

AirSenseWare: Sensor Network Middleware for Information Sharing

Keiro Muro¹, Takehiro Urano¹, Toshiyuki Odaka¹, and Kei Suzuki¹

¹ Central Research Laboratory, Hitachi, Ltd.

1-280, Higashi-koigakubo, Kokubunji, Tokyo 185-8601, Japan,

{keiro.muro.vn, takehiro.urano.pf, toshiya.odaka.en, kei.suzuki.zt}@hitachi.com

Abstract

We describe AirSenseWare, a sensor network middleware that serves abstraction models to user applications. The abstract modelling and its event delivering mechanism let the user handles the real world model instead of sensor node itself, and hide its unexpected dynamic behaviour such as node adding, removing, position changes. Those abstraction models are shareable, which leads the cooperative construction of the real world model. AirSenseWare also serves the application independent maintenance services that lead the easy maintenance for large-scale sensor network infrastructure.

1. INTRODUCTION

A sensor network is a system that consists of a huge number of sensors and actuators pervasively distributed in an environment. This promising paradigm will change our entire lifestyle by monitoring environments for safety, security and energy conservation. New information for preventive maintenance in factory machines, for efficiency estimation and optimization of our workplaces [1], for health care and for context-aware services will be analyzed.

To realise those applications, a *sensor network infrastructure* is needed that covers wide area and various types of observations. Sensor network infrastructure consists of distributed middlewares that handle multiple wireless personal area networks (WPANs) and various multiple sensor nodes, where sensor network applications are integrated and their information are shared with each other. Realising such sensor network infrastructure should satisfy the following requirements. The first requirement is keeping flexibility. The middleware must deal with various ubiquitous devices and user applications. Observation will vary from simple scalar values of temperature to variable-size arrays of 3-axis accelerations. Application will vary from just storing raw data to the real-time analysing and controlling the environment. The second requirement is managing abstraction model. For rapid and low-cost development, users and application developers need to be hidden from its unexpected dynamic behaviour such as node adding, removing, position changes. The third requirement is information sharing, or sharing those abstraction models between each application. Information sharing contributes for scalability by reducing the needless

calculations or storage cost for similar requirements. Model sharing also contributes to correlative programming and maintenance that save the excessive work. The final requirements are scalability. The middleware must deal enough speed performance that each application requires despite the number of nodes and applications. Moreover, maintenance services take importance for such large-scale sensor network infrastructure. A single user cannot maintain huge numbers and variations of sensor nodes. Sensor network middleware should support such services autonomously, or at least the administrators of the sensor networks handle such services and hide those from each user. Those services include replacement of the crashed node, planned node relocation, etc.

The basic features that the sensor network applications should possess can be categorised in three features. The first feature is event handling that collects events from sensor nodes, filters and delivers to applications. The second feature is action handling that handles user logics for querying and manipulating sensors, stores and analyses observed values, and control upper systems. The third feature is data management that manages data (properties, relations and histories) of the network topologies, node settings, observed values and statuses, and other abstract models. Several achievements about these features are made in several domains. The achievement for event handling is the *complex (or composite) event processing (CEP)* [2] in electronic commerce domain. It is based on publish-subscribe model and codifies event algebra operators for filtering multiple events. *RuleML* [3] standardises various event handling logics based on event-condition-action model. The achievement for action handling is the *Business Process Execution Language for Web Services (BPEL4WS)* [4] in Web Services domain. It enables to design flexible action workflow logics that construct integrated services by connecting local Web Services. The achievement for data management is the *Resource Definition Language (RDF)* [5] in Semantic Web domain. It enables to describe flexible definition of properties and relationships of abstract models and serves query functions such as *SPARQL Protocol and RDF Query Language (SPARQL)*.

The initial research for the sensor network middleware focused on the node-side programming abstractions for single sensor network. Those middleware hide the difficulty of handling limited node resources (battery, memory, and CPU

processing ability), or effective routing and aggregating from application developers. The initial server-side middlewares just serves functions (APIs or Web Services) to bridge to the upper layer systems. However, fast local interaction with sensor nodes is required for scalability reason. Therefore, simple and flexible sensor network middleware that supports those three features described above by itself in integrated manner is required. The purpose of this study is to design such middleware that possesses simple core framework and various plug-ins to fit the application specific needs.

The paper is organised as follows: Section 2 compares related works. Section 3 describes the brief concept of our abstraction model by introducing an air quality monitoring application. Section 4 describes the core architecture of AirSenseWare that achieves abstraction and information sharing to the users. Section 5 explains our plug-ins for constructing sensor network applications. Section 6 estimates the performance. Section 7 concludes this paper.

2. RELATED WORK


In sensor network domain, the mainstream research on server-side middleware is focused on the event handling approach [6]. In this area, the research most similar to that of this paper is *Global Sensor Network (GSN)* [8]. *GSN* achieves the flexibility of event delivery by scripting the peer-to-peer event delivering flows as *virtual sensors*, which possess translation rules to generate output streams from multiple input streams. Those translation logics are written in C code, and selected by extended mark-up language (XML) configurations. *GSN* solves the scalability problem by manipulating observed data nearby sensor node. However, *GSN* requires human manipulations to add rules when new sensor nodes are added. Our approach adds the feature of dealing with dynamic topology changes that are typical for sensor networks. The action-handling approach focused on node-side programmability. *Maté* [9] and *SensorWare* [10] controls inside single WPAN. Our approach focused on server-side programmability that controls multiple WPANs and other distributed systems, and adds functions for dynamic injecting and cancelling logics from remote monitoring centre. From the viewpoint of data management approach, our approach is based on the idea of *IrisNet* [11], which integrates the location relationships and the observed values in XML form, and gives the semantic methods for querying sensors. We redefine the data structure in a manner of ontology language *RDF* [5] and integrate event-delivery mechanisms for updating data.

3. APPLICATION OVERVIEW

Our final goal of constructing the sensor network infrastructure will be achieved in the following stages. First, construct a sensor network application that satisfies the user requirement. Second, add another application that shares the prior sensor network. Finally, construct the sensor network

infrastructure where any application can be created by just adding sensor nodes and user plug-in logics, without stopping and disturbing middleware and other applications.

Our current status is in the second stage. We provide a general sensor network system named *AirSense*TM, and a middleware named *AirSenseWare* [15]. *AirSense* is based on a ZigBee® ad-hoc mesh network platform [12] that consists of a gateway node (coordinator), router nodes, and battery-powered *portable sensor nodes* (sleepy end devices). The portable sensor node possesses extensibility for adding other sensors and actuators by its serial and Inter-Integrated Circuit (I²C) interfaces. The view image and the specifications of the portable sensor node are shown in Figure 1.



built-in sensors	temperature, humidity, 3-axis accelerations
interfaces	serial (2 ch), I ² C 1 ch
memory	64 kB E ² PROM
battery	1400 mAh (CR123A)
wireless IF	ZigBee 2400 - 2480 MHz (EmberZNet 2.2)
distance	30m - 100m per hop
size, weight	H: 45 mm, W: 60 mm, D: 25 mm, 100 g

Fig. 1: View image and the specifications of the portable sensor node.

For belief understanding of what *AirSenseWare* can do, we will introduce a typical sensor network application that we have developed. An air quality monitoring system for a food factory consists of 4 WPANs for each floor and 60 sensor nodes that monitor the temperature, humidity, and airborne particles inside the manufacturing premises every ten minutes. Figure 2 describes the abstraction model for the food factory. Physical objects such as sensor nodes and WPANs, logical objects such as floors, production processes, package air conditioners (PACs), its control systems (PAC Systems), applications such as map graphical user interfaces (GUIs), chart GUIs, mailers to notify warnings are described in each abstract model. In this case, node A and B are in first floor, node C and D are in second floor. Node B and C are in production process 1. Maps connect to each floor and display the floor map with the name, position, observed values of the sensor nodes inside those floors. Charts connect to the floors or production process to display the time series chart of the sensor nodes inside them. A PAC system connects to the sensor node C and D because it has responsibility to control their temperatures. PAC system consists of PAC E and F that are inside the second floor.

In *AirSenseWare*, query goes along the forward path of their relations. Events are delivered along the return path from the relation target to the source. Using this feature, application scenario consists of following parts. One, the maps and charts of each floors or production process are updated, triggered by the arrival of the observation event from the related sensor nodes. End users can change relations to display each GUI for each floor or production process. Two, newly installed sensor node will be added automatically to the floor when the WPAN detects its joining event. New sensor node is added on the related maps or charts. Three, the

observed values are determined to cast warning events when they exceed the threshold given for each sensor nodes. Those warning events are delivered to each application and processed or ignored according to their rules. Warning rules are related to each sensor node. Rule sharing is easy by relating same rule between those sensor nodes. Four, unusual temperature may derived from the abnormal behaviour of the PAC system, and other PACs related to the system may cause same problems. Therefore, every warning from sensor nodes are delivered to their related PAC systems, every related PACs alert the warning that is delivered to each GUI. The building manager has responsibility for updating the relation between nodes and PACs, and they are concealed from the air quality managers.

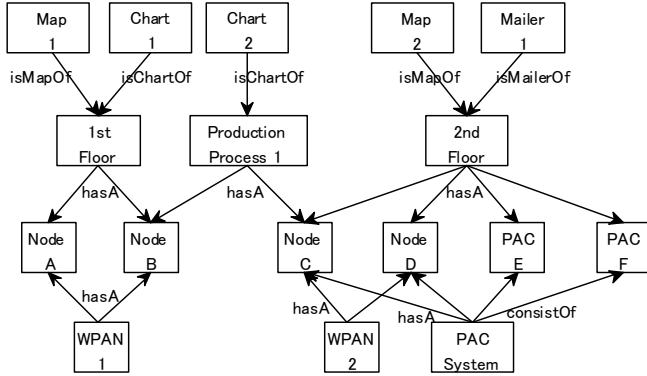


Fig. 2: Abstraction models for a food factory. Each rectangle represents an abstraction model, and arrows represent relationship between the source and the target model. Query goes along the forward path of the relation. Events are delivered along the return path from the relation target to the source.

4. AIRSENSEWARE ARCHITECTURE

A. Architecture Overview

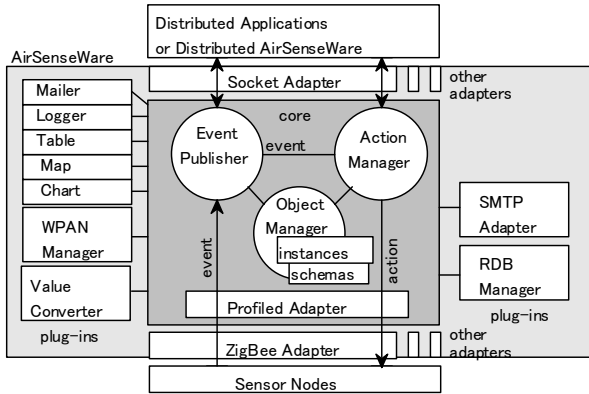


Fig. 3: Architecture of AirSenseWare consists of four core components and plug-ins for storing, converting, forwarding, analysing, and displaying data in GUI.

A rough sketch of the architecture of the AirSenseWare is shown in Figure 3. AirSenseWare is a Java based software program running on a server machine. AirSenseWare consists of four core components and various plug-in components that are adaptable according to application requirements. The core components consist of the Event Publisher that delivers events, the Action Manager that handles actions, the Object

Manager that manages data and the Profiled Adapter that converts payloads. In simplest configuration, AirSenseWare runs as a bridge program that forwards the observed values from sensor nodes to upper applications and transfers application commands to the sensor nodes. Based on its architectural simplicity, AirSenseWare programs can connect with each other in a cascade configuration for load balancing.

TABLE 1: DEFINITIONS OF EVENT, ACTION, INSTANCE, SCHEMA AND PROFILE

(a) Event	$\langle \text{Event} \rangle ::= \langle \text{eventType} \rangle$ $\langle \text{creator} \rangle ::= \langle \text{reporter} \rangle \langle \text{parameter} \rangle^*$ $\langle \text{creator} \rangle ::= \langle \text{id} \rangle \langle \text{timestamp} \rangle$ $\langle \text{reporter} \rangle ::= \langle \text{id} \rangle [\langle \text{timestamp} \rangle]$ $\langle \text{parameter} \rangle ::= \langle \text{name} \rangle \langle \text{value} \rangle$
(b) Action	$\langle \text{Action} \rangle ::= \langle \text{cmdName} \rangle \langle \text{parameter} \rangle^*$ $\langle \text{parameter} \rangle ::= \langle \text{Action} \rangle (\langle \text{name} \rangle \langle \text{value} \rangle)$
(c) Instance	$\langle \text{instance} \rangle ::= \langle \text{schema name} \rangle \langle \text{id} \rangle$ $(\langle \text{property} \rangle \langle \text{relation} \rangle)^*$ $\langle \text{property} \rangle ::= \langle \text{name} \rangle \langle \text{value} \rangle$ $\langle \text{relation} \rangle ::= \langle \text{name} \rangle$ $[\langle \text{comparator} \rangle] \langle \text{instance} \rangle^*$
(d) Schema	$\langle \text{schema} \rangle ::= \langle \text{schema name} \rangle \langle \text{plug-In name} \rangle$ $(\langle \text{s-property} \rangle \langle \text{s-relation} \rangle)^*$ $\langle \text{s-property} \rangle ::= \langle \text{name} \rangle \langle \text{data type} \rangle \langle \text{uom} \rangle$ $\langle \text{s-relation} \rangle$ $::= \langle \text{name} \rangle \langle \text{plurality} \rangle \langle \text{schema name} \rangle^*$
(e) Profile	$\langle \text{profile} \rangle ::= (\langle \text{id} \rangle \langle \text{rule} \rangle)^*$ $\langle \text{rule} \rangle ::= \langle \text{m-sequence} \rangle \langle \text{choice} \rangle$ $ \langle \text{element} \rangle \langle \text{matrix} \rangle$ $\langle \text{m-sequence} \rangle ::= \text{M-SEQUENCE} [\langle \text{name} \rangle]$ $\langle \text{sequence} \rangle^*$ $\langle \text{choice} \rangle ::= \text{CHOICE} [\langle \text{name} \rangle] \langle \text{sequence} \rangle^*$ $\langle \text{sequence} \rangle ::= \text{SEQUENCE}$ $(\langle \text{m-sequence} \rangle \langle \text{choice} \rangle \langle \text{element} \rangle)^*$ $\langle \text{element} \rangle ::= \text{ELEMENT} \langle \text{name} \rangle \langle \text{dataType} \rangle$ $\langle \text{matrix} \rangle ::= \text{MATRIX} \langle \text{columnNum} \rangle \langle \text{baseType} \rangle$
(f) Payload	$\langle \text{payload} \rangle ::= \langle \text{header} \rangle \langle \text{id} \rangle \langle \text{sequence} \rangle$ $\langle \text{sequence} \rangle ::= (\langle \text{m-sequence} \rangle \langle \text{choice} \rangle$ $ \langle \text{element} \rangle)^*$ $\langle \text{m-sequence} \rangle ::= \langle \text{number} \rangle \langle \text{sequence} \rangle^*$ $\langle \text{choice} \rangle ::= \langle \text{type} \rangle \langle \text{sequence} \rangle$ $\langle \text{element} \rangle ::= \langle \text{int8} \rangle \langle \text{timestamp} \rangle \dots \langle \text{matrix} \rangle$ $\langle \text{matrix} \rangle ::= \langle \text{number} \rangle \langle \text{element} \rangle^*$

To satisfy the flexibility requirement, we designed all the messages and data structures that the core components use in XML form. The Extended Backus-Naur form (EBNF) definitions of the structures of event, action, instance, schema, profile and payload are shown in Table 1.

The Event Publisher receives and distributes *events*. The structure of the event (Table 1(a)) is similar to that of prior studies [6][7]. The event consists of its type, ID of the creator of the event, timestamps, ID of the reporter that relays the event, and zero or more parameters. AirSenseWare receives primary events from sensor nodes where the creator IDs are IEEE extended addresses of those nodes and the reporter IDs are the combinations of IP addresses and port numbers of the gateway nodes those relay those events.

The Action Manager receives and executes *actions*. The action means a command or a combination of commands (script). The structure of the action (Table 1(b)) is based on a functional language like Lisp, where commands can include commands recursively. The results of inner commands are used as parameters of their outer command.

The Object Manager stores *instances*. The structures of *instance* (Table 1(c)) is similar to the ontology language *RDF* [5]. The instance embodies the abstract model defined by its *schema* (Table 1(d)). Application developers can define any schema for their own needs. An instance (such as a sensor node) possesses multiple properties that express its metadata (such as its location), node setting (such as its sleep period), and latest observed values (such as temperature) in a unified manner. The relation represents the relationship between instances. It is used to express the relation between WPAN and nodes. The relationship between user defined abstract models such as "a floor possesses rooms" can be expressed.

The Profiled Adapter encodes commands and decodes their responses or events from wireless *payloads* (Table 1(f)) according to their *profiles* (Table 1(e)). For ZigBee sensor networks, the combination of endpoint ID (the subsystem of the node) and the cluster ID (the interface point of the subsystem) is used to match the payload and the profile. Other ID types can be used for other ubiquitous systems such as non-ZigBee sensor networks or Radio Frequency Identification (RFID) systems by adding adapter plug-ins. To extend the structural flexibility to the wireless communication, we defined a binary XML payload. Several data types and their structures, multiple arrays, selective structures can be expressed. The combination of temperature and humidity or variable length arrays of 3-axis accelerations (x, y, z) can be expressed in this form. The constraint rules for payload designer are: (1) number of iterations must be prepended to the variable array (m-sequence); (2) chosen type enumeration must be prepended to the selective structure (choice). We defined several data types with various bit sizes and some special data types like a timestamp (16 bit seconds from 1970/1/1). We prepared a matrix type for speed performance. The matrix type represents variable-length arrays of the vector such as 3-axis accelerations. The profiled Adapter does not decode the matrix. The matrix will be stored as a binary large object (BLOB) in a database, and the final applications should own the responsibility for decoding.

B. Event Publisher

The Event Publisher supports publish subscribe model with content-based subscription. We prepared *comparators* that filter events by event types or event parameters. According to the subscription rule described by the combination of the comparators, events are delivered to the plug-in event handlers that convert the unit of measure (uom) of the observed values, store the latest or historical observed values, forward them to distributed applications, or send control commands to sensor nodes. We also support composite event filters such as a *not-comparator* that checks the absence of the event for a given period or a *history-comparator* that checks multiple occurrences of the event during a given period. For example, the *not-comparator* sends information about lost nodes (gone away or battery exhausted nodes) to the remote monitoring centre. The *history-comparator* reduces the delivery of warning events to avoid over-issuing of the alert mail.

C. Action Manager

The Action Manager is a script interpreter that supports the execution of action. We prepared several commands that includes logic control commands such as *if-command*, compare commands such as *greaterThan-command*, arithmetic commands such as *average-command*, logical operator command such as *and-command*, *or-command*, *not-command*, event handling command such as *whenever-command*, action delivering command such as *ask-command*, event delivering command such as *subscribe-command* and command for sensor nodes such as *nodeCommand-command*. Application developers are allowed to add new commands by adapting command handler plug-ins. For example, RDB Manager plug-in adds *insert-command* that inserts the observed values to the relational database.

Action Manager parses given action by depth-first search (backtracking). If a command possesses no inner command, Action Manager executes the command and uses its result for the parameter of its outer command. If a command is asynchronous, the command returns "reserved" which leads its ancestor commands to return "reserved" recursively and stop the Action Manager. Asynchronous command sends an event when it finishes its asynchronous job. When Action Manager receives the event, it swaps the asynchronous command to its result that is given in the event and restarts parsing from top again. In this way, Action Manager achieves the asynchronous execution of commands such as a command toward sleepy sensor nodes, which return their results after a long period. When parallel-type commands receive the "reserved" result from their inner commands, they parse sibling commands of the asynchronous command before they return "reserved". In this way, Action Manager achieves the parallel execution of asynchronous commands such as sending commands to multiple sensor nodes at the same moment and waits their replies.

The Socket Adapter adds *ask-command* that delivers its inner command to the distributed system and ask to execute. The *ask-command* is an asynchronous command that receives the result from the distributed system. Using this feature, users can write action logics that control upper application by remote procedure call (RPC). Moreover, the monitoring centre can issue diagnostic logics to each AirSenseWare in local sites, which contributes for scalability.

The *whenever-command* realises event handling. It subscribes its comparator to the Event Publisher and executes its inner command when events are delivered. In this way, event-handling logics are integrated with actions, which enable the dynamic delivering, execution, cancelling from the distributed applications. For example, the action written below expresses the logic that ask AirSenseWare to send a warning mail whenever the "observed" event does not arrive from node id=10 for 60 seconds. The *ask-command* delivers the *whenever-command* and the *sendMail-command*. The *not-comparator*, *and-comparator*, *type-comparator* that check event type and *isParam-comparator* that checks event parameter are used for event filtering.

```

<ask url="192.168.0.1" port="5000">
  <whenever>
    <not>
      <and>
        <type>observed</type>
        <isParam name="creator/id">10</isParam>
      </and>
      <time>60</time>
    </not>
    <sendMail>...</sendMail>
  </whenever>
</ask>

```

D. Object Manager

We designed the Object Manager as an XML based active database system [13]. The Object Manager stores properties and relationships of abstract models such as sensor nodes, WPANs, rooms, production processes, and working people. The Object Manager supports XPath [14] interface to search for a target. For example, the action below queries the latest temperature of the first sensor node that is in the floor that is related to the map id=1.

```

<getProperty XPath="Map[@id='1']/isMapOf
/Floor/hasA/Node[0]/temperature"/>

```

The Object Manager supports modification commands such as create or drop instances, update properties and bind or unbind relationships. Those commands generate secondary events such as created, dropped, updated, bound and unbound. Those events are delivered to the instances those have relations to the modified instance. To avoid the event number explosion, event filtering during the event delivery is supported. The comparators can be added to the relation of the instance (Table 1(c)). The event subscription to the Event Publisher starts when those instances are bound, and ends when unbounded. In this way, automatic setting of the event delivery is achieved. For example, the instance below defines that a floor has a property of temperature and relation to sensor nodes. The floor requests the delivery of the *updated*-event from its related sensor nodes. When a new sensor node is bound, the *updated*-events from the sensor node are delivered to the floor model and other models that relates to the floor..

```

<Floor id="1">
  <temperature>22.5</temperature>
  <hasNode>
    <comparator>
      <type>updated</type>
    </comparator>
    <Node id="2">
      <MAC>000d6f0000105a62</MAC>
      <temperature>22.0</temperature>
    </Node>
    <Node id="3">
      <MAC>000d6f0000105a63</MAC>
      <temperature>23.0</temperature>
    </Node>
  </hasNode>
</Floor>

```

For the secondary event such as *updated*-event, the creator ID becomes the ID of the instance that generates the event, and the reporter ID becomes the ID of the instance that relays the event. Therefore, event handlers can receive those events of the target instance by comparing its reporter ID.

Application developer can add event-handler plug-in components to the target instance by adding the name of the plug-in to the schema definition (Table 1(d)). AirSenseWare subscribes to deliver every events that are delivered to the instance of selected schema to that plug-in. Various GUI plug-ins receive events and triggered in this way.

5. PLUG-INS FOR APPLICATIONS

User logics and maintenance logics can be adapted as action scripts or plug-ins. Simple logics such as setting clock to sensor nodes periodically can be written in few commands using *whenever*-command. Logics that are more complicated can be written in Java code as event-handler plug-ins or command-handler plug-ins. We will introduce some of our plug-ins that are used in our application described in section 3.

The first plug-in is an event-handler that updates the observations of sensor nodes. The sensor nodes send *observed*-events that include multiple observed values. This plug-in queries the node instance by its MAC address written in the creator's ID of the event. It updates properties that have the same name with the name of the parameter of the event. The plug-in also compares the IP addresses and the port number of the WPAN of the node instance with the reporter's ID of the event. Match failure means the node has moved to another WPAN. Therefore, the plug-in rebind the node instance to the correct WPAN instance. The plug-in also subscribes node-lost event handler for newly arrived node. The node-lost event handlers are subscribed with *not*-comparator for each node. They detect the absence of event from selected nodes for given period. When they receive the events, they unbind the node instance from their WPANs and issues warning events for node lost. We prepared a command for node setting. The command possesses a parameter such as sleep period. The command sends a wireless command to the sensor node to update the setting. If the node replies successful result, the command modifies the property of the instance that has same name with its parameter. Changing sleep period triggers to change node lost threshold periods.

We add an event-handler that generates warning and warn ended events. The application requires warning events in some conditions. Warning event are generated by checking the condition of the input events. For simple example, when the temperature exceeds the given threshold, warning event is generated. There is possibility for the chattering of the warning and warn-end event if the observed value goes nearby the threshold. To avoid this chattering, the Warning end events are generated when the warning events do not arrive in a given period. It is realised by *not*-comparator. Warning mail consists of a user defined text such as "The temperatures of the refrigerator room #1 becomes 10°C that exceed the threshold 5°C at 10:00:00". This is realised by writing action that includes text generate command and property query commands..

The application requires being unaware of the physical changes of each sensor node. Such changes may happen by node replacement and planned relocation. When a sensor

node crashed or its battery exhausted, replacement to the new sensor node may happen. The MAC address that identifies the physical node will change, but we should hide this change from applications. For example, historical data should be continued in a single time series. In this case, changing the MAC address property of that node instance in the Object Manager solves the problem. On the other hand, user may wish to relocate the sensor node to obtain observations of another location, due to layout changes. This relocation and measuring the different place will cause misunderstanding for other users that share the sensor node. In this case, we create a new sensor node instance that has the same MAC address with the old one. "Currently using" property is added to distinguish each node.

For static observation applications, the fast construction of the network topology is required. The newly added sensor node does not know the right WPAN to be connected. Therefore, when several WPANs are placed nearby each other, the sensor node will connect to the most preferable WPAN, that is, the WPAN that response with best link qualities. This also happens when changing the network channel of the WPAN. Many nodes will escape to the different WPANs and the administrator will argue gathering up those nodes. The node relocation service solves this problem. It requires the *rejoin* command that force the node disjoin from current WPAN and join to the WPAN with given network channel and id. The service adds relationships to the target WPAN in the instances of sensor nodes. The service subscribe to watch the node events, and if the reporter id (that is the id of the current connected WPAN) if different from the target WPAN, then send a rejoin command.

6. PERFORMANCE EVALUATION

The XML based architecture supports enough flexibility instead of sacrificing the speed performance. We evaluated the speed performance of AirSenseWare by running on a personal computer of which CPU spec is Pentium 4, 1.8 GHz, with memory size of 512 MB. Table 2 shows the number of events processed per second. The event we used consists of two observation values of temperature and humidity. The result 1 shows the XML parsing speed is not bad. Our detailed estimation indicates that the bottleneck exists in socket interface and GUI plug-ins. Our estimation satisfies our requirement that is described in section 3.

TABLE 2: THE RESULT OF THE SCALABILITY ESTIMATION

#	condition	speed
1	Receive events that are generated by the middleware and count these events (memory operation only).	2,600 [e/sec]
2	Receive events from sensor nodes and store the observed values to the database.	190 [e/sec]
3	Adding to condition 2, send event to a distributed client and draw a time series chart.	77 [e/sec]

7. CONCLUSION

Researches for sensor network middleware are focused to the node-side abstraction, and server-side middleware are not

focused well. We have argued the needs for the server-side middleware that possesses flexible features of event handling, action handling and data management by itself for scalability reason. AirSenseWare, a sensor network middleware that we are developing, demonstrates that multiple sensor network applications can share their information each other. Its abstract model and event delivery mechanism enable multiple applications to run together. Those applications shares information that are maintained by different users. Dynamic behaviour such as node adding, removing, position changes are hidden from each applications. Maintenance services such as node replacements and relocations are manipulated dynamically from remote monitoring centre, and hidden from each application. However, there remains much to do. Our next challenge is to evaluate the scalability for fully distributed system for large-scale sensor networks.

ACKNOWLEDGEMENTS

We would like to thank GRAPESTONE Co., Ltd. and Takenaka Corporation for jointly developing the air quality monitoring system. We also would like to thank the Wireless Info Venture Company, Hitachi Ltd, and Hitachi Electronics Services Co., Ltd. for jointly developing AirSenseWare.

REFERENCES

- [1] D. O. Olguin, B. N. Waber, T. Kim, A. Mohan, K. Ara, and A. S. Pentland, "Sensible Organizations: Technology and Methodology for Automatically Measuring Organizational Behavior", IEEE Transactions on Systems (2007)
- [2] Coral8, Inc., "Complex Event Processing: Ten design Patterns", available from <http://www.coral8.com/developers/documentation.html>
- [3] A. Paschke, "ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language", Int. Conf. on RuleML'06, (2006)
- [4] P. Wohead, W. Aalst, M. Dumas and A. Hofstede, "Pattern Based Analysis of BPEL4WS", Technical Report FIT-TR-2002-04, QUT
- [5] W3C, "Resource Description Framework ", <http://www.w3.org/RDF/>
- [6] K. Römer and F. Mattern, "Event-Based Systems for Detectiong Real-World States with Sensor Networks: A Critical Analysis", in ISSNIP (2004).
- [7] K. Römer, "Programming Paradigms and Middleware for Sensor Networks", GI/ITG Fachgespräch Sensornetze, Karlsruhe, 26-27 (2004).
- [8] K. Aberer, M. Hauswirth, and A. Salehi, "The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks," Tech. Rep. LSIR-REPORT-2006-006, Ecole Polytechnique Fédérale de Lausanne, 2006.
- [9] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks", In proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pp. 85-95 ACM Press (2002).
- [10] A. Boulis, C. C. Han, and M. B. Srivastava, "Design and Implementation of a Framework for Programmable and Efficient Sensor Networks", In MobiSys (2003).
- [11] A. Deshpande, S. Nath, P. B. Gibbons, and S. Seshan, "IrisNet: Internet-Scale Resource-Intensive Sensor Network Services", ACM SIGMOD (2003).
- [12] ZigBee® Alliance, <http://www.zigbee.org/en/index.asp>
- [13] N. W. Paton and Oscar Diaz, "Active Database Systems", ACM Computing Surveys, Vol. 31, No. 1 (1999).
- [14] W3C, "XML Path Language", <http://www.w3.org/TR/xpath>
- [15] HITACHI Wirelessinfo Venture Company, AirSense™ (in Japanese language), <http://www.hitachi.co.jp/wirelessinfo/airsense/index.html>