

Design and Implementation of a Middleware for Development and Provision of Stream-based Services

Seungwoo Kang¹, Youngki Lee¹, Sunghwan Ihm², Souneil Park¹, Su-Myeon Kim³, Junehwa Song¹

¹Computer Science, KAIST, Korea

{swkang, youngki, spark, junesong}@nclab.kaist.ac.kr

²Computer Science, Princeton University, USA

sihm@cs.princeton.edu

³SAIT, Korea

sumyeon.kim@samsung.com

Abstract— This paper proposes MISSA, a novel middleware to facilitate the development and provision of stream-based services in emerging pervasive environments. The stream-based services utilize voluminous and continuously updated data streams as their input. The characteristics of data streams bring new requirements on the development and provision of the services. To satisfy the requirements, a unique service model and a runtime system are designed in MISSA. The key concept of our service model is to separate service logic from handling data streams. This significantly mitigates the burden on service developers by allowing them to only concentrate on the service logic. Job of handling data streams is completely delegated to the runtime. In this paper, we first present the importance of stream-based services and their requirements. Also, we describe a best route finding service as an example to motivate the need for MISSA. Then, we envision overall architecture for provisioning of stream-based services and detail our design of service model and runtime. Lastly, we demonstrate the efficiency of service model and runtime through experiments.

Keywords— Pervasive computing; sensor network; data stream; stream-based services; middleware; service model; runtime environment

I. INTRODUCTION

Due to advances in wireless communication and embedded device technologies, many useful data are generated and collected in various ways. Among them, continuous data streams generated by electronic devices such as sensors, RFID readers, and GPS devices are coming into the spotlight. This opens up new opportunities to provide value-added stream-based services in diverse application domains, e.g., location-based services (LBS), intelligent transportation services (ITS), intelligent logistics services, environmental monitoring, medical monitoring services, and so on [18][22][23]. For example, a service which provides traffic status exploits a large amount of data streams generated by traffic sensors deployed on the road or devices equipped in vehicles. Also, a child tracking or friend finder service is based on continuously changing location information.

It is totally different from conventional applications to develop and provide emerging stream-based services. It mainly results from the unique characteristics of data streams,

e.g., unbounded, diverse and voluminous. However, there has not been much effort to facilitate the development and provision of stream-based services. This motivates us to put our effort on support for the services. As a first step, we envision new and unique requirements of the stream-based services.

Abstraction of data stream processing: In addition to core computation logic, stream-based services require extra modules to properly receive and manage streams of data. It is arduous burden for a service developer to make such modules whenever she develops new stream-based services. Thus, freeing the developer from the details of data stream processing is an essential requirement.

Stream source independence: For a single service, multiple data stream sources can provide data streams of the same context. In such an environment, a service provider selects an appropriate stream source for the service, and even switches the stream source to meet QoS constraints or enhance business revenue. Thus, it is extremely important to make the service logic reusable with various stream sources.

Changeable execution mode: The stream-based services can run in two execution modes: push and pull. A push mode service continuously provides responses to users for a certain period of time, whereas a pull mode service gives one-time response upon a user request. Although the services differently interact with the users in two cases, core computation logic to generate results is almost the same. It is cumbersome for a developer to write different service logic in order to support each execution mode of a service. Thus, it is highly desirable to support different execution modes with the same service logic.

Efficient stream buffering: To serve user requests, a service computes its logic with a finite set of data. To obtain and maintain the necessary data set from unbounded and voluminous data streams, an efficient buffering mechanism is essential. The buffering mechanism should be capable of handling time-varying and unpredictable data rates. In addition, various buffering semantics should be considered since services need different set of data for the computation.

Concurrent long-lived service session: Session time of stream-based services is usually long, specifically in the push mode. This is mainly because the services continuously execute service logic to monitor users' interested information. Also, many service instances are concurrently alive in the

system to handle multiple user requests. Therefore, it is important to support lots of concurrent long-lived service sessions.

To address the above requirements, we propose MISSA, a middleware for easy development and provision of the stream-based services. MISSA includes a service model and a runtime. The MISSA service model defines programming interfaces and configurations to develop stream-based services and specifies how to implement them. The MISSA runtime provides a computing environment to run the services developed according to the MISSA service model.

The MISSA service model has three key features for easy service development and deployment. First, the MISSA service model requires a service developer only to write a service unit which includes service logic and a service description. The service developer does not need to concern about details of data stream processing such as stream source binding, network I/O processing, and stream buffering. Second, the service unit is developed without specifying data stream sources. Through simple configuration, a service provider can easily bind desirable sources for input of the service unit. Third, the service developer only needs to implement a single service unit regardless of execution modes. The service provider can configure an execution mode when activating the services.

The MISSA runtime has several unique characteristics. Most important, the MISSA runtime is an abstraction layer of managing data streams, which are essential for stream-based services. It takes care of stream source binding, network I/O processing, and stream buffering. Especially, the MISSA runtime provides an efficient buffering mechanism. For efficient stream buffering, the MISSA runtime concentrates on two aspects. First, it supports various buffer semantics for diverse needs of services, e.g., sampling conditions on a data stream. Currently, we support three types of buffering: buffering of recent N consecutive data, the most recent data per distinct key value, and N recent data per distinct key value. Second, the MISSA runtime supports buffer sharing among services if the services exploit similar data streams. The buffer sharing improves resource utilization and avoids redundant data delivery.

MISSA is differentiated from the previous research on data stream processing [1][2][3][4][5] in that MISSA uniquely targets on facilitating development and provision of generic stream-based services. The previous research concentrates on developing new query models and prototype systems for efficient processing of data streams. It is very difficult to develop and run a generic stream-based service by only using the declarative continuous query languages and stream processing engines (SPEs). Most value-added services need complex computation logic beyond query operations such as selection and join. For example, a best route finding service needs a Dijkstra’s shortest path algorithm which is infeasible to be expressed in the query language. MISSA is designed to provide more value-added services beyond the query operations provided. We envision that the role of the SPEs will be data processing and the role of MISSA will be service provisioning. MISSA and the

TABLE I. LINES OF CODE (LOC) OF COMPONENTS FOR A BEST ROUTE FINDING SERVICE

Service components	Lines of Code
Traffic stream resolver	10
Network I/O handler	196
Buffer manager	118
Etc. (server module, instance handler ...)	103
Best route finding algorithm	169
Total	596

SPEs will be complementary to each other for proliferation of the stream-based services.

The rest of the paper is organized as follows. Section II describes a motivating example. Section III briefs the architectural overview of MISSA. Section IV explains the MISSA service model in detail. Section V presents the design of the MISSA runtime. Section VI presents our current implementation and experimental results. Finally, Section VII discusses the related work and Section VIII concludes the paper.

II. MOTIVATING EXAMPLE

Data streams in various contexts bring unprecedented opportunities to create useful services. There already have been several example scenarios of stream-based services such as supply chain management [5], emergency medical service [18]. We strongly believe that the stream-based services will be widely required in the near future as continuous and real-time data collection becomes available. This section discusses an example of traffic information service in greater detail and motivates the need for MISSA.

Traffic information services provide traffic data of a certain region such as average speeds of road segments, events on roads, and weather information for numerous users through car navigation systems or web. Currently, many commercial services [19][20][21] providing traffic information have already become popular in daily life and proved their usefulness. In the near future, these services will utilize more fine-grained and real-time traffic data streams from RFIDs and sensors deployed on the roads and cars equipped with GPS to generate precise and value-added information. For example, a best route finding service will predict the most appropriate path for users based on real time road condition, weather, and current events obtained from sensors and various devices.

To develop such a service, a service developer will encounter new challenges. For empirical data, we actually implemented a simple best route finding service over simulated traffic data streams. Table I describes the implemented components for the service and their code sizes. First, traffic stream resolver contacts a stream source and requests for necessary data. In our example, the stream source sends a tuple of (road_id, segment_id, direction, speed) every 0.1 second. Network I/O handler receives those

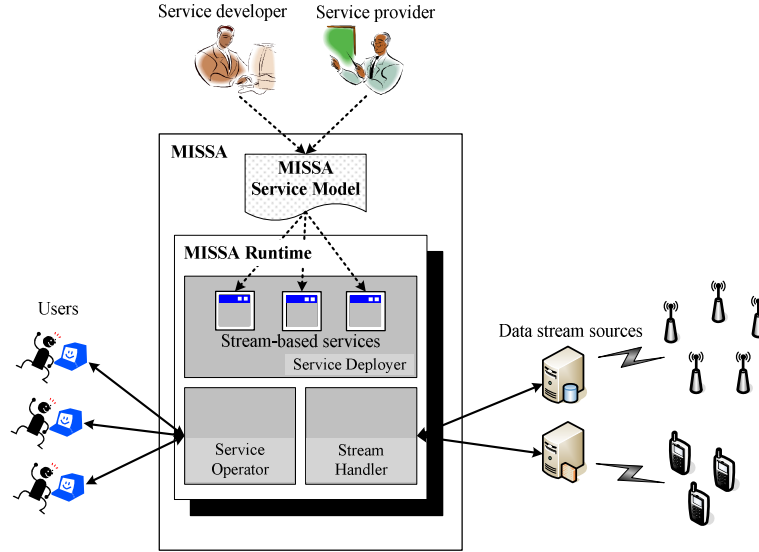


Figure 1. Overall architecture of MISSA.

tuples using non-blocking I/O and transfers them to the buffer manager. In many cases, network I/O handler is not easy to implement since variable size tuples are continuously inputted from multiple sources. Also, naïve I/O mechanism may defer the service delivery by blocking other components.

The best route finding algorithm requires an average road speed of every road segment to generate complete and correct result. Thus, the buffer manager buffers recent tuples for every road segment. The efficient collection and updates of the required data entail much effort on a developer due to the variable and high rates of data streams. Besides handling data streams, we should also consider continuous executions of the algorithm to serve a user who requires real-time updates of new best paths. In addition, it is troublesome to implement supplementary code for server functionality to deal with concurrent multiple users. In the table, the core algorithm is only a small part of entire service; it converts data in the buffer into graphs representing a map and computes the best route. Note that the large part of the service is taken by the components for traffic data retrieval, network I/O, and data buffer management.

These technical difficulties put heavy burden on a service developer even for the creation of the simple best route finding service. To alleviate the difficulties and expedite the implementation process, system-level support is highly desirable. Motivated by the needs, we design MISSA service model and runtime, which significantly accelerates and ease the service development and provision.

III. ARCHITECTURE OVERVIEW

Fig. 1 shows the overall architecture of MISSA. Externally, MISSA interacts with four parties. They are service developers, service providers, users, and data stream sources. The service developers write service logic of a stream-based service according to the MISSA service model. The service providers host a computing node to provide the stream-based service for users. They couple the service logic

developed by the service developers and appropriate data stream sources to make the service logic executable. Note that we separate the role of the service developers from the service providers although the service providers can play a role of service development. This is advantageous to the service providers because they can reuse the service logic developed by the third-party service developers. As a result, the service provider can quickly build new services reusing existing service logic and provide them to the users.

The users request the services provided by the service providers and receive their responses. The data stream sources provide various data streams such as location data streams and traffic data streams to the service providers. We envision that the sources will be the third-party entities independent of the service providers. Currently, service providers and data sources are tightly coupled in general. That is, the service providers have their own data sources used for service provision. However, in the near future, data sources will be pervasive. It will be a new business opportunity to independently provide data for various services. Thus, it is important to consider the data sources separately from the service providers and to support the service development and deployment in such an environment. We assume that the sources are able to process continuous queries over raw data streams generated by sensors or mobile devices. Stream processing engines or sensor proxies are examples of the sources.

MISSA includes the MISSA service model and the MISSA runtime. The MISSA service model defines how to implement service logic. In addition to the service logic, it specifies three configuration scripts, each of which is written by the service developer, service provider, and stream sources, respectively for easy coupling of services with sources and simple service provision. The MISSA runtime is a computing environment that hosts the stream-based services developed in conformity with the MISSA service model. It is comprised of three major components – Service

Deployer, Stream Handler, and Service Operator. The Service Deployer takes charge of the functions to register the services developed by the service developer and deploy the services configured by the service provider. The Stream Handler interacts with the data stream sources to receive the data streams necessary for the services. The Service Operator computes service logic in order to respond to the users. In the following sections, we describe the MISSA service model and present the design of the MISSA runtime in detail.

IV. MISSA SERVICE MODEL

In this section, we detail our novel service model for stream-based services, which maximizes reusability and re-configurability of the services. There are three design principles of the MISSA service model. First of all, our model clearly separates the roles of participating parties: service developers, data stream sources, and service providers. Secondly, it supports dynamic binding of service logic and data stream sources at runtime with description languages. Finally, it allows the service developers to write service logic independently of data stream sources and execution modes. Based on these principles, we design the MISSA service model that consists of three components: Service Unit (SU), data stream source, and Runnable Service Unit (RSU). Fig. 2 shows the components and their relationship.

The role of a service developer is to create a SU that includes service logic and a Service Unit Description Language (SUDL) file which describes the service logic. Since the SU and its required runtime parameters are coupled not in the development phase but in the execution phase, the SU can be easily reused. Data stream source providers write and advertise Stream Source Description Language (SSDL) files which describe the semantics of their data streams. The SU can be bound to any data stream source whose data stream semantic conforms to the input semantic of the SU. The binding is not hard coded so that it is easy to select and replace data stream sources. A service provider writes a Service Unit Activation Language (SUAL) file which specifies a SU, SSDLs, and the runtime parameters in order to provide a stream-based service. Based on the specified information, the MISSA runtime binds the SU and data stream sources described in the SSDLs. Then, it operates the stream-based service according to the runtime parameters. Since the binding and the runtime parameters can be re-configured in the execution phase, the services can be flexibly operated.

A. Service Unit (SU)

1) Service Logic (SL)

SL is the core business logic of a stream-based service. Service developers can make the SL by implementing *ServiceLogic* interface as shown in Fig. 3. There are five abstract functions to implement: *init()*, *uninit()*, *process()*, *getBufferSemantic()*, and *tupleHandler()*. These functions are classified into two groups. The first three functions are related to service execution, the others are related to data stream handling.

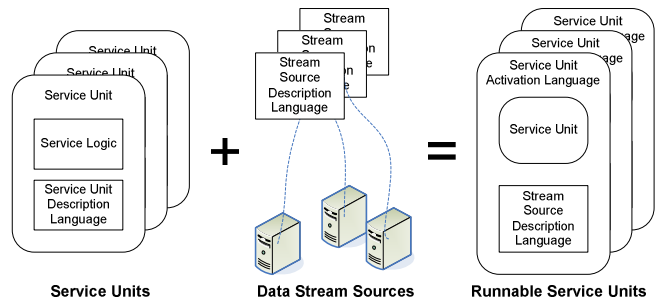


Figure 2. Components of the MISSA service model.

```

public abstract interface ServiceLogic {

    // constants
    public static final int BUFSEM_1 = 0x00000001;
    public static final int BUFSEM_2 = 0x00000002;
    public static final int BUFSEM_3 = 0x00000003;

    // functions related to service execution
    public abstract boolean init();
    public abstract boolean uninit();
    public abstract Object process(Object[] userParam);

    // functions related to data handling
    public abstract Object getBufferSemantic();
    public abstract boolean tupleHandler(Object[] tuple);

}

```

Figure 3. Service Logic interface.

For the service execution, *init()* handles initialization of member variables or resource allocation which are performed at the beginning of the service execution. *process()* expresses main service logic to generate responses. *uninit()* is in charge of cleaning up or releasing resources at the end of the service execution.

For the data stream handling, *getBufferSemantic()* specifies the semantics of buffer which service logic requires. Return value of the function includes a type of buffer semantic and corresponding parameters such as maximum buffer size and key attributes. Currently, the MISSA service model supports three types of buffer semantics. They are recent N tuples ordered by their arrival times (*BUFSEM_1*), tuples categorized by their key attributes (*BUFSEM_2*), and recent N tuples categorized by their key attributes (*BUFSEM_3*). *tupleHandler()* specifies how to update data tuples in the buffer into data structures such as lists or trees used in *process()*. The update of the data tuples is automatically performed by the MISSA runtime. Additionally, the MISSA service model provides *MISSA Data Stream Connectivity (MDSC) API* which has the same functionality as a standard SQL database access interface such as JDBC API. Using the API, service developers can access and retrieve data as if they are programming over persistent databases.

With the implemented functions, the MISSA runtime runs the service as follows. At first, it invokes *init()* for service initialization. During the initialization, the runtime

```

<?xml version="1.0"?>
<serviceUnit>
  <name>Shortest Path Service</name>
  <description>.....</description>
  <inputSchema>
    <name>input1</name>
    <description>.....</description>
    <element name="id" type="int"/>
    <element name="time" type="time"/>
    <element name="averageSpeed" type="int"/>
  </inputSchema>
  <inputSchema>
    <name>input2</name>
    .....
  </inputSchema>
  <outputSchema>
    <description>.....</description>
    <element name="id" type="int"/>
    <element name="time" type="time"/>
    <element name="averageSpeed" type="int"/>
  </outputSchema>
  <userParameter>
    <description>.....</description>
    <element name="startPoint" type="string"/>
    <element name="endPoint" type="string"/>
  </userParameter>
  .....
</serviceUnit>

```

Figure 4. An example of SUDL.

prepares buffers required by the service, whose semantic is obtained by calling *getBufferSemantic()*. While running a service, *process()* is invoked after calling *tupleHandler()* whenever a tuple arrives in a push mode. In a pull mode, *process()* is called after *tupleHandler()* is called over all currently buffered tuples upon receiving a user request. When finishing the service, *uninit()* is called by the runtime for service termination.

2) Service Unit Description Language (SUDL)

SUDL is an XML file which describes a SU. It contains a service name, an input schema, an output schema, user parameters, and all other related information about the SU. The input schema represents the schema of data streams which the SL expects to receive, and the output schema stands for the schema of processed results which the SL generates. Service providers refer the SUDL file when using the SU, just like accessing the Web Service Definition Language (WSDL) [12] file in Web Services. Fig. 4 depicts an example of SUDL.

B. Stream Source Description Language (SSDL)

The SSDL file represents information of a data stream source. It contains the name, location, and data stream schemas of the data stream source. We assume that the SSDL is registered in a central registry such as UDDI [13] or distributed registry [15][16][17]. Service developers/providers can search the registry and access appropriate stream sources they need to use. Fig. 5 shows an example of SSDL.

C. Service Unit Activation Language (SUAL)

SUAL acts like glue which associates a SU and SSDLs. It contains the location of the SU, the locations of the SSDL files, the queries that will be imposed on the data streams specified in the SSDLs, and execution mode. It is the queries

```

<?xml version="1.0"?>
<streamSource>
  <name>Traffic Information</name>
  <description>.....</description>
  <location>192.168.0.1:1234</location>
  <schema>
    <name>roadTraffic</name>
    <description>.....</description>
    <element name="roadID" type="int"/>
    <element name="avgSpeed" type="int"/>
    <element name="timeStamp" type="time"/>
  </schema>
  <schema>
    <name>carTraffic</name>
    .....
  </schema>
</streamSource>

```

Figure 5. An example of SSDL.

```

<?xml version="1.0"?>
<runnableServiceUnit>
  <name>Activated Shortest Path Service</name>
  <description>.....</description>
  <serviceUnit name="sps" location="./shortestPathService.jar"/>
  <dataStreamSource>
    <source name="s1" location="http://192.168.0.1/
    avgRoadSpd.ssd1"/>
    <source name="s2" location="....."/>
  </dataStreamSource>
  <binding target="sps.input1" query="select roadID, timeStamp,
  avgSpeed from s1.roadTraffic"/>
  <binding target="sps.input2" query="....."/>
  <executionMode type="push"/>
</runnableServiceUnit>

```

Figure 6. An example of SUAL.

that link the SU and the input data streams. The schemas of the query results should match the input schemas described in the SUDL file. It is like specifying appropriate arguments when we call a function. The execution mode defines in which mode the service will be executed. There are three options for the execution mode – push, pull, and both. Fig. 6 illustrates an example of the SUAL.

V. MISSA RUNTIME

This section describes the architecture of the MISSA runtime. First, we show the internal component design of the MISSA runtime and the processing flow among the components. Then, we present several issues of the components in detail.

A. Service Processing Flow

Fig. 7 depicts the three major components of the MISSA runtime, sub-components, and their interactions. The Service Deployer registers and activates the service units. The Service Unit Handler (SUH) receives a service unit from a service developer or a SUAL file from a service provider. When it receives a service unit, it stores the service unit in the Service Unit Registry (SUR) to enable service providers to use the service unit. Upon receiving a SUAL file, the SUH fetches the service unit specified in the SUAL file from the SUR, and then passes it to the Service Unit Activator (SUA) with the SUAL file.

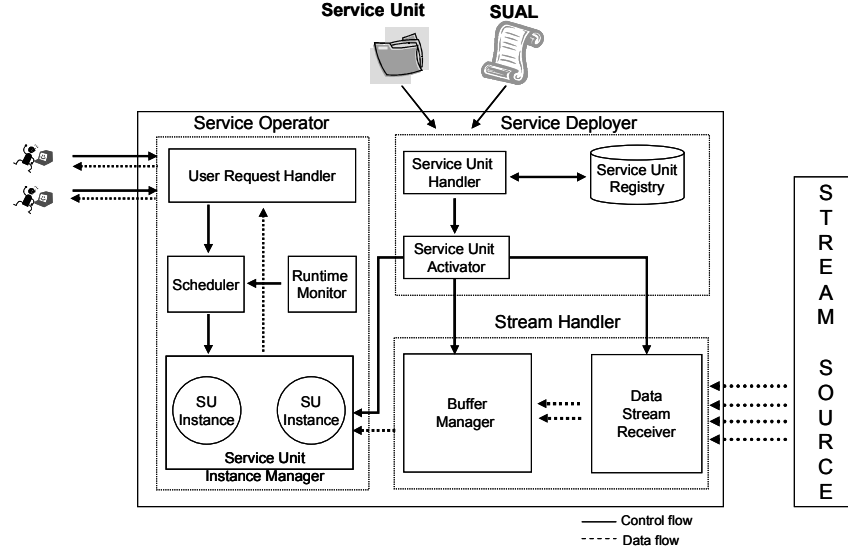


Figure 7. The MISSA runtime.

The SUA bootstraps a service unit in cooperation with the Stream Handler (SH) and the Service Unit Instance Manager (SUIM). The service bootstrapping includes three actions – buffer creation, stream source connection, and instance management initiation. For the buffer creation, the SUA parses the queries in the SUAL file and passes the schema information of the query results to the Buffer Manager. Then, the Buffer Manager creates buffers according to the schemas. For the stream source connection, the SUA provides the Data Stream Receiver (DSR) with the address of data stream sources and queries after interpreting the SSDLs specified in the SUAL file. Then, the DSR establishes connections with the sources and sends the queries in order to receive data. When the buffers are created and connections are established, the SUA signals the SUIM to prepare service instance management. It also informs the execution mode.

If a service unit is ready to execute, the Service Operator serves user requests, while the SH updates data from the sources. The User Request Handler (URH) receives the user requests and forwards them to the Scheduler. The Scheduler selects a request to serve and invokes the SUIM. Subsequently, the SUIM executes a service instance with the buffered data. After the execution, it delivers the result to the user through the URH.

B. Issues in the Functional Components

The SUA has to resolve overlapping answers of queries to reduce redundant transmission of data. For example, answers of a query asking road-id, speed, and timestamp will contain the answers of a query asking road-id, and speed. Redundant transmission of these answers is inefficient in terms of network I/O cost and memory consumption. To address this problem, the SUA supports query merging as in [14]. For this, it maintains a list of running queries. Upon a new query, the SUA determines if the query has overlapping answers with a query in the list. If so, it merges the query

with the existing query in the list. Also, it adjusts the running query in the DSR and the corresponding buffer in the BM. The detailed mechanism for the query merging is under development.

The buffer manager allocates one buffer for each query, and manages the buffered data according to the buffer semantic specified in the service logic. It also updates the buffered data to the service instances. For each update, the BM dynamically transforms an incoming data tuple to the input of the service logic through type conversion and attributes mapping. Since one buffer is created for one query in a service unit, the buffer is shared by multiple instances of the service. This saves the memory consumed for stream buffering.

The SUIM manages instances differently according to the execution mode. Since push mode services retain connections with users unlike pull mode services, we consider push mode services are stateful and pull mode services are stateless. For push mode services, the SUIM creates an instance per user request to serve the request during the specified life time. The instance is destroyed when the life time expires. On the other hand, the SUIM manages an instance pool and fetches an instance from the pool to serve pull mode service requests. After a result generation, it returns the instance back to the pool.

VI. IMPLEMENTATION AND EXPERIMENT

A. Implementation

We implemented an initial version of the MISSA runtime with JAVA, which includes all core components described in Section V. Among them, the Data Source Receiver (DSR) is implemented using non-blocking I/O multiplexing to handle multiple data streams efficiently. The Buffer Manager (BM) is implemented with hash tables and linked lists to deal with three buffer semantics. In the Service Unit Instance Manager

TABLE II. LINES OF CODE (LOC) OF TWO SERVICES

	w/o MISSA	w/ MISSA
Network I/O	206	0
Buffering	118	0
Etc.	103	0
Service Logic	169	186
Total	596	186

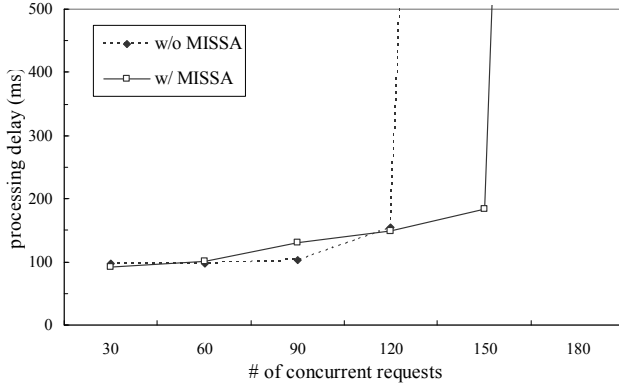


Figure 8. Processing delay over the number of concurrent requests.

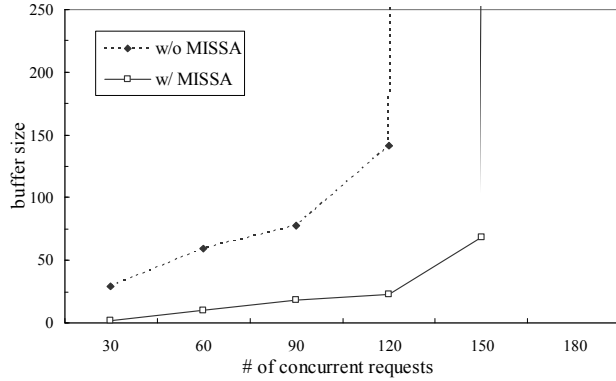


Figure 9. Buffer size over the number of concurrent requests.

(SUIM), a thread pool is used to avoid unnecessary thread creation and termination.

As an example service, we adopted the best route finding service described in Section II. For comparison, we implemented the service in two types: the service based on MISSA (*w/ MISSA service*) and the service without MISSA (*w/o MISSA service*). Note that the *w/o MISSA service* is different from the *w/ MISSA service* in that it creates a buffer and a connection with a data source upon each user request. To serve user requests, both services run Dijkstra's shortest path algorithm upon an incoming data tuple. We made a simple data generator and a user request generator for experiments. The data generator sends data streams in a configured rate, and the user request generator issues multiple requests at the same time.

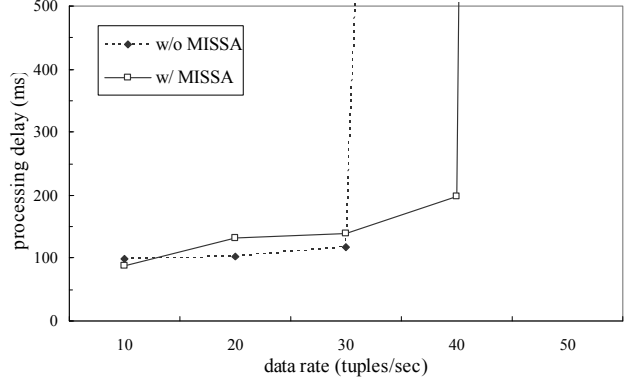


Figure 10. Processing delay over data rate.

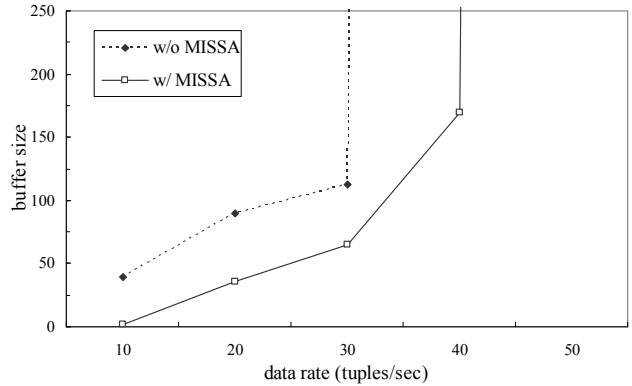


Figure 11. Buffer size over data rate.

B. Experimental Setup

The services run on the machine equipped with a 1.81 GHz CPU, 512MB RAM and Window XP. The data generator and the user request generator run on the machine equipped with a 2.4 GHz CPU, 256MB RAM and Linux. The machines are connected via 100Mbps link.

We evaluate the benefits of MISSA in two aspects: convenience of development and efficiency of service operation. We measure lines of code (LOC) to show the convenience of development. For efficiency metrics of the MISSA runtime, we measure buffer size and processing delay as functions of the number of concurrent requests and data rates. The buffer size is the number of data tuples in the buffers of each service, and the processing delay is the average time taken to generate a result upon an incoming data tuple.

C. Experimental Results

1) Convenience of Development

Table II shows LOC of two services. The total LOC of the *w/ MISSA service* is only 31.2% of the *w/o MISSA service*. The difference is caused since the *w/o MISSA service* contains more code for connection and buffer management in addition to the service logic. To write such code, specific knowledge on low-level system details and socket-level network programming is required. Generally, it is much more complicated to develop such parts compared to

the service logic. Since the MISSA runtime takes charge of all those burdens, the w/ MISSA service only contains the service logic. Although MISSA requires the developers to follow the MISSA service model, e.g., writing SUDL files, those jobs are relatively simple and can be automated. Thus, the developers can concentrate on the service logic.

2) Efficiency of Convenience of Development

We measured processing delay and buffer size while varying the number of concurrent requests. The number of concurrent requests was increased from 30 to 180 and the data rate was fixed to 10 tuples per second. Fig. 8 and Fig. 9 show the experimental results.

Both services show acceptable processing delay and small buffer size until the number of concurrent requests reaches 120. The processing delay slightly increase as the number of concurrent requests grows, ranging from 90 ms to 160 ms. Buffer size also grows similarly, however, the rate of increase of the w/o MISSA service is faster than that of the w/ MISSA service.

When the number of requests exceeds 120, the w/o MISSA service reaches the limit of processing capacity. Thus the processing delay and the buffer size increase exponentially. The w/o MISSA service performs poorly since it maintains a separate buffer and a data source connection for each user request. The buffer size increases since duplicate data are maintained in individual buffers. Since more data have to be processed, the processing capacity is reached earlier. On the other hand, the w/ MISSA service handles up to 150 concurrent requests since it shares the data source connection and the buffer.

We measured processing delay and buffer size while varying the data rates. The data rate was increased from 10 to 50 data tuples per second and the number of concurrent requests was fixed to 50. The results are depicted in Fig. 10 and Fig. 11. Similar to the above result, the w/ MISSA service exhibits better processing capacity due to the sharing effect. Tolerating high data rate is desirable since numerous data stream sources will generate data at high data rate in a real environment.

Table III shows processing capacities of two services. We assume that a service exceed the capacity when the processing delay is longer than 1000ms. The w/ MISSA service stably supports 21% more requests and 52% faster data rate compared to the w/o MISSA service. The performance gap between two approaches will grow as the sharing effect increases.

VII. RELATED WORK

Extensive research on data stream processing and its application has been conducted [1][2][3][4][5]. In Aurora [1], TelegraphCQ [2], STREAM [3] project, they proposed new query models and stream processing engines (SPEs). A common feature of the proposed SPEs is to process continuous queries over voluminous data streams in an adaptive and resource efficient manner. However, there has not been much effort to support for easy development of generic stream-based services. Most value-added services need complex computation logic beyond query operations such as selection, join, and aggregation. MISSA targets on

TABLE III. PROCESSING CAPACITY OF TWO SERVICES

	w/o MISSA	w/ MISSA
Maximum number of concurrent requests	137	166
Maximum data rate (tuples per sec)	31.3	47.6

simplifying development and provision of generic stream-based services beyond the query operations. We envision that MISSA and the SPE will be complementary to each other for proliferation of stream-based services.

For developing and executing context-sensitive applications, iQueue has been proposed. It includes a data composition framework [6] and a programming language for such applications [7]. iQueue targets on the applications based on networked pervasive data sources which are intermittent and heterogeneous. Thus, iQueue mainly supports continuous discovery and binding of the proper data sources. In contrast, MISSA targets on the environment where data streams are voluminous and rapidly incoming. In order to hide the complex management of such data streams, MISSA separates the stream processing from service logic; it supports efficient data stream management, e.g., data stream buffering. In addition, MISSA concentrates on reusability and re-configurability for the stream-based services, which iQueue does not mainly deal with.

As popular service platforms, Web Services [24], J2EE [9] and OSGi [8] are widely used. They provide standardized and component-based computing environments for networked services. However, they are not feasible solutions to emerging stream-based services. Data streams are not considered as input for services so that they lack in system support such as stream handling and diverse execution modes support. Although they do not target on the stream-based services, integrating the features of MISSA into the service platforms will broaden their application areas.

There have been much research efforts to develop efficient publish-subscribe systems such as Gryphon [25], Siena [26], Sieve [27], and SemCast [28]. Also, there have been standardization activities for publish-subscribe middleware systems such as OMG Data Distribution Service (DDS) standard specification [29] and commercial products based on the specification [30][31]. Publish-subscribe systems enable data dissemination among multiple data producers and data consumers. They create data distribution paths regarding different consumers' interests and network conditions, supporting content-based data filtering. They have been applied to a variety of application domains [32][33]. While these systems mainly focus on construction and maintenance of distributed data dissemination networks, MISSA concentrates on providing a development and execution environment for services based on continuous data streams. MISSA automatically loads or removes application service instances upon user requests, runs the instances according to its execution mode, and enable buffer sharing among the instances.

Finally, we give an overview of the previous works on middleware to support applications with data streams. It includes Grid-based distributed data stream processing

middleware and RFID middleware. In GATES project [10], they proposed a Grid-based middleware for applications requiring real-time analysis of data streams from distributed sources. GATES employs an adaptation algorithm to analyze data with the highest accuracy while meeting a real-time constraint. However, they do not consider how to deal with various challenges in developing and providing the stream-based services. Auto-ID center proposed a specification for RFID middleware, Savant [11], and some companies introduced commercial RFID middleware. The RFID middleware continuously obtains RFID tag data through RFID readers and forwards processed data to applications. The main focus of the RFID middleware is to correct erroneous data and reduce the large volume of data through the operations such as filtering, counting and aggregation. The RFID middleware concentrates on low level data processing rather than high level support for generic stream-based services.

VIII. CONCLUSION

Data streams generated by lots of sensors or mobile devices envision the prevalence of stream-based services. However, there was little consideration for the development and provision of stream-based services. This motivates us to design a novel middleware, MISSA that facilitates the development and provision of stream-based services. In this paper, we presented the essential requirements of stream-based services and described them through the best route finding service example. To meet the requirements, we proposed the MISSA service model that facilitates the development of stream-based services and the MISSA runtime that operates the services efficiently. Our experimental results show that our service model and runtime is greatly advantageous.

REFERENCES

- [1] D. Carney, et al., "Monitoring Streams: A New Class of Data Management Applications," Proc. VLDB, Hong Kong, August 2002.
- [2] Sirish Chandrasekaran, et al., "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," Proc. CIDR, January 2003.
- [3] R. Motwani, et al., "Query Processing, Resource Management, and Approximation in a Data Stream Management System," Proc. CIDR, January 2003
- [4] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," Stanford Technical Report, 2003
- [5] M. Franklin, et al., "Design Considerations for High Fan-in Systems: The HiFi Approach," Proc. CIDR, 2005
- [6] Norman H. Cohen, et al., "iQueue: a pervasive data-composition framework," Proc. MDM, Singapore, January 2002, pp. 146-153.
- [7] Norman H. Cohen, et al., "Composing pervasive data using iQL," Proc. WMCSA, New York, June 2002, pp. 94-104.
- [8] OSGi™ Alliance, "OSGi Service Platform Release 4," <http://www.osgi.org/Specifications/HomePage>
- [9] Java 2 Platform, Enterprise Edition (J2EE), <http://java.sun.com/j2ee/index.jsp>
- [10] L. Chen, et al., "GATES: A Grid-Based Middleware for Distributed Processing of Data Streams," Proc. HPDC, Hawaii, June 2004.
- [11] Sean Clark, et al., "Auto-ID Savant Specification 1.0," <http://www.rfida.com/nb/autosavant.htm>
- [12] Web Service Definition Language, <http://www.w3.org/TR/wsdl>
- [13] Universal Description, Discovery and Integration, <http://uddi.xml.org/uddi-org/>
- [14] Arturo Crespo, Orkut Buyukkokten, and Hector Garcia-Molina, "Query Merging: Improving Query Subscription Processing in a Multicast Environment," Proc. ICDE, 2003.
- [15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," Proc. the ACM SIGCOMM, August 2001.
- [16] Java Naming and Directory Interface (JNDI). <http://java.sun.com/products/jndi/>
- [17] P. Mockapetris, "Domain Name Standard: RFC 1034," November 1987.
- [18] J. Shneidman, et al., "Hourglass: An Infrastructure for Connecting Sensor Networks and Applications," Harvard Technical Report TR-21-04, 2004
- [19] MBC idio, <http://i-dio.imbc.com/>
- [20] RealTraffic Technologies, <http://www.realtrafficech.com/>
- [21] San Diego Area Traffic Report, <http://www.dot.ca.gov/dist11/d11tmc/sdmap/mapmain.html>
- [22] Chun-Te Wu and Hsing Mei, "Location-Based-Service Roaming Based on Web Services," Proc. the 19th International Conference on Advanced Information Networking and Applications, 2005, pp. 277-280.
- [23] Aloizio P. Silva and Geraldo R. Mateus, "Location-Based Taxi Service in Wireless Communication Environment," Proc. the 36th Annual Simulation Symposium, 2003.
- [24] Web services, <http://www.webservices.org>
- [25] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems," Proc. ICDCS, 1999.
- [26] A. Carzaniga, M. Rutherford, and A. Wolf, "A Routing Scheme for Content-based Networking," Proc. IEEE INFOCOM, 2004.
- [27] S. Ganguly, S. Bhatnagar, A. Saxena, R. Izmailov, and S. Banerjee, "A Fast Content-Based Data Distribution Infrastructure," Proc. IEEE INFOCOM, 2006.
- [28] O. Papaemmanouil, and U. Cetintemel, "SemCast: Semantic multicast for content-based data dissemination," Proc. IEEE ICDE, 2005.
- [29] Object Management Group (OMG), "Data Distribution Service for Real-Time Systems (DDS), version 1.2," <http://www.omg.org/>
- [30] Real-Time Innovations, Inc., "RTI Data Distribution Service," <http://www.rti.com/>
- [31] OpenSplice, "OpenSplice DDS," <http://www.opensplice.com/>
- [32] Eduardo R. B. Marques, Gil M. Gonçalves, João B. Sousa, "The use of real-time publish-subscribe middleware in networked vehicle systems," Proc. IFAC Workshop on Multivehicle Systems, 2006.
- [33] Marco Ryll and Svetan Ratchev, "Application of the Data Distribution Service for Flexible Manufacturing Automation," International Journal of Mechanical, Industrial and Aerospace Engineering, 2:3, 2008.