

# Design of a Scalable Data Stream Channel for Big Data Processing

Yong-Ju Lee\*, Myungcheol Lee\*, Mi-Young Lee\*, Sung Jin Hur\*, Okgee Min\*\*

\*Big Data SW Platform Research Department, \*\* Cloud Computing Research Department,  
Electronics and Telecommunication Research Institute,  
Daejeon, South Korea  
{yongju, mclee, mylee, sjheo, ogmin}@etri.re.kr

**Abstract**—This paper outlines big data infrastructure for processing data streams. Our project is distributed stream computing platform that provides cost-effective and large-scale big data services by developing data stream management system. This research contributes to advancing feasibility of big data processing for distributed, real-time computation even when they are overloaded.

**Index Terms**—Data Stream Processing, Big Data Platform

## I. INTRODUCTION

“Big data” is the buzzword of the day. It is a collection of data sets so large and complex that it becomes difficult to process, manage, and analyze data in a just-in-time manner. The explosive growth in the amount of data created in the world continues to accelerate and surprise us. Moreover, big data is more complicated to handle efficiently. An interesting Google Trend big data factoid is that India and South Korea are rated with the highest interest, with the USA a distant third. So, all of the big data vendors should now focus on India and South Korea [1]. In South Korea, the government will set up a new big data center to help its industry catch up with global technology giants. This will be the country’s first center which allows anyone to refine and analyze big data. The big data gets big boost in South Korea. Our project in South Korea is one of the alternative systems that enable to compute, store, and analyze big data. To unlock the stream processing system as key to big data’s potentials, we concerned about the technical barriers and breakthrough ideas.

## II. RELATED WORKS

Today, there are many big data processing frameworks to handle large volume of big data. In top level projects of the Apache Software Foundation (ASF), Apache Flink [2] and Apache Spark [3] are both general-purpose data processing platforms. They have a wide field of application and are usable for dozens of big data scenarios. Apache Spark is based on resilient distributed datasets (RDDs). This in-memory data structure gives the power to sparks functional programming paradigm. It is capable of big batch calculations by pinning memory. Especially, Spark Streaming wraps data streams into mini-batches. It collects all data that arrives within a certain period of time

and runs a regular batch program on the collected data. While the batch program is running, the data for the next mini-batch is collected. Apache Flink on the other hand is optimized for cyclic or iterative processes by using iterative transformation on collections. This is achieved by an optimization of join algorithms, operator chaining and reusing of partitioning and sorting. However, Apache Flink is also a strong tool for batch processing. Apache Flink streaming processes data streams as true streams, i.e., data elements are immediately “pipelined” though a streaming program as soon as they arrive. This allows to perform flexible window operations on streams. Most of all, two platforms must have the efficient communication techniques in parallel processing.

## III. DESIGN

### A. Channel Design Issues

There are many design issues in a big data infrastructure where storage, archiving, and access are scalable and reliable. In this paper, we focus on stream channel design from processing big data streams more efficiently.

#### 1) Multi-instance Task

In a data stream management system (DSMS), multi-task in a service and multi-instance in a task are commonly used. In other words, a service is composed of input, output, and tasks: The input constantly reads new messages and emits them to the stream. The output is also written streams in the specified format (e.g., a file). A task is executed often in response to newly arrived stream from the input or previous task emission. When a task would interfere with processing in other tasks, the task must be necessary to run multiple copies of the source (i.e., a single task). So, a single task splits multi-instance where resides in physical instances. Figure 1 represents an example of multi-instance task. A single instance task *b* is connected to the Task *a*. If the Task *b* has not an appropriate ability to handle streams, the status of Task *b* changes a single instance to multi-instance. A multi-instance task gives transparent operations which are equivalent to a single instance task’s operation. Task *a* just calls a send() method

to communicate with multi-instance *Task b*. In fact, the send () method is provided by our own transport protocol.

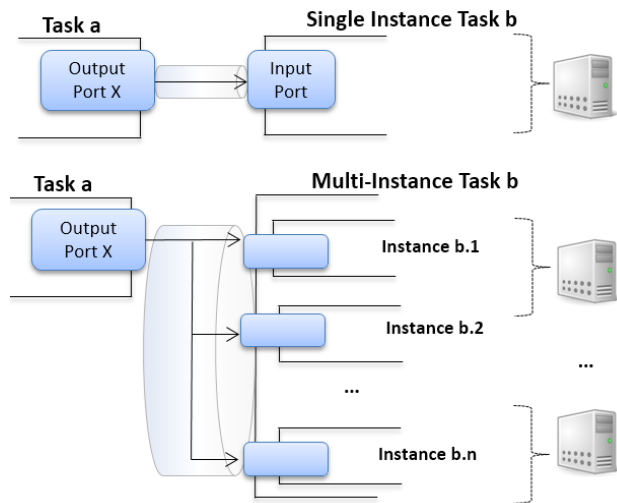


Figure 1. Multi-instance Task

## 2) Data Stream Channel

We design two kinds of basic transport protocols. One is a push/pull method. The other is a pub/sub method. A push method can distribute messages all connected peers which already negotiate the counterpart pulls. A pub method is similar to a multicast protocol. The pub method can spread streams into all connected peers. Using these two basic methods, we design fast-path data stream channel in-depth. It has two kinds of transmission methods. One is a round-robin fashion. The other is a key-value fashion.

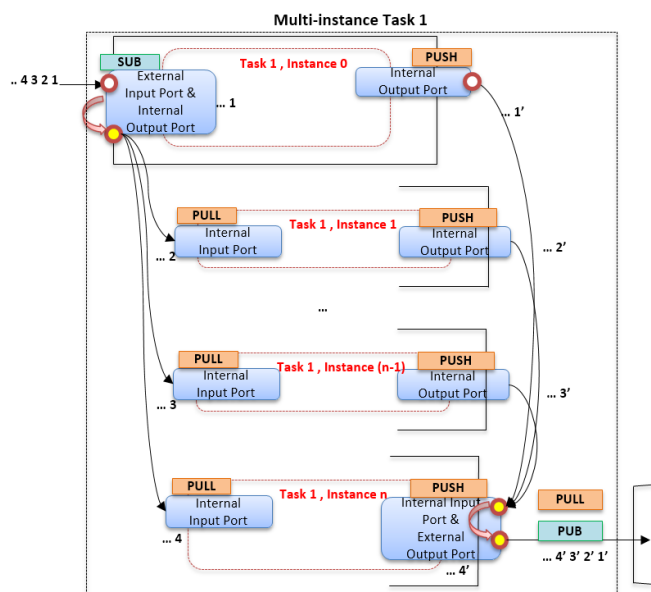


Figure 2. Round-robin Data Stream Model

At first, Figure 2 illustrates a round-robin transport example. Messages are round-robin to all connected downstream nodes. *Task 1, Instance 0* receives data stream sequences such as <1, 2, 3, 4>. It uses a pub / sub method to receive data sequences from external tasks. In detail, first data (i.e., 1) is processed by *Task 1, Instance 0* thread. The other three data (i.e., 2, 3, 4) are forwarded into *Task 1*'s other instances. After processing each stream, the result data emitted by the internal instances is sent as streams to a central instance. In this case, *Task 1, Instance n* conducts aggregation job, which combines internal result streams into appropriate result streams and then it will emit data to the next task.

Secondly, Figure 3 depicts a key-value fashion with three keys.

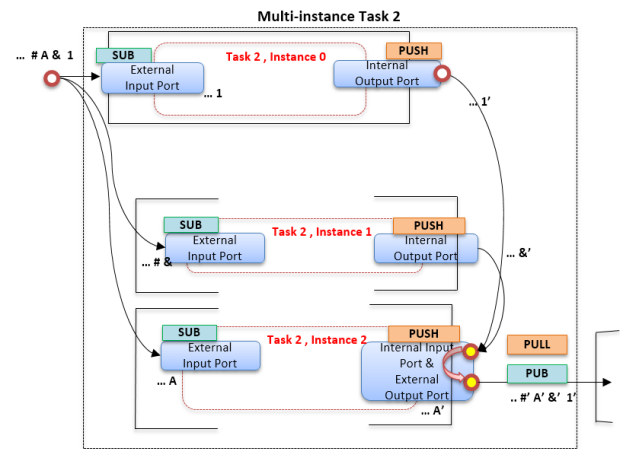


Figure 3. Key-value Data Stream Model

Messages in key-value fashion are distributed in a fan out fashion to all connected peers. Input stream (i.e., <1, &, A, #>) publishes all *Task 2*'s instances. They receive data stream via a filtering operation. The string of filtering operation is used by keys. For example, the receiver side of *Task 2, Instance 1* has a filtering operation such as *is\_special\_chars()* function. The sender side of each instance is similar to round-robin data stream model in Figure 2. The result data of each output port is forwarded to a central instance (i.e., *Task 2, Instance 2*) for aggregating data sequences.

## 3) Channel Synchronization Method

In general, a task can have one or more communication paths, which are connected from previous tasks. The communication path from a task, in this paper, is called by a port. If a task has two communication paths, it has two

input ports. Assume that one port is fast and the other port is slow. Two ports in the task lead to channel delay problem to synchronize data input. To speed up the communication between channels, we suggest two-channel synchronization methods. Especially, asynchronous channel can be invoked even when at least one of the data is arrived.

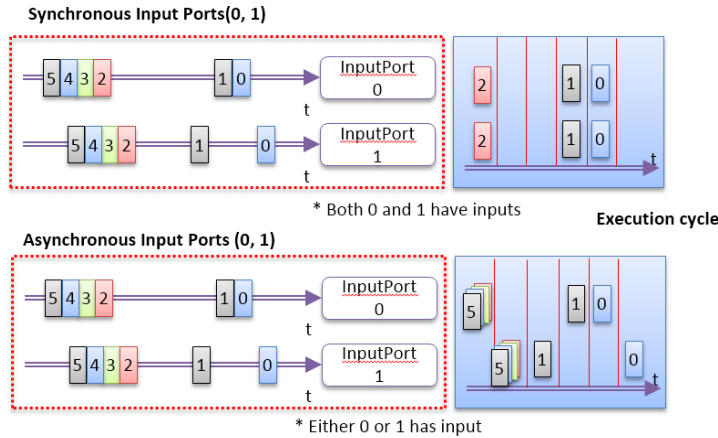


Figure 4. Two Channel Synchronization Methods

#### 4) Port Reusability

In general, a user in DSMS can submit many services, which conduct tasks associated with data streams. For example, a user develops a word count program that monitors the count of each word from an infinite stream of Twitter feeds. The program may consist of a stream's input task from Twitter, one or more split sentence tasks from the input task, a word count task, and a result output task. If the user would like to collect data streams from other data sources in addition to Twitter, he/she can make use of other services' tasks through port reusability. Figure 5 shows an example of port reusability between two services. *User 1, Service B* exports port 0 via a Uniform Resource Identifier (URI) format. The URI format is `task://InputTask#2_0`. This reusable design builds profit by eliminating the use of expensive data streams among tasks and reducing network costs.

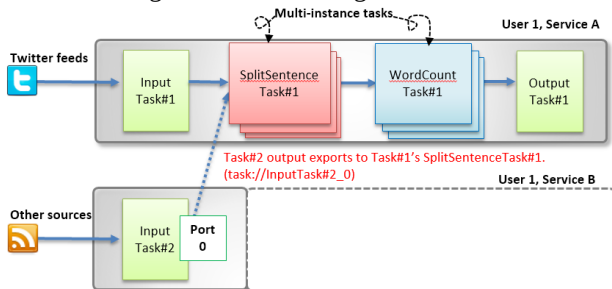


Figure 5. Port Reusability between Two Services

## B. Architecture

The stream processing subsystem architecture generally consists of an interface layer, a system layer, and a task layer. In an interface layer, we provide an Eclipse plugin for graphical DAG editing, command-line tools, and the set of stream processing APIs. The system layer conducts six kinds of management (i.e., user, metadata, service, service monitoring, node, and node monitoring) and provides four kinds of manager daemons (i.e., distributed scheduling, QoS, transport, and recovery manager). The task executor in the task layer spreads tasks among all available servers and coordinates task workflows associated with data streams. Three modules in the task layer are mainly useful to communicate between tasks and to process data streams efficiently. The stream I/O module provides functions such as round-robin data stream model and key-value data stream model. The channel communication module includes basic transport protocols (e.g., pub/sub, push/pull channels). The parallel deployment module is ongoing work with FPGAs to accelerate parallel stream processing.

## IV. PRELIMINARY RESULTS

### A. DAG Deployment

The Service Designer is a powerful Eclipse-based integrated development environment (IDE) which enables developers to design ease-of-use Directed Acyclic Graph (DAG) application [4]. This includes capabilities for source code generation, graphical DAG development and integration, XML editing and input, output, and task settings. Stream processing is based upon a record at a time. *InputTask* receives Twitter feed streams from Twitter website [4].

It can forward raw data to *Multi-instanceSplitTaskRR*, *Single-instanceTask*, and *Multi-instanceSplitTaskKV*. These three tasks conduct a split job with various ways. Split data is assigned to two kinds of merge task such as *Multi-instanceMergeTask* and *Single-instanceMergeTask*. The count of each word from the twitter example is collected by *OutputTask*.

Figure 6 represents an in-depth DAG graph.. Multi-instance tasks (i.e., *Multi-instanceSplitTaskRR* and *Multi-instanceSplitTaskKV*) are generated by our system. If the system has a burden of processing, then its job is much heavier. This will lead to expand the scope of parallel processing associated with two Multi-instance tasks.

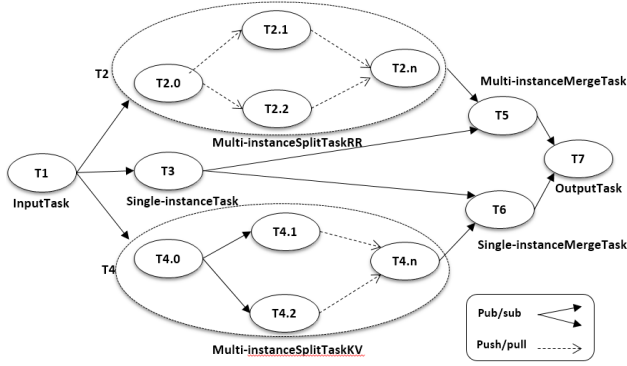


Figure 6. Sample DAG Graph

### B. Test Result

We implemented simple codes associated with channel communication. It includes multi-instance tasks and round-robin/key-value data stream model. Using these codes, we conducted the performance evaluation of sample DAG graph in Figure 6.

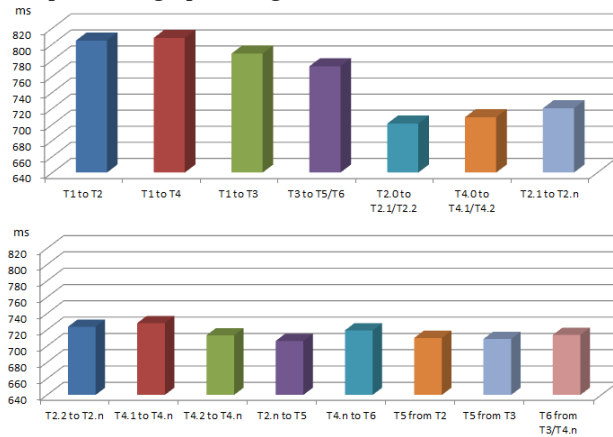


Figure 7. Elapsed Time of Sample DAG

Figure 7 shows an elapsed time of the sample DAG graph with six servers. The result graph describes our preliminary result to examine the data processing time. T1 to T2/T1 to T4/T1 to T3 are an initial source from InputTask T1. These three inputs are heavy jobs that import from external sources. Multi-instance jobs such as T2 and T4 have the smallest work load that their jobs partitioned sub-jobs in parallel. In particular, *Multi-instanceSplitTaskRR* and *Multi-instanceSplitTaskKV* have the nearly the same elapsed time. As a result, the Multi-instance tasks produce nearly the same channel throughput to the descendant tasks. It is the most important factor that multi-instance tasks have executed in parallel and have reduced the elapsed time evenly.

## V. CONCLUSION

In this paper, we have proposed the stream processing system associated with stream data processing in the big data platform. This is a prototype with basic communication module and task module. The proposed prototype provides multi-instance tasks, channel communication methods, and port reusability. We also explained a sample DAG graph and an Eclipse-based graphic designer for deployment and conducted performance evaluation. In experiments, the proposed data channel management provided parallel task execution from splitting jobs, and it is useful to process big data even when they are overloaded.

### Acknowledgements

This work was supported by the ICT R&D program of MSIP/IITP [R0126-15-1067, Development of Hierarchical Data Stream Analysis SW Technology for Improving the Realtime Reaction on a CoT (Cloud of Things) Environment].

## REFERENCES

- [1] [http://www.huffingtonpost.com/steve-hamby/the-big-data-nemesis-simp\\_b\\_1940169.html](http://www.huffingtonpost.com/steve-hamby/the-big-data-nemesis-simp_b_1940169.html)
- [2] <https://flink.apache.org>
- [3] <https://spark.apache.org>
- [4] <http://www.eclipse.org>
- [5] <https://dev.twitter.com/docs/streaming-apis>