Research Article


Naoya Namatame,1 Jin Nakazawa,2 and Hideyuki Tokuda1

1 Graduate School of Media and Governance, Keio University, Kanagawa 2520882, Japan
2 Faculty of Environment and Information Studies, Keio University, Kanagawa 2520882, Japan

Correspondence should be addressed to Naoya Namatame, namachan@ht.sfc.keio.ac.jp

Received 1 October 2012; Accepted 22 October 2012

Academic Editors: J. Li and Y.-C. Wang

Copyright © 2012 Naoya Namatame et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper focuses on a sensor network virtualization over multidomain sensor network and proposes an abstraction called “logical sensor network (LSN)” for sensor data processing. In the proposed abstraction, processing is a directed acyclic graph that consists of nodes and streams, which represents a small data processor and communication rules between them, respectively. We have added a notion of a trigger to this graph. A trigger represents a timing of the process execution. We have implemented the middleware named LSN-Middle to run a virtualized sensor network and proved its feasibility.

1. Introduction

In a future where sensor networks are ubiquitously around us, applications should interact with multiple sensor networks that belong to different domains. Many efforts to make a platform that provides applications a transparent access to sensor data from existing sensor networks with different domains have been made [1–8]. We call this type of platforms “multidomain sensor network (MDSN).” Sensor networks in MDSN include various types of hardware, are organized by different institutions, and are distributed over the world. Utilizing MDSN from applications has two undesirable features that are (1) sensor heterogeneity and (2) raw sensor data delivery. This means applications can only receive raw sensor data with heterogeneous data units and they are responsible for the data translation or the data preprocessing phases. This is caused by a feature that programmers cannot neither configure nor program the sensor networks which consist MDSN. Thus, we consider that an application development over MDSN will be more complicated than the development over a single homogeneous sensor network on which an application programmer has a full configurability and programmability.

For these undesirable features of MDSN, a virtualization of a sensor network can be an effective solution. Sensor network virtualization over MDSN can provide a dedicated sensor network to each application as shown in Figure 1. This figure illustrates application and physical, multidomain and virtualized sensor network. MDSN aggregates physical sensor networks that have been installed for specific purposes [9–13]. And virtualized sensor networks recompose MDSN and create sensor networks for arbitrary applications with arbitrary specifications. Many researchers have been investigating middleware which offers applications to create a virtualized sensor network [14–17]. The middleware allows an application to define a virtualized sensor network. The application can be developed and operated as if it has a dedicated sensor network with a preferred specification. Virtualized sensor network offers application programmers to define sensors and sensor data processing. These features cope with the two undesirable features of MDSN we have mentioned, respectively.

To enable application programmers to define a virtualized sensor network, well-defined abstractions for both sensor and sensor data processing are needed. Although the existing middleware researches and standardizations [18] have well discussed the abstraction of sensors, they have not well investigated the abstraction of sensor data processing. In this research, we propose a sophisticated abstraction for sensor data processing which enables programmers to express...
stateful processing, asynchronous eventing, and multistage data passing. We have also implemented a middleware, which runs virtual sensor networks.

This paper is organized as follows. First, we clarify the features of MDSN and how the sensor network virtualization can contribute to the application development over MDSN in Section 2. Also the problem of the existing researches of the virtualized sensor network will be organized in the section. Section 3, then, proposes LSN and explains details about the abstraction of sensor data processing. Sections 4 and 5 provide implementation and evaluation of LSN-Middle. Section 6 examines related researches. Finally, we summarize this paper in Section 7.

2. Applications on Multidomain Sensor Network

MDSN is a platform that provides applications a transparent accessibility to sensor data from existing sensor networks. To describe usages and features of MDSN, we give an example scenario of an application called “Just-In-Time Flurry Forecast.”

Just-In-Time Flurry Forecast. On a sunny day, Alice decides to go shopping to a supermarket nearby her house on foot. On the way to the supermarket, her smartphone suddenly alarms her. She picks up her phone and sees a message that says it will be raining so hard in 5 minutes where she is walking now. Of course she do not have an umbrella, she starts to walk a little faster towards her destination. The flurry starts so hard when she just arrives at the supermarket.

2.1. Applications. The notion of “Just-In-Time Flurry Forecast” is derived from an idea that an application provides a microscale weather forecast of where you are right now. The features of MDSN, which offers applications to access sensor data that published by others, are necessary to make this application possible. However, there are two undesirable features to implement this kind of application over MDSN.

(i) Sensor Heterogeneity. Since MDSN consists of physical sensor networks that belong to a large number of institutions and individuals, there is a huge variety in sensors. Therefore, it includes many data units to describe a single physical information. For example, in Just-In-Time Flurry Forecast scenario, the application needs to detect a rain amount (mm/h) within 5 km from the application’s location. However, MDSN may contain various data units (mm/10 mins, mm/hour, mm/day), the application have to translate sensor data with those units into one unit which is mm/10 mins in this case. In addition, the data sources may change when the location of the application moves. Therefore, the application needs to check every sensor data and translate it before the data processing phase.

(ii) Raw Sensor Data Delivery. Before providing data for high-level data processing, which we call an application’s main logic, processing steps of raw sensor data into characteristic values are necessary. For example, Just-In-Time Flurry Forecast needs (1) noise filtering each sensor data, (2) averaging multiple sensor data located in certain areas, and (3) calculating a direction of the rain travel, these are the required processing steps. Then, the application’s main logic finally calculates how long it takes before flurry arrives at the location of the application and decides whether a notification should be sent or not. Although these steps should be done in a sensor network [19, 20], it is impossible to program sensor networks that consist MDSN since they do not belong to the application programmer. Thus, the application needs to receive raw sensor data and is responsible for process it before its main logic.

2.2. Sensor Network Virtualization for MDSN. Sensor network virtualization can be a solution for these features. A
virtualized sensor network that is defined by an application
process. These states cannot be stored in a single function
that is introduced in the related researches. For example, a
noise filter such as lowpass filter that is mentioned in the
scenario requires historical sensor data to calculate the result.

(ii) Asynchronous Eventing. Some processes may wait to
execute their process until all the available data to be
delivered and other processes may want to execute its process
every delivery of sensor data. Process execution timing is
not considered in the related researches. For example, a noise
filter should be executed on every data delivery, on the other
hand, averaging should wait for its execution until a certain
amount of sensor data to be delivered.

(iii) Multistage Data Passing. Some processes may need
the results of many different processes which are executed
asynchronous timing. Multistage data passing cannot be
expressed in a synchronous function as introduced in the
related researches. For example, the application scenario
requires three steps to calculate final output.
The sensor data processing abstraction should be able
to express these three aspects. This research will focus on these
aspects of sensor data processing and propose an abstrac-
tion of sensor data processing that can be used in MDSN
middleware.

3. Logical Sensor Network: Abstraction for
Sensor Data Processing

We propose a simple abstraction of sensor data processing
named “Logical Sensor Network (LSN)” that has ability to
express stateful data processing, asynchronous eventing, and
multistage data passing. This model is a directed graph in
which a node (oval) represents a small data processor and a
stream (arrow) represents communication rules between
nodes. Also, a node can select its process execution timing
with a trigger. Top-level nodes and A bottom-level node of
group, which are exceptions, represent data sources and a
data sink, respectively. Data sources are expressed abstractly,
so that physical sensors that match to the condition will be
placed. With this simple abstraction, sensor data processing
in a virtualized sensor network for MDSN becomes simple
and expressive. Figure 3 shows a LSN that expresses sensor
data processing in the scenario of Just-In-Time Flurry

![Figure 2: Steps required to utilize MDSN from application and LSN’s responsibility.](image-url)
3.1. Node. The abstraction of node represents a data process. Node can have \(n\) input streams from other nodes and \(m\) output streams to other nodes as their input stream. Figure 4 shows possible combinations of the number of input and output streams. Each node contains states that are available as long as the node exists. This feature enables stateful data processing. Source, that is a node with no input stream, is mapped to physical sensor and provides sensor data. Sink, that is a node with 0 output stream, represents an interface to an application. Input streams of a sink can only be accessed by an application. Therefore, source and sink do not contain processing aspect. All input and output stream combinations which are shown in the Figure 4 can express various kinds of processing that are necessary for sensor data processing, especially preprocessing level. Figures 4(b), 4(c), 4(d), and 4(e) can be categorized as a processor.

They can be categorized into three types as shown in Table 1, which are process, source, and sink.

The number and entity of source nodes need to be changed dynamically since LSN should support the idea of virtual sensors. For this reason, source nodes should be defined in an abstract way. Source node should be defined with a data kind (e.g., temperature), a data unit (e.g., celsius), a data type (e.g., numeric: double), a sampling rate (3000 ms), a total number, and a condition (e.g., target area).

In Figure 3, Rain A1 and Rain A2, that are expressed as a shaded oval, are sources. Sink, that is shown as heavy lined node represents sink, and the other nodes, which are \(LPFilter\), \(Average\), and \(Direction\), represent process. Rain A1 is defined as Table 2. Rain A2 are defined just same as Rain A1 with different context.

3.2. Trigger. Trigger is an element that executes a node’s process. Each process node has one trigger. Trigger can be categorized into three types based on an aspect of a timing of a process execution. The categories are shown in Table 3. Flow is a type that executes a process when one of connected input streams brings sensor data. On the other hand, Rendezvous waits the execution until a specified percentage of the connected input streams brings their sensor data. And Timer executes the process periodically with a specified interval. The concept of the trigger enables programmers to easily handle an asynchronous eventing aspect of sensor data streams.

In Figure 3, the node with italic font has a flow type trigger. \(LPFilter\) has flow type trigger in this case. Nodes with underlined font, that are Average, have a rendezvous trigger. The percentage written under the node name specifies the requirement. \(Direction\), that is expressed with bold font, is a node with a timer trigger. The time written under the node name represents the execution interval.

3.1. Node. The abstraction of node represents a data process. Node can have \(n\) input streams from other nodes and \(m\) output streams to other nodes as their input stream. Figure 4 shows possible combinations of the number of input and output streams. Each node contains states that are available as long as the node exists. This feature enables stateful data processing. Source, that is a node with no input stream, is mapped to physical sensor and provides sensor data. Sink, that is a node with 0 output stream, represents an interface to an application. Input streams of a sink can only be accessed by an application. Therefore, source and sink do not contain processing aspect. All input and output stream combinations which are shown in the Figure 4 can express various kinds of processing that are necessary for sensor data processing, especially preprocessing level. Figures 4(b), 4(c), 4(d), and 4(e) can be categorized as a processor.

They can be categorized into three types as shown in Table 1, which are process, source, and sink.

The number and entity of source nodes need to be changed dynamically since LSN should support the idea of virtual sensors. For this reason, source nodes should be defined in an abstract way. Source node should be defined with a data kind (e.g., temperature), a data unit (e.g., celsius), a data type (e.g., numeric: double), a sampling rate (3000 ms), a total number, and a condition (e.g., target area).

In Figure 3, Rain A1 and Rain A2, that are expressed as a shaded oval, are sources. Sink, that is shown as heavy lined node represents sink, and the other nodes, which are \(LPFilter\), \(Average\), and \(Direction\), represent process. Rain A1 is defined as Table 2. Rain A2 are defined just same as Rain A1 with different context.

3.2. Trigger. Trigger is an element that executes a node’s process. Each process node has one trigger. Trigger can be categorized into three types based on an aspect of a timing of a process execution. The categories are shown in Table 3. Flow is a type that executes a process when one of connected input streams brings sensor data. On the other hand, Rendezvous waits the execution until a specified percentage of the connected input streams brings their sensor data. And Timer executes the process periodically with a specified interval. The concept of the trigger enables programmers to easily handle an asynchronous eventing aspect of sensor data streams.

In Figure 3, the node with italic font has a flow type trigger. \(LPFilter\) has flow type trigger in this case. Nodes with underlined font, that are Average, have a rendezvous trigger. The percentage written under the node name specifies the requirement. \(Direction\), that is expressed with bold font, is a node with a timer trigger. The time written under the node name represents the execution interval.

---

### Table 1: Types of nodes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>Process (n) inputs and sends (m) outputs</td>
</tr>
<tr>
<td>Source</td>
<td>Mapped to physical sensors and provides sensor data</td>
</tr>
<tr>
<td>Sink</td>
<td>An interface to an application</td>
</tr>
</tbody>
</table>

### Table 2: Definition of source nodes (Rain A1).

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data kind</td>
<td>Temperature</td>
</tr>
<tr>
<td>Data unit</td>
<td>Celsius</td>
</tr>
<tr>
<td>Data type</td>
<td>Number: double</td>
</tr>
<tr>
<td>Sampling</td>
<td>3000 ms</td>
</tr>
<tr>
<td>Target number</td>
<td>All available</td>
</tr>
<tr>
<td>Context</td>
<td>Within 5 km from current loc.</td>
</tr>
</tbody>
</table>

### Table 3: Types of triggers.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow</td>
<td>executes a process when one of connected input streams brings sensor data</td>
</tr>
<tr>
<td>Rendezvous</td>
<td>waits the execution until a specified percentage of the connected input streams brings their sensor data.</td>
</tr>
<tr>
<td>Timer</td>
<td>executes the process periodically with a specified interval</td>
</tr>
</tbody>
</table>
Table 3: Types of trigger.

<table>
<thead>
<tr>
<th>Name</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow</td>
<td>Italic</td>
<td>Each time one of the data from connected input arrives</td>
</tr>
<tr>
<td>Rendezvous</td>
<td>Underlined</td>
<td>When the data from all connected input arrive</td>
</tr>
<tr>
<td>Timer</td>
<td>Bold</td>
<td>At a certain time cycle</td>
</tr>
</tbody>
</table>

Table 4: Process execution timing and available data.

<table>
<thead>
<tr>
<th>Type</th>
<th>Timing</th>
<th>Available data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow</td>
<td>$d_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td></td>
<td>$d_2$</td>
<td>$d_2$</td>
</tr>
<tr>
<td></td>
<td>$d_3$</td>
<td>$d_3$</td>
</tr>
<tr>
<td></td>
<td>$d_4$</td>
<td>$d_4$</td>
</tr>
<tr>
<td>Rendezvous</td>
<td>$d_4$</td>
<td>$d_1, d_2, d_3, d_4$</td>
</tr>
<tr>
<td>Timer</td>
<td>$t_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td></td>
<td>$t_2$</td>
<td>$d_2, d_3$</td>
</tr>
</tbody>
</table>

Table 5: Types of stream.

<table>
<thead>
<tr>
<th>Name</th>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>A simple arrow</td>
<td>Sends data when a process is executed</td>
</tr>
<tr>
<td>Pull</td>
<td>A two direction arrow</td>
<td>Sends data when requested from the other node</td>
</tr>
</tbody>
</table>

Figure 5 shows the model of a data arrival to a node from three connected streams. A painted circles ($d_1$–$d_4$) on each stream represent sensor data coming into the node. Data with smaller number arrives at the node earlier. The dotted vertical lines ($t_1$, $t_2$) represent time that a timer trigger executes the process. When data arrives at the node from these streams, Table 4 shows the timing of the process execution and available sensor data at that time for each trigger type. In this case, when the node adopts a rendezvous trigger, the node receives two data from the same stream, namely, $d_1$ and $d_3$. In such case, whether utilizing the latest data, that is $d_3$, or utilizing average of both data depends on the implementation of the node. LSN abstraction does not concern this issue.

3.3. Stream. Stream connects two nodes and expresses a data communication as shown in Table 5. An output value from a node goes through a stream and provided to the other node as input value. A stream has two types that represent different communication aspects. One is called push that pushes data to the other node after a process execution. A simple arrow represents push link. The other is called pull which retrieves sensor data from the destination node when a node executes its process. An arrow with backward direction tail represents pull link. Connecting nodes by streams enables multistage data passing. And each node only receives data that are relevant to its process.

In Figure 3, data will be pushed by former nodes, as for LPFilter and Average. On the other hand, Direction has pull stream towards Average, which means Direction retrieves the latest average data on its process execution.

4. Framework Implementation

We have implemented a framework and middleware for defining and running LSN that is defined according to our abstraction of sensors and sensor data processing. Figure 6 shows how an application instantiate LSN. (1) First, an application sends a definition of LSN to a server with LSN-Middle. (2) When LSN-Middle receives the definition, it instantiate LSN according to the definition. Then it receives and processes sensor data from MDSN.(3) Finally, LSN-Middle sends processed sensor data to the application. Definition of LSN includes source sensor definition (such as Table 2) and a program of LSN that is described in this section. All or a part of a LSN definition can be published to the public and other applications can reuse it.

In this section, we will describe our java framework implementation for programming sensor data processing with the abstraction that we have discussed in the last section using LSN in Figure 3. Each node type shown in Figure 4 has a corresponding java abstract class and an application
instances of Average class in onUpdate(). setStream(String, instantiated according to new sources and reconnected to the instantiated, but instances of LowpassFilter class are newly
when binding of logical/physical sensors are changed. In the application developer should write initiation code of LSN should implement onInit() and reconstruction code in onUpdate() for the class that extends Logical Sensor Network class. The subclass code shown in Listing 3. LSNc an be defined with iajava
an a mean dat r i g g e ro b j e c t .

Once an application developer implemented processor nodes, he/she needs to connect them and create LSN. LSN that is shown in Figure 3 can be expressed as the source code shown in Listing 3. LSN can be defined within a java class that extends Logical Sensor Network class. The subclass should implement onInit() and onUpdate() methods. The application developer should write initiation code of LSN in onInit() and reconstruction code in onUpdate() for when binding of logical/physical sensors are changed. In the case of Listing 3, instances of Average class are not newly instantiated, but instances of LowpassFilter class are newly instantiated according to new sources and reconnected to the instances of Average class in onUpdate(). setStream(String, Node) or addStream(String, Node) is used to connect nodes. These methods need stream name and destination node as parameters. Stream name is useful for distinguishing the stream to read/write data when multiple input/output streams are there.

LSN source code is compiled and sent to LSN-Middle in jar file by an application and executed remotely. When the application sends a connect request with its current context to the LSN, LSN collaborates with MDSN and starts to send sensor data. Listing 4 shows application code that creates, connects, and updates a LSN instance on the remote LSN-Middle.

5. Evaluation

We have done an evaluation test to ensure that the LSN-Middle's design and implementation is feasible. In the evaluation, we have compared the time that required for a smartphone to acquire data from MDSN and process it with and without LSN-Middle. The LSN that we have used for the evaluation is shown in Figure 7. This comparison can discover whether LSN-Middle’s overhead cost for sending or dynamically loading jar file is acceptable or not. The test requires a smartphone to acquire an average value of \{10, 50, 100, 500, 1000\} lowpass filtered sensor data from MDSN with 2 patterns: with and without LSN-Middle. The test is conducted within a private network with a simulated delay and bandwidth. Bandwidth is set to 3000 kbps and RTT between the smartphone (Sensation, 1.2 GHz Dual-Core Qualcomm MSM8260, Android 2.3.3) and LSN-Middle (MacBook Air, 1.8 GHz Intel Core i7, MAC OS X 10.7) or MDSN platform is 25.8 (stddev: 1.44). MDSN platform that we have used is prepared for this evaluation and it generates simulated sensor data. A client that needs data from the MDSN sends a query with sensor conditions, and first the client gets an address list of matched sensors and then sensor data follows. All the specified amount of sensor data is generated within 3000 ms and sent to the client. The last sensor data is generated exactly on 3000 ms from the request as an anchor. We have measured a time that the smartphone took to get the average value from the time sending a request. This time includes the required time for MDSN to finish generating all the sensor data. Therefore, we have defined response time that is related to the efficiency to be MeasuredTime – 3000 ms.

The graph in Figure 8 shows the result. The error bar represents standard deviation. As a result, the performance with LSN-Middle is 12.57% better in average. This means, in spite of the additional required steps, the performance is better with LSN-Middle. There are two reasons that the response time becomes larger as the number of sensor data becomes larger. One is that sensor data processing time becomes larger in proportion to the number of sensor data. The other reason is that the cost of sending a list for sensor nodes from MDSN is becoming larger. The performance with LSN becomes better because these tasks are executed in the LSN-Middle where there are richer computational resources than that of a smartphone. When processing a heavier load task than averaging, the performance difference
public class RainForecastNetwork extends LogicalSensorNetwork {

    private Source[][] rain;
    private LowpassFilter[][] lpf;
    private Average[] ave;
    private Direction dir;

    public RainForecastNetwork() throws Exception {
        super("conf/rain_forecast.lsn");
    }

    @Override
    public boolean onInit() {
        rain = new Source[2][]; // Logical Sensors
        lpf = new LowpassFilter[2][]; // Lowpass Filters
        ave = new Average[2]; // Average Aggregators
        dir = new Direction("Direction", new TimerTrigger(10000));

        // Instantiating Processor Nodes
        for (int i = 0; i < 2; i++) {
            rain[i] = manager.getNodes("RainA" + (i+1));
            lpf[i] = new LowpassFilter[rain[i].length];
            for (int j = 0; j<lpf[i].length; j++)
                lpf[i][j] = new LowpassFilter("LPF"+i+j, new FlowTrigger());
            ave[i] = new Average("Ave" + i, new RendezvousTrigger());
        }

        // Connecting Processor Nodes
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j<rain[i].length; j++)
                rain[i][j].setPushStream("value", lpf[i][j]);
            lpf[i][j].setPushStream("value", ave[i]);
        }  
        ave[i].setPullStream("value", dir);

        dir.setPushStream("value", sink);
        return true;
    }

    @Override
    public boolean onUpdate() {
        // Disconnect Processor Nodes
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j<lpf[i].length; j++)
                lpf[i][j].removeStream(ave[i]);
        }

        // Get new binding and instantiate new LowpassFilters
        for (int i = 0; i < 2; i++) {
            rain[i] = manager.getNodes("RainA" + (i+1));
            lpf[i] = new LowpassFilter[rain[i].length];
            for (int j = 0; j<lpf[i].length; j++)
                lpf[i][j] = new LowpassFilter("LPF"+i+j, new FlowTrigger());
        }

        // Connect new LowpassFilters to Averages
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j<rain[i].length; j++)
                rain[i][j].setPushStream("value", lpf[i][j]);
            lpf[i][j].setPushStream("value", ave[i]);
        }

        return true;
    }
}

Listing 3: Source code for construction and update for logical sensor network in Java.
public class MicroForecast implements LSNListener, GPSListener{
  private LSNClient client;
  public MicroForecast(double lat, double lng) throws Exception{
    client = new LSNClient("lsnoperator.org", 12345, 22345);
    client.setListener(this);
    //Create LSN instance on a remote server.
    client.create("org.RainForecastNetwork","lsn.jar");
    //Connect LSN instance on the remote server.
    client.connect(new Location(lat, lng));
  }
  @Override
  public void dataArrived(LSNDataPacket packet){
    //Process data using packet. (Main Logic)
  }
  @Override
  public void gpsUpdated(double lat, double lng){
    //Let the remote server know about context update.
    client.update(new Location(lat, lng));
  }
}

Listing 4: Source code for construction and update for logical sensor network in Java.

Figure 7: Logical Sensor Network used in the evaluation.

Figure 8: Performance evaluation of sensor data processing with and without LSN-Middle.

6. Related Work

Researches that focus on processing of sensor data can be seen in the field of macroprogramming. Titan [21], proposes a simple abstraction for sensor data processing for heterogeneous physical sensor network. Titan represents data processing by a data flow from sensors to recognition results. The abstraction of processing is represented as task, data, and connection. Task represents an elementary computational functions such as classifier and filters, and connection represents data flow from task. The entire process is expressed as directional graph in which tasks are connected by connections. The process execution is distributed among multiple sensor nodes. For the distributed computation, Titan has a special task for a communication to send sensor data to a different sensor node. However, from the aspect of the process abstraction, a communication among physical sensors nodes should be transparent and not be shown in the process structure. If our middleware supports a distributed process execution, we will not change LSN structure, but add these settings as properties of a Node.

Abstract Task Graph (ATaG) [22] provides a sophisticated abstraction of sensor data processing. ATaG expresses sensor data processing with notion of task, data, and channel, which is called connection in Titan. Titan and ATaG have very close abstraction model, but in ATaG, task and channel have annotation, which can specify detailed behaviors.
Annotations for task can specify the policy of task execution timing. Annotations for channel can specify the policy of data flow and selection of target tasks. ATaG uses a notion of task network with a more detailed abstraction in eventing and a communication aspect. The major difference is that ATaG has an element for data in its taskgraph. ATaG has to express data element as a part of the taskgraph structure since ATaG’s assumption of task or process is too abstract that we can not recognize the output data from the name of the task or former tasks that are connected to it. On the other hand, the assumption of a process in LSN, which is called a task for ATaG, is a mathematical function that is small and general enough for people to know the output value from the name of the process. An implementation of process can be reused for other applications, by keeping it small and general.

7. Conclusion

In this research, we have claimed that the abstraction of sensor data processing proposed in the existing sensor network virtualization middleware over MDSN is not sufficient. Thus we have proposed a new abstraction called LSN to meet the requirements. An actual middleware and framework for sensor network virtualization was implemented and evaluated as well. Through the evaluation, we have found that despite of overhead that LSN-Middle requires, sensor data processing performance was 12.75% faster with LSN-Middle support in average compared to the result calculated in an application running on a smartphone. There are two future works for this research. One is to implement a graphical authoring tool for LSN to support application programmer to write and test LSN. And the other is to update the middleware for distributed execution of LSN since its structure is suitable for parallel processing. This can improve its performance and scalability.

References


Submit your manuscripts at http://www.hindawi.com