RASC: Dynamic Rate Allocation for Distributed Stream Processing Applications

Yannis Drougas, Vana Kalogeraki*
Department of Computer Science and Engineering
University of California-Riverside, Riverside, CA 92521
{drougas,vana}@cs.ucr.edu

Abstract

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. In this paper, we propose a distributed stream processing system that composes stream processing applications dynamically, while meeting their rate demands. Our system consists of the following components: (1) a distributed component discovery algorithm that discovers components available at nodes on demand, (2) resource monitoring techniques to maintain current resource availability information, (3) a scheduling algorithm that schedules application execution, and (4) a minimum cost composition algorithm that composes applications dynamically based on component and resource availability and scheduling demands. Our detailed experimental results, over the PlanetLab testbed, demonstrate the performance and efficiency of our approach.

1 Introduction

During the past few years, numerous applications that generate and process continuous streaming data have emerged. Examples include network traffic monitoring, financial data analysis, multimedia delivery and sensor streaming in which sensor data are processed and analyzed in real-time [1, 2]. In a typical stream processing application, streams of data are processed concurrently and asynchronously by one or more processing components. Examples of processing components include filtering operations (*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.

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*.

A number of stream processing infrastructures have been proposed in the literature, including Aurora[25], STREAM[1], TelegraphCQ[2] and the Cougar project[29, 16]. 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 [5] and have investigated composition and placement algorithms. The majority of these (including our previous work[6, 20]) 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

*This research was supported by NSF Awards 0330481 and 0627191.

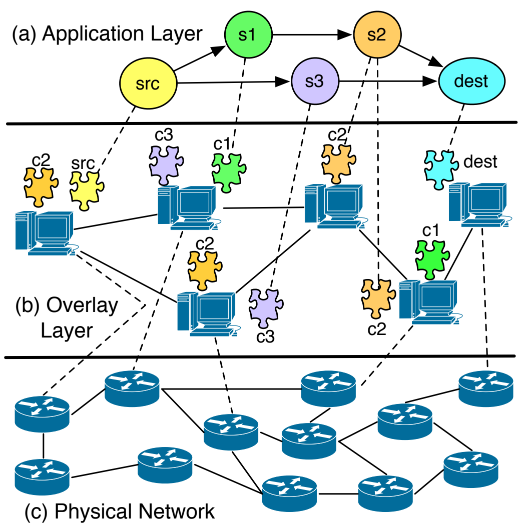

Figure 1. The architecture of a distributed stream processing system.

rates to streams based on their rate requirements.

In this paper 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 different 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[19], that demonstrate the performance and efficiency of our approach.

2 System Model

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

2.1 System Architecture

Our distributed stream processing system is illustrated in Figure 1. It consists of multiple nodes, connected in an overlay network. In our implementation we used the Pastry overlay network [22]. 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 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 and operation on a single data unit is independent of oper-

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

ations 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* $\mathbf{A}^n = [A_1^n, \ldots, A_k^n]$.

2.2 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 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: 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.

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 3 which represents the mapping of the service request graph of 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.

2.3 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*

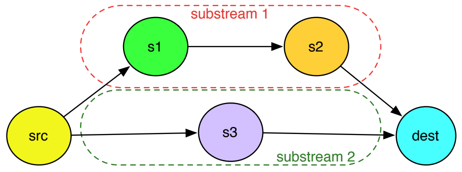

Figure 2. An example of a service request graph.

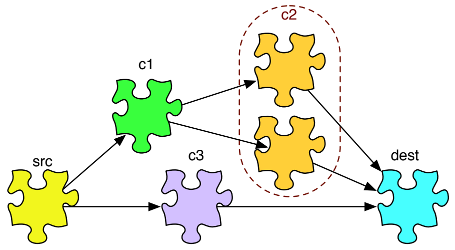

Figure 3. The execution graph represents the mapping of the service request graph on the overlay network in which one or more components can be used to instantiate a service.

requirements:

$$Minimize: \sum_{c_i} r_{in}^{c_i} \cdot drops^{n(c_i)} \qquad (1)$$

$$Subject\ to: r_l^{dest} = r_l^{req}, 1 \leq l \leq m \qquad (2)$$

$$\forall n \in \mathcal{N}, \sum_{c_i \in n} r_{in}^{c_i} \cdot u_j^{c_i} \leq A_j^n, j = 1, \ldots, k \qquad (3)$$

$$\forall c_i, \ r_{out}^{c_i} = R^{c_i} \cdot r_{in}^{c_i} \qquad (4)$$

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 jth resource consumed by component c_i when processing one data unit per second, A_j^n is the amount of the jth resource available on node n, r_l^{dest} is the resulting rate of the lth substream arriving at the destination given by G_{req} and r_l^{req}, $1 \leq l \leq m$ is the required rate of the lth substream arriving at the destination, given by the lth 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 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 3 expresses the capacity constraints of each link between two nodes in our system. Equation 4 expresses the *rate conservation requirement*: The output rate of each component should be proportional to its input rate. Finally, equation 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 3.2, monitoring of these metrics is required.

3 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.

3.1 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.* [7, 10, 11]). Thus, our solution manages to split the execution of individual services to more than one compo-

The minimum cost flow problem considers that the rate ratio of each component is equal to 1. In our problem definition, equation 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 1-4.

nents, 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.

3.2 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, $\mathbf{A}^n = [A^n_1, A^n_2] = [b^n_{in}, b^n_{out}]$. 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 3.4). 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.

3.3 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[22]. 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 $r^{c_i}_{in}$ and $r^{c_i}_{out}$ 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.

3.4 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 $arr^{c_i}_j$ 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: $arr^{c_i}_{j+1} = arr^{c_i}_j + p^{c_i}$. 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 $d^{du} = arr^{c_i}_{j+1}$ 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.

3.5 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 1-4. 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 $\mathbf{A}^n = [b_{in}^n, b_{out}^n]$, where b_{in}^n and b_{out}^n 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 $r_{max}(c_i, n) = \min\left\{\frac{A_1^n}{u_1^{c_i}}, \ldots, \frac{A_k^n}{u_k^{c_i}}\right\}$. 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, $r_{max}(c_i, n) = r_{max}(n) = \min\{b_{in}^n, b_{out}^n\}$).

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 3.

Example 1. *To better illustrate our method, let us consider a stream processing request with a request graph like the one in Figure 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 4). In that case, the graph of Figure 3 is just one of many possible execution graphs that can be constructed for the stream processing request graph of Figure 2. All the different possible combinations are shown in Figure 5. In order to devise the execution graph, one has to consider all the component execution combinations shown in Figure 5. This makes the optimal composition problem 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.*

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 $l$th substream.
    if no acceptable assignment was found then
        return error
    end if
    Update the node capacities.
end for
return the assignment found.
```

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 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. 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 [7, 10, 11]: 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 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). (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 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 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

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 ∞.

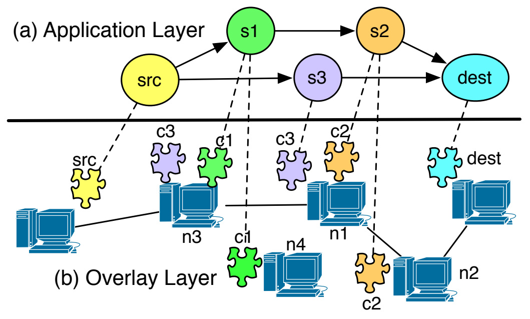

Figure 4. Composing a distributed stream processing application.

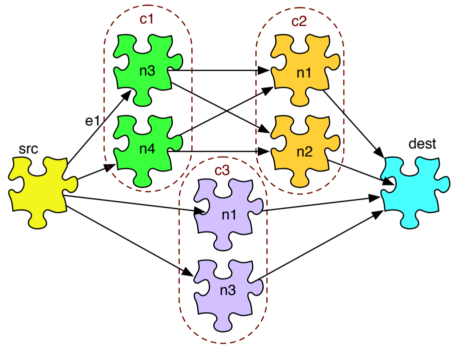

Figure 5. Component execution combinations for a distributed stream processing application.

composition algorithm is Algorithm 1.

4 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 over the PlanetLab [19] testbed.

4.1 Experimental Setup

RASC was implemented in about 13800 lines of code in Java. Service discovery and statistics collection were implemented using the FreePastry library [8], an open source implementation of Pastry. Each of the results presented is the average of 5 runs, each on 32 PlanetLab nodes. The 5 sets of experiments ran on different times and days to ensure that the results were not biased by a specific state of the PlanetLab nodes. There were 10 unique services offered in the system. Each node hosted 5 services, resulting in each service having an average replication degree of 16. Each service request included 2 to 5 services, chosen randomly among the services available on the system. The rate required by each service request ranged from $50 Kpbs$ to $200 Kbps$.

We compared the minimum cost composition method 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.

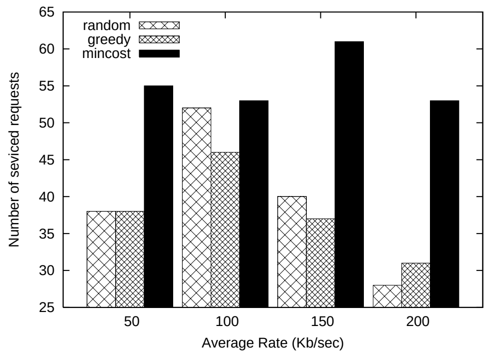

Figure 6. The number of requests that were successfully composed.

4.2 Results and Analysis

Composed Requests: Figure 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. What is also important, is that the rate of the streams does not affect the system when using the minimum cost composition algorithm. Random and greedy composition methods are more vulnerable to such changes, 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.

Average End-to-End Delay: The average end-to-end delay of the data units is shown in Figure 7. The minimum cost composition used in RASC offers 20% to 70% im-

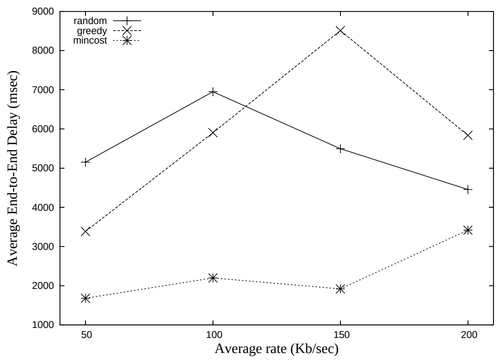

Figure 7. The average delay of the data units.

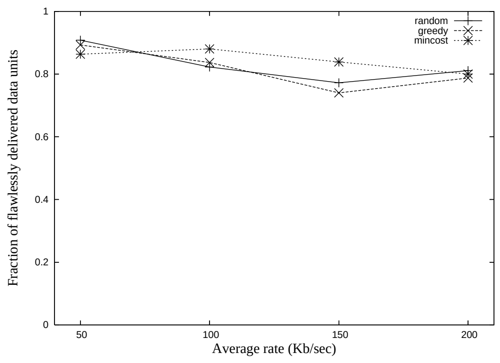

Figure 9. The fraction of data units that were delivered in a timely manner.

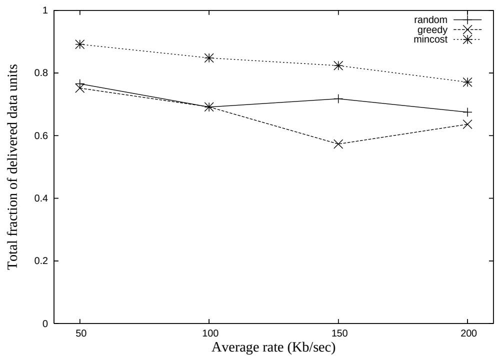

Figure 8. The fraction of data units that were not dropped.

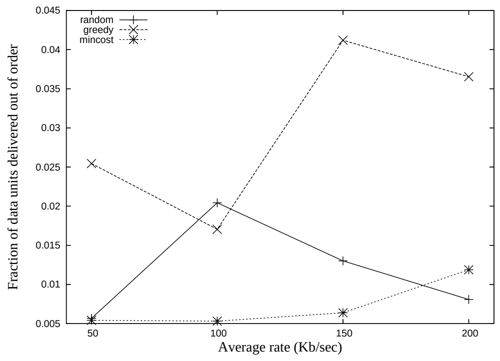

Figure 10. The fraction of data units that were delivered out of order.

provement over the random composition algorithm and 25% to 75% 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).

Delivered Data Units: The fraction of data units that were not dropped due to the system not being able to handle them, is shown in Figure 8. The minimum cost composition method manages to handle a greater fraction of the data units presented to the system. In addition, as will be shown later, 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 4 times the load when using the minimum cost approach than when using the greedy or random composition approach. This happens since the utilization of the system increases significantly, by using the 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. The frac-

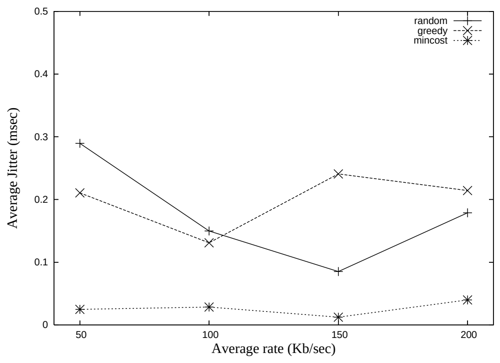

Figure 11. The average jitter.

tion of data units that were delivered in a timely manner (in respect to the total number of data units not dropped), is shown in Figure 9. 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. If the jth data unit of a stream arrives out of order, this means that the $(j+1)$th data unit or a subsequent one has already arrived to the destination before the jth data unit, rendering useless the data carried by the latter. The jth data unit is considered to not arrive in respect with the rate requirement, if it arrives at the destination at a much later time than the one dictated by the arrival time of the $(j-1)$th data unit and the required period given by the rate requirement $\mathbf{r}_{req}$ of the application. As shown in Figure 9, the fraction of delivered data units that did not arrive in a timely manner, is small.

Data Units Delivered Out Of Order: Figure 10 shows the fraction of data units that were delivered out of order (*i.e.*, later than at least one of the data units produced in their succession). This number remains low for all the algorithms. The minimum cost algorithm is shown to perform at least the same as the greedy composition approach. In many cases, it is twice as good as greedy. The random composition algorithm is up to 4 times worse than the other solutions, having up to 4% of the data units delivered out of order. The performance of both greedy and random is shown to improve when the required application rate is $200Kb/sec$. This is because greedy and random execute few of the requested applications, due to their inability to compose many of them.

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. In Figure 11, we can see that the minimum cost composition results in a system that presents 3 to 10 times less jitter than the random approach and 4 to 8 times times less than the greedy approach.

5 Related Work

Component Middleware: During the past few years, substantial effort has focused on developing standards-based component-oriented middleware such as OMG's CORBA [18], Sun's Enterprise JavaBeans [24] and Microsoft's COM [17]. 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[9]. Component-based architectures are employed more and more often in the development of distributed applications [13, 21, 27, 26].

Similar to our work, a component system is presented in [27] to meet real-time specifications given by the user. In [26], 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 [23]. However, they mainly focus on assembling and configuring these components to enable adaptive QoS management. Our algorithms can be implemented over their components.

Streaming Applications: Recent efforts have studied the problem of resource allocation in distributed stream processing environments [13, 15, 28]. Most optimal service composition is accomplished in [13], 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 [3] we have focused on the task scheduling algorithm, while in [4] we have described a decentralized media streaming and transcoding architecture. In [21], we considered re-using components in order to improve overall 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 [12] and PROMISE [14]. 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.

6 Conclusions

In this paper 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 processing technique on the PlanetLab testbed. Our experimental results demonstrate the efficiency, scalability and performance of our approach. For our future work, we intend to study the performance of our approach under multiple resource constraints.

References

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

[2] S. Chandrasekaran, O. Cooper, A. Deshpande, M. F. andJ.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, Asilomar, CA, Jan. 2003.

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

[4] F. Chen, T. Repantis, and V. Kalogeraki. Coordinated media streaming and transcoding in peer-to-peer systems. In *IPDPS*, Denver, CO, Apr. 2005.

[5] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR*, Asilomar, CA, Jan. 2003.

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

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

[8] FreePastry. http://freepastry.org/FreePastry, 2006.

[9] S. Gerard, J.-P. Babau, and J. Champeau. *Mode Driven Engineering for Distributed Real-time Embedded Systems*. Hermes, 2005.

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

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

[12] X. Gu and K. Nahrstedt. Distributed multimedia service composition with statistical QoS assurances. *IEEE Transactions on Multimedia*, 2005.

[13] X. Gu, P. S. Yu, and K. Nahrstedt. Optimal component composition for scalable stream processing. In *ICDCS*, Columbus, OH, June 2005.

[14] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava. PROMISE: Peer-to-peer media streaming using CollectCast. In *Multimedia*, Berkeley, CA, Nov. 2003.

[15] V. Kumar, B. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, Columbus, OH, June 2005.

[16] S. Madden and J. Gehrke. Query processing in sensor networks. *IEEE Pervasive Computing*, 3(1), Mar. 2004.

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

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

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

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

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

[22] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[23] P. K. Sharma, J. P. Loyall, T. H. 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, Oct. 2004.

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

[25] N. Tatbul, U. cCetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, Berlin, Germany, 2003.

[26] S. Wang, J. Merrick, and K. Shin. Component allocation with multiple resource constraints for large embedded real-time software design. In *RTAS*, Toronto, Canada, May 2004.

[27] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao. Real-time component-based systems. In *RTAS*, San Francisco, CA, Mar. 2005.

[28] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the Borealis stream processor. In *ICDE*, pages 791–802, Tokyo, Japan, Apr. 2005.

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