Proceedings of the IASTED International Conference
SOFTWARE ENGINEERING
February 12–14, 2008, Innsbruck, Austria
ISBN Hardcopy: 978-0-88986-715-4 / CD: 978-0-88986-716-1

A SCALABLE SENSOR APPLICATION FRAMEWORK BASED ON HIERARCHICAL LOAD-BALANCING ARCHITECTURE

Yoji Onishi † , Hiroshi Igaki ‡ , Masahide Nakamura ‡ , and Ken-ichi Matsumoto †

† Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, 630-0192 Japan
email: {yoji-o, matumoto}@is.naist.jp
‡ Graduate School of Engineering, Kobe University
1-1 Rokkodai, Nada, Kobe, Hyogo, 657-8501 Japan
email: {igaki, masa-n}@cs.kobe-u.ac.jp

ABSTRACT
Many integrated services with sensors and its appliances become common in our daily life. In most of those services, the application and sensors are tightly-coupled. This causes the implementation of the application becomes more complicated. Moreover, network load between the application and the sensor and processing load to evaluate the sensor data increase severely. As the number of sensors to connect increases, the problems become more serious.

In this paper, we propose a scalable sensor application framework. Using a standardized API, applications can access any sensors without implementing any sensor-specific procedure. Furthermore, the application delegates a part of evaluation process of the trigger conditions to the sensor. Due to the delegation, network load is minimized because the application communicates with the sensor only when the status of the registered condition is changed. Using these methods, we evaluate this framework qualitatively using sequence diagrams of sensor applications.

KEY WORDS
sensor driven service, sensor middleware, event based integration, loose-coupling, load-balancing

1 Introduction

With the emerging ubiquitous technologies, various objects including appliances and sensors have been equipped with network functionalities. Appliances mutually connected through the network provide users with value-added integrated services. Especially, integrated services consisting of sensors and appliances are expected to realize more sophisticated service behavior based on the states of environment, appliance and human.

For example, the sensor light[1] and the automatic door[2] use a human-detect sensor to activate a light and a door. The automatic climate control[3] and the automatic sensor faucet[4] use temperature sensors and a hand sensor. As a more complicated example, [5] proposes an automated living assistance system focusing on the support of handicapped and elderly people in their own homes with monitoring user's health condition. This kind of ubiquitous applications (hereafter, we call this kind of application a sensor application) is mainly installed in a home network system or a building management system.

In most of these applications, the application and sensors are *tightly-coupled*. Each application includes sensor-specific descriptions to access and interpret the sensor data within the implementation. As the numbers and variety of the sensors grow, the implementation of the application becomes more complicated. Excessive complexity of implementation causes lowering the software quality about maintainability, extensibility, etc.

Low scalability is another problem. Sensors monitor the values of environmental attributes continually. Similarly, the sensor application has to meet the changing values of sensors, and select adequate behaviors from pre-defined rules continually. If the numbers of applications and sensors increase, network load between the application and the sensor and processing load to evaluate the sensor data also increase severely.

In this paper, we propose a scalable sensor application framework to support development of various sensor applications using multiple sensors and appliances. In our framework, to avoid the tightly-coupled problem, we wrap the sensor device with a service layer. The service layer includes sensor-specific descriptions and publicizes standardized API using Web Service[6]. Applications based on any platforms and programming languages can access any sensors through the API without implementing any sensor-specific procedure.

Next, we focused on trigger conditions in sensor-driven services. If a trigger condition becomes true, the application activates corresponding actions based on pre-defined rules. The condition is commonly expressed as a threshold condition of the sensor property (For example, in the case of a temperature sensor, as a threshold condition, "if temperature becomes more than 27" is used).

In our framework, the application delegates a part of evaluation process of the trigger conditions to the service layer. The application registers a trigger condition to the sensor which is related to the condition. Each service layer evaluates the registered conditions continuously with the value of the sensor. If states of any condition are changed, the service layer notifies their states to the application. In this architecture, network load is minimized because the application communicates with the sensor only when the status of the registered condition is changed. Since a part of the trigger condition is evaluated by each service layer, the processing load is also decreased.

Furthermore, we propose a meta-sensor framework for more complicated sensor-driven service. The complicated services require evaluating more complicated conditions including multiple sensors. The meta-sensor framework provides hierarchical combination of multiple service layers. For instance, the following service can be developed. "$if\ temperature \geq 28\ and\ humidity \geq 70\%$, $air\_conditioner\ is\ set\ to\ 25\ degrees$"

In this paper, we classify problems of the conventional sensor applications, and describe the proposed sensor application framework to solve the problems. We qualitatively evaluate the framework using sequence diagrams.

2 Preliminary

2.1 Sensor Driven Service

In a sensor-driven service, various sensors (e.g., a temperature sensor, a humidity sensor, a human-detect sensor, an IR distance sensor and a light sensor) are used. Each sensor has a single property and monitors environmental attributes within the specification limits for the property [1].

We define $s_i \in S(1 \leq i \leq n)$ as a sensor, and every sensor s_i has one property p_i which has a property type t_i. p_i can take a value v_i which must be of type t_i. In general, every v_i is derived from an environmental attribute e_i. Note, however, that p_i takes a value depending on the implementation of s_i. Thus, sensors developed by different vendors usually represent different property values for the same environment attribute. Also, the v_i is not necessarily the same as the actual value of e_i. Usually, each s_i assumes a function f_i, which translates the measured v_i into e_i (i.e., $e_i = f_i(v_i)$). For instance, temperature sensor of Phidget[7] s_p has a property p_p. p_p can take a value v_p. v_p indicates values within the limits of type t_p (int $\{40..700\}$). If an application wants to get a temperature e_p, the application has to translate v_p into e_p with the function $f_p(v_p) = (v_p - 200)/4$.

The main role of sensors in sensor applications is to get current value of the environmental attribute. The application manages subscribed *environmental conditions* to activate pre-defined actions of sensor-driven services. To evaluate whether the conditions become true or not, the application communicates with sensors continually. In this paper, we define the environmental condition as follows.

[1]For simplify, we regard a sensor with two or more properties as multiple sensors with single property

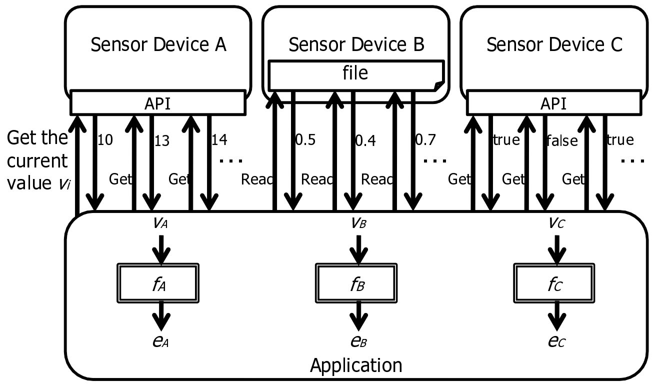


Figure 1. Conventional sensor applications

Let $E = \{e_1, e_2, ..., e_n\}$ be a given set of environmental attributes. An $atom(e_i)$ is any logical formula with respect to e_i. The $atom(e_i)$ is regarded as an atomic construct for environmental conditions $condE$. Every environment condition $condE$ is defined as a condition over $atom(e_i)$'s $(1 \leq i \leq n)$, specifically given by the following BNF.

$condE$ ::		
	$condE$ && $condE$	(AND)
	$condE$ || $condE$	(OR)
	! $condE$	(NOT)
	$(condE)$	
	$atom(e_i)$	

A sensor-driven service is defined as a combination of the environmental condition and actions. For instance, a sensor light service has a $condE$: "$human\_detect == true \&\& brightness < 10$" and an action "$turn\_on\_the\_light$".

Generally, the action executed by the application consists of a set of method invocation, such as appliance integrated services or other service APIs. In this paper, we don't focus on details of the actions.

2.2 Conventional Sensor-Applications

Figure 1 shows a typical example of conventional sensor applications with sensor-driven services (e.g., the sensor light[1], the automatic door[2], the automatic climate control[3], the automatic sensor faucet[4], and the automated living assistance system[5]).

As shown in the figure, different sensor requests different way for accessing from the application. Furthermore, different f_i is required for translation v_i of each sensor into e_i. These differences between each sensor affect the implementation of the application directly. Therefore, a developer of the applications faces the following two problems.

Problem P1: Tightly-coupling between sensors and applications

Problem P2: Low Scalability

The kind of sensors used in the application depends on contents of the services. To update the services and exchange sensors, a developer must update sensor-specific description and condition evaluator within the application. So, problem P1 increases complexity of the applications and prevents developers from flexible customization of the services.

Problem(P2) of low scalability is originated from network load and processing load in the application. As the number and the variety of the service grow, sensors used in the application increase. As shown in the figure 1, the application has to get the value from the sensors continuously. Namely, the application has to keep monitoring the sensed values and evaluating the environmental conditions to execute $atcions$ corresponding to the sensor-driven services. As a result, too many sensors weigh heavily in performance of the applications severely.

In the following section, we introduce our sensor application framework to resolve these problems.

3 Sensor Application Framework Based on Hierarchical Load-Balancing Architecture

3.1 Key Ideas

We propose the following three key ideas to solve problems of the application development for sensor-driven services.

(K1) Standardized API and Loose Coupling for Sensor Device

In order to use a sensor s_i, the following procedures are needed. (1)Access and acquire the data v_p of the sensor. (2)Translate v_p into e_p based on f_i. Conventionally, these procedures differ for every application and every sensor. Then, we wrap each sensor device in a service layer with standardization APIs of the procedures of (1) and (2). The service layer based on Web Service[6] doesn't depend on implementation of applications and sensors.

(K2) Delegation of Evaluation Process for Environmental Conditions

The continuing evaluation process for environmental conditions wastes the throughput of a network and a processor. In our framework, evaluation of environmental conditions is delegated to the service layer. The service layer continues monitoring values e_i of the sensor s_i and replies true or false of the conditions, only when its evaluation result changes. This hierarchical load-balancing architecture improves the network and processing load.

(K3) Meta Sensor for Complex Environmental Conditions

The service layer denoted by K1 and K2 wraps only one sensor. Therefore, the environmental condition which consists of two or more environmental attributes must be evaluated by the application. In order to support a sensor application development which copes with complex environmental conditions, we propose a *meta-sensor* framework. The framework consisting of the service layer and a meta sensor can correspond to $condE$ including multiple environmental attributes. The meta sensor divides $condE$ per $atom(e_i)$ and evaluate whole $condE$ based on notification from the service layer of the sensor corresponding to the $atom(e_i)$. This meta-sensor framework also adopts the hierarchical architecture. The processing and network load are distributed to the meta-sensor and the service layer of the sensors hierarchically.

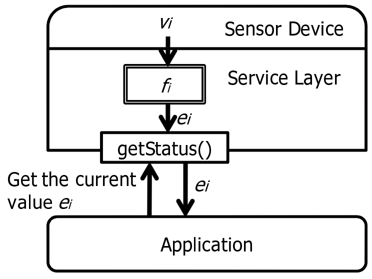


Figure 2. Standardized Interface of Propsed Sensor Framework

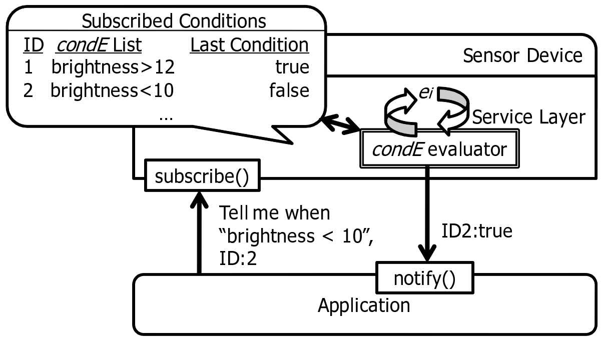


Figure 3. Subscribe and Notify Process in Service Layer

3.2 Standardized API and Loose Coupling

Figure 2 shows a service layer with standardized API. Every service is wrapped by the service layer. The service layer changes v_i of the sensor into e_i of the environmental attribute. An application can acquire the processed data e_i of the sensor s_i with standardized API $getStatus()$. The API of the service layer is realized by Web Service. Procedure to call the API doesn't depend on development platforms or programming languages of the application.

As a result, loose coupling between the application and the sensor is achieved by this service layer.

3.3 Delegation of Evaluation Process

The service layer denoted in Section 3.2 enables every application to access with standardized procedure. However, the application has to continue calling $getStatus()$ and evaluate environmental conditions continually. For load-balancing of network and processors, we realize hierarchical delegation of evaluation process for environmental conditions.

Figure 3 is the improved service layer. The $subscribe()$ method of the service layer receives subscriber id and the environmental condition related to the environmental attribute e_i of the sensor s_i as inputs.

Generally, a simple sensor-driven service is provided in form of $if(condE == true)then\{action\}$ or $while(condE == true)\{action\}$. In if-based service, the service layer has to notify the evaluation result to the application only when $condE$ becomes true. On the other hand, in $while$-based service, the application needs a notification, not only when $condE$ becomes true, but when it becomes false.

So, the service layer evaluates the subscribed environmental conditions continuously based on the value of e_i, and notifies evaluation results(true/false) and subscriber id to the corresponding application only when the results change.

Since continuous polling by the application to the sensor becomes unnecessary, and the load of condition evaluator is distributed by the multiple service layers used in the application, a network load and processing load of the application are reduced.

For instance, we consider a sensor-driven service "$while(brightness < 10)\{turn\_on\_the\_light\}$". First, the application subscribes the condition $brightness < 10$ with a subscriber ID to the sensor. So the service layer immediately evaluates the environmental conditions($brightness < 10$), and notifies the current condition status "$false$" to the application. The service layer continuously monitors $brightness$ and evaluate the subscribed conditions. If the evaluation result changes to "$true$", the service layer notifies the application of the result and subscriber ID. After that, if the result changes to "$false$", the service layer notifies again. Like this, the application can always receive the newest evaluation result from the sensor. In this example, the application continues to run $\{turn\_on\_the\_light\}$ only when the notification from the sensor is "$true$".

3.4 Meta Sensor for Complex Environmental Conditions

The framework described by Section 3.3 can enable a developer to create easily the application for $condE$ which contains only one kind of e_i. However, it is difficult to cope with complex $condE$ includes multiple environmental attributes by the application based on the framework. In this section, we explain a meta-sensor framework for processing complicated environmental conditions.

Figure 4 shows our meta-sensor framework. The framework consists of the service layer and a meta-sensor. The service layer is completely the same as what is described in Section 3.3. It receives $condE$ and subscriber ID as inputs of $subscribe()$. $condE$ is evaluated by the service layer and the evaluation result is notified to the application only when the result changes.

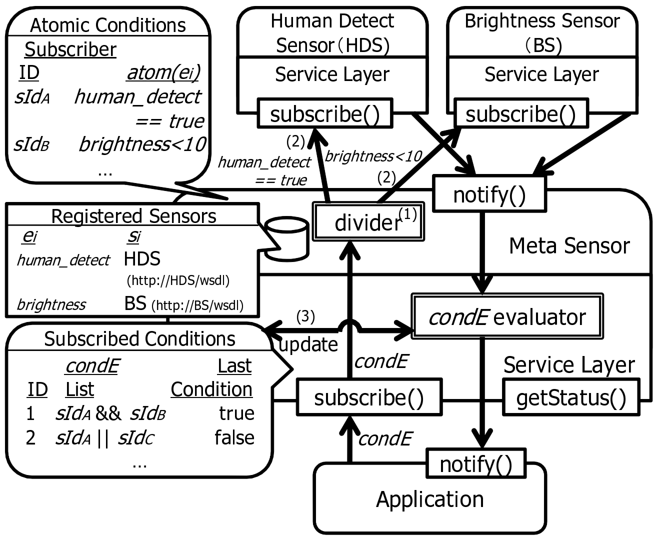


Figure 4. Proposed meta-sensor framework

The meta-sensor manages correspondence between sensor s_i(in form of WSDL[8]) and e_i in $condE$, and performs the following three procedures.

(1) divides $condE$ per $atom(e_i)$

(2) calls $subscribe()$ of s_i corresponding to e_i in the $atom(e_i)$ to register the atomic construct.

(3) updates the status(true/false) of $atom(e_i)$ in $condE$ managed in the service layer whenever notification from s_i is received.

Whenever $atom(e_i)$ is updated by the meta-sensor, the service layer evaluates the whole $condE$.

For example, we explain about executing the following sensor-driven service.

$condE$: "$human\_detect == true \&\& brightness < 10$"
$while(condE)\{action : "turn\_on\_the\_light"\}$

This service uses two sensors, a human-detect sensor and a light sensor. The human-detect sensor corresponds to the environment attribute "$human\_detect$", and the light sensor corresponds to the environment attribute "$brightness$". First, the application subscribes the $condE$ to the service layer of a meta-sensor, and waits notification about the $condE$. The meta-sensor divides the $condE$ into two atomic constructs, $human\_detect == true$ and $brightness < 10$. The $subscribe()$ of each sensor related to each construct is called by the meta-sensor. Then, the meta-sensor updates state of the atomic constructs within $condE$ in the service layer whenever the meta-sensor receives a notification from each sensor. The service layer also evaluates the whole $condE$ simultaneously. If the evaluation result of $condE$ changes, the service layer notifies

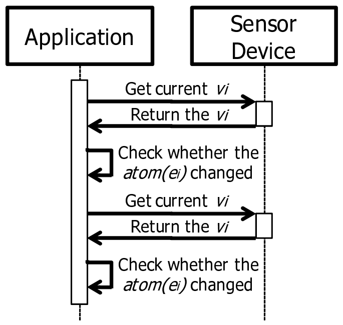
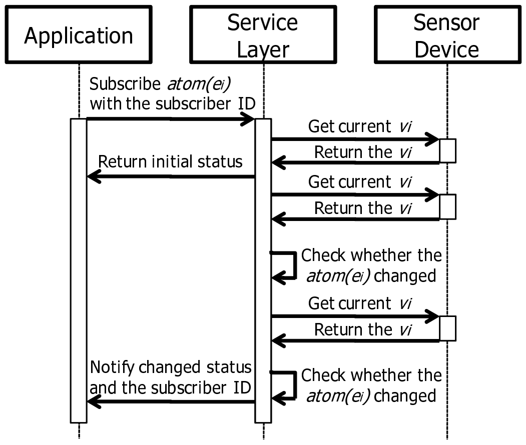


Figure 5. Sequence diagram of sensor-driven service by conventional sensor application



Application
Service Layer
Sensor Device
Subscribe atom(ei) with the subscriber ID
Get current vi
Return the vi
Return initial status
Get current vi
Return the vi
Check whether the atom(ei) changed
Get current vi
Return the vi
Notify changed status and the subscriber ID
Check whether the atom(ei) changed

Figure 6. Sequence diagram of a simple sensor-driven service by our sensor framework based application

the result to the application. Like this, the application can perform the sensor-driven service with complex environmental conditions easily.

4 Discussion

4.1 Qualitative Evaluation

Figure 6 shows a sequence diagram[9] of a simple sensor-driven service by our proposed sensor framework based application. The service contains only one environmental attribute. As shown in this figure, communication between the application and the sensor is performed only when the state of $atom(e_i)$ evaluated by the service layer changes. Based on the notification, this framework guarantees that the application always has the newest state of $atom(e_i)$. Figure 7 shows a sequence diagram in the case of more complicated sensor-driven service with our meta-sensor framework based application. In our meta-sensor framework, complex $condE$ is divided per an atomic construct, and processing of subscribe/notify is distributed per a sensor. As mentioned above, even if the number of sensors used in the application increases, processing load for evaluation of $condE$ is distributed to the meta-sensor and the service layer of each sensor, appropriately.

On the other hand, conventional sensor applications (Figure 5) have to check current data of the sensor and evaluate $condE$ continually to keep the newest state. In this architecture, both network load and processing load concentrate on the application.

As the number of the applications increases, the subjects about the standardized interface and load-balancing become more significant. Based on these perspectives, our hierarchical architecture based framework (meta-sensor and single sensor) is very beneficial to develop the applications providing various sensor-driven services.

A limitation in our framework, the service layer cannot evaluate the environmental conditions from which the threshold value changes dynamically, such as $condE$: "$e_i > e_j$". This is due to the limitation of the service layer which wraps only one sensor device. We think about more flexible architecture about the service layer which wraps two or more sensor devices as the need arises.

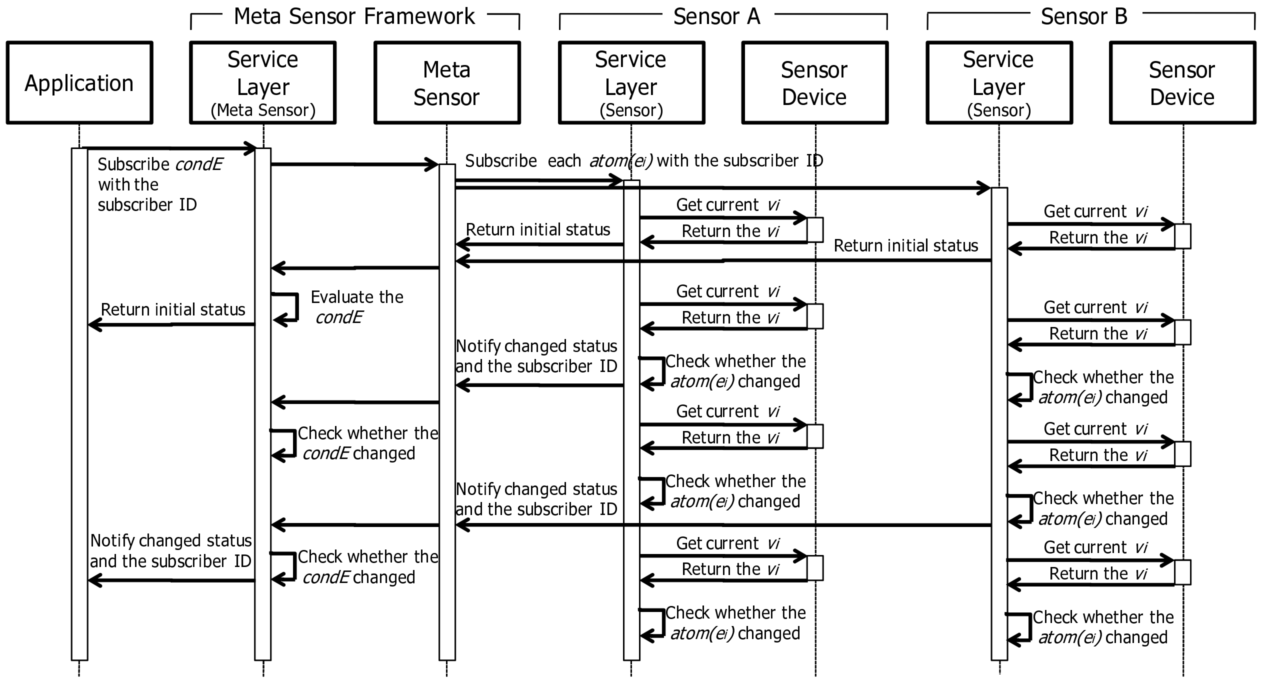
4.2 Related Research

Sashima et al.[10] propose Sensor-Event-Driven Service Coordination Middleware (SENSORD) to fill coordination gaps between higher-level services and lower-level sensors. The SENSORD system obtains and stores sensor data into an in-memory data container to achieve fast, complex analysis of the sensed data. Shankar et al.[11] propose a framework for policy-based management of a ubiquitous computing system. In the framework, sensor-driven services are defined as Event-Condition-Action-Post-Condition rules. Policy-based management enables the framework to detect interactions between services. Though these systems help implementers of context-aware application services to access sensor data with standardized procedure, load of network and processing are concentrated on the system. As a result, increase of the sensors affects the performance of the whole of system severely.

The research in [12] proposes a publish/subscribe based middleware for ubiquitous applications with sensors. With using publish/subscribe message exchange pattern, to some extent, the load of network and processing are distributed. The sensor notifies to the middleware only when subscribed conditions become true. Namely, the complex conditions including multiple sensors are not considered to be evaluated in the middleware.

5 Conclusion

In this paper, we proposed a scalable sensor application framework. This framework provides a standardized API and a load-balancing architecture for sensor applications. The API enables the application to access the data acquired by the various kinds of sensors without depending on the platforms or programming languages. The architecture improves performance of the applications which provide various sensor-driven services including multiple sensors.



Meta Sensor Framework
Sensor A
Sensor B
Application
Service Layer (Meta Sensor)
Meta Sensor
Service Layer (Sensor)
Sensor Device
Service Layer (Sensor)
Sensor Device
Subscribe condE with the subscriber ID
Subscribe each atom(ei) with the subscriber ID
Get current vi
Return the vi
Return initial status
Return initial status
Evaluate the condE
Return initial status
Notify changed status and the subscriber ID
Check whether the atom(ei) changed
Check whether the condE changed
Notify changed status and the subscriber ID
Check whether the condE changed

Figure 7. The sequence diagram of a complicated sensor-driven service by our meta-sensor framework based application

In the future, we plan to develop several sensor-applications which use dozens of sensor. Through the actual development, we compare conventional applications with our framework-based applications, quantitatively.

Acknowledgements

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B) (No. 18700062), Start Up (No.18800060), and Scientific Research (B) (No. 17300007), and by JSPS and MAE under the Japan-France Integrated Action Program (SAKURA).

References

[1] Matsushita Electric Works Ltd. Katteni switch. http://biz.national.jp/Ebox/katte_sw/.

[2] Nabtesco Corp. Automatic entrance system. http://nabco.nabtesco.com/english/door_index.asp.

[3] Daikin Industries Ltd. Air conditioner. http://www.daikin.com/global_ac/.

[4] TOTO USA Inc. Sensor faucet. http://www.totousa.com/prodcatalog.asp?cid=54.

[5] J. Nehmer, M. Becker, A. Karshmer, and R. Lamm. Living Assistance Systems: An Ambient Intelligence Approach. In *Proc. of the 28th Int'l Conf. on Software Engineering(ICSE'06)*, pages 43–50, 2006.

[6] World Wide Web Consortium. Web Services Activity. http://www.w3.org/2002/ws/.

[7] S. Greenberg and C. Fitchett. Phidgets: Easy Development of Physical Interfaces through Physical Widgets. In *Proc. of the 14th Annual ACM Symposium on User Interface Software and Technology(UIST'01)*, pages 209–218, 2001.

[8] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[9] Object Management Group Inc. Unified modeling language (uml), version 2.1.1. http://www.omg.org/technology/documents/formal/uml.htm.

[10] A. Sashima, Y. Inoue, and K. Kurumatani. Spatio-Temporal Sensor Data Management for Context-Aware Services: Designing Sensor-Event Driven Service Coordination Middleware. In *Proc. of the 1st Int'l Workshop on Advanced Data Processing in Ubiquitous Computing (ADPUC'06)*, 2006.

[11] C. S. Shankar, A. Ranganathan, and R. Campbell. An ECA-P Policy-Based Framework for Managing Ubiquitous Computing Environments. In *Proc. of the 2nd Annual Int'l Conf. on Mobile and Ubiquitous Systems(MOBIQUITOUS'05)*, pages 33–44, 2005.

[12] G. Gehlen, F. Aijaz, M. Sajjad, and B. Walke. A Mobile Context Dissemination Middleware. In *Proc. of the Int'l Conf. on Information Technology(ITNG'07)*, pages 155–160, 2007.