GATES: A Grid-Based Middleware for Processing Distributed Data Streams

Liang Chen Kolagatla Reddy Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
{chenlia,reedyk,agrawal}@cis.ohio-state.edu

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 distributed processing of data arising from a distributed set of sources. Thus, we believe that a grid environment is well suited for flexible and adaptive analysis of these streams.*

This paper reports the design and initial evaluation of a middleware for processing of distributed data streams. Our system is referred to as GATES (Grid-based AdapTive Execution on Streams). This system is designed to use the existing grid standards and tools to the extent possible. It flexibly achieves the best accuracy that is possible while maintaining the real-time constraint on the analysis. We have developed a self-adaptation *algorithm for this purpose.*

Results from a detailed evaluation of this system demonstrate the benefits of distributed processing, and the effectiveness of our self-adaptation algorithm.

1 Introduction

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 [13]. 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 location 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 [19]. At the same time, database systems community has been developing architectures and query processing systems targeting continuous data streams [3]. Recently, a workshop was held as part of FCRC in San Diego, targeting different aspects of data stream processing[1].

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. 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. It will be desirable if such adaptation can be supported in a middleware, and does not need to be hard-coded for a specific application. However, no existing grid middleware supports such functionality.

This paper reports the design and evaluation of a middleware for processing of distributed data streams. Our system is referred to as GATES (Grid-based Adaptive Execution on Streams). The three important aspects of this system are as follows. First, it is designed to use the existing grid standards and tools to the extent possible. Specifically, our system is built on the Open Grid Services Architecture (OGSA) model and uses the initial version of GT 3.0. Second, the system offers a high-level interface that allows the users to specify the algorithm(s) and the steps involved in processing data streams. The users need not be concerned with the details like discovering and allocating grid resources, registering their own data stream's web services and deploying the web services. Thus, the system is *self-resource-discovering.*

Third, and probably the most significant aspect of our system is that it 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 adjust the accuracy of the analysis. This is done by changing the sampling rate, size of the summary structure maintained, and/or the choice of the algorithm to be used. 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 a new algorithm for this purpose. In summary, the system is *self-adapting.*

We have carried out a detailed evaluation of this system using two application templates that are representative of the distributed stream-based applications. The main observations from our set of experimental results are as follows. First, distributed processing of data streams increases performance and can help meet real-time constraint, with only a modest loss of accuracy. Second, self-adaptation can help choose a balance between performance and accuracy, even as resource availability is varied widely. Third, our self-adaptation algorithm is able to tune adaptation parameters successfully, as network bandwidth or processing power may become a constraint.

The rest of this paper is organized as follows. We describe some typical applications requiring grid-based stream processing in Section 2. The overall design and API offered by the system is presented in Section 3. The algorithm for supporting self-adaptation is presented in Section 4. Detailed experimental evaluation is presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

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 [8]. 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

[1] Please see http://www.research.att.com/conf/mpds2003

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 [9]. 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 [24]. 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 [7]. 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 [14]. 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 makes such analysis challenging.

3 Middleware Design and Application Programming Interface

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

3.1 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) [16], and uses its reference implementation, Globus 3.0. 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 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.

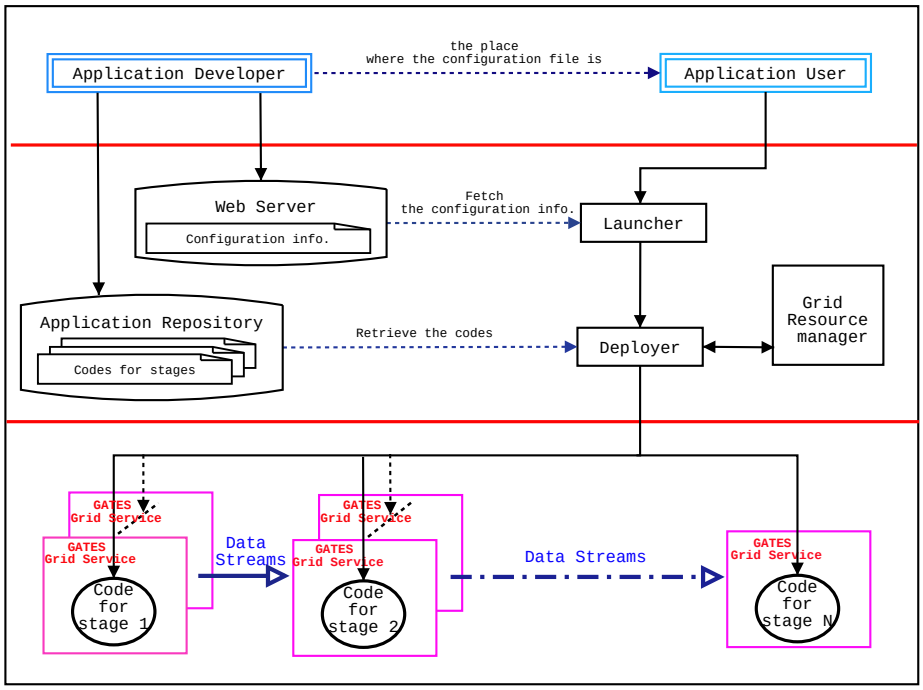

Figure 1. Overall System Architecture

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. For this, the middleware allows the application developers to expose one or more *adjustment* parameters at each stage. An *adjustment* parameter is a tunable parameter whose value can be modified to increase the processing rate, and in most cases, reduce the accuracy of the processing. Examples of such adjustment parameters are, rate of sampling, i.e., what fraction of data-items are actually processed, and size of summary structure at an intermediate stage, which means how much information is retained after a processing stage. The middleware automatically adjusts the values of these parameters to meet the real-time constraint on processing.

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 Globus 3.0, 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.

3.2 System Architecture and Design

The overall system architecture is shown in the Figure 1. The system distinguishes between an *application developer* and an *application user*. An application developer is responsible for dividing an application into stages, choosing adjustment parameters, and implementing the processing at each stage. Moreover, the developer writes an XML file, specifying the configuration information of an application. Such information 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 [15] that implements the self-adaptation algorithm and is able to contain and execute user-specified codes.

3.3 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 adjustment parameters. Specifically, the function *specifyPara(init_value, max_value, min_value, incre_or_decre)* is used to specify an initial value and a range of acceptable values of an adjustment 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 as follows.

```
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();
  }
}
...
}
```

In the example above, an adjustment 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 parameter decreases the performance. During data processing, the middleware's self-adaptation algorithm automatically keeps evaluating sampling rates. At the end of every iteration, a new sampling rate is returned by the function getSuggestedValue(), and the new rate is used for the computation in the next iteration.

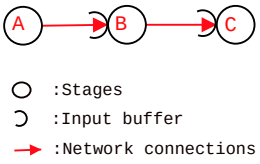

Figure 3. An application that comprises three stages

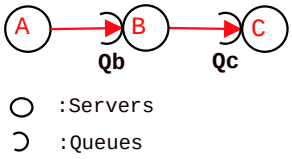

Figure 4. A Queuing model of the system

4 Self Adaptation Algorithm

As stated before, an important functionality of the GATES system is self-adaptation to meet the real-time constraint while keep processing as precise as possible. In this section, we present the algorithm we currently use in our system.

4.1 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 is presented in Figure 4.

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

If we have flexibility in controlling the accuracy of the analysis, our goal is to adjust the parameters to maintain a good balance between λ and μ. Clearly, if $\mu < \lambda$, the queue will saturate, and real-time constraint on processing cannot be met. In this case, we need

Symbols	Definition
Variables	
d	Current length of the queue
$\bar{d}$	Average of the d values in recent times
$\tilde{d}$	Long-term average queue size factor
t_1	The number of times the system was *over-loaded*
t_2	The number of times the system was *under-loaded*
w	The number of times the system was recently *over-loaded*
ϕ_1, ϕ_3	Functions reflecting queue's long-term load
ϕ_2	Functions reflecting queue's recent load
P	Adjustment parameter for a server
T_1	No. of *over-load* exceptions that the server reported to the sending server
T_2	No. of *under-load* exceptions that the server reported to the sending server
σ_1	A function to factor $\tilde{d}_B$ in parameter adjustment
σ_2	A function to factor $\phi_1(T_1, T_2)$ in parameter adjustment
Constants	
α	Learning rate for d
W	Window size
D	Expected length of the queue
C	Maximum capacity of the queue
P_1, P_2, P_3	Weights to ϕ_1, ϕ_2, ϕ_3, respectively
LT_1	Minimum threshold for the average queue size
LT_2	Maximum threshold for the average queue size

Figure 2. Summary of Symbols Used

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 λ is much lower than μ, we are under-utilizing the current server. In this case, we can speed up the processing at the sending server.

As λ and μ 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 adjustment parameters.

4.2 Detailed Description

This subsection gives a detailed description of the algorithm. The list of terms used in our algorithm is listed in Figure 2.

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 ϕ_1, and ϕ_2 and ϕ_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 $\bar{d}$. 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 ϕ_i as follows.

$$\phi_1(t_1,t_2) = \begin{cases} \frac{t_1-t_2}{t_1+t_2} & \text{if } (t_1+t_2>0) \\ 0 & \text{if } (t_1+t_2=0) \end{cases} \quad (1)$$

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

$$\phi_3(\bar{d}) = \begin{cases} \frac{\bar{d}-D}{D} & \text{if } \bar{d} < D \\ \frac{\bar{d}-D}{C-D} & \text{if } \bar{d} \geq D \end{cases} \quad (3)$$

Both ϕ_1 and ϕ_3 reflect the queue's long-term load, whereas, ϕ_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 $\tilde{d}$.

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

Here, P_1, P_2, P_3 are the factors that give weight to ϕ_1, ϕ_2, and ϕ_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 ϕ_i, $\tilde{d} \in [-C, C]$, and the closer $|\tilde{d}|$ to C, it is more likely that the unit is having very high or low load. Particularly, when $\tilde{d}$ exceeds the pre-defined interval $[LT_1, LT_2]$, 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 adjustment 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 adjustment parameter for processing at the server B in Figure 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 ϕ_1 can be computed by applying Equation 1.

We design Equation 4 to calculate the adjustment of P_B.

$$\Delta P_B = \tilde{d_B} * \sigma_1(\tilde{d_B}) - \phi_1(T_1,T_2) * \sigma_2(\phi_1(T_1,T_2)) \quad (4)$$

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. σ_1 and σ_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 ΔP_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.

5 Experimental Evaluation

This section presents results from a number of experiments we conducted to evaluate our GATES system. Specifically, we had the following goals:

- Show how distributed processing of data streams is more efficient.
- Show how a system with self-adaptation of parameters can achieve the right trade-off between efficiency and accuracy.
- 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 [27]. 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.

5.1 Applications

The experiments we report were conducted using two application templates, which are representative of the applications we described in Section 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 [17]. 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 communicate 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 storing $n/2$ 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 *adjustment* parameter used in this application.

The second application is `comp-steer`, based around the use of data stream processing 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 machine, 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 adjustment parameter used in this application.

5.2 Benefits of Distributed Processing

Processing Style	Average Performance (sec.)	Avg. Accuracy
Centralized	257.5	.99
Distributed	180.8	.97

Figure 5. Benefits of Centralized Processing: 4 Sub-streams

Our first experiment demonstrated the benefits associated with distributed processing 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 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. Figure 5 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 an accuracy of 1. This is because the algorithm we implemented just takes one pass on the data and is approximate [17].

Our results shows 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.

5.3 Impact of Self Adaptation

Our second application also used the `count-samps` applications. 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.

Figures 6 and 7 show the execution time and accuracy, respectively, of these five versions, and on the four different configurations. As the Figure 7 shows, the accuracy can be quite low if a very small value of

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

Figure 6. Execution Time of Different Versions

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

Figure 7. Accuracy of Different Versions

the adjustment parameters is chosen. Similarly, Figure 6 shows that the execution time can be very large if the value of the adjustment 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 Figure 6, 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.

5.4 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 8 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.

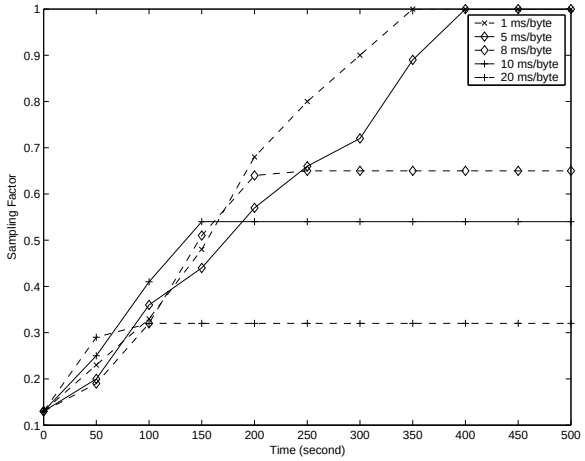

Figure 8. Self-Adaptation with Different Processing Requirements

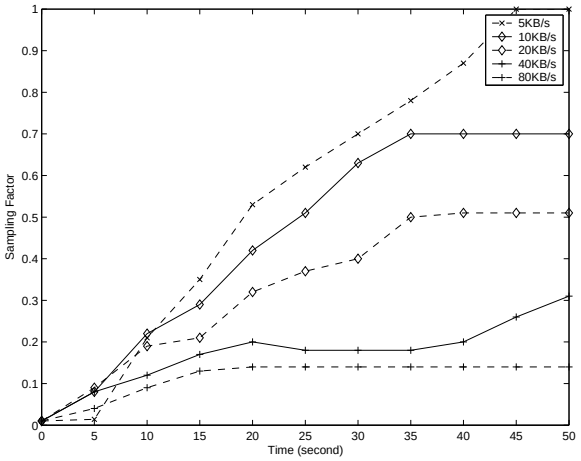

Figure 9. Self-Adaptation with Different Data Generation Rates

5.5 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 9, 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.

6 Related Work

The work that is probably the closest to our work is the dQUOB project [25, 26]. This system enables continuous processing of SQL queries on data streams. Our work is distinct in the following ways. First, we support an API to allow general processing, and not just SQL queries. Second, the processing can be done in a pipeline of stages. Third, our system is built on top of Globus 3.0, and thus, exploits the existing support for resource discovery. Finally, we support adaptation in a distributed environment. As part of the Linked Environment for Atmospheric Discovery (LEAD) project, work is ongoing to incorporate dQUOB-like support in grid environment.

Our work also derives from the DataCutter project at the Ohio State University [5, 4]. 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 the Open Grid Services Architecture (OGSA). Third, we enable easy deployment in a distributed environment, through an application container.

Data stream processing has received much attention in the database community [18]. Prominent work in this area has been done at Stanford [2], Berkeley [11], Brown and MIT [10], Wisconsin [28], among others. The focus in this community has largely been on centralized processing of a single data stream. Our focus is quite different, as we consider distributed processing of distributed data streams, and use grid resources and standards.

Application adaptation has been studied in many contexts, including as part of a grid middleware. Cheng *et al.* have developed an adaptation framework [12]. Adve *et al.* [1] 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 [6, 20, 21, 22, 23, 29]. Our work is different in considering grid-based distributed execution and meeting real-time constraint.

7 Conclusions

With scientific instruments and experiments that continuously generate data, and increasing network speeds, we expect processing of 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. GATES allows processing on distributed data streams to be specified as a pipeline of stages. By using Globus 3.0, the middleware allows resource discovery and automatic matching between the resources and requirements. By supporting an application container, the middleware allows enables easy deployment of a distributed application.

Most importantly, our GATES system is self-adapting and automatically tunes certain parameters to allow the most accurate analysis, while still meeting the real-time constraint. The middleware API allows the application developers to expose certain em adjust-

ment parameters. We have developed an algorithm for tracking the load at different processing stages and adjusting the values of these parameters at runtime. Our experimental results have shown that this algorithm is effective.

References

[1] Vikram Adve, Vinh Vi Lam, and Brian Ensink. Language and Compiler Support for Adaptive Distributed Applications. In *Proceedings of the SIGPLAN workshop on Optimization of Middleware (OM) and Distributed Systems*, June 2001.

[2] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. In *Proc. of the 9th International Conference on Data Base Programming Languages (DBPL '03)*, Sep 2003.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 2002 ACM Symposium on Principles of Database Systems (PODS 2002) (Invited Paper)*. ACM Press, June 2002.

[4] Michael D. Beynon, Tahsin Kurc, Umit Catalyurek, Alan Sussman, and Joel Saltz. A component-based implementation of iso-surface rendering for visualizing large datasets. Technical Report CS-TR-4249 and UMIACS-TR-2001-34, University of Maryland, Department of Computer Science and UMIACS, May 2001.

[5] Michael D. Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Optimizing execution of component-based applications using group instances. In *Proceedings of the Conference on Cluster Computing and the Grid (CCGRID)*, pages 56–63. IEEE Computer Society Press, May 2001.

[6] V. Bhargavan, K.-W. Lee, S. Lu, S. Ha, J. R. Li, and D. Dwyer. The TIMELY Adaptive Resource Management Architecture. *IEEE Personal Communications Magazine*, 5(4), August 1998.

[7] E. Borovikov, A. Sussman, and L. Davis. A High-Performance Multi-Perspective Vision Studio. In *Proceedings of the International Supercomputing Conference (ICS)*. ACM Press, June 2003.

[8] J. Bunn and H. Newman. Data-Intensive Grids for High-Energy Physics. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley and Sons, 2003.

[9] R. Carlson. Earthscope Workshop Report: Scientific Targets for the World's Largest Observatory Pointed at the Solid Earth. Report Published March 2002, see www.earthscope.org.

[10] D. Carney, U. Etintemel, M. Cherniak, C. Corvey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zodnik. Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of Conference on Very Large DataBases (VLDB)*, pages 215–226, 2002.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 269–280, 2003.

[12] Shang-Wen Cheng, David Garlan, Bradley Schmerl, Peter Steenkiste, and Ningning Hu. Software Architecture-based Adaptation for Grid Computing. In *Proceedings of IEEE Conference on High Performance Distributed Computing (HPDC)*. IEEE Computer Society Press, August 2002.

[13] A. Chervenak, I. Foster, C. Kesselman, C. Salisbusy, and S. Tuecke. The Data Grid: Towards An Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 2001.

[14] P. Dokas, L. Ertoz, V. Kumar, A. Lazarevic, J. Srivastava, and P. Tan. Data Mining for Network Intrusion Detection . In *Proc. of the NSF Workshop on Next Generation Data Mining*, November 2002.

[15] Ian Foster, Carl Kesselman, J. Nick, and Steven Tuecke. Grid Services for Distributed Systems Integration. *IEEE Computer*, 2002.

[16] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Open Grid Service Infrastructure Working Group, Global Grid Forum*, June 2002.

[17] Phillip B. Gibbons and Yossi Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *Proc. of the 1998 ACM SIGMOD*, pages 331–342. ACM Press, June 1998.

[18] L. Golab and M. Ozsu. Issues in data stream management. In *SIGMOD Record, Vol. 32, No. 2*, pages 5–14, June 2003.

[19] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering Data Streams. In *Proceedings of 2000 Annual IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 359–366. ACM Press, 2000.

[20] Z. Jiang and L. Kleinrock. An Adaptive Pre-Fetching Scheme. *IEEE Journal of Selected Areas in Communication*, 1999.

[21] B. Li and K. Nahrstedt. A Control Based Middleware Framework for Quality of Service Adaptations. *IEEE Journal of Selected Areas in Communication, Special Issue on Service Enabling Platforms*, 1999.

[22] B. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of 16th ACM Symposium on Operating Systems Principles*, October 1997.

[23] Ossama Othman and Douglas C. Schmidt. Issues in the Design of Adaptive Middleware Load-Balancing. In *Proceedings of the SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*, pages 205–213. ACM Press, June 2001.

[24] S. G. Parker, D. M. Weinstein, and C. R. Johnson. The SCIRun computational steering software system. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*. Birkhauser Press, 1997.

[25] Beth Plale. Leveraging Runtime Knowledge about Event Rates to Improve Memory Utilization in Wide Area Data Stream Filtering. In *IEEE High Performance Distributed Computing (HPDC)*, August 2002.

[26] Beth Plale and Karsten Schwan. Dynamic Querying of Streaming Data with the dQUOB System. *IEEE Transactions on Parallel and Distributed Systems*, 14(3), April 2003.

[27] Teragrid project partners. The TeraGrid: A Primer, September 2002. Available at www.teragrid.org.

[28] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002.

[29] M. Yarvis, P. Reiher, and G. Popek. Conductor: A Framework for Distributed Adaptation. In *Proceedings of 7th Workshop on Hot Topics in Operating Systems*, March 1999.