Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 1207–1214
DOI: 10.15439/2014F324
ACSIS, Vol. 2

Comparison of architectures for service management in IoT and sensor networks by means of OSGi and REST services

Jarogniew Rykowski, Daniel Wilusz
Poznań University of Economics, Niepodległości 10, 61-875 Poznań, Poland
e-mail: {rykowski, wilusz}@kti.ue.poznan.pl

Abstract—In this paper two alternative architectures for service management in IoT and sensor networks are discussed. The first one is based on Open Service Gateway (OSGi) framework and Remote Services for OSGi (R-OSGi) bundle. The second architecture extends the notion of REST (Representational State Transfer) paradigm. There were few purposes of the extension. First, efficient, dynamic searching for devices capable of fulfilling certain requests within actual context was enabled. Second, both the devices and controlling services were distributed. Next, the devices were orchestrated in order to provide complex functionality. Finally, the access to the devices' functionality was standardized. OSGi-based solution was found simpler and better suited for homogeneous sensor networks, while more complex REST-based framework appeared as better suited for heterogeneous and widely distributed IoT devices and services.

I. Introduction

The Internet of Things (IoT) is the continuously evolving concept, which influences the business processes and even society on a global scale. Historically, the IoT was perceived as the intelligent network connecting objects, information and human beings to enable remote coordination of resources by people and machines [1]. First proposals of architectures of IoT networks were based on their natural predecessor – sensor networks. Nowadays IoT is perceived as cutting edge phenomenon, no longer limited to electronic identification of objects, but defined as technology integrating devices with information network, where these devices act as active participants in business processes [2].

Multiple applications of Internet of Things have been identified and implemented in such economy areas, as manufacturing, supply chains, energy, healthcare, automotive industry, insurance, financial services or research laboratories, to mention a few [2][3][4]. In the nearest future, the expansion of Internet of Things outside the internal infrastructures of companies is not only expected, but becomes the reality. The increasing popularity of Internet of Things causes the need for proper architecture, which will meet the requirements of IoT environment.

This work was supported by the National Centre for Research and Development under grant POLLUX-II-/1/2014

As IoT is very dynamic and heterogeneous, efficient management system for this environment should address these features. In this paper we discuss two alternative architectures for service management in IoT and sensor networks. The remainder of the text is structured as follows. Section II presents basic characteristics of IoT and sensor networks, which allows an enumeration of basic requirements of these environments. The Open Service Gateway (OSGi) framework is briefly described in Section III. Section IV focuses on the introduction of the Representational State Transfer (REST) methodology. OSGi-based architecture for IoT management is proposed in Section V. Next, similar architecture based on extended REST services is described in Section VI. Finally, Section VII concludes the paper.

II. IoT and sensor networks – basic features

At the first view sensor networks and Internet of Things are similar. Both networks are composed of small hardware nodes with limited resources, such as memory and CPU capabilities, both are physically distributed over certain area, both communicate by means of certain standards related with TCP/IP protocol. However, this similarity is seeming. Below, a comparison of basic features of these two network types , as well generic requirements for data acquisition and overall architecture are provided.

A typical sensor network is composed of many nodes having similar purpose, common goal and fixed functionality. Usually, one single point of interaction is assured, to contact the network as a whole, rather than particular nodes directly. As the nodes are usually small battery-operated hardware devices, several techniques are applied for energy saving and optimization of information routing.

Sensor network acts as single entity, thus it is usually controlled in the centralized manner by a single owner/administrator. The network is accessed as a "black box" of certain functionality, with strict access rights to particular global functions. As there is no need for individual addressing of devices and their functions, external access and control of individual nodes is usually blocked, and the

978-83-60810-58-3/$25.00

network is self-manageable. For example, for energy-saving reasons, some nodes are temporary deactivated, gathered information is pre-processed to minimize network transfer, etc. [5]

On the contrary, a typical IoT network is composed of heterogeneous nodes of different purpose and functionality. There is no common goal defined for the network as a whole except for very abstract goals such as "user comfort" or "energy savings". Instead, each networked IoT device, via continuous environment monitoring and information exchange with other devices, tries to act as a "good servant" [6] – invisible, however useful to maximum extent.

IoT networks are composed ad-hoc and as such they have no centralized management, single owner/administrator, global access rights, etc. The interaction with humans is incidental, sometimes even transparent to users – they are not necessary conscious about the functions provided, even if these functions rise the level of "user comfort" (such as automatic heating, air-conditioning, etc.), "energy savings" (automatic switching off of some devices once no human is detected in the neighborhood), "safety" (face scanning allows for transparent control of visitors of an office, rising "silent alarm" once an unknown person is detected inside), etc. This interaction depends on local and global context, based on such parameters as geo-location and environment features (temperature, lightness, etc.).

As the main stress is put on efficient ad-hoc interaction among humans and IoT devices, portability is the key. Interaction should abstract of such details as local addresses of IoT devices, communication means, data format, etc. Instead, an efficient searching/filtering mechanism should be provided, allowing to choose "the right" device to serve the request in given context. More precisely, portable interaction should be characterized by:

- a need for individual addressing of given network functions abstracting the implementation of these functions, including strict device addressing;
- a need for filtering "the best" device to be activated to realize certain request for function; similar – a need for searching for and choosing the "right" device(s);
- a need to distinguish several searching modes: "exactly one device capable of action X", "at least one", "everyone", etc.;
- a need for monitoring overall activity of devices (and accessibility of their functions) ;
- a need for portability due to ad-hoc nature of interactions at several places and for different situations: at least portability of requests – a need for common semantics/communication language to formulate the requests despite devices' specificity;
- a need for scalability, both for number of requests as well as the devices.

As may be drawn from the above enumeration of the needs, there must be a global (thus centralized) management mechanism for controlling the set of IoT devices, similar to a typical way of the management of the sensor networks. This is somehow contradictory to the requirement for the autonomy of the IoT devices and their ad-hoc composition and usage. However, without such centralized mechanism it is not possible to group the devices into bigger conglomerates (to orchestrate and thus multiply their functions), to search for the device which is optimal to serve given request, to balance the system load/energy efforts/network traffic, etc. Having in mind this trade-off, one may find that the optimum solution is to provide a local catalogue of devices' possibilities, extended by some statistical operations such as current load for each device, information about its accessibility and possibly temporal unavailability, number of served requests, last activation time, etc. The catalogue may also play a role of the request broker, mapping syntax and semantics of the incoming requests to the syntax and semantics used to contact the (heterogeneous by their nature) devices.

If starting the discussion on choosing the technology to implement the above-mentioned catalogue of the devices and their functions, we may point out two frameworks for centralized management of distributed resources: OSGi platform for Java programming language implementing a dynamic service registry, and extended REST services capable of centralized management of distributed REST resources and servers. In the next chapters we are going to present and compare these two frameworks, taking into account the following operations:

- registering the device/function;
- providing individual proxy for interaction/communication mapping – starting, stopping, and suspending/resuming the service;
- monitoring real-device state and providing information about device accessibility;
- searching for the device(s) to serve given request;
- if a request is to be possibly served by several devices, choosing one device based on context and statistical information collected during past activations;
- orchestrating device functions taking into account local context;
- administrating device parameters (individual GUI for each device).

III. OSGi FRAMEWORK

Open Service Gateway (OSGi) is a popular framework, which provides specification for dynamic module system for Java [10]. The core function of OSGi implementation is related with efficient management of modules (bundles) lifecycle, which may be dynamically installed, started, stopped, uninstalled, etc. Bundles are building block of OSGi-based modular systems, which are able to mutually interact. Following interactions among bundles may be distinguished:

- sharing Java packages with classes and interfaces;

- registering and calling services;
- managing the lifecycle of other bundles;
- sending and handling events to trigger specified actions.

OSGi implementations provides service registry to control functions provided by the bundles. There is no reference needed to invoke the service, as OSGi environment provides services based on their interface and additional metadata describing the service. The OSGi service registry may return zero, exactly one or multiple services compliant with the request. If there is no service of specified interface or meeting the additional constraints, null value is returned and must be properly handled. If the caller tends to invoke only one service, but in the registry there are few possible services meeting the requirements, OSGi environment returns the service of the highest rank (specified during service registration) and the lowest identifier (i.e., the oldest one) [11] [12].

Event based communication is a useful feature of the OSGi framework. Primarily it allowed handling data changes caused by dynamic behavior of the bundles or framework itself. Each bundle may be programmed to individually react to specified events, e.g., to stop if a dependent bundle is stopped or uninstalled. Since OSGi v.4, the mechanism of sending and handling the events defined by software developers has been introduced. The event-based communication allows the definition of events' topics and accompanied metadata, which are handled by dedicated `EventHandler` services [11][13][14].

Dynamic modularity, service registry and event-based communication seem to be out of the box solution for IoT systems. However, OSGi does not meet all the requirements listed in Section II that is the reason why some adaptations are needed. We propose such extensions further in Section V.

IV. REST

The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application [7].

Key addressable entity of REST environment is a resource. Resources are any named pieces of information, being a target of hypertext links. Uniform Resource Locator (URL address) is used to get the value of a resource from the server: either static (if the resource is a file, piece of text, an image, etc.), or dynamic, being a result of an invocation of a piece of program code. In the latter case, the resource is treated as a "black box" from the point of view of its caller.

REST resources are frequently used to access the functionality of IoT devices, as this lightweight technology is well suited to limited hardware and software resources of the devices. Also, REST servers are used to proxy such access for very limited and non-standard devices [8]. Once the efficiency and simplicity of implementation rather than security is of primary concern, REST resources seem to be much better base than classical SOAP-based SOA services [8].

Although REST is a very useful proposition for the implementation of IoT-based framework, this technology must be substantially extended to meet the requirements presented in Section II. Thus, in Section VI we propose such extensions as a generic REST-based architecture for the Internet-of-Things environment.

V. OSGi-BASED MANAGEMENT

The IoT systems are characterized by their dynamic nature, as new devices appear, change their status including the availability, disappear or even suddenly break down. The OSGi framework with its support for dynamic modules seems to be the framework of the first choice to build on the IoT management system. Additionally, the OSGi service registry enables discovery of IoT specific features. Moreover, event-based communication allows ignoring the implementation details of services and substantially reducing the need for the standardization of interfaces. However, the IoT relies on network communication, which is not supported by pure OSGi specification. Another trait of IoT, which is difficult to be realized in OSGi framework, is a support for distributed vendors of IoT services. The next specific of IoT system is temporal unavailability of physical devices, which may be busy, broke down or temporary unplugged.

The requirement of choosing "the best" IoT service cannot be fully realized in pure OSGi framework because such choice is limited to fixed service ranking and age (e.g., a moment of first registration of the service). The last but not least important feature of OSGi-based IoT management system is the possibility to manage the devices manually by checking and modifying their status. The above described limitations were the main reasons to propose by us some extensions for OSGi. The architecture of extended OSGi framework for IoT management system is presented in Fig. 1

First, to deal with network communication among IoT services, the OSGi platform is extended by installing Remote Services for OSGi (R-OSGi) bundle developed in ETH Zürich [15]. R-OSGi provides dynamic proxy generation for remote invocation of services and register remote services, discovered in distributed registry, in local OSGi service registry.

Next, to handle modules provided by device vendors, the universal semantic (device descriptions) must be proposed, but this problem is out of the scope of the paper. Here, we

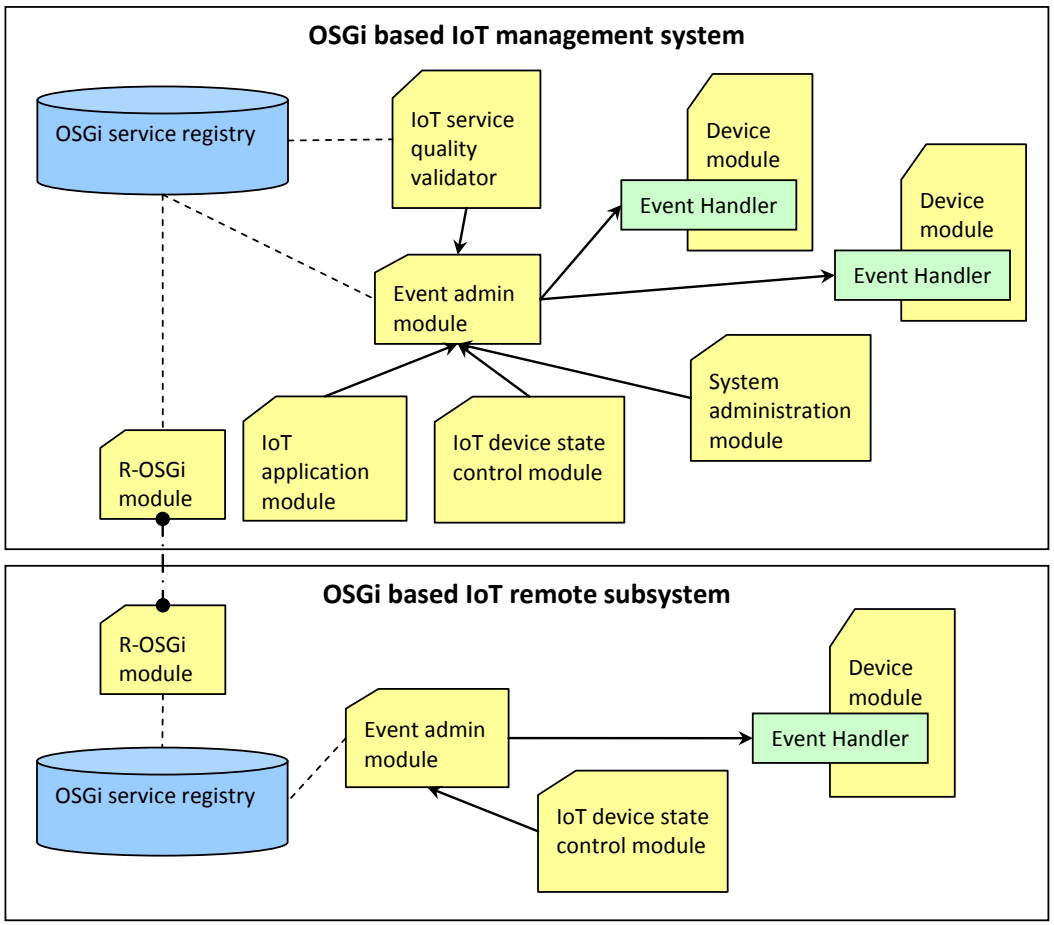

Fig. 1. OSGi based architecture for IoT environment

only point out the fact that OSGi enables specifying service metadata during service registration. The semantic information may be included in service metadata in the form of `java.util.Dictionary` object, containing OSGi event topics or additional properties.

In order to reflect the status of the IoT device related with given OSGi service, the metadata of the service may be dynamically changed, e.g., to restrict an access to unavailable devices. However, to continuously monitor the device status, such a function should be provided by the provider of the device module (corresponding OSGi bundle). IoT device-state control module, extending basic OSGi platform, should be able to send auditing events and get responses from related dependent services interested in the notifications of the device status.

The IoT management system should enable the user to invoke the most suitable service. As the capabilities of OSGi in this aspect are limited, the dedicated module needs to be provided. We propose to implement IoT statistics and validation module, which enable to validate the properties of the service and provide proper statistics on service performance. Such module should utilize the aspect oriented programming techniques to measure and store service properties such as performance time, moment of last invocation, number of invocations, number of generated exceptions (errors), etc.

For administration purposes, the OSGi console may be extended by the methods allowing manual management of available devices. The system administration module is responsible for the discovery of devices based on service metadata and allowing checking or changing the state of these devices.

We may list both the advantages as well as the disadvantages of the OSGi-based architecture, which are presented below.

The main disadvantages of OSGi based architecture are the following:

- Java dependent modules – the OSGi framework was developed for Java platform and in conclusion all module providers need to implement bundles in Java. As IoT is heterogeneous from its nature, the support for many programming languages should also be possible;
- limited built-in support for distributed services – OSGi was design to foster development of modular software running on one device (host), without taking into account distributed environments. The R-OSGi initiative mostly solves the problem, however the control over remote

services is limited and there is no built-in control over distributed bundles;
- lack of shared semantics – there is a need to propose standard semantic to enable unequivocal communication among system and the services – especially the ones provided by external entities;
- limited capabilities of service registry – the registry may provide additional information on services only in the form of `java.util.Dictionary` metadata. This solution is troublesome when extending the registry features to provide e.g., service quality information or device state control.

Despite of noticeable disadvantages, the OSGi framework is still prospective for building IoT management systems, as it has numerous advantages, which are listed below:
- support for dynamic modules – the bundle management allows to change the system capabilities in runtime. This feature is very useful, as in IoT system devices may be dynamically (dis)connected with the system at any time;
- event based communication – this trait of OSGi framework enables to separate the service invocation from its implementation. The OSGi events may carry orders in natural language, which are interpreted by event handlers, instead of directly invoking methods. What is more, requester does not need to know the location of a service, as event administrator sends events to all event handling services;
- code sharing and encapsulation – the OSGi bundles may share their code with one another, which prevents memory overhead. Additionally, each bundle directly specifies the code to share, thus enabling additional encapsulation by separating service interfaces from their implementations;
- compound services – the OSGi framework allows to compose compound services based on simple ones. The orchestration may be done just for calling specific feature, and the OSGi framework will provide the proper service and, after careful event handling implementation, even the most suitable for given request;
- providing interface for manual administration – OSGi framework provide possibilities to extend the console commands by a set of user-defined ones. This capability forms an easy way to implement IoT device administration interface.

VI. REST-BASED MANAGEMENT

The key requirement for the IoT system is an efficient method for the selection of device(s) for the incoming request in a given context. To this goal, the system must (1) know the devices' possibilities and characteristics, including their geo-locations (range of impact), (2) be able to search for the optimum device(s) to fulfill given request, and (3) activate the just-searched device in order to provide certain functionality. As already mentioned, these are basic tasks for a centralized catalogue. However, such catalogue is not a part of REST technology. Thus, we propose a uniform REST-based architecture with a dedicated catalogue (also REST-based) as the base for IoT system (Fig. 2). The architecture is based on using administrational extensions of REST servers to be observed as REST resources by the catalogue.

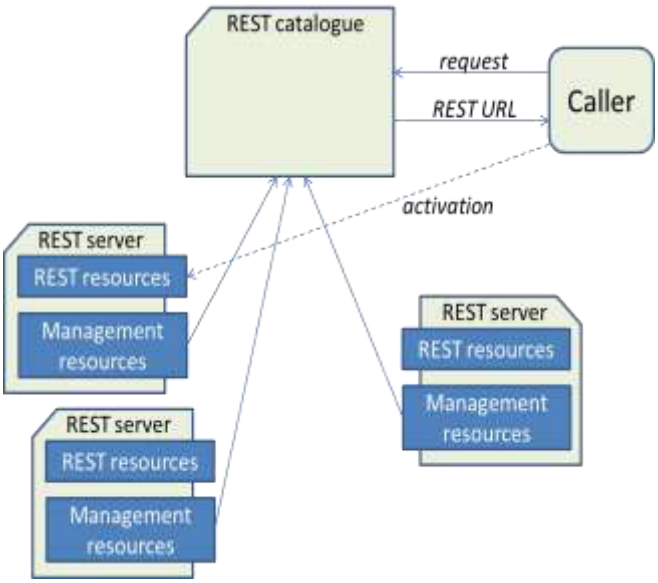

Fig. 2. REST-based architecture for IoT environment

The architecture is based on two principles: extending REST servers by some management resources, and providing mapping of semantically-expressed requests to URLs of REST resources.

To implement the first goal, we assume that each REST server is equipped with certain (predefined) resources to control and supervise the server, including:
- "management" resource to provide an access to some basic commands such as quit/suspend/resume/restart,
- "statistics" resource to provide some information about server usage (both divided to particular resources related with this server, as well as the server as the whole), for example: average service timings, number of requests served, number and description of invocation errors, etc.; this resource also provides an information about the real device (if any), connected to the resource and using this resource as its own proxy; in such way, the catalogue may be informed weather the device is accessible or temporary unavailable,
- "functionality" resource to provide some knowledge for the functions possibly served by this server (namely, by its resources); such function descriptions are defined according to common semantics for the whole system (c.f., interpretation of the request below).

Each REST server registers itself in the catalogue, providing its own URL locator. The locator (more precisely

– the resources mentioned above) may be periodically inspected by the catalogue to collect up-to-date information about server state. This information is used to search for ready-to-use devices (c.f., a description of request serving later on).

Once the internal state of the REST resource is changed, e.g., according to respective change of the state of real device connected to this resource, the server re-registers to the catalogue with the updated information.

Imagine one wants to activate certain function of the system. To this goal, he/she must address the catalogue with the semantic description of the desired action. This description is compared with the possibilities of the devices (however, only those declaring their state as currently accessible), and a device is chosen to meet the criteria. The caller obtains the locator of the resource linked with desired activity/device, to directly address respective REST service and, indirectly, the device. Note that the called URL was not known by the caller in advance, as it was (possibly dynamically) generated and sent by the catalogue. Once the situation is changed, some other resource/device may be activated according to the same request. Note also that the semantics of the URL locator of the resource to activate is not known to the caller, thus the details of the activation may be hidden towards the users of devices' functionality. This approach greatly improves portability of the system usage, on condition the semantics of the requests is common for all the callers and catalogues.

If given request is to be possibly served by several devices, the catalogue may choose one device based on context and statistical information collected from the management/statistical resources of the corresponding REST service. For example, one may address the less-overload resource, last-activated or most-unused device, the one with the shortest response time, etc.

We may also propose, instead of accessing a single resource for a single request, activating a set of resources – i.e., an orchestration of resources. To this goal, a mechanism is needed to map the request semantics to some program code, in turn responsible for the pipelining of the resources. The final result is provided as if all the activated resources are a single resource, thus the whole orchestration mechanism is transparent to the caller.

In most of the applications, connecting all the devices to a single host is not possible, due to (1) limited number of external connectors (such as USB), and (2) natural need for the distribution of the devices across a wider area. Thus, it is desirable to distribute not only the devices and hosts (proxies), but also the parts of the controlling framework. So far we assumed that there is only one central point for the control of the whole network. However, due to unrestricted distribution of REST resources it is possible to part this centralized point to a hierarchy of interconnected sub-parts, each one providing the control over certain network part (Fig. 3). To keep the control on the network as the whole, we propose to connect the sub-controllers as a graph and to span the controlling in the same manner as it is used to synchronize the resources of any peer-to-peer (P2P) network, with arbitrary restricted nesting level.

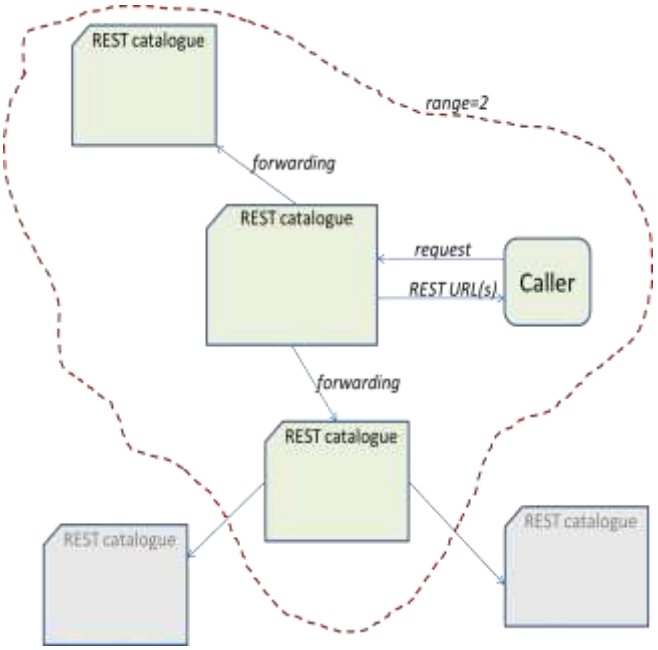

Fig. 3. Hierarchy of REST-based distributed management services

A device may choose any of the sub-controllers to register with. Then, this device is manageable locally by this sub-controlled in the direct way described above. Similar, all the local requests served by this sub-controller towards all its devices are processed locally. However, if there is a need to access remote (from the point of view of this controller) devices, then the sub-controller forwards the request to all its neighborhoods. In turn, if a neighbor is not able to process the request, forwards it to all its neighbors except the one which initiate the request, and so on. For each forwarding step, nesting level of the request (a range) is increased. While this level reaches certain value, the forwarding is stopped. Returning to Fig. 3, the request marked as "level=2" is forwarded only to three sub-controllers, while two "far" ones remain untouched. The stopping value is declared for each sub-controller by its administrator, and for the request by the initiator of this request – each time smaller of these two values is taken into consideration.

All the responses are collected by the forwarder of the request, and, if the request level is still greater than one, send as a common response to the caller. Finally, the originating node collects all the responses of all its neighbors, acting as a "global" response to the initial request.

To limit the possible cycles in the forwarding of the request, each request is identified, and the past-request identifiers are collected for some time in each of the sub-controllers. Once a request is coming already served in the past, this call is disregarded and no more forwarded. Thus,

even if the graph of interconnected sub-controllers contains cycles, these cycles are detected and never block the system.

In the same way we may obtain some global information about the network (statistics, information for certain-device of function availability, etc.), more precisely – about the local neighborhood ("no longer than N connections from the selected node"). More global is the request, more we must wait for the response, similar to typical P2P behavior.

As a typical IoT environment usually covers rather small geographical area (such as a room, a building or a public place – a market, a shop, a museum, etc.) – by restricting the level of spreading the requests to reasonable value one also limits the overall network traffic to the reasonable level, and the response delay is counting in parts of the second. We may also imagine restricting the bigger levels to those with special access rights, such as system administrators – for most of the requests, these will be addressed to local devices (level equal to 1). Then, both the possible delays and increased network traffic are not a sharp problem.

We may enumerate both the advantages and disadvantages of the proposed architecture, these are presented below.

The disadvantages are mainly related with two global observations – independence of the program code for the REST servers, and the need for shared semantics:

- there is no shared code even if identical parts of the program code (i.e., the libraries) are used by several REST servers. This feature results in large memory usage and may be relaxed by some shared-code techniques such as Dynamic Library Linking DLL;
- all the resources must know in advance the address of the catalogue, to register with. This restriction may be relaxed with non-standard usage of DHCP broadcasting;
- additional network traffic is observed to monitor device availability at real-time. However, with reasonable polling interval (for most of the system such timing as 5-10 seconds is completely enough) this restriction may be bypassed;
- one must provide strict definition of the semantics of requests and device activities (functions), shared for the whole system. This restriction is related with a need for the strict format of URL locators for additional REST resources (management/statistics). However, as this traffic is not observed by the end-users, this is a problem only for system designers;
- the activations of devices' functions are based on URL locators of the corresponding REST resources. Thus, only limited parameterization of such calls is possible – all the details must be coded and thus somehow hidden as URL locator parts. However, this is also mainly the problem of system designers, as end-users have no knowledge about the semantics of the device activations. Moreover, sometimes such information should be intentionally hidden for the end-users to limit the direct access to the devices (i.e., the access not controlled by the catalogue).

The advantages outweigh the above-presented restrictions and problems:

- it is a uniform approach – the whole traffic is realized as REST-compliant calls, which strongly facilitates the implementation of the system;
- small resources and servers dominate across the system, thus the consumption of computer resources such as CPU time is reasonably small. Our experiments showed that even hundreds of such servers on a single PC is not a problem, as a single resource typically consumes less than1% of computer resources;
- there is no problem with the synchronization of some resources "shared" among many REST servers, for example real devices, communication ports, etc. Such synchronization is needed only for a single REST server, however, as this server is programmed as a single entity and probably by a single programmer or small group of designers, such synchronization is easy to achieve;
- all the system parts (resources, devices) may be arbitrary distributed even in a local- or even wide-area network – on condition the distributed servers know and may access the catalogue host;
- REST servers may operate even with very limited hardware, also built-in to the networked devices,
- there is theoretically unlimited possibility of the orchestration of devices – providing "virtual devices" acting as real ones, however, possibly much more complex and powerful;
- the catalogue represents up-to-date information for device availability – continuous monitoring is undertaken not only for device state, but also some statistics for its usage;
- single- and group-based management for devices and their corresponding resources is possibly achieved, including server start, quit, restart, suspending/resuming, also GUI-based individual administration.

VII. CONCLUSIONS

As may be drawn from comparison of OSGi and REST based systems presented in Table I and discussed above, both approaches – OSGi-based and REST catalogue provide similar functionality and may be applied to implement an Internet-of-Things middleware. The amount of work for both approaches is similar – substantial extensions are needed to adapt the environment to the specific requirements of IoT applications. However, OSGi-based approach is better suited for sensor networks, i.e., the applications covering homogeneous devices and fixed, predefined system functionality, while the REST-based framework is more useful in ad-hoc, dynamic environment achieving heterogeneous devices and services. For the first, if we deal with shared device functionality and program code, we substantially limit memory usage. For the latter, we may

TABLE I.
COMPARING OSGI AND REST-BASED APPROACHES

IoT requirement	OSGi-based management	REST-based management
registering the device/function	X	X
providing individual proxy for interaction/communication mapping – starting, stopping, and suspending/resuming the service	X	X
monitoring real-device state and providing information about device accessibility	-	X
searching for the device(s) to serve given request	X	X
if a request is to be possibly served by several devices, choosing one device based on context and statistical information collected during past activations	X	X
orchestrating device functions taking into account local context (complex or virtual devices)	-	X
administrating device parameters (individual GUI for each device)	-	X
sharing basic libraries (functionality) for similar devices	X	-
adjusting scope of search for a device in the distributed access, managing a hierarchy of proxy servers	-	X

more easily provide distribution of proxies and devices as well as device/service orchestration, also in ad-hoc mode and based on some statistical information about past activations. If the amount of the shared code is low, due to the heterogeneity of devices/proxies, we do not observe the possible savings resulting from shared code, while still having the possibility of centralized management of the system as the whole as well as particular devices/resources.

REFERENCES

[1] D. L. Brock: The Electronic Product Code (EPC) A Naming Scheme for Physical Objects, Auto-ID Center, http://www.autoidlabs.org/uploads/media/MIT-AUTOID-WH-002.pdf,

[2] S. Haller, S. Karnouskos ,Ch. Schroth, "The Internet of Things in an Enterprise Context", in: Future Internet – FIS 2008 LNCS, vol. 5468, J. Domingue, D. Fensel, P. Traverso, Eds. Berlin Heidelberg: Springer-Verlag, 2009, pp. 14-28, doi: 10.1007/978-3-642-00985-3_2

[3] D. Wilusz, J. Rykowski, "The Architecture of Coupon-based, Semi-off-line, Anonymous Micropayment System for Internet of Things", in: Technological Innovation for the Internet of Things IFIP AICT, vol 394, L. M. Camarinha-Matos, S. Tomic, P. Graça, Eds. Berlin Heidelberg: Springer-Verlag, 2013, pp. 125-132, doi: 10.1007/978-3-642-37291-9_14

[4] D. Wilusz, J. Flotyński, M. Sielicka, "Supporting experimentation in a food research laboratory with the Internet of Things", in: PhD Interdisciplinary Journal no. 3/2013, Gdańsk: Gdańsk University of Technology, 2013, pp. 113-119.

[5] D. Waltenegus, Ch. Poellabauer, Fundamentals of wireless sensor networks: theory and practice, John Wiley & Sons, 2010

[6] M. Weiser, "The computer for the 21st century", in: Scientific American vol. 265, 1991, pp. 94-104, doi: 10.1038/scientificamerican0991-94

[7] R. T. Fielding, R. N. Taylor, "Principled design of the modern Web architecture", in: ACM Transactions on Internet Technology vol. 2 issue 2, New York: ACM, 2002, pp. 115-150, doi: 10.1145/514183.514185

[8] J. Rykowski , P. Hanicki, M. Stawniak, "Ontology Scripting Language to Represent and Interpret Conglomerates of IoT Devices Accessed by SOA Services", in: SOA Infrastructure Tools: Concepts and Methods, S. Ambroszkiewicz, J. Brzeziński, W. Cellary, A. Grzech, K. Zieliński, Eds. Poznań: Wydawnictwa Uniwersytetu Ekonomicznego w Poznaniu 2010, pp. 235-262

[9] D. Guinard, I. Ion, S. Mayer, "In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers' Perspective", in: Mobile and Ubiquitous Systems: Computing, Networking, and Services, A. Puiatti, T. Gu, Eds. Berlin Heidelberg: Springer-Verlag, 2012, pp. 326-337, doi: 10.1007/978-3-642-30973-1_32

[10] OSGi Alliance Technology / HomePage http://www.osgi.org/Technology/HomePage

[11] J. Flotyński, K. Krysztofiak, D. Wilusz, "Building Modular Middlewares for the Internet of Things with OSGi", in: The Future Internet LNCS, vol. 7858, A. Galis, A. Gavras, Eds. Berlin Heidelberg: Springer-Verlag, 2013, pp. 200-213, doi: 10.1007/978-3-642-38082-2_17

[12] R. S. Hall, K. Paulus, S. McCulloch, D. Savade, OSGI in Action: Creating Modular Applications in Java, Greenwich: Manning Publications, 2011

[13] OSGi Alliance *OSGi™ Service Platform Release 4 Version 4.2* http://www.osgi.org/javadoc/r4v42/

[14] A. de Castro Alves, OSGi in Depth, Greenwich: Manning, 2011

[15] J. S. Rellermayer, G. Alonso, T. Roscoe, "R-Osgi: Distributed Applications through Software Modularization", in: Middleware 2007 LNCS, vol. 4834, R. Cerqueira, R. H. Campbell, Eds. Berlin Heidelberg: Springer-Verlag, 2007, pp. 1-20, doi: 10.1007/978-3-540-76778-7_1