unit	
MON	Monitoring Unit
QoS	Quality of Service
RET	Request Execution Thread
SCU	Sensors Communication Unit
SHM	Shared Memory Unit
SOA	Service Oriented Architecture
SoS	System of Systems
SPU	Service Provision Unit
WSN	Wireless Sensor Network

List of abbreviations used in the text

References

[1] Sara Amendola, Rossella Lodato, Sabina Manzari, Cecilia Occhiuzzi, and Gaetano Marrocco. RFID technology for IoT-based personal healthcare in smart spaces. *IEEE Internet of things journal*, 1(2):144–152, 2014.

[2] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[3] ChuanTao Yin, Zhang Xiong, Hui Chen, JingYuan Wang, Daven Cooper, and Bertrand David. A literature survey on smart cities. *Science China Information Sciences*, 58(10):1–18, Oct 2015.

[4] Giancarlo Fortino, Antonio Guerrieri, Gregory MP 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):1934–1952, 2012.

[5] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, oct 2010.

[6] A. Ahmed and E. Ahmed. A survey on mobile edge computing. In *2016 10^{th} Int. Conf. on Intelligent Systems and Control (ISCO)*, pages 1–8, Jan 2016.

[7] M. Chiang and T. Zhang. Fog and IoT: An overview of research opportunities. *IEEE Internet of Things Journal*, 3(6):854–864, Dec 2016.

[8] Anind K Dey, Gregory D Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-computer interaction*, 16(2):97–166, 2001.

[9] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. Context aware computing for the internet of things: A survey. *IEEE Communications Surveys Tutorials*, 16(1):414–454, First 2014.

[10] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

[11] Vijay Raghunathan, Curt Schurgers, Sung Park, and Mani B Srivastava. Energy-aware wireless microsensor networks. *IEEE Signal processing magazine*, 19(2):40–50, 2002.

[12] Gregory J Pottie and William J Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, 2000.

[13] Huseyin Ugur Yildiz, Kemal Bicakci, Bulent Tavli, Hakan Gultekin, and Davut Incebacak. Maximizing WSN lifetime by communication/computation energy optimization of non-repudiation security service: Node level versus network level strategies. *Ad Hoc Networks*, 37:301–323, 2016.

[14] George Bravos, V Nikolopoulos, Mara Nikolaidou, A Dimopoulos, Dimosthenis Anagnostopoulos, and George Dimitrakopoulos. An autonomic management framework for multi-criticality smart building applications. In *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*, pages 1378–1385. IEEE, 2015.

[15] Alexandros C Dimopoulos, George Bravos, George Dimitrakopoulos, Mara Nikolaidou, V Nikolopoulos, and Dimosthenis Anagnostopoulos. A multi-core context-aware management architecture for mixed-criticality smart building applications. In *System of Systems Engineering Conference (SoSE), 2016 11th*, pages 1–6. IEEE, 2016.

[16] Basil Nikolopoulos, George Dimitrakopoulos, George Bravos, Alexandros Dimopoulos, Mara Nikolaidou, and Dimosthenis Anagnostopoulos. Embedded intelligence in smart cities through multi-core smart building architectures: Research achievements and challenges. In *Research Challenges in Information Science (RCIS), 2016 IEEE Tenth International Conference on*, pages 1–2. IEEE, 2016.

[17] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *IEEE transactions on Services Computing*, 3(3):223–235, 2010.

[18] Ismael Peña-López et al. ITU internet report 2005: the internet of things. Technical report, 2005.

[19] Chi Harold Liu, Bo Yang, and Tiancheng Liu. Efficient naming, addressing and profile services in internet-of-things sensory environments. *Ad Hoc Networks*, 18:85–101, 2014.

[20] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International journal of web and grid services*, 1(1):1–30, 2005.

[21] Tao Gu, HC Qian, Jian Kang Yao, and Hung Keng Pung. An architecture for flexible service discovery in OCTOPUS. In *Computer Communications and Networks, 2003. ICCCN 2003. Proceedings. The 12th International Conference on*, pages 291–296. IEEE, 2003.

[22] Marco Crasso, Alejandro Zunino, and Marcelo Campo. Easy web service discovery: A query-by-example approach. *Science of Computer Programming*, 71(2):144–164, 2008.

[23] Guanling Chen, Ming Li, and David Kotz. Data-centric middleware for context-aware pervasive computing. *Pervasive and mobile computing*, 4(2):216–253, 2008.

[24] Tim Kindberg and John Barton. A web-based nomadic computing system. *Computer Networks*, 35(4):443–456, 2001.

[25] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. In *International Conference on Pervasive Computing*, pages 167–180. Springer, 2002.

[26] Unai Alegre, Juan Carlos Augusto, and Tony Clark. Engineering context-aware systems and applications: A survey. *Journal of Systems and Software*, 117:55–83, 2016.

[27] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2012.

[28] Satoshi Asano, Takeshi Yashiro, and Ken Sakamura. Device collaboration framework in IoT-aggregator for realizing smart environment. In *TRON Symposium (TRONSHOW), 2016*, pages 1–9. IEEE, 2016.

[29] YIN Yuehong, Yan Zeng, Xing Chen, and Yuanjie Fan. The internet of things in healthcare: An overview. *Journal of Industrial Information Integration*, 1:3–13, 2016.

[30] Consumption of energy, Eustostat, June 2017. http://ec.europa.eu/eurostat/statistics-explained/index.php/Consumption_of_energy.

[31] European Commission (EC). Europe 2020: a strategy for smart, sustainable and inclusive growth. *Working paper {COM (2010) 2020}*, 2010.

[32] Bharathan Balaji, Chetan Verma, Balakrishnan Narayanaswamy, and Yuvraj Agarwal. Zodiac: Organizing large deployment of sensors to create reusable applications for buildings. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, pages 13–22. ACM, 2015.

[33] Arka A Bhattacharya, Dezhi Hong, David Culler, Jorge Ortiz, Kamin Whitehouse, and Eugene Wu. Automated metadata construction to support portable building applications. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, pages 3–12. ACM, 2015.

[34] Anika Schumann, Joern Ploennigs, and Bernard Gorman. Towards automating the deployment of energy saving approaches in buildings. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, pages 164–167. ACM, 2014.

[35] Carlos Kamienski, Marc Jentsch, Markus Eisenhauer, Jussi Kiljander, Enrico Ferrera, Peter Rosengren, Jesper Thestrup, Eduardo Souto, Walter S Andrade, and Djamel Sadok. Application development for the internet of things: A context-aware mixed criticality systems development platform. *Computer Communications*, 104:1–16, 2017.

[36] Konrad Lorincz, David J Malan, Thaddeus RF Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoffrey Mainland, Matt Welsh, and Steve Moulton. Sensor networks for emergency response: challenges and opportunities. *IEEE pervasive Computing*, 3(4):16–23, 2004.

[37] Geoffrey Werner-Allen, Konrad Lorincz, Mario Ruiz, Omar Marcillo, Jeff Johnson, Jonathan Lees, and Matt Welsh. Deploying a wireless sensor network on an active volcano. *IEEE internet computing*, 10(2):18–25, 2006.

[38] George Vellidis, Michael Tucker, Calvin Perry, Craig Kvien, and C Bednarz. A real-time wireless smart sensor array for scheduling irrigation. *Computers and electronics in agriculture*, 61(1):44–50, 2008.

[39] Zigbee alliance. www.zigbee.org.

[40] Zach Shelby and Carsten Bormann. *6LoWPAN: The wireless embedded Internet*, volume 43. John Wiley & Sons, 2011.

[41] Kyung Sup Kwak, Sana Ullah, and Niamat Ullah. An overview of IEEE 802.15.6 standard. In *Applied Sciences in Biomedical and Communication Technologies (ISABEL), 2010 3rd International Symposium on*, pages 1–6. IEEE, 2010.

[42] Cesare Alippi, Giuseppe Anastasi, Cristian Galperti, Francesca Mancini, and Manuel Roveri. Adaptive sampling for energy conservation in wireless sensor networks for snow monitoring applications. In *Mobile Adhoc and Sensor Systems, 2007. MASS 2007. IEEE International Conference on*, pages 1–6. IEEE, 2007.

[43] Ankur Jain and Edward Y Chang. Adaptive sampling for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*, pages 10–16. ACM, 2004.

[44] Giuseppe Anastasi, Marco Conti, Mario Di Francesco, and Andrea Passarella. Energy conservation in wireless sensor networks: A survey. *Ad hoc networks*, 7(3):537–568, 2009.

[45] Tracy Kijewski-Correa, Martin Haenggi, and Panos Antsaklis. Wireless sensor networks for structural health monitoring: A multi-scale approach. In *Structures Congress: 17th Analysis and Computation Specialty Conference*, pages 1–16, 2006.

[46] Amol Deshpande, Carlos Guestrin, Samuel R Madden, Joseph M Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 588–599. VLDB Endowment, 2004.

[47] Zesong Fei, Bin Li, Shaoshi Yang, Chengwen Xing, Hongbin Chen, and Lajos Hanzo. A survey of multi-objective optimization in wireless sensor networks: Metrics, algorithms, and open problems. *IEEE Communications Surveys & Tutorials*, 19(1):550–586, 2017.

[48] Mahmoud Ammar, Giovanni Russello, and Bruno Crispo. Internet of things: A survey on the security of iot frameworks. *Journal of Information Security and Applications*, 38:8–27, 2018.

[49] L. D. Xu, W. He, and S. Li. Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, Nov 2014.

[50] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the SOA-Based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *IEEE Transactions on Services Computing*, 3(3):223–235, July 2010.

[51] Gonçalo Cândido, François Jammes, José Barata de Oliveira, and Armando W Colombo. Soa at device level in the industrial domain: Assessment of OPC UA and DPWS specifications. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 598–603. IEEE, 2010.

[52] Hausi A Müller, Liam O'Brien, Mark Klein, and Bill Wood. Autonomic computing. Technical report, Carnegie-Mellon Pittsburgh PA Software Engineering Inst, 2006.

[53] Claudio Savaglio, Giancarlo Fortino, Maria Ganzha, Marcin Paprzycki, Costin Bădică, and Mirjana Ivanović. Agent-based computing in the internet of things: A survey. In *International Symposium on Intelligent and Distributed Computing*, pages 307–320. Springer, 2017.

[54] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, pages 259–268. IEEE Computer Society, 2007.

[55] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–69, 2013.

[56] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. on autonomous and adaptive systems (TAAS)*, 4(2):14, 2009.

[57] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.