한국컴퓨터정보학회 동계학술대회 논문집 제27권 제1호 (2019. 1)

Splash의 스트림 프로세싱 기능 구현

안재호[O], 노순현[*], 홍성수[*]
[O*]서울대학교 전기 정보 공학부
e-mail: {jhahn[O], shnoh[*], sshong[*]}@redwood.snu.ac.kr

Implementing stream processing functionalities of Splash

Jaeho AhnO, Soonhyun Noh[*], Seongsoo Hong[*]
[O*]Dept. of Electrical and Computer Engineering, Seoul National University

● 요 약 ●

To accommodate for the difficult task of satisfying application's system timing constraints, we are developing Splash, a real time stream processing language for embedded AI applications. Splash is a graphical programming language that designs applications through data flow graph which, later automatically generates into codes. The codes are compiled and executed on top of the Splash runtime system. The Splash runtime system supports two aspects of the application. First, it supports the basic stream processing functions required for an application to operate on multiple streams of data. Second, it supports the checking and handling of the user configurated timing constraints. In this paper we explain the implementation of the first aspect of the Splash runtime system which is being developed using a real time communication middleware called DDS.

키워드: stream processing, runtime system implementation, modular programming

I. Introduction

Overtime, a growing number of safety-critical systems have employed embedded AI to analyze sensor data in a timely manner[1][2]. For embedded AI applications to constantly consume multiple sensor data and quickly produce an accurate results, it often leverages real time stream processing[3][4][5].

For an application to support real time stream processing it is crucial for it to guarantee the system's timing constraints[6]. The development process for such applications requires an arduous and ad hoc task of independently testing for each timing constraint. This can lead to an overlooked exception and cause critical failures to occur.

To provide a solution to this problem we are currently developing Splash, a stream processing language geared towards supporting embedded AI applications. Splash differentiates itself from other stream processing languages by allowing users to configure the required timing constraints at the language level. Splash automatically detects the violation of the configurated timing constraints and allows the users to specify a way to handle the exceptions.

Splash is a graphical programming language that provides the users with data flow graph to visually express the application. Splash provides its own set of language constructs for the user to compose the graph with. Once the design for the application is finished, it is then generated into C++ codes. The C++ codes contain user defined areas which need to be programmed in order to fill in the detailed parts of the application. Once the programming is complete, the C++ codes are compiled and executed on top of the Splash runtime system.

In this paper we describe the implementation of the Splash runtime system. Specifically, we explain how it was implemented to support applications' stream processing functionalities. Its support for the monitoring and handling of the timing constraints will be discussed in a separate literature.

II. Data distribution service

In order to support a real-time and scalable dissemination of information, the OMG (Object Management Group) has defined

a communication middleware standard called DDS (Data Distribution Service)[7]. To present a clear view on how DDS was used to implement the Splash runtime system, this section will briefly introduces the communication model and the key elements of DDS.

1. Communication model

DDS uses the publisher and subscriber paradigm for their communication model. The pub/sub communication paradigm uses a virtual global data space called topic to identify each data stream circulating over the system. A task uses the publisher to send data to a particular topic and all other interested tasks use their subscriber to receive the data through the same topic.

2. Key elements

The key elements of DDS are as follows. A DomainParticipant is the container for all other DDS elements and acts as the entry point for accessing them. In order for other DDS elements to function, it must be connected to a DomainParticipant.

A Publisher manages the actual writing of an output data while a DataWriter act as a communication end point for tasks to request a data write. A DataWriter can associate it self with only one specific topic. N number of DataWriters can connect to a single Publisher.

Similar to a Publisher, a Subscriber manages the actual reading of an input data while the DataReader acts as the communication end point for tasks to request a data read. A DataReader can also associate it self with only one specific topic. N number of DataReaders can connect to a single Subscriber.

A Listener attaches it self to other DDS elements and acts as a generator of asynchronous interrupts. The Listener monitors the attached element for a pre-defined condition to be fulfilled. Once the condition is fulfilled it generates an asynchronous interrupt and calls upon a registered handler. Each condition can register a different handlers.

III. Stream processing implementation of Splash language constructs

Splash provides 3 types of language constructs that constitutes the data flow graph: component, port, and pipe. The component language construct represents the node of the data flow graph and functions as the stream processing operator. The port language construct represents the connection points of the data flow graph and functions as the entrance and exit of the stream data. The pipe language construct represents the edge of the data flow graph and functions as the delivery path for the stream data. This section will explain how each language constructs are implemented to support the basic stream processing of Splash applications.

1. Port language construct

A port language construct connects itself to a component language construct and becomes the communication interface for the component language construct. Each port can only communicate one type of data, therefore the data type and the topic name is chosen as the connection is made A port language construct is further divided into an input port and an output port.

An input port maintains a single queue for storing input data and it is comprised of two tasks that runs simultaneously. The first task receives the input data and stores it into the queue. The second task generates an asynchronous signal each time a data arrives and calls a registered handler. The action of the handler will often be to notify the connected component language construct of the input data, but it can differ depending on the type of the component language construct. The first task utilizes a DataReader to maintain and receive data into a queue. Second task utilizes a Listener to generate an asynchronous interrupts and to call a handler.

An output port also maintains a single queue for storing input data. However, it only requires a single task that stores the output data in its queue and publish it whenever possible. The task is Fig 1. Basic layout of the connection between port and component language construct blocked until the connected component language construct unblocks it to write output data. The task is implemented using a DataWriter.

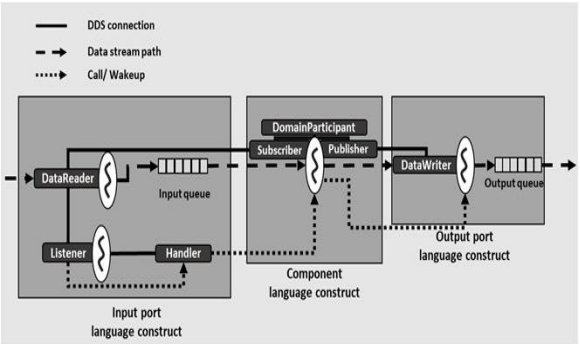

Fig. 1. Basic layout of the connection between port and component language construct

2. Pipe language construct

A pipe is implemented as the topic shared by the DataReader and the DataWriter of the input and output ports. When an

input and an output port is connected by a pipe, the DataWriter of the output port publishes to the topic that the DataReader of the input port subscribes to.

3. Component language construct

All component language construct implementations include a DomainParticipant and one or both Subscriber and Publisher. This allows the DataReader/Writer of the input/output ports to connect to the Subscriber/Publisher of the component language construct. This essentially makes it a node which the port language construct can connect to. Figure 1 displays the connections and the communications that occurs between the ports and the component language construct. There are six types of component language construct that operates on the stream data differently.

3.1 Source block and sink block

A source block receives input from external devices and outputs it into a data stream. Source block allows the user to program their own communication interface for the external devices. This was done to accommodate for the varying external devices Splash application will connect to. This means that the source block does not require any connection to an input port and does not need to carry a Subscriber. Instead, the source block will only connect to an output port.

A source block is comprised of a single task that receives data through the external device's communication interface. The same task will then transform the received external data into an internal data stream and publish it using an output port.

A sink block outputs data from its input stream to an external device. For the similar reason as the source block, it allows the user to program its own output communication interface. Therefore, it does not require a connection to an output port and does not need to carry a Publisher.

A sink block is comprised of a single task that uses the external device's communication interface to output the input data stream. This is implemented using the connected input port. The task is initially blocked, and each time the connected input port receives data, it wakes up the sink block's task and passes the received data.

3.2 Processing block

A processing block executes user defined operations on the input streams. It can connect to multiple input ports and each input port will have its own user defined operations. Only a single data stream can be operated at a time, and each data stream will have a different operations defined for them. The processing block can also connect to multiple output ports. Each user defined operation can access multiple output ports at a time, but they must output the data simultaneously.

A processing block is comprised of a single task that operates on an input stream and outputs the result. This is implemented using the connected input and output ports. The task is initially blocked, and each time a data arrives to an input port, it will wake up the processing block's task and pass the input data. Depending on which input port has received the data, the task will choose one of the user defined operations to process them.

3.3 Fusion operator

A fusion operator merges multiple input streams into a single output stream. Therefore the fusion operator can attach multiple input ports and a single output port. In order to provide a varying option to the users, the fusion operator can designate each individual input port as mandatory or optional. Fusion operator can also set the number of optional input ports that must have received data before the merging operation can begin. This means that fusion operator must have received data from all mandatory input ports as well as received data from the designated number of optional input ports, before the merging can begin.

A fusion operator is comprised of a single task that merges the multiple input data into a single output. This is implemented using the attached input and output ports. When an input port receives data, it notifies the fusion operator's task and passes the data. The fusion operator's task is initially blocked, so when an input arrives it wakes up to check if the condition has been satisfied for merging. If it has not been satisfied, it stores the data into a queue and goes back to a blocked state. If the condition has been satisfied, it chooses the appropriate data from input port queue to merge the data. Once the merging is finished it uses the output port and sends the data out.

3.4 Selection operator

The selection operator receives a single data stream and outputs it to a different data stream based on the user configuration. It can connect to a single input port and multiple output ports.

A selection operator is comprised of a single task that considers the user configuration and outputs the input data to a different data path. The task is blocked until an input data arrives. This is implemented using the attached input port and output ports. Whenever a data arrives to an input port it wakes up the selection operator's task and passes the input data. Selection operator's task then selects an appropriate data path by choosing which output port to dispatch the data with.

3.5 Factory

Factory encompasses all other language constructs and allows the data flow graph to be designed modularly. It can connect to multiple input ports and output ports. Input ports connected to a factory acts as the entry point to a module and disseminates data to the first component language constructs. Output ports connected to a factory acts as the exit point of a module and disseminates the output of the last component language constructs.

A Factory is comprised of multiple tasks that runs simultaneously and connects each external stream to an internal stream and vice versa. For each connected port there will be a task that connects the received external stream to the internal stream vice versa.

4. Evaluation

To evaluate Splash's stream processing capabilities, we have designed a simple version of an adaptive cruise control application for autonomous driving using Splash. It uses synthetic vision and radar data to calculate the desired acceleration. It was designed using 3 fusion operators, 1 selector, and 2 processing block. The application successfully transmitted the stream data across all component language constructs and produced plausible and accurate data. Experiment's hardware and software configuration is given in Table 1.

Table 1. Hardware and software configuration

Hardware	
CPU	Intel Core i7-7700 3.6G Hz
Memory	DDR4 8GB
Storage	25.6GB
Software	
DDS version	OpenSplice DDS 6.9.18
Operating System	Ubuntu 16.04 LTS

V. Conclusion

In this paper we presented the implementations of the Splash runtime system. Specifically we explained how the basic stream processing functions for each language construct was implemented. For our future works, we will add additional functions useful to sensor fusion as well as additional triggering mechanisms for initiating component language constructs.

REFERENCES

[1] Assuncao, Marcos Dias, Alexandre da Silva Veith, and Rajkumar Buyya. "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions." JNCA103 (2018): 1-17.

[2] Cui, Yanling, et al. "Towards Adaptive Sensory Data Fusion for Detecting Highway Traffic Conditions in Real Time." DSFAA. Springer, Cham, 2018.

[3] Zhao, Xin-Hua, et al. "Multifunctional sensor based on porous carbon derived from metal–organic frameworks for real time health monitoring." ACS applied materials & interfaces 10.4 (2018)

[4] Siegel, Joshua E., et al. "Real-time Deep Neural Networks for internet-enabled arc-fault detection." Engineering Applications of Artificial Intelligence 74 (2018): 35-42.

[5] B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them," in IEEE Transactions on Software Engineering, vol. SE-11, no. 1, pp. 80-86, Jan. 1985. doi: 10.1109/TSE.1985.231845

[6] Pardo-Castellote, Gerardo. "Omg data-distribution service: Architectural overview." Distributed Computing Systems Workshops, 2003. Proceedings.