UNIVERSITY OF CALIFORNIA
RIVERSIDE

Rate Allocation in Distributed Stream Processing Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Ioannis Drougas

December 2008

Dissertation Committee:

Dr. Vana Kalogeraki, Chairperson
Dr. Dimitrios Gunopulos
Dr. Mart Molle

The Dissertation of Ioannis Drougas is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I have had the good fortune of being around many remarkable individuals who have helped me complete this thesis. First, I would like to thank my dissertation committee members, Professor Vana Kalogeraki, Professor Dimitrios Gunopulos and Professor Mart Molle for their support and guidance.

I would like to extend sincere and very special gratitude to my advisor, Professor Kalogeraki, for being such an incredible mentor. Thank you for reading countless drafts of my papers, correcting numerous slides and verbal aspects of my presentations during our group meetings and providing invaluable comments and suggestions in such a timely manner. I have truly appreciated the exceptional research training that you have provided me, the confidence you have instilled in me, and all the advice you have given me throughout my five years of working with you.

I also want to thank my friends and colleagues from the computer science department - Vladimir Vacic, Dragomir Yankov, Thomas Repantis, Petko Bakalov and Kyriakos Karenos for all the encouragement and support. The completion of this dissertation would not have been possible without the help of these individuals.

Finally I would like to thank my family - my mom, Popi, my dad, Grigoris and my brother, Dimitris. I am truly grateful for all the sacrifices you have made for me. Your love and support made it possible for me to experience the journey of my Ph.D. study.

To my parents.

ABSTRACT OF THE DISSERTATION

Rate Allocation in Distributed Stream Processing Systems

by

Ioannis Drougas

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2008
Dr. Vana Kalogeraki, Chairperson

In today's world, 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. At the same time, service overlays that support distributed stream processing applications are increasingly being deployed in wide-area environments. The inherent heterogeneous, dynamic and large-scale nature of these systems makes it difficult to meet the Quality of Service (QoS) requirements of distributed stream processing applications. This has necessitated the investigation of mechanisms that improve their scalability, efficiency and performance. In the first part of this work we consider the problem of composing stream processing applications in a distributed stream processing system. First, we propose a distributed stream processing system that composes stream processing applications dynamically, while meeting their rate demands. Second, we address the load balancing problem for distributed stream processing applications and present a decentralized and adaptive algorithm that allows the composition of distributed stream processing applications on the fly across a large-scale system, while satisfying their QoS de-

mands. The algorithm fairly distributes the load on the resources and adapts dynamically to changes in the resource utilization or the QoS requirements of the applications.

In the second part, we present *BARRE* (Burst Accommodation through Rate REconfiguration), a method to address the problem of burst accommodation in a distributed stream processing system. Upon the emergence of a burst, BARRE dynamically reserves the 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 each node. Our experimental results over a real distributed stream processing testbed demonstrate the efficiency of our approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advances in processing and communication technologies have resulted in the development of *large-scale service overlays* (or *Peer-to-Peer* overlays). The new emerging Peer-to-Peer (P2P) model has become a very powerful paradigm for developing Internet-scale systems cycles, memory, storage space, network bandwidth) over large scale geographical areas. Peer-to-Peer overlays are logical networks of many nodes (peers), constructed on top of heterogeneous operating systems and networks. Such overlays are flexible and deployable approaches that allow users to perform distributed operations without modifying the underlying physical network. These have found popular applications in a number of domains including file sharing [12, 13, 35], content distribution [41, 36, 49, 46], multimedia streaming [34, 7, 24] and distributed services [45, 31].

During the past few years, a new class of 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 [4, 6]. 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 (*e.g.* selection of specific values or ranges of values, projection of specific attributes of the input), aggregation operators, or more complex operations, such as video transcoding. Such systems is possible to be implemented over distributed peer-to-peer overlays [6, 1, 24, 37]. In distributed stream processing applications, data produced by heterogeneous, autonomous and large numbers of globally-distributed data sources are composed dynamically to generate results of interest. These offer scalability and availability advantages by harnessing distributed processing elements in a cost-effective way. More advantages of distributed stream processing applications include their ability for customized delivery, for adaptation to different loads, and for resiliency to node failures. Distributed stream processing can also be applied to multimedia streams, to eliminate the need for a dedicated server with a high bandwidth connection and offer media services that can be composed on demand [28].

Processing of data streams brings significant challenges to the design of distributed stream processing systems: First, data streams are continuously produced in large volumes and high rates by external sources. The high rates of the streams along with their real-time rate requirements necessitate a highly scalable and adaptable stream processing system. Second, a distributed stream processing system consists of a number of nodes geographically distributed, where the functionality of a processing component is offered by a subset of the nodes of the system. Thus, stream processing in such a distributed stream processing system is achieved by combining components which are typically dispersed across the nodes

of the system. Third, computational and communication resources are shared by multiple concurrent and competing streams. Fourth, stream processing applications have inherent real-time requirements, in that the data must be delivered in a timely manner, *e.g.,* within a *deadline.*

1.1 RASC: Dynamic Rate Allocation for Distributed Stream Processing Applications

A number of stream processing infrastructures have been proposed in the literature, including Aurora[52], STREAM[4], TelegraphCQ[6] and the Cougar project[64, 39]. The majority of the current work has 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 [10] and have investigated composition and placement algorithms. The majority of these (including our previous work[15, 47]) focus on composition and placement techniques to manage and distribute the load equally across the system. However, many types of stream processing applications, such as media streaming, require that the data is processed and delivered to the user at a *required minimum rate.* For example, in a video streaming application, data needs to arrive to the destination at a rate high enough for the video to be properly presented and with small jitter. Allocating bandwidth among competing streams to satisfy application rate requirements is more complicated than performing "admission control". This requires that the processing of each data stream by individual application

components must be done at a required rate. In this work we propose such a distributed stream processing system designed to allocate and adjust rates to streams based on their rate requirements.

1.2 Load Balancing Techniques for Distributed Stream Processing Applications in Overlay Environments

Several characteristics make the provisioning of real-time and QoS support to distributed stream processing applications on large-scale service overlays a challenging problem. First, overlay nodes are typically heterogenous in terms of processor capacity, network inbound/outbound bandwidth, and software. Second, applications have multiple QoS demands including high throughput, small delay and jitter. Third, applications are composed at run-time, without a priori notification, posing stringent resource requirements on processor cycles and available network bandwidth along the streaming paths. As a result, the quality of the services may vary with time in an unpredictable way.

Current research does not focus on deciding the locations of the services when composing a distributed stream processing application dynamically to satisfy the end-to-end application QoS demands. This is a difficult problem in large-scale distributed environments, where well-established centralized solutions [38] cannot be applied directly. However, resource allocation becomes an important factor that affects the scalability of service overlays and the performance of the applications when deployed on a shared and heterogeneous infrastructure.

1.3 Burst Accommodation in Distributed Stream Processing Systems

Processing of data streams brings significant challenges to the design of distributed stream processing systems: First, data streams are continuously produced in large volumes and high rates by external sources. The high rates of the streams along with their real-time rate requirements necessitate a highly scalable and adaptable stream processing system. Second, a distributed stream processing system consists of a number of nodes geographically distributed, where the functionality of a processing component is offered by a subset of the nodes of the system. Thus, stream processing in such a distributed stream processing system is achieved by combining components which are typically dispersed across the nodes of the system. Third, computational and communication resources are shared by multiple concurrent and competing streams. Fourth, stream processing applications have inherent real-time requirements, in that the data must be delivered in a timely manner, *e.g.,* within a *deadline*.

The majority of composition and component placement algorithms in distributed stream processing systems (including our previous work [15, 47, 14]) focus on composition and placement techniques making the assumption that system and application characteristics remain constant through time. There is an increasing number of distributed stream processing applications that need to operate on highly dynamic environments. There are two reasons for that: First, distributed stream processing systems can be hosted on nodes along with other applications. These applications can incur substantial workload fluctua-

tions on the nodes. Second, the processing requirements of distributed stream processing applications are often highly variable, due to the nature of the applications. Such phenomena are called *bursts* of the input rate of the applications. As a result, bursts instantly increase the load of the nodes comprising the system. At the same time, relative deadlines of individual data units, are decreased. Decrease of relative deadlines arises since relative deadlines are inversely proportional to the input rate of the applications. Bursts are usually unexpected events that cannot be predicted in advance. For example, the input an application that performs network monitoring can increase considerably due to an unexpected DoS attack to many servers on the system. This can result in a large amount of data not being processed on time.

A common way to deal with such overload situations is to apply QoS degradation [57] or admission control in order to process part of the data units. The disadvantage of this solution is that admission control causes quality degradation, *i.e.*, some of the data are dropped due to the node's inability to process them. In a distributed stream processing system, it is preferred to address such overload situations by properly distributing successive load among the nodes of the system.

Resource reservation is another solution to address overloads [2]. Reserving an amount of the resources available on each of the nodes of the system is expected to address bursts by exploiting resources previously reserved. However, this results in under-utilization of the system and significant waste of resources when bursts do not actually happen. In addition, blindly reserving resources only has limited results. This is because an application typically needs for different types of resources and each one of them can become a bottleneck.

The dispersion of resources across many nodes needs also to be considered.

Another strategy is to predict overloads based on historical data about each application [63]. However, in many occassions, such data are not available. Bursts typically happen instantaneously due to reasons that cannot be predicted.

Finally, some solutions [9, 53] consider a user provided utility function which they try to optimize by dynamically changing system parameters. However, those methods optimize a linear utility metric. This strategy can make extensive use of some nodes to the favor of others. This can have a serious impact durin the start of the burst.

1.4 Our Contributions

In this thesis, we propose solutions to the aforementioned problems.

- In Chapter 2, we address the problem of instantiating a distributed stream processing application over a distributed stream processing system. Our method is based on (1) the resource requirements of the application, (2) the resources available on each of the nodes of the system and (3) the quality of service requirements requested by the user. Our algorithms employs *rate splitting* and runtime feedback from the nodes in order to minimize the amount of data that fail to meet the user's timing requirements.

- In Chapter 3 we address the load balancing problem in large-scale service overlays. We propose an adaptive and scalable load balancing technique for fair allocation of resources in large-scale service overlays, so that the QoS demands of distributed stream processing applications are satisfied. Our solution is based on a decentralized

algorithm that balances the load equally among the nodes of the system.

- In Chapter 4, we address the problem of burst accommodation in distributed stream processing systems. Our solution is based on reserving an amount of resources dynamically on each node. The criteria for the amount of reservation are the capacity of the nodes in addition to the requirements of each application and the likelihood of one or more applications overloading the system.

Chapter 2

RASC: Dynamic Rate Allocation for Distributed Stream Processing Applications

In this chapter, we present RASC (RAte Splitting Composition), a distributed stream processing system that performs composition of streams based on their rate requirements. Our system allocates and adjusts the rates of the streams based on the available processing capacity of the nodes and the bandwidth of the communication links. The goal is to minimize the number of data packets that fail to be delivered with the requested rate. This is an indication that some nodes are congested and that data packets may need to be dropped. A distinguishing characteristic of our approach is that it considers the number of the components required to perform a stream processing operation and the availability of the resources, and considers employing two or more instances of the same component on dif-

ferent nodes in order to split the processing requirements that would otherwise be assigned to a single processing component, to achieve the desired rate allocation. Our technique is entirely distributed and requires minimal knowledge in the form of the rate requirements of the streams and the congestion feedback from the nodes. We present detailed experimental results, over the PlanetLab testbed[44], that demonstrate the performance and efficiency of our approach.

2.1 System Model

In this section, we describe the model of our distributed stream processing system. First, we describe the system architecture. Then, we present the distributed stream processing application model. Finally, we describe the rate-based stream composition problem addressed in this chapter.

2.1.1 System Architecture

Our distributed stream processing system is illustrated in Figure 2.1. It consists of multiple nodes, connected in an overlay network. In our implementation we used the Pastry overlay network [49]. Each node in the overlay offers one or more *services* to the system. Each 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, that also includes state

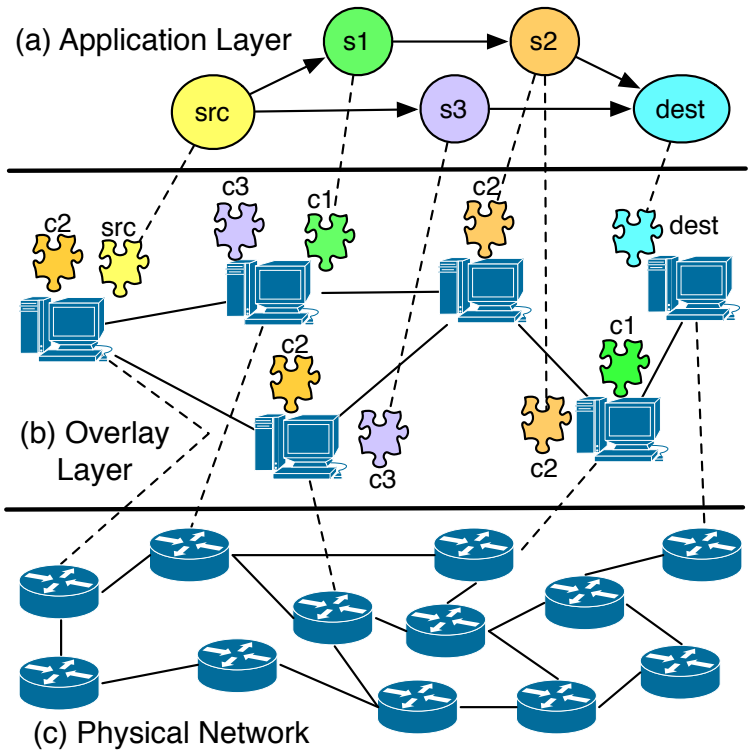

Figure 2.1: The architecture of a distributed stream processing system.

information about the service execution and the application for which the service is offered. 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 example, in an application that analyzes sensor network data). Nodes in our system are responsible for processing individual data units. The size of a data unit depends on the application. Upon reception of a data unit by a node, the data unit is put in a queue in order to be processed. To execute a data unit, the appropriate component is invoked. The role of a component is to receive data units created from other components in the network, process them and as a result, create data units that are forwarded to the

network for further processing. Thus, stream processing is performed by having one or more sequences of data units transferred through the overlay network and being processed by components hosted at the nodes of the network.

We consider that the nodes in our system are characterized by the resources they provide, (*e.g.* CPU or bandwidth). Let k be the number of resources of each node. Each node has a limited capacity for each resource and a component running on the node requires a fraction of that capacity. The amount of resources consumed by a component c_i is proportional to the arrival rate $r_{in}^{c_i}$ of data units arriving at c_i. We assume that all constraining resources can be measured in a rate-base. Examples of such resources include CPU cycles per second or bandwidth in bits per second. Let $u_j^{c_i}$ $(1 \leq j \leq k)$ be the amount of the jth resource required by component c_i when its arrival rate is equal to 1 data unit per second. The resource requirements of a component c_i, when its input rate is equal to 1 data unit per second, are represented by its *requirement vector* $\mathbf{u}^{c_i} = [u_1^{c_i}, \ldots, u_k^{c_i}]$. The requirements vector for a component only depends on the service that is executed by the component. Components operate on data units of fixed size[1] and operation on a single data unit is independent of operations on other data units. Thus, the resource consumption of c_i when processing r data units per second, is equal to $r \cdot \mathbf{u}^{c_i} = [r \cdot u_1^{c_i}, \ldots, r \cdot u_k^{c_i}]$. The amount of the jth resource available on a node n is represented by A_j^n, $1 \leq j \leq k$. The amount of resources available at node n is represented by the *availability vector* $\mathbf{A}^n = [A_1^n, \ldots, A_k^n]$.

[1]The data unit size is application dependent and should be set by the application designer.

2.1.2 Service Request Model

The user can request a stream processing application by submitting a *service request req for an application* to one of the nodes in the system. The request contains: (1) A *service request graph* G_{req}, like the one shown in Figure 2.2 and (2) the *rate requirements vector*, $\mathbf{r}^{req} = [r_1^{req}, \ldots, r_m^{req}]$ for the application that will be generated. The service request graph describes the components invoked by the application and the sequence of component invocation. A service request graph can consist of one or multiple *substreams*. Each substream is intended to be processed sequentially by a number of services. The rate requirement vector defines the delivery rate of data unit requested by the application $dest_{req}$, for each of the m substreams defined in the service request graph G_{req}. For example, there are two substreams in Figure 2.2: substream 1, to be processed by services s_1 and s_2 and then forwarded to the destination and, substream 2, to be processed by service s_3 before arriving to the destination. When submitting a request, the user expects from the system to *compose an application app* by invoking one or more services. The service request graph G_{req} specifies the combination of services invoked by the application. Each vertex of G_{req} represents the execution of a service s. One of the vertices of G_{req}, the *source* service src_{req}, represents the source of the data that needs to be processed. Another vertex of G_{req} represents the *destination service*, $dest_{req}$, which is the service that presents the results of the application to the user. A directed edge from a vertex that represent a service s_1 to a vertex that represents service s_2, means that the output of service s_1 needs to be forwarded to the input of service s_2, typically through the virtual link that connects the peers hosting the respective components.

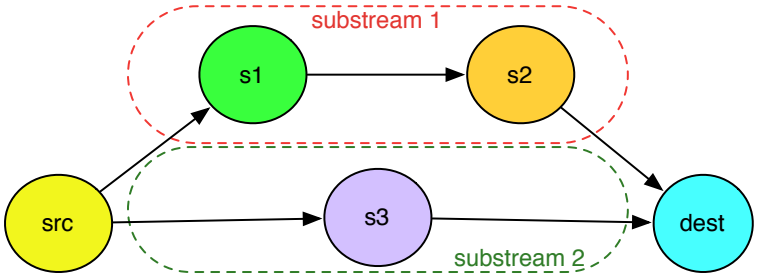

Figure 2.2: An example of a service request graph.

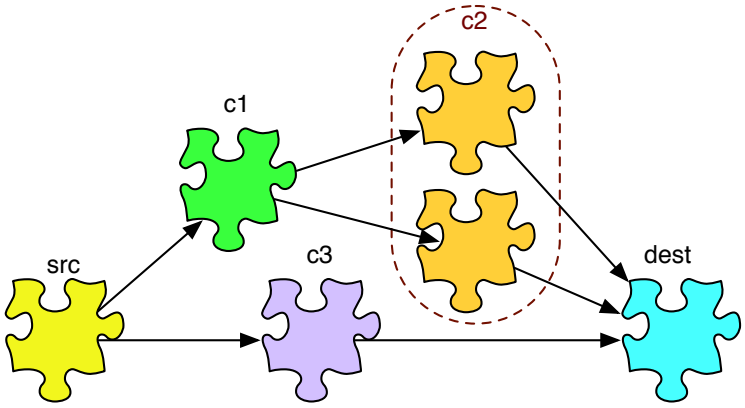

Figure 2.3: The execution graph represents the mapping of the service request graph on the overlay network in which one or more components can be used to instantiate a service.

The rate requirements vector $\mathbf{r}^{req}$ defines the *resource requirements* (CPU processing time, etc) of each component in order to process the data units as specified by G_{req}. It is important to note that since components are running individual operations, depending on the functionality of the components, the output rate of a component may be different from the input rate. For example, if a component c_i performs down-sampling of an audio input, then it is expected that its output will be forwarded to the next component at a rate $r_{out}^{c_i}$ lower than its input rate of $r_{in}^{c_i}$. This can be described in the service request graph, where each virtual link is characterized by a different rate, essentially, this would be the output rates of the components. In our current work we consider the case where the input and output rates of the components of the same substream are the same. However, our solution can be extended to consider the more general case where components can have different input and output rates. Components running intermediate services of substream l, $1 \leq l \leq m$, may experience higher or lower input rates than the rate defined in r_l^{req}, depending on the functionality each of them provides. We define the *rate ratio* $R^{c_i} = \frac{r_{out}^{c_i}}{r_{in}^{c_i}}$ as the ratio of the output rate to the input rate of component c_i. The problem is easier to be solved when $R^{c_i} = 1$. In the case that a data unit cannot be delivered at the requested rate (this happens when the data unit experiences processing or communication delays at the node because of queuing), the data unit will be dropped. This makes the problem of allocating the appropriate components more important, as the probability of dropping a data unit increases with the load of a node. The fraction of data units dropped by a component c_i is $drops^{c_i}$. In order to decrease the load incurred to individual nodes due to high input rates of components and thus to avoid dropping data units, it is possible to employ

a set of components $\{c_i^s\}$, at different nodes, for a service operation s_i requested by G_{req}. Such an example is shown in Figure 2.3 which represents the mapping of the service request graph of 2.2 in the overlay network. In this example, we employ two instances of service s_2, that is, two components c_2, running on different nodes. In such a case, each component processes a fraction of the data units to be processed for service s_2.

2.1.3 Problem Formulation

Our goal is, given a service request req, to compose an application that will execute the service requested by req so that it meets the application rate requirements $\mathbf{r}^{req}$. The application will be composed by instantiating the appropriate components on the nodes of the system. An *execution graph* will be the outcome of the composition algorithm. Essentially, this represents the mapping of the service request graph on the overlay network. If necessary, the execution graph should include more than one components per service of the request graph G_{req}, in order to satisfy resource constraints, to meet rate requirements, or to minimize the number of dropped data units. The composed execution graph will satisfy the rate requirements set by the application $\mathbf{r}^{req}$. The composition problem is defined as follows:

Definition 1 *Given a stream processing request* $req =< G_{req}, \mathbf{r}^{req} >$*, and a distributed stream processing system composed of a set* $\mathcal{N}$ *of nodes. The optimal rate-based composition problem is to find the execution graph that minimizes the total number of dropped data units while respecting the resource constraints on the nodes as well as the rate requirements:*

$$Minimize: \quad \sum_{c_i} r_{in}^{c_i} \cdot drops^{n(c_i)} \tag{2.1}$$

$$Subject\ to: \quad r_l^{dest} = r_l^{req}, 1 \leq l \leq m \tag{2.2}$$

$$\forall n \in \mathcal{N}, \sum_{c_i \in n} r_{in}^{c_i} \cdot u_j^{c_i} \leq A_j^n, j = 1, \ldots, k \tag{2.3}$$

$$\forall c_i, \quad r_{out}^{c_i} = R^{c_i} \cdot r_{in}^{c_i} \tag{2.4}$$

where $drops^{n(c_i)}$ *is the miss ratio of node* $n(c_i)$ *that hosts component* c_i, $u_j^{c_i}$ *is the amount of the* j*th resource consumed by component* c_i *when processing one data unit per second,* A_j^n *is the amount of the* j*th resource available on node* n, r_l^{dest} *is the resulting rate of the* l*th substream arriving at the destination given by* G_{req} *and* r_l^{req}, $1 \leq l \leq m$ *is the required rate of the* l*th substream arriving at the destination, given by the* l*th element of* $\mathbf{r}^{req}$. $r_{in}^{c_i}$ *and* $r_{out}^{c_i}$ *are the input and output rate of component* c_i *respectively, while* R^{c_i} *is the rate ratio of* c_i.

The goal in the above equation 2.1 is to minimize the number of data units dropped. Note, that, data units are dropped due to unexpected processing and communication delays at the nodes. This is an indication that the system is not able to process the data streams at the required rate. Thus, it is desirable to process data units at nodes with small drop ratio, so that we minimize the probability that data units are dropped. The above definition is a formulation of our problem as a minimum cost flow problem. Equation 2.3 expresses the capacity constraints of each link between two nodes in our system. Equation 2.4 expresses the *rate conservation requirement*: The output rate of each component should

be proportional to its input rate. Finally, equation 2.2 expresses the rate requirements of the user. Many of the metrics used in the above formulas, like A_j^n and $drops^{n(c_i)}$, change dynamically. As will be discussed in section 2.2.2, monitoring of these metrics is required.

2.2 Our solution

In this section, we present RASC (RAte Splitting Composition), a distributed stream processing system that dynamically composes applications while meeting their rate demands. RASC consists of: (1) a distributed component discovery mechanism to dynamically discover components at the nodes to compose the application, (2) resource monitoring techniques that keep track of the availability of the CPU and bandwidth resources, (3) a component scheduling algorithm that schedules component invocations to meet application timing requirements, and (4) the minimum cost composition algorithm, that composes applications dynamically based on the component and resource availability and the number of data units dropped.

2.2.1 Overview of RASC

Given a service composition request, RASC identifies the most appropriate set of components offered by nodes of the system, along with the rate with which data units should be sent to each of those components, in such a way so that (1) the rate requirement of the newly introduced stream processing application is met, while (2) the miss ratio of the application, *i.e.* the number of data units dropped as a result of the system being unable to maintain the requested rate, is minimized. RASC is based on reduction of the problem into

a minimum-cost flow problem. As a result, the rate assignment for individual components is integrated with the selection of the components themselves, rather than requiring a separate step. Minimum-cost flow is a well studied problem, for which well-established efficient solutions exist (*e.g.* [17, 21, 22])[2]. Thus, our solution manages to split the execution of individual services to more than one components, each of them running on a different node. Such distribution of components prevents exhausting the resources of individual nodes and increases the utilization of the system, making it possible to accommodate more service requests. This is because, contrary to other methods, RASC considers resources on multiple nodes while composing the execution of a stream processing request.

For each application composition, we follow these steps: (1) Discover the peers offering each of the requested services, using the underlying Pastry overlay. Determine all the possible component execution combinations for the application. (2) Obtain utilization information for the links towards the nodes hosting the above components. (3) Run the minimum cost composition algorithm in order to determine the configuration of the components and the rates. (4) Instantiate the respective components and run the stream processing application. Each step is discussed in the rest of this section.

2.2.2 Resource Monitoring

RASC can include various types of resources. In this work, however, we consider the constraining resources of a node to be the output and input bandwidth available (b^n_{out} and b^n_{in} respectively) for sending or receiving data from the node. Thus, $\mathbf{A}^n = [A^n_1, A^n_2] =$

[2]The minimum cost flow problem considers that the rate ratio of each component is equal to 1. In our problem definition, equation 2.4 does not consider this restriction. In the case where the rate ratio is not equal to 1, a linear programming method can be used to solve equations 2.1-2.4.

$[b^n_{in}, b^n_{out}]$. The input and output bandwidth utilized are calculated by continuously monitoring the rates of incoming and outgoing data units. To avoid miscalculations caused by transient behavior, we average the statistics over a window of size h, including the latest data units received. Additional performance statistics are obtained from the scheduling algorithm (presented in section 2.2.4). These statistics include: (1) The average running time t^{c_i} of a data unit processed by c_i, averaged over data units processed recently. (2) The rate of arrival $r^{c_i}_{in}$ for component c_i, also used by the scheduler in order to infer the *period* p^{c_i} of c_i. (3) The number $drops^{n(c_i)}$ of data units that were dropped by component c_i on peer $n(c_i)$, either due to insufficient resources (input queue size) on peer $n(c_i)$, or due to missed deadlines. Since the value of $drops^{n(c_i)}$ changes dynamically depending on the load of the peer and the rate requirements of the applications, it is essential to use feedback to monitor it.

2.2.3 Component Discovery

Prior to composing the new stream processing application, a node n needs to discover the nodes of the system that offer the required services. A service can be instantiated at one or more nodes in the overlay. Each component in the overlay has a unique ID, generated using a hash function (*i.e.,* SHA-1). The querying node can request a component by supplying its ID to the object discovery mechanism offered by the underlying overlay network. In our current implementation we use the Pastry overlay network[49]. Pastry is a decentralized, self-organizing overlay network, in which discovery messages are efficiently propagated among a large number of nodes. The expected number of forwarding steps in

the Pastry overlay network is $O(logN)$, while the size of the routing table maintained in each node at the overlay is only $O(logN)$. The querying node first generates the component ID and then uses the Pastry discovery mechanism to retrieve the list of hosts offering the requested component.

Given the hosts that offer a service, the next step is to retrieve the resource availability at each node, as well as the ratio of data units dropped at each host. In addition, each node monitors the arrival and departure rate $r_{in}^{c_i}$ and $r_{out}^{c_i}$ of each of its components c_i. Thus, it can easily infer its available input and output bandwidth, given their rates. Performance metadata is retrieved by requesting it directly from each host.

2.2.4 Scheduling Algorithm

A node n hosting a set C^n of components maintains a ready queue with all the data units to be scheduled at the node. The order of execution is decided based on running time and arrival rate statistics. Given the arrival time $arr_j^{c_i}$ of the jth data unit to be processed by c_i, the $(j+1)$th data unit is expected to arrive after time equal to the period p^{c_i} has passed. Thus, the expected arrival time of the $(j+1)$th data unit to be processed by c_i is: $arr_{j+1}^{c_i} = arr_j^{c_i} + p^{c_i}$. Generally, the execution of a data unit for component c_i has to be finished before the next data unit to be processed by c_i arrives. In a different case, it means that the system is unstable, having data units for c_i arriving faster than being processed. This means that the host cannot keep up with the incoming rate of c_i and that many of c_i's data units will arrive late to their destination (as they will keep piling up in the queue to be processed by c_i). In RASC, a scheduling strategy is designed, based on

this observation. To prevent this from happening, each jth data unit du that needs to be processed by c_i, is assigned a deadline equal to the expected arrival time $d^{du} = arr_{j+1}^{c_i}$ of the next data unit to be processed by c_i. The deadline d^{du} is calculated upon arrival of du at the host. Upon making the scheduling decision at time t, the laxity value $L(du) = t - (d^{du} + t^{c_i})$ is calculated for du. The laxity value represents a measure of urgency for the application. If the laxity value is positive, this indicates that the data unit will meet its deadline with high probability. If $L(du) < 0$, this means that du will most probably miss its deadline and thus, it is dropped. Out of the data units the laxity of which is positive, the one with the smallest laxity is chosen to be processed.

2.2.5 Minimum Cost Algorithm

Let us consider a service request req with a service request graph G_{req} and rate requirement vector $\mathbf{r}^{req}$ submitted to a node in our system. After discovering the available services and receiving the utilization feedback from the nodes, the next step is to determine the maximum rate that a node can offer to a component instantiated on it. We express the bandwidth constraints considered by our system in terms of the requirements and availability vectors used in equations 2.1-2.4. Since we consider the input and output bandwidth of a node to be the only constraints in our system, the requirements vector of a component c_i is given by $\mathbf{u}^{c_i} = [b_{in}^{c_i}, b_{out}^{c_i}]$, where $b_{in}^{c_i}$ and $b_{out}^{c_i}$ respectively represent the input and output bandwidth of the node hosting c_i consumed when c_i is receiving or transmitting at the rate of 1 data unit per second. The availability constraint of each node exists due to the available input and output bandwidth. Thus, the availability vector for

a node n is equal to $\mathbf{A}^n = [b^n_{in}, b^n_{out}]$, where b^n_{in} and b^n_{out} respectively represent the input and output bandwidth available on node n. Given the above resource requirements vector $\mathbf{u}^{c_i}$ for a component c_i and resource availability vector $\mathbf{A}^n$ for a node n, we determine the maximum rate $r_{max}(c_i, n)$ of component c_i when instantiated on node n using the following method: The maximum rate $r_{max}(c_i, n)$ is constrained by the most scarce resource of n (with respect to the requirements of c_i). This means that the maximum rate in discussion is equal to $r_{max}(c_i, n) = \min\left\{\frac{A^n_1}{u^{c_i}_1}, \ldots, \frac{A^n_k}{u^{c_i}_k}\right\}$. In our case, since input and output bandwidth are the metrics under consideration, the above formula means that the maximum rate of c_i is equal to the maximum rate that node n can transmit or receive (more formally, $r_{max}(c_i, n) = r_{max}(n) = \min\{b^n_{in}, b^n_{out}\}$).

The goal of the algorithm is to compose an execution graph for request req, given the peers that host each of the requested services, their utilization and the rate requirements. An example of an execution graph is shown in figure 2.3.

Example 2 *To better illustrate our method, let us consider a stream processing request with a request graph like the one in Figure 2.2. As discussed earlier, there are two different substreams in that request graph: One through services s_1 and s_2 and one through service s_3. Assume that service s_1 is offered by nodes n_3 and n_4, s_2 is offered by nodes n_1 and n_2 and that s_3 is offered by nodes n_1 and n_3 (Figure 2.4). In that case, the graph of Figure 2.3 is just one of many possible execution graphs that can be constructed for the stream processing request graph of Figure 2.2. All the different possible combinations are shown in Figure 2.5. In order to devise the execution graph, one has to consider all the component execution combinations shown in Figure 2.5. This makes the optimal composition problem*

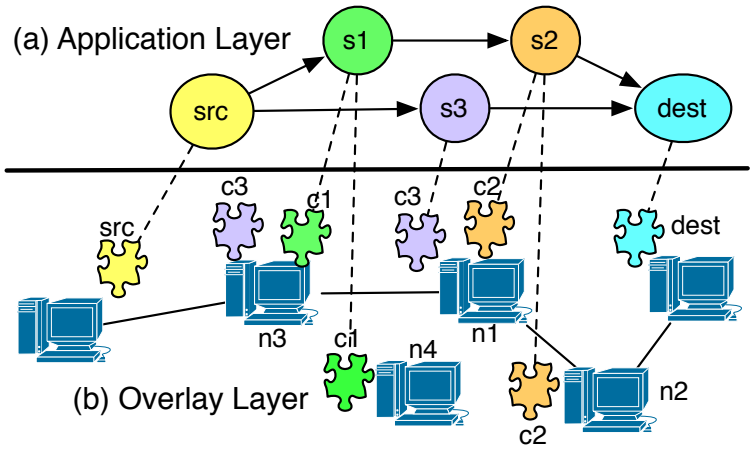

Figure 2.4: Composing a distributed stream processing application.

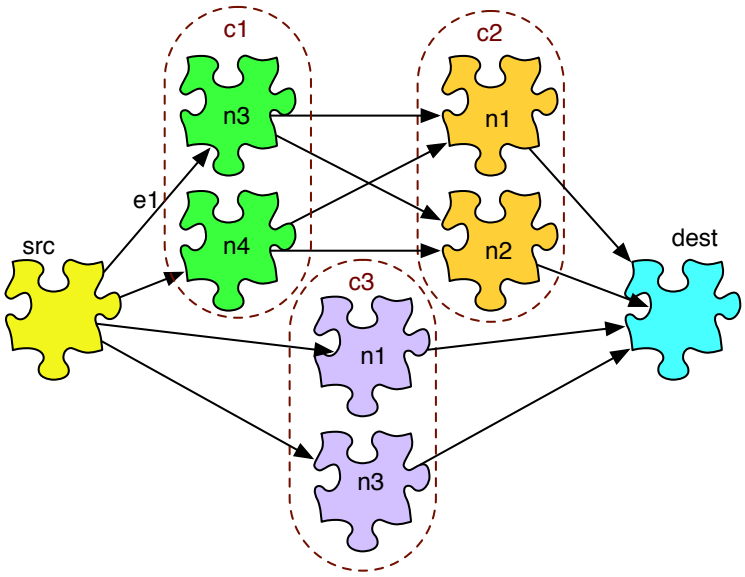

Figure 2.5: Component execution combinations for a distributed stream processing application.

too hard to solve with a naive approach, as the number of combinations is exponential with respect to the number of components. In addition to that, the rates on the edges of any devised execution graph can be also modified, making the problem even more complex.

We solve the aforementioned problem by repeating the following method in turn for each substream $substream_l$, $1 \leq l \leq m$ of the substreams in the request graph: For each edge e in the composition graph that refers to a connection for $substream_l$ we define the *capacity* cap^e of e to be equal to the maximum incoming rate of the node at the end of e. For example, in Figure 2.5, the capacity cap^{e_1} of edge e_1 is set to be equal to the maximum incoming rate $r_{max}(n_3)$ of node n_3[3]. In addition, we define the *cost* $cost^e$ of an edge e to be the ratio of dropped data units, observed during a specific time window. Then, the sub-problem for the lth substream is reduced to the minimum-cost flow problem [17, 21, 22]: We need to find a rate assignment for each of the edges of the composition graph that refers to the lth substream, so that: (1) The capacities of the edges of the composition algorithm will be taken into account (equation 2.3). (2) The sum of the rates arriving to the destination component by components running services of the lth substream of the service request graph, will be equal to the respective rate requirement r_l^{req} defined in the rate requirements vector $\mathbf{r}^{req}$ (equation 2.2). (3) The expected number of dropped data units, expressed by the weighted sum of the input rates of the respective components, will be minimized (expressed by equation 2.1).

After running the algorithm for the lth substream, the capacities on the edges of the composition graph are updated to reflect the assignment of components to nodes hosting

[3]In the case components on both ends of an edge e are to be invoked on the same node, the capacity of e is set to ∞.

Algorithm 1 CompositionAlgorithm

Discover the requested services using the DHT.

Gather the statistics about the relevant nodes.

for $l := 1$ to m **do**

Run the minimum cost algorithm for the lth substream.

if no acceptable assignment was found **then**

return error

end if

Update the node capacities.

end for

return the assignment found.

services that can be used by other substreams. After that, the algorithm is repeated for substream $l + 1$, until all the services requested by G_{req} are assigned to nodes. This way, the optimal rate assignment is efficiently performed. The composition procedure is shown in Algorithm 1.

2.3 Performance Evaluation

We present the performance of our approach and test its scalability against different input rates. We compare our approach with an algorithm that randomly decides on the placement of the component and a greedy algorithm that places each component to the node with the fewest dropped data units. All the experiments were run on a prototype implementation of RASC.

2.3.1 Experimental Setup

RASC was implemented in about 13800 lines of code in Java. Service discovery and statistics collection were implemented using the FreePastry library [16], an open source implementation of Pastry. Each of the results presented is the average of 5 runs, each on 4 nodes. The 90% confidence interval is also given on the results. There were 7 unique services in the system. Each node hosted 3.25 services on average. The average replication degree of each service was 1.86. A service request included 2 to 5 services, chosen randomly among the services available on the system. The service request rate was $100Kpbs$ to $250Kbps$.

We compared the minimum cost composition method of RASC with the following algorithms: (1) A *random algorithm* that does not take into account the capacity of the nodes when composing the execution graph. (2) A *greedy composition algorithm*, that iterates through components and places each of them to the node with the smallest drop ratio. Both of these algorithms considered the bandwidth capacity of the nodes.

2.3.2 Results and Analysis

Composed Requests: Figure 2.6 shows the number of requests that each algorithm was able to accommodate. Using the minimum cost composition algorithm, the system is able to compose many more applications, without violating the resource constraints of the nodes. This results in higher utilization of the system. Random and greedy composition methods are more vulnerable to rate increase, as they depend on the capacity of the most powerful nodes that offers the relevant services. Instead, minimum cost composition depends on the cumulative capacity of the nodes in the system, utilizing it most appropriately when needed.

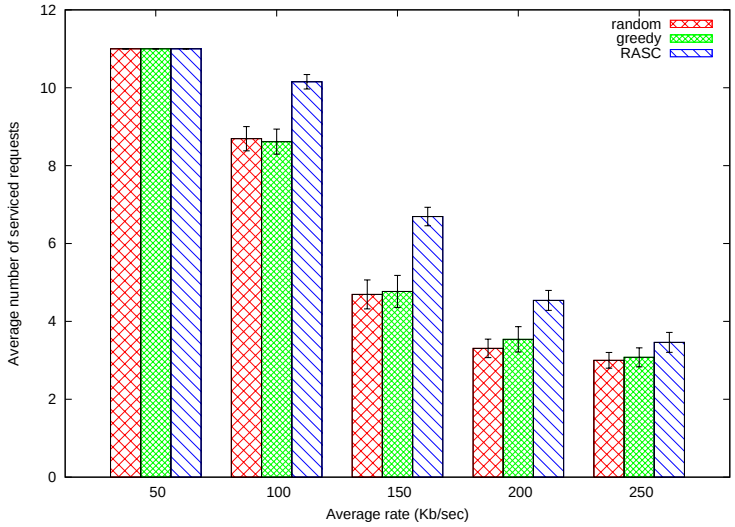

Figure 2.6: The number of requests that were successfully composed.

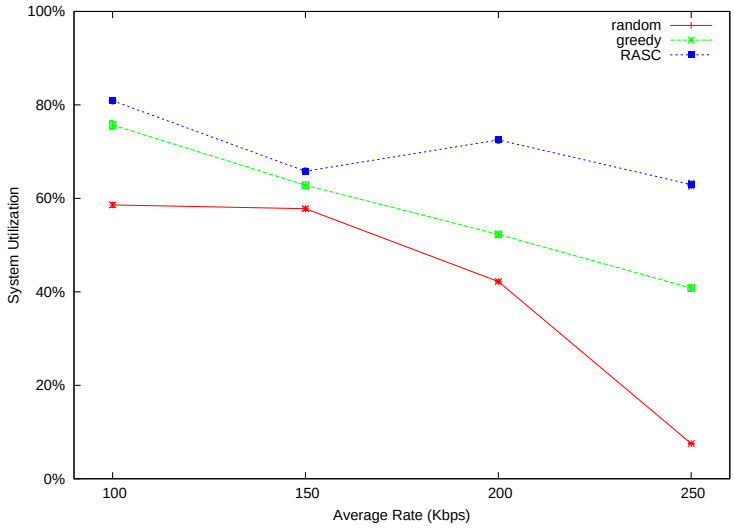

Figure 2.7: The average node utilization.

System Utilization: The purpose of RASC is to utilize the highest possible amount of resources on the nodes in order to accommodate as many streaming applications as possible. We measured the average node utilization of each node for all the given algorithms. The utilization was calculated for each node by dividing the average processing time on each node by the duration of the experiment. Figure 2.7 shows the average system utilization. As shown previously, increasing the input rate of the applications results in greedy and random not composing many of the requested applications. This results in fewer applications running in the system and hence, fewer data units being processed. On the contrary, RASC manages to keep the system utilization around 80%, due to the distribution of components.

Average End-to-End Delay: The average end-to-end delay of the data units is shown in Figure 2.8. The minimum cost composition used in RASC offers 23% to 41% improvement over the random composition algorithm and 27% to 44% improvement over the greedy approach. The reason of the poor performance of greedy approach is that in a single composition, it only calculates the miss ratio once. Thus, it keeps creating components on nodes with low miss ratio, until their maximum capacity is reached. Creating more than one components per service of the request graph, results in a big improvement for the minimum cost composition algorithm, since it is possible to share the load of computationally intensive services among many nodes. Note that using the minimum cost composition results in lower delays, despite experiencing higher system load than when random and greedy composition methods are used (since more applications are composed when using the minimum cost composition method).

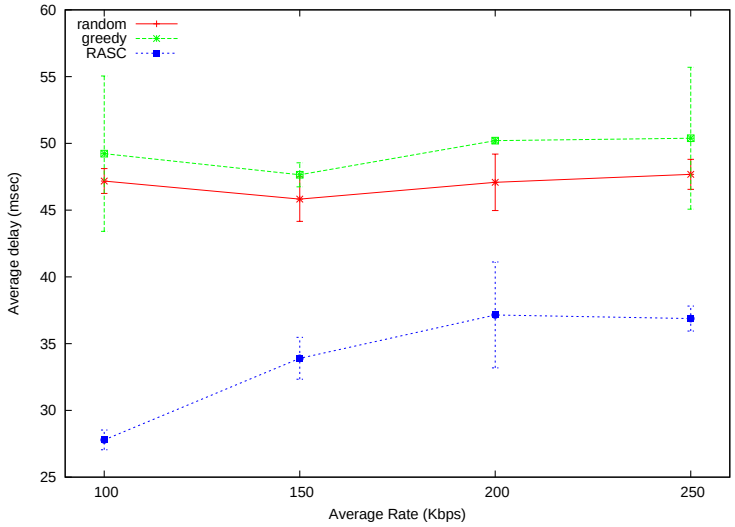

Figure 2.8: The average delay of the data units.

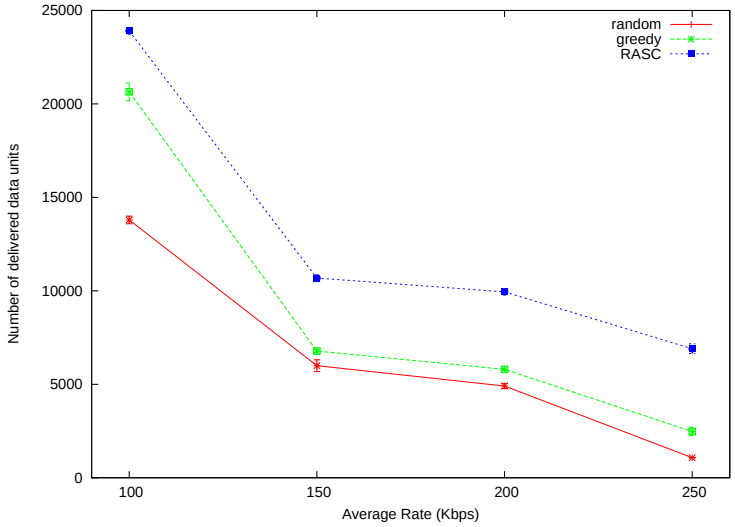

Figure 2.9: The number of data units that were delivered successfully.

Delivered Data Units: The total number of data units delivered successfully is shown in Figure 2.9. RASC manages to handle more of the data units presented to the system. This is because using minimum cost composition, the system managed to admit much more requests than the ones admitted when using the random or the greedy composition methods. This resulted in the system having to manage about double the load when using the minimum cost approach than when using the greedy or random composition approach. This is an immediate advantage of higher system utilization achieved by minimum cost composition: Computationally intensive services do not need to be instantiated on a single host. Instead, they are instantiated on more than one nodes. As a result, (1) services for which no node exists to provide enough resources, can still be accommodated by the system and (2) services that would significantly increase the miss ratio of a single node, are distributed across many nodes, that would otherwise remain idle. Once more, our solution performs better, despite servicing more applications.

Data Units Delivered On Time: A consideration against splitting a service in multiple components is the introduction of timing and synchronization problems among the data units of a service processed by different components. These would result in a reduced number of data units delivered in a timely manner. The number of data units delivered in a timely manner is shown in Figure 2.10. For a data unit to be delivered in a timely manner, it means that it has to arrive to the destination in order and in respect with the application's arrival rate requirement. As application input rates increase, the capacity of the system reaches its limits, thus forcing each of the algorithms to admit less applications.

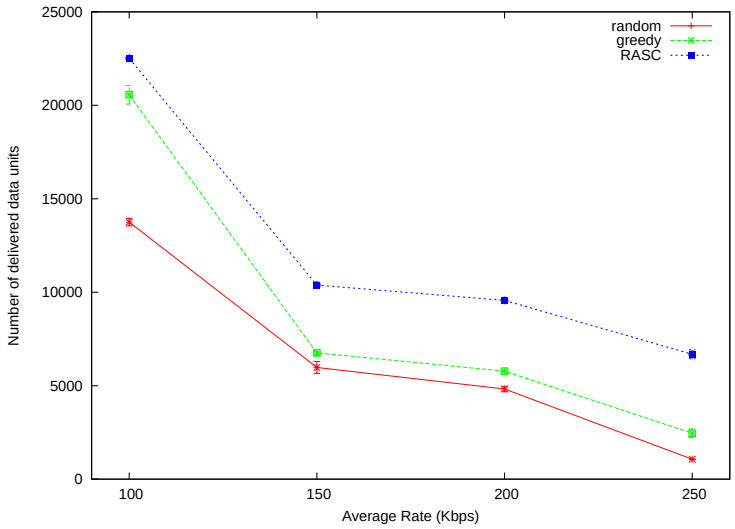

Figure 2.10: Number of data units that were delivered before their deadline.

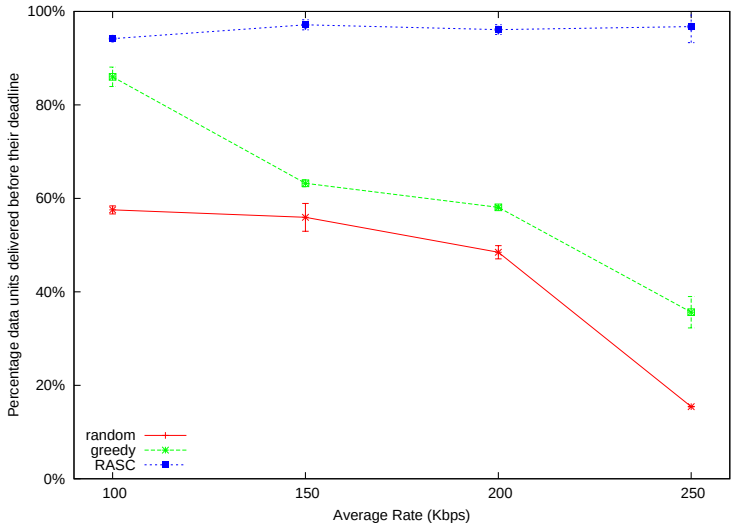

Figure 2.11: Percentage of data units that were delivered before their deadline.

Using RASC, delivery of more data units is achieved. The fraction of data units that were delivered in a timely manner (in respect to the total number of delivered data units of Figure 2.9), is shown in Figure 2.11. As we can see, RASC manages to deliver almost all of the data units in a timely manner.

Data Units Delivered Out Of Order: An issue to be considered is that data units could arrive out of order to their destination, due to differences in the performance of nodes that process data units of the same application. Figure 2.12 shows the fraction of data units that were delivered out of order (*i.e.*, later than some of the data units produced in their succession). When this number is too big, it means that there are some nodes on which data processing takes much more time that one would expect, resulting on the data units being processed too late in comparison to their counterparts processed by equivalent components. This does not happen, as this fraction remains below 6% for all algorithms.

Average Jitter: Jitter is presented in a stream processing application when a unit of a stream arrives at the destination later than the deadline set by the arrival of the data unit preceding it and the period set by the rate requirements. Jitter is the amount of time by which the data unit was delayed in respect to this deadline. It is undesirable, since it drops the requested rate and usually, it has annoying effects to the resulting stream delivered at the destination. Jitter is a metric of high consideration when designing video and other media streaming systems. Figure 2.13 shows that the minimum cost composition results in higher average jitter for each of the jittered data units. As shown, the average jitter for each data unit is about $0.5msec$. This is insignificant for most stream processing applications.

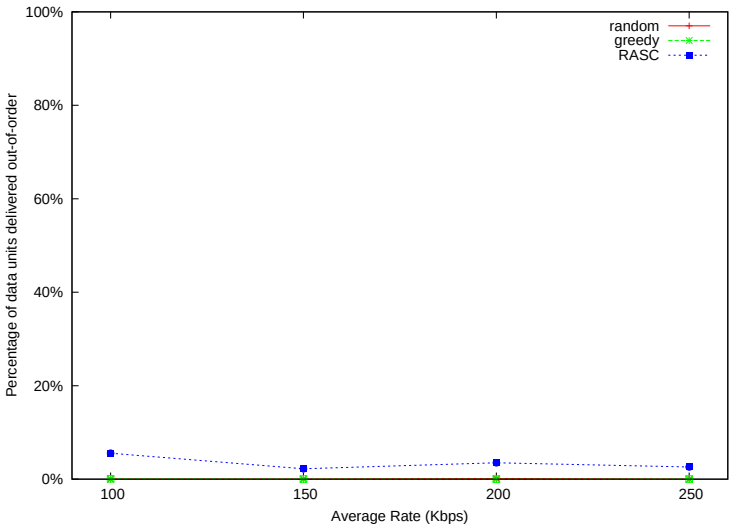

Figure 2.12: The fraction of data units that were delivered out of order.

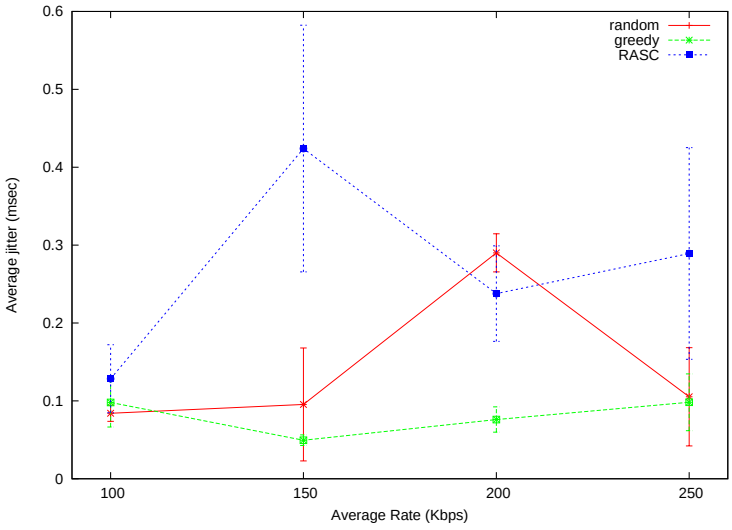

Figure 2.13: The average jitter.

Chapter 3

Load Balancing Techniques for Distributed Stream Processing Applications in Overlay Environments

In this chapter, we address the load balancing problem in large-scale service overlays. We propose an adaptive and scalable load balancing technique for fair allocation of resources in large-scale service overlays, so that the QoS demands of distributed stream processing applications are satisfied. Our solution is based on a decentralized algorithm. To react to dynamic changes in the resource utilization or the application behavior, quality adaptation mechanisms are used to trade off quality level with resource usage. We present

multimedia streaming and transcoding as a service example, but our techniques apply to any large-scale distributed stream processing application. Our approach allows composition of distributed stream processing applications dynamically, to satisfy their end-to-end QoS demands with high probability. We present experimental results, using a distributed media streaming and transcoding application over a large-scale overlay, to demonstrate the performance, efficiency and scalability of our techniques.

3.1 System Model

3.1.1 System Architecture

Service overlay networks are logical networks of nodes (peers) constructed on top of the physical network, in which the nodes are linked through virtual connections (Figure 2.1). Each peer p is identified by the IP address of the physical node it resides and the port it is listening to. Each node keeps a small number of connections to other peers; the number of connections is typically limited by the network bandwidth at the peer. The peers of a node can be randomly selected, defined a priori based on some optimization criteria (such as round-trip delays), or dynamically established and revised in response to the node interactions or changes in the processing and networking conditions [18].

Each peer p offers the set of services S_p and is constrained by the CPU speed $cycles_{total,p}$ and the limited amount of bandwidth $bps_{total,p}$ of the node. The communication link between any two nodes p and q is $band_{pq}$, where $\sum_q band_{pq} \leq bps_{total,p}$. This leads to a limit on the number of connections a node can maintain to other peers. We denote

the number of connections as $conn_p$. To compute the load of peer p, we consider both its processing load and network bandwidth. Thus, we compute the load of peer p as the weighted sum of its current processing and communication load, as follows: $l_p = w_c \cdot cpu_p + w_b \cdot band_p$, where $cpu_p = \frac{cycles_{used,p}}{cycles_{total,p}}$ and $band_p = \frac{bps_{used,p}}{bps_{total,p}}$ are the portions of the processing power and bandwidth currently being used at p. To give higher weight to the scarcest of the two resources (processing power or bandwidth), we define the weights as follows: $w_c = min(1, max(0, 0.5 + cpu_p - band_p))$ and $w_b = min(1, max(0, 0.5 + band_p - cpu_p))$. The weights w_c and w_b take values between 0 and 1 and have a sum of 1; thus, the weighted load $load_p$ of every peer p is between 0 and 1.

We assume that peers are organized in groups and in section 3.3 we show that balancing the load among the peers of each group performs comparably to balancing the load among all peers of the network. Peers can be grouped according to their geographical proximity, their network proximity, semantically, or even randomly. The organization of the network topology is outside the scope of this paper and several solutions have already been proposed [37, 47]. Similar to those, we assume that peers are grouped using some criterion and one or more peers in each group are responsible for resource allocation, as discussed in Section 3.2. The benefit of organizing the nodes into groups is that we can balance the load among the nodes of a group in a domain more efficiently. This is in contrast of using a centralized algorithm to balance the load among all nodes in the system. Such a choice would result in a single node being a central point of failure.

3.1.2 Application Service Graph

Distributed stream processing applications are modeled as sequences of invocations of services, which are executed in multiple nodes in the overlay network. We use a directed acyclic graph, which we call *application service graph*, to map a distributed application to the overlay network. The vertices of the application service graph represent the services being invoked at a set of peers to accomplish the application execution, while the edges represent connections between those peers. An edge connects two vertices v_i and v_j iff the output of the service corresponding to v_i is the input for the service corresponding to v_j.

Services implement functionalities that are performed at peers. A service s_i has a name, code, and input and output parameters. In order for a sequence of services s_i, s_j to be invoked for the execution of a task, the output of service s_i is directed to the input of service s_j, provided that they have the same parameters. Then s_i and s_j can be composed to form task $s_i \dot{} s_j$. This composition mechanism can be used to build more complex systems.

The application service graph is the outcome of the resource allocation algorithm, when an application execution request arrives. It is composed dynamically at run-time based on the application QoS requirements and the availability of the system resources and stored in all peers participating in the particular application.

The QoS of a service can be approximated using m discrete levels $Q : \{q_1, \ldots, q_m\}$. The resource requirements of a service s_i that provides a quality level q_j are $\{c_i(q_j), b_i(q_j)\}$, where c_i and b_i are the processor cycle and bandwidth requirements. Usually a higher QoS level means higher requirements in CPU cycles and bandwidth. The resource requirements can be approximately derived from the QoS requirements of the user using profiling [42].

3.2 Adaptive QoS-Aware Resource Allocation

The operation of our QoS-aware resource allocation approach consists of two steps: i) Selecting peers with available resources that offer the requested services to compose the application service graph, so that the QoS demands of the application are satisfied and fair allocation on the system resources on those peers is achieved (3.2.1). ii) Monitoring the resource usage and application behavior at run-time and dynamically adjusting the quality levels of the tasks, to improve their latencies or to react to changes in the behavior of the applications or the utilization of the system resources (3.2.2).

3.2.1 Resource Allocation

We describe the metric for fair load distribution in section 3.2.1, the data structure employed by our resource allocation algorithm in section 3.2.1 and finally the algorithm itself in section 3.2.1.

Fairness Index

The objective of the resource allocation algorithm is to equally distribute the load among the peers of the large-scale overlay. Because the allocation decisions are made by individual peers without global knowledge of the system and the resource loads, we need a metric that captures well the degree of uniformity of the load distribution and does not depend on scaling and magnitude factors. This metric is the *Fairness Index* [32].

Definition 3 *Let* $P = \{p_1, p_2, \ldots p_n\}$ *be a set of* n *peers. Let* $l_i \geq 0$ *be the load of peer* i*,* $1 \leq i \leq n$*. The Fairness Index of the load distribution* $\bar{l} = \{l_1, l_2, \ldots l_n\}$ *among the peers in*

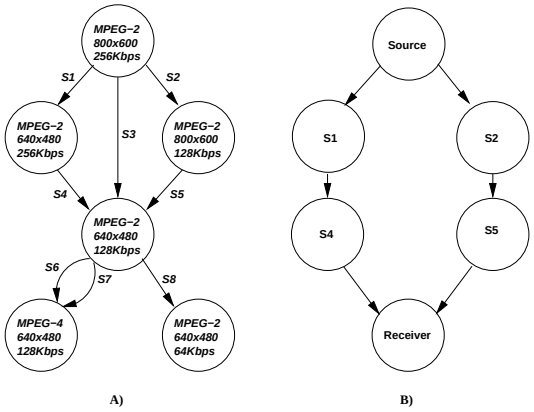

Figure 3.1: An example of a service composition graph (A) and of the produced application service graph (B).

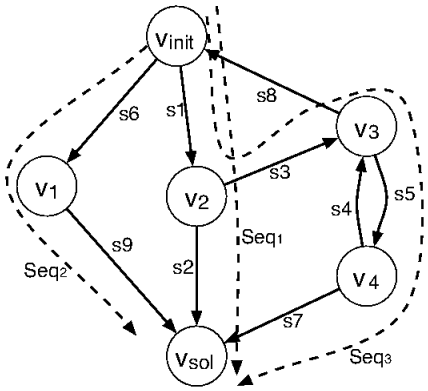

Figure 3.2: Example of a service composition graph.

P is defined as:

$$\mathcal{F}(\bar{l}) = \frac{(\sum_{i=1}^{n} l_i)^2}{n \cdot \sum_{i=1}^{n} l_i^2} \tag{3.1}$$

The Fairness Index has some useful properties that make it an ideal metric to evaluate the fairness of a load distribution. Its values are bounded in the interval $(0, 1]$, and are proportional to the uniformity of the distribution. It is population size and metric independent, it can be applied to any number of nodes and the unit of measurement does not matter. Thus, the Fairness Index can be applied directly to a large-scale overlay, no matter how many peers it incorporates or how loaded they are.

Service Composition Graph

The *service composition graph*, illustrates the possible application service compositions in a group of peers and then select the one that best meets the objective of the resource allocation algorithm. The vertices of the service composition graph represent specifications of service inputs or outputs, while the edges represent services offered by peers in the group. An edge connects two vertices v_i and v_j iff there exists a peer that offers a service that takes input of the form specified by v_i and produces an output of the form specified by v_j.

The service composition graph is stored at the peers of a group that run the resource allocation algorithm and is used by the resource allocation algorithm (1) to compose new applications, or (2) periodically to improve the latency of the tasks or to react to processor or resource faults. It refers just to the services that are available in one peer group and is therefore of bounded complexity. It is composed gradually, as peers and services are added to a group and the new connections are detected.

For example, figure 3.1(A) shows a service composition graph for a system that offers transcoding services. The transcoding quality level and the corresponding resource requirements can be tuned by selecting different implementations for the transcoding functions. Let us assume a user is interested in receiving a video in MPEG-2, 640x480, 128Kbps. Yet, the video is provided by the source in MPEG-2, 800x600, 256Kbps. One can see that we can follow three different paths to go from <MPEG-2, 800x600, 256Kbps> to <MPEG-2, 640x480, 128Kbps>. The choice of which path to use depends on the current resource conditions. Figure 3.1(B) shows an application service graph that could be produced by this service composition graph.

Resource Allocation Algorithm

The procedure followed by the resource allocation algorithm is the following: First, it finds all paths representing the desirable application using the service composition graph. For each of the paths, it determines whether the quality of service requirements are met. Only the paths representing a solution with which these requirements are met, are considered. Then, the algorithm determines the candidate application service graphs to construct, by mapping each of the paths to the overlay network. For each of the application service graphs, the resulting Fairness Index is estimated. Finally, it selects the service graph for which the Fairness Index is maximized.

Let us assume a service composition graph G_c, as shown in Figure 3.2. Let v_{init} be the application composition request given by the user (*i.e.,* specifications of the data to be processed) and v_{sol} represent the solution that satisfies the QoS requirements of the user

(*i.e.,* specifications of the output data). Each possible solution to the resource allocation problem can be represented as a path Seq_i from v_{init} to v_{sol}. Essentially the path represents a service composition sequence where each edge s_k in the path represents the invocation of a service. If there are multiple possible paths from v_{init} to v_{sol} that satisfy the user's QoS requirements, our goal is to select the one for which the fairest distribution of the load among the peers is achieved. For example, the graph in Figure 3.2 shows that there are three possible service invocation sequences (paths in G_c) $Seq_1 = \{s_1, s_2\}$, $Seq_2 = \{s_6, s_9\}$ and $Seq_3 = \{s_1, s_3, s_5, s_7\}$. If any of the possible service invocation sequences does not meet the QoS requirements due to processing or bandwidth limitations, these sequences will be ignored.

For each service invocation sequence , the algorithm evaluates the Fairness Index for the corresponding application service graph. This is achieved by calculating the contribution to the load on processor p made by the allocation of service s_j. The service load is derived from the size and specifications of the service input and output, the mean processing time and bandwidth required for the invocation of the service, and the quality of service requirements set by the user for the service. The algorithm then selects the sequence Seq_i that results in the maximum Fairness Index value.

The algorithm identifies the best solution but works in exponential time, because it evaluates all possible solutions. To improve the running time, (1) we avoid examining unacceptable solutions, by constantly checking the QoS, (2) we avoid looping through already examined outputs, and (3) we return as soon as a Fairness Index value greater than a threshold ϵ is found.

3.2.2 Coordinated Quality Adaptation

Our resource allocation algorithm composes services in such a way that the user QoS demands and their requirements in processor cycles and bandwidth can be accommodated. However, in a dynamic environment the resource loads of the nodes can vary at run-time, degrading the performance of existing tasks. Therefore, adaptation mechanisms that gracefully adapt to changes in the resource usage or the needs of the applications are required. Our approach [8] is to employ a quality adaptation mechanism, which trades off service quality level with resource usage. Locally adapting the quality of a service, might result in quality fluctuations of the end result. coordination of the local adaptation decisions based on feedback generated by the receiver of a composite service. Coordinated quality adaptation is triggered when the load on a peer or the latency of a task is too high, or when the user QoS requirements change at run-time. In such cases, reassignment of a task to resources and thus reconstruction of its application service graph might also be needed and can take place by running the resource allocation algorithm again. The cost of the application service graph recomposition is amortized over many application executions.

3.3 Experimental Evaluation

3.3.1 Experimental Setup

To evaluate the performance of our resource allocation techniques, we implemented in C++ a simulator for a media streaming and transcoding application over an overlay network. The underlying network topology we used was generated with GT-ITM [65],

Simulation Time	200 seconds
Receiver Buffer Time	2 seconds
Quality Interval Size	3
Media Object Bitrate	(200, 250, 300, 350, 400) Kb
Req. Streaming Bitrate	(50, 75, 100, 125, 150) Kb
# of Total Nodes	1000
# of Source Nodes	400
# of Transcoder Nodes	80
# of Peer Groups (avg.)	10
# of Streams	70
# of Substreams	232

Table 3.1: Simulation parameters.

consisted of 1476 routers, and had an approximate diameter of 750ms. Overlay nodes were randomly attached to different routers. Media sources had a connection bandwidth between 50Kb and 200Kb, while media transcoders had a connection bandwidth between 400Kb and 2Mb and a processing capability between 400M and 800M cycles per second. To take into account the fact that the resources of the system are not dedicated, we randomly added a fluctuating percentage (up to 20%) of extraneous load in the processors and of cross traffic in the network, throughout the execution of the experiment.

We simulated a transcoding application, in which the execution time of an operation was proportional to the size of a media unit. Data transformation operations on independent media units (*i.e.* groups-of-pictures of MPEG streams) were considered to be the services that needed to be allocated to peers. We simulated 10 levels of quality for all streams and utilized the same linear utility function that was proportional to the output quality level. The resource allocation algorithm was executed when an application request arrived in a peer group, while the local adaptation algorithm was executed every second.

We compared our adaptive and *fair* resource allocation algorithm with threshold $\epsilon = 0.8$ against a *random*, a *greedy*, and an *optimal* allocation algorithm. The random algorithm assigned service requests to transcoders blindly. The greedy algorithm searched among the list of the transcoders of a peer group to assign a request to a transcoder that can offer the required bandwidth and processor cycles and returned the first solution it found. The optimal algorithm assigned service requests trying to maximize the fairness of the whole network. Thus, it assumed a central approach, where global knowledge of the loads of all the peers in the network is attainable.

3.3.2 Results and Analysis

QoS under different loads

In the first set of experiments we investigated the behavior of the resource allocation algorithms, by analyzing the system's performance under different loads. We increased the number of streaming sessions gradually and were thus able to evaluate the scalability of the different algorithms.

Average End-to-end delay. Figure 3.3 illustrates the average end-to-end delay for all the media units of many streaming sessions. The figure shows that fair resource allocation achieves the lowest end-to-end delay, regardless of the number of streams in the system. Moreover, it maintains a bounded delay, proving that it can offer a scalable solution to the resource allocation problem, as long as the requested resources can be offered by the system. Randomly selecting transcoders results in extreme delays as the load of the system increases. Greedy resource allocation achieves bounded delays as well, since it takes into account the

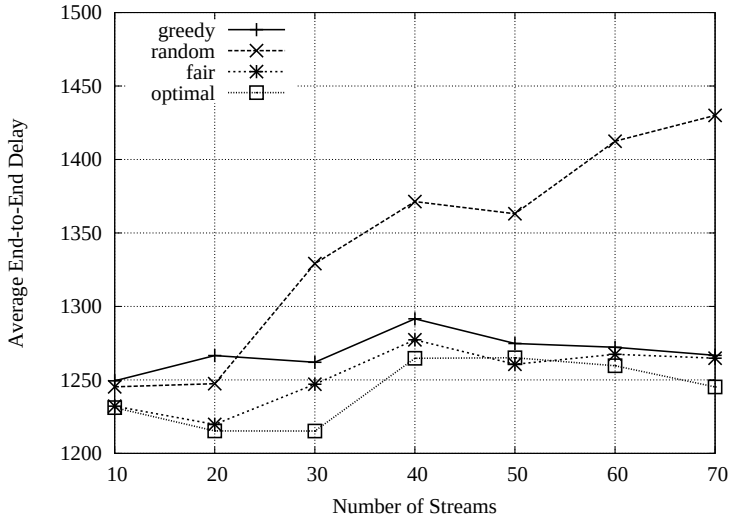

Figure 3.3: Average end-to-end delay of the media units of all streams vs. the number of streams.

required resources when assigning tasks to the transcoders. Yet, even though the media units arrive with relatively small delays they do not necessarily meet their deadlines, as will be shown in figure 3.4. It is noteworthy that our fair allocation algorithm achieves average end-to-end delays very close to those that would be achieved by an optimal allocation.

Media units with missed deadlines. Figure 3.4 shows the degree to which our fair resource allocation algorithm can help the system meet its QoS guarantees (the timing deadlines in particular). Inevitably, as the load becomes more than the system's resources can handle, media units will miss their deadlines. Our fair resource allocation algorithm postpones those negative effects for as long as possible and even then results in considerably less missed deadlines than all other allocation algorithms. As expected, random resource allocation results in missed deadlines even for low loads and extreme numbers of missed deadlines as the load increases. The number of media units with missed deadlines for the

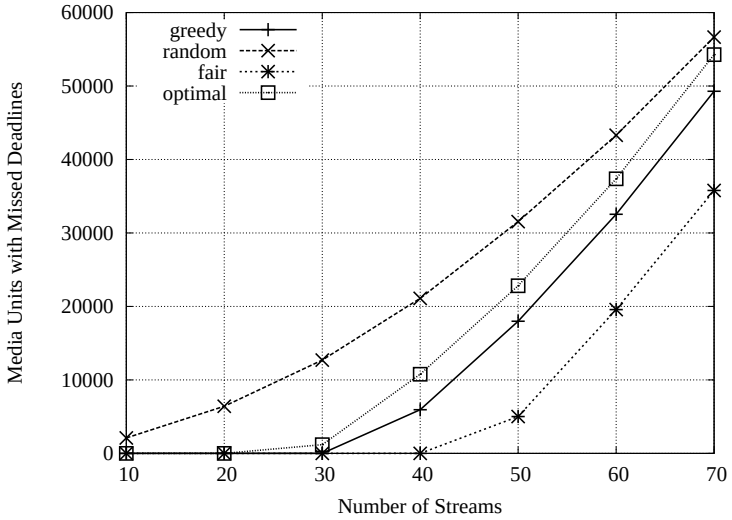

Figure 3.4: Media units that have missed their deadlines vs. the number of streams.

greedy allocation algorithm shows that just provisioning for the required resources does not suffice for achieving QoS guarantees, and that a more intelligent load distribution mechanism can have better results. The optimal allocation algorithm results in many missed media units from an unexpectedly low number of streams. This is to show that distributing the streaming and transcoding load fairly across the system is not as efficient as doing the streaming and transcoding locally, within the peer group in which the receiver belongs. This way the streams can reach their destination faster.

System utilization under different loads

In the second set of experiments we observed parameters related to the system utilization. Again, we increased the number of streaming sessions gradually and focused on the scalability of the different algorithms.

Average fairness. As was explained in section 3.2, how uniformly the load is divided

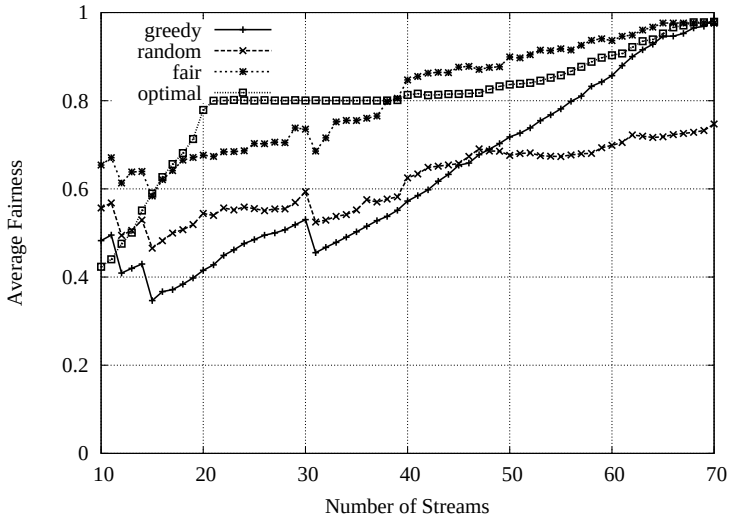

Figure 3.5: Average fairness of the load distribution across all peers, as a function of the number of streams.

among the processing nodes can affect the efficiency of the system. In figure 3.5 we present the average of the fairness indices of the individual peer groups, as new streams are admitted in the system. The fair resource allocation algorithm always achieves a more even distribution of the load. As the number of streams is increasing the average fairness increases as well, since tasks can be assigned to all transcoders. When the system becomes overloaded, greedy resource allocation also results in high average fairness, since the load is distributed among all transcoders and the selection of where to place tasks is of no particular importance anymore. Yet, randomly placing tasks, without paying attention to the current load of the nodes, results in low fairness even in those overloaded situations. The fairness of the optimal allocation algorithm is not directly comparable to the rest, since it compares the load distribution among all nodes of the system, instead of averaging the fairness indices of the individual peer groups.

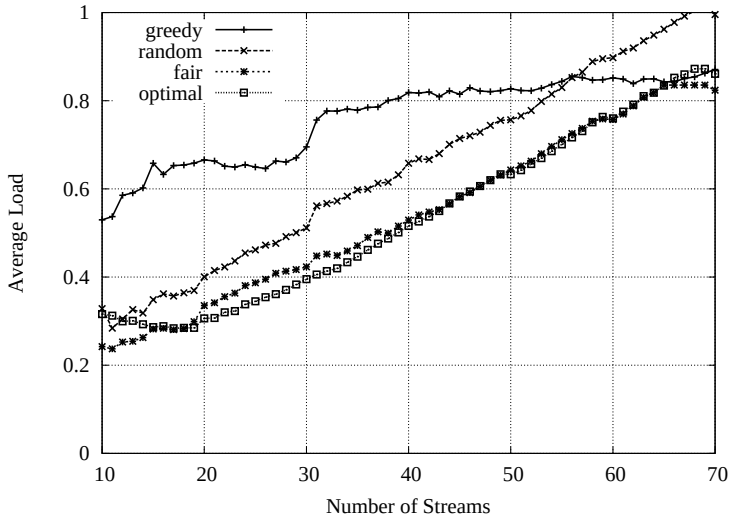

Figure 3.6: Average load of the system, as a function of the number of streams.

Average load. Figure 3.6 shows the average of the average loads of each peer group, only for transcoders that have been assigned tasks. We can see that fair load distribution within a peer group results in lower average system load. The average load will not be equal in all situations. For example, a distribution of tasks in transcoders in peer groups of $[(0.4), (0.4, 0.4)]$ results in an average load of 0.4, while a distribution of $[(0.8), (0.2, 0.2)]$ results in an average load of 0.5. Having low average load is useful, as it will avoid the overloading of certain peers, while other peers remain underloaded. To provision for emergency situations, certain nodes can be excluded from the resource allocation procedure, to always be available. The greedy resource allocation algorithm results in average load even higher than the random, since the same nodes are likely to be allocated more tasks, as long as they can accommodate them. The average load achieved by the optimal allocation refers to the whole system and not to the average of the average loads of each peer group.

Chapter 4

Burst Accommodation in Distributed Stream Processing Systems

In this chapter, we present *BARRE* (Burst Accommodation through Rate RE-configuration), a method to address the problem of burst accommodation in a distributed stream processing system. BARRE dynamically reserves the resources dispersed across the nodes of the system, based on the requirements of each application as well as the resources available on each node. This is done by allocating the input rates of individual processing components in such a way so that (1) current application requirements are met and (2) nodes are not over-utilized. As soon as a burst is detected, BARRE examines whether intervention is needed in order for the burst to be accommodated. It then modifies the input rates of individual components as needed. Our method's goal is to guarantee that

the maximum possible increase on an application's rate will be able to be accommodated. This is in contrast to other solutions that perform dynamic reconfiguration, in that we do not target to optimize a user-provided utility function. In the following, we formulate the problem of burst accommodation, provide a solution for that and evaluate this solution in a real distributed stream processing system.

4.1 System Model

In this section we provide the background and formulate the problem of burst provisioning in distributed stream processing systems. First, we describe the architecture of our system. Then, we present the streaming application model as well as burst model for the input rate of the applications.

4.1.1 System Architecture

Our distributed stream processing system is illustrated in Figure 2.1. It consists of multiple nodes, connected in an overlay network. In our implementation we used the Pastry overlay network [49]. Each node in the overlay offers one or more *services* to the system. Each 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 example, in an application that analyzes sensor network data). The size of a data unit depends on the application. Upon reception of a data unit by a node, the data unit is put 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 4.1(a). When submitting a request, the user expects from the system to create 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 run-time. 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 in Figure 4.1(b).

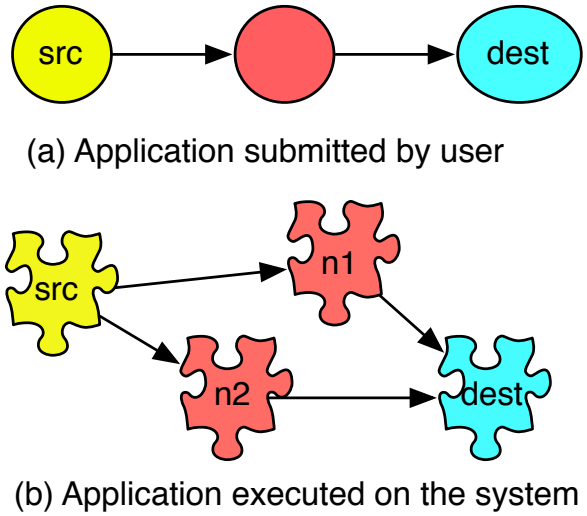

Figure 4.1: An example of an application.

4.1.2 Execution of Bursty Applications

Data to be processed by a stream processing system is typically produced at a constant rate. However, due to unexpected reason, the input rate of this data can be increased dramatically and without any warning. Characteristic examples of such situations are (1) a network monitoring application at the time of a DoS attack, (2) an application performing tracking of objects within a region on the event where multiple objects suddenly enter the region. Bursts of the input rate of a stream processing application can result in loss of data if not accommodated properly. This is because nodes and links between them can be overwhelmed by the increase of the load. It is important to note that a bursty stream processing application typically has an impact to other stream processing applications, when some of their components are hosted on the same processing nodes. The following example makes this fact clear.

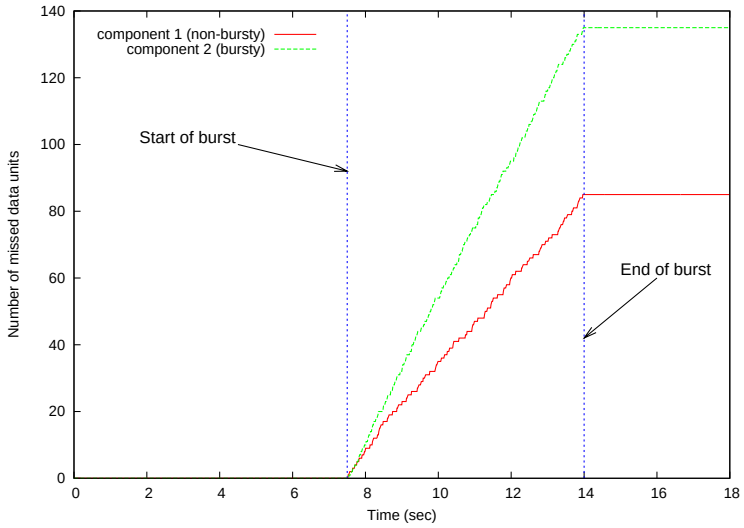

Figure 4.2: CDF of missed data units under a burst.

Example 4 *Let us consider an example with two components running on a single node. For the discussion of this section, it is irrelevant whether the components are functionally related to each other (i.e., if they belong to the same application). The rate of component 1 is* $100Kbps$*, while the rate of component 2 is* $80Kbps$ *(for simplicity, we assume that both components perform the same computation and thus have the same performance requirements). The node's capacity is such that the sum of rates for components 1 and 2 can be up to* $190Kbps$*. Initially, the input rates of the components are such that they can both accommodated by the node. However, at time* $t_1 = 7.5sec$*, the input rate of component 2 rises to* $160Kbps$ *(i.e., there is a burst). This lasts until time* $t_2 = 14sec$*. After that the rate of component 2 is reduced back to* $80Kbps$*. Under such a situation, one expects that data units to be processed by any of the components would miss their deadlines. This is because the increase of the total input rate beyond the node's capacity results to an increase*

of the queuing time of all data units waiting to be processed by the node (not just the ones to be processed by component 2). We performed such an experiment on our stream processing system. The results are shown in figure 4.2. The figure shows the CDF of the data units that could not be delivered in a timely manner. At the time of the burst, data units of both components miss their deadlines.

Data units are dropped because bursts are not taken into account during composition. A composition algorithm placing components on nodes and arranging the input rates of these components, can fully load some of the nodes. A node that is fully loaded under modest conditions, will become overloaded when the input of the applications increases (*i.e.*, on a burst). A way of preventing this from happening is to reserve an amount of the resources on each node. However, until a burst happens, these resources remain unused, resulting in under-utilization of the system. Moreover, blindly reserving resources on nodes may not be of any help to burst accommodation, as shown in the following example.

Example 5 *Let us now examine the behavior of resource reservation. As an example, consider a distributed stream processing system consisting of two nodes:* n_1 *and* n_2*. Two applications are assigned to run on* n_1 *and* n_2*. An example of each application is shown in figure 4.1. When instantiating the components, let us assume that (1) the reservation at each node is set to* $(100-x)\%$*. Thus, nodes are only allowed to be utilized up to* $x\%$ *of their capacity. Let us assume that the condition of the system is such that, in order to ensure maximum efficiency, the rate assignment algorithm places components so that* $x\%$ *of node* n_1*'s resources are utilized, by having all the data units of component 2 being processed by node* n_1*. At the same time, only some of the data units of component 1 are processed by*

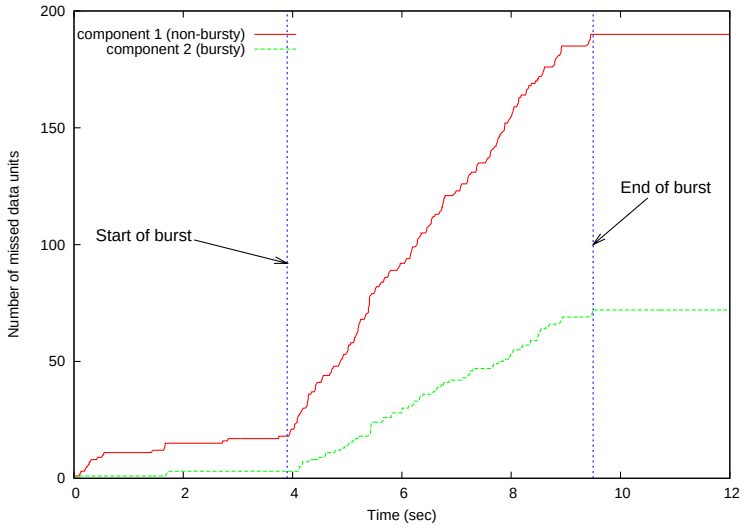

Figure 4.3: CDF of missed data (static reservation).

node n_1, while the rest are processed by node n_2. At some point in time, there is a burst on the input of component 2, resulting in node n_1 being overloaded. In other words, the reserved resources on node n_1 may not suffice to avoid overloading. The results of such an experiment run on our distributed stream processing prototype, are shown in figure 4.3. While the amount of total resources on nodes n_1 and n_2 are enough in order to process all data units, improper initial configuration of the system results on node n_1 being overloaded, while node n_2 is underutilized. Thus, data units are still being dropped.

In order to address such issues, authors of [9, 53] consider dynamic adaptation: When a burst is detected, the system is re-organized to accommodate the burst, resulting in few or no drops at all. However, the new plan needs time in order to be implemented in the system. During the time from the moment a new plan is selected until the plan is disseminated and implemented by the nodes of the system, a lot of data units are dropped.

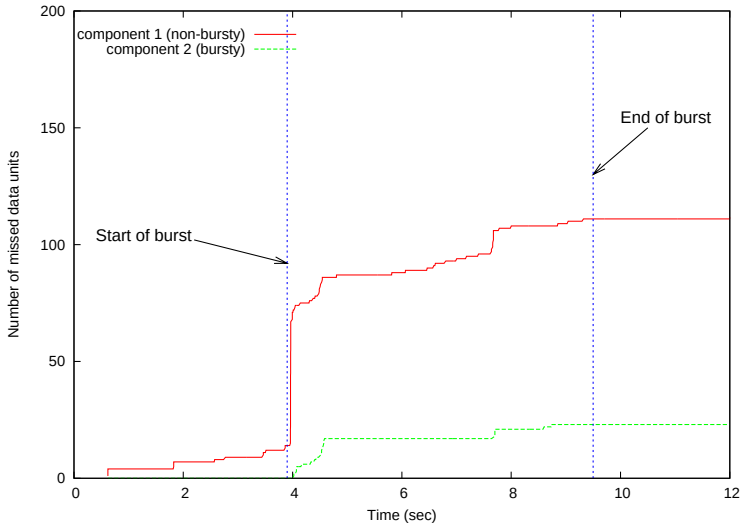

Figure 4.4: CDF of missed data (dynamic adaptation).

Example 6 *We conducted an experiment in our prototype using the setup of Example 5. In our implementation, the system monitors the input rate of each application. As soon as a burst is detected, the input rates of the components are re-adjusted so that no node will be overloaded. The results are shown in figure 4.4. Although reorganization appears to have better results than reservation, there is still a problem: The initial (mis)configuration of the system results in a lot of data units being dropped during the period from the start of the burst until the new rates are assigned to the components.*

In order to limit the above effect, a reasonable solution is to combine dynamic adaptation and careful reservation. Using dynamic adaptation, we can reconfigure our system dynamically, according to the instantaneous application rates. At the same time, by reserving some node resources, we can avoid momentary data unit drops until the system is fully adapted to accommodate a burst. Such an algorithm is demonstrated next.

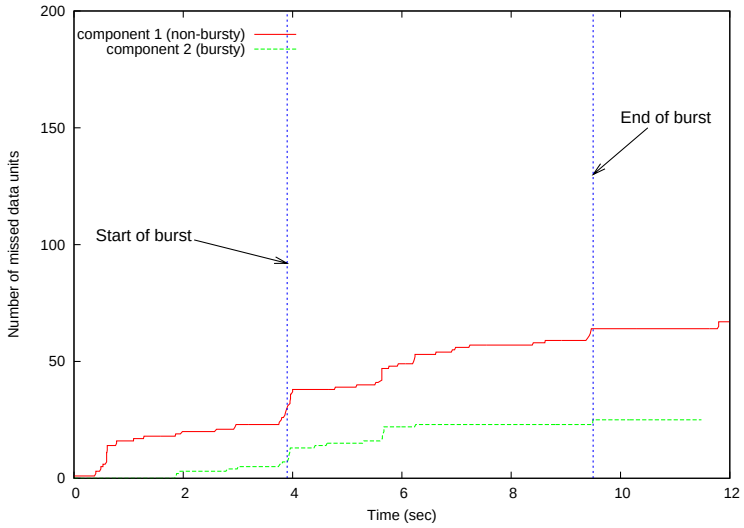

Figure 4.5: CDF of missed data under combination of dynamic adaptation and reservation.

Example 7 *We conducted an experiment in our prototype using the setup of Examples 5 and 6. In this implementation, (1) the system monitors the input rate of each application and (2) a small amount of resources was reserved on each node. The results on the number of dropped data units are shown in figure 4.5.*

As we can see from Figures 4.3, 4.4 and 4.5, reservation and dynamic adaptation are much more beneficial when used together. However, naively combining those two methods is not adequate. An algorithm that employs reservation and dynamic adaptation should take the following into account:

- Underutilization of the nodes should be avoided.
- New applications must be accommodated by the system as long as free resources are available on the nodes. Reservation should not obscure utilizing idle resources.
- Resource reservation should be done according to the streaming applications' needs.

4.1.3 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 amount A_j^n of resource j available on n. The same should be true for a link. Using to the notation presented in Section 2.1, the mathematical representation of the capacity constraints is the following:

$$\forall n \in \mathcal{N}, \sum_{c_i \in n} r_{c_i} \cdot u_j^{c_i} \leq A_j^n, 1 \leq j \leq J \tag{4.1}$$

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:

$$\forall c_i, \sum_{c_j \in \mathcal{D}(c_i)} r_{c_j} = sel^{c_i} \cdot r_{c_i} \tag{4.2}$$

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 (4.1) and (4.2) suggest that in the general case of Q applications, the feasible region is a convex polytope [60].

Example 8 *Let us consider the example shown in Figure 4.6. In this example, there are two applications, each with a single substream. The data units of Application 1 are processed by Service 1 (run by components c_1 in node A and c_3 in node B) and a final destination service $dest_1$ on a separate node. Similarly, the data units of Application 2 are processed by Service 2 (run by components c_2 in node A and c_4 in node B) and a final destination service $dest_2$ on a separate node. For the shake of brevity, we assume that the only resource consumed by components running on each node of the system is CPU cycles. For each component c_i, the amount of time needed in order to process a single data unit, is shown in*

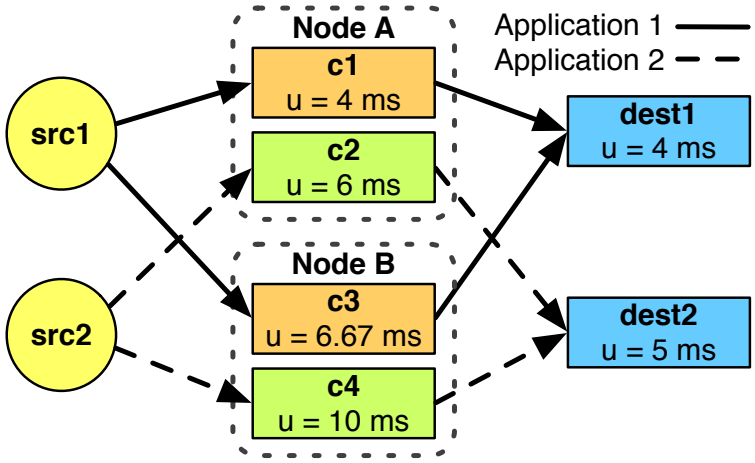

Figure 4.6: An example of two applications.

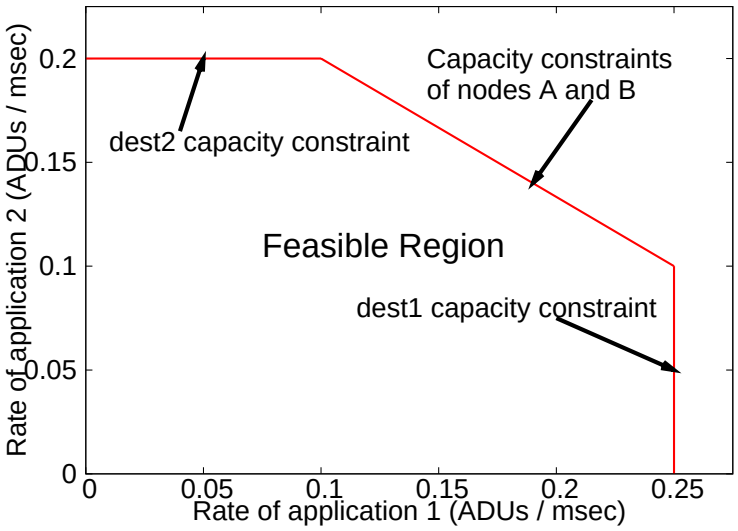

Figure 4.7: The feasible region for two applications.

the figure. This constitutes the CPU consumption $u_j^{c_i}$ for each component. In this case, the sum of the products $\sum u_j^{c_i} \cdot r_i$ for a node represents the CPU utilization of the node, while the capacity A_j^n of each node n is equal to 1 [33]. The feasible region is shown in Figure 4.7. 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 4.8 are some of the points that dominate point p. However, points p_1 and p_2 do not dominate p.

When a point p_1 dominates a point p_2, this indicates that the input rate combination represented by p_1 is "preferred" compared to the application input rate combination represented by p_2. This is because when $p_1 \succcurlyeq p_2$ is true, then (1) at least one of the application input rates represented by p_1 is bigger than the corresponding input rate for the same application represented in p_2 and (2) no application input rate in p_1 is smaller than the corresponding application input rate in p_2. Should we have a choice between two component input rate assignments, one that results in the application rates represented by p_1 and one that results in p_2, the first option should be preferred. Such a decision would result in higher burst tolerance of our system.

For a point p in the feasible region, if there is no point p' within the feasible region

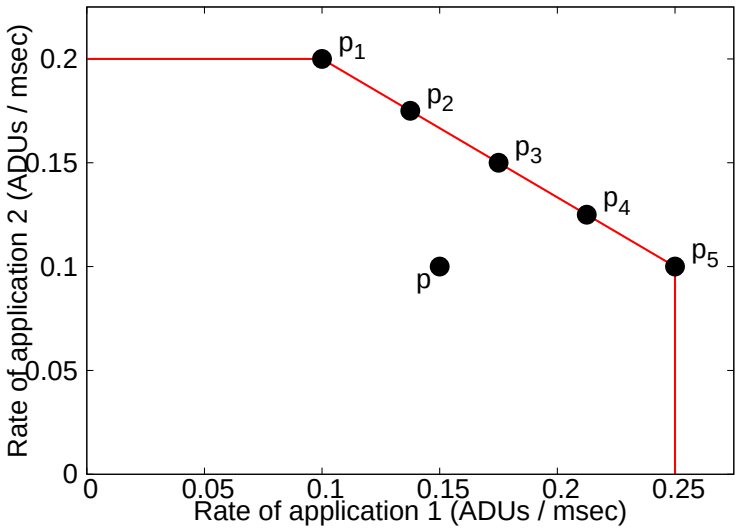

Figure 4.8: Some pareto points on the feasible region.

such that $p' \succcurlyeq p$, then p is a *pareto point* [19]. For example, points p_1, p_2, p_3, p_4 and p_5 in Figure 4.8 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 (4.3) must hold for each node $n \in \mathcal{N}$ of the system:

$$\sum_{c_i \in n} r_{c_i} \cdot u_j^{c_i} + \sum_{c_i \in n \cap \mathcal{C}^q} \delta_q \cdot \frac{r_{c_i}}{r_q} \cdot u_j^{c_i} \leq A_j^n \tag{4.3}$$

where $1 \leq j \leq J$. The first sum in Equation (4.3) 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 (4.3) can be extended to the case where multiple bursts can occur simultaneously:

$$\forall n \in \mathcal{N}, \sum_{q=1}^{Q} \sum_{c_i \in n \cap \mathcal{C}^q} r_{c_i} \cdot \left(1 + \frac{\delta_q}{r_q}\right) \cdot u_j^{c_i} \leq A_j^n \tag{4.4}$$

The bigger 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 is in order to minimize the number of data units that will be lost when reconfiguring the system in order to fully cope 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:

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

$$\boldsymbol{\delta} = \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_Q \end{bmatrix} \succcurlyeq \begin{bmatrix} c \\ \vdots \\ c \end{bmatrix} = \boldsymbol{\delta}^{eq} \tag{4.5}$$

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 (4.2) and (4.4) are not violated. Constraints (4.5) 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 $\boldsymbol{\delta}^{\hat{eq}}$ respectively. Given the above constraints, and the current input rates of the applications, we would like a rate assignment that maximizes:

$$\sum_{q=1}^{Q} \delta_q \tag{4.6}$$

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 9 *Let us consider the feasible region of Example 8 (Figure 4.7). Given an input rate combination* $p = (r_1, r_2)$*, maximizing* $\delta_1 + \delta_2$ *without violating constraints* (4.2)*,* (4.4) *and* (4.5)*, 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 4.9.*

On the other hand, if a burst results in the application input rate combination given in Figure 4.10, 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*.*

4.2 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. (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 rate assignment plan based on the pre-calculated assignments generated by the offline application composition phase and triggers this plan by adjusting the input rates of the components.

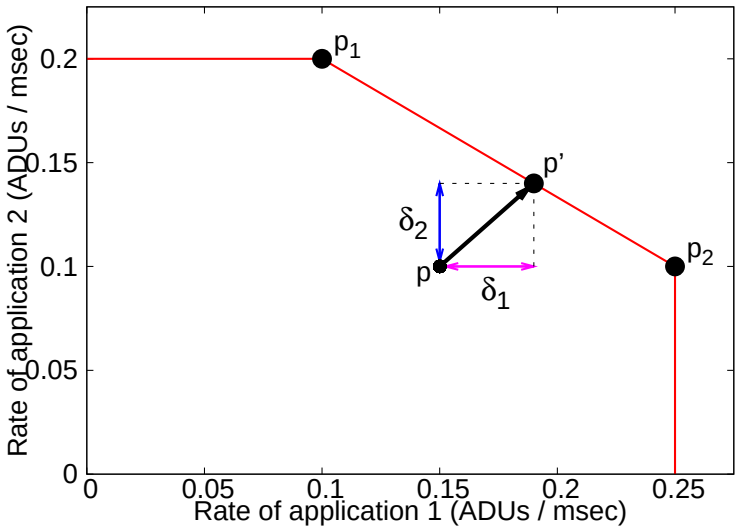

Figure 4.9: δ_1 and δ_2 for an example of 2 applications.

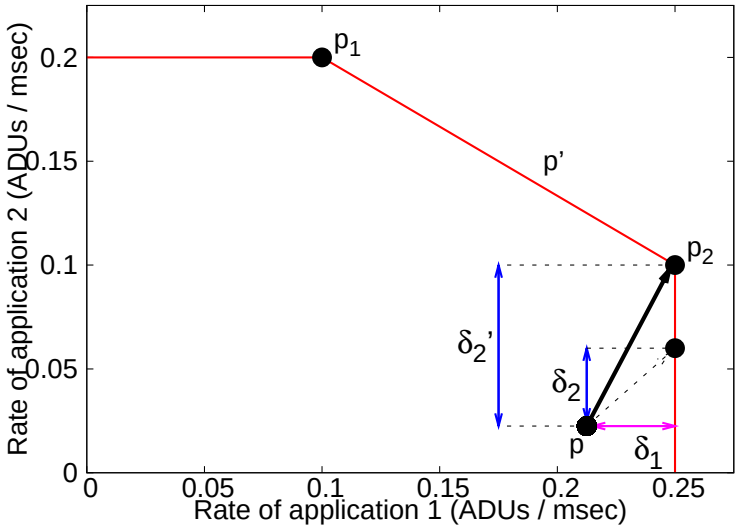

Figure 4.10: δ_1 and δ_2' for an example of 2 applications.

4.2.1 Burst Accommodation Overview

Contrary to our work in Chapters 2 and 3, in this work we assume that the initial input rate will not necessarily be kept stable. Rather, we focus on: (1) How to determine the limits of our infrastructure (*i.e.*, the set of application input rate combinations that form the boundary of our feasible region). (2) What is the optimal combination of individual component input rate in order for our system to perform under such situations. (3) How to accommodate these situations when we unpredictably encounter them. (4) How to configure the component input rates (initially or upon a burst) in such a way so that instantaneous data unit losses (during to future bursts) will be avoided. Once we are able to answer these questions at runtime, it is possible to modify the input rates of the various components in response to changes of the input rate of applications (application input rate bursts). This means that component input rates in our system are dynamically modified in order to accommodate bursty input rates. The steps of our burst accommodation system are:

Preparation of Rate Assignment Plans: In order to be able to respond in a timely manner to a change of an application's input rate, BARRE pre-calculates a number of rate allocation arrangements. Each arrangement is suitable for different combinations of application input rates. This strategy is necessitated from the requirements expressed by Equations (4.2), (4.4) and (4.5). Since Equations (4.4) and (4.5) are not linear, the rate assignment problem would need a rather time-consuming algorithm to run whenever reconfiguration was needed. Instead, we use a linear optimization method in order to precalculate the appropriate rate allocation for each situation.

Runtime Rate Monitoring: The input rate of each component of the system is continuously monitored. Bursts of the input rate of an application are detected by observing the input rates of individual components. In addition to burst detection, monitoring is also used in order to determine the amount of the resources that are used by each component.

Rate Adjustment Calculation: Upon detection of a burst, our system needs to respond in a timely manner by modifying the input rates of the components. BARRE then makes an assignment based on the rate allocations that were pre-calculated during the plan preparation phase. It is important to mention that the the new plan is selected so that the system will subsequently endure the highest possible future burst of the input rate of an application. The same method is used to address the cases where the input rates of more than one applications change simultaneously.

Component Input Rates Reconfiguration: During this phase, the component input rates calculated on the previous step are put into effect. This is done in a distributed fashion, by having each node adjust the output rates to its downstream components.

4.3 Feasible Region Determination

In section 4.1.3 we formulated our optimization problem. Constraints (4.2) and (4.4) are emerging due to the structure of each application as well as the architecture of the system (*i.e.*, the nodes that offer particular services and their capacities). Constraints (4.5) emerge due to our desire to provision for bursts on the input rate of every application, rather than optimizing for a single application. 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 4.8. 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 com-

ponent rate assignments will be used in order 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* [58]. 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 in order to identify the index points, the corresponding component rate assignments as well as the facets of a feasible region.

4.3.1 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:

$$\sum_{c_i \in \mathcal{S}(app_q)} r_{c_i} \tag{4.7}$$

where $\mathcal{S}(app_q)$ are the input rates to the input components (the components that instantiate the first service of the application). This is equal to the total input rate of the application. In order to find its maximum, we maximize the above equation subject to the capacity constraints given in Equation (4.1) and the selectivity and flow conservation constraints given in Equation (4.2). By solving this problem for each application, we determine an appropriate component rate allocation plan for the respective point in the feasible region.

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 [59].

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 (4.1) and (4.2). 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.

Example 10 *Consider the case in Figure 4.11(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 4.11(b) and 4.11(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 $\boldsymbol{\delta}^{\hat{e}q}$. Each triplet $< p, sol(p), \mathcal{F}(p) >$ is then indexed by $proj(p)$ on a spatial index [25]. 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

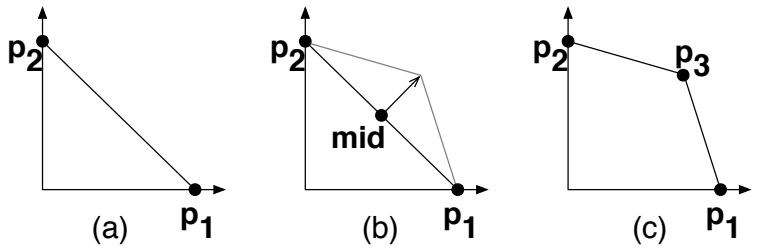

Figure 4.11: Finding the index points.

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.

Example 11 *The index points for the feasible region of Figure 4.7 are points p_1, p_2 and p_3 in Figure 4.12. 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 4.12, $proj(p)$ is closer to $proj(p_3)$ than to $proj(p_1)$ or $proj(p_2)$.*

4.4 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

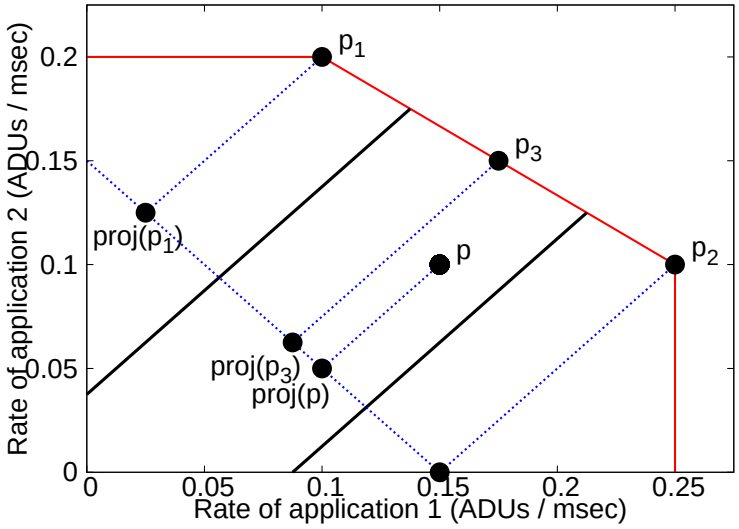

Figure 4.12: Index points, their projections and feasible region partitions.

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 4.1. 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 4.9 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. In order to do this, we calculate 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[1], *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 4.9, 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

[1]Remember that all of f's edges are index points

and was never inserted to the Point Index Database. Thus, p_i is selected. This is the case of Figure 4.10, 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 $sol(p_i)$.

4.5 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 numbers of applications. We compare our approach with several schemes.

4.5.1 Experimental Setup

BARRE was implemented in about 15000 lines of Java. Service discovery and statistics collection were implemented using the FreePastry library [16], an open source implementation of Pastry. Complex matrix operations were carried out using JAMA [29]. A spatial index implementation from [26] 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.

4.5.2 Experimental Results

BARRE Overhead: Figures 4.13, 4.14 and 4.15 demonstrate the time needed by BARRE to pre-calculate the rate assignment plans. Figure 4.13 shows the time needed by our method in order to construct 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 4.3.1, only a limited number of points needs to be kept in the Index Point Database. The size of the index, shown in Figure 4.14, 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 4.15. In that figure, the time needed in order 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, all applications' rate 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

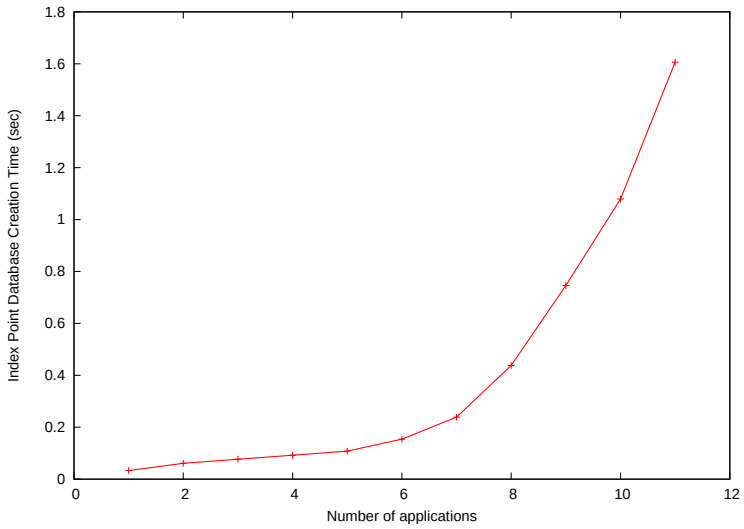

Figure 4.13: Index Point Database creation time.

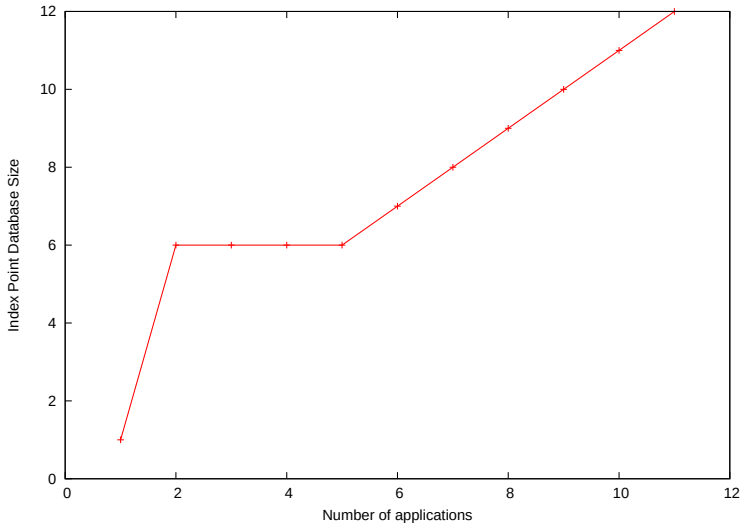

Figure 4.14: Index Point Database size.

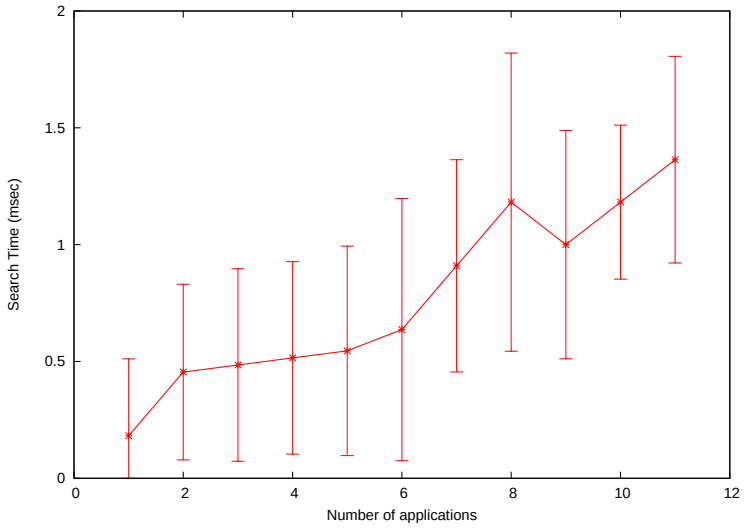

Figure 4.15: Index Point Database search time.

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 [53, 9]) and (4) an algorithm that performs both **Dynamic Adaptation and Static Reservation**.

BARRE Operation: In Figure 4.16 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:

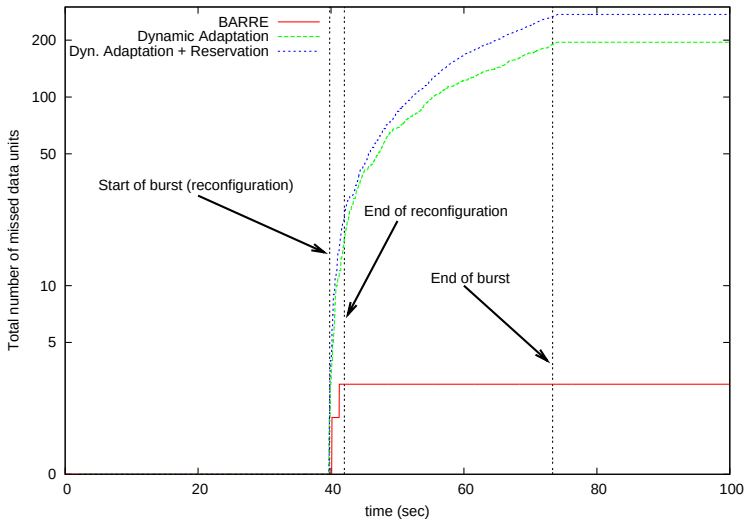

Figure 4.16: The time of data unit misses.

1. BARRE ends up dropping far less 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 2.3*sec* in this example), the 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.

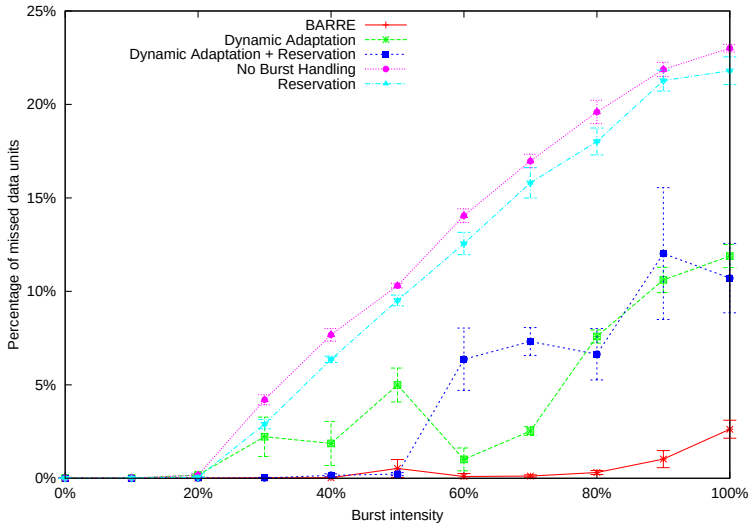

Figure 4.17: Percentage of missed data units.

Missed Data Units: The percentage of data units that were dropped is shown in Figure 4.17. 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

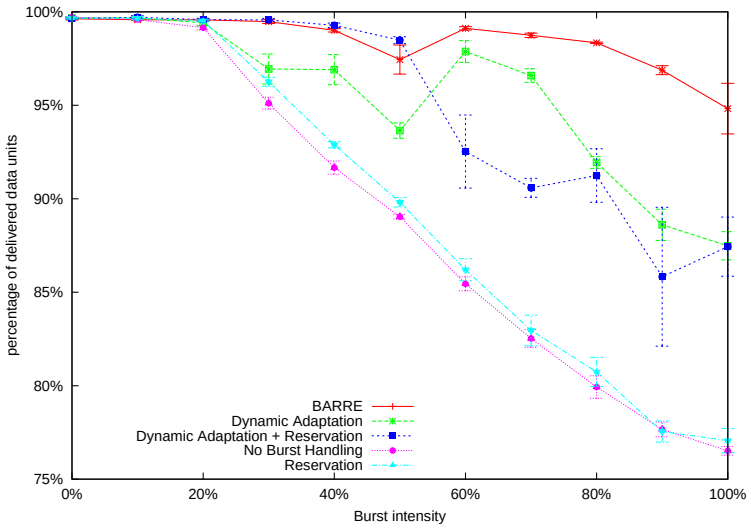

Figure 4.18: Percentage of flawlessly delivered data units.

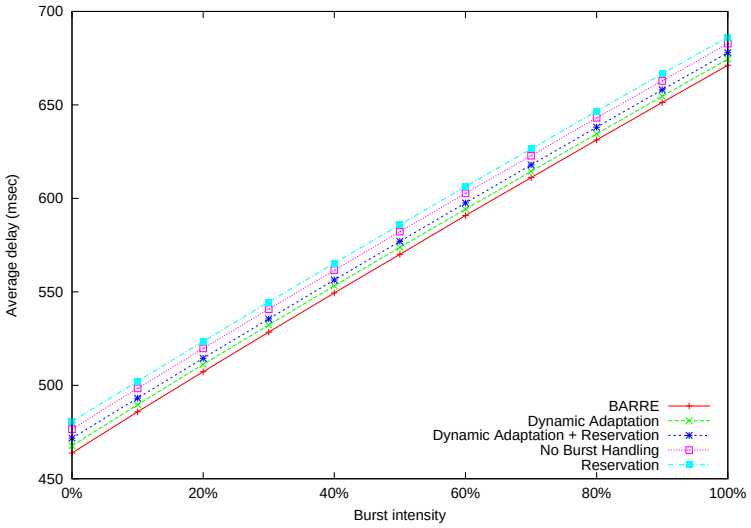

Figure 4.19: Average delay of data units.

manner is shown in Figure 4.18. 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 4.19. 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 in order to 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.

Chapter 5

Related Work

Streaming Processing Systems: Recent efforts have studied the problem of resource allocation in distributed stream processing environments [24, 37, 63]. Most optimal service composition is accomplished in [24], 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 fully utilizing the given resources of the system in order to maximize the QoS of the offered services, with the resource constraints of the nodes in mind. We have previously investigated different aspects of overlays for distributed applications. In [7] we have focused on the task scheduling algorithm, while in [8] we have described a decentralized media streaming and transcoding architecture. In [48], we considered re-using components to improve efficiency.

Recent research efforts have investigated the use of peer-to-peer overlays for media streaming support. The service graph construction has been the focus of works like SpiderNet [23] and PROMISE [28]. SpiderNet uses a probing protocol to setup the service graph,

while in PROMISE a receiver selects senders based on characteristics such as the offered rate, the availability, the available bandwidth, and the loss rate. The performance of these methods can be further increased by incorporating RASC.

Component Middleware: During the past few years, substantial effort has focused on developing standards-based component-oriented middleware such as OMG's CORBA [43], Sun's Enterprise JavaBeans [51] and Microsoft's COM [40]. These simplify the development of platform-interoperable, vendor-independent and language-neutral applications. Recently, Model Driven Development techniques and tools have been integrated with component middleware technologies to develop formally analyzable and verifiable building block components[20]. Component-based architectures are employed more and more often in the development of distributed applications [24, 48, 55, 56].

Similar to our work, a component system is presented in [55] to meet real-time specifications given by the user. In [56], the authors use an informed branch-and-bound algorithm employing a competence function and forward checking to expedite its execution. Both these algorithms are centralized and meant to be run off-line. We aim for optimal resource allocation, that is efficient and can be is based on current system conditions and application requirements. Also, our system is designed to address the demands of stream processing applications. In the QuO project, mechanisms for providing dynamic QoS support for component-based applications have been proposed [50]. However, they mainly focus on assembling and configuring these components to enable adaptive QoS management. Our algorithms can be implemented over their components.

Load Balancing in Distributed Stream Processing: Multimedia streaming has been studied extensively in the last few years, mainly from a centralized perspective and without focusing on dynamic user requirements. The importance of providing multimedia applications on overlay networks has only been recognized by recent efforts. These fall in one of the following main categories: (1) Construction of a service graph that satisfies the requested QoS requirements [28]. Layered streaming techniques [11] which address the problem of bandwidth heterogeneity by lowering the streaming quality via the omission of stream layers. (3) Proxy-based service provision coordination solutions [30] which make use of a central scheduler. (4) Multicasting mechanisms [5, 34, 54] for efficient content distribution. However, these solutions fail to support multimedia applications that require both communication and processing. More importantly, though, they fail to support multiple media services that can be composed into customized services.

In our previous work we have investigated fair resource allocation for data-sharing applications [13], as well as different aspects of overlays for distributed applications In particular, in [7] we have focused on the task scheduling algorithm, while in [8] we have described a totally decentralized media streaming and transcoding architecture. Our current work builds upon [47] and provides a detailed experimental evaluation.

The problem of resource allocation with fair quality levels in the context of real time control systems is considered in [27]. The goal is to allocate system resources so that the quality levels of the tasks will be as fairly distributed as possible. A peer's load is defined to be its CPU utilization factor, while the quantity to be fairly distributed is the quality level offered to each of the running tasks. Our system differs since we try to balance

peer load instead of the offered quality level. Also, we consider the problem in the context of wide-area, distributed peer-to-peer overlay networks.

Burst Accommodation: In [3], 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 [61] formulate the problem of distributed stream processing as a utility optimization problem. They then apply a well-known network routing algorithm in order 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 [62], 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 [53] consider load shedding in order 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 by assigning the execution of the excessive data units to alternative processing nodes.

System reconfiguration under the face of bursts is supported by MPRA [9]. However, authors assume that there is a set of applications with more than one acceptaple processing rates. Thus, in the face of bursts, authors choose to modify the operation rate of such applications. Decrease of 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.

Chapter 6

Conclusion And Future Work

Stram processing applications is a new class of applications with numerous examples of processing components. Such examples are filtering operations (*e.g.* selection of specific values or ranges of values, projection of specific attributes of the input), aggregation operators, or more complex operations, such as video transcoding. In order to sufficiently support such systems, our goal is to meet their real-time quality-of-service requirements. In this thesis, we have adressed a variety of problems in order to tackle the problem of rate allocation for Distributed Stream Processing Systems. We considered the problems of dynamic rate allocation, load balancing and burst accommodation in such systems.

In Chapter 2, we have studied the problem of dynamic rate allocation for distributed stream processing applications in large-scale overlays. We have proposed a dynamic composition algorithm that selects application components dynamically, while considering the rate requirement of the application, the resource availability and the number of data units that have missed their deadlines. We have implemented our distributed stream pro-

cessing technique on the Planet-Lab testbed. Our experimental results demonstrate the efficiency, scalability and performance of our approach.

In Chapter 3, we have proposed a distributed resource allocation algorithm to deal with challenges of heterogeneity and scalability in supporting distributed stream processing applications in large-scale systems. Our approach aims for a fair load distribution among the nodes of the system, while meeting the QoS requirements of the applications. We employ quality adaptation mechanisms to deal with dynamic changes in resource utilization and application behavior The design, implementation and evaluation of our approach is presented. We have evaluated our algorithms by simulating a distributed media streaming and transcoding application. Our results show that a fair and adaptive load distribution in a large-scale system can achieve scalability and maximize the probability of meeting the application requirements.

In Chapter 4, we addressed the problem of addressing unpredicted bursts of the input rate of distributed stream processing systems. We proposed an algorithm that uses dynamic resource allocation and reservation in order to tackle the problem. 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 a real testbed. Our experimental results demonstrate the efficiency, scalability and performance of our approach over techniques such as reservation and simple dynamic system adaptation.

Problems that remain unsolved in the area of distributed stream processin systems are dependencies among data units. This happens in applications where some data units of

an application need to be processed in conjunction with the data carried on previous data units of the application. This prolem becomes particularly hard in the case when these data units are processed in different nodes. Efficient methods need to be found in order to minimize the overhead that rate splitting would incur in such a case, due to the need of data unit duplication to multiple nodes.

Another important issue in DSPS is fault tolerance support. Fault tolerance refers to two problems: (1) How to recover the case of a stream processing node failure, with the minimum possible effect on the running stream processing applications. (2) How to handle a disorder and delays of input streams. These effects appear due to the underlying network of the distributed stream processing system. Disorder exists when the data units of an application are transmitted with the wrong order (due to delay variability of individual network packets). Delay happens when data units of an application delay arriving to their destination for an unreasonably big time interval. In addition to deadlines being missed, question is also raised whether the application has finished executing or not.

Bibliography

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. C etintemel, M. Cherniack, J.H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, January 2005.

[2] Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, 2004.

[3] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 71, Washington, DC, USA, 2006. IEEE Computer Society.

[4] 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. http://infolab.stanford.edu/stream, March 2005.

[5] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in a cooperative environment. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, October 2003.

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

[7] Fang Chen and Vana Kalogeraki. RUBEN: A technique for scheduling multimedia applications in overlay networks. In *Proceedings of the IEEE Global Telecommunications Conference (Globecom)*, Dallas, TX, November 2004.

[8] Fang Chen, Thomas Repantis, and Vana Kalogeraki. Coordinated media streaming and transcoding in peer-to-peer systems. In *IPDPS*, Denver, CO, April 2005.

[9] Yingming Chen, Chenyang Lu, and Xenofon Koutsoukos. Optimal discrete rate adaptation for distributed real-time systems. In *Real Time Systems Symposium*, Tucson, AZ, December 2007.

[10] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR*, Asilomar, CA, January 2003.

[11] Y. Cui and K. Nahrstedt. Layered peer-to-peer streaming. In *NOSSDAV, Monterey, CA, USA*, June 2003.

[12] Frank Dabek, Frank Mass Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001.

[13] Yannis Drougas and Vana Kalogeraki. A fair resource allocation algorithm for peer-to-peer overlays. In *8th Global Internet Symposium*, Miami, FL, March 2005.

[14] Yannis Drougas and Vana Kalogeraki. RASC: Dynamic rate allocation for distributed stream processing applications. In *21st International Parallel and Distributed Processing Symposium (IPDPS)*, Long Beach, CA, USA, March 2007.

[15] Yannis Drougas, Thomas Repantis, and Vana Kalogeraki. Load balancing techniques for distributed stream processing applications in overlay environments. In *ISORC*, Gyeongju, Korea, April 2006.

[16] Peter Druschel, Eric Engineer, Romer Gil, Andreas Haeberlen, Jeff Hoye, Y. Charlie Hu, Sitaram Iyer, Andrew Ladd, Alan Mislove, Animesh Nandi, Ansley Post, Charlie Reis, Dan Sandler, Jim Stewart, Atul Singh, and RongMei Zhang. Freepastry. http://freepastry.org/FreePastry, 2006.

[17] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[18] G. Fry and R. West. Adaptive routing of QoS-constrained media streams over scalable overlay topologies. In *IEEE RTAS, Toronto, Canada*, May 2004.

[19] Marc Geilen, Twan Basten, Bart Theelen, and Ralph Otten. An algebra of pareto points. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD)*, pages 88–97, Washington, DC, USA, 2005. IEEE Computer Society.

[20] Sebastien Gerard, Jean-Philippe Babau, and Joel Champeau. *Mode Driven Engineering for Distributed Real-time Embedded Systems*. Hermes, 2005.

[21] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[22] Andrew V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *Journal of Algorithms*, 22(1):1–29, 1997.

[23] Xiaohui Gu and Klara Nahrstedt. Distributed multimedia service composition with statistical QoS assurances. *IEEE Transactions on Multimedia*, 2005.

[24] Xiaohui Gu, Philip S. Yu, and Klara Nahrstedt. Optimal component composition for scalable stream processing. In *25th International Conference on Distributed Computing Systems (ICDCS)*, Columbus, OH, June 2005.

[25] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[26] Marios Hadjieleftheriou. Spatial index library. http://research.att.com/∼marioh/spatialindex, 2008.

[27] Fumiko Harada, Toshimitsu Ushio, and Yukikazu Nakamoto. Adaptive resource allocation control with on-line search for fair QoS level. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 352–359, May 2004.

[28] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava. PROMISE: Peer-to-peer media streaming using CollectCast. In *Proceedings of the 11th ACM International Conference on Multimedia*, Berkeley, CA, November 2003.

[29] Joe Hicklin, Cleve Moler, Peter Webb, Ronald F. Boisvert, Bruce Miller, Roldan Pozo, and Karin Remington. Jama: A java matrix package. http://math.nist.gov/javanumerics/jama, 2008.

[30] M. Hicks, A. Nagarajan, and R. Renesse. User-specified adaptive scheduling in a streaming media network. In *OPENARCH, San Francisco, CA, USA*, April 2003.

[31] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the 21st annual symposium on Principles of distributed computing*, pages 213–222, New York, NY, USA, 2002. ACM.

[32] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Have. A quantitive measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report DEC-TR-301, Digital Equipment Corporation, 1984.

[33] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 304, Washington, DC, USA, 1999. IEEE Computer Society.

[34] X. Jiang, Y. Dong, D. Xu, and B. Bhargava. GnuStream: A P2P media streaming prototype. In *IEEE International Conference on Multimedia and Expo*, Baltimore, MD, July 2003.

[35] Patrick Kirk. Gnutella Protocol Development. http://rfc-gnutella.sourceforge.net/, 2003.

[36] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, 2000. ACM Press.

[37] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Resource-aware distributed stream management using dynamic overlays. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 783–792, Washington, DC, USA, June 2005. IEEE Computer Society.

[38] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource QoS problem. In *20th Real-Time Systems Symposium (RTSS)*, Phoenix, AZ, December 1999.

[39] Samuel Madden and Johannes Gehrke. Query processing in sensor networks. *IEEE Pervasive Computing*, 3(1), March 2004.

[40] Microsoft Corporation. COM: Component Object Model Technologies. http://www.microsoft.com/com, 2006.

[41] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, San Diego, CA, August 2001.

[42] K. Nahrstedt, D. Wichadakul, and D. Xu. Distributed qos compilation and runtime instantiation. In *IWQoS, Pittsburgh, PA, USA*, June 2000.

[43] Object Management Group. The Common Object Request Broker: Architecture and Specification. Edition 2.4, formal/00-10-01, October 2000.

[44] PlanetLab Consortium. http://www.planet-lab.org, 2004.

[45] Michael Rabinovich and Amit Aggarwal. RaDaR: a scalable architecture for a global Web hosting service. *Computer Networks*, 31(11–16):1545–1561, 1999.

[46] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[47] Thomas Repantis, Yannis Drougas, and Vana Kalogeraki. Adaptive resource management in peer-to-peer middleware. In *WPDRTS*, Denver, CO, April 2005.

[48] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Middleware*, Melbourne, Australia, November 2006.

[49] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, Heidelberg, Germany, November 2001.

[50] P. K. Sharma, J. P. Loyall, T. Heineman G, R. E. Schantz, R. Shapiro, and G. Duzan. Component-based dynamic QoS adaptations in distributed real-time and embedded systems. In *DOA*, Agia Napa, Cyprus, October 2004.

[51] Sun Microsystems. Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/, 2006.

[52] Nesime Tatbul, Ugur cCetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, Berlin, Germany, 2003.

[53] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying FIT: Efficient load shedding techniques for distributed stream processing. In *International Conference on Very Large Data Bases*, pages 159–170, Vienna, Austria, September 2007.

[54] D. Tran, K. Hua, and T. Do. ZIGZAG: An efficient peer-to-peer scheme for media streaming. In *IEEE INFOCOM 2003, San Francisco, CA, USA*, April 2003.

[55] Shengquan Wang, Sangig Rho, Zhibin Mai, Riccardo Bettati, and Wei Zhao. Real-time component-based systems. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 428–437, San Francisco, CA, March 2005. IEEE Computer Society.

[56] Shige Wang, Jeffrey R. Merrick, and Kang G. Shin. Component allocation with multiple resource constraints for large embedded real-time software design. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 219, Toronto, Canada, May 2004. IEEE Computer Society.

[57] Yuan Wei, Vibha Prasad, Sang H. Son, and John A. Stankovic. Prediction-based qos management for real-time data streams. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 344–358, Washington, DC, USA, 2006. IEEE Computer Society.

[58] Eric W. Weisstein. Facet. `http://mathworld.wolfram.com/Facet.html`, 2008.

[59] Eric W. Weisstein. Normal Vector. `http://mathworld.wolfram.com/NormalVector.html`, 2008.

[60] Eric W. Weisstein. Polytope. `http://mathworld.wolfram.com/Polytope.html`, 2008.

[61] Cathy H. Xia, Don Towsley, and Chun 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*, page 68, Washington, DC, USA, 2007. IEEE Computer Society.

[62] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd international conference on Very Large Data Bases*, pages 775–786, Seoul, Korea, September 2006. VLDB Endowment.

[63] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the Borealis stream processor. In *21st International Conference on Data Engineering (ICDE)*, pages 791–802, Tokyo, Japan, April 2005.

[64] Yong Yao and Johannes Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

[65] E.W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *IEEE INFOCOM 1996, San Francisco, CA, USA*, March 1996.