Building Modular Middlewares for the Internet of Things with OSGi

Jakub Flotyński, Kamil Krysztofiak, and Daniel Wilusz

Department of Information Technology,
Poznań University of Economics
{flotynski,krysztofiak,wilusz}@kti.ue.poznan.pl

Abstract. The paper addresses an analysis of OSGi in the context of building modular middlewares for the Internet of Things. The Internet of Things (IoT) is an emerging approach to development of intelligent infrastructures combining various devices through the network. OSGi is a framework providing a number of specific mechanisms intended for building modular, fine-grained and loosely-coupled Java applications. Although a number of works have been devoted to OSGi, and several OSGi-based middlewares have been designed for the IoT, they do not thoroughly utilize mechanisms of OSGi. In this paper rich OSGi functions are analysed in terms of development middlewares for the IoT. An example implementation of the system illustrates the presented considerations.

Keywords: Internet of Things, modular, multi-layer, event-based, middleware, OSGi.

1 Introduction

Future Internet is a rapidly evolving phenomena, which significantly influence the business and the society. It incorporates two modern application areas of the networking: the Internet of Things (IoT), and the Service-Oriented Architecture (SOA) [1].

The IoT is a continuously developing concept introduced by MIT Auto-ID Center [2] as an intelligent infrastructure linking objects, information and people via the computer network. The IoT allows for universal coordination of physical resources through remote monitoring and control by humans and machines [3]. Nowadays, the role of the IoT is no more limited only to electronic identification of objects but it is perceived as a way to fill the gap between objects in the real world and their representations in information systems. According to [4] [5], the components of the IoT are participants of business processes seamlessly integrated through independent services.

That is why powerful middleware solutions are required to integrate heterogeneous devices and dynamic services for building complex systems for the IoT. The middleware is a system connecting different components and/or applications [6] in order to build complex layered applications [5]. The objective of the middleware is to hide details of different technologies to avoid dealing with technical, low-level issues [5].

The appropriate middleware solution for the IoT should enable system modularity to permit flexible management of devices and services. To facilitate development of modular, fine-grained and loosely-coupled applications, a few frameworks for Java and

A. Galis and A. Gavras (Eds.): FIA 2013, LNCS 7858, pp. 200–213, 2013.

.NET environments have been designed (e.g., OSGi [7], Jini [8] or MEF [9]). Among these platforms, OSGi (originally Open Services Gateway initiative) is currently the most popular one, supporting a number of specific mechanisms useful for building both desktop and IoT applications.

Although a number of tutorials have been devoted to OSGi, and several middlewares have been designed for the IoT, these works focus mainly on the proposed systems themselves and do not explain how to thoroughly utilize the diversity of OSGi features for building middlewares for the IoT. The main contribution of this paper is an original analysis of OSGi for building applications for the IoT. The considerations are illustrated by an example implementation of a middleware. The system enables integration and management of devices and services in IoT environments with a strong focus on the use of the rich OSGi functionality. Systems built according to the hints given in this paper can be used within secure houses, smart shops, cars, etc. However, it is not possible to describe all mechanisms contained in OSGi very exhaustively. They are explained in detail in the OSGi documentation [7] [16] [17]. In this paper, we explain some of them—the functions especially useful in the context of building systems for the IoT. It has not been our purpose to compare the presented example implementation to other systems proposed in previous works, e.g., in terms of structure or efficiency.

The remainder of the paper is structured as follows. In Section 2, the state of the art in the domain of OSGi-based middlewares is briefly described. Section 3 presents the functionality of the OSGi framework that may be useful during development of IoT systems. Section 4 addresses the requirements for the middleware for the IoT. In Sections 5, 6 and 7, the idea and an example implementation of the system are presented. Finally, Section 8 concludes the paper and indicates possible directions of future works.

2 Middlewares for the Internet of Things

Several middlewares for the IoT have been designed and implemented [10]. In general, these systems often address integration of RFID devices [11] [12] and sensor networks [13], and provide integrated software development environments [14].

In [14] a cooperative web framework integrating Jini into an OSGi-based open home gateway has been presented. The framework connects embedded mobile devices in heterogeneous network and enables discovery and management. Another solution integrating heterogeneous devices has been presented in [13]. The authors utilize the OSGi framework to provide a layer in high level of abstraction for communication with concentrators operating wireless sensors. The Arcada project [12] provides a middleware for management of RFID devices. As a result, a more advanced system dealing with the three characteristics of the IoT (heterogeneity, dynamicity and evolution) has been built upon the previous version [15]. This middleware utilizes OSGi to provide modularity and dynamicity and permits extending the system with hot deployment of components.

3 Selected Mechanisms of OSGi

OSGi [7] [16] [17] is a modular framework for Java applications facilitating software componentization. OSGi supports numerous mechanisms useful for building applications for the IoT, which are described in this section.

3.1 Modularity

OSGi-based Java applications are built from independent and separate modules referred to as bundles. The bundle is a Java project containing packages and classes that may be shared with other bundles as specified in a separated file assigned to the bundle. Bundles have a lifecycle encompassing the following states:

- uninstalled (not included in the application),
- installed (included without satisfied dependencies),
- resolved (stopped but ready to be started),
- active (started),
- transitive states: starting and stopping.

Starting a set of bundles is performed with regard to the dependencies among them but the lifecycles of the particular bundles may be managed independently. This feature is especially useful when some modules must be exchanged without shutting down the entire system (e.g., when new versions of modules are introduced). Once the state of a bundle changes, other bundles may be notified of this fact using the `BundleListener` service handling `BundleEvents`.

3.2 Runtime Configurations

The OSGi `runtime configuration` specifies a set of bundles to be started. Each bundle as well as each `configuration` is assigned an integer specifying a `start level`. Bundles are run in order from the lowest start level to the highest one. While starting the `configuration`, only bundles with `start levels` lower or equal than the `configuration start level` are launched. Such approach facilitates development of applications covering a wide range of functions. Consecutive `configurations` may be super-sets of bundles of their predecessors. `Configurations` with higher start levels can include additional modules implementing extended functionality based on the functionality of bundles contained in `configurations` with lower start levels.

3.3 Configurations of Bundles

Besides configuring the entire application, also individual bundles may be configured. It is performed by the `Configuration Admin Service` that decouples configurations of particular bundles from their source code. Configurations are specified by a set of properties stored, e.g., in files or in a database. Bundles whose configurations are specified by particular properties, implement methods listening for updates of these properties. Such methods properly modify the behaviour of the bundles.

3.4 Console

The OSGi framework has a built-in `console` that can be used to manage bundles and services (e.g., to start and stop them). The `console` is similar to these implemented in widely-used operating systems like Windows and Linux and may be accessed remotely via SSH allowing developers to manage the framework in a flexible way.

3.5 Event-Based Communication

OSGi supports flexible event-based communication among bundles providing one-to-many data exchange in both synchronous (with waiting for handling the event by all interesting modules) and asynchronous modes. OSGi `events` are broadcasted by the sender and may be filtered by their topics, hence delivered only to proper destination bundles. Using `events` in OSGi-based applications excludes the necessity of specifying direct dependencies between bundles, thus disabling cycles. The lack of cross-references implies effective implementing of systems working like the Controller Area Network (CAN) bus [18] in which event recipients do not need to be known in advance.

3.6 Local Services

OSGi `services` are classes implementing Java interfaces. They may be published, discovered and invoked. `Services` are accessed synchronously and locally (from the same machine). Alike message-passing, invoking `services` does not require keeping references to bundles containing them. Using the `service` requires the knowledge of the implemented interface. Customized instances of `services` are created by the `ServiceFactory` and cached in the framework. Bundles interested in particular `services` may be notified of registering, modifying and unregistering them through handling appropriate `ServiceEvents` in the `ServiceListener` (likewise `Bundle Events` handled by the `BundleService`). To obtain the `service` reference for a particular interface, the `context` of a bundle is used as a service repository.

`Services` are described by metadata specified as a set of `properties` (e.g., id, description, vendor). The metadata may be changed at any time. Required values of `properties` may be specified while discovering services using a query language provided by the OSGi framework, e.g., get references only to services from a given vendor, that have been introduced not later than two month ago.

3.7 Web Services

OSGi supports also remote communication between bundles distributed across different platforms. Web services may be implemented using Apache CXF [19] – a reference implementation of the `distribution provider` component of the OSGi Remote Services Specification [17]. Remote OSGi services can be exported by a Java Virtual Machine and imported by another one. Apache CXF supports both SOAP-based and RESTful web services. Obviously, Apache CXF services may be invoked by any web service clients, enabling flexible and easy integration with other systems.

4 Motivations and Requirements for an OSGi-Based Middleware

OSGi, as a platform designed for building scalable dynamic and modular Java-based systems, may be successfully utilized to develop multi-layer middlewares for the IoT. Although, several works on middlewares for the IoT have been conducted, they do not thoroughly explore the functionality of OSGi in the context of the IoT, instead they

mainly focus on the proposed systems – their architectures and functionality. These implementations primarily benefit from OSGi modularity and do not put stress on other rich OSGi features and functions, as well as the way in which the framework can be used for building complex middlewares.

The main contribution of this paper is an analysis how mechanisms of OSGi may be utilized to develop modular, scalable and fine-grained middlewares for the IoT.

The following requirements in terms of modularity, service-oriented architecture and communication between software components and devices have been specified for the proposed middleware to illustrate the robustness of rich OSGi mechanisms described in the previous section.

4.1 Modularity

1. The middleware is fine-grained, based on modules that compose the functionality of devices connected to the environment, e.g., a module responsible for closing a door can first obtain the state of a sensor to check whether the room is abandoned.
2. Complex modules are composed of simpler ones. For instance, close a door and turn on an alarm are the services combined into a complex 'secure the house' service.
3. The modules are loosely-coupled, they can exchange messages in relation one-to-many without keeping direct references, e.g., the service 'secure the house' does not need to know all its sub-modules performing simple actions (like close the door and turn on the alarm) to invoke them.
4. The modules perform actions with regard to the context of the interaction, e.g., the door may be closed only when nobody is inside the room, lights may be switched on only when it is dark outside.
5. Adding, restarting and updating versions of particular modules does not require restarting the entire platform and does not interrupt the work of other modules.
6. Adding new modules to the system that have to cooperate with older modules require neither updating nor restarting the platform, e.g., a new sub-service closing the door of a room should work as a part of the secure the house service regardless of previous modules involved in it.
7. The modules are notified each time when a new module is introduced to the system, e.g., a new thermometer has been added to report the temperature in a room.
8. The system may be started with various configurations of modules. The configurations may be changed dynamically. For instance, first, only modules operating on devices are started; second, interfaces and modules composing the functionality of the devices are started as well.
9. The configuration of particular modules should be decoupled from their source code and managed in a flexible way, e.g., to indicate that a particular number of bulbs should be switched on by a module.

4.2 Service-Oriented Architecture

10. Services are built upon modules that are available for the user through various interfaces, e.g., GUI, web services.

11. The middleware contains a service repository of references to services provided and accessible among the modules. The repository is used for service discovery, e.g., find services responsible for washing dishes.
12. The services are described by a set of properties that may be specified at service discovery, e.g., find services using thermometers that are currently switched on.
13. The middleware may notify the user about some events, e.g., an email is sent always when a temperature in a server room is high.

4.3 Communication with Devices

14. Diverse interfaces of devices incorporated in the middleware should be wrapped with a uniform network interface to enable physical distribution of devices and communication with them in a similar manner, i.e. each device is connected to a proxy hiding its implementation details, e.g., the method of measuring the temperature provided be a thermometer is wrapped with an analogous network service.
15. The interfaces do not limit the functionality of primary device interfaces, at most they change semantics of their invocation, e.g., exchange method invocations with create-read-update-delete (CRUD) services.
16. The devices may be queried by the modules as well as modules may be notified by devices of some events they are interested in. Communication between the modules and the devices is bi-directional, e.g., the thermometer can measure the temperature when invoked by a module, as well as notify interested modules of significant changes on its own.

5 Architecture of the Middleware

Modular Multi-layer Middleware for the Internet of Things (*MOMIT*) has been proposed based on the presented rich OSGi functionality. Functional components of the platform originate from the proposal of S. Bandyopadhyay et al. [10]. In this section, the middleware architecture is presented in detail with regard to the requirements and the OSGi mechanisms depicted in the two previous sections. To provide separation of concerns, the architecture of the system is multi-layer and fine-grained with loosely-coupled components (Fig. 1). It consists of the following layers: the device layer, the business logic layer and the interface layer.

5.1 The Device Layer

It is the bottom layer of the *MOMIT* that incorporates various independent *devices*. In the presented middleware, it is assumed the *devices* communicate with one another through the higher (business logic) layer. In fact, in IoT applications the machine-to-machine (M2M) communication is sometimes a necessary and significant aspect. In such cases the *devices* can communicate directly omitting other system components, but the flexibility of such solution may be limited.

Primarily, the *devices* are accessible through heterogeneous interfaces whose diversity may be a problem while building complex applications that utilize many of the

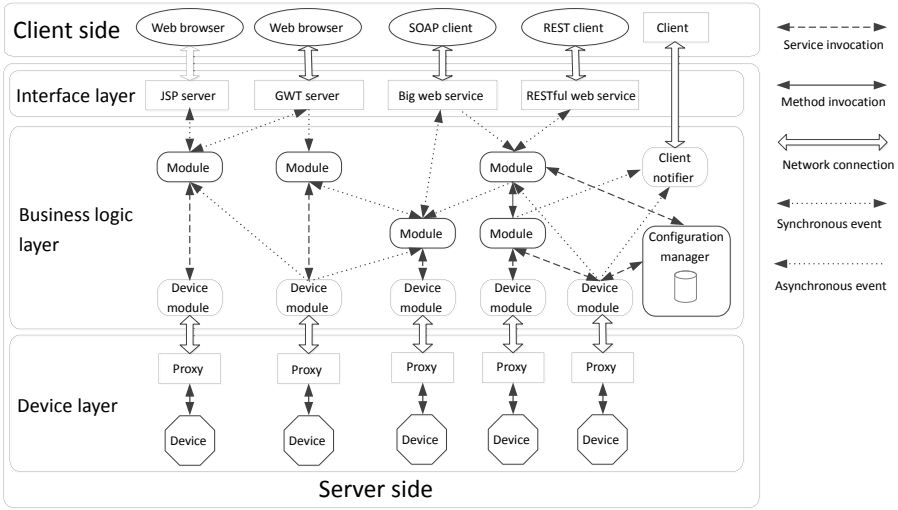

Fig. 1. The *MOMIT* architecture

available devices. To overcome this inconvenience, as well as to enable distribution of the *devices* and making them accessible via the network, each of them is associated with a *proxy*. The *proxies* handle queries from the *business logic layer*, addressed to the *devices*. The *proxies* wrap the *device* interfaces providing a single flexible and uniform interface for all the *devices* within the entire system (Sec. 4, req. 14), e.g., all the *proxies* provide a RESTful interface or all the *proxies* provide a SOAP-based interface. Moreover, the *proxies* do not limit the functionality of the *devices* they wrap. At most, only the semantics of the interfaces can be exchanged (req. 15), e.g., invoking *device* methods is translated to RESTful web service CRUD operations.

For instance, a RESTful web service receives a `HTTP PUT` request specifying the URI of lights in a room and the value `switched on`, and translates it into the method invocation `lights.switchOn(true)`. Communication between a *device* and its *proxy* depends on the specificity of the *device* and a particular application in terms of the software and hardware used, and it is not addressed in the *MOMIT* architecture.

Sometimes, it is desirable to inform other system components when specific conditions occur. In such case the communication is initiated by the *proxy* that has detected an event coming from the associated *device*. To satisfy this requirement, the *proxy* includes a web service client that notifies the *business logic layer* of the event (req. 16).

The *Proxies* may be implemented as individual OSGi `bundles` incorporating network interfaces based on such lightweight platforms as [20] or Apache Tomcat [21].

5.2 The Business Logic Layer

It is the middle layer of the *MOMIT* that utilizes the mechanisms of OSGi very thoroughly. The main purpose of this layer is to combine the functionality of the *devices* into complex services executed with regard to environmental conditions or the context

of the interaction between the client and the middleware. Individual elements of the system are described below.

Modules. Primary entities of the *business logic layer* are modules implemented as individual OSGi bundles. Two types: *device* and *service* modules are distinguished. Both types are built according to the service-oriented approach but they are used for different purposes and in different ways.

Device Modules. The device module serves as the *proxy* – it provides a uniform network interface for the *device* (req. 14). Alike *proxies*, *device modules* do not limit the functionality of the *devices* (req. 15). Every *device module* is associated with exactly one *proxy* and thus with a single *device*.

The *device module* is used to query the *device* via its *proxy*, as well as to allow it to notify other modules in the *business logic layer* of events coming from the associated *device*. In the first case, the *device module* invokes the *proxy* via its web service. In the second case, the *device module* provides a web service (req. 16) invoked every time when data transmission is initialized by the *proxy*. After getting a notification, an OSGi `event` is broadcasted, and it may be handled by all interested modules in the *business logic layer*. Modules that are not interested in the `event`, ignore it. Such an approach does not demand that the *Device module* knows all recipients and keeps references to them (req. 3). If a module needs to process an `event`, its developer should be familiar with this and implement its handling.

All the *device modules* of the *MOMIT* issue OSGi `services` wrapping the interface of an associated *proxy*. The `service` is described by a set of `properties` that depict the *device* and its current state (switched on/off) (req. 12). The `services` are created by the `ServiceFactory` and discovered by the `context` bundle.

When a bundle implementing a *device module* changes its state, a `BundleEvent` is generated to inform other modules (req. 7). Alike, a `ServiceEvents` are broadcasted when `services` have been started or stopped. For instance, from this moment *MOMIT* modules can start/stop invoking the `services` to which the `event` is related.

Service Modules. Orchestration of the functions of various *devices* accessible through *device modules* into services (req. 1) is the objective of *service modules*. For instance, a *service module* can use a motion sensor to check whether nobody is inside a room, switch off lights and close the door. Analogously, complex *service modules* may be built upon *device modules*, as well as other *service modules* (req. 2). Implementing complex modules may be performed using OSGi `events` as well as `services` created on demand by the `ServiceFactory` and discovered through the `context` (req. 11).

In some cases, it may be desirable to notify modules of a new device plugged in/out the system (req. 7). For this purpose, modules interested in new devices should be equipped with the `ServiceListener` handling `events` broadcasted when a *Device module* has been started or stopped.

Communication among Modules. Three types of communication among bundles supported in OSGi have been used in the *MOMIT*. In most cases, broadcasting `events` is

used to decouple modules, exempt them from knowing all recipients of the message, exclude cyclic references among bundles, as well as enable one-to-may communication and dynamic adding new modules without updating and restarting the entire system (req. 3, 6). Event-based communication can be utilized in both synchronous and asynchronous modes depending on particular software requirements.

The communication between *service modules* and *device modules* is synchronous and based on OSGi `services`. As described earlier, the `service` is issued by the *device module* and described by `metadata`. Such an approach enables flexible discovery without specified references to other modules. Hence the lack of cycles among bundles.

In some cases, direct method invocations may be used, in particular when the communication between two modules should be effective and synchronous. As this kind of communication requires direct references between the bundles, the destination module should be utilized by few other modules to avoid cycles of dependencies.

Context of the Service Execution. All actions are performed by the *MOMIT* in a particular context determined by variables distributed among the modules, retrieved from devices (e.g., temperature, battery level, etc.), as well as provided by people (e.g., activity results, data obtained from devices not connected to the system). Each variable is a pair *[URI, value]*. Modules connected to particular devices store variables obtained from them on their own, whereas variables coming from other sources may be loaded into modules designed especially for this purpose (e.g., equipped with a database). While storing the context depends on particular modules, accessing it is unified for the entire middleware. Execution of a service that demands a value from the context is preceded by broadcasting a request `event` by the module implementing the service. The `event` contains the *URI* of the required variable. It is handled by modules storing the variable and ignored by others. It is a concern of the software developer to ensure consistency of *URIs* and devices as well as to ensure unambiguous identification of all the variables. After getting a response, the service is performed with regard to the context (req. 4).

Return Channel. In some special cases it is necessary to inform the system administrator about some emergencies, e.g., a tap cannot be turned off or a fridge does not freeze due to a fault. It is permitted by a return channel implemented as a *client notifier* which is a *service module* handling `events` critical for the system reliability, and informing the user if necessary. It is required to implement a web service or to provide an email account on the client side to enable such notifications (req. 13).

5.3 The Interface Layer

It is the top layer of the *MOMIT* providing a multimodal-interface to allow users to utilize system functionality using various clients, e.g., web browser, desktop and web service clients (req. 10). The interfaces encompasses GUI, SOAP-based as well as RESTful web services, but the platform could be also extended with other interfaces such as CORBA or Java RMI. Each interface is implemented within an individual OSGi bundle, thus the interfaces may be added, updated and removed independently without affecting the work of other parts of the system.

All above interfaces enable the client-server interaction. For applications requiring uninterrupted data exchange (e.g., real time streaming), new clients and interfaces could be developed (e.g., based directly on TCP/IP).

6 Management of the Middleware

Management of the middleware includes the three aspects described below.

6.1 Lifecycles of Modules

Management of module lifecycles is performed using the OSGi `console`. It enables adding, restarting and updating modules without restarting the whole middleware (req. 5). Commands related to the lifecycle of bundles may be performed also via Secure Shell enabling remote execution of operating system commands.

6.2 Runtime Configurations

Runtime configurations encompassing different modules (OSGi bundles) may be created depending on requirements for a particular application of the Internet of Thing. Consecutive `configurations` are super-sets of their predecessors and may implement increasingly complex applications (req. 8). For instance, modules with higher start levels (started by higher configurations) implement complex functionality based on modules with lower start levels, which are run in advance. Starting a `configuration` with a given `start level` is performed using the OSGi `console`.

6.3 Configurations of Modules

Configurations of individual modules are managed by the `Configuration Admin Service`. The service is issued by a *configuration manager* service module implemented in the *business logic layer* (req. 9). The *configuration manager* stores configurations in a database and notifies other bundles on updates of the properties. The properties may be changed by using a separate application accessing a database. Such approach enables flexible and centralized management of the *MOMIT* modules.

7 Example Implementation of the Middleware

An example of the *MOMIT* system has been implemented to illustrate the presented idea of the middleware based on the rich OSGi functionality. The system is used at the Departament of Information Technology at the Poznan University of Economics. It is intended mainly for educational purposes and during Internet of Things course. The example implementation is depicted in Figure 2. Below, the system layers are described in detail. The *business logic layer* is comprised of the following modules.

1. `Alarm module` is a *device module* – that composes the functionality of an `alarm switch` through an OSGi `service`. Methods provided by the `alarm switch` are accessible through a RESTful web service issued by an `alarm proxy`. All RESTful web services implemented in the *device layer* of the presented example are based on Restlet [20]. The RESTful web service translates requests from the `alarm module` into the invocations of the `alarm.switchOn(true)` and `alarm.switchOn(false)` methods provided by the `alarm switch`.
2. `Alarm module v2` is an enriched version of the `alarm module` that extends its functionality taking into account whether somebody is inside the room. Two versions of the `alarm module` have been introduced to present the possibility of automatic selection of one of them provided in OSGi.
3. `Door module` is a `device module` that composes the functionality of a `door locker` through an OSGi `service`. Methods provided by the `door locker` are accessible through the RESTful web service of a `door proxy`. The RESTful web service translates requests from the `door module` into the invocations of the `door.open(true)` and `door.open(false)` methods provided by the `door locker`.
4. `Light module` is a *device module* that composes the functionality of a `light switch` through an OSGi `service`. Methods provided by the `light switch` are accessible through the RESTful web service of a `light proxy`. The RESTful web service translates requests from the `light module` into the invocations of the `lights.switchOn(true)` or `lights.switchOn(false)` methods provided by the `light switch`.

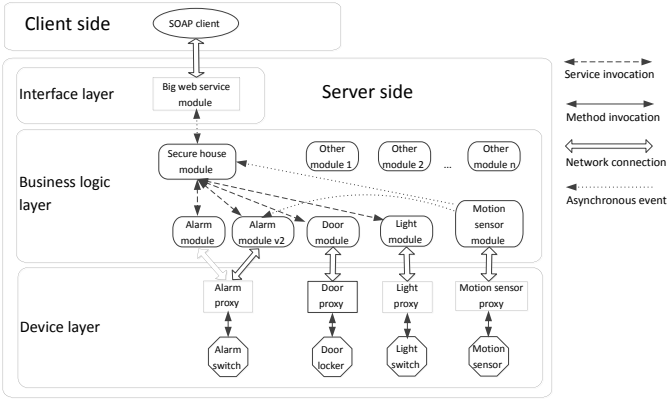

Fig. 2. An example implementation of the *MOMIT*

5. `Motion sensor module` is a *device module* that composes the functionality of a `motion sensor` through an OSGi `service`. Methods provided by the `motion sensor` are accessible through the RESTful web service of a `motion sensor proxy`. The RESTful web service translates requests from the `motion sensor module` into the invocations of the `motionSensor.check()` method provided by the `motion sensor`. The `motion sensor module` can be notified by the `motion`

sensor that somebody is inside the room. To enable such interaction, the motion proxy includes a web service client. After receiving such notification, an OSGi asynchronous event is broadcasted in the *business logic layer*. The alarm module v2 handles the event and processes it by suspending the offered OSGi service. The second module which handle the event is described below.

6. Secure the house module is a *service module* that invokes OSGi services provided by the the following device modules: the alarm module, the alarm module v2, the light module and the door module. To find an appropriate OSGI service issued by an appropriate device, the secure the house module specifies service properties while discovering, e.g., to find OSGi services responsible for switching the alarm which are currently available in the system. In addition, the secure the house module handles the OSGi asynchronous event when somebody is inside the room and invokes the OSGi service provided by alarm module v2 instead of the alarm module. The ServiceListener has been implemented to receive notifications of starting/stopping device modules.

The *interface layer* of the presented example contains a SOAP web service module that is integrated into an OSGi bundle using Apache CXF. The SOAP web service module provides an interface to allow users to call the service of the secure the house module using a web service client. After getting a user request, the SOAP web service module broadcasts an OSGi event which is handled only by the secure the house module. The runtime configuration of the example consists of the following start levels:

1. Alarm module, alarm module v2, light module: first, all the device modules and OSGi services issued by them are registered for the further usage.
2. Secure the house module: second, the service module and OSGi services issued by it are registered for the further usage for the SOAP web service module.
3. Motion sensor module: third, this module is registered for the further usage to optionally send an OSGI event.
4. Big service module: the module enabling an interface to the client is registered.

The proposed example implementation is summarized in Table 1 covering the system requirements satisfied by the considered OSGi mechanisms.

Table 1. Summary of the system requirements satisfied by the OSGi mechanisms

OSGi mechanisms\System requirements	**Modularity**	**Service-oriented architecture**	**Communications with devices**
Modularity	x	x	x
Runtime configurations	x		
Configurations of bundles	x		
Console	x		
Event-based communication	x	x	x
Local services	x	x	x
Web services		x	x

8 Conclusions and Future Works

In this paper the Modular Multi-layer Middleware for the Internet of Things has been presented. Providing scalable, modular, multi-layer middleware is crucial for development of powerful applications for the IoT. OSGi, that is a framework for building modular desktop Java applications, may be successfully used also in the area of the IoT providing a number of specific mechanisms. Although various middlewares have been proposed, they do not utilize these mechanisms very extensively. The main contribution of this paper is the analysis of the OSGi functionality in terms of building scalable, modular and fine-grained middlewares for the IoT.

Future works include the following aspects. First, the *MOMIT* will be secured through introducing access control both for users and modules of the system. In the first case, the middleware could be extended with a repository of user privileges. In the second case, the event-based communication, in which both senders and receivers are not known in advance, could be secured to protect modules against their malicious counterparts [22], i.e. trusted receivers should be protected against untrusted content sent by a malicious module, content sent by a trusted module should be protected against access of untrusted receivers. Second, the functionality of the devices in the system could be specified by an ontology to enable advanced querying device proxies by the system modules. Furthermore, modules should be notified of updates of some service properties (e.g., a service has been modified to switch off different number of bulbs). Finally, a repository of context properties can be introduced to be queried in order to obtain the value of a particular context property or a reference to this property. In some cases (e.g., frequently read variables), it is eligible to inquire only proper modules to decrease the system load.

References

1. Wilusz, D.: Privacy threats in the Future Internet and the ways of their limitation. In: Kształcenie w zakresie Internetu Rzeczy, Uniw. im. Adama Mickiewicza w Poz, pp. 84–103 (2011); in Polish: Zagrożenia dla prywatności w Internecie Przyszłości i możliwości jej ochrony
2. Ashton, K.: That 'Internet of Things' Thing. RFID Journal (July 22, 2009), http://www.rfidjournal.com/article/view/4986
3. Brock, D.L.: The Elect. Product Code (EPC). A Naming Scheme for Phys. Obj., Auto-ID Center, http://www.autoidlabs.org/uploads/media/MIT-AUTOID-WH-002.pdf (retr. June 8, 2012)
4. Haller, S., Karnouskos, S., Schroth, C.: The Internet of Things in an Enterprise Context. In: Domingue, J., Fensel, D., Traverso, P. (eds.) FIS 2008. LNCS, vol. 5468, pp. 14–28. Springer, Heidelberg (2009)
5. Atzori, L., Iera, A., Morabito, G.: The Internet of Things: A survey. In: Computer Networks, vol. 54, pp. 2787–2805. Elsevier (2010)

6. Eisenhauer, M., Rosengren, P., Antolin, P.: A development platform for integrating wireless devices and sensors into Ambient Intelligence systems. In: Giusto, D., Iera, A., Morabito, G., Atzori, L. (eds.) The Internet of Things. Springer Science+Business, New York (2010)
7. OSGi, http://www.osgi.org/ (retrieved May 26, 2012)
8. Apache River, http://river.apache.org/ (retrieved June 9, 2012)
9. Managed Extensibility Framework, http://archive.msdn.microsoft.com/mef (retr. June 8, 2012)
10. Bandyopadhyay, S., Sengupta, M., Maiti, S., Dutta, S.: A Survey of Middleware for Internet of Things. Communications in Computer and Information Science 162(pt. 2), 288–296 (2011), doi:10.1007/978-3-642-21937-5_27
11. Puliafito, A., Cucinotta, A., Minnolo, A., Zaia, A.: Making the Internet of Things a Reality: The Where X Solution. In: The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications, pp. 99–108. Springer Science+Business Media (2010)
12. Kefalakis, N., Leontiadis, N., Soldatos, J., Donsez, D.: Middleware Building Blocks for Architecting RFID Systems. In: Granelli, F., Skianis, C., Chatzimisios, P., Xiao, Y., Redana, S. (eds.) MOBILIGHT 2009. LNICST, vol. 13, pp. 325–336. Springer, Heidelberg (2009)
13. Vazques, J., Almeida, A., Doamo, I., Laiseca, X., Orduña, P.: Flexeo: An Architecture for Integrating Wireless Sensor Networks into the Internet of Things. In: Corchado, J.M., Tapia, D.I., Jose, J.B. (eds.) 3rd Symposium of Ubiquitous Computing and Ambient Intelligence. ASC, vol. 51, pp. 219–228. Springer, Heidelberg (2009)
14. Chen, Z.-L., Liu, W., Tu, S.-L., Du, W.: A Cooperative Web Framework of Jini into OSGi-based Open Home Gateway. In: Wu, Z., Chen, C., Guo, M., Bu, J. (eds.) ICESS 2004. LNCS, vol. 3605, pp. 570–575. Springer, Heidelberg (2005)
15. Gama, K., Touseau, L., Donsez, D.: Combining heterogeneous service technologies for building an Internet of Things middleware. Computer Communications 35(4), 405–417 (2012)
16. The OSGi Alliance, OSGi Service Platform Core Specification, http://www.osgi.org/Download/File?url=/download/r4v43/osgi.core-4.3.0.pdf/ (retr. May 26, 2012)
17. The OSGi Alliance, OSGi Service Platform Service Compendium, http://www.osgi.org/download/r4v43/osgi.cmpn-4.3.0.pdf (retrieved May 31, 2012)
18. Bosh, R.: CAN Specification Version 2.0, http://www.gaw.ru/data/Interface/CAN_BUS.PDF (retrieved May 26, 2012)
19. Apache CXF, http://cxf.apache.org/ (retrieved May 26, 2012)
20. Restlet, http://www.restlet.org/ (retrieved May 21, 2012)
21. Apache Tomcat, http://tomcat.apache.org/ (retrieved May 21, 2012)
22. Flotyński, J., Picard, W.: Transparent Authorization and Access Control in Event-Based OSGi Environments. In: Information Systems Archictecture and Technology, Service Oriented Networked Systems. Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, pp. 197-210 (2011) ISBN 978-83-7493-625-5