Elastic Stream Computing with Clouds

Atsushi Ishii\textsuperscript{1} and Toyotaro Suzumura\textsuperscript{1,2}
\textsuperscript{1}Tokyo Institute of Technology
\textsuperscript{2}IBM Research - Tokyo
Executive Summary

Data Stream Processing

- Many Varieties of Sensors
- Processing in real-time fashion
- Real-time Application

Streaming Digital Data
Data Stream Management System
Real-time Response

Problem Statement

Data Stream Processing Application
Burst of Data Rate
New nodes

Current Data Stream Processing Systems cannot dynamically assign or remove computational nodes

Our Approach

Optimization Problem

We present a method and an architecture to use virtual machines (VMs) in the cloud environment and to use optimization problem in an elastic fashion to stay ahead of the real-time processing requirements.

ElasticStream System

Experimental Results

Keeping the Application’s response latency low
Minimizing the economic cost for cloud environment
Agenda

1. Introduction
2. Our Approach
3. Implementation
4. Experiment
5. Related Work
6. Conclusion
Agenda

1. Introduction
2. Our Approach
3. Implementation
4. Experiment
5. Related Work
6. Conclusion
Stream Computing

- A new computing paradigm for processing streaming data in a real-time fashion.

- Data Stream Management System:
  - System S (IBM), S4 (Yahoo), Borealis (MIT)…
Stream Computing

- Application examples:
  - Latency-critical anomaly detection
  - Financial data analysis
  - Analyzing data from large scale sensor networks
System S and SPADE

System S can scale to large numbers of compute nodes

**SPADE Program**

**SPADE Compiler**

Execution files
Script Files
Configuration Files

*running on the commodity cluster such as Linux*

*Optimization Scheduler automates resource management*

**Processing Element Container**
**Processing Element Container**
**Processing Element Container**
**Processing Element Container**

System S Data Fabric
Transport
Operating System

X86 Blade
X86 Blade
X86 Blade
X86 Blade
X86 Blade
Current stream computing systems do not provide the feature that enables to add new nodes dynamically in run time even if the incoming data rate becomes bursty.
Problem Statement

- Recent real-time application needs low latency in the responses to the stream data.
- Bursts of data rate can change the latency:
  - To handle all the burst of data, it is needed to add new computational nodes dynamically.
- Other problems by adding new physical nodes:
  - Budget limitations, inadequate electrical supply, or even space for hardware...
Our Approach – Elastic Stream Computing with Clouds

- We present a method and an architecture that provides elastic stream computing platform with Clouds
  - adding new resources within a few minutes
  - need not consider where the new resources are located
  - dealing with situations where the data rate suddenly bursts by temporarily adding new VM (Virtual Machine)s
Agenda

1. Introduction
2. **Our Approach**
3. Implementation
4. Experiment
5. Related Work
6. Conclusion
The Definition of the Cloud

- Only an IaaS
  (Infrastructure as a Service)

- Examples:
  - Amazon EC2
  - Eucalyptus
Our Proposed System: ElasticStream
Overview of the ElasticStream (contd.)

- Application flow in the system can be divided into three parts:
  - Receiving the incoming data
  - Splitting the data up for multiple nodes
  - Processing the data in parallel

- The system also adds cloud VMs if the local environment is overloaded
Adding new cloud VMs

- The ElasticStream system calculates the required number of VMs, and then elastically add new virtual machines on the Cloud.

Diagram showing the process of adding new cloud VMs.
How can we solve the trade-off issue between latency and financial costs?

- Pricing system is “pay-as-you-go”
  - computation time, data transfer, usage of storage, etc…
  - (Show sample Amazon price here …)

- The trade-off between latency and costs exists
  - Too many VMs will increase the total costs
  - method to minimize the latency and total costs is needed
Optimizing the financial cost of using the Cloud environment

- We need to calculate the least number of VMs to keep latency low

- In this research, we formulate the trade-off between the latency and the costs into an optimization problem
Our Proposed Scheduling Policy

- We use the term *TimeSlot* for an interval for
  - solving the optimization problem
  - manipulating the cloud VMs

- To calculate the required number of VMs, we need to predict the future data rate for the next *TimeSlot*.

- An example of the algorithm for prediction:
  - SDAR Algorithm
    (Sequentially Discounting Auto Regression Model)
Target Application Types

- **Data Parallel Application**
  - distributes a data stream
  - computes in parallel
  - Most of the applications belong to this type
  - **This research focuses on this type**
    - e.g. Real-time mining for Twitter streams

- **Task Parallel Application**
  - distributes a computation process
  - duplicate input stream
    - e.g. Computation-intensive SST (Singular Spectrum Transformation) algorithm
Formulation

Objective Function

- Minimizing the cost for the cloud environment
- The solution is the numbers of the VMs for each instance types

Constraint:

- When the future data rate is larger than the amount of data that local nodes can handle,
- The rest of the data must be assigned to the Cloud VMs

\[ \text{Min:} \]
\[ Cost = \sum_{\text{type}} (P_{\text{type}} + P_{\text{Nin}} \times D_{\text{type}}) \times x_{\text{type}} \quad ... (1) \]

Where:
\[ \forall x_{\text{type}} \geq 0, \quad \forall x_{\text{type}} \in N, \]
\[ \sum_{\text{type}} (D_{\text{type}} \times x_{\text{type}}) \geq (D_{\text{next}} - D_{\text{local}}) \quad ... (2) \]

Sum of the data which can be uploaded to Cloud

The amount of the data which is needed to be uploaded to Cloud

- \( P_{\text{type}} \): price for running a VM
- \( P_{\text{Nin}} \): price for 1-GB data upload
- \( D_{\text{type}} \): data stream assigned each instance type
- \( x_{\text{type}} \): # of the VMs for each instance type
- \( D_{\text{next}} \): future data rate
- \( D_{\text{local}} \): data stream which local nodes can handle
Compared with ad-hoc scheduling policies

- When the data rate bursts, the system could add several nodes with several ad-hoc policies
  - Our optimization problem approach can obtain the cost-optimal numbers of VMs directly, and also support multiple instance types

- Optimization problem approach could be extended for other requirements:
  - e.g. Region for running VMs
  - Multiple Cloud providers
Agenda

1. Introduction
2. Our Approach
3. Implementation
4. Experiment
5. Related Work
6. Conclusion
About System S (again)

- Large-scale, distributed stream computing platform developed by IBM Research

- Describe the data-flow graphs by its special stream-application language called **SPADE** [Gedik, SIGMOD, 2008]*

- SPADE allows users to create **customized operations** written in C/C++ or Java
  - The *ElasticStream* system uses C++ UDOPs

SPADE: The Language for the Stream Application

• A stream-centric and operator-based language for stream processing application for System S
  • Also supports all of the basic stream-relational operators with rich windowing semantics

• System S treats operator as one processing unit
• Input/Output data of the operator is called “Tuple”
• System S describes the data flow graphs using operators

```scala
[Program]
vstream MySchema(symbol : String, tradedate : String, closingprice : Double, volume : Integer)
vstream aggregatedData(symbol : String, avgPrice : Double)

stream myODBCstream(schemaFor(MySchema))

stream StockMovingAverage (schemaFor(aggregatedData))
  := Aggregate(myODBCstream <count(20), count(1), pergroup>)
  [symbol]
  {Any(symbol), Avg(closingprice)}

Nil := DbAppend(StockMovingAverage)[connection:"DB2Person";
  access:"StockSink"]{}
```
Elastic Stream Processing on System S

- The ElasticStream system is built on top of System S and constructed with data flow graphs written in SPADE.
- We implemented C/C++ based UDOPs (User-Defined Operators) to extend System S to enable System S “Cloud-ready”.
- In current System S, restarting the job is required for adding nodes dynamically:
  - some data will be lost
  - Implemented the feature which enable to add/remove nodes in runtime as operators.
System Processing Flow

- **Application’s processing**
  1. Splits the incoming data up for each computational nodes
  2. Each nodes compute in parallel
  3. Aggregates the results and outputs them

- **Manipulating the cloud**
  1. Predicts data rate for the next *TimeSlot*
  2. Calculates the # of VMs
  3. Adds/Removes VMs on the cloud environment
Components for the application’s process

- **StreamManager**
  - Splits the data stream
  - Manages the TCP connection

- **LatencyAggregator**
  - Aggregates the latency result
  - Output a log

- **Computational Component on the Cloud**
  - The computational component of the prototype system is currently written in Ruby
Components for manipulating the cloud

- **FutureDetection**
  - Predicts data rates for next *TimeSlot*

- **Optimizer**
  - Calculates the numbers of the VMs for each instance types for next *TimeSlot*

- **VM Manager**
  - Communicates Amazon EC2
  - Manages VMs’ start/stop
Agenda

1. Introduction
2. Our Approach
3. Implementation
4. Experiment
5. Related Work
6. Conclusion
Performance Evaluation

Local Environment

- CPU AMD Phenom 9850 Quad-Core Processor 2.5GHz, Memory 8GB *1
  (Computational node)
- CPU AMD Phenom 9350e Quad-Core Processor 2GHz, Memory 4GB *1
  (For ElasticStream System)
- Software
  CentOS 5.4 kernel 2.6.18-128 AMD64, gcc version 4.1.2, Ruby 1.9.1

Cloud Environment

- Amazon Linux AMI Beta 2010.11.1
  - Small instance ($0.095/h)
  - Medium instance ($0.19/h)
- Region: US-West Latency: about 100ms (From Tokyo Tech)

1Gbps Network
Application for the experiment

- Regular expression matching application for a data stream like Twitter
  - Each tuples in the stream is 1KB
    - Data rate changes from 200KB/s to 2000KB/s
  - Outputs the data to the local nodes only when the matching process succeeds
Compare the static patterns

- **Static pattern**
  - **Local**: only use the local machine
  - **Static**: use some VMs with local machine
    - (VM: Small*1 + Medium*2)

- **Dynamic pattern**
  - **ElasticStream**: Our approach

- We used a component that provides a precise input data rate instead of using the future detection algorithm
  - This is intended for measuring the best performance, but this will be replaced with more sophisticated change point detection algorithms such as SDAR
Result 1 (1/3)

- *ElasticStream* system kept the latency low using cloud VMs

![Graph showing data rate, latency, and number of VMs over elapsed time]
Result 1 (2/3)

Unexpected bursts (within a sec.) are caused because the data distribution is stopped for a short while when new VM is added on the cloud (This issue will be solved for future )
Result 1 (3/3)

This is because the system used an average data rate value. To handle such burst, we could use maximum data rate value.
Result 2

- ElasticStream system was able to reduce the total current cost by 80%

Amazon EC2 charge cost every hour
This is a simulation score in the case of being charged every 5 minutes.

Reduced the total current cost by 80%, against the “Static” pattern
Discussion

- The reduction ratio of total costs
  - $T_{\text{All}}$: Total running time of the application
  - $T_{\text{Burst}}$: Total time when the data rate bursted
  - The reduction ratio of running costs is $T_{\text{Burst}} / T_{\text{All}}$
    - Only if the data transfer costs (or etc.) can be ignored

- The system cannot handle the burst whose interval is less than $\text{TimeSlot}$
  - One possible solution would be to shorten the $\text{TimeSlot}$ interval

- Making $\text{TimeSlot}$ too short may bring the additional overhead of the VM boot time
  - We could solve this issue by calculating optimal $\text{TimeSlot}$ interval by experiments, or allowing one to prepare extra VMs in advance
Agenda

1. Introduction
2. Our Approach
3. Implementation
4. Experiment
5. Related Work
6. Conclusion
Related work (1/2)

- Using cloud environment for batch processing [Bossche, Cloud, 2010]
  - They run a scheduling algorithm as a preprocessing step

- We scheduled and updated the combination of VMs periodically
  - we focus on data stream processing that needs to handle continuously arriving and potentially infinite data streams
Related work (2/2)

- Load balancing in the data stream management system
  - Load balancing by Load Shedding
    [Mozafari, ICDE, 2010]

- Elastic scaling of terminating threads in a operator
  [Schneider, IPDPS, 2009]

- Job scheduling
  - Focused on the locality of the input data and “fairness” of the jobs users submitted
    [Zaharia, EuroSys, 2010]
Agenda

1. Introduction
2. Our Approach
3. Implementation
4. Experiment
5. Related Work
6. **Conclusion**
Summary and Future Work

Summary

- Presented the *ElasticStream* system
- Presented optimization problem for cost-optimal usage for cloud environment
- Implemented a feature to assign or remove computational resources dynamically
- Evaluated these features using Amazon EC2

Future work

- To improve component that predicts future data rate
- To implement the proposed elastic features into a data stream management system itself