A GRID-BASED MIDDLEWARE FOR PROCESSING DISTRIBUTED DATA STREAMS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Liang Chen, M.S.

* * * * *

The Ohio State University

2006

Dissertation Committee:
Gagan Agrawal, Adviser
Hakan Ferhatosmanoglu
Han-Wei Shen

Approved by

Adviser
Graduate Program in Computer Science and Engineering
ABSTRACT

Increasingly, a number of applications rely on, or can potentially benefit from, analysis and monitoring of data streams. Moreover, many of these applications involve high volume data streams and require real-time and distributed processing of data arising from a distributed set of sources.

We believe that a grid environment is well suited for flexible and adaptive analysis of these streams. Thus, our research focuses on designing and evaluating a middleware to support the processing of these streams. Our system is referred to as GATES (Grid-based AdapTive Execution on Streams). It flexibly achieves the best accuracy that is possible while maintaining the real-time constraint on the analysis. We have developed self-adaptation algorithms for this purpose.

Further, we have addressed the problem of resource allocation in the GATES system. We design a static resource allocation algorithm to generate sub-optimal allocations and the experimental evaluation shows these allocations are very close to the optimal one. We also develop an infrastructure to support resource monitoring and dynamic resource allocation. Moreover, we implement efficient and low-cost dynamic migration for pipelined applications in a grid environment by using the notion of Light-weight Summary Structure (LSS).
We have studied various data stream applications with support of GATES to intensively evaluate our middleware. The experimental results have demonstrated the expected advantages of utilizing GATES.
This is dedicated to my parents who raise me up, give me a wonderful life, teach me diciplines, promote my confidence, encourage me to explore the world by myself, and offer me their full supports.

This thesis is also dedicated to my wife who shines loves on my life, inspirits and helps me to overcome challenges, brings happiness to me, and shares every moment with me.
ACKNOWLEDGMENTS

The person who deserves the most credit for this dissertation is my advisor Dr. Gagan Agrawal. He proposed the initial idea of implementing GATES and furnished lots of significant advices for the directions of my research. Dr. Han-Wei Shen provided the C++ source codes for the rendering application and helped revise the chapter 7.
VITA

1975 ........................................Born Jiangxi, China

1996 ........................................B.S. Computer Science, China

1999 ........................................M.S. Computer Science, China

1999-2001 .................................. Engineer Bell Labs, China

2001-present ................................. Graduate Research Associate,
                                        Ohio State University.

PUBLICATIONS

Liang Chen and G. Agrawal “Supporting Self-Adaptation in Streaming Data Mining

Q. Zhu, Liang Chen and G. Agrawal “Supporting a Real-Time Distributed Intrusion

Liang Chen and G. Agrawal “A Static Resource Allocation Framework for Grid-
Based Streaming Applications”. Concurrency Computation: Practice and Experience
Journal, Volume 18, Issue 6, Pages 653-666.

Liang Chen and G. Agrawal “Resource Allocation in a Middleware for Streaming

Liang Chen, K. Reddy and G. Agrawal “GATES: A Grid-Based Middleware for Pro-
cessing Distributed Data Streams”. The Thirteenth IEEE International Symposium
FIELDS OF STUDY

Major Field: Computer Science
TABLE OF CONTENTS

Abstract ........................................................................... ii
Dedication ........................................................................... iv
Acknowledgments ............................................................... v
Vita .................................................................................. vi
List of Tables ........................................................................ xi
List of Figures ....................................................................... xii

Chapters:

1. Introduction .................................................................... 1
   1.1 Motivation ................................................................. 1
   1.2 Motivating Applications .............................................. 3
   1.3 Specific Contributions ................................................ 6
   1.4 Outline ......................................................................... 9

2. Related Work .................................................................... 10
   2.1 Stream Data Processing .............................................. 10
      2.1.1 Middlewares for Data Stream Processing ............ 10
      2.1.2 Overview of Related Work Done by Database Community . 11
      2.1.3 In-Network Aggregation in Sensor networks and Stream-Based Overlay Networks ................................................... 16
   2.2 Adaptation Through a Middleware ................................ 18
   2.3 Grid Resource Allocation for Streaming Applications .... 19
   2.4 Dynamic Migration in Grid Environment .................... 20
   2.5 Adaptive Visualization ............................................... 22
3. System Overview and Initial Self-Adaptation Algorithm . . . . . . . 23
   3.1 Middleware Design and Application Programming Interface . . . 23
      3.1.1 Key Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
      3.1.2 System Architecture and Design . . . . . . . . . . . . . . . . 26
      3.1.3 Self-Adaptation API . . . . . . . . . . . . . . . . . . . . . . . . . 27
   3.2 Self Adaptation Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 28
      3.2.1 Algorithm Overview . . . . . . . . . . . . . . . . . . . . . . . 29
      3.2.2 Detailed Description . . . . . . . . . . . . . . . . . . . . . . . . 31
   3.3 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 34
      3.3.1 Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
      3.3.2 Benefits of Distributed Processing . . . . . . . . . . . . . . . . 35
      3.3.3 Impact of Self Adaptation . . . . . . . . . . . . . . . . . . . . . 37
      3.3.4 Self-Adaptation For Processing Constraint . . . . . . . . . . . 38
      3.3.5 Self-Adaptation for a Network Constraint . . . . . . . . . . . 39
   3.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

4. Improved Self-Adaptation Algorithm . . . . . . . . . . . . . . . . . . . 42
   4.1 Self-Adaptation Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 43
      4.1.1 Algorithm Overview . . . . . . . . . . . . . . . . . . . . . . . . 43
      4.1.2 New Algorithm and Parameter Adjustment . . . . . . . . . . . 45
   4.2 Streaming Data Mining Applications . . . . . . . . . . . . . . . . . . 50
   4.3 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 54
      4.3.1 Convergence In a Static Environment . . . . . . . . . . . . . . 59
      4.3.2 Adaptation in a Dynamic Environment . . . . . . . . . . . . . . 63
      4.3.3 Comparing Different Algorithms . . . . . . . . . . . . . . . . . 66
   4.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

5. Resource Allocation Schemes . . . . . . . . . . . . . . . . . . . . . . . . 72
   5.1 Static Resource Allocation Algorithm . . . . . . . . . . . . . . . . . . 72
      5.1.1 Problem Definition and Complexity . . . . . . . . . . . . . . . . . . . 73
      5.1.2 Algorithm Description . . . . . . . . . . . . . . . . . . . . . . . . . 75
      5.1.3 Additional Optimization . . . . . . . . . . . . . . . . . . . . . . . . . 78
   5.2 Dynamic Resource Allocation . . . . . . . . . . . . . . . . . . . . . . . 78
      5.2.1 Architecture for Dynamic Resource Allocation . . . . . . . . . 78
      5.2.2 Dynamic Resource Allocation Scheme . . . . . . . . . . . . . . . 83
   5.3 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 85
      5.3.1 Evaluating Static Allocation . . . . . . . . . . . . . . . . . . . . . . . . 86
      5.3.2 Evaluation of the Dynamic Allocation Scheme . . . . . . . . . . 91

ix
6. Dynamic Migration ......................................................... 95
   6.1 Implementing Dynamic Migration using LSS .................. 97
      6.1.1 Motivation ................................................. 97
      6.1.2 Light-Weight Summary Structure ......................... 98
      6.1.3 Middleware Implementation .............................. 100
   6.2 Streaming Applications for Dynamic Migration ............. 104
   6.3 Experimental Evaluation ........................................ 106
      6.3.1 Experiment Setup and Data Sets ......................... 107
      6.3.2 Memory Usage with LSS .................................. 108
      6.3.3 Efficient Migration ...................................... 109
      6.3.4 Benefits of Migration in a Dynamic Environment .... 114
      6.3.5 Processing Accuracy and LSS Migration ............... 115
   6.4 Summary ........................................................ 117

7. An Adaptive Image Rendering Application Implemented Using GATES 119
   7.1 Overall Design .................................................. 121
      7.1.1 Two-Step Rendering Algorithm ......................... 121
      7.1.2 Implementing the Two-Step Rendering Algorithm on GATES 123
   7.2 Experimental Evaluation ........................................ 126
      7.2.1 Convergence under Different Data Arrival Rates .... 128
      7.2.2 Convergence Irrespective of Different Initial Values 130
      7.2.3 Performance Comparison of Two Implementations in C++
           and Java .................................................. 132
   7.3 Summary ........................................................ 136

8. Conclusion and Future Work ........................................ 137
   8.1 Contributions .................................................. 137
   8.2 Future Work .................................................... 138
      8.2.1 Improving Dynamic Resource Allocation Scheme .... 139
      8.2.2 Porting GATES from GT3.0 to GT4.0 ................... 139
      8.2.3 Supporting Fault-Tolerance and High Availability ... 142
      8.2.4 Further Relieving Programming Burdens from Application
           Developers ............................................... 143
      8.2.5 Supporting Distributed Continuous Queries .......... 144

Bibliography .......................................................... 146
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Summary of Symbols Used</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Benefits of Centralized Processing: 4 Sub-streams</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>Execution Time of Different Versions</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Accuracy of Different Versions</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Summary of Symbols Used</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Summary of Load States</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>Migration and Execution Time for Clustream</td>
<td>109</td>
</tr>
<tr>
<td>6.2</td>
<td>Migration and Execution Time for Dist-Freq-Counting</td>
<td>109</td>
</tr>
<tr>
<td>7.1</td>
<td>Execution Time Breaking Down When Image Size is 256 × 256 and Error Tolerance is 0.0</td>
<td>135</td>
</tr>
<tr>
<td>7.2</td>
<td>Execution Time Breaking Down When Image Size is 128 × 128 and Error Tolerance is 0.0</td>
<td>135</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Overall System Architecture</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Example to Show Usages of GATES API Functions</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>An application that comprises three stages</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>A Queuing model of the system</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>Self-Adaptation with Different Processing Requirements</td>
<td>39</td>
</tr>
<tr>
<td>3.6</td>
<td>Self-Adaptation with Different Data Generation Rates</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Self Adaptation Algorithm</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Communication Topology for the CluStream Application</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Communication Topology for the Dist-Freq-Counting Application</td>
<td>51</td>
</tr>
<tr>
<td>4.4</td>
<td>Convergence under Different Data Arrival Rates for Clustream, When Number of Micro-Clusters is Initialized to 20 and 40</td>
<td>55</td>
</tr>
<tr>
<td>4.5</td>
<td>Convergence under Different Data Arrival Rates for Clustream, When Number of Micro-Clusters is Initialized to 80 and 110</td>
<td>56</td>
</tr>
<tr>
<td>4.6</td>
<td>Convergence under Different Data Arrival Rates for Dist-Freq-Counting, When Error Tolerance is Initialized to 0.001 and 0.002</td>
<td>57</td>
</tr>
<tr>
<td>4.7</td>
<td>Convergence under Different Data Arrival Rates for Dist-Freq-Counting, When Error Tolerance is Initialized to 0.004 and 0.0045</td>
<td>58</td>
</tr>
</tbody>
</table>
7.3 Pseudo Codes for the Third Stage of the Application . . . . . . . . . . 125
7.4 Java Version Invokes the Native Libraries via JNI . . . . . . . . . . . 127
7.5 Error Tolerance Converging Under Different Data Arrival Rates . . . 128
7.6 Images Rendered, Given Different Error Tolerances . . . . . . . . . . 129
7.7 Image Size Converging Under Different Data Arrival Rates . . . . . . 130
7.8 Error Tolerance Converging to a Same Value Irrespective of Different Initial values (Data arrival Rate 100Kbps) . . . . . . . . . . . . . . 131
7.9 Error Tolerance Converging to a Same Value Irrespective of Different Initial values (Data arrival Rate 200Kbps) . . . . . . . . . . . . . . 131
7.10 Image Size Converging to a Same Value Irrespective of Different Initial Values (Data Arrival Rate 100Kbps) . . . . . . . . . . . . . . 132
7.11 Image Size Converging to a Same Value Irrespective of Different Initial Values (Data Arrival Rate 200Kbps) . . . . . . . . . . . . . . 133
7.12 Comparing Different Implementations: Image Size is 256 × 256 . . . . 133
7.13 Comparing Different Implementations: Image Size is 128 × 128 . . . . 134
1.1 Motivation

The emergence of grids is providing an unprecedented opportunity to solve problems involving very large datasets. However, the existing work in this area has so far focused on static datasets resident in data repositories [38]. Increasingly, a number of applications across computer sciences and other science and engineering disciplines rely on, or can potentially benefit from, analysis and monitoring of data streams.

In the stream model of processing, data arrives continuously and needs to be processed in real-time, i.e., the processing rate must match the arrival rate. There are two trends contributing to the emergence of this model. First, scientific simulations and increasing numbers of high precision data collection instruments (e.g. sensors attached to satellites and medical imaging modalities) are generating data continuously, and at a high rate. The second is the rapid improvements in the technologies for Wide Area Networking (WAN), as evidenced, for example, by the National Light Rail (NLR) proposal and the interconnectivity between the TeraGrid and Extensible Terascale Facility (ETF) sites. As a result, often the data can be transmitted faster than it can be stored or accessed from disks within a cluster.
Many stream-based applications share a common set of characteristics, which makes grid-based and adaptive processing desirable or even a necessity. These characteristics are:

- The data in these streams arrives continuously, 24 hours a day and 7 days a week.
- The volume of data is enormous, typically tens or hundreds of gigabytes a day. Moreover, analyzing this data to gain useful knowledge requires large computations.
- Often, this data arrives at a distributed set of locations. Because of the volume of data, it is not feasible to communicate all data to a single source for analysis. These locations can be across multiple administrative domains and may only be connected over a WAN.
- It is often not feasible to store all data for processing at a later time. Also, it is important to react to any abnormal trends or change in parameters quickly. Thus, the analysis needs to be done in real-time or near real-time.

Realizing the challenges posed by the applications that require real-time analysis of data streams, a number of computer science research communities have initiated efforts. In the theoretical computer science or data mining algorithms research area, work has been done on developing new data analysis or data mining algorithms that require only a single pass on the entire data [58]. At the same time, database systems community has been developing architectures and query processing systems targeting continuous data streams [11]. Recently, a workshop was held as part of FCRC in San Diego, targeting different aspects of data stream processing\(^1\).

\(^{1}\)Please see http://www.research.att.com/conf/mpds2003
However, the existing efforts in this area have generally focused on data streams from a single source. Many real applications involve data streams from a distributed set of sources. We view the problem of flexible and adaptive processing of distributed data streams as a grid computing problem. We believe that a distributed and networked collection of computing resources can be used for analysis or processing of these data streams. Computing resources close to the source of a data stream can be used for initial processing of the data stream, thereby reducing the volume of data that needs to be communicated. Other computing resources can be used for more expensive and/or centralized processing of data from all sources. However, how to allocate Grid resources to such distributed processing poses big challenge. Further, because of the real-time requirements, there is a need for adapting the processing in such a distributed environment, and achieving the best accuracy of the results within the real-time constraint. In recent years, there has been much interest on adaptive or *autonomic* computing. Adapting applications or programs has been studied by many, and a variety of solutions have been proposed, including those through new algorithms [79], runtime/middleware [60, 25, 9, 117, 89, 99], and language/compilers [49, 46, 48]. However, no existing grid middleware supports such adaptation. It will be desirable if such functionality can be supported in a grid middleware, and does not need to be hard-coded for a specific application.

1.2 Motivating Applications

This section describes a number of applications and application classes that can benefit from grid-based and adaptive processing of data streams.

**Processing of Data from Scientific Instruments:** Many recent computational science or grid computing projects involve large volumes of data continuously collected
through scientific instruments or experiments. We will consider two specific examples here.

A large hadron collider (LHC) being setup in CERN (located at Geneva) is expected to generate hundreds of petabytes of data per year by the year 2007, and exabytes by the year 2012 [20]. It is planned that this data will be distributed to around 10 Tier 1 centers, and then onto around 50 Tier 2 centers. As the data is continuous or streaming in nature, and because of the very high volume of data, the initial analysis or filtering of data will need to be done at real-time. The storage capacities will require that the data is filtered by a factor of $10^6$ to $10^7$. Thus, it is important that crucial information is extracted by real-time analysis on continuous streams.

Another example we consider is the Earthscope project [21]. The goal of this project is to combine geophysical measurements from several sources and enable enhanced analysis. The total seismic data collected will be around 40 TB per year. The data from different sources will need to be amalgamated and analyzed in real-time. Such real-time analysis could enable prediction of ground motion from large earthquakes.

**Computational Steering:** A computational steering system allows the user to interactively control scientific simulations, while the computation is in progress. Examples of computational parameters that could be modified at runtime include the boundary conditions, model geometries, or resolutions at different parts of the grid [92]. Computational steering is typically done by analyzing the data generated at various time-steps. For example, if we detect certain features at a part of a grid, we may want to increase the resolution for that part of the grid.
This, however, poses two important challenges. First, it is important to analyze the data in a short time, so that simulation parameters can be modified quickly. This can be difficult if the volume of data is large. Second, analysis of data can require significant computational resources, which may not be available at the parallel platform being used for simulation.

With increasing WAN bandwidths, it is possible to do such online data analysis using grid resources. Also, there is usually some flexibility in the analysis of data, i.e., the data generated could be sampled and then analyzed. Clearly, the smaller the fraction of data that is analyzed, the greater is the risk of inaccuracy. Thus, we will like to analyze the largest fraction, as long as the analysis could be done in a timely fashion. Unfortunately, no middleware is currently available to support analysis with a time-constraint in a grid environment.

Similar issues also apply in the emerging class of Dynamic Data-driven Applications and Systems (DDAS). In this class of applications, simulation parameters are adjusted by real observed data. It is even more likely that data collection is at a different location than where simulation is executed. Still, the volume of collected data and need for adjusting simulation parameters in a timely fashion poses a challenge.

**Computer Vision Based Surveillance:** Multiple cameras shooting images from different *perspectives* can capture more information about a scene or a set of scenes. This can enable tracking of people and monitoring of critical infrastructure [18]. A recent report indicated that real-time analysis of the capture of more than three digital cameras is not possible on current desktops, as the typical analysis requires large
computations. Distributed and grid-based processing can enable such analysis, especially when the cameras are physically distributed and/or high bandwidth networking is available.

**Online Network Intrusion Detection:** Detecting network intrusions is a critical step for cyber-security. Online analysis of streams of connection request logs and identifying unusual patterns is considered useful for network intrusion detection [47]. To be really effective, it is desirable that this analysis be performed in a distributed fashion, and connection request logs at a number of sites be analyzed. Similar to the applications we described earlier, large volumes of data and the need for real-time response make such analysis challenging.

### 1.3 Specific Contributions

We believe that Grid environment is well suited for flexible and adaptive analysis of distributed streams. This thesis focuses on design and evaluation of a middleware that is referred to as GATES (Grid-based AdapTive Execution on Streams) [33]. The thesis involves four specific contributions.

**Utilizing Existing Grid Standards** First, we have designed the GATES middleware to use the existing grid standards and tools to the extent possible. Specifically, GATES is built on the Open Grid Services Architecture (OGSA) model and uses the initial version of Globus Toolkits (GT) 3.0’s API functions. It offers high-level interfaces that allows users to specify algorithm(s) and steps for processing data streams. GATES then can automatically create services for the steps specified by users, and deploy and pipeline the services.
Providing Self-Adaptation Functionality  The second contribution of this thesis is supporting GATES system to provide self-adaptation functionality to its applications. The system flexibly achieves the best accuracy that is possible while maintaining the real-time constraint on the analysis. To do this, the system monitors the arrival rate at each source, the available computing resources and memory, and the available network bandwidth, and automatically adjusts the accuracy of the analysis. This is done by changing values of adaptation parameter(s), e.g., changing the sampling rate, the convergence rate and size of the summary structure maintained, so as to determine a trade-off between accuracy and processing rate. While a user is required to expose the parameters that can be modified, choosing their values to meet the real-time constraint is done automatically by the system. We have developed algorithms for this purpose and carefully evaluated them. The evaluation results show our adaptation algorithm is able to quickly converge to proper and stable values of adaptation parameter in a static environment, and also adapt the processing rapidly in a dynamic environment.

Supporting Automatic Resource Allocation  Third, we have developed GATES as an autonomous system in terms of resource discovery and allocation. Though resource discovery and resource allocation have been active topics in grid community, the pipelined processing and real-time constraint required by distributed streaming applications pose new challenges. Our work is distinct in the following aspects: 1) We present a static resource allocation algorithm that can automatically generate mapping schema according to availabilities of Grid resources 2) We have developed a framework to support resource monitoring and dynamic resource allocation, and 3)
We also designed a dynamic resource allocation algorithm that supports automatic and seamless migrations of pipelined stages in varying environments. Our experimental results demonstrate that the allocations automatically generated by the static resource allocation algorithm are very close to the optimal one, and the overheads of using dynamic allocation are quite small.

**Supporting Efficient Dynamic Migration** Fourth, we enable GATES to support efficient dynamic migration for tightly-coupled and pipelined applications in a grid environment. We provide an alternative to basic checkpointing, using the notion of Light-weight Summary Structure (LSS), to enable efficient runtime migration. The idea behind LSS is that at certain points during the execution of a processing stage, the state of the program can be summarized by a small amount of memory. This allows us to perform low-cost process migration, as long as such memory can be identified by an application developer. We have observed from our experiments that the use of LSS introduces a very small overhead, and the dynamic migration can significantly improve the performance of long-running applications while not impact the accuracy of the processing.

We have studied various data stream applications with support of GATES to intensively evaluate our middleware. In particular, we have implemented an image rendering application on GATES. Such application renders images from volumes in a continuous stream, and with support of GATES, it can automatically adjust image size or quality to adapt to a varying execution environment. The evaluation of this application has proven the effectiveness of the self-adaptation functionality and also shown that GATES itself does not incur too many overheads.
1.4 Outline

The rest of this dissertation is organized as follows. We first compare our work with related efforts in Chapter 2. In Chapter 3, we present the overview of the GATES system and introduce the first version of our adaptation algorithm. An improved adaptation algorithm is explained in Chapter 4. Moreover, we carefully evaluate the GATES middleware in that chapter. In Chapter 5, we focus on the system’s static and dynamic resource allocation schemes. We then consider the problem of supporting dynamic migration for pipelined applications in a grid environment in Chapter 6. In Chapter 7, we study an adaptive visualization application with support of GATES. Finally, we conclude our work and propose directions for future research in Chapter 8.
CHAPTER 2

RELATED WORK

In this chapter, we first compare our work with other efforts on supports for data stream processing. Then, we focus on related work on adaptation through a middleware. Other associated work on grid resource allocation and dynamic migration for streaming applications are presented in 2.4 and 2.3, respectively. Section 2.5 explains related work on adaptive visualization.

2.1 Stream Data Processing

We give a survey of the work on processing (distributed) streaming data. In 2.1.1, we discuss some middleware systems that can support distributed computing. We then consider related work done by database community in 2.1.2. In Section 2.1.3, we give an overview of related work on stream-based overlay networks and in-network aggregation in sensor networks. We compare our work with the related work as summarizing it.

2.1.1 Middlewares for Data Stream Processing

Our work derives from the DataCutter project at the Ohio State University [14, 13]. Our API for specifying a pipeline of processing units is quite similar to what
DataCutter supports. However, our work is also distinct in many ways. First, we support adaptation to meet the real-time constraint. Second, our system is built on top of OGSA. Third, we enable easy deployment in a distributed environment, through an application container.

Stampede is a cluster middleware for supporting streaming applications [101, 102]. Our work is again distinct in consider grid resources and adaptation for real-time processing. Mazzucco et al. have looked at the specific support for merging multiple high speed data streams [83].

2.1.2 Overview of Related Work Done by Database Community

Applications that deal with continuous, unbounded streams of data are drawing increasing attentions from the database community. The community believes it is an intuitive view to use database operations to process streaming data, specifically by means of database query languages [96]. Hence, the research in this area generally concentrates on how to support continuous queries over streaming data.

Continuous Query Systems Early efforts in this direction has been focusing on designing new query languages, query operators and engines that manage data streams. Prominent work includes STREAM [10], Telegraphcq [27], Aurora [22], NiagraCQ [31], dQUOB [95, 97] and so on. We give an overview of the work as follows and highlight their techniques.

STREAM is a data stream management system that supports continuous queries over streams and traditional data sets. It defines CQL, a continuous query language
that implements abstract semantics. It also addresses performance issues and solves
them by using operator scheduling, load shedding, and adaptive infrastructure.

**TelegraphCQ** focuses on adaptive continuous queries over high-volume and highly
variable data streams. Meanwhile, it provides load balancing and fault tolerance fea-
tures.

**NiagraCQ** can be used to query and analyze XML streaming data. They have
done interesting work in rate-based optimization, and used techniques of sharing and
grouping operators to make queries more scalable.

**Aurora**’s focus is on the real-time data processing issues, such as QoS- and
memory-aware operator scheduling, semantic load shedding for coping with transient
spikes in incoming data rates. Note that query operations are connected in the DAG
fashion. Data streams’ schemas can be specified by users and use-defined queries
can be compiled to pre-defined operators that can be subsequently scheduled by the
system.

The major differences between the above work and ours are as follows.

First, the above work can be categorized to *continuous query systems* that process
data via various query operators. Our work supports general data stream processing
and GATES is similar to workflow systems that pipeline applications and process
streams stage by stage.

Second, these systems have largely been focusing on *centralized processing* of a
single data stream, or bringing distributed streaming data to a central site to process.
Instead, we push computation to remote computing nodes close to data sources, and
process the smaller number of intermediate data in a central site.
Finally, the above work only supports standard or pre-defined query operators, while our work can plug user-specified codes into the system to process streams.

**Distributed Continuous Query System** Recently, research attentions in continuous query systems have shifted to querying data streams in distributed environments [28, 37].

Aurora* is a framework for distributed processing of data streams within a single administrative domain [36]. But it does not support adaptative processing.

Medusa [127] is a distributed continuous query system using Aurora as the single-site processing engine. Medusa can distribute Aurora’s queries to multiple nodes. Unlike Aurora*, Medusa provides an infrastructure to manage federated operations of nodes distributed over multiple administrative domains [37]. In order to achieve adaptive load management, Medusa employs an agoric model that utilizes economic principles to regulate collaboration of organizations. In contrast, we exploit the queue theory and design an adaptation algorithm to adjust processing rate to match data arrival rate.

Developed at Brandeis/Brown/MIT, Borealis [3] combines the core functionality of processing streams from Aurora with the distribution functionality from Medusa. The work is motivated by the needs of modifying queries on the fly. It designs an model to optimize various QoS metrics in distributed environment. Our work is distinct in many ways. For instance, we process data streams in Grid environment and support self-adaptation for applications to meet real-time constraint of processing.

Extending an existing object relational DMBS engine to work in a Grid environment, the Grid Stream Database Manager (GSDM) [107, 68, 73] enables continuous
query processing in Grid. Calder [80], the successor of dQUOB, provides query grid
services that can be used to access other continuous query processing systems. Our
work is Grid-based but it is distinct from them in the following ways. First, stream
processing is done in a pipeline of stages. Second, our system addresses issues of Grid
resource scheduling. Finally, we support adaptation in distributed environments.

Adaptive Continuous Queries over Streaming Data Besides the DB-systems
work for distributed stream processing, there have been specific techniques studied in
DB community, such as load shedding and adaptive filtering. They have been used
to implement adaptive continuous queries over streaming data. Here, we summarize
related work that use these techniques.

Load Shedding: Most continuous query systems employ the load shedding tech-
nique, i.e., dropping some portion of unprocessed data, to enable systems to provide
continuous and up-to-date query responses [12]. The goals of such load shedding
are making processing rate matching data arrival rate while minimizing inaccuracy of
query results. Most load shedding algorithms, e.g. those in STREAM project [10] and
Aurora [22], are based on sampling. Tatbul et al [116] propose a shedding algorithm
that is based on sampling and also driven by QoS specification.

Instead of using load shedding techniques, GATES has an adaptation algorithm
that is targeting the same goals. But our adaptation algorithm does not drop any
data. Instead, according to system loads, the algorithm only suggests new values of an
adaptation parameter specified by an application. The application itself is responsible
for applying these new parameter values to data processing. Adaptation parameters
could be a sampling rate, error tolerances, number of clusters and so on. Only when the parameter is a sampling rate, our algorithm is similar to load shedding.

**Adaptive Filtering:** In the context of querying distributed data streams, filters are installed at remote data sources to reduce the volume of data transmitted to a central node. Olston et al [90] first propose the idea of using adaptive filters which adjust filter bounds to reduce communication costs when user-specified error tolerances are met. Cheng et al [34] apply the similar idea but they use non-value-based tolerance for continuous queries. Our work on adaptivity is distinct from theirs in the following aspects. First, they are solving specific problems. Our adaptation algorithm can support various data stream applications, because our algorithm is employing the queue theory and machine learning techniques to statistically predict what system loads will be in the near future and accordingly adjust processing rates. Second, they require users to specify error tolerances, while error tolerances can be one type of adaptation parameters our GATES is able to adjust. In sum, adaptation functionality provided by GATES is more generic.

Jain et al [69] recently use the Kalman Filter as a general and adaptive filtering solution for conserving resources. They use the filter to predict system’s inter-states and adaptively conserve communication costs. The method of this work is similar to ours, but it does not work well with non-linear systems and systems where statistic properties of noise are unknown. Our algorithm can give good adaptation supports to these systems.

In general, we find that GATES uses different approaches to deal with same issues. For example, GATES designs an adaptation algorithm to meet stream processing’s real-time constraint, while the related work usually employs load shedding, sliding
windows, operator dynamic scheduling and routing and so on to adaptively process streaming data. Both GATES and the distributed systems in database community can deploy computing units to remote sites and distribute tasks to these units. On the other hand, computing units in GATES are Grid services while theirs are query operators. Both of us have load monitoring and have designed resource scheduling (load distribution) mechanisms. However, GATES focuses on Grid resource allocation, and their work is usually applied to tightly coupled distributed environments.

2.1.3 In-Network Aggregation in Sensor networks and Stream-Based Overlay Networks

In-Network Aggregation in Sensor Network: In a large sensor network, aggregation queries are important [86]. Energy saving and communication reduction are active research topics in this area. Recent work [125, 81, 65] has focused on reducing size of transmitted data by performing in-network data aggregation in sensor networks. The main idea is combining partial results at intermediate nodes which results in saving communication cost and reducing energy consumed [86, 45, 124]. [45] exploits the tree hierarchy to evaluate aggregate queries in error-tolerant applications over sensor networks. [59] proposes a hierarchy topology. [86] argues that a tree and a hierarchy topology are not robust and proposes synopsis diffusion, i.e., combining multi-path routing schemes with techniques avoiding double-counting.

Similarly, GATES uses a user-specified tree to aggregate data from distributed data sources. We assume that processing in earlier stages of an data stream application can significantly reduce size of streaming data. Therefore, earlier stages are normally deployed closer to data sources. On the other hand, our work is more genetic. Applications can define their own layout trees, and it is not required for earlier
stages to reduce data volume. Another difference is our work is Grid-based while they usually aggregate data in homogeneous networks.

**Stream-Based Overlay Networks:** Overlay networks have been investigated by the network community for a while. Currently, how to exploit overlay networks to support streaming data processing is a new topic. A stream-based overlay network is a layer between a data stream processing system and the physical network. Such an overlay network determines placements of query operators for distributed data-stream processing systems [93]. One of the prominent work in this area is Hourglass [113] developed by Pietzuch et al. They propose a scheme to dynamically adapt to varying network conditions. By doing this, query operators can be partitioned efficiently. Moreover, they design an algorithm to optimize network-aware paths for stream-based overlay networks [93, 94].

Similarly, our work also considers how to map application stages to Grid resources to achieve good performance. Further, we design resource allocate algorithms to determine sub-optimal paths from data sources to the destination and adjust these paths dynamically when resource conditions are varying. However, our work is distinct in the following aspects. First, adaptivity, e.g., dynamically altering paths, has not been investigated by their work. Second, our work is based on Grid to provide supports for data stream processing. We take advantage of various well-provided Grid utilities to monitor and allocate resources, while they design their own registry, monitoring and allocation mechanisms.
2.2 Adaptation Through a Middleware

Application adaptation has been studied in many contexts, including through (grid) middleware. We briefly survey this work here and state how our work is distinct. As a quick summary, our work is different in focusing on adapting to meet real-time constraint on stream data processing, and in adapting the output of the application to do so.

Cheng et al. have developed an adaptation framework [35]. Adve et al. [6] have focused on language and compiler support for adapting applications to resource availability in a distributed environment. Our work is different in having runtime support for self-adaptation to meet real-time constraint.

A number of projects have focused on operating systems, middleware, and networking support for adapting applications to meet quality of service goals [15, 70, 75, 88, 91]. SWiFT is a software feedback toolkit to support program adaptation [114]. However, it does not adapt the computation or support the notion of adaptation parameters. Conductor is a middleware for deploying adaptation agents into a network [126]. It does not adapt the computation and is not specifically designed for meeting real-time constraints on processing.

DART [105] is a system facilitating quick development of adaptive applications. A runtime component is responsible for making adaptation decisions following a set of selected policies. Moura et al. present software support in the component-base programming context for construction of auto-adaptive applications [43]. It leaves applications the option to dynamically choose the most beneficial components. ROAM implements resource-aware runtime adaptation for device heterogeneity in mobile systems [60]. Schwan and his group take into account the runtime resource management
issues when supporting adaptive applications [99]. Similar ideas have been considered by Karamcheti and co-workers [29, 67]. Particularly, the notion of tunable parameters has similarity to our work, though their focus is not on streaming data. Isert and Schwan have developed a system called ACDS, which includes a monitoring and steering tool for adapting stream based computations [66]. These systems require that either the resource usage associated with each option be stated explicitly or the logic for making adaptation decisions be specified by the application developer. In comparison, we consider a more restrictive class of applications, but automate the adaptation process more.

2.3 Grid Resource Allocation for Streaming Applications

Resource allocation has been an important topic in the grid community. Most of the initial work has been on static matching of the resource requirements and the available resources [50, 30, 123, 64, 104, 128, 118]. However, none of these efforts considered pipelined or streaming applications. It should be noted that our approach does not require resource requirements to be explicitly stated by an application, in contrast to Condor’s matchmaking [103] or the Aurora system [22].

Much work has been done on resource discovery [71, 85, 39, 119], often using mobile agents or objects to do efficient search. [19] presents a market-based CPU allocation policy, i.e., agents bid for computational priority from hosts, and proves that the bidding strategy eventually results in a computable unique Nash Equilibrium. [112] proposes to clone agents to deal with the situation where agents cannot carry out their tasks because of insufficient resources. According to their scheme, agents may
clone, pass tasks to others, migrate, die or merge. Our focus is on resource allocations for pipelined applications, while mobile agents are usually independent of others.

Realtor [40] is a protocol for supporting survivability and information assurance by migrating components to safe locations under circumstances of external attack, malfunction, or lack of resources. Our work is distinct in considering resource degradation and application adaptation. Isert and Schwan have developed a system called ACDS, which includes a monitoring and steering tool for adapting stream based computations [66], including assigning alternative resources. In comparison, we consider a more restrictive class of applications, but automate the dynamic resource allocation process more.

Our work has some similarities with the grid-based (dynamic) workflow projects, including the SDSC Matrix project\(^2\), work by Abramson \textit{et al.} at Monash University [5], and by Deelman \textit{et al.} at ISI [44]. Our work is distinct in considering streaming data with real-time constraint on the processing.

Many researchers have proposed techniques for real-time resource allocation, including, for example [110]. The problem we consider is different because we focus on pipelined processing of streaming data and wide-area distributed environments.

\subsection{2.4 Dynamic Migration in Grid Environment}

In this section, we compare our work with existing work on checkpointing process migration. Checkpointing and process migration has been widely studied in distributed systems. Here, we restrict ourselves to the work done within parallel and grid computing.

\(^2\)http://www.npaci.edu/DICE/SRB/matrix/
Condor [118, 84] supports transparent migration of a process (through checkpointing) from one workstation to another. Our work is distinct in using LSS to make the checkpoints more efficient, and in focusing on streaming or pipelined applications. Krishnan and Gannon have focused on checkpointing for distributed components, in the context of XCAT3 [74]. They create a consistent global snapshot across multiple processes. We only support checkpointing and migration for a single processing stage, but make it more efficient using LSS. Vadhiyar and Dongarra have developed SRS, which is a system for developing malleable and migratable distributed applications [120].

Several researchers have addressed the problem of using an adaptive environment for executing parallel programs. In the context of Charm++, support for processor virtualization has been implemented using migratable objects [61]. Stellner developed a system called CoCheck [115], which performs process migration for MPI programs. Most of the earlier work considered a task parallel model or a master-slave model. In a version of PVM called Migratable PVM (MPVM) [26], a process or a task running on a machine can be migrated to other machines or processors. User Level Processes (ULP) [108] provides light-weight user-level tasks, which can be migrated from one machine to another. Piranha [55] was a system developed on top of Linda [16]. In this system, the application programmer has to write functions for adapting to a change in the number of available processors. Data Parallel C and its compilation system [87] have been designed for load balancing on a network of heterogeneous machines.
2.5 Adaptive Visualization

In this section, we compare our adaptive volume rendering supported by GATES with related work on adaptive visualization algorithms. Many adaptive visualization algorithms exist for polygonal or volume rendering, and can be classified into three categories: static heuristics, inter-frame feedback, and global optimization. Algorithms based on static heuristics[106, 2, 1] determine the quality of rendering using fixed criteria such as distance, view angle, or screen coverage. Whereas, the rendering quality in our volume rendering application is decided by the GATES middleware. Feedback control algorithms[76, 77, 109] adjust the level of detail (LOD) according to the difference between the desired and the actual rendering time from the previous frame. Global optimization algorithms [53, 57] rely on having rendering performance models and object benefit heuristics, where the objective of optimization is to maximize the image benefit while constraining the rendering cost according to the user-specified rendering budget. In each of the above cases, the adaptation technique is explicitly coded for each application. Our rendering application exploits and invokes GATES API functions which can provide similar functionality.
CHAPTER 3

SYSTEM OVERVIEW AND INITIAL SELF-ADAPTATION ALGORITHM

This chapter gives an overview and an initial evaluation of GATES for processing of distributed data streams. We first describe the middleware design and its architecture in 3.1. An initial self-adaptation algorithm is introduced in the 3.2, followed by the system evaluation presented in 3.3.

3.1 Middleware Design and Application Programming Interface

This section describes the major design aspects of our GATES system.

3.1.1 Key Goals

There are four main goals behind the design of the system.

1. Use the existing grid infrastructure to the extent possible. Particularly, our system builds on top of the Open Grid Services Architecture (OGSA) [52], and uses its reference implementation, GT3.0. The Globus support allows the system to do automatic resource discovery and matching between the resources and the requirements.

2. Support distributed processing of one or more data streams, by facilitating applications that comprise a set of stages. For analyzing more than one data
Figure 3.1: Overall System Architecture
stream, at least two stages are required. Each stage accepts data from one or more input streams and outputs zero or more streams. The first stage is applied near sources of individual streams, and the second stage is used for computing the final results. However, based upon the number and types of streams and the available resources, more than two steps could also be required. All intermediate stages take one or more intermediate streams as input and produce one or more output streams. GATES’s APIs are designed to facilitate specification of such stages.

3. Enable the application to achieve the best accuracy, while maintaining the real-time constraint. In the stream model of processing, data arrives continuously and needs to be processed in real-time, i.e., the processing rate must match the arrival rate. In view of this, an important goal of the GATES system is to allow the most accurate analysis, while still meeting the real-time constraint. To enable this, application developers can expose one or more adaptation parameters, along with a range of their acceptable values. A higher (or lower) value of the adaptation parameter results in more accurate but slower processing. Thus, the goal of the system is to determine the highest (or the lowest) value of the parameter which can still meet the real-time constraint. Moreover, the environment for processing the data streams can be dynamic, i.e., data arrival rates, and/or the available network bandwidth or CPU cycles can vary over time. In such cases, the system should be able to adjust adaptation parameters dynamically. Such functionality is achieved through a self-adaptation algorithm.
4. Enable easy *deployment* of the application. This is done by supporting a

*Launcher* and a *Deployer*. The system is responsible for initiating the different
stages of the computation at different resources.

GATES is also designed to execute applications on heterogeneous resources. The
only requirements for executing an application are: 1) support for a Java Virtual
Machine (JVM), as the applications are written in Java, 2) availability of GT3.0, and
3) a web server that supports the user application repository. Thus, the applications
are independent of processors and operating systems on which they are executed.

### 3.1.2 System Architecture and Design

The overall system architecture is shown in Figure 3.1. The system distinguishes
between an *application developer* and an *application user*. An application developer is
responsible for dividing an application into stages, choosing adaptation parameters,
and implementing the processing at each stage. Moreover, the developer writes an
XML file, specifying the configuration information of an application. Such informa-
tion includes the number of stages and where the stages’ codes are. After submitting
the codes to application repositories, the application developer informs an application
user of the URL link to the configuration file. An application user is only responsible
for starting and stopping an application.

The above design simplifies the task of application developers and users, as they
are not responsible for initiating the different stages on different resources. To support
the easy deployment and execution, the *Launcher* and the *Deployer* are used. The
Launcher is in charge of getting configuration files and analyzing them by using an
embedded XML parser. To start the application, the user simply passes the XML
file’s URL link to the Launcher. The Deployer is responsible for the deployment.
Specifically, it 1) receives the configuration information from the Launcher, 2) consults with a *grid resource manager* to find the nodes where the resources required by the individual stages are available, 3) initiates instances of GATES grid services at the nodes, 4) retrieves the stage codes from the application repositories, and 5) uploads the stage specific codes to every instance, thereby customizing it.

After the Deployer completes the deployment, the instances of the GATES grid service start to make network connections with each other and execute the stage functionalities. The GATES grid service is an OGSA Grid service [51] that implements the self-adaptation algorithm and is able to contain and execute user-specified codes.

### 3.1.3 Self-Adaptation API

We now describe the interface the middleware offers for supporting self-adaptation. As we stated earlier, the basis for self-adaptation is one or more *adjustable* parameters, whose value(s) can be tuned at runtime to achieve the best accuracy, while still meeting the real-time constraint. To use such functionality, stream processing applications are required to use a specific API to expose adaptation parameters. Specifically, the function `specifyPara(init_value, max_value, min_value, incr_or_decr)` is used to specify an initial value and a range of acceptable values of an adaptation parameter, and also state whether increasing the parameter value results in faster or slower processing.

An example to show the usages of these APIs is presented in Figure 3.2.

In the example above, an adaptation parameter, sampling rate, is specified. Using the function `specifyPara`, it is stated that the initial value of this parameter is 0.20, the range of values is between 0.01 and 1, and an increase in the value of this
public class Sampler implements StreamProcessor
{
    ...
    public void work(InputBufArray in, OutputBufArray out)
    {
        double sampling_rate;
        StreamServiceProvider.specifyPara(sampling_rate, 0.20, 1, 0.01, -1);
        ...
        // Process data
        while(true)
        {
            ...
            sampling_rate = GetSuggestedValue();
        }
    }
}
...
<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>Current length of the queue</td>
</tr>
<tr>
<td>$\bar{d}$</td>
<td>Average of the $d$ values in recent times</td>
</tr>
<tr>
<td>$\bar{d}$</td>
<td>Long-term average queue size factor</td>
</tr>
<tr>
<td>$t_1$</td>
<td>The number of times system was over-loaded</td>
</tr>
<tr>
<td>$t_2$</td>
<td>The number of times system was under-loaded</td>
</tr>
<tr>
<td>$w$</td>
<td>The number of times system was recently over-loaded</td>
</tr>
<tr>
<td>$\phi_1, \phi_3$</td>
<td>Functions reflecting queue’s long-term load</td>
</tr>
<tr>
<td>$\phi_2$</td>
<td>Functions reflecting queue’s recent load</td>
</tr>
<tr>
<td>$P$</td>
<td>Adjustment parameter for a server</td>
</tr>
<tr>
<td>$T_1$</td>
<td>No. of over-load exceptions occurring at the downstream server</td>
</tr>
<tr>
<td>$T_2$</td>
<td>No. of under-load exceptions occurring at the downstream server</td>
</tr>
<tr>
<td>$\sigma_1$</td>
<td>A function to factor $\bar{d}B$ in parameter adjustment</td>
</tr>
<tr>
<td>$\sigma_2$</td>
<td>A function to factor $\phi_1(T_1, T_2)$ in parameter adjustment</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Learning rate for $d$</td>
</tr>
<tr>
<td>$W$</td>
<td>Window size</td>
</tr>
<tr>
<td>$D$</td>
<td>Expected length of the queue</td>
</tr>
<tr>
<td>$C$</td>
<td>Maximum capacity of the queue</td>
</tr>
<tr>
<td>$P_1, P_2, P_3$</td>
<td>Weights to $\phi_1, \phi_2, \phi_3$, respectively</td>
</tr>
<tr>
<td>$LT_1$</td>
<td>Minimum threshold for the average queue size</td>
</tr>
<tr>
<td>$LT_2$</td>
<td>Maximum threshold for the average queue size</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of Symbols Used

### 3.2.1 Algorithm Overview

An application built on the GATES middleware comprises a set of pipelined stages. By modeling every stage as a server and viewing the input buffer of a stage as a queue of the server, we can get a queuing network model of the system. As an example, the model of the application shown in Figure 3.3 is presented in Figure 3.4.
Figure 3.3: An application that comprises three stages

Figure 3.4: A Queuing model of the system

Assume that the data arrives at a server in fixed-size packets. Let the average data arrival rate be denoted by $\lambda$. the rate at which the server is able to consume the packets is denoted by $\mu$.

If we have flexibility in controlling the accuracy of the analysis, our goal is to adjust the parameters to maintain a good balance between $\lambda$ and $\mu$. Clearly, if $\mu < \lambda$, the queue will saturate, and real-time constraint on processing cannot be met. In this case, we need to slow-down the processing that is performed by the sending server, i.e., make the processing more accurate. Alternatively, we can increase the rate of processing at the current server, possibly losing some accuracy. At the same time, if $\lambda$ is much lower than $\mu$, we are under-utilizing the current server. In this case, we can speed up the processing at the sending server.
As \( \lambda \) and \( \mu \) are not fixed at runtime, we focus on the current length of the queue, which is indicative of the ratio between the two. Our objective is to keep the average queue size within an interval between the two pre-defined thresholds. This goal could be achieved by dynamically adjusting the processing rates of the current and the preceding server, which, in turn, is done by properly tuning the value of adaptation parameters.

### 3.2.2 Detailed Description

This subsection gives a detailed description of the algorithm. The list of terms used in our algorithm is listed in Table 3.1.

The biggest challenge in the algorithm is to correctly weigh in the recent as well as long-term behavior of the queue. The idea is that we should be able to adjust to changes in the load quickly, but without making the system unstable. For this purpose, we introduce a long-term average queue size factor, denoted by \( \tilde{d} \). Thus, the two main steps in our algorithm are, evaluating \( \tilde{d} \), and adjusting parameters.

**Evaluating Long-Term Load:** This calculation is based upon three distinct load factors and learning by weighing these factors. These three load factors are denoted by \( \phi_1 \), and \( \phi_2 \) and \( \phi_3 \), respectively. A number of indicators of short-term and long-term load are used in computing these load factors.

If the current length of the queue, \( d \), is larger or less than some thresholds, we say that the queue is over or under-loaded. From the start of the system, \( t_1 \) is the number of times the system was found to be over-loaded and \( t_2 \) is the number of times the system was found to be under-loaded. \( t_1 \) and \( t_2 \) describe the long-term behavior of the system. To focus on the short-term behavior, we define the variable \( w \) and \( \tilde{d} \). We choose a window size \( W \) and record the last \( W \) times the system was observed to be
over or under-loaded. $w$ is a variable that is incremented by 1 for every occurrence of over-load within the window, and decremented by 1 for every occurrence of under-load within this window. Thus, $|w| \leq W$. $\bar{d}$ is the average of the $d$ values observed in recent times. Furthermore, $D$ is a user-defined expected length of the queue and $C$ is the capacity of the queue.

We compute $\phi_i$ as follows.

$$\phi_1(t_1, t_2) = \begin{cases} \frac{t_1 - t_2}{t_1 + t_2} & \text{if } (t_1 + t_2 > 0) \\ 0 & \text{if } (t_1 + t_2 = 0) \end{cases}$$ (3.1)

$$\phi_2(w) = \begin{cases} w \cdot \frac{1}{|w|} \cdot e^{\frac{|w| - W}{2}} & \text{if } (|w| \neq 0) \\ 0 & \text{if } (|w| = 0) \end{cases}$$ (3.2)

$$\phi_3(\bar{d}) = \begin{cases} \frac{\bar{d} - D}{D} & \text{if } \bar{d} < D \\ \frac{\bar{d} - D}{C - D} & \text{if } \bar{d} \geq D \end{cases}$$ (3.3)

Both $\phi_1$ and $\phi_3$ reflect the queue’s long-term load, whereas, $\phi_2$ reflects the queue’s recent load. The range of values of $\phi_i (i = 1, 2, 3)$ is $[-1, 1]$. Moreover, the closer $|\phi_i|$ is to 1, it is more likely that the unit is over or under-loaded.

Now, we can use the following equation to calculate $\bar{d}$.

$$\bar{d} = \alpha \ast \tilde{d} + (1 - \alpha) \ast (P_1 \ast \phi_1(t_1, t_2) + P_2 \ast \phi_2(w) + P_3 \ast \phi_3(\bar{d}))$$ (3.4)

Here, $P_1, P_2, P_3$ are the factors that give weight to $\phi_1$, $\phi_2$, and $\phi_3$, respectively, and satisfy the constraint $P_1 + P_2 + P_3 = 1$. Moreover, $0 < \alpha < 1$ is a pre-defined learning rate which helps remove transient behavior.

Similar to $\phi_i$, $\bar{d} \in [-1, 1]$, and the closer $|\bar{d}|$ to 1, it is more likely that the unit is having very high or low load. Particularly, when $\bar{d}$ exceeds the pre-defined interval
[LT₁, LT₂], the current server will report an under-load or over-load exception to the preceding server. The number of these exceptions is a factor used to tune adaptation parameters at the preceding server.

**Parameter Adjustment:** We now discuss how decisions about adjusting parameters are made. In the following discussion, we consider how to adjust the value of an adaptation parameter for processing at the server B in Figure 3.4. We assume that there is a parameter P_B at the server B; the increment of its value results in increasing the processing rate (and decreasing the accuracy). For such adjustment, we study both the average queue size, \( \tilde{d}_B \), and indicator(s) of load at the server C. The specific indicator of load at the server C that we use in our implementation is \( \phi_1(T_1, T_2) \). Here, \( T_1 \) and \( T_2 \) are the times of the over-load and under-load exceptions that the server C reported to the server B, and \( \phi_1 \) can be computed by applying Equation 3.1.

We design Equation 3.5 to calculate the adjustment of \( P_B \).

\[
\Delta P_B = \tilde{d}_B \ast \sigma_1(\tilde{d}_B) - \phi_1(T_1, T_2) \ast \sigma_2(\phi_1(T_1, T_2))
\]

(3.5)

The motivation behind the equation is as follows. If the value of \( \tilde{d}_B \) is higher, we want to increase the value of \( P_B \). This is because we want to reduce the load at B. At the same time, if the load at C is higher, i.e., \( \phi_1(T_1, T_2) \) is a high number, we want to slow down the rate at which B sends data to C. Therefore, we will like to decrease the value of \( P_B \). \( \sigma_1 \) and \( \sigma_2 \) are used to factor in the rate of variation of \( \tilde{d}_B \) and \( \phi_1(T_1, T_2) \), respectively. If the values of \( \tilde{d}_B \) and \( \phi_1(T_1, T_2) \) are unsteady, we want \( \Delta P_B \) to be large. Ultimately, \( P_B \) can quickly converge, and we also can keep the average queue size \( \tilde{d}_B \) within \([LT_1, LT_2]\) and eliminate the load exceptions reported from the server C.
3.3 Experimental Evaluation

This section presents results from a number of experiments we conducted to evaluate our GATES system. Specifically, we had the following goals: 1) Show how distributed processing of data streams is more efficient, 2) Show how a system with self-adaptation of parameters can achieve the right trade-off between efficiency and accuracy, and 3) Show how the self-adaptation algorithm currently implemented in GATES is able to choose the values of adaptation parameters when the execution configuration and/or the application’s resource requirements change.

For efficient and distributed processing of distributed data streams, we need high bandwidth networks and a certain level of quality of service support. Recent trends are clearly pointing in this direction, for example, the five sites that are part of the NSF funded Teragrid project expect to be connected with a 40 Gb/second network [100]. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single cluster. We introduced delay in the networks to create execution configurations with different bandwidths.

3.3.1 Applications

The experiments we report were conducted using two application templates, which are representative of the applications we described in Section 1.2.

Our first application is count-samps and implements a distributed version of the counting samples problem. The classical counting samples problem is as follows [56]. A data stream comprises a set of integers. We are interested in determining the n most frequently occurring values and their number of occurrences at any given point in the stream. Since it is not possible to store all values, a summary structure must be
maintained to determine the frequently occurring values. Gibbons and Matias have
developed an approximate method for answering such queries with limited memory.

The problem we consider is of determining frequently occurring values from a
stream, sub-streams of which arrive at different places. One option for solving this
problem is to communicate all sub-streams to a single location, and then apply the
original algorithm. However, bandwidth limitations may not allow this. An alternate
solution will be to create a summary structure for each sub-stream, and then commu-
nicate these to a central location. We can expect that larger the size of the summary,
more accurate the final results will be. For two sub-streams, we can range from stor-
ing \( n/2 \) frequently occurring values from each sub-stream to communicating entire
sub-streams. Thus, the number of frequently occurring values at each sub-stream is
the adaptation parameter used in this application.

The second application is **comp-steer**, based around the use of data stream pro-
cessing for computational steering. Here, a simulation running on one computer
generates a data stream, representing intermediate values at different points in the
mesh used for simulation. These values are sampled, communicated to another ma-
chine, and then analyzed. The processing time in the analysis phase is linear in the
volume of data that is output after the sampling. The sampling rate, denoting the
fraction of original values that are forwarded, is the adaptation parameter used in
this application.

### 3.3.2 Benefits of Distributed Processing

Our first experiment demonstrated the benefits associated with distributed pro-
cessing of data streams and used the **count-samps** application. Four different streams,
originating on four different machines, each produced 25,000 integers. Each of these
<table>
<thead>
<tr>
<th>Processing Style</th>
<th>Average Performance (sec.)</th>
<th>Avg. Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>257.5</td>
<td>.99</td>
</tr>
<tr>
<td>Distributed</td>
<td>180.8</td>
<td>.97</td>
</tr>
</tbody>
</table>

Table 3.2: Benefits of Centralized Processing: 4 Sub-streams

machines was connected to a *central* machine, where the answer to the query “*top 10 most frequently occurring integers and their frequency*” were desired at any given time. The available bandwidth between the stream sources and the *central* machine was 100 Kilo-Byte/second.

We considered two different versions. In the first, all data was forwarded to the central machine. All analysis was done at this machine. In the second, 100 most frequently occurring items at each stream were computed and forwarded to the central machine, where the final results were computed. Table 3.2 compares the execution time and accuracy between these two versions. The accuracy is measured by how often the top 10 most frequently occurring elements were correctly reported, and how correctly their frequency of occurrence was reported. Note that even the first version does not have an accuracy of 1. This is because the algorithm we implemented just takes one pass on the data and is approximate [56].

Our results show that distributed processing results in faster execution, with only a small loss of accuracy. Depending upon the rate at which data is generated, faster execution resulting from distributed processing can be crucial for meeting the real-time constraint. It should also be noted that this experiment had only four data sources, connected with a link having dedicated bandwidth to the *central* node. With larger...
number of data sources and/or other networking configurations, a larger difference can be expected.

### 3.3.3 Impact of Self Adaptation

<table>
<thead>
<tr>
<th>Network Bandwidth (Kilo-Byte/Sec.)</th>
<th>40 (sec.)</th>
<th>80 (sec.)</th>
<th>120 (sec.)</th>
<th>160 (sec.)</th>
<th>Adaptive Version (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>462.3</td>
<td>612.9</td>
<td>459.9</td>
<td>671</td>
<td>463.5</td>
</tr>
<tr>
<td>10</td>
<td>187.7</td>
<td>193.3</td>
<td>509.1</td>
<td>302.1</td>
<td>234.9</td>
</tr>
<tr>
<td>100</td>
<td>246.4</td>
<td>466.7</td>
<td>296.2</td>
<td>371.6</td>
<td>387.1</td>
</tr>
<tr>
<td>1000</td>
<td>240.4</td>
<td>298.8</td>
<td>307.7</td>
<td>478.0</td>
<td>399.9</td>
</tr>
</tbody>
</table>

Table 3.3: Execution Time of Different Versions

<table>
<thead>
<tr>
<th>Network Bandwidth (Kilo-Byte/Sec.)</th>
<th>40</th>
<th>80</th>
<th>120</th>
<th>160</th>
<th>Adaptive Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.891</td>
<td>.962</td>
<td>.981</td>
<td>.987</td>
<td>.986</td>
</tr>
<tr>
<td>10</td>
<td>.896</td>
<td>.963</td>
<td>.983</td>
<td>.992</td>
<td>.986</td>
</tr>
<tr>
<td>100</td>
<td>.887</td>
<td>.957</td>
<td>.979</td>
<td>.988</td>
<td>.974</td>
</tr>
<tr>
<td>1000</td>
<td>.879</td>
<td>.963</td>
<td>.983</td>
<td>.989</td>
<td>.988</td>
</tr>
</tbody>
</table>

Table 3.4: Accuracy of Different Versions

Our second experiment also used the count-samps application. Here, we focused on showing the impact of middleware-based self-adaptation on accuracy and execution time, as the available network bandwidth is varied. Similar to the previous experiment, there were four different data stream sources and the final results were desired at a central node. Five different versions of the application were created. The first four versions computed and communicated 40, 80, 120, and 160 most frequently.
occurring items at each data source. The last version used the *self-adaptation* supported by the middleware, and could automatically choose any value between 10 and 240. Four different networking configurations were considered, with a bandwidth of 1 KB/sec, 10 KB/sec, 100 KB/sec, and 1 MB/sec, respectively.

Tables 3.3 and 3.4 show the execution time and accuracy, respectively, of these five versions, and on the four different configurations. As Table 3.4 shows, the accuracy can be quite low if a very small value of the adaptation parameters is chosen. Similarly, Table 3.3 shows that the execution time can be very large if the value of the adaptation parameter is high and the bandwidth is small. The self-adapting version was able to provide a good trade-off between the execution time and accuracy, i.e., it never had very low accuracy, nor had very high execution times. In Table 3.3, note that higher value of parameter or lower bandwidth does not always increase the execution time. We believe that this is because of the impact of thread scheduler in JVM. Our future implementations will address this aberration in performance.

### 3.3.4 Self-Adaptation For Processing Constraint

Our third experiment used the *comp-steer* application to demonstrate how the middleware can perform self-adaptation to meet a processing constraint. Five different versions of the application were considered. The time required for post-processing was 1, 5, 8, 10, and 20 ms/byte, respectively, in these five version. The rate of data generation was approximately 160 bytes per second. The initial value of the sampling factor was fixed at 0.13 for all versions.

Figure 3.5 shows how the sampling factor chosen by the middleware varies over time. For the first two version, the value it converges to is 1, since processing is not a constraint. For the other three versions, it converges to .65, .55, and .31, respectively.
Thus, the middleware is automatically able to choose the highest sampling rate which still meets the real-time constraint on processing.

### 3.3.5 Self-Adaptation for a Network Constraint

Our last experiment also used the *comp-steer* application and focused on evaluating the self-adaptation in response to a networking constraint. Here, after sampling, the data is communicated over a link with a bandwidth of 10KB/sec. We considered five different versions, where the rate of data generation (before sampling) was 5KB/s, 10KB/s, 20KB/s, 40 KB/s, and 80KB/s, respectively. The initial rate of sampling factor was chosen to be 0.01 for all cases.

In Figure 3.6, we show how the middleware automatically converges to a sampling parameter for each of the different versions. Overall, this shows that the middleware is
able to self-adapt effectively, and achieve highest accuracy possible while maintaining the real-time processing constraint.

## 3.4 Summary

In this chapter, we have described the framework of the GATES middleware and its API functions. The rest of the chapter then have focused on the explanation of an initial self-adaptation algorithm and system evaluation.

The evaluation results have shown distributed processing supported by GATES can result in faster execution. Further, the results have indicated that the middleware is able to self-adapt effectively and achieve highest accuracy possible while maintaining the real-time processing constraint.

However, we found the initial self-adaptation algorithm converges not as fast as expected. Such slow convergence speed would eventually result in the self-adaptation
functionality’s failure, particularly when the system is under dynamic execution environment. We will introduce an enhanced self-adaptation algorithm in Chapter 4 to address this problem.
CHAPTER 4

IMPROVED SELF-ADAPTATION ALGORITHM

In the chapter 3, we have discussed an initial version of adaptation algorithm. This chapter will present and evaluate an improved runtime algorithm for supporting adaptive execution.

The improved self-adaptation algorithm has the following distinct characteristics from the initial one. First, it considers different possibilities for long-term loads at a processing stage and its next stages, and decides if the value of an adaptation parameter needs to be modified, and if so, in which direction. Second, to find the ideal new value of an adaptation parameter, it performs a binary search on the specified range of the parameter.

We have evaluated the new self-adaptation algorithm (“new” will not be specified in the rest of this chapter) extensively using these two streaming data mining applications. The main observations from our experiments are as follows. First, the algorithm is able to quickly converge to stable values of the adaptation parameter, for different data arrival rates, and independent of the initial value that is specified. Second, in a dynamic environment, the algorithm is able to adapt the processing rapidly. Finally, in both static and dynamic environments, the algorithm clearly outperforms
the initial algorithm described in Section 3.2 and an obvious alternative, which is based on linear-updates.

The rest of this chapter is organized as follows. The self-adaptation algorithm is described in Section 4.1. The streaming data mining algorithms we have implemented are described in Section 4.2. We evaluate our work in Section 4.3 and conclude in Section 4.4.

4.1 Self-Adaptation Algorithm

As we stated, the goal of this algorithm is to modify the value(s) of adaptation parameter(s) at runtime, so as to achieve highest level of accuracy while still meeting the real-time constraint. While the basic framework and some of the metrics used are identical to the adaptation algorithm presented in Section 3.2, the algorithm presented here is distinct in the following ways. First, it systematically considers different possibilities at a particular stage and its successor. Second, this algorithm does not require that certain functions and parameters be tuned for a particular application. Finally, it is able to converge to a stable value of an adaptation parameter faster.

4.1.1 Algorithm Overview

In Section 3.2.1, we have described the overview of the previous adjustment algorithm. Here we will like to emphasize the two challenges in such algorithms. The first challenge is to correctly weigh in the recent as well as long-term behavior of the queue. For this purpose, we introduce a long-term average queue size factor, denoted by $\tilde{d}$. The other challenge is to promptly have the adaptation parameter converge
### Symbols

<table>
<thead>
<tr>
<th>Variables</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>Current length of the queue</td>
</tr>
<tr>
<td>$d$</td>
<td>Average of the $d$ values in recent times</td>
</tr>
<tr>
<td>$\bar{d}$</td>
<td>Long-term average queue size factor</td>
</tr>
<tr>
<td>$t_1$</td>
<td>The number of times system was over-loaded</td>
</tr>
<tr>
<td>$t_2$</td>
<td>The number of times system was under-loaded</td>
</tr>
<tr>
<td>$w$</td>
<td>The difference in the number of times system was recently under/over-loaded</td>
</tr>
<tr>
<td>$\phi_1$</td>
<td>Functions reflecting queue’s long-term load</td>
</tr>
<tr>
<td>$\phi_2, \phi_3$</td>
<td>Functions reflecting queue’s recent load</td>
</tr>
<tr>
<td>$P$</td>
<td>Adaptation parameter for a server</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constants</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>Learning rate for $d$</td>
</tr>
<tr>
<td>$W$</td>
<td>Window size</td>
</tr>
<tr>
<td>$E$</td>
<td>Expected length of the queue</td>
</tr>
<tr>
<td>$L$</td>
<td>Capacity of the queue</td>
</tr>
<tr>
<td>$P_1, P_2, P_3$</td>
<td>Weights to $\phi_1, \phi_2, \phi_3$, respectively</td>
</tr>
<tr>
<td>$LT$</td>
<td>Maximum (Minimum) threshold for the load</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of Symbols Used

To an ideal value. This will allow the algorithm to be sensitive to a varying environment. The main two steps of the algorithm, evaluating $\bar{d}$, and adjusting parameters, respectively, overcome these two challenges. The list of terms used in our algorithm is listed in Table 4.1.

Regarding the first step of evaluating $\bar{d}$, we can use the following equation:

$$\bar{d} = \alpha \times \bar{d} + (1 - \alpha) \times (P_1 \times \phi_1(t_1, t_2) + P_2 \times \phi_2(w) + P_3 \times \phi_3(\bar{d})) \quad (4.1)$$

Equation 4.1 is identical to that used in the previous algorithm described in Section 3.2.1. Similarly, $\bar{d} \in [-1, 1]$, and the closer $|\bar{d}|$ to 1, it is more likely that the unit is having very high or low load. In particular, when $\bar{d}$ exceeds the pre-defined interval
$[-LT, LT]$, the current server will be thought of as being under-loaded or over-loaded. These load states indicated by $\tilde{d}$ are used to tune adaptation parameters.

### 4.1.2 New Algorithm and Parameter Adjustment

<table>
<thead>
<tr>
<th>State</th>
<th>Condition</th>
<th>Adjustment Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$d_B &lt; -LT$, $d_C &lt; -LT$</td>
<td>Decrease $\mathcal{P}_B$ and increase accuracy</td>
</tr>
<tr>
<td>2</td>
<td>$d_B &lt; -LT$, $-LT \leq d_C \leq LT$</td>
<td>Decrease $\mathcal{P}_B$ and increase accuracy</td>
</tr>
<tr>
<td>3</td>
<td>$d_B &lt; -LT$, $d_C &gt; LT$</td>
<td>Decrease $\mathcal{P}_B$ and increase accuracy</td>
</tr>
<tr>
<td>4</td>
<td>$-LT \leq d_B \leq LT$, $d_C &lt; -LT$</td>
<td>Do not change $\mathcal{P}_B$</td>
</tr>
<tr>
<td>5</td>
<td>$-LT \leq d_B \leq LT$, $-LT \leq d_C \leq LT$</td>
<td>Do not change $\mathcal{P}_B$</td>
</tr>
<tr>
<td>6</td>
<td>$-LT \leq d_B \leq LT$, $d_C &gt; LT$</td>
<td>Decrease $\mathcal{P}_B$ and increase accuracy</td>
</tr>
<tr>
<td>7</td>
<td>$d_B &gt; LT$, $d_C &lt; -LT$</td>
<td>Increase $\mathcal{P}_B$ to speed up the processing rate</td>
</tr>
<tr>
<td>8</td>
<td>$d_B &gt; LT$, $-LT \leq d_C \leq LT$</td>
<td>Do not change $\mathcal{P}_B$</td>
</tr>
<tr>
<td>9</td>
<td>$d_B &gt; LT$, $d_C &gt; LT$</td>
<td>Decrease $\mathcal{P}_B$ to and increase accuracy</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of Load States

We now describe our new algorithm, focusing on how the adaptation parameter is modified at runtime according to the evaluation of $\tilde{d}$. In the following discussion, we specifically focus on the server $B$ in Figure 3.4. The server $B$ receives data from the server $A$ and processes and forwards data to the server $C$. There could be two types of adaptation parameter $\mathcal{P}_B$ at the server $B$. One is called performance parameter. Incrementing its value results in increasing the processing rate, $\mu_B$, and decreasing the accuracy of the processing. Thus, a higher value of $\mathcal{P}_B$ will allow the server $B$ to process the data faster. However, this will also result in a higher load for the server $C$. Another type of adjustment parameter is accuracy parameter. Contrary to a performance parameter, changing the value of an accuracy parameter can contribute to the reverse impacts as we describe above. To make it clearer to
explaining the algorithm, we assume that the server $B$ has a performance parameter. While we considered both accuracy and performance parameters when we evaluated the algorithm.

There are two main issues for the parameter adaptation component of our algorithm. The first question is deciding when we should increase or decrease $P_B$, and when we should leave it unchanged. The second question is deciding the new value of $P_B$, when we need to change the parameter. To answer the first question, we define the load state for a given server.

**Definition 1** The load state for the server $B$, denoted by $S_B$, is based on the tuple $(\tilde{d}_B, \tilde{d}_C)$. Particularly, we consider three possibilities for each of $\tilde{d}_B$ and $\tilde{d}_C$, which are, $\tilde{d} < -LT$, $-LT \leq \tilde{d} \leq LT$ or $\tilde{d} > LT$. Thus, there are nine distinct load states for a server.

The nine possible load states are shown in Table 4.2. Among these states, we consider $S_4$, $S_5$ and $S_8$ convergent states. Each of the other 6 states is considered non-convergent. In a convergent state, there is no need to modify the adaptation parameter at the server $B$, whereas in a non-convergent state, the parameter $P_B$ needs to be modified.

The action taken by our algorithm in each of these cases is shown in Table 4.2. We now explain the rationale for the strategy for different cases.

Initially, we consider states $S_1$, $S_2$, and $S_3$. These three states are common in the fact that the server $B$ is under utilized. In such a case, we can improve the accuracy of processing at the server $B$, irrespective of the state of the server $C$. Note that in a streaming environment, the data arrival rate at the first stage cannot be modified.
Thus, for each of the subsequent stages, our goal is to be able to achieve highest possible accuracy, without making them the bottleneck.

Next, let us consider the convergent states, which are $S_4$, $S_5$ and $S_8$. When the server B’s state is $S_4$, it implies that the server B’s load is within the desired range, whereas, the server C is under-utilized. In such a case, the server $B$ does not need to make any change. Note, however, that the same algorithm is applied on the server $C$ also, and in this case, server $C$ can increase the accuracy of the processing, if it has a parameter to modify. In the state $S_5$, the load at both $B$ and $C$ is in the desired range, so clearly, no changes are required. The last convergent state is $S_8$. In this state, the server $B$ is overloaded, but the server $C$ is not underloaded. Thus, increasing the processing rate at the server $B$ can make $C$ overloaded, which is not desirable. Thus, the server $B$ does not make a change. Again, note that the same algorithm is being applied at the server $A$, and if possible, server $A$ should decrease the rate at which it forwards data to the server $B$.

The three remaining states are $S_6$, $S_7$ and $S_9$. In the state $S_6$, the server $C$ is overloaded. In this case, the server $B$ will adjust to slow down the rate at which it forwards the data to the server $C$. Note that it may be possible for $C$ to avoid the overload by adjusting a parameter locally, but the server $B$ does not assume that such a parameter exists. The action for the state $S_7$ is easy to explain. The server $B$ is overloaded, whereas the server $C$ is underloaded. Thus, $B$ needs to increase the rate of processing. The state $S_9$ is quite challenging, as both $B$ and $C$ are overloaded. In this case, the only likely acceptable solution will be to have the server $A$ slow down the processing. To facilitate this, the server $B$ further slows down the processing. This will reduce the load at $C$, but can increase the load at $B$. Since the server $A$
views the load information at the server $B$, and not the server $C$, this is most likely
to create convergence.

The second important challenge for our algorithm is to determine the new (higher
or lower) value for $\mathcal{P}_B$, when a change in its value is needed. There are several
considerations that must be used. First, the system should be able to converge
to an ideal value for the adaptation parameters, i.e., the one which allows the best
accuracy, while still meeting the real-time constraint. Second, this convergence should
be achieved quickly. This is important for adapting in a dynamic environment, and
for avoiding loss of packets when buffer sizes are small.

One obvious way for adjusting parameters will be a linear increase or decrease,
i.e., changing $\mathcal{P}_B$ by a fixed value in each iteration. As we will show through our
experimental evaluation, this scheme has the following shortcoming. If the amount of
the change is large, the system may never reach the ideal value. On the other hand,
if the amount of the change is small, a large number of iterations may be required for
convergence.

Therefore, we have designed a method which is similar to a binary tree search. The
overall algorithm is shown in Figure 4.1. Two variables, left border and right border,
are used in the third step of the algorithm. These define a range within which $\mathcal{P}_B$ can
be changed. Their initial values are the minimum and the maximum values of the
adaptation parameter, which the GATES API requires from application developers.
These values are denoted as min value and max value, respectively.

We proceed as follows. The current value is denoted as val. When it is determined
that $\mathcal{P}_B$ should be decremented, the new range is changed to $[\text{left border}, \text{val}]$, and
$\text{val}$ is updated to the mid-point of the new range. When it is determined that $\mathcal{P}_B$
Algorithm getSuggestedPara()

1. //Calculate the tuple $T$
   
   $d_B = \text{calcuateMyLongTermLoad}()$;
   
   next_server = getMyFollowingServer();
   
   $d_C = \text{next_server.calcuateMyLongTermLoad}()$;
   
   $T = (d_B, d_C)$

2. Determine which load state $T$ belongs to

3. //Update the adaptation parameter's value
   if the direction is "do not change $P_B"
   
   {  
     // the current value of $P_B$ is val
     do nothing
   }

   if the direction is "decrease $P_B"
   
   {  
     //The initial value of right border is max_value
     right_border = val
     
     val = \frac{\text{left border} + \text{right border}}{2}
   }

   if the direction is 'increase $P_B'
   
   {  
     //The initial value of left border is min_value
     left_border = val
     
     val = \frac{\text{left border} + \text{right border}}{2}
   }

4. return val

Figure 4.1: Self Adaptation Algorithm
should be incremented, the new range is $[val, right\_border]$, and again, $val$ is updated to the mid-point of the new range.

Therefore, $P_B$ will eventually converge to a value within the range $[max\_value - min\_value]$, assuming that the environment is static. In practice, our algorithm only requires between 3 and 5 steps to converge in a static environment.

Finally, we consider a *dynamic* environment. An environment is *dynamic* if the data arrival rate, available network bandwidth, and/or CPU cycle availability is varying. In such a case, the algorithm needs to be able to determine new ideal value. We modify the algorithm as follows. We store all previous ranges in a stack. When $P_B$ needs to be changed, and if current range is very small compared with the initial range, the previous range is popped from the stack.

### 4.2 Streaming Data Mining Applications

This section describes the two streaming data mining applications that were used for our experimental study. We show how these applications can be developed and deployed using GATES system’s support. We also show how each of these applications naturally has an adaptation parameter, which allows trade-off between processing rate and accuracy.
Figure 4.3: Communication Topology for the Dist-Freq-Counting Application
The first application is clustering evolving data streams [7], and is referred to as CluStream. Clustering involves grouping similar object or data points from a given set into clusters. The particular problem considered here is clustering data arriving in continuous streams, especially as the distribution of data can change over time.

The algorithm we consider [7] approaches the problem as follows. The clustering process is divided into two major steps. The first step involves computing micro-clusters that summarize statistical information in a data stream. The second step uses micro-clusters to compute the final clusters.

This two-step clustering algorithm can be easily implemented using the GATES middleware. Figure 4.2 shows the three stages that are used. The first stage is simply the data source, which sends streaming data to the second stage. The second stage computes micro-clusters. After a certain number of data points have been processed, it sends the computed micro-clusters to the third stage. The third and the final stage then apply the modified $k$-means algorithm [7] to create and output the final clusters.

Note that the final number of clusters desired is specified by the user. However, the number of micro-clusters computed by the second stage needs to be chosen by the algorithm. A larger number of micro-clusters result in better accuracy in computing the final clusters. But, the amount of computation at the second stage and the volume of communication between the second and third stages are both directly proportional to the number of micro-clusters. Thus, the number of micro-clusters becomes an accuracy parameter for this application.

The second application we have studied finds frequent occurring itemsets in a distributed data stream and is referred to as Dist-Freq-Counting [82]. The problem is of finding frequently occurring itemsets across a set of data streams. If the distribution
of data across the different streams is different, and if the communication bandwidth is limited, this problem can be quite challenging.

The algorithm we consider is an extension of a proposed algorithm for finding frequent items from distributed streams [82]. The algorithm addresses the problem stated above by arranging the nodes in a hierarchical structure. Figure 4.3 shows an example of such a structure. Each monitor node $M_i$ counts the frequencies of itemsets appearing in the stream $S_i$, and periodically sends this information to its parent node, which could be an intermediate node or the root node. Intermediate nodes combine the frequency information received from their children and pass them up to their parent node. Finally, the root node outputs the itemsets whose frequencies exceed the specified support threshold $\tau$.

To reduce communication loads, the monitor and intermediate nodes should avoid sending less frequent itemsets over the links. Therefore, the algorithm uses an error tolerance parameter $\epsilon$ at every node, except the data sources. Only the itemsets with frequency greater than this parameter are forwarded to the next node.

The value of a tolerance parameter impacts both the processing rate and the accuracy. With a higher value, we could miss itemsets which may be frequent overall. With a lower value, the volume of communication can be increased. Generally, it is desirable that all nodes at the same level use the same tolerance value, and the tolerance value (frequency) is increased as we move closer to the root node.

In our implementation, we consider the tolerance parameter at the monitor nodes as a performance parameter. This is because the communication volume at this stage can be the highest, and therefore, the tolerance parameter has the largest impact on the performance.
4.3 Experimental Evaluation

This section presents results from a number of experiments we conducted to evaluate the self-algorithm and the use of the GATES middleware for streaming data mining applications. Specifically, we had the following goals in our experiments:

- Demonstrate that the self-adaptation algorithm is able to quickly converge to the ideal value of the adaptation parameter, for different data stream arrival rates.
- Show that the algorithm is not sensitive to the initial value of adaptation parameter.
- Show that how the algorithm is able to vary the value of an adaptation parameter as the execution environment changes dynamically.
- Show that our algorithm is more efficient and effective than the algorithm presented in Section 3.2 and an obvious alternative, which involves the use of linear adjustments.

The experiments were conducted using the two streaming data mining applications described in the previous section. For the Clustream application, we used the KDD-CUP’99 Network Intrusion Detection dataset. For Dist-Freq-Counting, we use a dataset generated by the IBM data generator [8]. The average size of each transaction in this dataset is 6.

We conducted 3 sets of experiments, which are described in the rest of this section.
Figure 4.4: Convergence under Different Data Arrival Rates for Clustream, When Number of Micro-Clusters is Initialized to 20 and 40
Figure 4.5: Convergence under Different Data Arrival Rates for Clustream, When Number of Micro-Clusters is Initialized to 80 and 110
Figure 4.6: Convergence under Different Data Arrival Rates for Dist-Freq-Counting, When Error Tolerance is Initialized to 0.001 and 0.002
Figure 4.7: Convergence under Different Data Arrival Rates for Dist-Freq-Counting, When Error Tolerance is Initialized to 0.004 and 0.0045
4.3.1 Convergence In a Static Environment

Our first experiment demonstrated that the self-adaptation algorithm can choose the ideal values for the adaptation parameters under different data arrival rates, irrespective of the initial values of these parameters.

For Clustream, we initialized the number of micro-centers to 20, 40, 80, and 110. The allowed range of this parameter was [10, 110]. We used one data source, and controlled the data arrival rate at the second stage to be 50, 100, 200, and 400 Kbps, respectively. The results are shown in Figure 4.4 and 4.5. Let us consider the first chart in Figure 4.4, where the initial value is 20. The value of the number of micro-clusters converged to 110, 73, 44, and 20, for the four data arrival rates we considered. The convergence occurred in an average of 5 steps, which corresponds to an average of 53 seconds. The X-axis in this chart is the number of steps, which denotes the number of invocations of the Algorithm shown in Figure 4.1.

The other three charts in Figure 4.4 and 4.5 correspond to initial number of micro-centers being 40, 80, and 110, respectively. The algorithm converges to a stable value in each of the cases.

A similar set of experiments were also conducted using our second application, Dist-Freq-Counting. The results are shown in Figure 4.6 and 4.7. We set the range of \(\epsilon_2\) to be [0.0001, 0.0045]. We considered six different data arrival rates and four different initial values. The algorithm converges in each of the cases.

Figures 4.8, 4.9, 4.10, 4.11, 4.12 and 4.13 further illustrate the stable behavior of the algorithm. In these experiments, we considered a number of different initial values for a given data arrival rate, and see how the algorithm converges.
Figure 4.8: Convergence under Different Initial Values for Clustream, When Data Arrival Rate is 50kbps

Figure 4.9: Convergence under Different Initial Values for Clustream, When Data Arrival Rate is 100kbps
Data arrival rate is 400Kbps

Figure 4.10: Convergence under Different Initial Values for Clustream, When Data Arrival Rate is 400kbps

Data arrival rate is 10Kbps

Figure 4.11: Convergence under Different Initial Values for Dist-Freq-Counting, When Data Arrival Rate is 10kbps
Data arrival rate is 40Kbps

Figure 4.12: Convergence under Different Initial Values for Dist-Freq-Counting, When Data Arrival Rate is 40kbps

Data arrival rate is 120Kbps

Figure 4.13: Convergence under Different Initial Values for Dist-Freq-Counting, When Data Arrival Rate is 10kbps
Figure 4.8, 4.9, and 4.10 correspond to the Clustream application, with data arrival rates of 50, 100, and 400 Kbps, respectively. For each of these three data arrival rates, we consider initial number of micro-clusters to be 20, 40, 60, 80, and 100. We can see from these charts that while the initial value can impact how quickly we reach a stable value, the final converged value is independent of the initial value in each case. The final converged number of micro-centers is 100, 73, and 20, when the data arrival rates are 100, 200, and 400 Kbps, respectively.

Similarly, Figure 4.11, 4.12 and 4.13 correspond to Dist-Freq-Counting, with data arrival rates of 10, 40, and 120 Kbps, respectively. We consider five different initial values of the adaptation parameters, $\epsilon_2$, which are 0.001, 0.002, 0.003, 0.004, and 0.0045. Again, the final converged value is independent of the initial value.

### 4.3.2 Adaptation in a Dynamic Environment

In this subsection, we show that our binary search based adaptation algorithm can quickly adjust the value of an adaptation parameter in a dynamic environment. Such dynamic adaptation may be needed if the data arrival rate varies frequently, and/or if the available network bandwidth or CPU cycles can vary. For our experiments, we only considered variations in data arrival rates.

The two charts in Figure 4.14 consider the Clustream application. The allowed range of number of micro-clusters is [10, 100]. The initial value is 60. The initial data arrival rate is 400 Kbps. The data arrival rate is varied with two different frequencies, which are every 120 and 30 seconds, respectively. The first and the second charts in Figure 4.14 correspond to these two frequencies. The data arrival rate is varied
Figure 4.14: Algorithm Behavior in a Dynamic Environment: Clustream Application
Figure 4.15: Algorithm Behavior in a Dynamic Environment: Dist-Freq-Counting
between 40 Kbps and 400 Kbps, with a step of 60 Kbps, applied every 120 or 30 seconds.

The Y-axis in the charts in Figure 4.14 corresponds to both the data arrival rates, and the number of micro-clusters chosen by our algorithm. The scales for these values are shown in left and right side, respectively, of each chart. Our results show that our algorithm is able to vary the number of micro-clusters with the same frequency as rate of change of data arrival rate. As the data arrival rate increases, the number of micro-clusters goes down to the minimum possible value of 10. As the data arrival rate decreases, it goes back up to a higher value.

One interesting question is, how does the frequency of change of data arrival rates impact the algorithm. We can see that the range of number of microclusters is [10, 85] with the slower rate of change (first chart) and [10, 60] with the higher rate of change (second chart). This is because our algorithm needs to see an under-loaded server for a long duration to increase the accuracy of the processing to the highest levels.

We also conducted the same experiment with the other application. The results are shown in Figure 4.15. The results are very similar. The range of variation of $\epsilon_2$ is quite limited when the frequency in the change of data arrival rates is higher.

### 4.3.3 Comparing Different Algorithms

We now compare the self-adaptation algorithm with the obvious alternative, which is a linear update algorithm, and the algorithm presented in our previous work [33]. We compared these algorithms in both static and dynamic environments.

We first consider Clustream. We initially compared the binary search algorithm with the other two algorithms in a static environment. The results are shown in
Figure 4.16: Comparing Different Self-Adaptation Algorithms in a Static Environment: Clustream Application

Figure 4.16. For this experiment, the data arrival rate is fixed at 200 Kbps and the number of micro-centers is allowed to vary within the range [10, 110]. We consider two scenarios with the binary search algorithm, which correspond to the initial number of micro-clustering being 60 and 100. We also consider two scenarios with the linear-update algorithm. Though the initial number of micro-clusters is 60 in both the cases, the amount of update in each case is \((110 - 10)/10 = 10\) and \((110 - 10)/100 = 1\), respectively.

The binary search algorithm is able to converge to the ideal value of 46 within 6 to 8 steps in the both the cases. With the linear update algorithm with a step of 10, the algorithm never converges. Instead, after a few iterations, it starts alternating between 40 and 50. When we use a step of 1, the algorithm converges, but takes nearly
Figure 4.17: Comparing Different Self-Adaptation Algorithms in a Dynamic Environment: Clustream Application
Data arrival rate is 60Kbps

![Graph showing comparison of different self-adaptation algorithms.](image)

Figure 4.18: Comparing Different Self-Adaptation Algorithms in a Static Environment: Dist-Freq-Counting

13 steps, or about twice as long as the binary search algorithm. This algorithm shows the main limitation of the linear-update algorithm, which is the difficulty of choosing an appropriate step value. A large value can result in an unstable behaviour, whereas, a small step value can create large delays in convergence.

Finally, as we can see from the Figure, the number of micro-clusters continuously decreases with the algorithm presented in Section 3.2 [33]. This is because this algorithm did not consider the possibility of a compute-intensive stage being the bottleneck.

We also compared these three algorithms in a dynamic environment. We considered two cases, with the frequency of the change of data arrivals rates being once
Figure 4.19: Comparing Different Self-Adaptation Algorithms in a Dynamic Environment: Dist-Freq-Counting
every 60 seconds and 180 seconds. The results for these two cases are shown in Figure 4.17. Again, we considered linear-update algorithm with steps of 10 and 1. The conclusions from these experiments are similar to those from the static experiments. The linear-update algorithm with the step of 10 is unstable and with a step of 1, it is slower to adjust. Our previous algorithm slowly converges to the minimum value of the adaptation parameter.

Figure 4.18 and 4.19 show the results from the same set of experiments, but using Dist-Freq-Counting. The results are identical.

4.4 Summary

In this chapter, we have focused on discussing an improved self-adaptation algorithm. The improved self-adaptation algorithm has the following distinct characteristics from the initial one described in Chapter 3. First, it systematically considers different possibilities at a particular stage and its successor. Second, this algorithm uses a binary search to determine the convergence values of adaptation parameters.

We have implemented two streaming data mining applications using our middleware, and have extensively evaluated the enhanced adaptive capabilities of our middleware. The main observations from our experiments are as follows. First, the new algorithm is able to quickly converge to stable values of the adaptation parameter, for different data arrival rates, and independent of the specified initial value. Second, in a dynamic environment, the algorithm is able to adapt the processing rapidly. Finally, in both static and dynamic environments, the algorithm clearly outperforms the initial algorithm and an obvious alternative, which is based on linear-updates.
In this chapter, we consider the problem of resource allocation for an application using the GATES system. Though resource discovery and resource allocation have been active topics in grid community, the pipelined processing and real-time constraint required by distributed streaming applications pose new challenges. We make the following contributions: 1) We present a static resource allocation algorithm that is based on minimal spanning trees, 2) We have developed a framework to support resource monitoring and dynamic resource allocation, and 3) We describe a dynamic resource allocation algorithm and how we have implemented dynamic migration to support it.

This chapter is organized as follows. The resource allocation problem and our static algorithm are described in Section 5.1. Algorithm, architecture, and implementation used for dynamic resource allocation are described in Section 5.2. We evaluate our static and dynamic algorithms in Section 5.3 and conclude in Section 5.4.

5.1 Static Resource Allocation Algorithm

In this section, we describe our algorithm for static resource allocation. Initially, we describe the problem and argue about its complexity.
5.1.1 Problem Definition and Complexity

The resource allocation problem for a GATES application is essentially that of creating a deployment configuration. A deployment configuration of an application comprises the following components:

1. The number of data sources and their location.
2. The destination, i.e., the node where the final results are needed.
3. The number of stages in the application.
4. The number of instances of each stage.
5. How the instances connect to each other.
6. The node at which each instance is assigned.
Figure 5.1 shows two possible deployment configurations for an application that has 4 data sources and 5 stages.

Now, let us consider the problem of creating a deployment configuration for an execution of an application. The components 1, 2, and 3 of the deployment configuration are known when an application is initiated. Therefore, the problem is that of determining components 4, 5, and 6. Determining components 4, 5, and 6 involves both resource discovery and resource allocation. Our focus is on the problem of resource allocation. For resource discovery, we can make use of information services [42] provided by Globus toolkit 3.0 (GT3.0) to collect and aggregate resource information.

One possible approach for resource allocation is to enumerate and compare all possible configurations and find the one that will enable the best performance. However, such an exhaustive search algorithm has at least an exponential complexity. Given an application with \( m \) stages, \( n \) data sources and \( k \) available computing nodes for placement of stages’ instances, the number of possible configurations can be denoted by \( F(n, m, k) \), where,

\[
F(2, n, k) = 1
\]

\[
F(m, n, k) = \sum_{1 \leq i \leq n} \left( S_n^{(i)} \times F(m - 1, i, k - i) \times P_k^i \right)
\]

where, \( m \geq 3, n \geq 1, k \geq (m \times n) \), and \( P_k^i = \frac{k^i}{(k - i)!} \). \( S_n^{(i)} \) denotes the Stirling numbers of the second kind. If there are only 3 stages, the number of all possible configurations is:

\[
F(3, n, k) = \sum_{1 \leq i \leq n} \left( S_n^{(i)} \times P_k^i \right)
\]

74
The above derivation shows that a lower bound on the complexity of the exhaustive search algorithm is $\Omega(n^n)$. Therefore, it is not practical. Below, we will describe our algorithm that can find a deployment configuration in $O(nk^2)$ time.

### 5.1.2 Algorithm Description

The goal of our algorithm is to determine the deployment configuration that gives the application the best chance to achieve the real-time constraint, which still maintaining high accuracy in the processing. In this section, we only consider static resource allocation, and dynamic resource allocation is the topic for the next section.

We also assume that except for the final stage in the pipeline, where data from all sources must be combined together, stages in the application could be processed independently for each data source. The main observation in our algorithm design is that data arrival rates at the first one or two stages are typically so high that high network bandwidths are desired at these stages. After data have been processed by these stages, the arrival rates at the following stages typically decrease significantly.

Our algorithm is also based on the assumption that computation is typically not the bottleneck in the processing.

The algorithm has two main steps. First, we create a key path corresponding to each data source. Second, we merge these key paths to create a layout tree.

The algorithm proceeds as follows. We initially construct a weighted graph in which every node is viewed as a vertex, and a network connection between two nodes is viewed as an edge. By node, we mean any computing unit, which could be a cluster...
or an SMP machine, and is capable of executing multiple processes. The main idea is that communication bandwidth for processes within a node is much higher than the bandwidth between the nodes. The weight of an edge is the negative value of the network connection’s bandwidth. Given this formulation, our goal is to place stages so as to minimize the network communication time in the processing. Therefore, for every data source, we construct a Minimum Spanning Tree (MST) by making the data source the initial set and applying the Prim’s algorithm to the graph. Note that we prefer to apply the MST algorithm, rather than the shortest path algorithm, because it aggressively minimizes the weight at the top level of the tree.

Next, we seek the key path for each data source. This is the path from the data source node to the destination node in the MST corresponding to each data source. We start to mark nodes in the path, i.e., the parent node of the data source node is marked as the second stage, the grandparent node of the data source node is marked as the third stage, etc.

Ideally, we will like to have the last stage placed on the destination node. However, this will only happen if the number of tree nodes in a key path is equal to the number of stages being deployed. This may not be true in practice, and the number of stages could be both less than or greater than the number of nodes in the path. We make the following adjustments in such cases. If the length of a key path is longer than the number of stages \( m \), we insert some transport stages, called transporters, which simply forward the data they receive. Adding transporters does introduce some overheads. However, we believe that these overheads are nominal compared with the delays caused by choosing another lower bandwidth path, simply because it has exactly \( m \) number of nodes. Moreover, these transporters are always deployed at the end of a
pipeline, where the arrival rates are not high. Towards the end of this section, we will describe an optimization to reduce the number of transporter if possible. When the length of a key path is shorter, the additional stages are deployed at the parent node of the data source node. The reason again is that higher data arrival rates are typically seen at the beginning of a pipeline.

Now, we have a key path corresponding to each data source, and we need to create a layout tree. By default, we will have one instance of each stage for each data source, with the exception of the last stage. As we stated earlier, our goal is to minimize the communication time and computation is typically not a bottleneck. Therefore, we proceed as follows. Consider two different key paths which involve the same node. In general, the node could be executing different stages for different key paths. However, if it executes the same stage, we merge the paths. By repeating this step, we can get a layout tree from the set of key paths.

A layout tree determines the components 4 and 5 of a deployment configuration. To decide the component 6, we need to map a vertex in the layout tree to a computing node. We can query the node information service by specifying the resource requirements of the stage that is supposed to be deployed in this node. Thus, we can get a deployment configuration and then call the launcher program to automatically launch the application.

We now state the complexity of this algorithm. Recall that the number of data sources is $n$, the number of stages is $m$, and the number of available nodes is $k (k > m)$. The main cost is in the invocation of the Prim’s algorithm. Each invocation takes $O(k^2)$ time and we need $n$ invocations. Thus, the complexity of the algorithm is $O(nk^2)$. 

77
5.1.3 Additional Optimization

Although the overheads introduced by transporters are not critical, we will still like to find unnecessary transporters and eliminate them. Assume that a key path is denoted by $N_1, E_1, N_2, E_2, \ldots, N_i, E_i, \ldots, N_m$, where the data source is denoted by $N_1$, the destination is denoted by $N_m$, $N_i$ is the $i$th node in the path, and $E_i$ is the edge connecting $N_i$ with $N_{i+1}$. If the node $N_i$ is marked as a transporter, $N_{i-1}$ has an edge $e$ in the graph which connects to $N_j (j > i)$, and the weight of $e$ is smaller than that of the edges from $E_1$ to $E_{j-1}$. Then the nodes from $N_i$ to $N_{j-1}$ are unnecessary transporters. We can eliminate them and replace the edges from $E_i$ to $E_{j-1}$ with the edge $e$. As we will show in our experiments, an optimized configuration does achieve better performance than the non-optimized configuration.

5.2 Dynamic Resource Allocation

We now focus on dynamic resource allocation. We describe the overall architecture for resource monitoring and dynamic allocation, followed by the dynamic resource allocation algorithm and implementation, and a brief description of our implementation of dynamic migration.

5.2.1 Architecture for Dynamic Resource Allocation

As we stated earlier, we make use of information services [42] provided by GT3.0 to collect resource information in clusters. More specifically, at every computing node that has installed GT3.0, an information service is always running. This service periodically collects resource information from the node, such as the CPU loads and
Figure 5.2: Architecture of the Resource Allocation Components
the available memory. For every cluster, we also setup an Index Service to aggregate such resource information collected by the information services. Thus, to discover desired CPU and memory resources, we just need to query these index services. For the network resources, we could utilize the Network Weather Service [122] to measure end-to-end TCP/IP performance. We are more concerned about the performance of network links connecting different clusters, since high-speed interconnects are generally always available within the nodes in a cluster. Consistent with our approach for static resource allocation, we are assuming that the computing resources available to us are large clusters, which have adequate CPU and memory resources. Therefore, we view inter-cluster bandwidth as the most critical resource. Our overall strategy for dynamic resource allocation is that if CPU and memory resources become a bottleneck, the computing stages can be moved to a different set of nodes within the same cluster. However, if inter-cluster bandwidth becomes a bottleneck, a new series of clusters and corresponding inter-cluster network links have to be allocated.

Figure 5.2 shows the overall architecture. The important components are the Grid Resource Manager and the GATES Monitoring Service. The Grid Resource Manager is responsible for: 1) collecting network topology and bandwidth information, 2) constructing a weighted graph and keeping it updated, 3) searching for a proper node (or a set of nodes) in a cluster by querying its index service, and 4) applying the static and the dynamic resource allocation algorithms. The Grid Resource Manager is involved in both the static and the dynamic allocation. For static allocation, when an application is initiated, the Deployer sends the initial configuration information to the Grid Resource Manager that applies the static algorithm to generate a deployment configuration. According to the deployment configuration that is created, the
Deployer initiates GATES grid service and launches the application as described in Section 3.1.

For dynamic resource scheme, the GATES monitoring service also plays an important role. As indicated in the Figure 5.2, each instance of the GATES Grid Service has a corresponding monitor. These monitors are created by their corresponding grid service instances when they are initiated. The main function of a monitor is to determine if its corresponding grid service is the bottleneck of the pipeline. Specifically, the monitor at the bottleneck asks the Grid Resource Manager to find a new key path for it. After generating a new deployment configuration and notifying the Deployer to initiate the new GATES Grid Services, the Grid Resource Manager sends the new deployment configuration to all the impacted monitors. These monitors then instruct their grid instances to migrate to the new locations.

Two significant observations need to be made from the design we have presented here. First, instead of integrating the monitoring function into the GATES grid service, we have implemented them separately. We believe that this significantly reduces the performance overhead due to the monitoring, since the grid services can now focus on application execution only. Second, push (notification) approach is applied for the communication between the grid services and the monitoring services. Instead, if the pull (query) approach were to be used, the monitoring services’ continuous queries about the grid services’ buffer status could have resulted in high overheads.

The next two subsections explain how the Grid Resource Manager applies the dynamic resource allocation algorithm to generate a new deployment configuration, and the migration procedure.
Figure 5.3: An Example Pipelined Application

Figure 5.4: The Queuing Network Model of the Pipelined Application
5.2.2 Dynamic Resource Allocation Scheme

There are two major components to our dynamic resource allocation scheme. The first is an approach for determining the bottleneck in a pipeline. The second is our method for determining alternative path(s).

Applications are executed on our middleware in a pipelined fashion. Intuitively, to achieve the best performance from a pipeline, the pipeline needs to be balanced and no single stage should be allowed to become a significant bottleneck. Therefore, accurately determining the existing bottleneck is a prerequisite to any effective dynamic resource allocation scheme. Our overall strategy is to model the system as a queuing network and decide the bottleneck by observing each queue’s load. Specifically, to get a queuing network model, we can model every stage as a server and view the input buffer of a stage as the queue of the server. Similarly, we can also model the network connecting two consecutive stages as a server and view the output buffer of the first stage as the queue of this server. As an example, the Figure 5.3 shows a pipelined application and its queuing model is shown in Figure 5.4.

Such a queuing network comprises of several paths or pipelines. Each path in the queuing network could have a bottleneck. We just focus on a single path. We believe that a server is the bottleneck in a path if 1) the server’s queue is the currently overloaded, and 2) it is the last overloaded queue in the path. The reason for the second condition is that we prefer to identify at most one bottleneck, and given multiple overloaded servers, we prefer to relocate the last overloaded one first. If an input buffer of a stage instance is the last overloaded buffer, we conclude that the bottleneck is the stage instance and the CPU resources are scarce. In such a case, we need to find a new node with adequate CPU cycles, which we assume can be
done within the same cluster. Instead, if the output buffer of a stage instance is the
last overloaded buffer, then the bottleneck is the network connection. Therefore, a
network with higher bandwidth is desired.

We determine a new path as follows. We use the static algorithm to calculate
a path from the node that is outputting data to the bottleneck network link, to
the destination. This avoids the need for migrating any of the stages prior to the
bottleneck. However, it is possible that no better alternative path may be available
from the node that is outputting data to the bottleneck network link. Therefore, we
check if the path computed by the static algorithm is identical to the one currently
being used. If this is the case, we recompute a new path from the predecessor of the
node that is outputting data to the bottleneck network link. This process can be
repeated till a new path is determined.

In our framework, the above scheme is implemented as follows. Every monitor is
responsible for monitoring the output buffer and input buffers of the corresponding
grid service. When a buffer is overloaded in a grid service, the service sends an
overload notification to its monitor. In turn, this monitor contacts the monitors for
the later stages in the path to check whether the buffer being overloaded is the last
overloaded buffer. If the monitor finds out the stage overloaded in the monitor’s
grid service is the bottleneck, it will inform the grid resource manager to allocate a
new node with more available CPU cycles in the same cluster. If the bottleneck is a
network connection, the monitor will ask the grid resource manager to search a new
path. The grid resource manager then applies the static algorithm to recompute a new
path as described above. If a different path is found, a new deployment configuration
will be created and the Deployer will be called to deploy the new GATES grid services.
Afterwards, the new deployment configuration is sent to all impacted monitors. They then instruct the corresponding grid services to migrate to the locations specified by the new deployment configuration. How grid services migrate will be explained in Section 6.1.

5.3 Experimental Evaluation

This section presents results from a number of experiments we conducted to evaluate our algorithm. Specifically, we had the following goals: 1) Show that the deployment configuration created and optimized by our static algorithm can be as good as the best one among a large number of choices manually enumerated, and 2) Demonstrate that the algorithm-created configuration can outperform most of a large number of possible configuration, 3) Examine the overhead introduced by dynamic scheme,
and 4) Show that the dynamic algorithm can further improve performance, compared with the static one, especially under heavy and variable network traffic.

One of the challenges in conducting our experiments was to have a setup where network bandwidths can vary significantly and topology can be quite complex, and yet, repeatable and reliable experiments could be conducted. In a LAN environment or within a single organization, the bandwidths are unlikely to vary much, and the topology is usually very simple. In such scenarios, the resource allocation problem usually becomes quite trivial. At the same time, a WAN environment does not allow repeatable experiments. Therefore, to conduct our experiments, we set up an environment in which network bandwidths are simulated by inserting delays between packages and network topologies are created randomly. This allowed us to focus on our goal, which was to demonstrate that our algorithm is effective when network bandwidths can vary significantly and topology is quite complex.

The experiments we report were conducted using the counting samples application. In short, a data stream comprises a set of integers. We are interested in determining the $n$ most frequently occurring values and their number of occurrences at any given point in distributed streams. The detailed explanation of the application can be referred to Section 3.3.1.

5.3.1 Evaluating Static Allocation

For evaluating our static allocation scheme, we created the following three versions. A deployment configuration could be determined manually by comparing a large number of possible configurations. We call this configuration manual-config. The configuration automatically generated by the algorithm is called auto-config.
Figure 5.6: Comparing auto-config, manual-config, and opt-config
The configuration where we further apply the optimization of removing unnecessary transporters is called opt-config.

Based on these three configurations, we conducted three sets of experiments to evaluate the static allocation scheme, which are detailed below.

**Experiment 1:** Our first experiment demonstrated auto-config and opt-config are almost as good as a manual-config, which is based on enumerating a significant fraction of all possible choices. We had 4 data sources with fixed locations. We further made 6 clusters available to run the intermediate stage of the application. The results are shown in Figure 5.6. We considered two different cases, corresponding to 700,000 and 2,000,000 integers being produced by each data source. When each data source created 700,000 integers, the application using the manual-config is 5.3% faster than that using the auto-config and 3.6% faster than that using opt-config. In the second case, we see that the manual-config is just 1.4% faster than auto-config and 0.8% faster than opt-config.

Thus, the results indicate that the performances in the above three scenarios are very close, i.e., our algorithm is effective. Furthermore, the larger the size of the datasets, the differences are smaller.

**Experiment 2:** The second experiment was conducted in the same environment as the first one. We randomly selected 120 out of 1296 configurations that were possible and compared them with the opt-config. We did not carry out the extensive evaluation, as the total number of choices was very large. As shown in Figure 5.7, opt-config outperforms all but one of the randomly selected configurations. The one configuration that performs better than the optimized one is just 2% faster.
Figure 5.7: Comparing \texttt{auto-config} with 120 Other Configurations
all of the other configurations resulted in a slow-down by at least a factor of 2, and in many cases, up to a factor of 3. This again shows that our algorithm is effective.

**Experiment 3:** In the environment we had considered for the first two experiments, the total number of possible deployment configurations was very large. This did not allow us to perform exhaustive comparisons. Therefore, we used a different environment, in which the number of data sources was 3 and the number of available clusters was 4. The network topology was randomly created. The number of possible deployment configurations was now 64, which allowed exhaustive comparisons. We
compared the algorithm generated and optimized configuration, opt-config, against all 64 configurations.

The results are shown in Figure 5.8. opt-config outperforms all possible configurations, and in most cases, by at least a factor of 2. This confirms our observations from the previous 2 experiments.

### 5.3.2 Evaluation of the Dynamic Allocation Scheme

The following environment was used to conduct these experiments. We setup 3 data sources with fixed locations. We used 18 clusters. Eight of them were used to run the application statically and other 10 clusters were available for the application’s stages to migrate. To better evaluate the dynamic allocation scheme, we
simulate network utilization. If the network utilization is 20%, then only 80% of the network bandwidth is available to transmit the application’s data. Therefore, by continuously varying networks’ utilization, we can simulate a dynamic environment. More specifically, to simulate the dynamic network environment, we randomly choose a new utilization to replace the old one with some frequency, based on a probabilistic distribution. We make the probability larger and select larger utilization when we simulate a network having heavy traffic.

**Experiment 4:** In this experiment, we demonstrate how effectively our dynamic scheme works. We consider the situation where one or two network links being used by the application suddenly experience heavy traffic. Such a situation was simulated
by changing one network connection’s utilization to a high value, i.e. 60%, during the execution of the application. We evaluate the following three versions: 1) the static allocation version, which corresponds to opt-config described in the previous subsection, 2) the dynamic version, which is initiated with opt-config, but our dynamic reallocation is used, and 3) another version that we refer to as Immediate Reallocation. This version was created as follows. We again initiated execution with opt-config, but dynamic monitoring and reallocation was turned off. When the network bandwidth changed, we immediately switch to a new precomputed configuration, which corresponds to the use of our static algorithm on the new network topology.

The results are presented in Figure 5.9. The results indicate the dynamic version does not have a noticeable difference from the immediate reallocation version. The static version is about 80% slower than either of them. The performance of the static version shows that reallocation was crucial for good performance in this case. Because dynamic version has the same performance as that of immediate reallocation, we can see that our dynamic scheme is not only responsive to the changes in the execution environment, it also does not introduce any measurable overheads.

**Experiment 5:** Our last experiment focused on addressing the following question: if the network bandwidths change frequently, can our dynamic scheme outperform our static scheme? We compared the dynamic and the static schemes under varying environment, in which network links and their utilizations were randomly selected and changed. We considered 4 different cases. In the first three cases, 1000000, 2000000, 4000000 integers were produced by each data source. In the fourth case, 4000000 integers were also produced, but the network traffic was heavy. The number of times
each network utilization was changed was 4, 4, 4, and 20, respectively, for these four cases. The results are presented in Figure 5.10. The performance improvements from the dynamic scheme are 10%, 8%, 12% and 13%, respectively.

5.4 Summary

This chapter has investigated resource allocation for applications that involve processing of distributed data streams. We have presented a static resource allocation algorithm and introduced an initial version of dynamic resource allocation scheme.

We have experimentally evaluated our static and dynamic resource allocation techniques. Our results show the following: 1) our static algorithm generates configurations that are very close to optimal, and significantly better than most other possible configurations, 2) the overhead of using dynamic allocation and migration are quite small, 3) in environments where bandwidth availability changes frequently, our dynamic scheme performs better than the static scheme.
CHAPTER 6

DYNAMIC MIGRATION

One important characteristic of a grid environment is that availability of resources can vary very significantly over time. Execution and scheduling of applications in a grid environment must take such resource variability into account.

Most of the earlier work on grid computing focused on bag of tasks or the master-worker class of applications [24, 4, 23]. In scheduling the tasks associated with these applications, it is relatively simple to consider resource variability. However, in recent years, there has been a growing trend towards supporting more tightly coupled applications. Examples of such applications classes include scientific workflows [5, 44], applications that use pipelined or data-flow like processing [14], and streaming applications [33, 97, 98].

Dynamic reallocation of resources is even more important for these applications, because of two major reasons. The first is the long-running nature of these applications. The second is that these applications often require large volumes of data transfer between processing stages, and besides variability in availability of CPU cycles and memory, changes in network bandwidth can impact their execution very significantly. At the same time, implementing dynamical resource allocation is harder for these applications. This is because significant amount of state can be associated
with each processing stage, and allocation of resource for each stage cannot be done independently.

This chapter considers the problem of enabling GATES to support efficient dynamic migration for tightly-coupled and pipelined applications in a grid environment. We provide an alternative to basic checkpointing [84]. We use the notion of Lightweight Summary Structure (LSS) to enable efficient migration. The idea behind LSS is that at certain points during the execution of a processing stage, the state of the program can be summarized by a small amount of memory. This allows us to perform low-cost process migration, as long as such memory can be identified by an application developer, and migration is performed only at these points. The overall contributions of this chapter are as follows.

- We have proposed the notion of LSS, and shown how it can enable efficient process migration.
- We have demonstrated an implementation of process migration using LSS in the context of the GATES middleware.

We have extensively evaluated our implementation using three stream data processing applications. The main observations from our experiments are as follows. First, the use of LSS reduces the size of process state by a factor of 30-120, and enables efficient process migration. Second, the use of LSS and migration interface introduces a very small overhead for GATES applications. Third, we show that dynamic process migration can significantly improve the performance of long-running applications. Finally, we also show that our process migration implementation does not impact the accuracy of the processing.
The rest of this chapter is organized as follows. In Section 6.1, we discuss the notion of LSS and implementation of LSS-based process migration in GATES. In Section 6.2, we describe the applications we have implemented, and how LSS is used for each of these applications. Our experimental evaluation is presented in Section 6.3. We give a summarization in Section 6.4.

6.1 Implementing Dynamic Migration using LSS

This section describes our LSS based approach for supporting dynamic migration in streaming applications.

6.1.1 Motivation

Availability of resources in a grid environment usually varies dramatically. Therefore, dynamically allocating new grid resources and migrating applications from resources with high utilization to new resources could improve performance significantly. In particular, long running applications, such as applications that process streaming data over a long period of time, critically need dynamic migration. This is not only because bandwidth and/or CPU availability for the resources they are using can change, but new resources can also become available over time.

To support dynamic migration, checkpointing is usually used. Specifically, a snapshot of system’s runtime state, including processes’ execution points, memory stacks (pages), and CPU status, are taken and a checkpoint is created. This checkpoint is then transmitted to a new node, where the original execution environment is restored, and processes are restarted at the points when the checkpoint was taken.

Applying such a methodology in a grid environment poses several challenges. First, checkpoints are usually platform-dependent, i.e., they include information such
as CPU status and memory image, which are dependent on the hardware and operating system. A grid comprises heterogeneous resources, and grid standards have been designed to support applications that are independent of hardware and operating system. Thus, using basic checkpointing to migrate applications is not practical in a grid environment.

Furthermore, large-volume checkpoints can result in inefficient migration, especially for data stream applications. The size of a checkpoint could be considerable because it includes image of memory segments used by the application. Data stream applications process very large amounts of data, and this processing typically is done in memory. Thus, the size of the required memory is quite large, which results in large-sized checkpoints. Taking and transferring these large-volume checkpoints and restoring the original execution environment in a wide-area setting can incur significant overheads. The impact of this is even more severe on applications that process data streams, as they need to meet real-time constraint on processing of data that is arriving continuously.

### 6.1.2 Light-Weight Summary Structure

We now describe our approach for supporting migration, which is based on the notion of a *Light-Weight Summary Structure* (LSS).

The design of LSS is based on the observation that for many application classes, including data stream processing and other pipeline/data-flow like systems [14], the processing structure is as follows:

```java
... while(true) {
    read_data_from_streams();
}
```
During each loop, a number of data items from the stream are read and processed. The processing results are accumulated to form a *summary information*. We name the data structure storing such summary information as the *Light-Weight Summary Structure (LSS)*. Other data structures used by the application are considered *Auxiliary Structures*. Memory locations in auxiliary structures are always reset to initial values at the end of each loop.

Two additional observations are important with respect to LSS. First, for most applications, the size of LSS is much smaller than that of auxiliary structures, and therefore, also much smaller than the total memory used by the application. Second, since auxiliary structures are anyways reset at the end of each loop iteration, they do not need to be migrated, if the migration is occurring at the end of a loop iteration.

Based on these observations, LSS can be used for supporting dynamic migration. The middleware can provide functions to specify a block of memory to be LSS. When a migration operation is needed, only LSS is automatically transmitted to a new node. The middleware will also be responsible for restoring the LSS at the new node. As noted earlier, migration can only be supported at the end of each loop iteration. After migration, applications can resume from the start, instead of the specific execution point at which LSS is migrated.

The migration supported in this fashion is distinct from migration using normal checkpointing in the following ways:

- As noted earlier, migration procedure becomes more efficient. Only LSS, a much smaller portion of the overall memory needed by the applications, is migrated.
• Despite achieving efficiency, we do not negatively impact the accuracy of processing.

• An application developer’s effort in making the application capable of migration is also quite small. They only need to identify which variables belong to LSS, when implementing a processing stage. Other steps are taken care of by the middleware, as we will explain later.

• LSS is a logical data-structure and its contents are quite independent of the specific platform. We do not need program contexts to migrate, and therefore, it is easier to support migration across heterogeneous platforms.

6.1.3 Middleware Implementation

We now present details of how LSS is used in GATES for supporting migration.

The pseudo-code in Figure 6.1 shows how an application utilizes migration API functions. Before calling any migration API functions, an application needs to implement an LSS class that declares the summary structure as its member. In the application shown here, the LSS class is counting-lss.

We can use getLSS(Name-of-LSS-Class) to specify a LSS class to GATES. GATES will return an instance of the LSS class, which is either created locally, or cloned from the LSS instance at a remote node if the application migrates from that node. At the end of each loop, ifMigrationNeeded() is invoked to inquire GATES whether the condition to migrate is met. If migration is needed, the migrate(Instance-of-LSS-Class) function is called, in which the LSS instance is cloned to a new node. After the execution of the migrate() function, the application at the current node ends.

GATES implementation of the migration procedure is explained below.
1. //Initialize auxiliary structures
   initialize_auxiliary_structures();

2. //Get an LSS instance from GATES
   counting-lss lss = GATES.getLSS("counting-lss");

3. //Process streaming data
   while(true)
   {
     read_data_from_streams();
     process_data();
     accumulate_intermediate_results_to_LSS(lss);
     initialize_auxiliary_structures();
     //check if migration is needed
     if(GATES.ifMigrationNeeded())
     {
       GATES.migrate(lss);
       break;
     }
   }
Figure 6.2: Migration Procedure
**Migration Procedure:** Figure 6.2 shows the overall procedure. We assume that A, B and C are nodes where three pipelined stages of an application built on GATES are executing. The second stage, originally at B, is being migrated to a new node, B'. The procedure is triggered when B invokes the function \textit{migrate()}. The migration procedure comprises 8 steps, as we will describe now. These steps occur within GATES and are hidden from the application.

First, a new path from A to B' to C is created. This step involves launching a GATES grid service at B', establishing socket connections between A, B, B' and C, and initializing internal buffers for these socket connections. Second, we stop sending data from output buffers of A and B. At the third step, C is notified to move data which is in the \textit{obsolete} input buffer to the new one. Even though no data is sent out as the result of the second step, there might be some data items still residing in system's buffers or being transmitted in networks. Therefore, the third step does not end unless all data in system buffers and networks passed onto the obsolete buffer.

The fourth step involves copying data from the output buffer at B to the one at B', using a socket connection established between B and B' earlier. In the fifth step, the LSS instance is serialized at B and sent in the form of a byte stream to B', where a cloned LSS instance is deserialized from the byte stream. Thus, whenever \texttt{getLSS()} is invoked by the application at B', the cloned LSS instance is returned. The step 6 is similar to the step 4: we wait until all data in socket buffers and networks empty into the input buffer at B, and then copy them to the input buffer at B'. In the step 7, we load the corresponding application code into B' and initiate its execution. Finally, as the application at B' is processing data, A is notified to move data within
the obsolete output buffer to the new one. In the scenarios where B is connected to multiple upstream stages, the above procedure can still be applied.

We now focus on when and where a processing stage is migrated. The answers to the when and the where can be referred to the framework for resource monitoring and dynamic allocation that we have introduced in Section 5.2.1 and the dynamic resource allocation scheme in Section 5.2.2.

6.2 Streaming Applications for Dynamic Migration

This section describes three applications that we have supported so far using GATES migration services.

count-samps, introduced in Section 3.3.1, is a distributed version of the counting samples problem. A data stream comprises a set of integers. We are interested in determining the $n$ most frequently occurring values and their number of occurrences at any given point in the stream. Since it is not possible to store all values, a summary structure must be maintained to determine the frequently occurring values.

The problem we consider is of determining frequently occurring values from distributed streams. Our solution is to store $m$ frequently occurring values from each stream at a node close to the data source, and then merge them to determine the overall $n$ most frequently occurring values at a central location. The value of $m$ can be chosen to provide a tradeoff between the accuracy of the final results and the efficiency of processing. These $m$ values form the LSS at each remote node.

The second application is clustering evolving data streams [7], and is referred to as CluStream. Clustering involves grouping similar objects or data points from a given set into clusters. The particular problem considered here is clustering data arriving
in continuous streams, especially as the distribution of data can change over time. The detailed explanations of this application can be referred to Section 4.2

This clustering algorithm can be easily implemented using the GATES middleware. Figure 4.2 shows the three stages that are used. The first stage is simply the data source, which sends streaming data to the second stage. The second stage computes micro-clusters. After a certain number of data points have been processed, it sends the computed micro-clusters to the third stage. The third and the final stage then apply the modified $k$-means algorithm [7] to create and output the final clusters.

To make the second stage capable of migration, we store micro-clusters as an LSS object and implement this stage as indicated in Figure 6.1.

The third application we have studied finds frequent occurring itemsets in distributed data streams and is referred to as Dist-Freq-Counting [82]. It has been described in Section 4.2 and the structure of this application is presented in Figure 4.3. Each monitor node $M_i$ counts the frequencies of itemsets appearing in the stream $S_i$, and periodically sends this information to its parent node, which could be an intermediate node or the root node. Intermediate nodes combine the frequency information received from their children and pass them up to their parent node. Finally, the root node outputs the itemsets whose frequencies exceed the specified support threshold $\tau$.

To make monitors migratable, we have implemented a LSS class to store the unprocessed itemsets. This is based on the following observation: a monitor’s migration could happen when the monitor still accumulates itemsets. These accumulated itemsets, which have not yet been processed, need to be transmitted to the remote node. Therefore, we choose these itemsets as the summary information.
<table>
<thead>
<tr>
<th>Number of Micro-Clusters</th>
<th>Size of LSS (KB)</th>
<th>Size of Clustream (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7</td>
<td>1,067</td>
</tr>
<tr>
<td>20</td>
<td>13</td>
<td>1,143</td>
</tr>
<tr>
<td>40</td>
<td>25</td>
<td>1,437</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>1,673</td>
</tr>
<tr>
<td>100</td>
<td>62</td>
<td>2,382</td>
</tr>
</tbody>
</table>

Figure 6.3: Memory Usage of LSS for Clustream

6.3 Experimental Evaluation

This section presents results from a number of experiments we conducted to evaluate the notion of Light-Weight Summary Structure and our implementation of dynamic resource allocation and process migration. Specifically, we had the following goals in our experiments:

- Demonstrate that LSS uses a small amount of memory for our target applications, compared with the total memory usage of the applications and the middleware.

- Show that dynamic migration with LSS is efficient and does not impact overall system performance.

- Show that applications benefit significantly from the dynamic migration when execution environments change dynamically.

- Show that accuracy of processing is not impacted by the migration using LSS.


<table>
<thead>
<tr>
<th>Time Periods</th>
<th>Number of Transactions</th>
<th>Size of LSS (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>802</td>
<td>177</td>
</tr>
<tr>
<td>2</td>
<td>1557</td>
<td>305</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
<td>226</td>
</tr>
<tr>
<td>4</td>
<td>726</td>
<td>159</td>
</tr>
<tr>
<td>5</td>
<td>2437</td>
<td>464</td>
</tr>
<tr>
<td>6</td>
<td>740</td>
<td>161</td>
</tr>
<tr>
<td>Average:</td>
<td>1227</td>
<td>238</td>
</tr>
</tbody>
</table>

Figure 6.4: Memory Usage of the LSS for Dist-Freq-Counting

### 6.3.1 Experiment Setup and Data Sets

For distributed processing of streaming data in a grid environment, we need high bandwidth networks. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single Linux cluster. The cluster consists of 24 computing nodes. Each node has a Pentium III 933MHz CPU with 512MB of main memory and 300GB local disk space, interconnected with switched 100 Mb/s Ethernet.

The experiments were conducted using the three streaming data mining applications described earlier in this chapter. For the count-samps application, integer streams were generated by a simulator. For the CluStream application, we used the KDD-CUP'99 Network Intrusion Detection dataset. For Dist-Freq-Counting, we used a dataset generated by the IBM synthetic data generator [8]. We conducted 4 sets of experiments which we describe in the rest of this section. The first three were conducted by using both Clustream and Dist-Freq-Counting, and count-samps was used for the last experiment.
6.3.2 Memory Usage with LSS

This experiment demonstrates that compared to the entire application and the middleware, LSS uses a small fraction of memory.

GATES and its applications are implemented in Java. Unfortunately, Java does not provide any mechanism similar to C/C++’s `sizeof()` function to measure object sizes, making it difficult to get the exact size of an object in Java. We instead used two techniques to approximately estimate memory usage. We applied the technique described in [111] to measure the size of a GATES service and the application. To measure the size of an LSS object, we serialized the object into a file and then measured the file size.

For Clustream, LSS is the set of micro-clusters, the size of which depends on the number of micro-clusters. Therefore, we varied the number of micro-clusters and then measured memory usage of LSS and the entire application. The results are shown in Figure 6.3. When the number of micro-clusters were set to 10, 20, 40, 80 and 100, the LSS only occupied approximately 0.7%, 1.1%, 1.2%, 2.9% and 2.6%, respectively, of memory used by the entire application.

We further examined the memory usage for Clustream, when the number of micro-clusters was 100. The memory use by the middleware, the application, and LSS are 4,350KB, 2,382KB, and 62KB, respectively. Thus, LSS just used approximately 0.9% of the total memory consumed by GATES and the application. This clearly points to the efficiency of using LSS for checkpointing and migration.

We repeated the above experiment using Dist-Freq-Counting. Its LSS is the set of unprocessed transactions, and its size is proportional to the number of such transactions. This, in turn, depends on when migration occurs. Therefore, we migrated
the application at six random time instances, and measured the LSS’s memory usage and the corresponding number of unprocessed transactions. The results are indicated in Figure 6.4. We also measured the average size of Dist-Freq-Counting, which is 16,422KB. Thus, the LSS only used on average 1.1% of the total memory consumed by the middleware and the application.

### 6.3.3 Efficient Migration

We conducted two groups of experiments to show that migration using LSS is efficient.
First, we measured the time Clustream takes to migrate, given different dataset sizes. We compared the migration time with the application execution time. Irrespective of the dataset size, migration occurs only once. The results are shown in Table 6.1. As we would expect, migration time is not impacted by the dataset size. The migration time only accounts for 3% of the execution time when Clustream is executed with a dataset of size 3,200 KB. Therefore, for long running streaming applications, the time spent on migration is very small.

Second, we compared the performance of three different executions of Clustream. The first execution involves a version of the application that does not invoke GATES’s migration interfaces, and is referred to as the Without LSS version. The second and third execution involve a version that takes advantage of GATES’s migration support
and has the ability to migrate. In the second execution, no migration actually occurs, while in the third execution, one stage migrates once. The second and the third executions are referred to as non-migration and migration executions, respectively.

We varied dataset sizes to 200KB, 400KB, 800KB, 1,600KB and 3,200KB and compared the execution time of these three executions. The results are indicated in Figure 6.5. Due to the overheads of invoking GATES’s migration interface, the version with LSS is slightly slower than the version without LSS. Similarly, the execution with migration takes additional time as compared to an execution that does not migrate. However, the difference between the two versions and two executions is quite small, which shows that overheads of supporting a migration interface and
migrating during execution are quite small. Moreover, as shown in Figure 6.6, the time differences between executions with and without migration are always smaller than the corresponding migration time. This is because to reduce migration overheads, we carry out the step 7 of the migration procedure (Figure 6.2) with processing data in a new node.

Using *Dist-Freq-Counting*, we repeated the above experiments and obtained very similar results. They are presented in Table 6.2, Figure 6.7, and Figure 6.8, respectively.

Figure 6.7: Execution Time of Three Version: *Dist-Freq-Counting*
Figure 6.8: Migration Time and Difference of Migration and Non-Migration Execution Time: Dist-Freq-Counting
6.3.4 Benefits of Migration in a Dynamic Environment

In this subsection, we show that an application can benefit significantly from taking advantage of the GATES’s dynamic migration support in a dynamic environment, where CPU cycles and network bandwidths can vary. For our experiments, we only considered variation in network bandwidth.

We first considered Clustream. We varied the bandwidth of the network between a data source and an intermediate stage of Clustream. The migration version moved the intermediate stage as early as possible to a new node which continued to have a high-speed connection to the data source. The non-migration version, in comparison, stayed at the original nodes. We varied the network bandwidths from 1Mbps, 100Kbps, 10Kbps, to 1Kbps, then observed the execution time of these two versions.
As shown in Figure 6.9, though two versions’ execution time is close in situations where the bandwidths are 1Mbps and 100Kbps, the migration version is 4 and 33 times faster, respectively, than the non-migration version when the bandwidths are 10Kbps and 1Kbps. Figure 6.10 considered Dist-Freq-Counting and the results are very similar.

### 6.3.5 Processing Accuracy and LSS Migration

We now investigate how migration using LSS impacts accuracy of data stream processing. The *count-samps* application was used to conduct this experiment. As it is an approximate algorithm, two executions may not produce the same results. The methodology we followed for our evaluation was as follows.
We synthetically generated an integer stream in which 50% numbers are 1, 25% numbers are 2, 12.5% numbers are 3 and so forth. Thus, the top 10 frequently occurring numbers in the stream are 1, 2, 3, \ldots, 10. To compare accuracies of various counting results, we designed two criteria to quantify an output’s accuracy.

The first criterion considers how many numbers are correctly chosen. The ideal result is that numbers 1, 2, 3, \ldots, 10 are picked, regardless of their number of occurrences. Then the accuracy of the ideal result is 10, according to the first criterion. Similarly, the accuracy of the worst result, where no number in [1, 10] is picked, is 0.

We conducted 5 rounds of experiments for each migration version and non-migration version. We calculated average accuracy of 5 results’ for each version, and it turned out the average accuracy for both versions were identical, i.e. 8.8. Note that even the sequential algorithm for counting samples is approximate, and will not be completely accurate.

The first criterion does not consider how close a number’s occurrence frequency in a counting result is to its true occurrence frequency in the stream. To overcome the first criterion shortcoming, we designed the second criterion as shown below. Let \( R \) be the set of 10 most frequently occurring values determined by the algorithm. Then, we denote \( R_1 \) as

\[
R_1 = R \cup \{1, 2, \ldots, 10\}
\]

Let \( t_i \) denote how frequently a value occurs in the stream, and let \( T_i \) be the frequency reported by the count-samp application. Then, we can compute the accuracy of results, \( A \), as:

\[
S = \sum_{i \in R_1} t_i
\]
The idea of the criterion is to compare the counted frequency of each chosen number with its true frequency. The difference of two frequencies reflects how accurate a chosen numbers’ counted frequency is. The sum of these differences can reflect how accurately a set of number are chosen and counted.

We recalculated average accuracy of 5 results’ for both the migration and the non-migration versions, and they are 0.02725 and 0.02493, respectively, which shows that the results are very close. Thus, we can see that migration using LSS does not impact accuracy of processing.

### 6.4 Summary

This chapter has considered the problem of supporting and efficiently implementing dynamic resource allocation for tightly-coupled and pipelined applications in a grid environment, specifically for a GATES approach. We provide an alternative to basic checkpointing, using the notion of *Light-weight Summary Structure* (LSS), to enable efficient migration. The idea behind LSS is that at certain points during the execution of a processing stage, the state of the program can be summarized by a small amount of memory. This allowed us to perform low-cost process migration, as long as such memory can be identified by an application developer, and migration is performed only at these points.

We have extensively evaluated our implementation using three stream data processing applications. The main observations from our experiments are as follows. First, the use of LSS reduces the size of process state by a factor of 30-120, and
enables efficient process migration. Second, the use of LSS and migration interface introduces a very small overhead for GATES applications. Third, we show that dynamic process migration can significantly improve the performance of long-running applications. Finally, we also show that our process migration implementation does not impact the accuracy of the processing.
CHAPTER 7

AN ADAPTIVE IMAGE RENDERING APPLICATION IMPLEMENTED USING GATES

This chapter describes how we have used GATES to develop a volume rendering application. The goal of volume rendering is to create a 2D projection of a 3D data set (volume data). Although the speed of volume rendering has significantly increased in the past several years, the size of an average volumetric data set also continues to grow. To address the challenge of rendering large scale data sets, researchers have proposed various algorithms. Among the existing techniques, hierarchical rendering algorithms can effectively control the tradeoff between quality and speed, and thus show a great potential.

A more challenging problem considered here is that volume data arrive in continuous streams. To observe and analyze significant features of these volume data, scientists usually need to interactively visualize them in real-time. An example of such applications is rendering tissue volumes obtained from clinic instruments in real-time to aid a surgery. However, visualization operations are computationally intensive. As a result, computational resources could be easily consumed by these operations and real-time rendering cannot be guaranteed.
We believe Grid techniques would be a good solution to overcome the challenges. A typical visualization operation can be divided into a sequence of stages organized in a pipeline fashion. In a grid environment, these stages can be simultaneously executed in distributed computing nodes. To assure a real-time performance for the parallel executions, we should deploy the stages that involve data-intensive computation to the nodes that are close to the data sources. Furthermore, to ensure quick responses while still maintain good image qualities, volume rendering algorithms running in Grid need to be highly adaptive. Currently, visualization researchers have developed Level Of Detail (LOD) selection algorithm [17, 54, 78] running on a single computer to achieve adaptivity when the computation of visualization is too intensive or the volume of data is too large. These algorithms determine LODs according to some algorithm-specific parameters such as the user’s tolerance to visualization errors or the size and distance of the objects to the viewer. To balance visualization quality and speed, the users need to manually adjust these adaptation parameters by iterative trial-and-error. However, such manual parameter adaptation is very challenging when the applications are running on a grid, since the interaction between the multiply stages of the visualization pipeline on a grid is much more complex compared to a single processor execution environment. Furthermore, the available network bandwidth and computation resources on the grid could be highly changeable. Thus, the relationship between the speed of the visualization and the LOD selection parameters is hard to predict. Therefore, it is extremely desirable that the adaptation in visualizing volume could be automatically provided by a middleware with simple and well-defined APIs, in contrast to being implemented specifically for every application.
Our GATES middleware perfectly meets these needs. GATES can automatically allocate Grid resources for the rendering application, dynamically migrate it to new resources to adapt to varying Grid environments, and, most importantly, provide the self-adaptation functionality to the application to ensure real-time image rendering. Therefore, we divide a rendering process into two consecutive steps, and build these steps on the GATES middleware. We will illustrate such design in detail in the next section and evaluate the application in Section 7.2.

7.1 Overall Design

7.1.1 Two-Step Rendering Algorithm

As stated above, our rendering algorithm is composed of two steps.

The first step is to build a hierarchical volume structure called octree. In general, an octree is a multiresolution representation for a volume. We build such octrees using an algorithm described as follows.

Initially, the entire volume space is subdivided into smaller blocks, hereafter called subvolumes. For each subvolume, we create different levels of detail by repeatedly filtering the voxel data in the subvolume. Starting from the raw data, we average every $2 \times 2 \times 2$ voxels to create a lower resolution subvolume. We continue this filtering process until a predefined minimum resolution for the subvolume is reached. To avoid seams between adjacent subvolumes of different resolutions, we adopt the method proposed by Weiler et al.[121], which copies data points on the boundaries from low resolution to high resolution volumes to ensure a smooth transition when interpolation is performed. More details about the data filtering and seam prevention can be found in [121].
Our first step slightly differs from the method proposed by Weiler et al.\cite{121} in that the subvolumes in our case have different sizes. Instead of subdividing the volume uniformly into subvolumes of equal size, we take into account the volume’s spatial coherence when performing the subdivision. In regions where data values are more coherent, we merge the voxels together to form a larger subvolume. On the other hand, if the data values in a region have higher variations, we split the region into smaller subvolumes. There are two primary reasons for us to use subvolumes of different sizes. First, breaking a volume into subvolumes creates overhead since more slicing planes are needed when rendering the volume. Regions that have high spatial coherence can usually be rendered at lower resolutions uniformly so breaking into smaller ones is unnecessary. On the other hand, if the volume contains values that are less coherent, breaking the volume into smaller subvolumes allows us to use higher resolution data in certain local regions while using low resolution data elsewhere.

**The second step** is to use octrees and volume data to render images. Traversing an octree enables a volume to be rendered in different resolutions. The higher resolution, the better image quality while the slower rendering speed. Therefore, given an error tolerance, which is a user-defined resolution, traversal of an octree can trade image quality for a fast rendering speed.

We have created two dynamic libraries that respectively implement the steps. One denoted by liboctree.so is used to generate an octree from a volume data. The other, librendering.so, is to render images from the volume data and its octree. Both of them are written in C++.
7.1.2 Implementing the Two-Step Rendering Algorithm on GATES

This two-step rendering algorithm can be easily implemented upon the GATES middleware. Figure 7.1 shows the application’s communication topology where four stages are used. The first stage is simply the data source, which multicasts streaming volume data to the second and third stages. The second stage computes octrees and sends them to the third stage. After having had both volume data and corresponding octree, the third stage renders images and sends them to the final stage where images are stored and displayed.

We have built these stages on the GATES middleware and implemented them in Java. The implementations of the first and fourth stages are straightforward. Here, we focus on the implementation details of two stages in the middle. Figure 7.2 and 7.3 respectively present their pseudo codes.

Let’s first focus on the second stage. As can be seen, instead of re-implementing the function of generating octrees in Java, we load the liboctree.so library from Java via the JNI (Java Native Interface) and then invoke the native function to build an
public class GeneratingOctree implements StreamProcessor
{
    ...
    public void work(InputBufArray in, OutputBufArray out)
    {
        ...
        native = System.loadLibrary("liboctree.so");
        //Process data
        while(true)
        {
            octree = native.buildOctree(volumeData);
            out.sendToNextStage(octree);
        }
    }
}

Figure 7.2: Pseudo Codes for the Second Stage of the Application

octree. By taking advantage of these native codes, not only can we save unnecessary
costs of implementing the same function in multiple languages, but also we could
improve system performance by directly executing the native codes. The relationship
between the application, GATES, JNI and JVM is illustrated in Figure 7.4.

The implementation of the third stage is more complicated than that of the second
stage, because we need to consider adaptation parameters in this stage. There are
two adjustable parameters: error tolerance and image size. Image quality can be
controlled by choosing different values of error tolerance at the third stage. The
smaller values of error tolerance, the better image quality while the slower the process
of image rendering. Thus, we consider error tolerance the performance parameter.
The other adjustable parameter in the third stage is image size. Users might prefer
to a larger image. On the other hand, it takes more time to render it. Therefore, image
public class Rendering implements StreamProcessor {
    ...
    public void work(InputBufArray in, OutputBufArray out) {
        // Initialize error tolerance and image size
        double error_tolerance = 0.0;
        int imageSize = 128;

        // Either error tolerance or image size is the adaptation parameter
        // Specify it to GATES
        if(bOnlyApplyErrorTolerance == true)
            GATES.specifyPara(error_tolerance, 0.0, 0.1, 1);
        else if(bOnlyApplyImageSize == true)
            GATES.specifyPara(imageSize, 16, 512, 1);

        // Load the rendering library
        native = System.loadLibrary("librendering.so");

        // Process data
        while(true) {
            // Get octree from the second stage
            octree = in.getDataFromPrecedingStage();

            // Render an image
            image = native.Rendering(volumeData, octree,
                                      error_tolerance, ImageSize);

            // Send the image to the fourth stage
            out.sendToNextStage(image);

            // Either error tolerance or image size is the adaptation parameter
            // Get its suggested value from GATES
            if(bOnlyApplyErrorTolerance == true)
                error_tolerance = GATES.GetSuggestedValue();
            else if(bOnlyApplyImageSize == true)
                imageSize = GATES.GetSuggestedValue();
        }
    }
}

Figure 7.3: Pseudo Codes for the Third Stage of the Application
size is an accuracy parameter. As presented in Figure 7.3, we inform GATES of these existence of the two parameters by invoking GATES’s API function `specifyPara()`. Note that only one of the two parameters is specified to GATES, as GATES can only adjust one adaptation parameter. At the end of every iteration, a new value of error tolerance or image size is returned by the function `getSuggestedValue()`, and such new value is used for rendering in the next iteration.

This Java implementation described above is denoted by `Java-Imple`. To measure the overheads incurred by GATES and the Java language, we also implemented the rendering application in C++. This C++ implementation is denoted by `C-Imple`. C-Imple is a stand-alone version without the support of GATES and does not consider the self-adaptation functionality. Apart from these differences, it is same as Java-Imple.

## 7.2 Experimental Evaluation

This section presents results from a number of experiments we conducted to study the image rendering application with support of GATES and evaluate the self-adaptation algorithm in GATES. Specifically, we had the following goals in our experiments:

- Demonstrate that our self-adaptation algorithm is able to quickly converge to proper values of the adaptation parameters, for different data stream arrival rates.

- Show that the algorithm is not sensitive to initial values of the adaptation parameters.
Figure 7.4: Java Version Invokes the Native Libraries via JNI

- Show how efficient the GATES system is by comparing the performance of the rendering application implemented in Java and running on GATES with that of a stand-alone implementation in C++.

We conducted 3 sets of experiments within a single linux cluster. Each node in the cluster has a Pentium III 933MHz CPU with 512MB of main memory and 300GB local disk space, interconnected with switched 100 Mb/s Ethernet. The volume data used in experiments were generated from a fluid dynamic simulation which models the air flows around a particular wing. These experiments are described in the rest of this section.
Figure 7.5: Error Tolerance Converging Under Different Data Arrival Rates

7.2.1 Convergence under Different Data Arrival Rates

Our first experiment demonstrated that the self-adaptation algorithm can choose proper values for the adaptation parameters under different data arrival rates. As stated in 7.1, there are two adaptation parameters in this application: error tolerance and image size. Since GATES can only adjust one adaptation parameter, we first conducted experiments by fixing image size and only considering error tolerance the adaptation parameter.

We set image size to $128 \times 128$ and initialized error tolerance to 0.05. The allowed range of error tolerance was $[0, 0.1]$. We used one data source, and controlled the data arrival rate at the second stage to be 100, 150, 200, 250 and 300 Kbps, respectively. As present in Figure 7.5, the values of error tolerance are converged to 0.0023, 0.0094, 0.072, 0.10 and 0.10 respectively, for the five data arrival rates we considered. The
Figure 7.6: Images Rendered, Given Different Error Tolerances

X-axis in this chart is the number of rounds, which denotes the number of invocations of the rendering algorithm. One image is rendered and a new value of error tolerance is picked by GATES during each round. The convergence occurred in an average of 20 rounds, which corresponds to 130 seconds. Moreover, the images rendered at the moments when the convergences occur are presented in Figure 7.6.

A similar set of experiments were also conducted by fixing error tolerance to 0.075 and only considering image size the adaptation parameter. The results are shown in
Figure 7.7. We set the range of image size to be $[0, 512 \times 512]$, and initialized it to $128 \times 128$. We then considered 4 different data arrival rates at the second stage and the values of image size in each of the cases converge to $66 \times 66$, $86 \times 86$, $120 \times 120$ and $143 \times 143$ respectively. The convergence occurred in an average of 5 rounds, which corresponds to 34 seconds.

### 7.2.2 Convergence Irrespective of Different Initial Values

In this subsection, we show that the adaptation parameters can converge to the same values in the same execution environments irrespective of different initial values.

We first consider error tolerance the adaptation parameter. In environments where the data arrival rate was set to 100Kbps, we set the initial values of error tolerance to 0, 0.02, 0.05, and 0.08 respectively. As shown in Figure 7.8, error tolerance converged
Figure 7.8: Error Tolerance Converging to a Same Value Irrespective of Different Initial values (Data arrival Rate 100Kbps)

Figure 7.9: Error Tolerance Converging to a Same Value Irrespective of Different Initial values (Data arrival Rate 200Kbps)
to 0.0025 within 15 rounds in all cases. We reset the data arrival rate to 200kbps and repeated the experiments. The results are indicated in Figure 7.9: error tolerance eventually converged to 0.01 in each of the cases. We then consider image size the adaptation parameter and similar results, as shown in Figure 7.10 and Figure 7.11, have been gotten.

Therefore, our adaptation algorithm is not sensitive to initial values of adaptation parameter.

### 7.2.3 Performance Comparison of Two Implementations in C++ and Java

In this section, we compare the performance of the two implementations mentioned in Section 7.1. We first compare their overall performance. Each version fixed the
Figure 7.11: Image Size Converging to a Same Value Irrespective of Different Initial Values (Data Arrival Rate 200Kbps)

Figure 7.12: Comparing Different Implementations: Image Size is 256 x 256
values of image size and error tolerance to $256 \times 256$ and 0.0 respectively and executed rendering for 20 rounds. The execution times were recorded at the moments when 5, 10, 15, and 20 rounds were completed. Figure 7.12 shows the results: the execution time of both versions is linearly proportional to the number of rounds they have finished. Another observation from the figure is that C-Imple is approximately 2 times faster than Java-Imple.

In fact, a rendering task consists of two parts: actual rendering process and auxiliary operations. Auxiliary operations include allocating memory, copying data from one buffer to another and etc.. As for the actual rendering process, the rendering function in librendering.so is invoked directly by C-Imple to accomplish the process, while Java-Imple utilizes JNI to call the same function in order to render an image. We now compare these rendering processes in C++ and Java. We measured the total
time spent on the actual rendering process in the above experiment and the results are presented in Table 7.1. As can be seen, rendering contributed to 89% and 84% of the overall execution time respectively in C-Imple and Java-Imple. The rendering process in Java-Imple is 2 times slower than that in C-Imple. Therefore, in our case, overheads of invoking the native codes from Java close to the native codes’ actual running time. On the other hand, auxiliary operations in Java-Imple are 3.3 times slower than those in C-Imple.

We repeated these experiment by changing image size to $128 \times 128$. The results are similar, as shown in Figure 7.13 and Table 7.2.

<table>
<thead>
<tr>
<th></th>
<th>Rendering Process (seconds)</th>
<th>Auxiliary Operations (seconds)</th>
<th>Overall Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Imple</td>
<td>160.73</td>
<td>19.88</td>
<td>180.62</td>
</tr>
<tr>
<td>Java-Imple</td>
<td>335.54</td>
<td>65.66</td>
<td>401.20</td>
</tr>
</tbody>
</table>

Table 7.1: Execution Time Breaking Down When Image Size is $256 \times 256$ and Error Tolerance is 0.0

<table>
<thead>
<tr>
<th></th>
<th>Rendering Process (seconds)</th>
<th>Auxiliary Operations (seconds)</th>
<th>Overall Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Imple</td>
<td>47.44</td>
<td>2.01</td>
<td>49.45</td>
</tr>
<tr>
<td>Java-Imple</td>
<td>116.81</td>
<td>14.54</td>
<td>131.345</td>
</tr>
</tbody>
</table>

Table 7.2: Execution Time Breaking Down When Image Size is $128 \times 128$ and Error Tolerance is 0.0
7.3 Summary

In this chapter, we study an adaptive image rendering application and use it to re-examine our self-adaptation algorithm in GATES. Error tolerance and image size are two adaptation parameters in this application. The experimental results show that the self-adaptation algorithm can promptly converge to proper values of the adaptation parameters in different execution environments. The results also show that the algorithm is not sensitive to initial values of the adaptation parameters. We also compare the application performance of two implementations in Java and C++. By utilizing JNI to call native libraries, the Java version is slightly slower than the C++ version.
CHAPTER 8

CONCLUSION AND FUTURE WORK

This chapter summarizes the contributions of our thesis and presents directions for future research.

8.1 Contributions

With scientific instruments and experiments that continuously generate data, and increasing network speeds, we expect processing of distributed data streams to be an important application class for grid computing. We have taken an important step in this direction, by developing a middleware system called GATES. Further, we believe it is desirable for a grid middleware to support adaptive execution, especially, as the execution environment changes dynamically.

We contribute to the research area of supporting processing of distributed data streams in the following aspects.

- By using GT3.0, the middleware allows automatic resource discovery. GATES allows processing on distributed data streams to be specified as a pipeline of stages. By supporting an application container, the middleware enables easy deployment of distributed pipelined stages of an application.
- We have designed two algorithms to support adaptive execution for distributed data streams applications. In order to evaluate these algorithms, we have implemented various data stream applications using our middleware, and have extensively evaluated the adaptive capabilities of our middleware. The experimental results show the improved algorithm can quickly converge to proper values of the adaptation parameter in both static and dynamic environments.

- To automatically allocate Grid resources to pipelined stages, we have proposed static and dynamic resource allocation algorithms and implemented a framework to support dynamic resource allocation. We have experimentally evaluated these static and dynamic resource allocation techniques, and our results show they are effective.

- GATES supports efficient and low-cost dynamic migration for pipelined applications in a grid environment by using the notion of Light-weight Summary Structure (LSS). Our evaluation of LSS based process migration has shown that the use of LSS reduces the size of process state by a factor of 30-120, introduces a very small overhead for GATES applications, significantly improves the performance of long-running applications and hardly impact the accuracy of the processing.

8.2 Future Work

In this section, we discuss several directions for future research.
8.2.1 Improving Dynamic Resource Allocation Scheme

First, we will like to improve our running-time resource allocation scheme described in Section 5.2. The dynamic resource allocation algorithm currently applies the static algorithm to search new computing nodes when migration is needed. The static algorithm first constructs a weighted graph, in which the weight of an edge is the available bandwidth of the network connection represented by the edge. Then the Minimum Spanning Tree algorithm is applied to find new nodes [32]. Therefore, in this static algorithm, only network bandwidths are considered as the resource constraint to allocate new nodes. We need to develop a new metric that can consider both network bandwidths and CPU cycles as resource constraints.

8.2.2 Porting GATES from GT3.0 to GT4.0

As web service techniques have evolved and WS-Addressing has been defined, OGSI and its implementation have been superseded by WS-Resource Framework (WS-RF) [41, 62] and GT4.0, respectively. GT4.0 retains all functionalities provides by GT3.0, while adopting a new way, e.g. exploiting WS-Addressing and WS-Notification, to manage Grid services’ states.

There are two major differences between these two infrastructures. The first one is the way of managing service states. OGSI requires Grid services to be stateful and state information has to be embedded within services. Instead of putting states in web services, WS-RF keeps web services stateless while combines web services with separated state information to enable web services to be stateful. (Separated state information is referred to as Resource Properties in the following description.) The second difference is how they handle the notification of changes of service states. Using
WS-Notification, GT4.0 provides a more generic, hierarchical topic-based approach for publish/subscribe-based notification.

**Benefits Brought by GT4.0** Porting GATES from GT3.0 to GT4.0 is profitable. We could make use of the new features of GT4.0 in the following 3 aspects.

First, we could further improve the efficiency of dynamic migration by taking advantage of the feature that GT4.0 separates a service with its resources. As mentioned before, when a migration operation is needed, only LSS is automatically transmitted to a new node. However, under some circumstances that LSS is not small enough and services are frequently migrating because of drastically dynamic execution environments, the overheads caused by transmitting LSS could be significant. To overcome such overheads, we separate LSS with the service that is updating it. When a service migrates to a new place, it still remembers the location of its LSS and accesses it remotely. Such design could be inefficient in the situation where services frequently update their LSS while the network bandwidths between the services and the LSS are low. We may solve this problem by allocating a local copy of LSS in the service’s memory and let the service always update the local LSS. As the service is processing data in a new round, the local LSS can be replicated to the remote LSS by a replication service. We believe that processing data is more time-consuming than replicating LSS. By overlapping the two operations, the system can operate migration more efficiently.

Second, we can utilize WS-notification provided by GT4.0 to reduce the overhead caused by the adaptation algorithm in GATES. The adaptation algorithm can automatically adjust adaptation parameters so as to make applications adaptive to dynamic execution environments. To do it, the algorithm periodically collects load
states of two consecutive services. When the states change significantly, the algorithm then need tune adaptation parameters. However, the overhead of collecting such states could be large, especially when the collecting operation occurs frequently. As a solution, we can make the variable representing the load states a resource property and subscribe to it. When the load states change significantly, a notification can be sent to the consecutive service automatically. Because the notification only happens when it is necessary so the overhead due to the periodic information collecting can be eliminated and the overall overhead of the adaptation algorithm can be reduced. The WS-notification makes it easy to implement such notification mechanism.

Issues When Porting GATES to GT4.0 There are also some issues we need to consider when we port GATES from GT3.0 to GT4.0. These issues are associated with the two differences between OGSI and WS-RF we mentioned above. The first, and probably most notable, issue is that we need to distinguish variables representing GATES service’ states from temporary information in the services. The second issue is where these resource properties should be placed. GT4.0 does not constrain locations of resource properties. They could be in a node where their service is, or in another more reliable node. They could stay alone, or be clustered with other services’ resource properties. The solution to this issue depends on the our preferences to system reliability, flexibility and performance. For example, the system may achieve the best performance when a service and its frequently updated resource properties, e.g. load states, are located in the same machine, while unchangeable resource properties, e.g. placement information, can be placed in a more reliable machine to make the system more robust.
8.2.3 Supporting Fault-Tolerance and High Availability

Node failures and network congestion can result in blocking processing data streams. Many stream processing applications prefer approximate results to long delays but need to see the correct output streams [72, 63]. Without supporting fault-tolerance and high availability, distributed data stream processing systems may eventually fail to meet the real-time requirement. Therefore, such support is desirable and essential. GATES currently does not provide fault-tolerance for its applications.

To implement fault-tolerance functionality, we could place LSS with other important information in a service to a remote site. Once the original service fails, we can create a new service in the remote site and restore the failed service’s status from the stored information. We can also replicate the these information to multiple sites to strength tolerance capabilities.

To provide high availability, we can design a infrastructure similar to the one introduced by Borealis [3]. We can first replicate each path from a data source to the destination. One path is defined as a primary and others are secondaries. We can insert a duplicator between each data source and its downstream nodes. These duplicators can duplicate one input stream to multiple streams and send them to replicated paths. Meanwhile, the duplicator periodically inserts timestamped tokens into data streams. There are two kinds of tokens, the primary one and the secondary one. Primary tokens are inserted into primary paths. There are some mergers that can be used to merge a stream from the primary path with those from the secondary paths together. Eventually, only one output stream is generated by each merger. The merging procedure is as follows. Mergers exam the tokens in streams, and always output the data in the stream in which the tokens have earliest timestamps. If a
merger finds that the tokens in one secondary stream always have earlier timestamps than those in the primary stream, it will inform the corresponding duplicator to switch the secondary path and the primary one. By this means, when processing along the primary path is inefficient or fails, the secondary path can be automatically activated.

8.2.4 Further Relieving Programming Burdens from Application Developers

Continuous query systems, such as STREAM and Aurora, have means or languages to declare stream schemas/meta-data, so that they easily extract tuples from raw streaming data. GATES currently pushes this job to its applications. In other words, GATES only passes raw streaming data to its applications, which are responsible for extracting tuples. According to our experience, the extracting job is time-consuming and very similar in every application. Therefore, it is favorable to integrate such job to GATES. We could allow application developers to provide XML schemas that specify the meta-data of the streams, and embed the schemas to their application configuration file. By this means, GATES can read the schemas/meta-data, then separate tuples from each other in data streams, and pass them to its applications. There are lots of advantages of doing this, apart from relieving certain programming burdens from application developers.

First, because GATES can convert raw data streams to tuple streams, it can apply the techniques, e.g. memory shedding, sliding windows, presented in the continuous query systems to balance loads.

Second, it would be much easier to migrate GATES services, because when migrating them GATES itself is able to freeze the processing at the boundary of a tuple
and direct the rest of tuples in buffers to a new location. Currently, GATES needs assistance from its applications in order to locate the boundaries of tuples.

Third, in the scenarios where adaptation parameters are sampling rates (In fact, this happens a lot), GATES can delegate applications to automatically apply new sampling rate to the next round of processing. Currently, GATES only suggests new values of adaptation parameters and applications are responsible for using these new values. Since GATES could see tuples streams, and when the adaptation parameter happens to be the sampling rate, GATES can apply some sampling algorithm to tuples streams, assuming applications do not specify their own sampling algorithm.

8.2.5 Supporting Distributed Continuous Queries

As stated before, GATES plugs user-defined algorithms into remote sites while distributed Continuous Query systems deploy pre-defined query operators to distributed nodes. In fact, there is not significant difference between user-specified codes and pre-defined operators in terms of the processing procedure: both of them take input stream(s) and process them and generate one or multiple streams. By this observation, we could also support distributed continuous queries by replacing user-defined codes with pre-defined query operators. However, there are two issues we need to address.

First, we need to specify a set of query operators and a mechanism used to translate user-specified queries to pre-defined query operators. To accomplish this task, we could employ existing efforts done by the work presented in Chapter 2.

Second, a GATES service would be required to be able to generate multiple output streams. Presently, a GATES service can only output one data stream. However,
in practice, some query operators, for example splitting, generate multiple output streams.

Certainly, the contribution of GATES’s supports for continuous queries over distributed data streams may not be trivial. There is no continuous query system that can process distributed data streams in Grid environment and meanwhile support adaptive queries over streams. Further, such supports increase the number of candidates for GATES to compare with when we evaluate the system.


