The Role of Autonomous Aggregators in IoT Multi-core Systems

asil Nikolopoulos, Alexandros C. Dimopoulos, Mara Nikolaidou
eorge Dimitrakopoulos, Dimosthenis Anagnostopoulos

Department of Informatics and Telematics
Harokopio University of Athens
Athens, Greece

ABSTRACT

The Internet of Things constitutes a prominent field, integrating smart devices and people into complex systems that may vary in scale. To ensure the constant availability and performance of provided services, alternative distributed architectures should be explored, promoting system scalability. To this end, alternative architectures for the IoT are proposed. Commonly an intermediate layer consisting of aggregators, controlling sensors and actuators and providing a service interface to IoT applications, is incorporated in such architectures. To promote scalability of IoT systems, aggrerators should to operate as autonomous entities. For an aggregator to become autonomous, self-management policies should be enforced. In the paper, we discuss autonomous aggregator software, running on multi-core IoT systems to efficiently implement such policies. A demonstrator for smart buildings, developed as a proof of concept for the proposed concepts, is also presented.

ACM Classification Keywords

C.2.c. Sensors and Actuators: M.4.Service-oriented Architecture;

Author Keywords

Internet of Things, Mulit-core Systems, Autonomy, Aggregators and Sensors

INTRODUCTION

The Internet of Things (IoT) promotes the integration of smart devices and people to provide services anytime, anywhere, changing every-day activities. The application of the IoT technology is gaining momentum in different areas, such as e-health, transportation, smart city and building operation. As the IoT services become more popular, the systems supporting them become more complex.

There are numerous efforts [4] to promote distributed architectures for implementing complex IoT systems, ensuring their scalability. Following recent trends in Edge [1] and Fog [11] computing for the IoT, the concept of providing an intermediate computing layer operating closer to sensors and user devices promotes scalability, availability and performance. Corresponding components, called aggregators, gateways or edge nodes, control sensors and render services to IoT user applications. Autonomous operation of such components, utilizing the context they operate in [26], ensures the scalability of IoT systems, thus is of great importance.

In this paper, we discuss aggregator software for multi-core IoT systems, enabling them to act as autonomous entities. For an aggregator to become autonomous, self-management policies have to be enforced [21]. The properties and components of aggregator software running efficiently on multi-core systems in order to implement such policies is discussed in the following. The aggregators for a smart building management system have been implemented as a proof of concept for the proposed approach and different scenarios were explored targeting the minimization of energy consumption of aggregators and sensors, while maintaining the necessary performance to effectively support IoT application services. The aggregator software performance was also evaluated under different amount of load.

RELATED WORK

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 [4] 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 SOA, providing an interoperable way of communication. However, SOA concepts were originally designed for dealing mainly with few, complex and mainly static enterprise services [18]. Therefore, the trend is to create SOA web services, allowing the horizontal and vertical collaboration among IoT devices, online services, users, etc. [18]. 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 [25] while other researchers propose either three layers [5] or four [22].

Typically, a web service model consists of a service provider, a service registry and a service consumer [15]. For a client

IoT 2017, October 22–25, 2017, Linz, Austria

DOI: https://doi.org/10.1145/3131542.3131548

application (service consumer) to connect to such a server, a network address and port is essential to be known [17], as well as information on how to communicate (protocol, language, and mechanisms to use). These 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 [12] or other more recent forms of resource discovery [28, 10].

The systems of the IoT are becoming cheaper, smaller in size and more capable as time passes and one of the most enabling aspects for these augmented capabilities 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 [20] and work by Henricksen et al. [19] 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 [2].

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 [6]. 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.

All of the above characteristics should be integrated in order to effectively implement IoT SOA infrastructures.

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. [13]. 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 [3]. The existence of aggregators or gateways has also been adopted by potential standards, as Edge [1] and Fog [11] 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 [30]. Following the proposed concepts of Edge or Fog nodes, aggregator services may run in multi-core devices, as for example 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. Moreover, the existence of an aggregator may contribute in the energy conservation of each sensor and the whole network itself [29]. 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. Although, in many cases [4, 26, 15] aggregators become intelligent and may obtain some self-configuration features, there is a need for a generic approach towards an aggregator software architectural framework for implementing self-management policies in a unified and extendable fashion.

THE ROLE OF AGGREGATORS IN A SOA FOR MULTI-CORE IOT SYSTEMS

A Service-Oriented Architecture for multi-core IoT Systems in the context of EMC^2 ARTEMIS Joint project (grant agreement n^o 621429) is proposed in [24]. Although it focuses on smart buildings, it may be applied for the support of different IoT systems. Aggregator services (similar to Edge or Fog nodes) run in multi-core devices, as for example Raspberry Pis, to control sensors and provided services to IoT user applications entering their area of control. The concept of context-aware IoT [26] is also adopted, enabling aggregators and IoT user services to operate in an autonomous fashion.

Aggregators act as a middleware between sensors and IoT user services programmed and operated based on the service-computing paradigm (SOA). An aggregator's specialized hardware and software enables the communication with diverse WSNs based on different technologies, while at the same time, sensor's APIs are exposed as a set of REST services to be used by user services. Moreover, aggregators may provide extra services that derive from the available context of the system. For example, if we consider the system set in a building environment, a user may be interested in the mean temperature of a specific room. In another example, if the system is set in a car, a user may be interested in the mean fuel consumption or the minimum tyre pressure of the wheels. Sensors create clusters, and each aggregator is responsible for one cluster. The communication within this clusters is based on specific protocols compatible with the sensor devices (e.g. Zigbee). It is apparent that as the WSNs grow in heterogeneity, the number of aggregator units will grow, and this will create service discovery problems.

In order for an IoT user service to ask for a sensor service from an aggregator, it should know how to contact the said unit, or know where to find a specific service. For this reason, a Registry Unit is integrated in the architecture. The registry acts as a service discovery entity, where aggregators are required to register their services and users can poll the registry unit for information about services or aggregators. In order to cater for the multi-criticality aspect, the service requests come with some extra contextual parameters. Each call to a service is accompanied by a criticality level that the user would like this request to be treated with. The criticality level defines how important a service call is, and will be discussed in detail later

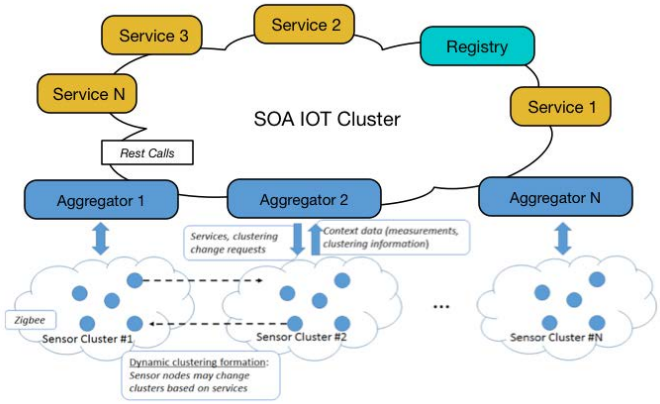

Figure 1: SOA for Multi-core IoT Systems

in the paper. By using the criticality level of each request, the aggregator unit is able to treat each request differently.

AUTONOMOUS AGGREGATOR SOFTWARE

In the following, we focus on policies enforced towards aggregator's autonomy in the context of the proposed architecture of Figure 1 and the way they may be implemented by aggregator software in a generic fashion. To retain autonomy, the aggregator has to manage its resources in order to adopt to the current state of the IoT system.

to adopt to current context (for example service load) and in order to achieve self-configuration [21], self-adaptation [27] and self-healing [16]. The resources the aggregator has to manage are: The aggregator energy consumption and/or battery power, the sensor energy consumption and/or battery power, the aggregator cores and the sensors managed by the aggregator. For the management of these resources a set of policies is imposed for achieving self-healing, self-adaptation and self-configuration.

Self-configuration policies

Multi-critical request management

In order to manage a multi-critical system, one should ensure that higher criticality requests are serviced faster than lower ones. In [9] 5 levels of criticality are proposed. Level 1 is the lowest criticality level, 2 and 3 mid criticality, and finally 4 and 5 the highest criticality levels available. The aggregator should decide upon the scheduling algorithm assigning jobs to cores, taking into consideration their criticality level. Numerous algorithms may be applied as discussed in [9].

Aggregator Resource Management

This policy aims at minimising the energy consumed by the aggregator unit by efficiently managing its cores by constantly monitoring its load per core. Upper and lower thresholds are defined during the aggregator configuration for each core utilisation. Thresholds may be determined for specific aggregator and hardware pairs after performing stress tests. Each core may be characterised as *utilised, over-utilised* and *under-utilised* and may be activated or deactivated accordingly. Under-utilised cores are deactivated to preserve energy. When active cores are over-utilised, another one is activated. In case all cores are over-utilised, the aggregator unit is deemed in "overloaded mode", then in addition to the next policy described, the aggregator assigns requests to cores in a round robin manner up until requests are dropped because they cannot be served. The aggregator exits "overloaded mode" when at least one core is not over-utilised anymore.

Aggregator Service QoS Preservation

The aim of this policy is to maintain the quality of service provided by the sum of the aggregator units as a system. In order to do so, aggregator units can ask other aggregator units to manage a sensor, provided that the aggregator receiving the sensor, can actually receive data from it, i.e. is in range. Whenever the aggregator enters "overloaded mode" it can start asking other aggregator units, if they can handle a sensor unit that was originally in its influence. The aggregator will choose the sensor unit by measuring how many requests each sensor unit receives, and choose the first in the list. After finding the sensor unit that should be passed over to another aggregator unit, the aggregator unit will then ask all aggregator units available through the registry unit if they can handle the specific sensor. The aggregator units will have to assess their load levels and whether they can receive that unit, by issuing a poll and checking if they can receive data from it, and reply accordingly to the aggregator unit that initiated the request. 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. If a candidate is not found, then the aggregator tries again with another sensor unit, until it exhausts its sensor list.

Self-healing Policies

Erroneous Data Filtering

The aim of this policy is to be able to filter out readings that are erroneous in order to provide correct data to the users. In order to accomplish that we have to add some extra context to the requests. The aggregator can provide the data requested in two ways: either by requesting the variable measured from a sensor, or by requesting a variable that is measured within a contextual or conceptual space, for our case, e.g. a specific room, but this can be expanded to include any contextual or conceptual grouping idea that can correlate sensors' readings among them for the same environmental variable.

Sensor Health Monitoring

The aim of this policy is to maintain a fresh list of sensor units that are up and running. To achieve this, the aggregator runs a periodic poll in the WSN that has to be answered by every sensor. In addition the aggregator can timestamp every time a sensor has communicated with it, and save it in its memory. This includes sensor reading answers and poll answers. If a sensor unit fails to communicate with the aggregator unit for a specific time interval, it is deemed as not any more in the influence area of this aggregator, and removed from the active sensors list. In order to renew its lease with the aggregator, a sensor unit has to respond to the periodic poll, or send a sensor reading within that timeframe.

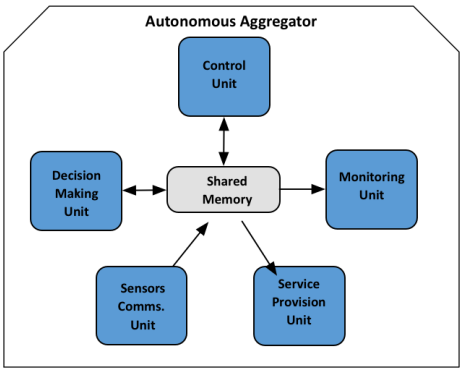

Figure 2: Autonomous Aggregator Software Compoments

Self-adaptation Policy

Sensor Communication Mode

The aim of this policy is to conserve energy by minimizing the communication needed to be made by the sensor units. Although this may seem straightforward at first, with a very basic logic being that the sensor unit should only communicate when polled to do so, a more sophisticated approach would be that depending on the situation, the sensor unit could push data or have its data pulled, or not polled at all for a reading and have a cached version of its data returned to user. Having said the above, the aggregator unit is able to ask a sensor to either use a push or a pull protocol for data retrieval. While in push mode the sensor emits readings and data at a constant rate, in order to provide fresh data to the aggregator unit. On the other hand, in pull mode, the aggregator unit queries the sensors for readings and expects an answer from the sensors. Both choices have their merits and drawbacks and should be used under the correct circumstances.

Aggregator Software Components

Applying self-management policies is not a trivial task, rising performance and complexity management issues in aggregator software implementation. Suggested aggregator components are depicted in Figure 2. Each software component provides a different utility to the aggregator. Some modules are essential for its operation, even if no self-management policies are enforced. Some others, as *Monitoring* and *Decision Making* units, utilise self-monitoring and self-management. The software components work independently and communicate with each other through a shared memory space, when needed.

The *Control Unit* is responsible for initialising the other software modules, and in addition has some basic control over the aggregator unit as a whole. It can spawn Request Execution Threads in order to serve requests, received via http by *Service Provision Unit* and is able to receive messages directly from the *Sensor Connectivity Unit*, responsible for handling sensors. None of them, on its-own contributes to the aggregator autonomy.

Service Provision Unit provides a JSON REST API to serve a) Sensing requests, forwarded to the WSN, b) Description requests, providing the description of a service provided, c) Sensor list requests, for currently available sensors and d) conceptual grouping area list requests, identifying available conceptual grouping areas (rooms) serviced by this aggregator and their according services provided. All requests have a criticality level (from 1 to 5). Sensor Communication Unit is the only part of the aggregator that is sensor dependent. It is responsible for communicating with the underlying WSN, and as such, for each different type of WSN sensor units, corresponding management and communication libraries have to be developed.

Shared memory provides the management of *monitoring variables*, describing the status of the resources managed by the aggregator and policies enforced for their management. Statistical data related to completed service requests, focusing on criticality data, are also maintained. Monitoring variables provide information for the aggregator to all its components and offer the context to enable a) the *Decision Making Unit* to enforce self-management policies and b) the *Monitoring Unit* to monitor the status of all the resources.

The Monitoring Unit monitors metrics that help with the overall management of the aggregator unit. It is invoked on predefined monitoring intervals and determines whether the status of the aggregator has changed within the last interval. If so, it invokes the Decision Making Unit to suggest changes in the aggregator configuration, for example turning on/off a core. It should be noted that the Decision Making Unit is responsible for suggesting configuration changes according to the policies enforced. The Control Unit is responsible for making the changes. The policies enforced are implemented within the Decision Making Unit in a form of event-action rules [7].

As an example, the rules corresponding to Aggregator Resource Management policy, enforced for each core, are listed in the following:

```
all (Current per core load >= Maximum per
core load) :- controlUnit.activateCore for
(x) (Current per core load >= Maximum per
core load) :- controlUnit.shutdownCore(x)
(all (Current per core load >= Maximum
per core load) and (Active Cores = Number
of Cores)) :- controlUnit.EnterOverloaded
Mode() (any (Current per core load >= Maximum
per core load) and (OverLoadedMode=On)) :-
controlUnit.ExitOverloadedMode
```

Events are depicted using Complex Event Processing (CEP) notation [23]. Simple events consist of simple comparisons between monitoring variables. Complex events are created based on simple ones utilising CEP operators. Actions indicate Control Unit interfaces to be invoked.

SMART BUILDING DEMONSTRATOR

The proposed autonomous aggregator software was adopted to implemented aggregators integrated within SOA infrastructure for smart building applications, constructed in the framework of EMC^2 ARTEMIS Joint project . The overall architecture of the system is depicted in Figure 3.

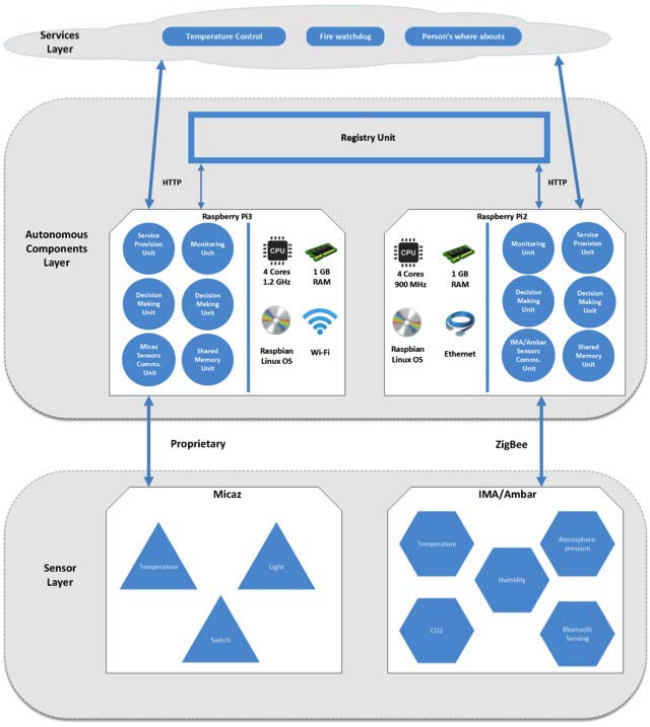

Figure 3: Smart Building Architecture

Two discrete sensor technologies were integrated within Smart Building SOA: sensors from the MicaZ family, and the IMA/Ambar Sensors. The sensors are setup to monitor a single room. The provided functionality from the MicaZ sensors includes: a) sensing room temperature, b) sensing levels of ambient light and c) the ability to turn on and off devices. For demonstration purposes, the switches were used to emulate a number of devices, which are, a ventilation system, an alarm, and fire sprinklers. IMA/Ambar sensors provide the following functionality: a) sensing room temperature, b) sensing humidity levels, c) sensing atmospheric pressure, d) sensing CO_2 levels, and e) sensing of Bluetooth Tags. Sensors of each category use different protocols for communication purposes: MicaZ uses a proprietary protocol and the IMA/Ambar uses Zigbee.

To seamlessly integrate both technologies within Smart Building infrastructure, two aggregator units were deployed, as shown in Figure 3, each running on different hardware. The aggregator software was developed in JAVA and is not hardware dependent. The first aggregator unit was deployed on a Raspberry Pi 3 (4 CPU cores @1.2 GHz, 1 GB of RAM and Integrated WiFi). The second aggregator unit was deployed on a Raspberry Pi 2, which is similar to Pi 3 but with the 4 CPU cores running at 900 MHz and without integrated WiFi capabilities. Both of the units ran Raspbian distribution of GNU/Linux. The Raspberry Pi 3 was used to provide aggregation to the MicaZ WSN while the Raspberry Pi 2 was used for the latter WSN. In addition, for the communication with the rest of the network, the MicaZ aggregator used WiFi communications while the IMA/Ambar Aggregator relied on Ethernet networking. The sensor communication unit is responsible of applying a pull or push protocol for communication purposes. Which to choose is part of the aggregator's self-adaptation Sensor Communication Mode policy, depending on the criticality level of the request served. The number of cores used is part of self-configuration Aggregator Resource Management policy, while forwarding the control of a sensor to a neighbour aggregator relate to self-configuration QoS Preservation policy. The implementation of these policies is demonstrated in the following, as an example of the provided autonomy features. All self-management decisions have a twofold goal: reduce energy consumption and ensure the overall system performance and seamless operation. Details on the provided aggregator API and the implementation of the Registry Unit can be found in [8] and [14]. In the following, we focus on the implementation of the aggregator module and its autonomy features.

IoT Services operating in Smart Building Demonstrator

To demonstrate the operation of autonomous aggregators under different load and criticality requirements, a few services were implemented. The services operate on a simple laptop computer or smart phone under conditions that may generate mild or heavy load to monitor aggregator behaviour under conditions leading to self-configuration or self-healing. For each service a simple UI was created in order to be able to see the actions taken from service side and enable to easily capture footage for the demonstration. They are:

Room Temperature Control

It controls the heating of a specific room. The user can set a desired temperature and the services applies it with the aid of one or more temperature sensors and one or more actuators that control heating units at the premises. The service is not considered of high criticality and all corresponding aggregator request are invoked by it with criticality level 2. One can see an example of the service running in Figure 4. As shown in the figure, for the demonstrator purposes, an initial temperature, lower than the room temperature was set as the temperature setpoint. This causes the service to invoke the corresponding request from the MicaZ aggregator which would turn on the ventilation system, and ask the action to be performed. In order for the ventilation to turn off, the room temperature has to drop at least 2 degrees below the wanted temperature. After turning on the ventilation system, a new temperature setpoint is set. The new temperature was at least two degrees higher than the current temperature in order to trigger the action of turning off the ventilation system (see Figure 4).

Fire Watchdog

This service retrieves data from both aggregator units in order to sense the room temperature, the CO_2 levels of the room and the ambient light. The service constantly monitors the room temperature levels using criticality level 3 (medium). If the temperature rises above a preset threshold, then the service starts using a higher criticality level on all of its requests (4) asking for CO_2 data. Should CO_2 rise above a certain level as well, then the service once more increases the criticality level of its calls and uses the high criticality level, 5. In addition it starts asking for ambient light data. Should this rise above a

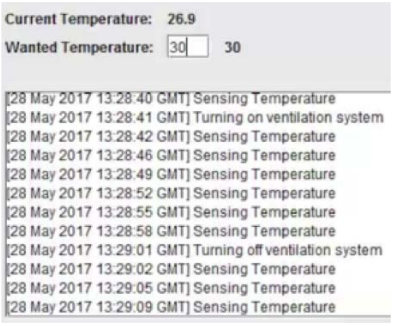

Figure 4: Room temperature control Service

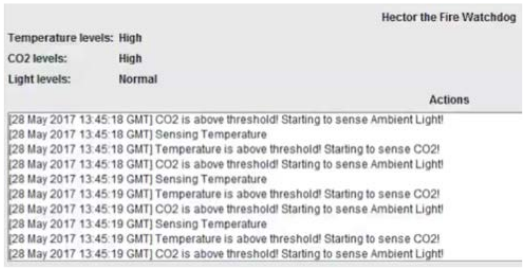

Figure 5: Fire Watchdog Service

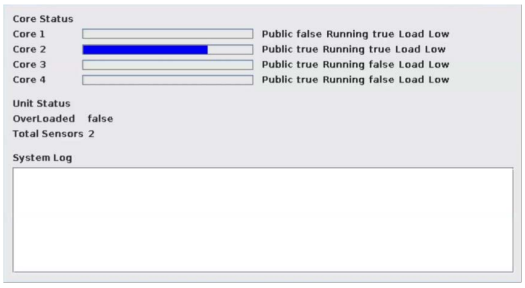

Figure 6: Aggregator Operation - System is beginning to work

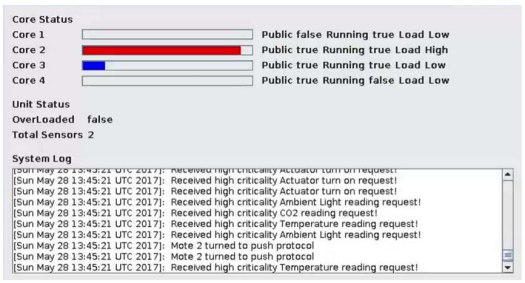

Figure 7: Aggregator Operation - Push protocol is activated

certain threshold, then an alarm is sounded and the sprinklers are activated. Should at any point a variable fall below the preset threshold, then the service stands down and resumes normal operation (see Figure 5).

When both services are operating simultaneously in different devices, the aggregators have to serve *temperature sensing* requests of different critically levels. Each temperature sensing request computes temperature as the average of all temperature sensors controlled by the aggregator.

Person's Whereabouts Service

The service searches for a person based on a Bluetooth tag the person bears, or the person's cellphone's Bluetooth. In its primitive form, it will search for a Bluetooth signal from a specific source, and report if the person is in the aggregator's influence, which means inside the room. The service can operate in two modes: 1) it seeks the Bluetooth Tag just once (criticality level 4) or 2) it constantly looks for the Bluetooth Tag and once found, it informs the user (criticality level 2).

Autonomous Aggregator's Operation

Within the rooms the two aggregators were controlling, their operation was tested under conditions of heavy load. Different scenarios of service operation were explored, generating different amount of load and criticality requirements. In the following, we focus on the Raspberry Pi 2 aggregator for demonstration purposes. The aggregator status screen is depicted in Figure 6. The unit's overall status and the total number of sensors managed by this unit is recorded. The horizontal bars show the load of each core, and next to that you can see whether the core is reserved, is running and the load level of the core. In the log window one may see self-configuration decisions suggested by Decision Making Unit, that would otherwise be transparent.

As shown in the figure, one core is reserved to execute the aggregator software itself, while only one core is utilised to serve service requests. Two sensors are currently active. In Figure 7, the aggregator receives enough load to cause the first core to be overloaded. A second core is turned on, and starts receiving requests. Decision Making Unit makes these decision taking into consideration the following monitoring variables, stored in *Shared Memory*: *Current per core load*, measuring active threads, and *Maximum and Lower per core load*, indicating the boundaries marking when a core is over- or under-utilised. In the second case, the core is turned off. In the first case boundaries are set during the aggregator's setup and are based on core stress test performed when configuring the aggregator. In this particular case, utilisation boundaries are measured as the max. and min. of threads that may run on a specific core. One should take into account that all the requests supported by the sensors in the demonstrator need almost the same processing power to be completed. Thus, active threads may be used as the indication of core load.

Another self-configuration option is also depicted in Figure 7. As recorded in the log, the Fire Watchdog service detected temperature rising and is requesting additional measurements (CO_2, ambient light) with a hight criticality level. This causes, the Decision Making Unit to be invoked and change the sensor communication protocol mode from 'Pull' to 'Push'. Push protocol enables request to be served faster, since updated sensor measurement values are stored in the aggregator's cache, but increases the energy consumption of sensors, working on

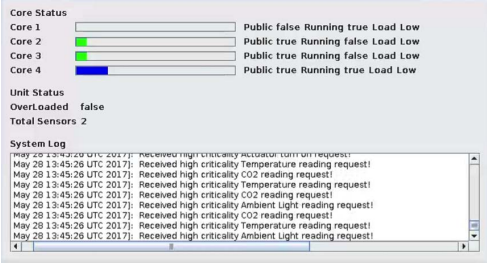

Figure 8: Aggregator Operation - Turning off cores

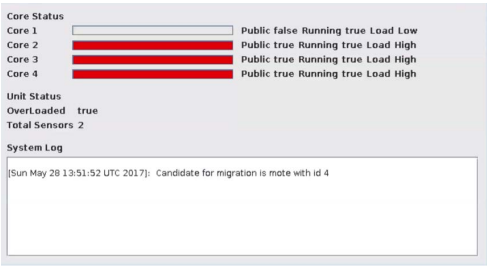

Figure 9: Aggregator Operation - Handover Sensor

batteries. The communication mode used is stored in *Communication Mode* monitoring variable. The communication protocol mode is set back to 'Pull' in the next monitoring interval (the *Monitoring Unit* is invoked periodically), whether no service request with `criticality level`$\geq$`4` has been received.

Load balancing option is depicted in Figure 8, as the load is balanced on 3 cores. The two cores with low loads (green ones) would be shut down in the next monitoring interval. Finally, in Figure 9 on may see what happens when all cores are overloaded. This causes the Overloaded flag to become true, and the self-healing properties come in effect. The aggregator starts asking its neighbors to accept a sensor of its, in order to relieve load. Currently, the sensor to migrate is chosen randomly. Other policies may be applied as well.

Discussion

The demonstrator serves two purposes: a) to explore the potential of implementing complex self-management policies, building adoptable aggregator software running on multi-core devices, that may serve as aggregators and b) to explore aggregator software performance issues, under different circumstances (configuration, load, etc).

In more detail, for the first purpose, the software development should be cost-efficient, while the software itself should be easily deployed in many devices. This is the reason why, it was decided to be implemented in Java. Efficient memory management techniques were adopted (using existing libraries) to ensure its performance. Another issue at hand was to design it in order to be easily extendable. The concept of monitoring variables associated with specific policies and the modular implementation of monitoring and, especially decision making unit, introduced to achieve self-management, enable different policies for discrete design issues to be integrated within a single implementation. For example, different algorithms may by applied for scheduling multi-critical requests in active cores. To integrate such feature the following steps should be taken: 1) Implement discrete algorithms as libraries, 2) A corresponding Monitoring Variable is maintained to indicate selected policy, 3) Corresponding self-configuration policy is implemented by adding proper rules in the Decision Making Unit.

While for the second purpose, the demonstrator indicated that the operation of the autonomous aggregator was efficient under heavy load created by supported service, though IoT configuration complexity was not explored at this stage. Neither the number of sensors nor the number of aggregators was extensive. However, the fact that aggregator software performance was efficient and not affected by different load conditions, indicated the potential of the proposed approach. The numbers of controlled devices is limited by the aggregator's own resources and affected by service load. Furthermore, the demonstrator focused on the aggregator operation itself. Neighoring aggregator efficient interaction should be further investigated.

The next step in evaluating autonomous aggregator software is to test its performance in large-scale IoT architectures, using both simulation tools and real-world case studies.

CONCLUSION

Promoting autonomous operation of aggregators integrated within SOA for the IoT multi-core systems was discussed in this work. The aggregator itself is designed as a system consisting of independent components. Self-management policies were introduced and the implementation of monitoring and decision making components, promoting their enforcement was discussed. Future work is concentrated in extending the decision making functionality, allowing neighboring aggregators to share their policies and experiences without enforcing any centralized control in decision making. Also performance issues should be explored in large-scale IoT systems.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement n^o 621429.

REFERENCES

1. A. Ahmed and E. Ahmed. 2016. A survey on mobile edge computing. In *2016 10th International Conference on Intelligent Systems and Control (ISCO)*. 1–8.
2. Unai Alegre, Juan Carlos Augusto, and Tony Clark. 2016. Engineering context-aware systems and applications: A survey. *Journal of Systems and Software* 117 (2016), 55–83.
3. Satoshi Asano, Takeshi Yashiro, and Ken Sakamura. 2016. Device collaboration framework in IoT-aggregator for realizing smart environment. In *TRON Symposium (TRONSHOW), 2016*. IEEE, 1–9.
4. Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010a. The Internet of Things: A survey. *Computer Networks* 54, 15 (oct 2010), 2787–2805.

5. Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010b. The internet of things: A survey. *Computer networks* 54, 15 (2010), 2787–2805.
6. Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. 2012. Scheduling real-time mixed-criticality jobs. *IEEE Trans. Comput.* 61, 8 (2012), 1140–1152.
7. Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. 2010. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing* 6, 2 (2010), 161–180.
8. George Bravos and et.al. 2015. An autonomic management framework for multi-criticality smart building applications. In *13th INDIN Conference*. IEEE, 1378–1385.
9. Alan Burns and Robert Davis. 2013. Mixed criticality systems-a review - 8th edition. *Department of Computer Science, University of York, Tech. Rep* (2013).
10. Guanling Chen, Ming Li, and David Kotz. 2008. Data-centric middleware for context-aware pervasive computing. *Pervasive and mobile computing* 4, 2 (2008), 216–253.
11. M. Chiang and T. Zhang. 2016. Fog and IoT: An Overview of Research Opportunities. *IEEE Internet of Things Journal* 3, 6 (Dec 2016), 854–864.
12. Marco Crasso, Alejandro Zunino, and Marcelo Campo. 2008. Easy web service discovery: A query-by-example approach. *Science of Computer Programming* 71, 2 (2008), 144–164.
13. Anind K Dey, Gregory D Abowd, and Daniel Salber. 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-computer interaction* 16, 2 (2001), 97–166.
14. Alexandros C Dimopoulos and et.al. 2016. A multi-core context-aware management architecture for mixed-criticality smart building applications. In *11th SOSE Conference*. IEEE, 1–6.
15. Schahram Dustdar and Wolfgang Schreiner. 2005. A survey on web services composition. *International journal of web and grid services* 1, 1 (2005), 1–30.
16. Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. 2007. Self-healing systems — survey and synthesis. *Decision Support Systems* 42, 4 (jan 2007), 2164–2185.
17. Tao Gu, HC Qian, Jian Kang Yao, and Hung Keng Pung. 2003. An architecture for flexible service discovery in OCTOPUS. In *Computer Communications and Networks, 2003. ICCCN 2003. Proceedings. The 12th International Conference on*. IEEE, 291–296.
18. Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Domnic Savio. 2010. 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 (2010), 223–235.
19. Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. 2002. Modeling context information in pervasive computing systems. In *International Conference on Pervasive Computing*. Springer, 167–180.
20. Tim Kindberg and John Barton. 2001. A web-based nomadic computing system. *Computer Networks* 35, 4 (2001), 443–456.
21. Jeff Kramer and Jeff Magee. 2007. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*. IEEE Computer Society, 259–268.
22. Chi Harold Liu, Bo Yang, and Tiancheng Liu. 2014. Efficient naming, addressing and profile services in Internet-of-Things sensory environments. *Ad Hoc Networks* 18 (2014), 85–101.
23. David Luckham. 2015. *Event Processing for Business: Organizing the Real-Time Enterprise*. Wiley Publishers.
24. Basil Nikolopoulos and et.al. 2016. Embedded intelligence in smart cities through multi-core smart building architectures: Research achievements and challenges. In *10th RCIS Conference*. IEEE, 1–2.
25. Ismael Peña-López and others. 2005. ITU Internet report 2005: the internet of things. (2005).
26. C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos. 2014. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys Tutorials* 16, 1 (First 2014), 414–454.
27. Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)* 4, 2 (2009), 14.
28. Michael F. Schwartz, Alan Emtage, Brewster Kahle, and B. Clifford Neuman. 1992. A comparison of internet resource discovery approaches. *Computing Systems* 5, 4 (1992), 461–493.
29. Huseyin Ugur Yildiz, Kemal Bicakci, Bulent Tavli, Hakan Gultekin, and Davut Incebacak. 2016. Maximizing WSN lifetime by communication/ computation energy optimization of non-repudiation security service: Node level versus network level strategies. *Ad Hoc Networks* 37 (2016), 301–323.
30. YIN Yuehong, Yan Zeng, Xing Chen, and Yuanjie Fan. 2016. The Internet of Things in Healthcare: An Overview. *Journal of Industrial Information Integration* 1 (2016), 3–13.