Large-Scale Middleware for Ubiquitous Sensor Networks

Young-Sik Jeong, Eun-Ha Song, and Gab-Byung Chae, Wonkwang University
Min Hong and Doo-Soon Park, Soonchunhyang University

As our information society transitions from human- to object-oriented information, especially in ubiquitous-network environments, the advanced technologies developed for ubiquitous sensor networks (USNs) have received considerable attention. In addition, the prevalence of USN computing environments raises the issue of how applications can take full advantage of context-aware information. The main tasks of ubiquitous computing include generating new information from objects on the basis of data received from sensors, transmitting the newly generated information through wireless networks, analyzing the information received, and performing the specific tasks from that analysis. To establish a ubiquitous-computing environment, various fields (administration, medical, transportation, environment, disaster prevention, and so on) have incorporated USN technologies (such as sensor node hardware, sensor networks, and USN middleware) and USN application services (such as ecosystem forecasting, sniper countermeasures, health monitoring, environment observation, and surveillance to detect intruders).

A USN involves the collaboration of several sensor nodes having heterogeneous capabilities. Once a sensor network is deployed, the middleware of the USN gathers sensor information (data sensed by each sensor) from these heterogeneous sensor nodes. Such networks usually cannot operate without any centralized entities, but rather must be connected to an external stationary network that monitors and controls the participating sensor nodes.

Existing USN middleware typically manages information collected from sensor nodes or responds to requests from USN application services. However, such middleware should be capable of managing and storing a large volume of information received from thousands of sensor nodes. Researchers have presented various USN middleware systems (see the “Related Work” sidebar). These systems, however, deal only with partial USN service, and they fail to consider various types of requests as well as the sensed data’s size.
Existing ubiquitous sensor network (USN) middleware systems include TinyDB, Cougar, SINA (Sensor Information Networking Architecture), DSWare (Data Service Middleware), Milan (Middleware Linking Applications and Networks), Impala, and Maté. TinyDB is database-supported server-side middleware that treats a wireless sensor network (WSN) as a virtual distributed database. TinyDB uses an operating server and in-network middleware for collaboration and supports a query language similar to Structured Query Language (SQL), along with in-network aggregation and query-processing optimization. However, it works only with sensor nodes that are compatible with TinyOS and must be debugged to query the module when a new node is added in the sensor network.

Cougar is also database-supported server-side middleware that supports an SQL-like query language. Cougar provides dynamic group management for sensor nodes and employs both server-side and in-network middleware. Milan maximizes a lifetime of sensor networks via resource analysis of the sensor network and quality-of-service (QoS) requests. However, it is tightly coupled with a particular USN application service and cannot support mobile-sensor nodes. Impala can dynamically update a sensor node’s function through a mobile-code-based binary command. But Impala is designed to run only on Hewlett-Packard computers, thus restricting its applicability. In addition, Milan and Impala fail to provide a service that abstracts information from heterogeneous nodes.

Maté can dynamically update a sensor node’s function using byte code and virtual machines. It can also support a USN application service’s QoS-demanding process. However, because the configuration of sensor nodes is relatively complex, Maté cannot efficiently interpret data from sensor nodes. Researchers have also proposed various middleware mechanisms to enhance context awareness using an agent-based network-side approach.

Therefore, we have designed and implemented novel middleware for larger-scale USNs, called Lamses (Large-Scale Middleware for Ubiquitous Sensor Networks). Lamses provides the basic functionalities of existing middleware mechanisms, but also has several additional features.

**Lamses Architecture**

Lamses effectively manages sensed information collected from sensor nodes and promptly responds to queries from USN application services. It also efficiently maintains meta-information on sensor networks and nodes. Lamses collects sensed information and forwards it to users. It also provides new status information by analyzing the collected sensed information. Figure 1 describes the Lamses architecture.

Lamses has five main components: service management (SVM), meta-information management, sensing management, state management, and control and query management. (USN: ubiquitous sensor network.)

**Service Management**

The SVM component contains two parts: the application server manager and the application server interface. The application server manager accepts query requests of a USN application service from clients. The application server interface acts as a communication interface between the application server manager and the clients. The meta-information management component requests meta-information from the requested SVM service or creates new meta-information.
Meta-information Management

The meta-information management component requests hardware information about a sensor node and a sensor network from a client and stores the created information in a meta-information management database. This component includes six parts: the command analyzer, the search engine, the meta-information maker, the XML generator, the XML transmitter, and the meta-information database.

When a client implements a meta-information-related event, this event transmits a command event from the meta-information management component through the CCI. Through the command analyzer, the command event received by the CCI analyzes the event that the client must implement. The command event then searches for related meta-information data and determines whether this data is a new meta-information input or not.

When a command for a new meta-information input is requested, the client calls the meta-information maker, which analyzes the meta-information-related XML data transmitted from the client and inserts the result into the meta-information database. When the client requests meta-information data, the meta-information maker creates a meta-information database search query through the search engine and searches the meta-information database on the basis of the created query. The XML generator creates an XML document containing the result of the data search through the search engine. The XML transmitter then forwards the created XML document to the client that initially requested the command through the CCI.

Each table of the meta-information database consists of a sensor network, sensor nodes, and transducer information. The sensor network table is a network in which many sensor nodes transmit data wirelessly to one another. A sensor node is a unit system composed of a transducer, a processor, a communication processor, and a battery, and it contains more than one sensor. The sensor node table sends the sensed information and the sensor-related specific event to a gateway. The transducer table shows the sensor information installed in sensor nodes. The sensing-type specification table shows a specific sensor’s sensing period and data type. The transducer hardware specification table shows the sensor hardware information.

There are two types of metadata in the entity relationship diagram of the meta-information management system: static and dynamic. Static metadata includes the sensor network ID, the sensor node ID, the sensor ID, the manufacturing company, the model name, the installation location, and the profile. Dynamic metadata includes the operating sensor, the sensor mode, the sensor detection range, the operating sensor network, and the sensing cycle.

Sensing Management

The sensing-management component is the core manager of Lamses. It processes large-scale sensed information and sends information from a specific sensor to the state management component when an event occurs. The state management component stores, in the database, the transmitted event log from the sensing-management component.

State Management

The state management component acquires, integrates, and analyzes the collected sensed information, and is implemented when an event is called. State management is a context-aware operation that processes an event context and handles event analysis, cognition, integration, and storage.

The state management component includes the CCI, the data receiver, the data filter, the meta-information requester, the state delimiter, the storage keeper, and the state repository.

The data receiver extracts the specific sensor’s ID from the data transmitted from the sensing-management component, sends this ID to the meta-information requester, and transmits the data to the data filter. Next, the data filter analyzes the transmitted data and extracts the data to be stored in the database. The meta-information requester uses the sensor ID to request meta-information. The state-management component then sends the extracted data from the data filter, as well as the meta-information from the meta-information requester, to the state delimiter. When the state delimiter receives events, it extracts and stores the necessary information. It then integrates this sensor event data and meta-information and sends the result to the storage keeper. The storage keeper in turn stores sensor-unit-divided data in the state repository.

Control and Query Management

The control and query management component creates a query containing sensor control commands for the USN application service, and sends control-related requests to sensors. This component also requests information from the database (such as sensor control commands, user information, and sensor installation information) and sends it to a client.

The control and query management component has the following parts: the CCI, the control and query analyzer, the database query generator, the request processor, the XML transmitter, the context provider, the rule discriminator, the reasoning engine, the query operator, and the database. The main tasks that the control and
query management component provides for a client are query requests and sensor control. The control and query analyzer receives query requests from events containing commands transmitted by the CCI and control requests, and calls a necessary function to respond to each request. When clients request a query, the control and query analyzer forwards the basic information in the requested query to the database query generator, which creates a database search query and sends it to the request processor. The request processor then enhances performance by implementing query optimization operations when the received query doesn’t have any errors. Query optimization includes

- query sort,
- query disintegration,
- common query extraction and implementation, and
- query results integration.

The request processor then searches the database using an optimized search query and stores the result in an XML document. The XML transmitter sends this document to the client, and the control and query analyzer calls a function to respond to the client’s control request. The context provider then extracts, from the client, a control command directing which sensors are to be controlled. The control request verifies the command’s validity through the rule discriminator and the rule database. The verified command is then forwarded to the query operator, which implements the sensor-control command. The control and query management component receives the event order from each manager by generating database questions and optimizing them according to the control command, then checks on sensor network availability and generates control statements to a sensor node.

**Sensed Data Integrated with Context Awareness**

The sensing-management component receives sensed information from sensor nodes, creates lightweight sensed data, integrates sensed data, analyzes lightweight data through a context-aware recognizer, and manages specific events.

**Weight-Based Sensed-Data Detection**

Achieving valuable information from the large-scale sensed data that is collected from the sensor network requires efficiently identifying the data type. Therefore, to enable this large-scale collection of sensed data and to parse its data packet structure, we propose a novel weight-based sensed-data detection scheme. Here, we describe this scheme in detail in terms of the definitions and theorems used.

**Table 1. Weights for an S(3, 2) system (three sensors, with two cases per sensor).**

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All off</td>
<td>$\omega(0) = 000_{(2)}$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$a_1$ on</td>
<td>$\omega(a_1) = 001_{(2)}$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$a_2$ on</td>
<td>$\omega(a_2) = 010_{(2)}$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$a_3$ on</td>
<td>$\omega(a_3) = 100_{(2)}$</td>
</tr>
<tr>
<td>${a_1, a_2}$</td>
<td>$a_1, a_2$ on</td>
<td>$\omega([a_1, a_2]) = 011_{(2)} = \omega(a_1) + \omega(a_2) = 001_{(2)} + 010_{(2)}$</td>
</tr>
<tr>
<td>${a_1, a_3}$</td>
<td>$a_1, a_3$ on</td>
<td>$\omega([a_1, a_3]) = 101_{(2)} = \omega(a_1) + \omega(a_3) = 001_{(2)} + 100_{(2)}$</td>
</tr>
<tr>
<td>${a_2, a_3}$</td>
<td>$a_2, a_3$ on</td>
<td>$\omega([a_2, a_3]) = 110_{(2)} = \omega(a_2) + \omega(a_3) = 010_{(2)} + 100_{(2)}$</td>
</tr>
<tr>
<td>${a_1, a_2, a_3}$</td>
<td>$a_1, a_2, a_3$ on</td>
<td>$\omega([a_1, a_2, a_3]) = 111_{(2)} = \omega(a_1) + \omega(a_2) + \omega(a_3) = 001_{(2)} + 010_{(2)} + 100_{(2)}$</td>
</tr>
</tbody>
</table>

**Theorem.** Suppose $A$ is the power set of $B = \{a_{11}, a_{12}, \ldots, a_{1(K-1)}, \ldots, a_{N1}, a_{N2}, \ldots, a_{N(K-1)}\}$, $|B| = N(K - 1)$ with cardinality $|A| = 2^{N(K-1)}$, where $N$ and $K$ are positive integers. Then, the elements of $A$ can be labeled with the numbers 0 to $2^{N(K-1)} - 1$ in a binary expansion with $N(K - 1)$ bits. So, the labeling of $A$ is all distinct.

**Proof.** Clearly, our theorem is true, because $N(K - 1)$ bits are needed in a binary expansion to represent decimal numbers 0 to $2^{N(K-1)} - 1$, and vice versa.

**Definition.** Let $C = \{b_1, b_2, \ldots, b_m\}$ be an element of $A$, where $|A| = 2^{N(K-1)}$; that is,

$$C = \{b_1, b_2, \ldots, b_m\} \subseteq B = \{a_{11}, a_{12}, \ldots, a_{1(K-1)}, \ldots, a_{N1}, a_{N2}, \ldots, a_{N(K-1)}\}$$

Next, define a weight ($\omega(t)$) of $C$ as

$$\omega(C) = \omega([b_1, b_2, \ldots, b_m]) = \sum_{i=1}^{m} \omega(b_i)$$

where, if for some $(i, j)$, $b_j = a_{ij}$ then $\omega(a_{ij}) = (000 \ldots 010 \ldots 000)_{(2)}$, and the 1’s position is

$$(K - 1)(i - 1) + j - 1 = (K - 1)(i - 1) + j,$$

$$1 \leq i \leq N, 1 \leq j \leq (K - 1)$$

Now, suppose there are $N$ sensors and $K$ possible detection states, including the case of no detection. We denote this system as $S(N, K)$.

**Example S(3, 2) System.** Suppose we have three sensors ($a_1$, $a_2$, $a_3$), and each can be either on or off. This means $N = 3$ and $K = 2$, and there are two subcases for each sensor. Therefore, binary expansion needs 3 bits to represent all $2^{3(2-1)} = 2^3 = 8$ subcases. Table 1 lists the weight for each case.
In Table 1, each sensor uses 1 bit to represent its status.

**Example S(3, 3) System.** Now, suppose we have three sensors \( a_1, a_2, a_3 \), and each can be on, off, or half on. This means \( N = 3 \) and \( K = 3 \), including the case in which nothing happens; so, there are three subcases for each sensor. Hence, binary expansion needs 6 bits to represent all \( 2^{3(3-1)} = 2^6 = 64 \) subcases, with each sensor using 2 bits to represent its status. Table 2 lists the weight for each case.

**Assigning Weight to Each Case.** When \( N \) sensors with \( K \) subcases for each sensor are given, suppose the system uses \( M = N(K-1) \) bits to represent all cases. Then, each sensor uses \( M/N = N(K-1)/N = K-1 \) bits to represent its own cases (that is, the subcases). Table 3 lists the weight for each subcase.

We obtain the weights of each composition case by summing the weights of its corresponding subcases. Sensed data collection in Lamses falls into one of two categories: periodic or nonperiodic. Periodic collection involves collecting sensed data at given time intervals. Nonperiodic collection can be specified as event, request, and response information collecting. The priority of the sensed-data collecting period is determined by **PacketType** from a packet header of sensed data. For example, when the sensed information turns out to be infrared, magnetic, vibrated, or acoustic (MIC), it consists of one sensor node. The sensor node’s sensing value ranges from 1 to 4. Once sensed information is received, Lamses prioritizes it as follows: infrared = 4, magnetic = 3, vibration = 2, and acoustic (MIC) = 1. When two or more sensors detect the same event, its priority weight is increased to the sum of all the priority weights that were assigned to this event multiplied by the number of sensors that detected it.

**Hamming Code.** Consider an \( S(N, K) \) system in which each sensor can detect a certain number \( (2^K-1) \) of subcases. We encode the weight of each subcase for error correction using Hamming code. Suppose we have an \( S(N, 6) \) system. For each sensor, the coding process is applied, respectively, so that the number of sensors is irrelevant.

The **input** of this system is the weight of \( 2^{K-1} = 2^{6-1} = 2^5 = 32 \) subcases, which is a **message symbol** with length \( K-1 = 6-1 = 5 \) bits. It is denoted by a dimension \( K \) (thus, \( K = 5 \)), which will be modified to \( K' \). This system’s **output** is a set of vectors, and each vector is called a **codeword** with length \( n \). In this case, we have \( r = n - K \) check symbols (the number of bits for the parity check).

Thus, the properties of the Hamming code are as follows:

- Length \( n = 2^r - 1 \) (for \( r = 2, 3, \ldots \)).
- Dimension \( K = 2^r - 1 - r \).

### Table 2. Weights for an \( S(3, 3) \) system (three sensors, with three cases per sensor).

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All off</td>
<td>( \omega(0) = 000000 )</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>( a_1 ) on</td>
<td>( \omega(a_1) = 000001 )</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>( a_2 ) half on</td>
<td>( \omega(a_2) = 000010 )</td>
</tr>
<tr>
<td>( a_3 )</td>
<td>( a_3 ) on</td>
<td>( \omega(a_3) = 000100 )</td>
</tr>
<tr>
<td>( a_4 )</td>
<td>( a_4 ) half on</td>
<td>( \omega(a_4) = 001000 )</td>
</tr>
<tr>
<td>( a_5 )</td>
<td>( a_5 ) on</td>
<td>( \omega(a_5) = 010000 )</td>
</tr>
<tr>
<td>( a_6 )</td>
<td>( a_6 ) half on</td>
<td>( \omega(a_6) = 100000 )</td>
</tr>
<tr>
<td>( a_7 )</td>
<td>( a_7 ) on, ( a_8 ) on</td>
<td>( \omega(a_7, a_8) = 001010 )</td>
</tr>
<tr>
<td>( a_9 )</td>
<td>( a_9 ) on, ( a_{10} ) half on</td>
<td>( \omega(a_9, a_{10}) = 010100 )</td>
</tr>
<tr>
<td>( a_{11} )</td>
<td>( a_{11} ) on, ( a_{12} ) on</td>
<td>( \omega(a_{11}, a_{12}) = 010111 )</td>
</tr>
<tr>
<td>( a_{13} )</td>
<td>( a_{13} ) on, ( a_{14} ) on, ( a_{15} ) on and ( a_{16} ) on</td>
<td>( \omega(a_{13}, a_{14}, a_{15}, a_{16}) = 111111 )</td>
</tr>
</tbody>
</table>

### Table 3. Weights for a system with \( N \) sensors and \( K \) subcases per sensor.

<table>
<thead>
<tr>
<th>Subcase</th>
<th>Weight (with bit positions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( K-1 )</td>
<td>00 ... 0000</td>
</tr>
<tr>
<td>( K-2 )</td>
<td>00 ... 0001</td>
</tr>
<tr>
<td>3</td>
<td>00 ... 0010</td>
</tr>
<tr>
<td>4</td>
<td>00 ... 0100</td>
</tr>
<tr>
<td>( K )</td>
<td>10 ... 0000</td>
</tr>
</tbody>
</table>

*For the last four bit positions, 4 means infrared, 3 means magnetic, 2 means vibration, and 1 means acoustic (MIC).*

- Number of parity checks = \( r \).
- Minimum distance = \( d = 3 \).

Hamming code is perfect single-error-correcting code and is unique up to equivalence. Let \( H \) be the \( (r \times n) \) parity check matrix of this Hamming code, and let all of its columns be nonzero \( r \) tuples. In this case, \( H \) is \([n, k, d]\) linear code. Then, the calculation of the \( r \) needed for our \( S(N, 6) \) system is as follows:

Given that \( K = 5 \) and \( K = 2^r - 1 - r \), we have \( 5 = 2^r - 1 - r \). Also, when
Context-Aware Middleware

For our system, we choose \( r = 4 \) for our system. Hence, \( n = 2^4 - 1 = 15 \), and \( K' = 2^4 - 4 - 1 = 11 \), which is greater than \( K \). Therefore, we have the (4 × 15) parity check matrix \( H \) shown in Figure 2a, and the (11 × 15) generator matrix \( G \) shown in Figure 2b, where \( A \) is the matrix whose columns are all possible codes of length 4 except 0000, and \( A^T \) is the transpose matrix of \( A \).

Figure 2. Hamming-code-related matrices for our S(\( N, 6 \)) system: (a) the (4 × 15) parity check matrix \( H \) and (b) the (11 × 15) generator matrix \( G \), where \( I \) is the identify matrix, \( A \) is the matrix whose columns are all possible codes of length 4 except 0000, and \( A^T \) is the transpose matrix of \( A \).

Detecting and Correcting Messages.
Suppose the message \( u = (u_1u_2 ... u_K) \) is encoded into the codeword \( x = (x_1x_2 ... x_n) \). The received vector \( y = (y_1y_2 ... y_n) \) can be different from \( x \).

Definition. The error vector is given by \( e = y - x = (e_1e_2 ... e_n) \).

Definition. The Hamming distance between two vectors \( x = (x_1x_2 ... x_n) \) and \( y = (y_1y_2 ... y_n) \) is the number of places in which they differ and is denoted by \( \text{dist}(x, y) \).

Definition. The Hamming weight of a vector \( x = (x_1x_2 ... x_n) \) is the number of nonzero \( x_i \) and is denoted by \( \text{wt}(x) \).

Definition. Let \( C \) be \([n, K]\) linear code (set of codewords) over a field codewords \( x \). Then, \( x = uG \) is the code. So, we have \( 2^K - 2 = 2^{11} - 2^5 \) dummy subcases.

Detecting and Correcting Messages.
Suppose the message \( u = (u_1u_2 ... u_K) \) is encoded into the codeword \( x = (x_1x_2 ... x_n) \). The received vector \( y = (y_1y_2 ... y_n) \) can be different from \( x \).

Definition. The error vector is given by \( e = y - x = (e_1e_2 ... e_n) \).

Definition. The Hamming distance between two vectors \( x = (x_1x_2 ... x_n) \) and \( y = (y_1y_2 ... y_n) \) is the number of places in which they differ and is denoted by \( \text{dist}(x, y) \).

Definition. The Hamming weight of a vector \( x = (x_1x_2 ... x_n) \) is the number of nonzero \( x_i \) and is denoted by \( \text{wt}(x) \).

Definition. Let \( C \) be \([n, K]\) linear code (set of codewords) over a field
with two elements. For any vector \( a \), the set \( a + C = \{ a + x | x \in C \} \) is called a coset (or translate) of \( C \). Each coset contains \( 2^k \) vectors. If the decoder receives the vector \( y \), then \( y \) must belong to some coset, \( y = a_i + x \ (x \in C) \). If codeword \( x' \) was transmitted, the error vector is \( e = y - x' = a_i + x - x' \). Therefore, we deduce that the possible error vectors are exactly the vectors in the coset containing \( y \).

**Syndromes.** There is an easy way to find which coset \( y \) is in.

We simply compute the vector as \( S = Hy^T \), which is called the syndrome of \( y \). The properties of the syndrome are as follows. First, \( S \) is a column vector of length \( n - k \). Second, for error detection, the syndrome of \( y \) (\( S = Hy^T \)) is 0 if and only if \( y \) is a codeword. If no errors occur, the syndrome of \( y \) is 0, but not conversely. In general, if \( y = x + e \), then

\[
S = Hy^T = Hx^T + He^T = He^T
\]

Given \( y \), we choose a minimum weight vector \( \hat{e} \) in the coset containing \( y \), and we decode \( y \) as \( \hat{x} = y - \hat{e} \). The minimum weight vector in a coset is called the coset leader. (If there is more than one vector with the minimum weight, we choose a random weight and call it the coset leader.) We then perform the following steps:

1. Modify the weight of \( 2^5 \) subcases to \( [u_1 u_2 \ldots u_6], K = 11 \) by attaching six zeros at the end.

Figure 4. Lightweight XML document for a sensed data packet.
Using \( x = uG \), obtain the code-words \( x = (x_1, x_2, \ldots, x_n), n = 15 \). \( C \) is the set of these codewords. (In our case, we actually have only 2^5 codewords, but there should really be 2^{11}.)

2. Find the cosets \( a_i + C = \{a + x \mid x \in C\} \) by evaluating each coset, one by one, and find the coset leader.

3. For coset leader \( a_i \), find the syndrome for each coset by using \( S = Ha_i^T \). Assign these syndromes to each respective coset.

4. When \( y = (y_1, y_2, \ldots, y_n) \) is transmitted, find its syndrome using \( S = Hy^T \). If the syndrome of \( y \) (\( S = Hy^T \)) is 0; then \( y \) is a codeword, indicating that no errors have occurred. If not, find the corresponding coset and coset leader for that syndrome, and accordingly correct \( y \) by \( \hat{x} = y - \hat{e} \), where \( \hat{e} \) is the coset leader; in this case, \( \hat{x} \) is the corrected message.

**Sensed-Data Collector and Weight-Lightening Algorithm**

The *sensed-data collector* for the sensing-information process collects large-scale sensed data and uses the weight-lightening algorithm to make this data lightweight. The sensed-data collector then stores this data into a lightweight XML document. It collects each sensor’s transmitted sensed data periodically or nonperiodically for the weight-lightening process of periodic and nonperiodic sensed data. The sensed-data collector has five parts: the sink scheduler, temporary data storage, the packet parser, the data coordinator, and the data converter.

The sink scheduler collects sensed data periodically or nonperiodically, using the weight-lightening algorithm on a priority basis. Collected large-scale sensed data is then saved in temporary data storage, which contains a queue that organizes sensed data. When the queue reaches some limit, another queue stores the data. When that queue is full, the data is sent to the packet parser, which dissects the data by its attributes, extracts the necessary information, and sends this information to the data coordinator. Then, the data coordinator creates an XML document object model (DOM) for each sensor using the attribute data and sends this XML DOM to the data converter. Finally, the data converter takes this XML DOM and converts it into an XML document.
Weight-Lightening Scheme

When a data packet is created from numerous sensor nodes, it is sent through a sink to Lamses, which processes the given data packet. Figure 3 (on p. 54) shows the weight-lightening process and structure of the sensed data packet collected through the weight-lightening algorithm.² This process decreases Lamses’ load and ensures effective sensed-data management. After receiving a data packet, Lamses extracts the necessary attributes using the parser. Then, through the transfer process, Lamses generates the lightweight XML document from those attributes.

A sensed data packet consists of PacketType, PacketNum, Sink_ID, Sensor_ID, PSensor_ID, PayloadNum, and Payload. PacketType stores packet information such as period, request, response, and event. PacketNum sets the number of packets created from the given sensor node.
Sink_ID sets the sink ID information of the given sensor node. Sensor_ID sets the ID information of the sensor node where the given data packet was found. PSensor_ID stores ID information of a parent sensor node when it exists. When there is no parent sensor node, PSensor_ID stores ID information of the sensor node itself. PayloadNum sets the number of payloads of sensing information that comprise the sensor node. Profile_ID of Payload stores ID information of the sensor node. SubProfile_ID stores ID information of a sensor at a given sensor node. DataLength stores the length of data, and Data stores information collected by a sensor.

Figure 4 (on p. 55) shows the contents of a lightweight XML document. This document contains the sinks, sensor nodes, number of sensing-information payloads, packet type of the sensed data, and semantic information of the sensed data.

Context-Aware Recognizer
The context-aware recognizer analyzes the lightweight XML document created by the sensed-data collector and lightener, and examines the data events. It then stores the verified data in the database and creates an event for the sensor node in which the events were found. The context-aware recognizer includes the following parts: the sensed-data extractor, the data parser, context-aware information, the context-aware engine, event extraction, and the repository. Context-aware information contains the sensing log of each sensor for status recognition. The sensed-data extractor uses the minimum memory necessary to parse a document, employing a SAX (Simple API for XML) parser to process and extract sensed data from a large-scale XML document. Extracted sensed data is then sent to the data parser, which verifies the value of the received sensed data. The context-aware engine receives the data from the data parser and compares it with the context-aware information to determine whether this data contains errors.

When the sensed data includes event information, it is stored in the repository. Then, this event information is sent to event extraction, which checks the number of payloads for the received sensed data and extracts information indicating that a given event has occurred and by which sensor. The extracted event calls the state management component through the data transmitter and implements the state management
process. Figure 5 (on p. 56) shows the Lamses control flow.

**Implementation**

Lamses has been implemented in a ubiquitous Korean Army defense system to detect external intruders. Figure 6 (on p. 57) shows snapshots of this implemented Lamses server.

Figure 6 also shows a meta-information database management view, which controls the sink list and the sensor node forming the network at each sink. Meta-information database management provides six menus. The sensor network menu is for inputting metadata with the location of a sink node or sensor node. The sensor node hardware specification menu is for inputting the hardware specification of a sensor used in a sensor node. The transducer menu is for inputting sensor meta-data. The transducer hardware specification menu is for configuring sensor hardware data. The sensing-type specification menu is for configuring the part of an initial setting that enables running the sensor hardware.

Lamses was also applied to a Korean Army military unit as an automatic surveillance system to detect intruders (see Figure 7). In this implementation, we experimented with approximately 100 sensors, along with five or six sink nodes capable of detecting magnetic field, oscillation, sound, and heat. In Figure 7, the surveillance system activates an emergency alert to a military unit.

In the future, we hope to extend Lamses so that it can work with various wireless communication protocols, such as ZigBee, Bluetooth, wireless local area networks (WLANs), and code division multiple access (CDMA), as well as various sensor node types such as Mote, Nano, and NeurFon. Our goal is to develop independent USN application services that have not been addressed anywhere else thus far.

**Acknowledgments**

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea and was funded by the Ministry of Education, Science, and Technology (no. 2009-0066815) and the IT R&D program of the Ministry of Knowledge Economy, Korea Evaluation Institute of Industrial Technology (10033915, adaptive fusion technology for large-scale sensor-node-based intelligent surveillance systems).

**References**


Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.