Load Adaptive and Fault Tolerant Distributed Stream Processing System for Explosive Stream Data

Myungcheol Lee*, Miyoung Lee*, Sung Jin Hur*, Ikkyun Kim**

*Big Data SW Research Department, ETRI(Electronics and Telecommunications Research Institute), 218 Gajeong-ro, Yuseong-gu, Daejeon, 305-700, Republic of Korea

** Cyber Security System Research Department, ETRI(Electronics and Telecommunications Research Institute), 218 Gajeong-ro, Yuseong-gu, Daejeon, 305-700, Republic of Korea

{mclee, mylee, sjheo, ikkim21}@etri.re.kr

***Abstract*—As smart devices such as sensors, smartphones, and CCTVs are becoming extensively utilized recently, stream data from those smart devices are consistently generated explosively. There are also increasing cases that we notice security attacks after already important assets are damaged by cyber-targeted attacks such as APT attacks due to the lack of real-time security log processing capability. Accordingly, the demand to process and analyse the exploding stream data in real-time and in advance is consistently increasing in many application domains. However, existing distributed stream processing systems like Storm and S4 are not well adaptive when there are drastic increase of input stream data. In this paper, we propose a distributed stream processing system which supports several load adaptation techniques utilizable for various circumstances of explosive data stream, and also supports fault tolerance mechanisms to fail over in several failure situations.**

***Keyword*—Big Data, Distributed Stream Processing, Load Adaptation, Data Explosion, Load Shedding, Task Scheduling, Fault Tolerance**

I. INTRODUCTION

As smart devices such as sensors, smartphones, and CCTVs are becoming extensively utilized recently, stream data from those smart devices are consistently generated explosively and the need to process and analyse the exploding data stream in real-time is increasing [1]-[2].

There are also increasing cases such as eBay hacking(2008), Stuxnet(2010), Nonghyup(2011), and recent 3.20 Cyber Incident (2013.03.20) that we notice security attacks after already important assets are damaged by cyber targeted attacks like APT (Advanced Persistent Threat) attacks. These APT attacks are designed to steal industrial secrets or military secrets from major government agencies or enterprises and customer information, and paralyse the industrial control system and consequently cause tremendous physical damages, or wage an act of war [3].

According to Verizon data breach report [4] published in 2010, even though there were attack and breach logs left for 86% of breach cases, attacked organization's attack detection mechanism was not able to alert security warnings before the actual damage, due to the lack of real-time processing capability.

For providing real-time data processing and analysis to handle these explosive large-volume of stream data and predict future, or detect attack and damages, researches on real-time distributed stream processing systems such as Apache Storm [5] and Yahoo S4 [6] are actively being studied.

Existing distributed stream processing systems support fundamental real-time stream processing functionalities [7], but are not dynamically scalable enough because they are not adaptive when there are drastic increase of input stream data.

In this paper, we propose a distributed stream processing system which supports several load adaptation techniques utilizable for various circumstances of explosive data stream, and also supports fault tolerance mechanisms to fail over in several failure situations.

The remainder of this paper is organized as follows: Section 2 describes the system architecture and programming model of our proposed distributed stream processing system. Section 3 describes several load adaptation techniques for the various circumstances of explosive stream data, which were applied to our proposed distributed stream processing system. Section 4 describes several fault tolerance mechanisms applied to the proposed system. Finally, Section 5 presents the conclusion and future works.

II. DISTRIBUTED STREAM PROCESSING SYSTEM

A. System Architecture

Our proposed techniques were implemented in a

Manuscript received April 29, 2015. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP). (B0101-15-1293, Cyber-targeted attack recognition and trace-back technology based on the long-term historic analysis of multi-source data)

Myungcheol Lee is with the Electronics and Telecommunications Research Institute, Daejeon, 305-700, Korea (corresponding author to provide phone: +82-42-860-1691; fax: +82-42-860-6699; e-mail: mclee@etri.re.kr).

Miyoung Lee is with the Electronics and Telecommunications Research Institute, Daejeon, 305-700, Korea (e-mail: mylee@etri.re.kr).

Sung Jin Hur is with the Electronics and Telecommunications Research Institute, Daejeon, 305-700, Korea (e-mail: sjheo@etri.re.kr).

Ikkyun Kim is with the Electronics and Telecommunications Research Institute, Daejeon, 305-700, Korea (e-mail: ikkim21@etri.re.kr).

Copyright © 2016 GiRI (Global IT Research Institute)

distributed stream processing system depicted in Fig. 1, which consists of a service manager, several service manager candidates, several task execution managers, several task executors, a cluster coordinator, and a metadata storage.

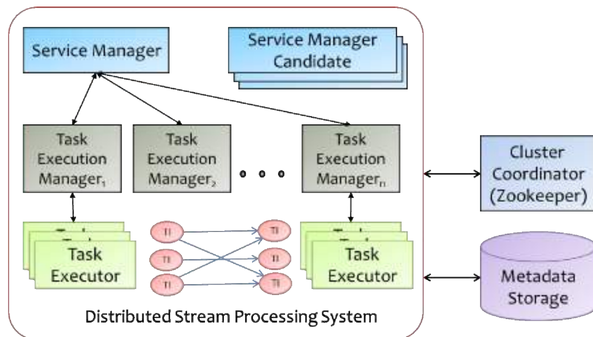

Fig. 1. Distributed Stream Processing System Architecture

Service manager manages entire cluster and schedules task instances of user-defined distributed stream processing service to distributed nodes for parallel execution. Task execution managers manage task executors on each node, and the task executors run each assigned task instances as separate threads in the same process space within the task executor. Task instances are cloned from user-defined task and they run on distributed nodes in parallel by sharing and partitioning the same input data stream.

Cluster coordinator is used for master election in case of master node's failure, and for shared storage and coordination of communication between several cluster components. Metadata storage is for the management of all the data related to the cluster, service, user, and QoS (Quality of Service) preferences.

B. *Programming Model*

The distributed stream processing system supports DAG(Directed Acyclic Graph) based distributed stream processing programming model to users as in Fig. 2, and the programming model contains input sources, task processing logics and output sources. LamaTask is a DAG node for representing task processing logic, and LamaInput and LamaOutput are DAG nodes for representing communication with external input and output sources.

Users write their distributed stream processing service according to the programming model, and the tasks containing processing logic belonging to the service are dispatched to distributed nodes and run continuously in parallel as task instances. The data communicated by each task instances are represented in key/value-based record stream.

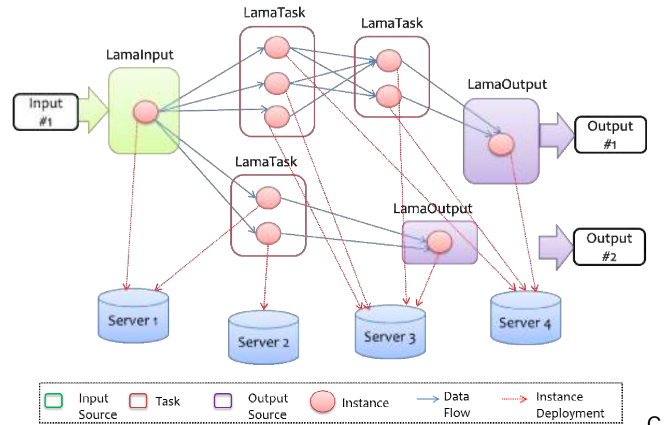

Fig. 2. Distributed Parallel Stream Processing Model

Common stream data model are defined as LamaRecord so that users can process various structured and unstructured stream data generated from diverse application environments. LamaRecord consists of key and value pair, where key is represented as String, and value is represented as POJO (Plain Old Java Object) which can be any primitive Java type or user-defined object type. Programming model components like input sources, tasks, output sources exchange data each other as a stream of LamaRecords as in Fig. 3.

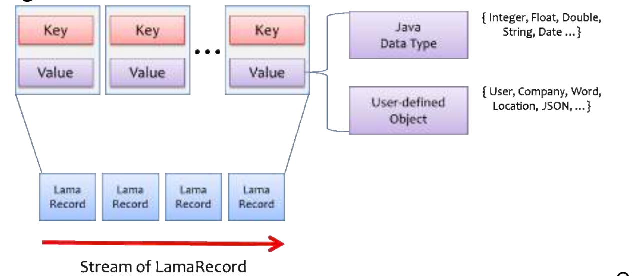

Fig. 3. Stream Data Model

We provide an abstract class called ILamaTask for users to implement their processing logic as task in Java language, and users define tasks by extending ILamaTask abstract class and implementing the processing logic inside the ILamaTask's execute() method just as defining the MergeTask class in Fig. 4.

```
ILamaTask
   config : Properties
   declarePorts(PortDeclarer) : void
   execute(InputCollector, OutputCollector) : void
   prepare() : void

        extends ILamaTask

public class MergeTask extends ILamaTask {
    static final Log LOG = LogFactory.getLog(MergeTask.class);
    public void execute(InputCollector inputs, OutputCollector outputs) {
}
```

Fig. 4. Implementation of User-defined Task

ILamaTask abstract class has other several methods supporting the implementation of user-defined tasks.

- void prepare(ServiceConf config) : define what needs to be prepared before initially running execute() method

Copyright © 2016 GiRI (Global IT Research Institute)

- void execute(InputCollector inputs, OutputCollector outputs) : user processing logic is implemented inside execute() method
- void declarePorts(PortDeclarer declarer) : define input port number, output port number, whether to synchronize all the input ports, etc.

C

```
// from any channel
List<LamaRecord> records = inputs.get();
for (LamaRecord record : records) {
    String key = record.getKey();
    Object value = record.getValue();
    // Process key/values here
    // …

    // Generate new key/values here
    String newKey  = …;
    Object newValue = …;
    // new LamaRecord is created inside emit()
    outputs.emit(newKey, newValue);
}
```

C

Fig. 5. Reading and Writing Key/Value Pairs

Fig. 5 depicts how users read and write the stream data inside execute() method. Users access input data by calling InputCollector.get(), LamaRecord.getKey(), and LamaRecord.getValue(), and send the processing results to next tasks by calling OutputCollector.emit().

Port is used for data transmission between input sources, task, and output sources as in Fig. 6. There is a channel for data communication established between preceding port (output port of any input source or task) and following port (input port of any output source or task), and the channels can communicate only one kind of data, or with the same schema (key, value).

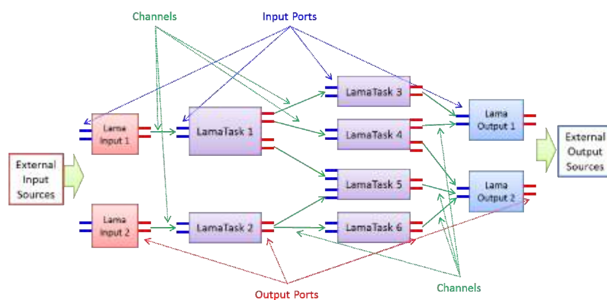

Fig. 6. Port-based Inter-Task Communication

Input source and output source are allowed to have exactly one input port and one output port, and tasks can have several input and output ports by user's definition, but the default number is one. For example, LamaTask 5 in Fig. 6 has 2 input ports and 1 output port, and input port 0 has 1 input channel from LamaTask 1's second output port and input port 1 has 1 input channel from LamaTask 2's first output port, and output port 0 has 1 output channel to LamaOutput 2's input port 0.

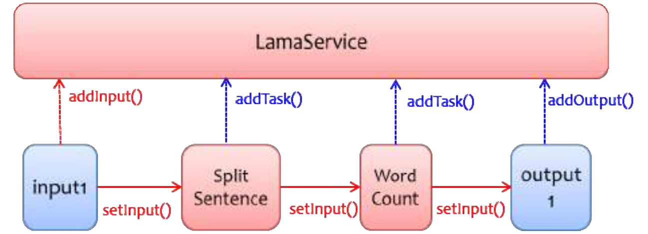

Fig. 7. Defining Distributed Stream Processing Service

Basic procedure for defining distributed stream processing service begins with creating LamaService object, and then create InputSource object and then register the created input source to the service object by calling LamaService.addInput() as in Fig. 7. All the subsequent tasks and output sources are registered to the service by calling previous tasks' LamaTask.setInput() method.

We support multiple input and output ports for tasks and multiple channels between ports, and users also define the specifics of input and output ports when they define tasks.

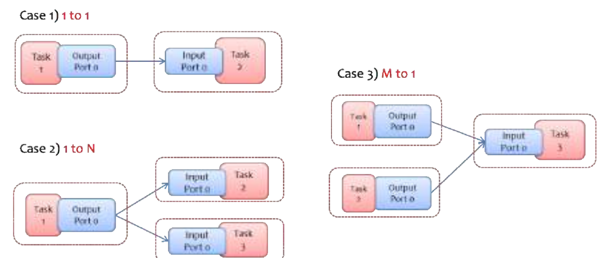

Fig. 8. Relationship between I/O Ports

Relationship between input and output ports are classified as in Fig. 8. 1 to 1 is for exchanging data directly between 1 input port and 1 output port, and 1 to N is used for broadcasting data from 1 input port to several output ports, and M to 1 is used for collecting output data generated from several tasks into a task and perform reduction operation.

Users can choose whether they would access synchronously or asynchronously data from multiple input ports as depicted in Fig. 9. In the synchronous mode, whenever data are ingested at *all* the input ports, user-defined ILamaTask.execute() method is called. Inside the execute() method, users can access data from each port by calling InputCollector.get(port) method with port index argument. In asynchronous mode, whenever data are ingested at *any* one of the input ports, user-defined ILamaTask.execute() method is called, and users can access the ingested data by calling InputCollector.get() method without port index argument.

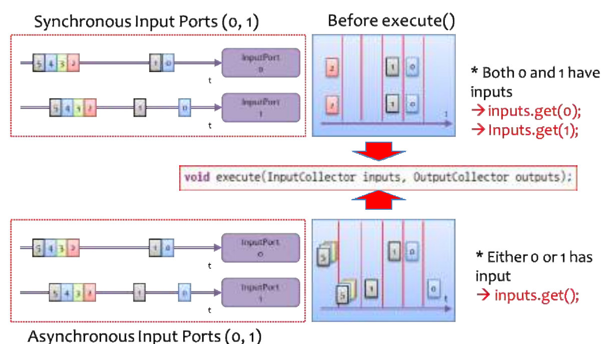

Fig. 9. Synchronous and Asynchronous Data Access from Multiple Input Ports

Copyright © 2016 GiRI (Global IT Research Institute)

The synchronized access mode between multiple input ports is predefined using PortDeclarer.syncInputPorts(boolean flag) method and the PortDeclarer is set to the task by overriding ILamaTask.declarePorts(PortDeclarer declarer) when extending ILamaTask.

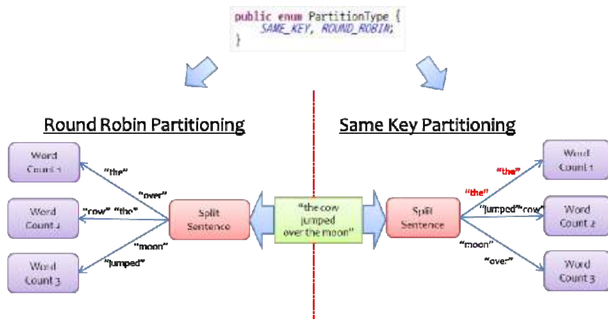

Fig. 10. Data Partitioning Scheme

Our proposed distributed stream processing system supports 2 kinds of partitioning scheme, which are used for partitioning data from several instances of previous task to several instances of next task. ROUND_ROBIN scheme is used for load balanced partitioning and SAME_KEY is used for sending the data with the same key to the same target instance, and which is useful for receiving the same classes of data, as in counting words shown in Fig. 10.

C. Dynamically Optimized Communication Channel

The distributed stream processing system supports optimized communication channels between task instances based on their execution location. If the two task instances, which communicate data each other, run on the same node, they run as separate threads in the same task executor process, and they exchange data using a common queue as the non-serialized data object itself. There are no serialization and deserialization overhead required for communicating the data in this case. On the other hand, if the two task instances, which communicate data each other, run on the different nodes, they run as separate threads in the different task executor processes on different nodes, and they exchange data using TCP-based sockets, and the data need to be serialized to array of bytes using Kryo [8] or Java's serialization mechanism [9] and after the transmission, the data need to be deserialized back to Java objects to be accessed by the consuming task instances.

In our experimental observation, in the extremely optimized distributed stream processing system execution, most of the overhead was generated by the object serialization and deserialization for TCP-based communication between nodes. Based on this observation, our proposed system tries to schedule task instances, which communicate data each other, on the same node as possible.

D. Stream Data Window

Our proposed system provides stream data windowing function so that users can process unbounded structured and unstructured stream data by limiting them as a sequence of bounded data batches. Time-based and count-based window types are supported, and both of the windows are defined by the window size and how much to slide the window as time elapses or as data are ingested. In a time-based window, window size and sliding size is defined in the millisecond unit, and in a count-based window, window size and sliding size is defined as the number of records.

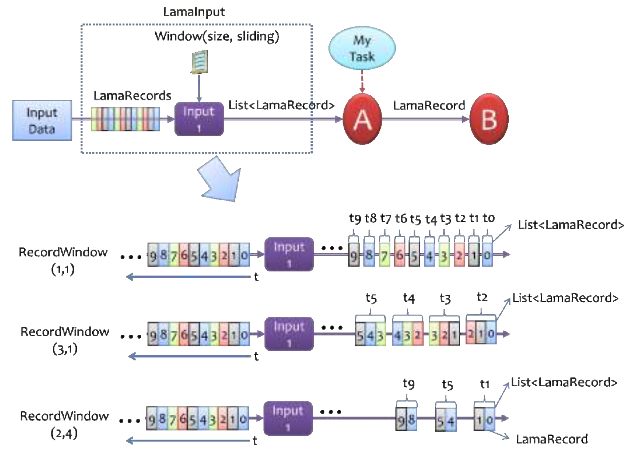

Fig. 11. Stream Data Window

Window is defined at the input source by calling e.g. LamaInput.setWindow(window_type, window_size, sliding_size) method and the input source slices the input data as defined by the window definition. Examples of more detailed window operations are depicted in Fig. 11.

III. Load Adaptation Techniques

All the load adaptation techniques applied to the proposed distributed stream processing system are summarized in Table I, and the detailed explanation for each technique follows in the next subsections.

TABLE I
LOAD ADAPTATION TECHNIQUES FOR DISTRIBUTED STREAM PROCESSING SYSTEMS

Purpose	Technique
Inter-node Load Balance	− Load-aware Instance Dispatch
Inter-task Load Balance	− Load-aware CPU Scheduling
Increasing Processing Capability	− Task Split and Merge − Task Migration − Adding More Nodes
Load Shedding	− Input Data Shedding

A. Load-aware Instance Dispatch

In a distributed stream processing system, user-defined tasks are split into several task instances for exploiting data parallelism and dispatched by global scheduler to different nodes for parallel execution. If we dispatch task instances on the node with the least load, we can shorten the processing time and the latency when users access the outputs.

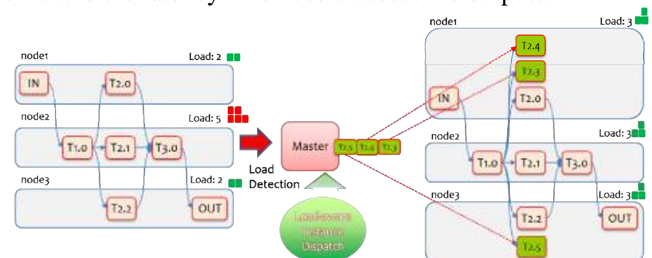

Fig. 12. Load-aware Instance Dispatch

Our global scheduler gathers load status information from

Copyright © 2016 GiRI (Global IT Research Institute)

each worker nodes and mark their status as “NORMAL” or “OVERLOADED”, and dispatch task instances to “NORMAL” nodes preferentially, for example node2 in Fig. 12 is the most loaded node, therefore the global scheduler dispatches T2.3 and T2.4 to node1, and T2.5 to node3, and no task instances to node2.

B. Load-aware CPU Scheduling

Task instances dispatched to each nodes are executed by task executors running on each nodes, and they run as separate threads inside the task executor processes and have their own input queue for receiving data from previous tasks or input sources.

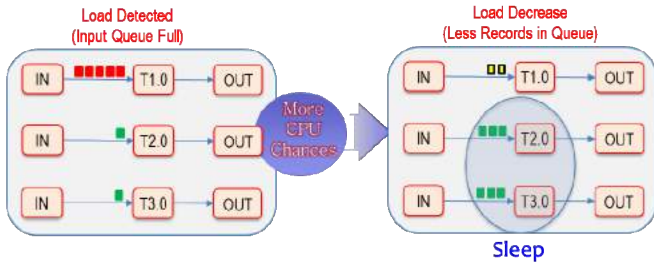

Fig. 13. Load-aware CPU Scheduling

Our local scheduler monitors each input queue’s load, or whether there are any waiting records. If there are any waiting records in the queue, it means that the task instance bound to the input queue is not processing fast enough for the input data rate. Local scheduler marks each instance as “NORMAL”, “YELLOW”, and “RED” according to the number of waiting records in their input queue, and try to make more CPU times assigned to the more loaded task instance by having the less loaded task instances to sleep for a while as in Fig. 13.

C. Task Split and Merge

If all the task instances, which belong to the same task, are overloaded in all the nodes, and there are still available resource in the cluster, global scheduler increases the data parallelism of the task by adding more task instances to the task. The added task instances can share the input data load and lower the burden of existing task instances as in Fig. 14.

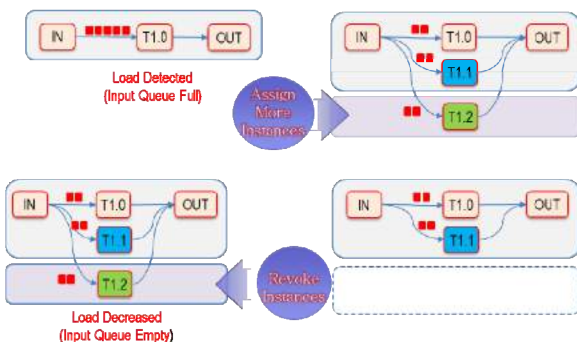

Fig. 14. Task Split and Merge

This task split technique can be applied to only the tasks that their data partitioning is ROUND_ROBIN, and the task instances before and after the newly added task instances are notified of the new task instances and their communication channels are re-established to exchange data with the new task instances.

D. Task Migration

If a node is fully utilized and the task instances running on the node cannot be split into more instances, then the global scheduler migrates the less loaded task instances to another idle nodes as in Fig. 15 so that they are not affected by other heavy task instances. The migrated task instances need to store their current input queue data and memory status into the persistent storage for the continuation of execution after the migration.

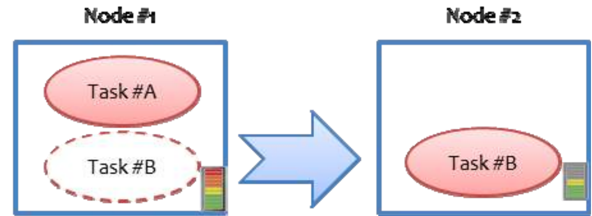

Fig. 15. Task Migration

E. Adding More Nodes

If load of all the nodes are close to their limit, system manager can add more nodes to the cluster so that entire cluster can handle more explosive data stream and have more distributed stream processing capability.

Global scheduler recognizes immediately the dynamically added nodes by system manager. By system manager’s option, the global scheduler performs immediate rebalancing among entire nodes or let the newly added nodes be utilized for later dispatch only.

When rebalancing is performed, service manager detects the most loaded node and migrates the task instances running on the node to the newly added node, and the inputs and outputs of the migrated task instances are all re-established according to the new service topology.

If service load becomes lower as time elapses, some of the nodes will become idle, and the idle nodes might be excluded from the cluster and returned back to free pool by system manager’s preference.

F. Load Shedding

In case there are no more resources available, global scheduler decides to drop data at the input source layer as in Fig. 16. Users can specify their satisfaction for service execution as QoS (Quality of Service) in our system, where four QoS items such as acceptable shedding rate, acceptable latency, whether to do recovery, and finally whether to preserve order between data are supported. Only the services for which user has specified shedding rate as more than 0 are chosen for load shedding.

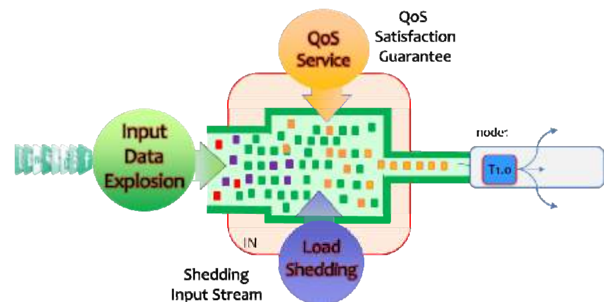

Fig. 16. Load Shedding

IV. Fault Tolerance

Our proposed distributed stream processing system

Copyright © 2016 GiRI (Global IT Research Institute)

supports 3 kinds of fault tolerance techniques (master failover, node failover, and instance failover) so that users' distributed stream processing service can seamlessly run even in the various failure situations.

A. Master Failover

In the normal cluster startup, service manager writes its address to ZooKeeper's master address node in ephemeral mode to notify service manager candidates and task execution managers that it is now the master. All the other components hold watcher on the ZooKeeper's master address node. If service manager dies, the address in the master address node disappears with the node itself because the node was ephemerally written. And the deletion of the master address node is notified to all the components which have left watchers for the node.

All the service manager candidates tries to write their address to the master address node, and one of them wins the race and becomes the new master, and the election of new service manager is notified to all the other components, and they try to communicate with the new service manager and report their status to the new service manager.

Eventually the new service manager gets to know the cluster status and the execution status of all the distributed stream processing service and task instances seamlessly.

B. Node Failover

If a node is in trouble, it is detected by the global scheduler, and all the task instances running on the failed node become the target of the new scheduling. The task instances are dispatched by the global scheduler to other normal nodes and the communication channels between the task instances and preceding/following task instances are reestablished using the instance channel information stored in ZooKeeper.

As a first step, the new task instances try to connect to their output targets using targets' channel information acquired from ZooKeeper, and they create new TCP ports for their input ports and store the new channel information to ZooKeeper, and in turn their preceding tasks need to update their output channels to the new task instances by getting channel information from ZooKeeper. Thus, only the instance channel information about receivers are stored in the ZooKeeper node in the name of "receiver_instance_id@input_port" with the value of "receiver_host_name:tcp_port".

All the task instances acquire the receivers' channel information from ZooKeeper and compare if both of the senders and receivers run on the same node by comparing the "receiver_host_name" with the host name of their own execution node. If they run on the same node, the sender establishes common queue based communication channel with the receiver. Otherwise, the sender establishes TCP socket channel to the receiver with the acquired "receiver_host_name:tcp_port" information.

Newly established instance channel information are again stored in ZooKeeper also for later failover activities.

C. Instance Failover

If a task instance fails suddenly, it is detected by global scheduler. And the global scheduler dispatches the failed task instance to another normal node, and the communication channels between new task instance and preceding/following tasks are reestablished as explained in the previous section. If the task instance continues to fail several times and the failure count reaches the predefined threshold, the task instance is marked as failure and excluded from rescheduling since then.

V. CONCLUSION

In this paper, we proposed a new distributed stream processing system which supports several load adaptation techniques to process explosive stream data in distributed stream processing environments and several fault tolerance techniques for resilient distributed stream processing in the various failure situations.

Our proposed system provides stream data model, and stream data programming model, multiport-based communication mechanism, several data partitioning schemes, and dynamic optimized communication channels between tasks. Our proposed load adaptation techniques are designed considering various conditions of nodes and tasks, and classified as inter-node load balancing, inter-task load balancing, increasing processing capability, and finally load shedding for the last way to avoid entire system halt due to the data explosion. Our proposed fault tolerance techniques are designed considering various failure situations like master, node, and instance failure.

We just implemented the load adaptation techniques and the fault tolerance techniques in the distributed stream processing system and are going to experiment the usefulness of them thoroughly in a real-world environment as future works. We hope our proposed techniques are useful as a hint for flexibly handling problems in many data processing domains experiencing input data explosion or fluctuation, and seamlessly providing distributed stream processing service in various failure situations.

ACKNOWLEDGMENT

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government(MSIP). (B0101-15-1293, Cyber-targeted attack recognition and trace-back technology based on the long-term historic analysis of multi-source data)

REFERENCES

[1] Mark A. Beyer, Anne Lapkin, Nocholas Gall, Donald Feinberg, Valentin T. Sribar, "Big Data Is Only the Beginning of Extreme Information Management," *Gartner*, 2011.

[2] John F. Gantz, "The Diverse and Exploding Digital Universe," *IDC*, 2008.

[3] Colin Tankard, "Advanced Persistent threats and how to monitor and deter them," *Network Security*, vol. 2011, no. 8, pp. 16-19, 2011.

[4] Verizon Inc., "2010 Data Breach Investigation Report," 2010, *http://www.verizonenterprise.com/resources/reports/rp_2010-data-breach-report_en_xg.pdf*

[5] Storm Project, Available at https://github.com/nathanmarz/storm/wiki/ (referenced at Sep. 1, 2014)

[6] Leonard Neumeyer, Bruce Robbins, Anish Nair, Anans Kesari, "S4: Distributed Stream Computing Platform," *Proc. of ICDMW 2010*, pp.170-177, Sydney, 2010

[7] Miyoung Lee, Wan Choi, "Trends of Big Data Processing Technology for Big Data Analytics," *Korea Information Processing Society Review*, vol.19, no.2, pp.20-28, 2012 (in Korean)

[8] Kryo Project, Available at http://code.google.com/p/kryo/ (referenced at Sep. 1, 2014)

[9] Java Object Serialization Spec, Available at http://docs.oracle.com/javase/8/docs/platform/serialization/spec/serial-arch.html (reference at Sep. 1, 2014)

Copyright © 2016 GiRI (Global IT Research Institute)

Myungcheol Lee (M'2015) received his Bachelor's Degree in Computer Engineering and his Master's Degree in Computer Engineering from Chungnam National University, Daejeon, Korea in 1999 and 2001, respectively. He became a Member (M) of IEEE in 2015. He is now a senior researcher at ETRI since 2001. His research interest includes Big Data processing and analytics, database, cloud computing, and distributed computing.

C

Miyoung Lee received her Bachelor's Degree in Food and Nutrition, and Master's Degree in Computer Science and Statistics from Seoul National University, Seoul, Korea in 1981 and 1983, respectively. She received her Doctor's Degree in Computer Engineering from Chungnam National University, Daejeon, Korea in 2005. She is now a principal researcher at ETRI since 1988. Her research interest includes big data management and processing, database, and distributed computing.

Sung Jin Hur received his Bachelor's Degree in Electronics, Master's Degree and Doctor's Degree in Computer Engineering from Kyungpook National University, Daegu, Korea in 1990, 1992 and 1999, respectively. He was a professor at Changsin University from 1999 to 2001, and is now a principal researcher at ETRI since 2001, and is leading Data Management Research Section. His research interest includes database, stream data processing, cloud computing

C

Ikkyun Kim received his Bachelor's Degree, Master's Degree and Doctor's Degree in Computer Engineering from Kyungpook National University, Daegu, Korea in 1994, 1996, and 2009, respectively. He was a visiting researcher at Purdue University from 2004 to 2005. He is now a principal researcher at ETRI since 1996, and is leading Network Security Research Section. His research interest includes network security, computer network, cloud security, big data analytics.

Copyright © 2016 GiRI (Global IT Research Institute)