PAPER Special Section on Parallel and Distributed Computing and Networking

# Rule-Based Sensor Data Aggregation System for M2M Gateways

**Yuichi NAKAMURA**[†,††*a)], **Akira MORIGUCHI**[††], **Masanori IRIE**[††], **Taizo KINOSHITA**[†††], ***Nonmembers***, ***and*** **Toshihiro YAMAUCHI**[†], ***Member***

**SUMMARY** To reduce the server load and communication costs of machine-to-machine (M2M) systems, sensor data are aggregated in M2M gateways. Aggregation logic is typically programmed in the C language and embedded into the firmware. However, developing aggregation programs is difficult for M2M service providers because it requires gateway-specific knowledge and consideration of resource issues, especially RAM usage. In addition, modification of aggregation logic requires the application of firmware updates, which are risky. We propose a rule-based sensor data aggregation system, called the complex sensor data aggregator (CSDA), for M2M gateways. The functions comprising the data aggregation process are subdivided into the categories of filtering, statistical calculation, and concatenation. The proposed CSDA supports this aggregation process in three steps: the input, periodic data processing, and output steps. The behaviors of these steps are configured by an XML-based rule. The rule is stored in the data area of flash ROM and is updatable through the Internet without the need for a firmware update. In addition, in order to keep within the memory limit specified by the M2M gateway's manufacturer, the number of threads and the size of the working memory are static after startup, and the size of the working memory can be adjusted by configuring the sampling setting of a buffer for sensor data input. The proposed system is evaluated in an M2M gateway experimental environment. Results show that developing CSDA configurations is much easier than using C because the configuration decreases by 10%. In addition, the performance evaluation demonstrates the proposed system's ability to operate on M2M gateways.

***key words:*** *M2M gateway, sensor data aggregation, in memory processing, IoT (the Internet of Things)*

## 1. Introduction

With the growth of machine-to-machine (M2M) technology, many companies have begun to offer M2M services, which provide new value by utilizing machine sensor data. For instance, construction and agricultural machine manufacturers provide remote monitoring services for their products by sending sensor and location data from products in the field to servers over the Internet [1]–[3]. These services reduce downtime and prevent machinery theft. In M2M services, sensors connect to local networks, such as Zigbee [4] or a controller area network (CAN) [5]. As a result, they cannot upload data directly to the Internet. In order to create a bridge between sensors and the Internet, M2M service providers utilize devices called M2M gateways. An M2M gateway is able to communicate with both the sensor network protocol and the IP [6]. For example, when a remote monitoring service is used for construction machinery, an M2M gateway is attached to the machinery, and it gathers data from the sensors via the CAN. Then, it uploads the data to a server via a wireless Internet connection. However, cost becomes an issue because wireless Internet contracts are usually measured rates, and server resource increases with an increase in the quantity of data. In addition, the number of M2M gateways can be in the tens of thousands, sometimes millions, and the quantity of data uploaded to the server also increases, substantially raising the cost. In order to reduce this cost, M2M service providers must reduce the quantity of data transferred between the M2M gateway and the server.

There are three main approaches used to reduce the quantity of data transferred between an M2M gateway and the server. The first approach reduces the protocol header overhead by using a lightweight protocol. HTTP is a popular means of communication between the server and the client. However, in M2M systems, HTTP overhead becomes an issue because the sensor data size is often about 100 bytes, whereas the HTTP header size is greater than 100 bytes. To reduce the overhead, lightweight protocols, such as MQTT [7], WebSocket [8], and CoAP [9], are effective because their protocol header size is around 10 bytes [10]. In addition to this protocol-based approach, the quantity of data can be reduced before being sent from the M2M gateway.

The second approach reduces the data quantity by processing the data on sensor nodes. TinyDB [11] processes a language similar to SQL on sensor nodes. SensorWare [12] also has a script language for sensor nodes. By using these systems, only the necessary data are sent from the sensors, thereby reducing the quantity of data sent from the M2M gateway. These technologies are suitable for controlling devices within sensor networks because their response times are fast: when a sensor detects an event, an action is immediately issued from the sensor. However, these technologies are not suitable for reducing the quantity of data transferred between an M2M gateway and the server for two reasons. First, these systems are not intended for processing data from multiple sensors. It is efficient to gather a

Manuscript received January 8, 2016.
Manuscript revised May 14, 2016.
Manuscript publicized August 24, 2016.

[†]The authors are with Okayama University, Okayama-shi, 700–8530 Japan.
[††]The authors are with Hitachi Solutions, Ltd., Tokyo, 140–0002 Japan.
[†††]The author is with Hitachi, Ltd., Tokyo, 101–8608 Japan.
[*]Presently, with Hitachi, Ltd. and Okayama University.
a) E-mail: yuichi.nakamura.fe@hitachi.com

DOI: 10.1587/transinf.2016PAP0020

sensor's data depending on another sensor value; however, these technologies do not support such complex processes. Second, the installation and management costs are substantial. Data processing middleware and configurations need to be installed on all sensor nodes. If there are 100 sensors per machine and 10,000 machines, the number of sensors required is one million. Installing and managing software for this many machines is difficult.

For such reasons, M2M service providers take the third approach, i.e., data aggregation on the gateway, for example by taking distributions and averages from sensors on M2M gateways [13]–[15]. In this approach, the installation of aggregation logic is restricted to the M2M gateway, which is easier than installing the logic on the sensors. Moreover, standardized technologies, such as TR-069 [16], used for managing software and configuring gateway devices, contribute to a reduction in installation and maintenance costs. M2M service providers can write programs for the sensor data aggregation, but there are two issues with this. First, there is the problem of developing programs on firmware for resource constrained environment of industrial usage. Since cost and reliability requirements for industrial M2M gateway are high, and gateways are also sometimes battery powered, CPU and RAM specifications for M2M gateways are much lower than enterprise servers. For example, battery powered M2M gateways to monitor construction machines, CPU is often 200Mhz-600Mhz, RAM is 1-64Mbytes, and 1Mbps CAN is used as a field area network protocol. For such resource constrained environment, the programming language for firmware development is usually C. Programming in C requires bothersome memory management, an issue that is hidden in higher-level languages. In addition, M2M service providers need to learn the development environment, such as the compiler and tool chains, which strongly depends on the gateway because M2M gateways are not standardized. Attention also needs to be given to the CPU and RAM usage of the aggregation program, particularly the matter of RAM usage. The input data rate cannot be predicted, and sufficient memory must be allocated in advance. However, when the input rate increases suddenly and too much memory is allocated, the system will crash because M2M gateways often do not have swap. Consequently, it is difficult for M2M service providers to write aggregation programs on M2M gateways. The second problem arises when changing the data aggregation logic on the firmware. In order to change the data aggregation logic, the firmware must be updated. However, firmware updates are risky because the gateway may not work if the update fails.

We propose a framework called a complex sensor data aggregator (CSDA) to aggregate sensor data on M2M gateways. CSDA enables M2M service providers to develop data aggregation logic without programming in C. The CSDA defines the framework that supports the sensor data aggregation. It splits the aggregation process into input, data processing, and output steps, whose behaviors are controlled by a configuration file. The aggregation logic can also be changed by simply updating the configuration file and restarting the CSDA. In addition, to work in memory-constrained environments, a static aggregation processor (where working memory size and number of threads are fixed) is launched at startup time, and the working memory size can be adjusted by configuring a sampling setting.

This paper presents the design of the CSDA and evaluates the proposed solution using an M2M gateway evaluation board. In summary, we offer the following contributions. Since a preliminary work of this paper was presented at proceedings [17], differences from the preliminary work are also described.

- We subdivide the components of the data aggregation logic into the categories of periodic statistical calculation, filtering, and data concatenation, and define a framework based on this categorization, which enables development and updating of data aggregation logic without C programming. This contribution is basically same as a preliminary work, although there are some refinements.
- We identify issues involved in implementing sensor data aggregation logic for a memory-constrained environment, e.g., the fact that memory usage needs to be within the size specified by the M2M gateway's manufacturer and should not change after startup. We note that in particular, the buffering area for statistical calculations can easily break the memory usage limit because of the unpredictability of the input rate. For our framework, we designed a static aggregation processor that is launched according to the configuration specified and is composed of threads and working memory whose size does not change after startup. We also developed a buffering technique in the processor that limits its memory consumption for unpredictable rates of data input by utilizing data sampling. This contribution is newly developed since [17], and a prototype was also newly implemented.
- We describe our implementation of a CSDA prototype including the above framework, processor, and buffering technique, and we evaluate the CSDA's effectiveness by using an evaluation board whose CPU clock is 400 mhz, RAM size is 64 Mbytes, and CAN network is used as a sensor network. Results show that developing CSDA configurations is much easier than using C because the configuration decreases by 10%. In addition, the performance evaluation demonstrates the proposed system's ability to operate on resource-constrained M2M gateways, i.e. the increase in memory usage compared with dedicated C logic is about 70 Kbytes, and memory usage can also be reduced by adjusting the sampling setting. This prototype also creates the basis of a commercial product called the Entier Stream Data Aggregator [18] and is being used in actual M2M gateways.

The remainder of this paper is organized as follows: Sect. 2 presents an overview of data aggregation process on M2M gateway and issues in developing aggregation logics.

Section 3 describes the design and implementation of our proposed system called CSDA. Section 4 describes an evaluation of the proposed system. Section 5 discusses previous works. Finally, Sect. 6 concludes this work.

## 2. Issues in Aggregation of Sensor Data

In the following sections, after the functions in the process of sensor data aggregation for M2M gateways are subdivided into categories, the issues involved in writing the aggregation logic are described.

### 2.1 Categories of Functions in the Process of Sensor Data Aggregation on an M2M gateway

#### 2.1.1 Process of Sensor Data Aggregation

Before the functions in the sensor data aggregation process can be subdivided into categories, the process of inputting sensor data into the M2M gateway needs to be described. An overview of an M2M system is shown in Fig. 1. Each sensor node in the sensor network is connected to sensor chips through a circuit. Sensor nodes are composed of a physical sensor network interface and a CPU, in which an embedded program fetches a value from the sensor chips and sends that value to the sensor network with the node's ID. The node ID identifies that node in the sensor network. The arbitration ID for the CAN protocol and the IEEE address for Zigbee are examples of node IDs. Note that multiple sensors may be connected to the same sensor node; in this case, multiple values are packed with that node's ID. The means by which fetched values are packed with node IDs, i.e., short or long, little-endian or big-endian, depend on the sensor node vendor. The M2M gateway and sensors are connected by various types of networks, and data are inputted into the gateway. For example, because the control of vehicles, industrial machinery, and medical equipment requires real-time communication, the CAN protocol, which has priority control features, is often used. When measuring temperature and humidity in a large area, such as a plant or building, the wireless protocol Zigbee is used in order to eliminate the cost of installing wires. M2M gateways have physical interfaces and device drivers in order to communicate with the sensor network protocol. They receive data frames, which include the previously described node IDs and sensor values.

The sensor data aggregation process runs on an M2M gateway, using node IDs and sensor values as input. The functions in the process can be subdivided into three categories: (1) periodic statistical calculations to reduce data quantity, (2) filtering to reduce data frequency, and (3) concatenation to reduce communication overhead.

#### 2.1.2 Periodic Statistical Calculations

The quantity of data is reduced by calculating statistical values, such as averages and occurrence counts, from multiple data periodically within a given time frame. For example, temperature sensor data for the cooling of water and oil within construction machinery may be summarized by the sum, average, minimum, maximum, and value occurrence counts as taken at specified intervals (such as every 100 ms or every second). These processed data are then sent to a server [13]. Similarly, engine speed and temperature sensor data of farm machinery may be condensed to the average, minimum, and maximum and then sent to the server [15]. In addition, various statistic values of sensor data are utilized, such as standard deviation [19], moving average [20] and Fourier transform [21]. Furthermore, within a periodic process, statistical calculations can be performed multiple times, and multiple sensor data can be combined. In the evaluation of deviation from the stable state of construction machinery, variances are calculated for multiple temperature and pressure sensor values at given intervals, and then the variances are summed [14].

#### 2.1.3 Filtering

The quantity of data from an M2M gateway can be reduced by filtering out the low priority data. Filtering is categorized into two types: input filtering using node IDs and threshold filtering.

- Node ID filtering: To obtain important sensor values, the M2M gateway only takes the data frame, which includes the specific node ID. For example, since engine and transmission sensor values in construction machines are the ones used to detect failure, only these sensor values are gathered and sent to the server [13].
- Threshold filtering: The purpose of this filtering is to gather only the values that indicate trouble. For example, when a statistical value calculated from a construction machine's sensor data exceeds some threshold, the calculated value is sent to the server. Otherwise, no data are sent [14].

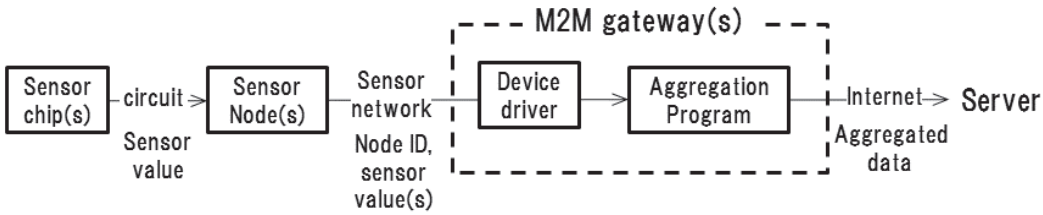


**Fig. 1** Overview of M2M system

#### 2.1.4 Concatenation

When data are sent to the server, protocol header information is attached. To reduce the overhead, multiple sensor data are joined and sent together. The timing of this depends on the type of data being collected and the reason for their collection. Referring again to the example of construction machinery, a summary of operation logs may be sent daily, and any engine sensor data indicating failure can be sent immediately [22].

However, additional statistical calculation combining results of previous periodic statistical calculations is not usually performed in M2M gateways because it consumes computing resource; this is normally the task of a server side application and is not within the scope of the aggregation process.

### 2.2 Issues with Sensor Data Aggregation Using Firmware Programming

The sensor data aggregation process previously described is programmed into the firmware of the M2M gateway. However, there are difficulties in developing and updating programs in firmware.

#### 2.2.1 Developing Firmware in Resource-Constrained Environment

For industrial usage, cost and quality requirements for M2M gateways are strict because they often operate in extreme environments, sometimes operate in battery powered environment and the number of deployed devices is large. As a result, memory and CPU resources are much more constrained than on enterprise server. In addition, since product life cycle is longer than consumer devices, the growth of CPU and memory is slow. A CPU clock for an M2M gateway is often around 200-600 MHz, and it usually has about 1-64 MB of RAM. For example, RAM size for a M2M gateway to monitor construction machines [23] is only 2-8Mbytes, and field area network is 1Mbps CAN. CAN is fast and RAM resource becomes especially limited. Second example is an industrial controller with M2M gateway functionality [24] to monitor production lines. RAM size for user program is 8-16 Mbytes and 10Mbps industrial Ethernet is used as a field area protocol. Another example can be found in general purpose industrial communication board, with a CPU clock of 250 MHz and 64 MB of RAM [25]. There are two issues with developing aggregation programs on firmware in a resource-constrained environment.

The first issue is the limitation of the SDK. Because it is difficult to use higher-level languages on this sort of hardware, the programming language is usually C. However, C is difficult to program, and the handling of memory is bothersome. Moreover, SDKs for M2M gateways vary depending on the gateway device. The development process typically consists of the following steps: writing aggregation logic in C on a PC, cross-compiling it for the gateway, producing a firmware image, and transferring it to the M2M gateway's flash memory. All of these processes are different depending on the M2M gateway because hardware, OS, and userland programs are not standardized.

The second issue is with managing limited memory, i.e., keeping within the limits specified by the M2M gateway's manufacturer and stable memory usage. In an M2M gateway, an out-of-memory condition can easily happen because the RAM size is often small and no swap space is prepared. If one application uses a lot of memory, other applications may not work because of the memory shortage. To avoid this problem, manufacturers of M2M gateways ask application developers to use a specified amount of memory and to test the memory usage of applications in advance. Therefore, memory usage of the aggregation logic must be kept within specified limits. In addition, the usage must also be static, i.e., the memory usage must be fixed at startup time and cannot change.

However, it is not easy to keep within a memory limitation because the data input rate is not stable. Buffer management for performing periodic statistical calculations is especially important. Data need to be stored in a buffer in order for statistical calculations to be performed for a specified time span. For example, when the average of temperature sensor data for 30 seconds is being calculated, all data generated from the temperature sensor during those 30 seconds are stored in buffer. The frequency of data generation varies depending on the configuration of the machine controlling the sensors. In sensors for a particular engine control unit, for example, the temperature sensor data are generated every second in its usual state, but when the engine begins to work hard, the control unit generates sensor data every 0.1 seconds. To process statistical calculation for sensor data arriving at such variable frequency, it would be easily handled if buffer were allocated dynamically, but memory usage needs to be static for M2M gateways. Therefore, buffers for statistical calculations must be allocated statically. Typically, static memory regions are managed by using a ring buffer [26]. However, statistical information will be lost when input traffic increases. For example, an array of size 3,000 is allocated for a ring buffer to calculate the 30-second average for a sensor. When the frequency of input data is every 10 ms, the 3,000-element array is fully used and the 30-second average is calculated without a problem, but when the frequency increases and exceeds every 10 ms, statistical information will be lost. At a frequency of every 1 ms, only the last 3 seconds of data will be stored in the buffer, and the average for 30 seconds cannot be calculated. To prevent such a case, the size of the ring buffer needs to be large enough, but this would exceed the memory limit.

In the above example, note that all data do not necessarily have to be saved to calculate only average, because it can be obtained from the sum and the number of data. However, in order to calculate other statistic values such as moving average, deviation and Fourier transform, all data within a specified time range have to be saved. In addition,

in calculating only average it is preferable to save all data in a buffer to reduce context switch overhead between data receiving process and data calculation process.

#### 2.2.2 Updating Firmware

In order to modify the aggregation logic after the M2M gateways are deployed, the firmware needs to be updated. However, there are difficulties in modifying programs, installing firmware, and recovering from a failure.

- Modifying programs: The C program needs to be modified in order to change the aggregation logic; this involves difficulties similar to those previously described for the development of the C program. In addition, the C program requires repeat testing in order to avoid regression.
- Installing new firmware: After new firmware is developed, it must be installed on the devices. There are two ways of doing this. In the first method, field engineers perform the installation. Although this is the most reliable method, it is very expensive when the number of devices is 10,000 or 100,000 or more. To reduce this human cost, the second method is used, in which the updating is performed through the network. However, network costs are still an issue in this scenario. Since firmware is usually implemented on read-only file systems, entire file system images, which often exceed 10 MB, must be delivered via the mobile network, and the network cost over all the devices becomes impossible to ignore.
- Recovering from a failure: When the writing of firmware from the network fails because of a power outage during the update, the firmware is broken. To avoid this, a backup area, whose size is the same as the firmware's, is a necessary component of the flash ROM. In addition, the recovery procedure must be performed by an engineer.

## 3. Proposal of Complex Sensor Data Aggregator

### 3.1 Basic Design of the CSDA

As described in the previous section, there are problems with programming and updating sensor data aggregation logic for M2M gateway firmware. Here, memory needs to be carefully managed. The memory usage of the aggregation logic must remain within the limit specified by the M2M gateway's manufacturer, and it should not increase after the aggregation logic begins to work.

In this section, we propose a framework called CSDA as a solution to these problems. An overview of the CSDA is shown in Fig. 2. The process of aggregation is specified in a configuration file, thus eliminating the need for programming in C. Typical logic components such as those for averaging and filtering are embedded in advance, and the

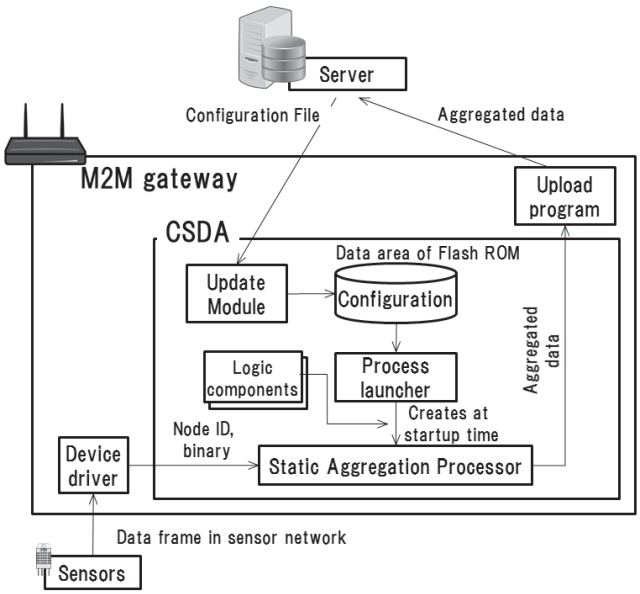


**Fig. 2** Architecture of the CSDA

combination of components to handle the required aggregation tasks is described in the configuration file. The process begins with the *process launcher*, which starts the *static aggregation processor*, which is responsible for the aggregation process described in the configuration file. In order to prevent an increase in memory usage after startup, the static aggregation processor is composed of static working memory and a static number of threads, which handle the aggregation tasks by combining embedded logic components. In addition, to enable the adjusting of memory usage within the specified limits, the working memory size can be reduced by modifying the sampling configuration of a buffer, the details of which are described in Sect. 3.2.3. The *update module* changes the configuration where aggregation logic is described, via the network. Here, it is assumed that the memory usage of the CSDA has been tested in a test environment and found to be within the limits before the configuration is changed. The detailed design of the static aggregation processor, configuration file, and update module are described in the following sections.

### 3.2 Design of Static Aggregation Processor

Figure 3 shows the design of the static aggregation processor. Three types of threads aggregate sensor data according to a specified configuration. The number of threads is defined by the configuration and does not change after startup, thereby preventing an increase in memory usage. Buffers between threads, called the *sampling buffer* and the *processing cache*, are also designed to control memory usage; the sampling buffer in particular has the ability to adjust memory usage. The design of the threads and buffers is described next.

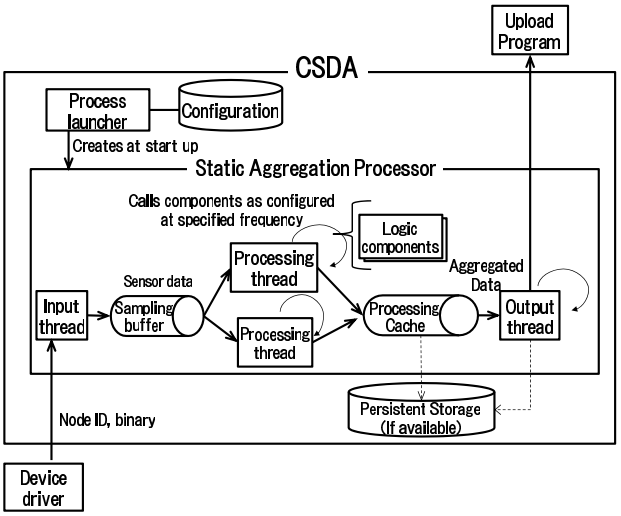


**Fig. 3** Design of Static Aggregation Processor

### 3.2.1 Threads for Aggregation Process

In order to cover the categorized functions of the aggregation process as described in Sect. 2.1, the static aggregation processor handles sensor data in three steps: input, data processing, and output. These are implemented in the following three kinds of threads:

- Step1, input thread: One thread is launched at startup time. In this thread, node IDs and binary data, including sensor values, are passed from the device driver, and the node IDs are filtered out. Then, sensor values are extracted from the binary code and buffered in a memory area called the *sampling buffer*.
- Step2, data processing threads: This step takes data from the sampling buffer periodically at a configured rate (e.g., every 100 ms), and statistical calculations, such as the average, sum, minimum, maximum, and occurrence counts, are performed on the extracted sensor values. Threshold filtering is also performed on the sensor values and the calculated values. The two tasks, statistical calculation and threshold filtering, can be combined. The results of this data processing are saved in a memory area called the *processing cache*. The number of threads launched at startup depends on the variations of the frequency of data processing specified in the configuration file. For example, if data processing is configured for every 100 ms and every 10 ms, two threads are launched, and no new threads are launched after that.
- Step3, output thread: It reads the result data of the processing thread from the processing cache and concatenates them; then, the data are periodically passed to the data upload program, or passed immediately if the data indicate an event requiring urgent attention. In addition, if persistent storage is available, data can be stored there. The number of threads launched at startup also depends on the variations of the frequency of output processing specified in the configuration.

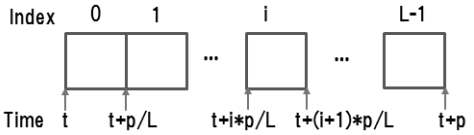


**Fig. 4** Array usage of sampling buffer

### 3.2.2 Memory Management Strategy

To ensure that memory usage remains within the CSDA's limit, the entire working memory size must be fixed at startup. However, the frequency of input data for processing threads and aggregated data from processing threads increases as sensor data traffic increases, and dynamic memory allocation can easily violate the memory usage limit as described in Sect 2.2.1. Therefore, for input and output of processing threads, static working memory area is prepared in order to establish the memory usage. The size of the static working area is defined by the CSDA's memory usage, which is the amount allowed by the manufacturer of the target M2M gateway. From the working area, a *sampling buffer* is allocated for input, and a *processing cache* is allocated for output at startup according to the configuration specified, and the allocated memory size does not change after that. The configuration needs to be adjusted not to consume the entire working memory area because startup will fail if the working memory area is fully used. The sampling buffer in particular is designed to adjust the memory usage because the quantity of sensor data input can become very large. The design of these buffers is described next.

### 3.2.3 Design of Sampling Buffer

As described in Sect 2.2.1, statistical information is lost when a simple ring buffer is used as the input buffer. To reduce such loss of statistical information and to enable memory usage adjustment, not all input data are saved in the sampling buffer when the frequency of data increases; instead, data are sampled at a configured rate. For example, when the sampling buffer is configured to save data every 100 ms and input data come every 10 ms, data are saved every 100 ms. This memory usage can be reduced by configuring the sampling rate because the amount of memory needed is smaller when the sampling rate is low.

Figure 4 shows the details of the sampling buffer. At CSDA startup, the sampling buffer is allocated from the static working memory area for each input sensor from the working area; the array layout is as shown in the figure. Here, $p$ is the time period of the stored data and $L$ is the length of the array; their values are specified in the CSDA configuration file. At index $i$, representative data for the time between t + (i - 1)*p/L and t + i*p/L are stored. The representative data can be configured from the maximum value,

minimum value, and latest value, and p/L is the sampling rate. For convenience, the working of this array may be explained by an example. Assume that data are stored from sensor A for 30 seconds (p = 30), L is configured as 300, and the sampling rate p/L is 0.1 seconds. In array element 0, representative data between t and t + 0.1 seconds are stored; in array element 1, representative data between t + 0.1 and t + 0.2 seconds are stored; and in array element i, representative data between t + i*0.1 and t + (i + 1)*0.1 seconds are stored. When the time reaches t + 30, data are stored from i = 0, and t is incremented by 30. In this way, data from sensor A are stored every 0.1 seconds for 30 seconds. Then, in the data processing step, data are taken from the buffer and used to perform the statistical calculations.

Developers of aggregation logic in CSDA can adjust *L* according to the memory usage limits of the target M2M gateways. For example, assume that 2 bytes of sensor data are averaged for 20 seconds and that 200 Kbytes of CSDA's working memory is available for the sampling buffer. In this case, the maximum value of *L* will be 10,000, and the maximum sampling rate will be 0.1 ms. On the other hand, when only 2 Kbytes of working memory is available for the sampling buffer, the maximum value for *L* will be 100 and the sampling rate will be 10 ms.

#### 3.2.4 Design of Processing Cache

This cache is used to store the result from the data processing step. There are two uses of this cache. The first one is simply to store the result of the data processing step and pass the result on to the next output step. The second use is to compute statistical values such as frequency counts over long time periods. For example, when a value is calculated every 100 ms and a frequency distribution for the value is created in the data processing step, in order to create a frequency distribution covering 1 minute, the frequency data must be stored in the processing cache.

The cache is also allocated from the static working memory area, and the allocation size is specified in the configuration file. When the cache becomes full, old cached data are removed or cached out to persistent storage if it exists. If data loss is not permitted, the cached-out data can also be sent to the server immediately.

### 3.3 Configuration Language

A configuration language was designed to describe aggregation for the CSDA. XML was adopted as the text representation for the configuration language because it can be easily parsed and handled by other programs such as configuration tools and GUIs. However, it is difficult to handle an XML-based text configuration file on an M2M gateway because the XML parser usually consumes memory resource. To save memory, an XML-based configuration is encoded into a binary format outside the M2M gateway, and it is delivered and then loaded to the CSDA at startup. The configuration file for the CSDA consists of three sections,

```
# Obtain data frame whose node ID is 1
1:<separator separatorId="1">
# Extract sensor data from data frame
# and identify them as 100,101
2:<offsetRawData rawDataId="100", dataType="short",
       length="8", offset="0"/>
3:<offsetRawData rawDataId="101", dataType="short",
       length="8", offset="8"/>
4:</separator>
# Configuration for sampling buffer: p is configured as 4000ms
# L is configured as 400
5:<rawCacheSetting size="400",period="4000",timeUnit="msec"/>
```

**Fig. 5** Part of configuration for input step

corresponding to the steps in the aggregation process as described in Sect 3.2.1, i.e., input, data processing, and output.

#### 3.3.1 Input Section

In this section, the node ID filtering, target sensor values, and configuration of the sampling buffer are described. Figure 5 shows an example of this configuration. Here, the *separatorId* attribute in line 1 refers to the node ID filtering. Only the input data whose node ID is described in separatorId are processed. In this example, only the data frames with node IDs of "1" are processed. The *offset* and *length* attributes correspond with the data extracted from the data frame payload. In line 2, 0 to 7 bits of payload are extracted and stored as a short data type (2 bytes) in a sampling buffer. Similarly, 8 to 15 bits of payload are extracted and stored in a sampling buffer. In order to identify the extracted data, the *rawDataId* attribute is used. Data extracted by line 2 are identified by "100", and data extracted by line 3 are identified by "101". Line 5 configures the sampling buffers. In this example, *p* is configured as 4,000 ms, and *L* is configured as 400. Therefore, the sampling rate (p/L) is 10 ms. As a result of the configuration, two sampling buffers with a length of 400, each of whose elements is 2 bytes (short) in size, are allocated from static working memory. This allocation is processed at CSDA startup, and if all working memory is used up, startup will fail with an error. If such a failure occurs, the *size* attribute will need to be reduced.

#### 3.3.2 Data Processing Section

In this section, the configuration for periodic data processing, which consists of statistical calculations, threshold filtering, and a combination of the two, is specified. An example configuration is shown in Fig. 6. The *sequencer* tag declares the start (line 1) and end (line 24) of the periodic data processing. A corresponding data processing thread is launched at startup. If there are two sequencer tag pairs, two threads are launched at startup. The *seqCacheSetting* tag in line 2 configures the processing cache. Here, 200 bytes are allocated from static working memory for the processing cache. The *runCondition* tag in lines 3-5 specifies the frequency of periodic processing. This configuration means that calculations within the sequencer tag (lines 1-24) are to be processed every 1,000 ms, and input data are to be extracted from the sampling buffer for the last 1,000 ms. After

```
 1:<sequencer sequencerId="1">
# Processing cache size is 200 bytes
 2:    <seqCacheSetting size="200"/>
# Line 6-24 are processed every 1000 ms
 3:<runCondition runType="periodic">
 4:   <term timeUnit="msec">1000</term>
 5: </runCondition>
# Average is taken for data identified as 100
 6:<step stepId="1">
 7:  <operation method="average">
 8:    <arg key="refRawId">100</arg>
 9:  </operation>
10:</step>
# If above average exceeds threshold, goto line 15
# Else go to line 23
11:<step stepId="2">
12:  <filter operator="greater equal" rightOperand="10"/>
13:   <nextstep false="4"/>
14:</step>
# Histogram is taken for data identified as 101
15:<step stepId="3" cache="yes">
16:  <operation method="histogram">
17:    <arg key="series">20</arg>
18:    <arg key="min">0</arg>
19:    <arg key="max">100</arg>
20:    <arg key="refRawId">101</arg>
21:  </operation>
22:</step>
# Some  calculations...
23:<step stepId="4" cache="yes">
    (snip)
24: </sequencer>
```

**Fig. 6** Part of the configuration for data processing step

the runCondition tag, processes are described in *step* tags, and each step tag has a step number specified in the stepId attribute.

A statistical calculation is specified by an *operation* tag within the step tag. Embedded logic components are called as identified by the *method* attribute. Lines 6-10 are an example of a statistical calculation. Here, the method attribute is "average", and the averaging calculation is called from embedded logic components. In order to refer to data from the input step, a *rawRefId* in an *arg* tag is used. Data corresponding to rawDataID = "100" are taken from the sampling buffer for the last 1,000 ms and are averaged. Lines 11-14 are an example of threshold filtering. If the result of the previous step (average) is greater than 10, then go to the next step; if it is not greater than 10, then go to step 4 (line 23) as specified by the *false* attribute. In the next steps (lines 15-22), a histogram of frequency counts is taken for data whose rawDataID = "101". Results of this calculation are also identified by a *stepId* attribute in the *step* tag, to be used by others. For example, the result of the histogram step is identified as "3". Processing cache usage is configured by the *cache* attribute in the step tag. If it is "yes", as in line 15, the result of the calculation is saved in the processing cache.

#### 3.3.3 Output Section

The process of concatenating and sending data to the server is specified. An example configuration for the output step is shown in Fig. 7. The *deliverer* tag declares output section. A corresponding output processing thread is launched at startup. If there are two deliverer tag pairs, two threads are launched at startup. In the example, only one thread is launched. In lines 2-8, every 60 minutes, data that are identified as "3" and "4" are taken from the processing cache and concatenated. Finally, the result is passed to another program to send the data to the M2M server.

```
1:<deliverer delivererId="101">
# Data is outputted every 60 min
2: <runCondition runType="periodic">
3:   <term timeUnit="min">60</term>
4: </runCondition>
# Data created in data processing
# (line 15-22,23-24 in previous figure) are concatenated
5: <dataFormat>
6:   <stepId>3</stepId>
7:   <stepId>4</stepId>
8: </dataFormat>
9:</deliverer>
```

**Fig. 7** Part of the configuration for output step

### 3.4 Update Module

This module downloads the configuration from the network and rewrites it. The configuration is written in data area in Flash ROM which is usually used to store various configuration data. Failure recovery must be considered similar to firmware update. To achieve this, aggregation processors and update modules run as separate processes. The update is performed as follows:

- Step 1: The update module downloads the configuration file onto the RAM, and a copy of the old configuration file is created in the flash ROM as a backup.
- Step 2: The update module writes a new configuration to the flash ROM.
- Step 3: The update module restarts the aggregation processor
- Step 4: The aggregation processor loads new configuration from the flash ROM

Even if writing the configuration fails in Step 2 because of an unexpected power outage, the update can be resumed from Step 1, and the old configuration can be used, regardless of whether the resuming download fails. The flash ROM size required for the backup is at most the size of the configuration.

## 4. Evaluation

CSDA was implemented in C, and the CSDA effectiveness was evaluated in an experimental environment similar to an M2M gateway. CSDA was compared with the development of dedicated logic in a traditional C program. First, to investigate the advantage of CSDA, the developmental process and the updating of aggregation logic on an M2M gateway was evaluated. In addition, the capability of the sampling buffer to adjust memory usage within the limits was also evaluated. Second, overhead was evaluated. CSDA has performance overhead compared with a dedicated C program because CSDA includes logic to process various calculations according to configurations. To verify that the level of

**Table 1** Embedded system used in the evaluation

| | Specification |
|---|---|
| CPU | Freescale i.MX25 (ARM926EJ-S) 400 MHz |
| RAM | LPDDR SDRAM 64 MB |
| Sensor network interface | ISO11898 compliant CAN interface |
| OS | Linux 2.6.26 |

overhead is acceptable, memory and CPU usage were measured and compared with those of a dedicated C program.

### 4.1 Experimental Setup

CPU clocks for M2M gateways are typically around 200-600 MHz, and RAM sizes are around 1-64 MB, as was stated in Sect 2.2.1. We used the Armadillo 420 [27] as an evaluation device because its CPU power and RAM size are similar to the above specifications and its SDK is available to the public. Its specifications are shown in Table 1. The CAN was selected as a sensor network because it is widely used in industrial devices. CSDA was implemented in C and ported to the device in the following evaluations.

### 4.2 Evaluation of Advantage of CSDA

To verify the advantage of CSDA over a dedicated C program, the development and update processes are compared. Next, the capability of the sampling buffer to adjust working memory usage within the set limits is evaluated.

#### 4.2.1 Development of the Aggregation Process

The productivity of C and of CSDA in the development of an aggregation process are compared. The following aggregation process was used for the evaluation:

- Input process step: Take two kinds of sensor value for Sensor A and Sensor B.
- Data processing step: When the 30-second average value for Sensor A is greater than a given threshold, create a histogram of the Sensor B values.
- Output step: Send the histogram data once each day.

The number of steps was calculated for the dedicated C code and the number of tag pairs was counted for the CSDA configuration. The number of steps required in the C program was 575, and the number of tag pairs in the CSDA configuration was 29. The semantics of C and XML are different, but the amount of configuration is much less than in the C code. The sampling buffer was not implemented in the dedicated C program; if it were to be implemented, the amount of the program would increase more.

In addition, there are the following three difficulties in writing C code. The first is that of bit operation. In the input step, sensor data are extracted from the binary frame. In order to handle binary communication, bit operations are used, as shown in Fig. 8; these operations are difficult to read. Eight variables and three arrays are used in a mere

```
* Part of C code
for(idx2 = 0; idx2 < byteLength ;idx2++){
  if(offsetRem > rightShift){
       output[idx2] = (data[idx3]>>rightShift)
       & forAnd[leftShift];
* CSDA configuration
 <extract rawId="10" offset="0" len="8"/>
```

**Fig. 8** Comparison of C and CSDA at input step

three lines. Using the CSDA, on the other hand, data extraction can be described in one simple line. The second difficulty is with memory operations. As is well known, failure to handle C arrays causes many problems. If there is a bug in the boundary handling, the memory may be destroyed, and this weakness can be also exploited in a buffer overflow attack [28]. The CSDA hides such memory operations. Third is thread management. The thread needs to be separated for the input process and data processing in order to enable the receiving of data during the data processing. Thread programming depends on the OS, and programmers need to learn manners. Moreover, the shared resource needs to be handled carefully because failure can lead to resource destruction or a deadlock. Here as well, the CSDA hides the operations.

#### 4.2.2 Updating Aggregation Logic

Updating of the aggregation logic in CSDA is compared with traditional firmware updating through consideration of the following aspects during the update process:

1. Delivery of update data: In a firmware update, the entire image needs to be delivered. In some cases its size can exceed 10 Mbytes. In CSDA, the binary configuration file needs to be delivered, but its size is small. For example, the size of the binary configuration used in the previous section is 1 Kbyte. This will save network bandwidth and loading of the delivery server.
2. Logic updating: Since the size of the binary configuration file is much smaller than a firmware image, the time for rewriting the configuration file will be also much smaller than that needed for rewriting a firmware image.
3. Failure recovery: In most M2M gateways, there is a failure recovery feature for firmware updates. If a firmware update fails, it is recovered from backup firmware. However, intervention by an engineer in the field is necessary since the M2M gateway is not functioning after the failure.
   In CSDA, there is a failure recovery feature as described in Sect 3.4. Recovery after a failure can be performed without the need for an engineer in the field because other parts of the M2M gateway and update module are working. For example, if there is a monitor function in the update module that watches the CSDA status, the monitor function can detect a failure and recover the configuration file from backup.

**Table 2** Result of memory usage measurement of CSDA with varying configuration of sampling buffer. L is length of array for sampling buffer and R is sampling rate.

| | CSDA L=30000 R=1 ms | CSDA L=3000 R=10 ms | CSDA L=300 R=100ms |
|---|---|---|---|
| Memory usage (Kbytes) | 448 | 259 | 240 |

#### 4.2.3 Capability of Sampling Buffer to Reduce Memory Usage

To verify that memory usage can be reduced by adjusting the sampling buffer, the CSDA memory usage was measured for varying array lengths of the sampling buffer. In the evaluation, it was assumed that data are inputted from three sensors whose data are 2 bytes in size and that three sampling buffers are prepared to store data from them for 30 seconds. The CSDA was simply configured to perform 30-second averaging of the sensor data stored in the sampling buffers.

Memory usage of the CSDA was measured as follows. As described in Sect 3.2.3, CSDA has static working memory for the sampling buffer. Logic to measure the size of CSDA's unused working memory ($A$) was inserted in the CSDA, and the total CSDA memory usage ($B$) was measured by free command. (Note that $B$ is the difference in the output of the free command before and after starting CSDA.) Here, the actual CSDA memory usage is $B$ - $A$. $A$ and $B$ were measured when the length of array for the sampling buffer was 300 (for a sampling rate of 100 ms), 3,000 (for a sampling rate of 10 ms), and 30,000 (for a sampling rate of 1 ms).

The measured memory usage for the sampling buffer is shown in Table 2. $B$ - $A$ is displayed there. For example, in the case of $L$ = 30,000, $A$ was 132 Kbytes and B 580 Kbytes; thus, the CSDA memory usage was 580 - 132 = 448 Kbytes. From the table, it is observed that CSDA memory usage decreases as $L$ decreases and that the rate of decrease is just 7 bytes. It is confirmed that the CSDA memory usage can be adjusted in a memory-constrained environment by configuring $L$. For example, assuming only 300 Kbytes is allowed by the M2M gateway's manufacturer, when $L$ = 30,000 ($R$ = 1 ms), the memory usage will be 448 Kbytes, exceeding the limits. In such a case, memory usage can be reduced by reducing $L$. The value of $L$ that achieves the best sampling rate within the memory usage limit can also be easily calculated. Since the rate of memory usage decrease is 7 bytes, by reducing $L$ by 21,143 (= 148 Kbytes/7), memory usage will be just 300 Kbytes. The resulting $L$ is 8,857 (= 30,000 - 21,143), and the sampling rate will be 3.4 ms (= 30 s / 8,857).

### 4.3 Evaluation of Overhead

When aggregation logic is implemented on the CSDA, memory usage is expected to be higher than that with a dedicated C program, because the CSDA is implemented in C and logic to handle various aggregations is included. CPU usage is expected to be higher as well. To verify that such overheads are acceptable, the RAM and CPU usage are compared with those in a dedicated C program.

#### 4.3.1 Overhead in RAM Usage

The overhead in memory usage was measured as follows:

- Aggregation logic that simply performs 30-second averages of data from three sensors and discards the outputs was implemented both in a dedicated C program and in CSDA. Neither the sampling buffer nor the processing cache were implemented in the dedicated C program; i.e., it reads 30 seconds of data into a dynamically allocated buffer.
- To determine the increase in CSDA memory usage for handling various aggregations, $C$ and $D$ were measured as follows:
  $C$: (memory usage of CSDA) - (size of sampling buffer)
  $D$: (memory usage of dedicated C program) - (size of dynamically allocated buffer)
  $C$ - $D$ thus represents the CSDA overhead.

The result of measuring the CSDA overhead ($C$ - $D$) was 72 Kbytes, where $C$ was 248 Kbytes and $D$ was 176 Kbytes. This increase is not significant, because even if the RAM size is 8 Mbytes, it is less than 1%.

#### 4.3.2 Overhead of CPU Usage

In order to see the CPU overhead, complicated aggregation logic was configured and evaluated on the device, the same as that of Table 1. The logic for the evaluation was a statistical calculation to predict faults in a construction machine using data from three sensors, which is composed of 30 calculation elements [14]. To simulate heavy traffic, sensor data were inputted every 1 ms, then the statistical calculation was performed every 100 ms. CPU time was measured for 1 second. CPU time was also measured for a dedicated C program where only the aggregation logic was coded.

The observed CPU usage of the CSDA was 1.5% (CPU time 15 ms), and that for the dedicated C program was less than 0.1% (CPU time less than 1 ms). The CPU usage of the CSDA was not significant even under heavy traffic with complicated aggregation logic, but was worse than for the C program. The performance loss is due to overhead in parsing the calculation rule. When a use case with more complicated aggregation logic is encountered in the future, it will need to be tuned.

### 4.4 Limitation Consideration

The limitation of the CSDA is that it cannot describe statistical calculations whose logic and configuration language are not already embedded. In order to use the CSDA in a variety

of cases, it is assumed that enough sets of statistical calculations are supported. In practice, "enough sets" are prepared after the CSDA is used in multiple situations. However, if "enough sets" are defined, the memory consumption may become too large because of the amount of embedded logic. To solve this problem, a plug-in framework would be useful. If new calculation logic is necessary, it could then be supported by adding plug-ins. Memory usage would thereby be saved since only the necessary plug-ins would need to be installed in the CSDA.

## 5. Related Works

OSGi [29] is used to facilitate program writing in an M2M gateway [30], [31]. Because the OSGi is a Java-based framework, it is easier to use than C when writing arbitrary logics. However, coding is still necessary, and it only works on a rich M2M gateway because Java requires enough CPU space and memory.

A data stream management system (DSMS) [32] partially removes data aggregation coding. It processes the input data stream based on the rule called query. However, because it is not originally intended for sensor data aggregation, the rule language does not cover the input and output steps. It is primarily intended for real-time processes, so it requires enough CPU and memory resource. There is a DSMS for resource-constrained devices [26], but its rules are static and a firmware update is necessary to change them. Yamamoto and Koizumi proposed DSMS for distributed environment on top of MySQL [33]. It facilitates describing data aggregation logic by utilizing SQL, but it uses traditional ring buffer for statistical calculation and not suitable for memory constrained environment.

AirSenseWare [34] processes data using sensor nodes and a rule-based server. Its focus is the framework for distributed data processing and not aggregation on a gateway.

SwissQM [35] processes data using sensor nodes that cooperate with a gateway device. Here, the role of the gateway is different from the one discussed in this paper. Gateways employ various types of query rules, such as XQuery and SQL, and data processing is performed on sensor nodes.

There are sensor data aggregation methods on M2M gateways by data compression techniques. Yamaguchi et al. [36] compressed sensor data on M2M gateway by utilizing multi-dimensional similarity of sensor data. Papageorgiou et al. [37] developed an algorithm to choose the best data compression method for sensor data aggregation. Matamoros et al. [38] aggregates sensor data on M2M gateways to reduce overhead of bridging sensor to the Internet. These works focus on data compression on specific situations. Problems are different from our work because we propose a framework to facilitate developing and updating aggregation logics for resource constrained environment.

There are works related to evaluation of effects of sensor data aggregation on M2M gateways. Lo et al. [39] evaluated relationship between protocol overhead and communication delay when sensor data are aggregated on gateways. Tsai et al. [40] simulated effect of data aggregation to energy consumption of M2M systems.Problems are also different from our work because these works do not mention development of aggregation logics.

## 6. Conclusions

To reduce the server load and communication cost of M2M systems, sensor data are aggregated in an M2M gateway. The aggregation logic is usually programmed in C rather than higher-level programming languages because the CPU and memory resources are constrained. However, there are difficulties with programming in C and with updating the programs. We proposed a framework called complex sensor data aggregator (CSDA) to solve such difficulties. The proposed CSDA is highlighted as follows.

- CSDA enables sensor data aggregation in M2M gateways without the need for programming. This CSDA supports categorized data aggregation methods in three steps: the input, the periodic data processing, and the output steps. In each step, behavior is configured using XML-based rules.
- In order to keep CSDA within the memory limit specified by the M2M gateway's manufacturer, the number of threads and the size of working memory is static after startup, and the size of the working memory can be adjusted by configuration of a sampling setting for a buffer for sensor data input.
- Experimental results on an evaluation board show that developing CSDA configurations is much easier than programming in C. The amount of configuration is less than 10% of the comparable C code. Memory usage can also be reduced by adjusting the sampling setting. In addition, the basic memory usage and CPU usage of the CSDA are not significant for M2M gateways; i.e., the CPU usage with the CSDA is about 1.5% for complicated aggregation logic, and the increase in memory usage compared with dedicated C logic is about 70 Kbytes.

## Acknowledgments

We would like to thank Noriko Takada and Kenji Okura in Hitachi, Ltd. for giving us information about data aggregation requirement from industrial use cases. We would also like to thank to people in Entier group in Hitachi Solutions, Ltd. for help in developing prototype.

### References

[1] Komatsu: Komtrax: http://www.komatsuamerica.com/komtrax
[2] Hitachi Construction Machinery: Global-eService: http://www.hitachi-c-m.com/global/businesses/products/global_e-service.html
[3] Yanmmar: Yanmar's Advanced Technology Gives Customers a SmartAssist, 2013: http://yanmar.com/news/contents/105278.php
[4] Zigbee Alliance, Interconnecting Zigbee & M2M Networks,

http://docbox.etsi.org/workshop/2011/201110_m2mworkshop/03_m2mcooperation/zigbee_taylor.pdf
[5] CiA: Controller Area Network, http://www.can-cia.org/index.php?id=can
[6] ETSI TS 102 690 V1.1.1, Machine-to-Machine communications (M2M) functional architecture, 2011.
[7] MQtt: http://mqtt.org
[8] I. Fette and A. Melnikov, Request for comments 6455: The WebSocket Protocol, IETF, 2011.
[9] Z. Shelby, K. Hartke, and C. Bormann, Request for comments 7252: The Constrained Application Protocol (CoAP), IETF, 2014.
[10] S. Bandyopadhyay and A. Bhattacharyya, "Lightweight Internet protocols for web enablement of sensors using constrained gateway devices," Proc. 2013 International Conference on Computing, Networking and Communications (ICNC2013), pp.334–340, 2013.
[11] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," ACM Transactions on Database Systems, vol.30, no.1, 2005.
[12] A. Boulis, C.-C. Han, R. Shea, and M.B. Srivastava, "SensorWare: Programming sensor networks beyond code update and querying," Pervasive and Mobile Computing, vol.3, no.4, pp.386–412, 2007.
[13] T. Murakami, T. Saigo, Y. Ohkura, Y. Okawa, and T. Taninaga, "Development of Vehicle Health Monitoring System (VHMS/WebCARE) for Large-Sized Construction Machine," Komatsu Tech Rep, vol.48, no.150, pp.15–21, 2003.
[14] J. Fujiwara and H. Suzuki, "Device for collection construction machine operation data," WIPO Patent, WO2013077309 A1, 2013.
[15] Y. Shinohara and H. Sakamoto, "Remote monitoring terminal device for traveling work machine or ship," WIPO patent, WO2013080712 A1, 2013
[16] BroadBand Forum, TR-069 Issue 1 Amendment 2, http://www.broadband-forum.org/technical/download/TR-069_Amendment-2.pdf
[17] Y. Nakamura, A. Moriguchi, and T. Yamauchi, "CSDA: Rule-based Complex Sensor Data Aggregation System for M2M Gateway," Proc. Eighth International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2015), pp.110–115, 2015.
[18] Hitachi Solutions, Ltd., Entier Stream Data Aggregator: http://www.hitachi-solutions.co.jp/entiersda/ (In Japanese)
[19] E. Guenterberg, H. Ghasemzadeh, R. Jafari, and R. Bajcsy, "A Segmentation Technique Based on Standard Deviation in Body Sensor Networks," Proc. 2007 IEEE Dallas Engineering in Medicine and Biology Workshop, pp.63–66, 2007.
[20] Y. Zhuang, L. Chen, X.S. Wang, and J. Lian, "A Weighted Moving Average-based Approach for Cleaning Sensor Data," 27th International Conference on Distributed Computing Systems, p.38, 2007.
[21] T. Canli, A. Gupta, and A. Khokhar, "Power Efficient Algorithms for Computing Fast Fourier Transform over Wireless Sensor Networks," Proc. IEEE Computer Systems and Applications, pp.549–556, 2006.
[22] T. Takishita, K. Murakami, K. Seki, and K. Morishita, "Application of ICT to Lifecycle Support for Construction Machinery," Hitachi Review, vol.62, no.2, pp.107–112, 2013.
[23] Quake Global: Q4000, http://quakeglobal.com/files/tinymce/uploaded/documents/Q4000%202013.pdf
[24] Hitachi Industrial Equipment Systems: HX series, http://www.hitachi.com/New/cnews/month/2015/11/151116.html
[25] Hitachi Super LSI Systems: Communication module for industrial devices, http://www.hitachi-ul.co.jp/system/cmodule/index.html (in Japanese)
[26] S. Katsunuma, S. Honda, K. Sato, and H. Tanaka, "The Static Scheduling Method in Data Stream Management for Automotive Embedded Systems," IPSJ Journal Database, vol.5. no.3, pp.36–50, 2012.
[27] Atmark Techno, Inc., Armadillo-420, http://armadillo.atmark-techno.com/armadillo-420
[28] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhange, and H. Hinton, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks," Proc. 7th USENIX Security Symposium, pp.63–78, 1998.
[29] OSGi Alliance, OSGi Service Platform, Release 3, IOS Press, Inc., 2003.
[30] Y. Li, F. Wang, F. He, and Z. Li, "OSGi-based service gateway architecture for intelligent automobiles," Proc. Intelligent Vehicles Symposium 2005, pp.861–865, 2005.
[31] Kura: https://eclipse.org/kura/
[32] The STREAM Group, STREAM: The Stanford Stream Data Manager IEEE Data Engineering Bulletin, 2003.
[33] M. Yamamoto and H. Koizumi, "An Experimental Evaluation of Distributed Data Stream Processing using Lightweight RDBMS SQLite," IEEJ Transactions on Electronics, Information and Systems, vol.133, no.11, pp.2125–2132, 2013.
[34] K. Muro, T. Urano, T. Odaka, and K. Suzuki, "AirsenseWare: Sensor-Network Middleware for Information Sharing," Proc. 3rd International Conference on Intelligent Sensors, Sensor Networks and Information 2007 (ISSNIP2007), pp.497–502, 2007.
[35] R. Muller, G. Alonso, and D. Kossmann, "Swiss QM: Next Generation Data Processing in Sensor Networks," Proc. 3rd Biennial Conference on Innovative Data Systems Research, pp.1–9, 2007.
[36] K. Yoi, H. Yamaguchi, A. Hiromori, A. Uchiyama, T. Higashino, N. Yanagiya, T. Nakatani, A. Tachibana, and T. Hasegawa, "Multi-dimensional sensor data aggregator for adaptive network management in M2M communications," Proc. 2015 IFIP/IEEE International Symposium on Integrated Network Management, pp.1047–1052, 2015.
[37] A. Papageorgiou, B. Cheng, and E. Kovacs, "Real-time data reduction at the network edge of Internet-of-Things systems," Proc. 11th International Conference on Network and Service Management, pp.284–291, 2015.
[38] J. Matamoros and C. Anton-Haro, "Data aggregation schemes for Machine-to-Machine gateways: Interplay with MAC protocols," Proc. Future Network & Mobile Summit, pp.1–8, 2012.
[39] A. Lo, Y. Law, and M. Jacobsson, "A cellular-centric service architecture for machine-to-machine (M2M) communications," IEEE Wireless Commun., vol.20, no.5, pp.143–151, 2013.
[40] S.-Y. Tsai, S.-I. Sou, and M.-H. Tsai, "Reducing Energy Consumption by Data Aggregation in M2M Networks," An International Journal of Wireless Personal Communications, vol.74, no.4, pp.1231–1244, 2014.

**Yuichi Nakamura** received his BS and MS degrees in physics from University of Tokyo in 1999 and 2001, and MS degree in computer science from The George Washington University in 2006. He worked for Hitachi Solutions from 2001 to 2015. He is working for Hitachi since 2015 and is also studying at Okayama University to obtain a PhD degree.

**Akira Moriguchi** received his BS and MS degree in Chemistry from University of Kyushu in 2006 and 2008. He has worked for Hitachi Solutions, Ltd. since 2008 and engaged in R&D for M2M technologies.

**Masanori Irie** has worked for Hitachi Solutions, Ltd. since 2003. He belongs to Entier group at Hitachi Solutions and is responsible for development of DBMS for embedded system.

**Taizo Kinoshita** is Vice President of IoT Business Division of IT Company, Hitachi, Ltd. He received his PhD in Electrical Engineering from Nagoya University. When he joined in Central Research Lab. in 1981, he engaged in high speed digital optical and wireless network research. After his status of Director in Central Research Lab, he established corporate venture company of Wirelessinfo in 2006. He contributed especially for social infrastructure IoT/M2M real application and business, of railway, highway, smart city. He is now responsible for IoT/M2M business in Hitachi, and is also Director of Next Generation M2M Consortium.

**Toshihiro Yamauchi** received BE, ME and PhD degrees in computer science from Kyushu University, Japan in 1998, 2000 and 2002, respectively. In 2001 he was a Research Fellow of the Japan Society for the Promotion of Science. In 2002 he became a Research Associate in Faculty of Information Science and Electrical Engineering at Kyushu University. He has been serving as associate professor of Graduate School of Natural Science and Technology at Okayama University since 2005. His research interests include operating systems and computer security. He is a member of IPSJ, IEICE, ACM, USENIX and IEEE.