Vehicle Embedded Data Stream Processing Platform for Android Devices

Shingo Akiyama*, Yukikazu Nakamoto*, Akihiro Yamaguchi†, Kenya Sato‡, Hiroaki Takada§
*Graduate School of Applied Informatics, University of Hyogo, Japan
†Center for Embedded Computing System, Nagoya University, Japan
‡Mobility Research Center, Doshisha University, Japan
§Graduate School of Information Science, Nagoya University, Japan

***Abstract*—Automotive information services utilizing vehicle data are rapidly expanding. However, there is currently no data centric software architecture that takes into account the scale and complexity of data involving numerous sensors. To address this issue, the authors have developed an in-vehicle data-stream management system for automotive embedded systems (eDSMS) as data centric software architecture. Providing the data stream functionalities to drivers and passengers are highly beneficial. This paper describes a vehicle embedded data stream processing platform for Android devices. The platform enables flexible query processing with a dataflow query language and extensible operator functions in the query language on the platform. The platform employs architecture independent of data stream schema in in-vehicle eDSMS to facilitate smoother Android application program development. This paper presents specifications and design of the query language and APIs of the platform, evaluate it, and discuss the results.**

Keywords*—Android, automotive, data stream management system

I. Introduction

Automotive information services utilizing vehicle data are rapidly expanding. Several standardizations have been put into place for the rapid deployment of such services. Vehicle data interfaces such as OpenXC and Mirrorlink have recently been standardized and are expected to become more popular, despite the fact that existing built-in car navigation systems use proprietary and closed vehicle data. OpenXC defines APIs that provide diagnostic data from a controller area network (CAN) bus in a vehicle network [1]. The Car Connectivity Consortium has recently standardized Mirrorlink, which defines interfaces that connect smartphones with vehicle information [2]. Google and Apple announced software platforms for automotive information services, Android Auto[3] and CarPlay[4], respectively. Android Auto provides functionalities with which Android devices communicate to a vehicle. CarPlay is an iOS virtual machine on top of OS in an in-vehicle system and enables communication with iOS devices.

Automotive control is now undergoing the same technology trends described above. Intelligent control systems have become popular due to their sophisticated, safe, and environmentally friendly control exploiting numerous types of data from a vehicle itself, its surroundings, and other vehicles. Typical systems that employ such technologies include pre-crash safety systems, adaptive cruise control, lane departure warning systems, and intelligent parking assist systems. These systems collect environmental data from sensors in a vehicle and make decisions on the basis of this data to control the vehicle on behalf of the driver. Jones used information obtained from multiple on-board sensors to perform evasive steering and, when collision is unavoidable, to activate brake intervention to dampen the impact, thus decreasing damage [5]. Such intelligent control systems acquire data from many sensors, such as cameras and millimeter-wave radar. Google and Urban Challenge, which is a competition funded by the Defense Advanced Research Projects Agency, revealed that self-driving in urban areas is both feasible and safe in terms of autonomous driving [6], [7].

There are also more advanced information and control techniques for automotive systems and services that use data through communication. The automotive industry has been studying cooperative intelligent transport systems (C-ITSs) to improve transport safety, productivity, and reliability by using data collected through vehicle-to-infrastructure [1] (V2I) and vehicle-to-vehicle (V2V) communications, as well as GPS from outside the vehicle and data collected on-board [8][9][10]. Thus, C-ITS technologies enhance automotive information and control systems.

AUTOSAR [2], which is an automotive software standardized organization, recently proposed a component-based software platform and software development methodologies to address the growing scale and complexity of automotive control software [11]. AUTOSAR does not, however, take into account the scale and complexity of data involving numerous sensors. Moreover, application integration is becoming more complex because access to one sensor requires communication with an application that manages the sensor when each application manages sensors by itself. When information from multiple sensors is integrated or when new sensors or algorithms need to be added, the software architecture needs to be redesigned and reorganized. Those technologies mentioned above do not provide solutions for those problems.

In response to these issues, the authors have researched and developed a data centric software architecture for automotive systems. The evaluation results of both a database management system (DBMS) and a data stream management system (DSMS) showed advantage of the latter system because the DSMS can already efficiently handle continuous incoming data such as vehicle sensor data [12]. The above observation

[1]Infrastructure is a specific term in ITS that refers to the roads, centers, and facilities around vehicles.

[2]http://www.autosar.org/

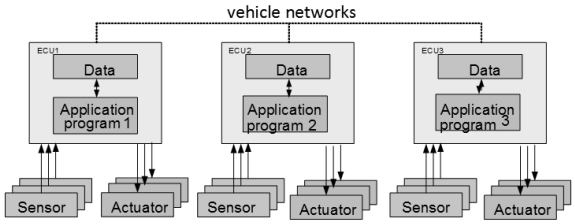

Fig. 1: Current architecture in an automotive system.

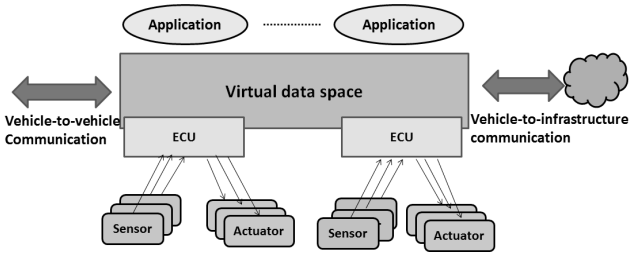

Fig. 2: Data centric architecture in an automotive system

becomes a basis for developing a data-stream management system for an automotive embedded system (eDSMS)[13], [14]. In addition, a data-stream-based local dynamic map (LDM) was previously proposed [15]. The data-stream-based LDM consists of layered data, including geography, circumjacent vehicle status road conditions, and congestion and weather circumstances and is a key technology in C-ITS.

Providing the data stream functionalities to drivers and passengers are highly beneficial. This paper describes a vehicle embedded data stream processing platform for Android devices. The rest of this paper is organized as follows. A brief overview of the data centric architecture and data stream management system for embedded systems (eDSMS) is presented in Sec. II. Sections III and IV describe the architecture and design, respectively, of the vehicle embedded data stream processing platform for Android devices whose target is vehicle information systems. SectionV shows a demonstration program utilizing the platform. Evaluation results are presented in Sec.VI. Related works are briefly discussed in Sec. VII and conclusions are described in Sec. VIII.

II. Data Stream Management System for Vehicle Embedded System: eDSMS

This section describes the data centric software architecture based on data-stream processing for automotive systems. The development of these systems has been discussed in previous works [12], [13], [14].

A. Background

The architecture of current automotive systems is shown in Fig. 1. Programs in an electrical control unit (ECU) [3] obtain data from numerous sensors, process the data, and output commands to actuators to provide control for automotive systems, including control and information systems. Sensor data are duplicated and processed in application programs in multiple ECUs because each application program has to process sensor data in its own ECU. The cost of developing and integrating application programs increases dramatically if the number of sensor data types increases. This occurs because many sensors are fixed to a vehicle or numerous data come from outside the ego-vehicle [4].

Figure 2 shows a proposed data centric software architecture for an automotive system. This architecture provides virtual data space for data not only in a vehicle but also from other vehicles and infrastructures, thus hiding the data's origin. This architecture separates sensors from application programs and provides common access methods for sensor data, which is managed collectively in the logical data space. Moreover, the proposed architecture increases opportunities for sensor fusion, which enhances one piece of sensor data with other sensor data.

To determine the feasibility of the data-centric software architecture in vehicle software, they evaluated a DBMS and a DSMS. The feasibility study used two application programs: adaptive cruise control and intelligent parking assist. The results of the evaluation demonstrated that DBMS is superior at processing queries featuring large amounts of data at low frequency while DSMS is superior at processing queries featuring small amounts of data at high frequency. This results in adopting DSMS in the data centric architecture of in-vehicle systems because most data in automotive systems are continuous sensor-generated data and have short lifetimes. At the same time, the architecture use DBMS for static data such as map information and convert the static data into stream data to pass the data on to DSMS.

B. In-vehicle eDSMS

An embedded DSMS (eDSMS) is suitable for embedded systems, especially in-vehicle embedded systems. Note that software in embedded systems must be particularly customized because CPU and memory capacities are limited and so real-time processing and high reliability of the systems are required.

This section presents an overview of DSMS. The DSMS input is stream data. DSMS obtains data in the stream specified by a query and then outputs that data as a stream. The DSMS query is issued for the data stream and is executed continuously, unlike a query in DBMS. There are two types of query language in DSMS: SQL-like declarative languages and procedural languages, specifically, dataflow languages [16][pp.723-743]. In the SQL-like query languages, a user specifies selection predicates from streamed input data. Stream data are then converted into relations in sliding windows in DSMS. A query is executed over the sliding window similarly to a conventional SQL query. The result of the query is then converted into a stream again. In the dataflow query languages, a user specifies a query with a dataflow graph, where a node is a query operator (discussed later) and an edge is a stream. Data from the input stream flows in the dataflow query. An operator processes data in the dataflow query and detects the specified data. In this way, a user can describe a query procedure explicitly in the dataflow query languages.

[3] An electric control unit (ECU) is a computer used for vehicle control.

[4] An ego-vehicle means the vehicle that is being focused upon.

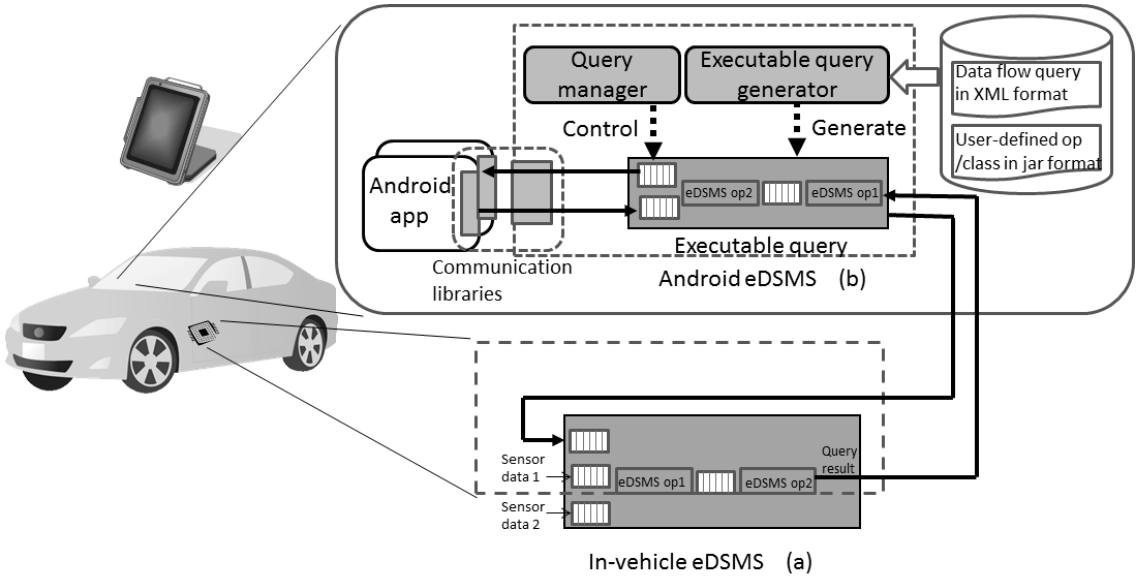

Fig. 3: Android eDSMS and in-vehicle eDSMS.

The in-vehicle eDSMS has three features of embedded systems: dataflow query language, static query processing, and optimization. First, a query language in the in-vehicle eDSMS is a dataflow language that is similar to the one in Borealis[17]. This is because a query in the dataflow language is more flexible in terms of customizing and tuning the processing for automotive applications. Second, a dataflow language constructs a hierarchical structure of data easily from physical layers to more abstract layers. Finally, a user can reuse data in the arbitrary level of the dataflow query, which avoids redundant usage of data.

In the automotive field, in-vehicle application programs do not change dynamically. This means there is no updating, adding, or deleting the application programs because the programs are fixed to guarantee reliability and safety after the long-term verification and validation of the programs. Thus, the query processing in in-vehicle eDSMS does not change dynamically either. The query, whose actual representation is XML-form, is converted into C/C++ source programs that are compiled into run-time routines on a PC, embedded into the ECU in a vehicle, and executed as part of the vehicle's programs. The execution time of a query is predictable in in-vehicle eDSMS. This predictability property is very important in real-time systems such as vehicle systems because they need to be guaranteed to finish their processes.

There are several optimization techniques in in-vehicle eDSMS query processing for reducing the processing overhead and ROM/RAM usage, including deleting operator dynamic linking, linking selected operator modules only, and decreasing the number of tasks used in in-vehicle eDSMS runtime.

III. eDSMS in Android platform

A. Requirements

Providing the data stream functionalities to drivers and passengers are highly beneficial. The role of an embedded data dream management system for an Android platform in a device (Android eDSMS or AeDSMS) is to provide not only straightforward usage of the in-vehicle data stream to drivers and passengers but also various usage data streams between in-vehicle and Android devices. The requirements of Android eDSMS are considered in the following cases:

- Case 1: Presenting services to drivers and passengers utilizing the data stream in the in-vehicle eDSMS. A simple application is to inform a driver of warnings and cautions. Another application is to retrieve information suitable for the driving situation (e.g., location, time, weather) from the stream data from the in-vehicle eDSMS and present it to the driver.
- Case 2: Sending input information from Android applications as data stream or parameters to an AeDSMS operator in the dataflow query to in-vehicle eDSMS. For example, a driver may control a vehicle or inform other vehicles of a traffic condition in the form of sensor data to the in-vehicle eDSMS.
- Case 3: Executing part of the eDSMS dataflow query from the in-vehicle in the Android platform. The purpose of this execution is to debug a query of the in-vehicle EDSMS in a more convenient programming environment or to offload query processing in an Android device to better utilize resources within the device itself. In offload usage, although the processing time of the query becomes shorter, predictability is lost because the operating system of the offloaded Android device has a general purpose OS.

B. Prototype

A prototype of eDSMS for Android devices has the following features for the feasibility evaluation [18]. The target

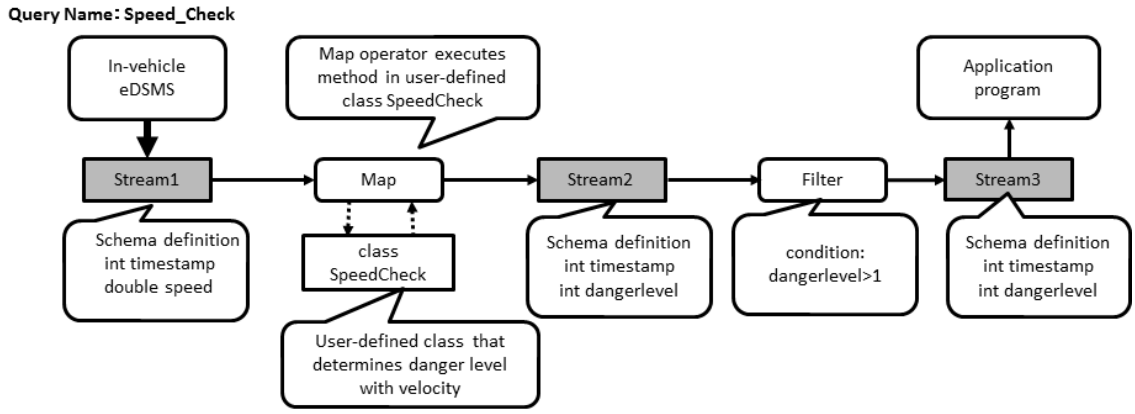

Fig. 4: Query example.

of the prototype is Case 1 from the previous subsection. In the Android eDSMS prototype, data is filtered from the in-vehicle eDSMS data stream and passed on to the Android eDSMS. The filtering is specified by an SQL-like query. The development process of an Android application using the prototype is as follows.

1) A developer creates a schema file in an XML format that defines the field information of the stream data acquired from the in-vehicle eDSMS.
2) The AeDSMS builder in a PC obtains a schema file as an input and generates the Java source codes of a query in the prototype, including the class of data received from the in-vehicle eDSMS.
3) A developer integrates application source codes with the query source codes and compiles them to produce an application program in the PC for an Android device.

If the schema of a data stream from the in-vehicle eDSMS changes, it is necessary to generate the query source codes and compile source codes again. This is neither convenient nor flexible.

C. Architecture of Android eDSMS

On the basis of the experience with the prototype development, an enhanced embedded data stream management system for an Android device has the following three features. The purpose is to address the issues mentioned above in Cases 1 and 2 and to add features to improve the developing efficiency from the perspective of the developer.

- Flexible query processing with a dataflow query language and extensible function of operators in the query language
- Facilitation of Android application program development with architecture independent of data stream schema in in-vehicle eDSMS and class libraries to hide implementation details
- Capability of various usages in multiple Android applications by providing query processing as an Android service

Android DSMS is part of a two-layer structure for an embedded data stream platform for automotive systems, shown in Fig. 3. Figure 3 (a), the bottom, is an in-vehicle eDSMS in ECUs, and Fig. 3 (b), the top, is an Android eDSMS. In this structure, the in-vehicle eDSMS obtains sensor data, processes the data in the form of general purpose usage, and sends it to the Android platform. The Android eDSMS receives the data stream from the in-vehicle eDSMS and provides various usages of the data stream between in-vehicle and Android devices, such as driving situation services including location, time, and weather.

D. A dataflow query of Android eDSMS

AeDSMS adopted a dataflow query language in the data stream query the same as the one in in-vehicle eDSMS, which is similar to the one in Borealis [17]. There are two types of dataflow query files in the XML format: a schema file and a query file. The schema file contains schema information of the stream in a query and the query file describes operators and connections between operators used in the query. Built-in query operators in AeDSMS are listed in Table I, which are also the same as in-vehicle eDSMS. An operator has input streams, output streams, and parameters required for execution. AeDSMS reads both files, translates them into an executable query, and executes the query in an Android device (see Fig. 3). A developer can extend the functions of an operator, which means these functionalities provide developers with easier, more flexible application development.

An example of a query is shown in Fig. 4. This is a simple query to determine the danger level according to the vehicle speed and to select only those levels higher than 1. Below is an excerpt of the query in the XML format.

TABLE I: Operators in Android eDSMS.

Operator	Description
Filter	read data from an input stream, perform filtering in accordance with the condition data, and write the result to an output stream
Map	read data from an input stream, perform the specified method on it, and write it to an output stream
Unite	read data from multiple input streams, merge and write it to an output stream
Join	read data from an input stream, hold it for certain period of time, combine the data held by the specific conditions, and write it to an output stream
Aggregate	read data from an input stream, hold it for certain period of time, perform aggregate functions on the data, and write it to an output stream

```
<query name="speed_check">--(1)
<box name="test_map" type="Map">--(2)
  <in stream="Stream1" />
  <out stream="Stream2" />
  <parameter name="speed_check"
    class="SpeedCheck"/>--(3)
</box>
<box name="speed_filter" type="Filter">
  <in stream="Stream2" />
  <out stream="Stream3">
  <parameter name="expression_0"
    value="dangerlevel>1" />--(4)
</box>
</query>
```

A `query` element represents a query. A query name is defined in a `name` attribute (1). In a `box` element, an operator is defined and a `type` attribute specifies an operator type (2). This example uses `Map` and `Filter` operators. The `box` element consists of an operator name, input and output streams for the operator, and operator parameters. A class `SpeedCheck` is specified as a user defined class called by the Map operator (3). The Filter operator specifies a condition where data is passed to Stream 3 only if the value of the variable `dangerlevel` representing the danger level exceeds 1 (4).

AeDSMS provides two operator extensions: an extensible operator with a user defined class and a user defined operator. A developer can write a class where a method is called from an operator to extend and change the process and execution conditions of that operator. Moreover, a developer can define and add a new operator for implementing a flexible query.

Three methods exist in AeDSMS to communicate with in-vehicle eDSMS: TCP, UDP, and Bluetooth. A developer can select the method by specifying it in a query file.

IV. Design of Android eDSMS

The Android eDSMS has the following parts (see Fig.3):

Executable query generator:
: The executable query generator reads the XML format queries, generates executable queries, and classes objects that have field information of the data in the data stream to receive the data in an Android device, as shown in Fig. 3(b).

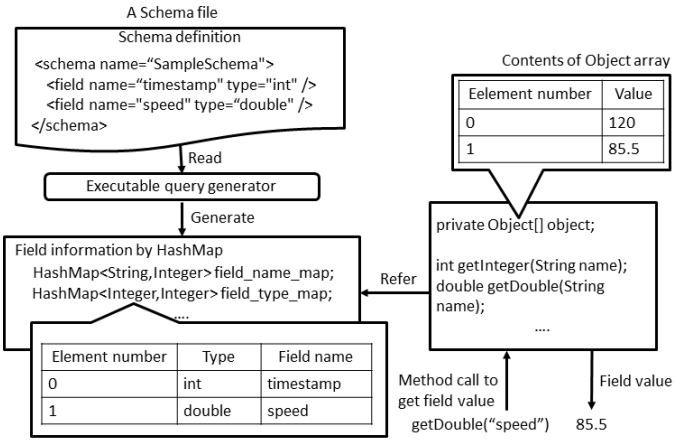

Fig. 5: Stream data access using HashMap generated from query file.

Executable query:
: The executable query is the run time part of the query.

Communication libraries:
: Communication libraries hide the detail of the inter-process communication and structure of AeDSMS.

Query manager:
: The query manager manages the executable query.

From the above architecture, especially the executable query generator, a developer can change the schema in an input data, the process, and the communication method simply by rewriting the query file without generating any AeDSMS code in a PC. A query in AeDSMS can dynamically change in order to provide new services while the program of in-vehicle eDSMS is embedded in a static structure. Also, it is possible for multiple Android applications to use AeDSMS through AeDSMS's Android service.

A. Executable Query Generator

The executable query generator reads the schema in a schema file and the query in a query file and then generates the hash map at run time, as shown in Fig. 5, instead of producing class files corresponding to the data from the in-vehicle eDSMS during the development, as described in Sec. III-B. At this time, a class `AeDSMSFieddInfo` is generated that contains the field name, type and element numbers. Those field are related in the hash map with `HashMap`. At this time, the field name, type, and element numbers are related in the hash map with `HashMap`. Data received from in-vehicle eDSMS is initially stored in an `Object` type array. A class `AeDSMSTupleData` corresponding to the tuple data received from the in-vehicle eDSMS has `getDouble()` and `getInteger()` methods that cast the data `Object` type to a type specified by the method name and retrieve the specified data with the hash map. Moreover, A class `AeDSMSTupleData` provides methods `putDouble()` and `putInteger()` to put data into a tuple data. Therefore, no modification of the Android program is required, even if the data schema obtained from the in-vehicle eDSMS has changed.

B. Extension of an operator

AeDSMS provides two operator extensions: an extensible operator with additional class and a user defined operator. For the extension of an operator, a developer can create a class that implements the interface `AeDSMSUserDefineClassBase`, which has the following methods.

```
interface AeDSMSUserDefineClassBase {
  abstract public AeDSMSTupleData[]
    execution(AeDSMSTupleData[] t,
      Object[] args);
  abstract public String getName();
}
```

An abstract method `execution()` is implemented in the operator with parameters `AeDSMSTupleData[] t` and `Object[] args`. The first parameter `t` is data for executing the extension and the second one is additional.

Another method for an operator extension is a user defined operator. Here, a developer writes a user defined operator in a q query file. In the operator definition, a `type` attribute is the name of the class where the user defined class is written.

```
<box name="test_map" type="UserOp">
  <in stream="Streamx" />
  <out stream="Streamy" />
  <parameter "UserOp parmeters"/>
</box>
```

A user defined operator can be defined with inheritance of an abstract class `AeDSMSOperatorBase`. The class `AeDSMSOperatorBase` has the following methods. In defining the user defined operator, a developer must write how to parse the parameters of the operator defined in the dataflow query file as above. To do that, the developer writes `parse()` in the class definition of a user defined operator. A method `parse()` is called when loading the dataflow query file containing a user defined operator with a `loadXMLQuery()` execution. A method `execution()` is called from the executable query when the operator is executed. A method `isExecutable()` returns information on whether the operator is executable.

```
abstract class AeDSMSOperatorBase {
  public abstract void
    parse(Nodelist node_list);
  public abstract void execution();
  public abstract boolean isExecutable();
}
```

A developer writes a class for extension of an operator and a user defined operator, compiles it, and stores it in Jar file format in an Android device. The executable query generator extracts class information from a user defined class in the Jar format and translates it into an executable Dalvik dex (Dalvik EXecutable) format, which is available on Android. A `dx` tool in the Android SDK performs the translation. We assume that classes for frequently used general processing can be prepared in advance.

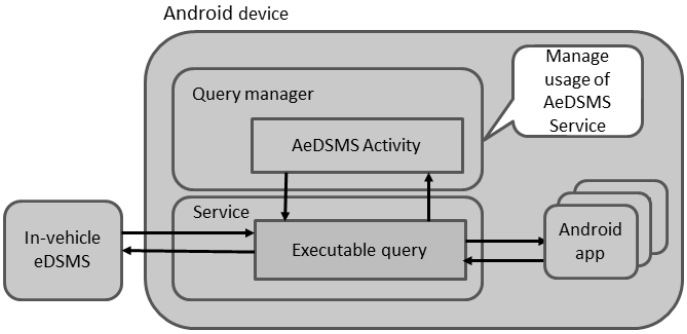

Fig. 6: Multiple process access to Android eDSMS by service.

The operators listed in Table I can also implement the `execution` method. A scheduler in the executable query calls the `execution()` method of each class instance of the operator when the method is executable.

C. Query management

An Android application can read multiple queries and make selections from within those queries. AeDSMS prepared a class `QueryManager` to manage the multiple queries. `QueryManager` also manages the multiple threads needed for a query execution. An excerpt of the member methods is shown below. `QueryManger` holds a query in `ArrayList`. A method `addQuery ()` adds a query in the List. A method `startQuery()` starts execution of the specified query.

```
class QueryManager{
  public void addQuery(AeDSMSQuery query);
  public void deleteQuery(String name);
  public void startQuery(String name);
  public void cancelQuery(String name);
  public AeDSMSQuery getQuery(String name);
}
```

D. Usage from multiple processes by service.

If activity in an Android application creates an AeDSMS instance and runs it, it means that the AeDSMS instance is equal to the number of applications that exist. Moreover, utilizing a single AeDSMS from multiple applications or remote usage of other Android devices is desired. Therefore, AeDSMS is separated from an application to solve this problem and is executed as a separate process with the Android service, shown in Fig. 6. A developer can either select an Android service in an application or perform AeDSMS in an application, where AeDSMS is executed as part of an activity.

E. Process communication and communication library

When running the Android service and an application in separate processes, using inter-process communication for exchanging data is necessary. Android eDSMS implements process communication with Messenger/Handler, which is one of inter-process communication mechanism in Android.

AeDSMS provide a communication library to hide the details of the inter-process communication and programming

in an Android application, rendering it independent of the inter-process communication. One class of the library is shown below as an example.

```
class AeDSMSComm{
  public void loadXMLSchema(String name);
  public void loadXMLQuery(String name);
  public void startAeDSMSQuery(String name);
  public void cancelAeDSMSQuery(String name);
  public AeDSMStream
    getAeDSMStream(String name);
}
class AeDSMSStream {
  public AeDSMSTupleData
    getAeDSMSTupleData();
  public void
    putTupleData(AeDSMSTupleData t);
}
```

A class `AeDSMSComm` provides the following methods.

- A method `loadXMLSchema` loads a `name` schema file and a method `loadXMLQuery` loads a `name` query file in the XML format into AeDSMS and generates the executable query.
- A method `startQuery` starts execution of the `name` query in the loaded query file and `cancelQuery` stops the execution.
- A method `getAeDSMStream` returns the result of the query in the named output stream in the query file.

A class `AeDSMSStream` contains a result of the query that is specified in the named output stream in the query file. The Android application program calls a method `getInteger()` or `getDouble()` with the field name a developer wants to get from the tuple and obtains the value of the tuple as

```
AeDSMSComm ac = new AeDSMSComm();
ac.loadXMLSchema("speed_check_schema.xml");
ac.loadXMLQuery("speed_check.xml");
ac.startAeDSMSQuery("speed_check");
              :
AeDSMSStream s = getAeDSMSStream("Stream3");
AeDSMSTupleData t = s.getAeDSMSTupleData();
int ts = t.getIneger("timestamp");
int dl = t.getInteger("dangerlevel");
```

When a developer send a tuple data to an input of an AeDSMS executable query, he or she generates an input stream for AeDSMS with `getAeDSMSStream` and a tuple data of `AeDSMSTupleData`. A method `putDouble()` puts a value to the specified field and `addTupleData()` adds the tuple data to the stream. Suppose that AeDSMS sends speed control values to the in-vehicle eDSMS. A speed value 50.0 is set to "speed" field in a tuple and the tuple is output to the `Stream4`, which is connected to the in-vehicle eDSMS.

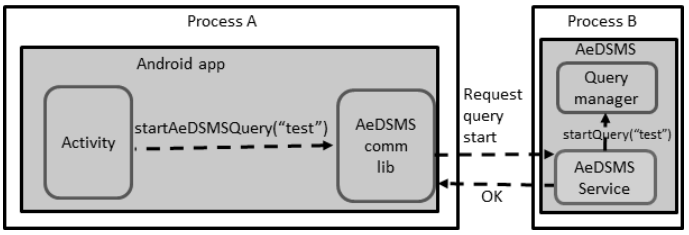

Fig. 7: Hiding implementation details by communication library.

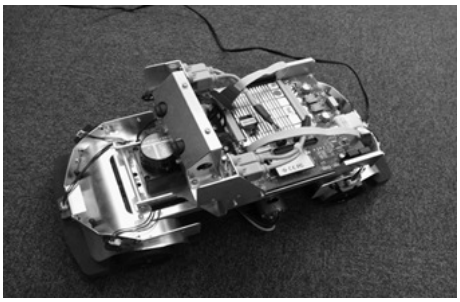

Fig. 8: Robocar 1/10.

```
AeDSMSStream s = getAeDSMSStream("Stream4");
AeDSMSTupleData t =
  new AeDSMSTupleData(finfo);
t.putDouble("speed", 50.0);
s.addTupleData(t);
```

A developer can write an application using `StartAeDSMSQuery()` and `getAeDSMSStream()`, as shown in Fig. 7, without being aware of the process communication.

V. Developing Android application using AeDSMS

This section presents a demonstration application using AeDSMS in NEXUS 7 with ASUS and Google. The vehicle in this demonstration is a miniature of ZMP's RoboCar 1/10[5], shown in Fig. 8. The Android application has a sensor information display function, a battery power display function, and a speed-meter display function.

Figures 9 and 10 show screen shots of the application. Figure 9 provides basic information of the driving. Steering angle and infrared sensor information are displayed using animation on the left side of the screen. A user can tap the speed on the screen of Fig. 9 and transit to the screen shown in Fig. 10, where the current speed measured by a meter is displayed. A developer can thus write various kinds of Android applications more easily and productively.

VI. Evaluation

This section describes performance evaluation of the Android eDSMS on NEXUS7 with the program in Fig. 4. The

[5]http://www.zmp.co.jp/?lang=en

TABLE II: Specification of NEXUS 7

CPU	NVIDIA Tegra 3 (1.3 GHz)
Memory	1GB
OS	Android 4.4.2

Fig. 9: User interface of Android application using Android eDSMS.

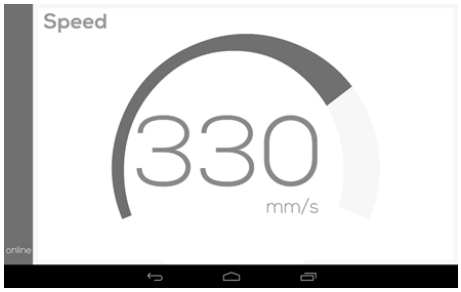

Fig. 10: Another user interface of Android application using Android eDSMS.

TABLE III: Measurement results

processing time with stream operators using AeDMSM	150 μs
processing time with empty operators using AeDMSM	90 μs

measurement time was the end-to-end duration with the operators from time at `Stream 1` to time at `Stream 3` in Fig. 4 as well as the duration of empty operators, which indicates the overhead of the query execution time in AeDSMS. This measurement was performed 100 times and the averages was calculated. The results are shown in TABLE III. These values show that the overhead of the query execution time is relatively small compared with the execution time of the operator, and thus demonstrating the feasibility of implementing AeDSMS.

The overhead of a query in a single process as an "Activity" and query execution using the inter-process communication as a "Service" are different because of the separate processes used. The execution contains an 'empty' query with one empty operator and two streams to measure this difference and found that it takes 4.70 ms to execute a method `loadXMLQuery();` in the single process and 8.76 ms using the AeDSMS service. This measurement was performed 100 times and the averages was calculated. Those executions involve garbage collection time. This difference is the overhead of the inter-process communication as a "Service." A developer should choose to run a query as a single process if there will be no multiple application usages of AeDSMS.

If an Android program consumes memory, garbage collection (GC) occurs and pauses program execution. The execution contains the same 'empty' query and observed its GC when a tuple with one `int` field was input at intervals of 10 ms over 500 s. GC occurred six times during the execution and the average pause time was 18.5 ms. GC is unavoidable in Java and Android, however, one way to prevent GC is that a program generates and reserves tuples that are used before program execution, and gets a tuple from the tuple reservation instead of the instance creation at runtime. In this case, there is a limit of the number of received tuples within a certain time.

VII. RELATED WORKS

For general-purpose DSMS, prototype and commercial systems of Aurora [19] and its successor Borealis [17] and STREAM [20] have been developed. Aurora and Borealis adopt a dataflow language as a query language while STREAM has an SQL-like query language. In the finance field, DSMSs are used in applications related to algorithmic trading and financial monitoring. In the case of algorithmic trading, it is necessary to reduce the response time of query processing, as this directly affects profit. In addition, finance-based DSMSs must update queries immediately when the algorithm is updated. These systems enhance features, such as providing several types of windows between stream operators for real world applications. However, all stream operators, queues (variable length), and TCP communication are embedded as the standard executable code, which leads to larger code size. Part of receiving a query result in an application can be executed in the same thread in the commercial version of STREAM to reduce receiving time latency. Conventional DSMSs are also often applied across the Internet. Such DSMSs process packets as a stream, requiring high throughput rather than adherence to any deadline, unlike in the automotive field. In addition, Internet-based DSMSs often use overlay networks, which are different from in-vehicle networks.

DSMS has started being utilized in embedded systems. The first utilizations have been in the automotive field. Schweppe et al. proposed on-board stream processing for engineering testing and diagnosis in vehicle systems [21]. One of their main features was the adaptation of the behavior of data stream processing in diagnosis when critical events occur, e.g., when the reading rate of sensor data increases. However, their streaming platforms cannot schedule data processing so as to meet deadlines. StreamCars, which is most similar to in-vehicle eDSMS in terms of purpose, proposed a software development platform for vehicle embedded systems. Although StreamCars provides sensor fusion operators, the performance and implementation have not been described in detail.

The Cooperative Cars (CoCar) project at Aachen University is developing a data stream mining platform for automotive systems [22]. Examples of its application include queue-end detection and traffic state estimation. These are processed on the server-side rather than in an automotive embedded system. Although such applications must perform spatial operations to determine which road a vehicle is driving on, the deadline constraint is looser than in driving assistance systems. Unlike

in-vehicle eDSMS, the CoCar platform processes spatial operations using an RDBMS. Data quality (DQ) is important in DSMS, but it is not extensive. Their group also generalized the DQ of a data stream using ontology [23].

Researching real-time scheduling of DSMS is popular because real-time processing from inputs and to outputs is a key property in embedded systems such as automotive systems. [24] presented a real-time scheduling algorithm to guarantee the required quality-of-service level in embedded DSMS. They define the quality-of-service level and resources needed for computation by DSMS operators and provide a framework where a user negotiates the quality in DSMS. Son et al. proposed a periodic query model for real-time applications and an admission control mechanism for an overload situation with irregular stream data arrival [25]. As another scheduling approach, a preemptive rate-based operator scheduling has been proposed [26]. The rate-based scheduling enables earlier execution operators on an operator path in the data stream to perform processing with higher priority. An operator with higher priority can be immediately executed by preempting the current executing operator if the operator is ready. In [27], a task processes data on an operator path in a dataflow query and an operator scheduling algorithm is examined in which a task is earlier executed corresponding to data in the stream with the earliest deadline among the waiting data.

In the second, data processing in a sensor network can be regarded as a data stream [28]. DSMSs are applied to applications such as traffic monitoring and environmental monitoring. As in the embedded field, a small footprint is required because low-specification nodes are often used. Additionally, many applications require distributed processing, and minimal network usage is necessary to preserve battery power and save precious network bandwidth. However, these networks are basically peer-to-peer, which differ from in-vehicle networks. Several previous works based on sensor network ideas, resource saving DSPSs that can be installed in embedded systems, have been developed for the purpose of aggregating and monitoring sensor data [29], [30]. Müller's DSMS[30] is for a wireless sensor node. A query is registered statically and converted into intermediate codes executed on a virtual machine. In the virtual machine, 37 instructions are borrowed from a Java virtual machine and 27 instructions are specified for the data stream processing. They adopted a declarative query language, making it possible to increase the abstraction level and enable in-network programming in a sensor network, reduce the program size in a sensor node, and easily reprogram sensor nodes.

Gigascope [31] is a DSMS for the network equipment in the base station. A query is statically registered and converted into C and C++ source codes, the same as in-vehicle in-vehicle eDSMS. Details have not been published for Gigascope, and there is no description of the optimization of in-vehicle eDSMS.

VEDAS [32] and Minefleet [33] are DSPSs that mainly target mobile computing devices. Their applications relate to data stream mining, i.e., vehicle-health monitoring and driver characterization. They distribute stream processing among mobile computing devices so as to reduce battery usage and wireless communication. However, they are not intended for in-vehicle networks.

VIII. Conclusion

This paper presented a vehicle embedded data stream processing platform for Android devices to provide the data stream functionalities to drivers and passengers with many benefits. The platform enables flexible query processing with a dataflow query language and extensible operator functions in the query language in the platform. The platform has an architecture independent of data stream schema in in-vehicle eDSMS to facilitate Android application program development. Future work includes adopting SQL-like query language for programming that is familiar to Android application programmers and asynchronous query execution for the data stream.

Acknowledgments

The authors thank Masanori Okamoto and Mohanmed Bhuiya for the prototype development of the Android eDSMS and Shinichi Ito, Naoyuki Shiba, Hideteru Shimada, Toshihiko Sugawara, and Naoya Suzuki for the in-vehicle eDSMS development.

Finally, the authors thank Yoshitaka Nakagawa, University of Hyogo, for creating GUIs for the demonstration application program using Android eDSMS.

The present study was supported in part by MIC SCOPE 121806015, JSPS KAKENHI Grant Numbers 25240007 and 24500045. This work was partly done in Education Network for Pratical Information Technologies (enPiT), Japan.

References

[1] OpenXC, "The OpenXC Platform." [Online]. Available: http://openxcplatform.com/

[2] Car Connectivity Consortium, "MirrorLink 1.0 specification," Tech. Rep., 2013.

[3] Google, "Android Auto," 2014. [Online]. Available: http://www.android.com/auto/

[4] Apple, "Apple CarPlay," 2014. [Online]. Available: http://www.apple.com/ios/carplay/

[5] W. D. Jones, "Keeping cars from crashing," *IEEE Spectr.*, vol. 38, no. 9, pp. 840–851, 2011.

[6] M. Buehler, K. Iagnemma, and S. Singh, Eds., *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*. Springer, 2010.

[7] E. Guizzo, "How Google's Self-Driving Car Works," 2011. [Online]. Available: http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/how-google-self-driving-car-works

[8] G. Toulminet, J. Boussuge, and C. Laurgeau, "Comparative synthesis of the 3 main European projects dealing with Cooperative Systems (CVIS, SAFESPOT and COOPERS) and description of COOPERS Demonstration Site 4," in *International Conference on Intelligent Transportation*, 2008.

[9] ETSI, "Intelligent Transport Systems (ITS): Communications Architecture,," 2010.

[10] D. Zhang, H. Huang, M. Chen, and X. Liao, "Empirical study on taxi GPS traces for Vehicular Ad Hoc Networks," in *Proc. 2012 IEEE International Conference on Communications*, 2012, pp. 581–585.

[11] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, P. Citroën, K. Nishikawa, and K. Lange, "AUTOSAR - A Worldwide Standard is on the Road," in *Proc. 14th International VDI Congress Electronic Systems for Vehicles*, 2009.

[12] M. Yamada, K. Sato, and H. Takada, "Implementation and evaluation of data management methods for vehicle control systems," in *Proc. IEEE 74th Vehicular Technology Conference*, 2011, pp. 1–5.

[13] S. Katsunuma, S. Honda, Y. Watanabe, Y. Nakamoto, and H. Takada, "Real-time-aware Embedded DSMS Applicable to Advanced Driver Assistance Systems," in *Proc. 33rd IEEE Symposium on Reliable Distributed Systems Workshops*, 2014, pp. 5–10.

[14] A. Yamaguchi, Y. Nakamoto, K. Sato, Y. Ishikawa, Y. Watanabe, S. Honda, and H. Takada, "AEDSMS: Automotive Embedded Data Stream Management System," in *31st IEEE International Conference on Data Engineering*, 2015 (accepted).

[15] K. Sato, H. Shimada, S. Katsunuma, A. Yamaguchi, M. Yamada, S. Honda, and H. Takada, "Stream LDM : local dynamic map (LDM) with stream processing technology," Doshisha University, Tech. Rep., 2012.

[16] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 3rd ed. Springer, 2011.

[17] D. J. Abadi, Y. Ahmad, M. Balazinska, J.-h. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The Design of the Borealis Stream Processing Engine," in *Proc. Second Biennial Conference on Innovative Data Systems Research*, 2005, pp. 277–289.

[18] Y. Nakamoto, M. Okamoto, M. Bhuiya, A. Yamaguchi, K. Sato, S. Honda, and H. Takada, "Android Platform based on Vehicle Embedded Data Stream Processing," in *2013 IEEE 10th International Conference on Ubiquitous Intelligence & Computing and 2013 IEEE 10th International Conference on Autonomic & Trusted Computing*, 2013, pp. 48–55.

[19] D. J. Abadi, Y. Ahmad, M. Balazinska, J.-h. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[20] D. P. Arvind, A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "Stream: The Stanford stream data manager," *IEEE Data Engineering Bulletin*, vol. 26, pp. 19–26, 2003.

[21] H. Schweppe, A. Z. Member, and D. Grill, "Flexible On-Board Stream Processing for Automotive Sensor Data," *IEEE Trans. Ind. Informat.*, vol. 6, no. 1, pp. 81–92, 2010.

[22] S. Geisler, C. Quix, S. Schiffe, and M. Jarke, "An evaluation framework for traffic information systems based on data stream," *Transportation Research Part C*, vol. 23, pp. 29–55, 2012.

[23] S. Geisler, S. Weber, and C. Quix, "Ontology-based Data Quality Framework for Data Stream Applications," in *Proc. 16th International Conference on Information Quality*, 2011.

[24] S. Schmidt, T. Legler, D. Schaller, and W. Lehner, "Real-Time Scheduling for Data Stream Management Systems," in *Proc. 17th Euromicro Conference on Real-Time Systems*. IEEE, 2005, pp. 167–176.

[25] Y. Wei, S. H. Son, and J. Stankovic, "RTSTREAM : Real-Time Query Processing for Data Streams," in *Proc. 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2006, pp. 141–150.

[26] M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis, "Preemptive Rate-based Operator Scheduling in a Data Stream Management System," in *Proc. 3rd ACS/IEEE International Conference on Computer Systems and Applications*, 2005, pp. 46–54.

[27] X. Li, Z. Jia, L. Ma, R. Zhang, and H. Wang, "Earliest Deadline Scheduling for Continuous Queries over Data Streams," *2009 International Conference on Embedded Software and Systems*, pp. 57–64, 2009.

[28] J. Gama and M. M. Gaber, Eds., *Learning from Data Streams*. Springer, 2010.

[29] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt : A Language for Streaming Applications," in *Proc 11th International Conference on Compiler Construction*, 2002, pp. 179–196.

[30] R. Müller, "Data stream processing on embedded devices," Ph.D. dissertation, ETH Zurich, 2010.

[31] C. Cranor, T. Johnson, and O. Spataschek, "Gigascope: a stream database for network applications," in *Proc. 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 647–651.

[32] H. Kargupta, R. Bhargava, K. Liu, M. Powers, P. Blair, S. Bushra, J. Dull, K. Sarkar, M. Klein, M. Vasa, and D. H. Vedas, "VEDAS: A mobile and distributed data stream mining system for real-time vehicle monitoring," in *Proc 4th SIAM International Conference on Data Mining*, 2004, pp. 300–311.

[33] H. Kargupta, K. Sarkar, and M. Gilligan., "Minefleet: an overview of a widely adopted distributed vehicle performance data mining system," in *Proc, 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 37–46.