Accommodating Bursts in Distributed Stream Processing Systems

Yannis Drougas[1], Vana Kalogeraki[1,2]
[1]Department of Computer Science and Engineering, University of California-Riverside
[2]Department of Informatics, Athens University of Economics and Business
{drougas,vana}@cs.ucr.edu

Abstract

Stream processing systems have become important, as applications like media broadcasting, sensor network monitoring and on-line data analysis increasingly rely on real-time stream processing. Such systems are often challenged by the bursty nature of the applications. In this paper, we present BARRE *(Burst Accommodation through Rate REconfiguration), a system to address the problem of bursty data streams in distributed stream processing systems. Upon the emergence of a burst, BARRE dynamically reserves resources dispersed across the nodes of a distributed stream processing system, based on the requirements of each application as well as the resources available on the nodes. Our experimental results over our Synergy distributed stream processing system demonstrate the efficiency of our approach.*

1. Introduction

During the recent years, numerous applications that generate and process continuous streaming data have emerged. Examples include network traffic monitoring, financial data analysis, multimedia delivery and sensor streaming in which sensor data are processed and analyzed in real-time [1], [2]. In a typical stream processing application, streams of data are processed concurrently and asynchronously by one or more processing components. Examples of processing components include filtering operations aggregation operators, or more complex operations, such as top-K querying and video transcoding.

A number of stream processing systems have been proposed in the literature, including Aurora [3], STREAM [1], TelegraphCQ [2] and the Cougar project [4]. These systems have primarily focused on designing new operators and new query languages, as well as building high-performance stream processing engines operating in a single node. Recent efforts have proposed distributed stream processing infrastructures and have investigated composition and placement algorithms. The majority of these (including our previous work [5], [6], [7]) focus on composition and placement techniques making the assumption that resource availability and application rate requirements are constant.

However, a significant number of distributed stream processing applications, although they typically operate under constant data rates, can suddenly experience a burst. The burst can be the result of increasing traffic volume due to a DoS attack in the example of a network traffic monitoring application, or a large number of alarm messages being generated when there is a time-critical event (such as a fire, chemical spill or an earthquake) in a sensor network field. In these settings, it is important that no data is lost and that time-critical events are processed and analyzed in a timely manner.

Accommodating such bursty data streams brings significant challenges to the design of distributed stream processing systems: First, bursty data streams are characterized by large volumes, variable and unpredictable rates. In these cases it is very difficult to determine a priori when a burst will occur. Second, computational and communication resources are shared by multiple concurrent and competing applications, thus, the occurrence of a burst can affect the performance of existing applications in the system. Third, a distributed stream processing system consists of a number of nodes geographically distributed, where the functionality of a processing component is offered only by a subset of the nodes of the system. Thus, meeting the real-time requirements of the distributed stream processing applications can entail considerable strain on both communication and processing resources.

A common way to deal with such overload situations is to apply admission control that decides at the time of request whether the new application should be accepted or to perform resource reservation that a priori commits resources to applications to address bursts [8]. However, these techniques result in under-utilization of the system and significant waste of resources when bursts do not occur. Furthermore, blindly reserving resources without considering the actual needs of the applications has limited advantage. Another solution to this problem is to perform QoS degradation [9] to adapt to changing resource conditions or application demands. However, this is a reactive strategy where a burst is treated only after it has occurred, and thus could lead to significant data loss. A third strategy

1-4244-0910-1/07/$20.00

is to predict overloads based on historical data about each application [10]. However, in many occasions, such data is not available.

In this paper, we present *BARRE* (Burst Accommodation through Rate REconfiguration), our system to accommodate bursts of data streams in distributed stream processing systems. The question we want to answer is the following: *"Can we accommodate a burst without having to reserve resources a priori? How should we react to a burst when it occurs?"* Our approach is as follows: BARRE works in two phases: (1) An offline phase during which the system proactively generates a number of possible rate allocation assignments based on the requirements of the applications as well as the resource availability on the nodes. During this phase, the system determines the rates of the data streams of the application components and allocates resources dispersed across the nodes of the system. (2) An online phase where the system monitors the resource usage and application behavior. Upon the onset of a burst, the system uses the pre-calculated rate allocation plans in order to modify the input rates of individual components to accommodate the burst in a timely manner. The goal of reconfiguration is to make provisions for future bursts. This is in contrast to other dynamic reconfiguration schemes [11], [12], that seek to optimize a given utility, expressed as a weighted sum of the application input rates. By calculating the plans in advance, the advantage is that the system can react to the burst in a timely manner. This is in contrast to existing solutions that either reserve resources a priori or perform dynamic reconfiguration reactively. Our experimental results over our distributed stream processing system Synergy [13] demonstrate the efficiency and benefits of our approach.

2. System Architecture and Model

2.1. System Architecture

We have built our scheme in our distributed stream processing system called Synergy [13] (Figure 1). Synergy consists of multiple nodes, connected in an overlay network. Each node in the overlay offers one or more *services* to the system. A service is a function that defines the processing of a finite amount of input data. Examples of processing are aggregation of sensor readings, data filtering or video transcoding. A stream processing application is executed collaboratively by peers of the system that invoke the appropriate services. The instantiation of a service on a node is called a *component*. A component is a running instance of a service. A component operates on individual chunks of data, named *data units*. Examples of data units are sequences of picture or audio frames (for example, in a multimedia application), or sets of measured values (for

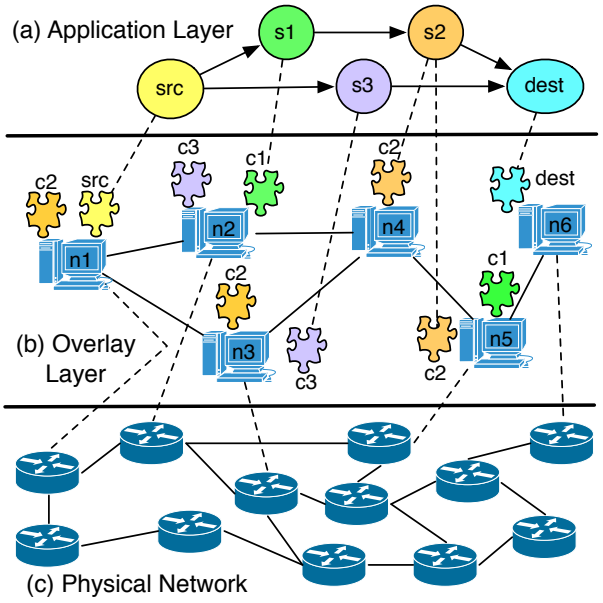

Figure 1. The architecture of a distributed stream processing system.

example, in a sensor data analysis application). The size of a data unit depends on the application. Upon reception of a data unit by a node, the data unit is inserted in the scheduler's queue waiting to be processed. To execute a data unit, the appropriate component is invoked.

The user submits a request for a *set of applications* $\{app_q\}, 1 \leq q \leq Q$ to one of the nodes in the system, along with their respective *initial rate requirements* r_q. Each application is described as a sequence of services that need to be invoked. The initial rate requirement r_q for an application represents the delivery rate of data units requested by the application. An example of an application requested by the user is shown in Figure 2(a). When submitting a request, the user expects from the system to instantiate the appropriate components on the system in order to perform the processing required by each application, at the rate required by the application.

Each invoked component c_i is characterized by its resource requirements $u_j^{c_i}$ for each resource j it uses (*e.g.* CPU or bandwidth) and its selectivity sel^{c_i}. The selectivity represents the ratio of output rate to input rate for the component. The rate requirements and the selectivity of a component are characteristics of the service run by the component. These can be provided by the user prior to application execution or acquired through profiling at runtime. Note that the execution of a service for an application can be assigned to more than one components with each component being responsible for a subset of the data that will be processed by the component. An example of a service being executed by multiple components is shown

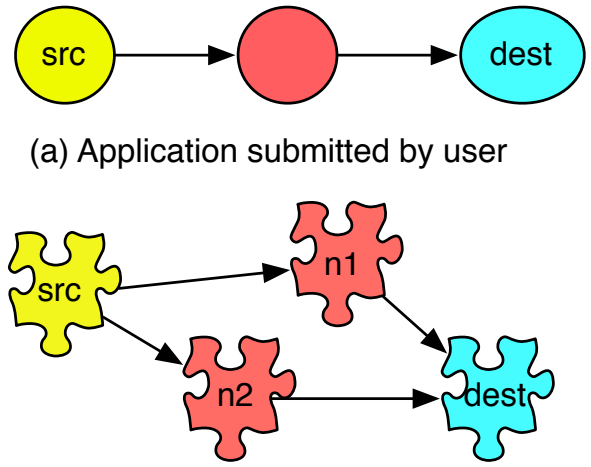

Figure 2. An example of an application.

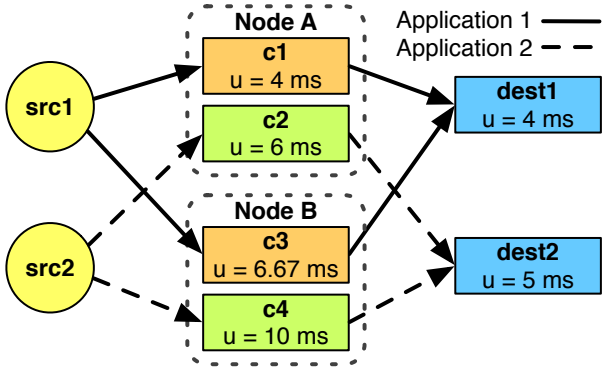

Figure 3. An example of two applications.

in Figure 2(b).

2.2. Problem Formulation

Let us assume a request for a set of applications submitted to our system. Our goal is to determine the rate assignment of the components invoked by each application app_q so that the application rate requirement r_q is met. If necessary, more than one components can be instantiated for each service requested.

To satisfy the node and link capacity constraints, for each resource j ($1 \leq j \leq J$) the sum of resource j's usage of the components running on a node n should be no more than the availability A_j^n of resource j on n. This is expressed as follows:

$$\forall n \in \mathcal{N}, \sum_{c_i \in n} r_{c_i} \cdot u_j^{c_i} \leq A_j^n, 1 \leq j \leq J \qquad (1)$$

where r_{c_i} and $u_j^{c_i}$ represent the assigned rate and resource needs for component c_i respectively. Additional constraints that need to be taken into account are the flow conservation constraints. Such constraints represent the relation between the input and the output rates of a component, defined by the *selectivity* of the component. The selectivity sel^{c_i} of a component c_i represents the average ratio of the number of output data units to the number of input data units of c_i. The selectivity of each component depends on the service run by the component. Let $\mathcal{D}(c_i)$ be the set of downstream components of component c_i. Then the flow conservation constraints are represented as:

$$\forall c_i, \sum_{c_j \in \mathcal{D}(c_i)} r_{c_j} = sel^{c_i} \cdot r_{c_i} \qquad (2)$$

Given the above constraints, the composition algorithm must come up with a rate assignment that will prove most beneficial on the event of a sudden burst. In other words, the resulting assignment must be such that the minimum number of data units will be missed upon an event of a sudden burst of one or more of the applications.

Assume we have Q applications. Consider the Q-dimensional Euclidean space $\mathbb{R}_+^Q$, where each dimension represents the input rate of the corresponding application. Each combination of input rates can be represented by a point in $\mathbb{R}_+^Q$. The component requirements and node capacities define a *feasible region* in this space. Thus, the feasible region is the set of all points (application input rates combinations) that nodes in the given distributed stream processing system can accommodate without any data unit being dropped. The form of linear constraints (1) and (2) suggest that in the general case of Q applications, the feasible region is a convex polytope [14].

Example 1. *Let us consider a simple example with two applications, shown in Figure 3. The figure shows the components invoked by the applications along with the corresponding times needed to process a single data unit. The feasible region is shown in Figure 4. In this example, the input rates of the applications are determined by the capacity constraints of each node.*

Next, we present the notion of *dominance* between two application input rate combinations. Given two application input rate combinations represented by points p_1 and p_2 in the feasible region, point p_2 *dominates* p_1 ($p_2 \succcurlyeq p_1$) if and only if: (1) The input rates represented by p_2 are greater than or equal to the corresponding input rates in p_1 ($p_2(q) \geq p_1(q), 1 \leq q \leq Q$). (2) There is at least one input rate such that $p_2(l) > p_1(l)$. For example, points p_3, p_4 and p_5 in Figure 5 are some of the points that dominate point p. However, points p_1 and p_2 do not dominate p.

When a point p dominates a point p', this indicates that the input rate combination represented by p is "preferred" compared to the application input rate combination represented by p'. This is because when $p \succcurlyeq p'$ is true, then (1)

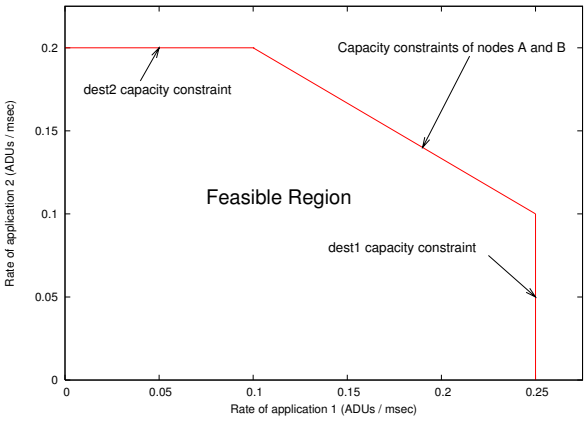

Figure 4. The feasible region for an example of two applications.

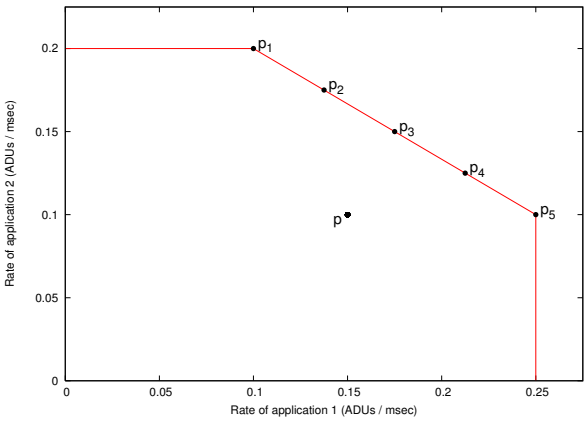

Figure 5. Some pareto points on the feasible region.

at least one of the application input rates represented by p is larger than the corresponding input rate for the same application represented in p' and (2) no application input rate in p is smaller than the corresponding application input rate in p'. Thus, if we have such a case to choose, it would be more beneficial to select the application rates that result to p.

For a point p in the feasible region, if there is no point p' within the feasible region such that $p' \succcurlyeq p$, then p is a *pareto point* [15]. For example, points p_1, p_2, p_3, p_4 and p_5 in Figure 5 are pareto points.

It can be easily understood that if there is a feasible solution for an input rate combination represented by a point p, then the input rate combinations represented by points dominated by p are also feasible. In addition, a component rate assignment calculated for p will also work for the dominated input rate combinations, since the rates of the components will be smaller than or equal to what was planned. Since the feasible region of our problem is convex, the pareto points lie on its boundary. Thus, for each point p in the feasible region (but not on its boundary), there is a pareto point p' on the boundary of the feasible region, that dominates p.

Let $\mathcal{C}^q$ be the set of components serving application q. If the input rate r_q of application q increases, the rate r_{c_i} of each component $c_i \in \mathcal{C}^q$ will increase as well. Let us assume that the input rate of application q increases by δ_q. Then, the amount of increase of r_{c_i} will be proportional to the fraction of the substream data being processed by c_i. Thus, r_{c_i} is expected to be increased by $\delta_q \cdot \frac{r_{c_i}}{r_q}$. In order for our distributed stream processing system to be able to sustain such an increase, Equation (3) must hold for each node $n \in \mathcal{N}$ of the system:

$$\sum_{c_i \in n} r_{c_i} \cdot u_j^{c_i} + \sum_{c_i \in n \cap \mathcal{C}^q} \delta_q \cdot \frac{r_{c_i}}{r_q} \cdot u_j^{c_i} \leq A_j^n \qquad (3)$$

where $1 \leq j \leq J$. The first sum in Equation (3) represents the current resource requirements of component c_i running on node n. The second sum represents the additional resource requirements due to the increase on the input rate of the application. By assuming a change δ_q to the rate of each application app_q, constraint (3) can be extended to the case where multiple bursts can occur simultaneously:

$$\forall n \in \mathcal{N}, \sum_{q=1}^{Q} \sum_{c_i \in n \cap \mathcal{C}^q} r_{c_i} \cdot \left(1 + \frac{\delta_q}{r_q}\right) \cdot u_j^{c_i} \leq A_j^n \qquad (4)$$

The larger the amount δ_q can become without violating the above equations for any of the nodes, the more substantial the burst of the corresponding application that can be sustained by the system without changing the rate assignment.

The objective of rate assignment is to minimize the probability of one or more bursts overloading the system. This helps in decreasing the number of data units that will be lost when reconfiguring the system so that it fully copes with the bursts. The input rate of any of the given applications can have a burst at any time. We would like our rate assignment to be able to sustain such an abrupt rate increase, no matter the application that demonstrates such behavior. In other words, there is the need that the values of all δ_q's to be as equal as possible. More formally, we would like:

$$p' = p + \boldsymbol{\delta} = \begin{bmatrix} r_1 + \delta_1 \\ \vdots \\ r_Q + \delta_q \end{bmatrix} \succcurlyeq \begin{bmatrix} r_1 + c \\ \vdots \\ r_Q + c \end{bmatrix} = p + \boldsymbol{\delta}^{eq} \Rightarrow$$

$$\boldsymbol{\delta} = \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_Q \end{bmatrix} \succcurlyeq \begin{bmatrix} c \\ \vdots \\ c \end{bmatrix} = \boldsymbol{\delta}^{eq} \qquad (5)$$

where $c \geq 0$. $\boldsymbol{\delta}^{eq}$ represents the case where all δ_q's have the same maximum possible value c so that constraints (2) and (4) are not violated. Constraints (5) express that the optimal solution should only maximize one or more δ_q's only in the case when this optimization will have no impact on the rest of the δ_q's. In what follows, we will usually need to refer only to the unit vectors that have the same direction with $\boldsymbol{\delta}$ and $\boldsymbol{\delta}^{eq}$. We represent those vectors as $\hat{\boldsymbol{\delta}}$ and $\hat{\boldsymbol{\delta}^{eq}}$ respectively. Given the above constraints, and the current input rates of the applications, we would like a rate assignment that maximizes:

$$\sum_{q=1}^{Q} \delta_q \tag{6}$$

It is important to note that for any input rate combination, maximizing $\sum \delta_q$ means that we will end up with an input rate combination on the border of the feasible region.

Example 2. *Let us consider the feasible region of Example 1 (Figure 4). Given an input rate combination $p = (r_1, r_2)$, maximizing $\delta_1 + \delta_2$ without violating constraints* (2), (4) *and* (5)*, would result in a component rate assignment that is equivalent to the combination of application input rates $p' = (r_1 + \delta_1, r_2 + \delta_2)$, as shown in Figure 6.*

On the other hand, if a burst results in the application input rate combination given in Figure 7, we would prefer a plan that provisions for an application input rate combination $p_2 = (r_1 + \delta_1, r_2 + \delta_2')$ rather than one that provisions for $p' = (r_1 + \delta_1, r_2 + \delta_2)$, since increasing δ_2 to δ_2' will have no impact on δ_1.

3. The BARRE Composition Algorithm

In the previous section, we formulated the component rate allocation problem. In this section, we present the operation of *BARRE (Burst Accommodation through Rate REconfiguration)*, our system that addresses the problem of burst accommodation in Distributed Stream Processing Systems. BARRE consists of the following components: (1) An offline application composition method that pre-calculates a number of rate assignment plans for different components of the system that can be used to accommodate bursts. Since equations (4) and (5) are not linear, the rate assignment problem is a rather time-consuming algorithm to run whenever reconfiguration was needed. Instead, we use a linear optimization method (as we show next) to pre-calculate the appropriate rate allocation. This phase takes place offline and the goal is to be able to generate and store some backup rate assignments a priori so that we can react in a timely manner whenever a burst occurs. (2) An online phase during which the system monitors the application incoming rates and the availability of the system resources to detect bursts. Upon the detection of a burst, our system needs to respond in a timely manner to address the burst. BARRE then generates a new plan by combining one or more of the pre-calculated assignments generated during the offline application composition phase. Then, it triggers this plan by adjusting the input rates of the components.

3.1. Feasible Region Determination

In section 2.2 we formulated our optimization problem. In the following we describe how to pre-select a few optimal points in the feasible region and the corresponding component input rate assignments that will be used to determine the optimal solution whenever a burst happens at runtime.

A problem that needs to be addressed concerns the application input rate combinations for which optimal rate assignment will be pre-calculated. Careful selection of such combinations is really important. Rate allocation should be calculated for as few input rate combinations as possible, since time and memory space overhead of the component rate allocation mechanism is proportional to the number of pre-calculated combinations. On the other hand, we need to make sure that our algorithm will come up with a rate allocation scheme whenever the input rates are such that the applications can be accommodated by the system.

BARRE determines the feasible region by taking advantage of its shape. Specifically, BARRE identifies the vertices of the feasible region that are also pareto points. In what follows, we will call such points *index points*. For example, let us consider the feasible region shown in Figure 5. As mentioned previously, the pareto points of the feasible region shown in the figure are all the points that lie on the straight line that stretches from point p_1 to point p_5. Points p_1, p_2, p_3, p_4 and p_5 are some of the pareto points for the particular problem. However, points p_1 and p_5 are the only pareto points that are also edges of the feasible region. Thus, they are the only index points when considering the particular feasible region. Like all points in the feasible region, an index point represents an application input rate combination. For each index point, BARRE calculates an appropriate component rate assignment that results in the respective application input rate combination. These component rate assignments will be used to construct the appropriate component rate allocation on the event of a burst. In addition, each index point keeps a list to at most Q *facets* [16]. A facet in a Q-dimensional feasible region is a $(Q-1)$-dimensional face of the region. In other words, a facet is one of the sides of the feasible region.

Next, we describe the methods we use to identify the index points and the related component rate assignments.

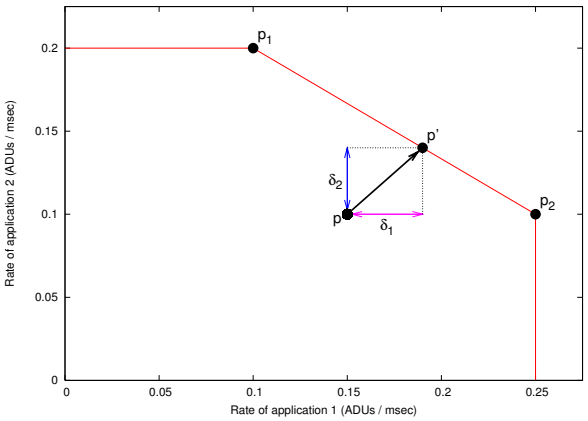

Figure 6. δ_1 and δ_2 for an example of 2 applications.

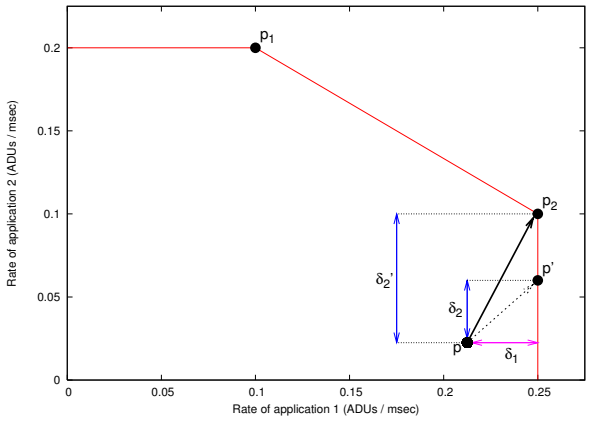

Figure 7. δ_1 and δ_2' for an example of 2 applications.

3.2. Identifying the Index Points

Find the maximum possible rate for each application. This step is equivalent to the max-flow problem. It is solved by constructing a linear problem. For each application app_q, $1 \leq q \leq Q$, we seek to maximize:

$$\sum_{c_i \in \mathcal{S}(app_q)} r_{c_i} \tag{7}$$

where $\mathcal{S}(app_q)$ are the input rates to the input components (the components that instantiate the first service of the application). This sum is equal to the total input rate of the application. To find its maximum, we maximize the above equation subject to the capacity constraints given in Equation (1) and the selectivity and flow conservation constraints given in Equation (2). By solving this problem for each application, we determine an appropriate component rate allocation plan for the respective point in the feasible region (as well as the point itself).

Finding the mid point. The set of points $\mathcal{P}$ created on the previous step are the vertices of a $(Q-1)$-simplex. We find the mid (average) point p_0 for this simplex, as well as the normal (perpendicular) vector to the simplex, d_0 [17].

Maximize the mid point. For the given p_0 and d_0, consider the line that starts from p_0 and moves towards d_0. Find the point that maximizes $\sum r_q$ on that line, subject to constraints (1) and (2). If this results in a point $p_1 = p_0$, no further action is needed, since the aforementioned simplex is a facet of the feasible region. On the other hand, if a point $p_1 \succcurlyeq p_0$ is found, there are two options:

1) If there is at least a point $p \in \mathcal{P}$ for which $p_1 \succcurlyeq p$, then p_1 will replace p in all associated simplexes.
2) Otherwise, a set of Q new simplexes are created each with all but one of the points of the original simplex, which is replaced by p_1. Each of those simplexes is then examined using the previous steps.

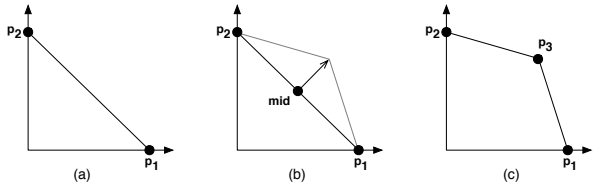

Figure 8. Finding the index points.

Example 3. *Consider the case in Figure 8(a), with $Q = 2$ and the maximum rates for applications 1 and 2 found, resulting in points p_1 and p_2. Then, we consider their mid point, from which we obtain index point p_3, as shown on Figures 8(b) and 8(c).*

Finally, we end up with a set of triplets of the form $< p, sol(p), \mathcal{F}(p) >$, where p represents an index point, $sol(p)$ represents the corresponding optimal component input rate allocation, while $\mathcal{F}(p)$ is the set of facets that p belongs to. For each index point p, let $proj(p)$ be p's projection to $\hat{\boldsymbol{\delta}}^{eq}$. Each triplet $< p, sol(p), \mathcal{F}(p) >$ is then indexed by $proj(p)$ on a spatial index [18]. This way, an *Index Point Database* is formed, which is utilized during runtime.

Note that the pre-calculation algorithm inserts into the Index Point Database not only the index points, but also some additional points, that happen to be the mid-points of facets at one point or another. As this does not affect the correctness of our solution, in the following we will use the term "index points" to refer to the set of points in the Index Point Database.

Using the Index Point Database, we partition the feasible region into two or more regions. Each region is the set of points $\{p\}$, the distance of the projections of which from the projection of an index point p_i is smaller than the distance of their projection from the projection of any other index point.

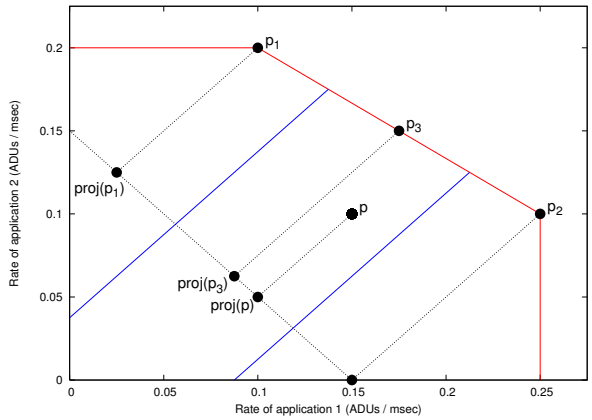

Figure 9. Index points, their projections and feasible region partitions.

Example 4. *The index points for the feasible region of Figure 4 are points p_1, p_2 and p_3 in Figure 9. Point p in that figure represents the current application input rates of the system. The index points partition the feasible region in three regions. Point p is in p_3's region. This is because, as shown in Figure 9, $proj(p)$ is closer to $proj(p_3)$ than to $proj(p_1)$ or $proj(p_2)$.*

4. Online Dynamic Component Rate Assignment

Let us assume a distributed stream processing system that is running Q applications, the input rates of which can be represented by a point p in the Q-sized Euclidean space. Upon the emergence of one or more bursts, the new input rates of the applications are represented by a new point p'. Our objective is to determine the appropriate component input rate allocation plan for p', that will maximize $\sum \delta_q$, with respect to the constraints presented in section 2.2. Our method is based on the following observations:

- The optimal component input rate allocation for any point p in the feasible region will be the same as the one for a point p' on one of the facets of the feasible region. For example, the optimal component input rate allocation for point p in Figure 6 is the same with the one for point p'.
- The optimal component input rate allocation for any point on the edge of the feasible region is the result of a linear optimization operation. Additionally, a point p that lies on a facet of the feasible region, can be expressed as a linear combination of the edges of the facet $p_1, p_2, \ldots p_Q$, *i.e.*, $p = a_1 \cdot p_1 + a_2 \cdot p_2 + \ldots + a_Q \cdot p_Q$, where $0 \leq a_q \leq 1$, $1 \leq q \leq Q$ and $\sum_{q=1}^{Q} a_q = 1$. Thus, the optimal component allocation for p is the linear combination of the respective solutions of p_1, p_2, ..., p_Q.

Given a new application input rate combination represented by point p while the system is running, the steps to find the optimal component input rate assignment $sol(p)$ are the following:

Find the index point closest to p'. We seek to find the index point p_i which is closer to p' than any other index point. This is realized by calculating the projection $proj(p)$ of p on $\hat{\boldsymbol{\delta}^{eq}}$. Notice that $proj(p) = proj(p')$. Let p_i be the index point stored in the Index Point Database, that is closest to p'. The triplet $< p_i, sol(p_i), \mathcal{F}(p_i) >$ is retrieved by making a nearest neighbor query for $proj(p)$ to the Index Point Database.

Find the appropriate facet of the feasible region. This is a facet f of the feasible region, on which the optimal solution $p' = p + \boldsymbol{\delta}$ lies. Facet f is one of the facets having p_i as one of their vertices ($f \in \mathcal{F}(p_i)$). This step is done by examining all facets in $\mathcal{F}(p_i)$, retrieved on the previous step. There are two cases on this step:

1) Let there be a facet f with edges $p_1, \ldots p_Q$, for which p' can be expressed as a linear combination of its edges, *i.e.*, $p' = a_1 \cdot p_1 + \ldots + a_Q \cdot p_Q$, $0 \leq a_q \leq 1$ and $\sum_{q=1}^{Q} a_q = 1$. Then p' lies on facet f and $\boldsymbol{\delta} = \boldsymbol{\delta}^{eq}$, *i.e.*, $\delta_1 = \delta_2 = \ldots = \delta_Q$. This is the case of Figure 6, where $p' = 0.5 \cdot p_1 + 0.5 \cdot p_2$.
2) If $p_i \succcurlyeq p'$, the facet that contains p' was eliminated during the pre-calculation phase and was never inserted to the Point Index Database. Thus, p_i is selected. This is the case of Figure 7, where p_2 is selected instead of p', since $\delta'_2 > \delta_2$.

Construct the optimal allocation for point p. The optimal allocation $sol(p)$ for point p is the optimal allocation $sol(p')$ for point p'. In the case where $\boldsymbol{\delta} = \boldsymbol{\delta}^{eq}$, since $p' = a_1 \cdot p_1 + \ldots + a_Q \cdot p_Q$, it is derived (from the linearity of the solutions) that $sol(p') = sol(a_1) \cdot sol(p_1) + \ldots + a_Q \cdot sol(p_Q)$. In the case where $p_i \succcurlyeq p'$, BARRE selects $sol(p_i)$.

5. Performance Evaluation

We implemented BARRE as part of our Synergy distributed stream processing system. We present the performance of our approach and test its scalability against different sizes of bursts and different number of applications. We compare our approach with several schemes as we describe next.

5.1. Experimental Setup

BARRE was implemented in about 15000 lines of Java. Service discovery and statistics collection were implemented using the FreePastry library [19], an open source

. Remember that all of f's edges are index points

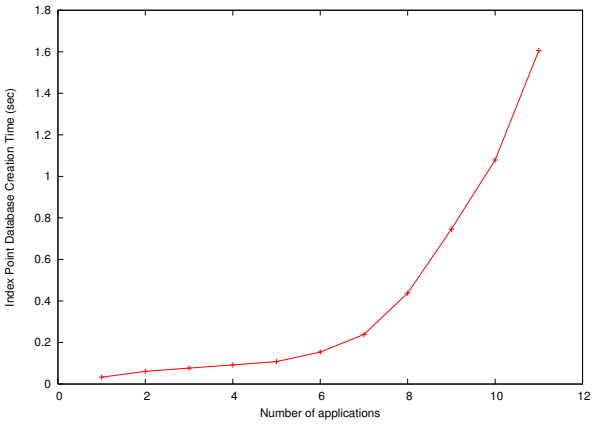

Figure 10. Index Point Database creation time.

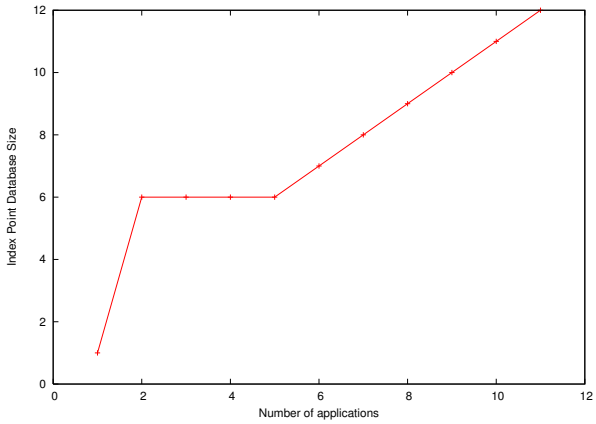

Figure 11. Index Point Database size.

implementation of Pastry. Complex matrix operations were carried out using JAMA [20]. A spatial index implementation from [21] was used to implement the Index Point Database. Each of the following results is the average of 5 runs, unless explicitly mentioned otherwise. Also, 90% confidence intervals are shown in all graphs, except in cases where these confidence intervals are too small or do not provide any insight. Each experiment was run with 11 applications, each containing 4 to 6 services.

We ran the experiments on a distributed stream processing testbed running Synergy in our lab. The experiments were run on a network of Debian Linux workstations, consisting of Intel Pentium 4 2.66GHz processors and Intel Xeon 3.06GHz processors, whose main memory varied from 1GB to 2GB of RAM. The workstations are interconnected with a 10/100 LAN, running Linux version 2.6.20. Our system is written entirely in Java. The employed JVM was Sun 1.6.0. We used the timing function provided by the JVM. The provided time granularity of $1msec$ was adequate for our experiments.

5.2. Experimental Results

BARRE Overhead. Figures 10, 11 and 12 demonstrate the time needed by BARRER to pre-calculate the rate assignment plans. Figure 10 shows the time needed for the construction of the Index Point Database, as a function of the number of applications in the system. Since adding an application corresponds to adding another dimension to our search space, the index creation time increases exponentially. One would expect similar delays to other functions of the Index Point Database. However, as explained in Section 3.2, only a limited number of points needs to be kept in the Index Point Database. The size of the index, shown in Figure 11, increases linearly with the number of applications.

This means that upon the occurrence of a burst, there is only a small number of index points in the database to choose from. Storing a small number of index points has a significant advantage during runtime, as shown in Figure 12. In that figure, the time needed to construct a new plan when a burst occurs, is shown. What is important is that the overhead due to combining multiple pre-calculated plans in order to devise a new one, instead of archiving a large number of pre-calculated plans, results in a significant speedup in the operation of the algorithm.

The following experiments demonstrate BARRE's responsiveness to bursts. There were 11 applications running on the system. At a specific time, the rate of all applications was increased by a certain percentage. The applications ran with the new rates and returned to their original rates after some time. We compared our results with (1) a simple **No Bursts Handling** algorithm that performs initial component rate assignment and that is burst-unaware, (2) a static **Reservation** method, that performs an initial component rate assignment in such a way so that 20% of the resources are reserved for future bursts on each node, (3) a **Dynamic Adaptation** algorithm that re-configures the system dynamically upon the appearance of a burst based on a linear optimization method (equivalent to the methods presented in [11], [12]) and (4) an algorithm that performs both **Dynamic Adaptation and Static Reservation**.

BARRE Operation. In Figure 13 we demonstrate the operation of BARRE. The figure shows the number of missed data units in our system for one of the applications when the burst intensity is 80%. The vertical axis shows the total number of data units that were dropped from the beginning of the experiment as a function of time. The results for the static methods are similar to the ones for the dynamic methods, with scaling being the only difference. They are not presented as they do not offer any insight. The following can be extracted:

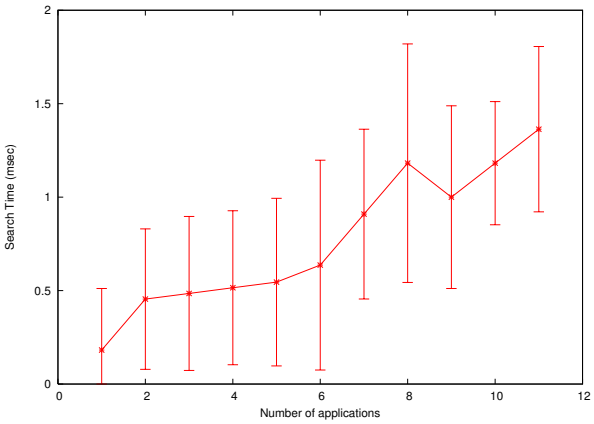

Figure 12. Index Point Database search time.

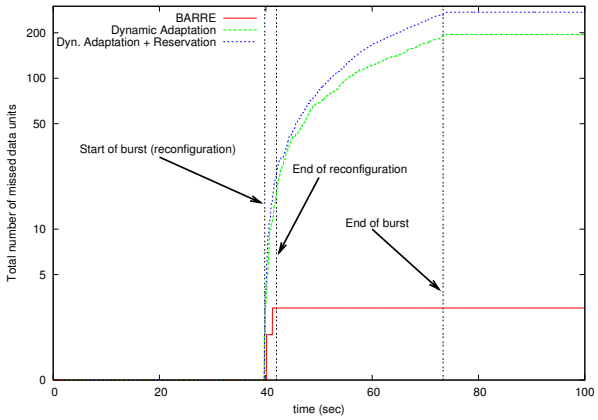

Figure 13. The time of data unit misses.

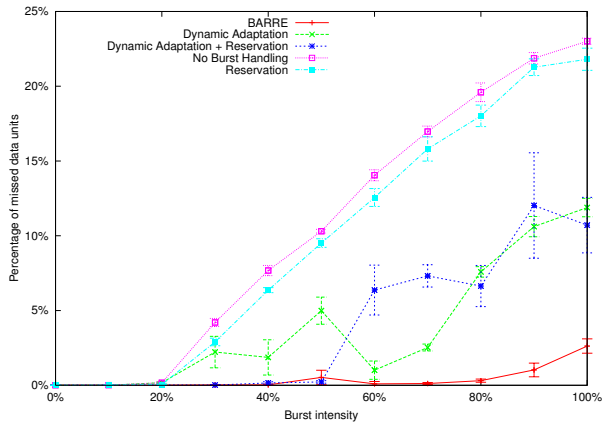

Figure 14. Percentage of missed data units.

1) BARRE ends up dropping far fewer data units than any of the other methods (note the logarithmic scale of the vertical axis).
2) The only time that BARRE can result in dropped data units is during reconfiguration, *i.e.*, the time after the start of a burst and until the system has finished reconfiguring according to the new plan. Not only is this time small (about $2.3sec$ in this example), the number of data units that are missed are very few as well, due to BARRE having provisioned for bursts.
3) BARRE employs a "safer" plan to tackle the burst than the one employed by the simple dynamic approach. The dynamic approach aims to optimize a linear objective, regardless of the effects on the load of the nodes. As a result, some nodes are 100% utilized during the burst. Thus, they are prone to drop data units due to unexpected short-term events.

Missed Data Units. The percentage of data units that were dropped is shown in Figure 14. As shown, BARRE performs much better than the other methods, resulting in fewer data units dropped. This results in BARRE being able to sustain an 80% increase in the rate of the applications without missing almost any data units. Other solutions can only sustain up to 20% bursts. At the same time, we can see that static reservation is not beneficial, as it reserves resources on nodes that are incapable of accommodating given bursts. On the other hand, BARRE's operation can be perceived as performing dynamic reservation of resources having in mind the workload imposed by the applications, not the load of each node individually.

Data Units Delivered On Time. An important issue when considering distributed stream processing is whether the data units are processed in a timely manner, *i.e.*, without any synchronization problems. The percentage of data units that were delivered in a timely manner is shown in Figure 15. It is clear that BARRE delivers almost all of the data units without delays in a timely manner.

Average End-to-End Delay. The average end-to-end delay of the data units is shown in Figure 16. A side effect of BARRE is that it decreases the end-to-end delay of the data units since it removes extraneous load from nodes with high processing capacity. The simple dynamic adaptation and the no burst handling methods end up overloading high capacity nodes so that they optimize the linear metric. On the other hand, static reservation does not help, since the amount of resources it reserves on a high capacity node can be needlessly high. Hence, too many data units are pushed to less powerful nodes, increasing the load of these nodes and (as a result) the average delay of each data unit.

6. Related Work

Distributed stream processing system have become increasingly popular in recent years for the development of distributed applications [13], [22], [23]. Recent efforts have studied the problem of resource allocation in distributed

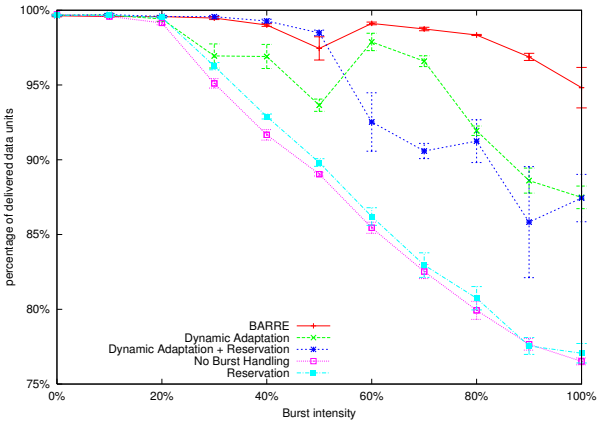

Figure 15. Data units delivered on time.

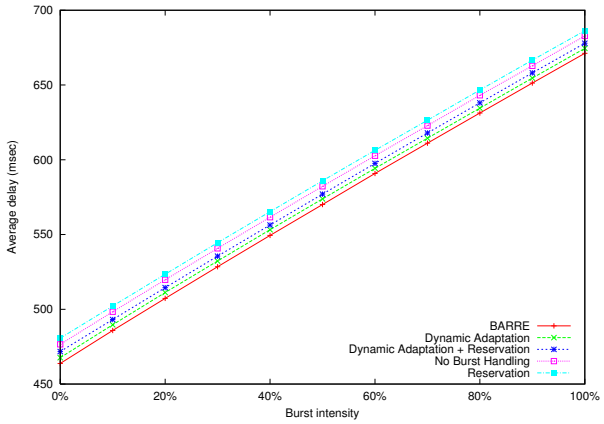

Figure 16. Average delay of data units.

stream processing environments [10], [22]. Most optimal service composition is accomplished in [22], using a probing protocol and coarse-grained global knowledge. The objective is to achieve the best load balancing among the nodes of the system, while keeping the QoS within requirements of the user. Our work targets maximizing the QoS of the offered services by fully utilizing the given resources of the system, with the node resource constraints of the nodes in mind.

We have previously investigated different aspects of overlays for distributed applications. In [24] we have focused on the task scheduling algorithm, while in [25] we have described a decentralized media streaming and transcoding architecture. In [13], we considered re-using components to improve overall efficiency.

In [26], the authors assume a distributed stream processing system similar to ours. They determine the optimal input and output rates, as well as the optimal CPU utilization for each component using a linear quadratic controller. Their ultimate goal is to maximize a global utility and they achieve that using Lagrange multipliers. Their methods rely on feedback from downstream components.

Authors in [27] formulate the problem of distributed stream processing as a utility optimization problem. They then apply a well-known network routing algorithm to allocate resources on stream processing nodes in an optimal way. Their method achieves optimal allocation of computing and bandwidth resources, however it needs time. This makes such methods inappropriate to address bursty input streams.

In ROD [28], authors consider stream bursts upon stream composition. They assign operators on a processing nodes in such a way so that the maximum possible input rate is supported for each operator. However, they do not consider distributing the computation for a single operator among two or more nodes, neither do they provide a means of dynamic re-configuration of the system in response to rate fluctuations.

Authors in [11] consider load shedding to avoid overloading in distributed stream processing systems. They propose a solution to optimally place and configure load shedding operators within an already given stream processing network. Their objective is to maximize the weighted query throughput. They address bursty traffic by re-configuring (dropping more or less) the load shedding operators in times of excessive load. The reconfiguration is based on redundant computations upon composition. Our solution avoids load shedding. Instead, we address bursts by assigning the execution of the excessive data units to alternative processing nodes.

System reconfiguration under the face of bursts is supported by MPRA [12]. However, the authors assume that there is a set of applications with more than one acceptable processing rates. Thus, in the face of bursts, the authors choose to modify the operation rate of such applications. Decreasing the processing rate is not allowed by applications considered in our system model. Our method avoids quality degradation by reassigning computation to nodes in response to load bursts.

7. Conclusions

In this paper, we considered the problem of accommodating unpredicted bursts of the data streams in distributed stream processing systems. We proposed an algorithm that proactively computes data stream allocations and uses them at runtime only upon the onset of a burst. Our method utilizes runtime statistics and the capacity of the nodes in order to handle sudden bursts in a timely manner. We have implemented our distributed stream processing burst accommodation technique on our Synergy distributed stream processing system. Our experimental results demonstrate the efficiency, scalability and performance of our approach

over techniques such as resource reservation and simple dynamic system adaptation.

References

[1] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "STREAM: The Stanford data stream management system," Mar. 2005.

[2] S. Chandrasekaran, O. Cooper, A. Deshpande, M. F. andJ.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *First Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, Jan. 2003.

[3] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager." in *Proceedings of the international conference on Very Large Data Bases*, Berlin, Germany, 2003, pp. 309–320.

[4] S. Madden and J. Gehrke, "Query processing in sensor networks," *IEEE Pervasive Computing*, vol. 3, no. 1, Mar. 2004.

[5] Y. Drougas and V. Kalogeraki, "RASC: Dynamic rate allocation for distributed stream processing applications," in *International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, CA, USA, Mar. 2007.

[6] Y. Drougas, T. Repantis, and V. Kalogeraki, "Load balancing techniques for distributed stream processing applications in overlay environments," in *ISORC*, Gyeongju, Korea, Apr. 2006.

[7] T. Repantis, Y. Drougas, and V. Kalogeraki, "Adaptive resource management in peer-to-peer middleware," in *WP-DRTS*, Denver, CO, Apr. 2005.

[8] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Systems*, vol. 27, no. 2, pp. 123–167, 2004.

[9] Y. Wei, V. Prasad, S. H. Son, and J. A. Stankovic, "Prediction-based QoS management for real-time data streams," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, 2006, pp. 344–358.

[10] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic load distribution in the Borealis stream processor," in *21st International Conference on Data Engineering (ICDE)*, Tokyo, Japan, Apr. 2005, pp. 791–802.

[11] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying FIT: Efficient load shedding techniques for distributed stream processing," in *International Conference on Very Large Data Bases*, Vienna, Austria, Sep. 2007, pp. 159–170.

[12] Y. Chen, C. Lu, and X. Koutsoukos, "Optimal discrete rate adaptation for distributed real-time systems," in *Real Time Systems Symposium*, Tucson, AZ, Dec. 2007.

[13] T. Repantis, X. Gu, and V. Kalogeraki, "Synergy: Sharing-aware component composition for distributed stream processing systems," in *Middleware*, Melbourne, Australia, Nov. 2006.

[14] E. W. Weisstein, "Polytope," http://mathworld.wolfram.com/Polytope.html, 2008.

[15] M. Geilen, T. Basten, B. Theelen, and R. Otten, "An algebra of pareto points," in *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD)*, 2005, pp. 88–97.

[16] E. W. Weisstein, "Facet," http://mathworld.wolfram.com/Facet.html, 2008.

[17] ——, "Normal Vector," http://mathworld.wolfram.com/NormalVector.html, 2008.

[18] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD Conference*, 1984, pp. 47–57.

[19] "FreePastry," http://freepastry.org/FreePastry, 2006.

[20] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington, "Jama: A java matrix package," http://math.nist.gov/javanumerics/jama, 2008.

[21] M. Hadjieleftheriou, "Spatial index library," http://research.att.com/∼marioh/spatialindex, 2008.

[22] X. Gu, P. S. Yu, and K. Nahrstedt, "Optimal component composition for scalable stream processing," in *25th International Conference on Distributed Computing Systems (ICDCS)*, Columbus, OH, Jun. 2005.

[23] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao, "Real-time component-based systems," in *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, San Francisco, CA, Mar. 2005, pp. 428–437.

[24] F. Chen and V. Kalogeraki, "RUBEN: A technique for scheduling multimedia applications in overlay networks," in *Globecom*, Dallas, TX, Nov. 2004.

[25] F. Chen, T. Repantis, and V. Kalogeraki, "Coordinated media streaming and transcoding in peer-to-peer systems," in *International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, Apr. 2005.

[26] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, 2006, p. 71.

[27] C. H. Xia, D. Towsley, and C. Zhang, "Distributed resource management and admission control of stream processing systems with max utility," in *Proceedings of the 27th International Conference on Distributed Computing Systems*, 2007, p. 68.

[28] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik, "Providing resiliency to load variations in distributed stream processing," in *Proceedings of the international conference on Very Large Data Bases*, Seoul, Korea, Sep. 2006, pp. 775–786.