Self-Adaptation in a Middleware for Processing Data Streams

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

1 Introduction

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 by the National Lambda Rail (NLR) proposal and the interconnectivity between the TeraGrid and Extensible Terascale Facility (ETF) site. 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. We view the problem of flexible and adaptive processing of distributed data streams as a grid computing problem. We have been developing a middleware for processing of distributed data streams. Our system is referred to as GATES (Grid-based AdapTive Execution on Streams). 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.**

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. For this purpose, our system flexibly achieves the best accuracy that is possible while maintaining the real-time constraint on the analysis. We have developed and implemented a self-adaptation algorithm. This paper focuses on describing and evaluating this algorithm.

2 Self Adaptation Support

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.

In adjusting the value of this parameter, the middleware needs to evaluate the load at any given stage, and then needs to decide how the value should be adjusted.

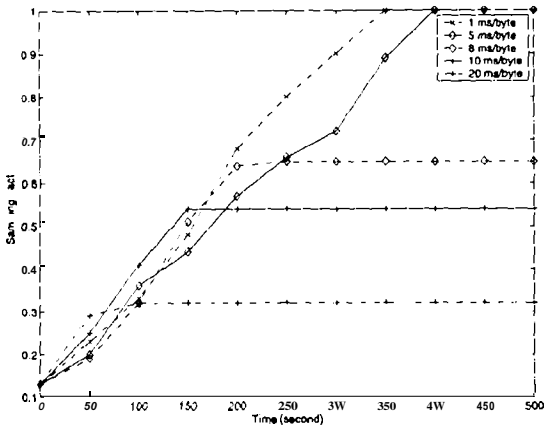

Figure 1. Self-Adaptation with Different Processing Requirements

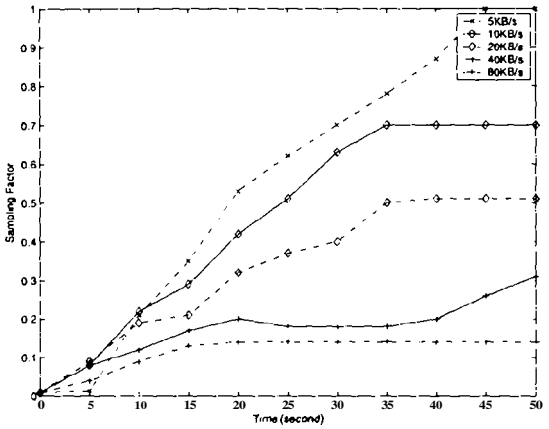

Figure 2. Self-Adaptation with Different Data Generation Rates

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. As λ and μ axe 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. 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.

3 Experimental Evaluation

The application template we used 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.

In the first experiment, 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 1 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.

Our second experiment 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 2, 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.